CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Simulating Traffic Flows and Analysing Road Network Design

Investigating the relationship between road network design and traffic congestion

Bachelor's thesis in Computer science and engineering

MARTIN BLOM
FELIX JÖNSSON
HANNES KAULIO
MARCUS SCHAGERBERG
JAKOB WINDT

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2023

Bachelor's thesis 2023

Simulating Traffic Flows and Analysing Road Network Design

Investigating the relationship between road network design and traffic congestion

MARTIN BLOM
FELIX JÖNSSON
HANNES KAULIO
MARCUS SCHAGERBERG
JAKOB WINDT

UNIVERSITY OF GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2023

Simulating Traffic Flows
Investigating the relationship between road network design and traffic congestion
MARTIN BLOM FELIX JÖNSSON HANNES KAULIO
MARCUS SCHAGERBERG JAKOB WINDT

Supervisor: Natasha Bianca Mangan, Interaction Design and Software Engineering
(if applicable) Advisor: Name, Company or Institute
Examiner: Name, Department

Bachelor's Thesis 2023
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
Gothenburg, Sweden 2023

Simulating Traffic Flows
Investigating the relationship between road network design and traffic congestion
MARTIN BLOM, FELIX JÖNSSON, HANNES KAULIO, MARCUS SCHAGERBERG, JAKOB WINDT

Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

This document is *only* a LaTeX template. It is not meant to suggest a particular structure. Also, even if this document is written in English, is is not meant to suggest a report language. You can adopt it to your language of choice. In this document, the bibliography is hand made. However, we suggest that you strongly consider using BibTeX, to further automate the creation of the bibliography.

Keywords: put, here, keywords, describing, areas, the, work, belongs, to

Acknowledgements

If you want, you can here say thank your supervisor(s), company advisors, or other people that supported you during your project.

Martin Blom, Felix Jönsson, Hannes Kaulio, Marcus Schagerberg, Jakob Windt
Gothenburg, April 2023

Contents

List of Figures

List of Tables

Glossary

Agent: Autonomous systems that inhabits an environment and act based on predefined rules.
Agent Based Model (ABM): A computer simulation model in which agents interact with each other and their environment to produce emergent behavior.
Data Structure: A way of organizing and storing data in a computer so that i can be accessed, manipulated, and modified efficiently. Some common examples are arrays, stacks, and linked lists.
Information Visualization: Field that focuses on creating meaningful and easy to to interpret graphical representations of data.
MonoBehaviour: Base class for Unity scripts. Provides access to event functions such as Start(), Update(), and so on.
Pooling: A technique used in programming to improve performance by reusing objects instead of creating new ones.
Prefab: A reusable object in Unity that stores a configuration and can be used as a template for creating assets.
Scrum: The scrum agile project management framework provides structure and management of work and is popular among software development teams.
Scrum-boards: A bulletin board that keeps track of a backlog, the current sprint, and completed stories.
Story: In the scrum framework, a story is essentially a set of tasks that will result in a new or updated desired functionality/product.
Unity: Cross-platform game engine.
Unity Asset: A file containing reusable content that can be imported into Unity projects. Can be accessed through Unity's official platform or imported via third-party repositories.
User Testing: A method of testing and evaluating a product by observing and gathering data from real users.
UI: User Interface (UI) is the point between human-computer interactions. It is what is used for user interactions with the program.
UX: User Experience (UX) refers to the overall experience of the actual user of a product. The goal of good UX design is to create intuitive and enjoyable products.
C#: C# is a programming language developed by Microsoft that runs on the platform .NET Framework. C# is pronounced as "C sharp" and belongs to the programming language family of C.
C++: C++ is one of the most popular general purpose programming languages. C++ is pronounced as "C plus plus" and belongs to the programming language family of C.
A*: A* is a popular graph traversal and path searching algorithm due to it's completeness and optimal efficiency. A* is used to find the shortest possible path from one specified node to another.
Repository: A repository acts as a container that stores a projects files and their individual revision history.
Cubic Bézier Curve:

Wire Frames: Wire Frames depict how the UI layout will appear during different stages of the program.
OSM:

1

Introduction

1.1 Background

1.2 Purpose

The purpose of the project is to design and construct a 3D traffic simulation tool with high accessibility that should provide detailed and accessible data that allows the user to evaluate the performance of different road networks and traffic scenarios, and make informed decisions about urban planning and infrastructure. Data should be presented both in real-time, and as post-simulation data through presentation of relevant statistics. By adjusting the parameters of the simulation, the user should be able to witness the effect of their tweaking, and easily see if their changes have a positive or negative impact across relevant environmental dimensions such as traffic flow, travel time, and emissions.

1.3 Related Work

1.3.1 Microscopic Traffic Simulations

There exists a plethora of different available tools for traffic simulation, which are in turn built upon different underlying models. In Nguyen's widely cited paper, he classifies the currently available simulations according to the following four categories with regards to their granularity of model: Macroscopic, Microscopic, Mesoscopic, and Nanoscopic. These tools allow researchers to answer complex questions and evaluate different scenarios in both real-time observations and through post-simulation data analysis.

Agent-based traffic models position themselves within the Microscopic category and allows for a highly realistic representation of traffic flow, where emergent behaviors such as congestion and bottleneck formation can occur due to the natural occurring interplay of the autonomous agents within the simulation.

Simulation of Urban MObility (SUMO) is a highly popular and freely available microscopic traffic simulation that was initially developed at the German Aerospace Center (DLR). It provides the users with the ability to model a range of trans-

portation agents, including cars, buses, bicycles, and pedestrians, in both urban environments. The simulation is deterministic by default, but users have the option to introduce stochastic processes in different ways, making it a highly versatile tool for traffic simulation and analysis.

The software offers various tools creating networks and editing these through a map editor which can also import and export network data from external sources. In addition to this, SUMO provides the user with features for visualizing the obtained data and analyzing it through various reports and plots. Users can also customize SUMO to accommodate their specific need through the application programming interface (API) and integrate the simulation with other software. SUMO is also capable of modeling emission based on vehicle type and speed.

Another popular for simulating traffic is the commercial software PTV Vissim designed by the German-based company PTV group which specializes in mobility and transportation solutions. It is designed to be quick and simple to set up with no scripting required by the user and comes with a highly customizable editor. The software is part of a larger suite named PTV Traffic Suite, which allows it to exchange data and collaborate across multiple platforms.

PTV Vissim offers a similar feature list to SUMO but differs in some important areas. Firstly, they are built upon different Car following models. SUMO implements the Krauss model which is based on the idea that drivers adjust their vehicle's speed and headway based on their perceived safety and comfort. Though a relatively easy to understand model, it has the disadvantage of assuming that the drivers only react to the speed and distance of the vehicle in front of them, and excludes a lot of factors such as the traffic signals, shape of the road, and driver psychology.

Meanwhile, PTV Vissim implements the Wiedemann models which share a lot of the same model parameters as the previously mentioned Krauss model but differ in their mathematical formulations and the way they calculate the acceleration of a vehicle. The model also introduces additional parameters, for example, a parameter for setting driver aggressiveness which regulates how risk-taking a driver is willing to be, and a parameter to regulate reaction time. Due to the additional parameters introduced here, the Wiedemann model is considered more realistic compared to the Krauss model, but is at the same time deemed to be more complex and requires a significant amount of parameter calibration.

Another crucial difference between the two simulation tools is that SUMO natively only supports graphical representation of a traffic environment in low detailed 2D, while PTV Vissim offers a feature rich 3D visualization. The latter provides a range of tools for customizing the 3D visualization, including options for importing third party 3D models, setting and creating custom textures, and defining various customized visual effects.

