Daniel Hammer, BSc

Entwicklung einer Softwareumgebung für einen Fahrsimulator

MASTERARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

Masterstudium Elektrotechnik

eingereicht an der

Technischen Universität Graz

Betreuer

Assoc. Prof. Dipl.-Ing. Dr. techn. Arno Eichberger

Institut für Fahrzeugtechnik

Begutachter

Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Eugen Brenner

Institut für Technische Informatik

Graz, Oktober 2015

Kurzfassung

Simulatoren sind heutzutage ein nicht mehr wegzudenkendes Instrument um Forschung und Entwicklung zu betreiben. Sei es zur Konzeption von neuen Komponenten, zum Testen von in Entwicklung befindlichen oder zur Rekonstruktion von Unfallhergängen, die Einsatzgebiete sind dabei praktisch unbeschränkt.
Im Rahmen des Forschungsprojekts *MueGen Driving* wurde am Institut für Fahrzeugtechnik der Technischen Universität Graz ein Fahrsimulator entwickelt, welcher momentan vorrangig zur Forschung rund um das große Gebiet der Fahrassistenzsysteme sowie dem autonomen Fahren eingesetzt wird. Diese Arbeit setzt sich mit den verschiedenen Aspekten der Entwicklung dieses Fahrsimulators auseinander.

Beginnend mit einem Überblick der verbauten Hardware- als auch Softwarekomponenten werden anschließend verschiedene Software-Module im Detail betrachtet. Dies beinhaltet vor allem Erläuterungen zur Erstellung verschiedener Programmbibliotheken welche für den Betrieb des Simulators unumgänglich sind.
Neben der Integration einer Echtzeitumgebung wird weiters auf die Einbindung der Visualisierung, der Akustiksimulation als auch eines Eye-Trackers eingegangen. Weitere Punkte stellen die Entwicklung von Tablet-Applikationen zur Visualisierung des Dashboards als auch zur Steuerung der Simulation an sich dar.
Einen wesentlichen Punkt in einem Fahrsimulator stellen gezielt erstellte Fahrszenarien dar. Aus diesem Grund wird tiefer gehend auf die Erstellung einer Bibliothek zur Erstellung eben solcher Szenarien eingegangen.

Abschließend werden nach einer Zusammenfassung Ideen zur Weiterentwicklung des Fahrsimulators präsentiert.

Abstract

Today, simulators have become an indispensable part of research and development. No matter if they are used for designing new concepts, for testing products which are currently in development or for the reconstruction of circumstances of an accident. The field of applications is more or less unlimited.
In the context of the research project *MueGen Driving*, a driving simulator has been developed at the Institute of Automotive Engineering at Graz University of Technology. It currently is primarily used in the area of Driver Assistance Systems and Autonomous Driving. This thesis deals with the different aspects of the development of said simulator.

After a summary of used hard- and software components, a more detailed view is given describing the different software modules being developed during the project.
This includes the integration of a real time environment, the incorporation of an existing visualisation and acoustic system as well as the integration of an eye-tracker. Furthermore two tablet applications were developed. One to mimic a dashboard which could also be used in the real car, and another one to enable the operator to control the simulator as a whole.
One important part of a driving simulator is the generation of different traffic scenarios. Therefore an application library has been developed which is explained more in-depth.

Finally, a summary as well as multiple ideas for further development of the driving simulator are presented.

EIDESSTATTLICHE ERKLÄRUNG

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am 28. Oktober 2015 .....
(Unterschrift)

STATUTORY DECLARATION

I declare that I have authored this thesis independently, that I have not used other than the declared sources / resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, October 28th 2015 .....
(signature)

Danksagung

An dieser Stelle möchte ich die Gelegenheit wahrnehmen um mich bei all jenen zu bedanken, die mich im Laufe der Jahre begleitet und vor allem unterstützt haben. Dies schließt insbesondere die Freunde und Kollegen mit ein die ich im Laufe des Studiums kennenlernen durfte und die mir stets mit Rat und Tat zur Seite gestanden sind.

Diese Arbeit entstand am Institut für Technische Informatik in enger Zusammenarbeit mit dem Institut für Fahrzeugtechnik der Technischen Universität Graz. Ich möchte mich vor allem meinen beiden Betreuern Assoc. Prof. Dipl.-Ing. Dr. techn. Arno Eichberger sowie Ao. Univ.-Prof. Dipl.-Ing. Dr. techn. Eugen Brenner für die Hilfsbereitschaft sowie die unzähligen investierten Stunden bedanken.

Zuletzt gilt mein allergrößter Dank meiner Familie, insbesondere jedoch meinen Eltern, die mir stets die volle Unterstützung haben zukommen lassen und zu jedem Zeitpunkt hinter mir gestanden sind. Ohne sie wären diese Worte nicht möglich.

Graz, im Oktober 2015 Daniel Hammer

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Kapitel 1

Einleitung

1.1 Motivation

Moderne technische Systeme haben heutzutage einen derart hohen Komplexitätsgrad erreicht, dass ohne Simulationen Neu- sowie Weiterentwicklungen praktisch nicht mehr realisierbar sind. Dabei ist es unerheblich welche Branchen man hierbei betrachtet. Durch die Vernetzung, Interdisziplinarität sowie der hohen Integrationsdichte verschmelzen Bereiche welche zuvor noch getrennt voneinander betrachtet werden konnten zusehends immer mehr.

Auch im Bereich der Mobilität schreitet dieser Trend unaufhaltsam voran. War die Automobilbranche vor einigen Jahren noch sehr stark auf den maschinenbaulichen Sektor konzentriert, so ist jetzt auch die Elektrotechnik, die Informationstechnik, das Chemieingenieurwesen, aber beispielsweise auch der psychologische Bereich zu großen Teilen vertreten. Es ist davon auszugehen, dass diese Entwicklung auch noch in Zukunft anhalten wird.

Um die Vielzahl an unterschiedlichen Komponenten im Verbund testen zu können, haben sich Simulatoren als das Mittel der Wahl herauskristallisiert. Diese ermöglichen es, bereits in einem frühen Stadium Komponenten miteinander zu verschmelzen um sie im Verbund testen zu können.
Simulatoren beschleunigen den Entwicklungsprozess enorm, wobei gleichzeitig eine Senkung der Kosten herbeigeführt werden kann. Durch die Simulation kann beispielsweise die Realisierung von aufwändigen Prototypen auf einen wesentlich späteren Zeitpunkt verschoben werden.

Es ist davon auszugehen, dass die Bedeutung von Simulatoren in Zukunft noch weiter zunehmen wird. Besonders im Bereich des autonomen Fahrens sowie der Car2Car-Kommunikation sind die Möglichkeiten praktisch unbegrenzt. Es ist daher naheliegend in diesen Bereich zu investieren und Know-how aufzubauen, um für zukünftige Erfordernisse entsprechend gerüstet zu sein.

1.2 Forschungsprojekt MueGen Driving

Das Forschungsprojekt MueGen Driving beschäftigt sich mit dem geschlechts- und altersspezifischen Fahrverhalten in kritischen Situationen bei verschiedenen Straßenverhältnissen.
Statistiken zeigen, dass mit fortschreitendem Einsatz von Fahrassistenzsystemen die Zahl an Verkehrsunfällen sinkt. Nichts desto trotz ist zu beobachten, dass tendenziell bei jüngeren Männern nicht angepasste Geschwindigkeit die Hauptunfallsursache darstellt, während das Risiko eines Unfalls im Rahmen von Abbiege- oder Wendemanövern sowie beim Ein- und Ausfahren bei Frauen erhöht ist.

Das MueGen Driving Projekt stellt nun die Frage, ob alters- oder geschlechtsspezifische Unterschiede im Fahrverhalten in verschiedenen Fahrsituationen bestehen und ob eine Adaptierung des Systemverhaltens von Fahrassistenzsystemen an das Fahrverhalten einen Komfort- und Sicherheitsgewinn nach sich ziehen kann.
Im Rahmen des MueGen Driving Projekts wurden Realversuche durchgeführt, wobei 10 Frauen sowie 10 Männer unterschiedlichster Altersklassen teilnahmen. Ziel war es, subjektive Bewertungen des Notfallbremsassistenten sowie des adaptiven Abstandstempomats in Relation zu objektiven Messdaten zu setzen. Zu diesem Zweck wurden Fahrmanöver definiert, wobei anschließend von den Versuchsprobanden ein detaillierter Fragebogen zu beantworten war. Die Fahrmanöver waren wie folgt:

- AEB: Auffahren auf stillstehendes Fahrzeug
- AEB: Auffahren auf kreuzenden Fußgänger
- ACC Autobahn: Auffahren auf langsames Target bei verschiedenen Zeitlücken
- ACC Autobahn: Folgefahrt variabel bei verschiedenen Zeitlücken
- ACC Autobahn: Überholen bei verschiedenen Zeitlücken
- ACC Autobahn: Cut-In bei verschiedenen Zeitlücken
- ACC City: Stop&Go bei verschiedenen Zeitlücken

Im Rahmen der Realversuche hat sich gezeigt, dass keine signifikanten Unterschiede bei der Wahrnehmung der untersuchten Assistenzsystemen zwischen Männern und Frauen bezüglich des Regelverhaltens der Assistenzsysteme festgestellt werden konnte. Sehr wohl wurden jedoch Abweichungen bei der Bewertung der Bedienbarkeit, des Komforts sowie des Sicherheitsgefühls festgestellt.

Als Folge der relativ kleinen Anzahl an Versuchspersonen sowie dem hohen Aufwand welcher zur Durchführung von Realversuchen notwendig ist, wurden alle weiteren Untersuchungen in einem, im Rahmen dieser Arbeit mitentwickelten, Fahrsimulator durchgeführt. Dies ermöglichte eine wesentlich größere Anzahl an Probandenversuchen bei gleichzeitiger Reduktion des Zeitaufwands pro Versuchsperson, eine Verminderung von Risiko und Kosten sowie ein größeres Spektrum an möglichen Manövern. Weiters war es machbar

verschiedene Umweltbedingungen praktisch nach Belieben nachzustellen, was in Realversuchen nicht ohne Weiteres möglich ist.

Zum Zeitpunkt dieser Arbeit wurden im Fahrsimulator Versuche mit ca. 100 Probanden durchgeführt, wobei der Testablauf an den zuvor erwähnten Realversuch angelehnt worden ist. Hinzugekommen ist ein Winterszenario bei vermindertem Reibwert, um den Einfluss von Schnee und Eis auf die Fahrassistenzsysteme sowie die Probanden untersuchen zu können. Eine Auswertung dieser Studie ist zum momentanen Stand noch in Bearbeitung, weshalb an dieser Stelle noch keine Ergebnisse präsentiert werden können.

1.3 Übersicht

Im Laufe der Masterarbeit wurde eine Vielzahl unterschiedlicher Arbeiten durchgeführt. Um einen groben Überblick über diese zu geben, seien nachfolgend die wichtigsten angeführt.

- Kapitel 3: Einarbeitung und Inbetriebnahme der Echtzeitumgebung PTWinSim
- Kapitel 3.4: Entwurf verschiedener Simulationsmodelle wie ACC, AEB, Radar, u.v.m. in Matlab/Simulink
- Kapitel 4, Kapitel 5: Implementierung der Fahrdynamiksimulationen AVL VSM sowie IPG CarMaker
- Kapitel 6: Entwicklung einer Schnittstelle zum IO Modul
- Kapitel 7: Implementierung einer Schnittstelle zur Kommunikation mit der Visualisierung
- Kapitel 8: Entwicklung von Android-Applikationen zur Simulation des Dashboards
- Kapitel 9: Entwicklung einer Applikation zum Lesen und Setzen von Daten zur Laufzeit, zur Generierung von Straßenverläufen sowie zum Auslösen von Simulations-Events
- Kapitel 10: Entwicklung einer Tablet-Applikation zur Steuerung der Fahrsimulation
- Kapitel 11: Implementierung einer Schnittstelle zur Kommunikation mit der Akustiksimulation
- Kapitel 12: Entwicklung einer Szenariensimulation zur Generierung von Verkehr
- Kapitel 13: Entwicklung einer Schnittstelle zu einem Eye-Tracker

1.4 State of the Art

Simulatoren haben bereits eine lange Geschichte hinter sich. In den 1930er Jahren entwickelte die Firma Link Aviation Devices, Inc Flugsimulatoren, welche unter dem Namen Link Trainer vertrieben wurden, wobei im Laufe der Zeit mehr als 500.000 Piloten der

alliierten Kräfte trainiert wurden.
Diese Simulatoren verfügten in ihrer ersten Ausführung weder über eine Visualisierung, Sound oder gar eine Bewegungsplattform. Der Trainer bestand maßgeblich aus einer Holzkiste mit darin verbauten Instrumenten sowie den Flight-Controls.
Wie so oft waren militärische Entwicklungen maßgeblich für den Fortschritt der Simulationstechnik verantwortlich, weshalb der Link Trainer in zweiter Revision bereits über eine sehr einfache Simulation von Bewegung sowie über eine Operator-Station verfügte.

Nach dem zweiten Weltkrieg fanden massive Fortschritte im Bereich der Simulatoren statt, wobei der Fokus nach wie vor hauptsächlich im Bereich der Flugsimulatoren lag. Fahrsimulatoren waren zur damaliger Zeit nur von zweitrangiger Bedeutung. Inzwischen haben jedoch auch die Simulationen von Geräuschen, Bewegung als auch Visualisierung den Weg in die Simulatoren gefunden.

Erst in den 80er Jahren haben sich Simulatoren auch in andere Bereiche als den Flugverkehr ausgebreitet. Neben dem Bahn- als auch Straßenverkehr war es auch hier wieder der militärische Bereich welcher zur weiteren Verbreitung beiträgt, wobei die Simulatoren diesmal zur Ausbildung der Benutzung von neuen Waffensystemen, Panzern oder zur Simulation von Truppenbewegungen eingesetzt werden.

1.4.1 Einsatzgebiete

Grundsätzlich können heutzutage die nachfolgenden Einsatzgebiete unterschieden werden.

Forschung

Einer der großen Vorteile von Simulatoren ist, eine großen Anzahl von Versuchen mit geringem Material- und Zeiteinsatz durchführen zu können. Aus diesem Grund eignen sich Simulatoren hervorragend, um Forschung unter Zuhilfenahme von Versuchsprobanden zu betreiben. Es ist möglich, Forschungsgebiete zu untersuchen, welche in Realfahrzeugen aufgrund der Sicherheits- und Gesetzeslage nicht ohne weiteres möglich wären. Dies schließt unter anderem folgende Bereiche mit ein:

- Untersuchung des Einflusses von Alkohol und, sofern erlaubt, Drogen
- Untersuchung des Einflusses von Übermüdung
- Untersuchung des Einflusses verschiedener Ablenkungen wie Smartphones, Computer oder ähnliches
- Untersuchung weiterer Einflüsse welche die Fahrtüchtigkeit beeinflussen

Neben oben genannten Punkten können weitere Punkte genannt werden, welche zwar auch in Realversuchen untersucht, aus oben genannten Gründen jedoch im Simulator unter wesentlich kontrollierten Bedingungen durchgeführt werden können. Einer der wichtigsten Vorteile ist die hundertprozentige Wiederherstellung der Randbedingungen bei Versuchsdurchführung, was eine sehr gute Vergleichbarkeit der Tests ermöglicht. Die weiteren Punkte umfassen unter anderem die

- Untersuchung der Usability von Human Machine Interfaces
- Untersuchung der Wahrnehmung und Wirkung verschiedenster Assistenzsysteme
- Untersuchung der Wirkung verschiedener Warneinrichtungen
- Untersuchung des Einflusses verschiedener Wetterbedingungen auf das Fahrverhalten
- Untersuchung verschiedener psychologischer Effekte beispielsweise bei schlechten Ampelschaltungen oder Stau

Entwicklung

Der Bereich der Entwicklung geht Hand in Hand mit jenem der Forschung, wobei hier der Schwerpunkt tatsächlich auf der Entwicklung der zu testenden Systeme an sich liegt. Die neu entwickelten Komponenten werden zu definierten Zeitpunkten Versuchsgruppen zugeführt um deren Feedback direkt in die Weiterentwicklung einfließen lassen zu können.

Bei der Entwicklung spielen HIL- (Hardware In the Loop) sowie DIL- (Driver In the Loop) Tests eine wichtige Rolle. So ist es möglich, neue Hardware-Komponenten bereits zu einem sehr frühen Zeitpunkt unter möglichst realen Bedingungen testen zu können, ohne auf einen Einbau in ein Realfahrzeug angewiesen zu sein. Mit fortschreitender Entwicklung können immer mehr in Software simulierte Komponenten durch echte Hardware ersetzt werden, bis schlussendlich der Einbau in ein Versuchsfahrzeug erfolgt.

Der Einsatz von Simulatoren im Rahmen der Entwicklung bringt unter anderem folgende Vorteile mit sich:

- Reduktion der Entwicklungskosten
- Beschleunigung der Entwicklung
- Auffinden von Fehlern zu einem frühen Zeitpunkt
- Einbinden der Komponente Mensch zu einem frühen Zeitpunkt
- Testen unter kontrollierten Randbedingungen bei hoher Reproduzierbarkeit
- Verminderung des Unfallrisikos aufgrund unfertiger Komponenten
- Unabhängigkeit von Wettereinflüssen

Training

Wie bereits zuvor kurz erläutert, werden Simulatoren vor allem in der Luftfahrt sowie im militärischen Bereich für das Training eingesetzt. Hierbei wird in der Regel ein Höchstmaß an Realitätstreue angestrebt, da

1. Geld für hochwertige Komponenten in diesen Bereichen in der Regel ausreichend vorhanden ist
2. Schnell eine große Anzahl an Menschenleben betroffen ist
3. Trainingsstunden im Realobjekt um ein Vielfaches teurer sind als jene im Simulator

Ein Höchstmaß an Realitätstreue schließt Bewegungsplattformen, hochwertige Akustiksimulationen, ausgeklügelte Visualisierungsmethoden sowie einen realitätsgetreuen Innenraum samt realistischen Force-Feedback Systemen mit ein.
Neben den Hardware-Komponenten müssen die simulierten Systeme exakt jenen des Originals entsprechen, da falsche antrainierte Automatismen verheerende Folgen im realen Umfeld haben können. Für falsch antrainiertes Verhalten wird allgemein der Begriff Negativ-Training verwendet.
Negativ-Training muss jedoch nicht immer aufgrund von falsch implementierten Systemen zu Stande kommen. Auch ein falsch benutzter Simulator kann trotz korrekter Implementierung ein Negativ-Training nach sich ziehen.
Trägt ein Kampfpilot oder Rennfahrer im Simulator beispielsweise keinen Helm, so hat dieser ein Blickfeld sowie Bewegungsspielraum des Kopfes, welcher in der realen Umgebung nicht gegeben ist. So kann es passieren, dass antrainierte Blickmuster oder Bewegungsabläufe im realen Umfeld nicht umgesetzt werden können, was schlussendlich zu unerwünschte Situationen führen kann. Aufgrund dessen müssen auch Trainingsleiter in der Regel eine Ausbildung absolvieren, um eine korrekte Ausführung des Trainings gewährleisten zu können.

1.4.2 Visualisierung

In der Simulationstechnik sind heutzutage die nachfolgenden 4 Konzepte zur Visualisierung der Umgebungsbedingungen verbreitet.

Monitore

Der im Rahmen des MueGen Driving Projekts entstandene Fahrsimulator arbeitet mit Hilfe von 8 Monitoren, wobei 4 im Bereich der Frontscheibe und 4 hinter den Seitenscheiben angebracht sind. Nähere Informationen zum genauen Aufbau sind Kapitel 2.1.5 zu finden.
Die Darstellung mit Hilfe von Monitoren wird vor allem bei kompakteren sowie günstigeren Simulatoren gerne verwendet, wobei folgende Vorteile für diese Art der Visualisierung sprechen:

- Es ist eine sehr kompakte Bauweise möglich
- Je nach Anforderungen sind preisgünstige Realisierungen möglich
- Es ist eine Vielzahl an Standardkomponenten verfügbar
- Durch moderne Bildschirme sind sehr hohe Auflösungen erreichbar
- Lange Lebensdauer
- Hohe Lichtstärke sowie hoher Kontrast

Diesen Vorteilen stehen jedoch auch einige Nachteile gegenüber:

- Beim Zusammenfügen mehrerer Monitore wird, selbst bei Verwendung von Spezialmonitoren, stets ein schwarzer Rand beim Übergang zu sehen sein

- In der Regel werden mehrere Computer/Grafikkarten benötigt als bei der Umsetzung mit anderen Methoden
- Der mechanische Aufbau ist in der Regel komplizierter als bei anderen Lösungen
- Durch das Zusammenfügen mehrerer Monitore erhält man geometrisch nicht glatte Übergänge was zu Bildverzerrungen führen kann
- Relativ starke Erwärmung in geschlossenen Räumen

Projektor

Visualisierungen welche mit Leinwänden und Projektoren arbeiten bilden in der Regel einen guten Kompromiss zwischen Kosten und erreichbarer Bildqualität und werden demnach gerne in semi-professionellen Simulator-Lösungen eingesetzt. Die wichtigsten Vorteile dieser Varianten wären:

- Gutes, homogenes Bild bei gleichzeitig großer Abdeckung
- 3 Projektoren reichen in der Regel für 180 Grad aus, womit es möglich ist, mit nur einem Computer auszukommen
- Gutes Preis/Leistungsverhältnis
- Relativ einfache Montage

Jedoch sind auch hier einige Punkte zu nennen, welche gegen den Einsatz von Projektoren sprechen

- Erreichbare Auflösungen im Vergleich zur Realisierung mit Monitoren relativ gering
- Erreichbare Kontraste im Vergleich zu Monitoren relativ gering
- Sensibel gegen Erschütterungen (Leinwand als auch Projektoren), wodurch laufende Nachjustierungen notwendig werden können
- Schnelle Alterung der Komponenten (Birnen), was sich in den laufenden Betriebskosten niederschlägt

Virtual Reality

Die mit Abstand neueste Variante um die Visualisierung in Simulatoren zu bewerkstelligen ist jene welche mit Hilfe von Head-Mounted Displays arbeitet. Hierbei wird mit Hilfe einer großen Virtual-Reality Brille jedem Auge ein eigener Bildschirm zugeführt, wodurch es möglich ist, einen stereoskopischen Eindruck zu erzeugen. Zusätzlich beinhalten diese Systeme noch Beschleunigungs- sowie Drehratensensoren um ein exaktes Head-Tracking zu ermöglichen.
Die großen Vorteile dieses Systems sind

- Extrem geringe Kosten im Vergleich zu allen anderen Visualisierungs-Lösungen

- Praktisch kein zusätzlicher Platzbedarf
- Stereoskopische Visualisierung
- Berücksichtigung der Kopfbewegung
- Reale Cockpits sind nicht notwendig, weshalb diese virtuell innerhalb von Sekunden ausgetauscht werden können

Den Vorteilen stehen jedoch auch eine Vielzahl an Nachteilen gegenüber, weshalb der Einsatz in Produktiv-Systemen genauestens überlegt werden muss.

- Die Technologie befindet sich noch im Anfangsstadium
- Es sind noch relativ wenige Werkzeuge zur Entwicklung verfügbar
- Die erreichbaren Auflösungen sind aufgrund des geringen Abstands der Augen zu den Bildschirmen noch sehr beschränkt
- Zum momentanen Zeitpunkt ist es noch nicht möglich das gesamte Blickfeld abzudecken
- Virtual-Reality Systeme haben wesentlich häufiger Simulator-Sickness zur Folge als andere Lösungen
- Die momentan noch großen Brillen können irritierend wirken, beziehungsweise kann es bei spezifischen Simulations-Szenarien möglich sein dass diese gar nicht einsetzbar sind
- Die Hände sowie die reale Umwelt sind nicht sichtbar, was das Einsatzgebiet stark einschränkt

Vor allem die fehlende Sicht auf die Hände sowie die Instrumente im Cockpit ist einer der größten Nachteile dieser Systeme. Trainieren beispielsweise Rennfahrer im Simulator, so müssen diese während einer Runde ständig mehrere Parameter auf dem Lenkrad verstellen um das Auto stets im Optimalbereich bewegen zu können. Dies ist mit Head-Mounted Displays praktisch nicht möglich. Zwar ist es machbar, Hände, Lenkrad sowie Drehregler virtuell in das Sichtfeld einzublenden, das wirkliche Interagieren mit den Bedienelementen ist jedoch nicht möglich.
Ähnlich verhält es sich beispielsweise bei Piloten. Auch diese müssen selbst in Standardsituationen eine Unmenge an verschiedenen Hardware-Elementen bedienen um einen korrekten Ablauf gewährleisten zu können. Zusätzlich kommen hier noch Trainingsszenarien, wo beispielsweise aufgrund von Rauchbildung im Cockpit Gasmasken aufgesetzt werden müssen, hinzu. Auch dies ist aufgrund der unhandlichen Brillen momentan noch nicht möglich.

Collimated Displays

Die in Simulatoren mit Abstand komplexesten Lösungen sind jene, welche mit Hilfe von so genannten Collimated Displays arbeiten. Zwar arbeitet dieses System auch mit Projektoren, doch werden hierbei die ausgesendeten Strahlen über ein Linsensystem sowie gekrümmten Spiegel auf die Augen gelenkt. Der Spiegel ist hierbei als Folie ausgeführt und befindet sich direkt vor den Probanden. Das Linsensystem hingegen sitzt in der Regel, zusammen mit den Projektoren, oberhalb des Simulator-Innenraums.
Der große Vorteil hierbei ist, dass die Lichtstrahlen so gut wie parallel auf die Augen auftreffen, wodurch diese, obwohl die Entfernung zum Spiegel nur ein bis drei Meter beträgt, so fokussieren, als würde sich das Bild sehr weit weg befinden. Der Proband muss demnach ständig umfokussieren wenn der Blick von der Umgebung in Richtung Cockpit-Inneres wandert, so wie es eben auch in der Realität der Fall wäre.
Zusammenfassend wären die wesentlichen Vorteile dieser Art der Visualisierung:

- Fokussierung des Auges gegen Unendlich und Umfokussierung bei Blick in das Cockpit-Innere
- 180 Grad Visualisierung
- Hoher Realitätsgrad

Die großen Nachteile können wie folgt genannt werden:

- Sehr hohe Kosten
- Hoher Platzbedarf, ähnlich wie bei Leinwand-Projektion
- Komplexes System da der Spiegel, welcher als Folie ausgeführt ist, durch Unterdruck in die richtige Position gezogen wird.
- Spiegel anfällig gegen mechanische Einflüsse

Collimated Displays werden heutzutage vor allem in Flugsimulatoren eingesetzt, da diese die höchste Ansprüche an den Realismus stellen. Nachdem die Reaktionszeit der Piloten im Falle einer Notfallsituation ein wesentlicher Punkt ist, ist die Tatsache, dass das Auge bei Blick auf die Instrumente neu fokussieren muss, von fundamentaler Bedeutung.

1.4.3 Bewegungsplattform

Stewart Platform

Heutzutage haben sich in Simulatoren Bewegungsplattformen nach dem Stewart-Prinzip, wie in Abbildung 1.1 dargestellt, durchgesetzt. Diese bieten 6 Freiheitsgrade, wobei es auch Simulatoren gibt, bei denen die Plattform noch zusätzlich auf ein oder zwei Schienensystemen angebracht ist. Durch diese Maßnahme vergrößert sich der Bewegungsspielraum enorm wodurch länger anhaltende Beschleunigungen wesentlich besser nachgestellt werden können.

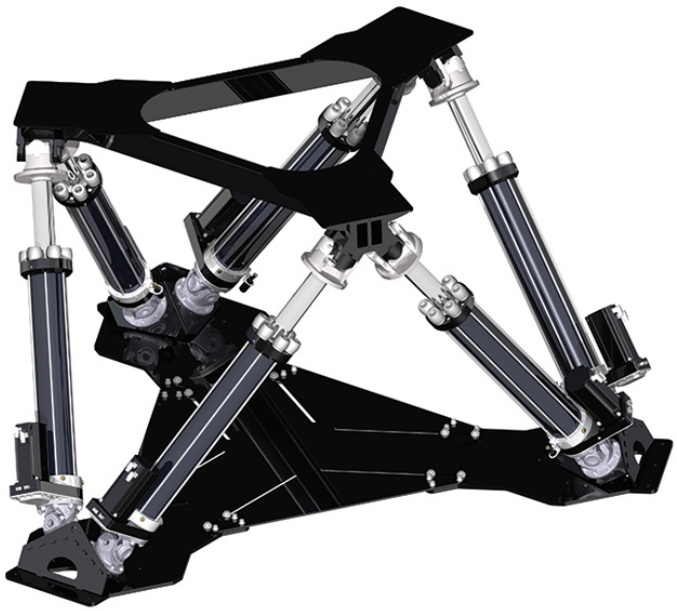

Abbildung 1.1: Stewart Motion Platform [1]

Achse	Beschleunigung	Auslenkung
Längs	$\pm$ 6.5m/s^2	$\pm$ 1.42m
Quer	$\pm$ 6.5m/s^2	$\pm$ 1.23m
Vertikal	$\pm$ 9.0m/s^2	$\pm$ 0.98m
Rollen	$\pm$ 160°/s^2	$\pm$ 29.0°
Neigen	$\pm$ 160°/s^2	$\pm$ 33.0°
Yaw	$\pm$ 240°/s^2	$\pm$ 33.0°

Tabelle 1.1: Typische Kennwerte einer mittelgroßen Bewegungsplattform [2]

Diese Art der Plattform gibt es in den verschiedensten Größen sowie Ausführungsformen, wobei sich heutzutage rein elektrische Systeme durchgesetzt haben. Zu Beginn der Entwicklung waren vor allem hydraulische Systeme relevant, welche jedoch wesentliche Nachteile aufwiesen. So waren diese durch die Hydraulik-Aggregate laut, wartungsanfällig, benötigten wesentlich mehr Bauraum und waren vor allem träge, was sich negativ auf die erreichbare Performance auswirkte.
Typische Werte einer modernen Plattform wären wie in Tabelle 1.1 angeführt.

Seilsysteme

Während die Stewart Platform schon lange auf dem Markt und bewährt ist, so ist zu beobachten, dass im Zusammenhang mit der wachsenden Verbreitung von Virtual-Reality Brillen neue Konzepte entwickelt werden. Mit diesen ist es beispielsweise nicht länger notwendig schwere Lasten zu tragen, um im Simulator reale Cockpits nachstellen

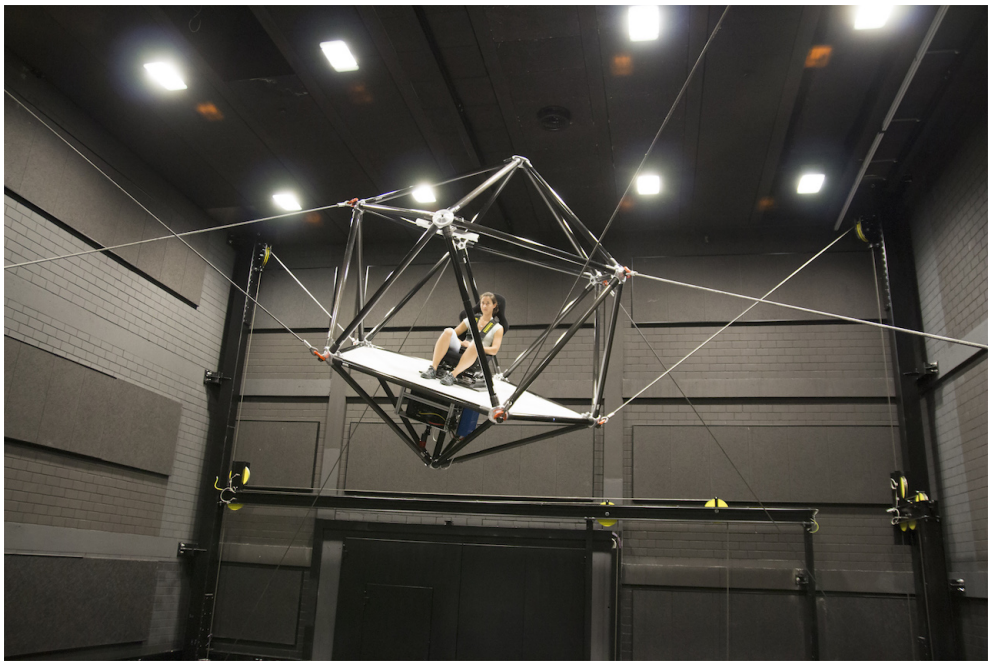

Abbildung 1.2: Cable-driven Robot [3]

zu können.

Eine völlig neue Entwicklung ist hierbei der Cable-driven Robot aus Abbildung 1.2, wo ein Kohlefaser-Käfig anhand von 8 Seilen in einer Halle verfahren werden kann. Im Vergleich zur Stewart-Plattform ist aufgrund der wesentlich größeren Auslenkungen die Dauer der möglichen Beschleunigungen bei gleicher Amplitude um ein Vielfaches größer.

Ein großer Nachteil ist jedoch die Flexibilität der Seile, weshalb hochfrequente Signale nicht so gut wiedergegeben werden können. Diesem Nachteil könnte jedoch durch zusätzliche Aktuatorik, direkt am Sitz angebracht, entgegengewirkt werden.

Kapitel 2

Überblick

Ein Fahrsimulator deckt eine große Anzahl von Komponenten ab. Die Software steht in direktem Kontakt mit der angebundenen Hardware, welche wiederum die Schnittstelle an den Fahrer repräsentiert. Nachfolgend soll kurz auf die einzelnen Bestandteile eingegangen sowie erläutert werden, wie diese untereinander in Verbindung stehen. Abbildung 2.1 gibt bereits einen Überblick darüber, wie die einzelnen Software- als auch Hardwarekomponenten miteinander kommunizieren, bzw. welche Abhängigkeiten zwischen ihnen bestehen.

Simulator PC: Zentrales Element ist der Simulator PC, dem unter anderem folgende Aufgaben zuteil werden:

- Ausführen sämtlicher Simulationsmodelle als Echtzeit-Tasks inklusive Fahrdynamik
- Simulation der Akustik
- Kommunikation mit dem Analog-IO-Device
- Kommunikation mit dem CAN-Interface
- Bereitstellen aller Daten für die Visualisierung
- Bereitstellen aller Daten für die HMIs welche als Tablets ausgeführt sind

Visualisierung: Wie in Kapitel 7 beschrieben, steht der Simulations PC in direktem Kontakt mit einem der 4 Visualisierungsrechner, welcher als zentrale Schnittstelle für die Umgebungsdarstellung fungiert. Dieser hat folgende Aufgaben:

- Bereitstellen der Umgebungsszenarien sowie das Verteilen selbiger an alles Slaves
- Verarbeiten der Positionsdaten aus der Simulation
- Synchronisation der Visualisierungsrechner untereinander
- Verarbeitung der vom Eye-Tracker zur Verfügung gestellten Daten

An jeden der 4 Visualisierungsrechner sind je 2 Monitore angeschlossen, was eine 180 Grad Abdeckung des Sichtfeldes erlaubt.

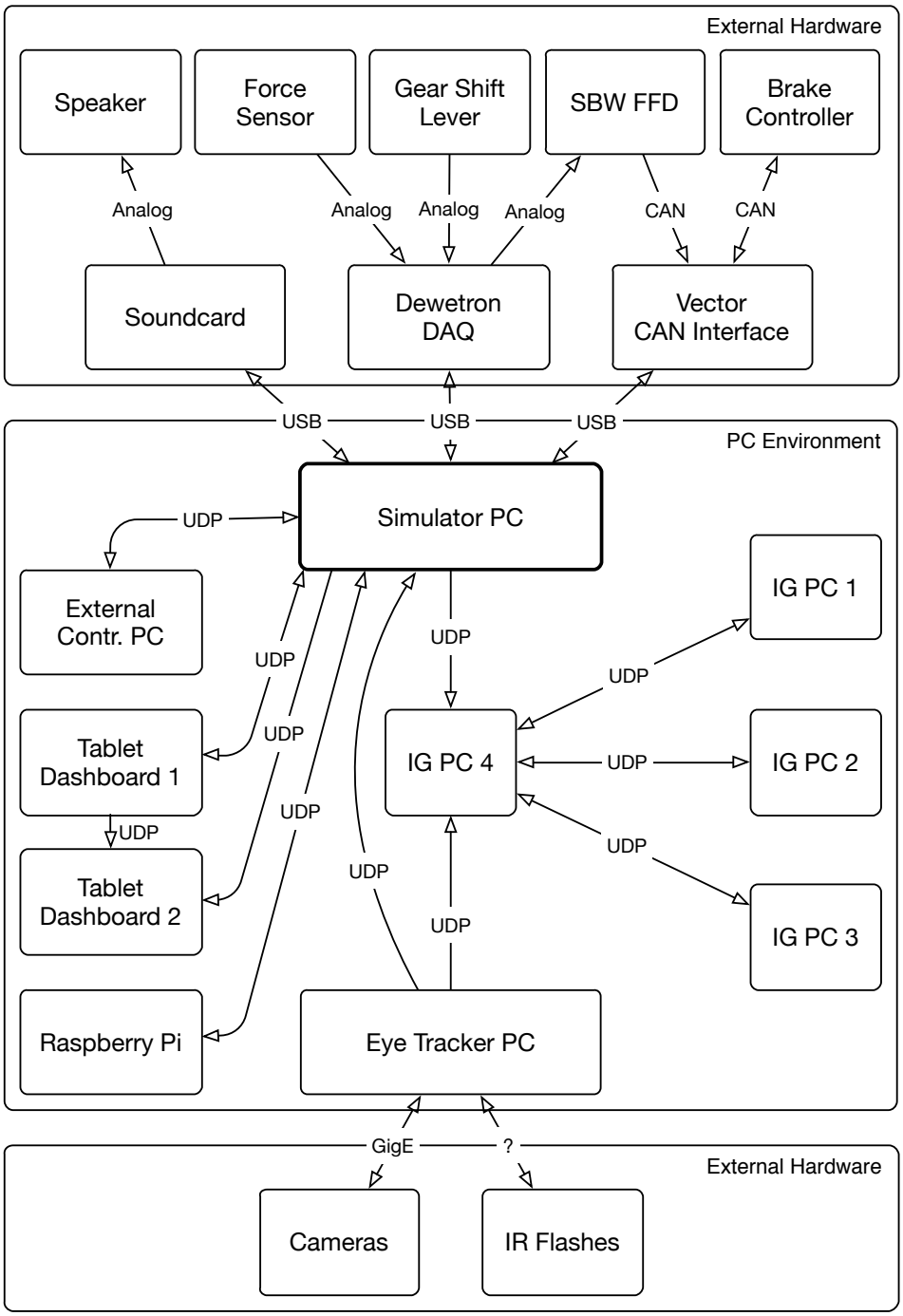

Abbildung 2.1: Grundstruktur des Fahrsimulators

Eye-Tracker: Es besteht die Möglichkeit, den Fahrsimulator im 3D-Betrieb ohne Anwendung etwaiger Shutterbrillen oder anderer störender Mechanismen zu betreiben. Dies wird durch den Einsatz von Parallax-Barrieren im Zusammenspiel mit einem Eye-Tracker zur Bestimmung der Augenpositionen erreicht. In den Kapiteln 2.1.5 und 13 wird anschließend etwas näher auf die Funktionsweise dieses Systems eingegangen. Der Eye-Tracker ist demnach zuständig für

- Das Bereitstellen der Augenpositionen in Echtzeit
- Die Bereitstellung von Points of Interest, sofern gewünscht
- Die Aufnahme von Videos zum Zwecke des Post-Processing

HMIs: Aus Gründen der Flexibilität werden die Bedienelemente als auch Tacho und sonstige Instrumente digital als Tablets ausgeführt. Diese übernehmen im momentanen Ausbaustadium folgende Aufgaben.

- Bedienung von CC, ACC, AEB samt Geschwindigkeitseinstellung
- Darstellung der eingestellten sowie aktuellen Geschwindigkeit
- Darstellung des aktuellen Status der Assistenzsyteme samt Warnungen

In Kapitel 8 werden die Details näher erläutert.

ForceFeedback: Neben der visuellen ist die haptische Wahrnehmung ein ganz wesentlicher Punkt welcher dem Fahrer wichtige Informationen über den Zustand seines Fahrzeugs vermittelt. Aus diesem Grund kommt für die Lenkung ein Force-Feedback System von der Firma SBW Technology zum Einsatz, während das Bremspedal am Institut für Fahrzeugtechnik der TU Graz entwickelt wurde. Beide Systeme sind elektrische Systeme, weshalb auf den Einsatz von Hydraulik verzichtet werden kann.
Bei der Schaltung findet der originale Automatik-Schalthebel Verwendung. Somit wird kein Kupplungspedal für die Simulation benötigt.

Akustik: Ebenso wichtig wie die visuelle und haptische Wahrnehmung ist auch die akustische. Aus diesem Grund werden neben der bestehenden Board-Anlage weitere Niederfrequenz-Boxen installiert um entsprechende Vibrationen in das Fahrzeug einbringen zu können.
Nähere Informationen werden nachfolgend in den Kapiteln 2.1.4 und 11 vermittelt.

2.1 Hardware

In den kommenden Kapiteln soll ein Überblick über die eingesetzte Hardware bzw. eine kurze Einführung in die mechanischen Gegebenheiten vermittelt werden.
Abbildung 2.2 zeigt die Außenansicht des Simulators. Erwähnenswert ist, dass ein Vollfahrzeug samt intakten Innenraum zum Einsatz kommt, was dem Fahrer ein zusätzliches Maß an Realismus vermittelt.
Der Aluminium-Aufbau, bestehend aus Item-Profilen, ist fix mit der Karosserie eines Mini

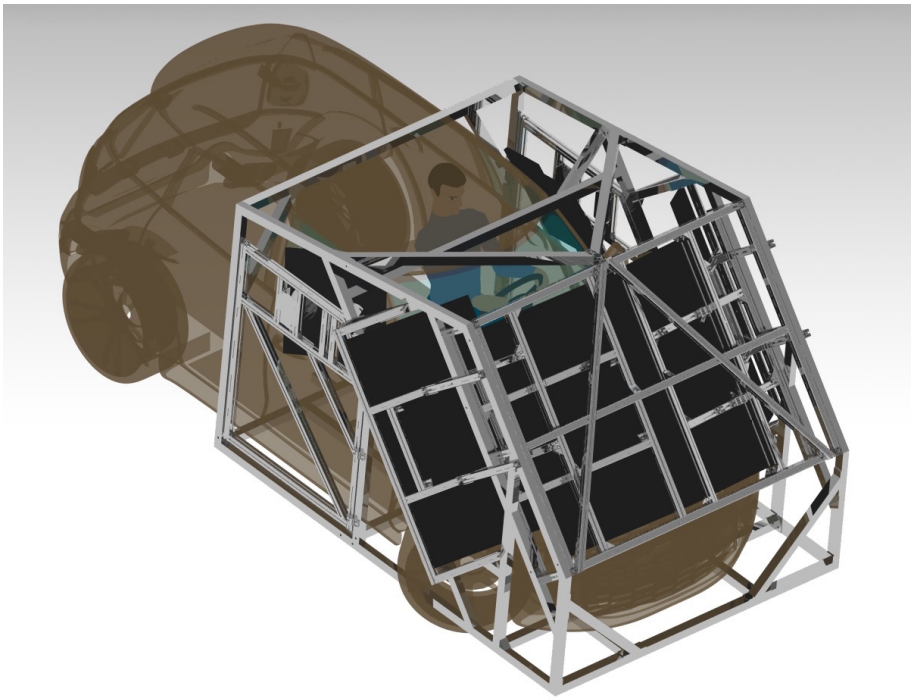

Abbildung 2.2: Aufbau des Simulators mit einem Mini Countryman

Countryman verbunden. In Hinblick auf den Einsatz auf einer Bewegungsplattform wurde das Gestell bis kurz hinter die B-Säule verlängert. Durch diese Maßnahme ist es in Zukunft möglich, das Auto an dieser Position abzutrennen und durch Diagonalstreben entsprechend auszusteifen. Dies hätte eine wesentliche Gewichtsreduzierung zur Folge, wodurch der Einsatz kompakterer und damit billigere Bewegungsplattformen ermöglicht wird.

Der Motorraum wurde so weit wie möglich ausgeräumt, um ausreichend Platz für die für den Simulationsbetrieb notwendigen Komponenten zu schaffen. Konkret wurden folgende Bestandteile im Motorraum verbaut:

- Bremsaktuaktor samt Ansteuerungselektronik
- 12V und 24V Netzteile
- Externe Soundkarte samt Endstufen für die Niederfrequenztöner
- Niederfrequenztöner welche in den Fußraum zeigen
- Analog IO-Device
- CAN-Interface

	Bezeichnung
Betriebssystem	Microsoft Windows 7 Enterprise Edition
Motherboard	ASUSTeK H87-PRO
CPU	Intel Core i7-4771 @ 3.5GHz
Arbeitsspeicher	16GB Kingston KHX1600C10D3/8GX 1.6GHz
Grafikkarte	Intel HD Graphics 4600
Festplatte 1	250GB Samsung SSD 840
Festplatte 2	2TB WDC WD2002FYPS-0

Tabelle 2.1: Konfiguration Simulationsrechner

2.1.1 Simulation

Wie zuvor bereits erwähnt laufen sowohl die Realtime-Simulation-Tasks als auch die Akustik auf einem einzigen Rechner. Zusätzlich sind an diesen Computer die externe Soundkarte als auch Analog-IO sowie CAN-Interface über USB verbunden. Aufgrund der Vielzahl an unterschiedlichen Aufgaben muss der Rechner stärker dimensioniert sein um ein deterministisches Verhalten der Simulation gewährleisten zu können.
Tabelle 2.1 listet die verwendeten Komponenten auf. Die Erfahrung hat gezeigt, dass ein Rechner vollkommen ausreicht um die geforderten Aufgaben zu bewältigen. Es konnte zu keinem Zeitpunkt eine kritische Rechenauslastung beobachtet werden.

Als kritisch hat sich hingegen die Anbindung der externen Komponenten über USB erwiesen. Nachdem sich die Computer außerhalb des Fahrzeuges in einem Serverschrank befinden, müssen relativ lange USB-Kabel verwendet werden um die Bestandteile welche im Motorraum verbaut sind an den Rechner anzuschließen. Es hat sich gezeigt, dass, obwohl laut Spezifikation erlaubt, 5m lange Kabel keinen problemlosen Betrieb gewährleisten können. Es waren stets unvorhersehbare Verbindungsabbrüche zu beobachten. Auch der Einsatz von aktiven Hubs zur Signalaufbereitung brachte keine Besserung.
Schlussendlich konnte dieses Problem nur durch den Einsatz von hochwertigen, 3m langen Kabeln mit zusätzlicher Schirmung gelöst werden. Gespräche mit Mitarbeitern des Projektpartners AVL List GmbH haben gezeigt, dass diese mit denselben Probleme zu kämpfen hatten. Sie konnten die Probleme durch den Einsatz von USB auf Ethernet-Adaptern lösen. Aufgrund der zusätzlichen Latenzen ist dies jedoch nur als suboptimale Lösung zu betrachten, wenn auch als einzige, wenn die Entfernungen es erfordern und keine anderen Schnittstellen zur Verfügung stehen.

