FINAL YEAR PROJECT 2021

IMPERIAL COLLEGE LONDON

DEPARTMENT OF ELECTRICAL AND ELECTRONIC ENGINEERING

Mixed Traffic Simulation for Autonomous Systems in Shared Spaces

Student:
Frederik Tobias Hillier

CID:
01412520

Project supervisors:
Prof Yiannis K. Demiris
Dr Xingchen Zhang

Second Marker:
Prof Deniz Gunduz

MEng Electonic and Information Engineering

June 16, 2021

Final Report Plagiarism Statement

I affirm that I have submitted, or will submit, an electronic copy of my final year project report to the provided EEE link.

I affirm that I have provided explicit references for all the material in my Final Report that is not authored by me, but is represented as my own work.

Abstract

Autonomous systems are becoming ever more common in modern society, prompting the creation of traffic simulators to help train and test these systems in a safe environment. This report covers the analysis of existing traffic simulators, and the design and implementation of a simulator to incorporate mixed traffic simulation. The proposed solution also looks at different ways to create custom environments of real-world locations. The project consolidates a variety of features such as autonomous vehicles, pedestrians and a large variety of external APIs. The project differs from existing mixed traffic simulators due to its simple, versatile design. The solution extends the open-source simulator called AirSim created by Microsoft. The project also looks at using reinforcement learning by applying ML-Agents from Unity to create autonomous vehicles for the simulator.

Acknowledgements

I would first and foremost like to thank Professor Yiannis Demiris, for supporting me and allowing me the freedom to explore what interests me and extend the project in any direction.

Additionally, I would like to thank Dr Xingchen Zhang, for the outstanding support and advice he provided throughout this project.

I would also like to thank my parents for their support throughout university.

Finally, I would like to thank the family for having me for weekend visits and for good times at Winchelsea Beach.

Contents

List of Figures

List of Tables

Acronyms

API	Application Programming Interface
AR	Augmented Reality
BC	Behavior Cloning
CLI	Command-Line Interface
CPU	Central Processing Unit
CV	Computer Vision
DT	Digital Twin
GAIL	Generative Adversarial Imitation Learning
GPU	Graphics Processing Unit
GUI	Graphical User Interface
IP	Internet Protocol
ML	Machine Learning
NPC	Non-Playable Character
OS	Operating System
PPO	Proximal Policy Optimization
RL	Reinforcement Learning
ROS	Robot Operating System
RPC	Remote Procedure Call
SAC	Soft Actor-Critic
SDK	Software Development Kit
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
UMA	Unity Multipurpose Avatar
VR	Virtual Reality

Chapter 1

Introduction

1.1 Context

For the last decade there has been a lot of talk about self-driving cars and other autonomous systems [1]. Back in 2010 most autonomous systems were only driven on closed circuits, but today there are a variety of car manufacturers offering autopilot on their cars. In 2021 Tesla have announced that all new cars they produce will have the hardware needed for full automation in almost all circumstances [2].

Compared to 2010 when the autonomous cars were mainly driven on closed tracks, the modern-day self-driving car will have to operate alongside other cars on the road as well as alongside pedestrians. This is where the concept of shared spaces comes in. The autonomous system must now take other entities into account when making decisions.

As other entities' actions can be complex to forecast, machine learning models can be used to predict the outcome of a situation more easily. These models can be trained much faster on a simulator than in real life. This is because the simulator can simulate many different configurations at once. An autonomous system in real life cannot replicate the exact same environment, whilst this can easily be done in a simulation.

Another advantage of using a simulator is that you can learn from doing mistakes, whilst in real life, mistakes can be both dangerous and costly. In the simulator, the autonomous system can learn without the risk of colliding with an actual person or damaging other cars. Running simulations is therefore a good way to train the machine learning model much faster and safer than on a real road.

The autonomous system does not have to be a vehicle, but anything that can artificial control itself. These could for example be mobile robots/wheelchairs or artificial controlled humans and cyclists.

This project will consist of three main parts. The first one being to research available simulators (Section 2.3). This will be necessary to get a good starting point for the project. Working on an existing simulator that is too complex would mean that it would take too long to get basic features working. Whilst working on a simulator that does not have enough features will mean that too much time will be spent implementing something that another

simulator will already have implemented. Exactly what these features are will be clarified in the Requirements Capture chapter (Chapter 3). The second part will be to implement the missing features from the simulator that we selected to work on. And the last part will be to try to simulate the mixed traffic interaction (Section 5.2).

The product generated from this project could be used for a variety of different use cases. Firstly it could be used to test the software for an autonomous system. Testing autonomous systems in a simulator would be much faster than trying to check for corner cases in real life. Another example would be to train a machine learning model. The simulator will aim to be a diverse tool to accommodate a variety of needs.

1.2 Project Objectives

The objectives of the project are:

- Survey the current state of the art in vehicle and mobile robot simulation and evaluate current simulators (e.g. CARLA, CrowdSim3D, Gazebo), list and understand the underlying simulation methodologies (e.g. types of physics engine), and understand and document their advantages and disadvantages, particularly in terms of their sensing and control abilities.
- Select the most suitable current simulator that offers potential for incorporating mixed traffic (for example, cars, mobile robots/wheelchairs, humans, cyclists, and so on), and 3D environments/maps.
- Create a process for incorporating and controlling agent models with an API (e.g. ability to add control abilities to the human model, for example to move their bodies, and to grasp objects).
- Add sensors to the simulated agents (e.g. cameras, LIDARS) and a perception API, where simulation users can obtain sensing data from the simulator, for example what can the agents "see" in their environment.

1.3 Report Outline

This report aims to introduce the project, show the necessary background research and explain in detail the design decisions and work done to achieve the desired result. This report consists of 9 chapters with 4 appendices.

- This **Introduction** describes the context of the underlying problem the project will try to solve as well as the project outline.
- The **Background** chapter will look into the motivation for doing this project. The chapter will also introduce the technical tools and theory needed to understand the project. This will include looking at what a game engine is, as well as briefly looking at the two major game engines used by most simulators. Thereafter comes an analysis of the simulators that have been studied with a conclusion for which simulator was chosen for this project.
- The **Requirements Capture** chapter will cover the requirements and deliverables expected from this project.

- The **Analysis and Design** chapter will provide a functional overview of AirSim as well as an explanation of the architectural design and limitations of the project.
- The **Implementation** chapter will study the three main components of the project. The first part will explain which features were added to the chosen simulator AirSim and how. The second part will look at how Unity's ML-Agents was used to implement Reinforcement Learning. The third part will examine how to create maps and environments for the simulator.
- The **Testing and Results** chapter will look at how the simulator was tested and briefly what results from the autonomous vehicles using ML-Agents produce.
- The **Evaluation** chapter will compare the outcome of the project against the requirements specified in the Requirements Capture chapter.
- The **Conclusion** chapter will present a short summary of the project outcome and challenges faced.
- The **Future Work** chapter looks at ways of extending and improving AirSim as well as future use cases for the simulator.
- The **Appendix** consists of 4 chapters where chapter B and D are the most important. Chapter B covers the indepth analysis of all the simulators researched. Chapter D is the user guide explaining in slightly more detail how to build and run the tools and programs mentioned.

Chapter 2

Background

2.1 Motivation

This project will look at creating a simulator that is flexible and diverse to incorporate mixed traffic simulation. Mixed traffic simulation will be a simulation that can simulate the interaction between agents which are not all the same, for example, pedestrians, vehicles and other autonomous systems. As will be looked at in Section 2.3, there are no easy to use, open-source simulators that allow users full flexibility to customise their environments, APIs and physics.

The project looks to create an open environment where users can simulate mixed traffic. The project is first and foremost aimed at researchers who need a platform to test their autonomous system.

There are three alternatives for how to develop the simulator. The first option is to use an existing simulator as a tool and then create a mixed traffic simulation inside that environment. The next option is to modify and extend an existing simulator. The last option is to begin from scratch.

After analysis of existing simulators, the project will look to extend AirSim. After trying to use Unreal Engine as the game engine, it was decided that Unity would allow for more flexibility.

Carla would probably be the simulator that resembles this project the most. This is because it has a lot of features such as pedestrians and autonomous vehicles which all can be controlled through APIs. The main advantage of this project over Carla is the flexibility of the platform. To load new maps into Carla is challenging as the traffic rules have to be embedded in the map. Carla is also a stand-alone project rather than a plug-in like AirSim making adding new features harder.

2.2 Background Theory

This section will cover the background theory needed to understand the project.

2.2.1 Physics Engines

The purpose of a physics engine is to allow programs to easily implement physics, without the creators having to implement their own from scratch every time. An example could be, if a program needed to calculate a trajectory, this formula could be implemented directly inside the physics engine [3]. This is quite a simple example, but simulating basic particle distribution or fluid dynamics would be much more complex.

There are a large variety of different game engines, but the ones that are most commonly used in the simulators listed below are PhysX and ODE.

Open Dynamics Engine (ODE)

Open Dynamics Engine is a free physics engine most commonly used in simulation projects [4]. ODE is written in C++ and is a popular choice for simulating robotics.

PhysX

PhysX is an open source-physics engine developed by Nvidia. This physics engine is commonly used in a lot of computer games. Both Unity and Unreal Engine use PhysX (Section 2.2.2). PhysX is quite unique among other physics engines. This is because most scientific-targeted simulation software would be more accurate. This physics engine however truncates the calculations leading to more efficient but less accurate simulations [5]. The physics engine is also non-deterministic, meaning there could be some variation between runs.

2.2.2 Game Engines

The purpose of a game engine is to work as a fundamental back-end for most video games and simulators. A game engine is a software tool that allows interactive digital content to be made using a framework that easily allows the user to run it on different platforms such as computers, smartphones and consoles [6]. Game engines are complex and consist of many components such as a physics engine, a rendering engine, and other interactive tools to help the creator. They give the creator the freedom to control everything from lighting and audio to animation and character behavior in their game.

There exists many different game engines, but the two main ones that are currently in existence are Unity and Unreal Engine. These allow beginners to easily create their first games, as well as professional companies creating their games for millions of users [7, 8].

Unreal Engine

Unreal Engine is a fast-growing game engine created by Epic Games [9]. The first version of Unreal Engine was released in 1998, but Unreal Engine 4 (UE4), which was released in September 2014, is their latest game engine. They have however announced that they will release Unreal Engine 5 (UE5) at the end of 2021 [10].

UE4 has a variety of different use cases as well as for games. TV broadcasters such as Sky Sports and BBC are using Unreal Engine for their graphical match analysis [11]. Professional architects are also using the product to illustrate their creations [12].

Unreal Engine 4 is free for personal use and businesses making less than $1 million in gross revenue [10]. The code for UE4 is available on GitHub[1] (For access to the repository you would need to sign up for an Epic Games account[2]). UE4 allows two different ways for developers to program their games. Either by using C++, which links with Visual Studio[3], or by using blueprints as illustrated in Figure A.1 in the appendix. Blueprints allow the creator most of the freedom code gives them, but in an easier drag and drop layout.

The majority of the simulators that will be looked at later use Unreal Engine as their game engine.

Unity

Unity is currently the most commonly used game engine with over 2.5 million registered developers [13]. Unity is developed by Unity Technologies and was first released in June 2005. Over 60% of augmented- and virtual reality games are created using Unity [6].

Unity is free for personal and educational projects. To create the game, Unity allows a lot of features to be used through the GUI, but for full freedom, the creator would need to program in C#. Unity provides lots of resources to learn C#. Unity also combines with Visual Studio where the IntelliSense can help with recommending functions as well as debugging[4].

What follows is a list of some of the terms used in this report to describe features in Unity[14]:

- **Scene** - In Unity several scenes can exist. This is the area where the environment is built. A scene usually consists of a camera, a light source and whatever the user wants in the environment.
- **Asset** - Assets is an item that can be used in the simulator. This could be an image, a 3D model or an audio file.
- **GameObject** - This is the funcamental object in a scene. A game object can represent a character, a building, a camera or something else. The GameObjects functionality is decided by the components attached to it.
- **Prefab** - This is a saved collection of GameObejcts. An example of this would be the vehicles which consists of different game objects such as the body and wheels.
- **Asset Store** - This is a website run by Unity where users can download both free and paid assets.
- **ML-Agents** - This is a machine learning framework that can be added to Unity which will help train the models reinforcement learning using PyTorch.

[1] https://github.com/EpicGames/UnrealEngine

[2] https://www.unrealengine.com/en-US/ue4-on-github

[3] https://docs.unrealengine.com/en-US/ProductionPipelines/DevelopmentSetup/VisualStudioSetup/index.html

[4] https://docs.microsoft.com/en-us/visualstudio/gamedev/unity/get-started/getting-started-with-visual-studio-tools-for-unity

- **Tick Speed** - This is how fast the simulator is running. A tick is an update in the simulation. This is measured at frames per second. The simulator usually runs at about 120 ticks per second, but more computational intensive tasks can decrease this. The tick speed can also be increased when training machine learning models to speed up learning, or set to 0 to pause the simulator.

2.2.3 Blender

Blender is an open-source tool that allows creators to render, model and animate objects [15]. Blender is currently one of the most popular free modelling tools available [16]. There are also lots of available plugins for Blender which allow to further extend the software. For this project, Blender will mainly be used to create the environments. This is because it has tools that can optimise the polygons which is how the models are constructed. By decreasing the number of polygons it becomes increasingly easy to compute for the game engine. Blender also gives the user the full freedom to customise the models which cannot be done in the game engine.

2.2.4 Levels of Automation

When talking about autonomous systems, it is important to know the different levels of autonomy. The modern day Tesla cars are level 3, which means they can be driven autonomously for the majority of the time, but do need a human driver as a fallback system [2, 17].

Level	Name	Description
0	No Automation	The driver is solely controlling the vehicle
1	Driver Assistance	Cruise control is an example of this
2	Partial Automation	Automatic parking
3	Conditional Automation	Modern day Tesla cars
4	High Automation	Vehicle almost fully automated
5	Full Automation	Vehicle never needs human input

Table 2.1: The table shows the various levels of autonomous systems.
`https://link.springer.com/article/10.1007/s40534-016-0117-3#Sec3`

For this project all our simulations will be level 5.

2.2.5 Mixed Traffic Simulation

The purpose of a mixed traffic simulation is to model the interaction between different types of agents, for example the the interaction between pedestrians, autonomous robots and vehicles. This is important because having a fully automated agent has to interact with a lot more than only other fully automated agents [18].

A mixed traffic simulator can be used to simulate different situations where an autonomous system has to react to other agents behavior. This will mean the implemented algorithm will have the opportunity to try out a lot more complex scenarios a lot faster than in real life. This will help speed up development time.

2.2.6 API

API stands for Application Programming Interface. This will allow a program to communicate with one-another. The API will define what kind of requests another program can make, how to make them, and the what the format should be. APIs also need to be documented so that other developers know how they work [19].

In the case of our simulators, APIs will be used to communicate between an external program and the simulator itself. These communications could contain information for how to control the agent, or it could contain information from what the agent observes.

2.2.7 Digital Twin

A digital twin is a term used to represent a real-time digital model of a physical system [20]. The model will continually adapt to changes in the environment using real physical sensors. There are several advantages of a digital twin. Firstly it can be used to predict what is going to happen to the physical system. Secondly, it can act as a testing environment without damaging the real-life object. What follows is a list of some of the terms used in this report to describe features in Unity

2.3 Analysis of Existing Simulators

In this section we will look at a large variety of different simulators, to determine which one best suits our purpose. We will be looking at which operating system and game engine the simulator uses, whether or not it is open source, and the pros and cons of each simulator. We will be particularly looking at the simulators sensing abilities, ability to add additional entities, map customisability, available APIs, and how user-friendly the simulator is. Aspects of the simulator which is not as important as how realistic the simulator physics is, and how visually good looking it is. These are criteria formed by the project specification (Section 1.2)

The purpose of this section is to get a good understanding of the different simulators currently in existence.

2.3.1 Overview

The table below contains an overview of the simulators explored for this project. More detailed information about each simulator can be found in the appendix Section B. The list consists of different kind of simulators. Robotics simulators are designed to accurately simulate the physics of moving parts and objects, soft body simulations are designed to simulate what happens to the physical material properties of an object when it for example collides. And traffic simulators, which are designed to simulate one or several vehicles with sensors autonomously driving around.

[5]"Any" means most commonly used Linux distributions, Windows and MAC
[6]"Actively Developed" means that there were several releases in the past year
[7]Reasoning for the decision can be found in the Appendix B
[8]Multiple agents is possible, but in a very limited number and has to be declared at the start.
[9]No sensing APIs
[10]Linux is the main platform and they are aiming to support Windows as well. Currently Carla does not work on Windows.
[11]More information can be found here: `https://www.coppeliarobotics.com/helpFiles/en/licensing.htm`
[12]`https://forum.coppeliarobotics.com/`
[13]`https://groups.google.com/d/forum/lpzrobots`
[14]`https://forum.rigsofrods.org/`
[15]`https://sourceforge.net/p/torcs/discussion/11281/`
[16]Only sensing APIs

Simulator	Open Source	OS[5]	Game Engine	Development[6]	Support	Pedestrians	Extensibility	Multiple Agents	Existing APIs	Worth Considering[7]
4DV-Sim	No	Linux	PhysX (Physics Engine)	Actively Developed	Company offers support	Yes	No	Yes	Yes	No
AirSim	Yes	Any	Primarily UE4, but also Unity	Actively Developed	GitHub Issues	No	Yes	Yes[8]	Yes	Yes
Apollo	Yes	Docker	Unity	Actively Developed	GitHub Issues	No	Difficult	No	No	No
Autoware	Yes	ROS	N/A	Actively Developed	GitLab Issues	No	Difficult	No	Limited[9]	No
Carla	Yes	Linux[10]	UE4	Actively Developed	GitHub Issues	Yes	Yes	Yes	Yes	Yes
CoppeliaSim	Yes[11]	Any	Several different physics engines	Actively Developed	Forum[12]	Yes	Difficult	Yes	Yes	No
CrowdSim3D	No	Any	N/A	Unknown	Company offers support	Yes	No	Yes	Yes	No
Deep Drive	Yes	Any	UE4	Last Commit June 2020	GitHub Issues	No	Yes	Yes	No	No
Donkey Car Simulator	Yes	Any	Unity	Actively Developed	GitHub Issues	No	Yes	No	Limited[9]	No
Gazebo	Yes	Any	Several different physics engines	Actively Developed	GitHub Issues	Yes	Yes	Yes	Yes	Yes
LPZRobots	Yes	Linux	ODE (Physics Engine)	Last Commit November 2018	Google Group[13]	Yes	Yes	Yes	No	No
LGSVL Simulator	Yes	Windows 10	Several, both Unity and UE4	Actively Developed	GitHub Issues	Yes	Difficult	Yes	Yes	No
Marilou	No	Linux and, Windows	N/A	Latest release was 2018	Non	Yes	No	Yes	No	No
rFpro	No	Windows	ISIMotor	Actively Developed	Company offers support	No	No	Yes	No	No
Rig of Rods	Yes	Linux and, Windows	Creates its own soft-body physics engine	Actively Developed	Forum[14]	No	Yes	Yes	No	No
TORCS	Yes	Linux and, Windows	Non, implemented from scratch	Latest release was 2016	Discussion page[15]	No	Yes	Yes	No	No
Webots	Yes	Any	ODE (Physics Engine)	Actively Developed	GitHub Issues	Yes	Yes	Yes	Limited[16]	Yes

Table 2.2: The table contains a brief overview over some of the simulators researched. For more detailed information on the simulators see Appendix B.

2.3.2 Further Simulator Analysis

The simulators this section will look closer at are **AirSim**, **Carla**, **Gazebo** and **Webots**. These are all open-source simulators that hopefully can be extended to suit the project requirements.

As AirSim, Carla and Webots have release builds available these were tried first.

AirSim was easy to use and has many available APIs. The simulator is also built using either Unreal Engine or Unity, and it allows for several different languages to interact with the APIs. There are several benefits of using Airsim. Firstly, there is a lot of documentation that helps to understand how the code is structured and what the simulator is capable of doing. The simulator also allows for drones if that could become necessary later on. The simulator is also actively developed, and questions are frequently answered on GitHub. The main drawback is that currently the simulator is only designed to handle one agent. This would be something that has to be fixed to allow for multiple agents. Also, there are no pedestrians in the simulator. However, as the simulator works using a game engine, these things should be possible to add.

Carla is very straightforward to get started with. The executable launches the environment, and then you can use scripts to add vehicles and control the simulation. Users can also drive around in the simulator by launching a new instance of Clara which will interact with the environment. Carla has a lot of APIs and a lot of existing features. It has pedestrians that can be controlled through the APIs as well as cars already driving autonomously around. The disadvantage with Carla is the ability to import custom maps [21]. Carla requires the map to consist of two layers. The first one being the map structure with buildings and roads, whilst the second one will consist of road rules, such as traffic lights, where the cars are allowed to drive, pedestrian crossings, and so on. RoadRunner[17] can be used to import maps, but this is not a free product.

