Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Florian Dannenberg

Simulation abstrakter Umweltszenarien zur Validierung kamerabasierter Fahrspur- und Elementerkennungsalgorithmen für den Einsatz auf autonomen Modellfahrzeugen

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Florian Dannenberg

Simulation abstrakter Umweltszenarien zur Validierung kamerabasierter Fahrspur- und Elementerkennungsalgorithmen für den Einsatz auf autonomen Modellfahrzeugen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stephan Pareigis
Zweitgutachter: Prof. Dr. Tim Tiedemann

Eingereicht am: 25. April 2018

Florian Dannenberg

Thema der Arbeit

Simulation abstrakter Umweltszenarien zur Validierung kamerabasierter Fahrspur- und Elementerkennungsalgorithmen für den Einsatz auf autonomen Modellfahrzeugen

Stichworte

Carolo-Cup, Unity3D, Unreal Engine, Gazebo, QT, Modellfahrzeug, autonom, autonomes Fahren, Objekterkennung, Game-Engine, HiL, SiL, Simulation, FAUST

Kurzzusammenfassung

Diese Arbeit befasst sich mit der Verbesserung von Testbedingungen durch die Anbindung eines Simulators zur Generierung von Testbildern im Bachelor-Projekt Carolo-Cup. Im Rahmen des Projektes werden autonome Modellfahrzeuge für die Teilnahme am Carolo-Cup der TU Braunschweig entwickelt.

Nach einer Vorstellung des Carolo-Projektes werden zunächst die aktuellen Rahmenbedingungen, wie etwa das Regelwerk des Wettbewerbs und die Testbedingungen an der HAW, sowie die daraus resultierenden Probleme des Carolo-Projekts aufgezeigt. Im Anschluss wird ein Überblick über verschiedene Testansätze für Embedded Systeme gegeben (dies umfasst Hardware in the Loop, Software in the Loop und Live Testing), bevor aktuelle Game-Engines und ein Robotersimulator vorgestellt werden, die für die Nutzung in dieser Arbeit in Frage kommen. Schließlich soll mit Hilfe des Gazebo-Simulators aufgezeigt werden, wie eine Simulationsumgebung für die verschiedenen Umweltszenarien des Carolo-Cups erstellt werden kann. Die Möglichkeiten der Weiterentwicklung von kamerabasierten Erkennungsalgorithmen und die Testbedingungen für autonome Fahrzeuge sollen durch diese Arbeit signifikant verbessert werden.

Florian Dannenberg

Title of the paper

Simulation of abstract environments to validate camera-based lane and element detection algorithms to use on an autonomous model car

Keywords

Carolo-Cup, Unity3D, Unreal Engine, Gazebo, QT, model car, autonomous, autonomous driving, object detection, game-engine, HiL, SiL, simulation, FAUST

Abstract

This paper is concerned with improving the testing conditions within the Carolo-Cup project

through the addition of a simulator for generating testing images. Within the Carolo project, autonomous model vehicles are developed for participation in the Carolo-Cup at TU Braunschweig.

After briefly presenting the Carolo project, parameters such as the competition's set of rules and the testing conditions at the HAW are presented, including the resulting difficulties for the Carolo project. Subsequently, an overview of different approaches for the testing of embedded systems will be given, including Hardware in the Loop, Software in the Loop and live testing, before presenting current game engines and a robotic simulator which are considered regarding their usefulness for the project. Finally, the Gazebo simulator will be used to show how a simulation environment can be created for the different environmental scenarios included in the Carolo-Cup. The aim of this paper is to advance the development of camera-based recognition algorithms and thus improve testing conditions for autonomous vehicles.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Motivation . . . 1
1.2. Zielsetzung . . . 2

2. Projekt Carolo-Cup **3**
2.1. Carolo-Cup . . . 3
2.2. Projekt . . . 3
2.3. Fahrzeugplattformen . . . 5
2.3.1. Software . . . 5
2.3.2. Hardware . . . 7
2.4. Testbedingungen und Probleme . . . 11

3. Testvarianten von Embedded-Systemen im Projekt Carolo-Cup **13**
3.1. Zweck des Software-Testings . . . 13
3.2. Testarten . . . 14
3.2.1. Hardware in the Loop Testing . . . 14
3.2.2. Software in the Loop Testing . . . 14
3.2.3. Live Testing . . . 15
3.3. Anforderungen an eine Testumgebung im Rahmen des Carolo-Projektes . . . 15
3.4. Auswahl einer Testmethode . . . 17

4. Überblick über aktuelle Simulations / Game Engines **18**
4.1. Unity Game Engine . . . 18
4.2. Unreal Engine 4 . . . 20
4.3. V-Play Engine . . . 21
4.4. Gazebo . . . 22
4.5. Auswahl eines Frameworks . . . 24

5. Eingesetzte Frameworks **25**
5.1. Qt . . . 25
5.1.1. Funktionsweise . . . 25
5.1.2. Eingesetzte Module . . . 27
5.2. Gazebo . . . 29
5.2.1. Architektur . . . 29
5.2.2. Komponenten . . . 30
5.2.3. Eingesetzte Komponenten . . . 33

6. Umweltszenarien im Carolo-Cup **34**
6.1. Genereller Aufbau der Umwelt des Carolo-Cup 34
6.2. Freie Fahrt 37
6.3. Hinderniskurs 37
6.3.1. Landstraße 37
6.3.2. Suburbanes Szenario 39

7. Simulation der Umwelt **45**
7.1. Simulierte Umweltelemente 45
7.1.1. Fahrbahn 45
7.1.2. Hindernisse 47
7.1.3. Fußgänger 48
7.1.4. Planare Streckenelemente 48
7.1.5. Straßenverkehrsschilder 48
7.2. Modellierung „Fat Lady“ 49
7.2.1. Kameramodell 49
7.2.2. Fahrzeugmodell 49
7.3. Komposition der Carolo-World 50

8. Schnittstelle zur Carolo-Software **52**
8.1. Verbinden des Simulators 52
8.2. Benutzung des Gazebo-Simulators mit der Carolo-Software 55
8.3. Steuerung der Simulation durch die Carolo-Software 56
8.4. Vergleich mit Bestandsdaten 57

9. Fazit **61**

Anhang

A. Modelldefinition Kamera **64**

B. Modelldefinition Fat Lady **66**

C. Modelldefinition Carolo-World **72**

Abbildungsverzeichnis

2.1. Der Carolo-Cup 2018 . . . 4
2.2. Verschiedene Fahrzeuge des Carolo-Cup 2018 . . . 4
2.3. Schematische Darstellung der Softwarearchitektur . . . 6
2.4. Die Fahrzeugplattform „LaK-XU 5000“ . . . 8
2.5. Die Fahrzeugplattform „Fat Lady“ . . . 9
2.6. Schematische Darstellung der Anordnung der Controller bei „Fat Lady“ . . . 10
2.7. Schematische Darstellung der Anordnung der Hauptrecheneinheit bei „Fat Lady“ 11
2.8. Die Teststrecke der HAW Hamburg . . . 12

6.1. Definition der Parksituation . . . 35
6.2. Die verschiedenen Mittellinienmarkierungen des Carolo-Cup . . . 36
6.3. Kreuzungstypen auf der Landstraße . . . 38
6.4. Schematische Darstellung der Sperrfläche . . . 40
6.5. Schematische Darstellung des Zebrastreifens . . . 41
6.6. Dimensionen eines Fußgängers . . . 42
6.7. Die verschiedenen Kreuzungstypen des Carolo-Cup . . . 43

7.1. Die PNG-Textur der Fahrbahn . . . 46
7.2. Die komponierte Carolo-World . . . 51

8.1. Die GUI im Überblick . . . 56
8.2. Vergleichsszenario Linkskurve . . . 58
8.3. Vergleichsszenario Kreuzung . . . 59
8.4. Vergleichsszenario Geschwindigkeitslimit . . . 60

Listings

5.1. Include Befehl von SDF . . . 30
5.2. Beispiel World File (vgl. Open Source Robotics Foundation (2018d)) . . . 32
5.3. Beispiel Model File (vgl. Open Source Robotics Foundation (2018e)) . . . 32

7.1. Auszug aus der Modelldefinition der Fahrbahn . . . 47
7.2. Auszug aus der Modelldefinition für ein statisches Hindernis . . . 48

8.1. Benötigte includes . . . 52
8.2. Verbindungsaufbau zum Gazebo-Server . . . 53
8.3. Auszug aus der Methode imageReceived() . . . 54
8.4. Starten des Gazebo-Servers . . . 55
8.5. Starten des Gazebo-Clients . . . 55
8.6. Starten des Gazebo-Servers in Kombination mit dem Gazebo-Client . . . 56

A.1. Modelldefinition Kamera . . . 64

B.1. Modelldefinition Fat Lady . . . 66

C.1. Modelldefinition Carolo-World . . . 72

Danksagung

Danke Karla, ohne Dich und deine kritischen Anmerkungen hätte ich das nicht geschafft.

Danke Team NaИ, es ist immer wieder eine Freude mit euch. Vielen Dank euch für den vielen Input und die schönen Wochenenden, auch wenn es mal spät wurde.

Danke Enrico, du hast uns großartig unterstützt und uns dennoch immer unseren Freiraum gegeben.

1. Einleitung

Die HAW Hamburg hat, seit dem ersten Carolo-Cup der TU Braunschweig im Jahr 2007, jedes Jahr mindestens ein Team zur Teilnahme an dem studentischen Wettbewerb gestellt.

Zunächst waren die Teams unter dem Namen der Forschungsgruppe FAUST vertreten, bis im Jahr 2016, aus eigener Initiative heraus, dass Team NaИ parallel zu der Projektgruppe weiter an dem Projekt gearbeitet hat.

Durch den so auch entstandenen hochschulinternen Wettbewerb und die mittlerweile angesammelte Erfahrung im Team NaИ konnten immer wieder weitere, und auch neue, Ansätze für die Fahrzeugplattformen, Bilderkennungen oder auch Regelungen entwickelt und evaluiert werden.

1.1. Motivation

Autonomes Fahren wird immer präsenter in den Medien. Viele bekannte Fahrzeughersteller, aber auch Internetriesen wie Google, investieren derzeit zunehmend in die Forschung zu diesem Thema. Die Auswirkungen, die vor allem in der Transport und Logistikbranche durch das autonome Fahren erwartet werden, sind eine erhöhte Sicherheit im Straßenverkehr, geringerer Kraftstoffverbrauch sowie verbesserte Arbeitsbedingungen für Berufskraftfahrer (vgl. Daimler AG, 2014).

Eine so erreichte erhöhte Sicherheit im gewerblichen Verkehrswesen wirkt sich auch direkt positiv auf alle anderen Verkehrsteilnehmer aus. Die prognostizierte Rate von Unfällen pro gefahrenem Kilometer bei autonom fahrenden Fahrzeugen wird mit etwa der Hälfte im Vergleich zu herkömmlichen Fahrzeugen geschätzt (Blanco et al., 2016, S. 41). Dies hat direkte Auswirkungen auf den Straßenverkehr der Zukunft.

Die Konzepte, die im Rahmen des Carolo-Cups erstellt und mit Modellfahrzeugen in die Tat umgesetzt werden, sind eine wichtige Ideenschmiede für die Industrie und Forschung. Nicht ohne Grund nehmen sich wichtige Vertreter aus der Automobilbranche die Zeit, um in der Fachjury die Konzepte der verschiedenen Teams zu bewerten.

Auch wenn Unfälle während des Carolo-Cups - die durch Fahrfehler verursacht worden sind - selten Schäden nach sich ziehen, so sind es hier die Strafmeter die vermieden werden

sollten, um eine gute Platzierung zu erreichen. Hierdurch und aufgrund der Teilnahme im Carolo-Projekt und an den Wettkämpfen des Carolo-Cup besteht eine persönliche Motivation, zum allgemeinen Forschungsstand in diesem Gebiet beizutragen.

Da die Weiterentwicklungsmöglichkeiten innerhalb des Carolo-Projekts langsam an ihre Grenzen gestoßen sind, wurde die Idee geboren, es dem Team durch eine neue Testumgebung zu ermöglichen, die Entwicklung orts- und fahrzeugunabhängig und dadurch flexibler und zeitsparender zu gestalten. Um diese Idee in die Tat umzusetzen wurde das im nächsten Abschnitt beschriebene Ziel gesetzt, auf dem die folgende Arbeit basiert.

1.2. Zielsetzung

Das langfristige Ziel dieser Arbeit ist es, die Testbedingungen des Team NaИ für die Weiterentwicklung im Rahmen des Carolo-Projekts zu verbessern. Daraus ergibt sich die Fragestellung, wie die Testbedingungen nachhaltig verbessert werden können.

Der Konsens im Team ist, dass das Hauptproblem bei der gemeinsamen Arbeit die räumliche Verteilung und der nicht immer gegebene Zugriff auf die Fahrzeugplattformen ist (vgl. Abschnitt 2.4), weshalb die Ermöglichung von orts- und fahrzeugunabhängigem Testen signifikant zur Verbesserung der Testbedingungen beitragen würde.

Daraus ergibt sich das unmittelbare Ziel für diese Arbeit, ein Proof of Concept für die Ermöglichung von orts- und fahrzeugunabhängigem Testen zu schaffen. Unter dem Proof of Concept ist in dem Kontext dieser Arbeit die Auswahl eines geeigneten Frameworks sowie die Implementierung eines Prototypen mit der benötigten Kernfunktionalität zu verstehen (vgl. Rat für Forschung und Technologieentwicklung, 2013, S. 2).

Dafür soll auf Basis einer Simulations / Game Engine eine dem Carolo-Cup nachempfundene Umwelt simuliert werden. Zugleich soll das Wettbewerbsfahrzeug des Team NaИ für diese Umwelt modelliert werden, sodass eine grundlegende Simulation der Wettbewerbsbedingungen unabhängig von Zeit, Ort und Fahrzeugverfügbarkeit ermöglicht wird.

Durch die Integration eines Simulators in die übrige Softwarelandschaft des Carolo-Projekts erwartet das Team eine signifikante Verbesserung der Testbedingungen.

2. Projekt Carolo-Cup

Zu Beginn dieses Kapitels soll zunächst der Carolo-Cup, in dessen Rahmen das Forschungsprojekt FAUST - Fahrerassistenz und Autonome Systeme - unter anderem tätig ist, vorgestellt werden, um diese Arbeit in einen praktischen Kontext einzuordnen. Anschließend werden das Projekt, die Fahrzeugplattformen und die aktuellen Testbedingungen betrachtet, um danach die daraus resultierenden Probleme zu erläutern.

2.1. Carolo-Cup

Der Carolo-Cup ist ein studentischer Wettbewerb der Technischen Universität Braunschweig. Laut der offiziellen Veranstaltungswebseite (TU Braunschweig, 2018) ist es das Wettbewerbsziel, Studententeams die Möglichkeit zu bieten, sich mit der Entwicklung und Umsetzung von autonomen Modellfahrzeugen auseinanderzusetzen.

In unterschiedlichen komplexen Szenarien, die an die Realität angelehnt sind, messen sich die Studententeams vor einer Fachjury mit Experten aus Wirtschaft und Wissenschaft.

Als Szenario für den Wettbewerb werden die Teams von einem fiktiven Fahrzeughersteller beauftragt, ein Gesamtkonzept für ein autonomes Fahrzeug zu entwickeln. Dies soll am Beispiel eines Modellfahrzeugs im Maßstab 1:10 und unter Berücksichtigung der Kosten- und Energieeffizienz demonstriert werden.

Während des Wettbewerbs werden die erarbeiteten Konzepte in Präsentationen bewertet, sowie durch möglichst schnelles und fehlerfreies Fahren in verschiedenen Disziplinen, die in Kapitel 6 erläutert werden, Punkte gesammelt (siehe auch Abbildung 2.1 und Abbildung 2.2).

2.2. Projekt

Das Forschungsprojekt FAUST beschäftigt sich mit der Weiterentwicklung und Erforschung bestehender und neuer Technologien für autonome Fahrzeuge und Fahrerassistenzsysteme (vgl. Christophers, 2012).

Hierbei existieren verschiedene Themenschwerpunkte, in denen die Forschungsgruppe Technologien erforscht und entwickelt.

Abbildung 2.1.: Der Carolo-Cup 2018

Abbildung 2.2.: Verschiedene Fahrzeuge des Carolo-Cup 2018

Hierzu gehören:

- Sensorik, Telemetrie und Bildverarbeitung
- Echtzeit- und Bussysteme
- Software- und Hardwarearchitekturen
- Algorithmik und Steuerung

Ein Tätigkeitsfeld des Forschungsprojektes FAUST, in dem alle Themenschwerpunkte vereint werden, ist die Entwicklung eines autonomen Modellautos zur Teilnahme am Carolo-Cup. Hierfür wurde 2016 das Team NaИ gegründet.

Für die Teilnahme am Carolo-Cup wurden vom Team NaИ verschiedene Fahrzeugplattformen entwickelt und erprobt. Zwei der aktuellen Fahrzeugplattformen sollen im nächsten Abschnitt kurz vorgestellt werden.

2.3. Fahrzeugplattformen

Im Laufe der Zeit wurden mehrere Fahrzeugplattformen vom Team NaИ entwickelt, getestet und teilweise auch wieder verworfen. Aktuell befinden sich zwei Plattformen im Einsatz: das primäre Testfahrzeug „LaK-XU 5000“, sowie das für den Wettbewerb eingesetzte Fahrzeug „Fat Lady“.

Beide Fahrzeuge unterscheiden sich in essentiellen Punkten im Hardwareaufbau. Jedoch kann durch eine passend gewählte Softwarearchitektur die hauptsächlich eingesetzte Software nahezu identisch auf beiden Plattformen eingesetzt werden.

2.3.1. Software

Das eigens für das Carolo-Projekt erstellte Framework, welches auf den Fahrzeugen zum Einsatz kommt, ist in einer klassischen Drei-Schichten-Architektur aufgebaut und in C++ implementiert. Das Carolo-Framework basiert dabei auf dem Qt-Framework in der Version 5.4. Durch das System der Signale und Slots des Qt-Frameworks (siehe hierzu Abschnitt 5.1) ist eine zeit und ereignisgesteuerte Bearbeitung des Inputs möglich.

Die Schichten sind:

- Physikalische Schicht
- Logische Schicht

- Erkennungsschicht

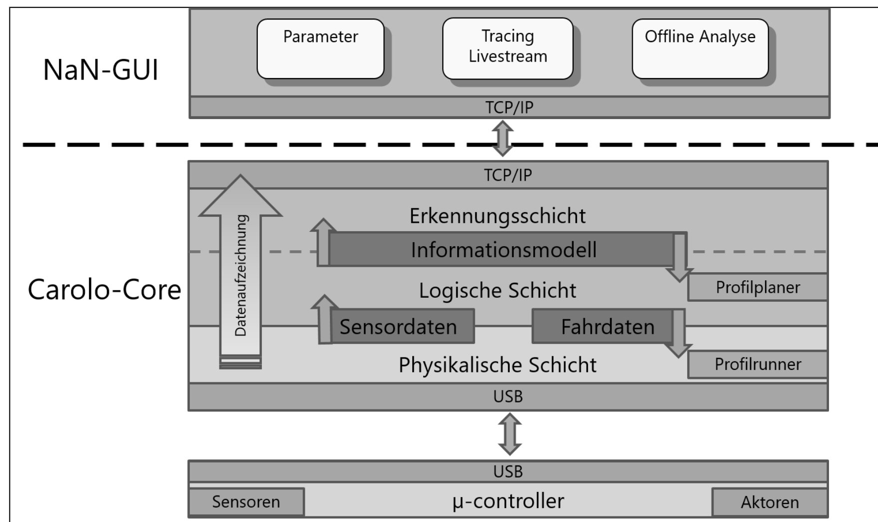


Abbildung 2.3.: Schematische Darstellung der Softwarearchitektur

Zur Validierung der selbst entwickelten Algorithmen wurde für den Einsatz auf den Entwickler-PCs eine Art vierte Schicht, die NaИ-GUI, hinzugefügt.

In der physikalischen Schicht (siehe Abbildung 2.3) sind die Hauptunterschiede der Software für die verschiedenen Fahrzeugplattformen zu finden. Dies ist die Abstraktionsebene in der die Fahrzeugeigenschaften, wie zum Beispiel Antrieb, Lenksensitivität oder Beleuchtung, aber auch die Kommunikation mit dem Mikrocontroller definiert werden. Die physikalische Schicht ist auch dafür zuständig, dass die in der logischen Schicht geplante Trajektorie vom jeweiligen Fahrzeug korrekt abgefahren wird.

Die logische Schicht verarbeitet die Daten und beeinflusst die Arbeitsweise der Erkennungsschicht. Die Einflussnahme auf die Erkennungsschicht geschieht anhand der Informationen aus der Erkennungsschicht und es werden in der logischen Schicht Annahmen getroffen, was als nächstes in der Erkennung zu erwarten ist, und diese entsprechend getriggert. Die Verarbeitung der Daten hat als Ergebnis die Trajektorie, welche der physikalischen Schicht übermittelt und von dieser abgefahren wird. Die logische Schicht ist für alle im Projekt eingesetzten Fahrzeuge gleich.

In der Erkennungsschicht werden alle Sensordaten gesammelt, ausgewertet und zu einem abstrakten Umweltmodell, der Szene, fusioniert. Eine solche Szene besteht aus diversen Szene-Items, welche einzelne Teile der Umwelt beschreiben. Jedes dieser Szene-Items kann von der logischen Schicht beeinflusst werden in Hinsicht darauf, ob es erwartet wird oder nicht. Ein Bei-

spiel hierfür soll der Zebrastreifen sein. Dieser wird nur in einem suburbanen Szenario erwartet (vgl. Abschnitt 6.3.2). Solange dieses Szenario nicht eingetreten ist, müssen die Daten nicht auf das Vorhandensein eines Zebrastreifens ausgewertet werden. Auch die Erkennungsschicht ist für alle im Projekt eingesetzten Fahrzeuge gleich.

