Real-Time Traffic Simulation

Alanna Søpler

N00192732

Report submitted in partial fulfilment of the requirements for the BSc (Hons) in Creative Computing at the Institute of Art, Design and Technology (IADT).

Declaration of Authorship

The incorporation of material without formal and proper acknowledgement (even with no deliberate intent to cheat) can constitute plagiarism.

If you have received significant help with a solution from one or more colleagues, you should document this in your submitted work and if you have any doubt as to what level of discussion/collaboration is acceptable, you should consult your lecturer or the Programme Chair.

WARNING: Take care when discarding program listings lest they be copied by someone else, which may well bring you under suspicion. Do not to leave copies of your own files on a hard disk where they can be accessed by others. Be aware that removable media, used to transfer work, may also be removed and/or copied by others if left unattended.

Plagiarism is considered to be an act of fraudulence and an offence against Institute discipline.

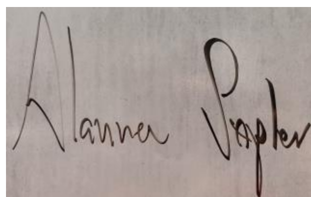
Alleged plagiarism will be investigated and dealt with appropriately by the Institute. Please refer to the Institute Handbook for further details of penalties.

The following is an extract from the B.Sc. in Computing (Hons) course handbook. Please read carefully and sign the declaration below

Collusion may be defined as more than one person working on an individual assessment. This would include jointly developed solutions as well as one individual giving a solution to another who then makes some changes and hands it up as their own work.

Declaration

I am aware of the Institute’s policy on plagiarism and certify that this thesis is my own work.

Signed: _______________________________________

Date: _____10.05.2024____

Failure to complete and submit this form may lead to an investigation into your work.

Abstract

This document contains research around Simulations to determine if it can be used to visualize problematic traffic and traffic flows in problematic areas. This was achieved though the creation of a Unity project taking network and vehicle information and positioning from SUMO Traffic Simulator. The user will be able to visualize actual networks from their local communities. The approach began with researching the different types of simulation tool and different companies' utilization of simulation tools. Focusing on what are important aspects of a simulation that are helpful for the user. Following the implementation of the simulation and thereafter performance testing of the network. Following with the result of the project and any further conclusions.

Acknowledgements

I would like to extend my thanks to the people who showed support and offered advice throughout the duration of this project. I would like to thank my project supervisor, Joachim Pietsch Naoise Collins, and my second reader Cyrill Connolly, for their help, communication, and support throughout the development.

Contents

Declaration 2
Abstract 3
Acknowledgements 4
Contents 5
1. Introduction 8
2. Research 9
Traffic Simulation 9
Broader Context and Importance of Traffic Simulations 9
Understanding Simulations and Models 9
The Process of Building a Model 10
Introduction to Traffic Simulation Modeling 11
Macroscopic Modelling Overview 11
Microscopic modelling Overview 12
Mesoscopic Modelling Overview 13
Traffic Simulation Tools 15
Fundamental Concepts 17
Aimsun Traffic Simulation 20
Core Models 20
Microscopic Logic 20
Mesoscopic Logic 21
SUMO 22
Preparing a Road Network to Simulate 23

Microscopic Demand Definitions 24
Importing and Using Origin/Destination Matrices 24
Challenges and Solutions in Demand Modeling 24
Why choose SUMO. 26
Unity as a 3D visualization tool 28
3. Requirements 31
Ford's Traffic Jam Assist 31
Tesla Autopilot 33
Functional Requirements 35
Non-Functional Requirements 35
Use Case Diagram 36
Conclusion 37
4. Design 38
Design 38
Technologies 38
Visual Studio Code & C# 38
Unity Hierarchy and Inspector 40
The Toolbar 41
The Assets folder 42
SUMO 45
Tool Bar 48
UX Design 50
Implementation (or Construction) 52
Introduction 52

Connection between SUMO and Unity 52
Set up Unity and SUMO connection. 55
Setting up the Vehicles 58
Traffic Lights 64
Render Lanes 72
The Ear Clipping Algorithm 85
The JunctionRender Script 88
SUMO Network 93
Net.xml file 93
Rou.xml 96
.sumocfg file 97
Creating the network 98
Generate random traffic flow 99
5. Testing and Analysis 102
6. Discussion 106
Discusstion 106
Future Development 108
7. Conclusion 110
References 111

1. Introduction

The initiative to create a traffic simulation was sparked by persistent local challenges in the outskirts of Dundrum, where residents face daily hazards navigating a heavily trafficked area devoid of pedestrian crossings. The locality, constrained by a stream and fencing, offers limited egress options from the residential zones. Despite ongoing appeals to the local council, efforts to improve this situation have remained fruitless. Notably, a pedestrian crossing exists near a pub that was closed during the pandemic, highlighting inconsistent infrastructure development. This simulation aims to demonstrate the urgent need for safer pedestrian pathways to the council, advocating for immediate action.

2. Research

Traffic Simulation

Broader Context and Importance of Traffic Simulations

As urban populations swell, cities are experiencing escalating vehicular density, intensifying transportation dilemmas that demand optimized road network solutions. Effective planning and modeling of transportation infrastructure are crucial for addressing these challenges. According to (Dorokhin, S. 2020), motion modeling is a cornerstone of traffic studies, which includes planning and evolving transport networks. By creating models that mirror real-world properties and dynamics, researchers can examine intricate traffic scenarios in controlled settings rather than unpredictable real-life conditions. Such modeling is often categorized into microscopic, macroscopic, and mesoscopic, each tailored to address prevalent transportation issues by allowing analysis of critical parameters like traffic volume, average speeds, delays, and time losses (Gorodnichev M., 2022).

Understanding Simulations and Models

Simulations are powerful tools for replicating and understanding real-world processes through mathematical models. As Jordi Vallverdú (2013) describes in 'What are Simulations? An Epistemological Approach,' simulations are our best mental representations of reality, functioning through systematic processes that mimic how things work in the real world. Similarly, Anu Maria emphasizes that a simulation operates a model of a system, highlighting the dynamic nature of this technology.

While both models and simulations serve to understand and predict complex phenomena, they are not interchangeable. A model is a simplified representation of reality, often a formal system that encapsulates essential features of a physical system, designed to test hypotheses, or understand mechanisms. According to Jaume Barceló (2010), building a model requires deep knowledge of the system and is based on assumptions expressed through mathematical or logical relationships.

The systemic approach is crucial in modeling. It views a system as more than the sum of its parts, focusing on interactions and interdependencies that lead to emergent properties not predictable by studying individual components alone. This concept is vital in traffic simulations where multiple dynamic interactions occur. For instance, Fundamentals of Traffic Systems (2010) outlines how models use a systemic approach to study complex traffic behaviors effectively.

The Process of Building a Model

Model building is an iterative learning process that aims to understand and solve the problems associated with the modeled system. It begins with system analysis, identifying all relevant components and their relationships. This step is crucial for developing an accurate model that reflects the complexities of real-world traffic systems. The model undergoes continuous refinement and validation to ensure it accurately represents the studied phenomena and can predict future states or behaviors effectively.

Introduction to Traffic Simulation Modeling

Traffic simulation modeling is an essential tool in understanding and optimizing the movement of people and goods, which are driven by daily social and economic activities. According to Boris S. Kerner and J. Barcélo (2010), efficient transportation systems are crucial for ensuring that mobility contributes positively to market demands.

Traffic simulation models vary in complexity. Basic Models might simplify intersections as single points for general studies. Advanced Models for more precise analysis, intersections are detailed with specific turning lanes and signal phases. Such models consider various elements like lane configurations, traffic volumes, delay functions, and explicit traffic control measures. For comprehensive simulations, it is crucial to incorporate the actual geometry of roads and intersections (Aume Barceló, 2010). This includes specifics such as lane numbers, widths, speed limits, and other infrastructure like traffic signals and surveillance systems. The granularity of the model depends on the intended use of the simulation, whether for high-level planning or detailed operational analysis.

Macroscopic Modelling Overview

Macroscopic traffic models describe the collective behavior of traffic flows over large stretches of road, rather than tracking individual vehicles (Aume Barceló, 2010). The objective is to model the evolution of traffic variables like volume, speed, and density over space and time (Aume Barceló, 2010). In practice, macroscopic models help traffic engineers and planners to predict and manage traffic flows, design road systems, and

implement traffic control measures effectively. These models are integral in simulations that support decision-making for urban development and infrastructure projects.

By understanding these concepts and their mathematical foundations, traffic models can be effectively implemented in traffic simulation software like SUMO, which can then be visualized and interacted with using tools like Unity for comprehensive traffic management and planning solutions.

Microscopic modelling Overview

Microscopic modelling studies the behavior of drivers and vehicles in traffic, including acceleration, deceleration, and lane changes (Aume Barceló, 2010). The model is based on detailed observations and mathematical formulations that describe how each vehicle interacts with its immediate surroundings, particularly the vehicle directly in front.

The development of microscopic traffic models started in the mid-20th century with pioneers like Reuschel and Pipes, who introduced car-following theories that describe how drivers maintain safe distances based on their speeds (Aume Barceló, 2010). Pipes' Theory introduced the concept that a driver should maintain one car length for every ten miles per hour of speed, which provides a simple yet effective rule for safe following distances.

The Linear Car-Following Model is a fundamental model where the following vehicle's acceleration or deceleration is directly proportional to the speed difference between it and the vehicle in front (Aume Barceló, 2010). This model assumes that the response (acceleration or deceleration) is linearly dependent on the stimulus (the relative speed).

Gazis, Herman, and Rothery proposed refinements to incorporate more realistic behaviors and adjust the sensitivity based on the headway (distance between two vehicles), improving the model's accuracy under different traffic conditions (Aume Barceló, 2010).

Microscopic models require extensive field data to calibrate and validate their assumptions. The accuracy of these models in predicting real-world traffic behaviors is crucial for their effectiveness in traffic management systems (Aume Barceló, 2010).

These models are implemented in traffic simulation software like SUMO, which allows for detailed analysis and visualization of traffic flows and helps in designing better traffic management strategies.

Mesoscopic Modelling Overview

Mesoscopic modeling is a method of traffic flow modeling that combines elements of both microscopic and macroscopic modeling (Aume Barceló, 2010). It helps to capture the essential dynamics of traffic flow without requiring the intensive data and computational demands of microscopic modeling.

The Mesoscopic models handles the behavior of individual vehicles or groups of vehicles (platoons) while also considering aggregated traffic dynamics like flow and density across larger segments of the traffic network (Aume Barceló, 2010).

There are two main approaches. Platoon-based models group vehicles into platoons, which move through the traffic network, reducing computational complexity (Aume Barceló, 2010). The second approach is Simplified vehicle dynamics, which uses

straightforward rules for individual vehicles' dynamics but don't require as detailed data as microscopic models (Aume Barceló, 2010).

Many mesoscopic models advance time in fixed steps, known as the simulation step, or asynchronously when events occur, such as a vehicle entering or leaving a link (Aume Barceló, 2010).

Link Modelling in Mesoscopic Traffic Simulations are typically divided into two segments. There is the Running Part and Queue Part. In the running part, vehicle dynamics may be governed by simplified car-following models aligned with macroscopic speed-density relationships, whereas in the queue part, the dynamics are controlled by the queue discharge processes (Aume Barceló, 2010).

Mesoscopic models are particularly valued for their balance between detail and computational efficiency. They are better suited for larger-scale simulations where microscopic details are less critical, but some individual vehicle interactions still need to be considered.

In conclusion, mesoscopic models offer a practical compromise between detailed microscopic models and broad macroscopic models. They provide sufficient detail for many applications while keeping computational demands reasonable, making them ideal for studying traffic dynamics over large areas where detailed individual behavior is less significant than the overall flow patterns.

Traffic Simulation Tools

Traffic simulation is a crucial tool for transport planners and traffic engineers. VISSIM is a microscopic, behavior-based traffic simulation that can analyze and optimize traffic flows. It has a wide range of urban and highway applications, integrating public and private transportation (Loren B., 2000).

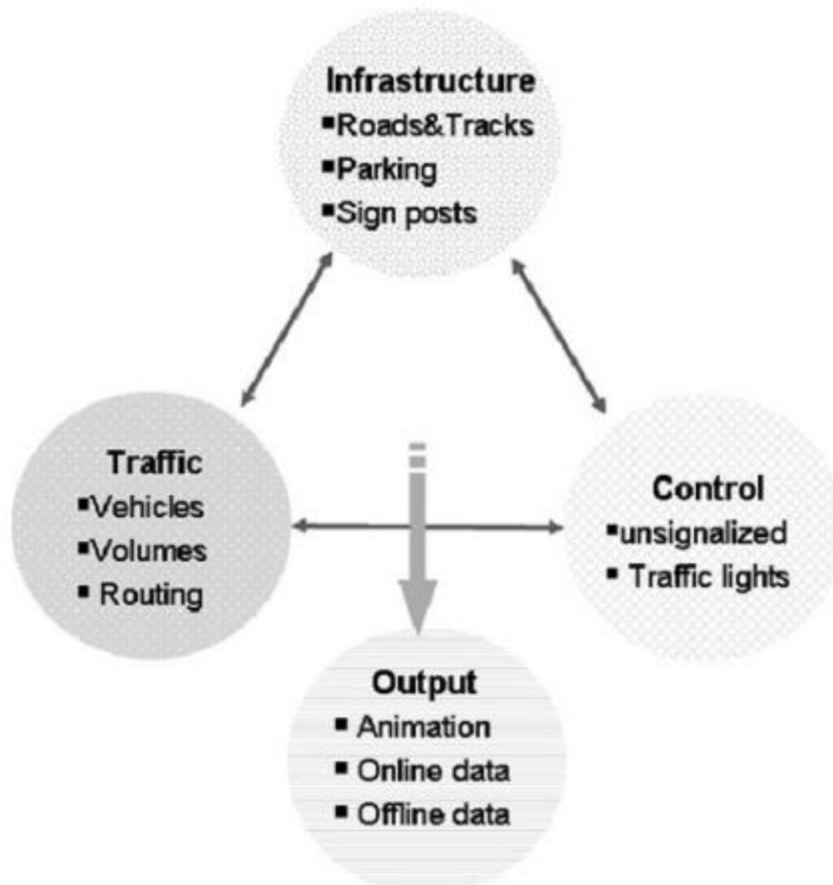
The software is primarily designed for traffic engineers, but an increasing number of transport planners are also using microsimulation (Aume Barceló, 2010). VISSIM is a microscopic traffic simulation system that models both motorway and urban traffic operations using various mathematical models (Aume Barceló, 2010). The system can be used to investigate private and public transport, as well as pedestrian movements. Traffic engineers and transport planners can create applications by selecting appropriate objects from a variety of primary building blocks. The system provides technical features for pedestrians, bicyclists, motorcycles, cars, trucks, buses, trams, lights, and heavy rail, and allows for customization options to simulate multi-modal traffic flows.

In traffic simulation systems like VISSIM, the complex interactions and dynamics of real-world traffic are modeled using a structured framework that divides the modeling environment into distinct but interrelated blocks (Aume Barceló, 2010).

The infrastructure block is the foundation of the simulation model where all physical and static components of the transportation system are represented (Aume Barceló, 2010). This includes roads and railways which are modeled to reflect the real-world network

including the number of lanes, lane widths, types of roads (motorways, arterial roads), and railway tracks (Aume Barceló, 2010).

The traffic flow block consists of the elements that make up the traffic system, namely the vehicles and their movement patterns (Aume Barceló, 2010). Different types of vehicles, including cars, buses, trucks, and bicycles, are modeled with specific attributes such as speed capabilities, dimensions, and other technical specifications.



Each of these blocks interacts with others to create a comprehensive model of the traffic system. For instance, changes in the Traffic Control Block can affect traffic flows and behaviors, which in turn may necessitate adjustments in infrastructure planning in the

Infrastructure Block (Aume Barceló, 2010). The Evaluation Block helps in understanding the outcomes of these interactions and in planning future improvements.

These interdependencies ensure that the simulation is dynamic and reflects the complex nature of real-world traffic systems, allowing planners and engineers to test and refine their approaches before implementing actual changes on the ground. Such simulations are invaluable tools in urban planning and traffic management, providing insights that help in making informed decisions to enhance the efficiency and safety of transportation systems.

Fundamental Concepts

Car-Following Models are used to simulate how drivers follow one another on the road, considering factors like vehicle speeds, distances, and driver reactions to changing traffic conditions (Aume Barceló, 2010).

Psycho-Physical Models approach integrates psychological perceptions (like visual cues) and physical actions (like braking or accelerating) to model driver behaviour.

Key Parameters and Their Functions :

- ax (Minimum Jam Distance): Represents the minimum distance a driver maintains from the vehicle in front when all vehicles are stopped (like at a traffic light) (Aume Barceló, 2010).

- abx (Minimum Following Distance): This is the critical distance below which a driver reacts to slow down to avoid getting too close to the vehicle in front.

- sdv (Speed Difference Threshold): The threshold speed difference at which a driver starts to notice that the leading vehicle is moving slower and may initiate deceleration.

- opdv (Overtake Preparation Distance Velocity): The point at which a driver realizes they are moving faster than the vehicle ahead and prepares to overtake or accelerate.

- sdx (Maximum Following Distance): This defines the farthest distance at which a driver still considers the behavior of a vehicle ahead in their driving decisions, typically several times the abx distance.

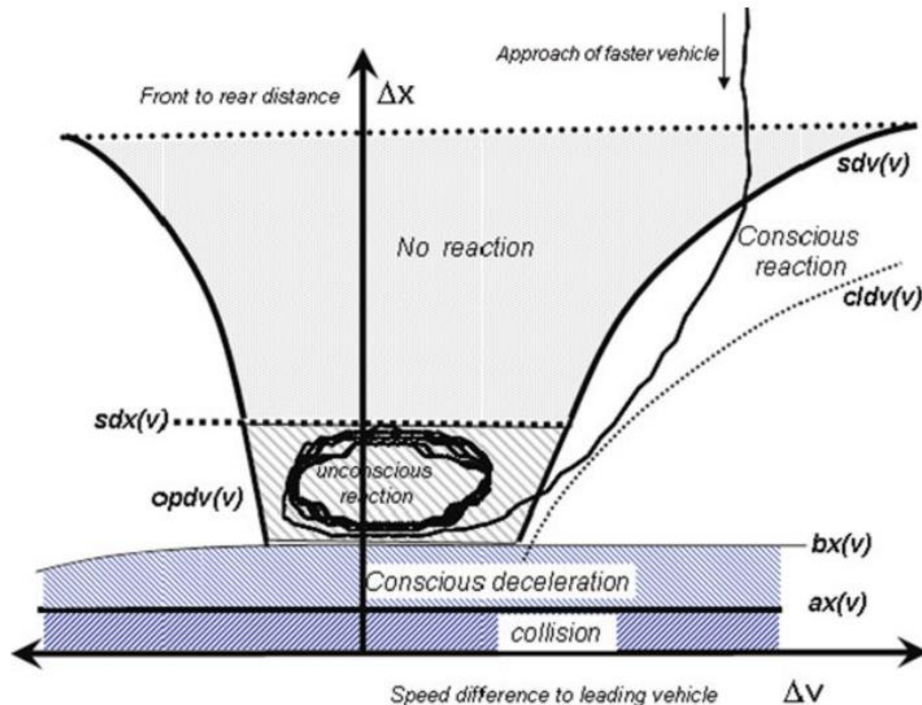


Fig. 2.7 Psycho-physical car-following model by Wiedemann

The dynamics described by these models show how a driver transitions from noticing a slower vehicle to deciding when to slow down or speed up based on the relative speeds and distances. The parameters ax, abx, sdv, and opdv are instrumental in defining how these transitions occur. The model also accounts for different driver risk profiles—risk-averse versus risk-taking—which influence how closely drivers follow the vehicle ahead (Aume Barceló, 2010).