WeBots was not as easy to use as the other two. Unlike AirSim and Carla, WeBots is a robotics simulator rather than a traffic simulator. This meant that when the simulator focuses more on physics rather than driving logic. WeBots was also not as intuitive to use as the other simulators. When compiling the source code for WeBots, several dependencies are required. Overall, WeBots was not as good as the other two and it would not be worth considering further. Especially as Gazebo is also an open area robotics simulator.

The next step was to try compiling the source code for AirSim, Carla and Gazebo. Section D.2 in the UserGuide explains how to compile the three simulators. One of the main challenges faced was trying to set up Unreal Engine on Ubuntu. This required upgrading the graphics driver, but eventually concluded that the graphics card could not handle the required driver version. Unreal Engine worked on Windows, but Carla would not compile on Windows. As there was no way of running Carla easily it was decided not to proceed with it.

The following points were key when comparing **AirSim** and **Gazebo** to see which one should be used:

[17] `https://uk.mathworks.com/products/roadrunner.html`

- AirSim with Unreal Engine is more feature-rich than Gazebo. Using Unreal Engine will also minimise the chances of finding out something is not possible later on.
- Both have the ability to import maps easily
- Both have the ability to train machine learning models.
- Gazebo has more sensing APIs such as GPS and Magnetometer sensors[18].
- Neither simulator is designed for having multiple entities. However, it seems like it will be easier to add this ability to AirSim than Gazebo.
- AirSim is designed for vehicles and traffic, whilst Gazebo is created as a more generic robotics platform.

Overall, AirSim was decided to be the better simulator to try to extend first. As mentioned above, an important requirement is to allow for a flexible simulator. AirSim works as a plugin for either Unreal Engine or Unity. This means that it gives the option to extend the simulator to anything the game engine allows.

2.3.3 Conclusion

AirSim was chosen to be the simulator this project would build on for several reasons. Firstly, the existing code base is a good starting point. It is not as complex as some of the other simulators, whilst at the same time having a lot of basic features. Secondly, the simulator is very flexible and versatile. As it is a plugin for Unity and Unreal Engine, it gives the option to use all the features those game engines have to offer. Another advantage of using AirSim is that as it is using a game engine, the chances that there is an issue later on which cannot be resolved is small. If a platform like Gazebo was used, adding a missing feature to the simulator could be a very complex task as there is no GUI for the program itself.

[18] `https://osrf-distributions.s3.amazonaws.com/gazebo/api/dev/classgazebo_1_1sensors_1_1Sensor.html`

Chapter 3

Requirements Capture

The objective of this project is to create a simulation environment where mixed forms of traffic can interact with one another. The simulator should be able to be flexible in its use cases and have a large variety of features. It is also very important that the simulator is easy and intuitive to use.

The main goal is to have a good base platform on which users can conduct experiments. The users should have the flexibility of adding custom models and environments to the simulator. These steps must be straightforward. The project should also demonstrate how this can be done by looking at several alternatives.

The system should also contain some machine learning models of mixed traffic interaction. This is used both to illustrate how this can be achieved, but also to show the complexity of the system.

3.1 Simulator Requirements

The project should look at existing simulators and **compare** them against the following criteria.

- **Usability:** The simulator should be simple to set up and use.
- **Extensibility:** The simulator must be extendable. This means adding new features and modifying existing features should be possible.
- **OS:** The simulator should run on Ubuntu.
- **Game Engine:** The game engine is the core component of a simulator. Using an outdated game engine could make future work challenging. Simulators with no game engines could be hard to understand and debug.
- **Development:** The simulators would ideally be actively developed. This minimises the chances of running into issues later on with outdated libraries.
- **Support:** If issues cannot be resolved having community support could be vital.

The simulator should have the **following features**:

- **Multiple Agents:** Having the simulator handle multiple agents is crucial to perform mixed traffic simulation. It does not however need to handle large crowds.
- **Variety of Agents:** Adding new models and behaviour to the simulator should be a simple process. This should include pedestrians.
- **Free movement:** The agents should be able to move freely around the environment. This means that they should not be restricted using any external rules.
- **Customisable environments:** It should be straightforward to change the simulation environment.
- **Controlling APIs:** It should be possible to control the agents from external programs. The control should include simple navigation, but also the ability to add and delete entities.
- **Perception APIs:** These could include cameras or LIDARS. Perception APIs would be APIs that could give the user the ability to obtain an understanding of the agents' surrounding.

3.2 Environment requirements

This section will list the requirements decided for the environment:

- **Real location:** It would be preferable to be able to replicate real locations.
- **Outdoors:** The environment should primarily be outdoors.
- **Simplicity:** To create the environment should not be a challenging task. Importing the model into the simulator is down to the simulator itself, but the environment should be of a format that makes this easy.
- **Customisable:** The process of creating the maps should allow the user to modify the maps if needed.

3.3 Deliverables

The main deliverables for this project will be:

- An analysis of available simulators with a conclusion for which one would be best to use for the project.
- A flexible simulator that has the ability to easily add additional models and simulate mixed traffic.
- A simulator with the incorporated sensing and controlling APIs.
- Scripts to demonstrate the features available to the simulator.
- Documentation explaining how to use the APIs.

Chapter 4

Analysis and Design

This chapter provides a high-level overview of the core components of the AirSim architecture. The chapter will look at how the simulator has been adapted for this project, the design decisions made, and the limitations.

As the project is extending an existing code-base, the design decisions aimed to limit the impact on the existing structure. This was to allow future updates to the master project, to benefit this one as well. Most of the changes to this project were made in Unity, but changes were also made in AirLib and the wrapper to allow for additional APIs.

4.1 AirSim Functional Overview

Figure 4.1 shows a simplified overview of the important components in the original AirSim architecture. It is key to understand each of these components, as they are all updated throughout the project.

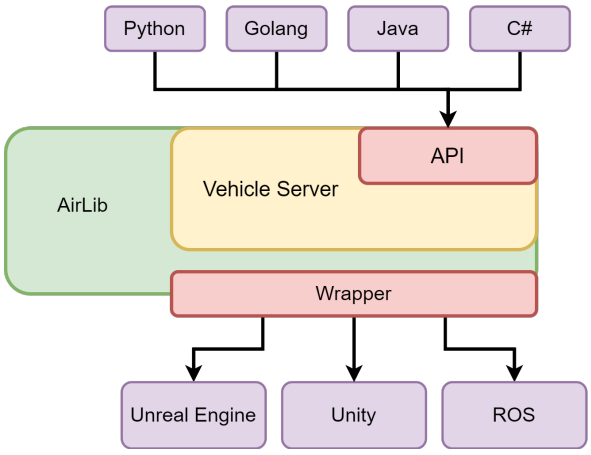

Figure 4.1: High-level overview of the core components of AirSim used for this project. Only one server exists and all API calls are passed to it.

The project consists of 4 main components, Unity, which is written in C# and contains most of the simulator logic, AirLib which is written in C++ and contains the server, the AirLib Wrapper which is written in C++ and acts as a bridge between AirLib and Unity, then finally the code where the user can interact with AirLib through the APIs. As this project is very

modular, it gives the user the freedom and flexibility to choose a game engine and input language.

This section will contain a brief overview of each of the 4 components. This is to better understand the design decisions made in Section 4.2. Figure 4.2 shows an example of how the user interaction layer can interact with Unity. Most APIs follow this layout.

4.1.1 AirLib

AirLib is the main component of AirSim and where the majority of the code is located. This self-contained library consists of four main components. The first one is a physics engine. This component is lightweight and designed to be simple to add new vehicles and drones. The next component is a sensor model. This component contains header-only models for external sensors such as GPS and Barometer. The third component is the vehicle module. Currently, the only model implemented is for PX4 QuadRotor[1], which is a platform that can be used to control drones. The last component is the control library. This part provides abstract classes for the APIs and implementations for specific platforms.

This project does not use the physics engine, sensors module and vehicle modules in AirLib. The project is instead using the Unity version of these components. Instead, this project focuses almost solely on extending the control library. On startup, AirLib creates AirLib creates a server using RPCLib[2] which can interact with the APIs over a TCP channel. The advantage of using RPC is that it allows for a range of different programs as well as being fast and lossless (Section 4.1.4). The design decision to move the physics engine and sensors to Unity is further discussed in Section 4.2.

4.1.2 AirLib Wrapper

A wrapper is used for AirLib to communicate with Unity. Using a wrapper allows for a variety of different game engines as well as other physical systems such as ROS. This makes AirSim more modular, which again can increase the extensibility of the program. The wrapper will implement the interfaces declared in the vehicle module (Section 4.1.1). At startup, Unity will pass function pointers to the wrapper so that later the APIs can call those functions. The wrapper will also create an instance of AirLib which will set up the RPC server. The wrapper also handles the conversion of the data from C# to C++.

Figure 4.3 shows an example of how AirLib links to the Unity Wrapper. AirLib has an interface for the available APIs that are then overridden in the wrapper. These overridden member functions will then access the function pointers initialised at startup by Unity. This is done by having Unity call the InitServerManager with the arguments being the function pointers. As was explained in Section 4.1.1, the API interacts with the server initialised at startup. In this case, the server is stored in the SimulatorServer class and will call functions overridden from the ServerSimApiBase class. Unity can also make calls to the wrapper at runtime. These are done through the UnityToAirSimCalls. The figure shows a slightly simplified for how Unity can start and stop the server. To create the server, the wrapper creates a new instance of the ApiServerBase and initialises it with a pointer to the ServerSimApi. It is clear from this figure that another game engine can easily be used by changing the overridden functions.

[1] `https://docs.px4.io/master/en/getting_started/`

[2] `https://github.com/rpclib/rpclib`

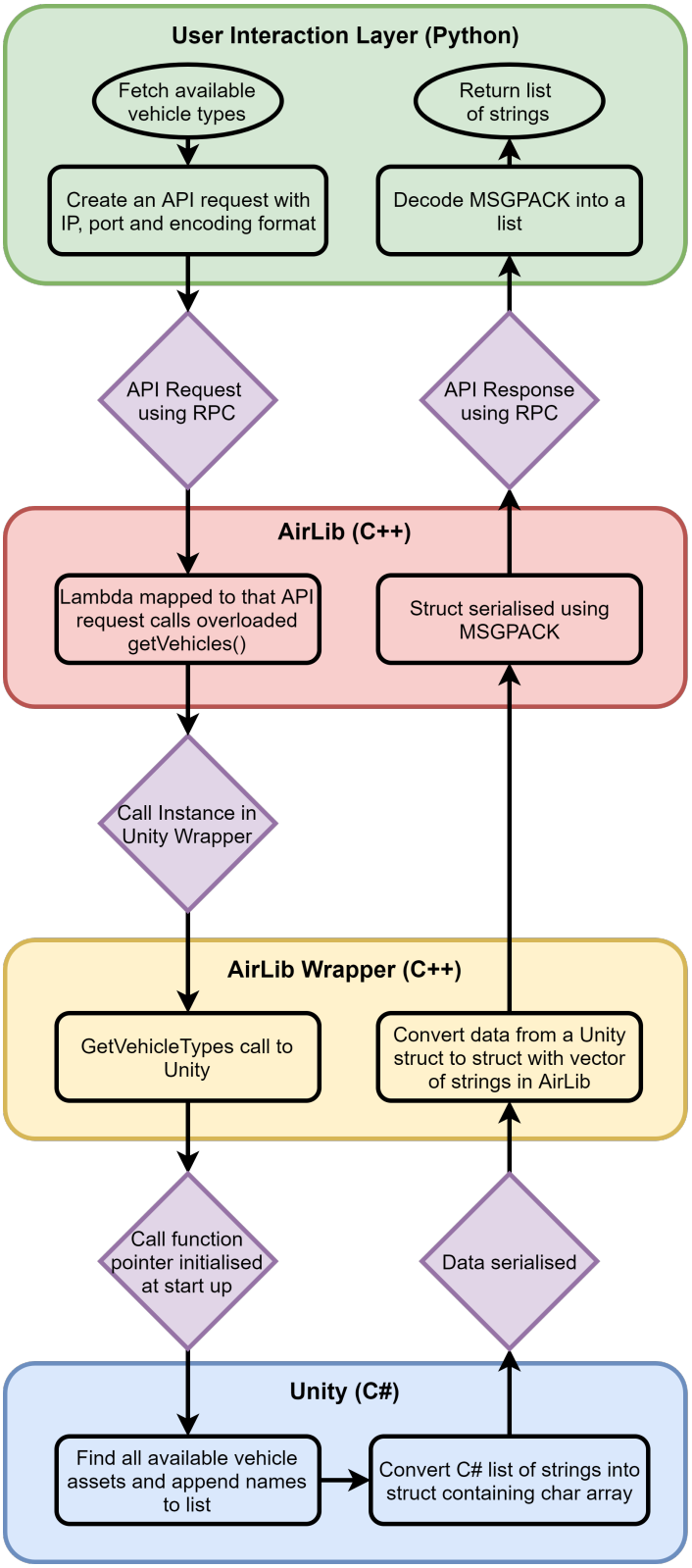

Figure 4.2: High-level overview of the API that fetches available vehicle types. For APIs that require arguments, these will be encoded and added to the API request. The main change between different API requests is what happens in Unity.

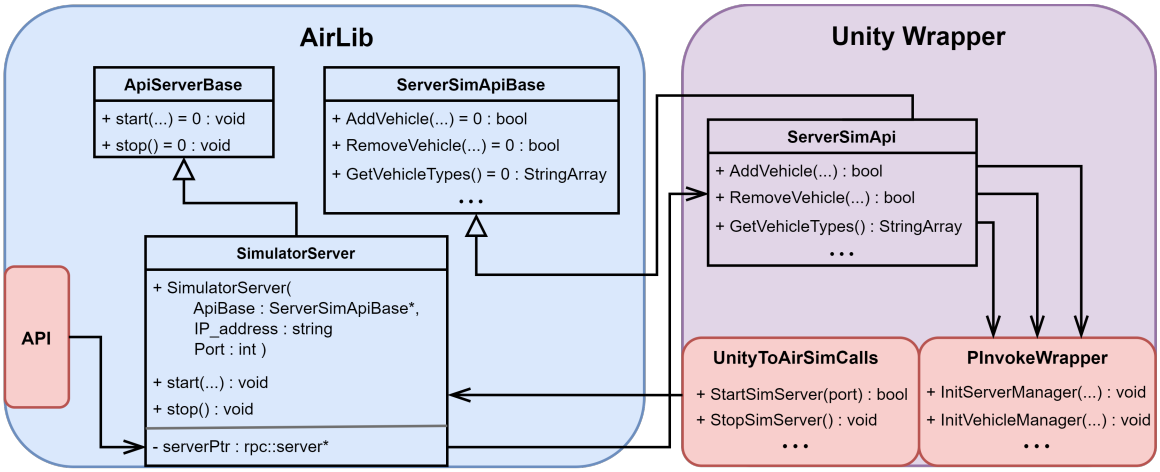

Figure 4.3: A simplified view of how AirLib and the Unity Wrapper link together. UnityToAirSimCalls and PInvokeWrapper are red boxes to illustrate that they are only files containing functions. The other objects are classes. The black arrows show examples of functions calls. The details of how Unity starts the server is left out in this diagram.

4.1.3 AirSim with Unity

As will be discussed in Section 4.2.1, Unity was chosen as the game engine for this project. Unity works as the physics engine and all of the simulator logic is solely done in Unity. One of the disadvantages of using Unity is that it is not thread-safe. This means that the AirSim plugin has to interact with the simulation through shared memory. This will be explained further in the implementation section (Section 5.1). All of the simulator logic is solely done in Unity.

AirSim works as a plugin for Unity. AirLib and the wrapper are compiled into a single DLL file which can be interacted with through Unity. This allows the user to use all the features in Unity when creating maps and environments. This also makes it easy to set up an AirSim environment as only the DLL and a setup object has to be added to the scene.

4.1.4 User Interaction Layer

AirSim allows for a variety of different languages to interact with AirLib. This is because the API calls use MessagePack[3] also known as MsgPack. MsgPack allows for an efficient binary serialisation[4]. MessagePack supports over 50 programming languages and environments. These include C#, Golang, Haskell and others. For this project, only Python will be used. This is because Python is a flexible language that makes prototyping quick and easy. The existing code for the user interaction layer is already written in Python, so it will be more beneficial to continue with this rather than starting from scratch.

In the Python project, three classes are implementing the APIs for each of the servers. This is to make it clear which type of entity the user is communicating with. It also avoids confusion when passing controls to the simulator, as pedestrian and vehicles use different controls. Python allows for external tools like OpenCV to process the images passed from the Simulator. OpenCV can generate a diffusion map using two images created by forward-

[3] `https://msgpack.org/`
[4] `https://github.com/msgpack/msgpack/blob/master/spec.md`

facing cameras on a car. The diffusion map highlights objects that are shifted more between the two images, i.e. the object has to be closer.

4.2 Architectural Design and Limitations

This section will look at some of the architectural decisions made when modifying the existing AirSim design. The section will also look at what limitations were introduced by these decisions. Section 5.1 will go into more detail about how the design decisions were implemented.

4.2.1 Unreal Engine or Unity

After AirSim was chosen as the simulator, the main decision to be made was which game engine to use. AirSim primarily uses Unreal Engine, but there is also a prototype version using Unity. As mentioned in the background section (Section 2.2.2), Unreal Engine has better graphics than Unity. Using Unreal Engine would also mean that the simulator more complete with many more features available such as dynamic weather, more sensors and more APIs. Generally, Unreal Engine also has better performance than Unity[22]. The disadvantage of using Unreal Engine is that it makes it harder to rapidly prototype and add new features. This is because the existing code is much more complex. It is also because Unreal Engine is not as simple to use as Unity. In Unity, the simulator can run in debug mode making it possible to update scripts whilst the program is running. This is not possible in Unreal Engine where a small change can take a couple of minutes to compile.

There are several advantages to using Unity over Unreal Engine. Firstly, Unity's asset store provides a large range of plugins and models that can be used. The key ones that are of interest for this project are the UMA models (Unity Multipurpose Avatar), which can be added to model pedestrians, and the ML-Agents tool kit, which can be used to train reinforcement learning models. Both of these addons will be explained further in the implementation section (Section 5.1). The second advantage of using Unity is the simplicity of having scripts. This makes debugging simpler as the scripts can be interacted with at runtime. The scripts can also be attached to objects which makes it easy to create several instances of an entity. Another advantage is that as using Unity as the game engine is currently only an experimental version, only the most important features have been implemented. This makes the existing code easier to navigate than the code written for Unreal Engine. This can also be seen as a disadvantage of using Unity. As only a few APIs have been implemented, a lot of work has to be put in to have the same features available in Unity as currently are available in Unreal Engine[5].

Overall using Unity for this project would be more beneficial than using Unreal Engine. Unity allows for rapid prototyping which is more important for this project. Unity's ML-Agent also makes it easier to train reinforcement models for the part of the autonomous system of this project.

4.2.2 Moving logic to Unity

Section 4.1.1 explained how AirLib is split into four parts and that this project would only focus on the vehicle module. AirSim is designed in such a way that software components are

[5] `https://microsoft.github.io/AirSim/unity_api_support.html`

easily exchangeable. As can be seen from Figure 4.1, ROS, which is a robot operation system, could be used instead of a game engine. AirSim would then be able to model the behaviour of the real robot without needing the game engine. As this project would only focus on the simulation, this was not a feature that was needed. Instead, it was easier to move all the physics calculation to Unity along with other logic such as controlling the vehicles. Making this design decision would make it much easier when adding the pedestrians. Currently, the physics engine in Unity would need a redesign to get something like autonomous wheelchairs and pedestrians to work.

The consequence of this is that the simulation behaviour could differ between the different game engines. AirSim would also not work for pedestrians without the Unity game engine. As this project is only a prototype, the simplification of moving the logic to Unity will still be beneficial.

4.2.3 Dividing the server

The decision to split the server in AirLib was the main change to the existing design of AirSim. This design change was needed for several reasons. Firstly, how the server is implemented currently, all API calls are single-threaded, which means that one call has to be processed before the next one is handled. This makes the API that captures images slow as it has to wait for one game tick before returning the image. (See Figure 5.2 in Section 5.1). Another reason for doing this was to simplify the code structure. As can be seen from Firgure 4.4, dividing the AirLib server to handle the game state, vehicles and pedestrians separately was a natural split, especially as the pedestrians would introduce a large number of new APIs. This split would make it clear which entity is being controlled and which APIs are available. Also, in the current design, the server only exists if there is a vehicle in the scene. If the vehicle is removed, the server is closed. This also means that there has to be a vehicle in the scene to start the server. This is not the desired behaviour as a user would want to be able to spawn in and delete both vehicles and pedestrians at any time.