Die vierte Schicht, die NaИ-GUI, kommt ausschließlich auf den Entwickler-PCs zum Einsatz. Sie dient zur Visualisierung von aufgezeichneten Fahrdaten und zur Steuerung der darunter liegenden logischen Schicht. So kann durch die NaИ-GUI ein aufgezeichnetes Bild geladen und an die Erkennungsschicht übergeben werden. Dieses wird dann in der Erkennungsschicht ausgewertet, bevor durch die logische Schicht eine Trajektorie geplant wird, welche dann wiederum in das angezeigte Bild in der NaИ-GUI eingezeichnet werden kann. Dadurch ist es dem Entwickler möglich, direktes visuelles Feedback für Anpassungen an Algorithmen oder Abläufen zu erhalten.

2.3.2. Hardware

Wie eingangs bereits angekündigt wurde, werden in diesem Abschnitt die zwei Farhrzeugplattformen, die im Team NaИ zum Einsatz kommen, vorgestellt. Dies sind zum einen das primäre Testfahrzeug „LaK-XU 5000“ und das Wettbewerbsfahrzeug „Fat Lady“.

LaK-XU 5000

Das Fahrzeug „LaK-XU 5000“ basiert auf dem Standard-Modellbausatz HPI TC-FD 4wd. Für den Einsatz als Testfahrzeug hat sich dieses Modell durch seine Robustheit und Langlebigkeit seit Gründung des Teams ausgezeichnet bewährt.

„LaK-XU 5000“ wird von einem Brushless Motor angetrieben. Hierbei wird über ein Zahnradgetriebe die Drehzahl reduziert und die Kraft über Zahnriemen an die beiden Differentialgetriebe an Vorder- und Hinterachse übertragen. Dadurch werden alle Räder von einem Motor angetrieben.

Für den Einsatz als autonomes Modellfahrzeug wurde das Modell um einen Turm kurz vor der Hinterachse erweitert (siehe Abbildung 2.4). An dem Turm ist der primäre Sensor, die monochrome Industriekamera UI-1226LE-M-GL der Firma IDS, angebracht (vgl. IDS, 2018).

Die Kamera arbeitet bei einer Auflösung von $752 * 480$ Pixeln mit einer Framerate von maximal 87,5 $\frac{Frames}{Sekunde}$. Die Lage kurz vor dem Drehpunkt des Fahrzeugs verringert die Auswirkungen eines Lenkwinkelwechsels auf den Bildinhalt. Betrieben wird die Kamera derzeit mit ~70 $\frac{Frames}{Sekunde}$. Aufgrund des Neigungswinkels der Kamera wird in der unteren Hälfte kein Mehrwert an Informationen erzeugt, da dort primär das eigene Fahrzeug zu sehen ist.

Abbildung 2.4.: Die Fahrzeugplattform „LaK-XU 5000“

Für eine schnellere Übertragung und Auswertung der Bildinformationen wurde daher entschieden, dass ausschließlich die obere Bildhälfte mit $752 * 240$ Pixel übertragen wird. Um die Fahrbahn möglichst in ganzer Breite zu erfassen wird das Weitwinkelobjektiv BT2120 der Firma Lensation eingesetzt (vgl. Lensation, 2017).

Des Weiteren kommen auf dem Fahrzeug Hall-Sensoren im Motor und ein Gyroskop zur Abschätzung der Eigenposition zum Einsatz. Die Schnittstelle zwischen den Aktoren, Sensoren und der Hauptrecheneinheit, einem ODROID-XU4 der Firma Hardkernel, wird über einen Teensy 3.1 Mikrocontroller realisiert. Ausgenommen ist hiervon die Kamera, welche direkt mit der Hauptrecheneinheit verbunden ist.

Der ODROID-XU4 ist ein ARM-basierter Einplatinencomputer mit einem maximalen Verbrauch von 20 Watt und wurde aufgrund seiner hohen Rechenleistung im Verhältnis zu Gewicht, Baugröße und Stromverbrauch ausgewählt.

Fat Lady

Das Wettbewerbsfahrzeug „Fat Lady“ ist im Gegensatz zum Testfahrzeug „LaK-XU 5000“ ein Eigenbau mit Gewichts- und Geschwindigkeitsoptimierung.

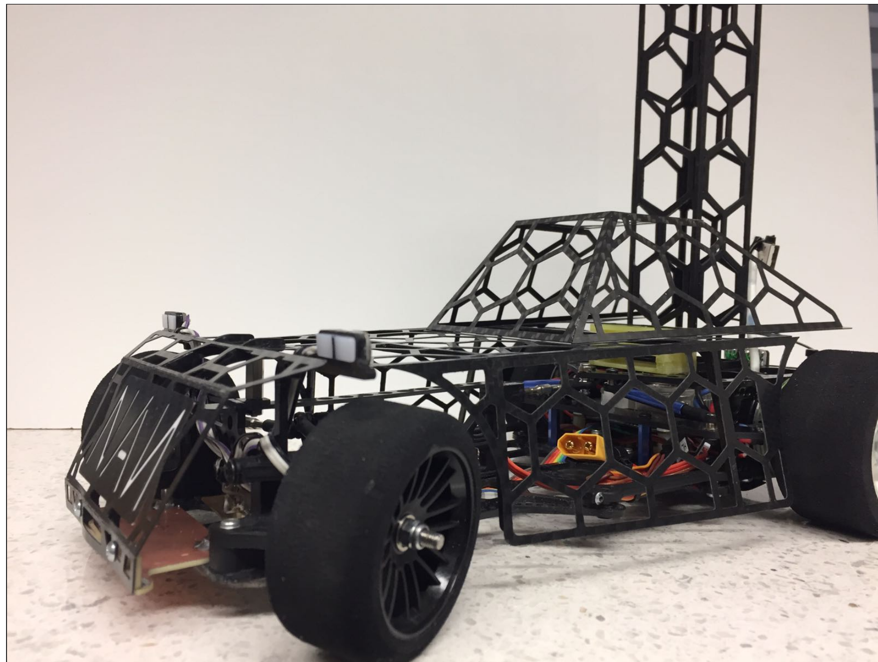


Abbildung 2.5.: Die Fahrzeugplattform „Fat Lady“

Durch die filigrane Carbon-Konstruktion und die möglichen höheren Geschwindigkeiten eignet sich das Fahrzeug primär zum Einsatz auf Strecken mit ausreichend Auslauffläche, so wie es auf der Wettbewerbsstrecke beim Carolo-Cup der Fall ist.

Die erste Besonderheit von „Fat Lady“ ist die Entkoppelung von Front und Heck. Dies macht es möglich, dass die Front bei Lenkbewegungen unabhängig vom Heck verkippt werden kann und stets alle Räder Bodenkontakt halten. Ein zweiter Vorteil hiervon ist, dass der Kameraturm stets in aufrechter Position verbleibt und das Kamerabild nicht zusätzlich durch Kippbewegungen gestört wird.

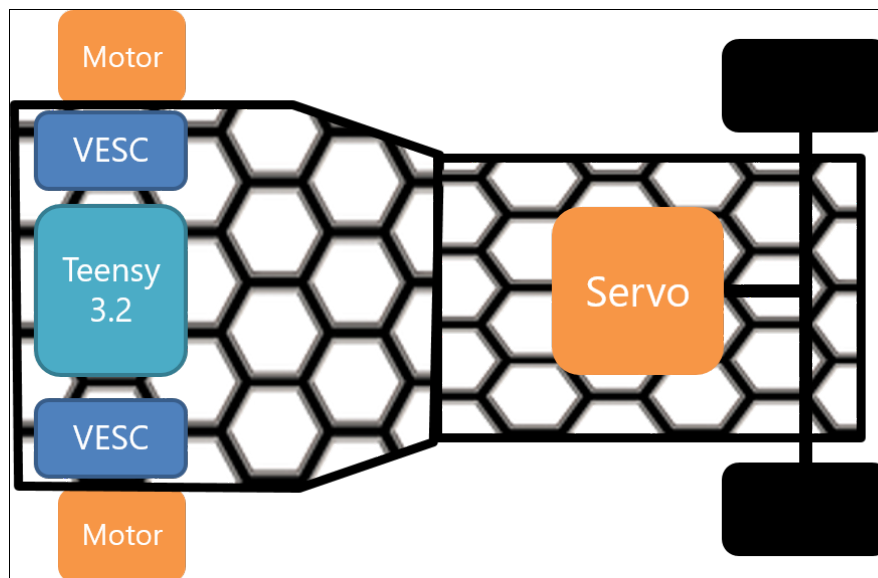


Abbildung 2.6.: Schematische Darstellung der Anordnung der Controller bei „Fat Lady“

Die zweite Besonderheit von „Fat Lady“ sind die einzeln angetriebenen Hinterräder. Dafür sind zwei separat ansteuerbare Motoren in den Hinterrädern verbaut (siehe Abbildung 2.6), die ursprünglich für den Einsatz in Quadcoptern entwickelt wurden. Jeder dieser Motoren hat eine maximale Leistung von 320 W, wodurch sich eine maximale Leistung von 640 W auf ein Gesamtgewicht von 0.95 kg ergibt. Mit dem Fokus der Gewichtsreduktion konnte das Team NaИ im Jahr 2018 bereits zum dritten Mal in Folge den vom VDI ausgelobten Sonderpreis für das leichteste Fahrzeug gewinnen (vgl. VDI, 2018).

Ein weiterer Unterschied zur Testplattform ist zudem der Einsatz des UP Core x86 Boards als Hauptrecheneinheit (siehe Abbildung 2.7). Dieses von einem Intel-Chipsatz betriebene Board hat den Vorteil, dass auf den Entwickler-PCs nativ kompiliert werden kann und keine Cross-Compiling Toolchain verwendet werden muss.

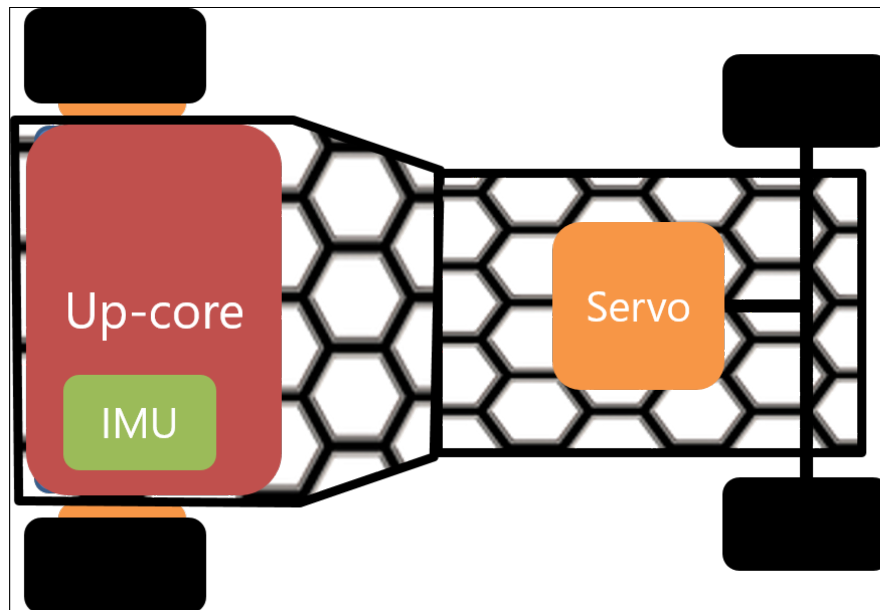


Abbildung 2.7.: Schematische Darstellung der Anordnung der Hauptrecheneinheit bei „Fat Lady“

Abgesehen von diesen gravierenden Unterschieden werden auf beiden Fahrzeugplattformen die gleichen Sensoren zur Umgebungserfassung und Eigenpositionsabschätzung verwendet.

2.4. Testbedingungen und Probleme

Innerhalb des Team NaИ gibt es verschiedene Herausforderungen zu überwinden. Die erste und größte Hürde ist die unterschiedlich starke geographische Distanz der einzelnen Teammitglieder zum Standort der HAW Hamburg. Obwohl die HAW Hamburg, insbesondere das Forschungsprojekt FAUST, Räumlichkeiten zum Arbeiten und sogar ein Teststreckenlabor (vgl. Abbildung 2.8) zur Verfügung stellt, ist es nicht immer allen Teammitgliedern möglich, vor Ort am Fahrzeug zu entwickeln und zu testen.

Eine weitere Hürde sind die baulichen Gegebenheiten im Zusammenspiel mit den Anforderungen an einen regelkonformen Streckenverlauf (Carolo-Cup, 2018, S. 15ff). Je mehr Elemente in das Regelwerk aufgenommen werden (vgl. Kapitel 6), desto schwieriger ist es, alle Elemente während einer Fahrt im Teststreckenlabor testen zu können.

Weiterhin müssen die verschiedenen eingesetzten und in ihrer Leistung sehr unterschiedlichen Hardwarekomponenten innerhalb des Teams berücksichtigt werden. Dies hat zur Folge,

Abbildung 2.8.: Die Teststrecke der HAW Hamburg

dass die Möglichkeit hergestellt werden muss, auf allen gängigen Systemen wie Windows, Linux und auch macOS testen zu können.

Hierfür ist ein erster Schritt die in Abschnitt 2.3.1 beschriebene NaИ-GUI. Durch die NaИ-GUI ist jeder Entwickler mit Hilfe von unzähligen aufgezeichneten Fahrten unabhängig von Zeit, Ort und Fahrzeugplattform in der Lage, Änderungen an Algorithmen zu testen.

Das Hauptproblem mit dieser Art zu testen ist, dass zwar die korrekte Erkennung von Elementen im aufgezeichneten Bild validiert werden kann, aber eine korrekte Reaktion des Fahrzeugs nicht direkt überprüfbar ist. Sollte es während der Erkennung und Planung einer Trajektorie zu Fehlern kommen, so wirken sich diese nicht auf die statischen Folgebilder aus. Der Input bleibt somit, auch wenn eine Trajektorie durch einen Auswertungsfehler einen anderen Weg als den aus den Bildern hervorgehenden Weg definieren würde, immer dieselbe Bildfolge, da die aufgezeichneten Bilder nicht mit der realen Reaktion des Fahrzeugs übereinstimmen können. Hierdurch ergibt sich immer eine Differenz zwischen Test und Realität.

Zur Lösung dieses Problems werden im folgenden Kapitel verschiedene alternative Möglichkeiten zum Testen von Systemen wie „LaK-XU 5000“ oder „Fat Lady“ vorgestellt und eines davon zur Umsetzung ausgewählt.

3. Testvarianten von Embedded-Systemen im Projekt Carolo-Cup

In diesem Abschnitt werden die verschiedenen Möglichkeiten zum Testen von Embedded-Systemen im Carolo-Projekt kurz betrachtet. Hierfür wird zur Orientierung zunächst eine Übersicht über gängige Testverfahren gegeben. Anhand der im Rahmen des Carolo-Projekts gestellten Anforderungen wird dann eines der Testverfahren ausgewählt werden, welches sich für die Zwecke des Projekts am Besten eignet.

3.1. Zweck des Software-Testings

Die Möglichkeit zu beweisen, dass eine Software vollständig korrekt ist, wurde bereits im Jahr 1937 von Alan Turing mit dem Beweis, dass das Halteproblem nicht entscheidbar ist, widerlegt (vgl. Turing, 1937, S. 259-263). Trotzdem wird es in jedem Softwareprojekt als notwendig angesehen, Tests durchzuführen, die die angestrebte Fehlerfreiheit der Software bestätigen.

Die Motivation zu testen erfolgt aus verschiedenen Gründen. Das Hauptziel der meisten Softwaretests ist es, kritische Bugs, welche zu einem Programmabsturz führen, und logische Bugs, welche unerwünschtes Verhalten zur Folge haben, in einer Software bestmöglich auszuschließen. Dies wird in Softwareentwicklungsprojekten angestrebt, um den Wartungsaufwand und die Risiken so gering wie möglich zu halten.

Auch bezogen auf Embedded Systeme und insbesondere das Carolo-Projekt ist ausgiebiges Testen der Soft- und Hardware zwingend notwendig. Auch wenn es sich hier „nur“ um einen Modellmaßstab handelt und Personenschäden somit so gut wie ausgeschlossen sind, so können fehlerhafte Programmabläufe in der Hauptsoftware, der Steuersoftware oder der Hardware zu Schäden am Modell, den Räumlichkeiten oder (in geringem Umfang) an Personen führen.

Übertragen auf die praktische Nutzung autonomer Fahrsysteme zur Personenbeförderung oder Nutzung im Straßenverkehr gilt es, besonders den letzten Punkt bestmöglich zu verhindern oder nach Möglichkeit ganz auszuschließen. Weitere Motivationen zu testen sind laut (Berger, 2008, S. 50) vor allem die Minimierung von Entwicklungs- und Wartungskosten sowie eine Verbesserung der Performance. Im Falle des Carolo-Projektes spielt eine Steigerung der

Performance die größere Rolle, da tatsächliche Entwicklungs- und Wartungskosten wenig Relevanz für ein von Studenten im kleinen Maßstab durchgeführtes Projekt haben. Auch hier muss aber der Zeitaufwand, der durch Weiterentwicklung und Wartung von unzulänglich getesteten Systemen gesteigert wird, selbstverständlich ebenfalls in Betracht gezogen werden.

3.2. Testarten

3.2.1. Hardware in the Loop Testing

Für Hardware in the Loop, im weiteren Text mit HiL abgekürzt, werden Komponenten eines Embedded Systems oder einer technischen Anlage getrennt vom Gesamtsystem getestet. Typischerweise geschieht dies durch herauslösen eines Bauteils aus dem Kontext des Gesamtsystems und modellieren und simulieren des restlichen Systems in Echtzeit.

Als Beispiel wird in der Literatur häufig ein Motorsteuergerät in Fahrzeugen angeführt (vgl. Paulweber und Lebert, 2014, S. 54, S. 59). Das Motorsteuergerät wird beim HiL-Testen losgelöst vom Fahrzeug über die Ein- und Ausgänge des Bauteils an eine Simulationsumgebung angeschlossen. In der Simulationsumgebung werden dann in Echtzeit die Eingaben an das Motorsteuergerät übertragen und mit den erwarteten Ausgaben des Bauteils überprüft. So können Prototyping und Validierung des Motosteuergeräts schneller und kostengünstiger durchgeführt werden, als durch die Verwendung eines echten Motors und Sensoren.

Angewandt auf das Carolo-Projekt könnte für das HiL-Testing Verfahren der in Abschnitt 2.3.2 vorgestellte Mikrocontroller als zu testendes Bauteil betrachtet werden. Hierfür könnte der Mikrocontroller, losgelöst vom Modellfahrzeug, an ein Testsystem angeschlossen werden, auf dem ein simuliertes Modell des Fahrzeugs die benötigten Eingaben erzeugt.

3.2.2. Software in the Loop Testing

Beim Software in the Loop Testing, im Weiteren mit SiL abgekürzt, wird komplett auf die eingesetzten Bauteile verzichtet, sofern das zu testende System ein Bauteil ist. Hier wird nicht nur die Umgebung und Sensorik, sondern auch das zu testende Bauteil durch Software abstrahiert. Die Simulation der Umgebung, der Sensorik und des zu testenden Bauteils kann hierbei auf ein und dem selben Gerät durchgeführt werden (vgl. Thomas Lehmann, 2014, S. 14), da die Anforderungen an die Echtzeitfähigkeit der Simulation beim SiL-Testen weniger streng sind als beim HiL-Verfahren.

Sollen hingegen Algorithmen, Implementierungen und Softwarekompositionen getestet werden, spricht man eher von Funktions-, Unit-, Integrations- oder Regressionstests.

Übertragen auf das Carolo-Projekt könnte für das SiL-Testing die Eingabequelle Kamera abstrahiert werden, sodass die Bilderkennungsalgorithmen auch mit einzelnen Bildern überprüfbar sind. Weitergehend könnte ein System eingesetzt werden, welches zu den Bildern die Ausgaben des zu testenden Systems verarbeitet und angepasste Folgebilder generiert.

3.2.3. Live Testing

Wie der Titel Live Testing schon andeutet wird in diesem Fall das System, welches getestet werden soll, als Ganzes betrachtet und unter möglichst realen Bedingungen überprüft. Der große Vorteil des Live Testings ist, dass das Zusammenspiel aller Komponenten des Systems unter Echtzeitbedingungen geprüft werden kann. Ein Nachteil ist dabei, dass Änderungen an der Code-Basis immer ein Re-Deployment auf das Ziel-System und ein Neustarten des Testlaufs erfordern.

Live Testing ist in den meisten Anwendungsbereichen nicht zu empfehlen, bis die SiL- und HiL-Tests nicht durch positive Ergebnisse ausreichend Sicherheit für erfolgreiche Live Tests geliefert haben.

Die Nachteile eines Live Testing Verfahrens sind neben den oben genannten die Voraussetzungen, die für eine Testumgebung gelten. Für das Carolo-Projekt ist der Zugriff auf das Fahrzeug und eine entsprechend präparierte Testfläche mit ausreichend Platz und einer Fahrbahn unabdingbar. Dies bedeutet, dass entweder jeder aus dem Team ein Testfahrzeug und einen Testraum in der Nähe zur Verfügung haben muss, oder, dass nur in dem von der HAW zur Verfügung gestellten Testlabor getestet werden kann.

3.3. Anforderungen an eine Testumgebung im Rahmen des Carolo-Projektes

Im Carolo-Projekt ist das Live Testing zurzeit die primäre Form des Testens. Dabei können durch die Anpassung live-editierbarer Parameter in den eingesetzten Tools - zum Beispiel über die NaИ-GUI - Abläufe wie die Erkennung der Fahrbahn, die Erkennung von Objekten oder die Fahreigenschaften während des Testlaufs beeinflusst werden, ohne dass ein Neustart erforderlich wird.

Anhand der in Abschnitt 2.4 aufgeführten Testbedingungen, die aktuell im Carolo-Projekt vorhanden sind, ergeben sich die nachfolgend beschriebenen Anforderungen an eine neue Testumgebung.

