FREIE UNIVERSITÄT BERLIN

SOFTWARE PROJECT

AutoNOMOS Model Car Simulation Using Unity3D

Authors:
Benjamin KAHL
Abbas MOHAMMED MURREY

Supervisors:
Prof. Dr. Daniel GÖHRING
Stephan SUNDERMANN

Summer Semester 2019

Dahlem Center for Machine Learning and Robotics

Department of Mathematics and Computer Science

October 7, 2019

Preface

The modern-day surge in autonomous driving research and development has given new-found importance to the virtual simulation of such systems.

Self-driving cars typically use combined data received from a wide range of sensors to construct a model of their surrounding environment. Commonly employed sensors include LiDAR, GPS, stereoscopic cameras and inertial measurement units. The sensory data from these is interpreted by a series of control systems which identify obstacles, navigation paths and signage to extrapolate the respectively expected driving-motion and set the vehicles actuators accordingly.

Being both cost-intensive and dangerous to test on real-world vehicles, the development of such systems is often carried out with a simulator or a miniaturized model.

1.1 AutoNOMOS Model Car

The *AutoNOMOS Model* car (henceforth referred to as *AMC*) is a 1:10 model vehicle developed at the *Freie Universität Berlin* for educational purposes.[2] Equipped with the sensors most commonly used in self-driving cars, it is meant to be programmed to drive fully autonomously.

To students, these are only available in limited numbers and only whilst present at the FU-Berlin robotics laboratory. The goal of this project is to provide a simulation that allows students to run and test their programs without immediate access to an actual AMC unit.

1.2 Project Goal

Most freely distributed simulation frameworks are based on complex physics-modelling meant to provide a high degree of physical fidelity. The down-scaled and simplified nature of the AMC makes most of this complexity redundant and computationally expensive.

Phenomena such as *oversteering* or *understeering* rarely occur within the limited confines of the AMC model due to the highly limited speed and weight possibilities.

In addition to the unnecessarily complex physics-computations, precise measurements of the vehicles physical parameters, such as weight distribution, suspension geometry and friction coefficients are required for a simulation to run smoothly. These values are not always readily available or can be subject to change.

Here we present the development of a mathematically lightweight, real-time simulation software designed specifically for usage with an AutoNOMOS Model car.

The completed project as well as its documentation can be found under following url: `https://github.com/Helliaca/AutoModelCar_Simulator`

Introduction

A fundamental facet of the proposed simulation is for a virtual vehicle to mimic the real-world movement of an AMC when delivered the same actuator parameters.

In order to circumvent the complex physical formulae required to reproduce car-movement on a tire-friction basis, we employ a highly simplified, deterministic model based on *Ackermann steering geometry*.

This chapter covers the mathematical basis of said model as well as a list generally defined goals the software aims to achieve.

2.1 Project Scope

As to not loose track of our progress, we set a list of clearly defined requirements the application aims to fulfill. In detail, the software needs to provide:

- Support for most common sensor types. Notably LiDAR, camera, GPS and motor-ticks.
- A reasonable approximation of real-world sensor data, given a respective virtual environment.
- A reasonable approximation for car movement relative to values passed to actuators.
- Allow controller programs to run like regular ROS nodes, making a real AMC and a simulated one interchangeable at will.
- Support for multi-car simulations.
- Allow additional sensors to be mounted on vehicles or placed arbitrarily into the environment.
- Allow obstacles and other environmental objects to be placed or moved at will.
- Allow for easy adaptability in case of changes to the car. (Such as different ROS topic names or datatypes)
- Provide the same results independent of client hardware or simulation performance.
- A simple and user-friendly UI that requires no expertise in how the underlying code functions.

Furthermore, the API and all code needs to be documented and unit-tested so that it can be maintained in the long term.

2.2 AMC API

Based on our past experience with ROS and AMCs, we outlined a rudimentary API of input- and output-topics.

Each simulated vehicle will be subject to the values of a *steering* and a *speed* topic, which are externally adjusted by the end-users control-program. These topics are not published to by the simulator.

Sensors can be mounted on vehicles or placed into the environment. Each sensor publishes its measured data to a singular topic. The following four sensor types are commonly mounted on each AMC:

- **Camera:** Continuously publishes images as seen by the camera. The AMC employs a stereoscopic camera and includes a depth-cloud. For the purposes of this simulation, rudimentary camera images will suffice.
- **Lidar:** Performs circular scans with a given frequency and publishes measured data after each scan. The data is provided in the form of a list of distances sorted by angle.
- **Gps:** Continuously publishes the current position and angular predicament of the sensor itself and, as an extension, of the vehicle it is attached to. Positional values depend on an arbitrarily chosen origin point and axis-arrangement.
- **Ticks:** Continuously publishes the 'ticks' the vehicles primary motor underwent in the last time-interval. A faster turning-speed will equate to a greater amount of ticks and vice versa.

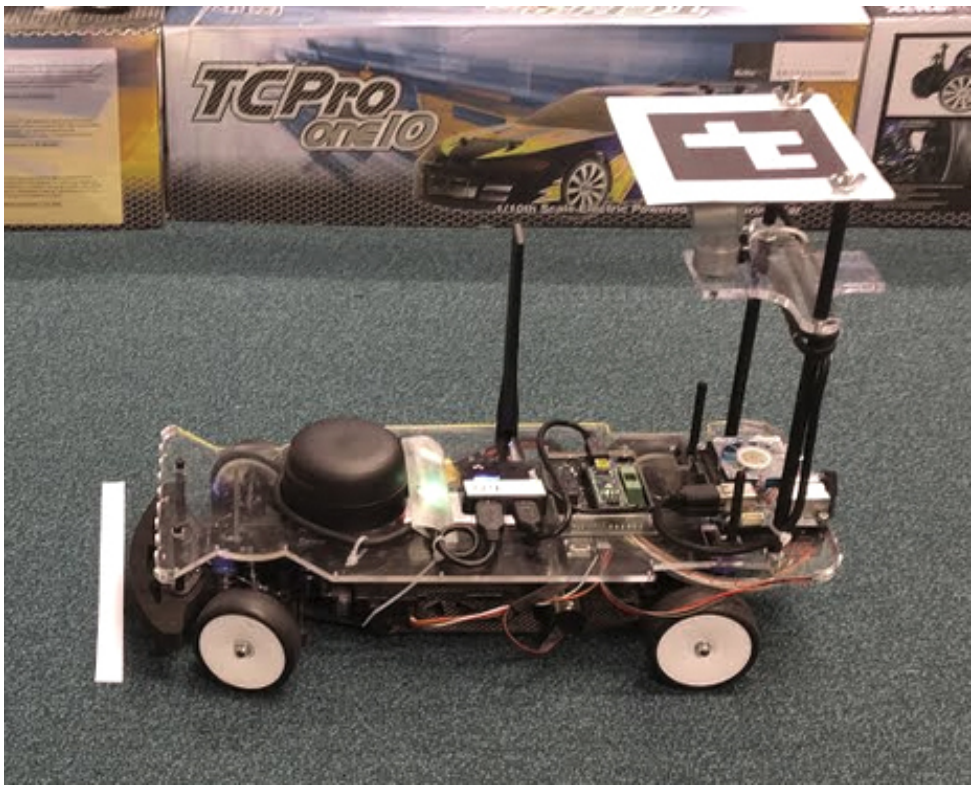

FIGURE 2.1: The "AutoNOMOS Model" Car (AMC) positioned at a starting line

2.3 Ackermann Model

At any given time, the differential rotation and position of a simulated car is derived from the *Ackermann steering model*.

