LEHRSTUHL FÜR REALZEIT-COMPUTERSYSTEME
TECHNISCHE UNIVERSITÄT MÜNCHEN
UNIV.-PROF. DR.-ING. G. FÄRBER

A Component-Based Image Processing Framework for Automotive Vision Applications

Simon Alois Hawe

Diplomarbeit

A Component-Based Image Processing Framework for Automotive Vision Applications

Diplomarbeit

Supervised by the Institute for Real-Time Computer Systems
Technische Universität München
Prof. Dr.-Ing. Georg Färber

Executed at Robotic and Automation Lab
Centre of Intelligent Information Processing Systems
University of Western Australia
Perth

Advisor: Assoc. Prof. Dr. rer. nat. habil. Thomas Bräunl
Dipl. Ing. Stephan Neumaier

Author: Simon Alois Hawe
Edelweißstr. 9
82377 Penzberg

Submitted January 6, 2008

Acknowledgements

Perth, January 6, 2008

Contents

List of Figures

List of Tables

List of Symbols

RCS	Lehrstuhl für Realzeit-Computersysteme
UWA	University of Western Australia
WHO	World Health Organization
SDK	Software Development Kit
IDE	Integrated Development Environment
GUI	Graphical User Interface
FLTK	Fast Light Toolkit
XML	Extensible Markup Language
HTML	Hypertext Markup Language
UML	Unified Modeling Language
BGL	Boost Graph Library
PCA	Principal Component Analysis
NN	Neural Network
SVM	Support Vector Machine
LP	Line Pixel
SAD	Sum of Absolute Differences
SSD	Sum of Squared Differences
NSC	Normalized Sample Correlation Coefficient
SVD	Singular Value Decomposition
DLL	Dynamic Link Library
$\mathbf{A}$	bold font and captial letter $\Rightarrow$ matrix
$\mathbf{a}$	bold font and small letter $\Rightarrow$ vector
$diag(x_1, \ldots, x_n)$	diagonal matrix of dimension $n \times n$ with all elements equal to zero, except the one in the trace $x_1, \ldots, x_n$
$\mathbf{A}^{-T}$	$(\mathbf{A}^{-1})^T = (\mathbf{A}^T)^{-1}$
$\mathbf{A}_{n \times m}$	matrix with n rows and m columns
GPU	Graphics Processing Unit

Abstract

Due to a continuously growing amount of traffic, and with it an increasing risk of car accidents, the necessity and demand for assisting a car driver is constantly rising. A possible way of providing the driver with additional information like warnings, is to equip cars with cameras and extract information from the obtained images via image processing procedures. A need for automotive vision frameworks to accelerate and simplify the development of image processing algorithms has been widely expressed. Using such a software can tremendously reduce time to market and thus saves costs. An extensible, modular and flexible open source component-based automotive vision framework called ImprovCV, that allows rapid and interactive development of image processing algorithms is presented in this thesis.

As car accident statistics attest, the main risks a driver is facing are from other vehicles. Consequently, developing on-board automotive driver assistance systems aiming to alert a driver about driving environments and possible collision with other cars is a crucial task. Knowing the own and other cars' position is essential for warning the driver about lane departure or detected obstacles in the lane the driver is traveling in. Here, an approach for detecting other vehicles using optical flow and shape information, which once detected are tracked in consecutive images via a correlation method is presented. Furthermore a lane detection system is introduced which uses the Probabilistic Hough Transform to detect lines, and a k-means cluster to combine these lines to lane-markings presenting the lanes. Finally a complete stereo approach from calculating the Fundamental Matrix, over rectifying the camera images to correspondence search with the goal of gaining depth information for distance estimation to other cars is shown. To rate the algorithm's performance, an evaluation of accuracy and execution time for each of them is given.

Zusammenfassung

Aufgrund einer beständig anwachsenden Verkehrsdichte und damit eines größeren Unfallrisikos, steigt die Notwendigkeit und Nachfrage nach Unterstützung eines Autofahrers konstant an. Eine Möglichkeit, den Fahrer mit zusätzlichen Informationen zu versorgen ist, das Auto mit Kameras auszustatten und mittels Bildverarbeitung Informationen aus den Kamerabildern zu extrahieren. Die Notwendigkeit eines “Vision Frameworks” zur Beschleunigung und Vereinfachung der Entwicklung von Bildverarbeitungsalgorithmen im Automobilbereich wurde vielfach geäußert. Die Nutzung einer solchen Software reduziert die Produkteinführungszeit ungemein und spart dadurch Kosten. In dieser Diplomarbeit wird ein erweiterbares, modulares und flexibles “open source” komponenten-basiertes Automotive Vision Framework, genannte ImprovCV präsentiert, das schnelle und interaktive Entwicklung von Bildverarbeitungsalgorithmen ermöglicht.

Wie Unfallstatistiken bestätigen, gehen die größten Risiken für einen Fahren von anderen Fahrzeugen aus. Aus diesem Grund ist das Entwickeln von “on-board” Fahrassistenzsystem mit dem Ziel, den Fahrer über seine Fahrumgebung und eventuell drohende Kollisionen mit anderen Fahrzeugen zu warnen, eine entscheidende Aufgabenstellung. Die Position des eigenen und der anderen Fahrzeuge auf der Straße zu kennen ist entscheidend, um den Fahrer vor Abkommen von der Spur, oder vor erkannten Hindernissen in der befahrenen Spur zu warnen. Hierfür wird ein Ansatz zur Detektierung von anderen Fahrzeugen in der Fahrspur vorgestellt, der den optischen Fluss und Information über Form nutzt, welche nach erfolgreicher Detektierung mittels einer Korrelations-Methode jeweils in aufeinander folgenden Bildern verfolgt werden. Außerdem wird ein Liniendetektionssystem eingeführt, das die probabilistische Hough-Transformation benutzt, um Linien zu detektieren, sowie ein k-means cluster, der diese Linien zu Farhbahnmarkierungen kombiniert, die die Fahrspur begrenzen. Zuletzt wird ein kompletter Stereoansatz aufgezeigt, von der Errechnung der fundamentalen Matrix, über Rektifizierung der Kamerabilder, bis hin zur Suche von Korrespondenzen, dessen Ziel die Gewinnung von Tiefeninformation zur Abschätzung von Entfernungen zu anderen Fahrzeugen ist. Um die Leistung des Algorithmus zu bewerten, wird jeweils eine Auswertung der Präzision und Ausführungszeit angegeben.

1 Introduction

This introductory chapter shows the underlying motivation of the thesis' topic, explains its objectives, and finally gives a brief overview of the thesis' outline.

1.1 Motivation

Worldwide, an estimated 1.2 million people are killed in road crashes each year and as many as 50 million are injured. That is one person dying every thirty seconds and one injured person nearly every half a second. Projections indicate that these figures will increase by about 65% over the next 20 years, unless there is new commitment to prevention. These shocking numbers have been presented by the World Health Organization (WHO) in their first *World report on road traffic injury prevention* [1].

Besides the fact of serious injuries, the costs arising due to car accidents consisting of hospital bills, damaged property and so forth are tremendously high and are expected to add up to $1\% - 3\%$ of the world's gross domestic product [2].

90% of all car accidents could be avoided if the car driver would react only one second earlier. An earlier reaction could be achieved by a system assisting the driver, which monitors the scene the driver sees, detects possible threats and either produces a warning message or as a last step intervenes actively.

With the aim of reducing injury and accident severity, pre-crash sensing has become an area of active research among automotive manufacturers, suppliers and universities. Car accident statistics attest that the main risks a driver is facing are from other vehicles. Consequently, developing on-board automotive driver assistance systems aiming to alert a driver about driving environments and possible collisions with other cars has attracted a lot of attention. In these systems, robust and reliable vehicle detection is the first step and can pave the way for vehicle recognition, vehicle tracking, and collision avoidance. Lane detection procedures can provide both estimations of the position and orientation of vehicles within the lane and a reference system for locating other vehicles or obstacles in the path of the one viewing the scene. The highly common problem of microsleep and resulting accidents could be reduced by knowing the lane and warning the driver when he is about to depart from it. Therefore, the detection of lanes is another reasonable part of a driver assistance system.

The usage of active sensors like laser, lidar or radar is currently the most common approach for vehicle detection. The sensors are called active because they actively detect the distance of an object by emitting a signal and measuring the travel time of this signal reflected by this object. Their main advantage is that they can measure certain quantities like distances directly and easily without requiring so many computational resources. Prototype vehicles employing active sensors have shown promising results. In 2007 for example, the autonomous vehicle "Junior" seen in 1.1 from the Stanford University crossed the Golden Gate Bridge in San Francisco during rush hour by only using the information provided by a 3D laser-scanner.

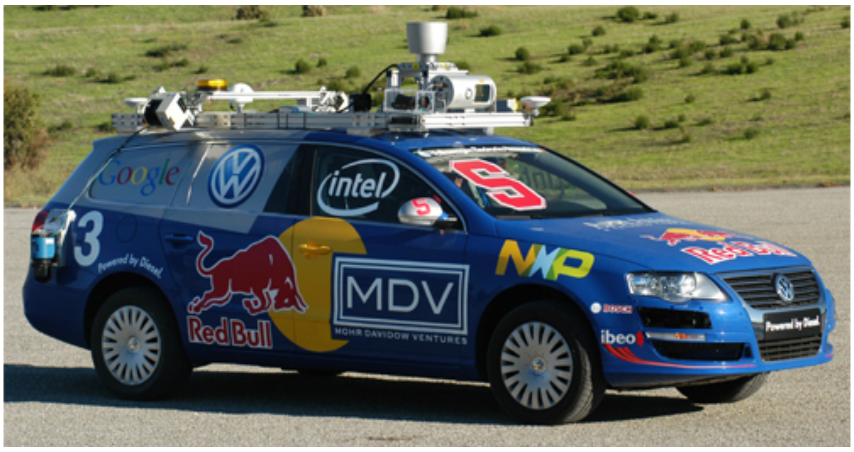


Figure 1.1: Junior, autonomous vehicle from Stanford University [3]

However, active sensors have several drawbacks, such as low spatial resolution, slow scanning speed or for sensors like a 3D-laser scanner very high costs. Moreover, interference among sensors of the same type applied to a large number of vehicles moving simultaneously in the same direction can produce big problems.

More powerful than active sensors are optical sensors like cameras, which are referred to as passive sensors, as no other sensor provides comparably rich information about the car's environment [4]. The low costs of cameras because of mass production combined with increasing image resolution and the constant rise of available processing power are another reason for using these sensors for vehicle detection, and provide the basis for growing vehicle's intelligence. By equipping vehicles with multiple cameras, a complete 360° view of the scene can be achieved. Using two cameras looking to the front and working in stereo makes it possible to gather 3D information of the current scene and can provide distance information of tracked vehicles. Furthermore, other applications besides vehicle detection, which are barely achievable with active sensors, can be realized, like the already mentioned detection of lane-markings, road sign recognition or pedestrian identification, see figure 1.2. Nevertheless, image processing is a very challenging task, as it underlies many variabilities like change of illumination, different shapes and colors of objects and weather conditions.

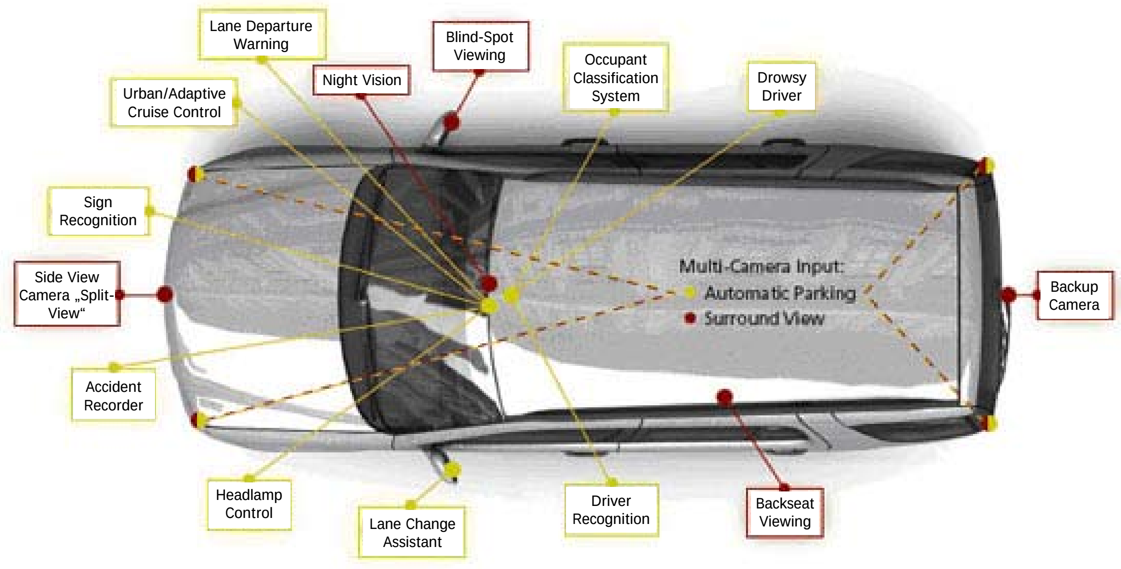


Figure 1.2: Different automotive computer vision applications [5]

To simplify and accelerate the development of image processing algorithms in any application of image processing, an open source framework providing standard operations like edge detection or noise filtering is a desirable tool. Faster development of new vision procedures means shorter time to market and cost reduction. Furthermore, giving developers the possibility of including their own algorithms into the framework would expand its power even more and eventually result in a continuous growth of available image processing algorithms, which can be combined again and create new ones.

1.2 Objectives

Two major goals are tried to be achieved in this thesis. The first one is the development of an open source image processing framework called ImprovCV. This framework is easy to use, flexible, modular and extensible. The goal of this framework is to give the user the feeling of "what you see is what you get", as several image processing filters can be combined and the resulting output is displayed on the screen. Both videos stored on the harddisk and live videostreams obtained from one or several cameras can be processed. The underlying image processing library is OpenCV [6] and all image processing operations' implementations contained in the framework are based on it. The user gets the opportunity to extend this framework with new image processing filters, as a Software Development Kit (SDK) is provided.

The second major goal of the thesis is the development and implementation of computer vision algorithms, that could be used for driver assistance. Three types of algorithms haven been implemented and run on the ImprovCV framework. The first algorithm implemented is a lane detection system based on a Hough Transform. The second one is the detection and tracking of an arbitrary number of vehicles without any initialization by a human operator, using optical flow and shape information. Finally, a stereo vision procedure with the important steps of obtaining a Fundamental Matrix, rectifying images and eventually finding stereo correspondences is implemented to gain depth information and a distances estimation.

1.3 Thesis Outline

The following chapter (2) will give an overview of related work concerning vision based drivers assistance, and provides a listing of state of the art image processing frameworks. An introduction to ImprovCV and its core features is presented in the third chapter (3). Furthermore, the underlying software design is outlined. After that, two chapters presenting the theory of the more sophisticated, important applied image processing operations (4) and the basics of stereo vision from epipolar geometry to correspondence search (5), follow. The sixth chapter (6) explains the developed computer vision algorithms for lane detection, vehicle detection and tracking and distance estimation of detected vehicles. The algorithms' results are evaluated concerning execution accuracy and time in the seventh chapter (7). Finally, a conclusion of the achieved goals and perspectives on future work are presented (8).

2 Related Work

As the availability of feasible technologies accumulated within the past 30 years of computer vision research and processor speed grew exponential, the way for running computation-intensive video processing algorithms even on a low-end PC in realtime has been paved and resulted in extensive research throughout universities and the automotive industry.

This chapter will present some selected approaches of computer vision in automotive applications, more precisely procedures for detecting cars in images and lane detection algorithms. Furthermore, some state of the art image processing tools both open source and commercial which simplify and accelerate the development of new vision approaches will be illustrated.

2.1 Literature Survey

Various approaches for recognizing and tracking vehicles from a moving camera have been proposed in respective technical literature. Here, three selected procedures using different methods like motion, symmetry or machine learning are presented. A review on some more vehicle detection approaches can be found in [7]. Furthermore, two different ways of detecting lanes are shown.

Vehicle Detection:
In [8] Frank Dellaert presented a car tracker, which is initialized by detecting bounding boxes in an image and selecting those bounding boxes as cars, for which an applied technique based on machine learning returns the highest fitness function. The model-based tracker [9], initialized with this method, relies on bayesian template-based image measurement techniques and an extended Kalman Filter.

Sun et. al formulated in [10] the detection of a vehicle as a two-class classification problem. The first problem is the extraction of features which they call the "multi-scale driven hypothesis generation step". They used the Principal Component Analysis (PCA), Wavelets, and Gabor filters to solve this task. The second step after having extracted the features is to evaluate and classify them correctly, called the "appearance-based hypothesis verification step", which was performed using Neural Networks (NN) and Support

Vector Machines (SVM).

A third approach for vehicle detection is using extracted motion information from images, arisen by moving cars. Choi suggested in [11] to use two different extraction methods for cars moving in the same direction as the one viewing the scene and cars traveling in the opposite direction, as they have distinct features. For cars in the coming traffic, an optical flow based detection is suggested as these cars represent distinct motion which is easy to extract. To detect vehicles traveling in the same direction, a Haar-like feature detector is suggested. Detected cars are tracked in consecutive frames by using a Kalman filter.

Lane Detection:
In [12] McCall and Trivedi used a steerable filter bank to detect lane-markings. This approach provides robustness to lighting changes and shadowing and performs well in picking out both circular reflector road-markings and painted line road-markings. Further post-processing based on the statistics of the road-marking candidates is applied to increase the robustness to occlusion by other vehicles and changing road conditions.

Instead of using a feature-based detection technique which localizes the lanes in the road images by combining low-level features, such as painted lines or lane edges etc., where lane segments are detected by traditional image segmentation, a model-based approach can be applied. Assuming the shape of a lane can be presented by either a parametrically straight line or a parametrically parabolic curve, the lane detection is done by calculating those model parameters.

In [13] a B-Snake-based model describing the perspective effect of parallel lines is supposed. Instead of detecting both boundaries, the mid-line is detected and the symmetry of boundaries is used. The vanishing point of lines obtained by applying a Canny Edge detector and a Hough Transform is used as the initial position for applying B-Snakes and creating a road model. By measuring the matching degree between the obtained model and the real edge map, the needed control points of the road model for lane detection and tracking are determined.

2.2 State of the Art Image Processing Tools

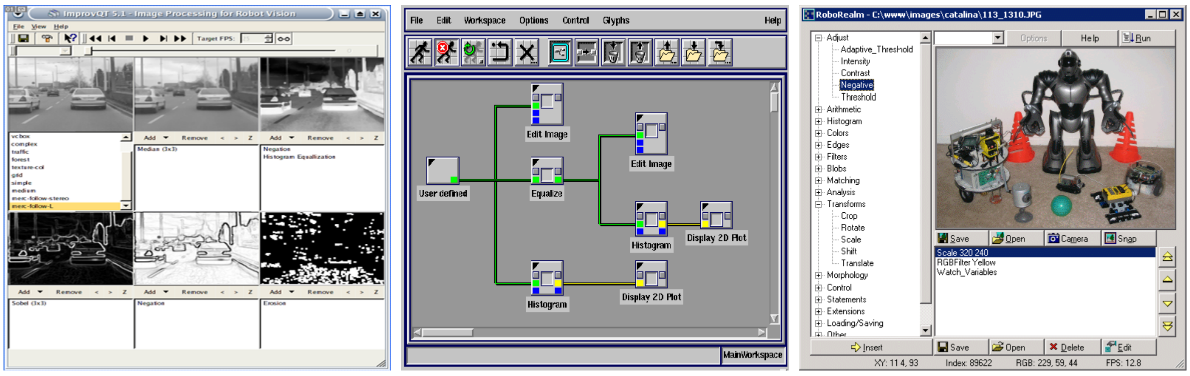
Over the past decade there has been an evolution of video and image processing frameworks used for several image processing applications like automotive vision processing. Here, some of them are briefly introduced but many more would be available, a fact that expresses the great demand of image processing frameworks. At first, currently available open source tools will be presented followed by commercial ones.

Open Source Frameworks:
The *Improv* [14] software is developed for real time robotics, uses and provides a stack-based interface allowing vision operators to be stacked on top of each other. Its Graphical User Interface (GUI) was developed with Trolltech® Qt® [15].

Another approach are dataflow visual language systems which allow users to graphically create a block diagram of their applications and interactively control input, output and system variables. An example for this architecture is *Khoros* [16], now known as VisiQuest [17], an integrated software development environment for information processing and visualization. It is particularly attractive for image processing because of its rich collection of tools for image and digital signal processing but sometimes the block diagram creations lead to cluttered and complex graphs.

Another free tool is called *RoboRealm* [18]. This is a powerful robotic vision software application for use in computer vision, image processing and robot vision tasks. It uses a simple point-and-click interface for image analysis and furthermore provides the possibility to create signals for controlling robots.

Each of the three frameworks presented above provides the user with a different interface to solve their image processing task. Figure 2.1 shows the three respective GUIs.


ImprovQT 5.1 - Image Processing for Robot Vision
User defined
Edit Image
Equalize
Histogram
Display 2D Plot
RoboRealm - C:\www\images\catalina\113_1310.JPG

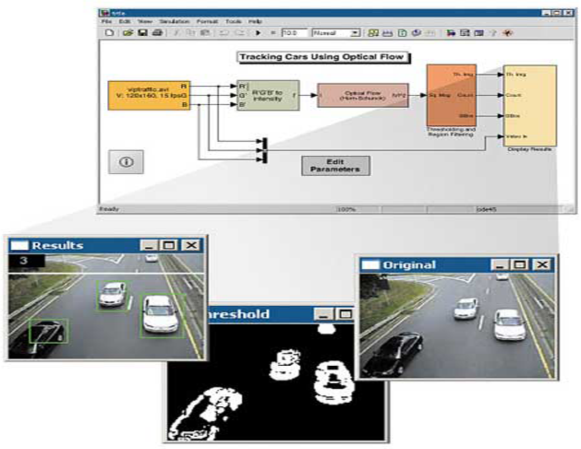
(a) Improv [14] (b) Khoros [16] (c) Roborealm [18]

Figure 2.1: Three different open source image processing frameworks

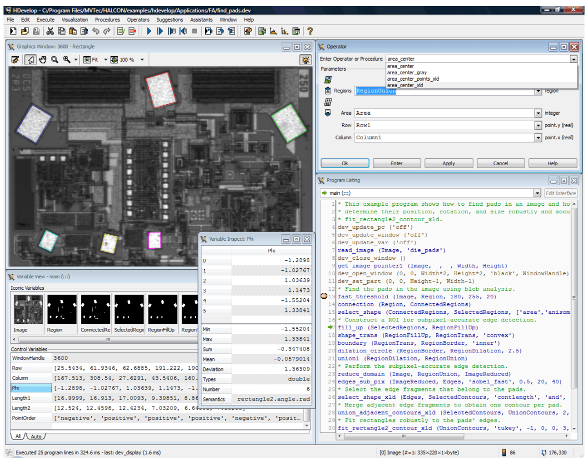
Commercial Frameworks:
A popular system for vision research is the combination of *MATLAB*® and *Simulink*® for image processing, using the Image Processing Toolbox (MATLAB) and the Video and Image Processing Blockset (Simulink) [19]. The Image Processing Toolbox provides a comprehensive set of standard algorithms and graphical tools for image processing, analysis, visualization, and algorithm development. The Video and Image Processing Blockset extends Simulink with a rich, customizable framework for the rapid design,

simulation, implementation, and verification of video and computer vision approaches. This environment provides an extensible programming system as well as a dataflow architecture. However, compiling and integrating the MATLAB and Simulink code for the target platform can be difficult.

An extremely powerful cross platform vision software used for applications in the area of Medical Image Analysis, Surveillance and Security, the Automotive Sector and Robotics is developed by MVTec and called *HALCON* [20]. The framework is shipped with more than 1300 different operators and provides its own Integrated Development Environment (IDE) for machine vision called HDevelop. Within this IDE, different HALCON operators can be combined by selecting them from a list and add them by click. The resulting chain of operators can be exported as C, C++, C# or Visual Basic source code, which can be integrated into an application. Figure 2.2 gives two examples of how these two frameworks look like.



(a) MATLAB and Simulink [14]



```
* This example program shows how to find pads in an image and ho
* determine their position, rotation, and size robustly and accu
* fit_rectangle2_contour_xld.
dev_update_pc ('off')
dev_update_window ('off')
dev_update_var ('off')
read_image (Image, 'die_pads')
dev_close_window ()
get_image_pointer1 (Image, _, _, Width, Height)
dev_open_window (0, 0, Width*2, Height*2, 'black', WindowHandle)
dev_set_part (0, 0, Height-1, Width-1)
* Find the pads in the image using blob analysis.
fast_threshold (Image, Region, 180, 255, 20)
connection (Region, ConnectedRegions)
select_shape (ConnectedRegions, SelectedRegions, ['area','anisom
* Construct a ROI for subpixel-accurate edge detection.
fill_up (SelectedRegions, RegionFillUp)
shape_trans (RegionFillUp, RegionTrans, 'convex')
boundary (RegionTrans, RegionBorder, 'inner')
dilation_circle (RegionBorder, RegionDilation, 2.5)
union1 (RegionDilation, RegionUnion)
* Perform the subpixel-accurate edge detection.
reduce_domain (Image, RegionUnion, ImageReduced)
edges_sub_pix (ImageReduced, Edges, 'sobel_fast', 0.5, 20, 40)
* Select the edge fragments that belong to the pads.
select_shape_xld (Edges, SelectedContours, 'contlength', 'and',
* Merge adjacent edge fragments to obtain one contour per pad.
union_adjacent_contours_xld (SelectedContours, UnionContours, 2,
* Fit rectangles robustly to the pads' edges.
fit_rectangle2_contour_xld (UnionContours, 'tukey', -1, 0, 0, 3,
```

(b) Halcon with HDevelop [20]

Figure 2.2: Two different commercial image processing frameworks

3 Image Processing Framework ImprovCV

Regarding the area of image processing, there is a great amount of standard operations like color conversion, edge detection, noise elimination and so on, that are very frequently used and therefore should not have to be implemented anew by everyone using them. A very powerful library that provides the user with these standard operations and a lot more, is the Intel® open source image processing library *OpenCV* [6]. However, it is not possible to quickly try a combination of filters and getting a first idea of how the output will look like, without writing a program that uses the library. Furthermore, once a program is written, adjusting the filter parameters is very time consuming as it either has to be started again if parameters can be read from a file, or the code has to be recompiled every time the parameters have been changed, to see their effect on the output. Therefore the extensible, flexible, and modular open source OpenCV-based image processing framework *ImprovCV* has been developed, with the goal of solving these problems and giving the user an impression of "look and feel". This is achieved by allowing the operator to apply an arbitrary combination of filters on a sequence of images by drag and drop and to adjust the filter parameters during runtime to gain immediate feedback.

This chapter will describe ImprovCV in detail by giving a general presentation of its functionality, pointing out the most important features and presenting the design of the software. First a brief overview and explanation of the used libraries is given.

3.1 Used Libraries

FLTK: FLTK (Fast Light Toolkit) [21] is an open source cross-platform C++ GUI toolkit for UNIX®/Linux® (X11), Microsoft® Windows®, and MacOS® X. In contrast to libraries like Trolltech® Qt® [15] and wxWidgets® [22], FLTK has the advantage of restricting itself to GUI functionality and using a more lightweight design, making the generated programs small and modular enough to be statically linked. On the other hand, the resulting disadvantage is that it offers fewer widgets by default than the libraries mentioned above. The version used for ImprovCV is version 1.1.7.

OpenCV: OpenCV is an open source computer vision library originally developed by Intel®. It is free for commercial and research use under a BSD license. The library is cross-platform, and runs on MacOS® X, Microsoft® Windows® and Linux®. It focuses mainly on real-time image processing, if it finds Intel's Integrated Performance Primitives (IPP) on the system, it will use these commercial and optimized routines to accelerate itself. It does not rely on external numerical libraries, although it can make use of some of them at runtime, if they are available. Shipped with more than 300 different functions in the field of linear algebra and image processing, OpenCV is a very powerful library and can be used in many different areas where image processing is employed like Motion Tracking, Object Identification, Face Recognition and Mobile Robotics. OpenCV was of key use in the vision system of Stanley [23], the autonomous vehicle of Stanford University which won entry to the 2005 DARPA Grand Challenge Race. Its big disadvantage is that many functions are only sparely documented or even not at all, which makes them sometimes hard to understand and use.