1.3.2 ABMU

Agent-Based Modelling Framework for Unity3D (ABMU) is an open-source 3D agent-based modeling platform developed with the Unity3D game engine. It was developed as a response to the lack of support for 3D ABMs, and offers an extensible and user-friendly programming interface for Unity's resources to create the foundation for a powerful and extensible model.

Some key features of ABMU include event scheduling and synchronous updates by a dedicated scheduler class to delegate events in a manner that is decoupled from Unity's native event execution order, ensuring a more robust and accurate simulation. Furthermore, ABMU introduces wrappers around native Unity methods, enabling them to be used as Steppers within the framework, and allowing for easy extension using existing Unity libraries which can provide complex behaviors such as advanced pathfinding and physics simulation systems.

ABMU also provides a set of example models, ranging from novel neighbor detection models to replications of well-known ABM models from literature, such as Epstein and Axtell's Sugarscape model, Reynolds' Boids model, and Schelling's segregation model. These examples demonstrate the flexibility and capability of the framework, as well as offer guidance for users looking to develop their own models.

1.4 Societal and ethical aspects

Ethical aspects can be broken down into two parts: aspects related to the method of the project, and possible consequences for users of the final product and society as a whole. With regards to making sure that our methods adhere to an ethical practice, the main thing to be aware of here lays in data handling during user testing. It will be crucial to ensure that data is both collected and stored in a responsible manner. This will involve structuring clear consent forms, anonymizing data, and adhering to relevant data protection regulations such as the General Data Protection Regulation (GDPR).

The ethical aspects of the finished product however, is accompanied by more complex considerations. One of the goals of this project is to create a tool that can be used by different end-users of various occupation connected to traffic planning, and offer these users insight about the efficiency and emission associated with different set up of road networks. Since these insights might lead to real-life decisions regarding actual infrastructure, careful consideration will have to be taken regarding the design we choose to implement and what sort of consequences these might have in the finished product. To instill model credibility and prevent model realism bias, we will have to communicate any assumptions and limitation of the model in an easy and accessible manner. ABMs are generally considered challenging with regards to validation and traceability[1], and failing to mitigate these might lead to decisions being implemented on obscure premises.

2

Theory

2.1 Unity

Unity is a cross-platform game engine used to create both 2D and 3D games. Unity supports a lot of features that speed up development time.

2.2 Bézier curves

A Bézier curve is a parametric curve between two points, that curves according to a set of intermediate points. The points are called control points, where the first and last point are the endpoints of the curve. A linear Bézier curve only has two points, which means that it is a line between the points. It is defined by the following function:

$$P(t) = P_0 + t(P_1 - P_0), \quad 0 \leq t \leq 1$$

Note that $P_1 - P_0$ is the vector starting in P_0 and ending in P_1. The parameter t is the ratio along the line, with $t = 0$ and $t = 1$ marking the endpoints. This is what is known as linear interpolation in mathematics. A linear Bézier curve is therefore simply a linear interpolation between the points P_0 and P_1. Let's define this as $P_0 \rightarrow P_1$. A visual representation can be found below.

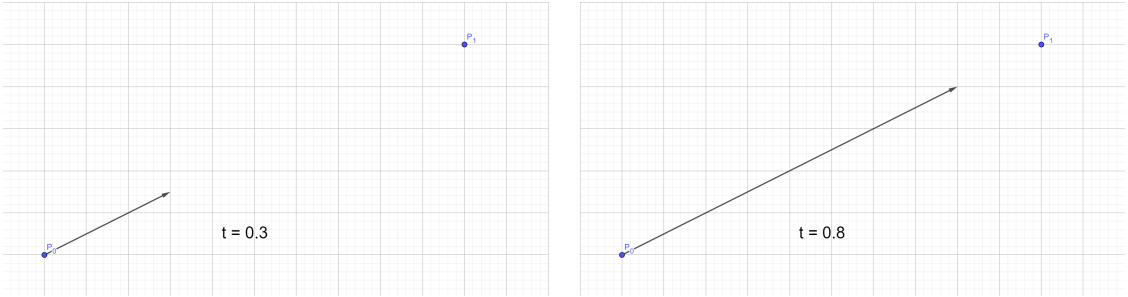

Figure 2.1: A linear interpolation for two values of t

Expanding on this, a quadratic Bézier curve consists of two linear interpolations:

$$A) \quad P_0 \rightarrow P_1$$

$$B) \quad P_1 \rightarrow P_2$$

It is then defined as the linear interpolation between these points, i.e $A \rightarrow B$. All linear interpolations in this case depend on the same t, which is what creates the curvature of the Bézier curves.

Since a quadratic Bézier curve has three points, it will have two endpoints as well as an additonal control point between them. By moving the control point, the shape of the Bézier curve can be altered. This is presented with the following examples, the first three of which have static endpoints demonstrating how the middle control point can be used to form the curve. The final example eludes to the fact that the control points can be placed anywhere without the requirement of any order.

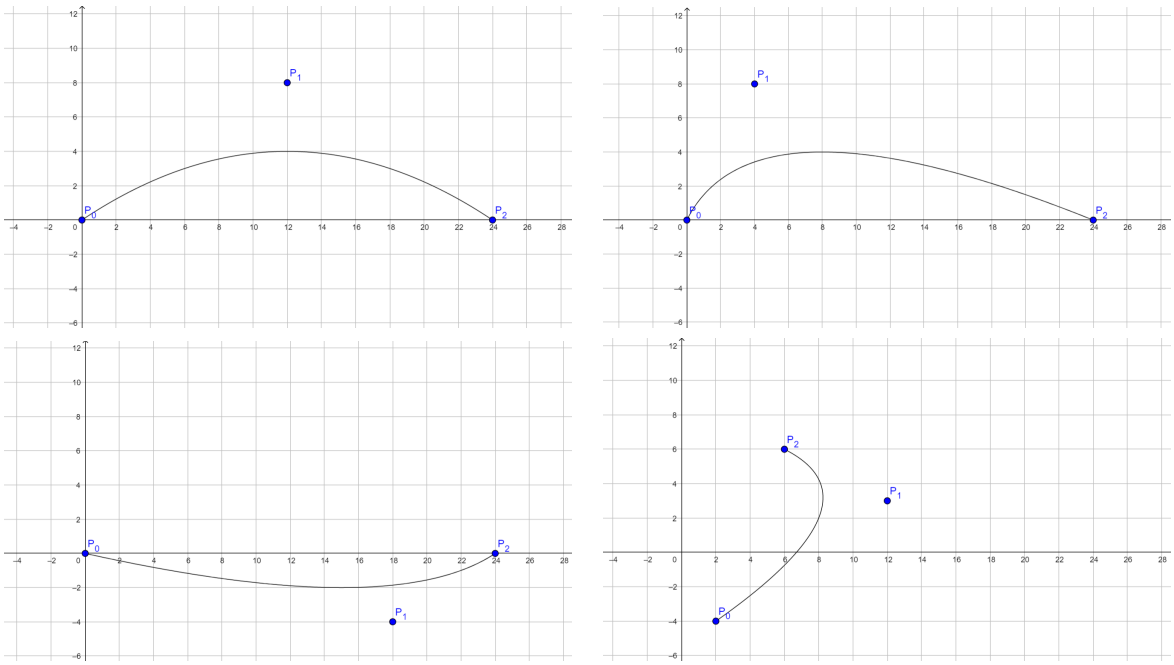

Figure 2.2: Four examples of quadratic Bézier curves

2.2.1 Cubic Bézier Curve

A cubic Bézier curve expands on the quadratic curve in the same fashion as the quadratic expanded on the linear Bézier curve, adding another layer of linear interpolations. A cubic Bézier curve has four control points, two of which are endpoints.

