Development of Traffic Simulation in a Game Environment

Entwicklung einer Verkehrssimulation in einer Spielumgebung

Master-Thesis von Jan Henno Lauinger aus Erbach (i. Odw.)

Tag der Einreichung:

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Quoc Hien Dang

TECHNISCHE UNIVERSITÄT DARMSTADT

Fachbereich Informatik
Knowledge Engineering

LIMBIC Entertainment

Development of Traffic Simulation in a Game Environment
Entwicklung einer Verkehrssimulation in einer Spielumgebung

Vorgelegte Master-Thesis von Jan Henno Lauinger aus Erbach (i. Odw.)

1. Gutachten: Prof. Dr. Johannes Fürnkranz
2. Gutachten: Quoc Hien Dang

Tag der Einreichung:

Ehrenwörtliche Erklärung

Erklärung zur Abschlussarbeit gemäß § 22 Abs. 7 APB der TU Darmstadt

Hiermit versichere ich, Jan Henno Lauinger, die vorliegende Master-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.
In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Datum: Unterschrift:

Abstract

By implementing a prototype traffic simulation in a current game engine, comparisons between different car-following models can be made and sources of long computation times identified. Vehicle recognition algorithms which provide parameters for simulation models are developed, optimized and analyzed for further optimizations. A path finding algorithm is enhanced to find faster paths in traffic.

The simulation framework, game engine characteristics and redundant or abundantly frequent calculations are identified as sources of bad performance instead of suboptimal simulation models. An alternative to lane-changing models is given which improves performance significantly. Further possibilities for improvements are given to be able to simulate tens of thousands of vehicles in a real-time game environment.

Contents

1 Introduction

Computer games, and video games in particular, are challenging to develop because of real-time requirements and the resources they consume. Game logic and graphics rendering, for example, must be computed "in parallel" without one part slowing down the other. Simulation games can use a lot of computing power to simulate complete economic systems. The simulation discussed in this thesis is intended for city-builder games, which simulate peoples' lives in a city and traffic on its roads as well as its economics. The need for games to simulate larger cities with more citizens and more vehicles in larger road networks calls for a traffic simulation that is being optimized to *look* realistic, such as vehicles' acceleration behavior, while being performant enough to allow other components of the game to be computed without a slowdown.

The kind of traffic simulation used in this thesis demands solving multiple problems for it to work properly: There needs to be an acceleration function for vehicles, which usually includes avoiding collisions with a leading vehicle; lane-changes and turns should only be made if there is no collision with other vehicles; a set of traffic rules has to be defined and path finding needs to be adapted to the simulation's needs and to traffic rules. Related work on car-following [5, 7, 11], lane-changing models [7] and on simulation of peoples' lives [6, 15] solve the mathematical problems and provide a model equation as a result. They mainly focus on finding bottlenecks in road networks, optimizing traffic flow in a city or on a highway as well as on replicating reality as close as possible. Existing literature does not focus on how to implement a well performing simulation that is suited for real-time environments, such as games. It often makes assumptions that may appear trivial on the first read but entail lots of computing time and implementation work later on, so that they can be used in a real-time setting.

The goal of this thesis was to implement a prototype simulation in Unreal Engine 4, measure frame times and identify a traffic simulation's components that would be best to optimize for better performance. *Unreal Engine 4*[1] (UE4) is developed by Epic Games[2] and freely available for non-commercial use. Being used commercially to develop games, Unreal Engine 4 has the benefits of being very close to the intended application context of the simulation, and not having to implement a large scale framework which is necessarily needed for a game. Its low-level programming language C++ combined with an Application Programming Interface[3] (API) and the source code[4] of the engine being provided gives a high level of control over implementing and optimizing the simulation.

Requirements for the simulation were being able to simulate at least 1000 vehicles with a frame rate of at least 30 FPS (frames per second), so graphics can be rendered smoothly, whereas higher frame rates would be preferred as they leave enough space for other parts of a game. Optimizations were originally thought to be made by choosing the best performing model, implementing it with attention to its performance and developing an equation that would perform better. After a first draft was tested on different road networks with different properties the highest potential for optimizations was found to be in vehicle recognition and visualization parts of the simulation. For implementing the simulation two algorithms for vehicle recognition were developed with regards to their performance trying to minimize redundant calculations.

Simulating traffic in city scenarios leads to traffic jams and gridlocks at high vehicle densities. While traffic simulations in related work try to identify the bottlenecks that lead to traffic jams and in video

1 https://www.unrealengine.com/
2 http://www.epicgames.com
3 https://docs.unrealengine.com/latest/INT/
4 https://github.com/EpicGames/UnrealEngine

games those places usually have to be spotted and corrected by the player, this thesis will propose an enhancement to Dijkstra's algorithm that avoids traffic jams and gridlocks and allows more vehicles to reach their destination.

In summary the contributions that are being made by this thesis are:

- Analyzing a prototype traffic simulation for components that can be optimized with regard to computation time.
- Providing algorithms for the recognition of leading and intercepting vehicles which may appear trivial in model descriptions.
- Proposing an alternative to lane-changing models at intersections without traffic rules.
- Enhancing Dijkstra's path finding algorithm to avoid traffic jams and gridlocks and to ensure more vehicles reaching their destinations faster.
- Identifying and proposing further optimizations depending on the simulation's usage.

2 Requirements for the Simulation

This chapter will describe the context in which the findings will be applied and the requirements that are given by this context.

2.1 Context

This thesis was conducted in collaboration with Limbic Entertainment GmbH[1], which is a game studio located in Langen. A recent change of the game engine to Unreal Engine 4 developed by Epic Games allowed development of up-to-date games with cutting edge technology but likewise posed new problems to be researched. The question that inspired this thesis was how many vehicles could be simulated in a 3D real-time game environment while keeping an acceptable frame rate of at least 30 FPS. The requirements on the simulation's properties and behavior are on the one hand governed by being applicable to city-builder games in general, but also by being able to be adopted to different kinds of settings and possibly other genres too, which results in partly specific, partly general requirements and rules or lack thereof.

2.2 Requirements for the Model

The minimum requirements for the model are set partly by the fact that it should be able to simulate traffic in a real-time video game environment, partly by the desire for multiplayer capabilities and partly by choice, for example the game genre it is intended for. Genre and choices are given by the context the simulation is intended for at Limbic Entertainment.

The model that will be developed in this thesis is intended to simulate traffic in a city-builder game with strong focus on economy. A city-builder is a game in which the player usually iteratively builds a city, starting with a few and ending – if at all – with a few hundred or up to thousand buildings. The cities are made up of housing, work places like office buildings, infrastructure like police stations, fire stations and hospitals, entertainment buildings like bars or bowling alleys, shopping areas, factories, power plants, parks and many more. Every building is usually connected to a road so that people can reach its entrance.

There are two categories of requirements for the model that will be described in the following sections: The model's properties which are mainly affected by choice of the genre, and its desired performance that is governed by being only one part of a video game and thus having to share resources with other parts, like game-play calculations, user input, rendering of graphics etc.

2.2.1 Properties

The model will simulate traffic in an economy driven city-builder. This determines a lot of the model's properties, which are explained in this section but still leaves room for choice for example in vehicles' properties like maximum velocity, constraints on how the road network can be built and a set of traffic rules or the lack thereof. Deciding on those options will be governed by the games that will use the simulation, maybe even change while playing. The focus on economy means that resources produced

[1] http://www.limbic-games.de

in one factory will be transported to another factory to be processed further. Transport duration of each resource influences the production of its derivatives, since the process will only be started when all resources are available at the factory. This leads to the need for each vehicle to be simulated individually. Vehicles either transport resources or people from one building to another, thus buildings mark the locations of any vehicle's origin or destination for path finding. To ensure preferably short travel times deviations from the shortest path are only to be taken if current traffic allows them to be quicker. Vehicles cannot be deleted if they do not reach their destination or if it improves performance because transported resources or citizens would be missing from the simulation. Vehicles should try to avoid collisions, therefore they need to keep a safe distance to their leading vehicle and look out for other vehicles they might collide with at intersections. Gridlocks have to be avoided, especially when multiple vehicles would decide to brake for each other at an intersection. There is no intention to support different driving behaviors like aggressive or defensive driving. Vehicles have a given set of properties like maximum velocity and maximum acceleration, which may vary from one vehicle type to another, though.

For the city being able to expand in size it should be able to contain thousands of citizens, each of them being potential road users. At peak times the simulation has to be able to simulate most of the road users simultaneously. The road network consists of buildings and intersections connected by roads. All roads have two lanes, one in each direction. Lanes of dead end roads are connected at the end so vehicles can turn around. Road users are bound to their lanes and cannot overtake other vehicles. There are no different types of roads and no obstructions on the roads like building sites or parking cars. Every vehicle is able to find a parking space inside the building at its destination. As far as the simulation is concerned all roads are straight and flat, which means there is no maximum or safe velocity depending on a vehicle's location on the road. Locations and distances on roads are continuous and can have any positive real value. Locations of intersections and therefore roads can be anywhere in 3D space and are not bound by cells or grid tiles. The visual representation of roads in the game can have curves, for example by using splines. The simulation will still be valid if locations of intersections are limited to a grid later on, to simplify road editing. Visual representations of vehicles will follow the visual representations of roads based on the simulated data. Vehicles can only enter roads or leave them by exiting or entering a building. Speed limits are not defined by roads but rather by the vehicles. There are no traffic lights at intersections and no set right of way.

The simulation should be designed having multiplayer capabilities in mind. Since the size of the simulation's data structures is limited by data transfer rates to other machines, data structures with small memory footprints have to be preferred where objects and properties are transferred directly over the network. When transferred values of properties are used to recreate objects and their states on another machine truly random values cannot be used in the calculations that recreate the object or its state. This could lead to the same object being different on different machines, for example if two vehicles at an intersection decide randomly which one can go first a different outcome on two machines would have two players see completely different scenarios. Simulation time or steps also must not depend on render or computation time because no two computers need the exact same amount of time to render every frame. Therefore a simulation loop with fixed time steps has to be introduced so traffic is simulated independent of render times.

2.2.2 Performance

Video games have to render graphics, handle user input and compute game logic, including traffic simulation. All subsystems have to work simultaneously, therefore computing power has to be shared amongst them. This means that the simulation needs to be optimized to be able to run in a real-time environment without using all computing power on its own. Usually, video games render at least at thirty to sixty frames per second (FPS), which corresponds to (16.66 – 33.34) ms per frame. On the one hand, this gives a time for simulation steps so that vehicle movement looks fluid, on the other hand it restricts the

order of magnitude in which each simulation step has to be executed. All simulation calculations, including visualization should not take longer than 33.34 ms per frame. Faster calculations will be preferred as they make smoother rendering possible and allow other components of the game to be computed as well.

With regard to a video game environment, the traffic simulation does not need to represent reality completely, like simulations for studying real traffic flow do but is rather bound by the properties above and the demand to at least *look* real. When video games cannot compute all necessary information in the available amount of time, or the logic cannot be optimized further or just would not be computable if it was recreating reality very closely, they usually simplify in one way or the other, knowingly deviating from reality. For example, collisions of objects are not calculated with the real bounds of the object, but rather with a bounding box or sphere to accelerate computations. In traffic simulation, this often means that vehicles are teleported when they are stuck or are only simulated when visible. The requirement for realistic-looking but not necessarily accurate simulation affects the simulation model which does not need to reproduce real speed-density-diagrams as long as the vehicles' movement looks plausible. It also leaves some room for deciding upon the properties of the simulation, for example the choice of roads and traffic rules.

3 Theoretical Foundation

This chapter provides the background necessary to understand all topics related to this thesis. Section 3.1 describes general usage of traffic simulations, classifications by the simulation's aggregation level, and gives mathematical descriptions of the compared models. Section 3.2 explains programming patterns that are used to optimize the simulation.

3.1 Traffic Simulation

Traffic simulation is generally used to simulate real traffic on a computer. The simulation's results can be used to find bottlenecks in the road network of a city, sources for traffic jams, and to show how closely a model simulates reality [see 16].

3.1.1 Classification

Traffic flow models can be classified by their mathematical structure or conceptional aspects [16, pp. 52-57]. This section will focus on a classification by the model's aggregation level, which means at which level a simulation object will be located. A very high level could be described by there being lots of traffic in a city at day and less at night. A very low level model could consider the thought process of a driver, his reaction time given his current fitness, and maybe even the force needed to press a pedal.

Macroscopic models aggregate traffic information at a given location on the road at a given time [16, p. 52]. Typical variables are vehicle densities, average velocities, traffic flow or pollution. The fundamental diagram displays traffic flow over vehicle density. It is often used to compare the models' simulation results to data measured in real traffic to show general behavior of the model. It displays how well traffic flows at different vehicle densities. Average velocities can be displayed instead of traffic flow as well. A velocity-density diagram can be seen in Figure 3.1.

At a *microscopic* level, movement of vehicles is simulated [16, pp. 53-54]. This means a microscopic simulation is not only able to make statements about traffic in itself, but also about single vehicles in traffic, for example their location and velocity at a given time, acceleration, leading vehicles, routes, or

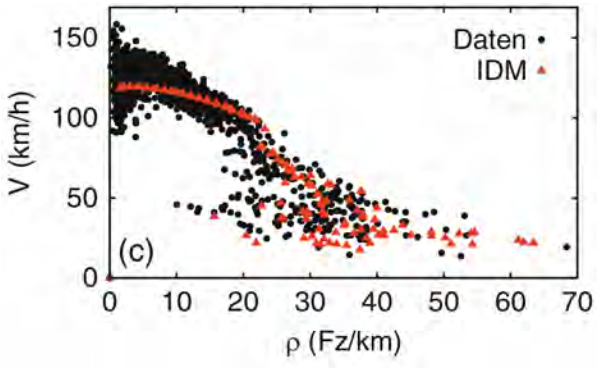
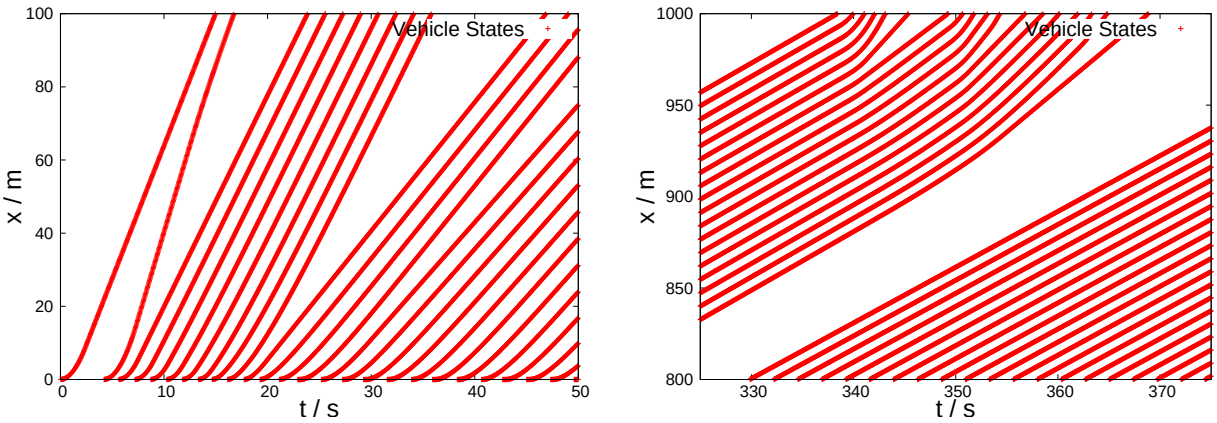


Figure 3.1.: Velocity-density-diagram of traffic on a motorway taken from [16, p. 167]. Simulated data (orange) is compared to measured data (black).



(a) Trajectories of vehicles accelerating at the beginning of a road.

(b) Trajectories of vehicles at the end of a road.

Figure 3.2.: Each data point in the graphs represents the location of a vehicle at a given time, the lines being formed by those points are the vehicles' trajectories. Vehicles are spawned with no velocity and accelerate until there is a slower vehicle in front of them or until they reach their maximum velocity. Vehicles that reach the end of the road are deleted, which results in the following vehicles to accelerate again. Parallel sets of trajectories are convoys of vehicles being slowed down by the front-most vehicle.

even fuel usage. This data can be plotted as location over time, where each trajectory, for example a line represents a vehicle. Microscopic models can be further divided into car-following or acceleration models (see Section 3.1.3), and lane-changing models (see Section 3.1.4). Results from microscopic simulations can be used to obtain macroscopic values like traffic flow and vehicle density. This category is useful if vehicles have to be differentiated, for example by size, maximum velocity, driving behavior or if routes have to be analyzed.

Mesoscopic models are a mix of macroscopic and microscopic approaches [16, p. 54] [see 14]. Microscopic functions could be used to control macroscopic model parameters, microscopic model parameters could be calculated by macroscopic values, or anything in between. In video games, simulating traffic with a macroscopic model, but displaying it to the player as moving vehicles – for example more vehicles at higher densities, faster vehicles at greater flows – that will disappear as soon as the camera turns away, could be considered as a mesoscopic solution.

Nanoscopic models will go another level deeper and describe a driver's behavior [see 4]. This can be driven by his needs, external impulses he receives, or his character traits. A person who is late for work might go over the speed limit, while someone who is an insecure driver might slow down or brake earlier. A person that is not simulated as a vehicle might also drive part of his route, then take the train, bus, or walk the rest to be faster or minimize cost.

3.1.2 Measurement

In real traffic, data is captured by sensors embedded into the road. They can only detect a vehicle at a given time at their location. By having multiple sensors at given distances, the velocity of vehicles can be calculated. These are microscopic values that can be used to calculate macroscopic values that can give a better view on the bigger picture [16, pp. 13-15].

The average velocity V is calculated by averaging the velocities v_α of all ΔN vehicles passing a road section at x over some time interval around t [16, p. 16].

$$V(x,t) = \frac{1}{\Delta N} \sum_{\alpha} v_\alpha \tag{3.1}$$

The traffic flow is defined by the number of vehicles ΔN passing a location x at a given time t over the interval Δt [16, p. 15].

$$Q(x,t) = \frac{\Delta N}{\Delta t} \tag{3.2}$$

These can be used to estimate traffic density ρ at a location x and time t.

$$\rho(x,t) = \frac{Q(x,t)}{V(x,t)} \tag{3.3}$$

This is known as the hydrodynamic equation which is defined with a spatial average velocity, not a temporal average, so using it will not give exact values [16, p. 17].

3.1.3 Acceleration Models

Acceleration or car-following models define how the acceleration of vehicles is calculated. A vehicle's acceleration is calculated with regard to a leading vehicle and its properties. If the model is considered *collision-free*, velocities and gaps are kept in a range where a vehicle is able to come to a complete stop without a collision at any time if the leading vehicle brakes with its maximum deceleration. If another vehicle appeared between the vehicle and its leading vehicle, for example after driving onto the road at an intersection, the gap to the new leading vehicle could not be safe for the current velocities, which means the vehicle may not have enough time to adjust its acceleration to avoid a collision. Therefore acceleration models are only valid as long as there are no lane changes, for example on a ring road with a fixed number of vehicles or on a straight road where vehicles appear on one side and disappear on the other.

Vehicles can have properties such as a maximum velocity v_0, maximum acceleration a_0, maximum deceleration or brake acceleration b_0 and minimum gap s_0 they try to uphold to their leading vehicle, i.e. the vehicle that is driving directly in front of them. Gaps are always measured from the front of a vehicle to the back of its leading vehicle. As the term *car-following* suggests, vehicles will know their leading vehicle and its properties, which define the acceleration for the next simulation step [16, pp. 139-151].

Gipps Model

Knowing the current gap s to a leading vehicle and its velocity v_l, a safe velocity v_{safe} can be calculated depending on a vehicle's maximum deceleration $b > 0$ and the minimum desired gap s_0 to any leading vehicle. The reaction time Δt controls the distance the vehicle will drive while the leading vehicle brakes at its maximum deceleration which is considered to be the same for all vehicles.

$$v_{\text{safe}}(s, v_l) = -b\Delta t + \sqrt{b^2\Delta t^2 + v_l^2 + 2b \cdot (s - s_0)} \tag{3.4}$$

The safe velocity is the highest velocity a vehicle can travel at without colliding with its leading vehicle, in case the leading vehicle brakes with the maximum possible deceleration. Since this velocity can be higher than the fastest allowed velocity, or exceed a vehicle's maximum acceleration capabilities, the actual velocity for a vehicle in the next simulation step is given by the model equation (3.5), where t is the time of the last simulation step, i.e. the last time where all vehicle's properties are known, and Δt the amount of time that will be simulated during the step, after which all vehicles' properties will be known again.

$$v(t+\Delta t) = \min\left[v + a\Delta t, \quad v_0, \quad v_{\text{safe}}(s, v_l)\right] \tag{3.5}$$

The first term of the model equation covers acceleration. If there is no vehicle in front, or the leading vehicle is far enough away, a vehicle will always accelerate with its highest possible acceleration. The second and third term assure that a vehicle will not drive faster than it is allowed or safe.

To avoid problems that arise when the order in which vehicles are simulated is not known, the velocity is always calculated with old velocities from the previous simulation step. This way, all vehicles accelerate at exactly the same time, even though accelerations and thus velocities are determined sequentially [10, 16, pp. 157-161].

Intelligent Driver Model

The intelligent driver model works with a desired gap s^* that a vehicle tries to keep to its leading vehicle, which depends on the vehicles velocity v, how fast it is approaching the leading vehicle (Δv), the following time T, which describes the desired time it should take a vehicle to be at the location of its leading vehicle, its maximum acceleration a and deceleration b.

$$s^*(v, \Delta v) = s_0 + \max\left(0, \quad vT + \frac{v\Delta v}{2\sqrt{ab}}\right) \tag{3.6}$$

The acceleration $\dot{v}$ is then calculated by modifying the highest possible acceleration a. The factor depends on the velocity v, the maximum velocity v_0, an exponent δ that controls how long a vehicle will accelerate slowly, i.e. v/v_0 is close to 0, and equally how long it takes a vehicle to reach its highest velocity, i.e. v/v_0 is close to 1. Figure 3.3 shows this behavior.

$$\dot{v} = a\left[1 - \left(\frac{v}{v_0}\right)^{\delta} - \left(\frac{s^*(v, v - v_l)}{s}\right)^2\right] \tag{3.7}$$

The closer a vehicle's desired gap s^* is to its actual gap s, the slower it will accelerate or brake. If the current gap is smaller than the desired gap, the vehicle will accelerate, if it is bigger it will brake [16, pp. 161-169].