2.1.2 Aktuatorik

Bremspedal

Abbildung 2.3 illustriert noch Detail wie die Kommunikation zwischen Simulation und Bremsaktuator aufgebaut ist.
Die Simulation benötigt als Eingang die aktuelle Bremspedalposition sowie Bremspedalgeschwindigkeit, und liefert als Ausgang, als Funktion der beiden Eingangsgrößen, ein

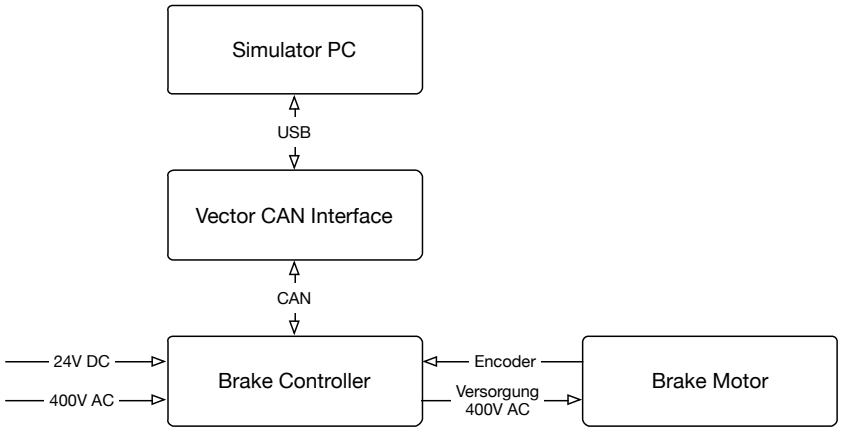

Abbildung 2.3: Topologie Bremsaktuator

gefordertes Bremsmoment.
Motor sowie Controller stammen von der Firma Infranor. Beim Motor handelt es sich um einen XtraforsPrime FP-3314 mit einem maximalen Drehmoment von 189Nm. Er zeichnet sich vor allem durch

- ein großes Drehmoment
- minimalstes Rastmoment durch ein nutenloses Statorkonzept
- einen guten Wirkungsgrad

aus. Zusätzlich ist ein Winkelencoder mit einer Auflösung von 4096 Punkten pro Umdrehung angebracht.
Beim Controller handelt es sich um einen XtrapulsCD1 vom Typ CD1-n-400/70. Dieser verfügt über CANopen sowie RS232 Schnittstellen und wird mit 24V DC als auch 400V AC, zur Bereitstellung der benötigten Energie des Motors, versorgt. Der maximale Strom des Controllers darf für eine Sekunde 7.2A betragen. Der erlaubte Dauerstrom ist mit 3.6A spezifiziert.
Es hat sich gezeigt, dass man beim Einbau des Controllers Sorgfalt bezüglich der elektromagnetischen Verträglichkeit walten lassen muss. Insbesondere in der Soundanlage konnten Störungen bei aktivem Controller beobachtet werden.
Die Geschwindigkeitsinformationen welche vom Controller geliefert werden haben sich leider als nicht brauchbar herausgestellt. Die Daten welche über den CAN-Bus angefordert werden können wiesen eine viel zu schlechte Auflösung auf. In weiterer Folge wurde deshalb die Geschwindigkeitsinformation aus dem Positionssignal mit Hilfe eines Kalman-Filters ermittelt.

Lenkrad

Beim FFD (Force Feedback Device) des Lenkrads handelt es sich um ein noch in Entwicklung befindliches System der Firma SBW Technology. Aus Geheimhaltungsgründen sind leider nicht viele Details zu dieser Komponente bekannt.

Grundsätzlich besteht das System aus einem relativ kleinen, bürstenlosen Gleichspannungsmotor, welcher über ein System von Kugelumlaufspindeln verhältnismäßig hoch übersetzt wird. Ein Mitgrund für die Auswahl dieses Systems ist die benötigte Versorgungsspannung von nur 12V. Dies ermöglicht theoretisch den Einsatz als Steer-By-Wire System in echten Fahrzeugen. Der Simulator wird hierbei verwendet, um Erfahrungen auf diesem Gebiet zu gewinnen.

Die Kommunikation erweist sich in der verbauten Generation leider als umständlich. Während man die Lenkradposition über den CAN-Bus erhält, erfolgt die Lenkmomentenanforderung durch Vorgabe einer analogen Spannung zwischen 0V und 10V. 5V entspricht hierbei einem Moment von 0Nm. Die Übermittlung der Momentenanfoderung über den CAN-Bus wäre wesentlich eleganter und mit weniger zusätzlichem Aufwand verbunden. Abbildung 2.5 verdeutlicht noch einmal die Zusammenhänge zwischen den einzelnen Komponenten.

Das maximale Moment welches das FFD liefern kann liegt bei 12.5Nm bei 7A Stromaufnahme. Leider lässt die installierte Software zum momentanen Stand nur Momente zu, welche das Lenkrad zurück in Neutralposition bringen. Momente welche das Lenkrad von der Neutralposition weg bewegen, werden nicht unterstützt.
Abbildung 2.4 zeigt den linearen Zusammenhang zwischen der analogen Eingangsspannung welche von der Simulation an das Lenkrad gesendet wird und das daraus resultierende Lenkmoment, beziehungsweise die zum Moment gehörende Stromaufnahme des Aktuators.

Im Zuge verschiedener Tests während der Inbetriebnahme hat sich gezeigt, dass der verbaute Prototyp noch nicht den Ansprüchen moderner Fahrsimulatoren genügt. Um genaue Daten diesbezüglich zu erhalten wurden Messungen durchgeführt, welche in einer parallel durchgeführten Masterarbeit [4] im Detail erläutert sind.

2.1.3 Eye-Tracker

Zur Detektion der Augenposition bzw. der Blickrichtung kommt eine Lösung der Firma Smart Eye AB mit dem Namen Smart Eye Pro zum Einsatz. Das System besteht aus 2 bis 8 Kameras, pro Kamera einem Infrarot-Flash sowie dem dazugehörigen Computer samt Software, wie in Abbildung 13.1 zu sehen.
Aus Kostengründen kommt im Fahrsimulator ein System mit nur 2 Kameras zum Einsatz, was den Bewegungsspielraum des Kopfes leicht einschränkt. Während die Detektion der Augenpositionen bei gerader Fahrt relativ zuverlässig funktioniert, genügen 2 Kameras nicht um bei stärkerer Drehung des Kopfes, wie es beispielsweise bei Abbiegevorgängen vorkommen kann, die Positionen der Augen fehlerfrei zu detektieren. Eine Erweiterung durch eine dritte Kamera würde neben einem vergrößerten Arbeitsbereich auch die Genauigkeit erhöhen.

Während Toleranzen der Positionen von etwa 1mm und der Blickrichtungen von 0.5° angegeben wurden, konnten im realen Fahrbetrieb Abweichungen von bis zu 1cm gemessen werden. Die geringen Toleranzen sind jedoch notwendig, um eine akkurate Berechnung

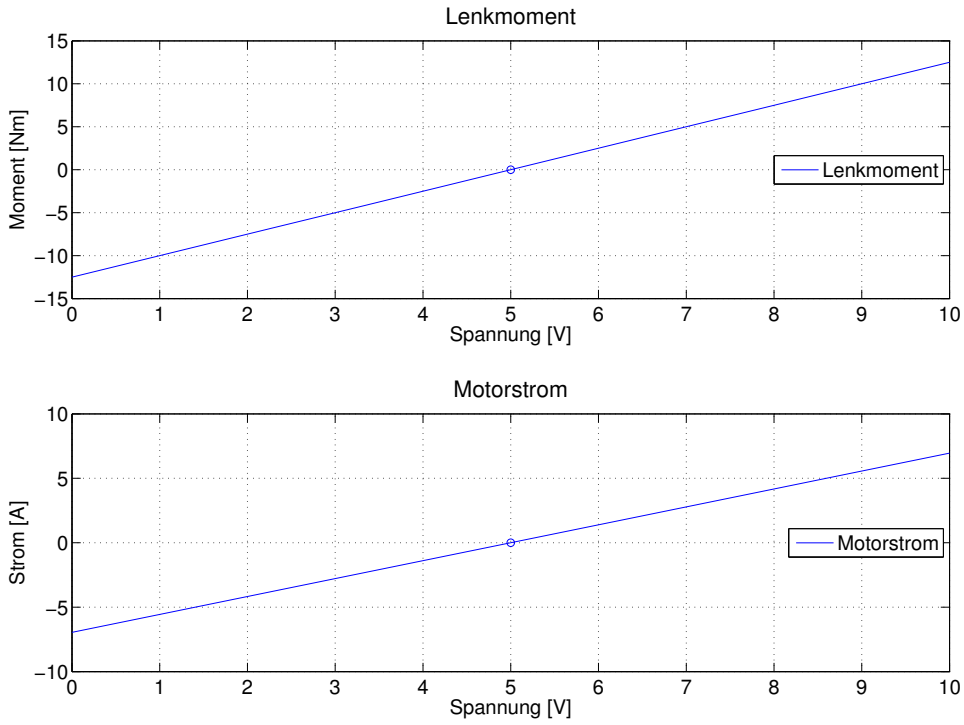

Abbildung 2.4: Lenkmoment- sowie Stromverlauf

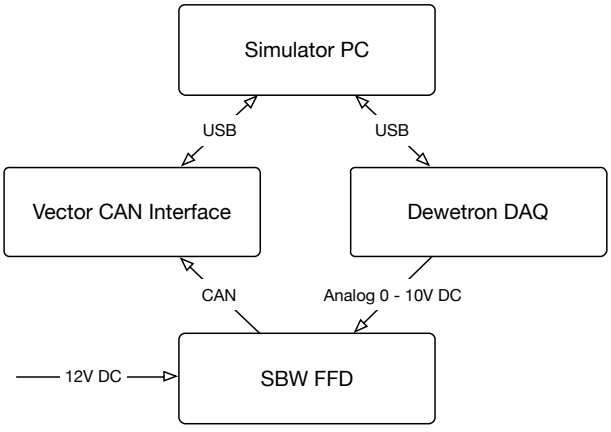

Abbildung 2.5: Topologie Lenkrad

	Bezeichnung
Computer	Dell Precision 1700
Betriebssystem	Microsoft Windows 7 Professional 32Bit
Motherboard	Unknown
CPU	Intel Xeon E3-1245 v3 @ 3.4GHz
Arbeitsspeicher	4GB Hynix/Hyundai HMT325U7EFR8A-PB
Grafikkarte	Intel HD Graphics P4600/P4700
Festplatte 1	120GB Samsung SSD 840 Pro
Festplatte 2	250GB LITEONIT LCS-256

Tabelle 2.2: Konfiguration EyeTracker PC

der 3D-Visualisierung durchführen zu können. Nachdem die Blickrichtungen zur Berechnung der 3D-Shader nicht benötigt werden, wurden auch keine detaillierten Messungen diesbezüglich durchgeführt.

Das verbaute System weist eine Sampling-Rate von 60Hz auf, wobei durch Erweiterung der Software ein Upgrade auf 120Hz erfolgen kann. Die Live-Daten können über TCP, UDP oder CAN übertragen werden. Es steht aber auch die Möglichkeit des reinen Loggings in ein Textfile zur Verfügung. Neben der Augenpositionen und Blickrichtungen liegen noch eine Vielzahl von weiteren Daten vor. Dies umfasst beispielsweise die Position und Orientierung des Kopfes, Daten über die Liedschlag-Häufigkeit sowie Liedschlag-Geschwindigkeit, Details über Sakkaden sowie Pupillen-Durchmesser und vieles mehr.
Das System bietet außerdem die Möglichkeit das Koordinatensystem, auf das sich die vom Eye-Tracker gelieferten Daten beziehen, praktisch beliebig zu definieren. Durch Platzieren eines Schachbrett-Musters mit genau bekannten Abmessungen kann man diesen Ursprung festlegen. Sofern sie die Kamera-Positionen und Orientierungen nicht ändern, muss diese Kalibrierung nur einmalig durchgeführt werden.

2.1.4 Akustik

Zur Akustiksimulation wird die board-interne Audio-Anlage verwendet, welche jedoch um 4 zusätzliche Niederfrequenztöner, auch Shaker genannt, erweitert wurde.
Bei der der Originalausstattung handelt es sich um ein System von Harman Kardon®. Im Detail sind

- 4 x 26mm Hochtöner
- 4 x 100mm Mitteltöner, sowie
- 2 x 217mm Tieftöner

Bestandteil des Harman Kardon® Soundsystems.
Im Fußraum sowie unter den vorderen Sitzen wurde zusätzlich 4 Lautsprecher zur Verbesserung der Niederfrequenz-Charakteristik verbaut.

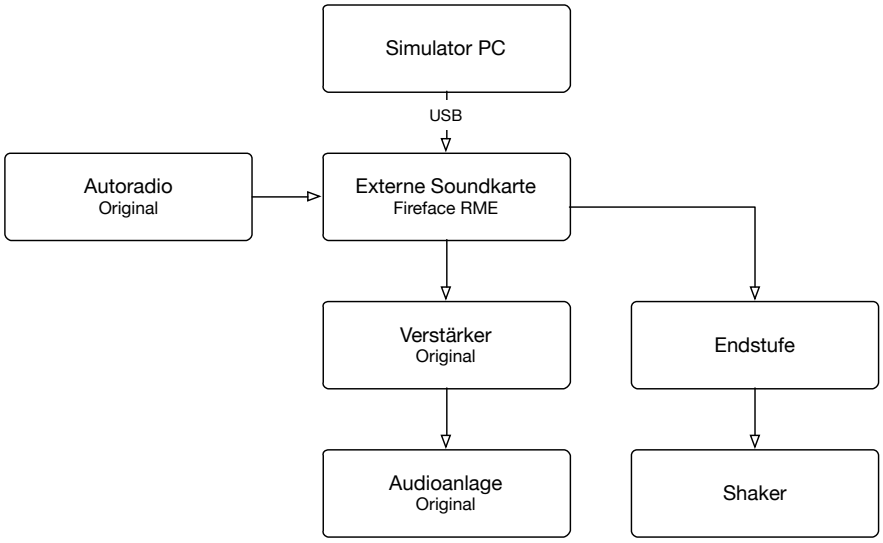

Abbildung 2.6: Topologie Soundsystem

Soundkarte: Als Soundkarte kommt ein Produkt Names Fireface UCX der Firma RME zum Einsatz. Diese kann man über USB als auch Firewire anschließen und bringt eine Vielzahl an Anschlüssen, siehe Tabelle 2.3, mit. Die mitgelieferte Software ermöglicht ein Justieren der verschiedensten Parameter praktisch nach Belieben.

Das Audiosignal, welches von der Akustiksimulation, siehe Kapitel 11, erzeugt wird, wird mit Hilfe der externen Soundkarte mit dem Signal des Autoradios vermischt. Die Lautstärke des Autoradios kann weiterhin vom Fahrer verändert werden, ohne Einfluss auf den Rest der Akustik-Simulation zu nehmen.
Die Soundkarte stellt zwei Ausgangssignale zur Verfügung. Einmal für das originale Soundsystem, welches hauptsächlich für den Mittel- und Hochtonbereich Verwendung findet. Und einmal für die zusätzlich verbauten Shaker, welche hauptsächlich zur Abbildung der niederen Frequenzen, und damit auch für die Generierung von Vibrationen, zuständig sind. Abbildung 2.6 illustriert den Aufbau noch einmal anschaulich.

2.1.5 Visualisierung

Abbildungen 2.2, 2.7 und 2.8 zeigen wie die Monitore zur Visualisierung der Umwelt um das Fahrzeug angebracht worden sind. Insgesamt finden vier 55 Zoll Monitore im Bereich der Frontscheibe sowie vier 23 Zoll Monitore im Bereich der Seitenverglasung Anwendung. Es wurden Exemplare ausgewählt, welche speziell für die Erstellung von Video-Walls entwickelt wurden. Diese weisen einen extra dünnen Rand auf was störende Effekte beim Übergang zwischen benachbarten Bildschirmen vermindert. Die genauen Bezeichnungen sind Tabelle 2.4 zu entnehmen.

Die Auflösung beträgt bei allen Monitoren 1920x1080 Pixel und der Anschluss an die Grafikkarten erfolgt über 10m DVI-Kabel. Wie auch schon bei den USB-Kabeln hat sich

Anzahl	Bezeichnung
8	Analog IO
2	Mic Preamp (digital steuerbar)
2	Instrument Preamp (digital steuerbar)
1	SPDIF IO koaxial
1	ADAT IO (oder 1x SPDIF IO optisch)
1	Wordclock IO
2	MIDI IO
1	Firewire 400
1	USB 2.0

Tabelle 2.3: Anschlüsse Soundkarte

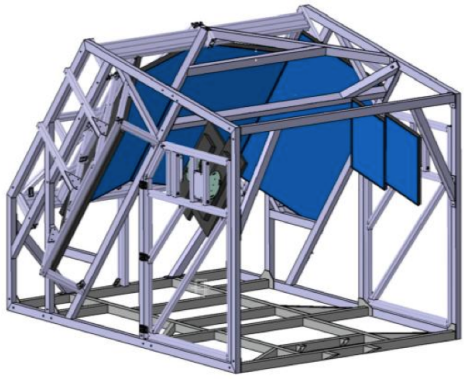

Abbildung 2.7: Aufbau Monitoranordnung

Abbildung 2.8: Aufbau Parallax Barriere

auch bei der Monitorverkabelung gezeigt, dass nicht alle nach den gleichen Qualitätskriterien zu erhalten sind. Bei billigeren Produkten konnten bei dieser Länge Ausfälle beobachtet werden.

Aufgrund des leicht überhängenden Einbau der Frontmonitore war zu beobachten, dass wegen der großen Bildschirmdiagonale eine Durchhängung der Bildschirmfläche von mehreren Millimetern auftritt. Da der Rahmen der Displays seitlich abgestützt ist, handelt es sich um ein Durchhängen der Displayfolie an sich. Dies hat maßgeblichen Einfluss auf den 3D-Effekt, da der Abstand der Parallax-Barriere, siehe nachfolgendes Kapitel 2.1.5, zur Bildschirmfläche genau bekannt sein muss um das Bild, passend zu den Augenpositionen,

Hersteller	Größe	Auflösung	Bezeichnung	Bereich
Samsung	55 Zoll	1920 x 1080	LH55UEC	Frontscheibe
Samsung	23 Zoll	1920 x 1080	MD230	Seitenscheibe

Tabelle 2.4: Konfiguration Monitore

	Bezeichnung
Betriebssystem	Microsoft Windows 7 Enterprise Edition 64Bit
Motherboard	ASUSTeK Gryphon Z87
CPU	Intel Core i7-4770 @ 3.4GHz
Arbeitsspeicher	16GB Kingston KHX1600C10D3/8GX 1.33GHz
Grafikkarte 1	NVIDIA GeForce GTX 780 4GB
Festplatte 1	WDC WD10EFRX-68JCSN0

Tabelle 2.5: Konfiguration Visualisierungsrechner

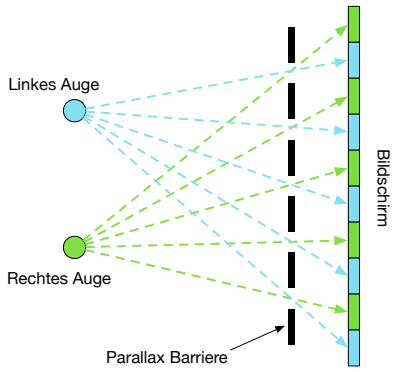

Abbildung 2.9: Funktionsweise Parallax Barriere

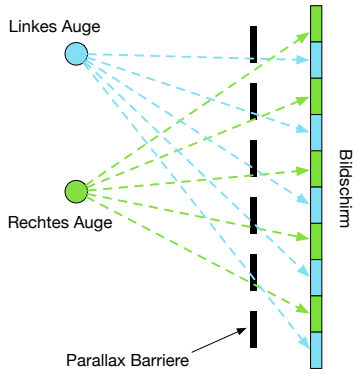

Abbildung 2.10: Funktionsweise Parallax Barriere; Neue Augenpositionen ohne Korrektur

korrekt berechnen zu können.
Weiters ist davon auszugehen, dass durch Erwärmung der Komponenten dieses Phänomen noch verstärkt wird und somit über die Zeit ein nicht konstantes Verhalten aufweist.

Parallaxbarrierre

Zur Generierung eines 3D-Effekts besteht neben der weit verbreiteten Anwendung von Polarisations- und Shutterbrillen die Möglichkeit, durch Anbringen einer so genannten Parallaxbarriere auch ohne Einsatz störender Brillen einen entsprechenden stereoskopischen Effekt zu vermitteln. Durch ein vor dem Bildschirm angebrachtes Muster kann man erreichen, dass beide Augen ein unterschiedliches Bild zu sehen bekommen. Dies setzt voraus, dass alle Geometrien in der Visualisierungskette genauestens bekannt sind. Ansonsten ist es nicht möglich, das auf dem Bildschirm darzustellende Bild entsprechend der Augenpositionen zu berechnen.

Abbildung 2.9 zeigt beispielhaft, wie mit Hilfe eines Streifenmusters eine Parallaxbarriere realisiert werden kann. Durch geeignete Ansteuerung des Displays können für beide Augen voneinander unabhängige Bilder dargestellt werden.
Verändern sich nun die Augenpositionen, die Parallaxbarriere sowie die Bilder jedoch

nicht, so tritt der Fall ähnlich wie in Abbildung 2.10 dargestellt, ein. Es kann keine Separierung der beiden Bilder erfolgen und der Betrachter sieht dementsprechend ein komplett widersprüchliches Bild. Es wird klar, dass entweder die Darstellung, oder aber die Barriere permanent an die Augenpositionen angepasst werden muss, um eine Separierung zu ermöglichen.
Wesentlich ist ebenfalls, dass es immer auch einen Bereich geben wird, welcher von beiden Augen eingesehen werden kann. Hier darf keine Darstellung erfolgen, und die Pixel müssen entsprechend auf schwarz oder einen Mittelwert der benachbarten Pixel geschaltet werden.

Im gegebenen Fall besteht die Parallaxbarriere aus einer 2cm dicken Plexiglasplatte, dessen dem Betrachter zugewandte Seite von der Display-Oberfläche einen Abstand von 5cm aufweist. Das Pattern wurde im Siebdruckverfahren aufgebracht, wobei als Farbton ein neutrales Grau gewählt wurde. Leider konnte der Lieferant trotz Zusicherung die benötigten Toleranzen des Drucks nur bedingt einhalten, was leider einen negativen Einfluss auf die erzielbaren Ergebnisse hatte. Abbildung 2.8 zeigt, wie die Platten in die Gesamtkonstruktion implementiert wurden.
Als Muster wurde ein Streifen-Pattern gewählt, dessen Linienbreiten- sowie Abstände als auch Orientierung einer Optimierung unterworfen wurden. Ziel war es, für eine gegebene Kopfposition mit Norm-Augenabstand eine größtmögliche Separierung der Pixel zu erreichen.

Nachdem es nicht möglich ist den subjektiven Eindruck der Barrieren in die Optimierung mit einfließen zu lassen, hat sich im Endeffekt das Muster der beiden linken Parallaxbarrieren als nur bedingt geeignet herausgestellt. Währen diese zwar eine gute Separierung ermöglichen, ist die subjektive Wahrnehmung aufgrund des relativ groben Streifenmusters nur bedingt zufriedenstellend. Im Probebetrieb hat sich gezeigt, dass diese eher als störend wahrgenommen wurden.
Aufgrund der aus der Vorstudie gewonnen Daten welche in [5] ausführlich analysiert wurden wurde beschlossen, die Plexiglasscheiben zu demontieren und den Simulator im herkömmlichen 2D-Modus weiter zu betreiben.

2.1.6 IO / CAN

Dewetron DAQ

Zur Ein- und Ausgabe von analogen als auch digitalen Signalen kommt das Datenerfassungsgerät DEWE-50-USB2-16 der Firma Dewetron GmbH zum Einsatz. Durch 16 verfügbare Einschubplätze und Anzahl von dazu passenden Modulen ist das Gerät extrem universell einsetzbar. Die wesentlichen Eigenschaften können Tabelle 2.6 entnommen werden.

Das verwendete Gerät unterstützt mehrere Serien an Einschubmodulen. Anwendung finden die Module DAQN-AIN zur einfachen Analog/Digital-Wandlung, DAQN-V-OUT zur Ausgabe der vom Lenkrad benötigten Analogspannung und das Module DAQP-STG, welches aufgrund der verbauten Vollbrücke zur Einbindung des Kraftsensors, welcher am Bremspedal angebracht ist, verwendet wird.

Feature	DEWE-50-USB2-16
Anzahl Einschübe	16
AD Wandler	
Sampling-Methode	Multiplexed
Auflösung	16Bit
Sampling-Rate	500kS/s insgesamt 31.25kS/s pro Kanal
Zähler und Digital IO	
Zähler / Digital IO	Durch entsprechende Einschübe

Tabelle 2.6: Überblick Dewetron DAQ

Intern verwendet das DEWE-50-USB2-16 einen IC der Firma National Instruments. Um das Device in die Simulation einbinden zu können, wurde das von NI angebotene SDK verwendet um eine Bibliothek zu erstellen, welche die Kommunikation mit der Data Acquisition Unit ermöglicht. Details hierzu können in Kapitel 6 gefunden werden.

Vector CAN

Zur Kommunikation mit CAN-getriebenen Geräten wie beispielsweise dem Lenkrad oder dem Controller des Brems-Aktuators, kommt ein Produkt der Firma Vector Informatik GmbH zum Einsatz. Das Device mit der genauen Bezeichnung VN1640 lässt sich über USB an unter Microsoft Windows betriebenen PCs anschließen und ermöglicht relativ einfach die Kommunikation mit externen BUS-Teilnehmern. Der Vollständigkeit halber werden die wichtigsten Features in Tabelle 2.7 aufgelistet.
Matlab/Simulink unterstützt ab R2012a unter Zuhilfenahme der Vehicle Network Toolbox das CAN-Interface native, was eine leichte Integration des CAN-Bus in die Gesamtsimulation ermöglicht. Nähere Details hierzu können Kapitel 3 entnommen werden.

Sollte man nicht in der Lage sein die Vehicle Network Toolbox verwenden zu können, so bietet Vector Informatik GmbH mit der XL-Drive-Library eine Programmierschnittstelle an mit der es möglich ist eigene Applikationen, ohne weitere externe Abhängigkeiten, zu entwickeln. Es steht eine .dll samt Header für C++ als auch ein zugehöriger .net Wrapper zur Verfügung. Beispielanwendungen inklusive Source Code sowie entsprechende Dokumentation der API können ebenfalls über die Vector Homepage bezogen werden.
Nachdem im Fahrsimulator die Vehicle Network Toolbok Verwendung findet, wird an dieser Stelle nicht näher auf die Implementierung dieser Bibliothek eingegangen.

2.2 Netzwerk

In Abbildung 2.11 ist die Netzwerk-Topologie des Fahrsimulators dargestellt. Ein WLAN-Router fungiert als Schnittstelle in das TUG-Netz, womit es unter anderem möglich ist, sich über eine Remote-Verbindung mit dem Simulatornetzwerk zu verbinden. Aufgrund der Wireless-Funktionalität ist dieser auch Knotenpunkt für alle Tablets, sei es nun zur

Feature	VN1640
Anzahl Kanäle	4
Unterstützte Protokolle	0 - 4 CAN 0 - 4 LIN 0 - 2 K-Line 0 - 4 J1708
Anzahl Stecker	4
SteckerTyp	D-SUB-9
Baudraten	CAN bis 8MBits/s LIN bis 330kBit/s
Durchschnittl. Reaktionszeit	$250\mu s$
Baudraten	CAN bis 8MBits/s LIN bis 330kBit/s
Error frame	Erkennung und Generierung
Betriebssystem	Windows XP (32 bit), Windows 7/8 (32 und 64 bit)

Tabelle 2.7: Überblick Vector VN1640

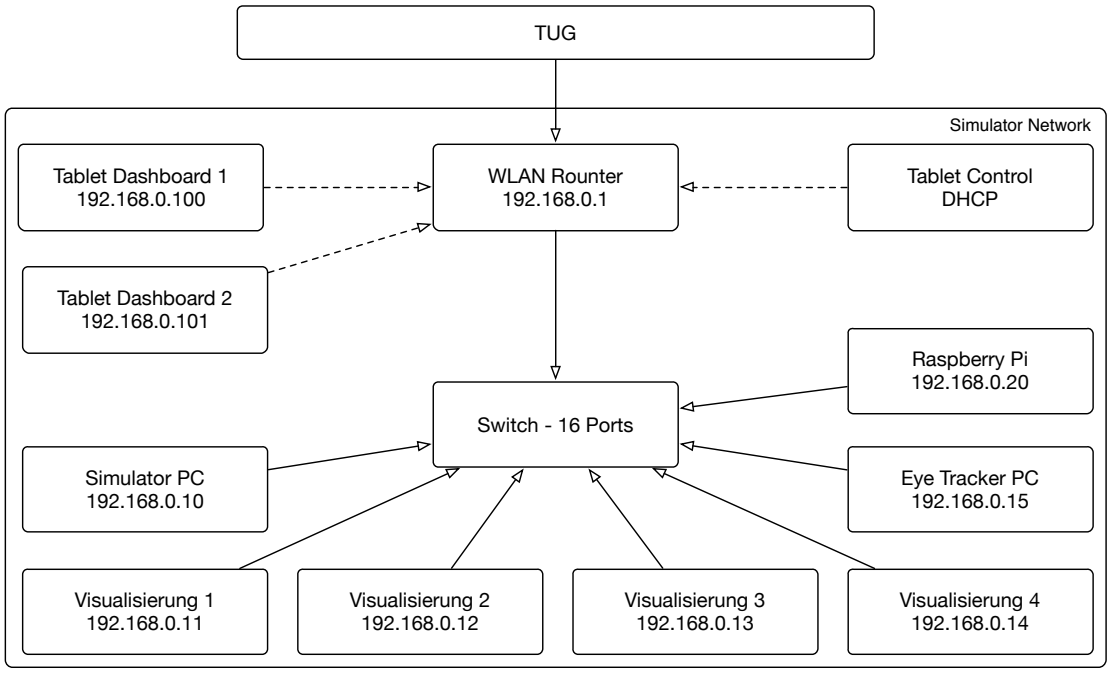

Abbildung 2.11: Überblick Netzwerktopologie

Gerät	Hersteller	Modell
WLAN Router	TP-LINK	TL-WR1043ND
16-Port Switch	ZyXEL	GS1100-16

Tabelle 2.8: Netzwerkgeräte

Darstellung des Dashboards oder zur Kontrolle der Simulation sowie externer Laptops welche sich mit dem Netz verbinden wollen.

Mit dem Router ist ein Switch verbunden, an welchen wiederum alle Computer über CAT6-Netzwerkkabel angeschlossen sind. Mit Ausnahme des Raspberry Pi verfügen alle über Kabel angebundenen Rechner über 1GBit-Netzwerkinterfaces.
In Tabelle 2.8 sind abschließend die genauen Typenbezeichnungen von Router sowie Switch angeführt.

2.3 Software

2.3.1 Versionierung

Bei Softwareprojekten ist es heutzutage State of the Art mit Versionsverwaltungssoftware wie beispielsweise Git, SVN, Mercurial, TFS oder ähnlichen zu arbeiten. Eines der Ziele ist, jegliche Änderungen und Fortschritte welche erzielt werden, genau zu dokumentieren um jeden beliebigen Stand der Software zu jedem beliebigen Zeitpunkt wiederherstellen zu können.
Während diese Art des Arbeitens aus der Softwareentwicklung entsprungen ist, bieten in der Zwischenzeit bereits viele andere Tools, wie beispielsweise Matlab, Unterstützung für solche Systeme. So ist es auch hier möglich, Änderungen an Simulink-Modellen genauestens nachvollziehen zu können.

Für den Fahrsimulator wurde als Versionsverwaltung Git eingesetzt, welches seinerzeit eigens für die Entwicklung des Linux-Kernels entwickelt wurde. Neben den lokalen Kopien wird das Repository zentral auf einem Server gespeichert, um einerseits stets eine Sicherung der Daten garantieren zu können und andererseits, um anderen Projektmitarbeitern stets Zugriff zu allen Dateien gewährleisten zu können. Als Anbieter wurde Bitbucket gewählt, da hier 2GB Speicherplatz kostenfrei zur Verfügung gestellt werden.
Nach Installation eines von unzähligen frei erhältlichen Git-Clients kann das gesamte Repository unter der Adresse https://ftg_fasi@bitbucket.org/fasi_tug/ftg_fasi.git geladen werden, womit der Letztstand auf den lokalen Rechner kopiert wird.

2.3.2 List of Software Revision

Bei einem so umfangreichen Projekt wie dem Fahrsimulator ist es unvermeidlich, dass eine Vielzahl an Programmen, SDKs und Bibliotheken zur Anwendung kommt. Tabelle 2.1 listet alle wichtigen Komponenten samt Versionsnummer auf, welche zum Zeitpunkt der Erstellung dieser Arbeit Verwendung gefunden haben.

	Version	Verwendung
PTWinSim	1.25.0.0	Verwaltung der Echtzeit-Simulations-Tasks
Matlab/Simulink	R2012b	Erstellung der Simulationsmodelle
Simulink Coder	R2012b	Codegenerierung von Simulink-Modellen
Vehicle Network Toolbox	R2012b	Kommunikation mit dem Vector CAN Interface
@Source Toolbox	V4.2	Erstellung der PTWinSim Targets
CreateEngineSound	N.v.	Tool zur Akustiksimulation
Pd-extended	0.43-4	Verwendet von CreateEngineSound
Instant Reality Framework	2.3.0	Darstellung der Visualisierung
Microsoft Visual Studio	V2013	Erstellung von Bibliotheken (*ScenarioLib*, *DaqLib*, ...) Erstellung der Echtzeitmodelle Erstellung der Tablet-Control-Software
Microsoft Visual Studio	V2008	Erstellung der InstantIO Plugins
Android Studio	1.2	Erstellung der HMIs
Java SE Development Kit	1.7.0_76	Erstellung der HMIs
AVL VSM	3.10.138	Fahrdynamik
IPG CarMaker PRO	4.0.3	Fahrdynamik
Vector CAN Driver	8.9.22	Ansprechen des Vector CAN Interfaces
NI USB-6251 Driver/SDK	NIDAQ980f0	Ansprechen des Dewetron DAQ Devices
RME Fireface USB Driver	1.0.46.0	Ansprechen und Konfiguration der externen Soundkarte
Smart Eye Pro	V6.1.2(4275)	Auswertung der Augenpositionen
C++ Boost Libraries	1.55.0	Verwendung in den Simulations-Modulen
Qt Libraries	5.2	C++ Framework für GUI-Applikationen
Qwt-Library	6.1.2	Widgets für Qt
DK3 Driver	2.8.0.11	USB Dongle Podium Technology

Tabelle 2.9: List of Software Revision

Kapitel 3

Echtzeitumgebung

In komplexeren Systemen ist der Einsatz von Echtzeitumgebungen von substantieller Bedeutung. So wird beispielsweise beim Standard-Reglerentwurf von zeitdiskreten Systemen, welche beim Einsatz von digitalen Rechnern aufgrund der Abtastung der analogen Signale automatisch vorliegen, von konstanten Zeitintervallen ausgegangen, in denen die Berechnung der jeweiligen Ausgangsgrößen zu erfolgen hat.
Werden die Update-Intervalle nicht eingehalten, so kann die Stabilität des Regelkreises nicht länger garantiert werden. In Hinblick auf Fahrdynamik-Regelungen kann im Worst-Case eine Verletzung der Echtzeitbedingungen fatale folgen haben.

Im Fahrsimulator verhält es sich ähnlich wie im echten Fahrzeug. Verschiedene Prozesse müssen miteinander kommunizieren und zu gegebenen Zeitpunkten verschiedenste Berechnungen durchführen. Zwar hat im Falle der Simulation eine Verletzung der Deadlines keine unmittelbaren Folgen, jedoch ist eine deterministische Abarbeitung der Kalkulationen unumgänglich um reproduzierbare Ergebnisse zu erzielen.
Liegen zum Zeitpunkt einer neuen Berechnung aufgrund der fehlenden Synchronisation die benötigten Daten noch nicht wie geplant vor, so werden alte Datensätze herangezogen, was schlussendlich zu falschen Ergebnissen führen kann.

Abbildungen 3.1 und 3.2 zeigen das Beispiel der Berechnung von Auto-Positionen mit identer Geschwindigkeit innerhalb der Simulation sowie die Ausgabe selbiger in der Visualisierung.
Liegt eine Abarbeitung der Tasks nach dem festgesetzten Schema vor, so ist in jedem sichtbaren Frame die korrekte Autoposition wahrnehmbar. Wird jedoch beispielsweise das zweite Fahrzeug unabhängig vom ersten als auch der Visualisierung berechnet, so kann es passieren, dass die Positionen beginnen zu divergieren. Im zweiten Frame liegen die neuen Positionsdaten des zweiten Fahrzeuges noch nicht vor, weshalb der alte Standort an die Visualisierung gesendet wird.
Bis zum dritten Frame jedoch wurde die Berechnung der Position einmal zu oft durchgeführt, was einen Versatz in die andere Richtung als zuvor hervorruft.
Insgesamt wird man sprunghafte Änderungen des Abstandes der beiden Fahrzeuge wahrnehmen können, obwohl sich diese mit der identen Geschwindigkeit bewegen.
Selbiges gilt natürlich auch für eventuell simulierte Sensoren. Auch diese würden dann

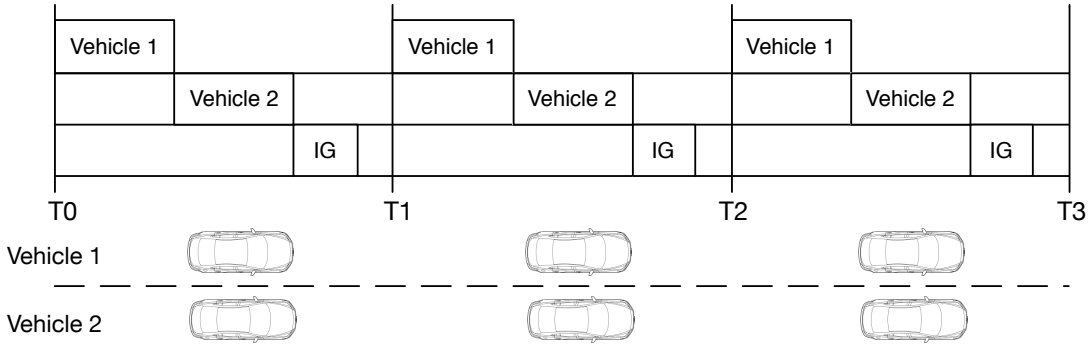

Abbildung 3.1: Deterministische Abarbeitung der Prozesse

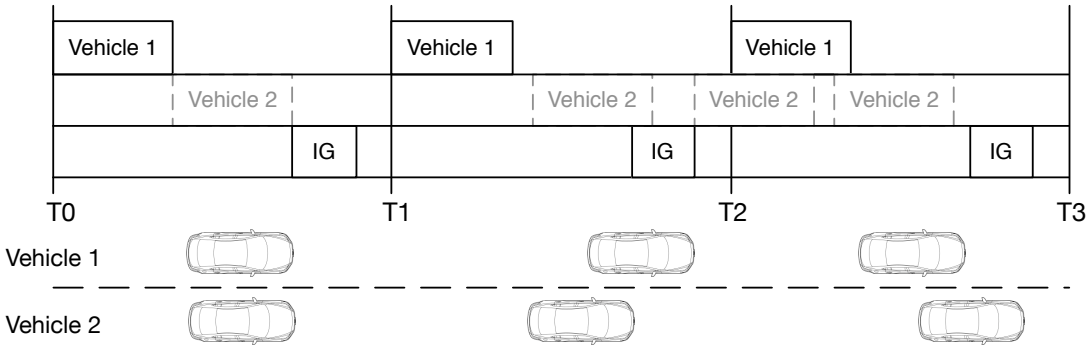

Abbildung 3.2: Nicht-deterministische Abarbeitung der Prozesse

dieses Verhalten sehen und entsprechende Reaktionen in den darunterliegenden Regelalgorithmen hervorrufen.
In einem so vielfältigen System wie einem Fahrsimulator, bei dem Hardware-IO, Visualisierung, Sound und verschiedenste Simulationsprozesse aufeinandertreffen, ist es ein leichtes, eine fast schiere Anzahl von Beispielen zu finden wo es zu ungewollten Phänomen aufgrund von nicht deterministisch arbeitenden Prozessen kommen kann.

3.1 PTWinSim

Im Fahrsimulator kommt zur Sicherstellung deterministischer Arbeitsabläufe ein Framework der Firma Podium Technology Ltd. [6] zum Einsatz. Hierbei handelt es sich einerseits um die @Source Toolbox für Matlab/Simulink, andererseits um die Soft-Echtzeitumgebung Namens PTWinSim welche die aus Simulink-Koppelplänen erstellten Binaries ausführt. Den Ursprung hat diese Arbeitsumgebung im Rennsport, im speziellen bei der Entwicklung von (Motor-)Steuergeräten, und wird aktuell von einer Vielzahl von Rennteams eingesetzt. Dies lässt bereits im Vorfeld einen hohen Grad an Ausgereiftheit dieses Software-Produkts erahnen.
Weiters bietet der modellbasierte Ansatz basierend auf Matlab/Simulink den Vorteil, dass

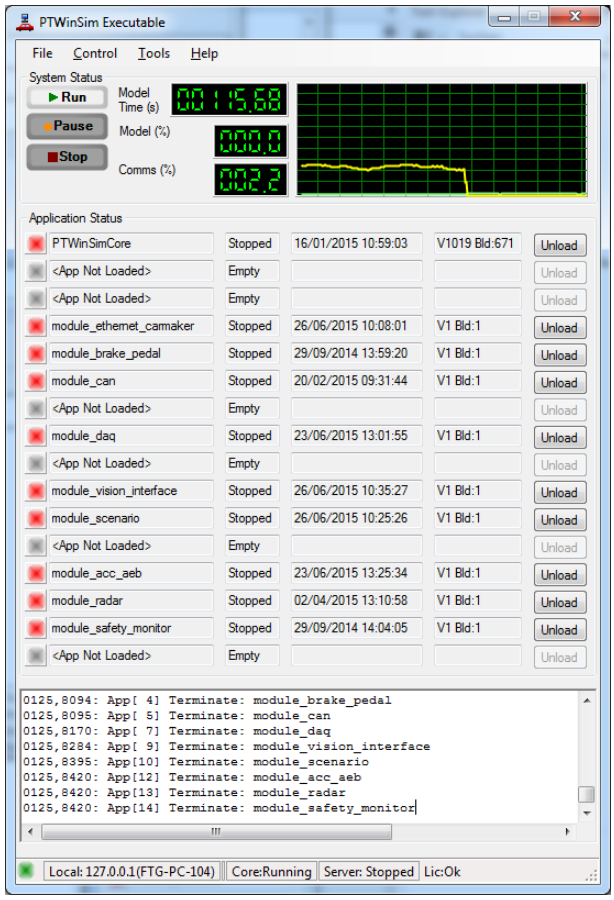

Abbildung 3.3: Applikation zur Ausführung der Echtzeit-Tasks

auch Ingenieure ohne programmiertechnischen Background sehr leicht ihre Module in die Gesamtsimulation integrieren können. Die Einbindung bestehender Modelle kann aufgrund der hohen Flexibilität des Frameworks und je nach Komplexität mitunter innerhalb von wenigen Minuten erfolgen.
Ein weiterer nicht zu unterschätzender Vorteil ist, dass die Modelle unter Zuhilfenahme aller (Analyse-)Funktionalitäten welche Matlab/Simulink bietet, bereits im Vorhinein offline getestet werden können, noch bevor diese tatsächlich in den Fahrsimulator integriert werden. Dadurch können mehrere Entwickler gleichzeitig an verschiedenen Modulen arbeiten, ohne auf die beschränkte Ressource Fahrsimulator warten zu müssen.

Da es sich bei Microsoft Windows um kein echtzeitfähiges Betriebssystem handelt, kann durch den Einsatz dieser speziellen Laufzeitumgebung nichtsdestotrotz ein hoher Grad an Determinismus erreicht werden.
Die genaue Implementierung konnte vom Hersteller aus Geheimhaltungsgründen nicht eruiert werden. Ein typischer Ansatz ist jedoch die Reservierung von vielen Systemressourcen, von denen jedoch nur ein geringer Prozentsatz tatsächlich für die Berechnung der Simulationsmodelle eingesetzt wird. Dadurch kann trotz Unterbrechungen durch das Betriebssystems in den meisten Fällen eine Einhaltung der Deadlines erreicht werden.

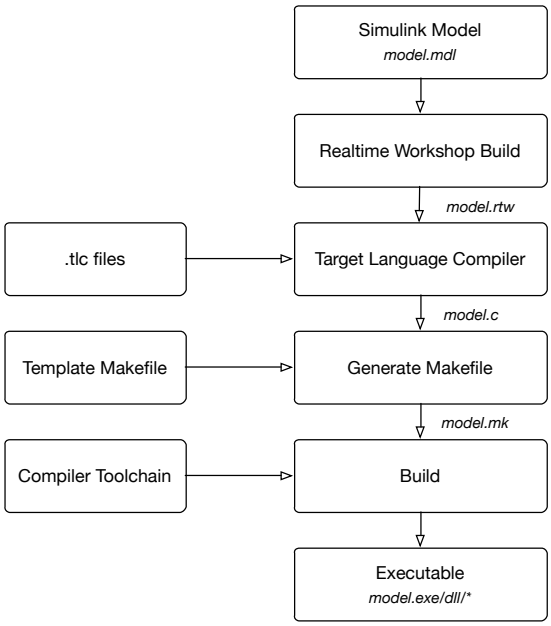

Abbildung 3.4: Codegenerierung mit Matlab/Simulink

Untersuchungen durch Ausgabe von Logmeldungen bei Verletzungen der Deadlines haben ergeben, dass PTWinSim tatsächlich einen sehr hohen Grad an Determinismus zur Verfügung stellen kann. Trotz einer Vielzahl an zusätzlichen Programmen welche parallel zu PTWinSim laufen, unter anderem *pd extended* zur Soundgenerierung, welches ebenfalls sehr hohe Anforderungen an einen deterministischen Ablauf stellt, konnten praktisch keine Verletzungen der Deadlines beobachtet werden.

3.2 Code-Generierung

Wie bereits in 3.1 erwähnt, werden die Tasks welche innerhalb von PTWinSim ausgeführt werden, aus Simulink-Modellen heraus unter Zuhilfenahme der automatischen Codegenerierung mit Simulink-Coder (vormals Realtime Workshop), erstellt.
Um diesen Prozess besser verstehen zu können, sei kurz auf die generelle Vorgangsweise zur Codegenerierung unter Matlab/Simulink eingegangen [7].

1. **Modell-Interpretation:** Im ersten Schritt wird, ähnlich wie es die Common Intermediate Language in der .Net Umgebung darstellt, ein Zwischen-Modell aus dem Simulink-Koppelplan heraus erstellt. Theoretisch kann diese Struktur, welche eine hierarchische Abbildung des Modells enthält, auch anderweitig erstellt werden, praktisch ist zum gegebenen Zeitpunkt aber keine weitere Softwarelösung neben Matlab/Simulink bekannt, welche in der Lage ist, .rtw Dateien zu erzeugen. Dieser Zwischenschritt ist notwendig da sich dieses Format für den weiteren Prozess wesentlich besser eignet als das Simulink-Modell an sich.