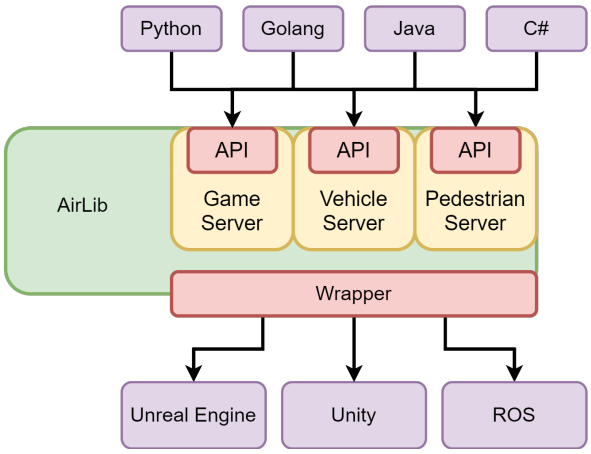

Figure 4.4: Separate APIs into several servers on different ports. This makes it easier to expand the APIs and add additional features.

The division was done by replicating a simplified version of the server creation and then moving the APIs for the game state over to this one. As mentioned in the functional overview (Section 4.1) AirLib has a lot of features that this project does not need. When replicating the server initialisation, these parts were left out. In Unity, the code was changed in such

a way that at startup, the game server will start, and the user can communicate with the server, such as printing, spawning and deleting vehicles and pedestrians, and so on. When the first vehicle is spawned the server vehicle server will start, and when the last is deleted, the server will close. This behaviour is the same for the pedestrians.

This project benefits in several ways when dividing the server. Firstly, as mentioned above, the server would now start running at startup. This means that users can control the game state without vehicles having to be in the scene. Secondly, dividing the APIs makes the code more readable. Originally it is difficult to see if an API call acts on the server or the vehicle. Another benefit of the split is that the game server can communicate instantly with the server instead of having to wait for the other calls to the vehicle server to finish. Finally, with the current implementation, it would be easy to open up a separate server for every vehicle. This could be beneficial as separate scripts could run a specific vehicle. This change is now possible as the code is designed in such a way that it can start a new server on request.

However, there are several limitations and disadvantages to this new approach. Firstly, if too many servers are launched the overhead will be too big and the performance would decrease. Currently, this is not an issue with only three servers, but with one server per entity, this could be slow. Secondly, if for example, a developer wants to use different means of communication for the APIs, rather than RPC, this would require several changes in multiple files. Finally, as this is a major change to the existing AirSim, pulling new features from the master repository will no longer be straightforward.

Overall the benefits still outweigh the disadvantages. As having multiple entities in the scene at once is a key part of this project, separating the APIs relating to the game from the APIs relating to the vehicle allows for this. This also works as a proof of concept if performance becomes an issue and a server per entity becomes necessary.

Chapter 5

Implementation

This chapter looks at what features were implemented or has been attempted throughout the project. The chapter has been split into 3 main sections. The first section will cover how the missing features were added to AirSim and how the simulator was adapted to better fit this project. The second part will cover how reinforcement learning using ML-Agents was used to create autonomous systems so that every agent would not have to be controlled at once. And the last section will cover how to create maps and environments that can be used within the simulator.

5.1 Expansion of AirSim

This section will cover which features were modified and added to AirSim to reach the desired behaviour. Each section will look at the challenges faced and discuss previous approaches. The sections will also explain which bits of existing code the feature is based off, or if the feature was written from scratch.

5.1.1 Multiple Entities

The first modification to be made to the simulator was to make sure that it could handle multiple agents. This was an essential step as the simulator could not be used if having several agents at once was not possible. Currently, the simulator is primarily designed for one agent. After initial research, the simulator should have been able to handle two vehicles if this was added to the startup configuration. However, this did not work in Unity.

The first step was to modify the existing APIs so that the vehicles could be accessed individually. To do this, all vehicles were added to a global list. Each vehicle was also given a unique identifying name. The next step was to add an argument to each API specifying the vehicle. When a vehicle API was called, Unity would first iterate over the map looking for the corresponding vehicle. Once the entity was found Unity would then forward the API request to that vehicle. This change had to be made throughout AirSim tracing the call from the user interaction in Python to Unity.

The main challenges faced when doing this was originally trying to adapt the configuration file. As this had not been properly implemented in Unity yet, time was spent trying to debug this issue. Eventually, it was discovered that adding the vehicles manually to the scene before starting would be easier.

Currently, if two vehicles are given the same name the second vehicle will spawn, but the API calls will only be redirected to the first vehicle. This can easily be changed so that either both entities should behave in the same way, or that the second entity does not spawn. This behaviour however was seen as unimportant and have been left out for the time being.

5.1.2 Spawning Entities at Runtime

Being able to spawn entities at runtime was a desired feature as it would allow the users to have full control of the simulation through the APIs. This was not an existing feature as the simulator was not designed to simulate several entities at once. As mentioned in the section above, multiple vehicles could be declared in the configuration file. This file however is only loaded at the start and never reloaded whilst the simulator is running.

The first change that had to be made to AirSim was to add the new API. The API would take in four arguments: the vehicle type, the identifying name, the spawn coordinates and the initial rotation. The first issue that had to be resolved was that Unity is not thread-safe. This means that the server could not directly interact with the simulator behaviour. As can be observed from Figure 5.1, this was resolved by creating a request which was added to a queue. The fixed update cycle would then check if this queue was empty on every game tick. This means that the user can quickly send several requests to the server and they all get spawned within a few game ticks of each other.

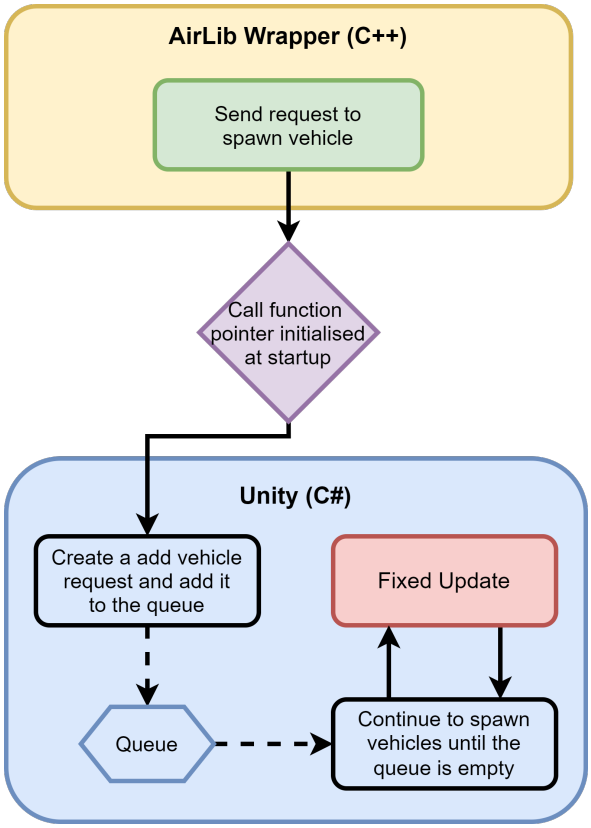

Figure 5.1: Unity is not thread-safe, so the server has to interact with the simulator through shared memory. For simplicity, the figure ignores everything that happens before the wrapper.

Another change that had to be made was when the server was instantiated. In the original implementation, the server was connected to the vehicle itself. This meant that the server was only running when there was a vehicle in the scene. To fix this a server object was added to the game scene. The server would now open when the server started, and close when the simulator stopped. (See Figure A.2 in the Appendix). Moving the server to a separate game object caused a few issues with the action order. However, these were fixed by moving the vehicle initialisation to the awake stage. Pedestrians would use the same structure once added.

The alternative to spawning the entities at runtime is to have an object pool, where all of the entities can wait until needed. However, this is not needed as the vehicles are fast to load. For more complex entities this could be a better option. The disadvantage of using this method is that there will then be a limit to the number of entities as users cannot generate

more if they run out.

5.1.3 Video feed

One of the requirements for this project was to have sensing APIs (Chapter 3). With the Unity game engine, AirSim can return a captured frame to the user interaction layer. The return is a struct that consists of the camera settings, image type, image width and height and the binary image data. This is then converted back into an image in Python. This API had to be updated as the frames rate returned by the server was low, especially when introducing more vehicles.

Figure 5.2 shows the existing implementation of how the fetch image API works. Note, only the AirLib wrapper and Unity components are displayed in the diagram. The first part is to pass the image request to the correct vehicle. The image request contains the camera name and type. Unity supports different image types such as scene, which renders the textures as usual, depth and infrared. The call then waits for one game tick and then returns the image response. As can be seen, this method has little impact on the simulator performance as capturing one image does not take long. However, the issue arises when several image requests are happening at once. As the server is single-threaded, the first request has to return before the next one is processed. Using coroutines in Python does not resolve this issue either as the requests will just be queued. One option is to open several servers as mentioned in Subsection 4.2.3. This however can produce a large overhead when there are many entities. The simulator is running at 120 game ticks per second.

Up to 5 cameras, this method works well. Assuming the calls happen one after the other and a capture takes one game tick, all 5 cameras can return an image in 5 ticks. This means that the frame rate is 24 frames per second. However, two vehicles would half that to 12 FPS, and so on. The output frame rate would therefore be $120/(n*c)$ where n is the number of vehicles and c is the number of cameras per vehicle. The advantage of this method is that it does not impact the tick rate of the simulator.

The following changes aim to increase the output frame rate and optimise the APIs so that calls take less time, whilst trying to limit the impact on the simulator tick rate.

When the vehicle starts, all attached cameras are loaded into a list. Instead of waiting for a request, the images are continually streamed to the server. This is illustrated in Figure 5.3. The user can enable and disable cameras to optimise the process. Every enabled camera stores the captured image in the server mapped to the vehicle and camera name. The figure shows how the API request does not have to interact directly with Unity, but can instead fetch the image directly from the server. This change means the server can request several hundred frames per second resulting in little delay between each API request.

Figure 5.3 makes it clear that an increased number of vehicles and cameras makes the update cycle slower. This means the game tick speed decreases. Figure 6.2 in the Testing and Results chapter (Chapter 6) shows the hyperbolic effect of how the game tick speed decreases with an increased number of cameras. It is important to note that the values are averages and could be impacted by other processes running simultaneously on the computer. The output frame rate will obviously decrease as well, but it will still match the game speed, so this does not matter. A way to counteract the increased number of cameras is not to render them on every game tick. The figure shows the average tick rate for the different ratios. A large ratio means that cameras are captured less often. This could be an issue if the vehicles were

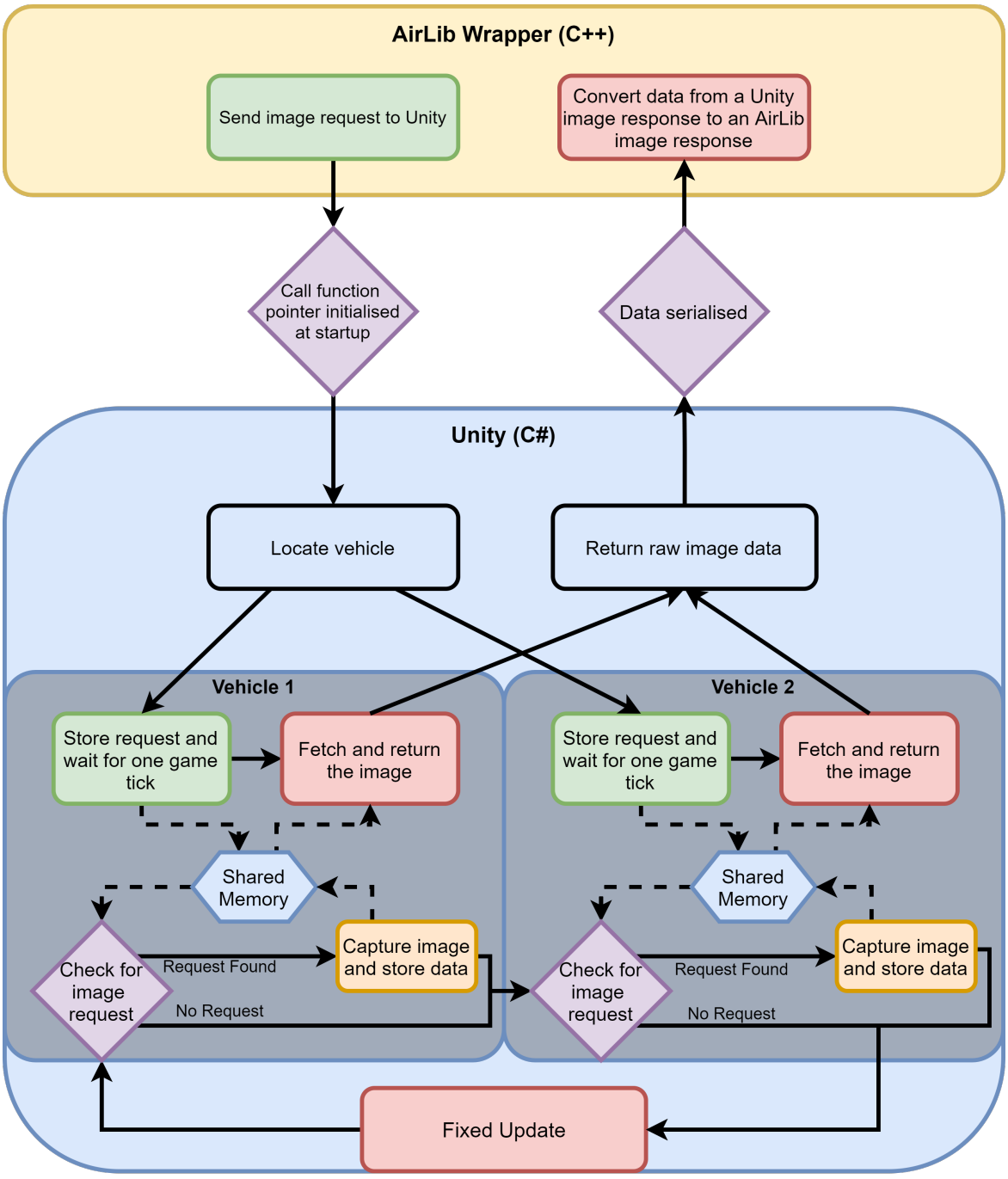

Figure 5.2: The figure shows how the image capture API works in Unity before it was updated to increase the performance. The response takes one game tick meaning that the many API calls can take a long time.

driving fast. It is also worth noting that a slow tick rate does not result in this issue. If the simulator tick rate is slow, and images are captured on every tick, the vehicle would only have moved a small amount as the simulation is running slowly. This means that compared to the simulator, the Python script can make a fast decision. It is also possible to deliberately decrease the tick speed if the processing in Python is slow.

Overall, the outcome of this change is that there is a drastic increase in the framerate from the server. However, for a large number of cameras, the tick speed in Unity decreases. Future work could look at using UDP streaming to further increase the frame rate.

5.1.4 Adding Pedestrians

This section will cover how pedestrians were added to the simulator. Pedestrians are one of the main forms of traffic wanted in this simulator. The pedestrians would be controllable through APIs in the same way as vehicles.

The first step was to figure out how to create the pedestrians. The first option would be to use the standard Unity objects to create a character. This would be a simple capsule with a face. This would have been a bit simple for this project and having something more human-looking would be preferable. Another alternative was to download characters from mixamo[1]. Mixamo is a technology company owned by Adobe which designs 3D characters and animation. The advantage of using Mixamo is that the characters come with a large range of animations as well as looking like humans. The disadvantage was having to download a large number of assets. Another way of creating character would be with the Unity character generation tool known as UMA2 (Unity Multipurpose Avatar). This allows for the character to have a variety of different shapes and attires. UMA can generate random models or they can be generated manually. An extremely large range of customisable features are available such as eyebrow shape or mouth width. Figure 5.4 shows some randomly generated characters inside Unity. These models will be used as the pedestrians.

The next problem that had to be resolved was how to control the pedestrians. This was surprisingly simple. A character control script was already downloaded from the asset store when trying to use the Mixamo characters. Simply attaching this control script as well as the animation controller from the same asset made the characters run around. To make them walk, slowing down the direction speed worked. Another designed choice was made to have the controls for the pedestrians work similarly to the vehicles, i.e. pressing left and right would rotate the pedestrian rather than have it walk sideways on a grid.

To make the pedestrians separated from the vehicles the server was divided into three components, the game server, pedestrian server and vehicle server. This was explained in the design chapter (Subsection 4.2.3). Most of the basic APIs such as fetching available pedestrians, controlling the pedestrians and image capturing had to be reimplemented. These features were done similarly to how they work for the vehicles. The main difference being the struct with control commands being different.

The next issue to be resolved was how to efficiently have UMA as a part of the project. UMA2 is over 500 MB and it would be unnecessary to have all of this pushed to GitHub. The solution was instead to have UMA as a requirement from the asset store. Prefabs were then created which would interact with the downloaded code. Script relating to AirSim would then be attached to these objects.

[1] `https://www.mixamo.com/`

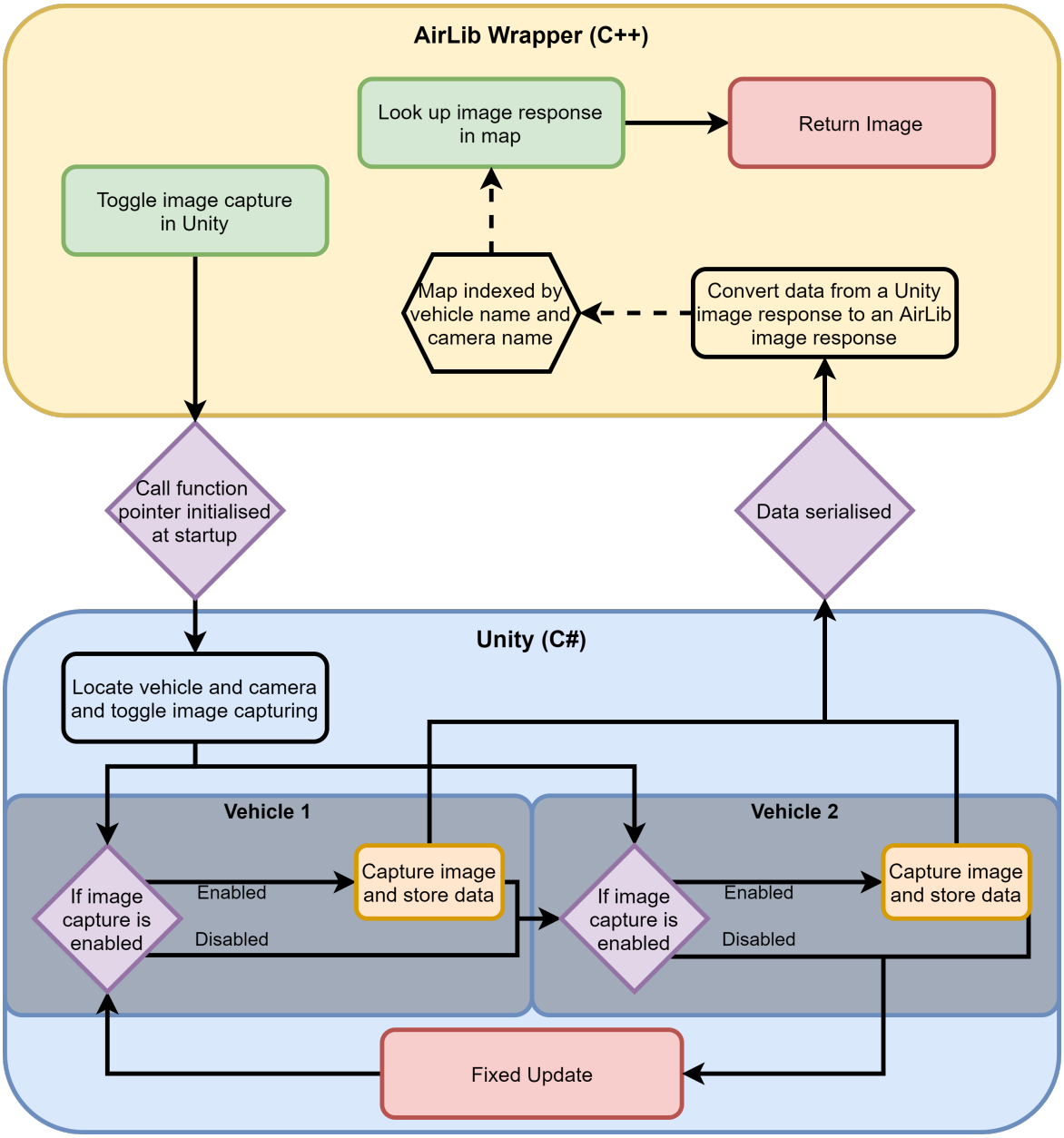

Figure 5.3: The figure shows the updated overview over how the server stores the image to allow for faster API request, but in return slows down the Fixed Update loop which links to the game tick speed.

