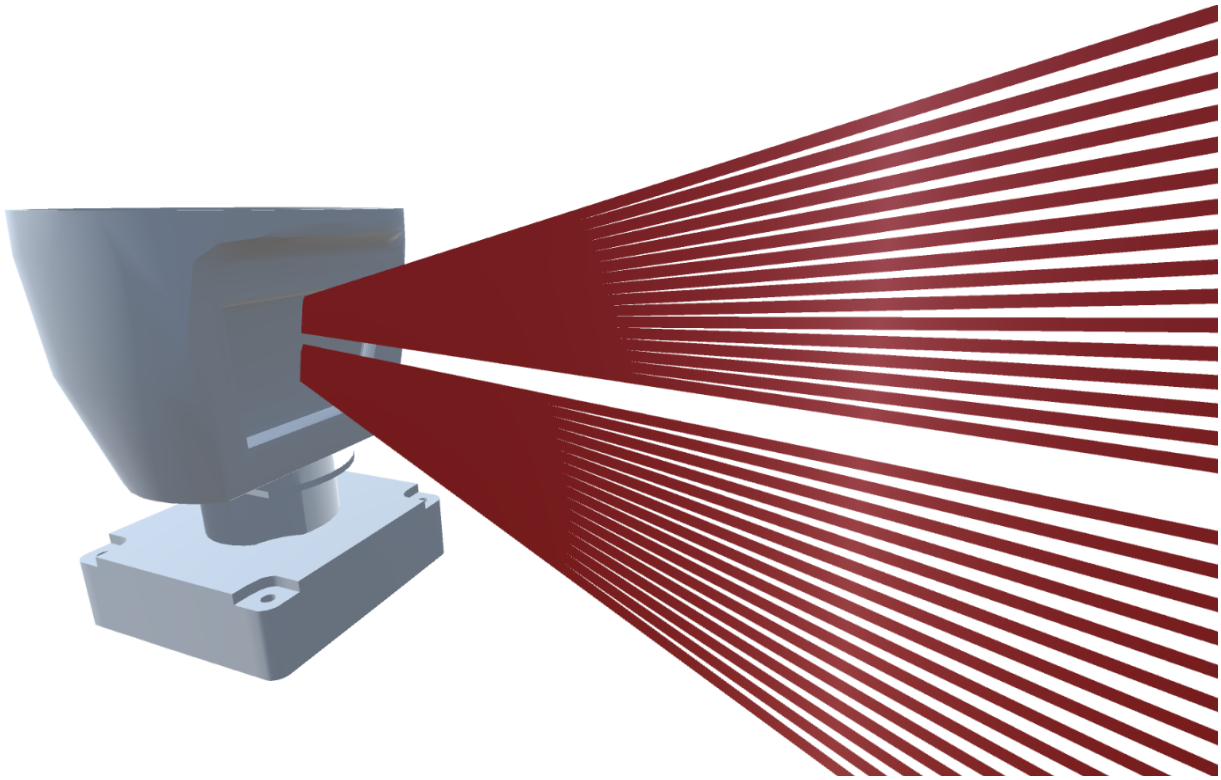
CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG



Virtual Generation of Lidar Data for Autonomous Vehicles

Simulation of a lidar sensor inside a virtual world

Bachelor thesis in Data and Information technology

Tobias Alldén, Martin Chemander, Sherry Davar, Jonathan Jansson, Rickard Laurenius, Philip Tibom

Department of Computer Science and Engineering
UNIVERSITY OF GOTHENBURG
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2017

Bachelor thesis 2017:10

Virtual Generation of Lidar Data for Autonomous Vehicles

Tobias Alldén, Martin Chemander, Sherry Davar
Jonathan Jansson, Rickard Laurenius, Philip Tibom

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
University of Gothenburg
Chalmers University of Technology
Gothenburg, Sweden 2017

Virtual Generation of Lidar Data for Autonomous Vehicles

© TOBIAS ALLDEN, 2017.
© MARTIN CHEMANDER, 2017.
© SHERRY DAVAR, 2017.
© JONATHAN JANSSON, 2017.
© RICKARD LAURENIUS, 2017.
© PHILIP TIBOM, 2017.

Supervisors: Vincenzo Gulisano, Dep. of Computer Science and Engineering.
Marco Fratarcangeli, Dep. of Applied Information Technology.
Examiner: Arne Linde, Dep. of Computer Science and Engineering.

Bachelor Thesis 2017:10
Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
SE-412 96 Gothenburg

Cover: a 3D model of a lidar sensor emitting lasers.

Department of Computer Science and Engineering
Gothenburg, Sweden 2017

Virtual Generation of Lidar Data for Autonomous Vehicles

Tobias Alldén
Martin Chemander
Sherry Davar
Jonathan Jansson
Rickard Laurenius
Philip Tibom

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

Abstract

The area of autonomous vehicles is a growing field of research which has gained popularity in the later years. Companies such as Tesla and Volvo continuously work on creating vehicles that can autonomously navigate traffic with the use of different sensors and algorithms. However, given the cost and risks of testing these in real world scenarios there may be a need for simulation tools in which the algorithms can be tested during development without the need for a real autonomous car. Thus opening the area of research for independent researchers and other actors with limited financial means. This thesis presents the creation of such simulation tools.

It is shown that sufficiently realistic simulation tools can be created using the game engine Unity along with free assets from the unity asset store. The simulation can be run on an off-the-shelf computer with good results. However, some aspects that can influence the resolution of the sensor in real life scenarios, such as weather conditions are not implemented.

Sammanfattning

Självkörande bilar är ett växande fält inom fordonsindustrin, något som blivit mer vanligt under de senare åren. Företag som Tesla och Volvo jobbar kontinuerligt på att skapa självkörande fordon som kan navigera i trafiken med hjälp av olika sensorer och algoritmer. Givet kostnaden och de risker som finns med att testa dessa i riktiga scenarier kan det finnas ett behov för en realistisk simulation där dessa kan testas kontinuerligt under utvecklingen. Detta öppnar även fältet för oberoende forskare och andra aktörer med begränsade finansiella medel. Denna tes presenterar skapandet av en simulation som löser detta problem.

Resultatet visar på att en verklig simulation kan skapas med hjälp av spelmotorn Unity tillsammans med kostnadsfria tillägg från Unitys asset store. Simulationen kan köras på en dator med modern hårdvara under hög upplösning. Däremot existerar det aspekter som kan påverka upplösningen av en verklig sensor som inte är inkluderade i simulationen, såsom väder.

Keywords: lidar,autonomous vehicles,simulation.

Acknowledgements

We would like to express our gratitude to our two supervisors Vincenzo Gulisano and Marco Fratarcangeli for their support in helping us in the creation of the simulation during the course of this thesis. We would also like to thank them for their help in answering administrative questions. Further, we would like to thank Fia Börjesson for helping us with structuring this report.

Contents

List of Figures

1

Introduction

The automotive industry is one of the larger and more influential industries in the world [1]. It has become an essential component in logistics, finance and a large portion of the common people's day-to-day life [2]. One of the future milestones in the evolution of vehicles is likely to be the autonomous vehicles, also referred to as self-driving vehicles.

There are many potential benefits with autonomous vehicles where some of the key benefits include road safety, decreased traffic congestion, lower fuel consumption, improved mobility for the elderly and the disabled, etc. Autonomous vehicles could theoretically improve the life quality of the entire planet. For example, roughly 90% of all vehicle accidents are caused by human error [3].

Some semi-autonomous vehicles are already being produced and are driven in traffic [4]. Semi-autonomous vehicles contains features such as auto parking, and lane-assisting; following the vehicle in front and holding the vehicle steady between the road lines. However, semi-autonomous features still requires the attention of the driver. Fully autonomous vehicles are currently being developed and tested [5]. These are supposed to navigate any kind of traffic situation without the intervention of a human driver.

One of the most widely used components in an autonomous vehicle today is the lidar (Light detection and ranging) sensor [6], [7]; sometimes referred to as a 3D scanner. It can also be explained as a light-based radar. It is used to measure distances to other vehicles and various entities in the surroundings. The lidar contains one or several lasers that will emit rays around the vehicle, thus scanning the environment. The result of the scan is a point cloud where each registered hit is a point in 3D space. Given a point cloud, various object recognition algorithms can be used to identify objects, thus, letting the vehicle navigate properly.

To create and test these algorithms, lidar data must first be collected. The traditional method to record lidar data is to use an actual lidar sensor. Using a lidar sensor in the real world is highly impractical, and cost inefficient. For example, many specific traffic situations are difficult to find and record. Therefore, there is a great need to generate lidar data virtually.

1.1 Purpose

The purpose of this thesis was to create a simulator, by implementing a software model of a lidar sensor within a virtual environment, to generate lidar data virtually. The simulator would model real-life traffic scenarios in such a fashion that object recognition algorithms could be tested.

The simulator aims to open up the area of lidar research to a wider range of researchers that may not be able to partake due to the financial requirements of using a real sensor, and to streamline the development of algorithms. Providing researchers with an efficient simulation tool for generating lidar data will help to test their work on a wider range of scenarios and thus not limiting them to existing data sets.

The completed simulator should contain the following components.

- A simulated lidar sensor.
- A simulated urban environment including both static and dynamic objects, such as pedestrians, poles, buildings, vehicles, etc.
- A user controlled, movable vehicle, on which the simulated lidar sensor will be attached. So that the user can move around in the virtual environment.
- The possibility for the user to adjust and set up different custom scenarios with an editor. As in the possibility of placing pedestrians and other objects in the virtual environment.
- The possibility to generate and export data collected by the lidar sensor to a file.
- Possibility to visually inspect the generated data with a visualization tool.

1.2 Problem Specification

There are many ways to construct a software to simulate a lidar sensor, this thesis approach to the problem is to determine whether the simulator could be created by using a game engine. This due to the fact that game engines comes with features that could speed up the development, such as a 3D environment with physics.

A number of game engines are available on the market, providing similar functionality. The game engine of choice within this project is Unity, as it meets all the requirements. These requirements includes a free license, an integrated 3D physics engine with ray-casting, and a fully featured development environment.

1.3 Scope

The main focus within the project is to make the lidar data be as accurate as possible. Therefore, the amount of work on areas that will not affect the the lidar sensor and its output will be kept minimal. For example textures and realistic vehicle physics, do not affect the lidar sensor and is therefore not prioritized.

The simulator should execute smoothly on consumer grade hardware, and if not possible in real-time, a slow-motion feature should be available.

A lidar sensor may be influenced by weather conditions. Studying this phenomenon may be interesting for algorithm testing purposes. However, implementing weather is outside the scope of this thesis, because of the physical complexity with light reflections and water [8, p. 57].

During the course of this thesis, a real lidar sensor is not provided, thus making some parameters difficult to analyze, for example reflection intensity and noise. For that reason, such parameters have been excluded from the simulator and only time stamps and precise coordinates are being exported.

2

Technical Background

This chapter introduces the technical backgrounds that several parts of the thesis depend upon.

2.1 Lidar Sensors

A lidar sensor is a tool for measuring distances to objects, and to examine the surface of various entities using infrared lasers [9]. Lidar is used in various applications, such as forestry, urban planning and autonomous vehicles.

Given that the speed of light is relatively constant in air, the distance to objects can be determined with high accuracy by the elapsed flight time for a ray of light. The sensor emits a pulse of light that hits an object and is reflected back to the source. From this exchange it is possible to calculate the distance between the sensor and the object. As such a coordinate in space can be determined where the object resides. When increasing the number of lasers that scan the area and the frequency with which they fire, an increasingly detailed view of the vicinity of the sensor can be determined.

2.2 Game Engine

A game engine is a framework for creating games and applications. Features that are provided by game engines include graphics rendering software, a physics engine and simple AI [10], [11]. These are presented in the following sections.

2.2.1 Physics Engine

A physics engine is one of the core features that is provided by popular commercial 3D game engines. It is the software layer that simulates the motion and physical interactions of objects. These behaviors are simulated by several different systems. One of these systems handles rigid body dynamics [12, p. 1], based on Newton's law of motion, whereas another system handles collision detection and collision response [13, p. 247].

Collision detection is the problem of determining when different objects are intersecting each other [14, p. 295]. When intersection occur, a simulation of the resulting behavior (collision response) will be calculated by the physics engine. Figure 2.1 illustrates a collision response where a sphere is bouncing between two walls in a pong game.

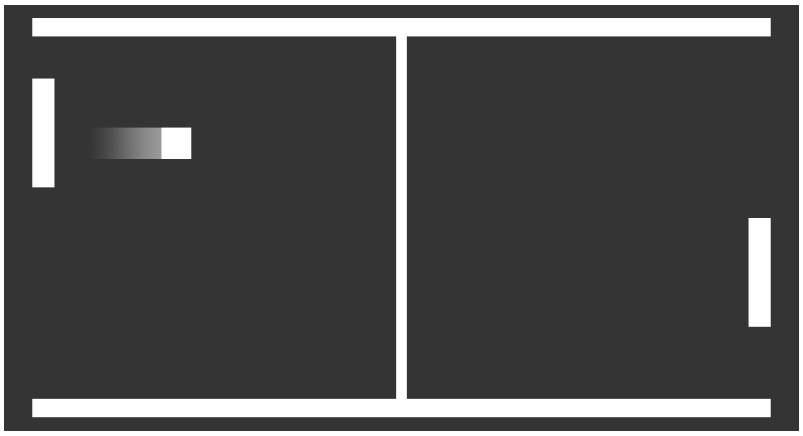


Figure 2.1: An illustration of how a box is changing direction after a collision has occured

2.2.2 Graphics Engine

Most of the visual representation in games and applications is made up by 3D graphics that is based on polygonal meshes [15]. Polygon meshes are collections of polygons that together constitutes a 3D surface. Polygons are 2D shapes made up of edges and points. The most common polygon is a triangle. Modern graphics engines creates 2D images from these meshes in a way that gives an illusion of a 3D model.

