Diploma Project

ADDING TRAFFIC SIMULATION TO THE DIGITAL TWIN OF NICOSIA

Vasilis Elia

UNIVERSITY OF CYPRUS

DEPARTMENT OF COMPUTER SCIENCE

May 2026

UNIVERSITY OF CYPRUS

Faculty of Pure and Applied Sciences

DEPARTMENT OF COMPUTER SCIENCE

ADDING TRAFFIC SIMULATION TO THE DIGITAL TWIN OF NICOSIA

Vasilis Elia

Supervisor

Yiorgos Chrysanthou

The Thesis was submitted in partial fulfillment of the requirements for obtaining the Computer Science degree at the Department of Computer Science of the University of Cyprus

May 2026

I declare that this dissertation and any accompanying software are my own original work, conducted in full compliance with the University of Cyprus Code of Conduct for Research (https://www.ucy.ac.cy/legislation/kodikas-deontologias-kai-politikes/), and with full accountability on my part for the accuracy, integrity, and validity of all content."

Acknowledgements

I would like to express my gratitude to my supervisor, Dr. Yiorgos Chrysanthou, for giving me the opportunity to work on this topic and for his continuous guidance, support, and advice throughout the development of this thesis.

I would also like to extend my warm thanks to Dr. Panagiotis Charalambous for his valuable advice and support at important stages of this work.

ABSTRACT

The purpose of this thesis is to integrate pedestrian and traffic microscopic simulation into the Digital Twin of Nicosia, also referred to as iNicosia. The thesis presents the design and implementation of a system for visualising pedestrian and vehicular simulation using SUMO (Simulation of Urban MObility) within a 3D Unity environment.

The project can be divided into three main steps. First, the selected area of Nicosia is extracted from OSM (OpenStreetMap) and converted into a SUMO road network, including roads, sidewalks, crossings and pedestrian walking areas. Second, the simulation is initiated and controlled through the TraCI (Traffic Control Interface) API and data about the simulated agents is sent to Unity. Third, Unity visualises the Digital Twin through the Cesium plugin. The received simulated data is then converted into georeferenced Unity positions so that vehicles and pedestrians can be visualised over the Digital Twin.

Although specific areas of iNicosia were used for testing and evaluation the proposed system can be adapted to other areas of Nicosia or to other cities, provided that suitable OpenStreetMap data and their corresponding Cesium Digital Twin are available.

TABLE OF CONTENTS

Chapter 1

Introduction

1.1 Motivation

Urban mobility is a major part of any city, especially in denser ones, such as Nicosia. Therefore, urban mobility should also be part of a city's digital representation, such as a Digital Twin. To achieve this, a simulation needs to operate within the city's road network, where interactions between vehicular traffic and pedestrian movement can be shown, in a realistic 3D environment. This can not only promote a better understanding of the city's mobility systems but also be used in digital museums to showcase the city's features.

1.2 Traffic Simulation

Different kinds of pedestrian and vehicular simulators and simulation models exist, and in different scopes. Some focus more on interactions among pedestrians, such as the Traditional Pedestrian Model 2.6.1, while others focus more on higher-level traffic representation in a city, such as SUMO 2.6.4. The focus of this project was pedestrian and vehicular simulation in an urban environment. Therefore, there was a need for a simulator capable of handling both types of simulations and the interactions among all kinds of agents. SUMO was eventually chosen as

the preferred simulator because it provides useful tools not only for handling such simulations but also for incorporating them in an urban area.

1.3 Visualisation

The major goal of this project was to split the simulation and visualisation into separate parts. We did this to provide greater flexibility and fewer limitations for future use and improvement. The visualisation platform that was decided to be used for the system is the game engine Unity 2.8. Unity offers useful tools for not only visualising the simulation, but also the part of the city that is used for the simulation. A big part of the visualisation was figuring out how to align the simulation data to the digital city model.

1.4 System Overview

The system consists of three major parts.

The first is converting the desired area to visualise the traffic simulation into a SUMO-readable road network. This requires exporting the area from OSM 2.7 and using the appropriate SUMO tools for the conversion.

The second is to run the simulation and send the data to Unity. For this part, TraCI 2.6.4.7 was used, which is a SUMO library capable of reading and modifying elements of the SUMO simulation at runtime.

Finally, the received simulation data are received in Unity and visualised. Unity also handles the visualisation of the Digital Twin using tools that allow importing part of the model into the scene. These tools basically allow Unity to convert the GCS coordinates 2.10.2.2 of each position in the model to Unity Cartesian coordinates. The coordinates in the simulation data are also converted using the same tools to the same coordinate system, and the simulation agents are displayed on top of the city model in the scene. Techniques, such as rendering the steps a bit after they are received and object pooling, are used to make the visualisation and interpolation of the agents smoother.

1.5 Document Structure

This document consists of several chapters, structured as follows.

In Chapter 2, the theoretical background is described, which mainly includes definitions of topics and tools used in this thesis. Some of the topics include different kinds of simulations as well as descriptions of the software used for the main architecture of the system.

In Chapter 3, the requirements for the completion of this project are listed. The reasons for choosing the selected tools and methods are also explained.

In Chapter 4, the overall architecture of the system is shown and described.

Chapter 5 describes the implementation process and its main components.

Chapter 6 presents the results of the thesis, what worked best and some of the issues.

Chapter 7 summarises the thesis and outlines potential future work for this project to improve and utilise it in additional ways.

Chapter 2

Background and Previous Work

In this chapter, all of the different tools and concepts that this thesis uses are described. Literature mainly addresses information about the Digital Twin and different kinds of simulation, which is an important concept, as the decision for choosing a specific type is explained later 3.3. The tools mentioned in this chapter are used to build and run the simulation in SUMO and visualise it in Unity.

Previous work on the Digital Twin of Nicosia is also presented in the last section of this chapter.

2.1 Smart City

A Smart city is one that uses technology and data collection to improve the quality of life. A smart city can collect data from traffic lights, road sensors, buildings, and surveillance cameras. Smart cities have become a lot more common in recent years, as governments are embracing

the idea of adopting technological solutions for improving city management and services [1].

2.2 Digital Twin

A digital twin is a digital representation of a real-world thing. This can be either an object, a building or even a city. The digital twin can not only be used to visualise and display its physical counterpart but also to synchronise its data, so it can show its current state. This can be done through various sensors.

A digital twin of a smart city is one that utilizes the technology and data collection it contains to visualise its different components. A city's digital twin can include buildings, infrastructure, energy systems, water systems, public spaces, environmental conditions and transportation networks [2].

The purpose of a digital twin is to help city planners and decision-makers monitor how the city is functioning, predict future problems, and test possible solutions before applying them in the real world. This means that digital twins can be used for real-time visualisation of the data collected or for different kind of simulations, such as vehicular and pedestrian. This kind of simulation can also be used in virtual museums.

2.3 Traffic Simulation

Traffic simulation is the computer-based modelling of transportation systems, such as roads, junctions, roundabouts, and urban networks. It is used to analyse, plan, design, and manage traffic systems by testing different scenarios and visualising their possible outcomes. The agents in a traffic simulation can be any kind of entity that uses the network transportation system, such as a car, a bus, a cyclist, or a pedestrian.

2.4 Traffic Simulation Classes

Traffic simulation can be divided into four classes based on the detail of the simulation [3].

2.4.1 Macroscopic Traffic Simulation

Macroscopic traffic simulations model the traffic flow as the basic entity, meaning there is no individual control over each specific agent. The simulation is basically controlled as a fluid. Microscopic simulations are mostly used for realistic traffic simulations.

2.4.2 Microscopic Traffic Simulation

Microscopic traffic simulation is the individual simulation of the movement of each agent. This kind of simulation gives characteristics (such as type and preferences) to each of the agents (car, pedestrian, cyclist, etc.) and controls them based on those. The interactions between each of the agents in this kind of simulation are way more detailed than in a macroscopic simulation. Agents can also obey traffic laws, such as following traffic signs, speed limits and stopping at traffic lights.

2.4.3 Mesoscopic Traffic Simulation

Mesoscopic traffic simulations are a mixture of microscopic and macroscopic traffic simulations. These kinds of simulations mostly use queue approaches, where each vehicle or pedestrian is moved one at a time.

2.4.4 Sub-microscopic Traffic Simulation

Sub-microscopic traffic simulations are even more detailed simulations than microscopic ones. The simulation controls each specific agent based not only on their basic characteristics, but also on a lot more detailed information about them, such as the vehicle's engine rotation speed or the agent's individual way of thinking about a specific situation.

2.5 Offline vs Online Traffic Simulation

2.5.1 Offline Simulation

Offline simulation uses prepared parameters or data gathered prior to simulation execution to generate agent movement in exported files [4]. This kind of simulation can be realistic, depending on the data gathered or the configuration used, but it does not represent the current state of the real-world system it aims to simulate.