Another small design decision that was made was how the cameras should behave. The animations make the pedestrians casually look around when standing still. To avoid this being an issue, the cameras are fixed facing forward from each eye. This means that the cameras are completely still when the pedestrians are walking.

Overall the pedestrians took the most time to add to this project. Including dividing the server there was a lot of code that had to be changed throughout AirSim. One of the main challenges was that Unity does not display the error if there was an error caused by the AirLib DLL file. This made debugging the servers very difficult. Another issue that had to be resolved was that the original Unity version was too old to use all the UMA features. Updating Unity to a new version meant resolving other issues in AirSim and downloading outdated extensions.

Figure 5.4: Randomly generated pedestrians using UMA2. UMA can be extended further by downloading additional assets which will add additional clothes, hair styles and more. UMA also allows for custom looking characters.

5.1.5 Additional APIs

This section will look at what additional APIs were added or modified. It is worth noting that some of these features took a long time to add as they had to be done completely from scratch, whilst others could be done in less than an hour.

Vehicle APIs

- Updated **enableAPIControls** - This is used to toggle the driving mode from through the API or manual inside the simulator.
- Updated **simPrint** - Prints a debug statement from the vehicle.
- Added **getVehicleTypes** - This returns a list of strings of the available vehicle types. This required significant changes to the simulator with roughly 15 files changed and 150 lines of code. The list of strings get converted into a char arrays with null terminated strings in C# and then converted back into C++ strings in AirLib. Characters in C are 16 bits, so every other character is skipped when reading the char array in C++. This whole process was illustrated in Figure 4.2. The commit with this can be found

here[2]. A lot of time was spent trying to pass the arrays of strings from Unity to AirLib and then AirLib to Python.It is also worth nothing that this list could be extended with autonomous wheel chairs and other entities that would need the same controls and sensing abilities as a car.

- Added **simGetAllVehicles** - Returns a list of all vehicles in the scene. This is based off the getVehicleTypes API.
- Updated **setCarControls** - Updated how this was handled in Unity with several vehicles. Struct of car controls is passed from Python to Unity.
- Updated **getCarState** - Updated to handle several vehicles. Returns information about the car state such as speed, direction and location.
- Added **simGetLidar** - Added an API to fetch values form a Lidar beam. This is implemented in Unity using a ray cast to measure the distance to another object. Ideally this would be configurable in the request, but currently the beam is always from fixed points.
- Updated **simGetImages** - Fetches images using coroutines in Python. Decodes them and stores images in a dictionary indexed by the vehicle and camera name.
- Added **simGetCameras** - Gets the names of the available cameras attached to the vehicle.

Pedestrian APIs

- Added **ping** - Shows there is a connection with the pedestrian.
- Added **reset** - Reset the pedestrian back to its original starting state.
- Added **setPedestrianPose** - Sets the pedestrian position and rotation.
- Added **getPedestrianPose** - Gets the pedestrian position and rotation.
- Added **enableAPIControls** - This is based off the vehicles API.
- Added **simPrint** - Prints a debug statement from the pedestrian9.
- Added **simGetAllPedestrians** - Returns a list of all pedestrians in the scene. This is based off the getVehicleTypes API.
- Updated **simGetImages** - Fetches images using coroutines in Python. Decodes them and stores images in a dictionary indexed by the vehicle and camera name.
- Added **simGetCameras** - Gets the names of the available cameras attached to the pedestrian.

Server APIs

- Added **ping** - Shows there is a connection with the server.
- Added **getTickRate** - Fetches the current simulator tick rate in frames per second.

[2] `https://github.com/tobhil98/MastersProject-AirSim/commit/22ab7c`

- Added **simAddVehicle** - Explained in detail above how it is implemented. Spawns in a vehicle at a rotation at a specific location
- Added **simAddPedestrian** - Spawns in a random pedestrian.
- Added **simRemoveVehicle** - Removes the vehicle.
- Added **simRemovePedetrian** - Removes the pedestrian.
- Moved **simPause** - APIs used to control the simulation speed. Moved from the old vehicle server.
- Moved **simLogPrint** - Sends a debug print statement to Unity.

5.1.6 Minor Features Added

This section will briefly list other features added to the simulator.

- **Camera movement** - To be able to look around the simulator several different camera configurations have been implemented. These are to use in the simulator and cannot be controlled through APIs. One option is a free camera which allows the user to fly around the environment. The second option is to follow a specific entity. These options have been mapped to three number keys, key 1 for free movement, key 2 to follow vehicles and key 3 to follow pedestrians. When following vehicles or pedestrians the user can cycle through them by pressing tab (or shift+tab to cycle the other direction). The user can also easily move from following an entity to free camera movement by pressing the left shift key.
- **Rebuild script** - This is a simple update to the build script that compiles the AirLib dll. This change allows for faster compile time. This is very beneficial as it reduces the time to compile from over 10 minutes to just over 1 depending on the change made to the code. This change was a very small change to the build script which meant only files that had to be recompiled were recompiled, instead of rebuilding the whole project.
- **Upgraded Unity version** - The version Unity version used by AirSim by default is 2019.3.12, however there is a bug in this version which means internal projects cannot communicate with each other. The main issue here was that there was no way of communicating with the UMA code from the AirSim code. This was resolved by upgrading the version to 2019.3.13. This introduced a bunch of dependency issues as well as a couple of code issues that had to be fixed. This change should not impact future pull requests from the master repository as most features that are commonly used were backwards compatible.

5.1.7 Other Features Considered

This section will briefly mention other features considered when adding features to Unity. The main reason for not adding these was in the interest of time.

- **Unity Navmesh**[3] - This would allow pedestrians and vehicles to navigate around autonomously without needing to train a network. The navigation mesh can be laid on any surface to indicate where entities can go.

[3] `https://docs.unity3d.com/Manual/nav-CreateNavMeshAgent.html`

- **Unity Cloud Build**[4] - This is a framework that would allow testing the system online. However, this requires an advanced Unity license, so was not feasible for this project.

5.1.8 Extensibility

The advantage of using AirSim and Unity is that the whole system is very flexible. As mention in the design chapter (Chapter 4), AirSim allows for a variety of different languages to interact with the APIs. The whole project has made sure that future extensions should be simple to add if needed. This section will briefly look at how the existing features can help extend a future design.

- **Additional APIs** - This can be slightly complicated depending on the API. The API to base the new API off should be simGetImages. This is because it passes a struct of arguments to Unity, does some complex processing and passes a struct back. The struct has to be converted to different formats as it goes through AirSim as was shown in Figure 4.2.
- **Additional Vehicle types** - This is done by adding the control script to the vehicle and then adding the prefab to the asset manager in the scene. To get all available vehicles for example the user could use the getVehicleTypes API which returns the names of the different types. Several types of vehicles have been added including cars and trucks as shown in Figure5.5.
- **Completely new types like wheelchairs** - This can either be added like an additional vehicle type with different physical properties. This would then have to use the same vehicle controls. It is also easy to look at how the pedestrian server works. Now the servers have been divided, creating more similar servers to how pedestrians were created is simple.
- **Customise cameras** - Adding additional cameras to an object is done by creating a camera object onto the model and renaming the camera. The vehicle controller will find the object called Capture Cameras and iterate over the object to find all the cameras. The fetch available camera API listed above fetches all cameras attached to a specific object, so different vehicles can have a different number of cameras.

As AirLib works as a plugin to Unity, all Unity features are still available to use. This means adding something like VR or AR to the simulator should not be difficult if required.

[4] `https://unity3d.com/unity/features/cloud-build`

Figure 5.5: These vehicles are from the asset store and have been added to the simulator. These have the same features but different physics.

5.2 ML-Agents

This section will look at how ML-Agents[5] can be used to create machine learning models for the vehicles and the pedestrians. ML-Agents is an open-source project designed to train agents using reinforcement learning and imitation learning. The purpose of this is to allow users to have multiple entities in the scene without controlling all of them. ML-Agent is both imported as an addon to Unity, but it also runs as a separate Python library. When the simulator starts, Unity would then open a port to communicate with Python.

For this project, the hope was to simulate the interaction between the different forms of traffic. However, creating and uploading the simulator onto an external server to train can take a long time. The priority was therefore to train a model for the vehicles. This was done by first training on one vehicle and then looking at collaboration learning. As will be discussed in chapter 6, the vehicles have mixed results and time was spent trying to improve this rather than developing the pedestrians further. The ML training scripts can simply be attached to the pedestrians as shown in figure 5.11.

5.2.1 Network Model

A variety of different model designs have been attempted throughout this project. This section will look at what decisions were made and why that was the case. More information on the tuning can be found in Chapter 6.

Model inputs

The first thing to look at is the inputs to the model. The common way when looking online for making autonomous vehicles using ML-Agents is to use checkpoints that span the road. The vehicles would then learn to drive towards the checkpoint and away from the walls. The reason why this would not work for this project is that the rays get blocked by the

[5] `https://github.com/Unity-Technologies/ml-agents`

checkpoint. If there is a pedestrian or a car on the other side the vehicle will not observe them. The solution was instead to guide the vehicles in the direction they should drive. The vehicle is therefore given the angle between its direction and the vector going to the target. This was done by using the calculate angle command in Unity which only returns positive values. For the vehicle to know which way to turn, a simple calculation was made where if the target was on the left the angle would be multiplied by -1. The input angle was then scaled to be between -1 and 1.

To further increase the performance, the angle was passed through different scaling functions. This can be seen in Figure 5.6. The reason for doing this was to strengthen the input single for smaller angles. This would improve the performance for two reasons. Firstly, as the car cannot turn more than 60 degrees the input angle could be scaled in such a way that for larger angles the input would be close to 1. Passing the input angle through a tanh function strengthens the sensitivity for smaller angles. Therefore, the vehicles use $tanh(2.5x/180)$ where x is the angle.

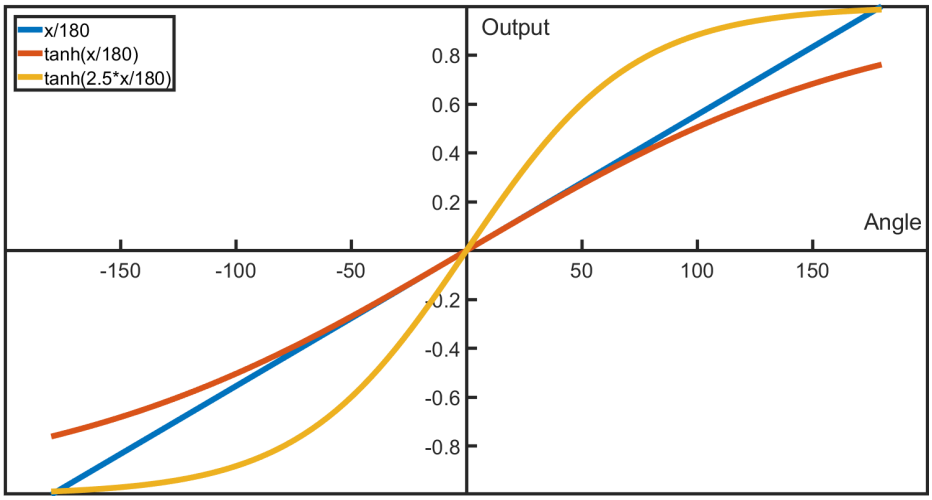

Figure 5.6: The plot shows different functions applied to the input angle. The aim of using $tanh(2.5x/180)$ is that it decreases the change for larger angles, but increase for smaller.

For the sensing APIs, the options are either to use raycasts or to use a camera sensor. The advantage of using raycasts is that network does not have to be as complex as for the image processing. Another advantage is that when building the simulator as a server build, enabling the rendering is slightly complicated and decreases the speed of the simulator. The advantage of using the camera sensor is that it can more accurately simulate the real environment as a complex network could distinguish what the camera is capturing. In Unity, this is not needed as the rays can distinguish what type of object it is detecting. However, in the real world, this is impossible. The next step would be to calculate how many inputs from the raycasts there will be. The 3D rays come with several configurations[6] such as the number of rays per direction, max ray degrees, detectable tags and observation stacking. The total size of the created observations is $(ObservationStacks)*(1+2*RaysPerDirection)*(NumDetectableTags+2)$. Observation stacking can be used to give the network a limited short term memory. This is done by repeating observations from previous steps. For example,

[6] https://github.com/Unity-Technologies/ml-agents/blob/main/docs/Learning-Environment-Design-Agents.md#raycast-observations

if the input from a sensor was

$$\begin{aligned} step1 &: 0.1 \\ step2 &: 0.2 \\ step3 &: 0.3 \\ step4 &: 0.4 \end{aligned} \tag{5.1}$$

then the stacked input would then look something like this:

$$\begin{aligned} step1 &: [0.1, 0.0, 0.0] \\ step2 &: [0.2, 0.1, 0.0] \\ step3 &: [0.3, 0.2, 0.1] \\ step4 &: [0.4, 0.3, 0.2] \end{aligned} \tag{5.2}$$

Changing observation stacking made very little difference to the performance of the network, so this value was left at 1.

The next two values to decide was the number of rays per direction and the spread of these rays. Ideally, the agent should use as few as possible whilst at the same time having good coverage of the surrounding area. When using too many inputs, the model was had difficulty figuring out where to drive as the signals from the ray sensors would overpower the one pointing the direction. The best result was found with about 6 rays per direction and an angle of 120 degrees. This would be a total of 13 sensors as one is pointing forward. The angle is wide enough so that if an entity comes from behind the vehicle would know not to turn into it. To compensate for fewer sensors the ML agents use a sphere cast rather than a ray cast. This means that the detection area is much larger than a single point. This will help detect pedestrians as can be seen in Figure 5.11. This limits the probability of the pedestrians ending up being between two rays.

The last input that was given to the model was speed so that the model would have some understanding of its momentum. Ideally, the vehicle would use this to control how much braking was necessary.

Model outputs

The output of the model matches the vehicle controls. Therefore, The output consists of two values, one for the throttle and breaking and one for turning. The main design decision that had to be made was whether to discrete or continuous output values. Chapter 6 will look further into this, but basically discrete values work better. This is because the vehicle speed grows with the square root of the input value. Discrete values worked therefore better as they produce strong decisions.

5.2.2 Learning Environments

When training using reinforcement learning it is important to have a good environment. Four different environments were used with increasing complexity. To help improve the robustness of the models, the start position and rotation would vary slightly between runs.

The first map (Figure 5.8) was designed to help the agents learn to drive forward and turn. Every corner had a checkpoint where they received a reward. The issue with this map was

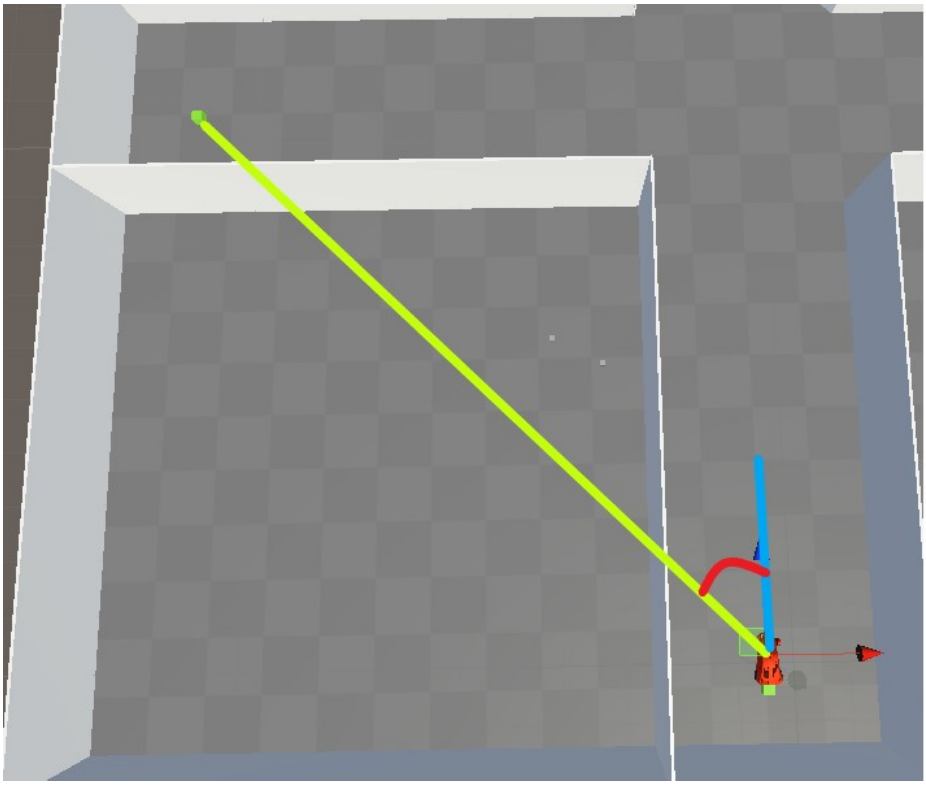

Figure 5.7: The input angle to the model is the angle between the forward vector and the vector going from the vehicle to the target.

that there was not enough randomness, so the map was straightforward no matter where the agent started.

The second map (Figure 5.9) was a more complex map where the agents would have to turn both left and right with different amounts as they drove around. When training on this map the checkpoint order would flip every other run. This decreased the vehicle performance on this map but increased the robustness.

The final training map (Figure 5.10) was designed as an intersection where the vehicles would have to navigate around each other to avoid a collision. Up to four vehicles would spawn in one of the checkpoints positions and then drive to a different one. The map was designed in such a way that vehicles would have to wait for each other if they were both going to the same location.

5.2.3 Game controller

The game controller controls the learning environment. When the episode starts, the controller will spawn the vehicle in a random location with a random target. The controller keeps track of how long the episode has gone on and aborts the training if it takes too long. The controller will then reset the map and spawn a new vehicle in a random position. If a vehicle reaches its target, or if it collides with the wall, the game controller would remove that vehicle from the scene.

The game controller is also responsible for giving team rewards for collaboration learning. This will be explained further in Section 5.2.5.

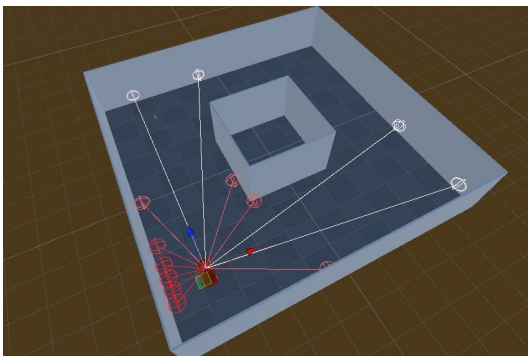

Figure 5.8: Simple training environment for the vehicles to drive around.

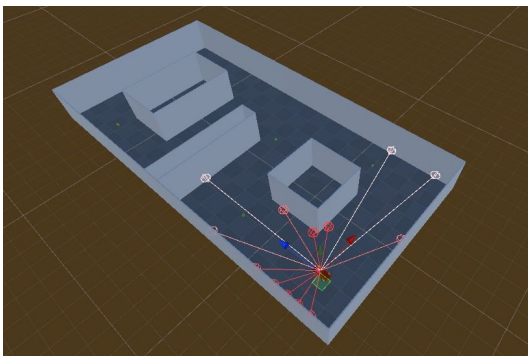

Figure 5.9: A slightly more complex map where the vehicles would have to do both left and right turns.

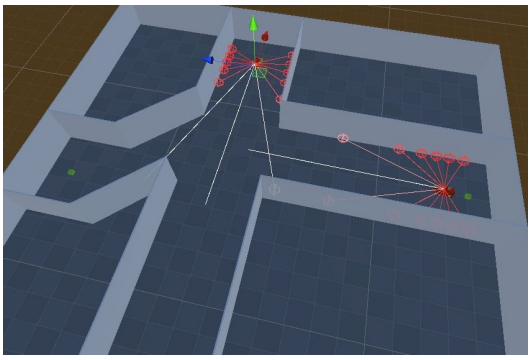

Figure 5.10: An intersection map to help the collaboration learning.

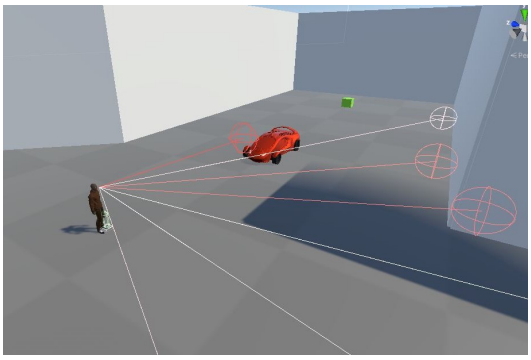

Figure 5.11: Intersection map where a pedestrian meets a vehicle.