Unabhängigkeit von der eingesetzten Hardware

Alle Entwickler im Team NaИ sollen unabhängig von den individuell eingesetzten Entwickler-PCs in der Lage sein, die gewählte Testumgebung nutzen zu können. Dies heißt, die Testumgebung soll auf allen gängigen Plattformen, insbesondere macOS, Linux und Windows, ausführbar sein.

Unabhängigkeit von einem Testlabor oder Orten

Alle Entwickler im Team NaИ sollen frei entscheiden können, an welchem Ort die Tests durchgeführt werden. Die Abhängigkeit von dem Testlabor soll durch die virtuelle Testumgebung minimiert werden, sodass zum Beispiel auch kleinere Zeitfenster, in denen sich die Fahrt zum Testlabor nicht lohnen würde, für das Projekt genutzt werden können.

Unabhängigkeit von aufgezeichneten Daten

Die Algorithmen sollen nicht für aufgezeichnete Daten optimiert werden, da Änderungen an den Algorithmen ein verändertes Verhalten des Fahrzeugs nach sich ziehen. Die aufgezeichneten Daten können dies nicht mehr korrekt wiedergeben.

Ein- und ausgehende Kommunikation mit der zu testenden Software über definierte Schnittstellen

Änderungen an den Anwendungen des Carolo-Projekts oder der Testumgebung müssen ermöglicht werden, ohne dass die Testumgebung nicht mehr nutzbar ist. Hierfür sollen fest definierte Schnittstellen geschaffen werden, die unabhängig von den restlichen Modulen die Funktionalität erhalten.

Weiche Echtzeitanforderung

Es ist nur eine weiche Echtzeitanforderung vorgegeben. Während der vollständig simulierten Ausführung ist es nicht relevant, ob einzelne Teile der Berechnung durch die erhöhte Last auf dem System langsamer werden. Der Ablauf soll so bleiben, dass ein empfangenes Bild ausgewertet wird, woraufhin eine Reaktion in Form einer Fahrbewegung ein Folgebild generiert.

Einsetzbarkeit auf einem Gerät zur Minimierung des Testaufwands

Als eine Vereinfachung für die Entwickler im Team NaИ soll die Testumgebung parallel zu den bestehenden Anwendungen auf einem einzelnen Gerät ausgeführt werden können. Dies ermöglicht es den Entwicklern, tatsächlich autark Verbesserungen an den Algorithmen vorzunehmen und die Verbesserungen direkt zu testen.

3.4. Auswahl einer Testmethode

Um die in Abschnitt 3.3 aufgezählten Anforderungen zu erfüllen wird in dieser Arbeit und zwecks Optimierung des Testings im Carolo-Projekt die Variante des Software in the Loop Testings gewählt.

SiL-Testing ist die einzige Möglichkeit den Anforderungen gerecht zu werden. Beim SiL-Testing kann die Fahrzeugplattform in einem simulierten Modell abstrahiert werden, wodurch die Abhängigkeit von der Hardware verringert wird. Eine Abstraktion aller Komponenten, insbesondere des Kamerasensors, ermöglicht es zudem, dynamisch Tests in definierten Szenarien durchzuführen, anstatt aufgezeichnete Daten nutzen und vorhalten zu müssen.

Die Nutzung von SiL-Testing erfüllt zugleich die Anforderung, dass die Testumgebung auf einem Gerät ausgeführt werden kann. Dadurch ist jeder Entwickler in der Lage, Änderungen an Algorithmen und die dadurch entstehenden Verhaltensänderungen des Fahrzeugs im Systemkontext zu evaluieren.

Durch den selbst gesetzten Fokus, vor allem die Bildgebung in der Testumgebung zu simulieren, ergibt sich die Notwendigkeit, eine passende Simulationssoftware auszuwählen und diese über Schnittstellen in die bestehende Systemlandschaft des Carolo-Projekts zu integrieren. Diese Entscheidung bildet die Grundlage der vorliegenden Arbeit und es wird in den folgenden Kapiteln aufgezeigt, wie eine solche Testumgebung unter Zuhilfenahme eines modernen Frameworks zur Umweltgenerierung erstellt werden kann.

4. Überblick über aktuelle Simulations / Game Engines

In diesem Kapitel werden verschiedene aktuelle Simulations / Game Engines mit ihren Vor- und Nachteilen im Hinblick auf die Anforderungen aus Abschnitt 3.3 vorgestellt und eine davon für den Einsatz im Carolo-Projekt ausgewählt.

Zunächst werden hierfür zwei Game Engines, die Unity Game Engine und die Unreal Engine 4, betrachtet. Diese sind in der Spieleindustrie weit verbreitet und sind für ihre detaillierten Renderergebnisse bekannt. Dazu wird eine Cross-Plattform Game Engine vorgestellt: die V-Play Engine, die, wie die im Carolo-Projekt eingesetzte Software, auf das Qt-Framework aufbaut. Abschließend wird das Roboter-Simulationsframework Gazebo beleuchtet, das durch den Fokus auf Robotersimulationen vielversprechende Schnittstellen bereitstellt.

4.1. Unity Game Engine

> „Unity Technologies bietet eine Plattform zur Erstellung fantastischer und packender 2D-, 3D- , VR- und AR-Spiele und Apps.“ (Unity Technologies, 2018a)

Die Unity Game Engine von Unity Technologies, auf die sich das vorangegangene Zitat bezieht, ist eine leistungsstarke Cross-Plattform Engine zur Erstellung von Spielen jeglicher Art.

Unity Engine beinhaltet neben einer leistungstarken Render-Pipeline auch eine Physik Engine, die realistische Simulationen von Objekten ermöglicht (Unity Manual, 2018).

Um Spiele oder Simulationen zu erzeugen liefert Unity Technologies einen Editor. In diesem können die Szenen eines Spiels oder einer Simulation editiert und im Spielemodus direkt getestet werden. Jedes Objekt in einem mit der Unity Game Engine erstellten Spiel ist ein „GameObject“, dass durch verschiedene Komponenten definiert wird. Hierbei handelt es sich um Komponenten wie zum Beispiel Lichtquellen oder auch einfache Formen wie eine Box. Damit die „GameObjects“ eines Spiels auch definierte Aktionen durchführen können, werden diese über die „Unity Scripting API“ angesteuert. Hierbei wird das Verhalten durch in C# geschriebene Methoden definiert.

Systemanforderungen

Alle aufgeführten Anforderungen gelten nur für Desktop-Betriebssysteme (vgl. Unity Technologies, 2018b).

Anforderungen für die Entwicklung sind:

- Windows 7 + oder Mac OS X 10.9+
- CPU mit Support für SSE2 Instruktions Set (siehe (Martin Corden, 2010))
- Grafikkarte mit DirectX 10+
- Rest abhängig von der Komplexität des Projektes

Anforderungen für die Ausführung:

- Windows Vista SP1 +, Mac OS X 10.9+, Ubuntu 12.04+, SteamOS+
- Grafikkarte mit DirectX 10+
- CPU mit Support für SSE-Befehlssatz

Vor- und Nachteile

Die Unity Game Engine bietet eine komplette Spieleentwicklungsumgebung mit einer leistungstarken Render- und Physikpipeline. Neben der sehr ausführlichen Dokumentation und den vielen Beispielen und Tutorials, die den Einstieg in Unity erleichtern, existiert eine große und aktive Community.

Durch die realistischen Rendermöglichkeiten liegt es nahe, diese Engine zur Umweltsimulation für das Carolo-Projekt zu nutzen. Auch die Möglichkeit, mit Unity erstellte Projekte auf unterschiedlichen Plattformen auszuführen, sollte nicht außer Acht gelassen werden.

Jedoch ist die Unity Game Engine darauf ausgelegt, Spiele als Vollbild-Anwendung zu erstellen. Damit eine sinnvolle Simulation der Umwelt im eigenen Framework genutzt werden kann, ist es nicht zwangsläufig vonnöten, dass die Simulation eine Bildschirmausgabe erzeugt, sondern vor allem dass die Daten, die ein Kamera-Sensor erfassen würde, an das eigene Framework weiter gereicht werden.

Um dies mit Unity zu erreichen müsste eine eigene Render-Pipeline implementiert werden, was den Rahmen dieser Arbeit übersteigen würde. Zudem müssten von allen Objekten 3D-Modelle erzeugt werden; eine deskriptive Möglichkeit Objekte zu erstellen gibt es nicht. Darüber hinaus kann nicht gewährleistet werden, dass die Simulation mit der Unity Engine auf allen

im Team NaИ vorhandenen Entwickler-PCs ausgeführt werden könnte (siehe Abschnitt 2.4). Ein weiterer Nachteil ist auch die Einführung der weiteren Programmiersprache C#, die im Carolo-Projekt bisher nicht genutzt wird.

4.2. Unreal Engine 4

Die Unreal Engine 4 von Epic Games ist eine Sammlung von Entwickler-Tools für das Erstellen von hoch realistischen Spielen oder Animationen (vgl. Epic Games, 2018a). Beworben wird die Engine mit photorealistischem Rendering in Echtzeit.

Auch die Unreal Engine 4 wird mit einem eigenen Editor zur Bearbeitung von Szenen ausgeliefert. Zusätzlich bietet die Engine die Möglichkeit, über „Blueprint visual scripting“ interaktiven Inhalt zu erstellen, ohne dass dafür Code geschrieben werden muss.

Neben den herausragenden Möglichkeiten hoch realistische Bilder zu erzeugen, ist die Unreal Engine 4 ausgestattet mit der PhysX 3.3 Engine von Nvidia, die es ermöglicht, exakte physikalische Interaktionen zwischen den einzelnen Komponenten zu simulieren.

Systemanforderungen

Alle aufgeführten Anforderungen für Desktop-Betriebssysteme sind im Folgenden aufgeführt (vgl. Epic Games, 2018b).

Anforderungen für die Entwicklung sind:

- Windows 7 64-bit +, Mac OS X 10.12.6 +, Ubuntu 15.04 +
- Quad Core CPU mit 2.5 GHz oder mehr
- 8 GB RAM
- Dedizierte Grafikkarte
  - DirectX 11 kompatibel (Windows)
  - Nvidia Geforce 470 GTX oder besser (Linux)
  - Metal 1.2 kompatibel (Mac OS X)
- Zusätzliche Software
  - Visual Studio 2015 Professional / Community (Windows)
  - Xcode (Mac OS X)
  - Kernel 2.6.32 + (Linux)

  - glibc 2.12.2 + (Linux)
  - clang 3.5.x + (Linux)

Die Anforderungen für die Ausführung sind für die Hardware dieselben, für die Software muss jeweils nur die Runtime vorhanden sein.

Vor- und Nachteile

Die Unreal Engine 4 ist eine sehr leistungsstarke Engine für die Entwicklung von physikalisch und grafisch sehr aufwändigen und hochwertigen Spielen, Animationen oder Simulationen.

Für den Anspruch einer möglichst detailgetreuen Simulation der Umwelt scheint die Unreal Engine 4 bestens geeignet. Dagegen spricht hingegen, dass auch in diesem Fall eine eigene Render-Pipeline für den Zugriff auf die Bilddaten erstellt werden müsste. Dies und die durchaus sehr anspruchsvollen Hardware-Anforderungen, selbst bei der Bearbeitung eines Projektes, sprechen gegen den Einsatz dieser Engine. Wie bereits zuvor beschrieben sind im Team NaИ die unterschiedlichsten Hardware-Konfigurationen vorhanden, worauf bei der Auswahl eines Frameworks zur Simulation Rücksicht genommen werden muss.

Positiv ins Gewicht fällt dafür, dass für Unreal Engine 4 Projekte keine weitere Programmiersprache die Komplexität erhöht, sondern mit C++ entwickelt werden kann.

4.3. V-Play Engine

Die V-Play Engine ist eine Erweiterung des Qt-Frameworks. In Kombination mit der Qt-Meta-Language (kurz QML), einer deklarativen Sprache, die das Qt-Framework erweitert, können in relativ kurzer Zeit Cross-Plattform-Anwendungen erstellt werden.

Zusätzlich zu den Kernfunktionen von Qt stellt die V-Play Engine Komponenten für die Visualisierung, physikalische Berechnungen oder auch die Monetarisierung einer Anwendung zur Verfügung. Der Fokus der V-Play Engine liegt hierbei auf „2D Casual Games“ für mobile Geräte, wie z.B. „Tower Defense“-Spiele oder „Plattformer“-Spiele im klassichen Jump’n’Run Aufbau.

Systemanforderungen

Die Anforderungen für Desktop-Betriebssysteme sind im Folgenden aufgeführt (vgl. V-Play GmbH, 2018b).

Anforderungen für die Entwicklung sind:

- Xcode (Mac OS X)

- Build-Essentials, g++, GLU Bibliotheken für die Grafik, Pulse Bibliotheken für Audio (Linux)
- Qt in aktueller Version
- Für Windows existieren keine speziellen Anforderungen

Anforderungen für die Ausführung:
Es sind keine speziellen Anforderungen aufgeführt. Da der Fokus auf „2D Casual Games“ für mobile Geräte liegt, ist davon auszugehen, dass die Hardwareanforderungen für die Ausführung von jedem Desktopsystem erfüllt werden können. Die Softwareabhängigkeiten werden durch das Deployment der Anwendung abgedeckt.

Vor- und Nachteile

Die V-Play Game Engine ist durch QML ein einfach zu bedienendes Framework, das die Funktionen des im Projekt eingesetzten Qt-Frameworks noch erweitert. Die Anbindung an das eigene Projekt kann dadurch mit nur kurzem Zeitaufwand realisiert werden. Die geringen Hardware- und Softwareanforderungen sprechen eindeutig für diese Engine. Zudem wird für den Rest des Projektes bereits das Qt-Framework eingesetzt und somit ist die nötige Entwicklungsumgebung bereits bei allen Entwicklern vorhanden.

Bei näherer Betrachtung fällt aber auf, dass der Fokus auf „2D Casual Games“ erhebliche Abstriche in der Qualität der simulierten Umwelt bedeuten würden. Sobald man die Referenzprojekte genauer ansieht, fällt eine comicartige Grafik auf, die darauf schließen lässt, dass eine zufriedenstellende Simulation der abstrakten Umwelt der Carolo-Cup Szenarien nicht möglich ist (vgl. V-Play GmbH, 2018a).

4.4. Gazebo

Gazebo ist ein Open Source 3D-Simulator der, obwohl auf ähnliche Weise wie eine Game Engine aufgebaut, in erster Linie für die Simulation von Robotern in verschiedenen, komplexen Szenarien erstellt wurde. Dies macht sich sowohl durch einen höheren Genauigkeitsgrad in der Berechnung der Physik und ein breites Spektrum an definierten Sensoren als auch durch diverse Schnittstellen für Benutzer und Anwendungen bemerkbar (vgl. Open Source Robotics Foundation, 2018a).

Der Gazebo-Simulator bietet die Möglichkeit, eine von vier verschiedenen Physik-Engines zu nutzen - ODE (Open Dynamics Engine), bullet, Simbody oder DART (Dynamic Animation and Robotics Toolkit). In der Standardkonfiguration wird die ODE Physik-Engine verwendet.

Zusätzlich dazu können Modelle über individuelle Plugins direkt auf die Gazebo-API zugreifen und dadurch im Funktionsumfang erweitert werden.

Systemanforderungen

Im Folgenden werden die Anforderungen für Desktop-Betriebssysteme aufgeführt (vgl. Open Source Robotics Foundation, 2018a,b).

Anforderungen für die Entwicklung sind:

- Mindestens 500 MB freier Speicherplatz
- Mindestens Intel I5 CPU oder vergleichbar
- Dedizierte Grafikkarte
- Ubuntu 14.04 +, Mac OS X 10.11 + oder Windows 7 +

Empfohlen wird für die Entwicklung das Ubuntu Betriebssystem.

Eine Angabe der Anforderungen für die Ausführung des Gazebo-Simulators gibt es nicht.

Vor- und Nachteile

Der Gazebo-Simulator vereint verschiedene Vorteile. Neben den bereits erwähnten genaueren physikalischen Berechnungen als in den meisten Game Engines ist auch die leistungsstarke Render Engine OGRE ein Vorteil des Simulators (vgl. OGRE, 2018). Ein weiterer Vorteil ist die Bereitstellung von verschiedenen konfigurierbaren Sensortypen, sodass diese nicht komplett selbst umgesetzt werden müssten. Durch die Verfügbarkeit eines Kamera Sensortyps entfällt für das Gazebo-Framework die Notwendigkeit, eine eigene Render-Pipeline zu erstellen. Stattdessen kann direkt auf die Bilddaten zugegriffen werden. Dies ist ein sehr großer Vorteil des Gazebo-Frameworks.

Dass simulierte Modelle mit in C++ geschriebenen Plugins in ihrer Funktionalität erweitert werden können ist ein zusätzlicher Vorteil. Neben diesen Punkten bietet der Gazebo-Simulator viele Anwendungsbeispiele, von einfachen Einstiegsmodellen bis hin zu komplexen Modellen wie dem Pioneer 3-AT, einer Roboterforschungsplattform der Firma Omron (vgl. Omron Adept, 2016). Dies ermöglicht auch Entwicklern, die das Framework noch nicht kennen, einen schnellen Einstieg.

Nachteilig am Gazebo-Simulator ist die eingeschränkte Möglichkeit im Gazebo-Client Modelle zu bearbeiten. Jedoch fällt dies nur gering ins Gewicht, da Modelle und Umgebungen sehr genau mithilfe von SDF (Simulation Description Format), einem XML Format zur Beschreibung

von Objekten und Umgebungen speziell für den Einsatz in der Robotersimulation, definiert werden können (vgl. Open Source Robotics Foundation, 2014d).

4.5. Auswahl eines Frameworks

Anhand der in diesem Kapitel vorgestellten Vor- und Nachteile der ausgewählten Simulations / Game Engines wird für den weiteren Verlauf dieser Arbeit das Roboter-Simulationsframework Gazebo ausgewählt. Die vom Gazebo-Framework angebotenen Schnittstellen sowie die Integration von Sensoren unterstützen die Annahme, dass das Gazebo-Framework die in Abschnitt 3.3 vorgestellten Anforderungen am besten erfüllen kann.

Durch die im Vergleich zu den Game Engines geringeren Hardwareanforderungen von Gazebo wird angenommen, dass die Anforderung, die Testumgebung auf allen im Carolo-Projekt eingesetzten Entwickler-PCs einsetzen zu können, erfüllt wird. Hierdurch liegt es nahe, dass die Anforderungen an eine orts- und fahrzeugunabhängige Testumgebung durch den Einsatz auf allen Entwickler-PCs umgesetzt werden kann.

Die Möglichkeit, direkt auf die Sensordaten und somit auf die simulierten Bilddaten zugreifen zu können, spielt eine weitere positive Rolle in der Entscheidung für das Gazebo-Framework. Weiterhin ist anzunehmen, dass durch die einfachen Erweiterungsmöglichkeiten mithilfe von Plugins in der bereits im Carolo-Projekt eingesetzten Programmiersprache C++ die benötigte Funktionalität umsetzbar ist.

Der Einsatz des Roboter-Simulationsframeworks Gazebo ist aus den vorgenannten Gründen das einzige Framework, das zur Simulation der Carolo-Umwelt in Frage kommt. Bei den anderen vorgestellten Game Engines sind entweder die Hardwareanforderungen zu hoch (Unity Game Engine und Unreal Engine 4) oder die Ergebnisse der Simulation können nicht genau genug erwartet werden (V-Play Engine). Das Gazebo-Framework bildet hier den benötigten Kompromiss aus Hardwareanforderungen und Genauigkeit, die für einen flexiblen Einsatz im Carolo-Projekt sprechen.

5. Eingesetzte Frameworks

In diesem Kapitel sollen die Frameworks, die für den weiteren Verlauf dieser Arbeit eingesetzt werden, detaillierter beschrieben werden. Hierbei wird zunächst das C++-Framework Qt, welches die Basis für die Carolo-Projekt-Anwendungen im Team NaИ ist, vorgestellt. Anschließend wird das in Abschnitt 4.4 kurz vorgestellte Roboter-Simulationsframework Gazebo näher betrachtet.

5.1. Qt

Für die Software, die im Carolo-Projekt entwickelt wird, bildet das C++-Framework Qt die Grundlage. Das Framework der Firma „The Qt Company“ ermöglicht es auf simple Weise, für verschiedene Plattformen wie macOS, Unix oder Windows Anwendungen mit nativer C++-Performance zu erstellen, ohne dass dafür die Code-Basis angepasst werden muss. Darüber hinaus stellt es einfach zu verwendende Methoden und Module zur Verfügung, welche die regulären Möglichkeiten von C++ erweitern oder vereinfachen.

5.1.1. Funktionsweise

Hierfür verwendet Qt einen Quellcode-Generator namens MOC (Meta Object Compiler), der Standard-C++ um weitere Elemente wie zum Beispiel Meta-Objekt-Daten, die zur Introspektion genutzt werden können, ergänzt.

Qts Meta-Objekt-System

Das Meta-Objekt-System ist die Zentrale Komponente des Qt-Frameworks. Es stellt die Grundlagen für den Signale und Slots-Mechanismus sowie für die Introspektion zur Verfügung.

Introspektion bedeutet, dass einem Programm seine eigene Struktur bekannt ist, und dass das Programm in der Lage ist, die Methoden und Eigenschaften - wie etwa Datentypen von Methodensignaturen - eines Objektes zu benennen und zu modifizieren.

Der Quellcode-Generator MOC, der C++-Klassen nach Qt-Makros wie zum Beispiel `Q_OBJECT`, `signals`, `slots`, oder `emit` durchsucht, erzeugt Quellcode, der die für die Introspektion

nötigen Meta-Objekt-Daten implementiert. Hiermit erzeugter Quellcode ist C++-Standard konform und kann mit allen handelsüblichen Compilern übersetzt werden. Unter der Voraussetzung einer korrekt eingerichteten Cross-Compiling-Toolchain mit einem Compiler für die Zielplattform versetzt dies den Entwickler in die Lage, die gleiche Code-Basis zu benutzen um Anwendungen für alle Plattformen zu kompilieren.

