Master Thesis at the Institute of Computer Science of Freie Universität Berlin in Cooperation with Hella Aglaia Mobile Vision GmbH

Object Detection and Tracking with Side Cameras and RADAR in an Automotive Context

Peter Hofmann
Student Number: 4491321
peter.hofmann@fu-berlin.de

Supervisor: Dr. Andreas Tewes
Submitted to: Prof. Dr. Raul Rojas

Berlin
September 9, 2013

Zusammenfassung

Deutsche Version:

Auf der Basis von RADAR-Sensoren entwickelt Hella Aglaia unter anderem Fahrerassistenzsysteme zur Vorhersage potentieller Auffahr– und Seitenaufprallunfälle. Da in letzter Zeit vor allem durch Einparksysteme vermehrt Kameras auch im Seitenbereich Einzug in die Serienproduktion erhalten, bietet es sich an diese auch für andere Aufgaben zu verwenden. Besonders für die Vorhersage von Unfällen in Kreuzungsbereichen oder Ausparkszenarios müssen Fahrzeuge im Seitenbereich zuverlässig detektiert werden können. Im Rahmen dieser Masterarbeit wird untersucht, inwieweit durch zusätzliche Kameras mit Weitwinkelobjektiven im Seitenbereich RADAR-Trackingergebnisse von Längs- und Seitenverkehr verbessert werden können. Dafür wird ein Versuchsfahrzeug mit entsprechenden Kameras ausgestattet. Dies umfasst die Auswahl der Kameras sowie deren Kalibrierung und die Abstimmung mit den anderen Komponenten des Versuchsträgers. Dafür wurde ein Standard Pin-Hole Kameramodell mit mehreren Verzerrungsparametern mit dem aktuelleren Scaramuzza-Kameramodell verglichen. Weiterhin werden Algorithmen zur Verbesserung von RADAR-Objekt-Tracks auf Basis von Deformable Model Fitting und der Vorhersage von optischem Fluss untersucht und weiterentwickelt. Zudem wird eine Methode vorgestellt, mit der es möglich ist künstliche Daten für das Testen einzelner Module der RADAR-Warnsysteme von Hella Aglaia zu erzeugen um so den Test- und den Entwicklungsprozess zu vereinfachen.

Abstract

English Version:

Hella Aglaia develops, amongst others, driving assistance systems based on RADAR sensors for the prediction of potential rear and side crash accidents. Within the last years, more cameras have been built into production vehicles - especially for assisted parking systems. Certainly, these cameras could be used for other tasks as well. Particularly in situations when backing out of parking spaces or at crossroads, side traffic should be reliably detected. In the scope of this thesis, ways are explored in which additional side cameras equipped with wide angle optics can be exploited to enhance and refine RADAR tracking results. For this purpose a test bed vehicle is set up. This includes the choice of suitable cameras, their calibration and the set up of the software framework. For the process of camera calibration, a standard pin-hole camera model with distortion parameters and the more recent Scaramuzza camera model are compared and evaluated. Furthermore, approaches for optimization and refinement of RADAR object tracks based on deformable model fitting and optical flow prediction are examined, evaluated and further developed. Additionally a method for creating artificial data for the testing of specific modules of Hella Aglaia's RADAR based warning systems is demonstrated which simplifies the testing and development process.

Eidesstattliche Erklärung

Ich versichere hiermit an Eides statt, dass diese Arbeit von niemand anderem als meiner Person verfasst worden ist. Alle verwendeten Hilfsmittel wie Berichte, Bücher, Internetseiten oder ähnliches sind im Literaturverzeichnis angegeben, Zitate aus fremden Arbeiten sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungskommission vorgelegt und auch nicht veröffentlicht.

9. September 2013

____

Peter Hofmann

Acknowledgement

First and foremost I offer my sincerest gratitude to my supervisor at Hella Aglaia, Dr. Andreas Tewes, who has supported me throughout my thesis with his patience and knowledge. Additionally, I would like to thank Prof. Dr. Rojas for supervising my thesis from the side of Freie Universität Berlin.

My sincere thanks also go to all other colleagues at Hella Aglaia as well as to my friends who supported me throughout the time of my thesis.

Last but not the least, I would like to thank my family, especially my parents for giving birth to me in the first place and supporting me throughout my life.

Contents

Chapter 1

Introduction

1.1 Motivation

Hella Aglaia is a company that develops, amongst others, driving assistance systems based on Radio Detection And Ranging (RADAR) sensors. Those RADAR systems track surrounding vehicles and add additional safety through warning mechanisms to warn the driver in the case of potential accidents. Currently, the Sensor Solutions group provides several RADAR based products. Pre-Crash Rear (PCR) is an application which provides a warning system for potential rear-end collisions. Car producing companies can use the output of this system to send a direct warning to the driver or to prepare the car for a potential impact, e.g. by warning the driver or tightening the safety belts. The second product – Front/-Side RADAR (FSR) – provides a similar system for potential side impacts which improves safety especially at cross roads. Hella Aglaia has strong backgrounds in camera based driving assistance systems in its history as well. However, the FSR project at Hella Algaia allows vehicle tracking solely based on RADAR measurements. Car manufactures tend to build an increasing number of sensors into their cars, e.g. cameras, short range and long range RADAR systems or ultrasound sensors. These sensors have become standards even in mid-range priced cars. With a wider availability of different sensors, it is reasonable to explore ways of combining these sensors and improve the performance of applications. Thus, one goal of this thesis is to set up a test bed vehicle which allows the fusion of side RADAR and side camera data in the FSR context. This includes the identification of requirements of the hardware and the respective evaluation of different camera sensors and optics.

Sensor fusion in general exploits advantages of different sensors by compensating disadvantages with information from other sensors. The advantages of RADAR sensors are good velocity and distance measurements with rather limited angular resolution which come at the cost of imprecise lateral extent estimation. Monocular cameras, on the other side, allow good estimation of the width and height of an object if the distance is known. Based on the test bed, fusion methods for combining the advantages of both sensors are explored to improve the performance of the RADAR tracker of FSR.

Additionally, the performance of the warning systems is mainly evaluated based on scenarios of a test catalog with given specifications how the system should react under certain circumstances. For example, a test scenario defines the ego car driving with a specific velocity on a turn in the road. If another vehicles overtakes the ego vehicle, the warning system should not execute. Some of the described maneuvers require advanced driving skills and consume a lot of time to record properly. Instead, a simpler method to generate the data which would be acquired from test drives is preferable for development and testing. However, the data has to be physically reasonable and consistent with the test catalog. Therefore, the thesis explores and implements a way to intuitively generate artificial ground truth data that can be used for development and module tests of warning systems.

1.2 Sensor Fusion

Over the last three decades there have been several architectures and methods to fuse sensor data. The idea of sensor fusion is not very new and can be found in nature and humans. For humans, even basic tasks, e.g. grabbing an object, require processing input from several sensors. Information from the eyes as well as motor sensory feedback is used to find an optimal grip on the object. The general concept combines advantages of different sensors by compensating their respective disadvantages. A definition of Sensor Fusion can be found in The Handbook of Multisensor Data Fusion.

> *Data fusion techniques combine data from multiple sensors and related information to achieve more specific inferences than could be achieved by using a single, independent sensor.* [HL01]

Especially with new sensor types emerging in affordable price ranges and higher processing power, fusion concepts become more important. Abstractly, one can define three main methods of sensor fusion that are used and adopted in a wide range of applications. In [HL01] these are referenced as *Data Level Fusion*, *Feature Level Fusion* and *Declaration Level Fusion*. In this context, a few terms have to be defined. A sensor is a device which provides information of the environment in any form of raw data. Feature extraction is defined as the process of extracting meaningful information from the raw data of a sensor, e.g. points that represent corners in a camera image. State estimation describes the current state based on the given input, e.g. the position of a tracked vehicle.

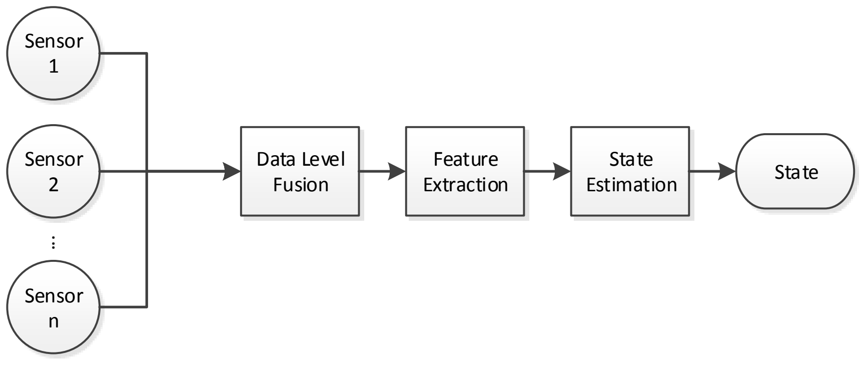


Figure 1.1: Sensor fusion at data level (Low Level Sensor Fusion)

Data Level Fusion or Low Level Sensor Fusion (Figure 1.1) describes a method of combining raw data from different sensors with each other, e.g. having a calibrated camera and a calibrated Time-of-Flight (ToF) depth-sensor which creates a depth map of the environment, each camera pixel can be mapped to a distance measurement of the ToF sensor and vice versa.

In *Feature Level Fusion* (Figure 1.2) or Mid-Level Fusion, the sensor data is presented via feature vectors which describe meaningful data extracted from the raw data. These feature vectors build the base for fusion. This method can be found in 3D-reconstruction for example. In this approach image features from different cameras are extracted to identify corresponding points in each image of different camera views.

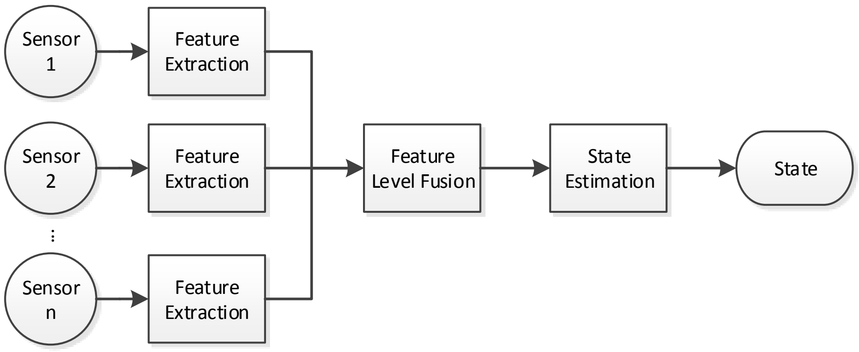


Figure 1.2: Sensor fusion at feature level (Mid-Level Sensor Fusion)

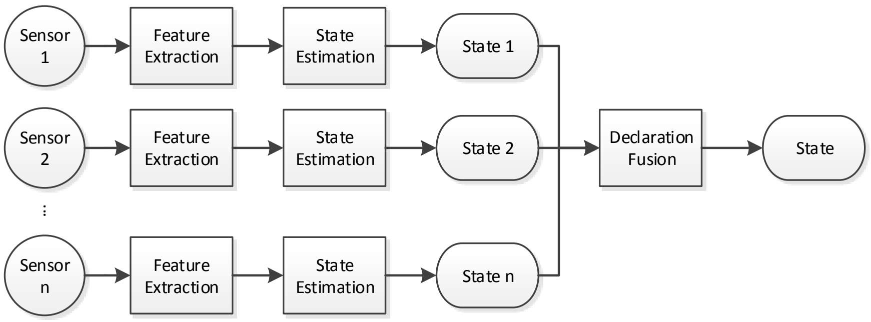


Figure 1.3: Sensor fusion at declaration level (High Level Sensor Fusion)

Declaration Level Fusion or High Level Fusion (Figure 1.3) is the combination of independent state hypotheses from different sensors. All sensors estimate the state individually. The final state is a fusion of all state hypotheses from the different sensors. The most famous implementation of this approach is probably the Kalman Filter [KB61]. Each state from the individual sensors with their respective error covariance is used for correcting the state estimate in the Kalman Filter. The error covariance represents the trust in the state estimation, e.g. a camera image is reliable for estimating the width of objects but distance or speed measurements are very inaccurate. In contrast a RADAR sensor provides very accurate distance and velocity measurements. Thus, in the final state estimate, velocity information and distance will be closer to the RADAR measurements, while the size would be closer to the measurements from the camera which, in theory, should result in a better final state estimate.

1.3 Preliminaries

Working with different sensors involves different types of coordinate systems that, at some point, have to be transformed into each other. The following pages introduce mathematical notations and coordinate systems used in this thesis.

1.3.1 Mathematical Notations

Frequently used elements in this thesis are scalars, vectors and matrices. An angle is a special kind of a scalar and annotated with Greek letters. The respective notations can be found in Table 1.1.

Type	Notation	Explanation
scalar	s	normal font, small letters
angle	φ	normal font, Greek letters
vector	$\mathbf{v}$	small letters, bold
matrix	$\mathbf{M}$	big letters, bold
coordinate transform	$\mathbf{T}_{B\leftarrow A}$	big bold letters, transforms points from coordinate system A to B with $\mathbf{x}_B = \mathbf{T}_{B\leftarrow A}\mathbf{x}_A$

Table 1.1: Mathematical notations

Homogeneous Coordinates and Coordinate Transforms

Coordinate representations of n-dimensional space can be defined as homogeneous coordinates, which represent lines in $n+1$-dimensional space by adding 1 as a constant to the vector.

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{1.1}$$

This allows combining rotations and translations in a more compact way. These coordinate transforms are annotated as $\mathbf{T}_{B\leftarrow A}$ and can be used for arbitrary transformations from n-dimensional coordinate system A to d-dimensional coordinate system B but usually define a transformation matrix containing a rotation matrix $\mathbf{R}$ and a translation vector $\mathbf{t}$ (eq. (1.2)) that is used to transform homogeneous coordinates.

$$\mathbf{T}_{B\leftarrow A} = \begin{pmatrix} & \mathbf{R} & & \mathbf{t} \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.2}$$

$$\mathbf{x}_B = \mathbf{T}_{B\leftarrow A}\mathbf{x}_A \tag{1.3}$$

Transformations can be concatenated if the resulting dimensions match. The transformation chain can be recognized by reading the indices from right to left, e.g.:

$$\mathbf{x}_C = \mathbf{T}_{C\leftarrow B}\mathbf{T}_{B\leftarrow A}\mathbf{x}_A \tag{1.4}$$

A point $\mathbf{x}_A$ in coordinate frame A is first transformed to coordinate system B followed by a transformation from B to the final coordinate system C.

1.3.2 Coordinate Systems

Several coordinate systems are used in the thesis. A world coordinate system is used as a general reference. The ego vehicle coordinate system defines a local coordinate system which is carried along with the motion of the vehicle. Camera models make use of two other coordinate frames – the 3-dimensional camera coordinate frame and the image frame which references to pixel coordinates. RADAR sensors utilize polar coordinates to reference to detected targets. These coordinate systems are explained in detail in the following.

World Coordinate System

When referring to the world reference frame or world coordinate system (Figure 1.4), the $x - y$-plane is defined as the ground plane and the z axis is pointing upwards. Thus, the coordinate system is right handed. One unit in this system is equal to one meter.

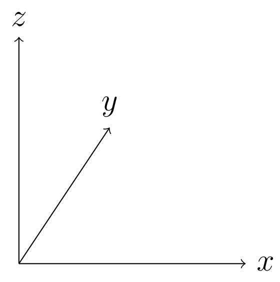


Figure 1.4: World reference frame, right handed, z pointing upwards

Ego Vehicle Coordinate System

The ego coordinate system E is a reference frame which is carried along with the motion of the car. The right handed ego vehicle coordinate system's origin is situated in the middle of the front axle (see Figure 1.5). The y-axis is aligned with the front axle and x points into the driving direction. The z-axis is pointing upwards.

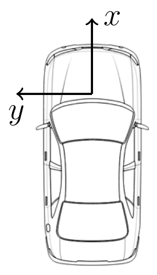


Figure 1.5: Ego Vehicle coordinate system, y-axis aligned with front axle

Camera Coordinate System

The origin of the camera coordinate system C is in its viewpoint with the z-axis pointing away from the camera in the direction of the view as seen in Figure 1.6.

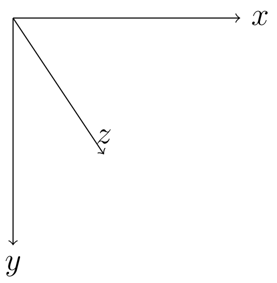


Figure 1.6: Camera coordinate system

Due to a slightly different definition of the coordinate system as being applied by Scaramuzza [Sca07], it has become necessary to introduce the following transformation to be compliant with the camera coordinate definition.

$$\mathbf{T}_{C \leftarrow Scara} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{1.5}$$

The matrix itself is orthonormal, thus $\mathbf{T}_{C \leftarrow Scara}^{-1} = \mathbf{T}_{C \leftarrow Scara}^{\top}$. As the transpose of the matrix is the same as the matrix itself, it can be used for converting 3-dimensional points in both directions, thus $\mathbf{T}_{Scara \leftarrow C} = \mathbf{T}_{C \leftarrow Scara}$.

Image Coordinate System

The image coordinate frame is used in images captured from camera sensors. It can be either centered at the principal viewpoint or at the top left position of the image and references to pixel coordinates. Coordinates are annotated as (u, v) with u pointing to the right and v pointing downwards.

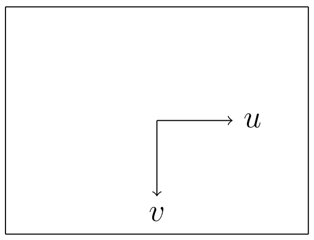


Figure 1.7: Image coordinate system, centered at image middle.

RADAR Coordinate Systems

Based on the principle of RADAR systems, polar coordinates are the utilized coordinate system (Figure 1.8). In general, the canonical coordinate system for RADAR consists of an angle φ, the distance r and the velocity v_r of the object in the direction of the sensor origin. Therefore a target can be stored in a vector $\mathbf{x}_R$ in coordinate system R.

$$\mathbf{x}_R = \begin{pmatrix} \varphi \\ r \\ v_r \end{pmatrix} \tag{1.6}$$

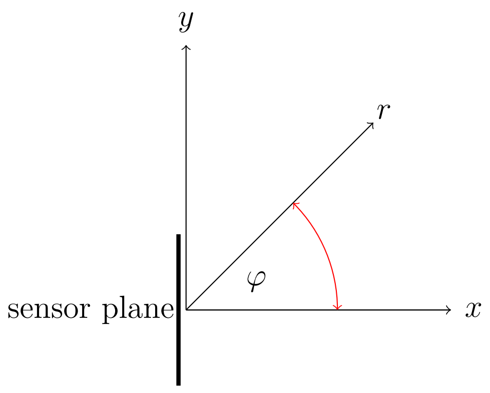


Figure 1.8: RADAR coordinate system of the sensor

At Hella Aglaia, the coordinate system has some historically grown properties. To achieve coordinates independent from the sensor mounting angle, an offset angle is added which turns the x-axis in the backwards direction of the car. Thus, the x-axis of the RADAR coordinate system R is parallel to the ego coordinate system E's x-axis but points in the opposite direction and still originates in the sensor mounting position. Thus, a target $\mathbf{x}_R = (0, r, 0)^\top, r > 0$ would be located directly behind the car aligned with the sensor origin.

1.4 State of the Art

This section is about giving a representative overview of the state of the art techniques for environment perception with focus on vehicle detection. The approaches are separated in approaches that solely rely on RADAR or camera data and, as a last point, a section about fusion approaches combining information from several sensors.

1.4.1 RADAR-based Tracking Approaches

RADAR sensors allow a good estimation of distance and velocities of detected objects. New research on tracking systems solely based on RADAR in the automotive area is barely available. Most papers date back to the early 2000's. The approach in [MHG$^+$06] describes a crash mitigation system which tries to minimize the impact of side crashes to the passenger cabin by extending the dampers by 10 cm within 300 ms and thereby raising the car right before the impact. This deflects parts of the impact energy into a direction below the car and into the harder parts of the door. The system utilizes a 24 GHz RADAR mounted at the height of the indicators in the front fenders.

1.4.2 Camera-based Approaches

In this section, approaches that are based solely on different kinds of cameras are presented. These approaches cover monocular cameras, wide angle cameras, multi-camera setups which include 3D vision systems and surround view cameras based on catadioptric systems. Catadioptric systems are optics that use a mirror to capture viewing angles of up to 360° but introduce strong distortions.

A team from the University of Nevada presented a multi–camera system for crash prediction in 2006 [SBM06]. Cameras in front direction built into the side mirrors and windshield are used to detect front and side traffic, while a camera in the rear window is used for rear traffic. The cameras are especially optimized for night vision. In the first step the camera image is low pass filtered using a Gaussian filter. In the second step horizontal and vertical edges are extracted from scaled versions of the input image and the peaks are used to generate car hypotheses. The group evaluates several features and classifiers for the verification step. Images representing cars are used to estimate an Eigen space to minimize the feature space of a potential vehicle image as proposed in [TP91]. The largest Eigen vectors of this space are used to generate feature vectors. Other sets of features are haar wavelets and truncated haar wavelet sets (removed the wavelets with the smallest coefficients). Furthermore, Gabor features (used for edge detection) are extracted. Additionally, a combination of Gabor and Haar features has been tested. The acquired feature vectors are used to train a neural network and a Support Vector Machine (SVM) classifier. The combination of wavelet and Gabor features classified by an SVM showed the best results.

DaimlerChrysler presented an approach for object recognition and crash avoidance based on stereo camera systems in 2005 [FRBG05]. The cameras record the front view of the vehicle. The ego motion is subtracted from the optical flow of the video stream. Moving objects are eventually segmented using 3D information and optical flow annotations. These moving objects are tracked with Kalman filters [KB61]. As Kalman filters are prone to inaccurate initialization parameters, for each object three filters are set up. The three filters are initialized with $v_1 = 0m/s$ and $v_{2,3} = \pm 10m/s$. The quality of the estimations is finally compared with the following measurement of position and speed. This quality is expressed in the Mahalanobis distance between prediction and measurement and used for weighting each filter. The three filters are then combined to a single prediction based on their weight. According to the paper, the combined multi-hypothesis Kalman filter converges faster than choosing a single filter.

The approach of fitting deformable 3-dimensional object models by projecting them into 2-dimensional images is not new, however, a recent attempt has been presented by [LM09]. The method estimates a model parameter set which describes the shape, orientation and position of the model in 3-dimensional space by fitting the projected model edges to edges in the camera image. The authors argument that the approach of fitting a simple car model is about 20 years old. The original approach itself can be found in [Low91]. With increased computing power and increased image resolution, it is possible to fit complexer models. Thus, they use several car type specific CAD models with different polygon counts with 12 to 80.000 faces. A vehicle model consists of a 3D mesh describing the general appearance, 2D polygons which describe the appearance of parts of the model and a mapping which maps these onto said 3D model. To reduce the parameter space of a general deformable model based on their model database, they use Principle Component Analysis (PCA) and reduce the parameter space to m model parameters represented by the largest Eigen values. On the assumption of a known ground plane, the parameter space is represented by a (x, y)-translation and a yaw angle rotation θ as well as m shape parameters deduced by PCA. The corresponding 2D-projection of the model's initial pose is fit to edges found in the Canny filtered input image [Can86]. Within multiple iterations the distance to the detected edges is minimized. The group was able to fit 6 different vehicle types but fitting required good initial poses which are acquired by manually placing the model on the vehicle in the image.

In [LTH$^+$05], the same model fitting approach is used. There is no assumption of camera motion. Thus, object hypotheses are obtained by motion changes in the camera image. Additionally, the detected models are tracked through several frames by an Extended Kalman filter with an underlying bicycle model.

DaimlerChrysler also showed a bird view system which allows a driver to have a top view of his surroundings on a monitor on his dashboard [ETG07]. Two omnidirectional cameras on the top rear corners of a truck capture the surroundings of the vehicle. These catadioptric systems based on a camera pointing on a convex mirror achieve an opening angle of 360°. As a result, two of these cameras render sufficient to capture the rear environment. The image is back projected on the ground plane around the car which eventually renders a bird view of the environment as seen from the top of the car.

1.4.3 Sensor Fusion Approaches

Sensor fusion approaches are the most recent developments in the area of environment perception. The fusion of several sensors allows to combine advantages of different sensors, e.g. distance and velocity measurements of RADAR and classification abilities of cameras, and reduce disadvantages at the same time. Current research mostly focuses on sensors which provide 3-dimensional information of the environment. Thus, papers that focus on camera-RADAR fusion are rather rare.

One of the recent approaches which utilize sensor fusion is [ZXY12]. The approach combines a Light Detection And Ranging (LIDAR) sensor with camera data. LIDAR sensors measure the time light travels to a certain object point to measure its distance. This allows high resolution distance maps. These maps provide good information for obstacle detection but less for classification. The distance map is used to classify parts of the image to ground plane or obstacles using RANSAC and 3D adjacency. At the same time the image patches are classified to different classes by evaluating color histograms, texture descriptors from Gaussian and Garbor filtered images as well as local binary patterns. Resulting in 107 features, each patch is classified by a Multi-Layer-Perceptron to different categories. The size of the object determined by the LIDAR obstacle estimation as well as labels from the image analysis are passed into a fuzzy logic framework which returns three labels – environment, middle high or obstacle. The fusion approach reaches better classification results than the individual sensors.

In [VBA08] a framework for self-location and mapping is presented. This approach is based on an occupancy grid that is used to fuse information from RADAR and LIDAR. An occupancy grid discretizes the environment space into small two dimensional grid tiles. The LIDAR sensor is used to estimate a probability of a grid cell to be occupied. A high probability means there is an obstacle in the respective grid cell. Moving objects are detected through changes in the occupancy of grid cells. Basically, if a cell is occupied at a time point, and in the next time step, the adjacent cell is detected to be occupied, this is assumed as a motion. As RADAR data is relatively sparse compared to LIDAR measurements, it is used to verify or reject motion estimations based on LIDAR data.

A similar method of fusing short range RADAR and laser measurements is proposed in [PVB$^+$09]. The system covers more performance optimizations than [VBA08]. Instead of estimating a probability representation, the grid cells are either occupied or empty based on the number of measurements of the laser scanner for the respective cell. Adjacent cells that are occupied are merged to an object. A Kalman filter [KB61] is used to track the detected objects. Unfortunately the fusion process of RADAR and laser data is not described in detail.

The approach proposed in [ABC07] shows a fusion of RADAR and camera in which the camera is used to verify and optimize RADAR object tracks. The camera is oriented to the front. The center points of RADAR objects are projected into the image based on the camera calibration. The symmetry of image sections around the projected point is estimated. Ideally, the symmetry is large around the RADAR hypothesis as front views and rear views of vehicles are symmetric. Searching for higher symmetry values in a predefined environment around the RADAR hypothesis is used to correct the position estimate. However, this approach does only work for scenarios in which vehicles are viewed from the rear or front.

Chapter 2

Test Bed Setup

This chapter describes the general hard- and software setup of the test bed vehicle. The test bed is an Audi A8 equipped with an industry computer and several sensors. These include a front camera and two RADAR sensors mounted at the front bumper observing front and side traffic. Furthermore, side short range ultrasound sensors are mounted to observe the sides near the doors.

Cassandra, the used software framework for prototyping of algorithms, recording and replaying of test drive data, is presented in section 2.2. The process of choosing appropriate cameras for the project is demonstrated in section 2.3. Section 2.4 introduces the calibration process of camera systems and compares a pin-hole model with distortion parameters with the more recent Scaramuzza camera model [SMS06a]. Finally, the hardware interconnections and the recording and replaying of test drive data in the Cassandra Framework is explained.