Initially developed by Georg Lankensperger in 1817 and subsequently patented by Rudolph Ackermann in 1818, the Ackermann model is designed to solve the problem of two wheels having different turning radii despite being on the same axle of a four-wheeled vehicle.

For the purposes of this simulation we utilize it to calculate the expected turning circle of a car, provided that parameters such as inter-wheel distances and steering angles are available. Working under the assumption that a vehicle solely moves in a circular path around a point dramatically simplifies the required calculations for movement prediction.

2.3.1 Turning Angles

As can be observed in figure 2.2, angling both front wheels of a car equally when performing a steering maneuver would lead to the individual wheel-trajectories intersecting, as all tires follow the same turning radius but on different offsets.

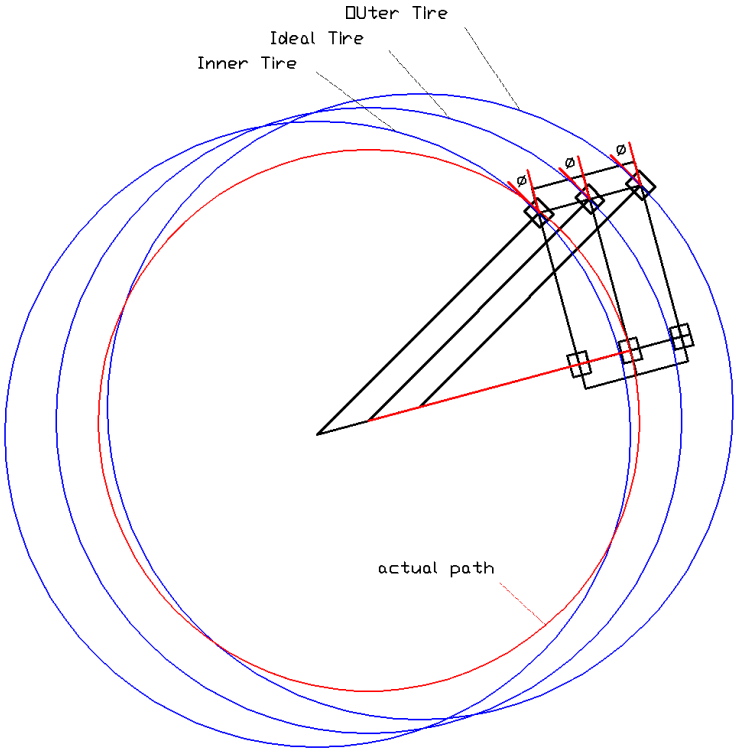

FIGURE 2.2: A Situation when turning both front tires by the same angle

Instead, to avoid the friction produced from keeping the tires off of their natural path, each tire is ascribed its individual angle of rotation derived from the angle of a virtual front wheel that lies at the center of the vehicle (Henceforth called the *ideal tire*). This predicament is depicted in figure 2.3.

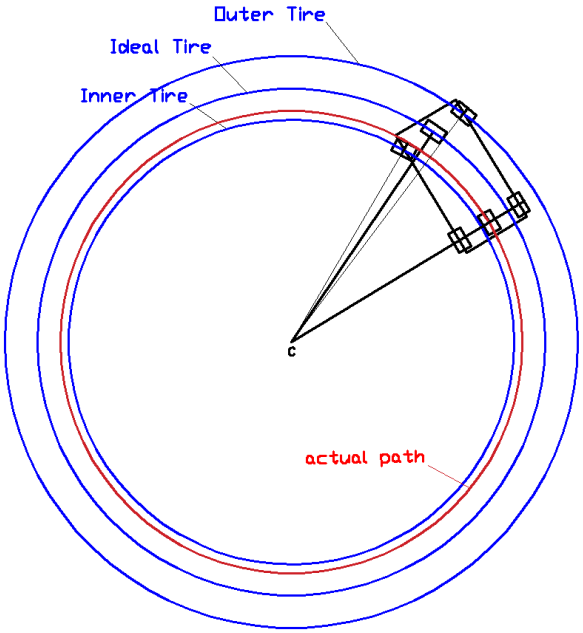

FIGURE 2.3: When using Ackermann steering model

The complete motion of a vehicle can thus be described by a rotation of the center-back wheel around C, where C corresponds to the intersection point of two lines, one along the front axle and the other along the back axle.

The angle of rotation performed at each simulation step would be proportional to the cars velocity and the time of motion.

The approach of regarding all car movement as a simple rotation has the added benefit of providing deterministic results, independent of frame-rate. Unlike when performing individual steps of rotation and subsequent translation, here the car never leaves the given circle, even at multiple revolutions per calculated frame.

Given ϕ as the angular disposition of a front tire and L as the distance to the respective back-tire, rudimentary trigonometry yields the distance r between the back-tire and the turning center:

$$\tan(\phi) = \frac{L}{r} \qquad \Rightarrow \qquad r = \frac{L}{tan(\phi)} \tag{2.1}$$

The same equation applies to each tire-pair, yielding following three equations:

$$\phi_0 = \arctan\left(\frac{L}{r+W}\right) \tag{2.2}$$

$$\phi = \arctan\left(\frac{L}{r}\right) \tag{2.3}$$

$$\phi_1 = \arctan\left(\frac{L}{r-W}\right) \tag{2.4}$$

where W corresponds to half of the axle length, as shown in figure 2.4.