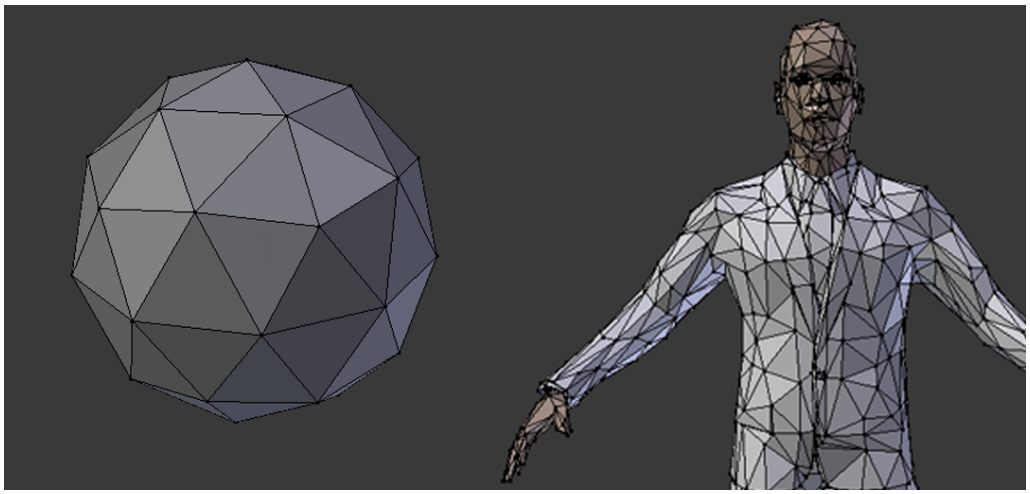


Figure 2.2: Left: A 3D mesh shaped like a sphere. Right: A 3D mesh shaped like a human

2.2.3 3D Animation

Simulating the motion of walking pedestrians as seen by a lidar sensor is a part of the project and when complex motion like that of humans is represented in a virtual world, an animation system is used. One way of making 3D animations is called skeletal animation. It involves two components. Firstly, a mesh of polygons in 3D space that represents the visible surface of an object. Secondly, a collection of so-called bones [16]. A visual representation of the animation bones can be seen in figure (2.3).

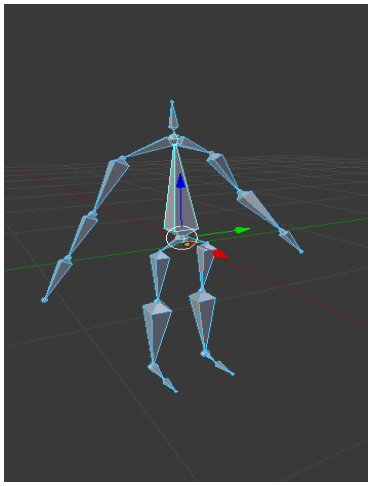


Figure 2.3: Animation bones without a 3D mesh.

Bones are usually represented by their position, rotation and scale in 3D space. When a bone moves, the bones that are connected to them will also move. For example, when animating a human, a change in the upper arm bone will affect the hand and finger bones as well.

The mesh that makes up the visible object is affected by changes in the bones such that moving the bones will result in a corresponding deformation of the mesh. Moving a bone will move some vertices in the mesh more than others, this because the connections between the bones and the vertices are weighted differently depending on the bone. An example would be that moving a leg bone will deform the 2D mesh close to the leg while the rest of the figure does not move.

Most 3D animations that have to do with a character being animated are made by using key-frames. They store the location, rotation and scale of the animation bones at certain times in the animation. When an animation is being played, the movement of the character will be interpolated frame by frame between the poses that the character holds in the key-frames [17, Ch. 6].

2.2.4 Pathfinding

To make the simulated pedestrians and cars move through the virtual world, a branch of artificial intelligence called pathfinding is used. Pathfinding is a common challenge in a wide range of computer applications and involves finding the shortest path between two points in space while taking all obstacles into account.

The most common solutions to pathfinding problems involves representing the area through which to find the path as a graph of connected nodes. The A* algorithm is commonly used for finding the shortest path to a destination through a graph. A* (pronounced A-star) is a more efficient version of the well-known Djikstra's algorithm, which finds the shortest path through a graph of interconnected nodes [18, pp. 633-643].

Another common way of representing a traversable area is by using navigation meshes, which are 3D polygonal meshes used for pathfinding. To find a path across many polygons, it is common to treat the polygons themselves as nodes in a graph and use an algorithm like A*. When moving inside just one polygon, finding a path is trivial because everything inside is walkable territory. Usually, when avoiding non-static obstacles inside a polygon, a separate algorithm is used for obstacle avoidance, like reciprocal velocity obstacles [19].

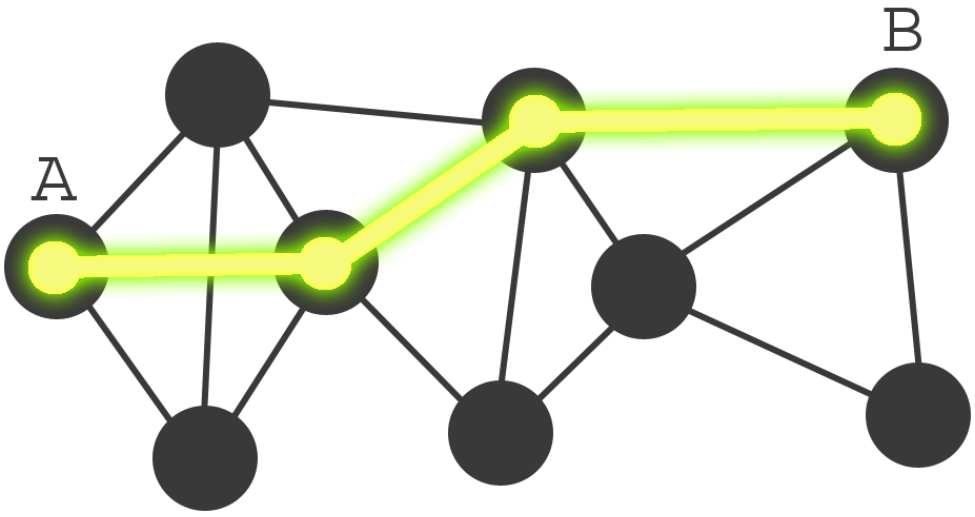


Figure 2.4: A path from node A to node B through a graph

3

Methods

This section describes the methodology that has been used to implement the different components of the product. Furthermore, it describes the requirements of the various parts that were created and the choices that were made in developing these.

3.1 Simulation of a Lidar Sensor

To realistically simulate a lidar sensor, a real lidar sensor is used as a model. The Velodyne HDL-64E sensor was chosen as it is commonly used all over the world. To simulate this type of lidar sensor, there are a few key components to consider. The amount of lasers, each individual laser position and angle, and the rotational speed.

Each laser is represented in the game engine by using a method called ray-casting. In short mathematical terms, ray-casting is a directional three-dimensional vector, which checks for intersections with other geometries, and then returns a coordinate of the intersected position. Thus, it is considered as a realistic representation of a laser.

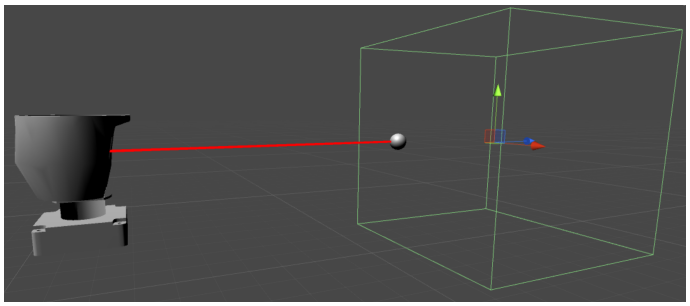


Figure 3.1: Virtual lidar with a single laser, colliding with a cube.

Unity handles ray-casts by executing them within the physics engine. Multiple ray-casts can be executed within a single physics frame, which gives the result of simultaneous actions. Although, the ray-casts are not executed in parallel, because Unity API calls must be executed within the main-thread [20].

3.2 Simulated World

A simulated environment, populated with objects with different geometries that represents real life objects, was assembled in order to produce a realistic environment. The objects that is produced is both static and dynamic for the same reason that both types are included in real lidar data.

Each object in the simulation needs its shape to be seen by the ray-casts of the sensor. This is handled by collision detection and is described in the following section along with dynamic objects and the car that carries around the lidar in order to produce lidar data.

3.2.1 Collision Detection

As described in section 2.2.1, collision detection determines when multiple objects intersect. The physics engine inside Unity that is used for this project, handles collision detection with components called colliders. Colliders define an object's shape for the purpose of managing collisions [21]. Each object in the simulator makes use of a collider in order for the ray-cast to be able to hit objects in the environment.

Colliders can either use mathematical representations of geometric shapes, or 3D meshes to define the shape of an object. The former are called primitive colliders, and often come in the shape of spheres, capsules and boxes as shown in figure 3.2. The latter is called a mesh collider and it works like a collection of a number of smaller colliders representing the triangles in the mesh. The three figures below (figure 3.3, 3.4 and 3.5) gives a visual representation of a mesh collider and the corresponding object it is attached to.

Figure 3.2: An illustration of primitive colliders.

Figure 3.3: A barrel object

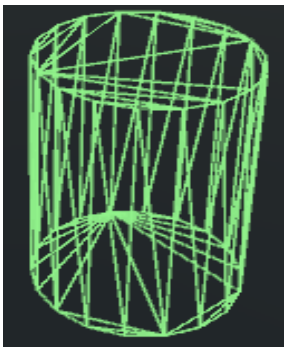


Figure 3.4: Mesh collider.

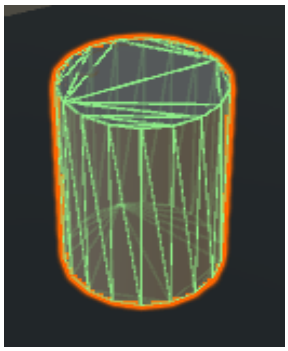


Figure 3.5: Mesh collider attached.

Colliders affect the performance cost of the ray-cast, where mesh colliders are the most expensive. The reason for that is that mesh colliders work as if they are made up of as many separate colliders as there are triangles in the mesh. Therefore, approximating the shapes of the objects with primitive colliders were it is appropriate, will increase performance [22].

3.2.2 Dynamic Objects

In the real world, many objects are moving dynamically, for instance cars and pedestrians. Therefore, simulating motion is an important part of producing simulated sensor data that is similar to real data. Unity has a path-finding system based on navigation meshes and the A* algorithm that can make objects find their way to a destination through an environment avoiding obstacles along the way.

With objects like cars, that do not change shape while moving, a navigation system would be enough, but for humans, who change pose constantly while walking, an animation system needed to be implemented to accurately describe their motion. It was also necessary to get the animation system to not just affect the visuals, but to also move actual colliders that can be detected by the simulated sensor. Unity's physics engine is able to simulate rigid body dynamics and wheel physics, which was useful when building a car and other objects that could interact with the world and other physical objects in a believable way.

3.2.3 User-Controlled Vehicle

A user controlled vehicle is desired as a feature in the simulator to be able to move the lidar sensor through the simulated world. This allows the user to generate lidar data for a moving vehicle in various traffic conditions, as such it is desirable for the vehicle to behave in a realistic way.

To achieve high realism without sacrificing performance, multiple options were tested and evaluated. These options include two techniques to control vehicles, by using the integrated physics engine and its force system [23], [24], and by constructing a controller which mathematically translate and rotate the vehicle. A physics related solution was expected to be more realistic while being more performance heavy. Meanwhile, the solution without physics was expected to have the opposite characteristics. Thus, the solutions was to be evaluated based on the trade-off between performance and realism.

3.3 Lidar Data Management

A lidar sensor generates large amounts of data, thus, an efficient way of storing generated data is desired. As such, a method finding a suitable data storage solution was conducted. Finding an efficient storage solution prevents the storage of data from draining the available computational performance of the simulation. Further, a tool allowing the user to manually inspect the collected data in a visual manner could enable the user to inspect whether the collected data is of sufficient quality for their purposes. As such, a visualization tool was created.

Finally, the ability to export the generated data for further use in other applications was deemed necessary. This would allow the user to use the data in various other applications, such as algorithm design.

3.3.1 Storing Point Cloud Data

A method for determining a data storage solution, commonly referred to as a data structure, was performed by evaluation of various data structures. This evaluation was based on the complexity of various data structures for the operations that was relevant for the data, measured by the ordo-notation [25, pp. 77-86]. Basing an evaluation of various data structures on the complexity is justified by reviewing the time increase for various complexities. This increase in time is shown in figure 3.6. Given that the created data needed to be inserted in the data structure in rapid succession, a data structure with an efficient insertion complexity was chosen.

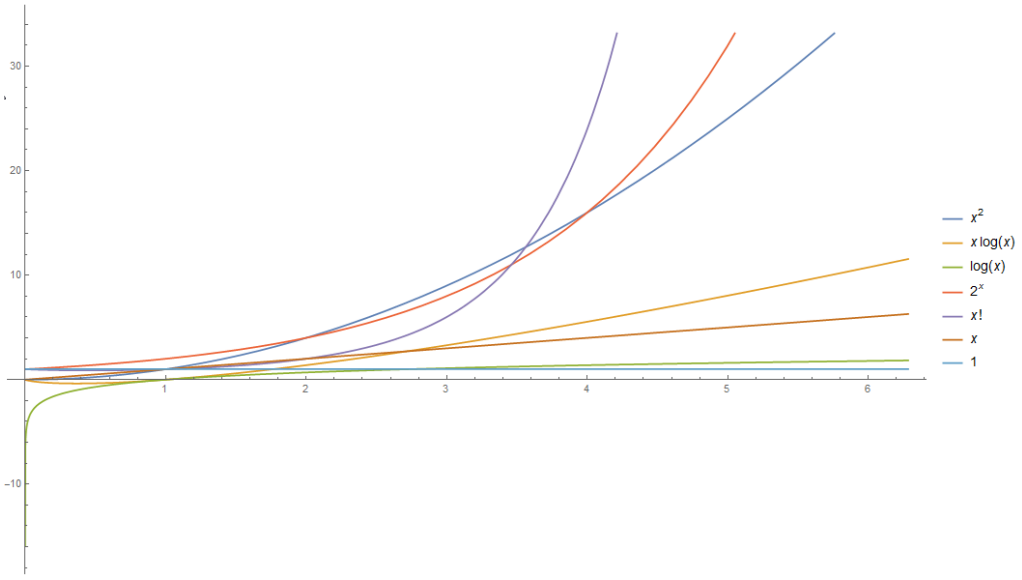


Figure 3.6: Time increase for common complexities, x axis is number of elements and y axis is the time

Moreover, the evaluation was also based on examining the appearance of the data that is generated from the sensor. The data that is generated are coordinates of the scanned area during a span of time. These are stored in a hash-table as it allows for mapping coordinates to the time in which they were created. This data structure has an efficient insertion complexity of $O(1)$ [26].