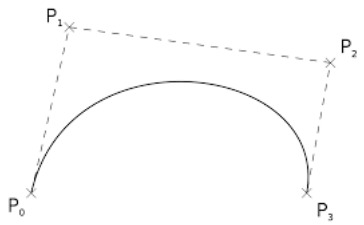

Figure 2.3: Cubic Bézier curve with control points P_0, P_1, P_2 and P_3

The cubic Bézier curve can be defined by the formula[2]:

$$P(t) = (1-t)^3 P_0 + 3t(1-t)^2 P_1 + 3t^2(1-t)P_2 + t^3 P_3, \quad 0 \leq t \leq 1$$

Properties of the Cubic Bézier curve relevant to this paper are the following:

1. The endpoints P_0 and P_3 lay on the curve
2. The curve is continuous, infinitely differentiable, and the second derivatives are continuous.
3. The tangent line to the curve at the point P_0 is the line P_0P_1. The tangent to the curve at the point P_3 is the line P_2P_3.
4. Both P_1 and P_2 lay on the curve only if the curve is linear.
5. A Bézier curve is contained within the convex hull of the control points. For the application in Unity, this means that a Bézier curve is completely contained within the bounding box created by its control points.

2.2.2 De Casteljau's algorithm

In 1959 the french mathematician Paul de Casteljau constructed an algorithm for dividing a Bézier curve into two. The union of these Bézier segments is equivalent to the original curve.

2.2.3 Bézier Clipping

Finding the intersection points between two Bézier paths is not as straight forward as for something like two lines. To solve this, an algorithm called Bézier clipping explained in [3] can be used. It utilises the convex hull property of Bézier curves - and therefore Bézier paths - as well as de Casteljau's algorithm for splitting curves. An implementation for finding all intersection points between two Bézier paths using Bézier clipping is outlined below.

intersections ← Empty list
epsilon ← A value >0 small enough for the desired accuracy
procedure FindBezierPathIntersections(A, B)

if $A.BoundingBox$ does not intersect $B.BoundingBox$ **then**
return
end if
if $A.BoundingBox.Size + B.BoundingBox.Size < epsilon$ **then**
$intersections \leftarrow$ Midpoint between A and B
return
end if
$A_1, A_2 \leftarrow SplitWithDeCasteljau(A, 0.5)$
$B_1, B_2 \leftarrow SplitWithDeCasteljau(B, 0.5)$
$FindBezierPathIntersections(A_1, B_1)$
$FindBezierPathIntersections(A_1, B_2)$
$FindBezierPathIntersections(A_2, B_1)$
$FindBezierPathIntersections(A_2, B_2)$
end procedure

Worth noting is the *Midpoint*, which is one of many possible approximations of the intersection point. For a small enough *epsilon*, the approximation used is trivial as the segments approaches points as *epsilon* approaches 0.

2.2.4 Composite Bézier curve

A composite Bézier curve is a spline made out of Bézier curves. The series of Bézier curves are joined together end to end with the start point of one curve coinciding with the end point of the other curve. This is used in the projects as it allows for chaining of cubic Bézier path segments creating a spline. This is used for creating the roads and railroads.

2.3 A* Algorithm

A* is an algorithm widely used for pathfinding and graph traversal [4]. Peter Hart, Nils Nilsson, and Bertram Raphael first presented the algorithm in 1968 as part of a project focused on constructing a mobile robot capable of autonomously devising its actions. It is classified as an informed search algorithm since it greedily explores the pathfinding environment by taking into account both the cost of the path from the starting node to the one that is currently being explored, as well as a heuristic function that estimates the distance between the currently explored node and the goal node. Given a start and end node in a weighted graph, the algorithm will find the shortest path between the nodes. Together, these two form an estimate function of the best path towards the goal. A* is complete under the precondition that the search space is finite, and the branching factor is also finite, which guarantees that if a path exists, it will be found. Furthermore, if some additional conditions are fulfilled with regards to the heuristic function, A* can be guaranteed to return an optimal path. For this to be the case, the heuristic function needs to be admissible or consistent, since a consistent function is also, by definition, admissible.

In order to implement the A* algorithm, an open and closed set of nodes is utilized, as well as a few essential variables and functions. These are the key elements used in the algorithm:

- Start node s: The initial position from which the search begins.
- Current node n: The node being evaluated during the search process.
- Target set T: Contains one or more goal nodes that the algorithm is trying to reach.
- Total estimated cost function $f(n)$: The sum of the cost from the start node to node n (denoted by $g(n)$) and the heuristic estimate of the cost from the node n to the target node (denoted by $h(n)$).

With these definitions in place, the A* algorithm can be described using the following steps:

1. Label the start node s as "open" and compute $f(s)$.
2. Choose the open node n with the smallest 'f' value. Break ties randomly, but always prioritize nodes in the target set T.
3. If n is in T, label n as "closed" and conclude the algorithm.
4. Otherwise, mark n as closed and generate all adjacent nodes by exploring the neighboring nodes that can be reached from n in the graph. Compute f for each adjacent node of n and label each adjacent node not already marked closed as "open". If a closed node n_i is an adjacent node of n and its current $f(n_i)$ is smaller than its previous f value when it was marked closed, relabel it as "open". Return to Step 2.

2.4 Procedural mesh generation

All physical objects in Unity have an associated mesh, i.e. their surfaces. A cube for example can be thought of as having a mesh consisting of 6 different surfaces. In computer graphics, a triangle mesh is a type of mesh where the surfaces are created through a set of points, called vertices. These vertices are then joined together by a set of triangles. Going back to the cube example, a cube in its simplest form would have 12 triangles and 8 vertices. The eight vertices are at the corners of the cube. Each face of the cube has the shape of a square, which can be created with two triangles, hence double the amount of triangles as square faces.

2.5 Scrum and Agile Software Development

Agile Software Development is a software development framework which emphasizes vertical development cycles, where software should be delivered frequently in atomic slices to enable quick feedback and high flexibility with regards to how the product

develops. When developing complex products, and especially when the development team has not worked on anything similar to the current developed product, implementing an Agile framework can be particularly important. Since features is delivered in small complete chunks, this minimizes the investment risk compared to a more horizontal feature development.

2.6 ABM

Agent-Based Modeling (ABM) is a computational modeling approach that facilitates the analysis and simulation of complex systems by depicting a system's individual elements (agents) and their interactions. This method enables researchers to investigate how the combined behavior of a system emerges from the attributes and actions of its individual components. In contrast to conventional models, which typically depend on mathematical tractability and differential equations for portraying behavior from a macroscopic viewpoint, ABMs face fewer restrictions and can encompass more aspects of real-world systems. As a result, these models can simulate intricate scenarios without relying on equally complex mathematics, while still achieving satisfactory, and sometimes, even more precise outcomes compared to models that overlook the individual behaviors ABMs are capable of representing. It should be noted, though, that ABMs can also integrate more sophisticated mathematics and techniques, like neural networks or advanced learning approaches, to more accurately depict the complexities and dynamics of individual agents within the system.

2.6.1 Key features

ABMs consist of individual agents that interact with each other and their environment. Agents can represent various entities such as organisms, humans, businesses, and so on. These agents are characterized by their uniqueness, local interactions, and autonomy. They can have different attributes such as size, location, and resource reserves, and they interact with their neighbors in a specific "space," such as a geographic area or a network. The mentioned space is typically relative small in the scope of the total simulation space. Agents act independently and pursue their own objectives, adapting their behavior according to their current state, the state of other agents, and their environment.

2.6.2 Emergence and across-level modeling

ABMs are particularly useful for studying emergent system behaviors that arise from the interactions and responses of individual components to each other and their environment. This allows researchers to explore how a system's dynamics is linked to the characteristics and behaviors of its individual components. Due to this, ABMs are considered across-level models because they focus on the interactions between the system level and the individual agent level. In these across-level models, the agents' behaviors and decision-making processes are modeled explicitly by the

researchers, while the emergent properties of the system as a whole stem from these micro-level interactions that occurs at run-time.