2. **Code-Generierung:** Der Target Language Compiler (TLC) generiert aus den .rtw

Dateien unter Zuhilfenahme von .tlc Files C oder C++ Dateien sowie die zugehörigen Header. Die .tlc Dateien spezifizieren wie aus der Zwischen-Repräsentation des Modells der entsprechende Sourcecode erzeugt werden soll. Je nach Zielplattform oder Anforderungen werden entsprechend angepasste Target-Files verwendet.

3. **Makefile-Generierung:** Basierend auf einem Template-Makefile, welches wiederum an die jeweilige Zielplattform angepasst ist, wird ein Makefile erstellt welches im weiteren Prozess während des Build-Vorgangs Verwendung finden wird.

4. **Kompilierung:** Anhand des zuvor erzeugen Sourcecodes sowie des Makefiles wird nun unter Verwendung eines unterstützten Compilers das entsprechende Executable erzeugt. Je nach Betriebssystem unterstützt Simulink Coder prinzipiell eine Vielzahl von Compilern, unter Windows jedoch ist man, sofern mal alle Features nutzen möchte, auf den Einsatz der Microsoft-Compiler beschränkt.
   Bei dem erzeugten Binary kann es sich je nach Target-Auswahl um eine Standalone-Applikation, eine Standard-Bibliothek oder sonstige proprietäre Formate handeln. Die Target-Spezifikationen in den vorhergehenden Schritten definieren das Endprodukt.

Das Framework, welches Podium Technology Ltd. zur Verfügung stellt, bringt die entsprechenden .tlc Dateien als auch Template-Makefiles zur Generierung der PTWinSim-Executables bereits mit. Dadurch kann ein entsprechender PTWinSim-Task innerhalb kürzester Zeit erstellt werden. Bei Auswahl des entsprechenden .tlc files wird automatisch das richtige Template Makefile ausgewählt. Abbildung 3.5 veranschaulicht dies noch einmal.
Es ist ebenfalls möglich über die .tlc Dateien zusätzlich Parameter zu konfigurieren, welche dann direkt über die Einstellungen des Simulink-Modells gesetzt werden können. So ist es, wie in Abbildung 3.6 veranschaulicht, beispielsweise möglich einen *Application Slot* zu definieren, in welchen dann das kompilierte Modell in PTWinSim geladen wird. Zum momentanen Zeitpunkt ist es möglich 15 Modelle in die Echtzeitumgebung zu laden.
Weitere wichtige Parameter sind zum Beispiel die Optimierungs-Level des Compilers, bzw. wenn mehrere Compiler vorhanden sind die Auswahl, welcher überhaupt Verwendung finden soll.

3.3 Einbindung eigener Bibliotheken

Während die Code-Generierung für Simulink-Modelle welche nur Standard-Blöcke bzw. Blöcke aus der @Source Bibliothek verwenden ohne zusätzliche Arbeitsschritte reibungsfrei funktioniert, so hat man jedoch für die Einbindung eigener C/C++ Bibliotheken noch etwas zusätzlichen Aufwand zu betreiben.
Für den Fahrsimulator wurden einige C++ Bibliotheken entwickelt welche in die Matlab/-Simulink Umgebung integriert wurden. Im Detail wären das

- **ScenarioLib:** Zur Generierung von umher fahrenden Verkehr
- **DaqLib:** Interface zur Data Acquisition Unit der Firma Dewetron für Analog Input/Output

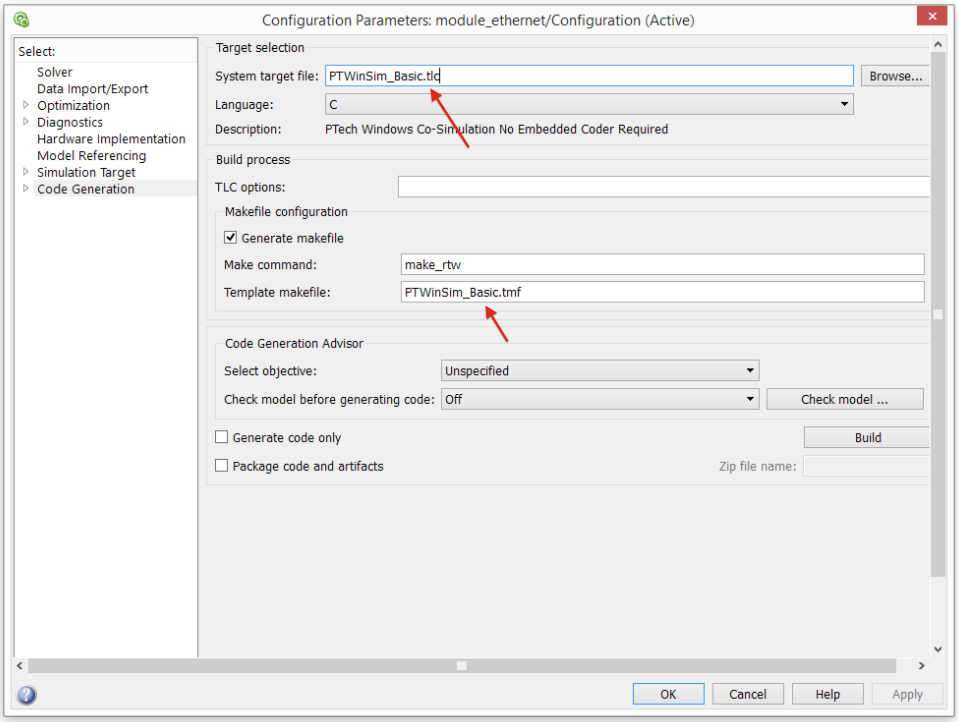

Abbildung 3.5: Auswahl des PTWinSim TLC Files

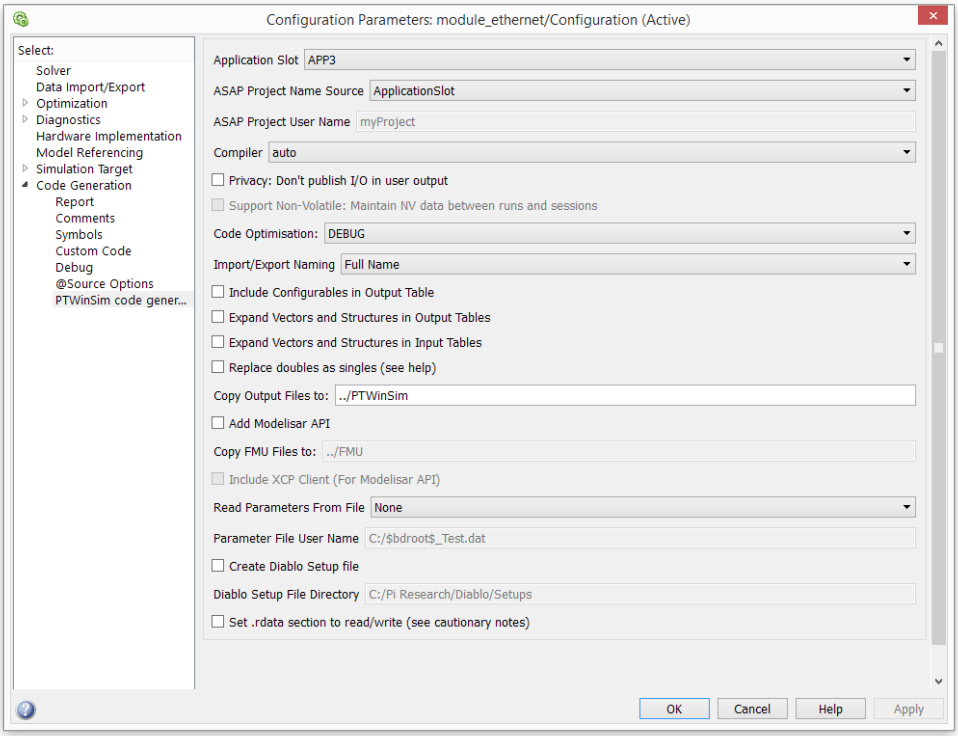

Abbildung 3.6: Target-spezifische Parameter

- **VisionInterfaceLib:** Interface zur Visualisierung über UDP-Pakete
- **SmartEyeLib:** Interface zum Eye-Tracker zum Zwecke des Loggings der Augenpositionen bzw. Blickrichtungen über UDP-Pakete
- **LogLib:** Bibliothek zum Loggen von gewünschten Signalen zum Zwecke der Nachbearbeitung

Matlab/Simulink bietet zwei Möglichkeiten an eigene Bibliotheken zu integrieren. Einerseits über C/C++ s-functions, andererseits über den Aufruf von Bibliotheken welche über ein Standard C-Interface verfügen aus Matlab s-functions heraus.

C/C++ s-functions sind im Prinzip normale Bibliotheken mit den Matlab-eigenen Endungen .mexw32 oder .mexw64, welche jedoch über ein genau definiertes Interface zur Interaktion mit der Simulink-Engine verfügen müssen. Die Erstellung dieser s-functions ist verhältnismäßig komplex, da eine große Anzahl von Abhängigkeiten zu Matlab-spezifischen Header-Files sowie Bibliotheken gegeben ist.
Ein großer Vorteil ist die nahtlose Integration in Matlab/Simulink und die Unterstützung von Inputs/Outputs variabler Länge. Ist bereits eine Standard C-Bibliothek vorhanden und möchte man trotzdem nicht auf die Vorzüge der s-functions verzichten, so besteht natürlich die Möglichkeit die s-function als eine Art Wrapper zu verwenden. Dies führt jedoch zu einer weiteren Schicht im Abhängigkeitsbaum der unter Umständen vermieden werden kann.

Matlab/Simulink bietet, wie zuvor bereits erwähnt, auch die Möglichkeit Bibliotheken mit Standard-C Interface direkt anzusprechen. Dies hat einerseits den Nachteil, dass nicht alle Features welche Matlab/Simulink zur Verfügung stellt benutzt werden können, andererseits jedoch erspart man sich unter Umständen das Schreiben des zusätzlichen s-function Wrappers. Je nach Konfiguration der Entwicklungsumgebung kann ein weiterer Vorteil die leichtere Testbarkeit der jeweiligen Software-Module sein, da man nicht auf eine funktionierende Matlab/Simulink Installation angewiesen ist.
Der große Nachteil bei dieser Herangehensweise ist, dass aufgrund des fehlenden, genau definierten Interfaces welches die s-functions beinhalten, Matlab/Simulink nicht die Charakteristiken der Inputs und Outputs eruieren kann. Aus diesem Grund sind zusätzliche Matlab-Dateien zur genauen Definition der Funktionsaufrufe, die Matlab s-functions, notwendig. Auch müssen für jede Funktion extra .tlc Files erstellt werden, um die Function-Calls, im, aus dem Simulink-Modell heraus erstellen Code, richtig zu implementieren.

Nachdem die *ScenarioLib* auch noch von anderen Applikationen neben dem Fahrsimulator verwendet wird und in allen Bibliotheken nur einfache Datentypen mit Matlab/Simulink ausgetauscht werden, wurde der Zugang eines reinen C-Interfaces, ohne Verwendung von s-functions, verwendet. Tatsächlich hat sich gezeigt, dass durch diese Wahl zu keinem Zeitpunkt eine Einschränkung der Funktionalität gegeben war. Nachfolgend soll nun anhand des Beispiels der Data Acquisition Library gezeigt werden, wie solch ein Interface aussehen kann.
Wie in Listing 3.1 zu sehen ist, handelt es sich hierbei um eine sehr einfache Bibliothek. Neben den Init und Stop Aufrufen stehen lediglich Methoden zum Setzen und Lesen von

Analogwerten der Data Acquisition Unit zur Verfügung. Weiters wird das Einlesen von Werten eines Arduino-Boards unterstützt, was im momentanen Stadium jedoch keine Verwendung mehr findet.
Die genaue Implementierung welche im Hintergrund für die Kommunikation mit dem Dewetron-Device zuständig ist, ist für die folgenden Betrachtungen irrelevant. Es sei nur soviel erwähnt, dass die DAQ-Unit auf einem Chipset der Firma National Instruments basiert und deshalb auch das SDK von National Instruments verwendet werden konnte.

```
1  #pragma once
2
3  #ifdef __cplusplus
4  extern "C" {
5  #endif
6
7  // Get analog voltage from specified channel
8  __declspec(dllexport) double getValue(unsigned int channel);
9  // Get value from arduino device
10 __declspec(dllexport) double getSerialValue(void);
11 // Initialize library
12 __declspec(dllexport) void init(void);
13 // Set analog voltage on specified channel
14 __declspec(dllexport) void setValue(double newValue);
15 // Stop communication with daq unit
16 __declspec(dllexport) void stop(void);
17
18 #ifdef __cplusplus
19 }
20 #endif
```

Listing 3.1: DaqLib.h

Erstellt man nun die entsprechende .dll, so kann man diese beispielsweise aus einem Matlab-Script heraus wie folgt verwenden.

```
1 loadlibrary('/path/to/libDir/DaqLib', '/path/to/headerDir/daqlib.h');
2 calllib('DaqLib', 'init');
3 calllib('DaqLib', 'getValue', 0);
4 calllib('DaqLib', 'stop');
5 unloadlibrary('DaqLib');
```

Listing 3.2: Verwendung der *DaqLib* in Matlab

`loadlibrary` in Zeile 1 lädt die .dll, wobei das erste Argument den Pfad zur .dll ohne Dateiendung und das zweite Argument die zugehörige Header-Datei angibt. Zeilen 2 bis 4 rufen die entsprechenden Funktionen `init`, `getValue` mit der entsprechenden Kanalnummer sowie `stop` auf. In Zeile 5 wird die Bibliothek wieder freigegeben.
Die absolute Pfadangabe für Bibliothek als auch Header-File kann vermieden werden, wenn man diese zu den Matlab-Such-Pfaden hinzufügt.

Möchte man die Bibliothek nun auch aus Simulink heraus ansprechen, so muss für jeden Funktionsaufruf eine Matlab s-function mit zugehörigem .m file erstellt werden, welches den Simulink-Block genau spezifiziert. Für die `getValue` Funktionalität sei dies exemplarisch in Listing 3.3 angeführt. Abbildung 3.7 zeigt den zu Listung 3.3 entsprechenden Simulink-Block samt Parameter.
Auf nähere Erläuterungen wird an dieser Stelle verzichtet, da aufgrund der Kommentare die einzelnen Passagen ausreichend genau beschrieben sein sollten.

```
function DaqLibGet(block)
    setup(block);
end

function setup(block)
    % Specify a single output
    block.NumInputPorts = 0;
    block.NumOutputPorts = 1;

    % Set up port properties to be dynamic
    block.SetPreCompInpPortInfoToDynamic;
    block.SetPreCompOutPortInfoToDynamic;

    % Set input to a scalar int16
    block.OutputPort(1).Dimensions = 1;
    block.OutputPort(1).DatatypeID = 5; % 5 means int16
    block.OutputPort(1).Complexity = 'Real';
    block.OutputPort(1).SamplingMode = 'sample';

    % Set 2 parameters for specifying the sample time and channel number
    block.NumDialogPrms = 2;
    block.DialogPrmsTunable = {'Nontunable', 'Nontunable'};

    % Set discrete sample time from first parameter
    block.SampleTimes = [block.DialogPrm(1).Data, 0];

    % Specify same sim state as a built-in block
    block.SimStateCompliance = 'DefaultSimState';

    %Specify the implemented callbacks
    block.RegBlockMethod('WriteRTW', @WriteRTW); % Called when RTW build is triggered
    block.RegBlockMethod('Start', @Start);
    block.RegBlockMethod('Outputs', @Outputs); % Required
    block.RegBlockMethod('Terminate', @Terminate); % Required
end

function WriteRTW(block)
% Save the parameters to an RTW file for code generation.
% Later used in .tlc file
    channel = sprintf('%f', block.DialogPrm(2).Data);
    block.WriteRTWParam('string', 'Channel', channel);
end
```

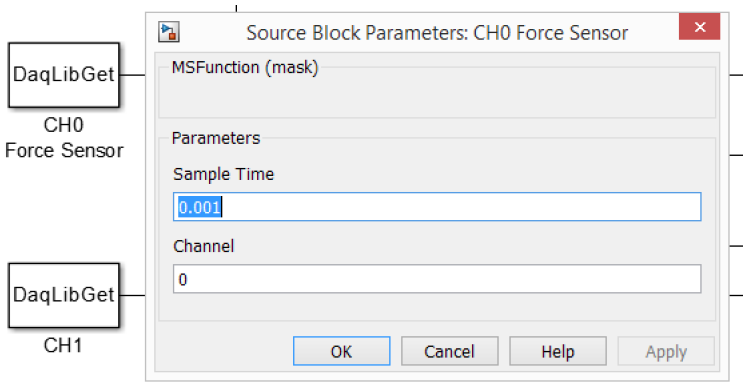

Abbildung 3.7: *DaqLib* - getValue Simulink-Block

```
43
44 % Called at start - not used here.
45 function Start(block)
46 end
47
48 % Call the function during every timestep.
49 function Outputs(block)
50     % Call corresponding library
51     % block.DialogPrm(2).Data represents the Channel
52     block.OutputPort(1).Data = double(calllib('DaqLib', 'getValue',
53         uint16(block.DialogPrm(2).Data)));
54 end
55
56 % Called when simulation is over
57 function Terminate(block)
58 end
```

Listing 3.3: Spezifikation des *DaqLibGet* Simulink-Blocks

Möchte man nun aus dem Simulink-Modell mit einer eigens eingebundenen Bibliothek mit Hilfe des Simulink Coder/Realtime Workshop ein Binary erzeugen, so ist für jeden Simulink-Block ein .tlc File zu erstellen welches beschreibt, wie der entsprechende C/C++ Code erstellt werden soll. Listing 3.4 zeigt den Inhalt solch einer Datei für den bereits zuvor behandelten *DaqLibGet* Block.

```
1 %implements DaqLibGet "C"
2
3 %function BlockTypeSetup(block, system) void
4     %<LibAddToCommonIncludes("daqlib.h")>
5 %endfunction
6
7 %function Outputs(block, system) Output
8   /* %<Type> Block: %<Name> */
9   %assign y = LibBlockOutputSignal(0, "", "", 0)
```

```
10    %assign ch = ParamSettings.Channel
11    %<y> = getValue((unsigned int)%<ch>);
12  %endfunction
```

Listing 3.4: .tlc File zum Zwecke der Codegenerierung

Das `<LibAddToCommonIncludes("daqlib.h")>` Kommando teilt dem Target Language Compiler mit, eine entsprechende include Anweisung zum daqlib.h Header im generierten Source Code hinzuzufügen.
Zeile 9 weist der lokalen Variablen y den ersten Ausgang des Simulink-Blocks zu.
In Listing 3.3 haben wir gesehen, dass im WriteRTW Callback der Channel-Parameter in den RTW-Parametern des Simulink-Blocks abgespeichert wird. Genau auf diese Variable können wir nun im .tlc File wieder in den ParamSettings in Zeile 10 zugreifen.
Zeile 11 definiert schließlich wie der genaue Aufruf im erzeugen C/C++ Code aussehen soll.
Im angegebenen Beispiel wird folgender C++ Code vom Target Language Compiler erzeugt.

```
1  /* M-S-Function Block: <Root>/CH0 Force Sensor */
2  rtb_CH0 = getValue((unsigned int)0);
```

Listing 3.5: Generierter C++ Code

Jedes Simulink-Modell welches eigens geschriebene Libraries verwendet, verfügt zusätzlich über einen Init sowie Terminate Block. Wie der Name schon erwarten lässt, werden diese Aufrufe nur einmalig zu Beginn sowie Beendigung der Ausführung aufgerufen.
Typische Anwendungen wären zum Beispiel

- Öffnen und Schließen von Netzwerkverbindungen
- Initialisieren und Beenden von externem Hardware-Zugriff
- Einlesen von Konfigurationsfiles
- Sonstiger Init- sowie Terminate-Code welche zur korrekten Ausführung sowie sauberem Shutdown notwendig ist

3.3.1 Matlab Workspace

In Abschnitt 3.3 wurde bereits erwähnt, dass Matlab/Simulink die Header-Dateien sowie die Bibliotheken an sich finden können muss. Um die Entwicklung unter verschiedenen Arbeitsumgebungen mit unterschiedlichen Ordnerstrukturen zu erleichtern, wurde ein Matlab-Init File erstellt, welches als Einziges vom jeweiligen Entwickler an die eigenen lokalen Anforderungen angepasst werden muss. Im Grunde definiert dieses nur, wo die Header und Binaries zu finden sind.

```
%% Change paths here
daqLibDir = '/path/to/libDir/DaqLib';
daqLibIncludeDir = '/path/to/headerDir/DaqLib';

visionLibDir = '/path/to/libDir/VisionInterfaceLib';
visionLibIncludeDir = '/path/to/headerDir/VisionInterfaceLib';

scenarioLibDir = '/path/to/libDir/ScenarioLib';
scenarioLibIncludeDir = '/path/to/headerDir/ScenarioLib';

logLibDir = '/path/to/libDir/LogLib';
logLibIncludeDir = '/path/to/headerDir/LogLib';

smartEyeLibDir = '/path/to/libDir/SmartEyeLib';
smartEyeLibIncludeDir = '/path/to/headerDir/SmartEyeLib';

%% Do not change
daqLibName = 'DaqLib.lib';
visionLibName = 'VisionInterfaceLib.lib';
scenarioLibName = 'ScenarioLib.lib';
logLibName = 'LogLib.lib';
smartEyeLibName = 'SmartEyeLib.lib';

% Save paths to .mat file
file = 'libPreferences.mat';
save(file, 'daqLibName', 'daqLibDir',
 'daqLibIncludeDir');
save(file, 'visionLibName', 'visionLibDir',
 'visionLibIncludeDir', '-append');
save(file, 'scenarioLibName', 'scenarioLibDir',
 'scenarioLibIncludeDir', '-append');
save(file, 'logLibName', 'logLibDir',
 'logLibIncludeDir', '-append');
save(file, 'smartEyeLibName', 'smartEyeLibDir',
 'smartEyeLibIncludeDir', '-append');

% Add paths to matlab workspace
addpath(daqLibIncludeDir)
addpath(daqLibDir)
addpath(visionLibIncludeDir)
addpath(visionLibDir)
addpath(scenarioLibIncludeDir)
addpath(scenarioLibDir)
addpath(logLibIncludeDir)
addpath(logLibDir)
addpath(smartEyeLibIncludeDir)
addpath(smartEyeLibDir)
```

Listing 3.6: Init File zur Initialisierung des Matlab-Workspace

Die soeben nach Listing 3.6 erstellte .mat Datei kann nun in einem Build-Script zur einfachen Code-Generierung wiederverwendet werden.

```
function build_daq()
    % Load paths
    load('libPreferences.mat');
    % Define simulink model to be build
    module = 'module_daq';
    % Open specified model
    load_system(module);
    % Retrieve current configuration set of the model
    cs = getActiveConfigSet(module);
    % Add header path to configuration
    cs.set_param('CustomInclude', daqLibIncludeDir)
    % Add library path to configuration
    cs.set_param('CustomLibrary', strcat(daqLibDir, daqLibName))
    % Build the model
    rtwbuild module_daq
end
```

Listing 3.7: Code-Generierung mit Hilfe einer Matlab-Funktion

Listing 3.7 zeigt, wie auf einfache Weise mit Hilfe der gespeicherten Pfade ein im Prinzip beliebiges Simulink-Modell aus Matlab heraus kompiliert werden kann. Die repräsentativ angegebene Funktion ist Teil eines übergeordneten Build-Scripts wo durch Angabe des Simulink-Modells die entsprechende Build-Funktion aufgerufen wird.

3.4 Module

In PTWinSim ist es, wie bereits erwähnt, möglich, mehrere voneinander unabhängige Tasks zu definieren, welche jedoch untereinander Daten austauschen können. Um einen möglichst großen Grad an Flexibilität zu erhalten und die Entwicklung neuer Simulationskomponenten zu erleichtern, wurden verschiedenste Module definiert, welche für sich jeweils eigenständige Tasks innerhalb der Echtzeitumgebung repräsentieren. Diese wären zum Zeitpunkt der Erstellung dieser Arbeit wie folgt:

- Modul VSM
- Modul VSM / Carmaker Interface
- Modul ACC / AEB
- Modul CAN
- Modul DAQ
- Modul Ethernet
- Modul Logging
- Modul Radar
- Modul Scenario
- Modul Vision Interface
- Modul Brake Pedal
- Modul Safety Monitor

3.4.1 Modul VSM

AVL VSM^{TM} stellt die Fahrdynamiksimulation der Firma AVL List GmbH dar, welche im Rahmen des MueGen-Projekts für den Fahrsimulator zur Verfügung gestellt wird.
VSM ist aufgrund der Komplexität sowie der hohen Update-Rate von 2kHz die mit Abstand rechenintensivste, jedoch auch wichtigste Komponente des Fahrsimulators. Nähere Informationen können Kapitel 4 entnommen werden.

3.4.2 Modul VSM/CarMaker-Interface

Nachdem die Realtime-Binary von VSM aus Geheimhaltungsgründen immer nur in Zusammenarbeit mit der AVL List GmbH erstellt werden kann und eine Austauschbarkeit von VSM durch eine andere Fahrzeugsimulation leicht bewerkstelligbar sein soll, wurde ein separates Interface-Modul implementiert. Dieses kann einerseits direkt mit VSM, oder aber beispielsweise auch über UDP mit IPG CarMaker oder sonstiger Simulations-Software kommunizieren. Das Interface stellt all jene Daten bereit, welche von den nachfolgenden Modulen aus der Fahrdynamik benötigt werden.
Durch diese Konstellation muss bei Verwendung einer neuen Fahrzeugsimulation nur das Interface-Modul adaptiert werden, ohne die anderen Module aufgrund des gleichbleibenden Interfaces anpassen zu müssen. Wie zuvor sind auch hier weitere Informationen in Kapitel 4 zu finden.
Die Update-Rate des Tasks liegt bei 100Hz, um die Daten zeitsynchron mit der ACC-Regelung zur Verfügung stellen zu können.

3.4.3 Modul ACC/AEB

Der Abstandstempomat sowie der Notfallbremsassistent wurden ebenfalls in einem separaten Modell untergebracht. Der Regler zur adaptiven Abstandsregelung besteht aus einem Beschleunigungsregler, dem eine Distanz- sowie Geschwindigkeitsregelung überlagert ist. Als Stellgröße des Beschleunigungsreglers dient das Gas- sowie Bremspedal der Fahrdynamiksimulation, während die äußeren Reglerstrukturen eine Längsbeschleunigung als Sollvorgabe für den Beschleunigungsregler ausgeben.
Parallel berechnet der automatische Notfallbremsassistent ständig aus der Distanz d sowie Relativgeschwindigkeit v_{rel} zum jeweils nächsten Fahrzeug die Time To Collision (TTC). Diese ergibt sich ganz simpel aus

$$TTC = \frac{d}{v_{Rel}} \tag{3.1}$$

Für den AEB sind unterschiedliche Schwellwerte definiert, wobei zwischen Überland- sowie Stadtbetrieb unterschieden wird. In Tabelle 3.1 sind die Werte und die dazugehörigen Aktionen aufgelistet.

Während der Geschwindigkeitsregler aufgrund der integrierenden Regelstrecke als reiner Proportionalregler aufgebaut ist, findet für den Beschleunigungsregler ein PID-Regler mit Anti-Windup-Maßnahmen Verwendung. Für den Distanz-Controller wird ein Reglerkonzept nach Abbildung 3.9 verwendet, wobei hierbei τ einen Tuningparameter darstellt, mit dem das Verhalten wesentlich beeinflusst werden kann. Während diese einfache Struktur nur das Konzept verdeutlicht, wurden für den Fahrsimulator mehrere Kennlinien sowie

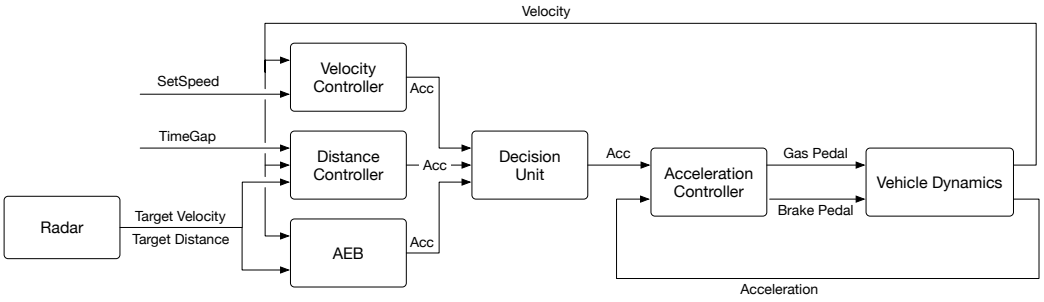

Abbildung 3.8: Reglerstruktur ACC-Controller

TTC	Aktion
Überland	
< 2.6	Warnung
< 1.6	Teilbremsung
< 0.6	Vollbremsung
Stadt	
< 1.5	Warnung
< 1.0	Vollbremsung

Tabelle 3.1: AEB Schwellwerte

Faktoren integriert, mit dem das Verhalten weiter angepasst werden kann. Dies war notwendig, um dem System das Verhalten des echten Referenzfahrzeugs aufprägen zu können. Auch bei diesem Modul findet eine Update-Rate des Modells von 100Hz Anwendung, was aufgrund der verhältnismäßig großen Zeitkonstanten des Fahrzeugmodells ausreichend für ein stabiles Gesamtsystem ist.

Abbildung 3.8 zeigt stark vereinfacht die grundlegende Struktur der Assistenzsysteme sowie die dazugehörigen Größen.
In der tatsächlichen Implementierung besteht ein nicht unerheblicher Teil aus Logik welche dafür zuständig ist, stets die richtigen Beschleunigungen aufzubringen beziehungsweise die Systeme beim Auftreten unterschiedlichster Events zu deaktivieren.

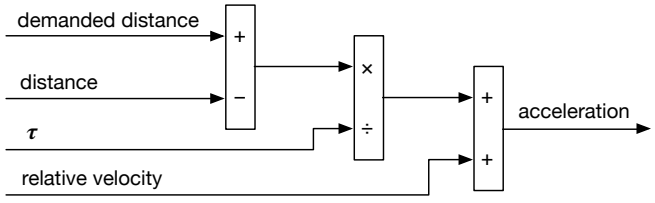

Abbildung 3.9: Reglerstruktur Distanz-Controller

Die Umsetzung mit Hilfe vorangehend abgebildeter Kaskadenstruktur bringt zwei wesentliche Vorteile mit sich:

- Die Grenzen der Beschleunigungen oder Verzögerungen können beliebig gesetzt werden, was einen Komfort- sowie Sicherheitsgewinn nach sich zieht.
- Es muss nur der innere Regelkreis an das jeweilige Fahrzeug angepasst werden. Ist dieser Regler ausreichend gut parametriert, so verhält sich der Abstandsregeltempomat stets gleich, unabhängig davon, ob das System einen PKW oder LKW regelt.

3.4.4 Modul CAN

Die CAN-Kommunikation mit der CAN-Schnittstelle nach Kapitel 2.1.6 findet, wie bereits erwähnt, unter Zuhilfenahme der Vehicle Network Toolbox statt.
Der Controller des Bremsaktuators benötigt nach erfolgter Initialisierung eine Momentenanforderung und liefert die Position sowie Geschwindigkeit des Bremspedals zurück. Auch der Lenkaktuator stellt der Simulation die momentane Position über CAN zur Verfügung.
Neben der Aktuatorik welche nachträglich in das Fahrzeug integriert wurde, ist ein weiterer Kanal mit dem bestehenden Bord-CAN-Netz des Mini Countryman verbunden, um Schalterpositionen auslesen oder Events triggern zu können.

Jegliche CAN-Kommunikation findet in einem separaten Simulink-Modell und damit auch Echtzeit-Prozess statt, wobei hier eine Update-Rate von 1kHz gewählt wurde.
Jedem gelieferten Positionswert des Brems-Controllers muss ein entsprechender Request vorangehen. Nachdem die Vehicle Network Toolbox die Senderate auf 200Hz beschränkt, ist es somit auch nicht möglich in einem kleineren Intervall als 5ms Daten des Brems-Controllers zu empfangen. Dies ist insbesondere für Regelungen von großer Bedeutung, da beispielsweise bei Notbremsungen im Bremspedal so hohe Dynamiken entstehen, dass je nach implementierten System Instabilitäten entstehen können. Alle CAN-Messages welche zum Betrieb des Lenkrads sowie Bremsaktuators benötigt werden können aus [8] entnommen werden.

3.4.5 Modul DAQ

Jegliche Kommunikation welche mit dem DEWETRON IO-Device aus Kapitel 2.1.6 unter Verwendung der *DaqLib* aus Kapitel 6 bewerkstelligt wird, ist in diesem Modul integriert.
Dies umfasst vor allem das Einlesen der Position des Gangwahlhebels, der Position des Gaspedals, dem Signal der Messbrücke zur Auswertung des Kraftsensors welcher am Bremspedal montiert ist als auch die Ausgabe einer analogen Spannung zur Momentenanforderung des Lenkaktuators. Dieser Prozess arbeitet mit einer Update-Rate von 1kHz, was für eine Vielzahl von Regelsystemen ausreichend Reserven bieten sollte. Bei höheren Datenraten hat sich gezeigt, dass das Dewetron-Device anfängt Daten zu puffern und somit keine echten Realtime-Daten mehr zur Verfügung stellen kann.
Weitere detaillierte Informationen sind Kapitel 6 zu entnehmen.

3.4.6 Modul Ethernet

Jegliche Netzwerk-Kommunikation welche zum Betrieb der Simulation notwendig ist wird in diesem Modul bewerkstelligt. Dies umfasst vor allem folgende Schnittstellen:

- Kommunikation mit dem Debug-Tool aus Kapitel 9
- Kommunikation mit dem Sound-Prozess zur Akustik-Simulation welcher in Kapitel 11 erläutert ist
- Kommunikation mit den Tablets zur Visualisierung des Dashboards aus Kapitel 8
- Kommunikation mit dem Control-Tablet des Operators zur Steuerung der Simulation wie in Kapitel 10 beschrieben

Zum momentanen Zeitpunkt finden aufgrund des geringen Overheads ausschließlich UDP-Pakete Verwendung. Während das TCP-Protokoll im Gegensatz zu UDP die korrekte Übertragung der Nachrichten sicherstellt und Congestion-Control Algorithmen implementiert, so sind UDP-Pakete in der Regel kleiner und aufgrund des nicht vorhandenen Acknowledge auch schneller. Es hat sich gezeigt, dass zum momentanen Zeitpunkt keine Notwendigkeit besteht das bestehenden System auf TCP umzustellen. Es konnten keine merklichen Datenverluste festgestellt werden.

Momentan ist das Modul auf 100Hz konfiguriert, wobei nur das Debug-Interface tatsächlich mit dieser Datenrate sendet. Alle anderen implementierten Interfaces Senden die Daten mit wesentlich geringeren Datenraten. Die Tablets für das Dashboard erhalten die Signale momentan beispielsweise nur mit 10Hz, da für die reine Anzeige keine höheren Datenraten benötigt werden.

3.4.7 Modul Logging

Um die in den Simulator-Fahrversuchen ermittelten Daten auch im Post-Processing entsprechend aufarbeiten zu können, wurde ein Modul exklusiv dem Logging gewidmet. Grund hierfür ist unter anderem, dass Dateizugriffe in Echtzeit-Tasks aufgrund des nicht-deterministischen Verhaltens stets einen kritischen Punkt darstellen.
Es wurde eigens eine C++-Bibliothek erstellt, welche die Logging-Funktionalität zur Verfügung stellt. Aufgrund der relativ geringen Datenmengen konnte, wie Tests gezeigt haben, trotz einer Update-Rate von 100Hz auf Caching-Mechanismen verzichtet werden.

Als Dateiformat kommt ein Comma-Separated-File zur Anwendung, da dieses von einer Vielzahl unterschiedlicher Programme und Bibliotheken verarbeitet werden kann.

3.4.8 Modul Radar

Während im echten Fahrzeug ein Radar, welches in modernen Varianten als FMCW (Frequency Modulated Continuous Wave Radar) oder Dauerstrichradar ausgeführt ist, die für die Assistenzsysteme benötigten Informationen liefert, findet im Fahrsimulator ein

vereinfachtes Radarmodell, basierend auf rein geometrischen Zusammenhängen, Verwendung.

Die Szenarien-Bibliothek aus Kapitel 12 liefert dem Radar-Modell die Positionen der Fahrzeuge als auch eine ID, welche den Fahrzeugtyp wie zum Beispiel LKW oder PKW, beschreibt. Anhand dieser ID werden vier Eckpunkte des jeweiligen Fahrzeugs generiert und für alle weiteren Berechnungen herangezogen.
Als Parameter werden dem Radarmodell die maximale Reichweite als auch der Öffnungswinkel übergeben, welche in der Simulation jedoch geschwindigkeitsabhängig ausgeführt sind. Während im realen Fahrzeug mehrere Radarmodule bzw. Kombinationen mit anderen Sensortechnologien wie beispielsweise Kameras Verwendung finden um den gesamten Geschwindigkeits- als auch Winkelbereich abdecken zu können. reicht es beim verwendeten, vereinfachten Modell aus, nur ein einen Sensor zu verwenden, sofern die Parameter entsprechend adaptiv ausgeführt sind.

Abbildung 3.10 zeigt exemplarisch, wie anhand der vier Eckpunkte eine Abschattung eines dahinter liegenden Fahrzeugs erfolgt und entsprechend vom Radarmodell nicht ausgegeben wird. Die jeweils am äußersten liegenden Eckpunkte spannen demnach jenen Bereich auf, der für alle nachfolgenden Berechnungen einen abgeschatteten Bereich definiert.
In Abbildung 3.11 ist dargestellt, wie die Berechnung der einzelnen Distanzen zu den erkannten Fahrzeugen erfolgt. Zuerst wird der Mittelpunkt der vier Eckpunkte gebildet und eine Linie zwischen eben jenem Punkt sowie dem Ego-Fahrzeug gebildet. Anschließend wird der zum Ego-Fahrzeug nächste Schnittpunkt der soeben gebildeten Linie sowie der Linien welche sich aus den Eckpunkten ergeben herangezogen, um die Distanz zu berechnen.
Liegt der Mittelpunkt der vier Eckpunkte außerhalb des Radar-Kegels wird das Objekt vom Radar-Modell nicht erkannt. Dementsprechend kann es also passieren, dass sich Teile eines Fahrzeugs bereits im Radarkegel befinden, es jedoch trotzdem nicht erkannt wird. Diese Tatsache stellt jedoch kein Problem dar, da auch reale Radarsensoren erst ab einer gewissen Schwelle Fahrzeuge korrekt interpretieren.

Weitere Daten welche vom Radarmodell berechnet werden sind die Relativgeschwindigkeit, der Winkel zwischen Ego- sowie Targetfahrzeug, der Winkel zwischen den Geschwindigkeitsvektoren von Ego- und Targetfahrzeug als auch ein Zieltarget für den Abstandsregeltempomat.
Als Besonderheit sei angemerkt, dass das Radarmodell die Fahrstreifen von allen Targetfahrzeugen als auch vom Ego-Fahrzeug kennt. Aufgrund dieser Tatsache kann es einfach das jeweils nächste Fahrzeug, welches auf dem selben Fahrstreifen fährt, als Zieltarget auswählen. Aufgrund dieser Tatsache sind keine Objektklassifizierungen sowie Zielauswahl-Algorithmen notwendig.

In der Zwischenzeit wurde am Institut ein realeres, auf Raytracing basierendes Radarmodell entwickelt, welches ebenfalls mit Kenntnis von vier Eckpunkten von potentiellen Target-Fahrzeugen arbeitet. Aufgrund der selben Input-Daten sollte es ein leichtes sein das vorhandene Modell durch das mehr der Realität entsprechenden zu ersetzen.

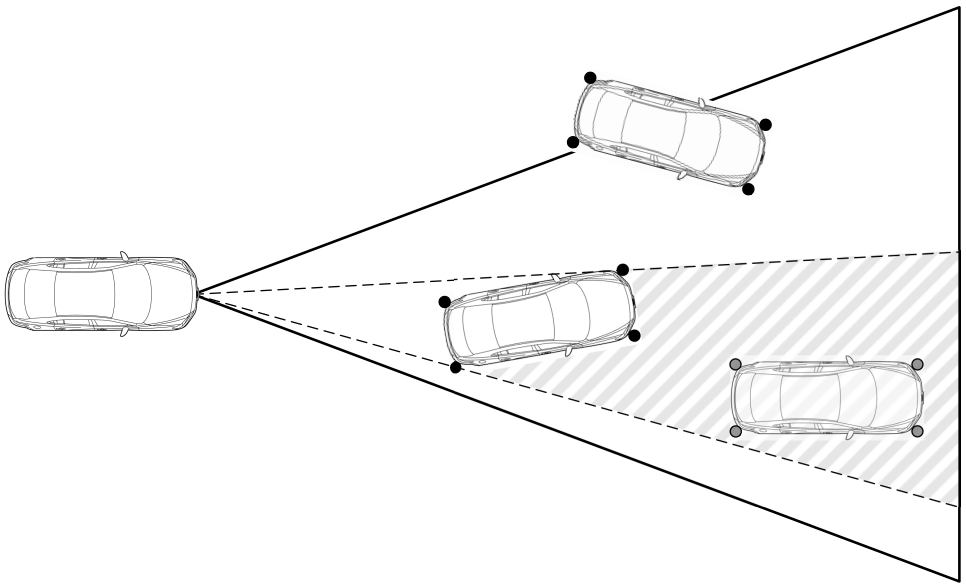

Abbildung 3.10: Geometrisches Radarmodell - Abschattung

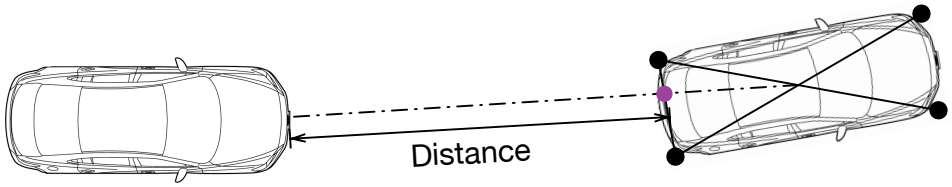

Abbildung 3.11: Geometrisches Radarmodell - Berechnung der Distanz

3.4.9 Modul Szenario

Ein wesentlicher Aspekt in einem Fahrsimulator ist die Generierung von Traffic. Aus diesem Grund wurde eine C++-Bibliothek entwickelt welche in der Lage ist, eben jenen Verkehr zu erzeugen und Fahrmanöver welche im Realversuch durchgeführt wurden auch in der Simulation nachzustellen.
Dieses Modul implementiert die darunter liegende Library und stellt der Simulation die generierten Daten zur Verfügung. Detaillierte Informationen sind in Kapitel 12 zu finden.

3.4.10 Modul Vision-Interface

Der Visualisierung kommt in einem Fahrsimulator eine bedeutende, wenn nicht sogar die bedeutendste Rolle zu. Eine realitätsgetreue Darstellung der Umwelt trägt maßgeblich dazu bei die Probanden in einen Zustand zu versetzten, in dem sie sich tatsächlich in ein reales Fahrzeug versetzt fühlen und damit auch entsprechend mit der Fahrsimulation interagieren.
Um die aktuelle Fahrzeugposition als auch die Positionen der umherfahrenden Fahrzeuge und Fußgänger darstellen zu können ist es notwendig, eine Schnittstelle zwischen dem Visualisierungssystem und der restlichen Fahrsimulation zu schaffen. Zu diesem Zweck wurden sowohl ein Plugin, welches im Kontext des Visualisierungs-Frameworks läuft als auch eine Bibliothek, welche in der Echtzeitumgebung innerhalb des Vision-Interface Moduls integriert ist, erstellt.
Ausführliche Informationen zur Integration sind in Kapitel 7 zu finden.

3.4.11 Modul Brake Pedal

Während die Lenkmomente direkt aus der Fahrdynamiksimulation AVL VSM zur Verfügung gestellt werden, bietet diese keine Möglichkeit die benötigten Momente am Bremspedal zu generieren.
Im Simulator wurde deshalb ein einfaches Modell hinterlegt, welches die Pedalkraft in Abhängigkeit der Pedalposition berechnet. Aus der Literatur [9] wurde ein Referenzmodell gewählt, wobei neben leichten subjektiven Anpassungen zusätzlich noch Effekte wie Reibung sowie Dämpfung simuliert werden.
Abbildung 3.12 illustriert hierbei den mittleren Verlauf der benötigten Bremspedalkraft, welcher zum momentanen Zeitpunkt im Fahrsimulator Verwendung findet.

Aufgrund der Realisierung des Bremspedals durch einen elektrischen Aktuator ist es zusätzlich leicht möglich, trotz fehlender Hydraulik im Fahrsimulator Effekte wie beispielsweise ABS zu simulieren. Weiters ist es möglich, adaptive Bremspedalcharakteristiken zu hinterlegen um beispielsweise auf der Autobahn ein anderes Verhalten zu erzeugen als im urbanen Gebiet.

3.4.12 Modul Safety Monitor

Nachdem der Elektromotor zur Aktuierung des Bremspedals in der Lage ist knapp 190N m aufzubringen ist es von wesentlicher Bedeutung, zur Vermeidung von Verletzungen

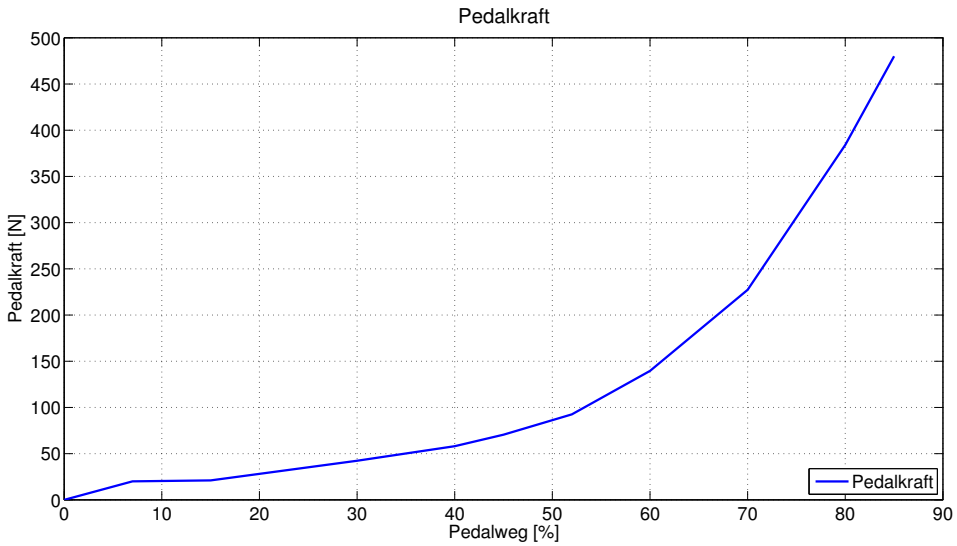

Abbildung 3.12: Bremspedalkraft

der Fahrer entsprechende Sicherheitsmechanismen zu treffen. Aus diesem Grund wurden mehrere Maßnahmen getroffen:

- Einbau eines Notaus-Schalters im Cockpit-Bereich
- Einbau von Endschaltern am Bremspedal zur Aktivierung des Notaus-Signals des Brems-Controllers bei Erreichen von Bremspedal-Positionen über den vorgesehenen Limits
- Überwachung des verbauten Kraftsensors in Software
- Überwachung der Momentangeschwindigkeit in Software
- Überwachung der elektrischen Leistung in Software
- Überwachung der Position in Software

Alle Parameter aus vorangehender Liste welche durch Software überwacht werden, werden in diesem Echtzeit-Modul laufend auf deren Wertebereich hin überprüft und gegebenenfalls eine Abschaltung des Aktuators in die Wege geleitet.
Die Grenzwerte wurden so gewählt, dass bei einer durchgeführten Notbremsung gerade noch keine Abschaltung aufgrund von Überschreitungen durchgeführt wird.

