Universität Passau

Fakultät für Informatik und Mathematik

Bachelorarbeit

Automatisierung der Kartographierung von Straßenschildern

Verfasser:

Philipp Unger

11. März 2013

Prüfer:

Prof. Dr. Tomas Sauer

Lehrstuhl für Mathematik mit Schwerpunkt Digitale Bildverarbeitung

Innstraße 43, D-94032 Passau

Inhaltsverzeichnis

1 Einleitung

1.1 Motivation

In der heutigen Zeit, in der man immer und überall Internet zur Verfügung hat, sei es auf Notebooks, Smartphones, Tablets oder auch auf einem PC zu Hause, nutzt man, wenn es um die Planung von Routen oder um das Auffinden eines Ortes geht meist von *Google*[1] bereitgestelltes Kartenmaterial. Das Benutzen der auf *GoogleMaps* bereitgestellten Karten ist zwar kostenlos, aber nicht frei. Will man die bereitgestellten Karten vervielfältigen oder auf seiner Homepage nutzen, fallen zum Teil hohe Kosten für die proprietären Karten an. Darüber hinaus erhält man nur Zugriff auf das fertige Layout der kartographischen Daten und hat keine Möglichkeit, das Kartenmaterial mit eigenen Daten zu modifizieren.

Eine aufstrebende Alternative zu den proprietären Karten von *GoogleMaps* ist *OpenStreetMap* (OSM). Die Nutzung der von OSM bereitgestellten Geodaten[2] ist kostenlos und frei. Hier erhält der Nutzer vollen Zugriff auf die Geodaten selbst, was ihm ermöglicht, die Daten individuell zu ändern und an seine Bedürfnisse anzupassen. Auch das Erstellen eigener Kartenlayouts ist möglich.

Alle von OSM bereitgestellten Geodaten kommen von den Benutzern selbst, die weltweit alle Straßen, Gebäude und vieles mehr selbst mappen[3]. Dabei liegt das Interesse schon lange nicht mehr nur auf der Lage dieser Objekte, sondern auch auf detailreicheren Informationen über diese. So findet man auf den heutigen Karten Restaurants mit ihren Öffnungszeiten, Geschäfte mit Namen und Telefonnummer und vieles mehr. Besonderes Interesse liegt aber auf dem komplexen Straßennetz, das all das oben genannte miteinander verbindet. Straßen tragen viele wichtige Informationen, die dem Verkehrsteilnehmer über Verbots- und Gebotsschilder sowie Hinweisschilder gegeben werden.

Das Erfassen aller wichtigen kartographischen Attribute ‚zu Fuß' ist ein großer Aufwand für die Mapper[4] und stellt im aktiven Straßenverkehr auch ein großes Risiko dar. Diese Arbeit befasst sich mit der Erfassung und Speicherung von Objekten im Straßenverkehr und den dazugehörigen Global Positioning System Daten (GPS Daten) mittels Android Applikation, sowie der automatischen Erkennung und Kartographierung dieser Objekte mittels eines Java Programms.

1.2 Ziel der Arbeit

Das Ziel dieser Bachelorarbeit war es, wie unter 1.1 bereits kurz erwähnt wurde, ein Programm zur automatisierten Erkennung von Objekten im Straßenverkehr zu entwickeln und mit der Auswertung des aufgezeichneten GPS Exchange Format Tracks[5] (GPX-Track) deren geographischen Ort zu bestimmen.

Dabei soll für die Aufzeichnung straßenspezifischer Daten eine Android Applikation erstellt werden, die es dem Mapper ermöglicht, beim Fahren mit dem Auto, Fahrrad oder ähnlichem, die gefahrene Strecke aufzuzeichnen. Mit einem Java Programm soll der Mapper dann zuhause am PC die Videos

[1] Homepage: http://www.google.de/
[2] Digitale Informationen, denen eine Lage auf der Erdoberfläche zugewiesen werden kann.
[3] Das persönliche Erfassen von GPS Daten vor Ort.
[4] *OpenStreetMap*-Nutzer, welcher selbst Daten erhebt, Karten erstellt und diese online bereitstellt.
[5] Dicht aufeinanderfolgende Track-Punkte, die beispielsweise von einem GPS Gerät aufgezeichnet wurden und einen Pfad bilden.

automatisch auswerten lassen können. Dabei sollen zusätzlich, mit einem speziell zur Straßenschild-erkennung entwickelten Plugin, die Schilder automatisch erkannt und abgespeichert werden.

Zur Weiterverarbeitung der gefundenen Informationen, werden die Straßenschilder anschließend an der erkannten Position als Waypoint[6] (WP) in der aufgezeichneten GPX-Datei und als JPEG Bild in einem gewählten Verzeichnis abgespeichert. So kann ein Schild einem genauen Punkt auf der Karte zugeordnet werden.

1.3 Vorgehensweise

Zur Aufzeichnung der gesuchten Objekte im Straßenverkehr wird in dieser Arbeit ein Smartphone mit Android Betriebssystem benutzt. Android ist weit verbreitet und bietet dem Entwickler alles, was er zur Entwicklung komplexer Systeme braucht. Dabei erhält er auch vollen Zugriff auf die Hardware des Smartphones, was den Anforderungen an Kamera und GPS in dieser Arbeit zu Gute kommt. Der technische Fortschritt macht vor allem im Bereich der Mobilfunktechnologie unaufhörlich große Schritte nach vorne. So verfügen aktuelle Mobilfunkgeräte bereits über integrierte *Full High Definition*[7] (Full HD) Farbkameras und einen integrierten GPS-Empfänger.

Mit der Kamera des Android Smartphones werden der Straßenverkehr und die Straße, sowie der Bereich links und rechts neben der Fahrbahn aufgezeichnet.

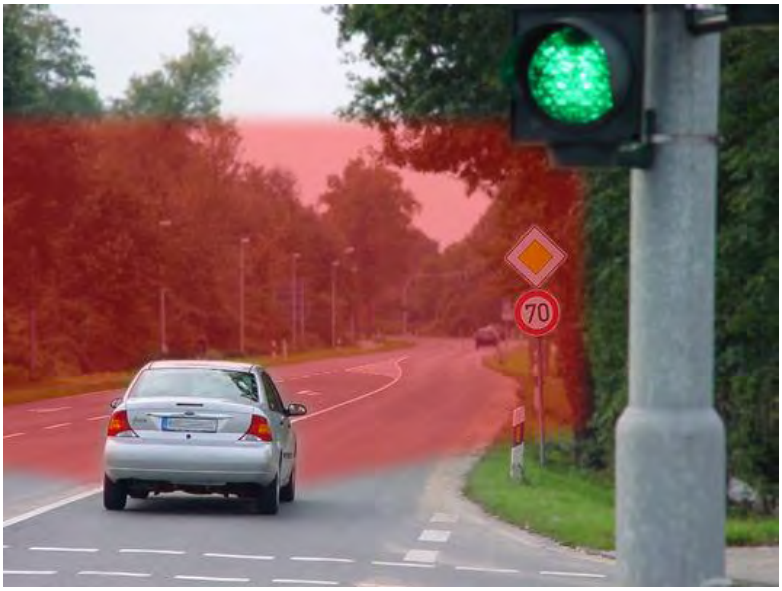

Abbildung 1-1: Kamerasichtfeld durch die Windschutzscheibe eines PKW. Das Sichtfeld wird rot dargestellt. [1]

Dazu wird das Smartphone beispielsweise hinter der Windschutzscheibe des Kraftfahrzeugs angebracht. Der integrierte GPS-Empfänger zeichnet kontinuierlich die aktuellen GPS-Positionen auf, sobald die Videoaufnahme gestartet wird. Nach beenden der Aufnahme werden das Video und die GPS-Daten im Speicher des Smartphones abgelegt. Für die simultane Aufzeichnung von Video- und GPS-Daten wurde eine Android-Applikation namens *GPStreetCam* entwickelt.

Die Daten werden später am PC ausgelesen und mit einem Java Programm geöffnet. Das Programm namens *GPStreetTracker* ist ein in dieser Arbeit entwickeltes Framework, welches die vom

[6] Ein einzelner Ortspunkt, dem Informationen angefügt werden können.
[7] Bezeichnet die höchste, heutzutage für den Konsumbereich verwendete Auflösung von 1920 × 1080 Pixeln.

Smartphone aufgezeichneten Videodateien einlesen und verarbeiten kann. Über vom Benutzer ausgewählte Plugins können dann verschiedene Objekte im Video erkannt werden. Das Programm bietet dem Nutzer außerdem die Möglichkeit, die erkannten Objekte zu selektieren und diese dann automatisch in den zum Video gehörenden GPX-Track eintragen zu lassen.

Den Kern der Arbeit bildet ein Plugin zur Erkennung von Straßenschildern. Das Plugin namens *HoughSignRecognition* verwendet dazu unter anderem verschiedene Farbfilteralgorithmen, um die *Regions Of Interest*[8] (ROI) in der Objektumgebung zu finden, und Hough-Transformationen zur Formerkennung und bietet dem Benutzer die Möglichkeit, alle Parametereinstellungen selbst anzupassen.

Ein weiterer Arbeitsschritt ist die Klassifikation der erkannten Verkehrszeichen. Im Rahmen dieser Arbeit findet nur eine sehr einfache Klassifikation statt. Wurde ein Schild anhand seiner Farbe und Form erkannt, wird nur unterschieden, ob das Schild der Gruppe der runden, drei-, vier- oder achteckigen Schilder angehört. Die Unterscheidung der Straßenschilder ist in dieser Arbeit für den Benutzer von keiner Bedeutung, da dieser als Ergebnis nur die Bilder der Schilder und ihre jeweilige Zuordbarkeit auf der Karte erwartet. Daher dient die einfache Klassifikation lediglich dem Tracking der erkannten Schilder.

Ein Tracking ist notwendig, da ein Verkehrszeichen in einem Video auf mehreren Frames hintereinander zu sehen ist. Dabei kann es auch vorkommen, dass das Schild zwischendurch von anderen Objekten teilweise oder ganz verdeckt wird oder aufgrund von Verzerrungen der Kamera oder schlechter Ausleuchtung der einzelnen Frames, von den Erkennungsalgorithmen nicht erkannt werden kann. Daher muss der Tracker[9] auf diese Verhältnisse angepasst werden können und dazu in der Lage sein, ein Schild über diese Erkennungslücken hinweg zu verfolgen.

Am Ende der Erkennungs- und Verfolgungsphase bietet das Programm dem Benutzer die Möglichkeit, die Schilder auszuwählen, welche er als Bilddatei im JPEG Format abspeichern und in den GPX-Track als Waypoint eintragen lassen möchte. In die Subinformationen[10] des Waypoints wird dabei der Name der Bilddatei eingetragen, welcher später in einem zur GPX-Betrachtung geeigneten Programm, wie beispielsweise *JavaOpenStreetMap* (JOSM), angezeigt werden.

1.4 In dieser Arbeit

In dieser Bachelorarbeit werden zunächst wichtige Begriffe des Themas, sowie die verwendeten Technologien und Mittel genannt und kurz erklärt. Anschließend folgt eine kurze Einführung in das Gebiet der Objekterkennung, speziell das der optischen Objekterkennung. Dabei wird genauer auf die computer-visuelle Erkennung(*Computer Vision Object Recognition*) eingegangen und aktuelle Methoden und Vorgehensweisen dazu vorgestellt. Danach werden die in dieser Arbeit zur Straßenschilderkennung verwendeten Methoden gezeigt und die Gründe genannt, warum diese verwendet wurden. Dazu werden detailliert die einzelnen Verfahren hinter den verwendeten Methoden erklärt. Im Anschluss wird genauer auf die Entwicklung der Android Applikation *GPStreetCam* und des Java Programms *GPStreetTracker* eingegangen und auf die Umsetzung der

[8] Ein Bereich in einem Digitalbild, auf dem ein besonderes Interesse liegt.
[9] Ein Algorithmus, welcher zur Verfolgung von *StreetObjects* verwendet wird.
[10] Neben der Hauptinformation des Waypoints, seinem Längengrad- und Breitengrad-Wert, dient diese Information als Notiz, welche eine nähere Beschreibung des Waypoints liefert.

vorgestellten Objekterkennungsmethoden. Am Ende der Arbeit wird in einem Testkapitel gezeigt, wie gut die entwickelten Programme die geforderten Anforderungen erfüllen.

2 Definitionen und verwendete Mittel

In diesem Kapitel werden einige Begriffe, die im Zusammenhang mit dem Themengebiet der Objekterkennung stehen, sowie die verwendeten Technologien und Mittel, die im Rahmen dieser Arbeit verwendet wurden, genannt und jeweils kurz erklärt.

2.1 Definitionen

2.1.1 Bildverarbeitung

Der Begriff *Bildverarbeitung* wird im täglichen Gebrauch sehr oft mit dem Begriff *Bildbearbeitung* verwechselt. Dabei haben diese beiden Begriffe im Grunde nur das Wort *Bild* gemeinsam.

Wie man an der Endung des Wortes bereits gut erkennen kann, geht es im Bereich der Bildbearbeitung um die Bearbeitung der digitalen Bildinformationen, sprich um die Manipulation der einzelnen Pixel eines Digitalbildes. Eine weithin bekannte Bildbearbeitungssoftware ist Photoshop[11]. Mit ihr lassen sich Bilder in jeder nur erdenklichen Art und Weise manipulieren, deshalb ist das Programm aus heutigen Medien und Werbebranchen nicht mehr wegzudenken wäre.

Im Gegensatz dazu, will man in der Bildverarbeitung keine eigenen Informationen in das Bild einbringen und die Farbinformationen somit manipulieren, sondern vorhandene Informationen aus dem Digitalbild extrahieren. So auch im Bereich der Objekterkennung. Hierzu werden auf den gegebenen Pixelinformationen spezielle Algorithmen angewendet, welche das Bild anhand seiner Farb- und Helligkeitsinformationen verarbeiten und somit Flächen, Kanten und Formen erkennen können.

2.1.2 Objektraum

Ein Objektraum fasst die Menge aller Objektpunkte zusammen, die ein optisches System abbilden kann. Ein optisches System wäre beispielsweise eine Kameralinse, welche die von der Umgebung reflektierten Lichtstrahlen bricht und auf einen Kamerasensor wirft. Ein Objektpunkt ist dabei der kleinstmögliche Punkt, den dieses System erfassen kann.

2.2 Technologien und Mittel

2.2.1 Global Positioning System

Das Global Positioning System (GPS) ist ein satellitenbasiertes System zur Bestimmung der Lage von Punkten auf der Erdoberfläche.

2.2.1.1 Aufbau des Systems

Das Global Positioning System besteht aus immer mindestens 24 Satelliten, nur so kann gewährleistet werden, dass man von einem beliebigen Punkt der Erdoberfläche aus immer Kontakt zu mindestens 4 Satelliten hat. Die Satelliten selbst umkreisen die Erde in einer Höhe von ca. 20200 Kilometern, ihre Umlaufzeit beträgt dabei knapp 12 Stunden.

2.2.1.2 Funktionsweise und Positionsbestimmung

Ein GPS Satellit sendet 24 Stunden am Tag sein Signal zur Erde. Das Signal ist in *Subframes* unterteilt, wobei die zeitliche Länge eines Subframes 6 Sekunden beträgt. Ein Subframe enthält die Information, wie viele Subframes in der letzten Bezugszeiteinheit gesendet wurden. Eine Bezugseinheit hat eine

[11] Homepage: http://www.adobe.com/de/products/photoshopfamily.html

zeitliche Länge von 7 Tagen und beginnt bzw. endet jeweils am Sonntag um 0 Uhr. Ein Subframe besteht wiederum aus 10 *Words*, welche wiederum aus 30 *Nachrichtenbits* bestehen. Die Zeitliche Länge eines Nachrichtenbits beträgt also:

$$\frac{6\,Sekunden}{10\,Words\,\times\,30\,Nachrichtenbits} = 0.02\,Sekunden = 20ms$$

Jedes Nachrichtenbit besteht aus 20 Codeblöcken, die jeweils eine zeitliche Länge von 1ms besitzen. Die Codeblöcke beinhalten einen einmal pro Millisekunde generierten, pseudozufälligen[12] Code, den sogenannten *Coarse/Aquisition Code* (C/A Code), auch *Chip* genannt. Diese *Chips* enthalten keine Nutzerinformation, sondern dienen nur zur Demodulation mittels Korrelation mit der Codefolge selbst [2]. Zur Zeitbestimmung trägt jeder Satellit mindestens eine Atomuhr an Bord, über deren Frequenz von 10,23MHz auch der C/A Code generiert wird.

Aus den beim Empfänger angekommenen Paketen kann nun die Zeit berechnet werden, indem der Empfänger den gleichen C/A Code erzeugt, wie er auch in den *Chips* des Satelliten enthalten ist, und die beiden Codeströme dann solange gegeneinander verschiebt, bis sie deckungsgleich sind. Die Anzahl der *Chips* um die die Codeströme verschoben worden sind, entspricht der zeitlichen Entfernung des Satelliten in Millisekunden. In der Theorie kann nun die Entfernung zum Satelliten vom Empfänger über die einfache Formel

$$Strecke = Geschwindigkeit \times Zeit$$

berechnet werden, da die Geschwindigkeit der gesendeten Chips, der Geschwindigkeit eines Radiosignals entspricht (im Vakuum 299792,5 km/s) [3] und somit bekannt ist. Da sich die Satelliten jedoch mit einer Geschwindigkeit von knapp 4km/s durchs All bewegen, erscheint die Zeit an Bord der Satelliten gedehnt. Dieser Effekt lässt sich mit der Speziellen Relativitätstheorie erklären. Diesem Effekt entgegen wirkt die Allgemeine Relativitätstheorie, welche besagt, dass der Gang einer Uhr umso langsamer vonstattengeht, je größer das Gravitationspotenzial am Ort dieser Uhr ist. Aus diesem Grund laufen die Uhren auf der Erdoberfläche langsamer als die an Bord des Satelliten. Um diesen Effekt auszugleichen, werden die Atomuhren an Bord der Satelliten verlangsamt, indem man sie auf eine niedrigere Grundfrequenz einstellt [4].

Mit den Entfernungen zu drei Satelliten, kann nun die Position des Empfängers bestimmt werden. Die Ausbreitung der Signale der Satelliten geschieht kugelförmig, mit dem Schnitt zweier solcher Kugeln bekommt man eine Schnittkreisfläche, schneidet man den Umkreis dieser Fläche mit einer dritten Kugel, bekommt man zwei Schnittpunkte. Um den genauen Ort zu bestimmen muss einer dieser beiden Schnittpunkte ausgeschlossen werden. Dies kann über das Signal eines vierten Satelliten erreicht werden. Die vierte Entfernungsmessung ist auch notwendig, da zusätzlich zu den drei Positionskomponenten der Fehler in der Empfängeruhr ausgeglichen werden muss. Eine kleine Abweichung der Uhr von schon einer $\frac{1}{1000} Sekunde$ würde zu einer Abweichung von mehreren 100 Kilometern führen.

2.2.1.3 Fehlerquellen

Bei der Ortsbestimmung durch GPS kann es immer zu Fehlern und Ungenauigkeiten kommen. Grund dafür ist eine falsche Entfernungsberechnung zu den Satelliten. Das kann viele verschiedene

[12] Ein aus der 10,23 MHz Frequenz der Satellitenatomuhr gewonnener Code mit den statischen Eigenschaften von zufälligem Rauschen.

Ursachen haben. Zum Beispiel können die Satelliten in einem ungünstigen Winkel zum Empfänger stehen, sodass das Signal von der Atmosphäre gestört wird. Es kann außerdem zu Störungen kommen, wenn das Signal von Bäumen, Gebäuden oder anderen Objekten reflektiert wird, was vor allem in städtischen Gebieten oft passiert.

Im Gegensatz zu Anwendungsgebieten, wie Verkehrsnavigation oder Landvermessung, bei denen eine hohe Genauigkeit der Ortsbestimmung zwingend von Nöten ist, haben hohe Abweichungen in dieser Arbeit kein großes Schadenspotenzial. Da der Mapper sich die Position der im Straßenverkehr erkannten Objekte, anhand seiner gefahrenen Strecke und des Kartenmaterials, leicht erschließen kann. Jedoch kommen so hohe Abweichungen selbst bei Ortung mit einem Smartphone nur äußerst selten zustande. Die durchschnittliche Genauigkeit liegt hier meistens bei unter 2 Metern.

2.2.2 OpenStreetMap

2.2.2.1 Was ist OpenStreetMap?

OpenStreetMap (OSM) ist ein im Jahr 2004 gegründetes Projekt mit dem Ziel eine freie Weltkarte zu erschaffen. Weltweit werden dafür Daten über Straßen, Bahngleise, Flüsse, Wälder, Häuser und allem Anderen gesammelt, was man kartographisch darstellen kann. Die Daten werden von den Benutzern selbst erhoben und nicht aus bereits existierenden Datenbanken oder Karten entnommen. Somit gehören auch alle Rechte an diesen Daten den Nutzern selbst. Die *OpenStreetMap*-Daten darf jeder lizenzkostenfrei einsetzen und beliebig weiterverarbeiten [5].

Ziel der ganzen Unternehmung ist es, eine kostenfreie Weltkarte zu erstellen. Denn Geoinformationen sind heutzutage selten frei erhältlich und man bekommt sie nur durch den Kauf einer unter Umständen sehr teuren Lizenz. Darunter fallen auch Karten, die beispielsweise für Unterricht-, Forschungs- oder Lehrzwecke benötigt werden. Hierfür werden oft *Google*-Karten aus *GoogleMaps* oder *GoogleEarth* verwendet. Die Benutzung dieser Karten ist zwar kostenlos, aber keineswegs frei. *Google* gewährt dem Benutzer eine nicht exklusive und nicht übertragbare Lizenz zum Zugriff auf den *GoogleMaps*-Dienst. Ohne schriftliche Genehmigung von *Google* darf der Inhalt weder ganz noch teilweise kopiert, übersetzt, abgeändert oder abgeleitete Werke daraus erstellt werden [6]. Aus diesem Grund darf der *OpenStreet*-Mapper[13] seine Informationen nicht aus diesen proprietären Karten beziehen, sondern muss diese selbst vor Ort erheben. Zudem kommt noch hinzu, dass zwar das Kartenmaterial zur Verfügung gestellt wird, jedoch nicht die zugrundeliegenden Geodaten. Der Preis für solche Daten wird einem schnell bewusst, wenn man ein neues Navigationsgerät erwirbt und selbst bei dem hohen Einkaufspreis dieser Geräte, sind die enthaltenen Daten oft veraltet oder unvollständig.

Mit *OpenStreetMap* soll die Abhängigkeit von proprietären Daten beendet werden und dem kreativen Benutzer kostenfreies, weltweites Kartenmaterial zur Verfügung gestellt werden, das er selbst nach Belieben vervollständigen und an seine Zwecke anpassen kann.

2.2.2.2 Kartographie für Jedermann

OpenStreetMap stellt für alle seine Benutzer freies Geodatenmaterial zur Verfügung. Auch selbst gesammelte Geodaten können von den Benutzern eingebracht werden. So kann man beispielsweise mit einem tragbaren GPS-Gerät selbst Daten erheben und diese später in OSM einbringen. Auch dafür zugelassene, freie Luftbilder können verwendet werden, um Straßen, Flüsse und Häuser

[13] Person, welche für OpenStreetMap Karten erstellt und online bereitstellt.

nachzuzeichnen. Eine grundlegende Ortskenntnis zur Namensgebung der Straßen und deren Verlauf ist dafür jedoch unumgänglich.

Möglich wird dies dem Benutzer unter anderem durch die freie Java Software namens *JavaOpenStreetMap*[14] (JOSM). Mit JOSM kann der Benutzer Geodaten von der zentralen OSM-Datenbank abrufen und sich diese in einem Kartenlayout darstellen lassen. Die Software bietet dem Benutzer eine intuitiv bedienbare Umgebung, mit der er die OSM Daten vervollständigen oder auch Fehler ausbessern kann. Es werden viele Funktionen bereitgestellt, darunter auch vorgefertigte Templates für die Erstellung von Brücken, Zufahrten zu Häusern, Restaurants oder Parkplätzen. Um dem Benutzer das Bearbeiten dieser Karten möglichst leicht zu gestalten, wird in diesem Programm mit Ebenen gearbeitet. Diese kennt man schon von Bildbearbeitungsprogrammen wie Photoshop oder GIMP[15]. So können zum Beispiel auch die oben bereits genannten Luftbilder oder auch andere Kartenbilder leicht über die Geodaten gelegt werden, wodurch die Orientierung des Benutzers auf der Karte um ein Wesentliches erleichtert wird. Des Weiteren bietet die Software eine Upload-Funktion an, mit der der Benutzer über einen angelegten Account die hinzugefügten oder geänderten Geodaten in die OSM-Datenbank hochladen kann. Nach einer Überprüfung werden diese Daten dann veröffentlicht und weltweit allen OSM-Nutzern zur Verfügung gestellt.

JOSM ist nur ein Beispiel dafür, wie die Daten der *OpenStreetMap*-Datenbank genutzt werden können. Da die gesamten Geodaten des OSM Projekts kostenlos und frei sind, sind der kreativen Freiheit des Nutzers schier keine Grenzen gesetzt. Alle von den Benutzern hochgeladenen Daten unterliegen seit dem 12. September 2012 der *Open Database Licence (ODbL) 1.0*. Diese besagt, dass jegliche Art der Nutzung von OSM-Daten, auch gewerblich, zulässig ist, solange die Lizenzbedingungen eingehalten werden. So muss zum Beispiel bei einer auf Papier gedruckten Karte ein entsprechender Quellenhinweis (siehe Abbildung 2-1) angegeben werden.

```
Daten von <a href="http://www.openstreetmap.org/">OpenStreetMap</a> - Veröffentlicht
unter <a href="http://opendatacommons.org/licenses/odbl/">ODbL</a>
```

Abbildung 2-1: Quellenhinweis einer *OpenStreetMap*-Karte.

Abgeleitete Datenbanken dürfen unter der ODbL weiterverteilt werden.

2.2.3 Android

Android ist ein weit verbreitetes Betriebssystem für Mobilgeräte, das auf Smartphones, Tablets und auch manchen Netbooks zu finden ist.

2.2.3.1 Was ist Android?

Android ist ein Software Stack für Mobilgeräte, welcher Betriebssystem, benötigte Middleware[16] und vom System bereitgestellte Programme beinhaltet. Software Stack bedeutet, dass die verschiedenen Komponenten in einem Schichtenmodell aufgebaut sind, wobei die Programme, welche Applikationen oder auch App genannt werden, die oberste Schicht bilden. Darunter folgen die verschiedenen Systemkomponenten. Die unterste Schicht bildet der Linux Kernel [7].

[14] Homepage: http://josm.openstreetmap.de/
[15] Homepage: http://www.gimp.org/
[16] Eine Vermittlungssoftware, welche die Kommunikation zwischen den Prozessen unterstützt.

2.2.3.2 Eigene Applikationen programmieren

Für die Entwicklung von Android Applikationen werden von *Google* eine Reihe umfangreicher Tools bereitgestellt. Ein Beispiel dafür ist das *Android Software Development Kit* (Android SDK). Mit dem Android SDK können komplexe Anwendungen für eine große Anzahl verschiedener entwickelt werden. Das SDK bietet dafür eine Vielzahl von *Application Programming Interfaces*[17] (API) für den Zugriff auf die Hardware-Komponenten des Gerätes, wie zum Beispiel den GPS Sensor, Beschleunigungs- und Lagesensoren oder der integrierten Kamera [8].

Als Entwicklungsumgebung wurde in dieser Arbeit die *Eclipse*-IDE (*Integrated Development Environment*) der *Eclipse* Foundation[18] verwendet. *Google* stellt mit dem *Android Developer Toolkit* (ADT) ein Plugin für *Eclipse* zur Verfügung, welches viele Tools des SDKs direkt in die *Eclipse*-IDE integriert. Es bietet beispielsweise einen XML-Editor zum direkten Bearbeiten der Konfigurationsdateien und ein Tool zum intuitiven Erstellen grafischer Layouts per Drag and Drop. Das SDK beinhaltet dabei alle für die Entwicklung notwendigen Bibliotheken, Tools, Skripte, Dokumentationen und vieles mehr. Die Installation erfolgt leicht und schnell über die in *Eclipse* integrierte Updatefunktion zum Hinzufügen neuer Software. Über den Downloadmanager wird das Plugin in die IDE integriert. Das Android SDK beinhaltet auch einen SDK Manager, über den die Plattformen und Dokumentationen zu den bisherigen Android Releases einfach installiert werden können.

Zusätzlich beinhaltet das Android SDK auch ein *Android Virtual Device* (AVD) über dessen AVD-Manager man einen Android Emulator konfigurieren kann. Auch der Emulator ist über das ADT Plugin in *Eclipse* integriert, sodass der Programmcode auch ohne angeschlossenes Smartphone ausgeführt und getestet werden kann.

2.2.3.3 Vergleich mit Apple iOS

Obwohl viele Applikationen sowohl für Android als auch für Apple iOS verfügbar sind, gibt es wesentliche Unterschiede in der Entwicklung der Apps für das jeweilige Betriebssystem.

