3D TRAFFIC SIMULATION IN UNITY: HOW TO USE SUMO TO PRODUCE 3D TRAFFIC IN UNITY

Chadjiminas Chrysostomos 932795

A Thesis

Submitted in Partial Fulfillment of the

Requirements for the Degree of

Bachelor of Science

at the

University of Cyprus

Recommended for Acceptance

by the Department of Computer Science

June, 2019

APPROVAL PAGE

Bachelor of Science Thesis

3D TRAFFIC SIMULATION IN UNITY: HOW TO USE SUMO TO PRODUCE 3D TRAFFIC IN UNITY

Presented by

Chadjiminas Chrysostomos 932795

Research Supervisor _____
Yiorgos Chrysanthou

Committee Member _____
Panayiotis Charalambous

University of Cyprus

June, 2019

ABSTRACT

In this thesis we explore methods of integrating a traffic simulation in a game engine. More specifically we employ a server-client architecture; SUMO, an open source microscopic traffic simulator, acts as simulation server whereas we develop an interactive visualisation client in the Unity game-engine side. We analyse the challenges that arise when using a server based simulator along with unity to produce a realistic 3D traffic simulation that takes performance into account. We describe solutions to this problems along with a deeper understanding of their cause so other people that will attempt to make similar projects will have some insight of possible challenges and resolutions. This report also serves as a guide to understand the created tool so that people that want to extend the tool with other functionality will be able to.

ACKNOWLEDGEMENTS

First and foremost I would like to thank my family for supporting me through this journey of not only my thesis by my whole academic life in the university as well. I would also like to thank my friends for always being there for me, for helping when I needed them the most and for encouraging me through all my hardships. Warm thanks to my advisor Prof. George Chrysanthou for trusting me with this thesis and for always being supportive and positive throughout the thesis development. Special thanks to Mr. Panayiotis Charalambous for helping me and advising me through each step of my thesis and for helping me overcome the challenges that I faced. I would also like to thank the RISE staff for welcoming me to their offices when I didn't have a laptop. Last but not least, I would also like to thank the the graphics lab research group for always being more than happy to help me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

Introduction

1.1 Motivation

In this current time and age, there are is a strong need for training data and training environments. a lot of applications also require not only data in the form of files or simple images but videos as-well.

Usually, autonomous car manufacturers that want to train vehicles, have two main ways of gathering training data. they either outsource to proprietary datasets from other companies or produce their own homegrown datasets, something that requires effort and time. Collecting real life videos of traffic for different scenarios can be very cumbersome and capturing special events such as traffic jams or very rainy weather can sometimes be situational and hard.

There is a third alternative way of collecting data to train vehicles. This is through 3d traffic simulators that use game engine to produce a realistic traffic scenarios. Autonomous vehicles use Q-learning (or more appropriately deep Q-learning since it involves convolutional neural networks) which is a reinforcement learning algorithm. reinforcement learning, by definition, is

a way of machine learning that requires that the learning agent learns by positive and negative rewards.

Even though the number of accidents with autonomous is expected to be low and the frequency of them lower than with human operated cars, accidents do happen and when they do people tend to lose their trust to this new technology. Moreover, a human life shouldn't be put to risk. If a fatal accident can be avoided then all the measures to avoid that accident should be taken.

A 3D traffic simulator traffic can provide an adequate environment for self driving agents to train and test without the risk of causing real harm. A high fidelity graphics 3D traffic simulator can provide realistic data that are similar to real life and could be used for training. Through such a simulator, training data can be of the form of images used by self driving cars that rely on computer vision, LIDAR, IMUs or other sensors.

Although self driving cars can benefit from 3D traffic simulator, their applications are not limited only for training autonomous vehicles. Many more applications could use 3D simulators. For example open world games where a realistic traffic system is desired, 3D animations, VR driving simulators, research for smart city, city planning, advertisements and many more.

The main challenge of a 3D traffic simulator is the accuracy and the simulation of real life traffic since to create realistic simulation there are a lot of factors to take into account, such as human inaccuracies when steering their car and the routing inside a traffic network to name a few. Thankfully, applications that produce such simulations exist.

1.2 Traffic simulators

Traffic simulators are generally used to study mobility patterns and are able measure many things, from the average time spend idle in a traffic jam to the Co2 emissions over the course of an hour. Using such traffic simulators, one can test new communication protocols or design his own road network to evaluate his/hers proposal in a realistic traffic network.

Mainly, traffic simulators can be separated to three categories depending on their scope[12]

Microscopic simulations: Microscopic simulations are simulations that operate on a per-vehicle basis. These simulations model individual car movement and focus on criteria such as Human Mobility Patterns (the interaction of a car with other cars based on human motions that are described by mathematical models), Lane changing (when and how humans decide to change lane when driving), and intersections (how cars act on intersections).

Macroscopic simulations: Macroscopic simulations focus on trip and path generation. Macroscopic models define and assign routes to vehicles based on a graph that describes the road topology with nodes usually being points of interest (i.e a super market or a hospital). When a vehicles is assigned an initial and destination position on the graph a path must be generated based on a path generation algorithm.

Mesoscopic simulations: Mesoscopic simulations combine both microscopic and macroscopic properties.

1.3 Why use a game-engine?

Most traffic simulator software provide visualisation (i.e VISSIM[31]) of the simulations but most come with the software and provide no means of adding other functionality or adapting

other systems to it. For this reason, if someone would want to provide this feature then they could use the simulator along with a game-engine to present the traffic simulation in a 3D environment. This method allows for scalable solutions.

A game-engine is a software that provides a framework for people to design and build video games. A game-engine framework provides multiple components of which some are crucial and some are not for developing a game. The engine's most important components are a rendering component that can render 3D and/or 2D graphics, a component that handles physics and collisions, an audio component, a scripting component, animation. Other components exist, such as post-processing, depending on the game-engine.

With all these features, game-engines can be used not only for developing video games but for many other applications such as physic simulations, cinematic animations, and research purposes to name a few.

A game-engine, as previously stated, can be used along a traffic simulator. The use of a game-engine with a traffic simulator presents significant benefits, depending on the use case. For the purpose of realism and the creation of training data, a game-engines components could be proven very helpful. Multiple scenarios, independent of the traffic simulator, can be created. For example, a rainy, a foggy and a clear day can be rendered for the same training simulation scenario. That could be proven very useful for training autonomous vehicles since they must be trained under different weather conditions. Additionally, game engines can produce and simulate the same views as those used by autonomous vehicles sensors in real life. Microsoft's AirSim[2] is a very good example for this. AirSim can be used to gather training data and can provide views such as Lidar, IMU, GPS, Magnetometer, Distance Sensors and also comes with a

Computer Vision mode that can collect images like depth and disparity. This plugin is provided for Unreal Engine and experimentally for Unity as well.

In addition, a game-engine provides a common platform for other applications that use the same game-engine to interact with each other. For example, if the simulator doesn't provide a pedestrian simulation, if a pedestrian simulation exists in a game engine then it could communicate through the game scene so that both the pedestrians and vehicles could coexist in the same 3D environment to achieve realistic results. A limitations for this method is how many and what state changing features do the two (or more) collaborating systems use and how well they can communicate with each other. Adding to the previous example, if a vehicle can not respond to a dynamic object entering the road network then that would mean that the vehicle wouldn't be able to react to the approaching pedestrian. A solution to this particular example would be to simulate the slow down and halting of the vehicle when the pedestrian approaches. That would mean that the traffic simulator allows such state changing command and that additional middleware code should be written to simulate this behaviour.

Unfortunately, using a game-engine with a simulator presents many challenges as well. Different combinations of game-engine/traffic simulator present different challenges. This is because different traffic simulators provide different APIs and have different capabilities and so do game-engines. Regardless the challenges for the communication, the goals are mostly the same, consistency, realism and performance.

Optimally, the state of the traffic simulation should be the same with the traffic simulators state. In other words, the states of the simulations should be consistent with each other. For example, if a vehicle is at a certain position and with a certain orientation at the time τ in the

traffic simulators simulation then at the same time τ it's 3D representation should be at the same position with the same orientation. With some traffic simulators this is desirable but for others not because it could lead to unnatural movements since a certain simulator was not meant to be used along a 3D visualisation tool. In such cases, additional measures should be taken to make the 3D representation more realistic while sacrificing some consistency in the process.

Realism is also a very important factor for 3D simulations since not having realism would defeat the purpose of using a game-engine. As previously stated, realism could be compromised for consistency and that some measures should be taken to make the vehicle movement more realistic. Another factor that should be taken into account when considering realism is of-course the 3D environment in which the vehicles will move. Especially for applications that require high fidelity graphics like for autonomous vehicle training.

Another challenge that can be posed is of course that of performance. Depending on the traffic simulator and the communication medium, communicating with the simulator would add an extra overhead. For example, if a traffic simulator offers its service as a server then each simulation step would happen through a socket and that would add a significant overhead that would result in unpleasant and clunky results with frame drops. Performance is not a problem if no real-time commands should be sent to the traffic simulator since a simple solution would be to just run and bake the traffic simulation beforehand. However, for many applications, this is not desired since many applications have dynamic events that should react with traffic simulation.

1.4 Independent traffic simulator vs Game-engine dependent simulator

Another question would be, why use an independent traffic simulator when you can use a traffic simulator that is developed specifically for that game-engine?

First of all, we must understand the benefits of using a game-engine targeted traffic simulator. A traffic simulator developed for a specific game-engine will, usually, be optimised for that game-engine. There will be no middleware, be it either simple API calls or calls to a server, to be developed to support the communication. Consequently, interfacing the simulator should be just as easy as using any other components of the game-engine. Additionally, having a traffic simulator for a game-engine can provide easier communication with other components that are developed for that game-engine. Furthermore, most of the times, since the simulator will be designed to be used by a game-engine, it will use the game-engines physics engine or animations that would provide a more realistic movement than using traffic simulators that weren't developed with the visualisation part in mind.

Many times, there is no such possibility simply because a traffic simulator may not have been developed for that specific game-engine. Other times, traffic simulator may exist but may be too expensive or a proprietary asset. If such a traffic simulator exists for the game-engine, then there is a high probability that it will not meet our requirements when we want to achieve realism. In many cases, traffic simulators used specifically for game-engines are oriented for use in games or cinematics and do not follow realistic traffic patterns.

A solution for that is to create a homegrown traffic simulator specifically tailored for the application that the developers want with only the features and realism that is required by the

customer. This of course , is one of the most costly and expensive solutions since all the traffic simulation logic should be implemented from scratch

Another drawback is that, traffic simulators targeted toward a specific game-engine can have significant costs to port to another game-engine and can be more costly than creating a new traffic simulator scratch. On the contrary, traffic simulators that are independent from a game-engine can be easier and faster to port to different game-engines, depending on the traffic simulator.

1.5 The thesis

In this thesis, we describe a process of how one can use sumo, a server based traffic simulator, alongside with unity, to present vehicle traffic in a 3D environment. We also present the challenges that arise when you have to use unity with a server. The purpose of this thesis it to find out what are the capabilities and limitations of using SUMO alongside unity.

1.6 SUMO: A brief introduction

Sumo[14] is an open-source microscopic traffic simulator developed by the Institute of Transportation Systems at the German Aerospace Center. It is designed to handle large road networks and it can work with road networks from sources such as Open Street Maps[20], GIS data (shapefiles), and VISSIM[31]. SUMO and it's use for this thesis will further be described in "SUMO" and "Sumo In Unity" chapters respectively. Additionally, a thorough documentation of the sumo suite can be found at the sumo wiki[25].

1.7 Unity: A brief introduction

Unity is a cross-platform game engine developed by Unity Technologies. Developers can use the engine to create both 3D and 2D games. The engine is also used by industries outside video gaming to create simulations, and films and use it for architecture, automotive, engineering, and construction. The Unity engine is implemented in C++ but provides a scripting component that is in C# which is an object oriented language. This provides an easy and developer friendly environment for creating large projects fast and in a clean way. Unity provides the UnityEngine library that has classes such as **`Monobehaviour`** and **`ScriptableObject`** with the first one being the one used mostly.

`MonoBehaviour` is the base class which every Unity script derives. Scripts that derive from MonoBehaviour are the only scripts that can be attached to unity game-objects. A **`MonoBehaviour`** script that is attached to a game-object is called a component. Any other classes that do not derive from **`MonoBehaviour`** still behave normally and can be used just as well but can not be attached as component to game-object. Game-object are the entities that can exist on the scene. A GameObject can be a sphere, a player or even just a placeholder to attach a component. A script should derive from **`MonoBehaviour`** if it needs to react to unity's callbacks such as `Update()`, `Start()`, `Awake()`, and `OnDestroy()` to name a few. Another reason would be so that configurable variables to be exposed in the unity game object inspector so that they can be changed without recompiling. This is particularly convenient when testing in play-mode.

Chapter 2

Overview

2.1 Sumo Communication With Unity Overview

2.1.1 TraCI

In order to use sumo with unity a communication between the game-engine and the sumo server that runs the simulation must be established with a client issuing commands and the server responding back with the simulation state.

The whole communication works under TCP. In the application layer, the protocol used is TraCI(Traffic Control Interface)[32], a protocol that was developed specifically for sending and receiving commands from and to a sumo server that runs a traffic simulation. To be more precise, TraCI is a more generic architectural pattern, or otherwise a guide that was developed for traffic simulators in general. TraCI creators provided a TraCI protocol implementation for sumo along with their publication. This protocol allows for interfacing a sumo simulation server and controlling the behaviour of vehicles during simulation run time. Consequently, the resulting

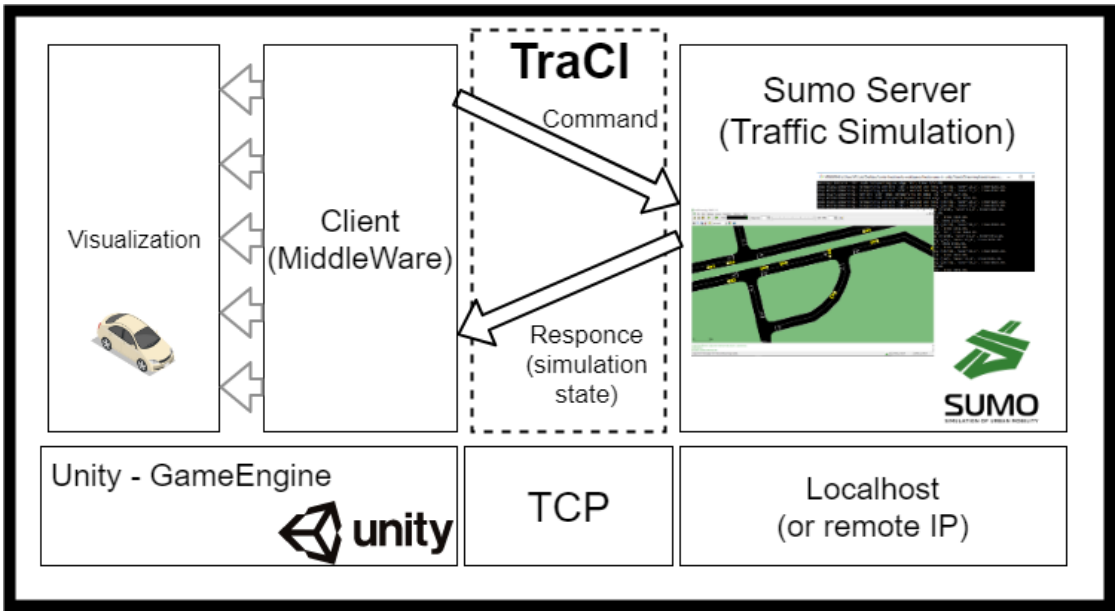


Figure 1: From su mo simulation to unity visualisation overview. The sumo server is launched and a client from the unity game-engine side connects to it. The communication uses TraCI, an application layer protocol, that is supported by TCP. The client sends commands such as execute simulation step and retrieves the simulation step each step. The client component has a sub-component that simplifies and wraps the retrieved result to be used easily by the visualisation component that lives at the game-engine side.

simulation can be dynamic and can react to changes that happen during run time. As stated in the introduction, some changes that are outside the sumo simulation environment should be simulated. For example, if a new pedestrian enters the road network the vehicle should slow down to halt. This can be simulated by sending a command to the vehicle to slow down when the pedestrians enters the road.