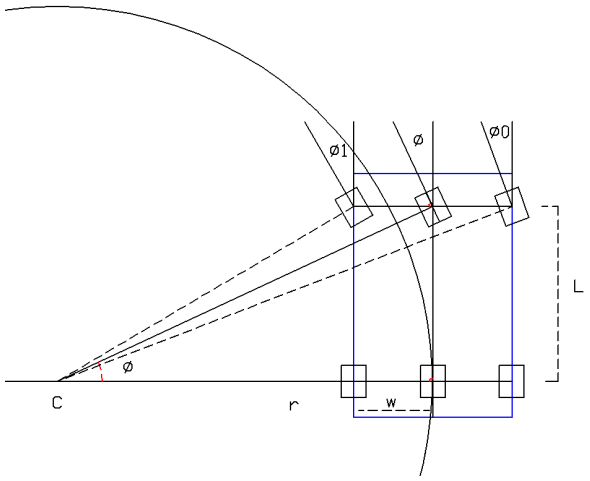

FIGURE 2.4: Turning each tire by a different angle

2.3.2 Center of Rotation

Given the distance r as calculated in the previous section, the center of rotation C is now trivial to obtain:

Assuming the origin of the vehicle to be located at its ideal back tire B, denoting its forward-vector as $\overrightarrow{D}$ and the global upwards axis as $\overrightarrow{U}$, we can form their cross product and obtain a vector $\overrightarrow{V}$ pointing along the axle of the back tire. (See fig. 2.5)

$$\overrightarrow{V} = \overrightarrow{U} \times \overrightarrow{D} \tag{2.5}$$

Normalizing $\overrightarrow{V}$ and scaling it by the radius r yields the vector pointing from the tire to the center of rotation:

$$C = \overrightarrow{B} + r * \overrightarrow{V} \tag{2.6}$$

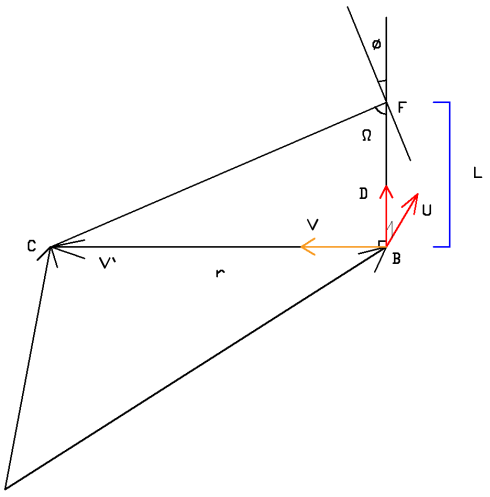

FIGURE 2.5: Obtaining the center of rotation C

Dependencies

3.1 Unity3D

While initially contemplating the possibility of building an OpenGL-based simulator from scratch, we ultimately decided to utilize the powerful tools provided by the Unity Engine as a fundament for the necessary rendering and collision calculations.

The Unity Engine[5] is developed by Unity Technologies and provides a highly flexible framework for all kinds of real-time 3D rendering software, including games, simulations as well as architecture and engineering tools.

The engine was chosen over its peers due to the convenience of providing a plethora of assets and tools to accelerate the development process, helping us create a finished and usable application within the given, limited time-span.

Furthermore, the manifold of supported platforms would allow us to deploy the finished software to various systems as well as expand upon it through the integrated support of controllers, VR-hardware, mobile support etc.

3.2 RosSharp

RosSharp (or simply ROS#) is a set of C#-based open source libraries that enable easy communication between ROS and .NET applications.

Provided by Siemens[4], the library includes ready-to-go classes for a variety of uses such as publishing of laserscan- and camera-topics as well as Urdf parsing.

Given its seamless integration with the Unity Engine, ROS# was chosen as our primary interface with the ROS environment.

3.3 ROSBridge

In order to establish a reliable communication between ROS# and ROS itself, a RosBridge server is employed.

Rosbridge provides a programming language agnostic, JSON-based API to ROS functionalities for non-ROS programs. Topics and their values become accessible through websockets and are sent in accordance to the rosbridge protocol.

The rosbridge-suite package is a collection of packages that implement the rosbridge protocol, which comes by default with any regular installation of the ROS melodic distribution.

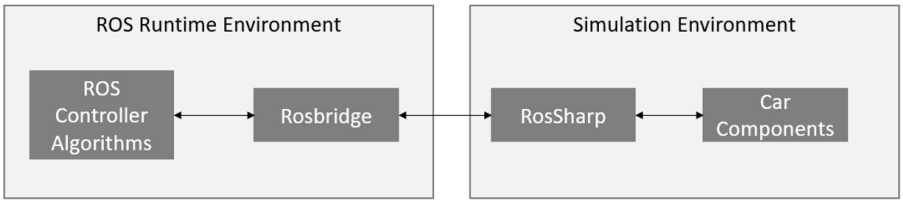

FIGURE 3.1: Overview of the simulation API-pipeline

Implementation Details

4.1 Program Structure

A globally defined *Anchor* object acts as the central hub that all simulation components are connected to.

The *RosConnector* object, used to publish and subscribe to topics, is referenced here as well as an API for console output.

In addition, a list of props and cars define the additional objects present in the scene. Props are simple obstacle objects with a 3D model and defined collision boundaries. Cars act exactly the same as props, except that they can also be moved through the steering- and speed-topics.

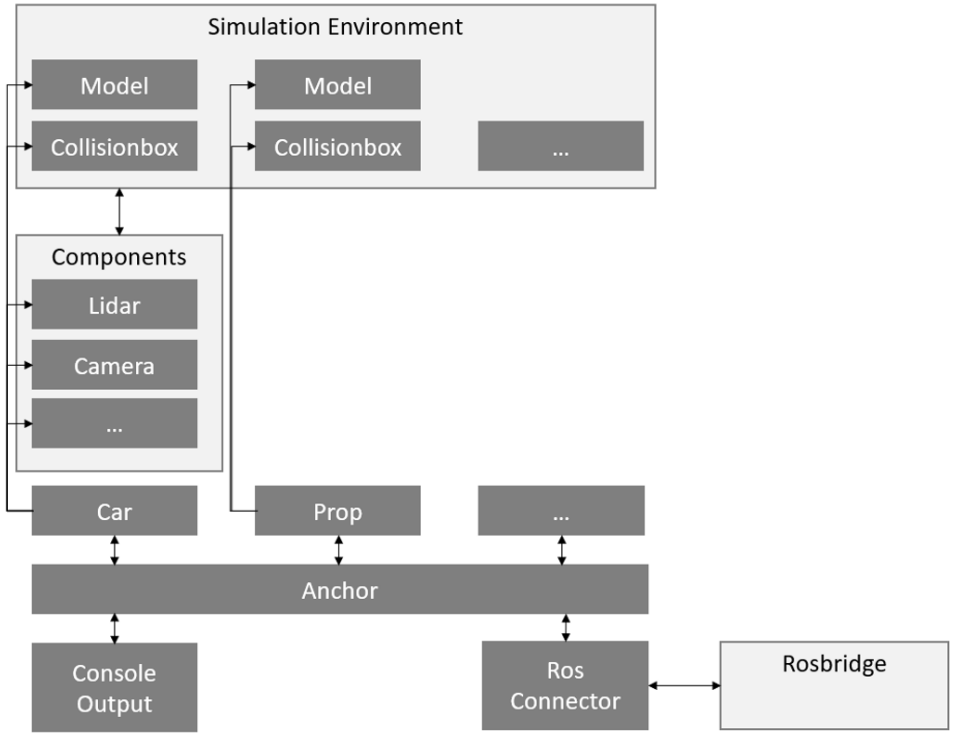

FIGURE 4.1: Program structure overview

4.2 API

To communicate with the rosbridge-server we employ a ROS# object of the *RosConnector* class. All components publish and subscribe to topics exclusively through this interface. This section describes the implemented APIs accessible by external ROS control programs.