The model has the ability to handle sudden traffic changes, including scenarios where drivers exceed normal deceleration rates to avoid collisions. This allows traffic engineers and planners to adjust thresholds and parameters to model and predict road capacities under various traffic conditions and driver behaviors. By doing so, they can design more efficient traffic management systems, road layouts, and safety measures.

Aimsun Traffic Simulation

Aimsun is a software that was originally developed as a project at the University of Catalonia (UPC) under the acronym "Advanced Interactive Microscopic Simulator for Urban and Non-Urban Networks" or AIMSUN (Aume Barceló, 2010). It started as a traffic simulation software that could handle a wide range of traffic simulations, including macroscopic, mesoscopic, and microscopic models. The software has now evolved and is currently in its sixth major commercial release.

Aimsun is used for traffic planning and analysis tasks that require detailed simulation of traffic patterns but do not need to be conducted in real-time (Aume Barceló, 2010). Aimsun has grown from a project focused on specific simulation needs to a robust platform that accommodates a broad spectrum of traffic modeling requirements (Aume Barceló, 2010). It offers sophisticated tools for both detailed traffic engineering and real-time management, making it a versatile choice for modern traffic solutions. It serves as a tool for specific traffic simulation tasks, as well as a component of larger, integrated traffic management and planning strategies.

Core Models

Microscopic Logic

The microscopic model of traffic simulation uses a time slice-based approach, which is enhanced with a scheduled event calendar (Aume Barceló, 2010). This enables the simulation process to update traffic control events, such as traffic light changes, and the

statuses of roads and intersections in real-time. Every cycle updates the status of all entities, vehicles and processes new vehicle entries, and data collection.

Mesoscopic Logic

The mesoscopic model of traffic simulation operates on a discrete-event basis, where the simulation clock advances between events, instead of fixed time slices (Aume Barceló, 2010). This approach allows for dynamic adjustments based on events such as vehicle movements, traffic light changes, and traffic demand updates. Key events include vehicle generation, traffic control changes, and data collection. These events guide the simulation, adjusting the network state dynamically to reflect both scheduled and conditional changes.

Modeling Microscopic

Two main driver behavior models, car following and lane changing, dictate how vehicles interact on the road (Aume Barceló, 2010). These models consider factors such as vehicle speed, surrounding vehicles, and road conditions. Based on Gipps' model, this approach calculates acceleration and deceleration based on the vehicle's current speed and desired speed, considering both the vehicle's capabilities and the influence of leading vehicles.

In essence, Aimsun's approach to traffic simulation, with its sophisticated models of car following and lane changing, allows for a nuanced understanding of vehicular dynamics. It offers robust tools for managing both everyday traffic flow and unusual or emergent traffic conditions effectively.

SUMO

SUMO (Simulation of Urban Mobility) is a microscopic road traffic simulation software that was developed by the Center for Applied Informatics Cologne and the Institute of Transportation Systems at the German Aerospace Center in 2000 (Aume Barceló, 2010). The software has undergone significant evolution since its inception and has become an integral part of various traffic management projects.

SUMO was developed as a comprehensive and open-source traffic simulation tool that could be easily adapted and reused in academic research (Aume Barceló, 2010). It was released under the GNU General Public License, which ensures that it is freely accessible and modifiable. The software was designed to run on various operating systems and to simulate large city areas efficiently, making it ideal for handling complex simulations without excessive computational demands.

SUMO has been used to evaluate traffic light systems, simulate traffic management strategies, and explore the impacts of various traffic management technologies. It is suitable for real-time applications in traffic management and forecasting.

SUMO's development philosophy ensures that it remains relevant and progressively enhanced by a growing community of users and developers. The software has become increasingly popular among traffic professionals and computer scientists due to its open-source nature, adaptability, and the breadth of its application in traffic management research.

Preparing a Road Network to Simulate

The primary method for setting up road networks in SUMO involves importing existing digital road network data. SUMO uses NETCONVERT Tool, which is SUMO's network importer tool, capable of reading various formats such as VISUM, TIGER, ArcView shape files, Vissim, Robocup Rescue League folders, OpenStreetMap, and native XML representations of road networks (Aume Barceló, 2010). Given that many imported network formats are not directly suited for microscopic traffic simulation (which requires detailed lane-level information, intersection behaviors, traffic light statuses, etc.), SUMO uses the NETCONVERT tool to calculates which turns are allowed from each road. Also, determining how lanes are connected across roads and at intersections. Furthermore, identifying the type of intersection (stop, signalized) and establishing priority rules. Lastly, the placement and operational logic of traffic lights, and refines the geometric layout of roads and intersections. Despite this, the initially converted network often requires manual inspections and corrections to address any inaccuracies or missing data.

Microscopic Demand Definitions

SUMO's preferred format involves a list of vehicles with specified departure times and the roads they travel between. This data allows SUMO to compute complete paths through the network using shortest path algorithms, resulting in a detailed set of vehicle routes for the simulation (Aume Barceló, 2010). For more dynamic and detailed traffic modeling, SUMO incorporates methods that simulate dynamic user equilibrium. This technique adjusts vehicle routes based on current traffic conditions, enhancing the realism of the simulation.

Importing and Using Origin/Destination Matrices

The conversion process typically involves converting Origin/Destination (O/D) matrices into lists of trip details for individual vehicles. This is followed by a dynamic user assignment to realistically distribute these trips across the traffic network.

Challenges and Solutions in Demand Modeling

While detailed per-vehicle data is rare and labor-intensive to generate due to the need for extensive sociological data, it is ideal. Synthetic populations or agent-based models can provide this data. More commonly, SUMO simulations rely on importing O/D matrices, which are more readily available and provide a satisfactory level of detail for large-scale simulations.

The granularity of O/D matrices should match the simulation's scale. High-detail simulations require fine-grained O/D data that aligns closely with the network's physical roads rather than aggregated district-level data.

Sometimes, manual adjustments and additional planning are necessary to map O/D data effectively onto the simulated network, ensuring that traffic patterns and vehicle behaviors in the simulation closely mimic real-world conditions.

Traffic demand modeling in SUMO involves creating realistic vehicle movements within simulations. This requires defining each vehicle's route, departure time, and other characteristics. There are different methods to establish this demand, ranging from using highly detailed individual trip information to leveraging aggregated origin-destination (O/D) matrices that represent broader traffic flows.

Why choose SUMO.

Traffic simulation tools are essential for transport planners and traffic engineers to analyze, optimize, and visualize traffic flows. While tools like VISSIM offer detailed behavior-based microsimulation capabilities across various urban and highway contexts, SUMO (Simulation of Urban MObility) stands out due to its adaptability and extensive applicability in both academic research and practical traffic management.

Aimsun offers advanced features and integrated modeling environments for microscopic, mesoscopic, and macroscopic simulations. However, it is a commercial product, and organizations must purchase licenses.

SUMO, on the other hand, is an open-source, microscopic road traffic simulation tool that provides high adaptability and extensive customization for scenarios that require it. It is freely available under the GNU General Public License, allowing users to modify the code to suit specific project needs without the constraints of licensing fees or proprietary restrictions.

SUMO can run on multiple platforms and support large-scale simulations. It can integrate easily with other simulation packages, allowing for the importation of detailed road networks and demand data from diverse sources.

SUMO excels in simulating complex urban mobility scenarios. It can model a wide array of vehicles and traffic behaviors, making it ideal for real-time applications in traffic management and forecasting, such as evaluating traffic light systems or exploring the impacts of traffic management technologies.

SUMO divides the traffic modeling environment into structured blocks that interact dynamically, enhancing the realism of simulations. The Infrastructure Block models all physical components of the transportation system, including intricate road and railway networks. The Traffic Flow Block focuses on the vehicles themselves and their movement patterns. The Traffic Control Block includes mechanisms for detailed control of the traffic system, such as traffic lights and priority rules. The Evaluation Block provides comprehensive outputs from the simulation, such as travel times, queue lengths, and throughput metrics.

SUMO's flexibility is evident in its ability to be tailored for specific research or practical applications. Whether it's a detailed analysis of pedestrian movements, public transport systems, or multi-modal traffic flows, SUMO provides the tools necessary to create precise and adaptable simulation environments.

For organizations and researchers who require a robust, customizable, and cost-effective solution for traffic simulation, SUMO offers significant advantages. Its ability to handle complex, large-scale simulations and adapt to specific project requirements makes it an invaluable tool in both academic research and practical traffic management solutions.

Unity as a 3D visualization tool

Unity is a powerful software that is highly regarded for its robust features, making it an excellent choice for 3D visualization across a variety of industries. This software can handle real-time position information and display it in 3D, which is important for dynamic simulations like traffic and autonomous vehicle navigation (Ismail B.,2017).

An article titled "3D Traffic Simulation for Autonomous Vehicles in Unity and Python" discusses a simulation that integrates real-time positional data obtained from street cameras. This data is then used to animate and control the movements of vehicles within the Unity simulation environment. This means that Unity can handle live data feeds, which is crucial for scenarios that depend on up-to-the-minute data, such as traffic management systems or real-time testing of autonomous vehicle algorithms. Unity also offers the capability to switch between a global bird's-eye view and a local ground-level perspective of individual vehicles seamlessly. This flexibility is key for analyzing traffic flow from different vantage points (Ismail B.,2017).

Furthermore, Unity not only visualizes traffic but also serves as a reinforcement learning platform. It accepts user inputs to control vehicles within the simulation, providing a feedback loop to deep learning programs (Ismail B.,2017). The system can simulate a variety of traffic scenarios and collect data on vehicle behavior in response to different driving commands, contributing to the training of more robust autonomous driving models.

Another article titled "Vehicle-Pedestrian Interaction in SUMO and Unity3D" highlights the benefits of using Unity as a visualization tool in a few significant ways. The article emphasizes the integration of Unity with the Simulation of Urban Mobility (SUMO) via the Traffic Control Interface (TraCI) Protocol and TraCI as a Service (TraaS) library. This integration leverages Unity's powerful 3D graphics engine to visualize traffic scenarios generated in SUMO, which is primarily a 2D traffic simulation tool. This combination brings the detailed traffic modeling capabilities of SUMO into the visually rich 3D world of Unity, enhancing both the analytical and experiential aspects of traffic simulation studies.

Unity's ability to handle real-time data and update the simulation environment accordingly is crucial for studying dynamic interactions and conducting responsive analyses. In this setup, pedestrian and vehicle data from SUMO are sent to Unity, where they are visualized in real time. This feature is essential for applications such as testing pedestrian safety measures or vehicle routing algorithms under varied traffic conditions.

The use of Unity enables flexible visualization options, such as switching between different camera views—such as bird's-eye view or ground-level perspectives—thus offering diverse insights into traffic dynamics and interactions. This capability is invaluable for developing advanced driver-assistance systems (ADAS) and autonomous vehicle systems, where understanding the environment from multiple perspectives is crucial.

By simulating the interaction between pedestrians and vehicles in a 3D space, Unity allows researchers and developers to observe and analyze the impact of various factors on road safety. This can include visualizing the movement of vulnerable road users (VRUs) near vehicles, which is crucial for developing systems aimed at reducing pedestrian fatalities and enhancing urban traffic safety.

Unity’s robust development environment allows for extensive customization and scalability. Users can develop custom scripts and integrate various sensors and data inputs to extend the simulation capabilities according to specific research needs or project requirements.

Overall, the article underscores Unity's role in advancing traffic simulation by providing a visually engaging and technically robust platform that complements traditional traffic simulation tools like SUMO. This integration enhances the practicality and applicability of traffic simulations in urban planning, autonomous vehicle development, and safety analysis, demonstrating the significant benefits of using Unity as a visualization tool in these contexts.

3. Requirements

In this chapter the requirements for the application will be explored and discussed. Research for application development will involve searching for similar existing simulations. The requirements, both functional and nonfunctional, will then be listed, to get an overview of the most important features to include.

Ford's Traffic Jam Assist

Ford has integrated various Advanced Driver-Assistance Systems (ADAS) in its vehicles to improve safety and enhance the driving experience, especially in traffic. These ADAS features are discussed in detail in the article "Application of Advanced Driver-Assistance Systems in Police Vehicles." This article is helpful for those interested in exploring traffic simulation or data visualization for safety technologies in specialized vehicles, such as police cars.

Advanced Driver-Assistance Systems (ADAS) have been developed to help manage and improve traffic flow in congested road conditions. Traffic Jam Assist is one such example of ADAS that specifically aims to enhance traffic flow. However, there are several other types of ADAS that can also be beneficial for traffic management.

Traffic Jam Assist combines the functionalities of adaptive cruise control and lane-keeping assistance to help the driver navigate through congested traffic with less effort. It controls the car's acceleration, braking, and steering during heavy traffic scenarios at low speeds.

Simulation tools are particularly valuable for developing and refining systems like Traffic Jam Assist (TJA), which are designed to operate in complex, dynamic environments like congested traffic scenarios. Here's how these tools are typically used in the context of TJA development:

Traffic Jam Assist needs to be effective in various traffic conditions and setups. Simulation tools allow developers to create diverse traffic jam scenarios, including varying levels of congestion, different types of vehicles, and various road layouts. This helps in testing how the TJA system reacts to slow-moving traffic, sudden stops, and lane changes by other vehicles. TJA relies heavily on data from sensors like cameras, radar, and ultrasonic sensors to monitor the vehicle's surroundings closely. Simulation tools can generate synthetic sensor inputs that mimic real-world data, allowing the TJA system to be tested against a wide array of traffic situations without the need for costly real-world driving tests.

Simulation tools also enable testing of the user interface and interactions between the driver and the TJA system. This includes how information is displayed to the driver and how the driver can override or interact with the system if needed. Ensuring that these interactions are intuitive and safe is crucial for the acceptance and efficacy of TJA systems.

Traffic Jam Assist functions must adhere to various safety standards and regulations. Simulations can be used to demonstrate compliance with these standards by showing how the system performs in scenarios that could be risky or dangerous. This includes testing

the system’s ability to prevent collisions, maintain safe following distances, and safely execute lane-keeping maneuvers.

By leveraging simulation tools, developers can ensure that Traffic Jam Assist systems are well-prepared to handle real-world conditions safely and effectively, enhancing both the technology's reliability and the driver's trust in the system.

Tesla Autopilot

Autonomous driving systems have been available for some time now, and Tesla's Autopilot was one of the first commercially available systems. However, there have been concerns about the safety of the Autopilot system following multiple accidents involving its use. The article "Survey on Autonomous Vehicle Simulation Platforms" cites specific incidents, such as the 2016 accident in Florida where the Autopilot system failed to differentiate a white truck from the sky, resulting in a fatal crash.

It is evident that extensive simulation testing is necessary considering the Autopilot incidents. These simulations allow developers to observe how the system behaves in diverse, controlled environments, which can replicate rare or dangerous situations that are difficult to test safely in real-world conditions. The article suggests that continuous refinement through simulations can enhance the functionality and safety of systems like Tesla's Autopilot, potentially reducing the likelihood of accidents and improving public trust in autonomous vehicle technology.

The article discusses "Dynamic Environment and Behavior Simulation" as an advanced approach to enhance the realism and applicability of simulation technologies for autonomous driving systems like Tesla's Autopilot. This section explains how dynamic simulations incorporate both environmental variability and the complex behaviors of different actors within the traffic ecosystem, such as pedestrians, cyclists, and other vehicles.

Dynamic simulations are critical because they provide a controlled yet realistic and interactive setting for testing autonomous driving systems. They allow developers to evaluate how these systems react to changes in the environment, such as varying weather conditions, unexpected pedestrian movements, or sudden changes in traffic flow. This is crucial for developing robust ADAS systems that must reliably make split-second decisions in real-world driving situations.

Functional Requirements

1. Real-time Response: The system should be able to process data and respond in real-time to accurately simulate the decision-making process of autonomous vehicles in dynamic environments.

2. Environment Simulation: The system should be able to create varied driving environments, including different weather conditions, road types, and traffic densities, to test the robustness of autonomous driving technologies.

3. Actor Behavior Modeling: Simulation tools should realistically model the behavior of various actors (such as other vehicles and pedestrians) to understand how the autonomous system interacts with them.

4. Scenario Testing: The system should support the configuration and testing of different driving scenarios, traffic jams, and at various speeds.

5. Data Integration: The system should be capable of integrating real-world data, such as actual traffic patterns and accident data, to enhance the accuracy of the simulations.

Non-Functional Requirements

1. Scalability: The system must be able to handle increasing simulation complexity and size, allowing the addition of more elements, such as more actors or larger geographical areas, without performance degradation.

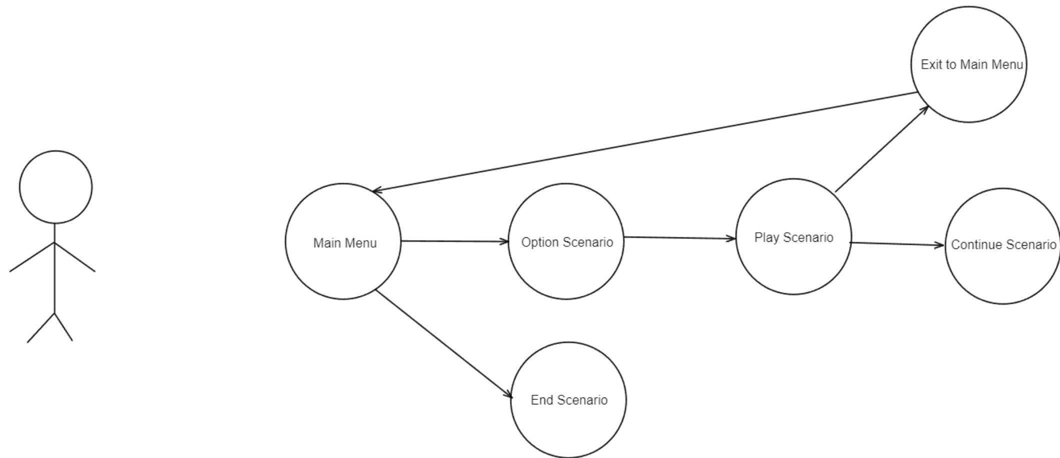
2. Reliability: The system must be highly reliable to ensure that simulations consistently run as expected without failures, especially when testing safety-critical systems.

3. Usability: The system should have user-friendly interfaces for setting up simulations, running them, and interpreting results without requiring deep technical knowledge of the underlying algorithms.

4. Performance: The system should be efficient in simulation execution, with minimal latency to mimic the real-time responses of autonomous systems in live environments.

Use Case Diagram

To outline the simulation's flow and possible user actions, a Use Case Diagram was created. The diagram depicts each scene and option that the user will encounter while playing the simulation. Starting from the main menu, the user can choose the network they wish to simulate, quit the simulation, or return to the main menu.



Conclusion

After conducting research on how other companies utilize simulation tools, it has become apparent what will make a simulation useful for traffic developers and council members. This includes features such as environmental factors to create a more realistic simulation, scalability to handle an increase in demand and see how road infrastructure handles the increase, and the ability for others to create networks using the same project.

4. Design

Design

The traffic simulator was created by using Unity game development engine as the frontend and SUMO traffic Simulator as the backend. C# programming language was used within Visual Studio Code. This chapter explains the design process of the simulation and provides a basic understanding of how each software works.