Across-level models allow for a more nuanced understanding of complex systems, as they enable researchers to bridge the gap between micro-level interactions and macro-level outcomes. By capturing the heterogeneity of agents and their responses to their environment, across-level models can shed light on the mechanisms that drive system-level behavior, facilitating the identification of key feedback loops and dependencies within the system.

Additionally, across-level models enable researchers to investigate the impact of various factors at both the individual and system levels, such as how changes in individual behaviors or environmental conditions may affect the overall system dynamics. This approach allows for a more thorough exploration of the robustness and adaptability of the system, providing valuable insights for policy development and system management.

2.6.3 Advantages and applications

ABMs can address complex, multilevel problems that are too difficult to tackle with traditional models. Predator-prey dynamics serve as a classic example of a system traditionally modeled using differential equations and advanced calculus. However, these systems can also benefit from Agent-Based Modeling (ABM). By employing ABMs to study predator-prey interactions, researchers can gain deeper insights into the adaptive behaviors and decision-making processes of individual organisms, as mentioned by B. Harvey and J. Giske. ABMs allow for the representation of heterogeneous agents and the examination of emergent properties arising from their interactions, which can be particularly valuable in understanding the complexities of real-world systems. Applying ABMs can bridge the gap between theoretical and empirical research, highlighting gaps in our knowledge of individual behaviors. and contribute to refining existing theories.

Although the method appears straightforward to apply, researchers argue that this can create a false impression that the underlying concepts are just as simple to grasp. While ABM might seem technically uncomplicated, it possesses considerable conceptual depth, which frequently results in its incorrect utilization.

2.7 Design Patterns

In software engineering, design patterns are common solutions for recurring problems encountered when building complex software. A design pattern can be described as a tried and tested blueprint based on well-known object-oriented principles, such as the SOLID[1] principles that can be applied in many different contexts to solve

[1]SOLID is an acronym that represents five important design principles for object-oriented programming. These are: Single Responsibility Principle, Open/Closed Principle, Liskov Substitution Principle, Interface Segregation Principle, and Dependency Inversion Principle. The purpose of these principles is to improve maintainability, readability, and extensibility.

various problems.

Christopher Alexander initially introduced the idea of patterns in his book, "A Pattern Language: Towns, Buildings, Construction." In this work, he presented a "vocabulary" for designing urban landscapes. The building blocks of this vocabulary consist of patterns that address various aspects of urban design, such as the height of windows, the number of floors in a structure, the size of green spaces within a community, and other similar elements.

This concept was later adapted by the "Gang of Four" (Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides) and translated to the domain of software engineering in their seminal book "Design Patterns: Elements of Reusable Object-Oriented Software," which was published in 1994. This book offers a catalog of 23 reusable design patterns for object-oriented programming that are based on industry experience and observations from the authors.

Today, many design patterns have been integrated into programming languages themselves and are therefore taken for granted by users. For example, the Visitor pattern is a behavioral design pattern that allows you to separate the algorithm from the object structure it is supposed to operate on. One concrete realization of the Visitor pattern's integration into a modern programming framework can be found in the ubiquitous for-each loop. The for-each loop allows for iteration over a collection of elements without the need for an explicit counter index, effectively separating the algorithm responsible for the iteration from the underlying data structure.

```
List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };

for (int i = 0; i < numbers.Count; i++)
{
    Console.WriteLine(numbers[i] * 2);
}
```

Figure 2.4: For-loop

```
static void DoubleAndLog(int number)
{
    Console.WriteLine(number * 2);
}

List<int> numbers = new List<int> { 1, 2, 3, 4, 5 };
numbers.ForEach(DoubleAndLog);
```

Figure 2.5: For-each loop implementing Visitor pattern

2.7.1 The Observer Pattern

One of the first design patterns introduced in the aforementioned book by the "Gang of Four", the Observer pattern is a behavioral pattern that addresses several different key problems in object-oriented programming. It is implemented by creating two separate interfaces: the publisher who is responsible for publishing events of interest for the rest of the system, and subscribers who are interested in knowing when the publisher has published such an event. Implementing these interfaces allows the system to achieve loose coupling, as the publisher and subscribers can evolve independently, promoting a maintainable and adaptable system. Developers can easily introduce new subscribers with minimal modifications. Furthermore, the pattern facilitates dynamic relationships between scripts that can change at runtime by having the subscribers unsubscribe from the publisher. This, accompanied by the publisher's event broadcasting ability, ensures that the entire system remains in a consistent and traceable state.

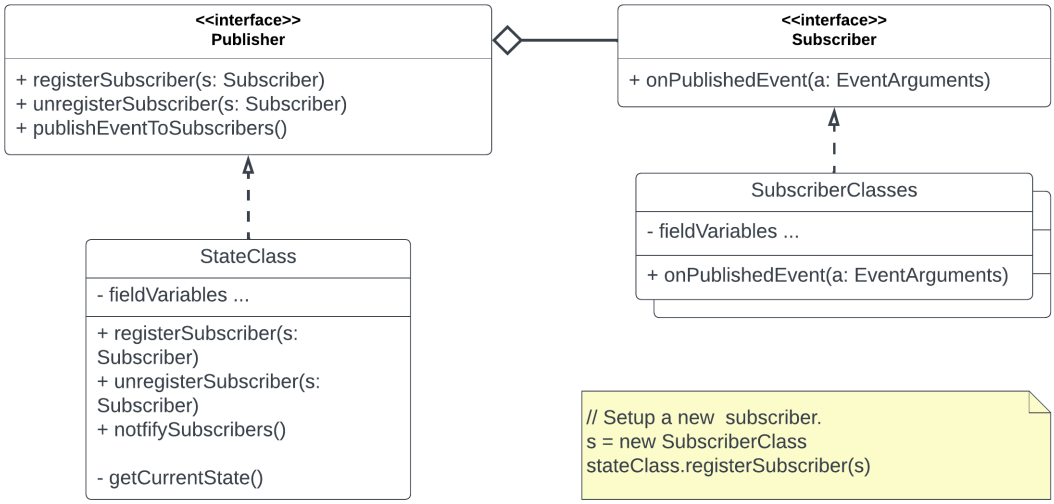

Figure 2.6: UML diagram of generic publisher/subscriber-relation implementation.

2.7.2 Overview of the Observer Pattern in Unity and C#

Both the C# programming language and the Unity API includes several features that facilitates the use of the Observer pattern. Most modern languages opt for a method-based implementation of the Observer Pattern, as compared to the class-based implementation as seen in figure 2.7. The list that follows highlights some of the most prominent components that are used to promote the usage of this pattern:

- **Delegates (C#)**: Define a type representing a specific user-defined method signature, allowing for a method-based event system. This custom type can be realized with any method sharing the same signature as the delegate, enabling a type-safe way of passing methods as arguments. They resemble C++ func-

tion pointers but provide object-oriented capabilities by encompassing both a function and its associated object instance.

- **Event Handlers (C#)**: Methods defined in the subscriber class that conform to a specific delegate signature, typically a delegate with two parameters: one object representing the publisher and one event data object containing event-specific information. These event handlers are responsible for processing incoming events and performing any necessary actions when the triggering event is published.
- **Events (C#)**: A convenient feature in C# built upon the foundation of delegates. They provide an easy way to define, subscribe, and publish events in C#. Publishers define the event, while subscribers can subscribe or unsubscribe using the event handler. Events enforce encapsulation by allowing only the class that owns them to publish them while still enabling other classes to subscribe or unsubscribe at run-time.
- **UnityEvents (Unity)**: A built-in event class offering a flexible and powerful way to facilitate event-driven systems that can be configured in a user-friendly way through the Unity editor. Being serializable, they can easily be set up and managed through the editor using a drag-and-drop approach within the editor.