4.2.1 Initial interface

The initially implemented vehicle interface followed an older (outdated) archetype used by AMCs:

topic name	type	description	rw
`/manual_control/speed`	`Int16`	Speed actuator	Subscribe
`/steering`	`UInt8`	Steering actuator	Subscribe
`/localization/odom`	`nav_msgs.Odometry`	Car position and rotation	Publish
`/ticks`	`UInt8`	Motor ticks	Publish
`/scan`	`sensor_msgs.LaserScan`	Lidar scan	Publish
`/camera/color/image_raw`	`sensor_msgs.Image`	Camera image	Publish

TABLE 4.1: Initial topic API

The base ROS# version provides most datatypes used in ROS applications, but the specialized types such as UInt8 and Int16 were not recognized. In order to provide a seamless plug between simulation and reality, the simulated vehicle had to recognize and use these types.

For this purpose the ROS# project was accordingly modified and a newly compiled .dll file substituted.

4.2.2 Autominy interface

AutoMiny[1] is a ROS-based software stack meant to facilitate and standardize the utilization of AMCs.

A newer, updated simulation API using the corresponding autominy datatypes was introduced at a later stage of the development process and is mostly identical with the actual API provided by an AMC:

topic name	type	description	rw
`/actuators/steering_pwm`	`autominy_msgs.SteeringPWMCommand`	Steering actuator	Sub
`/actuators/steering`	`autominy_msgs.SteeringCommand`	Steering actuator	Sub
`/actuators/steering_normalized`	`autominy_msgs.NormalizedSteeringCommand`	Steering actuator	Sub
`/actuators/speed_pwm`	`autominy_msgs.SpeedPWMCommand`	Speed actuator	Sub
`/actuators/speed`	`autominy_msgs.SpeedCommand`	Speed actuator	Sub
`/actuators/speed_normalized`	`autominy_msgs.NormalizedSpeedCommand`	Speed actuator	Sub
`/communication/gps`	`nav_msgs.Odometry`	Car position and rotation	Pub
`/sensors/arduino/ticks`	`autominy_msgs.Tick`	Motor ticks	Pub
`/sensors/rplidar/scan`	`sensor_msgs.LaserScan`	Lidar scan	Pub
`/sensors/camera/color/image_raw`	`sensor_msgs.Image`	Camera image	Pub

TABLE 4.2: Initial topic API

The specialized autominy-datatypes had to be incorporated into the ROS# library in the same manner as described above. The forked ROS# repository that contains all of the autominy datatypes can be found under following url: `https://github.com/Helliaca/ros-sharp`

The topic names listed above are adopted by default, but may also be changed during run-time. A `settings.txt` file describes the utilized naming scheme that ascribes each component its default topic.

The three macro symbols `{ID}`, `{NAME}` and `{TYPE}` can be utilized here to include the id-number, name or type of a component in its topic-string. In multi-car setups, these may prevent several cars from publishing to the same topics by, for example, appending a `"/vehicle_{ID}"` as a prefix to each topic.

4.3 Simulation Environment

In order to ensure reasonable fidelity of a cars camera images, we reconstructed the FU-Berlin robotics laboratory as a virtual simulation environment. The 3D models and their UV coordinates were created using Blender 2.79[3]. Textures were condensed from a series of reference-photographs as well as images from the public domain.

Once assembled, regular Unity-native tools and shaders were used to apply appropriate lighting to these objects.

FIGURE 4.2: FU Berlin robotics lab (top) and virtual reconstruction (bottom)

For use-cases where camera images are less crucial, two further scenes were created each with substantially less detail and, as a result, far better performance.

FIGURE 4.3: Regular, detailed and minimal scene configurations

The UI includes an inspector panel which allows the user to perform simple modifications to the environment such as placing obstacles, attaching sensors and moving objects. UI elements will be presented in greater detail in section 4.5.

4.4 Car Components

Components (equivalent to sensors) can be attached to a car or obstacle. This section describes the implementation of each component type in detail.

4.4.1 Chassis

The chassis is responsible for a cars movement and is divided into two axles:

- The *steering axle* subscribes to the steering topics listed in table 4.2 and calculates the respective turning circle of the car by means described in chapter 2.

  The various interpolation-parameters of the different topics are listed in their respective settings files (such as `steeringaxle_interp_nrm.txt` for the normalized topic) and can be adjusted by the user.

- The *propulsion axle* derives the cars speed from the three speed topics listed in table 4.2, rotates the vehicle by the respective amount around the circle-center obtained from the steering axle and finally produces the appropriate amount of *ticks* in proportion to the distance travelled.

  Similarly to above, `speedaxle_interp_nrm.txt` and similar files define the interpolation values for speed parameters. How these values were obtained is described below.

4.4.2 Calibration Of Speed And Acceleration

To obtain a reasonable approximate for speed-interpolation and tick-frequency, we measured these values on three different AMCs, with the third one being discarded due to an unusually high discrepancy to the other ones.

The cars would drive in a circular path with a diameter less than $2m$ to allow high speeds to be measured as well.

Car#	129			123			126		
	Speed	**Ticks per Second**	**Speed m/s**	**Speed**	**Ticks per Second**	**Speed m/s**	**Speed**	**Ticks per Second**	**Speed m/s**
	0	0	0	0	0	0	0	0	0
	0.1	19	0.10982659	0.15	30	0.17647059	0.1	20	0.06153846
	0.2	66	0.38150289	0.2	52	0.30588235	0.2	125	0.38461538
	0.3	114	0.65895954	0.3	100	0.58823529	0.3	209	0.64307692
	0.4	163	0.94219653	0.4	146	0.85882353	0.4	315	0.96923077
	0.5	211	1.21965318	0.5	195	1.14705882	0.5	410	1.26153846
	0.6	259	1.49710983	0.6	244	1.43529412	0.6	#	#
	0.7	304	1.75722543	0.7	290	1.70588235	0.7	#	#
	0.8	353	2.04046243	0.8	340	2	0.8	#	#
	0.9	398	2.30057803	0.9	385	2.26470588	0.9	#	#
	1	448	2.58959538	1	440	2.58823529	1	#	#
Tick's Counter	173 t/m			170 t/m			325 t/m		

FIGURE 4.4: Performance of three car samples (First column corresponds to the value published to the normalized speed topic)