3.1.4 Lane-Changing Model

Acceleration models can only describe a vehicle's acceleration function with regard to its leading vehicle. If a vehicle wants to merge into another lane it has to look for a gap that is large enough for it to not only have a safe gap s_{safe} to its new leading vehicle but also to its new following vehicle. The same applies if it merges or turns into a priority road. The merging vehicle is assumed to know its gap to the new

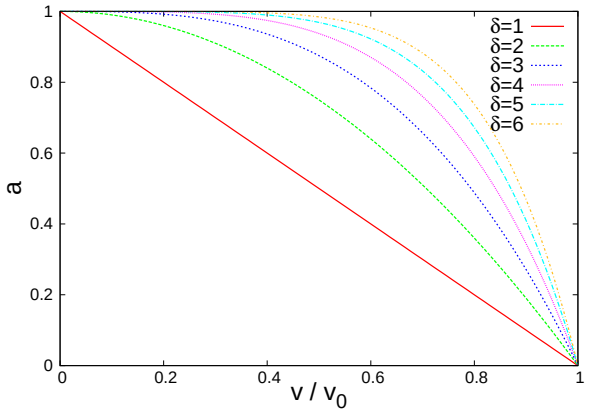


Figure 3.3.: Acceleration behavior for different δ values. The higher δ, the longer acceleration will be at its maximum value.

leading vehicle at the point in time when it will have merged. This gap is considered being safe if a full brake of the new leading vehicle does not lead to a collision if the merging vehicle brakes with a safe deceleration b_{safe}.

For the intelligent driver model the safe gap is given by

$$s_{\text{safe}}(v_h, v) = \frac{s^*(v_h, v_h - v)}{\sqrt{\frac{a_{\text{free}}(v_h)}{a} + \frac{b_{\text{safe}}}{a}}} \quad , \tag{3.8}$$

where v_h is the velocity of the following vehicle, v is the velocity of the leading vehicle, s^* is the desired gap between two vehicles, a is the maximum acceleration and $a_{\text{free}}(\tilde{v})$ the calculated acceleration of the vehicle with velocity $\tilde{v}$ if it does not have a leading vehicle. If the safe gap is compared to the predicted gap to the new leading vehicle, v_h is the velocity of the merging vehicle and v the velocity of the new leading vehicle. If it is compared to the gap to the new following vehicle, v_h is the velocity of the new following vehicle and v the velocity of the merging vehicle. Calculating a safe gap is a special case of the more general *security criterion*, which does not allow merging if any involved vehicle has to exceed the safe deceleration [16, pp. 197-200, 204-205].

3.2 Relevant (Game-) Design Patterns

Video games are complex software that exhaust the power of a computer to the last. This would not be possible if the code was not optimized to fully utilize every last bit of power. This section briefly showcases some design patterns that are frequently used in games and relevant to the optimization component of this thesis.

3.2.1 Game Loop / Tick

Computer programs are a long sequence of operations that can be influenced by user input. For this the program has to be able to read input whenever it can process it. This is done by a big loop that collects all input, fires corresponding events, and loops back again. In game engines one full round-trip of the game loop results in a frame, which is a still image of the game's current state that is displayed on an output device. To let objects and actors know that they can react to new input or just run their common routines, every object can have a `Tick` function that is called during every frame. It gets passed the

amount of time that was needed to calculate the last frame, which is the time since the last displayed game state [12, pp. 161-176].

3.2.2 Scene Graph

A scene graph is usually actually a tree, which is used to organize objects in the three dimensional game world, and operations upon those objects. The virtual world of video games can usually be viewed from at least three different coordinate-systems: The *object system*, where the origin is the center, or any other arbitrarily set point of the object, and the location of all vertices of that object are given in relation to that point; the *world coordinates*, where all objects' locations are given in relation to the center of the virtual world; and the *camera coordinates* that are relative to the virtual camera that views the scene. A 3D artist who creates virtual objects for the game wants to operate in object-space, a level designer who assembles those objects to a scene operates in world-space, and a graphics programmer who wants to render the world needs camera-space, where it is easier to determine which objects can be seen, and which are obstructed or out of frame.

Coordinate space transformations are made easier by the scene graph by having a root object for the world, and other objects as children of the root object or other objects. Each object has a transform property, which stores location, rotation and scale relative to its parent. To get the world coordinates of an object's vertex all transform operations of the subsequent parents have to be chained until the root is reached. To get camera coordinates from world coordinates an inverse transform operation of the camera has to be calculated.

Those coordinates are usually cached whenever they are requested for the first time, so they do not have to be calculated with each frame. On the other hand, this means that whenever an object is moved or rotated, the next time any coordinates of its children are requested they have to be recalculated.

Another benefit of a scene graph is that objects can be reused. There do not have to be four 3D models for the wheels of a vehicle, only for different transform properties that refer to one model. Rotation of those wheels can easily be done by aligning their model to a central axis and just animating the value of one of the three rotational coordinates, instead of having to fully animate location and rotation of all vertices of the wheel's model.

3.2.3 Dirty-Flag

There are many of variables in a simulation that are derived from properties of the simulation or from other variables. Those variables can be reevaluated every time the simulation's state changes and stored locally. If there are many of those variables that are calculated with every update but may not be needed every update this wastes computing time. This can be optimized by deferring calculation of those variables to whenever they are requested. Then there may be some of those variables that are requested very often so the same value is calculated over and over again. The two methods can be combined and extended with a *dirty flag*. The dirty flag is set every time anything that influences the variable changes in the simulation, which serves as a mark that the value has to be recalculated. Whenever the variable is requested it is calculated, cached and the dirty flag is reset if it was set before, whereas the cached value is returned if the dirty flag is not set. This reduces calculations to a minimum and is useful if a variable is not needed in every update but also may be requested multiple times during the same update [12, pp. 345-360].

4 Traffic Simulation in Video Games

Traffic simulations in video games aren't always the same. There can be a lot of variation firstly by genre and subgenre, but also by the game's focus inside the genre. To get an overview of what is possible and already used in games, the following sections describe traffic simulations in different games of different genres and ages.

4.1 The Settlers II / 10th Anniversary Edition

The Settlers II is a strategy game originally from 1996. Settlers land with a ship on unknown territory and have to build a new empire. Soldiers have to be equipped with weapons, that have to be produced from resources. Workers have to be fed with meat from hunters or bread from bakers.

Every building is connected to a road that is divided by flags. On each road section there is one settler – on heavily used roads an additional donkey – transporting resources from one flag to the other, and others back. This way resources are transported from production to storage, or from storage to where they are used up.

Figure 4.1.: Headquarters with settlers on dirt paths and donkeys on cobblestone roads, connecting blue flags.

The simulation does not need to handle collisions since there is only one settler on each road section. On the other hand this can lead to large build-ups of resources at bottlenecks. When soldiers move from the main building to their barracks or watch towers, they use the same roads and simply pass by other settlers. The simulation in The Settlers II is more fit to simulate a village, not a city.

4.2 GTA V

Grand Theft Auto V (GTA V) from 2013 (for consoles) is an open world game and not a city builder. The player can complete missions that contain car chases, bank robberies and objectives better known from the shooter genre where the player has to kill all enemies, usually with a gun. He or she can also freely roam the city, buy or steal cars, complete taxi, ambulance or fire fighter missions, or get into fights with gangs. In contrast to the other games, GTA V is usually played in third or first person perspective.

The player cannot influence traffic by building roads, but by driving or walking on them, blocking traffic or threatening drivers with a weapon. The latter causes drivers to panic, trying to get away at full speed, or getting out of the car and simply running. This indicates at the simulation rather being microscopic than macroscopic because behavior of drivers is simulated, not properties of traffic.

To save computing power, vehicles can disappear if the player is looking in another direction. This would not be feasible if every vehicle is transporting resources or people to specific locations as in other games like The Settlers II, OpenTTD or Tropico 5. There are also only few different car models seen on the street at any given time, to save RAM on the graphics card, even though many different car models are available.

A game like GTA V in which the action is happening close to the street is not viable for a city builder that simulates economics, which rather happens from far away.

4.3 Anno 2070

Anno 2070 is a business simulation with real-time strategy (RTS) elements from 2011. The game is the fifth part of the Anno series of city builders and business simulation games, each part is set in a different century. As the title says, its story takes place in the year 2070, when the player has to found colonies on islands to mine resources, produce different goods to satisfy citizens' needs, build a city with factories and housings, and build military units to defend the city, mostly at sea.

Figure 4.2.: Early-game screenshot of Anno 2070. People walk on roads between the city center and their homes, and trucks transport resources from a mine to a processing plant.

There is not that much traffic on the streets as in usual street traffic, but nonetheless resources and products are transported from mines to factories and from factories to warehouses.

Having military units that can move freely without being restricted to roads, Anno 2070 features a type of vehicle movement that is much more likely to exist in a video game than usual traffic is. One way to implement this kind of movement is by using a NavMesh.

A NavMesh is usually represented by a plain where characters can move, with other objects or zones marking locations the characters cannot move to. When a character gets the command to move to a new location, it calculates the shortest path, avoiding collisions with objects or forbidden zones. When a character would collide with another moving character, there are rules to avoid a collision, like always dodging to the right. Other characters can also be implemented as moving objects on the NavMesh that have to be avoided, so whenever another character crosses a character's path, it recalculates the shortest path around it to its destination.

This kind of movement isn't practical for street traffic, because of the amount or traffic rules. The NavMesh could be restricted to roads, which prevents vehicles from driving on the sidewalk, but they would still drive on the wrong side of the road, if it meant a shorter path. A NavMesh is also only a good choice if there are few characters moving on it, thus trying to simulate thousands of vehicles with a NavMesh is probably not feasible.

4.4 OpenTTD

Inspired by Transport Tycoon Deluxe from 1996, OpenTTD is an open source transport simulation, first published in 2004, that is still being worked on and had new releases in 2016. It is available for PC and Android.

Figure 4.3.: Roads with bus stops and train tracks with a train transporting coal from a mine to a power plant.

The game features maps with factories and mining facilities, small cities with roads, mountains and rivers. The player has to build a transport company that transports resources to factories, processed products to other factories and so on. He or she can also build public transport systems, roads and railways. Everything in the game is grid-based, which leaves vehicles only four directions to move in: north, east, south and west. While driving around corners or on a turnout, vehicles can also move diagonally.

The graphics of OpenTTD are very basic because it is inspired by a 20-year-old game. This allows it to perform very efficiently and therefor run on mobile devices as a fully featured game. Since the game's mechanisms also seem like they are 20 years old it does not fit the requirements and will not be used as a model to built the simulation on.

4.5 City Life

(a) Straight roads arranged on a grid, bridges, big buildings and busy traffic.

(b) Vehicles waiting and turning at an intersection with three different types of road (two, four and six lanes).

Figure 4.4.: Roads and traffic in City Life.

City Life is a city-builder published in 2006. The player can build residential-, work-, infrastructure- and entertainment buildings, and roads to which buildings are always connected. While work places and entertainment buildings always provide a place for certain citizen groups, housings are generic and inhabited by citizens who can find work and entertainment nearby.

This allows City Life's special game-play mechanism, where the player has to separate rivaling citizen groups by building corresponding buildings with enough space between them, or put a third group, who get along with both rivaling groups between them as a buffer. Roads can be up- or downgraded to more ore less lanes, to provide space for more vehicles or to save maintenance cost.

Every citizen is identifiable and has a home. Every filled work place employs one of the citizens, so every citizen has to be simulated individually. Even though this implies a microscopic simulation traffic seems to be simulated mesoscopically. The player can click on any vehicle or pedestrian and get his or her name, home address and work place. A small amount of citizens can be saved to be accessed quickly later. Vehicles can appear and disappear on the roads, though. Cars accumulate at intersections with red traffic lights only up to a set distance. Any other vehicle that drives towards the waiting cars will just disappear. Vehicles also appear and disappear at dead ends, which can be seen in Figure 4.5.

Figure 4.5.: A vehicle disappearing at the end of a road in City Life.

This way of simulating traffic makes it possible to save computing power with efficient macroscopic simulations that are only visualized by individual cars. This also makes traffic not look static when the player does not provide enough roads, and thus brings more life into the city.

With traffic becoming denser on roads and vehicles taking longer to their destination players are usually incentivized to build a larger road network. Vehicles disappearing when roads are jammed in City Life makes the player feel as though his actions do not have consequences. The mesoscopic simulation also does not fit the requirements for this thesis.

4.6 Sim City (2013)

2013's remake of the classic city-builder from 1989 features small areas where the player can put roads, infrastructure buildings and residential, commercial and industry areas. The reason why the game only contains small areas is that everything in the game is simulated microscopically.

Roads are not only used to transport people in cars, but also for electricity, sewage and messages, like a burning house calling for fire fighters. All these packages, visible or not, travel from wells to be collected by sinks.

In a talk about the development of the game's simulation system at Game Developers Conference (GDC), Dan Moskowitz talks about building a core game mechanic and using it for everything that it is suited for, for example the roads not being restricted to cars, but also being used for abstract messages. He also talks about traffic simulation being one of the trickiest parts of game development, because of all the possibilities for gridlocks [1].

Sim City can be compared to the requirements of this thesis and therefore can be used as inspiration for the prototype simulation.

4.7 Tropico 5

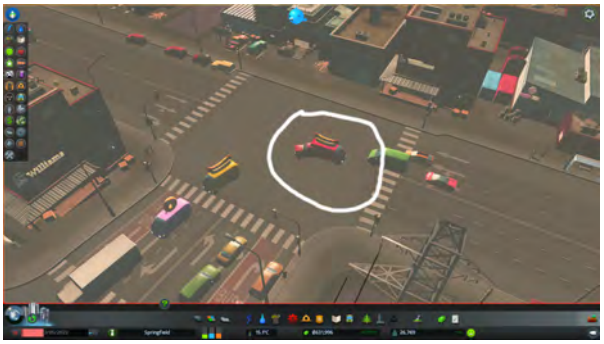
Figure 4.6.: Traffic on a tropical island in the World War age. Mostly trucks use the road, since very little tropicans have vehicles yet.

In Tropico 5 from 2014 the player has to build a colony on a tropical island by harvesting resources, cultivating plants and trading everything for a profit. Products and building materials have to be transported over the road network the player builds. People can wander freely between the buildings, sometimes crossing the road. Vehicles do not brake for them but simply pass through.

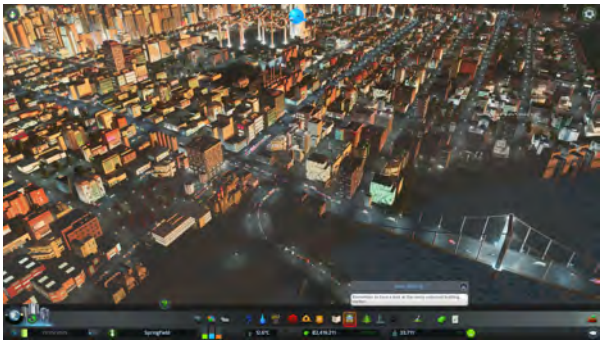
The game simulates every single citizen of the island. Builders use roads to travel from their construction office to a building site. Teamsters can transport building materials, resources and finished products on the road from a production facility to the place where it is used or sold.

The player can build every single building individually, and is only depending on citizens to move to the island and fill free jobs. Buildings and intersections can only be placed on a grid, roads can have curves and people can walk freely, though. Compared to the requirements of this thesis the traffic in Tropico 5 has a smaller scale and is more restrictive, for example in intersection placement.

4.8 Cities: Skylines



(a) Screenshot from Cities: Skylines that shows two cars taking a turn at an intersection, overlapping.



(b) Vehicles waiting and turning at an intersection, different road types with multiple lanes, parking spaces, trees or bus lanes.

Figure 4.7.: Roads and traffic in Cities: Skylines.

In Cities: Skylines from 2015 the player usually starts by building roads. Housing, commercial and industrial areas can be assigned to grids adjacent to roads. While the player can only place special buildings like power plants, water pumps, hospitals or schools, the game will automatically create all other buildings in their assigned areas.

Traffic is quite busy in Cities: Skylines. Workers have to travel to their work places and back home again, trucks transport goods from factories to harbors, and fire trucks rush from the fire station to extinguish a burning building. All vehicles find the shortest or fastest route when they are spawned, which can lead to a traffic jam on one lane of a highway, while two lanes are completely empty [2]. Roads can be upgraded to have more lanes, higher allowed velocities, one-way roads, or roads with bus lanes. Subway tubes can be built beneath the city, which form a public transport system besides buses and taxis. People can also walk on sidewalks or special pedestrian paths. There is always the need to take some load off the roads, which can also cause noise pollution, if they aren't lined with trees.

The game is usually played from a far-away perspective, where some vehicles and most pedestrians won't be displayed. The closer the camera zooms into the road, the more road users can be seen. Vehicles can then sometimes be observed to drive through each other when turning at intersections as in Figure 4.7 (a). This is assumed to have been a deliberate choice to save computation time. In the correct overlay, vehicles can be selected to show information of their origin, destination and cargo. There's also an overlay for traffic density to identify bottlenecks, which inspired the debug mode for the prototype traffic simulation of this thesis.

Cities: Skylines gives the player a lot of control over traffic. It supports different types of road with multiple lanes, different forms of transportation and overall a good overview of what is going on in traffic. This makes it even more frustrating when all vehicles line up on one lane of the highway while a second lane is completely empty. Nonetheless this is closest to the requirements for the simulation of this thesis, it even supports more features in terms traffic.

5 Model Description

This chapter describes the model that was used to implement the prototype simulation which in turn was used for performance measurements. The model is a combination of the existing acceleration and lane-changing models that were explained in Chapter 3. It is enhanced by vehicle recognition algorithms and path finding, which are necessary for a simulation but are not covered by the model equations.

5.1 Data Structure

The traffic simulation consists of logic components, immovable objects and movable objects. Figure 5.1 shows a simplified UML class diagram of the traffic simulation classes to give an overview over those components and as a reference for names used in this and the following chapters. Naming of classes and functions that are used for explanations is based on the naming convention of Unreal Engine 4. Therefore classes start with an `A`, `F` or `U` followed by an upper case name and names of functions and variables also begin with an upper case letter.

5.1.1 Road Network

Road network classes have in common that they never, or at least rarely change during simulation. The road network consists of intersections that are connected by roads. A dead end is an intersection with only one adjacent road. An intersection with two adjacent roads is just a distinctive point on a longer road, for example a corner or bends of a longer road, that can be used in a visualization of the road network to set their locations.

Every road has two lanes, one for each driving direction. Having two lanes on each road instead of just the road itself and a direction flag for each vehicle helps to separate the vehicles into the two directions. It also makes positioning of visual representations of vehicles easier, faster and more precise. If roads are not straight in the three dimensional world, calculating the offset from the middle of the roads for the different directions becomes more difficult than just adding a vector that is constant for a road. There is also always a longer lane and a shorter lane in a bend so having two lanes instead of one road helps to store those differences.

Lanes also become necessary at intersections if vehicles have multiple options to turn. An intersection creates auxiliary lanes, each representing the path of one possible turn. In this context driving straight is considered as taking a turn as well. At a dead end the only auxiliary lane connects both lanes to make it possible for vehicles to make a U-turn.

Unlike roads, lanes can only be traveled in one direction, from their start node to their end node. A lane node references all lanes that come out of the node, either one at the start node of a lane on a road, or multiple auxiliary lanes from other roads at an intersection. It also references lanes that go into the node, meaning one at the end of a lane on a road, or multiple auxiliary lanes that come from other roads at an intersection.

Figure 5.2 shows an example from the Unreal Engine 4 editor to visualize the data structure. The editor was used to generate road networks by arranging the visual representations of roads (gray planes), buildings (gray boxes) and intersections (yellow squares). For this purpose a scenario was created beginning with a single intersection, extending roads from existing intersections to new ones, connecting

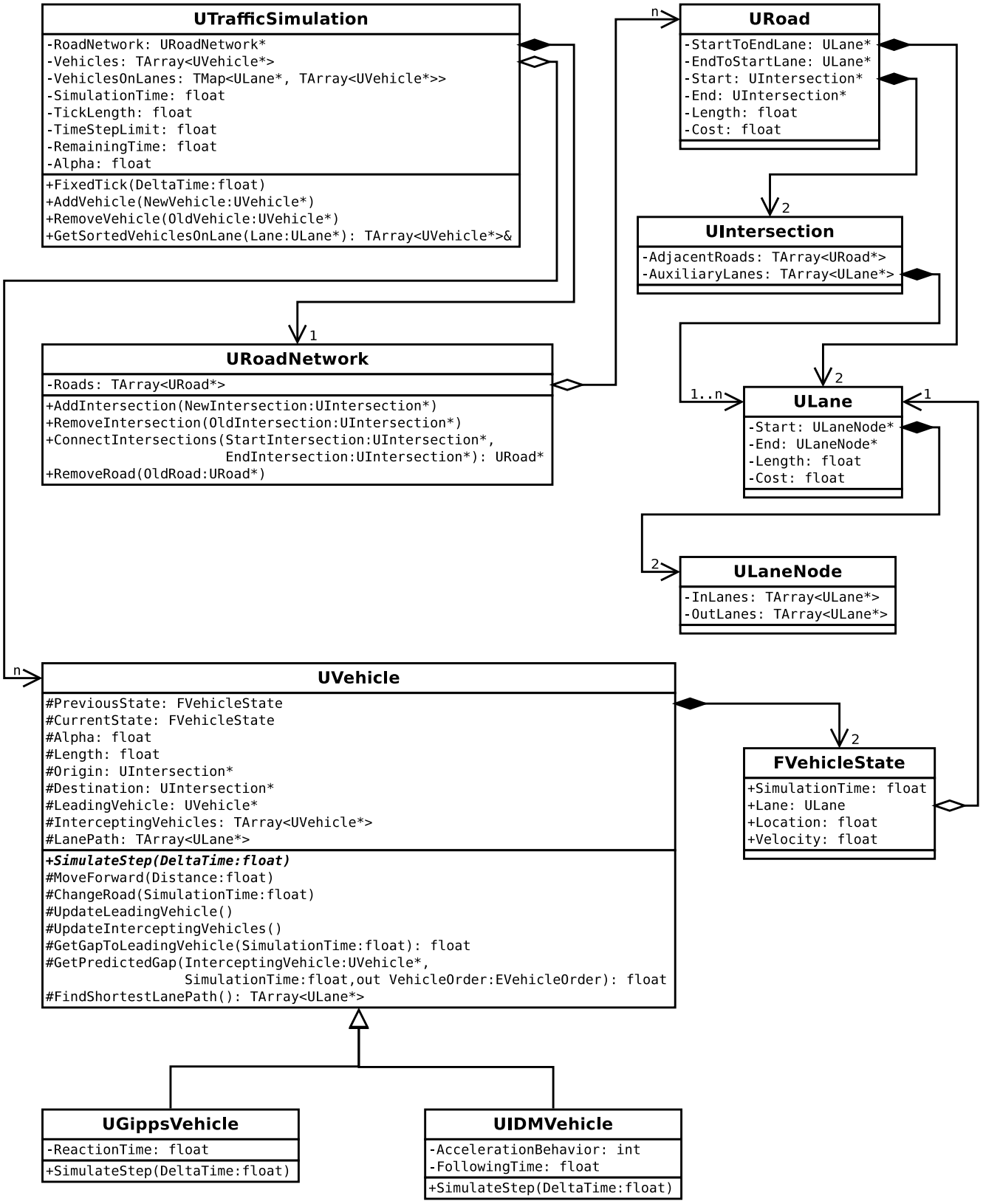


Figure 5.1.: UML class diagram of simulation related classes. Some functions and parameters are omitted for better readability.

