

Entwicklungsumgebung für das autonome Fahren

Am Beispiel eines Modelfahrzeugs

Diplomarbeit/Masterarbeit

Zur Erlangung des akademischen Grades

Master of Science

der Fachhochschule FH Campus Wien

Vorgelegt von:

Sergiu-Petru Tabacariu BSc.

Personenkennzeichen

1410538003

Erstbegutachter:

FH-Prof. Dipl.-Ing. Dipl.-Ing. Dr. techn. Dr. techn. Dr.-Ing. Gernot Kucera

Zweitbegutachter:

FH-Prof. Dipl.-Ing. Gerhard Engelmann

Eingereicht am:

05.08.2018

Erklärung:

Ich erkläre, dass die vorliegende Diplomarbeit/Masterarbeit von mir selbst verfasst wurde und ich keine anderen als die angeführten Behelfe verwendet bzw. mich auch sonst keiner unerlaubter Hilfe bedient habe.

Ich versichere, dass ich diese Diplomarbeit/Masterarbeit bisher weder im In- noch im Ausland (einer Beurteilerin/einem Beurteiler zur Begutachtung) in irgendeiner Form als Prüfungsarbeit vorgelegt habe.

Weiters versichere ich, dass die von mir eingereichten Exemplare (ausgedruckt und elektronisch) identisch sind.

Datum: Unterschrift:

Danksagung

Ich möchte mich an dieser Stelle bei all jenen bedanken, die durch ihre fachliche und persönliche Unterstützung zum Gelingen dieser Masterarbeit beigetragen haben.

Besonderer Dank gilt meinen Professoren und Betreuern Dipl.-Ing. Dipl.-Ing. Dr. techn. Dr. techn. Dr.-Ing. Gernot Kucera und Dipl.-Ing. Gerhard Engelmann. Durch ihre wertvollen Leitfragen sowie Ratschläge haben sie mich motiviert und wesentlich zum Gelingen dieser Arbeit beigetragen. Auch konnte ich mir in ihrem Unterricht viele fachliche Kompetenzen aneignen, die in diese Arbeit eingeflossen sind.

Meinen Eltern Rosa und Stefan Tabacariu danke ich für ihre Unterstützung und Ermutigung, stets über eigene Schatten und Grenzen zu springen. Sie haben mir nicht nur das Studium ermöglicht, sondern mir dabei geholfen, Herausforderungen zu meistern. Genauso danke ich auch meinen Schwiegereltern Elisabeth und Heimo Schodterer für ihren moralischen und praktischen Beistand.

Meiner geliebten Ehefrau Hanna danke ich für ihre vielfältige Unterstützung. Durch die zahlreichen Stunden des Korrekturlesens hat die Arbeit sehr an Qualität und Verständlichkeit gewonnen. Dazu war sie mir eine ständige motivierende Hilfe und hat mich ermutigt, über mich selbst hinauszuwachsen. Sie ist meine beste Freundin und wichtigste Unterstützerin.

Kurzfassung

Die vorliegende Masterarbeit beschäftigt sich mit der Erforschung des autonomen Fahrens. Dazu wurde eine modulare, mobile und quelloffene Entwicklungsumgebung erstellt, die als autonomes Modellfahrzeug umgesetzt wurde. Dieses ist Teil der Entwicklungsumgebung, dient selbst als Entwicklungswerkzeug und kann zur Durchführung von HIL-Tests für Softwareapplikationen für das autonome Fahren verwendet werden. Diese kostengünstige und quelloffene Plattform soll anderen Entwicklern den Zugang zur weiteren Erforschung des autonomen Fahrens und seiner Teilaufgaben erleichtern, da auf dem Prototyp Tests ohne Risiko durchgeführt werden können. Deren Erkenntnisse können später auf Fahrzeugen im vollen Maßstab eingesetzt werden.

Für die Entwicklungsplattform wurden auch Applikationen, die Teilaufgaben des autonomen Fahrens umsetzen, erstellt. Die Systemstruktur ist modular aufgebaut, sodass die einzelnen Applikationen unabhängig voneinander erweitert oder durch andere ausgetauscht werden können. Die so erstellte Entwicklungsumgebung konnte auf einer Teststrecke im Innenraum erfolgreich getestet werden, wobei eine Fahrspur verfolgt sowie Stoppschilder und Hindernisse erkannt wurden. In einer späteren Weiterentwicklung können sowohl die Sensorik als auch die softwaretechnischen Umsetzungen der Erkennungsverfahren optimiert werden.

Abstract

This master thesis deals with the investigation of autonomous driving. For that purpose, a modular, mobile and open-source development environment was built and implemented on an autonomous model vehicle. This vehicle is part of the development environment, can be used as a development tool as well as for HIL-tests of software applications for autonomous driving. This economical and open-source platform is meant to facilitate access to further investigations of autonomous driving and its partial aspects, as tests with this prototype can be conducted without risk. The results obtained can later be applied to full-scale vehicles.

Additionally, for the development platform, applications that cover partial aspects of autonomous driving were compiled. The structure of the system is modularly organized, in order to upgrade or replace individual applications independently. The thus compiled development environment was successfully tested on an indoor test track, including the following of a traffic lane and the recognition of stop signs and obstacles. In further developments, the sensor system as well as software-related implementations of recognition techniques might be optimized.

Abkürzungsverzeichnis

ABS	Acrylnitrit-Butadien-Styrol-Copolymer
ABS	Antiblockiersystem
ADAS	Advanced Driver Assistance System
ADTF	Automotive Data and Time-Triggered Framework
ANN	Artificial Neural Network
ASCII	American Standard Code for Information Interchange
AUTOSAR	Automotive Open System Architecture
CAN	Controller Area Network
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
CV	Computer Vision
CVS	Current Versions System
DARPA	Defense Advanced Research Projects Agency
DGPS	Differential Global Positioning System
DIN	Deutsches Institut für Normierung
DNN	Deep Neural Network
DSP	Digital Signal Processor
DVCS	Distributed Version Control System
ESC	Electronic Speed Controller
ESP	Elektronisches Stabilitätsprogramm
FPS	Frames per Second
GCC	GNU Compiler Collection
GND	Ground
GPIO	General Purpose Input/Output
GPS	Global Positioning System
GPU	Graphic Processing Unit
GTK	GIMP Tool-Kit
HDMI	High Definition Multimedia Interface
HIL	Hardware in the Loop
HMI	Human Machine Interface
HOG	Histogram of Oriented Gradients
I ² C	Inter-Integrated Circuit
IC	Integrated Circuit
IDE	Integrated Development Environment
IEC	International Electrotechnical Commission

IPT	Inverse Perspective Transformation
ISO	International Organization for Standardization
LIDAR	Light Detection and Ranging
MCU	Motor Control Unit
NiMH	Nickel-Methalhybrid
OE	Output Enable
OMG	Object Management Group
OpenCV	Open Computer Vision
OS	Operating System
PC	Personal Computer
POSIX	Portable Operating System Interface
PWM	Pulsweitenmodulation
RADAR	Radio Detection and Ranging
RAM	Random Access Memory
RF	Radio Frequency Module
ROI	Region of Interest
SAE	Society of Automotive Engineers
SCL	Serial Clock
SDA	Serial Data
SIFT	Scale-invariant Feature Transform
SoC	System on a Chip
SSH	Secure Shell
StVO	Straßenverkehrsordnung
SURF	Speed Up Robust Features
UML	Unified Modeling Language
USB	Universal Serial Bus
V2I	Vehicle to Infrastructure
V2V	Vehicle to Vehicle
V2X	Vehicle to X
VEHIL	Vehicle Hardware in the Loop
VNC	Virtual Network Computing
WLAN	Wireless Local Area Network
XML	Extensible Markup Language
zFAS	Zentrales Fahrzeugassistentenzsystem

Schlüsselbegriffe

Autonomous Driving
Advanced Driver Assistance System (ADAS)
Autonomous Robot
Mobile Robot
Remote Controlled Robot
Development Tools
Development Environment
Lane Detection
Line Detection
Object Detection
Traffic Sign Detection
Obstacle Detection

Inhaltsverzeichnis

1. EINLEITUNG	1
1.1 Stand der Technik	2
1.2 Motivation	4
1.3 Aufgabenstellung.....	4
1.4 Methoden	5
1.5 Verwandte Arbeiten	5
2. GRUNDLAGEN	9
2.1 Systemarchitektur.....	9
2.1.1 Sensorik	10
2.1.2 Sensordatenverarbeitung	14
2.1.3 Planung	15
2.1.4 Fahrzeugsteuerung	16
2.2 Bildverarbeitung.....	17
2.2.1 Fahrbahnerkennung	18
2.2.2 Verkehrsschilderkennung	24
2.2.3 Optische Abstandsmessung.....	28
2.3 Entwicklungswerzeuge.....	29
2.3.1 Modellierung.....	30
2.3.2 Integrierte Softwareentwicklungsumgebung.....	31
2.3.3 Versionsverwaltung	31
2.3.4 Softwarebibliothek.....	32
2.3.5 Evaluierung und Simulation.....	32
3. KONZEPT UND DESIGN.....	34
3.1 Systemdesign.....	34
3.2 Hardwarekomponenten	35
3.3 Systemzustände.....	38
3.4 Regelungsentwurf.....	41
3.5 Datendesign.....	43
3.5.1 Systemzustandsdaten	44
3.5.2 Bilddaten	44
3.5.3 Fahrbahnerkennungsdaten	45
3.5.4 Verkehrsschilderkennungsdaten	46
3.5.5 Hinderniserkennungsdaten	47
3.5.6 Trajektoriendaten	47
3.5.7 Fahrzeugdaten	48

3.5.8	Daten für die graphische Benutzerschnittstelle	49
3.5.9	Konfigurationsdaten	49
3.6	Betriebssystem und Installation.....	50
3.7	Monitoring.....	51
4.	HARDWAREREALISIERUNG	53
4.1	Chassis und Gehäuse.....	53
4.2	Kamerastativ.....	55
4.3	Abstandsmessung	56
4.4	Motorsteuerung.....	59
5.	SOFTWAREREALISIERUNG.....	62
5.1	Initialisierung.....	63
5.2	Konfigurationsmodule	64
5.2.1	Intrinsische Kamerawerte	65
5.2.2	Extrinsische Kamerawerte.....	68
5.3	Wahrnehmungsmodule	72
5.3.1	Bilderfassungsmodul	72
5.3.2	Fahrbahnerkennungsmodul.....	73
5.3.3	Stoppschilderkennungsmodul	77
5.3.4	Hinderniserkennungsmodul.....	78
5.4	Fahrtplanungsmodul	80
5.5	Fahrzeugsteuerungsmodul	82
5.6	Fernsteuerungsmodul	86
5.7	Benutzerschnittstellenmodul.....	89
5.8	Betriebsmodi	90
5.8.1	Standby-Modus	90
5.8.2	Autonomer Fahrmodus.....	92
5.8.3	Entwicklungsmodus.....	95
5.8.4	Fernsteuerungsmodus	98
5.8.5	Konfigurationsmodus.....	100
5.8.6	Kalibrationsmodi.....	102
5.8.7	Informationsmodus.....	104
5.8.8	Fehlermodus	106
5.8.9	Beendigungsmodus.....	106
6.	ERGEBNISSE	107
6.1	Teststrecke	107
6.2	Autonomes Fahren	108
6.2.1	Geradeausfahrt	108
6.2.2	Kurvenfahrt.....	109

6.2.3	Stoppschilderkennung.....	111
6.2.4	Kollisionsvermeidung	111
6.3	Systemleistung.....	112
6.4	Safety	115
6.5	Grenzen und Ausblick	116
7.	ZUSAMMENFASSUNG	120
ANHANG	124
ABBILDUNGSVERZEICHNIS.....		264
TABELLENVERZEICHNIS		267
LITERATURVERZEICHNIS.....		268

1. EINLEITUNG

Aktuelle Errungenschaften in der Automobilindustrie zeigen zwei technologische Trends. Zum einen werden immer mehr Assistenzsysteme wie Spurhalteassistenz, automatische Notbremsassistenz oder Einparkautomatik in Fahrzeuge eingebaut. Diese tragen zur Sicherheit im Straßenverkehr bei und erhöhen den Komfort. Zum anderen ermöglichen neue drahtlose Übertragungstechnologien die Kommunikation zwischen dem Automobil und dem Internet, um Stau-, Wetter- und weitere Routendaten auszutauschen. Damit können Fahrer und Fahrzeug früher auf kritische Verkehrssituationen reagieren. Die Weiterentwicklung dieser Assistenz- und Kommunikationssysteme zum vollständig autonomen Fahren ist das Ziel vieler Fahrzeughersteller, wie Demonstrationen von *Daimler*, *Volkswagen*, *Audi*, *BMW*, *Volvo*, *Tesla*, aber auch Informationstechnologiefirmen wie *Google* oder *Uber* zeigen. (vgl. [D18], [VW18a], [VW18b], [A18], [V18], [T18a], [W18a], [U18])

Die *Society of Automotive Engineers* (SAE) definiert sechs Stufen der Autonomie, an denen sich sowohl Gesetzgeber als auch Technologiehersteller weltweit orientieren. Die erste Stufe mit der Bezeichnung *Level 0* ist so definiert, dass keine Automation stattfindet. Der Fahrer ist für das Lenken, das Beschleunigen, das Bremsen, das Navigieren und für die Beobachtung der Umwelt zuständig. Dieser wählt auch den geeigneten Fahrbetrieb aus, entscheidet also, wo das Ziel ist, welche der verfügbaren Routen gewählt wird, wie schnell gefahren wird oder wann eingeparkt werden kann. Mit zunehmender Automationsstufe werden diese Aufgaben immer mehr vom Fahrzeug übernommen. Spurhaltesysteme oder Geschwindigkeitsregelanlagen ermöglichen demnach ein assistiertes Fahren auf dem *Level 1*. Vollautonom, was dem *Level 5* entspricht, fährt ein Fahrzeug dann, wenn kein menschlicher Fahrer mehr für die Fahrt benötigt wird. Der Autonomiegrad derzeitiger Entwicklungen befindet sich auf den Stufen 3 bzw. 4. Diese Fahrzeuge können in bestimmten Umständen, zum Beispiel auf Autobahnen oder im Stau selbstständig lenken, beschleunigen, bremsen und navigieren. Der Fahrer muss in der dritten Stufe das System und die Umgebung überwachen und bestimmen, welcher Fahrmodus geeignet ist. Eine detailliertere Beschreibung der Automationsstufen findet sich im SAE J3016 Standard. (vgl. [SAE18], [BAS12])

Diese Stufen umzusetzen, stellt keine triviale Herausforderung an das selbstständig fahrende Automobil dar. Die Fahrbahn, Verkehrsschilder, Hindernisse und andere Verkehrsteilnehmer müssen wahrgenommen und erkannt werden. Das Automobil muss entsprechend der Umweltsituation und der gesetzlichen Vorgaben korrekt und rechtzeitig agieren. Schließlich ist es ein wichtiges Ziel der Erforschung des autonomen Fahrens, die Sicherheit im Straßenverkehr zu erhöhen.

Durch die Vielzahl an Konzepten und Modellen mit verschiedenen Sensoren, Aktoren und entsprechenden Softwareroutinen ergeben sich viele Möglichkeiten, ein selbstständig fahrendes Auto zu entwickeln. Eine modulare und kostengünstige Forschungs- und Entwicklungsumgebung kann die Entwicklung eines solchen Fahrzeugs vereinfachen.

Im ersten Kapitel dieser Arbeit wird eine Einleitung in die Thematik gegeben. Der Stand der Technik, die Forschungsfrage und die Methodik werden hier besprochen. Ein Überblick

über verwandte Arbeiten aus dem Forschungsbereich zum autonomen Fahren und zu Fahrerassistenzsystemen ist ebenfalls enthalten. Das zweite Kapitel enthält eine Aufarbeitung der wichtigsten Grundlagen zur digitalen Bildverarbeitung sowie zur Objekt- und Fahrbahnerkennung, die zum autonomen Fahren benötigt werden. Auch werden derzeit verfügbare quelloffene Software und geeignete kostengünstige Hardware zur Erstellung eines autonom fahrenden Prototypenmodells verglichen. Ein konkretes Konzept für eine Entwicklungsumgebung wird im dritten Kapitel vorgeschlagen. Die dafür ausgewählten Hard- und Softwarekomponenten werden beschrieben und für den Entwurf eines Prototyps eingesetzt. Kapitel vier geht auf die Details der Realisierung eines autonom fahrenden Modelfahrzeugs zur Demonstration der Entwicklungsumgebung ein, die im Zuge dieser Masterarbeit durchgeführt wurde. Im fünften Kapitel wird die Softwarerealisierung der Entwicklungsumgebung thematisiert. Hier werden die eingesetzten Module und Modi sowie die Übergänge zwischen diesen und der Ablauf des Programms erklärt. Auch auf die Umsetzung der graphischen Benutzeroberfläche wird hier eingegangen. In Kapitel sechs werden die Ergebnisse des Projekts präsentiert. Auch werden hier die Erkenntnisse aus den Ergebnissen diskutiert und ein Ausblick für den Einsatz der Entwicklungsumgebung und deren Weiterentwicklung gegeben. Im letzten Kapitel werden schließlich, um einen Überblick über die Arbeit zu geben, die wichtigsten Aspekte, Ergebnisse und Erkenntnisse zusammengefasst.

1.1 Stand der Technik

Die Demonstrationsfahrzeuge der eingangs genannten Automobilhersteller zeigen, dass das autonome Fahren in bestimmten Situationen, wie auf Autobahnen und im urbanen Verkehr, mit dem aktuellen Stand der Technik bereits möglich ist, aber nicht fehlerfrei funktioniert. Noch befinden sich alle Systeme in der Entwicklung und benötigen die ständige Überwachung eines Menschen. Keines der vorgestellten Fahrzeuge bietet eine quelloffene oder erwerbbare Entwicklungsplattform an. Im Folgenden werden einige Entwicklungswerkzeuge für das autonome oder assistierte Fahren vorgestellt, die bereits am Markt erhältlich sind. Sie werden auch von einigen Fahrzeugherstellern in ihren Demonstrationsfahrzeugen eingesetzt.

Der finnische Softwarehersteller *Elektrobit* (EB) bietet mit *EB Assist ADTF (automotive data and time-triggered framework)* eine Softwareentwicklungsplattform für *Advanced Driver Assistance Systems* (ADAS) an, die sich bereits als Industriestandard durchgesetzt hat (vgl. [EB18]). Es verfügt über Schnittstellen zur Verarbeitung von Sensordaten und zu einem Online-Kartensystem, um den globalen Standpunkt, das Ziel und den Weg des Fahrzeugs dahin zu berechnen. Auch Funktionen zur Fahrzeug-zu-Fahrzeug- (*vehicle to vehicle*, V2V) bzw. Fahrzeug-zu-Infrastruktur-Kommunikation (*vehicle to infrastructure*, V2I) werden zur Verfügung gestellt. Die Fahrzeugsteuerung wird über Automobilstandards wie AUTOSAR oder CAN durchgeführt. Im *Audi Autonomous Driving Cup* wird *EB Assist ADTF* zur Entwicklung von Software für autonome Modelfahrzeuge im Maßstab 1:8 eingesetzt. Um mit *EB Assist ADTF* erstellte Software auszuführen, wird allerdings eine Lizenz benötigt.

Für diese Masterarbeit werden jedoch quelloffene Softwarewerkzeuge gesucht, welche den Zugang zur Entwicklung eines autonomen Fahrzeugs erleichtern. (vgl. [A18a], [EB18])

Das *Blackberry*-Tochterunternehmen QNX bietet Betriebssysteme und Entwicklungswerkzeuge für ADAS, Fahrerinformations- und Unterhaltungssysteme an. Damit können Anwendungen für die Fahrzeugkommunikation, sowohl V2V, als auch V2X, *Computer Vision* (CV), digitale Benutzerschnittstellen und Navigationssysteme entwickelt werden. Die passende Hardware wird von Entwicklern, wie zum Beispiel *Renesas* mit dem Modell *R-Car V3H* geliefert. Auch andere Hersteller erweitern ihre Fahrzeuginformations- und Unterhaltungssysteme um Funktionen zum assistierten oder autonomen Fahren. Diese Systeme sind aber ebenfalls geschlossen und benötigen eine Lizenz zum Gebrauch. (vgl. [BB18])

Der Fahrzeugherrsteller *Audi* entwickelt das *Zentrale Fahrzeugassistenzsystem* (zFAS) für das teil- und vollautonome Fahren. Dabei handelt es sich um eine Hardwareplatine, welche alle Assistenzsysteme zentral ausführt. An diese Plattform werden Sensoren wie Kameras, Ultraschall, RADAR und LIDAR angeschlossen. Die Kommunikation zwischen dem zFAS und der Motorsteuerung wird über standardisierte Anschlüsse durchgeführt. Das zFAS ist mit Prozessoren der Hersteller *Altera*, *Infinion*, *Mobileye* und *NVIDIA* ausgestattet. *Audi* integrierte diese Plattform in mehreren Demonstrationsfahrzeugen, wie zum Beispiel 2017 im *Audi A7* und im aktuellen Serienfahrzeug *A8*. Sie steht jedoch weder zum Kauf, noch zur Weiterentwicklung durch Dritte zur Verfügung. (vgl. [VW18b])

NVIDIA bietet die *Drive*-Familie an. Dabei handelt es sich um Platinen für den Automobilbereich, die für Anwendungen mit künstlicher Intelligenz (KI) und artifiziellen neuronalen Netzen (ANN) ausgelegt sind. Auch an diesen Platinen können die genannten Sensoren und das Motorsteuergerät angeschlossen werden. Zu diesen Hardwarelösungen werden auch entsprechende Softwareentwicklungspakete, wie zum Beispiel *NVIDIA DriveWorks*, angeboten. Diese Entwicklungsumgebung ist aber ebenfalls nicht quelloffen. (vgl. [N18a], [N18b])

Hardwarekomponenten für ADAS entwickelt auch *Texas Instruments* mit *Jacinto TDAx System-on-Chip* (SoC). Die aktuell angebotenen Chips bestehen unter anderem aus einer *ARM Cortex-A15 Central Processing Unit* (CPU) und einem C66x digitalen Signalprozessor (DSP). Damit können Sensorsignale und digitale Bilddaten verarbeitet werden. *Texas Instruments* bietet auch Evaluierungskits an, welche allerdings keine vollständigen autonomen Systeme sind. (vgl. [TI18a], [TI18b])

Neben der Technik zum autonomen Fahren sei noch erwähnt, dass auch die rechtlichen Rahmenbedingungen derzeit erst am Entstehen sind. Immer mehr Staaten genehmigen den Betrieb von Testfahrzeugen im öffentlichen Raum oder auf bestimmten Teilstrecken. In keinem Fall jedoch darf derzeit ein System ohne menschliche Überwachung fahren. Diese überwachende Person trägt schließlich die Verantwortung und muss das System im Notfall anhalten können.

1.2 Motivation

Wie im Abschnitt Stand der Technik gezeigt wurde, gibt es verschiedene Entwicklungswerkzeuge für ADAS Anwendungen. Die Prototypenfahrzeuge der Automobilhersteller demonstrieren, dass mit diesen Werkzeugen das Bauen autonomer Fahrzeuge sowie das Entwickeln von Anwendungen dafür möglich ist.

Allerdings werden weder die Fahrzeuge, noch die Anwendungen oder die Ergebnisse zur Weiterentwicklung für Dritte bereitgestellt. Zwar sind einzelne Werkzeuge erwerbar, doch stellen diese keine vollständige Entwicklungsumgebung dar und sind auch nicht als solche konzipiert. Außerdem sind lediglich elektronische Hardwarekomponenten verfügbar, jedoch ist ein Entwicklungsfahrzeug für außenstehende Entwickler nicht zugänglich.

Als Entwicklungsumgebung wird in dieser Arbeit eine Auswahl an Software- und Hardwarewerkzeugen definiert, die es ermöglichen, ein vollständiges autonomes Fahrzeug zu entwickeln, oder aber auch nur einzelne Anwendungen dafür zu erstellen und zu evaluieren. Teil dieses Werkzeugsets ist auch ein Modelfahrzeug, welches selbst ein Werkzeug ist, als Träger für die übrigen Werkzeuge fungiert und in die Entwicklung miteinbezogen wird. Dieser portable Prototyp ermöglicht zusätzlich den Transport und den Einsatz der Umgebung in verschiedenen Testfeldern sowohl innerhalb von Gebäuden als auch außerhalb. Außerdem ist es von Vorteil, wenn die Umgebung skalierbar ist, um sie später auch auf anderen Fahrzeugtypen einzusetzen. Die einzelnen Soft- und Hardwarekomponenten sollen dabei modular integrierbar sein, damit sie ausgetauscht und erweitert werden können.

Da eine solche Entwicklungsumgebung bisher nicht frei verfügbar, aber für die Weiterentwicklung des autonomen Fahrens wichtig und sinnvoll ist, soll im Rahmen dieser Arbeit eine solche erstellt werden. Damit soll der Zugang zu quelloffenen und kostengünstigen Werkzeugen ermöglicht werden, um die Erforschung und Erstellung neuer Anwendungen zu vereinfachen.

1.3 Aufgabenstellung

In dieser Arbeit wird eine Entwicklungsumgebung für das autonome Fahren erstellt. Anhand des Stands der Technik und aktueller Veröffentlichungen in dem Bereich werden geeignete Werkzeuge, Hardwarekomponenten und Softwaremethoden vorgeschlagen, um ein selbstfahrendes Modellautomobil zu entwickeln, das beliebig erweiterbar ist. Mithilfe eines kostengünstigen Embedded-Systems, sollen mittels bildgebender Sensorik erfasste Daten so ausgewertet werden, dass das Fahrzeug die Fahrroute selbstständig plant und befährt.

Die zu verfolgende Fahrbahn verfügt über Begrenzungsmarkierungen, die vom System wahrgenommen werden soll. Die Detektion von Hindernissen auf der Fahrbahn ist ebenfalls Teil der Aufgabe. Mithilfe dieser Entwicklungsumgebung soll unidirektionales Fahren mit konstanter Geschwindigkeit umgesetzt werden, wobei der maximale Lenkeinschlag für Richtungsänderungen beschränkt bleiben soll.

Die erfassten Sensordaten sollen zum Monitoring bzw. zur Parametrisierung drahtlos auf einen Computer übertragen werden. Das Modellfahrzeug soll im Testlauf nach einem Startsignal den Fahrbahnmarkierungen folgen und beim Erkennen von Hindernissen anhalten. Wird das Hindernis entfernt, muss der Testlauf neu gestartet werden.

1.4 Methoden

Um eine geeignete Entwicklungsumgebung zu erstellen, werden zunächst notwendige Grundlagen für das autonome Fahren anhand bestehender Projekte und Veröffentlichungen erarbeitet. Auch werden Verfahren zur Fahrbahn- und Objekterkennung erörtert. Derzeit verfügbare offene Softwarewerkzeuge zur Lösung des Erkennungsproblems und des Maschinenlernens werden analysiert. Anschließend wird eine Auswahl an geeigneten Verfahren für die Realisierung einer Steuerung mit einem Embedded-Board vorgestellt. Mit diesen Ergebnissen wird anschließend ein autonomer Prototyp entworfen, wobei ein *Raspberry Pi Model B 3* als Steuerungseinheit dient.

Dieses Modellfahrzeug wird im nächsten Schritt realisiert und wird mittels einer konventionellen USB-Webkamera und entsprechender Algorithmen die Fahrbahn und andere Objekte erfassen. Durch Softwareroutinen zur digitalen Bildanalyse wird die Fahrspur erkannt und verfolgt sowie das Modellfahrzeug gesteuert. Für die Lenkung werden ein Servomotor und für den Antrieb ein Bürstenmotor verwendet. Zur Protokollierung der Messergebnisse, zur Parametrisierung sowie zur Fernsteuerung des Fahrzeugs wird ein weiterer Computer genutzt. Dieser erhält Echtzeit-Videodaten und Telemetrie-Daten über eine WLAN-Schnittstelle.

Die Funktionen der Realisierung werden auf einer Teststrecke evaluiert und die Ergebnisse präsentiert. Schließlich sollen aus den Testergebnissen Schlussfolgerungen für die Weiterentwicklung des Prototyps und mögliche Folgeprojekte gezogen werden.

1.5 Verwandte Arbeiten

Im Folgenden wird Bezug auf verwandte wissenschaftliche Arbeiten und Projekte genommen, die sich mit Entwicklungsumgebungen für das autonome Fahren beschäftigen.

Mit der Ausschreibung der *Defense Advanced Research Projects Agency* (DARPA) 2003 wurde ein Wettbewerb zur Erstellung eines autonom fahrenden Automobils gegründet, bei dem viele Forschungseinrichtungen, Universitäten und Firmen teilgenommen haben. Im Rahmen der *Urban Challenge* 2007 sollten diese Fahrzeuge auch im urbanen Umfeld zum Ziel gelangen und bestimmte Aufgaben lösen. Die Erkenntnisse aus diesen Projekten sind in zahlreichen Publikationen ersichtlich und stellen wichtige Grundlagen für die Entwicklung des autonomen Fahrens dar. Damit sind sie auch für die Erstellung einer Entwicklungsumgebung wesentlich. Die Erkenntnisse über das Systemdesign, sowie die Methoden zur Fahrbahn- und Objekterkennung in den DARPA-Projekten werden auch in dieser Arbeit eingesetzt. (vgl. [BBR07], [BR12])

Das Gewinnerprojekt der *Urban Challenge* aus dem Jahr 2007 ist *Boss*, das unter der Leitung der *Carnegie Mellon Universität* vom *Tartan Racing Team* entwickelt wurde. Dieses autonome Fahrzeug ist mit GPS, Laser-, RADAR- und Kameratasensoren ausgestattet, um sich zu orientieren, Hindernisse zu erkennen und sich im Straßenverkehr richtig zu verhalten. Die Architektur besteht aus drei logischen Schichten: eine zur Aufgabenplanung, eine zur Verhaltensplanung und eine zur Fahrplanung. Für eine Entwicklungsumgebung ist dieses Schichtenmodell von Interesse, um die Gesamtaufgabe des autonomen Fahrens in einzelne Aufgabenschritte zu unterteilen. Dadurch können auch notwendige Soft- und Hardwarekomponenten zur Erfüllung derselben gewählt werden. (vgl. [UAB08], [WSK13])

Ein weiteres autonom fahrendes Automobil, das im Rahmen der DARPA *Urban Challenge* präsentiert wurde, ist *Spirit of Berlin*. Dieses Fahrzeug verfügt zwar über die gleichen Sensorarten wie *Boss*, doch wurde für dieses Projekt zusätzlich eine eigene Simulationssoftware entwickelt. Diese ermöglicht es, neue Systemkomponenten zuerst virtuell zu testen und dann in das Projekt zu integrieren (vgl. [RR07]). Gietelink et al. haben eine *Vehicle Hardware-In-The-Loop-Simulation* (VEHIL) für ADAS erstellt. Ziel dieser Arbeit ist, es eine günstige, sichere und einfach zu handhabende Entwicklungs- und Validierungsmethode anzubieten. Der Vorteil einer solchen Simulationssoftware ist, Kosten für teure reale Hardware zu sparen (vgl. [GPD06]). Eine weitere Evaluationsmethode stellen Zhou et al. vor. Sie simulieren mit *Matlab* und *Simulink* Funktionen für das autonome Fahren. Systemfunktionen werden als Closed-Loop-Regelungssysteme realisiert und erstellen damit einfache Simulationsmodelle. Dies kann zum schnellen Entwurf eines autonomen Systems beitragen, stellt aber keine vollständige Entwicklungsumgebung dar (vgl. [ZCX09]). Das Konzept der Simulation von Algorithmen zur Evaluation wird auch in dieser Masterarbeit in vereinfachter Form angewandt. Das Modelfahrzeug der vorliegenden Arbeit wird ebenfalls nach dem VEHIL-Entwicklungsprinzips eingesetzt.

DESERVE ist eine von der Europäischen Kommission geförderte Entwicklungsplattform für das sichere und effiziente Fahren. Im Rahmen eines dreijährigen Projekts sind neue Standards und Werkzeuge für Fahrerassistenzsysteme erforscht worden. Mit ihrer Hilfe sollen schließlich Soft- und Hardware für diese Systeme kostengünstiger und schneller realisiert werden können. In den Ergebnissen des Projekts werden ähnlich wie beim *Boss*-Projekt drei logische Schichten, nämlich die Perzeption, die Applikation und die Intervention, vorgeschlagen. Für die Softwareentwicklung werden bereits in der Industrie eingesetzte Programme wie *Pro-SiVIC*, *EB Assist ADTF*, *Matlab* und *Simulink* vorgeschlagen. Diese Werkzeuge werden auch im Rahmen dieser Arbeit betrachtet und zum Teil eingesetzt. (vgl. [BK15], [KPK14], [R12], [HGB10])

Eine mobile Plattform zur Positionserfassung über GPS wird von Jaskot und Babiarz von der *Silesian University of Technology* vorgestellt. Dabei handelt es sich um ein Modelfahrzeug, das mit einem Kommunikations- und Steuerungsmodul sowie mit GPS ausgestattet ist. Dieses Projekt dient zur Vorbereitung für die Entwicklung eines Autopiloten. Für das autonome Fahren ist GPS alleine keine ausreichende Methodik zur Positionserfassung, da der Empfang nur bei Sichtkontakt, daher nur im Freien möglich ist. Wie in Kapitel 2.1.1 genauer erläutert wird, bedarf es zusätzlicher Sensorik zur Erfassung der Umwelt in Echtzeit. (vgl. [JB10], [JB11])

Pannu et al. entwerfen und implementieren in einer Publikation einen monokularen, autonom fahrenden Prototyp unter Verwendung eines *Raspberry Pi* Embedded-Boards. Mit einer *Raspberry Pi*-Kamera, Ultraschalsensoren und der *OpenCV*-Bibliothek gelingt die Fahrbahn- und Hinderniserkennung. Damit ist der Nachweis gegeben, dass die Rechenleistung des *Raspberry Pi* der ersten Generation für die Umsetzung eines simplen, autonom fahrenden Modellautos ausreichend ist. Dieser Ansatz ist für eine kostengünstige mobile Entwicklungsumgebung von Interesse. Im Gegensatz zu dieser Publikation wird in der vorliegenden Masterarbeit die leistungsfähigere Nachfolgegeneration des *Raspberry Pi* eingesetzt, um auch eine Objekterkennung zu realisieren. (vgl. [PAG15])

Ein Studententeam der *Universität Göteborg* hat ein autonomes Modellfahrzeug entwickelt, das ein *Android*-Smartphone und dessen integrierte Kamera für die Fahrbahnerkennung und Verhaltensentscheidung einsetzt. Das Fahrzeug selbst wird über einen *Arduino*-Mikrokontroller gesteuert. An diesen sind, neben der Motorsteuerung, auch sechs Ultraschallsensoren zur Abstandsmessung angeschlossen. Die Kommunikation zwischen dem Smartphone und dem Mikrokontroller findet drahtlos über Bluetooth statt. Das Projektfahrzeug kann, neben der Fahrbahn, auch Hindernisse erkennen, ihnen ausweichen und automatisch einparken. Die Ergebnisse und Quellcodes des Projekts sind online frei zugänglich. Da die am Smartphone integrierte Kamera verwendet wird, ist die Möglichkeit zur robusten Montage am Modellfahrzeug allerdings eingeschränkt. (vgl. [PCH18])

Wang realisiert mit dem *Raspberry Pi* 2 und einem Modellfahrzeug ebenfalls einen autonomen Prototyp. In diesem Projekt erfasst eine *Raspberry Pi*-Kamera die Bilddaten und überträgt sie via WLAN auf einen PC. Das Embedded-Board misst mithilfe eines Ultraschallsensors die Distanz vor dem Fahrzeug zum nächsten Hindernis und überträgt diese ebenfalls auf den PC. Über einen zusätzlichen *Arduino Uno* wird das Fahrzeug gesteuert. Die Auswertung der Bilddaten erfolgt auf dem Empfangs-PC. Dieser schickt dann entsprechende Lenkbefehle zurück an das Embedded-Board. Dieser Ansatz ist allerdings für das autonome Fahren nicht interessant, da die Sensordatenverarbeitung von einer funktionierenden drahtlosen Datenverbindung abhängig ist. Diese kann jedoch nicht immer garantiert werden. (vgl. [W18c])

An der *Georgia Tech* wird die *AutoRally*-Plattform entwickelt. Dabei handelt es sich um ein Modellfahrzeug im Maßstab 1:5, das mit einer Geschwindigkeit bis zu 27 m/s einem Rundkurs folgt. Dieser besteht aus einer Randmarkierung und einer unbefestigten Fahrbahn. Die Fahrtrichtung wird anhand der Aufnahmen zweier Kameras sowie mittels GPS bestimmt. Eine Bauanleitung und der entwickelte Code ist quellloffen auf der Webseite des Projektes verfügbar. Ziel dieses Projektes ist es, das autonome Fahren bei hohen Geschwindigkeiten zu erforschen. Für eine Entwicklungsplattform ist die Systemarchitektur daher interessant. (vgl. [GT18])

Für die Forschung und Lehre am *Massachusetts Institute of Technology* (MIT) wurde 2015 das Projekt *Rapid Autonomous Complex-Environment Competing Ackermann-steering Robot* (RACECAR) initiiert. Es ist als Open-Source-Plattform ausgelegt und wird im Rahmen einer Vorlesung eingesetzt. Ziel ist es, eine Software zu entwickeln, die das RACECAR eine bestimmte Route möglichst schnell autonom befahren lässt. Als Steuerungsplattform wird ein *NVIDIA JETSON TK1*-Embedded-Board eingesetzt, das mithilfe eines LIDAR-Sensors die Umgebung wahrnimmt. Eine Kamera ist zwar ebenfalls

inkludiert, dient allerdings nicht der Fahrbahnerkennung. Damit unterscheidet sich der Ansatz dieses Projekts zur vorliegenden Arbeit. (vgl. [MIT18])

Die *Technische Universität Braunschweig* in Deutschland veranstaltet seit 2008 den *Carolo-Cup*, einen Hochschulwettbewerb zur Realisierung eines autonomen Modellfahrzeugs, das unterschiedliche Aufgaben zu lösen hat. Dazu gehören die Fahrbahnverfolgung, die Hinderniserkennung und das Einparken. Im Rahmen dieses *Cups* entstehen zahlreiche Projekte und daraus einige Bachelor- und Masterarbeiten, wie zum Beispiel [J08], [J13] oder [N11]. Zwar lösen sie zum großen Teil die gestellten Herausforderungen, stellen aber an sich keine Entwicklungsumgebungen dar. Die genannten Arbeiten umfassen verschiedene Fahrbahnerkennungsverfahren, beschreiben aber nicht das gesamte System. Die Ergebnisse dieser Projekte bieten dennoch hilfreiche Informationen zur Erreichung des Ziels der vorliegenden Arbeit. Auch die Anforderungen des *Carolo-Cups* sind von Interesse, weil sie vereinfachte Anforderungen für ein autonomes Modellfahrzeug darstellen. Damit können für die vorliegende Masterarbeit Teilziele auf den Weg der Entwicklung formuliert werden. (vgl. [TUB18], [J08], [J13], [N11])

2. GRUNDLAGEN

Um eine Entwicklungsumgebung für das autonome Fahren zu entwerfen, müssen zunächst die grundlegenden Anforderungen und Konzepte dazu betrachtet werden. Die bestehenden Projekte verdeutlichen, dass es vielfältige Lösungswege gibt, um ein autonomes Fahrzeug zu entwickeln. Die zahlreichen Veröffentlichungen zu den Themen Fahrbahn- und Objekterkennung, Abstandsmessung und Regelungstechnische Mechanismen für mobile Roboter schaffen sehr viele Möglichkeiten zur Implementierung geeigneter Algorithmen und Systeme. In dieser Arbeit wird lediglich ein Einblick in die Vielfalt gegeben und eine Auswahl dieser Verfahren mit dem Ziel betrachtet, eine kostengünstige mobile Entwicklungsplattform zu entwerfen.

Die Hauptaufgabe eines autonom fahrenden Automobils ist es, Passagiere oder Güter sicher von einem Startpunkt über verfügbare Verkehrsstraßen zu einem gewünschten Ziel zu bringen. Das automatische Verfolgen dieser Routen mithilfe eines globalen Positionierungssystems (GPS) und bestehender Kartendaten ist dafür nicht ausreichend. Ungenaue Positionsdaten, ungenügender Satellitenempfang und das Fehlen wichtiger Echtzeitinformationen über die unmittelbare Umgebung behindern die Positionserfassung des Fahrzeugs. Sich dynamisch verändernde Hindernisse, wie Fußgänger, Fahrzeuge, aber auch neue Gebäude, Straßen oder Baustellen müssen ebenfalls vom System beachtet werden. Neuartige Online-Systeme können zwar aktuelle Echtzeitverkehrsinformationen, wie Staus, Baustellen oder Unfälle aufnehmen und anzeigen, benötigen aber dazu eine ständige Internetverbindung. Die größten Herausforderungen stellen unvorhersehbare und sich plötzlich verändernde Verkehrssituationen dar. Deshalb ist die Wahrnehmung und Verarbeitung der unmittelbaren Umgebung um das System herum unerlässlich. Erst auf Basis dieser Informationen können geeignete Verhaltensentscheidungen vom Fahrzeug getroffen werden.

2.1 Systemarchitektur

Zu Beginn ist ein Blick auf bestehende Systemarchitekturen hilfreich, um wichtige Bausteine für das autonome Fahren zu identifizieren. In den Ergebnissen der DARPA *Urban Challenge*-Projekte, die zwar unabhängig voneinander entwickelt wurden, sind jedoch einander durchaus ähnliche Architekturen zu finden. Die Umsetzungen basieren auf vier logische Schichten, nämlich der Erfassung der Umwelt mittels geeigneter Sensorik, der Verarbeitung dieser Informationen, der Planung der Fahrtroute, auch Trajektorie genannt, und, abhängig davon, der Steuerung des Fahrzeugs. Je nach Projekt ist auch eine Schicht zur Aufgabenplanung und eine Schnittstelle zum Benutzer vorgesehen. Auch Siegwart et al. präsentieren eine allgemeine Einführung zu autonomen mobilen Robotern, in der diese Schichten erwähnt werden. Ein autonomes Fahrzeug ist im Grunde eine Spezialisierung davon. (vgl. [BFK07], [BR12], [CEH10], [MBB08], [UAB08], [BT15], [SNS11])

Abbildung 1 stellt diese Schichten und die Kommunikationsrichtungen untereinander graphisch dar. Die Messdaten von verschiedenen Sensortypen müssen zunächst

aufgenommen und verarbeitet werden. Sie müssen in geeigneter Weise korreliert werden, sodass ein Systemzustand mit Lage, Position und Umgebungsinformationen eruiert werden kann. Basierend auf diesen Daten entscheidet die Planung über das geeignete Verhalten des Fahrzeugs und über die Fahrtroute. Schließlich wird die tatsächliche Fahrzeugsteuerung anhand der Trajektorie durchgeführt. Im Folgenden werden die einzelnen Schichten dieser Systemarchitektur genauer betrachtet. (vgl. [WSK13], [SNS11])

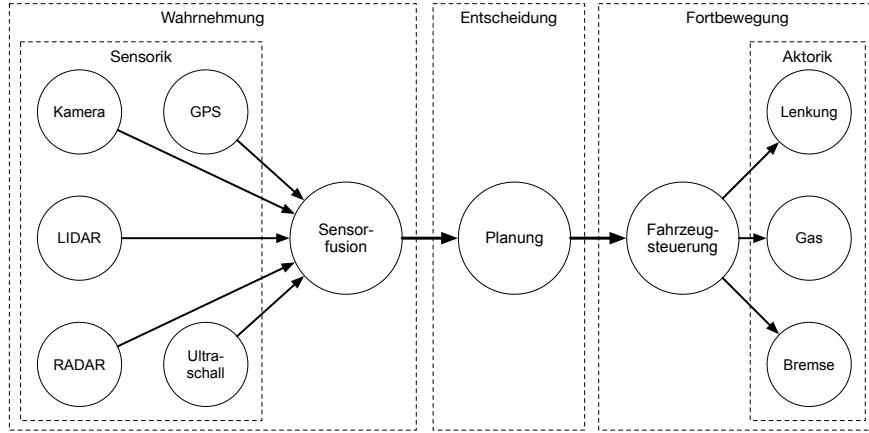


Abbildung 1: Systemarchitektur für das autonome Fahren (vgl. [WSK13], [SNS11])

2.1.1 Sensorik

Wie eingangs erwähnt, reicht die Navigation alleine mit GPS-Informationen, wie sie heutzutage zu Verfügung stehen, nicht aus, um autonom zu fahren. Sich ständig und plötzlich verändernde Hindernisse im Verkehr müssen zur sicheren Fortbewegung ebenfalls beachtet werden. Zur Wahrnehmung der Umgebung werden deshalb Umweltdaten von Sensoren akkumuliert. Daher müssen die Abstände, die Richtung, die Geschwindigkeit und die Positionen des Fahrzeugs relativ zur Umwelt und den Objekten darin gemessen werden. Diese bedürfen auch einer Identifikation, um beispielsweise die Fahrbahn, Verkehrsschilder oder andere Verkehrsteilnehmer voneinander zu unterscheiden, bzw. als solche zu erkennen. Die am häufigsten genutzten *exterozeptiven* Sensoren in den DARPA-Projekten sowie den Projekten der Automobilhersteller, sind digitale Kameras, RADAR-, LIDAR-, Ultraschall- und GPS-Sensoren. (vgl. [BFK07], [BR12], [CEH10], [MBB08], [UAB08])

Grund für den Einsatz mehrerer und verschiedener Sensoren ist, die Stärken und Schwächen dieser auszugleichen und somit ein zuverlässigeres Modell der Umwelt zu erstellen. Ein vollständiges und genaues Abbild der Umgebung ist allerdings nicht erfassbar. Die Schwächen aller Sensorarten sind Unsicherheiten, die durch Messfehler, physikalische Grenzen oder Rauschen entstehen. Die Bewältigung dieser Herausforderungen wird im nächsten Kapitel besprochen.

Eine der wichtigsten Informationen für das autonome Fahrzeug ist der Abstand zu Hindernissen in der Umgebung. Dafür werden RADAR-, LIDAR- oder Ultraschall-Sensoren

eingesetzt. Der Unterschied zwischen den Sensoren liegt in der Messmethode, der Genauigkeit und der Reichweite. (vgl. [E12] S. 34-43)

Ultraschall

Ultraschall wird im Nahbereich von bis zu 5 m eingesetzt, da in diesem Bereich eine zuverlässige Schallreflexion erwartet werden kann. Aufgrund der physikalischen Eigenschaften des Schalls wird der räumliche Ausstrahlungsbereich mit zunehmendem Abstand von der Quelle größer. Damit lassen sich unmittelbare Hindernisse erkennen, aber nicht die genaue Position im Ausstrahlungsbereich. (vgl. [SNS11] S. 126-129) In der schematischen Darstellung in Abbildung 2 ist die Funktionsweise des Sensors abgebildet. Über einen Sender wird ein Ultraschallsignal abgeschickt, das beim Auftreffen auf ein Hindernis ein Echo zurückwirft. Ein Empfänger am Sensor kann diese Reflexion aufnehmen. Daraus kann, aufgrund der bekannten Ausbreitungsgeschwindigkeit des Schalls im Raum, durch eine Zeitmessung auf eine Distanz zwischen dem Sensor und dem Hindernis rückgeschlossen werden. Die Schallgeschwindigkeit in der Luft bei einer Raumtemperatur von 20 °C entspricht etwa 343 m/s. Daher wird umgerechnet 1 m in 2,91 ms zurückgelegt. Zur Vereinfachung wird angenommen, dass bei einem normalen Luftdruck von 1 bar, die Geschwindigkeit nur von der Temperatur abhängig ist, was in der Praxis ausreicht. (vgl. [SNS11] S. 126-129, [CP01], [E12] S. 40-41)

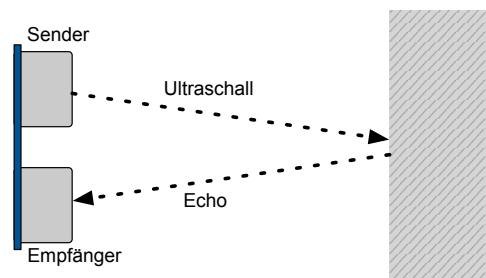


Abbildung 2: Funktionsweise eines Ultraschallsensors

RADAR und LIDAR

Für größere Distanzmessungen eignen sich RADAR und LIDAR. Hauptsächlich unterscheiden sich diese zwei Technologien durch das Messsignal. RADAR verwendet Funksignale im Gigahertzbereich, die entweder in einem kleinen Winkel und mit größerer Distanz, oder in einem größeren Winkel und mit kürzerer Distanz ausgestrahlt werden. Damit können verschiedene Bereiche um das Fahrzeug herum abgetastet werden. Der Vorteil im Vergleich zu Ultraschall ist, dass neben der Messung des Abstands zu einem Hindernis, auch die Position desselben und die relative Geschwindigkeit dazu erfasst werden kann (vgl. [SNS11] S. 140-141). RADAR wird bereits für ADAS in Serienfahrzeugen der meisten Automobilhersteller eingesetzt und tastet Distanzen von 50 m (Kurzstrecken-RADAR) bis 250 m (Langstrecken-RADAR) um das Fahrzeug ab. (vgl. [SNS11] S. 133-135, [E12] S. 36-38)

Genauere Messergebnisse liefert das LIDAR, welches Infrarotsignale einsetzt, die durch Laser erzeugt werden. Durch die feinere Abtastung der Umgebung ist allerdings auch die erzeugte Datenmenge höher, was wiederum mehr Ressourcen in der Verarbeitung benötigt. Die Reichweite von LIDAR ist etwas geringer als bei RADAR. Mit 3-D-LIDAR

können gleich mehrere Raumdimensionen abgetastet und so ein virtuelles 3-D-Bild der Umgebung erstellt werden. In der Praxis werden, zum Zeitpunkt der Erstellung dieser Arbeit, von mehreren Fahrzeugherstellern LIDAR-Sensoren für ADAS getestet, aber noch in keinem Serienfahrzeug eingesetzt. Sowohl RADAR- als auch LIDAR-Sensoren sind um ein Vielfaches teurer als Ultraschallsensoren. Die Entwicklungsplattform dieser Arbeit wird auf kurzen, zu einem 1:10 Modellfahrzeug maßstabgetreuen Teststrecken eingesetzt, die nur wenige Meter umfassen. Die sich ergebenden Distanzen können daher von Ultraschall-Sensorik ausreichend gut abgetastet werden. (vgl. [SNS11] S. 133-135, [E12] S. 38)

GPS

Wie für einen menschlichen Fahrer, ist es für das autonome Fahrzeug hilfreich, wenn es Informationen über die eigene globale Position hat. *Navigational Satellite Timing and Ranging Global Positioning System* (NAVSTAR GPS) ist mittlerweile in allen Applikationen Standard, in denen Positionierungsinformationen auf der Erde notwendig sind. Über vier Satelliten kann der Empfänger mittels Signallaufzeit seine Position auf etwa 10 m genau bestimmen und die eigene Geschwindigkeit berechnen. Für eine Positionserfassung mit einer Genauigkeit von unter 1 m kann differenzielles GPS (DGPS) angewendet werden. Dazu wird die Differenz der gemessenen Position und einer bekannten Position eines weiteren stationären GPS-Empfängers in der Nähe berechnet. Ein Nachteil von GPS ist, dass sich der Empfänger im Sichtkontakt zu den vier Sendern befinden muss. Damit werden in Wäldern oder Städten mit hohen Gebäuden unzuverlässige Messdaten erfasst. In geschlossenen Räumen fällt der Empfang gänzlich aus. Etwas genauere Positionierungssysteme, wie zum Beispiel *Galileo*, befinden sich derzeit noch in Entwicklung. (vgl. [SNS11] S. 123-124, [E12] S. 313-340)

In dieser Arbeit wird auf GPS verzichtet, da die globale Position für die Aufgabenstellung zunächst eine geringere Priorität hat. Das zu entwickelnde Modellfahrzeug wird zunächst hauptsächlich in Innenräumen eingesetzt werden. Für nächste Entwicklungsschritte außerhalb des Rahmens dieser Arbeit und mit der Anwendung im Freien wird ein solcher Sensortyp sinnvoll sein.

Bildsensoren

Neben der Distanzmessung zu den Objekten im Straßenverkehr ist auch deren Identifikation für das autonome Fahrzeug notwendig. Dazu eignen sich digitale Kameras, mit denen Helligkeits- und Farbinformationen erfasst werden können. Das Funktionsprinzip und die wesentlichen Bestandteile sind in Abbildung 3 dargestellt. Lichtstrahlen gelangen über ein Objektiv, das aus einer oder mehreren Linsen besteht, in das Innere des Kameragehäuses. Dort treffen sie auf den Bildsensor. Über den Blendenverschluss lässt sich die Belichtungszeit des Sensors steuern. Zusätzlich befindet sich in der Kamera eine Steuerungselektronik für den Sensor. (vgl. [E12] S. 1011-1020)

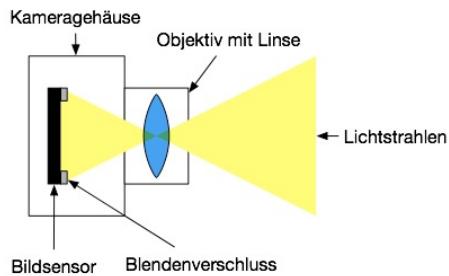


Abbildung 3: Modell einer digitalen Kamera

Complementary-Metal-Oxide-Semiconductor-Bildsensoren (CMOS) haben sich gegenüber alternativen Systemen, dank der einfacheren Produktion, des geringeren Energieverbrauchs und der niedrigeren Kosten, in den meisten Anwendungen am Markt durchgesetzt. Über eine Matrix aus kleinen Einzelsensoren werden über eine gewisse Zeitdauer Photonen aufgenommen. Diese Sensorelemente werden auch Pixel (engl. Kunstwort aus *picture element*) genannt. Um Farben aufnehmen zu können, werden physikalische Filter über jedes Pixel gelegt. Beim Single-Chip-CMOS wird das Bayer-Verfahren mit einer 2x2-Farbreihenfolge eingesetzt, wie in Abbildung 4 dargestellt ist. Zwei Pixel nehmen die Farbintensität für rot bzw. blau auf, zwei weitere die für grün. Damit reduziert sich allerdings die Auflösung der Aufnahme im Vergleich zur Sensorgröße. Eine Alternative ist es drei separate Pixelmatrizen für jede Farbe zu nutzen. Durch additives Mischen der Intensitätsbilder jedes Kanals, ergibt sich ein Farbbild. Dies führt allerdings zu einem teureren Sensor. (vgl. [SNS11] S.144-147, [E12] S. 1011-1020)



Abbildung 4: RGB-Single-Chip mit Bayers Pixelanordnung (vgl. [SNS11] S. 147)

Handelsübliche Foto- oder Webkameras verfügen über eine automatische Blende- und Belichtungsanpassung. Dazu benötigen sie aber oft einige Zeit, insbesondere bei starken und plötzlichen Wechseln der Lichtverhältnisse, wie das beim Herausfahren aus einem Tunnel der Fall ist. Die Notwendigkeit dieser Anpassung liegt in der physikalischen Beschränkung des dynamischen Bereichs (engl. *dynamic range*) des Sensors. Definiert wird dieser Bereich durch die maximale Photonenaufnahmekapazität pro Pixel im Verhältnis zum kleinsten messbaren Signal. Dieses untere Limit wird wiederum vom Bildrauschen bestimmt. Je größer ein Sensorpixel ist, desto lichtempfindlicher ist dieses. Die gesamte Sensorgröße muss aber, je nach Anwendungsbereich, beschränkt werden, um sie zum Beispiel in mobilen Applikationen zu installieren. Mit mehr als einer Kamera lassen sich auch stereoskopische Systeme implementieren und so Tiefeninformationen errechnen. Zwar sind Bildsensoren sehr kostengünstig, liefern aber, je nach Auflösung des Sensors, auch große Datenmengen. Um Informationen daraus zu erhalten, sind allerdings zusätzliche Verarbeitungsschritte notwendig, die in Kapitel 2.2 besprochen werden. Für diese Arbeit wird ebenfalls eine günstige handelsübliche Webcam ausgewählt. (vgl. [SNS11] S.144-147, [E12] S. 1011-1020)

Weitere Sensoren

Der interne Systemzustand ist für das autonome Fahren ebenfalls von Interesse. Wichtige Messgrößen sind die Geschwindigkeit, die Motordrehzahl und der Lenkwinkel, aber auch die Beschleunigung und die Lage. Die meisten *propriozeptiven* Sensoren gehören bereits zur Standardausstattung in Serienfahrzeugen. Erwähnt seien Inkrementalgeber, Beschleunigungssensoren, Gyroskope oder Magnetschalter. Damit sind Systeme wie ABS, ESP oder die Servolenkung ausgestattet. In dieser Arbeit werden zunächst keine Sensoren dieses Typs eingesetzt, da der Fokus der gesetzten Aufgabenstellung die Umwelterfassung ist. Für die Weiterentwicklung des in dieser Arbeit entwickelten Prototyps sollten sie allerdings in Betracht gezogen werden, um Applikationen für ein dynamisches Fahrverhalten zu erstellen. (vgl. [BFK07], [BR12], [CEH10], [MBB08], [UAB08])

2.1.2 Sensordatenverarbeitung

Nach der Akquirierung der Sensordaten muss das System des autonomen Fahrzeugs diese so interpretieren und korrelieren, dass es ein Modell des eigenen aktuellen Zustands und der Umgebung erstellen kann. Ein Teil dieses Modells kann zum Beispiel die Orientierung und Position des autonomen Fahrzeugs in Relation zu einer Fahrbahn sein. Während die besprochenen Abstands- und Positionssensoren die benötigten Messwerte liefern, müssen Bilddaten zusätzlich verarbeitet werden, um die gewünschten Werte zu erhalten. Wie im vorherigen Abschnitt besprochen, können Mess- oder Rechenfehler, sowie Rauschen die genaue Zustandserfassung fehlerbehaftet oder gar unmöglich machen. Wahrscheinlichkeitstheoretische Filter wurden mit der Beachtung eben dieser Herausforderungen entwickelt. Des Weiteren können diese Filter zur Bestimmung der Trajektorie des autonomen Fahrzeugs eingesetzt werden. Im Folgenden werden zwei Verfahren vorgestellt, die in den meisten Arbeiten zum autonomen Fahren zur Navigation, zur Spurerkennung und zur Spurverfolgung (engl. *tracking*) zu finden sind. Sie dienen als Basis optimierter oder spezialisierter Lösungen. Der Partikelfilter, oder auch sequenzieller *Monte-Carlo-Filter*¹ genannt, und der *Kalman-Filter*² basieren auf dem bayesschen Konzept der bedingten Wahrscheinlichkeiten. (vgl. [E01], [E02], [WB06], [TBF06], [SNS11])

Um den Systemzustand mit dem Partikelfilter abzuschätzen, wird zunächst eine definierte Anzahl an möglichen Systemgrößen, sogenannten Hypothesen, generiert. Diese Größen können zum Beispiel die Position, die Geschwindigkeit, Abstände oder Temperaturen sein. Sie sind zufällig in einem bestimmten Wertebereich verteilt. Anhand von tatsächlichen Messungen mithilfe von Sensoren kann überprüft werden, wie wahrscheinlich jede dieser Hypothesen ist. Je näher eine Hypothese einem Messwert ist, desto höher ist die Wahrscheinlichkeit, dass die Hypothese korrekt ist. Anders ausgedrückt bedeutet das, dass zunächst der Sensormessung eher vertraut wird, als der Hypothese. Je nach Genauigkeit werden diese gemessenen Werte entsprechend gewichtet. Im nächsten Schritt wird erneut

¹ Der *Monte-Carlo-Filter* wird nach dem gleichnamigen Casino in Monaco benannt.

² Der *Kalman-Filter* wurde nach seinem Entwickler Rudolf E. Kálmán (1930-2016) benannt (vgl. [K60]).

eine gewisse Anzahl von Hypothesen zufällig generiert, wobei sie im Bereich der Werte liegen, die zuvor am höchsten gewichtet waren. Dieser Schritt wird *Resampling* genannt. Des Weiteren werden die Hypothesen so verschoben, dass sie entsprechend eines mathematischen Modells des Systems den nächsten Zustand vorhersagen. Handelt es sich zum Beispiel um Positionsdaten, kann anhand der Geschwindigkeit die zukünftige Position zu einem bestimmten Zeitpunkt kalkuliert werden. Mit neuen Messungen können diese Hypothesen erneut überprüft und gewichtet werden. Das Verfahren wird wiederholt, sodass der nächste Systemzustand mit immer genaueren Hypothesen beschrieben werden und diesem nach einigen Iterationen immer mehr vertraut werden kann. Liegt ein Messwert dann plötzlich weit außerhalb der Hypothesen, kann dieser als Fehler erkannt werden. Partikelfilter eignen sich besonders für die Abschätzung nichtlinearer und nichtgaußscher Systeme. (vgl. [F02], [TFB00], [TBF06] S. 238-275, [KFM04])

Der *Kalman-Filter* ist ebenfalls ein mathematisches System, mit dem eine optimale Abschätzung des tatsächlichen unbekannten Systemzustands erreicht werden soll. Dies geschieht durch die Abschätzung bestimmter Größen des Systems sowie durch die entsprechenden Messwerte dieser Größen. Es handelt sich um ein Feedback-Control-System, das aus einem Zustandsbeobachter (engl. *state observer*) besteht, der über die Differenz zwischen der geschätzten und der tatsächlich gemessenen Größe einen Schätzungsfehler berechnet. Die Schätzung wird mithilfe eines mathematischen Modells des Systems gemacht. Ziel ist es nun, den Schätzungsfehler zu verringern, sodass die geschätzte gegen die gemessene Größe konvergiert. Da dieser Vorgang abhängig von dem mathematischen Modell ist, welches das reale System nicht genau beschreiben kann, könnte diese Konvergenz unbestimmt lange dauern. Ein zusätzlicher Verstärkungsfilter, der *Kalman-Gain*, kann diese Konvergenz beschleunigen. Kalman besagt, dass die optimale Abschätzung des Systemzustands aus der Multiplikation des geschätzten und des gemessenen Zustands resultiert. (vgl. [K60], [WB16], [TBF06] S. 39-81)

Es gibt neben den Besprochenen noch weitere Konzepte, welche ähnliche Ansätze verfolgen. Hier sei auf die Arbeiten von Elmenreich [E01] und [E02] hingewiesen. Softwaretechnische Umsetzungen dieser Filter werden, wie eingangs erwähnt, zum *Tracking* von Objekten, Fahrspuren oder Positionen eingesetzt. Jede Lösung hat Vor- und Nachteile, die sich unter anderem in der Genauigkeit der Ergebnisse und der Effizienz der Ausführung zeigen. Da in dieser Arbeit ein Embedded-Board mit eingeschränkten Rechen- und Speicherkapazitäten zum Einsatz kommt, muss eine ressourcenschonende Implementation gewählt werden. In Kapitel 5.3.2 wird darauf näher eingegangen.

2.1.3 Planung

In der Planungsschicht wird anhand der aufgearbeiteten Daten der Sensorfusion das Verhalten des Systems bestimmt. Dieses ist abhängig von der aktuellen Position des autonomen Fahrzeugs in Relation zur wahrgenommenen Umwelt und des Ziels. Das System berechnet in dieser Situation eine Trajektorie, die sich für gewöhnlich in der Mitte des Fahrstreifens befindet. Plötzlich auftauchende Hindernisse müssen beachtet und

Kurskorrekturen geplant werden. Damit wird in der Planungsschicht auch das zukünftige Systemverhalten abgeschätzt. (vgl. [BFK08], [MBB08], [UAB08])

Um nicht nur Ziele in der unmittelbaren, durch die Sensorik erfassbaren Umgebung anzusteuern, sondern auch Strecken über mehrere Kilometer fahren zu können, werden bereits bekannte Routeninformationen, wie Straßen, Kreuzungen und Gebäude, zur Trajektorienberechnung hinzugezogen. Damit kann das autonome Fahrzeug zum Beispiel in einer Kreuzung oder auf einer mehrspurigen Fahrbahn die korrekte Richtung einschlagen. In den zu Beginn genannten Projekten werden Online-Navigationsdienste eingesetzt, die diese Informationen bereitstellen. Sie können aber auch neu erfasste Daten aufnehmen, um sie mit anderen Teilnehmern zu teilen. Das ist zum Beispiel bei neuen Straßen oder Kreisverkehren hilfreich. Solche Dienste wären *Open Street Maps*, *TomTom*, *Google Maps* oder *HERE*. (vgl. [T18a], [W18a])

Ein weiteres Konzept, das im Moment erforscht und entwickelt wird, ist die V2X-Kommunikation. Zwei wichtige Unterkategorien davon sind die Fahrzeug-Fahrzeug-Kommunikation (V2V) und die Fahrzeug-Infrastruktur-Kommunikation (V2I). Dabei kann das Automobil mit der Umgebung interagieren und so zusätzliche Echtzeitinformationen über den Verkehr, wie beispielsweise Staus oder Baustellen, sowie über die Umwelt, das Wetter oder die Fahrbahnverhältnisse akquirieren. (vgl. [R14], [WSK13], [VGG10])

In dieser Schicht muss sich das Verhalten des Fahrzeugs auch an den gesetzlichen Bestimmungen orientieren. Verkehrsschilder mit Geschwindigkeitsbegrenzungen sind einzuhalten, im Bereich von Zebrastreifen haben Fußgänger Vorrang und auch ungeregelte Kreuzungen müssen gefahrlos überquert werden können. Diese gesetzlichen Vorgaben regeln allerdings nicht alle Verkehrssituationen lückenlos. Ein defensives Fahrverhalten ist durch das Reduzieren der Geschwindigkeit oder durch eine Abbremsung bis zum Stillstand auch beim Fehlverhalten anderer Verkehrsteilnehmer sinnvoll. (vgl. [MBB08], [BR12], [MGL15])

Schließlich muss in der Planungsschicht auch auf Benutzerinteraktionen, die etwa über eine entsprechende Benutzerschnittstelle (*human machine interface*, HMI) eingegeben werden, reagiert werden. Eine solche Eingabe kann zum Beispiel eine Fahrzieleingabe oder der Befehl zum autonomen Einparken sein. (vgl. [KPK14], [WSK13])

2.1.4 Fahrzeugsteuerung

Für die physische Fortbewegung des Fahrzeugs ist nun die Schicht zur Fahrzeugsteuerung zuständig. Sie bestimmt die konkrete Geschwindigkeit und die Lenkrichtung des Fahrzeugs. Abhängig von der berechneten Trajektorie der Planungsschicht wird hier eine Befehlsfolge für die Aktorensteuerungen generiert und an diese weitergeleitet. Zu einem großen Teil verfügen Fahrzeuge heutzutage über elektronischen Steuerungssysteme (engl. *drive by*

*wire*³) für die Lenkung, das Gas und die Bremse. Damit ist die notwendige Infrastruktur für die Kommunikation zwischen dem autonomen System und dem Fahrzeug bereits vorhanden. Diese Kommunikationskanäle sind auch in den Standards *CAN-Bus*⁴ und *FlexRay*⁵ definiert. Das in dieser Arbeit verwendete Modellfahrzeug verfügt über eine einfachere Motorsteuerungsmethode, nämlich über pulsweitenmodulierte Signale (PWM). Die Aufgabe der Fahrzeugsteuerung bleibt dennoch die gleiche, unabhängig von der Signalisierungsmethode. (vgl. [ISO13], [ISO15], [RBL09], [E12] S. 237)

Die Berechnung des Lenkwinkels ist abhängig von mehreren Werten, wie der Geschwindigkeit, der Lenkwinkeldifferenz der einzelnen Räder, des Reibungswerts der Reifen sowie der Länge und der Breite des Fahrzeugs. Eine vereinfachte Möglichkeit zur Annäherung ist es, die vier Räder des Automobils virtuell auf zwei zu reduzieren, die sich jeweils in der Mitte der Achsen befinden. Der Lenkwinkel wäre hierbei nur vom Kurvenradius und vom Radstand abhängig, also dem Abstand zwischen den Radachsen. Zu beachten ist, dass der so geschätzte Winkel nur für kleinere Geschwindigkeiten korrekt ist. Bei höheren Geschwindigkeiten müsste zum Beispiel auch die Fliehkraft beachtet werden, die gegen die Lenkrichtung wirkt. In dieser Arbeit wird ebenfalls ein dynamisches, wenn auch vereinfachtes Lenkmodel implementiert. (vgl. [RBL09], [E12] S. 168-187)

An die Motorsteuerung muss für den Vortrieb ein Akzelerationswert übertragen werden, der, wie bei einem Gaspedal, die Motordrehzahl regelt und somit beschleunigt. Bei einem elektrischen Modellauto erlaubt der elektronische Geschwindigkeitsregler (engl. *electronic speed controller*, ESC) die Steuerung der Energiezufuhr des Motors. Dieser Regler ermöglicht auch das Rückwärtsfahren, ohne zusätzliches Getriebe. Das ist durch die Umpolung des Motors möglich. Über das Festlegen der Beschleunigungswerte muss auch eine Bremsfunktion umgesetzt werden, weil kein eigenes Bremssystem zur Verfügung steht. In realen Fahrzeugen mit Otto- oder Dieselmotoren, wird die Motordrehzahl über das Brennstoffluftgemisch geregelt. Da sich diese Motortypen nur in eine Richtung drehen können, benötigen sie ein automatisches Getriebe mit dem, über mehrere Vorwärtsgänge und einen Rückwärtsgang, die Räder angetrieben werden. (vgl. [JKK14], [JKK15], [SCB13], [R14])

2.2 Bildverarbeitung

Alleine die Bilddurchsuche einer digitalen Kamera ist nicht ausreichend, um Entscheidungen für das autonome Fahren treffen zu können. Eine geeignete Interpretation dieser Daten ist

³ In *Drive-by-Wire*-System wird das Fahren oder Steuern des Fahrzeugs, statt über mechanische Befehlsübertragungen, über elektrische Signale und elektromechanische Aktoren, wie Servomotoren bewerkstelligt.

⁴ CAN, oder *Controller Area Network*, ist ein ISO standardisiertes Bussystem (ISO 11898) zur Datenübertragung zwischen zwei oder mehreren Teilnehmern über eine gemeinsame Leitung. (vgl. [ISO15])

⁵ FlexRay ist ein ISO standardisiertes Bussystem (ISO 17458) zur fehlertoleranten und deterministischen, seriellen Datenübertragung zwischen zwei oder mehreren Teilnehmern über eine gemeinsame Leitung. (vgl. [ISO13])

notwendig. Dabei ist das Erkennen relevanter Informationen in einer dynamischen Umwelt nicht trivial. Wie auch ein menschlicher Führerscheinneuling, muss das autonome System zuerst lernen, worauf zu achten ist. Andere Fahrzeuge, Personen, Hindernisse, Straßen, Verkehrsschilder oder Gebäude haben keine einheitlichen Farben, Formen, Texturen, Lagen und Positionen. Dennoch können sie anhand spezifischer Merkmale unterschieden werden. In Objekterkennungsverfahren werden zwei Methoden dafür angewendet. Eine Möglichkeit ist, dass das System nach Merkmalen sucht, die mit einem Modell des gesuchten Objekts übereinstimmen. Dieses Modell kann zum Beispiel graphisch oder mathematisch beschrieben sein. Die andere Möglichkeit ist, dass das System in einem ersten Schritt selbst relevante Merkmale des Objekts mithilfe eines Extraktions- und Lernverfahrens findet. Im zweiten Schritt wird nach den gefundenen Objekteigenschaften im Bild gesucht. Beide Varianten werden im Weiteren betrachtet. (vgl. [S14], [IO14])

Wie in der Funktionsweise einer Kamera in Kapitel 2.1.1 beschrieben, ist die Bildaufzeichnung abhängig vom verfügbaren Licht. Je mehr Licht vorhanden ist, desto deutlicher werden Inhalte in der Aufzeichnung sichtbar. Zu viel Licht erzeugt allerdings auch Schatten oder Gegenlicht, die wiederum die Wahrnehmung durch die Kamera, genauso wie beim Menschen, stören. Es kann zu Fehlinterpretationen kommen. Vor der Objekterkennung ist eine Bildverarbeitung notwendig, um die Helligkeit und den Kontrast anzupassen, aber auch um das Rauschen zu reduzieren.

Diese Arbeit konzentriert sich auf zwei Funktionen des autonomen Fahrens, die Fahrbahn- und die Verkehrsschilderkennung. Dazu werden bestehende Verfahren betrachtet und anhand ihrer Eignung zur Implementierung auf einem Embedded-Board mit eingeschränkten Ressourcen ausgewählt. Es existieren allerdings noch wesentlich mehr Verfahren als die hier beschriebenen, zumal dieses Gebiet nach wie vor erforscht wird.

2.2.1 Fahrbahnerkennung

Die Fahrbahnerkennung ist zweifelsohne der wichtigste Bestandteil für die akkurate Steuerung des Automobils auf öffentlichen Straßen. Die besondere Herausforderung ist, dass eine Fahrbahn in ihrer Beschaffenheit, Form und Farbe im realen Verkehr stark variieren kann. Hilfreich für die Identifikation sind Fahrbahnmarkierungen. Es gibt einfache und doppelte, durchgezogene oder unterbrochene Linien, die ein Überfahren erlauben oder verbieten. Abbildung 5 zeigt die möglichen Längsmarkierungen nach der österreichischen Straßenverkehrsordnung (StVO). Fahrzeuge, die über Landesgrenzen hinausfahren, müssen zudem auch andere rechtliche Vorgaben einhalten. Sperrlinien untersagen das Überfahren bzw. Wechseln einer Fahrspur. Am Straßenrand kennzeichnen sie als Begrenzungslinien den Straßenrand. Leitlinien befinden sich meist in der Fahrbahnmitte und dürfen zum Spurwechseln überfahren werden. Doppelte Sperrlinien und Leitlinien sind auf mehrspurigen Straßen in der Fahrbahnmitte zu finden. Die kombinierte Sperr- und Leitlinie gestattet das einseitige Überfahren der Markierung von der Seite der Leitlinie her. In Österreich sind Fahrbahnmarkierungen weiß bzw. im Baustellenbereich orange (vgl. [BMV02]).

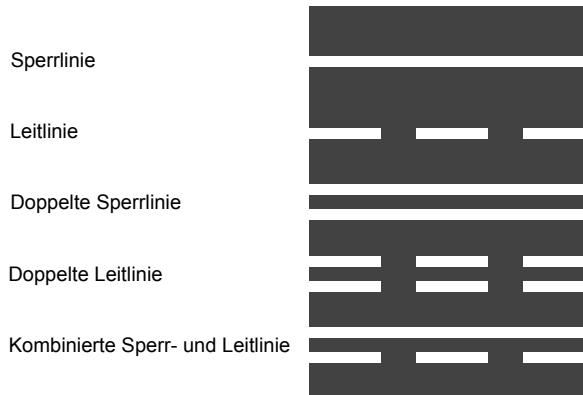


Abbildung 5: Mögliche Längsmarkierungen nach der StVO (vgl. [BMV02])

In Abbildung 6 ist ein Modell einer Fahrbahn abgebildet, wie es beim *Carolo-Cup* zu finden ist und in dieser Arbeit ebenfalls verwendet wird. An den Außenrändern der Fahrbahn befinden sich durchgezogene Begrenzungslinien. Wenn diese Bahn aus mehr als einer Fahrspur besteht, werden diese durch unterbrochene Leitlinien zwischen den Spuren markiert. Zwar ist in der Realität die Länge der Linienabschnitte bzw. der Abstände zwischen ihnen gesetzlich definiert, doch sind diese je nach Land unterschiedlich. Die Markierung soll jedoch unabhängig von diesen Varianten erkannt werden. Es wird angenommen, dass der Fahrbahnuntergrund, also zum Beispiel der Asphalt, deutlich dunkler als die Markierungen ist. Im tatsächlichen Straßenverkehr sind Begrenzungszeichnungen nicht immer verfügbar. Über den Winter können Risse oder Löcher in der Fahrbahn entstehen, welche die Markierungen ebenfalls beschädigen und unterbrechen. Auch Schmutz und Hindernisse machen die Linie abschnittsweise unsichtbar. Die Detektion muss die Fahrbahn trotz solcher Lücken zuverlässig erkennen. (vgl. [TUB18], [TUM13], [S09], [S13], [NVF13])



Abbildung 6: Fahrbahnmodell eines geraden Straßenabschnitts

Kaur et al., Kumar et al. und Maldlik et al. vergleichen in ihren Arbeiten Algorithmen für die Fahrbahnerkennung und -verfolgung. Abbildung 7 stellt den Ablauf der meisten dieser Algorithmen dar. Zuerst wird eine Vorverarbeitung durchgeführt, um die Fahrbahnmarkierung sichtbarer zu machen. Als nächstes wird eine Kantenfilterung auf das Bild angewendet, um die Markierung vom Untergrund zu extrahieren. Der letzte Schritt dient der Verfolgung der Fahrbahn. Diese Schritte werden im Folgenden genauer beschrieben. (vgl. [JK05], [YGD11], [SL14], [W18c]).

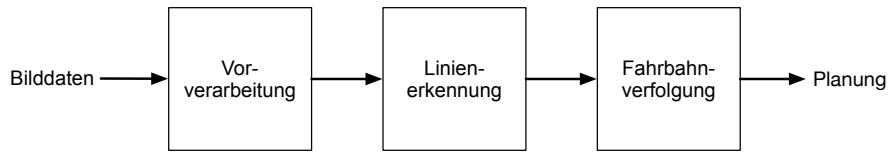


Abbildung 7: Erkennung einer Fahrbahnmarkierung (vgl. [K07], [YGD11])

Je nach Lichtverhältnissen und Kamerasensor wird ein Rauschen mitaufgezeichnet, das in der weiteren Verarbeitung störend ist. Zwei Methoden schaffen hier Abhilfe. In den erwähnten verwandten Arbeiten ist ersichtlich, dass keine hohe Auflösung der Fahrbahnbilddaten notwendig ist. Mit der Reduktion der Auflösung wird auch das Bildrauschen geringer. Daneben können auch die weiteren Verarbeitungsschritte schneller durchgeführt werden, da weniger Pixel bearbeitet werden müssen. Die Kamera ist meist so platziert, dass nicht nur die Fahrbahn aufgezeichnet wird. Hier lassen sich zusätzlich Pixel einsparen, indem nur ein relevanter Bildausschnitt (engl. *region of interest*, ROI) analysiert wird. (vgl. [BGL15])

Eine weitere Maßnahme zur Rauschreduktion ist die Weichzeichnung des Bildes. Dazu wird eine Faltungsmatrix, auch Faltungskern genannt, auf das Bild angewendet. Je nach Faltungart wird eine quadratische Matrix mit ungerader Abmessung, typischerweise mit 3x3 oder 5x5 Feldern, gewählt, die wie ein Fenster über jedes Pixel und dessen Nachbarn gelegt wird. Ein neuer Pixelwert wird nun wie folgt berechnet, wobei I das Ursprungsbild, I^* das Ergebnisbild, a die Position des mittleren Wertes der Faltungsmatrix, k die Faltungsmatrix und n die Größe der Faltungsmatrix ist. Bekannte Weichzeichnenfilter sind der Median- oder der Gaußfilter. (vgl. [BGL15])

$$I^*(x, y) = \sum_{i=1}^n \sum_{j=1}^n I(x + i - a, y + j - a) * k(i, j)$$

Der nächste Schritt in der Vorbereitung des Bildes ist die Anpassung der Helligkeit und des Kontrastes. Ziel ist es, die Helligkeitsintensität der einzelnen Pixel in Relation zur maximalen Intensität in der Aufnahme zu setzen. Wie in Kapitel 2.1.1 beschrieben, nimmt eine handelsübliche Kamera diese Einstellungen automatisch, aber mit einiger Verzögerung vor. Die ROI könnte abhängig von den Lichtverhältnissen in einem dunkleren Teil des Bildes liegen, wo der höchste Helligkeitswert nicht dem des Gesamtbildes entspricht. Deshalb kann es hilfreich sein, zusätzliche Änderungen der Helligkeit und des Kontrasts in diesem Bereich vorzunehmen. Objekte oder Fahrbahnmarkierungen im Schattenbereich werden damit deutlicher. Die folgende Formel zeigt den Zusammenhang zwischen Helligkeit und Kontrast in einem Bild. Die Matrix I beinhaltet das Ursprungsbild, der Parameter α steht für die Helligkeit, β für den Kontrast und I^* für das Ergebnisbild.

$$I^*(x, y) = \alpha * I(x, y) + \beta$$

In den meisten Arbeiten zur Fahrbahnerkennung ist ein weiterer Verarbeitungsschritt zu finden, nämlich die inverse Perspektivtransformation (engl. *inverse perspective transformation*, IPT). Um Objekte vor dem autonomen Fahrzeug mit einer Kamera zu erfassen, ist deren Bildsensorfläche nicht parallel zur Fahrbahn ausgerichtet. Sie liefert daher ein perspektivisch verzerrtes Bild der Fahrbahn. Um die Lage, die Position und die

Abstände gerader Straßenmarkierungen besser zu erkennen bzw. zu messen, kann diese Verzerrung rückgängig gemacht werden. Abbildung 8 zeigt ein Beispiel für die perspektivische Transformation einer schematischen Straßendarstellung. Sind die intrinsischen Kameradaten, wie die Brennweite, die Bildmitte, die Pixelskalierung, sowie die extrinsischen Positionsinformationen, wie der Kamerawinkel zur Fahrbahn, der Winkel der Kamera in Fahrtrichtung und die Höhe der Kamera bekannt, kann eine Transformationsmatrix berechnet werden. Das Problem ist, dass nicht alle diese Werte verfügbar sind. Zum Beispiel geben manche Kamerahersteller keine genaue Auskunft über die Objektivbrennweite. Dennoch können fehlende Werte rechnerisch mit einem Kalibrierungsverfahren annäherungsweise bestimmt werden, wie in Kapitel 5.2.1 gezeigt wird. Ein anderer pragmatischer Weg ist es, vier Koordinatenpunkte im Bild zu wählen, welche die Fahrbahn umfassen. Aufgrund der perspektivischen Verzerrung schließen sie ein konkaves Viereck oder Trapez ein. Diese Punkte bildet man auf vier weiteren Punkten ab, die ein Rechteck formen. Das Ergebnis dieser Abbildung ist eine Transformations- oder Homographiematrix, die Informationen über die Translation, die Rotation, die Verzerrung und die Skalierung der Bildflächen enthält. (vgl. [B95])

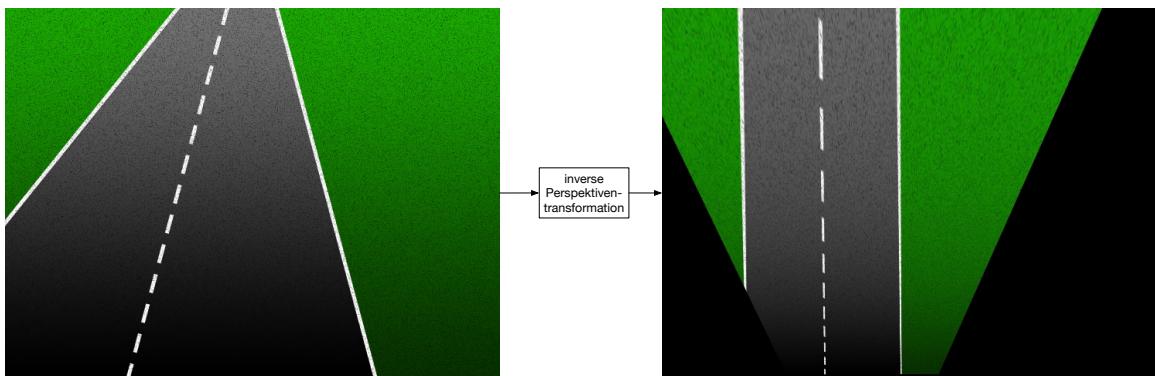


Abbildung 8: Inverse Perspektiventransformation

Fahrbahnmarkierungen sind im Normalfall deutlich heller als der Untergrund, auf dem sie gezeichnet sind. Um ihre Position in einem Bild zu finden, können sie mit einem Kantenfilter vom Hintergrund extrahiert werden. Eine weitverbreitete Methode dafür ist der *Canny-Filter*⁶. Er basiert auf dem Prinzip des *Sobel*-Operators, bei dem Kanten über Sprünge im Helligkeitsverlauf eines Bildes erkannt werden. Dazu wird die erste Ableitung der Helligkeitswerte berechnet. Gleichzeitig werden die Pixel orthogonal zur Ableitungsrichtung weichgezeichnet. Die Ableitungen werden durch die folgenden Formeln durchgeführt, wobei I das Ursprungsbild, I_x^* das Ergebnisbild in X-Richtung und I_y^* in Y-Richtung ist. S_x und S_y sind die *Sobel*-Operatoren in den entsprechenden Richtungen. Die Summe beider Ergebnismatrizen ergibt ein Kantenbild I^* unabhängig von der Richtung.

$$I_x^* = S_x * I, \quad S_x = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} \quad \text{und} \quad I_y^* = S_y * I, \quad S_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

$$I^* = \sqrt{I_x^{*2} + I_y^{*2}}$$

⁶ Der Canny-Filter wurde von John F. Canny (1958) 1986 entwickelt (vgl. [C86]).

