Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Diplomarbeit

Dennis Berger

Lane Inference System für den Fahrspuralgorithmus
Three Feature Based Lane Detection Algorithm (TFALDA)

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Dennis Berger

Lane Inference System für den Fahrspuralgorithmus Three Feature Based Lane Detection Algorithm (TFALDA)

Diplomarbeit eingereicht im Rahmen der Diplomprüfung
im Studiengang Softwaretechnik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. rer. nat. Stephan Pareigis
Zweitgutachter : Prof. Dr. Andreas Meisel

Abgegeben am 26. August 2008

Dennis Berger

Thema der Diplomarbeit

Lane Inference System für den Fahrspuralgorithmus Three Feature Based Lane Detection Algorithm (TFALDA)

Stichworte

Lane Inference System, TFALDA, Sobel, Tiefpass, Vertikale Prüfung, Testsystem, Blackboxtest, C++ , Boost, OpenCV

Kurzzusammenfassung

Diese Arbeit dokumentiert die Entwicklung des Lane Inference Systems für den Fahrspuralgorithmus TFALDA. Dabei werden drei Methoden zur Korrektur vorgestellt. Anschließend wird eine Implementation prototypisch angefertigt. Gleichzeitig muss der TFALDA um ein System zur dynamischen Positionierung der Region Of Interest erweitert werden. Welchen Einfluss letztendlich das Lane Inference System auf die Fahrspurerkennung hat, wird mittels wiederholbarer Tests analysiert und anschließend kritisch betrachtet.

Dennis Berger

Title

Lane Inference System für den Fahrspuralgorithmus Three Feature Based Lane Detection Algorithm (TFALDA)

Keywords

lane inference system, TFALDA, sobel, lowpass, vertical check, testsystem, blackboxtest, C++ , Boost, OpenCV

Abstract

This thesis documents the development of the lane inference system for the lane algorithm TFALDA. Therefore, three methods for correction will be presented and their implementation will be demonstrated in prototype. Also, TFALDA has to be extended by a system for dynamic positioning on the region of intererst. The impact of the lane interference system on the lane recognition will be analysed by repeatable tests and subsequently assessed critically.

für Janin

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

Listings

1. Einleitung

Der Three Feature Based Lane Detection Algorithm, kurz TFALDA, wie er von YOUNG-UK YIM UND SE-YOUNG OH beschrieben wurde, ist ein System zur Erkennung von Fahrbahnmarkierungen und in der Folge geeignet, ein Fahrzeug auf einer Fahrspur zu halten, während es autonom fährt. In dieser Arbeit wird die letzte Phase der drei Phasen des TFALDA, das Lane Inference System, entworfen und implementiert. Anschließend wird ein Testsystem entwickelt, mit dem gezeigt werden kann, welchen Einfluss das Lane Inference System auf die Fahrspurerkennung hat und vor allem, mit welcher Wahrscheinlichkeit die Fahrspur getroffen wurde.

1.1. Aufbau

Für das Projekt steht ein 1:10 Modellfahrzeug zur Verfügung, welches über Sensoren, einen PC und eine Kamera verfügt. Bei den Sensoren handelt es sich um Ultraschallsensoren vorne und hinten, Infarotsensoren an den Seiten sowie Hallsensoren an der Radaufhängung. Auf dem Fahrzeugdach ist eine Kamera montiert, welche Schwarzweissbilder in einer Auflösung von 752x480 Pixel mit einer Frequenz von 50 Bildern pro Sekunde liefert. Sie ist nach vorne gerichtet und nicht geneigt, sodass der Horizont etwa in der Bildmitte liegt. Die verwendete Linse hat einen Bildwinkel von 155-160°, damit ist es möglich, in einer Kurve die Fahrbahnmarkierung direkt vor dem Fahrzeug noch zu erkennen. Der PC ist mit einem 1GHZ VIA Prozessor ausgestattet, auf diesem wird das Programm zur Fahrspurerkennung laufen.

1.1.1. Teststrecke

Die Strecke, auf der das Modellfahrzeug fahren soll, richtet sich nach den Vorgaben des Carolo-Cup Regelwerks und wird folgende Kriterien erfüllen. Auf der Strecke dürfen sich keine Hügel oder Vertiefungen befinden, somit wird die Strecke insgesamt vollständig eben sein. Es wird keine toten Streckenabschnitte geben, es wird im Gegenteil ein Rundkurs entworfen. Aus diesem Rundkurs kann nicht entwichen werden, etwa durch Ausfahrten. Eine

Skizze der Fahrbahn in Abbildung 1.1.1 zeigt, welche Maße für diese Strecke gelten (vgl. Braunschweig, 2008, S.6f).

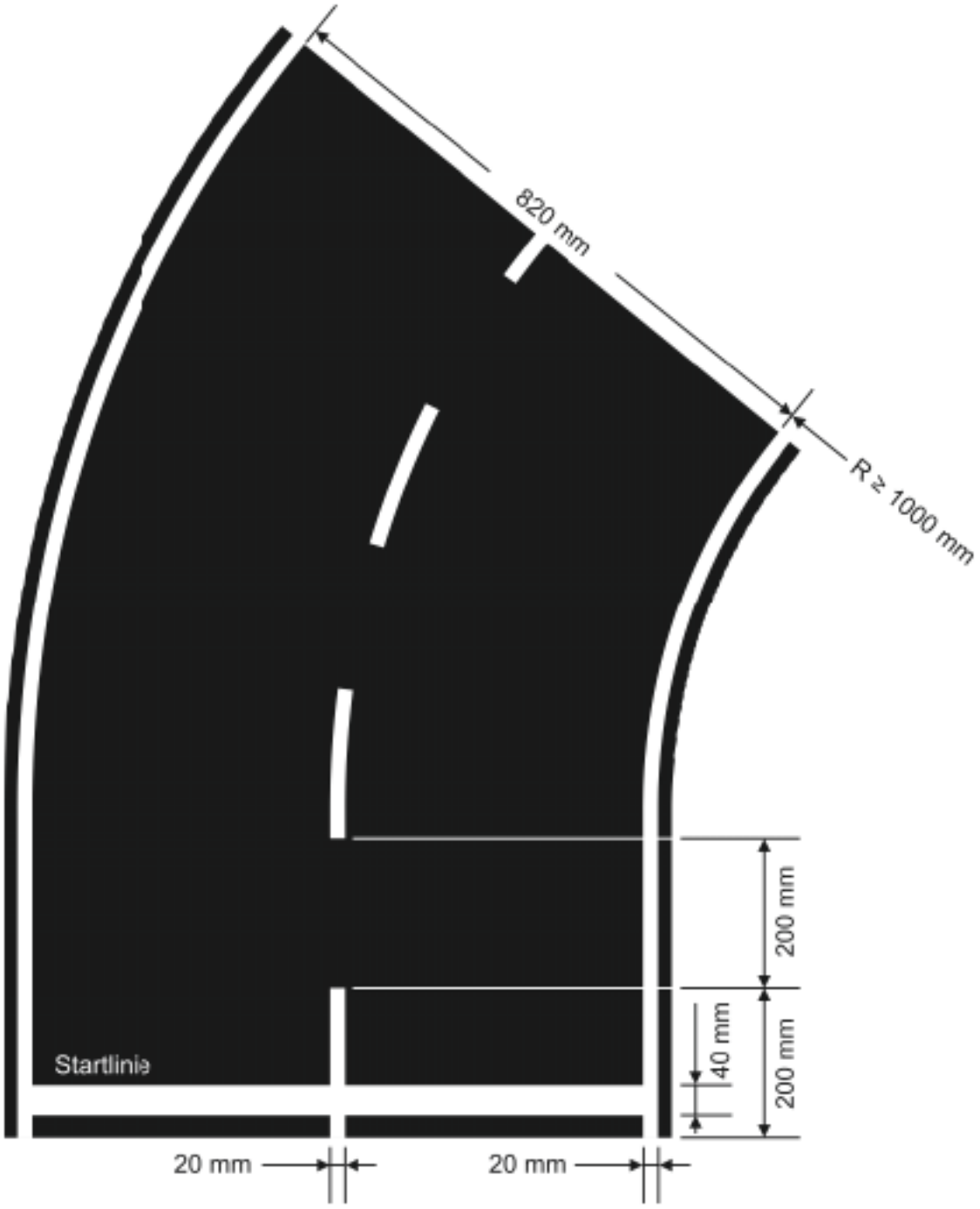

Abbildung 1.1.: Streckenabschnitt mit Maßen

1.2. Ziel

Ziel dieser Arbeit ist die Entwicklung eines Lane Inference Systems für den Algorithmus TFALDA. Dieses System soll es ermöglichen, die Stabilität des TFALDA zu erhöhen. Dies soll dadurch gewährleistet sein, dass das Lane Inference System als einzige Komponente eine globale Sicht auf die Fahrspur hat. Während der TFALDA nur auf den einzelnen ROIs arbeitet, wird eine globale Sicht es ermöglichen zu prüfen, ob die vom TFALDA gefundenen Lanes insgesamt zu einer gültigen Fahrspur führen. Dies gilt es anschließend mit Hilfe von wiederholbaren Tests zu zeigen, deshalb ist dafür ein Testsystem zu entwerfen.

1.3. Vorgehen

In Kapitel 2 werden Ideen gesammelt und ein Entwurf für das Lane Inference System vorgestellt.

Kapitel 3 zeigt eine prototypische Implementation des vorher entworfenen Modells. Dabei werden ebenfalls die verwendeten Bibliotheken vorgestellt.