2.1 Hella RADAR Sensors and Applications

Hella is one of the biggest global suppliers of automotive electronics and lighting solutions. One of their products is a low-cost 24 GHz RADAR sensor [HEL12]. The sensor allows the detection of targets in a distance of up to 100 m with velocities of ± 70 m/s in cycles of 50 ms. The maximum field of view is 165°. Not only does Hella provide the sensor itself, but also several driver safety system applications. These systems are in use at several OEMs. Blind Spot Detection (BSD) warns drivers of unsafe conditions when trying to change lanes. Traveling on the highway is made safer by the Lane Change Assistant (LCA), which allows warning distances of 70 m when changing lanes. A similar application - Rear Pre-Crash (RPC) - observes rear traffic and detects potential collisions. This information can be used by OEMs to prepare for impacts. Safety measures that could be executed are for example the fastening of seat belts or further preparations of air bags. Rear Cross Traffic Alert (RCTA) secures situations in which the driver backs out of parking spaces and warns the driver about oncoming rear traffic.

2.2 Video & Signal Processing Framework Cassandra

Cassandra is a C++ development environment developed at Hella Aglaia [Hel13]. It supports both rapid prototyping and product development. It has been released for external use in 2013. The framework will be used throughout the whole thesis to implement most of the concepts explained. Its main focus is computer vision in the automotive area. It supports a variety of cameras and video input, as well as Controller Area Network (CAN) hardware for capturing and sending CAN messages. Furthermore a lot of OpenCV's [Its13b] functionality is already available. OpenCV is an open source computer vision library written in C and C++ that provides optimized algorithms in the area of computer vision and machine learning.

Instead of implementing the whole processing chain of computer vision algorithms at once, Cassandra allows to break up the processing chain into replaceable and reusable blocks called stations. A station can contain input ports for input data, e.g. CAN messages or single images from a video, and output ports for the processed data. Cassandra also provides several predefined stations for camera calibration and image processing based on OpenCV. Each station has several adjustable parameters for the algorithm it represents.

A simple example of such a processing chain is an edge detection algorithm. The input is a RGB video stream from a Logitech Webcam Pro 9000, thus, it has to be converted to uint8 gray scale and then fed into the Canny edge detection algorithm [Can86]. The simplest approach to achieve this is to use a player station which sends out the video one frame at a time to the Canny filter station. The final output is a binary image that highlights edges.

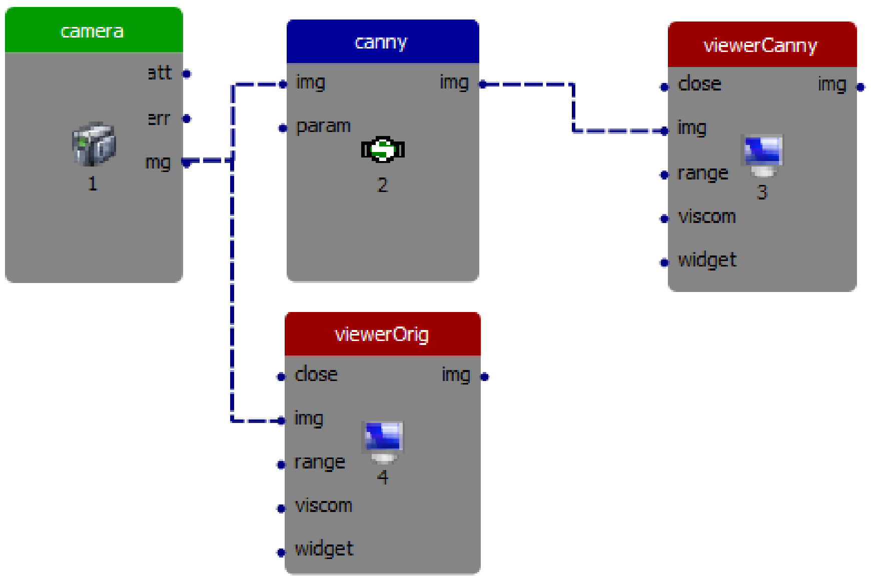
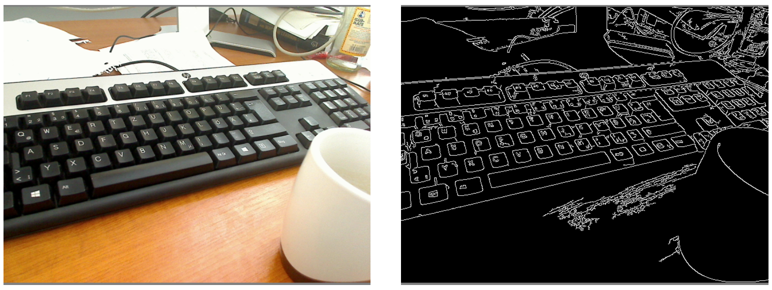


Figure 2.1: Cassandra processing chain design for a simple Canny filter

This setup has been created as a Cassandra processing chain in which the stations are linked together (Figure 2.1). At first, the camera station grabs an RGB image from the web-cam and sends it to the Canny filter station and to a viewer which shows the image in a new window. The uint8 gray scale conversion is done implicitly by the Canny station when connecting an RGB image port to a port which requires uint8 input. In the Canny station, the filter processes the received image and creates a new image which contains the detected edges. This is sent to the output port which connects to another viewer that finally shows the edge image (Figure 2.2).

2.3 Choice of Cameras

Choosing the right camera for computer vision tasks is a sophisticated process. Requirements strongly depend on the application. The cameras should be used to support RADAR tracking. This requires the cameras to have a similar field of view as the RADAR sensors which have a wide opening angle of 120° for the FSR application. The side cameras should both capture most of the radar's view but also provide enough image quality to detect cars in a distance of up to 30 m. Compared to front or rear cameras, videos captured from a side position feature



(a) Original image **(b)** Canny filtered edge image

Figure 2.2: Original and processed image from a Canny filter processing chain in Cassandra

a higher amount of movements as they are perpendicular to the driving direction. Fast movements often result in blurred images which render them less suitable for image processing. Therefore, the cameras should contain relatively big pixels on the sensor plane which renders the camera able to capture more incoming light and reduces noise and blurring in the output image.

Unfortunately, the most practical available mounting positions are under the side mirrors, which requires the cameras to be positioned outside of the vehicle. This adds further points to the requirements list – the camera has to be weatherproof and small enough to be mounted below the mirrors in appropriate housings.

The Cassandra Framework, which acquires data from both RADAR and camera, supports different types of cameras (USB, FireWire). Currently Cassandra is limited to the Windows platform.

Another point, though often underestimated, is the synchronization of data from different sensors. The RADAR tracker sends new data every 50 ms which results in a rate of 20 Hz. This is taken as a reference for the camera. The test bed features a CAN/LIN Input/Output - Module (CLIO) that sends the RADAR data collected from the CAN over FireWire and tags it with a FireWire time stamp. If the camera is connected to the same FireWire bus, images from the camera can be tagged by the same clock and assigned to the collected RADAR data. The use of FireWire adds a throughput limitation (800 Mbit/s) to the system. Both camera data and the RADAR data have to be passed over the same bus and should not exceed its transfer speed limit. This has to be taken into account when considering image resolution and frame rate.

Final Requirements:

- Two color cameras, one for each side
- Operating system support: Windows XP 32bit and Windows 7 64Bit
- Optics viewing angle: $\approx 120^\circ$ horizontal, central focal point
- Horizontal resolution: at least 700 px (1 px $\geq$ 10 cm in 30 m distance in the image center)
- weather-proof housing which is small enough for side mirror mounting
- sensitive chip with low noise
- Connection: FireWire 1394b (800 Mbit/s)
- $\geq$ 20 fps
- Budget: max 1000 € per side

To match both technical and financial requirements, the AVT F-046C from Allied Vision Technologies [All12] combined with a wide angle lens from GOYO OPTICAL Inc. [GOY03] have been chosen. The optics feature a 103.6° opening angle (132° diagonal) with a focal length of 3.5 mm. The camera itself provides a resolution of 780 px × 580 px with up to 62 fps. The 1/2 inch CCD sensor captures enough light to produce sharp images even with high dynamics. The transfer rate of two uint8 gray image streams from the cameras result in in a transmission rate of

$$780 \text{ px} \times 580 \text{ px} \cdot 20 \text{ Hz} \cdot 8 \text{ Bit} \cdot 10^{-6} \cdot 2 = 144.7 \text{ Mbit/s}$$

plus protocol overhead which leaves enough bandwidth for other applications on the FireWire bus.

The final size of the setup is 31 mm × 31 mm × 78.5 mm but has to be sealed in a custom built housing. Costs for camera and optics are 747 € for each side resulting in 1494 € in total.

The setup of the FireWire cameras created a few problems as the utilized version of the Cassandra Framework only supports cameras through the Unibrain interface [Uni13]. Unibrain is a FireWire stack for Windows with the goal to unify FireWire interfaces of cameras and other devices. Those devices usually ship

Product		Price
Allied Vision Technologies AVT F-046C		620 €
Resolution:	780 x 580	
Sensor Diagonal:	1/2 inch Sony ICX415	
Connector:	FireWire 1394b	
Size:	44.8 mm × 29 mm × 29 mm incl. connectors	
Mass:	80g	
Optics Connector:	C-Mount	
GOYO OPTICAL INC. GM23514MCN		127 €
Angle of View (H×V×D):	$103.6^\circ \times 76.00^\circ \times 132.1^\circ$	
Focal Length:	3.5 mm	
Aperture:	f/1.4 to close	
Size:	∅31 mm ×30.5 mm	
		747 €

Table 2.1: Camera and optics for the test bed

their own drivers for the Windows platform which renders concurrent operation of different devices complicated. However, the cameras and Unibrain, despite being officially supported, rendered prone to timing issues due to the lengths of the cables placed in the test vehicle. These problems were partly overcome by increasing the allowed time for a device to register at the Unibrain stack.

2.4 Camera Calibration and Evaluation of Camera Models

This section gives an overview about two geometric camera models and their respective calibration and why these are needed for object detection and tracking. In general, there exists a wide range of different camera hardware – different kinds of sensors and, more important, different kinds of optics. Beside standard lenses, there is a variety of wide angle and fish-eye lenses that capture a wider area of the environment but introduce stronger distortions to the final image. Lately in computer vision, catadioptric systems have been introduced, that have an even wider opening angle of up to 360° compared to wide angle lenses which can reach up to around 220°.

Especially when trying to identify objects in such images, distortions introduced by the optics can interfere with the actual detection process, e.g. straight lines in the real environment become bent in the images captured by the camera. Additionally, the form of an object in the might look different through the distortions. Distortions are geometric projection errors that are generated by the optics of the system. Hence, for object detection, knowing the geometries that cause such distortions to undistort images and remove the influence of the distortions can greatly improve detection performance. This process is also called rectification. Most camera lenses introduce two kinds of distortions, radial distortions emitting from the center of the image and tangential distortions. Radial distortions are

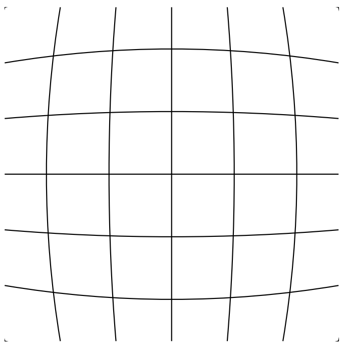


Figure 2.3: Radial barrel distortion around the image center

radially symmetric around the image center as most lenses are manufactured radially symmetric. Especially for wide angle lenses the zoom level in the middle of the image is usually higher than at the borders which is called barrel distortion (Figure 2.3). Tangential distortions, for example, can be seen in images taken from a camera which is positioned behind the windshield of a car. The bottom part of the image might exhibit a higher zoom level than the upper part because of the different distances to the windshield and the different ways the light rays take through the glass.

2.4.1 Intrinsic Calibration

A camera model defines a geometric system that describes how a ray of light hits the camera sensor. Most of the models require a single viewpoint which means, that all incoming light rays, restricted by the geometries of the optics, intersect in a defined point, the view point or focal point. In the following pages two models will be explained in detail. The standard pinhole model with radial distortion parameters available in OpenCV [Its13b] can be used for most cameras. With wide angle optics however, a more sophisticated model may be needed. The Scaramuzza model [SMS06a, Sca07] can both model wide angle lenses and catadioptric systems.

The term calibration refers to defining parameters for a camera model. One differentiates between extrinsic and intrinsic parameters. Intrinsic parameters are parameters of the camera model, e.g. the focal point distance or the center of the image. Extrinsic parameters define the position (translation) and orientation (rotation) of the camera with respect to a reference coordinate frame. Altogether there are three coordinate systems – the world coordinate frame, which defines a general reference system, the camera coordinate system which originates in the viewpoint of the camera and the image coordinate system which is a two dimensional system describing the position of a pixel in the actual image. Details about these reference systems can be found in section 1.3.

OpenCV Camera Model

The camera model used in the famous computer vision library OpenCV [Its13b] dates back to a paper from Microsoft Research presented in 2000 [Zha00]. It has been extended several times to model different kinds of distortions. The model is defined through a simple pinhole-model and a distortion function that maps the ideal pinhole projection image points to distorted image points based on distortion parameters.

The camera matrix $\mathbf{A} \in \mathbb{R}^{3\times 3}$ contains the following parameters for transforming a point $(X, Y, Z)^\top \in \mathbb{R}^3$ from the cameras reference 3-dimensional pinhole coordinate system to the final image $(u, v) \in \mathbb{R}^2$ with f_x, f_y being the focal lengths in pixel units and (c_x, c_y) being the principal point. The scaling factor s is equal to z in this case.

$$s\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} = \underbrace{\begin{pmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{pmatrix}}_{\mathbf{A}} \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} \tag{2.1}$$

Cameras in general are not distortion free, thus, the OpenCV model introduces up to 3 radial distortion parameters k_i and two parameters p_j to account for tangential distortions. The distortion function repositions the ideal pinhole pixel position to the distorted pixel position. Distortion modeling is introduced in the following way.

$$x' = x/z \tag{2.2}$$
$$y' = x/z \tag{2.3}$$
$$r^2 = x'^2 + y'^2 \tag{2.4}$$
$$x'' = x'(1 + k_1r^2 + k_2r^4 + k_3r^6) + 2p_1x'y' + p_2(r^2 + 2x'^2) \tag{2.5}$$
$$y'' = y'(1 + k_1r^2 + k_2r^4 + k_3r^6) + p_1(r^2 + 2y'^2) + 2p_2x'y' \tag{2.6}$$
$$u = f_x \cdot x'' + c_x \tag{2.7}$$
$$v = f_y \cdot y'' + c_y \tag{2.8}$$

The coordinates x' and y' are the ideal pinhole coordinates and x'' and y'' their respective repositioning based on the distortion parameters. Unfortunately, the reverse operation $(u, v) \mapsto \lambda(X, Y, Z)$ is not as trivial as the distortion polynomial has to be solved for x' and y'.

If the world reference frame (X_w, Y_w, Z_w) is different from the camera coordinate frame, another transformation has to be added. Homogeneous coordinates are used for this transformation as this allows including rotations and translation in a single matrix. The parameters r_{ij} combine rotations along the three main axes and an additional offset is presented by $(t_1, t_2, t_3)^\top$.

$$\mathbf{T}_{C\leftarrow W} = \begin{pmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.9}$$

$$\begin{pmatrix} X \\ Y \\ Z \\ 1 \end{pmatrix} = \mathbf{T}_{C\leftarrow W} \begin{pmatrix} X_w \\ Y_w \\ Z_w \\ 1 \end{pmatrix} \tag{2.10}$$

The acquired camera coordinates (X, Y, Z) in the camera reference frame are then further processed as described in eq. (2.1).

Scaramuzza Camera Model

The simple pin-hole model, even incorporating distortions, is mathematically not able to model fields of view with angles greater than 180°. In practice, problems occur with much smaller angles already. In 2006/2007 Davide Scaramuzza presented a new camera model in a paper and his PhD thesis [SMS06a, Sca07] which allows higher viewing angles. The proposed approach describes a general camera model for mirror lens optics (catadioptric, fig. 2.4c) and wide angle lenses (fish–eye, figs. 2.4a and 2.4b). Those systems exhibit strong radial distortions. In fish–eye lenses, this can be recognized when the middle of the image has a higher zoom factor than the edges. Resulting from these distortions, straight lines become bent and relative sizes change.

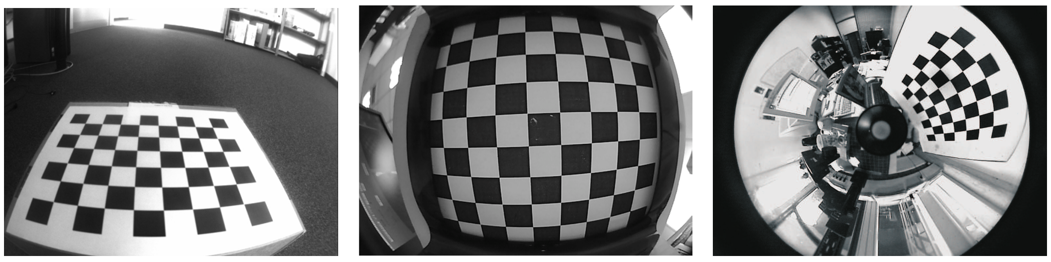

(a) Wide angle optics **(b)** Fish-eye optics **(c)** Catadioptric system

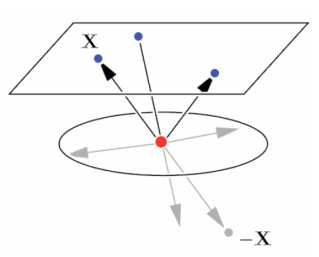
Figure 2.4: Images taken with different optics

Instead of simply repositioning the pixels on the image plane with a distortion function, Scaramuzza's model computes a vector $(x, y, z)^\top \in \mathbb{R}^3$ emanating from the single viewpoint to an image sphere pointing in the direction of the incoming light ray for each pixel position $(u, v) \in \mathbb{R}^2$. The reference frame (u, v) originates in the center of the image. The difference of the central projection used in the pin-hole model and Scaramuzza's approach can be seen in Figure 2.5.

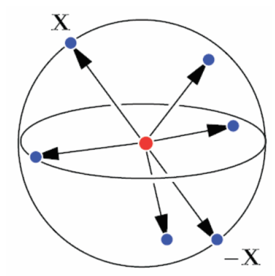
Scaramuzza utilizes the properties of the spherical projection to estimate a function $g(\rho)$ so that a point $(X, Y, Z) \in \mathbb{R}^3$ in camera coordinates can always be represented as a point on the ray $\lambda(u, v, g(\rho))^\top$ with $\lambda \in \mathbb{R}$ being an arbitrary scaling factor. The function $g(\rho)$ is defined as a Taylor polynomial with coefficients a_i that define the intrinsic parameters with $\rho = \sqrt{u^2 + v^2}$ being the Euclidean distance of the pixel position (u, v) from the image center. This function $g(\rho)$ models the radial distortion at the same time.

$$g(\rho) = a_0 + a_1\rho + a_2\rho^2 + a_3\rho^3 \dots a_n\rho^n \tag{2.11}$$

$$\begin{pmatrix} X \\ Y \\ Z \end{pmatrix} = \lambda \begin{pmatrix} u \\ v \\ g(\rho) \end{pmatrix} \tag{2.12}$$



(a) Central projection limited to points that lie in front of the projection plane



(b) The spherical model distinguishing two scene points lying on opposite half-lines

Figure 2.5: Perspective projection limited to view angles covered by the projection plane and spherical projection which covers the whole camera reference frame space [SMS06a]

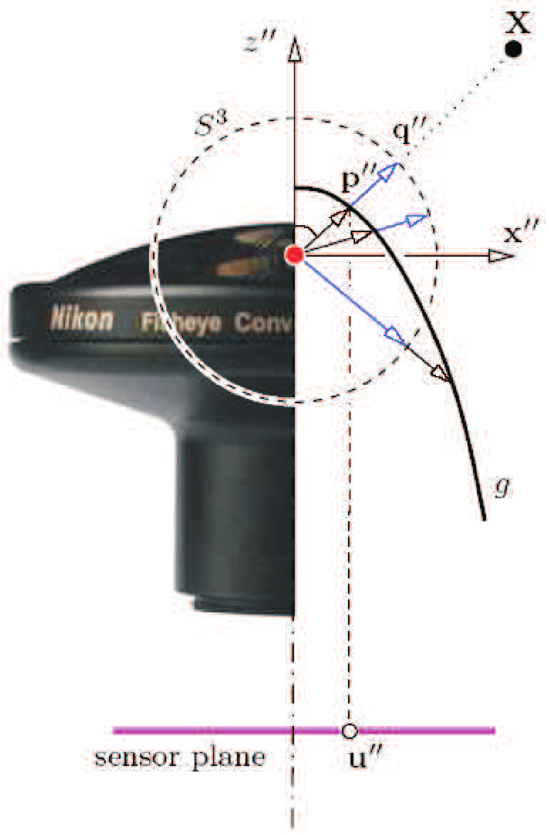


Figure 2.6: Scaramuzza model for a fish–eye lens [Sca07]

Certainly, to project a point (X, Y, Z) in camera coordinates onto the image, eq. (2.12) has to be solved for u and v which requires solving the polynomial of $g(\rho)$. As this is computational inefficient, Scaramuzza estimates an inverse Taylor polynomial $f(\theta)$ which describes the radial positioning on the image plane based on the angle θ of a light ray to the z axis.

$$f(\theta) = b_0 + b_1\theta + b_2\theta^2 + b_3\theta^3 ... b_n\theta^n \tag{2.13}$$
$$r = \sqrt{X^2 + Y^2} \tag{2.14}$$
$$\theta = \operatorname{atan}(Z/r) \tag{2.15}$$
$$u = \frac{X}{r} f(\theta) \tag{2.16}$$
$$v = \frac{Y}{r} f(\theta) \tag{2.17}$$

The model applied on a fish-eye lens can be seen in Figure 2.6. Further details of the model can be found in [SMS06a, Sca07, SMS06b]. During the calibration process, several parameters are estimated (Table 2.2).

Description	**Parameters**
Scaling of sensor to image plane	c, d, e with $\begin{pmatrix} u' \\ v' \end{pmatrix} = \begin{pmatrix} c & d \\ e & 0 \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix}$
Taylor polynomial g	a_i
Image projection Taylor polynomial	b_i
Image center	(x_c, y_c)

Table 2.2: Parameters of an intrinsic Scaramuzza calibration

Calibration with OpenCV

The calibration process of OpenCV is fairly automated and requires only XML files that describe which model parameters to be used and images or a video stream of the chosen pattern in different poses. Both circle patterns and checkerboard patterns are supported by OpenCV. The process is described in detail in [Its13a]. The pattern which is used to calibrate the camera is a checkerboard pattern with 5 cm $\times$ 5 cm squares. The calibration was done on 30 static images showing the checkerboard pattern in different poses (Examples in Figure 2.7).

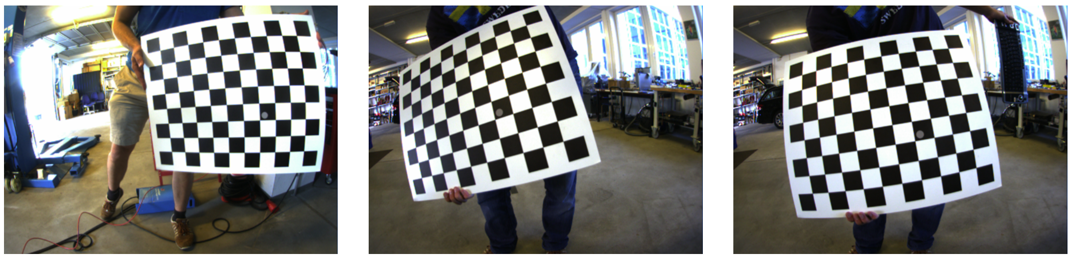


Figure 2.7: Calibration images

For calibration, three radial parameters are estimated and tangential distortion is neglected. The number of radial parameters has been chosen as it still allows to solve $(u, v) \mapsto \lambda(X, Y, Z)$ using Newton's method. This is necessary for the calculation of light ray directions. To compute the incoming ray for a given image point is necessary for optical flow estimation methods which are introduced in chapter 3. OpenCV allows more distortion parameters which might result in a better calibration but come with higher computing costs.

Calibration with Scaramuzza

Scaramuzza provides a Matlab toolbox [SMS06b] which allows the calibration from static images showing a checkerboard pattern. The same images that were used for the OpenCV calibration are used for the Scaramuzza calibration. The calibration process is described in [Sca13]. In the first step the checkerboard corners are semi-automatically extracted from the input images. After this, a first estimate of the Taylor polynomial is computed. The use of 4 coefficients for the polynomial showed the best results in the error evaluation. From this first calibration the principal point is optimized. This step is required to compensate for optics that are not exactly placed over the sensor center. In a further refinement step the calibration is optimized for the new principal point. Figure 2.8 shows the evaluated Taylor polynomial a_i of the left side camera calibration and the corresponding angle of the ray to the z-axis of the camera system. Additionally, Figure 2.9 shows the re-projection of the pattern coordinates into the image. The red circle represents the estimated principal point projected into the image.

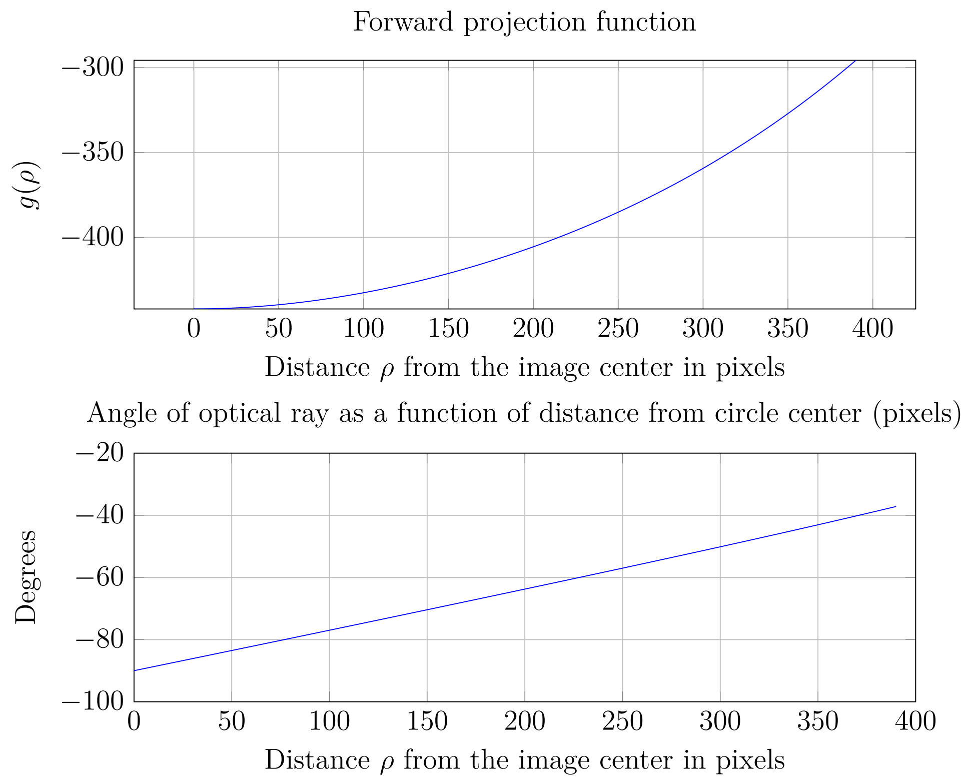


Figure 2.8: Scaramuzza calibration results of the left camera calibration: $g(\rho)$ and the angle of the rays from the z-axis in relation to the distance ρ from the image center

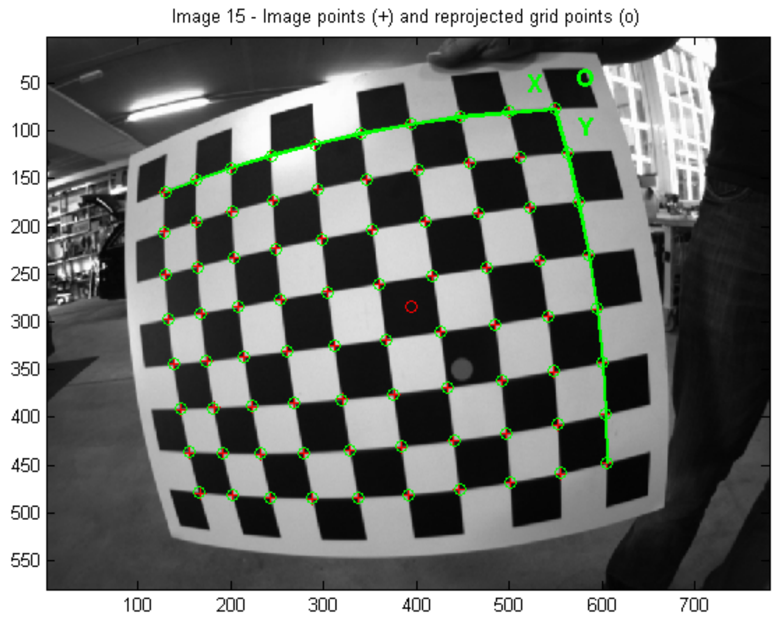


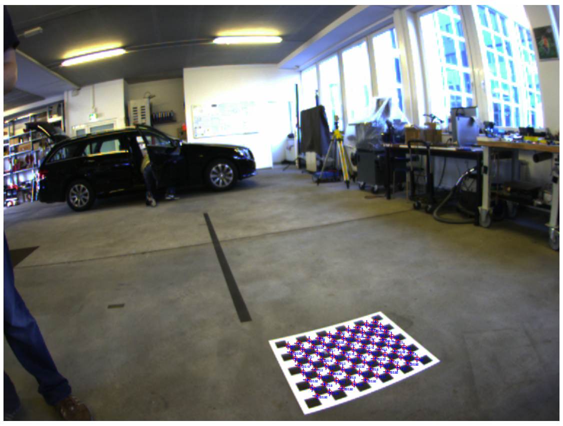
Figure 2.9: Re-projection of the extracted 3–dimensional corner world coordinates back into the image. Red crosses represent the detected corner points and green circles their respective re-projection

2.4.2 Extrinsic Calibration

The translation of the camera can be roughly measured with standard measurement tools. The rotation of the camera, however, is computed by estimating the transformation to a reference coordinate system. Both the OpenCV and Scaramuzza model provide functions to estimate a transformation from given world coordinates in reference system P and corresponding image coordinates I to the camera coordinate system C based on the camera model. The reference frame P is represented by a checkerboard aligned with the ground plane. Its axes are parallel to the vehicle's ego coordinate system (Figure 2.10).