Von dem Filter werden Graustufenbilder verarbeitet. Das Ergebnis der Filterung ist ein Binärbild mit den detektierten Kanten. Da das Bildrauschen unerwünschte Sprünge in der Helligkeitsintensität eines Bildes bewirken kann, muss dieses auf ein Minimum reduziert werden. Beim Canny-Filter wird deshalb zusätzlich ein Normalverteilungsfilter zur Glättung eingesetzt. Je stärker die Glättung, desto robuster ist der Filterungsalgorithmus. Das wiederum kann dazu führen, dass feinere Kanten im Bild übersehen werden. Abbildung 9 zeigt das Ergebnis der Kantenfilterung der IPT aus Abbildung 8. (vgl. [K07], [YGD11])

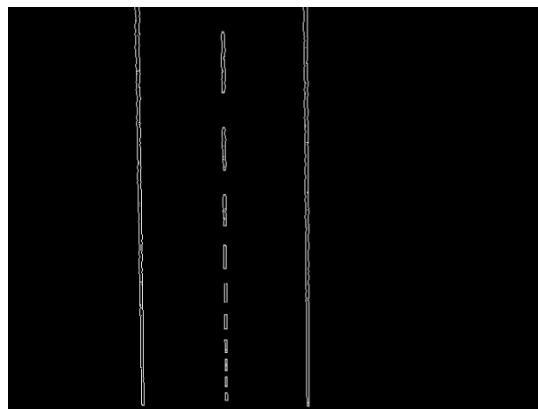


Abbildung 9: Canny-Filterung

Es ist möglich, dass nach dem letzten Verarbeitungsschritt dennoch Kanten detektiert worden sind, die nicht Teil einer Fahrbahnmarkierung sind. In vielen Arbeiten wird die *Hough*⁷-Linien-Transformation dazu genutzt, um eine zusätzliche Filterung von geraden Linien durchzuführen (vgl. [JK05], [YGD11], [K07]). Aus dem Binärbild der Canny-Filterung wird für jeden hellen Punkt mit der Geradengleichung in der hessischen Normalform $\rho = x * \cos(\theta) + y * \sin(\theta)$ eine Kurve in einem Parameterraum (ρ, θ) gezeichnet. Für den Winkel θ gilt $0 < \theta < 2\pi$ und für den Abstand $\rho > 0$. Abbildung 10 stellt diesen Zusammenhang noch einmal graphisch dar. Die *Hough*-Transformation nimmt für jeden Punkt der Gerade im linken Diagramm alle theoretisch möglichen Linien durch diesen Punkt an und berechnet die genannten Parameter dafür, wie im rechten Diagramm zu sehen ist. Alle Punkte, die auf einer gemeinsamen Geraden liegen, müssen daher nach der Transformation zumindest ein gleiches Parameterpaar (ρ, θ) haben.

⁷ Das ist ein Verfahren, welches 1962 von Paul V. C. Hough entwickelt und anschließend patentiert wurde.

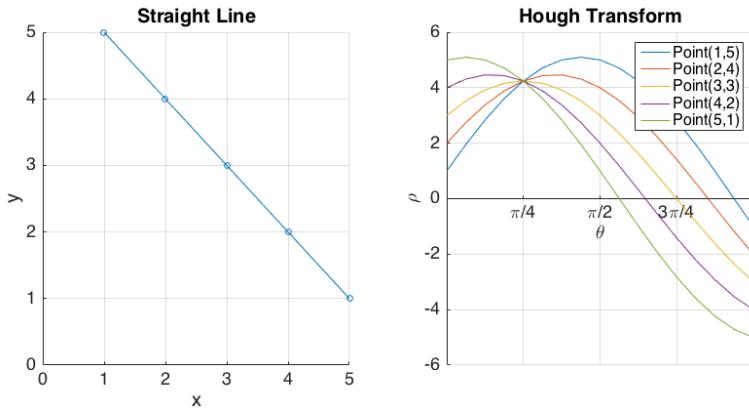


Abbildung 10: *Hough*-Transformation einer Geraden

Die Häufigkeit dieser Parameterpaare lässt sich in eine Matrix eintragen. Je länger eine Gerade ist, desto höher ist der Häufigkeitswert der entsprechenden Parameter (ρ, θ) . Das Ergebnis der *Hough*-Transformation ist schließlich eine Auflistung aller gefundenen Linien. Die *Hough*-Transformation der Kanten aus Abbildung 9 ist in Abbildung 11 dargestellt. Die hellen Stellen entsprechen den Häufungspunkten und kennzeichnen die drei erkannten Linien. (vgl. [JK05], [YGD11], [K07])

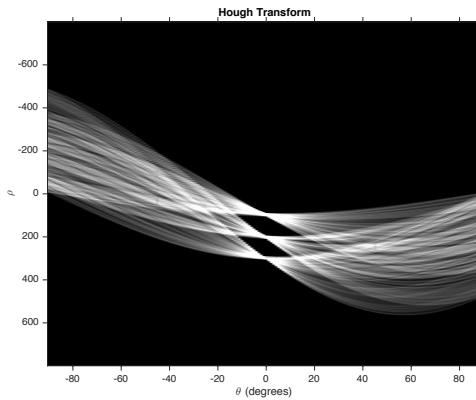


Abbildung 11: *Hough*-Transformation einer geraden Fahrbahn

Anhand der Linien und deren Verlauf im Bild kann nun auf die Position des autonomen Fahrzeugs relativ zu den Markierungen zurückgeschlossen werden. Wenn die Kamera entlang der Longitudinalachse des Fahrzeugs angebracht ist, dann ist die Bildmitte auch die Fahrzeugmitte. Über die Distanzen zwischen den erkannten Markierungen und der Fahrzeugmitte kann nun auch die Ist- und Sollposition des Fahrzeugs innerhalb der Fahrbahn berechnet werden. Als Sollposition wird in diesem Fall die Fahrbahnmitte angenommen, kann aber bei anderen Fahrtroutenplanungen, zum Beispiel beim Überholen, auch anders definiert werden. Der Winkel θ der Linien gibt auch Auskunft über die Lage dieser zu einander. Ist der Winkel gleich, so sind die Linien parallel zueinander. Diese Winkel können auch zur Lenkwinkelberechnung herangezogen werden. Im folgenden simulierten Fall aus Abbildung 12 befindet sich das Fahrzeug innerhalb der blauen und der roten Markierung, welche die Mittel- sowie die rechte Begrenzungslinie markieren. Die vordere Stoßstange des Fahrzeugs und die Geradeausfahrttrichtung sind zur Orientierung gelb eingezeichnet. Zu erkennen ist, dass die Longitudinalachse des

Fahrzeugs nicht in der Fahrbahnmitte ist. Eine Richtungskorrektur wäre sinnvoll, aber nicht dringend notwendig, weil sich das Fahrzeug immer noch zwischen den Markierungen befindet. (vgl. [LSA09], [BGL15], [YGD11])

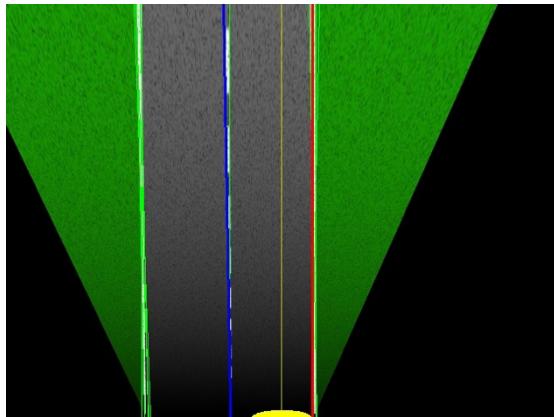


Abbildung 12: Straßenmarkierungserkennung

Bei einem fahrenden Auto ist die Aufnahme und die Auswertung mehrerer Bilder pro Sekunde notwendig, um rechtzeitig auf Veränderungen im Fahrbahnverlauf zu reagieren. Die Kantenfilterung und Linienerkennung ist trotz reduzierter Bildauflösung rechenintensiv, da jedes einzelne Bild Pixel für Pixel gefiltert und transformiert werden muss. Eine Vorhersage der Position der Markierung im nächsten Bild kann die Verarbeitung wesentlich beschleunigen, indem nur ein entsprechend kleiner Ausschnitt des Bildes analysiert wird. Der *Kalman*- oder Partikelfilter aus Kapitel 2.1.2 kann hierfür eingesetzt werden, wie in den oben genannten Arbeiten gezeigt wird.

2.2.2 Verkehrsschilderkennung

Im Straßenverkehr gibt es eine Vielzahl von Objekten, für die kein allgemeines Modell wie im vorherigen Kapitel definiert werden kann. Manche können ihre Position verändern und damit plötzlich auf die Fahrbahn gelangen. Andere verändern ihre Form, wie das bei einem sich bewegenden Menschen der Fall ist. Bestimmte Objekte unterstützen das autonome Fahrzeug darin, sich korrekt zu verhalten, weshalb sie von diesem detektiert werden müssen. Besonders wichtig sind hier Verkehrsschilder. Sie kommunizieren Vorschriften, Gefahren und wichtige Hinweise. Die geltenden gesetzlichen Verkehrsregeln sind dabei genau einzuhalten. Wird zum Beispiel aufgrund einer Baustelle auf einem Straßenabschnitt ein Gefahrenschild und eine reduzierte Höchstgeschwindigkeit angezeigt, muss auch das selbstfahrende Fahrzeug rechtzeitig die Geschwindigkeit entsprechend reduzieren.

In Publikationen zur Objekterkennung in ADAS und zum autonomen Fahren sind zwei unterschiedliche Ansätze zu finden. In dem einen werden klassische Verfahren zur Merkmalerkennung mit Maschinenlernverfahren kombiniert. Neuere Forschungen hingegen setzen auf künstliche neuronale Netze (engl. *artificial neural network*, ANN) und simulieren damit das Lernprinzip des menschlichen Gehirns. Beide Ansätze benötigen eine Vorbereitungsphase. Im Gegensatz zur Fahrbahnerkennung, bei der einfache linienförmige

Muster erkannt werden, müssen bei der Objekterkennung auch komplexe Formen vom System identifiziert werden. Ähnlich wie der Mensch kann ein Computer ebenso die visuellen Merkmale, die ein bestimmtes Objekt ausmachen, erlernen. Diese Lernphase wird auch Training genannt. Dazu werden dem System positive Bilder (engl. *samples*), in denen sich das zukünftig zu erkennende Objekt befindet, und negative Bilder, in denen das Objekt nicht vorkommt, gezeigt. Je mehr positive und negative Bilder für das Training verwendet werden, desto besser bzw. fehlerfreier wird die Objekterkennung schließlich durchgeführt. Dieser Vorgang ist rechenintensiv und nimmt viel Zeit in Anspruch. Ein Grund dafür ist, dass eine große Menge an Lernmaterial, das heißt positive wie negative Samples, erstellt oder aufgenommen werden muss. Um diesen Teil der Trainingsphase zu beschleunigen, gibt es Softwarewerkzeuge zur automatischen Bildergenerierung. Diese verändern vorhandene Samples in ihrer Größe, Orientierung oder Position, um neue Variationen zu erhalten. In weiterer Folge muss in jedem Sample, je nach Lernverfahren, nach Eigenschaften gesucht werden, die das gewünschte Objekt am besten beschreiben. Im Folgenden werden die drei populärsten Objekterkennungsverfahren vorgestellt. Sie unterscheiden sich hauptsächlich in der Definition der visuellen Merkmale eines Objektes. (vgl. [B12], [HHH09], [NCP07], [VJ01], [W18c])

Ein Maschinenlern- und Objekterkennungsverfahren, das für Computer mit eingeschränkter Rechenkapazität entwickelt wurde, ist die Kaskadenklassifizierung von Viola und Jones [VJ01], welche auf der Arbeit von Papageorgiou et. al [POP98] aufbaut. Weiter unten in Abbildung 14 ist der Ablauf des Verfahrens dargestellt. Dem System werden auch hier positive Samples eingegeben. Aber auch negative Samples werden dem System mit der Information vorgelegt, dass das zu suchende Objekt auf dem Bild nicht zu sehen ist.

In dem Eingabebild wird nach bestimmten Merkmalen gesucht, die einer *Haar*⁸-*Wavelet*⁹-Transformation ähneln. Diese Merkmale sind rechteckige Hell-Dunkel-Kombinationen, wie in Abbildung 13 abgebildet, wobei auch Variationen von diesen möglich sind.

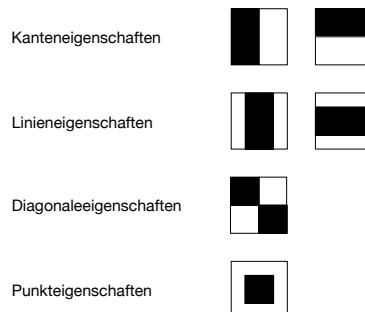


Abbildung 13: *Haar*-Merkmale (vgl. [POP98], [B12], [VJ01])

Um ein Objekt mit diesen Merkmalen zu beschreiben, wird mithilfe eines Suchfensters das Eingabebild Abschnitt für Abschnitt analysiert. Dieses Fenster wird noch einmal in zwei, drei oder vier Rechtecke unterteilt. Innerhalb jedes Rechtecks wird die Summe der

⁸ Alfréd Haar (1885 – 1933) war ein ungarischer Mathematiker.

⁹ Ein *Haar-Wavelet* ist eine rechteckige Transformationsfunktion zur Analyse von n-dimensionalen Funktionen, die vom Mathematiker Alfréd Haar vorgeschlagen wurde.

Helligkeitsintensitätswerte berechnet. Im nächsten Schritt werden die Differenzen zwischen den Summen aus den einzelnen Rechtecken verglichen. Höhere Summen werden dabei als helle Felder, niedrigere als dunkle interpretiert. Betrachtet man nun das gesamte Suchfenster, entstehen Muster, wie sie in Abbildung 13 zu sehen sind. Das am besten passende *Haar-Merkmal* wird zur Beschreibung des einzelnen Suchfensters gewählt. Dieser Vorgang wird für jedes Suchfenster wiederholt. Nach und nach entsteht so eine Auswahl an Merkmalen, welche auch komplexere Formen, wie Fahrzeuge oder Personen, beschreiben können. (vgl. [POP98], [VJ01])

Mit einer Optimierung namens *Adaptive Boosting (AdaBoost)* wird die *Haar-Kaskadenklassifizierung* so beschleunigt, dass besonders häufige Merkmale als wichtiger eingestuft werden und daher in neuen Bildern insbesondere nach diesen gesucht wird. Um nun festzustellen, welche der Eigenschaften wichtig ist, muss die Klassifizierung in mehreren Stufen durchgeführt und nach häufig auftretenden Merkmalen gesucht werden. In jeder Stufe wird nur nach einem Teil der Eigenschaften gesucht. Werden die Merkmale gefunden, wird das Sample für eine nächste Kaskadenstufe bereitgehalten, andernfalls wird es verworfen.

Das Ergebnis der *Haar-Kaskadenklassifizierung*, wird in einer XML-Datei gespeichert. Nach dem Lernverfahren wird diese zur Detektion eingesetzt. Mit einem Suchfenster, in der Größe der zum Training verwendeten Samples, wird im Eingabebild nach den Eigenschaften, die das gewünschte Objekt beschreiben, gesucht. Für eine Vielzahl von Objekten gibt es jedoch bereits fertige Klassifizierungsdaten. (vgl. [HHH09], [VJ01])

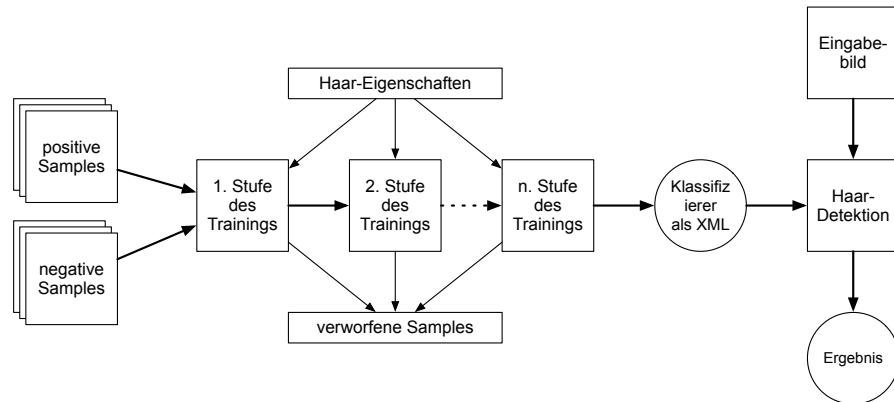


Abbildung 14: Kaskadenklassifizierung

Ein anderes Verfahren wurde von Dalal und Triggs 2005 vorgeschlagen, welches visuelle Objekteigenschaften anhand des Histogramms der Gradientenrichtungen (engl. *histogram of oriented gradients*, HOG) in einem Bild definiert (vgl. DT05). Dazu wird dem Verfahren in einem ersten Schritt ein Bild mit dem zu suchenden Objekt eingegeben. Von diesem Bild wird eine Ableitung in X- und Y-Richtung berechnet. Dabei wird die Differenz der Helligkeitswerte für jedes Pixel im Vergleich zu denen der unmittelbaren Nachbarpixel in den jeweiligen Richtungen ermittelt. Hierfür kann auch der *Sobel-Operator* mit einer Kernelgröße 1, also mit $S_x = [-1 \ 0 \ 1]$ und $S_y = \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$, eingesetzt werden. Durch die Bildfaltungen $G_x = I * S_x$ und $G_y = I * S_y$ lassen sich die Gradienten für die jeweiligen

Richtungen berechnen. Mithilfe dieser können die Beträge mit $|G| = \sqrt{G_x^2 + G_y^2}$ und die Richtungen $\theta = \arctan(\frac{G_x}{G_y})$ für jedes Pixel bestimmt werden. Das Eingabebild wird in gleichmäßige 8x8-Pixelfenster aufgeteilt. Für jedes Fenster wird aus den berechneten Beträgen $|G|$ und Richtungen θ der Pixelgradienten, ein Histogramm erstellt. Dieses entsteht so, dass die Beträge anhand ihres Winkels zu einer von neun Richtungsklassen zugeordnet werden. Die Summen der Beträge jeder Klasse werden verglichen und die höchste wird gewählt, um die Gesamtrichtung für das Fenster zu bestimmen. Wenn die Ausleuchtung des Objektes nicht optimal ist, sind die Beträge im Schattenbereich niedriger, als die im Lichtbereich. Dem kann mit einer Normierung der Histogramme von vier benachbarten Fenstern entgegengewirkt werden. Im letzten Schritt wird ein Merkmalsvektor aus allen normierten Histogrammen zusammengesetzt, der das Objekt beschreibt. Dieser Vektor kann ebenfalls mit einem Klassifizierungssystem verwendet werden.

Die genannten Verfahren haben seit ihrer ersten Vorstellung Modifikationen und Weiterentwicklungen erfahren. Sie werden zum Beispiel in den weitverbreiteten Objekterkennungsverfahren *SIFT*¹⁰, *SURF*¹¹ oder *FAST*¹² eingesetzt. Eine optimale Lösung hängt stets von den angestrebten Zielen ab. Genauere Verfahren tendieren dazu, eine höhere Berechnungszeit aufzuweisen oder mehr Speicherkapazität zu benötigen. Für diese Arbeit wurde die *Haar-Kaskadenklassifizierung* gewählt, weil sie speziell für Computer mit eingeschränkten Ressourcen, wie den hier eingesetzten, entwickelt wurde. Natürlich kann das Projekt in Zukunft, mit einer passenderen Lösung modifiziert werden.

In aktuellen Arbeiten zum autonomen Fahren kommen artificielle neuronale Netze zur Objekterkennung zum Einsatz. Sie imitieren die Struktur und das Lernverhalten eines menschlichen Gehirns, indem sie vereinfachte künstliche Neuronen bzw. Knoten miteinander verknüpfen. In Abbildung 15 wird ein solches Netz schematisch dargestellt. Dieses sieht so aus, dass Knoten einer Eingabeschicht (engl. *input layer*) mit denen einer Ausgabeschicht (engl. *output layer*) über die Knoten einer oder mehrerer versteckter Zwischenschichten (engl. *hidden layers*) verbunden werden. Dem Netz wird ein Eingabebild mit dem zu erkennenden Objekt darauf vorgelegt. Jedes Pixel wird einem Knoten der Eingabeschicht zugeordnet. Von dort werden Informationen über die Knoten der verborgenen Schichten weitergeleitet. So ein künstliches Neuron verfügt über einen oder mehrere Ein- und Ausgänge, die unterschiedlich gewichtet werden können. Diese Gewichtungen stellen Schwellenwerte für die Annahme eines Eingangswertes dar. Das Neuron selbst manipuliert die Werte über eine vordefinierte Funktion und gibt das Ergebnis an die Ausgänge weiter. Das Netz lernt, indem Verbindungen zwischen Neuronen der einzelnen Schichten gewichtet, neue Verbindungen hinzugefügt oder getrennt werden.

¹⁰ Die *Scale-invariant Feature Transform* (SIFT) wurde 1999 von Lowe entwickelt (vgl. [L99]).

¹¹ Das Verfahren *Speed Up Robust Features* (SURF) ist eine Weiterentwicklung von SIFT und wurde von Bay et al. 2006 vorgestellt (vgl. [BET06]).

¹² FAST ist ein schnelles Merkmalerkennungsverfahren, das von Rosten et al. 2005 vorgestellt wurde und seitdem weiterentwickelt wird (vgl. [RD05], [RD06]).

Schließlich gelangt an der Ausgabeschicht die Information an, um welches Objekt es sich handelt.

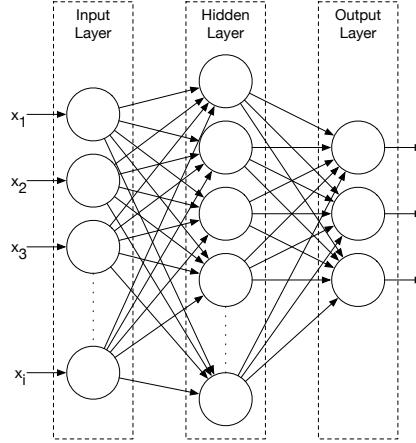


Abbildung 15: Schematische Darstellung eines künstlichen neuronalen Netzwerks

Durch das Training eines ANN mit positiven und negativen Samples wird die Erkennung des Objekts immer zuverlässiger. Es gibt verschiedene ANN-Konzepte und Trainingsverfahren, wobei diese zu erläutern, über den Rahmen dieser Arbeit hinausgeht. Hier sei auf die einführende Arbeit von Goodfellow et al. [GBC16] verwiesen. Der besondere Vorteil von ANN ist, dass beliebige Objekte trainiert werden können, ohne dem System die gesuchten Eigenschaften zuvor zu vermitteln. Damit kann das Netz auch die Erkennung komplexer und sich verändernder Objekte erlernen. Derzeit befindet sich der Einsatz von ANN für das autonome Fahren noch in der Erforschung und Entwicklung. Die Firma NVIDIA beispielsweise arbeitet an Lösungen für spezielle Embedded-Systeme mit *Graphic Processing Unit* (GPU) (vgl. [BDD16]). Der Einsatz von ANN führt jedoch über das Ziel dieser Arbeit hinaus. Die Leistungsansprüche an das ausführende System sind noch zu hoch für einen Embedded-Computer, wie er hier zum Einsatz kommt. Für einen Einstieg in ANN und *Deep Learning* sei unter anderem auf die Arbeiten von Bengio und LeCun verwiesen (vgl. [BL07], [GBC17], [GB07]).

2.2.3 Optische Abstandsmessung

In einem monokular aufgezeichneten Bild können neben der Fahrbahn und anderen Objekten auch die Abstände des Fahrzeugs zu ihnen berechnet werden. Das Prinzip basiert auf einem geometrischen Modell, bei dem der Abstand und der Neigungswinkel zwischen der Kamera und dem Boden bekannt sind. Daraus kann die Distanz des Kamerasensors von einem Objekt P vor dem Fahrzeug berechnet werden. Diese Zusammenhänge sind in Abbildung 16 dargestellt, wobei d die Distanz, h die Höhe der optischen Achse, also die Mitte der Kameralinse, α der Neigungswinkel der Kamera und f die Brennweite der Kameralinse sind. Die Koordinaten x und y bzw. x_0 und y_0 entsprechen den Koordinaten auf der Bildfläche. Des Weiteren sind x_0 und y_0 die Koordinaten des Schnittpunktes

zwischen der optischen Achse und dem aufgenommenen Bild. Die Werte x und y sind die Koordinaten des Objektes P im Bild. (vgl. [JLL04])

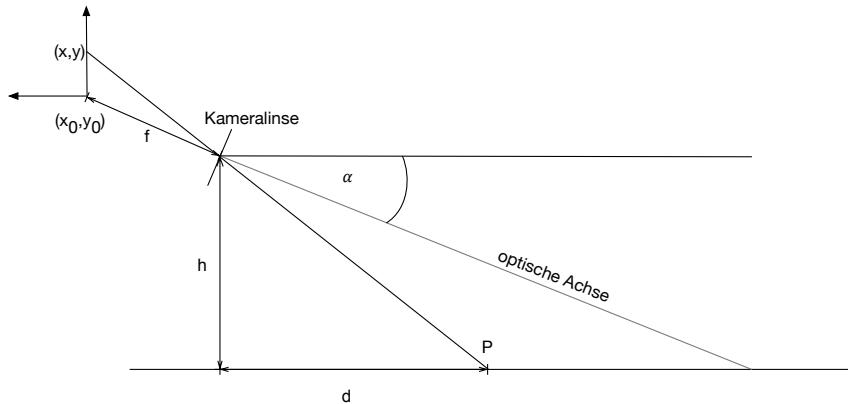


Abbildung 16: Geometrisches Modell zur optischen Abstandsmessung (vgl. [JLL04])

Aus diesem geometrischen Zusammenhang lässt sich die trigonometrische Formel $d = \frac{h}{\tan(\alpha + \arctan(\frac{y-y_0}{f}))}$ ableiten. Um die tatsächliche Distanz zu berechnen, muss auch die

Größe eines Sensorpixels bekannt sein. Sind (u, v) Pixel im digitalen Ergebnisbild und die oben genannten Koordinaten (x, y) die des Kameraensors, so ist $x = (u - u_0) \cdot d_x$ bzw. $y = (v - v_0) \cdot d_y$. Die Koordinaten (u_0, v_0) beschreiben die Ergebnisbildmitte und (d_x, d_y) die Pixeldimensionen.

Die Ergebnisse jeder digitalen photographischen Messmethode sind fehlerbehaftet. Grund dafür ist, dass durch die Kameralinse Verzerrungen im aufgenommenen Bild entstehen. Diese nehmen von der Sensormitte zu den Rändern hin zu. Eine softwareunterstützte Kamerakalibration kann dem entgegenwirken, die Verzerrung aber nicht gänzlich auslöschen. Je größer die zu messenden Abstände vor der Kamera sind, desto ungenauer werden auch die berechneten Ergebnisse. Auch sei hier erwähnt, dass nicht alle Kamerahersteller Informationen zur Brennweite oder Sensorgröße bereitstellen. Ergänzend zu dieser Messmethode empfiehlt es sich daher, zusätzliche Abstandsmessungen mit Ultraschallsensoren oder RADAR durchzuführen. (vgl. [JLL04])

2.3 Entwicklungswerkzeuge

Bevor die besprochenen Konzepte dieses Grundlagenkapitels praktisch umgesetzt werden, ist es sinnvoll, verfügbare Entwicklungswerkzeuge und -prozesse für diese Arbeit einzuführen. Einmal mehr sind die DARPA-Projekte hilfreich, einen Überblick dazu zu gewinnen. Inkrementelle, agile Entwicklungsprozesse und schnelle Prototypenerstellung (engl. *rapid prototyping*) werden in der Industrie und Forschung zunehmend angewendet. Anforderungen, aber auch Lösungswege, um diese zu erfüllen, ändern sich jeweils mit der Entwicklung eines innovativen Projekts. Neue Erkenntnisse und Probleme, die bei Tests entdeckt werden, sollen damit einfacher integriert werden, als in streng hierarchischen Prozessen, wie zum Beispiel beim Wasserfallmodell. Die besondere Herausforderung

besteht darin, dass Software zwar einfacher verändert werden kann, als Hardware, aber weitaus schneller in der Komplexität und im Umfang anwachsen kann. Abbildung 17 zeigt den grundlegenden Prozessablauf iterativer Methoden. Zuerst müssen Anforderungen an das System definiert werden. Diese lassen sich zum einen aus der Forschungsfrage, zum andern aus Annahmen definieren. Aus der Analyse dieser Anforderungen wird ein erstes Konzept entworfen. Dieses wird implementiert und schließlich das Ergebnis evaluiert. Basierend auf den Tests können nun Verbesserungen und Veränderungen als neue Anforderungen formuliert werden. Damit kann das Konzept wieder entsprechend angepasst werden und die nächsten Schritte werden erneut durchlaufen. Die Entwicklung ist dann abgeschlossen, wenn ein vorher definierter „Reifegrad“ erreicht wird. Dieser wird entweder vom Ziel bzw. von den Anforderungen des Projektes bestimmt oder von einem Kunden vorgegeben. (vgl. [BBR07], [BR12], [VGG10])

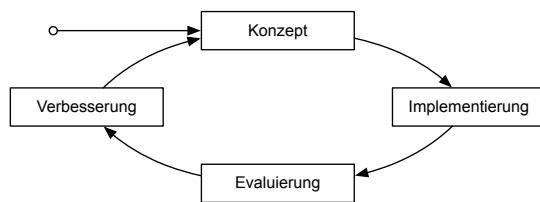


Abbildung 17: Iterativer Entwicklungsprozess

2.3.1 Modellierung

In der Konzeptionsphase eines Programms, muss dieses zunächst modelliert bzw. entworfen werden. Dabei ist zu ermitteln, welche Anforderungen an das System gestellt werden, welche Aufgaben es lösen und aus welchen Komponenten es bestehen soll. Dafür wurde von der *Object Management Group* (OMG) eine Entwurfssprache zur Visualisierung, Spezifizierung, Konstruktion und Dokumentation von Systemen standardisiert, die sowohl schriftliche, wie auch graphische Teile umfasst. Diese heißt *Unified Modeling Language* (UML), hat sich auch in der Softwareentwicklung etabliert und wird daher in dieser Arbeit eingesetzt. In dieser Sprache werden die Bezeichnungen gängiger Modellierungsbegriffe, wie auch die Beziehungen zwischen diesen definiert. In verschiedenen Diagrammtypen können unterschiedliche Aspekte und Perspektiven auf das System dargestellt werden. Ein Ziel von UML ist es, allen Interessensgruppen (engl. *stakeholder*), also Kunden, Projektleitern und Entwicklern, eine gemeinsam verständliche Sicht auf und Sprache für das System zu geben. Es gibt auch Werkzeuge für den Export von fertigen Modellen als Programmcode in verschiedene Programmiersprachen. Im Folgenden werden einige weitere Hilfsmittel zur System- und Softwareentwicklung besprochen, die für die obengenannten Prozessschritte in dieser Arbeit eingesetzt werden. (vgl. [OMG17])

2.3.2 Integrierte Softwareentwicklungsumgebung

Für das Erstellen eines Programmcodes gibt es Werkzeugsets, welche alle notwendigen Programme enthalten. Ein solches Set wird als integrierte Softwareentwicklungsumgebung (engl. *integrated development environment*, IDE) bezeichnet. Wesentliche Teile davon sind ein Texteditor, ein Compiler, ein Linker und oftmals auch ein Debugger. Zu dem Set gehören ebenfalls Werkzeuge zur Projektdateiverwaltung.

Es existieren bereits einige kommerzielle Entwicklungsprogramme für ADAS, die auch für die Realisierung eines vollständig autonomen Fahrzeugs verwendet werden können. Das von der EU geförderte Projekt *DESERVE* schlägt für ADAS *EB Assist ADTF* und *RTMaps* vor (vgl. [BK15]). Beide Programme verfügen über Funktionen zur ADAS-Applikationsentwicklung, sowie zur Analyse von Aufzeichnungen verschiedener Sensortypen (vgl. [R12], [EB18], [RTM18]). Eine kostenlose Evaluierungsversion der Programme wird ebenfalls angeboten. Diese vorgeschlagenen Programme sind weder quelloffen, noch frei zugänglich. Für diese Arbeit werden jedoch Werkzeuge benötigt, die diesen Anforderungen gerecht werden.

Je nach eingesetztem Betriebssystem, eignen sich verschiedene IDEs. Während für *Microsoft Windows*-Systeme *VisualStudio* und für *Apple macOS* *Xcode* von den Herstellern empfohlen werden, gibt es für *Linux* keine konkreten Programmempfehlungen. Quelloffene und frei zugängliche Lösungen für die drei genannten Systeme sind die Programme *Eclipse*, *Code::Blocks* oder *Geany*. Sie lassen sich durch Konfiguration und Erweiterungen an die Bedürfnisse der Entwicklung anpassen. Neben dem Texteditor zur Programmcodeerstellung gibt es Funktionen zur Projekt- sowie zur Bibliotheksverwaltung, zum Testen und zum Debugging des Codes. Auch können die IDE für die Kompilierung des Codes für andere Betriebssysteme eingestellt werden.

Ist keine graphische Benutzeroberfläche verfügbar, wie das bei einem textbasierten Fernzugriff auf ein *Linux*-System der Fall ist, so bleibt noch der klassische Entwicklungsweg über einen Texteditor, wie *nano* oder *vim*, sowie über einen Compiler und Linker, wie *GCC*. Ein populäres Werkzeug, das sich um die Softwarekomplilierung und Paketerstellung kümmert, ist *CMake*. Es ist auch möglich, auf einem Computer Software für ein anderes Computersystem zu entwickeln und es für dieses zu übersetzen. Das in dieser Arbeit eingesetzte Embedded-Board ist so leistungsfähig, dass es ein *Linux*-Betriebssystem samt graphischer Benutzeroberfläche ausführen kann. Der Programmcode kann daher direkt am *Raspberry Pi* entwickelt werden. Mit dem Betriebssystem dieses Embedded-Computers wird die IDE *Geany* mitgeliefert, welche ressourcenschonend arbeitet und sich daher für den Einsatz in dieser Arbeit eignet. (vgl. [GNU18b], [K18], [M18b])

2.3.3 Versionsverwaltung

Während des Entwicklungsprozesses ist es hilfreich, Zwischenzustände oder Alternativen des Programmcodes zu speichern. Die zwei populärsten Werkzeuge dafür sind *Subversion* und *Git*. Beide bieten die Möglichkeit der Versionsverwaltung an. Der größte Unterschied zwischen den beiden Werkzeugen ist die Datenverwaltung. *Subversion* bietet eine

zentralisierte Versionsverwaltung (engl. *concurrent versions system*, CVS) an, bei der ein Projekt an einer zentralen Stelle gelagert wird. Die Entwicklung kann von unterschiedlichen Personen durchgeführt und Änderungen an die zentrale Stelle geschickt werden (vgl. [CS14] S. 3, [ASF18]). *Git* wurde von *Linux*-Entwicklern erstellt, um von unterschiedlichen Orten aus gleichzeitig am Kernel des Betriebssystems zu arbeiten. Jede mitentwickelnde Person hat eine vollständige Kopie des Projekts auf ihrem Arbeitscomputer. Wird ein Programmteil fertig, so kann dieser an alle anderen verteilt werden. Dieses System wird deshalb auch als verteiltes Versionskontrollsystem (engl. *distributed version control system*, DVCS) bezeichnet. Der Vorteil ist, dass keine Abhängigkeit von einer zentralen Quelle gegeben ist. Verfügen Entwickler temporär über keine Internetverbindung, stehen ihnen die benötigten Ressourcen auch in einer lokalen Kopie zur Verfügung. (vgl. [CS14] S. 4, [SFC18])

Für beide Versionssysteme gibt es zahlreiche freie Onlineangebote zur zentralen Speicherung des Codes. Die Wahl ist in dieser Arbeit auf *Github* gefallen, da das Onlineservice kostenlose Accounts anbietet.

2.3.4 Softwarebibliothek

Eine umfangreiche quelloffene Softwarebibliothek für die digitale Bildverarbeitung ist *OpenCV*. Das Projekt wurde unter der Leitung der *Itseez* Organisation geführt und 2016 von *Intel* übernommen. *OpenCV* stellt Funktionen für das Maschinenlernen bereit, womit zum Beispiel die *Haar-Kaskadenklassifizierung* trainiert und zur Objekterkennung eingesetzt werden kann. Portierungen für die marktführenden Betriebssysteme *Microsoft Windows*, *Apple macOS* und *Linux* stehen zur Verfügung. Damit lässt sich *OpenCV* auch auf dem *Raspberry Pi* installieren. (vgl. [OCV18])

Es gibt einige ähnliche Bibliotheken, wie *VLFeat* oder *VXL*. Diese bieten ebenfalls Funktionen zur digitalen Bildverarbeitung sowie Erkennungsalgorithmen an. Im Vergleich zu *OpenCV* ist ihr Umfang deutlich geringer. Neben diesen Bibliotheken existieren zahlreiche Implementierungen, die ausschließlich spezialisierte Funktionen für das Maschinenlernen anbieten. In dieser Arbeit wird aufgrund des gebotenen Umfanges, der großen Unterstützung und der aktiven Weiterentwicklung *OpenCV* gewählt. (vgl. [VF07])

2.3.5 Evaluierung und Simulation

Als Evaluierungs- und Simulationsprogramm für Algorithmen und mathematische Modelle eignet sich das Mathematikprogramm *MATLAB* des Herstellers *Mathworks*. Über zusätzliche Toolbox-Erweiterungen stehen zahlreiche Funktionen zur digitalen Bildverarbeitung und -analyse zur Verfügung. Mittlerweile gibt es auch eine *Automated Driving Toolbox* mit Funktionen zur Fahrbahn- und Objekterkennung (vgl. [M18a]). Ein alternatives quelloffenes Programm ist *Octave*. Dafür stehen ebenfalls Softwarepakete zur Erweiterung der Funktionalität zur Verfügung. *Octave*-Skripts sind in großen Teilen zu

MATLAB kompatibel und umgekehrt (vgl. [GNU18a]). Es gibt allerdings weit weniger Werkzeugerweiterungen für *Octave*, als für *MATLAB*. Wie Berger et al. anmerken, sind diese Plattformen alleine nicht ausreichend um Softwarearchitekturen zu implementieren (vgl. [BR12] S. 10).

Gruyer et al. und Berger et al. haben für ihre Projekte Simulationsumgebungen entwickelt, um sowohl Software als auch Hardware zunächst ohne Risiko virtuell testen zu können. Das spart nicht nur Kosten in der Entwicklung der Fahrzeuge, sondern auch Zeit in der Erstellung unterschiedlicher Testszenarien. In beiden Projekten wird jedoch auch die Notwendigkeit des Tests des Fahrzeugs bzw. der Hardware unter realen Bedingungen unterstrichen. Diese Prozedur wird *Vehicle Hardware in the Loop* (VEHIL) bzw. *Hardware in the Loop* (HIL) genannt. Dieser HIL-Ansatz ist auch Teil der Motivation für die Entwicklung der vorliegenden Arbeit, da ein autonomes Modellfahrzeug ebenfalls eine kostengünstige HIL-Simulation ermöglicht. (vgl. [BR12], [VGG10], [GPG13])

Eine einfache HIL-Test- und -Simulationstechnik ist es, dem autonomen System statt eines Echtzeitkamerabildes, eine im Voraus erstellte Videoaufzeichnung zuzuführen. Diese lässt sich genauso als Folge von Einzelbildern laden und auf die gleiche Weise analysieren, wie die Echtzeitaufnahme. Der Vorteil dieser Methodik ist, dass die Reaktion des Systems auf Bildrauschen, reale Lichtverhältnisse oder einen eingeschränkten Dynamikbereich der Kamera beobachtet werden kann, ohne dass sich das System tatsächlich von der Stelle bewegen muss. In der vorliegenden Arbeit werden sowohl *Octave* zu Evaluierung von Algorithmen für digitale Bildverarbeitung, als auch vorher erstellte Videodaten für die Simulation dieser Algorithmen eingesetzt.

3. KONZEPT UND DESIGN

Die Entwicklungsumgebung dieser Arbeit besteht aus einer Auswahl der besprochenen Werkzeuge und Methoden, welche im vorhergehenden Kapitel vorgestellt wurden. Sie werden im Folgenden zum Entwurf eines Prototyps eingesetzt, welcher autonom einer markierten Fahrbahn folgt und Hindernisse erkennt. Dieses autonome Fahrzeugmodell dient dabei zwei Zielen. Zum einen werden die gewählten Werkzeuge anhand eines konkreten Beispiels, nämlich der Prototypentwicklung, auf ihre Eignung hin überprüft und angepasst. Zum anderen ist der Prototyp selbst ein wesentlicher Teil der Entwicklungsumgebung, mit dem wiederum neue Applikationen für das autonome Fahren entwickelt und evaluiert werden können. Das hier vorgeschlagene Systemkonzept ist allerdings nicht auf das verwendete spezifische Modellfahrzeug oder die verwendete Hardware beschränkt. Es lässt sich genauso mit anderen Komponenten realisieren, die über einen ähnlichen Funktionsumfang verfügen, wie die hier vorgeschlagenen.

Zu Beginn dieses Kapitels wird ein Systemdesign aus den besprochenen Grundlagen entworfen. Anschließend werden konkrete Hardwarekomponenten vorgestellt, mit welchen die genannten Aufgaben des autonomen Fahrens in dieser Arbeit umgesetzt werden. Der nächste Abschnitt definiert die Betriebsmodi des Systems und die Übergänge zwischen diesen Systemzuständen. Für den autonomen Fahrmodus wird ein Regelungsentwurf skizziert, anhand dessen wichtige Module identifiziert werden, welche in weiterer Folge in Kapitel 5 als Software implementiert werden. Ebenfalls für die softwaretechnische Umsetzung ist das Datendesign wesentlich. Darin werden die benötigten Informationen beschrieben und Klassen dafür beschrieben. Um den Softwarecode entwickeln und ausführen zu können, wird auch das Betriebssystem und die Installation benötigten Softwarewerkzeuge erklärt. Im letzten Abschnitt wird noch eine kurze Einführung zum Fern-Monitoring des Systems gegeben.

3.1 Systemdesign

Um die Aufgaben des autonomen Fahrens zu bewältigen, wird hier für den Prototyp eine Systemarchitektur vorgeschlagen, die sich an den Entwürfen der DARPA-Projekte sowie am Entwurf von Siegwart et al. orientiert und den in Kapitel 2.1 vorgestellten Schichten entspricht (vgl. z.B. [SNS11]). In Abbildung 18 ist das Systemdesign für den Prototyp graphisch dargestellt. Die oberste Schicht, die Benutzerschnittstelle, ist sowohl über ein am System angeschlossenes Display, als auch über einen Fernzugriff von einem PC aus erreichbar. Sie zeigt Informationen über den aktuellen Systemzustand, das Kamerabild, die erkannten Objekte sowie Fahrzeugdaten an und ermöglicht Benutzereingaben. Aus der Vielzahl der vorgestellten Sensorarten werden, wie in Kapitel 2.1.1 angemerkt, ein Ultraschallsensor für die Abstandsmessung und eine USB-Webkamera für die Objekterkennung gewählt. Diese zwei Messverfahren werden jeweils durch Softwarealgorithmen in der Sensorfusionsschicht durchgeführt. In der Planung wird, abhängig von den erarbeiteten Informationen der vorherigen Schicht, die Fahrtroute

bestimmt. Des Weiteren wird auch das Verhalten bei der Erkennung eines Hindernisses definiert. Eine zusätzliche Applikation ermöglicht statt der autonomen Routenplanung, die Fernsteuerung durch einen Benutzer durchzuführen. Die tatsächliche Lenkung und Beschleunigung wird von der Fahrzeugsteuerungsschicht durchgeführt und über entsprechende Funktionen an einen Motortreiber geschickt. Dieser generiert Steuerungssignale und kommuniziert sie an ein Modul, welches wiederum pulsweitenmodulierte Signale (PWM) erzeugt. Die Aktoren empfangen anschließend diese Signale. Der Servomotor stellt den Lenkwinkel des Fahrzeugs ein und eine elektronische Motorsteuerung (engl. *motor control unit*, MCU) die Umdrehungszahl des Bürsten- bzw. Antriebsmotors.

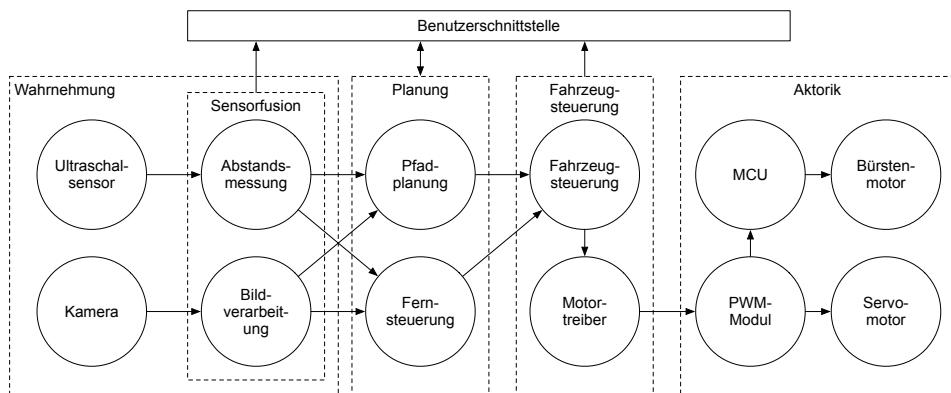


Abbildung 18: Systemdesign

3.2 Hardwarekomponenten

Die Sensorfusion, die Planung und die Fahrzeugsteuerung des autonomen Prototyps wird softwaretechnisch gelöst. Um diesen Programmcode auszuführen, bedarf es eines Computers, an den auch die besprochenen Sensoren und Aktoren angeschlossen werden können. Das Gerät soll nicht größer als das Trägerchassis des Modellautos sein, um es bei einer Kollision während des Betriebs vor Beschädigungen zu schützen. Für diese Arbeit werden *Raspberry Pi* Embedded-Boards gewählt. Die wesentlichen Vorteile hierbei sind die kleine Platinengröße, der geringe Anschaffungspreis sowie die weitverbreitete Unterstützung durch Soft- und Hardwaredritthersteller. Das *Raspberry Pi 3 Model B* besitzt einen ARM Cortex-A53 Prozessor mit vier 1,2 GHz getakteten Kernen und 1 GB RAM. Über vier USB- und *General Purpose Input/Output*-Anschlüsse (GPIO) können analoge und digitale Peripheriekomponenten mit dem Computer verbunden werden. Das System verfügt über keinen integrierten Sekundärsspeicher, aber über einen Micro-SD-Speicheranschluss. Für die vorliegende Arbeit ist eine Speicherkarte mit einer Größe von 16 GB ausreichend. In der ersten Entstehungsphase dieser Arbeit wurde auch das leistungsschwächere Modell, der *Raspberry Pi 2 Model B* verwendet. Dieses ist mit einem ARM Cortex-A7 900 MHz Vierkernprozessor ausgestattet. Auch die Taktrate der Grafikkarte und des Hauptspeichers sind hier etwas geringer. Da die implementierte Software mehr Rechenleistung erfordert,

empfiehlt sich jedoch das leistungsfähigere Modell zu wählen. (vgl. [RP18a], [RP18b], [RP18g])

Für die Benutzerinteraktion mit dem autonomen Fahrzeug wird ein 7 Zoll großes, kapazitives Touchscreen des *Raspberry Pi* Herstellers eingesetzt. Das Display wird an der Oberseite des Fahrzeugs befestigt, sodass die Erreichbarkeit desselben gewährleistet ist. Diese Position schützt es auch vor einfachen Schlägen, die bei einem Unfall vorkommen könnten. Während der Softwareentwicklung und solange das Modelfahrzeug steht, wird auch ein größerer handelsüblicher Monitor mit einem HDMI-Anschluss eingesetzt. Weiters können über die verfügbaren USB-Anschlüsse des *Raspberry Pi* eine Maus und Tastatur angeschlossen werden. (vgl. [RP18c])

Beim *Raspberry Pi 3 Model B* sind drahtlose Kommunikationsmöglichkeiten, WLAN und Bluetooth bereits auf der Platine installiert. Das Vorgängermodell benötigt ein zusätzliches WLAN-USB-Modul. Das hier gewählte Modul ist *Edimax EW-7811n*, welches wie der integrierte WLAN-Chip des *Raspberry Pi 3*, die Standards 802.11 b, g und n unterstützt. Mit theoretisch möglichen 150 Mb/s ist die Transferrate groß genug, um Bildinformationen in ausreichender Qualität zu übertragen, wie Messungen in Kapitel 6 beweisen. Die Übertragungsreichweite hängt stark von der Umgebung des Systems ab, beträgt aber in der Theorie in etwa 30 m. Die drahtlose Datenübertragung wird zur Aktualisierung des Betriebssystems und anderer installierter Software benötigt. Besonders wichtig ist, dass damit der Datenaustausch zur Überwachung des autonomen Fahrzeugs und die Fernsteuerung möglich ist. (vgl. [E18], [RP18d])

Die Abstandsmessung wird durch einen nach vorne gerichteten Ultraschallsensor mit der Modellbezeichnung *HC-SR04* vorgenommen. Damit lassen sich laut Datenblatt des Herstellers Distanzen zwischen 3 cm und 400 cm mit einer maximalen Abweichung von 0,3 mm messen. Dieser Sensor kann über vier Anschlussleitungen an die GPIO-Pins des *Raspberry Pi* angeschlossen werden.

Bilder werden über die USB-Kamera *Microsoft LifeCam HD-3000* in einer maximalen Auflösung von 1280x720 Pixel erfasst. Diese Bildgröße reicht für die Fahrbahn- und Objekterkennung aus. Die Kamera wird zentral in Fahrtrichtung an der Längsachse des Fahrzeugs mit einem leichten Neigungswinkel, nach unten zur Fahrbahn gerichtet, montiert.

Um das Modelfahrzeug zu bewegen, werden Befehle über ein PWM-Modul vom Embedded-Board an einen Servomotor zur Lenkung und an eine MCU zur Fortbewegung geschickt. Die Hauptkomponente des Moduls ist ein *NXP PCA9685* Chip, der über ein I²C-Bus¹³ angesprochen werden kann. Bis zu 16 Motoren lassen sich über Dreidrahtleitungen mit Erdung (engl. *ground*, GND), 5 V und PWM-Signal daran anschließen. Das Modelfahrzeug empfängt im Werkzustand, Steuersignale über ein Funkmodul. Dieses wird durch das PWM-Modul ersetzt.

Abbildung 19 zeigt alle beschriebenen Systemkomponenten im Überblick. Versorgt wird das System von zwei Akkumulatoren. Einer der beiden hat eine Kapazität von 4000 mAh und liefert eine Spannung von 5 V mit 2 A an den Computer. Der Zweite versorgt das Modelfahrzeug und wird im überraschenden Absatz genauer beschrieben.

¹³ Das *Inter-Integrated Circuit* (I²C) ist ein serieller Bus zur Datenübertragung.

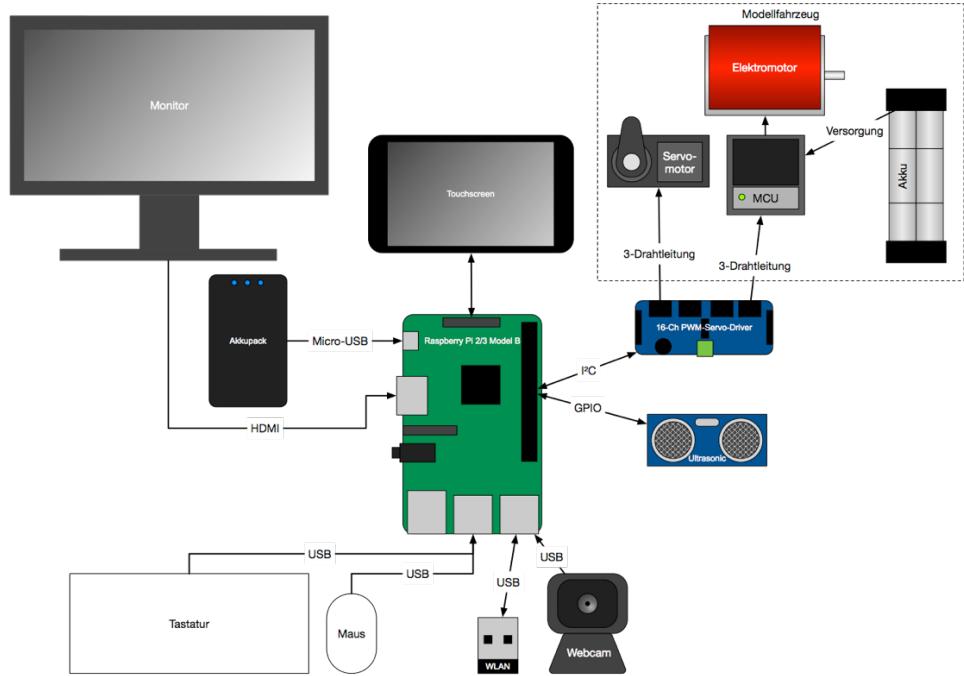


Abbildung 19: Komponenten für den Prototyp zum autonomen Fahren

Die fahrende Basis für den Prototyp ist ein Modellfahrzeug im Maßstab 1:10. Über vier Räder und zwei Achsen, die je ein Räderpaar verbinden, bewegt sich das Fahrzeug fort. Das in Fahrtrichtung vordere Räderpaar ist lenkbar. Ein Servomotor ermöglicht über eine Lenkvorrichtung die Bewegung der Achse. Der Lenkeinschlag kann in der entsprechenden Position gehalten oder dynamisch verändert werden. Zu beachten ist, dass, aufgrund von bautechnischen Gegebenheiten bei dem hier verwendeten Modell, der maximale Lenkeinschlag bei etwa 45° pro Richtung beschränkt ist. Neben der Erdung und 5 V wird über die dritte Leitung ein pulsweitenmoduliertes Signal (PWM) empfangen, welches im Werkszustand des Fahrzeugs von einem Funkmodul gesendet wird. Abhängig von diesem Signal wird die Achsdrehung des Motors und damit die Lenkung gesteuert.

Über einen weiteren bidirektionalen Antrieb, welcher aus einer MCU und einem elektrischen Bürstenmotor besteht, wird die Vor- und Rückwärtsbewegung des Fahrzeugs durchgeführt. Die MCU, oder im Modellbau auch *Electric Speed Controller* (ESC) genannt, ermöglicht eine stufenlose Drehzahlregelung des Motors. Zusätzlich versorgt sie das System mit Strom von einem Nickel-Metalhybrid-Akkumulator (NiMH-Akku), welcher 7,2 V und eine Kapazität von 4000 mAh liefert. Ein integrierter Spannungsteiler erzeugt 5 V, die für die Steuerungselektronik, also das *Radio Frequency Module* (RF) oder PWM-Modul, bestimmt sind. Der elektrische Bürstenmotor erreicht laut Hersteller bis zu 22.000 Umdrehungen pro Minute. Dieser treibt über Zahnräder ein Differenzial und damit die Hinterräder an. Eine Kardanwelle überträgt zusätzlich die Kraft von den Hinterrädern auf die Vorderräder, wodurch das Fahrzeug über vier angetriebene Räder verfügt.

Informationen zum Lenkwinkel und zur Geschwindigkeit werden beim Modellfahrzeug im Werkszustand über eine Fernbedienung zum Funkmodul gesendet. Die Übertragung findet im 2,4 GHz Funkkanal statt. Die erhaltenen Signale werden vom Modul in entsprechende Steuerungssignale konvertiert und an den Servomotor und die MCU geschickt. Um ein

autonom fahrendes Fahrzeug zu steuern, müssen diese Signale, statt von dem RF-Modul, vom eingebauten Computer generiert werden und entsprechend an die Motoren weitergeleitet werden. Aus diesem Grund wird hier statt des RF-Moduls ein PWM-Modul verwendet, welches vom Computer angesteuert werden und diese Signale erzeugen kann.

Abbildung 20 zeigt den schematischen Aufbau des Modellfahrzeugs im Werkzustand ohne Räder und Karosserie. Die Achsen des Fahrzeugs sind gefedert und gedämpft. An der Vorderseite des Fahrzeugs befindet sich eine Schaumstoffstoßstange, um Stöße abzufangen. Alle Komponenten sind auf einer Aluminiumplatte befestigt. Das Chassis, der Antriebsstrang und die Radaufhängungen ähneln denen eines Fahrzeugs im vollen Maßstab. Über zusätzliche Halterungen kann eine Kunststoffkarosserie befestigt werden. Diese bietet den inneren Bauteilen ebenfalls Schutz vor Kollisionsschäden. Eine detaillierte Abbildung und Erklärung aller im Zuge dieser Arbeit vorgenommenen Umbauten am Modellfahrzeug findet sich in der Realisierung in Kapitel 4.

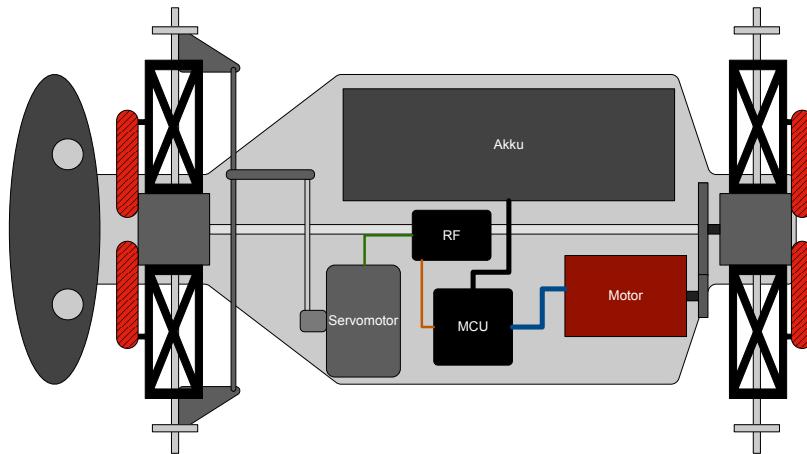


Abbildung 20: Aufbau des Modellfahrzeugs im Werkzustand

3.3 Systemzustände

Bevor der Prototyp autonom eine markierte Teststrecke befahren kann, müssen zunächst einige Systemzustände definiert werden, die das Überwachen, Ausführen und Fernsteuern ermöglichen. Das autonome System ist allerdings auch aus sicherheitskritischer Perspektive zu betrachten. Fällt ein Teil der Steuerung aus, so kann dies zu Unfällen und damit zu Personenschäden führen. Bei einem autonomen Fahrzeug im vollen Maßstab kann leider nicht ausgeschlossen werden, dass es im schlimmsten Fall zum Tod der Passagiere oder zu Umweltschäden kommen kann. Es dürfen daher keine unbekannten Zustände erreicht werden, auch wenn unerwartete Ereignisse, wie zum Beispiel der Ausfall der Kamera, auftreten. Aus diesem Grund müssen Fehlerfälle vom System abgefangen werden, wobei immer zu beachten ist, dass ein absolut sicheres System nicht existiert. Diese und viele weitere Anforderungen werden in ausführlicherer Form in der Norm IEC 61508 (vgl. [IEC10]) für die funktionale Sicherheit elektrischer, elektronischer oder programmierbarer elektronischer Systeme durch die *International Electrotechnical Commission* festgelegt.

Commission (IEC) definiert. Darin werden weitere Maßnahmen beschrieben, um Risiken, systematische oder zufällige Fehler schon beim Systementwurf zu beachten. Für den Automobilbereich gilt es auch, die Norm der *International Organisation for Standardization* (ISO) ISO 26262 (vgl. [ISO11]) zu beachten, welche die vorher genannte Norm für diesen Bereich genauer spezifiziert. Diese beiden Normen werden in der vorliegenden Arbeit nicht über die oben genannten Anforderungen hinaus implementiert. Für die spätere Weiterentwicklung des Projekts sind sie dennoch von Interesse.

Abbildung 21 stellt ein Zustandsdiagramm des Programms zum autonomen Fahren in dieser Arbeit dar. Wird das System gestartet, so nimmt es zuerst den Zustand, im Folgenden auch als Modus bezeichnet, *Initial* ein. In diesem werden vorher gespeicherte Einstellungen geladen. Da diese Einstellungen vom Benutzer manipuliert werden dürfen, müssen sie zuerst auf ihre Korrektheit verifiziert werden. Dies ist durch den Vergleich mit definierten Grenzwerten möglich, welche entweder vom Hersteller oder durch Versuche in dieser Arbeit bestimmt wurden. Der Verifikationsprozess wird in Kapitel 5.2 genauer beschrieben. Sind die Einstellungen korrekt, so kann das System ausgeführt werden. Tritt bereits an dieser Stelle ein Problem auf, wird das System sofort beendet. Damit soll sichergestellt werden, dass es zu keinem undefinierten und somit potentiell riskanten Systemverhalten kommt.

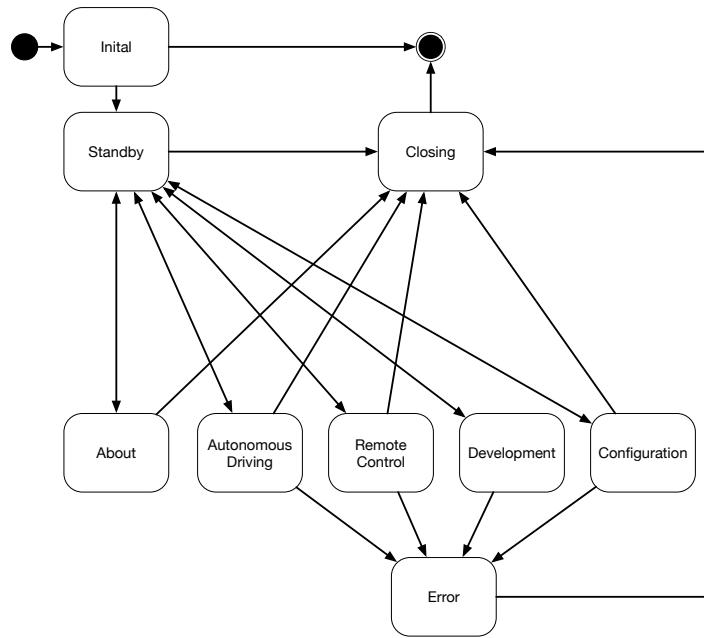


Abbildung 21: Zustandsdiagramm des Programms zum autonomen Fahren

Nach dem Start wird der Zustand *Standby* eingenommen. Hier wird dem Benutzer über ein Display ein Echtzeitbild der Kamera und das Hauptmenü angezeigt. Über dieses besteht die Möglichkeit in fünf weitere Betriebsmodi zur wechseln. In jedem dieser Zustände gibt es die Möglichkeit das System zu beenden, wodurch in den Zustand *Closing* gewechselt wird. Das entspricht der Funktion eines Not-Stopps, bei dem alle laufenden Prozesse kontrolliert heruntergefahren werden und das Fahrzeug zum Stillstand kommt. Für diese Arbeit wird angenommen, dass das angehaltene Modelfahrzeug zugleich der sicherste Zustand des Systems ist, weil damit keine Person zu Schaden kommen kann. In einem Fahrzeug im

vollen Maßstab sind allerdings wesentlich mehr Bedingungen zu beachten, was jedoch über den Rahmen dieser Arbeit hinausgeht. Genannt sei hier zum Beispiel, dass das Halten im Kreuzungsbereich oder auf einem Bahnübergang eindeutig nicht sicher ist.

Wie die Bezeichnung des Zustandes *Configuration* verrät, wird hier die Einstellung wichtiger Parameter des Systems vorgenommen. Dazu gehört die intrinsische und extrinsische Kalibrierung der Kamera sowie die Einstellung von Bildparametern. Beim Start des Konfigurationsmodus wird überprüft, ob eine externe Konfigurationsdatei, welche Voreinstellungen enthält, existiert oder ob Standardwerte geladen werden sollen. Auf Benutzerwunsch können Veränderungen der Systemkonfigurationen ebenfalls in dieser Datei abgespeichert werden. Wurden Einstellungen falsch vorgenommen oder gibt es mit den bestehen Probleme, werden alle Einstellungen auf Standardwerte zurückgesetzt, um ein Fehlverhalten des Systems zu verhindern. Dafür sorgt eine Validierungsfunktion. Aus dem Zustand kann zurück in *Standby* gewechselt werden.

Der Zustand *Autonomous Driving* startet den autonomen Fahrbetrieb, indem das System zunächst die Fahrbahn und Verkehrsschilder erkennt. Wird eine Fahrbahn gefunden, beginnt das Fahrzeug anschließend die Fahrt entlang derselben. Über eine Ausgabe an einem Bildschirm ist für den Benutzer sichtbar, welche Daten wie verarbeitet werden. Auch Informationen über das Fahrzeug, wie die Geschwindigkeit und der Lenkwinkel, werden angezeigt. Endet die Fahrbahn oder wird diese aufgrund eines Detektionsfehlers verlassen, wird das Fahrzeug umgehend angehalten. Vor einer erkannten Stopptafel wird der Fahrvorgang, wie gesetzlich vorgeschrieben, ebenfalls gestoppt. Auch Hindernisse, die sich vor dem Fahrzeug befinden, verursachen dasselbe Verhalten. Verlassen wird der Zustand immer so, dass das Fahrzeug zuerst zum Stillstand kommt. Aus dem autonomen Modus kann jederzeit wieder ins *Standby* gewechselt oder, wie weiter oben erwähnt, das System beendet werden.

Im *Remote Control*-Modus wird das Fahrzeug über Tastatureingaben ferngesteuert. Dieser Zustand ist dann notwendig, wenn das Fahrzeug sich nicht in der Nähe der bedienenden Person befindet und es zurück auf die Fahrbahn gelenkt werden soll. Die Tasten *W*, *A*, *S*, *D* bilden eine Form von Steuerkreuz und eignen sich daher für die Steuerung. Diese Tasten liegen auf allen verfügbaren Tastaturen an jeweils derselben Position. Für das Vorwärtsbeschleunigen muss die Taste *W* gedrückt werden. Zur Reduktion der Geschwindigkeit und zum Rückwärtsfahren ist die Taste *S* zuständig. Das Links- bzw. das Rechtslenken ist über *A* bzw. *D* möglich. Weiters dient die Leertaste als Bremse bzw. als Not-Stopp und bringt das Fahrzeug aus jeder Fahrtrichtung oder Geschwindigkeit zum Stillstand. In diesem Zustand ist eine Übertragung der Bildinformation von der Kamera am Fahrzeug zum steuernden Computer ebenfalls sinnvoll, da sich das Fahrzeug auch außerhalb des Sichtbereichs der bedienenden Person befinden kann. Ein Wechsel in die Zustände *Standby* und *Autonomous Driving* ist möglich.

Während der Evaluation neuer Objekterkennungsverfahren für das autonome Fahren wird besonders die Sensorik des Systems benötigt. In frühen Entwicklungsphasen sollen aus naheliegenden Sicherheitsgründen noch keine Steuerungsbefehle automatisch an die Motoren geschickt werden. Um Bildverarbeitungsalgorithmen dennoch testen zu können, ist der Entwicklungsmodus *Development* integriert. Im Prinzip handelt es sich dabei um eine Kombination der Erkennungsverfahren aus dem autonomen Fahrbetrieb und der

Fernsteuerung. Über ein Menü können die verfügbaren Verfahren einzeln aktiviert oder deaktiviert werden. Die Tastatur kann von einer bedienenden Person wieder zur physischen Fahrzeugsteuerung verwendet werden. Wie erwähnt, dient dieser Zustand in erster Linie zum Testen neuer Funktionen und erlaubt nur den Wechsel zurück zu *Standby*.

In einem Informationszustand *About* werden allgemeine aktuelle Informationen zum Projekt angezeigt, wie der Versionsstand, die Projektbezeichnung und der Autor samt Kontaktmöglichkeit. Dieser Zustand dient nicht unmittelbar dem Ziel des autonomen Fahrzeugs, soll allerdings einen schnellen Zugriff auf relevante Systeminformationen für die Softwareentwicklung bieten. Auch aus diesem Modus kann wieder in den *Standby*-Modus gewechselt werden.

Ein besonderer Systemzustand ist *Error*. Grundsätzlich sollen möglichst alle Fehler im System bereits in den jeweiligen Softwaresubroutinen, in denen sie auftreten können, abgefangen und kompensiert werden. Im Normalfall müsste das System also nie in den Fehlerzustand gelangen. Ist das sofortige Abfangen jedoch nicht möglich, wird ein Wechsel initiiert. Bevor allerdings in den Fehlermodus gewechselt wird, werden alle laufenden Prozesse so weit möglich beendet und eine entsprechende Fehlermeldung ausgegeben. Der *Error*-Modus initiiert schließlich einen Wechsel in den Beendigungsmodus, wo das System sicher heruntergefahren wird. Ebenfalls wichtig ist, dass das Fahrzeug den sichersten Zustand erreicht, also stehen bleibt.

Der Beendigungszustand *Closing* dient dazu, das System sicher abzuschalten. Wurde dieser erreicht, so wurden bereits im vorherigen Zustand alle dazugehörigen Abläufe beendet. Zur Absicherung wird der Motortreiber zurückgesetzt und die Motordrehzahl auf null gestellt. In jedem Fall soll in diesem Zustand der Prototyp zum Stillstand gebracht werden, bevor das System beendet wird.

Für jeden der beschriebenen Betriebsmodi steht eine dafür konzipierte graphische Benutzerschnittstelle bereit. Über die angeschlossenen Bildschirme werden Kamerabilddaten und deren Verarbeitung angezeigt, sowie Menüs zur Systemmanipulation. Status- und Debugging-Informationen werden nicht nur graphisch, sondern auch textuell über die Konsole ausgegeben. Benutzereingaben werden ausschließlich von der Tastatur angenommen.

3.4 Regelungsentwurf

Das Systemdesign, welches in Kapitel 3.1 eingeführt wurde, lässt sich auch als Closed-Loop- bzw. Feedback-Regelungsprinzip verstehen. Die Schleife entsteht dadurch, dass das System durch die Verhaltensplanung auf die Wahrnehmung der Umwelt reagiert und damit wiederum die Wahrnehmung beeinflusst. Genauer stellt Abbildung 22 diesen Ablauf für den autonomen Fahrbetrieb dar. Die Regelungsschritte entsprechen dabei den Elementen des Systemdesigns in Abbildung 18. Zuerst wird mittels der Kamera und des Ultraschallsensors die Umwelt aufgenommen. In entsprechenden Verarbeitungsroutinen werden die Fahrbahn, Verkehrsschilder und die Abstände zu Hindernissen ermittelt. Das Ergebnis wird zur Planung weitergeleitet, in der, wie bereits besprochen, das Verhalten des Systems

bestimmt wird. Abhängig von diesem, wird der Lenkungsmotor und die Motordrehzahl für den Vortrieb eingestellt und dadurch in Bewegung gesetzt. Das Modellfahrzeug beginnt die Fahrt, womit sich auch aus dessen Sicht die Umwelt, insbesondere die Distanzen zu ihr, verändern. Die Sensormessung muss erneut durchgeführt und es muss entsprechend reagiert werden. Dieser Ablauf wiederholt sich solange, bis die Schleife entweder vom Benutzer beendet oder durch einen Fehler unterbrochen wird.

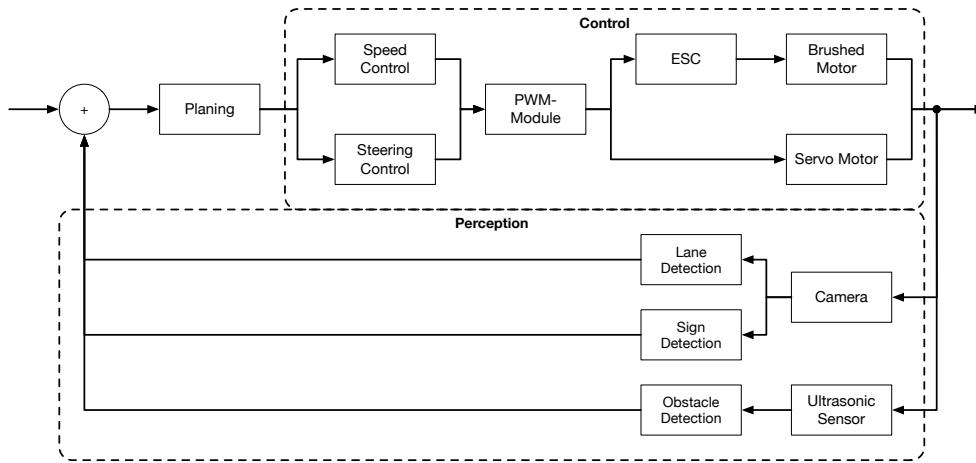


Abbildung 22: Regelungsentwurf für das autonome Fahren

Jeder Regelungsschritt benötigt eine gewisse Zeit um durchlaufen zu werden. Das hängt von mehreren Faktoren ab. Einfluss darauf haben die Prozessorart, also ob es ein DSP, eine CPU oder eine GPU ist, und dessen Taktrate. Mitentscheidend ist auch der eingesetzte Programmalgorithmus und damit verbunden die Zahl der Rechenschritte. Je komplexer die Regelung, desto schwieriger kann abgeschätzt werden, wie lange ein Durchlauf dauert. Wird die Regelung als Softwareprogramm mit einem herkömmlichen Betriebssystem ausgeführt, kann überhaupt nicht garantiert werden, ob und wann sie ausgeführt wird. Grund dafür ist, dass Betriebssysteme für gewöhnlich nicht deterministisch arbeiten, wie in Kapitel 3.6 weiter erläutert wird.

Ziel ist es, trotz der genannten Faktoren die Antwortzeit des Systems möglichst gering zu halten. Gemeinhin wird beim Menschen mit einer Reaktionszeit von etwa einer Sekunde gerechnet. In Gefahrensituationen liegt der Wert deutlich darunter (vgl. [NN73]). Das hier entworfene System muss ebenfalls eine Reaktionszeit unter einer Sekunde erreichen, um einen Sicherheitsvorteil für den Menschen zu erreichen. Ein möglicher Ansatz die tatsächliche Reaktionszeit zu bestimmen, ist es, die Zeitspanne zwischen der Bildaufnahme und der Aktorensteuerung zu messen. Diese Methode wird zur Evaluierung der Ergebnisse in Kapitel 6.3 angewendet.

Eine strikt sequenzielle Abarbeitung des Regelungsverlaufs addiert die Durchlaufzeiten der einzelnen Schritte. Aktuelle Prozessoren verfügen über mehrere Rechenkerne. Das gilt auch für die CPU der *Raspberry Pi Modelle 2 und 3*. Diese Eigenschaft ermöglicht es, Softwareprogramme gleichzeitig auszuführen. Darüber hinaus kann auch ein einzelnes Programm so implementiert werden, dass Teile davon parallel ablaufen können. Diese Technik wird als *Multithreading* bezeichnet, wobei ein Thread ein Teil eines Prozesses, also eines laufenden Programms ist. Eine Besonderheit bei der Threadprogrammierung ist, dass

der Speicherbereich von Threads desselben Prozesses gemeinsam verwendet wird. Deshalb muss darauf geachtet werden, dass gemeinsam genutzte Daten in diesem Bereich nicht gleichzeitig von mehreren Threads bearbeitet werden können, um deren Konsistenz zu gewährleisten.

Wie in Abbildung 22 dargestellt, sind die beiden Erkennungsverfahren unabhängig voneinander. Auch innerhalb der Verfahren können die Kameraaufnahme, die Bildmanipulation und die Anzeige auf einem Bildschirm gleichzeitig ausgeführt werden. Messungen in Kapitel 6.3 bestätigen eine schnellere Systemreaktion im Vergleich zur sequenziellen Verarbeitung. Tatsächlich können die gesamte Wahrnehmung, die Planung und die Fahrzeugsteuerung parallel verlaufen. Während die Planungsschicht eine Trajektorie aus den Daten der Wahrnehmung kalkuliert, kann bereits die nächste Aufnahme der Umwelt durchgeführt und verarbeitet werden. Genauso kann die Fahrzeugsteuerung die Motorstellwerte bestimmen, während die Planung gleichzeitig die Trajektorie an die nächste Messung anpasst. Die Reihenfolge der Abarbeitung ist unwichtig, solange die Ergebnisdaten der einzelnen Ebenen konsistent sind. Dies kann über einen synchronisierten Zugriff gewährleistet werden, wie im nächsten Kapitel zum Datendesign besprochen wird.

3.5 Datendesign

Die Verbindungen zwischen den einzelnen Regelungsschritten in den Diagrammen in Abbildung 21 und Abbildung 22 deuten Kommunikationsflüsse an. In diesen werden Informationen in Form von Daten übertragen. Sie enthalten erfasste Bilder, Abstände und Systemzustände. Ohne weitere Maßnahmen laufen diese Schritte in einem Multi-Thread-Programm jedoch asynchron. Das bedeutet, dass die Daten nicht zwangsläufig gleichzeitig in einer gültigen Form zur Verfügung stehen. Wird zum Beispiel ein Bild von einem Thread aufgenommen und im selben Moment von einem anderen Thread angefordert, bevor die Aufnahme fertig ist, so kann keine sinnvolle Objekterkennung durchgeführt werden. Das Bild wäre unvollständig. Aus diesem Grund ist bereits in der Entwurfsphase der Software, ein besonderes Augenmerk auf die Datenkapselung und -synchronisierung zu legen. Dies wird so sichergestellt, dass zuerst die Daten als Klassen definiert werden, deren Attribute nicht direkt gelesen oder bearbeitet werden dürfen. Der Zugriff auf diese Eigenschaften ist nur über dezidierte Auslese- und Schreibfunktionen möglich, welche zu jedem Zeitpunkt jeweils nur einem Thread den Zugriff gewähren. Ein solcher wechselseitiger Ausschluss kann durch *Mutex-Locks* (engl. *mutual exclusion*) sichergestellt werden. Zusammengefasst funktionieren diese so, dass sie über eine Variable anzeigen, ob eine Ressource belegt ist oder nicht. Diese Technik ist ein wichtiger Bestandteil in einigen der folgenden Datenklassen. Im Folgenden werden die Datenklassen für den Systemzustand, die Bilddaten, die Fahrbahn-, die Verkehrsschild- sowie die Hinderniserkennung, die Trajektoriendaten, die Fahrzeugdaten, die Daten der graphische Benutzerschnittstelle und die Konfigurationsdaten besprochen.

3.5.1 Systemzustandsdaten

Um die beschriebenen Systemzustände aus Kapitel 3.3 softwaretechnisch abzubilden, eignet sich das so genannte *State Pattern*. Dieses definiert einen endlichen Zustandsautomaten, der jeweils einen von mehreren bestimmten Zuständen annehmen kann. Ein Übergang zwischen den Zuständen wird im Allgemeinen durch externe oder interne Ereignisse ausgelöst. In dieser Arbeit wird der Automat an sich als aktueller Systemzustand, mit der naheliegenden Bezeichnung *SystemState* definiert. Dieser enthält den derzeit aktiven Betriebsmodus *SystemMode*. Eine weitere binäre Variable *running* zweigt an, ob das System ausgeführt wird oder nicht. Alle Attribute sind im Sinne der Datenkapselung privat und deren Zugriff daher nur innerhalb der Klasse möglich. Die Funktionen *setMode()* und *getMode()* erlauben das Festlegen bzw. Auslesen des aktuellen Modus. Im Klassendiagramm in Abbildung 23 wird der Zustandsaufbau genauer verdeutlicht. Der *SystemMode* ist als abstrakte Schnittstelle definiert. Er wird von den jeweiligen konkreten Klassen, wie zum Beispiel *StandbyMode* oder *AutonomousMode*, implementiert. Damit ist sichergestellt, dass zukünftig neue Modi hinzugefügt werden können und das System erweitert werden kann. Wie die Aggregationsbeziehung in Abbildung 23 darstellt, ist im Systemzustand immer genau ein definierter Modus bestimmt. Das entspricht der in Kapitel 3.3 genannten Forderung eines sicheren Systems. Die Implementierung der Klassen und Funktionen der Systemmodi werden in Kapitel 5.8 genauer erläutert.

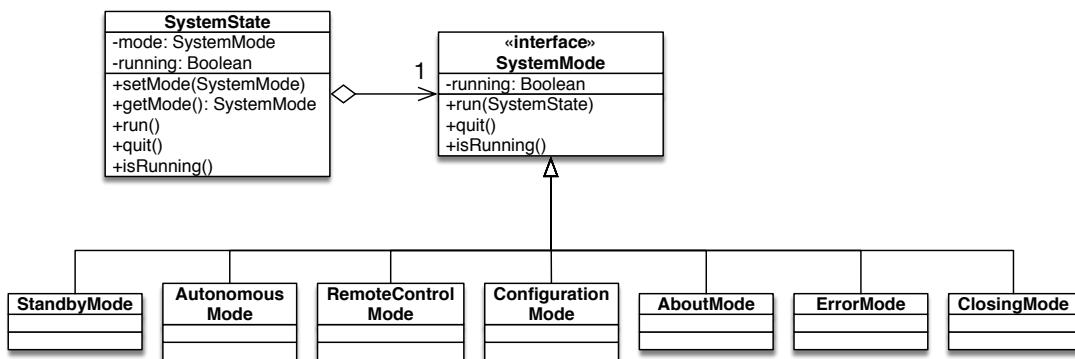


Abbildung 23: Klassendiagramm für den Systemzustand

3.5.2 Bilddaten

Eine weitere, besonders wichtige Datenklasse sind Kamerabilder. Diese werden als dreidimensionale Matrix gespeichert. Jede Dimension steht für einen der drei Farbkanäle blau, grün und rot. Diese Farbreihenfolge entspricht der Bildmatrizendefinition der Softwarebibliothek *OpenCV*. Ein zusätzliches Datenfeld *timeStamp* speichert den Aufnahmezeitpunkt. Diese Information ist insbesondere für die Kalkulation der verarbeiteten Bilder pro Sekunde relevant, wie weiter in Kapitel 6.3 beschrieben wird. Auch dient sie zur Feststellung, ob ein neues Bild aufgenommen wurde oder nicht. Damit bleibt dem System ein rechenintensiverer Bildmatrizenvergleich erspart. Den Zugriff auf die Daten regelt ein Mutex-Lock. Wieder sind alle Klassenattribute privat und deshalb nur innerhalb der Klasse *ImageData* sichtbar. Über die Funktionen *read()* und *write()* sind synchronisierte

Datenzugriffe, wie im Flussdiagramm in Abbildung 24 dargestellt, möglich. Werden die Funktionen aufgerufen, so wird zuerst versucht den Mutex-Lock zu bekommen. Ist er in diesem Moment frei, so wird der Zugriff für andere gesperrt und der nächste Schritt durchgeführt, also eine Variable gesetzt oder gelesen. Anschließend wird der Lock wieder freigegeben und die Funktion beendet.