Technologies

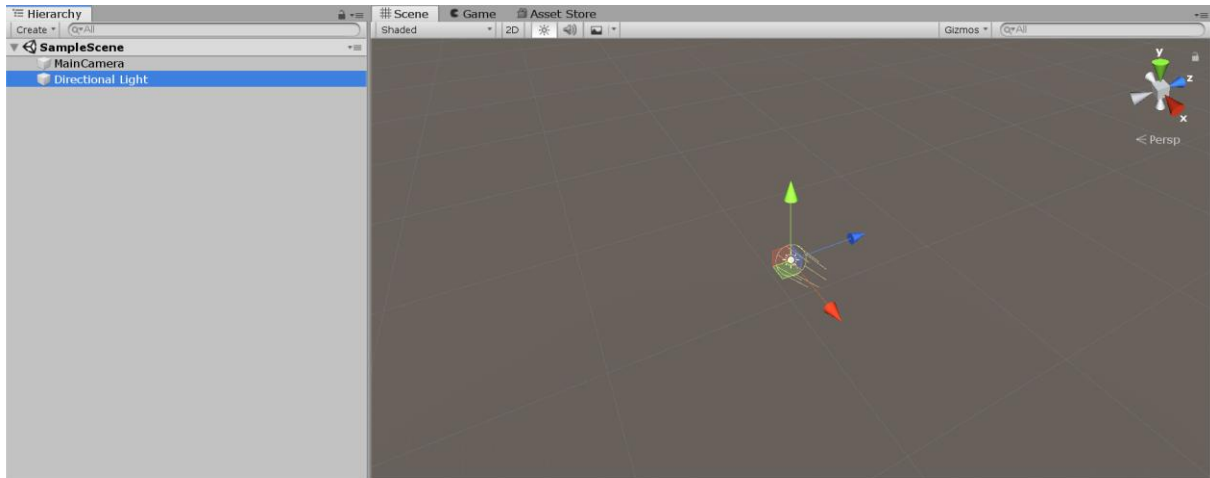
Visual Studio Code & C#

Although other code editors are compatible with Unity, Microsoft's Visual Studio Code was chosen for this project. Visual Studio Code is a simplified, streamlined version of Microsoft's Visual Studio, which makes it easy to focus exclusively on writing code. It is also highly customizable, and extensions can be added to aid the user in various areas, such as line auto-completion or formatting.

All scripts were written in C#, which is natively supported by Unity and can be easily integrated into Unity projects. C# is an object-oriented language derived from C and is quite similar in syntax to C++. It is a high-level language, meaning it is developer-friendly and simple to debug and maintain, compared to low-level languages such as Assembly code. High-level languages, such as C#, can also be run on any platform, whereas low-level languages are entirely machine-oriented.

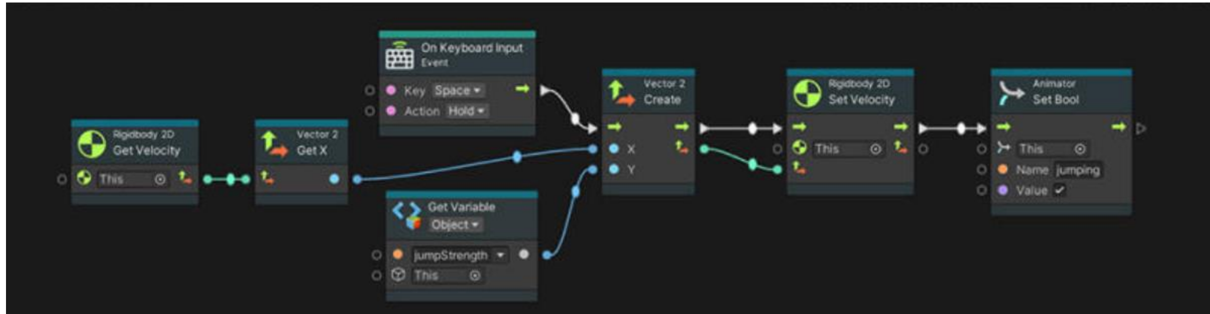
Unity

The Unity game development engine was used to create the frontend of the application, and it was programmed using the C# language. Specifically, version 2018.4.36f1 was used. Unity is a user-friendly game development engine that is available for free. It offers a graphical interface and uses C# code natively, although Java, C/C++, and Lua scripting can also be used. Although Unity can be used for 2D game development, it is mainly designed for 3D development. 3D game development can be easily carried out using the "scene view" window, where 3D assets can be created and placed in a three-dimensional game world. These assets can be easily adjusted as needed. This graphical interface also allows users to see real-time changes made to their code or track errors that arise.



Unity has a programming feature called Visual Scripting, which allows users to create games visually by dragging and dropping gameplay functions without writing any code. This makes game development more accessible, especially for beginners. However,

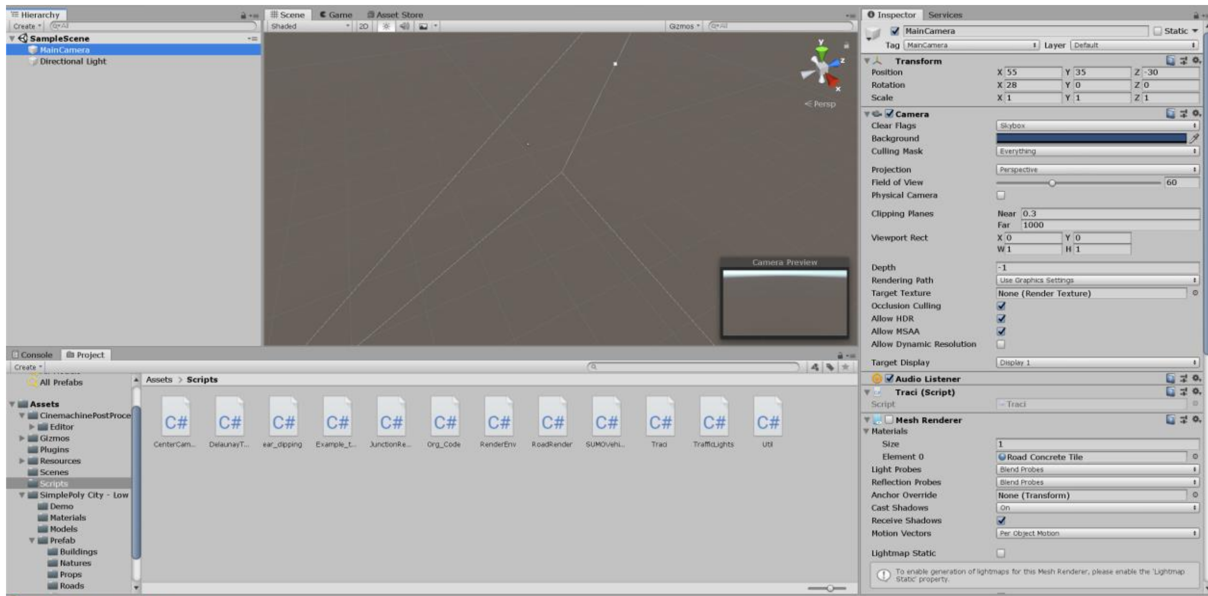
Visual Scripting was not used in the development of this project. Instead, all scripting was done using Visual Studio Code.



Furthermore, Unity is compatible with a wide range of operating systems and hardware. It can be used to develop games and applications for Windows, MacOS, and Linux, as well as for mobile platforms such as iOS and Android. It also supports virtual reality (VR) development for platforms like Oculus Quest, HTC Vive, and Steam VR.

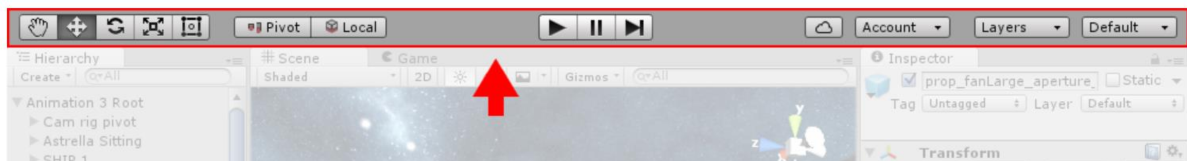
Unity Hierarchy and Inspector

The Hierarchy panel, located on the left side, displays the assets that are currently in the scene. You can perform various actions from here, such as deleting them, rearranging their order, and selecting them to view in the Inspector panel on the right side. The Inspector panel shows the properties of the selected game object, including its name, tags, layer, position, and size. You can make changes to these values through the Inspector panel.



The Toolbar

The Toolbar consists of seven basic controls. Each relate to different parts of the Editor.

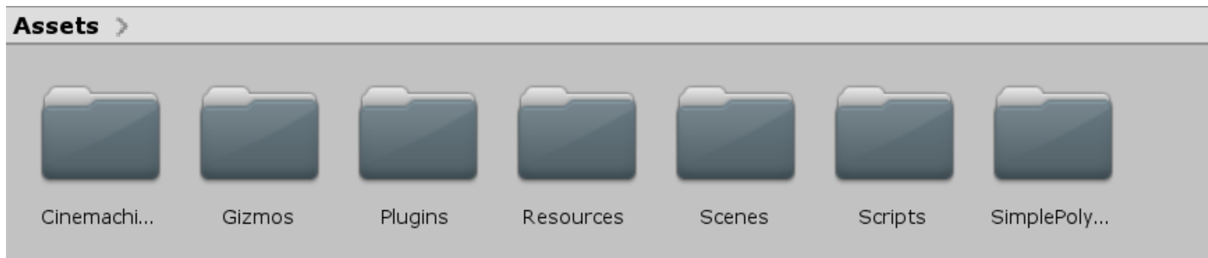


- Transform Tools – used with the Scene View
- Transform Gizmo Toggles – affect the Scene View display
- Play/Pause/Step Buttons – used with the Game View
- Cloud Button - opens the Unity Services Window.
- Account Drop-down - used to access your Unity Account.

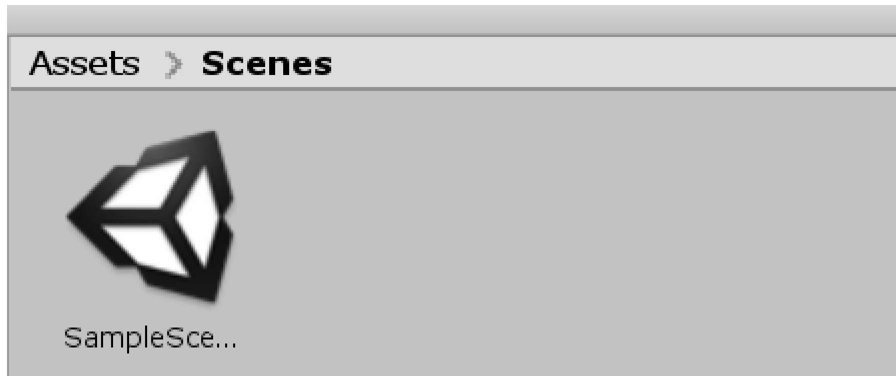
- Layers Drop-down – controls which objects are displayed in Scene View
- Layout Drop-down – controls arrangement of all Views

The Assets folder

The Assets folder is the primary folder for Unity projects, containing all the necessary materials like scenes, models, prefabs, plugins, textures, materials, and scripts. Users can also create subfolders to organize assets for easy location and selection. Whenever an asset pack is imported from the Unity Asset Store, a new folder is automatically created inside the project, containing the new assets.

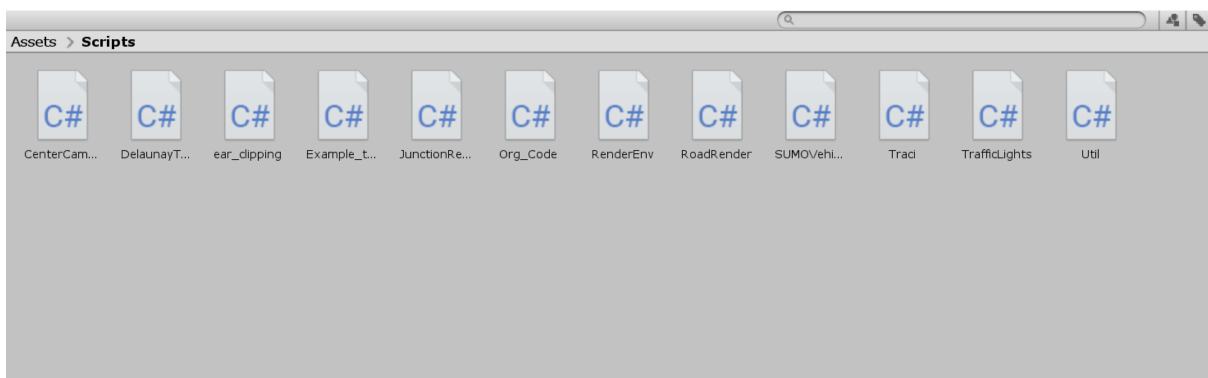


The Scenes folder in Unity is a crucial feature that should be used to organize and store all scene files. These scenes are containers that enable you to set up environments, gameplay areas, menus, and other interactive elements of your game or application. Each scene file represents a different level or screen of the game.



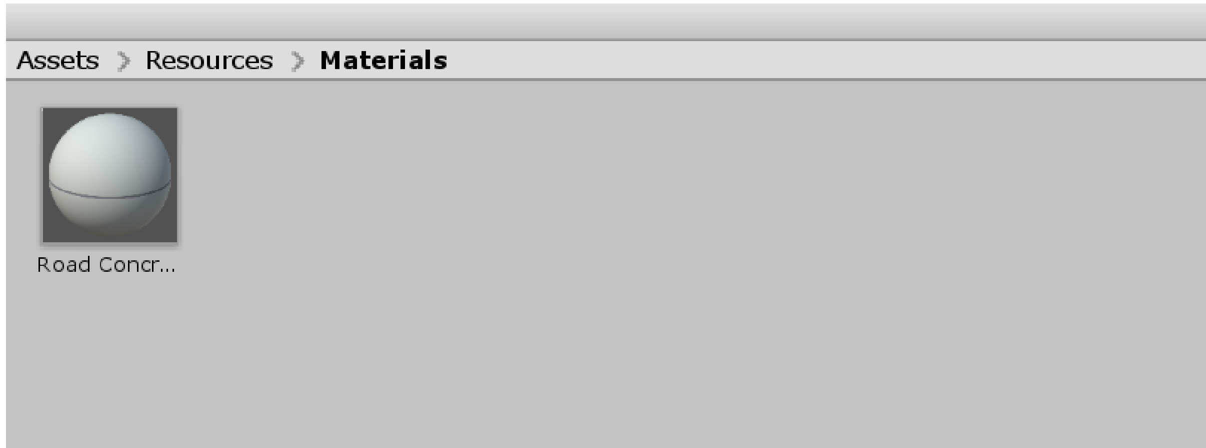
The Prefabs folder is another essential folder in Unity that is responsible for housing "Prefabs." These are pre-configured, reusable game objects or asset collections that can be instantiated multiple times throughout a project.

The Scripts folder is crucial for storing all the script files (CS files if using C#) that control the behavior of game objects and the game environment. Scripts in Unity are primarily written in C# and are used to implement game logic, define object behaviours, handle user input, manage scenes, and interact with other game components.

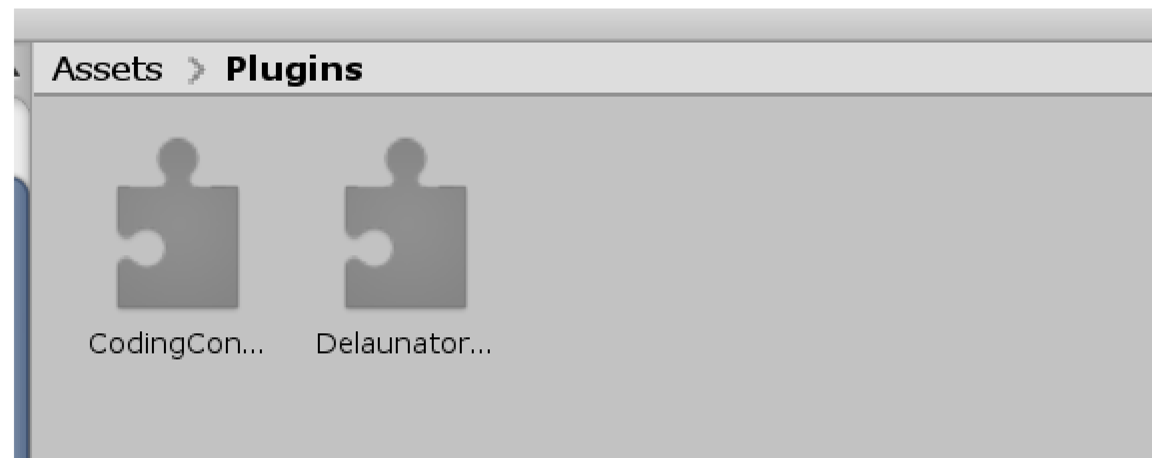


The Materials folder, on the other hand, is conventionally used to store material assets that define the appearance of surfaces in 3D environments. Materials in Unity are used to

specify how objects reflect light, display textures, and overall look within the scene. The Materials folder is essential for defining and managing the visual aesthetics of the Unity project and plays a crucial role in the look and feel of the game or application, allowing for customization and detailed control over surface appearances.



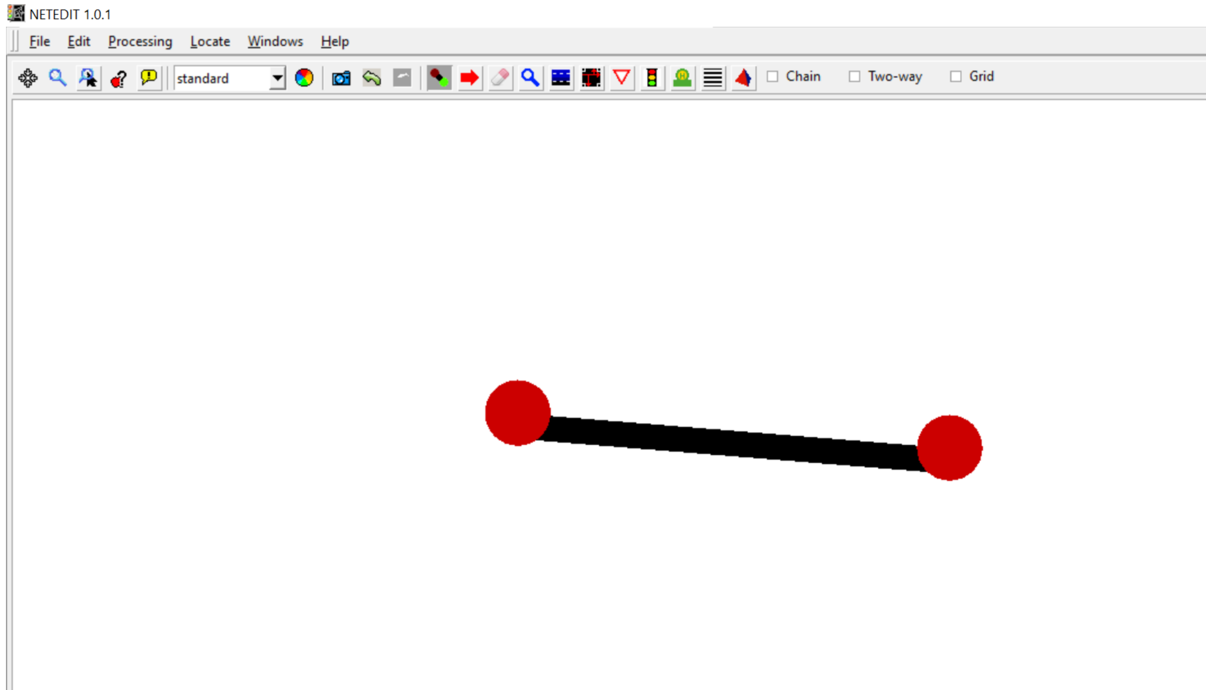
Lastly, the Plugins folder in Unity is used to store native plugins and manage assemblies that extend the functionality of the project. Code in the Plugins folder is compiled before the scripts in the rest of the project, which allows you to call these functionalities from the regular C# scripts within Unity. This setup helps in creating more modular and maintainable code.