The algorithms for estimating extrinsic parameters from a pattern have been implemented in Cassandra. In the first step the image coordinates and corresponding checkerboard coordinates are extracted. The method `findChessBoardCorners` for automatic checkerboard detection provided by OpenCV is robust but fails in some cases. However, to detect the checkerboard patterns in the distorted images of the cameras, the algorithm for automatic detection of checkerboards on blurred and distorted images from [RSS08] has been used. This approach performs better in areas of stronger distortions. After extracting the corner points of the pattern, the corners are refined using OpenCV's `cornerSubPix` method with a window size of 4×4 pixels to acquire sub–pixel accurate corner positions.

Pairs of image coordinates (u_i, v_i) and their respective points on the checkerboard $(x_i, y_i, 0)$ are used to estimate a transformation matrix $\mathbf{T}_{C \leftarrow P}$ which transforms pattern coordinates into camera coordinates of the respective camera model. Both OpenCV and Scaramuzza provide methods to estimate this transformation. Scaramuzza's extrinsic calibration method has been implemented based on the Matlab code of the Scaramuzza Calibration Toolbox [SMS06b] and the description in [SMS06a, Estimation of the extrinsic parameters, p. 31] in Cassandra. OpenCV provides this functionality with the `solvePnP` function which computes the transformation from the 3D-2D point correspondences given an intrinsic calibration. Assuming that the world coordinates lie in the same plane as the ego coordinate system E and that coordinate system P's axis are parallel to the car reference frame E the rotation angels can be computed. In this method, the angles are defined as concatenated rotations $\mathbf{R}_z(\theta_z)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)$.


(a) Camera view of the pattern aligned with the car.

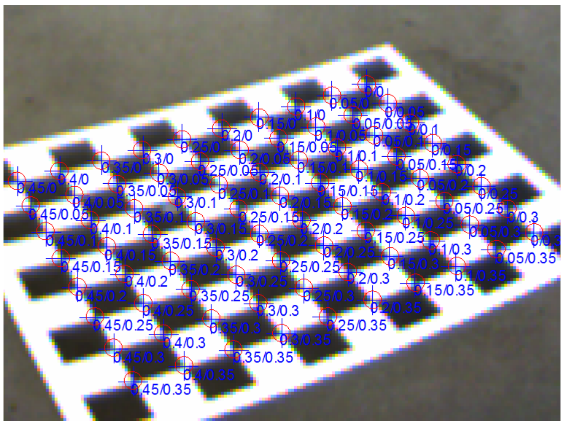

(b) Zoomed pattern with annotated pattern coordinate system.

Figure 2.10: Extrinsic calibration using a checkerboard pattern – Crosses represent the extracted corners from the image and circles the reprojected pattern coordinates

Assuming that all points of the chessboard pattern reference frame lie in a plane, the z–coordinate is set to 0. The size of each square is 0.05 m $\times$ 0.05 m – the top row is used as x-axis and the right column as y-axis. For each pattern point $(x, y, 0)$ the corresponding image point (u, v) is detected and used to estimate the extrinsic transformation matrix for homogeneous coordinates in the following form:

$$\mathbf{T}_{P\leftarrow C} = \begin{pmatrix} & & & t_1 \\ \multicolumn{3}{c}{\mathbf{R}_z(\theta_z)\mathbf{R}_y(\theta_y)\mathbf{R}_x(\theta_x)} & t_2 \\ & & & t_3 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.18}$$

The matrices in eqs. (2.19) and (2.20) represent the transformation from the pattern reference system to the camera coordinate system.

$$\mathbf{T}_{C_{\text{scara}}\leftarrow P} = \begin{pmatrix} -0.9548 & 0.2894 & -0.0672 & 0.4460 \\ 0.1242 & 0.1832 & -0.9751 & 0.5533 \\ -0.2699 & -0.9394 & -0.2109 & 1.4497 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.19}$$

$$\mathbf{T}_{C_{\text{ocv}}\leftarrow P} = \begin{pmatrix} -0.9559 & 0.2847 & -0.0708 & 0.4579 \\ 0.1314 & 0.1997 & -0.9709 & 0.5315 \\ -0.2623 & -0.9375 & -0.2284 & 1.4591 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.20}$$

The inverse matrices correspond to the inverse transformations.

$$\mathbf{T}^{-1}_{C_{\text{scara}}\leftarrow P} = \mathbf{T}_{P\leftarrow C_{\text{scara}}} = \begin{pmatrix} -0.9548 & 0.1242 & -0.2700 & 0.7486 \\ 0.2895 & 0.1833 & -0.9395 & 1.1314 \\ -0.0672 & -0.9752 & -0.2110 & 0.8754 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.21}$$

$$\mathbf{T}^{-1}_{C_{\text{ocv}}\leftarrow P} = \mathbf{T}_{P\leftarrow C_{\text{ocv}}} = \begin{pmatrix} -0.9560 & 0.1315 & -0.2623 & 0.7507 \\ 0.2847 & 0.1998 & -0.9376 & 1.1314 \\ -0.0708 & -0.9710 & -0.2284 & 0.8819 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{2.22}$$

Using the method presented in [Sla99], the angles θ_x, θ_y and θ_z are extracted from the transformation matrices (Table 2.3). Furthermore, the pattern represents the ground plane. Therefore the translation in $\mathbf{T}_{P\leftarrow C_{\text{ocv}}}$ and $\mathbf{T}_{P\leftarrow C_{\text{scara}}}$ represents the camera position in pattern coordinates. Accordingly, the z-coordinate of the translation corresponds to the height of the camera. The computed angles can be used as a good starting point for the extrinsic calibration.

Using this extrinsic calibration and projecting the RADAR targets back into the image returns reasonable results which can be seen in Figure 2.11.

	OpenCV	**Scaramuzza**
θ_z	76.76°	77.79°
θ_y	16.34°	16.31°
θ_x	4.06°	3.85°
Z	0.8819	0.8754 m
Z measured	0.905 m	
Y measured	0.97 m	
X measured	−0.79 m	

Table 2.3: Extrinsic parameters computed from a single image with OpenCV and Scaramuzza Camera models

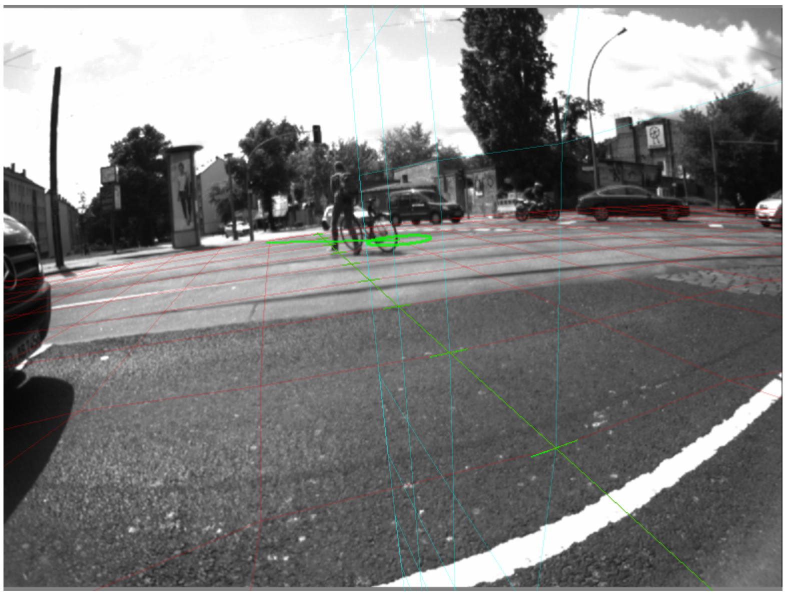


Figure 2.11: A bicycle tracked by radar visualized in the camera image, thin lines represent the car's ego coordinate system projected into the image

The Cassandra graph that computes transformations based on detected patterns for both Scaramuzza and OpenCV can be found in Figure C.1 in the appendix on page 104.

2.4.3 Results

To evaluate both models on accuracy, transformations $\mathbf{T}_{C \leftarrow P}$ are computed for several different images as described in section 2.4.2. The pattern reference points are transformed into 3-dimensional camera coordinates and are reprojected back into the image. Finally, these reprojected points (u_i', v_i') are compared to the detected image coordinates (u_i, v_i) extracted by the pattern detector. To measure the performance, error criteria are defined. The Euclidean distance e_i is used to define the error from the detected image coordinate of the pattern (u_i, v_i) in the original image to the reprojected point (u_i', v_i') using the respective camera model. From those, average error, variance and the accumulated square error over all n pattern points in one image are computed.

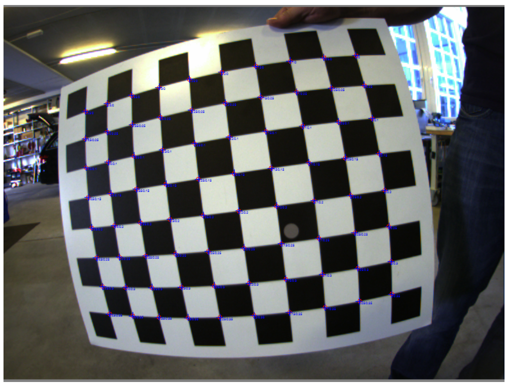
$$e_i = \sqrt{(u_i - u_i')^2 + (v_i - v_i')^2} \tag{2.23}$$

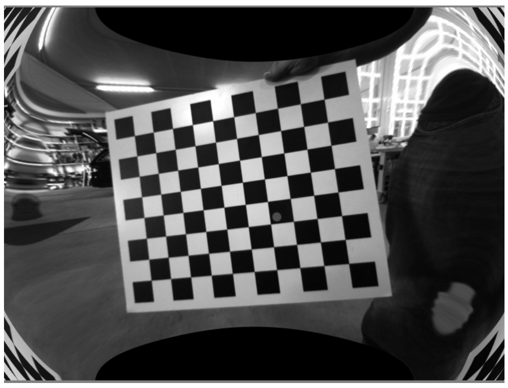
$$\overline{e} = \frac{1}{n} \sum_{i=1}^{n} e_i \tag{2.24}$$

$$\sigma^2 = \frac{1}{n} \sum_{i=1}^{n} (\overline{e} - e_i)^2 \tag{2.25}$$

$$E = \sum_{i=1}^{n} e_i^2 \tag{2.26}$$

The results are shown in Tables B.1 and B.2 on pages 102 and 103 in the appendix for both left and right camera. At first it looks like OpenCV performs slightly better than the Scaramuzza model. Both models perform with good accuracy. Even though it looks like the OpenCV model is the better choice, the Scaramuzza model has an unelectable advantage – its performance in approximating the geometries close to the image borders. The image from Figure 2.12a was undistorted with both models in Figures 2.12b and 2.12c. When undistorting the camera image based on the intrinsic model, OpenCV introduces strong distortions close to the image borders as the radial correction does not generalize well in these sections. This effect might be introduced by the higher order polynomial which OpenCV approximates for the distortion model. Both models remove the distortions in the middle of the image very well. Lines become straight and the


(a) Original distorted image from camera


(b) Undistorted image from OpenCV which introduces new distortions near the image borders

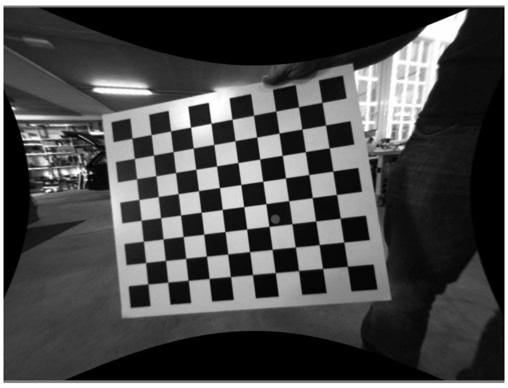

(c) Undistorted image from Scaramuzza which is distortion free.

Figure 2.12: Comparison of undistorted images from Scaramuzza and OpenCV model

fish-eye effect is removed. When looking at the buildings in the top right corner, it is notable that the Scaramuzza approach fixes the outer parts significantly better while OpenCV introduces mirroring and strong distortions in the outer parts. This can be overcome by using more radial distortion parameters ($k_4 - k_6$). This is supported by OpenCV but makes the calculation of the direction of a light ray even more computational inefficient. The Scaramuzza approach, however, allows efficient transformations from image space to camera space and from camera space to image space. Furthermore it is already used internally in other projects at Hella Aglaia and supports the potential switch to optics with an even wider angle as well as catadioptric systems. Therefore, this model is used for all further work.

Overall both models have their advantages and disadvantages which are summarized in the following.

OpenCV Advantages

- Automatic calibration process on both video and static images with a variety of supported pattern types, e.g. checkerboards or circle boards
- C/C++ implementations available
- Models tangential distortion parameters

OpenCV Disadvantages

- Does not generalize well in border regions for wide angle lenses if only used with 3 radial distortion parameters
- No computing-efficient way to estimate the direction of the light ray which hits the image plane
- Generally restricted to opening angles less than 180°

Scaramuzza Advantages

- Both transformations image to camera coordinates and camera to image are efficiently computable
- Good generalization near image borders on wide angle lenses

- C implementation for basic transforms available

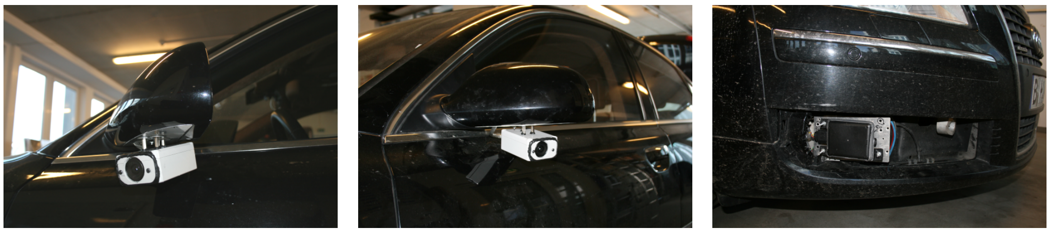
Scaramuzza Disadvantages

- Only semi-automatic calibration process on static images in Matlab available which requires a license of the Matlab Optimization Toolbox
- No tangential distortion modeling

2.5 Hardware Setup

This section describes the general setup of the hardware as well as the interconnections between processing units and sensors. Furthermore, the sensor mountings are explained.

2.5.1 Sensor Mounting



(a) Camera with housing mounted to mirror **(b)** Camera with housing mounted to mirror **(c)** Side radar sensor mounted to bumper

Figure 2.13: Sensor mountings

Protecting the cameras from bad weather conditions is crucial. This fact made it necessary to build custom housings which have been attached to metal plates mounted to the bottom of the side mirrors. The housings were made from PVC and sealed with silicone which can be seen in figs. 2.13a and 2.13b. The RADAR sensors are attached to the front bumper 75° to the left and right as shown in fig. 2.13c. They cover an area of 120°. The exact mounting parameters of the RADAR sensors are shown in Table 2.4.

x_s:	680 mm
y_s:	±725 mm
z_s:	348 mm (related to street level)
angle:	$\pm 75^\circ$ (0° conforms to straight front direction)
opening angle:	$\pm 60^\circ$
pitch angle to top:	$3 - 5^\circ$

Table 2.4: Mounting positions of RADAR sensors

The cameras have been mounted so that they cover as much of the radar view as possible. Hence, the camera's yaw angle is rotated about 10° to the front. Furthermore, areas that usually do not contain objects should not be captured, thus, the pitch angle is about 10° to the ground floor direction to capture less sky area. A top view with the sensor overlap and a side view of the mounting positions can be found in Figures A.1 and A.2 in the appendix on pages 99 and 100.

2.5.2 Telematics and Data Flow

Instead of a spare wheel, most of the processing hardware and the main connections are situated in the trunk. Figure 2.14 shows the setup. The main processing unit (1) is an industry PC featuring an Intel Core i7 running Windows XP and Windows 7 in dual boot mode. It includes a CAN XXL Board and FireWire interconnections. CAN data can be directly recorded through the CAN XXL Board or synchronized to the FireWire clock by the CLIO device (4). The latter is directly connected to the CAN Hub (5). In the current setup it captures both data from the vehicle CAN and the private CAN of the RADAR sensors.

The complete data flow is presented in Figure 2.15. The RADAR sensors are separated into master and slave sensor. The master sensor includes a Communication Processor (KP) which is provided with raw targets by the slave sensor. The KP merges the raw target lists for each $50ms$ interval from the left and right sensor and processes the object tracking. Both raw targets and object tracks are sent through the private CAN. The CLIO device captures both data from the private CAN and the vehicle CAN and forwards it, packed into a FireWire stream, to the FireWire connection of the industry computer. All available cameras are connected to the FireWire board as well.

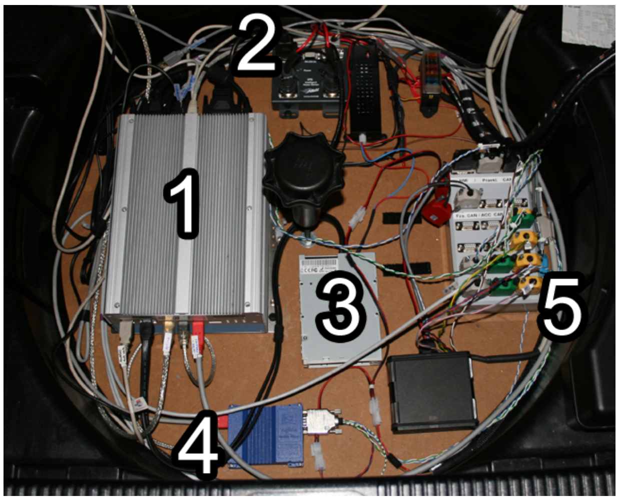


Figure 2.14: Hardware situated in the trunk of the test bed. 1: Industry Computer, 2: Power Switch, 3: S-ATA hard disk case, 4: CLIO, 5: CAN Interlink

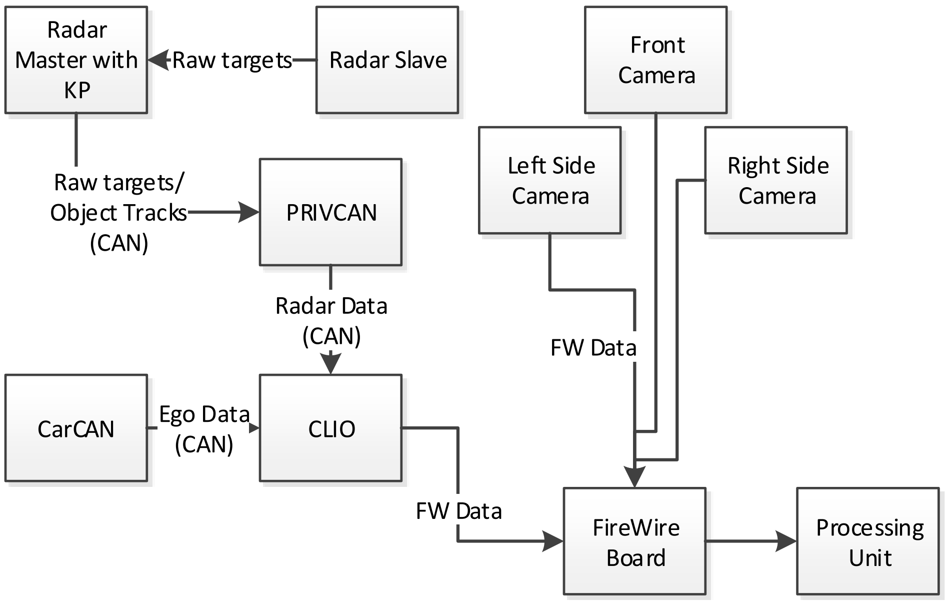


Figure 2.15: Dataflow of the test bed

2.6 Recording and Replaying of Data

Offline processing of data is a necessary need for development as the test bed vehicle is not always available and repeated testing of algorithms on the same data is crucial. Cassandra allows to continuously record sensor data and save it to video streams and CAN data files. Cassandra offers a DriveRecorder module for this purpose. The DriveRecorder provides an internal buffer to avoid frame drops and writes the video streams from the left and right camera to uncompressed Audio Video Interleave (AVI) files. The CAN data is continuously stored in *.rec files which also synchronize the camera frames to the internal clock. This results in two video streams for the two side cameras and two *.rec files for private CAN and vehicle CAN. This process requires high throughput, therefore the data streams are stored on an SSD to avoid data loss.

The Cassandra Graph for storing the data can be found in the appendix on page 105. At first the CAN data is collected from the CLIO and captured by the `clio` station which forwards it to the `ClioDemuxer` station that demuxes the private and the vehicle CAN messages to two different message streams which are connected to the two message ports of the `DriveRecorder` station. The two camera stations (`avt_left` and `avt_right`) capture uint8 gray scale images from the cameras at 20 Hz and forward it to the respective ports of the `DriveRecorder` station. There is no further processing of the input data. The `DriveRecorder` station stores the data streams in a memory buffer to avoid frame drops due to hard disk latency and subsequently writes the data to the SSD.

A minimal graph for replaying the data within the Cassandra framework can be found in Figure C.3 on page 106. The player station has to be instructed which stream to play at which port. Therefore, for each record a *.scene file is created that contains the names of files which are associated to the recording (listing 2.1). As previously mentioned, there are two files for the captured video data and two CAN message files.

```
img1=RADAR_20130604_125156_1.avi
msg1=RADAR_20130604_125156_1.rec
img2=RADAR_20130604_125156_2.avi
msg2=RADAR_20130604_125156_2.rec
```

Listing 2.1: *.scene file

After loading such a file in the player, the left video stream is sent through port `img1` and the right through `img2`. CAN messages captured from the private CAN are forwarded at the `msg1` port and the vehicle messages can be found at port `msg2` of the `Player` station.

The data captured from the CAN bus has to be decoded for further processing which is done in the FSR sub graph. This sub graph also fetches information from the `SensorSystem` station which allows to enter extrinsic parameters of the master and slave RADAR sensors. This information is needed to transform coordinates from the RADAR sensor's local coordinate frame to the ego vehicle reference frame. The mounting position in the ego coordinate frame and the respective alignment are summarized in Table 2.4 on page 37.

The FSR sub macro (Figure C.4 on page 107 in the appendix) decodes the CAN messages from the private CAN and translates it to a list of tracked objects and a list of respective RADAR raw targets for a given measurement cycle. In detail, the raw CAN messages are sent to the `msg` port of the `msg2data` station which decodes the data based on a database file and transforms it into a generic data message format of the Cassandra Framework which is then passed to the `P-CAN-Interface` station. This station collects messages containing raw targets of the master (`targetList_0`) and slave sensor (`targetList_1`) and the final object tracks (`objList`). In the last step the corresponding raw target lists are synchronized and merged into a combined list and forwarded with the respective object list for further processing.

Chapter 3

Sensor Fusion of Radar and Camera for Object Tracking

This chapter introduces a sensor fusion concept for side RADAR and side camera sensors. Features and methods usually utilized for classification and detection of vehicles in front or rear cameras are not sufficient for the processing of images from wide angle side cameras. Vehicles appear in different angles which renders the symmetry assumptions of front views and rear views inapplicable. Therefore other methods are explored in this chapter. Pre-processing methods and the definition of Regions of Interest (ROIs) are introduced in the first part. Following from this, a method to predict optical flow in an image sequence based on RADAR measurements is explained. Section 3.4 describes an optimization process to fit 3-dimensional vehicle models to position hypotheses acquired from RADAR sensors. This process is refined by exploiting the prediction of optical flow based on RADAR velocity and position information of moving objects in the last part.

3.1 Image Correction

The side camera images exhibit strong radial distortions which can influence performance of classification algorithms. These distortions can be removed by geometric transforms based on the camera model. This process is called rectification or undistortion. Mappings between the pixels of the undistorted and distorted image and vice versa are needed to identify ROIs in both images. OpenCV already provides methods for realizing geometric transforms by remapping pixel coordinates and interpolating between them from a source image (src) to a destination image (dst) in the form

$$\mathrm{dst}(x, y) = \mathrm{src}(f_x(x, y), f_y(x, y)) \tag{3.1}$$

Thus, the two mappings $f_x(x, y)$ and $f_y(x, y)$ which map the pixel coordinates of the undistorted image (x_u, y_u) to pixel coordinates in the distorted source image (x_d, y_d) have to be created.

Undistortion can be realized by imitating a pin-hole camera and projecting the camera image to a plane parallel to the sensor plane in the camera coordinate system with an arbitrary but fixed focal length f. The images have width w and height h. The roll angle α, resulting from the mounting of the camera, is also partly corrected with the help of a rotation of the plane around the z-axis in the 3D camera coordinate system. The resulting 3D points are then projected back from camera coordinates to the image plane by the camera model's `cam2img` function which is a transformation $\mathbb{R}^3 \mapsto \mathbb{R}^2$ (see section 2.4.1 for details). $\mathbf{R}_z \in \mathbb{R}^{3x3}$ is the rotation matrix around the z-axis.

$$f(x_u, y_u) \longrightarrow (x_d, y_d) : \tag{3.2}$$

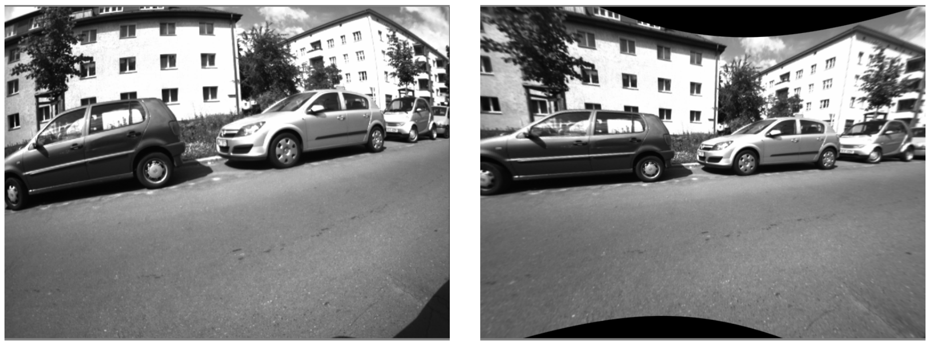
$$\mathbf{p} = \begin{pmatrix} x_u - w/2 \\ y_u - h/2 \\ w/f \end{pmatrix} \tag{3.3}$$

$$\mathbf{p}' = \mathbf{R}_z(\alpha)\mathbf{p} \tag{3.4}$$

$$(x_d, y_d) = \mathrm{cam2img}(\mathbf{p}') \tag{3.5}$$

The resulting image does not show the barrel distortions. Lines that are straight in the real world are straight in the captured image as well (Figure 3.1).