2.5.2 Online Simulation

Online Simulation communicates in real time with a different system that feeds information about the simulation it is trying to run [4]. The data feeding system can be real-world sensors of a separate network simulator, which sends the generated network data to the traffic simulator.

2.6 Types of Pedestrian Simulators

This section presents different types of simulators and simulation models, ranging from traditional approaches to more recent methods. These models are discussed because several of them were initially considered for this project but were later discarded in favour of a more practical approach.

2.6.1 Traditional Pedestrian Model

This pedestrian model is one of the most traditional ones, which provides realistic interactions between pedestrian agents. It primarily uses social forces and heuristics, and agents use as input their surroundings and other nearby pedestrians to decide their path in order to avoid collisions [5].

2.6.2 Reinforcement Learning Based Model

Studies have examined the use of reinforcement learning to train pedestrian agents to act based on their configured type. Specifically, a study on CCP (Configurable Crowd Profiles) [6] was

conducted to train agents on different policies simultaneously. At the end, the user could use the model to configure the agents' profiles in real time, and the agents changed their behaviour accordingly. Several profiles were studied, including goal-seeking, collision avoidance, grouping, and interaction with points of interest. This methodology showed very good results, but the agents were trained in a specific environment and for a duration of 4 days.

2.6.3 Unity NavMesh

NavMesh [7] is a Class in the Unity game engine 2.8 that creates walkable paths in the Unity scene. Agents can then use a NavMeshAgent component to walk in the created walkable paths. This method uses an A* search algorithm to guide the agents and avoid collisions.

2.6.4 SUMO

2.6.4.1 General

SUMO is an open-source traffic simulation platform created by the Institute of Transportation Systems at the German Aerospace Center. It is used to model and analyse the movement of pedestrians and vehicles and the public transportation systems in urban environments [8]. It allows researchers and engineers to simulate real-world traffic conditions and evaluate existing and future systems. SUMO was primarily a vehicle-only traffic simulator, but has been updated to also include pedestrians as additional agents [9].

In SUMO, one can create their own road network manually or export urban areas from platforms such as OSM 2.7 and convert them to the corresponding road network using the netconvert 2.6.4.2 tool. Before running the simulation, one can also fix or alter the road network as one sees fit using the netedit tool. After creating the network, random trips for the agents can be created using randomTrips.py 2.6.4.4 and the corresponding routes using duarouter 2.6.4.5. Alternatively, one can use TraCI 2.6.4.7 to perform similar tasks. TraCI can also be used not only to control the simulation in real time, but also to communicate with the SUMO simulator and gather data about the simulation and its agents.

2.6.4.2 netconvert

netconvert [10] is a SUMO tool that uses as input maps from various sources, such as OSM 2.7, and converts them to a SUMO road network. The user can specify different flags depending on the road system they want to get. The road network can include roads, sidewalks, pedestrian areas, roundabouts, traffic lights, and more. In addition, the network can be imported into other tools for editing or simulation. Chapter 3 5.2.2 contains more information about how this can be used.

2.6.4.3 netedit

netedit [11] is a GUI-Application of SUMO that is used for creating from scratch or editing existing SUMO networks. Existing SUMO networks can be a result of a conversion using the netconvert tool 2.6.4.2. netedit is also very useful for visualising the SUMO network as seen in Figure 2.1.

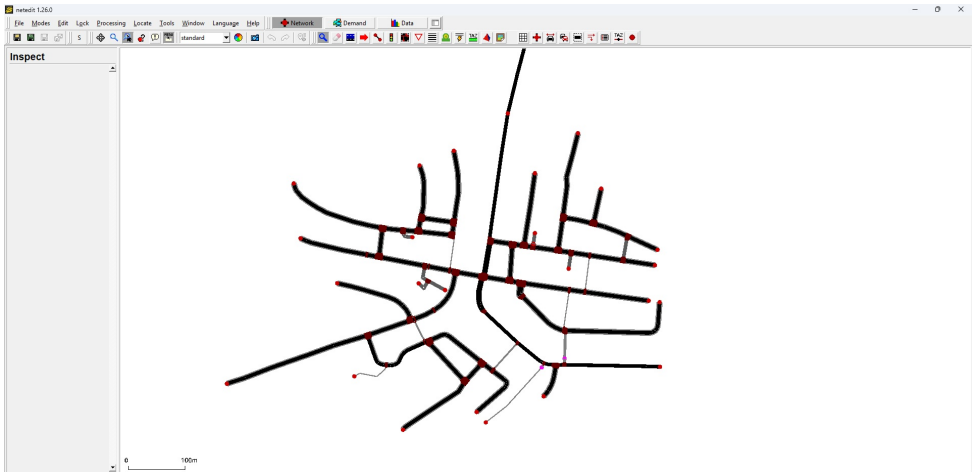

Figure 2.1: netedit sample of Strovolos, Nicosia

2.6.4.4 randomTrips.py

randomTrips.py [12] is a SUMO script that creates random trips on the SUMO network for agents to use. It takes random edges from the network and creates pairs of starting and ending points. It provides options for the user to choose the type of trip they want to create, the direction

of travel, the preferred lane on each edge, the type of agent to have on the trip, and more. The generated trips are used as input to duarouter 2.6.4.5 in order to create the corresponding routes for the agents.

The following command can be used to create random trips of pedestrian agents between 0 and 1000 simulation time:

python randomTrips.py –pedestrians -n network.net.xml -e 1000 -o pedTrips.xml

2.6.4.5 duarouter

duarouter [13] is a SUMO tool used to create routes for all trips in the resulting file from randomtrips.py 2.6.4.4. For each trip (a start-end pair of edges), duarouter finds and assigns the shortest possible path. If a valid route cannot be found, the corresponding trip is discarded. Some of the options for this tool include the starting and ending positions on the route's edge, as well as the departure speed.

The following command can be used for routing random pedestrian trips generated from randomtrips.py:

duarouter -n network.net.xml –route-files pedTrips.xml -o routes.rou.xml

2.6.4.6 config.sumocfg

config.sumocfg is a mandatory configuration file for running a SUMO simulation [14]. It is required regardless of whether one uses prebuilt trips 2.6.4.4 and routes 2.6.4.5 or creates them at runtime using TraCI 2.6.4.7. If using the prebuilt ones, they must be referenced in the configuration file. The configuration file can also specify various characteristics of the simulation, such as the total simulation time and the simulation step length.

2.6.4.7 TraCI

TraCI (Traffic Control Interface) [15] is a SUMO tool for communicating with the simulation using a TCP connection. It can be used to retrieve any kind of data about the current state of a running simulation. Data can include various information about the active agents [16], such as

their type, position, and velocity. In addition, it can be used to retrieve information about the network itself, including the edges [17] that compose it, the lanes on each edge [18], and other components such as traffic lights.

Furthermore, TraCI can also be used to control the simulation at runtime. This means that trips 2.6.4.4 and routes 2.6.4.5 can be excluded from the sumo configuration file 2.6.4.6 and instead be created from a script that uses TraCI to communicate with the simulation. In this case, the simulation is not started via the command line and the SUMO configuration file, but rather by a script that references the configuration file.

2.7 OpenStreetMap

OSM (OpenStreetMap) [19] is an open-source platform that users can retrieve geographic data, such as buildings, road systems and points of interest. OSM is maintained by users themselves, who can add data they collect using GPS devices, satellite imagery, and other sources to the platform.

An important feature of OSM is the ability to export part of the maps using GCS coodinates 2.10.2.2. These exported map segments can be used as input to the SUMO netconvert tool 2.6.4.2 to retrieve the corresponding road network.

2.8 Unity

Unity [20] is a real-time 2D and 3D development platform used to create video games, simulations, Virtual Reality and more.

Unity in this project is mainly used as visualisation platform for the simulation. This is because it supports all the necessary components to achieve that, such as tools for animation, scripting, physics, as well as the ability to install useful plugins, in order to communicate with other platforms, such as Cesium for Unity 2.9.2.

2.8.1 Raycast

Unity Raycast [21] is a functionality provided from Unity's physics library. It allows shooting a ray from a specified position, in a specified length. When the ray collides with an object that has a physics collider component attached to it, it returns information about the object hit. There is also the possibility of specifying the specific physics layer to collide with, in order to reduce the amount of collision checks happening and improve performance.

2.9 Cesium ion

2.9.1 Main Platform

Cesium [22] is an open-source platform for creating 3D maps. It is mostly used for visualising terrain, cities and city components such as buildings, roads and points of interest. It also supports different kinds of levels of detail.

Cesium also uses GCS coordinates to reference positions. This is a key point as in later chapters, it is explained how it is used in combination with OSM 2.7.

2.9.2 Cesium for Unity

Cesium for Unity [23] is a Unity plugin that integrates Cesium with Unity. It allows users to load data from Cesium, and stream it to Unity. This is done through the use of assets that come with the plugin such as Cesium World Terrain, and Cesium OSM buildings.