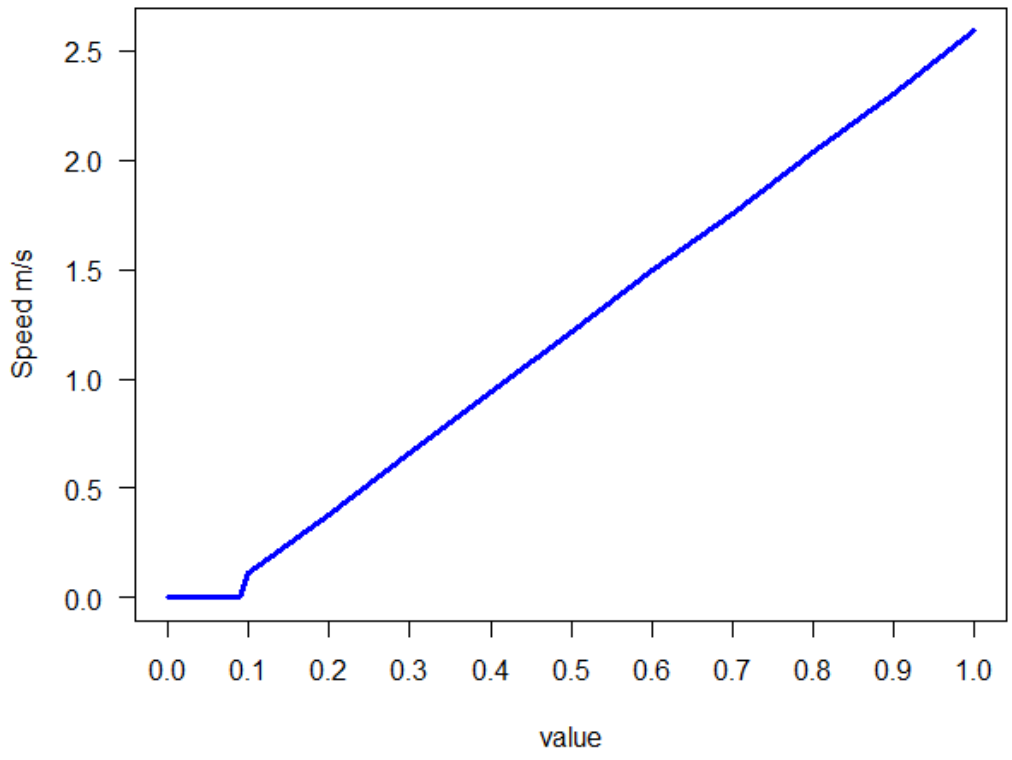

FIGURE 4.5: Reached speed m/s for given normalized value

For any values greater than 0.1, the relation is evidently a linear one. The interpolation-curve points for the normalized topic were thus adopted to mirror this graph. The regular speed topic would follow a simple 1:1 linear relation without any offsets.

4.4.3 Tick's Counter

The data collected in figure 4.4 also yields a reasonable estimate of ≈ 173 ticks produced per meter, or one tick emitted every 0.00578 meters.

$$\frac{1}{173} \quad \approx \quad 0.00578 \quad meters \tag{4.1}$$

In order to emit the adequate amount of ticks, a propulsion axle requires the respective distance travelled since the last emission. Given that the simulation moves vehicles exclusively on a rotation-basis, we need to calculate the arc-length of any given rotation in order to extrapolate the respective distance travelled.

We denote P_0 as the position of a car in previous frame, and P_1 as its position in the current frame. The distance by which the object has been moved from the previous frame to current frame is given by the following formula:

$$L = r * \phi[rad] \tag{4.2}$$

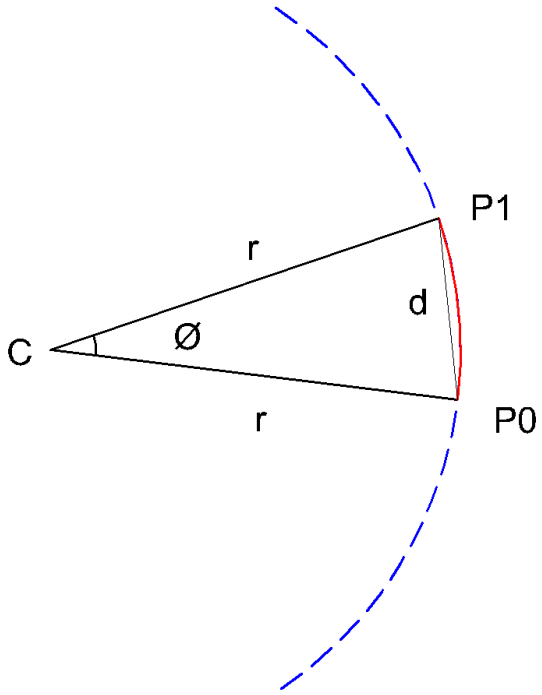

FIGURE 4.6: Arc-Length

This distance is proportional to the cars speed v by the time difference between P_0 and P_1, which we denote as Δt:

$$v * \Delta t = r * \phi \tag{4.3}$$

This yields the relation between the cars velocity (given by its speed topics) and the angle of rotation:

$$\frac{v * \Delta t}{r} = \phi \tag{4.4}$$

We accumulate the total distance travelled until it reaches the threshold given by (4.1), at which point we emit ticks equal to the following fraction:

$$\text{ticks emitted} = \left\lfloor \frac{\text{distance travelled}}{0.00578} \right\rfloor \tag{4.5}$$

The remainder of this division is kept for further accumulation in subsequent frames.

4.4.4 Camera

Camera images are rendered through a regular unity camera-object and compressed to a `jpeg` format before being published.

We chose a focal length of 43.6 (equivalent to 30.8 FOV) as a rough estimate from sample images produced by AMCs. This value as well as any other parameters are modifiable by the end user through UI elements (see 4.5).

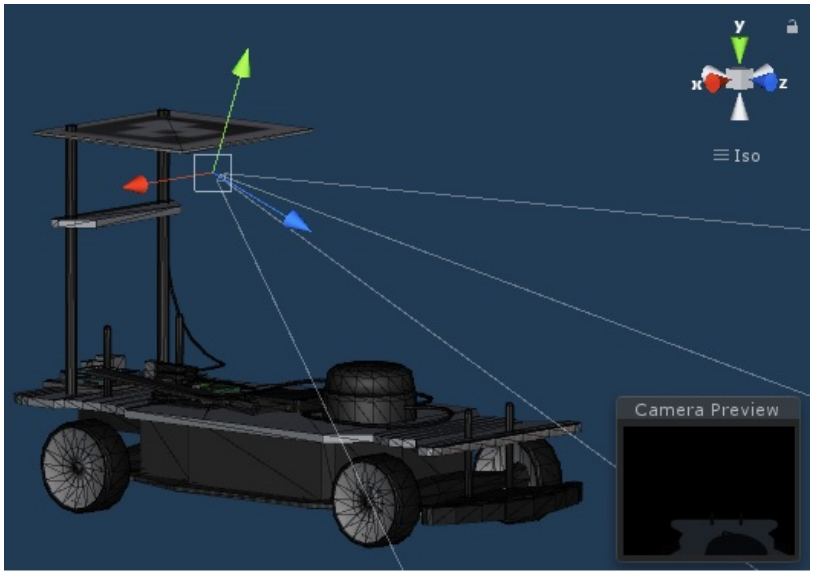

FIGURE 4.7: Camera position and rotation on a virtual AMC as seen in the Unity Editor

4.4.5 Lidar

The Unity engine provides a native `Raycast(...)` method which returns a point of contact on a collision-model when given a ray origin and propagation vector.