The reverse mapping $g(x_d, y_d) \mapsto (x_u, y_u)$ which maps pixels of the distorted image (x_d, y_d) to pixels in the undistorted image (x_u, y_u) is achieved by calculating the intersection of the light ray falling into pixel (x_d, y_d) with the virtual pin-hole plane previously constructed.



(a) Distorted **(b)** Undistorted and roll angle correction

Figure 3.1: Distortion and roll angle corrected camera image

At first, the plane with plane point $\mathbf{p} = (0, 0, w/f)$ and plane normal $\mathbf{n} = (0, 0, 1)^\top$ (along the z-axis) has to be constructed.

$$g(x_d, y_d) \longrightarrow (x_u, y_u) : \tag{3.6}$$

$$\mathbf{n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.7}$$

$$\mathbf{p} = \begin{pmatrix} 0 \\ 0 \\ w/f \end{pmatrix} \tag{3.8}$$

$$\mathbf{q} = \text{img2cam}(x_d, y_d) \tag{3.9}$$

The function `img2cam` returns a vector describing the direction of the corresponding light ray falling into that pixel. The intersection with this ray $\mathbf{q}$ and the plane is the point on the virtual pin-hole plane.

For the intersection $(\lambda\mathbf{q} - \mathbf{p}) \cdot \mathbf{n} = 0$ holds true, thus λ can be calculated as it follows.

$$\lambda = \frac{p_x \cdot n_x + p_y \cdot n_y + p_z \cdot n_z}{q_x \cdot n_x + q_y \cdot n_y + q_z \cdot n_z} \tag{3.10}$$

Finally, the point is rotated back into the virtual plane to take account of the roll angle correction. From the resulting point the x and y coordinates represent the pixel position in the undistorted image.

$$\mathbf{q'} = \mathbf{R^{-1}}(\alpha)_z \lambda \mathbf{q} \tag{3.11}$$

$$(x_u, y_u) = (q_x, q_y) \tag{3.12}$$

3.2 Extracting Regions of Interest

The RADAR tracker provides position information in coordinates $(x, y, 0)^\top \in \mathbb{R}^3$ in the ego coordinate system. With extrinsic and intrinsic calibration, these coordinates can be transformed into image coordinates. The region in the image that is occupied by a vehicle largely depends on the distance to the camera. ROIs should include the vehicle and avoid capturing too much unnecessary environment. The following heuristics rendered useful for achieving reasonable ROIs.

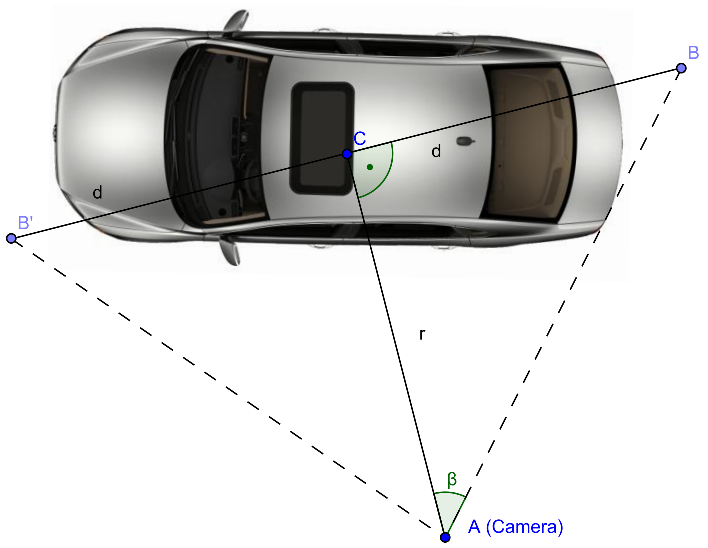


Figure 3.2: Extraction of regions of interest – C defines the tracked object center provided by the radar tracker

Figure 3.2 shows how the camera position A and the tracked object's center $C = (x_c, y_c, 0)$ are aligned. The distance from the camera to the car can be simply defined as $r = |C - A|$. The length of a car is approximately between 3 (total front view) and 6 meters (total side view). It is assumed that projecting C into the image results in image coordinates close to the center of the vehicle in the image. Thus, the area $d = 3$ meters left and right of the center C should capture both ends B and B' of the vehicle. The approximation of those two points is achieved by a rotation around the z-axis in reference point A with the rotation angle β in ego coordinates.

$$\tan\beta = \frac{d}{r} \tag{3.13}$$
$$B = \mathbf{R}_z(\beta) \cdot C \tag{3.14}$$
$$B' = \mathbf{R}_z(\beta) \cdot C \tag{3.15}$$
$$\tag{3.16}$$

Finally we set the z-coordinate of B to -0.5 (half a meter below the ground plane) and z of B' to 2.5 (2.5 meters above the ground plane).

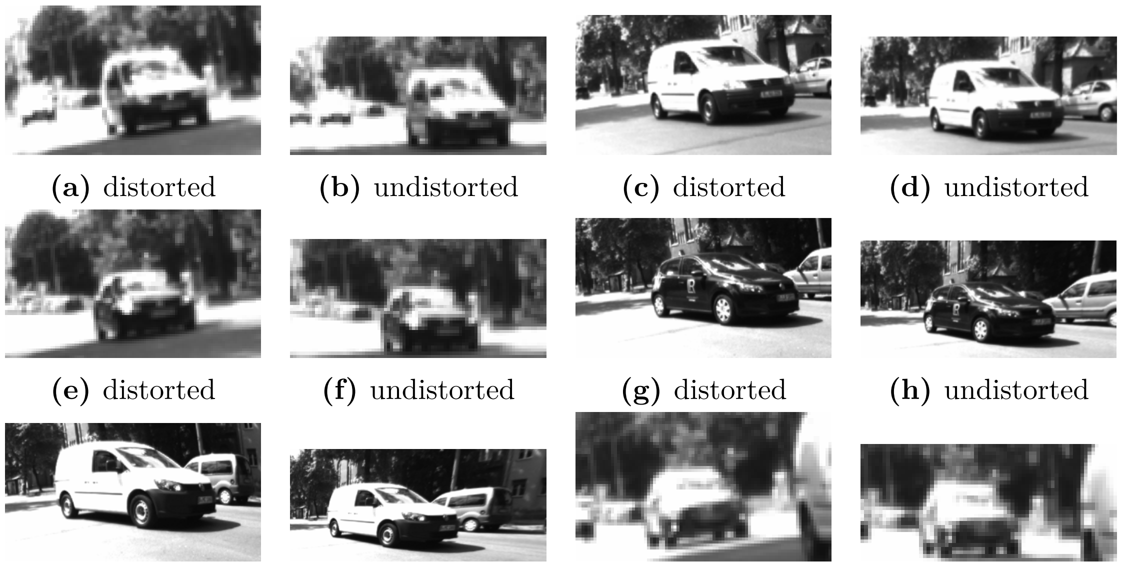
$$B_z = -0.5 \tag{3.17}$$
$$B'_z = 2.5 \tag{3.18}$$

Projecting the points B and B' back into the camera image results in the top left corner and bottom right corner of a rectangle that includes the vehicle if the tracked object's center was assumed correctly. Some extractions with different distances r can be seen in Figure 3.3. As the scaling of cars in the image is mainly dependent on the distance to the camera, the extracted ROIs are size invariant when scaled to the same image size.

The camera image still contains distortions and influence of the roll angle of the camera. Extracting the ROIs from the undistorted image might give better results for further processing. The function $\text{cam2img}(x, y, z) \mapsto (x_d, y_d) \in \mathbb{R}^2$ defines the camera coordinate to image pixel mapping of the used camera model. Instead of directly extracting the rectangle defined by $\text{cam2img}(B)$ and $\text{cam2img}(B')$, the transformation $g(x_d, y_d) \longrightarrow (x_u, y_u)$ presented in section 3.1 is applied to the rectangles corner points. The new rectangle corner points $C = g(\text{cam2img}(B))$ and $C' = g(\text{cam2img}(B'))$ describe a similar rectangle in the undistorted image. Figure 3.3 shows extracted rectangles from one of the recordings both from the distorted original camera image and the undistorted image.

3.3 Image Segmentation Based on Predicted Optical Flow

Optical flow in general is defined as the offset of pixels through movement in two consecutive images. A pixel at position $\mathbf{u}_t = (u_t, v_t)$ in the image at time



(a) distorted **(b)** undistorted **(c)** distorted **(d)** undistorted
(e) distorted **(f)** undistorted **(g)** distorted **(h)** undistorted
(i) distorted **(j)** undistorted **(k)** distorted **(l)** undistorted

Figure 3.3: Extracted ROIs from the distorted image and the undistorted image with slight roll correction

point t can be found in the consecutive image at time point $t+1$ at position $\mathbf{u}_{t+1} = (u_{t+1}, v_{t+1})$. In conclusion, the optical flow is defined as $\mathbf{f} = (\Delta u, \Delta v)^{\top}$. The RADAR tracker provides relatively accurate velocity information. However, the position estimate, and following from this, the size of the object might be imprecise. The motion of an object in 3D-space indirectly defines the flow in the 2D image. If the motion in 3D space is known, the motion in the image can be predicted by the help of a camera model which translates 3D points into image space.

This section presents a way to fuse velocity and position information acquired from the RADAR sensors with optical flow from the camera images to segment ROIs that move with similar velocity to the RADAR target. These regions are more likely to contain the actual moving object and cover its full dimensions. In this approach a few assumptions are made:

1. Optical flow is highly dependent on the distance to the camera, e.g. close vehicles have higher optical flow response than vehicles further away from the camera, despite having the same velocity.

2. Knowing position $\mathbf{x}_C = (x, y, z)^\top$ and velocity $\mathbf{v}_C = (v_x, v_y, v_z)^\top$ in camera coordinates, the optical flow of the corresponding image point can be predicted for the next frame.
3. Moving vehicles have a good response to optical flow based on Kanade-Lucas-Tomasi Tracker (KLT Tracker) feature tracking method [TK91] because they contain a high amount of structure compared to most environments.

3.3.1 Prediction of Optical Flow

The RADAR tracker returns a radar object state as a vector containing the position $(x_e, y_e, 0) \in \mathbb{R}^3$ and its velocity $(v_{e,x}, v_{e,y}, 0) \in \mathbb{R}^3$ in the ego vehicle coordinate system E. These are transformed into the camera reference frame C by the help of the extrinsic calibration of the camera. The velocity vector $\mathbf{v}_E$ is only rotated and not translated. To skip the translation in homogeneous coordinates the last entry of the vector is set to 0.

$$\mathbf{x}_C = \mathbf{T}_{C \leftarrow E}\mathbf{x}_E = \mathbf{T}_{C \leftarrow E} \begin{pmatrix} x_e \\ y_e \\ 0 \\ 1 \end{pmatrix} \tag{3.19}$$

$$\mathbf{v}_C = \mathbf{R}_{C \leftarrow E}\mathbf{v}_E = \mathbf{T}_{C \leftarrow E} \begin{pmatrix} v_{e,x} \\ v_{e,y} \\ 0 \\ 0 \end{pmatrix} \tag{3.20}$$

The prediction of the optical flow of a point $\mathbf{x}_t = (x_{c,x}, y_{c,y}, z_{c,z})^\top$ in camera coordinates moving with $\mathbf{v}_t = (v_{c,x}, v_{c,y}, v_{c,z})^\top$ can be achieved by linear forward motion with the estimated velocity for the time difference of two consecutive frames. Both RADAR and camera generate data with a frequency $f = 20Hz$ in the current setup. Therefore, the time difference of two consecutive frames is $\frac{1}{f} = 0.05s$. Thus, the new position $\mathbf{x}_{t+1}$ of an object in the consecutive frame is estimated as it follows.

$$f = 20\text{Hz} \tag{3.21}$$

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \frac{1}{f}\mathbf{v}_t = \begin{pmatrix} x_t + \frac{1}{f}v_{t,x} \\ y_t + \frac{1}{f}v_{t,y} \\ z_t + \frac{1}{f}v_{t,z} \end{pmatrix} \tag{3.22}$$

Using the chosen camera model, both $\mathbf{x}_t$ and $\mathbf{x}_{t+1}$ in camera coordinates are transformed into image coordinates $\mathbf{u}_t = (u_t, v_t)$ and $\mathbf{u}_{t+1} = (u_{t+1}, v_{t+1})$. Thus, the predicted optical flow $\mathbf{f}_t = (\Delta u, \Delta v)^\top$ is the difference of both image coordinates.

$$\mathbf{u}_t = \text{cam2img}(\mathbf{x}_t) \tag{3.23}$$

$$\mathbf{u}_{t+1} = \text{cam2img}(\mathbf{x}_{t+1}) \tag{3.24}$$

$$\mathbf{f}_t = \begin{pmatrix} \Delta u \\ \Delta v \end{pmatrix} = \mathbf{u}_{t+1} - \mathbf{u}_t = \begin{pmatrix} u_{t+1} - u_t \\ v_{t+1} - v_t \end{pmatrix} \tag{3.25}$$

The prediction of optical flow for arbitrary points of the image based on a single RADAR object is achieved by placing a 3-dimensional plane d parallel to the projection plane of the camera in the camera reference frame. This plane is fixed in the RADAR object's position $\mathbf{x} = (x, y, z)^\top$ with a plane normal $\mathbf{n} = (0, 0, 1)^\top$. The whole plane is moved with velocity $\mathbf{v}_C$ for one time cycle. For each image pixel $\mathbf{u} = (u, v)$ in the image at time point t a ray originating in the camera's viewpoint is pointed to the plane d and the intersection point $\mathbf{h}_t$ is computed. This 3-dimensional intersection point $\mathbf{h}_t$ is then forward predicted to time point $t+1$ through a motion for one time step with velocity $\mathbf{v}_C$ as described in eq. (3.22). Figure 3.4 visualizes this approach.

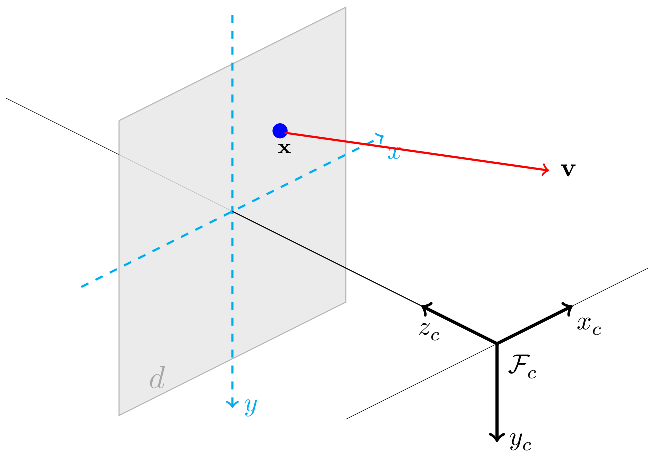


Figure 3.4: Prediction of optical flow through a plane d originating in the RADAR object's position $\mathbf{x}$ in camera coordinates and moving this plane with velocity $\mathbf{v}$ for one time frame

To find the points on the virtual plane d for time frame t, the light rays of the camera model have to be intersected with the plane. The Scaramuzza camera model defines the direction of a light ray $\lambda\mathbf{q}, \lambda \in \mathbb{R}, \mathbf{q} \in \mathbb{R}^3$ falling into the camera's viewpoint for each image coordinate $\mathbf{u} = (u, v)$ by the function img2cam(u, v) which maps the image coordinate to a 3-dimensional vector that represents the direction of the light ray $\mathbf{u} = (u, v) \in \mathbb{R}^2 \mapsto \mathbf{q} = (x, y, z) \in \mathbb{R}^3$. The intersection of this light ray $\lambda\mathbf{q}$ with the virtual plane d is a potential object position for which the predicted flow is calculated. The following equations show the algorithm for an arbitrary image point $\mathbf{u}$ and RADAR object position $\mathbf{x}$.

$$\mathbf{n} = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.26}$$

$$\mathbf{x} = \begin{pmatrix} x \\ y \\ z \end{pmatrix} \qquad \text{RADAR target in camera coordinates} \tag{3.27}$$

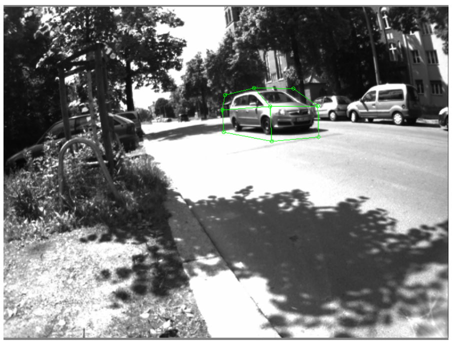
$$\mathbf{q} = \text{img2cam}(\mathbf{u}) \qquad \text{light ray for pixel position } \mathbf{u} \tag{3.28}$$

The intersection point can be calculated by solving the plane's normal form $(\lambda\mathbf{q} - \mathbf{x}) \cdot \mathbf{n} = 0$ for λ. Thus, the corresponding potential object point $\mathbf{h}_t$ on the plane d is $\lambda\mathbf{q}$.

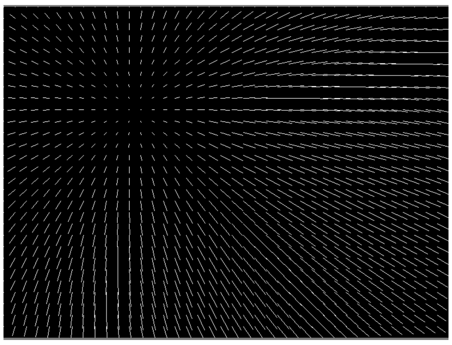
$$\lambda = \frac{x_x \cdot n_x + x_y \cdot n_y + x_z \cdot n_z}{q_x \cdot n_x + q_y \cdot n_y + q_z \cdot n_z} \tag{3.29}$$

$$\mathbf{h}_t = \lambda\mathbf{q} \tag{3.30}$$

The optical flow for the camera coordinate point $\mathbf{h}_t$ can be predicted as described in eq. (3.22). Figure 3.5 shows a dense field of the predicted optical flow for a given



(a) Camera image which contains a moving object tracked by RADAR



(b) Predicted optical flow field for the whole image

Figure 3.5: Predicted flow based on RADAR tracked object

RADAR object. The following section shows how such a field can be exploited to narrow a ROI of a moving object in the camera image.

3.3.2 Comparison of Predicted and Lucas Kanade Optical Flow

The previous section explained how to predict the optical flow for a single RADAR object at arbitrary positions (u, v) in the camera image. In the next step image parts have to be located in which the optical flow is similar to the predicted optical flow. A standard method to extract optical flow information from two consecutive images is a method presented by Bruce D. Lucas and Takeo Kanade [Kan81]. This method finds nearby motions of pixels in a close neighborhood. The approach has been improved several times. Tomasi, Lucas and Kanade presented a method to track corners over several frames which is known as KLT Tracker [TK91]. In another paper called "Good Features to Track" [ST94], Jianbo Shi and Carlo Tomasi added another step in which good initial features suitable for tracking are chosen. The optical flow from two consecutive camera images is acquired by applying the Lucas Kanade method [TK91] implemented in OpenCV [Bou01, Its13b].

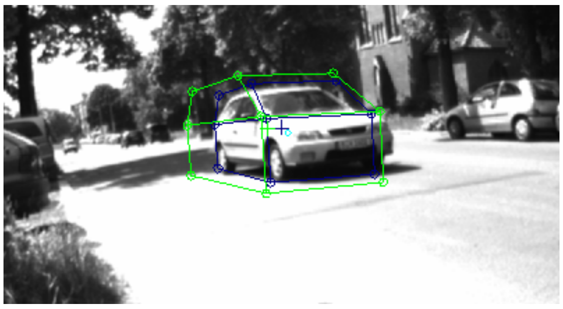
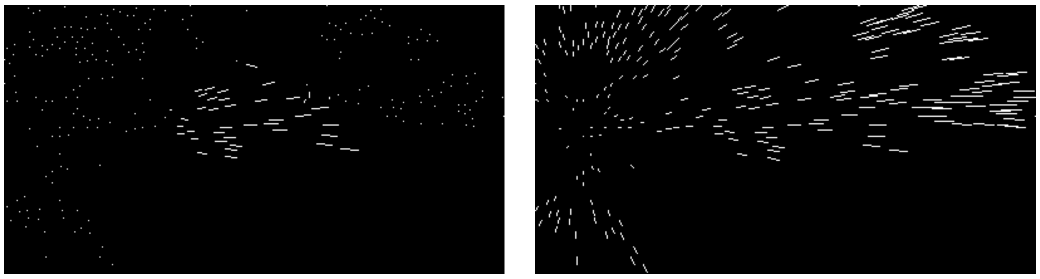


Figure 3.6: Oncoming vehicle with $v = 40.95$ km/h

Figure 3.6 shows a scenario in which the ego vehicle is standing in a parking spot. The other vehicle moves with a velocity of 40.95 km/h towards the ego vehicle. The Lucas Kanade flow vectors in the area of the moving vehicle are similar to those predicted while those outside of the vehicle area are not.

Figure 3.7 shows the predicted flow and the actual Lucas Kanade flow at the feature positions detected by the KLT Tracker. The predicted flow presented in section 3.3.1 is annotated as $\mathbf{f}_p$ while the flow based on the Lucas Kanade method is annotated as $\mathbf{f}_l$ in the following paragraphs. The base for comparison are the corner features detected by the KLT Tracker. The tracker returns a set F of m corner feature points $\mathbf{u_i} = (u_i, v_i)$ with $i \in \mathbb{N}$ and $0 < i \leq m$ that could be tracked from one image to the other and their respective flow $\mathbf{f}_l(\mathbf{u}_i) \in \mathbb{R}^2$. For the



(a) Optical flow computed with Lucas Kanade method

(b) Predicted Optical flow at KLT feature points

Figure 3.7: Comparison of optical flow and predicted optical flow for a specific RADAR target

comparison of the flow at image position $\mathbf{u}_i = (u_i, v_i) \in F$, the following equation defines a similarity measure $\mathbf{u}_i \mapsto w_i \in [0, 1]$ based on the Euclidean distance of the normalized flow vectors in eq. (3.32).

$$n_i = \max(\|\mathbf{f}_p(\mathbf{u}_i)\|, \|\mathbf{f}_l(\mathbf{u}_i)\|) \tag{3.31}$$

$$w_i = 1 - \frac{\|\mathbf{f}_p(\mathbf{u}_i) - \mathbf{f}_l(\mathbf{u}_i)\|}{2n_i} \tag{3.32}$$

A similarity of 1 is a perfect match, while a similarity of 0 means, that the flow vectors are showing in opposite directions. To save computation time, weights that do not exceed a certain threshold f_{min} are removed. It shows useful to normalize and rescale weights within defined thresholds $[f_{min}, f_{max}]$ to the interval $[0, 1]$ to be used in the model fitting approach presented in section 3.4. For each w_i a corresponding refined weight w_i' is computed. Weights that do not reach the minimum thresholds are discarded from the set.

$$s_i = \frac{w_i - f_{min}}{f_{max} - f_{min}} \tag{3.33}$$

$$w_i' = \begin{cases} \text{discard} & \text{if } s_i \leq 0 \\ 1 & \text{if } s_i > 1 \\ s_i & else \end{cases} \tag{3.34}$$

This method allows weighting of the detected feature points in the image. Certainly, weighting a wider area around the detected feature point is preferred as the moving object very likely spreads around the feature point in a certain environment. The distribution around the feature point however, is dependent on the distance of the object to the camera, e.g. a vehicle 50 m away from the camera may only consist of 3 feature points and is about 10 pixels wide. Therefore, a

Gaussian function with a standard deviation depending on the distance to the camera is introduced.

$$\mathrm{f}_i(u,v) = w_i' \cdot e^{\frac{(u_i-u)^2+(v_i-v)^2}{2\sigma^2}} \tag{3.35}$$

The standard deviation estimation in eq. (3.38) has shown to return reasonable estimates supported by several experiments. The vector $\mathbf{x}_e = (x, y, 0, 1)^\top$ defines the object's position in homogeneous ego coordinates taken from the RADAR measurement. It is transformed into the camera coordinate system to calculate its distance d to the camera center.

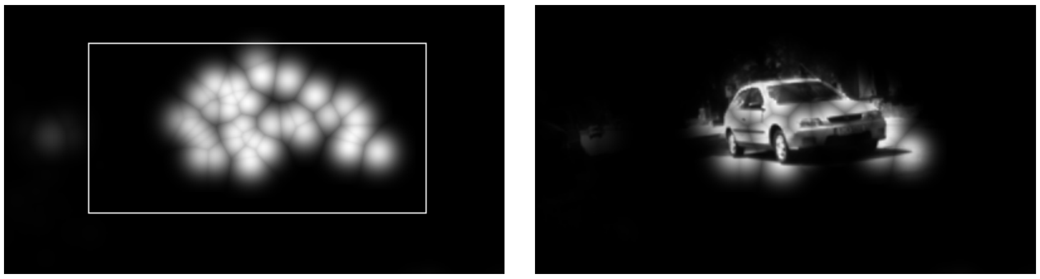
$$\mathbf{x}_c = \mathbf{T}_{C \leftarrow E}\mathbf{x}_e \tag{3.36}$$

$$d = \|\mathbf{x}_c\| \tag{3.37}$$

$$\sigma = 15 \cdot \frac{1}{1+0.1d} \tag{3.38}$$

To define weights for every pixel of the image. two weighting methods have been explored that showed reasonable results during experiments. The example images show the weight maps for the scenario introduced previously (Figure 3.6). The ego vehicle stands still and another vehicle is coming from the left with a velocity of 40.95 km/h.

Weight Distribution Method 1:



(a) Weight distribution with annotated ROI of the detected vehicle

(b) Weight distribution multiplied with the input image

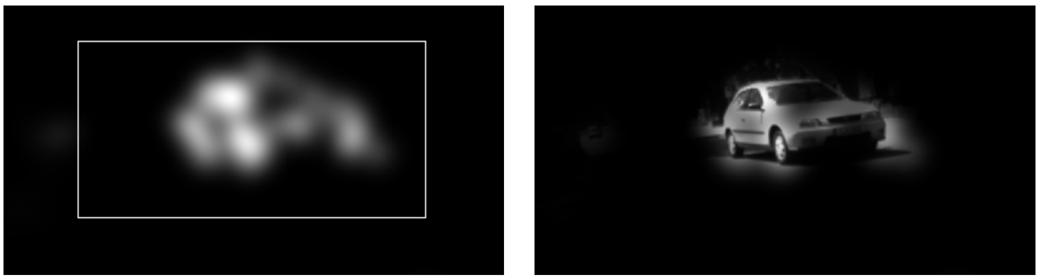
Figure 3.8: Optical flow based weight method 1

As aforementioned, the distribution of w_i' around the detected corner feature point (u_i, v_i) is approximated by a Gaussian-like function $\mathrm{f}_i(u,v)$. At intersections of two or more distributions, the highest weight is chosen.

$$w(u,v) = \max_i(\mathrm{f}_i(u,v)) \tag{3.39}$$

The result is a weight map of the whole camera input image for each RADAR object. Higher weights define areas which move with a velocity similar to the moving object detected by the RADAR (Figure 3.8).