2.9.2.1 CesiumGeoreference Object

CesiumGeoreference object [24] comes with the Cesium for Unity plugin. Its job is to align the Unity scene with geographic locations on Earth using their GCS coordinates. This is vital as Unity uses a Cartesian coordinate system to position objects in the Scene.

2.10 Georeferencing

2.10.1 Definition

Georeferencing is the process of linking digital data to real-world coordinates that use a Spatial reference system 2.10.2. It is basically our way of locating real-world objects in maps.

2.10.2 Spatial Reference Systems

2.10.2.1 Definition

Spatial Reference Systems (SRS) are different systems used to specify the locations of objects in the real world. Each system uses its own type of coordinates.

2.10.2.2 Geographic Coordinate System

A Geographic Coordinate System (GCS) is a kind of SRS that uses coordinates based on latitude (degrees north or south of the equator) and longitude (degrees west or east of a prime meridian). GCS can only specify locations on the surface of the Earth.

2.10.2.3 Geocentric Coordinate System

A Geocentric Coordinate System or Earth-centred Earth-fixed (ECEF) Coordinate System is a Cartesian SRS that uses X, Y and Z measurements from its center of mass to specify the locations of objects. Locations can be in the interior or exterior of the Earth.

2.10.2.4 Haversine formula

The Haversine formula is a useful method for calculating distances between two GCS coordinates. Given the two points (ϕ_1, λ_1) and (ϕ_2, λ_2), where ϕ represents latitude and λ represents longitude, all angles are first converted from degrees to radians. The differences between the coordinates are then calculated as:

$$\Delta\phi = \phi_2 - \phi_1$$

$$\Delta\lambda = \lambda_2 - \lambda_1$$

The intermediate value a is computed as:

$$a = \sin^2\left(\frac{\Delta\phi}{2}\right) + \cos(\phi_1)\cos(\phi_2)\sin^2\left(\frac{\Delta\lambda}{2}\right)$$

The central angle c is then calculated by:

$$c = 2\arctan 2\left(\sqrt{a}, \sqrt{1-a}\right)$$

Finally, the distance d between the two points is obtained as:

$$d = R \cdot c$$

where R is the Earth's radius, commonly taken as 6371 km.

2.11 iNicosia

iNicosia [25] is the Digital Twin of Nicosia and is developed by the team of the CYENS Centre of Excellence. The Digital Twin contains accurate representations of the city's buildings and monuments, along with the road network, and includes live traffic data from sensors placed around the city. Figure 2.2 shows the main digital platform of iNicosia and Figure 2.3 shows an image of the model taken from the same platform.

2.11.1 Importance

Nicosia is unique because it is the last divided capital city in the world. iNicosia can therefore help with the unification of the city, through the collaboration of the people and the interactions that are necessary for building this project.

Figure 2.2: Image of the iNicosia main platform

Figure 2.3: Image of a part of the model of iNicosia

Chapter 3

Requirements

3.1 Vehicular and Pedestrian Traffic Simulation

Two options were considered for adding pedestrian simulation to the Digital Twin of Nicosia.

The first one would use Traditional approaches 2.6.1 or emulate similar models to focus on realistic interactions among pedestrians. This would be good for a virtual museum and would feel more real, but it would limit the potential to add more features, such as simulating panic and evacuation situations or retrieving and simulating real-world data 2.5.2.

The second one would be to add a microscopic pedestrian simulation 2.4.1, with agents trying to reach different points of interest via routes. This would allow more flexibility to add features in the future. This meant, though, that for a realistic microscopic simulation, vehicles would have to be added and interact correctly with pedestrians. Interactions such as these include vehicles waiting for pedestrians to finish crossing the road or pedestrians waiting for vehicles to pass before crossing.

For this thesis, the second option was chosen, and the requirements changed as described below 3.2.

3.2 City Road Network

A big requirement for adding vehicular and pedestrian simulation to a Digital Twin is having access to the city's road network. In addition, such a road network must include not only roads but also sidewalks, road crossings, and pedestrian areas.

One way to achieve this would be to load the Cesium model into Unity using the plugin 2.9, and try to build the road network system. Building the road system manually could work, but it would not be practical for a big city. Figure 3.1 shows an attempt to build such a system on a demo city [26] by adding a manual mesh on top of the road to mark it and then using Unity Navmesh 2.6.3 to guide the car. A solution like this, though, would be too time-consuming and limited to Unity NavMesh for simulation. Therefore, it was clear that a more robust, automated solution was necessary.

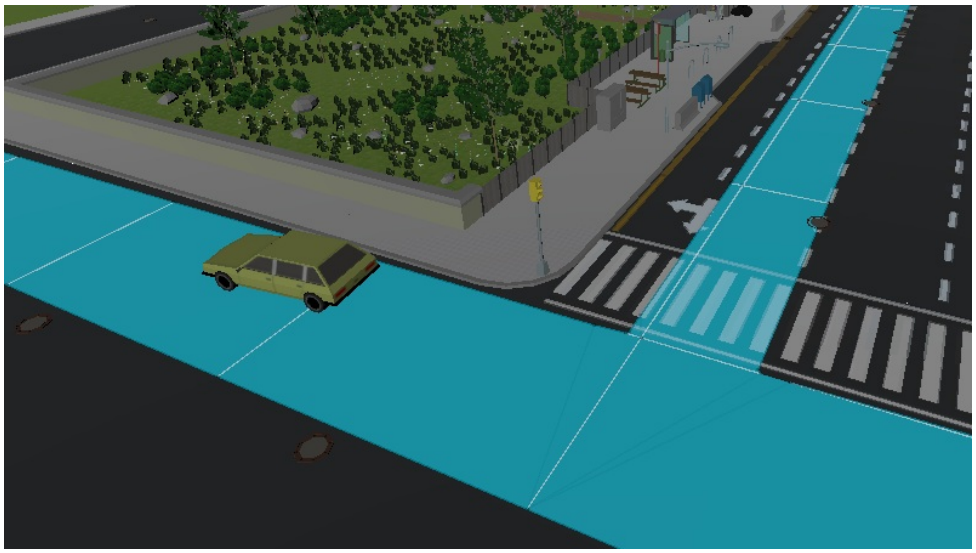

Figure 3.1: Creating a Road Network Manually in Unity

3.3 Appropriate Simulator

Different types of simulators and simulation models exist, as discussed in 2.6. The traditional model 2.6.1 would only work for pedestrian interactions and has been replaced by newer, easier-to-implement simulators. CCP 2.6.2 was also a potential option at the beginning of the object, but it would require a lot more training, and would not be practical for the scale of a Digital

Twin. Unity NavMesh 2.6.3 was a realistic option assuming a solution for the road network was found, but it would be a limited solution to the Unity platform.

SUMO 2.6.4 had the necessary capabilities for supporting vehicular and pedestrian microscopic simulation, with interactions between the two types of agents, as well as the ability to load and automatically convert a selected part of a city to a road network through the netconvert SUMO tool 2.6.4.2. Similar work was also done [27], which sent the simulated data to Unity for 3D visualisation; therefore, choosing SUMO for simulation and Unity as a platform for visualising the simulation seemed the most practical choice.

3.4 Syncing Simulation to Digital Twin

The last important requirement was to match the simulation of the agents' movement to the Digital Twin, which could be streamed in Unity using the Cesium for Unity plugin 2.9.2.

For this requirement, the solution was to use the SRS systems 2.10.2 to match the simulated data to the appropriate positions in the Digital Twin. This was possible, since SUMO could retrieve the GCS coordinates 2.10.2.2 from the imported OSM network, and Unity could also convert these coordinates to its local Cartesian coordinate system through the Cesium Georeference Object 2.9.2.1.

Chapter 4

System Design

4.1 General

This chapter describes a high-level overview of the whole system and its core components. The general purpose of the system is to perform a traffic simulation in a part of the Digital Twin of Nicosia and visualise the results in the Unity 3D game engine. The components of the system include building a corresponding road network for the specific part of the Digital Twin, building and running the simulation, communicating with Unity to send the appropriate data and visualising the simulation on top of the Digital Twin in Unity. A diagram of the design of the system is shown in Figure 4.1.

4.2 Building the Road Network

Building the SUMO road network that the simulator will use requires first exporting a segment of the city from OSM, converting it into the corresponding road network using SUMO tools, and fixing any faulty parts.

4.3 Building and Running the Simulation

Once the SUMO road network is ready, the simulation needs to be built. This process includes creating appropriate random trips for vehicles and pedestrians and spawning them at intervals. A set of configuration steps and settings is required for this segment to work properly. In addition, there are two ways of completing this part of the system. The first one uses scripts such as randomtrips.py 2.6.4.4 and duarouter 2.6.4.5, while the second one uses only TraCI 2.6.4.7 to do all the work. Similarly, to run the simulation, one must either use the previous resulting configuration from randomtrips.py and duarouter, or use TraCI to create everything dynamically as the simulation is running. After experimenting with both ways, TraCI was chosen as the preferred method for building and running the simulation, as it was more versatile.