TinyXML: TinyXML [24] is a very small and simple open source XML parser for the C++ language. It can be easily integrated into programs to parse an XML document and build a Document Object Model (DOM) from it. The DOM can then can be read, modified and saved. It also allows the user to construct own XML documents with C++ objects and write these to the harddisk or another output stream. As the name implies, it is tiny and does not support Document Type Definition (DTD) or extensible Stylesheet Language (XSL) and in terms of encodings, it only handles files using UTF-8 or an unspecified form of ASCII not entirely dissimilar from Latin-1.

3.2 General Description

The ImprovCV software is an open source image processing framework based on the above mentioned libraries, developed for applications in the field of automotive vision, but can be used for any other image processing application as well. It presents to the user an easy to use graphical interface to construct and test image processing algorithms. Picture 3.1 shows the GUI of ImprovCV created with FLTK. On the top of the GUI, there is a menubar for executing different things like opening a video or quitting the program. Below the menubar, a number of control buttons can be seen. On the left side there are three different lists for selecting:

1. Different videos depending on how many are loaded
2. Groups of filters
3. The filters from the selected group.

The big beige area is the so called processing window. The black window on the upper right is a display for showing the output of a selected filter and is called preview display.

The empty grey area leaves space for control items to adjust the respective parameters, which are dynamically loaded depending on the selected filter. The thin white bar on the bottom displays warning messages to provide the user with information if he did something wrong, like connecting a filter that returns a multichannel image with a filter that needs a single channel image.

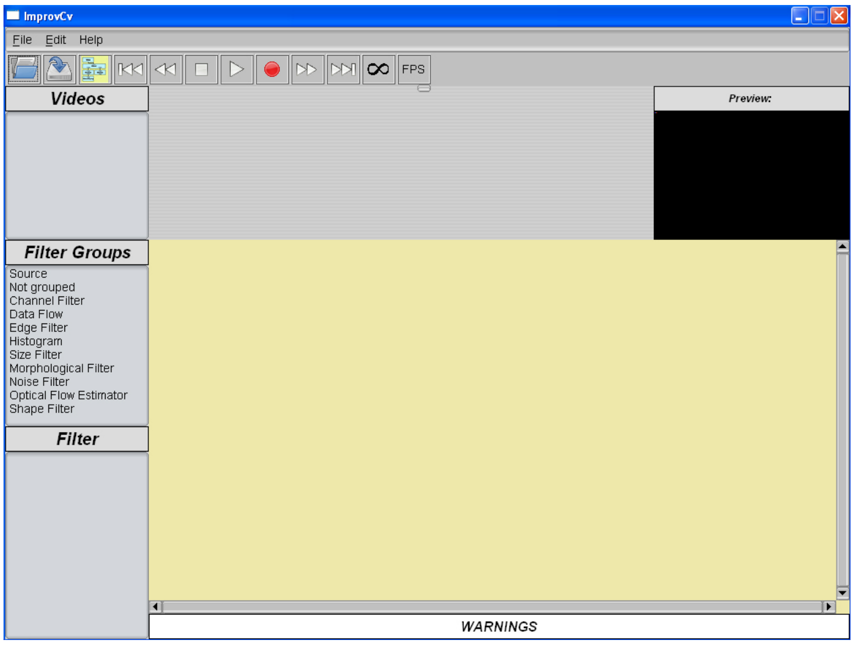


Figure 3.1: ImprovCV

The data to be processed can be both a video file and a live stream from a camera depending on the user's choice. The input can be changed at any time, to provide feedback of how an algorithm effects different image sequences. Furthermore, an arbitrary number of videos can be loaded and processed with any algorithm at the same time, whereby the user can directly compare results of different algorithms, or the same ones with different parameters. Depending on the number of loaded videos, the processing window is divided into several smaller processing windows, one for each video. The program has the ability to grab frames of multiple cameras for multiple vision. Two connected cameras can be synchronized by using a thread, meaning they both grab images at the same time to perform real and accurate stereo vision.

The software provides an amount of filters, implemented as a Dynamic Link Library (DLL), which are dynamically loaded during the start of the program. Thus the GUI is separated from the actual implementations of the image processing filters. This allows the user to write his own filters, which have to follow a certain convention, and import them into the program. An algorithm, consisting of a number of different filters, is created by simply selecting a filter group, choosing the wanted filter from the filter list, drag it over the processing window and drop it behind the filter which provides the needed input.

Removing a filter from its respective processing window is also possible. For each filter, a documentation in Hypertext Markup Language (HTML) format is given which provides information about the filter's in- and output, the adjustable parameters and a description of what the filter does.

Filters are visualized by blue boxes, showing their name in the middle, the time they needed for execution, the percentage of the whole algorithm execution time, their input channels with respective type on the top and their output channels with respective type on the bottom. In- and outputs of two consecutive filters are automatically connected after the filter has been dropped, and the connection is shown by a connecting line between the respective channels. If a filter is selected by clicking on it, the respective box changes the color to red, the control elements for adjusting the filter parameters will be loaded and the filter's output image will be shown in the preview display. Image 3.2 shows an example of how two connected filter boxes look like.

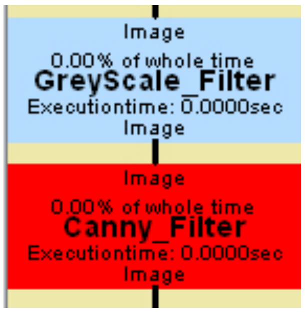


Figure 3.2: Two connected filters visualized in ImprovCV

It would be very unhandy and would make the software less useful if a developed algorithm that consists of a number of filters with certain adjusted parameters would have to be created again every time by selecting those filters, dropping them onto the processing window and adjusting the parameters. Therefore it is possible to store a chain of filters, together with all respective filter parameters, in an XML file which then can be reloaded at any time.

The underlying software design, which makes the described extensible, flexible and modular way of creating and combining filters possible will be explained in the next section.

3.3 Software Design

One paradigm providing modular, flexible and extensible software that has been thoroughly embraced in the business software world is *Component-Based Software*. Component-Based Software Engineering (CBSE) (also known as Component-Based Development

(CBD) or Software Componentry) is a branch of the software engineering discipline, with emphasis on decomposition of the engineered systems into functional or logical components with well-defined interfaces used for communication across the components. Components are considered to be of a higher level of abstraction than objects and as such they do not share state and communicate by exchanging messages carrying data. Cai et. al define in [25] three main features of a component, here named **F1**-**F3**:

- **F1**: independent, non-context-specific and replaceable part of a system that fulfills a clear function
- **F2**: works in the context of a welldefined architecture
- **F3**: communicates with other components by its interfaces

In the presented software system it should be possible to place and remove image processing filters solving different tasks from everywhere in a filter chain (**F1**) and these filters should communicate and exchange data with each other (**F3**) as one filter processes the results of an other, independent from their concrete functionality. Thus a filter perfectly fulfills the requirements of a component. Consequently the filters are interpreted as components and ImprovCV is engineered as a component-based software supplying the required architecture (**F2**) to embed components.

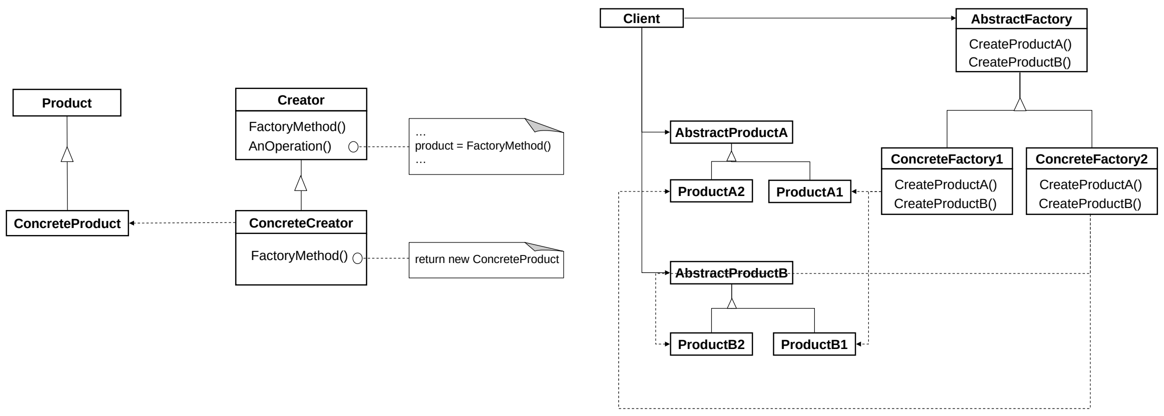
The following three key features are provided by ImprovCV

- A central repository to construct and manage filters
- The ability to plug in custom filters
- The ability to intelligently connect filters with a standard communications mechanism for data exchange

3.3.1 Abstract Pluggable Factory

The idea of handling a filter as a component implies that its creation has to be independent from its actual implementation, so the program using the filters can create each filter by calling the same method and does not have to be changed every time a new type of filter has been added or a filter has been changed. The *Factory Method Pattern* [26] is an object-oriented design pattern related to the group of creational pattern, which solves the problem of creating objects without specifying the exact class of the object to be created. It handles this problem by defining a separate method in a base class for creating instances of objects, whose derived subclasses can then overwrite the creation method to specify the concrete objet that will be created. When a new object has been added, not the real application using an instance of this object has to be changed to create one of the new objects, but only the factory has to be modified. All products that are created with the same factory method have to belong to one family.

An even more general type of creational pattern is the *Abstract Factory Pattern* as it deals with the creation of products belonging to multiple families without specifying their concrete classes. Many times, the Abstract Factory Pattern is combined with several Factory Methods, one for each product-family to be created [27]. Image 3.3 shows the respective UML-diagrams for the Factory Pattern Method and the Abstract Factory Pattern.



(a) Factory Method Pattern (b) Abstract Factory Pattern

Figure 3.3: UML diagram for Factory Method Pattern and Abstract Factory Pattern

The Abstract Factory Pattern is a very good way to create the components of the software described here, but still each time a new type of filter has been added the respective factory has to be modified as the factory itself has to know what it is creating. A *Pluggable Factory* as proposed in [28] solves this problem by allowing plug-ins to automatically extend the application's functionality without requiring any modification to the Abstract Factory code itself.

The implementation of a Pluggable Factory requires a central repository, called registry, where all "plugged-in" and available components are registered and can be looked up. Furthermore, a method is required to create the components. Thus each class that needs to be accessible via the factory has to have a method to add its information to the factory's registry and has to provide a method that allows to create a copy of itself, which is known as the *Prototype Design Pattern*. The automatic registration of a class is achieved by adding a static copy of each class to itself which is always initialized during the application's startup process, as the copy is static. Because a **class** is initialized, the constructor has to be called. By adding an instance of the newly created class to the factory's registry during the respective constructor call, the new class is registered. Once a component is registered, it can then be created by looking for the desired class type in the registry, construct it with the provided creation method, and return the resulting instance.

The following C++ source code shows a possible implementation body of the base class, which is referred to as the factory, allowing the creation of any subclass, and a subclass inheriting from it which is a plug-in example.

```
class Baseclass{
    private:
        static map<string, Baseclass* > registry;
    protected:
       void registerClass(string name){
            registry.insert(make_pair(name,this));
       }
       virtual Baseclass* create() = 0;

    public:
        static Baseclass* createAnyObject(string name) {
            //Find the object which uses this name in the registry
            map<string, Baseclass* >::iterator mapIter = registry.find(name);
            //call the create method, and return the created instance
            //if existing other wise return 0
            if(mapIter != registry.end())return registry[name]->create();
            else return NULL;
        }
};

class NewComponent : public Baseclass{
    public:
        NewComponent(string name){//doing what to do};

    private:
        //Constructor to register this class
        NewComponent(){registerClass("NewComponent");};
        //creation method
        Baseclass* create(){return new NewComponent("");};
        //static copy to call constructor during app. start
        static const NewComponent registerThis;
}

//these lines would be placed in the respective cpp source files
NewComponent NewComponent::registerThis;
map<string, Baseclass*> Baseclass::registry;
```

As the default constructor of any subclass inheriting from the base class is used to register the respective new subclass, and this has to happen only once, another constructor with any other body (in the example above it is NewComponent (string name)) has to be added to the subclass for the real construction of an object's instance that will be used.

With the Abstract Pluggable Factory design pattern, a lightweight system is given to construct, manage and extend the components of the framework. Still the standard communication interface is required for connecting components to exchange data. The next section will provide a solution for this task.

3.3.2 Communication Mechanism

The chosen communication mechanism for the interoperable components of ImprovCV is build on top of the Abstract Pluggable Factory described in the previous section, by providing an abstract uniform interface to the objects that enables the connections between components. The paradigm implemented to achieve this is a dataflow architecture. The communication's interface between the objects is represented as a directed graph. The open source Boost Graph Library (BGL) [29] is employed as it provides efficient and generic graph classes.

Fundamentally, a graph consists of a set of vertices and edges, where an edge is something that connects two vertices in the graph. In a directed graph, edges are ordered pairs, connecting a source vertex to a target vertex. This definition of a graph is vague in certain respects, it does not say what is represented by a vertex or edge. They could be cities with connecting roads, or web-pages with hyperlinks. These details are left out of the definition of a graph for an important reason: they are not a necessary part of the graph abstraction. By leaving out the details a theory can be constructed, which is reusable and can help solving lots of different kinds of problems. Thus, a directed graph perfectly suits a component-based software to provide the connection and communication interface between the components.

The presented framework has two different kinds of vertices to be connected: filters and variables. The connections between filter and variable are the edges of the presented graph system. A variable contains any kind of data like images, lines, points and so forth. A filter processes a given number of inputs obtained from the same number of variables and then can produce any number of outputs, which each of them will be connected to a variable again, see picture 3.4 (a). Thus, each object being part of the dataflow has to define a number of input- and output-channels to assure the communication.

The graph containing the filters and variables is maintained by the factory. Furthermore, the factory provides the management of the graph, meaning adding, removing, rejecting and following connections. Each filter itself has to store a list of the output formats it produces and the input formats it accepts. The name of an edge corresponds to the name of the input- and output-channel format it is connecting. An input-channel of any filter can only be connected with an output-channel of a variable if they are of the same data format. All variables with the same parent node are not necessarily connected to the same child node filter but can be connected to any number of filters, see figure 3.4 (b). Likewise, the variables connected to the same child node do not have to be connected to the same parent node, 3.4 (c). One variable can also be connected to multiple filters.

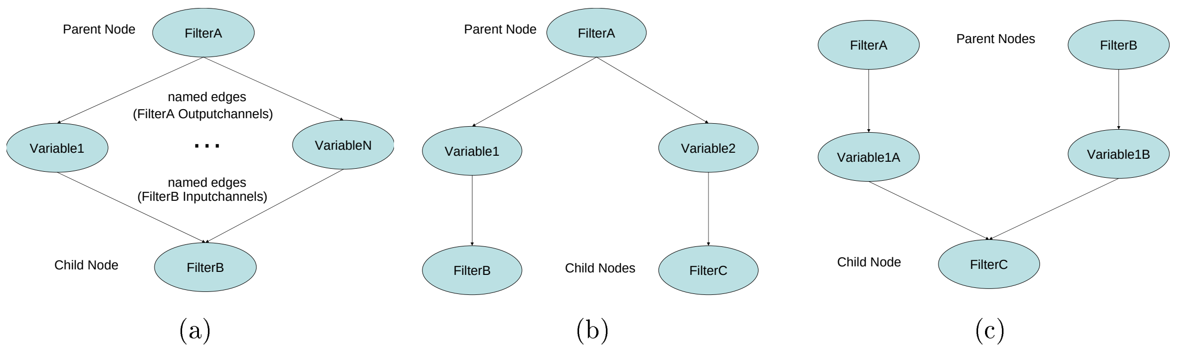


Figure 3.4: The framework's graph structure

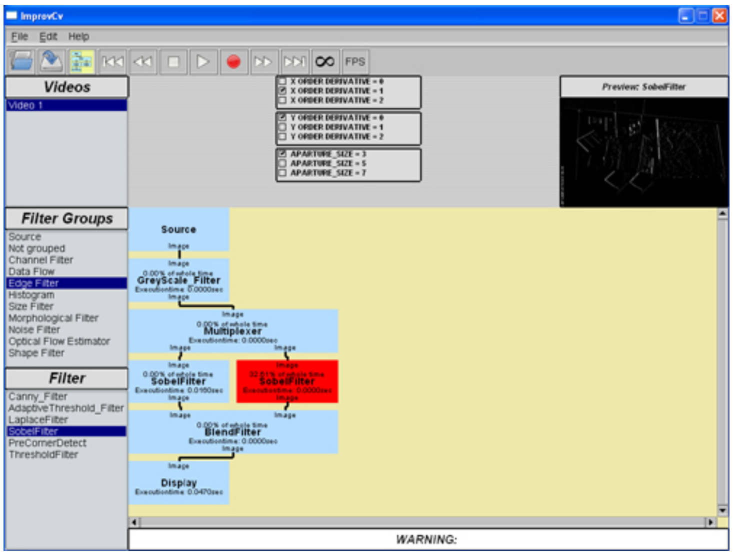
The factory class is also responsible for managing dataflow during execution. The graph is traversed in a modified breadth-first search such that each filter operates on its input data and passes the results to its respective variables, which in turn are read by the next filters, operated on and passed down the graph to the terminating node to perform an action such as issuing control commands or displaying results on the screen. After reaching the final node, the execution will start from the beginning.

The introduced system allows each image processing operation to be implemented in an independent separate filter that can be connected to any other filter that fits the respective outputs. Possibly required filter parameters can be either adjusted manually or generated by another filter.

3.3.3 Graph Visualization and Management

The previous two sections explained the method behind the creation and communication of the components of ImprovCV. As the user has to interact with the graph, the filters and respective connections of a currently existing graph have to be displayed in an appealing way. Picture 3.5 gives an example of how a graph is visualized in the presented framework.

Each graph created with ImprovCV has a source node providing the data to be processed and a terminating node. As ImprovCV is an image processing tool, the source node supplies the image to be processed, and the terminating node performs the displaying of the processing result to the screen. As mentioned before, a filter is displayed as a box in the processing window. By adding a new box representing a filter to the processing window, the respective filter is added to the graph. An instance of the filter is created by the factory with the knowledge of the name of the box, as this is the same name as the underlying implemented class. Each time a new filter has been added, connection lines between the connected in- and outputs are drawn and the positions of all boxes in the processing window have to be updated.



- ☐ X ORDER DERIVATIVE = 0
- ☑ X ORDER DERIVATIVE = 1
- ☐ X ORDER DERIVATIVE = 2

- ☑ Y ORDER DERIVATIVE = 0
- ☐ Y ORDER DERIVATIVE = 1
- ☐ Y ORDER DERIVATIVE = 2

- ☑ APARTURE_SIZE = 3
- ☐ APARTURE_SIZE = 5
- ☐ APARTURE_SIZE = 7

Figure 3.5: Complete graph in ImprovCV

The positioning is performed in a recursive way explained later in this section. Each box has to know its previous boxes (parent nodes) and directly following ones (child nodes). This is done by storing a pointer to each respective parent and child node. As mentioned above, a filter is added to the processing window via a drag and drop method. The new filter's parent nodes are determined by comparing the position at which the user has dropped the new box with all positions of already existing boxes lying above this position and selecting the closest ones to be the parent nodes. Once the parents are determined, their outputs have to be first disconnected from their former child nodes, then connected with the new filter's input and finally the new filter's outputs have to be connected to the former child nodes inputs. The newly added box is then moved to the position of its new child node with the largest distance to the nearest parent of the new box. The moving of the new box and all other ones lying below it, is done by their respective parents calling a recursive positioning function which uses the pointers to their child nodes, shown in the following pseudo code implementation:

```
void position(int x, int y){
    //to set the own position to x and y
    setMyPosition(x,y);
    //setting positions of all child nodes
    for(i = 1; till number of children; i++)
        children[i]->position(this->x(), this->y() + this->height() + gap);
}
```

As child nodes of one box are the parents of other boxes, they will call the positioning function for their children and so forth. Like this all nodes lying below the new one added are repositioned, the visualized graph is always correctly organized and allows an easy usage. The required respositioning of the graph after removing a box is performed in the same way.

An additional feature of ImprovCV, the preview window, which allows the user to see intermediate steps of filters which are not directly connected to the final node is realized by having an additional node in the graph, which is not displayed on the processing window. This node can be connected to any filter by simply clicking on the box representing the filter whose output is wanted to be inspected, and performs the displaying of its currently connected filter output in the preview window.

The presented coupling of an Abstract Pluggable Factory concept with a component-based dataflow graph provides an extremely flexible, modular, configurable and extensible software platform. The possibility of implementing a filter as a DLL separates the framework from the actual filters which makes the whole system even more flexible. Furthermore, it gives developers the opportunity of publishing their filters, without having to publish the respective code. Thereby the amount of filters can grow constantly, which makes the software very powerful and promising. The runtime overhead required for this system as opposed to a hard-wired system is a look up in a hash table for object creation and a few extra pointer dereferencing operations during object execution. Although there are many more operations required during the initial set up, this additional time is minimal in comparison to that of other commercial component-based architectures.

4 Feature Extraction and Clustering

Images show many different kinds of features like edges or shapes which all contain different information. By extracting these features with different methods, the information can be obtained and further processed. Often, feature extraction yields an amount of features with redundant information. These features can be combined in order to reduce the data to be further processed, with a clustering procedure mostly concerning a certain distance or similarity measurement. This chapter will present the Hough Transform to extract line features, optical flow for motion feature extraction resulted from movement between two consecutive images and a clustering method.

4.1 Hough Transform

Detecting certain forms and shapes like lines, circles or ellipses in an image is an important task in the field of Image Processing. In 1962 Paul Hough patented a powerful global method for detecting parameterized boundaries or curves which is named after him, the Hough Transform [30]. The Hough Transform became famous for detecting 2D-Shapes like lines and circles, but is not restricted neither to the dimensions nor to the shapes. In the following sections the Hough Transform for detecting lines will be described in detail. Both the standard method and a method called Probabilistic Hough Transform for reducing calculation time will be presented.

4.1.1 Standard Hough Transform

Considering the 2D-Cartesian space, a line is described by an infinite number of points $P(x, y)$ fulfilling equation 4.1

$$y = m * x + t \tag{4.1}$$

Instead of presenting the line mentioned above by an infinite number of points in the 2D-cartesian space, a 2D-parameter space represented by slope m and intercept t can be created by reversing equation 4.1

$$t = y - m * x \tag{4.2}$$

Now every line in the cartesian space is represented by a single point in the parameter space.

However, the representation by slope and intercept is not a favorable option because both parameters are unbounded. As a line becomes more and more vertical, the magnitudes of t and m grow towards infinity. To avoid this, Duda and Hart [31] defined lines by the so called normal parametrization. It represents a line by its distance r between the line and the origin, and the angle Θ of its normal (figure 4.1). This line can be written as

$$r = x * \cos\Theta + y * \sin\Theta \tag{4.3}$$

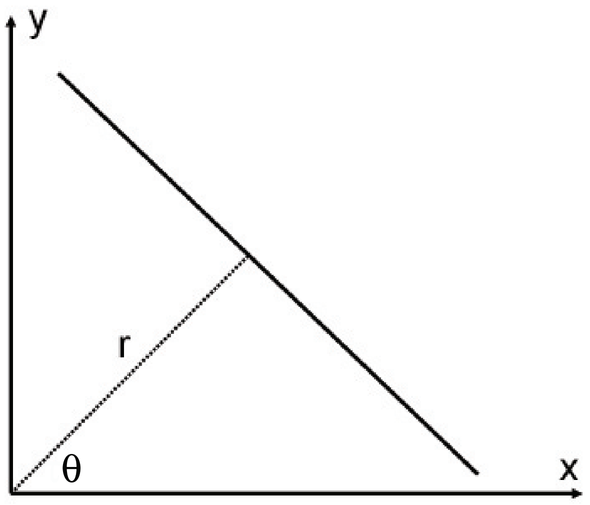


Figure 4.1: Line in image space with distance r and angle Θ

By either restricting the angle Θ to the interval $[0, \pi]$ and $r \in \mathbf{R}$ or $\Theta \in [0, 2\pi]$ and $r \geq 0$ the normal parameters for a line are unique and every line in the $x/y-$plane corresponds to a unique point in the $\Theta/r-$plane. The $\Theta/r-$plane is also called Hough space.

After transforming lines in an appropriate parameter space as shown above, it is possible to detect lines in an image with the following algorithm:

- Detection of possible line pixels LP by applying an edge detector (Canny Filter, Sobel Filter) to the image
- Discretization of the two parameters of the chosen parameter space, either t and m or r and Θ, in the following called $p1$ and $p2$. Discretization is needed, because otherwise the number of possible line parameters is infinite.
- By discretizing the parameter space, it is not continuous any more but represented by an array of $N_1 \times N_2$ rectangular cells. This array is called accumulator array A and its elements are accumulator cells $A(p1, p2)$ with start value 0

- For each LP, all allowed lines going through the LP with its parameters $p1$ and $p2$ are calculated and the corresponding accumulator cell $A(p1, p2)$ is incremented by one. This step is often referred to as voting.
- Accumulator cells with a high value are lines in the image, which are specified by the two line parameters $p1$ and $p2$
- Hence line detection in an image is reduced to finding local maxima in the accumulator space

As described above, the detection of lines is a two stage algorithm. The first stage is the incrementation or voting stage, the second is the maxima search stage. For the parameter space suggested by Duda and Hart $O(M * N_\Theta)$ operations are required in the voting stage and $O(N_r * N_\Theta)$ in the search stage, where M is the number of LPs and $N_{r/\Theta}$ the number of discretization steps of r/Θ where M is much bigger than N_Θ. Therefore computation time of line detection is dominated by the voting stage. An algorithm for reducing the computation time of this stage is called Probabilistic Hough Transform and will be described in the following section.

4.1.2 Probabilistic Hough Transform

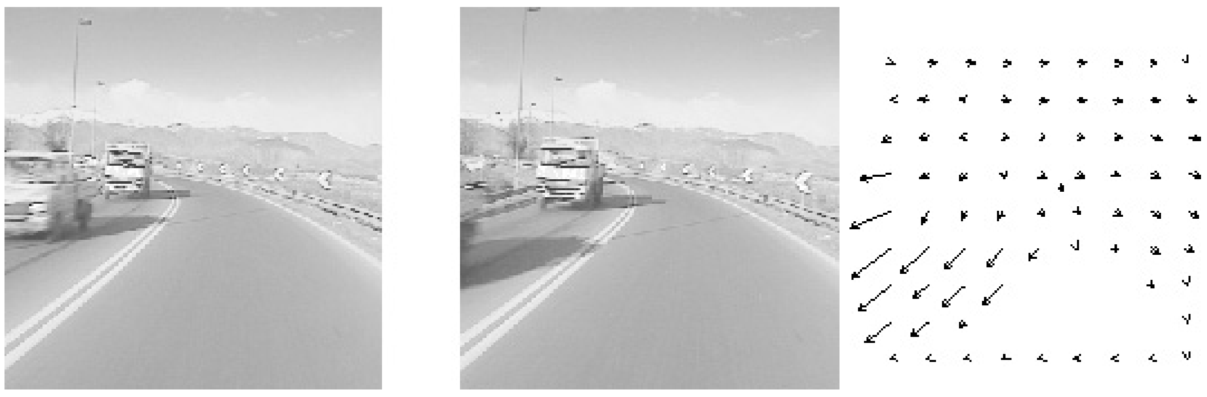
The Probabilistic Hough Transform was first introduced by N. Kiryat, Y. Eldar and A.M. Bruckstein in 1991 [32]. They suggested not to use all M detected LPs but only m with $m = M * \alpha$ where $0 < \alpha \leqq 1$ and therefore $m < M$. These m LPs are selected randomly from all M LPs. Thus a small value of m results in a fast algorithm because the number of operations in the voting stage is proportional to $m * N_\Theta$ which directly affects the complete algorithmic execution time. The lower boundary of m is limited by the need of a high probability of detecting a feature, which has a significant peak in the accumulator array of the standard method, even when using only m LPs.