Eingesetzt wird dieser Vorteil im Carolo-Projekt um die Anwendung für die Testplattform „LaK-XU 5000" (vgl. Abschnitt 2.3.2) zu übersetzen. Dies ist nötig, da die auf dem Fahrzeug eingesetzte Hauptrecheneinheit - im Gegensatz zu den Entwickler-PCs - ARM-basiert ist und der Byte-Code für die Ausführung auf dieser Plattform anders zusammengesetzt werden muss.

Durch das Meta-Objekt-System von Qt, welches durch die Introspektion ermöglicht wird, werden zudem die Signale und Slots-Mechanismen, die Laufzeit-Typ-Informationen von Objekten (run-time type information RTTI) und das dynamische Eigenschaftensystem bereitgestellt.

Signale und Slots

Der Signale und Slots-Mechanismus von Qt ermöglicht eine Kommunikation zwischen Objekten. Der große Vorteil des Signale und Slots-Mechanismus ist, dass die einzelnen Objekte nicht zwangsläufig untereinander bekannt sein müssen. Ein Signal wird von einem Objekt ausgesendet (Qt-Makro „emit"), wenn der Objektzustand auf eine Weise verändert wurde, die für andere Objekte von Interesse sein könnte. Ob ein Signal von einem anderen Objekt empfangen wird ist hierfür nicht relevant.

Slots hingegen können in Objekten zum Empfangen von Signalen verwendet werden. Ein Slot ist in erster Linie eine normale Member-Funktion eines Objekts, die mit dem Qt-Makro „slot" gekennzeichnet wurde. Auch hier ist zunächst nicht relevant, ob ein verbundenes Signal für diesen Slot existiert, da die als Slot definierten Member-Funktionen, wie andere Member-Funktionen auch, direkt aufgerufen werden können.

Durch den Signale und Slots-Mechanismus ist eine echte Informationskapselung und die Erstellung von unabhängigen Komponenten möglich. Gleichzeitig können die verschiedenen Komponenten lose gekoppelt werden, indem eine Verbindung zwischen einem Signal eines Objekts und einem Slot eines anderen Objekts aufgebaut wird. Das Meta-Objekt-System von Qt sorgt dann dafür, dass sobald ein Signal ausgesendet wird der verbundene Slot - beziehungsweise die dahinter liegende Member-Funktion - aufgerufen wird.

Bei der Verbindung spielt es keine Rolle, welche Sichtbarkeit für die Member-Funktion definiert ist. So kann es sein, dass durch ein Signal eines beliebigen Objekts ein privater Slot eines nicht verwandten Objekts aufgerufen wird, sofern das Objekt mit dem privaten Slot das Signal verbunden hat.

Dieses Prinzip erlaubt eine Modul-Kommunikation innerhalb der Carolo-Projekt Anwendungen bei gleichzeitig loser Kopplung der einzelnen Module, welche die in Abschnitt 2.3.1 aufgeführten Schichten ausmachen. Neben den Cross-Plattform-Entwicklungsmöglichkeiten ist dies der Hauptgrund für den Einsatz von Qt im Carolo-Projekt.

Der Signale und Slots-Mechanismus ermöglicht es zudem, die Anwendungen zeit- und eventgesteuert ablaufen zu lassen. Das heißt, dass einzelne Threads der Anwendung in regelmäßigen Abständen ausgeführt werden müssen, zum Beispiel die serielle Kommunikation mit dem Mikroprozessor oder das Auswerten der Daten von der Kamera. Andere Threads der Anwendung hingegen müssen nur bei einer Zustandsänderung aufgerufen werden.

Modularität

Weiterhin ist das Qt-Framework in verschiedene Module unterteilt. Dies ermöglicht es, in einer Anwendung nur die Module zu nutzen, die tatsächlich gebraucht werden. Dadurch sind kleinere und schnellere Programme möglich, die auch auf Embedded-Hardware performant ausführbar sind.

Innerhalb der Modularisierung wird zwischen Modulen der Kategorie „Qt Essentials“ und der Kategorie „Qt Add-Ons“ unterschieden. Hierbei sind in den „Qt Essentials“-Modulen die Grundlagen von Qt für alle Plattformen enthalten. Beispiele für Grundlagenmodule sind „Qt Core“, welches die Grundlage für jede Qt-Anwendung ist und das nötige Qt-Meta-Objekt-System bereitstellt, sowie „Qt GUI“, welches Basisklassen für grafische Benutzeroberflächen beinhaltet. Neben diesen Modulen existieren noch weitere Grundmodule, die eine Abstraktionsschicht für plattformspezifische Implementierungen wie beispielsweise Netzwerkfunktionalitäten („Qt Network“) enthalten und somit zur Vereinfachung der Cross-Plattform-Entwicklung beitragen.

Dahingegen stellen „Qt Add-Ons“-Module Klassen und Methoden für spezielle Anwendungsgebiete mit zum Teil plattformspezifischer Bindung bereit. Hierunter fallen Module wie „Qt Serialport“, welches Zugriff auf serielle Schnittstellen ermöglicht, oder „Qt WebSockets“, welches die Kommunikation über WebSockets vereinfacht.

Eine vollständige Liste der in Qt verfügbaren Module kann in der offiziellen Dokumentation eingesehen werden (The Qt Company, 2018).

5.1.2. Eingesetzte Module

Im Carolo-Projekt werden von den oben vorgestellten Modulen die im Folgenden aufgeführten genutzt.

Qt Core

Dieses Modul wird aufgrund des dadurch bereitgestellten Meta-Objekt-Systems und der Möglichkeit die Signale und Slots-Mechanismen zu nutzen eingesetzt.

Qt GUI

Dieses Modul wird für die NaИ-GUI eingesetzt und wird ausschließlich auf den Entwickler-PCs verwendet, da die Anwendung auf dem Fahrzeug keinerlei GUI-Elemente benötigt.

Qt Widgets

Dieses Modul stellt weitere Klassen und Methoden bereit, die den Funktionsumfang des „Qt GUI"-Moduls erweitern. Es wird ebenfalls nur auf den Entwickler-PCs eingesetzt.

Qt Network

Dieses Modul wird zur Kommunikation zwischen den Fahrzeugplattformen und der NaИ-GUI auf den Entwickler-PCs genutzt.

Qt Serialport

Dieses Modul wird für die Kommunikation zwischen der Carolo-Anwendung auf der Hauptrecheneinheit und dem Mikroprozessor eingesetzt.

Qt WebSockets und Qt WebChannel

Diese Module werden eingesetzt, um eine Schnittstelle für webbasierte Anwendungen zur Verfügung zu stellen. Die Schnittstelle wird derzeit nicht genutzt, ist aber in der Software weiterhin enthalten.

Version

Die aktuelle Version von Qt ist 5.10. Die im Carolo-Projekt verwendete Version von Qt und den Modulen ist 5.4. Dies liegt daran, dass für eine korrekte Cross-Compiling-Toolchain auch das Qt-Framework für die Zielplattform übersetzt werden muss. Da dies ein langwieriger Vorgang ist und in den neueren Versionen keine zwingend benötigten Funktionalitäten oder zwingend benötigte Module hinzugekommen sind, wurde auf ein Upgrade auf die aktuelle Version verzichtet.

5.2. Gazebo

Wie in Abschnitt 4.5 bereits begründet, wird für den weiteren Verlauf dieser Arbeit das Roboter-Simulationsframework Gazebo verwendet. Begonnen hat die Entwicklung von Gazebo im Herbst 2002. Dies geschah aufgrund der damaligen Notwendigkeit von Dr. Andrew Howard und seinem Studenten Nate Koenig von der University of Southern California, einen Simulator nutzen zu können, der Roboter in unterschiedlichen Außenumgebungen und unter verschiedenen Bedingungen simulieren kann. Im Jahr 2009 wurde das Robot Operating System (ROS) in das Gazebo-Projekt integriert und ist seitdem eines der meistgenutzten Tools in der ROS Community. 2012 wurde die Verwaltung des Open-Source-Projekts Gazebo von der Open Source Robotics Foundation (OSRF) übernommen und wird seitdem von OSRF und einer aktiven Community weiterentwickelt (vgl. Open Source Robotics Foundation, 2014a).

In den folgenden Abschnitten werden die Funktionsweise und der Aufbau von Gazebo detaillierter vorgestellt.

5.2.1. Architektur

Der Robotersimulator Gazebo verwendet eine verteilte Architektur. Hierbei werden unterschiedliche Bibliotheken für die physikalische Simulation, das Rendering oder die Erzeugung von Sensordaten verwendet.

Zudem ist Gazebo mit einer Client-Server-Architektur in zwei verschiedene ausführbare Programme unterteilt, die unterschiedliche Aufgaben realisieren. Das erste Programm ist der Server „gzserver“ und ist dazu da, die Physik zu simulieren sowie das Rendering und die Sensordaten zu berechnen. Das zweite Programm ist der Client „gzclient“, der die Simulation für den Benutzer visualisiert und eine Interaktion mit der Simulation ermöglicht.

Die beiden Programme kommunizieren im Publish/Subscribe-Pattern durch „Google Protobuf“-Nachrichten über die „boost::ASIO“-Bibliothek. Ein Beispiel für solch eine Nachricht ist ein Positionsupdate für ein Objekt innerhalb der Simulation. Diese Methode erlaubt es, eine laufende Simulation zu inspizieren und von außen zu kontrollieren (vgl. Open Source Robotics Foundation, 2014a).

Auf dieser Methode basiert auch die Anbindung an die Anwendungen im Rahmen des Carolo-Projekts, welche in Kapitel 8 ausführlicher beschrieben wird.

5.2.2. Komponenten

Für die Ausführung einer Gazebo-Simulation sind verschiedene Komponenten erforderlich. Nachfolgend sind die einzelnen an einer Simulation beteiligten Komponenten aufgeführt (vgl. Open Source Robotics Foundation, 2014b).

World Files

Hiermit wird die zu simulierende Welt mit allen ihren Elementen definiert. Dies beinhaltet die Definitionen von Robotern, Beleuchtung, Sensoren oder anderen statischen Elementen. Diese Datei, die mit der Dateiendung `*.world` angelegt werden sollte, wird von dem Gazebo-Server eingelesen, der daraufhin die simulierte Welt generiert.

Ein (leeres) Beispiel für ein World File ist in Listing 5.2 aufgeführt.

Model Files

Eine Modell-Datei beschreibt, im gleichen Format wie ein World File, ein einzelnes Element der Welt, beispielsweise ein Fahrzeug, welches „Modell“ genannt wird. Dies ist sinnvoll, damit einzelne Modelle gekapselt und wiederverwendet werden können. Zusätzlich hilft es dabei, die World Files übersichtlicher zu gestalten, da die gekapselten Modelle mit

```
<include>
    <uri>model://model_file_name</uri>
</include>
```

Listing 5.1: Include Befehl von SDF

in ein World File integriert werden können.

Eine Modell-Datei besteht wiederum aus verschiedenen Elementen. Das erste Hauptelement ist ein Link (`<link>`), hiervon können mehrere in einem Modell existieren. Ein Link ist ein Teil eines Modells; so kann ein Link zum Beispiel ein Rad sein, wenn das Modell ein Fahrzeug repräsentiert.

Der Link wird wiederum durch bestimmte Attribute beschrieben, die unter anderem die physikalischen Eigenschaften des Links definieren. Zunächst muss über ein oder mehrere Kollisionselemente (`<collision>`) die geometrische Größe des Elements zur Kollisionsberechnung mit anderen Elementen definiert werden. Optional kann zu einem Link ein visuelles Element definiert werden (`<visual>`). Dies ist die sichtbare Repräsentation des Modellteils und kann durchaus von der geometrischen Repräsentation abweichen. Genutzt wird die Diskrepanz zwischen visueller und geometrischer Repräsentation zur Vereinfachung des

Renderings oder der physikalischen Berechnungen, je nachdem, welche Komponente einfacher gestaltet wird. Über das Trägheits-Element (`<inertial>`) können die Eigenschaften der Masse definiert werden. Weiterhin kann ein Link ein Sensor-Element (`<sensor>`) enthalten, welches Daten aus der simulierten Umgebung sammeln und bereitstellen kann.

Das zweite, optionale Hauptelement ist ein Gelenk (`<joint>`). Auch hiervon können mehrere in einem Modell existieren. Ein Gelenk verbindet zwei Links in einer Eltern-Kind Beziehung miteinander. Hierfür muss der Gelenk-Typ, zum Beispiel ein Drehgelenk (`<joint type="revolute">`) mit einer definierten Rotationsachse, angegeben werden.

Das dritte, optionale Hauptelement ist die Angabe, ob und welches Plugin (`<plugin>`) ein Modell verwendet. Ein Plugin ist eine gemeinsam genutzte Bibliothek, um zum Beispiel ein Modell zu steuern.

Innerhalb eines Modells kann durch Verschachtelung ein weiteres Modell definiert werden. Der elegantere Weg wäre jedoch, dass das verschachtelte Modell separat definiert und über ein `<include>` hinzugefügt wird, um die Kapselung der einzelnen Modelle zu erhalten.

Ein (leeres) Beispiel für ein Model File ist in Listing 5.3 aufgeführt. Viele weitere Beispiele, die einen guten Einstieg in Gazebo ermöglichen, können im offiziellen Modell-Repository von Gazebo eingesehen werden (Open Source Robotics Foundation, 2018c, Repository).

Gazebo-Server

In dieser Komponente geschieht die eigentliche Arbeit von Gazebo. Der Gazebo-Server liest das World File ein und simuliert unter Verwendung der Physik-Engine die darin enthaltenen Elemente. Der Gazebo-Server kann einzeln gestartet werden, wenn keine Visualisierung der Simulation benötigt wird. Gestartet wird der Gazebo-Server über den Konsolen-Befehl `gzserver <world_filename>`.

Gazebo-Client

Diese Komponente dient der Visualisierung und Modifizierung einer im Gazebo-Server laufenden Simulation. Damit der Gazebo-Client gestartet werden kann, muss eine Instanz eines Gazebo-Servers bereits gestartet sein. Dies kann, nach dem Starten einer Gazebo-Server Instanz, über den Konsolen-Befehl `gzclient` getan werden. Zur Vereinfachung hierfür gibt es zusätzlich die Möglichkeit über den Konsolen-Befehl `gazebo <world_filename>` einen Server und einen Client gemeinsam zu starten.

Plugins

Durch Plugins können zur Laufzeit verschiedene Funktionalitäten für eine Simulation ergänzt werden. Zum Einen bieten Plugins die Möglichkeit Modelle zu steuern, zum Anderen kann über Plugins eine Kommunikation mit Gazebo hergestellt werden, die weitere Modifikationen an einer laufenden Simulation ermöglichen.

Beispiele

Hier sind zwei Beispiele für die zuvor angeführten Komponenten World Files und Model Files angeführt.

```
<?xml version="1.0" ?>
<sdf version="1.5">
<world name="default">
    <physics type="ode">
    ...
    </physics>
    <scene>
    ...
    </scene>
    <model name="box">
    ...
    </model>
    <light name="spotlight">
    ...
    </light>
</world>
</sdf>
```

Listing 5.2: Beispiel World File (vgl. Open Source Robotics Foundation (2018d))

```
<?xml version="1.0" ?>
<sdf version="1.5">
<model name="box">
    <pose>0 0 0.5 0 0 0</pose>
    <static>false</static>
    <link name="link">
        ...
    </link>
    <joint type="revolute" name="my_joint">
```

```
10          ...
11      </joint>
12      <plugin filename="libMyPlugin.so" name="my_plugin"/>
13  </model>
14  </sdf>
```

Listing 5.3: Beispiel Model File (vgl. Open Source Robotics Foundation (2018e))

5.2.3. Eingesetzte Komponenten

Für die Entwicklung der Simulation der abstrakten Carolo-Umwelt werden alle vorgenannten Komponenten von Gazebo eingesetzt. So sind die einzelnen Umweltelemente in separate Modelle gegliedert und werden über das World-File `nan-world.world` zu einem Szenario komponiert. Wie so ein Szenario zusammengesetzt sein kann wird im Folgenden Kapitel 6 beschrieben. Während der Entwicklung der Simulation hat es sich zudem als vorteilhaft erwiesen, die definierten Modelle im Gazebo-Client zu visualisieren.

Während der eigentlichen Simulation hingegen ist die Möglichkeit die Gazebo-Simulation „headless", also nur den Gazebo-Server ohne den Gazebo-Client, starten zu können sehr nützlich (vgl. Abschnitt 8.2). Hierdurch kann die Rechenlast auf dem ausführenden Entwickler-PC weiter reduziert werden, da nicht noch eine, für die Abläufe der Simulation unnötige, Visualisierung der Carolo-World und den darin enthaltenen Modellen berechnet werden muss.

Die verwendete Version von Gazebo ist die 8.4.0. Dies war zu Beginn dieser Arbeit die aktuelle Version des Gazebo-Frameworks. Seit Januar 2018 steht die aktuelle Version 9.0 zur Verfügung.

6. Umweltszenarien im Carolo-Cup

Der studentische Wettbewerb Carolo-Cup ist in zwei Leistungsklassen unterteilt. Die Klassen „Basis-Cup“ - ehemals „Basis Leistungsklasse“ - und Carolo-Cup - ehemals „Erweiterte Leistungsklasse“ - unterscheiden sich unter anderem darin, dass für den „Basis-Cup“ die in Abschnitt 6.3.2 beschriebene suburbane Situation nicht vorgesehen ist oder zum Beispiel die Anzahl von Parkversuchen abweicht.

Da das Team-NaИ in der Vergangenheit in der Leistungsklasse Carolo-Cup angetreten ist, sind die im Folgenden beschrieben Umweltszenarien als geltend für die höhere Leistungsklasse zu verstehen.

Für alle Umweltszenarien gilt es, innerhalb einer limitierten Zeit so viel Fahrstrecke wie möglich zurückzulegen. Dabei wird ein Fahrfehler des Fahrzeugs mit Strafmetern belegt. Diese werden von der insgesamt gefahrenen Strecke abgezogen. Das Team, welches nach Abzug aller Strafen die längste Strecke gefahren ist, erhält die volle Punktzahl. Alle anderen Teams werden anteilig zu dem besten Ergebnis bewertet.

Für den Carolo-Cup sind zwei unterschiedlich komplexe Umweltszenarien, beziehungsweise Fahraufgaben, vorgesehen (vgl. Carolo-Cup, 2018, S. 15). Zunächst wird das generelle Umweltszenario des Carolo-Cup definiert und anschließend werden die zwei unterschiedlichen Fahraufgaben vorgestellt.

6.1. Genereller Aufbau der Umwelt des Carolo-Cup

Für den Carolo-Cup ist ein abstraktes Landstraßenszenario vorgegeben. Die Modellierung der Landstraße besteht aus einer Fahrbahn mit je einer Fahrspur pro Richtung. Diese Landstraße besteht aus längeren geraden Abschnitten, aber auch engen Kurven und Kreuzungen, inklusive angedeuteten seitlich abgehenden T-Kreuzungen.

Der Beginn der Strecke ist durch eine schwarz-weiß karierte Startlinie von circa 50 mm Breite gekennzeichnet. In einem maximal 20 m langen Bereich direkt hinter der Startlinie existieren links und rechts neben der Fahrspur markierte Parklücken (vgl. Abbildung 6.1); der Parkbereich wird zusätzlich zu der Startlinie durch das zugehörige Straßenverkehrsschild angezeigt. Dieser

Parkbereich ermöglicht es, entweder ein Parkmanöver in Längsaufstellung (rechts, parallel zur Fahrbahn) oder ein Parkmanöver in Senkrechtaufstellung (links, orthogonal zur Fahrbahn) durchzuführen. Dabei ist zu beachten, dass eine durch ein weißes Kreuz markierte oder durch Belegung blockierte Parklücke nicht verwendet werden darf. Eine Belegung ist gegeben, wenn ein anderes Fahrzeug bereits in der Parklücke steht. Andere Fahrzeuge auf der Strecke sind beim Carolo-Cup durch weiße Kartons modelliert (vgl. Abbildung 6.1).