Kapitel 4

AVL VSM™ - Fahrdynamik

4.1 Einleitung

Mit der AVL List GmbH konnte ein Projektpartner gewonnen werden, welcher eine große Expertise im Bereich der Simulationstechnik vorweisen kann. Mit AVL VSM steht ein Softwarepaket zur Verfügung, welches weltweit in den verschiedensten Einsatzbereichen Anwendung findet.
AVL VSM ist eine Simulationssoftware zur Entwicklung und Verifikation von dynamischen Fahrdynamikmodellen, wobei praktisch alle gängigen Fahrzeugtypen, angefangen bei Agrarmaschinen, über Anhänger und PKWs bis hin zu Formel 1 Fahrzeugen, simuliert werden können. Neben der üblichen Offline-Simulation ist es auch möglich, AVL VSM im Zusammenhang mit Hardware in the Loop Lösungen in Echtzeit zu betreiben, um bestimmte Komponenten der Simulation durch bereits realisierte Hardware zu ersetzen.
Diese Fähigkeit wird im Fahrsimulator genutzt, wobei die üblichen Inputs wie Gas-, Brems- und Kupplungspedal, Gang-Wahlhebelstellung als auch Lenkradstellung von extern, durch die in Kapitel 2.1 beschriebenen Devices ersetzt werden.

AVL VSM arbeitet perfekt mit Matlab/Simulink zusammen, was eine ideale Integration in die restliche Simulationsumgebung ermöglicht. AVL VSM erlaubt es außerdem, eigene Simulink-Modelle als Kern der Simulation zu verwenden, wobei alle wesentlichen Module wie beispielsweise die ECU, Motor oder Kupplung als s-functions zur Verfügung gestellt werden. Jede dieser s-functions kann durch ein eigenes Modell ersetzt werden, so lange die Inputs und Outputs jeder einzelnen Komponente beibehalten werden.
Während in der Offline-Simulation ein simulierter Fahrer für die Generierung der benötigten Größen wie beispielsweise Gaspedalstellung zuständig ist, werden diese, wie zuvor erwähnt, im Fahrsimulator durch den eigentlichen Fahrer generiert und entsprechend in die Fahrdynamiksimulation eingespeist.

Zur Generierung eines Echtzeit-Targets wie in Kapitel 3 beschrieben, benötigt man neben dem vollständigen Simulink-Modell auch sämtlichen C/C++ Code von AVL VSM. Da es sich bei diesem um höchst sensible Daten handelt, wurde die Erstellung des Binaries zusammen mit Mitarbeitern der AVL List GmbH durchgeführt. Trotz der hohen Komplexität der Software war es relativ schnell möglich, eine erste lauffähige Version für

PTWinSim zu generieren.
Neben PTWinSim läuft Simulink auch auf Hard-Realtime-Systemen wie beispielsweise dSpace, Speedgoat, vTag oder den AVL-internen Lösungen.
Um den großen Ressourcenansprüchen gerecht zu werden welche bei Optimierungen bestimmter Parameter-Sets anfallen können besteht auch die Möglichkeit, AVL VSM im Verbund in der Cloud laufen zu lassen. So kann man auch komplizierteste Aufgaben in kurzer Zeit lösen.

Die Fahrdynamik-Simulation rechnet mit dem mit Abstand geringstem Update-Intervall von allen in der Simulation beteiligten Prozessen. In der momentanen Konfiguration wird das Modell mit 2 kHz upgedated. Dadurch kann auch bei sehr kritischen Fahrzeug-Konfigurationen jederzeit ein stabiles Verhalten gewährleistet werden.
Während die Simulation von Serienfahrzeugen verhältnismäßig geringe Ansprüche an die Solver stellt, so ist dies im Racing-Bereich eher kritisch. Hohe Federsteifigkeiten und starke Nichtlinearitäten resultieren oft in steifen Systemen, welche nur mit speziellen Solvern, oder eben kleinen Schrittweiten, vernünftig gelöst werden können.
Die Update-Rate wurde vom Projektpartner AVL empfohlen und hat sich bereits in vielen verschiedenen Konfigurationen bewährt. Der Realtime-Task benötigt in dieser Konfiguration etwa 30% der verfügbaren Echtzeit-Kapazitäten, was noch mehr als genug Spielraum für andere Tasks lässt.

4.2 Integration

Wie zuvor erwähnt müssen zwischen AVL VSM und den restlichen Modulen der Simulation Daten ausgetauscht werden. Die umfasst einerseits die grundlegenden Input-Signale des Fahrers, andererseits werden aber auch dynamische Daten der Fahrdynamik wie beispielsweise Längs- und Querbeschleuigungen benötigt, um entsprechende Regelkreise und Assistenzsysteme entwickeln zu können. Nachfolgende Tabellen 4.1 und 4.2 listen das zum Zeitpunkt der Erstellung der Arbeite implementierte Interface auf. Neue Kanäle können innerhalb weniger Minuten implementiert werden, wobei man hier aufgrund des unter Verschluss gehaltenen Source-Codes von AVL VSM wieder auf Unterstützung des Projektpartners angewiesen ist.

Abbildungen 4.1 und 4.2 zeigen einmal den IO Part des Simulink-Modells von VSM, und einmal wie die Outputs von VSM im VSM-Interface Modul eingelesen werden. Die Namen der Input/Output Blöcke müssen übereinstimmen, da ansonsten PTWinSim nicht in der Lage ist die entsprechenden Korrespondenzen herzustellen.

AVL VSM benötigt in der Regel Höheninformationen über das Gelände in dem sich das Fahrzeug gerade bewegt. In der Offline-Simulation erfolgt dies über AVL VSM selbst. Es können entsprechende Streckenverläufe samt Höhenprofil und Banking definiert werden, welche dann von der Fahrdynamik verwertet werden.
Im Fahrsimulator werden diese Informationen in der Regel von der Visualisierung bereitgestellt und von außen in die Fahrsimulation eingespeist. Hierzu benötigt das Sichtsystem einerseits die Radpositionen in absoluten Koordinaten und liefert auf der anderen Seite die Höheninformationen an jeder einzelnen Radposition retour.

Bezeichnung	Einheit	Wertebereich
Shift Neutral	[-]	Steigende Flanke
Shift Up	[-]	Steigende Flanke
Shift Down	[-]	Steigende Flanke
Steering Wheel Angle	[deg]	-360 - 360
Brake Pedal	[-]	0 - 100
Gas Pedal	[-]	0 - 100
Init Position	3 x [m]	entsprechend Szenerie
Init Velocity	[m/s]	0 - Max. Geschw. Auto
Init Heading	[rad]	$-\pi$ - π
Commanded State	[-]	0 - 10
Simulation State	[-]	0 - 10
Gear Shift Lever	[-]	entsprechend Setup
Surface Friction	[-]	0.1 - 2

Tabelle 4.1: VSM Interface Inputs

Bezeichnung	Einheit	Wertebereich
Position	3 x [m]	entsprechend Szenerie
Roll Angle	[rad]	-180 - 180
Pitch	[rad]	-180 - 180
Yaw Angle	[rad]	-180 - 180
Handwheel Torque	[N m]	entsprechend Setup
Engine Speed	[rpm]	0 - Max. Setup
Current Gear	[-]	-1 - Max. Setup
Acceleration Long	[g]	entsprechend Setup
Acceleration Lat	[g]	entsprechend Setup
Vehicle Speed	[m/s]	entsprechend Setup
Wheel Slip	4 x [-]	-1 - 1
Slip Angle	4 x [deg]	entsprechend Setup
Engine Torque	[N m]	entsprechend Setup
State	[-]	0 - 10

Tabelle 4.2: VSM Interface Outputs

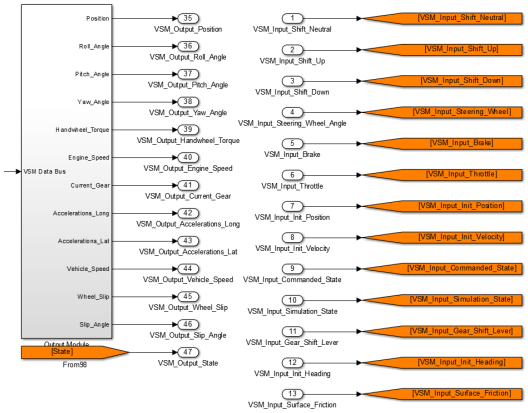

Abbildung 4.1: VSM Modell - IO

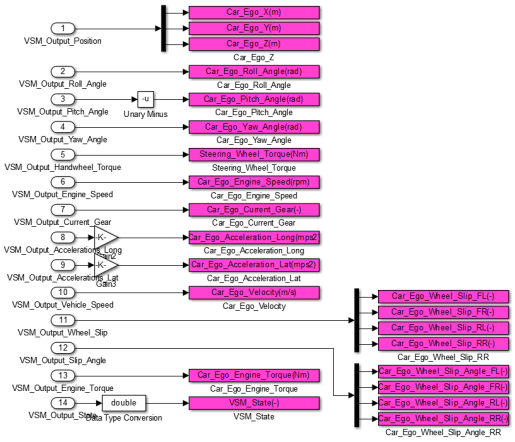

Abbildung 4.2: VSM Interface - Lesen der VSM Outputs

Nachdem die Visualisierung dieses Feature jedoch noch nicht unterstützt, wird VSM-intern auf einer ebenen Strecke bei Höhe 0m gefahren. Möchte man diese 2-Wege Kommunikation in Zukunft jedoch implementieren, so muss auch das Interface um die entsprechenden Kanäle erweitert werden. Im VSM Modell ist für diesen Mechanismus bereits alles vorbereitet, sodass nur mehr das Interface ergänzt werden muss.

Kapitel 5

IPG CarMaker® - Fahrdynamik

5.1 Einleitung

Nachdem das Institut für Fahrzeugtechnik der TU Graz intensiv IPG CarMaker nutzt und auch eine gute Partnerschaft mit IPG Automotive pflegt, lag es natürlich nahe, auch IPG CarMaker für die Fahrdynamik nutzen zu können. Wie bei AVL VSM besteht auch bei CarMaker die Möglichkeit, eigene Simulink-Modelle im Zusammenspiel mit der eigenen Fahrdynamik zu verwenden. Der grundsätzliche Zugang ist sehr ähnlich: Wie bei VSM nutzt auch CarMaker s-functions, um Komponenten der Fahrdynamik in Matlab/Simulink zu integrieren.
Nachdem der Source Code jedoch nicht zur Verfügung steht bzw. auch keine nähere Zusammenarbeit mit IPG bezüglich dem Fahrsimulator zum Zeitpunkt der Erstellung geplant war, kann kein Echtzeit-Task für PTWinSim erstellt werden. CarMaker muss demnach als eigenständiger Prozess in Simulink, unabhängig von der Echtzeitumgebung, laufen und ist demnach nicht zeitsynchronisiert.
An dieser Stelle sei auf Kapitel 3 verwiesen wo anschaulich illustriert wird, welche Folgen aus dieser Tatsache resultieren können.

Möchte man IPG CarMaker dennoch als eigenständige Applikation außerhalb der Echtzeitumgebung laufen lassen, so sollte man zumindest die Traffic-Generierung als auch das Interface zur Visualisierung direkt im CarMaker-Modell integrieren. Auf diesem Wege werden ungewollte visuelle Artefakte, welche aus unterschiedlicher Berechnung der Positionen von Ego- und Target-Fahrzeugen resultieren, vermieden.
Momentan ist aufgrund der zufriedenstellenden Funktionsweise von AVL VSM keine Umstellung auf IPG CarMaker geplant. Möchte man in Zukunft dennoch CarMaker anstatt VSM verwenden, so lautet die Empfehlung jedoch eindeutig sich mit IPG in Verbindung zu setzen um eine Lösung zur Integration von IPG CarMaker in die Echtzeitumgebung PTWinSim zu finden.

5.2 Integration

Nachdem CarMaker nicht als PTWinSim-Task läuft, muss auch die Kommunikation mit den restlichen Modulen auf eine andere Weise implementiert werden als es bei VSM der Fall

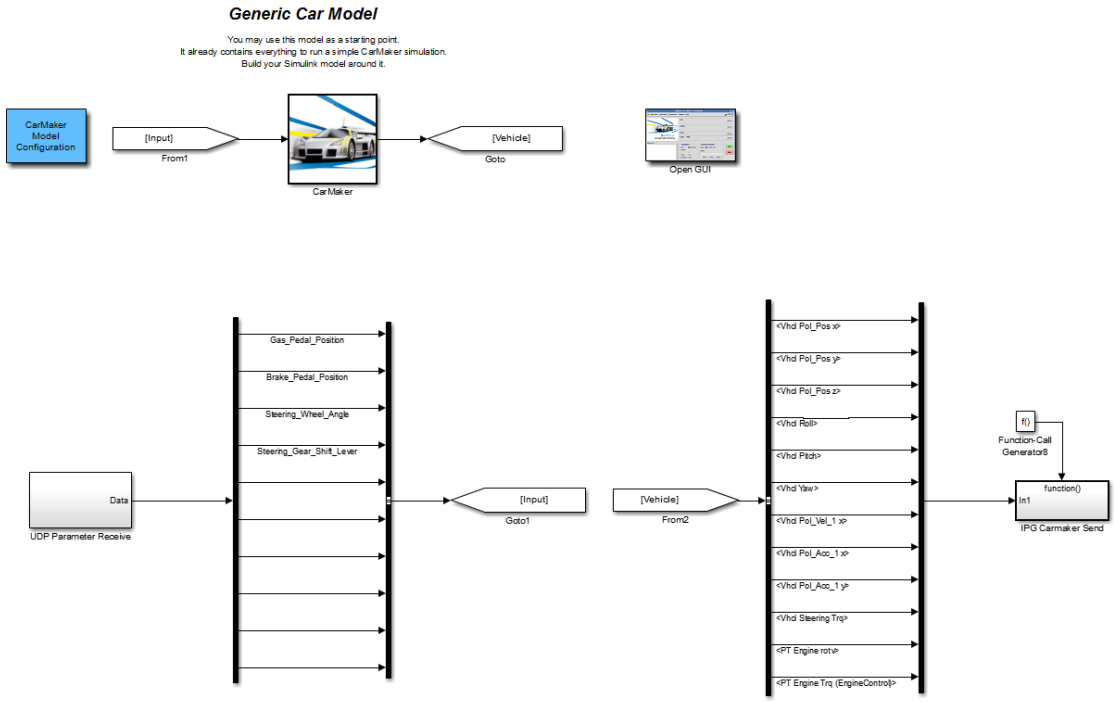

Abbildung 5.1: CarMaker - Interface via UDP

ist. Der Einfachheit halber fiel die Wahl auf ein einfaches UDP-Interface, nachdem dies sehr simpel mit den Standard-Simulink Bibliotheken integriert werden kann. Die Daten welche ausgetauscht werden sind ident zu AVL VSM, nur dass der Datenaustausch eben über ein Netzwerk-Protokoll und nicht innerhalb der Echtzeitumgebung passiert. In Abbildung 5.1 ist zu sehen, wie die Input- und Ouput-Signale über UDP in das Simulink-Modell integriert sind.
Abbildung 5.2 verdeutlicht schließlich, wie die Controls von CarMaker durch jene Signale, welche über UDP geliefert werden, ersetztwerden.

Möchte man die Simulation von VSM auf CarMaker umstellen so genügt es, den VSM- als auch VSM-Interface-Task aus PTWinSim zu entfernen, und stattdessen das CarMaker-Interface zu laden und CarMaker an sich zu starten.
Durch den hohen Grad an Modularität ist es selbstverständlich auch möglich, andere Simulations-Tools wie beispielsweise Dymola nach der gleichen Vorgangsweise zu integrieren.

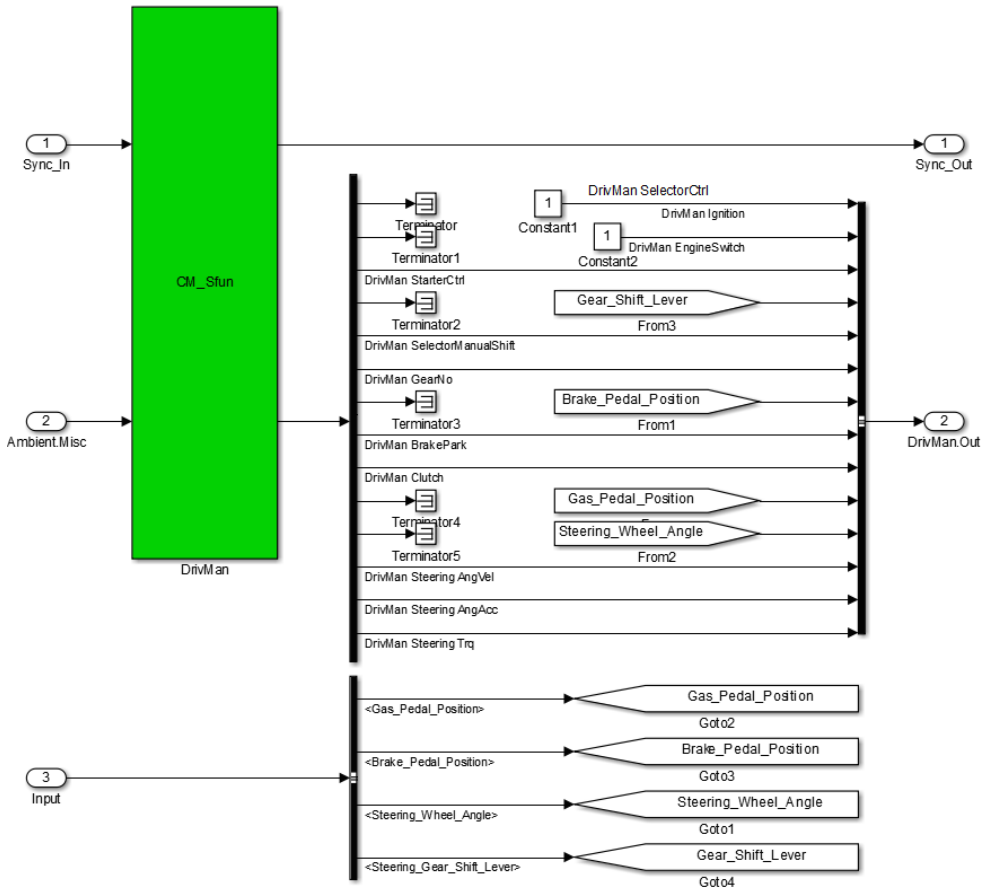

Abbildung 5.2: CarMaker - Überschreiben der internen Controls

Kapitel 6

Analog IO

6.1 NI USB-6251

Wie bereits in Kapitel 2.1.6 beschrieben, findet im Fahrsimulator ein Produkt der Firma Dewetron mit der Bezeichnung DEWE-50-USB2-16 Verwendung, welches in Abbildung 6.1 dargestellt ist. Normalerweise wird die Data Acquisition Unit im Zusammenspiel mit der eigens entwickelten Software zur Messdatenaufzeichnung verwendet, für einen Real-Time-Betrieb im Fahrsimulator stellt diese jedoch keine entsprechende Funktionalität zur Verfügung.

Es hat sich gezeigt, dass besagtes IO-Device intern ein Modul der Firma National Instruments mit der Bezeichnung NI USB-6251 verwendet, welches auch in OEM-Ausführungen verfügbar ist. Dieses bietet 16 16bit Analogeingänge, zwei Analogausgänge, 24 digitale IO-Kanäle sowie zwei 32Bit Counter/Timer. Weitere wichtige Spezifikationen sind in Tabelle 6.1 aufgelistet.

6.2 NI-DAQmx

National Instruments stellt mit der NI-DAQmx Driver Software ein Programmier-Interface bereit mit dem es möglich ist, von NI unterstützte Data Acquisition Units anzusprechen. Für die Windows-Plattform stehen APIs für Visual Basic, C# sowie C/C++ zur Verfügung. Weiters wird auch Linux sowie Mac OS X unter C/C++ unterstützt.

Abbildung 6.1: DEWE-50-USB2-16 [10]

Channels	8 Differential oder 16 Single Ended
ADC Auflösung	16bit
Sampling Rate	1.25MS/s bei Erfassung eines Kanals
Zeitauflösung	50ns
Zeitgenauigkeit	50ppm der Sampling Rate
Input Range	± 0.1V, ± 0.2V, ± 0.5V, ± 1V, ± 2V, ± 5V, ± 10V
Maximale Eingangsspannung	± 11V
CMRR	100dB
Kleinsignalbandbreite (-3dB)	1.7MHz
Input Bias Current	±100 µA
Eingangsimpedanz	>10GΩ parallel mit 100pF

Tabelle 6.1: Spezifikationen Analogeingänge NI USB-6251

Mit den genannten Bibliotheken ist es nun möglich, das DEWE-50-USB2-16 selbstständig, ohne auf externe Software angewiesen zu sein, in die Fahrsimulator-Umgebung zu integrieren. Zu diesem Zweck wurde eine C++-Bibliothek entwickelt, welche den Zugriff auf das IO-Device unter Matlab/Simulink, und schlussendlich auch innerhalb der Echtzeitumgebung, ermöglicht.

Abbildung 6.2 veranschaulicht den Ablauf, um mit Hilfe der NI-DAQmx Bibliothek Analog-Daten von einem unterstützten IO-Device zu empfangen.

1. Zuallererst wird ein Task angelegt, in dessen Kontext jede weitere Kommunikation von statten geht. Für den Fahrsimulator existieren zwei getrennte Tasks. Einmal für Analog-Input, sowie einmal für das Schreiben von Analogwerten.

2. Anschließend werden die zu lesenden oder schreibenden Kanäle instanziert und den entsprechenden Tasks zugeordnet. Hier werden auch die minimalen sowie maximalen Spannungspegel mit übergeben.

3. Nun muss der Kanal konfiguriert werden. In diesem Schritt werden vor allem die Sampling-Raten sowie die Buffer-Größen definiert.

4. Ist alles konfiguriert muss der Bibliothek ein Callback übergeben werden, welcher beim Eintreffen neuer Daten aufgerufen wird. Dies geschieht genau dann, wenn eine definierte Anzahl an Samples geschrieben oder gelesen wurden.

5. Anschließend erfolgt das Starten der Tasks, womit die tatsächliche Kommunikation mit dem Device beginnt.

6. So lange keine Stop-Nachricht gesendet wird, wird der Callback entsprechend der Konfiguration laufend mit den neu erhaltenen Daten aufgerufen.

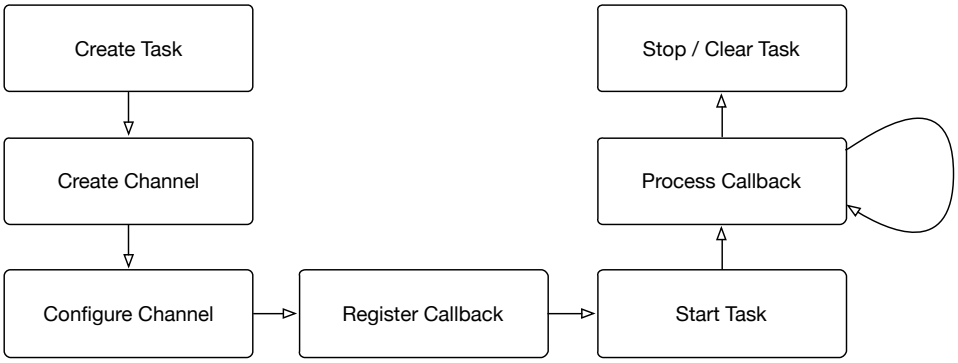

Abbildung 6.2: NI-DAQmx Ablaufdiagramm

7. Möchte man die Kommunikation beenden, so wird eine Stop sowie Clear Botschaft gesendet, womit das Device wieder in seinen Ausgangszustand versetzt wird.

6.3 DaqLib

Um das Dewetron IO-Device bequem aus Matlab/Simulink heraus ansprechen zu können, wird die NI-DAQmx API in einer eigens entwickelten C++-Bibliothek gekapselt. Diese stellt ein einfaches Interface bereit, welches in Listing 6.1 dargestellt wird. Entsprechend der Vorgangsweise aus Kapitel 3.3 werden auch hier für jeden Funktionsaufruf korrespondierende Matlab s-functions für die Einbindung in Simulink sowie .tlc Files für die Codegenerierung erstellt.

```
1  #pragma once
2
3  #ifdef __cplusplus
4  extern "C" {
5  #endif
6
7      // Get analog value of given channel number
8      __declspec(dllexport) double getValue(unsigned int channel) const;
9      // Initialize Library
10     __declspec(dllexport) void init(void);
11     // Set analog output with given value
12     __declspec(dllexport) void setValue(double newValue);
13     // Stop communication and clean up
14     __declspec(dllexport) void stop(void);
15
16 #ifdef __cplusplus
17 }
18 #endif
```

Listing 6.1: Interface der *DaqLib*

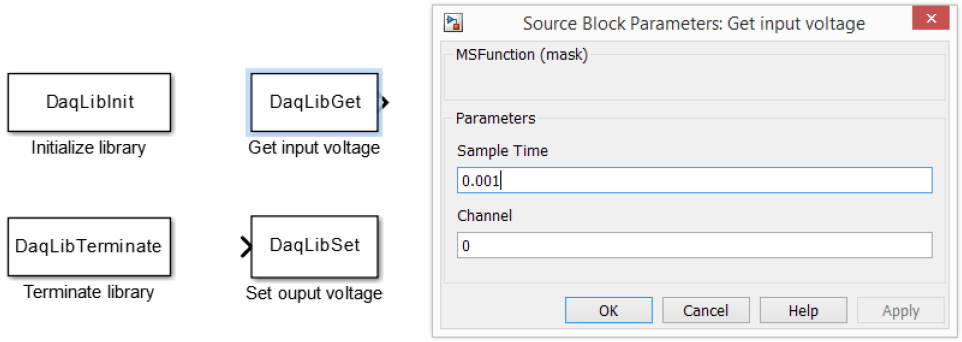

Abbildung 6.3: *DaqLib* - Simulink Integration

Abbildung 6.3 zeigt die zum Interface aus Listing 6.1 gehörenden Simulink-Blöcke. Zum Einlesen eines Analogwerts ist beispielsweise lediglich eine Sampling-Rate mit der der Bibliotheks-Aufruf erfolgen soll sowie eine Kanalnummer anzugeben. Selbstverständlich können mehreren Schreib- und Leseblöcke samt unterschiedlichen Kanälen gleichzeitig verwendet werden. Jeder Kanal entspricht hierbei genau einem einem IO-Steckmodul im Dewetron-Device, wobei diese in aufsteigender Reihenfolge verbaut sind.

Der Init- sowie Terminate-Block muss in jedem Simulink-Modell inkludiert sein welches die Bibliothek implementiert. Diese Blöcke sind für die korrekte Initialisierung sowie das saubere Beenden zuständig.
Der Vollständigkeit halber sei beispielhaft das .tlc File zum Einlesen der Analogwerte in Listing 6.2 angegeben.
Weitere Informationen wurden bereits in Kapitel 3 angeführt.

```
1  %implements DaqLibGet "C"
2
3  %function BlockTypeSetup(block, system) void
4      %<LibAddToCommonIncludes("daqlib.h")>
5  %endfunction
6
7  %function Outputs(block, system) Output
8    /* %<Type> Block: %<Name> */
9    %assign y = LibBlockOutputSignal(0, "", "", 0)
10   %assign ch = ParamSettings.Channel
11   %<y> = getValue((unsigned int)%<ch>);
12 %endfunction
```

Listing 6.2: TLC-File des DaqLibGet-Simulink-Blocks

Kapitel 7

Visualisierung

Abbildung 7.1: Panorama der Visualisierung

Wie bereits in Kapitel 2.1 angeführt, besteht das System zur Visualisierung der Umwelt aus 4 Rechnern sowie 8 Bildschirmen mit einer Full High Definition (Full HD) Auflösung von je 1920x1080.
Ein Master-Rechner dient hierbei als Schnittstelle zur Simulation und hält die Daten der Szenerie für die Slave-Rechner bereit. Weiters ist der Master für die Synchronisation der Visualisierungs-Rechner untereinander zuständig.

7.1 InstantReality Framework

Nachdem die Visualisierung in Zusammenarbeit mit dem Fraunhofer-Institut für Graphische Datenverarbeitung entwickelt wurde, kommt das plattformunabhängige InstantReality Framework [11], entwickelt von der selben Institution, zur Anwendung.
Das InstantReality Framework wurde mit Fokus auf hohe Performance entwickelt und wird bereits in einer Vielzahl von Augmented und Virtual Reality Projekten verwendet. Das System unterstützt das Prinzip des Clusterings, also dem Aufteilen des Renderings auf

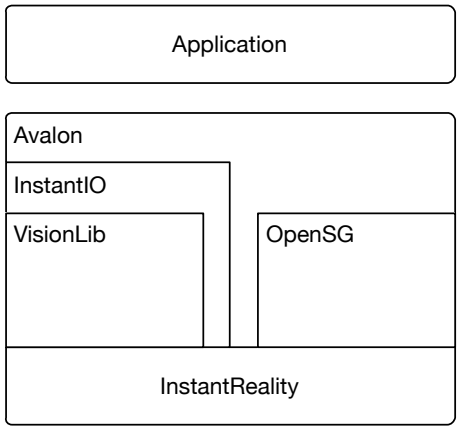

Abbildung 7.2: Überblick über das InstantReality Framework

verschiedene Cluster-Nodes, Multi-Resolution Datasets um die Render-Performace adaptiv anpassen zu können. So können beispielsweise mittels eigens geschriebener Plugins IO-Nodes integriert werden, um die Interaktion mit Gesten- oder Head-Trackern zu ermöglichen oder um, wie im Falle des Fahrsimulators, bestimmte Knoten des Szenengraphs über das Netzwerk gezielt steuern zu können.
An der Universität Darmstadt beispielsweise wird das InstantReality Framework verwendet, um die so genannte HEyeWall 2.0 anzusteuern. Hierbei finden 24 Projektoren sowie 48 PCs Verwendung, um eine High-Resolution Projektion zu realisieren. Dies ist nur eines von vielen Beispielen, wo die Leistungsfähigkeit des Software-Pakets gezeigt wird.
An der Technischen Universität Graz wird selbiges Framework in der DAVE (Definitely Affordable Virtual Environment) [12] verwendet, einem von 4 Seiten durch Rückprojektion beleuchteten Raum, in dem man sich interaktiv mit Hilfe einer 3D-Brille sowie Head-Traking in praktisch beliebige virtuelle Umgebungen versetzen kann.

Weiters stehen eine Vielzahl kleinerer Tools zur Verfügung welche das Arbeiten mit der InstantReality-Umgebung erleichtern. So sind beispielsweise Plugins für Autodesk 3D Studio Max sowie Autodesk Maya erhältlich um Szenerien und Animationen in das vom InstantReality Framework bevorzugte X3D Format zu übertragen.
X3D steht für Extensible 3D und ist ein auf dem XML Standard basierendes Dateiformat zur Repräsentation von 3D-Modellen. X3D hat in der Zwischenzeit den Vorgänger, VRML (Virtual Reality Modeling Language), abgelöst.
Der InstantPlayer setzt auf dem InstantReality Framework auf und dient als Standalone-Applikation schlussendlich zur Darstellung der .x3d Dateien. Zusammen mit dem Instant-Cluster ist es nun möglich, einen Szenengraph auf verschiedene Rechner zu verteilen und im Verbund synchronisiert darstellen zu lassen.

Abbildung 7.2 veranschaulicht den prinzipiellen Aufbau des InstantReality Frameworks. Der Software-Stack besteht aus 4 grundlegenden Komponenten:

- **Avalon** ist für das Szenenmanagement sowie die Szenenmanipulation zuständig.

- **InstantIO** dient zur Einbindung von Netzwerk- sowie Hardwareknoten in die Verarbeitungspipeline.
- Die **VisionLib** repräsentiert die Vision-/Image-Pipeline Processor Unit.
- **OpenSG** ist ein OpenSource Szenengraphsystem und ist für das Rendering der Szenerie mit Hilfe von OpenGL zuständig.

7.2 InstantIO Plugin

Nachdem bereits in Abbildung 2.1 die Topologie der Anbindung der Visualisierung an die restliche Simulation grob dargestellt wurde, so ist diese nun etwas näher in Abbildung 7.3 veranschaulicht.
Der Simulationsrechner stellt alle benötigten Daten welche zur korrekten Darstellung benötigt werden zur Verfügung. Dies beinhaltet vor allem

- Position und Orientierung des Ego-Fahrzeuges und damit des Viewpoints aus der Vehicle Dynamics Simulation,
- Position und Orientierung der restlichen Target-Fahrzeuge aus der Szenarien-Library,
- Position und Orientierung der Fußgänger aus der Szenarien-Library.

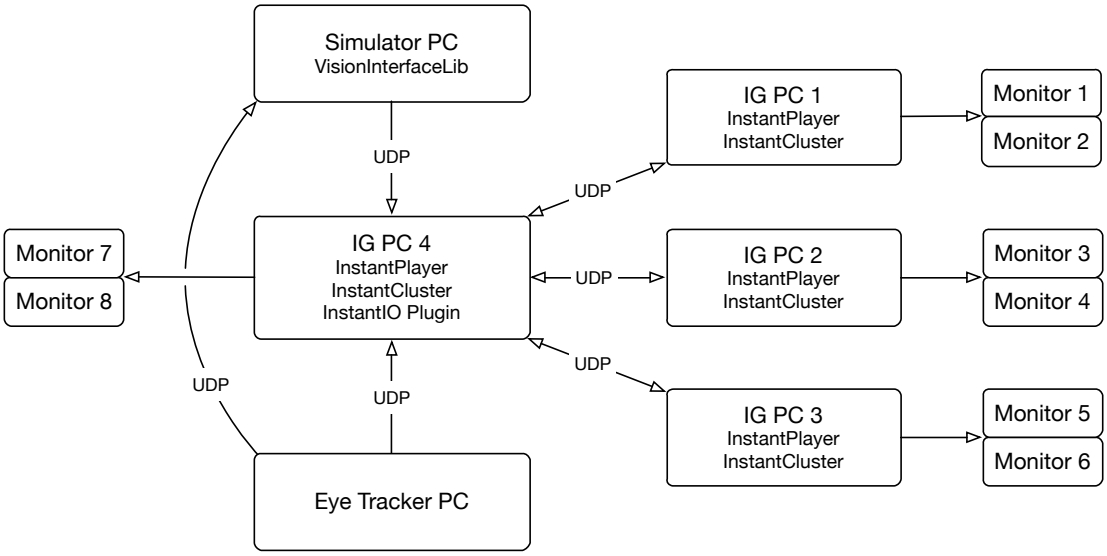

Abbildung 7.3: Detailübersicht der Visualisierungstopologie

Der ImageGenerator 4 stellt die Schnittstelle zur Visualisierung dar, während IG1, IG2 sowie IG3 nur untereinander mit dem Master kommunizieren. Auf allen Instanzen des Visualisierungssystems laufen der InstantCluster zur Synchronisation und Verteilung der Rechenaufgaben sowie ein InstantPlayer zur eigentlichen Darstellung der Umgebung.

Der InstantPlayer stellt eine REST-Schnittstelle zur Manipulation der verschiedenen Knoten im Szenengraphen zur Verfügung. Es hat sich jedoch schnell gezeigt, dass diese Art der Kommunikation nicht annähernd jene Performance liefern kann, welche für eine flüssige Manipulation des Viewpoints notwendig ist. Aus diesem Grund wurde eigens ein InstantIO-Plugin, welches vom InstantPlayer geladen werden kann, entwickelt, welches über UDP mit dem Kommunikationsrechner kommuniziert.
Nachfolgend soll der grundlegende Ablauf zur Einbindung solch eines Moduls beschrieben werden.

7.2.1 Einbindung des InstantIO Plugin

Listing 7.1 zeigt einen kleinen Ausschnitt einer .x3d Datei, welche unter anderem im Fahrsimulator Anwendung findet. Die vollständige Datei zur Beschreibung von Kirchdorf samt Autobahn umfasst insgesamt in etwa 37.000 Zeilen.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE X3D PUBLIC "ISO//Web3D//DTD X3D 3.0//EN"
    "http://www.web3d.org/specifications/x3d-3.0.dtd">
<X3D xmlns:xsd='http://www.w3.org/2001/XMLSchema-instance' profile='Full'
    version='3.0'
    xsd:noNamespaceSchemaLocation='http://www.web3d.org/specifications/x3d-3.0.xsd'>

  <Scene DEF='scene'>
    <IOSensor DEF='IO' type='ScenarioNode' port='8856'>
      <field accessType='outputOnly' name='position'
                  type='SFVec3f'/>
      <field accessType='outputOnly' name='rotationRoll'
                  type='SFRotation'/>
      <field accessType='outputOnly' name='rotationPitch'
                  type='SFRotation'/>
      <field accessType='outputOnly' name='rotationYaw'
                  type='SFRotation'/>
    </IOSensor>

    <NavigationInfo type='"EXAMINE" "ANY"' speed='20'/>

    <Transform DEF='VIEWPOINT'>
      <Transform DEF='DRIVER_POS_YAW'>
        <Transform DEF='DRIVER_POS_PITCH'>
          <Transform DEF='DRIVER_POS_ROLL'>
            <Viewpoint DEF='DRIVER_POS' description='Default View'
                position='0 10 0'/>
          </Transform>
        </Transform>
      </Transform>
    </Transform>

    <ROUTE fromNode='IO' fromField='position'
                              toNode='DRIVER_POS'
                              toField='set_position'/>
```

```
33
34      <ROUTE fromNode='IO' fromField='position'
35                           toNode='DRIVER_POS_ROLL'
36                           toField='center'/>
37      <ROUTE fromNode='IO' fromField='position'
38                           toNode='DRIVER_POS_PITCH'
39                           toField='center'/>
40      <ROUTE fromNode='IO' fromField='position'
41                           toNode='DRIVER_POS_YAW'
42                           toField='center'/>
43      <ROUTE fromNode='IO' fromField='rotationRoll'
44                           toNode='DRIVER_POS_ROLL'
45                           toField='set_rotation'/>
46      <ROUTE fromNode='IO' fromField='rotationPitch'
47                           toNode='DRIVER_POS_PITCH'
48                           toField='set_rotation'/>
49      <ROUTE fromNode='IO' fromField='rotationYaw'
50                           toNode='DRIVER_POS_YAW'
51                           toField='set_rotation'/>
```

Listing 7.1: Einbindung InstantIO-Plugin

In Zeile 7 erfolgt die Definition eines so genannten IOSensors, welche im Prinzip das zu ladende Plugin beschreibt. Der Name welcher beim `type` angegeben wird, muss mit jenem der zu ladenden .dll übereinstimmen. Weiters muss die Bibliothek im selben Verzeichnis liegen wie die Binary des InstantPlayers. Zusätzlich ist es möglich, weitere Parameter an das Plugin zu übergeben. Im gegebenen Fall wird beispielsweise der Port, auf welchen das UDP-Interface für eingehende Daten hören soll, angegeben.
Die nachfolgenden Zeilen beschreiben die In- und Outputs des Plugins. In unserem Fall hat das Plugin nur Ausgänge. Einmal die Position des Viewpoints, also die Position des Ego-Fahrzeuges, und zusätzlich die Rotationen um die einzelnen Achsen. Jene Daten werden von der Fahrsimulation generiert und an das Plugin gesendet. Nicht angeführt sind der Einfachheit halber die Positionen und Orientierungen für die Target-Fahrzeuge und Fußgänger. Das Grundprinzip ist jedoch ident.
Die Zeilen 19 bis 28 beschreiben mit Hilfe der Transform-Knoten die Position und Orientierung des Viewpoints welche aus dem Plugin heraus gesetzt werden soll.
Die `ROUTE`-Direktiven verknüpfen nun die Ausgänge des Plugins mit den Eigenschaften von definierten Szenen-Knoten. So wird beispielsweise in den Zeilen 30 bis 32 das Datenfeld `position` vom Typ `SFVec3f` an das Feld `set_position` des Viewpoints gebunden. Die Position wird ebenfalls an das `center` Feld der Orientierung-Transforms gebunden, damit die Drehung um den Viewpoint herum statt findet. Würde man das `center` Feld nicht auf die Viewpoint-Position setzen, so würde die Drehung um den Welt-Koordinaten-Ursprung herum erfolgen.
Gleich wie die Positionen können dann auch die Rotationen an die entsprechenden Transforms gebunden werden. Es ist zu beachten, dass die Transformationsreihenfolge ident mit jener der Simulation ist, andernfalls wird die Orientierung nicht korrekt abgebildet.
Ändert sich nun ein Ausgang des Plugins, so spiegelt sich die Änderung automatisch in

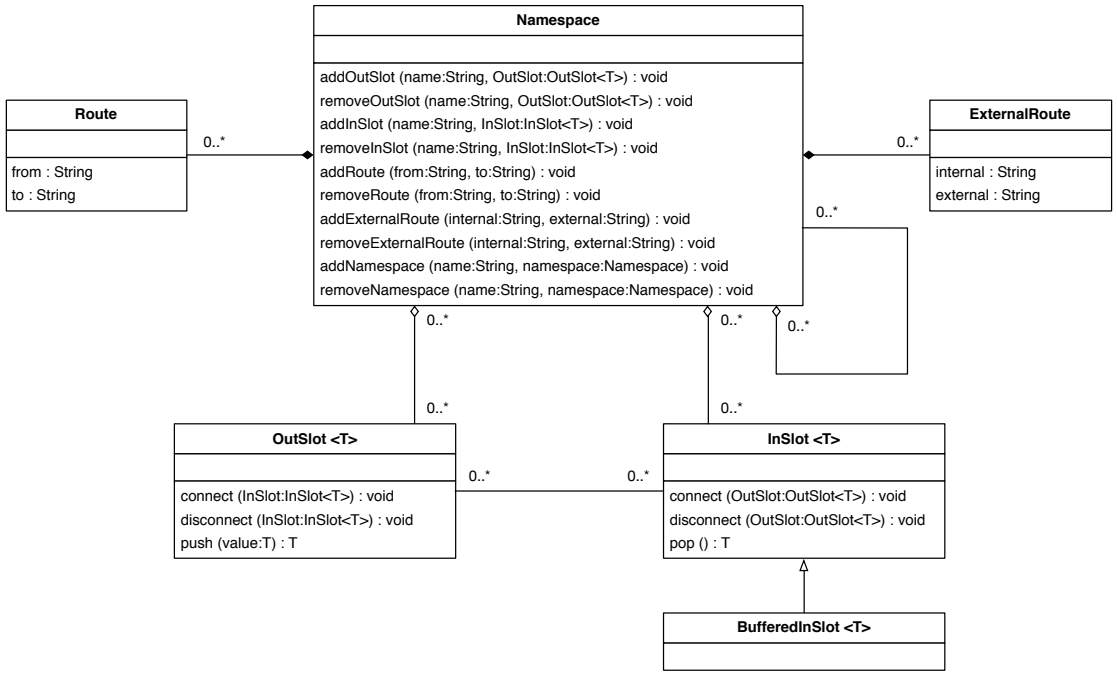

Abbildung 7.4: UML Diagram des InstantIO Interface

der Visualisierung entsprechend der definierten Bindings wieder.

7.2.2 Erstellen des InstantIO Plugin

Wie bereits erwähnt besteht die Möglichkeit Plugins für den InstantPlayer zu erstellen, um Knoten des Szenegraphen zu beeinflussen. Neben Java stehen auch Interfaces für C++ seitens des InstantReality Frameworks zur Verfügung. Nachfolgend soll kurz die Vorangehensweise bei der Erstellung eines C++ Plugins erläutert werden.

Abbildung 7.4 zeigt ein UML-Diagramm welches einen kleinen Bruchteil der zur Verfügung stehenden Funktionen und Klassen zur Erstellung eines eigenen Sensor-Knotens visualisiert. Es zeigt sich schnell die Ähnlichkeit welche zwischen der Interface-Definition und der Einbindung im X3D-File besteht. In beiden Fällen besteht die Möglichkeit Routen zu definieren und auch die In- sowie Outslots lassen darauf schließen, dass diese zur Kommunikation mit gewissen Parametern der X3D-Knoten dienen.
Nachfolgend sei das genaue Interface einer Minimal-Implementierung eines InstantPlayer-Plugins aufgelistet.

```
1 #pragma once
2
3 // Declares __declspec(dllimport/dllexport) INSTANT_IO_API
4 #include "definitions.h"
5 #include <string>
6 // Include needed header files
```

```
7  #include <InstantIO/ThreadedNode.h>
8  #include <InstantIO/NodeType.h>
9  #include <InstantIO/Rotation.h>
10 #include <InstantIO/Vec3.h>
11
12 namespace InstantIO
13 {
14     template <class T> class OutSlot;
15     class INSTANT_IO_API InstantNode : public ThreadedNode
16     {
17     public:
18         InstantNode();
19         virtual InstantNode();
20
21         // Create setter and getter methods for an arbitrary field
22         // Here we create setter/getter e.g. for the port of an udp interface
23         INSTANTIO_SETTER_GETTER(unsigned int, Port, m_port);
24         // Factory method to create an instance of InstantNode
25         static Node *create();
26         // Factory method to return the type of InstantNode
27         virtual NodeType *type() const;
28     protected:
29         // Gets called when the InstantNode is initialzed
30         virtual void initialize();
31         // Gets called when the InstantNode is closed
32         virtual void shutdown();
33         // Thread method which is started as soon as a slot is connected
34         virtual int processData();
35
36     private:
37         InstantNode(const InstantNode &);
38         const InstantNode &operator=(const InstantNode &);
39         // Holds the port specified at IOSensor definition
40         // Value is assigned by INSTANTIO_SETTER_GETTER macro
41         // Can be used e.g. by an udp interface
42         unsigned int m_port;
43         // Define dynamic slots
44         // Values are set in processData() call
45         OutSlot<Vec3f>* m_viewpointPosition;
46         OutSlot<Rotation>* m_viewpointRotationRoll;
47         // Type and description attributes of the plugin
48         static NodeType m_type;
49         static const char* m_typeName;
50         static const char* m_shortDescription;
51         static const char* m_longDescription;
52         static const char* m_author;
53         static Field m_fields[1];
54     };
55 } // namespace InstantIO
```

Listing 7.2: Interface eines InstantIO-Plugins

In Listing 7.3 wird beispielhaft dargestellt, wie Outslots für die Viewpoint-Position sowie den Roll-Winkel erstellt werden können. Vorzugsweise erfolgt dies im `initialize` Call des Plugins.
In den Zeilen 8 und 17 wird den Slots jeweils ein Name, in diesem Fall `position` sowie `rotationRoll`, übergeben. Vergleicht man diese nun mit den `name`-Attributen der `field` sowie `ROUTE` Knoten in den Zeilen 7, 9, 30 und 42 des X3D-Files in Listing 7.1, so ist zu sehen, dass die Namen übereinstimmen. Auf diese Art und Weise erfolgt demnach die Zuordnung der Slots aus dem Plugin heraus hin zur X3D-Datei. Wesentlich ist, dass die Datentypen im Plugin und im `IOSensor` Knoten ident sein müssen.