To put it in a nutshell, it is sufficient to compute the Hough Transform only for a certain amount $0\% < \alpha \leqq 100\%$ of all possible LPs, taken from a randomly chosen uniform probability density function defined over the image. The results of [32] show, that an α approximately between 5% and 15% results in a remarkable reduction of computation time with a negligible reduction of performance.

4.2 Optical Flow

Detecting motion from image sequences can be used in a variety of image processing applications, like computer vision or pattern recognition. A moving object causes temporal varieties of the image brightness between two following images. These intensity changes are used to estimate motion. Horn and Schunk describe Optical Flow in [33] as: "Optical flow is the distribution of apparent velocities of movement of brightness patterns in an image".

In other words, by estimating the Optical Flow, the 2D-motion and the speed for every pixel between two consecutive images of an image sequence can be approximated by different methods, and result in a vector field shown in 4.2.



(a) Two consecutive images from a sequence (b) Optical flow

Figure 4.2: Optical flow between two images

This field arises primarily from relative motion between objects and the viewer, but also from brightness changes due to different illumination, reflections and other light effects. The term optical flow only describes this vector field but is often incorrectly referred to as the process or algorithm estimating the optical flow from an image. Optical flow estimation techniques can be separated into two classes:

Differential Techniques, which compute the velocity from spatiotemporal derivations of the image intensity which results in a dense flow field and

Area Matching Techniques, which define the velocity as the shift that returns the best fit between image areas at different instances of time which results in a sparse flow field.

4.2.1 Differential Techniques

The differential techniques derive the velocity at an image point by computing the spatiotemporal derivatives of the image intensity. These methods assume that a point occurring in the image has the same intensity in every following frame. In other words, the brightness of a scene point is considered to be constant over time. The basis of all differential optical flow estimation algorithms is known as the **motion constraint equation**.

Let $I(x, y, t)$ be the continuous space-time intensity function of an image point at time t with its x- and y-coordinates x and y, and $I(x + \delta x, y + \delta y, t + \delta t)$ the intensity function of the same point moved by δx and δy in time δt. As mentioned above, the intensity of a point remains constant over time, which means that

$$I(x, y, t) = I(x + \delta x, y + \delta y, t + \delta t) \tag{4.4}$$

A constant brightness also means that $\frac{dI}{dt} = 0$. Applying first order Taylor series to $I(x, y, t)$ and neglecting higher derivative orders, assuming them to be too small, results in

$$I(x + \delta x, y + \delta y, t + \delta t) = I(x, y, t) + \frac{\partial I}{\partial x}\partial x + \frac{\partial I}{\partial y}\partial y + \frac{\partial I}{\partial t}\partial t \tag{4.5}$$

Inserting 4.4 in 4.5 and dividing the result by ∂t leads to

$$\frac{\partial I}{\partial x}v_x + \frac{\partial I}{\partial y}v_y + \frac{\partial I}{\partial t} = 0 \tag{4.6}$$

where $v_x = \partial x/\partial t$ is the velocity component in x direction and $v_y = \partial y/\partial t$ the velocity component in y direction which are referred to as optical flow. For easier reading and writing, in the following the partial derivatives will be written as $I_x = \partial I/\partial x$, $I_y = \partial I/\partial y$ and $I_t = \partial I/\partial t$. Now 4.6 can be compactly written as

$$[I_x, I_y] * [v_x, v_y]^T + I_t = 0 \tag{4.7}$$

However, a unique solution for equation 4.7 can only be determined if motion is parallel to the brightness gradient. This problem is known as the aperture problem, which says that motion of a homogeneous contour can be locally ambiguous. Regarding image contours through an aperture, different physical motions are indistinguishable. As shown in figure 4.3 the two different movements shown in picture (b) and (c) appear to be the same regarding them through an aperture. Only the normal component v_n can be determined but not the flow velocity $\mathbf{v} = [v_x, v_y]^T$. To solve this problem, additional constraints have to be made.

Horn and Schunk suggest in [33] to add the so called smoothness constraint. It says that not every point of a brightness pattern moves independently because objects of finite size undergo rigid motion or deformation. In this case, neighboring points on the objects have similar velocities and the velocity field of the brightness patterns in the image varies smoothly almost everywhere. By minimizing 4.8 the optical flow can be computed.

$$\int_D ([I_x, I_y] * [v_x, v_y]^T + I_t) + \lambda^2 \left[\left(\frac{\partial v_x}{\partial x}\right)^2 + \left(\frac{\partial v_x}{\partial y}\right)^2 + \left(\frac{\partial v_y}{\partial x}\right)^2 + \left(\frac{\partial v_y}{\partial y}\right)^2 \right] dxdy \tag{4.8}$$

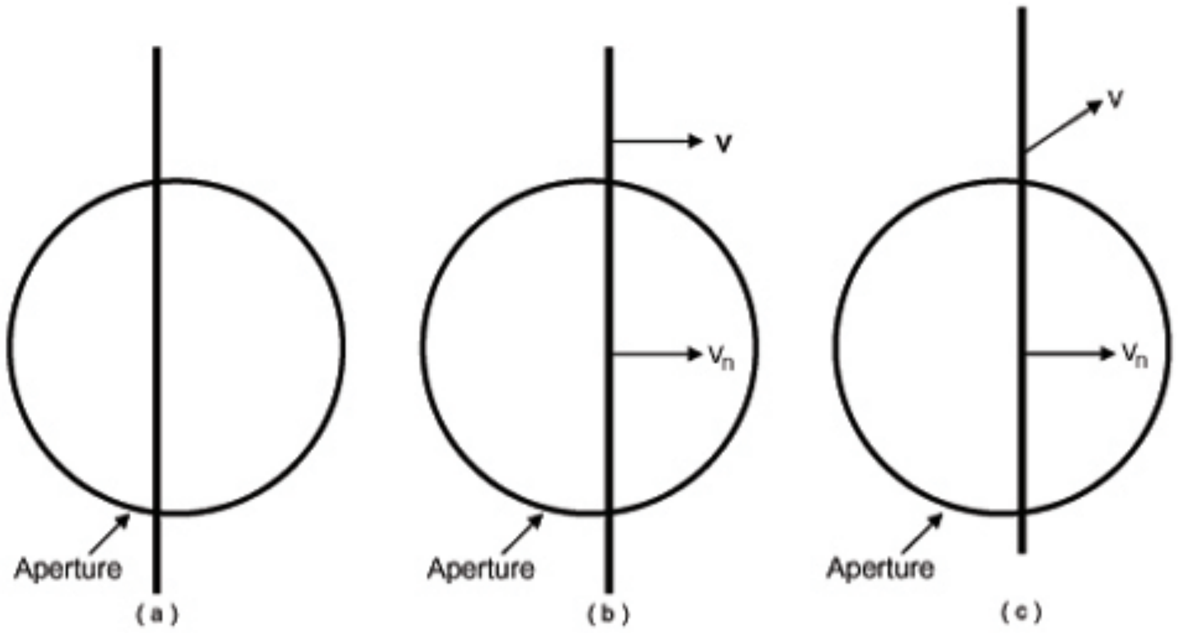


Figure 4.3: Regarding image contour through aperture

D denotes the domain, the whole image, over which the equation is defined and λ defines the relative influence of the smoothness term. Horn and Schunk suggest using iterative methods as Gauss-Seidel algorithm for minimizing 4.8 and obtaining the optical flow. This global method results in a very dense flow field but is sensitive to noise.

A second popular approach for determining the optical flow is presented by Lucas and Kanade in [34]. They assume that motion, and with it the flow velocity $\mathbf{v}$ in a small area, defined by a window Q of size $M \times M$, is constant. This assumption leads to $n = M^2$ equations, one for each pixel in the window.

$$\begin{aligned} I_{x_1} * v_x + I_{y_1} * v_y + I_{t_1} &= 0 \\ I_{x_2} * v_x + I_{y_2} * v_y + I_{t_2} &= 0 \\ &\vdots \\ I_{x_n} * v_x + I_{y_n} * v_y + I_{t_n} &= 0 \end{aligned} \tag{4.9}$$

With this there are more than two equations for the two unknowns, which means the system is over-determined. This system can be solved by a least squares method. Rewriting the equations of 4.9 in matrix vector notation follows in

$$\begin{aligned} \begin{bmatrix} I_{x_1} & I_{y_1} \\ I_{x_2} & I_{y_2} \\ \vdots & \vdots \\ I_{x_n} & I_{y_n} \end{bmatrix} * \begin{bmatrix} v_x \\ v_y \end{bmatrix} &= - \begin{bmatrix} I_{t_1} \\ I_{t_2} \\ \vdots \\ I_{t_n} \end{bmatrix} \\ \mathbf{A} \quad * \quad \mathbf{v} \quad &= \quad -\mathbf{b} \end{aligned} \tag{4.10}$$

The flow vector $\mathbf{v}$ can then be determined with the Moore-Penrose pseudoinverse of matrix $\mathbf{A}$

$$\mathbf{v} = (\mathbf{A}^T \cdot \mathbf{A})^{-1} \cdot \mathbf{A}^T \cdot (-\mathbf{b}) \tag{4.11}$$

Additionally Lucas and Kanade introduced a weighting function W that gives more prominence to the pixels in the center of Q. The final equation for determining the flow vector which is assigned to the center of Q is

$$\mathbf{v} = (\mathbf{A}^T \cdot W^2 \cdot \mathbf{A})^{-1} \cdot \mathbf{A}^T \cdot W^2 \cdot (-\mathbf{b}) \tag{4.12}$$

In contrast to the method introduced by Horn and Schunk, this local approach does not result in a very high density of flow vectors but is more robust against noise. For further reading, a fast implementation of Lucas and Kanade algorithm, using pyramid segmentation is presented in [35].

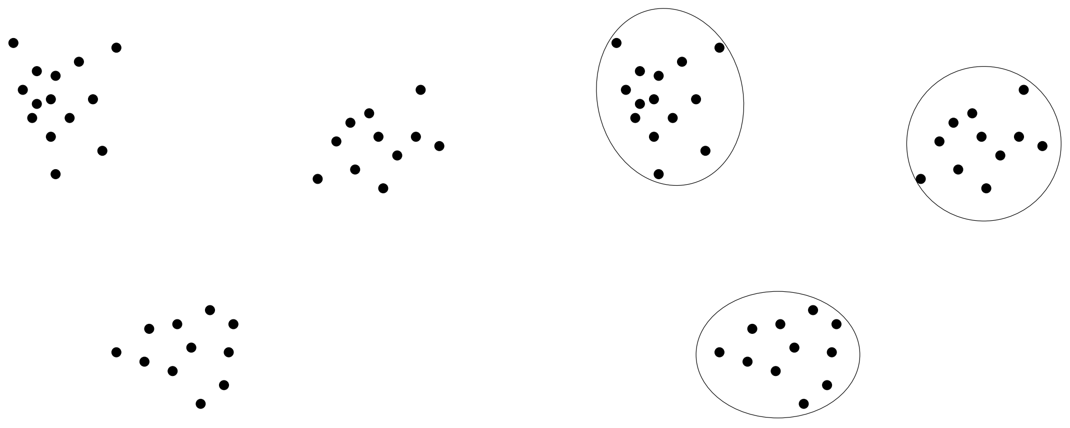
4.2.2 Area Matching Techniques

Algorithms based on numerical differentiation may be impractical due to noise or aliasing artifacts. In these cases, area matching techniques can be used to achieve better results. These algorithms define the velocity $\mathbf{v}$ as the shift $\mathbf{d} = (dx, dy)$ between image areas at different instances of time that returns the highest similarity. Similarity can be measured by different methods, such as cross correlation, Sum of Absolute Differences (SAD) or Sum of Squared Differences (SSD), which will be explained in section 5.4.

A popular method that can be referred to as the class of area matching techniques is the **Kanade-Lucas-Tomasi** feature tracker (KLT) [36]. The basic principle of the KLT is the termination of features that can be tracked well. A good feature is thereby defined as a textured patch with high intensity variation in both directions x and y. Therefore, points with high eigenvalues, such as corners in an image are potentially good candidates. Two kinds of image notion models are used: the simple translation model and the affine model, which represents translation plus linear warping. For estimating the movement of an area between two consecutive frames, the standard Lucas-Kanade algorithm as presented in chapter 4.2.1 will be applied and similarity will be measured restricted to pure translational motion. After that, a root-mean squared error dissimilarity measure between the new frame and the first one the feature occurred in will be applied, using the full affine motion model. This step is called monitoring step, and tries to secure that motion is determined for the same features. If the dissimilarity grows too high, the feature is likely to be lost and will therefore be abandoned.

4.3 Clustering

In order to organise and reduce the amount of an unknown set of data, clustering is the procedure that subdivides this set of data into so called clusters, which can be seen in figure 4.4.



(a) Set of points (b) Clustered points

Figure 4.4: Cluster example

The data in each cluster share a common feature or similarity defined by a certain distance measure, while every cluster is different. Clustering is used in a broad field of data analysis e.g. bioinformatics, data-mining, machine-learning and image analysis.

As mentioned above, a distance measure to determine the similarity of two elements is an essential part of every clustering algorithm and will highly influence the resulting cluster shape. Commonly used distance functions are the Euclidean distance, Manhattan distance or Mahalanobis distance.

Generally, there are two types of clustering algorithms called hierarchical and partitional described in the following.

4.3.1 Hierarchical Clustering

Hierarchical clusters are divided into agglomerative ("bottom-up") or divisive ("top-down") methods. The more commonly used agglomerative method starts with each element of a data set as a separate cluster, and step by step merges them to bigger clusters, whereas the divisive method starts with a single cluster including the whole data and dividing it into smaller clusters. The hierarchical methods result in a tree structure which can be visualized as a dendrogram.

4.3.2 Partitional Clustering

In contrast to building clusters out of already established clusters, as done with hierarchical methods, partitional methods determine them directly from the whole data set. There are a lot of different partitional clustering algorithms, like the EM-Algorithm [37], spectral clustering [38] or self-organizing maps but the most commonly used one is the **k-means algorithm** which will be described in detail.

First introduced by J.B. MacQueen [39] in 1967 the k-means algorithm became very popular because of producing good clustering results while requiring low computation time. The procedure follows a simple way to classify a given data set through a certain number **k** of clusters fixed a priori. It starts by choosing the value k and dividing the data set into k different sets. These sets are either chosen randomly or by using heuristic data. After that, the mean point, called centroid, is calculated for each of the k parts. The next step is associating every element of the data set to its nearest centroid. At this point, k new centroids have to be re-calculated as barycenters of the clusters resulting from the previous step. After having defined these k new centroids, a new binding has to be built between the data set points to be processed and the nearest new centroid. A loop has been generated. As a result of this loop it may be noticed, that the k centroids change their location step by step until no more changes are done. Mathematically speaking the algorithm tries to minimize a squared error function 4.13

$$E = \sum_{i=1}^{k} \sum_{x_j \in S_i} |x_j - \mu_i|^2 \tag{4.13}$$

with k clusters $S_i, i = 1, \ldots, k$ and μ_i the centroid of all data points $x_j \in S_i$.

The algorithm consists of the following five steps:

1. Selection: Selecting the number k of clusters
2. Initialization: Choosing k cluster centroids
3. Assignment: Assigning each data point to the nearest centroid

4. Recalculation: Recomputing the centroids
5. Repetition: Repeating step 3 and 4 until some convergence criterion is met (usually until the assignment has not changed)

K-means does not necessarily find the most optimal configuration, corresponding to the minimum of the objective function 4.13 because the result highly depends on the initial random choice of the k cluster centroids. This effect can be reduced by running k-means several times with different initialization.

5 Stereo Vision

One of the problems in image processing is that a single camera viewing a three dimensional scene only produces a two dimensional image which means a loss of one dimension and therefore a loss of information. This problem can be solved by comparing two images of the same scene, either taken by two different cameras at the same time or, in case of a static scene, taken by a single camera from two different positions at different time and looking for corresponding points between the two images. The missing depth information can then be calculated via triangulation. The crucial part thereby is the detection of correct correspondences. In the following sections an image preprocessing step and the required mathematical background for simplifying the correspondence search will be presented. After that, three different methods for finding similarities will be explained.

5.1 Epipolar Geometry

The Epipolar Geometry describes the projective relation between points in two camera images. An example set of two cameras C1 and C2, observing an arbitrary 3D point $\mathbf{m}$ is shown in figure 5.1 with its known optical centers $\mathbf{c}_1$ and $\mathbf{c}_2$. The connection of the two camera centers is called baseline B which intersects the imageplanes I_1 and I_2 in the points $\mathbf{e}_1$ and $\mathbf{e}_2$ which are known as *epipoles*. The epipoles can be interpreted as the projection of one camera center into the image plane of the other camera. The position of the epipoles only depend on the position of the cameras. If the cameras' optical axes are parallel, the epipoles lie at infinity. The baseline B and a point $\mathbf{m}$ create a plane Π called *epipolar plane*. Given an image point $\mathbf{m}_1$ in I_1 the corresponding 3D point $\mathbf{m}$ could be any point on the semi line $\mathbf{c}_1\mathbf{m}_1$, which also lies on Π. The projection of $\mathbf{c}_1\mathbf{m}_1$ on I_2 is called *epipolarline* ℓ_2 of $\mathbf{m}_1$ which also results in intersecting Π and I_2. This line passes the epipole $\mathbf{e}_2$ and, most important, contains $\mathbf{m}_2$, which is the image point of $\mathbf{m}$ on I_2. By rotating the epipolar plane around the baseline every possible 3D point is captured, all epipolarlines are created and result in the so called pencil of epipolarlines intersecting in the epipole. This leads to the *epipolar constraint* which says that the correspondence of a point in the first image must lie on the epipolarline in the second image. Therefore the search space is reduced from two dimensions to one, which results in a significant reduction of computation time. An example for two images taken from different views, some points of interest and the corresponding epipolarlines can be seen in figure 5.2. The Epipolar Geometry is algebraically described by the **Fundamental Matrix F**.

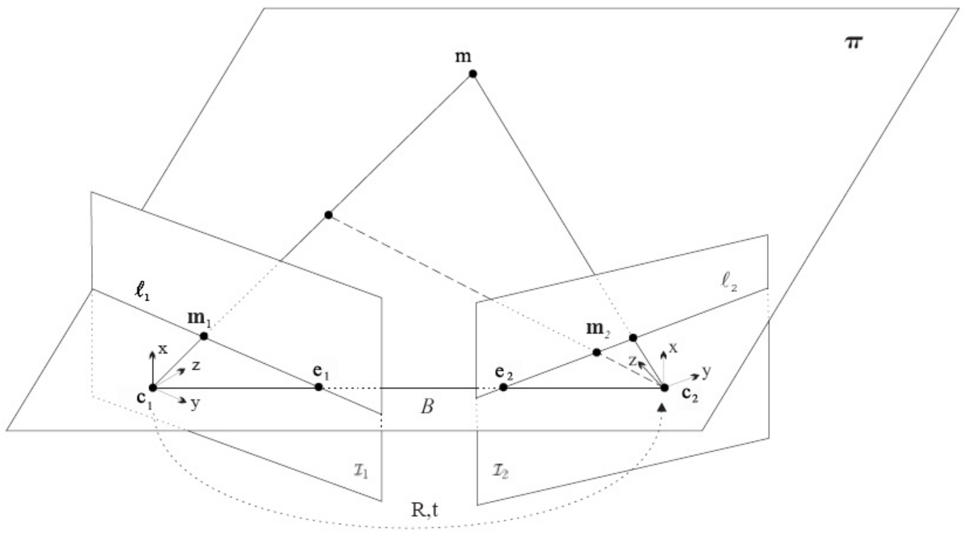


Figure 5.1: Epipolar Geometry

5.2 Fundamental Matrix and Essential Matrix

Given a pair of images, it was seen in 5.1 that to each point $\mathbf{m}_1$ in image one, the corresponding point $\mathbf{m}_2$ in the second image must lie on the epipolarline ℓ_2. Thus, there is a map

$$\mathbf{m}_1 \mapsto \ell_2 \tag{5.1}$$

from a point in one image to its corresponding epipolar line in the other image, which is represented by the Fundamental Matrix.

A 3D point in the coordinate system of the first camera $\mathbf{m}_{C1} = [x_{C1}, y_{C1}, z_{C1}]^T$, can be expressed in the coordinate system of the second camera $\mathbf{m}_{C2} = [x_{C2}, y_{C2}, z_{C2}]^T$ by a rigid transformation

$$\mathbf{m}_{C2} = \mathbf{R} \cdot \mathbf{m}_{C1} + \mathbf{t} \tag{5.2}$$

where $\mathbf{R}$ is the orthogonal rotation matrix and $\mathbf{t}$ is the displacement vector between the two camera centers calculated by $\mathbf{t} = \mathbf{c}_1 - \mathbf{c}_2$. In the following it is assumed that the camera performs an exact perspective projection, meaning the cameras are calibrated. $\tilde{\mathbf{m}}_1$ and $\tilde{\mathbf{m}}_2$ are the projection of the 3D point $\mathbf{m}$ on the respective image planes, represented in normalized homogenous coordinates[1)] meaning that the z-coordinate equals one. Inserting

[1)] For two vectors $\mathbf{a}$ and $\mathbf{b}$ in homogenous coordinates the following mathematical conditions are valid: $\mathbf{a} \times \mathbf{a} = \mathbf{0}$ where $\mathbf{0}$ is the zero vector $[0, 0, 0]^T$ and $\mathbf{a}^T \cdot (\mathbf{b} \times \mathbf{a}) = 0$

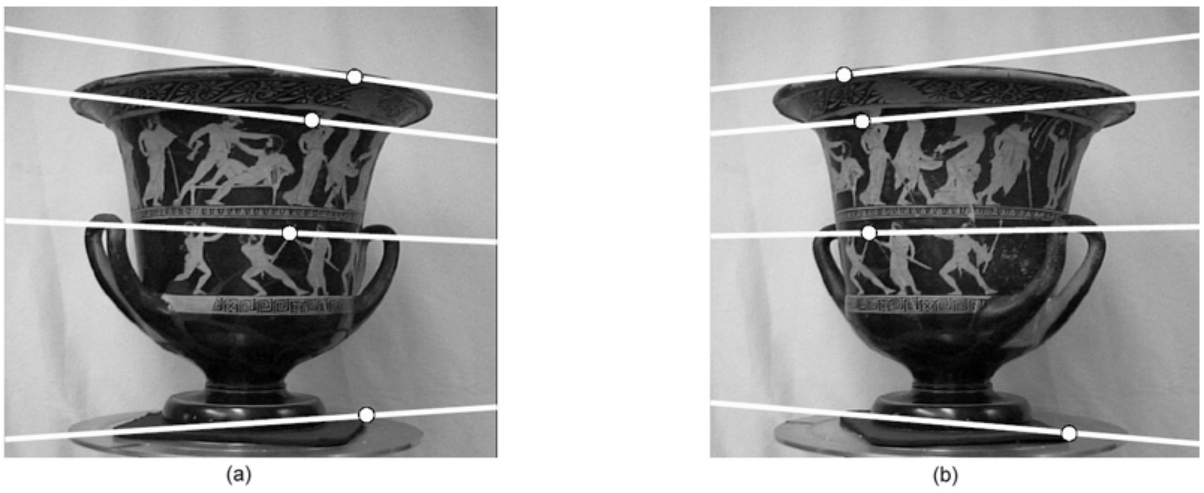


Figure 5.2: Unrectified Epipolar Lines

them in 5.2 leads to the central equation of the epipolar geometry

$$\tilde{\mathbf{m}}_2^T \cdot [\mathbf{t}]_\times \mathbf{R} \cdot \tilde{\mathbf{m}}_1 = 0 \tag{5.3}$$

$[\mathbf{t}]_\times$ is known as the antisymmetric matrix defined by $\mathbf{t}$, such that $[\mathbf{t}]_\times = \mathbf{t} \times \mathbf{x}$ for all 3D vectors $\mathbf{x}$

$$[\mathbf{t}]_\times = \begin{bmatrix} 0 & -t_z & t_y \\ t_z & 0 & -t_x \\ -t_y & t_x & 0 \end{bmatrix} \tag{5.4}$$

The cross product $[\mathbf{t}]_\times \mathbf{R}$ results in the matrix $\mathbf{E}$ of dimension 3×3, called the **Essential Matrix** which was first introduced by Longout-Higgins in [40]. This matrix describes the geometric relation between corresponding points in camera coordinates of both cameras of the stereo system. Because $[\mathbf{t}]_\times$ has $rank = 2$, $\mathbf{E}$ has the same rank and therefore only two rows or columns are linearly independent.

Now assuming the case of a non-calibrated camera system and thus a non-perfect perspective projection, the camera coordinates are not equal to the pixel coordinates but undergo a transformation depending on the internal camera parameters which are focal length f, principal point $[u_0, v_0]^T$, skew coefficient $[k_u, k_v]^T$ (angle between x and y axes) and distortions s. These parameters can be combined in the *intrinsic cameramatrix*

$$\mathbf{A} = \begin{bmatrix} f \cdot k_u & s & u_0 \\ 0 & f \cdot k_v & v_0 \\ 0 & 0 & 1 \end{bmatrix} \tag{5.5}$$

With the intrinsic cameramatrix of the respective camera, a point in camera coordinates $\tilde{\mathbf{m}}_1$,$\tilde{\mathbf{m}}_2$ can be transformed into pixel coordinates $\mathbf{m}_1$,$\mathbf{m}_2$ by

$$\begin{aligned}\mathbf{m}_1 &= \mathbf{A}_1 \cdot \tilde{\mathbf{m}}_1 \\ \mathbf{m}_2 &= \mathbf{A}_2 \cdot \tilde{\mathbf{m}}_2\end{aligned} \tag{5.6}$$

Inserting 5.6 in 5.3 leads to

$$\begin{aligned}\mathbf{m}_2^T \cdot \mathbf{A}_2^{-T} \cdot [\mathbf{t}]_\times \mathbf{R} \cdot \mathbf{A}_1^{-1} \cdot \mathbf{m}_1 &= 0 \\ \mathbf{m}_2^T \cdot \mathbf{A}_2^{-T} \cdot \mathbf{E} \cdot \mathbf{A}_1^{-1} \mathbf{m}_1 = \mathbf{m}_2^T \cdot \mathbf{F} \cdot \mathbf{m}_1 &= 0\end{aligned} \tag{5.7}$$

The matrix product $\mathbf{A}_2^{-T} \cdot \mathbf{E} \cdot \mathbf{A}_1^{-1}$ defines the Fundamental Matrix $\mathbf{F}$, of same dimension as $\mathbf{E}$ and also with $rank = 2$. It has the following important properties:

- **Transpose**: if $\mathbf{F}$ is the Fundamental Matrix between C1 and C2 then $\mathbf{F}^T$ is the Fundamental Matrix between C2 and C1
- **Epipoles**: for any point $\mathbf{m}_1 \neq \mathbf{e}_1$ the epipolar line $\ell_2 = \mathbf{F} \cdot \mathbf{m}_1$ contains the epipole $\mathbf{e}_2$. Thus $\mathbf{e}_2$ satisfies $\mathbf{e}_2^T \cdot (\mathbf{F} \cdot \mathbf{m}_1) = (\mathbf{e}_2^T \cdot \mathbf{F}) \cdot \mathbf{m}_1 = 0$ for all $\mathbf{m}_1$. It follows that $\mathbf{e}_2^T \cdot \mathbf{F} = \mathbf{0}$ where $\mathbf{0}$ denotes the zero vector $[0,0,0]^T$, i.e. $\mathbf{e}_2$ is the left null-space of $\mathbf{F}$. Accordingly, $\mathbf{F} \cdot \mathbf{e}_1 = \mathbf{0}$, i.e. $\mathbf{e}_1$ is the right null-space of $\mathbf{F}$
- **Epipolarlines**: for any point $\mathbf{m}_1$ in the first image, its epipolar line is $\ell_2 = \mathbf{F} \cdot \mathbf{m}_1$ in the second image. Similarly, $\ell_1 = \mathbf{F}^T \cdot \mathbf{m}_2$ represents the epipolar line in the first image corresponding to $\mathbf{m}_2$ in the second image. A line thereby is presented by three coefficients $\ell = [a,b,c]^T$ and the following equations

  $$\begin{aligned} a * x + b * y + c &= 0 \\ \ell^T [x,y,1]^T &= 0 \end{aligned} \tag{5.8}$$

  In the more common and appealing representation the slope m is equal to $-a/b$ and the intersect t is equal to $-c/b$

The last point mentioned above is the most important one for the correspondence search problem, because it eventually performs the reduction of the search space by one dimension.