In der Android-Entwicklung wird Java als Programmiersprache genutzt. Somit kann auf viele freie IDEs, darunter auch das oben genannte *Eclipse*, zurückgegriffen werden. Viele davon unterstützen die Entwicklung mit integrierten, auf die Android-Entwicklung zugeschnittenen Plugins. Das entsprechende Plugin für *Eclipse*, sowie die für die Android-Entwicklung benötigte Android-SDK werden in 2.2.3.2 bereits genannt und beschrieben. Auch für die iOS-Entwicklung wird eine Entwickler-SDK benötigt. Im Gegensatz zur Android-SDK, ist die sogenannte iPhone-SDK nicht kostenlos und muss aus dem *Mac App Store*[19] geladen werden. Für die iOS-Entwicklung wird nicht Java, sondern Objective C als Programmiersprache verwendet. Daher stehen für die iOS-Entwicklung keine freien IDEs wie *Eclipse* zur Verfügung, sondern es muss eine eigene Entwicklungsumgebung benutzt werden. Diese Umgebung bietet ähnliche Tools und Funktionen, wie die Android-ADT. Eine weitere Ähnlichkeit ist der iOS-Simulator. Hier wird die Applikation aber nicht wie beim Android-Emulator für die jeweilige Smartphone-Hardware kompiliert, sondern für das System, auf welchem der Simulator läuft. Ein Vorteil hierbei ist, dass dies wesentlich schneller von statten geht, als die Kompilierung für den Emulator. Ein großer Nachteil ist, dass nicht sichergestellt werden kann, ob die

[17] Programmschnittstelle, über die Quelltext von anderen Programmen in das eigene Programm eingebunden werden kann.
[18] Homepage: http://www.eclipse.org/
[19] Von Apple betriebene Plattform, welche Software für den Benutzer zum Download bereitstellt.

Applikation auf dem Smartphone genauso ausgeführt wird, wie auf dem Simulator. Hier ist der Android-Emulator vorteilhafter, denn er gewehrleistet, dass die Applikation auch auf der eingestellten Hardware funktioniert.

Für diese Arbeit wurde die Entwicklung der Applikation in Android gewählt, da hier Java als Programmiersprache verwendet werden kann und somit auf alle für Java verfügbaren Bibliotheken und Funktionen zurückgegriffen werden kann. Zudem gewehrleistet Android den vollen Zugriff auf alle Framework-APIs, was es ermöglicht, auf die benötigten Hardwarekomponenten wie den GPS Sensor, die integrierte Kamera und den Speicher des Smartphones zuzugreifen.

2.2.4 OpenCV

Mit preisgünstigen und leistungsstarken PCs ist es heutzutage einfacher als jemals zuvor, anspruchsvolle Bildverarbeitungsanwendungen zu betreiben. Für jeden Benutzer, der seine eigenen Anwendungswerkzeuge zur Bildverarbeitung und Bildmanipulation entwickeln will, stellt OpenCV eine Vielzahl von Algorithmen bereit.

2.2.4.1 Was ist OpenCV?

Open Source Computer Vision (OpenCV), ist eine Open Source Bibliothek mit mehr als 2500 für die Bild- und Videoanalyse optimierten Algorithmen. *OpenCV* wurde schon im Jahr 1999 eingeführt und hat sich seitdem als Entwicklertool in Forschung und Entwicklung weit verbreitet. Ursprünglich wurde *OpenCV* von Intel unter der Leitung von Leiter Gary Bradski, als Initiative für die Forschung im Bereich der *Computer Vision*[20], für rechenintensive Bildverarbeitungsanwendungen entwickelt. Nach einer Reihe von Beta Releases wurde 2006 die Version 1.0 zur Verfügung gestellt. Im Jahr 2009 kam als große Neuerung ein Interface für C++ hinzu [9].

Die Algorithmen der *OpenCV*-Bibliothek sind für die Programmiersprachen C und C++ geschrieben. Mittlerweile gibt es aber viele Interfaces für *OpenCV*, was die Nutzung der optimierten Computer Vision Algorithmen in vielen Programmiersprachen ermöglicht. Darunter auch Python, Java, MATLAB oder Ruby, um nur ein paar zu nennen. *OpenCV* läuft dabei auf vielen gängigen Betriebssystemen, wie Windows, Linux, MacOSX, Android und vielen mehr.

Für diese Arbeit wurde das *OpenCV* Interface für Java namens *JavaCV* verwendet, zusammen mit OpenCV 2.4.3, welche die aktuellste verfügbare Version seit dem 3. November 2012 ist.

2.2.4.2 Bildverarbeitung mit OpenCV und JavaCV

Die von *OpenCV* zur Bildverarbeitung bereitgestellten Datenstrukturen und Algorithmen können in drei große Bereiche unterteilt werden:

1. **Kern**, *Core (Paket: open_cv.core)*
   Der Kern stellt die untere Komplexitätsebene dar und beinhaltet die Kernfunktionen von *OpenCV*. Hierunter finden sich auf die Funktionalität von *OpenCV* optimierte Datenstrukturen zur Darstellung der grundlegenden Geometrie, dynamische Strukturen zur Sequenzierung von Daten und zur Speicher-Reservierung. Auch einige auf die Strukturen von *OpenCV* angepasste, logische Grundoperationen, Listenoperationen oder Zeichenfunktionen finden sich hier.
2. **Bildverarbeitung**, *Image Processing and Computer Vision (Paket: open_cv.imgproc)*

[20] Deutsch: computerunterstütztes Sehen, maschinelles Sehen.

Die Bildverarbeitung ist die höhere Komplexitätsebene und beinhaltet Funktionen und Methoden zur Bildverarbeitung. Darunter befinden sich laufzeit- und speicheroptimierte Filterfunktionen, Methoden für geometrische Operationen, Werkzeuge zur Bildmanipulation, sowie einige Methoden zur Realisierung bekannter Objekterkennungsverfahren.

3. **Ein- und Ausgabe**, *Highlevel GUI (Paket: open_cv.highgui)*
   Der Ein- und Ausgabeteil dient der Interaktion des Programms mit den zu verarbeitenden Daten und beinhaltet diverse Methoden zum Einlesen und Schreiben von Bild- und Videodateien, sowie auch Methoden um auf Bildmaterial von Hardwaregeräten, wie beispielsweise einer Webcam, zugreifen zu können.

[10]

JavaCV adaptiert nicht alle, aber die gängigsten Computer Vision Algorithmen von *OpenCV* in die Programmiersprache Java. Die *OpenCV*-Methoden können somit wie gewöhnliche Java-Methoden verwendet werden und werden von *JavaCV* automatisch als nativer Maschinencode ausgeführt. Für die Bild- und Videoverarbeitung müssen auf dem System die erforderlichen Codecs vorhanden sein, auf die *JavaCV* zurückgreifen kann. Zur Verarbeitung von lokalen Video- oder Bilddaten verwendet *JavaCV* unter Linux standardmäßig Codecs aus der Sammlung von *FFmpeg*[21] und äquivalent dazu unter Windows die *Video for Windows*[22] (VfW) Codec-Sammlung. Je nach Anwendungsbereich bietet *JavaCV* die Möglichkeit, auf viele weitere Codecs zurückzugreifen. Beispiele dafür wären *libdc1394*[23] zur Kontrolle von IEEE 1394 basierten Kameras, *Fly Capture*[24] für den Zugriff auf Geräte über Firewire oder auch auf Codecs von *Open Kinect*[25], womit man die *OpenCV* Algorithmen auch auf den Sensordaten des *Xbox Kinect Sensor Device* anwenden kann.

[21] Homepage: http://www.ffmpeg.org/
[22] Homepage: http://msdn.microsoft.com/
[23] Homepage: http://damien.douxchamps.net/ieee1394/libdc1394/
[24] Homepage: http://www.ptgrey.com/products/pgrflycapture/flycapture_camera_software.asp
[25] Homepage: http://openkinect.org/wiki/Main_Page

3 Objekterkennung

Objekterkennung beschreibt ein Verfahren zum Identifizieren eines bekannten Objektes innerhalb eines Objektraums mittels optischer, akustischer oder anderen physikalischen Erkennungsverfahren [11]. Die Automatisierung dieser Verfahren findet heutzutage in vielen verschiedenen Bereichen Anklang.

3.1 Optische Objekterkennung

Das Hauptaugenmerk liegt in dieser Arbeit auf der optischen Erkennung. Am Anfang der optischen Objekterkennung steht, wie bei anderen Erkennungsmethoden auch, die Datensammlung. Die Objektumgebung wird dabei von Sensoren wahrgenommen und in Datenform gebracht. Dabei kann auf viele Arten von Sensoren zurückgegriffen werden. Neben elektromagnetischen Sensoren, Lagesensoren oder Drucksensoren gibt es auch lichtempfindliche, optische Sensoren. Zu diesen optischen Sensoren zählen neben den primitiven Helligkeits- oder Farbwertsensoren auch die komplexen Bildsensoren in Kameras. Diese müssen nicht immer Bildmaterial liefern, das man von Digitalkameras gewohnt ist und welches in etwa dem Wahrnehmungsspektrum des menschlichen Auges entspricht. Infrarotsensoren nehmen beispielsweise auch noch sehr langwellige Lichtspektren wahr, die für den Menschen nicht mehr sichtbar sind. Die optischen Sensoren wandeln dabei die Welleninformationen des Lichts, unter Ausnutzung von photoelektrischen Effekten, in auswertbare elektrische Signale um.

Eine wichtige Grundlage der optischen Objekterkennung, ist die für den jeweiligen Einsatzzweck qualitativ hochwertige Aufnahme der zu verarbeitenden Objektumgebung. Für die Erkennung von Straßenschildern wird daher eine hochauflösende Kamera benötigt, welche die Objektumgebung so wenig wie möglich verfälscht. Zudem soll auch bei wenig Lichteinfall kein Farbrauschen auftreten und die Linse der Kamera das Bild nicht verzerren. So können Straßenschilder schon aus weiter Entfernung gesichtet werden und befinden sich länger im Sichtfeld der Kamera, was die Chancen einer Erkennung erhöht.

3.1.1 Bildaufbereitung

Die Bildaufbereitung ist ein erster wichtiger Schritt der Bildverarbeitung und kann von dieser nicht wirklich abgegrenzt werden. Der Begriff wird hier lediglich zur besseren Übersicht von dem Begriff Bildverarbeitung getrennt. Bei der Bildaufbereitung werden die Eigenschaften der Bilder, wie der gespeicherte Farbwert der Pixel oder die Größe, so verändert, dass sie den benötigten Anforderungen des verwendeten Objekterkennungsalgorithmus entsprechen. So können für eine bessere Farbenerkennung beispielsweise die Farben des Bildes dahingehend manipuliert werden, dass die Farben in ihrer Intensität gesteigert werden oder aber auch, wenn für den Erkennungsprozess keine Farbe erwünscht ist, das Bild in ein Grauwertbild umgewandelt werden.

3.1.2 Bildverarbeitung

In der Bildverarbeitung geht es darum, wie in 2.1.1 schon beschrieben, die in dem Bild einer Objektumgebung gegebenen Informationen zu extrahieren und zu verarbeiten. Dabei gilt es, in einem Bild die Gebiete zu finden, an denen man interessiert ist. Einziger Anlaufpunkt sind hierbei die Informationen der einzelnen Pixelwerte des Bildes. Diese geben Auskunft über Farbe und Helligkeit eines Punktes in der Objektumgebung. Mithilfe verschiedener Methoden können diese Pixelwerte untersucht werden und somit Zusammenhänge, Ähnlichkeiten und Unterschiede der Pixel zueinander ausgewertet werden. So ist es möglich, Bereiche mit gleichen Eigenschaften in der

Objektumgebung zu finden, wie beispielsweise Flächen mit gleicher Farbe, starke Kontraste, die oft auf Kanten hinweisen oder auch bestimmte Pixelmuster, über die das gesuchte Objekt gefunden werden kann.

3.2 Methoden zur Objekterkennung/Erkennungsmechanismen

Es gibt viele Möglichkeiten ein gesuchtes Objekt in einem gegebenen Objektraum zu finden und es existieren dazu bereits viele verschiedene Methoden, welche Vor- und Nachteile mit sich bringen.

3.2.1 Template Matching

Eine Möglichkeit der Objekterkennung ist das Template Matching. Vereinfacht dargestellt, werden hierbei kleine Bildschnipsel, sogenannte Templates, auf denen das gesuchte Objekt abgebildet ist, mit dem Bild der Objektumgebung verglichen. Hier erkennt man schnell die ersten Probleme die man dabei zu lösen hat. Template Matching ist abhängig von der Skalierung und Rotation der Templates und auch der Objektumgebung. Darüber hinaus muss der Betrachtungswinkel des gesuchten Objekts mit dem des Templates übereinstimmen, sowie die Grauwertverteilung, sprich Helligkeit der einzelnen Pixel, aufgrund von Lichteinfall.

In den meisten Fällen findet das Template Matching auf Grauwertbildern statt. Es gibt aber verschiedene Strategien, um gegen die oben genannten Probleme vorzugehen. Damit die Matching Qualität nicht mehr von Helligkeitsabweichungen zwischen den Bildern beeinflusst wird, matcht[26] man auf den Kanten der Bilder.

Kanten sind markante Formmerkmale in Bildern. Beim Matching auf den Kanten der Bilder, ist die Helligkeit der Pixel nicht relevant, solange der Kontrast hoch genug ist, um Kanten finden zu können. Beim direkten Matching der Kanten müssen Bild und Template zu 100% übereinstimmen. Über ein distanzbasiertes Maß zwischen den Kanten, können auch ähnliche Objekte mit zueinander ähnlichen Kantenmerkmalen gematched werden. Die Extrahierung der Kanten aus dem Bild erfolgt beispielsweise mit Hilfe des *Canny*-Algorithmus zur Kantendetektion, dieser wird in Abschnitt 3.3.6 genauer erklärt.

Mit diesen Matching-Kriterien können nun zwei Bildausschnitte miteinander verglichen werden. Die beiden Ausschnitte sind ähnlich, wenn ihre korrespondierenden Kanten nah aneinander liegen. Hier findet man gleich ein weiteres Problem. Aufgrund verschiedener Umgebungseinflüsse können im Bild mehr Kanten gefunden werden, als im Template. Aber auch der umgekehrte Fall kann eintreten, dass durch verschiedene Lichtverhältnisse oder einer Überdeckung des Objektes im Objektraum, im Bild weniger Kanten gefunden werden, als im Template [12].

Template Matching ist für den Einsatz als Straßenschild-Erkennungsmethode nur sehr aufwendig zu realisieren und ist im Wesentlichen nur möglich, wenn man die Kanten des zu vergleichende Gebietes vorher anhand von Kanten mit hohem und Kanten mit niedrigem Interesse einschränkt, sowie die Templates unter verschiedenen Verzerrungen, Skalierungen und Rotationen mit den interessanten Gebieten der Objektumgebung vergleicht.

In der Masterarbeit von *Florian Janda* wird Template Matching zur Klassifizierung von Verkehrsschildern verwendet [13]. Zur Schilddetektion werden jedoch andere Methoden wie zum

[26] Prüfen zweier Bilder auf Übereinstimmung mit Hilfe von Template Matching.

Beispiel die Radiale Symmetrietransformation verwendet. Diese würden jedoch den Rahmen dieser Bachelorarbeit sprengen und so wurde von der Verwendung dieser Methoden abgesehen.

3.2.2 Feature Detection

Feature Detection ist eine auf definierten Eigenschaften (Features) basierende Erkennungsmethode. Diese kennt man zum Beispiel von der Gesichtserkennungsfunktion von Digitalkameras oder Smartphones. Hier kommt oft die sogenannte *Haar Feature Detection* zum Einsatz, da diese mit wenig Rechenaufwand betrieben werden kann. *Haar Feature Detection* arbeitet dabei mit sogenannten *Haar Features* oder auch *Haar like Features*, die mit ausgewählten Flächen im Bild verglichen werden.

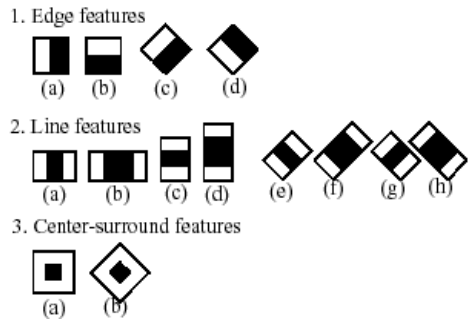

Abbildung 3-1: Einige für die Feature Detection benutzte Features. [14]

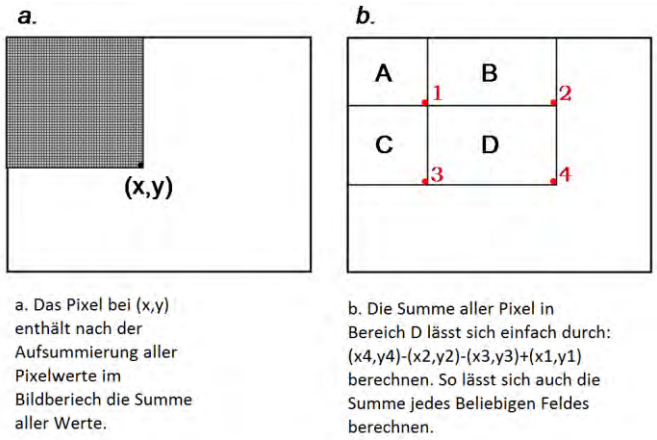

Abbildung 3-2: Funktionsweise eines Integralbilds (Integral Image). [15]

Häufig verwendet wird hierbei die Methode von *Paul Viola* und *Michael Jones*, welche nach vier Schlüsselprinzipien arbeitet:

1. Es werden einfache Rechteck-Features verwendet (vgl. Abbildung 3-1: 1. u. 2.).
2. Zur schnellen Durchschnittsberechnung der Pixelwerte wird ein Integralbild verwendet (Abbildung 3-2).
3. Um viele schwache Klassifizierer zu einem starken verbinden zu können wird eine *Adaptive Boosting* – Methode verwendet.
4. Die Klassifizierer werden kaskadiert, um eine effiziente Erkennung zu ermöglichen.

[15]

Neben der *Haar Feature Detection* gibt es noch viele weitere auf Merkmalen basierende Erkennungsmethoden. Nicht nur die Verwendung von fixen Merkmalen, wie im Beispiel der Gesichtserkennung, ist möglich. Die Merkmale können auch durch Detektoren aus den Bildern extrahiert werden. Der *Scale Invariant Feature Transform* – Algorithmus (SIFT – Algorithmus, zu Deutsch: *skaleninvariante Merkmalstransformation*) ist beispielsweise dazu in der Lage, über Kanten- und Eckenerkennungsmethoden, charakteristische Merkmale aus dem Bild zu extrahieren. Er ist dabei rotations- und skalierungsunabhängig und liefert bei gleichen Bildern immer gleiche Merkmale.

Die Extrahierung und auch die Erkennung von Merkmalen finden auf den Helligkeitswerten der Pixel, also auf Grauwertbildern, statt. Für die Erkennung von Straßenschildern würde eine sehr große Menge an zu untersuchenden Merkmalen benötigt werden. Außerdem wurde angenommen, dass

durch dieses Verfahren auch eine hohe Anzahl von Falscherkennungen produziert wird, beispielsweise durch Radkappen, Hausdächer oder andere Vorkommnisse, deren Form- und Helligkeitswerte denen von Straßenschildern sehr ähneln.

3.2.3 Farbenbasierte Objekterkennung

Um auf Bildmerkmale und deren Detektoren verzichten zu können, wird in dieser Arbeit auf andere Art und Weise vorgegangen. Anstatt Straßenschilder anhand ihrer markanten Ecken und Kanten zu suchen, wird auf eine wesentliche Eigenschaft von Straßenschildern zurückgegriffen. Denn im Gegensatz zu Objekten, die sich in Form und Beschaffenheit stark unterscheiden, sind die meisten Straßenschilder gleich und beruhen auf simplen geometrischen Formen. Eine weitere Eigenschaft von Verkehrsschildern ist es, dass sie im Vergleich zu anderen im Straßenverkehr auftretenden Vorkommnissen, sich durch ihre Form und Farbe vom Hintergrund abheben, um somit für den Fahrer deutlich erkennbar zu sein.

Diese speziellen Eigenschaften der Verkehrsschilder wurden in dieser Arbeit genutzt, um eine effiziente Methode zur Schilderkennung zu entwickeln. Die Erkennung lässt sich dabei in 3 Bereiche unterteilen:

1. **Bildaufbereitung und Filterung**
   In der Bildaufbereitung werden als erstes die Farbintensitätswerte der zu untersuchenden Bilder erhöht, um die Farben des Bildes besser auszuprägen. Somit heben sich die monochromen Verkehrsschilder noch besser von ihrer Umgebung ab. Zudem wird so durch die teilweise Eliminierung von zu hellen und zu dunklen Stellen ein Störfaktor ausgeglichen, welcher durch ungleichmäßige Lichtverhältnisse entsteht. Dies geschieht über Pixelwert-manipulation im HSV Farbraum, wie es unter Punkt 3.3.1.3 ausführlich beschrieben wird. Danach werden die aufbereiteten Bilder, mittels eines Schwellwertverfahrens (siehe Abschnitt 3.3.2), in Binärbilder umgewandelt. Das Binärbild stellt somit eine Filtermaske des im Schwellwertverfahren angewandten Farbwertes dar.
2. **Zusammenführung und Segmentierung**
   Nach der Filterung werden die einzelnen Farbmasken (Binärbilder) zu einem Binärbild zusammengeführt (siehe Abschnitt 3.3.3), welches nun eine Maske aller gefilterten Farben darstellt. Anschließend wird das Binärbild, anhand von zusammenhängenden Flächen, segmentiert (siehe Abschnitt 3.3.4) und diese Segmente der Formerkennung übergeben.
3. **Formerkennung**
   Als letzter Schritt werden die gefundenen Segmente mit Hilfe von Hough-Transformationen (siehe Abschnitt 3.3.7) auf die geometrischen Formen von Straßenschildern untersucht. Um die Erkennungsrate hierbei zu steigern, wird auf den Binärbildsegmenten vor der Hough Transformation ein Gaußscher Weichzeichner (siehe Abschnitt 3.3.5) angewendet, um eventuelle Lücken, die durch fehlerhafte Farbinformationen der Pixel der Originalbilder zustande gekommen sind, auszugleichen.

Wird ein Objekt am Ende des Prozesses mittels Hough-Transformation erkannt, besitzt es also eine der gefilterten Farben und bildet eine der straßenschildtypischen, geometrischen Formen ab. Es handelt sich bei dem Objekt somit mit hoher Wahrscheinlichkeit um ein Straßenschild.

3.3 Verwendete Methoden

Im Folgenden werden die verschiedenen Bildaufbereitungs- und Erkennungsmethoden erklärt, die in dieser Arbeit zur Verkehrsschilderkennung benutzt wurden.

3.3.1 Farbmanipulation über Farbräume

Ein Farbraum ist die Art der Darstellung der Farben der einzelnen Pixel in einem Digitalbild. Dabei werden die Farbwerte sowie andere Eigenschaften der Farben mit Hilfe von einfachen Werten oder auch Werte-Paaren repräsentiert.

3.3.1.1 Verschiedene Farbräume

Der primitivste Farbraum ist der Grauwertfarbraum. Dieser Farbraum hat nur einen Kanal, also nur einen Wert pro Pixel. Dieser Wert gibt Aussage über die Helligkeit des dargestellten Grautons. Bei einer Farbtiefe von 8 Bit ergeben sich für den Wertebereich eine Anzahl von $2^8 = 256$ verschiedenen Graustufen.

Der Farbraum, auf den man am häufigsten trifft, ist wohl der RGB beziehungsweise BGR-Farbraum. RGB steht für Rot (*Red*), Grün (*Green*), Blau (*Blue*) und steht für die Farben, welcher jeder einzelne der 3 Farbkanäle dieses Farbraums repräsentiert. Der BGR-Farbraum ist dabei äquivalent zum RGB Raum, nur sind hier die Kanäle für Rot und Blau vertauscht. RGB ist ein additiver Farbraum, was bedeutet, dass die einzelnen Farben des darstellbaren Farbspektrums durch additive Farbmischung der 3 Grundfarben Rot, Grün und Blau entstehen. Im RGB-Raum können bei einer Farbtiefe von $8\,Bit\,pro\,Kanal = 2^8 \cdot 2^8 \cdot 2^8 \approx 16{,}8\,Millionen$ Farben dargestellt werden. Ändert sich nur ein Wert des RGP-Tripels, so ändert sich somit auch die dargestellte Farbe in ihrem Farbton. Will man hingegen gewollt die Helligkeit oder die Farbsättigung des Bildes um einen definierten Wert ändern, so müssen aufwendig alle 3 Farbwerte dementsprechend richtig angepasst werden.

Anders ist dies in anderen, speziell auf diese Anforderungen angepassten Farbräume, wie beispielsweise dem HSV oder dem HSL-Farbraum. Der HSV-Raum beschreibt die dargestellte Farbe der einzelnen Pixel mit Färbung (*Hue*), Sättigung (*Saturation*) und Dunkelstufe (*Value*). Die Färbung(Farbton) beschreibt den Farbwinkel der Farbe (siehe Abbildung 3-3). Die Grundfarben sind dabei jeweils 60° zueinander versetzt und dazwischen befindet sich ein durch additive Farbmischung entstehender Farbübergang. Die Sättigung beschreibt die Vergrauung der Farbe. Bei einem niedrigen Wert vergraut die Farbe bis sie schließlich in einen Grauwert übergeht. Bei einem hohen Wert wird die Farbe intensiver. Die Dunkelstufe oder auch Intensität der Farbe beschreibt, wie der Name schon sagt, den Dunkelwert der Farbe. Bei einem niedriger werdenden Wert wird die Farbe dunkler und geht über in ein reines Schwarz, bei einem steigenden Wert wird die Farbe heller, bis hin zur reinen Farbe. Weiß kann im HSV Farbraum also nur durch eine entsättigte Farbe mit hoher Intensität dargestellt werden, wie es in Abbildung 3-3 zu sehen ist.

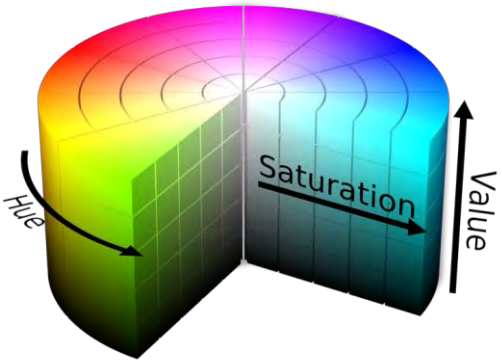

Abbildung 3-3: Darstellung des HSV Farbraums in einem Zylinderdiagramm [16]

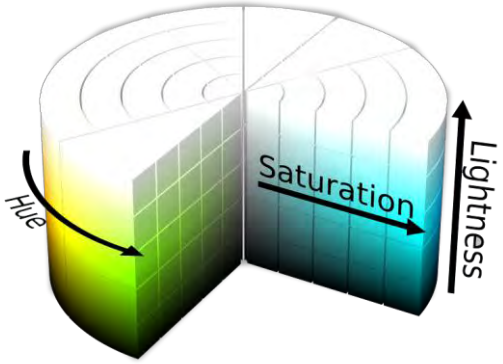

Abbildung 3-4: Darstellung des HSL Farbraums in einem Zylinderdiagramm [16]

Ähnlich zu HSV ist der HSL-Farbraum. Die Darstellung der Pixelwerte im HSL-Raum ist ähnlich der Darstellung im HSV-Raum. Auch hier werden ein Färbungswert (*Hue*) und ein Sättigungswert (*Saturation*) verwendet. Die Dritte Komponente ist hier aber nicht der Dunkelwert, sondern ein relativer Helligkeitswert (*Lightning*). Die Mitte des Wertebereichs des Helligkeitswertes ist in HSL die reine Farbe. Mit zunehmendem Wert verblasst die Farbe und geht über in reines Weiß, mit abnehmendem Wert verdunkelt die Farbe und geht über in ein reines Schwarz (siehe Abbildung 3-4).

Ähnliche Farbräume, welche auf demselben Prinzip basieren, wären der HSB-Farbraum, welcher den Wert der absoluten Helligkeit (*Brightness*) enthält und der HSI-Farbraum, welcher satt des Dunkelwerts den Lichtintensitätswert(*Intensity*) beschreibt.

3.3.1.2 Konvertierung von Farbräumen

Digitale Bilder können zum Teil über einfache Methoden von einem Farbraum in einen anderen übergeführt werden. Dabei ist dies oft nicht ohne Informationsverluste möglich, wenn die Farbräume zueinander nicht ‚kompatibel' sind.

RGB zu Grauwert

Die einfachste Konvertierung ist hierbei die Übersetzung eines Bildes aus dem RGB Farbraum in ein Grauwertbild. Eine Möglichkeit für die Konvertierung des Bildes wäre eine einfache Berechnung des Durchschnitts der RGB Werte zur Bestimmung des Grauwertes Y.

$$Y = \frac{R + G + B}{3}$$

Da die subjektiv wahrgenommene Helligkeit von Rot und Grün aber viel heller ist, als die von Blau, werden folglich blaue Gebiete zu hell wiedergegeben. Um die Gewichtung des jeweiligen Farbwerts herauszufinden, müsste die jeweilige Helligkeit der einzelnen Farben vorher gemessen werden. Die wahrgenommene Intensität ergibt sich dann aus:

$$Y = R \cdot Y_R + G \cdot Y_G + B \cdot Y_B \qquad mit \qquad Y_R + Y_G + Y_B = 1$$

Gute Näherungswerte wären zum Beispiel $Y_R = 0.299, Y_G = 0.587$ und $Y_B = 0.144$. Diese Formel gilt jedoch nur näherungsweise, da die subjektiv wahrgenommene Intensität nicht linear von den einzelnen Farbwerten abhängt [17].