Carrying out a circular series of raycasts in compliance with the corresponding lidar parameters would provide a reasonable approximation for a real lidar reading. Conveniently, ROS# provides a *LaserScanReader* class specifically for this purpose as well as three different tools (lines, spheres and mesh) to visualize lidar-scans.

In accordance with the lidar mounted on AMCs, the virtual sensor takes 360 samples in counterclockwise order with a minimum range of 0.15m and maximum range of 16m. As with any other sensor, these default values can be adjusted for each virtual car as the user sees fit.

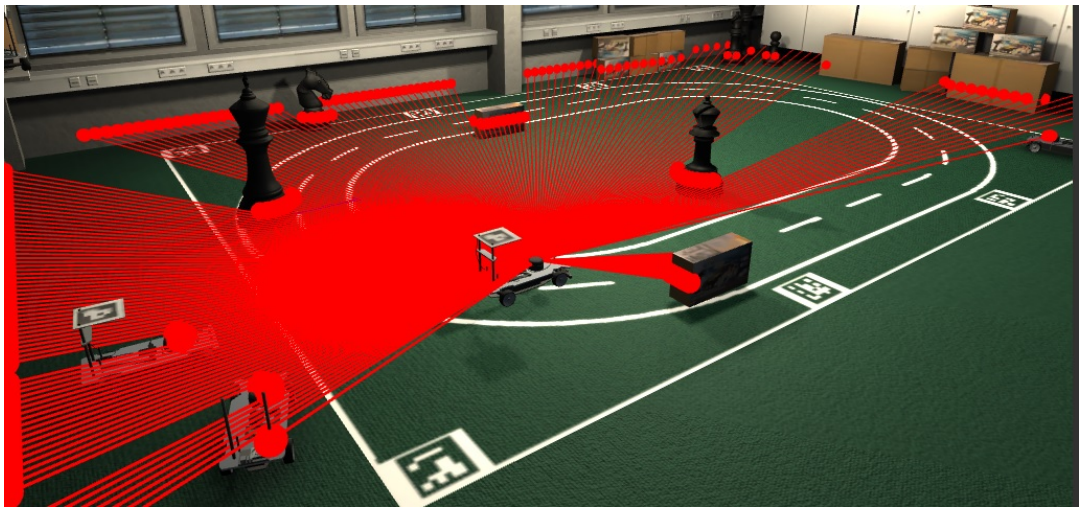

FIGURE 4.8: Virtual lidar data visualized with a set of lines and spheres at the impact point.

4.4.6 GPS

To avoid needless unit-conversion, we chose 1 Unity3d-unit to be equivalent to 1 meter in the real world. The GPS sensor would thus only have to broadcast the cars position in world coordinates.

However, the axes-arrangement of coordinates provided by an AMC differ from the axes-standard in the Unity engine, thus requiring a rudimentary coordinate-conversion process of swapping, as well as reversing, the y- and z-axis.

The quaternion describing the vehicles world-space rotation follows the same pattern.

4.4.7 Collision Detection

The simple fact that AMCs are typically not intended for utilization in crash tests makes the complex physics calculations of a realistic collision redundant to this project.

Instead, using native Unity collision-detection, we implemented a simple system that produces a warning-message and (optionally) sets a cars velocity to zero after it enters any obstacles collision box, displaying a rough estimate for the point of contact.

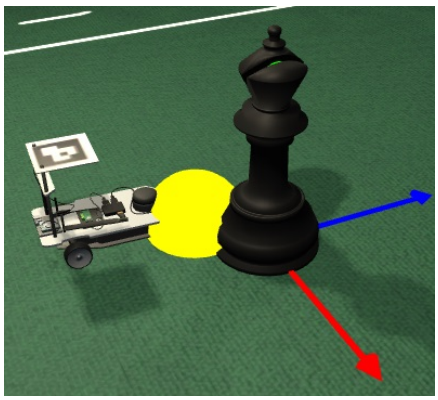

FIGURE 4.9: Point of contact marked with a yellow sphere post collision.

4.5 UI

A significantly large portion of development time was dedicated towards fleshing out a usable UI that would provide any necessary functionalities to the user without requiring a full recompilation of the program. These are organized into a total of 14 windows, each with its own distinct purpose.

Describing each in detail would go beyond the scope of this report, instead we discuss the most important ones here, leaving a more detailed description (as well as instructions of use) on the simulators github documentation (See [6]).

4.5.1 Inspector/Context Panel

The *inspector* panel serves as the primary tool to make changes to components or the scene itself. It displays a list of props each with their respectively attached components. Selecting any of these components brings up a *context* panel that allows the user to change any of its attributes. This includes moving objects about, rotating or scaling them as well as changing sensor-parameters.