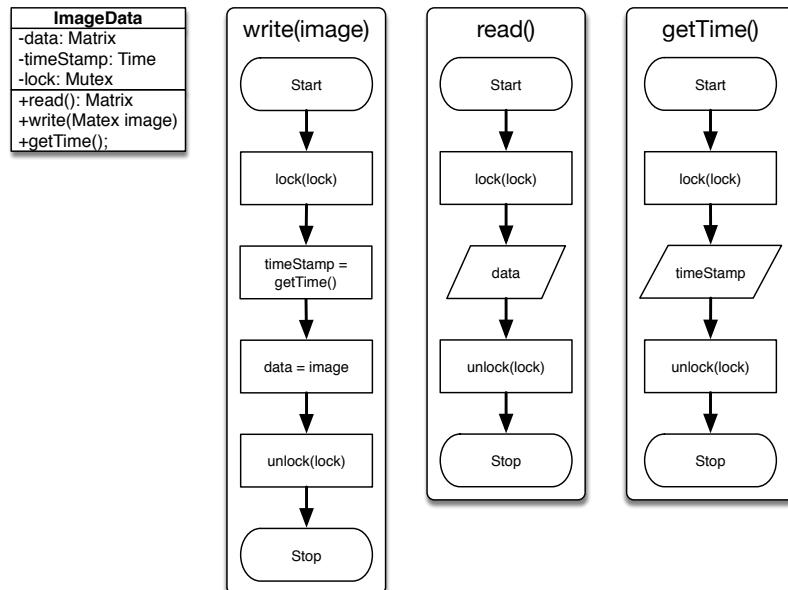


Abbildung 24: Bilddatenklassendiagramm und Flussdiagramm der Zugriffsfunktionen

3.5.3 Fahrbahnerkennungsdaten

Ähnlich wird der Zugriff auf die Daten aus den Erkennungsverfahren gesichert, auch wenn die Datenstruktur umfangreicher ist. Das Modell der Fahrbahn im Grundlagenkapitel 2.2.1 beinhaltet Markierungstypen wie in Abbildung 5 sowie das gänzliche Fehlen von Markierungen. Sie können demnach aus unterbrochenen, durchgezogenen oder gar keinen Linien bestehen. Eine Straßenmarkierung kann gerade oder gekrümmmt sein. Mathematisch könnte sie mit einem Polynom beschrieben werden. Für diese Arbeit reicht es aber, ihren Verlauf anhand von Punkten zu beschreiben. Werden die Punkte miteinander verbunden, entsteht graphisch eine Polygonlinie, also ein Verlauf zusammenhängender Geraden, wodurch ein Straßenmarkierungsverlauf approximiert werden kann. In Abbildung 25 werden beispielhaft im rechten Bildteil drei Polygonlinien für die drei Fahrbahnmarkierungen im linken Bildteil dargestellt.

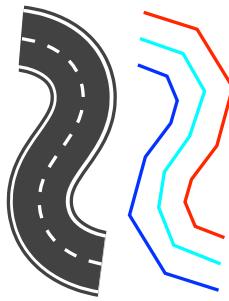


Abbildung 25: Fahrbahnsymbol und dazugehörige Polygonlinien

Im Klassendiagramm in Abbildung 26 beschreibt der Enumerator des Markierungstyps die Varianten, welche in Abbildung 5 dargestellt sind, mit der oben genannten Erweiterung. Eine Straßenmarkierung wird hier im System als Vektor mit den Bildkoordinaten des Verlaufs und des Markierungstyps definiert. Der Zugriff auf diese Attribute muss nicht direkt geschützt werden, weil diese Klasse eine allgemeine Beschreibung darstellt. Erst die aktuelle Fahrbahninformation in *LaneData* beinhaltet die erkannte linke sowie rechte Markierung. Der Zugriff dazu muss hingegen geschützt werden.

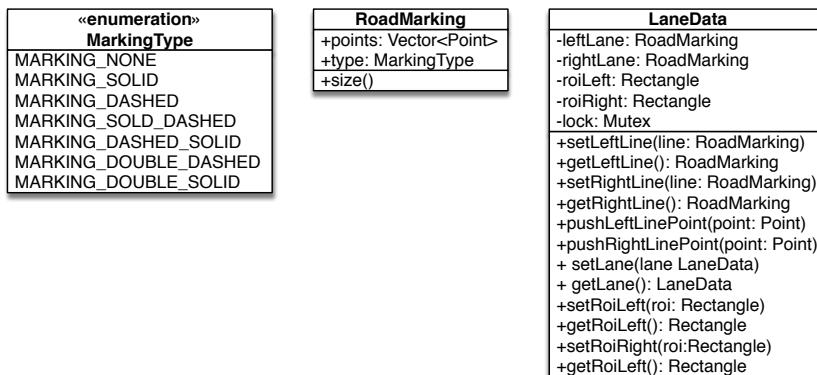


Abbildung 26: Klassendiagramm für die Fahrbahnerkennung

3.5.4 Verkehrsschilderkennungsdaten

Eine weitere Datenstruktur speichert Informationen über die erkannten Verkehrsschilder. Das Klassendiagramm dazu ist in Abbildung 27 zu sehen. Über einen Enumerator wird der Typ des Verkehrsschildes festgehalten. Für diese Arbeit ist ausschließlich das Stopperschild relevant, weil nur die Erkennung dieses Zeichens beispielhaft implementiert wird. In der Klasse *TrafficSign* wird neben dem Typ auch die Position der Tafel im Bild mithilfe eines Rechtecks beschrieben. Ein Datenfeld für die Distanz zwischen dem Schild und der Fahrzeugstoßstange und eines für die relative Geschwindigkeit zum Schild wird ebenfalls definiert. Eine weitere Klasse *TrafficSignData* speichert einen Vektor mit allen erkannten Verkehrsschildern. Auch hier ist ein Mutex-Lock für den synchronisierten Zugriff notwendig. Die Klassenoperatoren entsprechen den Funktionen für die Vektoren der Programmiersprachenbibliothek.

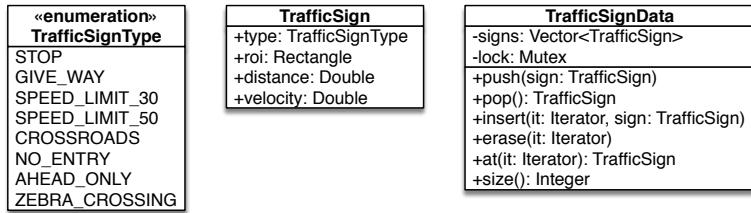


Abbildung 27: Klassendiagramm für die Verkehrsschilderkennung

3.5.5 Hinderniserkennungsdaten

Das Hinderniserkennungsverfahren benötigt ebenfalls eine Datenklasse, welche im Klassendiagramm in Abbildung 28 dargestellt ist. Sie ist der Verkehrsschilderkennung ähnlich und wird *ObstacleData* genannt. Die Datenklasse beinhaltet einen Vektor von erkannten Hindernissen, welche wiederum in der Klasse *Obstacle* definiert sind. Darin werden Informationen über den Abstand zwischen Fahrzeugstoßstange und Hindernis, die relative Geschwindigkeit des Hindernisses und dessen Position im Bild gespeichert. Die Attribute relative Geschwindigkeit *velocity* und Position *roi* werden hier nicht verwendet und sind als Erweiterungsmöglichkeit für ein zukünftiges, visuelles Hinderniserkennungsverfahren zu verstehen.

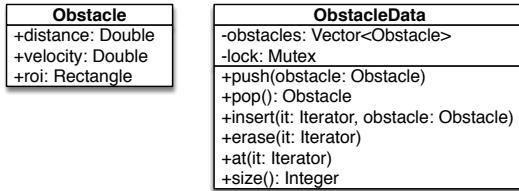


Abbildung 28: Klassendiagramm für die Hinderniserkennung

3.5.6 Trajektoriendaten

Die Planungsschicht greift auf alle Daten der Erkennungsverfahren zu, die in Abbildung 29 dargestellt sind, und erzeugt daraus eine Trajektorie. Diese wird in der Datenklasse *TrajectoryData* gespeichert. Sie besteht aus einem Punktevektor, wobei jeder Punkt von zwei Koordinaten definiert wird. Wieder sichert ein Mutex-Lock den Zugriff auf die Daten. Die Zugriffsfunktionen entsprechen denen für Vektoren, wobei immer zuerst der Lock angefordert und freigegeben wird. Typische Vektoroperationen sind das Hinzufügen, das Entfernen und das Auslesen von Elementen.

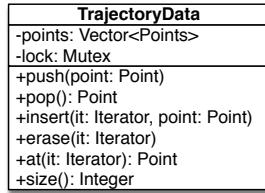


Abbildung 29: Klassendiagramm für die Trajektoriendaten

3.5.7 Fahrzeugdaten

Für die Fortbewegung des Prototyps muss der aktuelle interne Status des Fahrzeugs erfasst und manipuliert werden. Die Klassen im Diagramm in Abbildung 30 beschreiben ein erweiterbares Modell wichtiger Fahrzeugattribute. Eine Datenstruktur *VehiceData* besteht aus den Werten für die Beschleunigung, die Geschwindigkeit, den Gang, die Lenkrichtung und den Lenkwinkel des Fahrzeugs. Um Letzteres korrekt zu berechnen, sind auch Informationen über die Außenmaße notwendig. Sie werden mit der Klasse *VehicleDimensions* erfasst. Die Grenzwerte des Fahrzeugs, wie die maximale Geschwindigkeit oder der maximale Lenkwinkel, werden von den *VehicleLimits* beschrieben. Die zwei Enumeratoren *VehicleDirection* und *VehicleGear* definieren, wie ihre Bezeichnung verrät, Werte für die Lenkrichtung und den Getriebegang. Für diese Arbeit ist nur der Vorwärts-, der Neutral- und der Rückwärtsgang notwendig, da beim verwendeten Modellauto keine weiteren Gänge existieren. Sämtliche Daten aus der Klasse *VehicleData* können ausschließlich mittels zugriffsgeschützten Funktionen ausgelesen oder manipuliert werden.

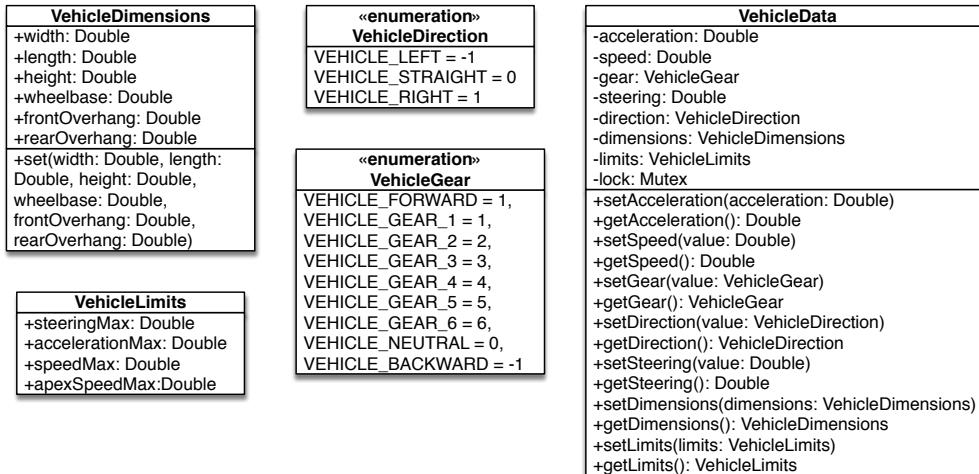


Abbildung 30: Klassendiagramm für die Fahrzeugdaten

3.5.8 Daten für die graphische Benutzerschnittstelle

Über eine graphische Benutzerschnittstelle werden aktuelle Systeminformationen und das Echzeit-Kamerabild ausgegeben. Die Schnittstelle ist abhängig vom aktuellen Systemzustand. Es bietet sich hier ebenfalls an, ein *State Pattern*, wie in Abbildung 31 dargestellt, anzuwenden. Der *UserInterfaceState* beinhaltet neben dem aktuellen Modus, ein alphanumerisches Zeichen und einen Lock. Das Zeichen beschreibt die letzte vom Benutzer eingegebene Taste. Ist sie einem bestimmten Befehl zugeordnet, wird ein Zustandswechsel initiiert.

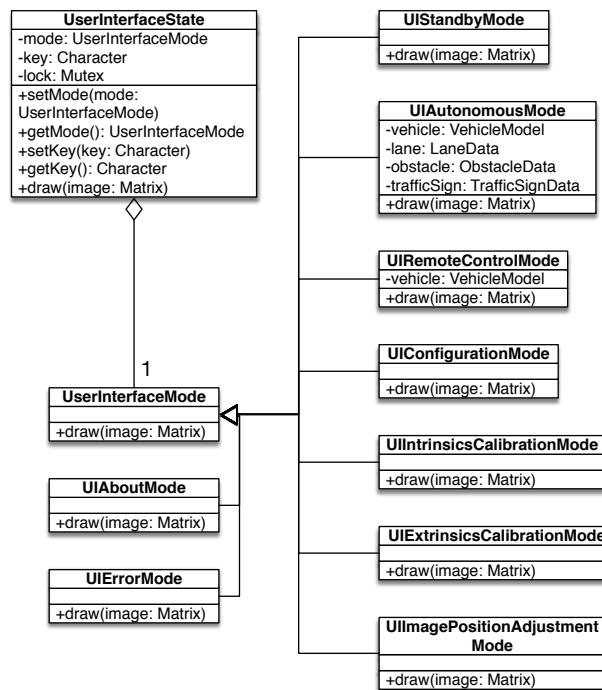


Abbildung 31: Klassendiagramm für den Zustand der graphischen Benutzerschnittstelle

3.5.9 Konfigurationsdaten

Alle bisher genannten Datenklassen dienen der internen Systemkommunikation und Statusspeicherung während der Laufzeit des Programms. Wird das System jedoch abgeschaltet, so gehen alle diese Informationen verloren. Es ist allerdings hilfreich, bestimmte Daten, wie zum Beispiel jene für die Kamerakalibration, so zu persistieren, dass sie nach einem Systemneustart wieder verfügbar sind. Neben der Möglichkeit, Daten als Konstanten im Programmcode festzulegen, können sie auch in eine externe Datei geschrieben und von dort wieder ausgelesen werden. Das ist insbesondere für die Speicherung von Konfigurationsdaten hilfreich. Die Manipulation dieser Daten ermöglicht es schließlich, Einstellungen vorzunehmen, ohne den Programmcode ändern zu müssen. Im Diagramm in Abbildung 32 werden dafür sechs Datenstrukturen dargestellt. Die verstellbaren Eigenschaften der Kamera sind die Identifikationsnummer, die Aufnahmeauflösung, die zu erfassende Anzahl an Bilder pro Sekunde und die Belichtungszeit. Der Kameratreiber des Betriebssystems bietet zwar noch mehr

Einstellungsmöglichkeiten an, diese werden aber nicht von allen Herstellern unterstützt und in dieser Arbeit nicht benötigt. Für die Kalibration der Kamera können eine Vielzahl von Parametern eingestellt werden, die in Kapitel 5.2 beschrieben werden. Die beiden Klassen für die Verkehrsschild- und Hinderniserkennung verfügen lediglich über eine Markierung, welche angibt, ob die entsprechenden Module verwendet werden sollen oder nicht. Die Attribute der Fahrzeugkonfiguration sind die bereits im Absatz zu den Fahrzeugdaten definierten Dimensionen und Grenzwerte. Es ist sinnvoll, sie über die Konfigurationsdaten manipulieren zu können, jedoch nicht während das Programm ausgeführt wird. Teil der Konfigurationsdaten ist auch die Datenklasse für die Einstellung der Benutzerschnittstelle. Hier werden der Name sowie die Größe des Programmfensters, die Breite des Hauptmenüs und die Bildwiederholungsrate angegeben. Der Zugriff auf die Konfigurationsklassen wird durch den *Configurator* verwaltet. Die Details dazu finden sich ebenfalls in Kapitel 5.2.

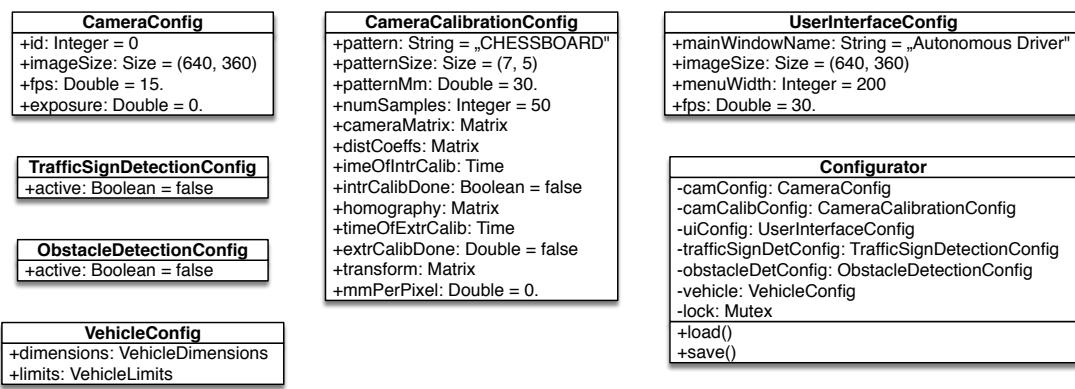


Abbildung 32: Klassendiagramme für die Konfigurationsdaten

3.6 Betriebssystem und Installation

Die hier entwickelte Software wird in der Programmiersprache C++ unter *Linux* entwickelt und benötigt auch dieses Betriebssystem, um ausgeführt zu werden. Der Hersteller des *Raspberry Pis* bietet dazu eine vorkonfigurierte, quelloffene Variante an. Diese heißt *Raspbian* und ist ein *Debian-Linux*-Derivat. Um die Installation dieses Systems zu vereinfachen, wird auch eine Software namens *NOOBS* (engl. *new out of box software*) angeboten. Sie kann kostenlos auf der Internetseite der *Raspberry Pi Foundation* bezogen werden. Um *NOOBS* auf dem *Raspberry Pi* ausführen zu können, muss es zuerst auf eine formatierte Micro-SD-Karte übertragen werden. Eine simple Anleitung dazu ist ebenfalls bei der Bezugsquelle unter [RP18f] verfügbar. Die Speicherkarte kann anschließend in den *Raspberry Pi* eingeschoben werden. Es empfiehlt sich neben der Stromversorgung, auch eine Maus, eine Tastatur und einen Monitor am Embedded-Board anzuschließen. Eine Benutzeroberfläche vereinfacht den Installationsvorgang.

Die vorkonfigurierte *Raspbian*-Version verfügt bereits über einige Softwarepakete und Softwarebibliotheken, die in dieser Arbeit benötigt werden. Dazu gehört die integrierte Entwicklungsumgebung *Geany*, der *GNU-Compiler GCC* und die Versionsverwaltung *Git*.

Weitere Programme und Bibliotheken lassen sich nachträglich über die Paketverwaltung *apt-get* installieren. Welche jedoch genau benötigt werden, hängt von den konfigurierten Repository-Servern ab. Zu beachten ist, dass in der Standardkonfiguration hauptsächlich Pakete mit Langzeitunterstützung zu finden sind. Das erhöht zwar die Sicherheit der angebotenen Programme, bietet aber nicht den gesamten Funktionsumfang der neuesten Versionen. Diese können dennoch manuell heruntergeladen, konfiguriert und kompiliert werden.

Für die digitale Bildverarbeitung wird hier *OpenCV* eingesetzt, das manuell nachinstalliert werden muss. Eine aktuelle Version lässt sich von *Github* herunterladen. In dieser Arbeit wurden die Versionen 3.2.0 bis 3.4.1 eingesetzt, wobei keine Unterschiede bei den verwendeten Funktionen zu bemerken waren. *OpenCV* setzt voraus, dass bereits eine Reihe weiterer Pakete installiert sind. Die wichtigsten sind *GCC*, *CMake*, *Git*, *GTK+2.x*, *pkg-config*, *Python 2.6*, *ffmpeg* und *libav*. Für diese Arbeit sind auch Pakete für den Export und Import von Bildern, wie *libjpeg-dev*, *libpng-dev*, *libtiff-dev*, *libjasper-dev* und *libdc1394* wichtig.

Zuletzt sei noch ein ergänzender Hinweis zur Systemreaktion gegeben. Wie bereits erwähnt, arbeiten die meisten Betriebssysteme, einschließlich des hier eingesetzten *Raspbian*, nicht deterministisch. Das bedeutet, dass nicht vorhergesagt werden kann, wann das Betriebssystem ein Programm ausführt. Laufen mehrere Prozesse zeitgleich nebeneinander, kann auch keine Prognose zur Abarbeitungsreihenfolge gemacht werden. Dies stellt für sicherheitskritische Applikationen, die eine Reaktion in Echtzeit benötigen, ein Problem dar. Für ein echtzeitfähiges System muss sich unter anderem angeben lassen, wie groß die längste Reaktionszeit ist. Eine mögliche Lösung ist der *PREEMPT_RT-Patch*, welcher den *Linux*-Kernel um Echtzeitfunktionen erweitert. Auch wenn bereits viele Teile des Patches in den Kernel eingeflossen sind, gibt es derzeit noch keine offizielle Echtzeitversion des Kernels (vgl. [B18a]). Die zwei wesentlichen Funktionserweiterungen des Patches sind *Sleeping Spinlocks* und *Threaded Interrupt Handler*. Erstere ersetzen Spinlocks durch Echtzeit-Mutexe, womit beim Warten auf den Lock keine CPU-Zeit benötigt wird. Wie der Name der zweiten Erweiterung verrät, werden Interrupt Handler als Kernelthreads ausgeführt. Damit muss das System nicht sofort auf Interrupts reagieren, was Reaktionsverzögerungen vermeiden kann. In dieser Arbeit wurde auf den Einsatz eines *PREEMPT_RT*-Patches verzichtet, weil dies über das Ziel hinausgehen würde. Für die Optimierung von Applikationen für das autonome Fahren ist dieser aber in Betracht zu ziehen, um die normierten sicherheitskritischen Anforderungen aus Kapitel 3.4 zu erfüllen. (vgl. [AE18], [B18a])

3.7 Monitoring

Fährt der Prototyp autonom auf einer Teststrecke, so ist für die Evaluierung von Applikationen eine Möglichkeit zur Überwachung und Fernsteuerung notwendig. Mit *Raspbian* werden die Applikationen *Secure Shell* (SSH) und *Virtual Network Computing* (VNC) mitgeliefert. Das erste der beiden Programme ermöglicht eine sichere Verbindung über ein Netzwerk zu einer Konsole am Embedded-Board. Darüber kann das gesamte Betriebssystem gesteuert und Applikationen gestartet oder beendet werden. Im

Flussdiagramm der Abbildung 33 wird die Befehlsfolge zum Start einer SSH-Kommunikation gezeigt. Die Konsole selbst erlaubt nur textuelle Ein- und Ausgaben. Um Kamerabilder zu sehen, wird eine zusätzliche graphische Benutzerschnittstelle benötigt. (vgl. [RVN18], [OSS18])

Abhilfe schafft die VNC-Applikation, welche das gesamte Desktopbild des *Raspberry Pi*s an den Monitoring-Computer sendet. Zunächst muss dafür ein VNC-Server auf dem *Raspberry Pi* gestartet werden. Auf dem Computer, der zur Fernsteuerung genutzt wird, ist eine VNC-Client-Anwendung erforderlich. Sowohl für den Server, als auch für den Client gibt es zahlreiche kostenfreie wie kommerzielle Realisierungen für die gängigen Betriebssysteme *Microsoft Windows*, *Apple macOS*, *Apple iOS*, *Google Android* und *Linux*. In diesem Projekt werden Programme von *RealVNC*, dem Erfinder von VNC, eingesetzt. Wie oben erwähnt, ist unter *Raspbian* ein VNC-Server bereits installiert. Ist eine Verbindung zwischen dem *Raspberry Pi* und dem steuernden Computer hergestellt, so ist am Client eine graphische Desktop-Umgebung zu sehen, welche wie gewohnt über die Maus und die Tastatur bedient werden kann. (vgl. [RVN18])

Die VNC-Bildübertragung benötigt ebenfalls Rechenressourcen, die mit der Software für das autonome Fahren geteilt werden müssen. Je höher die zu übertragende Bildauflösung und die CPU-Auslastung des *Raspberry Pi*s ist, desto mehr Verzögerung ist am Client wahrnehmbar. Eine genaue Messung der übertragenen Bilder pro Sekunde ist nicht möglich, da das Kommunikationsprotokoll dieser Technologie nur jene Pixel transferiert, welche sich an der Quelle, also am *Raspberry Pi*, geändert haben. Im Rahmen dieser Arbeit wurde mithilfe einer Stoppuhr eine ungefähre Messung durchgeführt, wobei eine Verzögerung von knapp unter einer Sekunde festgestellt wurden, was für diesen Kontext ausreichend ist. (vgl. [RVN18])

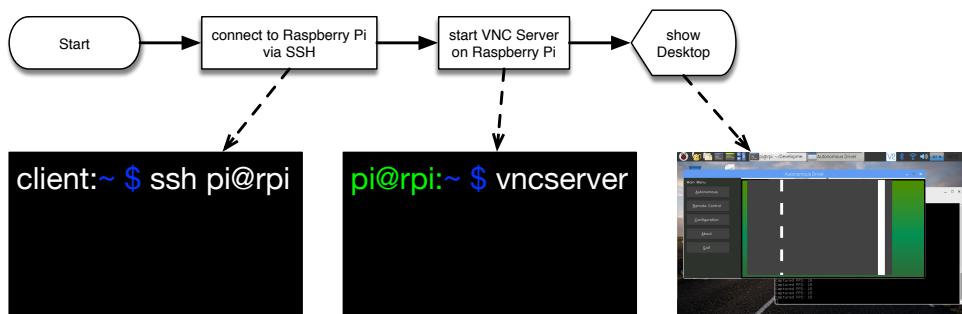


Abbildung 33: Flussdiagramm zur SSH- und VNC-Kommunikation

Alternativ können als zukünftiger Entwicklungsschritt dieser Software, ein Server und ein Client integriert werden, welche eine optimierte Kommunikation zwischen dem Embedded-Board und dem Monitoring-Computer erlauben. Über eine graphische Benutzeroberfläche am Client können Steuerungsbefehle an den Server am autonomen Fahrzeug übertragen werden. Umgekehrt können vom Server Statusinformationen und Bilder in effizienter Form zurückgesendet werden. Dies kann sogar ohne den Start einer Benutzeroberfläche am Server geschehen, womit zusätzlich Rechenressourcen eingespart werden können.

4. HARDWAREREALISIERUNG

Die gewählten Hardwarekomponenten aus Kapitel 3.2 werden im Folgenden zur Realisierung eines Prototyps eingesetzt. Dafür müssen einige Modifikationen und ergänzende Anbauten am Modellfahrzeug durchgeführt werden, um das Embedded-Board, den Akkumulator, die Kamera, den Ultraschallsensor und den Motortreiber daran zu befestigen. Während sich die Kamera ohne weiteres über USB anschließen lässt, müssen der Ultraschallsensor und der Motortreiber mit bestimmten Stiftkontakten des Mikrocomputers verbunden werden. Diese Anschlüsse werden hier ebenfalls beschrieben. In Abbildung 34 wird ein Überblick über den Aufbau des Prototyps gegeben, der in den folgenden Kapiteln genauer ausgeführt wird. Mit der Umsetzung dieses mobilen Prototyps wird ein wichtiges Ziel der in der Einleitung beschriebenen Aufgabenstellung erfüllt. Er ist Teil der hier erstellten Entwicklungsumgebung und dient sowohl als modifizierbares Werkzeug wie auch als HIL-Evaluierungsplattform für Applikationen des autonomen Fahrens.

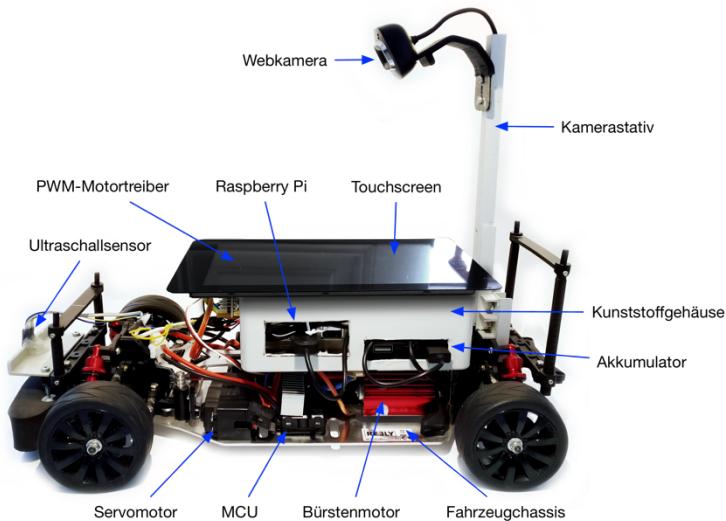


Abbildung 34: Aufbau des Prototyps

4.1 Chassis und Gehäuse

Eine wesentliche physikalische Anforderung an die Entwicklungsplattform ist, dass diese robust sein muss. Das bedeutet, dass Kollisionen mit Hindernissen bei den erreichbaren Geschwindigkeiten weitgehend unbeschadet bestanden werden müssen. Auch dürfen sich Komponenten bei plötzlichen Richtungsänderungen oder durch Vibrationen nicht selbstständig lösen. Das Chassis des in Kapitel 3.2 vorgestellten Modellautos besteht aus Aluminium und Kunststoff. Es ist so aufgebaut, dass es weitgehend verwindungssteif und stabil ist. Darauf kann ein weiteres Trägergehäuse für die Elektronik und ein Kamerastativ befestigt werden.

Eine robuste Verbindmöglichkeit zwischen zwei gleichen oder verschiedenen Werkstoffen ist das Verschrauben. Falls notwendig, erlaubt diese Methode, Bauteile auch wieder zu demontieren oder zu ersetzen. Aus diesem Grund werden alle tragenden Elemente mit M2-Schrauben¹⁴, Muttern und Beilagscheiben direkt am Aluminiumchassis des Modellfahrzeugs befestigt. Abbildung 35 stellt die Montierung dieser Bauteile dar. Eines dieser Teile ist ein ABS-Kunststoffgehäuse¹⁵ mit verschraubbarem Deckel. Es hat laut dem Hersteller eine Wandstärke von etwa 3 mm und ist elektrisch isolierend, lässt aber elektromagnetische Signale durch. Dieses wird mit M2-Distanzbolzen und Schrauben in der Mitte des Trägerchassis des Fahrzeugs befestigt. In diesem Gehäuse wird die Steuerelektronik stoßgeschützt platziert. Auf dem Gehäuse wird ein Deckel montiert, auf welchem der *Raspberry Pi* Touchscreen ebenfalls mit M2-Bolzen und M2-Schrauben befestigt wird. Für die Anschlüsse zu den Motoren, zur Kamera, zum Display und zu einer externen Stromversorgung, werden entsprechende Öffnungen in das Gehäuse gefräst.

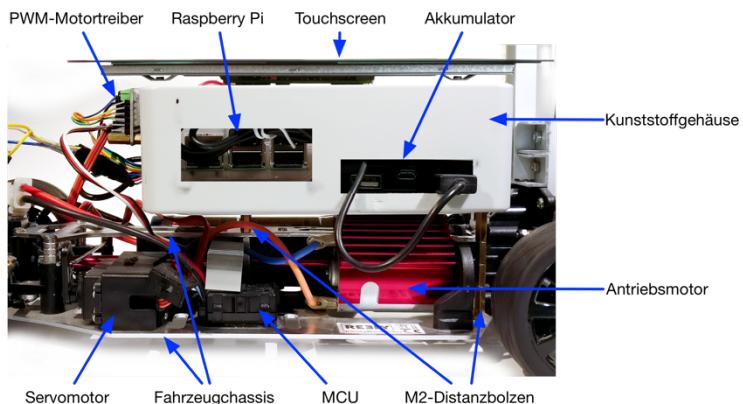


Abbildung 35: Befestigung der Elektronik mit dem Fahrzeugchassis

Das Embedded-Board wird zusätzlich in ein weiteres, kleineres Gehäuse geklemmt, welches in das oben beschriebene verschraubt ist. Somit ist die Elektronikplatine elektrisch besser isoliert und mechanisch geschützt. Ein weiterer Vorteil des zweiten Gehäuses ist, dass diese Hülle aus zwei Teilen besteht, die mit Klemmen zusammengehalten werden. Muss das Board ausgetauscht werden, lässt es sich einfach und ohne Werkzeug öffnen. An der Deckelseite des Gehäuses wird ein kleiner Ventilator montiert, der für die aktive Kühlung des Boards zuständig ist.

Damit das Fahrzeug sich frei fortbewegen kann, muss der *Raspberry Pi* über einen Akkumulator mit Strom versorgt werden. Das Trägergehäuse ist so dimensioniert, dass die Batterie neben dem *Raspberry Pi* befestigt werden kann. Dazu werden zwei Streifen widerstandsfähigen Klettbands verwendet. Diese erlauben es, den Akku einfach auszutauschen. Selbst nach mehreren Kollisionen bei hoher Geschwindigkeit mit einer Wand hat sich wie gewünscht keines der besprochenen Bauteile gelöst, gelockert oder verschoben.

¹⁴ M2 ist ein metrischer ISO-Standard nach der Norm DIN 13-1, welcher die Dimensionierung von Schraubengewinden definiert.

¹⁵ ABS, oder Acrylnitrit-Butadien-Styrol-Copolymer, ist ein temperatur- und witterungsbeständiger Kunststoff.

4.2 Kamerastativ

Auch die Kamera muss stabil am Modelfahrzeug befestigt werden. Die Montageposition der Kamera wird unter anderem von der verbauten Kameralinse bestimmt. Ziel ist es, einen möglichst großen Bereich vor dem Fahrzeug zu erfassen. Um eine Fahrbahn aufzuzeichnen, beginnt dieser Bereich am Stoßfänger des Fahrzeugs und reicht bis zum Horizont. Im realen Straßenverkehr befinden sich Verkehrsschilder einige Meter über der Fahrbahn, also ist auch dieser Bereich oberhalb des Horizonts für die Bildaufnahme von Interesse. Der Hersteller der hier eingesetzten Webkamera gibt einen diagonalen Sichtfeldwinkel von $68,5^\circ$ an. Um eine optimale Position zu finden, ist es sinnvoll, Aufnahmen von verschiedenen Punkten des Chassis aus zu erstellen und so den oben beschriebenen Bildausschnitt zu ermitteln. Für diese Arbeit sind Montagepunkte im Bereich der hinteren Achse geeignet. Diese Montageposition wird auch von vielen Projekten beim *Carolo-Cup* verwendet. Dieser Wettbewerb beschränkt zusätzlich die maximale Fahrzeughöhe auf 300 mm, was in dieser Arbeit ebenfalls beachtet wird.

Für die Konstruktion eines robusten und leichten Stativs eignen sich handelsübliche 12 mm bzw. 14 mm breite u-förmige Aluminiumprofile. In Abbildung 36 wird der Aufbau des Kamerastativs dargestellt. Das breitere der Profile wird vertikal positioniert und auf eine Länge von 80 mm gebracht. Es bildet den Unterteil des Stativs. Das Oberteil, an dem die Kamera befestigt wird, ist 12 mm breit und 205 mm lang. Werden die Profile gegengleich ineinander gesteckt, bilden sie einen schützenden Kanal für das USB-Kabel der Kamera und weisen eine hohe Verwindungssteife auf. Der Überlappungsbereich beträgt in etwa 75 mm und bietet daher genügend Reibung, um die zwei Teile zusammenzuhalten. Zur Sicherheit werden sie dennoch zusätzlich mit einer M2-Schraube und einer Mutter fixiert. Das Stativ lässt sich somit für den Transport einfach demontieren. Das untere Teilstück des Stativs muss am Fahrzeugchassis befestigt werden. An der Hinterachse befindet sich eine Halterungsvorrichtung für das Antriebsdifferenzial und für die hinteren Stoßdämpfer. Da dieses Bauteil aus Kunststoff ist, sorgt ein zusätzlicher Aluminiumrahmen für Versteifung. Daran wird auch das Stativ verschraubt.

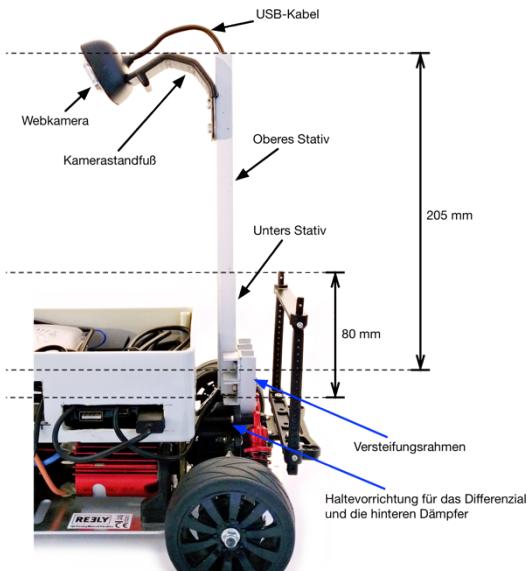


Abbildung 36: Kamerastativ des Prototyps

Die Kamera selbst verfügt über einen flexiblen Standfuß, der für die Aufstellung auf einer ebenen Fläche oder Fixierung an einem Bildschirmrand konzipiert ist. Nach einer entsprechenden Bohrung kann dieser aber auch fest an das obere Ende des schmäleren Aluprofils geschraubt werden. Die Beweglichkeit des Standfußes bleibt dabei erhalten, sodass die Kameraausrichtung nachjustiert werden kann. Bei heftigeren Kollisionen oder einer Demontage des Kamerastativs verstellt sich der eingestellte Winkel allerdings leicht. Ein anderer Mechanismus zur Winkelfixierung könnte zukünftig von Vorteil sein, wurde aber in dieser Arbeit nicht berücksichtigt, da weitere Umbaumaßnahmen am Kameragehäuse nötig gewesen wären, aber hier keinen wesentlichen Mehrwert gebracht hätten.

Der Kameraneigungswinkel und die Positionierung der Kamera ist nicht nur vom zu erfassenden Bildausschnitt, sondern auch vom Fahrbahnerkennungsverfahren abhängig. Muss eine extrinsische Kalibration durchgeführt werden, wie sie in Kapitel 5.2.2 beschrieben wird, so funktioniert dies nur mit einer Vorwärtsneigung der Kamera zur Fahrbahn hin. Grund dafür ist das verwendete Mustererkennungsverfahren der Kalibrierungsroutine von *OpenCV*. Ist die Kamera zu niedrig montiert und der Neigungswinkel dadurch zu klein, wird das Muster fehlerhaft oder gar nicht detektiert.

4.3 Abstandsmessung

Aus den Kameradaten kann mithilfe der Formel aus Kapitel 2.2.3 der Abstand des Fahrzeugs zu Hindernissen errechnet werden. Das setzt allerdings voraus, dass Hindernisse, also Objekte, die auf der Fahrbahn liegen, als solches erkannt werden. Aufgrund der eingeschränkten Rechenkapazitäten des in dieser Arbeit eingesetzten Computers, wird jedoch nicht die Erkennung aller im Straßenverkehr vorkommenden Objekte vorgenommen. Beispielsweise wurde aber die Detektion von Stoppschildern

implementiert. Wie in Kapitel 2.2.3 beschrieben, sind die Ergebnisse der Abstandskalkulation mittels Kamerabild mit einer gewissen Ungenauigkeit behaftet.

Ultraschallsensoren bieten die Möglichkeit einer günstigen und genaueren Messmethode, wie sie in Kapitel 2.1.1 erklärt wird. Der in dieser Arbeit eingesetzte Sensor hat laut dem Hersteller einen Messfehler von 0,3 mm und eine Reichweite von bis zu 4 m. Der Ultraschallsensor ist mittig an der vorderen Stoßstange des Fahrzeugs montiert und daher in Fahrtrichtung ausgerichtet. Am Sensor befinden sich die vier Anschlusspins für *Versorgung* (V_{CC}), *Ground* (GND), *Trigger* (TRIG) und *Echo* (ECHO). Die V_{CC} -Leitung muss an einen Versorgungsanschluss mit 5 V und der Ground an eine Masse des *Raspberry Pi*s angeschlossen werden. Über den Trigger-Pin kann ein Ultraschallsignal ausgelöst und über den Echo-Pin der Empfang desselben weitergeleitet werden. Abbildung 37 zeigt die physikalische Nummerierung der Pins am Embedded-Board, wobei das sowohl für den *Raspberry Pi 2* als auch 3 gilt. Die Anschlüsse 2 und 4 liefern die notwendigen 5 V, Pin 6 oder 14 liefern die Masse. Trigger und Echo können mit Pin 11 und 13 verbunden werden. Das Blockschaltbild dieser Verbindungen wird in Abbildung 38 gezeigt. Beim Programmieren ist zu beachten, dass diese physikalische Pin-Nummerierung in Abbildung 37 nicht gleich ist wie die GPIO¹⁶-Nummerierung des *Raspberry Pi*-Prozessorherstellers *Broadcom* (BCM). Diese beschriebenen Verbindungen sind im Blockschaltbild der Abbildung 38 zu sehen. Dies sei deshalb hier erwähnt, weil das bei der Softwareimplementierung beachtet werden muss. Dort kann mithilfe der Bibliothek *wiringPi* eine Nummerierungskonvention festgelegt werden.

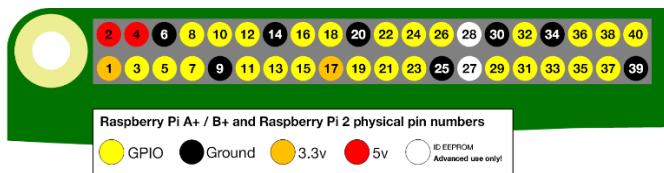


Abbildung 37: Pinbelegung am *Raspberry Pi 2* [RP18e]

Eine weitere wichtige Anmerkung ist, dass die GPIO-Pins nicht mit einer Spannung von über 3,3 V belastet werden dürfen. Wird am hier eingesetzten Sensor ein Ultraschallsignal empfangen, so liegen 5 V am Echo-Pin an. Mit einem Spannungsteiler lässt sich die Spannung reduzieren. Bei kleinen Strömen ist es ausreichend, den Spannungsteiler durch zwei Widerstände zu implementieren. Mit der folgenden einfachen Berechnungsformel werden die notwendigen Widerstandswerte bestimmt.

$$V_{out} = V_{in} * \frac{R_2}{R_1 + R_2}$$

Die Spannung V_{in} entspricht den 5 V des Echosignals. V_{out} sind die 3,3 V, die am *Raspberry Pi* anliegen sollen. In diesem Projekt beträgt der Widerstand R_1 , der sich zwischen V_{in} und V_{out} befindet, 330 Ω . Deshalb muss R_2 470 Ω groß sein. Je nach verfügbaren Bauteilen

¹⁶ General Purpose Input/Output (GPIO) ist eine Anschlussmöglichkeit elektronischer Bauteile, wie beispielsweise integrierter Schaltkreise (IC) oder SoCs, deren Verwendungszweck frei programmierbar ist.

kann R_1 und R_2 auch anders gewählt werden, wobei die obige Gleichung beachtet werden muss.

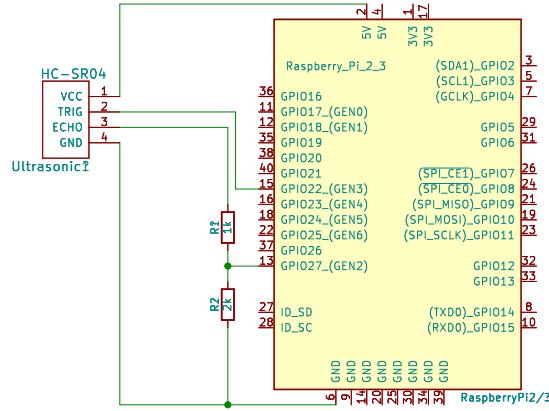


Abbildung 38: Verbindungsschaltplan zwischen *Raspberry Pi 3* und Ultraschallsensor

Auch ein autonomes Fahrzeug muss „vorrausschauend“ und somit nicht schneller fahren, als die Reichweite der Sensorik erlaubt. Mit steigender Geschwindigkeit steigt auch der Bremsweg deutlich an. Eine Faustregel, wie sie zum Beispiel in Fahrschulen gelehrt wird, lautet $l_{\text{Bremsweg}} = \left(\frac{v}{10}\right)^2$. Damit darf das Modellfahrzeug nicht schneller als 20 m/s bzw. 72 km/h fahren, um innerhalb des Abtastbereichs des Ultraschallsensors anhalten zu können. Das setzt allerdings voraus, dass die Reaktionszeit des Systems 0 Sekunden beträgt, was nicht möglich ist. Bis das Fahrzeug tatsächlich zum Stillstand kommt, ist also zum Bremsweg ein Reaktionsweg hinzuzurechnen. Beim Menschen wird hier mit einer Näherungsformel $l_{\text{Reaktionsweg}} = 3 * \left(\frac{v}{10}\right)$ für eine Reaktionszeit von einer Sekunde gerechnet. Damit dürfte eine Maximalgeschwindigkeit von 2,778 m/s bzw. 10 km/h nicht überschritten werden, um rechtzeitig vor dem Hindernis stehenbleiben zu können. Bei diesen Berechnungen handelt es sich nur um grobe Schätzungen, welche allerdings die Systemgrenzen verdeutlichen sollen. Der Hersteller des hier eingesetzten Modellautos gibt eine Maximalgeschwindigkeit von 30 km/h an. Das autonome System muss diese also zumindest um das Dreifache reduzieren. (vgl. [TK10] S. 157)

Es ist zu beachten, dass der hier vorgeschlagene Ultraschallsensor einen relativ engen Detektionsbereich aufweist. Das bedeutet, dass nur Abstände zu Objekten, die sich mittig vor dem Sensor befinden, zuverlässig gemessen werden können. Wenn sich ein Hindernis in einer Kurve vor dem Fahrzeug befindet, wird dieses deutlich später detektiert. Deshalb werden in Fahrzeugen in vollem Maßstab mehrere Ultraschallsensoren mit unterschiedlichen Abstrahlungswinkeln verbaut. Zukünftig kann das auch für die Weiterentwicklung des Prototyps dieser Arbeit in Betracht gezogen werden. Wie in den DARPA-Projekten demonstriert, empfiehlt es sich darüber hinaus, zusätzliche Abstandsmesssysteme mit einer größeren Reichweite, wie zum Beispiel RADAR, einzusetzen.

4.4 Motorsteuerung

Die Motoren des Modellfahrzeugs werden über einen PWM-Servotreiber angesteuert. Das Kommunikationsprotokoll des Treibers mit dem Embedded-Board ist I²C, welches über vier Anschlussleitungen mit dem *Raspberry Pi* kommuniziert. Abbildung 39 stellt die Verbindungen in einem Schaltplan graphisch dar. Über V_{CC} muss eine Versorgungsspannung von 3,3 V angeschlossen werden, die zum Beispiel Pin 1 am Embedded-Board liefert. GND kann an die Masse am Pin 8 angeschlossen werden. Die I²C-Datenleitung *Serial Data* (SDA) und die Takteleitung *Serial Clock* (SCL) werden mit Pin 3 und 5 verbunden. Mit dem Pin 11 des *Raspberry Pis* wird die Signalausgangssteuerung *Output Enable* (OE) des Treibers verbunden. Mittels dieser lassen sich im Notfall durch das Senden eines High-Signals alle Motoren abschalten. Am Treiber selbst können bis zu 16 Servomotoren mit Dreidrahtleitungen angeschlossen werden. Neben den Anschlüssen für 5 V und der Masse führt die dritte Leitung das pulsweitenmodulierte Signal, welches die eigentliche Motorstellung bzw. -drehzahl bestimmt.

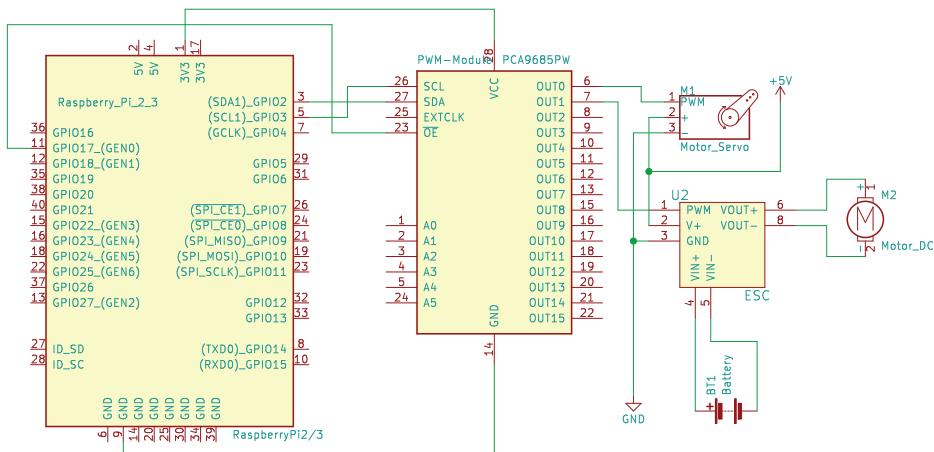


Abbildung 39: Verbindungsschaltplan von *Raspberry Pi* 3, PWM-Modul und Motoren

Der hier eingesetzte Servomotor wird über ein 50-Hz-Signal angesprochen. Durch die mechanischen Gegebenheiten des Fahrzeugs können die Räder nur bis zu einem Winkel von etwa 45° nach links bzw. nach rechts gedreht werden. Der Servomotor könnte allerdings von 0° bis 180° schwenken, wobei 0° der maximale Lenkeinschlag nach rechts und 180° der maximale Lenkeinschlag nach links ist. Im Winkel von 90° sind die Räder geradeaus gerichtet. Das System muss bei der Motoransteuerung die mechanischen Einschränkungen des Fahrzeugs beachten, um eine Überbelastung des Servos zu vermeiden. Das notwendige PWM-Signal hat bei 50 Hz eine Periodendauer von 20 ms. Die Pulslänge bestimmt den Stellwinkel des Servos, wie in Abbildung 40 dargestellt wird. Eine Pulslänge von 1 ms entspricht einem Winkel von 0°, bei 2 ms sind es bereits 180°. Um also geradeaus zu fahren, muss das Signal alle 20 ms einer Pulslänge von 1,5 ms entsprechen und daher einen *Duty Cycle* von 7,5 % aufweisen.

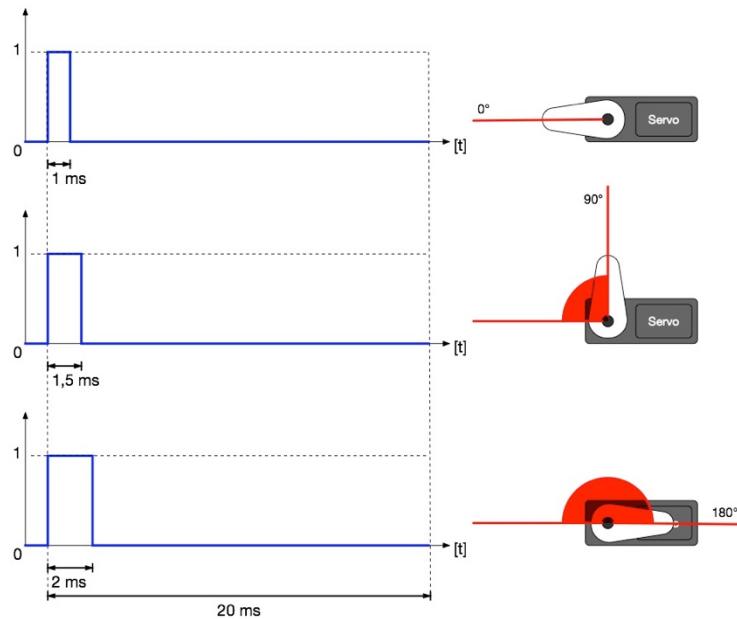


Abbildung 40: Pulsdauer und Servostellwinkel

Das hier eingesetzte Treibermodul unterstützt eine Pulsweitenangabe in der Auflösung von 12 Bit, also von $4096 \mu\text{s}$ mal $4,883 \mu\text{s}$ langen Samples. Eine einfache Rechnung zeigt, dass in etwa 307 Samples notwendig wären, um einen 1,5 ms Puls zu generieren. Aufgrund mechanischer Toleranzen im Modellbau werden Lenkwinkelstellungen des Motors jedoch nicht mit dieser Genauigkeit umgesetzt. Der Servomotor ist über eine Lenkstange mit dem Achsschenkellenker verbunden, der ein leichtes Spiel aufweist. Bei diesem Prototyp und den damit erreichbaren Geschwindigkeiten erweist sich diese Ungenauigkeit jedoch als unproblematisch. Ohne dies muss das System die Fahrrichtung fortlaufend so korrigieren, dass das Fahrzeug innerhalb der Fahrspur bleibt.

Die Motor Control Unit des Antriebsmotors wird ebenfalls über ein PWM-Signal mit 50 Hz gesteuert. Wie beim Servomotor ist die Pulslänge von 1 ms das Minimum und mit 2 ms das Maximum. Diese Werte stehen für die Rotationsrichtung und Drehzahlgeschwindigkeit des Motors. Ein PWM-Signal mit 1 ms Pulsdauer lässt den Motor mit maximaler Umdrehungszahl im Uhrzeigersinn rotieren. Über die angekoppelten Zahnräder und Differentialle wird diese Drehung in eine Rückwärtsumdrehung der Räder umgewandelt. Entsprechend verursacht die Pulsdauer von 2 ms die maximale Vorwärtsumdrehung. Damit bleibt der Motor bei einer Pulslänge von 1,5 ms oder 307 Samples stehen. Ein Versuch zeigt, dass kein linearer Zusammenhang zwischen der Umdrehungszahl der PWM-Signalveränderung herrscht. Um den Motorstillstand bei 1,5 ms ist ein Toleranzbereich von ca. $\pm 0,1$ ms, bei dem ebenfalls keine Bewegung des Motors initiiert wird.

Wird die MCU in Betrieb genommen, ertönt ein Startsignal. Dabei führt sie einen Selbsttest durch, bei dem die Neutralposition des Reglers bzw. der MCU kalibriert wird. Dafür wird ein externes PWM-Signal erwartet, welches dieser Position entspricht. Das wären in dem Fall ein Samplewert von 307 bzw. eine Pulsdauer von 1,5 ms. Durch hörbare Töne signalisiert die MCU das Akzeptieren des Signals und eine erfolgreiche Kalibrierung. Ab diesem Moment können zur Motorsteuerung auch alle anderen gültigen PWM-Werte, also solche

die zwischen den Pulsdauergrenzen von 1 ms bis zu 2 ms liegen, übertagen werden. Auch während der Kalibrierungsphase werden alle diese Signale akzeptiert. Wird jedoch ein anderer Wert als der tatsächliche Neutralwert übertragen, so nimmt der Regler diesen fälschlicherweise an und legt die Neutralposition auf diesen fest. Dadurch ist der Steuerungsbereich über und unter der Neutralposition nicht mehr gleich verteilt. Dies führt dazu, dass eine unterschiedliche Anzahl von Samples pro Fahrtrichtung akzeptiert wird und damit der Vorwärts- wie der Rückwärtsgang nicht mehr gleich skaliert sind. Dadurch verschiebt sich die Fahrtrichtung und die Motordrehzahl in Bezug auf das Pulswidensignal, was zu Problemen bei Ansteuerung, sowie bei der Interpretation von Testergebnissen führt. Es ist daher wichtig, während der Kalibrierung das richtige Signal mit einer Pulsdauer von 1,5 ms zu senden.

Die beschriebene Ansteuerung des Motors verursacht eine Bewegung desselben, die über Zahnräder an ein Differential geleitet wird. Von dort wird sie an die Hinterachse zu den Rädern des Fahrzeugs übertragen. Eine zusätzliche Kardanwelle verbindet das hintere Differential mit einem vorderen, welches die Kraft der Antriebswellen mechanisch an die Vorderachse weiterleitet. Ein leichtes Spiel und damit ein kleiner Leerlauf der Kardanwelle ist bei diesem Modellfahrzeug zu bemerken. Dennoch erfolgt die Kraftübertragung auch bei kleineren Drehzahlen problemlos.

5. SOFTWAREREALISIERUNG

Nach der Betrachtung der Hardwarerealisierung des Prototyps für das autonome Fahren wird im Folgenden die Softwarerealisierung desselben besprochen. Eine wesentliche Aufgabe des autonomen Systems ist es, die Fahrbahn und Objekte auf und um diese zu erkennen. Die visuelle Detektion mit einer Kamera orientiert sich dabei im Wesentlichen an der menschlichen Wahrnehmung. Diese ist abhängig von der Sichtweite des Menschen und dessen persönlicher Erfahrung. Erstere ist allerdings nicht nur vom Menschen beeinflussbar. Nebel, Regen oder Schnee stellen widrige Sichtbedingungen dar. Ebenso sind starke Kontraste und Belichtungsunterschiede störend. Das Auge und das Gehirn versuchen diese Defizite auszugleichen. Die Erfahrung hilft dem Menschen abzuschätzen, wo die Fahrbahn anfängt, aufhört und wie sie verläuft, auch wenn diese nicht vollständig sichtbar ist. Genauso werden Verkehrsschilder und andere Objekte erkannt, obwohl ihre Form, Farbe, Position oder Lage nicht immer gleich sind.

Wie in Kapitel 2.2 beschrieben, sind dies keine trivialen Aufgabestellungen für die Programmierung zum autonomen Fahren. Kameraaufnahmen benötigt zusätzliche Verarbeitungsschritte, um optische Defizite zu kompensieren. Dazu gehört eine Reihe von Korrekturen von Verzerrungen oder der Belichtung und des Kontrasts im Bild. Anschließend können Erkennungsverfahren durchlaufen werden.

Der Programmcode wird hier ausschnittsweise zitiert bzw. als Flussdiagramm dargestellt. Der vollständige Code samt Dokumentation ist hingegen im Anhang dieser Arbeit zu finden. Die folgenden Erläuterungen entsprechen auch der Organisationsstruktur des Programms. Dabei werden Modi und Module unterschieden. In einem Modus oder Systemzustand werden verschiedene Funktionen für das autonome Fahren durchgeführt. Module hingegen sind solche Funktionseinheiten, die konkrete Teilaufgaben des Gesamtziels lösen, die in Kapitel 3.4 identifiziert wurden. Sie werden von den Modi als Threads gestartet, wobei in einem Modus mehrere Module zum Einsatz kommen können.

Die im Folgenden dargestellten Module sind Implementierungen einer Schnittstellenklasse *Module*, welche einen Modulprototyp beschreibt. Diese Schnittstellenklasse definiert die in jedem Modul ausführbaren Funktionen *quit()*, *isRunning()* und *isError()*. Diese Klasse ist hier als Teil der Entwicklungsumgebung zu verstehen und ermöglicht die Erweiterung derselben durch neue Funktionen. Solche können hinzugefügt werden, ohne dass dabei das übrige System modifiziert werden muss. Auch jeder im Folgenden beschriebene Modus ist eine Spezialisierung einer abstrakten Schnittstellenklasse *Systemmode*, wie in Abbildung 23 dargestellt ist. Diese erlaubt auch auf dieser Ebene eine Erweiterung des Systems mit zusätzlichen Modi, ohne dieses dazu grundlegend verändern zu müssen. Diese flexible Realisierung ist Teil der für diese Arbeit entworfene modulare Entwicklungsumgebung. In jedem Modul sind die drei Funktionen *run()*, *quit()* und *stopModules()* implementiert. Hiermit wird die Softwaremodularität, die Teil der eingangs genannten Aufgabenstellung ist, ermöglicht. Damit ist ein weiterer Aspekt der Zielsetzung dieser Arbeit erfüllt.

Nachdem zunächst der *Initialisierungszustand* des Systems beschrieben wird, muss die *Konfiguration* desselben genauer betrachtet werden. Dazu gehören auch die beiden Module zur *Kamerakalibration*. Als nächstes werden vier weitere Module vorgestellt, welche

für die *Bilderfassung*, die *Detektion*, die *Pfadplanung* sowie die *Fahrzeugsteuerung* zuständig sind. Die ersten beiden entsprechen der Systemwahrnehmung. Darin sind deshalb die drei Erkennungsverfahren dieser Arbeit für die Fahrbahn, für Verkehrsschilder und Hindernisse enthalten. Als nächstes wird das Modul für die graphische Benutzerschnittstelle vorgestellt, die sowohl über einen Touchscreen am Modelfahrzeug, als auch über einen Fernzugriff über ein drahtloses Netzwerk bedient werden kann. Das Modul stellt für den jeweiligen Modus, in dem sich das System gerade befindet, speziell angepasste graphische Oberflächen dar. Diese graphischen Benutzeroberflächen sind als Zustände des graphischen Benutzerschnittstellenmoduls zu verstehen. Daher werden diese im letzten Abschnitt des Kapitels gemeinsam mit den verschiedenen Systemzuständen, zu denen sie gehören, erklärt. In der Erklärung der Modi werden auch deren Abläufe dargestellt. Diese nutzen die Funktionalität der vorher besprochenen Module.

5.1 Initialisierung

Das Programm startet in der Hauptroutine *main()*. Hier wird zuerst die *Signal-Handler*-Funktion *signalHandler()* initialisiert. Diese ist dafür zuständig, auf bestimmte Betriebssystemssignale, wie *SIGINT*, *SIGQUIT* und *SIGTERM*, welche das Programm beenden würden, zu reagieren. In jedem Fall wird das laufende Programm so beendet, dass die Motoren abgeschaltet werden. Dadurch erreicht das Fahrzeug den in Kapitel 3.4 besprochenen sicheren Zustand des Fahrzeugstillstands, bevor das Programm beendet wird. In Abbildung 41 wird das Flussdiagramm zum erklärten Ablauf gezeigt.

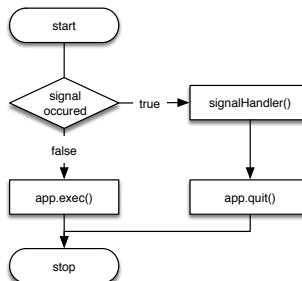


Abbildung 41: Flussdiagramm der Funktion *main()*

Die Applikationsklasse *AutonomousDriver* bietet die Möglichkeit, das System auszuführen bzw. sicher zu beenden. Sie enthält den aktuellen Systemzustand *state*. Abbildung 42 zeigt das Diagramm für die Klasse und den Ablauf der Hauptfunktion *exec()*. Wird diese Funktion ausgeführt, dann ist zuerst zu überprüfen, ob es gespeicherte Systemkonfigurationsinformationen gibt und die darin enthaltenen Werte korrekt sind. Dazu wird die Datei *config.xml* und der Ordnerpfad einer statischen *Configurator*-Instanz *config* übergeben. Das Suchen der Datei und das Laden der Einstellungswerte daraus wird mit der Funktion *load()* durchgeführt.

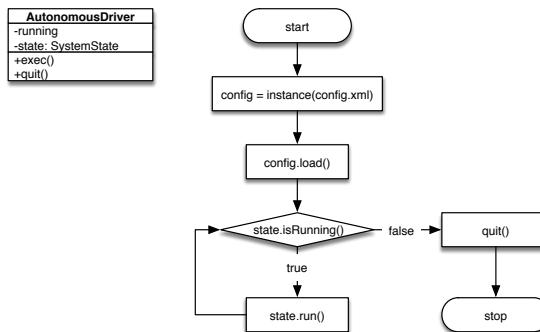


Abbildung 42: Flussdiagramm der Funktion exec()

Um die Konfigurationsdaten allerdings sicher verwenden zu können, müssen sie zuerst validiert werden. Das ist deshalb notwendig, weil die Konfigurationsdatei von einem Benutzer manipuliert werden kann und dadurch möglicherweise fehlerhafte Werte eingegeben worden sein könnten. Den Ablauf des Ladeprozesses der Konfiguration zeigt Abbildung 43. Sind die Daten fehlerhaft oder konnten sie nicht geladen werden, so wird versucht, eine Datei default.xml zu öffnen, welche Standardwerte enthält. Wenn eine gefunden wird, so wird auch diese validiert, da sie ebenfalls mit falschen Werten bearbeitet worden sein könnte. Falls die Standardwerte fehlerhaft sind oder keine gefunden wurden, dann wird eine neue Datei mit dem Namen default.xml generiert, die notwendige Mindestangaben für die Programmausführung enthält. Diese Angaben und weitere Details zur Systemkonfiguration werden im nächsten Abschnitt betrachtet.

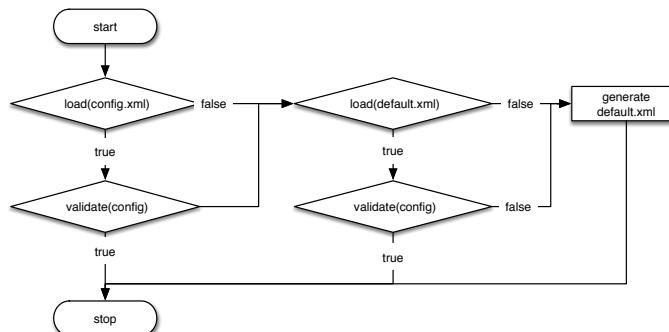


Abbildung 43: Flussdiagramm zum Laden der Konfiguration

Nach diesem Ladevorgang der Konfiguration wird der Systemzustand mit *run()* gestartet. Der hier beschriebene Ablauf entspricht dem Zustand *Initial*, welcher in Abbildung 21 dargestellt ist. Sobald *run()* zum ersten Mal ausgeführt wird, startet das System mit dem Folgemodus *Standby*.

5.2 Konfigurationsmodule

Die Konfiguration des Systems in dieser Arbeit ist zum einen über die im vorigen Abschnitt erwähnte Datei config.xml und zum anderen über den gleichnamigen Systemmodus möglich. Für das Erstellen und Auslesen der Konfigurationsdatei wird die FileStorage-Klasse von OpenCV benutzt. Sie ermöglicht das Abspeichern aller primitiven C++- und einiger

spezifischer OpenCV-Datentypen, wie Matrizen oder Koordinaten. Sind die Daten in einer XML-Datei gespeichert, so können sie mit jedem beliebigen Texteditor betrachtet oder bearbeitet werden. Die Werte selbst werden mit der Syntax `<name>value</name>` zwischen die XML-typischen Tags gesetzt.

Der Konfigurationsmodus bietet eine graphische Benutzeroberfläche zur Kamerakalibration und zur Einstellung der Ausgabebildposition. Die Kalibration ist aufgrund der Bauweise von Kameras notwendig. In den Grundlagen in Kapitel 2.1.1 wird erläutert, dass digitale Kameras im Allgemeinen aus einem Gehäuse, einem Bildsensor und einer Blende bestehen. Davor ist eine Linse angebracht, die einfallende Lichtstrahlen so bündelt, dass sie auf den Sensor treffen. Dadurch entsteht jedoch ein Problem. Lichtstrahlen, die an den Außenrändern der Linsen auftreffen, werden so gebrochen, dass das resultierende Bild an den Randbereichen verzerrt ist. Mit mathematischen Methoden lassen sich diese Verzerrungen in der Bildverarbeitung allerdings deutlich minimieren. Nach dieser Entzerrung können Objekte, Formen, Abstände und Positionen im aufgenommenen Bild genauer bestimmt werden. Die folgenden Softwareroutinen ermöglichen dem System, Fehler in der Bildaufzeichnung selbstständig zu korrigieren.

5.2.1 Intrinsische Kamerawerte

Für die Kalibration der internen Kameraparameter wird hier ein modellbasiertes Objekterkennungsverfahren eingesetzt, wie es Gábor in der OpenCV-Dokumentation [G18a] oder Kaehler und Bradski [KB17] vorschlagen. Dazu wird in einem Bild nach einem bekannten Muster, wie beispielsweise einem Schachbrett- oder Punktemuster, gesucht. Durch die Kameralinse kann die Aufnahme radial oder tangential verzerrt sein. Eine radiale Verzerrung entsteht durch die Wölbung der Linse, die mit dem Wölbungsradius zunimmt. Tangential wird das Bild dann verzerrt, wenn die Linse nicht exakt parallel zum Bildsensor liegt. Mithilfe entsprechender Gleichungen können die Kalibrierungswerte, also sowohl die Kameramatrix als auch der Koeffizientenvektor $(k_1, k_2, p_1, p_2, k_3)$ berechnet werden, mit denen das Bild entzerrt werden kann. Die Matrix hat die Form

$$\begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

und enthält die

Brennweite $f = (f_x, f_y)$ sowie die optische Achse $c = (c_x, c_y)$. Je mehr Bilder mit dem bekannten Muster zur Kalibration verwendet werden, desto genauer können die Matrix und der Vektor berechnet werden.

Bevor die Kalibration selbst durchgeführt wird, muss die `runIntrinsicsCalibration()`-Funktion des Bilderfassungsmoduls CameralImageAcquisitor, welches im Kapitel 5.3.1 erklärt wird, als Thread gestartet werden. Das Diagramm in Abbildung 44 zeigt den Programmfluss des Threads. Zur Ausführung des Threads werden das Aufnahmefeld aus `inputImage` als Datentyp `ImageData` und die noch leere Bildmatrix `outputImage` ebenfalls als Datentyp `ImageData` benötigt, in die das entzerrte Ausgabebild abgespeichert werden soll. Für die spätere Speicherung der berechneten Kalibrierungswerte wird auch der Zugriff auf die aktuellen Benutzereingaben benötigt, die von `uiState`, welches ein `UserInterfaceState`-Typ ist, bereitgestellt werden. Der erste Schritt ist es, die Konfigurationsdaten für das Modul zu laden. Dazu gehören das zu erkennende Muster, die vertikale und horizontale Anzahl der Musterelemente sowie die Größe eines Elements in Millimetern und die Anzahl der für die

Kalibrierung vorgegebenen Bildsamples. Diese Werte sind zugleich Teil der Mindestangaben für den Systemstart, weil ohne sie keine Kalibration durchgeführt werden kann.

Als nächstes wird eine Programmschleife ausgeführt. In dieser wird überprüft, ob das intrinsische Kalibrierungsmodul weiter ausgeführt werden soll. Wurde durch den Benutzer oder einen Modus die Funktion *quit()* aufgerufen, dann wird die Schleife beendet. In der Schleife wird dann das aktuelle Eingabebild, welches von einer Kamera aufgezeichnet wurde, geladen. Sind Daten in der Bildmatrix vorhanden und ist die in den Konfigurationsdaten geforderte Anzahl der Samples für die Kalibration noch nicht erreicht, so soll die Kalibration mit der Funktion *calibrateIntrinsics()* und den aktuellen Eingabebilddaten durchgeführt werden. Der Ablauf dieser Funktion wird weiter unten genauer beschrieben. Ist die Funktion erfolgreich, also wurde ein Muster erkannt und die Werte der Matrix sowie des Vektors berechnet, dann wird der Samplezähler einmal erhöht. Für ein besseres Kalibrierungsergebnis ist es von Vorteil, das Muster in verschiedenen Abständen und Ausrichtungen vor der Kamera zu platzieren. Dadurch werden Verzerrungen im Bild deutlicher erkannt. Aus diesem Grund wird die Kalibration als nächstes für 250 ms ausgesetzt. Der Benutzer kann in dieser Zeit das Muster neu platzieren. Wird während der Kalibration kein Muster erkannt, dann werden keine Samples gezählt, aber die oben erwähnte Pause dennoch eingehalten. Ist die nötige Anzahl der zu erfassenden Samples erreicht, so wird keine Kalibration mehr durchgeführt.

Nach jedem gefundenen Muster werden die Kameramatrix und der Koeffizientenvektor in der Funktion *calibrateIntrinsics()* neu berechnet. Sie werden dazu verwendet, im nächsten Schritt das Eingabebild mit der OpenCV-Funktion *undistort()* zu entzerren. Das Ergebnis dieses Vorgangs wird als Ausgabebild gespeichert und dem Benutzer angezeigt. Gibt es keine Kalibrierungswerte, so wird das Originalbild angezeigt, aber nicht entzerrt.

Die Kalibrierung der intrinsischen Kamerawerte ist dann erfolgreich, wenn die vorgegebene Sampleanzahl erreicht wurde und Daten in der Kameramatrix sowie im Verzerrungskoeffizientenvektor vorhanden sind. Ist dies der Fall, dann werden der Variable *camCalibConfig* diese Werte für den weiteren programminternen Gebrauch übergeben. Weiters wird auch der Kalibrierungszeitpunkt gespeichert und die *intrCalibDone*-Flag markiert. Nachdem die Werte nach Beendigung der Programmschleife in die Variable *camCalibConfig* übernommen wurden, können sie nach Wunsch des Benutzers auch in die zentrale XML-Konfigurationsdatei abgespeichert werden. Diese Speicherung kann während der Laufzeit der oben besprochenen Programmschleife durch das Drücken der Taste S vom Benutzer beantragt werden. Dadurch können die Werte nach einem Programmneustart wieder aufgerufen werden und die Kalibration muss nicht jedes Mal neu ausgeführt werden.

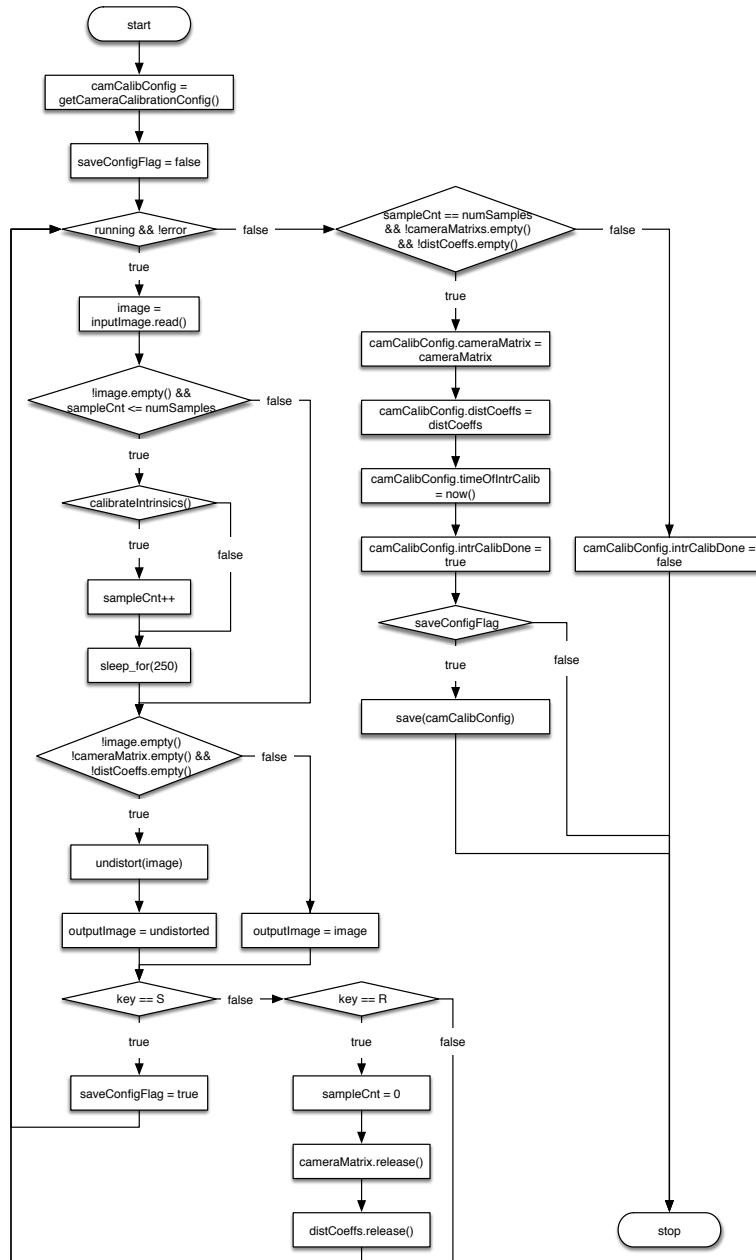


Abbildung 44: Flussdiagramm des Moduls zur intrinsischen Kamerakalibration

Wie oben angekündigt, wird hier die Funktion `calibrateIntrinsics()` genauer beschrieben. Sie führt die eigentliche Berechnung der Kalibrationswerte durch. Die Softwarebibliothek `OpenCV` bietet dafür einige Verfahren an, die hier zum Einsatz kommen. Abbildung 45 beschreibt den hier besprochenen Programmablauf. Bevor ein Muster im Eingabebild gesucht werden kann, wird das Bild mit der `OpenCV`-Funktion `cvtColor()` in ein Graustufenbild umgewandelt. Dadurch wird aus der dreidimensionalen Farbbildmatrix der Eingabe eine eindimensionale Matrix mit einem bestimmten Helligkeitswert für jedes Pixel. Mit der `OpenCV`-Funktion `findChessboard()` wird in dieser Graustufenmatrix nach einem Schachbrettmuster gesucht, dessen Größe in den Konfigurationsdaten angegeben ist. Dort ist allerdings nicht die Anzahl der Felder angegeben, sondern die Anzahl der Musterecken

innerhalb der äußersten Elementreihe. Bei dem in dieser Arbeit verwendeten Schachbrettmuster von 6x8 Feldern ergibt dies 5x7 Musterecken. Werden die Schachbrettfelder gefunden und deren Eckpunktpositionen ermittelt, dann wird die OpenCV-Funktion `cornerSubPix()` aufgerufen. Diese berechnet die Position der Ecken genauer und verbessert damit das Ergebnis des Verfahrens. Die Punkte werden bei jedem Durchlauf der `calibrateIntrinsics()`-Funktion zu einem Vektor `imagePoints` hinzugefügt. Sind schließlich Daten in diesem Ergebnisvektor abgelegt, dann werden mit der Funktion `calcBoardCornerPosition()` die theoretischen Sollpositionen der Eckpunkte berechnet, als wäre das Muster perfekt normal zur Sichtachse der Kamera. Diese Sollpositionen werden im Vektor `objectPoints` abgespeichert.

Der Vergleich der Vektoren `imagePoints` und `objectPoints` gibt Aufschluss über die Bildverzerrung. Diese Analyse führt die OpenCV-Funktion `calibrateCamera()` durch und kalkuliert dabei die Kameramatrix sowie den Koeffizientenvektor. Am Ende wird überprüft, ob die Resultate gültige Zahlenwerte enthalten. Das Ergebnis wird an den Thread `runIntrinsicsCalibration()` zurückgegeben und die Programmschleife des Threads wird, wie oben beschrieben, fortgesetzt.

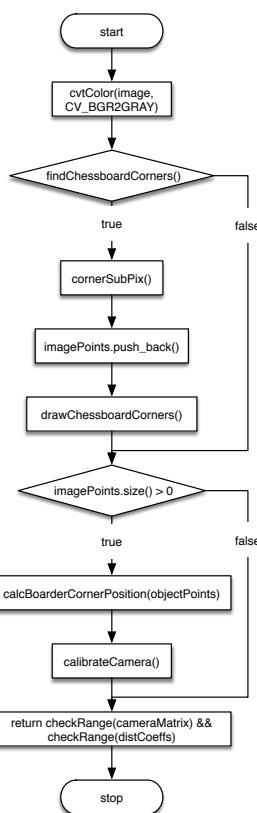


Abbildung 45: Flussdiagramm der intrinsischen Kalibrierung

5.2.2 Extrinsische Kamerawerte

Durch die in Kapitel 4.2 beschriebene Positionierung der Kamera, kann die Fahrbahn nicht von oben aufgenommen werden, sondern ist in der Aufnahme räumlich verzerrt und muss daher für die Fahrbahnerkennung entzerrt werden. In Kapitel 2.2.1 wurde dafür die inverse

Perspektivtransformation vorgestellt und ihre Auswirkung in Abbildung 8 dargestellt. Diese kann hier eingesetzt werden, um eine Art Vogelperspektive der Fahrbahn zu erhalten. Dafür wird ein Schachbrettmuster mittig vor die vordere Stoßstange des Fahrzeugs auf die Fahrbahn gelegt. Die so akquirierten Kamerabilder werden im Folgenden für die extrinsische Kamerakalibrierung als Eingabebilder herangezogen.

Die Funktion `runExtrinsicsCalibration()` ist ebenfalls Teil des in Kapitel 5.3.1 erklärten Moduls `CameraImageAcquisitor` und läuft ähnlich der Funktion für die intrinsische Kalibration ab, wie in Abbildung 46 zu sehen ist. Die Eingabeparameter mit `inputImage`, `outputImage` und `uiState` werden auch hier benötigt. Wieder wird die aktuelle Kalibrierungskonfiguration geladen und eine Bearbeitungsschleife gestartet. Sind Eingabebilddaten vorhanden, dann kann die Kalibration mit der Funktion `calibrateIntrinsics()` durchgeführt werden. Diese wird wiederum weiter unten genauer erklärt. Das Ergebnis der Kalibration ist eine Homographiematrix, in welcher Informationen über die Rotation, die Translation, die Verzerrung und die Skalierung der Istposition gegenüber der Sollposition der Bildfläche angegeben sind. Für die Benutzerschnittstelle wird das Bild mit der OpenCV-Funktion `warpPerspective()` perspektivisch verzerrt, wobei zuvor überprüft wird, ob Werte für das Eingabebild und die Homographiematrix vorhanden sind. Auch in diesem Modul kann der Benutzer das Abspeichern der Kalibrationswerte durch das Drücken der Taste S initiieren. Dafür werden die Homographiematrix, der Kalibrationszeitpunkt sowie die Flag `extrCalibDone` an die Systemkonfiguration übergeben und in der zentralen Konfigurationsdatei abgelegt.

Da sich durch die perspektivische Verzerrung auch die Abstände der Schachbrettmustereckpunkte im Bild verändern, kalkuliert die Funktion `calcMmPerPixel()` wie viele Millimeter ein Pixel im verzerrten Bild hat. Das tut sie durch die erneute Detektion des Schachbrettmusters im transformierten Bild. Da die Größe eines Musterfeldes mit 30 mm bekannt ist, kann aus den Pixelabständen zwischen den dort gefundenen Eckpunkten die Millimetergröße eines Pixels kalkuliert werden. Zu beachten ist, dass die Eckpunkte im Hintergrund des Ursprungsbilds, also jene deren Abstand zur Kamera größer ist, mit weniger Pixel aufgenommen werden, als jene, welche im Vordergrund liegen. Durch die Transformation entsteht daher eine Unschärfe der hinteren Ecken. Aus diesem Grund wird der Pixelabstand zwischen allen Ecken gemessen und der Durchschnitt davon errechnet. Das Ergebnis liegt nach mehreren Messungen bei etwa 5 Pixeln pro Millimeter bei einer Eingabebildauflösung von 640x360 px. Diese hohe Auflösung ist für die vorliegende Arbeit ausreichend. Die Umrechnung von Millimetern auf Pixel ist nützlich, um im weiteren Programmablauf den Abstand zweier erkannter Fahrbahnmarkierungen in einem Bild zu messen.