Weight Distribution Method 2:



(a) Weight distribution with annotated ROI of the detected vehicle

(b) Weight distribution multiplied with the input image

Figure 3.9: Optical flow based weight method 2

This weighting method is based on the total probability $P(A)$ of an arbitrary image point A belonging to the moving object. If two or more detected feature points are next to each other and have similar optical flow, the probability of the points lying between them to belong to the moving object increases. Let A be an arbitrary image point (u, v) and B_i n detected feature points with $P(B_i)$ being the probability of belonging to the moving object. Therefore, the total probability of A being a point of a moving vehicle under the given distributions of B_i, is defined as the sum over the conditional probabilities $P(B_i)P(A|B_i)$. $P(A|B_i)$ is the probability of A belonging to the moving object at point B_i.

$$P(A) = \sum_{i=1}^{n} P(B_i) \cdot P(A|B_i) \tag{3.40}$$

Assuming that the estimated weight w_i' of a corner feature point (u_i, v_i) detected by the KLT Tracker is similar to the likelihood of the feature point containing the moving object which is looked for, this translates to

$$w(u, v) \propto s(u, v) = \sum_{i}^{m} w_i' \cdot e^{\frac{(u_i - u)^2 + (v_i - v)^2}{2\sigma^2}} \tag{3.41}$$

Certainly, this process is neither continuous nor normalized anymore. This could be normalized such that the integral over the entire image area equals one, though this is of no use for the further processing. The weight map presents a good heuristic for the likelihood if an area of the image belongs to the moving object. The summed up Gaussian values might exceed the desired weight interval $[0, 1]$. Therefore a rescaling is needed. This can be achieved by a division by the final maximum value of the weight map. This scales the weight map $w(u, v)$ to $[0, 1]$ in eq. (3.42).

$$w(u, v) = \frac{s(u, v)}{\max(s)} \tag{3.42}$$

An example map is presented in Figure 3.9. The weights are distributed smoother over the image compared to the first weighting method. The comparison of optical flow and the derived weight maps allows to find regions in the image that move with similar velocity as the vehicle tracked by the RADAR tracker.

3.4 3D Model Fitting

A relatively old approach for object detection and tracking is deformable model fitting. First papers date back in the early 90's [Low91], but the idea has been revived in several new papers as well, e.g. [LM09, LTH$^+$05]. A model in this case is a three dimensional model of the object which is looked for. In general the approach tries to fit the projected edges of the model to edges found in the camera image. In the first step, hypotheses are generated which estimate a ROI and initial parameter sets. In the second step an error measurement is evaluated, e.g. distance of reprojected points to edges in the image. This error is used for an optimization step that incrementally optimizes the parameters of the model to fit to the extracted edges in the image. Both papers assume a very simple camera model which does not model distortions. Furthermore, the camera is assumed to be static. In the following sections, the approach will be tested on strongly distorted images utilizing the Scaramuzza camera model [SMS06a] on the test bed's wide angle camera optics. Additionally, approaches to compensate for camera motion and inaccurate starting hypotheses based on RADAR are presented.

3.4.1 The 3D Vehicle Model

Both [Low91, LM09] fit detailed and complex 3-dimensional models to the camera images. Unfortunately, this renders the processing costs very high and is far from being able to perform in real-time. Instead, a simple model is used in this thesis to achieve better processing speed.

The coordinates of the model points in the reference model coordinate system are annotated as $\mathbf{x}_m = (x_m, y_m, z_m)^\top \in \mathbb{R}^3$. The ego vehicle coordinates are labeled with sub index e and camera coordinates with sub index c.

The model parameters are scaling factors along the models axes (s_x, s_y, s_z) with x referring to length, y to width and z to height of the vehicle (Figure 3.10). Furthermore, the vehicle is rotated by a yaw angle ψ and positioned by an offset on the ground plane $(t_x, t_y, 0)$ relative to the ego vehicle reference frame.

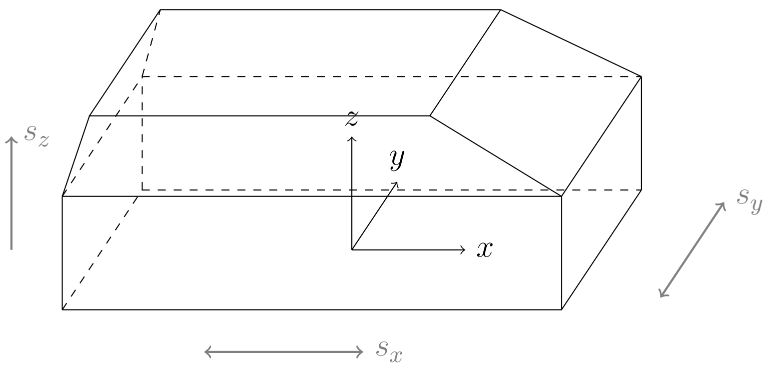


Figure 3.10: 3D model wire frame

The transformation from model coordinates to ego vehicle coordinates can be achieved by a transformation matrix $\mathbf{T}_{E\leftarrow M}$ for homogeneous coordinates.

$$\mathbf{T}_{E\leftarrow M} = \underbrace{\begin{pmatrix} \cos\psi & -\sin\psi & 0 & t_x \\ \sin\psi & \cos\psi & 0 & t_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Rotation + Offset}} \cdot \underbrace{\begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}}_{\text{Scaling}} \tag{3.43}$$

$$= \begin{pmatrix} s_x\cos\psi & -s_y\sin\psi & 0 & t_x \\ s_x\sin\psi & s_y\cos\psi & 0 & t_y \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.44}$$

These ego coordinates are transformed into the camera's reference frame. The

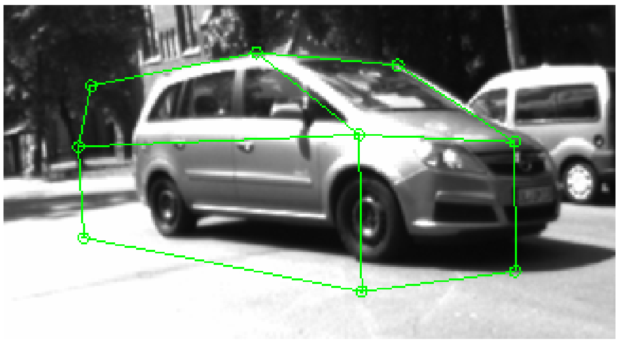


Figure 3.11: 3D model initialized on a RADAR hypothesis projected into the image

final model wire frame can be projected into the image as seen in Figure 3.11.

3.4.2 Edge Distance Error Measurement

The fitting of a model can be formulated as the minimization of a cost function [Low91, LM09, LTH+05]. A good error measurement is needed to apply the optimization. [Low91] proposes the closest distance from projected lines to lines in a Canny filtered edge image. Assume (u, v) are image coordinates and (x, y, z) camera coordinates. The vector $\boldsymbol{\beta} = (\psi, s_x, s_y, s_z, t_x, t_y)$ contains all parameters of the vehicle model. In the first step the camera image is Canny filtered [Can86]. The result is a binary image in which each non-zero pixel is an edge candidate. Points from the model's projected lines should match lines in the Canny image. Hence, a distance from a point on a projected model line to the closest line detected in the Canny image is defined in the next paragraphs. The underlying idea can be seen in Figure 3.12.

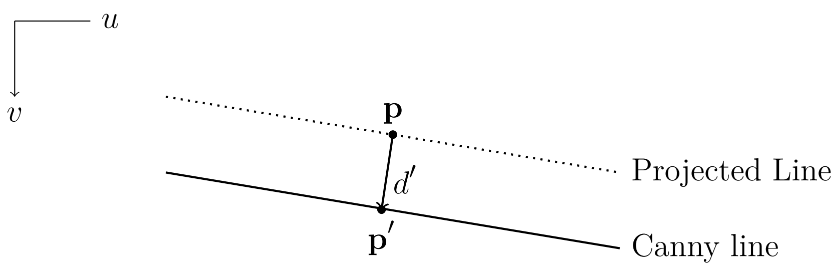


Figure 3.12: Distance measurement d' of the projected model edge to the closest image edge candidate

A line in image coordinates (u, v) can be defined by its closed form.

$$u \cos\theta - v \sin\theta - d = 0 \tag{3.45}$$

When evaluating the left side of the equation with arbitrary points (u, v) the orthogonal distance from the closest point on this line to the evaluated point is calculated. Points which lie directly on this line have a distance of 0. Thus, the error function is

$$d'(u, v) = u \cos\theta - v \sin\theta - d \tag{3.46}$$

The line parameters θ and d are computed by the help of two points $\mathbf{x}_{m,1}$ and $\mathbf{x}_{m,2}$ of the model edge that are projected into the image as (u_1, v_1) and (u_2, v_2). From these points, the parameters are computed as proposed in [Low91].

$$l = \sqrt{(u_2 - u_1)^2 + (v_2 - v_1)^2} \tag{3.47}$$

$$\cos\theta = \frac{u_2 - u_1}{l} \tag{3.48}$$

$$\sin\theta = \frac{v_2 - v_1}{l} \tag{3.49}$$

$$d = u_1 \cos\theta - v_1 \sin\theta \tag{3.50}$$

To find the closest edge from a projected point $\mathbf{p} = (u, v)$, a perpendicular search line has to be defined in a window size w around the projected model point. Both $\cos\theta$ and $\sin\theta$ which were used for the definition of the line distance are also utilized to define a perpendicular line segment to the projected model line intersecting at $\mathbf{p}$.

$$\mathbf{p_1} = \begin{pmatrix} p_x - \sin\theta \cdot w \\ p_y + \cos\theta \cdot w \end{pmatrix} \quad \mathbf{p_2} = \begin{pmatrix} p_x + \sin\theta \cdot w \\ p_y - \cos\theta \cdot w \end{pmatrix} \tag{3.51}$$

The image points $\mathbf{p}_1$ and $\mathbf{p}_2$ define a line segment through image points which are perpendicular to the projected model edge. The distance algorithm checks for the closest non-zero point in the Canny filtered image and returns the distance to $\mathbf{p}$ as the error. A very accurate and fast way to sample pixels on the line segment between $\mathbf{p}_1$ and $\mathbf{p}_2$ is the Bresenham algorithm [Bre65], which returns a set B with all pixels (u, v) on the search line. These pixels are checked for the closest pixel to $\mathbf{p}$ which contains an edge in the Canny filtered image. A simple pseudo code version of the algorithm is presented in algorithm 1. Points with errors equal

Algorithm 1 Finding the closest edge point to a projected model point

```
1: Input: $p_x, p_y$
2: Output: $e$
3: $\mathbf{p_1} = \begin{pmatrix} p_x - \sin\theta \cdot w \\ p_y + \cos\theta \cdot w \end{pmatrix}$
4: $\mathbf{p_2} = \begin{pmatrix} p_x + \sin\theta \cdot w \\ p_y - \cos\theta \cdot w \end{pmatrix}$
5: $e := \infty$
6: $B \leftarrow \text{bresenham}(\mathbf{p}_1, \mathbf{p}_2)$
7: for all $b = (u, v) \in B$ do
8:     $edge \leftarrow \text{canny}(u, v)$
9:     if $egde > 0$ then
10:        Calculate distance to edge point:
11:        $dist \leftarrow u \sin\theta - v \cos\theta - d_b$
12:        if $||dist|| < ||e||$ then
13:            $e \leftarrow dist$
14:        end if
15:    end if
16: end for
17: return e
```

to ∞ should be discarded and not taken into account for the final solution.

3.4.3 Iterative Cost Minimization Based on Edge Distance

According to [Low91, LM09] it is sufficient to locally optimize and use an iterative approach to find an optimal solution. However, hypotheses for fitting should be relatively accurate already for the optimization to converge to a final solution. The initial parameters which are acquired from the Radar sensor are distance r, angle θ and velocity in the direction of the sensor v_r in the sensor's coordinate system. From these the position in the ego coordinate system (t_x, t_y) is computed. Over time the tracker also collects information about the velocity (v_x, v_y) in the ego coordinate system. From the latter, a good value of the yaw angle ψ of the object can be estimated by utilizing $\operatorname{atan2}(y, x)$ with respect to ego motion in eq. (3.53).

$$\operatorname{atan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right) & x > 0 \\ \arctan\left(\frac{y}{x}\right) + \pi & y \geq 0, x < 0 \\ \arctan\left(\frac{y}{x}\right) - \pi & y < 0, x < 0 \\ +\frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases} \tag{3.52}$$

$$\psi = \operatorname{atan2}(v_y + v_{ego,y}, v_x + v_{ego,x}) \tag{3.53}$$

For the optimization process, a Jacobi matrix is utilized to locally linearize the effect of parameter changes to the distance error measurement defined in section 3.4.2. The model parameters are collected in d-dimensional vector $\boldsymbol{\beta} = (\psi, s_x, s_y, s_z, t_x, t_y)^\top$. The initial position and yaw angle are acquired from RADAR measurements. The initial scaling parameters are set to 1.

Through linear interpolating between edge points of the model in coordinate system M, n model points $\mathbf{x}_{m,i}, 0 < i \leq n$ are sampled. The vector $\mathbf{e}$ holds the edge distance errors for each model point projected to the image with initial parameters $\boldsymbol{\beta}_s$. These distance terms are defined by searching for edge candidates perpendicular to the projected model line in the image (see algorithm 1 for further details).

The line i of this Jacobi matrix with respect to model point $\mathbf{x}_{m,i}$ and its respective projection in the image (u_i, v_i) is defined by the partial derivatives of the error function with respect to the model parameters.

$$d'_i(u_i, v_i) = u_i \cos\theta_i - v_i \sin\theta_i - d_i \tag{3.54}$$

$$\mathbf{J}_i = \begin{pmatrix} \frac{\delta d'_i}{\delta\psi} & \frac{\delta d'_i}{\delta s_x} & \frac{\delta d'_i}{\delta s_y} & \frac{\delta d'_i}{\delta s_z} & \frac{\delta d'_i}{\delta t_x} & \frac{\delta d'_i}{\delta t_y} \end{pmatrix} \tag{3.55}$$

Instead of deriving the error function for each parameter, values in a small ϵ environment $[-\epsilon, +\epsilon]$ around the current value of the parameter is used to linearize. Values in the environment [-0.2, 0.2] have shown good results. This is done for each model parameter of $\boldsymbol{\beta}$ while all others remain constant. Let $\mathrm{u}(\mathbf{x}_{m,i}, \boldsymbol{\beta})$ and $\mathrm{v}(\mathbf{x}_{m,i}, \boldsymbol{\beta})$ project a sampled model point $x_{m,i}$ to image coordinates (u, v) according to model parameter set $\boldsymbol{\beta}$.

For yaw angle ψ this can be achieved in the following way:

$$\boldsymbol{\beta}_1 = \boldsymbol{\beta} \text{ with } \psi \rightarrow \psi - \epsilon \tag{3.56}$$

$$\boldsymbol{\beta}_2 = \boldsymbol{\beta} \text{ with } \psi \rightarrow \psi + \epsilon \tag{3.57}$$

$$\frac{\delta d'_i}{\delta\psi} = \frac{d'_i(\mathrm{u}(\mathbf{x}_{m,i}, \boldsymbol{\beta}_2), \mathrm{v}(\mathbf{x}_{m,i}, \boldsymbol{\beta}_2)) - d'_i(\mathrm{u}(\mathbf{x}_{m,i}, \boldsymbol{\beta}_1), \mathrm{v}(\mathbf{x}_{m,i}, \boldsymbol{\beta}_1))}{2\epsilon} \tag{3.58}$$

Each partial derivative describes how the error measurement would change if the respective parameter was changed. In general, this comes down to solving eq. (3.60) for $\delta\boldsymbol{\beta}$ for all visible projected model points by applying the least-squares method and iteratively refining $\boldsymbol{\beta}$ by the estimated change $\delta\boldsymbol{\beta}$.

$$\mathbf{J}\ \delta\boldsymbol{\beta} = \mathbf{e} \tag{3.59}$$

$$\delta\boldsymbol{\beta} = (\mathbf{J}^\top\mathbf{J})^{-1}\mathbf{J}^\top\mathbf{e} \tag{3.60}$$

This technique is prone to over-fitting, especially, if only a few points could be sampled for error measurement. The model refinement might jump through the image or estimate very unusual model parameters, e.g. a very small height scaling s_z which flattens the model to unrealistic extents. Regularization is a common technique to avoid over-fitting. [Low91] uses Tikhonov regularization [TA77]. This technique is also common in ridge regression and introduces a parameter λ which defines a trade-off between fitting and the distance of the refined parameters to the initial parameters. Furthermore a weight matrix $\mathbf{W} \in \mathbb{R}^{d\times d}$ that includes weights of model parameters according to their error distribution is introduced. This matrix serves for weighting the importance of each parameter to the final solution, e.g. the initial yaw angle ψ might be very accurate but the position estimate (t_x, t_y) might not, hence, the solution should prefer changes in position instead of changes of the yaw angle. The weight matrix $\mathbf{W}$ is defined by [Low91] as a diagonal matrix containing the standard error deviations of the parameters. $\mathbf{d}$ is a d-dimensional vector containing the default values from which the solution should not differ too much. As the solution of the Jacobian optimization is a delta to the initial parameter set, this vector is a zero-vector, thus $\mathbf{d} = \mathbf{0}$.

$$W_{ii} = \frac{1}{\sigma_i} \tag{3.61}$$

The cost function that should be minimized is

$$||\mathbf{J}\delta\boldsymbol{\beta} - \mathbf{e}||^2 + \lambda||\mathbf{W}(\delta\boldsymbol{\beta} - \mathbf{d})||^2 \tag{3.62}$$

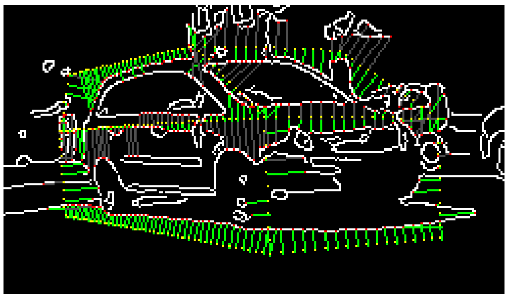
The minimum of the cost function can be found by setting the first derivative with respect to $\delta\boldsymbol{\beta}$ to zero and solve for $\delta\boldsymbol{\beta}$.

$$0 = 2\mathbf{J}^\top(\mathbf{J}\delta\boldsymbol{\beta} - \mathbf{e}) + 2\lambda\mathbf{W}^\top(\mathbf{W}\delta\boldsymbol{\beta} - \mathbf{W}\mathbf{d}) \tag{3.63}$$

$$(\mathbf{J}^\top\mathbf{J} + \lambda\mathbf{W}^\top\mathbf{W})\delta\boldsymbol{\beta} = \mathbf{J}^\top\mathbf{e} + \lambda\mathbf{W}^\top\mathbf{W}\mathbf{d} \tag{3.64}$$

$$\delta\boldsymbol{\beta} = (\mathbf{J}^\top\mathbf{J} + \lambda\mathbf{W}^\top\mathbf{W})^{-1}(\mathbf{J}^\top\mathbf{e} + \lambda\mathbf{W}^\top\mathbf{W}\mathbf{d}) \tag{3.65}$$

A first iteration already gives good results in many cases. Figure 3.13 shows the distances measured in the first iteration and the corresponding refinement.


(a) Distance (green) to next edge point from the projected model points in the Canny filtered image

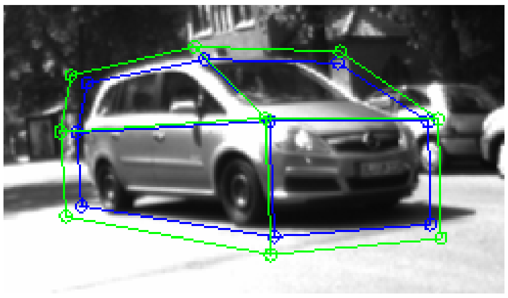

(b) Initialization (green) and refined model (blue) projected into the image

Figure 3.13: Model parameter refinement after one iteration

3.4.4 Motion Similarity-based Edge Weighting

The presented approach already provides good results with good starting hypotheses. Unfortunately, the starting hypothesis acquired from the RADAR are not always a good estimate and are too far away from the actual position in the image. This often results in model edges fitting to environment edges like parked cars on the side of the road. Section 3.3 described how the prediction of optical flow in two consecutive camera images is compared to the actual optical flow to find regions in an image that move with a given velocity in a given distance to the camera. This section is about using the computed weight map to optimize the position estimate of a RADAR tracker object.

It is a reasonable assumption that areas in the camera image that move with the same velocity as the RADAR tracker object contain edges that belong to said object. Therefore, fitting to these edges should return better results. Let $\mathbf{g}_i = (g_i, h_i) \in \mathbb{R}^2$ be the image coordinate of the edge point which was detected when evaluating the distance measure function $\mathrm{d'}_i(u, v)$ at image position (u_i, v_i) for the Jacobian row $\mathbf{J}_i$. Furthermore let $\mathrm{w}(u, v)$ be the flow weight map presented in section 3.3. Higher weights relate to a higher probability of the image point being part of the moving object detected by RADAR.

The importance of this row to the final solution should factor in the weight of the edge point. The row of the Jacobian and the corresponding error should therefore be changed to include the weight of the edge point in the following way.

$$w = 1 + \mathrm{w}(g_i, h_i) \tag{3.66}$$

$$\mathbf{J}'_i = \begin{pmatrix} w\frac{\delta d'_i}{\delta \psi} & w\frac{\delta d'_i}{\delta s_x} & w\frac{\delta d'_i}{\delta s_y} & w\frac{\delta d'_i}{\delta s_z} & w\frac{\delta d'_i}{\delta t_x} & w\frac{\delta d'_i}{\delta t_y} \end{pmatrix} \tag{3.67}$$

$$e'_i = w \cdot e_i \tag{3.68}$$

This method reduces edge fitting to parked vehicles or shadows of trees.

3.4.5 Motion Similarity Centroid-based Position Correction

The cameras are aligned perpendicular to the street. Distances of objects are not trivial to measure with mono cameras. However, lateral measurements can be done precisely. This section shows how the optical flow prediction and similarity measurement presented in Section 3.3 is exploited to refine the position of a vehicle initialized from a RADAR track. Section 3.2 presented a method to extract regions of interest from starting hypotheses. Furthermore section 3.4.4 showed how a weight map based on optical flow similarity between predicted flow and actual flow in the image can be created.

In this section a method is proposed that refines the position of the tracked vehicle $(x, y, 0)$ in the ego vehicle coordinate system based on the centroid of the detected optical flow motion. Within the defined ROI the centroid of the flow similarity distribution is estimated (see section 3.2 for the method of ROI estimation). The left upper point of the ROI is given as (r_u, r_v) in image coordinates as well as the height r_h and width r_w. The centroid is estimated based on the weights in the flow similarity image in the ROI.

$$n = \sum_{u=r_u}^{r_u+r_w} \sum_{v=r_v}^{r_v+r_h} w(u, v) \tag{3.69}$$

$$c_u = \frac{1}{n} \sum_{u=r_u}^{r_u+r_w} \sum_{v=r_v}^{r_v+r_h} w(u, v) \cdot u \tag{3.70}$$

$$c_v = \frac{1}{n} \sum_{u=r_u}^{r_u+r_w} \sum_{v=r_v}^{r_v+r_h} w(u, v) \cdot v \tag{3.71}$$

$$\mathbf{c}_f = \begin{pmatrix} c_u \\ c_v \end{pmatrix} \tag{3.72}$$

This centroid defines the centroid of the optical flow weights in the given ROI. In conclusion, it defines the middle point of the moving object in image coordinates. In section 3.4.1 a 3D wire frame model is presented which is used for the optimization based on edge fitting. The centroid of this model in model coordinates M is $\mathbf{c}_m = (0, 0, \frac{h_m}{2})^\top$ with h_m being the height of the model. With the transformations described in section 3.4.1, this model centroid is transformed into image coordinates $\mathbf{c}_h = (c_{h,u}, c_{h,v})^\top$ based on an initial parameter set $\beta = (\psi, s_x, s_y, s_z, t_x, t_y)^\top$.

$$\mathbf{c}_h = \text{cam2img}(\mathbf{T}_{C \leftarrow M} \mathbf{c}_m) \tag{3.73}$$

Based on the assumption that the optical flow centroid $\mathbf{c}_f$ defines the centroid of the moving vehicle in the image, a new error term is introduced to the Jacobian. The error term defines the distance from the hypothesis centroid $\mathbf{c}_h$ to the flow centroid $\mathbf{c}_f$. The distance in the v (height) axis of the image is neglected as it varies strongly based on the height of the vehicle. Thus, the error measurement becomes $d(\mathbf{c}_f, \mathbf{c}_h)$.

$$d(\mathbf{c}_f, \mathbf{c}_h) = c_{f,u} - c_{h,u} \tag{3.74}$$

The scaling parameters s_x, s_y, s_z and yaw angle ψ are not suitable for optimization using this error measurement as these are independent from the position of the centroid. The centroid is mainly dependent on the two parameters t_x and t_y. A new row is added to the Jacobian matrix to include the optimization based on the centroid distance. This row only contains two entries for t_x and t_y and the others are set to zero.

$$\mathbf{J}_i = \begin{pmatrix} 0 & 0 & 0 & 0 & \frac{\delta d}{\delta t_x} & \frac{\delta d}{\delta t_y} \end{pmatrix} \tag{3.75}$$

The partial derivatives are calculated similarly to the partial derivatives for edge distances. $\boldsymbol{\beta} = (\psi, s_x, s_y, s_z, t_x, t_y)^\top$ contains the initial hypothesis parameters. An ϵ environment of [-0.2, 0.2] showed reasonable results.

Partial derivative for parameter t_x:

$$\boldsymbol{\beta}_1 = \boldsymbol{\beta} \text{ with } t_x \to t_x - \epsilon \tag{3.76}$$

$$\boldsymbol{\beta}_2 = \boldsymbol{\beta} \text{ with } t_x \to t_x + \epsilon \tag{3.77}$$

$$\frac{\delta d}{\delta t_x} = \frac{d(\mathbf{c}_f, \mathbf{c}_h(\boldsymbol{\beta}_1)) - d(\mathbf{c}_f, \mathbf{c}_h(\boldsymbol{\beta}_2))}{2\epsilon} \tag{3.78}$$

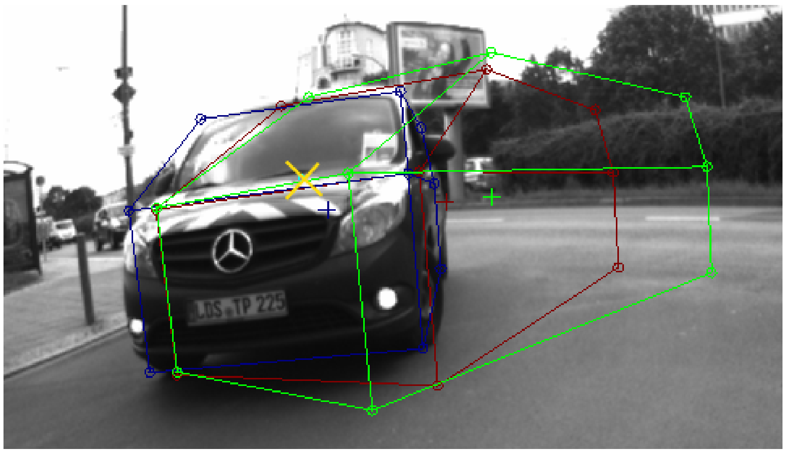
Partial derivative for parameter t_y:

$$\boldsymbol{\beta}_1 = \boldsymbol{\beta} \text{ with } t_y \to t_y - \epsilon \tag{3.79}$$

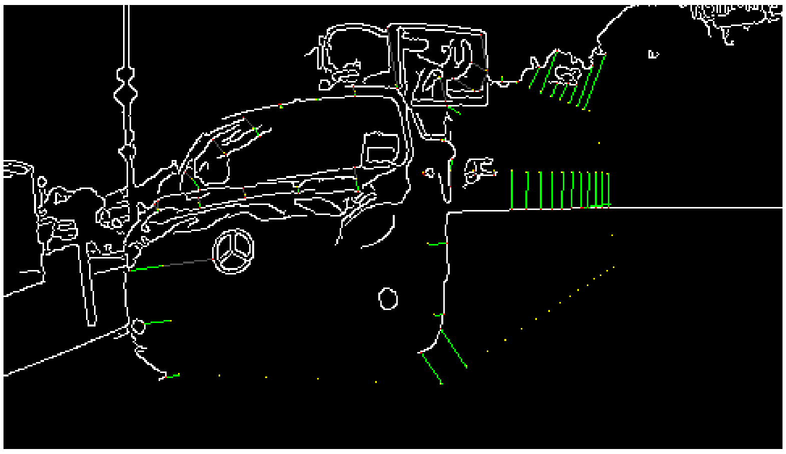
$$\boldsymbol{\beta}_2 = \boldsymbol{\beta} \text{ with } t_y \to t_y + \epsilon \tag{3.80}$$

$$\frac{\delta d}{\delta t_y} = \frac{d(\mathbf{c}_f, \mathbf{c}_h(\boldsymbol{\beta}_1)) - d(\mathbf{c}_f, \mathbf{c}_h(\boldsymbol{\beta}_2))}{2\epsilon} \tag{3.81}$$

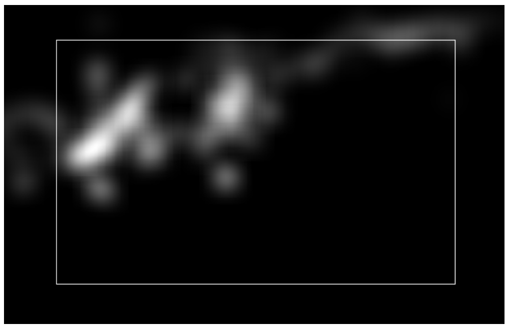
The scenario in Figure 3.14 shows a driveway onto a bigger street. The Mercedes on the left is turning into the street. The ego vehicle is moving with relatively low velocity into the driveway. The optical flow centroid of the Mercedes is marked with a big X. The green RADAR hypothesis is relatively far off the actual vehicle. Fitting solely based on edge distance fits the front but fails in the back of the van as there is no structure near the model edges. The blue model uses both weighting of edges based on flow as well as flow centroid fitting. The flow centroid is very far from the RADAR hypothesis (green cross). Accordingly, the model is moved in the direction of the flow centroid and fits to the edges of the vehicle nearly perfect.