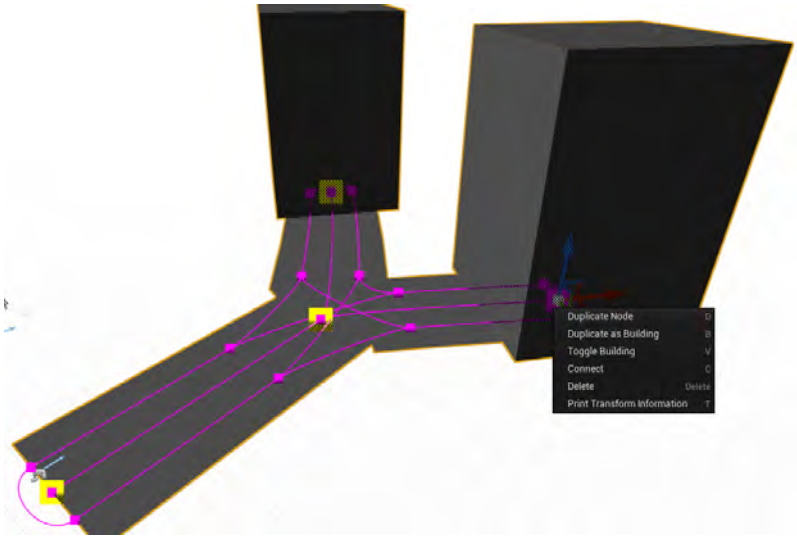


Figure 5.2.: Example view of the road editor. Yellow squares represent intersections, magenta lines represent lanes, small magenta squares represent lane nodes. Lines directly between yellow squares show the path of a road. Lanes at intersections show all possible turns a vehicle can take. The dark gray boxes are buildings where vehicles will spawn. The intersection of the right building is currently selected and the context menu contains possible operations. Only intersections can be moved, roads will be built between intersections, lanes align with the roads and are automatically generated at intersections.

or deleting intersections and turning them into buildings where vehicles would spawn during simulation. Whenever two intersections are connected a road is created automatically with its two lanes and the new auxiliary lanes at the two intersections.

5.1.2 Simulation

Simulation classes handle the dynamic behavior of the traffic simulation by being created, destroyed and changed frequently during simulation. At its core there is `UTrafficSimulation` that contains a road network and all vehicles that move on the roads. Its `FixedTick` function takes any time interval `DeltaTime` and cuts it into equally sized steps as described in Section 5.4. It tells every vehicle to simulate its movement and acts as a connection between vehicles and the road network, so the road network classes do not need to know about vehicles, and vehicles do not need to store redundant information about roads and lanes.

The `UVehicle` class contains all information all vehicles have in common, and serves as an interface for all vehicle classes, so the traffic simulation only needs to call functions on `UVehicle` objects. Inside this class, `SimulateStep` is the main function that handles everything during a simulated step. Its implementation in one of the child classes `UGippsVehicle` and `UIDMVehicle` will calculate an acceleration according to its respective acceleration model and then move the vehicle forward according to its current velocity.

At any necessary point in the simulation of a step `UpdateLeadingVehicle` is called to look for the next vehicle that is currently driving in front of the vehicle on its path (see Section 5.2.3). `UpdateInterceptingVehicles` similarly finds all vehicles it might collide with at the next intersection (see Section 5.2.3).

5.1.3 Optimization Patterns

The design patterns described in Section 3.2 were used to optimize performance where it was needed most, meaning in components that were performing worst, as detailed below.

Blending Between States

Since the simulation step length is set to be the same on every machine (see Section 5.4), faster machines that render more frames than they simulate would display choppy movement of vehicles. This is avoided by linearly interpolating a vehicle's state between the last two simulated states for any time between those simulation times. This can also be used to reduce the load of the simulation by increasing the step length to a level where the results of interpolation cannot yet be seen immediately. Instead of simulating 17 ms at 60 FPS or 34 ms at 30 FPS, simulation step length can be increased to around 500 ms, thus reducing load to a fifteenth or thirtieth.

Caching

In foresight to a multiplayer mode, automatic generation of auxiliary lanes in the road network are deferred and not saved, to reduce the amount of data that needs to be transferred over the network. They are generated upon creation, so they can be seen in the editor, and whenever a level is loaded.

More importantly, the interpolated render state of each vehicle is only calculated when it is first requested. Whenever α, i.e. the rendered simulation time changes, a dirty flag is reset.

Update Pattern

The update pattern is used by the game engine to notify all objects to make their calculations for the current frame [12, pp. 177-192]. In Unreal Engine this function is called `Tick(float DeltaTime)` or `TickComponent(float DeltaTime)`, where `DeltaTime` is the time that has passed since the last call of that function. In buildings this function is used to add new vehicles to the simulation and the visual representation of vehicles uses the `Tick` function to get its simulation location for the current time and to update its location in the world in between simulated steps. The simulation component uses this function to call an update function `SimulateStep` on each vehicle, which calculates acceleration, velocity and location of the vehicle.

Game Engine Settings

The Unreal Engine brings a lot of features like a physics engine, predefined controls, or high-end render settings that can be used immediately, and are used in most games. These features may have a positive effect on player satisfaction or ease of use for developers, but usually have a negative effect on performance. Since the simulation models handle all physics that are needed for the vehicles, engine physics and overlap calculations for notifications and events were disabled for the vehicles.

Distance culling was also introduced for vehicles at a random distance within a given range for each vehicle. This way all vehicles can be seen till a certain distance, and one by one disappear if they are further away from the camera, which means in close-up traffic every vehicle can be seen, further away rough movements can still be made out from single moving vehicles.

5.2 Implementation of Models

After the data structure is modeled and implemented the model equations for acceleration and lane changes need to be implemented as well to complete the traffic simulation. The models described in Sections 3.1.3 and 3.1.4 are implemented with C++ and the Unreal Engine's API to be used in the prototype traffic simulation made with Unreal Engine 4. While choosing and implementing the equations themselves is straight forward, providing them with necessary information turns out to be more complex. This section provides algorithms for finding a leading vehicle, finding intercepting vehicles relevant for lane changes and choosing when to apply the lane-changing model to an intercepting vehicle.

5.2.1 Acceleration Model

To be able to compare the performance of a more complex model that fits all criteria of Section 2.2, to a basic model, a `UGippsVehicle` class was implemented which moves as described in the Gipps model in formula (3.5) in section 3.1.3.

There are several other acceleration models to choose from:

- *Optimal-Velocity-Model* [16, 144]: Vehicles try to drive with an optimal velocity, which is given by a gap-dependent function.
- *Full-Velocity-Model* [16, 147]: This model expands the Optimal-Velocity-Model by adding a sensitivity that depends on the velocity difference to the leading vehicle.
- *Newell-Modell* [16, 148]: Vehicles will always drive at the optimal velocity without considering limitations of acceleration.

All of the listed models are either not complete or not collision free and they do not simulate real driving behavior.

The Intelligent-Driver-Model, implemented in `UIDMVehicle`, is another model based on a driver strategy, describes a more realistic driving behavior and is also easy to use with a lane-changing model, therefore it was chosen to be compared to the Gipps model.

Gipps Model

Whenever a vehicle is told to update, i.e. to simulate a step with duration `DeltaTime`, the safe velocity is calculated. It is then compared to a maximum velocity `v_0` and an accelerated velocity and set as the current velocity. Afterwards the vehicle is told to move forward with the new velocity. This updates its location, changes roads if necessary, and tells following vehicles to update their leading vehicle if it changed road.

Listing 5.1:

```
float SafeVelocity = -b * ReactionTime + Sqrt(Square(b * ReactionTime) +
   Square(v_l) + 2 * b * (s - s_0));
float DesiredVelocity = Min(SafeVelocity, v + a * DeltaTime, v_0);
SetCurrentVelocity(DesiredVelocity);
MoveForward(DesiredVelocity * DeltaTime);
```

Listing 5.1 shows the quite straight forward implementation of the Gipps model equation if the leading vehicle is known at all times. A fully implemented version can be seen in the appendix A.1.1. The model equation only comprises four lines of code out of the approximately 9000, which is a first indicator for there being many other points for optimizations.

Intelligent Driver Model

Just like the Gipps model, the intelligent driver model is implemented quite straight forward, which can be seen in Listing 5.2. A fully implemented version, which is bloated by the lane changing model can be seen in the Appendix A.1.3. As an adjustment for non-perfect lane changes where vehicles might overlap the actual gap `s` has to be clamped to zero. Without the lane-changing model this also comprises just four of the approximately 9000 lines of code. Adding lane changes bloats this to 109 lines, which is an indicator that the far more complex lane changes are more promising for optimizations.

Listing 5.2:

```
float DesiredGap = s_0 + Max(0.0f, v * FollowingTime + v * (v - v_l) * 0.5
   f * InvSqrt(a * b));
float RelativeGap = DesiredGap / Max(0.0f, s);
float RelativeVelocity = v / v_0;
AccelerationForLeadingVehicle = a * (1.0f - Pow(RelativeVelocity, delta) -
    Square(RelativeGap));
```

5.2.2 Lane-Changing Model

In the visual representation of the road network, which is actually used to create roads and intersections, an intersection can be moved freely, and there are no special roads like priority roads. To avoid calculating an order of roads at intersections – which would allow rules like *yield to the right* – and the corresponding need to store and handle this order in the simulation, there is no usual way to find out who has the right of way.

To minimize collisions, and to be able to apply the lane changing model described in Section 3.1.4 a vehicle will predict the gap to all intercepting vehicles (see Section 5.2.3), approximately at the time when it drives onto the lane they have in common.

There are two main distinctions to be made about the gap to an intersecting vehicle:

Intercepting vehicle will be behind vehicle

If the gap from the intercepting vehicle to the vehicle is smaller than the safe gap, i.e. the gap at which the intercepting vehicle would not need to adjust its velocity, the vehicle can keep on driving. Otherwise it will brake as if the intercepting vehicle was driving at the node at which the vehicle will enter the intersection. To adjust for negative predicted gaps, i.e. the prediction of an actual collision, and for too late recognition of the intercepting vehicle, its velocity will be divided by two for the calculation of deceleration.

There are more conditions that try to avoid braking of both vehicles or too early braking. A vehicle will only consider braking if it is not yet on the intersection, and the intercepting vehicle is either on the intersection, or exists longer than the vehicle. If a collision is predicted, a vehicle will also consider braking for an intercepting vehicle that does not exist as long as itself.

Intercepting vehicle will be in front of vehicle

If the intercepting vehicle is predicted to be in front of the vehicle, acceleration is calculated normally, only with the predicted gap as the current gap, and the current velocities as though they would not change until the vehicle enters the mutual lane. This acceleration will only be calculated if the intercepting vehicle already is on the intersection, if the intercepting vehicle exists longer, or if a collision is predicted and the vehicle is not yet on the intersection.

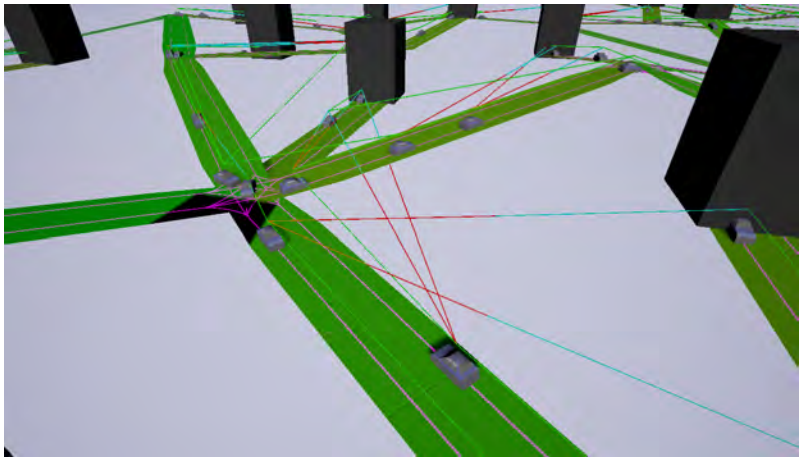


Figure 5.3.: Debug view of a simulation. Light green lines indicate leading vehicles, red and cyan lines indicate intercepting vehicles. The red end of these lines marks the older vehicle, which is used for right of way. The color of the roads show the higher density between the two lanes of a road, where green means $\rho = 0\,\mathrm{m}^{-1}$ and red means $\rho = (l_{\text{vehicle}} + s_0)^{-1}$.

In both cases, only the smallest acceleration is considered and later compared to a normal acceleration derived from the gap to the leading vehicle. An alternative to the chosen way of deciding when to yield is implementing traffic rules that can be found in real traffic, for example traffic lights, priority roads or yielding to the right, all of which entail additional objects, special variations of existing objects or logic for those rules. This in turn costs performance and more importantly restricts the use cases of the simulation, therefore it was decided against those options.

5.2.3 Vehicle Recognition

For car-following models to work, each vehicle needs to know the next vehicle driving in front of it at all times. For the lane-changing model, vehicles need to know which vehicle will drive in front of them and which behind them on the next road, even before they enter the intersection at which they turn. Figure 5.3 shows the debug view of a simulation, where leading and intercepting vehicles can be identified.

Leading Vehicle

For car-following models to work correctly the leading vehicle has to be known at all times, otherwise a vehicle would be able to accelerate too much and collide with, or rather drive through other vehicles. In a one lane scenario a leading vehicle only has to be assigned once a vehicle spawns, there will never be another leading vehicle because there is nowhere for other vehicles to come to the lane except at the very beginning. Since the simulation supports intersections, and thus vehicles having different paths and merging onto a road, the leading vehicle will change over time.

Vehicles first look for a leading vehicle on the lane they are currently driving on. The list of vehicles is sorted for their location on the lane and searched for the index of the vehicle that is looking for a leading vehicle. Then the next index is picked from the list of vehicles to be the leading vehicle.

If there are no other vehicles in front of the vehicle on its current lane, the vehicles on the next lane in its path are sorted for their location, and the vehicle closest to the beginning of the lane is picked as leading vehicle. If there are still no vehicles on the next lane, and if the next lane does not lead to an intersection, but rather is on an intersection, there is only one possible second to next lane, therefore it will be searched as well.

Listing 5.3:

```
LeadingVehicle = FindLeadingVehicle(GetCurrentLane());
if (!LeadingVehicle)
{
  LeadingVehicle = FindLeadingVehicle(GetNextLaneOnRoute());
  if (!LeadingVehicle)
  {
    {
      TArray<ULane*> NextLanes = GetNextLaneOnRoute() ?
         GetNextLaneOnRoute()->GetEnd()->GetOutgoingLanes() :
         EmptyLaneArray;
      if (NextLanes.Num() == 1)
      {
        // there is only one possibility of lanes after next lane
        LeadingVehicle = FindLeadingVehicle(NextLanes[0]);
      }
    }
  }
}
```

Listing 5.3 shows an implementation of updating the leading vehicle. `FindLeadingVehicle` will search a list of all vehicles that are currently on the given lane. Every vehicle knows its index in that list and it is always kept sorted by adding vehicles at the end and updating the index of all vehicles when the first vehicle (index 0) exits the lane. To find the leading vehicle in the list the item with the next lower index than the current vehicle's index has to be returned if it exists. An alternative to looking for a leading vehicle on the current lane, the next lane and the lane after that if there is only one possibility is to go along the whole path of the vehicle until a leading vehicle is found. This may also be limited by only looking as long as a certain distance is not exceeded, for example the distance the vehicle needs to come to a halt.

If still no leading vehicle can be found the vehicle will accelerate as if there was a vehicle standing still at the end of the furthest searched lane. This will prevent the vehicle from driving too fast when it eventually sees another vehicle in front of it. The vehicle will not actually come to a halt at the end of the furthest lane because it will look further down its path for a leading vehicle by the time it enters that lane at the latest.

Blocking Vehicles

With acceleration models being implemented, vehicles driving on the roads and turning at intersections sooner or later two vehicles will meet each other. Since they only adjust their velocities to their leading vehicles they cannot see that the other vehicle will merge to their lane in front of them, therefore when the new leading vehicle is on a vehicle's path the vehicle may drive too fast and crash into the other vehicle. This problem is solved by lane-changing models, but they afford recognition of all intercepting vehicles and multiple additional acceleration calculations for each vehicle, therefore a more naive approach is proposed.

To prevent vehicles from driving into each other at intersections virtual vehicles are introduced as a way of notifying other vehicles that a vehicle is turning onto a lane. Whenever a vehicle drives onto an intersection it creates virtual vehicles that are invisible and not controlled by the simulation or any acceleration model, but by the vehicle that created them. Each virtual vehicle is put onto a different lane that ends at the same lane node the actual vehicles lane ends at. The location of the virtual vehicles is

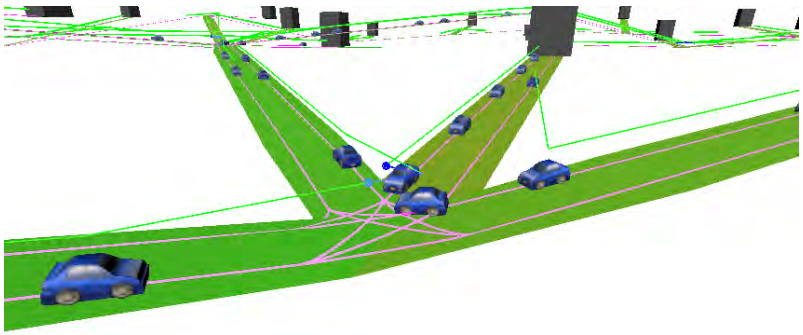


Figure 5.4.: Debug view of a simulation. Light green lines indicate leading vehicles, blue spheres represent virtual blocking vehicles. They are connected to the vehicle that created them with a blue line. The vehicle in the bottom left sees one of the blocking vehicles as its leading vehicle, indicated by the green line.

given by them having the same distance to the end node on their lane as the actual vehicle on its lane. The velocity is set as equal to the actual vehicles velocity. These virtual vehicles are seen by other vehicles that drive towards the intersection and onto the same road the actual vehicle is leaving the intersection on. They will cause those other vehicles to brake accordingly, so they will not collide with the vehicle that entered the intersection first. Figure 5.4 shows the debug mode of a simulation where blocking vehicles can be seen as blue spheres.

While observing some traffic simulations it became apparent that there are several problems with this approach which have an influence on driving behavior of vehicles and the correctness of the simulation. When a virtual vehicle suddenly appears in front of a simulated vehicle the real vehicle either suddenly stopped, exceeding its maximum deceleration, or it drove through or crashed into the blocking vehicle if deceleration was actually capped. Vehicles exceed their maximum deceleration because the acceleration equation does not return decelerations higher than the maximum as long as a safe distance is kept at all times, which can be violated when vehicles can suddenly appear in front of a vehicle. If the acceleration models are only used without lane changes this works fine, with lane changes acceleration has to be capped at a maximum deceleration, though. When a real vehicle cannot brake fast enough and does not stay behind a virtual vehicle it creates virtual vehicles of its own which cause the vehicle that created the first set of blocking vehicles to brake for the second set, therefore both real vehicles brake for each other. They only keep on accelerating and driving again if one of them had enough inertia to exit the intersection before coming to a complete halt. Letting vehicles brake with a maximum deceleration and exceed that limit if they would otherwise overlap with another vehicle eliminates vehicles braking for each other but still does not simulate correct behavior. Therefore a lane-changing model was implemented as well and compared to the naive approach in terms of performance and behavior.

Intercepting Vehicles

Since the road network does not have designated priority roads, vehicles have to consider all vehicles at an intersection at which they turn, or even drive on straight. For the lane-changing model to work correctly, exact locations and velocities of relevant vehicles have to be known at a time in the future, when a vehicle will have driven onto the next lane. The location will be used to find the gap between vehicles that can be combined with the velocities to determine whether it is save for the vehicle to turn or not.

While the model just assumes that all parameters are known it is not as trivial as it might seem to efficiently retrieve them. Gap and velocity to all intercepting vehicles will be predicted in every simulation

step since a vehicle might accelerate or decelerate due to its leading vehicle before it reaches the intersection. Velocities will be predicted as the same velocities vehicles have at the present. To predict the gap, a vehicle calculates how long it will take to reach the intersection at its current velocity, then predicts where the other vehicle will be after that amount of time at its current velocity.

When a vehicle looks for intercepting vehicles, it considers all vehicles that are currently driving on a lane that leads to the vehicle's next intersection, and that will leave that intersection on the same lane as the vehicle. All other vehicles, that will leave the intersection on different lanes, are not considered as intercepting vehicles. That means that the vehicle might collide with those vehicles at the intersection. In this case, *collision* means that vehicles simply drive through each other.