In Kapitel 4 wird ein Testsystem entworfen und implementiert. Anschließend werden die Tests des Lane Inference Systems durchgeführt und diskutiert.

Abschließend wird ein Fazit gezogen und das entworfene System kritisch betrachtet.

1.4. Begriffe

In diesem Abschnitt werden für die folgenden Kapitel wichtige Begriffe geklärt. Es kann ebenfalls das etwas kürzere Glossar verwendet werden.

TFALDA Der TFALDA besteht aus drei Phasen. Der Vorverarbeitung, dem automatischen Kantenerkennungssystem und einem Bewertungssystem. Im Folgenden wird von Preprocessing, Automatic Lane Detection und Lane Inference System gesprochen. Die gefundenen Fahrspuren, im Folgenden Lanes genannt, werden an die Fahrzeugsteuerung übergeben. Abbildung 1.2 zeigt das Blockdiagramm, wie es im Artikel Yim und Oh (2003) beschrieben wird. Es wird empfohlen, die Artikel Berger (2008, vgl. S.5ff) sowie Yim und Oh (2003) für einen detaillierteren Einblick zu lesen.

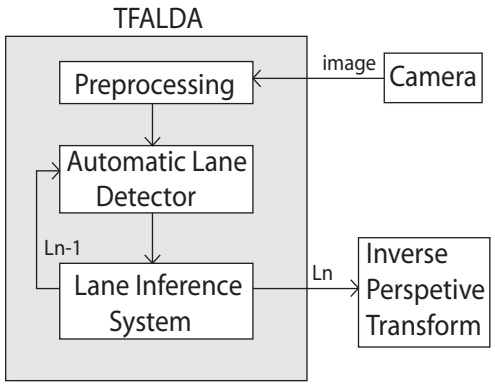

Abbildung 1.2.: TFALDA Blockdiagramm nach YOUNG UK YIM UND SE-YOUNG OH (vgl. Yim und Oh, 2003, S.220)

ROI Die Region Of Interest ist ein Bereich des Bildes, der definiert wird und in dem Algorithmen wie Resampling, Sobel oder der TFALDA arbeiten. In dieser Arbeit werden vier ROIs definiert, welche eine feste Höhe besitzen, jedoch in der Breite variabel sind (vgl. Abbildung 2.6, S.19).

Lane, Fault Lane Eine Lane ist die vom TFALDA durch die Distanzfunktion gefundene Fahrbahnmarkierung innerhalb einer ROI. Die Distanzfunktion ist ihrerseits Teil des Automatic Lane Detectors. Das Lane Inference System kann eine Lane als ungültig markieren, dann wird von einer Fault Lane gesprochen.

Sobel-Operator Der Sobel-Operator ist ein Filter in der Bildverarbeitung, das den Effekt einer Kantenverstärkung hat. Dazu kann der Sobel mittels Faltung einer Matrix, für diese Arbeit eine 3x3 Matrix, ein Gradientenbild erzeugen. Das Gradientenbild weist je nach Art der Matrix eine Kantenverstärkung in horizontaler oder in vertikaler Richtung auf. Je nach Anforderung kann entschieden werden, welche Matrix gebraucht wird. In der Implementation wird eine dynamische Variante gewählt, welche je nach Winkel der gefundenen Lane die Richtung der Matrix für die Berechnung des nächsten Bildes vorgibt.

2. Entwurf des Lane Inference Systems

In diesem Kapitel wird das Lane Inference System vorgestellt, dabei werden die einzelnen Komponenten, aus denen das System besteht, detailliert beschrieben und deren Zusammenspiel dargestellt. YOUNG UK YIM UND SE-YOUNG OH haben bereits ein solches System vorgesehen und kurz beschrieben (vgl. Yim und Oh, 2003, S.221). Ihre Idee des Plausibilitätschecks der Fahrbahnbreite mit Hilfe eines Tiefpassfilters, der über die letzten gefundenen Lanes gelegt wird, wird hier in dieser Arbeit ebenfalls wieder aufgegriffen und prototypisch implementiert. Außerdem werden Korrekturmechanismen für die Platzierung der ROIs vorgestellt. Diese werden im Wesentlichen aus Beobachtungen der Blickrichtung des Menschen, während er geradeaus oder in eine Kurve fährt, entworfen. Auch wird jede Seite der Fahrbahn einer vertikalen Prüfung unterzogen, so dass gewährleistet werden kann, dass es sich um eine zusammenhängende Fahrbahnmarkierung handelt. Das System kann als Zustandsautomat mit Hilfe von UML2 modelliert werden. SCOTT W. AMBLER gibt in Ambler (2004) eine gute Einführung in die Syntax. Als Kurzreferenz sei ein Auszug dieses Buches empfohlen Ambler (2006). Abbildung 2.1 zeigt das Lane Inference System als Zustandsautomat, dabei gilt folgende Legende:

`e101`	Event — neue Lanedaten kommen in das System
`lis`	Instanz der Klasse LaneInferenceSystem
`lowpass`	Tiefpassfilterung — siehe Kapitel 2.1.2
`is_minmax_road`	Minima- und Maximaprüfung — siehe Kapitel 2.1.4
`roadwidth_is_ok`	Prüfung der Spurbreite nach Tiefpass — siehe Kapitel 2.1.3
`is_vertically_ok`	Vertikale Prüfung — siehe Kapitel 2.2
`dec_ERRcount`	Fehlerzähler dekrementieren
`inc_ERRcount`	Fehlerzähler inkrementieren
`save_lane`	Lane speichern — siehe Kapitel 2.1.2
`take_old_lane`	Ersetzt die aktuelle Lane — siehe Kapitel 2.4

Tabelle 2.1.: Legende des Zustandsautomaten

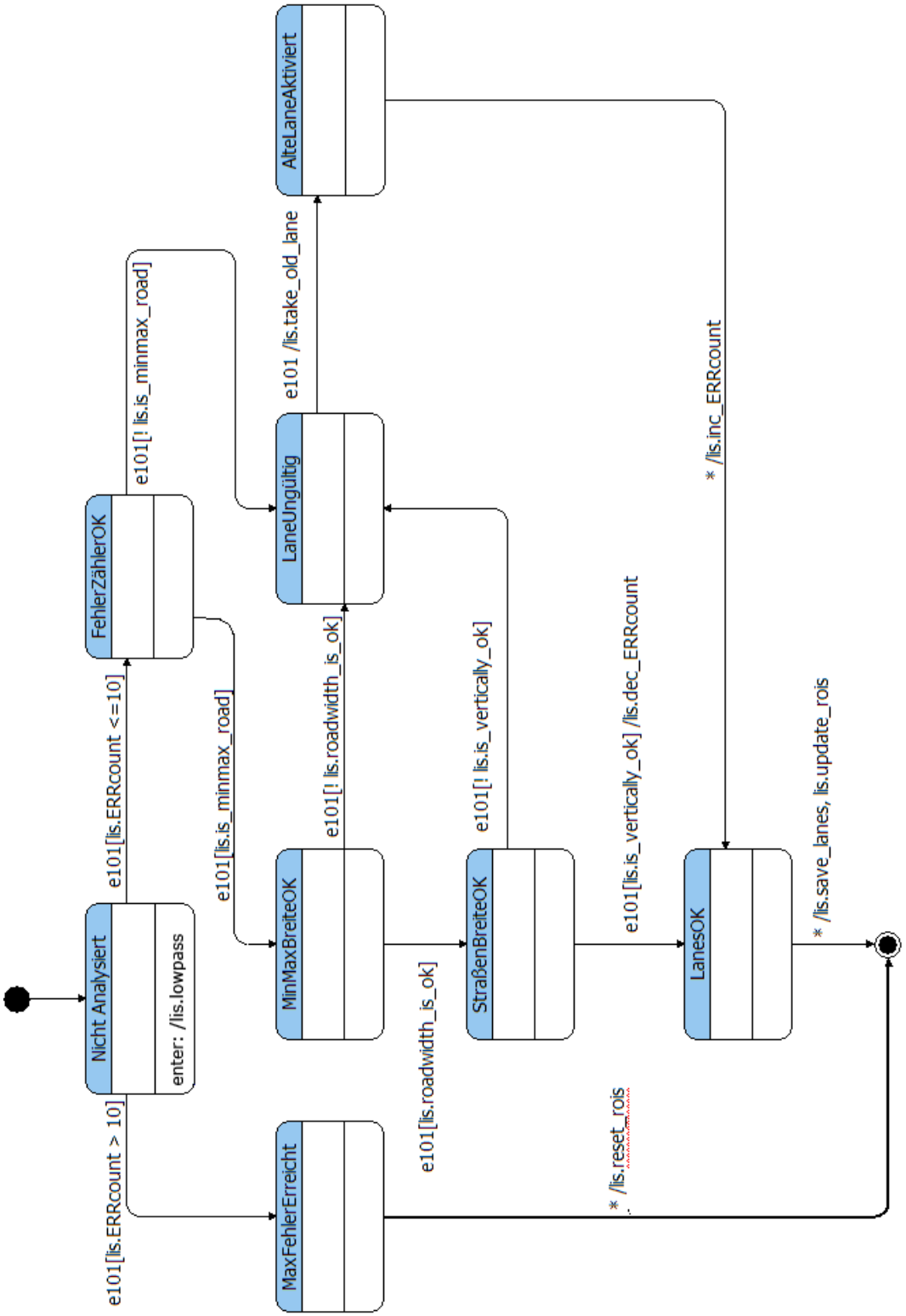

Abbildung 2.1.: Zustandsautomat des Lane Inference Systems

2.1. Plausibilitätscheck der Fahrspurbreite

Um die Plausibilität einer gefundenen Lane beurteilen zu können bedarf es eines Systems, welches die Fahrbahnbreite erkennt und entscheiden kann, ob eine gefundene Lane des TFALDA auch im Gesamtbild zu einer gültigen Fahrbahn führt. Diesem System könnte also zuerst ein empirisch ermittelter Wert der Fahrspurbreite mitgeteilt werden, gegen den validiert wird. Ein Toleranzbereich von max 10% sei zugelassen. Dennoch ergeben sich erheblichere Probleme als zuerst vermutet. Anhand der folgenden Abbildungen 2.2 und 2.3 wird deutlich, warum das so ist.