```
1   // Create new OutSlot of Type Vec3f for the position
2   // and initialize to 0
3   m_viewpointPosition = new OutSlot<Vec3f>
4           ("Position of viewpoint", Vec3f(0.f, 0.f, 0.f));
5   assert(m_viewpointPosition != 0);
6   m_viewpointPosition->addListener(*this);
7   // Add the OutSlot with the name 'position'
8   addOutSlot("position", m_viewpointPosition);
9
10  // Create new OutSlot of Type Rotation for the roll angle
11  // and initialize to 0
12  m_viewpointRotationRoll = new OutSlot<Rotation>
13      ("Rotation Roll of viewpoint", Rotation());
14  assert(m_viewpointRotationRoll != 0);
15  m_viewpointRotationRoll->addListener(*this);
16  // Add the OutSlot with the name 'rotationRoll'
17  addOutSlot("rotationRoll", m_viewpointRotationRoll);
```

Listing 7.3: Erstellen der Slots

Innerhalb der Thread-Methode `processData()` werden in einer while-Schleife, wie in Listing 7.4 dargestellt, im gewünschten Zeitintervall die Inhalte der Slots entsprechend gesetzt. Während im dargestellten Source-Code in den Zeilen 12 und 21 nur konstante Werte gesetzt werden, werden diese Aufrufe im Fahrsimulator durch Größen ersetzt, welche über ein UDP-Interface vom Plugin aus der Fahrsimulation heraus empfangen werden.

```
1   // Position of ego vehicle
2   Vec3f position;
3   // Roll angle of ego vehicle
4   Rotation rotRoll;
5
6   // Wait for 5ms
7   while (waitThread(5))
8   {
9       // Set ego position to x=10, y=20, z=1
10      if (m_viewpointPosition != 0)
11      {
```

```
12              position.set(10.f, 20.f, 1.f);
13              m_viewpointPosition->push(position);
14          }
15
16          // Set ego roll angle to 0.1rad
17          if (m_viewpointRotationRoll != 0)
18          {
19              // Define axis of rotation
20              Vec3f rotVecRoll(0.f, 0.f, 1.f);
21              rotRoll.set(rotVecRoll, 0.1f);
22              m_viewpointRotationRoll->push(rotRoll);
23          }
24      }
```

Listing 7.4: Setzen der Daten

In der `shutdown()`-Methode müssen schlussendlich die erstellten Slots wieder entfernt werden. Dies kann einfach mittels der `removeOutSlot` und `removeInSlot` Aufrufen, wie in Listing 7.5 gezeigt, erfolgen. Auch hier werden wieder die zuvor verwendeten Namen als Identifier verwendet.

```
1       // Remove slot for viewpoint position
2       if (m_viewpointPosition != 0)
3       {
4           removeOutSlot("position", m_viewpointPosition);
5           delete m_viewpointPosition;
6           m_viewpointPosition = 0;
7       }
8       // Remove slot for roll angle
9       if (m_viewpointRotationRoll != 0)
10      {
11          removeOutSlot("rotationRoll", m_viewpointRotationRoll);
12          delete m_viewpointRotationRoll;
13          m_viewpointRotationRoll = 0;
14      }
```

Listing 7.5: Löschen der Slots

Das UDP-Paket welches aus der Simulation an das Plugin gesendet wird beinhaltet nur einen Timestamp sowie x, y, und z Koordinaten für Position und Orientierung, sowohl für den Viewpoint als auch die Target-Fahrzeuge, sowie für Fußgänger.
In einer weiteren Ausbaustufe sollen auch Höheninformationen sowie Kollisionserkennung von der Visualisierung zurück an die Fahrdynamik gesendet werden können. Dies ermöglicht dann auch das Fahren auf nicht-ebenen Geländeprofilen.

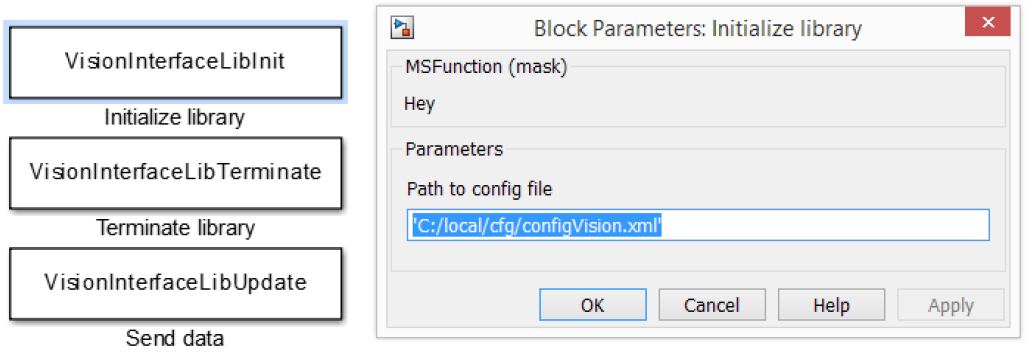

Abbildung 7.5: VisionInterface - Definition des Config File

7.3 VisionInterface Library

Während im vorhergehenden Kapitel erläutert wurde wie man ein Plugin für den InstantPlayer erstellen kann, so muss es auf der Gegenseite einen Prozess geben, welcher die benötigten Daten über das Netzwerk an das Plugin sendet. Im Zuge der Entwicklung des Fahrsimulators wurde eine Bibliothek entwickelt, welche die UDP-Kommunikation hin zum InstantIO-Plugin übernimmt. Die grundsätzliche Vorgangsweise zur Erstellung solch einer Bibliothek ist ident zu jener wie in Kapitel 3.2 illustriert.
Im zur Bibliothek gehörigen Init-Block im Simulink-Koppelplan, wie in Abbildung 7.5 zu sehen, kann ein Pfad zu einer Konfigurationsdatei angegeben werden. Diese .xml Datei beinhaltet neben Informationen zum internen Logging auch die Spezifikation an welchen Endpoint die UDP-Pakete übermittelt werden sollen. In der vorliegenden Konfiguration sind gemäß Listing 7.6 eine Broadcast-Adresse sowie ein mit dem Plugin übereinstimmender Port angegeben. Die Verwendung einer Broadcast-Adresse hat den Vorteil, auch auf anderen Computern innerhalb des Simulator-Netzwerks die Daten empfangen und damit auch mitansehen zu können, was innerhalb des Simulators dargestellt wird. Voraussetzung hierfür ist nur eine Installation des InstantPlayers samt Plugin sowie der entsprechenden X3D-Datei auf einem beliebigen Computer innerhalb des selben Netzwerks.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <VisionConfig>
3    <!-- Use broadcast address or otherwise ip-address of IG4 -->
4    <RemoteHost>192.168.0.255</RemoteHost>
5    <!-- Specify port where data should be sent to -->
6    <Port>8856</Port>
7    <!-- Path where log file of ScenarioLib should be written -->
8    <LogFilePath>C:/local/log/</LogFilePath>
9    <!-- Name of log file -->
10   <LogFileName>visionLog</LogFileName>
11   <!-- Define log level -->
12   <LogMode>INFO</LogMode>
13 </VisionConfig>
```

Listing 7.6: *VisionInterfaceLib* Konfigurationsfile

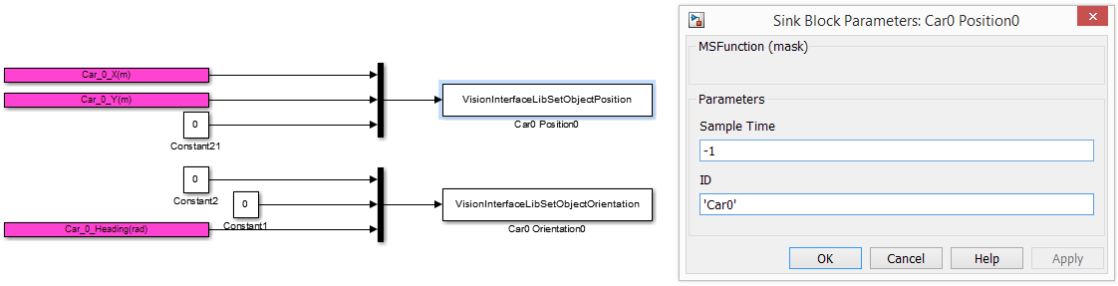

Abbildung 7.6: VisionInterface - Setzen der Visualisierungsdatem

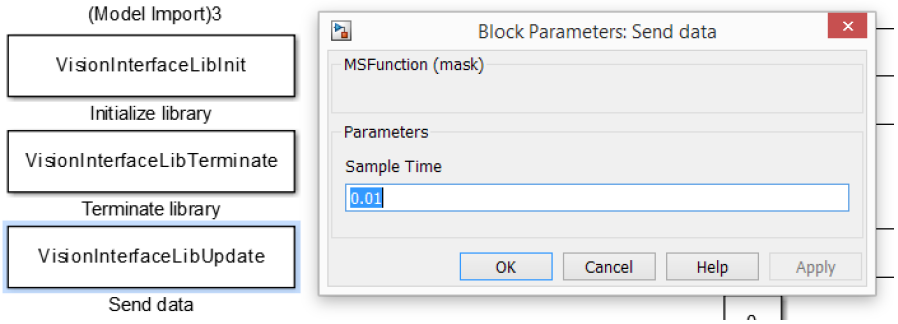

Abbildung 7.7: VisionInterface - Aufrufen der Update-Routine

Die Daten an sich werden mittels den zugehörigen Funktionsaufrufen `VisionInterfaceLibSetObjectPosition` und `VisionInterfaceLibSetObjectOrientation` gesetzt. Als ID kann 'Viewpoint' sowie 'Car0' bis 'Car19' sowie 'Pedestrian0' bis 'Pedestrian4' angegeben werden. Dementsprechend können im momentanen Stadium 20 zusätzliche Fahrzeuge sowie 5 Fußgänger visualisiert werden.
Das Datenpacket wird an das InstantIO-Plugin gesendet sobald die `update()` Methode der *VisionInterfaceLib* aufgerufen wird. Wie in Abbildung 7.7 zu sehen, geschieht dies zum gegebenen Zeitpunkt mit 100Hz.

Kapitel 8

Instrumentierung

8.1 Überblick

Das für den Simulator verwendete Fahrzeug Mini Countryman verfügt über zwei analoge Instrumenteneinheiten, eine für den Tachometer, und eine für die Geschwindigkeitsanzeige. In beiden Instrumenten sind zusätzlich Warn- als auch Informationsleuchten beziehungsweise Displays verbaut.
Aufgrund der Tatsache, dass mehrere Komponenten wie beispielsweise der Motor samt Steuergerät aus Gewichts- sowie Platzgründen aus dem Fahrzeug entfernt wurden, gestaltet es sich schwierig, den CAN-Bus des Wagens in einen voll funktionstüchtigen Zustand zu versetzen. Die Komplexität der heutigen Fahrzeuge verlangt es, dass alle Steuergeräte entsprechende Signale liefern, um die vorkommenden CAN-Knoten entsprechend zu initialisieren.
Prinzipiell ist es möglich, durch eine Restbussimulation die fehlenden Teile ohne die entsprechende Hardware einen voll funktionsfähigen CAN-Bus zu erhalten. Dies setzt jedoch einerseits die Kenntnis aller benötigten CAN-Messages, als auch detailliertes Wissen über die Funktionsweise der einzelnen Komponenten an sich, voraus.

Nachdem Automobil-Hersteller diese Informationen nur in Ausnahmefällen nach extern weitergeben, wurde zur Visualisierung des Dashboards eine Lösung basierend auf 2 Tablets entwickelt. Diese Herangehensweise hat unter anderem den großen Vorteil, dass diese jederzeit an neue Gegebenheiten sowie Aufgabenstellungen angepasst werden können, und somit zu jedem Zeitpunkt eine perfekte Anbindung an die Fahrsimulation gewährleistet werden kann. Man ist nicht auf die bestehenden Instrumente angewiesen, was insbesondere bei neuen Forschungsgebieten wie beispielsweise dem autonomen Fahren, einen nicht zu unterschätzenden Faktor darstellt.

Ein Tablet, welches direkt hinter dem Lenkrad angebracht ist, dient zur reinen Visualisierung, während im Bereich der Mittelkonsole ein zweites Tablet montiert wurde, welches auch zur Interaktion mit Fahrzeugsystemen Verwendung findet. Abbildung 8.1 zeigt die Positionierung der Tablets im Fahrsimulator.

Abbildung 8.1: Visualisierung des Dashboards durch Tablets

8.2 Features

Im momentanen Stadium bietet das Tablet zur Konfiguration der Fahrzeug-Funktionen folgende Möglichkeiten:

- Aktivieren / Deaktivieren des Tempomaten (CC - Cruise Control),
- Aktivieren / Deaktivieren des Abstandstempomaten (ACC - Adaptive Cruise Control),
- Aktivieren / Deaktivieren des Notfallbremsassistenten (AEB - Automatic Emergency Braking),
- Setzten der Sollgeschwindigkeit für CC / ACC,
- Setzten der Zeitlücke für den Abstandstempomaten.

Das virtuelle Rundinstrument stellt neben der aktuellen Geschwindigkeit des Fahrzeugs auch die eingestellte Sollgeschwindigkeit einerseits grafisch durch Abbildung eines feineren Balkens innerhalb des Kreises dar, andererseits aber auch textuell am unteren Rand des Instruments. Das Einstellen der Geschwindigkeit kann durch direktes Tappen auf die jeweilige Position des Tachos erreicht werden, oder aber durch Aktivieren der Plus- und Minus-Schaltflächen neben der eingestellten Geschwindigkeit. Tap&Hold zur schnellen Veränderung wurde ebenfalls implementiert.

Abbildung 8.2: Erläuterung der Dasboard-Komponenten

Bei eingeschaltetem Tempomaten wird das entsprechende Symbol innerhalb des Rundinstruments dargestellt.
Der Tachometer wird direkt im Code ohne Verwendung von Grafiken gezeichnet, weshalb es nahezu beliebig, ohne Qualitätsverlust, skaliert werden kann.

Im linken Bereich des Tablets sind vor allem die Einstellungen des ACC-Systems zu finden. Neben den Schaltflächen verdeutlicht eine grafische Abbildung einerseits, ob ACC aktiviert bzw. deaktiviert ist sowie die aktuell eingestellte Zeitlücke.

8.3 Interface

Für die Kommunikation der Tablets untereinander als auch mit der restlichen Simulation wird auch hier ein simples UDP-Protokoll verwendet. Zur einfacheren Handhabung soll, wie auch schon bei anderen Modulen des Simulators, möglichst viel innerhalb der Simulink-Modelle implementiert werden. Aus diesem Grund ist man auch hier bei den UDP-Paketen auf solche mit nur gleichen Datentypen angewiesen, da die UDP-Receive- sowie Send-Blöcke von Simulink keine gemischten Datentypen unterstützen. Aus diesem Grund wird jedes Datenfeld durch einen Double-Wert repräsentiert auch wenn bool-Datentypen für das jeweilige Feld ausreichen würden.
Tabellen 8.1 und 8.2 zeigen die Inhalte der Datenpakete sowie deren Bedeutung. Abbildung 8.3 verdeutlicht durch die Pfeilrichtungen zusätzlich, welche Pakete an welche Komponente des Fahrsimulator versendet werden.

Beim Receive-Interface der Simulation besteht zusätzlich die Möglichkeit, die vom Tablet gesendeten Werte durch andere zu überschreiben. Vor allem während der Testphase hat sich dies als nützlich herausgestellt, um Funktionen, welche auf dem Tablet noch nicht implementiert waren, mit Hilfe des Debug Interface aus Kapitel 9 zu testen.
Der Datenaustausch von der Simulation in Richtung Tablet geschieht in der momentanen Implementierung mit 10Hz, während umgekehrt Daten vom Tablet in Richtung Simula-

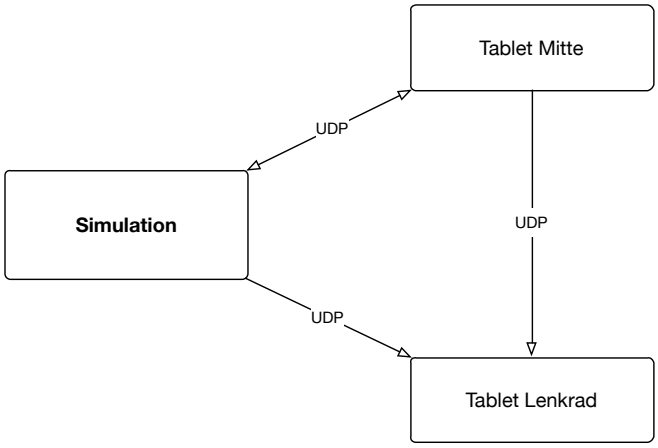

Abbildung 8.3: Dashboard Message Flow

Feld	Einheit	Typ. Bereich	Beschreibung
ccActive	-	0 - 1	Signal ob Tempomat aktiviert
accActive	-	0 - 1	Signal ob ACC aktiviert
aebActive	-	0 - 1	Signal ob Notfallbremsassistent aktiviert
setSpeed	m/s	0 - 56	Eingestellte Sollgeschwindigkeit
timeGap	-	1 - 4	Eingestellte Stufe der Zeitlücke

Tabelle 8.1: UDP Paket Tablet → Simulation / Tablet → Tablet

Feld	Einheit	Typ. Bereich	Beschreibung
velocity	km/h	0 - 56	Geschwindigkeit des Ego-Fahrzeugs
distance	m	0 - 500	Distanz zum Target
ttc	s	0 - 5	Time To Collision zum Target
aebWarning	-	0 - 1	Signal zur visuellen Kollisionswarnung
aebActive	-	0 - 1	Signal ob Notfallbremsassistent aktiv
brakePedal	-	0 - 1	Position des Bremspedals
gasPedal	-	0 - 1	Position des Gaspedals
detected	-	0 - 1	Signal ob ein Target detektiert wurde
standby	-	0 - 1	Signal ob ACC in Standby-Mode
time	s	0 - 10000	Zeitspanne in Sekunden seit Simulationsstart
gear	-	-1 - 15	Aktueller selektierter Gang
lightIndicator	-	-1 - 1	Signal ob Blinklicht aktiviert

Tabelle 8.2: UDP Paket Simulation → Tablet

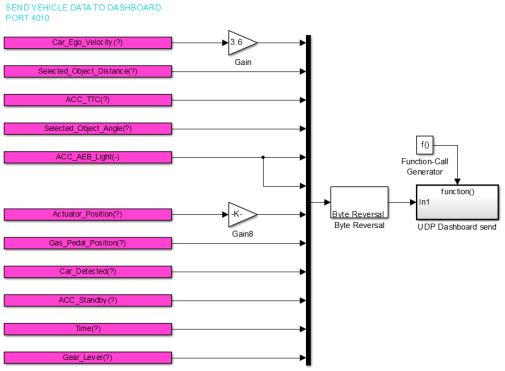

Abbildung 8.4: Simulink Send UDP-Interface

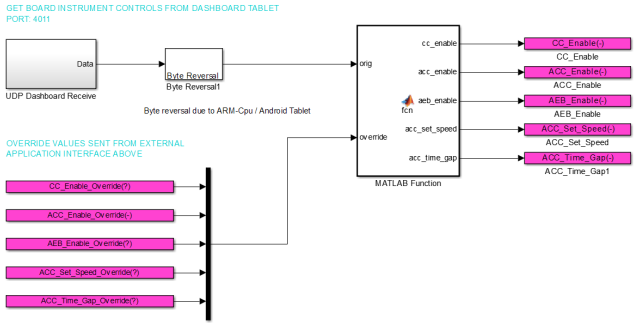

Abbildung 8.5: Simulink Receive UDP-Interface

tion nur on demand, sprich bei tatsächlicher Änderung der Daten, beispielsweise durch Aktivieren eines Buttons oder Setzen der Sollgeschwindigkeit, gesendet werden.

8.4 Aufbau

In Abbildung 8.6 ist ein vereinfachtes UML-Diagram zu sehen, welches die interne Struktur der Applikation verdeutlichen soll. Betrachtet man die Komponenten genauer, so wird ersichtlich, dass zusätzlich ein Interface integriert wurde, welches mit einer Spracherkennung in Verbindung steht. Tatsächlich wurde ein Voice-Recognition System in die Simulation integriert, welches es ermöglicht, basierend auf der in Microsoft Windows integrierten Speech Recognition API, alle auf dem Tablet inkludierten Funktionen auch über Sprache zu steuern. Erste Tests haben gezeigt, dass das System gut funktioniert und beim gegebenen Funktionsumfang so gut wie keine keine Fehlinterpretationen auftreten.

Der Aufbau der Applikation ist prinzipiell sehr einfach. Haupteinsprungspunkt stellt eine, wie unter Android üblich, Activity, nämlich die `DashboardActivity` dar, welche die drei Interfaces `SpeedIndicatorDelegate`, `UdpInterfaceDelegate` sowie `UdpInterfaceSpeechRecognizerDelegate` implementiert.
Die beiden UDP-Interfaces rufen bei Erhalt eines Datenpakets die entsprechenden Callback-Routinen der `DashboardActivity` auf.
Selbiges gilt für den `SpeedIndicatorView`. Wird die Sollgeschwindigkeit durch den Fahrer verstellt, erfolgt der Aufruf der im Interface deklarierten Methode in der `DashboardActivity`, woraufhin diese ein Datenpaket über das `UdpInterface` an die Simulation verschickt.

Werden Datenpakete aus der Simulation empfangen, werden diese dahingehend überprüft, ob bestimmte Aktionen getroffen werden müssen. Tritt der Fahrer beispielsweise trotz aktiviertem ACC auf die Bremse, so wird der Tempomat deaktiviert und wieder ein entsprechendes Datenpaket versendet. Ähnlich verhält es sich beispielsweise mit dem Standby-Modus des ACC. Steht das Fahrzeug für 3s still so wird aus der Simulation das `standby`-Feld des Datenpakets entsprechend auf 1 gesetzt, woraufhin das Tablet diese

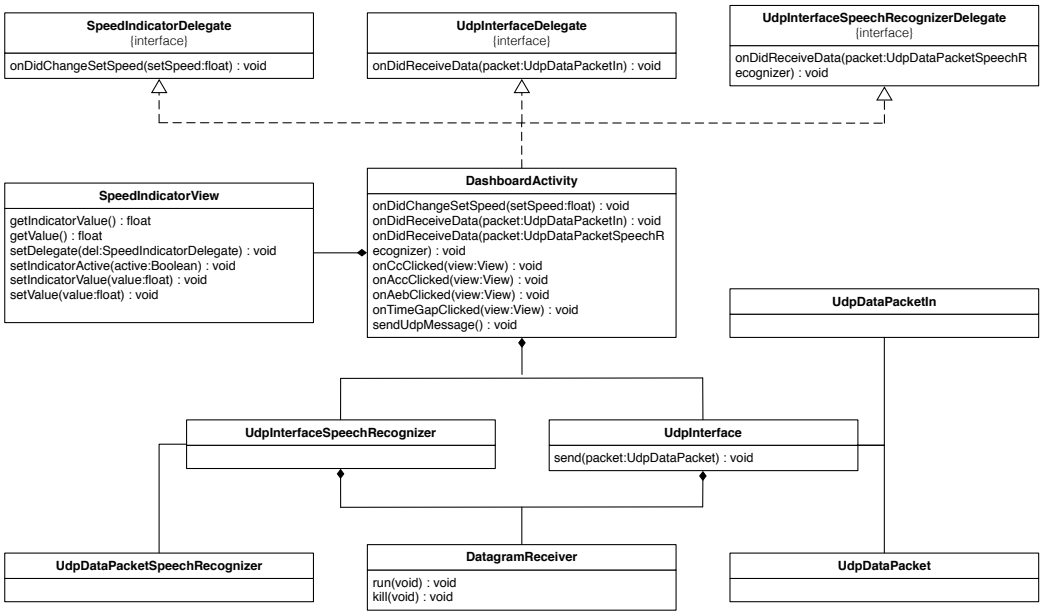

Abbildung 8.6: UML Diagram Dashboard Application

Tatsache entsprechend visualisiert. Alle bekannten komfort- sowie sicherheitsrelevanten Features der bestehenden ACC-Systemen wurden in der Applikation realisiert.

8.5 Usability Studie

Im Laufe der Inbetriebnahme des Fahrsimulators wurde eine Vorstudie mit 20 Probanden durchgeführt, in der die Systeme der Simulation gegen reale Messdaten validiert wurden. Im Rahmen dessen wurde ein umfangreicher Fragebogen erstellt, welcher unter anderem die Bedienbarkeit als auch Darstellung der Tablets miteinbezog.
Die Visualisierung der Instrumentierung mit Hilfe der Tablets erhielt bei der Vorstudie eine Gesamtnote von 4,65 auf einer Skala von 1 bis 6, wobei 6 hierbei die Bestnote repräsentiert. Von 20 verschiedenen Komponenten beziehungsweise Parametern welche durch die Versuchspersonen bewertet wurden, ist dies die dritthöchste Bewertung.

Trotz persönlicher Skepsis gegenüber Touch-basierten Bedienelementen im Automobil wurden die Tablets von den Probanden überraschend positiv aufgenommen. Nichts desto trotz sind weitere Tests mit verschiedenen Interaktionsszenarien sowie einer größeren Personenanzahl notwendig, um wirklich aussagekräftige Rückschlüsse ziehen zu können.

Kapitel 9

Debug Interface

9.1 Überblick

Während der Entwicklung von Simulationsmodulen wie beispielsweise ACC oder AEB ist es unerlässlich, zur Laufzeit online Daten aus der Simulation betrachten zu können. Prinzipiell ist es zwar möglich sich mit einem Debugger in den Simulationsprozess zu hängen, jedoch hat diese Vorgangsweise gleich mehrere fundamentale Nachteile:

- Der aus den Simulink-Modellen generierte Code ist schwer lesbar und damit die gesuchten Signale nur schwer auffindbar.
- Entwickler aus Informatik-fremden Bereichen sind oft mit den Tools zum Debuggen nicht vertraut.
- Es muss stets ein Binary in Debug-Konfiguration erstellt werden.
- Generell ist das Steppen durch dynamische Simulationsmodelle sehr aufwändig.

Auch das Loggen von Signalen ist keine zufriedenstellende Lösung, da es im Nachhinein ohne Visualisierung oft schwierig ist, die Fahrsituationen entsprechend zuzuordnen.

Unter anderem aus diesen Gründen wurde ein Tool entwickelt, welches es ermöglicht, beliebige Signale während der laufenden Simulation zu betrachten. Das Featureset der Applikation hat sich im Laufe der Zeit folgendermaßen erweitert:

- Betrachten und Plotten von Live-Daten aus der Simulation.
- Das Setzen von Parametern innerhalb der Simulation über das Tool.
- Das Überschreiben der Werte der Bedienelemente wie Lenkrad, Gas/Bremspedal sowie ACC-Einstellungen.
- Das Betrachten sowie Erstellen von Straßenverläufen samt automatischer Generierung angrenzender Fahrstreifen.
- Import von CSV-Dateien zur Erzeugung von Fahrstreifen.
- Konfiguration der Startbedingungen der Fahrzeuge.

- Generierung entsprechender XML-Szenariendefinitionen.
- Simulation des Szenarios samt Fast-Forward Funktion innerhalb der Applikation ohne auf die Gesamtsimulation angewiesen zu sein.
- Auflisten sowie Auslösen der Trigger-Events bei laufender Simulation.

Die Abbildungen 9.1 bis 9.5 zeigen ein paar dazu passende Screenshots der Applikation.

9.2 Interface

Die Kommunikation zwischen dem Programm sowie der Simulation geschieht nach den selben Prinzipien wie beispielsweise auch beim Control Interface aus Kapiel 10 oder der Instrumentierung aus Kapitel 8. Über einfache UDP-Pakete werden die Signale ausgetauscht. Nachdem sich die Daten welche zur Anzeigen gebracht werden sollen aufgrund der Bedürfnisse des Entwicklers jederzeit ändern können musste ein Weg gefunden werden, mit dem es möglich ist, ohne fix definierte Datenpakete auszukommen. Schlussendlich wurde ein Matlab-Script entwickelt welches in der Lage ist, aus Simulink heraus ein XML-Definitionsfile der zu übertragenen Signale zu exportieren. Diese Datei wiederum kann in jede beliebige externe Applikation geladen werden, um das UDP-Interface entsprechend zu konfigurieren.

Abbildungen 9.6 und 9.7 zeigen Ausschnitte jener Teile des Simulink-Modells, in welchem die Signale in den entsprechenden UDP-Blöcken münden beziehungsweise entspringen. Wesentlich ist hierbei die Benennung der Koppelleitungen. Diese muss nach dem Schema `__GRUPPE__NAME` erfolgen, um in der Debug-Applikation die Daten den entsprechenden Gruppen zuordnen zu können. Dies erleichtert besonders bei einer größeren Anzahl von Daten das Auffinden der gesuchten Signale.

Wie eine resultierende XML-Datei zur Beschreibung des UDP-Pakets aussehen könnte, ist in nachfolgendem Listing 9.1 zu sehen.

```
<?xml version="1.0" encoding="utf-8"?>
<InterfaceDefinition version="1.0">
    <SendInterface>
        <Message datatype="double" group="GENERAL">Time</Message>
        <Message datatype="double" group="VEHICLE">Car_Ego_Velocity</Message>
        <Message datatype="double" group="GENERAL">Simulation_State</Message>
        <Message datatype="double" group="BRAKE">Brake_Pedal_Torque</Message>
    </SendInterface>
    <ParameterInterface>
        <Message datatype="double" group="TEST">Amplitude</Message>
        <Message datatype="double" group="BRAKE">Brake_Torque_Gain</Message>
        <Message datatype="double" group="BRAKE">Brake_Friction_Gain</Message>
    </ParameterInterface>
</InterfaceDefinition>
```

Listing 9.1: Debug Interface UDP-Beschreibung

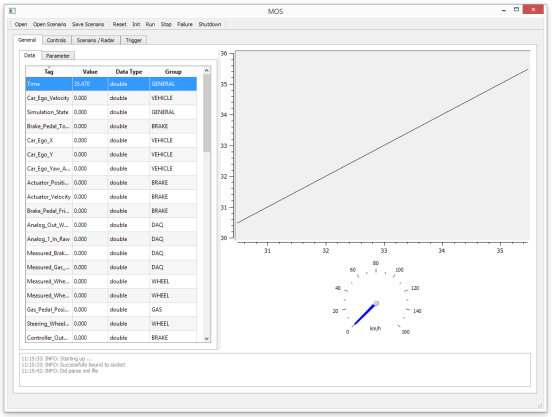

Abbildung 9.1: Liste der Signale

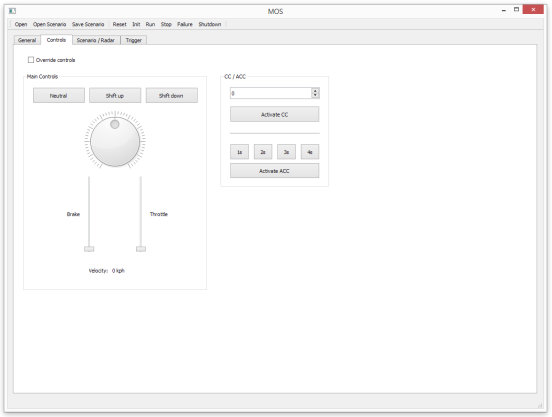

Abbildung 9.2: Überschreiben der Controls

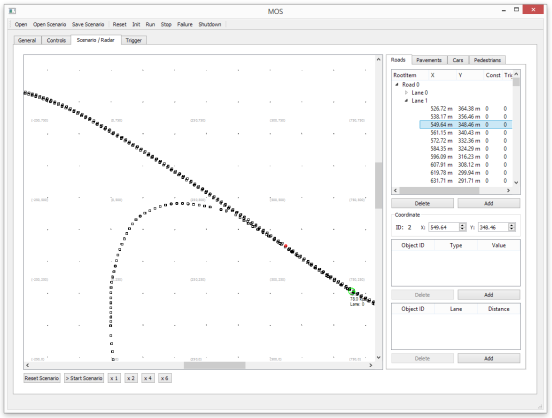

Abbildung 9.3: Ansicht der Straßen

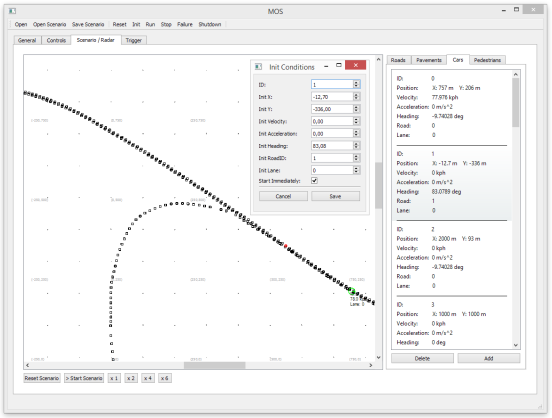

Abbildung 9.4: Konfiguration der Autos

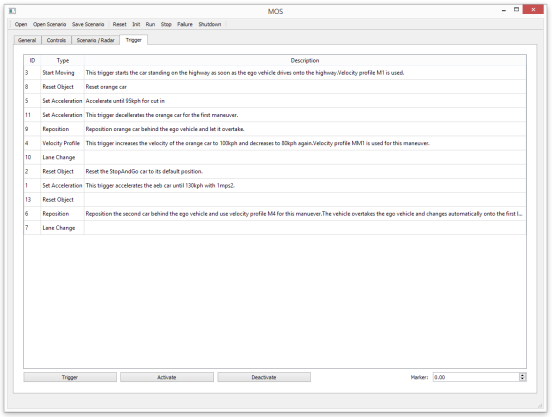

Abbildung 9.5: Liste der Triggers

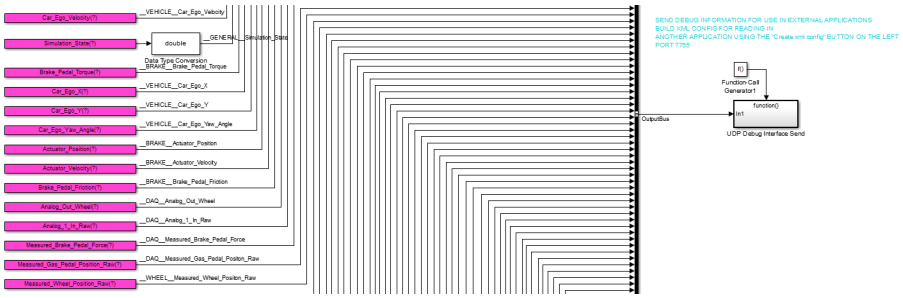

Abbildung 9.6: Debug Interface - Send Interface

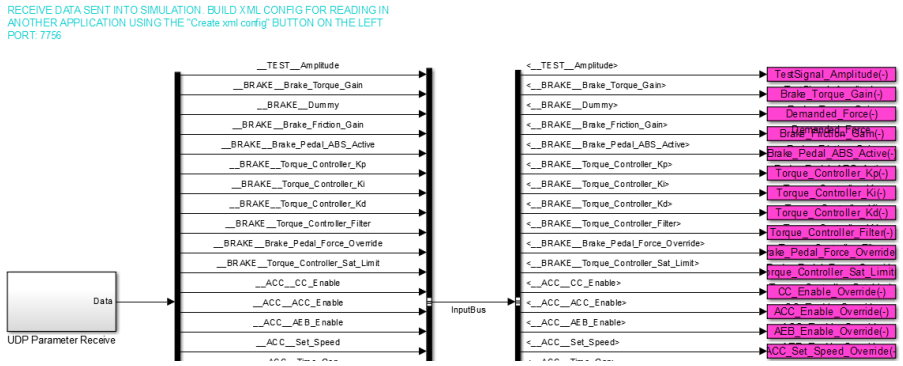

Abbildung 9.7: Debug Interface - Receive Interface

Das `datatype`-Attribut des Signals entspricht der tatsächlichen Korrespondenz im Simulink-Koppelplan, während das `group`-Attribute, wie zuvor erwähnt, aus dem Namen der Signalleitung geparst wird. Das `SendInterface` repräsentiert alle Daten, welche von der Simulation zur Verfügung gestellt werden, während das `ParameterInterface` definiert, welche Werte von extern gesetzt und somit in der Simulation empfangen werden können.
Auf die Auflistung des Skripts zur Erzeugung des XML-Files wird an dieser Stelle aufgrund der Komplexität verzichtet. Es kann jedoch vollautomatisch bei Erstellung des Realtime-Binaries ausgeführt werden, womit jederzeit die zum laufenden Modul passende Interface-Beschreibung zur Verfügung steht.
Lädt man die generierte XML-Datei in die Debug-Applikation, so steht eine Liste aller Daten, sowohl jener welche aus der Simulation geliefert werden als auch solcher, welche in die Simulation geschickt werden können, zur Verfügung. Abbildung 9.1 zeigt diesen Sachverhalt. Durch einfaches Drag&Drop lassen sich die Daten entsprechend in einem Live-Plot visualisieren.
Die Datenrate mit welcher die Signale aus der Simulation zur Verfügung gestellt werden lässt sich im Simulink-Koppelplan einfach konfigurieren. Ein Datenrate von 100Hz hat sich in der Praxis jedoch bewährt. Daten welche von der Applikation in Richtung Simulation gesendet werden, werden nur bei Änderung der Parameter upgedatet.

9.3 Aufbau

Die Applikation baut auf der Szenario-Bibliothek aus Kapitel 12 auf. Diese stellt alle Funktionalitäten zur Verfügung, welche mit dem Szenario an sich zu tun haben. Darunter fallen vor allem

- Lesen von Szenariendefinitionen samt Fahrstreifen, Fahrzeugen, Fußgängern und Triggern,
- Schreiben von Szenariendefinitionen,
- Aufbau aller inneren Datenstrukturen zur Synchronisation mit der GUI,
- Berechnung von parallelen Fahrstreifen,
- Simulation des Szenarios.

Für die Erstellung der graphischen Benutzeroberfläche finden die Qt-Bibliotheken Verwendung. Diese bieten den Vorteil, dass sich die Programme auf allen großen Platformen wie Microsoft Windows, Mac OS X sowie den verschiedensten Linux-Distributionen erstellen lassen.
Als externe Abhängigkeit wird die Qwt-Bibliothek verwendet, welches eine Vielzahl an Widgets zur Datenvisualisierung beinhaltet. Die Library wird in einer Vielzahl an Programmen eingesetzt und kann mit einer guten Dokumentation sowie vielen Beispielen aufwarten.
Nachdem die *ScenarioLib* aus Kapitel 12 keine plattformspezifischen Aufrufe beinhaltet und die Boost-Libraries auf denen die *ScenarioLib* aufbaut als auch die Qwt-Library für alle genannten Plattformen zur Verfügung stehen, ist es ohne Weiteres möglich, die Applikation auch für andere Betriebssysteme als Windows zu erstellen.

Weiters stellt Qt mehrere Funktionalitäten bereit, welche weit über das reine Erstellen von Benutzeroberflächen hinaus gehen. Darunter fallen unter anderem:

- Qt Bluetooth sowie Qt NFC zur Drahtlos-Kommunikation,
- Qt Network für verschiedenste Netzwerk-Funktionalitäten samt einer großen Unterstützung verschiedenster Protokolle wie UDP oder TCP/IP,
- Qt Multimedia für Audio- sowie Video-Wiedergabe als auch Aufnahme,
- Qt XML sowie Qt Json für einfaches Parsen und Schreiben von XML sowie Json Dateien,
- und vieles mehr.

Qt setzt bei allen Elementen welche bei der Erstellung der Benutzeroberfläche Verwendung finden auf das Model-View-Controller (MVC) Design Pattern. Daraus resultiert eine strikte Trennung zwischen den Daten und dem jeweiligen Anzeigeelement. Abbildung 9.8 verdeutlicht, wie der Informationsfluss bei diesem Entwurfsmuster definiert ist.

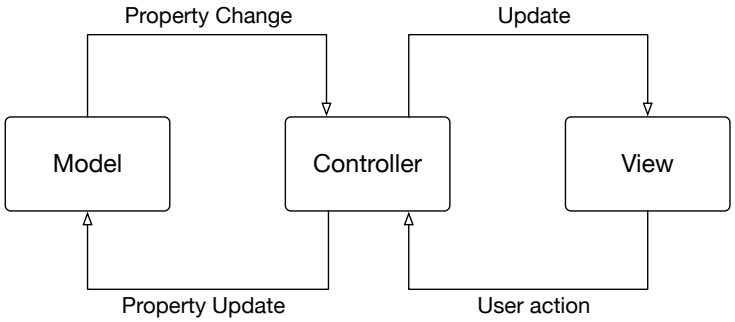

Abbildung 9.8: MVC - Model View Controller

Ähnliche Entwurfsmuster wären beispielsweise Model-View-ViewModel (MVVM) oder Model-View-Presenter (MVP). Das weit verbreitete Windows Presentation Framework (WPF), welches beispielsweise in der Simulation-Control Applikation aus Kapitel 10 Anwendung findet, setzt auf das MVVM Design-Pattern.
Zusammen mit der Interface-Beschreibungssprache sowie der umfangreichen Verwendung von Bindings wird auch hier eine strikte Trennung zwischen den Daten sowie der Visualisierung erreicht.

Nachdem das Projekt fast 40 verschiedene Klassen umfasst, welche wiederum oft von Qt-Klassen abgeleitet wurden, wird an dieser Stelle nur eine einfache schematische Übersicht über den Aufbau der Applikation gegeben.
Wie in Abbildung 9.9 zu sehen ist, fungiert die *ScenarioLib* als Datenlayer. Diese hält alle Informationen welche zur Visualisierung Szenario-relevanter Daten von Nöten sind.
Was das UDP-Interface betrifft, so werden die über das Netzwerk empfangenen Daten in von Qt zur Verfügung gestellten Standard-Models verpackt. Diese stellen bereits Schnittstellen bereit, welche perfekt an die entsprechenden View-Klassen angepasst sind.

Während im Fahrsimulator die Update-Routine der *ScenarioLib* durch den Echtzeitprozess erfolgt, so wurde zur Simulation des Szenarios innerhalb der Applikation ein eigener `Executor` samt `ScenarioPlayer` implementiert, welcher selbstständig in der Lage ist, die entsprechenden Update-Calls auszuführen. Vorteil dieser Herangehensweise ist, dass auch ohne Weiteres eine Fast-Forward Funktion implementiert werden kann. Dies bedeutet, dass die Simulation mit einem Vielfachen der üblichen Ausführungsgeschwindigkeit durchgeführt wird, und somit der Aufbau des Szenarios entsprechend beschleunigt werden kann.

Während der Entwicklung der grafischen Oberfläche hat sich gezeigt, dass das Design der *ScenarioLib* als gelungen angesehen werden kann. Es waren praktisch keine Anpassungen notwendig und auch die Integration in das Umfeld der Qt-Bibliothek war ohne Probleme möglich.
Zum Zeitpunkt der Erstellung des Debug-Interfaces war es mit den Qt-Bibliotheken noch nicht möglich, Modern UI Applications basierend auf WinRT zu entwickeln, wie es im

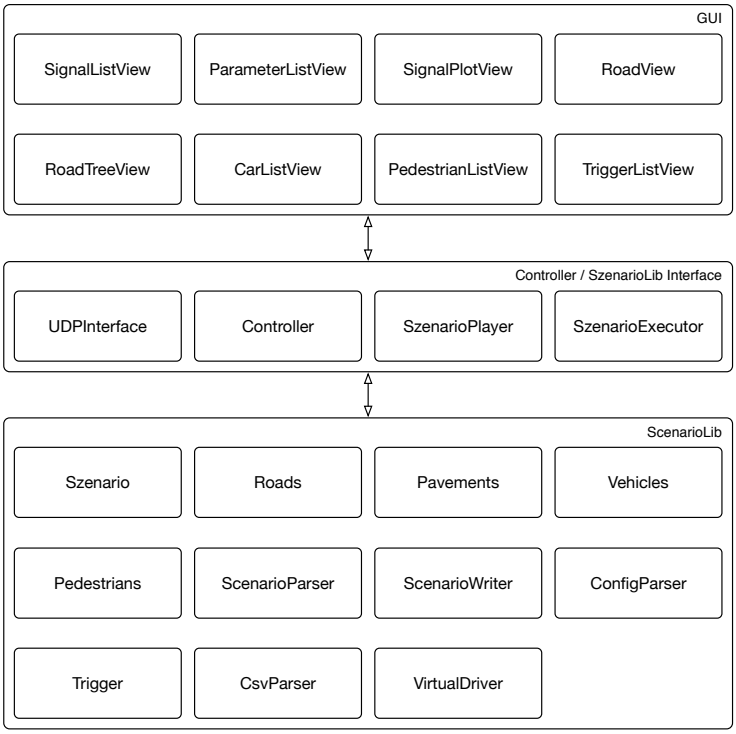

Abbildung 9.9: Applikation - Überblick

nachfolgenden Kapitel 10 bei der Simulation-Control Applikation getan wird. Aus diesem Grund war es für das die Simulation-Control notwendig, bereits bestehende Funktionalitäten in C# zu reimplementieren.
Ab Version 5.4 unterstützt jedoch Qt WinRT, weshalb es empfehlenswert ist, in zukünftigen Projekten beide Applikationen zusammenzuführen und so nur eine gemeinsame Codebasis pflegen zu müssen.

Kapitel 10

Simulation Control

10.1 Features

Damit der Operator den Fahrsimulator auch alleine, aus dem Inneren des Fahrzeugs heraus, bedienen kann, wurde eine Touch-Applikation basierend auf einem Windows-Tablet entwickelt.
Diese erfüllt zum momentanen Zeitpunkt folgende Anforderungen:

- Starten / Stoppen der Echtzeitumgebung,
- Starten / Stoppen der Simulation,
- Umstellen der Visualisierung von Sommer- auf Winterszenario,
- Konfiguration des verwendeten Reibwertkoeffizienten der Straße,
- Konfiguration spezieller experimenteller Features des AEB,
- Auslösen und Aktivieren sowie Deaktivieren der Trigger, welche in Kapitel 12.4 beleuchtet werden.