Listing 5.4:

```
InterceptingVehicles.Reset();
TArray<ULane*> OtherIncommingLanes;
OtherIncommingLanes = GetCurrentLane()->GetEnd()->GetIncommingLanes();
if (OtherIncommingLanes.Num() > 1)
{
  // on intersection
  OtherIncommingLanes.Remove(GetCurrentLane());
}
else
{
  // driving towards intersection
  OtherIncommingLanes = GetNextLaneOnRoute() ? GetNextLaneOnRoute()->
     GetEnd()->GetIncommingLanes() : EmptyLaneArray;
  OtherIncommingLanes.Remove(GetNextLaneOnRoute());
}

for (ULane* Lane : OtherIncommingLanes)
{
  // add all intercepting vehicles on intersection
  TArray<UVehicle*> Vehicles = Simulation->GetVehiclesOnLane(Lane);
  InterceptingVehicles.Append(Vehicles);
  for (ULane* IncommingLane : Lane->GetStart()->GetIncommingLanes())
  {
    // add all intercepting vehicles on road leading to intersection
    for (UVehicle* Vehicle : Simulation->GetVehiclesOnLane(IncommingLane)
       )
    {
      if (Vehicle->GetNextLaneOnRoute() == Lane && !InterceptingVehicles.
         Contains(Vehicle))
      {
        InterceptingVehicles.Add(Vehicle);
      }
    }
  }
}
```

Listing 5.4 shows code for identifying intercepting vehicles. The number of vehicles may be reduced to the first few vehicles found on each lane, which could produce false behavior if this drops an intercepting vehicle that the vehicle will collide with and is not able to brake for in time when it sees it.

5.3 Simulation Properties

Variable	Name	Value Range	Default Value
t_{step}	Simulation Step Duration	$[0.0001\,\text{s}, 1\,\text{s}]$	$0.5\,\text{s}$
v_0	Maximum Velocity	$[0\,\text{m/s}, 30\,\text{m/s}]$	$15\,\text{m/s}$
v_{var}	Velocity Variation	$[0\,\text{m/s}, 10\,\text{m/s}]$	$0\,\text{m/s}$
a	Maximum Acceleration	$[0.1\,\text{m/s}^2, 3\,\text{m/s}^2]$	$1\,\text{m/s}^2$
b	Maximum Deceleration	$[1\,\text{m/s}^2, 30\,\text{m/s}^2]$	$1.5\,\text{m/s}^2$
s_0	Minimum Gap	$[0\,\text{m}, 5\,\text{m}]$	$1.5\,\text{m}$
l_{vehicle}	Vehicle Length	$[1\,\text{m}, 5\,\text{m}]$	$2.6\,\text{m}$
δ	Acceleration Behavior	$[1, 10]$	4
$c_{\text{SimulationSpeed}}$	Simulation Speed	$[0.01, 1000]$	1

Table 5.1.: Properties of the simulation, that can be set in the editor.

Table 5.1 lists all variables that are used in equations and can be set as properties of the simulation inside the editor. The simulation step duration t_{step} is the amount of time simulated in each step as described in Section 5.4. To be able to see acceleration and braking on a single lane, the maximum velocity v_0 can be set as a random value within a given range $[v_0 - v_{\text{var}}, v_0 + v_{\text{var}}]$. This results in some vehicles being slower than others, therefore the other vehicles have to brake. If all vehicles had the same maximum velocity and started at the beginning of the lane with $v = 0\,\text{m/s}$, every vehicle would accelerate equally, and the behavior of the acceleration model could not be observed very well. The minimum gap s_0 is the gap between front and back of two vehicles when they are in a traffic jam. This can be set quite short because the acceleration models handle safety distances. The length of vehicles l_{vehicle} can be adjusted to fit the used 3D model of a car. The simulation speed factor will multiply the time given to `FixedTick` to speed up or slow down the simulation. The length of a single simulated step is not affected.

5.3.1 Path Finding

A comparison of different path finding algorithms concluded, that Dijkstra's algorithm belongs to the fastest on real road networks [17], so it was used for path finding.

When a vehicle is created, it has an origin, and a destination is set. It will then retrieve the road network in form of lanes and lane nodes. The Dijkstra algorithm is used upon the lane nodes as nodes of a graph, lanes as edges, and spatial length of lanes and stored additional cost per lane as cost of an edge. The algorithm returns an ordered list of lanes which form the shortest path. Later on, when the vehicle is moving and reaches the end of a lane, it looks up the next lane in this list. Instead of using buckets to find the next node for an interation of the algorithm the list of unvisited but reachable nodes was sorted every step. If path finding becomes a problem for performance using buckets as mentioned in [17] it can be an improvement. Buckets are an ordered array of several unordered nodes within a certain range. For path finding the first bucket could contain all nodes that can be reached within 10 m, the second bucket all nodes that can be reached within 20 m and so on. Whenever the shortest path to a node changes it is put in the corresponding bucket. When the next node is needed only the first non-empty bucket needs to be sorted, which is more efficient than sorting the whole list in each iteration.

The additional cost of lanes is used to represent road usage, and thus additional travel time. If a lane is completely empty, a vehicle can drive at its maximum possible velocity – if there is no other vehicle on the next lane. If there are other vehicles on the lane, a vehicle will probably have to slow down to avoid collisions, so its travel time will increase. The more vehicles there are on a lane, i.e. the higher its

vehicle density, and the slower the average velocity of vehicles on this lane, the higher the possibility for a vehicle of not being able to travel at maximum velocity. This additional travel time can be seen as the lane getting longer, therefore if the *fastest* path is preferred to the *shortest* path adding expected travel times to the cost for path finding is a better estimate than just the length of roads.

The average velocity v_{average} and the maximum velocity v_0 can be used to calculate how much longer a lane will seem with a slower than maximum velocity. This results in the additional cost factor for velocity adjustment c_v.

$$c_v = \frac{v_0}{v_{\text{average}}} \cdot l_{\text{lane}} \tag{5.1}$$

The vehicle density ρ is given in number of vehicles per meter, this the additional cost factor for vehicle density c_ρ is the percentage of a lane occupied by vehicles. If the whole lane is backed up by vehicles, every vehicle occupies space given by $l_{\text{vehicle}} + s_0$, where l_{vehicle} is the vehicles length and s_0 the minimum gap to the next vehicle.

$$c_\rho = \rho \cdot (l_{\text{vehicle}} + s_0) \tag{5.2}$$

If there is only one very slow vehicle on a very long lane, a second vehicle can travel most of the lane at a higher velocity, therefore c_v has to be modified by c_ρ, which leads to the additional cost c.

$$c = c_\rho \cdot c_v = \rho \cdot (l_{\text{vehicle}} + s_0) \cdot \frac{v_0}{v_{\text{average}}} \cdot l_{\text{lane}} \tag{5.3}$$

When a certain road is backed up this additional cost causes new vehicles to avoid this road by taking a diversion that might be a longer distance, but faster to travel. This is only an estimate for additional travel times though because on the one hand a very slow vehicle at the beginning of a lane might block the whole lane, on the other hand vehicles that are standing still while the travel time is calculated might accelerate and leave the lane empty at the point in time when another vehicle avoids this lane.

5.4 Fixed Tick

The implementation of the Gipps model uses the Euler method to get the velocity if a vehicle is accelerating at its maximum acceleration.

$$\dot{v}(t + \Delta t) = a \tag{5.4}$$
$$v(t + \Delta t) = v(t) + a \cdot \Delta t \tag{5.5}$$

The Euler method is just an estimate of an integral if the differential equation is time dependent. In this case, this means

$$v(t + 2\Delta t) \neq v(t + \Delta t) + a \cdot \Delta t \quad . \tag{5.6}$$

Even though the maximum acceleration is constant, meaning it is not time dependent, velocity – and thus acceleration – depends on three conditions, as seen in equation (3.5). If a vehicle reaches its maximum velocity or its safe velocity is smaller than an accelerated velocity during a simulation step, acceleration

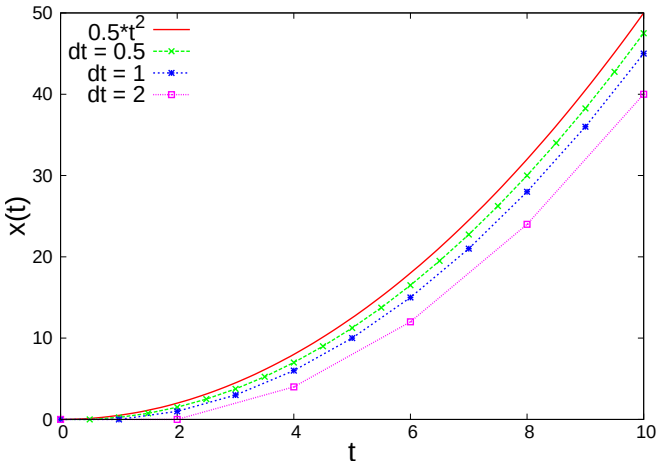


Figure 5.5.: Function for location $x = \frac{a}{2} \cdot t^2$ with $a = 1\,\mathrm{m/s^2}$ numerically integrated with different step sizes $\Delta t = 0.5\,s$, $\Delta t = 1\,s$ and $\Delta t = 2\,s$. The different graphs show how the result of numeric integration depends on the step size Δt. This causes the need for simulation step lengths to be fixed resulting in a simulation's outcome not depending on the power of the machine it was simulated on.

is not constant during this step anymore. Even though the formula just picks the slowest velocity out of the three, that does not mean that it is the slowest velocity at all times during the simulation step, only at the end.

This results in the vehicles' velocities and positions, and thus the whole simulation depending on how much time is simulated during one step, which can be seen in Figure 5.5. The tick function described in Section 3.2.1 tries to produces as many frames per second (FPS) as it can, thus the faster the computer that runs the simulation, the smaller `DeltaTime` becomes, and therefore the time simulated in a step. Even on the same machine, `DeltaTime` can vary because of other applications or the operating system needing CPU-time every now and then. To have the same simulation, no matter on which computer it runs, the duration of all simulated steps always has to be the same.

Since Unreal Engine 4 only offers a tick function with variable step sizes, a function `FixedTick` was implemented, guided by [3] and [12, p. 170]. While [3] uses this for the physics engine, the method is just as useful for traffic simulation purposes.

Listing 5.5:

```
if (DeltaTime > TimeStepLimit)
{
  DeltaTime = TimeStepLimit;
}
RemainingTime += DeltaTime;
while (RemainingTime > TickLength)
{
  for (UVehicle* Vehicle : Vehicles)
  {
    Vehicle->SimulateStep(TickLength);
  }
  SimulationTime += TickLength;
  RemainingTime -= TickLength;
}
Alpha = RemainingTime / TickLength;
```

The function shown in Listing 5.5 takes the variable tick time `DeltaTime` and cuts it into equal pieces of size `TickLength`. Each fixed step that fits into `DeltaTime` is simulated and any remaining time saved for the next fixed tick. Usually the actual simulation time does not exactly fit multiples of the fixed step sizes, thus to optain a close approximation of the state at the actual simulation time the last simulated step is considered to be the step at the next exact multiple of `TickLength`. The remaining time is then used to linearly blend between the previous step and the next step via `Alpha`.

This way simulation data can be equal independent of on which machine it was calculated, and the player will see a state that was interpolated between two exact states.

5.5 Imitation of Game Properties

Implementing a complete video game usually takes companies two to three years in development. Since the time frame of a master thesis is only six months, several aspects of a video game that would use the developed traffic simulation were imitated with simpler methods to be able to see how well the simulation fits in a gaming environment and to identify weaknesses and optimize behavior.

Buildings

A city builder usually simulates the lives of thousands, if not tens of thousands of people who drive to work, back home again, go to free-time activities, have their own personalities and motives. In the prototype simulation these complex actions are substituted by having buildings with randomized spawn rates. Given a time range, each building will randomly pick a time in this range. During simulation, it will wait its chosen time until it creates a vehicle and puts it on a road next to the building, if the space is not occupied by another vehicle, and reset its timer.

This way a steady stream of vehicles is guaranteed, as long as the roads are not filled by so many vehicles that they back up to a building's exit road. This will not simulate rush hours where most of the citizens want to use the roads. A game without explicit times will try to distribute road usage anyway to achieve that vehicles are able to reach their destination, traffic jams are avoided, and there is always something moving on the screen, which looks busy and generally pleasing to a player.

Travel Destinations

Each vehicle will select a random building, except the one it was created at, as its destination and finds the shortest path from its origin to its destination. This way most roads will be used by some vehicles, routes have different lengths, and vehicles from one building will not always take the same route to exactly one other building. If a building is seen as residential, every inhabitant works at a different work place. If its seen as an office building, every worker lives at a different address, which is closer to reality than every worker of a company living at the same place.

Vehicle Types

In a game, there might be different vehicle types with different properties, for example long, slow trucks, that accelerate and brake slowly, normal vehicles, that are faster than trucks, and sports cars, that can accelerate quickly and go at high velocities. This was imitated by allowing a range of those properties, where every new vehicle picks a random value out of the range to set their property.

The built-in distance culling of Unreal Engine 4 will only hide objects but still call all operations on them. There is the possibility to deactivate objects, which results in their tick function not being called anymore, which is not feasible for vehicle actors because they would not move anymore, so it would not be noticed if they would have moved into the players field of view.

A simple distance culling function which does not write the actors' locations into their transform matrices, but rather into a cached property will compare the distance of the cached property to the camera, and only set the real transform if it is in view distance. Otherwise the object will be hidden, the cached property still updated, but the expensive `MoveComponent` operation not executed.

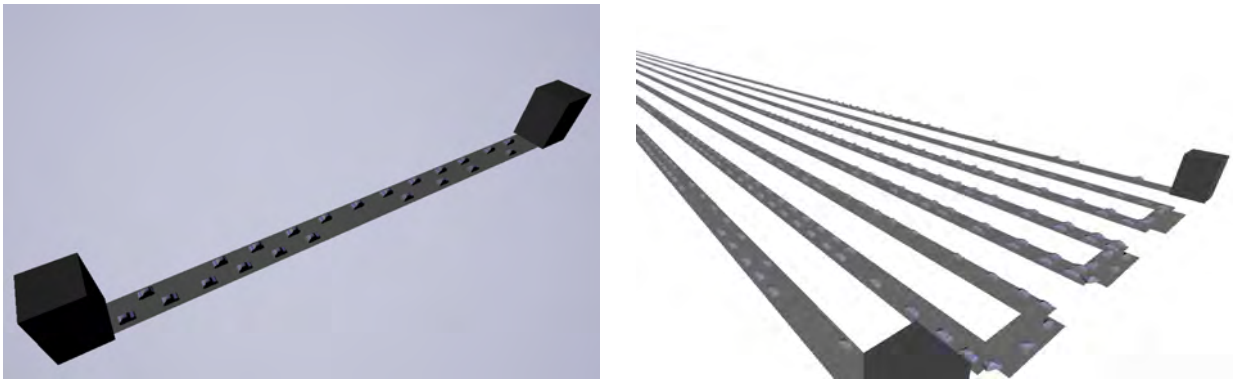
6 Comparison of Different Simulation Models in Different Scenarios

When a new traffic simulation model is developed that aims to optimize performance in real-time scenarios, it is best to identify existing models' most expensive aspects. For this purpose, a simplified Gipps model will be compared to the intelligent driver model in this chapter. The intelligent driver model is also analyzed to find the parts that have the most influence on frame times. Some proposals for improvements will be compared to an original implementation to see if there is even room for improvement. The performance is measured with the Unreal Engine 4 profiler in frame rate or frame time of a simulation running in a game setting. For every frame the profiler measures the time any marked function takes. The frame time of those functions can then be displayed on the y-axis as a function over the game time on the x-axis. Since the text in profiler screenshots is difficult to read all times that are discussed are also displayed in tables. The screenshots give a sense of the graphs and how they change over time, thus they are also included. Everything is run on a 2012 Lenovo ThinkPad X230i with an i5-3320 CPU at 2.60 GHz and 8 GB RAM in 64Bit Microsoft Windows 10 installed on a 250GB SSD.

6.1 Traffic Scenarios for Measurements

The simulation was tested in multiple scenarios with different properties to see if different properties of the road network have different effects on the performance. The three main properties that were changed between those scenarios are length of road sections, number of buildings, and interconnectivity, that means number of intersections and number of roads at each intersection.

The simplest scenario is a single *straight* road with a building at each end, which can be seen in Figure 6.1 (a). It is most useful to investigate car-following behavior without lane changes, without traffic coming to a complete standstill and without path finding. Since there is only one road and two buildings there is also little overhead from these objects. The Road can be stretched to support more vehicles or shortened to be able to survey all of them. It can also be used to identify the influence of the length of road sections on simulation performance.



(a) Straight road scenario. (b) Curvy scenario.

Figure 6.1.: Screenshots of straight and curvy scenarios with longer road sections.

To be able to have more vehicles in a small area and therefore to better observe them than on a long, straight road, a *curvy* scenario with serpentine roads was created as seen in Figure 6.1 (b). It has long straight sections with a turn at each end. Each turn consists of two intersections with a road in between, thus vehicles only have one possible direction at these intersections to drive to. Traffic still moves between just two buildings, making path finding negligible. This scenario can be used to test object overhead of large numbers of vehicles with the least amount of computation for simulation while still being able to observe traffic.

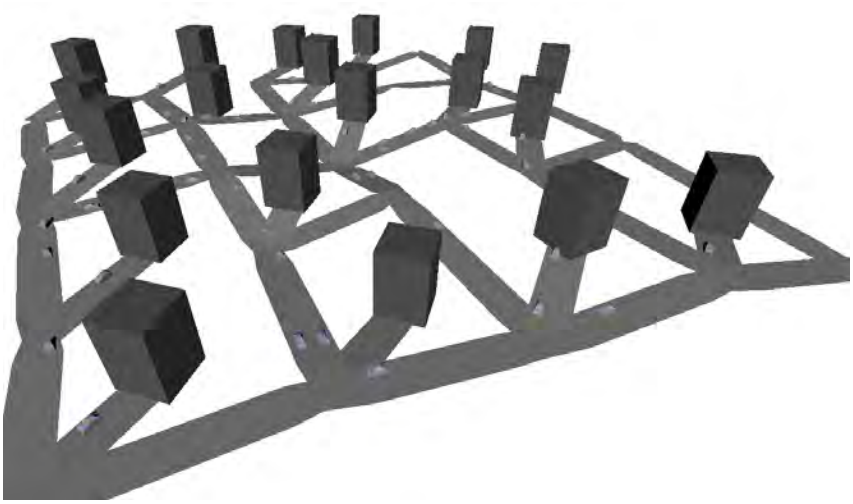


Figure 6.2.: Screenshot of a city scenario with multiple buildings and intersections in a grid.

The default scenario to test new features always was a *city*-like road network, where intersections are organized in a grid. Intersections were connected with neighbors, given a certain connectivity probability, then left-out connections were made until every intersection had a path to every other intersection. Buildings were attached to intersections by a single road, given a certain building rate. All intersections had a small offset to their grid position, to have different road lengths. Figure 6.2 shows an example of how this scenario can look like. By having the city generated randomly different large cities could be created very quickly. This scenario is closest to an in-game situation with multiple buildings, multiple intersections and different paths between buildings. The road network theoretically can support a large number of vehicles in a small area, has multiple possible paths from one building to another, with bottlenecks where there are two sections of the city connected by only one road, multiple destinations for vehicles to choose from, and numerous of buildings to ensure a high and well distributed spawn rate of vehicles. It can test all aspects of the simulation: Path finding, car-following and lane changes. There are also enough possibilities for gridlocks to appear.

6.2 Comparing Performance of Acceleration Models to Each Other

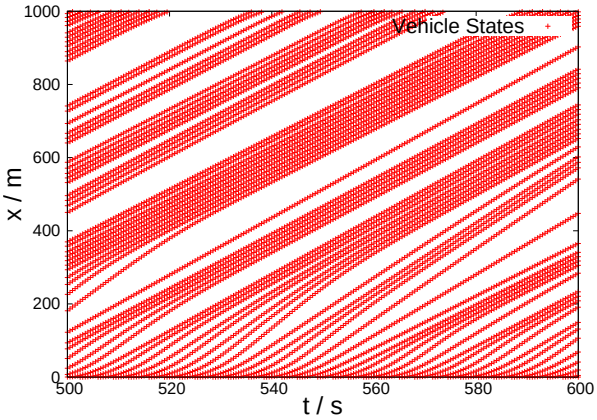
While implementing the simulation, all aspects and features were implemented with the Gipps model in mind, which was performing quite well, except for not being able to avoid collisions when turning at an intersection, which is because vehicles moving from one road onto another is considered a lane change, which the model does not support by itself. To further assess the performance of the Gipps model, it was compared to an implementation of the intelligent driver model.

Table 6.1 compares frame times of a simulation of the Gipps model to one of the intelligent driver model. Both simulations ran in a curvy scenario with three 1 km sections simulated with 0.5 s steps accelerated at 50 times real-time. The times are listed in order of execution: A vehicle first identifies its leading vehicle and calculates the gap to it. Those steps are identical in both models, so measuring almost the same frame times in both models confirms the expectations. A new acceleration is then calculated according to each model, then the vehicle is moved forward assuming a constant velocity during the simulated step. This last part uses the same implementation in both models so it is surprising to see quite different times. Since the intelligent driver model, which has the higher frame time, simulated less vehicles, this cannot be explained with the given data. Acceleration calculation took almost twice as much time with

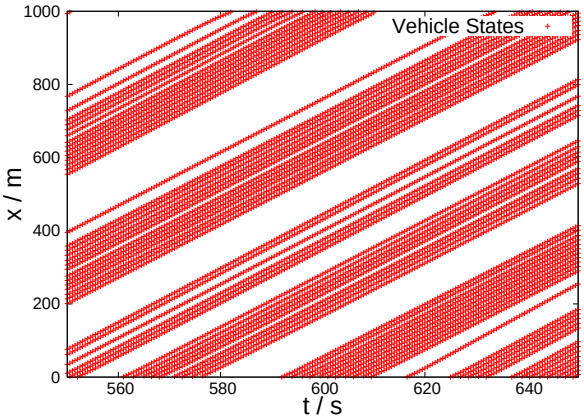
	Frame Time Gipps	Frame Time IDM
Finding Leading Vehicle	0.242 ms	0.241 ms
Calculate Gap To Leading Vehicle	0.075 ms	0.073 ms
Calculate Acceleration	0.066 ms	0.122 ms
Move Simulation Vehicle	0.107 ms	0.122 ms
Sum	0.490 ms	0.558 ms

Table 6.1.: Average frame times of the main parts of the simulation function of the Gipps model and intelligent driver model (IDM). Approximately 280–320 Gipps vehicles and 260–290 IDM vehicles, respectively were driving on 3 km of road. 0.5 s of movement of all vehicles was simulated every 10 ms (with simulation accelerated by a factor of 50).