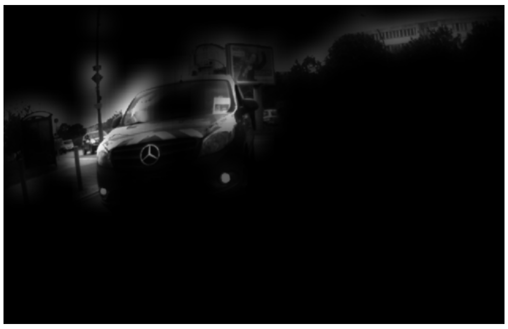
(a) Green: initial hypothesis, red: Model fitting without flow information, blue: Model fitting with flow centroid and flow weights, yellow: estimated flow centroid



(b) Canny edge image with annotated fitting distances of the red model



(c) Flow weight image



(d) Input image multiplied with flow weights

Figure 3.14: Model refinement using optical flow information

3.5 Results and Conclusion

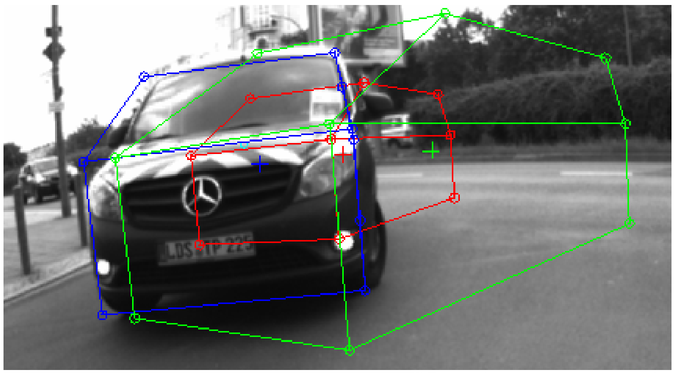
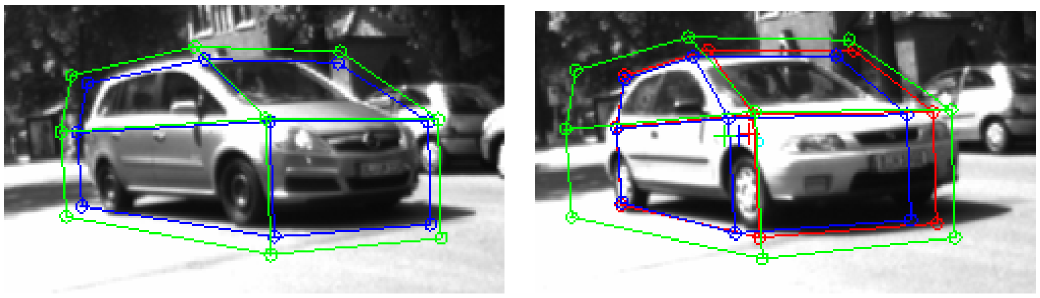


Figure 3.15: Recovered track with optical flow and edge fitting, green: starting hypothesis, red: edge fitting, blue: flow + edge fitting

The presented method works well for good starting hypotheses. The approach is able to recover positions which are less than 1 m off of the true position. Furthermore, fitting quality increases if there is less image structure around the vehicle so that the Canny image mostly contains edges of the vehicle. Especially the position correction presented in section 3.4.5 was able to recover tracks that are further off the initial hypothesis. An example is shown in Figure 3.15. Fitting based solely on edge information fails to fit the vehicle. The integration of flow information - flow centroid and flow weighting - could recover the true position (blue).

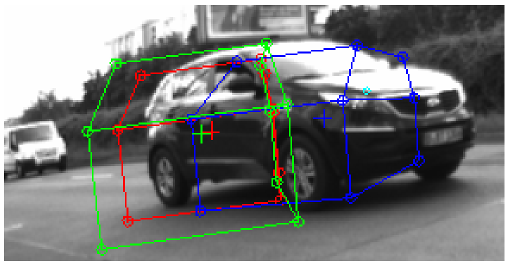


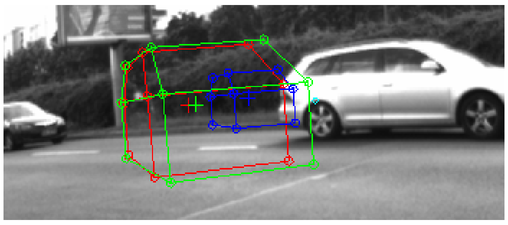
(a) Van-like vehicle **(b)** Different vehicle type

Figure 3.16: Model fitting of different vehicle types, green: starting hypothesis, red: edge fitting, blue: flow + edge fitting

The wire frame model which was used for edge fitting is very general and works best for van-like vehicles as seen in Figure 3.16a. Nonetheless, good fitting results are achieved with non-van type vehicles as well (Figure 3.16b).

However, even with optimizations based on optical flow, the convergence of the algorithm to a good solution which contains the vehicle is very dependent on the initial hypothesis. If the hypothesis does not cover most of the vehicle, the algorithm is very likely to converge to a solution which fits to edges that do not belong to the actual vehicle. The optimization tends to fit to parked cars at the side of the roads or to structures containing straight edges. Figure 3.17 shows


(a) Incorrect initial yaw angle


(b) Incorrect yaw angle and position estimation

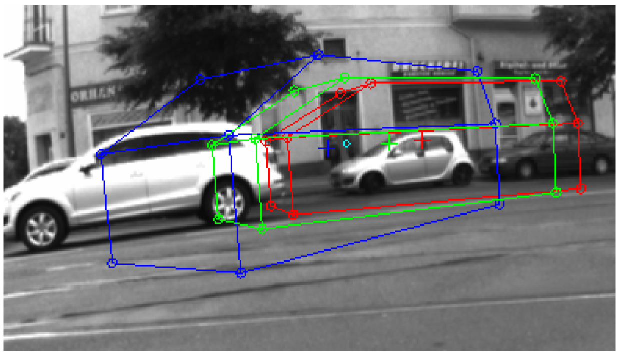

(c) Distance (green) to next edge point from the projected model points in the Canny filtered image

Figure 3.17: Incorrect fitting due to incorrect initial parameters

examples of insufficient initial hypotheses. The position estimation in Figure 3.17a is relatively accurate but the yaw angle has been estimated incorrectly. The model fitting is not able to recover as other edges are closer to the projected model lines. Figure 3.17 shows an example in which the algorithm fits to the edges of the signpost and the rear end of the vehicle which actually belongs to the tracked object. In Figure 3.17c the optimization which takes optical flow into account converges further into the direction of the vehicle which should be tracked but is still too far away from the desired fit.

Another reason for inaccurate fitting is the generality of the chosen vehicle model. Even though the actual fit covers the vehicle, it does not correctly fit to it as seen in Figure 3.18. The truck does not fit to the model, and the variance allowed in height and length for the scaling factors is not large enough. However, the flow weighting allows the algorithm to fit to edges which are outside of the initial parameter set, so that the length scaling factor is increased. The model fitted to the vehicle in Figure 3.18b does not cover the hood part. Thus, it fits partly to the hood and partly to the A-pillar.

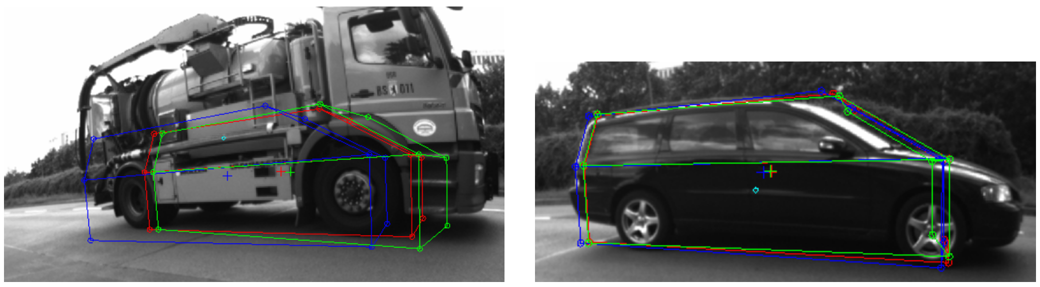

(a) Vehicle type does not match the abstract vehicle model **(b)** The abstract vehicle model does not cover the hood of the vehicle

Figure 3.18: Incorrect fitting due to abstraction in the chosen vehicle model

In conclusion, advantages of the approach are the computing performance compared to pattern based approaches and the lack of a training phase. With accurate starting hypotheses, the algorithm is able to converge to an accurate refinement of the initial parameters. The optical flow based weighting has shown to improve the fitting process and the flow centroid estimation increases the probability to fit to the actual vehicles if starting hypotheses are further away from the optimal solution. Even though the technique showed good results in many cases, it requires good edge extractions and accurate initial hypotheses.

3.6 Future Work

Vehicle detection and tracking in side cameras is an area less explored than the detection in front and rear cameras. Therefore there are a lot of opportunities and research questions.

The presented approach relies on an accurate extrinsic camera calibration. The current calibration stand at Hella Aglaia however, focuses on front and rear camera calibration. A thought experiment for a simple and inexpensive solution to extend the calibration stand with a side camera calibration has been presented in the appendix on page 108. Additionally, approaches to dynamically adapt the extrinsic calibration while driving may be another area of research.

Certainly, a goal is to include the refined positions of the camera in the RADAR tracker. This should lead to better initial hypotheses as the camera information is fused back into the tracker. The fitting approach converges to inaccurate solutions sometimes, therefore a reasonable outlier detection has to be implemented.

Another area of research which might increase the accuracy of fitting is the choice of features, e.g. a separate wheel detection might return a set of potential wheel positions to fit the model to. Furthermore [LM09] uses complexer 3D-models which generalize better to different vehicle types. As more computing power will be available in up-coming production vehicles, this approach should be taken into consideration as well.

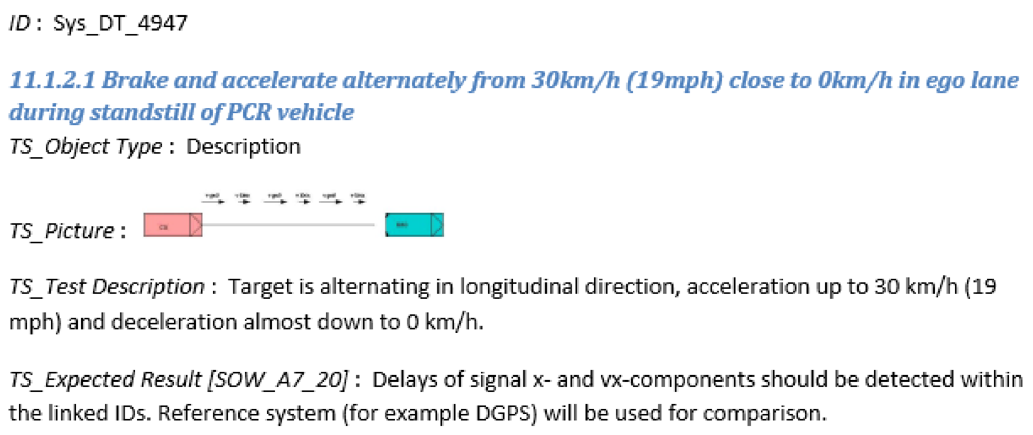
Chapter 4

Synthetic Ground Truth Data Generation for Radar Object Tracking

This chapter describes an intuitive method to generate artificial test data for RADAR-based crash warning algorithms developed at Hella Aglaia. At the moment, two warning algorithms are in development – FSR and PCR. Each of these systems has special requirements that have to be fulfilled. Driving scenarios with defined behavior of the warning algorithms are compiled into a test catalog (example: Figure 4.1). Some of the described maneuvers require advanced driving skills and consume a lot of time to record properly. Instead, a simpler method to generate the data which would be acquired from test drives is preferable for development. However, the data has to be physically reasonable and consistent with the test catalog.

The data flow of the warning systems consists of three main processing steps. At first raw targets from the RADAR sensors are collected and classified. In the second step these RADAR targets are grouped together, filtered and tracked as moving objects. The moving objects from the tracker are used to estimate crash probabilities in the warning algorithm by forward prediction of movements (Figure 4.2).

There are several possibilities for the generation of test data in the whole process. One could either generate raw targets for the tracking algorithm or generate



ID : Sys_DT_4947

11.1.2.1 Brake and accelerate alternately from 30km/h (19mph) close to 0km/h in ego lane during standstill of PCR vehicle

TS_Object Type : Description

TS_Picture :

TS_Test Description : Target is alternating in longitudinal direction, acceleration up to 30 km/h (19 mph) and deceleration almost down to 0 km/h.

TS_Expected Result [SOW_A7_20] : Delays of signal x- and vx-components should be detected within the linked IDs. Reference system (for example DGPS) will be used for comparison.

Figure 4.1: Example of a test catalog scenario for PCR

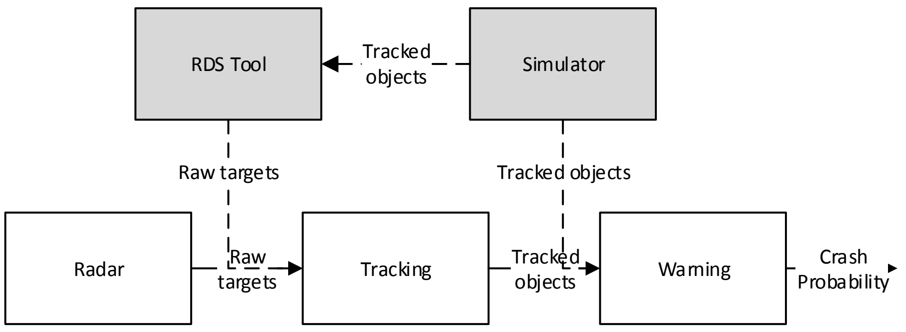
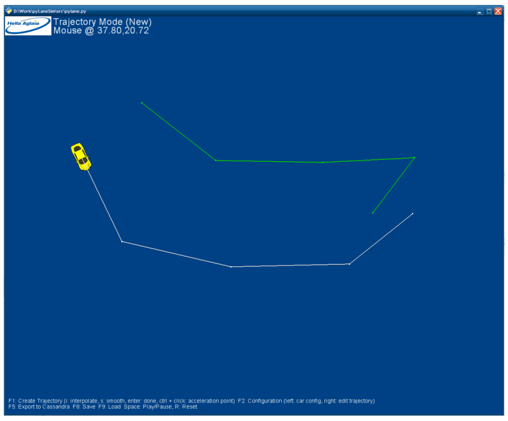


Figure 4.2: Data flow of warning algorithm (white) and supply of artificial test data through pyLaneSim and RDS Tool (gray)

tracking objects for the actual warning algorithm. Having test data at either step, it is possible to create independent module tests. This chapter will focus on generating ground truth data for the tracker, i.e. positions and velocities of moving vehicles in the environment as well as other physical properties which are usually generated by the RADAR tracker. Additionally, Hella also develops the RawDataSynthesis (RDS) Tool which is able to simulate radar sensor data given information about objects in the environment. It models the physical process of electromagnetic waves traveling and reflecting in the environment to generate raw data for the sensor itself. In the last section an interface between the simulator and the RDS Tool is presented. Thus, it is possible to create test data for the tracker module based on the scenarios designed in the simulator (Figure 4.2). This renders a form of development possible in which development of one module, e.g. the warning algorithm is not dependent on the availability of the preceding modules. Thus, tracker and warning algorithm development could be started at the same time without necessity of the output of the tracking algorithm as artificial test data could be supplied through the simulator.

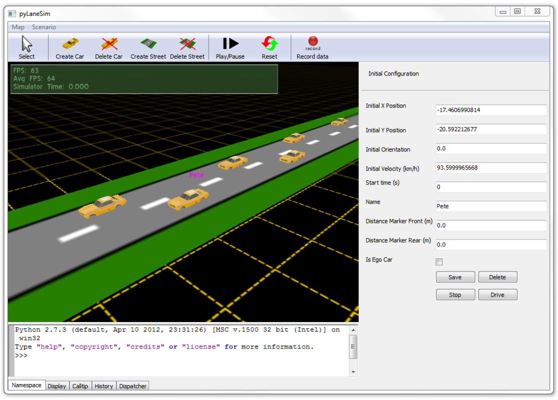
4.1 User Interface



Trajectory Mode (New)
Mouse @ 37.80,20.72

F1: Create Trajectory (i: interpolate, s: smooth, enter: done, ctrl + click: acceleration point) F2: Configuration (left: car config, right: edit trajectory)
F5: Export to Cassandra F8: Save F9: Load Space: Play/Pause, R: Reset