2.7.3 Singleton

The Singleton pattern is a creational pattern that ensures only one instance of a specific class can be created at any given time, providing global access points to that instance. The pattern has garnered criticism for violating core object-oriented principles, such as The Single Responsibility Principle[2], promoting tight coupling, and making testing more difficult due to challenges in isolating tests, replacing instances with mocks, and managing shared global state. However, some argue for its responsible usage, applied only to classes that genuinely require a single instance and where global access is necessary. It is crucial to manage dependencies and shared states carefully to minimize the risk of creating hard-to-maintain, tightly-coupled code. Common use cases for the Singleton pattern in game development include managing access to different manager classes, such as managers for input, audio, or pooling.

The pattern is implemented by having a private static field in the singleton class for storing the instance of the class. This instance is instantiated through a public static creation method, which uses "lazy initialization" to create a new singleton object instance through a private constructor if it is the first time the instance is being called, or returns the pre-existing instance otherwise. To ensure thread-safety in multi-threaded applications, a locking mechanism can be implemented to prevent multiple threads from creating separate instances simultaneously. This

[2]The 'S' in SOLID. States that a class should only have one responsibility, promoting good separation of concern and modularity.

can be achieved using the "double-checked locking" pattern, where the lock is only acquired if the instance is null, reducing the performance overhead of locking in cases when the instance is already created.

The pattern is implemented by having a private static field in the singleton class for storing the instance of the class. This instance is instantiated by having a public static creation method, that through "lazy initialization creates the new singleton object instance through a private constructor if it is the first time the instance is being called, and if not, simply returns the pre-existing instance. To ensure thread-safety in multi-threaded applications, a locking mechanism can be implemented to prevent multiple threads from creating separate instances simultaneously. This can be achieved by using the "double-checked locking" pattern, where the lock is only acquired if the instance is null, reducing the performance overhead of locking in cases when the instance is already created.

2.7.4 State

The State pattern is a behavioral pattern that creates a modular and extensible system architecture for managing transition between different object states. The pattern does so by decoupling the logic for each possible state into a separate interface that a main class then manages by offering methods for interacting with different state objects and delegating any necessary command to the current state when told to. This main class can be described as mediator between different states, and offers the developers an user-friendly way of managing state actions and transitions.

The pattern was first introduced by the aforementioned book "Design Patterns: Elements of Reusable Object-Oriented Software", and draws its inspiration from the concept of finite state machines (FSM) which are computational model used across a wide spectrum of domains such as control systems and artificial intelligence. A FSM consists of a finite amount of states, the initial state, and the adhering transitions between them. While both FSM and the State pattern deals with managing system behavior through various states and transition, FSM is a more general concept that focuses on the overall structure of a system, while the State pattern is a specific object-oriented concept focusing on adhering to good object-oriented principles.

By seperating the state logic into a seperate interface, this makes it easy to accommodate for change and extensibility when modifying or adding a new state. This adheres to the Open/Closed principle[3] as it allows developers to introduce new states without altering the existing state classes or the main class responsible for managing state transitions.

The pattern is implemented by defining the common State interface, that should be able to handle any state specific requests and transitions. This interface is then realized by concrete states classes that provides their own unique logic and behaviour.

[3]The 'O' in SOLID. Says that a class should be easy to extend without needing to modify any existing code.

To mediate between these states, you then implement a main class, sometimes called the Context class, that holds a reference to the current state and delegates function calls to it. This main class is also responsible for changing the current state based on transition logic defined in the concrete State classes.

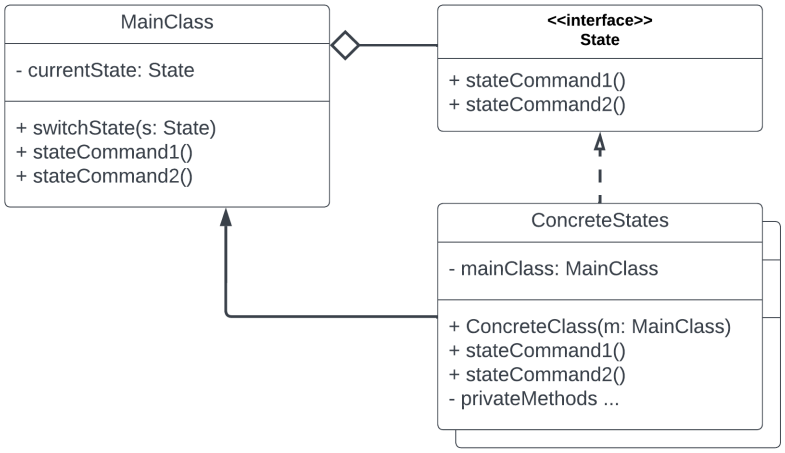

Figure 2.7: UML diagram of generic State pattern implementation.

- https://www.amazon.com/-/dp/0195019199
- Design Patterns: Elements of Reusable Object-Oriented Software
- https://gameprogrammingpatterns.com/observer.html

2.8 OpenStreetMap

OpenStreetMap is a free database which contains geographical data. The database is maintained and updated by volunteers. Volunteers can collect data about geographical areas and add to the database for everyone to use. Data about features such as roads, railroads, buildings, trees, etc and their properties exist.

3

Methods

3.1 Tools

3.1.1 Unity

The traffic simulation tool is built in a well-known game-engine called Unity. There are a few reason why it was chosen as the development platform for the project instead of a similar game-engine like Unreal Engine. To begin with, C# is the main programming language supported by Unity, which some of the team members had previous experience with. Furthermore, C# is a higher level language compared to C++, the main language of Unreal Engine, making it easier for the team members without experience to learn. Because of this, the time it took to begin programming in the early stages of the project was most likely shorter, compared to if Unreal Engine was chosen as the platform.

Another reason would be that Unity comes with the Unity Asset Store, a marketplace for acquiring creator made assets. This feature is important because, for example, instead of having to create custom models for the vehicles, they could instead be purchased using the given budget. This saves a lot of time, that could be better spent on other parts of the project. One of the more notable purchased assets is Edy's Vehicle Physics that are used to rig vehicle models with realistic physics. Instead of having to develop custom vehicle physics for each model, the team could instead use the asset to quickly configure a model with physics.

The final reason why Unity was chosen, is because of its flexible developing structure. The level of customization available inside the engine is a lot greater when comparing to Unreal Engine. However, because of this, Unity ends up being more unstable whereas Unreal is far more stable and robust.

3.1.2 GitHub

A commonly used tool when developing software in larger groups is Git. Git is a free and open-source version control system that allows its users to collaborate in a efficient and easy way.

GitHub is an online software development platform that utilizes Git to store and track software projects. It allows for users to work in their own separate branches,

and later merge those into the main repository. Before a team member could merge their new code to the main repository, the code would have to be reviewed by at least one other team member to ensure that the code was well commented, functional, and that it follow the C# coding standard.

3.1.3 Trello

It was decided early on that the projects work flow should follow the SCRUM and Agile software development practices. Trello is a website that hosts scrum-boards in an user-friendly way. This allowed the team to keep track of what needs to be worked on in the project during the sprints. A sprint is a set time period when new tickets are made, and completed.

3.1.4 Balsamiq Wireframes

During the first stage of creating a UI, its important to start with a simple mock-up design. This is what the tool, Balsamiq Wireframes, is used for. The user can quickly design wire frames depicting how the UI will appear during different times in the program. This includes everything from buttons to pop-up menu's that might appear in the simulation tool.