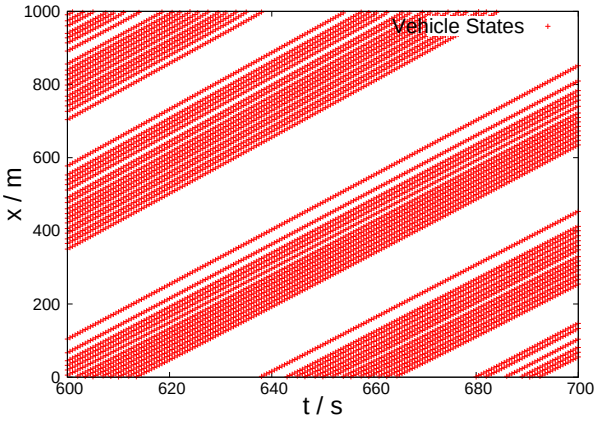
the intelligent driver model than the Gipps model. The intelligent driver uses more calculations to get an acceleration, therefore a higher frame time was expected, but its extent might be an artifact of too few vehicles being simulated at too big time intervals.



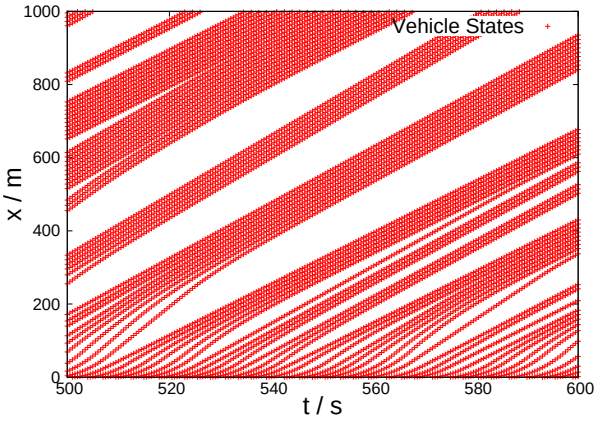
(a) Intelligent driver model: Starting lane.



(b) Intelligent driver model: Middle lane.



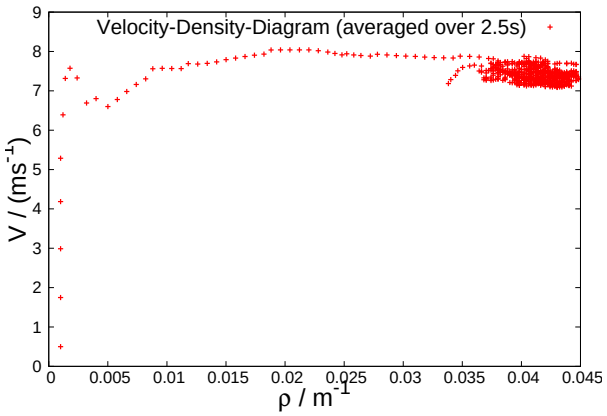
(c) Intelligent driver model: End lane.



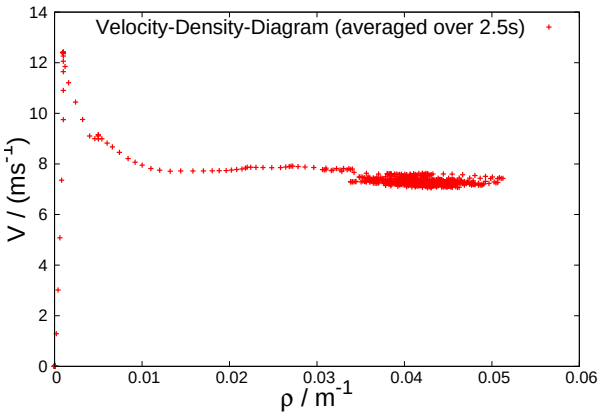
(d) Gipps model: Starting lane.

Figure 6.3.: Trajectories on three road sections in a curvy scenario simulated with Gipps model and intelligent driver model. Each data point is the location x of a vehicle on the lane at a time t. Velocities can be derived from the tangent of a set of data points of the same vehicle, which can be seen as dotted lines in the graphs. The thick red stripes are convoys of vehicles that are slowed down by a slower vehicle in front.

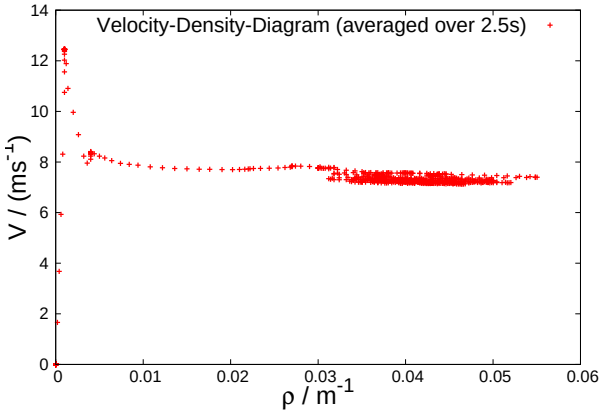
Figure 6.3 shows the trajectories of that simulation for the first (6.3 (a)), second (6.3 (b)) and third (6.3 (c)) section of the three-section road. For the intelligent driver model there is an extract of the trajectories of all three road sections. At the bottom of the graph for the first road section each trajectory has the same time distance on the x-axis, which is the spawn time set for the simulation. The tangents are all parallel to the x-axis, which means all vehicles start with a velocity of $v_{\text{start}} = 0\,\text{m/s}$. Further down the road, thus further up on the y-axis trajectories become more interesting. Vehicles accelerate, so the trajectories are curved, until they either reach their maximum velocity, so the trajectories tangent becomes constant, or they need to slow down for another vehicle in front of them, which can be seen as another trajectory above with a smaller derivative. This results in convoys of multiple vehicles moving with the velocity of the slow vehicle in front. These can be seen as broad stripes in the graph and since vehicles cannot overtake, the slow vehicle also has a great influence on the velocity-density-diagram, which can be seen in Figure 6.4. The trajectories of the middle road section continue to have those broad stripes since the slow vehicles cannot drive off the road and neither can the faster vehicles overtake. At the end of the third road section vehicles reach their destination, so they are removed. This gives faster vehicles behind them the opportunity to accelerate again, which cannot be seen in the graph though because the gap between the vehicles is so small that they barely accelerate until they already reach the end of the road of their own. Comparing the trajectories of the first road section that was simulated with the Gipps model no differences to the intelligent driver model can be identified simply by looking at those graphs.



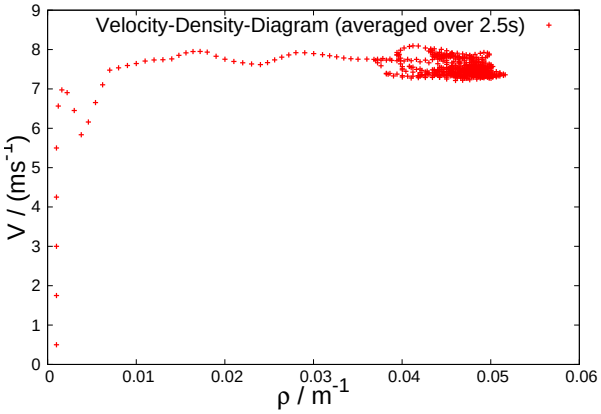
(a) Intelligent driver model: Starting lane.



(b) Intelligent driver model: Middle lane.



(c) Intelligent driver model: End lane.



(d) Gipps model: Starting lane.

Figure 6.4.: Velocity-density-diagrams of three road sections in a curvy scenario simulated with Gipps model and intelligent driver model. Data was averaged over 2.5 s.

The velocity-density-diagrams from Figure 6.4 do not look like expected if they are compared to Figure 3.1. At low densities ρ the average velocity ranges from low to high velocities. This is because there are low vehicle densities when there are few vehicles on the road. In this simulation this is at the beginning of the simulation when there is only one vehicle. While it accelerates it gains higher velocities, but the vehicle density remains the same. The more vehicles there are, the higher the density, and the less velocity variance. At higher vehicle densities between $0.03\,{}^{\text{Vehicles}}/_{\text{m}}$ and $0.06\,{}^{\text{Vehicles}}/_{\text{m}}$ there are numerous data points at equal densities with different average velocities. This happens at the point in the simulation when there are almost as many vehicles entering a road section as there are vehicles leaving it. The average velocity is governed by the slow vehicles that were mentioned above. There seems to be a plateau of the average velocity at about $8\,{}^{\text{m}}/_{\text{s}}$, which may be caused by the maximum velocity being set randomly at $(8 \pm 5)\,{}^{\text{m}}/_{\text{s}}$ for each vehicle.

6.3 Interpolating Vehicle States

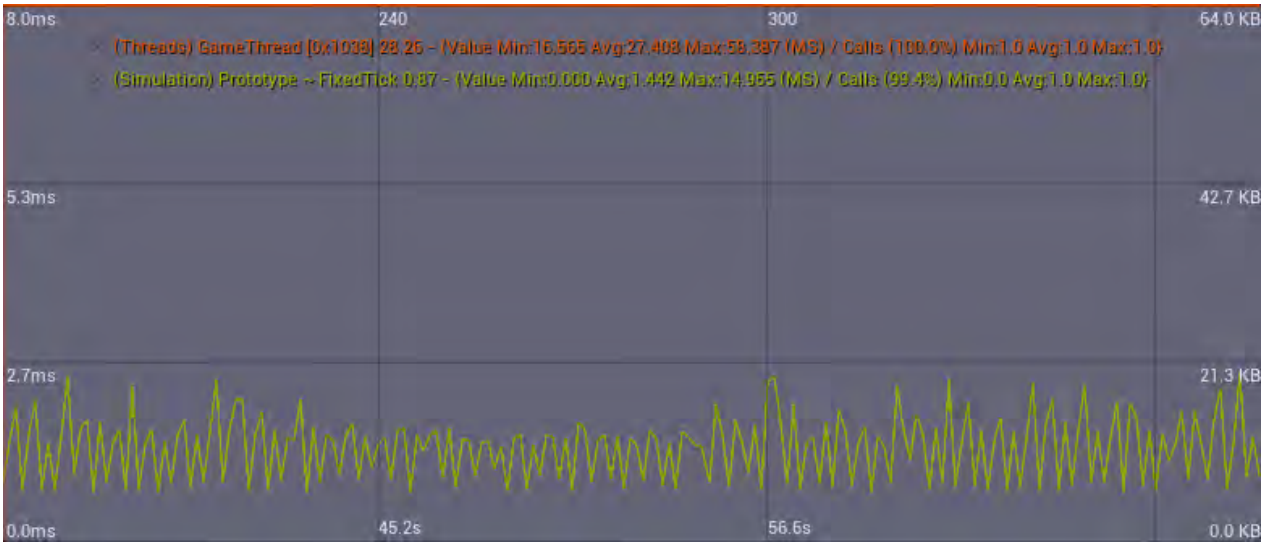


Figure 6.5.: Profiler graphs of 8 km of curvy road with about 450 vehicles show frame times (y-axis) of simulation (dark yellow) over the game time (x-axis). With short simulation steps of 16.7 ms vehicles are simulated almost every frame. Since frame times vary over time the average of 1.442 ms each frame is used for comparisons.

The aim of any interactive application with a fully rendered 3D world is to meet the users' requirements of smooth movements, thus high frame rates of 60 frames per second are desired. Simulating about 450 vehicles on 8 km of curvy road with 16.7 ms steps took an average of 1.442 ms each frame. This can be extrapolated to an absolute maximum of 5200 vehicles being simulated, not considering moving their 3D models. To be able to simulate more vehicles at a high frame rate and also display them in the 3D world the vehicle's state is not simulated every frame, but rather every 0.5 s and interpolated for every frame in between, which is performing much faster. Pure simulation times of 260–290 vehicles will then be around 0.558 ms as can be seen in Table 6.1. Even with 2000 vehicles the average frame time for simulation functions is only around 0.562 ms as can be seen in Table 6.7 and Figure 6.17. Interpolating between two states of 2000 vehicles only takes 0.150 ms on average in that scenario.

6.4 Performance of Vehicle Recognition

In a city scenario with few vehicles, finding the leading vehicle by sorting all vehicles that are on a lane, finding a vehicle's index and selecting the next vehicle as leading vehicle had an acceptable performance

Figure 6.6.: Profiler graphs of a curvy scenario with about 2000 vehicles on 32 km of road with 500 m sections show frame times of (from top to bottom): All game logic added up (yellow), simulation (orange spikes), which includes finding leading vehicle (green) and finding intercepting vehicles (yellow). With all game logic taking an average of 16.379 ms per frame the game performs very smoothly.

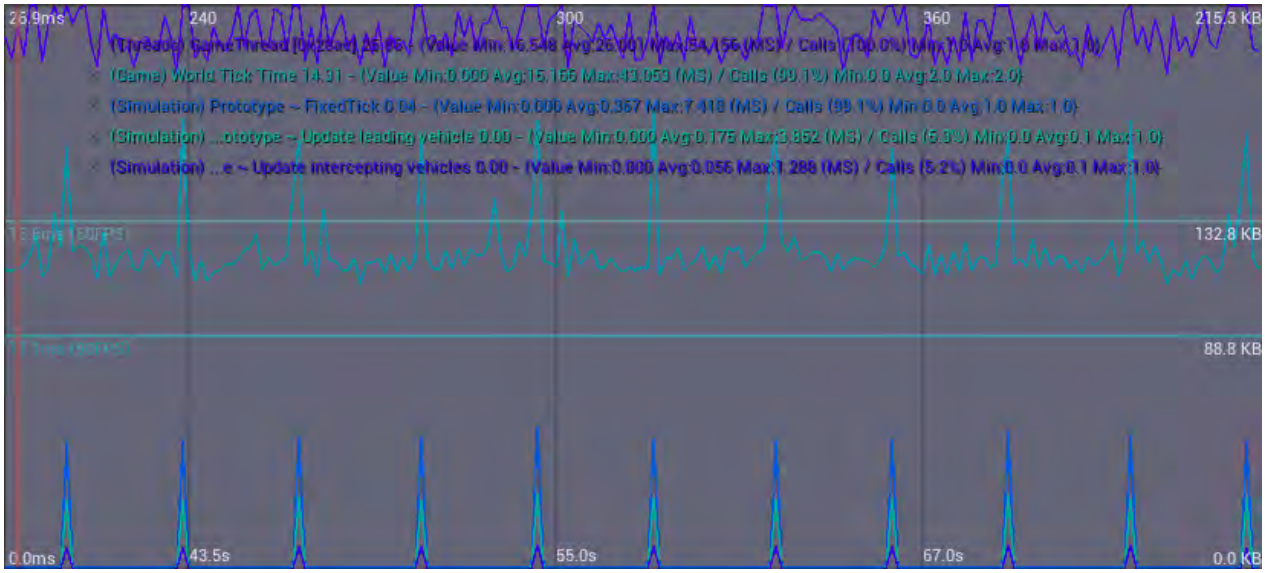


Figure 6.7.: Profiler graphs of a curvy scenario with about 2000 vehicles on 32 km of road with 1 km sections show frame times of (from top to bottom): All game logic added up (light blue), simulation (blue spikes), which includes finding leading vehicle (blue) and finding intercepting vehicles (dark blue). With all game logic taking an average of 15.166 ms per frame the game performs even better than it does with shorter road sections.

Figure 6.8.: Profiler graphs of a curvy scenario with about 2000 vehicles on 32 km of road simulated with improved sorting show frame times of (from top to bottom): All game logic added up (orange), simulation (violet spikes), which includes finding intercepting vehicles (cyan spikes), finding leading vehicle (green spikes) and acceleration calculation (purple spikes). The simulation performs better than it does with the old method of sorting but with an average of 18.142 ms per frame all game logic added up performs worse.

that allowed smooth rendering. When bigger cities were tested with more than 100–200 vehicles frame rates dropped so much that the simulation was visibly lagging. As an improvement, the sorted list was cached, so it was sorted until a vehicle left or drove onto the road. Sorting almost sorted lists was performing well enough with $(50 - 100)$ m long roads, i.e. 10–20 vehicles. Letting vehicles drive in a *curvy* scenario where there are multiple 1 km long roads with 80–100 vehicles on each lane increased the time it took to sort drastically, which was visible by frame rates of about 5 FPS. Maintaining sorted lists by always removing vehicles from the front, adding new vehicles at the end, and keeping an index stored for each vehicle improved the simulation enough to support thousands of vehicles in a curvy scenario instead of a few hundred.

	Average Frame Times		
Simulation Part	500 m sections (old)	1 km sections (old)	1 km sections (new)
`FixedTick`	0.367 ms	0.367 ms	0.315 ms
Update Leading Vehicle	0.134 ms	0.175 ms	0.062 ms
Update Intercepting Vehicles	0.063 ms	0.056 ms	0.072 ms

Table 6.2.: Frame times of finding leading vehicle in curvy scenarios with 32 km of road in sections of different lengths and the number of vehicles capped at 2000. *Old* columns show times where vehicle lists are sorted whenever requested, *new* column shows lists being kept sorted when elements are added or removed.

Table 6.2 shows frame times of curvy scenarios with different road section lengths where the lists of vehicles is sorted every time they are requested and one with the lists being kept sorted as a comparison. Comparing 500 m road sections to 1 km road sections an increase in frame time for finding the leading vehicle can be seen, which was expected. Unreal Engine 4 uses quicksort to sort its internal data structures, which performs in $\mathcal{O}(n \cdot \log(n))$ so more shorter lists should perform better than less bigger lists

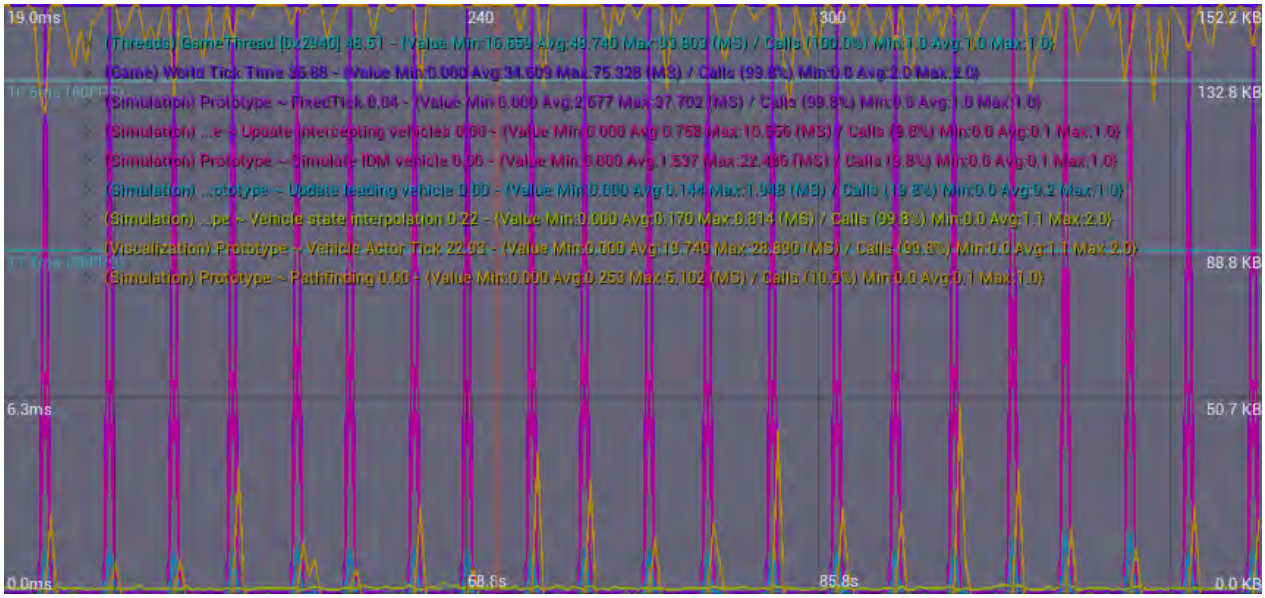


Figure 6.9.: Profiler graphs of a ten times ten city grid with about 2000 vehicles on 100 m road sections show frame times of (from top to bottom): All game logic added up (blue purple), vehicle actors (orange), simulation (purple spikes), which includes calculating acceleration (pink), finding intercepting vehicles (pink) and finding leading vehicle (light blue), path finding (orange), interpolating simulated location (yellow). With the simulation taking as long as all other game logic does in frames where there are no vehicles simulated, the game would run smooth with a lag spike every 0.5 s.

if the total number of items stays the same. Comparing 1 km road sections to the same scenario with vehicle lists being kept sorted the latter performs notably better, almost three times. Keeping lists sorted adds operations when adding and moving vehicles to and from lanes, which is accounted to `FixedTick`. An increase would be expected but a decrease in frame time can be seen, which cannot be explained else than it coming from the randomness of vehicles' distribution on the roads since all operations other than sorting vehicles were identical. It is also notable that frame time of finding intercepting vehicles increases in the third scenario, which also has to be attributed to the randomness of vehicles' distribution. Between the two scenarios with different road lengths finding intercepting vehicles also performs differently, which is not expected. Every simulated step, every vehicle requests a list of lanes that lead to the lane they will travel on after they pass the next intersection on their paths without their current lane, which will be an empty list in the curvy scenario. Therefore the frame time should either depend on the number of vehicles in that scenario, which is not the case here, or on the number of intersections because with frame times as short as these the difference may come from the list that keeps a list of vehicles for every lane having twice as many items.

In a city scenario vehicles do not only have to accelerate according to their leading vehicle but also consider intercepting vehicles they might collide with at intersections. Figure 6.9 shows the profiler result of a city scenario being simulated with the number of vehicles being capped at 2000, Table 6.3 shows measured frame times. It can be seen that identifying all intercepting vehicles takes longer than identifying the leading vehicle. All roads that go into the intersection have to be checked for all vehicles that will drive onto the same road, which takes longer than just checking one road for the next vehicle. Frame time of acceleration calculations in a curvy scenario should be comparable to a city scenario with the same number of vehicles. Finding a leading vehicle performs better, presumably because there are less intersections and longer road sections so less often the next road on a vehicles path has to be searched. This also means that the gap between two vehicles less often spans across multiple lanes,

Simulation Part	Average Frame Times Intercepting (100 m)	No Intercepting (100 m)	Intercepting (500 m)	Curvy
`FixedTick`	2.677 ms	0.348 ms	1.306 ms	0.315 ms
Update Leading Vehicle	0.144 ms	0.108 ms	0.077 ms	0.062 ms
Update Intercepting Vehicles	0.758 ms		0.367 ms	0.072 ms
Total Acceleration Calculation	1.537 ms	0.077 ms	0.726 ms	0.043 ms

Table 6.3.: Frame times of vehicle recognition in city scenarios with 100 m road sections and the number of vehicles capped at 2000. With disabled recognition of intercepting vehicles lane-changes were still working just like before, except for vehicles crashing into each other when turning into the same road.

which explains the slightly better performing acceleration calculations. The latter takes most of the simulation's time in a city scenario, so it has to be looked at more closely.