2.1.1. Scheinbare Veränderung der Fahrspurbreite

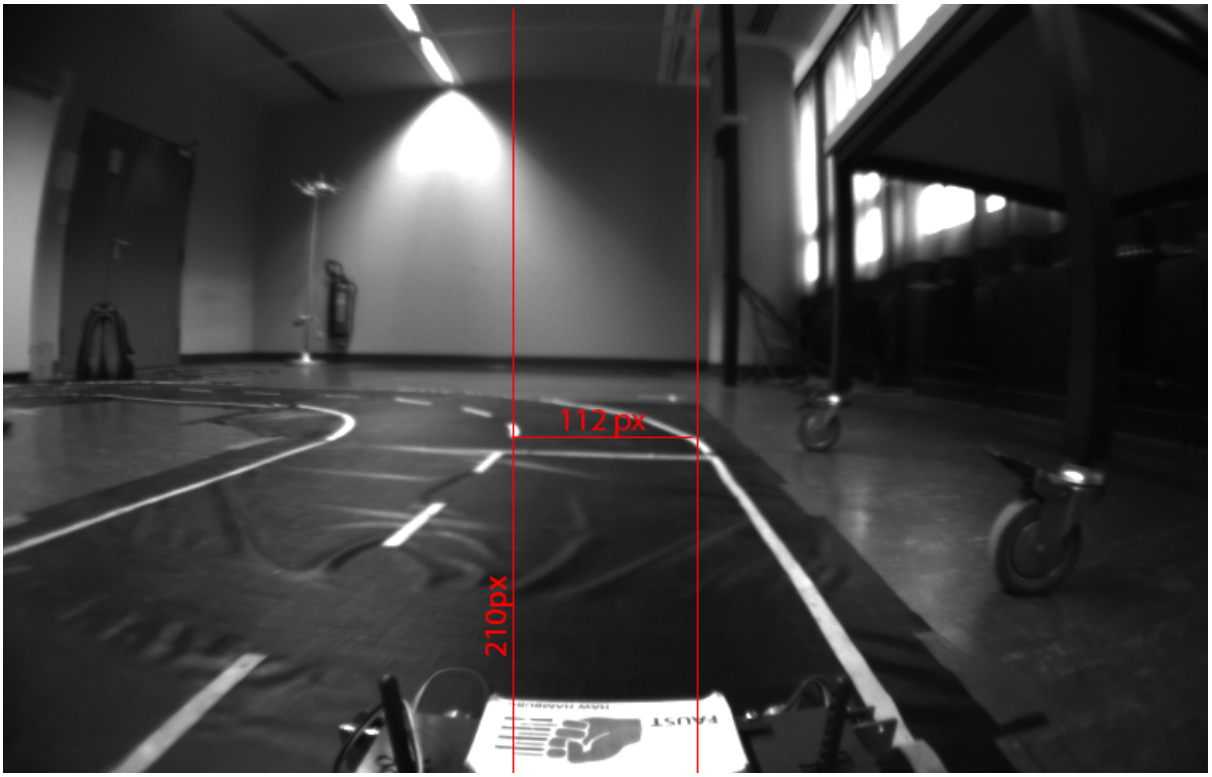

Abbildung 2.2.: Fahrspurbreite in der Ferne — 112 Pixel breit

Es ist zu sehen, dass die Fahrspur im Bereich der oberen ROI 112 Pixel breit ist. Die ROIs sind dabei so angeordnet, wie es in Abbildung 2.6 zu sehen ist. Die exakten Positionen können der Tabelle 2.4 entnommen werden. Entscheidend ist, dass hier die Höhe, in welcher die Fahrspurbreite gemessen wurde, 210 Pixel vom unteren Bildrand beträgt.

Im Vergleich zu Abbildung 2.2 zeigt Abbildung 2.3 eine Kurvenfahrt, auch hier ist in der Höhe von 210 Pixel die Breite der Fahrbahn eingezeichnet. Diesmal werden 175 Pixel gemessen. Durch die unterschiedlichen Messwerte ergibt sich eine scheinbare Veränderung der Fahrbahnbreite. Es leuchtet ein, dass die Fahrbahn in Wirklichkeit nicht breiter geworden ist, sondern die verschiedenen Beobachtungen der Tatsache entstammen, dass nicht direkt von oben auf die Fahrbahn geschaut wird, sondern aus Sicht des Modellfahrzeugs mit einem definierten Neigungs- und Blickwinkel. Der Veränderung, die unsere Wahrnehmung durch die Optik erfährt, sind wir uns beim Fahren eines Kraftfahrzeugs jedoch nicht bewusst. Das Gehirn vollzieht hier eine Ergänzung.

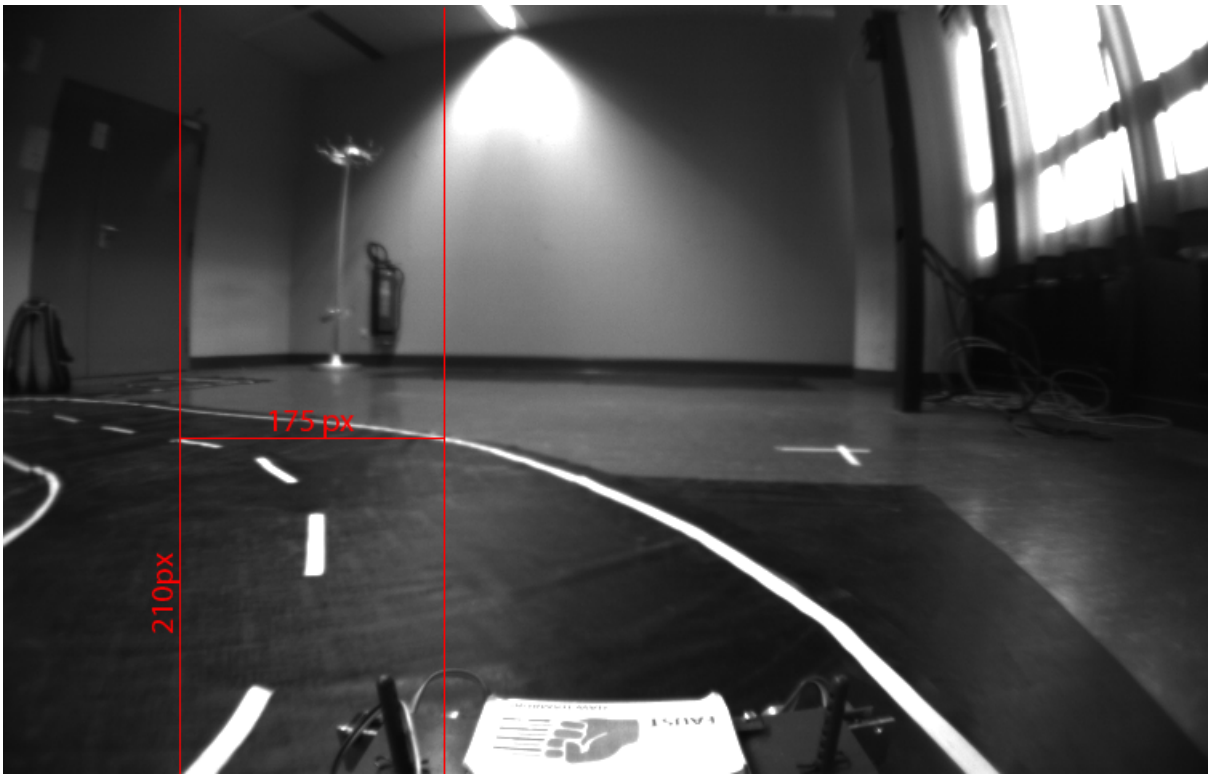

Abbildung 2.3.: Fahrspurbreite in der Ferne – 175 Pixel breit

Da sich durch die optische Verzerrung eine Fahrspur in ihrer Breite scheinbar erheblich verändert, und sich eine tatsächliche Veränderung von diesen Beobachtungen schwer trennen und ablesen liesse, scheidet eine Steuerung auf Basis einer fest definierten Fahrspurbreite aus. Der nachstehenden Tabelle können die durch die Optik unterschiedlichen Meßwerte der Fahrspurbreite in Pixel entnommen werden.

Sequenznummer	Breite in Pixel
80	102
82	104
84	96
86	102
88	110
90	114
92	122
94	181
96	161
98	151
100	138

Tabelle 2.2.: Scheinbare Änderung der Fahrspurbreite

2.1.2. Tiefpass: permanente Angleichung der Fahrspurbreite

Die Lösung liegt in der dynamischen Anpassung der Fahrspurbreite während des Fahrens. Zur Modellierung eines solchen Systems werden zunächst die Randbedingungen geprüft, Anforderungen spezifiziert und diese als Fragen formuliert.

1. Wie viele Bilder pro Sekunde kann die Kamera liefern?
2. Wie viele Bilder pro Sekunde kann das System verarbeiten? Reicht der Prozessor aus?
3. Wie schnell kann maximal gefahren werden?
4. Wie viele Bilder pro Strecke braucht man mindestens, um eine zuverlässige Vorhersage treffen zu können?
5. Wie schnell soll gefahren werden?
6. Wie weit kann vorrausgeschaut werden?