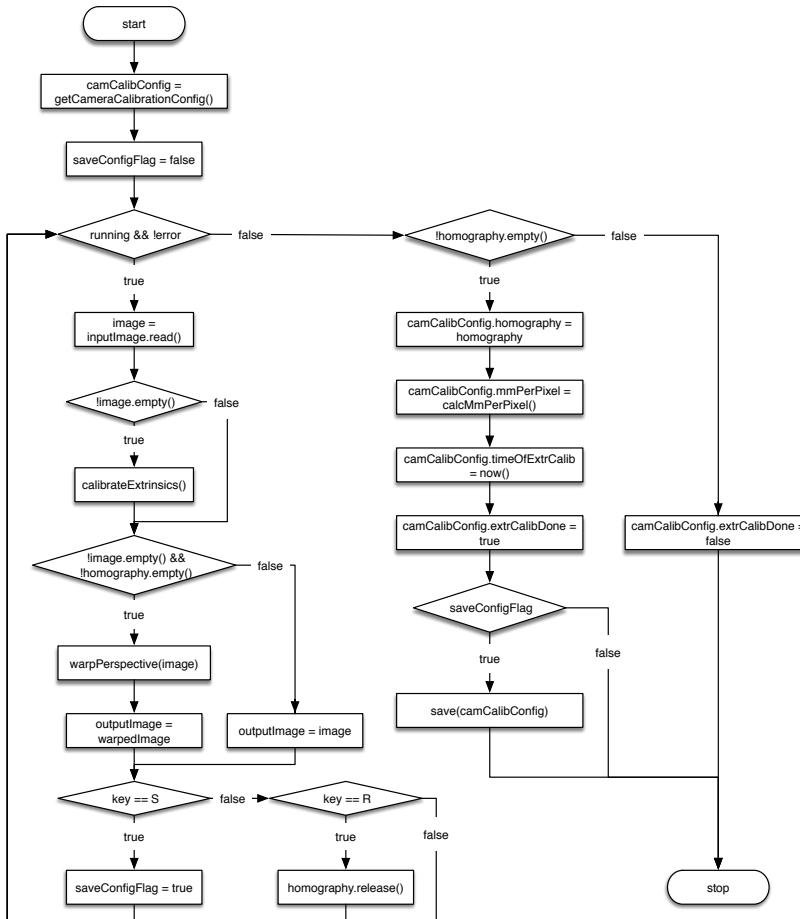


Abbildung 46: Flussdiagramm des Moduls zur extrinsischen Kamerakalibration

Wie oben angekündigt, wird hier die Hauptfunktion des besprochenen Moduls *calibrateExtrinsics()* beschrieben. Sie setzt den in Kapitel 2.2.1 beschriebenen pragmatischen Ansatz zur Homographiefindung für die Perspektiventransformation um, wie das Flussdiagramm in Abbildung 47 zeigt. Dazu wird das Eingabebild mit der OpenCV-Funktion *cvtColor()* in eine Graustufenmatrix umgewandelt. In dem Ergebnisbild wird, wie beim Verfahren für die intrinsische Kamerakalibration, nach einem Schachbrettmuster gesucht. Wird es gefunden, so erhöht die OpenCV-Funktion *cornerSubPixel()* die Genauigkeit der detektierten Eckpunkte. Für die Perspektiventransformation braucht es ausschließlich die vier äußersten der gefundenen Eckpunkte. Sie werden im Vektor *imagePoints* gespeichert. Die gewünschte Transformation entsteht durch die Verzerrung dieser Eckpunkte zu einem Rechteck. Die Sollpositionen werden aus der Anzahl der Schachbrettmusterecken und der Größe eines Feldes in Millimetern berechnet. Der Vektor *objectPoints* speichert diese Werte. Mit der OpenCV-Funktion *getPerspectiveTransform()* wird durch den Vergleich der Ist- und Sollpositionen der jeweiligen Eckpunkte eine entsprechende Transformations- bzw. Homographiematrix kalkuliert.

Im Eingabebild wird die linke obere Ecke des Schachbrettmusters nach der Transformation mit der OpenCV-Funktion *warpPerspective()* im Ausgabebild auf der Position (0,0), das heißt auf der linken oberen Ausgabebildecke, angezeigt. Dadurch ist nur das Schachbrettmuster zu sehen, der Rest der Fahrbahn, auf dem das Muster liegt, jedoch

nicht. Um das zu korrigieren, muss die Homographiematrix nach rechts unten verschoben werden. Das geschieht durch die Multiplikation derselben mit einer Skalierungsmatrix. In horizontaler Richtung soll das Schachbrett muster im Ausgabebild mittig platziert werden. Die Berechnung dieser Verschiebung geschieht durch $x_{shift} = (-b + n_{horizontal}) \cdot z$, wobei b die Bildbreite, $n_{horizontal}$ die horizontale Musterfeldanzahl und z ein Skalierungsfaktor ist. Ähnlich ist die Verschiebung in vertikaler Richtung zu berechnen. Da sich das Schachbrett muster an der Fahrzeugfront befindet, muss die Verschiebung also so durchgeführt werden, dass die unterste Musterreihe am unteren Ausgabebildrand platziert wird. Dies lässt sich mit $y_{shift} = -h + (\frac{n_{vertical}}{2}) \cdot z$ berechnen, wobei h die Bildhöhe, $n_{vertical}$ die vertikale Anzahl der Musterfelder und z der Skalierungsfaktor ist. Diese berechneten Werte werden in eine 3×3 -Skalierungsmatrix eingesetzt und mit der Homographiematrix multipliziert $H^* = H * \begin{bmatrix} x_{shift} & 0 & 0 \\ 0 & y_{shift} & 0 \\ 0 & 0 & z \end{bmatrix}$. Der Skalierungswert z wirkt dabei wie ein Vergrößerungsfaktor und wird hier auf $\frac{1}{2}$ gesetzt. Dadurch wird das Ausgabebild so verkleinert, dass von der Fahrbahn vor dem Fahrzeug mehr sichtbar wird. Diese korrekt skalierte Homographiematrix wird wiederum an den Thread `runExtrinsicsCalibration()` zurückgegeben und die Programmschleife des Threads wird, wie oben beschrieben, fortgesetzt.

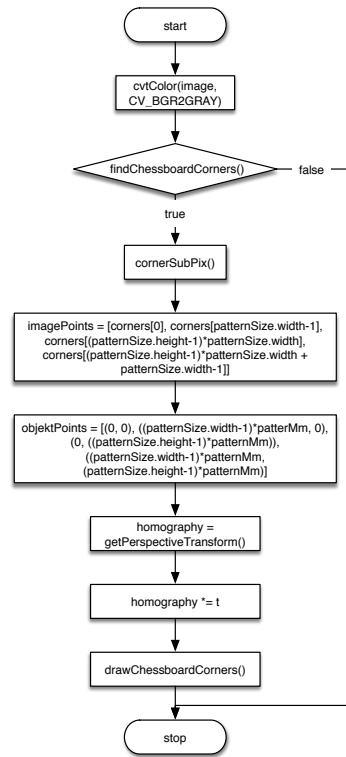


Abbildung 47: Flussdiagramm der extrinsischen Kalibrierung

5.3 Wahrnehmungsmodule

Die in Kapitel 2.2 beschriebenen Konzepte zur Bilddatenaufnahme und -verarbeitung lassen sich ebenfalls mithilfe von *OpenCV* umsetzen. Es stehen dafür einige Bibliotheksfunktionen zur Verfügung, die in den entsprechenden Algorithmen zur Fahrbahn- und Verkehrsschilderkennung eingesetzt werden können. Über die Kamera aufgenommene Bilder werden nicht in der vollen Auflösung von 1280x720 Pixel benötigt und können daher von vornherein in einer geringeren Auflösung aufgenommen werden. Dieser Schritt bringt, wie in Kapitel 2.2.1 beschrieben, mehrere Vorteile. So wird störendes Bildrauschen, das in jeder Bilddatenaufnahme wahrzunehmen ist, deutlich reduziert. Damit muss das Bild auch nicht mit einem starken Weichzeichnungsfilter gefaltet werden, womit wiederum notwendige Details erhalten bleiben. Die benötigte Zeit zur Manipulation eines Bildes steigt mit der Anzahl der Pixel. Daher können Bilder mit einer geringeren Auflösung schneller bearbeitet werden, was sich positiv auf den Gesamtablauf des Programms auswirkt.

5.3.1 Bilderfassungsmodul

OpenCV verfügt grundsätzlich über drei Arten, ein Bild aufzuzeichnen. Die erste Möglichkeit ist, ein Bild in den gängigen Formaten BMP, JPEG, PNG oder TIFF mit dem Befehl *imread()* aus einer Datei zu laden. Die anderen Möglichkeiten sind, mehrere Bilder in Form eines Videos oder einer Bilderserie mit der Klasse *VideoCapture* zu erfassen. Diese können entweder aus einer Datei oder von einer Kamera geladen werden. Nach dem Aufrufen eines solchen Kameragerätes, können Einstellungen an diesem durchgeführt werden, sodass die erfassten Bilder die gewünschte Auflösung haben. Wie bei der *Initialisierung* in Kapitel 5.1 angesprochen, ist die Einstellung von Kameraparametern von der Unterstützung der Kamera durch den V4L-Treiber abhängig. Die verfügbaren Funktionen des Treibers können mit dem Befehl *v4l2-ctl -L* ausgelesen werden. In dieser Arbeit ist lediglich die dritte der beschriebenen Bilddatenzeichnungsarten implementiert, die Bilderserien über eine USB-Webkamera aufnimmt. Für Tests können alternativ aber auch vorher abgespeicherte Bilderserien oder Videos aus Dateien geladen werden.

Für den Start des Moduls *CameralImageAcquisitor* wird eine Eingabebildmatrix *imageData* als Datentyp *ImageData* benötigt, worin die Daten des aufgenommenen Bildes abgelegt werden. Wie in Abbildung 48 zu sehen ist, wird danach die Kamerakonfiguration geladen. Diese enthält die Kamerasytemidentifikation, die Aufnahmefeldauflösung sowie die Anzahl der zu erfassenden Bilder pro Sekunde. Bei der Initialisierung der Klasse *VideoCapture* kann mittels eines Identifikationswerts die gewünschte Webcam ausgewählt werden. Dieser Wert wird vom Betriebssystem vorgegeben. Wenn nur eine Kamera angeschlossen ist, hat sie den Wert 0. Mit dem *set()*-Befehl kann als nächstes die Bildauflösung festgelegt werden. Für diese Arbeit sind 640x360 px das Optimum, da höhere Auflösungen eine höhere Prozessorauslastung verursachen und daher eine geringere Anzahl an Bildern pro Sekunde verarbeitet werden können. Zwar würde eine noch kleinere Auflösung als 640x360 px die erwähnten Parameter positiv beeinflussen, doch wäre das Bild in der Ausgabe für den Benutzer dann sehr klein. Das Bild nach der Aufnahme für die Verarbeitung zu verkleinern und die Ergebnisse für die Ausgabe entsprechend zu skalieren,

wäre auch eine Möglichkeit, welche allerdings zusätzliche Rechenzeit für die Skalierung benötigen würde.

Kann die Kamera mit den vorhin festgelegten Einstellungen geöffnet werden, so wird in einer Programmschleife ein Bild aufgezeichnet und in der Variable *capturedImage* abgelegt. Wenn diese Bildmatrix Daten enthält, dann wird das Bild für die Verarbeitung durch andere Verfahren des Programms in *imageData* abgespeichert. Die Bilderfassung wird nicht gestartet, wenn keine Kamera aufgerufen werden kann, weil dadurch auch keine Bilder erfasst werden können. Das autonome Fahrzeug kann in diesem Fall nicht fahren und der Thread wird mit einer Fehlermeldung beendet. Der Benutzer müsste die Verbindung der Kamera zum System sowie den Identifikationswert der Kamera in der Konfigurationsdatei überprüfen. Danach kann das Modul neu gestartet und verwendet werden.

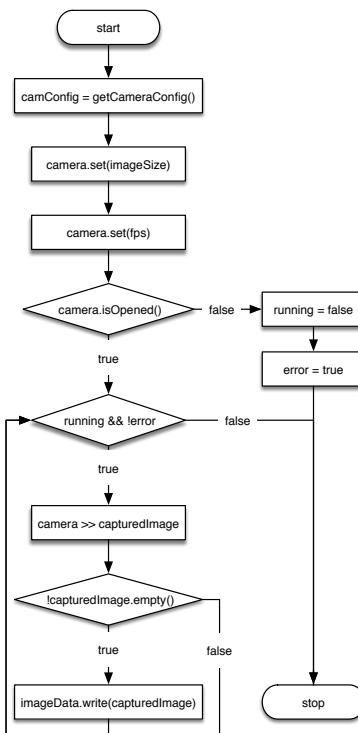


Abbildung 48: Flussdiagramm für das Bilderfassungsmodul

5.3.2 Fahrbahnerkennungsmodul

Im Wesentlichen führt das Modul zur Fahrbahnerkennung in dieser Arbeit die in Kapitel 2.2.1 beschriebenen Schritte durch. Der Ablauf dazu wird in Abbildung 49 dargestellt. Für den Start des Moduls *LaneDetector* werden die Eingabebildmatrix in *inputImage* vom Datentyp *ImageData*, für die Speicherung der Ausgabebilder *outputImage* ebenfalls vom Typ *ImageData* und für das Abspeichern der erkannten Fahrbahndaten *lane* als *LaneData* benötigt. Dann werden zuerst die Kamerakalibrierungseinstellungen geladen. Diese sind insbesondere für die inverse Perspektivtransformation notwendig. Ein weiterer wichtiger Schritt, bevor die Erkennung durchgeführt werden kann, ist die Initialisierung eines *Kalman*-Filters pro Straßenmarkierung. Hier wird auf die gleichnamige OpenCV-Implementierung zurückgegriffen. Der Filter wird sowohl zur Auswahl passender

Linien, welche die Fahrbahnmarkierung beschreiben, als auch zur Glättung der Positionsdaten dieser Linien verwendet. Für die inverse Perspektiventransformation wird als nächstes die Homographiematrix aus den Konfigurationsdaten geladen. Stehen keine Homographiedaten zur Verfügung, so wird der Thread mit einer Fehlermeldung beendet. Der Benutzer muss erneut eine extrinsische Kamerakalibration durchführen. Erst dann kann der Thread wieder gestartet werden.

In einer Programmschleife wird ein Eingabebild eingelesen und mit der *OpenCV*-Funktion *warpPerspective()* transformiert. Der nächste Schritt ist es, das Bild für die Markierungserkennung mit *prepareImage()* vorzubereiten. Vor jeder Bildaufnahme führt die Kamera eine automatische Belichtungseinstellung durch. Das Ergebnis lässt sich allerdings noch optimieren, um die helle Straßenmarkierung deutlicher vom Untergrund zu unterscheiden. Durch die Skalierung der Helligkeitswerte des Bildes über den gesamten Wertebereich, also zwischen 0 und 255 Helligkeitsstufen, werden dunkle Bereiche dunkler und helle Bereiche heller. Dafür wurde in dieser Arbeit die Funktion *autoAdjustBrightness()* geschrieben. Eine weitere im Rahmen dieser Arbeit entwickelte Funktion *whiteColorFilter()* konvertiert zuerst das Farbbild der Kamera in ein Graustufenbild und filtert das Bild, sodass nur noch weiße Bildbereiche zu sehen sind. Eine morphologische Bildoperation, die mit der *OpenCV*-Funktion *erode()* durchgeführt werden kann, verbessert das Ergebnis der Filterung für die spätere Detektion der Fahrbahnmarkierung. Sie lässt alle zusammenhängenden weißen Flächen schrumpfen. Kleinere weiße Bildstellen werden dadurch stark reduziert, sodass sie die darauffolgende Markierungserkennung weniger stören.

In diesem mehrfach gefilterten Graustufenbild wird im nächsten Algorithmuschritt mit der Funktion *detectLines()* nach linienförmigen Markierungen gesucht. Die *OpenCV*-Funktion *Canny()* wendet den besprochenen *Canny*-Algorithmus aus Kapitel 2.2.1 auf das Bild an. Das Ergebnis ist ein Binärbild, in welchem nur die detektierten Kanten durch helle Pixel markiert werden, der Hintergrund ansonsten jedoch dunkel ist. Diese Matrix wird mit dem ebenfalls in Kapitel 2.2.1 beschriebenen *Hough*-Verfahren transformiert, um Linien darin zu detektieren. *OpenCV* bietet dafür die Funktion *HoughLinesP()* an. Dabei handelt sich um eine effiziente Implementierung der probabilistischen *Hough*-Transformation nach Matas et al. [MGK00]. Ein besonderer Vorteil dieser Funktion ist, dass auch unterbrochene Linien damit erkannt werden können. Das Resultat der Transformation ist ein Vektor mit den Anfangs- und Endpunkten aller erkannten Linien, welcher im Vektor *lines* abgelegt wird. In diesem *Hough*-Verfahren kann es vorkommen, dass für ein und dieselbe Fahrbahnmarkierung mehrere nebeneinanderliegende Linien, nämlich deren Außenkanten, detektiert werden. Das macht ein Aussortieren notwendig, sodass nur je eine Linie pro Markierung übrigbleibt. In dieser Arbeit wurde die Strategie gewählt, die beiden Linien auszuwählen, die am nächsten zur vertikalen Bildmitte liegen. Der Grund dafür ist, dass sich das Fahrzeug in der Bildmitte am unteren Rand der Aufnahme befindet. Ist eine Linie in der linken bzw. rechten Bildhälfte zu sehen, dann wird hier angenommen, dass es sich dabei um eine linke bzw. rechte Fahrbahnmarkierung handelt.

Jede Markierung wird im hier erklärten Verfahren durch zwei Punkte beschrieben, wodurch für jedes Bild jeweils vier Punkte für zwei Markierungen bestimmt werden. Durch Erkennungsfehler oder Markierungsunterbrechungen ist es möglich, dass das Fahrzeug in

der Vorwärtsbewegung auch falsche oder stellenweise gar keine Linien erkennt. Damit würden die errechneten Punkte, an denen sich die Trajektorie orientiert, entweder plötzlich die Position wechseln oder gänzlich fehlen. Das würde dazu führen, dass sich auch die Trajektorie, von Bild zu Bild sprunghaft verändern und das autonome Fahrzeug instabil fahren würde. Mit dem oben besprochenen *Kalman*-Filter kann das Verfahren jedoch stabilisiert werden. Dazu werden die linke und die rechte Markierung mit *predictLine()* einzeln geglättet. In dieser Funktion wird zuerst der Prognosevorgang *predict()* des *Kalman*-Filters ausgeführt. Anschließend werden die erfassten Linienelemente mit *correct()* dem Filter zur Verbesserung der Prognose hinzugefügt. Diese liefert anschließend geschätzte Koordinaten für alle eingegebenen Punkte. Nach mehreren Iterationen wird diese Schätzung immer genauer und gegen falsch erkannte oder fehlende Linien robuster. Die Ergebnisse dieser Schätzung werden für die weitere Verarbeitung in der Fahrplanung in die Fahrbahndatenstruktur *lane* gespeichert. Für das Monitoring werden die erkannten Markierungen in einem Ausgabebild eingezzeichnet. Die Programmschleife wird solange ausgeführt, bis ein Beendigungsbefehl mit der Funktion *quit()* vom jeweiligen aufrufenden Modus übermittelt wird.

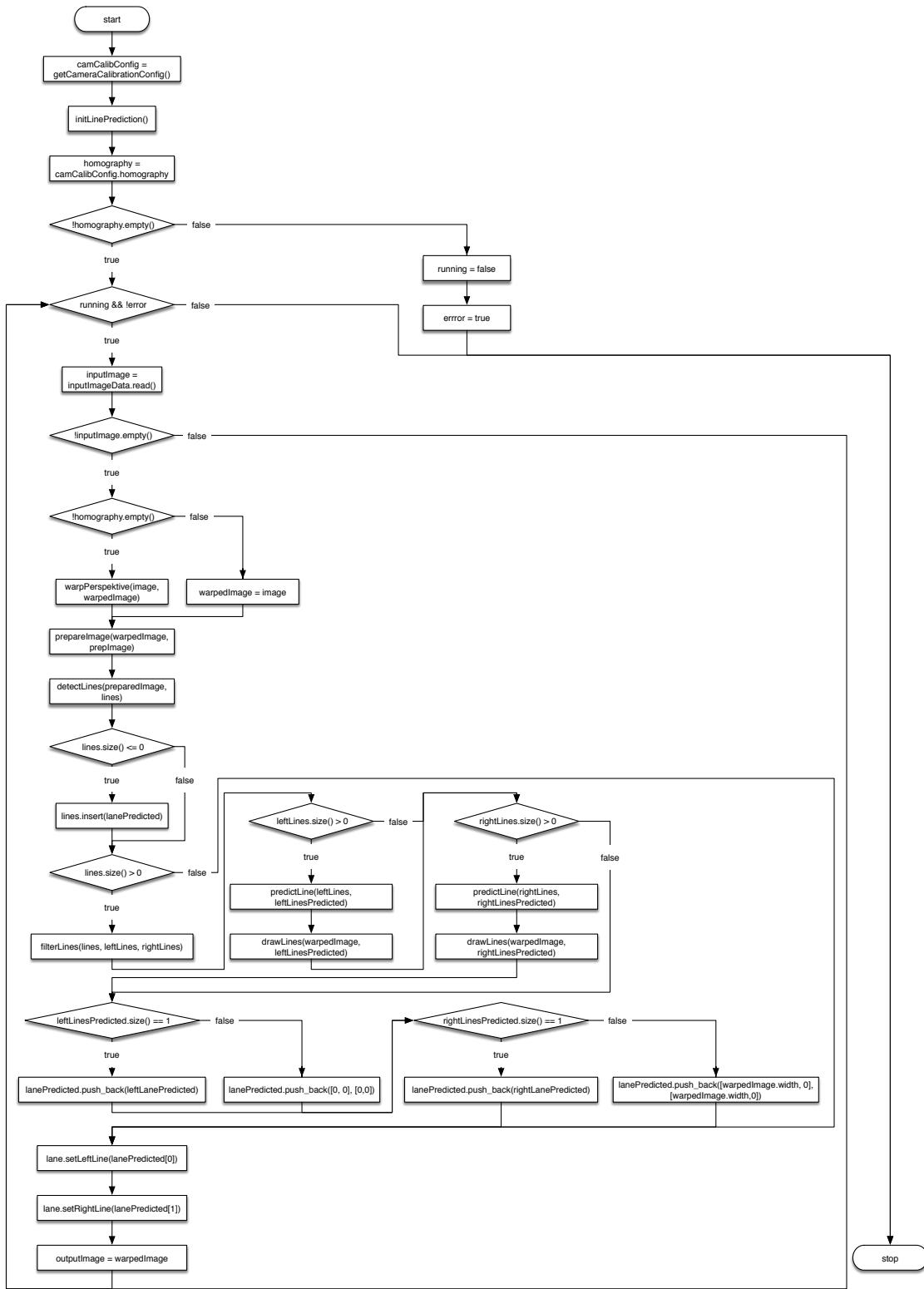


Abbildung 49: Flussdiagramm des Moduls zur Fahrbahnerkennung

5.3.3 Stoppschilderkennungsmodul

Verkehrsschilder können mit Objekterkennungsverfahren in einem Eingabebild detektiert werden. In Kapitel 2.2.2 wurde die *Haar-Klassifizierung* als ein mögliches Verfahren vorgestellt, das auch auf einer Computerplattform mit beschränkten Ressourcen eingesetzt werden kann. In dieser Arbeit wurde ein Klassifizierer von Gaynor [G18b] gewählt, welcher bereits für Stoppschilder trainiert ist. Dieser muss als XML-Datei in das *CascadeClassifier*-Objekt *stopSignCascade* geladen werden. Für den Start des Moduls *TrafficSignDetector* werden wieder die Eingabebildmatrix in *inputImage* als *ImageData*-Typ, für die Speicherung der Ausgabebilder *outputImage* als *ImageData* und für das Abspeichern der erkannten Verkehrsschilder *trafficSignData* als *TrafficSignData* benötigt. Der Ablauf des Moduls zur Verkehrsschilderkennung wird in Abbildung 50 gezeigt. Nach dem Laden der Konfigurationsinformationen der Kamerakalibration und der Verkehrsschilderkennung wird überprüft, ob eine XML-Datei mit einem trainierten Klassifizierer verfügbar ist. Sie wird benötigt, um die weitere Erkennung von Stoppschildern durchzuführen. Wird diese Datei nicht gefunden, dann wird das Modul zur Verkehrsschilderkennung mit einer entsprechenden Fehlermeldung beendet. Der Benutzer muss dem Modul eine gültige Klassifiziererdatei übergeben. Erst dann kann der Thread ausgeführt werden.

Auch in diesem Modus wird eine Programmschleife ausgeführt. Darin wird das aktuelle Kamerabild geladen. Dieses muss für das Erkennungsverfahren ebenfalls in ein Graustufenbild konvertiert werden. Mit der OpenCV-Funktion *detectMultiScale()* wird im Bild nach Stoppschildern gesucht. Das Ergebnis wird als Vektor *stopSigns* mit rechteckförmigen Markierungen an den Stellen im Bild abgespeichert, an denen Stoppschilder gefunden wurden. Ist in dem Vektor mindestens eine Stoppschildmarkierung abgelegt, dann kann der Abstand dessen zum Fahrzeug berechnet werden. In dieser Arbeit wird ein Stoppschild mit einer immer gleichbleibenden Größe von 3x3 cm verwendet. Für die erstmalige Erkennung wird dieses mit einem Sicherheitsabstand von 20 cm vor dem stehenden Fahrzeug platziert. Nach der Detektion durch das hier beschriebene Verfahren, werden die Seiten der Rechtecke, welche die Position des Schildes im Bild markieren, gemessen. Die Abmessungen des Rechtecks im Bild entsprechen einer Größe von 35x30 px. Diese Pixelzahl wird als Referenz zur weiteren Distanzabschätzung während der Fahrt verwendet. Je näher das Fahrzeug dem Stoppschild kommt, desto länger werden die Rechteckseiten der Markierungen im Bild. Wenn das Rechteck größer als der Referenzwert ist, so befindet das Verkehrsschild bereits innerhalb des Sicherheitsabstandes des Fahrzeugs. Das Ergebnis der Verkehrsschilderkennung wird zum Monitoring auch graphisch im Ausgabebild wiedergegeben, wo die erkannten Verkehrsschilder ebenfalls mit einer rechteckigen Umrandung markiert werden. Auch wird das Ergebnis in *trafficSignsData* abgespeichert, worauf die Fahrplanung zugreifen kann. Die Programmschleife wird solange wiederholt, bis der aufrufende Modus das Modul mit der Funktion *quit()* beendet.

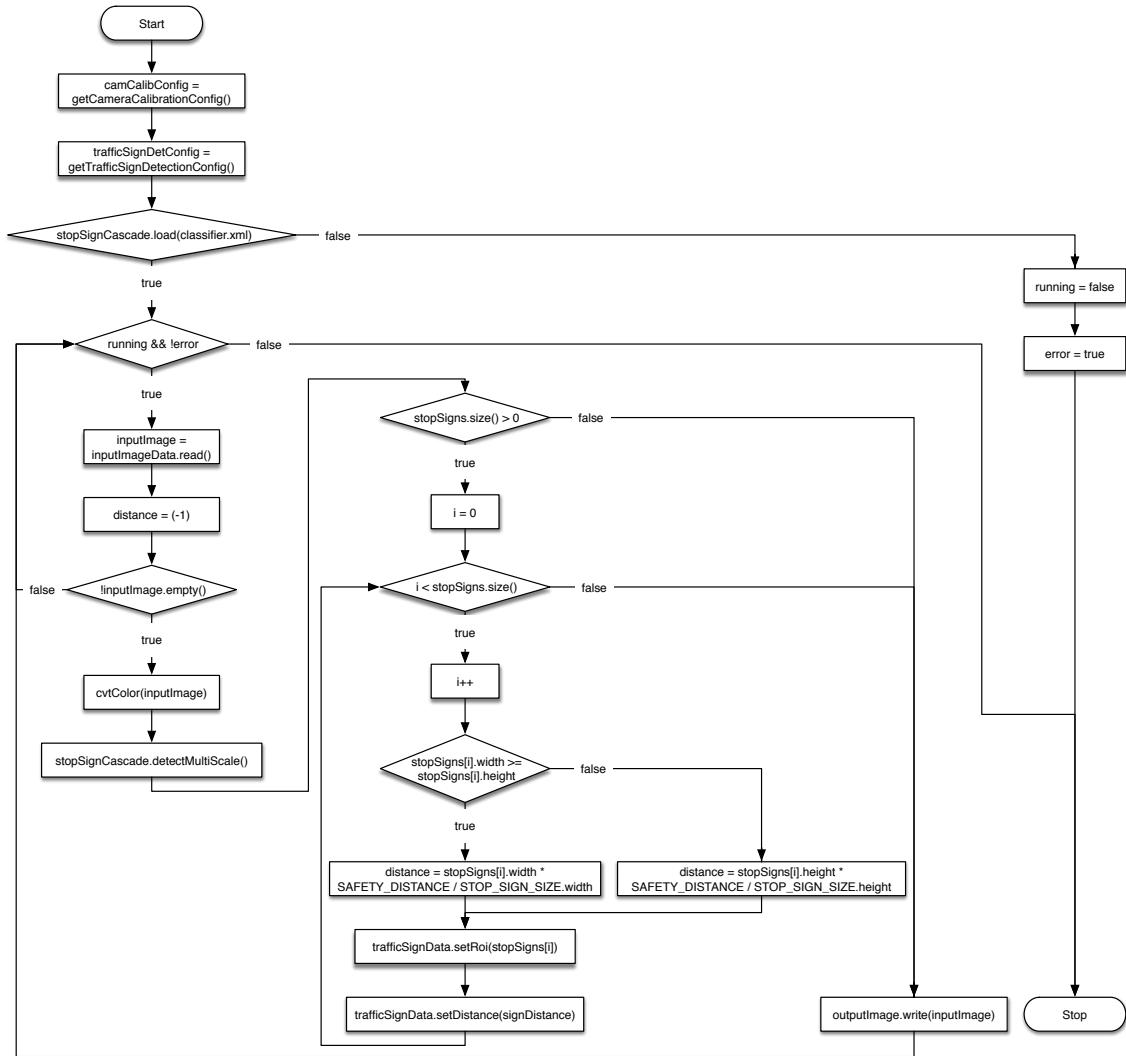


Abbildung 50: Flussdiagramm zum Modul für die Verkehrsschilderkennung

5.3.4 Hinderniserkennungsmodul

Die bisher besprochenen Module verarbeiten die Ausgabe der USB-Kamera. Neben der Erkennung der Fahrbahn und der Verkehrsschilder können mit entsprechenden digitalen Bildverarbeitungsverfahren auch Hindernisse auf der Fahrbahn erkannt werden. Für das autonome Fahrzeug ist jedoch weniger relevant, um welches Objekt es sich bei dem Hindernis handelt. Von Interesse ist der Abstand zu dem jeweiligen Hindernis, da rechtzeitig davor abgebremst werden muss. Allerdings enthalten Bilder, die über eine monokulare Kamera aufgenommen werden, keine Tiefeninformationen. Der Abstand könnte nur über die Hindernisgröße berechnet werden, weshalb zuvor eine Objekterkennung durchgeführt werden müsste. Das würde die Rechenkapazitäten des hier eingesetzten Embedded-Boards aber über Gebühr beanspruchen. Der Ultraschallsensor hingegen ermöglicht eine Distanzmessung zu jedem Hindernis vor dem Fahrzeug, ohne erkennen zu müssen, um welches Objekt es sich genau handelt.

Im Modul *ObstacleDetection* wird die Ultraschalldistanzmessung durchgeführt. Für den Start des Moduls wird für das Abspeichern der gemessenen Abstände die Datenstruktur *obstacleData* als *ObstacleData*-Typ benötigt. Zum Start des Modulablaufs wird die Hinderniserkennungskonfiguration in *obstacleDetConfig* geladen. Abbildung 51 zeigt den Ablauf des Hinderniserkennungsmoduls. Im nächsten Schritt werden die GPIO-Pin-Nummern, mit denen das Trigger- und das Echosignal verbunden sind, bestimmt. Für die Verwendung der Pins wird die *WiringPi*-Bibliothek benutzt, die unter *Raspbian* bereits vorinstalliert ist. Für den Ultraschallsensor dieser Arbeit bietet Aflak eine simple Bibliothek [A18b] an, die hier eingesetzt wird. Kann *WiringPi* nicht gestartet werden, so wird das Modul mit einer Fehlermeldung beendet. Der Benutzer muss die *WiringPi*-Installation überprüfen, um den Thread wieder ausführen zu können. Wird *WiringPi* initialisiert, wird auch der Ultraschallsensor aufgerufen.

In einer Schleife wird alle 250 ms die Distanz zu möglichen Objekten vor dem Sensor mit der Funktion *distance()* gemessen und gespeichert. Für die Zeitmessung innerhalb dieser Funktion wird die *delay()*-Funktion der *WiringPi*-Bibliothek genutzt, die wiederum die POSIX-Funktion *nanosleep()* aufruft. An dieser Stelle sei darauf hingewiesen, dass diese Zeitmessung vom Betriebssystem nicht präzise eingehalten werden kann. Sie ist mehr als eine Mindestzeitspanne zu verstehen, die abgewartet wird, bis der Ablauf fortgesetzt werden kann. Das liegt an der nicht deterministischen Arbeitsweise von *Linux*, die in Kapitel 3.6 beschrieben wird. Die Genauigkeit, mit der das System hier arbeitet, ist für diese Applikation allerdings ausreichend.

Die Distanzmessung selbst wird so durchgeführt, dass, nachdem ein Pulssignal von 10 µm am Trigger-Ausgang generiert wurde, nicht länger als 250 ms auf ein Signal am Echo-Eingang gewartet wird. Diese Zeitschranke soll verhindern, dass falsche Werte gemessen werden und dass der Messvorgang unnötig lange Rechenzeiten in Anspruch nimmt. Die Pause zwischen den einzelnen Messungen soll auch verhindern, dass unerwünschte Reflexionswellen aufgezeichnet werden.

Das Ergebnis der Distanzmessung wird in der Datenstruktur *obstacleData* für weitere Verarbeitungsschritte abgespeichert. Die Programmschleife wiederholt sich solange, bis die Modulfunktion *quit()* vom aufrufenden Modus ausgeführt wird.

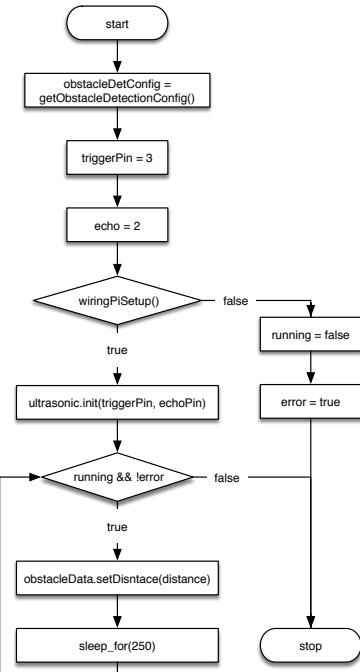


Abbildung 51: Flussdiagramm des Moduls zur Hinderniserkennung

5.4 Fahrplanungsmodul

Wurde die Umgebung, wie in Kapitel 5.3 beschrieben, wahrgenommen, folgt als nächstes die Planung der Fahrt. Der Fahrspurverlauf kann anhand der erkannten Straßenmarkierungen bestimmt werden. Das autonome Fahrzeug soll dazu mittig zwischen diesen fahren. Wurde ein Stoppschild erfasst, soll das System, wie gesetzlich vorgeschrieben, den Prototyp anhalten. Dasselbe geschieht auch, wenn ein Hindernis im entsprechenden Abstand vor dem Fahrzeug wahrgenommen wird. Um das Modul *PathPlanner* aufzurufen, wird erneut eine Eingabebildmatrix *inputImage* als *ImageData*, eine Verkehrsschilddatenstruktur *trafficSigns* als *TrafficSignData*, eine Hinderniserkennungsdaten *obstacles* als *ObstacleData*, Fahrzeugdaten *vehicle* als *VehicleData* und eine Datenstruktur zur Abspeicherung der hier berechneten Trajektorie *trajectory* als *TrajectoryData* benötigt. In Abbildung 52 ist der Verlauf dieser Verarbeitungsroutine dargestellt.

Das Planungsmodul benötigt die Einstellungsinformationen der Kamera. Wie in Kapitel 3.5.6 beschrieben, soll die Trajektorie von der Fahrplanung als Linienverlauf gespeichert werden. Auch hierfür wird ein *Kalman*-Filter initialisiert. Diese Filterung der Trajektorie führt zu einer Stabilisierung des Fahrverhaltens. In einer Schleife wird dann der aktuelle Fahrbahnverlauf, welcher von der Fahrbahnerkennung gespeichert wurde, geladen.

Vor der Kalkulation der Trajektorie wird der Sicherheitsabstand zu möglichen Objekten vor dem Fahrzeug mittels des Ultraschallsensors überprüft. Da in dieser Arbeit eine geringe

Motordrehzahl für den Vortrieb des Modellfahrzeugs gewählt wird, beträgt der Bremsweg etwa 20 cm, wenn der Motor abgeschaltet wird. Daher wird an dieser Stelle im Programm ein Sicherheitsabstand von 25 cm überprüft. Bei einer variablen Geschwindigkeit müsste hier der Sicherheitsabstand entsprechend angepasst werden, sodass das Fahrzeug innerhalb dessen sicher anhalten kann. Wenn das Hinderniserkennungsmodul nicht aktiv ist, dann sind die Abstandswerte in *obstacle* gleich 0. Da dieser Wert kleiner als der Sicherheitsabstand ist, bleibt das Fahrzeug auch in diesem Fall stehen. Dies ist eine Sicherheitsvorkehrung, damit das autonome Fahrzeug nicht ohne aktive Hinderniserkennung fahren kann. Im nächsten Schritt wird in der Schleife überprüft, ob ein Stoppschild erkannt wurde und in welcher Distanz es vor dem Fahrzeug steht. Ist die längere Rechteckseite der Markierung eines erkannten Stoppschilds kleiner als die in Kapitel 5.3.3 besprochene Referenz, so ist das Schild in einer größeren Distanz vor dem Fahrzeug und dieses kann weiterfahren. Ist die genannte Seite gleich groß oder größer, so muss das autonome Fahrzeug eine Bremsung einleiten, da sich das Stoppschild hier bereits innerhalb des Sicherheitsabstands vor dem Fahrzeug befindet. Ist jedoch innerhalb des Sicherheitsabstands vor dem Fahrzeug weder ein Hindernis noch ein Stoppschild zu erkennen und sind Daten für die linke sowie die rechte Straßenmarkierung vorhanden, dann kann die Fahrtroute mit der Funktion *calcTrajectory()* berechnet werden. Darin wird zuerst überprüft, ob die Fahrbahn durch eine linke bzw. eine rechte Markierung begrenzt wird. Sind beide Markierungen vorhanden, so wird die Fahrbahnmitte berechnet und, wie in Kapitel 5.3.2 beschrieben, mit *predictLine()* gefiltert. Ist nur eine Markierung vorhanden, so wird diese als Fahrbahnmitte angenommen. Der Grund für die hier entwickelte Strategie liegt in der Möglichkeit, eine zweite Markierung in der Nähe der erkannten zu finden oder mögliche Erkennungsfehler zu korrigieren. Befindet sich das Fahrzeug links oder rechts neben einer markierten Fahrbahn, so kann die Kamera nur eine der beiden Begrenzungslinien erfassen. Lenkt das Fahrzeug auf diese zu, so taucht im Kamerabild die zweite Linie auf. Werden anschließend beide Markierungen erkannt, dann kann die Mitte zwischen diesen bestimmt werden. Das autonome Fahrzeug ist damit fähig, die Trajektorie zu korrigieren, auch wenn es kurzzeitig die Fahrbahn verlassen hat. Es kann aber auch einer Fahrbahn folgen, welche nur durch eine Markierung begrenzt ist. Wird keine Fahrbahn erkannt, hält das autonome Fahrzeug an. Die in der Funktion *calcTrajectory()* berechnete Trajektorie wird für die Weiterverarbeitung in der Datenstruktur *trajectory* abgelegt. Die Programmschleife kann vom aufrufenden Modus mit der Modulfunktion *quit()* beendet werden.

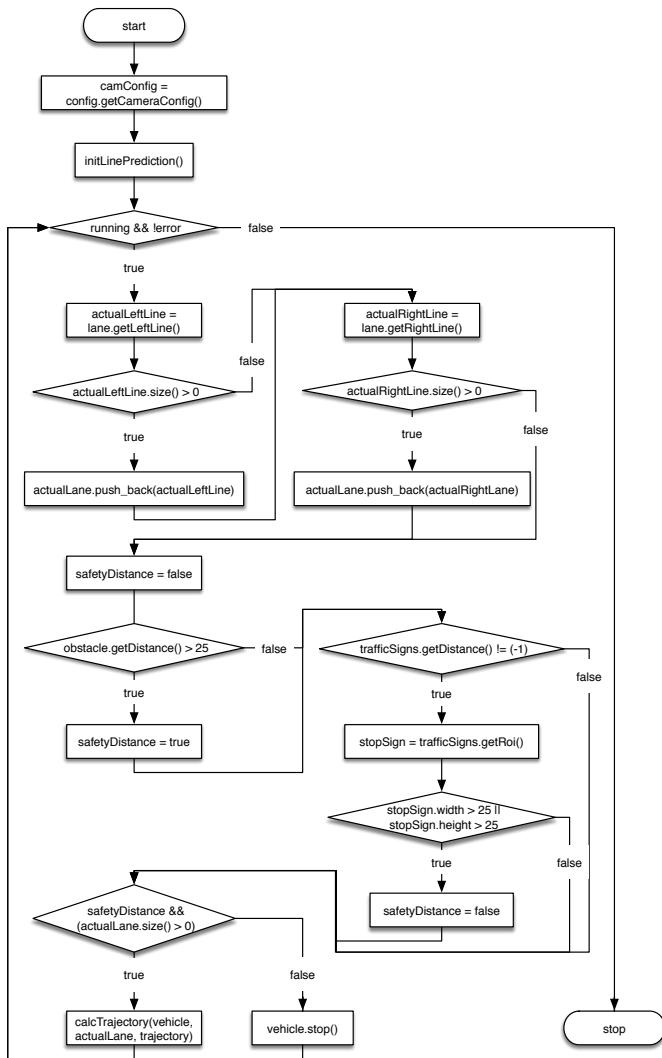


Abbildung 52: Flussdiagramm des Moduls zur Fahrplanung

5.5 Fahrzeugsteuerungsmodul

Das Fahrzeugsteuerungsmodul wandelt den Trajektorienverlauf, der in der Fahrplanung im vorangegangenen Kapitel ermittelt wurde, in konkrete Werte zur Ansteuerung des Fahrzeugs um. Dazu wird die Lage der Trajektorie in Relation zur Längsachse des Fahrzeugs bestimmt. Durch die Kamerapositionierung und die erfolgreiche extrinsische Kalibration kann angenommen werden, dass sich das autonome Fahrzeug mittig am unteren Bildrand befindet. Das gilt sowohl für die unverarbeitete Kameraaufnahme, wie auch für das perspektivisch transformierte Bild. Für den Aufruf des *VehicleController*-Moduls wird die Trajektorie *trajectory* als *TrajectoryData*-Typ und die Fahrzeugdaten *vehicle* als *VehicleModel*-Typ benötigt. Das Flussdiagramm des im Folgenden beschriebenen Moduls *VehicleController* ist in Abbildung 53 dargestellt.

Am Beginn des Modulablaufs werden Kamerakonfigurationsdaten geladen und der Motortreiber mit der Funktion `init()` initialisiert. Teil davon ist es, die MCU zu starten. Wie in Kapitel 4.4 erklärt, erwartet diese nach dem Start ein neutrales Ansteuerungssignal, also einen PWM-Wert von 307 Samples. Schlägt die Initialisierung fehl, weil zum Beispiel kein Treiber angeschlossen ist, dann wird das Modul mit einer Fehlermeldung beendet, ohne einen Fahrbefehl auszugeben. Der Benutzer muss den Anschluss zum Motortreibers überprüfen. Wird der Treiber anschließend erfolgreich initialisiert, dann wird eine Programmschleife gestartet.

In dieser Schleife wird überprüft, ob in der Fahrzeugdatenstruktur `vehicle` ein Not-Stopp ausgelöst wurde. Dieser Not-Stopp setzt den Motortreiber zurück, wodurch die Ansteuerung aller Motoren unterbrochen wird und das Fahrzeug dadurch zum Stehen kommt. Um die Sicherheit des Systems zu erhöhen, ist es möglich, diesen in allen Modulen, welche Zugriff auf die Fahrzeugdaten `VehicleModel` oder den Motortreiber haben, auszulösen. Der nächste Schritt ist es, zu überprüfen, ob in den Eingabedaten eine Trajektorie mit mindestens zwei Punkten verfügbar ist. Im Weiteren wird ausschließlich eine Gerade zwischen diesen ersten beiden Punkten des Trajektorienvektors betrachtet. Diese Trajektoriengerade bildet einen Winkel θ zur vertikalen Bildachse. Dieser wird mit $\theta = \arctan\left(\frac{y_{end} - y_{start}}{x_{end} - x_{start}}\right)$, wobei (x_{start}, y_{start}) der Startpunkt und (x_{end}, y_{end}) das Ende der Trajektorie im jeweiligen Bild ist, kalkuliert. Die Funktion `getTheta()` implementiert diese Berechnung. Ist der Winkel θ kleiner als 10 % oder größer als 90 % von π , so steht das Modellauto annähernd normal zur Fahrbahn bzw. zu einer Markierung. Eine Weiterfahrt ist in diesem Fall aus den beiden folgenden Gründen nicht sinnvoll. Zum einen könnte sich bei der Normalen um eine Stopplinie handeln, also ist ein Stehenbleiben ohnehin gesetzlich vorgeschrieben. Zum anderen kann das hier implementierte System ohne weitere Zieleingabe keine sinnvolle Entscheidung treffen, in welche Richtung weitergefahren werden soll. Sowohl das Abbiegen nach links als auch nach rechts könnte das Fahrzeug eventuell zurück auf die Fahrbahn bringen, aber die Fahrtrichtung könnte falsch sein.

Ist der berechnete Winkel $0 < \theta < (\frac{\pi}{2} * 0.9)$, so muss das Fahrzeug nach links lenken. Nach rechts wird gelenkt, wenn der Winkel $(\frac{\pi}{2} * 1.1) < \theta < \pi$ ist. In Versuchen wurde ersichtlich, dass der Lenkwinkel für die hier gefahrene Geschwindigkeit steiler sein muss, als der hier berechnete Winkel θ . Um den tatsächlichen Lenkwinkel α zu erhalten, wird hier daher für die Linkslenkung $\alpha_l = \theta - \frac{\pi}{8}$ bzw. für die Rechtslenkung $\alpha_r = \theta + \frac{\pi}{8}$ berechnet. Ist der Winkel $(\frac{\pi}{2} * 0.9) < \theta < (\frac{\pi}{2} * 1.1)$, also annähernd gleich ausgerichtet wie die vertikale Bildachse, dann wird für den Lenkwinkel $\alpha = \theta$ gewählt. Dadurch fährt das Fahrzeug gerade aus.

Der Lenkwinkel ist allerdings auch von der Fahrzeugposition im Verhältnis zur Position der Trajektorie einzustellen. Befindet sich Letztere zu weit links oder zu weit rechts von der Bildmitte, so ist ebenso das Fahrzeug in der gleichen Weise von der zu fahrenden Spur verschoben. In diesem Fall muss das Fahrzeug nicht mit der Trajektorie lenken, sondern zu ihr hin. Die horizontalen Differenzen $x_{diff_start} = x_{traj_start} - \frac{Bildbreite}{2}$ und $x_{diff_end} = x_{traj_end} - \frac{Bildbreite}{2}$, mit der X-Koordinate des Trajektorienstartpunkts x_{traj_start} und mit der X-Koordinate des Trajektorienendpunkts x_{traj_end} , geben Aufschluss über diese Verschiebung. Sind $x_{traj_start} > 10$ und $x_{traj_end} > 10$, so ist das Fahrzeug rechts von der

Fahrspur. Sind $x_{trajstart} > -10$ und $x_{trajend} > -10$, befindet sich das Fahrzeug links von der Fahrspur. Ziel ist es, die Fahrzeulgängsachse über der Trajektorie zu halten. Deshalb wird zur Positions korrektur der Lenkwinkel von $\alpha_l = \pi/4$ für links bzw. $\alpha_r = 3\pi/4$ für rechts gewählt. Damit steuert das Fahrzeug mit vollem Lenkeinschlag auf die Trajektorie zu, wenn es sich zu weit abseits von dieser befindet. Der Lenkwinkel wird in Bogenmaß an die Funktion `setSteering()` übergeben. Bei diesen Berechnungen handelt es sich um rudimentäre Fahrtrichtungsabschätzungen, welche auf der Basis von Versuchen entstanden sind. Für höhere Kurvengeschwindigkeiten müsste ein anderes Verfahren angewendet werden. Das ist jedoch nicht Ziel dieser Arbeit, da das vorliegende Modul nur beispielhaft zur Realisierung der gewünschten Entwicklungsplattform implementiert wurde. In einer späteren Weiterentwicklung des Prototyps, sollte dennoch ein leistungsfähigeres Verfahren eingesetzt werden.

Nach dem vorherigen Einstellungsschritt wird der Funktion `setAcceleration()` auch die Beschleunigung des Antriebsmotors in Prozentangaben übermittelt. Dieser Beschleunigungswert entspricht der Gaspedalstellung bei einem Fahrzeug in vollem Maßstab. Ist der Wert 100 %, so entspricht das dem maximalen Vorwärtsantrieb. Bei einem Wert von 0 % bleibt das Fahrzeug stehen. Die Rückwärtsfahrt wird über negative Werte geregelt, das heißt -1 % bis -100 %. Fährt der Prototyp gerade aus, dann wird die Beschleunigung auf 18 % gestellt. Dieser Wert wurde ebenfalls durch Versuche als Optimum für das hier umgesetzte Verfahren ermittelt. Dabei wurde darauf geachtet, dass der Benutzer das Fahrzeug im Notfall gehend erreichen kann und bei einem Unfall keine Schäden an der Umwelt oder am Fahrzeug entstehen. Bei den maximalen Lenkwinkeln, wird die Geschwindigkeit um 0,5 % reduziert. Dadurch wird auch die Kurve enger gefahren.

Der oben eingestellte Lenkwinkel und der Beschleunigungswert müssen als nächstes in entsprechende Steuerungsbefehle für den Motortreiber übersetzt werden. Das hier eingesetzte PCA9685-Modul wird über der I²C-Bus des *Raspberry Pis* angesteuert. Das PWM-Modul verfügt über 16 individuell ansprechbare Kanäle, wobei in diesem Projekt nur die ersten beiden für den Servomotor, also die Lenkung, und für die MCU, also den Antrieb, verwendet werden. Zur Ansteuerung des Moduls lassen sich zahlreiche quelloffene und freie Softwarebibliotheken finden. In dieser Arbeit wird die von Todorov entwickelte und von Wessman für C++ angepasste Version [T18b] sowie [W18b] gewählt und für diesen Prototyp adaptiert. Das Funktionsprinzip der Motorsteuerung wurde bereits in Kapitel 4.4 erläutert. Über die Angabe des Kanals und eines 12-Bit-Befehls kann diesem mitgeteilt werden, welchem Motor welcher PWM-Wert zugeschickt werden soll.

Wie in Kapitel 4.4 bereits erwähnt, erlauben die mechanischen Eigenschaften der Fahrzeuglenkung nur Lenkwinkel von 45° bis 135° bzw. von $\pi/4$ bis $3\pi/4$. Ein Lenkwinkel θ wird vom Bogenmaß mit $\alpha = \alpha_{min} + \frac{\theta}{\pi}$ in den Samplewert α umgerechnet. Die

Werte α_{max} bzw. α_{min} sind 409 bzw. 204 Samples. Sie entsprechen der Berechnung $\alpha_{max/min} = t_{max/min} \cdot f \cdot s$, mit $t_{max} = 2\text{ ms}$, $t_{min} = 1\text{ ms}$, $f = 50\text{ Hz}$ und $s = 4096\text{ Samples}$. Das Vor- und Rückwärtsfahren wird über den zweiten Kanal des PWM-Moduls gesteuert. Der passende Beschleunigungswert wird mit $a = a_{neutral} + \frac{a\%}{100}$

bestimmt, wobei $a_{neutral}$ der Samplewert der Neutralstellung, $a\%$ der Beschleunigungsprozentwert und a_{max} bzw. a_{min} die Samplewerte des Minimums bzw.

Maximums der Beschleunigung, also 409 und 204 Samples sind. Dadurch ist $a_{neutral} = a_{min} + \frac{a_{max}-a_{min}}{2}$.

Zu beachten ist, dass wie bei einem Fahrzeug in vollem Maßstab auch beim Modelfahrzeug dieser Arbeit eine Rückwärtsfahrt nach einer Vorwärtsfahrt nur in Folge eines Stillstands des Autos möglich ist. Die MCU lässt es anders nicht zu. Folgt in der Ansteuerung dennoch ein Rückwärtsfahrtbefehl direkt auf einen Vorwärtsfahrtbefehl, wird der Motor gestoppt und ein Neutralstellungsbefehl erwartet. Weiters sei angemerkt, dass die hier eingesetzte MCU die Energiezufuhr am Antriebsmotor in Abhängigkeit von den PWM-Werten regelt. Diese be- und entschleunigen das Fahrzeug zwar, lassen aber keine Rückschlüsse auf die Geschwindigkeit desselben zu. Ist die anzutreibende Last höher, so ist auch mehr Energie nötig, um diese zu bewegen. Daher ist kein linearer Zusammenhang zwischen der Energiezufuhr und der Geschwindigkeit gegeben und keine direkte Umrechnung möglich. Die tatsächliche Fahrzeuggeschwindigkeit lässt sich zum Beispiel über die Umdrehungsgeschwindigkeit der Räder mit $v = d \cdot \pi \cdot n$ berechnen, wobei v die Umdrehungsgeschwindigkeit, d der Raddurchmesser und n die Anzahl der Umdrehungen ist. In dieser Arbeit wurde auf eine genaue Geschwindigkeitserfassung verzichtet, weil sie für die Zielsetzung der Erstellung einer Entwicklungsumgebung nicht unmittelbar relevant ist. In einer weiteren Entwicklung und für die Implementierung eines dynamischen Fahrmodells ist sie allerdings notwendig.

Wird die Programmschleife durch den aufrufenden Modus mit der Modulfunktion `quit()` beendet, dann muss das autonome Fahrzeug vor dem Ende des Modulablaufs auf jeden Fall zum Stillstand kommen. Da nur dieses Modul für die Erteilung der Steuersignale an die Motoren zuständig ist, müssen am Ende des Moduls die Motoren in ihre Neutralstellung gebracht werden, sodass sie sich nicht mehr bewegen. Deshalb wird zu allerletzt auch der Motortreiber zurückgesetzt.

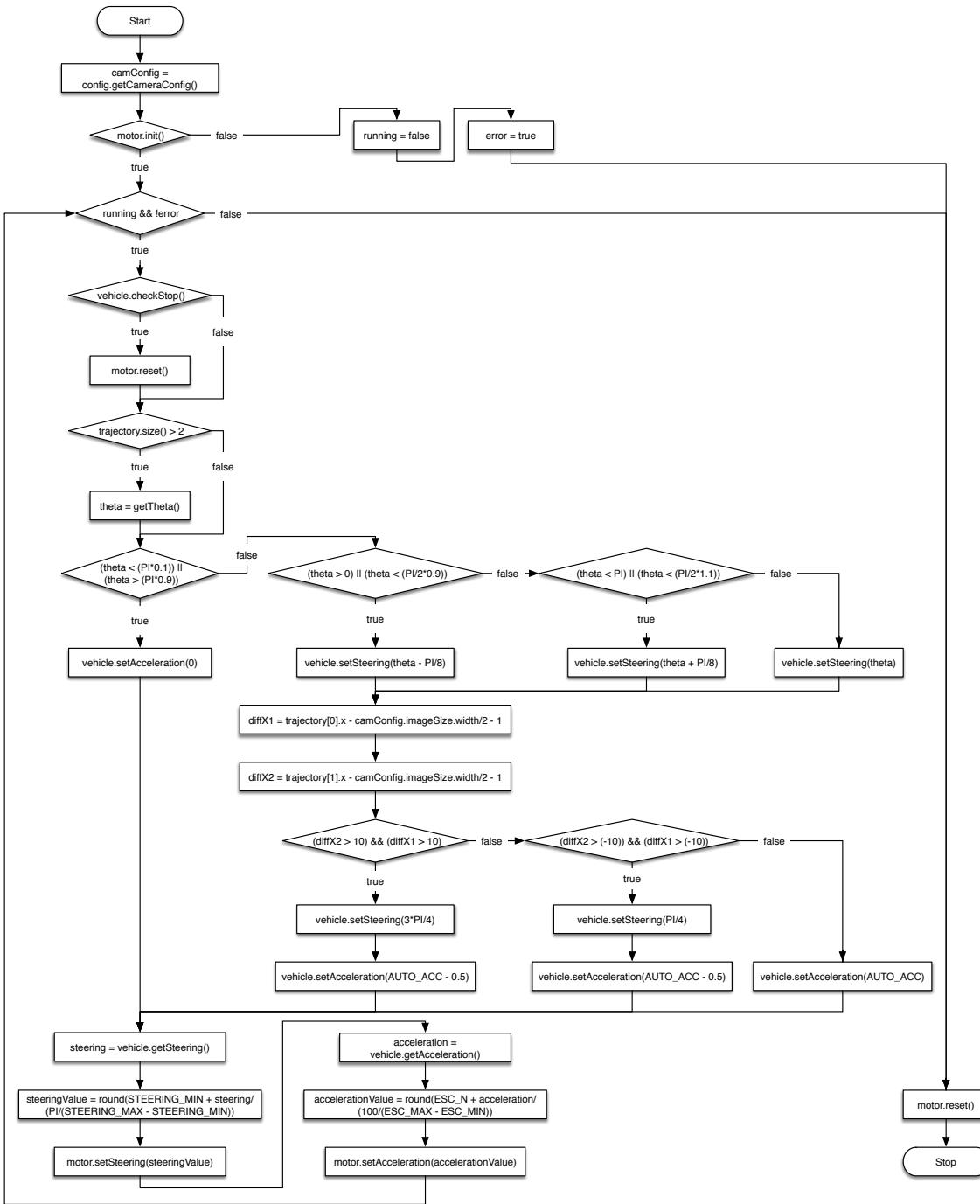


Abbildung 53: Flussdiagramm für das Modul der Fahrzeugsteuerung

5.6 Fernsteuerungsmodul

Neben dem autonomen Fahrbetrieb wurde in dieser Arbeit ein weiteres Modul zur Fahrplanung implementiert. Ein manueller Fahrbetrieb erlaubt es dem Benutzer, das Fahrzeug fernzusteuern. Dies ist dann notwendig, wenn der Prototyp aus einem der in den vorhergehenden Kapiteln genannten Gründen stehen bleibt und dem System eine selbstständige Weiterfahrt nicht möglich ist. Auch ist dieses Modul nützlich, wenn sich der

Prototyp außerhalb der physischen Reichweite des Benutzers befindet und auf einer Teststrecke manövriert werden muss. Als Interaktionsschnittstelle zwischen dem System und dem Benutzer wird hier die Tastatur gewählt. Auch wenn die meisten Tastaturen über Pfeiltasten verfügen, gibt es doch Unterschiede in der Interpretation dieser. Das liegt an den sogenannten *Scan-Codes*, die dem Betriebssystem übermitteln, welche Taste gedrückt wurde. Für Nicht-ASCII-Zeichen sind diese *Scan-Codes* nicht auf allen Tastaturen einheitlich kodiert. Aus diesem Grund werden im Folgenden die Tasten *W*, *A*, *S* und *D* zur Fernsteuerung verwendet. Sie befinden sich auf den meisten Tastaturen an derselben Position.

Das *RemoteController*-Modul benötigt die Fahrzeugdaten *vehicle* als Datentyp *VehicleModel* und die aktuelle Benutzereingabe über die Benutzerschnittstelle *uiState* als Typ *UserInterfaceState*. Der Algorithmus dieses Moduls ist in Abbildung 54 als Flussdiagramm dargestellt. Das Modul manipuliert ausschließlich den Lenkwinkel und die Akzeleration des Fahrzeugs. Die Umwandlung dieses Lenkbefehls in Motorsteuerbefehle wird im bereits besprochenen Fahrzeugsteuerungsmodul durchgeführt. Am Beginn des Ablaufs wird der Lenkwinkel auf $\pi/2$, was einer Geradeauslenkung entspricht, und die Motorbeschleunigung auf 0 % gesetzt. In einer Schleife wird die aktuelle Tastatureingabe mit *getKey()* geladen. Wird keine Taste vom Benutzer gedrückt, so ist der Wert -1. Für den weiteren Programmablauf sind auch die aktuellen Werte des Lenkwinkels und der Beschleunigung notwendig. Die Funktionen *getSteering()* und *getAcceleration()* liefern diese.

Im nächsten Schritt werden die Tasteneingaben überprüft. Wird die Taste *W* gedrückt, so wird der aktuelle Beschleunigungswert um 0,5 % erhöht. Umgekehrt wird der aktuelle Wert um 0,5 % reduziert, wenn die Taste *S* gedrückt wird. Wird die Taste losgelassen, so ist der Tastenwert wiederum -1. Wird mehrmals hintereinander dieselbe Taste gedrückt, so wird die Geschwindigkeit jedes Mal weiter erhöht bzw. erniedrigt. Damit können variable Geschwindigkeiten gefahren werden. Ein sofortiges Bremsen des Systems wird von der Leertaste ausgelöst. Sie setzt die Beschleunigung, unabhängig von deren vorherigem Wert, auf 0 %. Es ist zu beachten, dass das Modelfahrzeug über einen, in Relation zum Gesamtgewicht, sehr leistungsfähigen elektrischen Bürstenmotor verfügt. Auch hohe Beschleunigungswerte werden mit kaum merkbaren Anlauf- bzw. Anfahrverzögerungen umgesetzt und das Fahrzeug schnellt sofort los. Deshalb werden hier Wertveränderungen pro Tasteneingabe von nur $\pm 0,5\%$ gewählt, um eine verzögerte Wirkung auf das Beschleunigungsverhalten zu erzielen. Im weiteren Programmablauf werden auch die Maximalwerte der Beschleunigung von $\pm 100\%$ auf $\pm 30\%$ gedrosselt.

Die Einstellung der Lenkung wird ähnlich durchgeführt, wie die der Beschleunigung. Die *A*-Taste reduziert den aktuellen Lenkwinkel um $\pi/8$ und die *D*-Taste erhöht diesen um $\pi/8$. Wie im vorherigen Kapitel besprochen, initiieren Werte, die kleiner als $\pi/2$ sind, eine Linkslenkung, während größerer Werte eine Rechtslenkung bewirken. Da sich die Räder mechanisch nicht mehr als $\pi/4$ in jede Richtung drehen lassen, wurden die Maxima hier auf $\pi/4$ für links bzw. $3\pi/4$ für rechts festgelegt.

Die in der Programmschleife festgelegten Lenk- und Beschleunigungswerte werden mit *setSteering()* und *setAcceleration()* in den *VehicleModel*-Daten gespeichert. Vor der Wiederholung der Schleife wird eine möglicherweise durch das Drücken der Leertaste

erfolgte Bremsung aufgehoben. Das Fahrzeug fährt jedoch nicht ohne die erneute Eingabe der Tasten *W* oder *S* im nächsten Schleifendurchlauf los. Wird die Schleife mit der Modul-Funktion *quit()* beendet, so wird vor dem Verlassen des Ablaufs die Motorsteuerung zurückgesetzt, wodurch das Fahrzeug stehen bleibt.

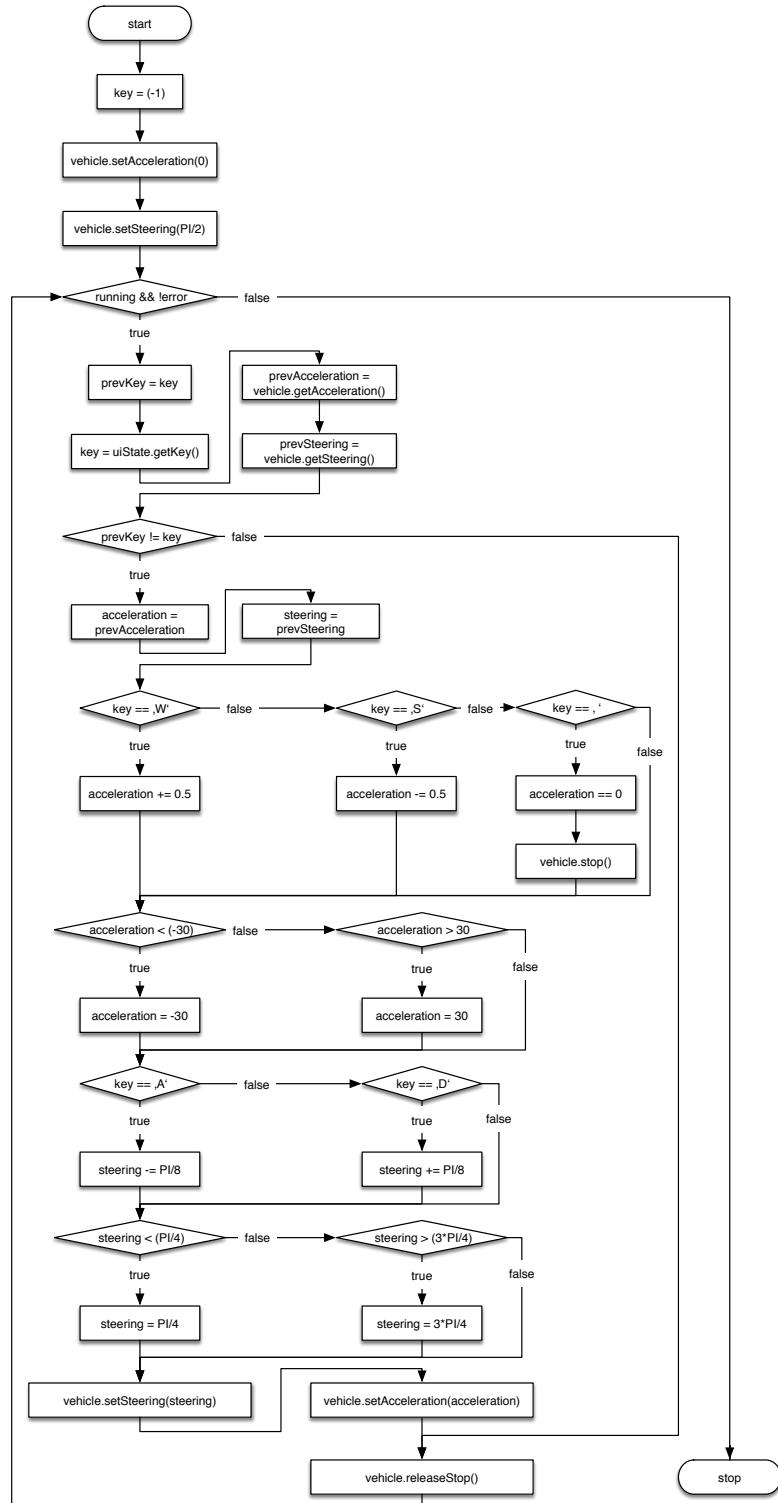


Abbildung 54: Flussdiagramm für das Fernsteuerungsmodul

5.7 Benutzerschnittstellenmodul

Auch wenn der hier beschriebene Prototyp autonom fahren soll, benötigt der Benutzer dennoch eine Interaktionsmöglichkeit mit dem Fahrzeug. Dieser kann zwischen den Systemzuständen wechseln und diese auch überwachen. Eine für den Benutzer intuitive Form der Interaktion ist eine graphische Bedienoberfläche, welche auf einem Display ausgegeben wird. Diese besteht in der vorliegenden Arbeit aus einem Menü und der Anzeige des aktuellen Kamerabildes oder des bearbeiteten Bildes, die auf einem *Raspberry Pi*-Touchscreen bzw. einem Monitoring-Computer angezeigt werden. Jeder Systemmodus, in dem der Benutzer mit dem Fahrzeug interagieren kann, hat eine eigene graphische Schnittstelle. Die Realisierung dieser wird im Folgenden in Kapitel 5.8 gemeinsam mit dem jeweiligen Modus erläutert. Der Ablauf des Benutzerschnittstellenmoduls *UserInterface* ist in Abbildung 55 zu sehen. Für den Modulaufruf werden Eingabedaten *imageData* vom Datentyp *ImageData* und der Zustand der Benutzerschnittstelle *uiState* vom Typ *UserInterfaceState* benötigt.

Das hier entwickelte System lässt sich grundsätzlich sowohl über die Tastatur, als auch die Maus oder einen Touchscreen bedienen. Diese Funktionalität wird durch die in *OpenCV* integrierte Bibliothek *HIGHGUI* ermöglicht. Neben den Eingabemöglichkeiten für den Benutzer, ermöglicht diese Softwarebibliothek auch die graphische Ausgabe von Bildern, Formen und Farben. Mithilfe von *OpenCV*-Funktionen aus *HIGHGUI* hat Bevilacqua eine einfache Benutzeroberflächen-Bibliothek mit der Bezeichnung *CVUI* [B18b] entwickelt, welche in dieser Arbeit verwendet wird.

Nach dem Start des Moduls werden die aktuellen Konfigurationsdaten der Kamera geladen. Diese sind für die Bildanzeige notwendig. In einer Programmschleife wird das aktuelle Kamerabild aus *imageData* abgerufen. Sind keine Bilddaten vorhanden, dann wird ein einfarbig dunkelgraues Bild erstellt, welches dieselbe Auflösung wie ein Kamerabild hat.

Im nächsten Schritt wird eine weitere Bildmatrix *outputImage* mit einem ebenfalls dunkelgrauen Hintergrund erzeugt. In dieser werden in weiterer Folge alle graphischen Anzeigeelemente, wie Bilder und Bedienflächen platziert. Dazu wird das Kamerabild mittels der *CVUI*-Funktion *image()* in *outputImage* rechtsbündig platziert. Links davon werden in einem weiteren Schritt mit der *draw()*-Funktion des entsprechenden Benutzerschnittstellenmodus Schaltflächen und Textinformationen ausgegeben. Die Funktion liefert auch Informationen darüber, welche der von ihr gezeichneten Schaltflächen aktiviert wurde. Um das so zusammengesetzte Ausgabebild *outputImage* dem Benutzer anzuzeigen, wird die *CVUI*-Funktion *imshow()* aufgerufen. Ist jedoch kein Ausgabebild vorhanden, so bleibt die Anzeigefläche für dieses grau.

Die Benutzereingabe ist durch das Klicken mit der Maus auf Schaltflächen, aber auch über das Drücken bestimmter Tasten der Tastatur möglich. Die unterstrichenen Buchstaben der Schaltflächen, lösen deren Aktivierung aus. So kann auch mit der Tastatur durch das Programm navigiert werden. Zusätzlich wurde für diese Arbeit die Funktion *getConsoleInput()* entwickelt, die auch eine Eingabe der entsprechenden Tasten über die Console erlaubt. Wird keine Taste oder Schaltfläche gedrückt, dann hat die Benutzereingabe stets einen Zahlenwert -1. Der aktuelle Tastenwert wird mit der Funktion

`setKey()` in `uiState` abgespeichert. Die Programmschleife wird wie in den anderen Modulen mit der Funktion `quit()` durch den aufrufenden Modus beendet.

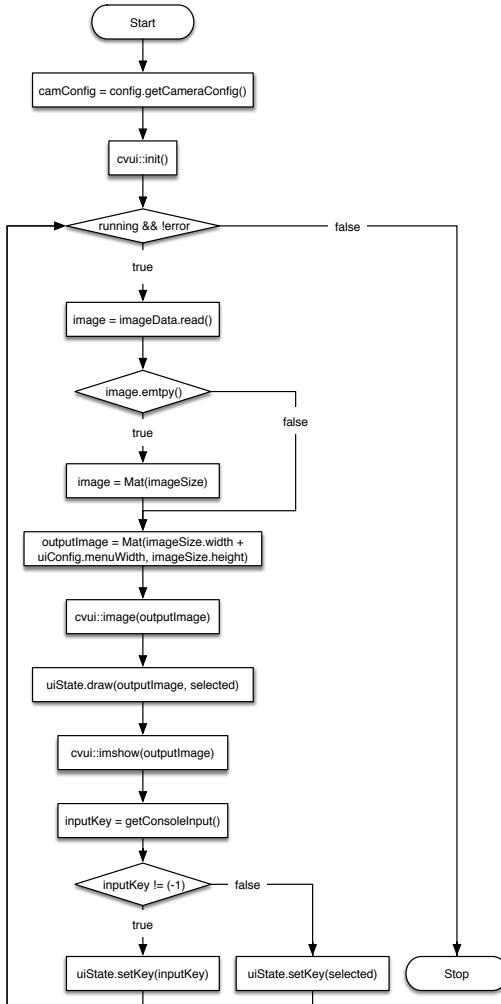


Abbildung 55: Flussdiagramm zum Benutzerschnittstellenmodul

5.8 Betriebsmodi

Die bisher beschriebenen Module werden für die Umsetzung der Systemzustände, welche in Kapitel 3.3 entworfen wurden, benötigt. Im Folgenden wird ein solcher Zustand auch als *Modus* bezeichnet. Die Realisierung der einzelnen Modi wird in den folgenden Kapiteln erläutert. Die Reihenfolge der besprochenen Modi orientiert sich dabei am Aufbau des Hauptmenüs der graphischen Schnittstelle. Der Ablauf der einzelnen Modi ist jeweils ähnlich, daher wird der erste Modus genauer erklärt. In den weiteren Modi wird auf die Unterschiede zum ersten und speziellen Erweiterungen eingegangen.

5.8.1 Standby-Modus

Der erste Modus, der nach der in Kapitel 5.1 beschriebenen Initialisierungsphase automatisch erreicht wird, ist *Standby*. Dies geschieht deshalb, weil nach dem

Programmstart der Systemzustand *state* bei der Instanziierung gleich auf *Standby* gesetzt wird. Aus diesem Modus kann der Benutzer in alle anderen Zustände wechseln. Der Algorithmus dafür ist in Form eines Flussdiagramms in Abbildung 57 dargestellt. Zu Beginn des Modulablaufs wird auch der Zustand für die graphische Benutzeroberfläche *uiState* mit der Funktion *setMode()* auf den Zustand *UIStandbyMode* gesetzt. Um die Benutzerschnittstelle auch graphisch anzuzeigen, müssen die entsprechenden Module als Threads gestartet werden. Dafür wird hier die Standard-C++-Bibliothek *thread* verwendet. Wie in Kapitel 3.3 definiert wurde, soll zur Kontrolle der Fahrzeugausrüstung das aktuelle Kamerabild angezeigt werden. Dafür wird das *UserInterface*- und das *CameraImageAcquisitor*-Modul mit den entsprechenden Funktionsparametern aktiviert. Die jeweiligen Threads des vorliegenden Modus werden als *uiShowThread* und *imageAcquisitionThread* bezeichnet. Damit das Ausgabebild des *CameraImageAcquisitor*-Moduls zum *UserInterface*-Modul übertragen werden kann, wird beiden Threads eine Bilddatenstruktur *inputImage* als Parameter übergeben. Abbildung 56 zeigt das Ausgabebild des Modus. Auf der linken Bildseite ist das Hauptmenü zu sehen, welches die Bedienelemente für die Benutzer-System-Interaktion zeigt. Rechts davon befindet sich das aktuelle Kamerabild, welches unbearbeitet angezeigt wird, weil im *Standby*-Modus noch keine Bildverarbeitung gestartet wurde.

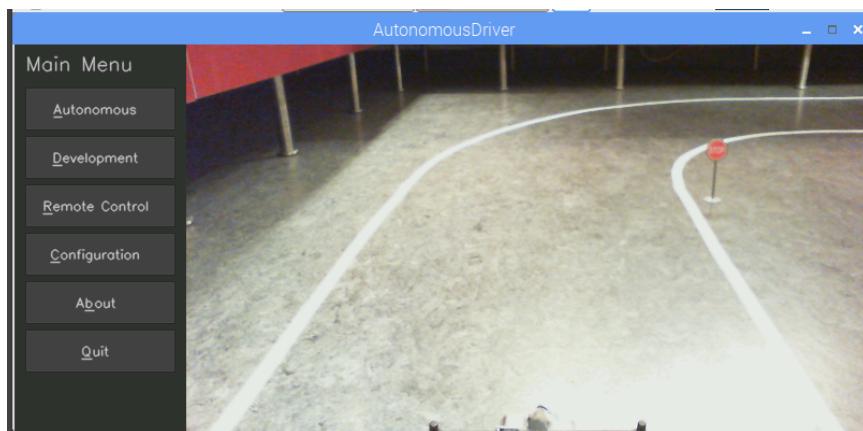


Abbildung 56: Benutzerschnittstelle des *Standby*-Modus

In einer Schleife überprüft das Programm, ob der Benutzer eine Taste oder eine entsprechende Schaltfläche gedrückt hat, die einen Zustandswechsel verursachen würde. Nach dieser Überprüfung pausiert der Algorithmus für 50 ms. Zum einen ist keine häufigere Überprüfung notwendig, da die Reaktionszeit des Benutzers deutlich länger ist. Zum anderen können damit Systemressourcen gespart werden. Diese Pause beeinträchtigt die Threads nicht, weil diese parallel zu dem hier beschriebenen Ablauf ausgeführt werden. Wie im Zustandsdiagramm in Abbildung 21 gezeigt, können aus diesem Modus die weiteren Modi *Autonomous Driving*, *Development*, *Remote Control*, *Configuration*, *About* und *Quit* erreicht werden. Vor dem Moduswechsel werden alle laufenden Module geschlossen. Um das möglich zu machen, wird für jedes Modul die Funktion *quit()* aufgerufen, welche wiederum die Modulinterne *running*-Variable auf *false* gesetzt. Dies beendet die Programmschleife des Moduls. Mit der Funktion *join()* wird abgewartet, bis die Threads tatsächlich beendet wurden. Je nach eingegebener Taste oder gewähltem Bedienfeld wird in den entsprechenden Modus gewechselt. Zum Schluss wird das *Standby*-Objekt

gelöscht, weil es nicht weiter benötigt wird. Soll später wieder in diesen Modus gewechselt werden, dann wird eine neue Instanz dafür erzeugt. Damit wird sichergestellt, dass immer nur ein Modus aktiv und verfügbar ist. Passiert in einem der vom Modus aufgerufenen Module ein Fehler, so werden alle Module des Modus beendet. Anschließend wechselt das System in den Fehlermodus, der einen weiteren Wechsel in den Beendigungsmodus vornimmt um das System sicher zu beenden.

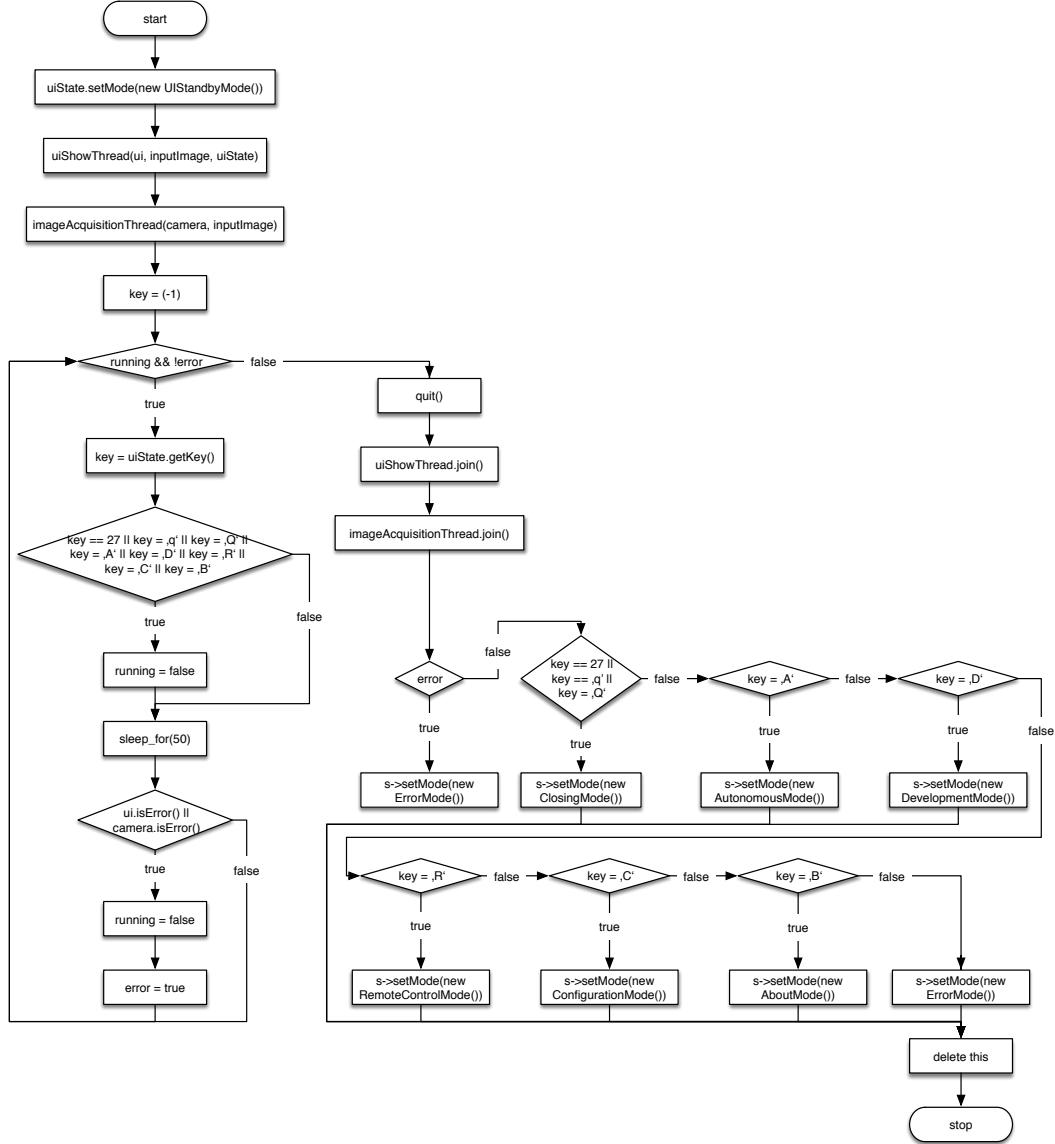


Abbildung 57: Flussdiagramm für den *Standby-Modus*

5.8.2 Autonomer Fahrmodus

Der Hauptmodus des hier implementierten Systems ist *Autonomous Driving*. Nach dem Start wird der Modus der Benutzerschnittstelle auf *UIAutonomousMode* gestellt. Diese Schnittstelle wird in Abbildung 58 dargestellt. Im linken Teil des Bildes werden dem Benutzer Bedienmöglichkeiten zur Systemsteuerung angezeigt. Über die Schaltfläche *Back*

wird der vorherige Modus, in diesem Fall der *Standby*-Modus, erreicht und der aktuelle autonome Fahrmodus beendet. Sollte es im Notfall nötigt sein, so kann das System über die Schaltfläche *Quit* sofort beendet werden. Diese Schaltfläche initiiert einen Wechsel des Systems in den Beendigungsmodus, der es sicher beendet. In den Textausgaben unter den Schaltflächen werden Informationen über den aktuellen Lenkwinkel und den Beschleunigungswert angezeigt. Darunter befinden sich Informationen über die erkannte Umgebung. In den ersten beiden Zeilen dieser Umgebungsinformationen werden die Start- und Endpunktkoordinaten der linken und der rechten Fahrbahnmarkierung ausgegeben. Anschließend ist die gemessene Distanz zum nächsten Hindernis in Zentimetern angeführt. Zum Schluss werden die Koordinaten des Mittelpunkts des erkannten Stopperschildes angegeben. Zusätzlich zu dieser textuellen Ausgabe wird das bearbeitete Kamerabild rechts im Fenster angezeigt. Dieses ist perspektivisch transformiert und enthält eine blaue Linienmarkierung für die erkannte linke sowie eine rote Linie für die rechte Fahrbahnmarkierung. Wird auch ein Verkehrsschild detektiert, dann wird es mit einem grünen Rechteck umrahmt. Dieses ist in der Ausgabe allerdings auch perspektivisch verzerrt und hat deshalb eine Trapezform.