Figure 6.10 shows the profiler result of the same scenario without having vehicles check for intercepting vehicles at intersections. Measured frame times are also included in Table 6.3. They show that most of the time needed for calculating a vehicle's acceleration comes from assessing whether to brake for intercepting vehicles or not. This can be attributed to relatively high vehicle densities leading to a lot of vehicles needing to be checked for interception and then checking for each of those vehicles if the current vehicle needs to brake. When road sections become longer, finding a leading vehicle also performs better in a city scenario. The profiler result for that scenario can be seen in Figure 6.11. The explanation for that has to be different, though, because there are not less intersections, only longer road sections thus a higher total length of the road network, so the traffic simulation prototype seems to perform better in those situations.

To accelerate updating intercepting vehicles and acceleration calculation for them only the first few found vehicles could be considered. This would reduce the number of calculations if there are for example 20 vehicles driving from one other road onto the same road and the current vehicle already has to brake for the second one, making the next 18 acceleration calculations obsolete. This could lead to problems though if the current vehicle does not have to brake for those first few vehicles because they drive fast enough and the gap to the last one is big enough, but by the time the next vehicle is considered it may be too late to brake in time to avoid a collision. A more secure possibility is to only consider intercepting vehicles if the current vehicle will not be able to come to a halt at the intersection. This would greatly accelerate calculations if there are more intercepting vehicles than one since an additional acceleration calculation has to be made despite there possibly being no intercepting vehicles at all. Calculating if a vehicle is able to come to a complete halt at a given distance is also not as expensive in frame time as calculating its acceleration. A great way of optimizing finding intercepting vehicles at high densities is to cache additional lists of vehicles having a certain lane as the next or second to next one in their path. This would be calculated when the first vehicle that will drive into that lane is requesting intercepting vehicles and can be used for all other vehicles that will drive there in the same frame, greatly reducing redundant calculations. Possibly the most effective way of optimizing frame times for lane changes is to introduce real traffic rules like giving way to vehicles coming from one's right, coming from ahead when turning left or simply traffic lights. This will increase the amount of data being stored for each intersection and road, namely spatial information, and adds the need for traffic light circuits, which may reduce traffic flow. These rules also might not fit into the game's setting, where a *first come, first served* habit may be fitting better.

Figure 6.10.: Profiler graphs of a ten by ten city grid with about 2000 vehicles on 100 m road sections with disabled recognition of intercepting vehicles show frame times of (from top to bottom): All game logic added up (green), vehicle actors (cyan), simulation (violet), which includes finding leading vehicle (light orange) and calculating acceleration (cyan), path finding (orange), interpolating simulated location (blue). Although simulation takes much less time than it does with enabled lane-changing model, all game logic added up is not performing well enough with an average of 39.234 ms per frame.

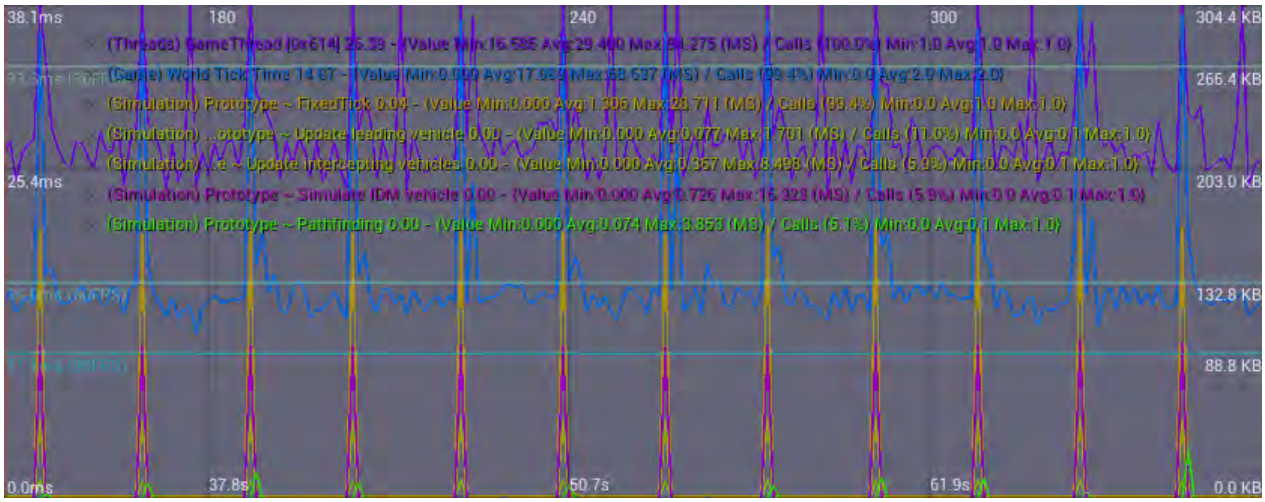


Figure 6.11.: Profiler graphs of a ten by ten city grid with about 2000 vehicles on 500 m road sections show frame times of (from top to bottom): All game logic added up (blue), simulation (orange), which includes acceleration calculation (pink), finding intercepting vehicles (yellow) and finding leading vehicle (yellow green), path finding (green). On longer roads simulation performs better. Performing better in all game logic than it does with disabled lane-changing model can be attributed to distance culling disabling more vehicles because their distance to the camera is larger.

Figure 6.12.: Profiler graphs of a five by five city grid with about 370–390 vehicles on 100 m road sections show frame times of (from top to bottom): All game logic added up (light blue), vehicle actors (blue), simulation (light green), which includes calculating acceleration (violet), finding intercepting vehicles (aligned blue green spikes) and finding leading vehicle (pink), path finding (offset blue green spikes, interpolating simulated location (blue green line). With an average of 7.314 ms per frame for all game logic, the simulation performs very well in this scenario, due to the low number of vehicles.

6.5 Effect of Path Finding on Performance

Whenever a new vehicle is spawned it chooses a random building as its destination. The aim of any vehicle is to reach their destination as quick as possible, so it has to find the shortest path. If there is much traffic on the roads the shortest path does not have to be the fastest, so expected travel times may be introduced for path finding. To find the effect of path finding and the calculation of travel times on the simulation's performance, up to 2000 vehicles were simulated in city scenarios of different sizes.

	5x5 Frame Times		10x10 Frame Times		15x15 Frame Times	
Simulation Part	average	maximum	average	maximum	average	maximum
`FixedTick`	0.144 ms	4.676 ms	2.677 ms	37.702 ms	1.534 ms	28.913 ms
Path Finding	0.019 ms	2.014 ms	0.253 ms	6.102 ms	0.578 ms	19.054 ms

Table 6.4.: Frame times of path finding in different city scenarios, each with 100 m road sections. The small scenario only had enough roads to simulate 370–390 vehicles while the other two were capped at 2000 vehicles.

The maximum frame times of path finding which are listed in Table 6.4 grow with the size of the road network. This is not entirely due to the greater distances and therefore higher number of road sections that have to be searched and found but mostly because it is more likely for two vehicles to spawn in the same frame. In Figure 6.9 the smallest spikes of the path finding graph all have approximately the same height, which is one vehicle being added and searching for the shortest path. The next bigger set of spikes approximately has twice the height – two vehicles being added. The average frame times are more comparable for this purpose. The large city has $1.5^2 = 2.25$ times the number of roads of the

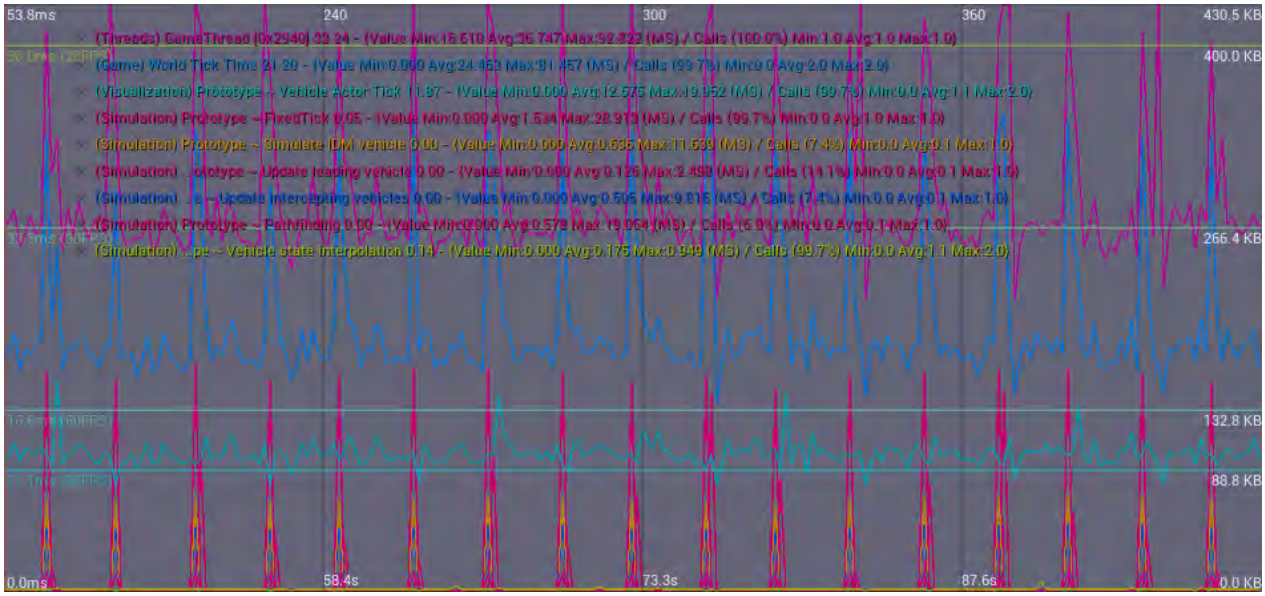


Figure 6.13.: Profiler graphs of a 15 by 15 city grid with about 2000 vehicles on 100 m road sections show frame times of (from top to bottom): All game logic added up (blue), vehicle actors (cyan), simulation (pink), which includes calculating acceleration (orange), finding intercepting vehicles (blue) and finding leading vehicle (pink), path finding (offset pink spikes), interpolating simulated location (yellow). In the large city simulating all vehicles in one frame takes as much time as all other game logic, which causes lag spikes. Path finding spikes are almost the same as well.

middle sized one and its average path finding frame time is close to 2.25 times the average frame time of the middle sized city. The path finding spikes are at a slight offset to the simulation spikes because vehicles are spawned from buildings' tick functions at random time intervals. When the maximum number of vehicles is reached no new vehicles are spawned until another vehicle reaches its destination and is removed. Removal happens in `FixedTick` thus spawning new vehicles occurs in the same frame or – more often – in the next frame. It is also interesting to note that `FixedTick` takes longer in the middle sized city than it takes in the large scenario. This may be caused by 2000 vehicles distributing themselves to more roads, so less vehicles are on a road section on average and thus less vehicles have to be considered for finding leading vehicles and intercepting vehicles and for acceleration calculations for the latter.

	With Expected Travel Time		Length Only	
Simulation Part	average	maximum	average	maximum
`FixedTick`	2.677 ms	37.702 ms	3.218 ms	51.222 ms
Path Finding	0.253 ms	4.411 ms	0.114 ms	2.014 ms

Table 6.5.: Frame times of path finding with and without expected travel time in a ten by ten city scenario with 100 m road sections with vehicles capped at 2000.

Table 6.5 compares path finding of the ten by ten city scenario with calculation of expected travel times to the same scenario only considering length of roads, which were taken from Figure 6.14. It can be seen that frame times of path finding more than double when expected travel times are calculated but the simulation performs better because traffic jams and gridlocks are avoided. Simulation frame time without expected travel times is mostly caused of predicting gaps to intercepting vehicles at the intersection where they collide, which may be due to handling special cases if one or both vehicles

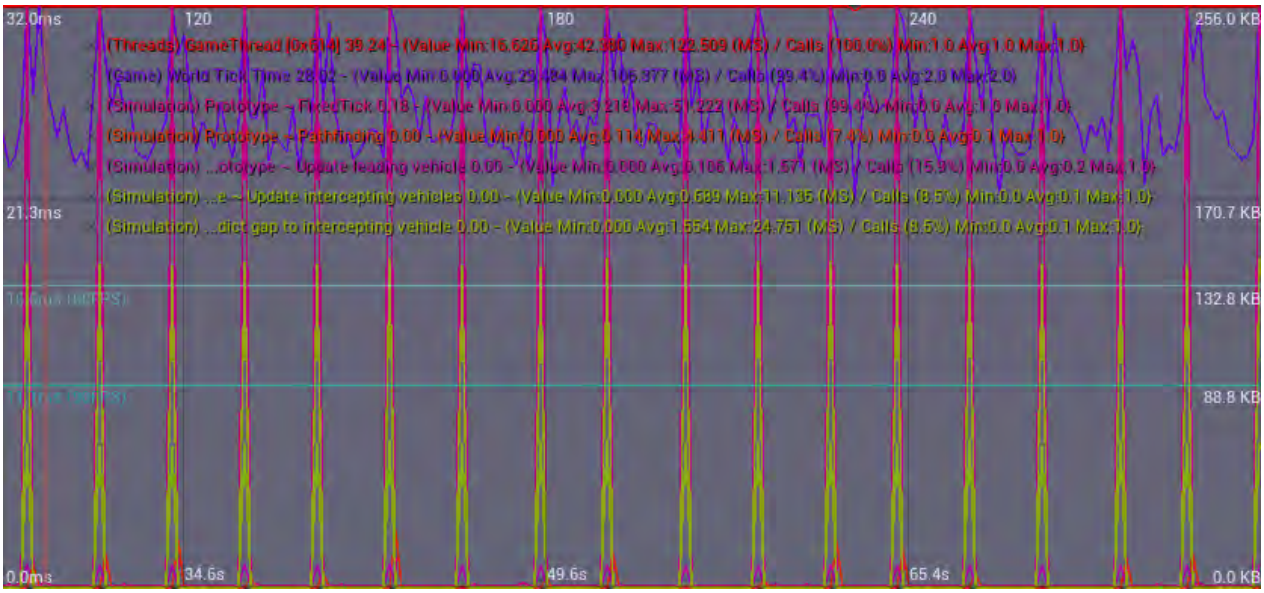
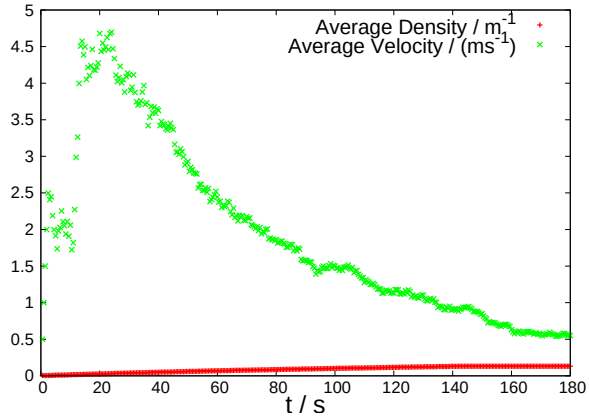
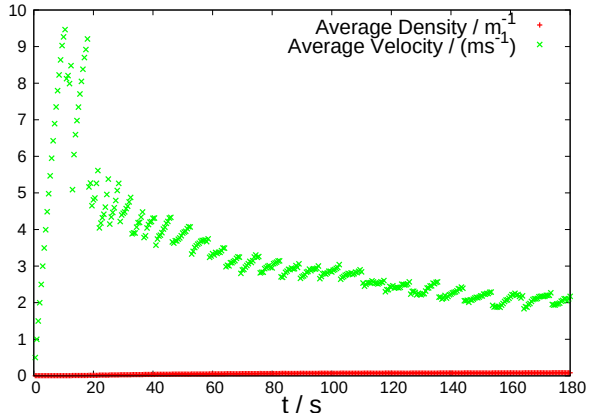


Figure 6.14.: Profiler graphs of a ten times ten city grid with about 2000 vehicles on 100 m road sections and path finding only considering road lengths show frame times of (from top to bottom): All game logic added up (violet), simulation (pink spikes), which contains predicting gaps (yellow), finding intercepting vehicles (yellow green) and finding leading vehicle (dark pink), path finding (red).

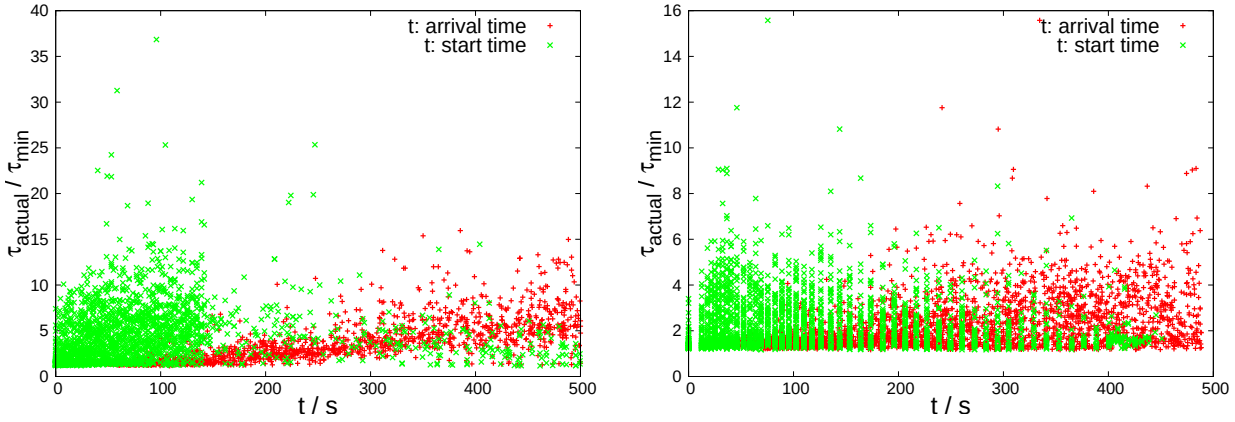


(a) Length only.



(b) With expected travel time.

Figure 6.15.: Average velocity of all vehicles and average density of lanes with vehicles in a ten by ten city scenario simulated with intelligent driver model. With expected travel times average velocities are generally higher and approach approximately 2 m/s as opposed to 0.5 m/s with length-only path finding. The jumps in the graph are caused by the simulation being accelerated at thirty times real time so all buildings spawn vehicles at the same time after tens of seconds being simulated.



(a) Length only. (b) With expected travel time.

Figure 6.16.: Relative difference of actual travel times τ_{actual} to minimum travel times τ_{min} in a ten by ten city scenario simulated with intelligent driver model. Using expected travel times for path finding results in smaller deviations because traffic jams are avoided by new vehicles, also more vehicles actually reach their destination.

do not move. Other parts are in the range of the simulation with expected travel times. Considering expected travel times for path finding also has an effect on average velocities and actual travel times as can be seen in Figures 6.15 and 6.16. If vehicles do not actively avoid traffic jams more vehicles will be caught up in them so average velocities decrease. As soon as traffic jams form, gridlocks can form as well, which causes vehicles that spawn after a gridlock was formed not to be able to reach their destination. If there is no gridlock vehicles will move very slowly nonetheless causing actual travel times to be a multiple of minimum travel times, which are calculated by assuming a vehicle always moves at its maximum velocity. Having vehicles not reach their destination also results in less vehicles being spawned if the number of vehicles is being capped so path finding has to be called less often, which explains path finding being twice as fast if only road lengths are considered.

Avoiding traffic jams and especially gridlocks will not only help vehicles reaching their destination faster or at all but will also help to avoid unnecessary calculations for intercepting vehicles. Having vehicles reevaluate their paths every once in a while will further help to reduce traffic jams and guarantee their arrival but at the cost of additional frame time for path finding.

6.6 Comparison of Simulating Data and Applying Results to Visualization

Frame times were measured with the Unreal Engine 4 profiler in a curvy scenario with approximately 2000 vehicles. The two buildings spawned vehicles, which drove on a 16 km long road to the other building. The road was separated into eight 1 km long segments, with a turn at the end of each segment. Continuous spawning of new vehicles every $(2-5)$ s and removal of the ones that reached their destination caused the number of vehicles to fluctuate. To reach that number of vehicles the simulation was running at an accelerated pace, simulating about 8 h of traffic, then turned to real-time, when approximately the same number of vehicles were spawned as reached their destination. Vehicles' movement was simulated in 0.5 s steps, their location linearly interpolated for every frame in between.