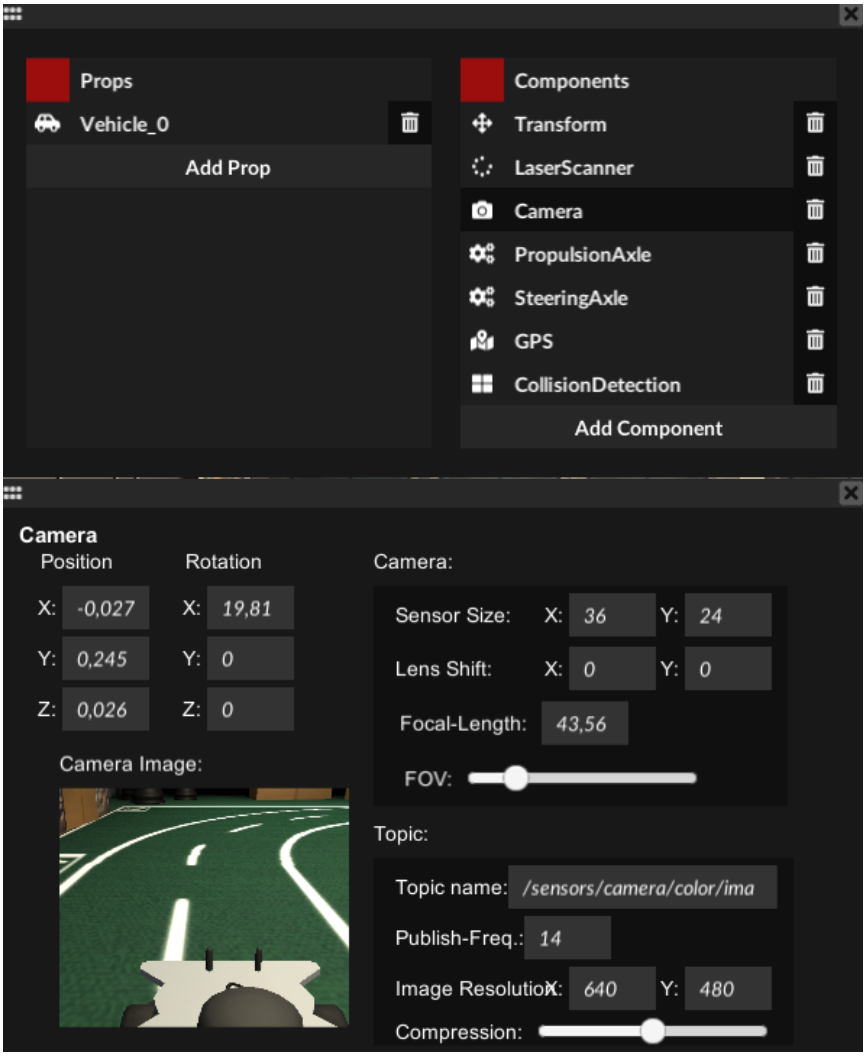

FIGURE 4.10: Inspector(top) and context(bottom) panels when selecting a camera components of a vehicle.

4.5.2 Utilities

In addition, the simulator provides a wide range of utility tools meant to facilitate generic tasks such as moving/placing cars, keeping track of topic values etc.

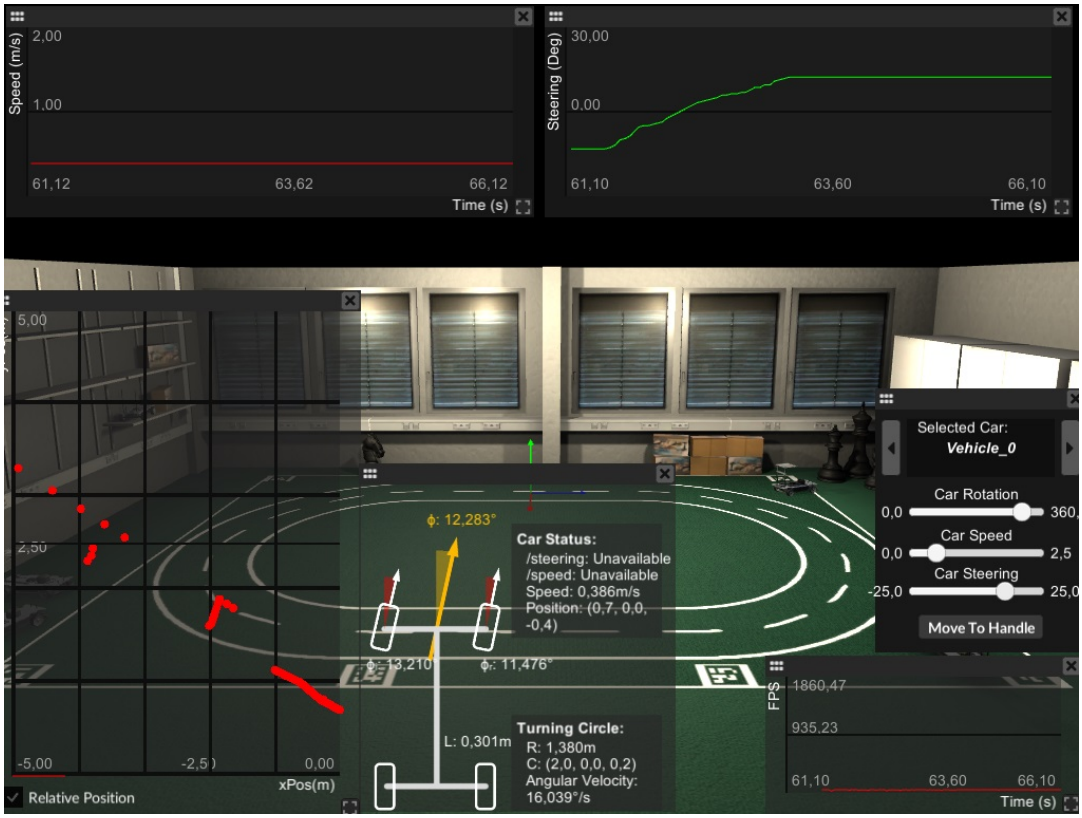

FIGURE 4.11: Some of the simulator utility tools in use.

4.5.3 Developers Console

Before implementing the bulk of UI elements, the programs core functionalities could be accessed through a command-shell-like console. It also serves as an output for generic information such as errors and warnings. A complete list of available commands can be found within the github documentation.

One may bring up this console by pressing the `LeftCtrl` or `Tab` key.

Result Evaluation

5.1 Sensor Data Comparison

5.1.1 Camera

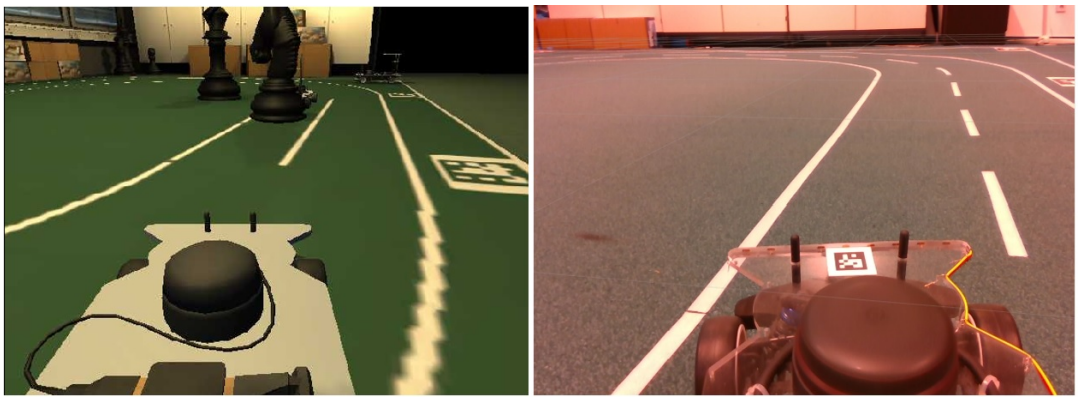

FIGURE 5.1: Camera image from simulation (left) and from reality (right)

Figure 5.1 shows camera images produced by a real AMC and a simulated AMC side-by-side. It becomes clear that the simulated image suffers from some jagged edges and aliasing issues as opposed to the real one. Upon further inspection some compression artifacts can be found as well.

These problems could be remedied by applying anti-aliasing filtering to the rendered image as well as reducing the used compression factor of 50%.

In addition, the real image is subject to a slight red hue, presumably produced by the mounted camera. Introducing this hue into the simulation would require changing the lighting settings and recalculating lighting textures or, alternatively, applying a color-correction filter to the rendered image.