The TraCI protocol is thoroughly documented in the sumo wiki[25][28] and describes all the available commands and data that can be received, the data types and their representations, and the format for the commands and responses. The protocol and how it will be used will be further described in the chapters ”SUMO”3 and ”Sumo In Unity”4 respectively.

For the 3D traffic simulation we have two simulations, the simulation at the game-engine side and the simulation at the sumo side. The game-engine simulation tries to replicate the sumo simulation as a 3D representation. This replication means that for each vehicle in the sumo simulation there is a corresponding car [1] in the game-engine side. This also applies for other elements such as lanes. Even more abstract elements, like routes (paths that the vehicles must follow), can have a representation in the game-engine side. In our application, only the a subset of the elements have a representation but the code was carefully crafted to easily support a representation for all the sumo variables[2] through the use of C#'s object oriented architecture. The process of going from the simulation from the sumo server to the 3D visualisation in the unity engine can be seen in brief in figure 1 and is thoroughly described later in the chapter "Sumo In Unity" 4.

It's good to note that, because of the sumo server limitations, for the simulation to make a step a command must be issued by the client. Contrary to how many other simulations run, sumo requires an explicit step command to calculate the next simulation state. This makes it harder to run the traffic simulation in parallel with the client since sumo doesn't support concurrency by design.

2.2 Traffic scenario Creation for sumo

To create a realistic traffic scenario for sumo can be a challenge by itself. Some sumo scenarios and the process for their creation are described in the sumo wiki[25][26]. In the wiki

[1] From now on, to distinct between the sumo server and game-engine side, the term **vehicle** is used for the first and the term **car** is used for the latter. An exception to this rule is when referring to a **vehicle instance** that is an instance of the `Vehicle` class in the game-engine side

[2] Another term for a sumo element

three notable examples are presented with their representing sumo configuration files[3] freely available for public use.

LuST(Luxembourg SUMO Traffic)[7] Scenario is a sumo scenario that was developed to model 24 hours of mobility in Luxembourg based on real data that describe the traffic demands and mobility patterns of the city. In the publication, the process and challenges of the creation of the scenario are presented. The metrics under of which the creators assess their work is the size of the network and the demand, the realism of the scenario and the duration. In the publication they also give a brief overview of the scenario characteristics and some use cases where the scenario can be applied. The whole scenario is available at a github repository[6].

A similar project was created for Bologna[5]. It is a smaller scale scenario (models 2 hours of mobility) that was created for public use. The scenario can be obtained from sourceforge[4]. Another project that was made public (GPLv3 license) is the MoST(Monaco SUMO Traffic)[10] scenario. The scenario models vulnerable road network uses such as pedestrians, cyclist, and motorcyclists. The creators of these scenario aim to provide a playground to study parking management solution, other transport modes applications while being realistic. The scenario can also be obtained from github[8].

In this thesis, the creation of realistic scenarios is not the main goal but the use of a realistic scenario that was researched and well developed scenario, such as the examples above, is highly encouraged. The realism of a 3D simulation is assessed by not only the realism of the vehicle movement, but also by the demand and mobility patterns of the vehicles. For our application, we assume that realistic scenarios with demands and mobility patterns that are based on real

[3].sumo.cfg is a configuration file that links all the input files that are needed for the simulation

life data exist. However, the aforementioned scenarios were not used in this thesis since we want to show a more realistic scenario that takes into account 3D realism rather than the traffic simulation mobility patterns realism. For this reason, we chose to use a high fidelity graphic city for our demo. In the unity asset store, Windridge city is an asset of a 3D model of a city with a road network that can be used to move the cars on. This assets is used in the AirSim[2] demo asset that was developed for unity.

For the purpose of this thesis, we describe the process to create simple random scenarios for testing. We also describe the process of creating scenarios that use OpenStreetMaps[20]. Simple scenarios were created for Nicosia in an attempt to match the network simulation with Nicosia models we have. We also describe how we represent the road network from sumo into the unity game-engine side and how that network is used. This process is described at the chapter Sumo In Unity4.

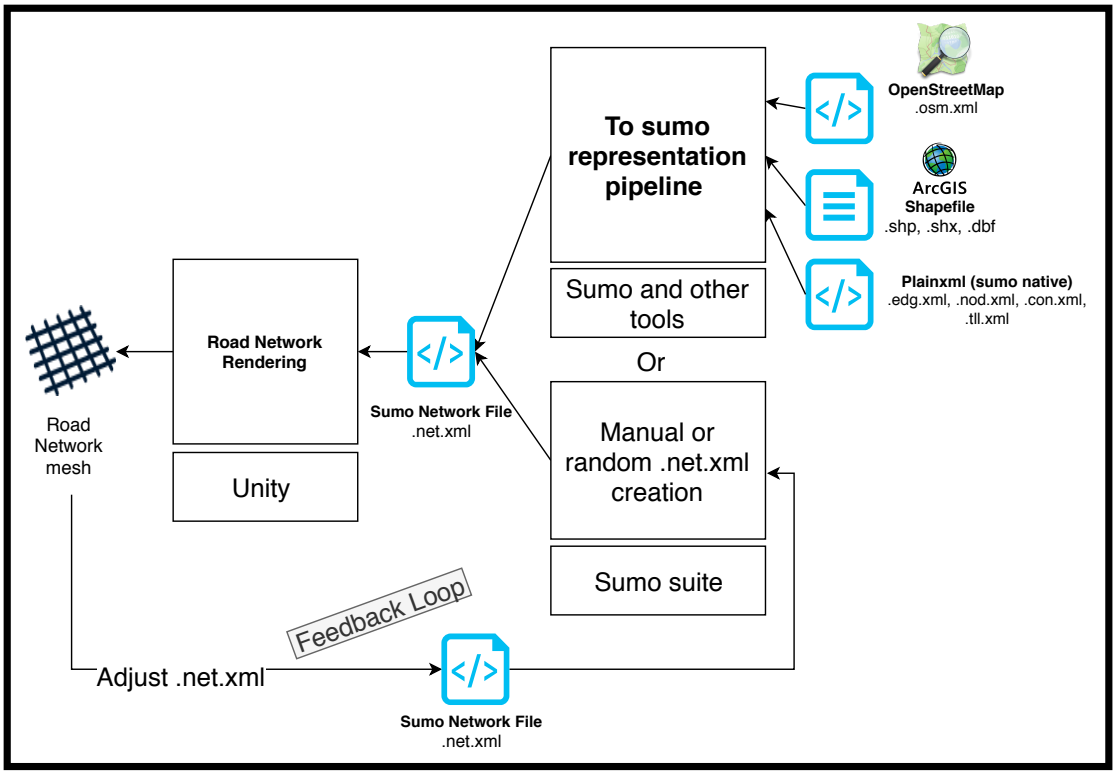


Figure 2: An overview of the creation of a sumo road network file. Going from different representations to an accurate .net.xml file is not a trivial task and requires manual processing of both the resulting .net.xml and the source representation networks. After creating an admissible .net.xml file, a component takes the sumo network file as an input and gives the road network representation in unity. After the generation of the road network mesh the user must test it and readjust the .net.xml through tools such as NETEDIT in a feedback loop until he/she is satisfied.

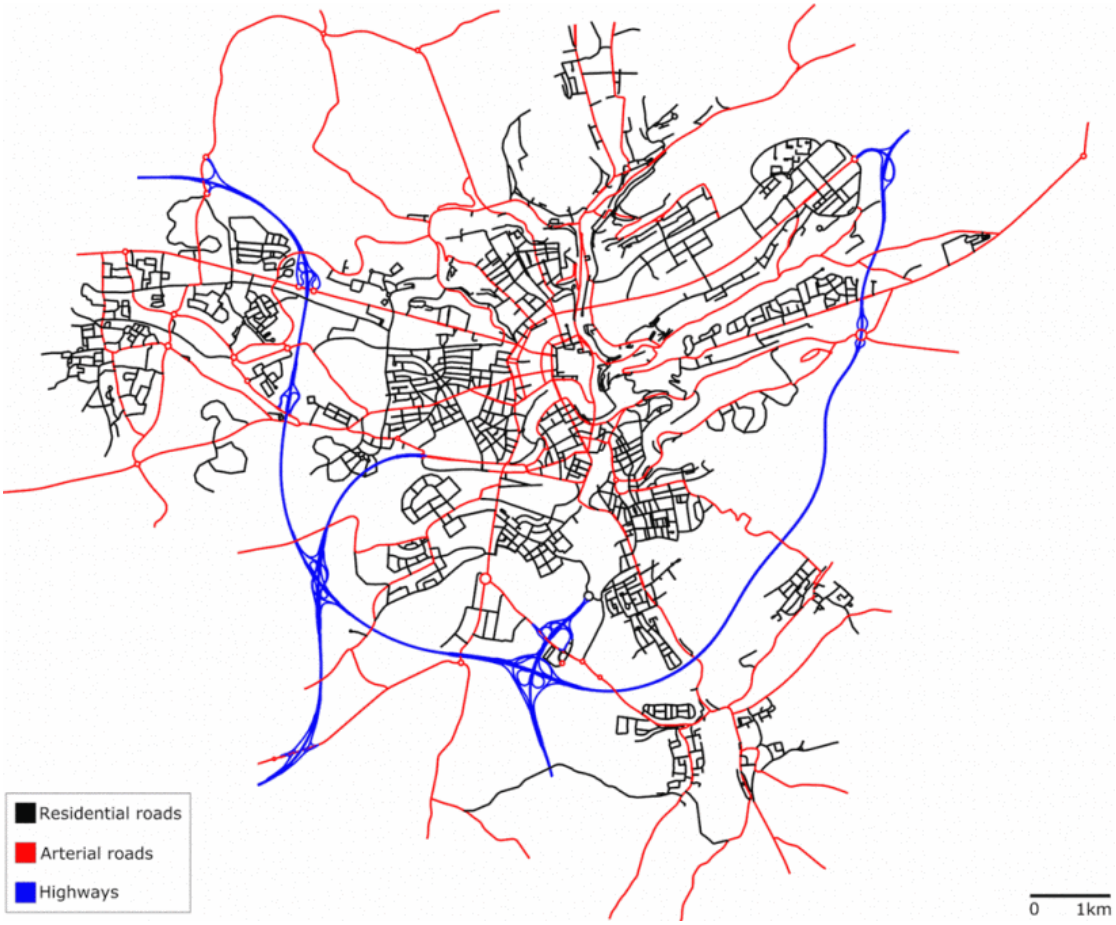


Figure 3: LuST Scenario Topology

Chapter 3

SUMO

3.1 Summary

SUMO[15] Simulation of Urban MObility is an open-source traffic simulation. It is a microscopic and continuous road traffic simulation package designed to handle large road networks.

SUMO works as a server which runs a traffic simulation and offers an online communication to clients that connect to it. In order to use it one can use SUMO or SUMO-GUI[1] to launch the server locally with and without GUI respectively. The client can then connect to it using the TraCI3.4 protocol.

Sumo can run embedded as-well. This can be done with libsumo. This could prove very useful since using a socket to connect to a server, even when if it runs locally, is much slower.

[1] whatever can be done by SUMO can also be done by SUMO-GUI

3.2 Why SUMO

Sumo was chosen over other traffic simulations because it offers an open-source and microscopic traffic simulation. It comes with a big set of on-line simulation control that meets our requirement since we want to stop and control vehicles when pedestrians are about to cross the road. Additionally, sumo is open source and with an active community which makes it easier for troubleshooting. By subscribing to the sumo mailing list one can ask any question to resolve any problems that can not be found in the documentation. What is more, sumo is updated and is quite active which makes it a very good choice since new features are implemented and existing features are improved. Sumo also comes with a set of tools and executables that gives the users the opportunity to create digital networks and simulations that fit the users need. For example one can use NETCONVERT[17] to create files from various formats to .net.xml3.6.3 which is very useful since it is very easy to create maps from OpenStreetMaps[20] or any other sources that NETCONVERT support.

3.2.1 tools

In order to create road networks, routes, vehicles, vehicle types, there are many tools that come with sumo.

3.2.1.1 SUMO - SUMO GUI

SUMO is the main tool that launches the sumo server. The IP and the port to launch the sumo server can be set as command line arguments. If the user wants to have a graphical interface then SUMO-GUI can be used. It is the same as sumo but also produces a GUI with the vehicles and

the road-network visualised mainly in 2D views. Whatever can be done with SUMO can also be done with SUMO-GUI. It must be noted that SUMO is faster than SUMO-GUI since when using SUMO there is no rendering. Once a sumo server is launched by using this command, then a client can connect to it or the server can run independently. It should be noted that, **once a client connects to a sumo server, the simulation can only make a simulation step only if it is issued by the client**. SUMO also offers the opportunity for multiple clients to connect to the same sumo server but the clients must communicate with the server in a round robin fashion always ending with a simulation step command.

3.2.1.2 NETGENERATE

NETGENERATE[19] provides an easy way to create a road network for testing purposes. Through this tool one can easily create an abstract road network of three types: grid, spider and random. In this thesis it is mainly used for creating spider grid to test performance.

3.2.1.3 NETEDIT

NETEDIT[18] provides a graphical interface for creating or editing a road network by hand. With this tool one can create and edit lanes, edges, junctions, traffic light systems and configure right of way. With the new version of sumo(1.2.0), NETEDIT also allows for demand management through this editor.

In our thesis, it is mainly used to micro adjust road network so it matches the road network model that exists in unity.

3.2.1.4 NETCONVERT

NETCONVERT[17] is a tool that allows the conversion of road networks of different formats to and from .net.xml. This is a very useful tool that allows us to grab digital road networks from OSM to use them for sumo. This tool was used to create an old Nicosia digital road network using openstreetmaps[20]. The full process that was followed is shown at figure 6.

3.2.1.5 Creating random trips

randomtrips.py is a python script that creates and assigns trips to vehicles given a road network(net.xml). The output of this script is a trips.xml file that can be used by DUAROUTER to create a route file.