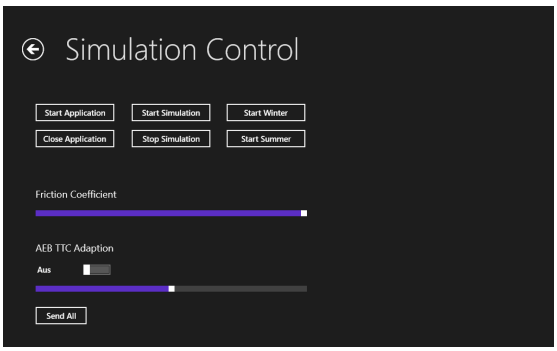

Abbildung 10.1: Control Interface - Allgemeine Aktionen

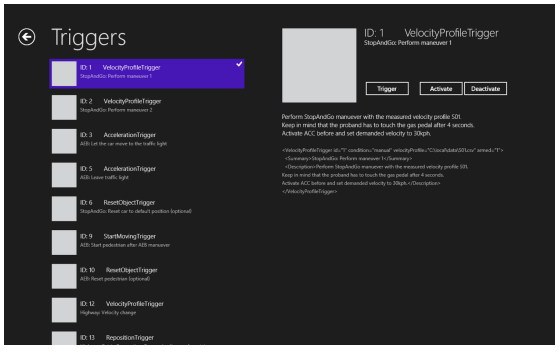

Abbildung 10.2: Control Interface - Steuerung der Events

10.2 Aufbau

Die Applikation kann beliebige Szenariendefinitionen, wie in Kapitel 12.3 beschrieben, öffnen sowie die darin enthaltenen Informationen extrahieren. Aus dem Definitionsfile werden die zur Verfügung stehenden Trigger geparst und wie in Abbildung 10.2 dargestellt, zur Anzeige gebracht. Dies ermöglicht es dem Operator, die definierten Events jederzeit händisch auszulösen.

Die *ScenarioLib* aus 12.2 verfügt ja bereits über einen vollwertigen Parser für die Szenariendefinition. Nachdem die Simulation-Control Applikation auf C# basiert, bestand entweder die Möglichkeit, einen Wrapper um die *ScenarioLib* zu erstellen und diesen dann in C# zu verwenden, oder aber den Parser für das XML-File in C# zu implementieren.
Nachdem das Programm nur die ID, das Hidden-Attribut, die Summary sowie Description lesen muss, war der Aufwand einer Neuerstellung des Parsers in C# mit deutlich weniger Arbeit verbunden als das Erstellen eines Wrappers. Der stark reduzierte Parser konnte in nur 50 Codezeilen implementiert werden.
Ein weiterer Aspekt welcher für eine Umsetzung in C# spricht ist, dass aufgrund der nicht vorhandenen externen Abhängigkeiten die Entwicklung prinzipiell einfacher von Statten geht. Insbesondere bei weniger erfahrenen Personen kann dies einen Vorteil bedeuten.

```
1  <VelocityProfileTrigger id="1" condition="manual"
       velocityProfile="C:\local\data\S01.csv" armed="1">
2    <Summary>
3      StopAndGo: Perform maneuver 1
4    </Summary>
5    <Description>
6      Perform StopAndGo manuever with the measured velocity profile S01.
7      Keep in mind that the proband has to touch the gas pedal after 4 seconds.
8      Activate ACC before and set demanded velocity to 30kph.
9    </Description>
10 </VelocityProfileTrigger>
```

Listing 10.1: VelocityProfileTrigger

Listing 10.1 und Abbildung 10.3 zeigen, wie die Daten aus dem Definitionsfile mit den Elementen des User-Interface zusammenhängen. Summary sowie Description können vom Operator nach Belieben vergeben werden und dienen nur zur Hilfestellung während dem Betrieb.
Wäre im VelocityProfileTrigger zusätzlich das Attribut `hidden="true"` definiert, so würde das Event in der Liste nicht aufscheinen.

10.3 Interface

Auch hier kommt ein simples UDP-Interface für die Kommunikation zwischen Simulation und Tablet zum Einsatz. Wie zuvor bereits mehrfach angedeutet, ist man aufgrund der Verwendung von Matlab/Simulink und den darin enthaltenen UDP-Blöcken auf Pakete welche nur gleiche Datentypen beinhalten, eingeschränkt.

Abbildung 10.3: Control Interface - Trigger Details

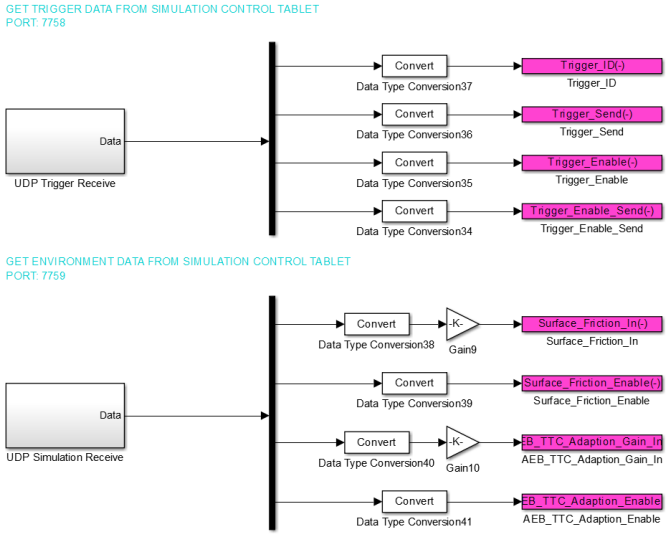

Abbildung 10.4: Control Interface - Simulink Interface

Über das Interface ist es möglich, einen Trigger manuell auszulösen bzw. dessen automatische Triggerung zu aktivieren bzw. zu deaktivieren. Tabelle 10.1 zeigt das entsprechende Datenpaket. Die Übernahme der Werte in die Simulation erfolgt bei steigender Flanke des entsprechenden Felds mit dem Zusatz `Send`. Sind mehrere Trigger mit der selben ID konfiguriert, so werden diese zeitgleich ausgelöst.

Tabelle 10.2 repräsentiert das Datenpaket, mit dem nicht Trigger-spezifische Einstellungen getroffen werden können. So ist es momentan möglich, zur Laufzeit den Reibungskoeffizienten der Straße zu konfigurieren, um so verschiedenste Straßenbeschaffenheiten nachstellen zu können. Zusätzlich wurde in das Modell des Notfallbremsassistenten ein Parameter integriert welcher es ermöglicht, das Verhalten an den Straßenzustand anzupassen. Dieses Feature bzw. dessen Parameter können ebenfalls über das Control-Tablet konfiguriert werden.

Feld	Einheit	Typ. Bereich	Beschreibung
triggerId	-	0 - 1	ID des Triggers
triggerSend	-	0 - 1	Ausführen des Triggers bei steigender Flanke
triggerEnable	-	0 - 1	Signal ob Trigger aktiviert/deaktiviert werden soll
triggerEnableSend	-	0 - 1	Übernehmen des triggerEnable Felds

Tabelle 10.1: UDP Packet Trigger

Feld	Einheit	Typ. Bereich	Beschreibung
surfaceFriction	-	0.1 - 1	Wert für Reibungskoeffizienten der Straße
surfaceFrictionSend	-	0 - 1	Übernehmen des Reibungskoeffizienten
aebTtcAdaptionGain	-	0 - 1	Wert zur Anpassung der TTC des AEB
aebTtcAdaptionGainSend	-	0 - 2	Übernehmen des aebTtcAdaptionGain Felds

Tabelle 10.2: UDP Packet Simulation

Wie auch schon beim Datenpaket der Trigger werden alle Felder erst bei steigender Flanke des entsprechenden `Send`-Felds übernommen. An dieser Stelle sei auch darauf hingewiesen, dass es sich beim momentanen Stand um eine reine One-Way-Kommunikation vom Tablet in Richtung Simulation handelt. Für zukünftige Aufgabenstellungen ist es sicher angebracht, die Schnittstellen entsprechend einer Two-Way-Kommunikation anzupassen um garantieren zu können, das angeforderte Änderungen oder Events von der Simulation auch tatsächlich umgesetzt wurden.

10.4 RemoteControlServer

Wie bereits erwähnt ist es auch möglich, über das Tablet die Realtime-Umgebung PTWinSim bzw. die Simulation an sich zu Starten sowie zu Stoppen. Weiters ist es möglich, die Szenerie von Sommer- auf Winter und wieder zurück zu wechseln.
Nachdem sowohl PTWinSim als auch die Visualisierung nicht über die entsprechenden Schnittstellen verfügen um die jeweiligen Aktionen über das Netzwerk direkt starten zu können musste eine Applikation erstellt werden, welche auf dem Simulationsrechner im Hintergrund läuft und die Kommandos vom Control-Tablet empfängt und entsprechend umsetzt.

Umstellen der Visualisierung: Um die Szenerie umzustellen stehen auf dem Simulationsrechner Batch-Skripte zur Verfügung, welche sich bei Aufruf über `PsExec` auf dem Master-IG einloggen um dort wiederum Batch-Skripte auszuführen. Die Skripte beenden die InstantReality-Umgebung und starten diese mit der gewünschten Szenerie neu. Der RemoteControl Server führt diese Batch-Skripte bei der entsprechenden Anforderung aus.

Steuern von PTWinSim: Auch das Starten von PTWinSim geschieht über den Aufruf einer Batch-Datei. Diese war ohnehin bereits vorhanden, da beim Start zusätzliche

Feld	Type	Beschreibung
openApplication	bool	Starten von PTWinSim
startSimulation	bool	Starten der Simulation
stopSimulation	bool	Stoppen der Simulation
closeApplication	bool	Beenden von PTWinSim
startSummer	bool	Starten der Sommer-Szenerie
startWinter	bool	Starten der Winter-Szenerie

Tabelle 10.3: UDP Packet RemoteControlServer

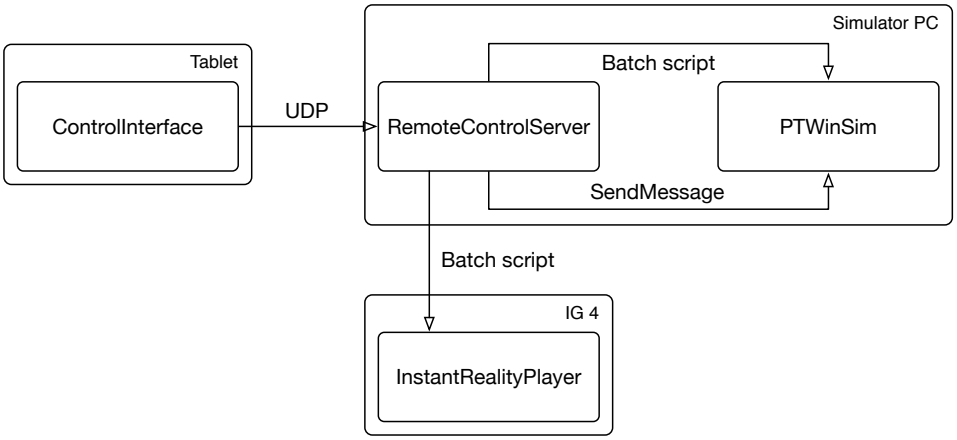

Abbildung 10.5: Control Interface - Struktur RemoteControlServer

Pfad-Variablen zu den benötigten Bibliotheken ergänzt werden. Das Stoppen sowie Steuern der Simulation geschieht über die Message functions welche vom Betriebssystem zur Verfügung gestellt werden. Insbesondere mit der `SendMessage`-Function ist es möglich, verschiedenste Events an bestimmte Fenster zu senden.

Für die Kommunikation zwischen Server und Client wird die selbe einfache Struktur wie auch für alle anderen Applikationen eingesetzt, um die bestehenden UDP-Interface-Klassen weitestgehend wiederverwenden zu können. Das Datenpaket welches vom Tablet gesendet wird ist in Tabelle 10.3 angeführt, wobei die jeweilige Aktion bei `true` des jeweiligen Feldes ausgeführt wird.
In Abbildung 10.5 ist schlussendlich noch einmal die Grobstruktur, wie die Komponenten an den RemoteControlServer angebunden sind, dargestellt.

Kapitel 11

Akustik

11.1 Einleitung

Neben der Visualisierung und den Force-Feedback-Devices ist die Akustik einer jener Punkte, welcher maßgeblich mitverantwortlich für den Gesamteindruck des Fahrsimulators ist. Nachdem in der momentanen Ausbaustufe noch keine Bewegungsplattform implementiert ist, nimmt die Bedeutung der Soundsimulation noch zusätzlich zu.
Realisiert wurde das System in Zusammenarbeit mit der AVL List GmbH mit Hilfe eines Tools welches am Institut für Elektronische Musik und Akustik (IEM) an der Universität für Musik und darstellende Kunst Graz entwickelt wurde.

Das Fundament der Akustik-Simulation stellt das Open-Source Projekt Pd-extended dar. Pd ist eine graphische Echtzeit-Entwicklungsumgebung zur Verarbeitung von graphischen, aber insbesondere Audio-Prozessen und ist unter allen gängigen Betriebssystemen lauffähig.
Der Einsatzbereich erstreckt sich von der Sound-Effekt-Generierung, über VeeJaying bis hin zur Audioanalyse. Die Oberfläche entspricht in etwa dem was man von Simulink gewohnt ist, mit dem Unterschied, dass Objekte sowie deren Verbindungen zur Laufzeit hinzugefügt und geändert werden können. Pd unterscheided nicht zwischen dem Erstellen des Programms und dessen Ausführung und läuft im Prinzip permanent, was auch der Grund dafür ist, weshalb es oft bei Live-Performances von Künstlern Verwendung findet. Über die Jahre hinweg hat die Community eine Vielzahl an zusätzlichen Bibliotheken entwickelt, sei es zum Zwecke der Einbindung von unterschiedlichen Codecs oder der Erweiterung der Video- und Netzwerk-Funktionalitäten.

11.2 IEM Engine Sound Tool

Wie zu Beginn des Kapitels erwähnt, wurde vom IEM ein Tool zur Generierung von Motorengeräuschen entwickelt. Abbildung 11.1 zeigt die zugehörige Konfigurationsmaske. Das Tool basiert auf Matlab und stellt ein Interface für das darunter liegende Pd-Modell zur Verfügung. Es stellt eine Vielzahl von Analyse- sowie Synthese-Funktionen zur Verfügung. Durch Laden unterschiedlicher Profile ist es möglich, innerhalb weniger

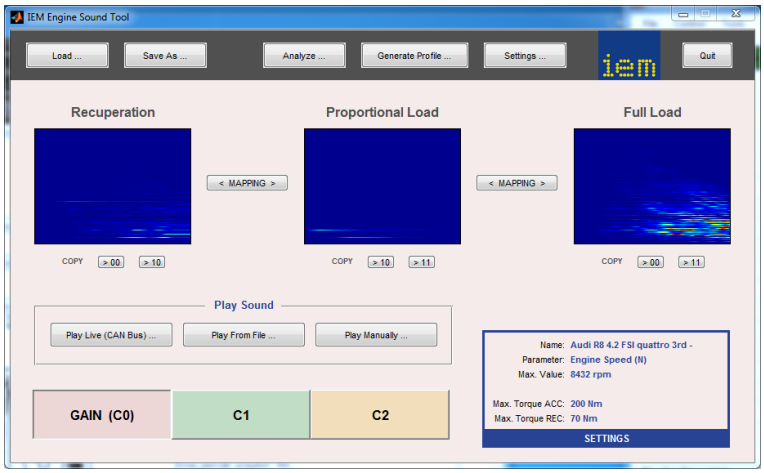

Abbildung 11.1: IEM Engine Sound Tool

Sekunden das Motorengeräusch beliebig auszutauschen. Es wird der Betrieb von Elektro- als auch Verbrennungsmotoren unterstützt.

Neben der Simulation des Motorengeräusches werden zusätzlich Eindrücke aus der Umwelt simuliert. Dies betrifft vor allem Abroll- sowie Windgeräusche. Diese zusätzlichen Effekte sind unumgänglich, um einen realistischen Gesamteindruck zu gewährleisten. Durch die unter dem Fahrersitz sowie im Fußraum verbauten Tieftöner werden dadurch, trotz fehlender Bewegungsplattform, Vibrationen in die Fahrgastzelle eingebracht, welche zusätzlich zu einer realistischen Wahrnehmung der Simulation beitragen.

Neben dem Ego-Fahrzeug werden auch andere umherfahrende Target-Fahrzeuge simuliert. Anhand der Abstände sowie der Relativgeschwindigkeiten werden Motor- und Umweltgeräusche als auch der charakteristische Dopplereffekt simuliert.
Um auch generische Sounds wie beispielsweise Warnungen oder Blinkergeräusche abspielen zu können, wurde zusätzlich ein Interface implementiert um beliebige Soundfiles abspielen zu können. Auf diese Weise ist zusätzliche Flexibilität gegeben ohne das Pd-Modell ändern zu müssen.

Abbildungen 11.2 und 11.3 zeigen die erste sowie zweite Ebene des Pd-Modells der Akustiksimulation. Vor allem in Abbildung 11.3 wird schnell einer der größten Schwachpunkte der Pd-Entwicklungsumgebung deutlich. Die graphische Oberfläche an sich erfüllt leider nur die notwendigsten Funktionalitäten. Methoden zur Strukturierung des Koppelplans, wie man es beispielsweise von Simulink her gewohnt ist, sind praktisch nicht vorhanden. Auch Masken zur bequemen Eingabe von Parametern stehen nicht zur Verfügung. Entsprechend schnell verliert das Modell damit auch jegliche Struktur und Beziehungen können nur schwer nachvollzogen werden. Das Engine-Modul beinhaltet noch eine Vielzahl an weiteren Unterebenen. Auf dessen Darstellung wird an dieser Stelle jedoch verzichtet.

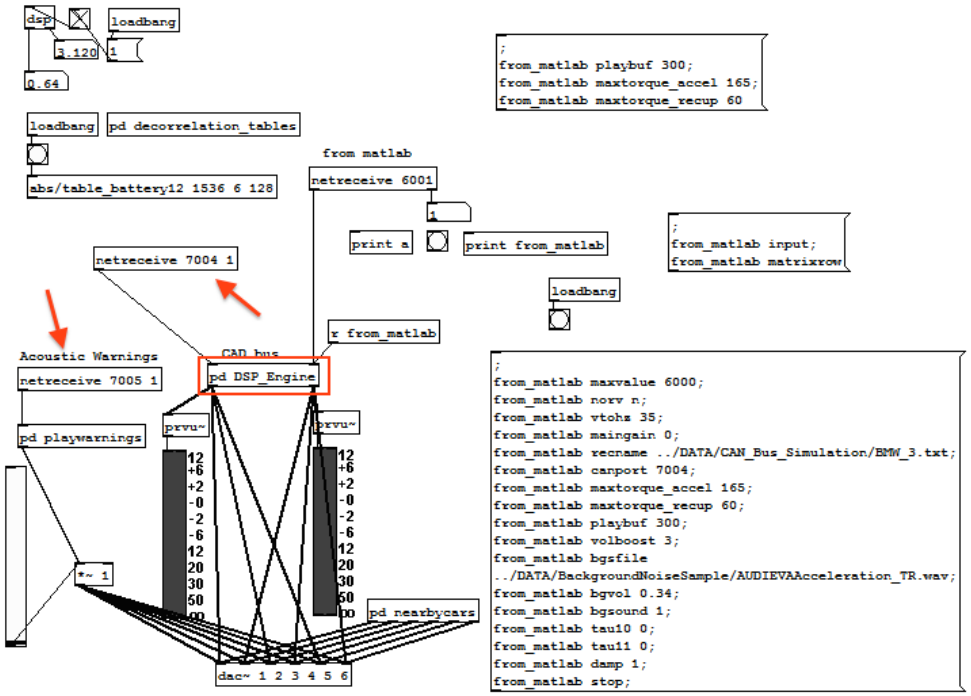

Abbildung 11.2: Einsprungspunkt des Pd-Modells

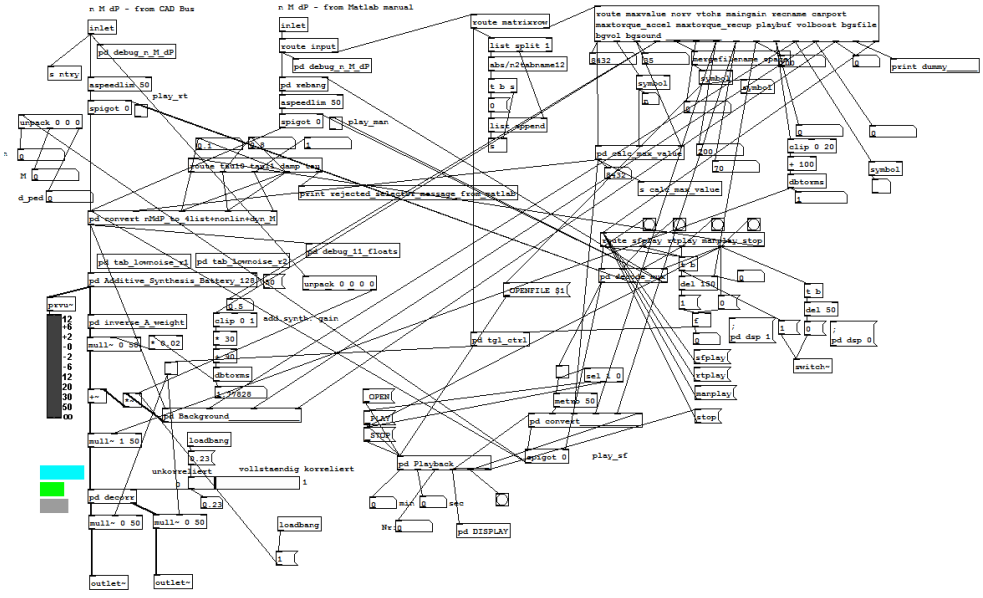

Abbildung 11.3: Pd Motormodell

	Quelle	Wertebereich
Motordrehzahl	VSM	0 - 1
Motordrehmoment	VSM	-1 - 1
Gaspedalposition	ACC/AEB-Modul	0 - 1
Verstärkungsfaktor Umwelt	VSM	0 - 9

Tabelle 11.1: Größen zur Akustiksimulation des Ego-Fahrzeuges

	Quelle	Wertebereich
AEB Warnung	ACC/AEB-Modul	steigende Flanke
Platzhalter 1	-	steigende Flanke
Platzhalter 2	-	steigende Flanke
Platzhalter 3	-	steigende Flanke
Platzhalter 4	-	steigende Flanke
Platzhalter 5	-	steigende Flanke

Tabelle 11.2: Größen für generische Sounds

11.3 Interface

Die farblichen Hervorhebungen in Abbildung 11.2 zeigen einerseits die Einbindung der zweiten Ebene aus Abbildung 11.3, andererseits die Einbindung der `netreceive` Blöcke um eine UDP-Kommunikation zum Übertragen der von der Akustik-Simulation benötigten Daten gewährleisten zu können. Dem Pd-Modell werden aus der Fahrsimulation die Größen aus den Tabellen 11.1 und 11.2 zur Verfügung gestellt.
Die Target-Fahrzeuge werden in je einem separaten Modell simuliert, welches vom Aufbau her jedoch sehr ähnlich jenem des Ego-Fahrzeuges ist. Zur Berechnung des Doppler-Effekts werden jedoch mehr Daten benötigt als dies für das Ego-Fahrzeug der Fall ist. Tabelle 11.3 listet die zugehörigen Größen auf.
Eine Besonderheit hierbei ist, dass die motorbezogenen Daten nicht absolut, sondern normiert erwartet werden. Aus diesem Grund müssen die Motorparameter des simulierten Fahrzeugmodells mit jenem der Akustiksimulation übereinstimmen. Anderenfalls kommt es zu einer nicht realitätsgetreuen Nachbildung der Motorgeräusche. In Abbildung 11.1 sind im rechten unteren Bereich jeweils die verwendeten Parameter des momentan geladenen Modell-Setups dargestellt. Maximaldrehzahl, maximales Drehmoment sowie maximales Drehmoment während dem Rekuperieren müssen übereinstimmen.

Pd verwendet zur Netzwerk-Kommunikation das eigens entwickelte Protokoll namens FUDI. Es ist ein String-basiertes Protokoll, dessen Nachrichten üblicherweise durch Semikolons abgeschlossen werden.
Jede Nachricht besteht aus so genannten Atoms, allgemein ausgedrückt aus Werten, die durch Leerzeichen getrennt sind. Als Leerzeichen können hierbei das Leerzeichen an sich (ASCII 32), ein Tab (ASCII 9) oder das Newline Character (ASCII 10) verwendet werden.

	Quelle	Wertebereich
Motordrehzahl	Scenario-Modul	0 - 1
Motordrehmoment	Scenario-Modul	-1 - 1
Gaspedalposition	Scenario-Modul	0 - 1
Abstand longitudinal	Radar-Modul	0 - 9999
Abstand lateral	Radar-Modul	0 - 9999
Abstand absolut	Radar-Modul	0 - 9999
Relativgeschwindigkeit longitudinal	Radar-Modul	0 - 9999
Relativgeschwindigkeit lateral	Radar-Modul	0 - 9999
Relativgeschwindigkeit absolut	Radar-Modul	0 - 9999
Verstärkungsfaktor Motor	Scenario-Modul	0 - 9
Verstärkungsfaktor Umwelt	konstant	0 - 9

Tabelle 11.3: Größen zur Akustiksimulation der Target-Fahrzeuge

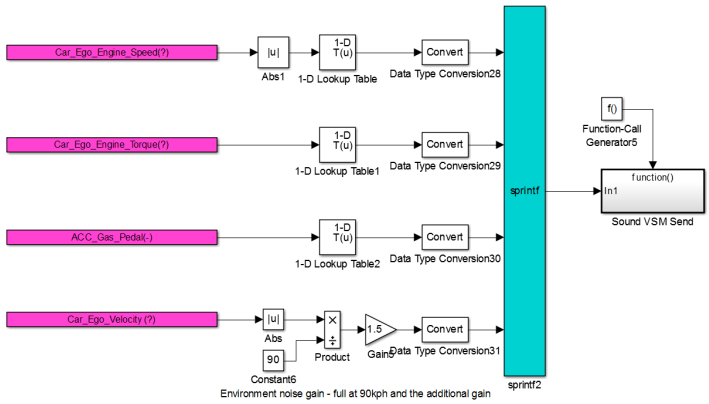

Abbildung 11.4: Pd Interface Ego-Fahrzeug

Während die Standard Simulink Toolboxen keine String-Behandlung ermöglichen, stellt jedoch die @Source Toolbox aus Kapitel 3.1 diesbezüglich eine gewisse Funktionalität, wenn auch eingeschränkt, zur Verfügung. Mittels des sprintf-Blocks kann ein entsprechender Datenstream erzeugt werden welcher dann über UDP an Pd übermittelt werden kann. So wird im sprintf-Block in Abbildung 11.4 durch folgende Anweisung eine entsprechende FUDI-konforme Nachricht erstellt und anschließend über UDP übertragen.

```
1 %04.4f %04.4f %04.4f %04.4f;\n
```

Listing 11.1: Format zur Generierung der FUDI-konformen Message

Kapitel 12

Szenariengenerierung - ScenarioLib

12.1 Einleitung

Wesentlich für den Betrieb eines Fahrsimulators ist die Simulation von dynamischen Verkehr. Ohne diesen wäre das Einsatzgebiet sehr stark eingeschränkt.
Da der Schwerpunkt des Simulators bei der Untersuchung von Fahrassistenzsystemen und dem autonomen Fahren und weniger bei der Entwicklung der Dynamik des Fahrzeugs liegt, war von Anfang an klar, dass die Verkehrssimulation ein wesentlicher Punkt bei der Entwicklung werden wird.

Im Rahmen des MueGen-Projekts wurden verschiedene Untersuchungen mit dem ACC eines Audi A6 sowie dem Notfallbremsassistenten einer Mercedes S-Klasse durchgeführt. Die Testläufe welche im Zuge dessen bewerkstelligt wurden, galt es nun im Simulator mit den selben Probanden welche auch im Realversuch teilgenommen haben, exakt nachzustellen und die objektiven und subjektiven Unterschiede herauszufinden. Dadurch soll eine Basis geschaffen werden mit der es möglich ist, Rückschlüsse auf die Realität zu treffen. Nähere Informationen zum MueGen-Projekt können Kapitel 1.2 zu Beginn dieser Arbeit entnommen werden.

Die Realversuche umfassten unter anderem folgende Manöver, welche anschließend auch im Simulator entsprechend den aufgenommen Messdaten nachgestellt wurden.

- AEB: Auffahren auf stehendes Fahrzeug
- AEB: Auffahren auf bewegten Fußgänger
- ACC: Stop&Go mit verschiedenen Zeitlücken
- ACC: Auffahren auf Fahrzeug mit konstanter Geschwindigkeit mit verschiedenen Zeitlücken
- ACC: Folgefahrt variabel mit verschiedenen Zeitlücken
- ACC: Automatische Überholen mit verschiedenen Zeitlücken

- ACC: Cut-In mit verschiedenen Zeitlücken

12.2 Scenario Library

Um diese Manöver in der Simulation zur Verfügung stellen zu können, wurde in C++ eine Bibliothek implementiert, welche exakt an die gefahrenen Manöver angepasst werden kann. Wie in Kapitel 3 beschrieben, erfolgte auch hier die Integration in Matlab/-Simulink um auch Nicht-Programmierern die Interaktion in einer gewohnten Arbeitsumgebung zu ermöglichen. Selbstverständlich fügt sich die Bibliothek nahtlos in den Code-Generierungsprozess mit Hilfe des Simulink Coder ein.
Die Definition des Szenarios erfolgt in einem .xml File, welches nachfolgend in Kapitel 12.3 genau erläutert wird. Dieselbe Datei wird auch noch von anderen Applikationen verwendet. Mehr dazu kann man in den Kapiteln 10 und 9 finden.

Mit zu definierenden Triggern ist es möglich, bestimmte Events in der Simulation wie beispielsweise einen Fahrstreifenwechsel, einen Bremsvorgang oder das Einspielen eines vorgegebenen Geschwindigkeitsprofils, auszulösen. Dies kann vollautomatisch aufgrund von bestimmten Bedingungen oder manuell durch den Operator erfolgen. Die Definition erfolgt ebenfalls im .xml File des Szenarios und dieselben Datensätze werden auch vom Control-Tablet des Operators geladen um die Aktionen auslösen zu können. Nähere Infos dazu sind im nachfolgenden Kapitel 12.4 zu finden.

Bei der Initialisierung wird der Bibliothek ein Pfad zu einem Config-File übergeben, welches beispielhaft in Listing 12.1 dargestellt ist. Dort ist unter anderem in Zeile 4 ein Pfad zu einer Szenario-Definition definiert.
Im nächsten Schritt wird dann dieses .xml geparst und intern die entsprechenden Objekte instanziert. Ist dies geschehen, so müssen die Fahrzeuge und Fußgänger regelmäßig einen Update-Call erhalten, um deren internen Zustand entsprechend zu aktualisieren. Abbildung 12.1 veranschaulicht den prinzipiellen Ablauf.
Das Intervall der Update-Calls muss exakt mit jenem übereinstimmen, mit welchem die Update-Routine aus dem Simulink-Modell heraus aufgerufen wird. Dies ist wesentlich, da die Integratoren der *ScenarioLib*, welche aus den Beschleunigungen Geschwindigkeiten und schlussendlich Positionen berechnen, auf das korrekte Zeitintervall angewiesen sind. Zur Initialisierung ist das Update-Intervall im Config-File in Zeile 12 angegeben. Es kann jedoch auch jederzeit dynamisch zur Laufzeit verändert werden.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <ScenarioConfig>
3    <!-- Path of Scenariodefinition -->
4    <ScenarioFilePath>/path/to/scenario_definition.xml</ScenarioFilePath>
5    <!-- Path where log file of ScenarioLib should be written -->
6    <LogFilePath>C:/local/log/</LogFilePath>
7    <!-- Name of log file -->
8    <LogFileName>scenarioLog</LogFileName>
9    <!-- Define log level -->
10   <LogMode>INFO</LogMode>
```

```
    <!-- Defines how often update-routine is called -->
    <UpdateInterval>0.002</UpdateInterval> <!-- in s -->
</ScenarioConfig>
```

Listing 12.1: *ScenarioLib* Konfigurationsfile

Das C-Interface welches in Simulink verwendet wird, ist im nachfolgenden Listing 12.2 aufgeführt. Intern steht natürlich eine ungleich größere Anzahl an Funktionalität zur Verfügung. Das Interface implementiert jedoch nur jene Features welche auch tatsächlich aus den Simulink-Modellen heraus angesprochen werden.

```
#pragma once

#ifdef __cplusplus
extern "C" {
#endif
    // Enable Trigger with given ID
    __declspec(dllexport) void enableTrigger(const unsigned int id, const bool enabled);
    // Get Car with given ID
    __declspec(dllexport) Car* getCar(unsigned int id);
    // Get Pedestrian with given ID
    __declspec(dllexport) Pedestrian* getPedestrian(unsigned int id);
    // Get estimated Lane for given Coordinates
    __declspec(dllexport) int getEstimatedLaneId(const double x, const double y, const double
        radius);
    // Get estimated Road for given Coordinates
    __declspec(dllexport) int getEstimatedRoadId(const double x, const double y, const double
        radius);
    // Init Library with path to config file
    __declspec(dllexport) void init(const char* configPath);
    // Set Information from Vehicle Simulation - used e.g. for distance-depended trigger
    __declspec(dllexport) void setEgoCar(const Car* car);
    // Stop ScenarioLib
    __declspec(dllexport) void stop(void);
    // Perform trigger with given ID or multiple trigger with same ID
    __declspec(dllexport) void trigger(const unsigned int id);
    // Perform update step iof ScenarioLib
    __declspec(dllexport) void update(void);

#ifdef __cplusplus
}
#endif
```

Listing 12.2: *ScenarioLib* C-Interface

enableTrigger: Dieser Aufruf erlaubt es gewisse Trigger in der Simulation zu aktivieren oder zu deaktivieren. Dies ist vor allem dann nützlich, wenn automatische Trigger, welche nicht vom Operator händisch sonder vollautomatisch ausgelöst werden, deaktiviert werden

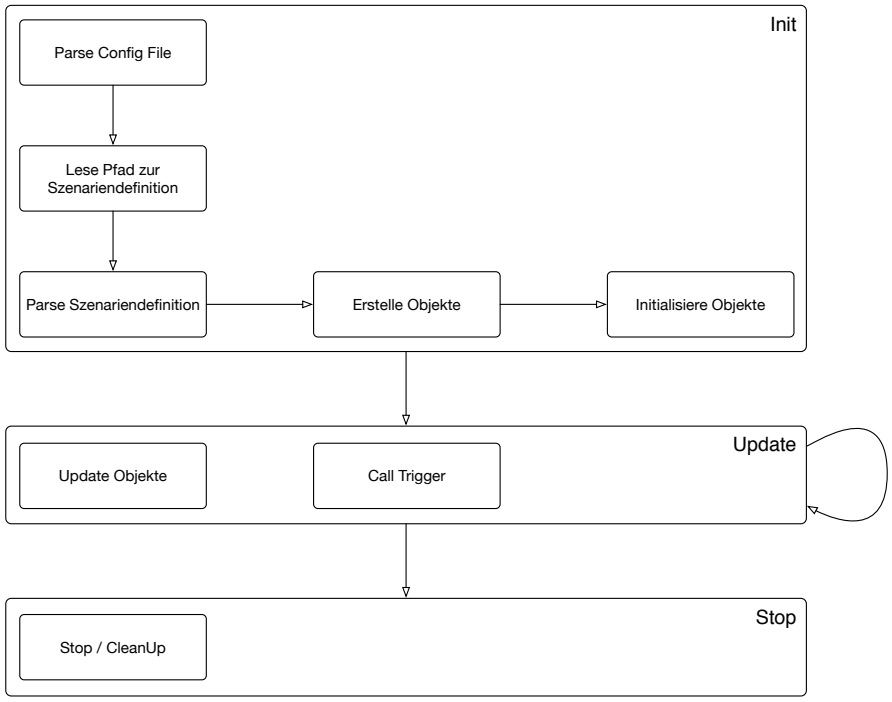

Abbildung 12.1: Ablauf der Phasen der *ScenarioLib*

sollen. So kann es sein, dass beispielsweise ein automatischer Trigger bei einem bestimmten Abstand von Target- und Ego-Fahrzeug einen Bremsvorgang des Targets einleitet. Möchte man dies jedoch zeitweise unterbinden, so kann man das mit der enableTrigger-Funktion bewerkstelligen.

getCar: Diese Funktion liefert das Fahrzeug mit übergebener ID zurück. Dies wird einerseits verwendet, um die Koordinaten und Orientierung jedes einzelnen Objekts an die Visualisierung zu übermitteln, um die Koordinaten und Geschwindigkeiten in das Radarmodell und damit ACC und AEB zu übertragen und weiters, um die notwendigen Parameter an die Akustiksimulation zu liefern.

getPedestrian: Dieser Aufruf ist im Prinzip ident zur vorhergehenden, mit dem Unterschied, dass die Daten der Fußgänger und nicht jene der Fahrzeuge zurückgegeben werden.

getEstimatedLane/Road: Diese beiden Funktionen liefern, sofern ein Ergebnis gefunden werden kann, die IDs des Fahrstreifens als auch der Straße zurück, auf denen sich die gegebene Koordinate befindet. Beide Funktionen liefern -1 wenn kein Fahrstreifen oder keine Straße gefunden werden kann. Da das momentan Verwendung findende Radarmodell noch stark vereinfacht ist, wird diese Funktionalität verwendet um leichter detektieren zu können, ob sich das Ego-Fahrzeug auf dem selben Fahrstreifen befindet wie beispielsweise das vorherfahrende Target-Fahrzeug. Dies erleichtert die Zielauswahl für ACC und AEB enorm.

init: Dieser Aufruf übernimmt als Parameter den Pfad zum Konfigurationsfile, parst dieses, und initialisiert anschließend alle Objekte und Trigger welche in der Szenariodefinition beschrieben sind.

setEgoCar: Wie zuvor erwähnt erlauben Trigger aufgrund von unterschiedlichen Bedingungen Events auszulösen. Manche von diesen Triggern können als Bedingung den Abstand zum Ego-Fahrzeug oder aber auch die absoluten Koordinaten abfragen. Genau diese Informationen, und noch viele mehr, werden hier der Szenario Library aus der Fahrdynamiksimulation zur Verfügung gestellt.

stop: Dieser Aufruf wird bei Beendigung der Simulation aufgerufen und führt die entsprechenden Destruktoren aus.

trigger Diese Funktion erlaubt es, einen oder mehrere Trigger mit einer bestimmten ID auszuführen. Dies wird vor allem vom Operator dazu verwendet, um über das Control-Tablet bestimmte Aktionen, wie beispielsweise einen Fahrstreifenwechsel, zu initiieren.

update: Der Update-Call ist wesentlich, da bei jedem Aufruf die Integration aller Objekte, beziehungsweise die Überprüfung der Bedingungen der Trigger, ausgeführt werden. Wie zuvor erwähnt, muss die externe Aufruf-Frequenz mit jener übereinstimmen, auf welche die Bibliothek konfiguriert ist.

12.3 Szenario-Definition

Um ein Szenario zu definieren, können folgende Elemente in der XML-Datei zur Beschreibung definiert werden:

- Straßen: Eine Auflistung aller verfügbaren Straßen.
- Fahrstreifen: Jede Straße sollte zumindest einen Fahrstreifen beinhalten.
- Wegpunkte: Jeder Fahrstreifen sollte zumindest 2 Wegpunkte beinhalten.
- Fahrzeuge: Definition der Fahrzeuge und deren Startbedingungen.
- Fußgänger: Definition der Fußgänger und deren Startbedingungen.
- Trigger: Den Fahrzeugen und Fußgängern können Trigger zugewiesen werden, um bestimmte Aktionen durchzuführen.

Während einfachere Beschreibungen noch relativ schnell von Hand erstellt werden können, so ist dies bei komplizierteren Straßenverläufen nicht mehr so ohne Weiteres möglich. Aus diesem Grund wurde die *ScenarioLib* in eine GUI-Applikation integriert, über welche in Kapitel 9 ein kurzer Überblick gegeben wird.

Eine minimale Beschreibung, welches das Fahren eines Autos ermöglicht, ist in Listing 12.3 angeführt. Hierbei wird eine Straße definiert, welche aus einem einzigen Fahrstreifen besteht. Sollen mehrere Fahrstreifen angelegt werden, so ist entsprechend eine weitere `Lane` samt Wegpunkten innerhalb der Straße zu definieren. Selbstverständlich können auch mehrere Straßen definiert werden, welche wiederum beliebig viele Fahrstreifen beinhalten können.
Die Definition der Autos erfolgt wie in den Zeilen 23 bis 41 angeführt. Das `type`-Attribut dient der Definition, ob es sich bei dem Auto um einen PKW, LKW oder sonstigen Typ handelt.
Die Init-Bedingungen geben an, auf welchen Zustand sich das jeweilige Fahrzeug zu Beginn der Simulation, beziehungsweise bei einem Reset initialisiert. Neben den absoluten Welt-Koordinaten sind auch die Beschleunigung, Geschwindigkeit sowie der Yaw-Winkel des Fahrzeugs anzugeben. Sämtliche Werte sind in SI-Einheiten definiert.

Wesentlich ist auch die Definition der aktuellen Straße sowie Fahrstreifens. Zu Beginn setzt sich das Fahrzeug auf die Init-Position und sucht den nächstgelegenen Wegpunkt des angegebenen Fahrstreifens, welcher in einer Halbebene vor dem Fahrzeug liegt. Kann kein Wegpunkt gefunden werden, so steuert das Fahrzeug den ersten Wegpunkt an, welcher im gegebenen Fahrstreifen der Straße definiert ist. Wie auch schon bei den Autos, können beliebig viele Streifen sowie Straßen definiert werden. Nachdem zu Beginn alle Wegpunkte in den Speicher geladen werden und kein selektives Nachladen erfolgt, sollte die Dichte der Wegpunkte nicht zu klein gewählt werden um die Anzahl an Wegpunkten in einem akzeptablen Rahmen zu halten. Ein Abstand zwischen 5m und 15m hat sich bewährt.

`StartImmediately` definiert, ob das Fahrzeugmodell bei Beginn der Simulation seinen internen Status updated, oder nicht. Der Vorteil zur Definition einer Startgeschwindigkeit von 0m/s liegt in einer geringeren CPU-Auslastung, da beispielsweise die Integratoren sowie andere Komponenten keine Berechnungen durchführen müssen. Dies kann beispielsweise für parkende Fahrzeuge verwendet werden.

Das `Pavement`-Element wird von einem Fahrzeug-Objekt nicht verwendet und dementsprechend ignoriert.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <ScenarioDefinition version="1.0">
3
4    <Roads>
5      <Road id="0">
6        <Lane id="0">
7          <Waypoint id="0">
8            <Coordinate>
9              <X>0</X>
10             <Y>0</Y>
11           </Coordinate>
12         </Waypoint>
13         <Waypoint id="1">
14           <Coordinate>
```

```
15                <X>10</X>
16                <Y>0</Y>
17              </Coordinate>
18            </Waypoint>
19          </Lane>
20        </Road>
21    </Roads>
22
23    <Cars>
24      <Car id="0" type="5">
25        <InitConditions>
26          <Position>
27            <Coordinate>
28              <X>0</X>
29              <Y>0</Y>
30            </Coordinate>
31          </Position>
32          <Velocity>10</Velocity>
33          <Acceleration>0.00</Acceleration>
34          <Heading>0</Heading>
35          <Lane>0</Lane>
36          <Road>0</Road>
37          <Pavement>0</Pavement>
38          <StartImmediately>true</StartImmediately>
39        </InitConditions>
40      </Car>
41    </Cars>
42
43  </ScenarioDefinition>
```

Listing 12.3: Szenariodefinition - Straßen sowie Fahrzeuge

Die Definition der Gehsteige und Fußgänger erfolgt auf die selbe Art und Weise wie für die Straßen sowie Autos, wobei der Einfachheit halber ein Gehsteig nur eine einzige Liste von Wegpunkten, im Gegensatz zu den Fahrstreifen einer Straße, halten kann. Die Startbedingungen sind exakt gleich definiert, wobei diesmal `Road` und `Lane` ignoriert werden und `Pavement` als die ID geparst wird, auf dessen Gehsteig sich der Fußgänger initialisiert.

Die IDs der Straßen, der Fahrstreifen als auch der Wegpunkte müssen innerhalb des jeweiligen Scopes einmalig sein. Diese IDs werden in der Szenario-Bibliothek zur Referenzierung der jeweiligen Objekte verwendet.

Zur Validierung der Definitionsfiles wurde ein korrespondierendes Schema-File erstellt. Mit diesem ist es einerseits bereits zum Zeitpunkt der Erstellung der Szenarien-Definition möglich, das erstellte .xml File auf Gültigkeit zu überprüfen, andererseits kann zum Zeitpunkt des Ladens durch die Szenario-Bibliothek eine nochmalige Validierung erfolgen.

Auf eine Auflistung des Schemas wird an dieser Stelle aufgrund der Komplexität des Files verzichtet.

```
<Pavements>
  <Pavement id="0">
    <Waypoint id="0">
      <Coordinate>
        <X>0</X>
        <Y>6</Y>
      </Coordinate>
    </Waypoint>
    <Waypoint id="1">
      <Coordinate>
        <X>10</X>
        <Y>6</Y>
      </Coordinate>
    </Waypoint>
  </Pavement>
</Pavements>

<Pedestrians>
  <Pedestrian id="0">
    <InitConditions>
      <Position>
        <Coordinate>
          <X>0</X>
          <Y>6</Y>
        </Coordinate>
      </Position>
      <Velocity>1.3</Velocity>
      <Acceleration>0.0</Acceleration>
      <Heading>0</Heading>
      <Lane>0</Lane>
      <Road>0</Road>
      <Pavement>0</Pavement>
      <StartImmediately>true</StartImmediately>
    </InitConditions>
  </Pedestrian>
</Pedestrians>
```

Listing 12.4: Szenariodefinition - Gehsteige und Fußgänger

Neben der Definition von Straßen, Gehsteigen, Autos sowie Fußgängern, ist es auch möglich, mehrere verschiedene Events zu definieren. Diese können durch eine Vielzahl verschiedener Trigger ausgelöst werden, welche nachfolgend erläutert werden sollen.

StartMovingTrigger	StopMovingTrigger
ResetObjectTrigger	LaneChangeTrigger
AccelerationTrigger	RepositionTrigger
VelocityProfileTrigger	

Tabelle 12.1: Überblick über die verfügbaren Trigger

12.4 Trigger

Tabelle 12.1 gibt einen Überblick darüber, welche Trigger implementiert wurden und auch im Betrieb Verwendung finden. Die Events können einerseits vollautomatisch aufgrund verschiedener Bedingungen, oder aber manuell durch den Operator wie in Kapitel 10 angeführt, ausgelöst werden.

Die Definition der Trigger ist Fahrzeug-spezifisch, weshalb diese in der Szenariendefinition auch innerhalb des `Car`-Scopes nach der Definition der Startbedingungen erfolgt. Prinzipiell sollte sämtlichen Triggern eine unterschiedliche ID vergeben werden. Man kann jedoch bewusst auch idente IDs vergeben, um bei manueller Triggerung mehrere Trigger mit nur einer Aktion auslösen zu können. Wird demnach ein Trigger mit beispielsweise ID 4 ausgelöst, werden auch alle anderen Trigger mit ID 4 aktiviert. Auf automatische Auslösung ohne Zutun des Operators haben die mehrfach vergebenen IDs jedoch keinen Einfluss.