The following sections will describe two methods of determining the Fundamental Matrix of any stereo camera system, with the help of image points in the two camera images assumed to represent the same 3D world point (already known correspondences).

5.2.1 Eight-Point Algorithm

The **Eight-Point Algorithm** is a linear method of calculating $\mathbf{F}$ out of at least eight corresponding points and was first introduced in [40]. The detection of these correspondences can be done by methods like Harris corner detection or the method of Lucas and

Kanade described in section 4.2, but will not be explained here.
Given is a set of $n \geq 8$ corresponding points in homogenous coordinates between the two images denoted by $\mathbf{m}_{k,i} = [u_{k,i}, v_{k,i}, 1]^T$ $k = 1, 2$ $i = 1, \ldots n$ where point $\mathbf{m}_{1,i}$ corresponds to $\mathbf{m}_{2,i}$. Inserting these points into 5.7 leads to n equations of the following form

$$u_{1,i}u_{2,i}f_{1,1}+v_{1,i}u_{2,i}f_{1,2}+u_{2,i}f_{1,3}+u_{1,i}v_{2,i}f_{2,1}+v_{1,i}v_{2,i}f_{2,2}+v_{2,i}f_{2,3}+u_{1,i}f_{3,1}+v_{1,i}f_{3,2}+f_{3,3} = 0 \tag{5.9}$$

where $f_{l,r}$ is an element of $\mathbf{F}$ in line l and row r. Creating the vectors
$\mathbf{p}_i = [u_{1,i}u_{2,i}, v_{1,i}u_{2,i}, u_{2,i}, u_{1,i}v_{2,i}, v_{1,i}v_{2,i}, v_{2,i}, u_{1,i}, v_{1,i}, 1]^T$ and
$\mathbf{f} = [f_{1,1}, f_{1,2}, f_{1,3}, f_{2,1}, f_{2,2}, f_{2,3}, f_{3,1}, f_{3,2}, f_{3,3}]^T$

The equation can be written in a compact form

$$\mathbf{p}_m^T \cdot \mathbf{f} = 0 \tag{5.10}$$

Combining all n vectors $\mathbf{p}$ into the matrix $\mathbf{P} = [\mathbf{p}_1, \ldots, \mathbf{p}_n]^T$ leads to the final linear system of equations

$$\mathbf{P} \cdot \mathbf{f} = 0 \tag{5.11}$$

The goal of determining $\mathbf{F}$ and $\mathbf{f}$ respectively can be achieved by solving the least squares problem

$$\min_{\mathbf{F}} \sum_n (\mathbf{m}_{2,i}^T \mathbf{F} \mathbf{m}_{1,i})^2 \tag{5.12}$$

or

$$\min_{\mathbf{f}} \|\mathbf{P}\mathbf{f}\|^2 \tag{5.13}$$

If $n \geq 8$ and $rank(\mathbf{P}) \geq 8$, meaning the points are not coplanar, the vector $\mathbf{f}$ is defined up to a scaling factor. For finding a solution other than the trivial one $\mathbf{f} = \mathbf{0}$, the Singular Value Decomposition (SVD) of $\mathbf{P}$ can be used. It results in

$$\mathbf{P} = \mathbf{U}\mathbf{D}\mathbf{V}^T \tag{5.14}$$

where $\mathbf{D}$ is a diagonal matrix, containing all singular values which are positive and in decreasing order $\sigma_1 \geq \sigma_2 \geq \ldots \geq \sigma_9$ in its trace and $\mathbf{V}$ containing the corresponding eigenvectors. The minimization problem 5.13 is satisfied by choosing $\mathbf{f}$ equal to the eigenvector that corresponds to the smallest singular value of $\mathbf{P}$. Using the decreasing order of the singular values, σ_9 is the smallest, and therefore the last column of $\mathbf{V}$ is the wanted solution for $\mathbf{f}$. It is trivial to get the wanted Fundamental Matrix $\mathbf{F}$ out of $\mathbf{f}$. However, in presence of noise and due to incorrect correspondence measures, the resulting $\mathbf{F}$ will not fulfill $rank(\mathbf{F}) = 2$, meaning that the epipolar lines will not all pass through a common point, in other words there will not exist real epipoles but they will be "smeared out" to a small region. This problem can be solved by performing another SVD, $\mathbf{F} = \hat{\mathbf{U}}\hat{\mathbf{D}}\hat{\mathbf{V}}^T$, setting the smallest resulting singular value to 0 and calculating the rank 2 final Fundamental Matrix by $\mathbf{F} = \hat{\mathbf{U}}\tilde{\mathbf{D}}\hat{\mathbf{V}}^T$ with $\tilde{\mathbf{D}} = diag(\sigma_1, \sigma_2, 0)$.

By just taking the pixel coordinates of the corresponding points without any preprocessing step, the algorithm shows poor performance due to poor conditioning. To deal with this problem, R. Hartley proposed in [41] to apply a normalization of input data before executing the eight-point algorithm. This preprocessing consists of two steps

1. Translating the points so that their centroid lies in the image origin
2. Scaling the points such that the average distance of all points (already translated as mentioned in step one) to the origin is equal to $\sqrt{2}$

Doing this preprocessing for both images independently, and executing the eight-point algorithm with the preprocessed data, the results presented in [41] were nearly as good as the more complex iterative methods described in [42], but less computationally expensive.

5.2.2 Eight-Point Algorithm and RANSAC

The problem of the above described Eight-Point Algorithm and all least-squares approaches is that they can not cope with outliers. Because of the quadratic error function, a single outlier that differs strongly from the true solution completely biases the final result. It is very challenging to segment the set of correspondences in inliers and outliers before having obtained the correct solution.

A way to solve this problem was proposed by Fischler and Bolles in [43]. Their algorithm, called RANSAC (RANdom SAmpling Consensus), was not developed with the goal of improving the calculation of the Fundamental Matrix, but can be applied to this and many other kinds of least-squares problems.

Having a set of $n > 8$ matching pairs, a random minimal subset of $k = 8$ pairs is taken out of this subset and $\mathbf{F}$ is calculated with these 8 pairs and the eight-point algorithm. With a certain probability this subset contains no outliers and the correct solution for $\mathbf{F}$ is found. A point is referred to as outlier if the distance to its epipolar-line, calculated with $\mathbf{F}$, is bigger than a certain threshold τ. Otherwise it is called inlier. The RANSAC algorithm maximizes the probability of having an subset without outliers, by repeating the procedure of choosing random subsets and calculating $\mathbf{F}$. Each time the set of n pairs is divided into in- and outliers by the method mentioned above, using the Fundamental Matrix calculated in the respective step. The solution for $\mathbf{F}$ with the largest amount of inliers is identified as the correct one. The crucial point is to determine the amount of repetitions necessary for getting a high probability of inliers. Assuming the fraction of outliers in the whole set is ε, then the probability that the subset of k pairs contains no outlier is $(1-\varepsilon)^k$ and the probability that a number s of different subsets contain one or more outliers is $(1-(1-\varepsilon)^k)^s$. Finally, the probability that at least one of these s random subsets has no outlier is

$$P = 1 - (1 - (1-\varepsilon)^k)^s \tag{5.15}$$

Thus the number s of iterations for getting a subset without outliers with the probability of P is

$$s = \frac{ln(1-P)}{ln(1-(1-\varepsilon)^k)} \tag{5.16}$$

Typically, the threshold τ for distinguishing in- and outliers is 0.5 or 1 pixel. The probability P should be chosen between 0.95 and 0.99. The algorithm can easily deal with a fraction of up to $\varepsilon = 50\%$ of outliers. Above this percentage, the number of iterations becomes very high, as shown in table 5.1.

fractions of outliers ε	5%	10%	20%	30%	40%	50%	60%	70%
iterations s	3	6	17	51	177	765	4597	45658

Table 5.1: Number of iterations necessary to achieve a probability of 0.95 to have one subsample of $k = 8$ pairs containing no outlier, for different fixed fractions of outliers

5.3 Image Rectification

In the previous sections the Epipolar Geometry was introduced, and it was shown that a point in the first image must a lie on a line, the epipolar line, in the second image. With this knowledge the search space can be reduced significantly from two to one dimension. However, the epipolar lines normally are slanted as shown in figure 5.2 and therefore search along these lines is difficult and time-consuming. In order to speed up the search algorithms, the epipolar lines are desired to be axis-aligned (usually to the horizontal axis) and parallel, as points in one image then have the same y-coordinate as their corresponding point in the other image shown in 5.3. This can be achieved by a process called **image rectification**.

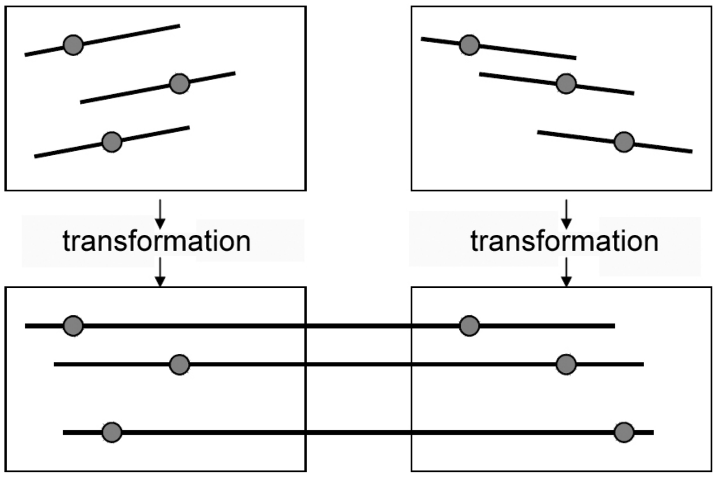


Figure 5.3: Rectification goal

Given is a set of two cameras with non-parallel optical axes and thus non-coplanar image planes. As described in section 5.1, all epipolar lines intersect in their respective epipole, and the epipole in image two is the projection of camera center one into image plane two and vice versa. Thus, epipolar lines would be parallel, if the epipoles lay at infinity, because parallel lines intersect at infinity. Epipoles lie at infinity if the image planes are coplanar, which can be achieved by reprojecting the two image planes onto a common plane, parallel to the baseline (the connection between the two camera centers) as seen in figure 5.4. The transformation of each plane is accomplished by applying a so called **homography**.

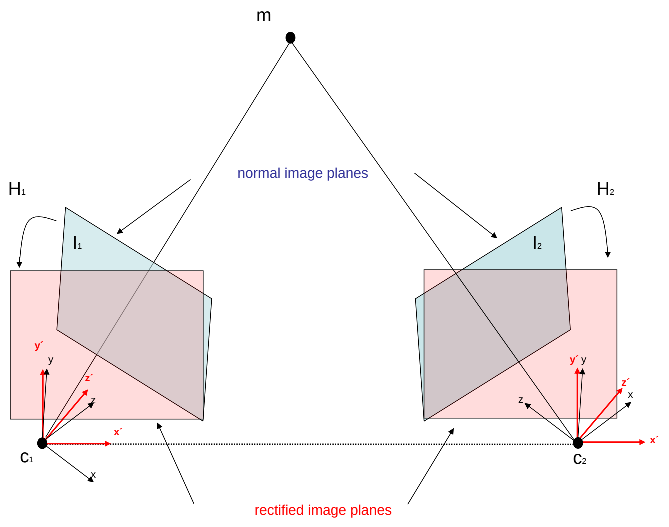


Figure 5.4: Set of 2 cameras with imageplanes and the wanted rectified planes

Let $\mathbf{H_1}$ and $\mathbf{H_2}$ be the homographies applied to image I_1 and I_2 respectively, then the positions of $\mathbf{m}_1 \epsilon I_1$ and $\mathbf{m}_2 \epsilon I_2$ in the rectified images are calculated by

$$\hat{\mathbf{m}}_1 = \mathbf{H}_1 \cdot \mathbf{m}_1 \qquad \hat{\mathbf{m}}_2 = \mathbf{H}_2 \cdot \mathbf{m}_2 \tag{5.17}$$

If $\mathbf{m}_1$ and $\mathbf{m}_2$ fulfil equation 5.7 with given $\mathbf{F}$, then $\hat{\mathbf{m}}_1$ and $\hat{\mathbf{m}}_2$ will fulfil it accordingly, with a matrix $\hat{\mathbf{F}}$. As the epipoles should lie at infinity and epipolar lines should be aligned to the x-axes and have the same y-coordinate as the corresponding point in the other image, the matrix $\hat{\mathbf{F}}$ is given by[2)]

$$\hat{\mathbf{F}} = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & -1 \\ 0 & 1 & 0 \end{bmatrix} \tag{5.18}$$

[2)] To prove that $\hat{\mathbf{F}}$ fulfils the requirements, the properties of a Fundamental Matrix presented in section 5.2 can be used. As an epipolar line ℓ_2 of a point $\mathbf{m}_1 = [x_1, y_1, 1]^T$ is calculated with $\hat{\mathbf{F}}$ by $\ell_2 = \hat{\mathbf{F}} \cdot \mathbf{m}_1$ and results in $\ell_2 = [0, -1, y_1]$ or with 5.8 $y = y_1$, it is proved that lines are parallel to the x-axis and have the corresponding y-coordinate. The epipoles at infinity can be expressed in homogenous coordinates by $\mathbf{e}_1 = [1, 0, 0]^T$ and $\mathbf{e}_2 = [-1, 0, 0]^T$. The given $\hat{\mathbf{F}}$ satisfies $\hat{\mathbf{F}} \cdot \mathbf{e}_1 = \hat{\mathbf{F}}^T \cdot \mathbf{e}_2 = \mathbf{0}$. Thereby all requirements are fulfilled.

Inserting 5.17 into 5.7 with 5.18 yields

$$\mathbf{m}_2^T \mathbf{H}_2^T \hat{\mathbf{F}} \mathbf{H}_1 \mathbf{m}_1 = 0 \tag{5.19}$$

and therefore

$$\mathbf{F} = \mathbf{H}_2^T \hat{\mathbf{F}} \mathbf{H}_1 \tag{5.20}$$

The matrices $\mathbf{H}_2$ and $\mathbf{H}_1$ are not unique but should be calculated with the aim of minimizing image distortion. Loop and Zhang show a solution for this problem in [44]. They suggest to split up the matrices $\mathbf{H}_1$ and $\mathbf{H}_2$ respectively into three different matrices:

1. $\mathbf{H}_p$: a specialized projective transform which performs the transformation of the epipoles to infinity, which results in parallel epipolar lines
2. $\mathbf{H}_r$: a similarity transform that rotates the epipolar lines in a way that they are aligned with the x-axis plus a translation in one image into y-direction, such that corresponding epipolar lines lie on the same scanline, meaning they have the same y-coordinate in both images
3. $\mathbf{H}_s$: a shearing transform for reducing horizontal distortion

For a mathematical description of these matrices and how to derive them, please refer to [44]. The complete homographies are computed by multiplying the three respective matrices described above $\mathbf{H}_1 = \mathbf{H}_{s,1}\mathbf{H}_{r,1}\mathbf{H}_{p,1}$ and $\mathbf{H}_2 = \mathbf{H}_{s,2}\mathbf{H}_{r,2}\mathbf{H}_{p,2}$. An example image pair of how a rectification result can look like is shown in figure 5.5.

Apart from the Loop-Zhang method, Fusiello et al. present in [45] a method for finding

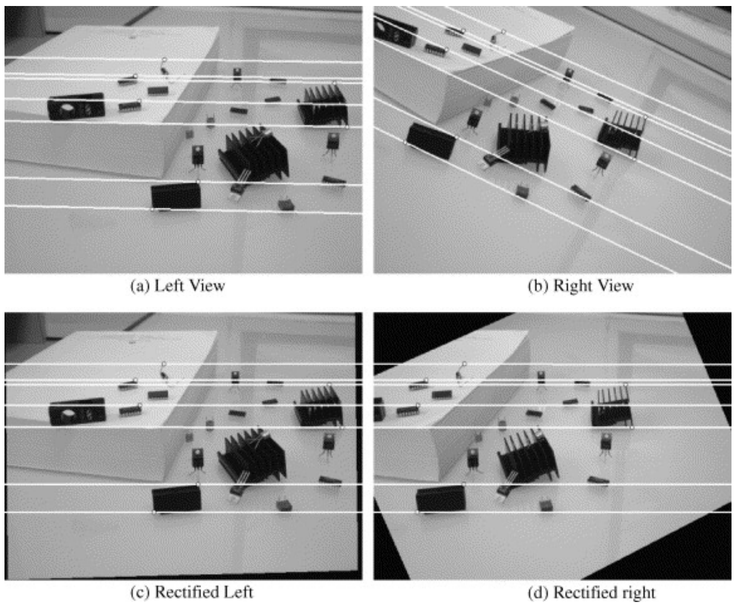


Figure 5.5: Example for the rectification procedure. The original images are shown in (a) and (b), overlaid by their respective epipolar lines. After rectification, these lines become collinear and parallel with the image's x-axis and have the same y-coordinate, as shown (c) and (d).

the homographies for calibrated cameras with the help of a given pair of projection matrices. In [46] Pollefeys et al. proposed a simple method, which only requires the oriented Fundamental Matrix, guarantees minimal image size and works for all possible camera configurations.

5.4 Correspondence Analysis

After firstly preprocessing the stereo images, meaning the Fundamental Matrix is known and with it the epipolar lines can be calculated, and secondly rectifiying images, the last step is finding correspondences between the two stereo images for obtaining depth information. As already pointed out knowing a point in one image, due to image rectification its match in the other image will now lie on the horizontal scan-line in the other image with the same y-coordinate as the known point. The question that arises is how to determine the corresponding point on this line. In [47] Lane and Thacker compared a dozen of different methods for finding correspondences. They distinguish between *area based methods* and *feature based methods*. Here, only area based approaches will be presented.

Finding correspondences is achieved by comparing similarities, therefore a method for measuring the similarity of points has to be found. One problem to deal with is that the only information given by a single point in an image is its intensity value. By only comparing point to point it would be nearly impossible to find the correct match, because with a high probability there will be more than one point in the other image with the same intensity and a decision which is the correct one can not be made due to ambiguity. The solution for this problem is to define a fixed region of a certain size around the point $\mathbf{m}_1 = [x_1, y]$ whose correspondence is wanted. A window of the same size as the respective region is shifted step by step along the epipolar line in the other image and the similarity between the two resulting regions is measured at each step. The point $\mathbf{m}_2 = [x_2, y]$ in the center of the window in the second image with the highest similarity is considered to be the correct match. Once a point is determined to be the correct match, the so called *disparity* has to be calculated. The disparity is the distance between the corresponding points in the image which is the result of putting the two stereo images on top of each other. Because of rectified images here the disparity d is defined as the difference of the x-coordinates of the two matching points

$$d = x_2 - x_1 \tag{5.21}$$

The disparity of each point is stored in a *depth map*. The next sections will explain three different methods of how similarity can be measured.

5.4.1 Sum of Absolute Differences

The first method presented is called **SAD** (**S**um of **A**bsolute **D**ifferences). As mentioned above, in order to find correspondences regions of a size $s_x{+}1{\times}s_y{+}1$ are defined around $\mathbf{m}_1$

in the first image and then compared with regions of the same size in the second image. In the following, $[x_1, y]^T$ are the coordinates of $\mathbf{m}_1$ with intensity $I_1(x_1 - s_x/2 + i, y_1 - s_y/2 + j)$ of a pixel in the fixed region in the first image at position $[x_1 - s_x/2 + i, y_1 - s_y/2 + j]$ with $0 \leq i \leq s_x$ and $0 \leq j \leq s_y$. Respectively $I_2(x_2 - s_x/2 + i, y - s_y/2 + j)$ is the intensity of the corresponding point $\mathbf{m}_2$ in the second image. Similarity is measured by calculating the difference of all corresponding pixel intensities $I_1(x_1 - s_x/2 + i, y - s_y/2 + j) - I_2(x_2 - s_x/2 + i, y - s_y/2 + j)$, and summing up the respective absolute values.

$$SAD = \sum_{j=0}^{s_y} \sum_{i=0}^{s_x} |I_1(x_1 - s_x/2 + i, y - s_y/2 + j) - I_2(x_2 - s_x/2 + i, y - s_y/2 + j)| \quad (5.22)$$

The center point of the region in the second image with the smallest SAD-value along the epipolar line is the one with the highest similarity and assumed to be the correct match. The disparity is finally calculated as in 5.21 and inserted into the depth map. The search is repeated until every pixel has been processed.

5.4.2 Sum of Squared Differences

The second method is called **SSD** (**S**um of **S**quared **D**ifferences). The procedure for SSD is the same as for SAD, except that the differences are not regarded with their absolute values, but are squared. Squaring the differences has more influence on single outlier pixels with a very high difference value.

$$SSD = \sum_{j=0}^{s_y} \sum_{i=0}^{s_x} (I_1(x_1 - s_x/2 + i, y - s_y/2 + j) - I_2(x_2 - s_x/2 + i, y - s_y/2 + j))^2 \quad (5.23)$$

5.4.3 Normalized Sample Correlation

The last way for measuring similarity presented here is called **NSC** (**N**ormalized **S**ample **C**orrelation). Let $R(\hat{x}_2, \hat{y})$ be the upper left point of the region in the second image with $\hat{x}_2 = x_2 - s_x/2$ and $\hat{y} = y - s_y/2$, and $T(\hat{x}_1, \hat{y})$ the upper left point of the fixed region in the first image with $\hat{x}_1 = x_1 - s_x/2$ respectively. In the following, for easier reading and writing $\sum_{j=0}^{s_y} \sum_{i=0}^{s_x}$ will be written as $\sum_{i,j}$, $T(\hat{x}_1 + i, \hat{y} + j)$ as $T(i, j)$ and $R(\hat{x}_2 + i, \hat{y} + j)$ as $R(i, j)$. The NSC (r) is then calculated by

$$r = \frac{p_T \sum_{i,j}(T(i,j)R(i,j)) - (\sum_{i,j} T(i,j))(\sum_{i,j} R(i,j))}{\sqrt{p_T(\sum_{i,j} T(i,j)^2)(\sum_{i,j} T(i,j))^2}\sqrt{p_T(\sum_{i,j} R(i,j)^2)(\sum_{i,j} R(i,j))^2}} \quad (5.24)$$

The scalar factor p_T is equal to the number of pixels with non-zero brightness-value in the region of the first image. The resulting value r is dimensionless and $r \leq 1$ where 1 denotes perfectly correlated, whereas a smaller r implies a bigger discrepancy of the two regions. Instead of finding the smallest value as for SAD and SSD the highest value has to be detected, the rest of the procedure does not change.

The three methods described above all result in dense depth maps, because the disparity for every pixel is calculated. Well textured images return better results than images with big homogenous areas, as the difference between the search windows is higher. Furthermore, false correspondences result in discontinuities of disparity. Concerning computation time, SAD is the fastest algorithm. It needs as many operations as SSD but instead of calculating squares it only calculates the absolute value which is not that computationally expensive. The slowest method is the NSC algorithm, which needs the highest amount of operations and the most expensive ones, like squaring and square-roots. A concrete comparison of the methods performances concerning execution time and accuracy is shown in chapter 7.

6 Driver Assistance Modules

In this chapter concrete algorithms for vision based driver assistance will be described in detail. All these algorithms have been implemented in C++ for ImprovCV, the image processing framework introduced in chapter 3. The vision information is provided by a camera mounted behind a vehicle's windshield. All algorithms are performed on 8-bit single channel grey-scale images. As all algorithms should assist the driver, they have to perform fast and therefore it is necessary to achieve low computation time for all algorithms. A concrete performance analysis for each algorithm is shown in chapter 7.

The first section (6.1) will describe a method for detecting lane-markings by applying the Probabilistic Hough Transform as shown in section 4.1.2. After that, a procedure for detecting vehicles using optical flow and shape information (as in 4.2) and tracking them in consecutive images with a correlation method will be shown (section 6.2). Finally, stereo vision algorithms with the goal of estimating distances to other vehicles will be presented in section 6.3.2.

6.1 Lane Detection

Lane detection is an essential component of many intelligent vehicle applications, including Lane Following (LF), Lane Keeping Assistance (LKA), Lane Departure Warning (LDW), lateral control, Intelligent Cruise Control (ICC), Collision Warning (CW) and eventually autonomous vehicle guidance. The lane detection procedure can approximate the position and orientation of the vehicle within the lane, and can also provide a reference system for locating other vehicles or obstacles in the path of the one viewing the scene. Therefore the detection of lanes is a reasonably important part of a driver assistance system. Lane detection is often complicated because of different road markings, clutter from other vehicles and complex shadows, lighting changes from overpasses, occlusion from vehicles, and varying road conditions.

The algorithm presented in this section uses the Probabilistic Hough Transform and a k-means clustering method as presented in 4.3.2. Furthermore, not only the information of the current image is used but as the direction and number of lanes does not change rapidly, also the information obtained from previous frames. Picture 6.1 shows the raw unprocessed image which is the starting point of the detection method.

The presented algorithm can be roughly divided into three steps. The first is preprocessing the image for eliminating useless information, the second detecting lane-markings at the current image, and the third and last is verifying the markings using information gained from previous images. In the following, "lanes" are the wanted lane-markings and "line" describes a line in general.

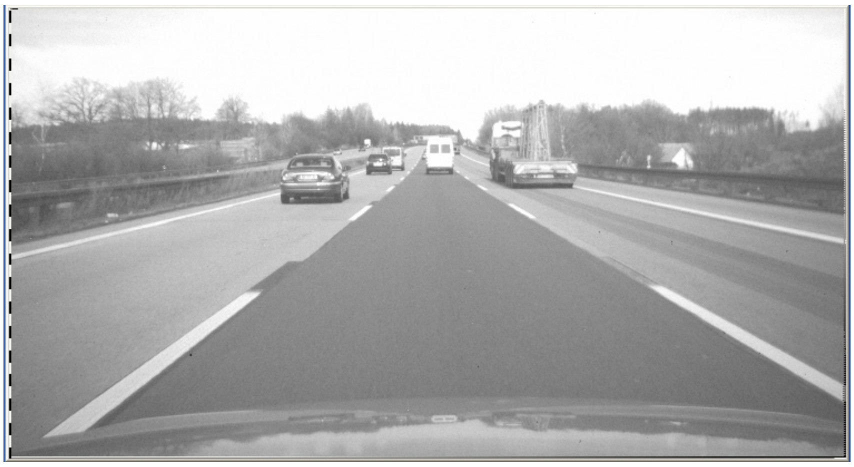


Figure 6.1: Unprocessed raw image data, taken by a camera mounted behind the car's windshield

6.1.1 Image Preprocessing

The detection of lines by the Hough Transform could be done on the raw image data like picture 6.1 which would be computationally very expensive and take long time. By preprocessing the image, non-relevant information can be eliminated. As it can be seen in 6.1 an estimated 30% of the image's upper part is useless data for lane detection because it shows the sky and therefore can be cut out of the image. Furthermore, about 10% of the lower part is covered by the cars front lid and can be clipped as well. By these simple steps the image's size is reduced by about 40% without losing necessary information for lane detection which will speed up the following procedures. Figure 6.2 shows the resulting image.