5.2.4 Learning by Demonstration

Learning by demonstration is used to help the agents learn faster. This is also known as imitation learning. This is done by recording some test runs which the agents would then try to replicate. Imitation learning has been crucial for this project as without it the agents never built up enough momentum to move. When using continuous output values the agents never figured out that holding down forward over a longer time would gain momentum. This was because the reward was only given when the agent moved towards the target. ML-Agents supports learning from demonstration by using Generative Adversarial Imitation Learning (GAIL). GAIL works by creating a second neural network that will try to learn if the observed behaviour is produced from a demonstration or from the agent. If the GAIL network guesses incorrectly the vehicle model receives a reward. This can be computationally slower than other options as two networks are being trained at once.

Alongside GAIL behaviour cloning can be used. BC aims to train the network in such a way that it will mimic the behaviour from the demonstration files. Behaviour cloning cannot generalise past the examples shown in the demonstration, but together with GAIL it often produces results than without as can be seen in figure 5.12.

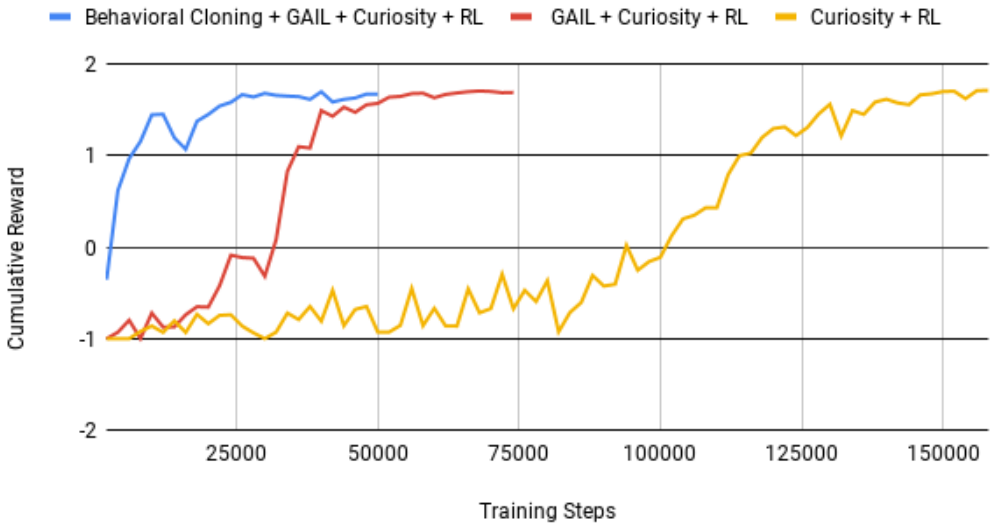

Figure 5.12: The figure shows that using BC and GAIL decreases the number of training steps it takes to train the model.
`https://github.com/Unity-Technologies/ml-agents/blob/0.15.0/docs/Training-Imitation-Learning.md`

5.2.5 Rewards and Collaboration Learning

This section will look at the reward system implemented and how that affects collaboration learning. The reward system changes as the vehicles improved. In the early stages, a reward was given when the vehicles moved forwards. Gradually increasing the difficulty is called Curriculum Learning. The agents were also given an additional reward if they faced in the direction of the target. Eventually, this simple behaviour was not needed as later learning was based on this initial model. Currently, vehicles receive a reward every time they reduce the distance between themselves and the target. This caused the vehicles to understand that driving directly towards the target is not necessarily the optimal solution. The vehicles re-

ceived a large reward when the target was reached and received a large penalty if it collided into the wall. A small penalty was also given every time step to reward faster completions.

For collaboration learning the reward system is slightly different. MA-POCA (MultiAgent Posthumous Credit Assignment) can be used to give team rewards as well as individual rewards. This will teach the agents to work together. When using collaboration learning teams received a reward every time a vehicle reached its target but received a negative reward if a collision occurred. MA-POCA can also be used to train teams against each other. This feature will not be used for this project.

5.2.6 Training

This part will coverer how training with ML-Agents works. Hyperparameter tuning will be discussed in Section 6.2. ML-Agent supports several kinds of models, but for this project, Proximal Policy Optimisation (PPO) and MultiAgent Posthumous Credit Assignment (MA-POCA) is used[23].

Models

For individual learning, PPO or Soft Actor-Critic were considered. PPO is shown to be more general-purpose and stable than SAC[24]. SAC uses learn experiences to train. The experiences are placed in a replay buffer which it randomly selects from during training. PPO however learns directly from the environment. SAC is better designed for slower environments, so an autonomous vehicle would be too fast for it. PPO was therefore used for this first part.

For collaboration learning MA-POCA. This was described in Section 5.2.5. When training only one vehicle the episode would end as soon as either the vehicle reached the destination or it collided with the wall. If there were several vehicles in the scene having the whole episode end if one vehicle collided with the wall seemed illogical, as it would make it too hard for the other vehicles to learn. The solution is therefore to only remove the vehicle which collided or reached the target. POCA is designed in such a way that even removed vehicles still receive additional team rewards.

Improvements

Curiosity[7] was also tried. Curiosity allowed for random rewards to help the network explore different options.

As mentioned in Section 5.2.4, GAIL and BC were used to help speed up the training by replicating the saved demonstrations.

Environments

There are two ways of speeding up the training. The first one is to create multiple instances of the learning environment (Figure 5.13). This allows several agents to learn simultaneously. The game controller belongs to the environment prefab, so for collaboration learning, each map can operate by itself.

[7] `https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md#curiosity-for-sparse-reward-environments`

The second option is to compile the simulator into a server build that has no GUI. This was often compiled to Ubuntu so that the simulator could train on external servers. When ML-Agents start it can instantiate several instances of the simulator which can run in parallel. Not having the GUI and being able to run on a separate computer drastically decreases the learning time.

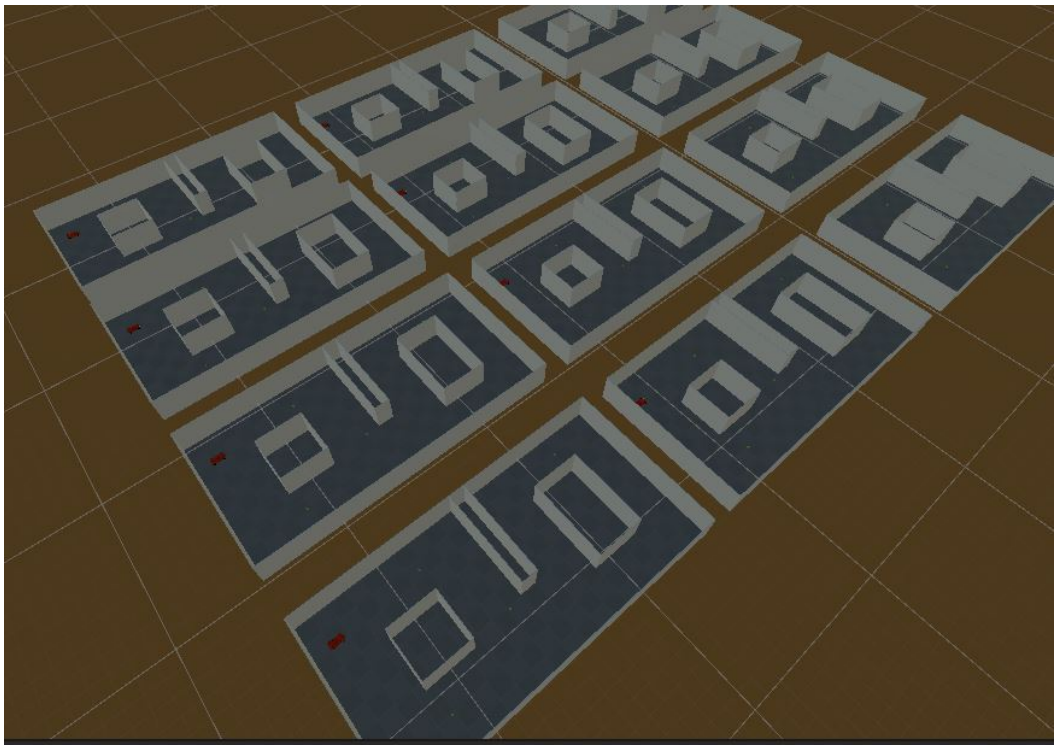

Figure 5.13: More instances of the learning environment helps speed up the training.

5.3 Maps and environments

This section will look at different ways of importing or creating city maps in Unity. The purpose of this is to give the agents an environment that more accurately replicates the real world. There are two main ways of doing this. The first option is loading the maps in at runtime. This option would give the agent an unbounded area to navigate around. The second option is to create a Unity asset. This option allows the user to create a 3D model of the environment and load the object into the scene at compile time.

5.3.1 Unity Map SDKs

There are several different Map SDKs available for Unity. The advantage of using an SDK over an asset is that it allows the user to navigate any place. As the agent navigates around, new areas of the map will be loaded. Another advantage is that the maps are updated and can provide real information such as traffic congestion. A disadvantage is that these maps need to be download at runtime. This requires access to the internet and can be slow to load. Another disadvantage is that these services are subscription-based. This means that there is a limit to the number of requests that can be made [8].

[8] `https://www.mapbox.com/pricing/`

Google Maps Unity SDK

Google Maps Unity SDK contains several developer tools which allow the user to create mobile games with real-world locations[25]. The advantage of using this SDK is that it provides additional tools such as the ability to extract place IDs as well as the name of geographic features. The SDK also includes real-world features from particular locations. The disadvantage with the Google Maps SDK is that it currently only supports mobile applications. The simulator is primarily designed to run on a computer so this SKD would therefore not work for this project.

MapBox Unity SDK

The MapBox Unity SDK is a toolbox that can be used to create worlds with continuous road networks, points of interests and street labels. It also can use satellite images to create realistic terrain. The advantage of the MapBox SDK is that it is free and easy to set up. The disadvantage is that the generated textures do not look as good as other map options. There are also some missing buildings as can be seen from Figure 5.14. MapBox does allow the users to modify the maps online, but this has to be approved before the maps are updated.

Wrld3D Unity SDK

Wrld3D Unity SDK is a dynamic 3D mapping platform that can replicate indoor and outdoor environments. Similar to the SDK created by Google, famous landmarks and features are recreated. The SDK comes with a variety of different APIs. These include, for example, select and highlight buildings, enter and exit indoor maps and visualise the transport network. As can be seen from Figure 5.15, the applied textures look better than the textures provided from the MapBox SDK (Figure 5.14). Wrld3D costs 20 USD per months making it one of the more expensive options. The maps are however more visually appealing than the other options. In addition, the maps have smoother and more accurate surfaces making it easier for the simulation, compared to for example the Google 3D maps loaded into Blender (Figure 5.18).

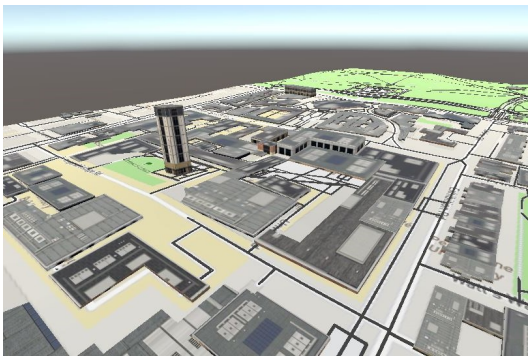

Figure 5.14: Imperial Campus loaded into Unity using the MapBox SDK. All objects have been given an arbitrarily texture. Also, the building scale is off.

Figure 5.15: Wrld3D has a tool online where you can look at the map before purchasing the SDK subscription. The image is a screenshot from this tool[a].

[a] `https://maps.wrld3d.com/?mapscene=632ba4b`

5.3.2 Unity Map Assets

Loading map assets into Unity requires more initial work as they require 3rd party tools. An advantage of creating assets is that it gives the user the freedom to modify the maps before importing them into the simulator. Blender is commonly used as a free program for creating and modifying 3D models[16](Section 2.2.3). Another advantage is that the maps are loaded at compile-time. This means that the model loads quickly and the simulator does not require access to the internet. The disadvantage is that the agents will be limited to only that environment. Large maps can be computationally hard to render and cause the simulator to lose performance. This is due to the large polygon count which requires more memory and CPU usage. (More information in the User Guide Section D). Depending on the method used to create the models, creating better-looking options using Blender can be very time-consuming and requires additional skills to use the tool.

OpenStreetMap2World

OSM2World is an open-source program that takes maps generated by Open Street Map[9] and them into object files which can be loaded into Unity. There are several advantages of using OSM2World to create assets. Firstly, this program is easy to set up and generate object files. The user can either use the GUI or use the CLI. Another advantage is OSM2World creates objects which consist of several layers. These layers are different features the world consists of, for example, roads, junctions, footpaths and buildings. Having these layers allows the user to interact with each of them separately. Users can manually modify these layers on the OpenStreetMap website[10].

The disadvantage of using OSM2World is that it does not allows to render the objects with texture in Unity. It is also difficult to make adjustments to the environment without using another tool like Blender. These adjustments could be updates that the user would like to have for their environment, but are not changes that should be uploaded to the map itself, like adding additional walls to close the area.

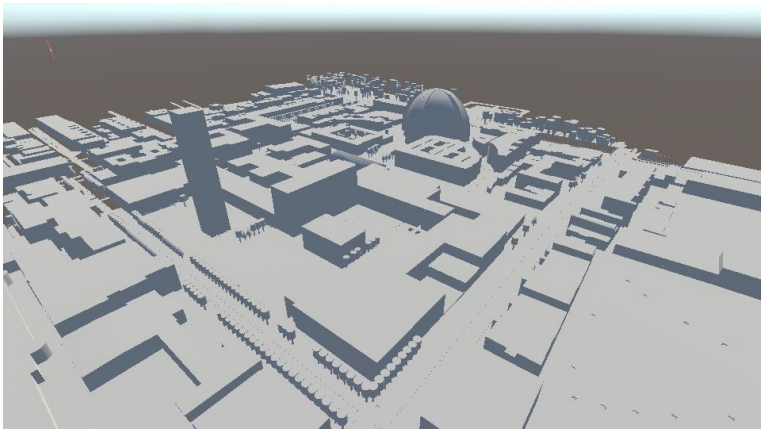

Figure 5.16: South Kensington Campus created using OpenStreetMap2World loaded into Unity

More information on how to create an object file from OpenStreetMap can be found in the user guide (Chapter D in the appendix).

[9] `https://www.openstreetmap.org/#map=16/51.4976/-0.1715`
[10] `https://www.openstreetmap.org/`

BlenderGis

BlenderGis is an open-source Blender addon that allows users to import GIS (geographic information system) files. The addon includes the option to download a map area from either Google or OpenStreetMaps directly into Blender without having to download the GIS files externally. One advantage of using BlenderGis over OSM2World is that GIS files include the world heightmap. This means that this method can more accurately map the terrain.

BlenderGIS can also be used to import structures and buildings like OSM2World, and the quality is somewhat similar. This Blender addon also includes the different map layers like OSM2World, but there are not as many options. Different kinds of roads, junctions and so on are all just marked as highway.

The main advantage of using BlenderGIS over OSM2World is that BlenderGIS allows adding texture to the map. The satellite image is projected onto the map ground, making the scene much more recognisable. There is also a way of projecting the textures onto the buildings. As the satellite images are only taken from above, there is no way to add texture to the sides of the buildings without using a generic front.

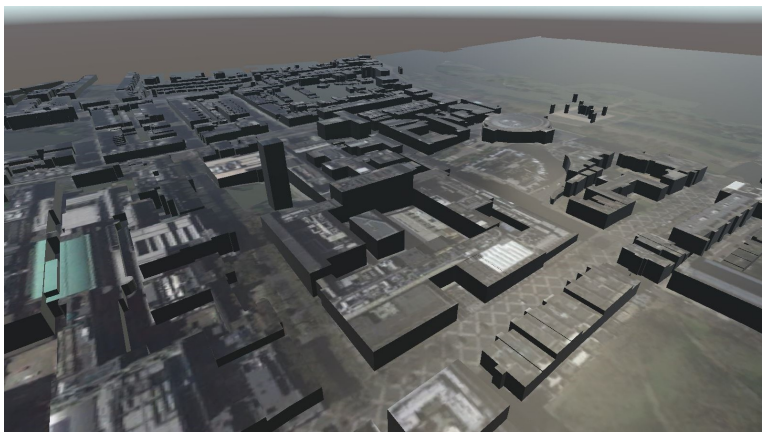

Figure 5.17: South Kensington Campus created using BlenderGis, then load into Unity. Some buildings are missing and the texture looks quite flat.

Google Maps GPU Intercept

For personal projects, Google Maps' 3D view can be imported into Blender. These steps are quite convoluted, but the user guide (Chapter D in the appendix) explains how to do this in more detail. In essence, renderDoc can be used to intercept the data going from Google Chrome to the GPU. This data can then be loaded into Blender by using MapsModelImporter[11]. This step can take a long time as many polygons have to be loaded. Once the map has been imported into Blender, the user can freely update the model. One update that should be made is to reduce the polygon count. Blender has a tool for this. This is needed as there are several thousand polygons, and this process can remove almost half without a noteworthy difference (Figure 5.18). Finally, the model along with the textures can be exported to Unity.

The advantage of using this method is that it looks a lot better than the other options. There is also a lot more detail which makes the environment more realistic. This can be useful

[11] `https://github.com/eliemichel/MapsModelsImporter`

when training machine learning models as the trained model would not overfit to a perfect environment.

However, there are several disadvantages to using this method. Firstly, creating maps using this method is a lot more time-consuming. Importing the map into Blender, then exporting it to Unity can take several hours due to a large number of polygons. This method is also a lot more computationally expensive when running in Unity, as each polygon has to be rendered. Secondly, the roads are not even and flat, as cars on the roads are incorporated into the model. This can be smoothed using tools in Blender, but this is once again time-consuming. The last disadvantage is that the model does not include map layers which the other method do. This means there is no way of distinguishing different objects, like buildings, trees and parked cars, apart.

Figure 5.18: South Kensington Campus loaded from Google Maps into Blender, then exported to Unity. Number of polygons reduced by 60%

5.3.3 3D World Scanners

Another way to create an environment is to use a mobile application to scan the surrounding area. This will generate a 3D model which can be imported into Blender. Similar to the GPU intercept method, the created environments contain a very high polygon count and can have some rough edges.

There are several apps which does this, but most are not free. Display.land was widely used until it was taken down a few months ago. According to their website[12], the company has plans to release a new version soon. The best application found for Android turned out to be Scann3D[13]. It can produce simple models of objects, but struggles with larger areas. The quality of the output is not very good. According to different sources online, trnio[14] works well for iOS, but this has not been tested.

5.3.4 Conclusion

For this project using map assets rather than an SDK would be better. As the simulation is about the interaction between different kind of agents, an infinite map is not required. Using

[12] `https://get.display.land/`

[13] `https://play.google.com/store/apps/details?id=com.smartmobilevision.scann3d`

[14] `https://www.trnio.com/`

an SDK is a good alternative if the specification for the project changed and training agents at several different random locations became more important. The benefit of the faster loading time and not having to rely on an external API at runtime makes loading the environment as a Unity asset the best option.

In regards to which of the asset methods to go for really depends on the task. For a visual demonstration, the optimal choice would be to overlay the data from Google Maps on top of the height map generated by BlenderGis. This can be seen in Figure 5.19. This approach would then allow for smooth roads whilst keeping the detail on the buildings and other structures.

If the visual is not important, using something like BlenderGis would work well. This allows for a layered map as well as providing the terrain height. It is also easy to set up and use.

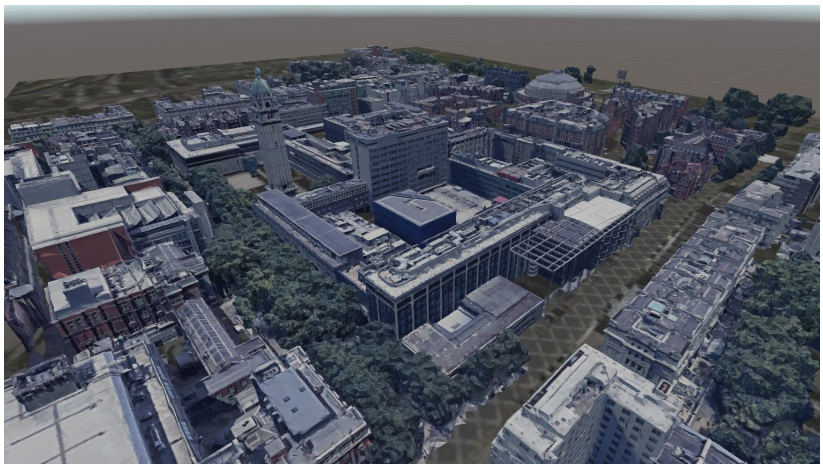

Figure 5.19: BlenderGis and Google maps combined. Allows for smooth roads whilst keeping the aesthetics of the buildings. Downside is that its computationally intensive to run and file size over 100MB.