4.4 Sending the Simulation to Unity

Sending the simulation to Unity, requires first collecting the appropriate data, mainly for the current positions of the agents and then sending them to the corresponding UDP port that Unity listens to. Data can be collected using the TraCI tool in SUMO via a script. An important part of this process is that no data is sent to Unity until Unity first communicates with the script to let it know it has started listening, and to also specify the area of the map it is interested in for retrieving the simulation.

4.5 Matching the Simulation to Cesium in Unity

Cesium streams data for the Digital Twin through the Cesium ion for Unity plugin 2.9.2. Unity also knows how to locate positions in reference to the streamed model using GCS coordinates via the Georeference Cesium Object 2.9.2.1. Therefore, using the agents' coordinates, Unity can position them at the appropriate locations.

4.6 Rendering the Simulation

The last step is to render the simulation in Unity, meaning to visualise all the agents at their corresponding locations on top of the streamed Cesium model of the Digital Twin. This requires several steps, with the most important ones being using Unity physics to position the agents at the correct height and interpolating their movement across simulation steps. In order for the interpolation to be smooth, it was required that the visualisation was not performed at the exact frame of a received simulation step. For this reason, a buffer of simulated steps is stored for each active agent.

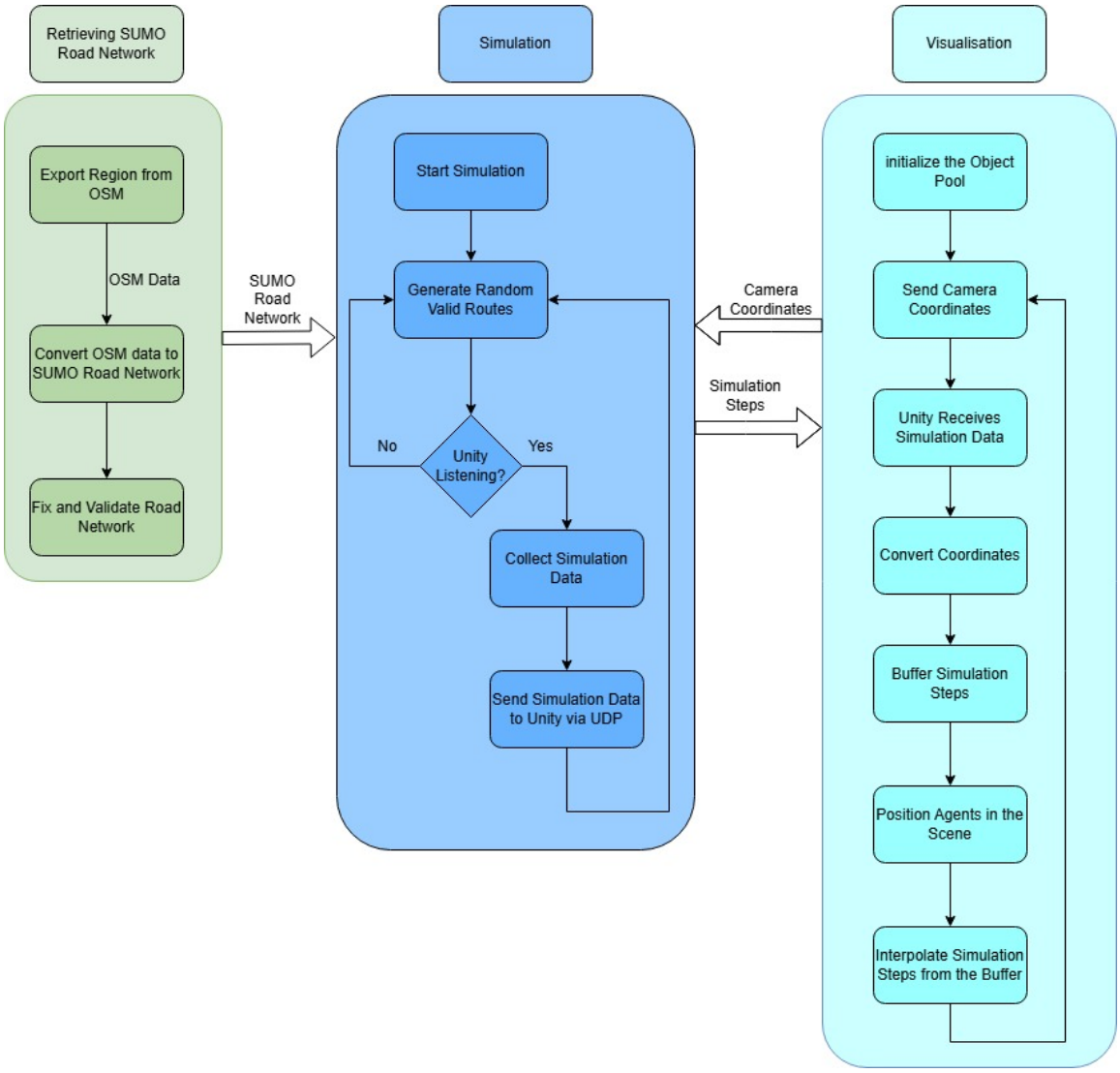

Figure 4.1: Diagram of the design of the system

Chapter 5

Implementation

5.1 General

In this chapter, the implementation of each part of the system's design is explained. For each part, the steps are listed, along with any required code. Any important decisions throughout the implementation are also explained.

5.2 SUMO Road Network Generation

This section describes the required steps to retrieve a SUMO road network for the area of the Digital Twin where we want to visualise the simulation. The steps consist of exporting the selected area from OSM, converting it into the SUMO road network, and then fixing any potential issues that may arise after the conversion.

5.2.1 OSM Area Export

The first step in generating the SUMO road network is exporting the desired simulation area from OSM. OSM supports exporting an area using GCS coordinates (latitude and longitude). Figure 5.1 shows an example of exporting a selected area of Nicosia. The corresponding longi-

tude and latitude of the specific area are shown in the left area of the figure.

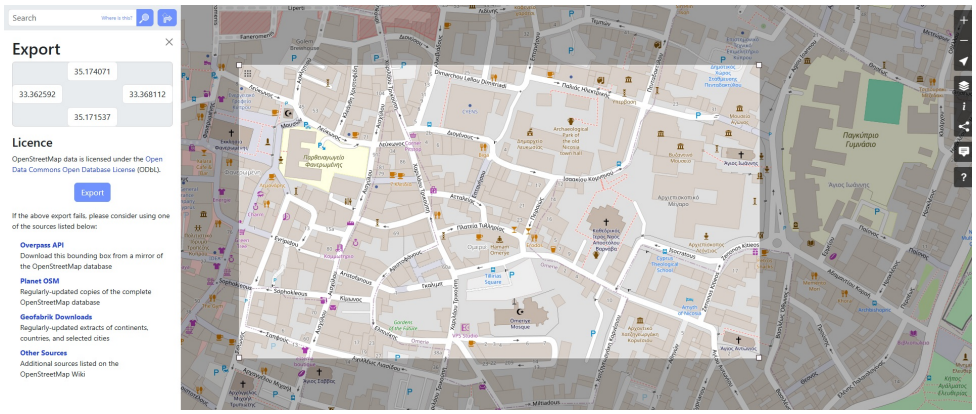

Figure 5.1: Exporting a selected area from OSM

5.2.2 Converting OSM Area to a SUMO Network

The conversion of the OSM area to a SUMO road network was done using the SUMO tool netconvert 2.6.4.2. Several flags were used to improve the quality of the resulting network. The decision to select the flags was made to ensure the generated network matched the real road network and the Digital Twin as closely as possible, and to automatically generate sidewalks or other road segments when this data was not included in the original OSM exported area. The resulting netconvert command was this:

```
netconvert --osm-files nicosia.osm
  -o nicosia.net.xml
  --ramps.guess
  --tls.guess-signals
  --tls.discard-simple
  --tls.join
  --sidewalks.guess
  --crossings.guess
  --osm.sidewalks
  --lefthand true
  --remove-edges.isolated true
  --no-turnarounds.except-deadend
```

Some important flags [10] that were used are –sidewalks.guess and –crossings.guess. These allow SUMO to predict some of the sidewalks and crossings for roads that have sidewalk, but there is no information about them in OSM. This does create some inconsistencies with the Digital Twin. However, it solves the problem of having pedestrians walk on the road and blocking traffic, which is a problem that appears on roads that allow pedestrians.