The strong intensity contrast between pavement (grey) and lane-markings (white) is another feature that can be used to reduce data. This jump from one pixel to the next is known as an *edge*. By applying an edge-detector as sobel- or canny-filter, the amount of image data is reduced significantly and useless information is filtered out, while preserving the important structural properties of an image. The edge-detection results in a binary image only containing white edge-pixels and black non-edge-pixels. Only the white ones will be used for the Hough Transform. Picture 6.3 shows the result of the canny edge-detector performed on 6.2. Instead of using the canny filter also two sobel-filters,

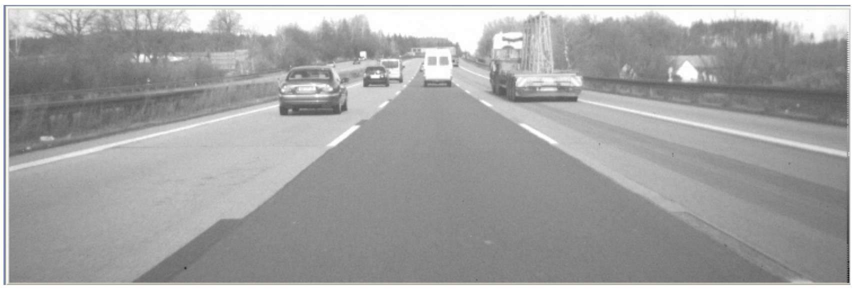


Figure 6.2: Image where upper part (the sky) and lower part (the car's front lid) have be clipped

one for each direction x and y could be applied.

Altogether, the image preprocessing reduces the amount of data, to be finally processed to a fraction of only $2-4\%$ of the original image data. The upcoming section will describe the procedure of detecting the lane-markings out of the preprocessed data.

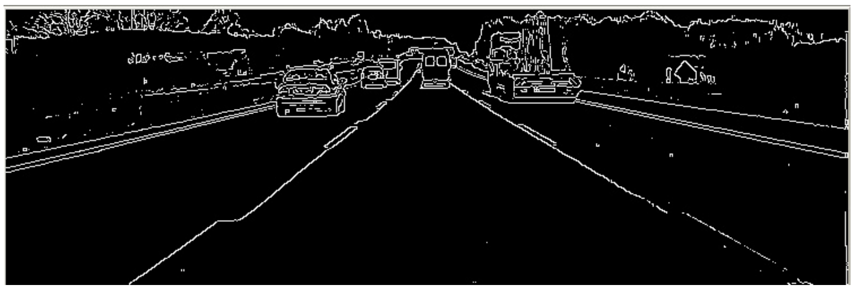


Figure 6.3: Image showing edges detected by a canny filter.

6.1.2 Lane-Marking Detection

After having preprocessed the input image the first step is to find all the lines in the image and afterwards separate them into possible lane-markings (or parts of lane-markings), called *candidates*, and arbitrary lines which are thrown away. The line detection is done by applying the Probabilistic Hough Transform presented in 4.1.2 to the image. Although the normal Hough Transform could be applied for the line detection and producing slightly more accurate results, the Probabilistic Hough Transform has been selected because of faster execution with negligible poorer results. A function for extracting Hough-lines is implemented in OpenCV. The lines resulting from the transform are shown in figure 6.4. As seen, the image contains many lines not being part of lane-markings. Thus the second step is to separate candidates from useless lines.

A good starting point for that is looking at the extreme absolute values of the lines' slopes, zero and infinity. As lane-markings are rarely horizontal or nearly horizontal in respect to the car's direction, all lines with absolute slope value smaller than a threshold θ_0 can be neglected. Although in many cases lane-markings are vertical to the car's direction they do not appear as vertical lines on the camera image, because of camera position and perspective distortion. They appear vertical, if they lie on the the vertical plane that contains the camera center and is perpendicular to the image plane. As the camera is installed in the middle of the windshield, that is only the case during lane-changing and only for a single lane-marking. This is why lines with an absolute slope value bigger than θ_∞ can also be neglected. All remaining lines are assumed to be part of lane-markings.

As the Hough Transform has been applied to an edge-filtered image, each marking has at least two lines, one for the right, the other for the left edge of the marking. Furthermore, dashed markings result in many lines that actually represent the same lane border. A way to combine these lines to a single one for each boarder has to be chosen, which leads to the third step, the clustering of candidate lines.

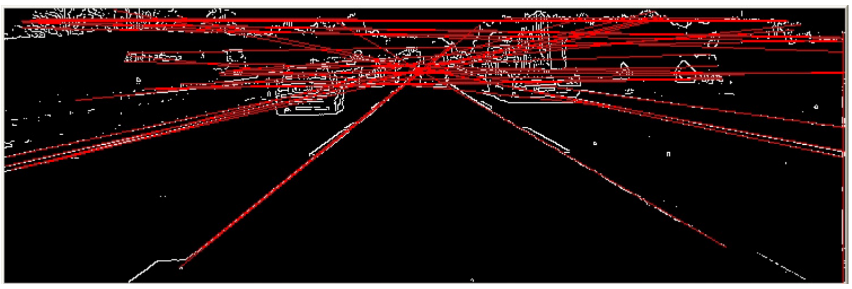


Figure 6.4: Lines detected by a Probabilistic Hough Transform.

To cluster the lines the k-means clustering algorithm introduced in section 4.3 is used. The problem with this algorithm is that the number k of clusters has to be known prior to execution. In other words, the number of lane-markings has to be known, but as they differ from road to road, and the task is to detect them, this is never the case. Therefore, a method to dynamically determine the number of needed clusters with regard to the current road scene has to be found.

The algorithm's input is an unknown number of lines, presented by slope m and intersect t. Lines are considered to be part of the same lane-marking, if their m and t do not vary too much. Thus the clustering is performed in the $m - t$ space as the lines can be represented by a single point and the fast and simple k-means point-clustering can be used. The algorithm starts by applying the k-means clustering with $k = 2$ to the unclustered data-set resulting in two new clusters. These clusters are stored in a clusterdepot $cd1$.

For each cluster of $cd1$ the variation σ of m and t is calculated by

$$\sigma = \sum_{i=1}^{n_{c_j}} \frac{(m_i - m_{avg})^2 + (t_i - t_{avg})^2}{n_{c_j}} \tag{6.1}$$

where n_{c_j} is the number of cluster points in the cluster c_j, m_{avg} the average of all m of c_j and t_{avg} the average of all t of c_j. If the variation is smaller than a threshold $\sigma < \theta_{acc}$ the cluster is stored in the final cluster depot $cd2$ and removed from $cd1$. All clusters in $cd2$ are considered to be lane-markings represented by their respective m_{avg} and t_{avg}. If the cluster's variation is bigger than $\sigma > \theta_{acc}$ it has to be decided whether the data should be completely deleted or processed again. The decision is made by comparing the variation of the cluster with an upper threshold θ_{high}. If $\sigma > \theta_{high}$ and the cluster only consists of a number of lines smaller than $\theta_{lowerNbr}$ this cluster will be removed from $cd1$. The algorithm is repeated for each remaining cluster of $cd1$. The whole procedure stops when no clusters are left in $cd1$. The flow chart representing the procedure is shown in 6.5.

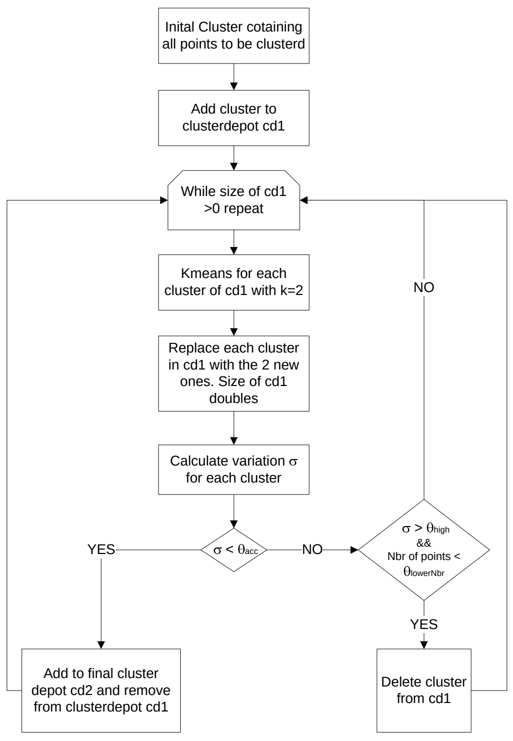

Inital Cluster cotaining all points to be clusterd
Add cluster to clusterdepot cd1
While size of cd1 >0 repeat
Kmeans for each cluster of cd1 with k=2
Replace each cluster in cd1 with the 2 new ones. Size of cd1 doubles
Calculate variation σ for each cluster
σ < θacc
YES
NO
σ > θhigh && Nbr of points < θlowerNbr
NO
YES
Add to final cluster depot cd2 and remove from clusterdepot cd1
Delete cluster from cd1

Figure 6.5: Flow chart for clustering

As mentioned before, the clusters are only assumed to be lane-markings. The next section will deal with the third and final step, the lane-marking verification.

6.1.3 Lane-Marking Verification

To verify that the detected and clustered lines are real lane-markings and not arbitrary lines, information gained from prior images is used. This results in an initialization phase after having started the algorithm where no lane-markings can be verified because of no or not enough prior information. Once having gained this information, the fact that lane-markings neither change their slope m nor their intersect t over a short sequence of images is used. A credit system is introduced to determine whether a real lane-marking has been detected or not. First detected, three credits are assigned to each detected but not yet verified lane-marking, and their parameters slope, intersect and credits are stored. In each frame all newly detected lines are compared with respect to m and t with the already stored ones. If a new line matches an old one, the credits of the old one will be increased by 1, if no match could be found the new one is stored and 3 credits are assigned. The credits of all stored lines not detected, meaning no similar line could be found again, will be decreased by one. If the credits of a stored line are equal to zero it will be deleted. A stored line with credit value bigger than 5, meaning being detected in at least four consecutive frames, is verified as a real lane-marking. The possible credit value of a lane-marking is limited to 30 which prevents lane-markings which have been detected throughout many consecutive frames from being taken as an existing one even after they have disappeared because of still high credit value. The problem of not detecting real lane-markings in some frames, because of bad edges, bad light conditions or occlusion is also solved by the credit system for already detected lane-markings as they will assumed to be present even without detecting them because of having a high enough credit value.

The final lane-marking detection result can be seen in figure 6.6. With these lane-markings the real lanes are simply considered to lie between two consecutive lane-markings starting from the outer left detected one and going to the right or vice versa.

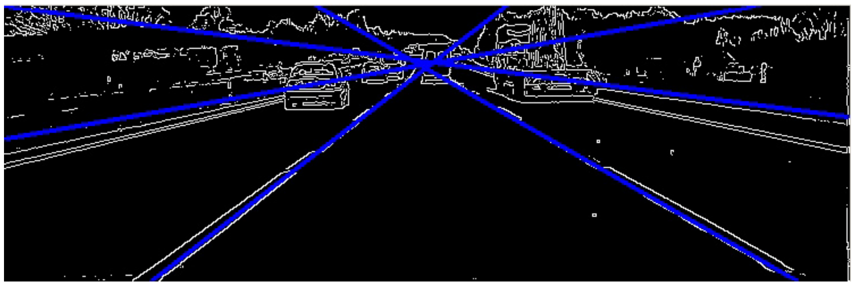


Figure 6.6: Lane-marking detection result

6.2 Car Detection and Tracking

Vehicle accident statistics disclose that the main threats a driver is facing are from other vehicles driving on the same road. Consequently, developing on-board vision-based automotive driver assistance systems aiming to alert a driver about driving environments and possible collisions with other vehicles has attracted a lot of attention. In these systems, robust and reliable vehicle detection is the first step.

This section presents a car detection system based on optical flow and shape attributes of cars. The cars, once detected, will be followed in consecutive images by a tracking system using a template matching technique. The procedure is done in four steps

1. Optical flow estimation
2. Optical flow clustering, resulting clusters are candidate cars
3. Verification of car candidates by examining their shape and a correlation with predefined templates
4. Tracking of cars with templates gained from the previous step

In the following, the points mentioned above will be described in detail.

6.2.1 Optical Flow Estimation

A possible way of estimating the presence of moving obstacles in general is using motion information produced by them. This information can be extracted from consecutive images by estimating the optical flow. This can be done in different ways as presented in section 4.2. Here, the method of Lucas and Kanade described in chapter 4.2.1 in combination with image pyramids is used. The advantage of this pyramid algorithm over Horn and Schunk and the standard Lucas and Kanade is the much lower execution time resulting from gaining only a sparse flow vector field. This sparse field is not a real disadvantage as, depending on the algorithms initialization, the resulting flow vectors concur with the position of the wanted cars. The algorithm used for calculating the flow vectors is already implemented in OpenCV.

As mentioned before, the result of the algorithm highly depends on how it is initialized, because for a certain chosen number of points in image I_{t-1} that have to be selected prior to execution it finds the matching points in image I_t and the flow vector is represented by the corresponding points. Good points to chose are points with high eigenvalues like edges or corners as these points are easy to distiguish and therefore easy to detect. Cars usually produce very strong edges, thus they are good features and suit this method for calculating the flow vectors.

A flow vector in I_{t-1} is represented by the x/y-position $p(x_{t-1}, y_{t-1})$ of its corresponding point in I_{t-1}, and the angle α and magnitude mag between this point and its match point $p(x_t, y_t)$ in I_t. Angle and magnitude are calculated by[1)]

$$\begin{aligned} \alpha &= atan2(y_t - y_{t-1}, x_t - x_{t-1}) \\ mag &= \sqrt{(x_t - x_{t-1})^2 + (y_t - y_{t-1})^2} \end{aligned} \tag{6.3}$$

In figure 6.7 an example image for flow vectors detected with this method is shown. It

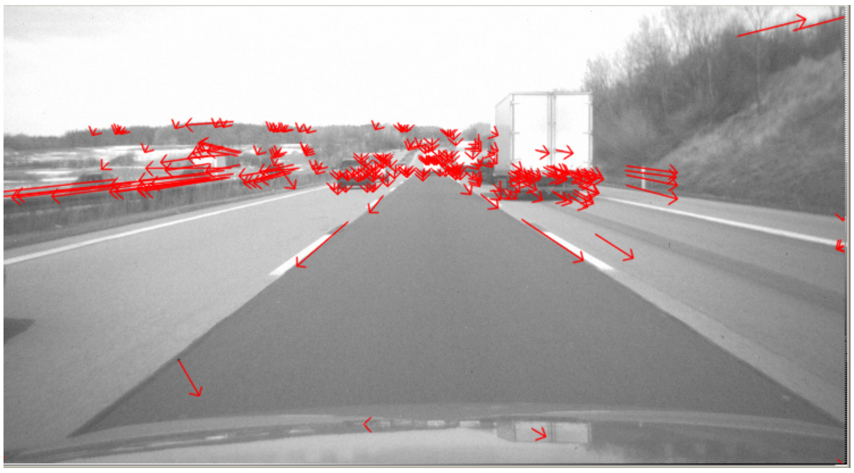


Figure 6.7: Flow vectors detected with Lucas and Kanade and pyramid images.

can be seen, that every car in the image contains a number of flow vectors, but due to the fact that the images are taken out of a moving car, not-self-moving objects like lane-markings, guardrails or side posts also have flow vectors. A decision has to be made which flow vectors actually belong to a car or other moving object to get an estimation of the number of cars and their positions in the currently viewed scene. Furthermore, the vectors of none-moving objects should be eliminated. A solution for these problems will be presented in the upcoming section.

6.2.2 Optical Flow Clustering

All vectors arising from motion of the same car occur in a small region and have similar value of magnitude. The absolute value of the angle is similar too, but depending on

1) $atan2$ is defined using the standard arctan function, whose range is $(-\pi/2, \pi/2)$, as follows:

$$atan2(y,x) = \begin{cases} \arctan(\frac{y}{x}) & x > 0 \\ \pi + \arctan(\frac{y}{x}) & y \geq 0, x < 0 \\ -\pi + \arctan(\frac{y}{x}) & y < 0, x < 0 \\ \frac{\pi}{2} & y > 0, x = 0 \\ -\frac{\pi}{2} & y < 0, x = 0 \\ \text{undefined} & y = 0, x = 0 \end{cases} \tag{6.2}$$

the object's motion the angle's sign can be different. These facts can be used to group different vectors and using their resulting mean x/y-coordinates as an initial guess for the presence of a car at this position. The grouping is done in two stages and again, as for the lane-marking detection, the k-means clustering algorithm is used. Before starting the two-stage clustering, flow vectors with a very large magnitude can be deleted as they usually occur due to false similarity detection and thus are no real flow vectors. By limiting the magnitude, flow vectors of static objects near to the camera will also be deleted as their magnitude is large as well.

The first stage performs a k-means clustering in the $magnitude/angle$-space on all flow vectors remaining after the limitation of magnitudes. As the number k for the clustering again is not known, the procedure as shown in picture 6.5 and explained in section 6.1.2 is used, but performed in different spaces. This first grouping is done, because as already mentioned above, flow vectors from the same object have similar magnitude and angle. Thus a grouping concerning these two attributes is a reasonable but not sufficient step for object position estimation. A second grouping on each cluster obtained from the first stage now performed in the x/y-space is required as objects at different positions can have similar flow vectors with respect to angle and magnitude and therefore can not be distinguished by only concerning these two attributes. The same kind of clustering procedure as in the first stage done in x/y-space plus an additional criteria is applied. This criteria helps to find optimal clusters concerning the cluster's variation in position of the flow vectors. The variation of C_1 is compared with the respective variations of the two clusters C_{11} and C_{12} created with k-means out of cluster C_1. If the mean variation of C_{11} and C_{12} is bigger than C_1's, and the variation of C_1 is smaller than a threshold θ_{acc2}, while at the same time the number of points of C_1 lies in the interval $I_{acc} = [low_{acc}, high_{acc}]$, C_1 is stored as a final cluster, otherwise C_{11} and C_{12} and clusters to be further processed. The final clusters are represented by the mean of magnitude, angle and position of all flow vectors assigned to them.

After having clustered the flow vectors, a similar credit system as presented for the lane-marking detection is applied to the clusters. Similarity between old stored clusters and new detected ones needed for the credit system is measured by comparing the above mentioned parameters. As the position of the cars can change more rapidly than the lane-marking's, the maximum credit value is limited to 10 and comparison between new and already stored clusters is done regarding position, angle and magnitude. All other steps of applying credits and removing clusters is done exactly as presented in section 6.1.3. An example result for the clustering of flow vectors is shown in figure 6.8 where a dot is drawn at the position of each cluster. The resulting clusters are the car candidates for the methods described in the next section.

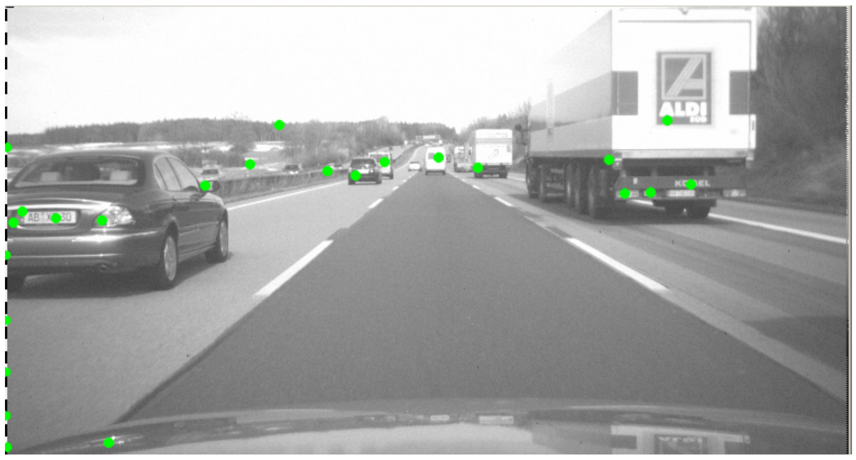


Figure 6.8: Resulting position of clustered flow vectors represented by dots.

6.2.3 Car Verification

As picture 6.8 shows, the clustered flow points are a good estimation for the presence and position of cars. However, the clusters not only coincide with cars and sometimes there exist more than one cluster for each car. Furthermore, only a x/y-position, which is not necessarily the cars' rearview center point, is known and no information about the size of the cars exist. Thus the following two problems have to be solved

1. Separating cars from arbitrary objects.
2. Obtaining size information and the rearview center point.

The chosen solution is comparable to the method presented by Betke et. al in [48] who use shape information and a template matching technique.

As cars' rearviews in an image appear as rectangular objects with a certain ratio, the first step for the verification of car candidates is to look for rectangular objects in the area around the clustered flow vectors, which is done by evaluating horizontal and vertical edges. Therefore, a median filter is applied to the image to eliminate unwanted noise followed by two sobel-filters extracting the vertical and horizontal edges which are stored separately. As the result of a sobel-filter is the gradient of the image intensity at each point and here only the appearance of strong edges is needed and not the gradient value, a threshold is applied to each sobel-filtered image to achieve binary images. Since the horizontal edges are normally more pronounced (higher gradient value) than the vertical ones, different thresholds are recommended otherwise the vertical ones would be "thresholded away". The resulting binary images are called **IV** for the vertical edges and **IH** for the horizontal edges respectively. For evaluating the presence of rectangular objects and calculating their size and position, the horizontal projection vector **w** and vertical projection vector **v** are used.

Let $\mathbf{H}$ be any rectangular search region in $\mathbf{IH}$ with size $m \times n$ and $\mathbf{V}$ any rectangular search region of $\mathbf{IV}$ of the same size as $\mathbf{H}$ and the same coordinates, then $\mathbf{v}$ and $\mathbf{w}$ are calculated by:

$$\begin{aligned} \mathbf{v} = (v_1, \ldots, v_n) &= (\sum_{i=1}^{m} \mathbf{H}(x_i, y_1), \ldots, \sum_{i=1}^{m} \mathbf{H}(x_i, y_n)) \\ \mathbf{w} = (w_1, \ldots, w_m) &= (\sum_{i=1}^{n} \mathbf{V}(x_1, y_i), \ldots, \sum_{i=1}^{n} \mathbf{V}(x_m, y_i)) \end{aligned} \tag{6.4}$$

A large projection value of vector $\mathbf{v}$ at position j indicates pronounced horizontal edges along $\mathbf{H}(x, y_j)$. An example image showing a horizontal edge map and the resulting vertical projection vector can be seen in figure 6.9. A large projection value of vector

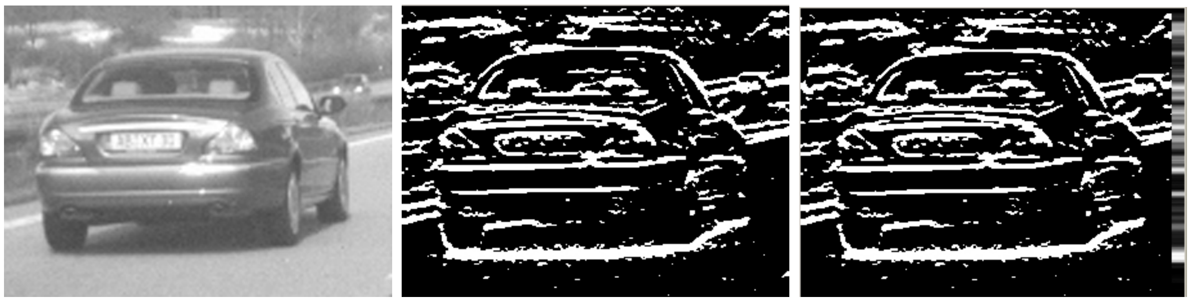

(a) (b) (c)

Figure 6.9: Image (a) shows the image to be processed. (b) shows the resulting horizontal edge map $\mathbf{H}$ of image (a) created by applying a sobel edge detector for horizontal edges followed by a threshold. In (c) the resulting vertical projection vector $\mathbf{v}$ is shown on the right side

$\mathbf{w}$ at position j indicates pronounced vertical edges along $\mathbf{V}(x_j, y)$. An example image showing a vertical edge map and the resulting horizontal projection vector can be seen in figure 6.10. A projection value is assumed to be large, if it is bigger than half the value of the largest projection coefficient of its respective projection vector. For $\mathbf{v}$ and $\mathbf{w}$ the thresholds $\theta_{\mathbf{v}}$ and $\theta_{\mathbf{w}}$ are formulated as

$$\begin{aligned} \theta_{\mathbf{v}} &= 0.5 \max\{v_i | 1 \leq i \leq n\} \\ \theta_{\mathbf{w}} &= 0.5 \max\{w_i | 1 \leq i \leq m\} \end{aligned} \tag{6.5}$$

Searching for the rectangle's sides is performed as follows. The horizontal projection vector is searched, starting from the beginning until an entry is detected that lies above $\theta_{\mathbf{w}}$. The entry's position or index is assumed to be the x-coordinate of the left side x_L of the potential object. The x-coordinate of the right side x_R is detected by searching $\mathbf{w}$ from the other direction starting from the end until an entry is detected that lies above $\theta_{\mathbf{w}}$ and assigning its index to x_R. The same search is performed on $\mathbf{v}$ to detect the top and bottom sides' y-coordinates of the object. The bottom coordinate y_B is found by

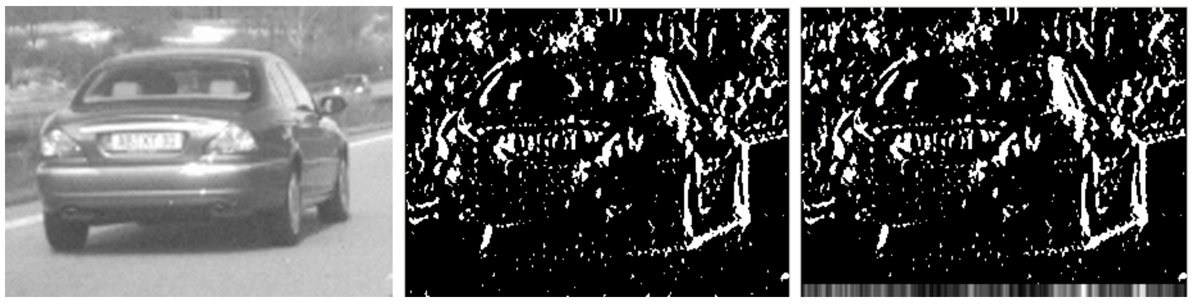

(a) (b) (c)

Figure 6.10: Image (a) shows the image to be processed. (b) shows the resulting vertical edge map $\mathbf{V}$ of image (a) created by applying a sobel edge detector for vertical edges followed by a threshold. In (c) the resulting horizontal projection vector $\mathbf{w}$ is shown on bottom side.

starting at the beginning of $\mathbf{v}$ until an entry is found, bigger than $\theta_{\mathbf{v}}$ and the top y_T by starting from the end of $\mathbf{v}$. The wanted rectangle, representing a potential object, can then be described by its upper-left-coordinate $\mathbf{ul} = [x_L, y_T]^T$ and lower-right-coordinate $\mathbf{lr} = [x_R, y_B]^T$.

In the presented case of car detection, the search regions $\mathbf{H}$ and $\mathbf{V}$ are placed in the way that their center points coincide with the position of the car candidate $cand_i$ to be evaluated. Their width w_{search} and height h_{search} are initially chosen to $w_{search} = hw_{start}$ and $h_{search} = hw_{start}$ meaning the search regions are square. The rectangular search is performed in this regions, and an object is considered a potential car if the aspect ratio $ar = (x_R - x_L)/(y_T - y_B)$ of the detected rectangle's width and height lies between 0.7 and 1.3.

Unfortunately rectangular objects are not necessarily cars but can also be objects like traffic signs or buildings or even only a cluttered region containing strong edges is assumed to be a rectangular object. This is the reason why the information of shape alone is not sufficient for the detection of cars. To reduce the number of false positives, the detected rectangular region is correlated with a car template. The car template is chosen from a set of existing images, showing the rear of cars, by comparing the mean grey value of the rectangular region with the mean grey value of each template. The template with the smallest difference in mean grey value is selected. Depending on the similarity of template and the detected region, a car is recognized or rejected.