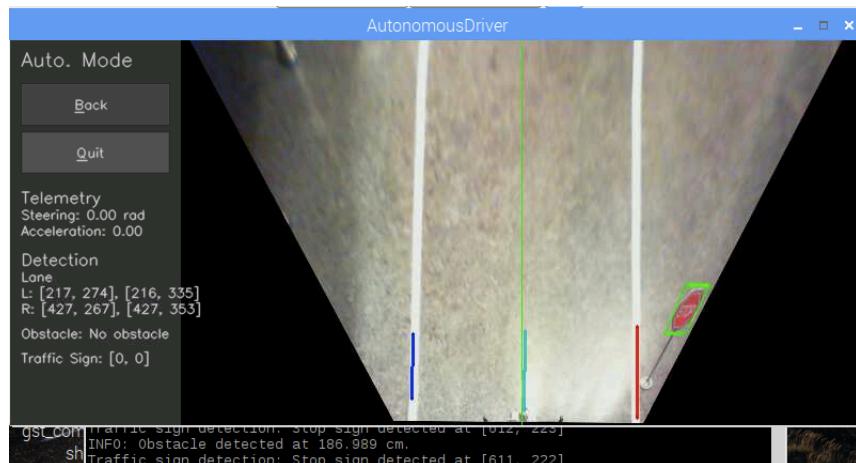


Abbildung 58: Benutzerschnittstelle für den autonomen Fahrmodus

Der Programmfluss des autonomen Fahrmodus ist in Abbildung 59 dargestellt. Für die Benutzerausgabe wird erneut ein Thread mit der Bezeichnung *uiShowThread* gestartet. Die Bilderfassung geschieht jedoch im *imageAcquisitionThread*. In diesem Modus werden den beiden Threads nicht dieselben Bilddaten übergeben, weil das Eingabebild vor der Ausgabe zunächst bearbeitet werden muss. Dem *UserInterface*-Modul wird daher eine *outputImage*-Variable und dem *CameraImageAcquisitor* eine *inputImage*-Variable übergeben. Um aus den Bilddaten bestimmte Objekte in der Umwelt zu erkennen, werden die beiden Module *LaneDetector* und *TrafficSignDetector* in den Threads *laneDetectionThread* sowie *trafficSignDetectionThread* gestartet. Beide Module erhalten das Eingabebild *inputImage* als Parameter. Dadurch, dass die Fahrbahnerkennung die Bildverarbeitung durchführt, bekommt sie zusätzlich das *outputImage* als Parameter, in das sie die Ergebnisse ablegt. Dem Benutzer wird dieses bearbeitete Bild auf der Benutzeroberfläche gezeigt. Die Verkehrsschilderkennung zeichnet eine rechteckige Markierung um die erkannten Stoppschilder und zeichnet das Ergebnis im Eingabebild ein. Dieses gelangt samt Markierungen zur Fahrbahnerkennung, wobei sich die gezeichneten Rechtecke auf die

Straßenmarkierungserkennung nicht störend auswirkt. Diese wird parallel zur Stoppschilderkennung im selben Eingabebild durchgeführt. Die Ergebnisse der Erkennungsverfahren werden schließlich in den jeweiligen Variablen *lane* bzw. *trafficSigns* gespeichert. Für die Hinderniserkennung wird auch das *ObstacleDetector*-Modul im Thread *obstacleDetectionThread* ausgeführt. Die Ergebnisse dieses Moduls werden in der Variable *obstacle* abgelegt.

In Kapitel 5.4 zur Fahrplanung wurde beschrieben, wie aus den erkannten Fahrbahn- und Verkehrsschildern sowie aus dem Abstand zum nächsten Hindernis eine Trajektorie kalkuliert wird. Deshalb werden alle diese Daten an das *PathPlanner*-Modul weitergeleitet, welches im *pathPlaningThread* ausgeführt wird. Die Fahrt selbst wird im *VehicleController*-Modul durch die Ausführung als *vehicleControllerThread* initiiert. Dieses benötigt Informationen über die Trajektorie und das Fahrzeug um die Fahrbefehle auszuführen.

Wie im letzten Modus wird in einer Programmschleife die Benutzereingabe überprüft und anschließend 50 ms gewartet. Wurde eine gültige Tastatur- oder Bedienfeldeingabe gemacht, dann wird die Schleife verlassen und auf die Beendigung der oben gestarteten Threads mit *join()* gewartet. Wurde das Feld *Back* oder die Taste *B* gedrückt, dann wird in den Systemzustand *Standby* gewechselt. Am Ende wird die Objekt-Instanz dieses Modus gelöscht. Ereignet sich in einem der aufgerufenen Module ein Fehler, dann werden alle Module des autonomen Fahrmodus beendet. Im Gegensatz zum *Standby*-Modus wechselt das System hier nicht in den Fehlermodus, sondern in den *Standby*-Modus, sodass fehlerhafte Einstellungen überprüft und korrigiert werden können.

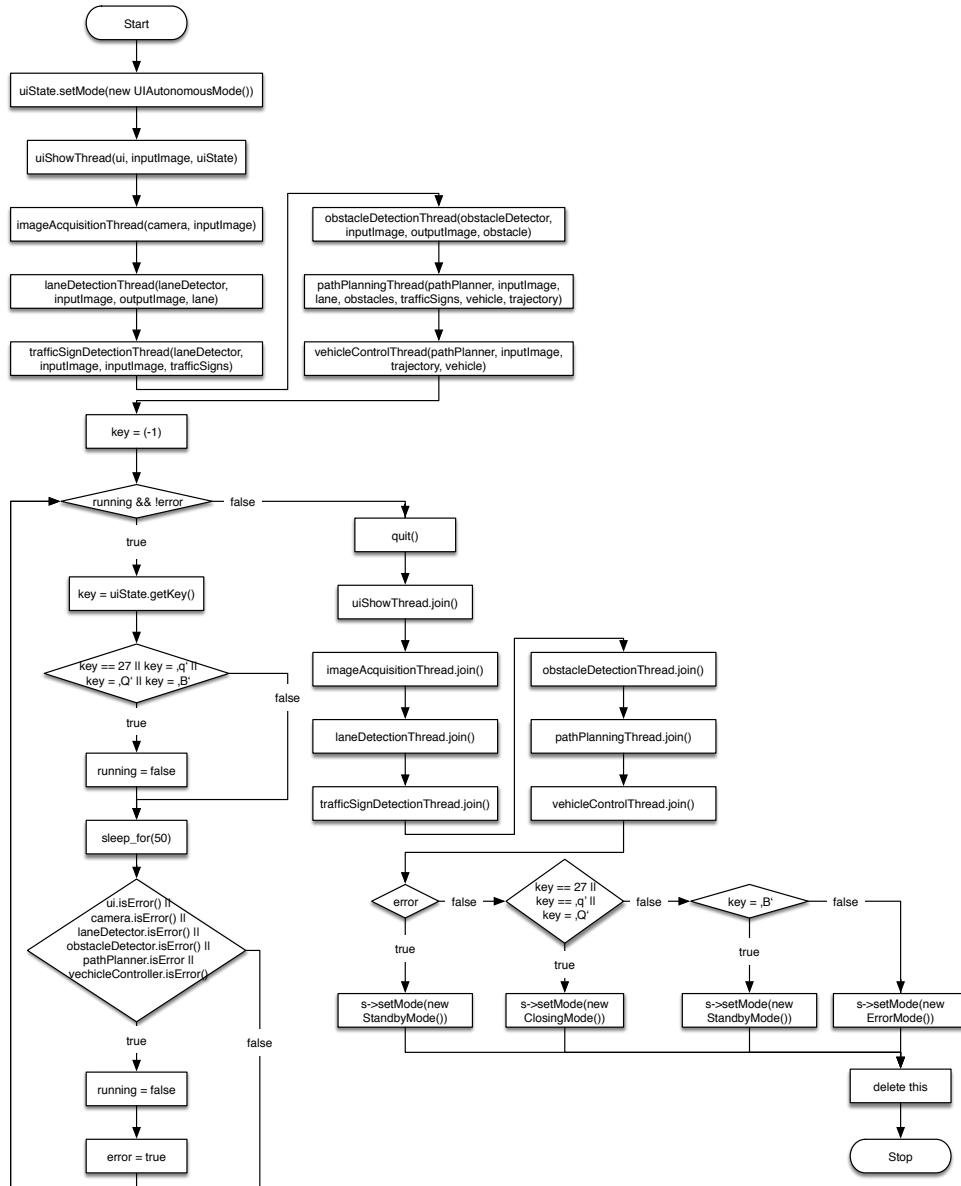


Abbildung 59: Flussdiagramm für den autonomen Fahrmodus

5.8.3 Entwicklungsmodus

Wie in den Abläufen der Modi der letzten beiden Kapitel zu erkennen ist, laufen diese ähnlich ab und unterscheiden sich hauptsächlich in den gestarteten Modulen bzw. Threads. Aus diesem Grund wird hier beim Ablauf des Entwicklungsmodus, welcher in Abbildung 60 dargestellt ist, nur auf die Unterschiede zum Modus davor eingegangen.

Dieser Modus stellt eine Kombination aus dem autonomen Fahrbetrieb und dem Fernsteuerungsbetrieb dar, welcher im nächsten Kapitel beschrieben wird. Für die Benutzeroaus- und -eingabe wird wieder ein *uiShowThread* gestartet. Im Entwicklungsmodus sollen Erkennungsverfahren evaluiert werden können, ohne dass sie Auswirkungen auf das physische Verhalten des Fahrzeugs haben. Für die Erkennungsverfahren ist erneut die Bilderfassung mit *imageAcquisitionThread* nötig. Die in

dieser Arbeit implementierten Detektionsprogramme werden dann in den Threads *laneDetectionThread*, *trafficSignDetectionThread* und *obstacleDetectionThread* ausgeführt und deren Ergebnisse über die graphische Benutzeroberfläche angezeigt. Statt dem im autonomen Fahrmodus gestarteten *PathPlanner*-Modul wird hier das *RemoteController*-Modul im *remoteControlThread* gestartet. Dieses gibt dem Benutzer die Möglichkeit, das Fahrzeug über die Tastatur oder über Bedienfelder am Bildschirm zu lenken. Die Ausführung der Steuerbefehle übernimmt erneut der *vehicleControlThread*. Hier ist anzumerken, dass dem letztgenannten Thread ein Trajektoriendatenobjekt ohne Daten gegeben wird, weshalb keine autonome Steuerung durchgeführt wird. Es werden nur die Werte für den Lenkwinkel und die Akzeleration aus den Fahrzeugdaten gelesen, die vom Fernsteuerungsmodul auf der Basis der Benutzereingaben dort abgelegt wurden. Diese werden in Motorsteuerbefehle umgewandelt. Daher kann der Benutzer während der Fernsteuerung des Fahrzeugs die Ergebnisse der Detektionsverfahren sehen und evaluieren.

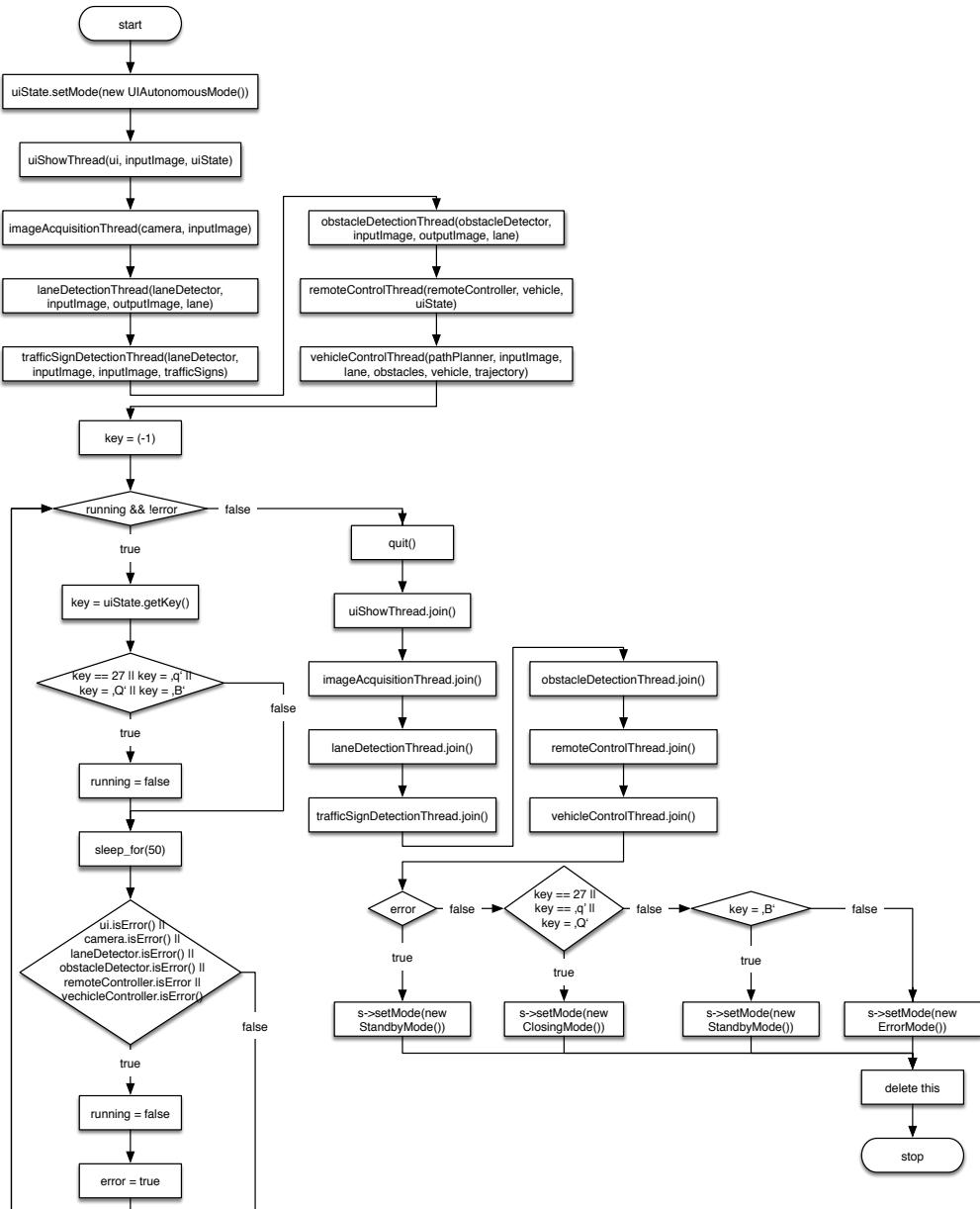


Abbildung 60: Flussdiagramm für den Entwicklungsmodus

Die graphische Benutzerschnittstelle für den Entwicklungsmodus ähnelt ebenfalls der des autonomen Fahrmodus, bietet aber darüber hinaus Bedienfelder zur Einstellung des Lenkwinkels und der Beschleunigung. Abbildung 61 zeigt diese Schnittstelle. Die Eingabe der Steuerung kann jedoch genauso über die Tastatur erfolgen. Wie in Kapitel 5.6 beschrieben, erhöht die Taste *W* die Akzeleration, *S* reduziert sie wieder. Mit den Tasten *A* und *D* kann nach links bzw. nach rechts gelenkt werden. Die Leertaste bringt das Fahrzeug zum Stillstand. Aus diesem Modus kann zurück zu *Standby* gewechselt oder das System beendet werden. Passiert jedoch in einem der aufgerufenen Module ein Fehler, so werden, wie im autonomen Fahrmodus alle Module des vorliegenden Modus beendet. Anschließend wechselt das System auch hier in den *Standby*-Modus, um die Überprüfung und Korrektur fehlerhafter Einstellungen zu ermöglichen.

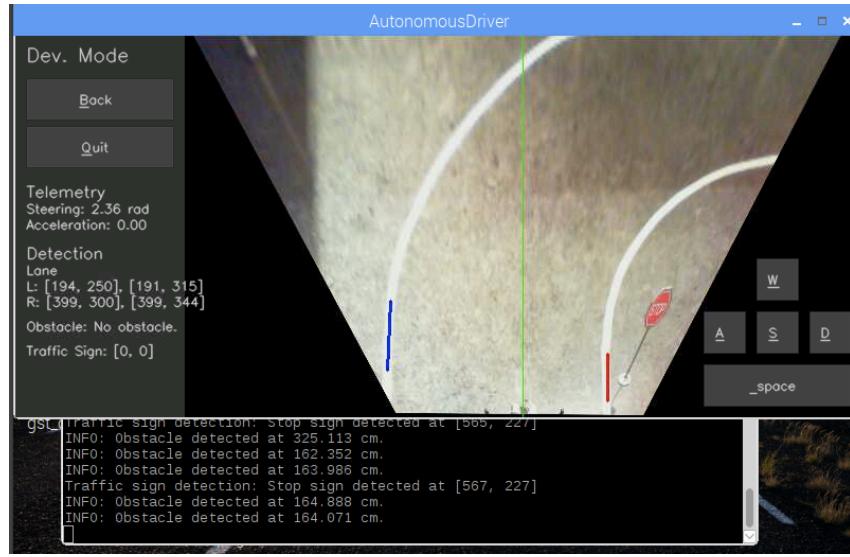


Abbildung 61: Benutzerschnittstelle für den Entwicklungsmodus

5.8.4 Fernsteuerungsmodus

Der Fernsteuerungsmodus erlaubt dem Benutzer die Kontrolle der Fahrt zu übernehmen. In diesem Modus sind keine Erkennungs- oder Planungsverfahren notwendig. Damit der Benutzer sieht, wohin das Fahrzeug fährt, ist die unverarbeitete Kameraaufnahme und die Benutzeroberfläche notwendig. Ähnlich wie in den Kapiteln zuvor werden die Threads *uiShowThread*, *imageacquisitionThread*, *remoteControlThread* und *vehicleControlThread* ausgeführt. Die Abbildung 62 zeigt den Programmfluss des Fernsteuerungsmodus.

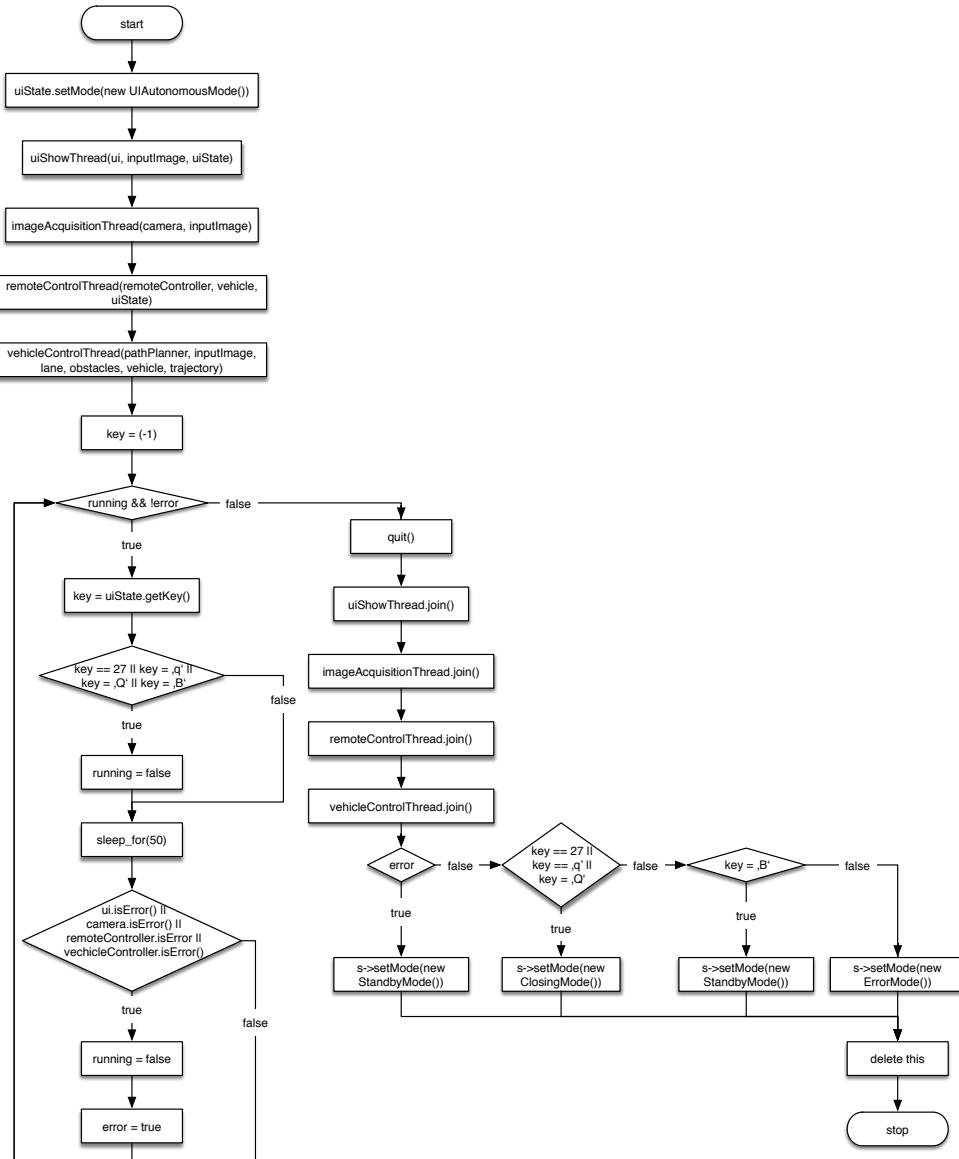


Abbildung 62: Flussdiagramm für den Fernsteuerungsmodus

In Abbildung 63 ist die graphische Schnittstelle für den Benutzer samt den Bedienmöglichkeiten abgebildet. Wie im vorherigen Kapitel beschrieben, sind auch hier die Steuerflächen sowie -tasten *W*, *A*, *S*, *D* und die Leertaste implementiert. Über die Bedienfläche *Back* kann wieder in den *Standby*-Modus gewechselt und über *Quit* das System heruntergefahren werden. Wird in einem der aufgerufenen Module eine Fehlermeldung ausgegeben, so werden auch hier alle Module des Modus beendet. Anschließend wird ein Wechselt des Systems in den *Standby*-Modus durchgeführt, sodass die entsprechenden Einstellungen auf ihre Korrektheit überprüft und gegebenenfalls korrigiert werden können.

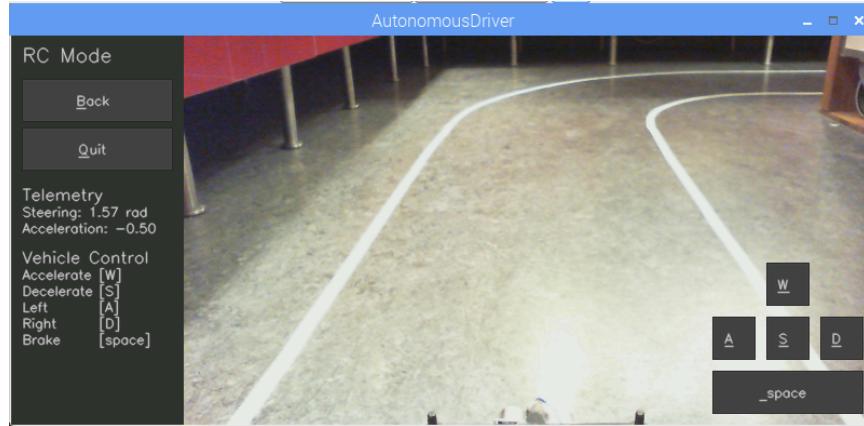


Abbildung 63: Benutzerschnittstelle für den Fernsteuerungsmodus

5.8.5 Konfigurationsmodus

Der Modus zur Systemkonfiguration bietet die Möglichkeit, Einstellungen des Systems zu verändern, zurückzusetzen oder zu speichern. Im Rahmen dieser Arbeit wurden die beiden Möglichkeiten zur intrinsischen und extrinsischen Kamerakalibration implementiert. Diese benötigen zwei zusätzliche Modi, die im nächsten Abschnitt erläutert werden.

Im Konfigurationsmodus wird ausschließlich ein Menü mit der Auflistung der Einstellungsmöglichkeit, sowie das aktuelle Kamerabild ausgegeben. Letzteres wird angezeigt, um die Kamerapositionierung überprüfen und später eine korrekte Kalibration durchführen zu können. Der Programmfluss in Abbildung 64 dargestellt. Die Benutzeraus- und -eingabe wird erneut über das Starten des *uiShowThreads* möglich. Für das Kamerabild wird der *imageAcquisitorThread* benötigt.

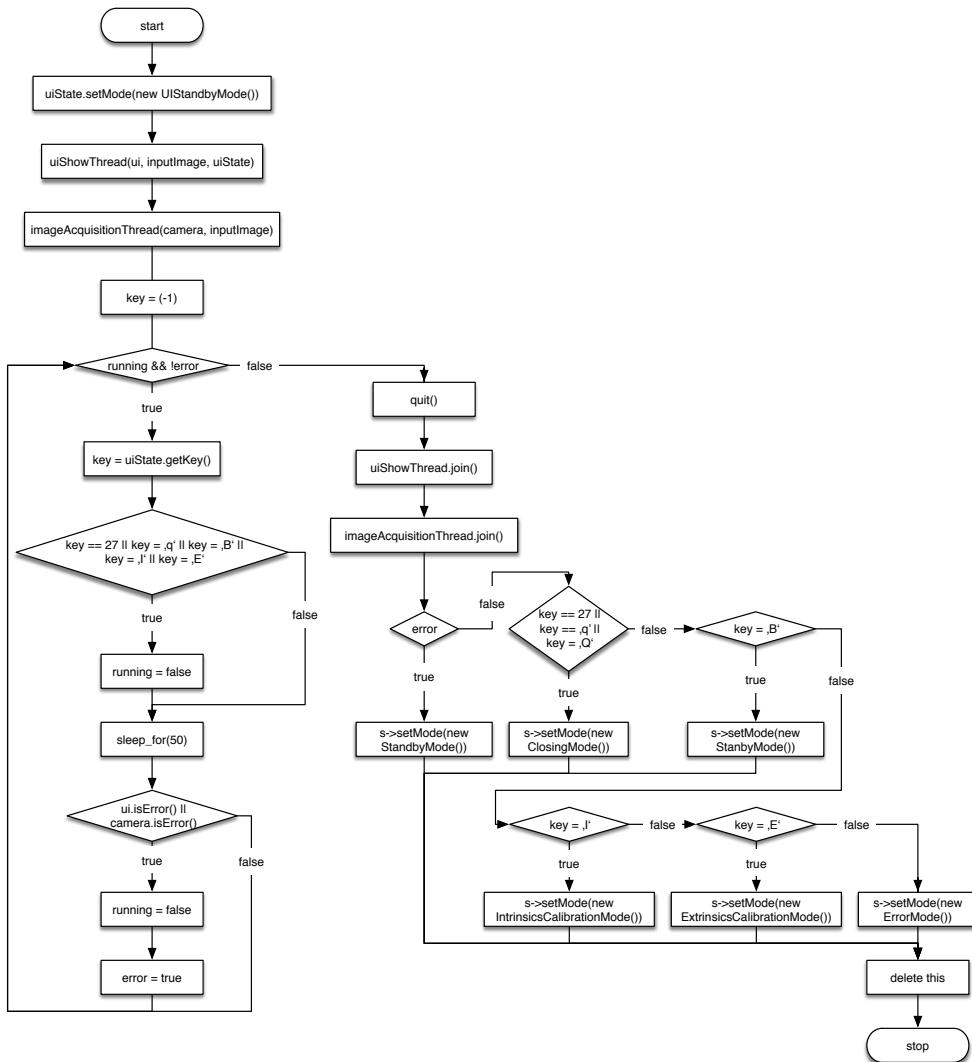


Abbildung 64: Flussdiagramm für den Konfigurationsmodus

Die Benutzerschnittstelle für den Konfigurationsmodus ist in Abbildung 65 dargestellt. Im Menü links sind die Entstellungsmöglichkeiten und die entsprechenden Bedienflächen platziert. *Back* oder die Taste *B* führt zum *Standby*-Modus, *Quit* oder *Q* fährt das System herunter, *Intrinsics* oder *I* führt zum Kalibrationsmodus für die intrinsischen Kameraparameter und *Extrinsics* oder *E* führt zum Modus für die extrinsischen Kameraparameter.

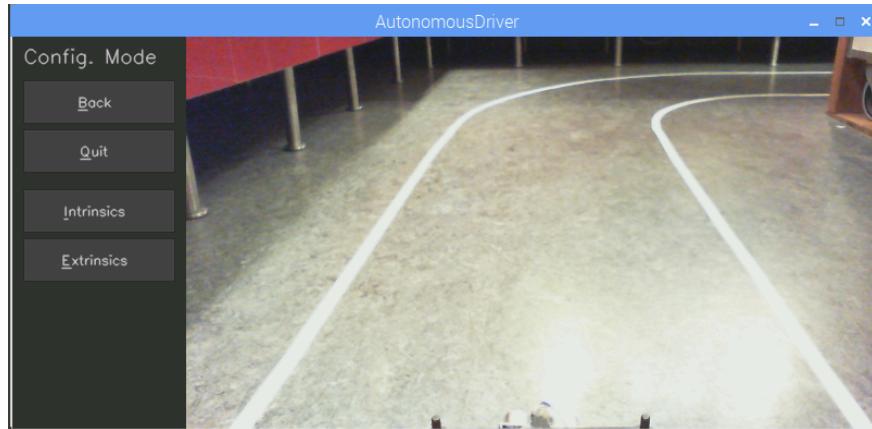


Abbildung 65: Benutzerschnittstelle für den Konfigurationsmodus

5.8.6 Kalibrationsmodi

Die beiden Modi für die intrinsische und die extrinsische Kamerakalibration sind ausschließlich über den im vorigen Kapitel besprochenen Konfigurationsmodus erreichbar. Im intrinsischen Kamerakalibrationsmodus wird durch das entsprechende Modul die Kameramatrix und ein Verzerrungskoeffizientenvektor berechnet, wie in Kapitel 5.2.1 beschrieben wird. Die extrinsische Kalibration ist in Kapitel 5.2.2 erklärt und erstellt eine Homographiematrix, mit der das Kamerabild perspektivisch transformiert werden kann. Der Ablauf der beiden Modi ist beinahe gleich und wird in Abbildung 66 dargestellt. In beiden muss ein Kamerabild aufgenommen und verarbeitet werden. Dafür wird ein uiShowThread und ein imageAcquisitorThread gestartet. Beide Kalibrierungsmodule sind in dieser Arbeit als Funktionen der *CameraImageAcquisitor*-Klasse implementiert. Sie werden jeweils mit den Namen *runIntrinsicCalibration()* bzw. *runExtrinsicCalibration()* im *calibrationThread* gestartet. Passiert in einem der durch die beiden Modi aufgerufenen Module ein Fehler, so werden alle Module des jeweiligen Modus beendet. Anschließend wechselt das System in den Konfigurationsmodus, sodass der Benutzer fehlerhafte Einstellungen überprüfen und korrigieren kann. Anschließend kann der gewünschte Kalibrationsmodus wieder ausgeführt werden.

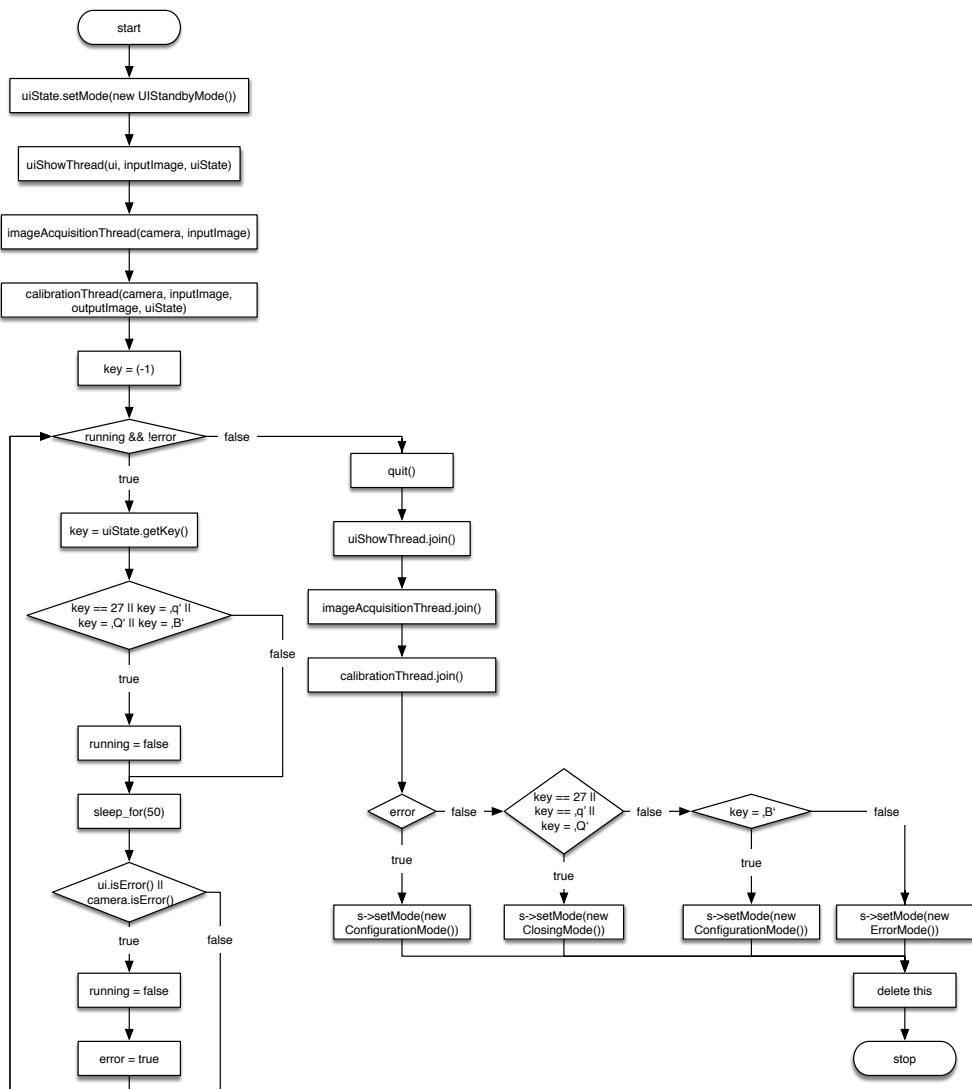


Abbildung 66: Flussdiagramm zum Kalibrierungsmodus

Über die Bedienfelder *Back* und *Quit* sowie den Tasten *B* bzw. *Q* können aus beiden Kalibrierungsmodi entweder zum *Standby*-Modus gewechselt oder das System beendet werden. In beiden Zuständen können die Daten der durchgeführten Kalibrationen abgespeichert oder zurückgesetzt werden. Bei der Speicherung werden die berechneten Daten in eine XML-Datei abgelegt. Das Zurücksetzen der Daten verwirft die kalibrierten Werte, wenn das Ergebnis nicht zufriedenstellend ist. Abbildung 67 zeigt die Benutzerschnittstelle des intrinsischen Kalibrierungsmodus. Neben den Bedienfeldern wird auch das Eingabebild angezeigt. Dieses wird allerdings nach einer erfolgreichen Kalibration entzerrt dargestellt. So kann der Benutzer das Ergebnis begutachten und eine gewünschte Auswahl treffen und gegebenenfalls eine neue Kalibration initiieren.

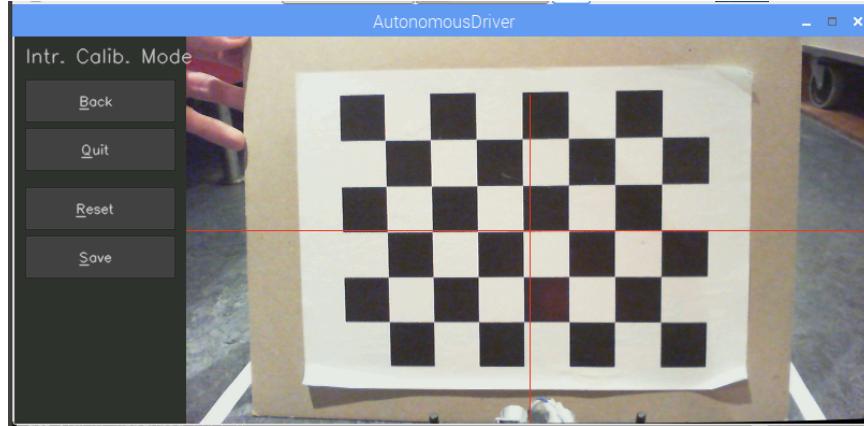


Abbildung 67: Benutzerschnittstelle für den intrinsischen Kamerakalibrierungsmodus

Der extrinsische Kalibrierungsmodus zeigt, im Vergleich zum intrinsischen Modus, im Ausgabebild, noch bevor eine Kalibration durchgeführt wurde, die horizontale und vertikale Bildachse. Diese Anzeige wird in Abbildung 68 gezeigt. Sie soll dem Benutzer helfen, nochmals die Ausrichtung der Kamera zu überprüfen. Nach der erfolgreichen Kalibration wird das perspektivisch transformierte Bild gezeigt. Auch hier werden die Achsen eingezeichnet. Damit kann die Verzerrung auf ihre Korrektheit überprüft werden. Erscheint das Kalibrierungsmuster im transformierten Bild mittig und an der Ausgabebildunterseite, dann wurde die Homographiematrix richtig berechnet. Damit ist die Kalibration erfolgreich und wird abgeschlossen.

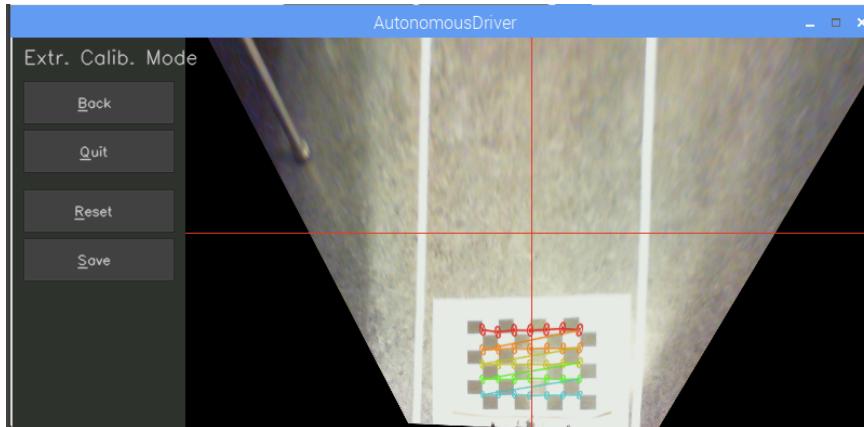


Abbildung 68: Benutzerschnittstelle für den extrinsischen Kamerakalibrierungsmodus

5.8.7 Informationsmodus

Der Informationsmodus dient ausschließlich der Projektentwicklung. Er zeigt den aktuellen Arbeitstitel, das Erstellungsdatum und eine Versionsnummer des Programms an. An dieser Stelle können zukünftig auch andere projektrelevante Hinweise gegeben werden. Dieser Modus nützt nur die Benutzerober- und -eingabe. Daher ist das Ausführen eines `uiShowThread()` ausreichend, wie im Ablauf in Abbildung 69 dargestellt ist.

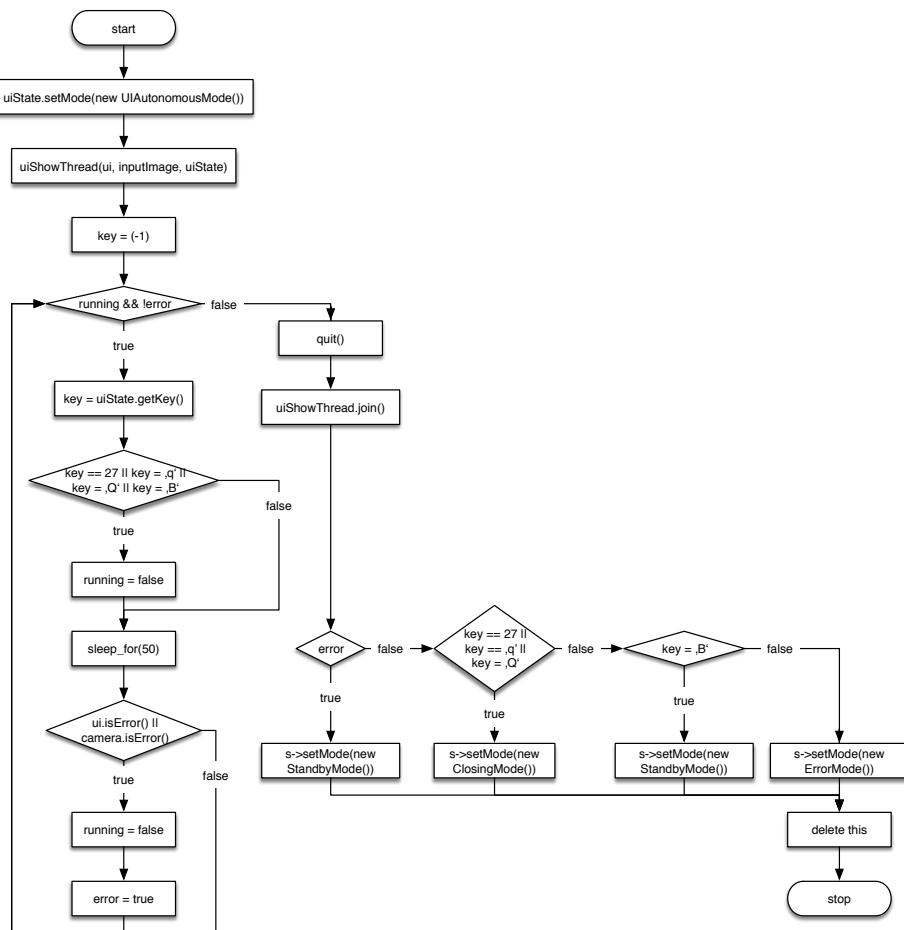


Abbildung 69: Flussdiagramm zum Informationsmodus

Das Bedienmenü wird auch in diesem Modus links ausgegeben. Der Ausgabetext wird anstelle eines Kamerabilds mittig auf der graphischen Oberfläche platziert. Aus diesem Zustand kann wieder in den *Standby*-Modus gewechselt oder das System beendet werden. Abbildung 70 stellt die beschriebene Benutzerschnittstelle graphisch dar.



Abbildung 70: Benutzerschnittstelle für den Informationsmodus

5.8.8 Fehlermodus

Die Aufgabe des Fehlermodus ist es, nicht kompensierbare Fehler der anderen Modi abzufangen. Diese Fehlerfälle sind in den jeweiligen Modulen genauer beschrieben. Tritt ein Fehler in einem Modul auf, wechselt der aufrufende Modus in den Fehlermodus. Im Vergleich zu den bereits besprochenen Modi verfügt der *Error-Mode* über keine graphische Benutzeroberfläche. Der Grund dafür ist, dass potentielle Fehler, die im Benutzerschnittstellenmodul auftreten könnten, damit vermieden werden sollen. Die Implementierung dieser Arbeit führt im Fehlermodus lediglich einen Wechsel in den Beendigungsmodus durch. An dieser Stelle können in zukünftigen Entwicklungsschritten weitere Maßnahmen für den sicheren Systemabschluss getroffen werden. Der genaue Ablauf dieses Modus ist in Abbildung 71 dargestellt.

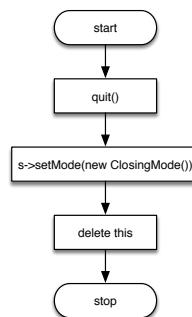


Abbildung 71: Flussdiagramm für den Fehlermodus

5.8.9 Beendigungsmodus

Der Beendigungsmodus soll das System sicher beenden und muss von jedem anderen Zustand erreicht werden können. In Abbildung 72 ist der Ablauf des Modus zu sehen. Dazu werden alle Motoren des Prototyps zum Stillstand gebracht und der Motortreiber zurückgesetzt. Dadurch hält das Fahrzeug an und erreicht damit den in Kapitel 3.3 besprochenen sicheren Zustand. Zum Schluss wird auch die Systemabschaltfunktion *quit()* aufgerufen, die den aktuellen Modus, dadurch alle aktiven Threads und das System beendet. Eine Benutzeroberfläche für diesen Modus ist ebenfalls nicht sinnvoll, weil sonst erneut Module dafür ausgeführt werden müssten, die das System in einen potentiell unsicheren Zustand bringen könnten.

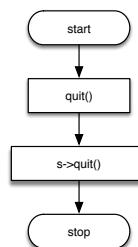


Abbildung 72: Flussdiagramm für den Beendigungsmodus

6. ERGEBNISSE

Ziel dieser Arbeit war es, zum einen eine Entwicklungsumgebung für das autonome Fahren vorzuschlagen, zum anderen diesen Vorschlag in Form eines Prototyps umzusetzen. Mit der in Kapitel 4 und 5 vorgestellten Hardware- und Softwarerealisierung werden im vorliegenden Kapitel eine Reihe von Tests durchgeführt. Diese und deren Ergebnisse werden hier beschrieben.

Zunächst kann festgehalten werden, dass das eingangs gesetzte Ziel erreicht wurde und eine modulare quelloffene Entwicklungsumgebung erstellt werden konnte. Die in dieser Arbeit entwickelte Software des ebenfalls hier konstruierten Prototyps wird im Folgenden getestet. Sie ist als beispielhafte Umsetzung einer Applikation für das autonome Fahren zu verstehen und kann in späteren Entwicklungsschritten beliebig erweitert oder modifiziert werden. Dabei können die hier verwendeten Verfahren weiter verbessert oder durch effizientere Lösungen ersetzt werden, was in Kapitel 6.5 genauer ausgeführt wird. Durch die im Folgenden beschriebenen Tests konnte jedoch nachgewiesen werden, dass mit der hier präsentierten Entwicklungsumgebung eine Applikation für das autonome Fahren erfolgreich entwickelt und evaluiert werden konnte.

6.1 Teststrecke

Der hier realisierte Prototyp soll im autonomen Fahrmodus entlang einer gekennzeichneten Fahrbahn fahren. Diese besteht aus einem dunklen Untergrund und einer hellen Begrenzungsmarkierung, wie bereits in Kapitel 2.2.1 definiert und in Abbildung 6 dargestellt wurde. Für das hier eingesetzte Erkennungsverfahren ist es von Vorteil, wenn der Kontrast zwischen dem Fahrbahnuntergrund und der Markierung möglichst groß ist. Für diese Arbeit wurden verschiedene Teststreckenverläufe mit weißem handelsüblichen Isolierband, welches eine Breite von 2 cm hat, auf dunkelgrauen Fußböden in Innenräumen aufgeklebt. Dieses Breite ist zwar nicht zwingend erforderlich, hebt sich jedoch deutlich vom Untergrund ab.

Die Fahrspur, also der Bereich zwischen zwei Markierungen muss auf geraden Abschnitten zumindest die Breite des Fahrzeugs selbst, also 20 cm, aufweisen. Aufgrund des hier eingesetzten Verfahrens würde der Prototyp zwar auch auf einer schmäleren Fahrbahn fahren, weil keine Überprüfung der Fahrbahnbreite implementiert ist, doch gehört dieser Testfall nicht zum Ziel dieser Arbeit. Wird bei der Erstellung der Teststrecke ein zusätzlicher Sicherheitsabstand von 2 cm pro Fahrzeugseite beachtet, dann erlaubt das dem System, kleine Kurskorrekturen durchzuführen und dennoch innerhalb des Fahrstreifens zu bleiben. Kurvenmarkierungen müssen etwas breiter angelegt werden als die für das Geradeausfahren. Der Grund dafür ist, dass das Fahrzeug beim Einlenken eine größere Fläche befährt. Je steiler, das heißt je kleiner, der Kurvenradius ist, desto größer wird diese Fläche. Der Kurvenverlauf selbst kann beliebig sein, solange der Prototyp zwischen den Markierungen Platz hat. Die Wettbewerbsstrecke des *Carolo-Cups* hat laut dessen Regelwerks eine konstante Fahrbahnbreite von 82 cm, die in zwei Fahrstreifen unterteilt ist (vgl. [TUB18]). Eine solche Spur ist daher 40 cm breit. Im *Audi Autonomous Diving Cup*

wird eine ähnliche Spurbreite von 44 cm angegeben (vgl. [A18a]). Deshalb wurden auch für die folgenden Testläufe Straßenabschnitte in der Breite von 40 cm angelegt.

6.2 Autonomes Fahren

Lässt man das Modelfahrzeug im autonomen Fahrmodus die im vorherigen Kapitel beschriebene markierte Modellstraße befahren, gelingt dies, wobei in Versuchen auch einige Grenzen der Implementierung ersichtlich wurden. Erkennt das Fahrzeug Fahrbahnmarkierungen, fährt es, wie vorgesehen, selbstständig los. Der Benutzer kann das Verhalten über die Benutzeroberfläche, welche über VNC an einen Hostcomputer übertragen wird, mitverfolgen. Gerade Straßenabschnitte werden, wie gewünscht, mit konstanter Geschwindigkeit und innerhalb der Markierungen befahren. Kurskorrekturen werden durch kleine Lenkwinkelanpassungen automatisch umgesetzt. Wird das Fahrzeug vor der Fahrt nicht parallel zur geraden Fahrbahn platziert, so fährt dieses los und korrigiert die Position durch entsprechend größere Lenkwinkel selbstständig. Diese Korrektur wird solange durchgeführt, bis die Fahrzeuglängsachse parallel über die Fahrspurmitte gebracht wird.

6.2.1 Geradeausfahrt

Das hier implementierte Verfahren zur Linienerkennung ist gegen Unterbrechungen in der Markierung robust. Wie in Kapitel 5.3.2 erklärt, werden Linien, die links und rechts von der mittig gelegenen vertikalen Bildachse beginnen, gesucht. Die probabilistische *Hough*-Funktion kompensiert kleinere Unterbrechungen oder Störungen in der Markierung und erkennt die Linie daher zuverlässig. Wie groß eine Fehl- oder Störstelle einer Markierung sein darf, hängt von der Parametrisierung der *Hough*-Transformation ab. Die Parameterwerte in dieser Arbeit wurden anhand von Versuchen ermittelt. Fehlt eine der beiden Begrenzungslinien gänzlich oder wird sie nicht erkannt, so steuert das Fahrzeug auf die andere erkannte Linie zu und fährt entlang dieser weiter. Durch die *Kalman*-Filterung der detektierten Linien und der Trajektorie, verursachen fehlende Messungen keine ruckartigen Kurskorrekturen, weil stets die Abschätzung der Markierungsposition, nicht jedoch die direkten Messdaten, zur Weiterverarbeitung herangezogen werden. In Kapitel 5.3.2 wird definiert, dass die Fahrt nicht gestartet oder fortgeführt werden soll, wenn sich die erkannten Begrenzungslinien normal zur Fahrzeuglängsachse befinden. Wie vorgesehen, bleibt das Fahrzeug auch dann stehen, wenn keine Markierungen erkannt werden.

Durch die Position der Fahrzeuglängsachse im Vergleich zur Trajektorie wird der Lenkwinkel bestimmt. Befindet sich diese Achse zur Gänze links bzw. rechts von der Trajektorie, dann wird der maximale Lenkwinkel von $\pi/4$ zur Trajektorie hin eingeschlagen. Wie in Kapitel 5.5 erklärt wurde, kann der Wendekreis des Fahrzeugs durch die Reduktion der Fahrgeschwindigkeit verkleinert werden. Aus diesem Grund wird die Geschwindigkeitsreduktion für volle Lenkeinschläge auch in dieser Arbeit durchgeführt. Während der Testfahrt wirkt sich dieses Verhalten positiv auf eine schnelle Kurskorrektur aus. Beginnt die Trajektorie, anders als oben beschrieben, auf der einen Bildseite und endet

sie auf der anderen, dann bedeutet das für das System, dass entsprechend mit der Trajektorienrichtung gelenkt werden muss. Dazu wird die Geschwindigkeit nicht reduziert. Die Kantenerkennung und Trajektorienberechnung der Geradeausfahrt sind in Abbildung 73 zu sehen.

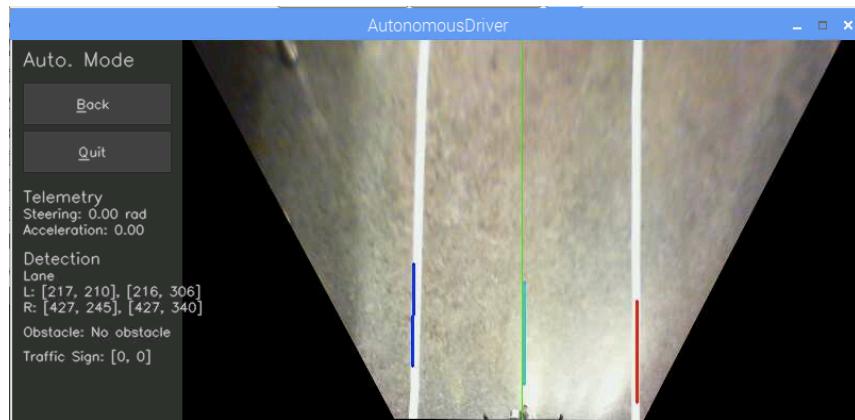


Abbildung 73: Geradeausfahrt mit Trajektorie

6.2.2 Kurvenfahrt

Über die Benutzerschnittstelle ist zu erkennen, dass Linien auch bei kurvenförmigen Markierungen erkannt werden. Je größer der Kurvenradius ist, desto gestreckter sind die Markierungen und desto länger ist auch die im Bild erkannte Linie. Bei genauerer Analyse des Ergebnisses der Linienerkennung ist zu sehen, dass entlang einer Kurve mehrere Geraden erkannt werden. Durch das hier implementierte Liniensortierungsverfahren, werden davon stets diejenigen, welche am nächsten zur unteren Bildrandmitte sind, gewählt.

Dabei werden nur weniger stark gekrümmte Fahrbahnmarkierungsabschnitte als Linien interpretiert. Fährt das Modellauto weiter, dann verschiebt sich die gewählte Linie aufgrund der *Kalman*-Filterung entlang des Kurvenscheitels. Da die rechte und die linke Markierung in einer Kurve nicht dieselbe Krümmung aufweisen, können die dafür erkannten Linien unterschiedlich weit von der Bildunterseite entfernt sein. Auch sind die Linien in solchen Fällen nicht parallel. Am Kurvenausgang wird der Kurvenradius wieder größer. Damit werden eher Linien im oberen Bildbereich gewählt. Durch die unterschiedliche Krümmung der rechten und der linken Straßenmarkierung schließt die erkannte Linie der Kurveninnenseite einen größeren Winkel mit der Fahrzeulgängsachse ein als die der äußeren. Die in so einem Fall berechnete Trajektorie, weiß eine Richtung auf, die dem Mittelwert der Winkel der beiden Markierungslinien entspricht. Bei Tests dieser Einstellung, erweist sich der so errechnete Lenkeinschlag als nicht ausreichend. Aus diesem Grund wurde in der Implementierung dieser ein fixer Korrekturwert von $\pi/8$ zu dem Winkel der Trajektorie addiert. Ein genaueres Kurvenerkennungsverfahren könnte im nächsten Entwicklungsschritt noch präzisere Werte liefern. Das Ergebnis einer korrekten Bilderkennung wird in Abbildung 74 gezeigt.



Abbildung 74: Korrekte Kurvenerkennung

Bei einigen der Kurvenfahrversuche wurde festgestellt, dass eine der beiden Markierungen außerhalb des Erfassungsbereichs der Kamera liegen kann. Dies kann sowohl die Detektion der Markierung entlang der Kurveninnenseite, als auch die entlang der Außenseite betreffen. Wenn das Fahrzeug daher nicht rechtzeitig oder weit genug einlenkt, dann bewegt es sich in Richtung Kurvenaußenrand. In engeren Kurven kommt es deshalb auch vor, dass das Fahrzeug über die Kurve hinausfährt. Dabei kann sich das Fahrzeug so positionieren, dass die Kamera nur mehr Bilder des äußeren Fahrbahnrandes aufnehmen kann. Lenkt das autonome Modelfahrzeug während der Kurskorrektur für die Kurvenfahrt hingegen zu weit nach innen, dann kann die Kamera nur die Kurveninnenseite aufzeichnen. Je nach Kurvenradius und Krümmung der Fahrbahnmarkierung kommt es daher auch vor, dass an manchen Stellen überhaupt keine Linie aufgezeichnet und damit erkannt werden kann. Solange eine der beiden Markierungen wahrgenommen wird, führt der Prototyp die Fahrt jedoch fort und lenkt entlang der erkannten Markierung. In den Tests ist ersichtlich, dass dadurch die Kurve dennoch korrekt abgefahren werden kann. Allerdings schneidet das Fahrzeug dabei die Kurve. Sobald das Fahrzeug den Kurvenausgang erreicht, werden wieder beide Markierungen detektiert und der Kurs korrigiert. In Abbildung 75 ist das oben beschriebene Phänomen bei der Erkennung einer Kurvenaußenseite abgebildet.

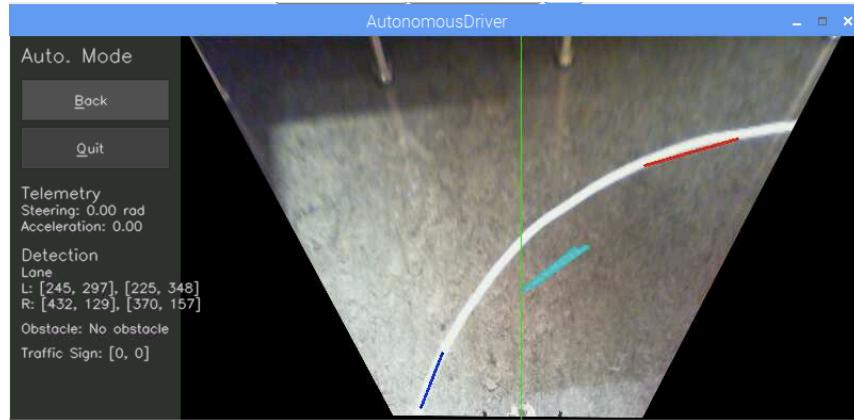


Abbildung 75: Erkennung einer Kurvenaußenseite

6.2.3 Stoppschilderkennung

An beliebigen Stellen der Teststrecke wurde während der Versuche dieser Arbeit ein Stoppschild aufgestellt. Die Größe des hier verwendeten Stoppschilds beträgt 30x30 mm und ist auf einem 55 mm hohen Standfuß befestigt. Im Aufnahmebild hängt die Größe des Schildes mit der Distanz zur Kamera und damit auch zum autonomen Fahrzeug ab. Das hier implementierte Verfahren hält das Fahrzeug an, sobald das erkannte Stoppschild eine Höhe oder Breite von 30 px im Aufnahmebild aufweist. Das entspricht, wie in Kapitel 5.3.3 gezeigt wurde, einer Distanz von 20 cm zwischen dem Schild und der Stoßstange des Prototyps. In den Versuchsläufen wird das Schild erkannt und das Fahrzeug solange angehalten, bis das Schild entfernt wird. Auf die genaue Position des Stoppschilds im Bild wird hier nicht geachtet, weil es im Straßenverkehr bzw. auf der Teststrecke an unterschiedlichen Stellen platziert sein könnte. Die Erkennung mit dem hier verwendeten Verfahren ist allerdings nicht immer korrekt. Es werden manchmal Stoppschilder in Bildern erkannt, obwohl entlang der Fahrbahn keine platziert sind. Das liegt am trainierten Klassifizierter und an der Parametrisierung der entsprechenden *OpenCV*-Funktion. In dieser Arbeit wurde kein Versuch unternommen, das Verfahren neu zu trainieren oder ein treffsichereres Verfahren zu wählen, weil das über die inhaltliche Zielsetzung und den zeitlichen Rahmen dieser Arbeit hinausgehen würde. Für eine Weiterentwicklung wird dies jedoch empfohlen. Die Erkennung eines Stoppschildes wird in Abbildung 58 dargestellt.

6.2.4 Kollisionsvermeidung

Hindernisse auf der Fahrbahn erkennt das hier entwickelte autonome Modellfahrzeug mit einem Ultraschallsensor, welcher an der Fahrzeugfront montiert ist. Wie in Kapitel 5.3.4 definiert, bleibt das Fahrzeug in allen Testläufen stehen, sobald die Distanz zwischen einem Hindernis und dem Fahrzeug kleiner oder gleich 25 cm ist. Das gelingt allerdings nur dann optimal, wenn sich das Hindernis geradeaus vor dem Fahrzeug befindet. Fährt der Prototyp eine Kurve, so ergibt sich das Problem, dass der Detektionsbereich des Sensors deutlich schmäler als die möglichen Lenkwinkel des Modellautos ist. Befindet sich also ein Hindernis in einer Kurve vor dem Fahrzeug, so kommt es vor, dass dieses zu spät erkannt wird. Ist der Abstand zum Hindernis bei dessen verspäteten Erkennung dann zu gering, kann der Prototyp nicht mehr rechtzeitig angehalten werden. Eine Kollision kann hier nicht vermieden werden. Als Sicherheitsvorkehrung zur Minimierung von Schäden wurde die Höchstgeschwindigkeit des autonomen Fahrzeugs begrenzt.

Eine mögliche Lösung für die Weiterentwicklung des Systems ist es, den Erfassungsbereich der Hinderniserkennung mithilfe von zwei oder vier weiteren Sensoren zu vergrößern. Diese müssten jeweils rechts und links des mittleren geradeausgerichteten Sensors mit einem unterschiedlichen Neigungswinkel im Vergleich zu diesem montiert werden. Werden Sensoren auch auf der Rückseite des Fahrzeugs montiert, dann könnten weitere Applikationen des autonomen Fahrens damit umgesetzt werden. Beispiele dafür wären das automatische Einparken oder Überholen. Es sei darauf hingewiesen, dass der *Raspberry Pi* nur über eine eingeschränkte Anzahl an GPIO-Pins verfügt, weshalb nur eine begrenzte Anzahl von Sensoren verbaut werden kann. Für diese Weiterentwicklungen wäre es besser, hier einen zusätzlichen Mikrocontroller mit mehr Pins einzusetzen, der die

Ultraschallmessungen durchführt und die gemessenen Distanzwerte regelmäßig an den steuernden Embedded-Computer schickt.

6.3 Systemleistung

Während der Versuche zum autonomen Fahren, führt das hier entwickelte System die Erfassung von Bildern, die Erkennung der Fahrbahn und der auftauchenden Stopperschilder sowie die Messung von Abständen zu Hindernissen vor dem Fahrzeug durch. Zusätzlich plant es währenddessen die Fahrtroute, basierend auf dieser auch das Fahrverhalten und trifft die dabei entstehenden Entscheidungen. Alle diese Aufgaben benötigen die Rechen- und Speicherressourcen des Embedded-Computers. Für die Entwicklung eines autonomen Fahrzeugs ist es daher von Interesse, ob dieser Computer die gestellten Aufgaben bewältigen kann oder an seine Grenzen gerät. Eine allgemeingültige Definition dieser Grenzen oder der Systemleistungsanforderungen gibt es nicht. Dennoch wird hier der Versuch unternommen, die Leistung des autonomen Systems anhand der CPU- und der Speicherauslastung sowie der Verarbeitungszeit pro Bild zu analysieren. Damit soll abgeschätzt werden, ob die Leistung des *Raspberry Pi 3 Model Bs* für die Ausführung des hier entwickelten Programms ausreicht und ob noch Raum für die Implementierung weiterer Entwicklungen ist.

Die Systemauslastung wird in dieser Arbeit mit dem *Linux*-Programm *htop* überwacht. Dazu wird nicht die CPU-Auslastung des hier entwickelten Programms beobachtet, sondern die des gesamten Systems. Dabei werden auch alle anderen laufenden Systemprozesse beachtet, wie zum Beispiel die *Linux*-Benutzeroberfläche oder die *Linux*-Treiber. Schließlich benötigt das hier entwickelte Programm das Betriebssystem und die damit ausgeführten Prozesse, um selbst zu funktionieren. Gemessen wird die aktuelle CPU-Auslastung als Mittelwert über alle vier Prozessorkerne des *Raspberry Pis* in Prozent. Die letztere der Angaben ist dabei nicht über alle Kerne gemittelt. Das autonome Programm wird über die SSH-Schnittstelle samt VNC-Server und *htops* nach dem Betriebssystemstart in jedem zu evaluierenden Betriebsmodus für 10 Minuten ausgeführt. Es wird erwartet, dass der *Standby*-Modus die geringste Auslastung aufweist, weil in diesem nur das Modul zur Kamerabildfassung und das Modul für die graphische Benutzeroberfläche gestartet werden. Die CPU-Auslastung schwankt hier zwischen 10 % und 15 %. Wird der VNC-Server für das Monitoring eingeschaltet, so erhöht sich der Wert auf bis zu 45%. Die höchste Auslastung ist hingegen im autonomen Fahrmodus zu erwarten, was durch Messungen bestätigt werden konnte. Hier schwankt der Prozentsatz der CPU-Auslastung zwischen 75 % und 80 %. Die Auslastung durch die zusätzliche Übertragung über die VNC-Schnittstelle erhöht den Wert auf bis zu 90%. Die Ergebnisse werden im Folgenden genauer betrachtet, wobei in dieser Messung keine FNC-Übertragung aktiv war.

Über die Konfigurationsdatei kann die Anzahl der aufzunehmenden Bilder pro Sekunde (FPS) eingestellt werden. Das hier verwendete Kameramodell unterstützt die in Tabelle 1 aufgelisteten FPS. Messungen bestätigen die vor den Tests getroffene Annahme, dass eine höhere FPS-Zahl zu einer höheren CPU-Auslastung führt. Der Anstieg der Auslastung ist allerdings im autonomen Fahrbetrieb nicht so hoch wie im *Standby*-Modus. Um den Grund

dafür zu finden, ist zu analysieren, wie lange die Verarbeitung eines Bildes im autonomen Fahrmodus dauert.

Tabelle 1: CPU-Auslastung im *Standby*- und autonomer Fahrmodus

FPS	Standby	Autonomous Driving
30	33 %	93 %
20	24 %	93 %
15	21 %	91 %
10	16 %	88 %
7,5	14 %	88 %

Diese Bearbeitungszeit kann durch das Messen der Dauer zwischen der Aufnahme eines Bildes und dessen Ausgabe abgeschätzt werden. Für diese Messung wird in dem hier implementierten Programm ein Zeitstempel benötigt, der für jedes erfasste Kamerabild mitaufgezeichnet wird. Bei der Bildausgabe im *uiShowThread* wird dann die Differenz zwischen der aktuellen Zeit und dem Erfassungszeitpunkt berechnet. Zur Kontrolle, ob das System tatsächlich die eingestellte Anzahl an Bildern pro Sekunde aufzeichnet, wird auch im *CameraImageAcquisitor* die Zeit zwischen den Aufnahmen betrachtet. Es wurde festgestellt, dass die FPS-Werte einer gewissen Schwankung unterliegen, weil die Threads vom Betriebssystem nicht immer im gleichen Zeitabstand ausgeführt werden. Aus diesem Grund werden die Messungen hier über eine Zeit von 5 Sekunden gemittelt. In der Tabelle 2 werden die Ergebnisse aller ermittelten FPS angegeben. Sie enthält die FPS bei der Bildaufzeichnung sowie die bei der Ausgabe. Ebenso wird angegeben, wie viele Prozent aller aufgenommenen Bilder auch verarbeitet werden konnten.

Tabelle 2: FPS im autonomen Fahrbetrieb

Aufzeichnung	Ausgabe	Verarbeitet	Antwortverzögerung
30	15	50 %	2 s
20	15	75 %	1,33 s
15	15	100 %	1 s
10	10	100 %	1 s
7,5	8	100 %	1 s

In der Spalte für die Antwortzeit der obigen Tabelle wird aus den gemessenen FPS, die Zeit für die Verarbeitung eines einzelnen Bildes berechnet. Diese Zeitangaben dienen als grobe Schätzung und Richtwert, um eine optimale FPS-Einstellung zu finden. Aus den Ergebnissen lässt sich ableiten, dass das System nicht mehr als 15 FPS schafft. Es ist also nicht sinnvoll, den Prototyp mit mehr FPS zu betreiben. Die errechnete Antwortzeit ist trotz gemittelter Messwerte nicht garantiert. Das liegt daran, dass, wie in Kapitel 3.6 erläutert wurde, das hier eingesetzte Betriebssystem nicht deterministisch arbeitet. Werden mehr Hintergrundprogramme oder zusätzliche Funktionen gestartet, so verlängert sich auch die Reaktionszeit des Systems.

Während überprüft wurde, ob die eingestellte Anzahl der FPS tatsächlich zum gewünschten Ergebnis führen, wurde festgestellt, dass die hier verwendete Webkamera eine automatische Belichtungseinstellung vornimmt. Diese hat Auswirkungen auf die von der Kamera gelieferte Anzahl an Bildern pro Sekunde. Eine größere Blendeneinstellung führt zu sichtbar und messbar kleineren FPS. Das bedeutet, dass, obwohl an der Kamera 30 FPS eingestellt werden, bei geringer Beleuchtung nur 8 FPS aufgenommen werden. Grund dafür ist, dass bei diesem Gerät die Blendengröße mit der Belichtungszeit zusammenhängt. Daraus folgt auch, dass ein Betrieb des autonomen Fahrzeugs bei mehr Umgebungslicht zu einer besseren Leistung der Bilderfassung und damit des Systems führt. Diese Erkenntnis ist besonders für den Betrieb des Prototyps im Innenraum relevant.

Um das Belichtungsverhalten der Kamera zu analysieren, wurden hier die *Linux*-Programme *v4l2-ctl* und *qv4l2* verwendet. Ersteres gibt Einstellungsmöglichkeiten und Informationen über am Embedded-Board angeschlossene Kameras aus. Beim Auslesen der Werte für die angeschlossene Kamera wurde festgestellt, dass sie sowohl über den oben erwähnten automatischen Blendenmodus, als auch über einen manuellen verfügt. *OpenCV* bietet eine Möglichkeit, mit `set(CAP_PROP_EXPOSURE, value)` manuelle Blendenwerte einzustellen, welche auch von der Kamera akzeptiert und umgesetzt werden. Das versetzt allerdings die hier eingesetzte Kamera in den manuellen Betrieb und eine Umschaltung zurück zur Automatik ist mit keiner verfügbaren *OpenCV*-Funktionen möglich. Auch ist zu beachten, dass weder der Kamerahersteller noch der *Linux*-Treiber vollständige Angaben für die einstellbaren Blendenwerte zur Verfügung stellen. Aus dem, vom Treiber angegeben Wertebereich für die Blende von 5 bis 20.000, führen nur bestimmte Werte zu einer Blendeneinstellung. Eine genaue Analyse dieser Werte ergibt, dass ausschließlich Werte in der Form von 2^n mit $0 < n < 12$ zum gewünschten Ergebnis führen. Werden diese Zahlen nun über die *OpenCV*-Funktion `set()` eingegeben, führt dies dennoch zu keinem gewünschten Ergebnis. In den Tests konnte festgestellt werden, dass die Zahlen zuerst mit

$$D = \frac{\frac{max}{2^n} - min}{max - min}$$
, wobei $min = 5$ und $max = 20.000$, skaliert werden müssen. Das zweite oben erwähnte Programm *qv4l2* bietet eine graphische Benutzeroberfläche für die Manipulation der Kameratreiberwerte und ist somit als ergänzendes Analysewerkzeug für die hier vorgeschlagene Entwicklungsumgebung nützlich. Sie ermöglicht zusätzlich, die Kamera vom manuellen Blendenmodus wieder in den automatischen umzustellen. Mit dieser Applikation wurden auch die oben genannten Werteinstellungen ermittelt.

In den durchgeführten Tests zeigte das System insgesamt eine gute Performanz. Das Ziel, eine modulare Entwicklungsumgebung für das autonome Fahren zu erstellen, wurde erreicht. Sie konnte mit dem hier entwickelten mobilen Prototyp, welcher Teil dieser Umgebung ist, erfolgreich ausgeführt und getestet werden. Damit konnte auch eine Applikation zum autonomen Fahren entwickelt werden. Allerdings wurden in den Tests die Grenzen des Systems sichtbar. Diese werden in Kapitel 6.5 wieder aufgegriffen und gemeinsam mit dem Ausblick auf Weiterentwicklungsmöglichkeiten besprochen.

6.4 Safety

Bei der Erstellung des Prototyps dieser Arbeit wurde die Umsetzung von Sicherheitsvorkehrungen beachten. Solche wurden sowohl in der Hardware- als auch in der Softwarerealisierung eingebaut und werden im Folgenden besprochen.

Das hier vorgestellte autonome Modellfahrzeug verfügt über Schutzvorrichtungen für elektronischen Bauteile, wie das stabile Gehäuse, in dem sich das Embedded-Board und der Akku befindet. Die Motorsteuerung ist an der Außenseite des Gehäuses befestigt und dadurch gegen Stöße geschützt. Die Kamera wird durch ein stabiles und fest verschraubtes Stativ gehalten. Alle tragenden Bauteile sind zusätzlich durch Schrauben befestigt, um das Fahrzeug robuster zu machen. Wie in Kapitel 4 beschrieben, wurden nach Kollisionen während der Testläufe keine Schäden am Fahrzeug festgestellt.

Neben diesen Schutzvorrichtungen der Hardware, verfügt der Prototyp auch über einige softwaretechnische Mechanismen, um das potentielle Sicherheitsrisiko für den Bedienenden und für die Umwelt zu reduzieren. Für diese Arbeit wurde angenommen, dass der sicherste Betriebszustand der Stillstand des Fahrzeugs ist. Im Folgenden werden die in den jeweiligen Kapiteln erwähnten Sicherheitsmaßnahmen zusammengefasst und die Grenzen dieser sowie mögliche Lösungsansätze diskutiert.

Wie in Kapitel 3.3 erwähnt, wurde bereits während des Entwurfs darauf geachtet, dass das System in keinen undefinierten Zustand übergehen darf. In den Betriebsmodi in Kapitel 5.8 werden die Programmübergänge beschrieben, über welche im Fehlerfall in den Fehlerzustand gewechselt wird. Beim Erreichen des Fehlerzustands wird durch den Wechsel zum Beendigungsmodus die Motorsteuerung zurückgesetzt und abgeschaltet. Dadurch bleibt das Fahrzeug stehen und der sicherste Zustand ist erreicht. Durch die in Kapitel 5.1 beschriebene Funktion *signalHandler()* werden Betriebssystemssignale, welche das Programm unterbrechen würden, abgefangen. Auch hier wird in gleicher Weise die Motorsteuerung zurückgesetzt und abgeschaltet.

Während des autonomen Fahrbetriebs bleibt das Fahrzeug, wenn es keine Fahrspurmarkierungen erkennt, stehen. Dies ist notwendig, damit das Fahrzeug bzw. dessen Umwelt nicht durch die Fahrt auf unkontrolliertem Gelände Schaden nimmt. Ein Eingriff durch den Benutzer ist notwendig, indem dieser das Fahrzeug manuell oder per Fernsteuerung neu auf der Fahrbahn positioniert. Erst dann kann die autonome Fahrt wieder aufgenommen werden. Diese ist, wie in Kapitel 5.8.2 beschrieben zum Schutz des Fahrzeugs und der Personen im Umfeld nur mit einer niedrigen Motordrehzahl implementiert.

Das autonome Fahrzeug verfügt über eine Ultraschall-Abstandsmessung zur Erkennung von Hindernissen vor dem Fahrzeug. Dabei wurde ein Sicherheitsabstand von 25 cm festgelegt, innerhalb dessen das Fahrzeug mit der im autonomen Fahrbetrieb gewählten Geschwindigkeit gefahrlos anhalten kann. Ist, wie in Kapitel 5.3.4 beschrieben, kein Ultraschallsensor angeschlossen oder der angeschlossene Sensor schadhaft, dann werden sehr kleine Abstandswerte von unter einem Millimeter ausgelesen, die ein sofortiges Anhalten auslösen. Dies ist ebenfalls eine Sicherheitsvorkehrung, damit das Fahrzeug nicht ohne Hinderniserkennungsmechanismus fährt und so mögliche Kollisionen und Schäden vermeiden werden. Es kann vorkommen, dass ein Hindernis während der

Fahrt innerhalb des definierten Sicherheitsabstandes vor dem Auto auftaucht. Zwar ist in diesem Fall das Anhalten des Fahrzeugs ohne Kollision nicht mehr möglich, dennoch versucht das Fahrzeug zum Stillstand zu kommen, um weitere Schäden zu minimieren.

Wird ein Programm während der Entwicklung getestet, kann dies zu Abstürzen, Unterbrechungen und Exceptions führen. Dabei ist es jedoch möglich, dass der Motortreiber den letzten erhaltenen Befehl weiter ausführt. Für solche Fälle wäre ein physischer Not-Stopp-Knopf sinnvoll, um das System zu schützen und das Fahrzeug in den sichersten Zustand, also den Stillstand, zu bringen. Im derzeitigen Entwicklungsstand der vorliegenden Arbeit wurde der Not-Stopp-Knopf jedoch noch nicht berücksichtigt, da dies über den zeitlichen Rahmen sowie die inhaltliche Zielsetzung der Arbeit hinausgehen würde. In einer Weiterentwicklung des Prototyps sollte ein solcher jedoch eingebaut werden.

6.5 Grenzen und Ausblick

Im folgenden Abschnitt werden die bereits angemerkteten Grenzen der Arbeit, die anhand der besprochenen Ergebnisse deutlicher sichtbar wurden, wieder aufgegriffen. Dazu werden alternative Umsetzungsmöglichkeiten erwähnt sowie ein Ausblick für offene Forschungsfelder und Weiterentwicklungsmöglichkeiten gegeben.

Während der Entwicklung der Arbeit und der Evaluierung der Ergebnisse konnte beobachtet werden, dass die in Kapitel 3.7 gewählte Monitoring-Lösung bei Überlastung zum Teil nur stockend arbeitet. Ein Grund dafür ist, dass der *Raspberry Pi* derzeit das gesamte graphische Desktopbild des Betriebssystems *Raspbian*, inklusive der Benutzerschnittstelle der hier entwickelten Applikation, an den Monitoring-Computer schickt. Um dieses darzustellen, muss ein VNC-Server gestartet werden, der diese Übertragung vornimmt. Da der Modus des autonomen Fahrens bereits eine hohe CPU-Auslastung verursacht, gerät das Monitoring bei dessen zusätzlicher Ausführung ins Stocken. Die hier gewählte Lösung war für die vorliegende Arbeit zwar ausreichend, doch könnte die Datenübertragung zwischen dem Monitoring-PC und dem Fahrzeug in einem zukünftigen Entwicklungsschritt verbessert werden. Dafür könnte eine Server-Client-Applikation erstellt werden, die das graphische Zusammensetzen der Benutzeroberfläche auf den Monitoring-Computer auslagert. Damit würde keine komplette graphische Desktopumgebung mehr auf dem Embedded-Computer erstellt werden müssen, wodurch dieser entlastet werden würde. Gleichzeitig wäre die Kommunikation zwischen den Computern effizienter und damit die Reaktion des Embedded-Boards schneller.

Wie in Kapitel 4.3 erwähnt, wird für die Abstandsmessung zu einem Hindernis ein Ultraschallsensor eingesetzt. Dieser hat jedoch einen eingeschränkten Erfassungsbereich. Durch seine Positionierung an der Fahrzeugfront können nur Hindernisse, die sich auf einer geraden Strecke genau vor dem Fahrzeug befinden, erfasst werden. Damit ist das Erkennen von Hindernissen, die sich in einer Kurve befinden, erst sehr spät möglich. Dadurch kann der nötige Sicherheitsabstand für eine gefahrfreie Bremsung nicht eingehalten und eine Kollision mit dem Objekt nicht verhindert werden. Für eine

Weiterentwicklung des Prototyps, wird vorgeschlagen, zwei bis vier Ultraschallsensoren mit unterschiedlichen Abstrahlwinkeln zu verbauen. Dadurch könnte ein breiterer Bereich vor dem Fahrzeug kontrolliert und Hindernisse in Kurven früher detektiert werden. Auch könnten zusätzliche Ultraschallsensoren an der Rückseite des Fahrzeugs angebracht werden, wodurch neue Applikationen für das autonome Fahren, wie das sichere Rückwärtsfahren, das Einparken oder Überholen implementiert werden könnten. Zu beachten ist auch, dass der hier verwendete Sensor nur Distanzen bis zu 4 m erfassen kann. Dies beschränkt aufgrund des gebotenen Sicherheitsabstandes und des Bremsweges die maximale Geschwindigkeit des Fahrzeugs. Soll dieses später schneller fahren, müssen Sensoren mit einer größeren Reichweite verbaut werden. Weiters ist zu erwähnen, dass *Raspbian* nicht deterministisch arbeitet. Dadurch kann keine genaue Zeitmessung und damit auch keine ganz präzise Abstandsmessung durchgeführt werden. Soll das System zukünftig jedoch entlastet und die Abstandsmessung noch genauer durchgeführt werden, könnte ein Micro-Controller eingesetzt werden. Dieser sollte mit dem Sensor verbunden sein und dem Embedded-Board die fertigen Abstandsdaten im gewünschten Format zur Verfügung stellen. Die Daten könnten bei Bedarf vom Embedded-Computer abgerufen werden, wodurch dieser entlastet wird. Außerdem kann der Micro-Controller die Abstandsmessung präziser durchführen, da er eine genauere Zeitmessung vornehmen kann.

In Kapitel 5.5 wird erwähnt, dass für die vorliegende Arbeit auf eine Geschwindigkeitserfassung verzichtet wurde. Da hier lediglich eine Fahrbahn verfolgt werden soll, ist die Geschwindigkeit an sich nicht relevant. Sie wird indirekt durch die Beschleunigung gesteuert und beschränkt, was für die Zielsetzung dieser Arbeit ausreichend ist. Für die Weiterentwicklung des Prototyps und um ein dynamisches Fahrverhalten zu implementieren, ist die Möglichkeit einer Geschwindigkeitsmessung jedoch notwendig und sinnvoll. Dazu eignen sich beispielsweise Encoder-Scheiben, die am Motor oder an den Fahrzeugachsen angebracht werden. Diese arbeiten mit Licht- oder Hallsensoren und messen die Umdrehungsgeschwindigkeit der Räder.