Abbildung 3-5: Umwandlung RGB Bild zu Graustufenbild durch Setzen des Sättigungswerts im HSV Bild auf S=0.

Abbildung 3-6: Umwandlung RGB Bild zu Graustufenbild durch Berechnung des Durchschnitts der RGB Werte.

Dies ist zugleich ein Beispiel für eine unidirektionale Bildkonvertierung. Aufgrund des Verlustes der Farbinformation kann ein Grauwertbild nicht mehr in ein farbiges Bild gewandelt werden. Führt man das Bild trotzdem in den RGB Farbraum über, so bleibt das Bild schwarz-weiß, die einzelnen Pixel werden dann anstatt mit einem Grauwert mit einem RGB-Tripel dargestellt und es gilt:

$$R = G = B = Y.$$

Eine etwas schwierigere Methode ist es, das zu konvertierende Bild erst vom RGB in den HSV-Raum überzuführen um dann mittels Setzen des Sättigungswertes auf $S = 0$ das Bild in ein farbloses Bild zu wandeln. Wie in der Zylinderdarstellung des HSV-Farbraums (siehe Abbildung 3-3) schnell erkennbar ist, wird der Farbwinkel der einzelnen Pixel bei einem Sättigungswert von $S = 0$ irrelevant. Somit ergibt sich für die Rückrechnung von HSV zu RGB die Formel $R = G = B = V$ und der Grauwert Y kann direkt aus dem *Value*-Wert, also der Dunkelstufe, ausgelesen werden und es gilt $Y = V$. Selbiges gilt auch für den HSL Raum.

RGB zu HSV

Die Konvertierung eines RGB-Bildes in den HSV-Farbraum ist etwas aufwendiger als die Konvertierung in den Grauwertfarbraum, aber kann verlustfrei vollzogen werden. So erhält man nach einer weiteren Konvertierung des Bildes zurück in den RGB-Farbraum wieder das Ausgangsbild. Das Tripel (H, S, V) für die Pixelwerte des Bildes im HSV-Farbraum kann dabei wie folgt berechnet werden:

Die RGB Werte befinden sich in dem Intervall $[0, C_{MAX}]$.

Das Minimum der Werte ist $C_{min} = \min(R, G, B)$ und das Maximum $C_{max} = \max(R, G, B)$.

Der Wertebereich $\Delta C = C_{max} - C_{min}$ und die relativen Farbkomponenten ergeben sich aus

$$R' = \frac{C_{max} - R}{\Delta C}, G' = \frac{C_{max} - G}{\Delta C}, B' = \frac{C_{max} - B}{\Delta C}$$

Der Sättigungswert S lässt sich dann mit $S = \begin{cases} \frac{\Delta C}{C_{MAX}} & f\ddot{u}r\ C_{MAX} > 0 \\ 0 & sonst \end{cases}$ berechnen.

Und die Dunkelstufe (Value) V mit $V = \frac{C_{max}}{C_{MAX}}$.

Der Farbwert (Hue) H ergibt sich aus

$$H' = \begin{cases} B' - G' & \text{für } R = C_{max} \\ B' - B' + 2 & \text{für } G = C_{max} \\ G' - R' + 4 & \text{für } B = C_{max} \end{cases}$$

und der Normierung

$$H = \begin{cases} \frac{1}{6}(H' + 6) & \text{für } H' < 0 \\ \frac{1}{6}H' & sonst \end{cases}$$

[18]

HSV zu RGB

Die Rückrechnung eines Bildes im HSV Farbraum zurück in den RGB Farbraum kann man Folgendermaßen bewerkstelligen:

Als erstes berechnet man den Farbsektor: $H' = (6 \cdot H) mod\ 6$

Seien nun $v = V_{HSV}, s = S_{HSV}, c_1 = \lfloor H' \rfloor\ und\ c_2 = H' - c_1$, wobei V_{HSV} der Dunkelstufe (*Value*) ist und S_{HSV} der Wert der Sättigung (*Saturation*).

Weiter seien die Zwischenwerte x, y und z:

$$x = (1 - s) \cdot v$$

$$y = \left(1 - (s \cdot c_2)\right) \cdot v$$

$$z = \left(1 - \left(s \cdot (1 - c_2)\right)\right) \cdot v$$

Nun erhält man die Normalisierten RGB Werte im Intervall $[0,1]$:

$$(R', G', B') \leftarrow \begin{cases} (v, z, x)\ \ wenn\ c_1 = 0 \\ (y, v, x)\ \ wenn\ c_1 = 1 \\ (x, v, z)\ \ wenn\ c_1 = 2 \\ (x, y, v)\ \ wenn\ c_1 = 3 \\ (z, x, v)\ \ wenn\ c_1 = 4 \\ (v, x, y)\ \ wenn\ c_1 = 5 \end{cases}$$

Danach können die RGB Werte auf das Intervall $[0, C_{MAX}]$ skaliert werden [18]. Wobei C_{MAX} den maximal annehmbaren Wert in einem Farbkanal darstellt. Bei einer Farbtiefe von 8 Bit ist also $C_{MAX} = 255$.

3.3.1.3 Verwendung des HSV Farbraums

Ziel eines guten Farbmanagements ist es normalerweise, ein Bild das von einem Eingabegerät erfasst wurde, möglichst originalgetreu wiedergeben zu können. Bei der hier verwendeten Bildverarbeitung wird davon aber bewusst Abstand genommen und das Bild zur besseren Farbidentifikation aufbereitet.

Abbildung 3-7: Originalaufnahme des Straßenverkehrs

Abbildung 3-8: Maximierte Dunkelstufe (*Value*)

Abbildung 3-9: Maximierte Sättigung (*Saturation*)

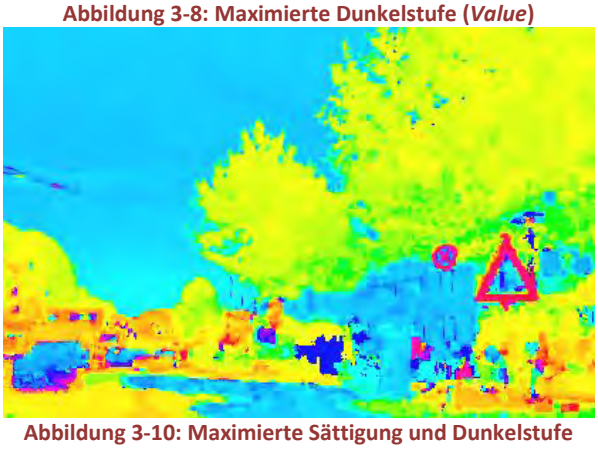

Abbildung 3-10: Maximierte Sättigung und Dunkelstufe

Zur Farbaufbereitung der aufgenommenen Bilder aus dem Straßenverkehr werden diese als erstes vom RGB in den HSV-Farbraum übergeführt. Durch einfache Manipulation des Werte-Tripels (H, S, V) kann nun die Helligkeit, Sättigung oder Farbwertverschiebung des Bildes beeinflusst werden. Wie man in Abbildung 3-7 - Abbildung 3-10 sehen kann, bewirkt die Manipulation eines Wertes eine nicht unerhebliche Veränderung des Bildes. Die gezeigten Bilder werden alle im RGB-Farbraum dargestellt.

Abbildung 3-7 zeigt die unveränderte Aufnahme im Original. Die Verkehrsschilder befinden sich unter den Baumkronen im Schatten, was eine Erkennung deutlich erschwert.

In Abbildung 3-8 wurde der Dunkelwert (*Value*) des Bildes maximiert, indem nach der Konvertierung des Bildes in den HSV-Farbraum der V Wert des Tripels (H, S, V) gleich dem maximal annehmbaren Wert gesetzt wurde. In einem Bild mit 8 Bit Farbtiefe also $V = 255$. Man sieht, dass sich die Farben der Schilder nun deutlich stärker von ihrer Umgebung abheben. Der schlechten Ausleuchtung, verursacht durch den Schattenwurf der Bäume, wird so ebenfalls entgegengewirkt.

Genau so wurde in Abbildung 3-9 verfahren, nur wurde hier anstelle des Dunkelwertes der Sättigungswert des Bildes maximiert. Auch hier sieht man die Vereinheitlichung der Rottöne im Bild, jedoch werden auch andere, rotähnliche Farbtöne in einen reinen Rotton gewandelt, wie man am linken Bildrand von Abbildung 3-9 erkennen kann. Darüber hinaus konnte die Verdunklung der Schilder durch den Schatten durch einen höheren Sättigungswert nicht ausgeglichen werden.

In Abbildung 3-10 wurden sowohl der Sättigungswert, als auch der Dunkelwert des Bildes maximiert. Hier wird wie in Abbildung 3-8 der Schatten eliminiert, jedoch auch rotähnliche Farbtöne in einen reinen Rot-Farbton gewandelt.

Es stellte sich heraus, dass die besten Ergebnisse bei der Farbfilterung, durch einen maximierten Dunkelstufen-Wert erzielt werden können. Nach der Manipulation dieses Wertes im HSV-Farbraum, werden die Bilder zur weiteren Verarbeitung wieder zurück in den RGB-Farbraum konvertiert.

3.3.2 Color Thresholding

Das *Color Thresholding* oder Schwellenwertverfahren beschreibt ein Verfahren zur Umwandlung eines Pixelwertes in einen Binärwert. Es ist ein sehr einfaches Verfahren, um Bilder in Bereiche mit derselben Farbinformation zu unterteilen. Das Schwellenwertverfahren wird angewandt um die aufgenommenen Bilder des Straßenverkehrs in Binärbilder umzuwandeln, welche nur noch Informationen über die gesuchten Farben enthalten. Dafür werden ein oberer und ein unterer Schwellwert in Form zweier (R, G, B) Tripel definiert.

Seien der obere Schwellwert durch $(r_{high}, g_{high}, b_{high})$ und der untere durch $(r_{low}, g_{low}, b_{low})$ definiert und (r, g, b) der Wert eines jeden Pixels des RGB-Bildes. Dann ergibt sich für den Wert y' des Binärbildes:

$$y' = \begin{cases} 1\ wenn\ (r < r_{high} \wedge g < g_{high} \wedge b < b_{high}) \wedge (r > r_{low} \wedge g > g_{low} \wedge b > b_{low}) \\ 0\ \ sonst \end{cases} \quad \text{(3.3.2-1)}$$

Da für den weiteren Verarbeitungsprozess 8 Bit Graustufenbilder verwendet werden, wird das Binärbild auf diesen Farbraum hochskaliert mit:

$$y = \begin{cases} 255\ wenn\ y' = 1 \\ 0 \quad\ wenn\ y' = 0 \end{cases}$$

Wobei y der Wert eines Pixels im Graustufenbild ist.

Abbildung 3-11: Schwellwertbild mit weitem Schwellwertbereich auf allen Farbkanälen.

Abbildung 3-12: Schwellwertbild mit schmalem Schwellwertbereich auf dem Rot-Kanal.

Mit geeigneten Schwellwerten können somit für die Schilderkennung uninteressante Bereiche aus dem Bild entfernt werden und es bleiben im Idealfall, wie in Abbildung 3-12 zu sehen ist, nur die gesuchten Schildformen übrig.

3.3.3 Merging

Will man verschiedenfarbige Schilder erkennen, müssen mehrere Filter gesetzt werden. Das aufgehellte Originalbild wird dabei mehrmals mit verschiedenen Schwellwerten im Schwellwertver-

fahren erst in ein Binärbild und danach in ein Graustufenbild umgewandelt. Eine Möglichkeit wäre nun, alle entstandenen Bilder einzeln in der weiteren Erkennung zu verarbeiten, jedoch ist dies zeitaufwendiger, als die Graustufenbilder zuvor zu einem Bild zu vereinen.

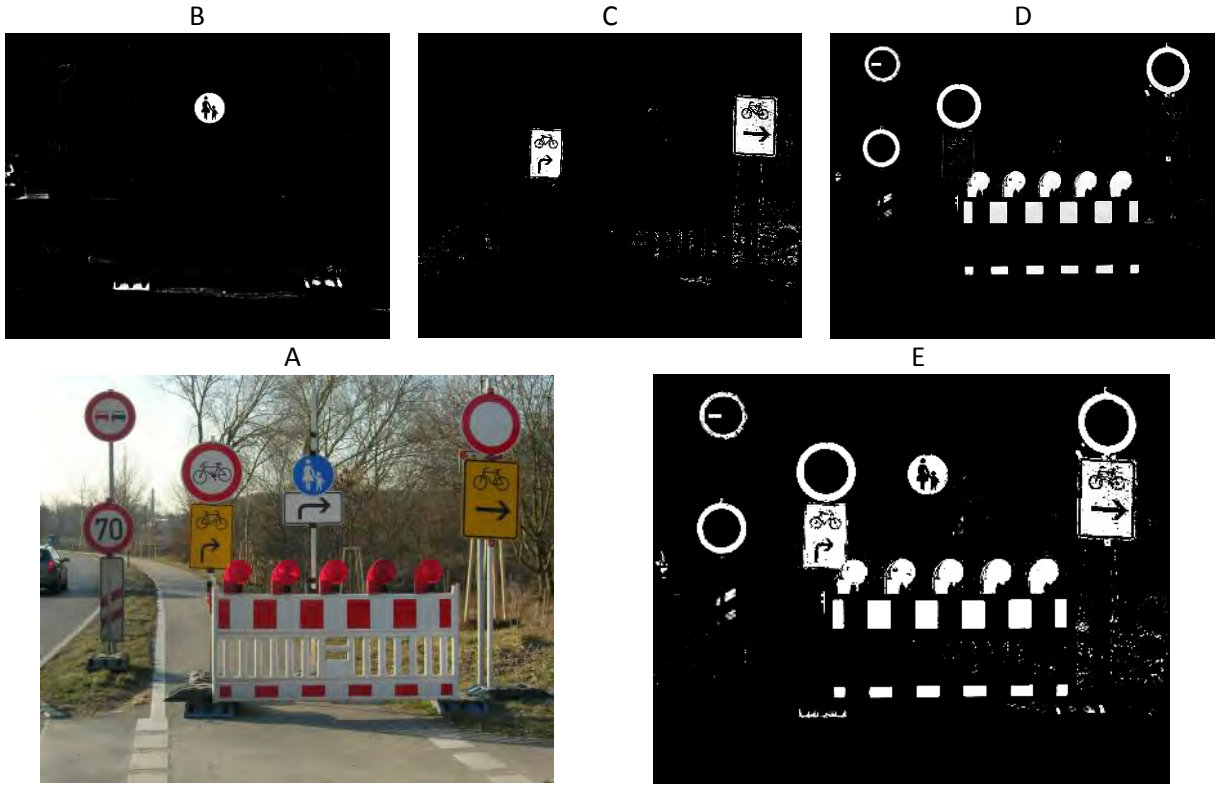

Abbildung 3-13: Bild A [31] ist das ungefilterte Originalbild. In Bild B wird ein Blaufilter verwendet, in Bild C ein Gelbfilter und in Bild D ein Rotfilter. Bild E ist das Summenbild aus B,C und D.

Die Vereinigung der Bilder geschieht nach einem einfachen Verfahren. Es werden dabei die Pixel aller Bilder nacheinander betrachtet und ihre Werte verglichen. Da es sich hier um aus Binärbildern skalierte Graustufenbilder handelt, ist der Pixelwert $y \in \{0{,}255\}$. Es genügt also zu untersuchen, bei welchen Pixeln $y \neq 0$ ist. Diese werden dann in ein Graustufenbild geschrieben. Abbildung 3-13 (E) zeigt das Graustufenbild, das durch die Vereinigung der Bilder B, C und D entstanden ist.

3.3.4 Bildsegmentierung durch *Blob Detection*

Nachdem die Filterbilder vereinigt wurden, folgt die Segmentierung des Bildes. Das Bild wird dazu mittels *Blob Detection* in mehrere kleine rechteckige Bilder unterteilt. Ein Blob oder auch Klecks, ist eine Menge zusammenhängender Pixel mit der gleichen Farbinformation. In dem Graustufenbild wird jedes Pixel untersucht, ob sein Farbwert $Y \neq 0$ ist. Ist dies der Fall, gehört es zu einem Blob. Ist der Wert eines unmittelbaren Nachbarn dieses Pixels auch ungleich 0, so gehört dieses ebenfalls zu diesem Blob. So verfährt man, bis keine unmittelbaren Nachbarn mehr dem Blob angehängt werden können. Nun kann ein Rechteck um diesen Blob gelegt werden, mit den Koordinaten:

$R(R_1, R_2)$ mit $R_{1_x} = \min(P_{1_x}, \dots, P_{n_x})$, $R_{1_y} = \min(P_{1_y}, \dots, P_{n_y})$, $R_{2_x} = \max(P_{1_x}, \dots, P_{n_x})$, $R_{2_y} = \max(P_{1_y}, \dots, P_{n_y})$. Wobei R_1 den Koordinatenpunkt der linken oberen Ecke und R_2 den Koordinatenpunkt der rechten unteren Ecke des Rechtecks R und $\{P_1, \dots, P_n\}$ die Menge der Koordinatenpunkte der Pixel des Blobs darstellen.

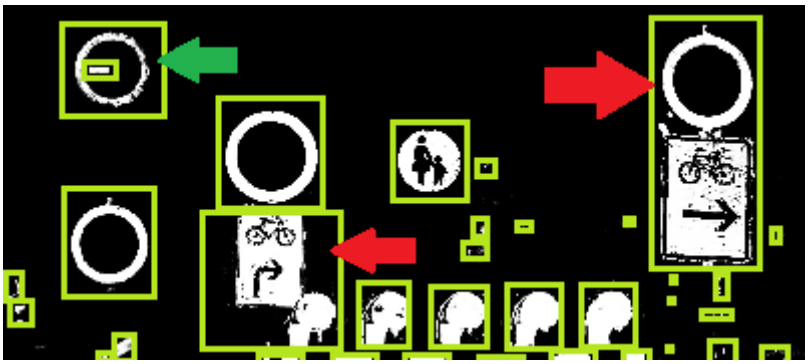

Abbildung 3-14: Bildausschnitt einer Verkehrssituation nach Farbfilterung mit erkannten Blobs.

Abbildung 3-14 zeigt einen Bildausschnitt von Abbildung 3-13 (E). Die grünen Rechtecke kennzeichnen die erkannten Blobs. Diese werden nun anhand ihrer Größe aussortiert. Zu kleine oder zu große Blobs werden nicht als Einzelbild extrahiert sondern verworfen. Nun kann es dazu kommen, dass mehrere, nah beieinander liegende Schilder oder ein Schild und ein gleichfarbiges Objekt in seiner Nähe zu einem Blob zusammengefasst werden, wie die roten Pfeile in Abbildung 3-14 zeigen. Dies stellt aber keinen Nachteil für die Schilderkennung dar. Da es in dieser Arbeit nur um die Detektion von Verkehrsschildern geht und nicht um deren Identifikation, genügt es also, wenn *ein* Schild in einem solchen Blob erkannt wird. Im Falle des rechten Blobs, welcher 2 Schilder enthält, erhöht dies sogar die Chancen einer Erkennung. Das erkannte Bild (siehe Abbildung 3-15) zeigt nun beide Schilder, die später vom Benutzer ausgewertet werden können. Der grüne Pfeil kennzeichnet zwei sich überlappende Blobs. Diese werden wie zwei nebeneinanderliegende Blobs behandelt, die nach der Erkennung einzeln segmentiert und weiterverarbeitet werden.

Abbildung 3-15: Bildsegment mit zwei Schildern.

3.3.5 Unscharfmaskierung

Die Unscharfmaskierung dient dazu, das Bild zu ‚verwischen'. Somit können die Kanten der Formen geglättet werden, wodurch diese Formen später besser erkannt werden können.

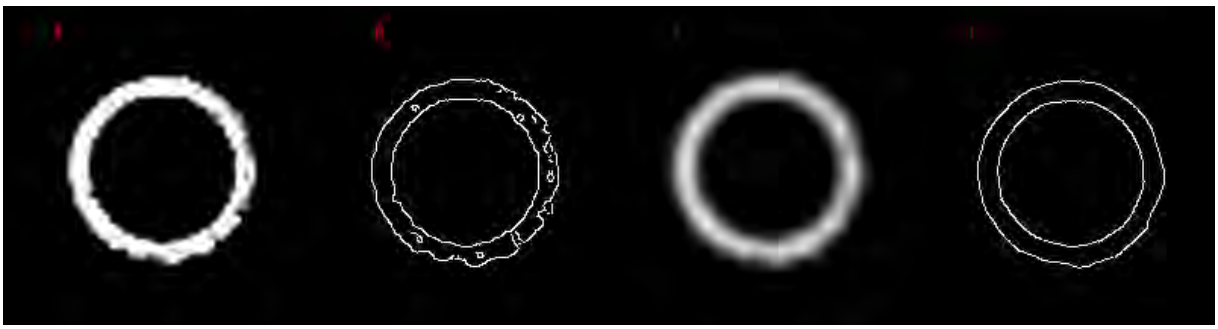

Abbildung 3-16: von links nach rechts: A: Grauwertbild eines Straßenschildes; B: Kantendetektion auf Bild A; C: Bild A mit angewandtem Weichzeichner; D: Kantendetektion auf Bild C.

Abbildung 3-16 zeigt ein segmentiertes Straßenschild aus Abbildung 3-13. Wie man in Bild A erkennt sind im Farbfilterprozess durch verschiedene Farbstörungen im Originalbild Lücken im Farbring des Schildes entstanden. Bild B zeigt das Segment nach einer Kantendetektion. Die Kanten sind, aufgrund der Pixellücken, ungleichmäßig und die Kreislinien stark gestört, wodurch eine Kreiserkennung mit

Hilfe der Hough-Transformation nur in seltensten Fällen gelingt. Lösen kann man dieses Problem mit einem Weichzeichner, welcher die Kanten glättet und die Pixellücken schließt. Bild C zeigt Bild A nach Anwendung eines *Normalized Box Filter* – Weichzeichners mit einer Kernel-Größe von 7x7 $Pixel$. Bild D zeigt eine Kantendetektion auf Bild C. Die Kreislinien sind hier gut ausgeprägt und können nun mittels Hough Transformation erkannt werden. Dasselbe Verfahren erleichtert auch die Linienerkennung bei eckigen Schildern.

Funktionsweise

Ein Weichzeichner manipuliert den Wert eines Pixels, anhand der Werte der Pixel in seiner nahen Umgebung. Eine einfache Methode dafür ist es den Wert eines Pixels mit Hilfe einer Linearkombination seiner Nachbarn zu berechnen. Das Pixel und seine Nachbarn werden dabei anhand eines Filter-Templates, auch Kernel genannt, gewichtet.

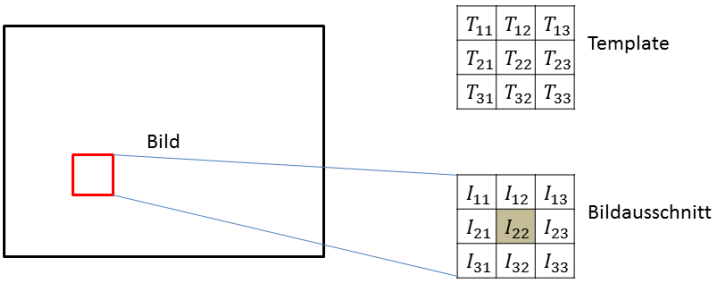

Abbildung 3-17: Ausschnitt eines Pixels mit seiner direkten Umgebung

Der Wert des Pixels I_{22} berechnet sich nun zu:

$$I_{22} = T_{11} \cdot I_{11} + T_{12} \cdot I_{12} + T_{13} \cdot I_{13} + T_{21} \cdot I_{21} + T_{22} \cdot I_{22}$$

$$+T_{23} \cdot I_{23} + T_{31} \cdot I_{31} + T_{32} \cdot I_{32} + T_{33} \cdot I_{33}$$

Für $I(x,y)$ ergibt sich also: $I(x,y) = \sum_i \sum_j I(x+i, y+j) \cdot T(i,j)$

Wobei $i\ und\ j$ die Anzahl der Zeilen und Spalten des Kernels darstellen [19].

In der Bildbearbeitung wird oft ein sogenannter *Gaußscher Weichzeichner* verwendet. Dieser liefert sehr gleichmäßig verwischte und störungsarme Bilder. Beim *Gaußschen Weichzeichner* wird zur Gewichtung der Pixelwerte die Normalverteilung, auch Gauß-Verteilung genannt, verwendet. Diese lässt sich mit der zweidimensionalen Gauß-Funktion berechnen [20]:

$$G(x,y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

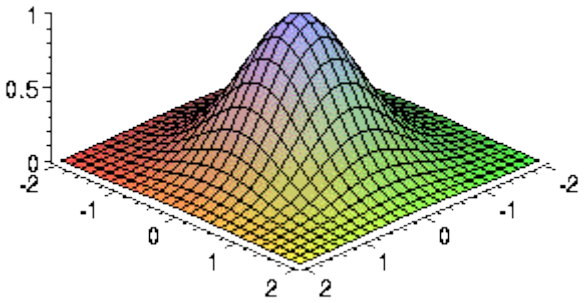

Abbildung 3-18: Graphische Darstellung der 2-Dimensionalen Gauß-Funktion. [21]

Die Gewichtung der Pixel ist im Zentrum des Kernels am höchsten und nimmt umso stärker ab, je weiter die Pixel vom Zentrum entfernt sind. Weit entfernte Pixel tragen demnach wenig zur Farbbestimmung bei.

Eine Filterung mit Hilfe des Gauß-Filters führt zu einem sehr guten Ergebnis, ist aber auch sehr rechenintensiv, weil dafür große Kernels gebraucht werden. Je größer der Kernel ist, desto mehr Zugriffe müssen pro Pixel auf die Pixel in seiner Umgebung durchgeführt werden. Es stellte sich heraus, dass solch eine Filterung im Rahmen der Verkehrsschilderkennung und unter der Verwendung von binären Graustufenbildern nicht notwendig ist. Deshalb wird für die Weichzeichnung der segmentierten Schildformen eine einfachere und schnellere Methode, namens *Normalized Box Filtering,* verwendet. Dabei werden alle Pixel aus dem Durchschnitt der Farbwerte ihrer Nachbarpixel im Bereich des Kernels berechnet. Der Kernel für das *Normalized Box* Filtering ist [22]:

$$T = \frac{1}{T_{Breite} \cdot T_{Höhe}} \cdot \begin{bmatrix} 1 & \cdots & 1 \\ \vdots & \ddots & \vdots \\ 1 & \cdots & 1 \end{bmatrix} \qquad \textbf{(3.3.5-1)}$$

Die Gewichtung aller Pixel ist dabei die Gleiche und die Summe der im Kernel befindlichen Gewichtungs-Werte ergibt den Wert 1, somit kann nach obiger Formel der Durchschnitt für $I(x, y)$ berechnet werden.

3.3.6 Kantendetektion

Damit man bei der Houghtransformation ein auswertbares Ergebnis erhält, muss auf den weichgezeichneten Bildsegmenten zuerst eine Kantenerkennung durchgeführt werden.

Ein bekannter und weit verbreiteter Algorithmus zur Kantenerkennung ist der *Canny*-Algorithmus, welcher bereits im Jahre 1986 von John Canny in dem Werk *‚A Computational Approach to Edge Detection‘* [23] veröffentlicht wurde. Der *Canny*-Algorithmus kann nur auf Grauwertbildern angewandt werden. Um die Effizienz zu steigern, sollten die Grauwerte dieser Bilder skaliert werden, damit der gesamte Farbraum ausgenutzt wird. Dies ist in den vorhergehenden Bildverarbeitungsschritten schon geschehen, somit kann direkt mit der Analyse der Gradienten begonnen werden.

Vereinfacht ausgedrückt, funktioniert der *Canny*-Algorithmus so, dass er die Stellen im Bild findet, bei denen sich die Intensität der Grauwerte stark verändert. Dazu werden die Gradienten der Grauwertverläufe des Bildes berechnet. Dies geschieht nach demselben Prinzip wie die in Abschnitt 3.3.5 gezeigte Unscharfmaskierung. Nur werden hier, anstelle eines neuen Pixelwertes, die

horizontalen und vertikalen Werte der Gradienten berechnet. Dies geschieht wie in Abschnitt 3.3.5 ebenfalls über Filtermasken.