Figure 6.17 shows the output of the visual profiler, Table 6.6 the frame times of all objects (World Tick Time), `MoveComponent` and `FixedTick`. The game thread includes all game logic (World Tick Time), for example the simulation, movement of game objects, physics engine and game engine internal data structures for accelerated rendering. Since the scenario was run from within the editor, the game thread also includes display of editor and profiler user interface, so the World Tick Time is more suited for

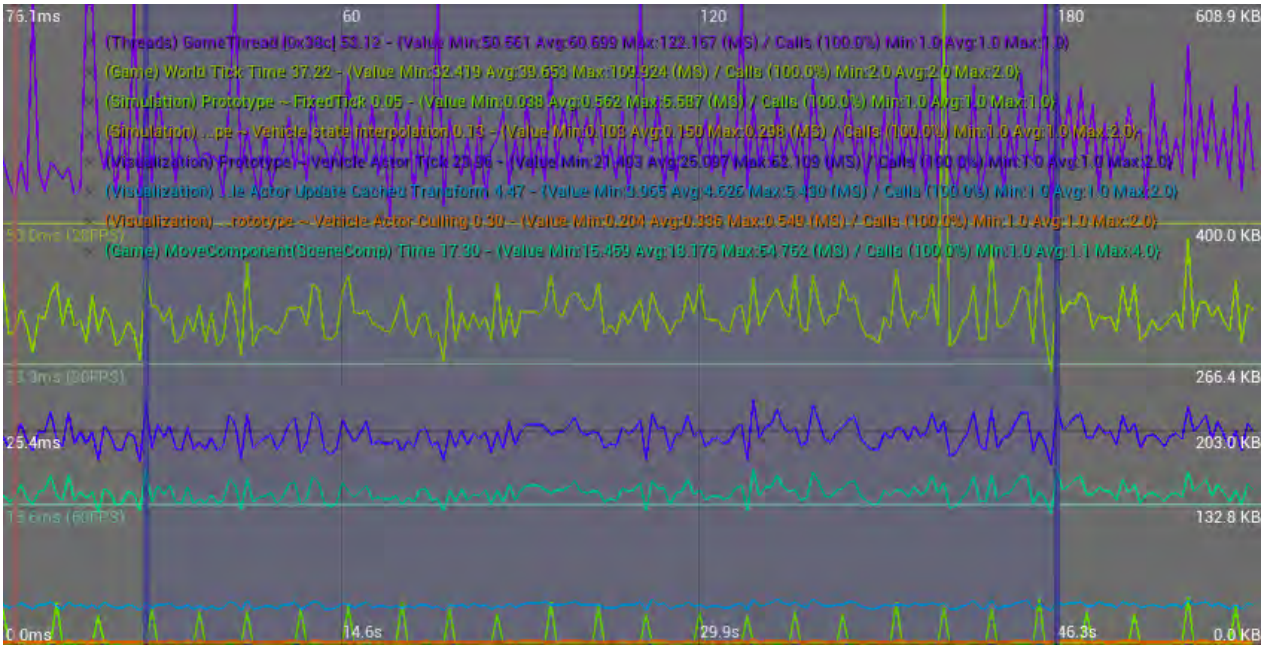


Figure 6.17.: Profiler graphs of 16 km of curvy road with about 2000 vehicles show frame times of (from top to bottom): All game logic added up (green), vehicle actors (purple), which includes moving actors (cyan) and calculating 3D location (light blue), simulation (green), interpolating simulated location (yellow). With `MoveComponent` taking an average of 18.176 ms each frame and the `FixedTick` taking a maximum of 5.587 ms every 0.5 s, hiding out-of-frame or far-away vehicles is far more efficient for this scenario than optimizing the simulation.

comparison to a real in-game situation. If the render thread takes longer than the game thread, the time difference is also included in the game thread, so it has to be subtracted to get a meaningful frame time. The frame time of `FixedTick` includes all calculations of the simulation itself, excluding visual representations and interpolation. The visual representation of vehicles, i.e. vehicle actors request a simulated location every frame, which is linearly interpolated between two simulated states. Interpolation only takes 0.150 ms per frame on average, so it is a good method to save frame time on the simulation. Simulated locations are then translated into a transform matrix with location and rotation in the 3D world, which takes 4.626 ms on average, which is about the same as simulating all vehicles once. When the new transform matrix is set for an object, `MoveComponent` is automatically called by the engine. It is used to update the transform matrices of the scene graph, collision data structures and physics poses. Vehicle actors' tick functions combine those, which is 25.097 ms on average.

The frame times show that in this scenario it is far more efficient to optimize `MoveComponent` calls than simulation logic. Even the maximum frame time of `FixedTick` is less than a third of the average frame time of `MoveComponent`. Since there is only one possibility at each intersection, i.e. turn for vehicles to drive to, there are never any intercepting vehicles, which reduces simulation frame times quite a bit. Intercepting vehicles neither have to be identified, nor possible collisions be found, nor resulting accelerations be calculated.

Reducing movement of objects seems to be a better solution non the less, for example if vehicles are obstructed by buildings or out of frame. A method seen in games (e.g. Cities: Skylines) is to only show all vehicles if the camera is close to the road, when only few vehicles are in frame. If the camera is further away from the road, some vehicles can be hidden to save frame time, some visible to convey movement of traffic. If the camera is very far away, all or nearly all vehicles are hidden, since the city's layout is more interesting than traffic at this point. This can also be applied to a tilted camera, where close and far-away objects are in frame. Hiding objects at a certain distance to the camera is called

Simulation Part	Average Frame Time	Maximum Frame Time
World Tick	39.653 ms	
`FixedTick`	0.562 ms	5.587 ms
Vehicle Actor Tick	25.097 ms	
Update Cached Transform	4.626 ms	
Culling Operations	0.336 ms	
`MoveComponent`	18.176 ms	

Table 6.6.: Frame times of different parts of the application with approximately 2000 vehicles driving on 16 km of road. `FixedTick` simulated movement of all vehicles every 0.5 s.

distance culling. To gradually thin out vehicles a random culling distance can be set for each vehicle within a range $[d_{\min}, d_{\max}]$, where all objects closer than $d_{\min}$ can be seen, all objects farther than $d_{\max}$ are hidden, and objects in between thin out depending on the function that gives random values.

Figure 6.18 shows the profiler result of the same scenario with active distance culling. Table 6.7 summarizes relevant frame times. There is an additional 0.909 ms on average for calculating vehicle actors' distances to the camera and deactivating or activating the objects. In both cases the transform matrix is calculated (4.626 ms without culling vs. 3.644 ms with culling), but the additional culling computations can reduce the `MoveComponent` time from 18.176 ms without culling to 1.969 ms with culling. With distance culling deactivated there is already an average frame time of 0.336 ms to check whether culling is active (orange graph at the bottom of figure 6.17). The differences in `FixedTick` and `MoveComponent`

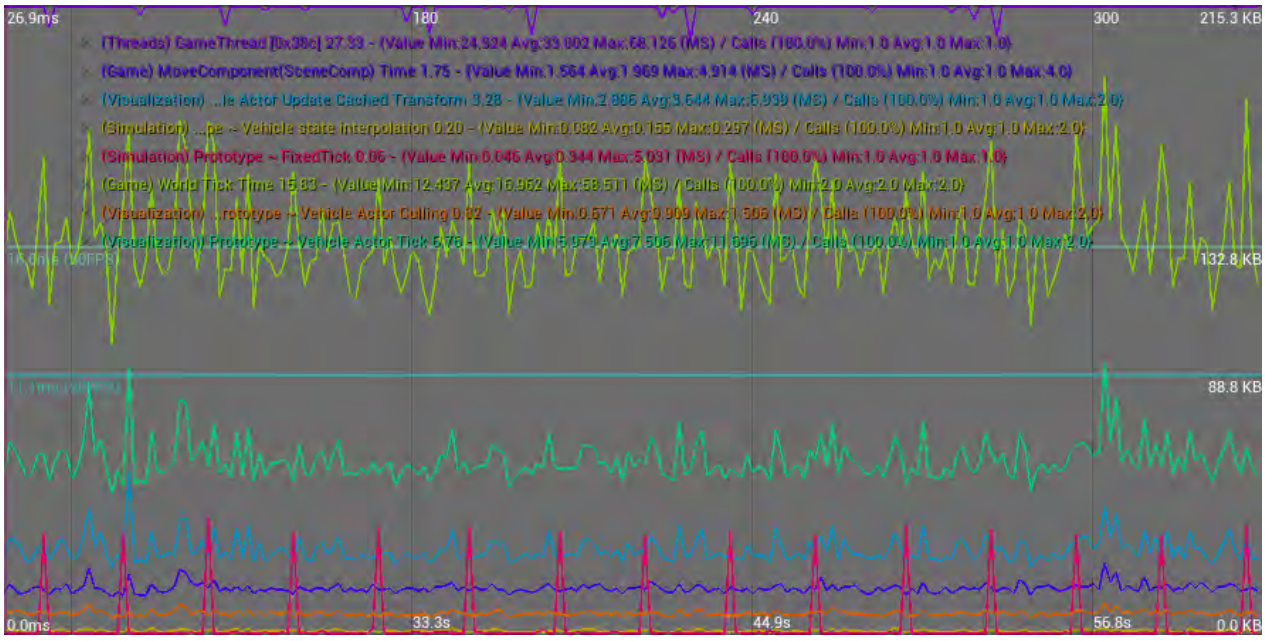


Figure 6.18.: Profiler graphs of 16 km of curvy road with about 2000 vehicles and active distance culling show frame times of (from top to bottom): All game logic added up (light green), vehicle actors (blue green), which includes calculating 3D location (blue), moving actors (purple) and performing distance culling (orange), simulation (pink), interpolating simulated location (yellow). Vehicle actors frame time is reduced to an average of 7.506 ms each frame, which is closer to the maximum of `FixedTick` (5.031 ms every 0.5 s), but still more than twenty times its average of 0.344 ms.

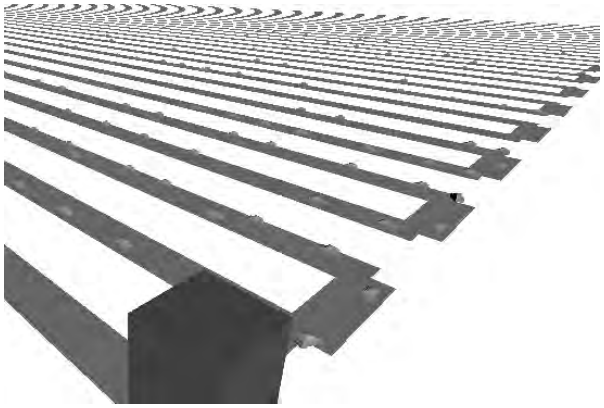


Figure 6.19.: Screenshot of 2000 vehicles in a curvy scenario with distance culling. Even displayed vehicles that are far away from the camera can barely be seen.

Simulation Part	Average Frame Time	Maximum Frame Time
World Tick	16.962 ms	
`FixedTick`	0.344 ms	5.031 ms
Vehicle Actor Tick	7.506 ms	
Update Cached Transform	3.644 ms	
Culling Operations	0.909 ms	
`MoveComponent`	1.969 ms	

Table 6.7.: Frame times of different parts of the application with approximately 2000 vehicles driving on 16 km of road and with active distance culling. `FixedTick` simulated movement of all vehicles every 0.5 s.

times can be attributed to the variance in number of vehicles. In this scenario about twenty vehicles were still visible with activated distance culling.

Figure 6.20 shows the profiler result of a simulation of the same scenario, where distance culling was switched off in the middle of the measurement. Since the given average values are for the whole measurement, they cannot be compared directly. The graphs show significant jumps in frame times though, in an almost identical situation. There may be some vehicles spawned, some removed and all are moved over time, but those differences are much smaller than between the two other measurements. The x-axis changes because frame times grow, so the number of frames and thus the number of data points shrinks. The scale of times on the y-axis is the same for the whole graph.

6.7 Blocking Vehicles: Lane Changing Alternative for Gipps Model

As an alternative to lane changing models where vehicles look for other vehicles that they might collide with, *blocking vehicles* are created as virtual vehicles by any vehicle that enters an intersection. Those virtual vehicles are not simulated and cannot be seen by the player, only by simulated vehicles. They are put on all auxiliary lanes at the intersection that end in the same lane node the actual vehicle's lane on the intersection ends, which means all other possibilities vehicles might come from to enter the road the vehicle will leave the intersection on. See also Section 5.2.3.

Table 6.8 compares frame times of the same city scenario, once simulated with the intelligent driver model and its lane changing implementation with intercepting vehicles (Table 6.3, Figure 6.9), once simulated with the Gipps model and blocking vehicles (Figure 6.21). The total lane changing calcula-

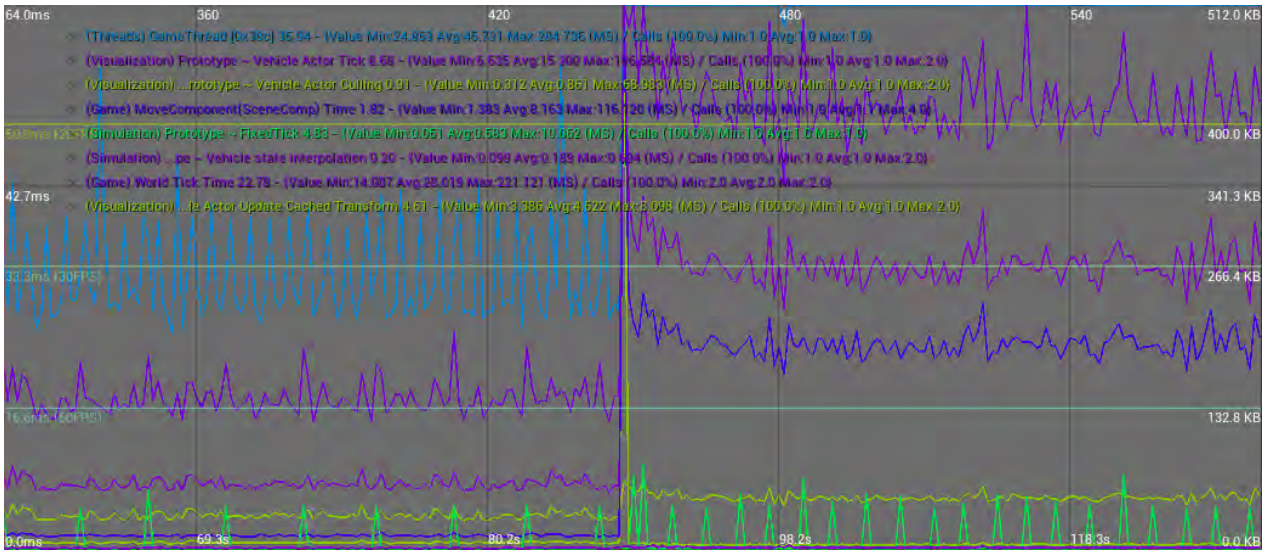


Figure 6.20.: Profiler graphs of 16 km of curvy road with about 2000 vehicles and distance culling switched off during measurement show frame times of (from top to bottom): All game logic added up (violet), vehicle actors (purple), which includes calculating 3D location (top light green), moving actors (dark purple) and performing distance culling (bottom light green), simulation (green), interpolating simulated location (bottom purple). Scale of x-axis changes after culling was switched off, but the graphs indicate relative frame time differences with nearly the same simulation environment.

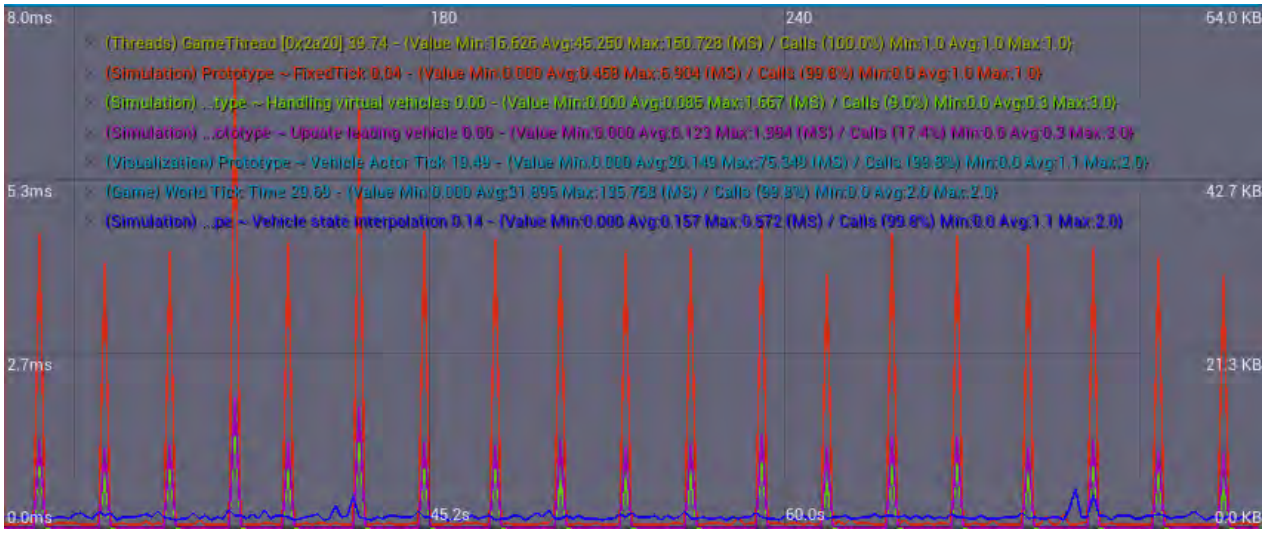


Figure 6.21.: Profiler graphs of a ten by ten city grid with about 2000 vehicles on 100 m road sections simulated with Gipps model and blocking vehicles show frame times of (from top to bottom): Simulation (orange), which includes finding leading vehicle (pink) and handling blocking vehicles (green), interpolating simulated location (blue). The impact on the frame rate of the 6.904 ms simulation spikes with blocking vehicles will not be as noticeable as the 37.702 ms simulation spikes with IDM lane-changing model.

Simulation Part	Average Frame Times IDM Intercepting Vehicles	Gipps Blocking Vehicles
`FixedTick`	2.677 ms	0.458 ms
Update Leading Vehicle	0.144 ms	0.123 ms
Update Intercepting Vehicles	0.758 ms	
Acceleration For Intercepting Vehicles	1.460 ms	
Handling Virtual Vehicles		0.085 ms
Total Lane Changing Calculation	2.218 ms	0.085 ms

Table 6.8.: Frame times of vehicle recognition in city scenarios with 100 m road sections and the number of vehicles capped at 2000. With disabled recognition of intercepting vehicles lane-changes were still enabled but vehicle crashed into each other when turning into the same road.

tion of the intelligent driver model simulation is composed of updating intercepting vehicles and total acceleration calculation time *without* intercepting vehicles – only car-following calculations – subtracted from total acceleration calculation time *with* intercepting vehicles. Acceleration calculations with blocking vehicles are the same as ones without them because blocking vehicles are seen just like any other leading vehicle. In that case the additional time comes from creating virtual vehicles and updating their positions as the same distance from the end lane node as the one from the actual vehicle that created them. Since the frame time of acceleration calculations with intercepting vehicles comes from two different measurements this is not an exact time, only an estimate. Update leading vehicle times which execute exactly the same operations regardless of intercepting vehicles varied by 25%. The Gipps model performed with only half the frame time for acceleration calculations compared to the intelligent driver model, which can be seen in Section 6.2. Taking this into account and constructing a worst case scenario blocking vehicles still perform almost ten times better than intercepting vehicles.

There is a downside to this alternative, though: Blocking vehicles are *not* collision free. For acceleration models to work correctly, vehicles cannot just appear in front of them, that is why there are dedicated lane changing models. Suddenly appearing vehicles might not leave other vehicles enough space to brake in time to avoid a collision. Having vehicles brake with their maximum deceleration at first, only stopping when they hit the virtual vehicle is still a better option than having vehicles overlap at intersections.

7 Discussion of Results

This chapter summarizes the work and findings of this thesis. Problems that were not obvious from the beginning will be highlighted and their need for future work discussed.

7.1 Simulation Properties

To analyze optimization possibilities in a traffic simulation for city builder games, existing simulation models for car-following and lane-changes were implemented with data structures that help optimizing performance. When lane-changing was introduced, several new issues arose that required additional checks, which caught minor errors to avoid crashing the whole simulation by better handling situations the models do not expect. The aim was to have a well performing simulation decoupled from its visual 3D representation to be able to simulate high numbers of vehicles that most of the time cannot be seen. To have simulated vehicle states that could be replicated on any machine with the same results a fixed simulation step was needed to be called from the Unreal Engine's tick function witch varying delta times. The simulation step lengths would have to be the same on each instance of the simulation to be able to produce equal results.

The Gipps model uses maximum deceleration only to calculate how long it will take a leading vehicle to come to a complete halt but not to cap a vehicles deceleration when braking, which can lead to unnatural looking braking maneuvers when lane changes are involved. This was circumvented by capping deceleration and only letting vehicles brake with higher decelerations when they hit their leading vehicle to avoid overlaps of vehicles. Since the Gipps model does not include lane changes in of its own blocking vehicles were introduced at intersections to signal to other vehicles that they might collide with a vehicle turning to the same road. Those virtual vehicles had better results than only detecting collisions when both vehicles moved onto the same road, possibly overlapping but not as good as knowing about possible collisions in time to brake early enough to avoid them.

Since the intelligent driver model does handle lane changes but assumes that possible collisions are known for any time in the future at all times, predicting those collisions emerged as a big part of the thesis. All vehicles that will drive onto the same road from all roads at an intersection were considered for possible collisions, which produced generally correct behavior. Some special cases, such as road sections being too short and velocities too fast, or both vehicles being close to the intersection at low velocities could result in them braking for each other and never driving on. A deliberate decision was to let other vehicles cross a vehicles path without taking them into consideration for braking, which leads to overlaps at intersections. This allowed the simulation to perform well enough to be able to be a part of a game, which was most important. For a player it is more important that a high frame rate ensures fluid motion, seeing vehicles accelerate, decelerate and follow each other correctly most of the time than it is not to have short moments of overlap of crossing vehicles at intersections. In a game a lot of components have to be weighed against each other to ensure best playability and entertainment for the player.

If path finding only considers spatial length of roads to find a shortest path, traffic jams can form at bottlenecks in the road network, so vehicles will take longer than expected to travel to their destinations. Traffic jams can lead to gridlocks, which in turn means less vehicles actually reach their destination. To solve this problem the Dijkstra algorithm was enhanced by expected travel times as additional cost, which had vehicles taking a detour at a certain point to avoid traffic jams. Having this additional cost for

road lengths ensured that travel times grew less as time went on and more vehicles were on the roads, more vehicles reached their destination at all and average velocities were generally higher.

7.2 Performance and Optimization

To find bottlenecks in the simulation's performance, traffic was simulated with varying settings in different scenarios. A straight road and a curvy scenario with lots of switchbacks were used to investigate car-following models with respect to road length, number of roads, number of vehicles and performance generally. A city scenario on a grid structure was used to analyze lane-changing models and path finding. City scenarios generally performed worse because path finding and lane changes actually were relevant in contrast to them not really weighing in in curvy or straight scenarios. Leading vehicles also had to be looked for on the next lane in a vehicles path more often than only on its current lane.

First measurements revealed vehicle recognition to be far worse performing than calculating the actual mathematical equations from the implemented models. Finding better ways to identify a leading vehicle and intercepting vehicles proved better for optimization than developing a slightly better performing model equation. The intelligent driver model was found to be worse performing in its acceleration calculations than the Gipps model, which does not describe lane changes, though. Since acceleration also seemed less abrupt in the intelligent driver model than in the Gipps model it was used as the intended model for the simulation from there on. Accelerations were calculated fast enough to be able to let them be calculated every frame or at least at a fixed rate which would be close enough to the desired frame rates but identifying leading and intercepting vehicles and checking whether to brake for the latter take too long, so having a long simulation step length and linearly interpolating vehicle states proved to reduce average frame times many times over while the effects of interpolation on vehicles' locations and velocities cannot be noticed at a first glance.

Identifying a leading vehicle could already be optimized to perform almost three times as fast if lists with vehicles on a lane were kept sorted instead of sorting and caching them every time they are requested. This optimization also removed the dependence of the length of road sections. Intercepting vehicles for lane changes were found to have a huge impact on performance depending on the length of road sections and the number of roads at intersections. Frame times of acceleration calculations were inflated by intercepting vehicles almost twentyfold because until a vehicle realizes that it has to brake with its maximum deceleration an acceleration has to be calculated for every intercepting vehicle.