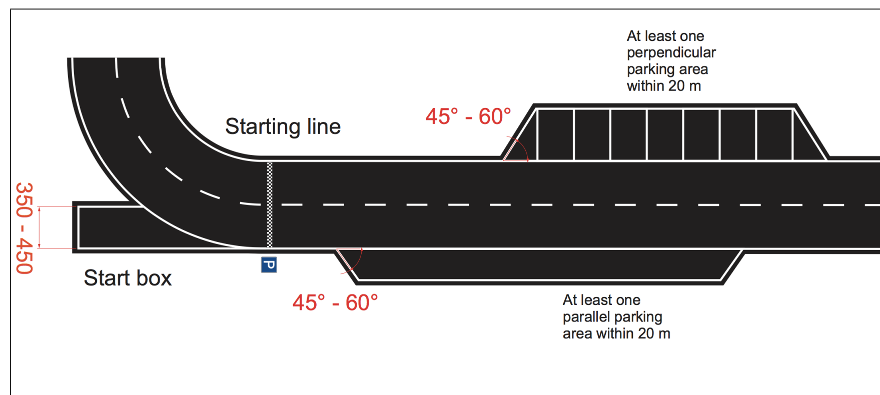


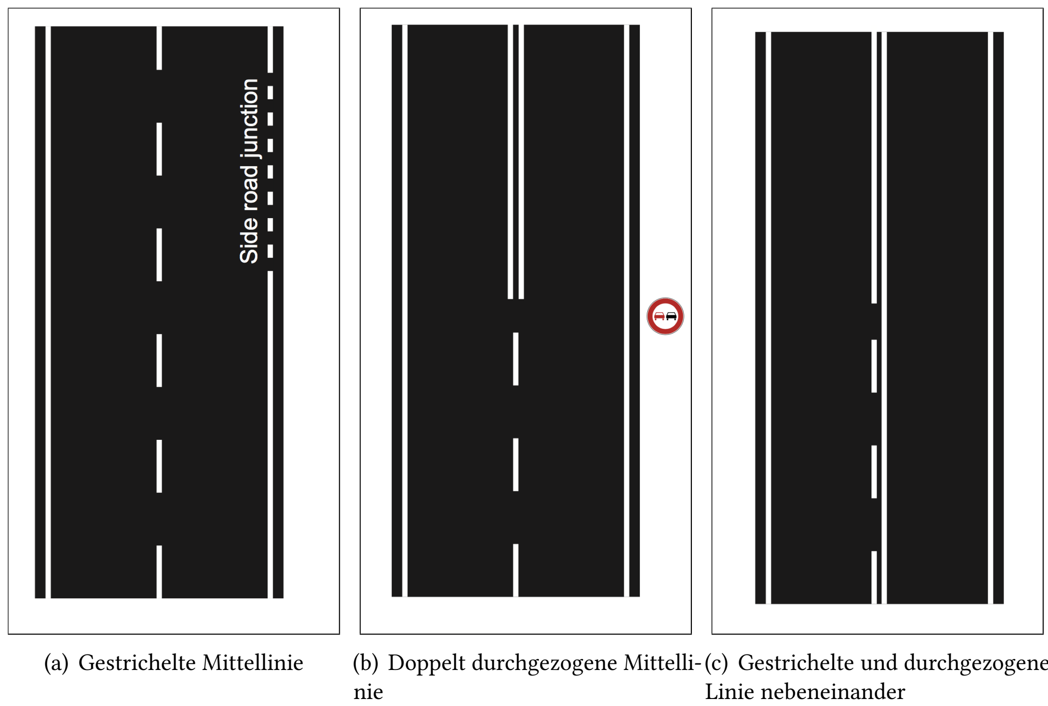
Abbildung 6.1.: Definition der Parksituation

Gestaltet werden die Fahrbahnen und die Parkbereiche durch weiße Linienmarkierungen auf schwarzem Untergrund. Typischerweise sind diese Markierungen, sofern nicht anders angegeben, zwischen 18 mm und 20 mm breit (vgl. Carolo-Cup, 2018, S. 16).

Die Dimensionen einer Fahrspur können zwischen 350 mm und 450 mm Breite, gemessen von der Innenseite der Markierungen, variieren. Beide Seiten der Fahrbahn haben aber stets die gleiche Breite.

Getrennt werden die Fahrspuren durch eine regelmäßig gestrichelte weiße Linie, mit einem Wechsel alle 200 mm. Unterbrochen wird dieses Muster nur von einer Kreuzung oder der Startlinie. Zusätzlich kann - nur im Carolo-Cup, nicht im „Basis-Cup" - statt der einfach gestrichelten Linienmarkierung eine doppelt durchgezogene Linienmarkierung oder eine Kombination aus gestrichelter und durchgezogener Linienmarkierung ein Überholverbot anzeigen (vgl. Abbildung 6.2). Hierbei bleibt eine Markierungsart für mindestens 1000 mm erhalten.

Fahrbahnen können in unmittelbarer Nachbarschaft liegen, jedoch ist ein Mindestabstand von 50 mm zwischen den Außenkanten der Linienmarkierungen beider Fahrbahnen gegeben. Die Strecke ist zum Großteil als eben anzunehmen, jedoch können Abschnitte mit einer Steigung von bis zu 10 % vorkommen. Dies wird durch das entsprechende Straßenverkehrsschild



(a) Gestrichelte Mittellinie

(b) Doppelt durchgezogene Mittellinie

(c) Gestrichelte und durchgezogene Linie nebeneinander

Abbildung 6.2.: Die verschiedenen Mittellinienmarkierungen des Carolo-Cup

angekündigt. Für Kurven in der Ebene gilt ein minimaler Radius von 1000 mm, in Abschnitten mit einer Steigung gilt der minimale Kurvenradius von 10 m. Nach einer mindestens 3 m langen geraden Strecke können Kurven mit dem minimalen Radius durch Richtungstafeln mit einem roten Pfeil auf weißem Grund in die entsprechende Richtung ausgewiesen sein.

Zusätzlich zu den oben definierten Linienmarkierungen kann es vorkommen, dass auf einer maximalen Länge von 1000 mm eine oder zwei Fahrbahnmarkierungen unterbrochen sind.

6.2. Freie Fahrt

Die erste Fahraufgabe umfasst eine freie Fahrsituation auf der vereinfachten Landstraße. Freie Fahrsituation bedeutet in diesem Zusammenhang, dass weder mit Hindernissen auf der Fahrbahn gerechnet werden muss, noch dass Vorfahrtsituationen oder Stopplinien, zum Beispiel an Kreuzungen, beachtet werden müssen.

Während der Fahraufgabe „Freie Fahrt“ muss in zwei verschiedenen Runden ein autonomes Parkmanöver absolviert werden.

Abweichend für die Disziplin „Freie Fahrt“ hat ein durch Mittellinienmarkierungen angezeigtes Überholverbot keine Gültigkeit. Andere Abweichungen der Umwelt sind in dieser Disziplin nicht zu erwarten.

6.3. Hinderniskurs

Die zweite Fahraufgabe, der „Hinderniskurs“, ist komplexer gestaltet als die „Freie Fahrt“, dafür fällt die Pflicht weg, ein Einparkmanöver zu absolvieren (vgl. Carolo-Cup, 2018, S. 20). Zusätzlich wird die Fahrbahn um weitere Elemente erweitert und ist in zwei Bereiche gegliedert. Diese Abweichungen zum generellen Umweltszenario werden in den folgenden Abschnitten definiert.

6.3.1. Landstraße

Der erste Bereich ist ein Landstraßenszenario außerhalb einer Ortschaft. Die Definition der Landstraße bleibt unverändert, jedoch können nun ergänzend statische und dynamische Hindernisse auf der eigenen Fahrbahn sowie auf der Gegenfahrbahn platziert sein.

Statische und dynamische Hindernisse

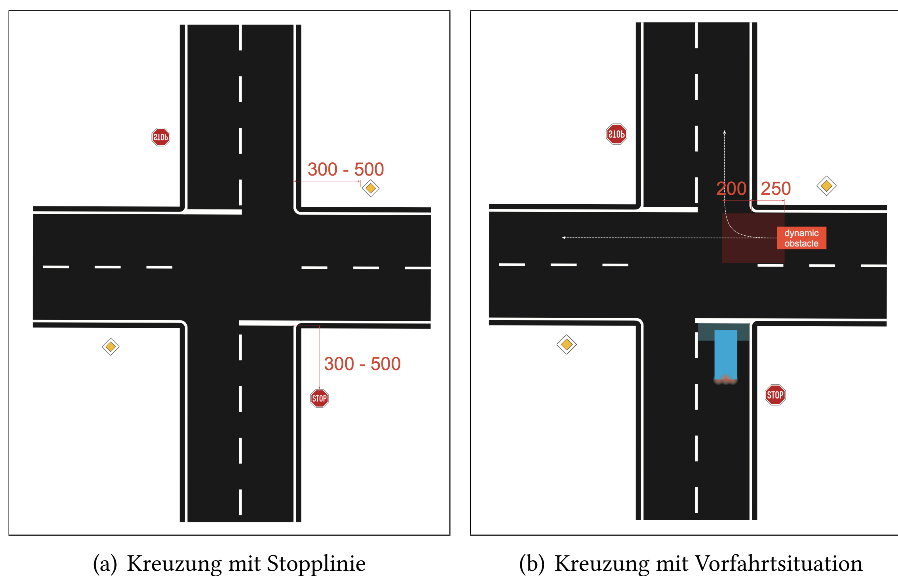
Statische und dynamische Hindernisse sollen andere Fahrzeuge auf der Strecke modellieren. Repräsentiert werden diese durch unterschiedlich große weiße Kartons. Hindernisse können

zwischen 100 mm und 400 mm breit sein. In der Höhe variieren die Hindernisse zwischen 100 mm und 240 mm. Die Länge ist mit mindestens 100 mm definiert. Die statischen Hindernisse haben während der Fahrt einen festen Platz, wohingegen die dynamischen Hindernisse mit einer maximalen Geschwindigkeit von $0.6\,\mathrm{m/s}$ automobil sind. Ein dynamisches Hindernis wird selbst keine Spurwechsel- oder Überholmanöver ausführen. In Bereichen, die mit einem Überholverbot gekennzeichnet sind, ist dem dynamischen Hindernis mit mindestens 300 mm Sicherheitsabstand zu folgen, bis der Überholverbotsbereich wieder aufgehoben wurde.

Kreuzungen

Eine Landstraße kann durch Kreuzungen mit anderen Teilen der Strecke verbunden sein. Die Streckenteile kreuzen sich hierbei orthogonal (vgl. Abbildung 6.3). An zwei der vier Kreuzungssegmenten ist durch eine zwischen 36 mm und 40 mm breite Stopplinie und zusätzliche Beschilderung die Vorfahrt geregelt (vgl. Abbildung 6.3(a)).

Befindet sich die Stopplinie auf der aktuell befahrenen Spur, so ist dort für 3 s zu halten. Darüber hinaus kann sich auf der kreuzenden Strecke von rechts kommend ein dynamisches Hindernis befinden. Ist dies der Fall, dann ist vor der Weiterfahrt die Vorfahrt zu gewähren (vgl. Abbildung 6.3(b)).



(a) Kreuzung mit Stopplinie (b) Kreuzung mit Vorfahrtsituation

Abbildung 6.3.: Kreuzungstypen auf der Landstraße

Kraftfahrstraße

Darüber hinaus kann, durch das Straßenverkehrsschild für eine Kraftfahrstraße, die normale Landstraße mit Gegenfahrbahn in eine zweispurige Kraftfahrstraße umgewandelt werden. Dieser Bereich ist für mindestens 10 m überwiegend gerade gehalten und es sind keine scharfen Kurven zu erwarten. Die Modellierung der Fahrbahn verändert sich für diesen Bereich nicht. Es gilt weiterhin das Rechtsfahrgebot und auf der rechten Spur ist nicht mit Hindernissen zu rechnen.

6.3.2. Suburbanes Szenario

Der zweite Bereich der Fahraufgabe „Hinderniskurs“ ist ein Landstraßenszenario innerhalb einer Ortschaft. Für diesen Bereich werden weitere Elemente auf und neben der Strecke modelliert. Der suburbane Bereich ist in die restliche Strecke eingebettet. Das heißt, dass es einen definierten Anfang und ein definiertes Ende für diesen Bereich innerhalb des ansonsten gültigen und in Abschnitt 6.3.1 beschriebenen Umweltszenarios gibt.

Geschwindigkeitslimit

Der Start des suburbanen Bereichs wird durch ein geschwindigkeitsbegrenzendes Straßenverkehrsschild markiert. Die Geschwindigkeitslimits können zwischen 10 km/h und 90 km/h liegen. Hier ist die tatsächlich gefahrene Geschwindigkeit, wie auch beim Fahrzeugmodell, im Maßstab 1:10 umzurechnen. Beispielsweise beträgt für das Geschwindigkeitslimit 30 km/h, umgerechnet im Maßstab 1:10, die Geschwindigkeit 0.83 m/s. Das Ende des suburbanen Bereichs wird durch das dazugehörige Straßenverkehrsschild, welches die Geschwindigkeitsbegrenzung aufhebt, definiert.

Innerhalb des suburbanen Bereichs treten weitere Elemente auf, die erkannt werden müssen und auf die dann entsprechend reagiert werden muss. Für jedes der nachfolgend vorgestellten Elemente sind am Fahrbahnrand die entsprechenden Straßenverkehrsschilder aufgestellt.

Sperrfläche

Das erste der zusätzlichen Elemente ist eine Sperrfläche. Dies ist eine trapezförmige Fläche auf der Fahrbahn mit abwechselnd schwarzen und weißen Markierungen. Hierbei sind die weißen Markierungen zwischen 36 mm und 40 mm breit und in einem Abstand von 150 mm angeordnet. Der Winkel der Markierungen innerhalb des Trapezes ist mit 27° definiert, der äußere Winkel der Umrandung ist mit 60° festgelegt (vgl. Abbildung 6.4).

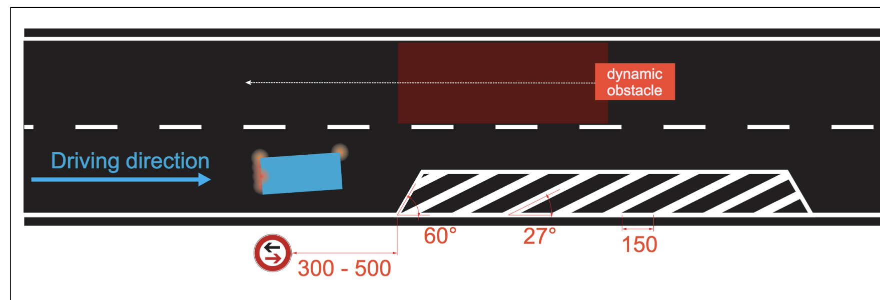


Abbildung 6.4.: Schematische Darstellung der Sperrfläche

Eine Sperrfläche ist mindestens 150 mm breit und maximal 2000 mm lang. Die Sperrfläche ist wie ein Hindernis zu überholen, jedoch muss mit Gegenverkehr gerechnet werden. In diesem Fall ist, wie in Abbildung 6.4 dargestellt wird, vor der Fläche zu warten und das entgegenkommende Fahrzeug passieren zu lassen, bevor der eigene Überholvorgang durchgeführt werden kann (vgl. Carolo-Cup, 2018, S. 22). Für den Fall, dass die Sperrfläche so schmal ist, dass das eigene Fahrzeug ohne Verlassen der eigenen Spur und überqueren der Sperrfläche weiterfahren kann, muss keine Vorfahrt gewährt werden.

Zebrastreifen

Das nächste Element im suburbanen Bereich ist der Zebrastreifen. Dieser wird durch weiße, parallel zur Fahrbahn verlaufende Markierungen modelliert. Die Markierungen sind zwischen 36 mm und 40 mm breit und 400 mm lang. Der Abstand zwischen den weißen Markierungen beträgt 40 mm.

Links und rechts neben dem Zebrastreifen können Fußgänger stehen, die den Zebrastreifen überqueren wollen (vgl. Abbildung 6.5).

Ein Fußgänger wird hierbei durch einen weißen Karton mit einem Piktogramm einer Person auf der dem Fahrzeug zugewandten Seite modelliert (vgl. Abbildung 6.6).

Befindet sich ein Fußgänger auf einer der beiden Fahrbahnseiten, so muss an dem Zebrastreifen gehalten werden, bis der Fußgänger diesen überquert hat.

Erweiterte Kreuzungen

Zusätzlich zu der in Abschnitt 6.3.1 definierten Kreuzung können im suburbanen Bereich weitere Kreuzungstypen vorkommen. Die erste weitere Kreuzungsart ist ähnlich der Stopplinien-Situation, die zuvor definiert wurde. Es kreuzt eine Vorfahrtstraße die eigene Strecke, jedoch nicht mit einer Stopplinie und Stoppschild, sondern mit einer Haltelinie und dem Straßen-

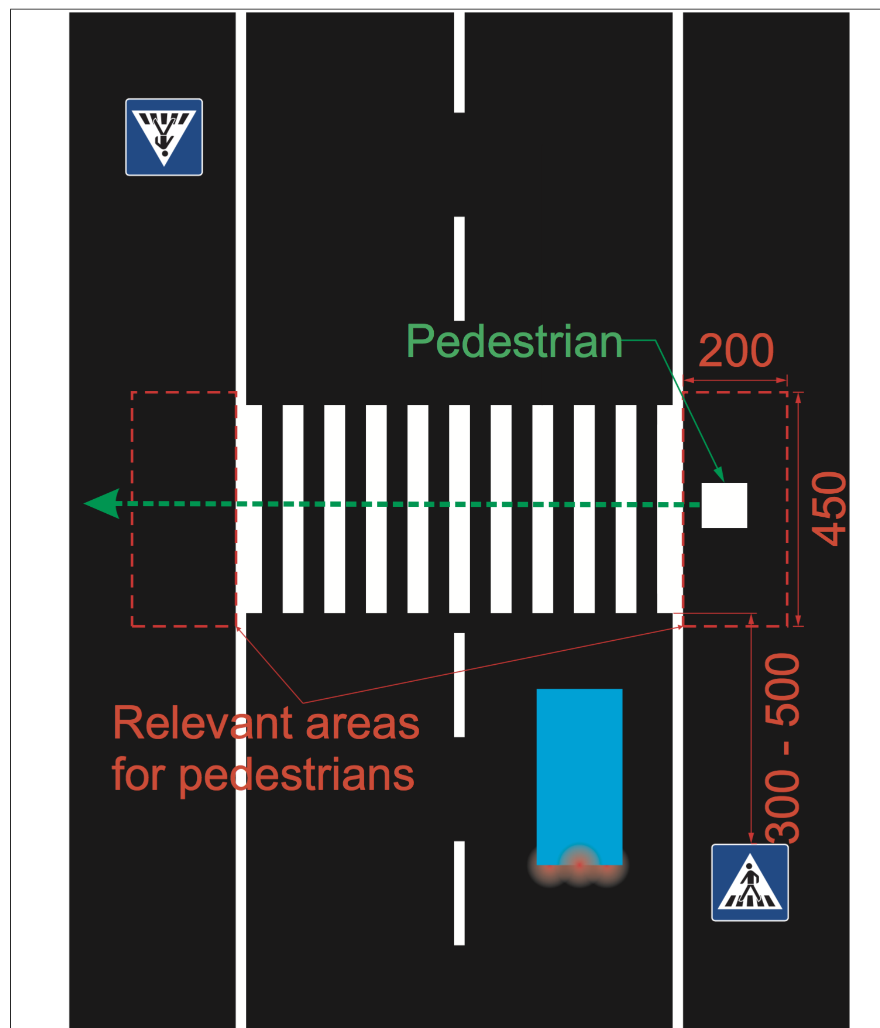


Abbildung 6.5.: Schematische Darstellung des Zebrastreifens

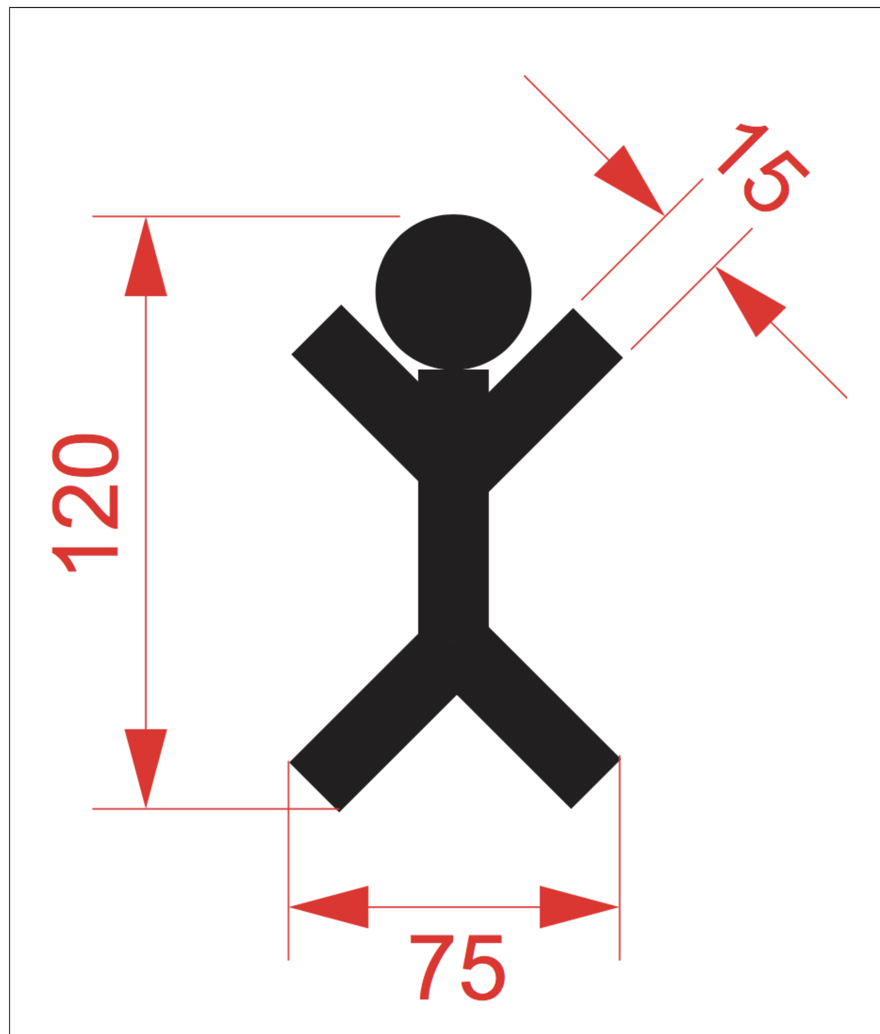
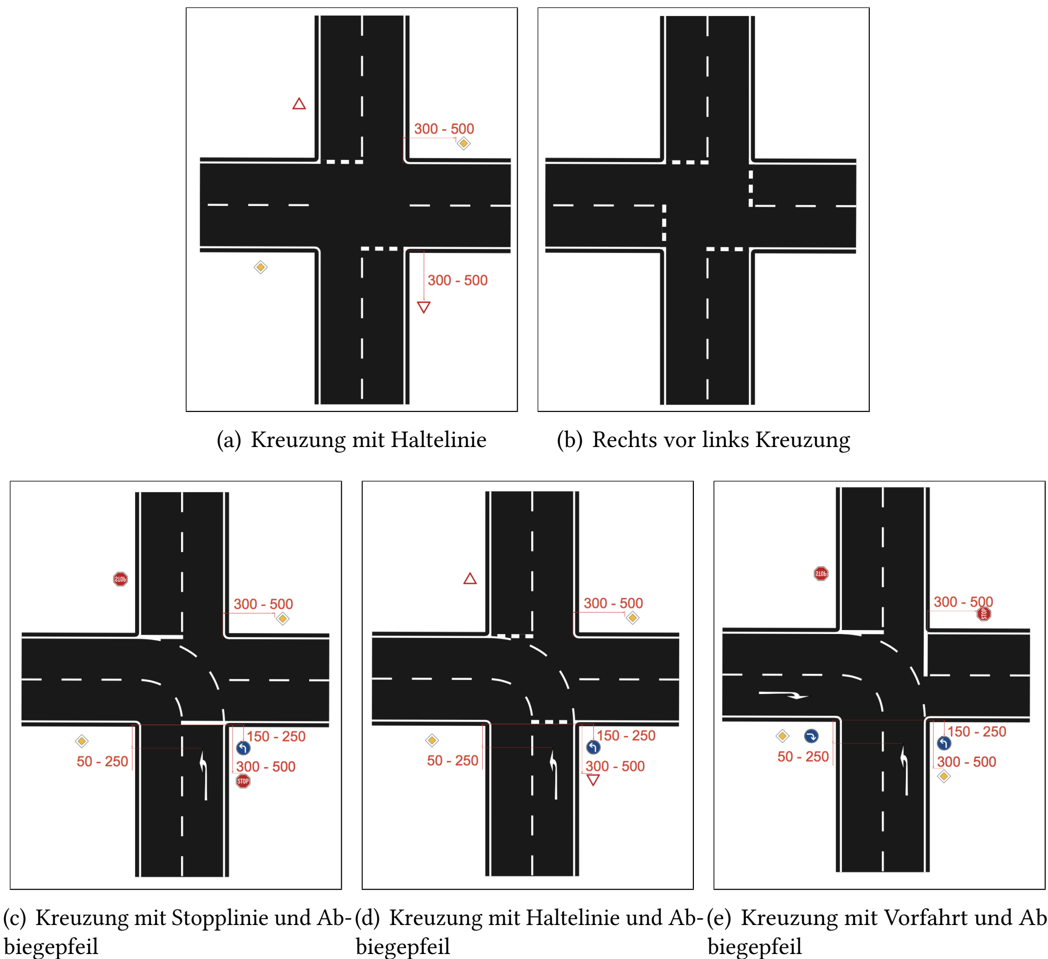


Abbildung 6.6.: Dimensionen eines Fußgängers

verkehrsschild „Vorfahrt gewähren“ (vgl. Abbildung 6.7(a)). Abweichend zu der Stopplinien-Kreuzung muss an der Haltelinie, sofern kein dynamisches Hindernis durchgelassen werden muss, nur 1 s gehalten werden, bevor die Fahrt fortgesetzt werden darf.