3.3 Sumo basic objects

There are several important objects in a sumo simulation. All sumo objects have their representing id that is unique for that particular type of object. The ids for each element can be found defined in the xml files. For example lane ids can be found in .net.xml file. Additionally all ids can be retrieved through TraCI3.4. Some of the main objects are described very briefly below.

3.3.1 Vehicles

Vehicles are the most important element of a sumo simulation. The vehicle can have a type that describes it's physical properties, a route, and information about the vehicle itself. Among other variables, each sim step, a vehicle's speed, acceleration, position and angle can be retrieved

using TraCI. In rou.xml3.6.2 file each vehicle is assigned an id, a type, a route, a departure and an arrival time.

3.3.2 Routes

Each vehicle is assigned a route that will follow unless the route changes online. A route is a set of edges that the vehicle will follow in the simulation (unless the simulation state changes online).

3.3.3 Edges

An edge represents a road. It is a set of points with the first point being the starting position the the last the end position of the edge. Because of this each edge has a direction. Therefor, a two direction road is represented by two edges with two different ids. Each edge is a set of lanes.

3.3.4 Lanes

A lane is described by a from and a to node. That way a lane has a direction. A node is a 2D or a position. Lane definitions can be found at .net.xml3.6.3.

3.4 TraCI

Summary TraCI(**Tra**ffic **C**ontroll **I**nterface)[32] is the protocol used for the communication between clients and the SUMO server. It's an application layer protocol that is supported

by the TCP protocol. The client(s) can then retrieve or alter the simulation state online. [2] . The TraCI protocol is important for issuing commands to the SUMO server and polling data from it. This protocol is thoroughly documented in the sumo wiki[25][28].

In general, there are Value Retrieval commands, also known as Get Commands, State Changing commands, also known as Set Commands and Subscription commands3.4.1 that are also commands for retrieving values from the simulation. There is also a special kind of command that executes a sim step. This command executes one simulation step and can also retrieve information subscription information if any subscriptions took place prior to this step. TraCI is the main way to interact with a running simulation. It provides complete flexibility of doing cross-platform, cross-language, and networked interaction with SUMO acting as a server.

3.4.1 Subscriptions

Subscriptions3.4.1 are commands that are used for value retrieval. Subscriptions allows for multiple data to be packed into one TCP packet. Subscriptions are retrieved each simulation step. When a sim step is issued with one or more prior issued subscriptions the sumo server sends a response with all the subscribed data.

There are two kind of subscriptions:

3.4.1.1 Variable Subscriptions

Variable Subscription[30] allows the the periodical retrieval of a set of subscribed variables for a subscribed sumo object. For example TraCI can be used to retrieve the speed, position and

[2] Given a sumo configuration file3.6.1 the sumo simulation is always the same unless the state is changed during run-time

orientation of the vehicle with id ”0” for a given time. This means that each sim step these three variables are packed in a response.

3.4.1.2 Context Subscriptions

A Context Subscription[29] allows the subscription to a sumo object, called EGO object[3] , to retrieve information about other sumo objects that are around that EGO object, inside what is called a context range. The context range is the radius around the EGO object to retrieve data for. For example the position and rotation of every vehicle that is in range of 50 meters or less around a lane with a given id can be retrieved.

3.4.2 Drawback

One major drawback is the communication overhead due to the protocol and the socket communication. This is very bad for our case since we want to have a real-time communication with the sumo simulation and we want the cars to move both smoothly and synchronised with the simulation. Thankfully, TraCI API is provided as a C++ library.

3.5 Libsumo : TraCI API as a C++ library

Libsumo is a library in C++ that offers the same functionality as with TraCI but without having to connect to a server and use sockets. According to the documentation[16], this library has the following properties:

[3]For a full list of context supported EGO objects please check https://sumo.dlr.de/wiki/TraCI/Object_Context_Subscription

1. C++ interface based on static functions and a few simple wrapper classes for results which can be linked directly to the client code
2. Function signatures similar to TraCI
3. Support for other programming languages via SWIG
4. Pre-built language bindings for Java and Python (using SWIG)

The first two points mean that transition from TraCI should be relatively easy. For this reason we experiment with TraCI and then try out the dll. The ability to support other programming languages via SWIG could be proven very useful.

3.6 Xml files

In order to for a sumo simulation to take place, several xml files are needed. The minimal amount of these files is a sumocfg.xml, a rou.xml and a net.xml

3.6.1 sumocfg.xml

This is a configuration file that is given as an input to sumo.exe. It references the rou.xml, net.xml and any other files that can be used or are needed for the simulation. A minimal example from sumo documentation:[13]

```
<configuration>
    <input>
        <net-file value="hello.net.xml"/>
        <route-files value="hello.rou.xml"/>
```

```
    </input>
</configuration>
```

3.6.2 rou.xml

Defines vehicles and routes or even vehicle types. Example from sumo documentation[23]:

```
<routes>
    <vType id="type1" accel="0.8" decel="4.5"
    sigma="0.5" length="5" maxSpeed="70"/>

    <route id="route0" color="1,1,0" edges="beg middle end rend"/>

    <vehicle id="0" type="type1" route="route0"
    depart="0" color="1,0,0"/>
    <vehicle id="1" type="type1" route="route0"
    depart="0" color="0,1,0"/>
</routes>
```

As we can see here a vehicle type is defined with vehicle properties. Vehicle types can also defined in an additional file instead of rou.xml. A route is also defined (color is for sumo gui) with a the set of edges (separated by space) the vehicles will follow. Additionally two vehicles of type "type1" depart at time 0.

3.6.3 net.xml

The net.xml defines the road network. It defines the edges and their lanes. It also defines junction, their types, traffic light programs and other road network related information. A simple example from documentation[27]:

```
<edge id="<ID>" from="<FROM_NODE_ID>" to="<TO_NODE_ID>"
priority="<PRIORITY>">
    <lane id="<ID>_0" index="0" speed="<SPEED>" length="<LENGTH>"
    shape="0.00,495.05 248.50,495.05"/>
    <lane id="<ID>_1" index="1" speed="<SPEED>" length="<LENGTH>"
    shape="0.00,498.35,2.00 248.50,498.35,3.00"/>
</edge>
```

Chapter 4

Sumo In Unity

Before moving on it is important to note that the term vehicle and car are not used interchangeably. For the simulation of the vehicles in unity an **MVC** like architecture was followed. The following terms will be used frequently:

vehicle : The vehicle from the SUMO simulation. Can be thought of as the **model** in an MVC architecture.

car : The 3D game-object representation of the vehicle in game-engine side. Can also be thought as **view** in an MVC architecture.

`Vehicle` : The class or class instance that holds the data and properties for a vehicle from SUMO simulation. Also provides convenient conversions from the 2D XY to Unity's XZ plane. Can be though of as another **view** in an MVC architecture.

`VehicleController` : A script that is attached to each car gameobject. Wraps sumo commands for a vehicle so that a vehicles state can easily change in the simulation. Can be thought of as a **controller** in an MVC architecture.

4.1 Summary

The most important part of this thesis is bringing sumo into unity. That means the creation of car gameobjects that represent the vehicles in sumo simulation and the synchronisation of their movement in unity. The result should be a realistic movement of cars in unity while a real-time communication with the sumo server should be available. This would allow other systems, such as pedestrian simulation in unity, to interact with vehicles and see their representing cars in unity change state.

4.2 Road network creation in unity

In order to create a road network in unity the net.xml3.6.3 file is used. For each lane3.3.4 a mesh object was created using the shape information find in the xml file in the lane attributes. The class used for this purpose is **`RoadNetworkRenderer`**. The name and the materials can be assigned using this class. Each lane can have 1 or more game objects that represent it. **`RoadNetworkRenderer`** exposes Length Threshold in unity editor where one can select what is the max length of a gameobject's threshold. If a lane has more length than Length Threshold then the length of the representing mesh is recursively split into smaller meshes until the length of each mesh is smaller than Length Threshold. Each lane game object has a script with the lane information. In case where a mesh is split into many, each gameobject has the

same lane information (represents the same lane from the sumo simulation). Additionally each lane gameobject has a collider and a lane controller script. The lane controller script allows for easy access to SUMO commands such as context subscription.

4.2.1 Usage

It must be noted that the rendered road network mesh is not meant to be shown during play mode since the this mesh is not realistic and the existing model's road is meant to be shown. However, the generated road network representation serves for two crucial purposes. Because we want to match the simulation road network with the model's road network, the mesh can be used as a guide to evaluate this. If the generated road network doesn't match the the model's road network then the vehicles will move outside the road producing weird artefacts such as cars driving through buildings. To fix the inconsistencies between the modelled road network and the sumo network we must enter a feedback loop of adjusting the .net.xml file, rendering the new network and testing how well the new rendered road network matches. An example of an inconsistency between the modelled road network and the created road network mesh can be seen in the figure4.2.1.

The other crucial reason for creating the road network representation in the unity side is because it's mandatory for the context subscription implementation. Each lane game-object has a MonoBehaviour script that holds the lane's id and position that can be used for this purpose. Only lane game-objects that are inside the camera's frustum will be considered for context subscription. More about how the lane game-object and the camera will be used for efficient context subscription will be described in a later section in performance chapter5

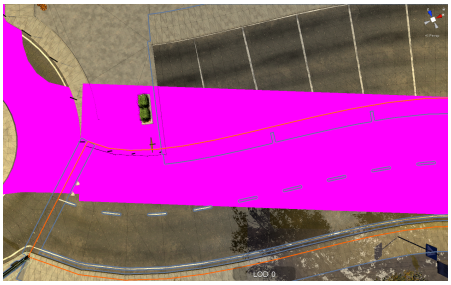


Figure 4: An inconsistency between the rendered road network and the model road. Notice that the the mesh that was rendered using the .net.xml file (pink colour) goes straight through the pavement and therefore would result in erroneous results where vehicles go through the pavement.

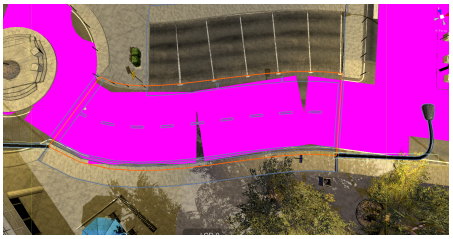


Figure 5: The same rendered road after going a feedback loop to fix the inconsistency. The .net.xml was edited using the NETEDIT3.2.1.3 tool to match the curvature of the model road.

4.3 Create Scenario using osm data

One of the initial requirements of this thesis was to use the city of Nicosia for the simulation. A model of Nicosia was given and the requirement to create a sumo scenario for that model. Because of the size of that model, creating a road network from scratch using would be very cumbersome and beneath the purpose of the thesis. As a result, it was decided that OpenStreetMaps[20] data will be used to create the scenario. However, it was quickly realised that the given model was different than the generated road network from the osm data was different and had a different source. It was speculated that the source was the Cyprus Department of Lands and Surveys. Consequently, it was decided to contact the department to receive the spacial data for the Nicosia city as they are officially recorded. As a result, the department had sent a shape-file with the data. Unfortunately, converting from shapefile to the sumo network

configuration is not trivial because the given shapefile lacked the lane directionality which is important for the conversion to have accurate results. Due to the specific format that the shapefile is required to be by the NETCONVERT tool, manipulating the shapefile might be necessary. Because converting from shapefile to the sumo road network representation is not very common and besides the purpose of the thesis the steps are omitted. If it required to create a sumo network from a shapefile please refer to the sumo-wiki[25][24].

Eventually, because the model of Nicosia was not realistic in it's current state the requirement to build a scenario for that model was ditched. Because the process of creating a sumo scenario from osm is very common it is described in the figure 6. To be more precise this process describes how the osm data was used to create a sumo configuration to match the Nicosia model road network.

1. First we download the osm data from an osm exporter. The user must give the bounding box for the data to download and the filters (osm data layers to ignore).

2. After fetching the osm data the user can optionally refine the the osm road network with the JOSM tool that can edit osm networks.

3. The osm fil is then used as an input in NETCONVERT to create the sumo network file. Some command line arguments are advised to be used to make the convertion better. Please refer to the sumo-wiki on how to import osm data [25][1].

4. The created sumo network file is rendered by the **`RoadNetworkRenderer`** component to create the generated road network mesh.

5. The generated road network mesh is then tested against the model's road network. The sumo network file should be edited to fix inconsistencies between the generated mesh and the road network model.4.2.1. This feedback loop should be repeated until an admissible sumo network is reached.

6. After reaching the last iteration of the sumo network, the randomTrips.py is be used along with DUAROUTER to create the route file (.rou.xml).

7. The sumo network file and the generated route file are then linked in a sumo configuration file.

8. The sumo configuration file can then be used to launch a sumo server with the scenario.

4.4 Establishing connection and data retrieval with SUMO

For moving the cars the data from the corresponding vehicles should be retrieved. In order to get information about the current state of the SUMO simulation the **TraCI** protocol 3.4 should be used. In order to use the protocol for unity one should create a wrapper (a.k.a client code) for it in C#. Thankfully, CodingConnected.TraCI.NET[11] offers a client library with almost complete API coverage. This client library offers convenient functions to connect to a sumo server and get data from it using data as explained in TraCI sections 3.4. Though the coverage of this library is almost complete, context subscriptions were not implemented. As it will be explained in the Performance chapter 5, context subscriptions are very important for improving the performance of the data retrieval. For this reason, the project was forked and the context subscription functionality was implemented. After implementing this functionality a pull request