12.4.1 Condition-Attribut

In den nachfolgenden Beispielen wird bei der Definition des Triggers als Attribut oftmals ein optionales `condition`-Attribut zu finden sein. Dies dient dazu, den Trigger vollautomatisch, ohne Zutun des Operators, auslösen zu lassen. Mehrere Bedingungen stehen zur Auswahl, wobei in den nachfolgenden Beispielen die Verwendung jeder verfügbaren `condition` einmal demonstriert wird.

ego_at_coordinate: Dies definiert, dass der Trigger bei einer bestimmten Position des Ego-Fahrzeuges auslöst. Wird diese `condition` definiert, so müssen stets auch eine Koordinate bei welcher ausgelöst werden soll, und ein entsprechender Einzugsbereich `distance` um diese Koordinate angegeben werden. Dieser Einzugsbereich dient als Toleranzband und muss den Anforderungen entsprechend angepasst werden.

at_coordinate: Diese Bedingung besagt, dass der Trigger bei einer gewissen Position des Target-Fahrzeugs ausgelöst werden soll. Wie bei der `ego_at_coordinate`-Bedingung muss auch hier zusätzlich eine Koordinate sowie Distanz angegeben werden.

at_relative_position: Wie der Name schon vermuten lässt, wird die Aktion bei Erreichen einer bestimmten Position des Target-Fahrzeugs relativ zum Ego-Fahrzeug ausgelöst.

Dies wird vor allem beim automatischen Einscheren eines Target-Fahrzeugs nach erfolgtem Überholvorgang verwendet. Befindet sich das Target-Fahrzeug in einem bestimmten Abstand vor dem Ego-Fahrzeug sowie links davon, so kann der Fahrstreifenwechsel eingeleitet werden.
Die Zusatzinformationen welche benötigt werden, sind bei dieser Bedingung etwas komplexer. So wird einerseits ein Vektor definiert, welcher ausgehend vom Ego-Fahrzeug den Trigger-Punkt definiert, andererseits wird jedoch auch ein Toleranzband in x- und y-Richtung angegeben, welches um besagten Trigger-Punkt herum definiert ist. Beispiel 12.4.8 erklärt die Parameter anhand des LaneChangeTrigger näher.

distance_smaller: Hier wird nur der Abstand zwischen Ego-Fahrzeug und Target-Fahrzeug als bestimmende Größe herangezogen. Der Trigger wird dann ausgelöst, wenn der Abstand die definierte Distanz unterschreitet.

distance_greater: Anders als bei der `distance_smaller`-Bedingung wird bei Überschreiten eines bestimmten Abstands das jeweilige Event ausgelöst.

manual: Der Vollständigkeit halber gibt es auch eine `manual`-Bedingung. Hierbei wird der Trigger nicht automatisch ausgelöst. Das Event kann nur durch den Operator aktiviert werden. `manual` ist ident zum Weglassen des `condition`-Attributs.

12.4.2 VelocityProfile-Attribut

Den Triggern kann ein optionales `velocityProfile`-Attribut mitübergeben werden. Hierbei kann ein Pfad zu einer .csv Datei definiert werden, in welcher in der ersten Spalte ein Zeitstempel, und in der zweiten zugehörige Geschwindigkeitswerte angegeben sind. Dieses Geschwindigkeitsprofil wird dann automatisch auf das jeweilige Objekt angewendet. Dies kann beispielsweise bei einem Cut-In Manöver verwendet werden, wo gleichzeitig mit dem Wechsel des Fahrstreifens ein bestimmtes Geschwindigkeitsprofil nachgefahren werden soll. Listing 12.5 zeigt beispielhaft, wie der Inhalt solch einer .csv Datei aussehen kann.

Gleichzeitig wird durch Ableiten des Profils die Beschleunigung berechnet. Ist das Ende des Geschwindigkeitprofils erreicht, so berechnet das dynamische Modell die Geschwindigkeit wieder aus seiner aktuellen Beschleunigung. Dies bedeutet, dass am Ende der .csv Datei zwei idente Werte stehen müssen, sofern sich das Fahrzeug mit konstanter Geschwindigkeit weiterbewegen soll. Anderenfalls wird die zuletzt berechnete Beschleunigung herangezogen und das Fahrzeug beschleunigt oder verzögert entsprechend dem Verlauf des eingeprägten Geschwindigkeitprofils.

```
1 0.2, 21.54
2 0.5, 21.73
3 0.6, 22.03
4 0.7, 22.24
5 0.8, 22.78
```

Listing 12.5: Szenariodefinition - VelocityProfileTrigger; CSV-Format

12.4.3 Armed-Attribut

Möchte man die zuvor aufgelisteten Bedingungen deaktivieren, so ist es möglich, dies im Vorhinein über das `armed`-Attribut zu definieren. Dieses Attribut aktiviert oder deaktiviert die automatische Überprüfung, ob ein Trigger ausgelöst werden soll oder nicht.
Der Operator hat jederzeit die Möglichkeit, über das Contro-Tablet die Überprüfung der Bedingungen ein- oder auszuschalten. Unabhängig davon, kann jederzeit eine manuelle Triggerung erfolgen.
Wird das Attribut nicht angegeben, so ist die selbständige Auslösung automatisch aktiviert.

12.4.4 Hidden-Attribut

Das optionale `hidden`-Attribut hat nur für das Control-Tablet des Operators Bedeutung. Dies definiert, ob der jeweilige Trigger in der Liste verfügbarer Trigger aufscheint oder nicht.
Dies dient hauptsächlich der Übersichtlichkeit, da der Operator oft nur eine begrenzte Anzahl manuell auslösen möchte. Vollautomatische Trigger werden in der Regel nicht durch den Operator bearbeitet und müssen demnach auch nicht zwangsweise sichtbar sein.

12.4.5 StartMovingTrigger

Zuvor wurde bereits erläutert, dass der `StartImmediately`-Parameter in den Startbedingungen dazu führt, dass der dynamische Zustand des Fahrzeugs nicht aktualisiert wird und demnach permanent stillsteht. Mit diesem Trigger ist es nun möglich, die Update-Routine zu aktivieren und das Fahrzeug entsprechend den Init-Bedingungen in Bewegung zu versetzen.
Im nachfolgenden Beispiel ist zu sehen, dass der Trigger vollautomatisch bei einer bestimmten Koordinate des Ego-Fahrzeugs ausgelöst wird. Aufgrund des angegebenen `velocityProfile`-Attributs wird gleichzeitig ein Geschwindigkeitsprofil nachgefahren. Das `Distance`-Element mit einem Wert von 5 gibt an, dass rund um die definierte Koordinate ein Kreis mit einem Radius von 5m gebildet wird, in welchem das Event ausgelöst wird.

```
<Triggers>
  <StartMovingTrigger id="0" condition="ego_at_coordinate"
      velocityProfile="/path/to/profile/p1.csv">
    <Summary>My Summary</Summary>
    <Description>My Description</Description>
    <Coordinate>
      <X>311.0</X>
      <Y>470.0</Y>
    </Coordinate>
    <Distance>5</Distance>
  </StartMovingTrigger>
</Triggers>
```

Listing 12.6: Szenariodefinition - StartMovingTrigger

Momentan wird der Trigger beispielsweise dafür eingesetzt, um ein bereits auf der Autobahn platziertes Fahrzeug bei Erreichen der Autobahn-Auffahrt des Ego-Fahrzeugs in Bewegung zu versetzen. Dadurch können immer sehr ähnliche Voraussetzungen für weitere Fahrmanöver geschaffen werden. Fährt das Ego-Fahrzeug nicht auf die Autobahn auf wird das Target-Fahrzeug nicht aktiviert und verbraucht demnach auch kaum Rechenzeit.

12.4.6 StopMovingTrigger

Als Gegenstück zum StartMovingTrigger deaktiviert der StopMovingTrigger alle Updates des internen Zustands. So kann beispielsweise bei Erreichen einer gewissen Distanz ein Fahrzeug deaktiviert werden, um es bei Unterschreiten einer gewissen Distanz mit dem selben Zustand wieder zu aktivieren.
Im gegebenen Beispiel wird bei einer größeren Entfernung als 2km das Fahrzeug deaktiviert. Weiters ist zu sehen, dass das `armed`-Attribut deaktiviert ist. Dies bedeutet, dass standardmäßig der Trigger nicht automatisch auslöst wird wenn der Abstand die 2km überschreitet. Der Operator muss explizit über das Control-Tablet den Trigger aktivieren, damit dieser bei Eintreten der Bedingung auch wirklich auslöst.
Die kann nützlich sein, um bei gewissen Fahrmanövern bestimmte Aktionen zu unterbinden.

```
<Triggers>
  <StopMovingTrigger id="1" condition="distance_greater" armed="0">
    <Summary>My Summary</Summary>
    <Description>My Description</Description>
    <Coordinate>
      <X>311.0</X>
      <Y>470.0</Y>
    </Coordinate>
    <Distance>2000</Distance>
  </StopMovingTrigger>
</Triggers>
```

Listing 12.7: Szenariodefinition - StopMovingTrigger

12.4.7 ResetObjectTrigger

Ein großer Vorteil der Simulation im Vergleich zu realen Fahrversuchen besteht darin, Fahrzeuge beliebig, auch während der laufenden Simulation, platzieren zu können, ohne auf anderen Verkehr oder Verkehrsregeln Rücksicht nehmen zu müssen.
Dieser Trigger setzt ein Objekt auf seine Init-Bedingungen zurück, welche in Listing 12.3 angeführt sind. Dies beinhaltet Position, Geschwindigkeit, Beschleunigung, Heading, Straße sowie Fahrstreifen als auch `StartImmediately`-Parameter.
Der ResetObjectTrigger ist vor allem dann nützlich, wenn mehrere Fahrmanöver oftmals hintereinander durchgeführt werden sollen, ohne gleich die gesamte Simulation neu starten zu müssen. So können zum Beispiel Stop&Go Manöver oftmals unter exakt gleichen Startbedingungen wiederholt werden, ohne dass der Proband aus der Illusion eines realen

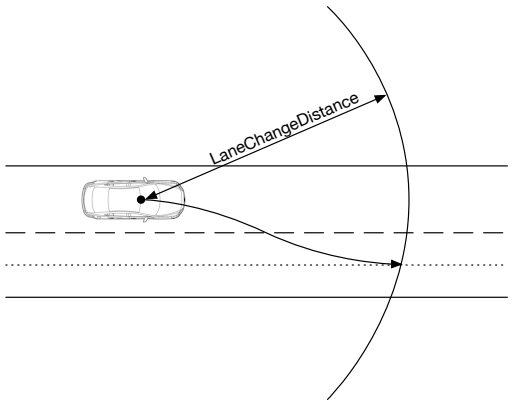

Abbildung 12.2: LaneChangeTrigger Schnittpunkt gefunden

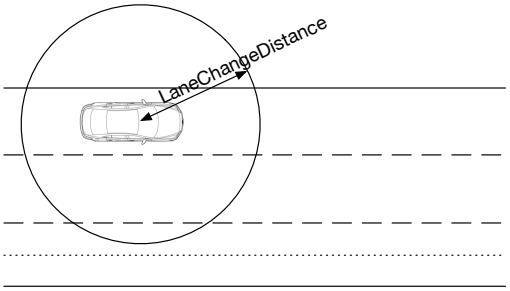

Abbildung 12.3: LaneChangeTrigger - Kein Schnittpunkt mit Lane 0

Fahrzeugs gerissen wird.
Im gegebenen Beispiel ist das `hidden`-Attribut auf 1 gesetzt, womit der Trigger im Control-Tablet in der Liste verfügbarer Trigger nicht aufscheint und damit vom Operator auch nicht ausgelöst werden kann. Nachdem die `condition` auf `manual` gesetzt ist, besteht demnach keine Möglichkeit, dass der Trigger in der Simulation jemals ausgelöst wird.

```
<Triggers>
  <ResetObjectTrigger id="2" condition="manual" hidden="1">
    <Summary>My Summary</Summary>
    <Description>My Description</Description>
  </ResetObjectTrigger>
</Triggers>
```

Listing 12.8: Szenariodefinition - ResetObjectTrigger

12.4.8 LaneChangeTrigger

Der LaneChangeTrigger ermöglicht es Fahrzeugen, innerhalb einer Straße den Fahrstreifen zu wechseln. Dafür muss sowohl ein `Lane`- als auch `LaneChangeDistance`-Element definiert werden.
Das `Lane`-Element gibt an, auf welchen Fahrstreifen der Wechsel erfolgen soll. Hierbei beträgt der Index des am weitest rechts liegenden Streifens 0, und wird dann mit jedem weiteren Fahrstreifen nach links inkrementiert. Ein Wechsel über mehrere Streifen ist möglich.
Das `LaneChangeDistance`-Element beschreibt, über was für eine Entfernung hinweg der Fahrstreifenwechsel andauert. Es wird ein Kreis mit der `LaneChangeDistance` um das Fahrzeug gebildet, wobei der Schnittpunkt dieses Kreises mit der Mittellinie des Fahrstreifens, auf welchen der Wechsel erfolgen soll, berechnet wird. Dieser Schnittpunkt beschreibt dann das Ende des Fahrstreifenwechsels. Abbildung 12.2 veranschaulicht diese Tatsache.

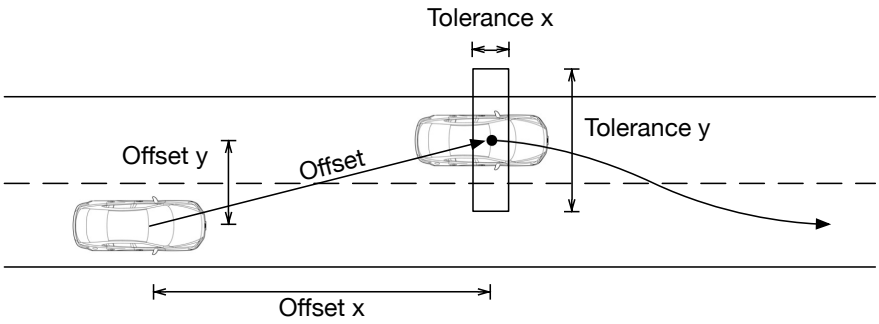

Abbildung 12.4: LaneChangeTrigger - Relative Bedingung

In Abbildung 12.3 ist ein Problem skizziert, welches bei zu klein gewählten Distanzen für den Fahrstreifenwechsel auftreten kann. Im dargestellten Fall soll ein Wechsel vom äußersten Fahrstreifen auf den innersten erfolgen, jedoch wurde eine so kleine `LaneChangeDistance` gewählt, dass kein Schnittpunkt gefunden werden kann.
In diesem Fall wird kein Fahrstreifenwechsel durchgeführt und eine entsprechende Nachricht in das Logfile geschrieben. Ein Wechsel in den mittleren Fahrstreifen wäre mit gegebener Distanz sehr wohl möglich, auch wenn daraus aufgrund des kleinen Radius ein äußerst unrealistisches Fahrverhalten resultieren würde.

```
<Triggers>
  <LaneChangeTrigger id="3" condition="manual" armed="1" hidden="1">
    <Summary>My Summary</Summary>
    <Description>MyDescription</Description>
    <LaneChangeDistance>80</LaneChangeDistance>
    <Lane>0</Lane>
  </LaneChangeTrigger>
</Triggers>
```

Listing 12.9: Szenariodefinition - LaneChangeTrigger

Listing 12.10 zeigt, wie die `at_relative_position`-Bedingung im Zusammenspiel mit dem LaneChangeTrigger verwendet werden kann. Dies stellt eine sinnvolle Kombination dar, da häufig Fahrmanöver aus Realversuchen, wie beispielsweise Cut-Ins, durch relative Positionen zum Fahrzeug charakterisiert werden.
Abbildung 12.4 zeigt, wie der Offset als auch das Toleranzband zu interpretieren sind. Die Toleranz definiert ein Rechteck, wobei der Offset beschreibt, wo sich dieses Rechteck relativ zum Ego-Fahrzeug befindet. Gelangt das Target-Fahrzeug in dieses Rechteck, wird der Trigger ausgelöst und der Wechsel entsprechend der `Lane`- sowie `LaneChangeDistance` durchgeführt. Es wird häufig so sein, dass die Toleranz in Fahrtrichtung möglichst klein gehalten wird, um einen akkuraten und reproduzierbaren Abstand zwischen den beiden Fahrzeugen zu erhalten. Die Toleranz in y-Richtung ist in der Regel wesentlich größer zu wählen, um eventuelle Fahrungenauigkeiten welche durch den Fahrer zustande kommen, ausgleichen zu können.

```
<Triggers>
  <LaneChangeTrigger id="4" condition="at_relative_position">
    <Summary>My Summary</Summary>
    <Description>My Description</Description>
    <LaneChangeDistance>40</LaneChangeDistance>
    <Lane>0</Lane>
    <RelativeTriggerOffset>
      <Vector>
        <X>10.0</X>
        <Y>4.0</Y>
      </Vector>
    </RelativeTriggerOffset>
    <RelativeTriggerTolerance>
      <Vector>
        <X>1.0</X>
        <Y>10.0</Y>
      </Vector>
    </RelativeTriggerTolerance>
  </LaneChangeTrigger>
</Triggers>
```

Listing 12.10: Szenariodefinition - LaneChangeTrigger; Relative Bedingung

12.4.9 AccelerationTrigger

Der AccelerationTrigger setzt die Beschleunigung eines Objekts. Es sind sowohl positive als auch negative Werte, welche zum Abbremsen des Fahrzeugs führen, möglich.
Neben der Größe der Beschleunigung bzw. Verzögerung ist zusätzlich noch ein `EndSpeed`-Element zu definieren. Dieses beschreibt, bis zu welcher Geschwindigkeit die Beschleunigung oder Verzögerung eingeprägt werden soll. Wird dieser Wert über- oder unterschritten, wird die Beschleunigung automatisch auf 0 gesetzt.
Der Trigger kann beispielsweise dazu verwendet werden, um ein Fahrzeug auf eine gewünschte Zielgeschwindigkeit zu beschleunigen, oder aber auch vor einer Ampel anhalten zu lassen.

```
<Triggers>
  <AccelerationTrigger id="5" condition="distance_smaller">
    <Summary>My Summary</Summary>
    <Description>My Description</Description>
    <Acceleration>1.0</Acceleration>
    <EndSpeed>27.78</EndSpeed>
    <Distance>12</Distance>
  </AccelerationTrigger>
</Triggers>
```

Listing 12.11: Szenariodefinition - AccelerationTrigger

12.4.10 RepositionTrigger

Mit dem RepositionTrigger ist es möglich, ein Fahrzeug während der Simulation relativ zum Ego-Fahrzeug zu positionieren. Dies wird beispielsweise dafür verwendet, um Cut-In Manöver mehrmals hintereinander durchführen zu können. Dafür wird das Target-Fahrzeug hinter dem Ego-Fahrzeug, um einen Fahrstreifen versetzt, positioniert und mit einer höheren Geschwindigkeit initialisiert. Das Ego-Fahrzeug wird daraufhin überholt und ein LaneChangeTrigger aus Kapitel 12.4.8 führt bei Erreichen einer relativen Koordinate vollautomatisch den Fahrstreifenwechsel aus.

Das `PositionOffset`-Element ist hierbei gleich definiert wie der `RelativeTriggerOffset` der `at_relative_position`-Bedingung. Alle weiteren Elemente entsprechen in etwa jenen, welche aus den Init-Bedingungen der Fahrzeuge aus Listing 12.3 bekannt sind. Ausnahme ist hierbei das Heading, welches relativ zum Ego-Fahrzeug und nicht absolut angegeben wird.

Es ist darauf zu achten, dass eine sinnvolle Kombination aus y-Offset und Fahrstreifen angegeben wird. Fährt das Ego-Fahrzeug beispielsweise auf dem innersten Fahrstreifen und der RepositionTrigger ist mit einem y-Offset von 4m, was in etwa dem nächsten Fahrstreifen entspricht, angegeben, der Fahrstreifen auf welchen initialisiert wird ist jedoch mit 3 definiert, so wird das Target-Fahrzeug zuerst einen Wechsel vom zweiten auf den vierten Fahrstreifen vollziehen, was unter Umständen nicht gewünscht ist.

```
<Triggers>
<RepositionTrigger id="9" condition="manual" armed="1" hidden="1">
  <Summary>My Summary</Summary>
  <Description>My Description</Description>
 <RelativeInitConditions>
  <PositionOffset>
    <Vector>
      <X>−10</X>
      <Y>4.0</Y>
    </Vector>
    </PositionOffset>
    <Velocity>41.6</Velocity>
    <Acceleration>0.0</Acceleration>
    <HeadingOffset>0.0</HeadingOffset>
    <Lane>1</Lane>
    <Road>0</Road>
    <Pavement>0</Pavement>
  </RelativeInitConditions>
</RepositionTrigger>
<Triggers>
```

Listing 12.12: Szenariodefinition - RepositionTrigger

12.4.11 VelocityProfileTrigger

Der VelocityProfileTrigger ist eine Ergänzung zu den vorhergehend beschriebenen Triggern. Während es auch bei allen anderen möglich ist Geschwindigkeitsprofile vorzugeben, so ist dies immer mit dem Auslösen der jeweiligen Events verbunden. Möchte man jedoch nur einen Verlauf einspielen, so kann dieser Trigger verwendet werden. Informationen zum Datenformat wurden bereits in Kapitel 12.4.2 dargebracht.

```
<Triggers>
<VelocityProfileTrigger id="10" condition="manual"
    velocityProfile="/path/to/profile/p2.csv">
  <Summary>My summary</Summary>
  <Description>My Description</Description>
</VelocityProfileTrigger>
</Triggers>
```

Listing 12.13: Szenariodefinition - VelocityProfileTrigger

12.5 Aufbau

Abbildung 12.5 gibt einen vereinfachten Überblick über die in der *ScenarioLib* verwendeten Klassen sowie deren Abhängigkeiten. Die Bibliothek verwendet zusätzlich die Boost-Libraries zum Schreiben der internen Log-Files sowie für die Algorithmen zur Integration der Massenpunkte mit Hilfe eines Runge-Kutta Verfahrens. Weiters wird die RapidXml-Bibliothek zum Parsen und Schreiben von XML-Dateien verwendet. In Kürze sollen anschließend die wichtigsten Komponenten beschrieben werden.

12.5.1 ConfigParser

Der `ConfigParser` ist dafür zuständig, das in Listing 12.1 angeführte XML-File zu parsen und die darin enthaltenen Informationen zur weiteren Verarbeitung entsprechend zur Verfügung zu stellen. Die betrifft vor allem den Pfad zur Szenariendefinition sowie das Update-Interval, mit der die Update-Routine von extern aufgerufen wird.

12.5.2 ScenarioParser

Der `ScenarioParser` ist eine der wichtigsten Komponenten der *ScenarioLib*. Unter Verwendung des Pfades welcher vom `ConfigParser` zur Verfügung gestellt wird, wird das XML-File zur Szenariendefinition, welches bereits in Kapitel 12.3 erläutert wurde, geparst und entsprechend das Szenario aufgebaut. Abbildung 12.6 zeigt das einfache Interface, über welches `Config`- als auch `ScenarioParser` angesprochen werden können.

12.5.3 ScenarioWriter

Als logisches Pendant zum `ScenarioParser` gibt es auch eine Komponente welche in der Lage ist, eine entsprechende Szenariendefinition zu erzeugen. Während der

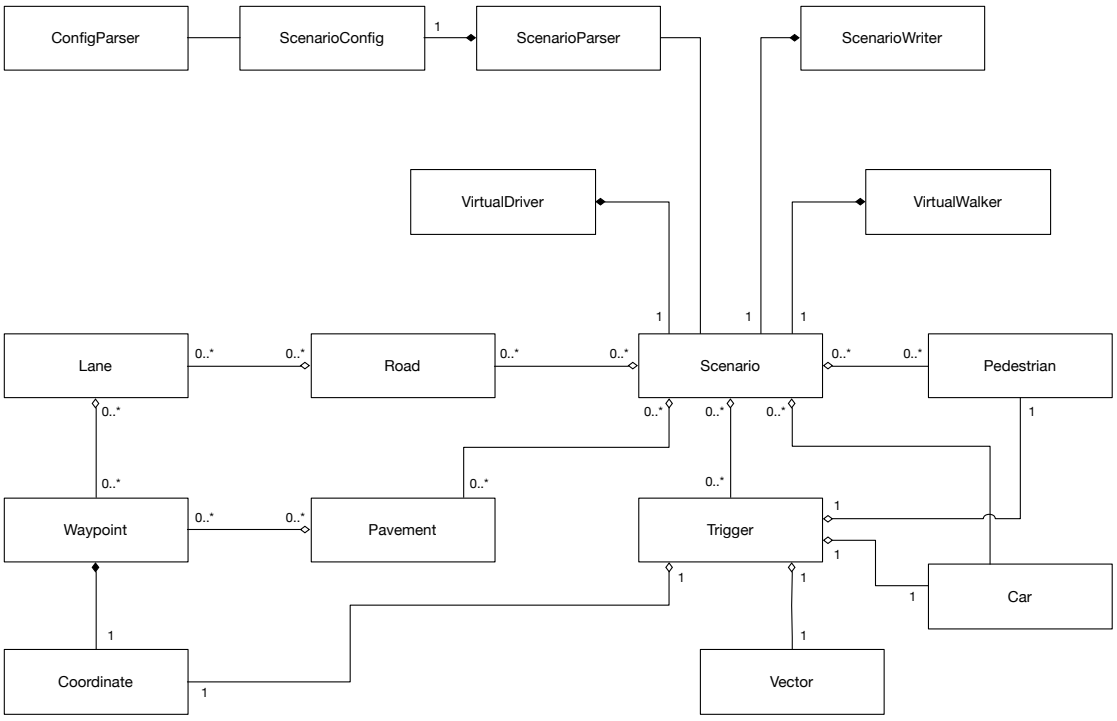

Abbildung 12.5: UML *ScenarioLib* - Überblick

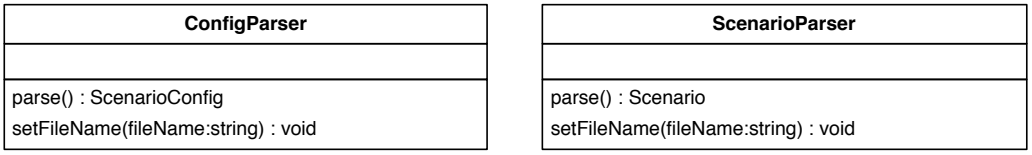

Abbildung 12.6: UML *ScenarioLib* - Parser

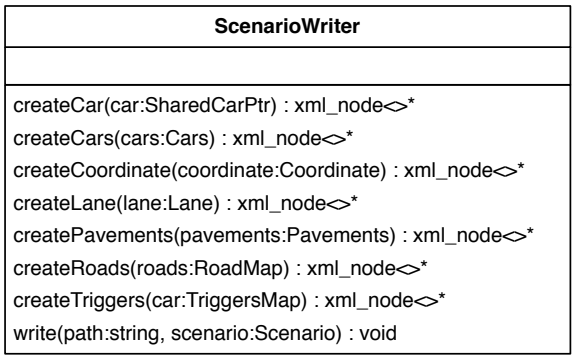

Abbildung 12.7: UML *ScenarioLib* - ScenarioWriter

`ScenarioWriter` in der reinen Simulation nicht gebraucht wird, so findet er jedoch in der GUI-Applikation, welche in Kapitel 9 gezeigt wird, Verwendung.
Abbildung 12.7 illustriert das zur Verfügung stehende Interface, wobei die `create`-Methoden nur zum Testen nach außen hin verfügbar gemacht wurden. Zum Erstellen des entsprechenden XML-Files wird einzig und allein die `write`-Methode benötigt.

An dieser Stelle sei darauf hingewiesen, dass der ScenarioWriter das Generieren der Trigger-Elemente zum gegebenen Zeitpunkt nicht unterstützt. Dieses müssen von Hand nach Erstellen der XML-Datei durch den `ScenarioWriter` hinzugefügt werden.

12.5.4 Scenario

Herzstück der Bibliothek ist die `Scenario`-Klasse, welche als Container für alle im Szenario vorkommenden Objekte dient. Dies umfasst vor allem die

- Straßen samt Fahrstreifen
- Gehsteige samt Wegpunkten
- Autos
- Fußgänger
- Trigger

Das zugehörige Objekt wird vom `ScenarioParser` mit Hilfe der in Abbildung 12.6 dargestellten `parse`-Methode generiert.
Wie zuvor erwähnt, kann eine korrespondierende XML-Datei aus einem `Scenario`-Objekt mit Hilfe des `ScenarioWriter` erzeugt werden. Das Interface der `Scenario`-Klasse ist in Abbildung 12.8 zu sehen.

Wesentlich sind die `update`- sowie `reset`-Aufrufe. Diese werden an die entsprechenden Aufrufe der Fahrzeuge, Fußgänger sowie Trigger weitergeleitet und sind wesentlich für die Funktion der Bibliothek.

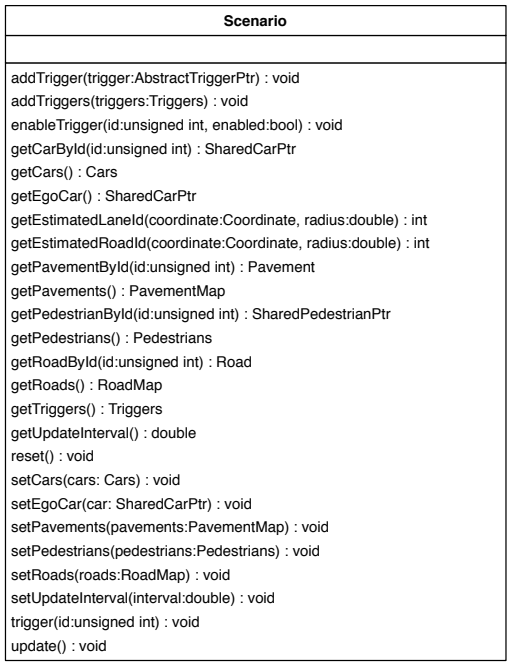

Abbildung 12.8: UML *ScenarioLib* - Scenario

12.5.5 MovingObject

Nachdem Autos sowie Fußgänger eine große Anzahl von Parametern sowie Eigenschaften gemein haben ist es naheliegend, ein gemeinsames Interface für diese Objekte zu definieren. Dies wird durch die Einführung zweier abstrakter Klassen, `AbstractObject` sowie `AbstractMovingObject` realisiert. `AbstractMovingObject` hält alle Daten, welche notwendig sind, um Massenpunkte entlang gewisser Wegpunkte bewegen zu lassen. Dies umfasst unter anderem Integratoren um von den Beschleunigungen auf die Positionen zu schließen, oder aber auch den jeweils nächsten Wegpunkt, auf welchen sich das Objekt zubewegt.

Die `Car`-Klasse berechnet intern neben Beschleunigung, Geschwindigkeit und Heading noch zusätzliche Parameter, welche für die Soundgenerierung notwendig sind. Dies umfasst beispielsweise Daten wie Motordrehzahl, Motormoment, Gang oder Gaspedalstellung. Nachdem keine vollwertigen Fahrzeuge sondern nur Massenpunkte simuliert werden, werden diese Daten aufgrund einfacher Kennlinien abgeschätzt.

In der `update`-Routine erfolgt die Integration mittels eines Runge-Kutta Verfahrens 4. Ordnung. Ist ein Geschwindigkeitsprofil definiert, so wird die Geschwindigkeit entsprechend überschrieben und die Beschleunigung durch Ableiten berechnet.
Die `start`- und `stop`-Aufrufe aktivieren oder deaktivieren das Aktualisieren des internen Zustands.
`reset` setzt das Objekt auf jenen Initialzustand zurück, welcher in der Szenariendefinition angegeben ist.

Abbildung 12.9 zeigt den kompletten Ableitungs-Baum samt abstrakter Klassen, durch welche die Eigenschaften der Objekte definiert werden. Durch diese Struktur ist es unwesentlich, ob bei einem Update oder Reset-Call das darunter liegende Objekt ein Fußgänger oder Fahrzeug ist, da beide über das selbe Interface verfügen.

12.5.6 Trigger

In Kapitel 12.4 wurden bereits die implementierten Trigger angeführt. Nachdem auch die Trigger eine Vielzahl an Eigenschaften miteinander teilen liegt es auch hier auf der Hand, ein gemeinsames Interface durch Definition einer abstrakten Basisklasse zu definieren. Spezielle Parameter können dann in den jeweiligen Spezialisierungen implementiert werden.

Wesentlich sind die `update`- sowie `trigger`-Calls. In der `update`-Routine werden, sofern definiert, die Bedingungen überprüft, ob der Trigger auslösen soll oder nicht. Wenn ja, wird die `trigger`-Methode aufgerufen, welche auch bei manuellem Aufruf durch den Operator benutzt wird. Abbildung 12.10 zeigt die abstrakte Basisklassen sowie die darauf aufbauenden Spezialisierungen.

12.5.7 Road/Lane

Die `Road`-Klasse dient als Container für die Fahrstreifen, und in weiterer Folge für die korrespondierenden Wegpunkte. Während die Straße eine Liste von Fahrstreifen enthält, enthält jeder Fahrstreifen eine Liste von Wegpunkten.
Der Einfachheit halber stellt die `Road`-Klasse einige Methoden bereit, um direkt auf Daten der entsprechenden Fahrstreifen zugreifen zu können.
In Abbildung 12.11 sind die Interfaces dargestellt.
Die Definition der Gehsteige ist sehr ähnlich. Einziger Unterschied ist der, dass ein Gehsteig direkt dessen Wegpunkte hält, da nicht mehrere Fahrstreifen benötigt werden.

12.5.8 VirtualDriver

Wie bereits mehrfach erwähnt, bildet die *ScenarioLib* nur einen einfachen Massenpunkt und kein Einspurmodell oder gar Vollfahrzeug, ab. Aus diesem Grund muss es eine Instanz geben, welche das Heading der Objekte entlang der Wegpunkte setzt. Genau dies macht die `VirtualDriver`-Klasse. Die Klasse kennt die Straßenverläufe sowie die sich darauf befindlichen Objekte. Wird die `update`-Routine aufgerufen, so iteriert der `VirtualDriver` über alle Objekte und setzt das Heading sowie den jeweils nächsten Wegpunkt.

Bei der Berechnung des Headings wird zusätzlich eine Glättung des Verlaufs durchgeführt, da ansonsten bei Erreichen eines Wegpunkts und dem Setzen des jeweils nächsten eine sprunghafte Änderung des Yaw-Winkels zu beobachten wäre.
Die Glättung erfolgt, wie in Abbildung 12.12 dargestellt. Hierbei werden ab einer definierten Umgebung um den Wegpunkt neue Zwischen-Wegpunkte berechnet, welche auf der direkten Verbindung der eigentlichen Wegpunkte liegen. Auf diese dynamisch berechneten Koordinaten wird dann das Heading des Fahrzeugs ausgerichtet, wobei die Neuberechnung

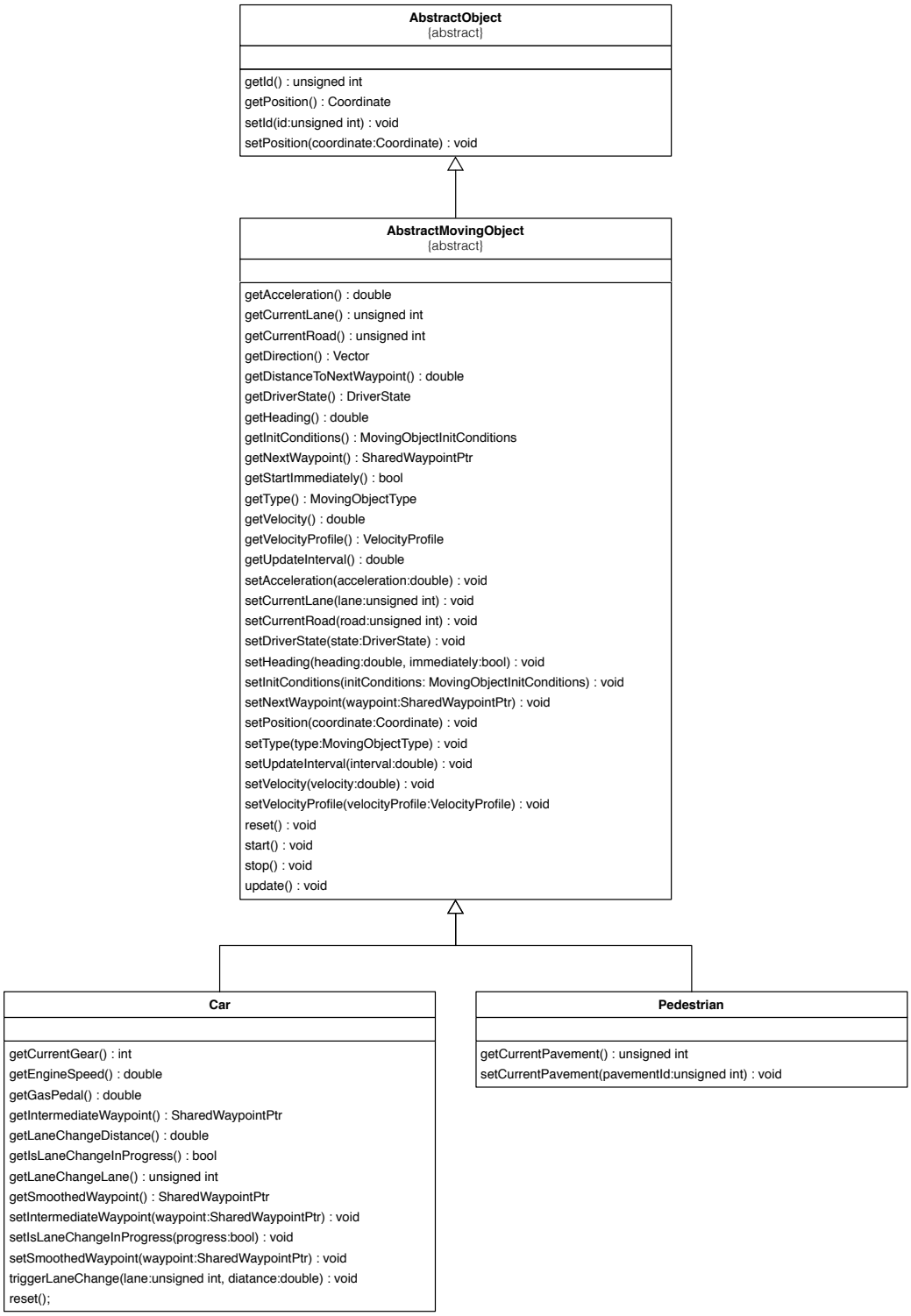

Abbildung 12.9: UML *ScenarioLib* - AbstractObject

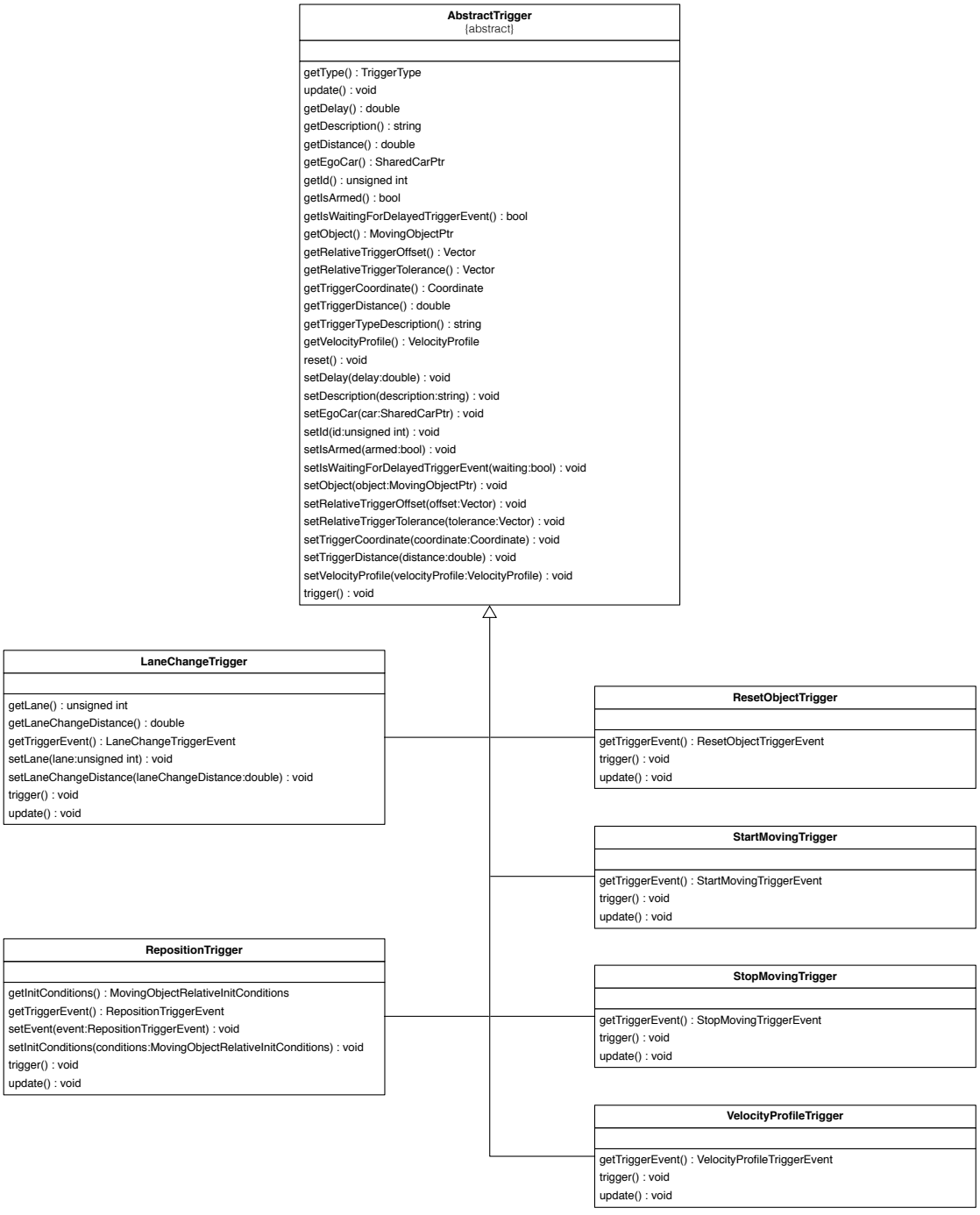

Abbildung 12.10: UML *ScenarioLib* - AbstractTrigger

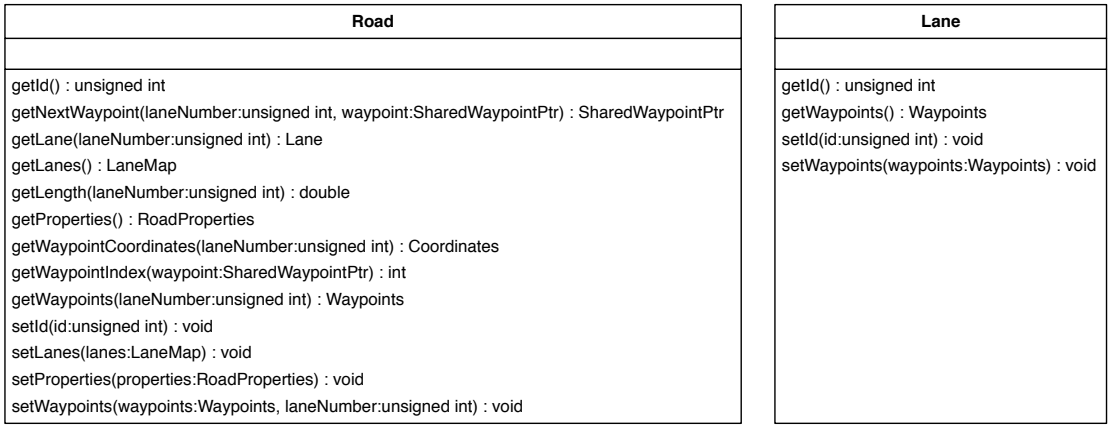

Abbildung 12.11: UML *ScenarioLib* - Road/Lane

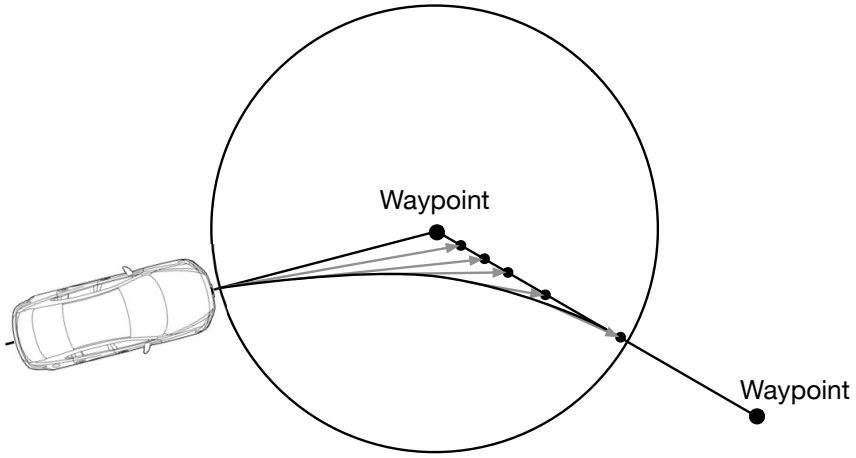

Abbildung 12.12: VirtualDriver - Glättung

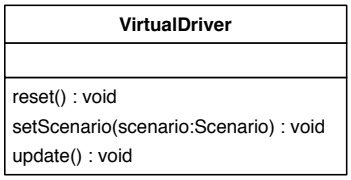

Abbildung 12.13: UML *ScenarioLib* - VirtualDriver

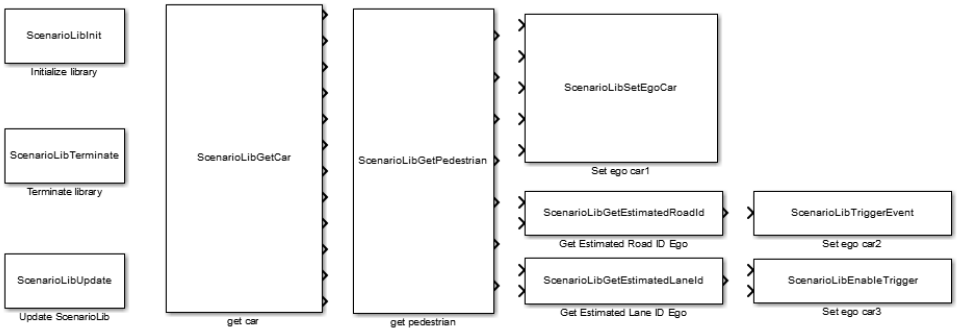

Abbildung 12.14: *ScenarioLib* - Simulink

in jedem Update-Schritt erfolgt. Durch diesen einfachen Mechanismus wird ein glatter Verlauf des Headings erreicht, ohne rechenintensive Spline-Algorithmen verwenden zu müssen.

12.6 Interface

Abbildung 12.14 zeigt all jene Simulink-Blöcke, welche durch das in Listing 12.2 definierte Interface angesprochen werden können.
Neben dem Init-Block, in dem der Pfad zum Konfigurationsfile aus Listing 12.1 angegeben werden muss, müssen in jedem Modell welches die ScenarioLib implementiert auch jeweils ein Update- sowie Terminate-Block vorhanden sein, um die Bibliothek entsprechend upzudaten sowie sauber zu beenden.
Ebenfalls vorhanden sein muss der ScenarioLibSetEgoCar-Block, in welchem der Library die Daten des Ego-Simulationsfahrzeuges übergeben werden. Dies ist notwendig, damit die Trigger aus Kapitel 12.4 entsprechend den angegebenen Bedingungen, wie beispielsweise einer gewissen Distanz, ausgelöst werden können.
Alle anderen Blöcke sind optional und liefern verschiedene Daten aus der Szenariengenerierung wie beispielsweise Signale der simulierten Fahrzeuge und Fußgänger, oder aber, sofern vorhanden, die ID des Fahrstreifens sowie der Straße auf dem sich das Ego-Fahrzeug gerade befindet.
Schlussendlich können über die Trigger-Blöcke Events aktiviert/deaktiviert sowie ausgelöst werden. Dies kann beispielsweise durch das Control-Tablet aus Kapitel 10 erfolgen.