5.2.3 Fixing network using netedit

SUMO netedit 2.6.4.3 was not used too much, but it was a helpful tool for debugging and also for the removal of roads that caused traffic to build up. It is also useful for removing any part of the network that should not exist, due to OSM data not matching the city network of the Digital Twin.

5.3 Building and Running the Simulation using TraCI

The process of creating and running the simulation is described. A SUMO simulation can be built on a SUMO road network using either the tools randomtrips.py and duarouter or the TraCI library. The decision to choose TraCI over the other tools is explained. The steps include starting the SUMO simulation and configuring it with the desired number of concurrent agents by selecting random valid trips within the SUMO road network.

5.3.1 TraCI over randomtrips and duarouter

After experimenting with creating a simulation using randomtrips.py and duarouter or just TraCI, ultimately, building and running the simulation directly from a Python file, using TraCI, was chosen for its easier debugging and configuration. The drawback of using TraCI was finding the best way to build the simulation from the given SUMO road network, but with the benefit of increased flexibility in configuring the network in the end.

5.3.2 Starting the Simulation

In order to run the simulation using TraCI, a .sumocfg configuration file needs to exist 2.6.4.6. Since no trips or routes were created and needed to be linked, the configuration file could be as simple as this:

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

This configuration file can then be used as a parameter in traci.start() with added flags such as –step-length, which is useful for specifying the time between simulation steps. This flag can only be used with an added real-time delay between simulation step executions to achieve a more realistic simulation. For example, using –step-length = 0.05s means that for a pedestrian walking at speed u, the next step would place them at the position they would reach after walking a distance of dx = u * step-length. Iterating the simulation steps in real-time, every 0.05s, also means that the simulation step is executed after the time it would actually take the pedestrian to move that distance. This makes the simulation more realistic and easier to control. The duration of the simulation can be controlled manually by keeping track of the current step and advancing the simulations steps using traci.simulationStep(). The maximum number of concurrent vehicles and pedestrians was controlled by reading the current lists of pedestrian and vehicle agents using traci.vehicle.getIDList() and traci.pedestrian.getIDList(), and additional agents were spawned only if fewer than the desired number existed in a simulation step. This allowed spawning the appropriate number of agents to occupy a specific simulation area.

5.3.3 Creating valid trips and routes

This step occurs only at the beginning of the simulation and never again. Creating valid random trips first requires retrieving all the network's edges. It is important to note that some edges are for pedestrians only, some for vehicles only, and some for both. Retrieving the edges can be

done with the following function:

traci.edge.getIDList()

An edge, though, can be any edge of the SUMO network. Therefore, it is important afterwards to filter the ones needed for the specific trip based on the type of agent to spawn. To do this, we must go through the lanes of each edge and check whether there is at least one lane that the specific type of agent is allowed to go on. The following code segment iterates over the lanes on each edge and marks those that allow any vehicle ('None') or specifically a passenger vehicle. Building the ID of each lane is explained here [28].

```
for i in range(lane_count):
    lane_id = f"{edge}_{i}"
    allowed = traci.lane.getAllowed(lane_id)

    if allowed is None or "passenger" in allowed:
        allows_vehicles = True
```

To create trips, similar to randomtrips.py 2.6.4.4 for pedestrians and vehicles, we first pick 2 random edges from the corresponding lists. For creating the corresponding routes, similar to duarouter 2.6.4.5, we can use the TraCI function:

stages = traci.simulation.findIntermodalRoute(start, end)

where 'start' and 'end' are the starting and ending edges of the route. This returns a set of stages, which contain the edges of the route in the road network. We can then use traci.person.add(pid, edgeID=start, pos=0), where pid must be a unique agent id we can create using the current simulation step, to add the person at the specific edge. The pedestrian has not been added to the simulation just yet, though. In order to add them, we need to use the following TraCI function:

```
traci.person.appendWalkingStage(
    personID=pid,
    edges=stages.edges,
    arrivalPos=0.0,
    speed=walk_speed)
return True
```

where walk_speed = random.uniform(1.0, 1.5). The specific range is the average walking speed of a pedestrian [29]. A similar process can be used to add vehicles to the simulation using

route = traci.simulation.findRoute(start, end) and

traci.route.add(route_id, route.edges),

where route_id is a unique id based on the current simulation step and vehicle number.

5.4 Sending Simulation to Unity

Before sending any simulation data to Unity, a validation that Unity has started listening for packets must be sent to the script running TraCI, along with Unity's camera's GCS coordinates and the radius for active agents in meters. The script listens for UDP packets and once at least one is received, the agents in the simulation near the camera's coordinates are retrieved and sent to Unity. To listen for Unity packets, a second daemon thread is used, since creating a socket and waiting for packets from Unity would block the main thread of the simulation.
In order to learn which agents are close to the camera, we first get their GCS coordinates:

```
x, y = traci.person.getPosition(pedestrian)
lon, lat = traci.simulation.convertGeo(x, y)
```

Then, using the Haversine formula 2.10.2.4 we calculate the distance between the agent and the camera in meters. Agents within the camera's radius are sent to Unity. For each agent sent to Unity, the following data are sent: its ID, the longitude and latitude of its position, its current speed, the angle it is facing, and its type. These are added in a JSON and sent to Unity via UDP.

5.5 Unity Setup

This section describes the Unity setup required to import the Cesium ion Digital Twin into Unity and configure its visual components. In addition, the classes created to specify the area around the camera and to receive the corresponding simulation data around that area are explained.

5.5.1 Cesium ion for Unity

Cesium ion for Unity plugin 2.9.2 was added as a tool for importing the Cesium data into Unity. More specifically, it contains assets for streaming terrain and buildings data into the Unity scene.

The assets that were used for terrain and road road data were Cesium OSM buildings and Cesium World Terrain. These two assets are automatically attached to a CesiumGeoreference object 2.9.2.1, in order for Unity to place them in its coordinate system. Figure 5.2 shows the Unity scene near the CYENS building in Old Nicosia after importing the assets. On the left the Hierarchy is shown with the Cesium assets under the CesiumGeoreference object. The buildings do not have textures from the available Cesium assets, therefore a simple texture was added to them to visualise them. Roads with no photorealistic imagery were preferred for clarity. The Unity scripts were attached to the SimulationManager and CameraAreaRequest objects.

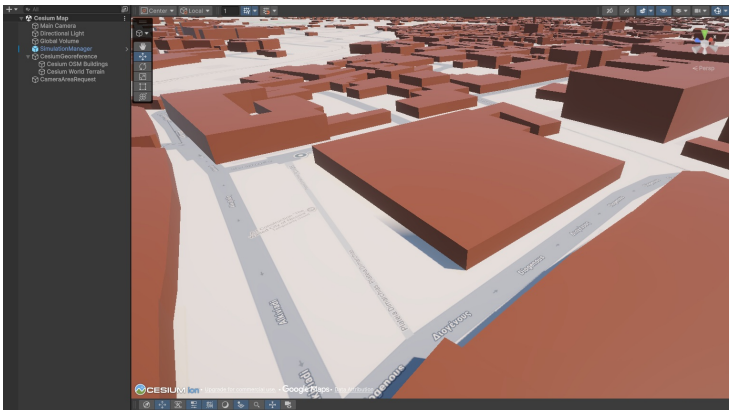

Figure 5.2: Unity scene with Cesium ion assets

5.5.2 Unity Camera Area Request

A user can use the Unity camera to move and see around the Digital Twin. Therefore, we only care about objects and agents close to the camera. For this reason, a Unity class was made to send the camera location and the radius in which we want the user to see the simulation. The Unity camera, though, has coordinates that match Unity's world coordinates, which are not coordinates that TraCI can use to filter the agents sent. Luckily, the Cesium Georeference object contains a method for first converting the Unity world coordinates to ECEF coordinates 2.10.2.3 called TransformUnityPositionToEarthCenteredEarthFixed()
and afterwards to GCS coordinates (longitude and latitude) with a method from the class CesiumForUnity called
CesiumWgs84Ellipsoid.EarthCenteredEarthFixedToLongitudeLatitudeHeight().

These are now coordinates that TraCI can use to find the position of the camera within the SUMO network.

5.5.3 Receiving the Simulation in Unity

Receiving the data of the simulation in Unity is done from a class that was made, which listens using a UDP socket for incoming packets. After receiving a packet it deserializes it from its JSON format to custom Unity classes that were made for each type of agent. Below is the Pedestrian class for pedestrian agents with the simulation attributes that were used:

```
public class Pedestrian
{
    public string id;
    public double lon;
    public double lat;
    public double height;
    public float speed;
    public float angle;
}
```

A similar class was used for the vehicle agents. Afterwards, the main Simulator object is called with the newly arrived simulation step to update the visualisation accordingly.