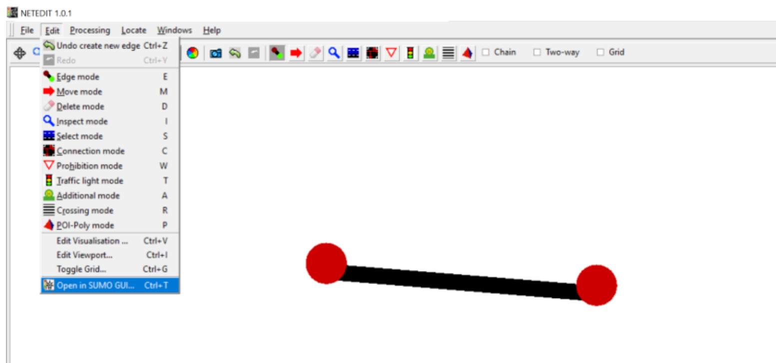
SUMO

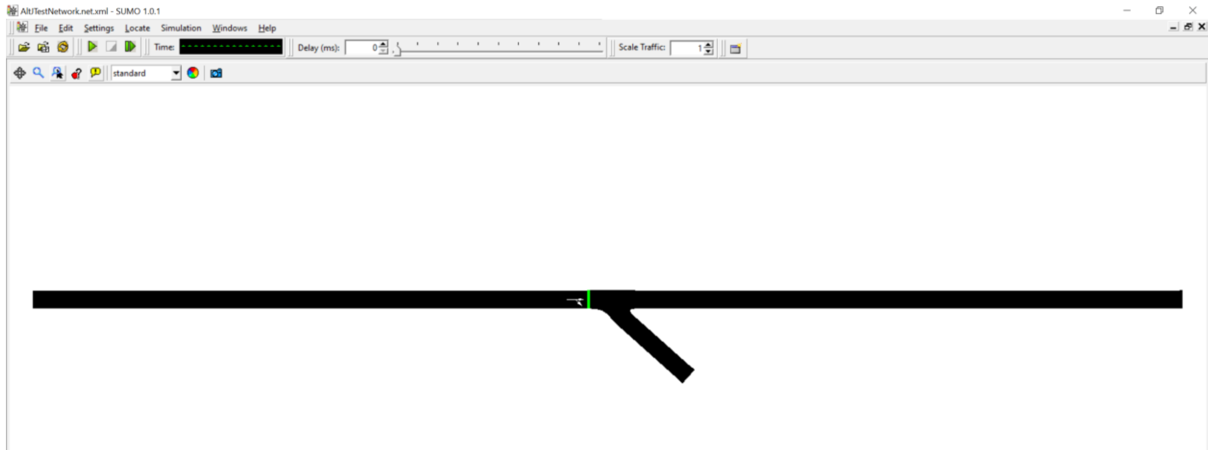
SUMO (Simulation of Urban Mobility) is a road traffic simulation package that is open-source, highly portable, microscopic, and continuous. It is designed to be able to handle large road networks. One of the tools included in the SUMO suite is Netedit, which is essential in creating and modifying network files used in simulations.

Netedit is a software tool that enables users to create and modify traffic networks using a graphical interface. It is user-friendly, allowing easy manipulation of elements such as roads, junctions, traffic lights, signs, and other objects. Netedit plays a critical role in configuring and updating the road networks that serve as the foundation for traffic simulations in SUMO.



Netedit is a tool that provides users with a Graphical User Interface (GUI) to visually create and edit traffic networks. It supports various network components like roads, junctions, lanes, and traffic control devices. Users can either create a new network from scratch or import and modify an existing one. Networks created or modified in Netedit can be directly used in SUMO for simulation purposes.





Apart from physical infrastructure, Netedit also allows for the management and editing of demand-related elements like vehicle routes, public transport lines, and traffic assignment. Users can manage different layers of the network, such as road signs, traffic lights, or transport routes, making complex edits more organized and manageable.

Netedit comes equipped with validation tools that check the network for errors, such as connectivity issues, undefined attributes, or conflicts in traffic flow logic, ensuring the network's operational feasibility before simulation. It also offers robust undo/redo capabilities that allow users to experiment and revert changes effortlessly, which is crucial in a design tool.

```
Running start /B "" "C:\Program Files (x86)\Eclipse\Sumo\/bin/sumo-gui" -n .
```

Tool Bar

SUMO's Netedit toolbar is designed to make it easy to create and manipulate different network elements. The toolbar includes the following tools:

1. Selection Tool: Usually represented by a cursor or arrow, this tool enables the selection of network elements such as nodes, edges, and lanes, for viewing or editing properties.

2. Move Tool: Often depicted as a hand or four-directional arrow, this tool allows you to move selected elements or navigate the view within the workspace.

3. Zoom Tool: Represented by a magnifying glass, sometimes with a plus or minus sign, this tool is used to zoom in or out of the network view to get a closer look or a broader perspective.

4. Create Edge Tool: Depicted as a line or road, this tool enables the creation of new edges (roads) by clicking and dragging between nodes (intersections).

5. Create Node Tool: Often shown as a dot or junction symbol, this tool is used for creating new nodes (junctions) within the network.

6. Create Traffic Light Tool: Represented by a traffic light icon, this tool enables the addition of traffic lights at junctions, which is crucial for controlling traffic flow.

7. Inspector Tool: Represented by a magnifying glass or an 'i' icon, this tool provides detailed information about selected network elements, including attributes and settings.

8. Delete Tool: Typically a trash can or an X symbol, this tool removes selected elements from the network.

9. Undo/Redo Tools: Represented by arrows pointing left for undo and right for redo, these tools allow you to revert or reapply changes made during the editing process.

10. Lane Addition/Modification Tool: Represented by lanes or a road branching, this tool facilitates the addition or alteration of lanes on existing roads.

11. Connection Tool: Often shown as a plug or link, this tool helps define how lanes connect across nodes, which is crucial for turn definitions and lane transitions.

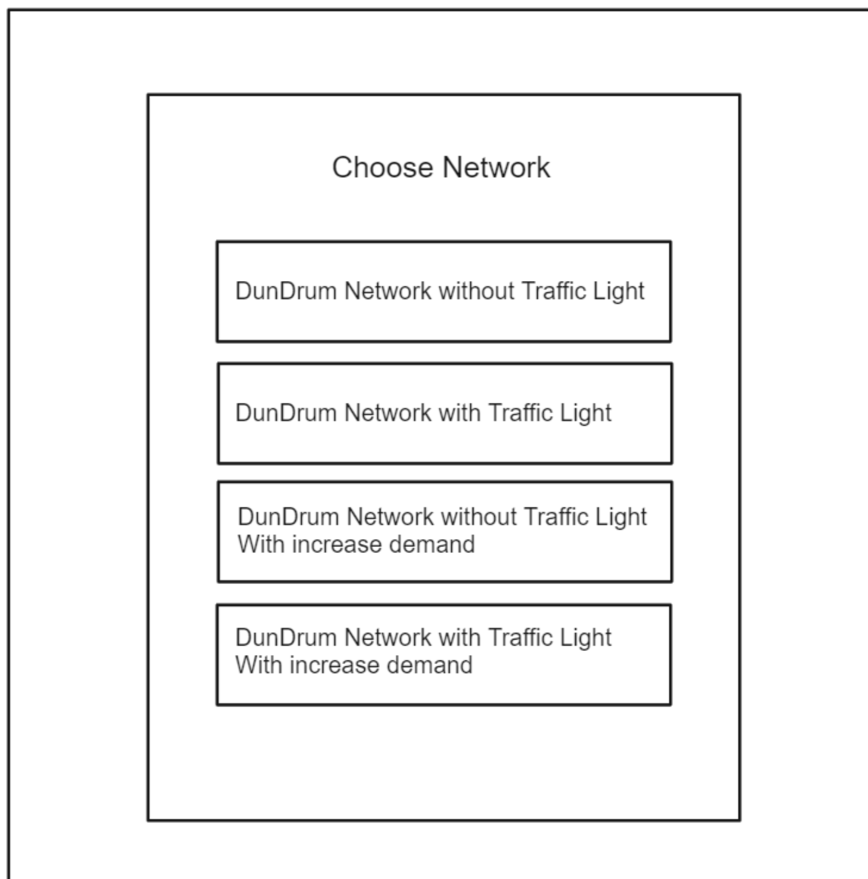
12. Traffic Demand Tool: Typically a car or flow chart, this tool is used to define and manage traffic flows, routes, and vehicle types, integrating transport demand aspects into the network.

13. Save and Load Tools: Represented by a floppy disk for save and a folder for load, these tools are essential for saving progress and loading existing projects or configurations.

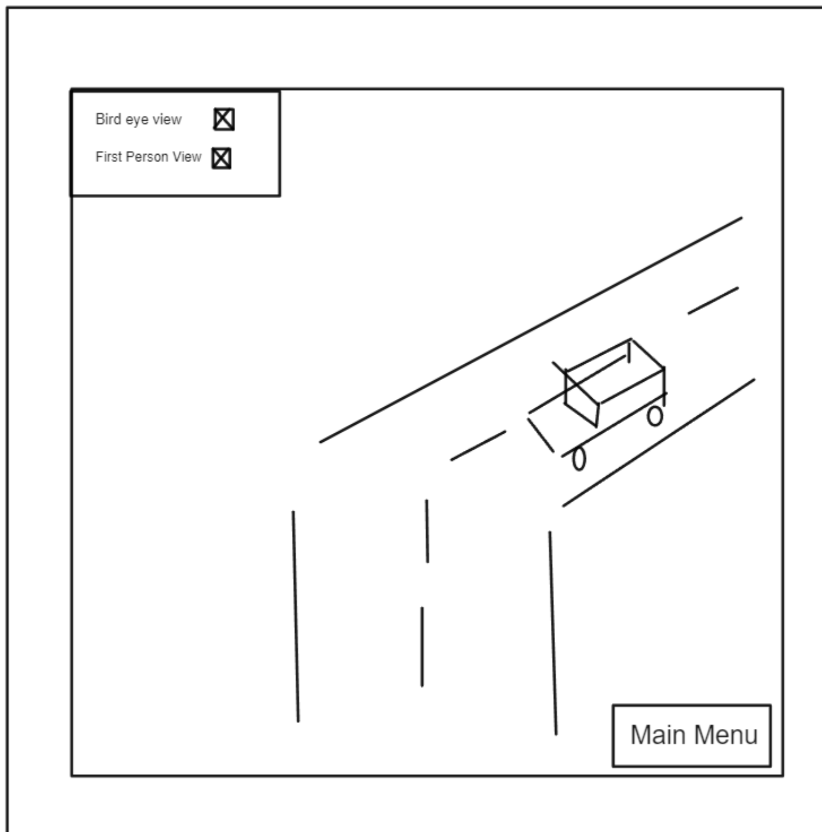
14. Simulation Start/Stop Tools: Represented by a play button for start and a square for stop, these tools allow you to run and stop simulations directly from Netedit to test and analyze the network behavior under traffic conditions.

UX Design

Wireframes were created to conceptualize the main screens of the simulation. A main menu was designed with the purpose of early going from one network to another.



The main is where the user views the simulation in action. The idea is to have the option for the user to view from the inside of the car so that they can be more enraptured in the simulation.



Implementation (or Construction)

Introduction

The implementation chapter of this document will provide detailed instructions on developing features within this project. Additionally, the thinking process behind the implementation. Including research efforts on feature methodology, highlighting the strengths and restrictions to keep in mind for each feature. The idea behind including a detailed implementation chapter is to build a foundation of understanding, making it possible for people wishing to attempt to develop a similar product and understand the reasoning behind each feature. This section of the document makes it easier to develop the project further and develop better and more efficient implementation methods.

Connection between SUMO and Unity

The initial task was to find a methodology for sending information to Unity for the SUMO simulation. The SUMO documentation talks about a way of retrieving data from the simulated object in the network and manipulating their behaviour while the simulation is running with the Traffic Control Interface, a.k.a. Traci (https://sumo.dlr.de/docs/TraCI.html). The purpose of using Traci is to set up a way of communicating and interacting with SUMO and external applications like Unity. Traci acts like a bridge, allowing Unity to retrieve data from the simulation in real time and manipulate the simulated objects' behaviour while the simulation runs.

The Traci works as a gateway to interact with the network with a real-time read traffic simulation. It employs a client/server architecture based on Transmission Control

Protocol, a.k.a. TCP, to connect with SUMO, one of the main protocols in the Internet Protocol Suite, often called TCP/IP. As mentioned in the previous chapter, TCP provides a dependable means of transmitting data over networks, making it suitable for applications that require reliable and ordered delivery.

However, SUMOs Traci method uses Python. The documentation also talks about using libsumo, which is supposed to perform better but uses C++. Therefore, it would be possible to implement this project using the Unreal game engine. Therefore, after conducting further research, there was a similar project about using CodingConected.Traci (https://github.com/CodingConnected/CodingConnected.Traci) has two implementations of the Traci protocol. The first offers an implementation of Traci in C#/.NET. To download the repository and covert into a plugin for a Unity 2018 project:

Clone the directory through the command line or download the '.zip' folder. In the directory where the repository was downloaded. Either throw the terminal or open it in a Code Editor. Inside the repository directory, find the .csproj file. This file contains the configuration and settings for the C# project. For this plugin, go to where the .csproj file is in the terminal and run 'dotnet build.' Then, there should be a .dll file in the output directory specified in the project settings. Compiling the C# code into a .dll file compiles it into a .dll file, which Unity can then use as a plugin. Copy the generated DLL file from the output directory. Open the Unity project and navigate to the Assets folder within the Unity Editor. Paste the copied DLL file into this folder. Unity will automatically detect

and import the DLL into the project. The DLL file appears in the Unity Editor's Project window.

Set up Unity and SUMO connection.

The Traci script oversees setting up a connection between SUMO using Traci, stepping the simulation to synchronize it with Unity, and closing the connection when the application quits. Additionally, it is in its script so that other scripts in the project can get the Client information. This way, every script does not require a new client connection, which would cause issues.

These directives import namespaces that contain classes and functionalities required for the script. This CodingConnected.TraCI.NET refers to the imported plugin. This directory provides classes for Traci's communication.

```
using System;
using System.Collections.Generic;
using CodingConnected.TraCI.NET;
using CodingConnected.TraCI.NET.Types;
using UnityEngine;
```

The Awake method within the Traci class is the first script to run when the user hits 'Play' in Unity. The 'Try' block creates a new instance of the Traci Client, which connects to SUMO simulating running locally on port '8813'. After that, the simulation will be synchronized using the ‘Client.Controle.SimStep()’ method with a fixed timestep of 0.02f.

```
4 references
public TraCIClient client;

0 references
void Awake()
{
    try
    {
        client = new TraCIClient();
        client.Connect("localhost", 8813); // Connects to SUMO simulation
        print("Connected to SUMO");
        client.Control.SimStep(); // Steps the simulation by one step
        Time.fixedDeltaTime = 0.02f;
    }
    catch (Exception e)
    {
        Debug.LogError("Error connecting to SUMO: " + e.Message);
    }
}
```

The connection must be synchronized to ensure that objects are rendered at the right time. For example, vehicles render when they are about to enter the simulation, making the initial rendering less taxing on the machine.

When SUMO and Unity are systematically synchronized, Unity can effectively avoid missing the initiation of a new vehicle. However, a delay could result in the removal on SUMO's side, which Unity has missed. It would result in Unity being unable to locate the vehicle it is being asked to delete, potentially causing significant issues in the integration process.

At the end of the Awake method, there's a 'Catch' block. This block is designed to throw an error if any part of the preceding 'Try' block fails. The error is then passed into the

console, providing valuable information for troubleshooting, and ensuring the smooth operation of the integration.

The last method in the Traci class is the OnApplicationQuite(). This method runs when the user ends the simulation or closes the application. Ensure the Traci client connection is properly closed when the scene ends.

```
0 references
private void OnApplicationQuit()
{
    client.Control.Close(); // Terminates the connection upon ending of the scene
    print("Connection closed");
}
```

This script is the foundation for further interactions between Unity and SUMO.

Setting up the Vehicles

The SUMOVehicle script handles the visualization and management of vehicles in the SUMO Simulation within Unity. Initially, this repository was used as a blueprint for creating and interacting with objects in SUMO and translating them into Unity (https://github.com/DarraghMac97/Real-time-Traffic-Simulation-with-3D-Visualisation/tree/master/Sumo%20Unity/Assets/Scenes).

Within the SUMOVehicle class, these variables are declared at the class level to reference the Traci Script, the Traci Client, the vehicle GameObject prefab, and a list to store instantiated vehicle GameObjects.

```
3 references
public Traci traci;
13 references
private TraCIClient client;
1 reference
public GameObject Vehicle_Car_color02;
10 references
public List<GameObject> carlist;
```

Within the start method, when the script is being initialized, it gets a reference to the Traci script attached to the same GameObject. Thereafter, it checks if the Traci script reference is valid and retrieves the Traci client from the Traci script. After that, it checks if the Client is valid.

```
void Start()
{
    // Get reference to the Traci script attached to the same GameObject
    traci = GetComponent<Traci>();

    // Check if Traci script reference is valid
    if (traci == null)
    {
        Debug.LogError("Traci script not found!");
        return;
    }
    // Get TraCI client from Traci script
    client = traci.client;

    // Check if TraCI client is valid
    if (client == null)
    {
        Debug.LogError("TraCIClient instance not found in Traci script!");
        return;
    }
}
```

The updated method is called each 0.02f, which is essential when getting the IDs of new and old vehicles and updating the vehicle's positioning. Within the 'Try' block, two critical operations are taking place to interact with the SUMO simulation via the Traci Plugin.

```
// Get the list of all vehicle IDs in the simulation
var vehicleidlist = client.Simulation.GetDepartedIDList("0").Content;
print("Vehicles entered: " + vehicleidlist.Count);
// Get the list of vehicles that have left the simulation
var vehiclesleft = client.Simulation.GetArrivedIDList("0").Content;
```

The first line retrieves a list of all vehicle IDs that have started their journey at the current simulation time step, emphasizing again the importance of synchronizing SUMO and Unity. This line provides real-time feedback about the number of vehicles that have started their journey.

Then, the 'vehiclesleft' variable retrieves the list of all vehicles that have arrived at the destination at the current simulation step.

The following section removes vehicles from the Unity scene once they have completed their journey in the SUMO simulation. This loop iterates through each vehicle ID in the list of vehicles stored in the 'vehiclesleft' variable.

```
foreach (var vehicleid in vehiclesleft)
{
    // Find the vehicle GameObject with the same name as the vehicle ID
    var vehicle = GameObject.Find(vehicleid);

    // Check if the vehicle GameObject exists
    if (vehicle != null)
    {
        try
        {
            // Destroy the vehicle GameObject
            GameObject.Destroy(vehicle);
            // Remove the vehicle ID from the list of vehicle IDs
            carlist.Remove(vehicle);
        }
        catch (Exception e)
        {
            Debug.LogError("Error: " + e.Message);
        }
    }
    else
    {
        Debug.LogWarning("Vehicle GameObject with ID " + vehicleid + " not found.");
    }
}
```

After that, if the vehicle GameObject was found in the scene, 'Try' to locate the vehicle with the ID of x, destroy the GameObject with that ID, and remove it from the scene. Additionally, try to remove this vehicle from the 'carlist', which will be described later.

In the 'else' statement, send the error message to the console if the vehicle is not found.

The block of code cleans up the Unity scene and ensures that the visual representation in Unity matches the current state of the SUMO simulation. It prevents clutter and potential performance issues due to unnecessary GameObjects, which is why this code block is written before populating and updating the vehicles.