3.1.5 Third-Party Assets

Built into Unity is their asset store. Instead of creating everything from scratch, the team opted to purchase some assets. An asset can be anything from a 3D model to animation and scripts. The two main assets purchased for the project are Edy's Vehicle Physics and European Road Signs. Edy's Vehicle Physics is a package that includes a tool that allows its user to easily implement realistic vehicle physics into 3d models. This saves a substantial amount of time in the end because there would be no need to create custom physics attribute for each vehicle model.

As the name states, the European Road Signs assets include a plethora of street signs, as well as an editor to customize them. Without this asset, there would have been a need to create custom 3D models and texture, which no team member had previous experience with.

3.2 Simulation Design and Implementation

3.2.1 ABM

3.2.2 Road Generation

In order to achieve realistic roads with adequate curves, composite Bézier curves were used. The Bézier control points will shape the road and its characteristics. A number of parameter can be changed in the Bézier path to change the appearance. The position and sharpness of the turn can be modified by changing were the control points are placed in relation to each other.

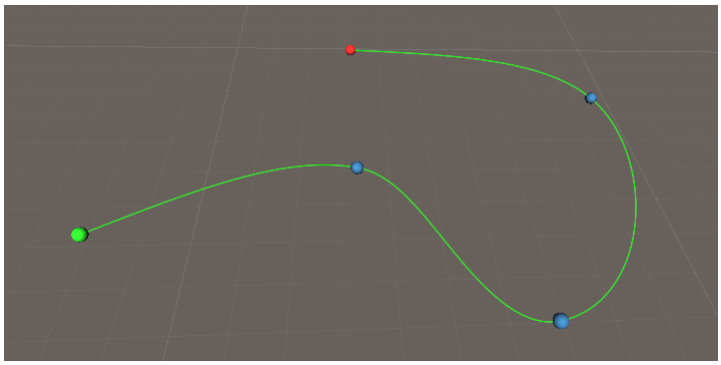

Figure 3.1: Composite Bézier path

While the Bézier curves give a good ground level for the road implementation, it is hard to implement and build logic based on it since it is a continuous path. The Bézier logic and its control points was abstracted away with a node implementation placed on top of the Bézier path. A number of nodes called RoadNodes is placed along the Bézier path at a rate dependent on curve of the road. The nodes are all connected the its previous and its next node along the path. The goal of these nodes is to carry enough information to procedurally build the road mesh as well as carry some logic needed for agents to navigate the environment.

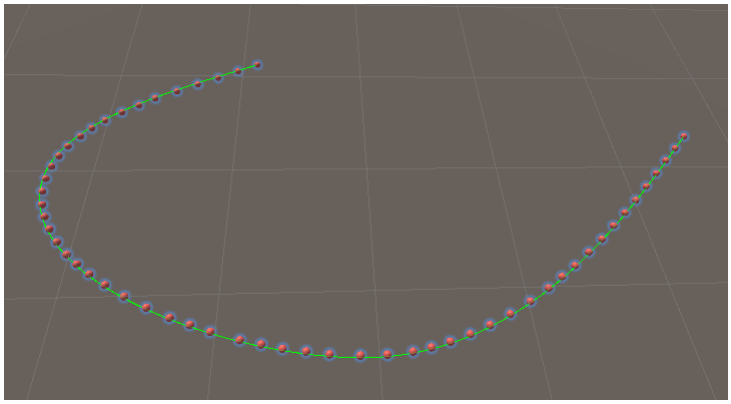

Figure 3.2: Visual representation of RoadNodes places along a Composite Bézier path

By using the RoadNodes, generating the road mesh is possible. The mesh is procedurally generated by placing vertices at the RoadNode and along its normal line in both direction at a length equal to the width of a lane. If multiple lanes is wanted, vertices can be continually added in each normal direction for the lane amount. In addition to these, vertices are also placed below them to add thickness to the road. Triangles are then drawn between these vertices to create the mesh. To add the road material, sub meshes is created for each lane. The power of procedural generation is the ability to customize the roads different parameters. The width of the lanes, width of the lines, thickness of the road and number of lanes can all be changed for each road.

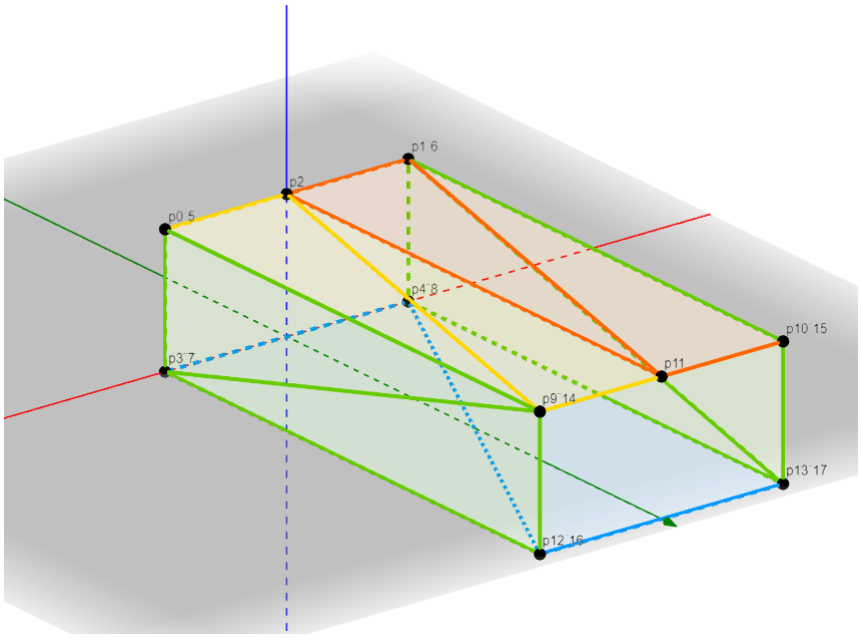

Figure 3.3: The vertices and triangles used to generate the one lane road mesh

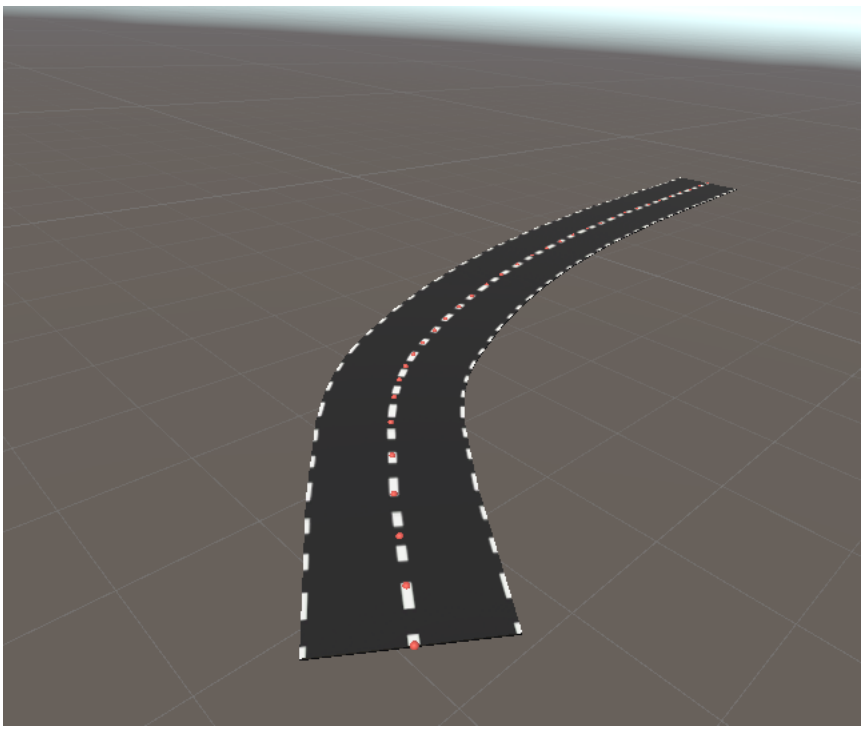

Figure 3.4: Road generated from its RoadNodes

While the RoadNodes carry a lot of critical logic, it is lacking logic for driving along the different road lanes. This logic is added with another node that is placed along the road, the LaneNode. LaneNodes are placed at both sides of the normal line of each RoadNode at the middle of the lanes. These nodes are responsible for the

road steering as well as notifying other agents if they are currently occupied by any agents.