Chapter 6

Testing and Results

This chapter will look at the performance of the Simulator as a whole as well as the performance of the ML-Agents. As this is only a prototype the performance is not the most important aspect. The chapter will also look at the experiments conducted.

The results studied in this chapter would be from individual features of the simulator rather than the simulator as a whole. The features that were added to the simulator were mentioned in the implementation chapter (Chapter 5).

6.1 Simulator Performance

This section will look at what experiments were conducted on the simulator to evaluate its performance. It is worth noting that as this is a prototype, the efficiency of the implementation is not important.

A lot of testing is also done manually. Features that are not done through the API such as camera control and vehicle handling was therefore not tested using scripts.

6.1.1 Basic Features - API

The python client folder on GitHub[1] contains a variety of scripts that could be used to test the simulator. There are no unit tests however as Figure 6.1 shows how a test can be implemented. The green boxes indicate actions that have to be done by the client, and the red boxes indicate API calls.

Inside the car folder the car_stress_test is very similar to this figure. Here a car will spawn and drive in a direction for a second return the fetch internal car state, stop, and then drive in another direction for a second. The internal car state will contain information such as the position and speed of the vehicle. This test shows that the APIs work and that the vehicle asset is available.

Another test car_test_many. This has been used to test how many entities the simulator can handle. If the cameras are not enabled the simulator can handle over 100 entities, however, as can be observed from Figure 6.2 not many the simulator suffers quite quickly when there are several cameras in the scene.

[1] `https://github.com/tobhil98/MastersProject-AirSim/tree/master/PythonClient`

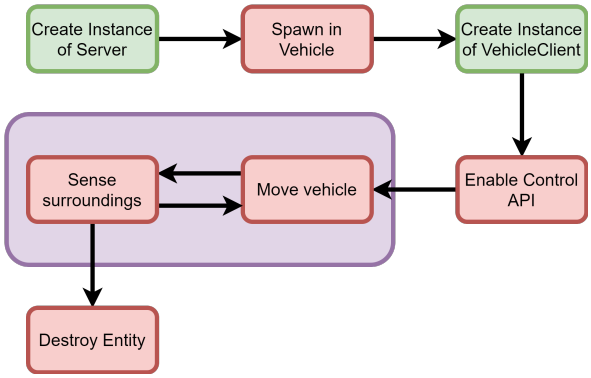

Figure 6.1: Red boxes indicates API calls, whilst green indicate processing in Python. The purple box is to indicate the functions happening when the simulator is running and the vehicles are moving around.

6.1.2 Testing the video feed

Figure 6.2 illustrates the decreasing tick rate of the simulator as the number of cameras increases. As was mentioned in Section 5.1.3, this was introduced to counter the slow response time. A key point to note here is that this diagram looks at the tick speed of the game, not just the frame rate. Cinemas typically show their films at 25 frames per second, and as can be observed from the figure, with around 75 cameras the simulator can run at around 20 ticks per second if the frame rendering ratio is 1:6. Even though the simulator looks smooth, it would be running much slower than usual, as the normal speed would be 120 ticks per second. The simulator is therefore running at roughly 16% speed, making moving around and driving a vehicle difficult. Currently, this is a limitation of Unity. Unity does not allow threading of monobehavior which contains all the functions that can interact with the simulator. (See appendix Figure A.2)

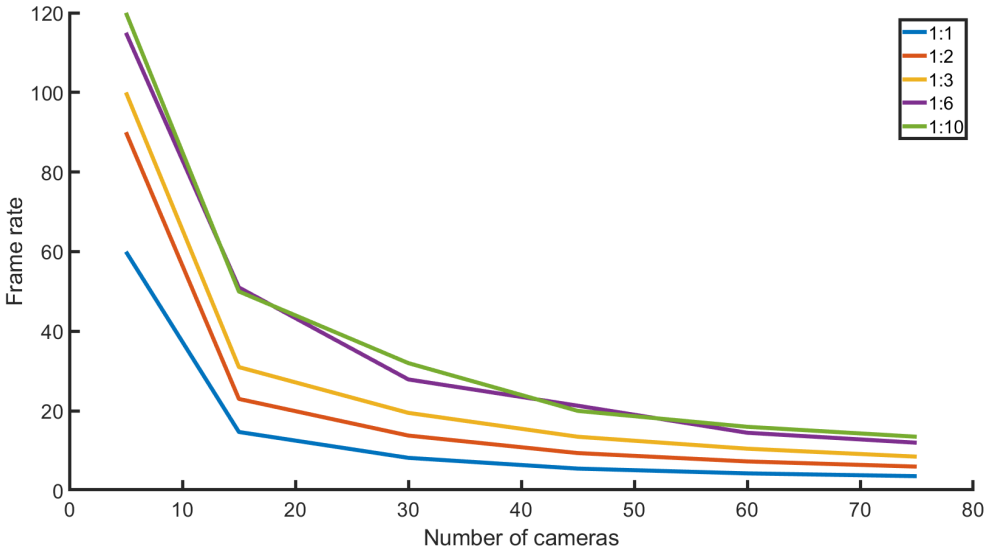

Figure 6.2: As the number of cameras are added to the scene the simulator frame rate decreases. By not rendering every camera on every frame the simulator frame rate increases. Each line represents a ratio for how often the camera renders, i.e. 1:2 means the camera frame is rendered every other game update.

6.1.3 Unity Profiler

This is just a quick overview of the profiling tool in Unity. This can be used to see how long a computation takes or how much resources it uses. Even though it is a bit small, most of the time whilst running is spent in the fixed update block (Figure 6.3).

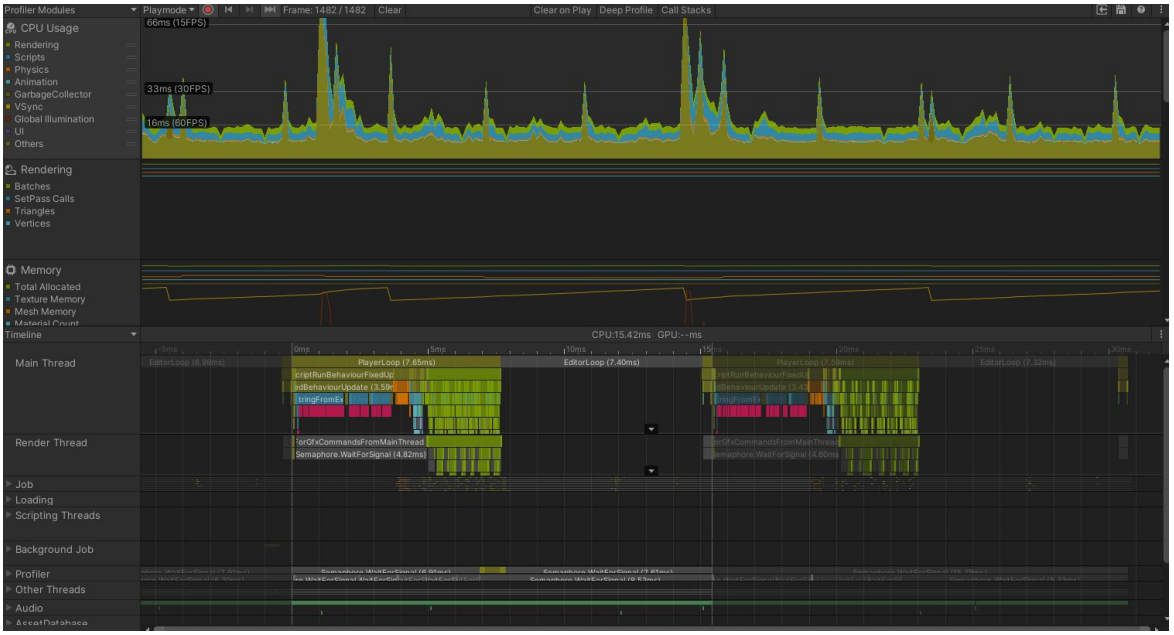

Figure 6.3: The Unity profiler allows the developer to see what the in the game engine how computationally a task is.

6.2 ML-Agents Performance

This section will look at the performance of autonomous vehicles as well as the tuning process. This section will be split into two parts. The first part is an environment where the vehicle could drive around by itself. The second part will look at the vehicles using collaboration learning.

Along with ML-Agent TensorBoard is imported. TensorBoard is a visual toolkit that allows displaying matrices such as the environment and loss from the learning.

6.2.1 Training using PPO

As mentioned in the implementation section, the two options that could be used for this section were PPO or SAC. As SAC is designed to run slower and make more complex decisions PPO would be the one to use.

To avoid too many plots only the most interesting runs have been extracted. It is also worth noting that the vehicle simulation is more of a proof of concept to prove what is possible with the simulator rather than a perfect solution.

Discrete vs Continuous

Figure 6.4 shows the cumulative reward per episode and episode length. It is clear that the light blue line reaches the reward target quickly whilst the other line stays at a negative value. This is using the simple map where the vehicles only had to drive in circles until it timed out. The hyperparameters are the same with the only difference being that the light blue model is producing discrete output values and the dark blue is producing continuous.

It can also be observed that there is a small dip at around 1M steps where after that the dark blue episode length started increasing. This was most likely caused by the vehicle not moving and the penalty received was the standard one received per time step.

It is quite clear from this plot that using discrete values worked better than continuous.

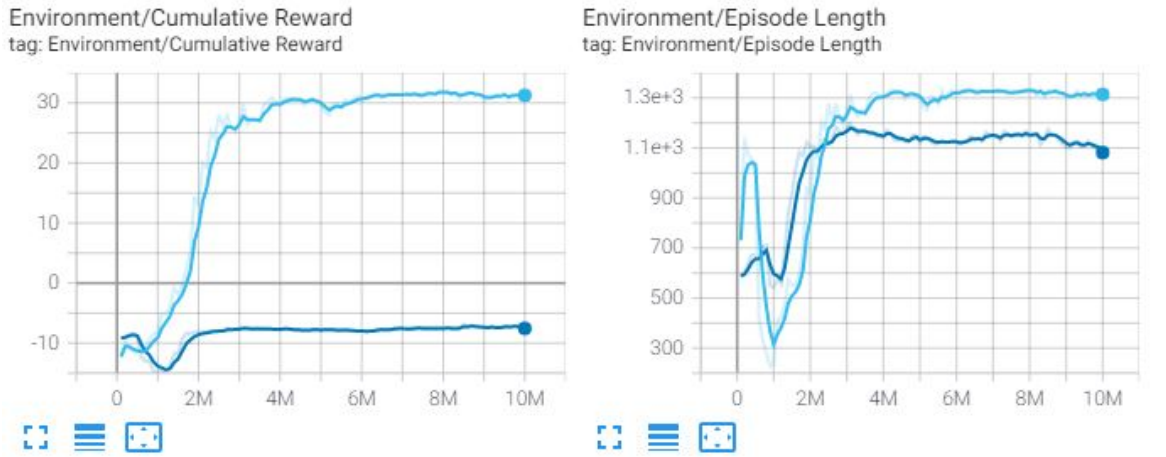

Figure 6.4: The light blue line indicates a discrete model output, whilst the dark blue shows the continuous. The other hyperparameters are the same in both runs.

Model depth

This part will look at the two alternatives to the network structure. ML-Agents recommend having between 2 to 3 hidden layers in the model. Figure 6.5 shows the cumulative reward and episode length on the simple map. The performance of the two models is very similar. However, training the deeper network took 30% longer time than when training the network with 2 layers. On the more complex map, the difference becomes more noteworthy. The wider shallower layer quickly adapts to reach the higher rewards. This could have something to do with the number of inputs. As it is quite large, having a shallower network distinguishes this better.

Other parameters tuned

There are several other parameters to tune. These include the number of network design of the GAIL model, learning rate, and policy variables. However, the models did not improve when these were tweaked.

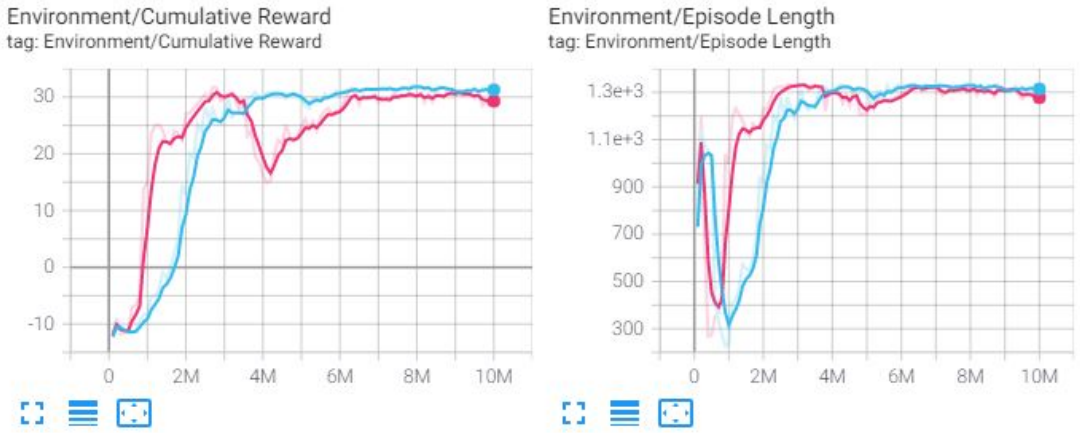

Figure 6.5: Pink line indicates model with 2 hidden layers and 128 neurons in each. Blue line indicates 3 hidden layers with 64 neurons in each. Same performance after not many iterations, but the deeper network took 30% longer.

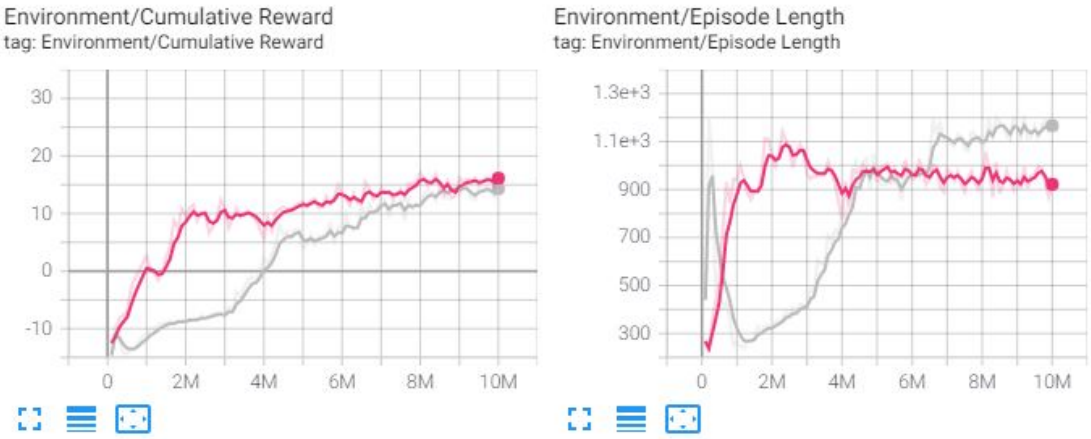

Figure 6.6: Pink line indicates model with 2 hidden layers and 256 neurons in each. Grey line indicates 3 hidden layers with 64 neurons in each.

6.2.2 Training using MA-POCA

When using collaboration learning there are two types of rewards, individual reward and team reward. As can be seen from Figure 6.7, The individual reward reached close to one. The maximum reward varies a bit depending on the starting point, but an optimal run could reach around 1.4. The group reward however flattened out around 1. Ideally, the group reward should be 2 as both vehicles would have reached their targets. Not much tuning was performed on the collaboration learning, but there were a few design changes in the game controller and map in the hope the vehicles would have to interact with each other at the intersection. The full results can be found here[2].

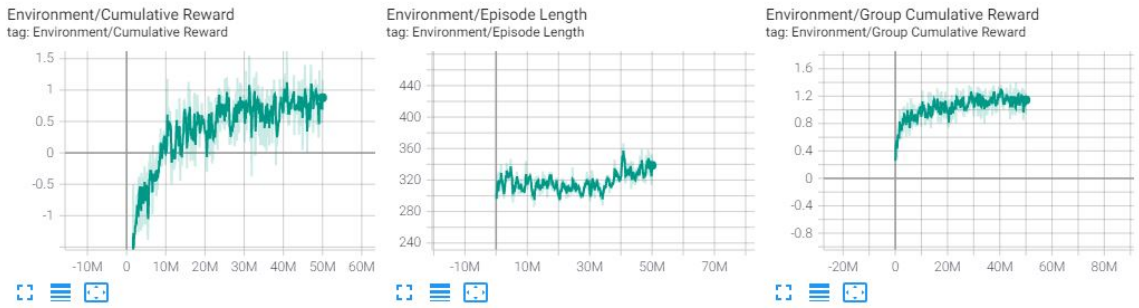

Figure 6.7: This was the optimal model for collaboration learning. The environment consisted of 2 vehicles.

[2] `https://github.com/tobhil98/FinalYearProject-TobyHillier`

Chapter 7

Evaluation

This chapter will look at what has been achieved and evaluate this against the requirement capture (Chapter 3).

The first part of this project was to analyse the existing simulator options. This was the main research part of the project. A surprisingly large number of simulators were applicable for this project, but eventually AirSim, Carla and Gazebo turned out to be the best options. Even though Carla had all the features this project was looking for Unreal Engine would not run on this Ubuntu set-up as the NividaGTX 780M graphics card could not handle Nvidia Vulkan drivers. AirSim was chosen over Gazebo as a traffic simulator were more applicable in regards to the requirements.

The other early decision that had to be made was which game engine to use. Unity was chosen for two reasons. Firstly, it is a simple platform with lots of additional libraries that could be used to further enhance the environment. Secondly, as the Unity build was only a prototype version, only the essential features had been implemented.

AirSim with Unity covers all of the requirements looked at in Section 3.1.

- **Usability:** The simulator is simple to build by only running one script and then importing the model into Unity. Also, as mentioned before, adding new models is a straightforward process.
- **Extensibility:** The simulator is proven to be extensible as most of the features requested have been added to the simulator.
- **OS:** The simulator has been run on both Windows and Ubuntu. Having a cross-platform simulator is very useful as it makes working on different computers much easier.
- **Game Engine:** The simulator allows for several backends, but Unity was the one chosen.
- **Development:** Updates to AirSim have been added from the main repository whilst the project was worked on. Currently, there are breaking changes in a pull request to this project which have not been merged.
- **Support:** The support was not needed, but this was a great place for finding how to implement for example the APIs.

The simulator covers all of the requirements well.

The next part was to implement the missing features.

- **Multiple Agents:** The simulator can easily handle multiple agents in the current design.
- **Variety of Agents:** Several different kinds of vehicles were added such as trucks and sports cars.
- **Free movement:** The agents are completely free and are not affected by any other rules than physical ones. This means that the cars can drive on the pavement, but they would not be able to drive through other cars or buildings.
- **Customisable environments:** The difficulty of adding custom maps depend on what kind of map the user wants. The simplest option is to drag models straight into the simulator whilst the textured maps require a bit more work.
- **Controlling APIs:** This is probably where the project has more potential. Currently, only APIs which directly control the entity are in place. This means that the controller can request the entity to go forward or turn left. There is no API in place yet which allows users to say go forward for 10 units. Ideally, the vehicle models could be added to these APIs to let the cars drive around autonomously. The models are however not accurate enough for that yet.
- **Perception APIs:** The simulator has two APIs available for perception. One is a camera that can have several different modes, and the other is a LIDAR sensor. The LIDAR sensor is quite primitive at the moment as it only returns results from fixed points which currently are not customisable.

Overall this project has created an open environment where mixed forms of traffic can interact with each other. Choosing AirSim with Unity has turned out to be a great decision. Even though the codebase was quite large, it was a lot easier to understand than the code for Unreal Engine. As Unity was a plugin it allowed for additional tools to be used such as ML-Agents and the Map SDKs. This has allowed for some very complex features which would not have been possible if they had to be implemented from scratch.

Chapter 8

Conclusions

Overall this project successfully implemented a traffic simulator that could simulate mixed traffic as required. AirSim was selected as the simulator to extend over many others. Most of the missing features were then added to the simulator. AirSim had a big diverge from the original version when the server was divided into 3 parts. This was done to allow for future possibilities where each vehicle would have its server. This split also made it simple to add completely new types of vehicles with a different set of controls and APIs.

The next big part of the project was to train the ML agents. The vehicles are far from perfect, but they have definitely learnt to follow limit contact with the wall and drive towards the target.

For this project, most of the time was spent trying to get used to tools and programs. It took a long time at the starts to get an understanding of how the AirLib code works as there was no guide over the code architecture. Trying to understand the code base whilst having to do other unrelated work made it harder. Also, getting used to tools like Unity and Blender took time. Some time was used customising the environments in Blender to make them look better which could be used in a demonstration.