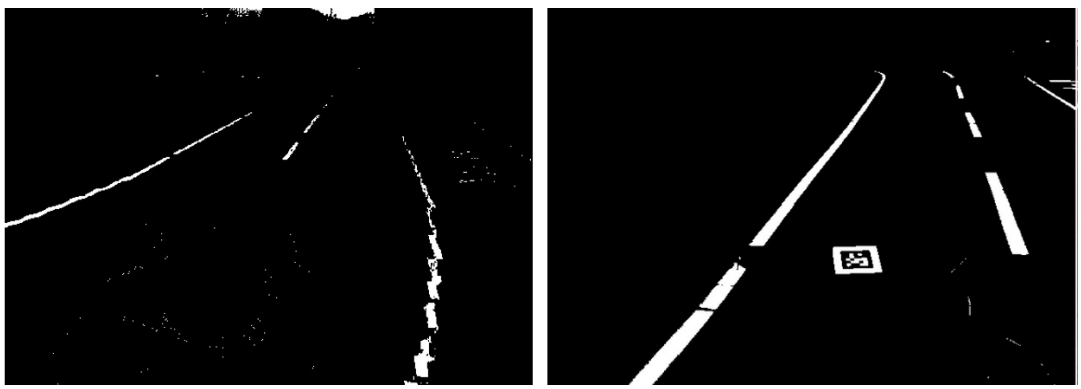

FIGURE 5.2: Camera image from simulation (left) and from reality (right) with an applied binary filter

As is commonly done in autonomous driving algorithms, applying a binary color filter yields a far smoother result on the real image than the simulated one, as seen in figure 5.2. A better result might be obtainable by using a higher resolution texture for the ground as well as implementing the above stated solutions.

5.1.2 Lidar

Figure 5.3 shows the path of a vehicle when running a simple obstacle avoidance program as well as the respective collected lidar data.

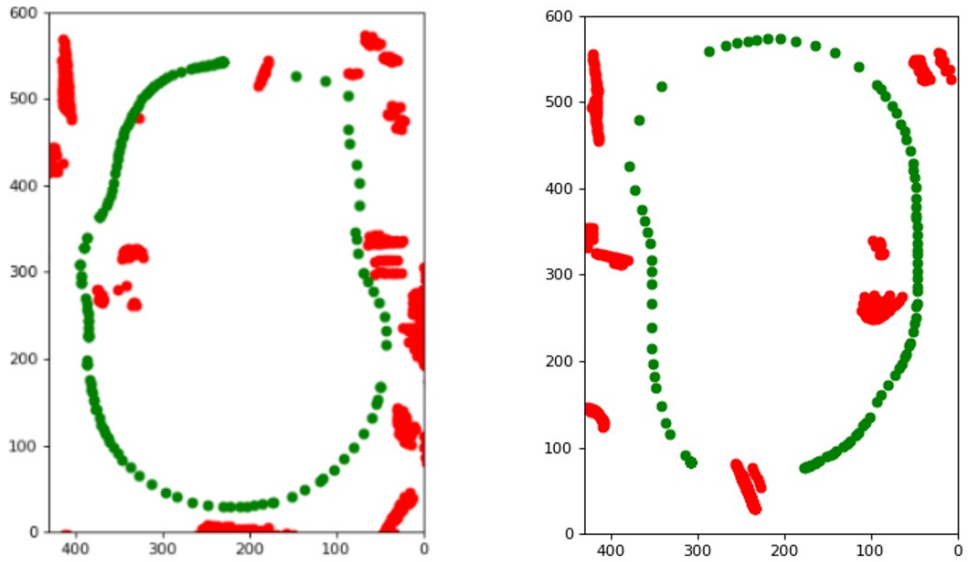

FIGURE 5.3: Reality (left) and simulation (right) when applying a simple obstacle avoidance program. Green dots are recorded vehicle positions and red dots are recorded lidar points.

The simulated data is far smoother but still fairly analogous to the real data. A simple way of increasing realism would be to introduce a random noise factor to measured lidar points.

5.2 Performance

The publishing frequency of sensors is, of course, bound by the applications overall frame-rate. Its heavy emphasis on image rendering makes the simulation heavily bottle-necked by the employed GPU, in particular when using multiple camera sensors.

Rough measurements for the most complex scene are listed below. When using the minimalist scene the frame-rate is significantly improved.

GPU type	Avg. frame-rate
High-end (Nvidia GTX 1070)	120
Low-end (Radeon HD 7550M)	40
Integrated	26

TABLE 5.1: Simulator frame-rates in dependence of the systems GPU when running the *detailed* scene.

5.3 Verdict

In total, we feel that most of the goals listed in chapter 2 have been met in a satisfactory way while still leaving some clear room for improvement. Here we list the most common points of contention:

- The emphasis on using simple maths instead of complex physics relieves some CPU stress found in other simulator frameworks such as *gazeebo*. However, the detailed environment puts a significant amount of strain on the GPU and produces camera images of limited quality.
- Simple changes to the car (like using slightly different lidar sensors) are simple to adjust in the simulator, but alterations on the underlying framework (like switching from regular datatypes to autominy datatypes) require a complete recompilation of the ROS# library as well as the simulator.
- Due to the rosbridge-based API the simulator is simple to set up even if running on a different machine than the controller program.
- We found that using a developers setup within the Unity Editor generally made using the simulator more flexible than using the built executable. This means that developers that are familiar with the Unity Engine can enjoy a highly flexible workflow and incorporate simple changes efficiently.
- The simulator can be easily expanded onto different platforms and hardware (mobile, VR, etc.) due to the flexibility of the Unity Engine.
- A core component missing from the simulated car is a depthcloud-component from the camera, making it unsuitable in specific use cases.
- The simplicity of moving and placing obstacles or sensors makes rudimentary changes to the environment an easy process.
- The simulator does not have a save/load function, making it highly session dependent. Example: Making lots of changes to car will have to be redone if the simulator is restarted.

References

[1] *Autominy repository and documentation*. URL: https://github.com/AutoMiny/AutoMiny.

[2] *Autonomos model car repository and documentation*. URL: https://github.com/AutoModelCar/AutoModelCarWiki/wiki.

[3] *Blender 2.79 release notes*. URL: https://www.blender.org/download/releases/2-79/.

[4] *Rossharp repository and documentation*. URL: https://github.com/siemens/ros-sharp.

[5] *Unity engine documentation*. URL: https://docs.unity3d.com/Manual/index.html.

[6] BENJAMIN KAHL, ABBAS MURREY, *Project github repository*, 2019. URL: https://github.com/Helliaca/AutoModelCar_Simulator.

[7] J.-S. ZHAO, Z.-J. LIU, AND J. DAI, *Design of an ackermann type steering mechanism*, Journal of Mechanical Engineering Science, 227 (2013).