Wie aus den Kapiteln 6.2.1 und 6.2.2 hervorgeht, ist das hier implementierte Fahrbahnerkennungsverfahren für die in dieser Arbeit gewählte niedrige Fahrgeschwindigkeit ausreichend. Dadurch dass für jedes Bild nur gerade Linienabschnitte erkannt werden, kann der Straßenmarkierungsverlauf in Kurven jedoch nur grob abgeschätzt werden. Bei langsamen Fahrgeschwindigkeiten ist diese Abschätzung ausreichend, um dem Straßenverlauf folgen zu können. Für die Fahrt mit höheren Geschwindigkeiten ist jedoch eine genauere Positionierung des Fahrzeugs auf der Fahrspur nötig, um in Kurven nicht von dieser abzukommen. Dafür ist auch eine genauere Erfassung des Kurvenverlaufs notwendig. Während der Erstellung dieser Arbeit wurde bereits Versuche unternommen, eine Straßenmarkierung mit mehr als nur zwei Punkten pro Bild zu beschreiben, was zu besseren Ergebnissen in der Erfassung geführt hat. Dabei werden Linien in kleineren Bildabschnitten analysiert. In den Versuchen wurden drei Bildabschnitte gewählt, wodurch pro Bild drei Linienabschnitte mit je zwei Punkten errechnet werden können. Für die Geradeausfahrt konnten mit dieser Maßnahme gute Ergebnisse erzielt werden. In Kurven jedoch tritt das Problem auf, dass beispielsweise Linienabschnitte der rechten Fahrbahnmarkierung, die in den unteren Bildabschnitten noch

auf der rechten Bildhälfte verlaufen, im obersten Bildabschnitt durch die Kurvenkrümmung in der linken Bildhälfte zu sehen sind. Daraus resultieren Probleme in der Interpretation der Linien und damit in der Berechnung der Trajektorie. Diese wurden für die vorliegende Arbeit jedoch noch nicht behoben, da sich das ursprüngliche Fahrbahnverfolgungsverfahren für die gewählte Fahrgeschwindigkeit als besser erwiesen hat. Jedoch wird die hier beschriebene Verbesserung des Verfahrens für spätere Weiterentwicklungen empfohlen, da das autonome Fahrzeug dadurch Kurven auch mit höheren Geschwindigkeiten stabil durchfahren kann. Eine weitere Verbesserungsmöglichkeit ist, das Bild nur im untersten Bildabschnitt in rechts und links zu unterteilen, um die sichtbaren Linien richtig interpretieren zu können. Im nächsten Bildabschnitt wird der Endpunkt der Linie aus dem ersten Bildabschnitt mit dem neuen Anfangspunkt verglichen. So können gekrümmte Linienverläufe zu der richtigen Markierung zugeordnet werden, auch wenn sie sich im aktuellen Bildausschnitt auf der gegenüberliegenden Bildhälfte befinden. Je kleiner die Bildabschnitte bzw. Suchfenster pro Bild sind, desto mehr Punkte zur Beschreibung einer Markierung sind vorhanden. Dadurch kann der Kurvenverlauf genauer ermittelt und die Trajektorie präziser bestimmt werden. Abgesehen von dem hier geschilderten Fahrbahnerkennungsverfahren gibt es noch viele andere Ansätze, wie in Kapitel 2.2.1 bereits erwähnt wurde. In den dort genannten Arbeiten finden sich Vergleiche verschiedener Verfahren zu Fahrbahnerkennung, die ebenfalls mit der hier vorgeschlagenen Entwicklungsumgebung umgesetzt werden können. Für die Weiterentwicklung werden daher die Arbeiten von Kaur et al., Kumar et al. und Maldlik et al. (vgl. [JK05], [YGD11], [SL14], [W18c]) empfohlen.

Wird die oben beschriebene Geschwindigkeitsmessung und die bessere Kurvenerkennung umgesetzt, kann im nächsten Entwicklungsschritt, wie in Kapitel 5.5 erwähnt, auch ein dynamisches Fahrverhalten implementiert werden. Diese passt die Geschwindigkeit des Fahrzeugs an den Trajektorienverlauf an und führt eine genauere Lenkung zur besseren Verfolgung des derselben durch. Das bedeutet, dass engere Kurvenradien langsamer durchfahren werden als größere. Für die in dieser Arbeit gewählte Geschwindigkeit war diese Entwicklung bisher nicht notwendig, sie wird jedoch für spätere Weiterentwicklungen empfohlen.

In den Ergebnissen in Kapitel 6.2.3 wurde ersichtlich, dass die Stoppschilderkennung zwar funktioniert, aber nicht ganz zuverlässig ist. Im Bild wurden Stoppschilder erkannt, wo jedoch keine Schilder aufgestellt wurden. Ein Neu-Trainieren des Klassifizierers könnte bessere Ergebnisse liefern. Auch das Testen anderer Verfahren könnte sinnvoll sein, wie z.B. die in Kapitel 2.2.2 genannten Verfahren HOG (vgl. [DT05]), SIFT (vgl. [L99]), SURF (vgl. [BET06]), FAST (vgl. [RD05], [RD06]) und ANN (vgl. [BL07], [GBC16]).

Aus Kapitel 6.3 geht hervor, dass der im Rahmen dieser Arbeit eingesetzte Embedded-Computer mit den hier umgesetzten Verfahren im autonomen Fahrbetrieb komplett ausgelastet ist. Die Implementierung der oben beschriebenen Verbesserungsvorschläge würden den Computer jedoch über seine Leistungsgrenzen hinaus beanspruchen. Daher wird für die Weiterentwicklung der Einsatz eines leistungsfähigeren Computers empfohlen. Wenn dieser Computer über eine eigene GPU verfügt, könnte auch die Bildverarbeitungs- und die Maschinenlernverfahren mit OpenCV beschleunigt werden. Beispielsweise würde

dafür die von *NVIDIA* angebotene *JETSON*-Serie in Frage kommen (vgl. [N18c]), die unter anderem von *RACECAR* (vgl. [MIT18]) eingesetzt wird.

In den Kapiteln 4.2 und 5.3.1 wird erwähnt, dass für die Weiterentwicklung der Plattform eine Kamera mit einer besseren *Linux*-Treiber-Unterstützung, mit einem Weitwinkelobjektiv sowie mit einer besseren Fixierungsmöglichkeit des Einstellwinkels sinnvoll wäre. Dadurch würde die Veränderung mehrerer Kameraeinstellungen möglich sowie die Erfassung eines größeren Raums um das Fahrzeug herum. Auch würde sich beim Transport des Fahrzeugs der einmal eingestellte Kamerawinkel nicht wieder verstehen, wodurch die Notwendigkeit einer Neukalibrierung nach dem Aufbau entfällt.

Wie die Sicherheitsnormen IEC 61508 und ISO 26262, die in Kapitel 3.3 erwähnt wurden, vorschreiben, wäre es sinnvoll, der Entwicklungsumgebung einen zusätzlichen physischen Not-Stopp-Knopf hinzuzufügen. In Kapitel 6.4 wird erklärt, dass trotz der Implementierung von Sicherheitsmaßnahmen im Softwaresystem unvorhergesehene Fehler passieren können. Diese stellen Sicherheitsrisiken für den Benutzer, das Fahrzeug und die Umwelt dar. Ein physischer Not-Stopp-Knopf, der die Stromzufuhr der Motoren unterbrechen kann, stellt sicher, dass das Fahrzeug beim Absturz der Software dennoch zum Stillstand gebracht werden kann. Ein weiterer Mechanismus zur Erhöhung der Sicherheit ist es, dem System eine *Watch-Dog*-Funktion hinzuzufügen. Diese könnte auf einem externen Micro-Controller laufen, das System überwachen und dieses bei einem Absturz oder einem unbekannten Fehler neu starten. Ist das Starten nicht möglich, soll auch hier die Stromversorgung der Motoren unterbrochen und damit das Fahrzeug angehalten werden.

7. ZUSAMMENFASSUNG

Aktuell werden von vielen Automobilherstellern weltweit Fahrassistenzsysteme in Serienfahrzeugen eingebaut. Gleichzeitig erforschen die Hersteller die Weiterentwicklung dieser Assistenzsysteme zum vollständig autonomen Fahrzeug. Dies soll vor allem die Sicherheit der Passagiere und anderer Verkehrsteilnehmer erhöhen und zum größeren Komfort beitragen. Zwar präsentieren die Demonstrationsfahrzeuge die Ergebnisse dieser Forschungen, doch stellen sie keine quelloffene oder erwerbbare Entwicklungsumgebung dar.

Die vorliegende Masterarbeit beschäftigt sich ebenfalls mit der Erforschung des autonomen Fahrens. Dazu wurde eine modulare, mobile und quelloffene Entwicklungsumgebung erstellt, die als autonomes Modelfahrzeug umgesetzt wurde. Dieses ist Teil der Entwicklungsumgebung, dient selbst als Entwicklungswerkzeug und kann zur Durchführung von HIL-Tests für Softwareapplikationen für das autonome Fahren verwendet werden. Diese kostengünstige und quelloffene Plattform soll anderen Entwicklern den Zugang zur weiteren Erforschung des autonomen Fahrens und seiner Teilaufgaben erleichtern, da auf dem Prototyp Tests ohne Risiko durchgeführt werden können. Deren Erkenntnisse können später auf Fahrzeugen im vollen Maßstab eingesetzt werden. Dazu werden die Ergebnisse und Erkenntnisse dieser Arbeit veröffentlicht und der Quellcode öffentlich zugänglich gemacht. Die aktuellste Version mit weiteren Verbesserungen kann unter https://github.com/stabacariu/autonomous_driver aufgerufen werden.

Im ersten Kapitel dieser Arbeit wird zunächst der Stand der Technik sowie verwandte Arbeiten betrachtet. Auch werden hier die Motivation, die Aufgabenstellung und die Methoden für die Durchführung derselben besprochen.

Das zweite Kapitel widmet sich den physikalischen, hardware- und softwaretechnischen Grundlagen des autonomen Fahrens, um die nötigen Bestandteile für dieses zu identifizieren. Dabei wird im ersten Schritt die Systemarchitektur analysiert, wobei die Sensorik, die Verarbeitung der Sensordaten, die Fahrplanung sowie die Fahrzeugsteuerung betrachtet werden. Weiters wird der Fokus auf die digitale Bildverarbeitung gelegt, da diese als wesentlicher Bestandteil autonomer Systeme identifiziert werden konnte. Hier wurden die Grundlagen der Fahrbahn- sowie der Verkehrsschilderkennung und der optischen Abstandsmessung erarbeitet. Schließlich wurde eine Auswahl an Softwarewerkzeugen zur Programmierung und -evaluation für das selbstständige Fahren getroffen.

Kapitel drei beinhaltet das Konzept und das Design der in der vorliegenden Arbeit erstellten Entwicklungsumgebung. Anhand der im vorigen Kapitel analysierten Systemarchitekturen wurde hier ein Systemdesign entworfen und die dafür nötigen Hardwarekomponenten gewählt. Weiters wurden mögliche Systemzustände definiert und ein Regelungsablauf entworfen. Für die Softwareentwicklung wurden die zu verarbeitenden Daten identifiziert und dafür notwendige Klassen designet. Für den Betrieb der Entwicklungsumgebung wurde ein entsprechendes Betriebssystem und die dafür notwendigen Softwarebibliotheken installiert sowie eine Möglichkeit des Monitorings vorgestellt.

In Kapitel vier wird schließlich die Realisierung der Hardware des Prototyps präsentiert. Dieser besteht aus einem handelsüblichen Modellfahrzeug, welches für die Bewältigung der gestellten Aufgaben umgebaut wurde. Dazu wurden stabile Befestigungsmöglichkeiten für die Elektronik und Sensorik geschaffen. Für die Umwelterfassung wurden eine USB-Webkamera und ein Ultraschallsensor gewählt, welche am Fahrzeug befestigt wurden. Für die Verarbeitung der Sensordaten wurde ein *Raspberry Pi 3 Model B* verbaut. Für die Motorsteuerung wurde ein PWM-Motortreiber installiert, an welchem der Lenkungs- und der Antriebsmotor angeschlossen wurden. Außerdem wurden zwei Akkumulatoren verbaut. Einer dient der Versorgung des Fahrzeugantriebs, der andere versorgt die Elektronik.

Das fünfte Kapitel beschreibt die Softwarerealisierung der Entwicklungsumgebung. Für die Bewältigung der einzelnen Teilaufgaben des autonomen Fahrens wurden verschiedene Module entwickelt, die von Systemmodi aufgeführt werden.

Im *Konfigurationsmodul* können wichtige Kameraparameter kalibriert werden, die für die weitere Objekterkennung benötigt werden. Die intrinsische Kalibration kalkuliert die Brennweite und die optische Achse der Kamera sowie den Verzerrungskoeffizienten für die Entzerrung des Eingabebildes, welches aufgrund der physikalischen Eigenschaften des Objektivs verzerrt sind. Die extrinsische Kalibration berechnet eine Homographiematrix, die benötigt wird, um eine inverse Perspektivtransformation durchzuführen, mittels derer das Eingabebild so verzerrt wird, dass die Fahrbahn in Vogelperspektive erscheint.

Das *Bilderfassungsmodul* erfasst Bilder, die durch die Kamera aufgezeichnet wurden und speichert sie für die Weiterverarbeitung durch die anderen Module ab. Das *Fahrbahnerkennungsmodul* erkennt Straßen anhand von Fahrbahnmarkierungen. Dabei werden mithilfe des *Canny*- und des *Hough*-Filters Linien entlang der Markierungen erkannt, die mittels *Kalman*-Filter zuverlässiger ermittelt werden. Das Ergebnis der Fahrbahnerkennung sind die linke und die rechte Begrenzungslinie der Fahrspur, die verfolgt werden soll. Das *Verkehrsschilderkennungsmodul* detektiert Stoppschilder mithilfe eines Objekterkennungsverfahrens. In dieser Arbeit wurde dafür der *Haar*-Kaskaden-Klassifizierer eingesetzt. Die so erkannten Stoppschilder werden für die weitere Verarbeitung als rechteckige Markierungen abgespeichert. Im *Modul für die Hinderniserkennung* werden mit dem Ultraschallsensor Distanzen vor dem Fahrzeug gemessen, wobei auch mögliche Hindernisse auf der Fahrspur erfasst werden.

Das *Fahrtplanungsmodul* bestimmt die Reaktion des Fahrzeugs auf Basis der ermittelten Sensordaten. Hier wird die Trajektorie aus den Informationen der Fahrbahnerkennung bestimmt. Der Fahrtroutenverlauf wird mittig zwischen den erkannten Begrenzungslinien kalkuliert. Dazu wird zur Prognose des Weiteren Fahrbahnverlaufs erneut der *Kalman*-Filter eingesetzt. Auch das Ergebnis der Stoppschilderkennung wird hier überwacht. Wird ein Stoppschild innerhalb eines bestimmten Sicherheitsabstands zum Fahrzeug erkannt, leitet das Modul eine Bremsung ein. Dasselbe passiert, wenn der Abstand zu einem erkannten Hindernis auf der Fahrbahn ebenfalls kleiner als dieser Sicherheitsabstand wird.

Das *Fahrzeugsteuerungsmodul* berechnet aus der Trajektorie den aktuell notwendigen Lenkwinkel und die Beschleunigung des Fahrzeugs. Die ermittelten Werte werden in Steuerbefehle umgewandelt und an den Motortreiber geschickt. Dieser steuert in weiterer Folge den Servomotor der Lenkung sowie die MCU des Antriebsmotors.

Im *Modul für die Fahrzeugfernsteuerung* werden Benutzereingaben, welche über die graphische Benutzerschnittstelle oder die Tastatur eines Monitoring-Computers eingegeben wurden zur Steuerung des Fahrzeugs verwendet.

Die *graphische Benutzerschnittstelle* dient der Kommunikation zwischen einem Benutzer und dem System sowie zur Überwachung durch diesen. Sie setzt graphische Elemente abhängig vom aktuellen Systemzustand zusammen und gibt sie über ein Display am Fahrzeug oder einen Monitoring-Computer aus.

Als nächstes werden die *Systemmodi* beschrieben, die die genannten Module parallel als Threads ausführen. Der *Initialisierungsmodus* wird nach dem Systemstart aufgerufen, worin die aktuellen Konfigurationsdaten aus einer XML-Datei geladen werden. Anschließend startet der Modus den Systemzustand *Standby*. In diesem wird dem Benutzer das aktuelle Kamerabild und ein Schaltflächenmenü angezeigt, über das dieser in andere Modi wechseln kann. Im *autonomen Fahrmodus* werden die Module der Erkennungsverfahren, der Fahrtplanung und der Fahrzeugsteuerung durchgeführt. Der Benutzer kann die Ergebnisse auf der entsprechenden graphischen Schnittstelle mitverfolgen. Der *Entwicklungsmodus* dient zur Evaluierung neuer Applikationen für das autonome Fahren und führt die Erkennungsmodule aus. Das Fahrzeug selbst wird jedoch vom Benutzer ferngesteuert. Im *Fernsteuerungsmodus* hingegen wird keine Erkennung durchgeführt, sondern dem Benutzer nur das aktuelle Kamerabild gezeigt, anhand dessen er das Fahrzeug mittels der Fernsteuerung navigiert. Im *Konfigurationsmodus* kann der Benutzer die Kalibrationsroutinen starten. Diese werden in den *Kalibrationsmodi* durch die Ausführung der entsprechenden Module durchgeführt. Der *Informationsmodus* enthält Daten über den aktuellen Stand der Entwicklung, die über die graphische Benutzeroberfläche angezeigt werden. Der *Fehlermodus* fängt Fehlermeldungen ab und initiiert den Wechsel in den *Beendigungsmodus*, der das Modelfahrzeug zum Stillstand bringt und das System sicher beendet.

Im letzten Kapitel werden die Ergebnisse der Arbeit, Sicherheitsaspekte sowie die Systemgrenzen besprochen und ein Ausblick für weitere Entwicklungsmöglichkeiten gegeben. Zur Evaluierung der erstellten Entwicklungsumgebung und des Prototyps wurde eine Teststrecke befahren und die Leistungsfähigkeit der einzelnen Module sowie des Systems überprüft. Dabei konnte gezeigt werden, dass das Geradeausfahren entsprechend der Erwartungen durchgeführt werden konnte. Auch die Kurvenfahrt konnte mit der hier gewählten langsamen Geschwindigkeit durchgeführt werden, wobei das Verfolgen der Kurve bei kleinen Kurvenradien ungenauer wurde. Hier können in späteren Entwicklungen genauere Kurvenerkennungsverfahren eingesetzt werden. Die Stoppschilderkennung funktionierte zwar in den meisten Fällen, sollte aber für eine Verbesserung des Systems weiterentwickelt werden, da manchmal Schilder erkannt wurden, obwohl auf der Fahrbahn keine platziert waren. Die Hindernisdetektion verlief trotz einiger Messfehler positiv. Aufgrund ungenauer Signallaufzeitmessungen durch das hier eingesetzte Embedded-Board, kam es bei einzelnen Messungen zu falschen Ergebnissen. Diese könnten z.B. durch einen *Kalman*-Filter verbessert werden. Die Fehler könnten auch durch eine bessere Zeitmessungsmöglichkeit, beispielsweise durch einen zusätzlichen Micro-Controller, behoben werden. Des Weiteren wurde die Systemleistung, also die CPU- und die Speicherauslastung analysiert. Dabei erwies sich diese bei der Ausführung des hier

entwickelten Programms als ausreichend. Optimale Ergebnisse konnten dabei im autonomen Modus mit einer Bildverarbeitungsrate von 15 Bildern pro Sekunde erzielt werden. Aufgrund der hohen CPU-Auslastung wird für die Weiterentwicklung jedoch ein leistungsfähigeres System empfohlen. Der Arbeitsspeicher erwies sich für die Verarbeitung der generierten Daten ebenfalls als ausreichend.

Wie aus den Testergebnissen ersichtlich wurde, konnte die Zielsetzung dieser Arbeit, eine quelloffene und modulare Entwicklungsumgebung für das autonome Fahren zu erstellen, erreicht werden. Mit dieser können auch in Zukunft einfache Applikationen für die selbstständige Fahrt erstellt und evaluiert werden. Neben den oben genannten Verbesserungsmöglichkeiten für den hier entwickelten autonomen Prototypen, kann dieser auch erweitert werden. Dazu können beispielsweise andere Kameras, zusätzliche Sensorik und leistungsfähigere Steuerungscomputer eingesetzt und getestet werden.

Anhang

Quellcode

MAIN.CPP.....	127
AUTONOMOUS_DRIVING.HPP	128
AUTONOMOUS_DRIVER.CPP	129
MODULE	130
Module.hpp.....	130
Lane_data.hpp	131
Lane_data.cpp.....	134
Lane_detection.hpp.....	136
Lane_detection.cpp	141
Obstacle_data.hpp.....	148
Obstacle_data.cpp.....	150
Obstacle_detection.hpp	151
Obstacle_detection.cpp.....	152
Traffic_sign_data.hpp.....	153
Traffic_sign_data.cpp.....	155
Traffic_sign_detection.hpp	156
Traffic_sign_detection.cpp	157
Image_data.hpp	159
Image_data.cpp.....	160
Image_acquisitor.hpp	161
Camera_image_acquisitor.hpp.....	162
Camera_image_acquisitor.cpp	166
Trajectory_data.hpp	174
Trajectory_data.cpp.....	176
Path_planning.hpp	177
Path_planning.cpp.....	179
Remote_control.hpp	181
Remote_control.cpp	182
Vehicle_data.hpp	184
Vehicle_data.cpp	189
Vehicle_control.hpp	192
Vehicle_control.cpp	193
Motor_driver.hpp	195
Motor_driver.cpp	197

MODI	198
System_state.hpp	198
System_state.cpp	200
System_mode.hpp	201
Standby_mode.hpp	202
Standby_mode.cpp	203
Autonomous_mode.hpp	205
Autonomous_mode.cpp	207
Development_mode.hpp	209
Development_mode.cpp	211
Remote_control_mode.hpp	213
Remote_control_mode.cpp	214
Configuration_mode.hpp	216
Configuration_mode.cpp	217
Calibration_mode.hpp	219
Calibration_mode.cpp	221
About_mode.hpp	225
About_mode.cpp	226
Error_mode.hpp	227
Error_mode.cpp	228
Closing_mode.hpp	229
Closing_mode.cpp	230
USER INTERFACE.....	231
User_interface.hpp	231
User_interface.cpp	233
User_interface_state.hpp	235
User_interface_state.cpp	236
User_interface_mode.hpp	237
Ui_standby_mode.hpp	238
Ui_standby_mode.cpp	239
Ui_autonomous_mode.hpp	240
Ui_autonomous_mode.cpp	241
Ui_development_mode.hpp	242
Ui_development_mode.cpp	243
Ui_remote_control_mode.hpp	245
Ui_remote_control_mode.cpp	246
Ui_configuration_mode.hpp	248
Ui_configuration_mode.cpp	249
Ui_calibration_mode.hpp	250

Ui_calibration_mode.cpp	252
Ui_about_mode.hpp	254
Ui_about_mode.cpp	255
Ui_error_mode.hpp.....	256
Ui_error_mode.cpp.....	257
TOOLS	258
Configuration.hpp.....	258
Configuration.cpp.....	260
Image_filter.hpp	262
Image_filter.cpp	263

Main.cpp

```
/*
 * @file main.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 27.6.2017
 *
 * @mainpage An autonomous driving development platform
 *
 * This program is an autonomous driving development platform for a model
 * car. It offers a ready to go autonomous driving system and
 * functions for custom system development.
 *
 * The autonomous driver system is designed to work with a generic USB
 * webcam, a Raspberry Pi 2/3, a PWM-servo-module and an ultrasonic sensor.
 * The driving base is an off-the-shelf ready-to-run remote controlled model
 * car with an ESC, a brushed driving motor and a servo steering motor.
 * The radio controller of the R/C car is replaced with the PWM-servo-module.
 * This module is connected to the Raspberry Pi 2/3 and communicates via I2C
 * with it.
 */

#include <iostream>
#include <csignal>
#include <opencv2/opencv.hpp>
#include "autonomous_driver.hpp"

AutonomousDriver app;

void signalHandler (int signal)
{
    switch (signal) {
        case SIGINT: std::cout << "ERROR: Signal caught: SIGINT" << std::endl; break;
        case SIGQUIT: std::cout << "ERROR: Signal caught: SIGQUIT" << std::endl; break;
        case SIGTERM: std::cout << "ERROR: Signal caught: SIGTERM" << std::endl; break;
    }

    app.quit();
}

/**
 * @brief The main function is the programs starting point
 *
 * The main function initializes the configuration, the system state
 * machine and the user interface. Then it starts the system state machine.
 *
 * @param argc The input argument counter
 * @param argv The input argument vector
 * @return System state value
 */
int main (int argc, char *argv[])
{
    std::signal(SIGINT, signalHandler);
    std::signal(SIGQUIT, signalHandler);
    std::signal(SIGTERM, signalHandler);

    // Start system
    app.exec();

    // Quit system after execution ended
    app.quit();

    return 0;
}
```

Autonomous_driving.hpp

```
/**  
 * @file autonomous_driver.hpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 6.4.2018  
 */  
  
#ifndef AUTONOMOUS_DRIVER_HPP  
#define AUTONOMOUS_DRIVER_HPP  
  
#include <iostream>  
#include <atomic>  
#include "system_state.hpp"  
#include "standby_mode.hpp"  
  
/**  
 * @brief An autonomous driving application class  
 *  
 * This class defines an autonomous driving application.  
 */  
class AutonomousDriver {  
public:  
    ~AutonomousDriver() = default;  
  
    /**  
     * @brief Execute system  
     *  
     * This function executes the system. It launches the set state.  
     */  
    void exec (void);  
  
    /**  
     * @brief Quit system  
     *  
     * This function quits the system. It stops all running modes and  
     * with that all running modules.  
     */  
    void quit (void);  
  
private:  
    SystemState state {new StandbyMode()}; //!< Current system state  
};  
  
#endif // AUTONOMOUS_DRIVER_HPP
```

Autonomous_driver.cpp

```
/**  
 * @file autonomous_driver.cpp  
 * @author Sergiu-Petru Tabcariu  
 * @date 6.4.2018  
 */  
  
#include "autonomous_driver.hpp"  
#include "configuration.hpp"  
  
void AutonomousDriver::exec ()  
{  
    // Load config file to configurator instance  
    Configurator& config = Configurator::instance("../config/config.xml");  
    //~ Configurator& config = Configurator::instance(); //< Load default.xml  
    config.load();  
  
    state.run();  
  
    while (state.isRunning()) {  
        state.run();  
    }  
  
    quit();  
}  
  
void AutonomousDriver::quit ()  
{  
    if (state.isRunning()) {  
        state.quit();  
    }  
}
```

Modules

Module.hpp

```
/**
 * @file    module.hpp
 * @author  Sergiu-Petru Tabacariu
 * @date    5.6.2018
 */

/**
 * @defgroup module Modules
 * @brief Autonomous Driving System modules
 */

#ifndef MODULE_HPP
#define MODULE_HPP

#include <atomic>

//! @addtogroup module
//! @{
//! @{

/**
 * @brief A module interface class
 *
 * This class defines an module acquistor interface.
 */
class Module {
public:
    ~Module() = default;

    /**
     * @brief Run module
     *
     * This function runs a module. Implement this function end start it
     * as thread.
     */
//~ virtual void run (void) = 0;

    /**
     * @brief Quit module
     *
     * This function quits the module.
     */
    virtual void quit (void) { running = false; };

    /**
     * @brief Checks if module is running
     *
     * This function checks if the module is running.
     *
     * @return True if module is running, else false.
     */
    virtual bool isRunning (void) { return running; };

    /**
     * @brief Checks if module error has occurred
     *
     * This function checks if a module error has occurred.
     *
     * @return True if module error, else false.
     */
    virtual bool isError (void) { return error; };

protected:
    std::atomic_bool running {false};
    std::atomic_bool error {false};
};

//! @}
#endif // MODULE_HPP
```

Lane_data.hpp

```
/***
 * @file lane_data.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 25.8.2017
 */

#ifndef LANE_DATA_HPP
#define LANE_DATA_HPP

#include <iostream>
#include <opencv2/opencv.hpp>
#include <mutex>

//! @addtogroup lane_detection
//! @{
//! @}

/***
 * @brief A enum for road marking type
 */
enum MarkingType {
    MARKING_NONE,           //!< No road marking
    MARKING_SOLID,          //!< Single solid marking
    MARKING_DASHED,         //!< Single dashed marking
    MARKING_SOLIDDASHED,    //!< Double marking, left solid, right dashed
    MARKING_DASHEDSOLID,    //!< Double marking, left dashed, right solid
    MARKING_DOUBLEDASHED    //!< Double dashed marking
};

/***
 * @brief A struct for road marking
 */
struct RoadMarking {
    std::vector<cv::Point> points; //!< Points describing road marking course in an image
    MarkingType type; //!< Type of road marking

    /**
     * @brief Size of road marking point vector
     *
     * This function returns the size of a road marking point vector
     *
     * @return Size of point vector
     */
    int size (void) { return points.size(); };
};

/***
 * @brief A lane data class
 *
 * This class defines the lane data and synchronises the access to it.
 */
class LaneData {
public:
    ~LaneData() = default;

    /**
     * @brief Set left road marking line
     *
     * This function sets a left road marking line to the lane.
     *
     * @param line Road marking line
     */
    //~ void setLeftLine(RoadMarking line);

    void setLeftLine(cv::Vec4i line);

    /**
     * @brief Get left road marking line
     *
     * This function gets the left road marking line.
     *
     * @return Road marking line
     */
}
```

```

*/
//~ RoadMarking getLeftLine(void);
cv::Vec4i getLeftLine(void);

/**
 * @brief Set right road marking line
 *
 * This function sets a right road marking line to the lane.
 *
 * @param line Road marking line
 */
//~ void setRightLine(RoadMarking line);

void setRightLine(cv::Vec4i line);

/**
 * @brief Get right road marking line
 *
 * This function gets the right road marking line.
 *
 * @return Road marking line
 */
//~ RoadMarking getRightLine(void);

cv::Vec4i getRightLine(void);

/**
 * @brief Push a left road marking line point
 *
 * This function pushes a left road marking line point to the lane
 *
 * @param point The point 2D coordinates
 */
void pushLeftLinePoint(cv::Point point);

/**
 * @brief Push a right road marking line point
 *
 * This function pushes a right road marking line point to the lane
 *
 * @param point The point 2D coordinates
 */
void pushRightLinePoint(cv::Point point);

/**
 * @brief Set lane
 *
 * This function sets a lane with left and right road marking lines
 *
 * @param lane The lane with left and right road marking lines
 */
void setLane(LaneData lane);

/**
 * @brief Get lane
 *
 * This function gets a lane with left and right road marking lines
 *
 * @returns The lane data
 */
LaneData getLane(void);

/**
 * @brief Set region of interest of left lane marking
 *
 * This function sets the region of interest of the left lane marking.
 *
 * @param roi Region of interest of left lane marking
 */
void setRoiLeft (cv::Rect roi);

/*

```

```

 * @brief Get region of interest of left lane marking
 *
 * This function gets the region of interest of the left lane marking.
 *
 * @return Region of interest of left lane marking
 */
cv::Rect getRoiLeft (void);

/**
 * @brief Set region of interest of right lane marking
 *
 * This function sets the region of interest of the right lane marking.
 *
 * @param roi Region of interest of right lane marking
 */
void setRoiRight (cv::Rect roi);

/**
 * @brief Get region of interest of right lane marking
 *
 * This function gets the region of interest of the right lane marking.
 *
 * @return Region of interest of right lane marking
 */
cv::Rect getRoiRight (void);

private:
RoadMarking leftLine; //!< Left lane marking
RoadMarking rightLine; //!< Right lane marking
cv::Vec4i leftLineVec4i; //!< @note Only for compatibility!
cv::Vec4i rightLineVec4i; //!< @note Only for compatibility!
cv::Rect roiLeft; //!< Region of interest of left lane marking
cv::Rect roiRight; //!< Region of interest of right lane marking
std::mutex lock; //!< Mutex lock for synchronised access
};

/** 
 * @brief A struct for the detected road data
 * @note Only for future implementation. Not implemented in this version!
 */
struct RoadData {
    std::vector<LaneData> lane;
};

/** 
 * @brief Convert road marking data to cv::Vec4i
 *
 * This function converts road marking data to 4 elements integer vector.
 * @note Road marking data must contain only two points with two
 *       two coordinates. If road marking data vector is longer only first
 *       and last point will be converted.
 *
 * @param line Road marking line
 * @return 4 element integer vector Vec4i
 */
cv::Vec4i cvtRoadMarkingToVec4i (RoadMarking line);

/** 
 * @brief Convert cv::Vec4i to road marking data
 *
 * This function converts 4 elements integer vector to road marking data.
 *
 * @param line 4 element integer vector Vec4i
 * @return Road marking line
 */
RoadMarking cvtVec4iToRoadMarking (cv::Vec4i line);

//! @} lane_detection
#endif // LANE_DATA_HPP

```

Lane_data.cpp

```
/***
 * @file lane_data.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 25.8.2017
 */

#include "lane_data.hpp"

//~ void LaneData::setLeftLine(RoadMarking line)
//~ {
//~     std::lock_guard<std::mutex> guard(lock);
//~     leftLine = line;
//~ }

void LaneData::setLeftLine(cv::Vec4i line)
{
    std::lock_guard<std::mutex> guard(lock);
    leftLineVec4i = line;
}

//~ RoadMarking LaneData::getLeftLine(void)
//~ {
//~     std::lock_guard<std::mutex> guard(lock);
//~     return leftLine;
//~ }

cv::Vec4i LaneData::getLeftLine(void)
{
    std::lock_guard<std::mutex> guard(lock);
    return leftLineVec4i;
}

//~ void LaneData::setRightLine(RoadMarking line)
//~ {
//~     std::lock_guard<std::mutex> guard(lock);
//~     rightLine = line;
//~ }

void LaneData::setRightLine(cv::Vec4i line)
{
    std::lock_guard<std::mutex> guard(lock);
    rightLineVec4i = line;
}

//~ RoadMarking LaneData::getRightLine(void)
//~ {
//~     std::lock_guard<std::mutex> guard(lock);
//~     return rightLine;
//~ }

cv::Vec4i LaneData::getRightLine(void)
{
    std::lock_guard<std::mutex> guard(lock);
    return rightLineVec4i;
}

void LaneData::pushLeftLinePoint(cv::Point point)
{
    std::lock_guard<std::mutex> guard(lock);
    leftLine.points.push_back(point);
}

void LaneData::pushRightLinePoint(cv::Point point)
{
    std::lock_guard<std::mutex> guard(lock);
    rightLine.points.push_back(point);
}

void LaneData::setRoiLeft (cv::Rect roi)
{
    std::lock_guard<std::mutex> guard(lock);
}
```

```
    roiLeft = roi;
}

cv::Rect LaneData::getRoiLeft (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return roiLeft;
}

void LaneData::setRoiRight (cv::Rect roi)
{
    std::lock_guard<std::mutex> guard(lock);
    roiRight = roi;
}

cv::Rect LaneData::getRoiRight (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return roiRight;
}

cv::Vec4i cvtRoadMarkingToVec4i (RoadMarking line)
{
    cv::Vec4i convertedLine;
    convertedLine[0] = line.points.at(0).x;
    convertedLine[1] = line.points.at(0).y;
    convertedLine[2] = line.points.at(1).x;
    convertedLine[3] = line.points.at(1).y;
    return convertedLine;
}

RoadMarking cvtVec4iToRoadMarking (cv::Vec4i line)
{
    RoadMarking convertedLine;
    convertedLine.points.push_back(cv::Point(line[0], line[1]));
    convertedLine.points.push_back(cv::Point(line[2], line[3]));
    return convertedLine;
}
```

Lane_detection.hpp

```

/***
 * @file lane_detection.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 30.6.2017
 */

/**
 * @ingroup module
 * @defgroup lane_detection Lane Detection
 * @brief A lane detection module
 */

#ifndef LANE_DETECTION_HPP
#define LANE_DETECTION_HPP

#include <iostream>
#include <fstream>
#include <atomic>
#include <opencv2/opencv.hpp>
#include "module.hpp"
#include "image_data.hpp"
#include "image_filter.hpp"
#include "lane_data.hpp"
#include "camera_image_acquisitor.hpp"

//! @addtogroup lane_detection
//! @{
//! @{

/***
 * @brief A lane detection class
 *
 * This class describes a lane detector
 */
class LaneDetector : public Module {
public:
    ~LaneDetector () = default;

    /**
     * @brief Run lane detection
     *
     * This function runs the lane detection thread.
     *
     * @param inputImage Input image captured by image acquisitor
     * @param outputImage Output image with result for showing on user interface
     * @param lane Actual detected lane
     */
    void run(ImageData& inputImage, ImageData& outputImage, LaneData& lane);

private:
    CameraCalibrationConfig camCalibConfig; //!< Camera calibration configuration
};

/***
 * @brief Get euclidian distance between two points
 *
 * This function calculates the euclidian distance between two points.
 *
 * @param pt1 First point
 * @param pt2 Second point
 */
float getEuclidDistance(cv::Point pt1, cv::Point pt2);

/***
 * @brief Get distance &rho; of a line to the origin
 *
 * This function calculates the distance &rho; of a line to the origin in
 * polar form (&rho;, &theta;).
 *
 * @param pt1 Starting point of line
 * @param pt2 Ending point of line
 * @return The distance &rho; of the line to the origin in polar form
 */

```

```

/*
float getRho (cv::Point pt1, cv::Point pt2);

/**
 * @brief Get the orthogonal angle &theta; of the slope of a line to the origin
 *
 * This function calculates the orthogonal angle &theta; of the slope
 * of a line in relation to the origin for the polar form
 * (&rho;, &theta;).
 *
 * @param pt1 Starting point of the line
 * @param pt2 Ending point of the line
 * @return The orthogonal angle &theta; of the line to the origin for the polar form
 */
float getTheta (cv::Point pt1, cv::Point pt2);

/**
 * @brief Get the middle line of a lane
 *
 * This function gets the middle line of a lane as starting and ending
 * point.
 *
 * @param lane Lane as vector of two lines with starting and ending point
 * @return Middle line as starting and ending point
 */
cv::Vec4i getLaneMid (std::vector<cv::Point> lane);

/**
 * @brief Convert lines as Vec4i to polar coordinates (&rho;, &theta;)
 *
 * This function converts a line vector containing lines as Vec4i to polar
 * coordinates (&rho;, &theta;).
 *
 * @param lines Lines to convert
 * @param linesPolar Result as (&rho;, &theta;)
 */
void cvtLinesToLinesPolar (std::vector<cv::Vec4i> lines, std::vector<cv::Vec2f>&
linesPolar);

/**
 * @brief Sort line direction
 *
 * This function sorts a vector of lines and changes their direction so
 * that all lines have the same direction.
 *
 * @param lines Lines to sort
 */
void sortLineDirections (std::vector<cv::Vec4i>& lines);

/**
 * @brief Detect lines in a gray scale image
 *
 * This function detects lines in a gray scale image with a Canny filter.
 *
 * @param grayImage Gray scale input image
 * @param lines Detected lines
 */
void detectLines (cv::Mat grayImage, std::vector<cv::Vec4i>& lines);

/**
 * @brief Draw line
 *
 * This function draws a line on to an image.
 *
 * @param image Image to draw on
 * @param l Line to draw
 * @param color Color of line to draw
 */
void drawLine (cv::Mat& image, cv::Vec4i l, cv::Scalar color);

/**
 * @brief Draw lines
 *
 */

```

```

* This function draws lines on to an image.
*
* @param image Image to draw on
* @param lines Lines to draw
* @param color Color of lines to draw
*/
void drawLines (cv::Mat& image, std::vector<cv::Vec4i> lines, cv::Scalar color);

/**
* @brief Draw an arrowed line
*
* This function draws an arrowed line on an input image with a color.
*
* @param image Image matrix
* @param line Line
* @param color Line color as RGB scalar value
*/
void drawArrowedLine (cv::Mat& image, cv::Vec4i line, cv::Scalar color);

/**
* @brief Draw an arrowed line
*
* This function draws an arrowed line on an input image with a color.
*
* @param image Image matrix
* @param lines Line vector
* @param color Line color as RGB scalar value
*/
void drawArrowedLines (cv::Mat& image, std::vector<cv::Vec4i> lines, cv::Scalar color);

/**
* @brief Draw image center line
*
* This function draws a centered line in an input image.
*
* @param image Image matrix
* @param color Color of center line
*/
void drawCenterLine (cv::Mat& image, cv::Scalar color);

/**
* @brief Filter detected lines in left and right image part
*
* This function filters left and right lines. It also filters the most
* left line in right image half and the most right line in the left
* image half.
*
* @param lines Lines to filter
* @param imageSize Image size of input image
* @param leftLines All filtered left lines
* @param rightLines All filtered right lines
* @param lane Lane composed of one left and one right line
*/
void filterLines (std::vector<cv::Vec4i> lines, cv::Size imageSize, std::vector<cv::Vec4i>&
leftLines, std::vector<cv::Vec4i>& rightLines, std::vector<cv::Vec4i>& lane);

/**
* @brief Calculate distance between two lines
*
* This function calculates the distance between two lines.
*
* @param line1 First line
* @param line2 Second line
* @return Distance between lines
*/
float distanceBetweenLines (cv::Vec4i line1, cv::Vec4i line2);

/**
* @brief Check if lines are parallel
*
* This function checks if two lines are parallel.
*
* @param lines Lines to check

```

```

 * @return True if lines are parallel, else false
 */
bool checkParallelLine (std::vector<cv::Vec4i> lines);

/**
 * @brief Check one of the detected lines is a stop road marking line
 *
 * This function checks if one of the detected lines is a stop road
 * marking line.
 *
 * @param lines Lines to check
 * @param stopLine Detected stop line
 * @return True if stop line detected, else false
 */
bool checkForStopLine (std::vector<cv::Vec4i> lines, cv::Vec4i& stopLine);

/**
 * @brief Get lane middle line
 *
 * This function gets the middle line of a lane. It calculates the half
 * of the distance between the two starting and ending points. These are
 * the starting and ending points of the middle line.
 *
 * @param lane
 * @return Middle line
 */
cv::Vec4i getLaneMid (std::vector<cv::Vec4i> lane);

/**
 * @brief Initialize a Kalman filter for line prediction
 *
 * This function initializes a Kalman filter to predict lines found in
 * an input image.
 *
 * @param kf Kalman filter
 * @param numValues Values to predict
 */
void initLinePrediction (cv::KalmanFilter& kf, int numValues);

/**
 * @brief Predict position of line
 *
 * This function predicts the line positions found in an input image.
 * It corrects the prediction by considering vector of found/measured
 * lines.
 *
 * @param lines Vector of lines describet by starting and and ending point
 * @param kf Kalman filter
 * @param numValues Values to predict
 * @param predLines Vector of predicted lines
 */
void predictLine (std::vector<cv::Vec4i> lines, cv::KalmanFilter& kf, int numValues,
std::vector<cv::Vec4i>& predLines);

/**
 * @brief Prepare image for line detection
 *
 * This function prepares an image for line detection. It applies the necessare filters for
 * line detection.
 *
 * @param image Image matrix
 * @param prepImage Detected lines
 */
void prepareImage (cv::Mat image, cv::Mat& prepImage);

/**
 * @brief Rest ROI
 *
 * This function resets the region of interest of the detected lines
 *
 * @param imageSize
 */
void resetRois (cv::Size imageSize);

```

```
/**  
 * @brief Get ROI of a line  
 *  
 * This function gets the region of interest of a line.  
 *  
 * @param line  
 * @param roi  
 * @param imageSize  
 */  
void roiOfLine (cv::Vec4i line, cv::Rect& roi, cv::Size imageSize);  
  
//! @} lane_detection  
  
#endif // LANE_DETECTION_HPP
```

Lane_detection.cpp

```


/***
 * @file lane_detection.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 30.6.2017
 */

#include "lane_detection.hpp"
#include "configuration.hpp"

void LaneDetector::run (ImageData& inputImage, ImageData& outputImage, LaneData& lane)
{
    std::cout << "THREAD: Lane detection started." << std::endl;
    running = true;

    Configurator& config = Configurator::instance();
    camCalibConfig = config.getCameraCalibrationConfig();

    // Initialize line prediction
    cv::KalmanFilter kfL(4, 4, 0);
    cv::KalmanFilter kfR(4, 4, 0);
    cv::KalmanFilter kfM(4, 4, 0);
    initLinePrediction(kfL, 4);
    initLinePrediction(kfR, 4);
    initLinePrediction(kfM, 4);
    std::vector<cv::Vec4i> leftLinesPredicted;
    std::vector<cv::Vec4i> rightLinesPredicted;
    std::vector<cv::Vec4i> lanePredicted;

    std::vector<cv::Vec4i> leftLine;
    std::vector<cv::Vec4i> rightLine;

    cv::Mat homography;
    homography = camCalibConfig.homography;
    if (homography.empty()) {
        running = false;
        error = true;
    }

    while (running && !error) {
        cv::Mat image;
        image = inputImage.read();

        if (!image.empty()) {
            cv::Mat warpedImage;
            if (!homography.empty()) {
                warpPerspective(image, warpedImage, homography, image.size(),
CV_WARP_INVERSE_MAP + CV_INTER_LINEAR);
            }
            else {
                image.copyTo(warpedImage);
            }

            cv::Mat prepImage;
            prepareImage(warpedImage, prepImage);

            // Detect lines
            std::vector<cv::Vec4i> lines;
            detectLines(prepImage, lines);

            // If no lines where found, take predicted lines
            if (lines.size() <= 0) {
                lines.insert(lines.end(), lanePredicted.begin(), lanePredicted.end());
            }

            if (lines.size() > 0) {
                // Show lines
                //~ drawArrowedLines(warpedImage, lines, cv::Scalar(0,0,255));

                // Filter lines
                std::vector<cv::Vec4i> leftLines;
                std::vector<cv::Vec4i> rightLines;


```

```

        std::vector<cv::Vec4i> actualLane;
        filterLines(lines, warpedImage.size(), leftLines, rightLines, actualLane);

        // Predict lane
        // Predict left line
        if (leftLines.size() > 0) {
            std::vector<cv::Vec4i> leftLine;
            leftLine.push_back(actualLane[0]);
            predictLine(leftLine, kfL, 4, leftLinesPredicted);
            drawLines(warpedImage, leftLinesPredicted, cv::Scalar(255, 0, 0));
        }
        // Check if exaktly one line was predicted
        //~ if (leftLinesPredicted.size() > 0) {
        if (leftLinesPredicted.size() == 1) {
            lanePredicted.push_back(leftLinesPredicted[0]);
            leftLinesPredicted.clear();
        }
        else {
            lanePredicted.push_back(cv::Vec4i(0, 0, 0, 0));
        }

        //Predict right line
        if (rightLines.size() > 0) {
            std::vector<cv::Vec4i> rightLine;
            rightLine.push_back(actualLane[1]);
            predictLine(rightLine, kfR, 4, rightLinesPredicted);
            drawLines(warpedImage, rightLinesPredicted, cv::Scalar(0, 0, 255));
        }
        // Check if exaktly one line was predicted
        //~ if (rightLinesPredicted.size() > 0) {
        if (rightLinesPredicted.size() == 1) {
            lanePredicted.push_back(rightLinesPredicted[0]);
            rightLinesPredicted.clear();
        }
        else {
            lanePredicted.push_back(cv::Vec4i(warpedImage.cols-1, 0,
warpedImage.cols-1, 0));
        }

        // Check lane has any line
        if (lanePredicted.size() >= 1) {
            //~ drawLine(warpedImage, getLaneMid(lanePredicted),
cv::Scalar(200,200,0));
            //~ lane.setLeftLine(cvtVec4iToRoadMarking(lanePredicted[0]));
            lane.setLeftLine(lanePredicted[0]);
            //~ lane.setRightLine(cvtVec4iToRoadMarking(lanePredicted[1]));
            lane.setRightLine(lanePredicted[1]);
        }
        lanePredicted.clear();
    }

    drawCenterLine(warpedImage, cv::Scalar(0, 255, 0));
    outputImage.write(warpedImage);
    outputImage.setTime(inputImage.getTime());
}
std::this_thread::sleep_for(std::chrono::milliseconds(10));
}

std::cout << "THREAD: Lane detection ended." << std::endl;
}

float getEuclidDistance(cv::Point pt1, cv::Point pt2)
{
    cv::Point diff = pt1 - pt2;
    return sqrt(pow(diff.x, 2) + pow(diff.y, 2));
    // or
    //~ return norm(pt1-pt2);
}

float getRho (cv::Point pt1, cv::Point pt2)
{
    return (pt1.y * pt2.x - pt2.y * pt1.x)/getEuclidDistance(pt1, pt2);
}

```

```

float getTheta (cv::Point pt1, cv::Point pt2)
{
    return atan2((pt2.y - pt1.y), (pt2.x - pt1.x));
}

void cvtLinesToLinesPolar (std::vector<cv::Vec4i> lines, std::vector<cv::Vec2f>&
linesPolar)
{
    linesPolar.clear();
    for (size_t i = 0; i < lines.size(); i++) {
        float theta = getTheta(cv::Point(lines[i][0], lines[i][1]), cv::Point(lines[i][2],
lines[i][3]));
        float rho = getRho(cv::Point(lines[i][0], lines[i][1]), cv::Point(lines[i][2],
lines[i][3]));

        linesPolar.push_back(cv::Vec2f(rho, theta));
    }
}

void sortLineDirections (std::vector<cv::Vec4i>& lines)
{
    for (size_t i = 0; i < lines.size(); i++) {
        cv::Point pt1 = cv::Point(lines[i][0], lines[i][1]);
        cv::Point pt2 = cv::Point(lines[i][2], lines[i][3]);

        if (pt1.y > pt2.y) {
            cv::Point ptTmp = pt1;
            pt1 = pt2;
            pt2 = ptTmp;
        }
        lines[i][0] = pt1.x;
        lines[i][1] = pt1.y;
        lines[i][2] = pt2.x;
        lines[i][3] = pt2.y;
    }
}

void detectLines (cv::Mat grayImage, std::vector<cv::Vec4i>& lines)
{
    lines.clear();

    cv::Mat imageEdges;
    Canny(grayImage, imageEdges, 10, 50);
    //~ Sobel(grayImage, imageEdges, CV_16S, 1, 0, 1);
    //~ convertScaleAbs(imageEdges, imageEdges);

    /**
     * Find lines with probabilistic Hough line function.
     * @note threshold 35 seems to be optimal
     * @note minLineLength 40 seems to be optimal
     * @note maxLineGap 10 seems to be optimal
     */
    HoughLinesP(imageEdges, lines, 1, CV_PI/180, 35, 40, 10);

    sortLineDirections(lines);
}

void drawLine (cv::Mat& image, cv::Vec4i l, cv::Scalar color)
{
    line(image, cv::Point(l[0], l[1]), cv::Point(l[2], l[3]), color, 2);
}

void drawLines (cv::Mat& image, std::vector<cv::Vec4i> lines, cv::Scalar color)
{
    for (size_t i = 0; i < lines.size(); i++) {
        cv::Vec4i l = lines[i];
        line(image, cv::Point(l[0], l[1]), cv::Point(l[2], l[3]), color, 2);
    }
}

void drawArrowedLine (cv::Mat& image, cv::Vec4i l, cv::Scalar color)
{

```

```

        arrowedLine(image, cv::Point(l[0], l[1]), cv::Point(l[2], l[3]), color, 2);
    }

void drawArrowedLines (cv::Mat& image, std::vector<cv::Vec4i> lines, cv::Scalar color)
{
    for (size_t i = 0; i < lines.size(); i++) {
        cv::Vec4i l = lines[i];
        arrowedLine(image, cv::Point(l[0], l[1]), cv::Point(l[2], l[3]), color, 2);
    }
}

void drawCenterLine (cv::Mat& image, cv::Scalar color)
{
    cv::Point pt1((image.cols/2)-1, 0);
    cv::Point pt2(pt1.x, image.rows-1);
    line(image, pt1, pt2, color, 1);
}

void filterLines (std::vector<cv::Vec4i> lines, cv::Size imageSize, std::vector<cv::Vec4i>&
leftLines, std::vector<cv::Vec4i>& rightLines, std::vector<cv::Vec4i>& lane)
{
    lane.clear();
    cv::Vec4i leftLine = cv::Vec4i(imageSize.width-1, imageSize.height-1, imageSize.width-
1, 0);
    cv::Vec4i rightLine = cv::Vec4i(0, imageSize.height-1, 0, 0);

    for (size_t i = 0; i < lines.size(); i++) {
        if (lines[i][2] <= ((imageSize.width-1)/2)) {
            leftLines.push_back(lines[i]);
            if (lines[i][2] <= leftLine[2]) {
                if (lines[i][3] >= leftLine[3]) {
                    leftLine = lines[i];
                }
            }
        }
        else {
            rightLines.push_back(lines[i]);
            if (lines[i][2] >= rightLine[2]) {
                if (lines[i][3] >= rightLine[3]) {
                    rightLine = lines[i];
                }
            }
        }
    }
    lane.push_back(leftLine);
    lane.push_back(rightLine);
}

float distanceBetweenLines (cv::Vec4i line1, cv::Vec4i line2)
{
    float m1, m2, b1, b2;

    // Check if lines parallel to x or y axis
    if ((line1[0] == line1[2]) || (line2[0] == line2[2])) {
        return 0;
    }
    else {
        m1 = (line1[3] - line1[1])/(line1[2]-line1[0]);
        m2 = (line2[3] - line2[1])/(line2[2]-line2[0]);
        b1 = (line1[2]*line1[1] - line1[0]*line1[3])/(line1[2]-line1[0]);
        b2 = (line2[2]*line2[1] - line2[0]*line2[3])/(line2[2]-line2[0]);

        if (m1 == m2) {
            return abs(b2-b1)/sqrt(pow(m1, 2)+1);
        }
        else {
            return 0;
        }
    }
}

```

```

bool checkParallelLine (std::vector<cv::Vec4i> lines)
{
    float thetaLeft = getTheta(cv::Point(lines[0][0], lines[0][1]),
cv::Point(lines[0][2], lines[0][3]));
    float thetaRight = getTheta(cv::Point(lines[1][0], lines[1][1]), cv::Point(lines[1][2],
lines[1][3]));
    //~ std::cout << "Left line theta: " << thetaLeft << ", Right line theta " <<
thetaRight << std::endl;
    if (thetaLeft == thetaRight) {
        std::cout << "Lines are parallel!" << std::endl;
        //~ std::cout << "Distance between lines is " << distanceBetweenLines(lines[0],
lines[1])*getPxPerMm() << std::endl;
        std::cout << "Distance between lines is " << distanceBetweenLines(lines[0],
lines[1]) << std::endl;
        return true;
    }
    else {
        return false;
    }
}

bool checkForStopLine (std::vector<cv::Vec4i> lines, cv::Vec4i& stopLine)
{
    for (size_t i = 0; i < lines.size(); i++) {
        float theta = getTheta(cv::Point(lines[i][0], lines[i][1]),
cv::Point(lines[i][2], lines[i][3]));
        if ((theta < (CV_PI*0.2)) || (theta > (CV_PI*0.80))) {
            stopLine = lines[i];
            std::cout << "Stop line detected!" << std::endl;
            return true;
        }
    }
    return false;
}

cv::Vec4i getLaneMid (std::vector<cv::Vec4i> lane)
{
    cv::Point ptl1 = cv::Point(lane[0][0], lane[0][1]);
    cv::Point ptl2 = cv::Point(lane[0][2], lane[0][3]);
    cv::Point ptr1 = cv::Point(lane[1][0], lane[1][1]);
    cv::Point ptr2 = cv::Point(lane[1][2], lane[1][3]);

    cv::Point ptmid1 = cv::Point(ptl1.x + (ptr1.x - ptl1.x)/2, ptl1.y + (ptr1.y -
ptl1.y)/2);
    cv::Point ptmid2 = cv::Point(ptl2.x + (ptr2.x - ptl2.x)/2, ptl2.y + (ptr2.y -
ptl2.y)/2);
    cv::Vec4i laneMid = cv::Vec4i(ptmid1.x, ptmid1.y, ptmid2.x, ptmid2.y);

    return laneMid;
}

void initLinePrediction (cv::KalmanFilter& kf, int numValues)
{
    kf.statePre = cv::Mat::zeros(numValues, 1, CV_32F);

    setIdentity(kf.transitionMatrix);
    setIdentity(kf.measurementMatrix);
    setIdentity(kf.processNoiseCov, cv::Scalar::all(500));
    setIdentity(kf.measurementNoiseCov, cv::Scalar::all(500));
    //~ setIdentity(kf.errorCovPre, cv::Scalar::all(1000));
    //~ setIdentity(kf.errorCovPost, cv::Scalar::all(500));
}

/**
 * @todo Measured lines unnecessary because measLines = lines
 */
void predictLine (std::vector<cv::Vec4i> lines, cv::KalmanFilter& kf, int numValues,
std::vector<cv::Vec4i>& predLines)
{
    for (size_t i = 0; i < lines.size(); i++) {
        cv::Mat prediction = kf.predict();
        cv::Vec4i predictedPts(prediction.at<float>(0), prediction.at<float>(1),
prediction.at<float>(2), prediction.at<float>(3));

```

```

cv::Mat measurement = cv::Mat::zeros(numValues, 1, CV_32F);
for (size_t j = 0; j < numValues; j++) {
    measurement.at<float>(j) = lines[i][j];
}

cv::Mat estimated = kf.correct(measurement);

cv::Vec4i statePts(estimated.at<float>(0), estimated.at<float>(1),
estimated.at<float>(2), estimated.at<float>(3));
predLines.clear();
predLines.push_back(statePts);
}
}

void prepareImage (cv::Mat image, cv::Mat& prepImage)
{
    autoAdjustBrightness(image);

    // Blur image
    //~ GaussianBlur(image, image, cv::Size(5, 5), 0);

    cv::Mat grayImage;
    whiteColorFilter(image, grayImage);

    cv::Mat kernel = getStructuringElement(cv::MORPH_RECT, cv::Size(5, 5), cv::Point(2,
2));
    erode(grayImage, grayImage, kernel);

    //~ cv::Mat yellowImage;
    //~ yellowColorFilter(image, yellowImage);

    // Merge images
    //~ bitwise_or(image, yellowImage, grayImage);

    // Blur image
    //~ GaussianBlur(grayImage, grayImage, cv::Size(5, 5), 0);
    GaussianBlur(grayImage, prepImage, cv::Size(5, 5), 0);

    //~ cvtColor(grayImage, image, CV_GRAY2BGR);
}

void resetRois (cv::Size imageSize)
{
    //~ setRoiLeft(cv::Rect(cv::Point(0, 0), cv::Point(imageSize.width-1, imageSize.height-
1)));
    //~ setRoiRight(cv::Rect(cv::Point(0, 0), cv::Point(imageSize.width-1,
imageSize.height-1)));
}

void roiOfLine (cv::Vec4i line, cv::Rect& roi, cv::Size imageSize)
{
    if (line[0] <= line[2]) {
        roi = cv::Rect(cv::Point(line[0]-10, line[1]-10), cv::Point(line[2]+10,
line[3]+10));
    }
    else {
        roi = cv::Rect(cv::Point(line[0]+10, line[1]-10), cv::Point(line[2]-10,
line[3]+10));
    }

    // Check if ROI in bound
    if (roi.tl().x < 0) {
        roi = cv::Rect(cv::Point(0, roi.tl().y), roi.br());
    }
    else if (roi.tl().x > (imageSize.width-1)) {
        roi = cv::Rect(cv::Point((imageSize.width-1), roi.tl().y), roi.br());
    }

    if (roi.tl().y < 0) {
        roi = cv::Rect(cv::Point(roi.tl().x, 0), roi.br());
    }
    else if (roi.tl().y > (imageSize.height-1)) {

```

```
    roi = cv::Rect(cv::Point(roi.tl().x, (imageSize.height-1)), roi.br());
}

if (roi.br().x < 0) {
    roi = cv::Rect(roi.tl(), cv::Point(0, roi.br().y));
}
else if (roi.br().x > (imageSize.width-1)) {
    roi = cv::Rect(roi.tl(), cv::Point((imageSize.width-1), roi.br().y));
}

if (roi.br().y < 0) {
    roi = cv::Rect(roi.tl(), cv::Point(roi.br().x, 0));
}
else if (roi.br().y > (imageSize.height-1)) {
    roi = cv::Rect(roi.tl(), cv::Point(roi.br().x, (imageSize.height-1)));
}
}
```

Obstacle_data.hpp

```
/***
 * @file obstacle_data.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 29.3.2018
 */

#ifndef OBSTACLE_DATA_HPP
#define OBSTACLE_DATA_HPP

#include <iostream>
#include <mutex>
#include <opencv2/opencv.hpp>

//! @addtogroup obstacle_detection
//! @{
//! @addtogroup obstacle_data

/***
 * @brief An obstacle data class
 */
class ObstacleData {
public:
    ~ObstacleData() = default;
    /**
     * @brief Set distance to obstacle
     *
     * This function sets the distance between the capturing sensor and
     * a detected obstacle.
     *
     * @param d distance in mm
     */
    void setDistance(double d);

    /**
     * @brief Get distance to obstacle
     *
     * This function gets the distance between the capturing sensor and
     * a detected obstacle.
     *
     * @return Distance in mm
     */
    double getDistance(void);

    /**
     * @brief Set velocity of obstacle
     *
     * This function sets the velocity of a detected obstacle relative
     * to the capturing sensor.
     *
     * @param v Velocity in m/s
     */
    void setVelocity(double v);

    /**
     * @brief Get velocity of obstacle
     *
     * This function gets the velocity of a detected obstacle relative
     * to the capturing sensor.
     *
     * @return Velocity in m/s
     */
    double getVelocity(void);

    /**
     * @brief Set ROI of obstacle
     *
     * This function sets the region of interest in an image where a
     * detected obstacle is.
     *
     * @param r Rectangle in pixel
     */
    void setRoi(cv::Rect r);
}
```

```
/**  
 * @brief Get ROI of obstacle  
 *  
 * This function gets the region of interest in an image where a  
 * detected obstacle is.  
 *  
 * @return ROI rectangle in pixel  
 */  
cv::Rect getRoi(void);  
  
private:  
    double distance {-1}; //!< Distance from sensor to the object  
    double velocity {-1}; //!< Objekt velocity relative to the sensor  
    cv::Rect roi; //!< Region of interest where objekt is in a given image  
    std::mutex lock; //!< Mutex lock for synchronized access  
};  
  
//! @} obstacle_detection  
#endif // OBSTACLE_DATA_HPP
```

Obstacle_data.cpp

```
/**  
 * @file obstacle_data.cpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 29.3.2018  
 */  
  
#include "obstacle_data.hpp"  
  
void ObstacleData::setDistance (double d)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    distance = d;  
}  
  
double ObstacleData::getDistance (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return distance;  
}  
  
void ObstacleData::setVelocity (double v)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    velocity = v;  
}  
  
double ObstacleData::getVelocity (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return velocity;  
}  
  
void ObstacleData::setRoi (cv::Rect r)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    roi = r;  
}  
  
cv::Rect ObstacleData::getRoi (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return roi;  
}
```

Obstacle_detection.hpp

```
/***
 * @file obstacle_detection.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 4.12.2017
 */

/***
 * @ingroup module
 * @defgroup obstacle_detection Obstacle Detection
 * @brief A obstacle detection module
 */

#ifndef OBSTACLE_DETECTION_HPP
#define OBSTACLE_DETECTION_HPP

#include <iostream>
#include <atomic>
#include <thread>
#include <wiringPi.h>
#include <opencv2/opencv.hpp>
#include "module.hpp"
#include "libSonar.h"
#include "obstacle_data.hpp"

//! @addtogroup obstacle_detection
//! @{
class ObstacleDetectionConfig {
public:
    ~ObstacleDetectionConfig() = default;

    bool load (cv::FileStorage fs);
    void save (cv::FileStorage fs);

public:
    bool active {false};
};

/***
 * @brief A obstacle detector class
 */
class ObstacleDetector : public Module {
public:
    ~ObstacleDetector() = default;

    /**
     * @brief Run obstacle detection
     *
     * This function runs the obstacle detection thread.
     *
     * @param obstacleData Detected obstacle data
     */
    void run (ObstacleData& obstacleData);

private:
    ObstacleDetectionConfig obstacleDetConfig;
};

//! @} obstacle_detection
#endif // OBSTACLE_DETECTION_HPP
```

Obstacle_detection.cpp

```
/*
 * @file obstacle_detection.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 4.12.2017
 */

#include "obstacle_detection.hpp"
#include "configuration.hpp"

bool ObstacleDetectionConfig::load (cv::FileStorage fs)
{
    if (!fs.isOpened()) {
        fs.release();
        std::cerr << "ERROR: File storage not opened" << std::endl;
        return false;
    }
    else {
        fs["obstacleDetectionActive"] >> active;
    }
    fs.release();
    return true;
}

void ObstacleDetectionConfig::save (cv::FileStorage fs)
{
    if (!fs.isOpened()) {
        fs.release();
        std::cerr << "ERROR: File storage not opened" << std::endl;
    }
    else {
        fs.writeComment("obstacle detection config");
        fs << "obstacleDetectionActive" << active;
    }
    fs.release();
}

void ObstacleDetector::run (ObstacleData& obstacleData)
{
    std::cout << "THREAD: Obstacle detection started." << std::endl;
    running = true;

    Configurator& config = Configurator::instance();
    obstacleDetConfig = config.getObstacleDetectionConfig();

    int triggerPin = 3; //!<< WiringPi lib pin number
    int echoPin = 2; //!<< Wiring Pi lib pin number

    if (!obstacleDetConfig.active) {
        running = false;
    }
    if (wiringPiSetup() == -1) {
        std::cerr << "ERROR: Couldn't init wiringPi library!" << std::endl;
        running = false;
        error = true;
    }

    Sonar ultrasonic;
    ultrasonic.init(triggerPin, echoPin);

    while (running && !error) {
        double distance = ultrasonic.distance(500);
        //~ std::cout << "INFO: Obstacle detected at " << distance << " cm." << std::endl;
        obstacleData.setDistance(distance);
        std::this_thread::sleep_for(std::chrono::milliseconds(500));
    }

    std::cout << "THREAD: Obstacle detection ended." << std::endl;
}
```

Traffic_sign_data.hpp

```

/***
 * @file traffic_sign_data.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 29.3.2018
 */

#ifndef TRAFFIC_SIGN_DATA_HPP
#define TRAFFIC_SIGN_DATA_HPP

#include <iostream>
#include <mutex>
#include <opencv2/opencv.hpp>

//! @addtogroup traffic_sign_detection
//! @{
enum TrafficSignType {
    STOP_SIGN
};

/***
 * @brief An traffic sign data class
 */
class TrafficSignData {
public:
    ~TrafficSignData() = default;

    /**
     * @brief Set distance to obstacle
     *
     * This function sets the distance between the capturing sensor and
     * a detected obstacle.
     *
     * @param d Distance in mm
     */
    void setDistance (double d);

    /**
     * @brief Get distance to obstacle
     *
     * This function gets the distance between the capturing sensor and
     * a detected obstacle.
     *
     * @return Distance in mm
     */
    double getDistance (void);

    /**
     * @brief Set velocity of obstacle
     *
     * This function sets the velocity of a detected obstacle relative
     * to the capturing sensor.
     *
     * @param v Velocity in m/s
     */
    void setVelocity (double v);

    /**
     * @brief Get velocity of obstacle
     *
     * This function gets the velocity of a detected obstacle relative
     * to the capturing sensor.
     *
     * @return Velocity in m/s
     */
    double getVelocity (void);

    /**
     * @brief Set ROI of obstacle
     *
     * This function sets the region of interest in an image where a

```

```
* detected obstacle is.  
*  
* @param r Rectangle in pixel  
*/  
void setRoi (cv::Rect r);  
  
/**  
 * @brief Get ROI of obstacle  
 *  
 * This function gets the region of interest in an image where a  
 * detected obstacle is.  
 *  
 * @return ROI rectangle in pixel  
 */  
cv::Rect getRoi (void);  
  
private:  
TrafficSignType type; //!< Traffic sign type  
double distance {-1}; //!< Distance from sensor to the object  
double velocity {-1}; //!< Objekt velocity relative to the sensor  
cv::Rect roi; //!< Region of interest where objekt is in a given image  
bool found {false}; //!< Traffic signs found flag  
std::mutex lock; //!< Mutex lock for synchronized access  
};  
  
/**  
 * @brief Get center point of detected traffic sign  
 *  
 * This function gets the mid point of a detected traffic sign.  
 *  
 * @param tl Top left corner point of traffic sign  
 * @param br Bottom right corner point of traffic sign  
 * @return Center point of traffic sign  
 */  
cv::Point getSignCenter (cv::Point tl, cv::Point br);  
cv::Point getSignCenter (cv::Rect roi);  
//! @} traffic_sign_detection  
#endif // TRAFFIC_SIGN_DATA_HPP
```

Traffic_sign_data.cpp

```
/**  
 * @file traffic_sign_data.cpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 29.3.2018  
 */  
  
#include "traffic_sign_data.hpp"  
  
void TrafficSignData::setDistance (double d)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    distance = d;  
}  
  
double TrafficSignData::getDistance (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return distance;  
}  
  
void TrafficSignData::setVelocity (double v)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    velocity = v;  
}  
  
double TrafficSignData::getVelocity (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return velocity;  
}  
  
void TrafficSignData::setRoi (cv::Rect r)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    roi = r;  
}  
  
cv::Rect TrafficSignData::getRoi (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return roi;  
}  
  
cv::Point getSignCenter (cv::Point tl, cv::Point br)  
{  
    return cv::Point(tl.x + (br.x - tl.x)/2, tl.y + (br.y - tl.y)/2);  
}  
  
cv::Point getSignCenter (cv::Rect roi)  
{  
    return cv::Point(roi.tl().x + roi.width/2, roi.tl().y + roi.height/2);  
}
```

Traffic_sign_detection.hpp

```
/***
 * @file traffic_sign_detection.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 30.6.2017
 */

/***
 * @ingroup module
 * @defgroup traffic_sign_detection Traffic Sign Detection
 * @brief A traffic sign detection module
 */

#ifndef TRAFFIC_SIGN_DETECTION_HPP
#define TRAFFIC_SIGN_DETECTION_HPP

#include <iostream>
#include <atomic>
#include <thread>
#include <opencv2/opencv.hpp>
#include "module.hpp"
#include "traffic_sign_data.hpp"
#include "image_data.hpp"
#include "camera_image_acquisitor.hpp"

//! @addtogroup traffic_sign_detection
//! @{
#define STOP_SIGN_SIZE cv::Size(30,30)
#define STOP_SIGN_SAFETY_DISTANCE 20.

class TrafficSignDetectionConfig {
public:
    ~TrafficSignDetectionConfig() = default;

    bool load (cv::FileStorage fs);
    void save (cv::FileStorage fs);

public:
    bool active {false};
};

/***
 * @brief A traffic sign detector class
 */
class TrafficSignDetector : public Module {
public:
    ~TrafficSignDetector() = default;

    /**
     * @brief Start traffic sign detection
     *
     * This function starts the traffic sign detection thread. This thread
     * detects traffic signs with Haar cascade classification.
     * It uses a pretrained model with is stored in "input/*.xml".
     *
     * @param inputImageData
     * @param outputImageData
     * @param trafficSignData
     */
    void run (ImageData& inputImageData, ImageData& outputImageData, TrafficSignData& trafficSignData);

private:
    CameraCalibrationConfig camCalibConfig;
    TrafficSignDetectionConfig trafficSignDetConfig;
};

//! @} traffic_sign_detection
#endif // TRAFFIC_SIGN_DETECTION_HPP
```

Traffic_sign_detection.cpp

```


/***
 * @file traffic_sign_detection.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 30.6.2017
 */

#include "traffic_sign_detection.hpp"
#include "configuration.hpp"

bool TrafficSignDetectionConfig::load (cv::FileStorage fs)
{
    if (!fs.isOpened()) {
        fs.release();
        std::cerr << "ERROR: File storage not opened" << std::endl;
        return false;
    }
    else {
        fs["trafficSignDetectionActive"] >> active;
    }
    fs.release();
    return true;
}

void TrafficSignDetectionConfig::save (cv::FileStorage fs)
{
    if (!fs.isOpened()) {
        fs.release();
        std::cerr << "ERROR: File storage not opened" << std::endl;
    }
    else {
        fs.writeComment("traffic sign detection config");
        fs << "trafficSignDetectionActive" << active;
    }
    fs.release();
}

void TrafficSignDetector::run (ImageData& inputImageData, ImageData& outputImageData,
TrafficSignData& trafficSignData)
{
    std::cout << "THREAD: Traffic sign detection started." << std::endl;
    running = true;

    Configurator& config = Configurator::instance();
    camCalibConfig = config.getCameraCalibrationConfig();
    trafficSignDetConfig = config.getTrafficSignDetectionConfig();

    cv::CascadeClassifier stopSignCascade;

    if (!trafficSignDetConfig.active) {
        running = false;
    }
    if (!stopSignCascade.load("../input/stopsign_classifier.xml")) {
        std::cerr << "ERROR: Couldn't load classifier data!" << std::endl;
        running = false;
        error = true;
    }

    while (running && !error) {
        cv::Mat inputImage;
        inputImage = inputImageData.read();

        std::vector<cv::Rect> stopSigns;
        double distance = (-1);

        if (!inputImage.empty()) {
            cv::Mat grayImage;
            cvtColor(inputImage, grayImage, CV_BGR2GRAY);

            stopSignCascade.detectMultiScale(grayImage, stopSigns, 1.1, 5, 0 |
cv::CASCADE_SCALE_IMAGE, cv::Size(10, 10));
        }
    }
}


```

```
if (stopSigns.size() > 0) {
    for (size_t i = 0; i < stopSigns.size(); i++) {
        rectangle(inputImage, stopSigns.at(i), cv::Scalar(0, 255, 0), 2);
        cv::Point signCenter;
        signCenter = getSignCenter(stopSigns.at(i));
        std::cout << "Traffic sign detection: Stop sign detected at [" <<
signCenter.x << ", " << signCenter.y << "]" << std::endl;

        trafficSignData.setRoi(stopSigns.at(0));
        trafficSignData.setDistance(1);
    }
} else {
    trafficSignData.setDistance(-1);
}
outputImageData.write(inputImage);
outputImageData.setTime(inputImageData.getTime());
}
std::this_thread::sleep_for(std::chrono::milliseconds(10));
}

std::cout << "THREAD: Traffic sign detection ended." << std::endl;
}
```

Image_data.hpp

```

/***
 * @file image_data.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 30.6.2017
 */

#ifndef IMAGE_DATA_HPP
#define IMAGE_DATA_HPP

#include <iostream>
#include <chrono>
#include <opencv2/opencv.hpp>
#include <mutex>

//! @addtogroup image_acquisition
//! @{

/**
 * @brief A image data class
 *
 * This class describes the image data which holds the image data as
 * an OpenCV Matrix and a mutex lock for synchronised accses.
 */
class ImageData {
public:
    ~ImageData() = default;

    /**
     * @brief Write image data
     *
     * This function writes the image data and the aquisition time.
     *
     * @param image
     */
    void write (cv::Mat image);

    /**
     * @brief Read image data
     *
     * This function reads the input image data.
     *
     * @return Image matrix and time
     */
    cv::Mat read (void);

    /**
     * @brief Set frame aquisition time
     *
     * This function sets the frame aquisition time.
     *
     * @param time Aquisition time
     */
    void setTime(std::chrono::high_resolution_clock::time_point time);

    /**
     * @brief Get frame aquisition time
     *
     * This function gets the frame aquisition time.
     *
     * @return Aquisition time
     */
    std::chrono::high_resolution_clock::time_point getTime();

private:
    cv::Mat data; //!< Image data as matrix
    std::chrono::high_resolution_clock::time_point timeStamp; //!< Time of capture
    std::mutex lock; //!< Mutex lock for synchronized access
};

//! @} image_acquisition
#endif // IMAGE_DATA_HPP

```

Image_data.cpp

```
/**  
 * @file image_data.cpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 30.6.2017  
 */  
  
#include "image_data.hpp"  
  
void ImageData::write (cv::Mat image)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    data = image;  
    //~ timeStamp = std::chrono::high_resolution_clock::now();  
}  
  
cv::Mat ImageData::read ()  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return data;  
}  
  
void ImageData::setTime(std::chrono::high_resolution_clock::time_point time)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    timeStamp = time;  
}  
  
std::chrono::high_resolution_clock::time_point ImageData::getTime()  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return timeStamp;  
}
```

Image_acquisitor.hpp

```

/***
 * @file      image_acquisitor.hpp
 * @author    Sergiu-Petru Tabacariu
 * @date      28.3.2018
 */

/***
 * @ingroup module
 * @defgroup image_acquisition Image Acquisition
 * @brief A module for image acquisition.
 */

#ifndef IMAGE_ACQUISITOR_HPP
#define IMAGE_ACQUISITOR_HPP

#include <atomic>
#include <opencv2/opencv.hpp>
#include "module.hpp"
#include "image_data.hpp"

//! @addtogroup image_acquisition
//! @{

/***
 * @brief A image acquisition class
 *
 * This class defines an image acquistor interface.
 */
class ImageAcquisitor : public Module {
public:
    ~ImageAcquisitor() = default;

    /**
     * @brief Read acquisited image
     *
     * This function reads the last acquisited image.
     *
     * @return Acquisited image
     */
    virtual cv::Mat read (void) = 0;

    /**
     * @brief Write image to image acquisitor
     *
     * This function writes an image to the image acquisitor.
     *
     * @param image Image matrix
     */
    virtual void write (cv::Mat image) = 0;

    /**
     * @brief Run image acquisition
     *
     * This function runs an image acquisition thread
     *
     * @param image Acquisited image
     */
    virtual void run (ImageData& image) = 0;

protected:
    ImageData image; //!< Acquisited image
};

//! @} image_acquisition
#endif // IMAGE_ACQUISITOR_HPP

```

Camera_image_acquisitor.hpp

```

/***
 * @file camera_image_acquisitor.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 30.6.2017
 */
/***
 * @ingroup image_acquisition
 * @defgroup camera_image_acquisiton Camera Image Acquisitor
 * @brief A module to capture a camera image
 */

#ifndef CAMERA_CAPTURE_HPP
#define CAMERA_CAPTURE_HPP

#include <iostream>
#include <opencv2/opencv.hpp>
#include "image_acquisitor.hpp"
#include "image_data.hpp"
#include "user_interface_state.hpp"

//! @addtogroup camera_image_acquisiton
//! @{
//! @{

/***
 * @brief Camera configuration data structure
 */
class CameraConfig {
public:
    ~CameraConfig() = default;

    bool load (cv::FileStorage fs);
    void save (cv::FileStorage fs);

public:
    int id {0}; //!< Camera ID initialized with 0
    cv::Size imageSize {640, 360}; //!< Camera image size
    double fps {15.}; //!< Frames per second captured by the camera
    double exposure {0.}; //!< Camera exposure value
};

/***
 * @brief Camera calibration configuration data structure
 */
class CameraCalibrationConfig {
public:
    ~CameraCalibrationConfig() = default;

    bool load (cv::FileStorage fs);
    void save (cv::FileStorage fs);

public:
    cv::Size imageSize {640, 360}; //!< Calibration image size
    std::string pattern {"CHESSBOARD"}; //!< Calibration pattern
    cv::Size patternSize {7, 5}; //!< Calibration pattern size
    double patternMm {30.}; //! Size of one pattern element in mm
    int numSamples {50}; //! Number of samples to use for calibration
    cv::Mat cameraMatrix; //! Camera matrix containing the focal length and principal image
    point
    cv::Mat distCoeffs; //! Distortion coefficients for distortion correction
    bool intrCalibDone {false}; //! Calibration done flag
    std::chrono::high_resolution_clock::time_point timeOfIntrCalib; //!< Intrinsics
    calibration time stamp
    cv::Mat homography; //!< Homography for perspective transform
    bool extrCalibDone {false}; //! Calibration done flag
    std::chrono::high_resolution_clock::time_point timeOfExtrCalib; //!< Extrinsics
    calibration time stamp
    cv::Mat transform; //!< Transformation matrix for image position
    double mmPerPixel {0.}; //!< Average mm per pixel
};

//!
*/

```

```

* @brief A camera image acquisition class
*
* This class describes a camera image acquisitor
*/
class CameraImageAcquisitor : public ImageAcquisitor {
public:
    ~CameraImageAcquisitor() = default;

    /**
     * @brief Write image to camera image acquisitor
     *
     * This function writes an image to the camera image acquisitor
     *
     * @param image Image matrix
     */
    void write (cv::Mat image) override;

    /**
     * @brief Read captured camera frame
     *
     * This function reads the actual image frame captured by the camera
     * image acquisitor
     *
     * @return Image frame
     */
    cv::Mat read (void) override;

    /**
     * @brief Run camera image acquisition
     *
     * This function runs a camera image acquisition thread.
     *
     * @param imageData A image data reference where to store captured image
     */
    void run (ImageData& imageData) override;

    /**
     * @brief Run intrinsic camera calibration
     *
     * @param inputImage Input image data
     * @param outputImage Output image data
     * @param uiState User interface state
     */
    void runIntrinsicCalibration (ImageData& inputImage, ImageData& outputImage,
UserInterfaceState& uiState);

    /**
     * @brief Run extrinsic camera calibration
     *
     * This function runs the extrinsic camera calibration in a thread.
     *
     * @param inputImage Input image data
     * @param outputImage Output image data
     * @param uiState User interface state
     */
    void runExtrinsicCalibration (ImageData& inputImage, ImageData& outputImage,
UserInterfaceState& uiState);

    /**
     * @brief Run image adjustment in frame
     *
     * This function runs the image adjustment in a
     * frame.
     *
     * @param inputImage Input image data
     * @param outputImage Output image data
     * @param uiState User interface state
     */
    void runImageAdjustment (ImageData& inputImage, ImageData& outputImage,
UserInterfaceState& uiState);

    /**
     * @brief Run chess board show

```

```

/*
 * This function runs a chess board showing thread.
 *
 * @param inputImage Input image data
 * @param outputImage Output image data
 */
void runChessBoardShow (ImageData& inputImage, ImageData& outputImage);

private:
    cv::Mat capturedImage; //!< Captured image
    CameraConfig camConfig; //!< Camera configuration data
    CameraCalibrationConfig camCalibConfig; //!< Calibration configuration data
};

/** 
 * @brief Calculate camera exposure for a given exposure step
 *
 * This function calculates the camera exposure for a given exposure step.
 * A step \f$ i\f$ is converted to \f$ 2^i\f$.
 *
 * @param exposureStep
 * @return Calculated camera exposure
 */
double calcExposure(int exposureStep);

/** 
 * @brief A function calibrate intrinsic camera parameters
 *
 * This function calibrates intrinsic camera parameters by searching for
 * chessboard pattern. This parameters are stored in a matrix \f$ A\f$ and
 * the distortion coefficients vector \f$ V\f$.
 *
 * @param image Image captured by the camera
 * @param cameraMatrix A 3x3 matrix \f$ A = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \f$
 * @param distCoeffs Distortion coefficients vector \f$ V = (k_1, k_2, p_1, p_2, k_3, k_4, k_5, k_6, s_1, s_2, s_3, s_4, \tau_x, \tau_y) \f$
 * @param imagePoints Detected calibration pattern corners
 * @param patternSize Calibration pattern dimensions
 * @param patternMm Calibration pattern size in mm
 */
bool calibrateIntrinsics (cv::Mat& image, cv::Mat& cameraMatrix, cv::Mat& distCoeffs,
std::vector<std::vector<cv::Point2f> > imagePoints, cv::Size patternSize, double
patternMm);

/** 
 * @brief A function calibrate extrinsic camera parameters
 *
 * This function calibrates extrinsic camera parameters by searching for a chessboard
pattern
 *
 * @param image Image captured by the camera
 * @param homography Perspective transform homography matrix
 * @param patternSize Calibration pattern dimensions
 * @param patternMm Calibration pattern size in mm
 */
void calibrateExtrinsics (cv::Mat& image, cv::Mat& homography, cv::Size patternSize, double
patternMm);

/** 
 * @brief A function to calculate how many pixels are in 1 mm.
 *
 * This function calculates how many pixels are in 1 millimeter.
 * It finds the corners of a captured chessboard image and calculates the
average distance between each chessboard field.
 *
 * @param image Input and output image
 * @param patternSize Calibration pattern dimensions
 * @param patternMm Calibration pattern size in mm
 */
float calcPixelPerMm (cv::Mat image, cv::Size patternSize, float patternMm);

*/

```

```
* @brief A function to draw chessboard on an image.  
*  
* This function draws a chessboard on an image.  
*  
* @param image Input and output image  
* @param patternSize Calibration pattern dimensions  
*/  
void drawChessBoard (cv::Mat& image, cv::Size patternSize);  
  
/**  
* @brief A function to adjust image position in frame.  
*  
* This function adjusts an image position inside an image frame.  
*  
* @param image Input image  
* @param adjustedImage Adjusted output image  
* @param key User input key  
* @param homography Perspective transform homography matrix  
*/  
void adjustImagePosition (cv::Mat image, cv::Mat& adjustedImage, char key, cv::Mat&  
homography);  
  
//! @} camera_image_acquisiton  
  
#endif // CAMERA_CAPTURE_HPP
```