(a) Old interface of the simulator



FPS: 63
Avg FPS: 64
Simulator Time: 0.000

Initial Configuration

Initial X Position	-17.4606990814
Initial Y Position	-20.592212677
Initial Orientation	0.0
Initial Velocity (km/h)	93.5999965668
Start time (s)	0
Name	Pete
Distance Marker Front (m)	0.0
Distance Marker Rear (m)	0.0
Is Ego Car	☐

```
Python 2.7.3 (default, Apr 10 2012, 23:31:26) [MSC v.1500 32 bit (Intel)] on
 win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

(b) Current interface

Figure 4.3: Interface development of the simulator

A first prototype features a 2–dimensional top view interface with mouse support in which the trajectories of cars can be created by simple clicks on the ground plane. Additionally these paths can be annotated with acceleration information (Figure 4.3a). The resulting trajectories might be physically unreasonable, e.g. a hairpin turn with 40 km/h. Such being the case, a car dynamics

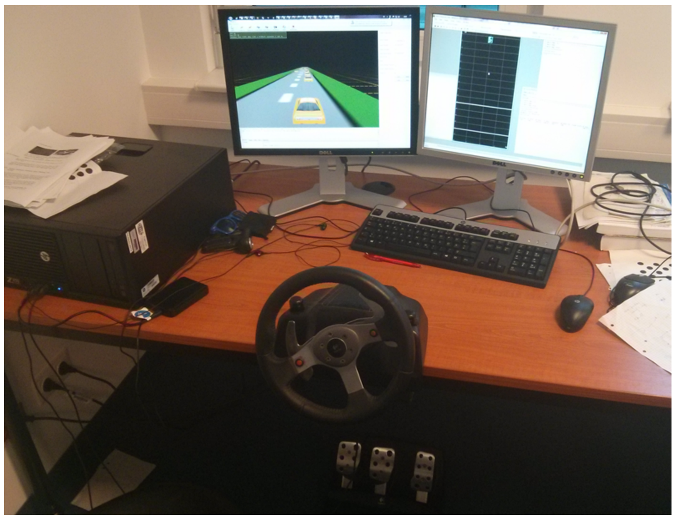


Figure 4.4: Set up of the driving simulator

model (see section 4.3) is introduced and a Proportional-Integral-Derivative controller (PID-controller) connected to the steering angle which keeps the vehicle on the trajectory. The PID-controller returned a steering angle based on the normal distance of the car to the trajectory. The parameters for the controller were chosen by intelligently searching the parameter space using the Twiddle algorithm presented in Sebastian Thrun's Stanford online lecture "How To Build A Robotic car" [Thr13]. The annotated acceleration information at the trajectory points are fed into said model which tries to follow the predefined trajectory. Unfortunately, this approach renders inconvenient for more complex scenarios that involve multiple cars. Many repetitions and refinements are needed to achieve precise timing.

The second prototype features a 3-dimensional interface and lets the user control cars from a cockpit or world view (Figure 4.3b). The most intuitive way to acquire data from driving scenarios is to actually drive. Thus, the new interface features steering wheel and pedal or keyboard input if no driving wheel is present (Figure 4.4). Cars and street parts can be freely positioned in the environment on the ground plane. Cars and their properties can be defined freely and controlled one after another by rerunning the simulation for each car.

The test cases usually define the environment as well, e.g. a crossroad or a four lane high way. The information about the environment does not influence the test data itself but is used for visualization. Therefore the simulator supports basic environment mapping with predefined map tiles aligned on a virtual grid. These map tiles can be positioned on the ground plane by mouse and connected to each other to form a road scenario. Figure 4.5 shows a rather complex scenario created using the mapping capability of the simulator.

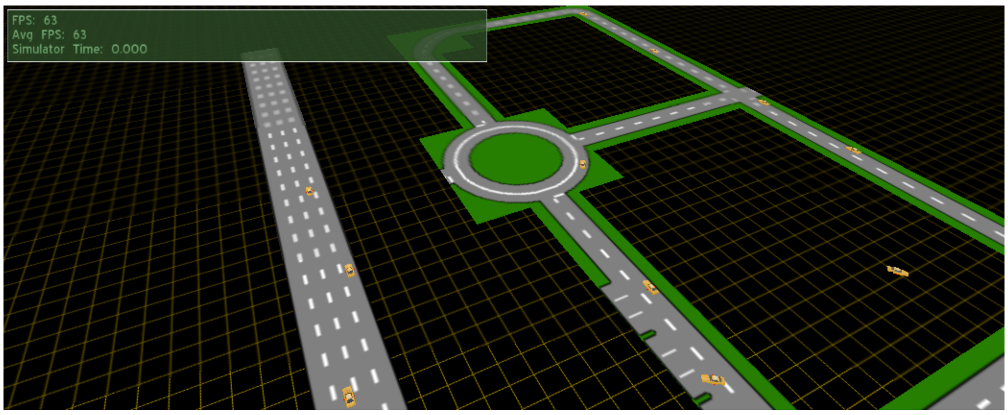


Figure 4.5: Complex road scenario created in the simulator

4.2 Implementation and Utilized Libraries

The simulator has been implemented in Python [Pyt13] which is an interpreted object oriented language. During the development of the simulator, it should be possible to implement new features very fast to try different ways of input and processing. Python allows very fast prototyping, offers a lot of functionality and can be easily extended by other libraries, e.g. for math or 3D visualization. The underlying physical models include several matrix operations for which NumPy [Num13] is used. NumPy includes, but is not limited to, a linear algebra library for scientific computing. The user interface was created using wxPython [wxP13] which also allows to embedded an Ogre instance. Python-Ogre are bindings for the famous open source 3D engine Ogre3D [Tor13]. The libraries have been carefully chosen to have low restrictions in their licenses as well as an active development and cross platform support.

Table 4.1 shows the licenses under which the used libraries are available. All licenses allow distribution in binary form and for commercial use.

In general, the simulator is divided into two parts. The user interface with the 3-dimensional Ogre visualization and the simulator core which manages simulated objects and their underlying physical models in a 2 dimensional x, y-plane. Ogre

Software	**License**
Python	PSF License
NumPy	BSD License
wxPython	wxWindows License (similar to LGPL)
Python-Ogre	LGPL

Table 4.1: Licenses of used libraries in the simulator

itself manages its 3-dimensional objects in a tree structure. The root node is the world reference frame. Each sub node is positioned in the parent's reference frame and has its own reference frame which can be rotated, scaled or moved. An easy example is the implementation of a cockpit view for one of the simulated cars. At first a `CarNode` is created for the car which contains the 3-dimensional model in an Ogre compatible format. The node is freely positioned as a sub node of the world node (root node) in the x, y-plane and updates itself in each render cycle by fetching the (x, y) position and yaw angle ψ from the respective simulator object. Instead of recalculating the cockpit camera position each time, a `CameraNode` is added as a sub node of the `CarNode` and positioned in its reference frame. The Ogre tree automatically updates the positions of the sub nodes if the higher nodes are changed.

Each `CarNode` holds a reference to a `SimulatorObject` which represents the underlying physical model. A `Simulator` instance holds references to all available `SimulatorObject`s and manages them with respect to the current simulation time. Each `SimulatorObject` has to implement a few general methods to be manageable by the `Simulator`. These methods return information about current position and orientation as well as speed. Currently, there is one concrete implementation – the `DriveCar` – which represents the dynamics of the underlying vehicle model.

This design is visualized as an UML class diagram in Figure 4.6. The diagram is simplified to the most important methods and classes for the sake of clarity.

4.3 Car Dynamics Models

To achieve physically-reasonable results for the dynamics of vehicles in the simulator, a dynamics model is introduced which models the behavior of cars based on steering information from the steering wheel or keyboard input. The simulator

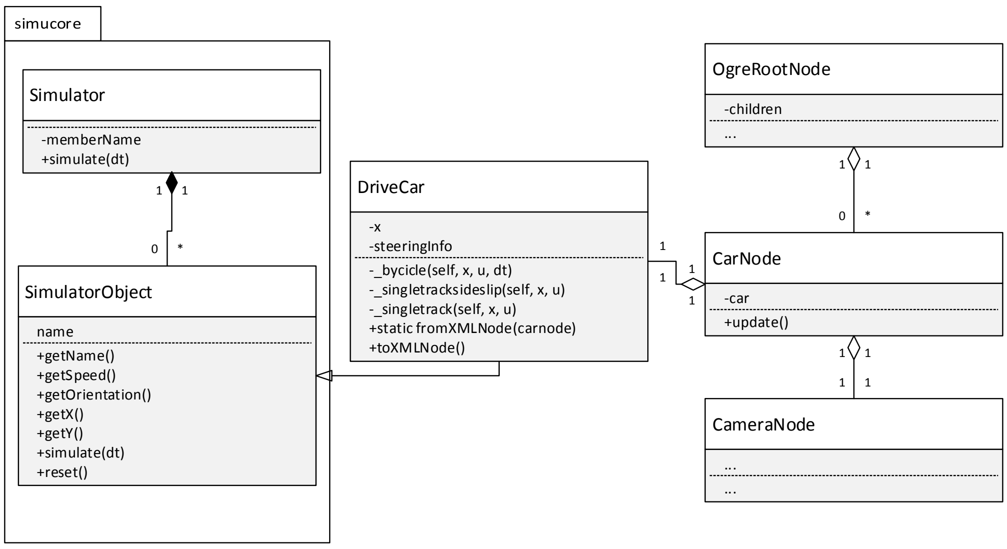


Figure 4.6: Simplified UML representation of the simulator structure

runs in a loop calculating the state $\mathbf{x}$ of each dynamic object every $25ms$ taking the current state and new steering information into account.

In the following, the dynamics models implemented in the simulator are explained. A single track model which models tire slip, as well as over- and understeer can be seen in Figure 4.7. Each of these models assumes the world to be flat, thus, the position of a car is limited to a position within the $x-y$-plane and its yaw angle. Roll and pitch information are discarded. The coordinate systems used are explained in section 1.3 Preliminaries. The world coordinate system is a right handed reference system in which the ground plane is represented by the $x-y$-plane and the z-axis points upwards. The dynamics models define their own right handed ego reference frame with the x-axis aligned with the vehicle's length axis pointing into the driving direction and the y-axis positioned on the front axle. The z-axis points upwards.

The model contains several parameters and physical properties. These contain the current position (x_w, y_w) of the object in world coordinates. ψ defines the yaw angle (orientation) while $\dot{\psi}$ defines the respective change rate. $\psi + \beta$ is the angle in which the actual motion is directed to. This angle differs to the yaw angle ψ if under- or oversteer are present which is represented by the slip angle β. The vehicle's mass is annotated as m.

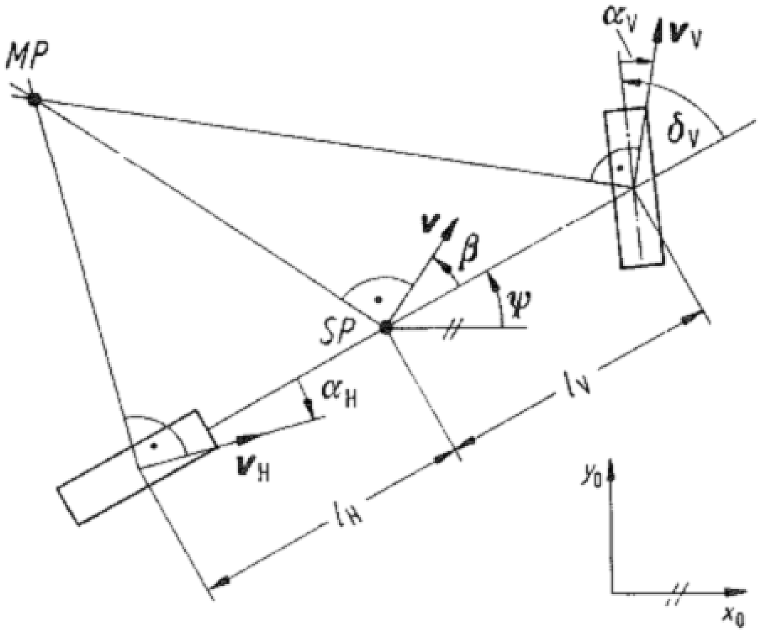


Figure 4.7: Single track model with all parameters aligned with world reference frame [Kau12, Slide 29]

l_v and l_h define the distance of the gravity center to the front and back. Modeling tire slip can be parametrized through cornering stiffness c_v and c_h for wheels on the front axle and rear axle. The reaction to steering can be influenced by the yaw moment of inertia parameter θ. A dynamic object can be controlled by a steering angle δ and an acceleration a_x in the object's ego reference frame in the direction of the x–axis which is aligned along the length axis of the car. A short summary of these car specific parameters required for the following models can be found in Table 4.2.

Symbol	Description	Unit
m	car mass	kg
l_v, l_h	distance to front (v), distance to back (h) from mass center	m
c_v, c_h	Cornering stiffness	$\frac{N}{Rad}$
θ	Yaw moment of inertia	$\frac{kg}{m^2}$
δ	Steering angle	Rad
v_x	Speed	m/s

Table 4.2: Car specific parameters for car dynamics models

Based on these models, the state of a dynamic object in the simulator consists of the following state vector $\mathbf{x} = (x_w, y_w, \psi, \dot{\psi}, \beta, v_x)^\top$.

The two control variables – steering angle δ and acceleration a_x – are combined in the control vector $\mathbf{u} = (\delta, a_x)^\top$.

4.3.1 Bicycle Model

The simplest vehicle dynamics model is possibly the bicycle model. It holds reasonable for velocities and steering angles that would not introduce tire slip and thus, over- or understeer. The only car property that is taken into account is the length $l = l_v + l_h$ of the car.

The system can be modeled by a function f which takes the current state $\mathbf{x}_t$, the new controls $\mathbf{u}_t$ and the time to simulate Δt as arguments.

$$\mathbf{x}_{t+1} = \mathrm{f}(\mathbf{x}_t, \mathbf{u}_t, \Delta t) \tag{4.1}$$

For very small steering angles, a straight line motion can be assumed. The new state variables for the next step Δt are calculated through f as it follows.

$$\begin{align} x_{w,t+1} &= x_{w,t} + \cos\psi_t \cdot v_{x,t} \cdot \Delta t \tag{4.2} \\ y_{w,t+1} &= y_{w,t} + \sin\psi_t \cdot v_{x,t} \cdot \Delta t \tag{4.3} \\ \psi_{t+1} &= \psi_t + \dot{\psi}_t \cdot \Delta t \tag{4.4} \\ \dot{\psi}_{t+1} &= \tan\delta_t \cdot \frac{v_{x,t}}{l} \tag{4.5} \\ \beta_{t+1} &= 0 \tag{4.6} \\ v_{x,t+1} &= v_{x,t} + a_{x,t} \cdot \Delta t \tag{4.7} \end{align}$$

For larger steering angles, the motion is estimated by the actual bicycle model. In the first step, the radius of the curve which would be driven with the given steering angle δ is computed. From this, the new position can be calculated.

$$r = \frac{l}{\tan \delta_t} \tag{4.8}$$

$$x_{w,t+1} = x_{w,t} + r(-\sin \psi_t + \sin(\psi_t + \dot{\psi}_t \cdot \Delta t)) \tag{4.9}$$

$$y_{w,t+1} = y_{w,t} + r(\cos(psi_t - \cos(\psi_t + \dot{\psi}_t \cdot \Delta t)) \tag{4.10}$$

$$\psi_{t+1} = \psi_t + \dot{\psi}_t \cdot \Delta t \tag{4.11}$$

$$\dot{\psi}_{t+1} = \tan \delta_t \cdot \frac{v_{x,t}}{l} \tag{4.12}$$

$$\beta_{t+1} = 0 \tag{4.13}$$

$$v_{x,t+1} = v_{x,t} + a_{x,t} \cdot \Delta t \tag{4.14}$$

4.3.2 Single Track Model with Tire Slip

The bicycle model does not contain more properties of the vehicle's physics than the length. However, the behavior of the car dynamics, especially in situations with higher velocities and strong cornering, depends on the distribution of mass. Thus, two new parameters l_v which is the distance of the center of gravity to the front of the car and l_h which is the distance to the back of the car are introduced. The model assumes the car to be a rigid body and shifts the center of gravity to an arbitrary position along the length axis of the car.

Instead of analytically solving the state from one time point to the next, a linearized model for a given time point t is utilized assuming that the state change $\dot{\mathbf{x}}$ can be approximated linearly for a small time frame Δt. Hence, the derivative $\dot{\mathbf{x}}$ of the state is linearized as it follows.

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \dot{\mathbf{x}}_t \cdot \Delta t \tag{4.15}$$

A simple linearized form of the Single track model was taken from the Daimler Autonomous Systems lecture [Bre12, Slide 43]. *ELG* is defined as the steer gradient and depends on the geometry of the car.

$$ELG = m \cdot \frac{c_h \cdot l_h - c_v \cdot l_v}{c_v \cdot c_h \cdot l} \tag{4.16}$$

$$\dot{\mathbf{x}} = \begin{pmatrix} \dot{x}_w \\ \dot{y}_w \\ \dot{\psi} \\ \ddot{\psi} \\ \dot{\beta} \\ a_x \end{pmatrix} = \begin{pmatrix} v_x \cdot \cos\psi \\ v_x \cdot \sin\psi \\ \frac{\delta \cdot v}{l + ELG \cdot v_x^2} \\ 0 \\ 0 \\ a_x \end{pmatrix} \tag{4.17}$$

4.3.3 Single Track Model with Over and Understeer

The previous model already gave a good idea of the behavior of a car depending on its physical properties. A more refined model which includes under- or oversteer can be found in [SHB10]. Even though the system models the behavior of a car in more detail than the other models, it also comes with more restrictions. In general the model is valid for constant velocities and small steering angle changes. The system is modeled through a linearized system matrix $\mathbf{A} \in \mathbb{R}^{6\times 6}$, which defines the system itself, and a control matrix $\mathbf{B} \in \mathbb{R}^{6\times 2}$ which takes the influence of steering information into account. The state change can be expressed in the following way.

$$\dot{\mathbf{x}}_t = \mathbf{A}\mathbf{x}_t + \mathbf{B}\mathbf{u}_t \tag{4.18}$$

$$\mathbf{x}_{t+1} = \mathbf{x}_t + \dot{\mathbf{x}}_t \cdot \Delta t \tag{4.19}$$

[SHB10] describes the state only as $(\beta, \dot{\psi})$, thus the matrices **A** and **B** have to be extended for the other state variables which can be expressed as it follows.

$$\mathbf{A} = \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & \cos(\psi+\beta) \\ 0 & 0 & 0 & 0 & 0 & \sin(\psi+\beta) \\ 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{v} \cdot \frac{c_v \cdot l_v^2 + c_h \cdot l_h^2}{\theta} & -\frac{c_v \cdot l_v - c_h \cdot l_h}{\theta} & 0 \\ 0 & 0 & 0 & -1 - \frac{1}{v^2} \cdot \frac{c_v \cdot l_v - c_h \cdot l_h}{m} & -\frac{1}{v} \cdot \frac{c_v + c_h}{m} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{4.20}$$

The influence of the control input vector $\mathbf{u}$ to the whole system can be expressed in the new matrix $\mathbf{B}$.

$$\mathbf{B} = \begin{pmatrix} 0 & 0 \\ 0 & 0 \\ 0 & 0 \\ \frac{c_v \cdot l_v}{\theta} & 0 \\ \frac{1}{v} \cdot \frac{c_v}{m} & 0 \\ 0 & 1 \end{pmatrix} \tag{4.21}$$

The restrictions of the model are violated in the simulator in some cases when an acceleration a_x and larger steering angle changes are introduced. During experimentation it showed that, with the given set of vehicle parameters, especially with low velocities ($v_x < 30$ km/h) and large steering angle changes, as well as with quickly alternating steering angles around 0°, the system tends to oscillate and turn into chaotic unrealistic states. This is caused by violating the assumptions made by the model, i.e. $\sin\delta$ is approximated by δ and the constant velocity assumption. This can be seen as the car rotating chaotically around its axis in the simulator. Experiments have shown that for higher velocities and lower sample rates (switched from 50 ms to 25 ms) the chaotic state changes do not occur. The model can only be applied if the violations do not have a strong influence on the state, therefore, the simulator changes from the simpler single track model explained in section 4.3.2 to this extended model only if the velocity reaches a certain threshold $v_x > 30$ km/h.

4.4 Ground Truth Data Output Formats

The following sections will explain the two main export formats that are supported currently. External applications that should use the ground truth data from the simulator usually support their own specific input format. Thus, the simulator supports adding formats by an exporter base class which has full access to the simulator core and has to implement three methods – `startRecording()`, `recordStep()` and `stopRecording()`. The first should set up basic data structures and store static information while the last should store all the collected information in the respective export data files. `recordStep()` is called repeatedly for each simulation cycle during a simulation and should handle conversion of coordinates and vehicle attributes for each time step.

4.4.1 Internal Data Format

All attributes of cars are stored internally in world coordinates, i.e. an *x*-*y*-plane with 1 unit equal to 1 meter. A scenario is defined as a constellation of cars and their respective motions and can be stored in an eXtended Markup Language (XML) data file (listing 4.1). For each car in the simulator a new car node is created which defines its initial attributes. The steering information (steering angle, gear, throttle and brake values) are stored as a new sub node under the `controlRecords` node for each time step.

```xml
<scenario>
    <cars>
        <car distanceFront="0.0" distanceRear="22.9999995232" isego="1" l="4.0" mRatio="0.3" name="tanja" orientation="-3.14159297943" type="DriveCar" v="0.0" x="-52.5424861908" y="-1.9460067749">
            <controlRecords />
        </car>
        <car distanceFront="0.0" distanceRear="0.0" l="4.0" mRatio="0.3" name="nora" orientation="-3.14159297943" type="DriveCar" v="27.7777767181" x="14.9835205078" y="-1.79802703857">
            <controlRecords>
                <control alpha="-0.00254073047638" brake="-0.0" gear="1" throttle="-0.0" timeStep="0" />
[...]
            </controlRecords>
        </car>
    </cars>
</scenario>
```

Listing 4.1: Scenario description file (*.scn) of the simulator

Users might want to have different scenarios in the same specific environment, e.g. a crossing. Thus, the street maps are stored separately in another file (listing 4.2). As the map consists of square parts positioned on a virtual grid, the state can be described by the center position and type of each map tile. Each map tile type can be identified by an unique id, thus, only id, position and orientation have to be saved.

```
1 45.000000 15.000000 0.000000
4 60.000000 15.000000 0.000000
1 75.000000 15.000000 4.712389
4 75.000000 0.000000 4.712389
```

Listing 4.2: Map description file (*.map) of the simulator. Format: unique id, center position (x, y) and orientation in rad

In general, map files and scenario files can be loaded independently into the simulator.

4.4.2 PCR and FSR Formats

To use the generated scenarios in Hella Aglaia's warning algorithms, they have to be converted into respective data formats. Both PCR and FSR use similar formats based on Comma Separated Values (CSV) files. A meta file ([name].pcr, listing 4.3) describes the format version and which ego data file and object data file belong to the simulation. Additionally, it contains the number of frames available for the simulation. A frame is a snapshot of the simulator objects every 50 ms.

```
{
  PCR_Version = 1U;
  StartFrame = 0;
  EndFrame = 330;
  EgoFile = "11-6-1-1-1dynamic-sheerPCR50kmh-ego80kmh.pcrEgoCsv";
  ObjectFile = "11-6-1-1-1dynamic-sheerPCR50kmh-ego80kmh.pcrObjCsv";
}
```

Listing 4.3: Meta description file for PCR and FSR simulation files

The ego file ([name].pcrEgoCsv, listing 4.4) contains information about the ego vehicle's attributes in each frame. These include velocity, acceleration, yaw rate, v_x, v_y and the lateral acceleration in ego coordinates. Furthermore, the left and right turn signal are encoded in the last field with 0: no signal, 1: left signal, 2: right signal and 3: both turn signals. The signal indicator is used to determine potential turning or lane changing of the ego vehicle driver in the FSR warning algorithm.

```
Frame; Speed; Acceleration; YawRate; vx; vy; accLat; signals
1; 13.85139; -1.50002; 0.00007; 13.85139; 0.00002; 0.00000; 0
2; 13.77639; -1.50002; 0.00004; 13.77639; 0.00004; 0.00000; 0
3; 13.70139; -1.50002; 0.00003; 13.70139; 0.00004; 0.00000; 0
[...]
```

Listing 4.4: Ego motion description file for PCR and FSR simulation files

The object motion file ([name].pcrObjCsv, listing 4.5) is similar. For each frame, the attributes of the non-ego cars are listed identified by their respective object id. All information are again in the ego vehicle's coordinate frame.

```
Frame; ObjectId; x; y; vx; vy; ax; ay;
1; 1; −24.00269; 3.75115; 5.55556; 0.00000; 0.00000; 0.00000;
1; 2; −10.36594; 20.52808; 2.77778; 0.00000; 0.00000; 0.00000;
2; 1; −23.72491; 3.75115; 5.55556; 0.00000; 0.00000; 0.00000;
2; 2; −10.22705; 20.52808; 2.77778; 0.00000; 0.00000; 0.00000;
...
```

Listing 4.5: Object motion description file for PCR and FSR simulation files

4.4.3 RawDataSynthesis Format for synthetic Radar Raw Target Generation

Based on the scenario data generated by the simulator, corresponding RADAR raw targets can be computed by the RDS Tool. It was developed by another student at Hella and used for the generation of raw targets from object motions. Similar to the first prototype of the simulator the trajectories of vehicles are based on given way points and speed markers. These are then combined to splines. The orientation of a car is implicitly calculated based on the direction of the trajectory. Thus, each support point of the spline contains the (x, y) position and a velocity v. Between two support points the system will interpolate the velocity linearly. Nevertheless, the trajectories do not include any physical constraints.

The simulator itself allows more degrees of freedom through the higher phase space and simulates vehicles within certain physical constraints. However, a close approximation can be achieved by creating many support points, i.e. sampling velocity and position from the simulator every 50 ms for each object. Sampling support points with this approach results in splines that are approximately the same as the trajectories created in the simulator. Albeit, there is one shortcoming to this approach. When the velocity reaches zero, no new way points can be sampled at the same point and there is no possibility to decide in these splines how long the velocity should stay at zero, e.g. when modeling stopping and starting at a crossing. This is in general not possible to design in the RDS tool.

The format (listing 4.6) is based on XML. It contains definitions of the track, which is currently only used for visualization in the RDS tool itself, thus, no information is taken from the simulator to model it as it is more restrictive than the simulator map design. Additionally, the `vehicles` node contains sub nodes for each vehicle that contain the described way points (spline support points). The ego vehicle is stored explicitly with more information about sensor mountings and calibration.