5.6 Visualising the Simulation in Unity

In order to visualise the simulation the necessary data are sent to the simulator class. The specific class is responsible for visualising the agents and positioning them at the appropriate positions in the scene. For the positioning, the coordinates are converted from longitude and latitude to Unity Cartesian coordinates. In addition, to improve performance in Unity, game object pooling for agent prefabs was used. Finally, to make the simulation smoother, the agents' rendering was done slightly after the data were received in Unity to ensure there were always enough simulation steps to interpolate between.

5.6.1 Simulator Class

A class for running and rendering the simulation was created and added to an object in the scene. This class receives the simulation data for the agents near the user's camera. It is also responsible for keeping a buffer of active agents from the most recent steps, updating their state, and rendering them in the scene at the appropriate position.

5.6.2 Active Agents

Active agents are those received in the latest simulation step or have been received recently. All of the active agents are stored in a dictionary based on their ID. For each active agent, we keep a state of their type, the time of their last simulated step and a list of simulation steps.
The time of the last simulated step is important since we remove agents completely only after a certain amount of time. This is because if the camera sees a green car, goes far away for a bit, and then comes back, we want that green car to be the same, at a close position relative to its last seen position. The following code removes the agent only after it has not been in a simulation step for removeAfterSeconds time.

```
if (timeNow - state.lastStepTime >= removeAfterSeconds)
{
    ReturnObjectToPool(state.obj, state.type);
    toRemove.Add(activeAgent.Key);
}
```

If the agent should not be removed first, we just disable it from the scene using SetActive(false). The pool is explained in the next section 5.6.3.

5.6.3 Game Objects Pooling

Using object pooling for the Unity game objects is an optimisation step. Game objects are the models for vehicles and pedestrians, and instantiating them takes resources. Therefore, an object pool of specified size is created initially. Whenever an agent with no game object needs to be rendered, a game object from the appropriate pool is given to it.
In addition, agents do not lose their game object instantly if they are not present in the latest

simulation step; instead, their game object is temporarily disabled from the scene. Also, the game object assigned to them is not deleted when they become inactive; instead, it is returned to the appropriate pool as shown above.

5.6.4 Positioning Agents in the Scene

In the simulated data, the agents' positions are sent in GCS coordinates. In order for Unity to be able to use these coordinates to place the agents inside its scene, they need to converted to Cartesian coordinates. This conversion is done by the Georeference object of Cesium, in a similar process to the one used for sending the camera position to TraCI 5.5.2, but with the conversion now being from GCS coordinates to Unity Cartesian coordinates.
The issue is that Cartesian coordinates require a height coordinate in addition to the two surface coordinates. Longitude and latitude though, only specify the position on the surface of Earth. For this reason a height value, is also given that is certain to be above the ground area, where the Cesium terrain is in the scene. In order to ground the agents that are now located way above the ground, Unity RayCasting 2.8.1 is used. The code below shows creating a ray and downwards and retrieving the position of the hit on the ground:

```
if (Physics.Raycast(
    rayStart,
    Vector3.down,
    out RaycastHit hit,
    rayDistance,
    groundMask))
{
    unityPos = hit.point + Vector3.up * groundOffset;
    return true;
}
```

Figure 5.3 shows three pedestrian agents in the SUMO GUI near the CYENS building, while Figure 5.4 shows the same pedestrians in the Unity visualisation at the same position.

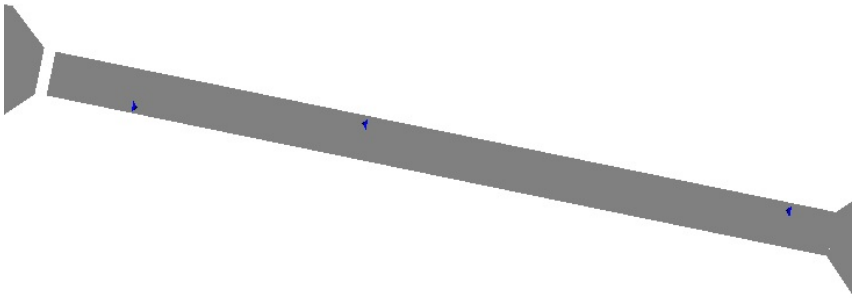

Figure 5.3: Pedestrians shown as blue points in SUMO GUI near CYENS building

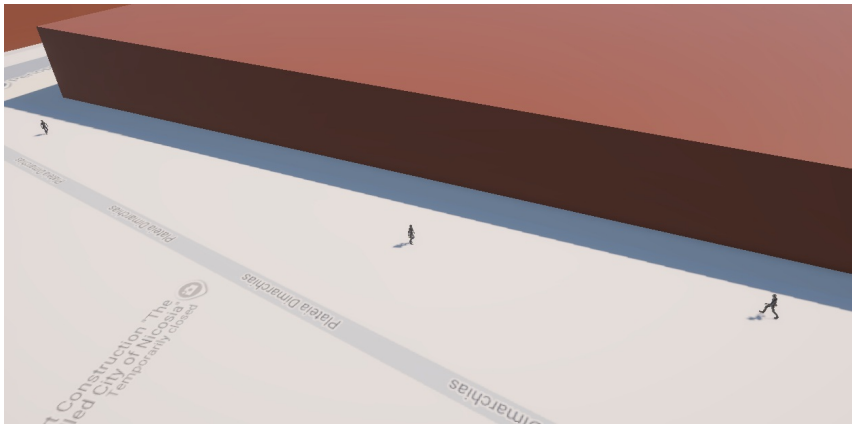

Figure 5.4: Pedestrians visualised in Unity near CYENS building

5.6.5 Rendering Time

As mentioned in 5.6.2, for each agent, we maintain a buffer of simulated time steps rather than just the current time step. This is done in order to improve the smoothness of the rendered agents and reduce jitter [30]. To apply this technique, a number of simulation steps need to be saved in a buffer. In addition, rendering uses simulation steps that have arrived slightly earlier. Therefore, the render time used for interpolating between simulation steps is calculated as:
renderTime = Time.time - interpolationDelay;
To avoid using old simulation steps, the buffer for each agent is updated to hold only simulation steps with time of arrival in Unity later than the specified render time. Once the render time reaches the arrival time of a simulation step, then interpolation can also happen. If only one step is available for an agent, or multiple steps are available but all have time after the render time, then the agent will be placed at the oldest valid position until interpolation can begin. Below is the code to keep only buffered simulation steps with time after the render time, or at least two simulation steps if there are multiple simulation steps with arrival time after the render time:

```
while (bufferedTimeSteps.Count >= 2 && bufferedTimeSteps[1].time <= renderTime) {
    bufferedTimeSteps.RemoveAt(0);
}
```

5.6.6 Interpolation

Interpolation of the agents happened every frame by using the two newest simulation steps such that the render time is temporally positioned in between them. Meaning one has time with bigger value than the render time and one value smaller. We then use the render time to linearly interpolate between simulation steps and obtain the corresponding position for each frame. Afterwards, we place the agent at the resulting position. Below is the code for the interpolation between the mentioned two steps:

```
if (bufferedTimeSteps.Count >= 2) {
    TimeStep a = bufferedTimeSteps[0];
    TimeStep b = bufferedTimeSteps[1];

    float t = Mathf.InverseLerp(a.time, b.time, renderTime);
    Vector3 interpolatedPosition = Vector3.Lerp(a.position, b.position, t);
```

Also, assuming there is only one available simulation step, because no simulation step exists with value of arrival time earlier than the rendering time, the agent will just stay still at the latest simulated position:

```
else {
    TimeStep latestStep = bufferedTimeSteps[0];
    state.obj.transform.position = latestStep.position; }
```

5.6.7 Animation

The pedestrian animation was based on the speed values obtained from the simulation steps. Using that value, an interpolation between the idling and walking animations was applied via the Unity Animator component.

Chapter 6

Results

6.1 Summary

In this chapter, the results of the system are presented. The evaluation is performed using two selected areas: one in Old Nicosia around CYENS and a smaller area in Strovolos. These areas are used to demonstrate the system's accuracy, as well as some of the encountered issues, mainly related to mismatches between OSM maps and Digital Twin buildings.

The results from Old Nicosia presented more alignment issues, while the results from Strovolos showed improved visualisation quality. In addition, the performance of both the Unity visualisation and simulation systems was evaluated.

6.2 Visualisation results of the simulation in Old Nicosia

Here are some results from the visualised simulated data for the Digital Twin of Old Nicosia. The specific run was tested with 20 concurrent cars and 200 pedestrians. This was enough to populate most of the areas around the selected area. Figure 6.1 shows the SUMO network using netedit.

Figure 6.1: SUMO network of Old Nicosia

Figure 6.2 shows the simulation in Unity in front of the CYENS building.