Camera_image_acquisitor.cpp

```


/***
 * @file camera_image_acquisitor.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 30.6.2017
 */

#include "camera_image_acquisitor.hpp"
#include <thread>
#include "configuration.hpp"

cv::Mat CameraImageAcquisitor::read ()
{
    return capturedImage;
}

void CameraImageAcquisitor::write (cv::Mat image)
{
    capturedImage = image;
}

void CameraImageAcquisitor::run (ImageData& imageData)
{
    std::cout << "THREAD: Camera capture started." << std::endl;
    running = true;
    error = false;

    Configurator& config = Configurator::instance();
    camConfig = config.getCameraConfig();
    camCalibConfig = config.getCameraCalibrationConfig();

    cv::Mat cameraMatrix, distCoeffs;
    cameraMatrix = camCalibConfig.cameraMatrix;
    distCoeffs = camCalibConfig.distCoeffs;

    // Initialize camera
    cv::VideoCapture camera(camConfig.id);
    camera.set(CV_CAP_PROP_FRAME_WIDTH, camConfig.imageSize.width);
    camera.set(CV_CAP_PROP_FRAME_HEIGHT, camConfig.imageSize.height);
    camera.set(CV_CAP_PROP_FPS, camConfig.fps);
    //~ camera.set(CV_CAP_PROP_EXPOSURE, calcExposure(12));
    //~ camera.set(CV_CAP_PROP_EXPOSURE, calcExposure(7));

    if (!camera.isOpened()) {
        std::cerr << "ERROR: Could not open camera!" << std::endl;
        running = false;
        error = true;
    }

    // FPS measurement
    std::chrono::high_resolution_clock::time_point frameTimerStart, frameTimerEnd;
    long frameCnt = 0;

    // Capture image
    while (running && !error) {
        camera >> capturedImage;

        if (frameCnt == 0) {
            frameTimerStart = std::chrono::high_resolution_clock::now();
        }

        if (capturedImage.empty()) {
            std::cerr << "ERROR: Couldn't acquire image data!" << std::endl;
        }
        else {
            // Undistort captured image
            if (!cameraMatrix.empty() && !distCoeffs.empty()) {
                cv::undistort(capturedImage, capturedImage, cameraMatrix, distCoeffs);
            }
        }

        // FPS measurement
        frameCnt++;
    }
}


```

```

        frameTimerEnd = std::chrono::high_resolution_clock::now();

        if (std::chrono::duration_cast<std::chrono::seconds>(frameTimerEnd -
frameTimerStart).count() >= 3) {
            std::cout << "INFO: Captured FPS: " << frameCnt/3 << std::endl;
            frameCnt = 0;
        }
        imageData.write(capturedImage);
        imageData.setTime(std::chrono::high_resolution_clock::now());
    }

}

std::cout << "THREAD: Camera capture ended." << std::endl;
}

void CameraImageAcquisitor::runIntrinsicCalibration (ImageData& inputImage, ImageData&
outputImage, UserInterfaceState& uiState)
{
    std::cout << "THREAD: Intrinsic calibration started." << std::endl;
    running = true;

    Configurator& config = Configurator::instance();
    camCalibConfig = config.getCameraCalibrationConfig();

    bool saveConfigFlag = false;

    cv::Mat image, undistorted, cameraMatrix, distCoeffs;
    std::vector<std::vector<cv::Point2f> > imagePoints;
    int sampleCnt = 0;

    while (running) {
        image = inputImage.read();

        if (!image.empty() && (sampleCnt <= camCalibConfig.numSamples)) {
            if (calibrateIntrinsics(image, cameraMatrix, distCoeffs, imagePoints,
camCalibConfig.patternSize, camCalibConfig.patternMm)) {
                sampleCnt++;
                std::cout << "INFO: Calibration sample counter: " << sampleCnt <<
std::endl;
            }
            std::this_thread::sleep_for(std::chrono::milliseconds(250));
        }

        if (!image.empty() && !cameraMatrix.empty() && !distCoeffs.empty()) {
            cv::undistort(image, undistorted, cameraMatrix, distCoeffs);
            line(undistorted, cv::Point(undistorted.cols/2, 0),
cv::Point(undistorted.cols/2, undistorted.rows), cv::Scalar(0, 0, 255), 1);
            line(undistorted, cv::Point(0, undistorted.rows/2), cv::Point(undistorted.cols,
undistorted.rows/2), cv::Scalar(0, 0, 255), 1);
            outputImage.write(undistorted);
            outputImage.setTime(inputImage.getTime());
        }
        else {
            line(image, cv::Point(image.cols/2, 0), cv::Point(image.cols/2, image.rows),
cv::Scalar(0, 0, 255), 1);
            line(image, cv::Point(0, image.rows/2), cv::Point(image.cols, image.rows/2),
cv::Scalar(0, 0, 255), 1);
            outputImage.write(image);
            outputImage.setTime(inputImage.getTime());
        }

        // Check user input
        switch(uiState.getKey()) {
            case 'S': saveConfigFlag = true; break;
            case 'R': sampleCnt = 0; cameraMatrix.release(); distCoeffs.release(); break;
        //!< Reset counter, restart calibration
        }
    }

    if ((sampleCnt == camCalibConfig.numSamples) && (!cameraMatrix.empty() &&
!distCoeffs.empty())) {
        camCalibConfig.cameraMatrix = cameraMatrix;
}
}

```

```

camCalibConfig.distCoeffs = distCoeffs;
camCalibConfig.timeOfIntrCalib = std::chrono::high_resolution_clock::now();
camCalibConfig.intrCalibDone = true;

if (saveConfigFlag) {
    config.setCameraCalibrationConfig(camCalibConfig);
    config.save();
    std::cout << "THREAD: Intrinsic camera calibration saved!" << std::endl;
}
else {
    camCalibConfig.intrCalibDone = false;
    std::cerr << "WARNING: Intrinsic camera calibration not successful!" << std::endl;
}

std::cout << "THREAD: Intrinsics calabarion ended." << std::endl;
}

void CameraImageAcquisitor::runExtrinsicCalibration (ImageData& inputImage, ImageData&
outputImage, UserInterfaceState& uiState)
{
    std::cout << "THREAD: Extrinsics calabarion started." << std::endl;
    running = true;

    Configurator& config = Configurator::instance();
    camCalibConfig = config.getCameraCalibrationConfig();

    bool saveConfigFlag = false;

    cv::Mat image, warpedImage, homography;

    while (running) {
        image = inputImage.read();

        if (!image.empty()) {
            calibrateExtrinsics(image, homography, camCalibConfig.patternSize,
camCalibConfig.patternMm);
        }

        if (!image.empty() && !homography.empty()) {
            warpPerspective(image, warpedImage, homography, image.size(),
CV_WARP_INVERSE_MAP + CV_INTER_LINEAR);
            line(warpedImage, cv::Point(warpedImage.cols/2, 0),
cv::Point(warpedImage.cols/2, warpedImage.rows), cv::Scalar(0, 0, 255), 1);
            line(warpedImage, cv::Point(0, warpedImage.rows/2), cv::Point(warpedImage.cols,
warpedImage.rows/2), cv::Scalar(0, 0, 255), 1);
            outputImage.write(warpedImage);
            outputImage.setTime(inputImage.getTime());
        }
        else {
            line(image, cv::Point(image.cols/2, 0), cv::Point(image.cols/2, image.rows),
cv::Scalar(0, 0, 255), 1);
            line(image, cv::Point(0, image.rows/2), cv::Point(image.cols, image.rows/2),
cv::Scalar(0, 0, 255), 1);
            outputImage.write(image);
            outputImage.setTime(inputImage.getTime());
        }

        switch (uiState.getKey()) {
            case 'S': saveConfigFlag = true; break;
            case 'R': homography.release(); break;
        }
    }

    if (!homography.empty()) {
        camCalibConfig.homography = homography;
        camCalibConfig.mmPerPixel = calcPixelPerMm(warpedImage, camCalibConfig.patternSize,
camCalibConfig.patternMm);
        camCalibConfig.timeOfExtrCalib = std::chrono::high_resolution_clock::now();
        camCalibConfig.extrCalibDone = true;
        std::cout << "INFO: Avg px per mm: " << camCalibConfig.mmPerPixel << std::endl;
        if (saveConfigFlag) {
            config.setCameraCalibrationConfig(camCalibConfig);
        }
    }
}

```

```

        config.save();
        std::cout << "THREAD: Intrinsic camera calibration saved!" << std::endl;
    }
} else {
    camCalibConfig.extrCalibDone = false;
    std::cerr << "WARNING: Extrinsic camera calibration not successful!" << std::endl;
}

std::cout << "THREAD: Extrinsics calibarion ended." << std::endl;
}

void CameraImageAcquisitor::runImageAdjustment (ImageData& inputImage, ImageData&
outputImage, UserInterfaceState& uiState)
{
    std::cout << "THREAD: Image position configuration started." << std::endl;
    running = true;

    Configurator& config = Configurator::instance();
    camCalibConfig = config.getCameraCalibrationConfig();

    cv::Mat image, adjustedImage;

    while (running) {
        image = inputImage.read();

        if (!image.empty() && !camCalibConfig.homography.empty()) {
            char key = uiState.getKey();
            adjustImagePosition(image, adjustedImage, key, camCalibConfig.homography);
            outputImage.write(adjustedImage);
            outputImage.setTime(inputImage.getTime());
        }
    }

    std::cout << "THREAD: Image position configuration ended." << std::endl;
}

void CameraImageAcquisitor::runChessBoardShow (ImageData& inputImage, ImageData&
outputImage)
{
    std::cout << "THREAD: Show chessboard started." << std::endl;
    running = true;

    Configurator& config = Configurator::instance();
    camCalibConfig = config.getCameraCalibrationConfig();

    cv::Mat image;

    while (running) {
        image = inputImage.read();

        if (!image.empty()) {
            drawChessBoard(image, camCalibConfig.patternSize);
            outputImage.write(image);
            outputImage.setTime(inputImage.getTime());
        }
    }

    std::cout << "THREAD: Show chessboard ended." << std::endl;
}

double calcExposure (int value)
{
    const int minValue = 5; //!< 500 us or 0.0005 s
    const int maxValue = 20000; //!< 20000 us or 2 s
    const double range = maxValue - minValue; //!< Exposure range in 100 us

    return ((maxValue / pow(2.0, value)) - minValue) / range;
}

void calcBoardCornerPosition (cv::Size patternSize, float patternMm,
std::vector<cv::Point3f>& corners)
{

```

```

corners.clear();
// Calculate corner position of chessboard squares
for (auto i = 0; i < patternSize.height; i++) {
    for (auto j = 0; j < patternSize.width; j++) {
        corners.push_back(cv::Point3f(j*patternMm, i*patternMm, 0));
    }
}

bool calibrateIntrinsics (cv::Mat& image, cv::Mat& cameraMatrix, cv::Mat& distCoeffs,
std::vector<std::vector<cv::Point2f>> imagePoints, cv::Size patternSize, double patternMm)
{
    double aspRatio = 1.0;
    cv::Mat grayImage;
    cvtColor(image, grayImage, CV_BGR2GRAY);

    std::vector<cv::Point2f> corners;

    bool found = cv::findChessboardCorners(image, patternSize, corners,
CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS);

    if (found) {
        cornerSubPix(grayImage, corners, cv::Size(11,11), cv::Size(-1,-1),
cv::TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));
        // Save corners
        imagePoints.push_back(corners);

        drawChessboardCorners(image, patternSize, corners, found);
    }

    cameraMatrix = cv::Mat::eye(3, 3, CV_64F); //< Intrinsic camera matrix
    distCoeffs = cv::Mat::zeros(8, 1, CV_64F); //!< Distortion coefficients
    std::vector<cv::Mat> rvecs; //< Rotation vectors
    std::vector<cv::Mat> tvecs; //< Translation vectors

    // Set aspect ratio
    cameraMatrix.at<double>(0, 0) = aspRatio;

    if (imagePoints.size() > 0) {
        // Calculate cornerpoints
        std::vector<std::vector<cv::Point3f>> objectPoints(1);
        calcBoardCornerPosition(patternSize, patternMm, objectPoints[0]);
        objectPoints.resize(imagePoints.size(), objectPoints[0]);
        // Calibrate camera and get reprojection error rms
        double rms = cv::calibrateCamera(objectPoints, imagePoints, cv::Size(image.cols,
image.rows), cameraMatrix, distCoeffs, rvecs, tvecs, CV_CALIB_USE_INTRINSIC_GUESS);
    }
    bool ok = cv::checkRange(cameraMatrix) && cv::checkRange(distCoeffs);

    //std::vector<double> reprojErrs;
    //double totalAvgError = computeReprojectionErrors(objectPoints, imagePoints, rvecs,
    tvecs, cameraMatrix, distCoeffs, reprojErrs, 1);
    return ok;
}

void calibrateExtrinsics (cv::Mat& image, cv::Mat& homography, cv::Size patternSize, double
patternMm)
{
    cv::Mat grayImage;
    cvtColor(image, grayImage, CV_BGR2GRAY);

    std::vector<cv::Point2f> corners;
    bool found = findChessboardCorners(image, patternSize, corners,
CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS);

    if (found) {
        cornerSubPix(grayImage, corners, cv::Size(11,11), cv::Size(-1,-1),
cv::TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));

        cv::Point2f objectPoints[4];
        objectPoints[0] = cv::Point2f(0, 0);
        objectPoints[1] = cv::Point2f((patternSize.width-1)*patternMm, 0);
        objectPoints[2] = cv::Point2f(0, (patternSize.height-1)*patternMm);
    }
}

```

```

        objectPoints[3] = cv::Point2f((patternSize.width-1)*patternMm, (patternSize.height-1)*patternMm);
        cv::Point2f imagePoints[4];
        imagePoints[0] = corners[0];
        imagePoints[1] = corners[patternSize.width-1];
        imagePoints[2] = corners[(patternSize.height-1)*patternSize.width];
        imagePoints[3] = corners[(patternSize.height-1)*patternSize.width + patternSize.width-1];

        homography = getPerspectiveTransform(objectPoints, imagePoints);

        // Shift to image center
        cv::Mat t = cv::Mat::eye(3, 3, CV_64F);
        t.at<double>(0, 2) = 0.5*(image.cols-1) * (-1) + 0.5 * (patternMm * 0.5 * (patternSize.width-1)); // Shift width
        t.at<double>(1, 2) = (image.rows-1) * (-1) + 1.5 * (patternMm * 0.5 * (patternSize.height-1)); // Shift height
        t.at<double>(2, 2) = 0.5;
        homography *= t;

        drawChessboardCorners(image, patternSize, corners, found);
    }
}

float euclidDist (cv::Point2f p1, cv::Point2f p2)
{
    cv::Point2f diff = p1 - p2;
    return sqrt(diff.x*diff.x + diff.y*diff.y);
}

float calcPixelPerMm (cv::Mat image, cv::Size patternSize, float patternMm)
{
    float pxDist;
    float avgPxDist = 0.0;
    int distCnt = 0;
    cv::Mat grayImage;
    cvtColor(image, grayImage, CV_BGR2GRAY);

    std::vector<cv::Point2f> corners;
    bool found = findChessboardCorners(image, patternSize, corners,
    CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS);

    if (found) {
        cornerSubPix(grayImage, corners, cv::Size(11,11), cv::Size(-1,-1),
        cv::TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, patternMm, 0.1));

        for (auto i = 0; i < (corners.size()-1); i++) {
            if ((i%patternSize.width) < (patternSize.width-1)) {
                pxDist = euclidDist(corners[i+1], corners[i]);
                avgPxDist += pxDist;
                distCnt++;
            }
            else {
                i++;
            }
        }
        avgPxDist /= distCnt;
        std::cout << "Avg distance in px " << avgPxDist << std::endl;
        std::cout << "Px per mm " << avgPxDist/patternMm << std::endl;
    }
    return avgPxDist/patternMm;
}

void drawChessBoard (cv::Mat& image, cv::Size patternSize)
{
    cv::Mat grayImage;
    cvtColor(image, grayImage, CV_BGR2GRAY);

    std::vector<cv::Point2f> corners;
    bool found = findChessboardCorners(grayImage, patternSize, corners,
    CV_CALIB_CB_ADAPTIVE_THRESH | CV_CALIB_CB_FILTER_QUADS);
}

```

```

    if (found) {
        cornerSubPix(grayImage, corners, cv::Size(11,11), cv::Size(-1,-1),
cv::TermCriteria(CV_TERMCRIT_EPS + CV_TERMCRIT_ITER, 30, 0.1));
        drawChessboardCorners(image, patternSize, corners, found);
    }
}

void adjustImagePosition (cv::Mat image, cv::Mat& adjustedImage, char key, cv::Mat&
homography)
{
    double xOffset = 0;
    double yOffset = 0;
    double zOffset = 0;

    if (!image.empty()) {
        // Zoom in
        if (key == '+') {
            zOffset = -5;
        }
        // Zoom out
        else if (key == '-') {
            zOffset = 5;
        }
        // Move left
        else if (key == 'a') {
            xOffset = -5;
        }
        // Move up
        else if (key == 'w') {
            yOffset = -5;
        }
        // Move right
        else if (key == 'd') {
            xOffset = 5;
        }
        // Move down
        else if (key == 's') {
            yOffset = 5;
        }

        if (homography.empty()) {
            homography = cv::Mat::eye(3, 3, CV_64F);
        }
        homography.at<double>(0, 2) += xOffset;
        homography.at<double>(1, 2) += yOffset;
        homography.at<double>(2, 2) += zOffset;
    }
}

bool CameraConfig::load (cv::FileStorage fs)
{
    if (!fs.isOpened()) {
        fs.release();
        std::cerr << "ERROR: File storage not opened" << std::endl;
        return false;
    }
    else {
        fs["cameraID"] >> id;
        fs["cameraImageSize_width"] >> imageSize.width;
        fs["cameraImageSize_height"] >> imageSize.height;
        fs["cameraFPS"] >> fps;
    }
    fs.release();
    return true;
}

void CameraConfig::save (cv::FileStorage fs)
{
    if (!fs.isOpened()) {
        fs.release();
        std::cerr << "ERROR: File storage not opened" << std::endl;
    }
    else {

```

```

        fs.writeComment("camera config");
        fs << "cameraID" << id
        << "cameraImageSize_width" << imageSize.width
        << "cameraImageSize_height" << imageSize.height
        << "cameraFPS" << fps;
    }
    fs.release();
}

bool CameraCalibrationConfig::load (cv::FileStorage fs)
{
    if (!fs.isOpened()) {
        fs.release();
        std::cerr << "ERROR: File storage not opened" << std::endl;
        return false;
    }
    else {
        fs["camCalibImageSize_width"] >> imageSize.width;
        fs["camCalibImageSize_height"] >> imageSize.height;
        fs["camCalibPattern"] >> pattern;
        fs["camCalibPatternSize_width"] >> patternSize.width;
        fs["camCalibPatternSize_height"] >> patternSize.height;
        fs["camCalibPatternMm"] >> patternMm;
        fs["camCalibnumSamples"] >> numSamples;
        fs["cameraMatrix"] >> cameraMatrix;
        fs["distCoeffs"] >> distCoeffs;
        //~ fs["timeOfIntrCalib"] >> timeOfIntrCalib;
        fs["camCalibIntrDone"] >> intrCalibDone;
        fs["homography"] >> homography;
        //~ fs["timeOfExtrCalib"] >> timeOfExtrCalib;
        fs["camCalibExtrDone"] >> extrCalibDone;
        fs["transform"] >> transform;
        fs["mmPerPixel"] >> mmPerPixel;
    }
    fs.release();
    return true;
}

void CameraCalibrationConfig::save (cv::FileStorage fs)
{
    if (!fs.isOpened()) {
        fs.release();
        std::cerr << "ERROR: File storage not opened" << std::endl;
    }
    else {
        fs.writeComment("camera calibration config");
        fs << "camCalibImageSize_width" << imageSize.width
        << "camCalibImageSize_height" << imageSize.height
        << "camCalibPattern" << pattern
        << "camCalibPatternSize_width" << patternSize.width
        << "camCalibPatternSize_height" << patternSize.height
        << "camCalibPatternMm" << patternMm
        << "camCalibnumSamples" << numSamples;

        fs.writeComment("camera calibration data");
        fs << "camCalibIntrDone" << intrCalibDone
        //~ << "timeOfIntrCalib" << timeOfIntrCalib
        << "cameraMatrix" << cameraMatrix
        << "distCoeffs" << distCoeffs
        << "camCalibExtrDone" << extrCalibDone
        //~ << "timeOfExtrCalib" << timeOfExtrCalib
        << "homography" << homography
        << "transform" << transform
        << "mmPerPixel" << mmPerPixel;
    }
    fs.release();
}

```

Trajectory_data.hpp

```
/***
 * @file trajectory_data.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 29.3.2018
 */

#ifndef TRAJECTORY_DATA_HPP
#define TRAJECTORY_DATA_HPP

#include <iostream>
#include <atomic>
#include <mutex>
#include <opencv2/opencv.hpp>

///! @addtogroup planning
///! @{
// This file is part of the OpenCV library
// Copyright (C) 2014, OpenCV Foundation, all rights reserved.
// Third party copyrights are property of their respective owners.
// OpenCV library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public License
// as published by the Free Software Foundation, either version 3 of
// the License, or (at your option) any later version.
// OpenCV library is distributed in the hope that it will be useful,
// but WITHOUT ANY WARRANTY; without even the implied warranty of
// MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU
// Lesser General Public License for more details.
// You should have received a copy of the GNU Lesser General Public
// License along with this library; if not, see
// http://www.gnu.org/licenses/.
// 
```

```
/***
 * @brief A trajectory data class
 *
 * This Class describes a trajectory data class. A trajectory is defined
 * by pixel coordinates in a captured image.
 */
class TrajectoryData {
public:
    ~TrajectoryData() = default;

    /**
     * @brief Set trajectory data point vector
     *
     * This function sets a trajectory data point vector. It must
     * be sorted in descending order so that  $x_n & y_n$  <  $x_m & y_m$ , with  $y_n < y_m$ .
     * It provides synchronised access with a mutex lock.
     *
     * @param pv Point vector
     */
    void set (std::vector<cv::Point> pv);

    /**
     * @brief Get trajectory data point vector
     *
     * This function gets a trajectory data point vector.
     * It provides synchronised access with a mutex lock.
     *
     * @return Point vector
     */
    std::vector<cv::Point> get (void);

    /**
     * @brief Size of trajectory point vector
     *
     * This function returns the size of the trajectory data vector.
     *
     * @return Size of trajectory point vector
     */
    int size (void);

    /**
     * @brief Clear trajectory point vector
     *
     * This function clears the trajectory point vector content.
     * The vector size is 0 after this operation.
     */
    void clear (void);

    /**
     * @brief Set trajectory line
     *
     * This function sets a trajectory line described by two points, which
     * are the starting and the ending points of the line.
     */
}
```

```
*  
* @note This function is for compatibility usage only.  
*  
* @param l Line as integer vector  
*/  
void setLine (cv::Vec4i l);  
  
/**  
 * @brief Get trajectory line  
 *  
 * This function gets a trajectory line described by two points, which  
 * the starting and the ending points of the line.  
 *  
 * @note This function is for compatibility usage only.  
 *  
 * @return Line as integer vector  
*/  
cv::Vec4i getLine (void);  
  
std::atomic_bool active { false };  
  
private:  
    std::vector<cv::Point> points; //!< Trajectory data point vector  
    cv::Vec4i line; //!< For compatibility only  
    std::mutex lock; //!< Mutex lock  
};  
  
//! @} planning  
  
#endif // TRAJECTORY_DATA_HPP
```

Trajectory_data.cpp

```
/**  
 * @file trajectory_data.cpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 30.3.2018  
 */  
#include "trajectory_data.hpp"  
  
void TrajectoryData::set ({  
    std::lock_guard<std::mutex> guard(lock);  
    points = pv;  
}  
  
std::vector<cv::Point> TrajectoryData::get (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return points;  
}  
  
int TrajectoryData::size (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return points.size();  
}  
  
void TrajectoryData::clear (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    points.clear();  
}  
  
void TrajectoryData::setLine (cv::Vec4i l)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    line = l;  
}  
  
cv::Vec4i TrajectoryData::getLine (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return line;  
}
```

Path_planning.hpp

```

/***
 * @file path_planning.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 25.8.2017
 */

/***
 * @ingroup module
 * @defgroup planning Planning
 * @brief A behavior planning module
 */

/***
 * @ingroup planning
 * @defgroup path_planning Path planning
 * @brief A module to plan a path autonomously
 */

#ifndef PATH_PLANNING_HPP
#define PATH_PLANNING_HPP

#include <iostream>
#include <atomic>
#include <opencv2/opencv.hpp>
#include "module.hpp"
#include "camera_image_acquisitor.hpp"
#include "lane_data.hpp"
#include "lane_detection.hpp"
#include "obstacle_data.hpp"
#include "obstacle_detection.hpp"
#include "traffic_sign_data.hpp"
#include "vehicle_control.hpp"
#include "vehicle_data.hpp"
#include "trajectory_data.hpp"

///! @addtogroup path_planning Path planning
///! @{
class PathPlanner : public Module {
public:
    ~PathPlanner() = default;

    /**
     * @brief Start path planning
     *
     * This function runs the path planning thread. This thread plans
     * a path according to a detected lane marking. It then sets a
     * certain acceleration and steering angle.
     *
     * @param inputImage Input image data
     * @param outputImage Output image data
     * @param lane Actual lane data
     * @param trafficSigns Traffic sign data
     * @param obstacle Obstacle data
     * @param vehicle Vehicle data
     * @param trajectory Trajectory data
     */
    void run (ImageData& inputImage, ImageData& outputImage, LaneData& lane,
TrafficSignData& trafficSigns, ObstacleData& obstacle, VehicleModel& vehicle,
TrajectoryData& trajectory);

private:
    CameraConfig camConfig;
    ObstacleDetectionConfig obstacleDetConfig;
};

/**
 * @brief Calculate trajectory
 *
 * This function calculates a trajectory to be driven by the autonomous car

```

```
* @param actualLane Actual lane composed of left and right road marking line
* @param trajectory Trajectory data
* @param kfT Kalman filter for the Trajectory
* @param imageSize Size of captured image
*/
void calcTrajectory (std::vector<cv::Vec4i> actualLane, TrajectoryData& trajectory,
cv::KalmanFilter kfT, cv::Size imageSize);

/** @brief Draw trajectory data point vector
*
* This function draws the trajectory by drawing a line between
* the first and the last point from the point vector.
*
* @param image
* @param trajectory
*/
void drawTrajectory (cv::Mat& image, TrajectoryData& trajectory);

/** @brief Draw trajectory line
*
* This function draws the trajectory by drawing the line describ
*
* @param image
* @param trajectory
*/
void drawTrajectoryLine (cv::Mat& image, TrajectoryData& trajectory);

//! @} path_planning

#endif // PATH_PLANNING_HPP
```

Path_planning.cpp

```
/*
 * @file path_planning.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 25.8.2017
 */

#include "path_planning.hpp"
#include "configuration.hpp"

void PathPlanner::run (ImageData& inputImage, ImageData& outputImage, LaneData& lane,
TrafficSignData& trafficSigns, ObstacleData& obstacles, VehicleModel& vehicle,
TrajectoryData& trajectory)
{
    std::cout << "THREAD: Path planning started." << std::endl;
    running = true;

    Configurator& config = Configurator::instance();
    camConfig = config.getCameraConfig();

    cv::KalmanFilter kfT(4, 4, 0);
    initLinePrediction(kfT, 4);

    cv::Vec4i actualLeftLine;
    cv::Vec4i actualRightLine;

    while (running && !error) {
        std::vector<cv::Vec4i> actualLane;
        actualLane.push_back(lane.getLeftLine());
        actualLane.push_back(lane.getRightLine());

        // Check safety distance before trajectory calculation
        bool safetyDistance = true;
        if (obstacles.getDistance() < 25.0) {
            safetyDistance = false;
        }
        // Check traffic sign distance before trajectory calculation
        if (trafficSigns.getDistance() > (-1)) {
            cv::Rect stopSign = trafficSigns.getRoi();
            // Get traffic sign distance by traffic sign size
            if (stopSign.width > 30 || stopSign.height > 30) {
                safetyDistance = false;
                //~ std::cout << "INFO: Stop sign in safety distance!" << std::endl;
            }
        }

        if (safetyDistance && (actualLane.size() > 1)) {
            vehicle.releaseStop();
            calcTrajectory(actualLane, trajectory, kfT, camConfig.imageSize);
            cv::Mat image;
            image = inputImage.read();
            if (trajectory.active) {
                drawTrajectoryLine(image, trajectory);
                outputImage.write(image);
                outputImage.setTime(inputImage.getTime());
                //~ trajectory.active = false;
            }
        }
        else {
            vehicle.stop();
            //~ vehicle.setSteering(CV_PI/2);
            //~ vehicle.setAcceleration(0);
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(20));
    }

    std::cout << "THREAD: Path planning ended." << std::endl;
}

void calcTrajectory (std::vector<cv::Vec4i> actualLane, TrajectoryData& trajectory,
cv::KalmanFilter kfT, cv::Size imageSize)
```

```

{
    std::vector<cv::Vec4i> trajectoryPredicted;
    bool leftLineFound = false;
    bool rightLineFound = false;

    if (actualLane.size() > 1) {
        // Check for left line
        if (actualLane.front() != cv::Vec4i(0, 0, 0, 0)) {
            leftLineFound = true;
        }
        // Check for right line
        if (actualLane.back() != cv::Vec4i(imageSize.width-1, 0, imageSize.width-1, 0)) {
            rightLineFound = true;
        }

        // @todo Convert from Vec4i to Point vector
        cv::Vec4i laneMid(imageSize.width/2-1, 0, imageSize.width/2-1, imageSize.height-1);
        cv::Vec4i viewMid(imageSize.width/2-1, 0, imageSize.width/2-1, imageSize.height-1);

        if (leftLineFound && rightLineFound) {
            laneMid = getLaneMid(actualLane);
            std::vector<cv::Vec4i> lM;
            lM.push_back(laneMid);
            predictLine(lM, kfT, 4, trajectoryPredicted);
            if (trajectoryPredicted.size() > 0) {
                laneMid = trajectoryPredicted.front();
            }
        }
        else if (leftLineFound && !rightLineFound) {
            laneMid = actualLane.front();
            std::vector<cv::Vec4i> lM;
            lM.push_back(laneMid);
            predictLine(lM, kfT, 4, trajectoryPredicted);
            if (trajectoryPredicted.size() > 0) {
                laneMid = trajectoryPredicted.front();
            }
        }
        else if (!leftLineFound && rightLineFound) {
            laneMid = actualLane.back();
            std::vector<cv::Vec4i> lM;
            lM.push_back(laneMid);
            predictLine(lM, kfT, 4, trajectoryPredicted);
            if (trajectoryPredicted.size() > 0) {
                laneMid = trajectoryPredicted.front();
            }
        }
        trajectory.setLine(laneMid);
        trajectory.active = true;
    }
    else {
        trajectory.active = false;
    }
}

void drawTrajectory (cv::Mat& image, TrajectoryData& trajectory) {
    std::vector<cv::Point> tPoints = trajectory.get();
    //~ line(image, tPoints.front(), tPoints.back(), cv::Scalar(200,200,0), 2);
    const cv::Point *pts = (const cv::Point*) cv::Mat(tPoints).data;
    int numPts = cv::Mat(tPoints).rows;
    polylines(image,&pts, &numPts, 2, false, cv::Scalar(200,200,0));
}

void drawTrajectoryLine (cv::Mat& image, TrajectoryData& trajectory) {
    cv::Vec4i tLine = trajectory.getLine();
    line(image, cv::Point(tLine[0], tLine[1]), cv::Point(tLine[2], tLine[3]),
    cv::Scalar(200,200,0), 2);
}

```

Remote_control.hpp

```
/***
 * @file remote_control.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 11.10.2017
 */

/***
 * @ingroup planning
 * @defgroup remote_control Remote Control
 * @brief A module to remote control the system by an user
 */

#ifndef REMOTE_CONTROL_HPP
#define REMOTE_CONTROL_HPP

#include <iostream>
#include <atomic>
#include <opencv2/opencv.hpp>
#include "module.hpp"
#include "user_interface.hpp"
#include "vehicle_data.hpp"

//! @addtogroup remote_control
//! @{
class RemoteController : public Module {
public:
    ~RemoteController() = default;

    /**
     * @brief Run remote control the system
     *
     * This thread enables user to control the system remotely.
     * The key w is used for acceleration.
     * The key s is used for reducing speed and for accalarating backwards.
     * The key a is used for steering to the left.
     * The key d is used for steering to the right.
     * The spacebar is used for breaking.
     *
     * @param vehicle Vehicle data
     * @param uiState User interface
     */
    void run (VehicleModel& vehicle, UserInterfaceState& uiState);
};

//! @} remote_control
#endif // REMOTE_CONTROL_HPP
```

Remote_control.cpp

```


/***
 * @file remote_control.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 11.10.2017
 */

#include "remote_control.hpp"
#include <thread>

void RemoteController::run (VehicleModel& vehicle, UserInterfaceState& uiState)
{
    std::cout << "THREAD: Remote control started." << std::endl;
    running = true;

    char key = (char)(-1);
    vehicle.setAcceleration(0);
    vehicle.setSteering(CV_PI/2);

    while (running && !error) {
        char prevKey = key;
        key = uiState.getKey();

        double prevAcceleration = vehicle.getAcceleration();
        double prevSteering = vehicle.getSteering();

        if (prevKey != key) {
            double acceleration = prevAcceleration;
            double steering = prevSteering;

            // Set acceleration from -100 to 100 %
            if (key == 'w') {
                acceleration += 0.5;
            }
            else if (key == 's') {
                acceleration -= 0.5;
            }
            // Stop vehicle
            else if (key == ' ') {
                acceleration = 0;
                vehicle.stop();
            }

            // Top speed at +- 30%
            if (acceleration < (-30)) {
                acceleration = -30;
            }
            else if (acceleration > 30) {
                acceleration = 30;
            }

            // Set steering from 0 to pi
            if (key == 'a') {
                steering -= CV_PI/8;
            }
            else if (key == 'd') {
                steering += CV_PI/8;
            }

            if (steering < (CV_PI/4)) {
                steering = (CV_PI/4);
            }
            else if (steering > (3*CV_PI/4)) {
                steering = (3*CV_PI/4);
            }

            if (prevAcceleration != acceleration) {
                std::cout << "INFO: Actual acceleration: " << acceleration << " %" <<
std::endl;
                vehicle.setAcceleration(acceleration);
            }
            if (prevSteering != steering) {


```

```
        std::cout << "INFO: Actual steering: " << steering << " rad" << std::endl;
        vehicle.setSteering(steering);
    }
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    vehicle.releaseStop();
}

std::cout << "THREAD: Remote control ended." << std::endl;
```

Vehicle_data.hpp

```


/***
 * @file vehicle_data.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 29.6.2017
 */

#ifndef VEHICLE_DATA_HPP
#define VEHICLE_DATA_HPP

#include <iostream>
#include <mutex>

//! @addtogroup vehicle_control
//! @{
//!

/***
 * @brief An enum to describe the steering direction of the vehicle
 */
enum VehicleDirection {
    VEHICLE_LEFT = -1,
    VEHICLE_STRAIGHT = 0,
    VEHICLE_RIGHT = 1
};

/***
 * @brief An enum to describe the driving direction and gear of the
 *        vehicle.
 */
enum VehicleGear {
    VEHICLE_FORWARD = 1,
    VEHICLE_GEAR_1 = 1,
    VEHICLE_GEAR_2 = 2,
    VEHICLE_GEAR_3 = 3,
    VEHICLE_GEAR_4 = 4,
    VEHICLE_GEAR_5 = 5,
    VEHICLE_GEAR_6 = 6,
    VEHICLE_NEUTRAL = 0,
    VEHICLE_BACKWARD = -1,
};

struct VehicleDimensions {
    double width; //!< Vehicle width in mm
    double length; //!< Vehicle length in mm
    double height; //!< Vehicle height in mm
    double wheelbase; //!< Vehicle wheelbase in mm
    double frontOverhang; //!< Vehicle front overhang in mm
    double rearOverhang; //!< Vehicle rear overhang in mm
    void set (double w, double l, double h, double wb, double fo, double ro); //!< Set
overall vehicle dimensions
};

struct VehicleData {
    double acceleration; //!< Accerelration from -100 to 100 percent
    double speed; //!< Speed in m/s
    VehicleGear gear; //!< Driving gear
    double steering; //!< Steering angle in radian
    VehicleDirection direction; //!< Direction from -1 to 1
    VehicleDimensions dimensions; //!< Vehicle overall dimensions
};

/***
 * @brief A class to describe a vehicle
 */
class VehicleModel {
public:
    ~VehicleModel () = default;

    /**
     * @brief A function to set the vehicle acceleration
     *
     * This function sets the vehicle acceleration from 0 to 100 %.
     */
};


```

```

/*
 * @param value Percentage from 0 to 100 %
 */
void setAcceleration (double value);

/**
 * @brief A function to get the vehicle acceleration
 *
 * This function gets the vehicle acceleration from 0 to 100 %.
 *
 * @return Acceleration from 0 to 100 %
 */
double getAcceleration (void);

/**
 * @brief A function to set the vehicle steering direction
 *
 * This function sets the vehicle steering direction.
 *
 * @param value Steering direction
 */
void setDirection (VehicleDirection value);

/**
 * @brief A function to get the vehicle steering direction
 *
 * This function gets the vehicle steering direction.
 *
 * @return Steering direction
 */
VehicleDirection getDirection (void);

/**
 * @brief A function to set the vehicle speed
 *
 * This function sets the vehicle speed in m/s.
 *
 * @param value Speed value in m/s
 */
void setSpeed (double value);

/**
 * @brief A function to get the vehicle speed
 *
 * This function gets the vehicle speed in m/s.
 *
 * @return Speed value in m/s
 */
double getSpeed (void);

/**
 * @brief A function to set the vehicle steering angle
 *
 * This function sets the vehicle steering angle in radians from 0 to &pi;.
 *
 * @param value Steering angle in radians from 0 to &pi;
 */
void setSteering (double value);

/**
 * @brief A function to get the vehicle steering angle
 *
 * This function gets the vehicle steering angle in radians from 0 to &pi;.
 *
 * @return Steering angle in radians from 0 to &pi;
 */
double getSteering (void);

/**
 * @brief A function to set the vehicle width
 *
 * This function sets the vehicle width in mm.
 */

```

```
* @param value Vehicle width in mm
*/
void setWidth (double value);

/***
 * @brief A function to get the vehicle width
 *
 * This function gets the vehicle width in mm.
 *
 * @return Vehicle width in mm
 */
double getWidth (void);

/***
 * @brief A function to set the vehicle length
 *
 * This function sets the vehicle length in mm.
 *
 * @param value Vehicle length in mm
 */
void setLength (double value);

/***
 * @brief A function to get the vehicle length
 *
 * This function gets the vehicle length in mm.
 *
 * @return Vehicle length in mm
 */
double getlength (void);

/***
 * @brief A function to set the vehicle height
 *
 * This function sets the vehicle height in mm.
 *
 * @param value Vehicle height in mm
 */
void setHeight (double value);

/***
 * @brief A function to get the vehicle height
 *
 * This function gets the vehicle height in mm.
 *
 * @return Vehicle height in mm
 */
double getHeight (void);

/***
 * @brief A function to set the vehicle wheelbase length
 *
 * This function sets the vehicle wheelbase length in mm.
 *
 * @param value Vehicle wheelbase length in mm
 */
void setWheelbase (double value);

/***
 * @brief A function to get the vehicle wheelbase length
 *
 * This function gets the vehicle wheelbase length in mm.
 *
 * @return Vehicle wheelbase length in mm
 */
double getWheelbase (void);

/***
 * @brief A function to set the vehicle front overhang
 *
 * This function sets the vehicle front overhang in mm.
 *
 * @param value Vehicle front overhang in mm
 */
```

```

*/
void setFrontOverhang (double value);

/**
 * @brief A function to get the vehicle front overhang
 *
 * This function gets the vehicle front overhang in mm.
 *
 * @return Vehicle front overhang in mm
 */
double getFrontOverhang (void);

/**
 * @brief A function to set the vehicle rear overhang
 *
 * This function sets the vehicle rear overhang in mm.
 *
 * @param value Vehicle rear overhang in mm
 */
void setRearOverhang (double value);

/**
 * @brief A function to get the vehicle rear overhang
 *
 * This function gets the vehicle rear overhang in mm.
 *
 * @return Vehicle rear overhang in mm
 */
double getRearOverhang (void);

/**
 * @brief A function to set the vehicle overall dimensions
 *
 * This function sets the vehicle overall dimensions in mm.
 *
 * @param w Width
 * @param l Length
 * @param h Height
 * @param wb Wheelbase
 * @param f0 Front overhang
 * @param r0 Rear overhang
 */
void setDimensions (double w, double l, double h, double wb, double f0, double r0);

/**
 * @brief A function to set the vehicle overall dimensions
 *
 * This function sets the vehicle overall dimensions in mm.
 *
 * @param d Dimensions
 */
void setDimensions (VehicleDimensions d);

/**
 * @brief A function to get the vehicle overall dimensions
 *
 * This function gets the vehicle overall dimensions in mm.
 *
 * @return Vehicle dimensions
 */
VehicleDimensions getDimensions (void);

/**
 * @brief Stop vehicle
 *
 * This function stops the vehicle
 */
void stop (void);

/**
 * @brief Release vehicle stop

```

```
*  
* This function releases the vehicle engine stop.  
*/  
void releaseStop (void);  
  
/**  
 * @brief Check if vehicle stop is active  
 *  
 * This function checks if vehicle stop is active.  
 *  
 * return Stop status  
 */  
bool checkStop (void);  
  
private:  
    //~ VehicleData vehicle;  
    double acceleration; //!< Accerelration from -100 to 100 percent  
    double speed; //!< Speed in m/s  
    VehicleGear gear; //!< Driving gear  
    double steering; //!< Steering angle in radian  
    VehicleDirection direction; //!< Vehicle direction from -1 to 1  
    VehicleDimensions dimensions; //!< Vehicle overall dimensions  
    bool stopFlag {false};  
    std::mutex lock;  
};  
  
//! @} vehicle_control  
  
#endif // VEHICLE_DATA_HPP
```

Vehicle_data.cpp

```


/*
 * @file vehicle_data.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

#include "vehicle_data.hpp"

void VehicleDimensions::set(double w, double l, double h, double wb, double f0, double r0)
{
    width = w;
    length = l;
    height = h;
    wheelbase = wb;
    frontOverhang = f0;
    rearOverhang = r0;
}

void VehicleModel::setAcceleration (double value)
{
    std::lock_guard<std::mutex> guard(lock);
    acceleration = value;
}

double VehicleModel::getAcceleration (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return acceleration;
}

void VehicleModel::setDirection (VehicleDirection value)
{
    std::lock_guard<std::mutex> guard(lock);
    direction = value;
}

VehicleDirection VehicleModel::getDirection (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return direction;
}

void VehicleModel::setSpeed (double value)
{
    std::lock_guard<std::mutex> guard(lock);
    speed = value;
}

double VehicleModel::getSpeed (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return speed;
}

void VehicleModel::setSteering (double value)
{
    std::lock_guard<std::mutex> guard(lock);
    steering = value;
}

double VehicleModel::getSteering (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return steering;
}

void VehicleModel::setWidth (double value)
{
    std::lock_guard<std::mutex> guard(lock);
    dimensions.width = value;
}


```

```

double VehicleModel::getWidth (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return dimensions.width;
}

void VehicleModel::setLength (double value)
{
    std::lock_guard<std::mutex> guard(lock);
    dimensions.length = value;
}

double VehicleModel::getLength (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return dimensions.length;
}

void VehicleModel::setHeight (double value)
{
    std::lock_guard<std::mutex> guard(lock);
    dimensions.height = value;
}

double VehicleModel::getHeight(void)
{
    std::lock_guard<std::mutex> guard(lock);
    return dimensions.height;
}

void VehicleModel::setWheelbase(double value)
{
    std::lock_guard<std::mutex> guard(lock);
    dimensions.wheelbase = value;
}

double VehicleModel::getWheelbase(void)
{
    std::lock_guard<std::mutex> guard(lock);
    return dimensions.wheelbase;
}

void VehicleModel::setFrontOverhang(double value)
{
    std::lock_guard<std::mutex> guard(lock);
    dimensions.frontOverhang = value;
}

double VehicleModel::getFrontOverhang(void)
{
    std::lock_guard<std::mutex> guard(lock);
    return dimensions.frontOverhang;
}

void VehicleModel::setRearOverhang(double value)
{
    std::lock_guard<std::mutex> guard(lock);
    dimensions.rearOverhang = value;
}

double VehicleModel::getRearOverhang(void)
{
    std::lock_guard<std::mutex> guard(lock);
    return dimensions.rearOverhang;
}

void VehicleModel::setDimensions(double w, double l, double h, double wb, double fo, double r0)
{
    std::lock_guard<std::mutex> guard(lock);
    dimensions.width = w;
    dimensions.length = l;
}

```

```
dimensions.height = h;
dimensions.wheelbase = wb;
dimensions.frontOverhang = f0;
dimensions.rearOverhang = r0;
}

void VehicleModel::setDimensions(VehicleDimensions d)
{
    std::lock_guard<std::mutex> guard(lock);
    dimensions = d;
}

VehicleDimensions VehicleModel::getDimensions (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return dimensions;
}

void VehicleModel::stop (void)
{
    std::lock_guard<std::mutex> guard(lock);
    stopFlag = true;
}

void VehicleModel::releaseStop (void)
{
    std::lock_guard<std::mutex> guard(lock);
    stopFlag = false;
}

bool VehicleModel::checkStop (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return stopFlag;
}
```

Vehicle_control.hpp

```
/***
 * @file vehicle_control.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 3.7.2017
 */

/***
 * @ingroup module
 * @defgroup vehicle_control Vehicle Control
 * @brief A module to control the vehicle state
 */

#ifndef VEHICLE_CONTROL_HPP
#define VEHICLE_CONTROL_HPP

#include <iostream>
#include <atomic>
#include <opencv2/opencv.hpp>
#include "module.hpp"
#include "trajectory_data.hpp"
#include "vehicle_data.hpp"
#include "motor_driver.hpp"

//! @addtogroup vehicle_control
//! @{
class VehicleController : public Module {
public:
    ~VehicleController() = default;

    /**
     * @brief Run vehicle control
     *
     * This function runs the vehicle control thread.
     * This thread controls the vehicle by converting given values and
     * sending them to the motor driver.
     *
     * @param trajectory Trajectory data
     * @param vehicle Vehicle to control
     */
    void run (TrajectoryData& trajectory, VehicleModel& vehicle);
};

//! @} vehicle_control
#endif // VEHICLE_CONTROL_HPP
```

Vehicle_control.cpp

```
/*
 * @file vehicle_control.cpp
 * @author Sergiu-Petru Tabcariu
 * @date 3.7.2017
 */

#include "vehicle_control.hpp"
#include "configuration.hpp"
#include "camera_image_acquisitor.hpp"
#include "lane_detection.hpp"

void VehicleController::run (TrajectoryData& trajectory, VehicleModel& vehicle)
{
    std::cout << "THREAD: Vehicle control started." << std::endl;
    running = true;

    Configurator& config = Configurator::instance();
    CameraConfig camConfig = config.getCameraConfig();

    MotorDriver motor;
    if (!motor.init()) {
        error = true;
        running = false;
    }

    while (running && !error) {
        if (vehicle.checkStop()) {
            //~ motor.reset();
            //~ vehicle.setSteering(CV_PI/2);
            vehicle.setAcceleration(0);
            std::this_thread::sleep_for(std::chrono::milliseconds(250));
        }
        else {
            cv::Vec4i tLine = trajectory.getLine();
            std::vector<cv::Point> tPoints = trajectory.get();
            if (trajectory.active) {
                // Convert Trajectory to steering and acceleration values
                float theta = 0;
                int diffX1 = 0;
                int diffX2 = 0;

                double acVal = 18;
                double brVal = 0;

                theta = getTheta(cv::Point(tLine[0], tLine[1]), cv::Point(tLine[2],
tLine[3]));

                if ((theta < (CV_PI*0.1)) || (theta > (CV_PI*0.9))) {
                    vehicle.setAcceleration(brVal);
                }
                else {
                    // Set steering angle
                    if ((theta > 0) && (theta < CV_PI/2*0.9)) {
                        vehicle.setDirection(VehicleDirection::VEHICLE_LEFT);
                        vehicle.setSteering(theta-CV_PI/8);
                    }
                    else if ((theta < CV_PI) && (theta > CV_PI/2*1.1)) {
                        vehicle.setDirection(VehicleDirection::VEHICLE_RIGHT);
                        vehicle.setSteering(theta+CV_PI/8);
                    }
                    else {
                        vehicle.setDirection(VehicleDirection::VEHICLE_STRAIGHT);
                        vehicle.setSteering(theta);
                    }
                }

                diffX1 = tLine[0] - (camConfig.imageSize.width/2-1); // If result
positive then lane is to much to the right. Car must steer to the right.
                diffX2 = tLine[2] - (camConfig.imageSize.width/2-1);
                //~ diffX1 = tPoints.front().x - (camConfig.imageSize.width/2-1); // If
result positive then lane is to much to the right. Car must steer to the right.
                //~ diffX2 = tPoints.back().x - (camConfig.imageSize.width/2-1);
            }
        }
    }
}
```

```
// Set acceleration percentage
// Vehicle is too much to the left
// Counter steer to the right
if ((diffX2 > 10) && (diffX1 > 10)) {
    vehicle.setSteering(3*CV_PI/4);
    vehicle.setAcceleration(acVal-0.5);
}
// Vehicle is too much to the right
// Counter steer to the left
else if ((diffX2 < -10) && (diffX1 < -10)) {
    vehicle.setSteering(CV_PI/4);
    vehicle.setAcceleration(acVal-0.5);
}
else {
    vehicle.setAcceleration(acVal);
}
}

// Calculate steering value from rad to a value from 0 to 4096
double steering = vehicle.getSteering();
int steeringValue = (int) round(STEERING_MIN + steering/((double)
CV_PI/(STEERING_MAX - STEERING_MIN)));
motor.setSteering(steeringValue);

// Calculate acceleration value from percent to value from 0 to 4096
double acceleration = vehicle.getAcceleration();
int accelerationValue = (int) round(ESC_N + acceleration/((double) 100/(ESC_N -
ESC_MIN)));
motor.setAcceleration(accelerationValue);

std::this_thread::sleep_for(std::chrono::milliseconds(10));
}

motor.reset();

std::cout << "THREAD: Vehicle control ended." << std::endl;
}
```

Motor_driver.hpp

```
/**
 * @file motor_driver.hpp
 * @author Sergiu-Tabacariu
 * @date 30.6.2017
 */

#ifndef MOTOR_DRIVER_HPP
#define MOTOR_DRIVER_HPP

#include <iostream>
#include <mutex>
#include <thread>
#include <chrono>
#include <wiringPi.h>
#include "PCA9685.h"

//! @addtogroup vehicle_control
//! @{

/** 
 * @brief An enum with PWM steering boundaries
 */
enum SteeringPWM {
    STEERING = 0,
    STEERING_MIN = 204,
    STEERING_MAX = 409,
    STEERING_LEFT = STEERING_MIN,
    STEERING_RIGHT = STEERING_MAX,
    STEERING_STRAIGHT = (STEERING_MIN+(STEERING_MAX-STEERING_MIN)/2)
};

/** 
 * @brief An enum with PWM ESC boundaries
 */
enum ESCPWM {
    ESC = 1,
    ESC_MIN = 204,
    ESC_MAX = 409,
    ESC_D = 358,
    ESC_R = 250,
    ESC_N = (ESC_MIN+(ESC_MAX-ESC_MIN)/2)
};

/** 
 * @brief An enum for motor driver driving direction
 */
enum MotorDriverDirection {
    MOTOR_DRIVER_DIRECTION_NONE,
    MOTOR_DRIVER_DIRECTION_FORWARD,
    MOTOR_DRIVER_DIRECTION_BACKWARD
};

/** 
 * @brief An enum for motor driver steering direction
 */
enum MotorDriverSteering {
    MOTOR_DRIVER_STEERING_NONE,
    MOTOR_DRIVER_STEERING_STRAIGHT,
    MOTOR_DRIVER_STEERING_LEFT,
    MOTOR_DRIVER_STEERING_RIGHT
};

/** 
 * @brief A class for the motor driver direction and acceleration
 */
class MotorDriver {
public:
    ~MotorDriver() = default;

    /**
     * @brief Initialize motor driver

```

```

/*
 * This function initializes the motor driver.
 *
 * @return Init done
 */
bool init ();

/**
 * @brief Set steering motor value
 *
 * This function sets the steering motor value.
 *
 * @param value Steering value from 0 to 4095
 */
void setSteering (int value);

/**
 * @brief Get steering motor value
 *
 * This function gets the steering motor value.
 *
 * @return Steering value between 0 to 4095
 */
int getSteering (void);

/**
 * @brief Set acceleration motor value
 *
 * This function sets the acceleration motor value.
 *
 * @param value Acceleration value from 0 to 4095
 */
void setAcceleration (int value);

/**
 * @brief Get acceleration motor value
 *
 * This function gets the acceleration motor value.
 *
 * @return acceleration value between 0 to 4095
 */
int getAcceleration (void);

/**
 * @brief A function to reset the motor driver
 *
 * This function resets the motor driver.
 */
void reset (void);

/**
 * @brief A function to stop the motor driver
 *
 * This function stops the motor driver.
 */
void stop (void);

private:
PCA9685 pwmModule; //!< PWM Motor driver
int outputEnablePin {0}; //!< WiringPi pin number
int steering {STEERING_STRAIGHT}; //!< VehicleDirection from 0 to 4095
int acceleration {ESC_N}; //!< Acceleration from 0 to 4095
bool initFlag {false};
//~ std::mutex lock; //!< Mutex lock for synchronized access
};

/// @} vehicle_control
#endif // MOTOR_DRIVER_HPP

```

Motor_driver.cpp

```


/***
 * @file motor_driver.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 30.8.2017
 */

#include "motor_driver.hpp"
#include <wiringPi.h>

bool MotorDriver::init ()
{
    // Initialize wiringPi for enabling the motor driver output
    if (wiringPiSetup() == -1) {
        std::cerr << "ERROR: Couldn't init wiringPi library!" << std::endl;
        return false;
    }
    else {
        pinMode(outputEnablePin, OUTPUT);
        digitalWrite(outputEnablePin, LOW);
    }

    if (pwmModule.init(1, 0x40)) {
        pwmModule.setPWFreq(50);
        pwmModule.setPWM(ESC, ESC_N);
        std::this_thread::sleep_for(std::chrono::seconds(3));
        initFlag = true;
        return true;
    }
    else {
        std::cerr << "ERROR: Couldn't initialize motor driver!" << std::endl;
        return false;
    }
}

void MotorDriver::setSteering (int value)
{
    if (initFlag) {
        digitalWrite(outputEnablePin, LOW);
        pwmModule.setPWM(STEERING, value);
    }
}

void MotorDriver::setAcceleration (int value)
{
    if (initFlag) {
        digitalWrite(outputEnablePin, LOW);
        pwmModule.setPWM(ESC, value);
    }
}

void MotorDriver::reset (void)
{
    if (initFlag) {
        pwmModule.setPWM(STEERING, 0);
        pwmModule.setPWM(ESC, 0);
        pwmModule.reset();

        // Switch motors off and on
        digitalWrite(outputEnablePin, HIGH);
        std::this_thread::sleep_for(std::chrono::milliseconds(250));
        digitalWrite(outputEnablePin, LOW);
    }
}

void MotorDriver::stop (void)
{
    digitalWrite(outputEnablePin, HIGH);
    std::this_thread::sleep_for(std::chrono::milliseconds(250));
    digitalWrite(outputEnablePin, LOW);
}


```

Modi

System_state.hpp

```

/***
 * @file system_state.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 10.4.2018
 */

/***
 * @defgroup system_state System State
 * @brief The system state
 */

#ifndef SYSTEM_STATE_HPP
#define SYSTEM_STATE_HPP

#include <iostream>
#include <thread>
#include <atomic>

//! @addtogroup system_state
//! @{
class SystemMode; //!< Forward declaration

/***
 * @brief A system state class
 *
 * This class describes the system state of the autonomous driver.
 */
class SystemState {
public:
    SystemState (SystemMode* m) : mode(m) {};
    ~SystemState () = default; //!< Default destructor

    /**
     * @brief Set the system state
     *
     * This function sets the system state. First it locks the state
     * information resource.
     *
     * @param m Reference to new system state to be set
     */
    void setMode (SystemMode* m);

    /**
     * @brief Get the system state
     *
     * This function gets the actual system state. First it locks the
     * state information resource.
     *
     * @return The reference to the system state
     */
    SystemMode* getMode (void);

    /**
     * @brief Run system state
     *
     * This function runs the system state.
     */
    void run (void);

    /**
     * @brief Quit system state
     *
     * This function quits the system state. It quits the running system
     * mode.
     */
    void quit (void);
}

```

```
/**  
 * @brief Is system running  
 *  
 * This function checks if the system is running.  
 *  
 * @return True if system is running, else false.  
 */  
bool isRunning (void);  
  
private:  
    SystemMode* mode; //!< Actual system mode  
    std::atomic_bool running {false}; //!< Running flag  
};  
  
//! @} system_state  
  
#endif // SYSTEM_STATE_HPP
```

System_state.cpp

```
/**  
 * @file system_state.cpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 6.12.2017  
 */  
  
#include "system_state.hpp"  
#include "system_mode.hpp"  
  
void SystemState::setMode (SystemMode* m)  
{  
    mode = m;  
    std::cout << "System state changed." << std::endl;  
}  
  
SystemMode* SystemState::getMode ()  
{  
    return mode;  
}  
  
void SystemState::run ()  
{  
    std::cout << "SYSTEM: Starting..." << std::endl;  
    running = true;  
    mode->run(this);  
}  
  
void SystemState::quit ()  
{  
    std::cout << "SYSTEM: Stopping..." << std::endl;  
    running = false;  
    if (mode->isRunning()) {  
        mode->quit();  
    }  
}  
  
bool SystemState::isRunning ()  
{  
    return running;  
}
```

System_mode.hpp

```
/***
 * @file      system_mode.hpp
 * @author    Sergiu-Petru Tabacariu
 * @date      5.5.2018
 */

/***
 * @ingroup system_state
 * @defgroup system_mode System Modes
 */

#ifndef SYSTEM_MODE_HPP
#define SYSTEM_MODE_HPP

#include <atomic>
#include "system_state.hpp"

//! @addtogroup system_mode
//! @{
//! @brief A system mode abstract class
//! @ This class is a system mode abstract class for interfacing system modes.
//!
class SystemMode {
public:
    virtual ~SystemMode () = default;

    /**
     * @brief Run system mode
     *
     * This function runs the system mode.
     *
     * @param s System state pointer
     */
    virtual void run (SystemState* s) = 0;

    /**
     * @brief Quit system mode
     *
     * This function quits the system mode.
     */
    virtual void quit (void) = 0;

    /**
     * @brief Check if system mode is running
     *
     * This function checks if the system mode is running.
     *
     * @return True if system mode is running, else false.
     */
    virtual bool isRunning (void) { return running; };

protected:
    std::atomic_bool running {false}; //!<< Running flag
    std::atomic_bool error {false}; //!<< Error flag
};

//! @} system_mode
#endif // SYSTEM_MODE_HPP
```

Standby_mode.hpp

```
/***
 * @file standby_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

/***
 * @ingroup system_mode
 * @defgroup standby_mode Standby Mode
 * @brief A mode for the system standby
 */

#ifndef STANDBY_MODE_HPP
#define STANDBY_MODE_HPP

#include <iostream>
#include <thread>
#include "system_mode.hpp"
#include "system_state.hpp"
#include "user_interface_state.hpp"
#include "image_data.hpp"
#include "user_interface.hpp"
#include "camera_image_acquisitor.hpp"

//! @addtogroup standby_mode
//! @{
class StandbyMode : public SystemMode {
public:
    ~StandbyMode() = default;

    /**
     * @brief Start standby system mode
     *
     * This function starts the system standby mode.
     * This mode displays the main menu and the actual camera view.
     *
     * @param s System state pointer
     */
    void run (SystemState* s) override;

    /**
     * @brief Stop standby system mode
     *
     * This function stops the system standby mode.
     */
    void quit (void) override;

    /**
     * @brief Stop autonomous driving mode modules
     *
     * This function stops all autonomous driving mode modules.
     */
    void stopModules (void);

private:
    ImageData inputImage; //!< Input image captured by an image acquisitor
    UserInterfaceState uiState; //!< User interface user input data
    UserInterface ui; //!< User interface
    CameraImageAcquisitor camera; //!< Camera image acquisitor
};

//! @} standby_mode
#endif // STANDBY_MODE_HPP
```

Standby_mode.cpp

```
/*
 * @file standby_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

#include "standby_mode.hpp"
#include "closing_mode.hpp"
#include "error_mode.hpp"
#include "autonomous_mode.hpp"
#include "development_mode.hpp"
#include "remote_control_mode.hpp"
#include "configuration_mode.hpp"
#include "about_mode.hpp"
#include "ui_standby_mode.hpp"

void StandbyMode::run (SystemState* s)
{
    std::cout << "-----" << std::endl;
    std::cout << "MODE: Standby Mode started." << std::endl;
    running = true;

    uiState.setMode(new UIStandbyMode());
    std::thread uiShowThread(&UserInterface::run, &ui, std::ref(inputImage),
                           std::ref(uiState));
    std::thread imageAcquisitionThread(&CameraImageAcquisitor::run, &camera,
                                       std::ref(inputImage));

    char key = (char)(-1);

    while (running && !error) {
        // Check valid user input
        key = uiState.getKey();
        if ((key == 27) ||
            (key == 'q') ||
            (key == 'Q') ||
            (key == 'A') ||
            (key == 'D') ||
            (key == 'R') ||
            (key == 'C') ||
            (key == 'B')) {
            running = false;
        }

        std::this_thread::sleep_for(std::chrono::milliseconds(50));

        // Check for module error
        if (ui.isError() ||
            camera.isError()) {
            error = true;
            running = false;
        }
    }
    quit();

    uiShowThread.join();
    imageAcquisitionThread.join();

    if (error) {
        s->setMode(new ErrorMode());
    } else {
        switch (key) {
            case 27: s->setMode(new ClosingMode()); break;
            case 'q': s->setMode(new ClosingMode()); break;
            case 'Q': s->setMode(new ClosingMode()); break;
            case 'A': s->setMode(new AutonomousMode()); break;
            case 'D': s->setMode(new DevelopmentMode()); break;
            case 'R': s->setMode(new RemoteControlMode()); break;
            case 'C': s->setMode(new ConfigurationMode()); break;
            case 'B': s->setMode(new AboutMode()); break;
        }
    }
}
```

```
        default: s->setMode(new ErrorMode()); break;
    }
    delete this;
}

void StandbyMode::quit ()
{
    stopModules();
    running = false;
    std::cout << "MODE: Quitting standby mode..." << std::endl;
}

void StandbyMode::stopModules ()
{
    std::cout << "MODE: Quitting Standby System Mode Modules..." << std::endl;
    ui.quit();
    camera.quit();
}
```

Autonomous_mode.hpp

```

/***
 * @file autonomous_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

/***
 * @ingroup system_mode
 * @defgroup autonomous_driving_mode Autonomous Driving Mode
 * @brief A mode for the autonomous driving
 */

#ifndef AUTONOMOUS_MODE_HPP
#define AUTONOMOUS_MODE_HPP

#include <iostream>
#include "system_mode.hpp"
#include "system_state.hpp"
#include "image_data.hpp"
#include "user_interface_state.hpp"
#include "user_interface.hpp"
#include "camera_image_acquisitor.hpp"
#include "lane_data.hpp"
#include "lane_detection.hpp"
#include "traffic_sign_data.hpp"
#include "traffic_sign_detection.hpp"
#include "obstacle_data.hpp"
#include "obstacle_detection.hpp"
#include "trajectory_data.hpp"
#include "path_planning.hpp"
#include "vehicle_data.hpp"
#include "vehicle_control.hpp"

//! @addtogroup autonomous_driving_mode
//! @{
class AutonomousMode : public SystemMode {
public:
    ~AutonomousMode () = default;

    /**
     * @brief Run autonomous driving mode
     *
     * This function runs the autonomous driving mode.
     */
    void run (SystemState* s) override;

    /**
     * @brief Quit autonomous driving mode
     *
     * This function quits the autonomous driving mode.
     */
    void quit (void) override;

    /**
     * @brief Stop autonomous driving mode modules
     *
     * This function stops all autonomous driving mode modules.
     */
    void stopModules (void);

private:
    ImageData outputImage; //!< Output image data to show on user interface
    UserInterfaceState uiState; //!< User interface user input data
    UserInterface ui; //!< User interface

    ImageData inputImage; //!< Input image data from an image acquisitor
    CameraImageAcquisitor camera; //!< Camera image acquisitor

    LaneData lane; //!< Lane data of actual detected lane
    LaneDetector laneDetector; //!< Lane detector
}

```

```
TrafficSignData trafficSigns; //!< Traffic sign data of detected traffic sign
TrafficSignDetector trafficSignDetector; //!< Traffic sign detector

ObstacleData obstacles; //!< Obstacle data
ObstacleDetector obstacleDetector; //!< Obstacle detector

TrajectoryData trajectory; //!< Trajectory data
PathPlanner pathPlanner; //!< Path planner

VehicleModel vehicle; //!< Vehicle data
VehicleController vehicleController; //!< Vehicle controller
};

//! @} autonomous_driving_mode

#endif // AUTONOMOUS_MODE_HPP
```

Autonomous_mode.cpp

```
/*
 * @file autonomous_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

#include "autonomous_mode.hpp"
#include "closing_mode.hpp"
#include "error_mode.hpp"
#include "standby_mode.hpp"
#include "ui_autonomous_mode.hpp"

void AutonomousMode::run (SystemState* s)
{
    std::cout << "-----" << std::endl;
    std::cout << "MODE: Autonomous Driving Mode started" << std::endl;
    running = true;

    uiState.setMode(new UIAutonomousMode(vehicle, lane, obstacles, trafficSigns));

    std::thread uiShowThread(&UserInterface::run, &ui, std::ref(outputImage),
    std::ref(uiState));
    std::thread imageAcquisitionThread(&CameraImageAcquisitor::run, &camera,
    std::ref(inputImage));
    std::thread laneDetectonThread(&LaneDetector::run, &laneDetector, std::ref(inputImage),
    std::ref(outputImage), std::ref(lane));
    std::thread trafficSignDetectionThread(&TrafficSignDetector::run, &trafficSignDetector,
    std::ref(inputImage), std::ref(inputImage), std::ref(trafficSigns));
    std::thread obstacleDetectionThread(&ObstacleDetector::run, &obstacleDetector,
    std::ref(obstacles));
    std::thread pathPlanningThread(&PathPlanner::run, &pathPlanner, std::ref(outputImage),
    std::ref(outputImage), std::ref(lane), std::ref(trafficSigns), std::ref(obstacles) ,
    std::ref(vehicle), std::ref(trajectory));
    std::thread vehicleControlThread(&VehicleController::run, &vehicleController,
    std::ref(trajectory), std::ref(vehicle));

    char key = (char)(-1);

    while (running) {
        // Check for valid user input
        key = uiState.getKey();
        if ((key == 27) ||
            (key == 'q') ||
            (key == 'Q') ||
            (key == 'B')) {
            running = false;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(50));

        // Check for module error
        if (ui.isError() ||
            camera.isError() ||
            laneDetector.isError() ||
            trafficSignDetector.isError() ||
            obstacleDetector.isError() ||
            pathPlanner.isError() ||
            vehicleController.isError()) {
            running = false;
            error = true;
        }
    }
    quit();

    uiShowThread.join();
    imageAcquisitionThread.join();
    laneDetectonThread.join();
    trafficSignDetectionThread.join();
    obstacleDetectionThread.join();
    pathPlanningThread.join();
    vehicleControlThread.join();
}
```

```
if (error) {
    s->setMode(new StandbyMode());
}
else {
    switch (key) {
        case 27: s->setMode(new ClosingMode()); break;
        case 'q': s->setMode(new ClosingMode()); break;
        case 'Q': s->setMode(new ClosingMode()); break;
        case 'B': s->setMode(new StandbyMode()); break;
        default: s->setMode(new ErrorMode()); break;
    }
}
delete this;
}

void AutonomousMode::quit ()
{
    stopModules();
    std::cout << "MODE: Quitting autonomous system mode..." << std::endl;
    running = false;
}

void AutonomousMode::stopModules ()
{
    std::cout << "MODE: Stopping all modules..." << std::endl;
    ui.quit();
    camera.quit();
    laneDetector.quit();
    trafficSignDetector.quit();
    obstacleDetector.quit();
    pathPlanner.quit();
    vehicleController.quit();
}
```

Development_mode.hpp

```
/***
 * @file development_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

/***
 * @ingroup system_mode
 * @defgroup development_mode Development Mode
 * @brief A mode to evaluate the implemented detection functions
 */

#ifndef DEVELOPMENT_MODE_HPP
#define DEVELOPMENT_MODE_HPP

#include <iostream>
#include "system_mode.hpp"
#include "system_state.hpp"
#include "image_data.hpp"
#include "user_interface_state.hpp"
#include "user_interface.hpp"
#include "camera_image_acquisitor.hpp"
#include "lane_data.hpp"
#include "lane_detection.hpp"
#include "traffic_sign_data.hpp"
#include "traffic_sign_detection.hpp"
#include "obstacle_data.hpp"
#include "obstacle_detection.hpp"
#include "trajectory_data.hpp"
#include "remote_control.hpp"
#include "vehicle_data.hpp"
#include "vehicle_control.hpp"

///! @addtogroup development_mode
///! @{
class DevelopmentMode : public SystemMode {
public:
    ~DevelopmentMode () = default;

    /**
     * @brief Run development mode
     *
     * This function runs the development mode.
     */
    void run (SystemState* s) override;

    /**
     * @brief Quit development mode
     *
     * This function quits the development mode.
     */
    void quit (void) override;

    /**
     * @brief Stop development mode modules
     *
     * This function stops all development mode modules.
     */
    void stopModules (void);

private:
    ImageData outputImage; //!
```

`< Output image data to show on user interface
UserInterfaceState uiState; //!`

`< User interface user input data
UserInterface ui; //!`

`< Input image data from an image acquisitor
CameraImageAcquisitor camera; //!`

`< Lane data of actual detected lane
LaneDetector laneDetector; //!`

```
TrafficSignData trafficSignData; //!< Traffic sign data of detected traffic sign
TrafficSignDetector trafficSignDetector; //!< Traffic sign detector

ObstacleData obstacle; //!< Obstacle data
ObstacleDetector obstacleDetector; //!< Obstacle detector

TrajectoryData trajectory; //!< Trajectory data
RemoteController remoteController; //!< Remote controller

VehicleModel vehicle; //!< Vehicle data
VehicleController vehicleController; //!< Vehicle controller
};

//! @} development_mode

#endif // DEVELOPMENT_MODE_HPP
```

Development_mode.cpp

```
/*
 * @file development_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

#include "development_mode.hpp"
#include "closing_mode.hpp"
#include "error_mode.hpp"
#include "standby_mode.hpp"
#include "ui_development_mode.hpp"

void DevelopmentMode::run (SystemState* s)
{
    std::cout << "-----" << std::endl;
    std::cout << "MODE: Development Mode started" << std::endl;
    running = true;

    uiState.setMode(new UIDevelopmentMode(vehicle, lane, obstacle, trafficSignData));
    std::thread uiShowThread(&UserInterface::run, &ui, std::ref(outputImage),
    std::ref(uiState));
    std::thread imageAcquisitionThread(&CameraImageAcquisitor::run, &camera,
    std::ref(inputImage));
    std::thread laneDetectonThread(&LaneDetector::run, &laneDetector, std::ref(inputImage),
    std::ref(outputImage), std::ref(lane));
    std::thread trafficSignDetectionThread(&TrafficSignDetector::run, &trafficSignDetector,
    std::ref(inputImage), std::ref(inputImage), std::ref(trafficSignData));
    std::thread obstacleDetectionThread(&ObstacleDetector::run, &obstacleDetector,
    std::ref(obstacle));
    std::thread remoteControlThread(&RemoteController::run, &remoteController,
    std::ref(vehicle), std::ref(uiState));
    std::thread vehicleControlThread(&VehicleController::run, &vehicleController,
    std::ref(trajetory), std::ref(vehicle));

    char key = (char)(-1);

    while (running) {
        // Check valid user input
        key = uiState.getKey();
        if ((key == 27) ||
            (key == 'q') ||
            (key == 'Q') ||
            (key == 'B')) {
            running = false;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(50));

        // Check for module error
        if (ui.isError() ||
            camera.isError() ||
            laneDetector.isError() ||
            trafficSignDetector.isError() ||
            obstacleDetector.isError() ||
            remoteController.isError() ||
            vehicleController.isError()) {
            running = false;
            error = true;
        }
    }
    quit();

    uiShowThread.join();
    imageAcquisitionThread.join();
    laneDetectonThread.join();
    trafficSignDetectionThread.join();
    obstacleDetectionThread.join();
    remoteControlThread.join();
    vehicleControlThread.join();

    if (error) {
        s->setMode(new StandbyMode());
    }
}
```

```
    }
    else {
        switch (key) {
            case 27: s->setMode(new ClosingMode()); break;
            case 'q': s->setMode(new ClosingMode()); break;
            case 'Q': s->setMode(new ClosingMode()); break;
            case 'B': s->setMode(new StandbyMode()); break;
            default: s->setMode(new ErrorMode()); break;
        }
    }
    delete this;
}

void DevelopmentMode::quit ()
{
    stopModules();
    std::cout << "MODE: Quitting development system mode..." << std::endl;
    running = false;
}

void DevelopmentMode::stopModules ()
{
    std::cout << "MODE: Stopping all modules..." << std::endl;
    ui.quit();
    camera.quit();
    laneDetector.quit();
    trafficSignDetector.quit();
    obstacleDetector.quit();
    remoteController.quit();
    vehicleController.quit();
}
```

Remote_control_mode.hpp

```

/***
 * @file remote_control.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

/***
 * @ingroup system_mode
 * @defgroup remote_control_mode Remote Control Mode
 * @brief A mode to manually remote control the autonomous driving system
 */

#ifndef REMOTE_CONTROL_MODE_HPP
#define REMOTE_CONTROL_MODE_HPP

#include <iostream>
#include "system_mode.hpp"
#include "image_data.hpp"
#include "user_interface_state.hpp"
#include "user_interface.hpp"
#include "camera_image_acquisitor.hpp"
#include "vehicle_data.hpp"
#include "trajectory_data.hpp"
#include "remote_control.hpp"
#include "vehicle_control.hpp"

//! @addtogroup remote_control_mode
//! @{
class RemoteControlMode : public SystemMode {
public:
    ~RemoteControlMode() = default;

    /**
     * @brief Start remote control mode
     *
     * This function starts the remote control mode.
     *
     * @param s System state pointer
     */
    void run (SystemState* s) override;

    /**
     * @brief Stop remote control mode
     *
     * This function stops the remote control mode.
     */
    void quit (void) override;

    /**
     * @brief Stop autonomous driving mode modules
     *
     * This function stops all autonomous driving mode modules.
     */
    void stopModules (void);

private:
    ImageData inputImage; //!< Input image data from an image acquisitor
    ImageData outputImage; //!< Output image data from an image acquisitor
    UserInterfaceState uiState; //!< User interface user input data
    UserInterface ui; //!< User interface
    CameraImageAcquisitor camera; //!< Camera image acquisitor
    VehicleModel vehicle; //!< Vehicle data
    TrajectoryData trajectory; //!< Trajectory data
    RemoteController remoteController; //!< Remote controller
    VehicleController vehicleController; //!< Vehicle controller
};

//! @} remote_control_mode
#endif // REMOTE_CONTROL_MODE_HPP

```

Remote_control_mode.cpp

```
/*
 * @file remote_control_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

#include "remote_control_mode.hpp"
#include "closing_mode.hpp"
#include "error_mode.hpp"
#include "standby_mode.hpp"
#include "autonomous_mode.hpp"
#include "ui_remote_control_mode.hpp"

void RemoteControlMode::run (SystemState* s)
{
    std::cout << "-----" << std::endl;
    std::cout << "Remote Control System Mode started." << std::endl;
    running = true;

    uiState.setMode(new UIRemoteControlMode(vehicle));

    std::thread uiShowThread(&UserInterface::run, &ui, std::ref(inputImage),
                           std::ref(uiState));
    std::thread imageAcquisitionThread(&CameraImageAcquisitor::run, &camera,
                                       std::ref(inputImage));
    std::thread remoteControlThread(&RemoteController::run, &remoteController,
                                    std::ref(vehicle), std::ref(uiState));
    std::thread vehicleControlThread(&VehicleController::run, &vehicleController,
                                    std::ref(trajectory), std::ref(vehicle));

    char key = (char)(-1);

    while (running) {
        // Check valid user input
        key = uiState.getKey();
        if ((key == 27) ||
            (key == 'q') ||
            (key == 'Q') ||
            (key == 'B')) {
            running = false;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(50));

        // Check for module error
        if (ui.isError() ||
            camera.isError() ||
            remoteController.isError() ||
            vehicleController.isError()) {
            running = false;
            error = true;
        }
    }
    quit();

    uiShowThread.join();
    imageAcquisitionThread.join();
    remoteControlThread.join();
    vehicleControlThread.join();

    if (error) {
        s->setMode(new StandbyMode());
    }
    else {
        switch (key) {
            case 27: s->setMode(new ClosingMode()); break;
            case 'q': s->setMode(new ClosingMode()); break;
            case 'Q': s->setMode(new ClosingMode()); break;
            case 'B': s->setMode(new StandbyMode()); break;
            default: s->setMode(new ErrorMode()); break;
        }
    }
}
```

```
    delete this;
}

void RemoteControlMode::quit ()
{
    stopModules();
    std::cout << "Remote Control System Mode stopped." << std::endl;
    running = false;
}

void RemoteControlMode::stopModules ()
{
    std::cout << "MODE: Quiting Remoute Control Mode Modules..." << std::endl;
    ui.quit();
    camera.quit();
    remoteController.quit();
    vehicleController.quit();
}
```

Configuration_mode.hpp

```
/***
 * @file configuration_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

/***
 * @ingroup system_mode
 * @defgroup configuration_mode Configuration Mode
 * @brief A mode to configure the system
 */

#ifndef CONFIGURATION_MODE_HPP
#define CONFIGURATION_MODE_HPP

#include <iostream>
#include "system_mode.hpp"
#include "system_state.hpp"
#include "image_data.hpp"
#include "user_interface_state.hpp"
#include "user_interface.hpp"
#include "camera_image_acquisitor.hpp"

//! @addtogroup configuration_mode
//! @{
class ConfigurationMode : public SystemMode {
public:
    ~ConfigurationMode() = default;

    void run (SystemState* s) override;
    void quit (void) override;
    void stopModules (void);

private:
    ImageData inputImage; //!< Input image captured by an image acquisitor
    UserInterfaceState uiState; //!< User interface input data
    UserInterface ui; //!< User interface
    CameraImageAcquisitor camera; //!< Camera image acquisitor
};

//! @} configuration_mode

#endif // CONFIGURATION_MODE_HPP
```

Configuration_mode.cpp

```
/*
 * @file configuration_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

#include "configuration_mode.hpp"
#include "closing_mode.hpp"
#include "error_mode.hpp"
#include "standby_mode.hpp"
#include "calibration_mode.hpp"
#include "ui_configuration_mode.hpp"

void ConfigurationMode::run (SystemState* s)
{
    std::cout << "-----" << std::endl;
    std::cout << "MODE: Configuration Mode started." << std::endl;
    running = true;

    uiState.setMode(new UIConfigurationMode());
    std::thread uiShowThread(&UserInterface::run, &ui, std::ref(inputImage),
    std::ref(uiState));
    std::thread imageAcquisitionThread(&CameraImageAcquisitor::run, &camera,
    std::ref(inputImage));

    char key = (char)(-1);

    while (running) {
        //Check for valid user input
        key = uiState.getKey();
        if ((key == 27) ||
            (key == 'q') ||
            (key == 'Q') ||
            (key == 'B') ||
            (key == 'I') ||
            (key == 'E')) {
            running = false;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(50));

        // Check for module error
        if (ui.isError() ||
            camera.isError()) {
            running = false;
            error = true;
        }
    }
    quit();

    uiShowThread.join();
    imageAcquisitionThread.join();

    if (error) {
        s->setMode(new StandbyMode());
    }
    else {
        switch (key) {
            case 27: s->setMode(new ClosingMode()); break;
            case 'q': s->setMode(new ClosingMode()); break;
            case 'Q': s->setMode(new ClosingMode()); break;
            case 'B': s->setMode(new StandbyMode()); break;
            case 'I': s->setMode(new IntrinsicCalibrationMode()); break;
            case 'E': s->setMode(new ExtrinsicCalibrationMode()); break;
            //~ case 'P': s->setMode(new ImageAdjustmentMode()); break;
            default: s->setMode(new ErrorMode()); break;
        }
    }
    delete this;
}

void ConfigurationMode::quit ()
```

```
{  
    stopModules();  
    running = false;  
    std::cout << "MODE: Configuration Mode stopped." << std::endl;  
}  
  
void ConfigurationMode::stopModules ()  
{  
    std::cout << "MODE: Stopping all modules..." << std::endl;  
    ui.quit();  
    camera.quit();  
}
```

Calibration_mode.hpp

```
/*
 * @file calibration_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 25.4.2018
 */

/**
 * @ingroup configuration_mode
 * @defgroup calibration_mode Calibration Mode
 * @brief A mode to calibrate the camera
 */

#ifndef CALIBRATION_MODE_HPP
#define CALIBRATION_MODE_HPP

#include <iostream>
#include <thread>
#include "system_mode.hpp"
#include "system_state.hpp"
#include "user_interface_state.hpp"
#include "image_data.hpp"
#include "user_interface.hpp"
#include "camera_image_acquisitor.hpp"

//! @addtogroup calibration_mode
//! @{
class IntrinsicsCalibrationMode : public SystemMode {
public:
    ~IntrinsicsCalibrationMode() = default;

    /**
     * @brief Start intrinsics calibration system mode
     *
     * This function starts the intrinsic camera calibration mode.
     *
     * @param s System state pointer
     */
    void run (SystemState* s) override;

    /**
     * @brief Stop intrinsics calibration system mode
     *
     * This function stops the intrinsic camera calibration mode.
     */
    void quit (void) override;

    /**
     * @brief Stop mode modules
     *
     * This function stops all mode modules.
     */
    void stopModules (void);

private:
    ImageData inputImage; //!< Input image captured by an image acquisitor
    ImageData outputImage; //!< Output image captured by an image acquisitor
    UserInterfaceState uiState; //!< User interface user input data
    UserInterface ui; //!< User interface
    CameraImageAcquisitor camera; //!< Camera image acquisitor
};

class ExtrinsicsCalibrationMode : public SystemMode {
public:
    virtual ~ExtrinsicsCalibrationMode() = default;

    /**
     * @brief Start extrinsics calibration system mode
     *
     * This function starts the extrinsic camera calibration mode.
     */
}
```

```

 * @param s System state pointer
 */
void run (SystemState* s) override;

/**
 * @brief Stop extrinsics calibration system mode
 *
 * This function stops the extrinsic camera calibration mode.
 */
void quit (void) override;

/**
 * @brief Stop mode modules
 *
 * This function stops all mode modules.
 */
void stopModules (void);

private:
ImageData inputImage; //!< Input image captured by an image acquisitor
ImageData outputImage; //!< Output image captured by an image acquisitor
UserInterfaceState uiState; //!< User interface user input data
UserInterface ui; //!< User interface
CameraImageAcquisitor camera; //!< Camera image acquisitor
};

class ImageAdjustmentMode : public SystemMode {
public:
    virtual ~ImageAdjustmentMode() = default;

    /**
     * @brief Start extrinsics calibration system mode
     *
     * This function starts the extrinsic camera calibration mode.
     *
     * @param s System state pointer
     */
    void run (SystemState* s) override;

    /**
     * @brief Stop extrinsics calibration system mode
     *
     * This function stops the extrinsic camera calibration mode.
     */
    void quit (void) override;

    /**
     * @brief Stop mode modules
     *
     * This function stops all mode modules.
     */
    void stopModules (void);

private:
ImageData inputImage; //!< Input image captured by an image acquisitor
ImageData outputImage; //!< Output image captured by an image acquisitor
UserInterfaceState uiState; //!< User interface user input data
UserInterface ui; //!< User interface
CameraImageAcquisitor camera; //!< Camera image acquisitor
};

//! @} calibration_mode
#endif // CALIBRATION_MODE_HPP