Es werden daraufhin folgende Größen festgehalten.

1. Die Kamera liefert 50 Bilder pro Sekunde.
2. Das System kann ca 70 Bilder pro Sekunde verarbeiten bei einer CPUlast von 100%.
3. Das Testfahrzeug fährt 50 km/h Maximalgeschwindigkeit.
4. Sechs Bilder pro Meter müssen reichen, um die Fahrbahn inklusive Kurven zu erkennen.

5. Daraus ergibt sich eine Maximalgeschwindigkeit von 8,3 Meter pro Sekunde. Dies entspricht etwa 30 km/h.
6. Die gewählte Kameraneigung und -höhe, das Objektiv sowie die Lage der ROIs erlauben eine Vorraussicht von 1,8 bis 2 Meter. Das obere ROI deckt in etwa 1 - 1,2 Meter ab.

In Tabelle 2.2 ist die scheinbare Änderung der Fahrspurbreite dargestellt, die zwischen 96 und 181 Pixel, was einer Differenz von 85 Pixel entspricht, variiert. Die aufgenommene Bildsequenz simuliert eine Geschwindigkeit von 15 km/h. Für die definierte Maximalgeschwindigkeit von 30 km/h wird deshalb jedes zweite Bild der Sequenz genommen. Somit liegen von dem Messpunkt mit der scheinbar kleinsten Breite zu dem Messpunkt mit der scheinbar größten Breite sechs Bilder vor. Deshalb wird festgelegt, dass für die Änderung der Breite der oberen ROI um 85 Pixel mindestens sechs Bilder vorliegen.

Für die digitale Simulation eines analogen Tiefpassfilters gilt folgende Gleichung,

$$y_n = \alpha x_n + (1 - \alpha) y_{n-1} \tag{2.1}$$

wobei

$$\alpha = \frac{\Delta t}{RC + \Delta t} \tag{2.2}$$

und

1. x_n n-tes Eingabesignal
2. y_n n-tes Ausgabesignal
3. Δt Intervall
4. RC Konstante

sind.

Um die Fahrspurbreite sukzessive messen und anpassen zu können, werden weitere Messpunkte definiert, und zwar die jeweils oberen und unteren Kanten der ROIs, wie es exemplarisch in folgender Abbildung 2.4 zu sehen ist. Die rot eingezeichneten Linien markieren die gefundenen Lanes des TFALDA. Die blauen Linien markieren die gemessenen Abstände vom End-, bzw. Anfangspunkt einer gegenüberliegenden Lane zur anderen.

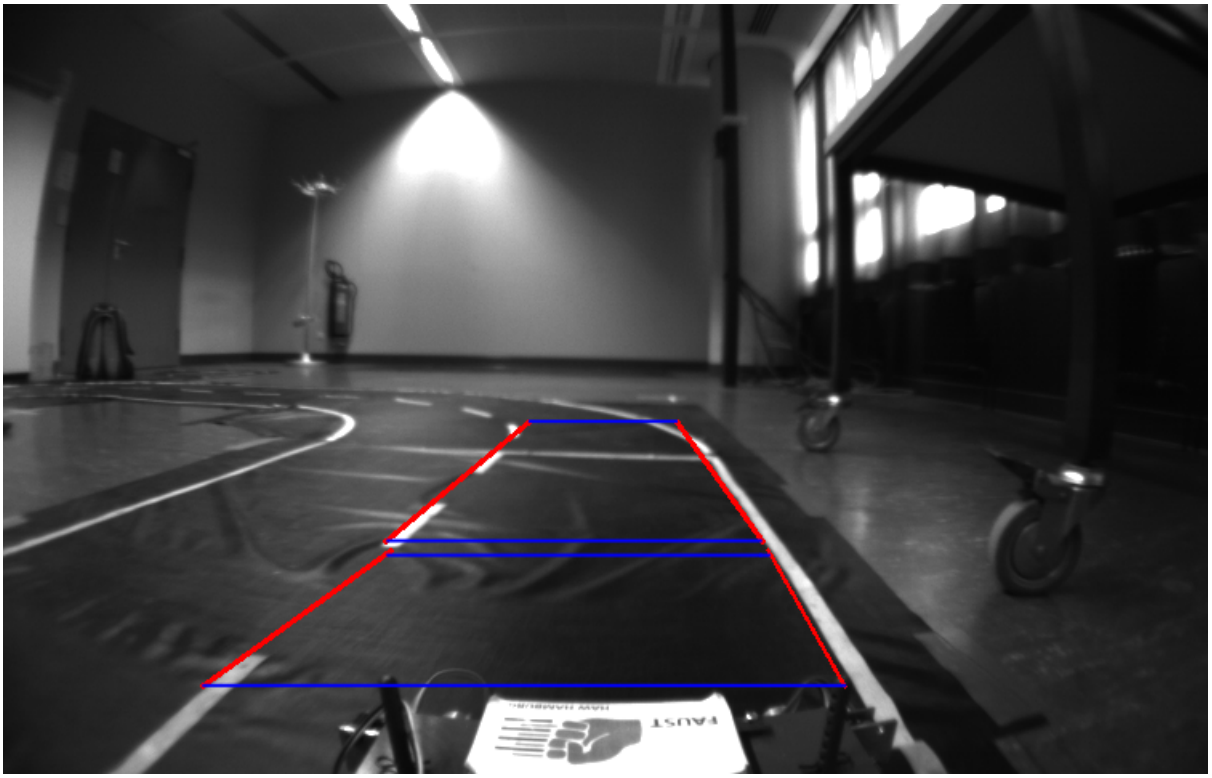

Abbildung 2.4.: Messlinien der Fahrspurbreite

Für die sukzessive Anpassung der Fahrspurbreite gelten daher folgende Werte.

1. x_n aktuelle Fahrspurbreite in Pixel
2. y_n errechnete Fahrspurbreite in Pixel
3. RC Konstante hier 85; denn die maximale Änderung pro Intervall beträgt 85 Pixel
4. Δt Intervall hier 14,166; denn die Strecke, die zur maximalen Veränderung der Fahrbahnbreite führt, beträgt mindestens einen Meter, und es werden sechs Bilder pro Meter gefordert. Demnach beträgt die maximale Änderung pro Bild von $\Delta t = \frac{85}{6} = 14,16$ Pixel.
5. und eingesetzt in die Gleichung 2.2 weiter ein α von 0,1429

Wird dies in obige Gleichung 2.1 eingesetzt kann daraus folgender Softwareentwurf abgeleitet werden:

Listing 2.1: Tiefpassfilter im Lane Inference System

```
1  class LaneInferenceSystem
2  {
3  int X_top_width[2];
4  int X_bottom_width[2];
5  static int Y_top_width[2];
6  static int Y_bottom_width[2];
7
8  void lowpass(const ROI& roi, double dt, int i){
9    int RC=85;
10   double alpha=dt/(RC + dt);
11   X_bottom_width[i] = roi[i].left().qk().x - roi[i].right().qk().x;
12   X_top_width[i] = roi[i].right().pi().x - roi[i].right().pi().x;
13
14   Y_bottom_width[i] = alpha * X_bottom_width[i] +
15                       (1-alpha) * Y_bottom_width[i];
16   Y_top_width[i] = alpha * X_top_width[i] + (1-alpha) *
17                    Y_top_width[i];
18 }
```

Da bei einem Fahrzeug die aufgenommenen Bilder nicht als Folge vorliegen sondern in Echtzeit in das System kommen, müssen die berechneten Werte, also das repräsentierte y in den Zeilen 5 und 6, über die Zeit hin aktualisiert werden. Es ist daher nötig, y statisch zu halten. Δt ist als Parameter gewählt worden, zu sehen in Zeile 8, um das Tiefpassfilter abhängig von der Geschwindigkeit steuern zu können. Denn bei einer Veränderung der Geschwindigkeit ändert sich auch die Menge an Bildern, die pro zurückgelegter Strecke erhalten wird.

2.1.3. Tiefpass: Entscheidungssystem

Die alleinige Tiefpassfilterung der letzten Spurbreiten kann jedoch keine Auskunft darüber geben, ob eine Fahrbahn noch als gültig oder als ungültig eingestuft wird. Hierzu bedarf es zusätzlich eines Entscheidungssystems. Dies wurde anhand der Anforderungen aus Kapitel 2.1.2, speziell Anforderung Nr. 4, modelliert. Das Entscheidungssystem fragt das Orakel, ob die Spurbreite in Ordnung war. Ist dies der Fall, wird die aktuell gefundene Lane akzeptiert. Ist dies nicht der Fall, orientiert sich das System an der letzten gültigen Lane und verwirft die aktuelle. Das Orakel ist folgendermaßen implementiert:

Listing 2.2: Orakel für eine gültige Spurbreite

```
1 boolean isLaneAcceptable(Lane l){
2   const int threshhold=11;
3   int difference=abs(roadwidth - lastRoadwidth);
4
5   if ( difference <= threshhold)
6     return true;
7   else
8     return false
```

An dieser Stelle sei noch betont, dass das zu jeder Zeit angewandte Tiefpassfilter die aktuelle Spurbreite aktualisiert, so dass bei einer als ungültig erkannten Fahrbahnbreite zwar die alte Lane genommen wird, die Fahrbahnbreite, das y unseres Systems, jedoch in jedem Fall durch das Filter verändert wird. Wie stark die Veränderung der Fault Lane die Spurbreite beeinflusst, wird durch die Wahl der Parameter des Tiefpassfilters geregelt.