(a) Kreuzung mit Haltelinie

(b) Rechts vor links Kreuzung

(c) Kreuzung mit Stopplinie und Abbiegepfeil

(d) Kreuzung mit Haltelinie und Abbiegepfeil

(e) Kreuzung mit Vorfahrt und Abbiegepfeil

Abbildung 6.7.: Die verschiedenen Kreuzungstypen des Carolo-Cup

Die Haltelinie ist hierbei eine gestrichelte Fahrbahnmarkierung mit 80 mm langen Markierungen, unterbrochen auf einer Länge von 60 mm (vgl. Carolo-Cup, 2018, S. 23).

Die zweite Kreuzungsart ist eine rechts-vor-links Situation. Hierfür sind alle Fahrspuren mit einer Haltelinie markiert (vgl. Abbildung 6.7(b)). Es gelten die gleichen Regeln wie für die Haltelinien-Kreuzung. Eine Kreuzung ohne Linienmarkierungen ist ebenso als rechts-vor-links Kreuzung zu betrachten (ohne Abbildung).

Darüber hinaus kann an einer Kreuzung das letzte zusätzliche Element, ein Abbiegepfeil, vorkommen. Dieser zeigt an, dass die Fahrt nicht geradeaus, sondern in der durch den Pfeil angezeigten Richtung fortzusetzen ist. In Kombination mit den zuvor definierten Kreuzungstypen ergeben sich daraus die Kreuzungssituationen, die in Abbildung 6.7 (c) bis (e) dargestellt sind.

7. Simulation der Umwelt

In diesem Kapitel wird dargestellt, wie ausgewählte Elemente des Carolo-Cup und das Wettbewerbsfahrzeug „Fat Lady“ unter Zuhilfenahme des Gazebo-Simulators modelliert werden. Es werden nur ausgewählte Elemente als separates Modell modelliert. Dies liegt daran, dass nicht alle Elemente zwangsläufig ein eigenes Modell benötigen; die planaren Streckenelemente etwa müssen nicht als separates 3D-Modell modelliert werden sondern können stattdessen in die Fahrbahnmodellierung aufgenommen werden.

7.1. Simulierte Umweltelemente

Für die Gazebo-Simulation der abstrakten Umwelt des Carolo-Cup wurden verschiedene Modelle erstellt. Dafür wurden die in Kapitel 6 vorgestellten Definitionen als Grundlage verwendet. Von dieser Grunddefinition ausgehend wurden die Modelle, die im Folgenden vorgestellt werden, vereinfacht modelliert.

7.1.1. Fahrbahn

Der Untergrund für die Carolo-Cup-Fahrbahn wurde als eigenes Modell umgesetzt. Der Grund dafür ist, dass in Gazebo ein Untergrund ein statisches Modell sein muss. Statisch deswegen, weil der Untergrund zwar als Kollissionsobjekt wahrgenommen werden kann, aber äußere Kräfte - wie zum Beispiel Gravitation - auf das Modell keinen Einfluss nehmen. Dadurch bleibt der Untergrund stationär und andere simulierte Elemente - wie zum Beispiel Roboter - können darauf liegen, beziehungsweise fahren.

Modelliert wurde der Untergrund in Anlehnung an das Modell „Asphalt Plane“ von Thomas Koletschka (2014), welches im offiziellen Modell-Repository von Gazebo zu finden ist.

Für die Adaption des Modells auf die Carolo-Umwelt wurden zunächst die Seitenlängen der Grundfläche auf $50\,\mathrm{m}$ vergrößert. Dies ergibt eine Grundfläche von $2500\,\mathrm{m}^2$. Auf den ersten Blick scheint dies sehr groß; es hat aber den Vorteil, dass das Modell des Wettbewerbsfahrzeugs bei Verlassen der Strecke nicht in einen imaginären Abgrund fällt und die komplette Simulation neu gestartet werden muss.

Für die Darstellung der Fahrbahn auf der Grundfläche wird eine PNG-Textur, die mit jedem beliebigen Grafikprogramm veränderbar ist, eingesetzt (vgl. Abbildung 7.1). Die Vorlage für den so erstellten Streckenverlauf ist die Teststrecke aus dem HAW Testlabor (vgl. Abbildung 2.8).

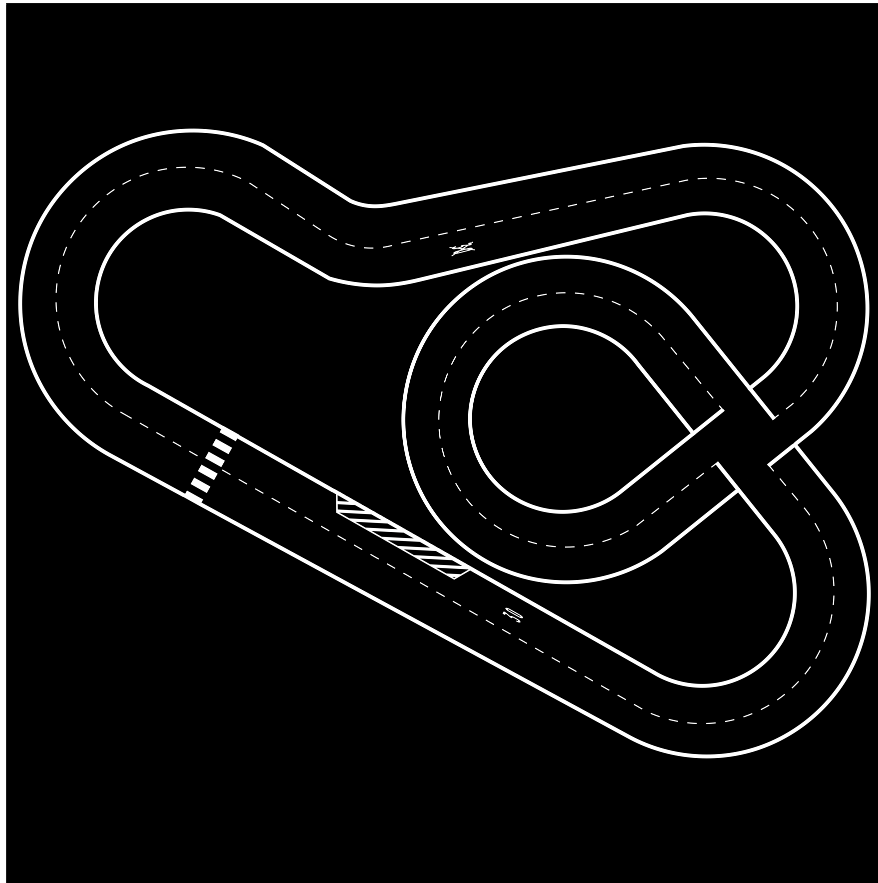


Abbildung 7.1.: Die PNG-Textur der Fahrbahn

In dem Auszug aus der Modelldefinition ist die Festlegung der Größe zu sehen, sowie die nötigen Importe, um eine Textur zu verwenden (vgl. Listing 7.1). Das Skript, welches sich hinter der Datei `carolo.material` verbirgt, lädt die vorbereitete Fahrbahntextur mit einem empirisch ermittelten Skalierungsfaktor. Dies sorgt dafür, dass die Textur nicht 1:1 in derselben Größe wie die Grundfläche vorliegen muss, sondern von der Rendering-Engine dem Wert entsprechend angepasst wird.

Zusätzlich sorgt ein Skalierungsfaktor ≤ 1 dafür, dass die Textur kachelförmig wiederholt wird. Für den gewählten Skalierungsfaktor von $0,32$ bedeutet dies, dass die Textur $3,125$ Mal in X- sowie in Y-Richtung wiederholt wird. Zum einen sorgt die Skalierung dafür, dass die

Fahrbahn die korrekten Dimensionen darstellt, und zum anderen wird durch diese Kachelanordnung die in Abschnitt 6.1 angesprochene unmittelbare Nachbarschaft von Fahrbahnen abgebildet.

```
<visual name="carolo_ground">
    <cast_shadows>false</cast_shadows>
    <geometry>
        <box>
            <size>50 50 .01</size>
        </box>
    </geometry>
    <material>
        <script>
            <uri>model://caroloplane/materials/scripts/carolo.
               material</uri>
            <uri>model://caroloplane/materials/textures/</uri>
            <name>Caroloplane/Image</name>
        </script>
    </material>
</visual>
```

Listing 7.1: Auszug aus der Modelldefinition der Fahrbahn

7.1.2. Hindernisse

Wie die Hindernisse, die beim Carolo-Cup auftreten können, aussehen, wurde bereits in Abschnitt 6.3.1 definiert. Hieraus ergibt sich die Notwendigkeit, ein 3D-Modell mit diesen Eigenschaften zu erstellen.

Da die Hindernisse die sehr einfache Form eines Quaders haben, kann für die 3D-Modellierung die in SDF definierte Grundform der „box“ genutzt werden. Für die Carolo-Umwelt wurden mehrere Modelle, die sich nur in den Maßen unterscheiden, definiert. Dies ist nötig, da die Hindernisse in verschiedenen Größen auftreten können.

In Listing 7.2 ist ein Auszug einer beispielhaften Modellierung eines Hindernisses mit der Grundfläche 100 mm * 100 mm und der Höhe 200 mm nachzuvollziehen. Die ausgelassenen Elemente definieren die Visualisierung mit den gleichen Maßen und einer weißen Oberfläche. Das Setzen des Parameters `<static>` auf `false` sorgt dafür, dass ein Hindernis von Kollisionen beeinflusst werden kann und dadurch seine Position verändert wird. Eine Eigenbewegung wird jedoch nicht aktiv durchgeführt.

Für den Proof of Concept der Simulation wurde auf eine Modellierung der dynamischen Hindernisse verzichtet.

```
<model name="obstacle_10_10_20">
    <static>false</static>
    <link name='body'>
        <pose>.50 0.0 .50 0 0 0</pose>
        <collision name='collision'>
            <geometry>
                <box>
                    <size>.10 .10 .20</size>
                </box>
            </geometry>
        </collision>
    </link>
</model>
```

Listing 7.2: Auszug aus der Modelldefinition für ein statisches Hindernis

7.1.3. Fußgänger

Der Fußgänger, der seitlich am Zebrastreifen auftauchen kann, wurde wie ein Hindernis modelliert und zusätzlich mit einer Textur, vergleichbar mit Abbildung 6.6, versehen. Die Textur wird analog zu Listing 7.1 geladen, jedoch ohne Skalierung. Die simple Methode die Textur ohne Skalierungsfaktor zu laden sorgt zwar dafür, dass alle Seiten mit dem Piktogramm dargestellt werden, kann aber für die Verwendung vernachlässigt werden, da während der Fahrt nur die Vorderseite des „Kartons“ betrachtet wird.

7.1.4. Planare Streckenelemente

Planare Streckenelemente sind die Geschwindigkeitsbegrenzungen und deren Auflösung, die Sperrfläche sowie der Zebrastreifen. Da dies 2D-Elemente sind, ergibt sich nicht die Notwendigkeit, 3D-Modelle hiervon anzufertigen.

Die planaren Streckenelemente, die auf der Strecke während des Carolo-Cup auftreten können, wurden mit in die Textur der Fahrbahn aufgenommen (vgl. Abbildung 7.1).

7.1.5. Straßenverkehrsschilder

Der aktuelle Objekterkennungsansatz des Team NaИ verzichtet auf eine Auswertung der am Fahrbahnrand aufgestellten Straßenverkehrsschilder. Die Idee dahinter ist, dass alle relevanten

Elemente zusätzlich zu der Ankündigung durch Straßenverkehrsschilder am Fahrbahnrand planar auf der Strecke zu erkennen sind. Aus diesem Grund wurde für die Modellierung der Umwelt ebenso darauf verzichtet, die Straßenverkehrsschilder zu modellieren.

Sollten modellierte Straßenverkehrsschilder in der Zukunft benötigt werden, so könnten diese als separate Modelle erstellt und ohne erheblichen Aufwand in die Komposition des Umweltszenarios integriert werden.

7.2. Modellierung „Fat Lady"

Die Modellierung des Fahrzeugs „Fat Lady" besteht aus zwei separaten Modellen. Zum einen wurde das Kernstück der Simulation, die Kamera, als eigenes Modell definiert, zum anderen wurde der Fahrzeugaufbau vereinfacht in die Simulation übertragen. Die Modellierung des Wettbewerbsfahrzeugs besteht aus diesen beiden kombinierten Definitionen.

7.2.1. Kameramodell

Die Modellierung der Kamera macht sich die Sensor-Klasse von Gazebo zunutze. Zunächst wurde ein Würfel mit einer Kantenlänge von $30\,\mathrm{mm}$ definiert. Dies entspricht etwa den Maßen der tatsächlich eingesetzten Kamera.

Der Hauptpart der Modellierung bezieht sich aber auf die Sensor-Klasse. In diesem Teil wird zunächst der horizontale Sichtbereich, der durch die eingesetzte Linse gegeben ist, auf $125.5°$ festgelegt (vgl. Abschnitt 2.3.2). Da nur die obere Hälfte des Bildes genutzt wird, wurde die Kamera direkt auf ein Bild mit der Größe $752 * 240$ Pixel definiert. Der Parameter `<clip>` gibt an, zwischen welchen Entfernungen überhaupt ein Bild gerendert werden soll.

Für eine realistischere Simulation wurde zusätzlich noch ein Sensorrauschen als `<noise>` definiert, welches jeden Pixel des gerenderten Bildes beeinflusst (vgl. Open Source Robotics Foundation, 2014c). Hierbei haben sich die Werte aus dem Gazebo-Beispiel direkt als annehmbar herausgestellt. Die Linse wurde mit dem gleichen horizontalen Sichtbereich definiert.

Zuletzt wurden die Sensoreigenschaften so definiert, dass die Kamera immer angeschaltet ist und mit einer für das Proof of Concept zunächst akzeptablen Updaterate von $20\ \frac{Frames}{Sekunde}$ arbeitet.

Für die komplette Modelldefinition der Kamera sei hier auf Listing A.1 im Anhang verwiesen.

7.2.2. Fahrzeugmodell

Das Fahrzeugmodell für das Wettbewerbsfahrzeug „Fat Lady" setzt sich aus mehreren Komponenten zusammen.

Als erstes wurden die hintere und die vordere Bodenplatte mit dem „box"-Element modelliert. Die beiden Platten wurden über ein starres Gelenk (`<joint>`) miteinander zu der kompletten Bodenplatte verbunden. Dies stellt die erste Vereinfachung an dem Modell dar, da das reale Fahrzeug an dieser Stelle nicht starr verbunden ist (vgl. Abschnitt 2.3.2).

Die zweite Vereinfachung ist ebenfalls Teil der Frontplatte. Für den Einsatzzweck, die Umwelt des Carolo-Cup zu simulieren damit die kamerabasierte Objekterkennung getestet werden kann, wird kein vollständig korrektes Modell des Fahrzeugs benötigt. Aus diesem Grund wurde auf die Modellierung der Ackermann-Lenkung an der Vorderachse verzichtet. Stattdessen wurde eine Lenkrolle (caster wheel) mit einem hohen Schlupf als Ersatz verwendet. Die Rolle ist fest mit der Frontplatte verbunden.

Das dritte Element der Modellierung ist der Kameraturm. Dieser wurde in seinen Dimensionen dem originalen Kameraturm nachempfunden und besteht auch aus dem „box"-Element. Auch der Kameraturm ist über ein starres Gelenk mit der hinteren Bodenplatte verbunden.

In der Modelldefinition für das Fahrzeug wird nach diesen Definitionen das oben beschriebene Kameramodell eingebettet (vgl. Listing B.1). Die Kamera wird starr mit dem Kameraturm verbunden. Anschließend wurden die Hinterräder als `<cylinder>`-Element definiert. Verbunden sind die Hinterräder mit der hinteren Bodenplatte über ein Drehgelenk.

Zuletzt wird das Plugin zur Steuerung des Modells geladen. Dieses Plugin basiert auf dem in Gazebo integrierten „DiffDrivePlugin", welches nur für die Übersetzung mit dem Qt-Creator (Qt-Creator verwendet qmake statt cmake) und im Namen angepasst wurde (vgl. Nate König, 2017).

Für die komplette Modelldefinition des Fahrzeugs sei hier auf Listing B.1 im Anhang verwiesen.

7.3. Komposition der Carolo-World

Die Carolo-World wird mit den verschiedenen oben beschriebenen Modellen befüllt. Hierbei wird ausgiebig von dem `<include>`-Befehl gebrauch gemacht. Für die Positionierung der einzelnen Elemente wurden die Modelle im Gazebo-Client an der richtigen Stelle platziert und anschließend die Werte in das World-File übernommen. Dadurch ergibt sich eine Testumgebung, die alle vom Team NaИ benötigten Elemente enthält (vgl. Abbildung 7.2).

Durch hinzufügen weiterer Modelle kann die Situation, die getestet werden soll, verändert werden. Zusätzlich können verschiedene Testsituationen in weiteren World-Files definiert werden.

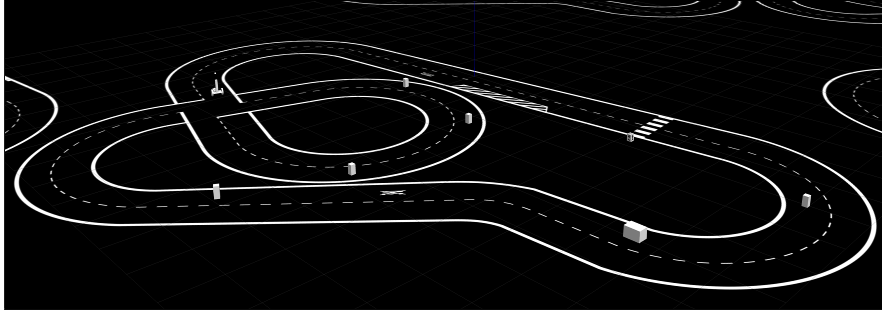


Abbildung 7.2.: Die komponierte Carolo-World

Für die vollständige Definition der Carolo-World sei hier auf Listing C.1 im Anhang verwiesen.

8. Schnittstelle zur Carolo-Software

In diesem Kapitel wird die Schnittstelle zwischen dem Gazebo-Simulator und der Carolo-Software, im speziellen der NaИ-GUI, beschrieben. Dafür wird zunächst darauf eingegangen, wie der Simulator mit der Carolo-Software verbunden wird. Anschließend wird die Benutzung des Simulators in Kombination mit der NaИ-GUI demonstriert.

Zum Schluss wird ein Vergleich der Simulation mit aufgezeichneten Bestandsdaten durchgeführt.

8.1. Verbinden des Simulators

Wie auch die restliche Carolo-Software ist die NaИ-GUI in unterschiedliche Module unterteilt. Die so erreichte Trennung der Zuständigkeiten (separation of concerns) sorgt dafür, dass auch dieser Softwareteil einfach erweiterbar ist.

Für das bisher genutzte Verfahren des Testens mit aufgezeichneten Daten existiert die Klasse `CImageLoader`, die nur dafür zuständig ist, die Bilddaten von der Festplatte einzulesen. Durch adaptieren dieser Klasse wurde eine weitere Klasse `CSimulationLoader` erstellt.

In dieser Klasse geschieht die komplette Kommunikation zwischen der Gazebo-Simulation und der Carolo-Software.