The similarity is measured by calculating the normalized sample correlation coefficient r as presented in section 5.4.3. If r is higher than 0.3 the rectangular region is assumed to be a car, the size and position of the rectangle is stored and all candidate cars lying in the rectangle's region are deleted, else the search for a rectangle is repeated with a bigger size for the search region $h_{search} = w_{search} = w_{search} + \delta$ but around the same point. The procedure is repeated until either a car is detected or the search region's size exceeds a limit $w_{search} = h_{search} > sz_{thresh}$ without detecting a car. When all car candidates have been processed in this way, the complete search is finished. The whole algorithm can be summarized in the following seven steps and will be called *car-detector* in the following:

1. Select a candidate car from the clustered flow vectors, called $cand_i$. If no candidate left, stop process, else set initial search window size $w_{search} = h_{search} = hw_{start}$
2. Extract square region $\mathbf{H}$, the horizontal edge map from $\mathbf{IH}$ and $\mathbf{V}$, the vertical edge map from $\mathbf{IV}$, with width w_{search}, height h_{search} and center point obtained from $cand_i$
3. Calculate the projection vectors $\mathbf{v}$ and $\mathbf{w}$ for $\mathbf{H}$ and $\mathbf{V}$ plus the respective thresholds $\theta_{\mathbf{v}}$ and $\theta_{\mathbf{w}}$
4. Search along $\mathbf{v}$ and $\mathbf{w}$ for the left x_L, right x_R, top x_T and bottom x_T sides
5. Calculate aspect ratio $ar = (x_R - x_L)/(y_T - y_B)$ of detected rectangle, if $0.7 \leq ar \leq 1.3$ go to next step, else go to step 7
6. Select appropriate car template and calculate r of detected rectangular region and car template. If $r > 0.4$ store position and size of rectangle as parameters of detected car, delete all car candidates lying in the rectangle and go to step 1, else continue with the next step
7. Set search window size to $w_{search} = h_{search} = w_{search} + \delta$,
   if $w_{search} = h_{search} > sz_{thresh}$. Delete $cand_i$ and go to step one, else go to step 2

6.2.4 Car Tracking

The previous section described a method to detect cars by evaluating shape information in a region around an estimated position obtained from the optical flow clustering in a single frame and the region's similarity to a template image. However, the immediate recognition of a car from one frame is very difficult and only works in presence of enough brightness-contrast between vehicles and background as strong edges are required. Therefore a tracking system is introduced to redetect cars recognized in previous frames and to update their positions and size information. Furthermore, the tracking process is used to filter out remaining false positives.

For each car detected by the car-detector a separate tracking process is created which initially stores a tracking-window represented by the parameters size and position of the detected rectangle. A template containing the part of the image that coincides with the detected rectangular region, in simpler words that shows the detected car, is stored as well. Furthermore, bonus and malus points are assigned during the tracking process. The malus points are initially set to 0 whereas the initial bonus points depend on the aspect ratio, the NSC-coefficient and the number of clustered flow vectors lying in the region of the tracking-window. The nearer the aspect ratio ar is to 1, the more credits will be assigned. The same applies for the NSC-coefficient r. More flow vectors in the detected region denote a higher probability that a car is detected and therefore more credits are added to the bonus points. If the accumulated bonus value is above a threshold and higher than the malus value, it is decided that the tracked object is a car. For visualization a rectangle will be drawn into the current frame by the tracking process, showing the tracking-window. In each frame a refined search within the tracking-window is done by first evaluating the edge maps to provide a new estimation of the outlines of the potential car followed by correlating the stored template with the region obtained from the refined search.

Due to low contrast between car and environment resulting in weak edges, or up and down movement of the camera viewing the scene because of uneven pavement or large movements of the tracked car, the aspect ratio can be out of the wanted range or not all outlines can be detected. To deal with these problems, the tracking-window's position has to be adjusted. If the aspect ratio is too high, meaning the detected width is much larger than the height, a search in the vertical edge map is performed. The number of edge-pixels in **V** lying in the region between the bottom border of the tracking-window and the detected bottom border of the car are summed up and compared with the sum of the pixel lying between the detected bottom border and top border of the tracking-window. If the lower sum is significantly higher, the tracking-window is shifted towards the bottom until the detected top coincides with the old tracking-window's top. Otherwise it is shifted towards the top until the detected bottom coincides with the old tracking-window's bottom. Picture 6.11 shows a window shifted downwards.

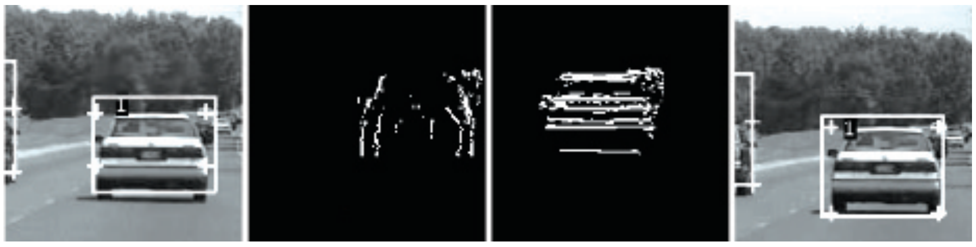


Figure 6.11: Pictures showing a performed downshift [48]

If the aspect ratio is too low, meaning much larger detected height than width, a similar procedure is performed on the horizontal edge map. The sum of edge-pixels in **H** lying between the left border of the tracking-window and the left detected border is compared with the sum of all pixels between left detected border and right tracking-window border. The tracking-window is shifted to the right if the sum on the right is higher than the one on the left, otherwise it is shifted to the left. An example image showing the left shift procedure is shown in figure 6.12.

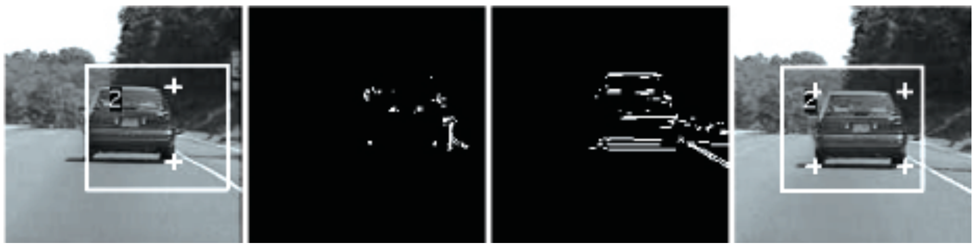


Figure 6.12: In the first picture, due to bad contrast the vertical edge map (second picture) shows weak edges. Because of this, the left and right side of the car are found in the same position. The horizontal edge map shown in the third picture shows significant horizontal edges to the left of the detected side. Thus, the tracking-window is shifted to the left, shown in the fourth image.

The two searches and shifts are also performed if the aspect ratio is correct, but the new detected width and hight are implausibly smaller than the old ones. After having shifted the tracking-window, the new size and position of the detected car are stored and the part of the image lying in the new tracking-window is correlated with the stored template. As the stored template should exactly represent the redetected region, the threshold for the calculated NSC-coefficient to accept that the redetected region is a car, is double the threshold for the initial detection, $r > 0.6$. If the car is redetected meaning $r > 0.6$ and $0.7 < ar < 1.3$, credits are assigned to the bonus account depending on the values of r and ar. If not, the tracking-window's size is expanded only once more and the whole process described in this section is repeated. If still no car can be detected, a malus value is assigned, also depending on r and ar. The farther these values are away from 1, the higher are the assigned malus points. If the number of malus points is higher than the number of bonus points, the tracking process is deleted and no car is assumed at this position anymore. The process is also deleted if for a number of consecutive frames not enough significant edges can be detected within the tracking-window, although the bonus rate may still be higher than the malus rate. This ensures that a process tracking a car does not drift away and starts tracking something else. After having tracked a car for 10 consecutive frames, a new template will be created by storing the currently detected region, but only if for this frame $r < 0.7$ is valid, in order to not delete a still very good working template.

To estimate the direction the tracked car is moving in, the tracking process stores the parameters of its tracking-window from five consecutive frames. By calculating the mean of the last five x-coordinates and comparing it with the current one, an estimation whether the tracked car moves to the left, right or just goes straight ahead with reference to the car viewing the scene is made. The tracked car's movement in x-direction $xdir$ is calculated by

$$xdir = x_t - (\sum_{i=1}^{5} x_{t-i})/5 \tag{6.6}$$

where x_t denotes the current x-coordinate of the tracking-window's center point, and x_{t-i}

the x-coordinate obtained from the frame i-frames before. A large positive value of $xdir$ means the tracked car moves to the right, a large negative one it moves to the left. The higher the absolute value of $xdir$ the larger is the tracked car's movement. An arrow to visualize the tracked car's moving direction is drawn into the tracking-window. The longer the arrow, the stronger the movement.

To get a rough estimation of the tracked car's movement in z-direction, meaning whether the car comes nearer to the car viewing the scene or goes farther away, the tracking-window's size is taken into concern. The tracked car's movement into z-direction $zdir$ is estimated by

$$zdir = area_t - (\sum_{i=1}^{5} area_{t-i})/5 \tag{6.7}$$

where $area_t = width_t * height_t$ meaning the surface area of the tracking-window at the current frame and $area_{t-i}$ the tracking-window's surface area i-frames before. If $zdir$ is positive, meaning the tracking-window is getting larger, the tracked car is coming nearer to the car viewing the scene whereas a negative value of $zdir$ signifies that the tracked car is going farther away. As a car coming nearer to the one viewing the scene poses a certain threat, the tracking-window is marked red, otherwise it is drawn green. An example result of the whole car-detection-tracking-process is shown in figure 6.13.

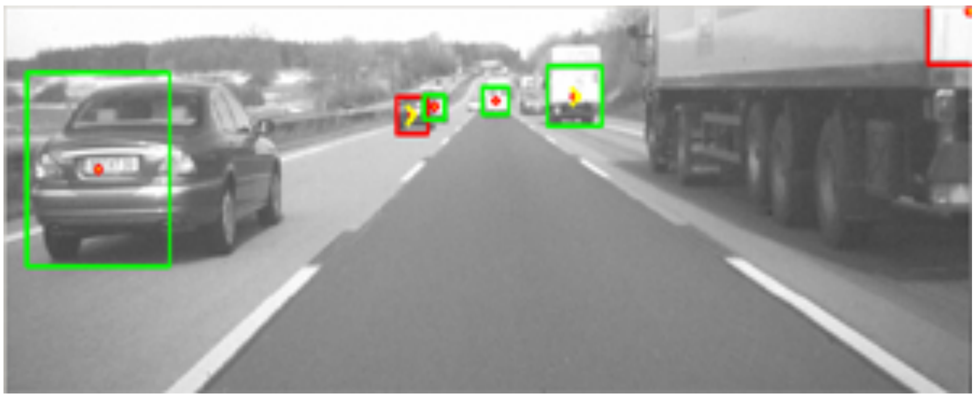


Figure 6.13: Result of the presented car-detection-tracking-process

6.3 Distance Estimation by Stereo Vision

A car-detection and -tracking algorithm has been presented in the previous section. A method for estimating the movement of tracked cars in z-direction by evaluating the change of the detected car's size has been recommended. However, this is only a very rough approximation and does not yield an estimation of distance in meters, but only a tendency whether the tracked car comes closer or goes farther away. Furthermore, the tracking-window's size can differ a lot, not only from the car's real movement but also due to false edge detection. As the distance information is very important for driver assistance, for example to warn the driver of possible collision, a second camera is used. By applying stereo algorithms, a better estimation of the tracked car's movement in z-direction and an estimation of the distance in meter can be obtained. The cars detected with the algorithm presented above are tried to be detected in the image of the second camera with stereo matching techniques. By calculating the disparities between the tracked cars in the first image and their matches in the second image, the distance between the camera viewing the scene and the tracked cars can be estimated.

In order to enhance the stereo matching, the epipolar geometry as presented in **??** is used. The methods presented in the current section deal with a set of uncalibrated cameras and therefore an unknown Fundamental Matrix $\mathbf{F}$. Thus the Fundamental Matrix between the two cameras has to be calculated at first, as it is needed for the epipolar geometry. Once the Fundamental Matrix is obtained, the camera images will be rectified and the epipolar geometry can be used to simplify and speed up the correspondence search.

6.3.1 Calculation of Fundamental Matrix and Image Rectification

Having a set of two cameras, the easiest way to calculate their Fundamental Matrix is by knowing their intrinsic camera parameters meaning to have calibrated cameras. For non-calibrated cameras, a robust and correct calculation of the Fundamental Matrix is much more difficult and challenging. Here, a modified version of the RANSAC-algorithm as described in **??** is applied.

To use the RANSAC-algorithm, or any other algorithm for calculating $\mathbf{F}$, the first step is to extract a set of points from image one and their corresponding points in image two. The more accurate the corresponding points are, the better the final Fundamental Matrix will be. Again, as for the calculation of the optical flow, points with high eigenvalues are chosen from image one as they are assumed to be detected more easily in the second image. For calculating their corresponding points, the pyramid implementation of the Lukas and Kanade algorithm as used for calculating the optical flow in section 6.2.1 is chosen. The number of points whose correspondences are searched should be very large (> 5000), as the probability of having correct matching pairs and the number of them rises with the number of points. Furthermore, as points are extracted only by taking into concern their eigenvalue and not looking at their actual position in the image, a high number of points have to be extracted to not only get points in the first image that are not even part of the second image due to the different position of the second camera.

After having extracted the corresponding points, RANSAC is applied and a first Fundamental Matrix $\mathbf{F}_1$ is calculated. The extracted points are then divided into outliers and inliers by calculating the distance between their corresponding point and their respective epipolar line calculated with $\mathbf{F}_1$. A distance larger than one pixel labels an outlier, otherwise the point is an inlier. Picture 6.15 shows an example stereo pair, where inliers are marked green and outliers marked red. The inliers and $\mathbf{F}_1$ are stored, as well as the percentage of inliers that $\mathbf{F}_1$ created, called μ_1. The value of μ_1 is a kind of quality measurement for $\mathbf{F}_1$, the more inliers, the higher μ_1 and the better $\mathbf{F}_1$. The stored matrix is called $\mathbf{F}_{final}$ with its respective μ_{final} in the following.

In the next step, after the cameras have grabbed a new pair of images, again corresponding points will be extracted in the same way as described above and a new Fundamental Matrix $\mathbf{F}_2$ will be calculated. Instead of only using the new extracted points for the calculation of the Fundamental Matrix, the points stored in the previous steps will be used too, as they are assumed to be good pairs, which raise the probability of obtaining the correct Fundamental Matrix. Image 6.14 shows a pair of images taken by a stereo camera set, where the points used for RANSAC are marked with a cross. Already stored points, detected again by the correspondence search, are only used once in the same Fundamental Matrix calculation. All points used for calculating $\mathbf{F}_2$ are again divided into in- and outliers and the percentage of inliers μ_2 is calculated. Newly detected inliers are added to the ones already stored. If μ_2 is larger than μ_{final}, $\mathbf{F}_{final} = \mathbf{F}_2$ and $\mu_{final} = \mu_2$. Otherwise $\mathbf{F}_2$ will not be stored and $\mathbf{F}_{final}$ still is assumed to be the correct Fundamental Matrix. For each new pair of images, this procedure is repeated until for 30 consecutive frames the Fundamental Matrix $\mathbf{F}_{final}$ has not changed, in other words it has not improved anymore, and the final Fundamental Matrix is found.

After having obtained the Fundamental Matrix for the current camera set, its images have to be rectified. The first step for the method used here is to find all scanlines in the images. A scanline contains all points lying on the same epipolar line, calculated with the Fundamental Matrix. Once all scanlines are obtained for both images, the images are rectified by plotting for both images all points of their respective scanline i into line i in other words to make the scanlines horizontal. The rectified images' heights are equal to the number of scanlines and the widths are equal to the longest scanline. Figure 6.16 shows a set of rectified stereo images using the presented methods.

After having rectified images, the correspondence search followed by the estimation of distance can be done and will be described in the next section.

6.3.2 Stereo Matching and Distance Estimation

As explained in section 5.3 once the images are rectified the correspondence search for a known point in image one $\mathbf{p}_1 = [x_1, y_1]^T$ is performed on a horizontal line in image two with the same y-coordinate as $\mathbf{p}_1$. This raises the accuracy of detecting the correct match while at the same time reduces the calculation time enormously. For giving an example of how much computation time is saved, the correspondence search is compared, once using the epipolar constraint and once not, by calculating the total amount of operations

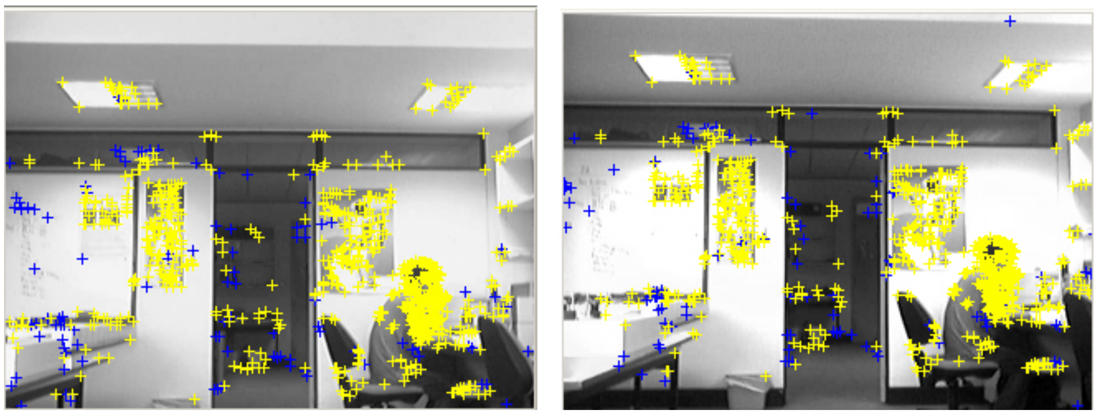


Figure 6.14: Example set of stereo images taken in the mobile robot lab at UWA with a set of uncalibrated webcams. The crosses mark corresponding points used for RANSAC, blue crosses are points detected in previous frames whereas the yellow ones are detected points from the current frames.

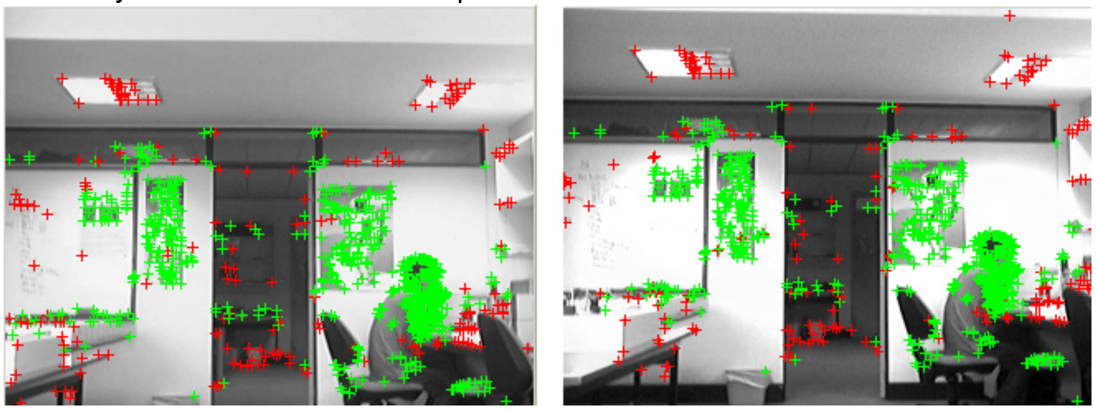


Figure 6.15: Same set of images as in 6.14; here, the green crosses mark points that have been declared to be inliers, red crosses are outliers.

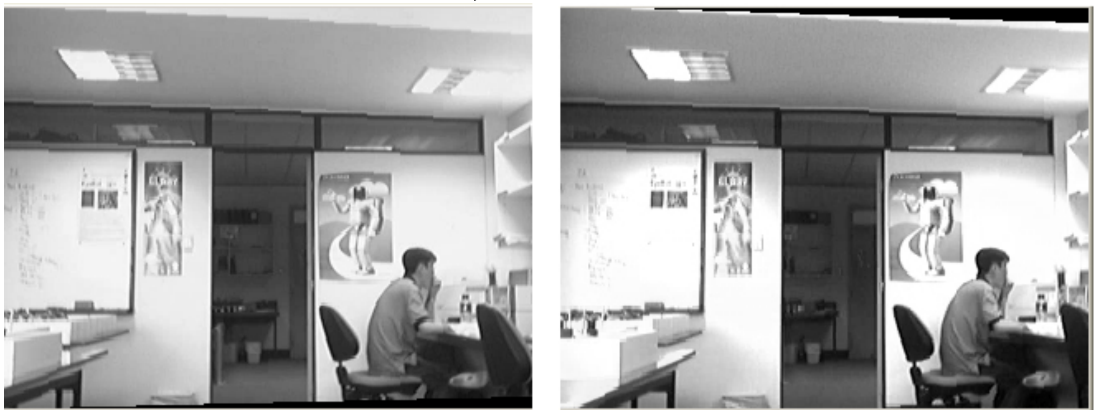


Figure 6.16: Rectification result of 6.14 with Fundamental Matrix calculated with RANSAC and the points from 6.14.

needed without concerning their respective execution time.

For measuring the similarity in this example, a search window of size $n \times n$ with uneven n and SAD as described in section 5.4.1 is used. SAD needs n^2 subtractions, n^2 absolute value operations and $n^2 - 1$ additions, which yields in total $3n^2 - 1$ operations. The images are of size $w \times h$. The total amount of points for which the similarity can be measured is $(w-(n-1)) \cdot (h-(n-1))$ as boarder points have to be cut out because the search window does not fit for them. The total amount of operations for searching the whole image is $(w-(n-1))^2 \cdot (h-(n-1)^2 \cdot (3n^2-1))$ and the number of operations by taking into concern the epipolar constraint is $(w-(n-1)) \cdot (h-(n-1)^2 \cdot (3n^2-1))$. Table 6.1 shows a comparison of the number of operations for different search window sizes on an image of size 320×240.

window size n	3	5	7	9
number of operations full search	$1.48e^{11}$	$4.11e^{11}$	$7.88e^{11}$	$1.27e^{12}$
number of operations using epipolar constraint	$4.68e^{8}$	$1.30e^{9}$	$2.51e^{9}$	$4.06e^{9}$

Table 6.1: Comparison of number of operations needed to perform SAD with different search window sizes for full window search and search along rectified epipolar lines for images of size 320×240.

The tremendous reduction of operations, and therefore the justification for using the epipolar constraint can be clearly seen from table 6.1. A further reduction of operations can be achieved by not searching correspondences along the full epipolar line but only along a fraction of it, as the maximum possible displacement is limited by the camera parameters. But still calculating a complete depth map needs too many operations to be performed in real time to estimate distances to every obstacle. Thus, for the task of finding distances to other cars, only the disparities d of the points lying in the tracking-window of the tracked cars are calculated for the driving assistance system, with equation 5.21.

As all points lying in the same tracking-window are part of the same object, they should result in nearly the same distance and therefore redundant information. Considering this, it would be enough to take only the center point of the tracking-window and calculate its disparity for getting a distance estimation, but as the detection of correspondences can return wrong results more points should be used to get a robust estimation. Here points with high eigenvalues are extracted from the tracking-window and their disparity and distance are calculated. All these distances are then compared with their resulting median and distances differing to much from the median are rejected, to delete large outliers. The tracked car's distance is assumed to be the average of all remaining distances. The question that has not been answered yet is how to actually calculate the distance from a known disparity.

Image 6.17 shows a set of two axes-parallel cameras with same known focal length f and known baseline length $B = B_1 + B_2$. The calculation of the distance D of a 3D world point M that creates the two corresponding points $\mathbf{p}_1 = [x_1, y]^T$ and $\mathbf{p}_2 = [x_2, y]^T$, in the cameras c_1 and c_2 respectively, is derived by using the *similar triangles theorem* (Thales).

$$\frac{D}{B_1} = \frac{f}{x_1} \qquad \frac{D}{B_2} = \frac{f}{x_2} \tag{6.8}$$

By summing up the two equations of 6.8, and using that x_1 and x_2 lie on different sides of

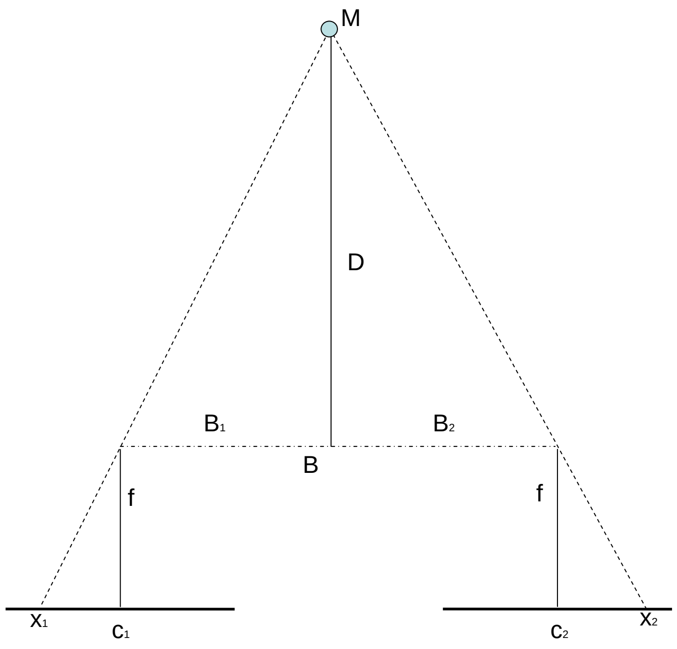


Figure 6.17: Similar triangles theorem (Thales)

their origin, meaning they have different signs, the equation for calculating the distance D can be derived and is shown in 6.9.

$$\frac{2D}{B_1 + B_2} = \frac{2f}{x_2 + (-x_1)} \Rightarrow D = f\frac{B}{x_2 - x_1} = f\frac{B}{d} \tag{6.9}$$

A disparity d of 0 means the 3D world point lies at infinity. This shows that an object at infinity appears at the same place on both image planes. Practically, the camera resolution that is the pixel width will limit the minimum measurable disparity, that is the maximum distance. The minimum measurable distance is given by the maximum disparity which is equal to the image's width. The focal length and the disparity must have the same unit, either meter or pixel are suggested. B should be given in meter so the resulting distance will have the same unit.

Equation 6.9 is only valid for axes-parallel stereo camera sets, but still can be applied for "nearly" axes-parallel cameras, meaning these cameras have only a small rotation between their optical axes, whose images are rectified. For the wanted task, the distance calculated with 6.9 is still accurate enough to get a good impression whether a tracked car moves farther away from the car viewing the scene or comes closer, even if it is not 100% correct. Thus, this method can be used for driver assistance to generate warnings of cars coming too close. The real distance can only be calculated correctly, in the case of none axes-parallel cameras, if both extrinsic and intrinsic parameters are known. The distance is then calculated by a triangulation method that will not be explained here but for further readings refer to [49].

7 Results

This chapter evaluates the results obtained by applying the algorithms presented in the previous chapters to a sample set of road scene images. A descriptive way is chosen to formulate the accuracy of these algorithms and their respective performances are measured concerning execution time. They have been tested on a standard computer with 1024MB RAM and a 1.7GHz Intel Mobile Centrino Processor running Windows XP Professional. As all algorithms have been executed on ImprovCV, there is always an offset in total execution time of about 30ms needed, caused by the framework for grabbing frames, displaying on the screen or in the preview window and transferring data between filters.

7.1 Accuracy

Accuracy of image processing algorithms is often difficult to measure and to formulate in numbers which makes it hard to compare different ideas solving the same problem. Furthermore, most existing algorithms perform better or worse under different circumstances like weather- or road-conditions. This section tries to describe the accuracy of the lane detection, vehicle detection and stereo algorithms.

7.1.1 Lane Detector

The lane detector has been tested on a number of road scenes differing in the number of lanes and the amount of traffic. It worked very well for both solid lane-markings and dashed ones. After a short initialization phase of $5 - 10$ frames, the outer highway boundaries and the lane the camera-quipped car is traveling in, have been detected in each case. Example detection results for four different road scenes are shown in figure 7.1. The graphics 7.8 (c) and (d) show the number of detected lanes in each frame of the two example videos corresponding to 7.1 (a) and 7.1 (b). 7.8 (c) quotes that most of the time five out of six lane-markings have been detected, which is good enough as both the important two outer highway boundaries and the ones of the traveled lane have always been detected. In 7.8 (c) it can be seen, that one lane-marking more than actually present has been detected in several frames, originated by a guardrail not close enough to the real outer lane-marking to be clustered. Furthermore the above mentioned initialization phase until the important markings are detected becomes apparent.