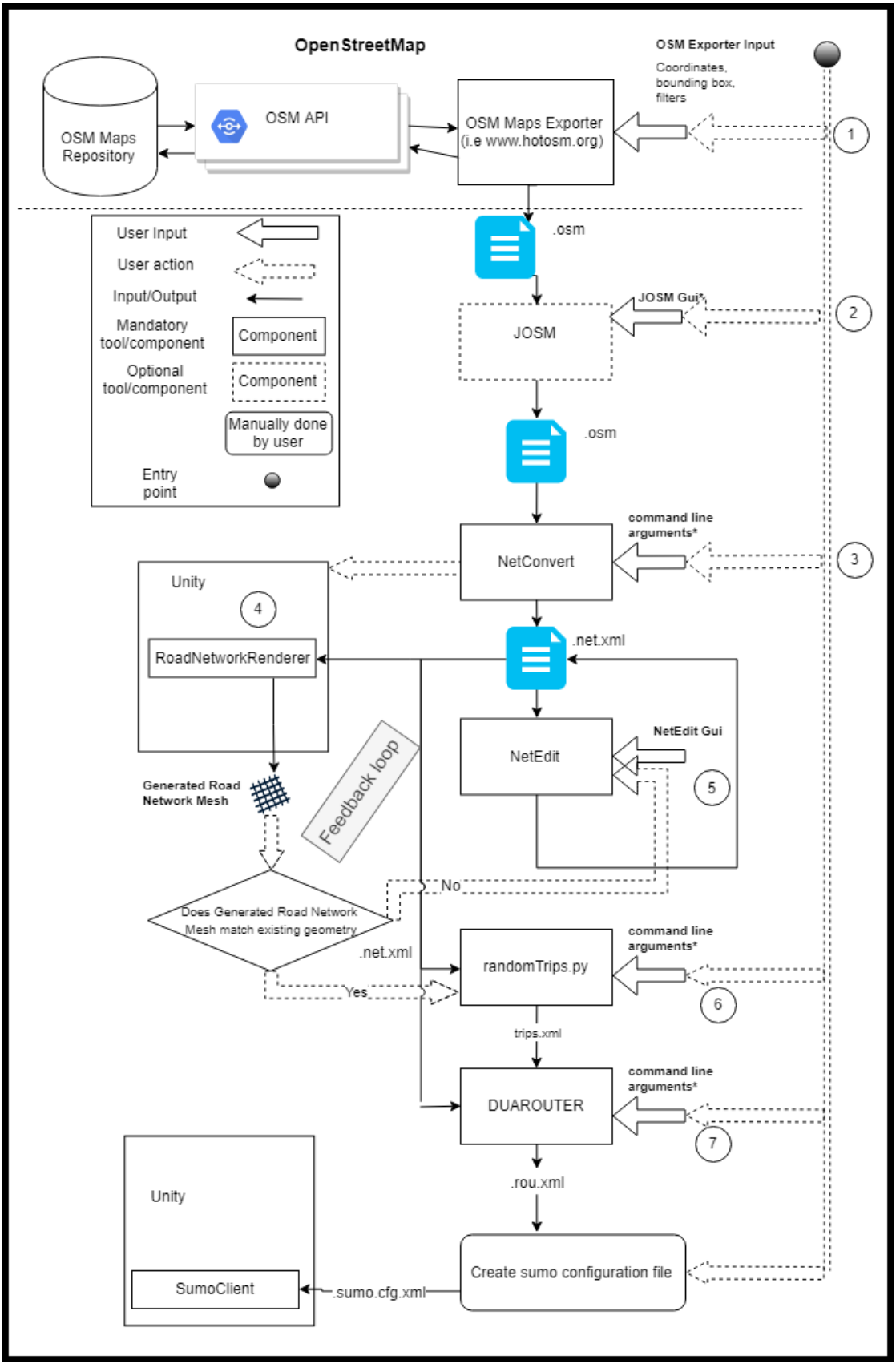


Figure 6: The process from getting an osm file to creating a sumo configuration file (sumo.cfg)

was made to contribute to making the library more complete. Consequently, the functionality was merged.

4.5 Application Manager

This component is used to invoke the client simulation step every `simulationStepLengthTime` seconds. Also creates the sumo server (by serving it using **`SumoClient`**) and connects the **`SumoClient`** to it. The sumo simulation is served with sumo.exe (or sumo-gui.exe) with the argument –step-length `simulationStepLengthTime`. That means that each step the simulated time passed is equal to `simulationStepLengthTime`. Each simulation steps happens every `simulationStepLengthTime` seconds in order to match the simulation time with the real world time so that the cars move at realistic speeds.

4.6 Sumo Client

This is the component that is responsible for creating a sumo simulation server, connecting to it and polling data from it. Sumo Client can connect to an existing Sumo Connection if that is available without it having to instantiate a new Sumo Simulation. This client should provide vehicle data regardless of the strategy that is chosen to be polled as described in section 4.4. In other words, for each way of polling the data **`VehicleSimulator`** or any other component that needs to read the data shouldn't change. The data retrieved from are wrapped in **`Vehicle`** instances which are simple data holders. This component is responsible for updating these instances with the data from the simulation.

4.7 Vehicle Simulator

This component is responsible for creating the car game-objects, attaching the right **`Vehicle`** reference to it's components and configuring them. It is responsible for adding and removing the cars from the unity simulation when their corresponding vehicles depart or arrive to their destination. When using context subscription it creates the car when the corresponding vehicle enters the context range and then handles the disposal of the car when the vehicle exits the context range.

4.8 Strategies for vehicle movement

Several strategies to visualise the vehicle movement were attempted in order to fix problems with performance, realism and synchronisation. Each strategy has its pros and its cons.

4.8.1 Vehicle Mover

`VehicleMover` is the parent class that is used for moving the car based on the **`Vehicle`** that corresponds to said car. The **`Vehicle`** instance that is assigned to the car is attached by the **`VehicleSimulator`**4.7 and is updated by the **`SumoClient`**. This component is attached to each car gameobject.

4.8.1.1 Simple Vehicle Mover

Whenever both the angle and the position of the **`Vehicle`** instance change the same rotation and translation is applied to the car game-object by applying these values to its transform.

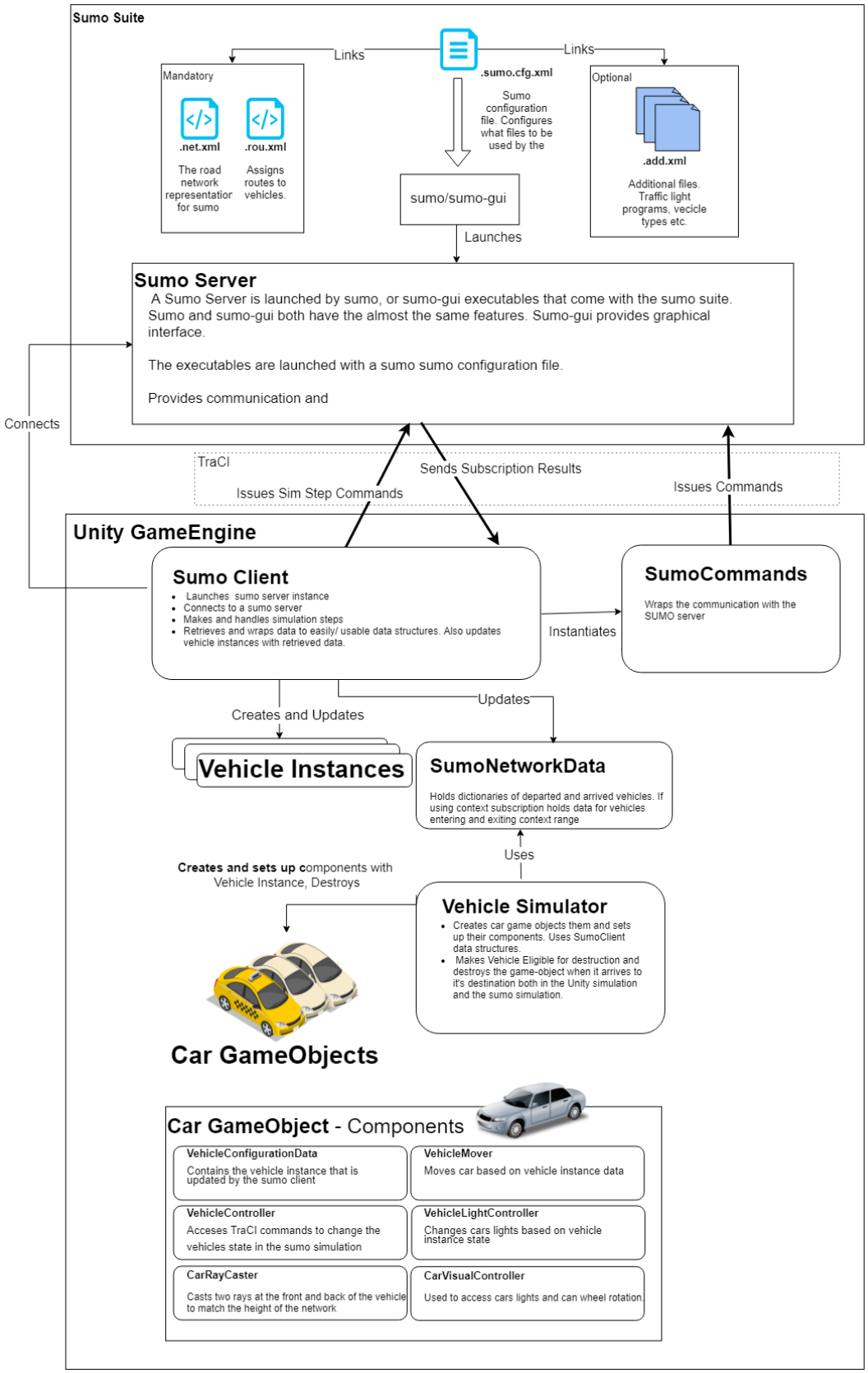


Figure 7: Overview of the different components and how they collaborate to create the interactive visualisation of the sumo simulation in unity.

Pros: * Perfect consistency with **`Vehicle`** instance and therefor with the vehicle in the sumo simulation.

Drawbacks: * Requires very frequent simulation steps with small simulation step length for the movement to be smooth and realistic. If the simulation steps are infrequent the cars seem to teleport. * Wheels spinning and steering must be calculated based on wheel radius and the speed of the vehicle and therefor can produce some unrealistic results.

The method that spins the wheels:

$$float distanceTraveled = speed * Time.deltaTime; float rotationInRadians = distanceTraveled / wheelRa$$

4.8.1.2 Vehicle Mover Interpolation

The main problem with directly placing the car gameobjects is that it requires a high frequency of simulation steps. In order for the frequency to get smaller the step length should be larger. As a result the retrieved position from the sumo server is much further from the current position of the car. For this reason, the positions from the current position to the one retrieved from the server should be interpolated in order to produce a smoother movement. The first position is the current position of the car and the second position is determined by the strategy selected. A number of lag steps can be used for each strategy where the position of the car is `numberOfLagSteps` behind the simulation. The reason for this is to add additional position forward to calculate the curvature for interpolating turns.

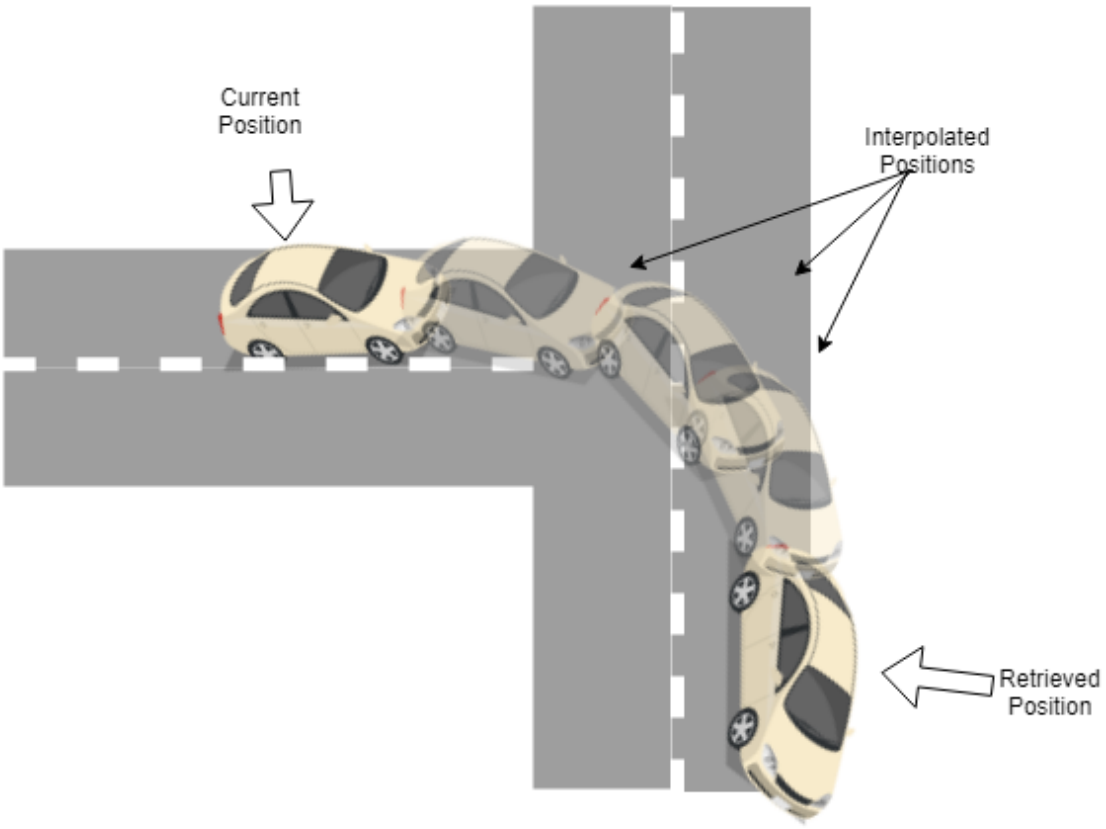


Figure 8: A car using VehicleMoverInterpolation. The car interpolates from it's position the position that was retrieved from the sumo simulation.

Linear Interpolation

With this strategy the position is just linearly interpolated between the current position and `Vehicle.Position` that is the last position polled from the simulation. This happens over `simulationStepLengthTime` in order for the car gameobjet to keep up with its corresponding vehicle in the sumo simulation.

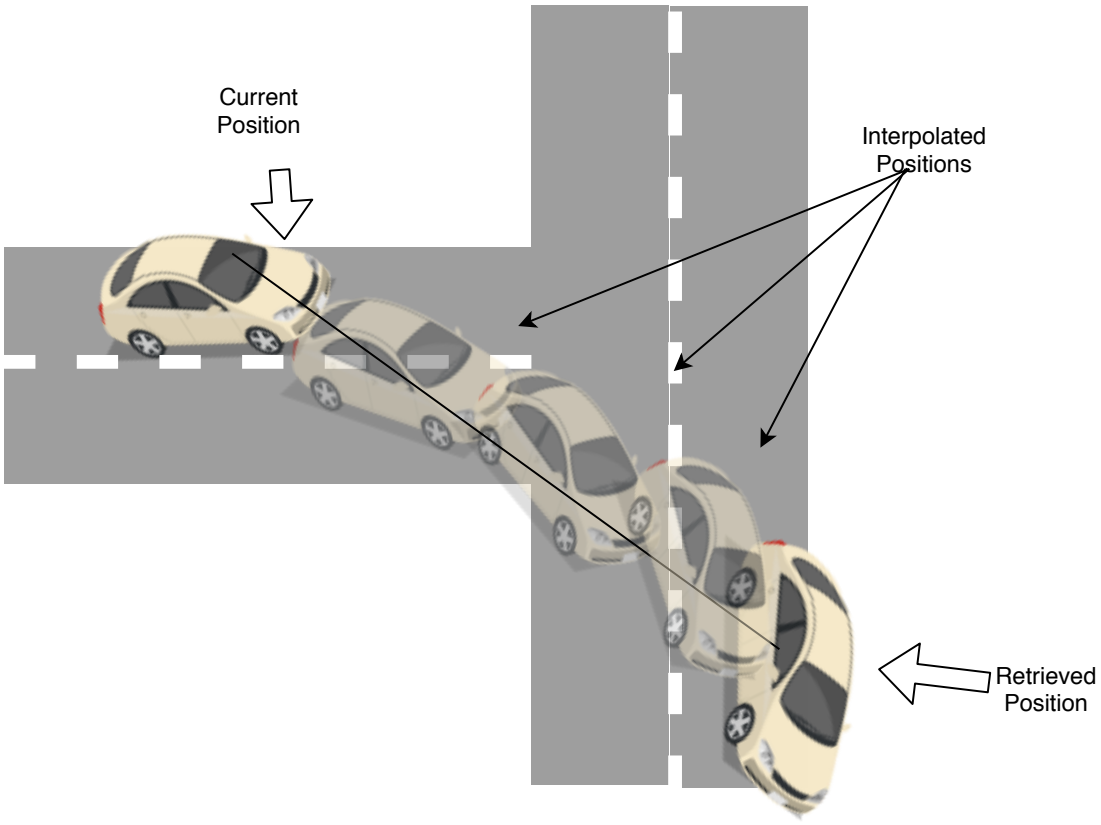


Figure 9: A car using VehicleMoverLinearInterpolation. The car linearly interpolates from it's position the position that was retrieved from the sumo simulation. Due to the nature of the linear interpolation, long simulation steps at road corners produce unnatural (and dangerous) car movement.