Hierfür werden zunächst die benötigten Header-Dateien von Gazebo geladen. Die in Listing 8.1 aufgeführten Header-Dateien stellen die nötigen Methodendefinitionen zur Verfügung, um eine Verbindung zu einem Gazebo-Server aufzubauen und über die Transportschnittstelle Nachrichten auszutauschen.

```
1 #include "gazebo/gazebo_client.hh"
2 #include "gazebo/transport/transport.hh"
3 #include "gazebo/msgs/msgs.hh"
```

Listing 8.1: Benötigte includes

Um eine Verbindung mit einer laufenden Gazebo-Serverinstanz aufzubauen wird die Methode `setup()` verwendet. Anschließend muss ein Transportknoten zu der generierten

Carolo-Welt erstellt werden (cgl. Listing 8.2). Sofern nur eine Gazebo-Serverinstanz (beziehungsweise nur eine Welt) aktiv ist, muss der Name nicht zwangsläufig übergeben werden; da aber Verbindungen zu anderen simulierten Welten ausgeschlossen werden sollen, wird der Name der für das Carolo-Projekt implementierten Welt mitangegeben.

Nachdem der Transportknoten initialisiert ist, können mit dem in Abschnitt 5.2.1 beschriebenen Publish/Subscribe Verfahren die nötigen Subscriber und Publisher definiert werden. Um in der NaИ-GUI besser nachzuvollziehen in welchem Status die Simulation sich befindet, wird auf das Gazebo-Topic `world_stats` verbunden. Dadurch können aktuelle Werte wie Simulationszeit, Pausezeit oder die Anzahl der Iterationen mit in der NaИ-GUI angezeigt werden. Für diesen Zweck wird beim Verbinden die Callback-Methode `statisticsMessageHandler` übergeben, die die Werte aus einer empfangenen Nachricht in die Datenbasis der Software schreibt.

```
1  // Create a new client (connect to a master gazebo server instance)
2  gazebo::client::setup();
3  // Create a new transport node
4  mNode = gazebo::transport::NodePtr( new gazebo::transport::Node() );
5  // Initialize the node with our world
6  mNode->Init( "CaroloWorld" );
7  // Listen to World Statistic Messages from the System ( Profiling )
8  mWorldStatsSubscriber = mNode->Subscribe("/gazebo/CaroloWorld/
      world_stats", &CSimulationLoader::statisticsMessageHandler, this)
      ;
9  // Control the world
10 mWorldControlPublisher = mNode->Advertise<gazebo::msgs::WorldControl
      >("/gazebo/CaroloWorld/world_control");
11 // We want to start paused
12 mWorldControlMessage.set_pause( true );
13 // Send the message
14 mWorldControlPublisher->Publish( mWorldControlMessage );
15 mWorldControlMessage.Clear();
16 // Connect to the transmitted imagedata
17 mImagesStampedSubscriber = mNode->Subscribe( "/gazebo/CaroloWorld/
      FatLady/camera/carolocam/carolocamera/image", &CSimulationLoader
      ::imageReceived, this );
18 // Create a publisher for the velocity commands
```

```
19 mFatLadyControlPublisher = mNode->Advertise<gazebo::msgs::Pose>("/
      gazebo/CaroloWorld/FatLady/vel_cmd");
```

Listing 8.2: Verbindungsaufbau zum Gazebo-Server

Damit die Bilddaten des in Abschnitt 7.2.1 definierten Kamerasensors empfangen werden können, wird auch hierauf ein Subscriber definiert (vgl. Listing 8.2). In der übergebenen Callback-Methode `imageReceived()` werden die Daten in das `QImage`-Format umgewandelt und für die restliche Software als Input zur Verfügung gestellt (vgl. Listing 8.3). Hierfür werden die Daten aus der Nachricht zunächst in den Gazebo eigenen `Image`-Datentyp umgewandelt und anschließend ein `QImage` daraus erstellt.

```
1  unsigned char *receivedImage = nullptr;
2  unsigned int rgbDataSize = 0;
3
4  gazebo::common::Image gazeboImage;
5
6  gazeboImage.SetFromData(
7              (unsigned char *) message->image(0).data().c_str(),
8              message->image(0).width(),
9              message->image(0).height(),
10             (gazebo::common::Image::PixelFormat) message->image(0).
                  pixel_format()
11             );
12
13 gazeboImage.GetRGBData( &receivedImage, rgbDataSize );
14
15 QImage qImage(
16             message->image(0).width(),
17             message->image(0).height(),
18             QImage::Format_RGB888
19             );
20
21 memcpy( qImage.bits(), receivedImage, rgbDataSize);
22
23 loadQImageIntoBuffer( qImage );
```

Listing 8.3: Auszug aus der Methode imageReceived()

Darüber hinaus werden in Listing 8.2 zwei Publisher definiert. Der erste Publisher verbindet auf das Gazebo-Topic `world_control`. Mit Nachrichten an dieses Gazebo-Topic kann die

Welt von außen manipuliert werden. Genutzt wird dies um ein schrittweises Vorgehen zu ermöglichen oder die Simulation zu pausieren.

Der zweite Publisher verbindet auf das Gazebo-Topic `vel_cmd` des „Fat Lady"-Plugins. Genutzt wird dies, um die in der Carolo-Software berechneten Fahrbefehle an das Modell zu senden. Dadurch setzt sich das Modell von „Fat Lady" in die gewünschte Richtung in Bewegung und der Kamerasensor liefert die zur Bewegung passenden neuen Daten.

8.2. Benutzung des Gazebo-Simulators mit der Carolo-Software

Der Gazebo-Simulator kann, wie in Abschnitt 5.2.2 bereits beschrieben, auf verschiedene Arten gestartet werden. Zunächst kann die Komponente Gazebo-Server mit dem Konsolenbefehl aus Listing 8.4 „headless" gestartet werden. Für eine Visualisierung im Gazebo-Client kann anschließend optional der Konsolenbefehl aus Listing 8.5 verwendet werden. Vereinfacht kann beides zusammen mit dem Konsolenbefehl aus Listing 8.6 gestartet werden. Die letzte Variante bietet sich an, wenn die Komposition der simulierten Umgebung überprüft oder verändert werden soll.

Für die Verwendung der Simulation in Kombination mit der Carolo-Software bietet es sich an, die erste Variante zu verwenden. Die „headless"-Verwendung verbraucht weniger Hardwareressourcen der Entwickler-PCs, da abgesehen von den Kamerasensordaten kein weiteres Rendering berechnet werden muss.

Soll jetzt eine Anpassung an einem Algorithmus validiert werden, so wird zunächst in einer Konsole der Befehl zum Starten des Gazebo-Servers ausgeführt (vgl. Listing 8.4). Anschließend wird die NaИ-GUI gestartet. Die Reihenfolge ist wichtig, damit die Verbindung zwischen den Komponenten aufgebaut werden kann.

Sofern eine Gazebo-Serverinstanz mit der korrekten Carolo-World aktiv ist, werden in der NaИ-GUI direkt die Daten des Kamerasensors empfangen. Da die Carolo-Software die Berechnungen mit einer transformierten Draufsicht durchführt, wird in der NaИ-GUI daraufhin das transformierte Bild angezeigt.

```
1 gzserver nan-world.world
```

Listing 8.4: Starten des Gazebo-Servers

```
1 gzclient
```

Listing 8.5: Starten des Gazebo-Clients

```
1 gazebo nan-world.world
```

Listing 8.6: Starten des Gazebo-Servers in Kombination mit dem Gazebo-Client

8.3. Steuerung der Simulation durch die Carolo-Software

Die NaИ-GUI besitzt in der rechten Seitenleiste verschiedene Schaltflächen (vgl. Abbildung 8.1). Von Oben nach Unten sind dies:

- Exit - Beendet die NaИ-GUI
- Settings - Diverse Einstellmöglichkeiten wie zum Beispiel welche Szene-Items bei Erkennung in das Bild gezeichnet werden sollen
- FAT / LAK - Verbindung zu dem Fahrzeug „Fat Lady“ oder „LaK-XU 5000“ herstellen
- K / H - Umschaltung zwischen Kurs- und Hindernismodus („Freie Fahrt“ beziehungsweise „Hinderniskurs“)
- Automode - Automatisch zum nächsten Datensatz springen (in Kombination mit N / P)
- N / P - Schrittweise Vor- / Zurückgehen (Next beziehungsweise Previous)

Abbildung 8.1.: Die GUI im Überblick

Für die Steuerung der Simulation sind hiervon die Schaltflächen Automode, N und P, sowie K und H von besonderer Bedeutung. Durch drücken einer der Schaltflächen K beziehungsweise H wird die dazugehörige Auswertung der Carolo-Software aktiviert. Die berechneten Fahrbefehle, wie der Lenkwinkel, werden durch betätigen der Schaltfläche N an das Fahrzeugmodell im Simulator geschickt.

Zusätzlich wird die Nachricht `set_multistep(1000)` an das `world_control`-Topic der Carolo-World gesendet, sodass 1000 Iterationsschritte der Simulation durchgeführt werden und die Simulation anschließend pausiert wird. Ein Iterationsschritt entspricht 1 ms, das heißt, die Simulation kann mit einer Schrittweite von 1 s Simulationszeit durchsprungen werden. Die tatsächlich benötigte Berechnungszeit für die Iterationen kann je nach Leistung des Entwickler-PCs davon abweichen. Wird die Schaltfläche Automode vor dem Betätigen von N aktiviert, so wird der nächste Schritt automatisch durchgeführt, sobald ein Frame abschließend ausgewertet wurde. Hierdurch wird ein „Live-Modus“ modelliert.

Die Schaltfläche P wurde für die Simulatoranbindung mit einer anderen Bedeutung überladen. In der bisherigen Nutzung wird die Schaltfläche dafür verwendet, den vorherigen Datensatz zu laden. Da die Simulation aber nur vorwärts berechnet werden kann, ist die selbe Verwendung so nicht möglich. Stattdessen wird die Schaltfläche dafür genutzt, alle Fahrbefehle auf 0 zurückzusetzen, das Fahrzeug also zum Stehen zu bringen, sowie die Simulation zu pausieren.

Durch diese Schaltflächen ist eine einfache Bedienung der im Gazebo-Server laufenden Simulation gegeben.

8.4. Vergleich mit Bestandsdaten

Für den Vergleich der Bestandsdaten mit den simulierten Ergebnissen wurden drei verschiedene Streckensituationen ausgewählt. Die Ergebnisse der Fahrspurerkennung werden von der Carolo-Software, beziehungsweise der NaИ-GUI, in den jeweiligen Frame eingezeichnet. So ist ein direkter Vergleich zwischen den Originalaufnahmen des Fahrzeugs und den simulierten Daten des in Gazebo definierten Kamerasensors möglich. Die gewählten Bilder sind Ausschnitte aus den sonst sehr großen in Draufsicht transformierten Daten. Das komplette transformierte Bild darzustellen hat für diese Vergleiche keinen Mehrwert.

Die Markierungen, die von der Carolo-Software eingezeichnet werden und hier Relevanz haben, sind:

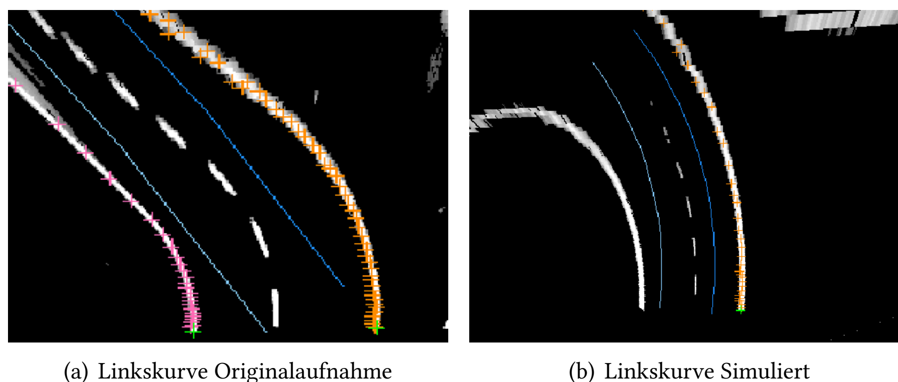
- Oranges Kreuz - Stützpunkt der erkannten rechten Fahrbahnmarkierung
- Pinkes Kreuz - Stützpunkt der erkannten linken Fahrbahnmarkierung

- Grünes Kreuz - Startpunkt(e) für die Liniensuche (vgl. Drauschke, 2016, S. 11ff)
- Hellblaue Linie - Berechnete linke Fahrspur
- Dunkelblaue Linie - Berechnete rechte Fahrspur

Die erste Streckensituation ist eine Linkskurve, wie sie auf der Teststrecke beim Carolo-Cup 2018 in Braunschweig zum Einsatz kam (vgl. Abbildung 8.2(a)). In der modellierten Strecke existiert keine exakt vergleichbare Linkskurve, dennoch werden ähnliche Ergebnisse in der Auswertung der Daten erzielt. Die Auswertung der simulierten Bilddaten in Abbildung 8.2(b) zeigt anhand der Markierungen deutlich, dass die rechte Fahrspurmarkierung erkannt wurde.

Die Tatsache, dass hier weniger eingezeichnete Markierungen auf der Fahrbahnmarkierung zu entdecken sind, liegt daran, dass die Beschreibung des Linienverlaufs weniger Stützpunkte benötigt als der Linienverlauf der originalen Aufnahme. Die blauen Linien zeigen an, dass eine Fahrspur berechnet wurde. Der Verlauf der Fahrspur folgt der Linkskurve innerhalb der Fahrbahnen, sodass hier von einer korrekten Berechnung auszugehen ist.

Abweichend fällt auf, dass die linke Fahrbahnmarkierung nicht korrekt erkannt wurde. Hier ist auch kein Startpunkt zu erkennen, was darauf schließen lässt, dass die Spurbreite in der Simulation nicht vollends korrekt eingestellt ist.



(a) Linkskurve Originalaufnahme

(b) Linkskurve Simuliert

Abbildung 8.2.: Vergleichsszenario Linkskurve

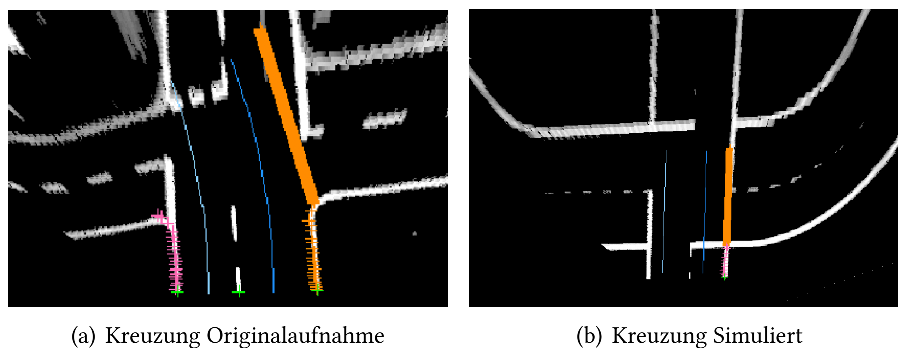
Die zweite Streckensituation stellt eine Kreuzung dar (vgl. Abbildung 8.3). Die Originalaufnahme stammt auch hier von der Teststrecke des Carolo-Cup 2018 in Braunschweig.

In den Originalaufnahmen in Abbildung 8.3(a) sind drei Startpunkte - je einer für die linke, mittlere und rechte Fahrbahnmarkierung - zu erkennen. Davon ausgehend werden die linke Markierung und die rechte Markierung bis zum Kreuzungseingang korrekt erkannt. Die Mittellinienerkennung folgt einem anderen Ansatz und wird nur verwendet, sofern die

Außenlinien nicht detektiert werden konnten, von daher ist dort keine weitere Markierung zu sehen (vgl. Drauschke, 2016, S. 35ff). Der hier deutlich sichtbare orange Balken mitten in die Kreuzung hinein zeigt an, dass tatsächlich eine Kreuzung erkannt wurde. Dies ist eine berechnete Verlängerung der rechts erkannten Fahrbahnmarkierung, wodurch die Lücke, die durch die Kreuzung entsteht, überbrückt werden soll. In der Originalaufnahme ist zu erkennen, dass die Berechnung hier nicht ganz korrekt ist, da die Verlängerung zu sehr nach links kippt.

In den Daten der simulierten Kreuzung in Abbildung 8.3(b) ist erneut zu erkennen, dass lediglich die rechte Fahrbahnmarkierung bis zur Kreuzung korrekt detektiert wurde. Die daraus berechneten Fahrspuren führen korrekt über die Kreuzung.

Zusätzlich fällt auf, dass die Berechnung der Verlängerung in diesen Daten ein besseres Ergebnis als die Berechnung aus der Originalaufnahme geliefert hat. Bei genauerer Betrachtung fällt auf, dass dies an den abgerundeten Markierungen im Originalbild liegt. In der Simulation sind die Ecken kantig mit einem 90°-Winkel angeordnet. Dadurch ist der Kreuzungseingang schärfer abzugrenzen und die Verlängerung wird korrekt berechnet.



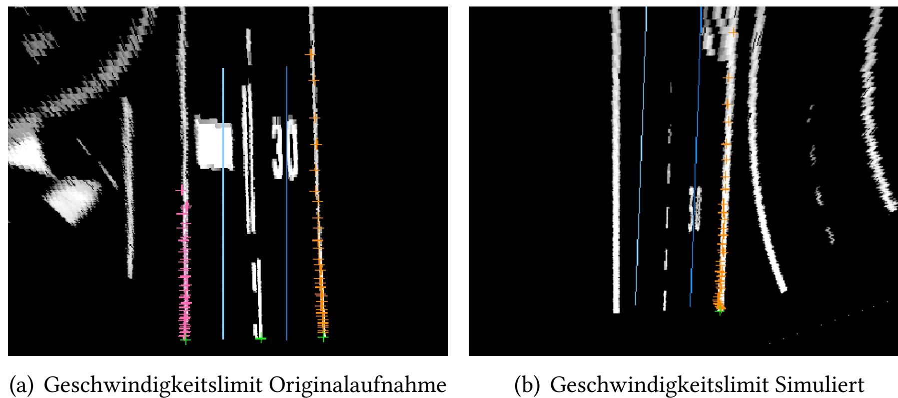
(a) Kreuzung Originalaufnahme (b) Kreuzung Simuliert

Abbildung 8.3.: Vergleichsszenario Kreuzung

Die dritte Streckensituation stellt eine Gerade mit dem Start des suburbanen Bereichs dar (vgl. Abbildung 8.4). Die Originalaufnahme in Abbildung 8.4(a), die diesmal von der Teststrecke des Testlabors der HAW Hamburg stammt, zeigt wieder die drei erwarteten Startpunkte. Davon ausgehend werden die linke und die rechte Fahrbahnmarkierung korrekt erkannt. Die berechnete Fahrspur folgt dem Streckenverlauf und wird als korrekt erkannt.

Die in Abbildung 8.4(b) zu sehende Auswertung der simulierten Daten zeigt das selbe Verhalten wie zuvor. Es werden ein Startpunkt sowie die rechte Fahrbahnmarkierung korrekt detektiert. Die geplanten Trajektorien der Fahrspuren folgen dem Streckenverlauf. Die Erkennung der Fahrspur wird auch hier - mit Einschränkung - korrekt durchgeführt.

In Abbildung 8.4(a) läuft die Spur oben zusammen, in Abbildung 8.4(b) geht die Spur jedoch leicht auseinander. Dass die Fahrbahnmarkierungen in den beiden Bildern unterschiedlich zusammenlaufen, liegt daran, dass die Teststrecke in der HAW Hamburg auf diesem Teilstück zusätzlich die möglichen unterschiedlichen Spurbreiten abdeckt und die Spur an dieser Stelle tatsächlich schmaler wird.



(a) Geschwindigkeitslimit Originalaufnahme (b) Geschwindigkeitslimit Simuliert

Abbildung 8.4.: Vergleichsszenario Geschwindigkeitslimit

Beim Vergleich der Bestandsdaten und den simulierten Daten fallen einige Gemeinsamkeiten auf. Zum Beispiel wird die rechte Fahrbahnmarkierung in allen Abschnitten korrekt detektiert. Auch die Darstellung der Fahrbahn und der Elemente in den simulierten Daten ist vergleichbar mit den Originaldaten der eingesetzten Kamera.

Abweichend fällt allerdings auf, dass die linke Fahrbahnmarkierung in den simulierten Daten in keiner Situation korrekt erkannt werden konnte. Dies deutet wie oben bereits beschrieben darauf hin, dass die Spurbreite nicht im korrekten Toleranzbereich angesiedelt ist. Zusätzlich ist zu erkennen, dass alle Elemente etwas kleiner dargestellt werden. Weiterhin ist zu beobachten, dass die Markierungen der Mittellinie schwächer dargstellt werden und früher aus dem Bild herausgefiltert werden.

9. Fazit

Das Ziel dieser Bachelorarbeit war es, die Testbedingungen des Carolo-Projekts zu optimieren. Die Lösungsstrategie zur Erreichung dieses Ziels wurde in den vorangegangenen Kapiteln erörtert. Es konnte gezeigt werden, dass das in Abschnitt 4.5 ausgewählte Gazebo-Framework zur Erstellung einer simulierten Carolo-Umwelt genutzt werden kann. Durch die eher geringen Hardwareanforderungen des Gazebo-Frameworks und die Möglichkeit, die Gazebo-Simulation ohne Visualisierung betreiben zu können, konnte die gesetzte Anforderung die Testumgebung auf einem Gerät auszuführen erreicht werden.

Der im vorangegangenen Kapitel 8 durchgeführte Vergleich zwischen der simulierten Carolo-Umwelt und den aufgezeichneten Daten der verschiedenen Teststrecken hat gezeigt, dass die eingeführte Abstraktionsebene eine erfolgreiche Auswertung durch die Carolo-Anwendungen ermöglicht. Die hierdurch erzielte Unabhängigkeit von dem Testlabor der HAW Hamburg und von der Fahrzeugplattform hat für die Entwickler im Team NaИ eine deutliche Erleichterung zur Folge, ferner kann mehr Zeit in das Testing investiert werden, um die Erkennungsalgorithmen weiter zu optimieren.