The next section of the script handles creating and managing vehicle GameObjects in Unity based on vehicle data retrieved from the SUMO simulation.

```
foreach (var vehicleid in vehicleidlist)
{

    var shape = client.Vehicle.GetPosition(vehicleid).Content;
    var angle = client.Vehicle.GetAngle(vehicleid).Content;
    var length = client.Vehicle.GetLength(vehicleid).Content;
    var width = client.Vehicle.GetWidth(vehicleid).Content;
    var color = client.Vehicle.GetColor(vehicleid).Content;

    // Instantiate a new vehicle GameObject
    GameObject newcar = GameObject.Instantiate(Vehicle_Car_color02);
    newcar.name = vehicleid; // Set object name to vehicle ID

    // Set position, rotation, scale, and color
    newcar.transform.position = new Vector3((float)shape.X, 1.33f, (float)shape.Y);
    newcar.transform.rotation = Quaternion.Euler(0, (float)angle, 0);
    //newcar.transform.localScale = new Vector3((float)length, 1, (float)width);
    newcar.GetComponent<Renderer>().material.color = new UnityEngine.Color((float)color.R / 255, (float)color.G / 255, (float)color.B / 255);

    carlist.Add(newcar);
    print("Vehicle ID: " + vehicleid + " Position: " + shape.X + ", " + shape.Y + " Angle: " + angle);
}
```

Initially, the loop iterates through the list of vehicle IDs present in the 'vehiclelist' List, which subsequently retrieves various attributes of each vehicle through the Traci Client, such as its position, angle of facing, length, width, and colour.

```
foreach (var vehicleid in vehicleidlist)
{

    var shape = client.Vehicle.GetPosition(vehicleid).Content;
    var angle = client.Vehicle.GetAngle(vehicleid).Content;
    var length = client.Vehicle.GetLength(vehicleid).Content;
    var width = client.Vehicle.GetWidth(vehicleid).Content;
    var color = client.Vehicle.GetColor(vehicleid).Content;
```

Next, create a new GameObject by cloning the pre-fab vehicle named GameObject('Vehicle_Car_color02'). The new GameObject's name is set to the 'vehicleid', which helps track each vehicle's lifecycle.

```
// Instantiate a new vehicle GameObject
GameObject newcar = GameObject.Instantiate(Vehicle_Car_color02);
newcar.name = vehicleid; // Set object name to vehicle ID
```

To set the position of the new vehicle, use the 'tranfrom.position' for the new vehicle and adjust the coordinates to the Unity structure. SUMO stores the vehicle coordinates in a 2D structure. Therefore, the X position is the same as that in SUMO for setting the coordinates for Unity. The Y position is how far from the ground the vehicles will render, which, now, is a hard-coded value. Lastly, the z coordinates are the y coordinates in SUMO.

```
// Set position, rotation, scale, and color
newcar.transform.position = new Vector3((float)shape.X, 1.33f, (float)shape.Y);
newcar.transform.rotation = Quaternion.Euler(0, (float)angle, 0);
//newcar.transform.localScale = new Vector3((float)length, 1, (float)width);
newcar.GetComponent<Renderer>().material.color = new UnityEngine.Color((float)color.R / 255, (float)color.G / 255, (float)color.B / 255);
```

'Transfrom.rotation' sets the vehicle orientation based on the angle from SUMO. Lastly, to get the vehicle's color, get the color from SUMO and convert it into RGB values for Unity's color system. The final step is to add the vehicle GameObject to the 'carlist' List, which tracks all vehicle GameObjects currently on the scene.

```
carlist.Add(newcar);
print("Vehicle ID: " + vehicleid + " Position: " + shape.X + ", " + shape.Y + " Angle: " + angle);
```

The last loop updates the position and orientation of each vehicle GameObject in the current scene based on their current state in the sumo simulation.

```
for(int i = 0; i < carlist.Count; i++)
{
    print("Carlist: " + carlist[i]);
    var carposition = client.Vehicle.GetPosition(carlist[i].name).Content;
    var carangle = client.Vehicle.GetAngle(carlist[i].name).Content;
    carlist[i].transform.position = new Vector3((float)carposition.X, 1.33f, (float)carposition.Y);
    carlist[i].transform.rotation = Quaternion.Euler(0, (float)carangle, 0);
    print("Car position: " + client.Vehicle.GetPosition(carlist[i].name).Content);
    print("Car angle: " + client.Vehicle.GetAngle(carlist[i].name).Content);

}
```

The Loop itterates over each vehicle located in the 'carlist' for each vehicle GameObject. The script gets the latest position, 'carposition,' and orientation, 'carangle,' from SUMO. Then, update the related GameObject's position in the Unity scene using the X and Y coordinates retrieved from SUMO. Also, update the GameObject's rotation to reflect the current correlation.

```
for(int i = 0; i < carlist.Count; i++)
{
    print("Carlist: " + carlist[i]);
    var carposition = client.Vehicle.GetPosition(carlist[i].name).Content;
    var carangle = client.Vehicle.GetAngle(carlist[i].name).Content;
```

```
carlist[i].transform.position = new Vector3((float)carposition.X, 1.33f, (float)carposition.Y);
carlist[i].transform.rotation = Quaternion.Euler(0, (float)carangle, 0);
```

Initially, the Traci script and the Vehicle script were in the script. However, the scripts were separated to make them more organized, which helped connect the Traci script to other scripts later.

Traffic Lights

The TrafficLights script manages, initializes, and updates the traffic lights within the Unity scene to reflect their real-time status in the simulation. This script has gone through multiple iterations to find the correct rotation of the Traffic Lights. Firstly, in SUMO, each Traffic Light is associated with a Junction. The first implementation was done by using the thought process of getting the initial position of the Traffic Light and, from there, getting all the points of the Junction and calculating the center point of the Junction. From there, find the angle between the two vectors and set the angle to the negative value of this calculation. However, the initial problem with this implementation was that the angle was calculated from the traffic light position to the center of the connecting Junction.

Calculating the angle of the Traffic Light results in the angle facing away from the lane instead of facing the lane directly. To assist with visualizing, a graph script will be used to show the traffic light and junction point using Python in Google Colab.

```python
import numpy as np
import cv2
import math
import plotly.graph_objects as go

# Define test points
test_points = np.array([[103.99, 50.00], [103.99, 46.80], [101.77, 46.35], [100.99, 45.80],[100.43, 45.02], [100.10,44.02], [99.98,42.79], [96.78,42.80], [96.34,45.02], [95.79,45.80], [95.01,46.36], [94.01,46.69], [92.79,46.80], [92.79,50.00]])

test_points = test_points.astype(np.float32)
# Find minimum enclosing circle
center, radius = cv2.minEnclosingCircle(test_points)

# Traffic light position
TLx = 100.00
TLy = 50.00

# Calculate the edge vector
edge_vector_x = center[0] - TLx
edge_vector_y = (center[1]) - TLy

#edge_vector_x = TLx - center[0]
#edge_vector_y = TLy - center[1]

# Calculate the orientation angle in radians
orientation_angle = math.atan2(edge_vector_y, edge_vector_x)
orientation_angle_degrees = orientation_angle * (180 / math.pi)

# Plot the shape and traffic light orientation
fig = go.Figure()

# Add shape points
fig.add_trace(go.Scatter(x=test_points[:, 0], y=test_points[:, 1], mode='lines+markers', name='Shape'))

# Add traffic light position
fig.add_trace(go.Scatter(x=[TLx], y=[TLy], mode='markers', name='Traffic Light'))

# Add line representing the edge vector
edge_vector_end_x = TLx + edge_vector_x
edge_vector_end_y = TLy + edge_vector_y
fig.add_trace(go.Scatter(x=[TLx, edge_vector_end_x], y=[TLy, edge_vector_end_y], mode='lines', name='Edge Vector'))

# Add arrow representing traffic light orientation
arrow_length = 2 * radius  # Adjust arrow length as needed
arrow_end_x = TLx + arrow_length * math.cos(orientation_angle)
arrow_end_y = TLy + arrow_length * math.sin(orientation_angle)
fig.add_annotation(x=TLx, y=TLy, ax=arrow_end_x, ay=arrow_end_y, xref='x', yref='y', axref='x', ayref='y', arrowhead=2, arrowsize=1, arrowwidth=2, arrowcolor='blue')

# Set layout
fig.update_layout(title='Traffic Light Orientation',
                  xaxis_title='X Coordinate',
                  yaxis_title='Y Coordinate',
                  showlegend=True)

# Show plot
fig.show()
```

The output of this code proves the error behind this thinking process. Below, the junction point is displayed, and the starting point of the line going across the Junction represents the starting point of the traffic light and the arrow at the end represents the angle at which the Traffic Light is facing.

At this point, the concept evolved to include the 'assosiated lane' and the 'center points ' od the lane. These elements, crucial to the process, are obtained from both the lane and the junction. The 'TrafficLights' class has the same start method as the SUMOVehicle class. It gets a reference to the Traci script attached to the same GameObject.

The following method, PopulateTrafficLights', creates Traffic Light Objects in Unity based on the data retrieved from the Traci simulation.

```
public void PopulateTrafficLights()
{
    Debug.Log("Populating Traffic Lights");
    var TLList = client.TrafficLight.GetIdList().Content;
    Debug.Log("Traffic Light ID list: " + TLList.Count);

    foreach (var tlId in TLList)
    {
        HashSet<Vector2> uniqueJunctionPoints = new HashSet<Vector2>();
        HashSet<Vector2> uniqueLanePoints = new HashSet<Vector2>();
        var controlLane = client.TrafficLight.GetControlledLanes(tlId).Content;
        print("Traffic Light: " + tlId + " is controlling lane: " + controlLane[0]);
        var totalLaneShape = client.Lane.GetShape(controlLane[0]).Content;
        var laneWidth = client.Lane.GetWidth(controlLane[0]).Content;
        var laneWidthFloat = (float)laneWidth;
        var halfLaneWidth = laneWidthFloat / 2;
        var pos =  client.Junction.GetPosition(tlId).Content;
        Vector2 Position = new Vector2((float)pos.X, (float)pos.Y);
        var totalLanePoints = totalLaneShape.Points;
        var shape = client.Junction.GetShape(tlId).Content;
        var points = shape.Points;


        foreach (var point in points)
        {
            uniqueJunctionPoints.Add(new Vector2((float)point.X, (float)point.Y));
        }
        foreach (var point in totalLanePoints)
        {
            uniqueLanePoints.Add(new Vector2((float)point.X, (float)point.Y));
        }

       CreateTrafficLightObject(tlId,uniqueJunctionPoints, uniqueLanePoints, halfLaneWidth, Position);
    }
}
```

Firstly, it fetches the list of Traffic Light IDs in the simulation.

```
public void PopulateTrafficLights()
{
    Debug.Log("Populating Traffic Lights");
    var TLList = client.TrafficLight.GetIdList().Content;
    Debug.Log("Traffic Light ID list: " + TLList.Count);
```

After that, iterate through the Traffic Lights and fetch relevant data for each one. This includes information about the associated junction and the lane being controlled by the traffic light.

```
foreach (var tlId in TLList)
{
    HashSet<Vector2> uniqueJunctionPoints = new HashSet<Vector2>();
    HashSet<Vector2> uniqueLanePoints = new HashSet<Vector2>();
    var controlLane = client.TrafficLight.GetControlledLanes(tlId).Content;
    print("Traffic Light: " + tlId + " is controlling lane: " + controlLane[0]);
    var totalLaneShape = client.Lane.GetShape(controlLane[0]).Content;
    var laneWidth = client.Lane.GetWidth(controlLane[0]).Content;
    var laneWidthFloat = (float)laneWidth;
    var halfLaneWidth = laneWidthFloat / 2;
    var pos =  client.Junction.GetPosition(tlId).Content;
    Vector2 Position = new Vector2((float)pos.X, (float)pos.Y);
    var totalLanePoints = totalLaneShape.Points;
    var shape = client.Junction.GetShape(tlId).Content;
    var points = shape.Points;
```

In addition, two 'vector2' variables will be created to store unique points for easier calculation later in the script. Since the shape of the lanes may consist of the same points, a HashSet is used to ensure that each point stored in these sets is unique and the order of the pints don’t matter. The process of populating these sets is shown below. After that, the necessary values are passed to create the Traffic Light game objects, and the angle of the Traffic Lights is calculated using the CreateTrafficLightObject method.

```
foreach (var point in points)
{
    uniqueJunctionPoints.Add(new Vector2((float)point.X, (float)point.Y));
}
foreach (var point in totalLanePoints)
{
    uniqueLanePoints.Add(new Vector2((float)point.X, (float)point.Y));
}

CreateTrafficLightObject(tlId,uniqueJunctionPoints, uniqueLanePoints, halfLaneWidth, Position);
```

The following method, 'CreateTrafficLightObject,' assists in creating a GameObject representing the Traffic Lights.Furthermore, calculating the angle of each of the Traffic Lights. If the number of points defined for the Junction or the lane is less than 3, the shape's center is calculated by averaging the coordinates of the available points.

```
private void CreateTrafficLightObject(string id, HashSet<Vector2> junctionPoints, HashSet<Vector2> LanePoints, float halfLaneWidth, Vector2 Position)
{
    Vector2 laneCenter = new Vector2();
    Vector2 junctionCenter = new Vector2();

    if (junctionPoints.Count < 3)
    {
        junctionCenter = new Vector2((junctionPoints.ElementAt(0).x + junctionPoints.ElementAt(1).x) / 2,
                                     (junctionPoints.ElementAt(0).y + junctionPoints.ElementAt(1).y) / 2);
        Debug.Log("Junction has less than 3 points");
    }

    if (LanePoints.Count < 3)
    {
        laneCenter = new Vector2((LanePoints.ElementAt(0).x + LanePoints.ElementAt(1).x) / 2,
                                 (LanePoints.ElementAt(0).y + LanePoints.ElementAt(1).y) / 2);
        Debug.Log("Lane has less than 3 points");
```

If at least 3 points are defined for the Junction, the center is calculated by subtracting the minimum and maximum coordinates among the junction points and averaging them out. This will, hopefully, ensure that the center of the Junction and lane shapes are determined accurately.

```
} if (junctionPoints.Count >= 3)
{
    Vector2 minX = new Vector2();
    Vector2 maxX = new Vector2();
    Vector2 minY = new Vector2();
    Vector2 maxY = new Vector2();
    foreach (var point in junctionPoints)
    {
        if (point.x < minX.x)
        {
            minX = point;
        }
        if (point.x > maxX.x)
        {
            maxX = point;
        }
        if (point.y < minY.y)
        {
            minY = point;
        }
        if (point.y > maxY.y)
        {
            maxY = point;
        }
    }
    junctionCenter = new Vector2((minX.x + maxX.x) / 2, (minX.y + maxX.y) / 2);
}
```

The direction of the Traffic Light is calculated as the vector from the center of the lane to the center of the junction. This line will be perpendicular to the line connecting the centers. Then, the orientation angle is calculated based on the direction vector using the 'Mathf.Atan2' functions, which will return the angle in radians. Then, convert the angle into degrees to make it compatible with Unity.

```
var edge_vector_x = laneCenter.x - junctionCenter.x;
var edge_vector_y = laneCenter.y - junctionCenter.y;
// Calculate the direction vector from the junction center to the lane center
Vector2 directionVector = new Vector2(edge_vector_y, -edge_vector_x); // Perpendicular to the line connecting the centers
// Calculate the orientation angle based on the direction vector
var orientation = Mathf.Atan2(directionVector.y, directionVector.x) * Mathf.Rad2Deg;
```

Laslty, creating a new GameObject representing the traffic light is creatied using the Props_Traffic prefab. With the name bing the id of the traffic light. The position of the

gameObject is set based on the position retrived from the Traci Client + adding half of the lane width to place the traffic light on the edge of the road. Lastly, the orientation of the traffic light is set to the calculated orientation angle.

```
GameObject newTL = GameObject.Instantiate(Props_Traffic);
newTL.name = id;
newTL.transform.position = new Vector3(Position.x, 1.33f, (Position.y+halfLaneWidth));
newTL.transform.rotation = Quaternion.Euler(0, -orientation, 0);
print("Orientation: " + orientation);
```

The final part of the script, the update method, is responsible for updating the visualization of traffic Lights based on their current phase in the SUMO simulation.

The 'update' method systematically iterates through each traffic light in the simulation. For each traffic light, it retrieves the current phase and the render component of the traffic light object. It then proceeds to access all the child objects, which correspond to the various lights of the traffic light.

```
0 references
public void Update(){
    try
    {
        client.Control.SimStep(); // Steps the simulation
        var currentTime = client.Simulation.GetCurrentTime("simulation").Content; // Get current simulation t
        print("Current simulation time: " + currentTime);
        var TLList = client.TrafficLight.GetIdList().Content;
        foreach(var tlId in TLList){
            var Phase = client.TrafficLight.GetState(tlId).Content.ToLower();
            var TLRenderer = GameObject.Find(tlId).GetComponent<Renderer>();
            var child = GameObject.Find(tlId).transform.childCount;
```

Depending on the current phase of the traffic light, set the color of the child object to either green, red, yellow, or nothing.

```
var child = GameObject.Find(tlId).transform.childCount;
if(Phase == "g")
{
    for(int i = 0; i < child; i++){
        if(i != 2){
            var childColour = GameObject.Find(tlId).transform.GetChild(i);
            childColour.GetComponent<Renderer>().material.color = UnityEngine.Color.black;
        }
        else{
            var childColour = GameObject.Find(tlId).transform.GetChild(i);
            childColour.GetComponent<Renderer>().material.color = UnityEngine.Color.green;
        }
    }
}
else if(Phase == "r")
{
    for(int i = 0; i < child; i++){
        if(i != 0){
            var childColour = GameObject.Find(tlId).transform.GetChild(i);
            childColour.GetComponent<Renderer>().material.color = UnityEngine.Color.black;
        }
        else{
            var childColour = GameObject.Find(tlId).transform.GetChild(i);
            childColour.GetComponent<Renderer>().material.color = UnityEngine.Color.red;
        }
    }
}
else if(Phase == "y")
{
    for(int i = 0; i < child; i++){
        if(i != 1){
            var childColour = GameObject.Find(tlId).transform.GetChild(i);
            childColour.GetComponent<Renderer>().material.color = UnityEngine.Color.black;
        }
        else{
            var childColour = GameObject.Find(tlId).transform.GetChild(i);
            childColour.GetComponent<Renderer>().material.color = UnityEngine.Color.yellow;
        }
    }
}
else
{
    for(int i = 0; i < child; i++){
        var childColour = GameObject.Find(tlId).transform.GetChild(i);
        childColour.GetComponent<Renderer>().material.color = UnityEngine.Color.black;
    }
}
```

Render Lanes

This RoadRender script is attached to a GameObject in the Unity project. The script generates and renders road meshes based on lane data from the traffic simulation using TraCI.

The LaneData method in the RoadRender script populates lane data by retrieving information about lanes from the traffic simulation using TraCI and generating road meshes for visualization in the Unity environment.