2.1.4. Maximale und minimale Fahrspurbreite

Dennoch lässt sich sagen, dass die Fahrbahn nicht beliebig schmal oder breit werden kann. Es gibt Grenzen, die für eine Straße in diesem Modell gemäß Carolo-Cup Wettkampfbedingungen gelten (vgl. Braunschweig, 2008, S.10). Der Toleranzbereich wird, wie schon im Kapitel 2.1.1 erkannt, damit groß, aber eben nicht unendlich. Konkret, ausgehend von Abbildung 2.4, sind die maximalen und minimalen Fahrspurbreiten definiert und in Tabelle 2.3 gezeigt.

Aus diesen Bedingungen lässt sich eine Komponente formulieren, die Maxima und Minima der Fahrspur prüft und ggf. korrigiert. Hierzu wird wie im Kapitel zuvor ein Entscheidungssystem aufgebaut. Lediglich das Orakel wird ausgetauscht und durch eines ersetzt, welches diese Prüfung vornimmt. Der Quelltext kann im Anhang nachgeschlagen werden (vgl. Anhang A.4, S.64).

ROI	Messpunkt der Lane	min Größe	max Größe
oben	oben	45	220
oben	unten	200	272
unten	oben	200	272
unten	unten	320	520

Tabelle 2.3.: Maximale und Minimale Fahrspurbreite

2.2. Vertikale Prüfung

Eine zweite Komponente stellt einen Zusammenhang zwischen den Lanes einer Seite her, also zwischen denen der linken Fahrbahnmarkierung (des Mittelstreifens) oder der rechten (der Außenmarkierung). Fahrbahnmarkierungen definieren generell Pfade, entlang denen der Verkehr fließen soll. Diese Pfade sind durchgängig, obgleich eine Fahrbahnmarkierung, wie es bei dem Mittelstreifen der Fall ist, unterbrochen sein kann. Daraus lässt sich ableiten: Bewegt sich das Fahrzeug zwischen den eine Fahrbahn rechts und links begrenzenden Pfaden, so bleibt es auf der Fahrbahn.

Nun sei der TFALDA mit zwei ROIs pro Seite initialisiert. Betrachtet man die Außenmarkierung und die Tatsache, dass der TFALDA jeweils eine Lane pro ROI liefert, so liegen für zwei ROIs damit mathematisch zwei Strecken vor, welche zusammen einen Pfad ergeben müssen. Dieser sollte dann einem Pfad entsprechen, der sich aus der Approximation der Außenmarkierung ergibt.

Formal ausgedrückt wird der X-Wert des Punktes P_3 gesucht, der auf der Geraden liegt, die durch die Punkte P_1 und P_2 der oberen Lane L_1 geht. Der Punkt P_3 wird deshalb wie folgt bestimmt. Sei der Y-Wert, genannt y_3, von P_3 definiert durch $\frac{y_4 + y_2}{2}$, so folgt aus der Zweipunkteform der Geradengleichung (vgl. Kemnitz, 1999, S.251):

$$\frac{y_2 - y_1}{x_2 - x_1} = \frac{y_1 - y}{x_1 - x} \tag{2.3}$$

aufgelöst nach x

$$x = x_1 - \frac{x_2 - x_1}{y_2 - y_1}(y_1 - y) \tag{2.4}$$

Daraus ergibt sich der X-Wert x_3 für den Punkt $P_3(x_3|y_3)$ durch Einsetzen in die Gleichung 2.4. Damit ist der Punkt bestimmt.

Diese Komponente enthält ebenfalls ein Entscheidungssystem, welches die X-Werte von P_4 und P_3 vergleicht und, falls eine Abweichung vorliegt, den X-Wert von P_4 auf den von P_3

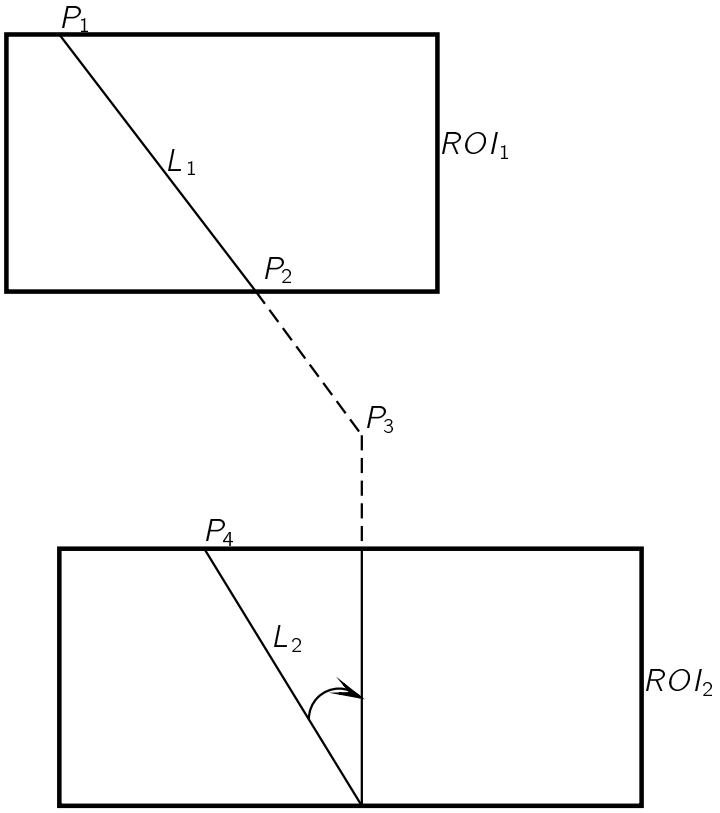

Abbildung 2.5.: Vertikale Angleichung

setzt. Diese Transformation ist in der Abbildung 2.5 als Pfeil dargestellt. Damit wird gleichzeitig die Bedingung gestellt, dass die Lane in ROI_1 die Richtung bestimmt, und die Lane in ROI_2 angeglichen wird. Das System orientiert sich also nicht an der nächstgelegenen unteren ROI, wie man auf Anhieb vermuten könnte, sondern an der entfernteren, oberen ROI. (Andernfalls würde eine stark gekrümmte Kurve wieder relativiert werden, womit das Fahrzeug eventuell aus der Fahrspur kommt. Es wird an dieser Stelle somit klar die Präferenz festgelegt, dass die oberen ROIs, in der Abbildung 2.5 als ROI_1 gekennzeichnet, die Richtung vorgeben. Erinnert sei an dieser Stelle auch an Überholmanöver oder Fahrspurwechsel, bei denen ebenfalls zuerst die oberen ROIs die Richtung ändern.)

2.3. Positionieren der ROIs

YOUNG UK YIM UND SE-QOUNG OH beschreiben bereits, wie der Algorithmus im einzelnen arbeitet (vgl. Yim und Oh, 2003, S.220ff). Dabei gehen sie im Detail darauf ein, wie der Automatic Lane Detector auf einer ROI arbeitet, lassen jedoch offen, wie die ROIs platziert werden müssen. Dieser Abschnitt liefert eine mögliche Lösung, die ROIs dynamisch und sinnvoll zu setzen.

2.3.1. Voreinstellung

Die ROIs werden während der Fahrt selbstständig in einen Bereich positioniert, in dem, im Hinblick auf die letzten gefundenen Lanes, mit der größten Wahrscheinlichkeit die Fahrspurmarkierung weiterführt. Ebenfalls wird ihre Größe angepasst. Zu Beginn einer Fahrt liegen jedoch keine Informationen darüber vor, wo sich die Fahrspur befindet, daher müssen vernünftige Werte für die ROIs bestimmt werden. Den ROIs werden folgende Größen und Positionen zu Beginn zugewiesen: [1]

ROI	Position	Breite	Höhe	XY-Koordinate der linken oberen Ecke
Oben	Links	150	75	195/260
Oben	Rechts	150	75	365/260
Unten	Links	180	85	80/340
Unten	Rechts	180	85	429/340

Tabelle 2.4.: Anfangspositionen der ROIs

Wenn man auf einem von der auf dem Fahrzeug montierten Kamera ausgegebenen Bild die ROIs mit einer roten Linie umrahmt, erhält man Abbildung 2.6

[1] Das Koordinatensystem beginnt mit 0/0 oben links, alle Angaben sind in Pixel.