Die Filtermasken für die horizontale Komponente G_x und die vertikale Komponente G_y sind:

$$G_x = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$$

$$G_y = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

Nun werden die Intensität G und die Richtung θ des Gradienten berechnet:

$$G = \sqrt{G_x^2 + G_y^2}$$

$$\theta = \arctan\left(\frac{G_y}{G_x}\right)$$

Als nächstes wird der Winkel θ auf den nächstgelegenen Winkel aus $\{0°, 45°, 90°, 135°\}$ gerundet [24]. Danach folgt eine Aussortierung der Maxima, wofür alle aneinander liegenden Pixel, deren Gradienten-Vektoren in dieselbe Richtung zeigen, untereinander verglichen werden. Der jeweils größte Intensitätswert bleibt erhalten, die anderen Werte werden verworfen [25]. Der letzte Schritt ist die Filterung der Werte mit einer Hysterese-Funktion. Dazu verwendet der Canny-Algorithmus 2 Schwellwerte, einen oberen und einen unteren. Liegt der Intensitätswert eines Gradienten über dem hohen Schwellwert bleibt er erhalten, liegt er unter dem unteren Schwellwert wird er verworfen. Die Werte, die zwischen dem unteren und dem oberen Schwellwert liegen, werden nur dann nicht verworfen, wenn sie mit einem Gradienten verknüpft sind, welcher über dem oberen Schwellwert liegt [24].

3.3.7 Hough-Transformation

Der letzte Schritt ist die eigentliche Erkennung der Straßenschilder. Die einzelnen Bildsegmente werden dazu mittels Hough-Transformation und geeigneten Detektions-Algorithmen validiert.

Die Grundidee der Houghtransformation ist, dass die gesuchten Objekte mit einer geeigneten Funktion vom Bildraum in einen Ergebnisraum übertragen werden sollen. Die Funktion muss dabei so gewählt werden, dass sie das gesuchte Objekt beschreibt. Im Idealfall bilden die gesuchten Objekte dann Häufungspunkte im Ergebnisraum. Die Komplexität hängt dabei von den freien Parametern der jeweiligen Funktion ab.

3.3.7.1 Linienerkennung

Ziel der Linienerkennung ist es, gerade Linien (Geraden) in den zuvor aufbereiteten Bildsegmenten zu finden, um dann anhand ihrer Schnittpunkte und Winkel zueinander eckige Schilder zu erkennen.

Geraden werden üblicherweise in einem kartesischen Koordinatensystem dargestellt. Das Problem hierbei ist, dass mit der Geradengleichung $y = a \cdot x + b$ keine zur x-Achse parallelen Geraden dargestellt werden können. Deshalb werden Geraden für die Houghtransformation in einem Polarkoordinatensystem dargestellt [13]. Eine Gerade wird in diesem durch 2 Parameter vollständig

beschrieben, zum einen durch den Abstand r, also die Länge des Lotes vom Koordinatenursprung auf die Gerade, und zum anderen durch den Winkel θ des Lotes zur x-Achse.

Die Geradengleichung kann also wie folgt aufgestellt werden:

$$y = \left(-\frac{\cos\theta}{\sin\theta}\right)x + \left(\frac{r}{\sin\theta}\right)$$

Nach Abstand r aufgelöst ergibt sich:

$$r = x \cdot cos\theta + y \cdot \sin\theta$$

Nun kann man für einen beliebigen Punkt (x_0, y_0) eine Familie von Geraden definieren, die diesen Punkt schneiden:

$$r_\theta = x_0 \cdot cos\theta + y_0 \cdot sin\theta$$

Jede Gerade die durch den Punkt (x_0, y_0) verläuft, wird nun von einem Paar (r_θ, θ) repräsentiert. Für ein gegebenes (x_0, y_0) kann nun die Familie von Geraden in den Ergebnisraum übertragen werden und man erhält eine sinusähnliche Kurve [26].

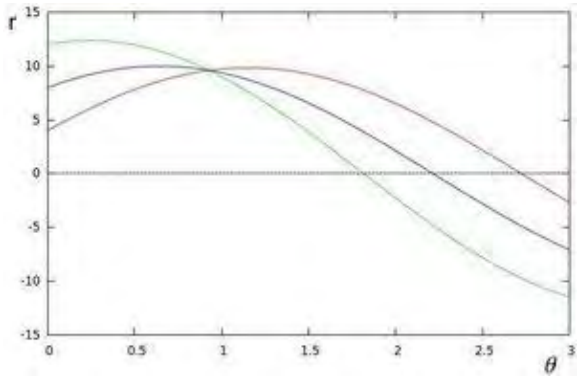

Abbildung 3-19: Kurven von Geraden-Familien im Ergebnisraum (r, θ) [26].

Dies macht man für jedes Pixel mit dem Grauwert $Y(x_0, y_0) \neq 0$ aus dem Bildsegment. Somit erhält man mehrere Sinusoide im Ergebnisraum. Scheiden sich zwei der Kurven in einem Punkt, so heißt das, dass die dazugehörenden Punkte (x_1, y_1) und (x_2, y_2) auf einer Geraden im Bildsegment liegen. Schneiden sich viele Kurven in einem Punkt (siehe Abbildung 3-19), ist die Wahrscheinlichkeit hoch, dass man eine Gerade im Bild gefunden hat. Die Koordinaten des Schnittpunkts im Ergebnisraum definieren die Gerade (r, θ).

3.3.7.2 Kreiserkennung

Die Houghtransformation zur Kreiserkennung funktioniert ähnlich wie die der Linienerkennung. Hier hat man anstelle von zwei, drei freie Parameter und einen dreidimensionalen Ergebnisraum. Zusätzlich zu den Koordinaten (x, y) kommt hier noch der Kreisradius hinzu. Hierbei spricht man auch oft von einer generalisierten Houghtransformation, da mit geeigneten Funktionen beliebige Formen im Bildraum gefunden werden können. Im Falle der Kreiserkennung können die Punkte auf einer Kreislinie, bei gegebenem Radius r, mit der Gleichung

$$x^2 + y^2 = r^2$$

berechnet werden. Die Vorgehensweise ist ähnlich der der Linienerkennung. Für jedes Pixel mit dem Grauwert $Y(x_0, y_0) \neq 0$ aus dem Bildsegment werden alle möglichen Kreismittelpunkte, die dieses Pixel auf ihrer Kreislinie beinhalten können, in den dreidimensionalen Akkumulator eingetragen. Dies geschieht für alle Radien $r \in [r_{min}, r_{max}]$. Häufungspunkte, also große Werte im Akkumulator, weisen darauf hin, dass viele Bildpunkte um den Punkt (x, y) mit Radius r liegen [27].

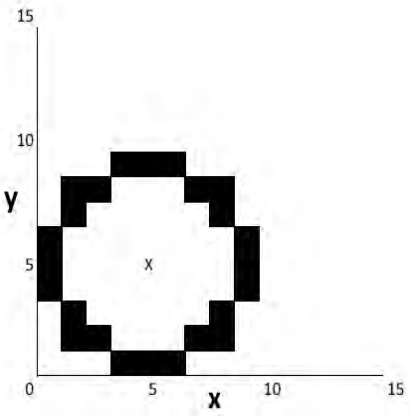

Abbildung 3-20: Einfache Darstellung eines Kreises in einem 15x15 Pixel Bildsegment.

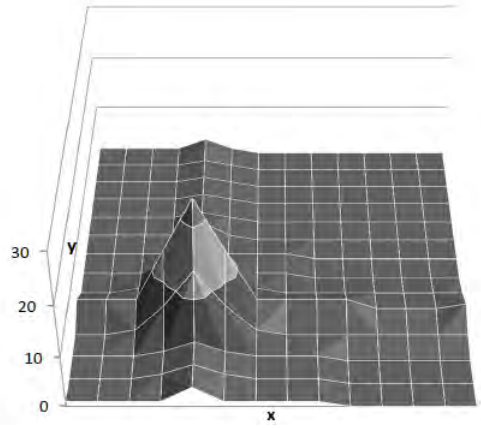

Abbildung 3-21: Ebene aus dem Akkumulator bei Radius $r = 5$.

Abbildung 3-20 zeigt das Beispiel eines Bildsegments der Größe 15x15 Pixel mit einem Kreis mit Radius $r = 5$. Für alle Punkte $\{(x, y) | x \in \{0, \dots, 15\}, y \in \{0, \dots, 15\}\}$ werden nun die möglichen Kreismittelpunkte in den Akkumulator eingetragen. Abbildung 3-21 zeigt die Ebene für Radius $r = 5$ aus dem Akkumulator-Raum. Es liegt ein Häufungswert im Punkt $(5,5)$ vor, weil viele Bildpunkte für diesen als Kreismittelpunkt für den Radius $r = 5$ gestimmt haben. Somit liegt der gesuchte Kreismittelpunkt bei den Koordinaten $(5,5)$ und der Kreis wurde erkannt.

3.3.7.3 Verkehrszeichendetektion

Nachdem Linien und Kreise erkannt worden sind, gilt es, daraus Rückschlüsse auf die Formen der Verkehrszeichen zu ziehen. Die Erkennung von runden Verkehrsschildern ist trivial. Wurde ein Kreis mittels Houghtransformation erkannt, gilt auch das runde Schild als erkannt. Die Erkennung der eckigen Schilder gestaltet sich etwas aufwendiger.

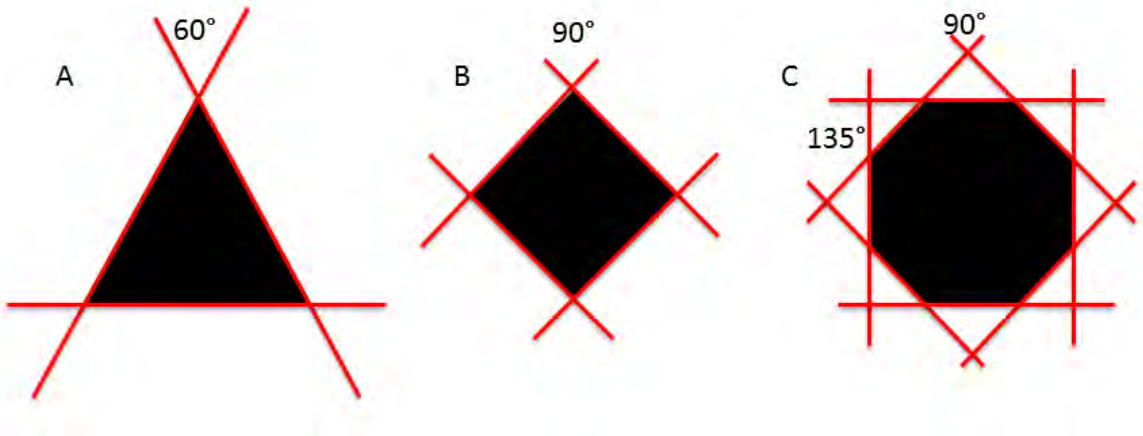

Abbildung 3-22: Schildformen und die Schnittwinkel ihrer Kanten.

Zur Erkennung von eckigen Straßenschildern werden die Schnittwinkel zwischen den gefundenen Geraden untersucht. Bei einem dreieckigen Schild (siehe Abbildung 3-22 (A)) schneiden sich die Geraden jeweils in einem spitzen Winkel von 60°. Dies kann man leicht untersuchen, da die Geraden bereits in Polarkoordinatenform mit den Parametern (r, θ) gegeben sind. Der Schnittwinkel α zweier Geraden $G_1(r_1, \theta_1)$ und $G_2(r_2, \theta_2)$ kann mit der Formel

$$\alpha = \big||\theta_1 - \theta_2| - \pi\big|$$

berechnet werden. Wenn sich 3 Geraden jeweils in einem Winkel von 60° schneiden, gilt das Dreieck als erkannt. Genau so verfährt man bei der Erkennung von Vierecken, nur müssen sich jeweils 2 Geraden in einem Winkel von 90° schneiden (siehe Abbildung 3-22 (B)). In einem Achteck schneiden sich die Geraden in mehreren verschiedenen Winkeln. Ordnet man die Geraden nach ihren Winkeln, schneidet sich eine Gerade mit der jeweils übernächsten immer in einem 90° Winkel. Auf diese Weise kann hier dasselbe Verfahren angewendet werden wie bei der Rechteckerkennung.

3.3.8 Tracking

Da es sich bei den zu verarbeitenden Bildern um Frames aus einem Video handelt, treten die Verkehrsschilder in sehr vielen Bildern hintereinander auf. Da ein auftretendes Schild aber auch nur als ein Schild erkannt werden soll, muss es getrackt[27] werden. Erkannte Schilder werden dafür in einer einfachen Form klassifiziert. Dabei wird nur zwischen runden, dreieckigen, viereckigen und achteckigen Schildern unterschieden. Die einfache Klassifizierung findet bereits in der Erkennung statt, da jede Erkennungsmethode nur eine bestimme Schildform erkennt.

Für die Schildverfolgung werden 2 verschiedene Tracking-Objekte verwendet, Tracking-Punkte und Tracking-Linien. Beide speichern 2 wichtige Informationen über das verfolgte Schild. Zum einen den Typ, also die Schild-Klasse und zum anderen einen Frame-Wert, wodurch das Alter der Objekte bestimmt werden kann. Abbildung 3-23 zeigt den Ablauf der Schilderverfolgung.

[27] Verfolgung eines Objekts mit Hilfe eines Trackingalgorithmus.

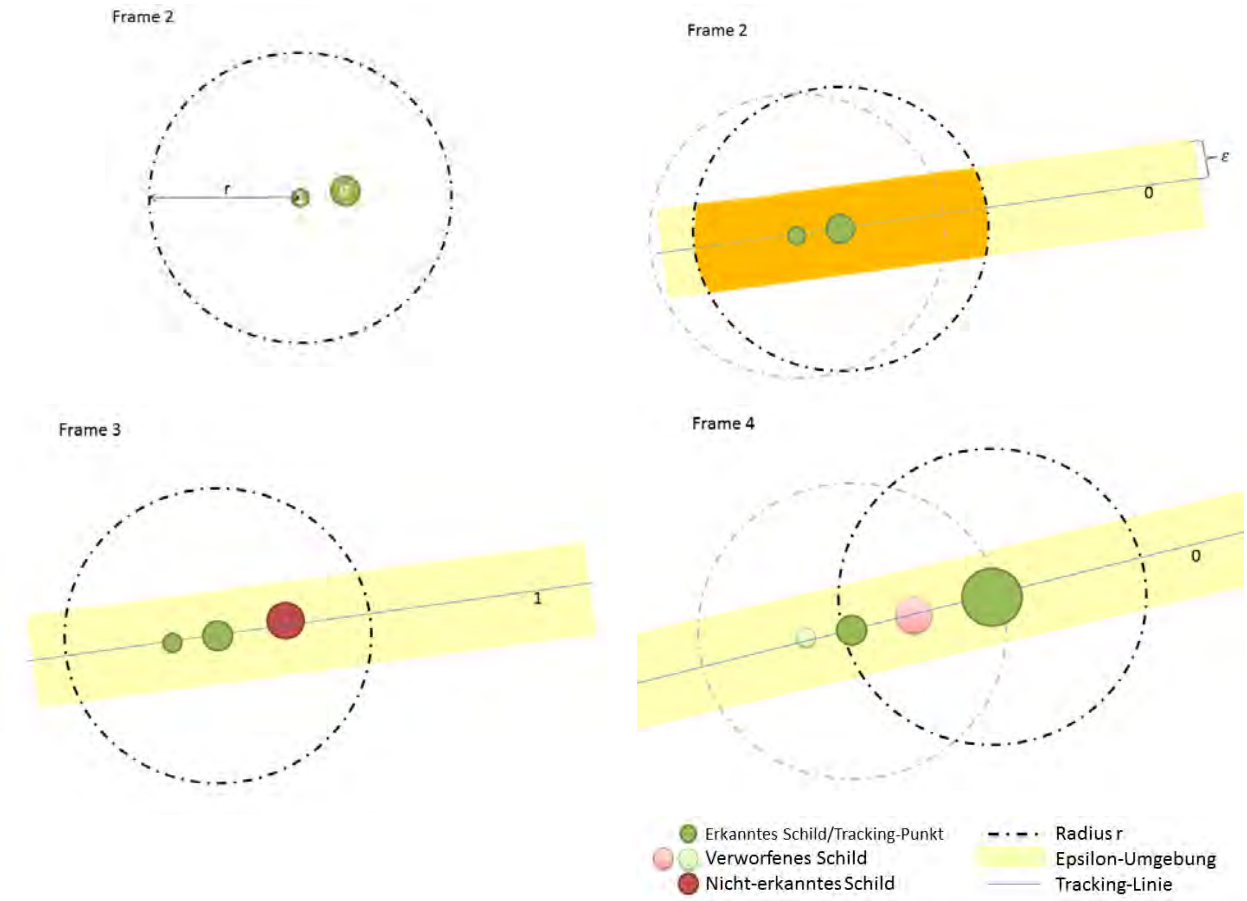

Abbildung 3-23: Grafische Darstellung des Schildverfolgungs-Algorithmus.

Die Schilder bewegen sich meist in einer geraden Linie oder einem leichten Kreisbogen von der Mitte in Richtung Rand des Bildes. Für die Schildverfolgung werden 2 feste Parameter gesetzt, der Erkennungsradius r um die Tracking-Punkte und die Erkennungsumgebung mit Abstand ϵ zu den Tracking-Linien. Die Verfolgung geschieht nach zwei einfachen Methoden:

1. **Point-to-Line**
   Jedes neu erkannte Schild wird über einen Tracking-Punkt dargestellt. Ist ein Punkt noch keiner Linie zugeteilt, wird in jedem neuen Frame nach einem neu erkannten Punkt desselben Typs im Radius r um diesen Punkt herum gesucht. Frame 2 in Abbildung 3-23 (links) zeigt einen Tracking-Punkt mit Alter 1. Dieser wurde bereits in Frame 1 erkannt. Der Tracking-Punkt mit Alter 0 wurde in Frame 2 frisch erkannt. Punkt 0 liegt im Tracking-Bereich von Punkt 1 und wird somit verfolgt. Dazu werden die beiden Tracking-Points in eine Tracking-Line umgewandelt. Diese Linie hat nun das Alter 0, da gerade ein neuer Punkt getracked wurde. Die Linie besitzt einen Tracking-Bereich mit Abstand ϵ zur Linie und einen Tracking-Radius um den ‚Kopf' der Linie, welcher sich an der Stelle befindet, an der zuletzt ein Schild getrackt wurde. Somit ist eine Richtung vorgegeben, in der das Schild weiter verfolgt wird.
2. **Line-to-Point**
   Befindet sich ein neuer Tracking-Punkt desselben Typs im Tracking-Bereich der Linie, wird dieser Punkt zur Linie hinzugefügt und die Koordinaten der Linie an den neuen Punkt angepasst. Der Tracking-Bereich der Linie ist der Schnitt zwischen der ϵ – Umgebung um die

Linie und dem Kreis um den ‚Kopf' der Linie mit Radius r (siehe Frame 2 in Abbildung 3-23 rechts). Der rote Kreis in Frame 3 stellt ein Schild dar, welches in den Erkennungsmethoden nicht erkannt wurde, somit befindet sich kein neues Schild im Tracking-Bereich der Linie und das Alter der Linie wird um 1 erhöht. In Frame 4 wurde ein neues Schild im Tracking-Bereich gefunden und zur Linie hinzugefügt. Die Koordinaten der Linie wurden dahingehend angepasst, dass die Linie nun durch ihren alten ‚Kopf' und dem neu gefundenen Punkt verläuft. Der neue Punkt stellt den neuen ‚Kopf' der Linie dar.

Ein Schild gilt als erfolgreich verfolgt und wird abgespeichert, wenn das Alter des dazugehörigen Tracking-Points bzw. der dazugehörigen Tracking-Line einen vorgegebenen Maximalwert überschreitet. Dies ist zum Beispiel der Fall, wenn das Schild den Objektraum verlässt, also die Kamera passiert hat.

4 Implementierung der Android Applikation GPStreetCam

4.1 Vorüberlegung

Das Ziel war es, mit *GPStreetCam* eine Android Applikation zu entwickeln, welche über die integrierte Kamera des Smartphones und den eingebauten GPS-Sensor, Daten der aktuell gefahrenen Straße sammelt. Dabei soll ein Video in der bestmöglichen Auflösung aufgenommen werden, was die spätere Auswertung der Bilddaten erleichtert. Anhand dieses Bildmaterials kann später durch das Java Programm *GPStreetTracker* eine automatische Objekterkennung durchgeführt werden. Zu dem Video soll, mittels GPS Sensor des Smartphones, die aktuelle Position einmal pro Sekunde aufgezeichnet werden. Mittels dieser GPS-Daten und der Frameposition des erkannten Objekts im Video, kann die Position der Kamera beim Passieren des Objekts bestimmt werden, was einen guten Näherungswert zur echten Position des Objekts darstellt. Die GPS-Daten sollen dafür in einem Format gespeichert werden, mit dem in der späteren Analyse effizient gearbeitet werden kann. Hierzu wurde das *GPS Exchange Format* (GPX-Format) gewählt, da dieses von den meisten GPS Anwendungen gelesen werden kann und sich dank der XML-Formatierung gut zur Weiterverarbeitung eignet. Die GPX-Datei soll zusammen mit dem Video im Medienspeicher des Smartphones abgelegt werden.

4.2 Erstellung der Android App GPStreetCam

Die Android Applikation soll die oben genannten Anforderungen erfüllen und soll dabei vor allem zweckmäßig sein. Für die Programmierung der Android Applikation wurde die *Eclipse*-IDE mit dem in Abschnitt 2.2.3.2 beschriebenen ADT-Plugin verwendet.

4.2.1 Videoaufzeichnung

Das Video wird mit dem von der Android SDK bereitgestellten **MediaRecorder** aus der **android.media**–Bibliothek aufgezeichnet. Hierzu müssen zuerst einige Konfigurationen vorgenommen werden. Eine Instanz der internen Smartphone Kamera muss geholt und an den **MediaRecorder** übergeben werden. Des Weiteren müssen in der Manifest-Datei der App alle nötigen Hardwarezugriffsrechte gesetzt werden. Als nächstes werden die für die Videoaufzeichnung wichtigen Parameter gesetzt. Dabei ist auf die genaue Reihenfolge der Angaben zu achten, damit der **MediaRecorder** fehlerfrei instanziiert werden kann. Die Reihenfolge für das Setzten der Parameter ist auf der Android Entwickler-Homepage [28] vorgegeben.

Als erstes wird die Kamera des Smartphones über eine sichere Methode instanziiert, da es sonst zu Laufzeitfehlern kommen kann, wenn andere Applikationen die Kamera gerade verwenden.

```
private static Camera getCameraInstance() {
        Camera c = null;
        try {
            c = Camera.open(); // attempt to get a Camera instance
        } catch (Exception e) {
            // Camera is not available (in use or does not exist)
        }
        return c; // returns null if camera is unavailable
    }
```

Quellcode 1: Sicheres Instanziieren der Smartphone-Kamera.

Um eine fehlerfreie und kompakte Darstellung der Rekorder-Ansicht der App gewährleisten zu können, ist die Bildschirmausrichtung fest auf *Landscape* eingestellt. Aus diesem Grund wird die Orientierung der Kamera direkt nach der Instanziierung auf eine Rotation von 90° gesetzt, damit die Bildvorschau in der Rekorder-Ansicht richtig ausgerichtet ist. Dies ist nur die Orientierung für die Kameravorschau, nicht aber für die Aufnahme. Die Orientierung für die Aufnahme muss im **MediaRecorder** gesetzt werden. Dies geschieht als letzter Schritt im Setup des **MediaRecorder**.

```
private int getRotation() {
        return degree > 0 && degree <= 90 ? 270
                : degree > 90 && degree <= 180 ? 0 : degree > 180
                        && degree <= 270 ? 90 : 180;
    }
public void onSensorChanged(int sensor, float[] values) {
        if (sensor == this.orientationSensor) {
            degree = values[0];
        }
    }
```

Quellcode 2: Berechnung des Rotationswinkels.

Die aktuelle Orientierung des Smartphones erhält man vom Orientierungssensor über die **onSensorChanged**-Methode des **SensorListener**-Interface aus der Bibliothek **android.hardware**. Die Rollneigung ist dabei ein Winkel zwischen 0° und 360°, welcher in der Methode **getRotation** bei Abruf auf einen von 4 Winkeln gerundet wird. Dabei wird immer auf den nächsten Dreiviertel-Kreis aufgerundet um einen korrekten Wert für die Video-Orientierung zu erhalten. Der errechnete Winkel wird als *Orientation Hint*[28] mit dem Video abgespeichert. Formate wie 3GP[29] oder MP4 unterstützen dies und so wird das Video von den meisten Playern richtig abgespielt. Da jedoch lediglich ein Hint gesetzt wird und nicht die Pixelmatrix selbst gedreht wird, sind bei einer Drehung des Smartphones auch die Rohdaten der Videos gedreht. Dies spielt jedoch keine Rolle für die Schilderkennung, da sich die geometrischen Formen der Schilder bei einer Drehung des Bildes nicht ändern.

Als Output-Format für den Encoder wird das statisch vordefinierte Kamera Profil **CameraProfile.QUALITY_HIGH** im **MediaRecorder** gesetzt. Mit dieser Option wird automatisch mit der maximal verfügbaren Auflösung aufgezeichnet, was mit heutigen Speicherkarten kein Problem mehr darstellt, und die spätere Auswertung, im Vergleich zu niedrig qualitativen Bildmaterial, deutlich verbessert.

Eine andere, wichtige Tatsache die es zu beachten gilt ist, dass der Android **MediaRecorder** Videos nur aufzeichnen kann, wenn eine dazugehörige Videovorschau auf der Benutzeroberfläche angezeigt wird. Für die Anzeige des Videos wurde die Klasse **CameraPreview** implementiert, welche von der Klasse **android.view.SurfaceView** abgeleitet ist. Die Klasse **SurfaceView** ist dabei selbst von der Klasse **android.view.View** abgeleitet und bietet eine dedizierte Zeichenoberfläche innerhalb einer View-Struktur. Das Format und die Größe dieser Struktur können dabei nach Belieben verändert werden. Um die korrekte Darstellung kümmert sich dabei die **SurfaceView** [29]. Des Weiteren implementiert die Klasse **CameraPreview** das Interface **android.view.Surfaceholder.Callback**, welches Methoden bereitstellt, die über die Änderungen des Surface Auskunft geben. So kann die

[28] Dieser Hinweis kann in manchen Videoformaten gespeichert werden und gibt dem Player Auskunft darüber, mit welcher Orientierung das Video abgespielt werden muss.
[29] Größtenteils auf MP4 basierendes Containerformat, welches oft von Mobilgeräten verwendet wird, die über eine geringe Videoauflösung verfügen. (unter 320x240)

Vorschau gestartet werden, wenn das Surface gestartet wird und gestoppt werden, wenn es gestoppt wird.

4.2.2 GPS-Aufzeichnung

Der zusätzlich zum Video, für die genaue Ortsbestimmung der Straßenschilder benötigte GPS-Track, wird mit Hilfe der Klasse **GPSLoactionListener**, welche das Interface **android.location.LocationListener** implementiert, aufgezeichnet. Dieses Interface enthält die benötigte **onLocationChanged**-Methode, die Auskunft über die aktuelle Position gibt.

```
private final class GPSLocListener implements LocationListener {

        public void onLocationChanged(Location location) {
            if (recording) {
                trackpoints = trackpoints
                        + ("<trkpt lat=\"" + location.getLatitude() + "\" lon=\""
                                + location.getLongitude() + "\"></trkpt> \n");
            }
```

Quellcode 3: Speicherung der GPS-Trackpoints für die GPX-Datei.

Für eine ausreichende Trackpoint-Dichte wurde eine Aktualisierungsrate von einem Trackpoint pro Sekunde im **LocationManager** des Smartphones gesetzt. Das Updateintervall wird in der ***LocationManager*.requestLocationUpdates**-Methode gesetzt. Der GPS-Track selbst wird dann in der bereits erwähnten **onLocationChanged**-Methode geschrieben. Hierzu werden die aktuellen Längen und Breitengrade aus dem **LocationListener** ausgelesen und im korrekten XML Format gespeichert (siehe Quellcode 3).

5 Implementierung GPStreetTracker

5.1 Vorüberlegung

Das Ziel von *GPStreetTracker* war es, ein Programm zu schaffen, welches dazu in der Lage ist, die von der App *GPStreetCam* aufgezeichneten Daten zu verarbeiten. Hierzu bietet *GPStreetTracker* ein Framework, in dem man mit erweiterbaren Plugins zur Objekterkennung, ein gegebenes Video analysieren kann. Die gefundenen Objekte können im Anschluss abgespeichert und ihre Lagepositionen in einer GPX-Datei gespeichert werden. Im Rahmen dieser Arbeit wurde ein Plugin namens *HoughSignRecognition* entwickelt, welches mit Hilfe von geeigneten Bildaufbereitungsmethoden, Filteralgorithmen und Hough-Transformationen die Formen von Straßenschildern erkennen kann.

5.2 Aufbau des Programms

Für die Implementierung des Programms wurde das Model-View-Controller-Pattern (MVC-Pattern) verwendet. In den folgenden Abschnitten wird genauer auf die Model-Schicht des Frameworks und des Plugins *HoughSignRecognition* eingegangen. Dabei werden die wichtigsten Klassen genannt und ihre Funktion kurz beschrieben, sowie der Ablauf des jeweiligen Programmteils, anhand der wichtigsten Methoden, kurz erklärt. Auf die View- und Controller-Schicht wird dabei nicht eingegangen.