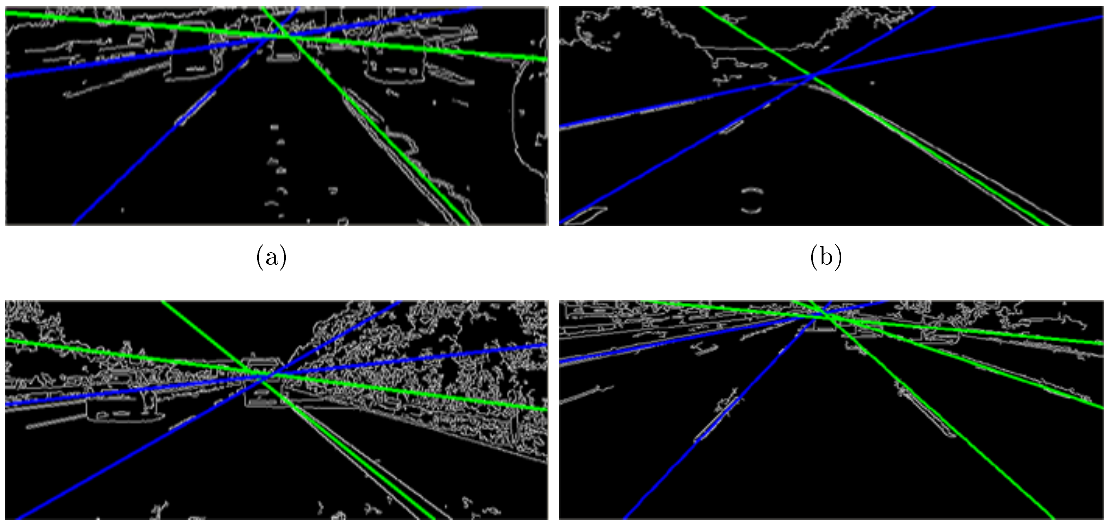

(a) (b) (c) (d)

Figure 7.1: Result of lane detection for four different road scenes

The absolute value of the slope representing the detected outer highway boundary lane-markings is often smaller than the real slope. This is due to the fact that guardrails with a smaller slope than the boundary markings are also detected as lines close to the outer boundary markings and combined to the same resulting marking. The detection algorithm combines close lines to a single one by calculating the mean line parameters of all combined lines which then results in the different slope.

For edge detection at the stage of image preprocessing as described in ??, a canny filter has been applied. Depending on the camera's quality, weather conditions or the presence of noise, the filter parameters have to be adjusted by inspection. Different filter parameters result in completely different images and effect both execution time and accuracy. The more cluttered the output the lower the accuracy and the higher the execution time. A tradeoff between noise reduction and strong edges has to be found. The same example image, canny filtered with different parameters is given in picture 7.2.

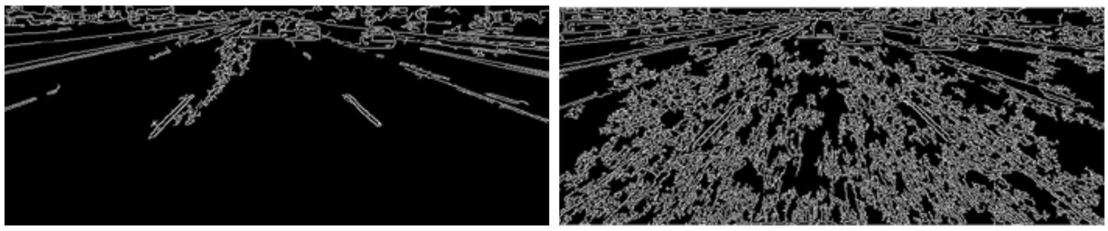
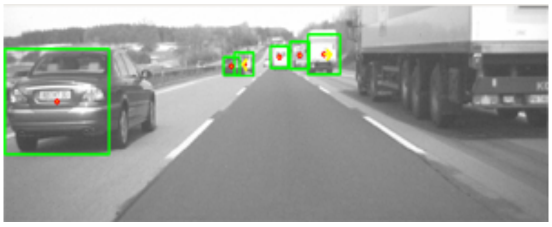

(a) Wanted result, good to process (b) Not wanted result, bad to process

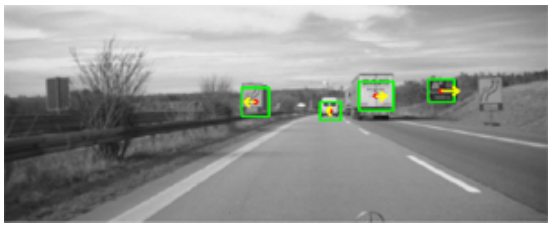
Figure 7.2: Canny filter with different parameters applied to same image

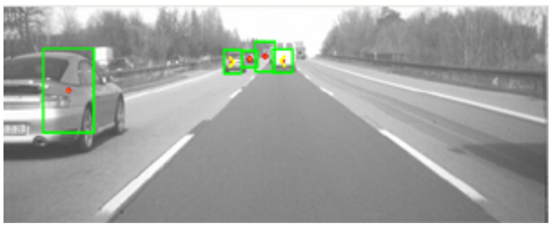
7.1.2 Vehicle Detection and Tracking

To evaluate the vehicle detection and tracking system introduced in chapter 6 images taken by a video camera from a moving car on German highways have been used. The images show very different types of cars from sports cars over normal cars to vans and trucks. The extraction of candidate cars via optical flow and the k-means cluster method works very well and yields one candidate position for nearly every present car, surprisingly even for distant cars producing only little motion. Also the rectangle search provides satisfying information, in presence of good contrast between cars and background and a good reduction of car candidates. As false positives still exist after the first two steps, a correlation technique with predefined car templates is applied, as described in 6. A good set of car templates which ideally represents all possible cars is required from which at each matching step one has to be chosen, which unfortunately is a weakness of the detection system. Furthermore, a threshold has to be defined if the resulting correlation coefficient is high enough to assume a car is detected or not. Is this threshold too low, too many false positives occur, is it too high, too many real cars are rejected. Once a car is detected, the correlation method with the respectively created template to track it in consecutive frames performs good. The presented position update concerning strong edges, further increases the tracker's robustness.

The whole system was able to detect in several test sequences up to six real cars at the same time 7.3 (a), and could even distinguish close cars which also can be seen in 7.3 (a). After an initialization period some cars could be tracked during the entire duration of the available video sequences, particularly cars driving directly in front of the one with the camera.


(a) Six cars detected at same time


(b) False detection due to traffic sign


(c) Facing sides of 2 cars detected as one

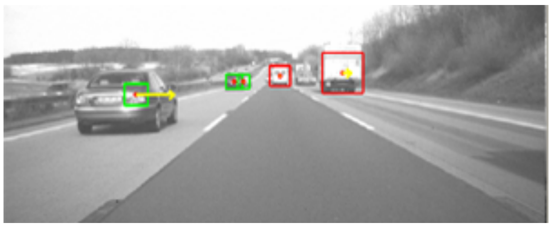

(d) Only small part of car detected

Figure 7.3: Four results of the vehicle detector and tracker

Problems of false detections mainly occur due to other rectangular objects like traffic signs (image 7.3 (b)) or buildings on the side of the road or very cluttered areas. To cope with this problem, the car-detector could be combined with the lane detector, and everything detected as a car outside the outer lane-markings could be rejected. Another kind of false detection or false positioning of the tracking-window could arise if two cars are traveling too close to each other because their flow vectors will be clustered together, and the two sides facing each other will be detected as a rectangle 7.3 (c). In some cases the tracking-window does not cover the whole car, but only parts of a it, see image 7.3 (d).

7.1.3 Stereo Procedure

In chapter 5, three different methods for analyzing stereo images in order to find correspondences have been presented. One of these with a certain search window size, performing fast and accurate enough, has to be chosen to generate the distance estimation of detected cars. To evaluate and compare the quality of the three different correspondence analysis methods SAD, SSD and NSC and their respective execution times for different search window sizes (see section 7.2), a test stereo image set from the University of Tsukuba seen in figure 7.4 has been used.

Both SAD and SSD perform comparably well concerning the quality of the resulting

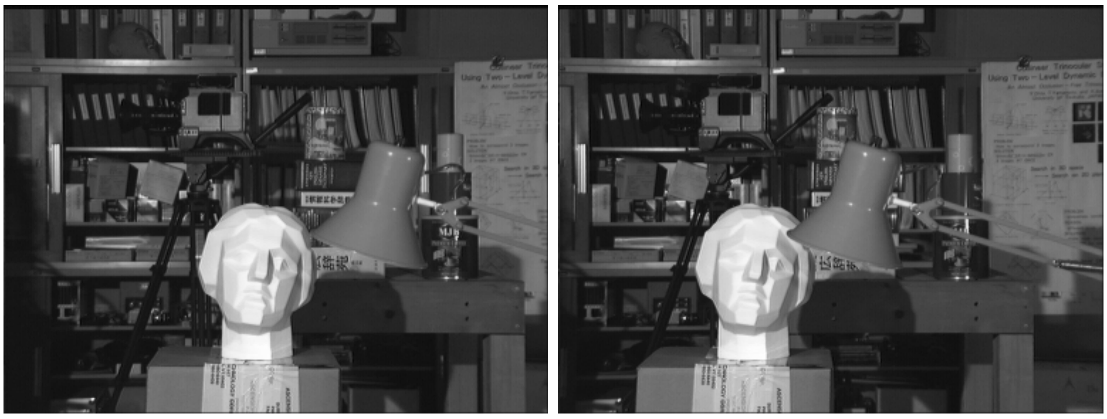

(a) Left stereo image (b) Right stereo image

Figure 7.4: Stereo test image set from the University of Tsukuba

depth map, independent from the chosen search window size, whereas NSC performs worse with lots of false detections. A comparison of the resulting depth map for the three methods with a search window size of 7×7 pixels applied on the test images is shown in picture 7.5.

The bigger the search window size is chosen, the less false detections and ambiguities occur and the smoother the resulting depth map. However, as the section about execution time will show, the needed creation time rises linear with a bigger window size. A compromise

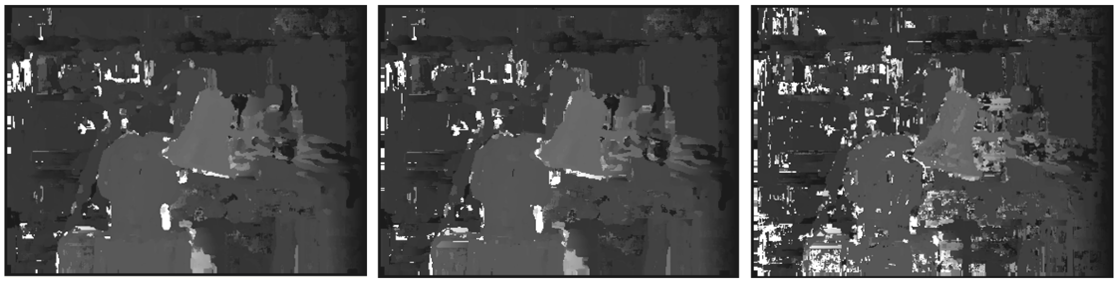

(a) SAD 7×7 (b) SSD 7×7 (c) NSC 7×7

Figure 7.5: Three depth maps of Tsukuba test image for three different methods with same window size

has to be found between high accuracy and low execution time. To visualize how depths maps change depending on the used window size created with the same analyzing method, three depths maps created with SAD and different window sizes are shown in image 7.6. It can be seen, that a larger window size yields a smoother depth map (not so many white dots), but it also smears out edges where neighbor pixels would have a large difference of disparity.

(a) Window size 3×3 (b) Window size 5×5 (c) Window size 9×9

Figure 7.6: Three depth maps of Tsukuba test image for SAD with different sized search windows

As a window size of 7×7 combined with SAD already results in a good and accurate enough depths map, it is chosen for the correspondence analysis to estimate distances to detected cars. The distance estimation by using the known distance between the two stereo cameras, their focal length and the calculated disparity of points representing detected cars worked well for the used test sequences. As no groundtruth data was available, the obtained distance information could not be compared to the real data but at least cars that are far away result in bigger distance then near cars and the results appear to be quite accurate as figure 7.7 shows.

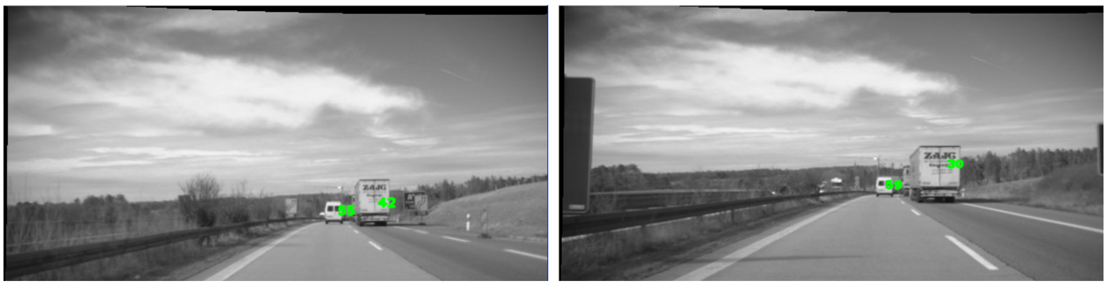

(a) (b)

Figure 7.7: Two example pictures for estimated distances to cars, plotted on the respective cars rear view.

7.2 Execution Time

Apart from accuracy, execution time is a very important criteria to evaluate the performance of an implemented algorithm. The most accurate approach is useless for a real time task like driver assistance when it needs too much calculation time. At leat every 0.2 seconds a new result should be available from the algorithm, or in other words 5 results a second to provide the driver with sufficient information. This section will show how the above evaluated algorithms perform concerning execution time.

7.2.1 Lane Detector

The lane detector has been evaluated on several videos and always needed similar execution time. Here, two samples are presented, one with six real present lane-markings and a resolution of 720x480 pixels called $V1$ (see figure above 7.1 (a)) and the other with three present lane-markings (above 7.1 (b)) and resolution 640x480 called $V2$. The resulting execution times for both road scenes are shown in figure 7.8. Not only the enire execution time from grabbing the frame until having detected lane-markings is evaluated, but also all intermediate steps which are grey-scale filtering, image clipping, canny filtering, Hough Transform and finally line clustering and verification step. Furthermore, for each processed frame the number of detected lane-markings is given to provide information about possible influence of the number of detected lanes on the actual execution time.

It can be seen that the whole algorithmic execution time in both cases is about 0.1 seconds which is satisfying performance and fast enough to use it for drivers assistance. Moreover, it has to be taken into account, that the framework needs about 0.03 seconds in each step, which is about 30% of the whole execution time. This means running the algorithm without the framework which eventually would be the case for a driver assistance system would result in an even higher information rate.

The calculation time for the video with the lower resolution $V2$ is consequentially shorter as less information has to be processed. As expected, the Hough Transform needs the highest calculation time and depends the most on the different image resolutions, followed by the canny detection. The k-means clustering and lane verification step only needs a fraction of about one percent of the entire execution time and has the lowest impact on the performance. This further means, that execution time is not directly affected by the number of present lane-markings. The grey scaling and image clipping together also influence the whole time negligible. A conclusion of the average total execution time, intermediate steps execution time and the respective percentage of the whole can be seen in table 7.1

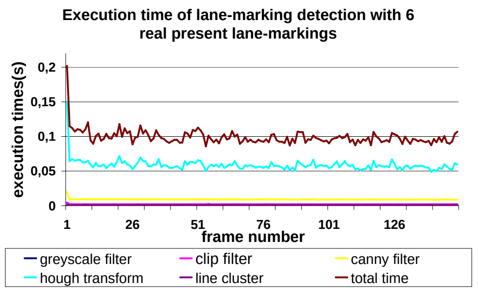
mean	Greyscale	Clipping	Canny	Hough	Cluster	ImprovCV	Total
time(s) $V1$	0,0018	0,0021	0,0090	0,0587	0,0010	0,0260	0,0988
% of whole	1,85	2,15	9,21	59,40	1,02	26,37	100
time(s) $V2$	0,0025	0,0030	0,0099	0,0425	0,0010	0,033	0,0922
% of whole	2,73	3,24	10,77	46,1	1,08	36,08	100

Table 7.1: Execution time regarding intermediate steps and resolution

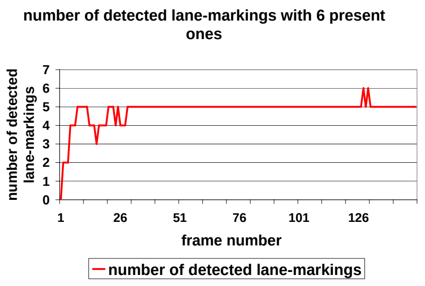
7.2.2 Vehicle Detection and Tracking

From the available test sequences two example videos have been elected for execution time evaluation. Their main difference is the number of present cars, where the first shows more cars than the second. These two were chosen to provide information whether execution time changes regarding the number of present cars. A similar procedure as for the lane detector has been chosen to measure the execution time of the vehicle detection and tracking system. The calculation time is evaluated separately of the whole procedure and of all intermediate steps to show which parts have the highest influence on the entire execution time. Furthermore, information about the number of candidate cars and detected objects is given. The number of points for which the optical flow is estimated has been defined to 150 as more points only result in a higher execution time but no better result.

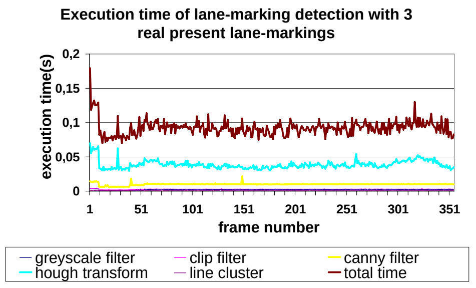
As for the lane detector, the k-means clustering works very fast and has the lowest influence on the execution time. The calculation of optical flow and the verification need the highest amount of time. Optical flow calculation takes longer, the more points have been chosen. The more cars are detected, the larger the execution time of the verification step primarily due to the higher number of time-expensive correlations. Regarding the entire algorithm's execution time, it performs fast enough even for a large number of detected vehicles. Thus taking the provided information rate into concern the system is applicable for a possible adoption to a driver assistance system.



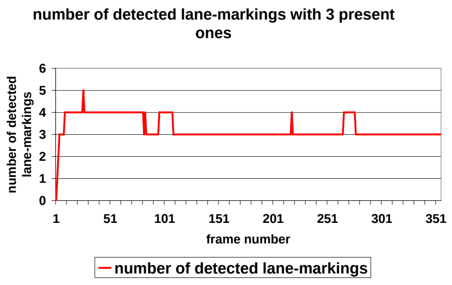
(a) Execution time of lane detection algorithm for a road scene with 6 present lane-markings, or 3 lanes



(b) Number of detected lane-markings in each frame with 6 present ones



(c) Execution time of lane detection algorithm for a road scene with 3 present lane-markings, or 2 lanes



(d) Number of detected lane-markings in each frame with 3 present ones

Figure 7.8: Execution time and detection rate of lane detection for two different road scenes

7.2.3 Stereo Procedure

As mentioned above, for choosing the best correspondence analysis method, not only accuracy plays an important role but for applying it to a driver assistance system which eventually should provide the driver with a high information rate, execution time is crucial. Thus an evaluation of the three presented methods SAD, SSD and NSC with five different search window sizes applied on the Tsukuba test stereo image set to create a full depth map has been performed. It was performed without any further reduction of the search area, which would have resulted in overall lower execution times but not a change of the wanted tendency. The resulting execution times for each depth map creation can be seen in table 7.2.

To be mentioned is the high computational expense for calculating complete depth maps, independent from the method which has been used, as the fastest method took over 15

Window Size	SAD	SSD	NSC
3×3	15,21s	23,48s	31,79s
5×5	17,05s	25,58s	34,13s
7×7	19,77s	27,68s	38,89s
9×9	21,74s	31,46s	44,67s
11×11	25,83s	38,41s	51,52s

Table 7.2: Calculation times for creating complete depth map of the Tsukuba Image size 320×288

seconds. As expected, the second remarkable fact is the increase of computation time with growing search window size, because the number of calculations rises with larger search windows. SAD's computation time increases nearly linear and slower than NSC's and SSD's. Finally, SAD performs the fastest as it needs only half the computation time, exactly in average $49,5\%$ of NSC's and 68% of SSD's computation time.

8 Conclusion and Future Work

A new open source image processing framework for automotive vision applications called ImprovCV has been introduced. ImprovCV provides the user with a set of standard and high-level image processing operators, based on the open source vision library OpenCV. Both videos and live camera streams can be processed. An image processing procedure containing any number of filters can be easily created via a drag and drop method and stored to reload it at a later time. During the procedure's execution, possible filter parameters can be adjusted to give the user a direct feedback. Furthermore, the user can process several videos at a time to directly compare results and the program has the capability of grabbing camera images synchronized for stereo vision applications. Its component-based software design makes it very flexible, modular and extensible. Giving developers the possibility of creating their own filters, and including them easily as a DLL into the framework yields a constant growth of available operators and makes ImprovCV very powerful and promising.

Currently ImprovCV is only available for Microsoft Windows but as all used libraries are cross-platform, a Version for Linux will follow. Further image processing operations in the field of machine learning are provided by OpenCV and can extend the program's ability. To directly utilize created procedures in an application without the overhead arisen by the framework, an easy C-code generator is desirable. As image processing on the Graphics Processing Unit (GPU) becomes more and more popular because it yields a high acceleration of operator's execution, an interface for GPU-based image processing filters is reasonable and possible to do as C libraries for processing on GPU do already exist, for example CUDA [50] and GPUCV [51].

For the purpose of driver assistance, and to show the ability of ImprovCV, three vision algorithms for lane detection, vehicle detection and tracking and a stereo application with the purpose of distance estimation have been presented and their respective results have been evaluated.

The lane detector is based on a Probabilistic Hough Transform and a k-means cluster method. It provides good information about the lane presently occupied by the vehicle and the outer lane-markings of the highway. Depending on adjustable parameters, the lane detection works well under several conditions, can detect both solid and dashed lane-markings and can cope with lane-markings partially occluded by cars. Concerning execution time, the algorithm performs fast enough for a driver assistance application as it yields a framerate of about $10-12$ frames per second. For further improvements a way for automatically adopting parameters, depending on the current conditions could be introduced. Adopting the algorithm to curved roads would result in further performance enhancement.

Detection of an arbitrary number of cars without any initialization by a human operator has been achieved by a system that uses motion information, obtained by extracted optical flow vectors further combined with a two stage k-means clustering, to get a first position approximation of present cars, the car candidates. An edge-based search for rectangular objects with a certain side ratio in the area of the resulting car candidates, followed by a correlation of these objects with predefined car templates has been applied to verify or reject the presence of a car in the searched area. Detected cars could be tracked in consecutive frames by using a correlation method. Under cooperative circumstances, meaning high contrast between cars and environment, this approach works robust and could detect up to six cars at the same time. The weak point of this system is the verification via predefined car templates as not all possible cars can be covered by the template database and some "magic numbers" have to be found to accept or reject car candidates. Satisfying performance concerning execution could be achieved as the procedure works with a frame rate of about 10 fps. To improve the algorithm, the car verification could be executed by using methods from the area of machine learning like Neural Networks or Support Vector Machines. Combining the car detection with the lane detection to reject false positives which are not on the road, like road signs or buildings would also lead to a further improvement.

A complete stereo vision approach has been introduced dealing with uncalibrated cameras. By obtaining the Fundamental Matrix with the 8-Point algorithm and the RANSAC method from assumed correspondences, the stereo images could be rectified. The fact that corresponding points of rectified images lie on the same scanline has been used for correspondence search. Three methods for finding correspondences, Sum of Absolute Difference (SAD), Sum of Squared Difference (SSD) and the Normalized Sample Correlation (NSC), have been implemented and evaluated with different search window sizes. SAD and SSD performed similar whereas NSC yields the poorest results. Concerning execution time, SAD is much faster than SSD and therefore the suggested method. A search window size of 7×7 is supposed as it is a tradeoff between highest accuracy and lowest execution time. The suggested methods used for finding matches of cars in a stereo set detected with the presented approach, and calculating the distance with the knowledge of disparity, the camera's focal length and the distance between the stereo cameras yields a good first distance estimation. The distance calculation can be improved to be more accurate by calibrating the cameras and using a triangulation method with the known intrinsic and extrinsic parameters. Dynamic programming for correspondence search and a cross validation of the detected matches could further improve this approach.

A Appendix

A.1 A small tutorial on ImprovCV

This tutorial gives a brief example of how to use ImprovCV. It shows how to get started and explains the core features of the program. The following image shows the GUI plus extra information added in red.

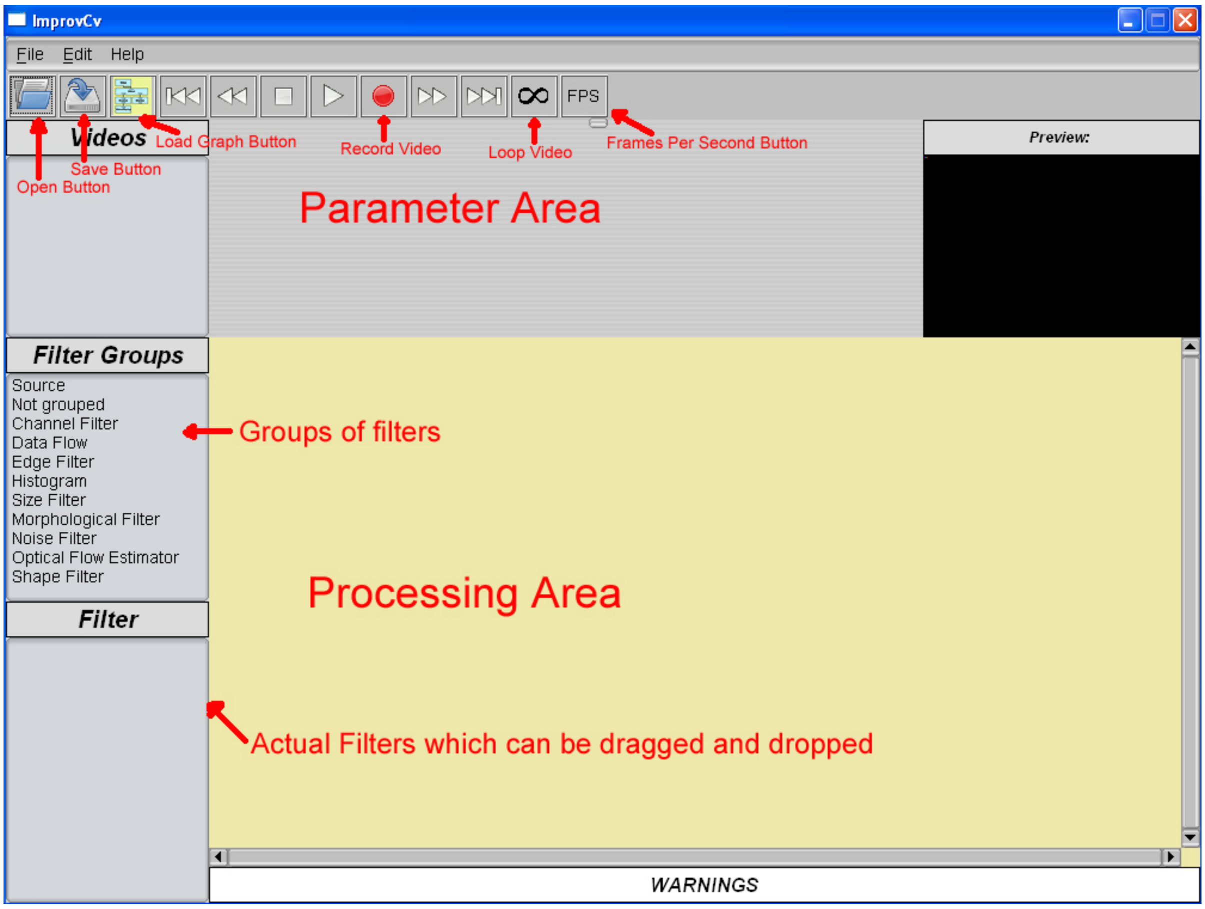


Figure A.1: Example Image showing how the plain GUI with additional information

Features:

- Loading a source

- Adding a filter
- The warning window
- The preview window
- Saving and Loading
- Deleting filter/s
- Stereo image processing
- Recording a video

A.1.1 Loading a source

There are 4 differnt ways of loading a video:

1. Click on *File* on the menubar and select the open menuitem, then simply brows the video you want to load via the file browser that will pop up
2. Click on the *Open* button on the button bar, rest is the same as above
3. Use shortcut ctrl+o, rest as above
4. Click on the *Source* item of the list *Filter Groups*, drag the Source listitem from the list appearing in Filter list, drop it in the processing window. Click on the occurring blue box called *source*, a checkmenubar occurs in the parameter area. Select *Connect Video*, rest is the same as above. The next image shows this case.