Firstly, it retrieves a list of lane IDs from the TraCIClient instance using the GetIdList method. This method returns a TraCIResponse<List<string>> object containing the IDs of all lanes in the simulation.

```
public void LaneData()
{
    Debug.Log("Populating Lanes");
    var laneIds = client.Lane.GetIdList().Content;
    print("Lane ID list: " + laneIds.Count);

    // Dictionary to store lane IDs and their corresponding opposite lanes
    Dictionary<string, string> oppositeLanes = new Dictionary<string, string>();

    foreach (var laneId in laneIds)
    {
        var oppositeLane = GetOppositeLaneId(laneId);
        if (oppositeLane != null)
        {
            oppositeLanes[laneId] = oppositeLane;
        }
    }
```

It initializes a dictionary named oppositeLanes to store pairs of lane IDs and their corresponding opposite lane IDs. The opposite lane of a given lane ID is the lane on the

opposite side of the road. The reason behind combining parallel roads was the shape provided by the SUMO.net.xml file. Straight lanes would have a shape of 2 points. However, lanes with curves would have a longer list of points or could have 3 points. If the lane only had 3 points then the lane would render as a triangle instead of a proper lane shape.

```
dge id="-592063756#0" from="257927907" to="43296492" priority="6" type="highway.tertiary" shape="314.44,441.42 289.60,440.81 214.08,437.90">
  <lane id="-592063756#0_0" index="0" disallow="tram rail_urban rail rail_electric ship" speed="13.89" length="90.59" shape="307.21,442.84 289.55,442.41 216.68,439.60"/>
edge>
```

Then, itterate over each lane ID in the laneIds list and call the GetOppositeLaneId helper method to retrieve the ID of the opposite lane, if available. If an opposite lane is found, it adds the pair to the oppositeLanes dictionary. Unique Lane initializes a list named uniqueLanes to store unique pairs of lane IDs and their opposite lane IDs. This list will be used to process each lane and its opposite lane together later in the method.

```
// List to store unique lanes along with their opposite lanes
List<(string laneId, string oppositeLaneId)> uniqueLanes = new List<(string, string)>();

foreach (var laneId in laneIds)
{
    if (oppositeLanes.ContainsKey(laneId))
    {
        var oppositeLaneId = oppositeLanes[laneId];
        if (!uniqueLanes.Contains((laneId, oppositeLaneId)) && !uniqueLanes.Contains((oppositeLaneId, laneId)))
        {
            uniqueLanes.Add((laneId, oppositeLaneId));
        }
    }
    else
    {
        // Add lanes without opposite lanes
        uniqueLanes.Add((laneId, null));
    }
}
```

It itterates over each lane ID in the ‘laneIds’ list again and checks if the lane has an opposite lane. If an opposite lane is found, it checks if the pair of lane IDs is already present in the ‘uniqueLanes’ list. If not, it adds the pair to the list.

```
// Process each lane and its opposite lane together
foreach (var (laneId, oppositeLaneId) in uniqueLanes)
{
    if ((laneId.StartsWith("-"))|| (laneId.StartsWith(":")))
    {
        // Skip processing lanes with negative IDs
        continue;
    }
```

It iterates over each unique pair of lane IDs and their opposite lane IDs stored in the uniqueLanes list. For each pair, it checks if the lane ID starts with a negative sign ("-") or a colon (":"), indicating that it should be skipped. Since lanes containing (":") are internal lanes and ids with ("-") indicates that the lane is a part of the parallel road. This was done due to performance issues.

```
List<Position2D> lanePoints = new List<Position2D>();
float roadWidth = 0f;
float roadLength = 0f;
```

For each valid lane, it initializes a list named 'lanePoints' to store the points defining the lane shape. It also initializes variables to store the total road width and length of the lane.

It iterates over each lane ID in the 'laneIds' list and checks if the current lane ID contains a specific pattern that indicates it is the opposite lane of the given laneId. The opposite lane ID is formed by the original lane ID with a hyphen ("-").

```
// Process the opposite lane if available
if (oppositeLaneId != null)
{
    var oppositeLaneShape = client.Lane.GetShape(oppositeLaneId).Content;
    lanePoints.AddRange(oppositeLaneShape.Points);
    roadWidth += (float)client.Lane.GetWidth(oppositeLaneId).Content;
    roadLength += (float)client.Lane.GetLength(oppositeLaneId).Content;
}
```

If a lane ID containing the specified pattern is found, and it is not the same as the original laneId, the method returns this lane ID as the opposite lane ID. Otherwise, if no matching lane ID is found, the method returns null, indicating that no opposite lane was found for the given laneId.

In summary, the GetOppositeLaneId method searches through the list of all lane IDs in the traffic simulation system to find the opposite lane ID corresponding to a given lane ID. It does this by checking for lane IDs that contain a specific pattern indicating they are the opposite lanes. If a matching opposite lane ID is found, it is returned; otherwise, null is returned. This method is used in conjunction with the LaneData method to pair each lane with its opposite lane for further processing and visualization of road meshes in the Unity environment.

The GenerateRoadMesh method in the RoadRender script is designed to generate a 3D mesh for a road segment based on a list of 2D positions (Position2D points), the width, and the length of the road. It adapts its mesh generation strategy based on the complexity of the road shape, determined by the number of points defining the segment.

Based on the number of points, the method chooses one of three approaches to generate the mesh:

Fewer than 3 Points: If there are fewer than 3 points, a simple shape mesh is generated using GenerateSimpleShapeMesh. This function creates a basic road shape for straight lanes.

```
public Mesh GenerateRoadMesh(List<Position2D> points, float roadWidth, float roadLength, string laneId)
{
    print("Points count: " + points.Count);

    if (points.Count < 3)
    {
        print("Simple shape mesh generated");
        return GenerateSimpleShapeMesh(points, roadWidth, roadLength, laneId);
    }
```

If there are 3 points, the road is triangular. This could call a hypothetical method like GenerateTriangleMesh to create a triangle-shaped mesh, although this pathway is commented out in the code and returns null.

```
if (points.Count == 3)
{
    print("Triangle shape mesh generated");
    // return GenerateTriangleMesh(points, roadWidth, roadLength);
    return null;
}
```

For more complex shapes with more than 3 points, GenerateComplexShapeMesh is called. This function utilizes the ear clipping method to triangulate the polygon defined by the points, suitable for creating complex, irregularly shaped road segments.

```
if (points.Count > 3)
{
    print("LaneId: " + laneId + " Complex shape mesh generated");
    return GenerateComplexShapeMesh(points, roadWidth, roadLength, laneId);
}
```

The GenerateSimpleShapeMesh method constructs a mesh that represents the road segment. It does so by generating vertices along the road path and defining triangles (the basic units of a mesh in 3D graphics) to form the surface of the road.

The mothod accepts the points, which contains a list of `Position2D` objects that provide the x and y coordinates defining the road's path. Additionally, the road width, road length and lastly the lane id.

Initially, the method sets up a new Mesh object, which will be returned at the end of the method. Additionally, a List for storing vertices and triangle indices. The vertices are a X, Y and Z coordinates. Every corner of a 3D object is marked by a vertex. For example, a simple 3D triangle requires three vertices, one for each corner. The Y coordinates of the vertices is set to 'onGround', which has a hardcoded value in it. Which ensures that all vertices lie at the same vertical position, simulating the road lying flat on the terrain. The list of triangles consists of three vertices connected. It is the simplest 2D shape that can exist in 3D space. Triangles are used to construct the surface of 3D models. Because they

are flat and have only three sides. Graphics hardware is optimized to render triangles efficiently, which is why they are the fundamental unit of 3D models in graphics.

```
private Mesh GenerateSimpleShapeMesh(List<Position2D> points, float roadWidth, float roadLength, string laneId)
{
    print("Generating simple shape mesh");
    Mesh mesh = new Mesh();
    List<Vector3> roadVertices = new List<Vector3>();
    List<int> triangles = new List<int>();
    float onGround = 1.33f;
```

The direction vector for each segment of the road is calculated using consecutive points. This vector is normalized to ensure consistent operations. A perpendicular vector is calculated for each direction vector, ensuring it extends to either side of the path. This is crucial for generating the width of the road.

```
// Compute perpendicular vector
Vector2 direction = new Vector2((float)(points[1].X - points[0].X), (float)(points[1].Y - points[0].Y)).normalized;
Vector2 perpendicular = new Vector2(-direction.y, direction.x);
```

The direction vector for each segment of the road is calculated using consecutive points. This vector is normalized to ensure consistent operations A perpendicular vector is calculated for each direction vector, ensuring it extends to either side of the path. This is crucial for generating the width of the road.

```
// Points along the road segment
for (int j = 0; j < numPoints; j++)
{
    float t = (float)j / (numPoints - 1); // Interpolation parameter
    Vector2 interpolatedPoint = A + t * (B - A);
    Vector2 leftPoint = interpolatedPoint - (roadWidth / 2) * perpendicular;
    Vector2 rightPoint = interpolatedPoint + (roadWidth / 2) * perpendicular;

    roadVertices.Add(new Vector3(leftPoint.x, onGround, leftPoint.y));
    roadVertices.Add(new Vector3(rightPoint.x, onGround, rightPoint.y));
```

Each segment between consecutive points is divided into smaller parts based on the calculated segment length and the specified road width, determining the number of interpolation points 'numPoints'. For each segment, interpolation points are calculated along the line. At each of these ponts, two vertices are placed to the left and right perpendicularly by half of the road width. This forms the actual width of the road.

The triangle formation is created by defining the connection between vertices. When moving along each segment, a pair of triangles is created between each set of four vertices. This will create a rectangle across the road width. The vertices are indexed in a way that ensures the triangles are rendered correctly in Unity. This involves adding two triangles for each set of four vertices. One triangle connects the first three vertices, and the second triangle connects the last three.

```
if (j < numPoints - 1)
{
    int index = 2 * j;
    triangles.Add(index);
    triangles.Add(index + 1);
    triangles.Add(index + 2);

    triangles.Add(index + 1);
    triangles.Add(index + 3);
    triangles.Add(index + 2);
}
```

The final step is to assign the list of vertices and triangles to the Mesh object. Normals are recalculated to ensure that lighting effects render correctly on the road surface. This step is essential as it affects how light interacts with the surface of the mesh, influencing visual realism.

```
mesh.SetVertices(roadVertices);
mesh.triangles = triangles.ToArray();
mesh.RecalculateNormals();

return mesh;
```

The fully constructed mesh is returned to the GenerateRoadMesh method, which is then returned to the LaneData method. The mesh object and the lane ID is finally passed to the RenderMesh method.

The GenerateComplexShapeMesh method is designed to create a 3D mesh from a set of vertices that form a complex polygonal shape. This method is typically used when the road shape is defined by more than three points, allowing for the creation of irregular or curved road segments. In this context, it employs an ear clipping algorithm for triangulation.

The unique part of this method is to convert each 'Position2D' into a 'Vector2'. This is necessary for the ear clipping algorithm which operates in 2D. Utilizing the ear clipping method to triangulate the polygon. This involves finding 'ears' in the polygon, which are triangles that can be clipped off without cutting through the polygon and progressively reducing the polygon until it's fully triangulated. The 'ear_clipping.EarClipping(vertices2D)' function returns a list of tuples, where each tuple represents a triangle with indices referring to the vertices in 'vertices2D'.

```
List<Vector2> vertices2D = new List<Vector2>();

foreach (var point in points)
{
    vertices2D.Add(new Vector2(((float)point.X), (float)point.Y));
}

var tris = ear_clipping.EarClipping(vertices2D);
```

Then to create the 3D vertices from the Triangulation. Iterate over each triangle produced by the ear clipping. Convert each 2D vertex of these triangles into 3D by assigning a consistent Y coordinate.

```
foreach (var tri in tris)
{
    int indexOffset = roadVertices.Count;

    // Ensure counterclockwise order
    roadVertices.Add(new Vector3(tri.Item1.x, onGround, tri.Item1.y));
    roadVertices.Add(new Vector3(tri.Item2.x, onGround, tri.Item2.y)); /
    roadVertices.Add(new Vector3(tri.Item3.x, onGround, tri.Item3.y)); /

    // Adjust triangle indices accordingly
    triangles.Add(indexOffset);
    triangles.Add(indexOffset + 2); // Reversed order
    triangles.Add(indexOffset + 1); // Reversed order
}
```

Each set of three vertices are added to 'roadVertices' which forms a triangle. Since the vertices are added in sets of three, the indices can be assigned sequentially. Then assign the vertices and triangles to the mesh. Then, recalculate the normals for the mesh to ensure correct lighting and shading. The mesh is now complete and can be returned. This mesh can then be used with Unity's `MeshFilter` and `MeshRenderer` components to render the complex road shape in the scene.

```
mesh.SetVertices(roadVertices);
mesh.triangles = triangles.ToArray();
mesh.RecalculateNormals();

return mesh;
```

The `GenerateComplexShapeMesh` method is ideal for roads that include curves, intersections, or any irregular shapes that cannot be simply represented by straight lines or simple triangles. By using the ear clipping method for triangulation, this approach

allows for the creation of visually accurate and topologically correct road meshes based on real-world or simulation data.

The RenderMesh method in the RoadRender script is responsible for taking a generated mesh and rendering it in the Unity game environment by attaching it to a new GameObject. This method involves a few critical steps to ensure that the mesh is properly displayed, utilizing Unity's rendering system components such as MeshFilter and MeshRenderer.

The Mesh object that contains the vertices, triangles, and normals needed to render the shape of the road. A string identifier for the lane, which can be used for naming the GameObject and possibly for debugging purposes.

```
public void RenderMesh(Mesh mesh, string laneId)
{
    if (mesh != null && mesh.vertexCount > 0)
    {
        print("Rendering road mesh for Lane" + laneId);
        GameObject road = new GameObject("Road" + laneId);
        road.transform.position = Vector3.zero;
        road.transform.rotation = Quaternion.identity;
        road.transform.localScale = Vector3.one;
```

Rendering Steps starts with checking if the mesh is valid and has vertices to render. A new GameObject is created to hold the mesh. This GameObject acts as a container for the mesh and its associated rendering components. The GameObject is named using the lane ID to ensure it can be easily identified within the Unity Editor or while debugging.

The transformation properties of the GameObject (position, rotation, scale) are set to defaults. Typically, the position is set to zero (the origin of the coordinate system), the rotation is set to no rotation (Quaternion.identity), and the scale is set to one (original size).

```
    MeshFilter meshFilter = road.AddComponent<MeshFilter>();
    meshFilter.mesh = mesh;

    MeshRenderer meshRenderer = road.AddComponent<MeshRenderer>();
    meshRenderer.material = roadMaterial;
    print("Road mesh generated successfully for Lane: " + road.name);
}
```

The 'roadMaterial' is a previously defined material that will dictate the appearance of the road surface, including its texture, color, and reflectiveness.

The Ear Clipping Algorithm

The ear clipping algorithm for triangulating simple polygons is used to decompose a polygon into triangles, which is a common requirement in graphics programming for rendering shapes. The ear clipping algorythms was create thanks to the Youtuber Two-bit coding(https://www.youtube.com/watch?v=hTJFcHutls8&t=1976s&ab_channel=Two-BitCoding)

The IsConvex Method takes three Vector2 points a, b, and c which represent consecutive vertices of a polygon. Which determines if the triplet forms a convex angle at ‘b’ by checking the sign of the z-component of the cross product of vectors (a-b) and (c-b). If it's non-negative, the corner is convex. Is to ensure that a potential ear triangle doesn't fold inwards.

```
1 reference
private static bool IsConvex(Vector2 a, Vector2 b, Vector2 c)
{
    return Vector3.Cross(b - a, c - b).z >= 0;
}
```

The next IsPointInTriangle Method takes a point 'p' and three vertices 'a', 'b', and 'c' of a triangle to determine if point ‘p’ is inside the triangle formed by a, b, and c. This method checks if 'p' is on the same side of each edge as the opposite vertex. Used to check if any point lies inside a candidate's ear, which is necessary to validate an ear before it is clipped.

```
1 reference
private static bool IsPointInTriangle(Vector2 p, Vector2 a, Vector2 b, Vector2 c)
{
    Vector2 ab = b - a;
    Vector2 ap = p - a;
    Vector2 bc = c - b;
    Vector2 bp = p - b;
    Vector2 ca = a - c;
    Vector2 cp = p - c;

    float crossAbAp = Vector3.Cross(ab, ap).z;
    float crossBcBp = Vector3.Cross(bc, bp).z;
    float crossCaCp = Vector3.Cross(ca, cp).z;

    return crossAbAp >= 0 && crossBcBp >= 0 && crossCaCp >= 0;
}
```

The IsEar Method takes in three consecutive vertices 'a', 'b', and 'c' and the list of all vertices. First checks if the vertices form a convex corner using IsConvex. If they do, it then checks if no other vertices lie inside the triangle formed by these three vertices using IsPointInTriangle.

The purpose is to identifies if a triangle (ear) can be safely clipped off without cutting through the polygon.

```
private static bool IsEar(Vector2 a, Vector2 b, Vector2 c, List<Vector2> vertices)
{
    if (!IsConvex(a, b, c))
        return false;

    foreach (var v in vertices)
    {
        if (v != a && v != b && v != c)
        {
            if (IsPointInTriangle(v, a, b, c))
                return false;
        }
    }
    return true;
}
```

The EarClipping method it's self takes a list of Vector2 vertices defining the polygon. Implements the main loop of the ear clipping algorithm. It iteratively checks each triplet of consecutive vertices to determine if they can form an ear. If an ear is found, it is added to the list of triangles, and the vertex forming the tip of the ear is removed from the list. The process repeats until only two vertices are left. This is done to decomposes the entire polygon into a series of triangles.

```
public static List<Tuple<Vector2, Vector2, Vector2>> EarClipping(List<Vector2> vertices)
{
    var triangles = new List<Tuple<Vector2, Vector2, Vector2>>();
    while (vertices.Count > 2)
    {
        int n = vertices.Count;
        for (int i = 0; i < n; i++)
        {
            var a = vertices[(i - 1 + n) % n];
            var b = vertices[i];
            var c = vertices[(i + 1) % n];
            if (IsEar(a, b, c, vertices))
            {
                triangles.Add(new Tuple<Vector2, Vector2, Vector2>(a, b, c));
                vertices.RemoveAt(i);
                break;
            }
        }
    }
    return triangles;
}
```

In Unity, this method allows for dynamic generation of meshes from arbitrary polygon shapes, which can be useful in numerous applications like dynamic terrain generation, procedural modeling, or game level design. The script utilizes Unity's Vector2 and Vector3 classes to handle geometric calculations, making it well-integrated within the Unity environment.

The JunctionRender Script

The script JunctionRender renders junctions in a simulation using data from a traffic control interface (TraCIClient). The script includes functionality to retrieve junction data, process it, generate a mesh using Delaunay triangulation, and render the resulting mesh in the Unity environment.

The JunctionData method is responsible for processing junction data fetched from a traffic control interface, creating meshes for these junctions, and then rendering them within the Unity environment.