Further, the points in the hash-table are stored within linked lists, that also maintains a $O(1)$ complexity. Within the hash table the data is stored with the start time of the lap as the key, and the collected coordinates during that lap as the values. Due to the fact that this allows for independently reviewing the laps during a simulation it was evaluated as a suitable storage solution for the data.

3.3.2 Visualizing Point Cloud Data

The necessity of allowing the user to view the collected data in a visual manner was apparent as this would allow the user to manually inspect the collected data and determine whether it was of further use for their purposes. As the generated data contains a set of coordinates, visualizing this as a point cloud was chosen. This way of visualizing the data would allow for easy evaluation of the data as the cloud would contain small particles at the collected positions within the simulation. As such, the visualization would at higher resolution show the collected area in a precise matter. It was determined that two different visualizations were needed, one of which would allow the user to view the data within the simulation in real time as it is being collected. The other option would exist as a post-simulation visualization so that the user can load data that has been created in a post-simulation time-frame.

Visualizing a point cloud can be achieved in several ways, example visualizing the data as a surface via the creation of a mesh, or using a particle system with fixed particles in a space [27], [28]. These two approaches were evaluated, as a result the mesh creation technique is used in the post-simulation visualization and the particle approach is used in the real time visualization.

3.3.3 Export of Point Cloud Data

In applications such as games and simulation tools where data is created, it is desirable to save the data to the hard drive for later use. There are a number of ways to export data, some of which are more suitable for communicating via the Internet, and some are more suitable for applications which needs to access the raw data. For example, JSON and XML which are provided within the Unity API are suitable for communication, while for example the CSV format is more suitable for storage.

As the simulator was expected to generate millions of points, an efficient storage format was prioritized. The CSV format is more efficient than JSON, as it has a smaller overhead [29]. Especially when it comes to managing large amounts of data. An illustration of the difference can be seen in figure 3.7, where CSV is shown to be using less bytes. Thus, the CSV format was used in the implementation of the export system.

JSON

```
{"Point": {
    "Time": "1.2",
    "X": "182.9825",
    "Y": "92.8812",
    "Z": "1.1776"
}}
{"Point": {
    "Time": "1.2",
    "X": "182.9817",
    "Y": "92.9418",
    "Z": "1.1743"
}}
```

CSV

```
Time X Y Z
1.2 182.9825 92.8812 1.1776
1.2 182.9817 92.9418 1.1743
1.2 182.9878 92.0054 1.1719
1.2 182.9438 92.1234 1.7896
1.2 182.9515 92.6578 1.1463
1.2 182.9423 92.9265 1.1653
1.2 182.9412 92.8892 1.4496
1.2 182.9321 92.6878 1.1953
1.2 182.9345 92.8012 1.1376
1.2 182.9345 92.1934 1.1776
1.2 182.9345 92.4795 1.1353
```

Figure 3.7: The JSON part displays only two points, while the CSV part displays 11 points.

The basic grammar of CSV consists of separating different records of data by commas and separating the records with line breaks. The various fields can also be enclosed by commas or any other delimiters [30].

3.4 Efficient Design of User Interface

To allow users to navigate through the different components of the simulator and to control different parameters, an editor with a user interface is required. As there are many aspects which can and should be controllable the design was based on enhancing usability of the simulator for the user to be able to use it efficiently [31].

The usability of a user interface can be divided into two different aspects, ease of use and efficiency [32]. Donald A. Norman divided the problems with ease of use in systems into two concepts defined as 'Gulf of Evaluation' and 'Gulf of Execution', in his book *The Design of Everyday Things* [33, pp. 38-40]. These concepts divide errors in execution of tasks as lack of understanding how to execute a task and lack of understanding if a task was executed in a correct way as described in figure 3.8.

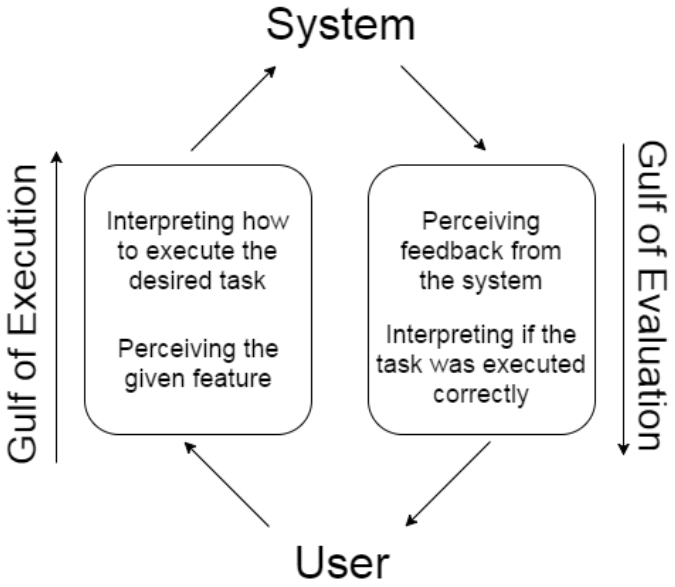


Figure 3.8: The concept of Gulf of Evaluation and Gulf of Execution

To avoid the problems and errors when designing interfaces one has to take the mechanisms of the human mind in consideration. There are many defined principles for how to handle this but most of them can be categorized as attention, perception or memory related [34, p. 407]. The concept of designing in consideration to attention and perception is to make it easier for the user to comprehend what information and components within the interface is currently relevant and how to utilize them [34, pp. 408-409]. This through adjusting the amount, placement and visualization of information. Meanwhile, the concept of designing in consideration to memory is to implement functions using the expected knowledge of the user, such as common experience to enhance understating [34, pp. 410-411]. An example of this is using a triangle as an icon for a play button.

3.4.1 Simplified Systematic Layout Planning

Beside the aspect of ease of use, usability also consists of the aspect of effectiveness. When optimizing a user interface based on effectiveness it is important to consider the relation between different components. There is a great benefit to place components used together or in series closer to each other as it is makes it easier to handle the combined information of the two. Moreover the amount of movement throughout the layout is minimized which improves effectiveness of the usage [34, p. 408].

To systematically improve the efficiency based on components relations, a method called SSLP (Simplified Systematic Layout Planning) is used. SSLP is a method invented by Richard Munther which evaluates the necessity of two components in a system being close to one another, and how to optimize the systems layout based on their relationship [35]. SSLP consists of three main parts; analyzing the systems layout, searching for better solutions and selecting the best solution for implementation. The first part is based on input data which requires an initial layout to analyze, therefore the layout of the initial user interface will be designed according to the developers' expectations of usage. Components expected to be used in series or together should as described be placed in groups or close to each other.

When a first version of an interface has been designed the first part of the SSLP is introduced by constructing a relationship diagram. A relationship diagram visualizes the serial usage of the components of an interface and highlights which components are most often used in series. The diagram is produced by analyzing and observing a system during usage and noting which components are used in series and how often. Additionally to ensure the quality of the analyze, multiple test users are used during the observation which are given a brief explanation of the purpose of the system and relevant technical information.

The relationship diagram is then used in the second part of SSLP to search for any design errors which should be dealt with. The most interesting part of the search is finding different layout solutions to place closely related components closer to each other. These solutions are then evaluated during the last part of SSLP to select the best solution found.

As an additional feature to the methodology, a post design selection validation will also be done. This through observing a new group of test users with the same prerequisites and then comparing the usage of features changed after the SSLP.

4

Results

This chapter aims to describe the functionality and features of the lidar simulation tool, as well as the underlying technical solutions. Furthermore the conducted methods and their corresponding results are presented within the sections.

4.1 Simulated Lidar Sensor

The most essential component in the simulated lidar sensor is the laser, which makes use of a method called ray-casting, see section 3.1. From now on, this component will be referred to as a "laser".

The simulated lidar is built from an indefinite number of lasers divided into two sets, where each set can be angled and spaced differently from one another, as well as the FOV (Field of View). The individual lasers are angled based on the FOV and the total number of lasers. This achieves an evenly distributed angle between each laser. By doing this, the angle between each laser becomes an approximation, and gives the ability to model different sensors. The two different sets of lasers, angles, and FOV is showcased in figure 4.1.

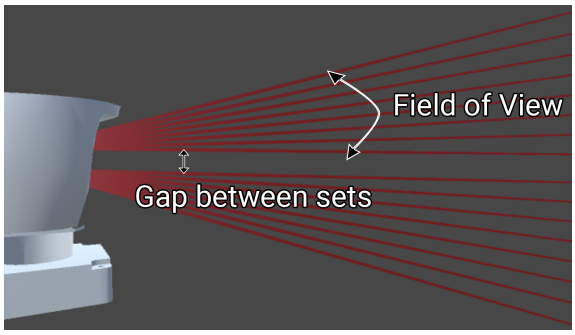


Figure 4.1: Virtual lidar, showing FOV and a seperation between two sets of lasers.

For the simulated lidar to work, it needs to use the given lasers, rotate with horizontal angular steps within a specific time frame, and record the hit position of each laser. See the visual representation in 4.2

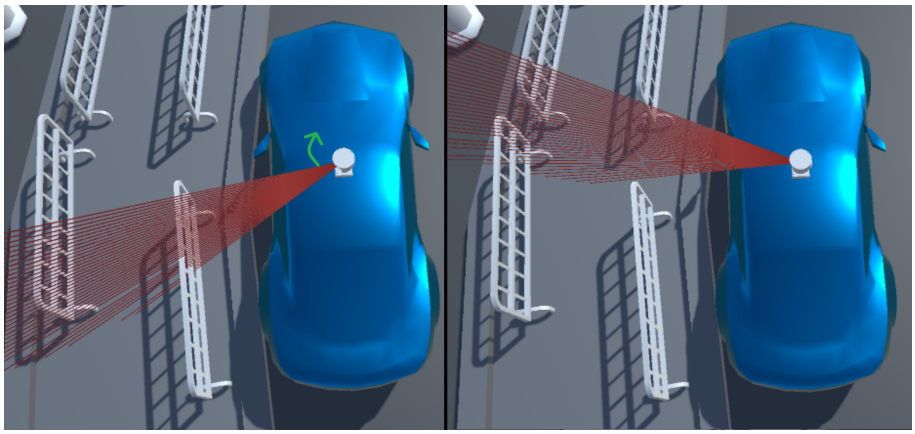


Figure 4.2: Virtual lidar, before rotational step to the left, and after rotational step to the right.

4.1.1 Ray-Casting in Real Time

There are two slightly different algorithms to implement a correct behavior of a lidar sensor, but both comes with different benefits and drawbacks.

The first algorithm is executing all of the given lasers, rotating one step, executing all lasers again, and repeat until an entire lap of 360 degrees is completed, all within one single physics frame. This works independently from the time steps in Unity Engine but it has a major drawback on the performance. While this lap is calculated once every given time point, it is all executed within one physics frame which means that everything in Unity Engine must wait for the lap to be completed before another frame can be calculated. Ray-casting is an expensive operation, and considering that one lap could contain over 2 million ray-casts within one single frame, the entire simulator would come to a complete stop.

The second algorithm is based on the same principle. But rather than completing one entire lap within one physics frame. It is executing a batch of lasers each rotational step. This algorithm is more beneficial to the user as it is possible to see the rotation happening step by step. Also since the calculations are spread out on different physics frames, the simulator will not be congested and it will appear to be smooth and functional. The drawback with this algorithm is that it is dependent on the amount of physics frames Unity can produce each second. By default Unity Engine runs at 50 physics steps per second[1], this value has been changed up to maximum value[2] of 5 000, to allow for real-time simulation. For example the Velodyne HDL-

[1]Default time step can be found within the Unity Engine.
[2]Unity's maximum value of 5 000 steps per second in Unity was found by testing.

64E can complete 38 000 steps in a second[3], while Unity Engine can only produce at most 5 000 steps in a second. This means that the simulator needs to process in slow-motion instead of real-time, to compensate for the difference.

The final solution that was implemented in the simulator is a combination of both above mentioned algorithms. The simulated lidar sensor automatically calculates number of steps that can be accomplished on the given computer, and the number of steps needed, and then pre-calculating additional rotations in each physics step. For example, if the number of steps needed is 38 000 and Unity can only complete 5 000 steps. It will perform $38000/5000 = 7.6$ rotational steps each physics frame. This forces the simulator to run at real-time speed if the hardware allows. However, it could still congest the simulator at very high resolutions, and if that is still the case the user may choose to run the simulation in slow-motion.

4.1.2 Configurable Settings

The simulated lidar sensor has been designed with flexibility in mind. So that it can be configured to behave in a precise manner. The main benefits of this feature is that it allows the user to customize and simulate almost any type of lidar sensor to date, not only the Velodyne HDL-64E. In case of running the simulator on a low-end computer, slow-motion might be required if the resolution of the simulated sensor is too high for the computer. Thus the user has the ability to lower the settings of the sensor and have it run at a smaller resolution but in real-time.

The following parameters are completely adjustable and can be set by the user via a menu:

- Number of lasers
- Laser range
- Rotation speed of the lidar sensor
- Rotation angle between scans
- Vertical offset between sets of lasers
- Vertical field of view for the top set
- Vertical field of view for the bottom set
- Angle from horizontal plane to normal of top set
- Angle from horizontal plane to normal of bottom set

[3]The value of 38 000 steps per second has been calculated based on the specifications in Velodyne's data sheets [36].