5.2.1 Framework

Hauptkriterium für den Entwurf des Programms *GPStreetTracker* war es, ein Framework zu kreieren, welches eine Umgebung zur Erkennung von Objekten in den einzelnen Frames eines Videos bietet. Die erkannten Objekte werden dann, anhand von einem parallel zum Video aufgezeichneten GPX-Track, geographisch lokalisiert und als Waypoint in die GPX-Datei eingetragen. Zudem werden zu den geographischen Daten auch Bildausschnitte der erkannten Objekte gespeichert. Das Framework sollte leicht erweiterbar sein und dadurch die Möglichkeit bieten, nicht nur Schilder, sondern beliebige Objekte in den Frames eines Videos zu erkennen, zu identifizieren und zu extrahieren.

5.2.1.1 Einlesen der Frames und Objekterkennung

5.2.1.1.1 Wichtige Klassen

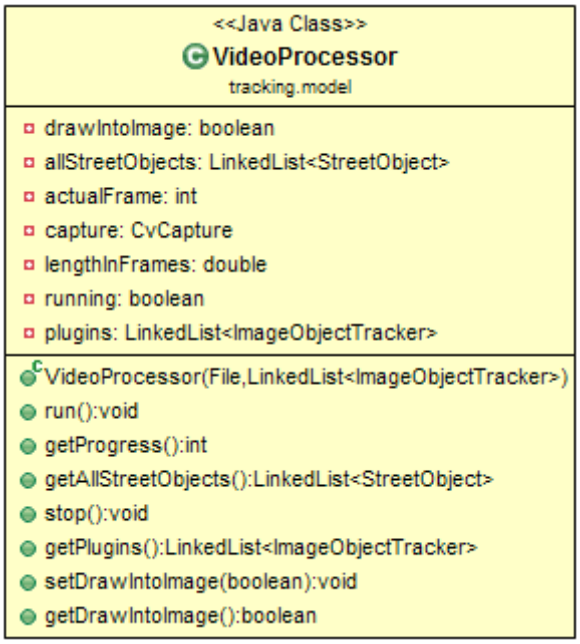

Abbildung 5-1: Die Klasse VideoProcessor.

class VideoProcessor
Beziehung: extends Observable, implements Runnable
Funktion: Einlesen und Verarbeiten der Video-Frames.
Beschreibung: Die Klasse VideoProcessor ist der Kern der Bildverarbeitung. Hier werden die einzelnen Frames nacheinander aus der gewählten Video-Datei extrahiert und mit den verfügbaren Plugins (*ObjectDetectionPlugin*) verarbeitet.

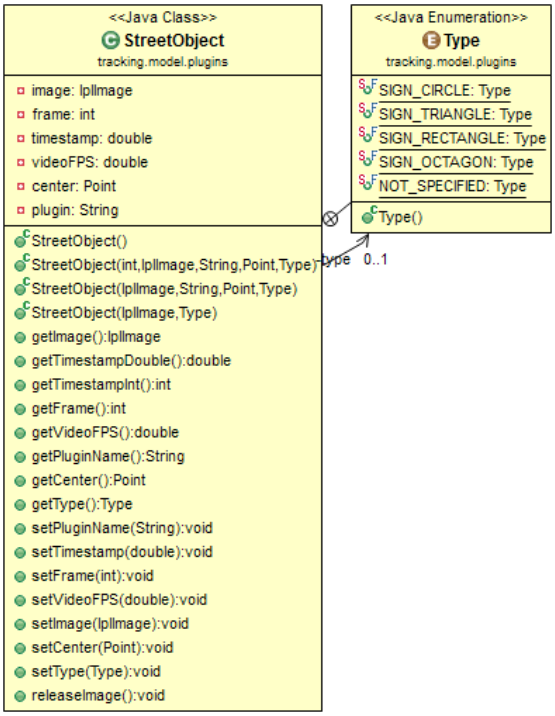

Abbildung 5-2: Die Klasse StreetObject.

class StreetObject
Funktion: Speicherung eines erkannten Objekts.
Beschreibung: Die Klasse StreetObject stellt ein erkanntes Objekt dar. Sie beinhaltet alle wichtigen Attribute und Informationen des Objektes.

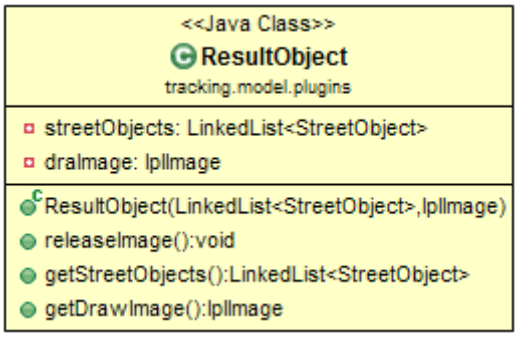

Abbildung 5-3: Die Klasse ResultObject.

class ResultObject
Funktion: Rückgabewert der Plugins.
Beschreibung: Die Klasse ResultObject dient der Kommunikation zwischen dem VideoProcessor und den Plugins. Sie enthält das bearbeitete Bild (*drawImage*), welches der Visualisierung Plugin-spezifischer Informationen dient, und eine Liste erkannter StreetObjects.

5.2.1.1.2 Ablauf

Vor der Videoverarbeitung wählt der Benutzer das zu verarbeitende Video und die Plugins aus, die er verwenden möchte. Nachdem alle Einstellungen in den Plugins getroffen sind, beginnt die Videoverarbeitung.

Der **VideoProcessor** extrahiert die einzelnen Bilder des gewählten Videos und reicht sie an die Plugins zur Verarbeitung weiter. Dazu wird das Video in ein **CvCapture** Objekt aus der **open_cv.highui**-Bibliothek geladen und über die statische Methode **cvQueryFrame** die Bilder nacheinander als **IplImage**-Objekte aus dem Video geladen. Der **VideoProcessor** besitzt eine Liste aller gewählten Plugins. Diese werden nacheinander abgearbeitet, indem bei jedem Plugin die **process**-Methode mit dem aktuellen Bild aufgerufen wird. Die erkannten **StreetObjects** werden, zusammen mit dem vom Plugin bearbeiteten Bild, in einem **ResultObject** zurück an den **VideoProcessor** gereicht. Dieser fügt die neu erkannten **StreetObjects** zu seiner Liste aller erkannten **StreetObjects** hinzu und schickt ein **update** mit dem nun von allen Plugins bearbeiteten Bild an die grafische Benutzeroberfläche.

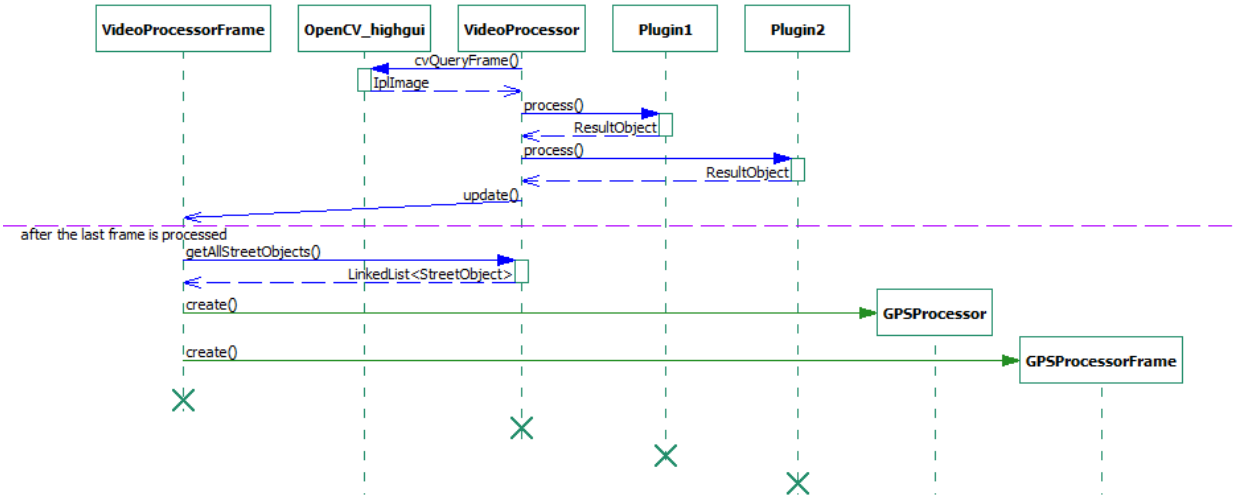

Abbildung 5-4: Das Sequenzdiagramm stellt das Einlesen und Verarbeiten der Video-Frames dar.

Abbildung 5-4 zeigt die Extrahierung eines Bildes aus dem Video und seine Verarbeitung in zwei Plugins. Nachdem das gesamte Video verarbeitet wurde, werden die gefundenen **StreetObjects** zu Ortsbestimmung und Speicherung an den **GPSProcessor** übergeben.

5.2.1.2 Verarbeitung der erkannten Objekte

5.2.1.2.1 Wichtige Klassen

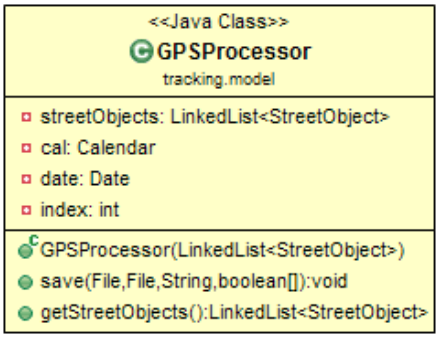

Abbildung 5-5: Die Klasse GPSProcessor.

class GPSProcessor
Beziehung: extends Observable
Funktion: Verarbeitung und Speicherung der gefundenen StreetObjects.
Beschreibung: Die Klasse GPSProcessor berechnet die Position der Schilder im GPX-Track, anhand der Nummer des Frames, in dem das Objekt gefunden wurde und speichert die StreetObjects ab.

5.2.1.2.2 Ablauf

Die in der Bildverarbeitung des **VideoProcessors** gefundenen **StreetObjects** werden im **GPSProcessor** in einem für den Benutzer zur Weiterverarbeitung geeignetem Format gespeichert. Die Berechnung der Position des Objekts im GPX-Track erfolgt über den Abgleich der Framenummer, in dem das Objekt gefunden wurde, mit der Framerate des Videos. Da die Aufzeichnungsrate des GPX-Tracks von einem GPS-Punkt pro Sekunde bekannt ist, kann die geographische Position des Aufnahmegeräts

zum Zeitpunkt der Erkennung des Objekts berechnet werden. Für die Bearbeitung der GPX-Datei werden Funktionen aus der *JDOM*-Bibliothek verwendet. *JDOM*[30] stellt eine komplette, Java-basierte Lösung für das Einlesen, Manipulieren und Schreiben von XML-Dateien bereit, was eine einfache Verarbeitung der GPX-Dateien ermöglicht. Für jedes Objekt wird ein Waypoint mit einem vom Benutzer gewählten Bezeichner angelegt und in die GPX-Datei geschrieben. Zusätzlich werden die gespeicherten Bilder der **StreetObjects** mit dem gleichen Bezeichner wie der zugehörige Waypoint in einem gewählten Verzeichnis im JPEG-Format gespeichert.

5.2.2 Plugins

5.2.2.1 Erweiterbarkeit des Framework durch Plugins

Das Programm *GPStreetTracker* bietet ein Framework, das die Extrahierung der Bilder aus dem Video und deren Speicherung regelt. Die Objekterkennung selbst findet in Plugins statt. Das Programm lässt sich leicht mit neuen Plugins erweitern, welche dazu das Interface **ObjectDetectionPlugin** implementieren müssen.

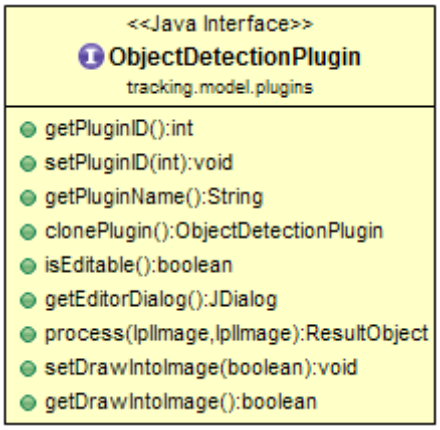

Abbildung 5-6: Das Interface ObjectDetectionPlugin.

Das Interface **ObjectDetectionPlugin** stellt die Schnittstelle zwischen Plugin und Framework dar. Über die vorgegebenen Methoden kann ein Plugin gesteuert und verwaltet werden.

Wichtige Methoden:

setPluginID(int):void, getPluginID():int dient der Identifikation der Plugins. Ein Plugin kann mehrfach für die Videoverarbeitung ausgewählt werden und es können auch mehrere Plugins mit demselben Namen verwendet werden. Bei der Plugin-Auswahl durch den Benutzer bekommt jedes Plugin vom Framework eine eindeutige ID zugewiesen.

getEditorDialog():JDialog liefert ein Einstellungsfenster zur Konfiguration des Plugins, wenn dieses eines bereitstellt.

process(IplImage, IplImage):ResultObject ist die Methode, über welche die Objekterkennung aufgerufen wird. Dem Plugin werden beim Aufruf zum einen der unveränderte Original-Frame des Videos und zum anderen ein ‚Zeichenbild' übergeben, welches auf der Benutzeroberfläche angezeigt wird. In dieses Bild können visuelle Informationen über die Objekterkennung an den Benutzer gezeichnet werden. Das Originalbild dient hingegen rein der Objekterkennung. Nach dem

[30] Homepage: http://www.jdom.org/

Erkennungsprozess werden gefundene Objekte als eine Liste von StreetObjects zusammen mit dem bearbeiteten Zeichenbild in einem ResultObject an den VideoProcessor übergeben.

5.2.2.2 Das Plugin HoughSignRecognition

Das Plugin *HoughSignRecognition* implementiert die in Abschnitt 3.2.3 vorgestellten Methoden zur farbbasierten, optischen Objekterkennung. Dabei wird, wenn möglich, auf in den *OpenCV*-Bibliotheken bereitgestellten Methoden und Objekte zurückgegriffen, da diese speziell für die optische Bildverarbeitung optimiert wurden und aufeinander abgestimmt sind. In diesem Abschnitt werden die wichtigsten Klassen der Model-Schicht des Plugins gezeigt und erklärt, sowie ein Überblick über den Ablauf einer Schilderkennung gegeben. Auf die View- und Controller-Schicht wird dabei nicht eingegangen.

5.2.2.2.1 Wichtige Klassen

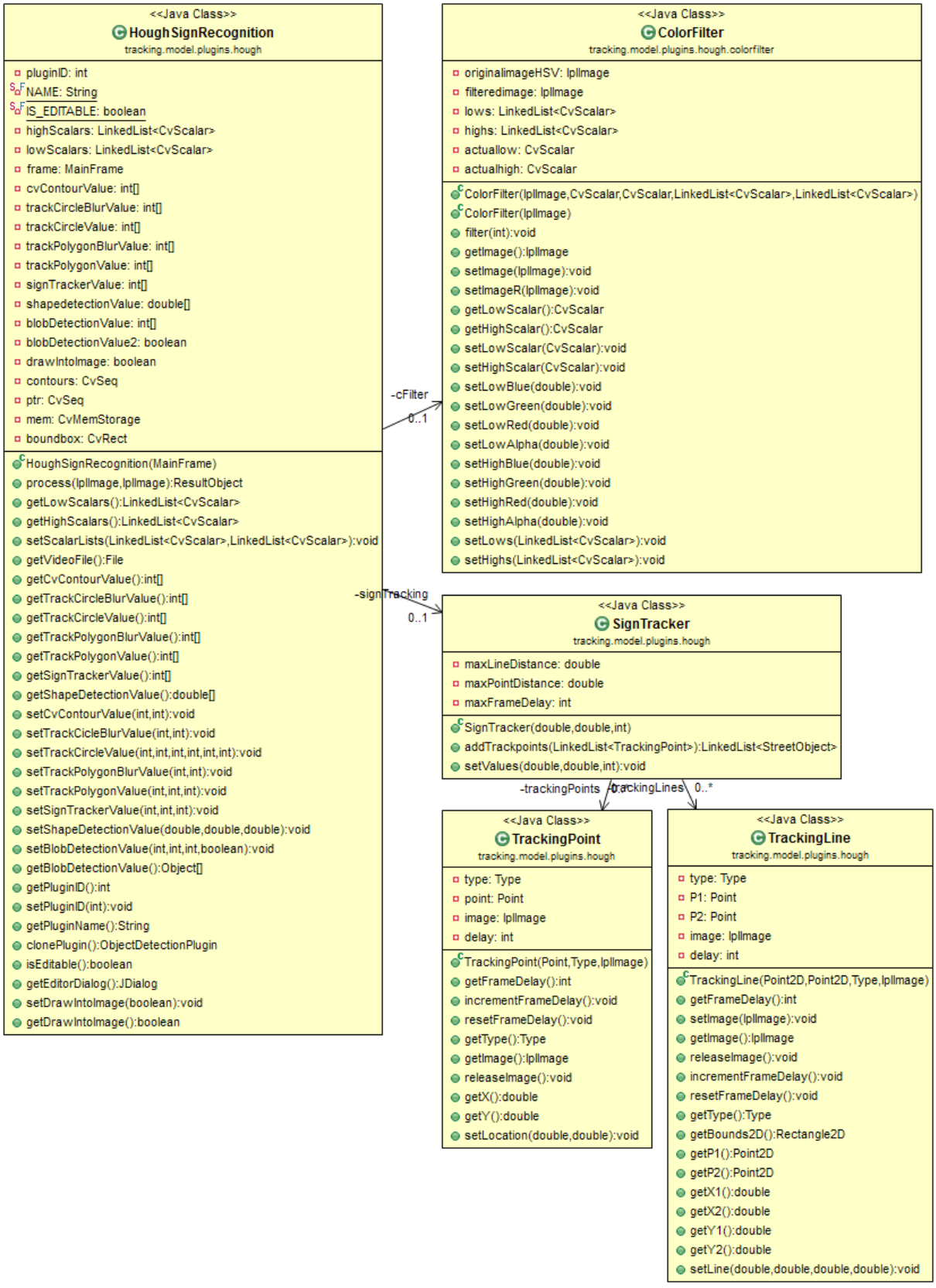

Abbildung 5-7: Hauptklassen des Plugins HoughSignRecognition.

class HoughSignRecognition
Beziehung: implements ObjectDetectionPlugin

Funktion: Zentrale Schnittstelle zwischen Framework und dem Plugin HoughSignRecognition.
Beschreibung: Die Klasse HoughSignRecognition ist die Hauptklasse des Plugins und die einzige Schnittstelle zum Framework. Sie implementiert das Interface ObjectDetectionPlugin und dient somit auch der Kommunikation mit dem VideoProcessor. Sie ist darüber hinaus auch Schnittstelle zum Einstellungs-Dialog des Plugins HoughSignRecognition und leitet die Einstellungen des Benutzers korrekt an die internen Strukturen weiter. Die Aufrufe der Schilderkennungsmethoden mit den korrekten Parametern, die Verwaltung der Schildverfolgung, sowie die Zusammenfassung der erkannten Straßenschilder in einen korrekten Rückgabewert, werden hier getätigt.

class ColorFilter
Beziehung: extends Observable
Funktion: Filterung der Bilder.
Beschreibung: Diese Klasse implementiert die in den Abschnitten 3.3.1 und 3.3.2 vorgestellten Methoden zur Farbfilterung. Sie filtert die zu untersuchenden Bilder, anhand der angegebenen Schwellwerte, und gibt ein Schwellwertbild zurück. Diese Klasse wird auch für die Einstellung der Farbfilter im Einstellungs-Dialog des Plugins HoughSignRecognition verwendet.

class SignTracker
Funktion: Verfolgung der Erkannten Straßenschilder.
Beschreibung: Die Klasse SignTracker implementiert die in Abschnitt 3.3.8 vorgestellte Methode zur Schildverfolgung. Dabei verwendet sie TrackingPoint und TrackingLine Objekte.

class TrackingPoint
Beziehung: extends Point2D
Funktion: Tracking-Punkt für die Schildverfolgung.
Beschreibung: Die Klasse TrackingPoint repräsentiert ein erkanntes Straßenschild und enthält, neben den 2D-Koordinaten an denen das Schild gefunden wurde, auch Informationen, wie Typ des Schildes und die Nummer des Frames, an dem das Schild entdeckt wurde.

class TrackingLine
Beziehung: extends Line2D
Funktion: Tracking-Linie für die Schildverfolgung.
Beschreibung: Die Klasse TrackingLine repräsentiert ein verfolgtes Straßenschild. Sie enthält wie die Klasse TrackingPoint Typ und Framenummer des Schildes und zusätzlich die Koordinaten einer Linie, die Bewegungsrichtung des verfolgten Schildes aufzeigt.

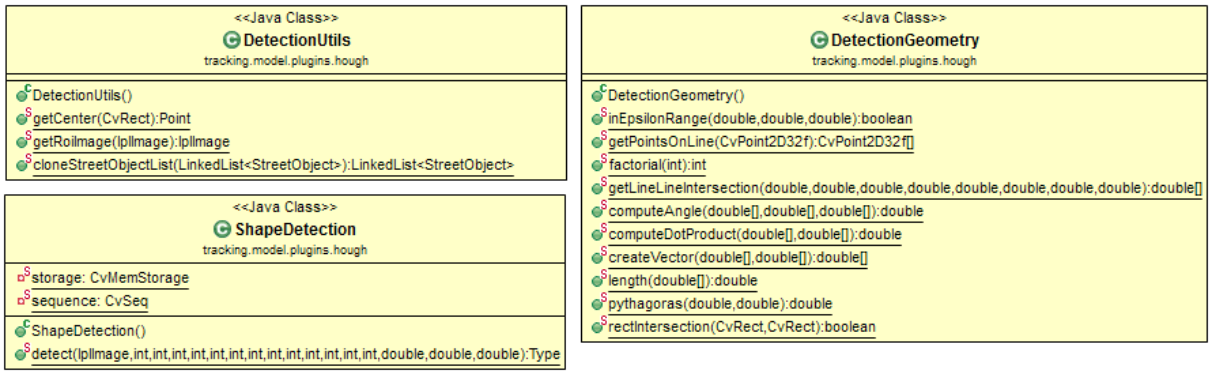

Abbildung 5-8: Statische Klassen mit Methoden zur Straßenschilderkennung.

static class DetectionUtils

Funktion: Stellt statische Methoden zur Verarbeitung der StreetObjects bereit.
Beschreibung: Die Klasse DetectionUtils stellt Methoden zur Extrahierung von bestimmten Bildbereichen(ROI) aus dem Video-Frame bereit.

static class DetectionGeometry
Funktion: Stellt statische Methoden für geometrische Rechenoperationen bereit.
Beschreibung: In der Klasse DetectionGeometry befinden sich Methoden, welche für Berechnungen auf den erkannten Linien benötigt werden.

static class ShapeDetection
Funktion: Stellt statische Methoden zur Straßenschilderkennung bereit.
Beschreibung: Die Klasse ShapeDetection implementiert die in Abschnitt 3.3.7.3 vorgestellten Methoden zur Formerkennung. Dabei werden auch Methoden aus den Klassen DetectionUtils und DetectionGeometry verwendet.

5.2.2.2.2 Ablauf

Der Ablauf einer Schilderkennung wird zur besseren Übersicht nur anhand der wichtigsten Schritte erklärt.

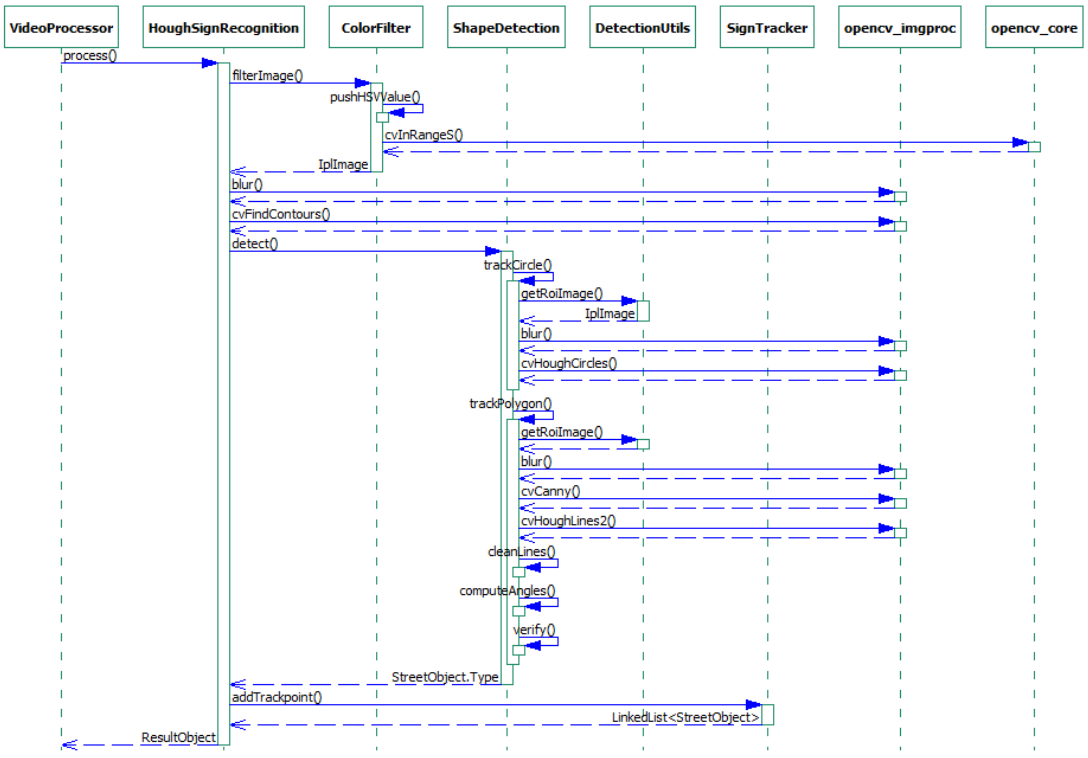

Abbildung 5-9: Das Sequenzdiagramm zeigt den Ablauf der wichtigsten Methodenaufrufe bei der Straßenschilderkennung mit dem Plugin HoughSignRecognition.

Abbildung 5-9 zeigt einen Überblick über die wichtigsten Methodenaufrufe des Plugins *HoughSignRecognition* bei der Verarbeitung eines Frames. Im Folgenden werden die verwendeten Methoden kurz erklärt, sowie genauer auf die Verwendung der *OpenCV*-Bibliotheken eingegangen.

Der **VideoProcessor** übergibt in der Methode **process(IplImage original, IplImage drawable)** zwei Bilder an das Plugin. Das **IplImage original** wird zur Straßenschilderkennung genutzt und geht weiter in die Bildverarbeitung, das **IplImage drawable** dient rein der Darstellung auf der grafischen Benutzeroberfläche, dazu später mehr. Der erste Schritt ist die Filterung des Bildes nach Farben, die der Benutzer über den Einstellungs-Dialog zuvor an das Plugin übergeben hat. Um eine besser Filterung und somit bessere Schilderkennung zu erreichen, wird im Bild wie in Abschnitt 3.3.1.3 beschrieben zuerst der Dunkelbereich-Wert maximiert.

```
private void pushHSVValue(IplImage src, IplImage dst) {
        cvCvtColor(src, dst, CV_RGB2HSV);
        ByteBuffer buff = dst.getByteBuffer();
        for (int i = 0; i < buff.capacity(); i++) {
            if (i % 3 == 2) {
                buff.put(i, (byte) (255)); // 255 max. unsigned byte value
            }
        }
        cvCvtColor(dst, dst, CV_HSV2RGB);
    }
```

Quellcode 4: Die Methode private void pushHSVValue().

Dies geschieht über die Methode **pushHSVValue**. Zur Konvertierung des Bildes vom RGB in den HSV-Farbraum und umgekehrt, wird die statische Methode **cvCvtColor(IplImage source, IplImage destination,int code)** aus der **opencv_imgproc**-Bibliothek verwendet, welche ein Bild ‚source' mittels der in Abschnitt 3.3.1.2 genannten Funktionen zwischen den oben genannten Farbräumen konvertiert und in Bild ‚destination' speichert. Der nächste Schritt ist die eigentliche Filterung der Bilder über Schwellwerte der einzelnen Farbkanäle. Zur Erzeugung des Schwellwertbildes wird die *OpenCV*-Methode **cvInRangeS(IplImage, CvScalar, CvScalar, IplImage)** verwendet. Diese Methode verwendet die Formel (3.3.2-1) zur Berechnung des Binärbildes und hat den Vorteil, dass sie direkt auf den Pixeldaten im Speicher operiert und somit sehr performant arbeitet. Nicht nur die Filterung findet im **ColorFilter** statt, sondern auch das Merging der Bilder. Das Merging geschieht einmal pro Filtervorgang. Der Benutzer kann eine beliebige Anzahl an Filterwerten setzen, diese werden nacheinander abgearbeitet und die entstandenen Binärbilder jeweils mit dem Haupt-Filterbild vereinigt (vergleiche Abschnitt 3.3.3). Dazu werden die Byte-Puffer (**ByteBuffer**) der beiden Bilder ausgelesen und miteinander verglichen. Alle weißen Pixel des neu gefilterten Bildes werden auch im Haupt-Filterbild als weiße Pixel gesetzt. Durch die direkten Lese-und Schreibzugriffe im Byte-Puffer des Bildes, wird auch hier eine hohe Performanz erreicht.