The method starts by retrieving a list of all junction IDs. This list represents the identifiers for all junctions that the traffic simulation knows about. Then iterates over each junction ID, skipping any that start with "-" or ":". It fetches the shape data for each valid junction, which includes a list of points defining the junction's geometric boundary.

```
public void JunctionData()
{
    var junctionIds = client.Junction.GetIdList().Content;

    foreach (var junctionId in junctionIds)
    {
        if (junctionId.StartsWith("-") || junctionId.StartsWith(":")) continue;

        try
        {
            var junctionData = client.Junction.GetShape(junctionId).Content;
            List<IPoint> junctionPoints = new List<IPoint>();
```

It uses a HashSet to store unique points since some junctions might have duplicate points. This ensures that each point is processed only once.

```
// Add unique points to the junctionPoints list
HashSet<(double, double)> uniquePoints = new HashSet<(double, double)>();
foreach (var point in junctionData.Points)
{
    var pointTuple = ((double)point.X, (double)point.Y);
    if (!uniquePoints.Contains(pointTuple))
    {
        uniquePoints.Add(pointTuple);
        junctionPoints.Add(new Point((float)point.X, (float)point.Y));
    }
}
```

The method checks that there are at least three unique points. A minimum of three points is necessary to form a single triangle, which is the simplest polygon in mesh generation. If there aren't enough points, the junction is skipped, and a warning is logged.

```
// Check if there are at least 3 unique points
if (junctionPoints.Count < 3)
{
    Debug.LogWarning($"Skipping rendering for junction '{junctionId}' because it has fewer than 3 unique points.");
    continue;
}
```

If there are sufficient points, the method attempts to create a mesh using these points by calling CreateJunctionMesh, which internally uses the Delaunator library to perform Delaunay triangulation. If the mesh creation is successful, it proceeds to render the mesh by calling RenderMesh.

```
public Mesh CreateJunctionMesh(List<IPoint> points)
{
    Mesh mesh = new Mesh();
    List<Vector3> roadVertices = new List<Vector3>();
    List<int> Triangles = new List<int>();
    float onGround = 1.33f;
```

The CreateJunctionMesh method is responsible for creating a 3D mesh from a set of points representing a junction. It uses Delaunay triangulation, a common technique for mesh generation, particularly when dealing with arbitrary point sets that need to be turned into a mesh. This process is ideal for creating efficient, non-overlapping triangles that cover the entire area defined by the points.

```
try
{
    // Use the Delaunator library to create a mesh from the points
    var delaunator = new Delaunator(points.ToArray());
```

A new Mesh object is initialized to hold the vertices and triangles. 'roadVertices' is a list to store the vertex data in 3D. Triangles is a list to define how vertices are connected to form triangles.

```
// Generate vertices and triangles based on Delaunay triangulation
for (int i = 0; i < delaunator.Triangles.Length; i += 3)
{
    for (int j = 0; j < 3; j++)
    {
        int vertexIndex = delaunator.Triangles[i + j];
        IPoint point = points[vertexIndex];
        roadVertices.Add(new Vector3((float)point.X, onGround, (float)point.Y));
    }
```

The method uses the Delaunator library to perform Delaunay triangulation on the given points. Delaunay triangulation ensures that no point lies inside the circumcircle of any triangle. The Delaunator object processes the points and provides a list of indices (Triangles) that describe how to connect the points into triangles.

```
        // Define triangles
        Triangles.Add(i);
        Triangles.Add(i + 1);
        Triangles.Add(i + 2);
    }
```

The triangles defined by the Delaunator are iterated over in steps of three (since each triangle is defined by three points). For each index in the triangle definition, the corresponding point is converted into a Vector3 with a fixed Y value (onGround) to position it slightly above the ground.

```
// Assign vertices and triangles to the mesh
mesh.vertices = roadVertices.ToArray();
mesh.triangles = Triangles.ToArray();
mesh.RecalculateNormals();
return mesh;
```

The triangles list is populated with indices that are adjusted to be sequential because they are directly referencing the roadVertices list. The vertices and triangles are assigned to the mesh.

```
1 reference
public void RenderMesh(Mesh mesh, string junctionId)
{
    GameObject roadObject = new GameObject("Junction " + junctionId);
    MeshFilter meshFilter = roadObject.AddComponent<MeshFilter>();
    MeshRenderer meshRenderer = roadObject.AddComponent<MeshRenderer>();

    // Apply material
    meshRenderer.material = roadMaterial;

    // Assign mesh
    meshFilter.mesh = mesh;
}
```

This method is used to create a visual representation of a junction in a traffic simulation where accurate and visually appealing junction modeling is required. The use of Delaunay triangulation ensures that the mesh is optimal in terms of both aesthetics and computational efficiency, making it suitable for real-time applications.

The RenderMesh method makes a mesh generated for a junction and render it within the Unity environment by attaching it to a GameObject.

SUMO Network

Net.xml file

The .net.xml file is a network file for SUMO that describes the traffic simulation environment.

The root element <net> defines the network. It consists of attributes like ‘version’ and schema location for validation reasons.

```
<net version="1.0" junctionCornerDetail="5
```

```
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/net_file.xsd">
```

The <location> element is responsible for defining the boundaries and coordinate system of a simulation. It uses attributes like netOffset, convBoundary, and origBoundary to transform geographic coordinates in the simulation environment.

```
<location netOffset="0.00,0.00" convBoundary="0.00,-50.00,50.00,100.00"
```

The <edge> element represents a connection or road in the network. Edges may serve different purposes like internal links within intersections or normal roads. Each edge has a unique identifier called id and a function attribute to describe its purpose.

```
<edge id="gneE0" from="gneJ0" to="gneJ1" priority="-1">
    <lane id="gneE0_0" index="0" speed="13.89" length="4
</edge>
```

```
<edge id=":gneJ1_0" function="internal">
    <lane id=":gneJ1_0_0" index="0" speed
</edge>
```

The <lane> element represents individual lanes on an edge. Each lane has an identifier called id, a position index called index, a maximum speed called speed, a length called length, and a geometric description called shape.

```
<lane id="gneE1_0" index="0" speed="13.89" length="42.80" shape="51.60,57.20 51.60,100.00"/>
```

The <junction> element describes intersections or nodes in the network. It has an identifier called id, a type attribute, and coordinates x and y. The junction's geometric shape is defined by the shape attribute. The <request> element defines the behavior of traffic at the junction, including permissible movements and conflicts.

```
<junction id="gneJ1" type="priority" x="50.00" y="50.00" incLanes=
    <request index="0" response="000000" foes="001000" cont="0"/>
    <request index="1" response="000000" foes="111000" cont="0"/>
    <request index="2" response="000000" foes="100000" cont="0"/>
    <request index="3" response="000011" foes="100011" cont="1"/>
    <request index="4" response="000010" foes="000010" cont="0"/>
    <request index="5" response="001110" foes="001110" cont="0"/>
</junction>
```

The <tlLogic> element is used to define traffic light logic at a junction. It has an identifier called id and a type attribute. The <phase> element describes a phase of the

traffic light cycle. Each phase has a duration in seconds and light states like G for green and R for red.

```
<tlLogic id="gneJ2" type="static" programID="0" offset="0">
    <phase duration="15" state="GG"/>
    <phase duration="3"  state="yy"/>
    <phase duration="10" state="rr"/>
</tlLogic>
```

The <connection> element shows how lanes connect between edges, especially at junctions. It has attributes like from, to, fromLane, toLane, dir, and state.

```
<connection from="gneE0" to="gneE2" fromLane="0" toLane="0" via=":gneJ1_0_0" dir="r" state="M"/>
```

This file structure helps SUMO simulate traffic flow by defining how roads, lanes, junctions, and traffic control mechanisms are interconnected in a virtual environment.

Rou.xml

The rou.xml file specifies the details of vehicle types, routes, and specific vehicle trips within a simulated network.

The root element, <routes>, provides the namespace and schema location for validation purposes.

```
<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/routes_file.xsd">
```

The <vType> element defines a vehicle type with specific attributes, including an identifier (e.g. "car"), maximum acceleration and deceleration abilities, driver imperfection (measured in sigma), vehicle length, minimum gap to the front vehicle, maximum speed, and graphical representation in the simulation.

```
<vType id="car" accel="0.5" decel="4.5" sigma="0.5" length="4"
```

The <route> element defines a route within the network, with an identifier and a sequence of edge IDs that make up the route.

```
<route id="lane" edges="gneE0 gneE4 gneE6"/>
```

The <vehicle> element defines individual vehicle instances that follow specified routes, with a unique identifier, vehicle type (linked to a <vType> defined earlier), route (linked to a <route> defined earlier), and a departure time.

```
<vehicle id="veh0" type="car" route="lane" depart="3"/>
```

.sumocfg file

This file is a SUMO (Simulation of Urban MObility) configuration file that is used to set up and control simulations in the SUMO traffic simulation suite.

The root element of this file is the <configuration> element that specifies the XML schema instance used for validation.

```
<configuration xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
```

The <input> element specifies input files necessary for the simulation. There are two input files specified in this configuration file:

* <net-file value="NetworkB.net.xml"/>: Specifies the network file for the simulation. This file contains the definition of roads, junctions, and other network features.

* <route-files value="simple_demand.rou.xml"/>: Specifies the route file, which includes

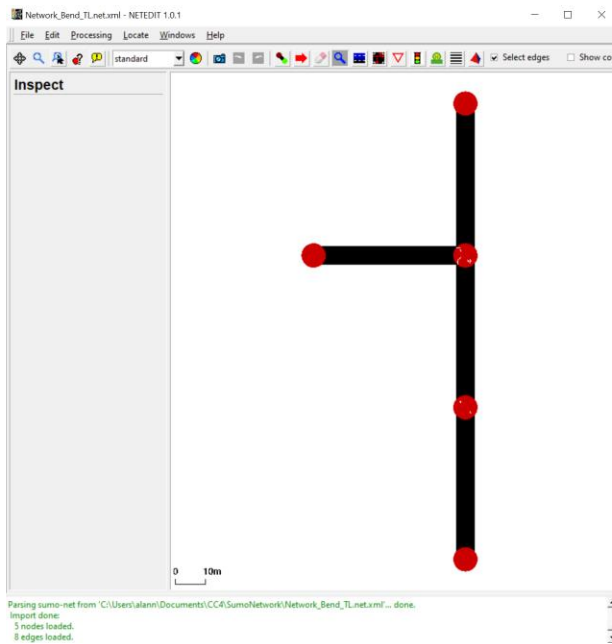
```
<input>
    <net-file value="NetworkB.net.xml"/>
    <route-files value="simple_demand.rou.xml"/>
</input>
```

the definitions of vehicle routes, departure times, and other traffic demand elements.

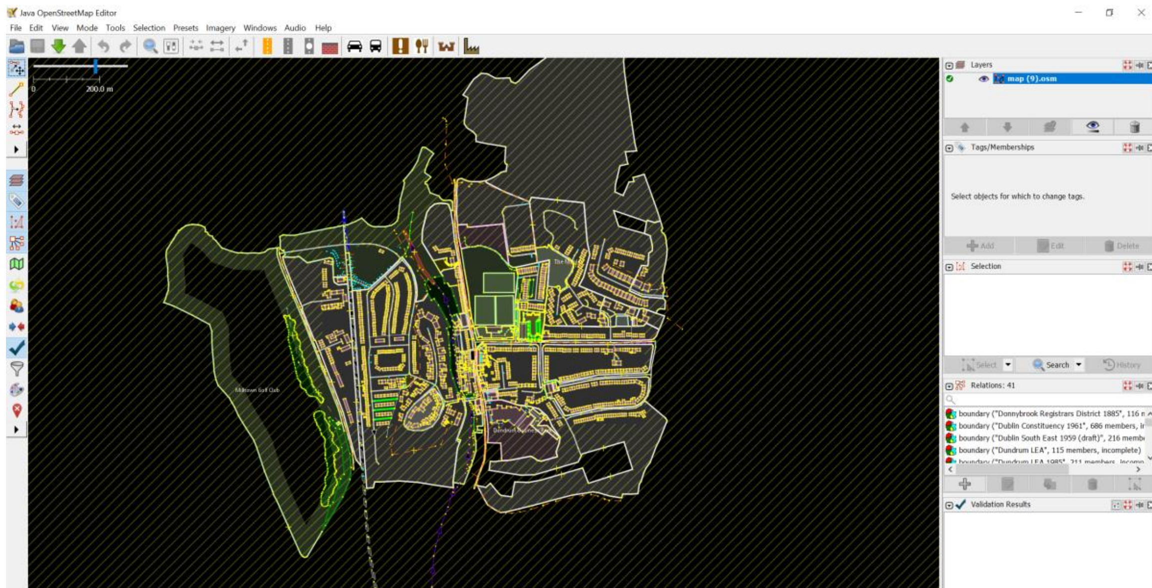
These files are essential for running simulations in the SUMO traffic simulation suite, and this configuration file provides the necessary instructions for setting up and controlling these simulations.

Creating the network

During this project, two methods of network creation were used. The initial method involved using software called Netedit. Netedit is a tool from the Simulation of Urban MObility (SUMO). It is designed to create and modify network files that describe road networks. Below is a depiction of the Straight Netowkr and the Bend_TL network:



The file that is exported by OSM has an extension of .osm and contains a lot of irrelevant information such as park details and housing data. To convert this file to a .xml file and remove the unnecessary data, I use the Java OpenStreetMap editor (JOSM).



Generate random traffic flow

The randomTrips.py script is provided by SUMO. It is used for creating random vehicle trips. It's a part of the SUMO tool suite and generates trip definitions for vehicles in a network. The script selects random origin-destination pairs to create a set of trips between them in a network.

To use the script, the network file (net.xml) is passed in. It contains all the necessary information about the road network, including edges, junctions, and connections. Once the network file is provided, the RandomTrips.py generates trips by randomly selecting start (origin) and end (destination) points on the network.

The trips file created by the script is usually a .tryps.xml file and contains detailed information about individual trips, including their origin, destination and departure time.

```
<routes xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:noNamespaceSchemaLocation="http://sumo.dlr.de/xsd/routes_file.xsd">
    <trip id="0" depart="0.00" from="gneE7" to="gneE2"/>
    <trip id="1" depart="1.00" from="gneE0" to="gneE1"/>
    <trip id="2" depart="2.00" from="gneE3" to="gneE1"/>
    <trip id="3" depart="3.00" from="gneE7" to="gneE6"/>
    <trip id="4" depart="4.00" from="gneE5" to="gneE5"/>
    <trip id="5" depart="5.00" from="gneE5" to="gneE4"/>
    <trip id="6" depart="6.00" from="gneE3" to="gneE1"/>
    <trip id="7" depart="7.00" from="gneE5" to="gneE4"/>
    <trip id="8" depart="8.00" from="gneE0" to="gneE4"/>
    <trip id="9" depart="9.00" from="gneE3" to="gneE5"/>
    <trip id="10" depart="10.00" from="gneE3" to="gneE4"/>
    <trip id="11" depart="11.00" from="gneE3" to="gneE6"/>
```

This script can be re-run everytime, before running the simmulation, using this command.

```
PS C:\Users\alann\Documents\CC4\SumoNetwork> python randomTrips.py -n Network_Straight.net.xml -r Straight_demand.rou.xm
l -b 0 -p 1 -o Straight.trips.xml
```

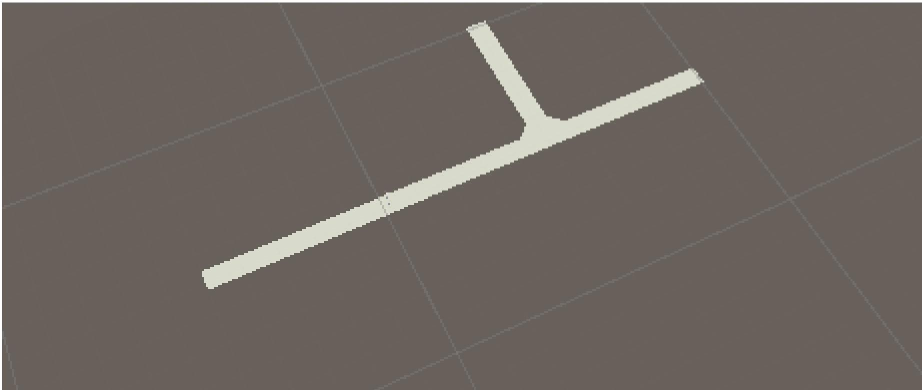
The command goes as follows:

- First initiates the Python interpreter to run the randomTrips.py script. Then -n flag specifies the network file to be used by the script.
- The -r flag specifies the output route file where the script will save the generated trips including the routes.
- The file Straight_demand.rou.xml will contain detailed route information for each trip.
- The -b option sets the beginning of the simulation period. Here it start when the simulation starts.

- The -p option sets the period or interval between departures of trips. Here, it's set to 1 second, meaning a new trip will be generated every second. Which is quite short.
- The -o flag specifies the output file for the trip definitions. This file does not contain the routes themselves but just the trip information.

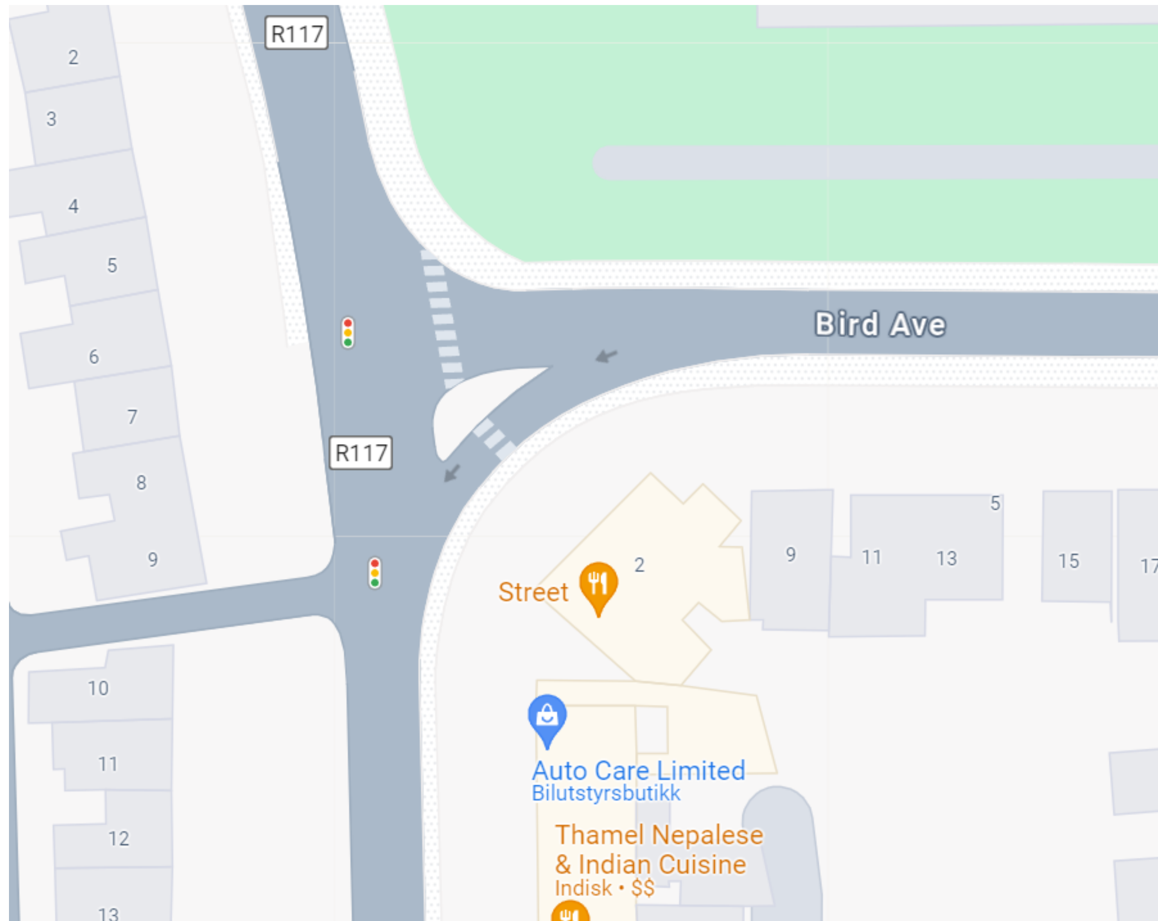
5. Testing and Analysis

To ensure optimal performance when rendering larger roads in a simulation environment such as Unity connected with SUMO, there are several key aspects to focus on.