```

Calibration_mode.cpp

```


/*
 * @file calibration_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 25.4.2018
 */

#include "calibration_mode.hpp"
#include "closing_mode.hpp"
#include "error_mode.hpp"
#include "standby_mode.hpp"
#include "autonomous_mode.hpp"
#include "remote_control_mode.hpp"
#include "configuration_mode.hpp"
#include "ui_calibration_mode.hpp"

void IntrinsicsCalibrationMode::run (SystemState* s)
{
    std::cout << "-----" << std::endl;
    std::cout << "MODE: Intrinsics Calibration Mode started." << std::endl;
    running = true;

    uiState.setMode(new UIIntrinsicsCalibrationMode());
    std::thread uiShowThread(&UserInterface::run, &ui, std::ref(outputImage),
                           std::ref(uiState));
    std::thread imageAcquisitionThread(&CameraImageAcquisitor::run, &camera,
                                        std::ref(inputImage));
    std::thread calibrationThread(&CameraImageAcquisitor::runIntrinsicCalibration, &camera,
                                  std::ref(inputImage), std::ref(outputImage), std::ref(uiState));

    char key = (char)(-1);

    while (running) {
        // Check for valid user input
        key = uiState.getKey();
        if ((key == 27) ||
            (key == 'q') ||
            (key == 'Q') ||
            (key == 'B')) {
            running = false;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(50));

        // Check for module error
        if (ui.isError() ||
            camera.isError()) {
            running = false;
            error = true;
        }
    }
    quit();

    uiShowThread.join();
    imageAcquisitionThread.join();
    calibrationThread.join();

    if (error) {
        s->setMode(new StandbyMode());
    }
    else {
        switch (key) {
            case 27: s->setMode(new ClosingMode()); break;
            case 'q': s->setMode(new ClosingMode()); break;
            case 'Q': s->setMode(new ClosingMode()); break;
            case 'B': s->setMode(new ConfigurationMode()); break;
            default: s->setMode(new ErrorMode()); break;
        }
    }
    delete this;
}

void IntrinsicsCalibrationMode::quit ()


```

```

{
    stopModules();
    running = false;
    std::cout << "MODE: Quitting Intrinsics Calibration Mode..." << std::endl;
}

void IntrinsicsCalibrationMode::stopModules ()
{
    std::cout << "MODE: Quitting Intrinsics Calibration System Mode Modules..." <<
std::endl;
    ui.quit();
    camera.quit();
}

void ExtrinsicsCalibrationMode::run (SystemState* s)
{
    std::cout << "-----" << std::endl;
    std::cout << "MODE: Extrinsics Calibration Mode started." << std::endl;
    running = true;

    uiState.setMode(new UIExtrinsicsCalibrationMode());
    std::thread uiShowThread(&UserInterface::run, &ui, std::ref(outputImage),
std::ref(uiState));
    std::thread imageAcquisitionThread(&CameraImageAcquisitor::run, &camera,
std::ref(inputImage));
    std::thread calibrationThread(&CameraImageAcquisitor::runExtrinsicCalibration, &camera,
std::ref(inputImage), std::ref(outputImage), std::ref(uiState));

    char key = (char)(-1);

    while (running) {
        // Check for valid user input
        key = uiState.getKey();
        if ((key == 27) ||
            (key == 'q') ||
            (key == 'Q') ||
            (key == 'B')) {
            running = false;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(50));

        // Check for module error
        if (ui.isError() ||
            camera.isError()) {
            running = false;
            error = true;
        }
    }
    quit();

    uiShowThread.join();
    imageAcquisitionThread.join();
    calibrationThread.join();

    if (error) {
        s->setMode(new StandbyMode());
    }
    else {
        switch (key) {
            case 27: s->setMode(new ClosingMode()); break;
            case 'q': s->setMode(new ClosingMode()); break;
            case 'Q': s->setMode(new ClosingMode()); break;
            case 'B': s->setMode(new ConfigurationMode()); break;
            default: s->setMode(new ErrorMode()); break;
        }
    }
    delete this;
}

void ExtrinsicsCalibrationMode::quit ()
{
    stopModules();
}

```

```

        running = false;
        std::cout << "MODE: Quitting Extrinsics Calibration Mode..." << std::endl;
    }

void ExtrinsicsCalibrationMode::stopModules ()
{
    std::cout << "MODE: Quitting Extrinsics Calibration System Mode Modules..." <<
    std::endl;
    ui.quit();
    camera.quit();
}

<**
 * @brief Image adjustment mode
 *
 * @note This function is experimental!
 */
void ImageAdjustmentMode::run (SystemState* s)
{
    std::cout << "-----" << std::endl;
    std::cout << "MODE: Image Adjustment Mode started." << std::endl;
    running = true;

    uiState.setMode(new UIImageAdjustmentMode());
    std::thread uiShowThread(&UserInterface::run, &ui, std::ref(outputImage),
    std::ref(uiState));
    std::thread imageAcquisitionThread(&CameraImageAcquisitor::run, &camera,
    std::ref(inputImage));
    std::thread calibrationThread(&CameraImageAcquisitor::runImageAdjustment, &camera,
    std::ref(inputImage), std::ref(outputImage), std::ref(uiState));

    char key = (char)(-1);

    while (running) {
        // Check for valid user input
        key = uiState.getKey();
        if ((key == 27) ||
            (key == 'q') ||
            (key == 'Q') ||
            (key == 'B')) {
            running = false;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(50));

        // Check for module error
        if (ui.isError() ||
            camera.isError()) {
            running = false;
            error = true;
        }
    }
    quit();

    uiShowThread.join();
    imageAcquisitionThread.join();
    calibrationThread.join();

    if (error) {
        s->setMode(new StandbyMode());
    }
    else {
        switch (key) {
            case 27: s->setMode(new ClosingMode()); break;
            case 'q': s->setMode(new ClosingMode()); break;
            case 'Q': s->setMode(new ClosingMode()); break;
            case 'B': s->setMode(new ConfigurationMode()); break;
            default: s->setMode(new ErrorMode()); break;
        }
    }
    delete this;
}

```

```
void ImageAdjustmentMode::quit ()
{
    stopModules();
    running = false;
    std::cout << "MODE: Quitting Image Adjustment Mode..." << std::endl;
}

void ImageAdjustmentMode::stopModules ()
{
    std::cout << "MODE: Quitting Image Adjustment System Mode Modules..." << std::endl;
    ui.quit();
    camera.quit();
}
```

About_mode.hpp

```
/***
 * @file about_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 *
 * @note This should be part of the User Interface.
 */

/***
 * @ingroup system_mode
 * @defgroup about_mode About Mode
 * @brief A mode provides system version information
 */

#ifndef ABOUT_MODE_HPP
#define ABOUT_MODE_HPP

#include <iostream>
#include "system_mode.hpp"
#include "system_state.hpp"
#include "image_data.hpp"
#include "user_interface_state.hpp"
#include "user_interface.hpp"

//! @addtogroup about_mode
//! @{
class AboutMode : public SystemMode {
public:
    ~AboutMode() = default;

    void run (SystemState* s) override;
    void quit (void) override;
    void stopModules (void);

private:
    ImageData outputImage;
    UserInterfaceState uiState;
    UserInterface ui;
};

//! @} about_mode
#endif // ABOUT_MODE_HPP
```

About_mode.cpp

```
/***
 * @file about_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 5.4.2018
 */

#include "about_mode.hpp"
#include "closing_mode.hpp"
#include "standby_mode.hpp"
#include "ui_about_mode.hpp"

void AboutMode::run (SystemState* s)
{
    std::cout << "-----" << std::endl;
    std::cout << "MODE: About Mode started." << std::endl;
    running = true;

    uiState.setMode(new UIAboutMode());
    std::thread uiShow(&UserInterface::run, &ui, std::ref(outputImage), std::ref(uiState));

    char key = (char)(-1);

    // Process user input
    while (running) {
        key = uiState.getKey();
        if ((key == 27) ||
            (key == 'q') ||
            (key == 'Q') ||
            (key == 'B')) {
            running = false;
        }
        std::this_thread::sleep_for(std::chrono::milliseconds(50));
    }
    quit();

    uiShow.join();

    switch (key) {
        case 27: s->setMode(new ClosingMode()); break;
        case 'q': s->setMode(new ClosingMode()); break;
        case 'Q': s->setMode(new ClosingMode()); break;
        case 'B': s->setMode(new StandbyMode()); break;
    }
    delete this;
}

void AboutMode::quit ()
{
    stopModules();
    std::cout << "MODE: Quitting about mode..." << std::endl;
    running = false;
}

void AboutMode::stopModules ()
{
    std::cout << "MODE: Stopping all modules..." << std::endl;
    ui.quit();
}
```

Error_mode.hpp

```
/**  
 * @file error_mode.hpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 5.4.2018  
 */  
  
/**  
 * @ingroup system_mode  
 * @defgroup error_mode Error Mode  
 * @brief A mode to catch all system errors and safely close the system  
 */  
  
#ifndef ERROR_MODE_HPP  
#define ERROR_MODE_HPP  
  
#include <iostream>  
#include "system_mode.hpp"  
#include "system_state.hpp"  
  
//! @addtogroup error_mode  
//! @{  
  
class ErrorMode : public SystemMode {  
public:  
    ~ErrorMode() = default;  
  
    void run (SystemState* s) override;  
    void quit (void) override;  
};  
  
//! @} error_mode  
  
#endif // ERROR_MODE_HPP
```

Error_mode.cpp

```
/**  
 * @file error_mode.cpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 5.4.2018  
 */  
  
#include "error_mode.hpp"  
#include "closing_mode.hpp"  
  
void ErrorMode::run (SystemState* s)  
{  
    std::cout << "-----" << std::endl;  
    std::cout << "MODE: Error Mode started." << std::endl;  
    running = true;  
  
    quit();  
    s->setMode(new ClosingMode());  
    delete this;  
}  
  
void ErrorMode::quit ()  
{  
    running = false;  
    std::cout << "MODE: Error Mode stopped." << std::endl;  
}
```

Closing_mode.hpp

```
/**  
 * @file closing_mode.hpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 5.4.2018  
 */  
  
/**  
 * @ingroup system_mode  
 * @defgroup closing_mode Closing Mode  
 * @brief A mode to close the system safely  
 */  
  
#ifndef CLOSING_MODE_HPP  
#define CLOSING_MODE_HPP  
  
#include <iostream>  
#include "system_mode.hpp"  
  
//! @addtogroup closing_mode  
//! @{  
  
class ClosingMode : public SystemMode {  
public:  
    ~ClosingMode() = default;  
  
    void run (SystemState* s) override;  
    void quit (void) override;  
};  
  
//! @} closing_mode  
  
#endif // CLOSING_MODE_HPP
```

Closing_mode.cpp

```
/**  
 * @file closing_mode.cpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 5.4.2018  
 */  
  
#include "closing_mode.hpp"  
  
void ClosingMode::run (SystemState* s)  
{  
    std::cout << "-----" << std::endl;  
    std::cout << "MODE: Closing Mode started." << std::endl;  
    running = true;  
  
    //~ resetMotorDriver();  
    quit();  
    s->quit();  
}  
  
void ClosingMode::quit ()  
{  
    running = false;  
    std::cout << "MODE: Quitting closing mode..." << std::endl;  
}
```

User Interface

User_interface.hpp

```
/*
 * @file user_interface.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 27.6.2017
 */

/**
 * @ingroup module
 * @defgroup user_interface User Interface
 * @brief A mode for the user interface
 */

#ifndef USER_INTERFACE_HPP
#define USER_INTERFACE_HPP

#include <iostream>
#include <atomic>
#include <mutex>
#include "module.hpp"
#include "camera_image_acquisitor.hpp"
#include "image_data.hpp"
#include "user_interface_state.hpp"

//! @addtogroup user_interface
//! @{
//! @brief User interface configuration class
//!
class UserInterfaceConfig {
public:
    ~UserInterfaceConfig() = default;

    bool load (cv::FileStorage fs);
    void save (cv::FileStorage fs);

public:
    std::string mainWindowName {"Autonomous Driver"}; //!< Main user interface window title
    cv::Size imageSize {640, 360}; //!< Image size of output image
    int menuWidth {160}; //!< Menu width
    double fps {30.}; //!< User interface frames per second
};

//!
 * @brief A user interface class
 */
class UserInterface : public Module {
public:
    ~UserInterface() = default;

    /**
     * @brief Run user interface thread
     *
     * This function runs the user interface thread.
     *
     * @param imageData Image data to show on user interface
     * @param uiState User interface input data
     */
    void run (ImageData& imageData, UserInterfaceState& uiState);

private:
    cv::Mat image;
    std::atomic_char inputKey {((char)(-1))};
    UserInterfaceConfig uiConfig;
    CameraConfig camConfig;
    std::mutex lock;
};

#endif
```

```
/**  
 * @brief Get user input from console  
 *  
 * This function gets the user input from the console without blocking  
 * wait  
 *  
 * @return User input key  
 */  
char getConsoleInput (void);  
  
/**  
 * @brief Draw a message on the center of an image  
 *  
 * This function draws a message on the center of an image.  
 *  
 * @param image Image matrix  
 * @param text Text string  
 */  
void drawMessage (cv::Mat& image, std::string text);  
  
//! @} user_interface  
  
#endif // USER_INTERFACE_HPP
```

User_interface.cpp

```
/*
 * @file user_interface.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 10.4.2018
 */

#include <chrono>
#include "user_interface.hpp"
#include "configuration.hpp"

#define CVUI_IMPLEMENTATION
#include "cvui.h"

void UserInterface::run (ImageData& imageData, UserInterfaceState& uiState)
{
    std::cout << "THREAD: User interface started." << std::endl;
    running = true;

    Configurator& config = Configurator::instance();
    uiConfig = config.getUserInterfaceConfig();
    camConfig = config.getCameraConfig();

    cvui::init(uiConfig.mainWindowName, (1000/uiConfig.fps)*(1.5));

    // Frame time measurement
    std::chrono::high_resolution_clock::time_point frameTimerStart, frameTimerEnd;
    int frameCnt = 0;

    while (running && !error) {
        image = imageData.read();
        if (image.empty()) {
            image = cv::Mat(uiConfig.imageSize.height, uiConfig.imageSize.width, CV_8UC3,
cv::Scalar(49, 52, 49));
            if (frameCnt == 0) {
                frameTimerStart = std::chrono::high_resolution_clock::now();
            }
        }
        else {
            if (frameCnt == 0) {
                frameTimerStart = imageData.getTime();
            }
            frameCnt++;
        }

        // Create output image composition from UI menu and output image
        cv::Mat outputImage = cv::Mat(image.rows, image.cols + uiConfig.menuWidth, CV_8UC3,
cv::Scalar(49, 52, 49));

        cvui::image(outputImage, uiConfig.menuWidth, 0, image);

        char selected = (char)(-1);
        uiState.draw(outputImage, selected);

        cvui::imshow(uiConfig.mainWindowName, outputImage);

        // Get key from console
        inputKey = getConsoleInput();
        if (inputKey != (char)(-1)) {
            uiState.setKey(inputKey);
        }
        else {
            uiState.setKey(selected);
        }

        frameTimerEnd = std::chrono::high_resolution_clock::now();
        if (std::chrono::duration_cast<std::chrono::seconds>(frameTimerEnd -
frameTimerStart).count() >= 3) {
            std::cout << "INFO: Processed FPS: " << frameCnt/3 << std::endl;
            frameCnt = 0;
        }
    }
}
```

```

        std::cout << "THREAD: User interface ended." << std::endl;
    }

    char getConsoleInput (void)
    {
        std::ios_base::sync_with_stdio(false);
        char key;
        if (std::cin.readsome(&key, 1) < 1) {
            key = (char)(-1);
        }
        std::ios_base::sync_with_stdio(true);
        return key;
    }

    void drawMessage (cv::Mat& image, std::string text)
    {
        cv::Point pt1(image.cols/2 - 200/2, image.rows/2 - 100/2);
        cv::Point pt2(image.cols/2 + 200/2, image.rows/2 + 100/2);
        rectangle(image, pt1, pt2, cv::Scalar::all(218), -1);

        int fontFace = CV_FONT_HERSHEY_DUPLEX;
        double fontScale = 1;
        int thickness = 1;
        int baseline = 0;
        cv::Size textSize = cv::getTextSize(text, fontFace, fontScale, thickness, &baseline);

        // Get center of the text
        cv::Point textOrg(pt1.x + (pt2.x - pt1.x)/2 - textSize.width/2, pt1.y + (pt2.y - pt1.y)/2 + textSize.height/2);
        putText(image, text, textOrg, fontFace, fontScale, cv::Scalar(0, 0, 255), thickness);
    }

    bool UserInterfaceConfig::load (cv::FileStorage fs)
    {
        if (!fs.isOpened()) {
            fs.release();
            std::cerr << "ERROR: File storage not opened" << std::endl;
            return false;
        }
        else {
            fs["uiMainWindowName"] >> mainWindowName;
            fs["uiImageSize_width"] >> imageSize.width;
            fs["uiImageSize_height"] >> imageSize.height;
            fs["uiMenuWidth"] >> menuWidth;
            fs["uiFPS"] >> fps;
        }
        fs.release();
        return true;
    }

    void UserInterfaceConfig::save (cv::FileStorage fs)
    {
        if (!fs.isOpened()) {
            fs.release();
            std::cerr << "ERROR: File storage not opened" << std::endl;
        }
        else {
            fs.writeComment("user interface config");
            fs << "uiMainWindowName" << mainWindowName
            << "uiImageSize_width" << imageSize.width
            << "uiImageSize_height" << imageSize.height
            << "uiMenuWidth" << menuWidth
            << "uiFPS" << fps;
        }
        fs.release();
    }
}

```

User_interface_state.hpp

```
/*
 * @file user_interface_state.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 25.4.2018
 */

/**
 * @defgroup user_interface_state User Interface State
 * @brief User Interface State
 */

#ifndef USER_INTERFACE_STATE_HPP
#define USER_INTERFACE_STATE_HPP

#include <iostream>
#include <atomic>
#include <mutex>
#include "user_interface_mode.hpp"

//! @addtogroup user_interface_state
//! @{
class UserInterfaceState {
public:
    ~UserInterfaceState() = default;

    /**
     * @brief Set user input key
     *
     * @param k User input key
     */
    void setKey (char k);

    /**
     * @brief Get user input key
     *
     * @return User input key
     */
    char getKey (void);

    /**
     * @brief Set user interface mode
     *
     * @param m User interface mode
     */
    void setMode (UserInterfaceMode* m);

    /**
     * @brief Get user interface mode
     *
     * @return User interface mode
     */
    UserInterfaceMode* getMode (void);

    /**
     * @brief Draw user interface
     *
     * This function draws the user interface.
     *
     * @param image Image matrix
     * @param selected Selected button
     */
    void draw (cv::Mat& image, char& selected);

private:
    std::atomic_char key {('c')(-1)}; //!< Actual user input key
    UserInterfaceMode* mode; //!< Actual user interface mode pointer
    std::mutex lock; //!< Mutex lock for synchronized access
};

//! @} user_interface_state
#endif // USER_INTERFACE_STATE_HPP
```

User_interface_state.cpp

```
/**  
 * @file user_interface_state.cpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 25.4.2018  
 */  
  
#include "user_interface_state.hpp"  
  
void UserInterfaceState::setKey (char k)  
{  
    key = k;  
}  
  
char UserInterfaceState::getKey (void)  
{  
    return key;  
}  
  
void UserInterfaceState::setMode (UserInterfaceMode* m)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    mode = m;  
}  
  
UserInterfaceMode* UserInterfaceState::getMode (void)  
{  
    std::lock_guard<std::mutex> guard(lock);  
    return mode;  
}  
  
void UserInterfaceState::draw (cv::Mat& image, char& selected)  
{  
    mode->draw(image, selected);  
}
```

User_interface_mode.hpp

```
/**  
 * @file user_interface_mode.hpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 1.5.2018  
 */  
  
/**  
 * @ingroup user_interface_state  
 * @defgroup user_interface_mode User Interface Mode  
 * @brief User Interface Modes to control the system states  
 */  
  
#ifndef USER_INTERFACE_MODE_HPP  
#define USER_INTERFACE_MODE_HPP  
  
#include <opencv2/opencv.hpp>  
  
//! @addtogroup user_interface_mode  
//! @{  
  
class UserInterfaceMode {  
public:  
    virtual ~UserInterfaceMode () = default;  
  
    /**  
     * @brief Draw user interface mode  
     *  
     * This function draws the user interface mode.  
     *  
     * @param image Image matrix  
     * @param selected Pressed button  
     */  
    virtual void draw (cv::Mat& image, char& selected) = 0;  
};  
  
//! @} user_interface_mode  
#endif // USER_INTERFACE_MODE_HPP
```

```
Ui_standby_mode.hpp
<**
 * @file ui_standby_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 1.5.2018
 */

#ifndef UI_STANDBY_MODE_HPP
#define UI_STANDBY_MODE_HPP

#include <opencv2/opencv.hpp>
#include "user_interface_mode.hpp"

//! @addtogroup user_interface_mode
//! @{

<**
 * @brief A user interface for the Standby Mode
 *
 * This user interface shows standby mode. It shows the actual captured
 * camera image and a system navigation menu.
 */
class UIStandbyMode : public UserInterfaceMode {
public:
    UIStandbyMode () {};
    ~UIStandbyMode () = default;
    /**
     * @brief Draw standby user interface mode
     *
     * This function draws the standby user interface mode.
     *
     * @param image Image matrix
     * @param selected Selected button
     */
    void draw (cv::Mat& image, char& selected) override;
};

//! @} user_interface_mode
#endif // UI_STANDBY_MODE_HPP
```

Ui_standby_mode.cpp

```
/**  
 * @file ui_standby_mode.hpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 1.5.2018  
 */  
  
#include "ui_standby_mode.hpp"  
#include "cvui.h"  
  
cv::Mat quitButton (int width, int height);  
  
void UIStandbyMode::draw (cv::Mat& image, char& selected)  
{  
    bool buttonPressed = false;  
  
    cvui::beginColumn(image, 10, 10, -1, -1, 5);  
    cvui::text("Main Menu", 0.6);  
    cvui::space(5);  
  
    if (cvui::button(140, 40, "&Autonomous")) {  
        selected = 'A';  
        buttonPressed = true;  
    }  
    if (cvui::button(140, 40, "&Development")) {  
        selected = 'D';  
        buttonPressed = true;  
    }  
    if (cvui::button(140, 40, "&Remote Control")) {  
        selected = 'R';  
        buttonPressed = true;  
    }  
    if (cvui::button(140, 40, "&Configuration")) {  
        selected = 'C';  
        buttonPressed = true;  
    }  
    if (cvui::button(140, 40, "A&bout")) {  
        selected = 'B';  
        buttonPressed = true;  
    }  
    if (cvui::button(140, 40, "&Quit")) {  
        selected = 'Q';  
        buttonPressed = true;  
    }  
    cvui::endColumn();  
  
    if (!buttonPressed) {  
        selected = (char)(-1);  
    }  
}
```

Ui_autonomous_mode.hpp

```
/*
 * @file ui_autonomous_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 1.5.2018
 */

#ifndef UI_AUTONOMOUS_MODE_HPP
#define UI_AUTONOMOUS_MODE_HPP

#include <opencv2/opencv.hpp>
#include "user_interface_mode.hpp"
#include "vehicle_data.hpp"
#include "lane_data.hpp"
#include "obstacle_data.hpp"
#include "traffic_sign_data.hpp"

//! @addtogroup user_interface_mode
//! @{

/**
 * @brief A user interface for the Autonomous Driving Mode
 *
 * This user interface shows the IPT camera image, the vehicle telemetry
 * and the detected lines, obstacle distance and traffic sign position.
 */
class UIAutonomousMode : public UserInterfaceMode {
public:
    UIAutonomousMode (VehicleModel& v, LaneData& l, ObstacleData& o, TrafficSignData& t) :
        vehicle(v), lane(l), obstacle(o), trafficSign(t) {};
    ~UIAutonomousMode () = default;

    /**
     * @brief Draw autonomous driving user interface mode
     *
     * This function draws the autonomous driving user interface mode.
     *
     * @param image Image matrix
     * @param selected Selected button
     */
    void draw (cv::Mat& image, char& selected) override;

private:
    VehicleModel& vehicle;
    LaneData& lane;
    ObstacleData& obstacle;
    TrafficSignData& trafficSign;
};

//! @} user_interface_mode

#endif // UI_AUTONOMOUS_MODE_HPP
```

Ui_autonomous_mode.cpp

```


/*
 * @file ui_autonomous_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 1.5.2018
 */

#include "ui_autonomous_mode.hpp"
#include "cvui.h"

void UIAutonomousMode::draw (cv::Mat& image, char& selected)
{
    cvui::beginColumn(image, 10, 10, -1, -1, 5);
    cvui::text("Auto. Mode", 0.6);
    cvui::space(5);
    if (cvui::button(140, 40, "&Back")) {
        selected = 'B';
    }
    if (cvui::button(140, 40, "&Quit")) {
        selected = 'Q';
    }
    cvui::space(5);

    cvui::text("Telemetry", 0.5);
    cvui::printf("Steering: %0.2f rad", vehicle.getSteering());
    cvui::printf("Acceleration: %0.2f %", vehicle.getAcceleration());
    cvui::space(5);

    cvui::text("Detection", 0.5);
    cvui::text("Lane");
    cv::Vec4i leftLine, rightLine;
    leftLine = lane.getLeftLine();
    cvui::printf("L: [%d, %d], [%d, %d]", leftLine[0], leftLine[1], leftLine[2],
leftLine[3]);
    rightLine = lane.getRightLine();
    cvui::printf("R: [%d, %d], [%d, %d]", rightLine[0], rightLine[1], rightLine[2],
rightLine[3]);
    cvui::space(3);

    double oD = obstacle.getDistance();
    if (oD > (-1)) {
        cvui::printf("Obstacle: %0.2f cm", oD);
    }
    else {
        cvui::text("Obstacle: No obstacle");
    }
    cvui::space(3);

    if (trafficSign.getDistance() > (-1)) {
        cv::Point stopSignCenter(getSignCenter(trafficSign.getRoi()));
        cvui::printf("Traffic Sign: [%d, %d]", stopSignCenter.x, stopSignCenter.y);
    }
    else {
        cvui::text("Traffic Sign: No traffic sign");
    }

    cvui::endColumn();
}


```

Ui_development_mode.hpp

```


/***
 * @file ui_development_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 1.5.2018
 */

#ifndef UI_AUTONOMOUS_MODE_HPP
#define UI_AUTONOMOUS_MODE_HPP

#include <opencv2/opencv.hpp>
#include "user_interface_mode.hpp"
#include "vehicle_data.hpp"
#include "lane_data.hpp"
#include "obstacle_data.hpp"
#include "traffic_sign_data.hpp"

//! @addtogroup user_interface_mode
//! @{

/***
 * @brief A user interface for the Development Mode
 *
 * This user interface shows development mode. It shows the same as the
 * autonomous driving mode, but offers on-screen controls for remote
 * controlling the system.
 */
class UIDevelopmentMode : public UserInterfaceMode {
public:
    UIDevelopmentMode (VehicleModel& v, LaneData& l, ObstacleData& o, TrafficSignData& t) :
    vehicle(v), lane(l), obstacle(o), trafficSign(t) {};
    ~UIDevelopmentMode () = default;

    /**
     * @brief Draw development driving user interface mode
     *
     * This function draws the development driving user interface mode.
     *
     * @param image Image matrix
     * @param selected Selected button
     */
    void draw (cv::Mat& image, char& selected) override;

private:
    VehicleModel& vehicle;
    LaneData& lane;
    ObstacleData& obstacle;
    TrafficSignData& trafficSign;
};

//! @} user_interface_mode

#endif // UI_AUTONOMOUS_MODE_HPP


```

Ui_development_mode.cpp

```


/*
 * @file ui_development_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 1.5.2018
 */

#include "ui_development_mode.hpp"
#include "cvui.h"

void UIDevelopmentMode::draw (cv::Mat& image, char& selected)
{
    cvui::beginColumn(image, 10, 10, -1, -1, 5);
    cvui::text("Dev. Mode", 0.6);
    cvui::space(5);
    if (cvui::button(140, 40, "&Back")) {
        selected = 'B';
    }
    if (cvui::button(140, 40, "&Quit")) {
        selected = 'Q';
    }
    cvui::space(5);

    cvui::text("Telemetry", 0.5);
    cvui::printf("Steering: %0.2f rad", vehicle.getSteering());
    cvui::printf("Acceleration: %0.2f %", vehicle.getAcceleration());
    cvui::space(5);

    cvui::text("Detection", 0.5);
    cvui::text("Lane");
    cv::Vec4i leftLine, rightLine;
    leftLine = lane.getLeftLine();
    cvui::printf("L: [%d, %d], [%d, %d]", leftLine[0], leftLine[1], leftLine[2],
leftLine[3]);
    rightLine = lane.getRightLine();
    cvui::printf("R: [%d, %d], [%d, %d]", rightLine[0], rightLine[1], rightLine[2],
rightLine[3]);
    cvui::space(3);

    if (obstacle.getDistance() > (-1)) {
        cvui::printf("Obstacle: %0.2f cm", obstacle.getDistance());
    }
    else {
        cvui::text("Obstacle: No obstacle.");
    }
    cvui::space(3);

    if (trafficSign.getDistance() > (-1)) {
        cv::Point stopSignCenter(getSignCenter(trafficSign.getRoi()));
        cvui::printf("Traffic Sign: [%d, %d]", stopSignCenter.x, stopSignCenter.y);
    }
    else {
        cvui::text("Traffic Sign: No traffic sign");
    }

    cvui::endColumn();

    // Draw vehicle control buttons
    cvui::beginRow(image, (image.cols - 100), (image.rows - 150), -1, -1, 10);
    if (cvui::button(40, 40, "&W")) {
        selected = 'w';
    }
    cvui::endRow();

    cvui::beginRow(image, (image.cols - 150), (image.rows - 100), -1, -1, 10);
    if (cvui::button(40, 40, "&A")) {
        selected = 'a';
    }
    if (cvui::button(40, 40, "&S")) {
        selected = 's';
    }
    if (cvui::button(40, 40, "&D")) {


```

```
    selected = 'd';
}
cvui::endRow();

cvui::beginRow(image, (image.cols - 150), (image.rows - 50), -1, -1, 10);
if (cvui::button(140, 40, "& space")) {
    selected = ' ';
}
cvui::endRow();
}
```

Ui_remote_control_mode.hpp

```
/**  
 * @file ui_remote_control_mode.hpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 1.5.2018  
 */  
  
#ifndef UI_REMOTE_CONTROL_MODE_HPP  
#define UI_REMOTE_CONTROL_MODE_HPP  
  
#include <opencv2/opencv.hpp>  
#include "user_interface_mode.hpp"  
#include "vehicle_data.hpp"  
  
//! @addtogroup user_interface_mode  
//! @{  
  
/**  
 * @brief A user interface for the Remote Control Mode  
 *  
 * This user interface shows the remote control mode. It shows the actual  
 * camera image and on-screen vehicle control options to remote control  
 * the system.  
 */  
class UIRemoteControlMode : public UserInterfaceMode {  
public:  
    UIRemoteControlMode (VehicleModel& v) : vehicle(v) {};  
    ~UIRemoteControlMode () = default;  
  
    /**  
     * @brief Draw remote control user interface mode  
     *  
     * This function draws the remote control user interface mode.  
     *  
     * @param image Image matrix  
     * @param selected Selected button  
     */  
    void draw (cv::Mat& image, char& selected) override;  
  
private:  
    VehicleModel& vehicle;  
};  
  
//! @} user_interface_mode  
#endif // UI_REMOTE_CONTROL_MODE_HPP
```

Ui_remote_control_mode.cpp

```
/*
 * @file ui_remote_control_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 1.5.2018
 */

#include "ui_remote_control_mode.hpp"
#include "cvui.h"

void UIRemoteControlMode::draw (cv::Mat& image, char& selected)
{
    cvui::beginColumn(image, 10, 10, -1, -1, 5);
    cvui::text("RC Mode", 0.6);
    cvui::space(5);
    if (cvui::button(140, 40, "&Back")) {
        selected = 'B';
    }
    if (cvui::button(140, 40, "&Quit")) {
        selected = 'Q';
    }
    cvui::space(5);

    cvui::text("Telemetry", 0.5);
    cvui::printf("Steering: %0.2f rad", vehicle.getSteering());
    cvui::printf("Acceleration: %0.2f %", vehicle.getAcceleration());
    cvui::space(5);

    cvui::text("Vehicle Control", 0.5);
    cvui::beginRow();
    cvui::beginColumn();
        cvui::beginColumn();
            cvui::text("Accelerate");
            cvui::space();
            cvui::text("Decelerate");
            cvui::space();
            cvui::text("Left");
            cvui::space();
            cvui::text("Right");
            cvui::space();
            cvui::text("Brake");
        cvui::endColumn();
        cvui::space();
        cvui::beginColumn();
            cvui::text("[W]");
            cvui::space();
            cvui::text("[S]");
            cvui::space();
            cvui::text("[A]");
            cvui::space();
            cvui::text("[D]");
            cvui::space();

            cvui::text("[space]");
        cvui::endColumn();
    cvui::endRow();
    cvui::endColumn();

    // Draw vehicle control buttons
    cvui::beginRow(image, (image.cols - 100), (image.rows - 150), -1, -1, 10);
    if (cvui::button(40, 40, "&W")) {
        selected = 'w';
    }
    cvui::endRow();

    cvui::beginRow(image, (image.cols - 150), (image.rows - 100), -1, -1, 10);
    if (cvui::button(40, 40, "&A")) {
        selected = 'a';
    }
    if (cvui::button(40, 40, "&S")) {
        selected = 's';
    }
    if (cvui::button(40, 40, "&D")) {

```

```
    selected = 'd';
}
cvui::endRow();

cvui::beginRow(image, (image.cols - 150), (image.rows - 50), -1, -1, 10);
if (cvui::button(140, 40, "& space")) {
    selected = ' ';
}
cvui::endRow();
}
```

Ui_configuration_mode.hpp

```
/**  
 * @file ui_configuration_mode.hpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 1.5.2018  
 */  
  
#ifndef UI_CONFIGURATION_MODE_HPP  
#define UI_CONFIGURATION_MODE_HPP  
  
#include <opencv2/opencv.hpp>  
#include "user_interface_mode.hpp"  
  
//! @addtogroup user_interface_mode  
//! @{  
  
/**  
 * @brief A user interface for the Configuration Mode  
 *  
 * This user interface shows configuration mode.  
 */  
class UIConfigurationMode : public UserInterfaceMode {  
public:  
    UIConfigurationMode() {};  
    ~UIConfigurationMode() = default;  
  
    /**  
     * @brief Draw configuration user interface mode  
     *  
     * This function draws the configuration user interface mode.  
     *  
     * @param image Image matrix  
     * @param selected Selected button  
     */  
    void draw (cv::Mat& image, char& selected) override;  
};  
  
//! @} user_interface_mode  
#endif // UI_CONFIGURATION_MODE_HPP
```

Ui_configuration_mode.cpp

```
/*
 * @file ui_configuration_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 1.5.2018
 */

#include "ui_configuration_mode.hpp"
#include "cvui.h"

void UIConfigurationMode::draw (cv::Mat& image, char& selected)
{
    cvui::beginColumn(image, 10, 10, -1, -1, 5);
    cvui::text("Config. Mode", 0.6);
    cvui::space(5);
    if (cvui::button(140, 40, "&Back")) {
        selected = 'B';
    }
    if (cvui::button(140, 40, "&Quit")) {
        selected = 'Q';
    }
    cvui::space(5);

    if (cvui::button(140, 40, "&Intrinsics")) {
        selected = 'I';
    }
    if (cvui::button(140, 40, "&Extrinsics")) {
        selected = 'E';
    }
    cvui::endColumn();
}
```

Ui_calibration_mode.hpp

```
/*
 * @file ui_calibration_mode.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 1.5.2018
 */

#ifndef UI_CALIBRATION_MODE_HPP
#define UI_CALIBRATION_MODE_HPP

#include <opencv2/opencv.hpp>
#include "user_interface_mode.hpp"

//! @addtogroup user_interface_mode
//! @{

/***
 * @brief A user interface for the Intrinsics Calibration Mode
 *
 * This user interface shows intrinsics calibration.
 */
class UIIntrinsicsCalibrationMode : public UserInterfaceMode {
public:
    UIIntrinsicsCalibrationMode() {};
    ~UIIntrinsicsCalibrationMode() = default;

    /**
     * @brief Draw intrinsics calibration user interface mode
     *
     * This function draws the intrinsics calibration user interface mode.
     *
     * @param image Image matrix
     * @param selected Selected button
     */
    void draw (cv::Mat& image, char& selected) override;
};

/***
 * @brief A user interface for the Extrinsic Calibration Mode
 *
 * This user interface shows extrinsics calibration.
 */
class UIExtrinsicsCalibrationMode : public UserInterfaceMode {
public:
    UIExtrinsicsCalibrationMode() {};
    ~UIExtrinsicsCalibrationMode() = default;

    /**
     * @brief Draw extrinsics calibration user interface mode
     *
     * This function draws the extrinsics calibration user interface mode.
     *
     * @param image Image matrix
     */
    void draw (cv::Mat& image, char& selected) override;
};

/***
 * @brief A user interface for the Image Adjustment Mode
 *
 * This user interface shows image adjustment mode.
 */
class UIImageAdjustmentMode : public UserInterfaceMode {
public:
    UIImageAdjustmentMode() {};
    ~UIImageAdjustmentMode() = default;

    /**
     * @brief Draw image position adjustment user interface mode
     *
     * This function draws the image position adjustment user interface mode.
     */
}
```

```
* @brief image Image matrix
*/
void draw (cv::Mat& image, char& selected) override;
};

//! @} user_interface_mode
#endif // UI_CALIBRATION_MODE_HPP
```

Ui_calibration_mode.cpp

```


/*
 * @file ui_calibration_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 1.5.2018
 */

#include "ui_calibration_mode.hpp"
#include "cvui.h"

void UIIntrinsicsCalibrationMode::draw (cv::Mat& image, char& selected)
{
    cvui::beginColumn(image, 10, 10, -1, -1, 5);
    cvui::text("Intr. Calib. Mode", 0.6);
    cvui::space(5);
    if (cvui::button(140, 40, "&Back")) {
        selected = 'B';
    }
    if (cvui::button(140, 40, "&Quit")) {
        selected = 'Q';
    }
    cvui::space(5);

    if (cvui::button(140, 40, "&Reset")) {
        selected = 'R';
    }
    if (cvui::button(140, 40, "&Save")) {
        selected = 'S';
    }
    cvui::endColumn();
}

void UIExtrinsicsCalibrationMode::draw (cv::Mat& image, char& selected)
{
    cvui::beginColumn(image, 10, 10, -1, -1, 5);
    cvui::text("Extr. Calib. Mode", 0.6);
    cvui::space(5);
    if (cvui::button(140, 40, "&Back")) {
        selected = 'B';
    }
    if (cvui::button(140, 40, "&Quit")) {
        selected = 'Q';
    }
    cvui::space(5);

    if (cvui::button(140, 40, "&Reset")) {
        selected = 'R';
    }
    if (cvui::button(140, 40, "&Save")) {
        selected = 'S';
    }
    cvui::endColumn();
}

void UIImageAdjustmentMode::draw (cv::Mat& image, char& selected)
{
    cvui::beginColumn(image, 10, 10, -1, -1, 5);
    cvui::text("Image Adjust. Mode", 0.6);
    cvui::space(5);
    if (cvui::button(140, 40, "&Back")) {
        selected = 'B';
    }
    if (cvui::button(140, 40, "&Quit")) {
        selected = 'Q';
    }
    cvui::space(5);

    if (cvui::button(140, 40, "&Reset")) {
        selected = 'R';
    }
    if (cvui::button(140, 40, "&Save")) {
        selected = 'S';
    }
}


```

```
    }
    cvui::endColumn();
}
```

Ui_about_mode.hpp

```
/**  
 * @file ui_about_mode.hpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 1.5.2018  
 */  
  
#ifndef UI_ABOUT_MODE_HPP  
#define UI_ABOUT_MODE_HPP  
  
#include <opencv2/opencv.hpp>  
#include "user_interface_mode.hpp"  
  
//! @addtogroup user_interface_mode  
//! @{  
  
/**  
 * @brief A user interface for the About Mode  
 *  
 * This user interface shows the actual system version informations.  
 */  
class UIAboutMode : public UserInterfaceMode {  
public:  
    UIAboutMode () {};  
    ~UIAboutMode () = default;  
  
    /**  
     * @brief Draw about user interface mode  
     *  
     * This function draws the about user interface mode.  
     *  
     * @param image Image matrix  
     * @param selected Selected button  
     */  
    void draw (cv::Mat& image, char& selected) override;  
};  
  
//! @} user_interface_mode  
#endif // UI_ABOUT_MODE_HPP
```

Ui_about_mode.cpp

```
/*
 * @file ui_about_mode.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 1.5.2018
 */

#include "ui_about_mode.hpp"
#include "cvui.h"

void UIAboutMode::draw (cv::Mat& image, char& selected)
{
    cvui::beginColumn(image, 10, 10, -1, -1, 5);
    cvui::text("About", 0.6);
    cvui::space(5);

    if (cvui::button(140, 40, "&Back")) {
        selected = 'B';
    }
    if (cvui::button(140, 40, "&Quit")) {
        selected = 'Q';
    }
    cvui::endColumn();

    cvui::beginColumn(image, 200, 30, -1, -1, 3);
    cvui::text("Master Thesis Project");
    cvui::text("Autonomous Driver", 0.6);
    cvui::text("An Autonomous Driving Development Platform");
    cvui::text("by Sergiu-Petru Tabacariu");
    cvui::text("<sergiu-petru.tabacariu@stud.fh-campuswien.ac.at>");
    cvui::text("Date 1.5.2018");
    cvui::text("Version 1.0.0");
    cvui::space(5);
    cvui::text("Get latest version at:");
    cvui::text("https://github.com/stabacariu/autonomous_driver");
    cvui::endColumn();
}
```

Ui_error_mode.hpp

```
/**  
 * @file ui_error_mode.hpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 1.5.2018  
 */  
  
#ifndef UI_ERROR_MODE_HPP  
#define UI_ERROR_MODE_HPP  
  
#include <opencv2/opencv.hpp>  
#include "user_interface_mode.hpp"  
  
//! @addtogroup user_interface_mode  
//! @{  
  
class UIErrorMode : public UserInterfaceMode {  
public:  
    UIErrorMode() {};  
    ~UIErrorMode() = default;  
  
    /**  
     * @brief Draw error user interface mode  
     *  
     * This function draws the error user interface mode.  
     *  
     * @param image Image matrix  
     * @param selected Selected button  
     */  
    void draw (cv::Mat& image, char& selected) override;  
private:  
    std::string message {"Error found. Closing system safely."};  
};  
  
//! @} user_interface_mode  
#endif // UI_ERROR_MODE_HPP
```

Ui_error_mode.cpp

```
/**  
 * @file ui_error_mode.cpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 1.5.2018  
 */  
  
#include "ui_error_mode.hpp"  
#include "cvui.h"  
  
void UIErrorMode::draw (cv::Mat& image, char& selected)  
{  
    cvui::text(image, image.cols/2, image.rows/2, message);  
}
```

Tools

Configuration.hpp

```
/***
 * @file configuration.hpp
 * @author Sergiu-Petru Tabacariu
 * @date 18.09.2017
 */

#ifndef CONFIGURATION_HPP
#define CONFIGURATION_HPP

#include <iostream>
#include <opencv2/opencv.hpp>
#include <mutex>
#include "camera_image_acquisitor.hpp"
#include "user_interface.hpp"
#include "traffic_sign_detection.hpp"
#include "obstacle_detection.hpp"

///! @addtogroup configuration
///! @{

/***
 * @brief A class for the configuration manager
 *
 * This class holds all configuration data. It synchronises access via
 * a mutex lock to be thread safe.
 */
class Configurator {
public:
    Configurator (const std::string file = "../config/default.xml") : fileName(file) {};
    ~Configurator () = default;

    /**
     * @brief Configurator instance as global singleton
     *
     * This static function describes a singleton for global accessibility.
     *
     * @param file Path to configuration file
     */
    static Configurator& instance (const std::string file = "../config/default.xml");

    /**
     * @brief Save configuration
     *
     * This function saves all configuration data.
     */
    void save (void);

    /**
     * @brief Load configuration
     *
     * This function loads all configuration data.
     */
    void load (void);

    /**
     * @brief Set camera configuration
     *
     * This function sets the camera configuration.
     *
     * @param c Camera configuration data
     */
    void setCameraConfig (CameraConfig c);

    /**
     * @brief Get camera configuration
     *
     * This function gets the camera configuration.
     */
}
```

```
* @return Camera configuration data
*/
CameraConfig getCameraConfig (void);

/** 
 * @brief Save camera configuration to XML-file
 *
 * This function saves the camera configuration data to a XML-file.
 */
void saveCameraConfig (void);

/** 
 * @brief Load camera configuration from XML-file
 *
 * This function loads the camera conrfiguration data from a XML-file.
 */
void loadCameraConfig (void);

void setCameraCalibrationConfig (CameraCalibrationConfig c);
CameraCalibrationConfig getCameraCalibrationConfig (void);

void setUserInterfaceConfig (UserInterfaceConfig c);
UserInterfaceConfig getUserInterfaceConfig (void);

void setTrafficSignDetectionConfig (TrafficSignDetectionConfig c);
TrafficSignDetectionConfig getTrafficSignDetectionConfig (void);

void setObstacleDetectionConfig (ObstacleDetectionConfig c);
ObstacleDetectionConfig getObstacleDetectionConfig (void);

private:
    std::string fileName {"../config/default.xml"}; //!< Default config file name
    CameraConfig camConfig; //!< Camera configuration data
    CameraCalibrationConfig camCalibConfig; //!< Camera calibration configuration data
    UserInterfaceConfig uiConfig; //!< User interface configuration data
    TrafficSignDetectionConfig trafficSignDetConfig; //!< Traffic sign detection
    configuration data
    ObstacleDetectionConfig obstacleDetConfig; //!< Obstacle detection configuration data
    std::mutex lock; //!< Mutex lock for synchronized access
};

//! @} configuration

#endif
```

Configuration.cpp

```
/*
 * @file configuration.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 18.09.2017
 */

#include "configuration.hpp"

Configurator& Configurator::instance (const std::string file)
{
    static Configurator instance(file);
    return instance;
}

void Configurator::save (void)
{
    cv::FileStorage fs(fileName, cv::FileStorage::WRITE);

    camConfig.save(fs);
    camCalibConfig.save(fs);
    uiConfig.save(fs);
    trafficSignDetConfig.save(fs);
    obstacleDetConfig.save(fs);
}

void Configurator::load (void)
{
    cv::FileStorage fs(fileName, cv::FileStorage::READ);
    if (!fs.isOpened()) {
        std::cerr << "ERROR: No file " << fileName << " found!" << std::endl;
        fs.release();

        // Try open default.xml
        fs.open("../config/default.xml", cv::FileStorage::READ);
        if (!fs.isOpened()) {
            std::cerr << "Error: No file ../config/default.xml found!" << std::endl;
        }
        else {
            fileName = "../config/default.xml";
            camConfig.load(fs);
            camCalibConfig.load(fs);
            uiConfig.load(fs);
            trafficSignDetConfig.load(fs);
            obstacleDetConfig.load(fs);
            fs.release();
        }
    }
    else {
        camConfig.load(fs);
        camCalibConfig.load(fs);
        uiConfig.load(fs);
        trafficSignDetConfig.load(fs);
        obstacleDetConfig.load(fs);
        fs.release();
    }
}

void Configurator::setCameraConfig (CameraConfig c)
{
    std::lock_guard<std::mutex> guard(lock);
    camConfig = c;
}

CameraConfig Configurator::getCameraConfig (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return camConfig;
}

void Configurator::setCameraCalibrationConfig (CameraCalibrationConfig c)
{
```

```
    std::lock_guard<std::mutex> guard(lock);
    camCalibConfig = c;
}

CameraCalibrationConfig Configurator::getCameraCalibrationConfig (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return camCalibConfig;
}

void Configurator::setUserInterfaceConfig (UserInterfaceConfig c)
{
    std::lock_guard<std::mutex> guard(lock);
    uiConfig = c;
}

UserInterfaceConfig Configurator::getUserInterfaceConfig (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return uiConfig;
}

void Configurator::setTrafficSignDetectionConfig (TrafficSignDetectionConfig c)
{
    std::lock_guard<std::mutex> guard(lock);
    trafficSignDetConfig = c;
}

TrafficSignDetectionConfig Configurator::getTrafficSignDetectionConfig (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return trafficSignDetConfig;
}

void Configurator::setObstacleDetectionConfig (ObstacleDetectionConfig c)
{
    std::lock_guard<std::mutex> guard(lock);
    obstacleDetConfig = c;
}

ObstacleDetectionConfig Configurator::getObstacleDetectionConfig (void)
{
    std::lock_guard<std::mutex> guard(lock);
    return obstacleDetConfig;
}
```

Image_filter.hpp

```
/**  
 * @file image_filter.hpp  
 * @author Sergiu-Petru Tabacariu  
 * @date 30.6.2017  
 */  
  
#ifndef IMAGE_FILTER_HPP  
#define IMAGE_FILTER_HPP  
  
#include <iostream>  
#include <opencv2/opencv.hpp>  
  
/**  
 * @brief A function to auto adjust image brightness  
 *  
 * This function adjusts the image brightness by spreading intensity levels.  
 *  
 * @param image  
 */  
void autoAdjustBrightness (cv::Mat& image);  
  
/**  
 * @brief A function to filter white "color"  
 *  
 * This function filters white "color" out of an RGB image.  
 *  
 * @param image  
 * @param filteredImage  
 */  
void whiteColorFilter (cv::Mat image, cv::Mat& filteredImage);  
  
/**  
 * @brief A function to filter yellow color  
 *  
 * This function filters yellow color. It converts a RGB image to HSV  
 * color space.  
 *  
 * @param image  
 * @param filteredImage  
 */  
void yellowColorFilter (cv::Mat image, cv::Mat& filteredImage);  
  
#endif // IMAGE_FILTER_HPP
```

Image_filter.cpp

```
/*
 * @file image_filter.cpp
 * @author Sergiu-Petru Tabacariu
 * @date 30.6.2017
 */

#include "image_filter.hpp"

void autoAdjustBrightness (cv::Mat& image)
{
    double minValue, maxValue;
    minMaxLoc(image, &minValue, &maxValue);
    double range = maxValue - minValue;
    double targetRange = 255;
    double alpha = targetRange/range;
    double beta = (-1) * minValue * alpha;
    image.convertTo(image, -1, alpha, beta);

    //~ cv::Mat yuvImage;
    //~ cvtColor(image, yuvImage, CV_BGR2YUV);
    //~ vector<cv::Mat> yuvChannels;
    //~ split(yuvImage, yuvChannels);
    //~ equalizeHist(yuvChannels[0], yuvChannels[0]);
    //~ merge(yuvChannels, yuvImage);
    //~ cvtColor(yuvImage, image, CV_YUV2BGR);
}

void whiteColorFilter (cv::Mat image, cv::Mat& filteredImage)
{
    //~ cv::Mat imageHLS;
    //~ cvtColor(image, imageHLS, CV_BGR2HLS);
    //~ inRange(imageHLS, cv::Scalar(175, 0, 0), cv::Scalar(180, 255, 255), filteredImage);
    //~ cvtColor(filteredImage, filteredImage, CV_HLS2BGR);
    cv::Mat grayImage;
    cvtColor(image, grayImage, CV_BGR2GRAY);
    threshold(grayImage, filteredImage, 230, 255, CV_THRESH_BINARY); // lower threshold 235
    - 240
}

void yellowColorFilter (cv::Mat image, cv::Mat& filteredImage)
{
    cv::Mat imageHSV;
    cvtColor(image, imageHSV, CV_BGR2HSV);
    inRange(imageHSV, cv::Scalar(17, 76, 127), cv::Scalar(30, 255, 255), filteredImage);
}
```

Abbildungsverzeichnis

Abbildung 1: Systemarchitektur für das autonome Fahren (vgl. [WSK13], [SNS11]).....	10
Abbildung 2: Funktionsweise eines Ultraschallsensors	11
Abbildung 3: Modell einer digitalen Kamera.....	13
Abbildung 4: RGB-Single-Chip mit Bayers Pixelanordnung (vgl. [SNS11] S. 147).....	13
Abbildung 5: Mögliche Längsmarkierungen nach der StVO (vgl. [BMV02]).....	19
Abbildung 6: Fahrbahnmodell eines geraden Straßenabschnitts.....	19
Abbildung 7: Erkennung einer Fahrbahnmarkierung (vgl. [K07], [YGD11])	20
Abbildung 8: Inverse Perspektiventransformation.....	21
Abbildung 9: Canny-Filterung	22
Abbildung 10: Hough-Transformation einer Geraden	23
Abbildung 11: Hough-Transformation einer geraden Fahrbahn.....	23
Abbildung 12: Straßenmarkierungserkennung.....	24
Abbildung 13: Haar-Merkmale (vgl. [POP98], [B12], [VJ01])	25
Abbildung 14: Kaskadenklassifizierung	26
Abbildung 15: Schematische Darstellung eines künstlichen neuronalen Netzwerks	28
Abbildung 16: Geometrisches Modell zur optischen Abstandsmessung (vgl. [JLL04])	29
Abbildung 17: Iterativer Entwicklungsprozess.....	30
Abbildung 18: Systemdesign	35
Abbildung 19: Komponenten für den Prototyp zum autonomen Fahren	37
Abbildung 20: Aufbau des Modelfahrzeugs im Werkzustand.....	38
Abbildung 21: Zustandsdiagramm des Programms zum autonomen Fahren	39
Abbildung 22: Regelungsentwurf für das autonome Fahren.....	42
Abbildung 23: Klassendiagramm für den Systemzustand.....	44
Abbildung 24: Bilddatenklassendiagramm und Flussdiagramm der Zugriffsfunktionen	45
Abbildung 25: Fahrbahnmarkierungen und dazugehörige Polygonlinien.....	46
Abbildung 26: Klassendiagramm für die Fahrbahnerkennung	46
Abbildung 27: Klassendiagramm für die Verkehrsschilderkennung	47
Abbildung 28: Klassendiagramm für die Hinderniserkennung	47
Abbildung 29: Klassendiagramm für die Trajektoriendaten.....	48
Abbildung 30: Klassendiagramm für die Fahrzeugdaten	48
Abbildung 31: Klassendiagramm für den Zustand der graphischen Benutzerschnittstelle.....	49
Abbildung 32: Klassendiagramme für die Konfigurationsdaten.....	50
Abbildung 33: Flussdiagramm zur SSH- und VNC-Kommunikation.....	52

Abbildung 34: Aufbau des Prototyps.....	53
Abbildung 35: Befestigung der Elektronik mit dem Fahrzeugchassis	54
Abbildung 36: Kamerastativ des Prototyps	56
Abbildung 37: Pinbelegung am <i>Raspberry Pi 2 [RP18e]</i>	57
Abbildung 38: Verbindungsschaltplan zwischen <i>Raspberry Pi 3</i> und Ultraschallsensor	58
Abbildung 39: Verbindungsschaltplan von <i>Raspberry Pi 3</i> , PWM-Modul und Motoren	59
Abbildung 40: Pulsdauer und Servostellwinkel	60
Abbildung 41: Flussdiagramm der Funktion <i>main()</i>	63
Abbildung 42: Flussdiagramm der Funktion <i>exec()</i>	64
Abbildung 43: Flussdiagramm zum Laden der Konfiguration	64
Abbildung 44: Flussdiagramm des Moduls zur intrinsischen Kamerakalibration...	67
Abbildung 45: Flussdiagramm der intrinsischen Kalibrierung	68
Abbildung 46: Flussdiagramm des Moduls zur extrinsischen Kamerakalibration..	70
Abbildung 47: Flussdiagramm der extrinsischen Kalibrierung	71
Abbildung 48: Flussdiagramm für das Bilderfassungsmodul	73
Abbildung 49: Flussdiagramm des Moduls zur Fahrbahnerkennung	76
Abbildung 50: Flussdiagramm zum Modul für die Verkehrsschilderkennung.....	78
Abbildung 51: Flussdiagramm des Moduls zur Hinderniserkennung	80
Abbildung 52: Flussdiagramm des Moduls zur Fahrplanung	82
Abbildung 53: Flussdiagramm für das Modul der Fahrzeugsteuerung.....	86
Abbildung 54: Flussdiagramm für das Fernsteuerungsmodul.....	88
Abbildung 55: Flussdiagramm zum Benutzerschnittstellenmodul	90
Abbildung 56: Benutzerschnittstelle des <i>Standby</i> -Modus	91
Abbildung 57: Flussdiagramm für den <i>Standby</i> -Modus	92
Abbildung 58: Benutzerschnittstelle für den autonomen Fahrmodus.....	93
Abbildung 59: Flussdiagramm für den autonomen Fahrmodus	95
Abbildung 60: Flussdiagramm für den Entwicklungsmodus.....	97
Abbildung 61: Benutzerschnittstelle für den Entwicklungsmodus	98
Abbildung 62: Flussdiagramm für den Fernsteuerungsmodus.....	99
Abbildung 63: Benutzerschnittstelle für den Fernsteuerungsmodus	100
Abbildung 64: Flussdiagramm für den Konfigurationsmodus.....	101
Abbildung 65: Benutzerschnittstelle für den Konfigurationsmodus	102
Abbildung 66: Flussdiagramm zum Kalibrierungsmodus	103
Abbildung 67: Benutzerschnittstelle für den intrinsischen Kamerakalibrierungsmodus	104
Abbildung 68: Benutzerschnittstelle für den extrinsischen Kamerakalibrierungsmodus	104

Abbildung 69: Flussdiagramm zum Informationsmodus	105
Abbildung 70: Benutzerschnittstelle für den Informationsmodus	105
Abbildung 71: Flussdiagramm für den Fehlermodus	106
Abbildung 72: Flussdiagramm für den Beendigungsmodus.....	106
Abbildung 73: Geradeausfahrt mit Trajektorie	109
Abbildung 74: Korrekte Kurvenerkennung	110
Abbildung 75: Erkennung einer Kurvenaußenseite.....	110

Tabellenverzeichnis

Tabelle 1: CPU-Auslastung im <i>Standby</i> - und autonomer Fahrmodus	113
Tabelle 2: FPS im autonomen Fahrbetrieb	113

Literaturverzeichnis

Bücher

- [CS14] S. Chacon und B. Straub, *Pro Git*, Apress, 2014
- [E12] A. Eskandarian, *Handbook of Intelligent Vehicles*, Springer, 2012
- [GBC16] I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning*, MIT Press, 2016
- [KB17] A. Kaehler und G. Bradski, *Learning OpenCV 3*, O'Reilly, 2017
- [MGL15] M. Maurer, J. C. Gerdts, B. Lenz, H. Winner, *Autonomes Fahren*, Springer Vieweg, 2015
- [SNS11] R. Siegwart, I. R. Nourbakhsh und D. Scaramuzza, *Introduction to Autonomous Mobile Robots*, MIT Press, 2011
- [TBF06] S. Thun, W. Burgard und D. Fox, *Probabilistic Robotics*, MIT Press, 2006

Zeitschriften

- [B95] A. Broggi, *Robust Real-Time Lane and Road Detection in Critical Shadow Conditions*, In Proceedings IEEE International Symposium on Computer Vision, IEEE, 1995, S. 353-358)
- [BAS12] *Rechtsfolgen zunehmender Fahrzeugautomatisierung*, Forschung kompakt, Bundesanstalt für Straßenwesen, 2012
- [BBR07] C. Basarke, C. Berger und B. Rumpe, *Software & Systems Engineering Process and Tools for the Development of Autonomous Driving Intelligence*, Journal of Aerospace Computing, Information and Communication, Vol. 4, Nr. 12, 2007, S. 1158-1174
- [BDD16] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, K. Zieba, *End to End Learning for Self-Driving Cars*, Cornell University, 2016
- [BFK08] J. Bohren et al., Little Ben: The Ben Franklin Racing Team's Entry in the 2007 DARPA Urban Challenge, *Journal of Field Robotics*, Wiley, 2008, S. 598-614
- [BGL15] F. Bounini, D. Gingras, V. Lapointe und H. Pollart, *Autonomous Vehicle and Real Time Road Lanes Detection and Tracking*, Vehicle Power and Propulsion Conference, IEEE, 2015, S. 1-6
- [BK15] F. Badstübner und R. Ködel, *DESERVE Platform – 2nd release*, DESERVE, 2014
- [BL07] Y. Bengio und Y. LeCun, *Scaling Learning Algorithms towards AI*, 2007
- [BR12] C. Berger und B. Rumpe, *Engineering Autonomous Driving Software*, Experience from the DARPA Urban Challenge, Springer, 2012
- [BT15] S. Behere und M. Törngren, *A Functional Architecture for Autonomous Driving*, WASA ,15, Proceedings of the First International Workshop on Automotive Software Architecture, ACM, 2015, S. 3-10

- [CEH10] M.Campbell, M. Egerstedt, J. P. How und R. M. Murray, *Autonomous driving in urban environments: approaches, lessons and challenges*, Philosophical Transactions A, The Royal Society, 2010, S. 4649-4672
- [CP01] A. Carullo, M. Parvis, *An Ultrasonic Sensor for Distance Measurement in Automotive Applications*, IEEE Sensors Journal, 2001, S. 143-147
- [E01] W. Elmenreich, *An Introduction to Sensor Fusion*, Research Report, Technische Universität Wien, 2001
- [F02] D. Fox, *KLD-Sampling: Adaptive Particle Filters*, In Advances in Neural Information Processing Systems 14, MIT Press, 2002
- [GB07] E. Grossi und M. Buscema, Introduction to artificial neural networks, European Journal of Gastroenterology & Hepatology, 2007, S. 1046-1054
- [GPD06] O. Gietelink, J. Ploeg, B. De Schutter und M. Verhaegen, *Development of advanced driver assistance systems with vehicle hardware-in-the-loop simulations*, Vehicle System Dynamics: International Journal of Vehicle Mechanics and Mobility, 2006, S. 569-590
- [GPG13] D. Gruyer, S. Pechberti und S. Glaser, Development of Full Speed Range ACC with SiVIC, a virtual platform for ADAS Prototyping, Test and Evaluation, Intelligent Vehicles Symposium, IEEE, 2013, S. 93-98
- [HGB10] N. Hiblot, D. Gruyer, J.-S. Barreio und B. Monnier, Pro-SiVIC and Roads, a software suite for sensors simulation and virtual prototyping of ADAS, Proceedings of DSC, 2010, S. 277-288
- [HHH09] S. Han, Y. Han und H. Han, *Vehicle Detection Method using Haar-like Feature on Real Time System*, World Academy of Science, Engineering and Technology, 2009, S.455-459
- [IO14] S. Isik, K. Özkan, *A Comparative Evaluation of Well-known Feature Detectors and Descriptors*, International Journal of Applied Mathematics, Electronics and Computers, 2014
- [JB10] K. Jaskot und A. Babiarz, Autonomous Mobile Platform, International Carpathian Control Conference, 2010, S. 179-183
- [JB11] K. Jaskot und A. Babiarz, Autonomous mobile platform II, Carpathian Control Conference, IEEE, 2011, S. 167-171
- [JK05] C. R. Jung und C. R. Kelber, *Lane following and lane departure using a linear-parabolic model*, Image and Vision Computing, Elsevier, 2005, S. 1192-1202
- [JKK14] K. Jo, J. Kim, D. Kim, C. Jang und M. Sunwoo, *Development of Autonomous Car—Part I: Distributed System Architecture and Development Process*, IEEE Transactions on Industrial Electronics, Vol. 61, Nr. 12, 2014, S. 7131-7140
- [JKK15] K. Jo, J. Kim, D. Kim, C. Jang und M. Sunwoo, *Development of Autonomous Car—Part II: A Case Study on the Implementation of an*

- Autonomous Driving System Based on Distributed Architecture, IEEE Transcations on Industrial Electronics, Vol. 62, No. 12, 2015, S. 5119-5132
- [JLL04] C. Jiangwei, J. Lisheng, G. L. Libibing und W. Rongben, *Study on Method of Detecting Preceding Vehicle Based on Monocular Camera*, Intelligent Vehicles Symposium, IEEE, 2004, S.750-755
- [KFM04] C. Kwok, D. Fox, M. Meila, *Real-Time Particle Filter*, Proceeding of the IEEE, IEEE, 2004, S. 469-484
- [KPK14] M. Kutila, P. Pyykönen, P. v. Koningsbruggen, N. Pallaro und J. Pérez-Rastelli, *The DESERVE project: Towards future ADAS functions*, International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation, IEEE, 2014, S. 308-313
- [LSA09] K. H. Lim, K. P. Seng, L. M. Ang und S. W. Chin, *Lane Detection and Kalman-Based Linear-Parabolic Lane Tracking*, International Conference on Intelligent Human-Machine Systems and Cybernetics, 2009, S. 351-354
- [MBB08] M. Montemerlo et al., Junior: The Stanford entry in the Urban Challenge, Journal of Field Robotics, 2008, Wiley, S. 569-597
- [MGK00] J. Matas, C. Galambos, J. Kittler, *Robust detection of lines using the progressive probabilistic hough transform*, Computer Vision and Image Understanding, 2000, S. 119–137
- [NCP07] P. Negri, X. Clady und L. Prevost, *Benchmarking Haar and Histograms of oriented Gradients features applied to Vehicle Detection*, Image Analysis and Processing, Proceedings of the Fourth International Conference on Informatics in Control, Automation and Robotics, 2007, S. 359-364
- [NN73] C. A. Nagler, W. M. Nagler, *Reaction time measurements*, Forensic Science, 1973, S. 261-274
- [NVF13] A. M. Neto, A. C. Victorino, I. Fantoni and J. V. Ferreira, *Real-time estimation of drivable image area based on monocular vision*, Intelligent Vehicles Symposium Workshops, IEEE, 2013, S. 63-68
- [PAG15] G. S. Pannu, M. D. Ansari und P. Gupta, *Design and Implementation of Autonomous Car using Raspberry Pi*, International Journal of Computer Applications, Foundation of Computer Science, 2015
- [R12] A. Rolfsmeier et al., Development Method, DESERVE, 2012
- [RBL09] F. W. Rauskolb et al., Caroline: An Autonomously Driving Vehicle for Urban Environments, Journal of Field Robotics, 2009, Wiley, S. 674-724
- [RR07] J. Rojo, R. Rojas et al., Spirit of Berlin: An Autonomous Car for the DARPA Urban Challenge Hardware and Software Architecture, Team Berlin, Freie Universität Berlin, 2007
- [SCB13] S. Sommer, A. Camek, K. Becker, C. Buckl, A. Zirkler, L. Fiege, M. Armbruster, G. Spiegelberg und A. Knoll, *RACE: A Centralized Platform Computer Based Architecture for Automotive Applications*, Electric Vehicle Conference, IEEE International, 2013, S. 1-6

- [SL14] M. Song und X. Liu, *Research in Video Detection of Lane Curve and Its Application in Speed Alert System*, Research Journal of Applied Sciences, Engineering and Technology, Maxwell Scientific Organization, 2014, S. 957-962
- [TFB00] S. Thrun, D. Fox, W. Burgard, F. Dellaert, *Robust Monte Carlo localization for mobile robots*, Artificial Intelligence 128, 2000, S. 99-141
- [UAB08] C. Urmson et al., Autonomous Driving in Urban Environments: Boss and the Urban Challenge, Journal of Field Robotics, Wiley, 2008, S. 425-466
- [VGG10] B. Vanholme, D. Gruyer, S. Glaser und S. Mammar, Fast prototyping of a Highly Autonomous Cooperative Driving System for public roads, Intelligent Vehicles Symposium, IEEE, 2010, S. 135-142
- [VJ01] P. Viola und M. Jones, *Rapid Object Detection using a Boosted Cascade of Simple Features*, Proceedings of the 2001 IEEE Computer Society Conference on Computer Vision and Pattern Recognition, IEEE, 2001, S. 511-518
- [WSK13] J. Wei, J. M. Snider, J. Kim, J. M. Dolan, R. Rajkumar und B. Litkouhi, Towards a viable autonomous driving research platform, Intelligent Vehicles Symposium (IV), IEEE, 2013, S. 763-770
- [YGD11] X. Yang, D. Gao, J. Duan und L. Yang, *Research on Lane Detection Based on Machine Vision*, Proceedings of the International Conference on Informatics, Cybernetics, and Computer Engineering, Springer, 2011, S. 19-20
- [ZCX09] J. Zhou, H. Chen und C. Xiu, *A Simulation Model to Evaluate and Verify Functions of Autonomous Vehicle Based on Simulink®*, 2009, S. 645-656

Diplomarbeiten und Dissertationen

- [B12] M. Blauth, *Detektion und Klassifikation von Verkehrszeichen in Bildsequenzen basierend auf Farb- und Formmerkmalen*, Fachhochschule Kaiserlautern, 2012
- [E02] W. Elmenreich, *Sensor Fusion in Time-Triggered Systems*, Technische Universität Wien, 2002
- [K07] A. Kant, *Bildverarbeitungsmodul zur Fahrspurerkennung für ein autonomes Fahrzeug*, Hochschule für angewandte Wissenschaften Hamburg, 2007
- [S09] J. Speth, *Videobasierte modellgestützte Objekterkennung für Fahrerassistenzsysteme*, Technische Universität München, 2009
- [S13] C. M. Schmidts, *Fahrbahnerkennung und -rekonstruktion in der Ebene durch Analyse von Umfeldveränderungen in Folgen von monokularen Grauwertbildern*, Technische Universität München, 2013
- [S14] A. Staudenmaier, *Ressourcenoptimierte Objektdetektion und teilüberwachtes Lernen zur Echtzeitanwendung mit konfidenzbasierten*,

kaskadierten Klassifikationssystemen, Technische Universität Dortmund,
2014

Internet

- [A18a] Audi Autonomous Driving Cup, Zugriff am 5.8.2018 unter <https://www.audi-autonomous-driving-cup.com/>
- [A18b] O. Aflak, *HC-SR04 Raspberry Pi C*, GitHub, Zugriff am 5.8.2018 unter <https://github.com/OmarAflak/HC-SR04-Raspberry-Pi-C>
- [AE18] J. Altenberg, H. Egger, *Echtzeit mit Linux, Embedded Software Engineering*, Vogel Comunications Group GmbH & Co. KG, Zugriff am 5.8.2018 unter <https://www.embedded-software-engineering.de/echtzeit-mit-linux-a-632464/>
- [ASF18] *Subversion*, The Apache Software Foundation, Zugriff am 5.8.2018 unter <https://subversion.apache.org>
- [B18a] E. Brown, *Inside Real-Time Linux*, Linux.com, The Linux Foundation, Zugriff am 5.8.2018 unter <https://www.linux.com/news/event/elce/2017/2/inside-real-time-linux>
- [B18b] F. Bevilacqua, *CVUI*, Zugriff am 5.8.2018 unter <https://dovyski.github.io/cvui/>
- [BB18] QNX, Blackberry, Zugriff am 5.8.2018 unter <http://blackberry.qnx.com/en/products/qnxcar/index>
- [D18] Autonom unterwegs, Daimler AG, Zugriff am 5.8.2018 unter <https://www.daimler.com/innovation/autonomes-fahren/special/veraenderungen.html>
- [E18] Edimax EW-7811un, Edimax, Zugriff am 5.8.2018 unter http://www.edimax.com/edimax/merchandise/merchandise_detail/data/edimax/global/wireless_adapters_n150/ew-7811un
- [EB18] EB Assist ADTF, Elektrobit, Zugriff am 5.8.2018 unter <https://www.elektrobit.com/products/eb-assist/adtf/>
- [G18a] B. Gabor, *Camera calibration with OpenCV*, OpenCV-Dokumentation, Zugriff am 5.8.2018 unter https://docs.opencv.org/3.3.1/d4/d94/tutorial_camera_calibration.html
- [G18b] M. Gaynor, Stopsigns, GitHub, Zugriff am 5.8.2018 unter <https://github.com/markgaynor/stopsigns>
- [GNU18a] Octave, GNU, Free Software Foundation, Zugriff am 5.8.2018 unter <https://www.gnu.org/software/octave/>
- [GNU18b] Nano, GNU, Free Software Foundation, Zugriff am 5.8.2018 unter <https://www.nano-editor.org/>
- [GT18] *AutoRally*, Georgia Tech, Zugriff am 5.8.2018 unter <https://autorally.github.io>

- [K18] *CMake*, Kiteware, Zugriff am 5.8.2018 unter <https://cmake.org>
- [M18a] *MATLAB*, Mathworks, Zugriff am 5.8.2018 unter
<http://de.mathworks.com/products/matlab/>
- [M18b] B. Moolenaar, *Vim*, Zugriff am 5.8.2018 unter <http://www.vim.org>
- [MIT18] *RACECAR*, Massachusetts Institute of Technology, Zugriff am 5.8.2018 unter <https://mit-racecar.github.io>
- [N18a] *NVIDIA Drive PX*, NVIDIA, Zugriff am 5.8.2018 unter
<http://www.nvidia.de/object/drive-px-de.html>
- [N18b] *NVIDIA Driveworks*, NVIDIA, Zugriff am 5.8.2018 unter
<http://www.nvidia.de/object/driveworks-de.html>
- [N18c] *NVIDIA JETSON*, NVIDIA, Zugriff am 5.8.2018 unter
<https://www.nvidia.de/autonomous-machines/embedded-systems-dev-kits-modules/?section=jetsonDevkits>
- [OCV18] *OpenCV*, Itseez, Zugriff am 5.8.2018 unter <http://opencv.org>
- [OSS18] *OpenSSH*, OpenBSD, Zugriff am 5.8.2018 unter <https://www.openssh.com/>
- [PCH18] D. Platis, Y. Caglar, A. Hontabat, S. Ivanov, D. Jensen, I. Karouach, J. Li, P. Theodoridou, The world's first Android autonomous vehicle, Zugriff am 5.8.2018 unter <https://platis.solutions/blog/2015/06/29/worlds-first-android-autonomous-vehicle/>
- [RP18a] Raspberry Pi 2 Model B, Raspberry Pi Foundation, Zugriff am 5.8.2018 unter <https://www.raspberrypi.org/products/raspberry-pi-2-model-b/>
- [RP18b] *NOOBS*, Raspberry Pi Foundation, Zugriff am 5.8.2018 unter <https://www.raspberrypi.org/downloads/noobs/>
- [RP18c] *Raspberry Pi Touch Display*, Raspberry Pi Foundation, Zugriff am 5.8.2018 unter <https://www.raspberrypi.org/products/raspberry-pi-touch-display/>
- [RP18d] *Secure Shell*, Raspberry Pi Foundation, Zugriff am 5.8.2018 unter <https://www.raspberrypi.org/documentation/remote-access/ssh/>
- [RP18e] *GPIO: Models A+, B+, Raspberry Pi 2 B and Raspberry Pi 3 B*, Raspberry Pi Foundation, Zugriff am 5.8.2018 unter <https://www.raspberrypi.org/documentation/usage/gpio/README.md>
- [RP18f] *NOOBS Installation*, Raspberry Pi Foundation, Zugriff am 5.8.2018 unter <https://www.raspberrypi.org/documentation/installation/noobs.md>
- [RP18g] Raspberry Pi 3 Model B, Raspberry Pi Foundation, Zugriff am 5.8.2018 unter <https://www.raspberrypi.org/products/raspberry-pi-3-model-b/>
- [RTM18] *RTMaps*, Intempora, Zugriff am 5.8.2018 unter <https://intempora.com/products/rtmaps.html>
- [RVN18] RealVNC Limited, Zugriff am 5.8.2018 unter <https://www.realvnc.com/>
- [SFC18] *Git*, Software Freedom Conservancy, Zugriff am 5.8.2018 unter <https://git-scm.com>

- [T18a] *Autopilot*, Tesla Inc., Zugriff am 5.8.2018 unter https://www.tesla.com/de_AT/autopilot
- [T18b] G. Todorov, *PCA9685*, Github, Zugriff am 5.8.2018 unter <https://github.com/TeraHz/PCA9685>
- [TI18a] *Driving Automotive ADAS technology leadership with TDAX SoCs*, Texas Instruments, Zugriff am 5.8.2018 unter http://www.ti.com/lscds/ti/processors/dsp/automotive_processors/tdax_adas_socs/overview.page
- [TI18b] *TDA3x SoC processor delivers cost-effective ADAS solutions for front, rear and surround view applications*, Texas Instruments, Zugriff am 5.8.2018 unter <http://www.ti.com/lit/wp/spry272a/spry272a.pdf>
- [TUB18] *Carolo-Cup*, Technische Universität Braunschweig, Zugriff am 5.8.2018 unter <https://wiki.ifr.ing.tu-bs.de/carolocup/>
- [U18] *Steel City's New Wheels*, Uber Technologies Inc., Zugriff am 5.8.2018 unter <https://www.uber.com/blog/pennsylvania/new-wheels/>
- [V18] *Drive Me*, Volvo Car Cooperation, Zugriff am 5.8.2018 unter <https://www.volvocars.com/intl/buy/explore/intellisafe/autonomous-driving/drive-me>
- [VF07] A. Vedaldi, B. Fulkerson, *VisionLab Features Library*, Zugriff am 5.8.2018 unter <http://www.vlfeat.org/man/vlfeat.html>
- [VW18a] *Autonom Fahren auf Knopfdruck*, Volkswagen AG, Zugriff am 5.8.2018 unter <https://www.volkswagenag.com/de/news/stories/2017/03/autonomous-driving-at-the-push-of-a-button.html>
- [VW18b] *Pilotiertes Fahren mit künstlicher Intelligenz: Audi kooperiert mit Top-Unternehmen der Elektronikindustrie*, Volkswagen AG, Zugriff am 5.8.2018 unter https://www.volkswagenag.com/de/news/2017/01/Audi_CES.html
- [W18a] Waymo Inc., Zugriff am 5.8.2018 unter <https://waymo.com/journey/>
- [W18b] T. Wessmann, *Adafruit (PCA9685) C Servo Controller (Raspberry Pi)*, Resources and tutorials, Zugriff am 5.8.2018 unter <http://tordwessman.blogspot.com/2013/12/adafruit-pca9685-c-servo-controller.html>
- [W18c] Z. Wang, *Self Driving RC Car*, Zugriff am 5.8.2018 unter <https://zhengludwig.wordpress.com/projects/self-driving-rc-car/>

Gesetze, Normen und Standards

- [BMV02] *Verordnung des Bundesministers für öffentliche Wirtschaft und Verkehr über Bodenmarkierungen*, 2002, Zugriff am 5.8.2018 unter <https://www.ris.bka.gv.at/GeltendeFassung.wxe?Abfrage=Bundesnormen&Gesetzesnummer=10012574>

- [IEC10] Functional Safety, IEC 61508, International Electronical Commission, 2010, Zugriff am 5.8.2018 unter <http://www.iec.ch/functionsafety/>
- [ISO11] Road Vehicles – Functional Safety, ISO 26262, International Organization for Standardization, 2011, Zugriff am 5.8.2018 unter <https://www.iso.org/standard/43464.html>
- [ISO13] Road Vehicles – *FlexRay communications systems*, ISO 17458-1:2013, International Organization for Standardization, 2013, Zugriff am 5.8.2018 unter <https://www.iso.org/standard/59804.html>
- [ISO15] Road Vehicles – *Controller Area Network*, ISO 11898-1:2015, International Organization for Standardization, 2015, Zugriff am 5.8.2018 unter <https://www.iso.org/standard/63648.html>
- [OMG17] *Unified Modelling Language Specification Version 2.5.1*, Object Management Group, Zugriff am 5.8.2018 unter <https://www.omg.org/spec/UML/2.5.1>
- [SAE18] *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*, SAE J3016, Society of Automotive Engineers, 2018, Zugriff am 5.8.2018 unter https://www.sae.org/standards/content/j3016_201806/

Persönliche Daten

Name:	Sergiu-Petru Tabacariu
Geburtsdatum:	29. Juni 1988
Geburtsort:	Vaslui, Rumänien
Nationalität:	Österreich

Schulbildung

Schule:	BRG 14, 1140 Wien
Hochschulzugang:	Matura
Abschlussdatum:	13.06.2006
Leistungskurse:	Naturwissenschaftlicher Schwerpunkt Informatik als Matura- und Wahlpflichtfach

Studium

Dauer:	Seit 25.08.2014
Hochschule:	FH Campus Wien, 1100 Wien Technische Universität Wien, 1040 Wien
Abschluss:	September 2018
Titel der Diplomarbeit/	Entwicklungsumgebung für das autonome Fahren
Masterarbeit:	Am Beispiel eines Modellfahrzeugs
BetreuerIn der Diplomarbeit/	FH-Prof. Dipl.-Ing. Dr. techn. Dr.-Ing. Gernot Kucera
Masterarbeit:	FH-Prof. Dipl.-Ing. Gerhard Engelmann
Studiengächer:	Masterstudium Embedded Systems Engineering Bachelorstudium Informationstechnologie und Telekommunikation Bachelorstudium Technische Informatik

Berufspraxis

02.12.2015 – heute	Autoservice Stefan, 1100 Wien Kundenservice, IT-Administrator, Werbung (Printmedien-Design, Webdesign)
14.11.2014 – 31.01.2015	FH Campus Wien, 1100 Wien Forschung und Entwicklung, Projekt „Horsevent“ (Software für Pferdebeatmungsgerät in Kooperation mit der Veterinärmedizinischen Universität Wien)
07.09.2011 – 02.07.2014	Tutor für Embedded Systems und Real-Time Operating Systems

02.01.2012 – 01.09.2014	Ford Stefan bzw. Autoservice Stefan, 1100 Wien Kundenservice, IT-Administrator, Werbung (Printmedien-Design, Webdesign)
12.11.2009 – 30.06.2011	Sagemcom Austria GmbH, 1230 Wien Praktikum als Projektleiter-Assistent
18.11.2006 – 20.05.2009	XXXLutz KG Wien 15, 1500 Wien Verkäufer und Einrichtungsberater (Kundenbetreuung und Beratung, Verkauf)

Sonstiges

Sprachkenntnisse	Deutsch (Zweitsprache, sehr gute Kenntnisse) Englisch (sehr gute Kenntnisse) Rumänisch (Muttersprache, sehr gute mündliche Kenntnisse, schriftliche Grundkenntnisse)
IT-Kenntnisse	<p>Programmiersprachen: C, C++, Java, Basiskenntnisse in Assembler</p> <p>Web: HTML, CSS, CMS, SQL, PHP</p> <p>Office: Microsoft Office</p> <p>Betriebssysteme: Microsoft XP/Vista/7/10, Linux, Mac OS X</p> <p>Grafik: Adobe Photoshop, Illustrator, InDesign, Flash</p> <p>Foto- & Videografie: Adobe Photoshop Lightroom, Premiere Pro</p> <p>Referenzen: www.autoservicestefan.at (Design, Grafik, CMS) www.eg-wien-meidling.at (Design, Grafik, CMS) www.friedmann-preis.org (Design, Grafik) </p>