Das mit allen Filterwerten gefilterte Binärbild wird als **IplImage** zurück an die Klasse **HoughSignRecognition** übergeben. Als nächster Schritt wird ein Weichzeichner auf das Binärbild angewandt, um es für die in Abschnitt 3.3.4 beschriebene Methode zur Bildsegmentierung vorzubereiten. Als Weichzeichner wird die **open_cv.imgproc**-Methode

```
blur(IplImage source, IplImage destination, cvSize kernelSize, cvPoint anchor=
cvPoint(0,0), int borderType=0);
```

Quellcode 5: Methodenaufruf blur.

verwendet, welche die Formel (3.3.5-1) für das schnelle *Normalized Box Filtering,* zur Weichzeichnung des Bildes, implementiert. Die Methode liefert ein Grauwertbild, auf dem nun die Blob-Erkennung durchgeführt werden kann. Dazu wird ebenfalls eine **open_cv.imgproc**-Methode

```
cvFindContours(IplImage source, CvMemStorage mem, CvSeq contours, int
headerSize=Loader.sizeof(CvContour.class), int mode=CV_RETR_CCOMP,int
method=CV_CHAIN_APPROX_SIMPLE, cvPoint offset=cvPoint(0, 0));
```

Quellcode 6: Methodenaufruf cvFindContours.

verwendet. Diese bietet ein optimiertes und schnelles Verfahren zur Blob-Erkennung und liefert eine Sequenz von Punktsequenzen[31], welche die zusammenhängenden Punkte der Blobs enthalten. Die Methode **cvBoundingRect** liefert das kleinstmögliche Rechteck, welches die Menge einer Punkte, also den Blob, umschließt. Anhand dieser Rechtecke kann das Bild nun für die weitere Verarbeitung segmentiert werden. Damit der Benutzer ein Feedback über die gefundenen Segmente bekommt, werden die Rechtecke in das Zeichenbild **drawable** eingezeichnet.

Aus Performancegründen wäre es besser, das Bild nicht sofort zu zerteilen, sondern die von der Klasse **IplImage** angebotene Methode **setImageROI** zu verwenden und die einzelnen Rechtecke der Blob-Erkennung als ROI zu setzen. Somit müsste die Bildmatrix nicht umkopiert werden. Da jedoch einige der im Folgenden verwendeten Methoden der *OpenCV*-Bibliotheken diese Option nicht unterstützen, beziehungsweise ignorieren, werden die erkannten Blobs anhand ihrer umschließenden Rechtecke in einzelne neue Bilder kopiert. Dazu wird für jeden erkannten Blob die ROI im Bild gesetzt und das Bild über die Methode **detect** an die Klasse **ShapeDetection** übergeben. In der Klasse **ShapeDetection** findet die Schilderkennung mittels Hough-Transformation für Kreis- und Linienerkennung statt. Für beide Hough-Transformationen muss das Bild zuerst vorbereitet werden. Bei beiden wird dafür in einem ersten Schritt die gesetzte ROI aus dem Bild ausgeschnitten und als eigenständiges **IplImage** gespeichert. Das geschieht über die Methode **getRoiImage** der Klasse **DetectionUtils**, welche einen Teil der Bildmatrix anhand der ROI extrahiert und in einem eigenen **IplImage** des ausgeschnittenen Bereichs zurückgibt. Das ausgeschnittene Bildsegment wird nun ein weiteres Mal mit der Weichzeichenmethode **blur** aus der OpenCv-Bibliothek bearbeitet. Dabei wird für die Kreiserkennung eine andere Kernelgröße des Filters verwendet, als bei der Linienerkennung. Da die Hough-Transformation für Kreiserkennung nur perfekte Kreise erkennt, wird hier mit einem wesentlich größeren Kernel gefiltert, was zu einem stärker ‚verwischten' Bild führt. Auf diese Weise können auch schräg-stehende, runde Schilder noch als Kreis erkannt werden. Bei der Linienerkennung wäre dies kontraproduktiv, da die starke Weichzeichnung hier zu einer hohen Falscherkennungsrate führen würde. Deshalb wird bei der Hough-Transformation für Linienerkennung ein kleiner Kernel verwendet.

Zur Kreiserkennung wird die Methode **cvHoughCircle** aus der **open_cv.imgproc**-Bibliothek verwendet.

```
CvSeq sequence = cvHoughCircles(IplImage roiImage, CvMemStorage storage, int
method=CV_HOUGH_GRADIENT, int accumulator, minDist, int thresholdCannyEdge, int
thresholdCircleAcc, int minRadius, int maxRadius);
```

Quellcode 7: Die Methode cvHoughCircles.

Viele Arbeiten befassen sich mit der Beschleunigung der Hough-Transformation [30] und es werden immer schnellere Algorithmen entwickelt. Manche davon werden auch in den *OpenCv*-Bibliotheken bereitgestellt. Die Methode **CV_HOUGH_GRADIENT** verwendet eine gradientenbasierte Hough-Transformation. Die Methode arbeitet im Grunde nach dem in Abschnitt 3.3.7.2 vorgestelltem

[31] Eine *OpenCV*-Sequenz ist eine Listen-Datenstruktur, welche eine Menge von *OpenCV*-Objekten speichern kann. Eine Punktsequenz enthält somit eine Menge von Punkten.

Verfahren zur generalisierten Hough-Transformation. Jedoch wird die Anzahl der Kandidaten für einen Kreisradius nicht anhand der Pixelwerte, sondern mit Hilfe eines Gradientenbildes bestimmt. Die Gradienten werden dafür ähnlich dem in Abschnitt 3.3.6 vorgestelltem Verfahren zur Kantenerkennung bestimmt. Mit dieser Methode kann eine sehr schnelle Kreiserkennung durchgeführt werden. Wird ein Kreis erkannt, wird das Enum **StreetObject.Type=SIGN_CIRCLE** an die Klasse **HoughSignRecognition** zurückgegeben, ansonsten wird eine Linien-Erkennung auf dem Bild durchgeführt.

Zur Linienfindung wird die **open_cv.imgproc**-Methode **cvHoughLines2** verwendet.

```
CvSeq sequence = cvHoughLines2(IplImage roiImage, CvMemStorage storage, int
method=CV_HOUGH_STANDARD, double pixelResolution=1, double angleResolution=Math.PI / 180,
int lineThreshold, int param1NotUsed=0, int param2NotUsed=0);
```

Quellcode 8: Die Methode cvHoughLines2.

Die Methode **cvHoughLines2** arbeitet nicht auf den Gradientenwerten des Bildes, sondern wendet das in Abschnitt 3.3.7.1 beschriebene Verfahren an. Im Gegensatz zur **cvHoughCircles** ist die Kanten-Erkennung in dieser Methode nicht integriert und muss vor dem Aufruf separat durchgeführt werden. *OpenCV* bietet dafür die Methode **cvCanny** an.

```
cvCanny(IplImage roiImage, IplImage roiImage, int thresholdLow, int thresholdHigh,int
size=3);
```

Quellcode 9: Die Methode cvCanny.

Die **cvCanny**-Methode implementiert das in Abschnitt 3.3.6 gezeigte Verfahren zur Kantendetektion und speichert als void-Methode die gefundenen Kanten im **IplImage roiImage**, welches anschließend an die Methode **cvHoughLines2** übergeben wird. Diese liefert nach der Linien-Erkennung eine Sequenz von **CvPoint2D32f**-Punkten, welche die gefundenen Linien über die Werte $x = r\ (Länge\ de\ Lotes\ vom\ Ursprung\ auf\ die\ Gerade)$ und $y = \theta\ (Winkel\ des\ Lotes\ zur\ x - Achse)$ definieren. Die gefundenen Linien werden anschließend über verschiedene Methoden hinweg verarbeitet und am Ende verifiziert. Dieser Vorgang kann in 2 Hauptschritte unterteilt werden:

- Schritt 1 ist die ‚Säuberung' der Linien. Um eine gute Erkennungsquote zu erzielen, empfiehlt es sich, den Schwellwert der Linienerkennung niedrig zu setzen. Somit steigt die Chance an, eine Linie zu erkennen, jedoch werden oft ganze Geradenbüschel erkannt (siehe Abbildung 5-10 (A)).

A B C

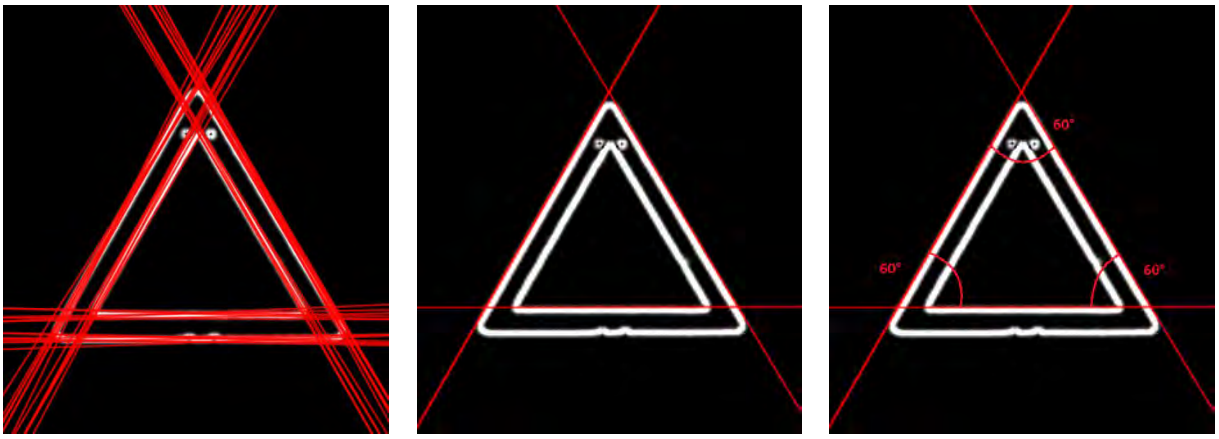

Abbildung 5-10: A zeigt die gefundenen Geradenbüschel nach der Linienerkennung. In B wurden die Büschel auf jeweils eine Mittelwertgerade reduziert. C zeigt die Winkel zwischen den Geraden.

Um eine Polygon-Erkennung auf den gefunden Geradenbüscheln durchführen zu können, müssen diese zuerst auf einen einfachen ‚Grundriss' reduziert werden. Dies geschieht in der private Methode **cleanLines** der Klasse **ShapeDetection**. In dieser Methode werden die gefundenen Geraden erst nach ihren Winkeln sortiert. Danach werden alle Geraden, die einen Winkel kleiner dem vorgegebenen Maximalwert ϵ zueinander haben, zu Gruppen zusammengefasst, welche dann nach dem Abstand θ sortiert werden. Nun werden sie nochmals in kleinere Gruppen zerlegt, doch diesmal werden alle Geraden gruppiert, welche einen Abstand kleiner dem vorgegebenen Maximalwert δ zueinander haben. Jede Gruppe enthält nun zueinander sehr ähnliche Geraden. Basierend auf Abstand und Winkel wird nun ein Durchschnittswert für jede Gruppe berechnet und die Gerade ausgewählt, die diesem am nächsten ist. Alle anderen Geraden werden verworfen und so bleibt nur eine Gerade pro Gruppe übrig. Abbildung 5-10 (B) zeigt das Ergebnis der **cleanLines**-Methode.

- Schritt 2 ist die Verifizierung der erkannten Linien. Dazu werden zuerst die Schnittwinkel zwischen den übrigen Geraden berechnet (siehe Abbildung 5-10 (C)) und anschließend mit der in Abschnitt 3.3.7.3 vorgestellten Methode untersucht. Wird ein Schild erkannt, wird das Enum **StreetObject.Type=SIGN_TRIANGLE**, **StreetObject.Type=SIGN_RECTANGLE** oder **StreetObject.Type= SIGN_OCTAGON**, je nach erkannter Schildform, an die Klasse **HoughSignRecognition** zurückgegeben.

Wird weder in der Kreis-, noch in der Linienerkennung ein Schild erkannt, erhält die Klasse **HoughSignRecognition** den Rückgabewert **StreetObject.Type=NOT_SPECIFIED**. In diesem Fall wird ein leeres **ResultObject** zurück an den **VideoProcessor** gegeben. Bei einer positiven Erkennung wird ein neuer **TrackingPoint** mit den Informationen des Schildes erstellt. Es wird gespeichert, wo im Bild das Schild gefunden wurde, um welchen Typ es sich handelt und es wird der Bildteil aus dem Originalbild kopiert, welcher dem Segment entspricht, indem das Schild gefunden wurde. Diese Informationen werden später in der Schildverfolgung benötigt. Zusätzlich wird der Typ eines erkannten Schildes, als Text in das entsprechende Segment in das Zeichenbild geschrieben, damit der Benutzer die Erkennung mitverfolgen kann.

Nachdem alle Bildsegmente des Filterbildes in den Erkennungsmethoden verarbeitet wurden, werden die gefundenen Schilder, welche alle als **TrackingPoint** gespeichert wurden, an die Klasse **SignTracker** zur Schildverfolgung übergeben. Diese implementiert das in Abschnitt 3.3.8 vorgestellte Verfahren in mehreren Methoden. Jede Schildverfolgung endet dabei erfolgreich, auch wenn ein Schild nur einmal erkannt wird. Von jedem verfolgten Schild wird das größte verfügbare Bild, zusammen mit dem Schild-Typ, in einem **StreetObject** gespeichert und zurück an die Klasse **HoughSignRecognition** übergeben. Hier werden dem **StreetObject** die restlichen Informationen beigefügt, wie die Nummer des Frames und Name und ID des Plugins, in dem das Schild erkannt wurde.

Als letztes wird eine **LinkedList<StreetObject>** der erkannten Schilder zusammen mit dem Zeichenbild in einem **ResultObject** zurück an den **VideoProcessor** übergeben.

6 Präsentation und Testfälle

6.1 GPStreetCam

6.1.1 Systemanforderungen

Zur Nutzung der Applikation *GPStreetCam* wird ein Android-Betriebssystem mit mindestens API Level 9, was der Android Version 2.3 (GINGERBREAD) entspricht, benötigt. Das Gerät muss darüber hinaus über einen GPS-Empfänger und eine Kamera verfügen. Als Mindestauflösung der Kamera werden 480p[32] mit mindestens 15 Bildern/Sekunde empfohlen, bei schlechteren Auflösungen kann eine spätere Schilderkennung nicht gewährleistet werden.

6.1.2 Bedienung des Programms

Die Benutzeroberfläche der App wurde bewusst schlicht gehalten. Auf ein Einstellungsmenü wurde verzichtet, stattdessen werden in der App automatisch die Einstellungen für die besten Aufnahmeergebnisse getroffen.

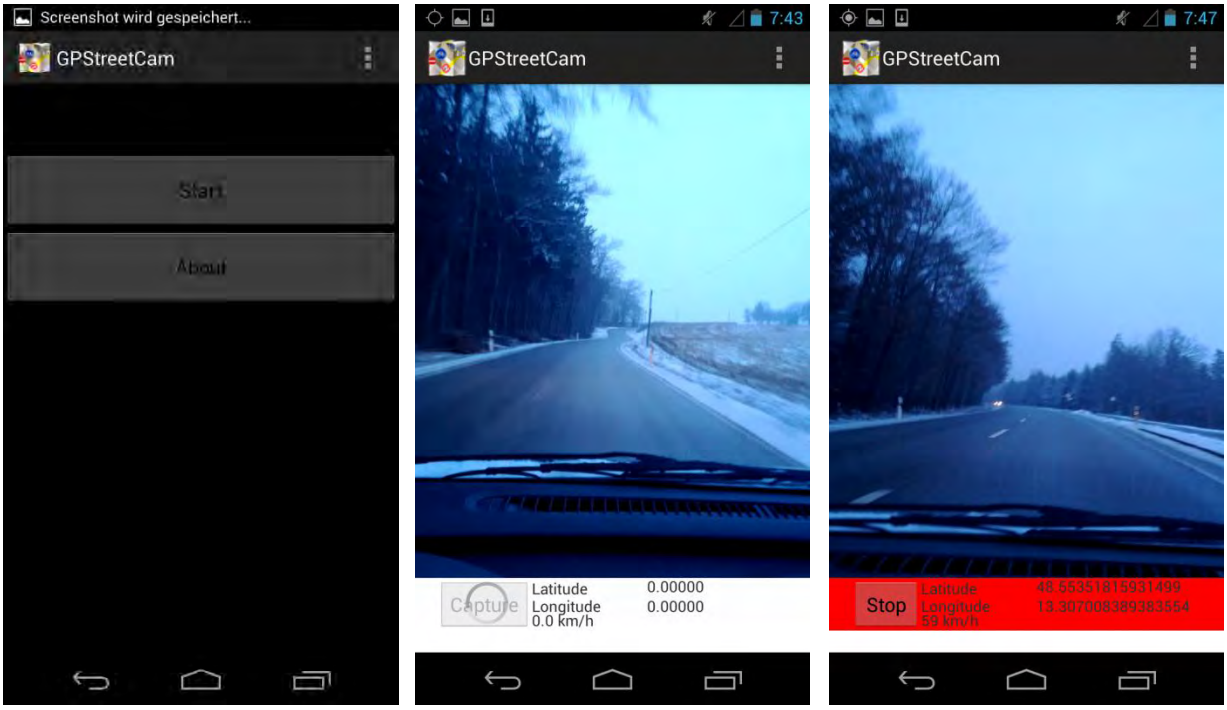

Abbildung 6-1: Hauptmenü- und Rekorder-Ansicht der App.

Abbildung 6-1 zeigt das Hauptmenü und die Rekorder-Ansicht der Applikation. Im Hauptmenü befinden sich 2 Buttons. Der Button *About* wechselt zu einem Informations-Screen, welcher die Funktionen der App kurz erklärt, über *Start* initiiert man den Rekorder und wechselt in die Rekorder-Ansicht. Falls die GPS-Funktion des Gerätes nicht aktiviert sein sollte, wird der Benutzer nun aufgefordert, diese zu aktivieren. Danach zeigt ein Lade-Symbol auf dem ausgerauten *Capture*-Button an, dass nach einer GPS-Verbindung gesucht wird. Steht die Verbindung, wird der *Capture*-Button aktiviert und die Aufnahme kann gestartet werden. Über die Anzeigen werden die aktuelle Position des Gerätes über Längen- und Breitengrad angezeigt, sowie die aktuelle Geschwindigkeit. Dies dient rein dem visuellen Feedback für den Benutzer, um zu zeigen, dass Daten empfangen

[32] Entspricht bei einem Bildverhältnis von 4:3 einer Bildgröße von 640x480, bei einem Verhältnis von 16:9 einer Größe von 854x480 Pixel, bei einem Verhältnis von 3:2 eine Größe von 720x480 usw.

werden. Über Betätigung des *Capture*-Buttons werden Video- und GPS-Aufzeichnung gestartet und die Anzeigen rot hinterlegt. Die Beschriftung des Buttons wechselt von *Capture* zu *Stop*. Nach Betätigen des *Stop*-Buttons werden Video- und GPS-Aufzeichnung gestoppt und die Daten im Medienspeicher des Gerätes gespeichert. Bei einer bestehenden GPS-Verbindung kann die Aufzeichnung beliebig oft gestartet und gestoppt werden, was es dem Benutzer leicht ermöglicht, nur die Situationen aufzuzeichnen, die er benötigt.

6.2 GPStreetTracker

6.2.1 Bedienung des Programms

Um das Java-Programm *GPStreetTracker* auszuführen wird eine Java Runtime Environment 7 (JRE 7), sowie ein eingerichtetes *OpenCV* Version 2.4.3 benötigt.

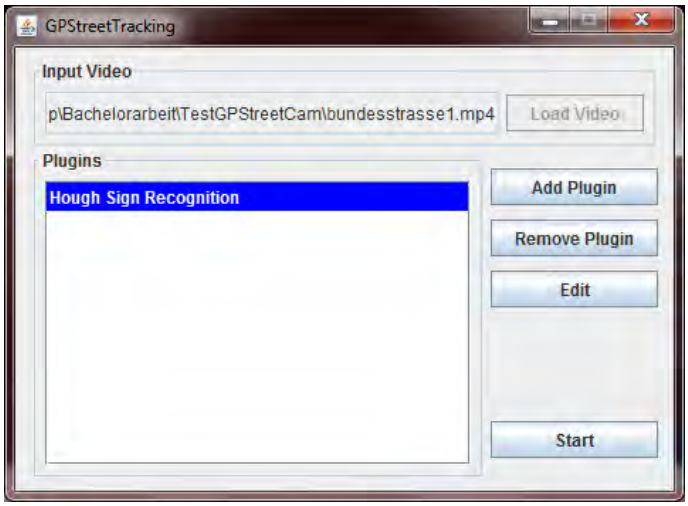

Abbildung 6-2: Startbildschirm und Hauptmenü der Programms GPStreetTracker.

Im Hauptmenü des Programms können das zu bearbeitende Video und die Plugins zur Videoverarbeitung ausgewählt werden. Dabei können beliebig viele Plugins über die Schaltfläche *Add Plugin* hinzugefügt werden. Über *Edit* gelangt man zum Einstellungsfenster des gewählten Plugins, sofern von diesem eines zur Verfügung gestellt wird. Das Plugin *HoughSignRecognition* bietet Einstellungsmöglichkeiten für alle im Erkennungsprozess verwendeten Parameter.

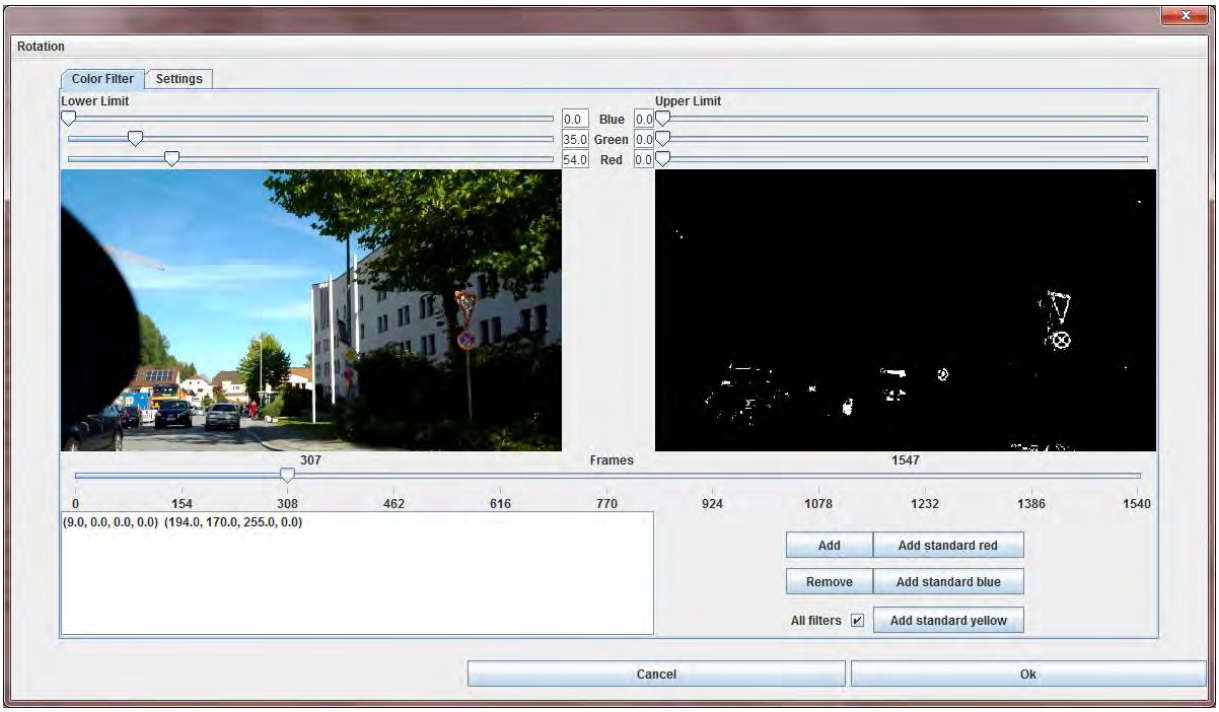

Abbildung 6-3: Einstellungsfenster für den Color Filter des Plugins HoughSignRecognition.

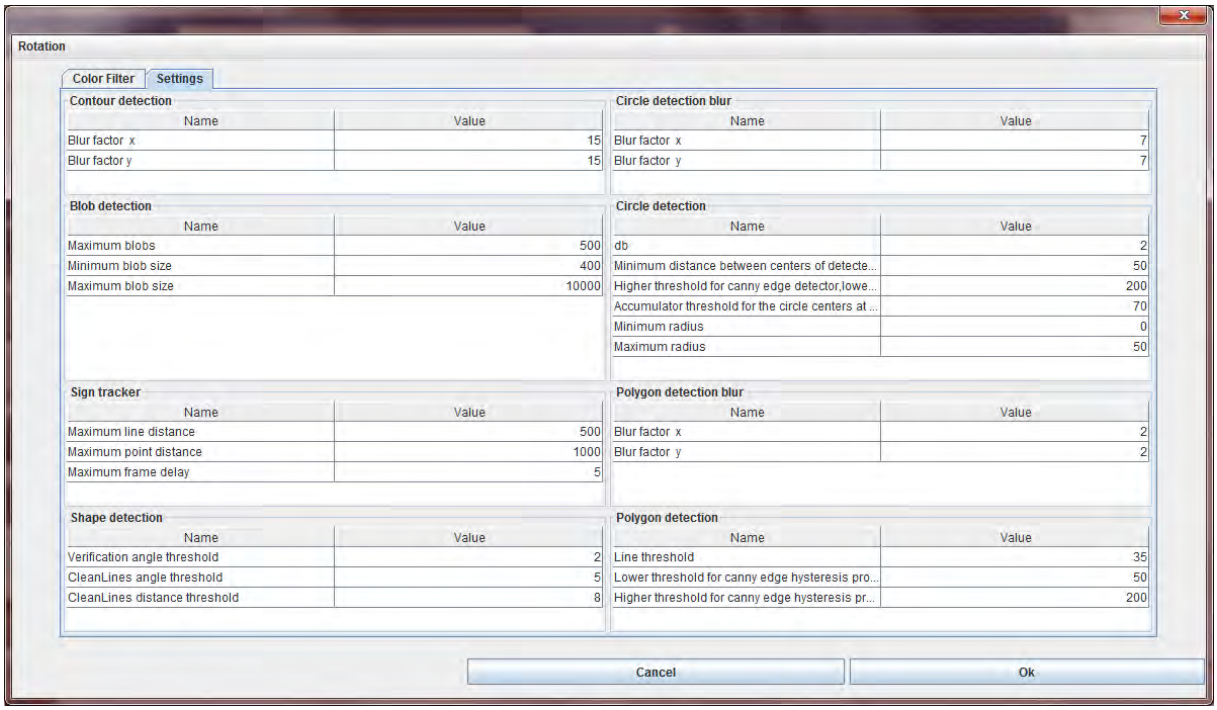

Abbildung 6-4: Einstellungsfenster für alle Parameter der Schilderkennung des Plugins HoughSignRecognition.

Abbildung 6-3 zeigt das Einstellungsfenster für die Schwellwerte des Farbfilters. Dabei kann über die beiden Farbregler eines jeden Farbkanals, der untere und obere Schwellwert separat gesetzt werden und der Filter somit auf jede beliebige Farbe eingestellt werden. Dazu bietet das Einstellungsfenster eine Vorschau an, in der das aktuelle Bild und das dazugehörige Filterergebnis angezeigt werden. Über den darunterliegenden Regler kann dafür ein Bild aus einer beliebigen Stelle des Videos

ausgewählt werden. Zusätzlich werden voreingestellte Standardwerte für rote, blaue und gelbe Schilder bereitgestellt, welche bei normalen Tageslichtverhältnissen ein gutes Ergebnis liefern.

Abbildung 6-4 zeigt das Einstellungsfenster für die Parametereinstellungen der in Abschnitt 5.2.2.2.2 gezeigten Methoden. Die voreingestellten Werte liefern in den meisten Fällen ein gutes Ergebnis und sollten nur verändert werden, wenn beispielsweise ein Video mit einer geringen Auflösung verarbeitet werden soll oder zu viele Falsch- oder Mehrfacherkennungen von Schildern vorliegen.

Wenn alle Einstellungen getroffen sind, kann über die Schaltfläche *Start* im Hauptmenü die Videoverarbeitung gestartet werden.