```
<?xml version="1.0" encoding="utf-8"?>
<Scenario simulateADCValues="True" simulateRawTargets="True">
    <Track>
        <TrackPart guardRailType="Single" guardRails="" laneWidth="
    3.8" lanes="4" length="1000" radius="0" />
    </Track>
    <Vehicles>
        <Vehicle>
            <Waypoints>
                <Waypoint speed="16.6666660309" x="1.27323913574" y="
    -20.3797801494" />
                <Waypoint speed="16.6666660309" x="1.27326160847" y="
    -19.5464468482" />
                 [...]
            </Waypoints>
            <VehicleData name="nora">
                <VehicleDimensions length="4.0" width="1.8">
                    <Axis x="0.8" />
                    <Axis x="3.6" />
                </VehicleDimensions>
            </VehicleData>
        </Vehicle>
    </Vehicles>
    <EgoVehicle>
        <Waypoints>
            <Waypoint speed="0.0" x="1.84082021713" y="43.444070816"
    />
        </Waypoints>
        <VehicleData name="tanja">
            <VehicleDimensions length="4.0" width="1.8">
                <Axis x="0.8" />
                <Axis x="3.6" />
            </VehicleDimensions>
            <Sensors>
                 [Definition of Sensor calibration and mounting
    positions]
            </Sensors>
        </VehicleData>
    </EgoVehicle>
</Scenario>
```

Listing 4.6: Export format of RDS Tool (*.rds)

4.5 Results and Conclusion

The PCR test catalog consists of 155 test cases for the PCR warning algorithm (example: Figure 4.1). This catalog is used as a basis to evaluate the usability of the simulator. Altogether it took 3 days for a single person to recreate the test catalog in the simulator and save a scenario file for each test case.

Every scenario of the test catalog has been exported to FSR and PCR warning algorithm formats as well as to the RDS tool. With the latter it is even possible to test the system at the raw target level. Both PCR and FSR delivered reasonable results when running the synthetic scenarios from the test catalog. Albeit, RDS showed problems in situations that include a full stop of a vehicle followed by an acceleration which cannot be modeled in the tool as already mentioned and explained in section 4.4.3. This special case appeared only in one test case, and therefore can be neglected.

The simulator is a valuable tool to cut down both time and costs for the generation of reasonable test data. It can be extended to support further export formats by adding or adjusting exporter classes. Additionally, the simulation of other objects, e.g. motor bikes or pedestrians, can be achieved by implementing more `simulatorObject` classes with other underlying physical models.

As an outlook into the future, the raw data synthesis from the RDS tool could be merged directly into the simulator to skip the export and import step. This would eliminate the restrictions of RDS induced by velocities of 0 km/h as well.

Chapter 5

Conclusion

The results and conclusions for each topic are placed in detail at the end of each respective chapter. Altogether, three main topics have been processed.

In Chapter 2, the choice of a reasonable camera system and the setup of a working framework to collect data from test drives with the test bed have been presented. Cameras from AVT have been chosen and successfully calibrated using the Scaramuzza camera model [SMS06a]. This model proved practical in all cases and for all further applications.

In Chapter 3, a sensor fusion approach utilizing camera and RADAR information has been presented. This approach showed an improvement of the accuracy of the RADAR object tracks if the initial RADAR hypotheses are relatively accurate estimates of the true position. The approach has been successfully extended to integrate optical flow information.

The last part of this thesis discusses a method for intuitive generation of test data for RADAR warning systems. Instead of collecting information from test drives equipped with the warning system, data can be acquired from a driving simulator. Next to the development of an intuitive interface, the data had to be physically reasonable. Therefore, vehicle dynamics have been modeled in the simulator which ensure this level of data quality. Combined with another project at Hella, the output of the simulator can be used to test warning system modules both at pre-processing as well as warn algorithm level.

List of Figures

List of Tables

Abbreviations

AVI Audio Video Interleave (Audio and video container format) 39

CAN Controller Area Network 16

CLIO CAN/LIN Input/Output - Module (Device which translates CAN messages to FireWire and tags the data with FireWire timestamps) 18

CSV Comma Separated Values 83

FSR Front/Side RADAR (Tracking and warning system mainly for cross sections) 1

KP Communication Processor 37

LIDAR Light Detection And Ranging 12

KLT Tracker Kanade-Lucas-Tomasi Tracker 47

PCA Principle Component Analysis 12

PCR Pre-Crash Rear (Warning system for rear crashes) 1

PID-controller Proportional-Integral-Derivative controller 73

RADAR Radio Detection And Ranging 1

RDS RawDataSynthesis (tool for generating raw RADAR targets from object tracks) 72

ROI Region of Interest 41

SVM Support Vector Machine (Classifier) 11

ToF Time-of-Flight (used in sensors for distance measurement) 3

XML eXtended Markup Language 82

Bibliography

[ABC07] Giancarlo Alessandretti, Alberto Broggi, and Pietro Cerri. Vehicle and guard rail detection using radar and vision data fusion. *IEEE Transactions on Intelligent Transportation Systems*, 8(1):95–105, March 2007.

[All12] Allied Vision Technologies. Data sheet AVT Guppy Pro, September 2012.

[Bou01] Jean-Yves Bouguet. Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm. 2001.

[Bre65] Jack E Bresenham. Algorithm for computer control of a digital plotter. *IBM Systems journal*, 4(1):25–30, 1965.

[Bre12] Dr. Gabi Breuel. Grundlagen autonomer Fahrzeuge - Situationsanalye, 2012.

[Can86] John Canny. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, (6):679–698, 1986.

[ETG07] Tobias Ehlgen, Markus Thom, and Markus Glaser. Omnidirectional cameras as backing-up aid. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, page 1–5, 2007.

[FRBG05] Uwe Franke, Clemens Rabe, Hernán Badino, and Stefan Gehrig. 6d-vision: Fusion of stereo and motion for robust environment perception. *Pattern Recognition*, page 216–223, 2005.

[GOY03] GOYO OPTICAL INC. GM23514MCN data sheet, June 2003.

[HEL12] HELLA. Driver assistance systems radar 24Ghz, 2012.

[Hel13] Hella Aglaia. Cassandra vision - rapid prototyping for image processing with OpenCV, July 2013. http://www.cassandra-vision.com/ accessed: 2013-07-07.

[HL01] David L. Hall and James Llinas. *Handbook of Multisensor Data Fusion*. CRC Press, June 2001. Published: Hardcover.

[Its13a] Itseez. Camera calibration with OpenCV — OpenCV 2.4.6.0 documentation, September 2013. http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html accessed: 2013-09-07.

[Its13b] Itseez. OpenCV, July 2013. http://opencv.org/ accessed: 2013-07-07.

[Kan81] Bruce D. Lucas Takeo Kanade. An iterative image registration technique with an application to stereo vision. 81:674–679, 1981.

[Kau12] Eberhard Kaus. Grundlagen autonomer Fahrzeuge - Fahrzeugsteuerung – putting it all together, 2012.

[KB61] Rudolph E. Kalman and Richard S. Bucy. New results in linear filtering and prediction theory. *Journal of Basic Engineering*, 83(3):95–108, 1961.

[LM09] Matthew J. Leotta and Joseph L. Mundy. Predicting high resolution image edges with a generic, adaptive, 3-d vehicle model. In *Computer Vision and Pattern Recognition, 2009. CVPR 2009. IEEE Conference on*, page 1311–1318, 2009.

[Low91] David G. Lowe. Fitting parameterized three-dimensional models to images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 13(5):441–450, 1991.

[LTH$^+$05] Jianguang Lou, Tieniu Tan, Weiming Hu, Hao Yang, and S.J. Maybank. 3-d model-based vehicle tracking. *IEEE Transactions on Image Processing*, 14(10):1561–1569, October 2005.

[MHG$^+$06] Marc-Michael Meinecke, Raimond Holze, Mark Gonter, Thomas Wohllebe, Ralph Mende, and Rajko Petelka. Side-pre-crash sending system for automatic vehicle height level adaptation. In *Proc. 3rd Int. Workshop Intell. Transp., Hamburg, Germany*, 2006.

[Num13] Numpy developers. NumPy, July 2013. http://www.numpy.org/ accessed: 2013-07-25.

[PVB$^+$09] Sylvia Pietzsch, Trung Dung Vu, Julien Burlet, Olivier Aycard, Thomas Hackbarth, Nils Appenrodt, Jürgen Dickmann, and Bernd Radig. Results of a precrash application based on laser scanner and short-range radars. *Intelligent Transportation Systems, IEEE Transactions on*, 10(4):584–593, 2009.

[Pyt13] Python Software Foundation. Python programming language, July 2013. `http://python.org/` accessed: 2013-07-25.

[RSS08] Martin Rufli, Davide Scaramuzza, and Roland Siegwart. Automatic detection of checkerboards on blurred and distorted images. In *Intelligent Robots and Systems, 2008. IROS 2008. IEEE/RSJ International Conference on*, page 3121–3126, 2008.

[SBM06] Zehang Sun, George Bebis, and Ronald Miller. Monocular precrash vehicle detection: features and classifiers. *Image Processing, IEEE Transactions on*, 15(7):2019–2034, 2006.

[Sca07] Davide Scaramuzza. *Omnidirectional vision: from calibration to robot motion estimation.* PhD thesis, Citeseer, 2007.

[Sca13] Davide Scaramuzza. OCamCalib: omnidirectional camera calibration toolbox for matlab - davide scaramuzza, September 2013. `https://sites.google.com/site/scarabotix/ocamcalib-toolbox` accessed: 2013-09-07.

[SHB10] Dieter Schramm, Manfred Hiller, and Roberto Bardini. *Modellbildung und Simulation der Dynamik von Kraftfahrzeugen.* Springer, Berlin; New York, 2010.

[Sla99] Gregory G. Slabaugh. Computing euler angles from a rotation matrix. 1999.

[SMS06a] Davide Scaramuzza, Agostino Martinelli, and Roland Siegwart. A flexible technique for accurate omnidirectional camera calibration and structure from motion. In *Computer Vision Systems, 2006 ICVS'06. IEEE International Conference on*, page 45–45, 2006.

[SMS06b] Davide Scaramuzza, Agostino Martinelli, and Roland Siegwart. A toolbox for easily calibrating omnidirectional cameras. In *Intelligent Robots and Systems, 2006 IEEE/RSJ International Conference on*, page 5695–5701, 2006.

[ST94] Jianbo Shi and Carlo Tomasi. Good features to track. In *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*, page 593–600, 1994.

[TA77] AN Tikhonov and V Ya Arsenin. Solutions of ill-posed problems. *WH Winston, Washington, DC*, 330, 1977.

[Thr13] Sebastian Thrun. Artificial intelligence: How to build a robot - udacity, September 2013. `https://www.udacity.com/course/cs373` accessed: 2013-09-07.

[TK91] Carlo Tomasi and Takeo Kanade. *Detection and tracking of point features*. School of Computer Science, Carnegie Mellon Univ., 1991.

[Tor13] Torus Knot Software. OGRE – open source 3D graphics engine, July 2013. `http://www.ogre3d.org/` accessed: 2013-07-18.

[TP91] Matthew Turk and Alex Pentland. Eigenfaces for recognition. *Journal of cognitive neuroscience*, 3(1):71–86, 1991.

[Uni13] Unibrain. Unibrain.com - global leader in FireWire development., September 2013. `http://www.unibrain.com/` accessed: 2013-09-08.

[VBA08] Trung-Dung Vu, Julien Burlet, and Olivier Aycard. Grid-based localization and online mapping with moving objects detection and tracking: new results. In *Intelligent Vehicles Symposium, 2008 IEEE*, page 684–689, 2008.

[wxP13] wxPROs. wxPython, July 2013. `http://www.wxpython.org/` accessed: 2013-07-25.

[Zha00] Zhengyou Zhang. A flexible new technique for camera calibration. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 22(11):1330–1334, 2000.

[ZXY12] Gangqiang Zhao, Xuhong Xiao, and Junsong Yuan. Fusion of velodyne and camera data for scene parsing. In *Information Fusion (FUSION), 2012 15th International Conference on*, page 1172–1179, 2012.

Appendix A

Sensor Mounting Positions

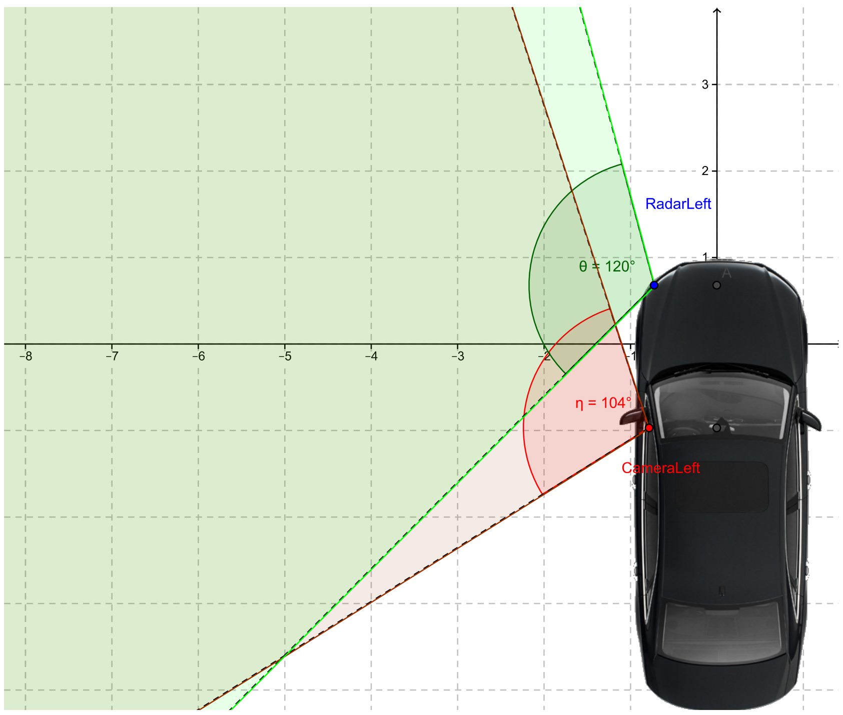


Figure A.1: Overlap of camera (mounted on side mirror) and radar (mounted on front bumper)

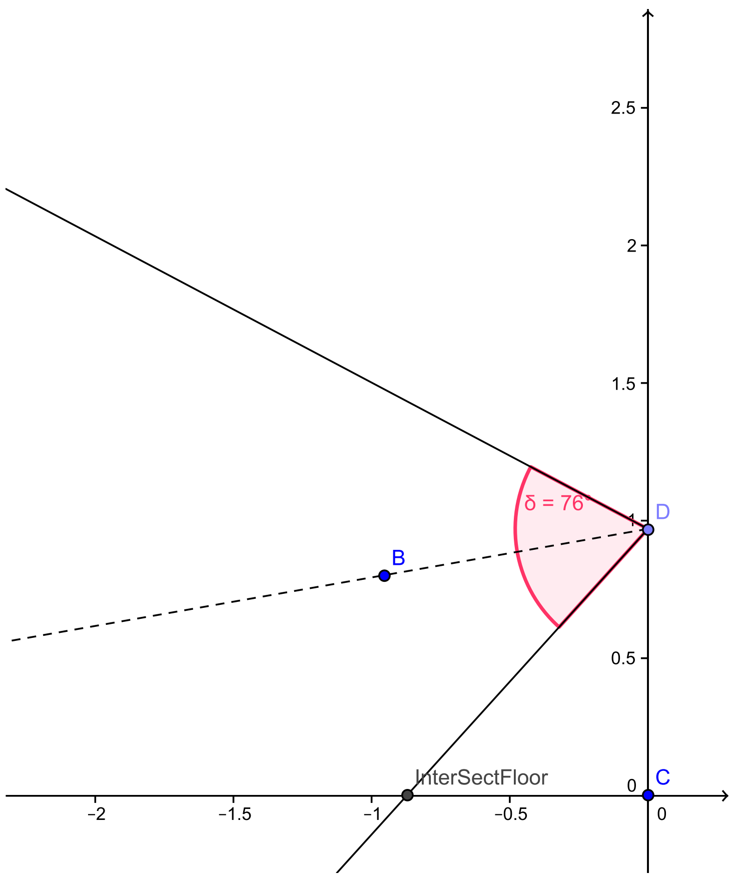


Figure A.2: Back view of the camera position

Appendix B

Calibration Errors of OpenCV and Scaramuzza Camera Models

	Scaramuzza			**OpenCV**		
File	$\overline{e}$	σ^2	E	$\overline{e}$	σ^2	E
002.bmp	0.370435	0.0231854	12.8326	0.275795	0.0217404	7.82427
003.bmp	0.185568	0.0112573	3.65543	0.247968	0.0184889	6.39818
004.bmp	0.365633	0.0306408	13.1463	0.221599	0.0118968	4.88025
005.bmp	0.165362	0.00743389	2.78228	0.186114	0.0133489	3.83899
006.bmp	0.394953	0.0232965	14.3427	0.333362	0.02819	11.1456
007.bmp	0.261203	0.0151738	6.67208	0.227663	0.0316865	5.34436
008.bmp	0.279649	0.0186588	7.74898	0.281389	0.110866	8.86929
009.bmp	0.300676	0.0152974	8.45626	0.250963	0.0161416	6.32993
010.bmp	0.259342	0.0126064	6.38917	0.257811	0.0156026	6.56552
011.bmp	0.483464	0.0522302	22.8774	0.32991	0.0283378	10.9743
012.bmp	0.497691	0.0483012	23.6798	0.188713	0.0170201	4.21061
013.bmp	0.301386	0.0146236	8.43658	0.215351	0.0101641	4.52322
014.bmp	0.387718	0.0244953	13.9856	0.309064	0.0227157	9.45891
015.bmp	0.300167	0.0185536	8.6923	0.162572	0.00817829	2.76863
016.bmp	0.487956	0.0624943	24.0476	0.321132	0.0270494	10.414
017.bmp	0.239249	0.0101859	5.39406	0.184098	0.00587861	3.18165
018.bmp	0.115092	0.00284178	1.28703	0.112031	0.00426226	1.34506
019.bmp	0.211512	0.00992366	4.37286	0.103195	0.0037201	1.14954
020.bmp	0.271937	0.0101612	6.72889	0.241402	0.0241196	6.59156
021.bmp	0.2419	0.0262558	6.78173	0.213922	0.0109082	4.53366
022.bmp	0.372876	0.0403377	14.3499	0.336342	0.0213889	10.7612

Table B.1: Calibration errors for Scaramuzza and OpenCV camera models calibrated for the left side camera

	Scaramuzza			OpenCV		
File	$\overline{e}$	σ^2	E	$\overline{e}$	σ^2	E
001.bmp	0.284246	0.0338398	9.17084	0.595858	0.146288	40.1068
002.bmp	0.271896	0.0175099	7.31501	0.280599	0.0324063	8.89137
004.bmp	0.486398	0.0662387	24.2258	0.516561	0.094324	28.8927
005.bmp	0.55324	0.0529232	28.7198	0.687035	0.109714	46.5386
006.bmp	0.186948	0.0133088	3.86066	0.157153	0.00907199	2.70152
007.bmp	0.286031	0.0215893	8.27223	0.158952	0.00617741	2.51546
008.bmp	0.307392	0.01705	8.92321	0.326994	0.026876	10.7041
009.bmp	0.350017	0.0249989	11.8009	0.300778	0.0358878	10.1084
010.bmp	0.309961	0.0353365	10.513	0.611178	0.117948	39.3189
011.bmp	0.210369	0.0113743	4.45034	0.257688	0.0243119	7.25722
012.bmp	0.418896	0.038353	17.1061	0.308436	0.0347724	10.3924
013.bmp	0.606872	0.0433953	32.9351	0.433804	0.0856371	21.9058
014.bmp	0.421563	0.028635	16.508	0.379008	0.050028	15.494
015.bmp	0.353267	0.0193904	11.5351	0.287304	0.0254661	8.64078
016.bmp	0.445758	0.0384478	18.9719	0.57627	0.115505	35.8074
017.bmp	0.600672	0.160608	41.7132	0.243548	0.107525	13.3472
018.bmp	0.398316	0.042942	16.1278	0.576793	0.128176	36.8693
019.bmp	0.324004	0.037779	11.4206	0.514831	0.0899744	28.402
020.bmp	0.349709	0.023557	11.6682	0.459687	0.0877708	23.9267
021.bmp	0.220668	0.00816057	4.5484	0.333378	0.0503491	12.9192
022.bmp	0.196176	0.00997511	3.70659	0.249798	0.0205774	6.73778

Table B.2: Calibration errors for Scaramuzza and OpenCV camera models calibrated for the right side camera

Appendix C

Cassandra Graphs

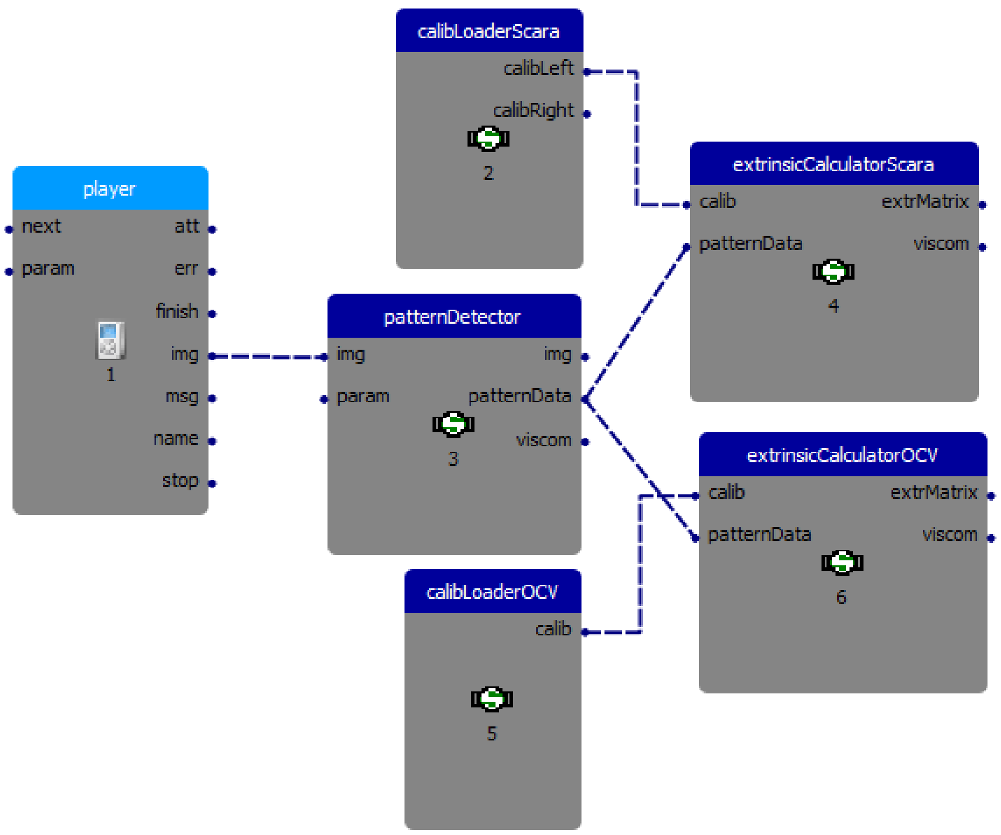


Figure C.1: Extrinsic calibration graph in Cassandra

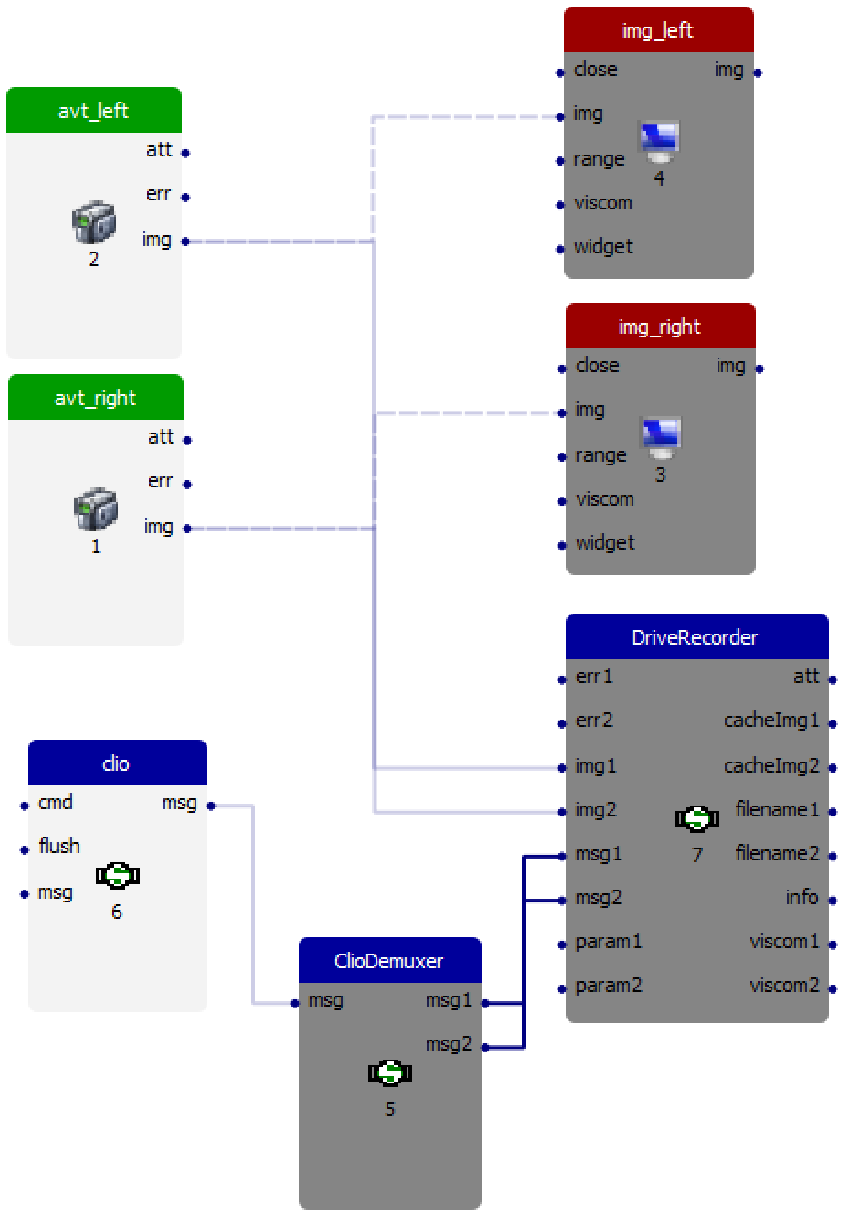


Figure C.2: DriveRecorder Graph for capturing CAN and Cameras in Cassandra

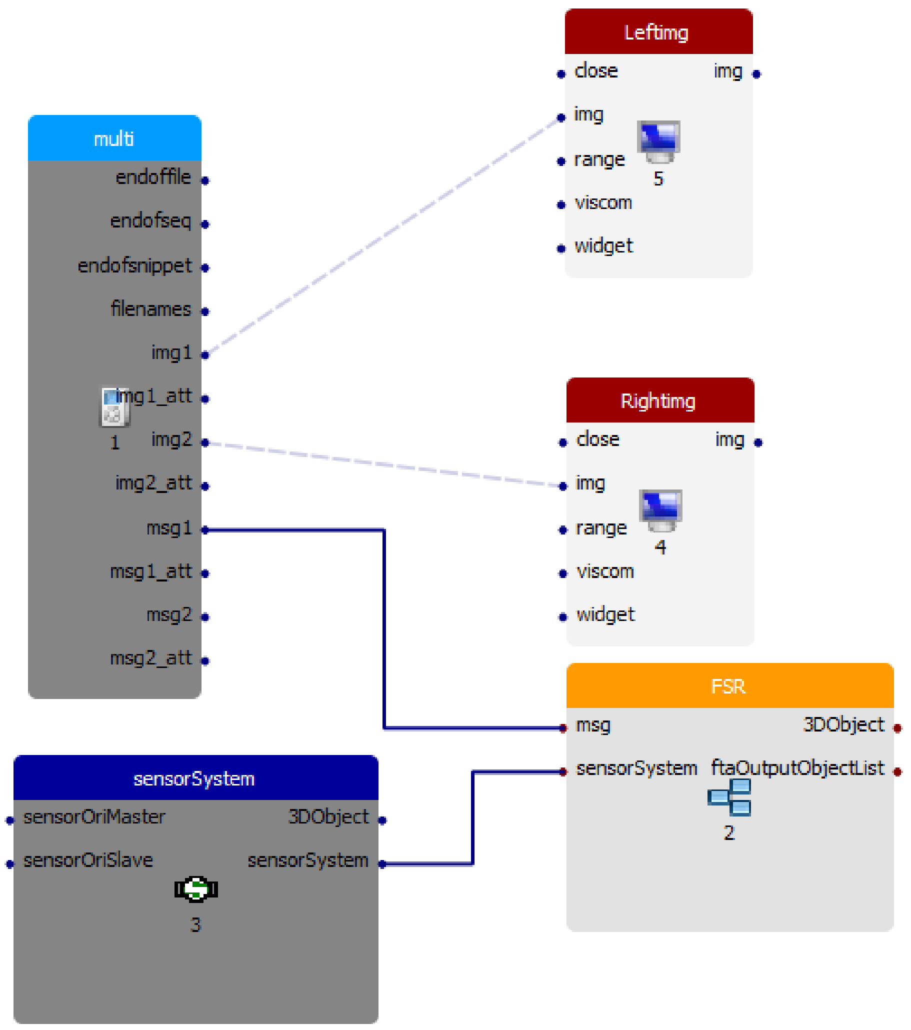


Figure C.3: Basic Graph for replaying recorded data

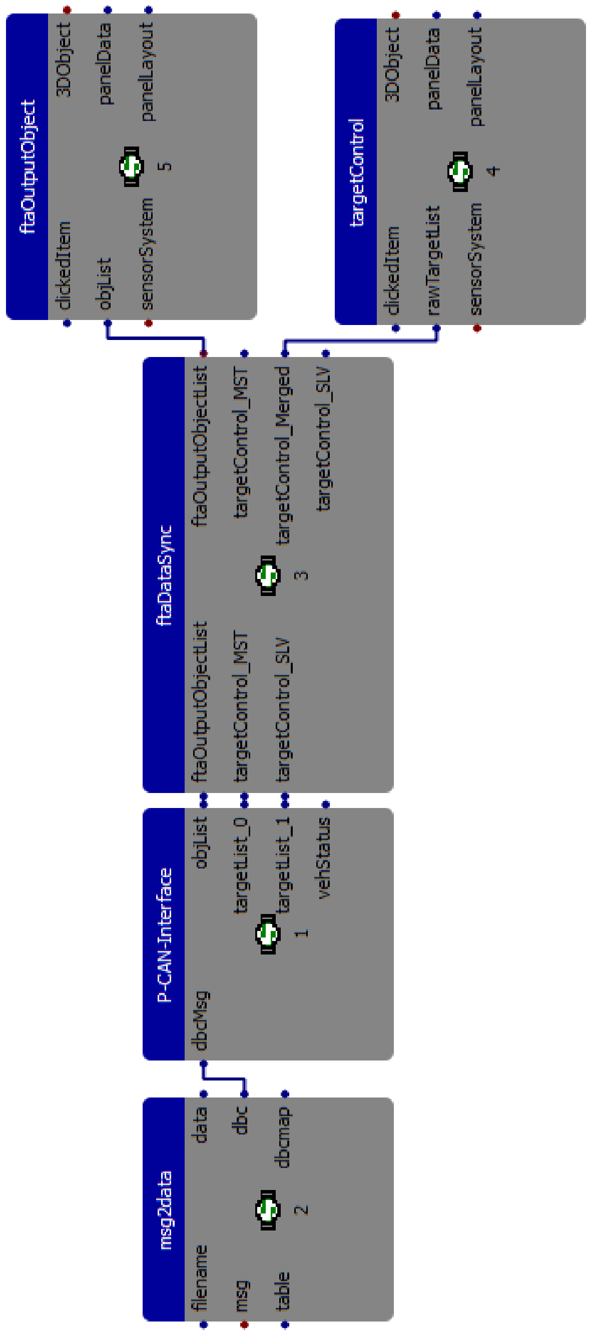


Figure C.4: FSR sub graph for decoding CAN messages

Appendix D

A lazy Approach for Extrinsic Wide Angle Camera Calibration

This is a thought experiment for the calibration of side cameras if a complete front camera calibration is available. The setup at Hella Aglaia for camera calibration supports front or back cameras with narrow opening angles. The following sections propose a method for the calibration of the side cameras used in the test bed. The proposal utilizes the already calibrated front camera. The standard setup for calibration is shown in Figure D.1.

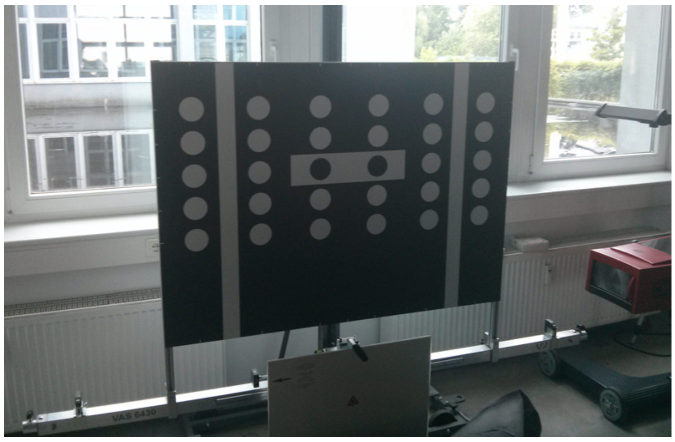


Figure D.1: Calibration stand

The calibration pattern consists of white dots on a black background. The pattern is not in the view of the side cameras. The goal is a repeatable setup that does not need a lot modification of the current setup. The front camera of the test setup car is already calibrated using Hella Aglaia's own setup. Hence, the world coordinates of the pattern points are known. It is assumed that the cameras have an intrinsic calibration based on the Scaramuzza model [SMS06a, Sca07]. With this model extrinsic parameters to a chessboard pattern positioned in the view area of the side cameras can be computed as shown in section 2.4.2.

The information missing are the relative position of the circle pattern P_F viewable from the front camera to the chessboard pattern P_S of the side cameras. This could either be constructed by precise measurement or estimated using a intrinsically calibrated camera seeing both patterns. This approach focuses on the latter. A list and description of all utilized reference frames can be found in Table D.1.

Reference Frame	Description
P_F	coordinate frame of the <u>f</u>ront chessboard pattern (corners, 3 dimensions with $z = 0$)
P_S	coordinate frame of the <u>s</u>ide chessboard pattern (corners, 3 dimensions with $z = 0$)
C_F, C_S	intrinsic 3–dimensional reference frames of the cameras (<u>F</u>ront and <u>S</u>ide)
E	car coordinate system aligned with the front axis. The x–axis points to the front and y to the left
K	Camera system of an arbitrary camera which is able to view both patterns P_F and P_S

Table D.1: Reference frames used for calibration

D.1 Finding a Link between two Pattern Reference Systems

As previously mentioned, we cannot use a single pattern to calibrate both cameras at the same time. Though, we need to find a transform that connects the two patterns.

For this, another internally calibrated camera K is needed which is able to see both patterns in a single image. Using a method to extract extrinsic parameters for each pattern from an image with both patterns visible, the transforms from each pattern coordinate system (P_S and P_F) to the camera coordinate system K can be estimated. The transforms will be noted as $\mathbf{T}_{K\leftarrow P_S}$ and $\mathbf{T}_{K\leftarrow P_F}$ which transform a point on the pattern $(x, y, 0)$ into the internal camera coordinate system K. The transform from one pattern coordinate system into the other then becomes a simple linear algebra exercise with the help of the coordinate system K.

To calculate the coordinates of pattern points from P_S in reference frame P_F, the points from pattern P_S are transformed into K. From K the inverse transform to P_F can be used. Hence, we get the following new transforms (eqs. (D.1) and (D.2))

$$\mathbf{T}_{P_F\leftarrow P_S} = \underbrace{\mathbf{T}_{P_F\leftarrow K}}_{\mathbf{T}^{-1}_{K\leftarrow P_F}} \mathbf{T}_{K\leftarrow P_S} \tag{D.1}$$

$$\mathbf{T}_{P_S\leftarrow P_F} = \underbrace{\mathbf{T}_{P_S\leftarrow K}}_{\mathbf{T}^{-1}_{K\leftarrow P_S}} \mathbf{T}_{K\leftarrow P_F} \tag{D.2}$$

With our new transforms we can switch between the two reference frames of the patterns.

$$\mathbf{x}_{P_F} = \mathbf{T}_{P_F\leftarrow P_S}\mathbf{x}_{P_S} \tag{D.3}$$

$$\mathbf{x}_{P_S} = \mathbf{T}_{P_S\leftarrow P_F}\mathbf{x}_{P_F} \tag{D.4}$$

D.2 Putting it all together

Based on our previous calculations, we can estimate the positions of both patterns, even though only one pattern is visible at a time. We fix the car in the calibration stand and calibrate the front camera with the help of the front pattern and the usual setup of the calibration stand. In the next step, we estimate the transforms between the two camera coordinate systems of the front and side cameras. At this point, in each camera only one of the patterns is visible. Pattern P_S is visible in the side camera C_S while pattern P_F is only visible in the front camera.

With both cameras we estimate the extrinsic parameters of the patterns and obtain the transforms $\mathbf{T}_{C_S\leftarrow P_S}$ and $\mathbf{T}_{C_F\leftarrow P_F}$, i.e. the transforms to transform points from the pattern coordinate system into the camera coordinate system. In the previous section we estimated the transforms $\mathbf{T}_{P_F \leftarrow P_S}$ and $\mathbf{T}_{P_S \leftarrow P_F}$, which we will now use as a bridge between the two camera coordinate frames in eqs. (D.5) and (D.6).

$$\mathbf{T}_{C_S \leftarrow C_F} = \mathbf{T}_{C_S \leftarrow P_S} \mathbf{T}_{P_S \leftarrow P_F} \underbrace{\mathbf{T}_{P_F \leftarrow C_F}}_{\mathbf{T}^{-1}_{C_F \leftarrow P_F}} \tag{D.5}$$

$$\mathbf{T}_{C_F \leftarrow C_S} = \mathbf{T}_{C_F \leftarrow P_F} \mathbf{T}_{P_F \leftarrow P_S} \underbrace{\mathbf{T}_{P_S \leftarrow C_S}}_{\mathbf{T}^{-1}_{C_S \leftarrow P_S}} \tag{D.6}$$

As the front camera calibration setup already provides us with a transform that allows us to convert between the front camera coordinate system and the car coordinate system E, we can derive the side camera transforms into the car coordinate system eq. (D.7) and vice versa eq. (D.8).

$$\mathbf{T}_{E \leftarrow C_S} = \mathbf{T}_{E \leftarrow C_F} \mathbf{T}_{C_F \leftarrow C_S} \tag{D.7}$$

$$\mathbf{T}_{C_S \leftarrow E} = \mathbf{T}_{C_S \leftarrow C_F} \mathbf{T}_{C_F \leftarrow E} \tag{D.8}$$