   If webcams are connected to your computer, there will be a checkbox called *Connect Camera*. By selecting this, depending on the number of connected cameras, a new list in this checkbox list will occur. Select the wanted camera.

After having loaded a source, the respective image will appear in the video window.

By default the video will be in loop mode, that means it will be repeated over and over again. By clicking on the looping button, you can switch so the video will not be repeated after it is finished.

Furthermore, you can select the Frames Per Second by clicking on the *fps* button. A drop down menu will occur from where you can select the number of frames per second you want. Once a source is loaded, there will be two boxes on the processing window source and video display. By clicking on the video display, the display window pops to the front.

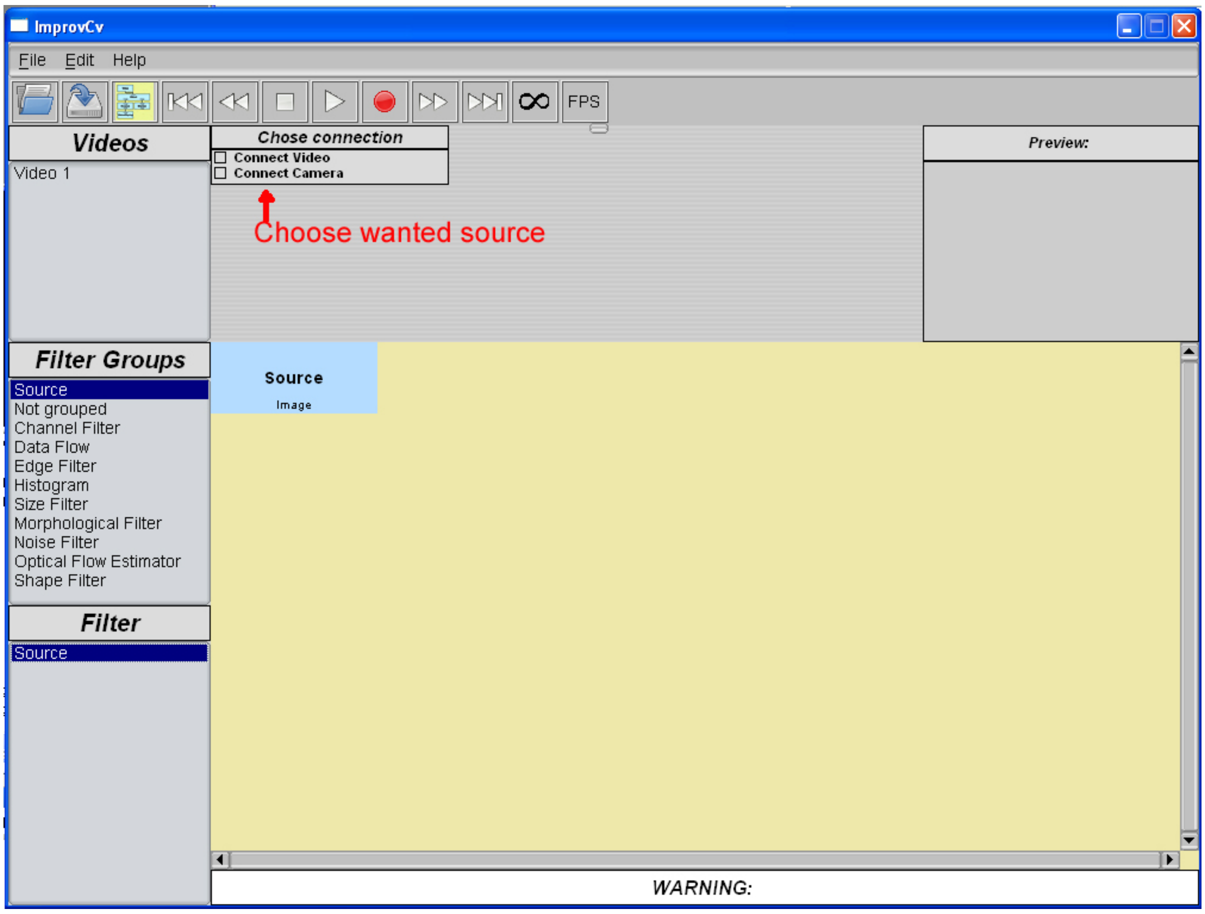


Figure A.2: Example Image showing how to connect a source

A.1.2 Adding a filter

In this easy example, an edge detection should be performed. Thus, after having loaded a source you choose the group *Edge Filter* from the filter group list. All filters being part of this group will be shown in the list of the Filter window. In this example you chose the sobel filter for performing the edge detection. Just select it by clicking on it, hold the mouse button and drop it on the position you wish to add it to. The result is shown in the next image.

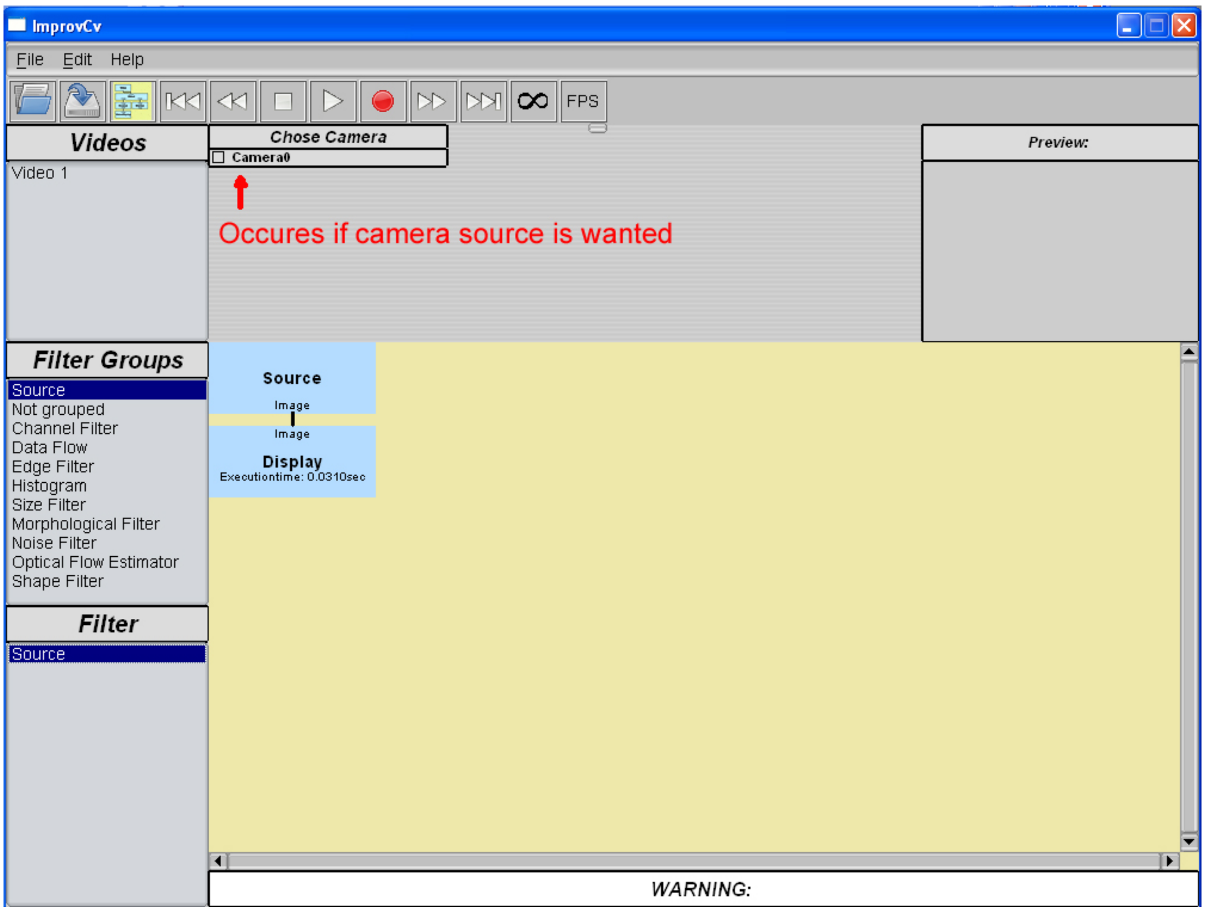


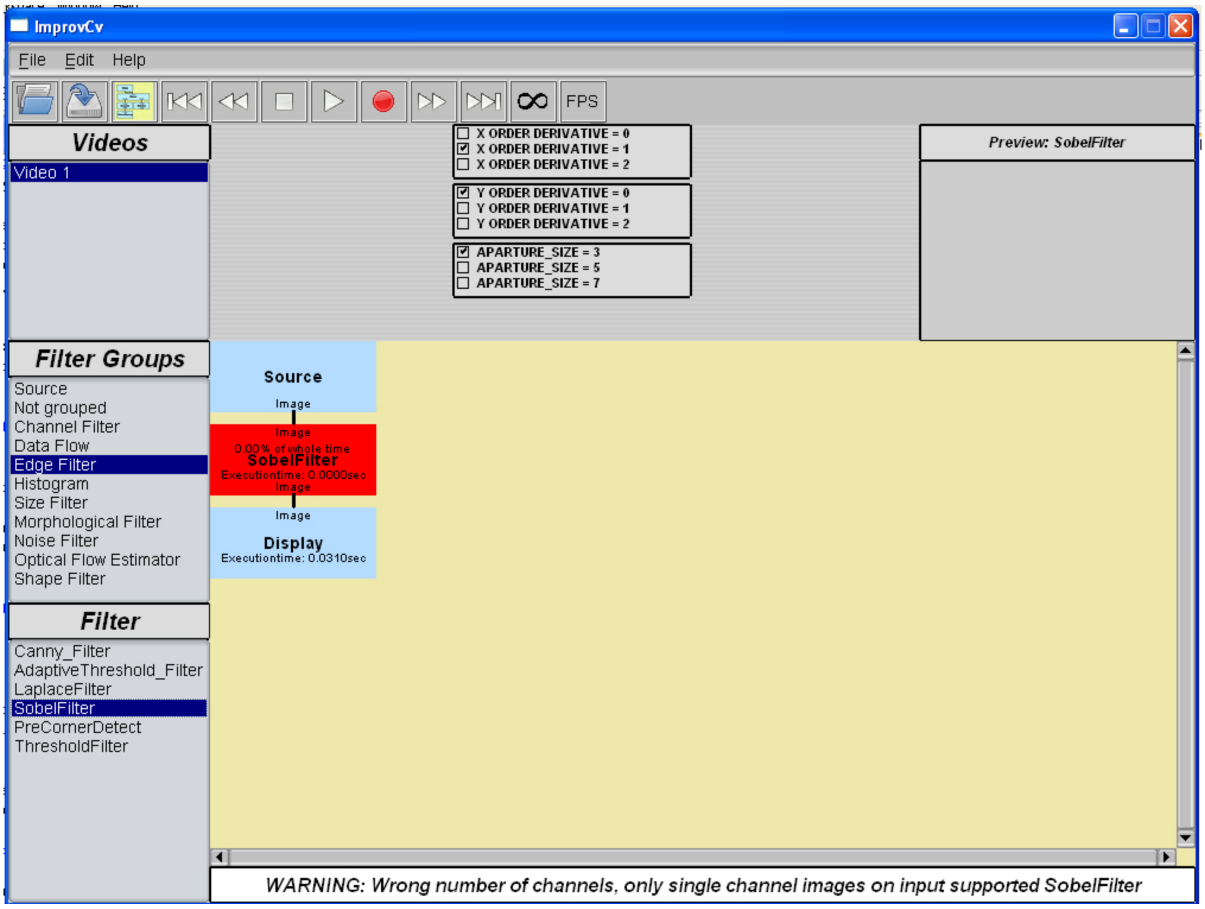
Figure A.3: Example Image showing how to connect a camera

A.1.3 The warning window

As the input image of the sobel filter is an RGB image, and sobel requires a greyscale image, no processing will be performed. Instead, a warning is given in the warning window on the bottom of the GUI. The problem is solved by adding a greyscale filter in front of the sobel filter from the Channel Filter group.

A.1.4 The preview window

Shown on the screen is always the output of the last filter in the graph. If you want to see the output of any other intermediate filter, you can either just click on the respective box and the output will be shown in the preview window plus in the headline over the preview window the name of the selected filter or you can add an extra display by adding the filter *ExtraDisplay* from the Data Flow group.



☐ X ORDER DERIVATIVE = 0
☑ X ORDER DERIVATIVE = 1
☐ X ORDER DERIVATIVE = 2

☑ Y ORDER DERIVATIVE = 0
☐ Y ORDER DERIVATIVE = 1
☐ Y ORDER DERIVATIVE = 2

☑ APARTURE_SIZE = 3
☐ APARTURE_SIZE = 5
☐ APARTURE_SIZE = 7

WARNING: Wrong number of channels, only single channel images on input supported SobelFilter

Figure A.4: Example Image showing a graph with one sobel filter

---

A.1.5 Saving and loading

Once the graph is created, you can store it in an XML file to reuse it. Saving can be performed in 3 different ways: either by selecting the *save* menu item from File in the menubar, with the shortcut ctrl+s or by using the save button from the button bar. Either way, you just have to select the place where you want to store the xml file via the file browser, then type in NAME.xml and click save.
For loading a file you either load it by selecting load from the file menu, click the *load* button from the button bar or the shortcut ctrl+l. Each time, a file browser will occur where you select the wished xml file. If you want to load a new file into an already existing graph, you have to click on *source* before you perform the steps mentioned above.

A.1.6 Deleting filter/s

If you wish to delete a filter from the graph, select the filter to be deleted by clicking on the respective box, then you can either click the *delete* button on you keyboard or select *Remove* from the *Edit* menu. If you want to remove all filters, you click on the source and hit ctrl+delete or select *clear graph* from Edit.

A.1.7 Stereo Image processing

As already described, you can process images from web cams as well. Furthermore, the program has the ability of grabbing frames from more than one source and even better if two webcams are connected, the grabbing will be synchronized so the frames provided by the cameras are taken at the same time.

The stereo processing is performed by first connecting a source (video or camera) as describe above, then you have to add the filter called *StereoSource* shown in the next image.

The boxes in the parameter area occur by clicking on the *StereoSource* box. If you want to connect another video, you just select *Connect Video* and perform it in the same way as described before. If you have two cameras connected to your computer, you will have the possibility of choosing *Connect Camera*. If you select this, the second camera will automatically be connected and synchronized with the first connected camera.

The output image from the already connected source is on the first image-output on the left of StereoSource, and the new one on the second output. You can change this by selecting *swap output images* as shown in the image above and you can swap them back by selecting the respective checkbox.

A.1.8 Recording a video

A resulting output can be recorded and stored in an .avi file. This is done by simply clicking on the *record* button, the red button in the button bar. Once clicked on it a file browser opens and you can select where to store the video and how to name it. When recording should be finished, click on the *pause* button in the button bar.

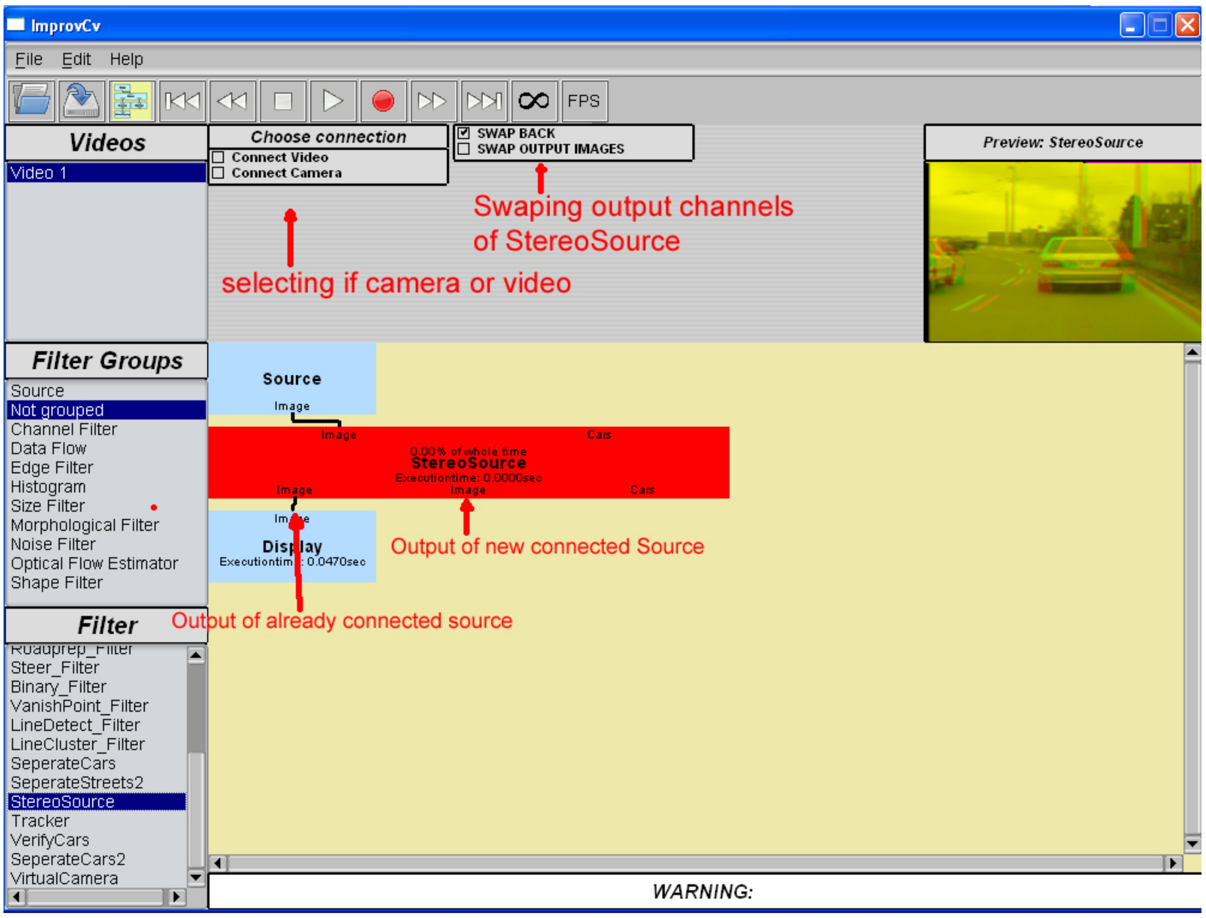


Figure A.5: Example Image for stereo processing

A.2 Filters provided by ImprovCV

- GreyScaleFilter
- ColorConvFilter; Converts images to a number of color spaces
- InvertFilter; Inverts each pixel
- ChannelSelector; separates channels from image
- Visualizer
- BlendFilter; Adds or subtracts two images
- Multiplexer
- ExtraDisplay
- Histogram
- Histoequ; Equalizes image's histogram

- HistoThresh; Thresholds or normalizes image's histograms
- Erosion
- Dilation
- Morphological
- Thinning
- MedianFilter
- GaussianFilter
- MinMaxMeanFilter
- CalcFundamentalMatrix
- CreateDepthMap
- CreateRedGreen
- FindOuterLanes; finds the most outer lane-markings
- FillWithBlack; fills everything with black beyond the detected outer lane-markings
- FlipFilter
- OpenCvDepth
- RectifyImages
- RedGreen; Splits am image into two images by selecting the red and green channel
- RoadprepFilter
- SteerFilter
- BinaryFilter
- VanishPointFilter
- LineDetectFilter
- LineClusterFilter
- SeperateCars
- SeperateStreets2
- StereoSource
- Tracker
- VerifyCars
- SeperateCars2
- VirtualCamera

- OpticalFlowFilter
- OpticalFlowHsFilter
- HoughLinesFilter
- HoughCirclesFilter
- ClipFilter
- ClipXY
- PyrDownImage; Performs image down-sampling by 2
- PyrUpImage; Performs image up-sampling by 2
- SkyClip

Bibliography

[1] Margie Peden. World report on road traffic injury prevention. Technical report, WHO, 2004. 1

[2] Willie D. Jones. Building saver cars. *IEEE Sprectrum*, pages 82–85, 2002. 1

[3] University of Stanford. http://cs.stanford.edu/group/roadrunner/media.html. viii, 2

[4] Uwe Franke, Dariu Gavrila, Axel Gern, Steffen Görzig, Reinhard Janssen, Frank Paetzold, and Chrisitian Wöhler. From door to door - principles and applications of computer vision for driver assistant systems. Technical report, Daimler Chrysler AG, 2001. 2

[5] Micron. http://www.micron.com/applications/automotive/. viii, 3

[6] Intel Corporation. http://www.intel.com/technology/computing/opencv/. 4, 9

[7] Zehang Sun, George Bebis, and Ronald Miller. On-road vehicle detection: A review. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 2006. 5

[8] Frank Dellaert. Canss: A candidate selection and search algorithm to initialize car tracking. Technical Report CMU-RI-TR-97-34, School of Computer Science Carnegie Mellon University Pittsburgh, October 1997. 5

[9] F. Dellaert and C. Thorpe. Robust car tracking using kalman filtering and bayesian templates. In *SPIE: Intelligent Transportation Systems*, 1997. 5

[10] Zehang Sun, George Bebis, and Ronald Miller. Monocular pre-crash vehicle detection: Features and classifiers. Technical report, Computer Vision Lab. Department of Computer Science, University of Nevada, Reno e-Technology Department, Ford Motor Company, Dearborn, MI, 2004. 5

[11] Jaesik Choi. Realtime on-road vehicle detection with optical flows and haar-like feature detector. Technical report, Computer Science Department University of Illinois at Urbana-Champaign Urbana, IL 61801, 2006. 6

[12] Joel C. McCall and Mohan M. Trivedi. An integrated, robust approach to lane marking detection and lane tracking. In *IEEE Intelligent Vehicle Symposium*, 2004. 6

[13] Yue Wanga, Eam Khwang Teoha, and Dinggang Shenb. Lane detection and tracking using b-snake. In *Image and Vision Computing 22*, 2004. 6

[14] Thomas Bräunl. http://robotics.ee.uwa.edu.au/improv/. UWA Robotics Lab. 7, 8

[15] Trolltech. http://trolltech.com/products/qt. 7, 9

[16] Konstantinos Konstantinides and John R. Rasure. The khoros software development environment for image and signal processing. *IEEE Transactions on Image Processing*, volume 3:pages 243–252, 1994. 7

[17] AccuSoft. http://www.accusoft.com/products/visiquest/. 7

[18] RoboRealm. http://www.roborealm.com/. 7

[19] The MathWorks. http://www.mathworks.com/products/image/ http://www.mathworks.com/products/viprocessing/. 7

[20] MVTec. http://www.mvtec.com/halcon/. 8

[21] Bill Spitzak et. al. http://www.fltk.org/. 9

[22] Julian Smart et. al. http://www.wxwidgets.org/. 9

[23] Hendrik Dahlkamp, Adrian Kaehler, David Stavens, Sebastian Thrun, and Gary R. Bradski. Self-supervised monocular road detection in desert terrain. In *Robotics: Science and Systems*, 2006. 10

[24] Lee Thomason. http://www.grinninglizard.com/tinyxml/. 10

[25] Xia Cai, Michael R. Lyu, Kam-Fai Wong, and Roy Ko. Component-based software engineering: Technologies, development frameworks, and quality assurance schemes. 13

[26] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlisside. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesly, 1995. 13

[27] Bruce Eckel. *Thinking in C++ 2nd edition Volume 2: Standard Libraries & Advanced Topics*, chapter 16, page 441. 1999. 14

[28] Dia Kharrat and Syed Salman Qadri. Self-registering plug-ins: An architecture for extensible software. In *Canadian Conference on Electrical and Computer Engineering*, pages 1324–1327, 2005. 14

[29] Jeremy G. Siek, Lie-Quan Lee, and Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison Wesley Professional, 2001. 16

[30] Paul Hough. Method and means of recognizing complex patterns. *U.S. Patent 3,069,654*, 1962. 20

[31] Richard O. Duda and Peter E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Communications of the ACM*, volume 15:pages 11–15, January 1972. 21

[32] N. Kiryati, Y.Eldar, and A.M. Bruckstein. A probabilistic hough transform. In *Pattern Recognition*, volume 24, pages 303–316, 1991. 22

[33] Berthold K.P. Horn and Brian G. Schunk. Determing optical flow. 1980. 22, 24

[34] B. D. Lucas and T. Kanade. An iterative image registration technique with an application to stereo vision. In *International Joint Conference on Artificial Intelligence (IJCAI81)*, pages 674–679, 1981. 25

[35] Jean-Yves Bouguet. Pyramidal implementation of the lucas kanade feature tracker. description of the algorithm. 26

[36] Jianbo Shi and Carlo Tomasi. Good features to track. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR'94)*, volume June, 1994. 26

[37] Frank Dellaert. The expectation maximization algorithm. Technical Report GIT-GVU-02-20, College of Computing, Georgia Institute of Technology, February 2002. 28

[38] Ulrike von Luxburg. A tutorial on spectral clustering. Technical Report TR-149, Max Planck Institute for Biological Cybernetics, August 2006. 28

[39] J. B. MacQueen, editor. *Some Methods for Classification and Analysis of Multivariate Observations*. California Press, 1967. 28

[40] H. C. Longuet-Higgins. A computer algorithm for reconstructing a scene from two projections. In *Nature*, volume 293, September 1981. 32, 33

[41] Richard I. Hartley. In defense of the eight-point algorithm. In *IEEE Transactions on Pattern Analysis and Machine Intelligence*, volume 19, pages 580–593, June 1997. 35

[42] Zhengyou Zhang. Determining the epipolar geometry and its uncertainty: A review. Technical Report RR-2927, INRIA, 1996. 35

[43] M. Fischler and R. Bolles. Random sampling consensus: A paradigm for model fitting with application to image analysis and automated cartography. In *Communications of the ACM*, volume 24, pages 381–395, 1981. 35

[44] Charles Loop and Zhengyou Zhang. Computing rectifying homographies for stereo vision. Technical Report MSR-TR-99-21, Microsoft Research, 1999. 38

[45] A. Fusiello, E. Trucco, and A. Verri. Rectification with unconstrained stereo geometry. In *Proceedings of the British Machine Vision Conference. BMVC 1997*, pages 400–409, September 1997. 38

[46] Marc Pollefeys, Reinhard Koch, and Luc J. Van Gool. A simple and efficient rectification method for general motion. In *ICCV (1)*, pages 496–501, 1999. 39

[47] R.A. Lane and N.A. Thacker. Stereo vision research: An algorithm survey, January 1996. 39

[48] Margrit Betke, Esin Haritaoglu, and Larry S. Davis. Multiple vehicle detection and tracking in hard real time. Technical Report CS-TR-3667, Computer Vision Laboratory, Center for Automation Research and Institute for Advanced Computer Studies University of Maryland College Par, MD 20742-3275, 1996. ix, 51, 55

[49] David Forsyth and Jean Ponce. *Computer Vision - A Modern Approach.*, chapter 13, pages 355–360. Prentice Hall, 2003. 63

[50] Nvidia. http://developer.nvidia.com/object/cuda.html. 73

[51] Jean-Philippe Farrugia, Patrick Horain, Erwan Guehenneux, and Yannick Alusse. https://picoforge.int-evry.fr/cgi-bin/twiki/view/gpucv/web/webhome. 73