Jedoch ist bei dem Vergleich in Abschnitt 8.4 erkannt worden, dass die Simulation die Carolo-Umwelt noch nicht zu 100 % korrekt wiedergibt. Auffällig war, dass zwar die rechte Fahrbahnmarkierung korrekt erkannt und daraus eine Trajektorie geplant werden konnte, die linke Fahrbahnmarkierung hingegen in keinem der Vergleichsszenarien detektiert wurde. Gleichzeitig wurde festgestellt, dass die planar simulierten Elemente - wie zum Beispiel die Geschwindigtkeitsbegrenzung - kleiner als in den aufgezeichneten Daten dargestellt werden. Dies lässt auf einen noch nicht korrekten Skalierungsfaktor für die Fahrbahn schließen. Dennoch wurde gezeigt, dass das Ziel die Entwicklung im Carolo-Projekt orts- und fahrzeugunabhängig zu gestalten durch die Implementierung des Proof of Concept in weiten Teilen erreicht werden konnte.

Um die Optimierung der Testbedingungen weiter voranzutreiben ist mit der Umsetzung der Simulation in dem Gazebo-Framework eine solide Basis geschaffen worden. Verbesserungspotential ist aber auch hier noch vorhanden. Ein erster Schritt wäre zum Beispiel, den Skalierungsfaktor für die Darstellung der Fahrbahn und der darin enthaltenen planaren Ele-

mente genauer einzustellen. Weiterhin ist es durch den modularen Aufbau der Carolo-World möglich, weitere Modelle - beispielsweise von Straßenverkehrsschildern - zu erstellen und in die Simulation zu integrieren. Zudem wurden an einigen Modellen - wie zum Beispiel beim Fahrzeugmodell - starke Vereinfachungen vorgenommen. Für eine die Realität genauer abbildende Simulation wäre ein Optimierungsansatz, die Modelle exakter zu beschreiben.

Anhang

A. Modelldefinition Kamera

```
<?xml version="1.0" ?>
<sdf version="1.5">
    <model name="camera">
        <pose>0 0 0.05 0 0 0</pose>
        <link name="carolocam">
            <inertial>
                <mass>0.01</mass>
            </inertial>
            <collision name="collision">
                <geometry>
                    <box>
                        <size>0.03 0.03 0.03</size>
                    </box>
                    </geometry>
            </collision>
            <visual name="visual">
                <geometry>
                    <box>
                        <size>0.03 0.03 0.03</size>
                    </box>
                </geometry>
            </visual>
            <sensor name="carolocamera" type="camera">

                <camera>
                <!--horizontal_fov is 125,5 degrees (approx
                   2,1903882 rad)-->
                    <horizontal_fov>2.1903882</horizontal_fov>
                    <image>
                        <width>752</width>
                        <height>240</height>
                    </image>
```

```
32                    <clip>
33                        <near>0.1</near>
34                        <far>100</far>
35                    </clip>
36
37                    <!-- Noise is sampled independently per pixel on
                          each frame.
38                        That pixel's noise value is added to each of
                              its color
39                        channels, which at that point lie in the
                             range [0,1]. -->
40                    <noise>
41                        <type>gaussian</type>
42                        <mean>0.0</mean>
43                        <stddev>0.02</stddev>
44                    </noise>
45
46                    <lens>
47                        <type>stereographic</type>
48                        <scale_to_hfov>true</scale_to_hfov>
49                        <cutoff_angle>2.1903882</cutoff_angle>
50                        <env_texture_size>512</env_texture_size>
51                    </lens>
52
53                </camera>
54
55                <always_on>1</always_on>
56                <update_rate>20</update_rate>
57
58            </sensor>
59        </link>
60    </model>
61 </sdf>
```

Listing A.1: Modelldefinition Kamera

B. Modelldefinition Fat Lady

```
<?xml version=’1.0’?>
<sdf version=’1.4’>
    <model name="FatLady">
        <static>false</static>

        <link name=’backplate’>
            <pose>0 0 .1 0 0 0</pose>
            <inertial>
                <mass>1</mass>
            </inertial>
            <collision name=’collision’>
                <geometry>
                    <box>
                        <size>.2 .2 .005</size>
                    </box>
                </geometry>
            </collision>

            <visual name=’visual’>
                <geometry>
                    <box>
                        <size>.2 .2 .005</size>
                    </box>
                </geometry>
            </visual>
        </link>

        <link name=’frontplate’>
            <pose>.2 0 .1 0 0 0</pose>
            <inertial>
                <mass>1</mass>
            </inertial>
```

```
33            <collision name='collision'>
34                <geometry>
35                    <box>
36                        <size>.2 .1 .005</size>
37                    </box>
38                </geometry>
39            </collision>
40
41            <visual name='visual'>
42                <geometry>
43                    <box>
44                        <size>.2 .1 .005</size>
45                    </box>
46                </geometry>
47            </visual>
48
49            <!--
50                Our model has two front wheels with an ackerman
                     steering,
51                but to keep the simulation "easy" we use a caster
                     wheel
52            -->
53
54            <collision name='caster_wheel_collision'>
55                <pose>0.08 0 -0.03 0 0 0</pose>
56                <geometry>
57                    <sphere>
58                        <radius>0.05</radius>
59                    </sphere>
60                </geometry>
61                <surface>
62                    <friction>
63                        <ode>
64                            <mu>0</mu>
65                            <mu2>0</mu2>
66                            <slip1>1.0</slip1>
67                            <slip2>1.0</slip2>
68                        </ode>
69                    </friction>
70                </surface>
```

```
71              </collision>
72              <visual name='caster_wheel_visual'>
73                  <pose>0.08 0 -0.03 0 0 0</pose>
74                  <geometry>
75                      <sphere>
76                          <radius>0.05</radius>
77                      </sphere>
78                  </geometry>
79              </visual>
80  
81          </link>
82  
83          <link name='cameratower'>
84              <pose>0 0 .25025 0 0 0</pose>
85  
86              <collision name='collision'>
87                  <geometry>
88                      <box>
89                          <size>.05 .05 .3</size>
90                      </box>
91                  </geometry>
92              </collision>
93  
94              <visual name='visual'>
95                  <geometry>
96                      <box>
97                          <size>.05 .05 .3</size>
98                      </box>
99                  </geometry>
100             </visual>
101         </link>
102 
103         <include>
104             <uri>file://../models/carolocamera</uri>
105             <!--
106                 The camera is positioned with 37.73416669 DEG which
                        transforms to ~0.658 RAD
107                 And with a height of 27.5 cm in the middle of the
                        camera
108                 <pose>0.04 0 .275 0 0.658 0</pose>
```

```
109
110                 But as this does not map to the correct view
111                 we set up the camera higher with a smaller angle to
112                 simulate the correct view
113             -->
114             <pose>0.04 0 .55 0 0.33 0</pose>
115         </include>
116
117
118         <link name='back_right_wheel'>
119             <pose>0 -0.115 0.1 0 1.5707 1.5707</pose>
120
121             <collision name='collision'>
122                 <geometry>
123                     <cylinder>
124                         <radius>.08</radius>
125                         <length>.03</length>
126                     </cylinder>
127                 </geometry>
128             </collision>
129
130             <visual name="visual">
131                 <geometry>
132                     <cylinder>
133                         <radius>.08</radius>
134                         <length>.03</length>
135                     </cylinder>
136                 </geometry>
137             </visual>
138         </link>
139
140
141         <link name='back_left_wheel'>
142             <pose>0 0.115 0.1 0 1.5707 1.5707</pose>
143
144             <collision name='collision'>
145                 <geometry>
146                     <cylinder>
147                         <radius>.08</radius>
148                         <length>.03</length>
```

```
149                 </cylinder>
150             </geometry>
151         </collision>
152 
153         <visual name="visual">
154             <geometry>
155                 <cylinder>
156                     <radius>.08</radius>
157                     <length>.03</length>
158                 </cylinder>
159             </geometry>
160         </visual>
161     </link>
162 
163     <!-- Joints -->
164     <joint type="fixed" name="front_back_joint">
165         <child>frontplate</child>
166         <parent>backplate</parent>
167     </joint>
168 
169     <joint type="fixed" name="camera_back_joint">
170         <child>cameratower</child>
171         <parent>backplate</parent>
172     </joint>
173 
174     <joint type="fixed" name="camera_box_joint">
175         <child>camera::carolocam</child>
176         <parent>cameratower</parent>
177     </joint>
178 
179     <joint type="revolute" name="left_wheel_back_hinge">
180         <pose>0 0 -0.03 0 0 0</pose>
181         <child>back_left_wheel</child>
182         <parent>backplate</parent>
183         <axis>
184             <xyz>0 1 0</xyz>
185         </axis>
186     </joint>
187 
188     <joint type="revolute" name="right_wheel_back_hinge">
```

```
189            <pose>0 0 0.03 0 0 0</pose>
190            <child>back_right_wheel</child>
191            <parent>backplate</parent>
192            <axis>
193                <xyz>0 1 0</xyz>
194            </axis>
195        </joint>
196
197        <!-- Plugin to control the driving -->
198        <plugin name="FatLadyDrive" filename="../plugins/
              libPluginFatLadyController.dylib">
199            <alwaysOn>true</alwaysOn>
200            <updateRate>20</updateRate>
201            <left_joint>left_wheel_back_hinge</left_joint>
202            <right_joint>right_wheel_back_hinge</right_joint>
203
204            <torque>20</torque>
205            <commandTopic>cmd_vel</commandTopic>
206            <odometryTopic>odom</odometryTopic>
207            <odometryFrame>odom</odometryFrame>
208            <robotBaseFrame>base_footprint</robotBaseFrame>
209        </plugin>
210
211    </model>
212 </sdf>
```

Listing B.1: Modelldefinition Fat Lady

C. Modelldefinition Carolo-World

```xml
<?xml version="1.0"?>
<sdf version="1.5">
    <world name="CaroloWorld">
    <!-- Lighting -->
    <include>
        <uri>model://sun</uri>
    </include>

    <!-- Underground Plane -->
    <include>
        <uri>file://../models/caroloplane</uri>
        <pose>1.81601 -2.55724 0 0 0 -0.953296</pose>
    </include>

    <!-- Modelcar Fat Lady simplified -->
    <include>
        <uri>file://../models/fatlady</uri>
        <pose>-3.25 -5.26 0 0 0 2.857</pose>
    </include>

    <!-- Obstacles on the track -->
    <include>
        <uri>file://../models/Obs_10_10_20/</uri>
        <name>Obs1</name>
        <pose>-1.12 -1.72 0 0 0 0</pose>
    </include>

    <include>
        <uri>file://../models/Obs_10_10_20/</uri>
        <name>Obs2</name>
        <pose>2.01 -5.6548 0 0 0 0</pose>
    </include>
```

```

    <include>
        <uri>file://../models/Obs_10_10_20/</uri>
        <name>Obs3</name>
        <pose>1.68 -2.332 0 0 0 0</pose>
    </include>

    <include>
        <uri>file://../models/Obs_10_10_20/</uri>
        <name>Obs4</name>
        <pose>8.45 -1.02 0 0 0 0</pose>
    </include>

    <include>
        <uri>file://../models/Obs_10_10_15/</uri>
        <name>Pedestrian</name>
        <pose>4.7 -0.72 0 0 0 0</pose>
    </include>

    <include>
        <uri>file://../models/Obs_10_10_24/</uri>
        <name>Obs5</name>
        <pose>1.66 -7.88 0 0 0 -0.53</pose>
    </include>

    <include>
        <uri>file://../models/Obs_30_20_20/</uri>
        <name>Obs6</name>
        <pose>7.03 -3.95 0 0 0 0</pose>
    </include>

    </world>
</sdf>
```

Listing C.1: Modelldefinition Carolo-World

Literaturverzeichnis

[Berger 2008] BERGER, Arnold S.: Chapter 2 - Testing. In: GANSSLE, Jack (Hrsg.) ; BALL, Stuart (Hrsg.) ; BERGER, Arnold S. (Hrsg.) ; CURTIS, Keith E. (Hrsg.) ; EDWARDS, Lewin A. (Hrsg.) ; GENTILE, Rick (Hrsg.) ; GOMEZ, Martin (Hrsg.) ; HOLLAND, John M. (Hrsg.) ; KATZ, David J. (Hrsg.) ; KEYDEL, Chris (Hrsg.) ; BROSSE, Jean L. (Hrsg.) ; MEDING, Olaf (Hrsg.) ; OSHANA, Robert (Hrsg.) ; WILSON, Peter (Hrsg.): *Embedded Systems*. Newnes, 2008, S. 47–73. – URL https://www.sciencedirect.com/science/article/pii/B9780750686259500051. – ISBN 978-0-7506-8625-9

[Blanco et al. 2016] BLANCO, Myra ; ATWOOD, Jon ; RUSSELL, Sheldon ; TRIMBLE, Tammy ; MCCLAFFERTY, Julie ; PEREZ, Miguel: *Automated Vehicle Crash Rate Comparison Using Naturalistic Data*. 2016. – URL https://www.vtti.vt.edu/PDFs/Automated%20Vehicle%20Crash%20Rate%20Comparison%20Using%20Naturalistic%20Data_Final%20Report_20160107.pdf. – Zugriffsdatum: 2018-04-24

[Carolo-Cup 2018] CAROLO-CUP: *Carolo-Cup Regulations 2018*. 2018. – URL https://wiki.ifr.ing.tu-bs.de/carolocup/regelwerk. – Zugriffsdatum: 2018-04-06

[Christophers 2012] CHRISTOPHERS, Enrico: *FAUST - Fahrerassistenz Und Autonome Systeme*. 2012. – URL http://faust.ful.informatik.haw-hamburg.de/index.php?section=about. – Zugriffsdatum: 2018-04-02

[Daimler AG 2014] DAIMLER AG: *Mercedes-Benz Future Truck 2025 | Daimler > Innovation > CASE > Autonomous*. 2014. – URL https://www.daimler.com/innovation/autonomes-fahren/mercedes-benz-future-truck.html. – Zugriffsdatum: 2018-04-24

[Drauschke 2016] DRAUSCHKE, Clemens: *Echtzeitfähige Startpunktalgorithmen für kamerabasierte Fahrspur-, Kreuzungs- und Hindernisidentifikation*. 2016

[Epic Games 2018a] Epic Games: *About Unreal Engine 4*. 2018. – URL https://www.unrealengine.com/en-US/features. – Zugriffsdatum: 2018-04-07

[Epic Games 2018b] Epic Games: *Hardware & Software Specifications*. 2018. – URL https://docs.unrealengine.com/en-us/GettingStarted/RecommendedSpecifications. – Zugriffsdatum: 2018-04-08

[IDS 2018] IDS: *Datenblatt UI-1221LE-M-GL*. 2018. – URL https://de.ids-imaging.com/IDS/datasheet_pdf.php?sku=AB.0010.1.25700.23. – Zugriffsdatum: 2018-04-05

[Lensation 2017] Lensation: *Datenblatt BT2120*. 2017. – URL https://www.lensation.de/pdf/BT2120.pdf. – Zugriffsdatum: 2018-04-05

[Martin Corden 2010] Martin Corden: *Intel® Compiler Options for Intel® SSE and Intel® AVX Generation (SSE2, SSE3, SSSE3, ATOM_SSSE3, SSE4.1, SSE4.2, ATOM_SSE4.2, AVX, AVX2, AVX-512) and Processor-Specific Optimizations | Intel® Software*. 2010. – URL https://software.intel.com/en-us/articles/performance-tools-for-software-developers-intel-compiler-options-for-sse-generation-and-processor-specific-optimizations. – Zugriffsdatum: 2018-04-19

[Nate König 2017] Nate König: *Osrf / Gazebo / Source / Plugins / DiffDrivePlugin.Hh — Bitbucket*. 2017. – URL https://bitbucket.org/osrf/gazebo/src/fad170569c8cb50f2dce5d8c0e518605dcf0bedc/plugins/DiffDrivePlugin.hh?at=gazebo8_8.1.1&fileviewer=file-view-default. – Zugriffsdatum: 2018-04-21

[OGRE 2018] OGRE: *OGRE - Open Source 3D Graphics Engine | Home of a Marvelous Rendering Engine*. 2018. – URL https://www.ogre3d.org/. – Zugriffsdatum: 2018-04-10

[Omron Adept 2016] Omron Adept: *MobileRobots Pioneer 3-AT (P3AT) Research Robot Platform*. 2016. – URL http://www.mobilerobots.com/ResearchRobots/P3AT.aspx. – Zugriffsdatum: 2018-04-10

[Open Source Robotics Foundation 2014a] Open Source Robotics Foundation: *Gazebo*. 2014. – URL http://gazebosim.org/. – Zugriffsdatum: 2018-04-09

[Open Source Robotics Foundation 2014b] OPEN SOURCE ROBOTICS FOUNDATION: *Gazebo : Tutorial : Gazebo Components*. 2014. – URL http://gazebosim.org/tutorials?tut=components&cat=get_started. – Zugriffsdatum: 2018-04-16

[Open Source Robotics Foundation 2014c] OPEN SOURCE ROBOTICS FOUNDATION: *Gazebo : Tutorial : Sensor Noise Model*. 2014. – URL http://gazebosim.org/tutorials?tut=sensor_noise. – Zugriffsdatum: 2018-04-21

[Open Source Robotics Foundation 2014d] OPEN SOURCE ROBOTICS FOUNDATION: *SDF*. 2014. – URL http://sdformat.org/. – Zugriffsdatum: 2018-04-10

[Open Source Robotics Foundation 2018a] OPEN SOURCE ROBOTICS FOUNDATION: *Gazebo : Tutorial : Beginner: Overview*. 2018. – URL http://gazebosim.org/tutorials?cat=guided_b&tut=guided_b1. – Zugriffsdatum: 2018-04-09

[Open Source Robotics Foundation 2018b] OPEN SOURCE ROBOTICS FOUNDATION: *Gazebo : Tutorial : Mac*. 2018. – URL http://gazebosim.org/tutorials?tut=install_on_mac&cat=install. – Zugriffsdatum: 2018-04-09

[Open Source Robotics Foundation 2018c] OPEN SOURCE ROBOTICS FOUNDATION: *Osrf / Gazebo_models*. 2018. – URL https://bitbucket.org/osrf/gazebo_models. – Zugriffsdatum: 2018-04-16

[Open Source Robotics Foundation 2018d] OPEN SOURCE ROBOTICS FOUNDATION: *SDF*. 2018. – URL http://sdformat.org/spec?ver=1.6&elem=world. – Zugriffsdatum: 2018-04-16

[Open Source Robotics Foundation 2018e] OPEN SOURCE ROBOTICS FOUNDATION: *SDF*. 2018. – URL http://sdformat.org/spec?ver=1.6&elem=model. – Zugriffsdatum: 2018-04-16

[Paulweber und Lebert 2014] PAULWEBER, Michael ; LEBERT, Klaus: *Mess- und Prüfstandstechnik: Antriebsstrangentwicklung - Hybridisierung - Elektrifizierung*. Springer, 2014 (Der Fahrzeugantrieb). – OCLC: 903689356. – ISBN 978-3-658-04452-7 978-3-658-04453-4

[Rat für Forschung und Technologieentwicklung 2013] RAT FÜR FORSCHUNG UND TECHNOLOGIEENTWICKLUNG: *Empfehlung Zu Einer Optimierten Proof-of-Concept-Unterstützung Im Wissenstransfer*. 2013. – URL http://www.rat-fte.at/tl_files/uploads/Empfehlungen/131203_ProofOfConcept_Empfehlung_NP.pdf. – Zugriffsdatum: 2018-04-23

[The Qt Company 2018] The Qt Company: *All Modules | Qt 5.10*. 2018. – URL https://doc.qt.io/qt-5.10/qtmodules.html. – Zugriffsdatum: 2018-04-15

[Thomas Koletschka 2014] Thomas Koletschka: *Osrf / Gazebo_models / Source / Asphalt_plane — Bitbucket*. 2014. – URL https://bitbucket.org/osrf/gazebo_models/src/ea482a46e821ac9fe9618e469c5bf65742b42eb1/asphalt_plane/?at=default. – Zugriffsdatum: 2018-04-20

[Thomas Lehmann 2014] Thomas Lehmann: *Software und Systems Engineering - Hardware in the Loop / Software in the Loop*. 2014

[TU Braunschweig 2018] TU Braunschweig: *Wiki Carolo-Cup*. 2018. – URL https://wiki.ifr.ing.tu-bs.de/carolocup/carolo-cup. – Zugriffsdatum: 2018-03-30

[Turing 1937] Turing, A. M.: On Computable Numbers, with an Application to the Entscheidungsproblem. s2-42 (1937), Nr. 1, S. 230–265. – URL http://dx.doi.org/10.1112/plms/s2-42.1.230

[Unity Manual 2018] Unity Manual: *Unity - Manual: Physics*. 2018. – URL https://docs.unity3d.com/Manual/PhysicsSection.html. – Zugriffsdatum: 2018-04-07

[Unity Technologies 2018a] Unity Technologies: *Unity - Fast Facts*. 2018. – URL https://unity3d.com/de/public-relations. – Zugriffsdatum: 2018-04-07

[Unity Technologies 2018b] Unity Technologies: *Unity - System Requirements*. 2018. – URL https://unity3d.com/de/unity/system-requirements. – Zugriffsdatum: 2018-04-07

[V-Play GmbH 2018a] V-Play GmbH: *Showcases*. 2018. – URL https://v-play.net/showcases/. – Zugriffsdatum: 2018-04-09

[V-Play GmbH 2018b] V-Play GmbH: *V-Play Installation | V-Play 2.15 | V-Play Engine*. 2018. – URL https://v-play.net/doc/vplay-installation/#install-requirements. – Zugriffsdatum: 2018-04-09

[VDI 2018] VDI: *Carolo Cup 2018: Dr. Drift setzt auf Schnelligkeit*. 2018. – URL www.vdi.de/technik/fachthemen/fahrzeug-und-verkehrstechnik/artikel/carolo-cup-2018-dr-drift-setzt-auf-schnelligkeit/. – Zugriffsdatum: 2018-04-05

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 25. April 2018 Florian Dannenberg