The biggest coding challenge was dividing the server. This meant that a lot of code had to be understood, moved or modified. This was important to get working as making the simulator more easily extendable was a key part.

A lot of time was spent researching and understanding GAIL and collaboration learning for the ML-Agents part of the project. Most evening several models have been training on lab computers. Due to slow internet and not being able to go into uni, made it very difficult to fix bugs, as uploading the build took several hours, and the environment could not be trained on this computer.

Overall this was an ambitious project that has tried a lot of different tools and features and combined them into one simulator. For a perfect system, more APIs could have been added to control the vehicles and the ML agents could have had better performance. However, this project was not looking for an optimal solution, but rather a prototype consisting of a large range of features.

Chapter 9

Further Work

This chapter consists of two sections. The first section will look at what needs to be done to the existing simulator to improve the quality. The second section will look at future use cases for the simulator.

9.1 Simulator extension

The first thing would be to further improve the ML-Agent model. The existing model can drive around, but not in a reliable way. More training and further parameter tuning would be needed. Also, training the pedestrians so that the interaction between pedestrian and vehicles can be simulated. Currently, the interaction is only handled through the APIs.

The next step would be to add more controlling APIs to the simulator. Being able to give the pedestrians more relaxed information such as walk forward but pause if something is in front would be very beneficial. This would be the same for the vehicles. Currently, the simulator is only a platform where all of these things have to be manually controlled or controlled through external programs using the APIs.

Another big improvement would be to optimise the video stream even further. This could potentially be done by using UDP streaming. However, as the image capturing has to be done on the main thread, this could be an unsolvable problem until Unity upgrades its game engine.

Lastly, there are a lot more features in AirLib which have not yet been added to AirSim. More advanced weather, additional sensors and several APIs that exist in Unreal Engine but not in Unity could be added.

9.2 Use cases

One option would be to try to use the simulator alongside an external sensor, like a traffic camera, to create a digital twin of the real environment. There are several use cases for this. Firstly, it could detect dangerous driving and report it to the police. This could both make people drive more safely as they know they are being surveyed, as well as making it possible to stop dangerous driving early on. It could also be used to track dangerous traffic junctions to monitor and detect near-collisions.

Another option could be to simulate an autonomous wheelchair that exists in the Imperial Robotics Lab. By modelling the wheelchair in AirSim we could try to create an autonomous system and then compare it to the behaviour in real life.

A third example could be to try to train a machine learning model for an autonomous robot so that it could navigate around crowded places with cars and pedestrians [26].

Bibliography

[1] J. Markoff, "Google cars drive themselves, in traffic," Oct 2010. [Online]. Available: https://www.nytimes.com/2010/10/10/science/10google.html

[2] Tesla. (2021) Autopilot. [Online]. Available: https://www.tesla.com/autopilot

[3] I. Millington, **Game physics engine development**. CRC Press, 2007. [Online]. Available: https://books.google.no/books?id=d0NZDwAAQBAJ&lpg=PP1&ots=2LlCAMdy58&dq=Physics%20engine&lr&hl=no&pg=PP1#v=onepage&q=Physics%20engine&f=false

[4] Yıldırım, Şahin and Arslan, Erdem, "Ode (open dynamics engine) based stability control algorithm for six legged robot," **Measurement : journal of the International Measurement Confederation**, vol. 124, pp. 367–377, 2018.

[5] J. C. Martinez-Franco and D. Alvarez-Martinez, "Physx as a middleware for dynamic simulations in the container loading problem," in **2018 Winter Simulation Conference (WSC)**. IEEE, 2018, pp. 2933–2940.

[6] B. Nicoll and B. Keogh, **The Unity Game Engine and the Circuits of Cultural Software**. Cham: Springer International Publishing, 2019, pp. 1–21. [Online]. Available: https://doi.org/10.1007/978-3-030-25012-6_1

[7] G. Dotan. (2015, Jan) Top 10 Unity Games Ever Made. [Online]. Available: https://blog.soomla.com/2015/01/top-10-unity-games-ever-made.html

[8] J. Drake. (2020, Jun) 15 Great Games That Use The Unreal 4 Game Engine. [Online]. Available: https://www.thegamer.com/great-games-use-unreal-4-game-engine/

[9] A. Kumar, **Introduction to the Software**. Berkeley, CA: Apress, 2020, pp. 9–13. [Online]. Available: https://doi.org/10.1007/978-1-4842-6077-7_2

[10] Epic Games. (2020, Jun) A first look at Unreal Engine 5. [Online]. Available: https://www.unrealengine.com/en-US/blog/a-first-look-at-unreal-engine-5

[11] ——, "Broadcast Live Events." [Online]. Available: https://www.unrealengine.com/en-US/industry/broadcast-live-events

[12] ——, "Automotive & Transportation." [Online]. Available: https://www.unrealengine.com/en-US/industry/automotive-transportation

[13] Arnia Software, "What Makes Unity So Popular in Game Development?" [Online]. Available: https://www.arnia.com/what-makes-unity-so-popular-in-game-development/

[14] Technologies, Unity, "Glossary." [Online]. Available: https://docs.unity3d.com/2020.1/Documentation/Manual/Glossary.html

[15] E. T. M. Guevarra, **Modeling and Animation Using Blender: Blender 2.80: The Rise of Eevee**. Apress, 2019.

[16] E. T. Mendoza Guevarra, **Creating Game Environments in Blender 3D: Learn to Create Low Poly Game Environments**. Berkeley, CA: Apress L. P, 2020.

[17] S. A. Bagloee, M. Tavana, M. Asadi, and T. Oliver, "Autonomous vehicles: challenges, opportunities, and future implications for transportation policies," **Journal of Modern Transportation**, vol. 24, no. 4, pp. 284–303, Dec 2016. [Online]. Available: https://doi.org/10.1007/s40534-016-0117-3

[18] B. S. Kerner, "Effect of autonomous driving on traffic breakdown in mixed traffic flow: A comparison of classical acc with three-traffic-phase-acc (tpacc)," **Physica A**, vol. 562, 2021.

[19] J. Wulf and I. Blohm, "Fostering value creation with digital platforms: A unified theory of the application programming interface design," **Journal of management information systems**, vol. 37, no. 1, pp. 251–281, 2020.

[20] **Digital Twin Technologies and Smart Cities**, 1st ed., ser. Internet of Things, Technology, Communications and Computing. Cham: Springer International Publishing, 2020.

[21] Carla. (2021, Jun) Tutorial - Add custom map. [Online]. Available: https://carla.readthedocs.io/en/latest/tuto_A_add_map

[22] A. Šmíd, "Comparison of unity and unreal engine," **Czech Technical University in Prague**, pp. 41–61, 2017.

[23] Unity-Technologies, "Ml-agents-overview," Apr 2021. [Online]. Available: https://github.com/Unity-Technologies/ml-agents/blob/main/docs/ML-Agents-Overview.md

[24] J. N. Foerster, N. Nardelli, G. Farquhar, P. H. S. Torr, P. Kohli, and S. Whiteson, "Stabilising experience replay for deep multi-agent reinforcement learning," **CoRR**, vol. abs/1702.08887, 2017. [Online]. Available: http://arxiv.org/abs/1702.08887

[25] (2021) Maps SDK for Unity Overview. [Online]. Available: https://developers.google.com/maps/documentation/gaming/overview_musk#supported_platforms

[26] Q. Chao, Z. Deng, and X. Jin, "Vehicle–pedestrian interaction for mixed traffic simulation," **Computer animation and virtual worlds**, vol. 26, no. 3-4, pp. 405–412, 2015.

[27] 4D-Virtualiz. (2021, Jan) 4d virtualiz - robotics simulator. [Online]. Available: https://www.4d-virtualiz.com/robotics-simulator

[28] Air Sim. (2021, Jan) Welcome to AirSim. [Online]. Available: https://microsoft.github.io/AirSim

[29] ApolloAuto. (2021, Jan) Github - Apollo Simulator. [Online]. Available: https://github.com/ApolloAuto/apollo

[30] The Autoware Foundation. (2021, Jan) Gitlab - Autoware Documentation. [Online]. Available: https://autowarefoundation.gitlab.io/autoware.auto/AutowareAuto

[31] A. Dosovitskiy, G. Ros, F. Codevilla, A. Lopez, and V. Koltun, "CARLA: An open urban driving simulator," in **Proceedings of the 1st Annual Conference on Robot Learning**, 2017, pp. 1–16.

[32] Voyage Deepdrive. (2021, Jan) DeepDrive Website. [Online]. Available: https://deepdrive.voyage.auto/

[33] Open Source Robotics Foundation. (2014) Gazebo Website. [Online]. Available: http://gazebosim.org

[34] Der, Ralf and Martius, Georg. (2021, Jan) LPZRobots Github. [Online]. Available: https://github.com/georgmartius/lpzrobots

[35] R. Der and G. Martius, **The LpzRobots Simulator**. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, pp. 293–308. [Online]. Available: https://doi.org/10.1007/978-3-642-20253-7_16

[36] LG Electronics America RD Center. (2021, Jan) LGSVL Simulator. [Online]. Available: https://www.lgsvlsimulator.com/about

[37] AnyKode. (2019) Marilou Simulator. [Online]. Available: http://www.anykode.com/index.php

[38] rFpro. (2021, Jan) Driving Simulation. [Online]. Available: https://www.rfpro.com/driving-simulation

[39] ——. (2021, Jan) Supervised Learning Autonomous Driving. [Online]. Available: https://www.rfpro.com/virtual-test/dlad-deep-learning-autonomous-driving

[40] Image Space Inc. (2021) ISI Motor Software Engine. [Online]. Available: https://imagespaceinc.com

[41] The TORCS racing board. (2017) TORCS Competition. [Online]. Available: http://www.berniw.org/trb/events/eventlist.php

[42] Cyberbotics. (2021, Jan) Webots Github. [Online]. Available: https://github.com/cyberbotics/webots/tree/released

[43] ——. (2021, Jan) Cyberbotics. [Online]. Available: https://cyberbotics.com

[44] R. C. Arkin, "Ethics and autonomous systems: Perils and promises [point of view]," **Proceedings of the IEEE**, vol. 104, no. 10, pp. 1779–1781, 2016.

[45] J. Borenstein, J. R. Herkert, and K. W. Miller, "Self-driving cars and engineering ethics: The need for a system level analysis," **Science and engineering ethics**, vol. 25, no. 2, pp. 383–398, 2019.

[46] A. Hevelke and J. Nida-Rümelin, "Responsibility for crashes of autonomous vehicles: An ethical analysis," **Science and Engineering Ethics**, vol. 21, no. 3, pp. 619–630, Jun 2015. [Online]. Available: https://doi.org/10.1007/s11948-014-9565-5

[47] A. Martinho, N. Herber, M. Kroesen, and C. Chorus, "Ethical issues in focus by the autonomous vehicles industry," **Transport Reviews**, vol. 0, no. 0, pp. 1–22, 2021. [Online]. Available: https://doi.org/10.1080/01441647.2020.1862355

[48] K. Miller, "Moral responsibility for computing artifacts: The rules," **IT Professional**, vol. 13, pp. 57 – 59, 07 2011.

[49] Open Source Initiative. (2020, Jun) The MIT License. [Online]. Available: https://opensource.org/licenses/MIT

[50] UK Gov. (2018) Automated and Electric Vehicles Act 2018. [Online]. Available: https://www.legislation.gov.uk/ukpga/2018/18/part/1/enacted

[51] ——. (2016, Oct) Robotics and artificial intelligence. [Online]. Available: https://publications.parliament.uk/pa/cm201617/cmselect/cmsctech/145/14506.htm

Appendices

Appendix A

Background Theory

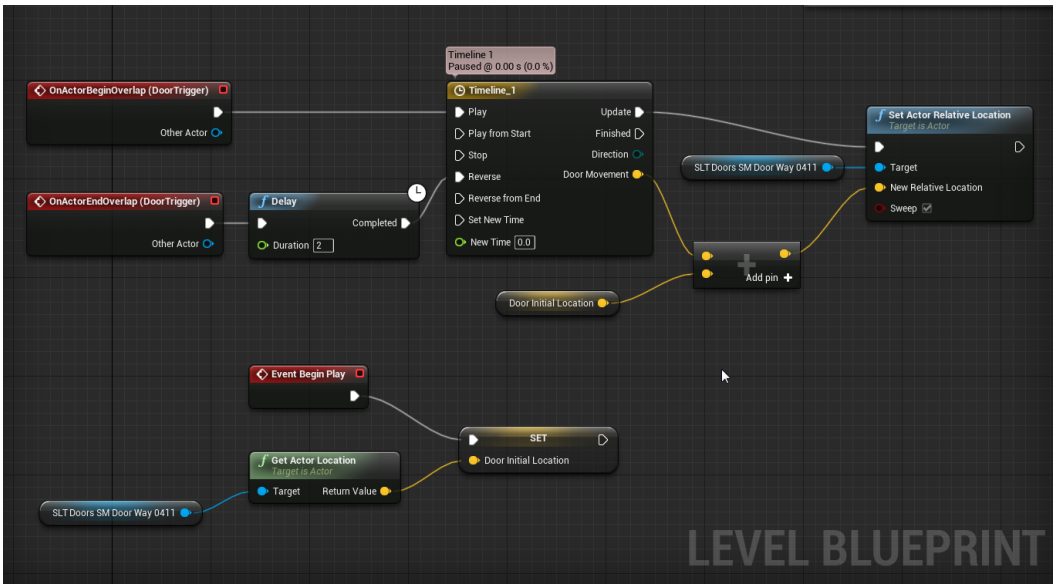

Figure A.1: Blueprints in Unreal Engine allows users to drag and drop blocks rather than having to program.
`https://docs.unrealengine.com/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Timelines/Examples/OpeningDoors/index.html`

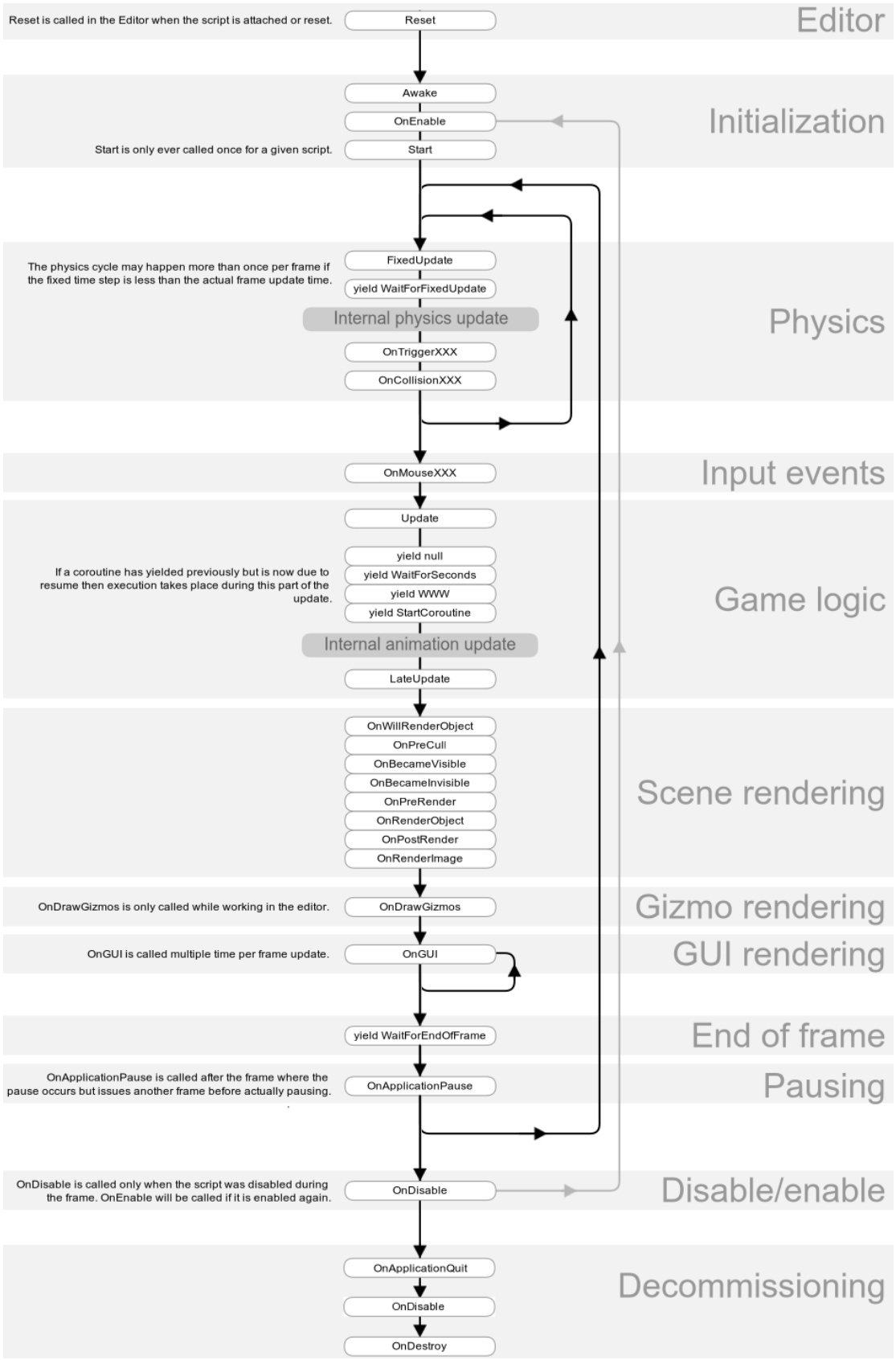

Figure A.2: Flow chart of the actions in monobehavior. Fetched from Unity website.

Appendix B

Simulator Research

B.0.1 4DV-Sim

Description: 4DV-Sim[1] is a simulator that is designed to emulate the hardware and sensors in autonomous systems. This is a professional product and has a variety of use cases from simulating farming to military equipment [27].

Open Source: No

Operating System: Linux

Game Engine: Non, but it does use PhysX for the physics engine

Pros: The simulator has a lot of available APIs. The simulator also comes with a configurable GUI to set up the simulation environment how you would like it. Also, as it is professionally made, it looks very good.

Cons: It is not designed to train machine learning implementations on the simulator, but rather emulate a current hardware setup. Also, as it is not open source, it will not be something that we could modify or expand upon to suit our purposes.

Conclusion: As 4DV-Sim is not an open-source product it is not something that can be used for this project. It is however interesting to see that simulators like this are needed not just for research purposes, but for customers who want to try their hardware setup in an emulated environment.

[1] Website: `https://www.4d-virtualiz.com/en/automotive-simulator`

Figure B.1: 4DV-Sim has most of the features this project is looking for, but not open-source so cannot be used.
`https://www.4d-virtualiz.com/en/automotive-simulator`

B.0.2 AirSim

Description: AirSim[2] is a simulator for cars and drones. It is open-source and works as a plugin for Unreal Engine, which means the simulator can be used with any environment which has been modeled inside the game engine. According to their website [28], the goal of the simulator is to create a platform for AI research to experiment with deep learning, computer vision, and reinforcement learning algorithms for autonomous systems.

Open Source: Yes

Operating System: Any operating system

Game Engine: Primarily Unreal Engine, but it also offers a prototype version in Unity

Pros: Offers a large range of existing APIs. The simulator also has an active community on both discord and Github. It also gives the option to add drones. It is also designed to train machine learning models on it.

Cons: The simulator is not as realistic as other simulators. The vehicle physics is not as good as some of the other simulators, for example the handling and collisions. Also, currently, there are no pedestrians in the game.

Conclusion: AirSim is worth looking closer into. As it is built using a game engine it should not be too hard to add the missing features, like for example adding and controlling pedestrians. In addition, as it is a plugin for Unreal Engine means that we can use other tools to import for example maps. Also, realistic vehicle physics was determined not to be an important factor for this project.

[2]Website: `https://microsoft.github.io/AirSim`

Figure B.2: AirSim is missing pedestrians and a few other features. However, the simulator is very flexible and adding new features should not be too difficult.
`https://microsoft.github.io/AirSim`

B.0.3 Apollo

Description: Apollo[3] is a simulator that is designed to emulate the hardware in autonomous vehicles so that it can be trained for machine learning models. According to their website [29], Apollo is a flexible architecture that accelerates the development and testing of autonomous vehicles.

Open Source: Yes

Operating System: Any system that can run Docker

Game Engine: Unity

Pros: Accurately models the vehicle physics to help improve the machine learning model's accuracy. The simulator is also actively being worked on by a large community.

Cons: It looks like quite a complex simulator, and it does therefore not seem like it will be easy to modify. The product is really specific towards training autonomous vehicles.

Conclusion: Due to the complexity of this simulator, it does not look like something that can easily be extended for this project.