Because the interpolation may take a bit more time to complete the sumo simulation may be two or three steps ahead of the car transforms position. As a result, two methods were devised to approach this problem.

The first is to just ignore the fact that the sumo simulation is ahead and just make the car interpolate from its current position to `Vehicle.Position`. By implementing this resolution the car can always keep up with the vehicle in the sumo simulation. The drawback is that the speed may be faster for this interpolation because the car will have to cover a bigger distance in `simulationStepLengthTime`. This can also result in unnatural movement where the car seems to move in discrete steps. This happens when the simulation is always two steps ahead and the car must always cover the distance twice as fast.

The second way to approach this problem is to enqueue any positions that come when the car is interpolating. With this approach the car transform position should interpolate between its current positions and the last dequeued position. As a result this produces a smoother movement. The drawback of this method is that any errors and the time delays accumulate over the time and this has as a result the car to lack behind the simulation in sumo. In order to fix this problem, the real time the interpolation actually takes is recorded in order to adjust the next interpolation to be faster to keep up the unity simulation with the sumo simulation.

4.8.1.3 Vehicle Mover Physics

A third more realistic way to move the car is by using Unity's physics engine. This can be done by attaching a rigid body to the car and provide a wheel collider for each wheel.

Using acceleration and angle

A first attempt was made by just replicating the acceleration from the sumo simulation by getting the value `Vehicle.Acceleration` and `Vehicle.Angle` and applying the right

motor torque to the car with the right steering. Firstly, we calculate force needed with the rigidbodies mass and the retrieved acceleration. Then we calculate the motor torque to apply to each wheel. Lastly, we calculated the steering based on the angle. Because the sumo simulation step is different from the fixed step length, the error for the car position accumulated over the time and as a result the car was at a completely different position than the vehicle. The simulation state in unity was completely inconsistent with the simulation state in sumo.

Car Ai Vehicle Mover

A second more informed way of using physics was attempted. This approach uses Unity's Standard Assets **`CarAiControll`** component that, given a position, will apply the right steering and acceleration to get to it. Using this component whenever a new `Vehicle.Position` was retrieved, the **`CarAiControll`** target was update to that position. The results of this were more realistic and more consistent than with the previous approach but the inconsistencies between the sumo simulation and the unity simulation were still big and unacceptable.

Chapter 5

Improving Performance

5.1 Summary

From early stages it was realised that the commands that are used for polling data from the SUMO server is the bottleneck of the fps count. To understand why the communication is so slow and what affects the communication the most we must first get a better look at the TraCI protocol.

5.1.1 A deeper dive into the TraCI protocol

A TCP message acts as container for a list of commands or results. Therefore, each TCP message consists of a small header that gives the overall message size and a set of commands that are put behind it. The length and identifier of each command is placed in front of the command. A scheme of this container is depicted below:

A TraCI TCP message contains a list of commands or results. Each TCP message contains a header with the length of the whole message. After this header the commands or command

responses follow withe their corresponding length and their identifier. You can find more about the responses of each command at https://sumo.dlr.de/userdoc/TraCI.html.

A TraCI TCP message has the following form:

From all the commands, the ones we care about for the performance of this application is the subscription commands. Both the responses for variable and context subscriptions, can be seen at tables 1 and 2 respectively.

Object ID	Variable Count		
Variable #1 Id	Variable #1 status	Return type of the variable #1	<VARIABLE_VALUE#1 >...
Variable #n Id	Variable #n status	Return type of the variable #n	<VARIABLE_VALUE#n >

Table 1: Variable subscription response

Object ID	Context Domain	Variable Count	Objects Count
Object #1 ID			
Object #1 / Variable #1 Id	Object #1 / Variable #1 status	Object #1 / Return type of the variable #1	Object #1 / <VALUE#1> ...
Object #1 / Variable #n Id	Object #1 / Variable #n status	Object #1 / Return type of the variable #n	Object #1 / <VALUE#n> ...
Object #m ID			
Object #m / Variable #1 Id	Object #m / Variable #1 status	Object #m / Return type of the variable #1	Object #m / <VALUE#1> ...
Object #m / Variable #n Id	Object #m / Variable #n status	Object #m / Return type of the variable #n	Object #m / <VALUE#n>

Table 2: Context subscription response

It can easily be observed that the size of each response is proportional to the amount of vehicles retrieved and the number of variables that were subscribed to. The result for each subscription variable subscribed is the Variable Id, the variable status, the return type and the value itself. The number of variables subscribed is the same as Variable Count that is retrieved from the response. For the variable subscription, if n variables are subscribed, then for each active vehicle in the simulation the TCP message will contain n results. For the context subscriptions, the same applies but for only the vehicles that are inside the context range. Using a socket to read data from a TCP connection and using the loop-back connection is equivalent to reading from a file. As a result, the larger this file is the more time it will take for the message to be read. Since this happens each sim step (because this is how subscription responses are retrieved) it is ideal to subscribe to as little variables as possible and retrieve only the responses for the vehicles that the camera can see. For the first part, the positions and the angle are enough for most of our visualisation strategies. Concerning the amount of vehicles that should be retrieved each step they are is not a way to get subscribe only to vehicles that our camera can see but we can achieve a very close approximation by using context subscriptions.

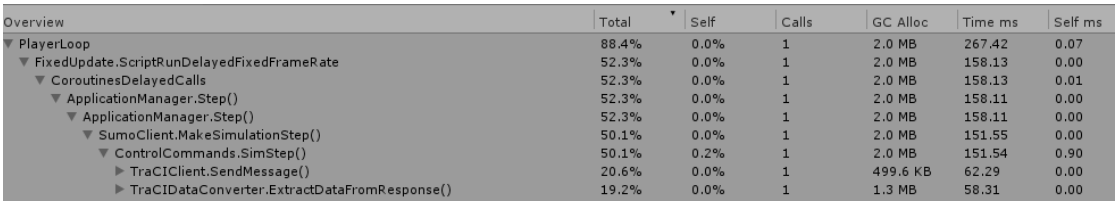
5.1.2 Challenges with reading from TraCI socket

It should be stated that, the socket read can only be a blocking operation because, at the time, this is how all the TraCI implementations are build. Another important factor that limits our flexibility is the fact that although a sumo server can support multiple clients, the clients must communicate with the server in a round robin fashion. This means that if two clients use the server then the first must communicate with the server and then the second and then the first one again. All the communications must end with a client sim step. The sim step could be given a negative value as to not change the simulation state. What is more, once at least one client connects to the server then sumo can only be issued to make a simulation step from the client(s). As a result, the sumo server doesn't support concurrency by design since it just can't run in the background providing it's state whenever a client requests it.

5.2 Profiling and testing setup

Performance is a huge issue when using SUMO with Unity. That is because SUMO runs it simulation as a server and one must connect to it and communicate via TCP commands. Even when SUMO runs on localhost it is pretty slow and therefor optimisation should take place to make the communication as light and fast as it can get.

This was realised very early by using the Unity Profiler. The fps count would plummet to a low amount (i.e 10 fps) with a small amount of vehicles (i.e 20 vehicles) as it can be seen in figure 10. We can see that most of the time is spent reading and extracting the data coming from the SUMO server.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
PlayerLoop	88.4%	0.0%	1	2.0 MB	267.42	0.07
FixedUpdate.ScriptRunDelayedFixedFrameRate	52.3%	0.0%	1	2.0 MB	158.13	0.00
CoroutinesDelayedCalls	52.3%	0.0%	1	2.0 MB	158.13	0.01
ApplicationManager.Step()	52.3%	0.0%	1	2.0 MB	158.11	0.00
ApplicationManager.Step()	52.3%	0.0%	1	2.0 MB	158.11	0.00
SumoClient.MakeSimulationStep()	50.1%	0.0%	1	2.0 MB	151.55	0.00
ControlCommands.SimStep()	50.1%	0.2%	1	2.0 MB	151.54	0.90
TraCIClient.SendMessage()	20.6%	0.0%	1	499.6 KB	62.29	0.00
TraCIDataConverter.ExtractDataFromResponse()	19.2%	0.0%	1	1.3 MB	58.31	0.00

Figure 10: Making a simulation step takes the most time

To improve performance and track how the performance increases with each iteration of improvement the same sumocfg 3.6.1 file was used for all tests. This configuration file references a big road network network that was created using NETGENERATE3.2.1.2 using the command:

Listing 5.1: Generating network for testing

```
--spider.arm-number 15      \
--spider.circle-number 15   \
--spider.space-radius 10    \
--spider.omit-center true   \
--lefthand                  \
```

This commands creates a spiderweb-like road network that is big enough to support more than 5000 vehicles, which is more than enough for our performance tests. Afterwards, a route file3.6.2, that the configuration file also references, was created using randomtrips.py3.2.1.5.

The command used is:

Listing 5.2: Generating rou.xml

```
randomTrips.py -n <.net.xml> -b 1  -e 101 -p <period_of_vehicle _departure>
-r rou<100/period_of_vehicle_departure>.rou.xml
```

i.e by doing:

Listing 5.3: Route file with 1000 vehicles example

```
randomTrips.py -n net.net.xml -b 1 -e 101 -p 0.1 -r rou1000.rou.xml
```

A vehicle will depart each 0.1 seconds for the period 1 to 101. Thus we will have 1000 cars departed by the 101th second.

It was decided that this process should be done for 1 5 10 50 100 250 500 750 1000 2500 5000 7500 100000 vehicles. In reality we only care for up to 2500 vehicles.

For each test sumo.exe (no GUI) was used with a step length of **0.05**ms.

Listing 5.4: Sumo Execution command

```
sumo -c <run<num_of_vehicles>.sumocfg> --step-length 0.05
```

Note that this server is launched from the C# code in Unity as a Process.

For the performance tests it was decided that nothing will be rendered and thus VehicleSimulator4.7 would not be used. The only component that will be used is **`SumoClient`** 4.6 that is responsible for polling the data from server. This way we can isolate the tests to only testing how well we can poll data from the sumo server.

In addition it must be noted that a build version of the simulation in Unity was used for testing the performance. The comparison measurement used is the fps count by the number of active vehicles. Each step all vehicles that are active in the road-network are polled for their rotation and their position among other information. On that note, the amount of variables that are retrieved significantly affect the time that the TCP socket takes to read them.

CodingConnected.Traci.Net To use the TraCI protocol one can either build a system from scratch or use an existing implementation. The most complete one that is constantly updated is

the one using python. An attempt was made to use IronPython that lets you use python from C# but it was quickly discovered that using IronPython the sumo tools that use python such as TraCI do not work correctly. For this reason, it was decided that a C# implementation of TraCI should be used so we can use it directly in Unity. Luckily, there is a incomplete but growing client library written in .NET, TraCI.NET[11]. The repository had a significant improvement from the time this thesis begun so a lot of features were added as the thesis was being developed. This library implements most of the functionality that is needed but not all that our project requires. For this reason it was decided that a fork should be made and any additional functionality that is required to improve performance should be implemented. The most important functionality that was added was that of Context Subscriptions. For more information about TraCI subscriptions please refer to the section 3.4.1.

Context Subscriptions could be proven very efficient for our performance since given the camera in unity, only the vehicles can be observed from the camera should be retrieved. There is no perfect way to get only the vehicles that will be visible but with context subscription a very good approximation can be achieved. As a result, the amount of vehicles that are retrieved each step is much less than the current number of active vehicles the size of the TCP packet that would have to be read from the socket would be less. Consequently the amount spent for the TCP read will also be less.

With this approach a new challenge arises. The question is what lane should be subscribed to get the best results? Optimally, the best choice would be a lane that is central enough and with a context range that will retrieve information only for vehicles that are inside the camera's frustum. This would be trivial if the camera was static or with a predefined path since we could assign

the lanes, the subscription time and context range for each lane manually beforehand. For the purpose of this thesis, it is required that the camera should be moving and it's movement should be dynamic. For this reason, a system should be developed that selects the best lane and the best context range each time the camera moves. The system should not affect the performance and should optimally subscribe to a lane and with a context range that retrieves information with tight fit.

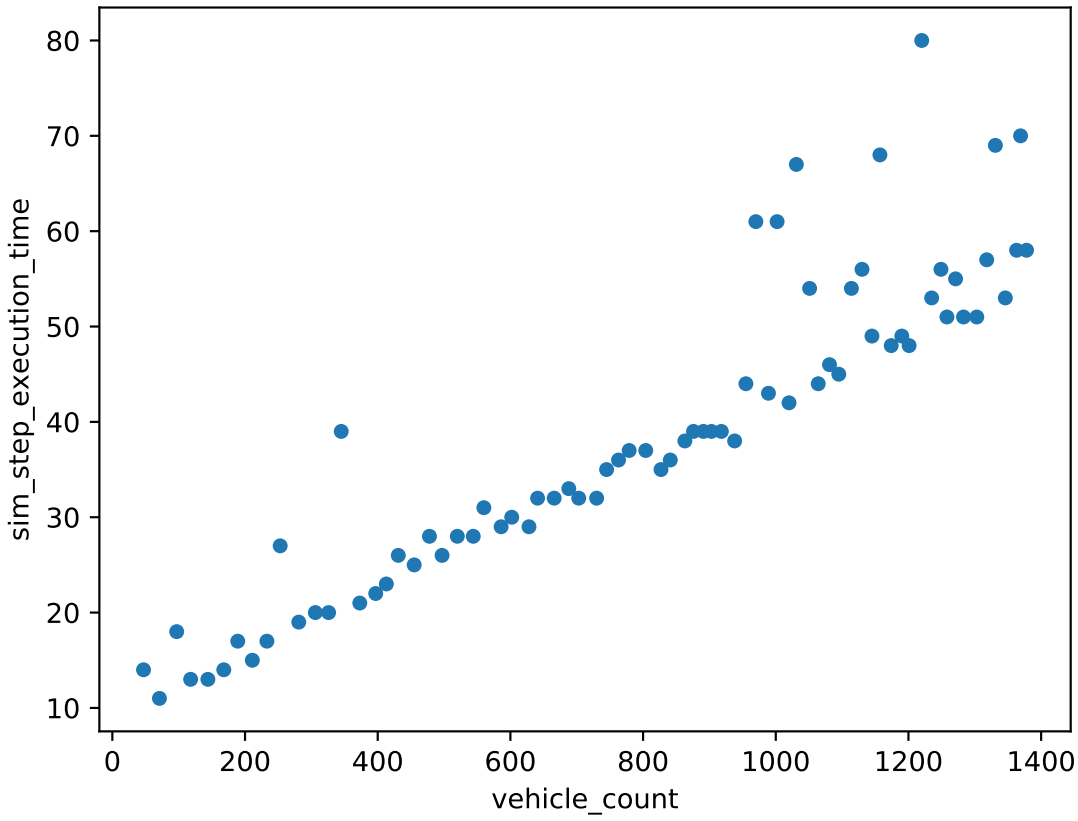


Figure 11: Execution time of sim step. No value retrieval (no get commands, no subscriptions). Used as control. We can observe that the simulation step length increases linearly even when no data is retrieved. This is because the sumo simulator server must calculate the state of the new step for each active vehicle.