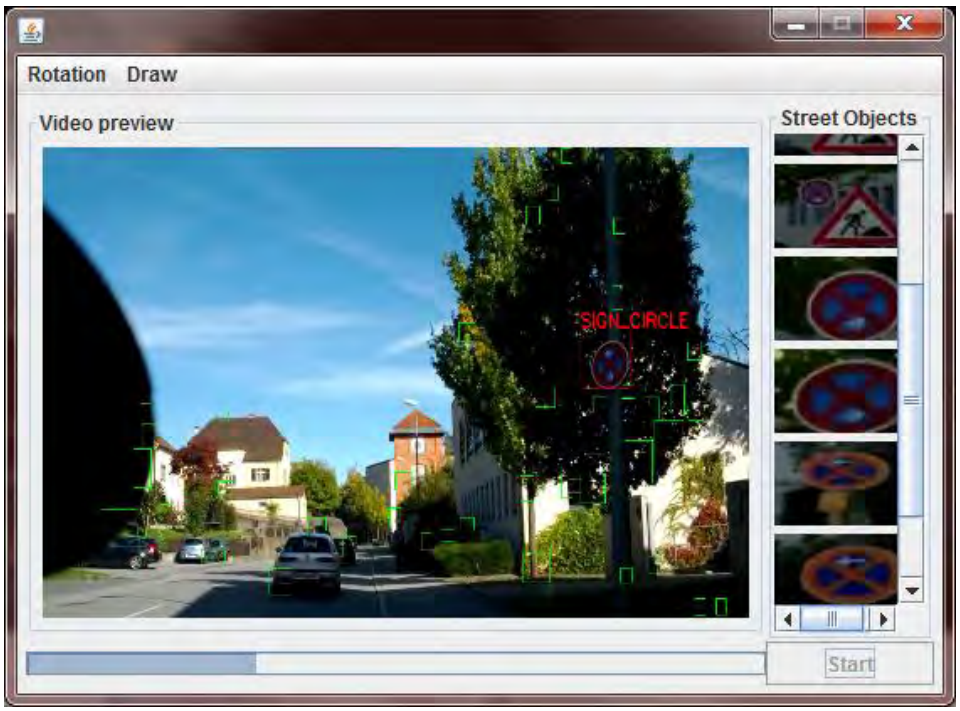

Abbildung 6-5: Das Videoverarbeitungsfenster zeigt den Fortschritt der Schilderkennung.

Das Videoverarbeitungsfenster (siehe Abbildung 6-5) ist einfach aufgebaut und gibt dem Benutzer Informationen über den Fortschritt der Objekterkennung. Eine Fortschrittsanzeige im unteren Bereich gibt Auskunft über die aktuelle Position im Video und an der rechten Seite werden dem Benutzer die bereits erkannten Objekte angezeigt. Das Vorschaufenster zeigt die eingezeichneten Blobs des Plugins *HoughSignRecognition*. Die einzelnen Zeichenfunktionen der Plugins können über das Menü *Draw* deaktiviert werden.

Nach der Verarbeitung des Videos können die gefundenen Objekte abgespeichert werden. Abbildung 6-6 zeigt das Speicherfenster. Hier kann der Benutzer im *Input*-Bereich die zum Video gehörende GPX-Datei auswählen und im *Output*-Bereich den Speicherort für die Bilder, sowie einen Namen angeben, unter dem die Objekte gespeichert werden sollen. Dabei können verschiedene Platzhalter für *Name* des Plugins(#name#), *Datum*(#date#), *Typ* des erkannten Objekts(#type#), die *Framenummer*, in dem das Objekt gefunden wurde(#frame#) und einen fortlaufenden *Index*(#index#) gewählt werden. Wird im Dateinamen kein Index gesetzt, werden die Dateien automatisch durchnummeriert. Im linken Bereich wird dem Benutzer eine Liste aller gefundenen Objekte angezeigt, in der er die gewünschten Objekte zur Speicherung auswählen kann. Die Bilder der Objekte werden dabei im JPEG-Format in dem angegebenen Verzeichnis gespeichert. In der GPX-

Datei werden Waypoints mit demselben Namen angelegt, damit die Bilder später den einzelnen Punkten zugeordnet werden können (siehe Abbildung 6-7).

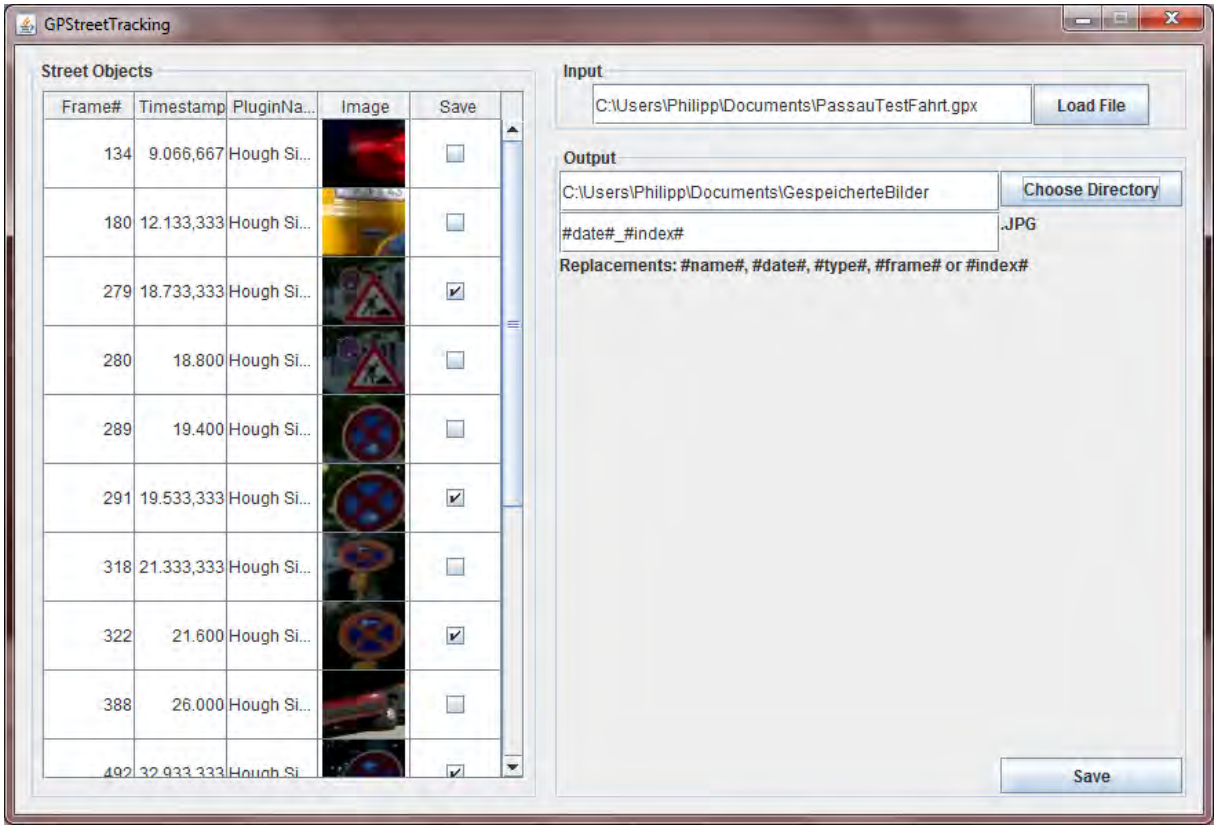

Abbildung 6-6: Objektverwaltungs- und Speicherfenster des Programms GPStreetTracker.

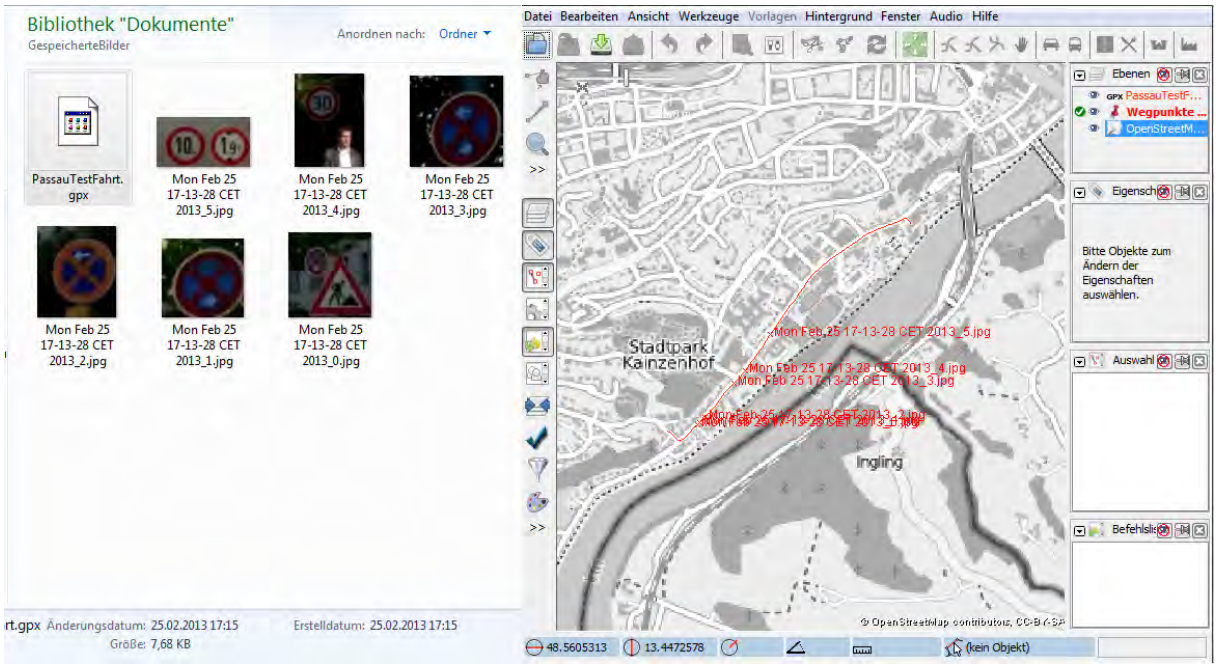

Abbildung 6-7: Zeigt die gespeicherten Bilder und den GPX-Track. Im Programm JOSM können der Track und die eingezeichneten Waypoints auf einer Karte dargestellt werden.

6.2.2 Testfälle

In diesem Abschnitt werden die Funktionsfähigkeit und die Zuverlässigkeit der Schilderkennungs-algorithmen des Plugins *HoughSignRecognition* demonstriert, sowie auch dessen Nachteile und Schwächen aufgezeigt. Auch die Auswirkungen von Parameteränderungen werden gezeigt und wie man durch eine gezielte Anpassung der Werte auf bestimme Situationen, eine bessere Erkennung erzielen kann. Anschließend folgt eine Auswertung der genannten Schilderkennungsalgorithmen, welche einen Überblick über die Berechnungszeiten und die Skalierbarkeit des Plugins *HoughSignRecognition* gibt.

6.2.2.1 Funktion und Zuverlässigkeit

Das Schilderkennungssystem besitzt viele Parameter, mit denen der Nutzer Einfluss auf die einzelnen Methoden und Erkennungsmechanismen nehmen kann. Diese Änderungen haben direkte Auswirkungen auf die Ergebnisse der Erkennungsmethoden. Damit eine Erkennung immer fehlerfrei gestartet werden kann, ist für jeden Parameter ein fester Maximalwert bzw. Minimalwert vorgegeben, den der Benutzer nicht überschreiten bzw. unterschreiten kann. Über das Einstellungsfenster können Einstellungen getroffen werden, über:

- Den Weichzeichner:
  In den Einstellungen kann die Kernelgröße des Weichzeichners gewählt werden. Die Größe des Kernels bestimmt den Grad der Weichzeichnung des Bildes. Es können 3 Kernelgrößen für die 3 verwendeten Weichzeichner in den Erkennungsmethoden gesetzt werden. Mit einer Veränderung der vorgegebenen Werte kann erreicht werden, dass die Erkennungsrate bei schlechten ‚Farbverhältnissen' der Schilder gesteigert werden kann. Dies kann der Fall sein, wenn die Schilder schmutzig oder schneebedeckt sind. Durch eine höhere Weichzeichnung, können die Farblücken besser ‚verwischt' werden.
- Die Blob-Detection:
  Hier können die maximale Anzahl der zu untersuchenden Blobs, sowie deren Größe über einen Minimal- und einen Maximalwert festgelegt werden. Durch eine geringere Anzahl von Blobs kann in erster Linie der Rechenaufwand bei Verarbeitung reduziert werden, jedoch steigt dadurch auch die Wahrscheinlichkeit, dass Straßenschilder übersehen und nicht verarbeitet werden. Durch das Anpassen des Minimal- und des Maximalgrößenwerts können zu kleine Farbflächen, wie etwa die Rückleuchten eines PKW, und zu große Flächen, wie beispielsweise Hausdächer, ausgeschlossen werden.
- Die Kreiserkennung:
  Die Parameter, sowie die Auswirkungen bei deren Änderung, werden in Tabelle 6-1 aufgelistet.
- Die Linien- und Formerkennung:
  Die Parameter, sowie die Auswirkungen bei deren Änderung, werden in Tabelle 6-2 aufgelistet.
- Die Schildverfolgung:
  Hier können 3 wichtige Parameter für die Schildverfolgung festgelegt werden. Der maximale Radius r und der Abstand ϵ, wie sie aus Abschnitt 3.3.8 bekannt sind, und die maximale Framezahl, nach der ein Schild als erkannt gilt, sofern es nicht weiter getrackt wird. Die Änderung dieser Parameter hat keinen Einfluss auf die Schilderkennung, jedoch auf die Mehrfacherkennung der Schilder. Finden zu viele Mehrfacherkennungen statt, kann dem durch eine Erhöhung der Werte entgegengewirkt werden.

Parameter	Zweck	Einfluss auf die Kreiserkennung
Akkumulatorgröße	Bestimmt die Größe des Akkumulators im Vergleich zum Originalbild.	Je kleiner der Akkumulator gewählt wird, desto mehr Kreise werden erkannt, aber auch die Anzahl an Falscherkennungen steigt.
Minimalabstand	Minimalabstand zwischen den Kreismittelpunkten.	Durch einen größeren Wert kann die Erkennung eines Kreises innerhalb eines anderen Kreises vermieden werden.
Threshold des Kantendetektors	Setzt den oberen Threshold für die Kantenerkennung des *Canny*-Kantendetektors, der untere beträgt die Hälfte dieses Wertes.	Durch das Setzen eines niedrigen Threshold wird die Anzahl an erkannten Kanten erhöht, was eine bessere Kreiserkennung bei schlechten Lichtverhältnissen ermöglicht. Es steigert jedoch auch die Anzahl der Falscherkennungen.
Threshold der Radien-Erkennung	Bestimmt die Anzahl der Kandidaten, die für einen Kreismittelpunkt stimmen müssen.	Das Ändern dieses Wertes dient demselben Zweck, wie das, des Kantendetektor-Threshold. Auch hier wird durch das Setzen eines niedrigeren Wertes die Möglichkeit einer Kreiserkennung, aber auch die einer Falscherkennung, erhöht. Durch einen höheren Wert können Falscherkennungen vermieden werden, denn es werden nur noch deutliche Kreise erkannt.
Minimaler Kreisradius	Bestimmt den kleinstmöglichen Kreisradius.	Durch das Setzen des Minimal- und Maximalwertes werden nur noch Kreise innerhalb dieser Schranken erkannt. Durch einen schmalen Wertebereich kann die Berechnungszeit verkürzt werden, da weniger Radien untersucht werden müssen.
Maximaler Kreisradius	Bestimmt den größtmöglichen Kreisradius.	siehe Minimaler Kreisradius.

Tabelle 6-1: Parametereinstellungen der Kreiserkennung und ihre Auswirkungen auf die Schilderkennung.

Parameter	Zweck	Einfluss auf die Linien- und Formerkennung
Threshold der Linienerkennung	Bestimmt die Anzahl an Kurvenschnittpunkten im Ergebnisraum, die zur Erkennung einer Linie nötig sind.	Bei einem niedrigen Wert können auch kurze Linien und somit kleine Schilder erkannt werden. Es wird aber auch die Zahl der Falscherkennungen gesteigert.
Oberer Threshold des Kantendetektors	Oberer Threshold für den Canny-Kantendetektor	Je niedriger dieser Wert ist, desto mehr Kanten werden im Bild erkannt. Dies kann die Schilderkennung bei schlechten Lichtverhältnissen steigern, führt aber auch zu mehr Falscherkennungen.
Unterer Threshold des Kantendetektors	Unterer Threshold für den Canny-Kantendetektor	Je größer der Wertebereich zwischen dem unteren und dem oberen Threshold ist, desto mehr nichteindeutige Kanten werden erkannt. Dies kann die Erkennung bei schlechten Lichtverhältnissen steigern.
Threshold des Winkels für die CleanLines-Methode	Gibt den Winkelbereich an, in dem die Geraden gelöscht werden.	Der Threshold für den Winkel, sowie auch der für den Abstand sollten nur verändert werden, wenn aufgrund einer sehr niedrigen Auflösung sonst keine viereckigen Schilder mehr erkannt werden können.
Threshold des Abstandes für die CleanLines-Methode	Gibt den Abstand an, innerhalb welchem die Geraden gelöscht werden.	siehe Threshold des Winkels für die CleanLines-Methode.
Threshold des Winkels für die Schildverifizierung	Maximalwinkel um den die Geraden von den vordefinierten Werten zur Formerkennung abweichen dürfen.	Mit einem hohen Wert, können auch sehr schräg stehende, eckige Verkehrsschilder noch erkannt werden. Es wird aber auch die Anzahl der Falscherkennungen gesteigert.

Tabelle 6-2: Parametereinstellungen der Linien- und Formerkennung und ihre Auswirkungen auf die Schilderkennung.

In den Abbildungen Abbildung 6-8 - Abbildung 6-15 wird die Erkennung verschiedener Straßenschilder unter verschiedenen Verkehrs- und Umweltverhältnissen gezeigt, sowie die Auswirkungen verschiedener Parametereinstellungen.

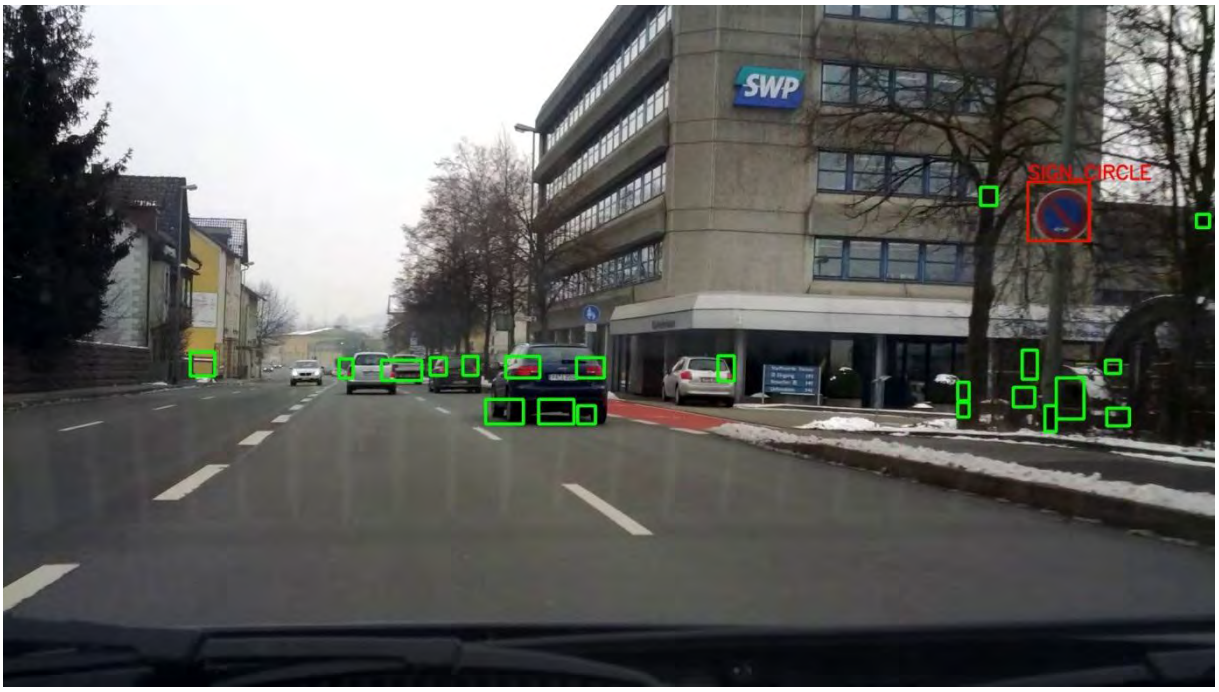

Abbildung 6-8: Erkennung eines runden Verkehrsschildes bei dunklem Hintergrund. Hier wurde kein Blaufilter gesetzt, darum ist das blaue Schild nicht als Blob gekennzeichnet.

Abbildung 6-9: Erkennung eines runden Verkehrsschildes vor hellem Hintergrund.

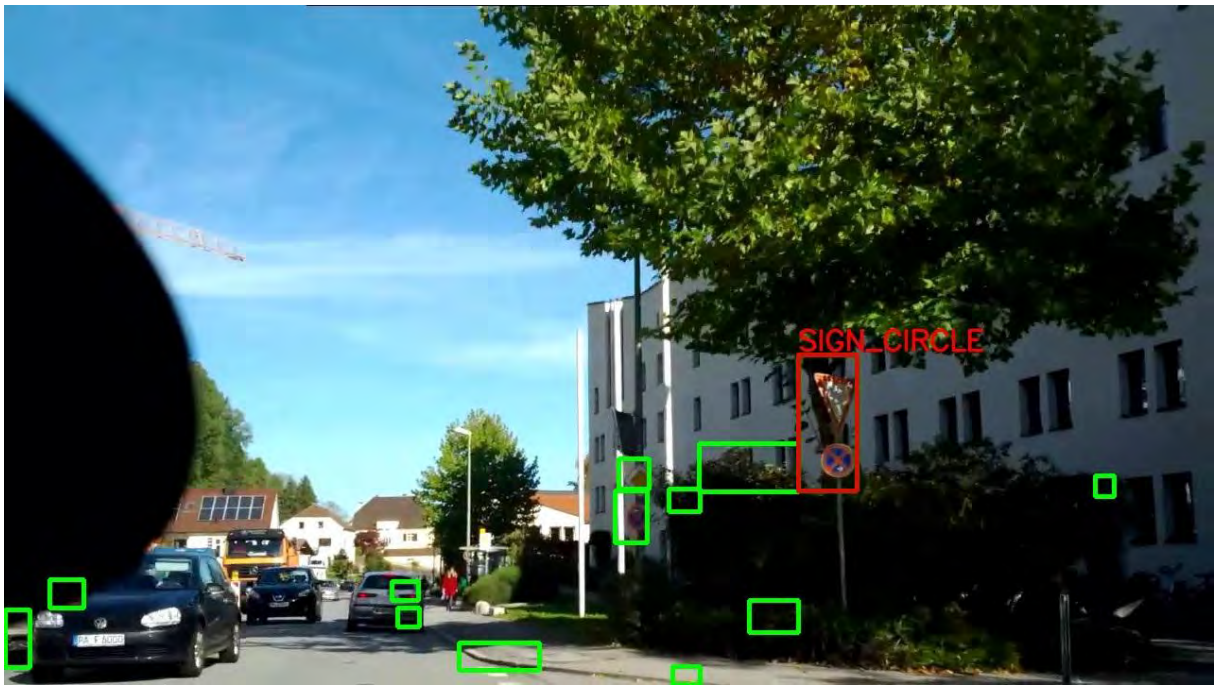

Abbildung 6-10: Erkennung eines runden Verkehrsschildes im Schatten, das dreieckige Schild wurde im selben Blob erfasst und wäre ansonsten, aufgrund der starken Lichtunterschiede auf dem Schild, nicht erkannt worden. Die beiden weiter entfernten Schilder wurden noch nicht erkannt.

Abbildung 6-11: Erkennung eines viereckigen Schildes. Hier wurde kein Rotfilter gesetzt, darum wird das rote Schild nicht erkannt.

Abbildung 6-12: Erkennung zweier runder Schilder in einem sehr dunklen Bildbereich.

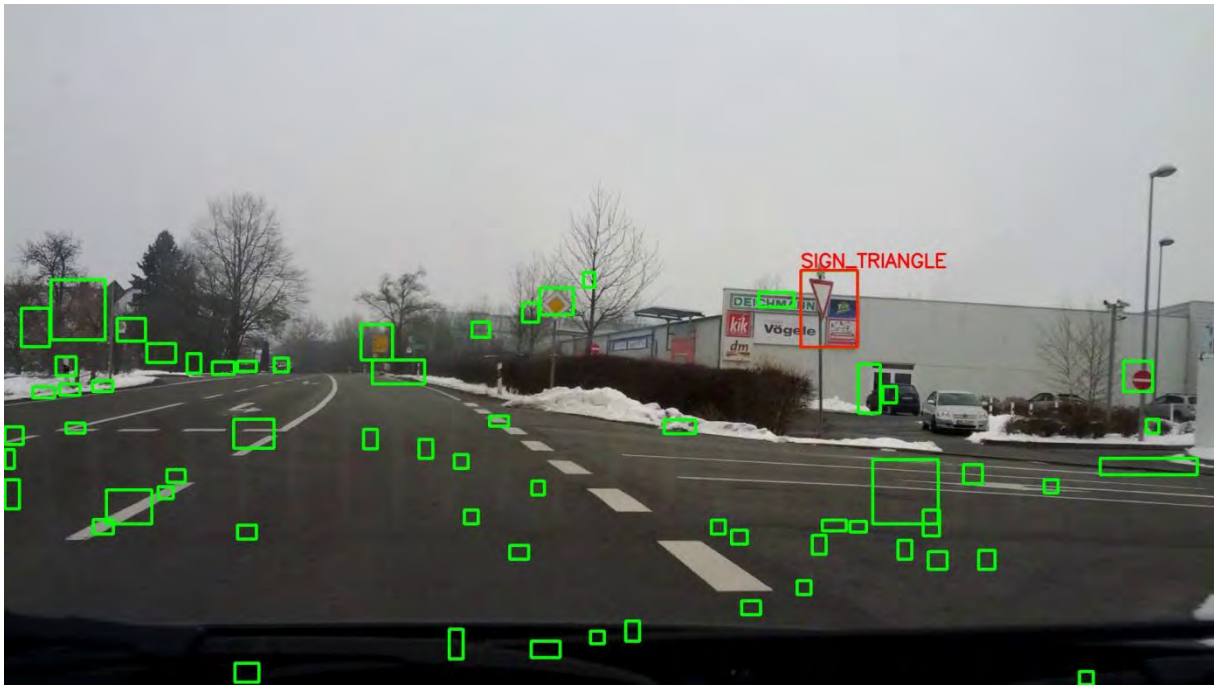

Abbildung 6-13: Erkennung eines schräg stehenden, dreieckigen Schildes. Das viereckige und das runde Schild wurden (noch) nicht erkannt.

Abbildung 6-14: Durch Anpassung des Rotwertes im Filter, konnte auch ein sehr ausgeblichenes, rundes Schild erkannt werden.

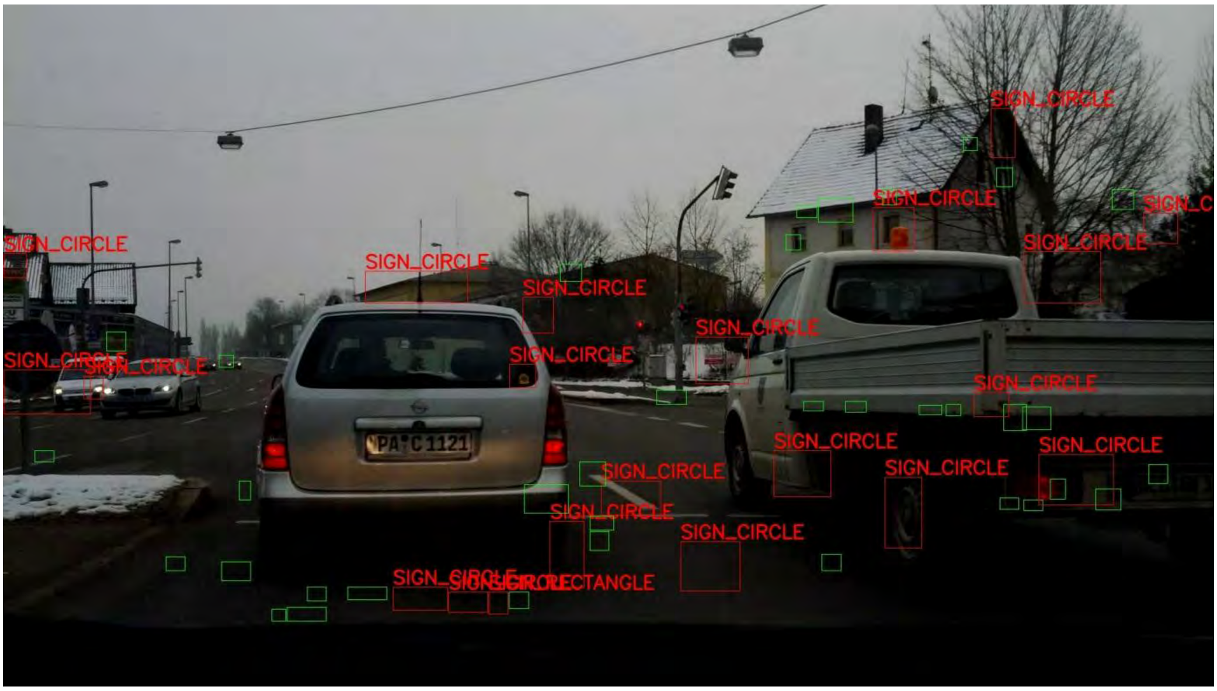

Abbildung 6-15: Durch das Setzen von sehr niedrigen Schwellwerten in der Kreiserkennung wurden Falscherkennungen provoziert.