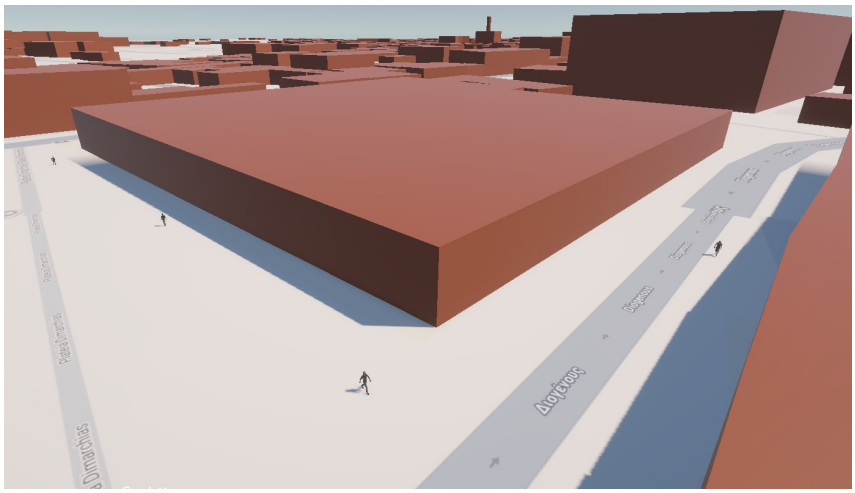

Figure 6.2: Unity visualisation in front of CYENS building

Figure 6.3 shows a part of the simulation containing both vehicles and pedestrians. In this example, some misalignment between the Cesium OSM Buildings and Cesium World Terrain assets is also visible, which was one of the issues encountered during development.

Figure 6.4 further demonstrates this issue, where some buildings overlap with roads or sidewalks due to inconsistencies in the alignment between the terrain and building assets. As a result, agents moving through these sections of the road network appear to pass through buildings. This issue was more significant in areas such as Old Nicosia, where the roads are narrower,

compared to areas such as Strovolos, which generally contain wider roads.

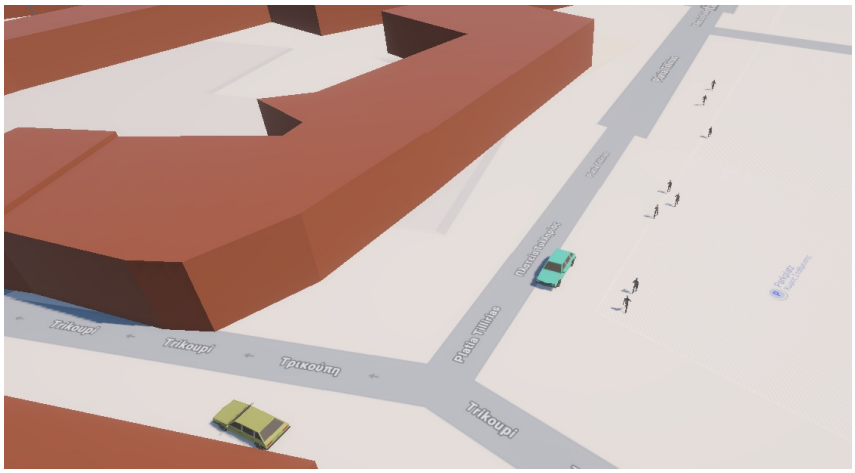

Figure 6.3: Unity visualisation in front of pedestrians and vehicles

Figure 6.4: Unity building and road mismatch at Old Nicosia

Another issue can be the sidewalks being misaligned with the actual Cesium network. This is because not all sidewalks are present in OSM, therefore –sidewalks.guess was used. This caused some of the added sidewalks to not match their actual width in the Cesium ion model. An example can be seen in Figure 6.5, where a pedestrian is walking inside the building.

Figure 6.5: Unity simulation error of pedestrian walking inside a building

6.3 Visualisation results of the simulation in Strovolos

Strovolos is a highly populated area of Nicosia characterised by wider roads. Additionally, OSM contains more detailed information about its road network compared to Old Nicosia. As a result, the visualisation quality of the simulation in this area is improved, since the roads and buildings from the corresponding assets are better aligned and do not overlap.

Figure 6.6 and Figure 6.7 show some of the simulation results in Unity. However, the roads are still not accurately represented in the visualisation, which remains one of the limitations of the specific assets used for streaming the terrain.

Figure 6.6: Visualisation results of the simulation in Strovolos

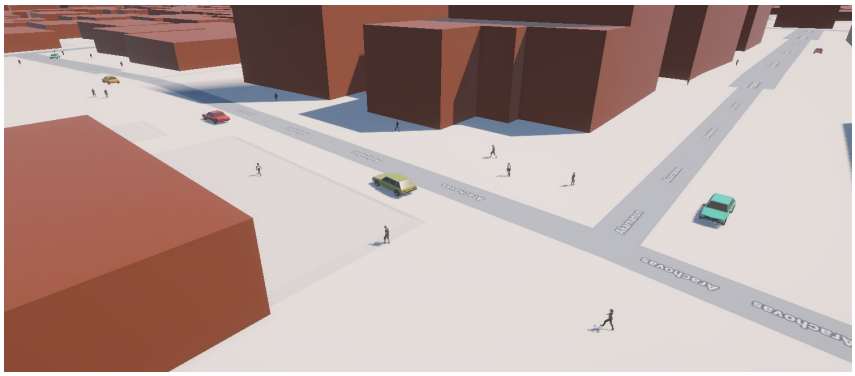

Figure 6.7: Additional visualisation results of the simulation in Strovolos

6.4 Performance Evaluation

Performance was evaluated for both the Unity visualisation and simulation components. Measurements were collected across varying numbers of agents, with the Unity evaluation focusing on frame rate (FPS) and the simulation evaluation focusing on the time overhead required to generate simulation steps.

6.4.1 Rendering Performance in Unity

To evaluate the performance for the visualisation in Unity, netedit was used to create a custom road network capable of a large number of agents within the camera radius. The camera radius was adjusted accordingly to ensure that all agents were visible during testing.

Since a single UDP packet can transfer data for up to 400 agents, 400 agents were transmitted to Unity using the custom road network and rendered in the scene. Unity maintained stable performance throughout the evaluation at 60 FPS. Multiple combinations of vehicles and pedestrians were tested while maintaining the same total number of agents, all producing similar results.

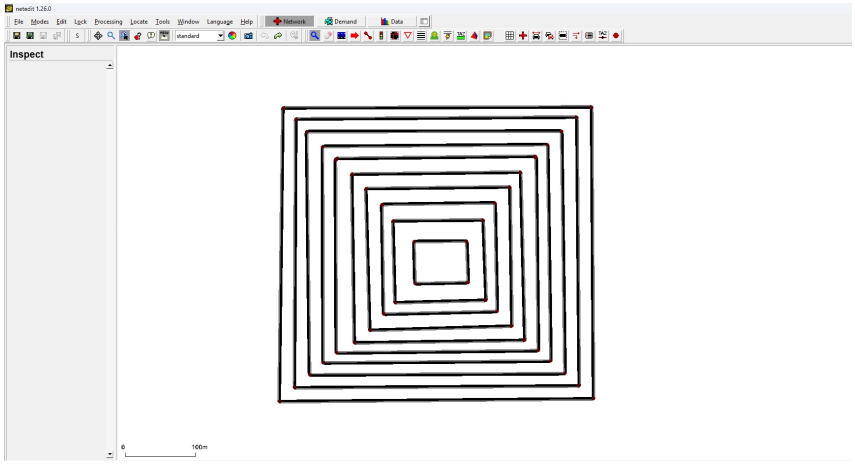

Figure 6.8: Road Network created for Benchmarking in netedit

6.4.2 Simulation Performance

The most computationally demanding operation on the simulation side was collecting the required simulation data at each simulation step. Therefore, the time required to gather the data and construct the UDP packet for each step was measured.

The evaluation was performed using different numbers of agents, with the average execution time recorded over 100 simulation steps for each configuration. The measured time includes retrieving vehicle and pedestrian data from SUMO through TraCI, converting the positions to GCS coordinates, filtering agents based on the camera radius, and finally constructing the data packet for transmission.

Initially, the time required to generate a simulation step was measured separately using only vehicles and only pedestrians. For both agent types, the processing time for 400 agents was approximately the same, requiring around 78 ms per simulation step.

It is important to note that this measurement corresponds to a scenario where the simulation contains 400 agents, all of which are processed and rendered. In simulations containing more than 400 agents, while still limiting the visualisation to 400 rendered agents, the processing overhead would increase, since data must still be retrieved and evaluated for all simulated agents before filtering.

Figure 6.9 shows that the time required to generate a simulation step scales linearly with the number of active agents. This indicates that the interval between transmitted simulation steps is not only determined by the predefined 0.05 s delay introduced in Section 5.3.2, but also by the computational overhead required to generate each simulation step. Issues start appearing when the total time between consecutive simulation updates exceeds either the SUMO simulation step duration or the rendering time in Unity visualisation 5.6.5.