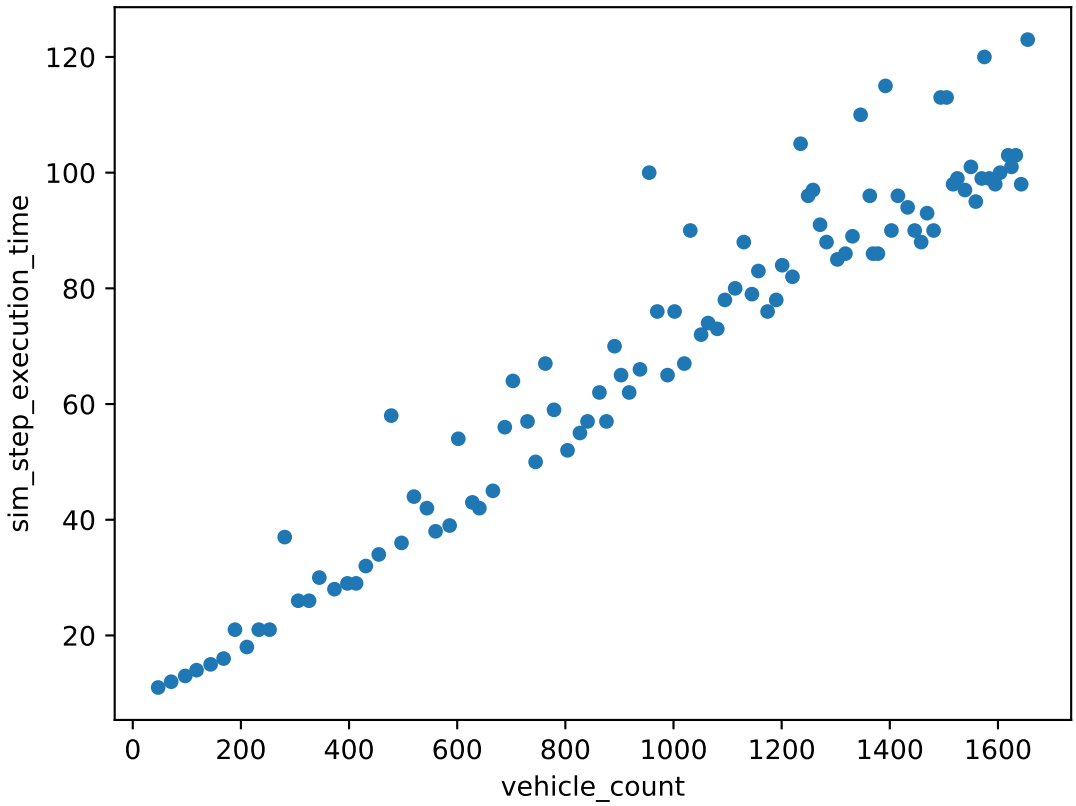


Figure 12: Execution time of sim step. Context subscription, context range 100, maximum 525 vehicles inside context range. Linear increase of simulation step execution time is observed. Just 120 ms for 1600 active vehicles in the simulation.

It was decided that the best way to subscribe to context is by using lanes. As it was described in section 4.2, for each lane parsed from the net.net.xml a mesh is created and a GameObject along with it. With this mesh a collider is also created. This collider is intended for collision for vehicles with physics, pedestrian movement and for triggering events on collision. For the purpose of context subscription multiples methods were tried.

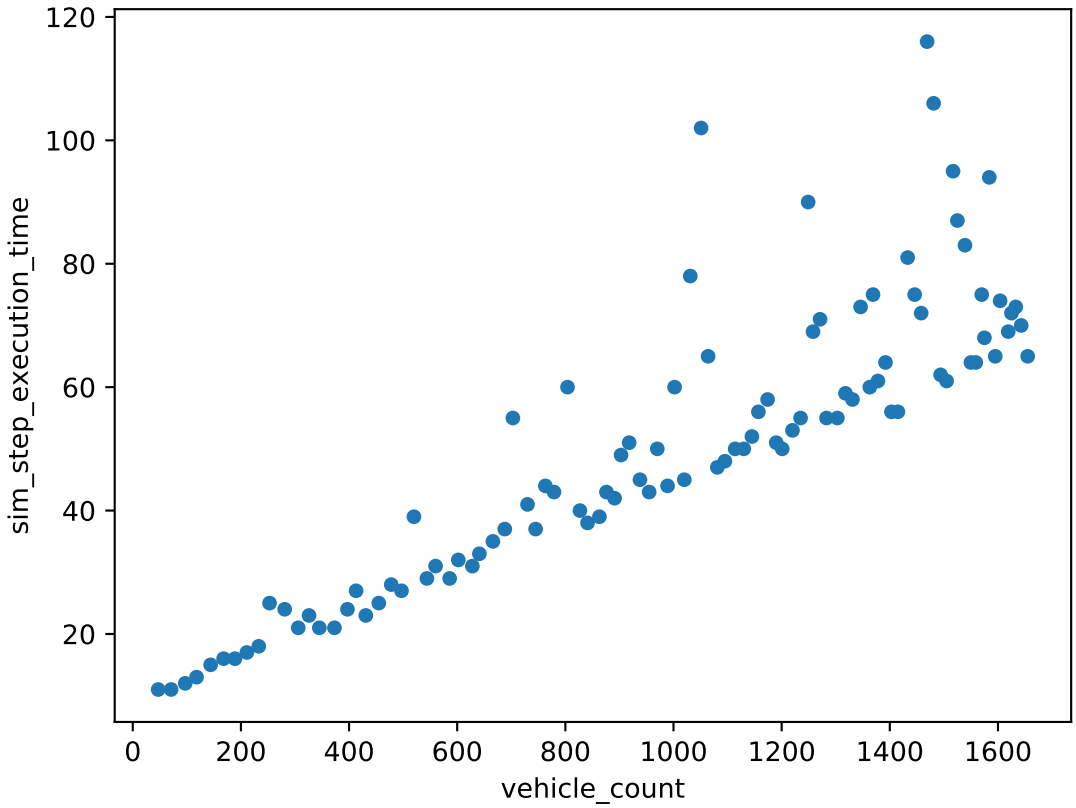


Figure 13: Execution time of sim step. Context subscription, context range 30, maximum 50 vehicles inside context range.

5.2.1 Lane selection for context subscriptionj

RayCast disk

As a first approach for the lane context subscription, raycasting was used. Each time a ray hits a new lane the lane should execute a context subscription for the vehicle context domain to obtain the vehicles that are inside the context range and to obtain their corresponding position and angle. Each time only one lane should have a context subscription. The context range should match what the camera sees. For this purpose the setup shown in figure 15 was implemented. A

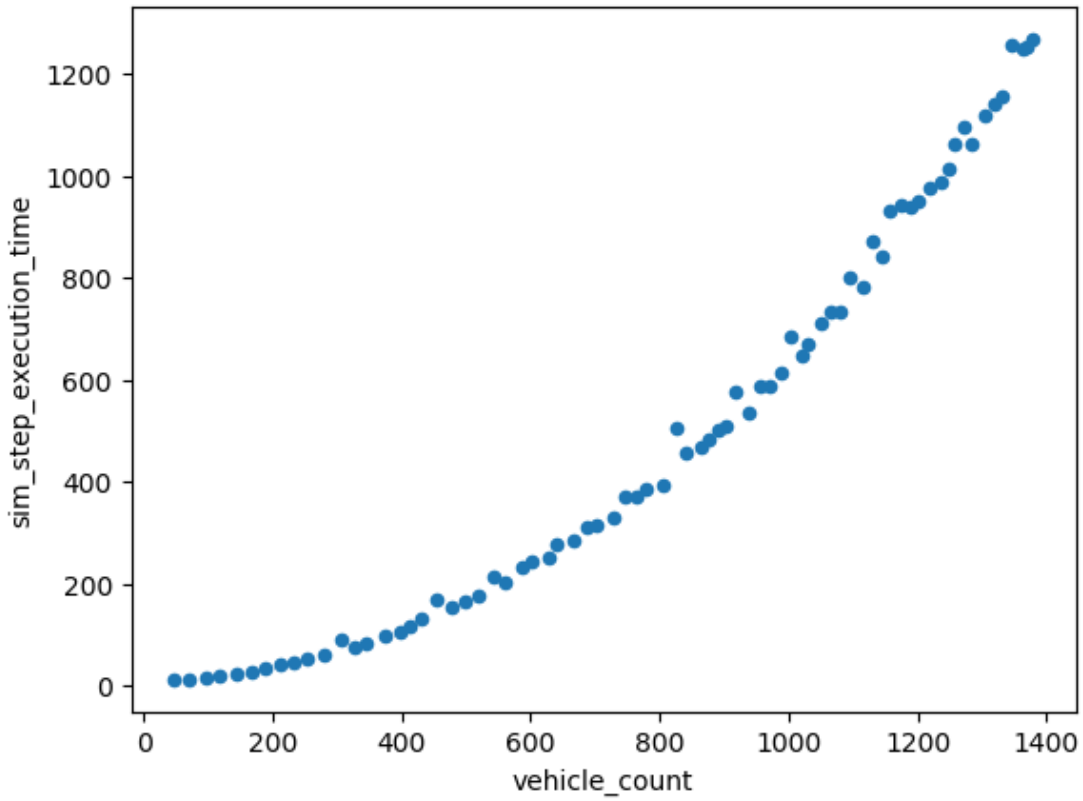


Figure 14: Variable subscription. We can see that the increase in time is quadratic. It takes 1.2 seconds to make a simulation step with 1400 vehicles.

disc of rays shoot in Y axis and a fixed distance ahead of the camera. The reason for shooting multiple rays instead of one is because a single ray can at often times miss our network and the lane game objects. The disc is always perpendicular to the xz plane where the Cars move. The circle of rays grows by lane width (to avoid skipping a lane) until it hits a lane. When the camera moves the test starts again.

This method was discarded in light of better solutions.

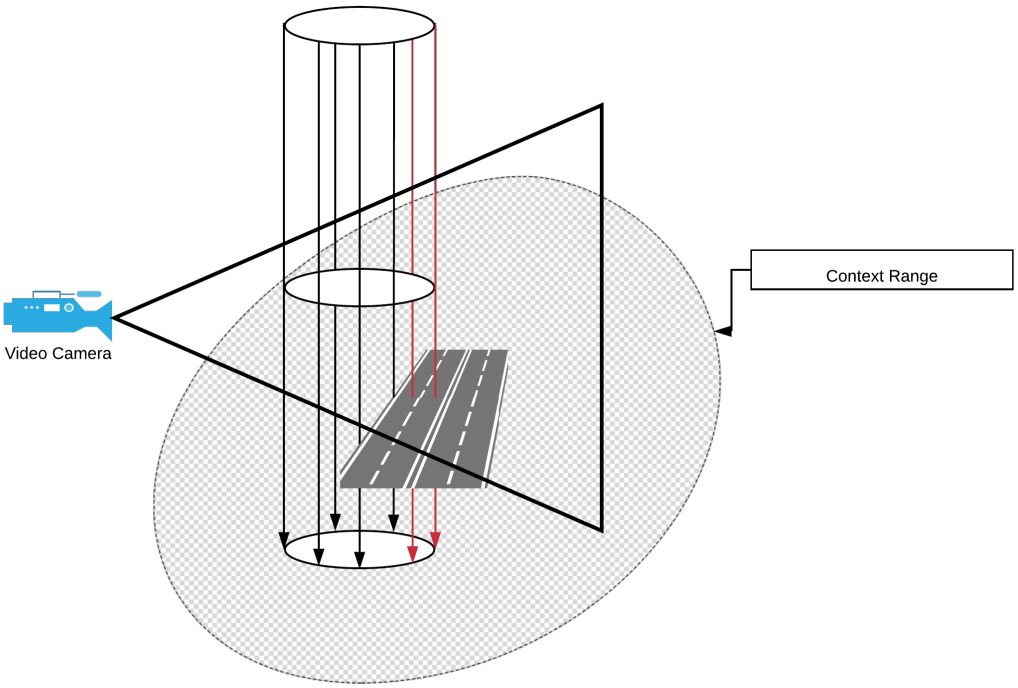


Figure 15: Lane selection for context subscription using a disk of rays cast downwards from the middle of the camera with the purpose of selecting a lane that is relatively in the middle of the camera frustum for context subscription.

Sphere Cast

Another similar approach for selecting a lane to subscribe, is by using Unity's SphereCast. A sphere-cast is like a ray-cast but instead of casting a ray, a sphere with a certain radius is cast. For this approach it was decided that the sphere would be cast from the top of the middle of the camera frustum. The sphere-cast is tested multiple times from smaller radius to bigger radius

until the sphere collides with a lane collider, at witch point the context subscription happens for that lane, or until the maximum radius is reached where it is assumed that no lanes are in the camera frustum.

Lane Visibility

A third and more robust approach is using Unity's Renderer's component isVisible property. A renderer component is what makes objects visible in a scene. After some testing, it was realised that a renderer isVisible is true even if the mesh is occluded or back face culled. This was perfect for our purposes since only the lane gameobjects that are inside the camera's frustum have this property as true. The following script was added to each lane gameobject.

```
...
private Rendered _renderer;
private Lane lane;
...
private void Start()
{
   ...
   _renderer = GetComponent<Renderer> ();
   ...
   \\ The visibility test isn't necessary to occur every update, so we use
   \\ a coroutine that executes every SOME_TIME
   StartCoroutine(VisibilityTest());
```

```
}

pivate IEnumerator VisibilityTest()
{
  while (true)
  {
    if (_renderer.isVisible)
    {
      if (!networkData.LanesInsideFrustum.ContainsKey(lane.ID))
      {
          // Adds self to visible lanes
          networkData.LanesInsideFrustum[lane.ID] = Lane;
      }
    }

   // When stop being visible must remove self
   // from visible lanes
    if (!_renderer.isVisible &&
    networkData.LanesInsideFrustum.ContainsKey(ID))
    {
      networkData.LanesInsideFrustum.Remove(ID);
```

```
        }
        yield return new WaitForSeconds(SOME_TIME);
    }
}
```

`networkData` is a ScriptableObject that is created as an asset and basically is a singleton that holds data that are shared accross the project. `LanesInsideFrustum` is a shared dictionary that contains all the lanes that are currently inside camera's frustum. It contains key value pairs where the ID of a lane is the key and the Lane instance is the value. [1] Whenever a lane's renderer `isVisible` it adds itself in `LanesInsideFurstum` after making sure that is not already contained in it. If the renderer is not visible and is inside frustum it is removed. With this script attached to each lane gameobject we can make sure that LanesInsideFrustum contains only the `Lane` instances who are attached to gameobjects that are inside the camera's frustum. One think to watch out is that `isVisible` is true even when the renderer is inside the editor's camera frustum. However, this is not a problem since in the final build only the main camera will be used. Moving on, to actually select the lane another script attached to the camera is used.`LaneContextSubscription` is a script that is attached to the main camera with the purposed to find the most central lane among the lanes that are inside the camera's frustum and to subscribe to it. To find the most central lane, a gameobject is instantiated as a child to the camera and is set to be at the centre of the camera's frustum. This gameobject is referred as `CameraFrustumCentre`. Because this gameobject's transform is a child of the camera it

[1] This pattern is used heavily in this project, where dictionaries store the id of the sumo type (such as a vehicle, lane, edge) and the corresponding C# instance created as the value

will be always be relatively to the same place for the camera. The purpose of this gameobject is to keep track of the camera's frustum centre. Once this gameobject is created, another coroutine is started that finds the closest Lane to the `CameraFrustumCentre`. Only the xz plane is considered for the distance. It is important to note that this test happens every some time since it would be meaningless for this method to be performed each update since a) the camera doesn't always move and b) the camera moves smoothly. For these to reasons this test happens only whenever the camera moves, or, when the camera is constantly moving, periodically. This approach provides us with reliable and consistent results but does not give much control to the `LaneContextSubscription` since it is always executed periodically.