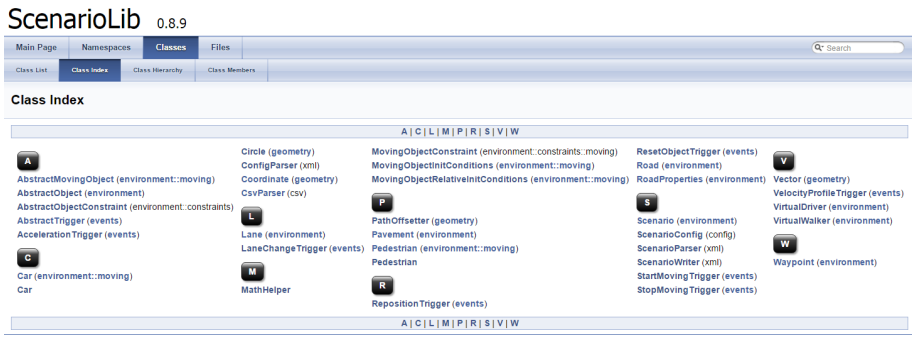

Abbildung 12.15: Doxygen - Class Index

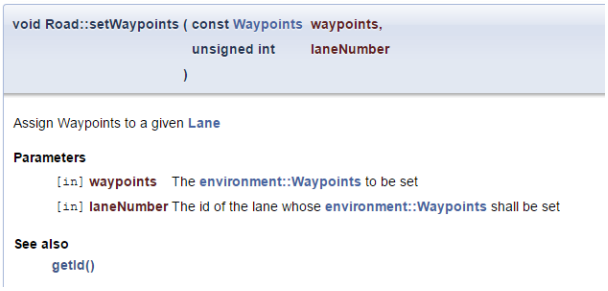

Abbildung 12.16: Doxygen - Member function

12.7 Dokumentation

12.7.1 Doxygen

Die gesamte Bibliothek wurde durchgehend mit Hilfe von Doxygen dokumentiert. Bei Doxygen handelt es sich um ein Open-Source-Tool welches in der Lage ist, aus Kommentaren im Quelltext eine übersichtliche Dokumentation zu erstellen.
Es werden praktisch alle verbreiteten Programmiersprachen unterstützt, wobei gleichzeitig auch verschiedenste Ausgabeformate zur Verfügung stehen. Die gängigsten wären hierbei wohl HTML, LaTeX, PDF, Manpages oder Markdown.

Bei der Ausgabe in HTML ist es möglich, durch Hyperlinks interaktiv durch die Dokumentation zu navigieren als auch zu suchen, was das Auffinden der entsprechenden Klassen und Methoden wesentlich vereinfacht. Zusätzlich ist es möglich, UML-Diagramme oder andere Grafiken in die Dokumentation einzubinden um zusätzliche Informationen zur Verfügung zu stellen.
Abbildungen 12.15 und 12.16 zeigen, wie Teile einer generierten HTML-Seite aussehen können.

In Listing 12.14 ist exemplarisch angeführt, wie durch Hinzufügen eines Kommentars der in Abbildung 12.16 generierte Abschnitt der Dokumentation zu Stande kommt. Wie unschwer zu erkennen ist, muss ein Trade-off zwischen Dokumentation und Quelltext gefunden werden, da die Größe der Files doch beträchtlich zunimmt. Dadurch leidet zwar

auf der einen Seite in gewisser Weise die Lesbarkeit, auf der anderen Seite jedoch ist durch Dokumentation direkt im Code unter Umständen das Nachschlagen eventuell vorliegender externer Dokumentation nicht länger notwendig.

```
1 /**
2 * Assign Waypoints to a given Lane
3 *
4 * @param[in] waypoints The environment::Waypoints to be set
5 * @param[in] laneNumber The id of the lane whose environment::Waypoints shall be set
6 * @see getId()
7 */
8 void setWaypoints(const Waypoints waypoints, unsigned int laneNumber);
```

Listing 12.14: Dokumentation direkt im Code

12.8 SUMO - Simulation of Urban MObility

Die vorgestellte *ScenarioLib* wurde aus der Anforderung heraus geboren, in der Realität durchgeführte Studien möglichst naturgetreu in der Simulation nachzubilden. Dabei lag der Fokus weniger auf möglichst detailgetreuer Nachbildung der optischen Gegebenheiten sowie Verkehr, als auf möglichst guter Rekonstruktion der dynamischen Aspekte des Verkehrsaufkommens basierend auf hochgenauen Messdaten. Aus diesem Grund wurde beispielsweise die Möglichkeit des Einspielens von Geschwindigkeitsprofilen integriert.

Was die vorgestellte Bibliothek nicht bietet, ist die automatische Interaktion der einzelnen Verkehrsteilnehmer untereinander bzw. eine vollautomatische Pfadgenerierung, ohne jedes einzelne Event extra definieren zu müssen.
Mit SUMO - Simulation of Urban MObility steht eine Open Source Verkehrsflusssimulation zur Verfügung, deren Entwicklung hauptsächlich vom Institut für Verkehrssystemtechnik des Deutschen Zentrums für Luft- und Raumfahrt vorangetrieben wird. SUMO wird seit 2001 laufend weiterentwickelt und wurde bereits für die Simulation von Verkehrsflüssen bei einer Großzahl von Großveranstaltungen, bis hin zu Fußballweltmeisterschaften, eingesetzt. Natürlich bietet SUMO auch die vollständige Integration von Gehsteigen und Fußgängern an.

SUMO basiert auf C++ und ist auf jedem gängigen Betriebssystem lauffähig. Externe Abhängigkeiten zu weiteren Bibliotheken sind nicht vorhanden. Es besteht die Möglichkeit, Straßennetze aus OpenStreetMap, OpenDrive, Vissim, VISUM sowie anderen gängigen Karten-Formaten zu importieren.
Die Verkehrsflusssimulation stellt nur eine minimale grafische Oberfläche, wie in Abbildung 12.17 zu sehen, zur Verfügung. Es kann nur das Endresultat der Simulation mittels einer sehr rudimentären Visualisierung betrachtet werden. Für alle anderen Schritte, welche für eine lauffähige Simulation von Nöten sind, müssen händisch oder mit Hilfe von einer Vielzahl von Kommandozeilentools, durchgeführt werden. Einzig für die Generierung des Straßennetztes steht eine sehr einfache GUI-Applikation zur Verfügung, welche aufgrund

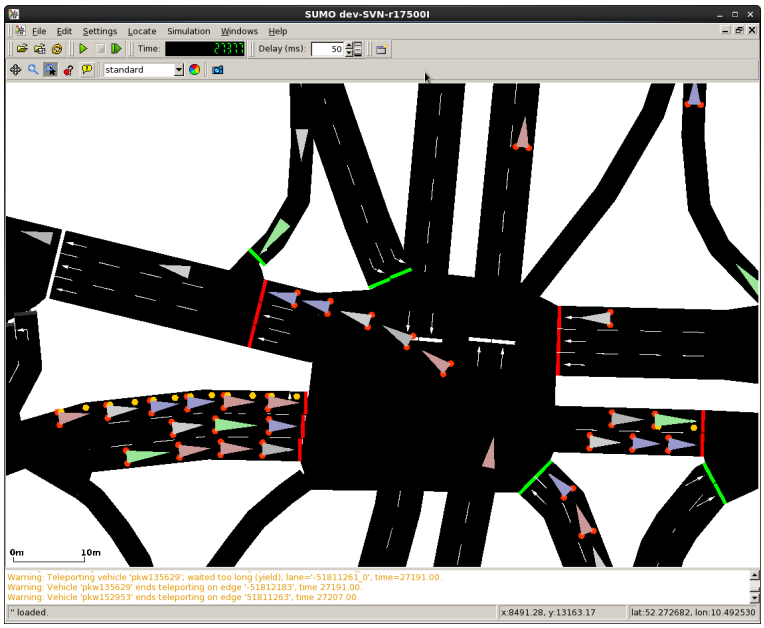

Abbildung 12.17: SUMO - Screenshot [13]

des eingeschränkten Funktionsumfangs jedoch nur bedingt Vorteile gegenüber den Kommandozeilentools bietet. Der prinzipielle Ablauf zur Benutzung von SUMO besteht aus 3 grundlegenden Schritten

- Generierung des Netzwerkes
- Beschreiben der Randbedingungen
- Simulation

12.8.1 Netzwerkgenerierung

Wie bereits erwähnt, muss für die Simulation ein Straßennetzwerk definiert werden. In diesem werden neben der gesamten Topologie der Fahrstreifen, Kreuzungen und Kreisverkehre noch viele andere, sofern benötigt, Informationen untergebracht. Dies umfasst unter anderem die Positionen von Ampeln und Haltestellen des öffentlichen Verkehrs, Induktionsschleifen und weitere allgemeine Verkehrsanbindungen.
Weiters müssen Geschwindigkeitsbegrenzungen, Einbahnen, verschiedenste Prioritäten, Ampelprogramme und vieles mehr definiert werden. Man sieht, dass aufgrund der hohen Funktionalität die Konfiguration einen hohen Komlexitätsgrad erreicht hat, welcher ohne grafische Benutzerführung einen recht schwierigen Einstieg in die Benutzung von SUMO zur Folge hat.

Unterstützt wird dieser Prozess durch das Kommandozeilenprogramm `netconvert`, welches in der Lage ist, eine Vielzahl von Dateiformaten zu import- als auch exportieren. Tabelle 12.2 gibt einen Überblick über die unterstützten Dateiformate. Es stehen eine

Import	Export
SUMO native XML descriptions	SUMO native XML descriptions
SUMO	SUMO
OpenDRIVE	OpenDRIVE
MATSim	MATSim
DLR proprietäres Navteq GDF	DLR proprietäres Navteq GDF
OpenStreetMap	
VISUM	
Vissim	
Shapefiles	
Robocup Rescue League	

Tabelle 12.2: SUMO - Überblick der unterstützten Dateiformate

Vielzahl von Tutorials zur Verfügung, welche die Erstellung der Netzwerke illustrieren. Auf weitere Details soll an dieser Stelle verzichtet werden.

12.8.2 Randbedingungen

Neben der Topologie des Straßennetzes muss eine Beschreibung der Verkehrsflusssimulation an sich erfolgen. Dieser Vorgang fällt unter den Sammelbegriff *Demand modelling*. Als die wichtigsten Informationen welche der Simulation zugeführt werden müssen gelten die Routen der einzelnen Fahrzeuge. Diese Routen beschreiben, wo und wann sich welche Fahrzeuge auf dem Straßennetzwerk bewegen. Es werden eine Vielzahl unterschiedlicher Beschreibungen unterstützt, wobei nachfolgend ein kurzer Überblick gegeben werden soll.

Fahrzeuge

Der Definition der Fahrzeuge kommt eine wesentliche Bedeutung zu. Hier werden Maximalgeschwindigkeiten sowie Beschleunigungen definiert, weiters Größe, Fahrzeugtyp und viele weitere Standardparameter. Zusätzlich können jedoch auch Emissionsklassen als auch gewissen Eigenschaften des simulierten Fahrers definiert werden. Momentan stehen 10 verschiedene Car-Following Modelle zur Verfügung, welche allesamt leicht unterschiedliche Eigenschaften und Parameter bieten. Die Wahl des Fahrermodells kann durchaus Einfluss auf den Ausgang der Simulation nehmen, weshalb Simulationen auch mit variierenden Fahrermodellen durchgeführt werden sollten. Die Fahrzeuge werden dann Routen zugeteilt, welche nach den nachfolgenden Prinzipien erstellt werden können.

Trip / Flow Definitionen

Ein Tool Namens `duarouter` ermöglicht es, unter Eingabe einer Trip oder Flow Definition sowie der Netzwerk-Topologie eine Routing-Datei zu erstellen, welche dann wiederum in weiterer Folge von der Simulation verwendet werden kann. Eine Trip-Definition gilt für eine exakte Anzahl von Fahrzeugen, wobei hier die Abfahrtszeiten genau definiert sind.

Das Intervall zwischen den Abfahrtszeiten ist hierbei konstant.
Bei der Flow-Definition wiederum wird eine definierte Anzahl in einem bestimmten Zeitintervall angegeben, wobei hier die Fahrzeuge in variierenden Abständen positioniert werden. Neben Abfahrts- und Ankunftsort können auch Zwischenstationen, Start- und Endposition sowie Geschwindigkeiten, Fahrstreifen bei Abfahrt sowie Ankunft und vieles mehr definiert werden.

Es gibt 2 Varianten, nach denen die Routen berechnet werden können.

- **Kürzester Pfad**: Hierbei werden die jeweils kürzesten Routen zwischen den einzelnen Wegpunkten herangezogen
- **Dynamische Zuweisung**: Hierbei wird für jedes Fahrzeug eine Route gesucht, so dass dessen Reisekosten optimal sind. Dementsprechend ist dieser Vorgang je nach Komplexität relativ rechen- sowie zeitintensiv

Zufällige Verteilung

Es wird ein Python-Skript zur Verfügung gestellt, welches zufälligen Verkehr generiert. Wenn gewollt, können eine Vielzahl von Einschränkungen in den Generierungsprozess eingebracht werden. So können beispielsweise die Fahrzeugtypen, Minimal- und Maximalgeschwindigkeiten, Minimal- und Maximalentfernungen usw. definiert werden.

OD Matrizen

OD Matrizen, oder Origin-Destination Matrizen, werden oft von Verkehrsbehörden zur Verfügung gestellt. In diesen Matrizen sind die Beziehungen zwischen den Zielen sowie den Quellen hinterlegt, wobei in der Regel eine Einteilung des untersuchten Gebiets in verschiedene Zonen erfolgt. Wie diese Matrizen zustande kommen ist nicht standardisiert. Oft werden jedoch Umfragen sowie Verkehrszählungen als Basis zur Erstellung dieser Matrizen herangezogen.
Mittels des `od2trips`-Programms können schlussendlich aus diesen Daten Routing-Files für SUMO generiert werden.

Beobachtungspunkte

Neben den OD-Matrizen wird von Behörden auch das Verkehrsaufkommen an gewissen Punkten des Straßennetzwerks zur Verfügung gestellt. Diese Daten werden meistens durch Messung mittels Induktionsschleifen und/oder händische Zählung ermittelt.
Mit dem Programm `dfrouter` ist es möglich, aus diesen Daten Routen für SUMO zu erstellen. An dieser Stelle soll angeführt werden, dass es sich bei der Rückrechnung von einzelnen Messpunkten auf den Verkehrsfluss um keine triviale Problemstellung handelt. Hinter diesem Vorgang steckt ein mehrstufiger Prozess mit durchaus komplexer Mathematik.

Bevölkerungsstatistiken

Eine weitere Möglichkeit Verkehr zu definieren, basiert auf der Verwendung von Bevölkerungsstatistiken. Mit dem Tool `activitygen` ist es möglich, aufgrund von verschiedenen Aktivitäten wie Arbeit, Schule, etc. und den zugehörigen Beförderungsmitteln entsprechende Routing-Files für SUMO zu generieren. Der Prozess ist sehr aufwändig da zuvor ein umfangreiches Parameterfile erstellt werden muss. Auf der anderen Seite werden sogar detaillierte Parameter wie Arbeitslosenquote, Pensionsantrittsalter oder Wahrscheinlichkeiten, dass die Einwohner den öffentlichen Verkehr dem privaten vorziehen, berücksichtigt. All das setzt natürlich ein sehr genaues Wissen über die Bevölkerungsstruktur im untersuchten Gebiet sowie dessen Umgebung, voraus.

12.8.3 Simulation

Sind das Straßennetzwerk sowie die Routen der Fahrzeuge definiert, so stehen alle wesentlichen Daten zur Verfügung welche für die Simulation benötigt werden. Es ist möglich, weitere Konfigurationsdateien zur Verfügung zu stellen. Die umfasst vor allem Ampelprogramme, Induktionsschleifen sowie Stationen des öffentlichen Verkehrs. Weiters können dynamische Geschwindigkeitslimits sowie Umleitungen, basierend auf verschiedenen Größen der Simulation, definiert werden. Detailliertere Informationen würden den Umfang der Arbeit sprengen weshalb an dieser Stelle auf die SUMO-Dokumentation verwiesen sei.

Offline-Simulation

Haupteinsatzgebiet von SUMO ist die Offline-Simulation. Dies bedeutet, dass der Anwender Simulationen so schnell wie nur möglich durchführen möchte um verschiedenste Varianten miteinander vergleichen zu können.
Als Output-Format kommt hauptsächlich XML zum Einsatz. Es können praktisch alle Daten, welche im Zuge der Simulation generiert werden, abgespeichert werden. Dies geht sogar so weit, dass Geräusch- sowie Abgasemissionen sowie Signale von Induktionsschleifen abgespeichert werden können.
Es stehen Skripte zur Verfügung, welche die generierten XML-Daten in CSV oder Protobuf Dateien zur weiteren Verarbeitung umwandeln

Online-Simulation

Möchte man SUMO in einem Fahrsimulator betreiben, so ist es wesentlich, dass alle Updates zeitsynchron durchgeführt werden. Weiters müssen gewissen Daten wie Fahrzeugpositionen sowie Orientierungen permanent an andere Prozesse übermittelt werden, um beispielsweise die Fahrzeuge in der Visualisierung anzuzeigen oder aber auch um als Eingangsdaten für das Radarmodell zu dienen.
SUMO bietet zu diesem Zweck eine Schnittstelle Namens TraCI - Traffic Control Interface an, welches es ermöglicht, SUMO im Online-Modus zu betreiben und ein Subset der von SUMO generierten Daten abzufragen, oder aber auch zu setzen.

TraCI basiert auf TCP/IP, wobei SUMO explizit als Server gestartet werden muss um eine Interaktion zu ermöglichen. Die Community stellt Interfaces für Java, Python, .Net sowie Matlab zur Verfügung. Für C++ können Teile der UnitTests, welche für die Entwicklung von SUMO Verwendung finden, als Referenz herangezogen werden. Weiters steht eine vollständige Dokumentation der Datenpakete zur Verfügung.
Über TraCI wird jeder Update-Schritt explizit angefordert wodurch eine Synchronisation ermöglicht wird. Außerdem ist es möglich, die Position des Ego-Fahrzeugs in die Verkehrsflusssimulation einfließen zu lassen, und somit eine dynamische Interaktion mit dem restlichen simulierten Verkehr zu ermöglichen. Natürlich können auch Positionen, Geschwindigkeiten und Beschleunigungen beliebiger Fahrzeuge gesetzt oder aber auch ein Fahrstreifenwechsel veranlasst werden. Weiters können sämtliche Ampeln kontrolliert oder aber auch Fahrstreifen sowie Straßen gesperrt werden.

12.8.4 Ausblick

Mit SUMO und dem Fahrsimulator ist es möglich, eine Simulationsumgebung zu entwickeln, deren Möglichkeiten praktisch unbegrenzt sind. Insbesondere in den Forschungsbereichen Car2Car sowie autonomes Fahren bietet sich die Integration von SUMO in den Fahrsimulator an.
Erste Tests haben gezeigt, dass die TraCI-Schnittstelle alle Features zur Verfügung stellt welche für eine erfolgreiche Einbindung von SUMO in den Fahrsimulator notwendig sind. Mit PTV Vissim steht ein kommerzielles Produkt zur Verfügung, welches ein sehr ähnliches Feature-Set hat wie SUMO, jedoch wesentlich benutzerfreundlicher ist. Leider bietet es zum momentanen Zeitpunkt noch keine Online-Interaktion an. Laut Auskunft des Herstellers ist diese Funktion jedoch für ein zukünftiges Release geplant.

Die Empfehlung des Autors geht dahin, die *ScenarioLib* in Zukunft durch eine vollständige Verkehrsflusssimulation zu ersetzen. Dies hat zwar einen relativ hohen Implementierungsaufwand zur Folge, dem stehen jedoch Vorteile viel größerem Ausmaßes gegenüber. Die Flexibilität und Funktionalität welche diese Produkte im Laufe der Zeit erhalten haben, sind durch eigene Implementierungen praktisch nicht zu erreichen.

Kapitel 13

Eye-Tracker

13.1 Einleitung

Wie bereits in Kapitel 2.1.3 beschrieben, findet im Simulator ein Eye-Tracking System der Firma Smart Eye Verwendung. Dieses System liefert neben der Kopfposition auch die Augenpositionen sowie Blickrichtungen, wobei die Augenpositionen für die Realisierung der 3D-Visualisierung mit Hilfe der Parallaxbarrieren benötigt werden. Die ermittelten Daten werden über UDP, oder wahlweise auch über CAN, an andere Prozesse zur weiteren Datenverarbeitung übermittelt. Die mitgelieferte Software aus Abbildung 13.1 ermöglicht neben dem Eye-Tracking an sich noch die Kalibrierung des Gesamtsystems als auch die Definition der zu übertragenden Daten.

Um die für das Stereo-Bild wichtigen Daten des Eye-Tracker zu bekommen, hat das Fraunhofer-Institut ein Plugin für das Instant-Reality Framework erstellt, welches nach dem selben Schema wie in Kapitel 7.2 beschrieben, erstellt und integriert wurde. Nachdem dieses Plugin diese Daten direkt empfängt, hat die restliche Simulation keinen Einfluss auf das korrekte Arbeiten des Visualisierungssystems.
Obwohl die Fahrsimulation die Daten des Eye-Trackers zum gegebenen Zeitpunkt nicht benötigt, so wurde doch ein Interface implementiert, um die Daten zeitsynchron mit all

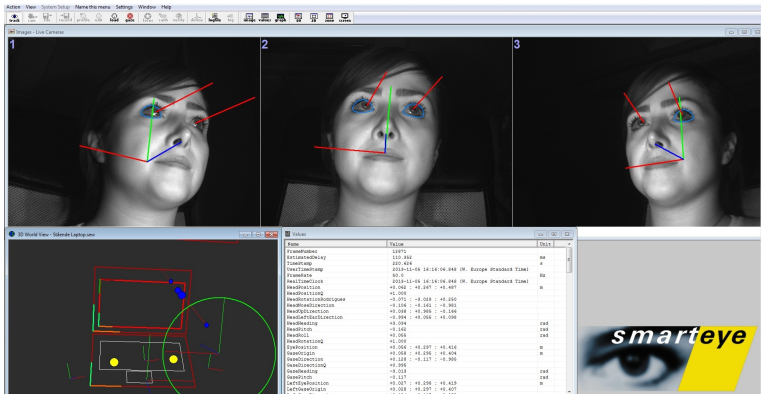

Abbildung 13.1: Smart Eye Pro [14]

den anderen geloggten Parametern aufzeichnen zu können. Sollten in Zukunft die Signale für beispielsweise Usability-Untersuchungen, Unfallursachenforschung oder Fahrertraining benötigt werden, so ist diese Funktionalität bereits implementiert.

13.2 SmartEyeLib

Nachdem die UDP-Datenpakete nicht immer fix sind und die Struktur in der Software jederzeit geändert werden kann, ist es nicht möglich, mit fixen Datenpaketen bzw. Datenstrukturen zu arbeiten. Man müsste die in den Daten enthaltenen Metadaten durchsuchen um mehr über den genauen Aufbau des Datenpakets zu erfahren. Glücklicherweise stellt Smart Eye eine Reihe von Header-Dateien zur Verfügung welche es ermöglichen, Bytestreams auf Gültigkeit zu überprüfen beziehungsweise gezielt nach gewissen Signalen zu durchsuchen sowie deren Datentypen zu parsen.
Basieren auf dieser SEPacketAPI, wie sie von Smart Eye genannt wird, wurde, wie in Kapitel 3.3 erklärt, eine C++-Bibliothek erstellt welche die einfache Integration in Matlab/Simulink ermöglicht.
Das Interface welches nach außen hin zur Verfügung gestellt wird ist in Listing 13.1 zu sehen. Zum momentanen Zeitpunkt werden nur Augenpositionen und deren Blickrichtungen zur Verfügung gestellt, wobei dies jederzeit einfach um weitere Signale, wie beispielsweise Informationen über den Lidschlag, erweitert werden kann.

```
1  #pragma once
2
3  #ifdef __cplusplus
4  extern "C" {
5  #endif
6
7      // Define datatypes
8      struct Vec3 {
9          double x;
10         double y;
11         double z;
12     };
13     typedef Vec3 EyePosition;
14     typedef Vec3 GazeDirection;
15
16     enum Eye {
17         EyeLeft = 0,
18         EyeRight = 1
19     };
20
21     // Initialize Library
22     __declspec(dllexport) void initSmartEyeInterface(const char* configPath);
23     // Get eye position
24     __declspec(dllexport) const EyePosition* getEyePosition(const Eye side) const;
25     // Get gaze direction
26     __declspec(dllexport) const GazeDirection* getGazeDirection(const Eye side) const;
27     // Shutdown library
```

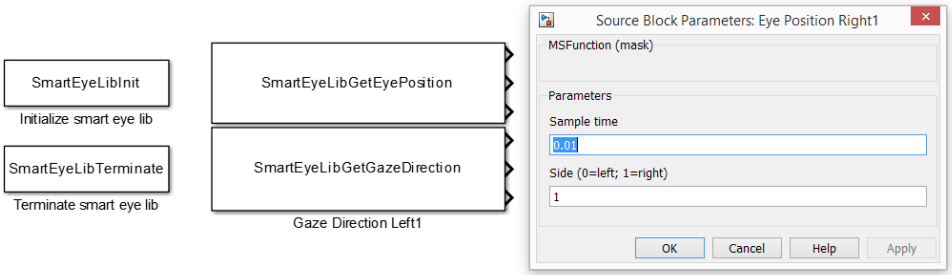

Abbildung 13.2: *SmartEyeLib* - Simulink

```
28      __declspec(dllexport) void stopSmartEyeInterface(void) const;
29
30  #ifdef __cplusplus
31  }
32  #endif
```

Listing 13.1: Interface der *SmartEyeLib*

Wie auch schon die *ScenarioLib* oder die *VisionInterfaceLib* verfügt auch die *SmartEyeLib* über ein Konfigurationsfile, welches zu Beginn der Initialisierung geladen und interpretiert wird. Listing 13.2 zeigt beispielhaft wie die Datei aussehen könnte.

```
1  <?xml version="1.0" encoding="utf-8"?>
2  <SmartEyeConfig>
3    <!-- Port where Eye Tracking Data is sent to -->
4    <Port>9000</Port>
5    <!-- Path where log file of SmartEyeLib should be written -->
6    <LogFilePath>C:/local/log/</LogFilePath>
7    <!-- Name of log file -->
8    <LogFileName>smartEyeLog</LogFileName>
9    <!-- Define log level -->
10   <LogMode>INFO</LogMode>
11 </SmartEyeConfig>
```

Listing 13.2: *SmartEyeLib* Konfigurationsfile

Wesentlich ist hierbei das Port-Element welches definiert, auf welchem UDP-Port auf die Eye-Tracking Daten gehört werden soll. Diese Konfiguration muss ident mit jener sein, welche innerhalb von Smart Eye Pro eingestellt ist. Weiters ist zu beachten, dass der Eye-Tracker die Daten auf eine Broadcast-Adresse verschickt, da ansonsten nicht das Plugin welches auf dem Visualisierungsrechner läuft als auch die *SmartEyeLib* auf dem Simulationsrechner gleichzeitig die Signale empfangen können.
An dieser Stelle sei auch noch einmal auf Abbildung 7.3 aus Kapitel 7.2.1 verwiesen, welche den Kommunikationsfluss zwischen den beteiligten Rechnern veranschaulicht

Kapitel 14

Zusammenfassung und Ausblick

14.1 Zusammenfassung

Im Rahmen dieser Masterarbeit erfolgte eine tiefgehende Auseinandersetzung mit dem großen Gebiet der Fahrsimulation, wobei hierbei verschiedenste Konzepte sowie Realisierungen von Software-Komponenten entstanden sind, welche für den Betrieb solch eines Simulators unumgänglich sind. Dies schließt sowohl Simulationsmodelle, Kommunikationsschnittstellen als auch Peripherie-Software mit ein.

Zu Beginn wurde in Kapitel 1 das Forschungsprojekt *MueGen Driving* vorgestellt, in dessen Rahmen der Aufbau des Fahrsimulators erfolgte. Dieses Projekt setzt sich mit dem geschlechts- und altersspezifischen Fahrverhalten in kritischen Situationen bei verschiedenen Straßenverhältnissen auseinander. Ein Simulator stellt optimale Voraussetzungen bereit, um unter gleichbleibenden Randbedingungen Studien mit einer großen Anzahl an Versuchspersonen durchzuführen.

Nach einer State of the Art Recherche in Abschnitt 1.4 wurden die genauen Anforderungen sowie die zu verbauenden Komponenten festgelegt. Dies umfasst unter anderem die Umsetzung des Lenkrads sowie Bremspedals durch elektrisch aktuierte Force Feedback Systeme sowie die Realisierung der Visualisierung mit Hilfe von 8 Bildschirmen. Als Besonderheit ist es möglich, den Simulator mit Hilfe des verbauten Eye-Trackers sowie der Parallax-Barrieren welche vor den Monitoren angebracht werden können, in einem 3D-Modus zur Erzeugung von stereoskopischen Bildern zu betreiben. Kapitel 2 beschreibt im Detail alle verbauten Komponenten.

In Abschnitt 3 wird tiefergehend auf die Einbindung der Echtzeitumgebung PTWinSim eingegangen. Es wird anhand eines Beispiels beschrieben, wie mit Hilfe von Simulink-Modellen Echtzeit-Tasks generiert werden können, als auch wie eigene C++ Bibliotheken integriert werden können.
Im Weiteren werden die verschiedenen Module welche innerhalb der Echtzeitumgebung laufen beschrieben. Dies umfasst beispielsweise die Fahrdynamik-Simulation AVL VSM, CAN- sowie andere IO-Interfaces, ACC- sowie AEB-Modelle als auch verschiedene Interface-

Komponenten welche für die Kommunikation mit der Visualisierung als auch der Akustiksimulation zuständig sind.

Eine der wichtigsten Komponenten eines Fahrsimulators ist die Simulation des Fahrzeugs an sich. Zu diesem Zweck fungierte die Firma AVL List GmbH als Projektpartner und hat mit AVL VSM die Simulation der Fahrdynamik zur Verfügung gestellt. Auch diese läuft innerhalb der Echtzeitumgebung, wobei die Integration in enger Zusammenarbeit mit den Projektpartner erfolgte. Abschnitt 4 setzt sich mit der Fahrdynamik-Simulation auseinander und beschriebt, wie der Datenaustausch mit der restlichen Fahrsimulation realisiert wurde.

Als Alternative zu AVL VSM ist es möglich, IPG CarMaker als Simulationsmodell einzusetzen. Kapitel 5 geht näher auf die Einbindung in die Gesamtsimulation sowie die daraus resultierenden Vor- und Nachteile ein.

Um verschiedene Analog-Signale einlesen sowie ausgeben zu können, wird ein Produkt der Firma Dewetron eingesetzt. Nachdem die zur Verfügung stehende Software keine Integration in eine Echtzeitumgebung ermöglicht, musste eine C++ Bibliothek entwickelt werden welche genau dies bewerkstelligen kann. Abschnitt 6 stellt das verwendete Device sowie dessen Spezifikationen im Detail vor und beschreibt wie anhand von Bibliotheken welche von Nation Instruments zur Verfügung gestellt werden, die Integration in die Echtzeitumgebung erfolgen konnte.

Die Visualisierung wurde in enger Zusammenarbeit mit dem Fraunhofer-Institut für Graphische Datenverarbeitung entwickelt. Um alle Objekte korrekt darstellen zu können, mussten zwei Software-Komponenten entwickelt werden. Eine C++ Bibliothek läuft innerhalb der Echtzeitumgebung und stellt aus der Simulation alle für die Visualisierung notwendigen Daten über UDP bereit. Als Gegenstück fungiert ein Plugin, welches im Kontext des InstantReality Frameworks auf dem Master der Visualisierungs-Rechner läuft. Dieses Plugin empfängt die UDP-Daten und manipuliert den Szenengraphen der Visualisierung entsprechend.
Kapitel 7 stellt beide Komponenten im Detail vor und beschreibt, wie der Datenaustausch sowie die Manipulation des Szenengraphen umgesetzt wurde.

Nachdem der für den Fahrsimulator verwendete Mini Countryman über keine fortschrittlichen Fahrassistenzsysteme und demnach auch nicht über die entsprechenden Bedienelemente verfügte musste ein Weg gefunden werden, eine Interaktion zwischen Fahrer und Simulation zu ermöglichen. Zu diesem Zweck wurden im Simulator zwei auf Android basierende Tablets installiert, wobei eines als reines Anzeigeinstrument fungiert während das zweite zur interaktiven Bedienung herangezogen wird. Kapitel 8 geht tiefergehend auf die Umsetzung sowie die Integration in die Gesamtsimulation ein.

Um die Entwicklung von neuen Komponenten für den Fahrsimulator zu erleichtern wurde eine Applikation entwickelt welche es ermöglicht, während der laufenden Simulation beliebige Daten der Modelle zu lesen beziehungsweise auch zu setzen. Abschnitt 9 beschreibt

dieses sowie weitere Features und erläutert, wie der Datenaustausch über UDP im Detail von statten geht.

Finden im Fahrsimulator Studien mit Probanden statt, so muss der Operator in der Lage sein den Simulator auch vom Beifahrersitz aus bedienen zu können. Um dies zu ermöglichen wurde eine Tablet-Applikation entwickelt, welche die wichtigsten Kontroll-Aufgaben umsetzen kann. Sie schließt einerseits das Starten und Stoppen der Simulation, das Wechseln von Winter- auf Sommerszenario, das Setzen von Straßenparametern als auch das Triggern von Events mit ein.
In Kapitel 10 wird dargestellt wie die Umsetzung erfolgte und wie die UDP-Datenpakete zur Kommunikation definiert wurden.

Neben der Fahrdynamiksimulation wurde auch die Umsetzung der Akustiksimulation in Kooperation mit der AVL List GmbH durchgeführt. In Abschnitt 11 wird auf die verwendeten Programme sowie auf die Einbindung in die Gesamtsimulation mittels eines einfachen UDP-Protokolls eingegangen.

Einen weiteren wesentlichen Punkt eines Fahrsimulators stellt der dynamische Verkehr dar. Um repräsentable Studien durchführen zu können muss die Möglichkeit gegeben sein, ein und das selbe Fahrmanöver bei höchster Reproduzierbarkeit unter den exakt selben Randbedingungen durchführen zu können. Um dies zu ermöglichen wurde wie in Abschnitt 12 beschrieben eine Bibliothek entwickelt welche in der Lage ist bestimmte Fahrmanöver zu definieren und nachzustellen.
Im erwähnten Kapitel werden detailliert das verwendete Dateiformat sowie die verschiedenen Parameter, welche für die Fahrmanöver zur Verfügung stehen, beschrieben.

Für die Umsetzung der stereoskopischen Visualisierung sowie zur Untersuchung von Points of Interest wurde im Fahrsimulator ein Eye-Tracker der Firma SmartEye verbaut. Abschnitt 13 beschreibt abschließend die Entwicklung einer Library welche in der Lage ist, verschiedenste vom Eye-Tracker über UDP übertragene Daten zu empfangen und innerhalb der Echtzeitumgebung anderen Komponenten zur Verfügung zu stellen.

14.2 Ausblick

Fahrsimulatoren eignen sich, wie bereits zu Beginn in Kapitel 1.4.1 erläutert, hervorragend dazu neue Systeme zu entwickeln beziehungsweise um auf den verschiedensten Forschungsgebieten aktiv zu werden. Gerade das große Gebiet rund um autonomes Fahren bietet eine schier unendliche Anzahl an Themen welche mit Hilfe des Simulators entwickelt sowie getestet werden können.

Gleichzeitig schreitet die Wandlung des Automobils von einer rein mechanischen Maschine hin zu einem Supercomputer ungehindert fort. In diesem Kontext gibt es ebenfalls viele Problemstellungen für die es momentan noch keine zufriedenstellenden Lösungen gibt. Vor allem die Car2Car-Kommunikation sowie das große Thema Security bieten noch großes Entwicklungspotential.

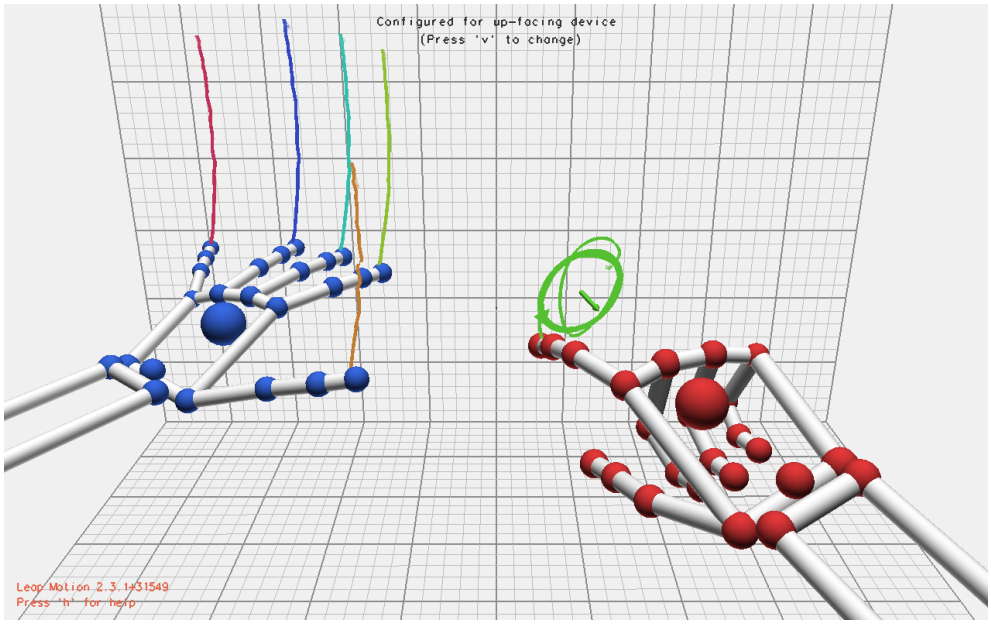

Abbildung 14.1: Gestenerkennung mittels Leap Motion

Nachdem dem Fahrer eines Automobils immer mehr Aufgaben des eigentlichen Fahrens abgenommen werden, treten andere Aspekte in den Vordergrund. So nimmt beispielsweise die Müdigkeitserkennung einen wichtigen Teil momentaner Forschungsaktivitäten ein, nachdem der Lenker immer weniger Aufmerksamkeit dem Fahren an sich widmet, jedoch trotzdem jederzeit in der Lage sein muss, korrigierend eingreifen zu können.

Auch Wandeln sich die Anforderungen welche die Endkunden an die Bedienbarkeit neuer Auto-Generationen stellen. In Zeiten moderner Smartphones sind Sprach- und Gestenerkennung fast schon zur Selbstverständlichkeit geworden und werden somit auch im Automobil erwartet. Auch dieser Bereich bietet großen Spielraum für zukünftige Forschungsaktivitäten. Erste Experimente zur Sprach- sowie Gestenerkennung mittels Leap-Motion konnten im Simulator bereits durchgeführt werden.

Ein wesentlicher Punkt beim momentanen Stand des Fahrsimulators stellt die fehlende Bewegungsplattform dar. Im Rahmen der Studien welche im Simulator durchgeführt wurden trat bei einigen wenigen Probanden Simulator-Sickness auf. Dies ist auf die Tatsache zurückzuführen, dass der Eindruck welcher durch die Visualisierung vermittelt wird nicht mit jenem übereinstimmt, welcher über den Gleichgewichtssinn aufgenommen wird. Dies ist besonders bei Beschleunigungs- sowie Verzögerungsvorgängen wahrzunehmen. Mit Hilfe einer Motion-Plattform kann dieser Diskrepanz entgegengewirkt werden und der allgemeine Realitätsgrad noch einmal erhöht werden.

Aus diesem Grund ist für die Zukunft die Einbindung einer entsprechenden Plattform geplant. Die Konstruktion des Fahrsimulator berücksichtigt bereits die mögliche Integration einer Bewegungsplattform und ermöglicht, aus Gründen der Gewichtsreduktion, die

Abtrennung des Fahrzeugs hinterhalb der B-Säule.

In Kapitel 12.8 wurde bereits näher auf das Thema der Verkehrsflusssimulation eingegangen. Auch hier wird bereits seitens des Instituts für Fahrzeugtechnik daran gearbeitet, solch ein System in die Fahrsimulation zu integrieren. Dies stellt eine ideale Kombination dar, da dann am Computer erstellte Modelle direkt mit Hilfe des Simulators, samt Visualisierung und Akustik, beurteilt werden können.

Abschließend soll noch auf die Durchführung möglicher HIL-Tests hingewiesen werden. Da das Institut bereits über einen hervorragenden Reifenprüfstand verfügt liegt es nahe, diesen im Verbund mit dem Fahrsimulator zu betreiben.
Weiters wird am Institut exzessiv im Bereich der Assistenzsysteme geforscht. Nachdem die Radartechnik ein wesentlicher Bestandteil dieser ist werden Überlegungen durchgeführt, den Fahrsimulator im Zusammenspiel mit einem echten Radar-Modul samt einem noch zu entwickelnden Radar-Stimulator zu betreiben. Durch diese Kombination wäre es möglich, echte Radar-Daten für die restliche Simulation zu verwenden, was eine Vielzahl neuer möglicher Forschungsprojekte ermöglicht.

Anhang A

Begriffsbestimmung

ABS	Anti Blockier System
ACC	Adaptive Cruise Control
AEB	Automatic Emergency Braking
API	Application Programming Interface
CC	Cruise Control
ECU	Engine Control Unit
DAQ	Data Acquisition
DAVE	Definitely Affordable Virtual Environment
DIL	Driver In the Loop
FFD	Force Feedback Device
FMCW	Frequency Modulated Continuous Wave
FTG	Institut für Fahrzeugtechnik Graz
GUI	Graphical User Interface
HIL	Hardware In the Loop
IEM	Institut für Elektronische Musik und Akustik
IG	Image Generator
LIN	Local Interconnect Network
NI	National Instruments
MFC	Model View Controller
MFC	Model View Presenter
MVVM	Model View View-Model
REST	Representational State Transfer
RTW	Realtime Workshop
SDK	Software Development Kit
TLC	Target Language Compiler
VRML	Virtual Reality Modeling Language
X3D	Extensible 3D

Literaturverzeichnis

[1] E2m. http://www.e2mtechnologies.eu/index.php/portfolio/eM6-720-4000/. Zuletzt aufgerufen: 2015-10-18.

[2] Moog. http://www.moog.com/products/motion-systems/motion-bases/electric-pneumatic-motion-basebrmb-ep-6dof-60-8000kg-br-formerly-e-cue-660-8000-/. Zuletzt aufgerufen: 2015-10-18.

[3] Cablerobot. http://www.ipa.fraunhofer.de/en/cable-driven_parallel_robots.html. Zuletzt aufgerufen: 2015-10-18.

[4] Markus Peer. Konzept, Konstruktion und Aufbau eines statischen PKW-Fahrsimulators. Master's thesis, Institut für Fahrzeugtechnik, TU Graz, 2015, laufend.

[5] Michael Josef Kappaun. Simulatorversuche zur Beurteilung der Fahrerinteraktion mit verschiedenen Assistenzsystemen. Bachelor thesis, Institut für Fahrzeugtechnik, TU Graz, 2015.

[6] Podium technology ltd. http://www.podiumtechnology.co.uk/. Zuletzt aufgerufen: 2015-08-10.

[7] Inc. The Mathworks. *Real-Time Workshop, Getting Started.* The Mathworks, Inc., Natick, USA, June 2004.

[8] Matthias Pirstinger. Einbindung der elektrischen Force-Feedback Komponenten in einen Fahrsimulator. Master's thesis, Institut für Fahrzeugtechnik, TU Graz, 2015.

[9] Bert Breuer und Karlheinz H. Bill. *Bremsenhandbuch: Grundlagen - Komponenten - Systeme - Fahrdynamik.* Springer Vieweg, 2012.

[10] Dewe image. https://ccc.dewetron.com/pr/dewe-50-usb2-16. Zuletzt aufgerufen: 2015-10-19.

[11] Instant reality. http://www.instantreality.org/. Zuletzt aufgerufen: 2015-08-16.

[12] Dave. http://www.cgv.tugraz.at/dave/. Zuletzt aufgerufen: 2015-08-16.

[13] Sumo screenshot. http://sumo.dlr.de/trac.wsgi/ticket/1522. Zuletzt aufgerufen: 2015-10-17.

[14] Smart eye pro. http://www.rogue-resolutions.com/system/smart-eye-pro-5-10-3d-eye-tracker/. Zuletzt aufgerufen: 2015-10-17.

[15] Smart eye pro. http://smarteye.se/wp-content/uploads/2014/12/Smart-Eye-Pro.pdf. Zuletzt aufgerufen: 2015-08-26.

[16] Instantio plugin. http://doc.instantreality.org/media//uploads/downloads/InstantIO_Manual-20080318.pdf. Zuletzt aufgerufen: 2015-08-22.

[17] Vn1640factsheet. http://www.vector.com/portal/medien/cmc/factsheets/VN16xx_FactSheet_DE.pdf. Zuletzt aufgerufen: 2015-09-01.

[18] Ni 6251 specifications. http://www.ni.com/pdf/manuals/375213b.pdf. Zuletzt aufgerufen: 2015-10-11.

[19] Dewe features. https://ccc.dewetron.com/dl/537467fe-3644-4494-a7e1-4756d9c49862. Zuletzt aufgerufen: 2015-09-01.

[20] Sumo. http://www.dlr.de/ts/en/desktopdefault.aspx/tabid-9883/16931_read-41000/. Zuletzt aufgerufen: 2015-09-17.

[21] Michael W. Halle. Raytracing for parallax 3-d display. *SPIE Proceeding*, February 1994.

[22] Szymon Jakubczak. Raytracing for parallax 3-d display. *MIT CSAIL*, February 2009.

[23] Héctor Benedí Bozalongo. Development of an ACC model for a vehicle dynamic simulation software. Master's thesis, Institut für Fahrzeugtechnik, TU Graz, 2014.

[24] Anna Schieben, Daniel Damböck, Johann Kelsch, Herbert Rausch, and Frank Flemisch. Haptisches feedback im spektrum von fahrerassistenz und automation. *Aktive Sicherheit durch Fahrerassistenz*, April 2008.

[25] Sajjad Samiee. Development of a Force-Feedback mechanism for a brake pedal. Technical report, Institut für Fahrzeugtechnik, TU Graz, April 2013.

[26] Philipp Duswald. Integration of an Eye Tracking System into a driving simulator for automatized tracking of the driver's gaze direction. Bachelor thesis, Institut für Fahrzeugtechnik, TU Graz, 2014.