4.1.3 Visualizing Continuous Scans

Being able to see how the sensor operates is very useful to the user. Lines are therefore being rendered along the lasers. This allows the user to quickly see that the lidar sensor is operating, and if it is doing so correctly. It also allows the user to see exactly what the lidar is reading, without having to display a point-cloud. A benefit with rendering lines compared to a point-cloud is that it does not affect the simulation performance. This feature has also been very useful while developing the rest of the project, such as verifying and testing the point cloud. For example the user can see that the lasers and point-cloud is matched visually. It is also useful when configuring the settings of the lidar sensor, as a visual preview shows exactly how the lasers are going to execute.

4.1.4 Validation of Generated Lidar Data

The generated lidar data is meant to be used for testing and creating object recognition algorithms for autonomous vehicles. As such, the generated data must be realistic to become useful.

Comparison with real lidar data available from KITTI [37] has been done visually by modeling a similar scenario. As seen in figure 4.3 it is very hard to identify specific differences between the generated data and real lidar data.

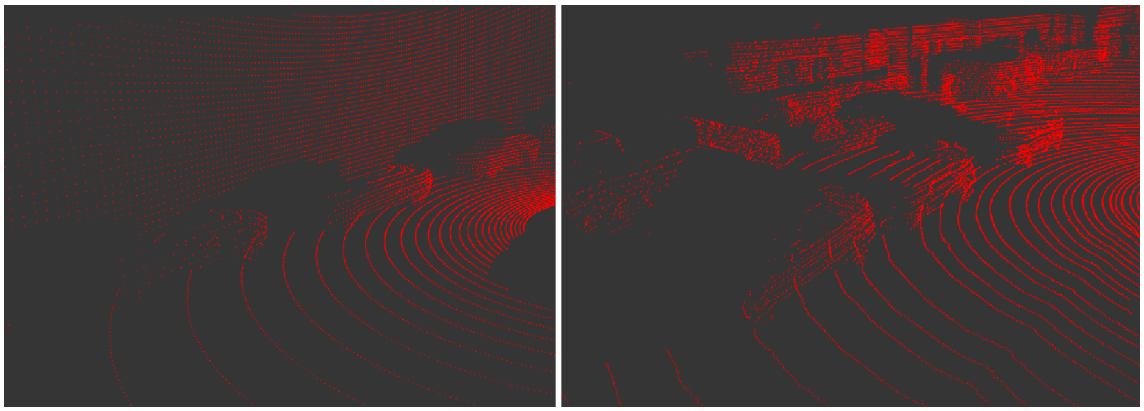


Figure 4.3: Left side is generated by using the simulator, and the right side is KITTI data.

Even though it is hard to differentiate the generated data in an image, validation has been done by comparing the patterns in a 3D environment. Further validation is restricted due to a real lidar sensor was being accessible during the course of this thesis.

4.2 Management of Point Cloud Data

As previously mentioned (section 3.3.1), there was a clear need to select a data structure for storing the points that the lidar sensor generates so that the storage of these would be efficient.

Moreover, in order for the user to be able to inspect whether the data that the lidar sensor creates is of use, a visualization of the generated point cloud was needed. This would support a high density of points without posing a major impact on the overall performance of the system. Following this, there was a need for two different visualizations, a real-time visualization that exists within the simulator, and a post-simulation visualization for expecting previously generated data.

Finally, there was a need for an option of exporting the data that the sensor generates for use in various purposes. These three parts are described in the following sections.

4.2.1 Storage Solution for Lidar Data

As described in section 3.3.1, the conducted evaluation of various data structures made the use of a hash-table as the primary storage solution suitable.

Due to the fact that the primary scripting language for the Unity engine is C# which does not have a concrete implementation of the hash-table, the dictionary is used. This is essentially the same data structure as the hash-table and as such has the insertion complexity of $O(1)$ [38]. Likewise, linked lists are used to hold the coordinates. This is the same implementation in C# as the general implementation of the data structure.

As the storing of points to the data structure does not impact the frame-rate of the application, it is evident that the implemented version of the data storage solution is efficient.

4.2.2 Post-Simulation Visualization

As discussed in section 3.3.2, a point cloud can be visualized by a few different approaches. In the post-simulation visualization, the surface mesh reconstruction is used. This is achieved by loading a previously collected set of points from a text file, where the data can either be from the simulation itself, or from a third party following the same data exportation conventions as the simulation (section 3.3.3). These points are translated from the string based representation of the text file into the data that is used in the program. When the data has been loaded and

translated successfully into the visualization, the points are translated into meshes (section 2.2.2). However, given that a mesh can only contain 65000 points, multiple meshes are created when the point count exceeds the limit [39].

When all the meshes have been created and loaded into the visualization, a point cloud representation of the data can be distinguished. To allow the user to inspect the area in depth, a movement script is attached to the main camera of the scene. This allows the user to "fly" around and inspect the data.

The external visualization shows the entirety of data collected during a simulation, and is as such a suitable representation of the quality of the collected data. This solution is also possible to be able to handle several million points effectively due to the fact that the system scales accordingly. The resulting visualization shows the data in an understandable fashion and is as such a very usable tool for inspecting the data in a post-simulation time frame. In figure 4.4 a part of the scanned environment are showed in the visualization.

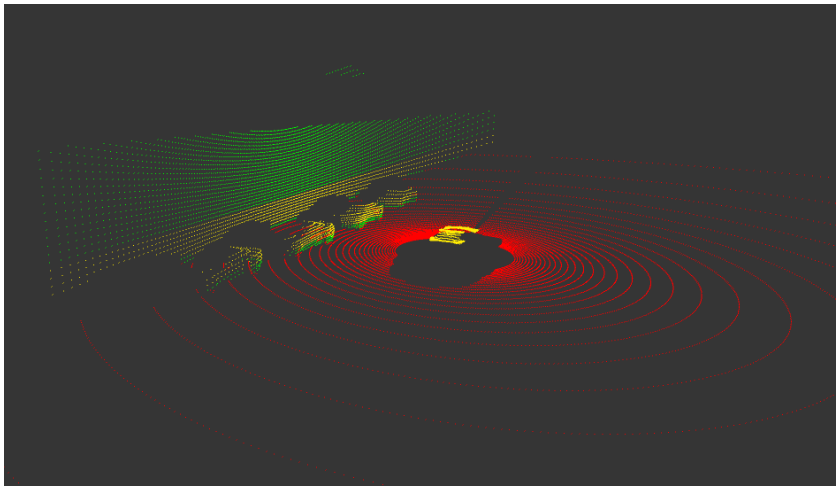


Figure 4.4: A section of the scanned environment containing a wall and several cars visualized in the post-simulation visualization

As the task of creating a mesh is rather time consuming for the program given that the data will have to be traversed and translated into vectors. Mesh creation is only a viable approach when the points needs to be loaded once. When the visualized points need to be updated continuously, a mesh representation is not an efficient solution, as such, the real time visualization needed to be created using another approach.

4.2.3 Real-time Visualization For Generated Data

As mentioned in the previous section, representing a point cloud that is updated continuously using a mesh is not a viable solution. As such, the real-time visualization

uses the built in particle system that Unity offers [40].

Although the particle system is mostly used to create explosions, weather conditions, fire and such, it can also be used to create a point cloud. Given the points will be fixed in space until they are redraw, some of the default properties of the particle systems are disabled as they are not needed, for example the emission of the particles.

Implementation of the real time visualization point cloud was achieved by the creation of several particle systems. Each one managing a fraction of the total amount of particles to be visualized. This solution became obvious when it was discovered that a single particle system could only effectively handle under 3000 points without major performance impact. These systems are reused when the old data is no longer relevant.

The particle systems are updated one at a time, this will create a rotation effect on the visualized point cloud. Due to this, the old points will still remain until they are replaced with new points.

Creation of multiple particle systems is a relatively lightweight operation when the number of created game-objects is limited. Each particle system can also only handle a finite amount of points without major impact on the efficiency. Thus the amount of needed particle systems P can be calculated with the following formula:

$$P = \left\lceil \frac{nL*360}{rotA*k} \right\rceil$$

Where nL is the number of lasers that the lidar is currently firing every step, and rotA is the angle of rotation for the sensor. k is the number of particles that each system can handle. It is shown by testing that $k = 500$ is a suitable value for an efficient simulation.

The resulting real time visualization has the ability of correctly showing the data as it is generated. However, even though the point cloud has been optimized the visualization still poses a small performance impact on the system, therefore it can be disabled if faster performance is desirable. The completed visualization is showin in figure 4.5.

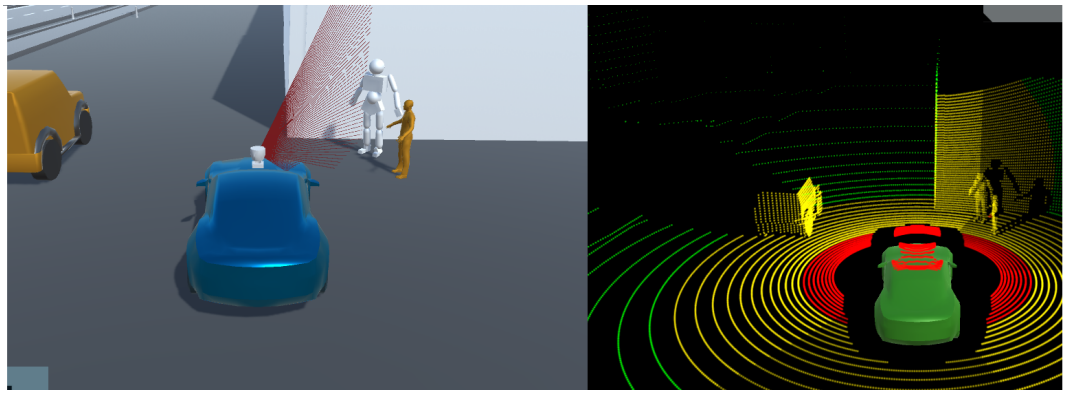


Figure 4.5: Realtime visualization of collected data showing a car and two pedestrians scanned and visualized.

4.2.4 Export of the Point Cloud Data

Saving data is desirable when the application creates data. An export tool is a part of the simulator and has the capability to save the data in CSV format. The sensor generates data continuously and the generated lidar data will be saved in a file by pressing the save button from the menu. When saving, the internal data is converted into CSV, where each row contains the time stamp, laser ID, xyz-coordinates, the spherical coordinate component where the car is the origin, and finally the id of the laser that fired the ray.

The limitation in the amount of records that the simulated lidar sensor can create is based on the hardware on which the simulation runs, specifically the memory, and the saved data file can be as large as the file system allows.

The recorded data is saved to a hard drive. To select location a file browser is used, to give the users flexibility. The file browser is a free asset from the Unity Asset Store [41]. To adapt this file browser for saving data, the select button is changed to a save button and the search bar string is used to type the name of the file to be saved. To make this file browser more user friendly, a confirm function is implemented which will ask the user to confirm or cancel saving the file.

4.3 The Resulting Simulated Environment

The following sections presents the implementation of the simulated environment in terms of implemented objects (both static and dynamic objects), collision detection, the behaviour of the car and the camera view that was used in the simulation.

4.3.1 Environmental Objects

As objects in the real world are both dynamic and static, both types were implemented in the simulated environment. Each implemented object represents a real life object such as a car, a building, pedestrians etc. Majority of the objects was imported, using already existing models from Unity's Asset Store [41]. The following list shows the objects that was imported.

- Phones.
- Traffic signs.
- Highway signs.
- Barrels.
- Railings.
- Hydrants.
- Road cones.
- Road blocks.
- Sewer caps.
- Power poles.
- Lamps.
- Benches.
- Traffic lights.
- Storage buildings.
- Vehicles.
- Roads.

Models that were not imported into the project were created from scratch, using external 3D modelling software. For example, the model of the lidar sensor was created in Blender [42]; and can be seen in figure 4.6. Another model that was created was a model of a static pedestrian, this model was created in a program called Makehuman [43]. This model can be seen in figure 4.7.

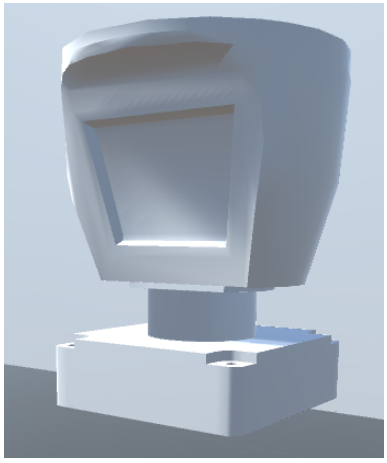


Figure 4.6: The final model of the lidar sensor.

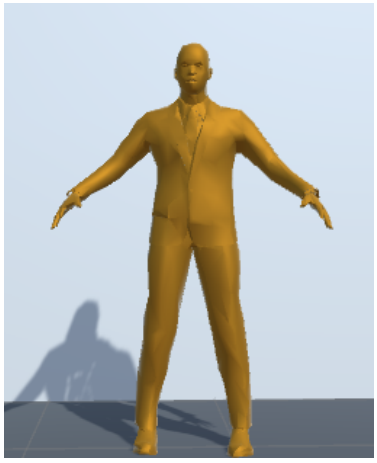


Figure 4.7: The created pedestrian model.

4.3.2 Collision Detection Optimization

When the shape of an object could be accurately described by one or more primitive geometric shapes, a compound collider made up by several primitive colliders were

used to represent the shape of the objects, while objects with more complex shapes used mesh colliders.

Compound colliders that is made up by several primitive shapes, were used when few geometric shapes could approximate an object well-enough. One of the objects that optimization's was applied to is shown in figure 4.8 and figure 4.9. Optimization's was also done to all the objects listed in section 4.3.1. The only object that used a mesh collider component was the barrel model.