A cleaner solution for the same problem, is by using events. The script that is attached to each lane gameobject is:

```
...
private Rendered _renderer;
private void Start()
{
    ...
    _renderer = GetComponent<Renderer> ();
    // subscribe to requset for visible lanes
    networkData.RequestVisibleLanes += RequestVisibleLanes();
    _AddLaneToVisibleLanes;
    ...
}
```

```
...
private void RequestVisibleLanes_AddLaneToVisibleLanes(object sender,
LanesInsideFrustumEventArgs e)
{
    if(_renderer.isVisible)
    {
        // Add self to visible lanes
        e.LanesInsideFrustum.Add(Lane);
    }
}
```

The event `RequestVisibleLanes` is triggered whenever someone requests to get all the lanes that are visible. For our purposes this is triggered by the script that is attached to the main camera. This script looks like this:

```
private void ContextSubscribeToLaneInFrustum()
{
  int numOfIntersectionsWithPlane =
  _cameraIntersect.FindIntersectionsWithPlane(
    out Vector3[] intersectionPoints
  );

  var positionToGetClosestLaneFrom = new Vector3();
```

```
float contextRange = 0f;

// if frustum intersects with the xz plane then we can
// know the optimal context range and the optimal centre
if (isCameraFullyIntersectingWithPlane)
{
  // Calculate the minimum enclosing circle
  //of the four intersection points
  var minimumEnclosingCircle =
  CalculateMinimumEnclosingCircle(intersectionPoints);
  positionToGetClosestLaneFrom = minimumEnclosingCircle.center;
  contextRange = minimumEnclosingCircle.radius;
}
else
{
  // If no intersection points with the xz plane where the cars move.
  positionToGetClosestLaneFrom = cameraFrustumCentre;
  contextRange = main.camera.farClippingPlane;
}

// Trigger the event that requests for all visible lanes.
```

```
    List<Lane> lanesInsideFrustum = networkData.RequestForVisibleLanes();

    closestLane =
    FindClosestLane(lanesInsideFrustum, positionToGetClosestLaneFrom);

    // Subscribe to new lane and unsubscribe from old one
    ...

    // Update currently subscribed lane
    ...
}
```

There are a few things to note here. As previously stated, the optimal lane for the context subscription must be selected to receive the just the vehicles that are inside the camera frustum. In the script above we can see that there is a component called `CameraIntersect`. This component is attached to a the camera and can be setup through the unity editor or by a script. The component is setup in such a way to retrieve the intersection points of the camera's frustum with the xz plane where the cars are moving. This component is very helpful since, whenever the camera frustum intersects the plane we can calculate the exact context range and the centre of the circle that would give us the optimal results. To find out if and where the camera intersects the xz plane four points are needed since the frustum is a rectangular pyramid. `CameraIntersect` casts four rays from the main camera to the four edges of the frustum pyramid with range equal

to the camera's far clipping plane. By knowing the height of the xz plane where the cars are moving, it can be easily calculated where and how many rays intersect the plane. If all four intersect, the frustum fully intersects the plane. Moving on, to calculate the centre and the radius for the context range, an algorithm that calculates the minimum enclosing circle for a set of points is used. This algorithm returns a centre with a radius. Unfortunately, there is no context subscription that can take only a position and a context range and so the closest lane to the centre of the circle must be found. After this process the subscription occurs making sure to unsubscribe from any previous lane. If the frustum does not fully intersect the xz plane, then the camera can see as far as the horizon, or as far as the clipping lane to be more precise. In this case, the `LaneContextSubscribe` script falls back to the previous solution where the lane closest to the camera frustum centre is used for the context subscription. For the context range it is a matter of whether accuracy or performance is prioritised for the application. If accuracy is more important than the performance then the context range can be set to be half the far clipping plane. With this, all the cars that should appear in the camera frustum will be visible. If performance is prioritised, then one can either shorten the camera's clipping plane or select a max distance for the context range. The latter is better in dense city scenes where the camera can not see far into the distance and so the accuracy of the visualisation is not affected.

What is more, context subscriptions can be applied to several sumo elements including vehicles. For this reason another system was implemented where we can follow a car and subscribe to the corresponding vehicle in the simulation. This task is easier since the camera follows a car and we only have to context subscribe to that vehicle with a context range we want.

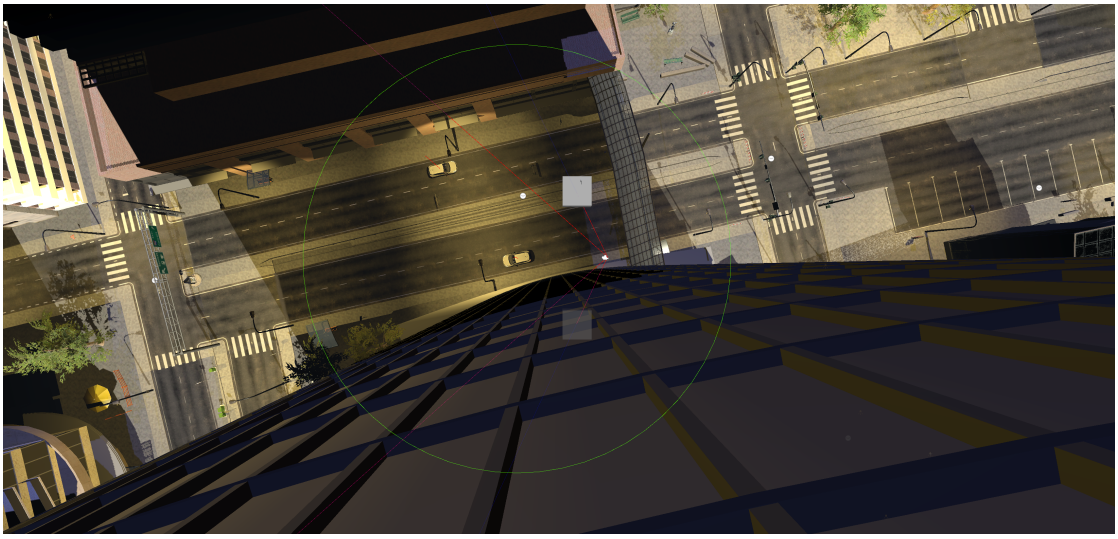


Figure 16: Vehicle context subscription. The camera follows a car and a context subscription exists for the corresponding vehicle, to bring data for the vehicles that are inside the context range.

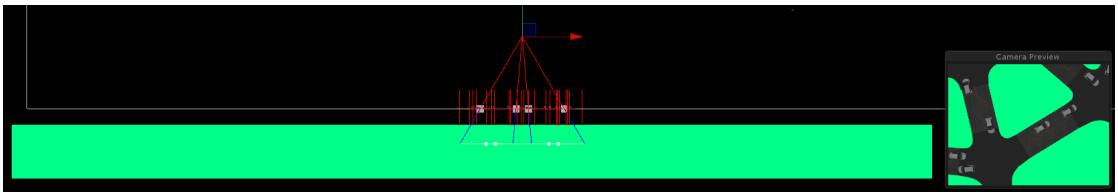


Figure 17: A 2D scene view of the context subscription results. The green rectangle is the length of the whole road network. Each red line represents a car. It can easily be observed that only what the camera can see is received from the sumo simulation. This results in a lighter socket communication between the client and the server.

This context subscription system can be extended and be used with other sumo elements as well, such as points of interest, as long as the corresponding visualisation parts and the corresponding controllers are creates in the game-engine side. The **`SumoClient`** and classnameVehicleSimulator collaborate to create and update only the cars that are inside the context range.

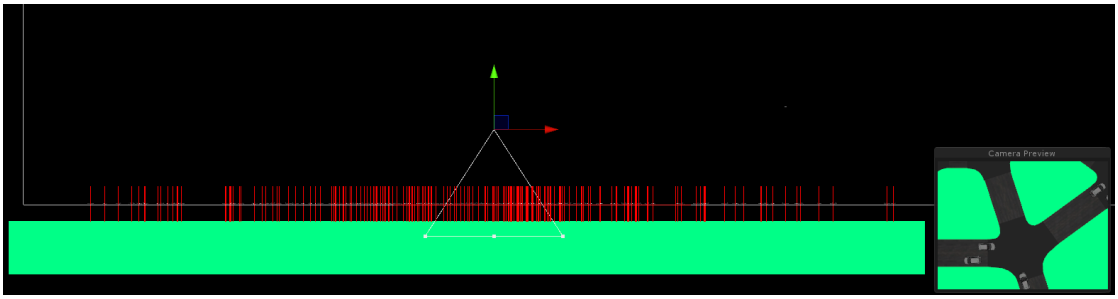


Figure 18: A 2D scene view of the variable subscription results. The green rectangle is the length of the whole road network. Each red line represents a car. We can see that most of the cars are outside of what the camera can see. As a result, a lot of additional overhead is added to the communication layer since much more data must be send via the socket.

Results With the subscriptions the main goal is to reduce the number of tcp requests and to reduce the size of them so the socket would have to read less. Variable subscriptions significantly improve the performance than by using individual Get commands for each vehicle because with the second method more TCP requests are sent (one for each Get command). With Variable subscription however only one TCP request happens per simulation step. The performance gain is big but the simulation was still slow, especially when big amount of vehicles exist in the network. This is due to the fact that more data should be handled by the socket. The fact that at any point only a subset of the city is visible is ideal for using context subscriptions. The context subscription system that TraCI provides was adapted for our needs so only the vehicles who can have a car representation that can be seen by the camera are received. This reduces number of data that have to be communicated from the socket by the sumo server to the client and, as a result, leads to big performance gains with up to four times more fps, especially in big networks where the camera sees only a very small subset of the network.

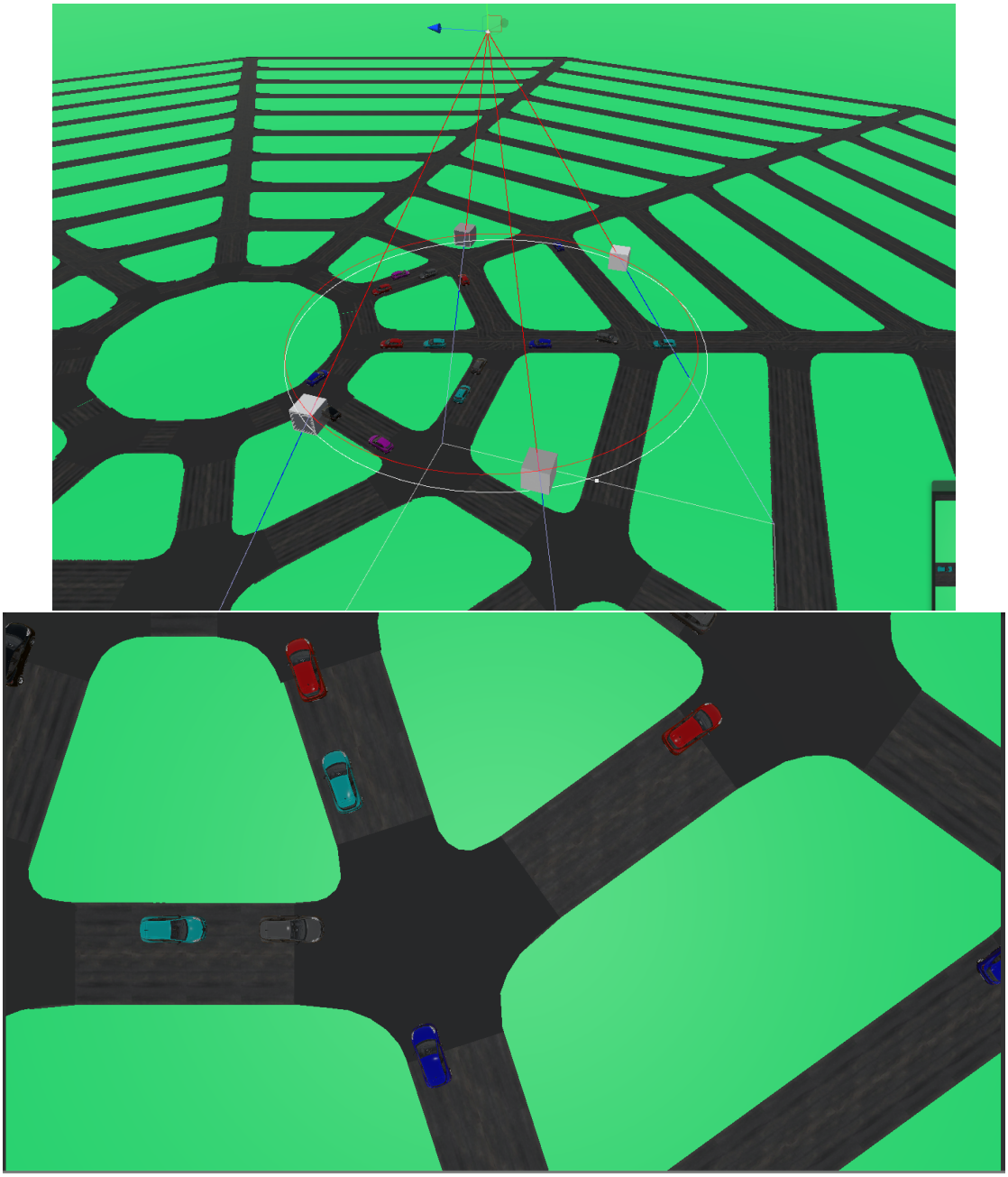


Figure 19: Scene and game view of the 3D simulation state using **context** subscription. Only the vehicles that are inside the context range in the simulation side are visualised in the game-engine. Because of the way the context subscription happens only the vehicles that can have a car representation inside the camera view are retrieved. Less amount of data communicated through the socket leads to a faster communication and higher fps.