6.2.2.2 Zeitauswertung

Das Plugin *HoughSignRecognition* bietet eine schnelle Methode zur Schilderkennung. Das Bild wird dazu anhand eines Farbfilters und Blob-Erkennung segmentiert. Die einzelnen Bildsegmente werden anschließend an die Formerkennung übergeben. Die folgenden Grafiken geben einen Überblick über die Verarbeitungszeiten der Bilder in den einzelnen Arbeitsschritten des Plugins.

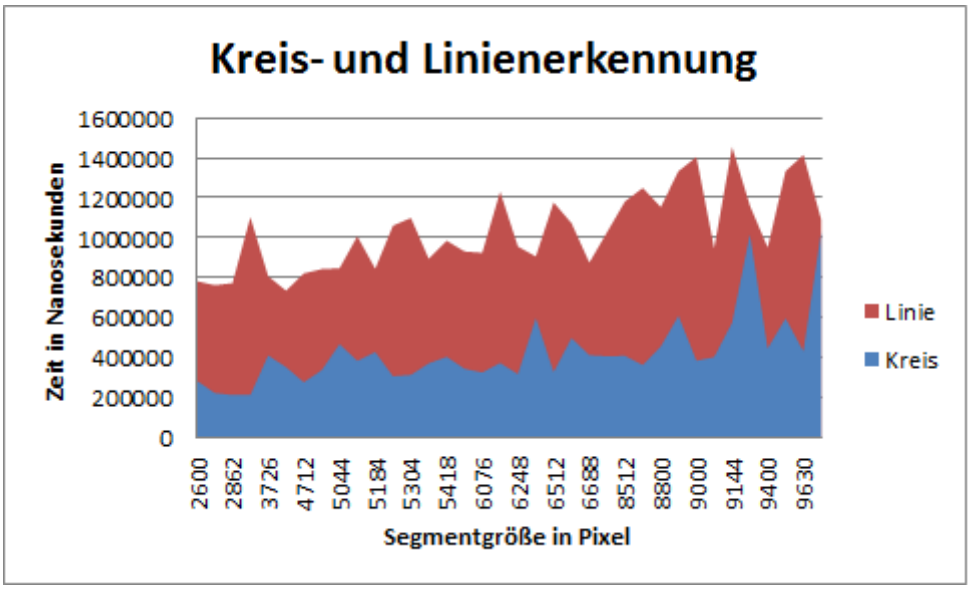

Abbildung 6-16: Verarbeitungszeiten der Kreis- und Polygonerkennung des Plugins *HoughSignRecognition*.

Abbildung 6-16 zeigt die Verarbeitungszeiten einer erfolgreichen Formerkennung. Die blaue Kurve zeigt dabei, wie lange die Erkennung eines Kreises in einem Bildsegment dauert. Die rote Kurve zeigt die Dauer einer Polygonerkennung. Die Verarbeitungszeit ist in Nanosekunden und auf der x-Achse sind die Größen der Bildsegmente in Pixel angegeben. Die Auswertung ergibt, dass die Erkennung eines Kreises (ca. 0,4ms) im Durchschnitt nur halb so lange dauert wie die Erkennung eines Polygons (ca. 0,8ms). Dies lässt sich wie folgt erklären. Die Hough-Transformation zur Linienerkennung ist zwar weniger aufwendig als die generalisierte Hough-Transformation zur Kreiserkennung, jedoch müssen für eine Polynomerkennung, wie etwa die Erkennung eines Dreiecks, alle gefundenen Linien gefiltert und untereinander verglichen werden. Die unter anderem dazu verwendeten Listenoperationen sind vergleichsweise langsam, was sich in der Verarbeitungszeit bemerkbar macht.

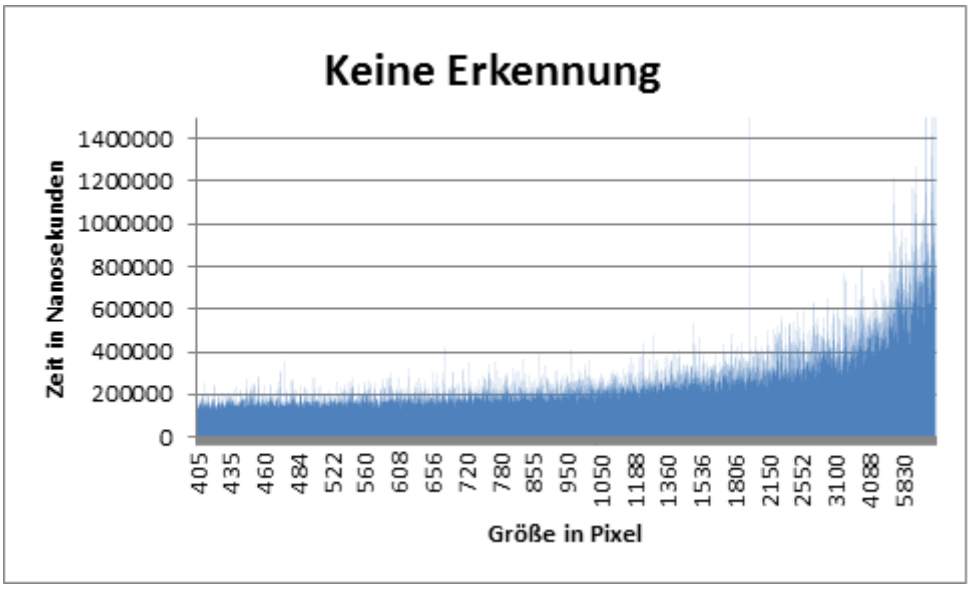

Abbildung 6-17: Verarbeitungszeiten von Bildsegmenten, in denen keine Form erkannt wird.

Abbildung 6-17 zeigt die Verarbeitungszeiten von Bildsegmenten, in denen keine Form erkannt wird. Dies ist bei über 90% aller in der Formerkennung verarbeiteten Bildsegmente der Fall. Die Abbildung zeigt die Auswertung von ca. 20000 Messwerten und bietet neben dem Zeitmesswert auch einen Überblick über die Häufigkeit der auftretenden Bildgrößen. Kleine Segmente (< 1000 Pixel) treten dabei viel häufiger auf, als große. Somit ist die durchschnittliche Verarbeitungszeit (ca. 0,3ms) einer Nichterkennung vergleichsweise gering.

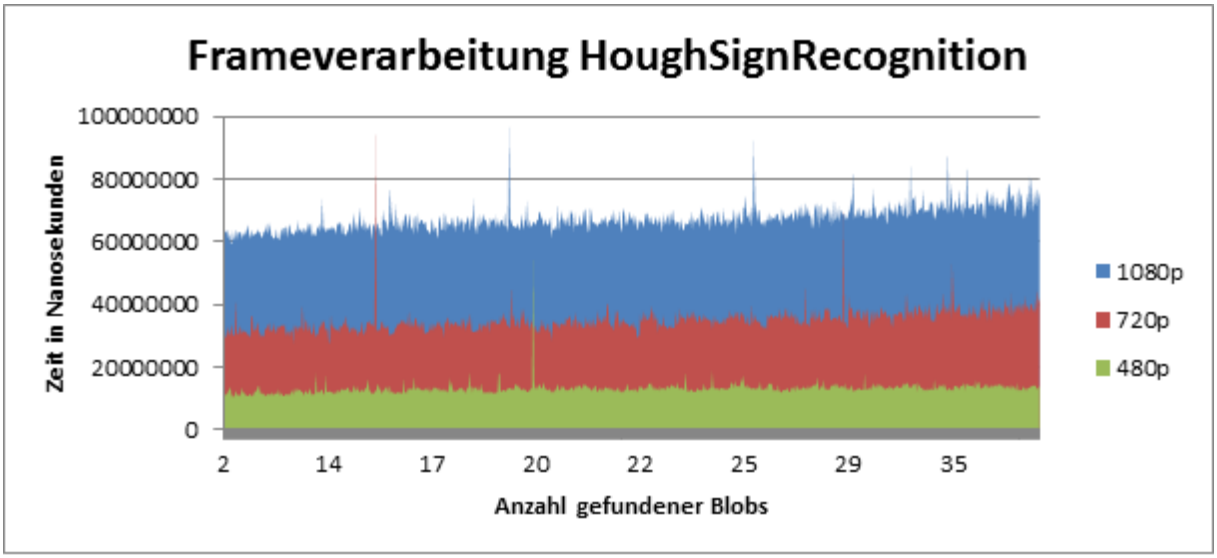

Abbildung 6-18: Verarbeitungszeit eines kompletten Frames im Plugin *HoughSignRecognition*.

In Abbildung 6-18 sieht man die Verarbeitungszeit eines Frames im Plugin *HoughSignRecognition*. Die Zeit wird dabei von der Übergabe des Frames an das Plugin bis zur Rückgabe der erkannten Schilder an den **VideoProcessor** gemessen. Untersucht wurden 3 Verschiedene Video-Formate. Die blaue Kurve zeigt die Verarbeitungszeit eines Frames mit der Auflösung 1920x1080 Pixel (Full HD), Frames mit der Auflösung 1280x720 werden rot und Frames mit der Auflösung 640x480 werden grün

dargestellt. Die Auswertung zeigt, dass die Verarbeitungszeit eines Frames bei ca. vierfacher Bildfläche um das Doppelte ansteigt. Die Anzahl der gefundenen Blobs, also die Zahl der auszuwertenden Bildsegmente, beeinflusst die Gesamtverarbeitungszeit nur geringfügig, da die Formerkennung nur einen Bruchteil dieser Zeit in Anspruch nimmt. Die Auswertung zeigt, dass die Aufbereitung und Segmentierung des Bildes, sowie das Tracking der gefundenen Schilder, die meiste Zeit brauchen.

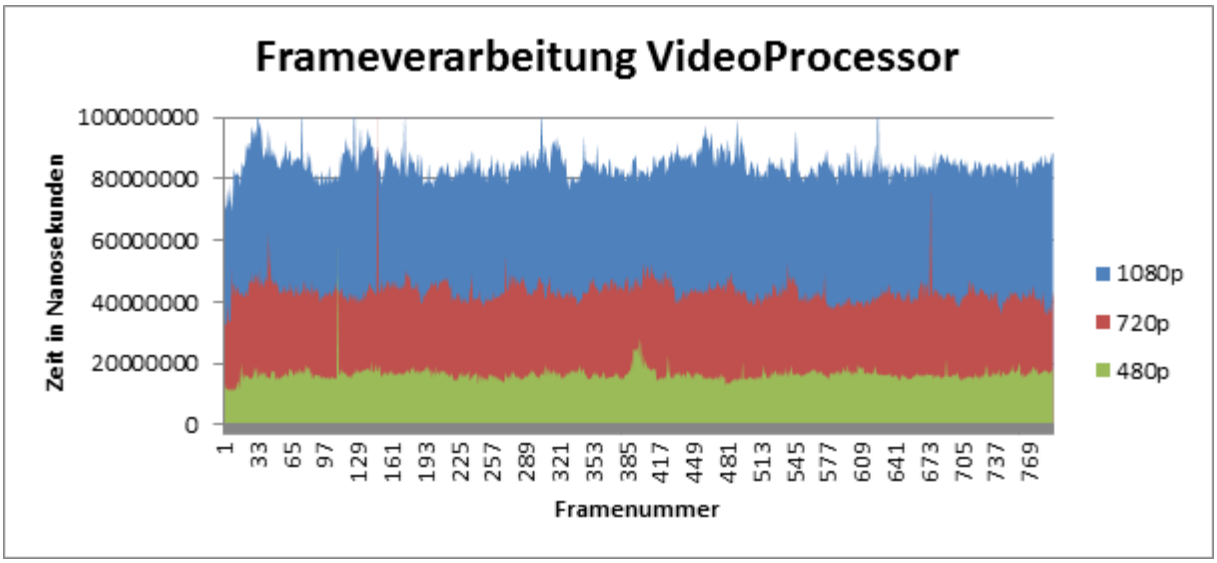

Abbildung 6-19: Verarbeitungszeit vom Einlesen eines Frames bis zum Einlesen des nächsten im VideoProcessor.

Abbildung 6-19 zeigt die Verarbeitungszeiten der Frames eines Videos. Hierbei wurde die Zeit vom Einlesen eines Frames in den **VideoProcessor**, über die Verarbeitung des Frames, bis zum Einlesen des nächsten Frames gemessen. Bei diesen Messungen wurde nur das Plugin *HoughSignRecognition* verwendet. Man sieht, dass die gemessenen Zeiten im Wesentlichen denen aus Abbildung 6-18 entsprechen. Es addiert sich lediglich die Zeit hinzu, die der **VideoProcessor** zum Extrahieren eines Frames aus dem Video braucht. Diese steigt mit der Auflösung des Videos.

7 Fazit und Ausblick

7.1 Fazit

In dieser Bachelorarbeit wurde ein funktionierendes Framework zur Objekterkennung im Straßenverkehr geschaffen und darüber hinaus mit dem Plugin *HoughSignRecognition* ein schnelles Verfahren zur Straßenschilderkennung, welches brauchbare Ergebnisse liefert.

Das Video- und GPS-Datenmaterial wird über die speziell dafür entwickelte Android Applikation *GPStreetCam* aufgezeichnet. Die Videoqualität und die Genauigkeit des GPS-Empfängers eines aktuellen Smartphones erweisen sich als sehr gut und sind mehr als ausreichend für das Ziel dieser Arbeit. Später am PC werden die aufgezeichneten Daten mit dem entwickelten Programm *GPStreetTracker* verarbeitet. Das Framework liest das Video ein und extrahiert nacheinander alle Frames aus dem Video. Diese Frames werden dann nacheinander an die einzelnen Plugins übergeben. Das Plugin *HoughSignRecognition* filtert das übergebene Bild zuerst anhand vorgegebener Farben und segmentiert die entstandenen Binärbilder danach anhand der erkannten Blobs. Die Bildsegmente werden als nächstes einzeln an die Formerkennungsmethoden übergeben. Die Formerkennung bietet als großen Vorteil gegenüber anderen Erkennungsmethoden, dass vorab keine speziellen Eigenschaften und auch keine Vergleichs-Templates der individuellen Schilder gespeichert werden müssen. Anhand der einfachen, geometrischen Formen, können aktuelle Verkehrsschilder, aber auch unbekannte Schilder erkannt werden. Wurde ein Schild anhand seiner Form erkannt und mit Hilfe der Tracking-Algorithmen getrackt, wird es als StreetObject an das Framework übergeben. Hier werden die gesammelten StreetObjects aller Plugins verwaltet. Nach Abschluss der Objekterkennung werden die StreetObjects als Bilder der erkannten Objekte und als Waypoint in dem zum Video gehörenden GPX-Track abgespeichert.

Auf diese Weise erfüllen die Android Applikation und das Java Programm das Ziel, ein automatisiertes Verfahren zur Objekterkennung und Kartographierung, zu erschaffen.

7.2 Ausblick

Das Framework *GPStreetTracker* sowie das Plugin *HoughSignRecognition* sind an vielen Stellen noch ausbaufähig. Es wäre beispielsweise eine Erweiterung der verarbeitbaren Videoformate wünschenswert. Derzeit ist nur die Verarbeitung von MP4-Codierten Videos möglich, da diese bei der Verarbeitung mit den *OpenCV*-Methoden die wenigsten Schwierigkeiten bereiteten.

Die größte Schwäche des Schilderkennungsalgorithmus des Plugins *HoughSignRecognition* ist der Farbfilter. Gelbe oder grüne Farbflächen können nur sehr schwer gefiltert werden, da diese Farben ähnlich große Werte in den RGB-Kanälen besitzen. Hier ist die Verwendung eines anderen Farbraums zur Filterung oder eine andere Farbmanipulation des Originalbildes zu überdenken. Die von *OpenCV* bereitgestellten Algorithmen zur Hough-Transformation liefern bei der Linien- und Kreiserkennung gute Ergebnisse. Hier könnte man mit generalisierter Hough-Transformation und geeigneten Funktionen aber auch direkt nach den gesuchten Formen, wie Drei- und Vierecken suchen, ohne dazu den Weg über die Formerkennungsalgorithmen zu gehen.

Als Erweiterung dieser Arbeit wäre die Entwicklung weiterer Plugins denkbar. Das Framework bietet dazu die Möglichkeit, eigene Objekterkennungsalgorithmen auf die Videoframes anzuwenden. Diese können dabei leicht als Plugin in das Framework integriert werden und müssen dazu lediglich das einheitliche Plugin-Interface *ObjectDetectionPlugin* zur Kommunikation mit dem Framework

implementieren. Die erkannten Objekte können später mit Hilfe des Frameworks in eine Karte eingetragen werden.

Anhang

Auf der beiliegenden CD befinden sich die entwickelten Programme, deren Quellcode, sowie diese Arbeit in digitaler Form. Eine Beschreibung zur Ausführung und Benutzung der Programme befindet sich in der Datei „README.txt“ im Grundverzeichnis der CD.

Verwendete Objekte und Methoden aus den OpenCv-Bibliotheken

Name	Typ	Bibliothek	Beschreibung
blur	met	*imgproc*	Weichzeichner-Methode.
CV_AA	int	*core*	Integer-Wert für Font-Style.
CV_CAP_PROP_FPS	int	*highgui*	Parameter für die cvGetCapturePropery-Methode.
CV_CAP_PROP_FRAME_COUNT	int	*highgui*	Parameter für die cvGetCapturePropery-Methode.
CV_CAP_PROP_FRAME_HEIGHT	int	*highgui*	Parameter für die cvGetCapturePropery-Methode.
CV_CAP_PROP_FRAME_WIDTH	int	*highgui*	Parameter für die cvGetCapturePropery-Methode.
CV_CAP_PROP_POS_FRAMES	int	*highgui*	Parameter für die cvGetCapturePropery-Methode.
CV_CAP_PROP_POS_MSEC	int	*highgui*	Parameter für die cvGetCapturePropery-Methode.
CV_CHAIN_APPROX_SIMPLE	int	*imgproc*	Integer-Wert für cvFindContours.
CV_FONT_HERSHEY_SIMPLEX	int	*core*	Integer-Wert für Font-Style.
CV_HSV2RGB	int	*imgproc*	Parameter für die cvCvtColor-Methode.
CV_RETR_CCOMP	int	*imgproc*	Integer-Wert für cvFindContours.
CV_RGB2HSV	int	*imgproc*	Parameter für die cvCvtColot-Methode.
cvBoundingRect	met	*imgproc*	Liefert das kleinste, eine Sequenz von Punkten umschießende Rechteck.
CvCapture	obj	*highgui*	Dient dem Einlesen eines Videos.
cvCloneImage	met	*core*	Klont ein IplImage-Objekt.
CvContour	obj	*core*	Sequenz von CvPoint-Objekten.
cvCreateFileCapture	met	*highgui*	Initialisiert ein CvCapture-Objekt.
cvCreateImate	met	*core*	Initialisiert ein neues IplImage-Objekt.
cvCvtColor	met	*imgproc*	Wandelt IplImage-Bilder von einen in den anderen Farbraum um.
cvFindContours	met	*imgproc*	Liefert Sequenzen von zusammenhängenden Pixeln.
CvFont	obj	*core*	Formatierung für Schrift-Layout.
cvGetCaptureProperty	met	*highgui*	Liefert die Eigenschaften eines CvCapture-Objekts.
cvGetImageRoi	met	*core*	Liefert ein CvRect-Objekt der ROI.
cvGetRectSubPix	met	*imgproc*	Liefert den durch ein CvRect definierten Teil einer Pixelmatrix.
cvGetSize	met	*core*	Liefert die Größe eines IplImage-Bildes.
cvInitFont	met	*core*	Initialisiert ein CvFont-Objekt.
cvInRangeS	met	*core*	Filtert eine Pixelmatrix anhand von Schwellwerten.
CvMemStorage	obj	*core*	Allokalisiert Arbeitsspeicher für OpenCv-Methoden.
cvPoint	met	*core*	Initialisiert ein CvPoint-Objekt.
CvPoint	obj	*core*	2D Punkt mit Integer Koordinaten.

CvPoint2D32f	obj	*core*	2D Punkt mit Float Koordinaten.
cvPutText	met	*core*	Schreibt Strings in eine Pixelmatrix.
cvQueryFrame	met	*highgui*	Liest den Frame eines Videos als IplImage aus.
CvRect	obj	*core*	Enthält die Koordinaten eines Rechtecks.
cvRectangle	met	*core*	Zeichnet Rechtecke in eine Pixelmatrix.
cvReleaseCapture	met	*highgui*	Befreit den Speicher eines CvCapture im Arbeitsspeicher.
cvReleaseImage	met	*core*	Befreit den Speicher eines Bildes im Arbeitsspeicher.
cvResetImageROI	met	*core*	Löscht die gesetzte ROI in einem IplImage.
cvResize	met	*imgproc*	Ändert die Größe von IplImage-Bildern.
cvSaveImage	met	*highgui*	Speichert ein IplImage als JPEG ab.
CvScalar	obj	*core*	Container für 1-,2-,3- oder 4-Tupel.
CvSeq	obj	*core*	OpenCv-Sequenz, eine Liste für OpenCv-Objekte
cvSetImageROI	met	*core*	Setzt die ROI in einem IplImage.
cvSize	met	*core*	Initialisiert ein CvSize-Objekt.
CvSize	obj	*core*	Pixelgröße eines Rechtecks.
IPL_DEPTH_8U	int	*core*	Integer-Wert für eine Bildtiefe von 8 Bit.
IplImage	obj	*core*	Eine Pixelmatrix zum Speichern von Bildern.

Tabelle 7-1: Verwendete Objekte und Methoden aus den OpenCV-Bibliotheken.

Literaturverzeichnis

[1]. **Wismann, Andreas.** http://www.fahrtipps.de/. [Online] 1997-2013. [Zitat vom: 28. 01 2013.] http://www.fahrtipps.de/.

[2]. **IT-Wissen.** IT-Wissen. [Online] [Zitat vom: 28. 01 2013.] http://www.itwissen.info/definition/lexikon/coarse-aquisition-C-A-C-A-Code.html.

[3]. **Abel, Heinrich.** Gps: Global positioning system - Funktionsweise und mathematische Grundlagen. [Online] [Zitat vom: 28. 01 2013.] http://www2.hs-esslingen.de/~abel/gps/Abel-GPS.htm.

[4]. **NAVSTAR.** NAVSTAR GLOBAL POSITIONING SYSTEM. *Navstar GPS Space Segment/Navigation User Interfaces.* El Segundo, USA : s.n., 07.03.2006. Revision D IRN-200D-001.

[5]. **OSM.** Open Street Map. [Online] http://www.fahrtipps.de/.

[6]. **Google.** Google Maps. [Online] [Zitat vom: 28. 01 2013.] http://www.google.com/intl/de_DE/help/terms_maps.html.

[7]. **LinuxForU.** A Developer's First Look At Android. [Online] 2008. [Zitat vom: 28. 1 2013.] www.linuxforu.com.

[8]. **Eileen Kühn, Ronny Pflug, Björn Bittins, Mathias Lenz.** *Programmierung mobiler Anwendungen.* Hochschule für Technik und Wirtschaft (HTW) Berlin : s.n., 2011.

[9]. **Laganière, Robert.** *OpenCV 2 Computer Vision Application Programming Cookbook.* Birmingham, B27 6PA, UK : Packt Publishing Ltd., 2011. ISBN 978-1-849513-24-1.

[10]. **OpenCV.** OpenCV Willow Garage. [Online] 2013. [Zitat vom: 02. 02 2013.] http://opencv.willowgarage.com/documentation/index.html.

[11]. **Wikipedia.** Wikipedia. [Online] http://de.wikipedia.org/wiki/Objekterkennung.

[12]. **Universität, Madeburg.** Grundlagen - Template Matching. [Online] [Zitat vom: 27. 01 2013.] http://wwwisg.cs.uni-magdeburg.de/bv/files/LV/Grundlagen_der_Computer_Vision/VL/L08_Template%20Matching.pdf.

[13]. **Janda, Florian.** Verkehrszeichenerkennung im automobilen Umfeld basierend auf einer Grauwertkamera. 2011.

[14]. **Phillip Ian Wilson, Dr. John Fernandez.** FACIAL FEATURE DETECTION USING HAAR CLASSIFIERS. [Online] 2006. [Zitat vom: 27. 01 2013.] http://nichol.as/papers/Wilson/Facial%20feature%20detection%20using%20Haar.pdf.

[15]. **Cognotics.** Cognotics - Resources for cognitive robotics. [Online] T & L Publications, Inc. , 2007. [Zitat vom: 02. 02 2013.] http://www.cognotics.com/opencv/servo_2007_series/part_2/sidebar.html.

[16]. **Bild.** Wikipedia. [Online] [Zitat vom: 04. 02 2013.] http://en.wikipedia.org/wiki/HSL_and_HSV.

[17]. **Wilhelm BURGER, Mark J. BURGE.** *Digitale Bildverarbeitung.* s.l. : Springer Verlag Berlin-Heidelberg, 2005. ISBN 3-540-21465-8.

[18]. **Franz, M.O.** Farbräume - Industrielle Bildverarbeitung. [Online] 2007. [Zitat vom: 27. 01 2013.] http://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=2&ved=0CE0QFjAB&url=http%3A%2F%2Fwww.ios.htwg-konstanz.de%2Fjoomla_mof%2Findex.php%3Foption%3Dcom_docman%26task%3Ddoc_download%26gid%3D90%26%26Itemid%3D102&ei=OrQ5UbmzBYGp4ATWooGQAw&usg=AFQjCNGqd.

[19]. **Hager, G. D.** Department of Computer Science. [Online] [Zitat vom: 07. 02 2013.] http://www.cs.jhu.edu/~hager/Teaching/cs461/Notes/LinearFiltering.pdf.

[20]. **RasterGRID.** Efficient Gaussian blur with linear sampling. [Online] 2010. [Zitat vom: 03. 02 2013.] http://rastergrid.com/blog/2010/09/efficient-gaussian-blur-with-linear-sampling.

[21]. **Bild.** rl7. [Online] [Zitat vom: 10. 02 2013.] http://www.rl7.bmstu.ru/rus/Library/Statistic/Gauss2D.gif.

[22]. **OpenCV.** Open CV Documentation Gaussian Blur. [Online] [Zitat vom: 10. 02 2013.] http://docs.opencv.org/doc/tutorials/imgproc/gausian_median_blur_bilateral_filter/gausian_median_blur_bilateral_filter.html.

[23]. *A computational approach to edge derection.* **Canny, John.** VOL. PAMI-8, NO. 6:679-698, 1986, Bd. IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE.

[24]. **OpenCV.** Open CV Documentation Canny Edge. [Online] [Zitat vom: 10. 2 2013.] http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html.

[25]. **CSE.** Canny Edge Detection. [Online] 23. 03 2009. [Zitat vom: 02. 02 2013.] http://www.cse.iitd.ernet.in/~pkalra/csl783/canny.pdf.

[26]. **OpenCV.** Open CV Hough Line Theorie. [Online] 2013. [Zitat vom: 10. 02 2013.] http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/hough_lines/hough_lines.html.

[27]. —. Open CV Hough Circle Theorie. [Online] 2013. [Zitat vom: 10. 02 2013.] http://docs.opencv.org/trunk/doc/tutorials/imgproc/imgtrans/hough_circle/hough_circle.html.

[28]. **Android.** Android Developer. [Online] [Zitat vom: 27. 01 2013.] http://developer.android.com/reference/android/media/MediaRecorder.html.

[29]. —. Android Developer. [Online] [Zitat vom: 27. 01 2013.] http://developer.android.com/reference/android/view/SurfaceView.html.

[30]. **Jiqiang Song, Michael R. Lyu.** A Hough transform based line recognition method utilizing both parameter space and image space. [Online] 19.02.2004. [Zitat vom: 02. 02 2013.] http://www.cse.cuhk.edu.hk/~lyu/paper_pdf/journal-PR-jiqiang.pdf.

[31]. **Bild.** ADFC BW. [Online] http://www.adfc-bw.de/uploads/RTEmagicC_StVO_Schilderwald_B31_01.JPG.JPG.

[32]. **Christian Große Lordemann, Martin Lambers.** *Objekterkennung in Bilddaten.* Westfälische Wilhelms-Universität Münster : http://wwwmath.uni-muenster.de/u/lammers/EDU/ws03/Landminen/Abgaben/Gruppe9/Thema09-ObjekterkennungInBilddaten-ChristianGrosseLordemann-MartinLambers.pdf, 2003/2004.

Abbildungsverzeichnis

Abkürzungsverzeichnis

ADT
: Android Developer Toolkit 13

API
: Application Programming Interface 13

AVD
: Android Virtual Device 13

C/A Code
: Coarse/Aquisition Code 10

GPS
: Global Positioning System 5, 9

GPX
: GPS Exchange Format 5

HD
: High Definition 6

IDE
: Integrated Development Environment 13

JOSM
: Java OpenStreetMap 12

JRE
: Java Runtime Environment 53

MVC
: Model-View-Controller 39

ODbL
: *Open Database Licence* 12

OpenCV
: Open Source Coputer Vision 14

OSM
: Open Street Map 5, 11

ROI
: Region Of Interest 7, 47

SDK
: Software Development Kit 13

VfW
: Video for Windows 15

WP
: Waypoint 6

Eidesstattliche Erklärung

Hiermit versichere ich, dass ich diese Bachelorarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich diese Bachelorarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, 11. März 2013

Philipp Unger