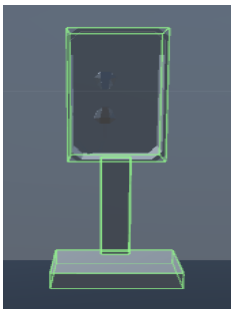


Figure 4.8: A phone model with 3 primitive colliders attached as a compound collider

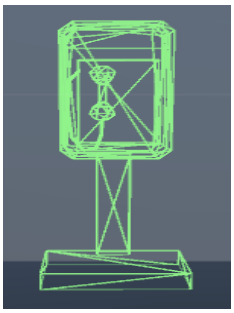


Figure 4.9: The mesh collider of a phone model, represented by a polygon mesh

4.3.3 Validation of Collision Detection Performance Difference

It was stated in section 3.2.1 that a change of colliders from mesh collider to primitive colliders could contribute to a performance increase. In order to validate this claim, two tests were made involving the model of a phone (see section above). On the first try, the environment was populated with 500 objects with mesh colliders, where each mesh contains 128 triangles each (see figure 4.9). On a second try, the environment was populated with 500 objects of the same type, each featuring a single compound collider made up by three primitive colliders of the box shape (see figure 4.8).

The resulting difference in performance was noticeable. Thus, the objects that were to be added to the environment would have compound colliders if their shapes could be reasonably approximated. The difference in performance is illustrated in figure 4.10.

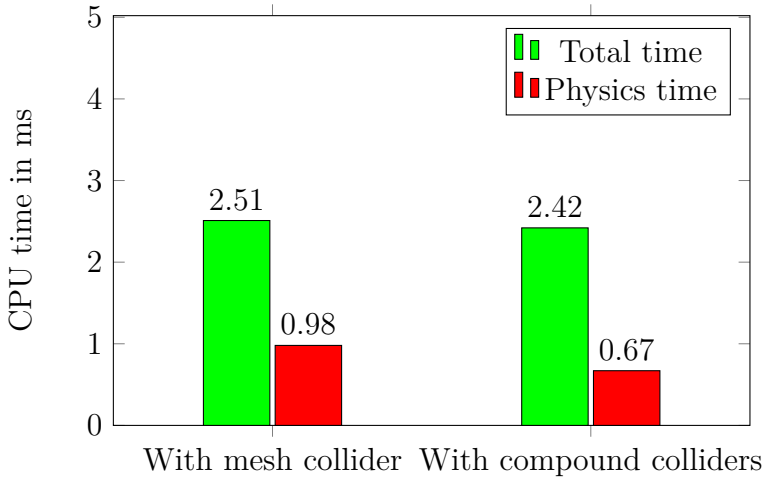


Figure 4.10: Consumed CPU time per physics frame

4.3.4 User-Controlled Vehicle

Initially a user-controlled vehicle was constructed using the integrated physics engine in the game engine to achieve a realistic behavior. This implementation worked through applying forces onto different parts of the vehicle in different directions representing the directional forces of an engine and the steering of a car. However due to the modifications of the physics engines steps, the calculations required for this implementation was proven to have a huge effect on the performance of the simulator. The reason for this is that the calculations had to be done much more frequently and therefore effecting performance more than expected.

To solve this performance degradation a new vehicle was constructed without using the physics engine. This was implemented in a way to mimic a physically realistic behavior by including parameters such as acceleration and a rotational pivot point in the middle of the rear axis as shown in figure 4.11.

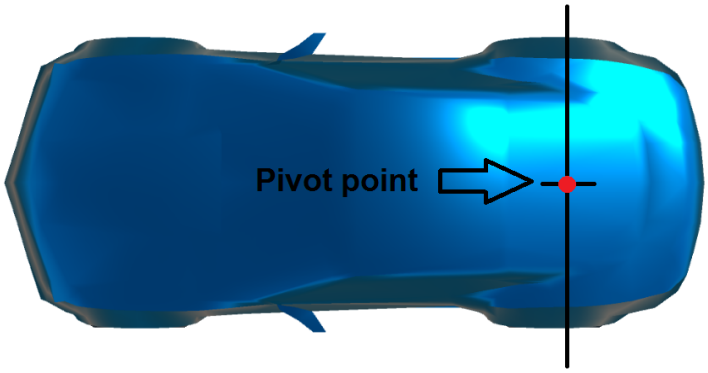


Figure 4.11: The pivot point of the implemented vehicle

To validate that this implementation did indeed solve the problem with the performance degradation a performance test was produced. This through testing the different kinds of control systems in an isolated environment in the simulator. The tests were done on the same computer with the specifications shown in table 4.1. The result, as presented in figure 4.12, validates the improvements through avoiding using the physics engine.

Table 4.1: Specifications of computer used for testing performance

	Maker	Model	Frequency
CPU	Intel	i5-4460	3.2GHz
GPU	Nvidia	GeForce GTX660 Ti	-
RAM	Kingston	8GB DDR3	1333mhz

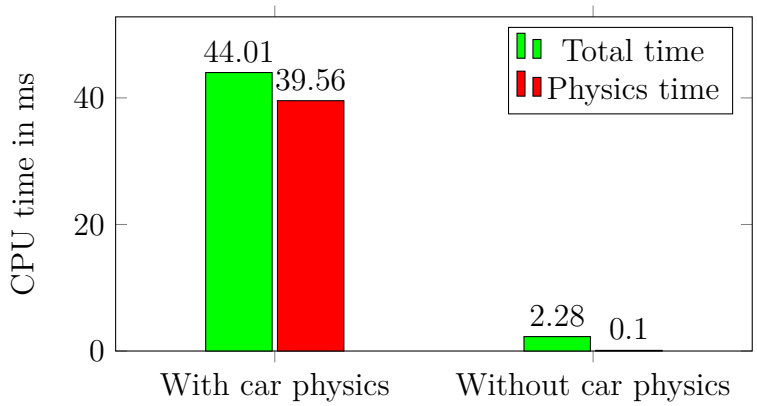


Figure 4.12: Consumed CPU time per physics frame

4.3.5 Camera Behavior

A 3D world needs to be viewed from somewhere, and most game engines use the concept of cameras, that can move around and show the 3D graphics from any angle. The simulation software is based on being able to transition between being in editor mode and simulation mode. Therefore, the camera behaves differently depending on what mode the program is in.

When the simulation mode is active, the camera is set to follow the user-controlled car. The user is able to freely move the camera around the car while the camera always looks in the direction of the car.

The movement of the camera has been smoothed out, to eliminate shakiness and prevent the camera from teleporting. If an object gets between the camera and the car, the camera will smoothly sweep forward to a position that is not obstructed by the object that got in the way.

When the editor mode is active, the camera switches to a bird's eye view of the 3D world. The user is able to scroll to the side by moving the mouse to the edge of the screen, or by pressing the arrow keys. It is also possible to zoom in and out by scrolling with the mouse wheel. When close to the ground, the zoom is slower while it gets faster the more zoomed out the user is.

4.3.6 Dynamic Objects

Aside from the static environment and objects, the simulation software features dynamic objects that can move along user-defined paths in the environment.

To achieve this, unity's navigation system has been used. A navigation mesh has been placed throughout the virtual world to define the areas that dynamic objects can move. Dynamic objects will move from point to point in a user-defined path and are detectable by the lidar sensor.

Paths are created and edited with the help of objects called waypoints. See figure 4.13 Each waypoint stores a reference to the next waypoint in the path and to the previous one. Linking them together in this way allows for constructing paths that moving objects can be ordered to follow. In edit mode, all waypoints are visible and able to be picked up and moved by the user. They also have a feature that draws a line between the waypoint and the next one it is pointing to, making it easy in edit mode to see how paths are connected. While in simulation mode, all waypoints are turned invisible.

Two types of dynamic objects that make use of the waypoints have been implemented in the simulator, cars and pedestrians. The moving cars function in the same way

that has been described above, by using the navigation system to move an otherwise static object around. The pedestrians however use unity's animation system to approximate the movements a person makes while walking, so as to not move along like statues on wheels.

The moving pedestrians use the fact that unity provides access to the animation bones of a model. This allows for physical objects with collision detection to be attached to the bones (see section 2.2.3). While animating, the bones will move the physical objects around, making it possible for the simulated sensor to accurately see the walking motion of the moving pedestrian. This adds another layer of realism when producing data from traffic situations involving pedestrians.

Figure 4.13: A waypoint object

The moving pedestrians are completely frozen while in editor mode, but start moving once the program switches to simulation mode.

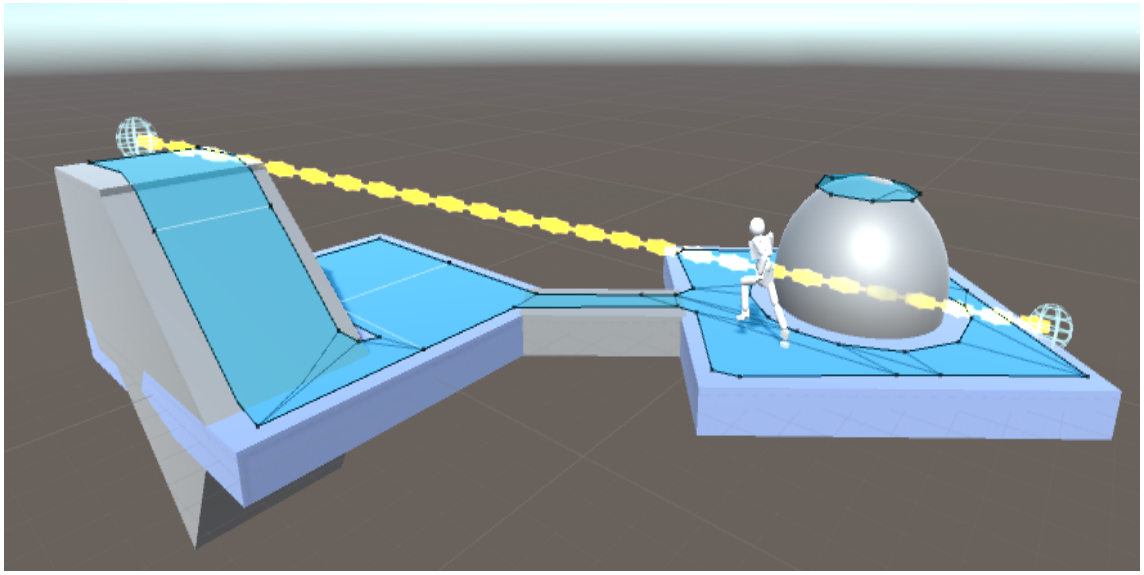


Figure 4.14: An animated pedestrian finding his way through a navigation mesh

4.4 World Editor

To allow users to easily edit the simulated world, the editor is implemented by allowing the user to drag objects from a menu straight into the world.

4.4.1 Editing the World

Every static or dynamic object in the simulator can be edited with the world editor function by clicking on them as shown in figure 4.15 and can then be positioned and rotated by clicking and dragging.

A transparency shader [44]; is used to make the editable object transparent while being moved by the user, to provide visual feedback to the user.

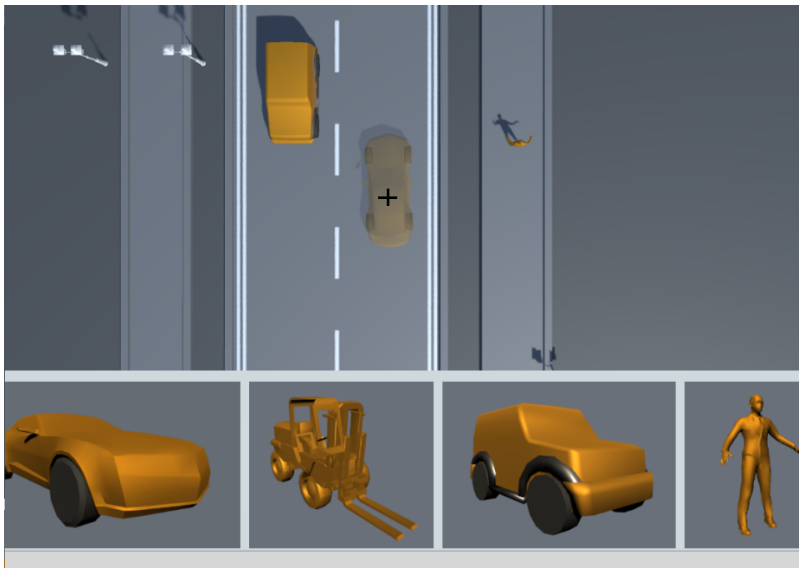


Figure 4.15: An example use case of the world editor. The black cross represents the mouse cursor. And the transparent vehicle is the object being moved.

The drag and drop system which is presented above utilizes ray-casting to determine where the user is trying to point at in the 3D world. This is because a cursor is located within the 2D-area of a monitor, but the cursor needs to get a coordinate from a 3D space. So, the ray-cast is directed from the camera, towards the mouse position within the camera space which will point into 3D space. When the ray-cast hits an object in the world, the 3D coordinate in the world of which the user tried to point, is returned. Thus, allowing the possibility to add objects to its location. Illustration of the concept is shown in figure 4.16 below.

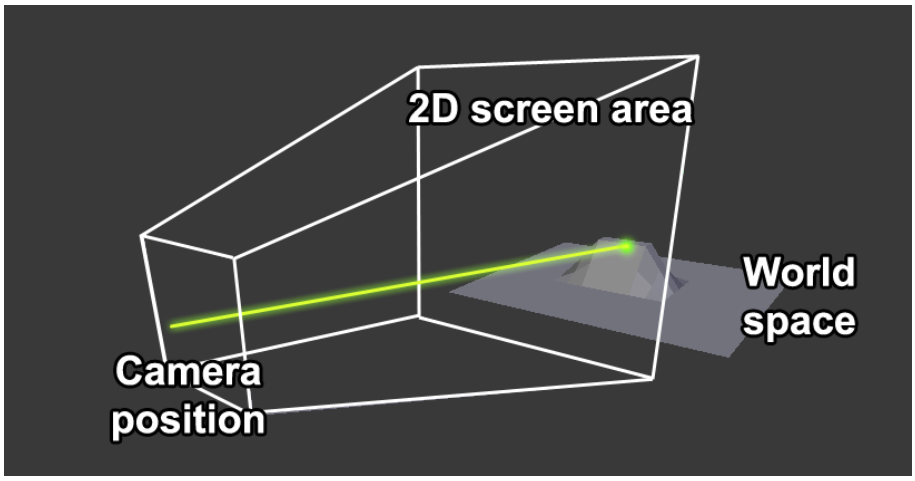


Figure 4.16: Illustration of a ray-cast from the camera to the cursor position, into 3D space.