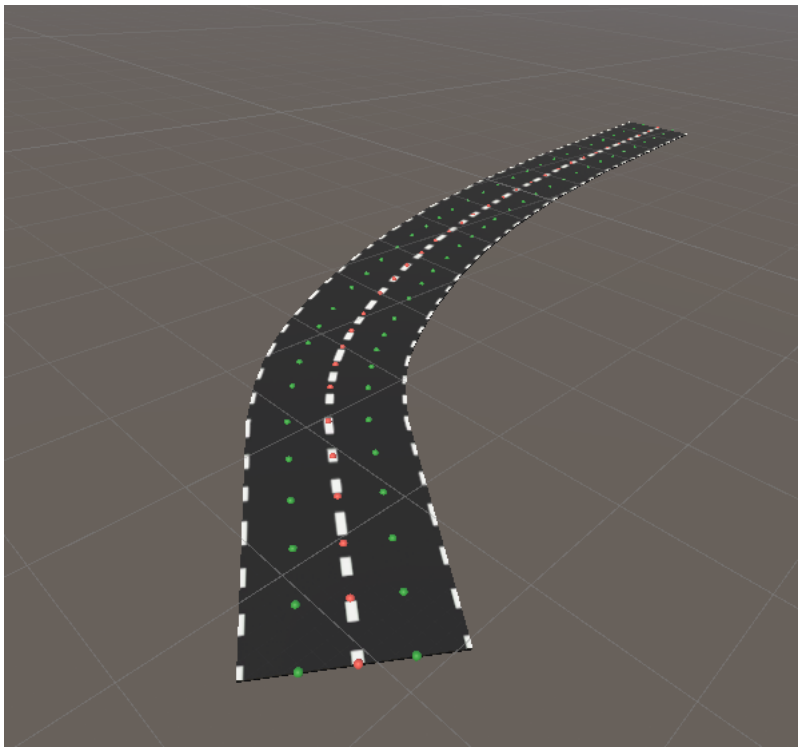

Figure 3.5: Visual representation of RoadNodes(Red) and LaneNodes(Green)

3.2.3 Intersection Generation

3.2.4 Navigation in intersections

3.2.5 City Generation

To aid in simulating cities and larger areas, existing real life locations are generated from OSM data. The OSM data specify the latitude and longitude of every road and its path. Real life building data is also used to generate a representation of the buildings. The OSM file is parsed and the roads are generated with its specified characteristics. The speed limit, road type and if the road is lit is all considered when generating the road.

3.2.6 Navigation

The basic navigational responsibility of each agent is the ability to follow the road lanes, avoid colliding into other agents, follow the traffic rules and being able to navigate to a given position.

In order to follow the lanes, each agent follow the LaneNodes on the road. The LaneNodes store all the information needed to navigate the road. The position of the node, the agent that is currently on the node and special traffic rules the vehicles

need to follow are stored on the node. Traffic signs such as stop sign are represented as a node and the traffic logic can be accessed by the agent when they encounter the node.

The agents steer towards a node that is a certain distance in front of the car. This distance is influenced by the current speed. By steering towards nodes that are in front of the agent, a smooth and reliable steering is achieved. Similarly to steering, breaking is accomplished by looking at nodes at a certain distance ahead. When a node with stop logic is found, the agent will break and stop before that node. This is done by looking for stop nodes at a distance ahead equal to the break distance of the agent. The agents also claim each node they are over so other agents can stop when the node at the break distance is claimed.

To enable the ability to navigate the roads, a weighted directed graph is created from the roads. The graph nodes are all the road endpoints and intersections and POIs. The edges between the nodes are weighted with a cost that is calculated as the distance * the speed limit. The agents navigate to a given end node by receiving a path of edges from the A* algorithm. When an agent doesn't have an active navigation path, it will be assigned path that will lead it to a given destination. The agent maps out the path and saves instructions for were to turn in each intersection that is passed when navigating to the destination.

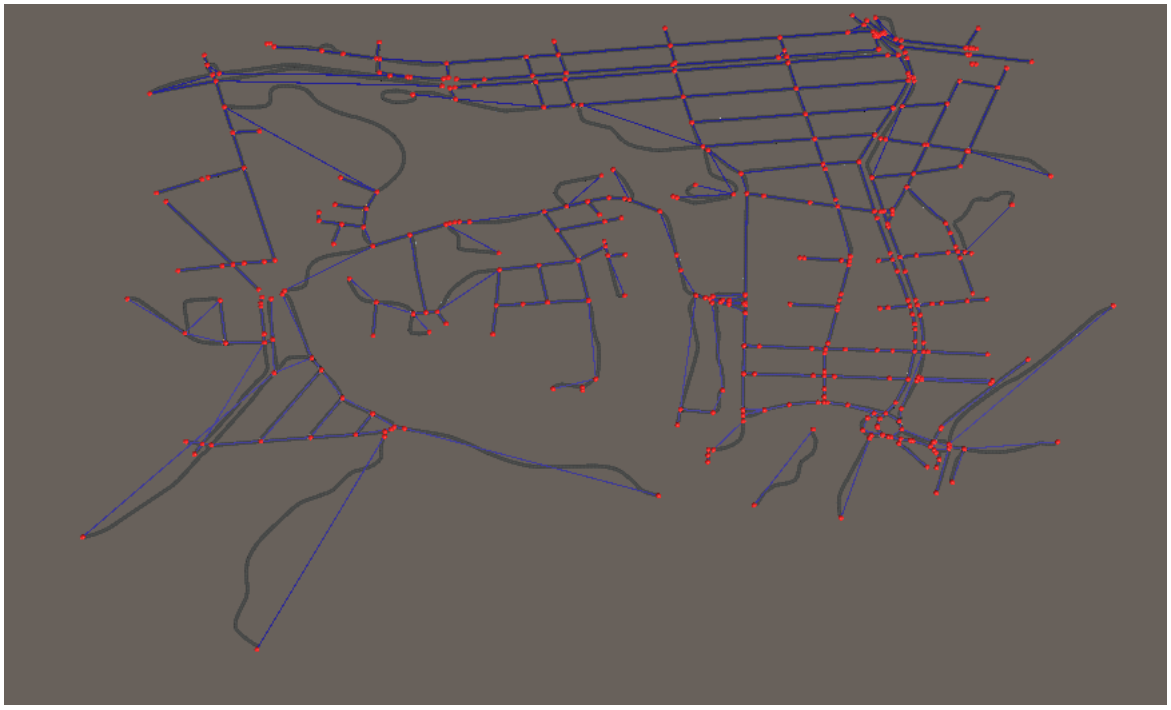

Figure 3.6: Visual representation of a navigation graph

3.3 Performance

3.3.1 Quality vs Performance

An important aspect of software is how well it runs. Therefore, it was decided early on that the functionality to change the quality level of the simulation should exist.

While testing the simulation during development, the most noticeable performance cost were the vehicles on the roads. This is because of the Edy's Vehicle Physics asset that simulates real-world physics to each vehicle in the network. To circumvent this issue, a vehicle performance mode was implemented. This performance mode would disable the EVP asset, and instead move the cars by offsetting their individual object transform. As a result, the performance cost of the vehicles would decrease, and allow for more cars in the road network.