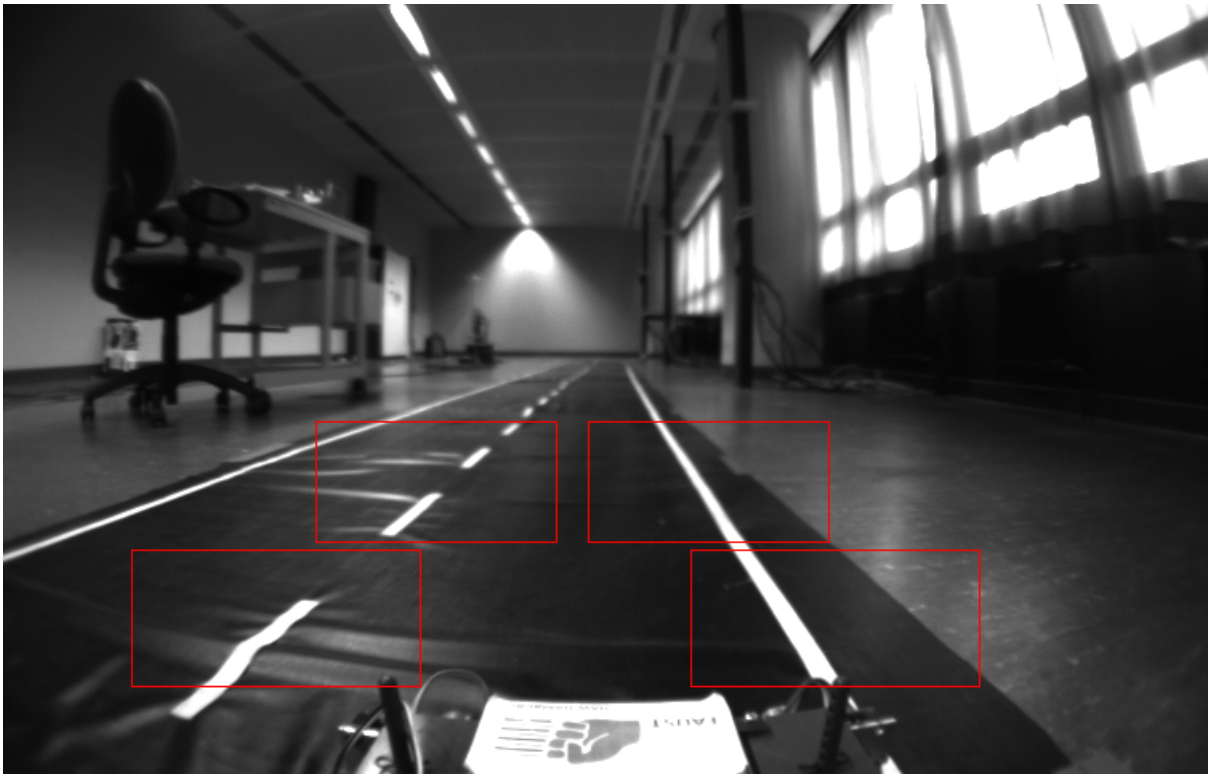

Abbildung 2.6.: Voreinstellung der ROIs

Damit wird natürlich festgelegt, dass die Fahrt auf einem geraden Streckenabschnitt beginnen muss. Stünde das Fahrzeug in einer stark gekrümmten Kurve, hätte das System keine Möglichkeit, nach einer Fahrbahnmarkierung zu suchen, da diese außerhalb der ROIs läge. Die Voreinstellung für die Positionierung der ROIs wird in der Klasse LaneRegion vermerkt (vgl. Anhang A.3, S.68). Bei Bedarf kann das Lane Inference System über die Funktion `reset_rois` die ROIs immer wieder in diese Position bringen. Im Kapitel 2.4 wird ein System für einen Reset vorgestellt, dort wird diese Funktion benutzt.

2.3.2. Dynamisches Positionieren

Für jedes Bild, das ins System kommt, müssen die ROIs neu positioniert werden. Die Position wird mit Hilfe dreier Kriterien bestimmt. Diese sind die vorherige Position *P*, ein Fahrzustand *FZ* und ein Regelsatz *R* für die Größe und Position der ROIs. Abbildung 2.7 zeigt das Aktivitätsdiagramm für einen Ablauf zum Aktualisieren der ROIs. Im ersten Schritt wird die Position der aktuellen ROIs abgefragt und lokal gespeichert, dieser Wert dient als Grundlage für die weitere Berechnung.
FZ sei ein Zustand, der beschreibt, ob sich das Fahrzeug in einer Kurvenfahrt, in einer Geradeausfahrt oder in einer beliebigen anderen Fahrt, z.B einer Kreuzungsfahrt, befindet. Die Kurvenfahrt sei weiterhin in Links- und Rechtsschwenk eingeteilt. Als Standard wird der Standardzustand erzeugt, der keine besonderen Eigenschaften besitzt. Das System kann befragt werden, in welchem Zustand es sich befindet. Dies ist im Aktivitätdiagramm unter *Prüfe Fahrzustand* zu sehen. Für spätere Überholmanöver oder Fahrspurwechsel können weitere Zustände hinzugefügt werden. Es kann auch ein Zustand von außen vorgegeben werden, der den internen überläd. So besteht die Möglichkeit, ein Fahrmanöver nicht nur passiv vom System verarbeiten zu lassen, sondern ihm aktiv vorzugeben, in welchem Zustand sich das Fahrzeug befinden soll. Dies wird bei geplanten Manövern wichtig, z.B dem Ausweichen.

Nachdem der Fahrzustand festgestellt wurde, wird die Korrekturmethode des jeweiligen Zustands aufgerufen, im Standardzustand wird per return die Funktion an dieser Stelle sofort wieder verlassen. Dies ist der einfachste Fall, und ist u.a. für die Geradeausfahrt vorgesehen. In der Kurvenfahrt wird nachempfunden, was ein Mensch tut, der ein Fahrzeug lenkt, nämlich in die Kurve blicken. Für die Positionierung der ROIs bedeutet das folgendes.

Gegeben sei eine Linkskurve, die sich vor dem Fahrzeug befindet, so dass die Kamera bereits einen Teil der Kurve im Bild hat. Die obere rechte ROI ist bereits deutlich breiter und nach links gerückt. Die obere linke ROI ist nicht mit nach links gerückt, und die darin befindliche Lane zeigt weiter eine Geradeausfahrt an. Innen ist der Winkel deutlich spitzer und es kann erst später eine Kurve erkannt werden. Anhand der Breite und Position der oberen rechten ROI wird bestimmt, ob es sich um eine Linkskurve handelt. Überschreitet diese ROI

die Fahrzeugmitte um x Pixel und ist y Pixel breit, so wird in den Zustand Linkskurve gewechselt. Zur neuen Positionierung der ROI wird wie sonst üblich die ROI zunächst über die neu gefundene Lane gelegt und rechts und links ein Abstand z eingefügt. Die Korrekturmethode des Linksschwenks prüft jetzt allerdings, ob sich die ROIs überschneiden. Liegt eine Überschneidung vor, so wird die linke ROI so weit nach links geschoben bis die Überschneidung aufgehoben ist. Auf diese Weise wird der Blick in die Kurve simuliert, um dort nach der neuen Fahrbahnmarkierung zu suchen. Analog gilt dies umgekehrt für eine Rechtskurve.

Zum Schluss wird ein Regelsatz *R* durchlaufen, der prüft, ob die Größe der ROIs innerhalb einer bestimmten Mindest- und Maximalbreite liegt, falls das nicht der Fall ist, werden die ROIs auf die entsprechenden Mindest- oder Maximalwerte angeglichen. Danach wird geprüft, ob die Bildgrenze nach links oder rechts überschritten wurde, d.h. die linke ROI darf nicht bei $x < 0$ beginnen, und die rechte nicht bei $x + width > 754$ aufhören. ROIs, die teilweise außerhalb der durch diese Werte bestimmten Grenze liegen, werden soweit reduziert, dass sie nicht mehr über das Bild hinausreichen. Anschließend werden die neuen Positionen gesetzt, und die Funktion kann verlassen werden.