Additionally, to the editable objects, if the chosen object was a dynamic object, the option to place waypoints will appear.

4.4.2 Placing Dynamic Objects

When a dynamic object is placed into the scene, the editor recognizes this and goes into path-placing mode. The user will get to place a waypoint object into the scene, and after a waypoint has been placed, the user gets to place another one. This goes on until either the user right-clicks or the user places the waypoint on another waypoint in the path.

If the user right-clicks, the path ends at the waypoint last placed in the scene. However if the user places the waypoint on another waypoint, the path closes at that spot and becomes circular. In either case, the path is finished and the editor exits path-placing mode. It is possible to edit paths in editor mode after they have been placed. The user needs simply to click at any waypoint in the scene to pick it up and be able to move it.

It is possible to redo paths that have already been placed. If the user left-clicks a dynamic object that have already been placed in the scene, all the waypoints that constitute its path are destroyed and it gets picked up and becomes movable by the editor once again. After the dynamic object has been placed in its new location, the editor will once again start the path placing mode to create a new path.

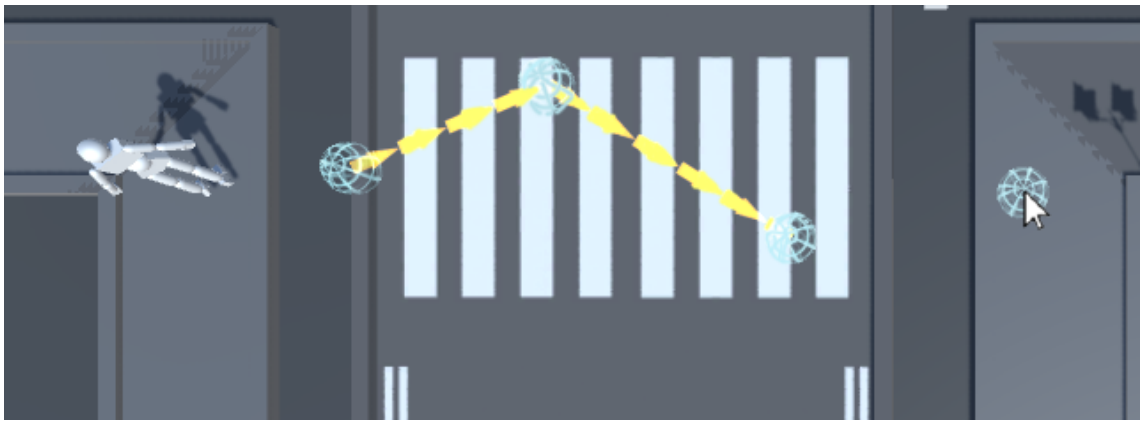


Figure 4.17: User placing waypoints in a path after having placed a moving pedestrian into the scene

4.4.3 Scenario Flexibility

The world editor combined with the implemented environmental and dynamic objects provides a high level of control of the scenarios for the user. Mainly because every static component but the ground within the world is possible to both move and rotate. Furthermore, as the possibility of defining both dynamic cars and pedestrians movements throughout the scenarios gives the user the tools to generate more realistic data.

A known issue for developers of autonomous vehicles is distinguishing humans which are partly hidden for example by a pole [45]. The scenarios shown in figure 4.18 gives two examples of similar scenarios set up with the implemented world editor.

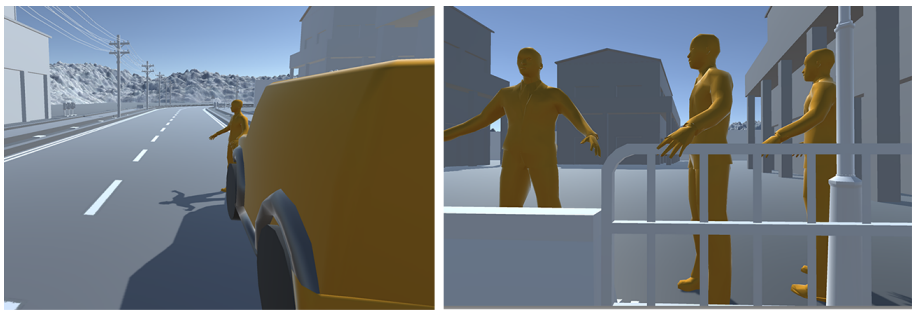


Figure 4.18: Two constructed scenarios of humans being hidden behind common objects

Beside those issues there are several more scenarios which developers of object detection algorithms struggle with. Another example among these are detecting and tracking high amounts of moving object [46, pp. 1707-1725]. An example of this is shown in figure 4.19 where multiple dynamic objects of different types are set up to mimic such a scenario. Furthermore the world editor allows users to easily adjust

the amount of objects in the scenario and generate multiple similar sets of data with different amounts of objects.

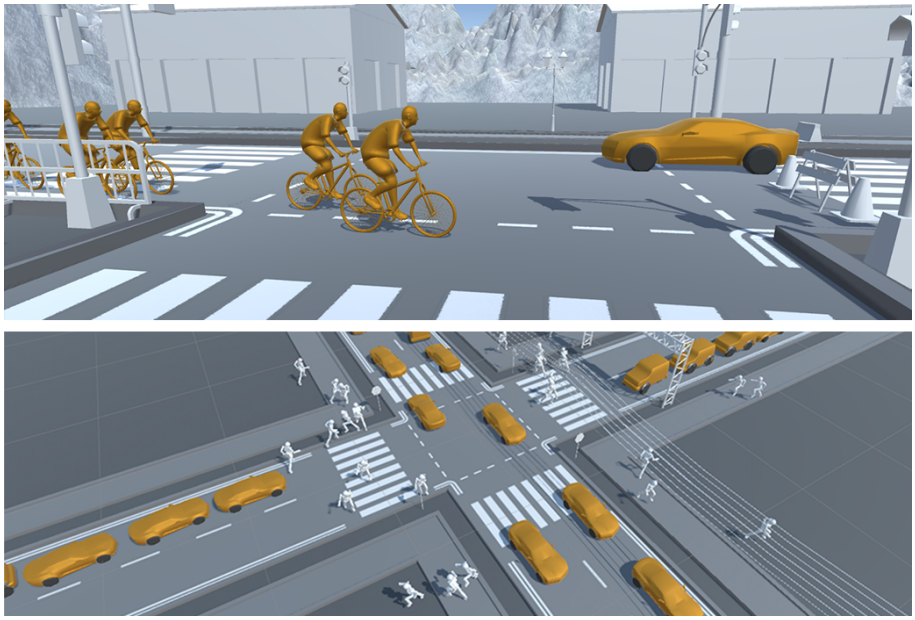


Figure 4.19: Two constructed scenarios including high amounts of objects

4.5 User Interface

To be able to navigate throughout the different components of the simulator and to make it possible to adjust scenarios and settings, a scenario editor was constructed. As the specifications of a lidar sensor and the functionality of the simulator can be quite complex to understand, the editor was designed in consideration to the constraints of the human mind. This through avoiding design errors which disadvantages humans in the way explained in section 3.4.

The choice of components in the user interface was based on enabling the user to adjust necessary parameters of the simulator. Moreover, as the editor was to be designed in consideration to attention, the fact that unnecessary information and functionality can make a user interface harder to perceive was taken in consideration [34, p. 408]. This resulted in implementing the following components while excluding any other additions:

- Buttons for starting, stopping and pausing the simulation
- Panel for controlling the settings of the lidar sensor
- Panel for adjusting the world in the simulator
- Component for adjusting the speed of the simulation
- Panel which shows relevant statistics of the simulation
- Button which toggles a visualization tool for a point cloud

- Button for saving collected points
- Button for going to the main menu of the program

As some of these components take up a lot of space and are not dependent of each other they were compressed through implementing a navigation menu to toggle between the lidar sensor settings panel and the panel for adjusting the world. Furthermore, the play and stop button were combined into a toggle. In addition to this compression some components were implemented to become invisible when not needed to further make the interface easier to comprehend in different states of the simulator.

4.5.1 Analysis of Initial User Interface

The initially constructed user interface for the editor which is shown in figure 4.20 displays the core components implemented. In the panel on the bottom left the play button is shown which is used to control the state of the simulator, toggling between simulation mode and editing mode. Additionally when entering simulation mode the bottom right panels are hidden from view as they do not have any use in this state of the simulation. To the top left the component which controls the speed of the simulation is shown which was implemented with a slider which controls the speed of the simulation.

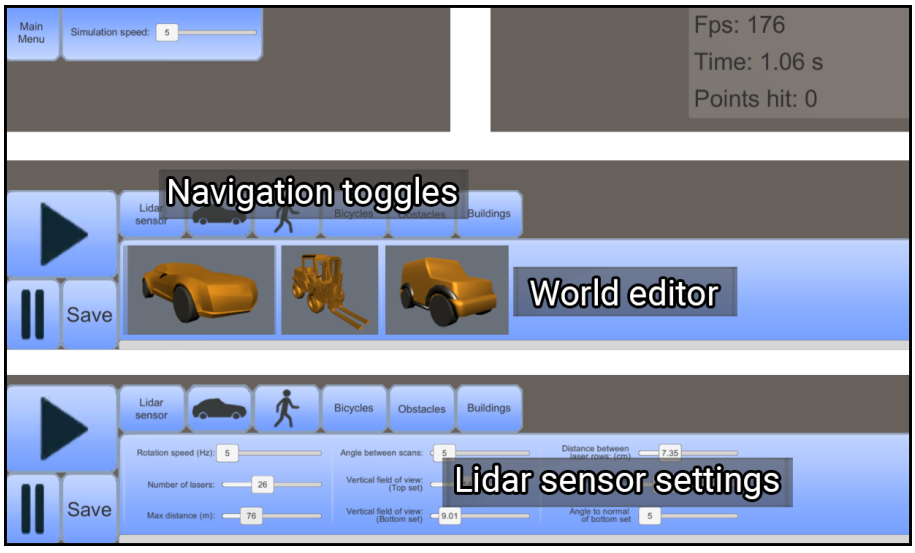


Figure 4.20: The top left part, top right part, world editor menu and the lidar settings menu of the initial user interface

The user interface was analyzed through observing usage by four people from outside the project group. The test users were given a brief explanation of what a lidar sensor is and the purpose of the simulator to have a fair understanding of how to utilize the different functions. They were observed during a ten minute period during which

they were instructed to try all of the functions in the user interface.

The relationship diagram shown in figure 4.21 was constructed based on the observations in order to analyze serial execution and task execution. The purpose of analyzing serial execution was to determine if any two components which are commonly used in series are placed in an unnecessary distance from one another. Meanwhile, the analysis of task execution determined the total number of singular executions needed to complete a task as well as the total distance traveled over the interface throughout the whole task. The relation between two components within the diagram is displayed by arrows connecting them with larger arrows representing a more frequent serial usage.

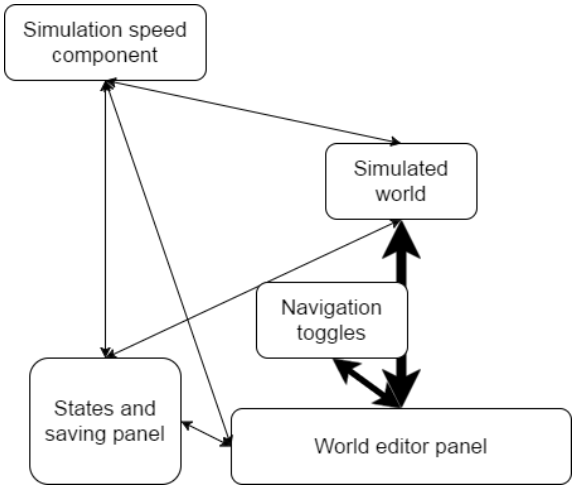


Figure 4.21: Relationship diagram over the user interface

The constructed relationship diagram determines that a major part of the usage was based on dragging new objects into the simulator from the world editor panel, resulting in a lot of unnecessary movement to and from the bottom of the interface. Furthermore, as there were many different menus for different types of objects, many serial executions included toggling between them.

4.5.2 Final Design of User Interface

To enhance the performance based on the relationship diagram shown in figure 4.21 the layout of the interface was redesigned to move the world editor panel closer to the world and removing obstacles between the two. Furthermore the different menus for objects were combined into more generic categories as dynamic objects such as pedestrians or cars and static objects such as buildings or signs. These adjustments resulted in the layout shown in figure 4.22.

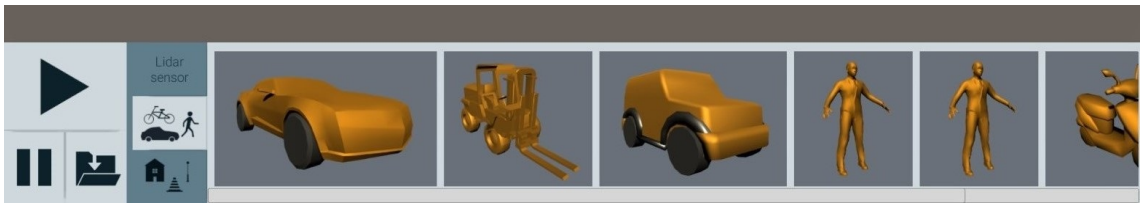


Figure 4.22: The menu for draging objects into the simulator

Moreover, the component for adjusting the speed of the simulation was moved closer to the middle of the interface as it was frequently used during simulation, resulting in the layout shown in figure 4.23.

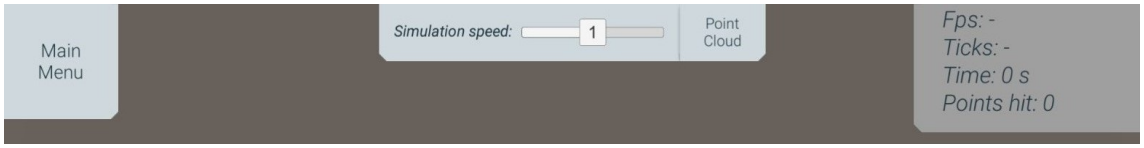


Figure 4.23: The top part of the user interface

A conclusion less highlighted in the relationship diagram was that the settings for the lidar sensors were hard to interpret. Therefore these settings needed a more easily explicable design.