Dijkstra's algorithm was implemented for vehicles to find the shortest path from the building where they spawn to their destination. In addition to a road section's length, the expected travel time was used as cost for path finding, which helped preventing traffic jams to develop to a complete gridlock. The relation of actual travel times to minimum travel times also decreased when vehicles avoided traffic jams known at the time they look for the shortest path and the number of vehicles actually reaching their destinations also increased. As one would expect, frame times of path finding are higher if expected travel times are considered, which cannot be solely attributed to the fact that additional calculations for expected travel times have to be made, though but also because less vehicles are being spawned. A side effect of vehicles taking longer paths to avoid traffic jams is them sometimes driving into dead ends at an intersection just to turn around at the end and take a turn at the same intersection again instead of driving straight the first time. This can occur when there is a traffic jam at the intersection on the lane that goes straight when the vehicle looks for the shortest path that may have resolved when it actually reaches the intersection.

The biggest impact on frame times was not found in the simulation component but in engine internal functions. Setting the location of an actor calls many of those functions, especially `MoveComponent` which is responsible for notifying children about the new location and setting a physics pose, which essentially means a 3D objects orientation in the world. The distance culling function of Unreal Engine only hides

objects but still calls those functions. To avoid this the location of actors was first stored in a cached property and only actually set for the actor if it is close enough to the camera. This made it possible to double the frame rate when only a fraction of the simulated vehicles are shown and updated in the 3D world.

Blocking vehicles which were developed as an alternative lane-changing model for the Gipps model performed much better than intercepting vehicle calculations with the downside of vehicles not noticing a potential crash at an intersection early enough and thus not being able to brake completely, followed by them stopping instantly.

7.3 Pitfalls of Implementation

The original aim of this thesis was to develop a new simulation model for acceleration and lane changes suited for use in a real-time gaming environment. A prototype simulation was implemented with existing models to better understand the essentials of traffic simulation software, but with first drafts only being able to simulate a few hundred vehicles at low frame rates where the mathematical formulas of the implemented models performed so well that they were almost invisible in the profiler, it soon became clear that there are parts of the simulation that were better suited for optimizations than the calculation of accelerations itself.

Using Unreal Engine 4 as existing software that provides most features that are needed for basic game programming seemed to be a choice that would accelerate development by a lot. In the end it became clear that although lots of components did not have to be implemented from scratch, which actually did accelerate development, getting used to the engine and its inner working took more time than initially thought. A great help were the online documentation[1] with a question hub and forums where all kinds of problems could be found already answered by the community, also being able to access the engine's source code[2] helped with more complicated problems that were intertwined with the engine's implementation specifics, for example to find out which sorting algorithm is used or how to add functionality to the editor to be able to create road networks. Using existing software also restricted the ability to completely implement features in the way they are intended and calls for workarounds more often than wished for. The Unreal Engine's own garbage collection system sometimes deleted objects that were still in use but not specifically marked to be not deleted, which caused crashes due to access violations. Usually a garbage collector will not delete an object if it still can be accessed by other objects, though. The provided distance culling feature only hides objects. Performance heavy operations like calculating physics poses were still executed, even though this is not necessary for objects that are not rendered. Finding the source of the unwanted frame time was counter-intuitive because this *physics* operation was executed with physics being disabled. Overlap calculations are another operation which unexpectedly appeared in the profiler. It was not caused by *collision* calculations which were disabled, but by calculations for events that fire when one object overlaps another object on the screen, which could be disabled by finding a corresponding bool variable.

Without having traffic lights or specific rules for the right of way it is very difficult and especially expensive to keep the simulation collision-free at intersections. At first, all special cases that can occur while turning were tried to be caught. Also vehicles should notice sooner whether they would have to brake to avoid collisions. When vehicles still drove through each other after turning into the same road at an intersection it was decided to add a fall-back which would at least look more pleasing whenever a mistake could not be caught. Usually vehicles' braking capabilities are capped. The limit will only be exceeded when they actually collide or overlap with another vehicle, keeping them behind the other

[1] https://docs.unrealengine.com/latest/INT/
[2] https://github.com/EpicGames/UnrealEngine

vehicle at half its velocity. This made it possible to focus on optimizing frame rates and usability for its intended purpose of a simulation within a game.

Vehicles were not simulated in any specific order so it had to be made sure that a vehicle would only use information that was available for each vehicle at the same simulation time. In particular this means that a vehicle actually does not know where its leading vehicle is at the simulation time its currently simulating but only the previously simulated step. Most of this information was kept in each vehicle object for the previously simulated step and the next simulated step so adding a simulation time to getter functions and returning the appropriate state's information was enough for those situations, information about which vehicles currently drive on which lane was stored in a simulation class, though. This made it necessary to defer adding and removing vehicles and updating the list of lanes with their current vehicles. For the visualization of frames in between the simulated ones it was also necessary to keep a list of lane changes that were made in between simulated states for each vehicle.

Most existing research of traffic simulation revolves around developing car-following models and lane-changing models that better replicate reality [5, 13, 15]. Those models often assume that all information is known about all vehicles at all times, which is theoretically possible to achieve in microscopic simulations but not at all trivial. For vehicles to drive onto another road it is necessary to know the vehicle that will be in front and the one that will be behind them on that road when they will have entered it to be able to assess whether it is safe to drive on that road or whether they need to brake. Finding eligible vehicles for this in a city scenario where *driving onto a road* means *turning at an intersection* alone took about a fourth of frame times of the simulation, let alone predicting gaps and calculating the resulting accelerations. To see the impact of the model equation on the implementation of a simulation, once all properties of a vehicle are found to fill out all variables of the equation, implementation took only five lines of code of the total of about 9000 lines.

7.4 Future Work

Replicating the simulation over network to other machines is an option given by Unreal Engine 4 in which the simulation was implemented. Whether the simulation has to be optimized for replication, for example reducing memory load if all objects are transferred over network as is or optimizing construction of objects if parameters for creation are transferred only could be analyzed in future work. Optimizations proposed in Chapter 6 can be implemented and analyzed for feasibility. Introducing traffic rules like traffic lights and right of way may be a way of optimizing lane changes may not be usable in all game scenarios, though. Better prevention of gridlocks is also an option to focus on in future work so players can concentrate on other parts of a game where micromanagement of traffic is not a part of the game's core mechanics. Reevaluating a vehicle's path every now and then for current traffic changes may be a solution, optimization of the path finding algorithm would be needed then, though. Using an informed algorithm like A^* with spatial distance as a heuristic or having landmarks that serve as a first iteration of path finding in the road network as proposed by [9] are options worth exploring. A survey of path finding algorithms revealed that dividing the problem into a smaller graph at a higher level to get a heuristic as does the approach with landmarks is most promising [8]. Vehicles will also have to find a new path if a road on their path was removed by the player, which should be minimized as much as possible since path finding for possibly hundreds of vehicles in one frame would be very costly for performance.

To make the performance measurements of a simulation more reliable and closer to its intended purpose in a game, fully implementing the features that were imitated by randomness is a good way to start, namely simulation of people who will drive their vehicles to and from work, buildings that produce resources that need to be transported and different types of vehicle with distinct properties. Implementation of these details was beyond the scope of this thesis.

Finally, distributing simulation of vehicles over multiple frames is the best way to reach the average frame times that were compared in Chapter 6 as has been mentioned by [1]. This will even out the spikes in frame time seen in profiler graphs, which are the cause of the game lagging every time all vehicles are simulated. Doing so will entail the problem of either not all vehicles being simulated at the same simulation times or the vehicles that are simulated last lagging behind, which would cause a delay of the whole simulation. The former can be approximately solved by linear interpolation, which was used in this thesis for vehicles' visualization.

8 Conclusion

In this thesis a traffic simulation was implemented with two different car-following models and a lane-changing model. Algorithms for finding a leading vehicle and intercepting vehicles were developed when no traffic rules are given at intersections or right of way is not guaranteed, they were implemented and optimized for faster calculation. As an alternative to lane-changing models blocking vehicles were proposed, also spatially shortest path finding was extended by expected travel times to avoid traffic jams and gridlocks. Furthermore the simulation's performance was analyzed for parts that can be further optimized and some optimizations were proposed. By implementing a prototype traffic simulation vehicle recognition algorithms, overhead caused by the game engine and unnecessarily simulating all vehicles in each frame were found to be causes for low performance, while implementations of model equations only made up a fraction of computation time.

Optimizations of algorithms lead to the following improvements:

- Finding leading vehicles could be improved to 35% of the original computing time where it made up 47.7 % of simulation time, afterwards making up 19.7 %.
- Implementing more appropriate distance culling could improve vehicle actor operations to 30% of the original computing time where it made up 63.3 % of all game logic, afterwards making up 44.3 %.
- Using the alternative to the lane-changing model could reduce lane-changing calculations to 3.8 % of computing time, which reduces simulation time to 17.1 %. The cost is worse driving behavior which vehicles have in contrast to the lane-changing model, though.

The best result was measured in a curvy scenario with distance culling. 2000 vehicles could be simulated and if close enough to the camera rendered with 60 FPS. The best result for a city scenario were 2000 vehicles with the alternative to lane-changing models being rendered at 30 FPS.

Handling lane changes can be improved by further selection of intercepting vehicles which can actually influence a vehicle's acceleration. Additionally distributing all simulation calculations from one frame every 0.5 s equally over that period could be able to simulate and render 20000 vehicles in a city scenario at 30 FPS.

While there is still room for improvement and future work, by implementing the prototype traffic simulation of this thesis several aspects of the simulation which are suitable for improvement were uncovered and options for improvement were proposed.

List of Figures

Bibliography

[1] Exploring simcity: A conscious process of discovery. http://www.gdcvault.com/play/1017948/Exploring-SimCity-A-Conscious-Process. Accessed: 2016-07-14.

[2] Game design deep dive: Traffic systems in cities: Skylines. http://www.gamasutra.com/view/news/239534/Game_Design_Deep_Dive_Traffic_systems_in_Cities_Skylines.php. Accessed: 2016-07-14.

[3] Gaffer on games – fix your timestep! http://gafferongames.com/game-physics/fix-your-timestep/. Accessed: 2016-11-02.

[4] Hussein Dia and Sadka Panwai. Nanoscopic traffic simulation: enhanced models of driver behaviour for its and telematics simulations. In *INTERNATIONAL SYMPOSIUM ON TRANSPORT SIMULATION, 8TH, 2008, SURFERS PARADISE, QUEENSLAND, AUSTRALIA*, 2008.

[5] Patrick AM Ehlert and Leon JM Rothkrantz. Microscopic traffic simulation with reactive driving agents. In *Intelligent Transportation Systems, 2001. Proceedings. 2001 IEEE*, pages 860–865. IEEE, 2001.

[6] Nathalie Farenc, Ronan Boulic, and Daniel Thalmann. An informed environment dedicated to the simulation of virtual humans in urban context. In *Computer Graphics Forum*, volume 18, pages 309–318. Wiley Online Library, 1999.

[7] Hans-Thomas Fritzsche and Daimler-benz Ag. Amodel for traffic simulation. *Traffic Engineering and Control*, 35:317–321, 1994.

[8] Liping Fu, D Sun, and Laurence R Rilett. Heuristic shortest path algorithms for transportation applications: state of the art. *Computers & Operations Research*, 33(11):3324–3343, 2006.

[9] Andrew V Goldberg and Chris Harrelson. Computing the shortest path: A search meets graph theory. In *Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 156–165. Society for Industrial and Applied Mathematics, 2005.

[10] Stefan Krauß. *Microscopic modeling of traffic flow: Investigation of collision free vehicle dynamics*. PhD thesis, 1998.

[11] Joaquin Maroto, Eduardo Delso, Jesús Félez, and Jose Ma Cabanellas. Real-time traffic simulation with a microscopic model. *Intelligent Transportation Systems, IEEE Transactions on*, 7(4):513–527, 2006.

[12] Robert Nystrom. *Design Patterns für die Spieleprogrammierung*. mitp, 2015. Übersetzt aus dem Amerikanischen von Knut Lorenzen.

[13] Craig W Reynolds. Steering behaviors for autonomous characters. In *Game developers conference*, volume 1999, pages 763–782, 1999.

[14] Jason Sewall, David Wilkie, and Ming C Lin. Interactive hybrid simulation of large-scale traffic. In *ACM Transactions on Graphics (TOG)*, volume 30, page 135. ACM, 2011.

[15] Tomoyoshi Shiraishi, Hisatomo Hanabusa, Masao Kuwahara, Edward Chung, Shinji Tanaka, Hideki Ueno, Yoshikazu Ohba, Makoto Furukawa, Ken Honda, Katsuyuki Maruoka, et al. Development of a microscopic traffic simulation model for interactive traffic environment.

[16] Martin Treiber and Arne Kesting. *Verkehrsdynamik und-simulation: Daten, Modelle und Anwendungen der Verkehrsflussdynamik*. Springer-Verlag, 2010.

[17] F Benjamin Zhan. Three fastest shortest path algorithms on real road networks: Data structures and procedures. *Journal of geographic information and decision analysis*, 1(1):69–82, 1997.

A Appendix

A.1 Source Code

A.1.1 Gipps Model Acceleration

```
void UGippsVehicle::SimulateStep(float DeltaTime)
{
  UpdateLeadingVehicle();

  // term b*ReactionTime from equation
  float BDT = ReactionTime * MaxDeceleration;
  float LeaderVelocity = INFINITY;
  if (LeadingVehicle)
  {
    LeaderVelocity = LeadingVehicle->GetVelocity(CurrentState.
       SimulationTime);
  }
  float Gap = GetGapToLeadingCar(CurrentState.SimulationTime);

  /// OVERLAP
  if (Gap < -GetLength())
  {
    UpdateLeadingVehicle();
    if (LeadingVehicle)
    {
      LeaderVelocity = LeadingVehicle->GetVelocity(CurrentState.
         SimulationTime);
      Gap = GetGapToLeadingCar(CurrentState.SimulationTime);
    }
  }
  /// END OVERLAP

  Gap = FMath::Max(Gap, -LeaderVelocity * LeaderVelocity / MaxAcceleration
      * 0.5f - MinimumDistance);

  // note: equation is different if leading vehicle has different
     MaxDeceleration
  float SafeVelocity = -BDT + FMath::Sqrt(BDT * BDT + LeaderVelocity *
     LeaderVelocity + 2 * MaxDeceleration * (Gap - MinimumDistance));
  CurrentState.SimulationTime += DeltaTime;
  float DesiredVelocity = FMath::Min3(SafeVelocity, CurrentState.Velocity
     + MaxAcceleration * DeltaTime, MaxVelocity);
```

```
  bIsTravelDistanceDirty = true;
  bIsRenderStateDirty = true;
  CurrentState.Velocity = DesiredVelocity;
  MoveForward(DesiredVelocity * DeltaTime);

  UpdateBlockingVehicles();
}
```

A.1.2 Intelligent Driver Model Acceleration

```
void UIDMVehicle::SimulateStep(float DeltaTime)
{
  UpdateLeadingVehicle();
  UpdateInterceptingVehicles();

  float Acceleration = GetMaxAcceleration();
  float AccelerationForLeadingVehicle = GetMaxAcceleration();
  float AccelerationForInterceptingVehicle = GetMaxAcceleration();
  UVehicle* NewAccelerationSource = nullptr;
  if (GetLeadingVehicle())
  {
    // calculate gap and desired gap
    float LeaderVelocity = GetLeadingVehicle()->GetVelocity(CurrentState.
       SimulationTime);
    float Gap = GetGapToLeadingCar(CurrentState.SimulationTime);

    Gap = FMath::Max(Gap, 0.0f);
    float DesiredGap = GetDesiredGap(GetVelocity(), GetVelocity() -
       LeaderVelocity);
    AccelerationForLeadingVehicle = GetAcceleration(DesiredGap, Gap);
  }
  else
  {
    // brake as if vehicle was standing at the furthest intersection
       considered for leading vehicle
    ULane* NextLane = GetNextLaneOnRoute();
    if (NextLane)
    {
      ULaneNode* FurthestNode = NextLane->GetEnd();
      TArray<ULane*> OutgoingLanes = FurthestNode->GetOutgoingLanes();
      if (OutgoingLanes.Num() == 1)
      {
        FurthestNode = OutgoingLanes[0]->GetEnd();
      }
      float Gap = GetDistance(FurthestNode, CurrentState.SimulationTime);
      float DesiredGap = GetDesiredGap(GetVelocity(), GetVelocity());
      AccelerationForLeadingVehicle = GetAcceleration(DesiredGap, Gap);
    }
  }
  if (bUseInterceptingVehicles && InterceptingVehicles.Num() >= 1)
```

```
{
  // check if intercepting vehicles would crash or have unsafe distance
  float MinPredictedLeadingGap = INFINITY;
  float MinPredictedFollowingGap = INFINITY;
  ULaneNode* MutualNode = GetCurrentLane()->GetEnd()->GetIncommingLanes
     ().Num() > 1 ? GetCurrentLane()->GetEnd() : GetNextLaneOnRoute()->
     GetEnd();
  for (UVehicle* Vehicle : InterceptingVehicles)
  {
    EVehicleOrder VehicleOrder;
    float PredictedGap = GetPredictedGap(Vehicle, MutualNode,
       VehicleOrder, CurrentState.SimulationTime);
    bool bGiveRightOfWay = GiveRightOfWay(Vehicle);
    bool bIsOnIntersection = IsOnIntersection();
    bool bOtherIsOnIntersection = Vehicle->IsOnIntersection();
    if (!bGiveRightOfWay &&
      ((LastAccelerationSource == Vehicle && Cast<UIDMVehicle>(Vehicle)
         ->LastAccelerationSource == this)))
    {
      // ignore other vehicle if both aren't moving or brake for each
         other
      continue;
    }

    if ((VehicleOrder == EVehicleOrder::FollowingVehicle && (
       bGiveRightOfWay || bOtherIsOnIntersection))
      || (VehicleOrder == EVehicleOrder::OverlappingFollowingVehicle &&
         (!bIsOnIntersection || bOtherIsOnIntersection)))
    {
      // brake if gap to leading intercepting vehicle is too small
      if (PredictedGap < MinPredictedFollowingGap)
      {
        MinPredictedFollowingGap = PredictedGap;
        float LeaderVelocity = Vehicle->GetVelocity(CurrentState.
           SimulationTime);
        float DesiredGap = GetDesiredGap(GetVelocity(), GetVelocity() -
           LeaderVelocity);
        LastDesiredGap = DesiredGap;
        float NewAcceleration = FMath::Max(GetAcceleration(DesiredGap,
           PredictedGap), GetAcceleration(DesiredGap, GetDistance(
           MutualNode, CurrentState.SimulationTime))); // accelerate as
           if other vehicle was leading vehicle
        if (NewAcceleration < AccelerationForInterceptingVehicle)
        {
          AccelerationForInterceptingVehicle = NewAcceleration;
          NewAccelerationSource = Vehicle;
        }
      }
    }
```

```
      else if ((VehicleOrder == EVehicleOrder::LeadingVehicle && ((
         bOtherIsOnIntersection && !bIsOnIntersection) || (!
         bIsOnIntersection && bGiveRightOfWay)))
        || (VehicleOrder == EVehicleOrder::OverlappingLeadingVehicle && (!
           bIsOnIntersection && bOtherIsOnIntersection)))
      {
        // brake if gap to following intercepting vehicle is smaller than
           a safe gap
        if (PredictedGap < MinPredictedLeadingGap)
        {
          MinPredictedLeadingGap = PredictedGap;
          float FollowerVelocity = Vehicle->GetVelocity(CurrentState.
             SimulationTime);
          if (PredictedGap < GetSafeGap(FollowerVelocity, GetVelocity()))
          {
            float Gap = FMath::Max(0.0f, GetDistance(GetCurrentLane()->
               GetEnd(), CurrentState.SimulationTime) - GetLength());
            float DesiredGap = GetDesiredGap(GetVelocity(), 0.5f *
               FollowerVelocity - GetVelocity()); // 0.5f to brake harder
            LastDesiredGap = DesiredGap;
            float NewAcceleration = GetAcceleration(DesiredGap, Gap); //
               brake at intersection
            if (NewAcceleration < AccelerationForInterceptingVehicle)
            {
              AccelerationForInterceptingVehicle = NewAcceleration;
            }
          }
        }
      }
    }
  }

  Acceleration = FMath::Min3(AccelerationForLeadingVehicle,
     AccelerationForInterceptingVehicle, GetFreeFlowAcceleration(
     GetVelocity()));
  Acceleration = FMath::Max(Acceleration, -GetMaxDeceleration());
  LastAcceleration = Acceleration;

  float DesiredVelocity = FMath::Clamp(GetVelocity() + Acceleration *
     DeltaTime, 0.0f, GetMaxVelocity());

  SetRenderStateDirty(true);
  CurrentState.Velocity = DesiredVelocity;
  CurrentState.SimulationTime += DeltaTime;
  MoveForward(DesiredVelocity * DeltaTime);
}
```

A.1.3 Fixed Tick

```
void UTrafficSimulation::FixedTick(float DeltaTime)
```

```
{
  if (bPaused)
  {
    return;
  }

  if (DeltaTime > TimeStepLimit)
  {
    DeltaTime = TimeStepLimit;
  }
  RemainingTime += SimulationSpeed * DeltaTime;
  while (RemainingTime > TickLength)
  {
    ExecuteAddVehicles();
    // update references of added vehicles
    ExecuteUpdateReferences();
    for (UVehicle* Vehicle : Vehicles)
    {
      if (Vehicle->IsInSimulation())
      {
        FVehicleState TempState = Vehicle->GetCurrentState();
        // reset temporary vehicle states that were added at each
        // lane change in the previously simulated step
        Vehicle->ResetLaneChangeStates();
        Vehicle->SimulateStep(TickLength);
        Vehicle->SetPreviousState(TempState);
      }
    }
    SimulationTime += TickLength;
    RemainingTime -= TickLength;
    ExecuteRemoveVehicles();
    // update references of removed vehicles
    ExecuteUpdateReferences();
  }
  // blending between last two simulated states by remaining time
  Alpha = RemainingTime / TickLength;
  for (UVehicle* v : Vehicles)
  {
    v->SetAlpha(Alpha);
  }
}
```

A.2 Acknowledgements

I would like to thank Christian Serra and everybody else at Limbic Entertainment GmbH for making this thesis possible, for letting me work with them and for helping me out with Unreal Engine 4 problems.

I would like to thank my siblings Annika Lauinger and Tobias Lauinger for being honest and very helpful proof readers.

I would also like to thank my girlfriend Marlene Wind for enduring my mood during stressful times.