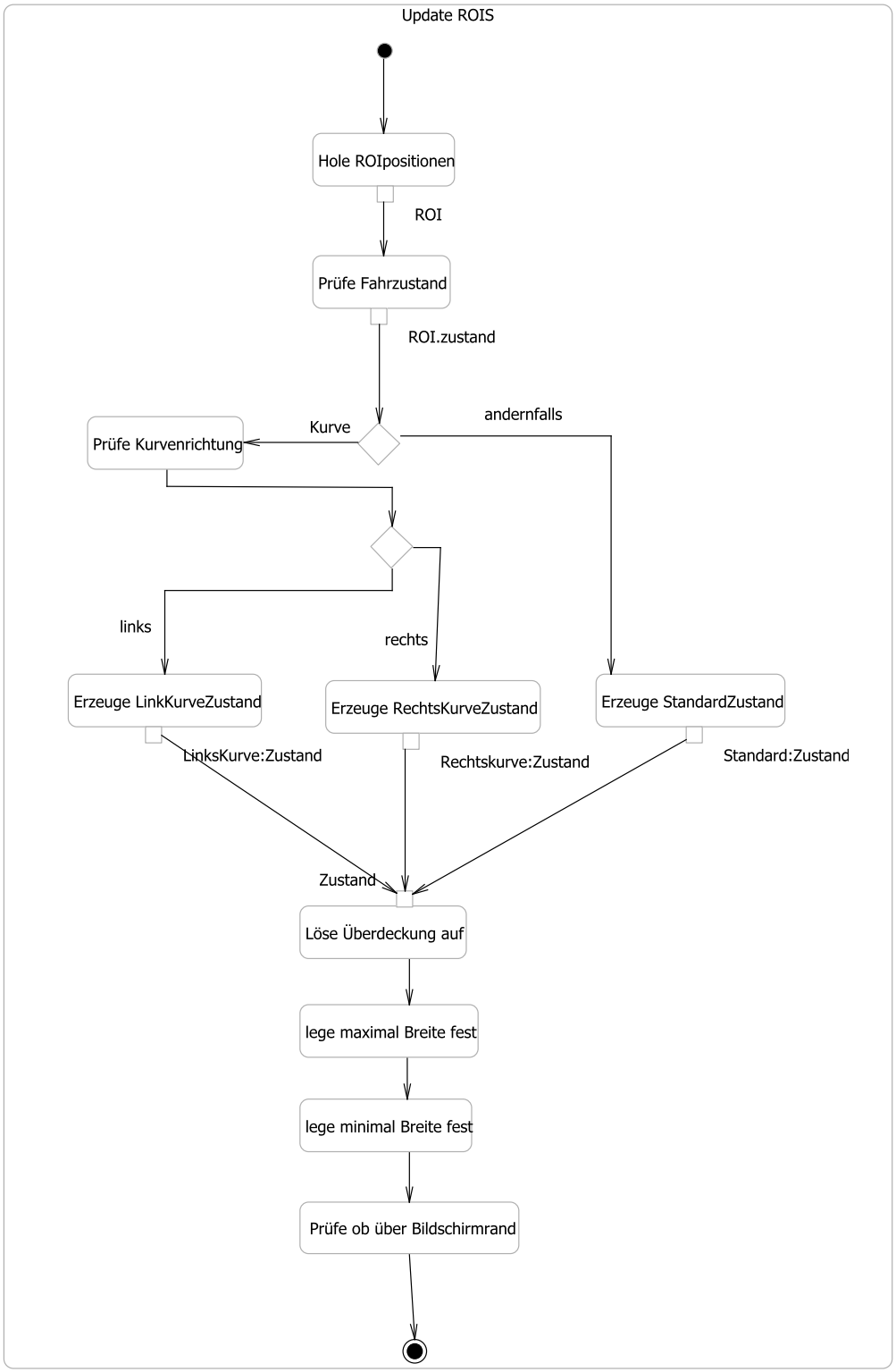

Abbildung 2.7.: Aktivitätsdiagramm zum Ablauf — Update der ROIs

2.4. Resetsystem

Die letzte Maßnahme des Lane Inference Systems zur Korrektur von Fault Lanes ist das Resetsystem. Dies ist ein Mechanismus zum Zurücksetzen der ROIs, der selbstständig vom Lane Inference System oder von außen angestoßen werden kann. Die Autoren LABYARADE U.A. beschreiben in dem Artikel Labayrade u. a. (2006) einen Resetmechanismus, den sie als *Auto Switch Off and Initialization* bezeichnen, der als zusätzlicher Stabilitätsfaktor eingebaut wird. Sie gehen insbesondere darauf ein, dass eine Vielzahl von Situationen eintreten kann, bei denen die Fahrbahn selbst oder das Tracking selbiger verloren gehen kann. In ihrer Arbeit wird das Erkennen, ob ein Verlust vorliegt, über einen *confidence value* geregelt. Liegt dieser über einem definierten *threshhold*, dort 20%, wird ebenfalls ein Fehlerzähler inkrementiert. Liegt er über der Marke von 10, so wird das System neu initialisiert. Diese Arbeit verwendet den Fehlerzähler *ERRcount*, der bereits im Zustandsdiagramm in Abbildung 2.1 zu sehen war. Überschreitet dieser Fehlerzähler eine Schwelle von *N*, in diesem System auf 10 festgelegt, so werden die ROIs auf ihre Anfangspositionen zurückgesetzt. Diese sind die gleichen, die beim Start verwendet werden. Anschließend werden alte Lanes und alte ROIs gelöscht, so dass bereits beim ersten Durchlauf keine Störungen durch alte Daten auftreten können.

3. Implementation

In diesem Kapitel wird für die Software ein Prototyp entworfen, der die Konzepte aus Kapitel 2 umsetzt und implementiert. Der Prototyp wird so entworfen, dass er nicht nur in einer Testumgebung sondern direkt auf dem Fahrzeug lauffähig ist. Er muss also den zusätzlichen Anforderungen eines Eingebetteten Systems entsprechen, dem im Gegensatz zur Laborumgebung wesentlich weniger Arbeitsspeicher und Rechenleistung zur Verfügung steht. Darüber hinaus wird das Debugging im Fehlerfall erschwert, so dass auf die Robustheit des Prototyps großen Wert gelegt werden muss. Das Grundgerüst für die Plattform bildet die Architektur, die von KORDIAN KUBAT in seiner Arbeit Kubat (2007) vorgestellt wird. Insbesondere der Entwurf für ein Reaktives System ist für diese Arbeit entscheidend (vgl. Kubat, 2007, S.13f). Für das Lane Inference System ist die von KORDIAN KUBAT eingeführte Komponente *Interpreter* von Bedeutung. Ein Interpreter hat die Aufgabe, Sensordaten zu interpretieren und zu verarbeiten. Anschließend wird das Ergebnis dem zentralen Container, dem Zustandsreflektor, wieder zur Verfügung gestellt. Dieser wird dann den nächsten Interpreter starten oder diejenige Fähigkeit, welche für die Sensordaten angemeldet ist, in diesem Fall die Fähigkeit der Fahrzeugsteuerung. Es liegen also der Zustandsreflektor als zentraler Container, der Interpreter für die Interpretation der Bilddaten (der Lane Interpreter) und die Fähigkeit der Fahrzeugsteuerung vor. Das Lane Inference System ist seinerseits ein Teil des Lane Interpreters und wird bei jedem Eingang eines Bildes aufgerufen.

Zusammengefasst ergibt sich folgendender Ablauf. Der Lane Interpreter meldet sich beim Zustandsreflektor zum Verarbeiten von Bilddaten an. Gleichzeitig stellt er die zuletzt gefundenen Koordinaten der Fahrbahnmarkierung zur Verfügung. Für diese meldet sich die Fahrzeugsteuerung ihrerseits beim Zustandsreflektor an. Kommt ein Bild der Kamera ins System, wird es vom Lane Interpreter verarbeitet, dieser wird das Lane Inference System auf Plausibilität der gefundenen Daten befragen und anschließend dem Zustandsreflektor die errechneten Koordinaten der Fahrspur mitteilen. Jetzt wird die Fahrzeugsteuerung gestartet, welche die soeben in den Zustandsreflektor eingegangenen Koordinaten der Fahrbahn verarbeitet.

Des weiteren wird das sich aus dem obigen Ablauf ergebende Zusammenspiel zwischen Lane Interpreter und Lane Inference System untersucht. Die Gesamtstruktur wird nicht weiter behandelt und ist kein Teil dieser Arbeit. Als weiterführende Literatur sei empfohlen die

Arbeit von HANNES REIMERS zur Fahrzeugsteuerung, Reimers (2008), sowie die Arbeit von KORDIAN KUBAT, Kubat (2007) zur Architektur.

3.1. Bibliotheken

Um bereits gelöste sprachliche sowie grafische Probleme nicht neu lösen zu müssen, wurde auf externe Bibliotheken zurückgegriffen. Für die Erweiterung der Sprache C++ wird das Framework Boost verwendet. Es stellt eine Reihe von Klassen und Methoden bereit, um ein breites Spektrum an Problemem komfortabel und gängig lösen zu können. Für diese Arbeit werden folgende Komponenten des Frameworks Boost benutzt:

- IOStreams
- Conversion
- Foreach

IOStreams stellen einen Satz an Klassen und Funktionen zur Verfügung, um C++ Streams zu erzeugen und den Zugriff einfach zu halten. Gleichzeitig können Filter mit Regulären Ausdrücken hinzugeschaltet werden. Ein Stream kann ebenfalls z. B. zip, gzip, bzip2, memory-mapped oder ein tcpstream sein. Für die TFALDALogging Klasse, u.a. den späteren Testtreiber, bedeutet das ein ungeheures Maß an Flexibilität, so kann das Logfile komprimiert oder unerwünschte Meldungen gefiltert werden. Conversion stellt Funktionen zur Umwandlung von Datentypen bereit, für diesen Prototypen wird die *String to Integer* Funktion verwendet. Foreach ermöglicht ein zusätzliches Konstrukt zur einfachen Iteration, welches bereits aus anderen Sprachen wie Java 1.5, Ruby oder Perl bekannt ist.

Als zweite Bibliothek wird OpenCV eingesetzt. Sie ermöglicht den Zugriff auf Bildmaterial für Echtzeitsysteme und dessen Verarbeitung. Für diese Arbeit werden folgende Komponenten benutzt:

- Die Sobel-Operation
- Datenstrukturen: Bild, Rechteck, Linie, Punkt
- Das Setzen von und Arbeiten mit ROIs
- Zeichen- und Anzeigeoperationen

3.1.1. Die Sobel-Operation *cvSobel*

Die Sobel-Operation aus OpenCV implementiert den Sobel-Operator, wobei durch Parameter entschieden werden kann, in welche Richtung gefaltet wird. In diesem Prototyp werden beide Gradientenbilder erzeugt, das heißt, dass sowohl in horizontaler Richung als auch in vertikaler Richtung gefaltet wird (vgl. Haberäcker, 1995, S.84). Über eine Managerklasse wird bestimmt, welche Faltungsmatrix verwendet wird. Die Entscheidung, welche der Faltungsmatrizen genommen wird, berechnet sich aus dem Winkel der letzten gefundenen Lane. Verläuft diese in einem Winkel von mehr als 45 $^{\circ}$ gegenüber dem Lot, so wird eine vertikale Faltungsmatrix ausgewählt, welche die horizontalen Kanten verstärkt, andernfalls wird als Standard die horizontale Faltungsmatrix angewendet. Durch diesen Manager wird vermieden, für jedes Bild zwei Sobel-Operationen durchführen zu müssen, um eine richtungsunabhängige Kantenverstärkung zu erhalten. Alternativ könnte der Laplace-Operator mit nur einer Matrix genommen werden, um immer ein richtungsunabhängiges Bild zu erhalten. Allerdings muss das Bild, bedingt durch die Implementation der Laplace-Operation in OpenCV, zusätzlich nach dem Aufruf der Laplace-Funktion von 16Bit wieder zurück in ein 8Bit-Bild konvertiert werden. Dies erfordert zuviel Rechenzeit, deshalb wurde die selektive Variante durch die Managerklasse mittels Sobel-Operator gewählt.