This problematic aspect was solved through implementing a mimic of the lidar sensors lasers in which the lasers positions and angles change according to adjustments in the settings, which is shown in figure 4.24. This allows for direct feedback to the user to know which parameter of the sensor each setting corresponds to.

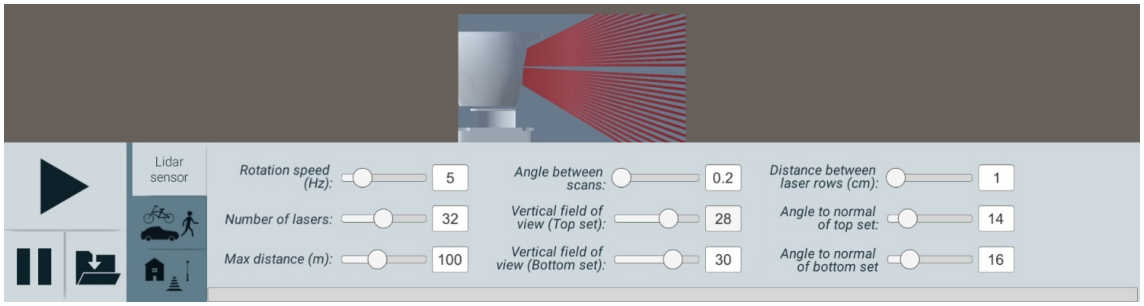


Figure 4.24: The menu for controlling settings of the lidar sensor

In addition to the mimic for the lidar settings an extra text box was included beside every settings slider to allow the user to input exact numbers through a keyboard as this could become easier when inputting exact numbers.

The redesigned version of the interface has a more efficient design as all components used in series repetitively are placed adjacent to each other, while lowering the amount of tasks in many serial executions.

4.5.3 Usability Validation

To validate the effectiveness of the redesigned version of the interface a second group of test users were used. To ensure reliable results none of the users from the first analysis were used. The new group were given the same information and instructions about the theoretical background and usage.

The results from the second analysis concluded that the mimic for the lidar settings solved the problem with interpretation of parameters. Furthermore the possibility to input exact numbers by using a keyboard allowed for quicker set up of the parameters in combination with the easier interpretation.

In parallel to the lidar settings, improvements was also shown in the world editor. First of all the amount of tasks performed while editing the world was lowered substantially as the user no longer had to toggle between many categories. Additionally the users adapted to the controls of the world editor much faster as the panel from which to drag objects was now placed adjacent to the actual world where they were to drop them.

5

Discussion

In this section the results of the various parts of the project is discussed. This includes the solution, challenges and the possibilities to further develop the created simulation.

5.1 Simulated Lidar Sensor and Data Generation

This section discusses whether the simulation of a lidar sensor within a game engine is able to generate realistic lidar data, by validating the accuracy of the final lidar data, and analyzing the performance.

5.1.1 Accuracy of the Lidar Data

Validating the accuracy of the generated lidar data is a very difficult task as we did not have any access to a real lidar sensor to compare with. However, there are data sets that can be downloaded from KITTI [37]; which has been recorded by using a real Velodyne HDL64-E sensor, and these data sets have been used as a reference.

The virtual lidar sensor is built using methods that make it theoretically resemble a real sensor (Velodyne HDL64-E) according to its specifications. The angle in between each laser has been approximated based on the field of view. So, it is not exact but should be close to reality.

Further, the generated data has been verified against the data sets from KITTI. The verification is done by displaying both data sets in a visual 3D point cloud. The generated points closely matches the patterns of the data sets from KITTI.

With the knowledge that the virtual sensor follows the specifications of a real sensor, and that the generated data closely matches the KITTI data sets. It can be concluded that the generated data is not exact, but close enough to be used for its purpose, which is to create and test object recognition algorithms.

5.1.2 Simulation Performance

Ray-casting is generally considered to be an expensive operation in regards to CPU time, and in this case it proves to be true. As the more ray-casts that are added, the slower the simulation will run. When utilizing this feature in a game engine, such as Unity, which executes all of the ray-casts within the main thread, certain limitations apply. These limitations affects the amount of work that can be done by the physics engine, such as calculating moving objects. This was circumvented by avoiding use of the physics engine (except from the ray-casting), and by doing more work within each physics frame, such as automatically calculating lidar scans ahead of time.

Normally in a game, ray-casts are used with for example, vehicles to detect the distance to the ground, or obstacles in its path. These ray-casts are usually few (0-100) and executed within a span of several seconds. While in this case, the simulator needs to be able to execute over 2 million ray-casts per second. Which is way beyond of what Unity is optimized and intended for.

Finally, it was possible to achieve an acceptable performance within a game engine. However, creating an optimized engine for the specific task of doing a lot of ray-casts, is likely a more effective alternative in terms of performance, but would require a lot more time and work to develop.

5.2 World Realism and Performance

Because physics simulation had to be cut out from the simulator, every collideable object is static, aside from those using predetermined animations. The user-controllable car had to be implemented with no physics, which means it can move through objects like a ghost. This however, has not proven to be a significant drawback because it does not necessarily impact the data being created from the simulation. Although the user needs to be a bit more mindful when driving the car, so as to not drive straight through a solid object.

The movement of the car is also not completely like that of a real car, however it should approximate it close enough that the generated point cloud data is just as useful.

The moving pedestrians does a good job of approximating walking motion, however its shape is currently made up of several smaller simple geometric shapes like cubes and spheres, making it resemble a posing doll used by artists. It is always possible to build a finer, more detailed human shape if that is needed. Any object recognition algorithm that functions properly should be able to recognize the moving pedestrian, if it was existed in the real world.

5.2.1 Performance of the Collision Detection

When shapes were approximated using compound colliders instead of mesh colliders, it could be seen from the performance tests that mesh colliders increased the total use of CPU time (39 %) allocated by the physics parts, then what the compound colliders did (27.7 %). This difference could differ greatly and depends on the amount of triangles there are in an object's mesh. The more detailed an object is, the more triangles there are.

In our case there was only 194 triangles in the object's mesh and only 500 objects of them. This can be considered to be small, for reasons that games can contain thousands of objects with greater details and more triangles in its mesh.

The approximation also depends on how detailed a compound collider is. For our case that used a visualization tool for external use, decided to do more detailed approximations (using more primitive colliders), this gives us better visuals of the object, for the price of less performance gain. Taking this into consideration, the amount of objects and the amount of triangles in the object's mesh, complicates the validation of the performance.

In general, it is concluded that if it was possible to approximate an object with hundreds or thousands of triangles in its mesh, with just a few primitive colliders, and if the approximation of its shape is close enough, then the performance increase makes it worth it.

5.3 Evaluation of Point Cloud Data Management

In this section the various parts of the managing of the point cloud data is discussed. This includes the various visualization methods and the performance of the exportation.

5.3.1 Evaluation of Visualization Methods

The visualizations that were created with the available resources proved that a point cloud representation was a suitable visualization technique for the data that the simulated sensor creates. The usability of which can be distinguished by simply reviewing a set of points in the visualization and comparing to the environment in which it was collected (see figure 4.5). When the sensor resolution is set to a high setting the visualization is indistinguishable from the actual scene with the physical characteristics in mind.

It was also shown that the two different approaches for visualizing the point cloud were usable in different applications. These consisted of using a built in particle system or rendering a surface mesh from the points. Initially the mesh approach was deemed the most suitable for both of the visualization options. However, due to the fact that mesh based representation was not optimal for visualizing a point cloud for the real time visualization, the particle system approach was used. However, the approach of using the mesh still applied to the post-simulation visualization and was as such used. This allowed the visualization of a large number of points without causing major performance drawbacks. Although, the time required for loading the points is still evident, and could potentially be solved by paralleling the tasks.

As such, the use of either a particle system or rendering points as a mesh, is a suitable approach for visualizing a point cloud. Although the two approaches have both advantages and disadvantages were both was suitable for this thesis.

5.3.2 Export Performance

As mentioned in section 4.2.4, the memory of the of the computer which the simulation is running on sets the limitation on the amount of data that can be created and saved. This memory limitation could be circumvented by implementing a cache on the hard drive, for example by periodically saving the points in memory to the disk.

5.4 Usability of the World Editor and User Interface

The constructed world editor was implemented with the features expected to be desired by future users, allowing them to set up interesting and realistic scenarios. Moreover the simulated lidar sensor was implemented with a tool to adjust relevant parameters to match different kinds of commonly used lidar sensor models. The combination of these components provides features to generate realistic data for a wide range of applications.

The tool for adjusting the lidar sensor was shown to be complex for a user to understand, as proven by the lack of understanding by the first test users. These complications were solved by visualizing the adjustments to the user in a way which was validated to be effective. Further, by focusing on designing the layout using the SSLP methodology and excluding unnecessary functions, the resulting interface became easy to interpret and easy to use.

As the usability of the user interface in some sense came with the trade-off with

extended functionality, one still has to consider if the exclusion of every function was worth it. As the implemented functions gives users the tools needed to generate many different and quite specific data sets, the current exclusions are considered a good trade-off. However, in future development more functions could be worth implementing to give users more control over the simulator.

5.5 Future Development

While the current simulation supports a number of different scenarios, and have fulfilled the purpose of the thesis, it could still be extended with the following functionality;

1. More models for covering a wider range of entities in the scene.
2. Various scenes
3. Weather conditions
4. Increased realism.

In order to support a higher number of possible scenarios, additional entities may be added, these will increase the possibilities of situations that the simulation supports. These may include but are not limited to; children, wildlife and trucks. This will, for example, allow for simulations where children are present or when a deer suddenly runs out into the road.

Further, the simulation currently contain a single scene and could thus be extended with additional scenes. For allowing various topographies and cityscapes.

A real life lidar sensor can have its resolution diluted in weather conditions such as snow or rain when the rays hit a snowflake or a raindrop. To allow for testing in such conditions, weather conditions could be added into the simulation, scrambling the collected data to resemble real weather conditions.

The movement of the pedestrians could be made more realistic by basing the animation on motion capture data, as opposed to being animated by hand. This would make motions life-like, and increase the realism of the simulation. The animation system could also be used to animate other moving things, like trees or animals.

The physics of the cars could also be improved and a traffic system for controlling the cars in the road network, providing AI-like behavior for the other cars in the scenes. This reduces the manual work that the user needs to perform in order to have realistic traffic conditions.

These improvements would make the application more modifiable and support a larger set of possible traffic situations. However, given the limited amount of re-

sources during the project, such as time, these were not implemented.

6

Conclusion

The purpose of this thesis is to create a software implementation of a lidar sensor, with a virtual environment for the sensor to scan. This was to be implemented using a game engine with ray-casting functionality in mind.

It has been concluded that a software model of a real lidar sensor can be created using Unity Engine with community available resources. This sensor can be configured to match the specifications of the real equivalent. Further, the product offers export functionality, visualization options and modifiable scenarios. The final product is, due to the configurable resolution of the sensor, able to run on a wide range of computers with various specifications.

As sensors such as the Velodyne-HDL64 requires financial means to acquire, thus limiting research surrounding this to parties with sufficient resources, a realistic simulation will open up the field of research for independent researchers. Due to this, this thesis has the potential of broadening the spectrum of researchers working with lidar data by allowing them to set up their own traffic conditions, thus not limiting them to pre-existing data-sets.

Bibliography

[1] J. Bell Rae and A. K. Binder, *Automotive industry - the modern industry*, 2005. [Online]. Available: https://global.britannica.com/topic/automotive-industry/The-modern-industry.

[2] *Automotive industry - growth - european commission*, Growth, Feb. 2017. [Online]. Available: https://ec.europa.eu/growth/sectors/automotive (visited on Feb. 7, 2017).

[3] B. W. Smith, *Human error as a cause of vehicle crashes*, 2013. [Online]. Available: http://cyberlaw.stanford.edu/blog/2013/12/human-error-cause-vehicle-crashes (visited on Feb. 7, 2017).

[4] The Tesla Team, *All tesla cars being produced now have full self-driving hardware*, Nov. 2016. [Online]. Available: https://www.tesla.com/blog/all-tesla-cars-being-produced-now-have-full-self-driving-hardware (visited on Feb. 7, 2017).

[5] Volvo-Cars, *Autonomous driving*, 2017. [Online]. Available: http://www.volvocars.com/intl/about/our-innovation-brands/intellisafe/autonomous-driving.

[6] B. Schweber, *The autonomous car: A diverse array of sensors drives navigation, driving, and performance | mouser*, 2017. [Online]. Available: http://eu.mouser.com/applications/autonomous-car-sensors-drive-performance/.

[7] P. E. Ross, *Velodyne plans a lidar megafactory*, 2017. [Online]. Available: http://spectrum.ieee.org/cars-that-think/transportation/sensors/velodyne-to-build-lidar-megafactory.

[8] R. Rasshofer, M. Spies, and H., Spies, "Influences of weather phenomena on automotive laser radar systems", *Advances in Radio Science*, vol. 9, pp. 49–60, 2011. DOI: 10.5194/ars-9-49-2011.

[9] Bluesky International Limited, *How lidar works*, 2017. [Online]. Available: http://www.lidar-uk.com/how-lidar-works/ (visited on Feb. 7, 2017).

[10] Unity Technologies, *Unity - engine features*, 2017. [Online]. Available: https://unity3d.com/unity/engine-features (visited on Mar. 15, 2017).

[11] E. games, *Engine features*, 2017. [Online]. Available: https://docs.unrealengine.com/latest/INT/Engine/.

[12] R. Featherstone, *Rigid body dynamics algorithms*, 1st ed. Springer, 2008.

[13] A. Aktas and E. Orcun, "A survey of computer game development", *The Journal of Defense Modeling and Simulation: Applications, Methodology, Technology*, vol. 13, no. 2, DOI: 10.1177/1548512914554405.