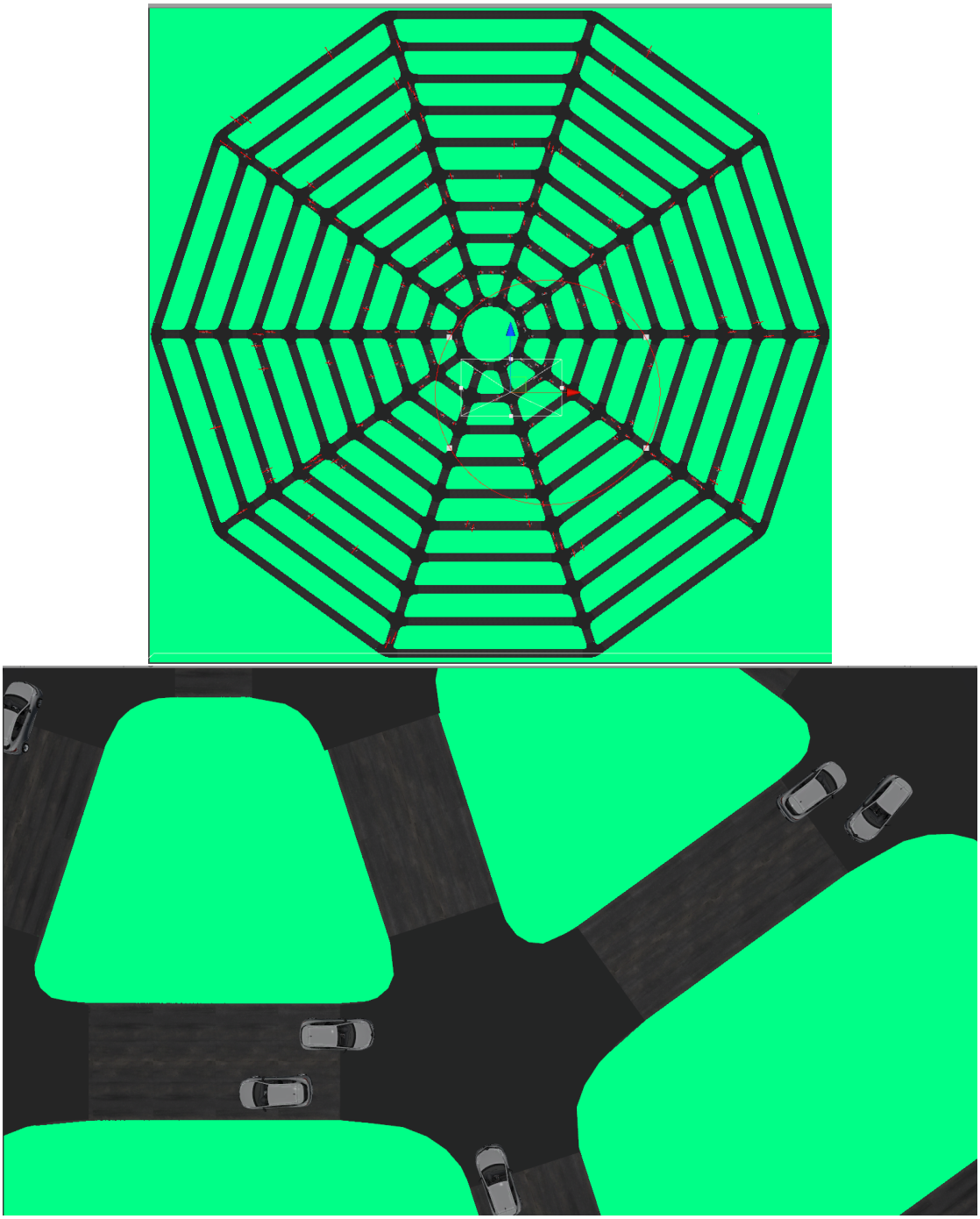


Figure 20: Scene and game view of the 3D simulation state using **variable** subscription. Red lines in scene view represent the cars. We can see that much more vehicles are received than what the camera can see. Consequently, the communication is heavier.

5.2.1.1 Multithreading

The need for more than a single thread From early on it was realised that using a single thread for polling data from SUMO was blocking the main thread which had as a result a low fps count with a low amount of vehicles. Because the sumo server is run on local host it is relatively much faster than reading from a network but still too slow for a real time application such as using sumo in unity. That is because using socket read is like reading from a file which is very slow. For this reason it was decided that the heavy lifting of reading from the socket should happen from a different thread.

Implementation The SUMO server itself does not support multithreading since if we want to have multiple clients connected to it each client take turns communicating with it and each package must be sent completely before it moves to the next command. For this reason it is up for the client to do it's own multithreading. Using subscriptions (both variable and context subscriptions) the most heavy lifting of the communication happens in sim step since this is where all the subscription data is received from. Any additional communication that happens Using unity profiler this was further confirmed. For this reason it was decided that the sim step that happens every step length (this decision is to synchronise the car movement in unity with the vehicle movement) should run on a different thread. !WARNING! when using multithreading then each time a command is issued the client must me locked by clientLock.

Results Using multithreading creates more problems than it solves because of problems with consistency and synchronisation. We believe, that it is possible to use multithreading for

our application but requires a considerable amount of time that was not available during the development of this thesis. Instead of multithreading, it is advised to first attempt to use libsumo, which is the TraCI API but in C++ and runs without a server.

5.3 Libsumo

Communicating with a server, even if done locally and even with optimisation will always be slower than directly using a library. Fortunately, as explained is section 3.5, there exists a TraCI API implemented as a library in C++. In order to use this functionality a dll must be created wrapping the functionality to be used by C#. This would mean a lot of boilerplate code but thankfully, the TraCI API was created in such a way that is easy to create a dll of any subset of the library using SWIG. The sumo team already provides bindings for python and java but not for C#. The project in unity only uses a small subset of the commands that TraCI provides. Consequently, a dll should be created for the subset of commands used in the unity project.

An attempt was made to use libsumo and to create a C# dll with this subset of commands. In order to test if libsumo really did improve performance, we used the generated bindings (created by swig) for python. To achieve this, we must build sumo from source and use the SWIG flags in CMAKE. After this, we checked the performance when using libsumo and when using TraCI for the same code in python. The results showed that using libsumo is twice as fast than using TraCI to communicate with the sumo server through a socket.

After we tested the performance we checked if libsumo supports all the functionality that TraCI offers. Unfortunately, context subscriptions were not supported and an error was thrown when using them. For this reason, the idea to create C# bindings for libsumo was discarded.

In conclusion we hope that in the future the libsumo library will have all the functionality that TraCI offers. We estimate that this will be the case in the future and that the creation of the C# bindings will be worth the efforts as it would offer a significant boost in the application performance.

A note for creating C# bindings for libsumo

For anyone that will attempt to create the C# bindings, it is advised to read the SWIG documentation and do the tutorials. To create the bindings an interface file (.i) is needed to tell SWIG what functions to create bind for. SWIG then goes and creates an Interop code for C# that allows the use of C++ code. Fortunately, this interface already exists because it is used for python and java. This same interface file can be used for C#. To do this, one must read the documentation and make the right changes in the cmakelists.txt file that is inside the libsumo folder.

Chapter 6

Conclusion, limitations and future work

Figure 21: An in-game view of the simulation at Wind-ridge city. We can see, on the upper left part, the **`Vehicle`** instance information. Not all data are retrieved since for performance reasons we only want position and angle. On the upper right part we can control the speed of the car.

The end result of this thesis is a highly extendable 3D interactive visualisation of the sumo traffic simulation. In the demo we can change the speed of the vehicle after selecting the car

and setting the speed. In truth, all the state changing commands are available for the vehicle and can be easily accessed via using the vehicle controller component of the car. In the future, we would want to represent more SUMO elements in the game-engine site. Most importantly, we would like to represent traffic lights and add the ability to control them from the game-engine. Implementing this feature shouldn't be hard because, creating traffic lights, would follow the same MVC (Model View Component) pattern that is used by the vehicles.

Additionally, the techniques to increase the performance of the visualisation and potential performance boosts are discussed. Currently, a context subscription system for vehicles and lanes is used that significantly increases performance. In the future, we hope that libsumo will be more complete and support all the functionality that the TraCI API offers. When this happens, we could create the C# bindings for the libsumo library and then adapt the application to use these bindings. It is estimated that the performance will be

Multiple techniques were used to achieve the movement of cars. Currently direct polling of the data is used since it's the most consistent with the simulation but it requires frequent and small simulation steps. However, the vehicle movement can sometimes be unrealistic. As a result, the movement of the cars can also be unrealistic. SUMO was not designed to be used by a game-engine and some extreme manoeuvres are sometimes allowed. In the future, we would like to extent the interpolation to be more informed. We would want to use a database and take into account the next n positions to calculate a more realistic vehicle trajectory.

In conclusion, SUMO is a a very good simulator that is open source and continually updated but with considerable drawbacks when it comes to using it along with a game-engine. Because of the popularity of the simulator and the fact that a lot of realistic scenarios are continuously

being created for it[7][5][10], we believe that attempts to create an interactive visualisation for the simulation should not be abandoned. This thesis offers a good groundwork and serves as a base to develop a more realistic and efficient traffic simulation, using SUMO, in unity.

Bibliography

[1] Netconvert import OpenStreetMap . https://sumo.dlr.de/wiki/Networks/Import/OpenStreetMap. Accessed: 20-05-2019.

[2] Airsim. https://github.com/microsoft/AirSim. Accessed: 05-16-2019.

[3] Michael Behrisch, Laura Bieker, Jakob Erdmann, and Daniel Krajzewicz. Sumo–simulation of urban mobility: an overview. In *Proceedings of SIMUL 2011, The Third International Conference on Advances in System Simulation*. ThinkMind, 2011.

[4] Laura Bieker, Daniel Krajzewicz, Antonio Pio Morra, Carlo Michelacci, and Fabio Cartolano. Bologna scenario sourceforge download. https://sourceforge.net/projects/sumo/files/traffic_data/scenarios/Bologna_small/. Accessed: 19-05-2019.

[5] Laura Bieker, Daniel Krajzewicz, Antonio Pio Morra, Carlo Michelacci, and Fabio Cartolano. Traffic simulation for all: a real world traffic scenario from the city of bologna. In *SUMO 2014*, May 2014.

[6] Lara Codecá, Raphaël Frank, Sébastien Faye, and Thomas Engel. Luxembourg sumo traffic (lust) scenario github repository. https://github.com/lcodeca/LuSTScenario. Accessed: 19-05-2019.

[7] Lara Codecá, Raphaël Frank, Sébastien Faye, and Thomas Engel. Luxembourg SUMO Traffic (LuST) Scenario: Traffic Demand Evaluation. *IEEE Intelligent Transportation Systems Magazine*, 9(2):52–63, 2017.

[8] Lara Codeca and Jérôme Härri. Monaco SUMO Traffic (MoST) Scenario github repository. https://github.com/lcodeca/MoSTScenario. Accessed: 19-05-2019.

[9] Lara Codeca and Jérôme Härri. Towards multimodal mobility simulation of C-ITS: The Monaco SUMO traffic scenario. In *VNC 2017, IEEE Vehicular Networking Conference, November 27-29, 2017, Torino, Italy*, Torino, ITALY, 11 2017.

[10] Lara Codeca and Jérôme Härri. Monaco SUMO Traffic (MoST) Scenario: A 3D Mobility Scenario for Cooperative ITS. In *SUMO 2018, SUMO User Conference, Simulating Autonomous and Intermodal Transport Systems, May 14-16, 2018, Berlin, Germany*, Berlin, GERMANY, 05 2018.

[11] FlorianJa and mennowo. Interfaces and tools to work with sumo and traci. https://github.com/CodingConnected/CodingConnected.Traci. Accessed: 02-04-2019.

[12] Jérôme Haerri, Fethi Filali, and Christian Bonnet. Mobility models for vehicular ad hoc networks: a survey and taxonomy. Technical Report EURECOM+1951, Eurecom, 03

2006.

[13] Hello sumo tutorial. https://sumo.dlr.de/wiki/Tutorials/Hello_Sumo. Accessed: 16-04-2019.

[14] Daniel Krajzewicz, Jakob Erdmann, Michael Behrisch, and Laura Bieker. Recent development and applications of SUMO - Simulation of Urban MObility. *International Journal On Advances in Systems and Measurements*, 5(3&4):128–138, December 2012.

[15] Daniel Krajzewicz, Georg Hertkorn, Christian Rössel, and Peter Wagner. Sumo (simulation of urban mobility)-an open-source traffic simulation. In *Proceedings of the 4th middle East Symposium on Simulation and Modelling (MESM20002)*, pages 183–187, 2002.

[16] Libsumo. https://sumo.dlr.de/wiki/Libsumo. Accessed: 18-04-2019.

[17] Netconvert. https://sumo.dlr.de/wiki/NETCONVERT. Accessed: 16-04-2019.

[18] Netedit. https://sumo.dlr.de/wiki/NETEDIT. Accessed: 16-04-2019.

[19] Netgenerate. https://sumo.dlr.de/wiki/NETGENERATE. Accessed: 16-04-2019.

[20] OpenStreetMap contributors. Planet dump retrieved from https://planet.osm.org . https://www.openstreetmap.org, 2017.

[21] Openstreetmaps. https://www.openstreetmap.org. Accessed: 16-04-2019.

[22] Create random trips. https://sumo.dlr.de/wiki/Tools/Trip. Accessed: 20-05-2019.

[23] Route file. https://sumo.dlr.de/wiki/Definition_of_Vehicles,_Vehicle_Types,_and_Routes. Accessed: 15-04-2019.

[24] Importing arcview databases ("shapefiles"). https://sumo.dlr.de/wiki/Networks/Import/ArcView. Accessed: 20-05-2019.

[25] Sumo. https://sumo.dlr.de/wiki/SUMO. Accessed: 16-04-2019.

[26] Sumo-wiki-scenarios. https://sumo.dlr.de/wiki/Data/Scenarios. Accessed: 19-05-2019.

[27] Sumo road networks. https://sumo.dlr.de/wiki/Networks/SUMO_Road_Networks. Accessed: 15-04-2019.

[28] Traci. https://sumo.dlr.de/wiki/TraCI. Accessed: 04-01-2019.

[29] Object context subscription. https://sumo.dlr.de/wiki/TraCI/Object_Context_Subscription. Accessed: 02-04-2019.

[30] Object variable subscription. https://sumo.dlr.de/wiki/TraCI/Object_Variable_Subscription. Accessed: 02-04-2019.

[31] VISSIM contributors. VISSIM trafic simulator . http://vision-traffic.ptvgroup.com/en-us/products/ptv-vissim/. Accessed: 17-05-2019.

[32] Axel Wegener, Michal Piorkowski, Maxim Raya, Horst Hellbrück, Stefan Fischer, and Jean-Pierre Hubaux. Traci: An interface for coupling road traffic and network simulators. *Proceedings of the 11th Communications and Networking Simulation Symposium, CNS'08*, 04 2008.