3.3.2 Performance Benchmarks

3.3.3 Optimization

When creating software of any kind, it's always important to make sure it's able to run smoothly. To allow the program to run well, optimizations had to be made. There are two areas in the simulation that are costly performance-wise: the vehicles on the roads and the roads themselves. To optimise the vehicles, as mentioned earlier, a performance mode was implemented. This allowed the simulation to skip calculating the physics for each vehicle.

Furthermore, since the simulation is in 3d, the details of the vehicle models had to be accounted for. A 3d model is created with vertices, which are points in a 3d dimensional space. Three of these points are used to construct a triangle, and the triangles in turn build the model. The amount of triangles in a model determines the overall detail of said model. To improve the performance of the simulation, the models that were chosen most contain a low count of triangle, usually less than 20,000.

For the road network,

3.4 Graphics

3.4.1 Animations

3.4.2 Environment Materials and Textures

3.5 User Interface

3.5.1 Design

To allow the user to interact with the simulation, an user interface was made. The UI consists of two main parts: the start menu and the overlay.

The start menu contains three buttons, one to start the simulation, one to enter the settings menu, and one to exit the program. The settings menu allows the user to change the volume, quality mode, enable a fps counter, and enter full screen.

The overlay that is visible while the simulation runs is what allows the user to interact with the simulation itself. There are two main types of buttons: the camera buttons and the menu buttons.

3.5.2 Statistics

3.6 Work flow

When developing any software larger than just a single use script, the amount of work and information can quickly grow beyond the level of ones own simultaneous comprehension. Therefore these kinds of projects require rigorous planning and strategizing to not get lost in all the different tasks and do them in a smooth and reasonable order, that allows for parallel continuous progress.

To achieve this a strict work flow framework was developed, where the first step was to analyze the work load and disposable time. This included drafting a time plan for the whole time scope of the project 3.7.

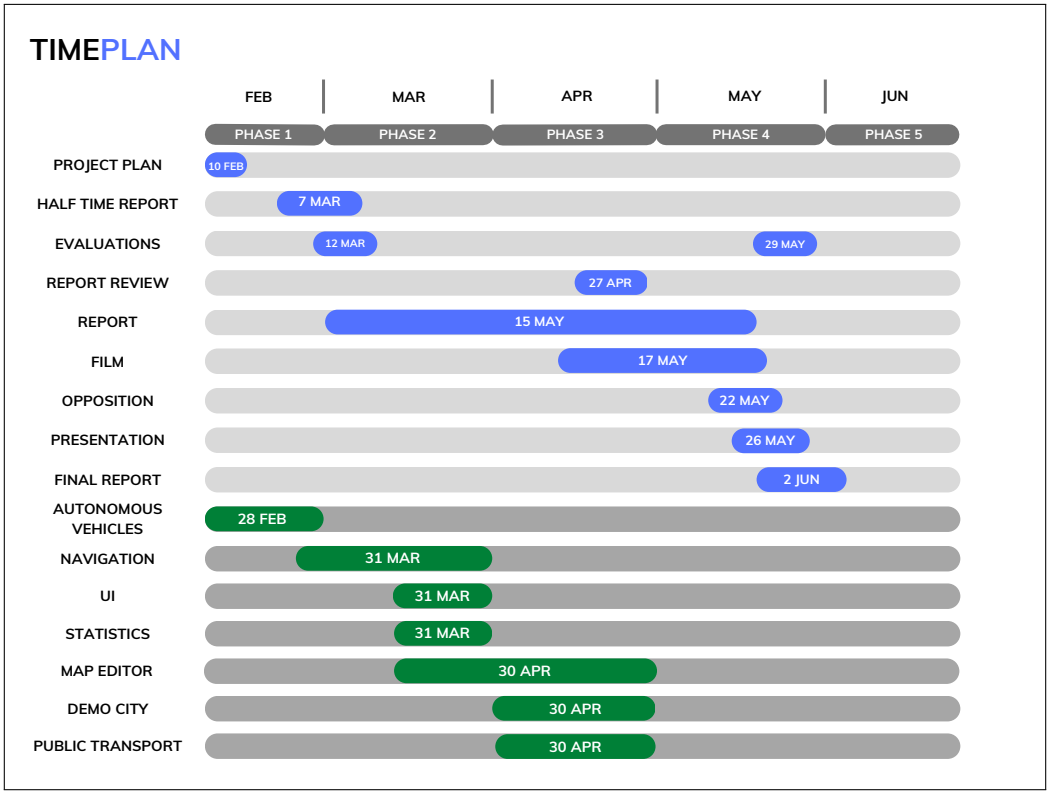

Figure 3.7: Project Time Plan

With this it's now much easier to keep track of the general progress of the project, as well as helping with planning short term goals. Coincidentally this is the next step of the work flow model. The short term goals where planned using a scrum framework with weekly sprints, explained in 3.6.1. These sprints were upheld for the duration of the project to keep a stead flow of progress, together with the time plan they create a very clear way of seeing the current state of completeness.

The third aspect of the work flow is the approval of progress. As mentioned earlier a large project requires a substantial amount of planning to not get lost. The approval of progress can be seen as just as important as the planning and execution itself. Without a popper method of approving new advancements/functionality, the project can quickly falter. If progress never goes through the process of approval many things can go wrong. Evidently, badly written code can cause issues that are easily preventable with a quick inspection. Code can even be considered good but with no input from the rest of the team, visions of how higher order elements will be implemented can differ. This can implicitly create more complex problems much further on, which can be very hard and time consuming to resolve. To solve this, code reviews 3.6.2 for every change made are part of the workflow.

3.6.1 Weekly Sprints

The weekly sprint model stems from the scrum framework, which is a framework for developing and sustaining complex products. The sprint model follows 4 repeating

stages of development: Planning, Implementation, Review and Retrospect.

Each sprint starts out in the planning stage, where a meeting is held to set up this sprints goals. This includes moving/creating stories for the backlog as well as the current sprint. The stories are mainly chosen by the project manager then developed in unison with the scrum master and input from the rest of the team.

The next stage of the sprint is the implementation itself. This is the time were the teams focus is solely on delivering good quality solutions to complete all of the current sprints stories, and eventually working on the backlog as time is presented.

Next up is the review stage, not to be confused with code reviewing 3.6.2. In this stage another meeting is held called a "Demo meeting", where all members get to do a small demonstration of all their progress during the sprint. This is an important step to onboard all members on new functionality and make sure that desired behaviour is achieved. When a story is regarded as fully complete it's archived to make room for new ones.

Lastly the retrospect stage, which is usually carried out following the review stage. In the retrospect stage the current sprints efficiency and quality is discussed. And plans/ways to increase these and the overall effectiveness are considered. When all is done the cycle begins anew until the project is done.

3.6.2 Code Reviewing

3.7 Testing

4

Results

Text ...

5

Conclusion

Text ...

Bibliography

[1] M. Mora-Cantallops, S. Sánchez-Alonso, E. García-Barriocanal, and M.-A. Sicilia, "Traceability for trustworthy ai: A review of models and tools," *Big Data and Cognitive Computing*, vol. 5, no. 2, 2021. [Online]. Available: https://www.mdpi.com/2504-2289/5/2/20

[2] A. A. Shavez Kaleem, "Cubic bézier curves," 2000. [Online]. Available: https://mse.redwoods.edu/darnold/math45/laproj/Fall2000/AlShav/bezier-dave.pdf

[3] T. Sederberg, "Computer aided geometric design," 2012. [Online]. Available: https://scholarsarchive.byu.edu/cgi/viewcontent.cgi?article=1000&context=facpub

[4] S. V. Konakalla, "A star algorithm," 2014. [Online]. Available: http://cs.indstate.edu/~skonakalla/paper.pdf

A

Appendix 1

This is where we will place appendix 1

B

Appendix 2

This is where we will place appendix 2