[14] Eberly, David H., "Physics engines", *Game Physics*, pp. 295–538, 2010. DOI: 10.1016/b978-0-12-374903-1.00006-2.

[15] *About polygon meshes*, 2017. [Online]. Available: http://softimage.wiki.softimage.com/xsidocs/poly_basic_PolygonMeshes.htm.

[16] M. Soriano, 2017. [Online]. Available: http://alumni.cs.ucr.edu/~sorianom/cs134_09win/lab5.htm.

[17] A. Beane, *3d animation essentials*, 1st ed. John Wiley & Sons, 2012, ch. 6.

[18] R. T. Michael T. Goodrich, *Data structures and algorithms in java*. John Wiley and Sons, Inc, 2011, ISBN: 978-0-470-39880-7.

[19] Unity Technologies, *Unity - manual: Inner workings of the navigation system*, 2017. [Online]. Available: https://docs.unity3d.com/Manual/nav-InnerWorkings.html.

[20] J. Bonastre, *Unity support: Why should i use threads instead of coroutines?*, 2017. [Online]. Available: https://support.unity3d.com/hc/en-us/articles/208707516-Why-should-I-use-Threads-instead-of-Coroutines-.

[21] Unity Technologies, *Unity - manual: Colliders*, 2017. [Online]. Available: https://docs.unity3d.com/Manual/CollidersOverview.html (visited on Apr. 19, 2017).

[22] R. Aguiar, *Physics best practices*, 2017. [Online]. Available: https://unity3d.com/learn/tutorials/topics/physics/physics-best-practices.

[23] Unity Technologies, *Unity - manual: Wheel collider*, 2017. [Online]. Available: https://docs.unity3d.com/Manual/class-WheelCollider.html.

[24] ——, *Unity - scripting api: Rigidbody.addforce*, 2017. [Online]. Available: https://docs.unity3d.com/ScriptReference/Rigidbody.AddForce.html.

[25] E. B. Koffman, P. A. T. Wolfgang, and E. B. Koffman, *Data structures*, 1st ed. John Wiley, 2010.

[26] E. Rowell, *Know thy complexities!*, 2017. [Online]. Available: http://bigocheatsheet.com/.

[27] T. A. Sugiura Takayuki and M. Okutom, "3d surface reconstruction from point-and-line cloud", *2015 International Conference on 3D Vision*, 2015. DOI: 10.1109/3dv.2015.37.

[28] M. Unbescheiden and A. Trembilski, "Cloud simulation in virtual environments", *Proceedings. IEEE 1998 Virtual Reality Annual International Symposium (Cat. No.98CB36180)*, DOI: 10.1109/vrais.1998.658451.

[29] C. Merrick, *Benchmark of csv, json, and avro*, 2017. [Online]. Available: https://blog.stitchdata.com/redshift-database-benchmarks-copy-performance-of-csv-json-and-avro-9062f71f8148.

[30] Y. Shafranovich, *Common format and mime type for comma-separated values (csv) files*, 2005. [Online]. Available: https://www.ietf.org/rfc/rfc4180.txt (visited on Mar. 13, 2017).

[31] J. Nielsen, *Usability 101: Introduction to usability*, 2017. [Online]. Available: https://www.nngroup.com/articles/usability-101-introduction-to-usability/.

[32] A. Feizi and C. Y. Wong, "Usability of user interface styles for learning a graphical software application", *2012 International Conference on Computer & Information Science (ICCIS)*, 2012. DOI: 10.1109/iccisci.2012.6297188.

[33] D. A. Norman, *The design of everyday things*, 1st ed. Basic Books.

[34] M. Bohgard, S. Karlsson, E. Love´n, L.-Å. Mikaelsson, L. Mårtensson, A.-L. Osvalder, L. Rose, and P. Ulfvengren, *Arbete och teknik på människans villkor*, 1st ed. Prevent, 2015.

[35] R. Muther, *Simplified systematic layout planning*, 1st ed. MANAGEMENT and IND. RESEAR, 1984.

[36] Velodyne LiDAR, *Hdl-64e*, 2017. [Online]. Available: http://velodynelidar.com/hdl-64e.html.

[37] C. S. Andreas Geiger Philip Lenz and R. Urtasun, "Vision meets robotics: The kitti dataset", *International Journal of Robotics Research (IJRR)*, 2013.

[38] *Microsoft api and reference catalog - dictionary.add*, 2017. [Online]. Available: https://msdn.microsoft.com/en-us/library/k7z0zy8k(v=vs.110).aspx#Anchor_2.

[39] K. Dimitrov, *Rendering a point cloud inside unity*, 2014. [Online]. Available: http://www.kamend.com/2014/05/rendering-a-point-cloud-inside-unity/.

[40] Unity Technologies, *Unity - manual: Particle system*, 2017. [Online]. Available: https://docs.unity3d.com/Manual/class-ParticleSystem.html (visited on Mar. 15, 2017).

[41] ——, *Unity asset store*, 2017. [Online]. Available: https://www.assetstore.unity3d.com/en/#!/.

[42] Blender Foundation, *About - blender.org*, 2017. [Online]. Available: https://www.blender.org/about/.

[43] The MakeHuman team, 2017. [Online]. Available: http://www.makehuman.org/.

[44] Unity Technologies, *Unity - manual: Materials, shaders & textures*, 2017. [Online]. Available: https://docs.unity3d.com/Manual/Shaders.html.

[45] J. Levinson, J. Askeland, J. Becker, J. Dolson, and et al., "Towards fully autonomous driving: Systems and algorithms", *2011 IEEE Intelligent Vehicles Symposium (IV)*, 2011. DOI: 10.1109/ivs.2011.5940562.

[46] A. Ess, K. Schindler, B. Leibe, and et al, "Object detection and tracking for autonomous navigation in dynamic environments", *The International Journal of Robotics Research*, vol. 29, no. 14, 2010. DOI: 10.1177/0278364910365417.

A

Appendix 1

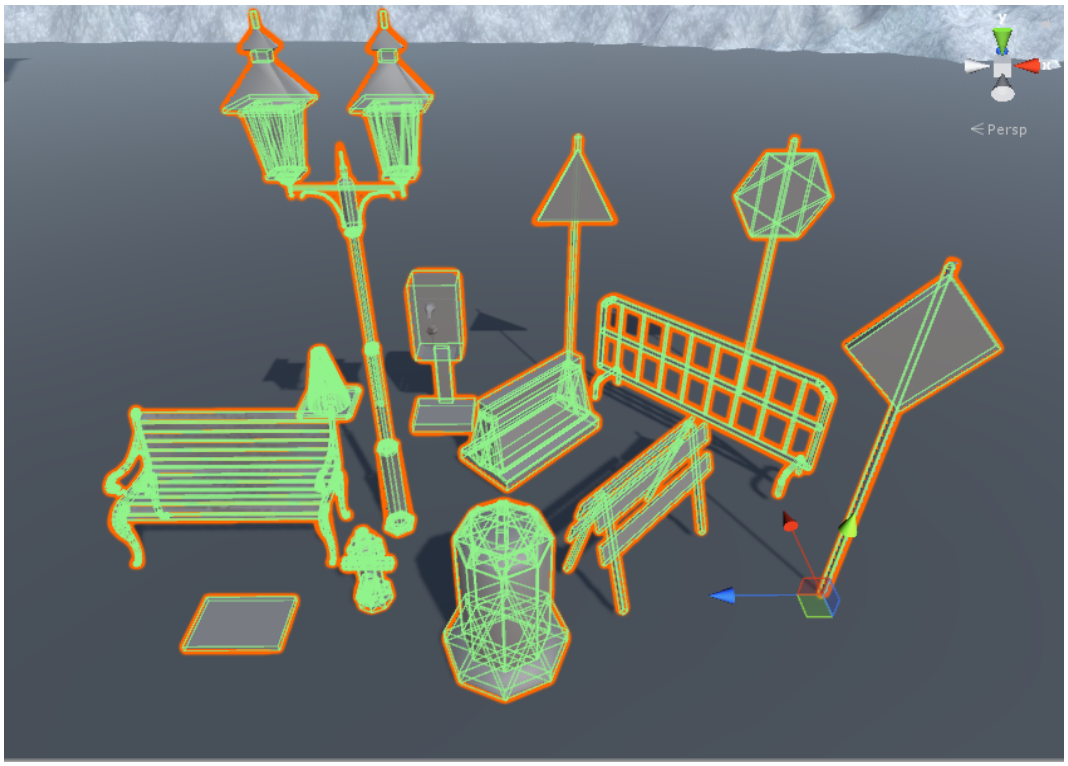


Figure A.1: Different models, each one built with compound colliders

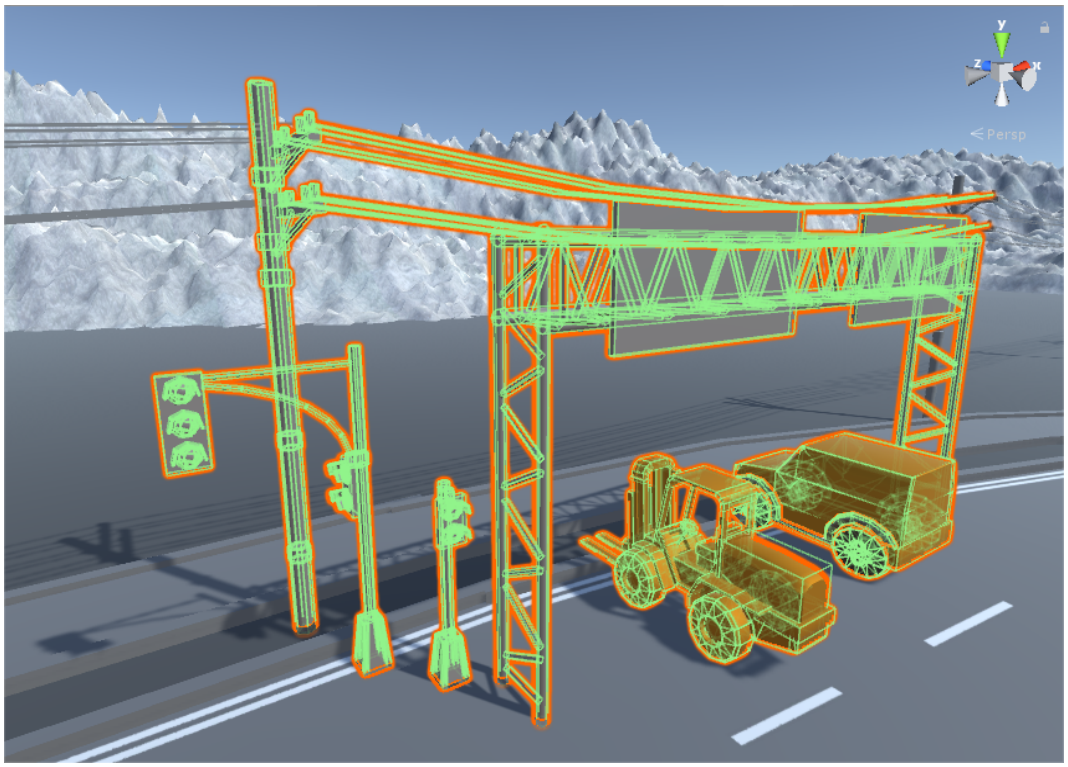


Figure A.2: Different models of vehicles and street elements, each one build with compound colliders

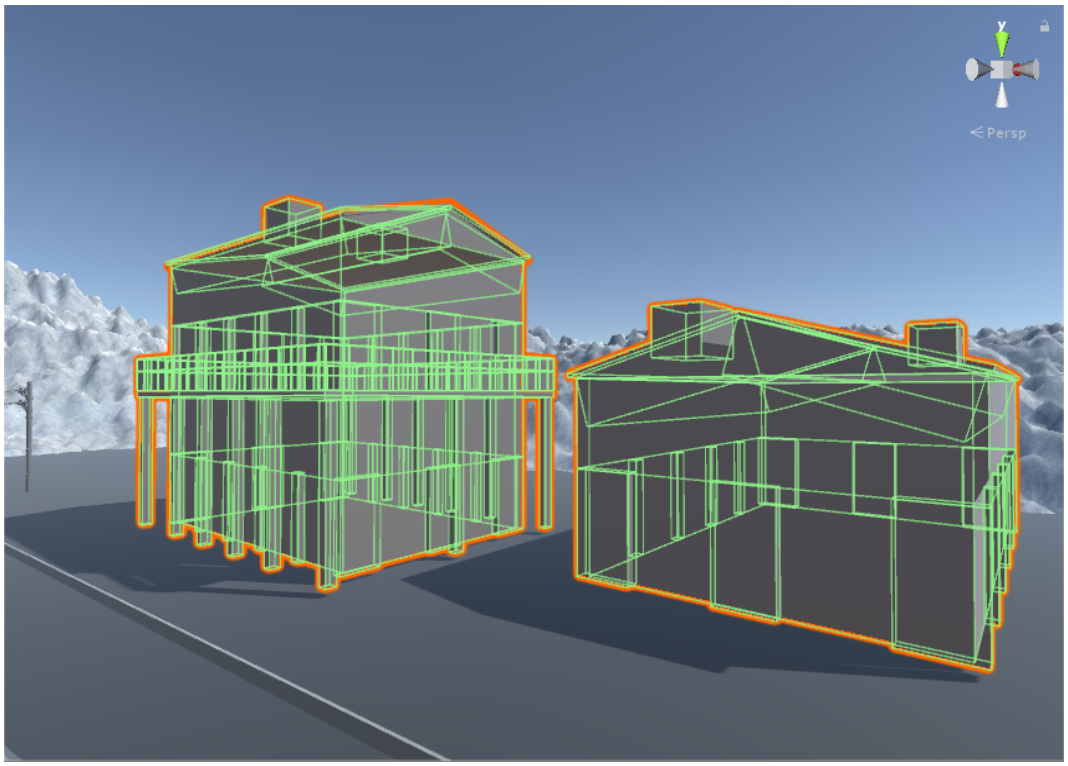


Figure A.3: Models of buildings, each one built with compound colliders