[3] `https://github.com/ApolloAuto/apollo`

Figure B.3: Apollo has realistic models and is designed for training machine learning models. However, due to its complexity it looks too challenging to adapt for this project.
Slide deck from Apollo Game Engine Based Simulation Talk at GDC 2019 - `https://bit.ly/2VSzwlF`

B.0.4 Autoware

Description: Autoware[4] is an open-source software for autonomous vehicles. It comes with a large variety of APIs [30]. Autoware is however not a simulator, but can be used on a simulated vehicle to make it autonomous.

Open Source: Yes

Operating System: Robot Operating System (ROS)

Game Engine: Na

Pros: As it runs on ROS it can easily be adapted to work on a real autonomous vehicle.

Cons: Currently it only works with a specific car model and sensor set up. The software also seems quite complex, and combining it with a simulator will probably be quite challenging.

Conclusion: As this is not a simulator this is not something that we can use for this project. We will see with the LGSVL simulator (B.0.12), this software can be used alongside a simulator to model the autonomous system.

B.0.5 Carla

Description: Carla[5] is an open-source simulator for developing autonomous vehicles. It contains a variety of APIs and is actively being developed. Carla is also designed for training machine learning models [31].

Open Source: Yes

Operating System: Primarily Linux, but also Windows

Game Engine: Unreal Engine

[4] `https://gitlab.com/autowarefoundation/autoware.auto/AutowareAuto`
[5] `https://carla.org/`

Pros: Has a lot of features already implemented, such as sensors, vehicle API, and the ability to add new objects. The simulator also has the ability to add pedestrians and plot their movement. Active community. Well documented and lots of information online.

Cons: Difficult to add new and custom maps. Vehicle handling is not as realistic as some of the other simulators.

Conclusion: Carla is worth looking into further as the simulator already has most of the features the project is looking for.

Figure B.4: Carla is a simulator designed for mixed traffic simulation and covers most of the features the project is looking for.
`https://www.unrealengine.com/en-US/spotlights/carla-democratizes-autonomous-vehicle-r-d-with-free-open-source-simulator`

B.0.6 CoppeliaSim

Description: CoppeliaSim[6] is an open-source simulator designed for education and robotics research. The simulator is actively maintained by Coppelia Robotics, based in Switzerland. CoppeliaSim is primarily designed for different robots, such as pickandplace machines and other human interactable robots.

Open Source: CoppeliaSim itself is open source. However, it uses geometric plugin which can be used to calculate distances and detect object collisions. The licences for this plugin are free for educational purposes[7].

Operating System: Any operating system. The projects are cross platform.

Game Engine: Supports several different physics engines. Bullet, ODE, Vortex Studio and Newton Dynamics.

Pros: The simulator has a lot of existing APIs which can be used to interact with the simulator. Offers a variety of physics engines to improve the simulator accuracy. The simulator also lets users create accurate robotics models.

Cons: This simulator is not directly designed for traffic like some of the other simulators. It also looks difficult to change the simulator itself.

[6] `https://www.coppeliarobotics.com/`
[7] `https://www.coppeliarobotics.com/helpFiles/en/licensing.htm`

Conclusion: CoppeliaSim is not a simulator worth considering for this project as it seems to difficult to add custom features and APIs to the simulator itself.

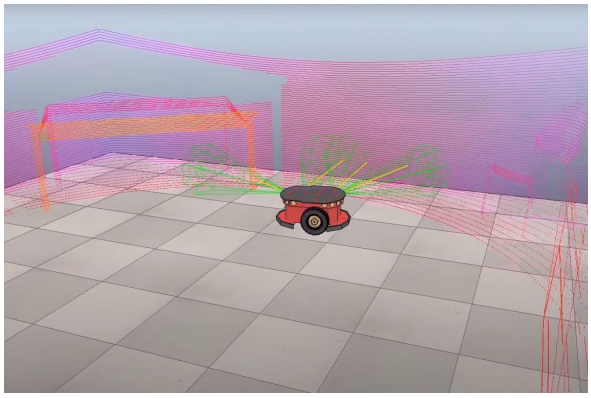

Figure B.5: CoppeliaSim is primarily designed for robotics simulation. Does not focus on cars and traffic as much as other simulators
`https://www.youtube.com/watch?v=MrwU6wJNTAc&t=23s`

B.0.7 CrowdSim3D

Description: CrowdSim3D[8] is primarily a simulator for modeling large crowds, but can also be used for vehicle traffic.

Open Source: No (£180)

Operating System: Any operating system

Game Engine: Not specified

Pros: Has the ability to control several pedestrians and vehicles in a shared space.

Cons: Does not look to be designed for machine learning models. Difficult to add new models. Also, as it is not open source it will not be possible to customise the product.

Conclusion: CrowdSim3D is not worth considering as the product cannot be adapted as it is not open source.

[8] `https://crowdsim3d.com`

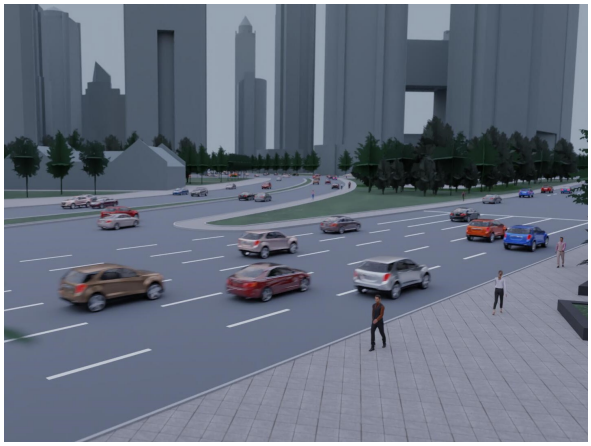

Figure B.6: As CrowdSim3D is not open source, the simulator cannot be modified to add missing features.

`https://crowdsim3d.com`

B.0.8 Deep Drive

Description: Deep Drive[9] is an open-source simulator aimed to train neural networks for self-driving cars [32]. They also provide you with a large data set to train your autonomous vehicle on.

Open Source: Yes

Operating System: Any operating system

Game Engine: Unreal Engine

Pros: Is able to handle a variety of different sensors and vehicle setups. It also has an active community and a leader board where you can compare your trained neural network against other developers.

Cons: Currently there are three maps available, but it is not easy to add your own maps. Also, it is not designed to be customised. Currently, there are no pedestrians and no APIs.

Conclusion: Deep Drive is not a simulator that is worth looking at as it is not designed to be customised. It also lacks most of the features looked for.

[9] `https://deepdrive.voyage.auto/`

Figure B.7: Deep Drive is mainly aimed at training machine learning models. The simulator also has no existing APIs making it difficult to add some from scratch.
`https://deepdrive.voyage.auto`

B.0.9 Donkey Car Simulator

Description: Donkey Car Simulator[10] is a simulator for the Donkey Car. The car itself costs roughly £200 and can be ordered online, or you can download the schematics for free. The simulator can be used to train a neural network in Python which can then run on your car.

Open Source: Yes

Operating System: Any operating system

Game Engine: Unity

Pros: Easy to use

Cons: Not what we are looking for as it is only used to train a real toy car.

Conclusion: Donkey Car is an interesting project to read about as it is a fun kit that introduces people to autonomous cars. However, it is not a simulator that would be worth extending for this project as the simulator is not customisable and there is only one vehicle.

Figure B.8: Donkey Car Simulator allows the user to create a digital twin to train a machine learning model that can be applied to the physical car.
`https://docs.donkeycar.com`

[10] `https://docs.donkeycar.com/guide/simulator`

B.0.10 Gazebo

Description: Gazebo[11] is a robotics simulator designed to design robots, test algorithms and train AI systems [33]. Gazebo offers the ability to simulate several robots at once in both indoor and outdoor environments.

Open Source: Yes

Operating System: Any operating system

Game Engine: ODE, but also Bullet, Simbody and DART which are additional physics engines.

Pros: Very versatile, can be used for much more than traffic simulations.

Cons: Due to the large number of different physics engines it might be complicated to fix something if a feature breaks. The documentations is not as clear as some other simulators.

Conclusion: Gazebo is worth looking at as it is widely used for robotics simulations. All the sensing APIs are already implemented.

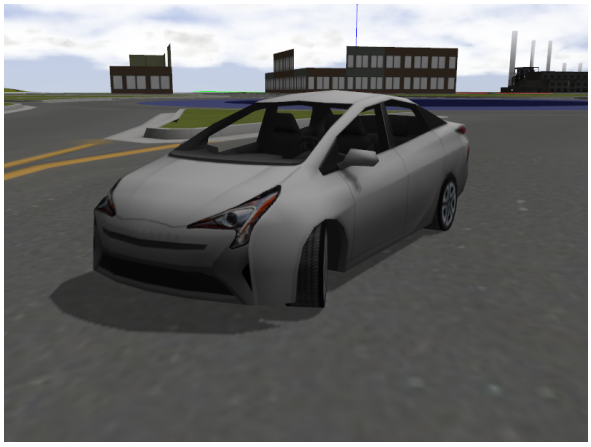

Figure B.9: Gazebo is a robotics simulator that can be used for autonomous vehicles. A very versatile simulator with a large variety of use cases.
`http://gazebosim.org/blog/vehicle%20simulation`

B.0.11 LPZRobots

Description: LPZRobots[12] is a robotics simulator created by the Robotics Group for Self-Organisation of Control at the University of Leipzig in Germany [34, 35]. The simulator aims to have an open environment to simulate the robot's physics.

Open Source: Yes

Operating System: Primarily Linux, but now also supports Windows

Game Engine: ODE

Pros: Free to add any kind of robot.

[11] `http://gazebosim.org`
[12] `https://github.com/georgmartius/lpzrobots`

Cons: The code has not been updated since 2018 and the last release was in 2016. Unlike other simulators, this one does not seem to have an active community.

Conclusion: As the development of the LPZRobots simulator has been inactive for several years, makes it not worth considering. The simulator also lacks most of the features this project is looking for.

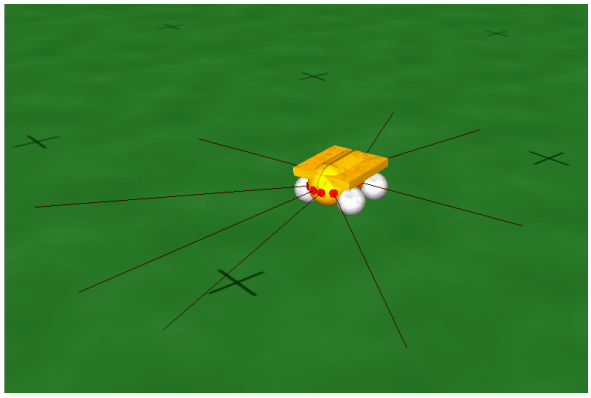

Figure B.10: LPZRobots is an open platform robotics simulator. This simulator has not had an update since 2018 however.

`https://itp.uni-frankfurt.de/~gros/StudentProjects/Robots_2016_ObstacleAvoidance`

B.0.12 LGSVL Simulator

Description: LGSVL[13] is a simulator created by the Advanced Platform Lab at the LG Electronics America R&D Center [36]. The simulator combines the vehicle from Autoware (Section B.0.4) with Apollo (Section B.0.3) which emulates the hardware. The LGSVL has lots of available APIs such as cameras, LiDAR, RADAR, and GPS. Some environmental parameters can also be changed such as the map, weather, and pedestrians.

Open Source: Yes

Operating System: Windows 10

Game Engine: A variety due to its complexness, but both Unreal Engine and Unity.

Pros: Has most of the features we are looking for.

Cons: Relies on a lot of different components. The codebase is therefore large and complex and it looks difficult to add our own features such as our own entity.

Conclusion: The LGSVL Simulator would probably not work for this project as the code-base seems too large and complex making it difficult to extend.

[13] `https://github.com/lgsvl/simulator`

Figure B.11: LGSVL is a very complex simulator using several different game engines. However, most of the features this project is looking for are implemented
`https://www.lgsvlsimulator.com/about`

B.0.13 Marilou

Description: Marilou[14] is a simulator created by ANYKODE [37]. Marilou is an open map simulator where the user can add objects and hindrances for the robot to navigate around. The simulator is designed to simulate simultaneous localisation and mapping (SLAM) and other localisation techniques.

Open Source: No (£350)

Operating System: Windows and Linux

Game Engine: Unknown

Pros: Accurately simulates sensors and robot behavior. Easy to add new objects and other controllable entities.

Cons: As the simulator is not open source, we cannot modify its behavior. Latest release 2018.

Conclusion: Marilou is not a simulator that can be used for this project as it is not open source. In addition, it is not designed with APIs in mind and controlling multiple objects at once looks impossible in the current program.

[14] `http://www.anykode.com/index.php`

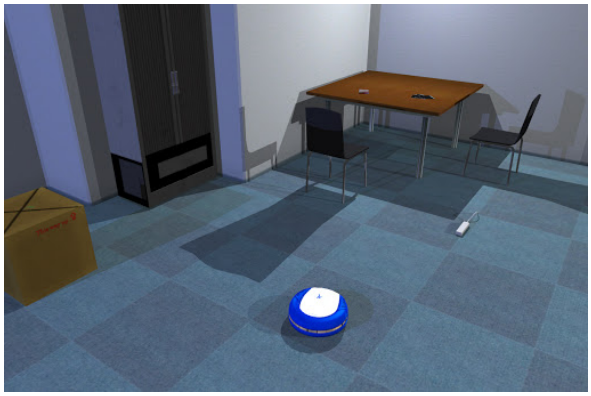

Figure B.12: Marliou is not able to control multiple entities as once, and as the simulator is not open source, this is not something that can be added.
`http://www.anykode.com/downloads.php`

B.0.14 rFpro

Description: rFpro[15] is a driving simulation software which focuses on road-vehicle simulation [38]. rFpro allows for a variety of use cases, from training machine learning models for autonomous driving [39], to motor racing.

Open Source: No

Operating System: Windows

Game Engine: ISIMotor - A game engine created by Image Space Inc [40]. The game engine is used for F1 and other racing games.

Pros: A professionally made simulator that has most of the features we are looking for. Very good graphics and accurate vehicle behavior.

Cons: rFpro does not give us access to any of the source code. The price is only available upon request, but it is most likely too expensive for this project.

Conclusion: Even though rFpro contains just about all of the features we are looking for, and is probably the best-looking simulator, it will not work for this project as it is not open source.

[15] `https://www.rfpro.com`

Figure B.13: rFPro contains most of the features this project is looking for. However, the project is not open source, and can therefore not be extended.
http:https://www.rfpro.com/driving-simulation

B.0.15 Rigs of Rods

Description: Rigs of Rods[16] (RoR) is an open-source physics simulator primarily designed to simulate vehicle physics. The simulator uses soft-body physics which means that if the vehicle collides, its structure will be deformed. This will result in a more accurate simulation.

Open Source: Yes

Operating System: Windows and Linux

Game Engine: Non, creates its own soft-body physics engine.

Pros: There is an active community creating modifications for the simulator. Also, the only simulator on the list which uses soft-body physics.

Cons: There are no APIs currently available.

Conclusion: As for this project, realistic features is not something that is required, but rather certain APIs. This simulator will require too much work to add the missing features. It is therefore not a simulator that is worth considering.

Figure B.14: Rigs of Rods is the only soft-body simulator looked at. Currently there are no implemented APIs making it difficult to use for this project.
https://docs.rigsofrods.org/gameplay/beginners-guide

[16] https://www.rigsofrods.org

B.0.16 TORCS - The Open Racing Car Simulator

Description: TORCS[17] is an open-source racing car simulator. It can be used as an ordinary racing game or as an AI racing research platform. The creators of TORCS used to host competitions on its website among players for who could create the best artificially intelligent racing car [41].

Open Source: Yes

Operating System: Linux and Windows

Game Engine: Non, implemented from scratch.

Pros: Easy to add and create new content.

Cons: Latest release 2016 but mainly developed in 2008. Also, has no available APIs and no pedestrians.

Conclusion: This simulator is lacking most of the features this project is looking for. It is also quite outdated. TORCS is therefore not worth considering.

Figure B.15: TORCS is a simple simulator, but has not been updated since 2016. There are also a lot of missing features such as pedestrians.
`https://sourceforge.net/projects/torcs`

B.0.17 Webots

Description: Webots[18] is an open-source robotics simulator developed by Cyberbotics [42]. Webots provides a complete developing environment to model, program and simulate the robots [43].

Open Source: Yes

Operating System: Any operating system

Game Engine: ODE

Pros: A lot of documentation on how to add new features. Seems to be able to generate new maps easily. Available chat page.

[17] `https://sourceforge.net/projects/torcs`
[18] `https://www.cyberbotics.com`

Cons: There does not seem to be many APIs to control multiple entities. All the APIs are mainly used for sensing.

Conclusion: Webots has most of the features that this project is looking for, however seems to lack the ability to easily control multiple entities at once. The simulator is also less advanced than Gazebo (Section B.0.10), which could make it easier to add missing features in Webots. This simulator is therefore worth considering.

Figure B.16: Webots is an open area robotics simulator which can simulate vehicles and pedestrians. However the simulator is currently lacking implemented APIs
`https://www.cyberbotics.com/doc/automobile/city-night`

Appendix C

Ethical, Legal and Safety Considerations

C.1 Ethical Considerations

As this is just a simulation, there are not really any ethical concerns as such.

If we however decide to use the knowledge we learn in our simulator on a physical system there are a few things worth thinking about. Ethical considerations for autonomous systems have been a discussion for a long time [44, 45]. There are a variety of different ethical concerns. To name just a few, will an autonomous system be responsible enough [45], who would be responsible for an accident involving a self-driving car [46, 47], and how could autonomous vehicles impact peoples behavior [48]. These are not ethical concerns for the project as is, but could become an issue in the future.

C.2 Legal Considerations

AirSim has an MIT license which means that we can use the simulator however we like [49]. In regards to the game engine, as we have chosen to use AirSim which uses Unreal Engine, we are free to distribute the simulator as long as we do not make a gross profit of more than $100k[1].

It could also be worth considering what the UK rules for autonomous vehicles are[2], if in the future work we decide to introduce a physical system [50, 51].

C.3 Safety Considerations

In regards to the project itself, there are no safety issues as it is a simulator being worked on from home.

[1] https://store.unity.com/compare-plans

[2] https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/929352/innovation-is-great-connected-and-automated-vehicles-booklet.pdf

As mentioned in the other two sections. If information from the simulator ever gets used in a physical product there are a few things worth considering. Firstly, it is important to remember these are only simulations. Training a machine learning model on the simulator will not accurately reflect the behavior in real life. Secondly, it is important when testing a physical system that it does not for example collide with someone. This could be prevented by driving slowly or not driving where other people or objects might be in the way.

Appendix D

User Guide

D.1 Building the project

The instructions on how to navigate the code or build the project can be found here - `https://github.com/tobhil98/MastersProject-AirSim`.

D.2 Simulator Research

D.2.1 Building Unreal Engine

On Windows Unreal Engine can be downloaded through the Epic Games launcher. However, on Ubuntu the project has to be built from source[1]. On Ubuntu UE requires a modern GPU to run.

D.2.2 Building AirSim

AirSim can be built on windows using this link[2], and this link to build it on Ubuntu[3]. Both have been tested and work. The only inconvenience found was that the simulator has to be built using the Visual Studio 2019 development terminal, but after installing Visual Studio 2017 which is used to build Carla, this no longer worked. Simply uninstalling VS2017 resolved that issue.

D.2.3 Building Carla

Carla built successfully on Ubuntu following this guide[4] and after doing these alterations:

- As I was using Ubuntu 20.04 I had to install Python2/Pip2 using curl
- Clang-8 was outdated so I installed Clang
- Clang-Tools-8 was outdated so I installed Clang-Tools

[1] `https://docs.unrealengine.com/4.26/en-US/SharingAndReleasing/Linux/BeginnerLinuxDeveloper/SettingUpAnUnrealWorkflow/`

[2] `https://microsoft.github.io/AirSim/build_windows`

[3] `https://microsoft.github.io/AirSim/build_windows`

[4] `https://carla.readthedocs.io/en/latest/build_linux`

- lld-8 was outdated so I installed lld

However, this build was never tested as Unreal Engine would not run.

Carla did not build successfully on Windows before, but an updated guide is now available here[5].

D.2.4 Building Gazebo

Gazebo built successfully on Ubuntu following these instructions [6].

D.3 ML-Agents

ML agents is well documented and can be found here - `https://github.com/Unity-Technologies/ml-agents/tree/main/docs`.

D.4 Maps and Environments

More detailed instructions for how to create the maps and environments can be found here - `https://github.com/tobhil98/FinalYearProject-TobyHillier`

[5] `https://carla.readthedocs.io/en/latest/build_windows/`

[6] `http://gazebosim.org/tutorials?tut=install_ubuntu&cat=install`