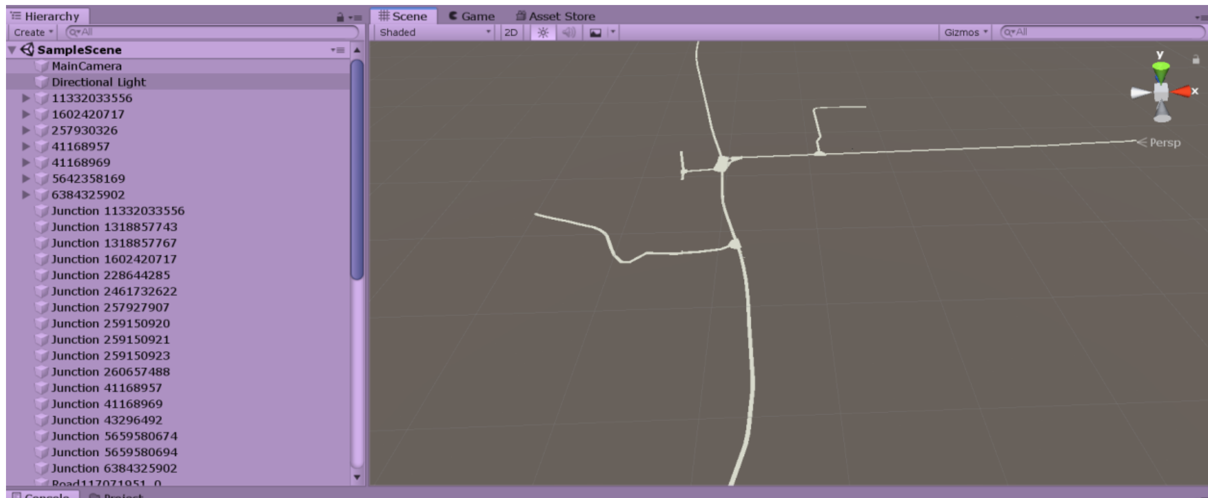


Performance benchmarks play a crucial role in this process. Initially, the baseline was determined by working from a simple single road network to a real-world network. To start, the network used a simple single road created using Netedit, and the LaneRender script generated all lanes the network provided, including internal lanes. These internal lanes, designated by IDs starting with ':', act as filler lanes for the external ones. They round out the ends of lanes or overlap gaps between junctions and lanes. These lanes looked adequate when testing the performance of such a small network. However, when using the OpenStreetMap application, the extracted network consisted of a lot more detailed lanes shape.

After converting the network to an .xml file that SUMO can read, playing the simulation in Unity most of the lanes were generated. However, it crashed when developing the lanes surrounding the junction below.



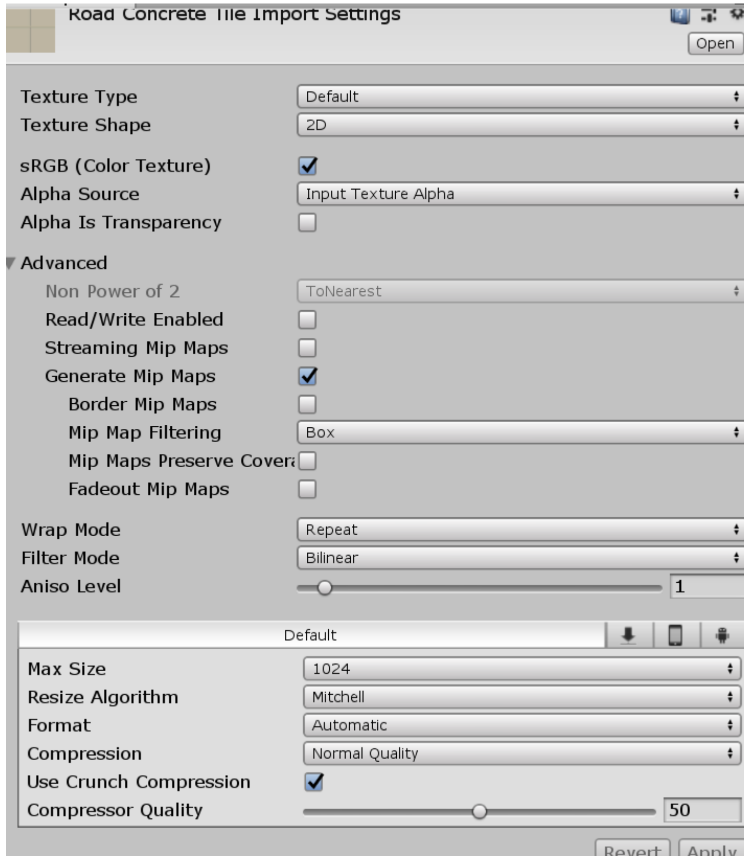
To test where in the process it was struggling, different parts of the algorithms were commented out to handle the road generation. It was clear that when only the two-point road generation script was running, the network would generate, but with missing lanes. However, when the ear-clipping was in the picture, Unity would crash after a few lanes being generated. This led to the process of optimizing the road render script.



Initially, the ear-clipping script handled lanes from three points and up. However, it became apparent that lanes with three points refer mainly to internal filler lanes and small lanes that have a curve to them (valid lanes with this issue was handled by combining its associated lane if found). This is why, in the end, the section of the script dealing with generating lanes with 3 points was commented out, but it was left in case there was a later use case that would require three-point mesh generation.

The next step was to only call the ear-clipping algorithm when necessary. After discovering that the lanes including ':' were not necessary, the 'LaneData' changed drastically. Firstly, the change was just to skip lanes with the ID of ':'. Then skip lanes with the symbol '-' due to these lanes only existing in association with their parallel lane. Lanes that had the '-' merged their shapes with their parallel lanes if found. Which meant that the ear clipping algorithm would be called less.

When looking at the render material, the idea was to make the material as lightweight as possible. The material was compressed to save space on the disk wherever possible. Which was a simple step. However, when generating larger networks it made a few second difference.



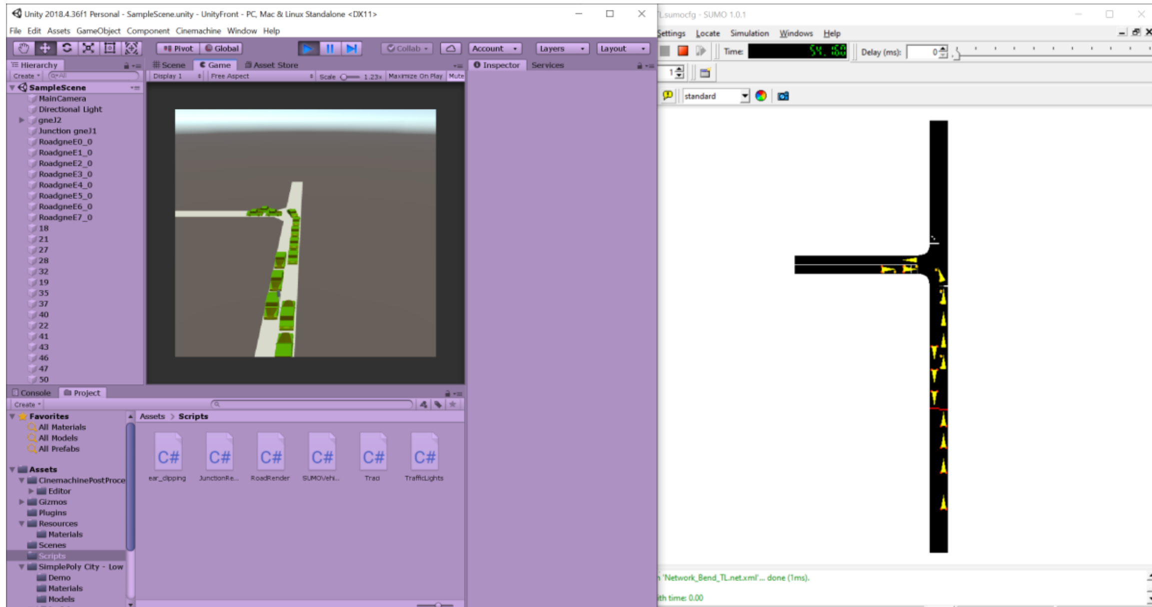
6. Discussion

Discusstion

This project serves as a starting point for creating a simulation that allows people to create their own local traffic simulation. Whilst analyzing the cutout of the Dundrum network, it was apparent during rush hour when bottlenecks would most frequently occur. More specifically, when vehicles must first stop for the traffic light and then stop to wait to drive up the junction. However, to make the simulation more durable, it would be beneficial to have additional parameters to edit the traffic flow. The demand can be changed though the command line by changing the number of seconds before a new vehicle generates. However, since not all roads are as populated depending on if they are main roads or roads leading to a neighborhood. The next step would be to add a restriction on how many percent of the vehicles go in certain directions.

There is a significant focus on generating roads within simulations to provide a comprehensive toolset to address multiple aspects of urban planning, making it a valuable feature. Understanding how traffic flows across a network of roads is important when analyzing congestion, designing traffic signals, and improving overall traffic efficiency. Furthermore, simulating traffic on realistic roads makes it easier to identify potentially dangerous areas and design interventions to enhance safety, such as optimizing road layouts, improving signage, and road markings.

Additionally, generating roads makes it easier for users to simulate their own networks. The vehicles will generate in the correct coordinates. However, the lanes and junctions would need to be added manually, which would be a tedious task. Auto-generating the lanes was an impactful step to make the simulation more flexible for each user's use case.



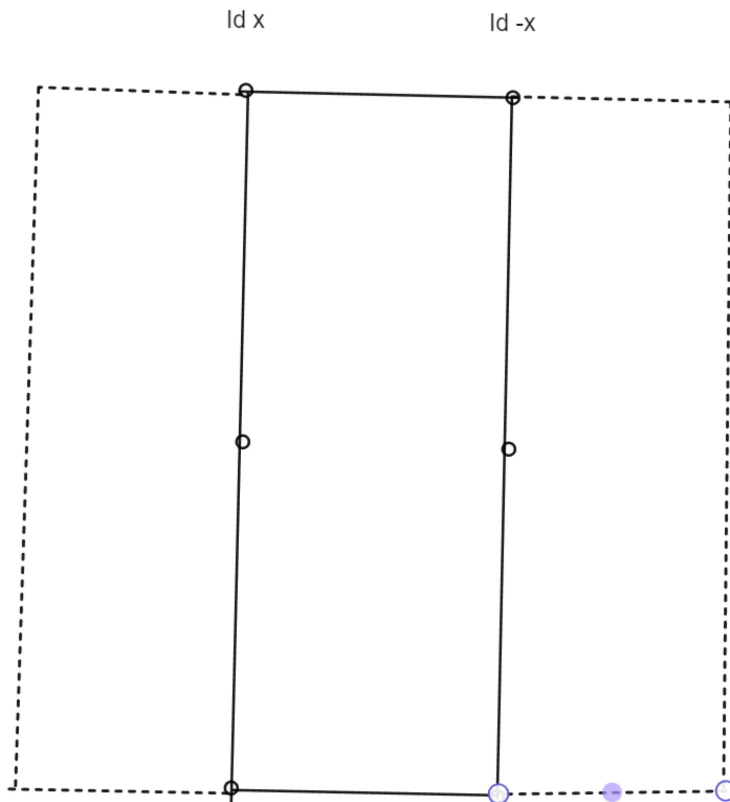
Future Development

The next steps for this simulation include properly handling the User Interface. Currently, the users are required to go into the command line to open the networks and develop new trips. It would be better for the user to connect to different networks from a home page as initially designed. This would require the command line code to be done on Unite's side. Also, it would be nice for users to have different viewpoints. Cinemabrain is already imported in the project. If the user would like to view the simulation though one of the vehicles it would be necessary modification to the SUMO Vehicle script or creating a new script depending on the methodology. Additionally, it would be interesting for the user to edit the time between each vehicle that is generated. To represent the decrease or increase in demand.

The traffic light script will need to be improved. When looking at the few other projects doing the same as this project, they would manually place the traffic lights. The traffic lights, as mentioned before, are associated with a junction. However, SUMO only gives one x,y position for traffic light affecting both directions. This means one will have to rethink how to find out if there are multiple traffic lights for one junction, perhaps by looking at the length of the traffic light phase.

Lastly, there is an issue with the lanes being generated by the ear clipping algorithm. SUMO only gives the center points of the road, which is why it was important to add the width and length of the road for the two-point lanes. However, the ear clipping algorithm doesn't consider the full width of the road. This means that if a car runs on a lane

constructed by the ear clipping algorithm, it looks like it is running on the side of the road, while the lane has generated half of the lane and not the total width.



7. Conclusion

Although there is still more work to be done, the completed features are significant. This project has helped me in terms of personal growth by demonstrating my ability to assert my preferences on how the project should appear, rather than timidly adding features. The project was challenging in multiple ways, not only because creating a simulation is not an easy feat, but because information gathering was challenging for this set up. Nevertheless, the outcome was positive due to what one has accomplished so far.

References

Boris S. Kerner, J. Barcélo, Introduction to Modern Traffic Flow Theory and Control, 2010,

https://www.academia.edu/36079982/Introduction_to_Modern_Traffic_Flow_Theory_and_Control

Daellenbach, Hand G., systems and decision making a management science approach, 1994,

https://www.academia.edu/23088679/Daellenbach_systems_and_decision_making_a_management_science_approach_wiley_1994_

Anu Maria, Introduction to modeling and simulation, 1997, https://dl.acm.org/doi/10.1145/268437.268440

Martin Treiber, Ansgar Hennecke and Dirk Helbing, Microscopic Simulation of Congested Traffic, https://mtreiber.de/publications/micro_tgf99.pdf

Wilco Burghout, Hybrid microscopic-mesoscopic traffic simulation, 2004, https://www.diva-portal.org/smash/get/diva2:14700/FULLTEXT01.pdf

Denso C. Gazis, THE ORIGINS OF TRAFFIC THEORY, 2002, https://pubsonline.informs.org/doi/pdf/10.1287/opre.50.1.69.17776

AcqNotes, Modeling & Simulation, 2021, https://web.archive.org/web/20230603104655/https:/acqnotes.com/acqnote/tasks/modeling-simulation-overview

Jaume Barcelo, Fundamentals of Traffic Simulation, 2010, file:///C:/Users/alann/Downloads/331%20(7).pdf

Martin Schönhof, Dirk Helbing, Transportation Research, 2009, http://web.math.unifi.it/users/cime/Courses/2009/01/200917-Notes.pdf

Jaume Barcelo, FUNDAMENTALS OF TRAFFIC SIMULATION, 2010, https://epdf.tips/fundamentals-of-traffic-simulation.html

Petter Arnesen, Odd A. Hjelkrem, AN ESTIMATOR FOR TRAFFIC BREAKDOWN PROBABILITY BASED ON CLASSIFICATION OF TRANSITIONAL BREAKDOWN EVENTS, 2017, https://sintef.brage.unit.no/sintef-xmlui/bitstream/handle/11250/2586781/final_Est_Breakdown_Prob.pdf?sequence=2

Zhijing J., Tristan S., Ramesh R., 2019, 3D Traffic Simulation for Autonomous Vehicles in Unity and Python, https://arxiv.org/pdf/1810.12552

Cristina O., Javier E., Alberto D., Carlos B., Luis S., Markus K., 2018,Connection of the SUMO Microscopic Traffic Simulator and the Unity 3D Game Engine to Evaluate V2X Communication-Based Systems, https://www.mdpi.com/1424-8220/18/12/4399

Leyre A., Cristina O.,2019, Vehicle-Pedestrian Interaction in SUMO and Unity3D https://www.jku.at/fileadmin/gruppen/344/Publications/Vehicle-Pedestrians_Interaction_SUMO.pdf

Shin I., Beirami M., Cho S., Yu, Y.,2015, Development of 3D Terrain Visualization for Navigation Simulation using a Unity 3D Development Tool, https://koreascience.kr/article/JAKO201522359516445.page

Antoine C., Bruno D., 2021, Intra-City Traffic Data Visualization: A Systematic Literature Review, https://ieeexplore.ieee.org/abstract/document/9484412

Ismail B., Serdar B., Gurcan B., A.P. B., Hairi K., Alias A., R., 3D MODELLING AND VISUALIZATION BASED ON THE UNITY GAME ENGINE – ADVANTAGES AND CHALLENGES, 2017, https://isprs-annals.copernicus.org/articles/IV-4-W4/161/2017/isprs-annals-IV-4-W4-161-2017.pdf

Alberto M., Gershon B., Mehler B., Bryan R., 2020, Driver-initiated Tesla Autopilot Disengagements in Naturalistic Driving, https://www.researchgate.net/publication/344378805_Driver-initiated_Tesla_Autopilot_Disengagements_in_Naturalistic_Driving

Guang Y., Yunzhi X., Lingzhong M., Pengqi W., Yuan S., Qinghong Y., Qian D., 2021, Survey on Autonomous Vehicle Simulation Platforms, https://dsa21.techconf.org/download/DSA2021_FULL/pdfs/DSA2021-1sP33wTCujRJmnnXDjv3mG/439100a692/439100a692.pdf

Sergey D., Alexander A., Dmitry L., Alexey N., Evgeniy S., 2020, IOP Conference Series: Materials Science and Engineering, https://iopscience.iop.org/article/10.1088/1757-899X/918/1/012058/meta

Jichao W., Lucas P., John R. M., Bin W., Chenn Z., 2015, Simulation and visualization of industrial processes in unity, https://dl.acm.org/doi/abs/10.5555/2874916.2874929

C. Donalek et al., "Immersive and collaborative data visualization using virtual reality platforms," 2014 IEEE International Conference on Big Data (Big Data), Washington, DC, USA, 2014, pp. 609-614, doi: 10.1109/BigData.2014.7004282. keywords: {Data visualization;Three-dimensional displays;Collaboration;Visualization;Big data;Virtual reality;Abstracts;astroinformatics;visualization;virtual reality;data analysis;big data;pattern recognition},

Akshay G., 2023, Visualization of Industrial Production Processes using 3D Simulation Software for Enhanced Decision-Making, https://www.researchgate.net/publication/374862435_Visualization_of_Industrial_Production_Processes_using_3D_Simulation_Software_for_Enhanced_Decision-Making

M. Dikmen, C. Burns, 2017, Trust in autonomous vehicles: The case of Tesla Autopilot and Summon, https://ieeexplore.ieee.org/abstract/document/8122757

M. Dikmen and C. Burns, "Trust in autonomous vehicles: The case of Tesla Autopilot and Summon," 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC), Banff, AB, Canada, 2017, pp. 1093-1098, doi: 10.1109/SMC.2017.8122757. keywords: {Automation;Vehicles;Advanced driver assistance systems;Analysis of

variance;Autonomous automobiles;Correlation;System analysis and design;trust in automation;automated driving;advanced driver assistance systems},

A. Eriksson a, V.A. Banks b, N.A. Stanton, 2017, Transition to manual: Comparing simulator with on-road control transitions, https://www.sciencedirect.com/science/article/abs/pii/S0001457517301094

G. Yang et al, 2021, Survey on Autonomous Vehicle Simulation Platforms, https://ieeexplore.ieee.org/abstract/document/9622937

G. Yang et al., "Survey on Autonomous Vehicle Simulation Platforms," 2021 8th International Conference on Dependable Systems and Their Applications (DSA), Yinchuan, China, 2021, pp. 692-699, doi: 10.1109/DSA52907.2021.00100. keywords: {Point cloud compression;Industries;Analytical models;Costs;Tools;Autonomous vehicles;Research and development;autonomous vehicle;test and verification;simulation test;simulation elements;simulation technology},

Olof E., Jennifer L., Padmini S., Hans-Martin H., Setting AI in context. A case study on defining the context and operational design domain for automated driving, https://arxiv.org/pdf/2201.11451

Schulke A., Mai Vi N., 2023, The introduction of self-driving / full-automation trucks: Will we live among these modern dinosaurs? https://www.econstor.eu/bitstream/10419/268391/1/1831592185.pdf

Marie-Ange L., Frédéric Le M., Eric M., Julien D., Richard D., 2014, VANET Applications: Hot Use Cases, https://arxiv.org/pdf/1407.4088

Vanessa N., David W., Farzaneh S., Maryam Z., 2021, Application of Advanced Driver-Assistance Systems in Police Vehicles, https://journals.sagepub.com/doi/epub/10.1177/03611981211017144