When the total update time exceeds the SUMO simulation step duration, some simulation steps may be skipped, causing agents in the visualisation to deviate from their actual simulated paths. Additionally, when the update time becomes greater than the rendering time in Unity, interpolation no longer starts from the beginning of the received simulation step. Instead, interpolation starts from an intermediate point between two steps, resulting in agents jumping between positions rather than transitioning smoothly.

To resolve these issues, the SUMO simulation step duration should first be adjusted to closely match the total time required to generate the simulation step packet. Similarly, the rendering time in Unity should be increased to be slightly larger than the packet generation time, so that the interpolation correctly starts from the initial point between consecutive simulation steps. Finally, the delay between generating simulation step packets can be adjusted accordingly to remain synchronised with the SUMO simulation step duration.

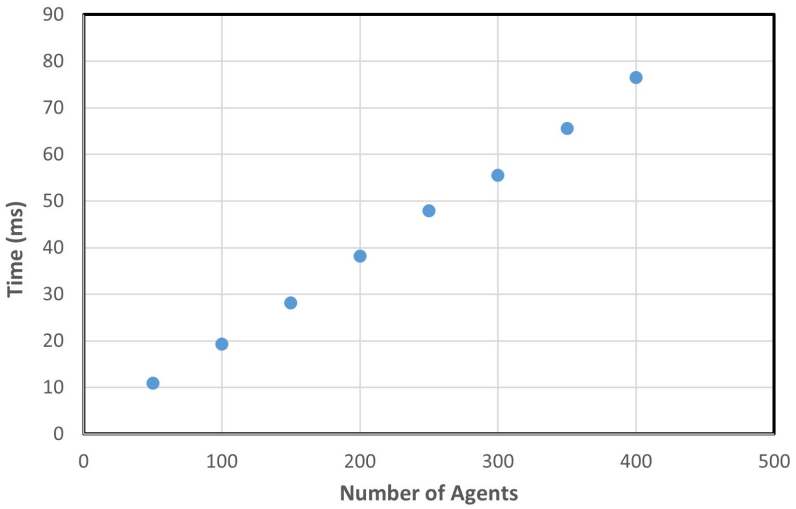

Figure 6.9: Simulation step generation overhead for different numbers of active agents

Chapter 7

Conclusion and Future Work

7.1 Conclusion

Pedestrian and vehicular traffic in the Digital Twin of Nicosia can benefit researchers for visualising crowd dynamics in the city, as well as users that want to experience a more realistic virtual museum.

The proposed system first extracts the road network from the selected part of the Digital Twin, runs the simulation on it using SUMO, and visualises the results in Unity. Running the simulation and visualising the results are done separately and use server-client communication over UDP. This means that after running the simulation, the data can be sent to any visualisation engine. Unity was chosen for this project, but any other engine can be used, assuming the Cesium ion model of the Digital Twin can be imported into it.

Results showed that this system can work for visualising the simulation on the Digital Twin but with some limitations. These were caused mainly from the inconsistency between the OSM road network and the Cesium ion road network from the Unity imported assets.

One of the biggest issue is the inconsistency between OSM data about the road networks and the corresponding data in the Cesium ion model. In places where the misalignment happened, agents in that position would be visualised in the wrong place in Unity.

In addition, OSM road networks do not include all data on sidewalks and pedestrian crossings. Therefore, sidewalks generated from SUMO could sometimes overlap with buildings, and pedestrian crossings might not match the real-world ones.

Lastly, some lanes in the converted SUMO network were shared by pedestrians and vehicles. This caused some pedestrian routes to overlap with vehicle routes, resulting in heavy traffic.

7.2 Future Work

The current system spawns the agents at random positions within the converted city network. This is done by selecting random edges that are valid for the specific agent type each time and then finding valid routes to reach the desired destination. This method does not limit the selection to edges corresponding to specific positions in the city. Therefore, live data from sensors in the actual city can be fed to the system and visualised as agents in the simulation.

TraCI subscriptions can be used to automatically receive relevant information from agents, rather than querying the simulation separately for each variable at every step. This can improve performance [27] and could be added to the current implementation. Similarly, libsumo eliminates the need for TCP communication with the SUMO simulation by adding SUMO as a library, which can also improve performance.

Finally, the photorealistic version of the Digital Twin of Nicosia can be used instead of the specific assets used for this project to improve the realism and visual quality of the simulation.

BIBLIOGRAPHY

[1] "What is a Smart City?" https://www.ibm.com/think/topics/smart-city, accessed: 02/05/2026.

[2] S. Mazzetto, "A review of urban Digital Twins integration, challenges, and future directions in Smart City development," *Sustainability*, vol. 16, no. 19, p. 8337, 2024. [Online]. Available: https://doi.org/10.3390/su16198337

[3] "SUMO Traffic Simulation Theory," https://sumo.dlr.de/docs/Theory/Traffic_Simulations.html, accessed: 02/05/2026.

[4] S. Zemouri, S. Mehar, and S.-M. Senouci, "HINTS: A novel approach for realistic simulations of vehicular communications," in *2012 Global Information Infrastructure and Networking Symposium*. IEEE, 2012.

[5] M. Moussaïd, D. Helbing, and G. Theraulaz, "How simple rules determine pedestrian behavior and crowd disasters," *Proceedings of the National Academy of Sciences*, vol. 108, no. 17, pp. 6884–6888, March 2011.

[6] A. Panayiotou, T. Kyriakou, M. Lemonari, Y. Chrysanthou, and P. Charalambous, "CCP: Configurable crowd profiles." Association for Computing Machinery, July 2022.

[7] "Unity NavMesh Documentation," https://docs.unity3d.com/6000.4/Documentation/ScriptReference/AI.NavMesh.html, accessed: 01/05/2026.

[8] "SUMO Documentation," https://sumo.dlr.de/docs/, accessed: 02/05/2026.

[9] J. Erdmann and D. Krajzewicz, "Modelling pedestrian dynamics in SUMO," May 2015.

[10] "netconvert Documentation," https://sumo.dlr.de/docs/netconvert.html, accessed: 02/05/2026.

[11] "Netedit Documentation," https://sumo.dlr.de/docs/Netedit/index.html, accessed: 04/05/2026.

[12] "Random Trips Documentation," https://sumo.dlr.de/docs/Tools/Trip.html, accessed:

04/05/2026.

[13] “duarouter Documentation,” https://sumo.dlr.de/docs/duarouter.html, accessed: 04/05/2026.

[14] “SUMO Command Line Applications Documentation,” https://sumo.dlr.de/docs/Basics/Using_the_Command_Line_Applications.html, accessed: 04/05/2026.

[15] “TraCI Documentation,” https://sumo.dlr.de/docs/TraCI/index.html, accessed: 04/05/2026.

[16] “TraCI Pedestrian Value Retrieval Documentation,” https://sumo.dlr.de/docs/TraCI/Person_Value_Retrieval.html, accessed: 05/05/2026.

[17] “TraCI Edge Value Retrieval Documentation,” https://sumo.dlr.de/docs/TraCI/Edge_Value_Retrieval.html, accessed: 05/05/2026.

[18] “TraCI Lane Value Retrieval Documentation,” https://sumo.dlr.de/docs/TraCI/Lane_Value_Retrieval.html, accessed: 05/05/2026.

[19] “OpenStreetMap,” https://www.openstreetmap.org/, accessed: 04/05/2026.

[20] “Unity,” https://unity.com/, accessed: 05/05/2026.

[21] “Unity Physics.Raycast Documentation,” https://docs.unity3d.com/ScriptReference/Physics.Raycast.html, accessed: 18/05/2026.

[22] “Cesium,” https://cesium.com/, accessed: 05/05/2026.

[23] “Cesium for Unity,” https://cesium.com/learn/unity/, accessed: 01/05/2026.

[24] “Cesium Georeference Documentation,” https://cesium.com/learn/cesium-unity/ref-doc/classCesiumForUnity_1_1CesiumGeoreference.html, accessed: 05/05/2026.

[25] “iNicosia,” https://inicosia.cyens.org.cy/, accessed: 18/05/2026.

[26] “City Package,” https://assetstore.unity.com/packages/3d/environments/urban/city-package-107224, accessed: 10/05/2026.

[27] C. Chadjiminas, “3D traffic simulation in Unity: How to use SUMO to produce 3D traffic

in Unity," June 2019.

[28] "SUMO Specification Documentation," https://sumo.dlr.de/docs/Specification/index.html, accessed: 06/05/2026.

[29] G. G. Løvås, "Modeling and simulation of pedestrian traffic flow," *Transportation Research Part B: Methodological*, vol. 28, no. 6, pp. 429–443, 1994.

[30] "Unity Netcode Interpolation Documentation," https://docs.unity3d.com/Packages/com.unity.netcode@1.3/manual/interpolation.html, accessed: 12/05/2026.