3.1.2. Das Setzen von und Arbeiten mit ROIs

Ein weiterer Vorteil von OpenCV ist die Möglichkeit, direkt ROIs auf einem Bild definieren zu können, so dass sämtliche Funktionen, u.a. die Sobel-Operation, nur auf dem definierten ROI arbeiten. Somit wird das Iterieren über Bildteile deutlich vereinfacht.

3.2. Der Lane Interpreter

Die Klasse LaneInterpreter implementiert alle notwendigen Schritte im Algorithmus des TFALDA. Das sind im Einzelnen zunächst die Bildvorverarbeitung mit Resampling und Kantenverstärkung durch den Sobel-Operator, anschließend der Automatic Lane Detector mit der Auswertung über die Distanzfunktion und die Verifikation der gefundenen Lanes durch Delegation an die Klasse LaneInferenceSystem. Die Abbildung 3.1 zeigt das entworfene Klassendiagramm zum Prototypen, anhand dessen dieser entwickelt wurde.

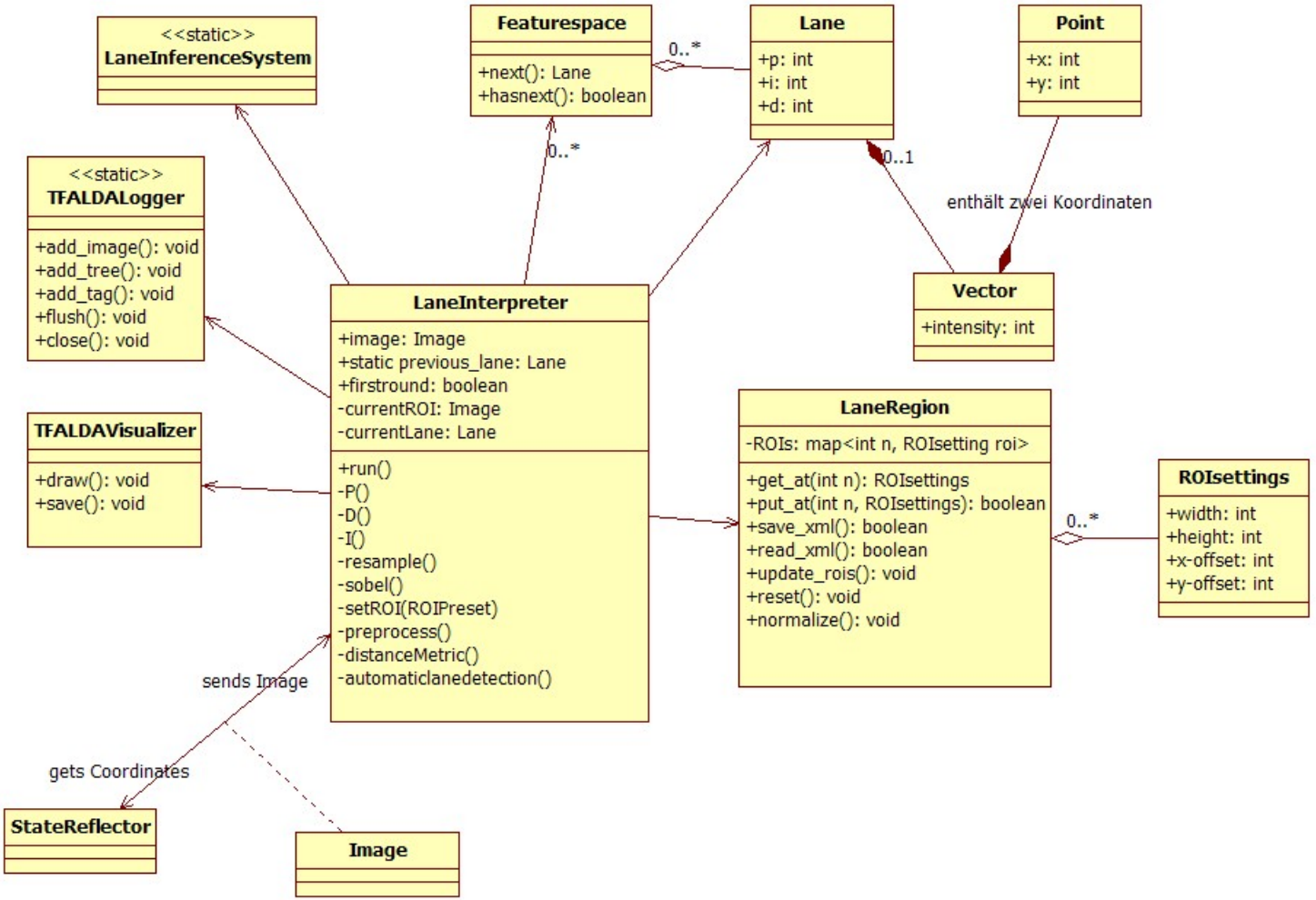

Abbildung 3.1.: Klassendiagramm des TFALDA

3.2.1. Preprocessing — Vorverarbeitung

Der Vorverarbeitungsschritt ist direkt in der Klasse LaneInterpreter als Methode implementiert und lässt sich wie folgt beschreiben.

- Setzen der ROI über `cvSetImageROI()`-Funktion
- Resampling mit Hilfe der `cvResize()`-Funktion
- Sobel-Operation über `cvSobel()`-Funktion, wobei der Sobel-Manager die Richtung des Sobel bestimmt

Der Quelltext zu den einzelnen Aufrufen kann im Anhang nachgeschlagen werden (vgl. Anhang A, S.51ff).

3.2.2. Automatic Lane Detection

Die Funktionsweise des Automatic Lane Detectors, wie sie von YOUNG UK YIM und SEQOUNG OH im Artikel Yim und Oh (2003) und von DENNIS BERGER in der Arbeit Berger (2008) beschrieben wird, kann dort im Detail nachgelesen werden.

3.2.3. Distanzfunktion

Die Distanzfunktion, also letztendlich das Entscheidungssystem des TFALDA, kann 1:1 aus dem Algorithmus in Software implementiert werden. Zu sehen in dem Listing 3.1 ist die Implementation als Methode der Klasse LaneInterpreter.

Listing 3.1: Distanzfunktion im Lane Interpreter

```
1
2  void LaneInterpreter::DistanceEvaluation
3    (const FeatureSpace& featurespace, const Candidate& ln_p,
4     Candidate& result )
5  {
6  float Kp=7.33f;
7  float Ki=1.0f;
8  float Kd=10.67f;
9
10 double lambda=987654321.0;
11 double max=100000000;
12 double term1,term2,term3;
13
14 ln_p.get_p();
15
16 int X_top_width[2];
17 int X_bottom_width[2];
18 static int Y_top_width[2];
19 static int Y_bottom_width[2];
20
21 BOOST_FOREACH(Candidate candidate, featurespace._candidates ){
22   term1 = Kp * abs(candidate.get_p() - ln_p.get_p());
23   term2 = Kd * abs(candidate.get_d() - ln_p.get_d());
24   term3 = Ki * (ln_p.get_i() - candidate.get_i());
25   lambda = term1 + term2 + term3;
26
27   if( lambda < max){
28     max = lambda;
29     result = candidate;
30     result.set_L(lambda);
31   }
32 }
```

3.3. Das Lane Inference System

Die Klasse LaneInferenceSystem stellt nach außen eine öffentliche Methode zur Verfügung, die die Korrektur der Lanes übernehmen soll. Folgender Methodenrumpf wird deklariert. Es werden also eine Liste der gefundenen Lanes sowie ein Intervall `dt` an die Methode übergeben. Auf `actual_lanes` wird innerhalb der Klasse LaneInferenceSystem schreibend zugegriffen, und es kann deshalb nicht als const deklariert werden. Es ist ja gerade erwünscht,

Listing 3.2: Methodenrumpf des Lane Inference Systems

```
1 void LaneInferenceSystem::infernewlane(TFALDARegionOfInterest&
    actual_lanes, const int dt);
```

dass das Lane Inference System eine Korrektur vornimmt, weshalb `actual_lanes` auf keinen Fall const sein darf.

3.3.1. Die Methode infernewlane()

Es sei folgende Reihenfolge der Algorithmen definiert.

1. Wende Tiefpassfilter über Fahrspurbreite an
2. Prüfe und korrigiere anhand der MinMax-Prüfung
3. Prüfe und korrigiere anhand der Fahrspurbreite
4. Prüfe und korrigiere anhand der vertikalen Abhängigkeit
5. Merke Fahrspurbreite

Jede einzelne Aufgabe kann ihrerseits in eine eigene Methode gekapselt werden, so dass sich für `infernewlane()` lediglich folgender Code in Listing 3.3 ergibt. Alle Aufrufe entsprechen exakt den Definitionen aus Kapitel 2. Lediglich eine Operation sticht durch eine Besonderheit heraus. Der Funktionsaufruf für das Tiefpassfilter `lowpass()` bekommt einen Parameter von oben aus dem Methodenkopf durchgeschleift, nämlich die Stellgröße Δt des Tiefpassfilters. Damit ist es möglich, bei unterschiedlicher Anzahl von Bildern pro Wegstrecke das Filter anzugleichen, wie es für Geschwindigkeitsänderungen notwendig ist. Für diesen Prototyp ist Δt für den ermittelten Maximalwert fest vorgegeben, und gilt auch für geringere Geschwindigkeiten. Der komplette Quelltext kann im Anhang nachgelesen werden (vgl. Anhang A.2, S.63f).

Listing 3.3: Die Methode infernewlane()

```
1  void infernewlane(RegionOfInterest& roi, const int dt){
2
3    //Über alle Lanes iterieren
4    for(int i=0; i< CI_ROIQUANTITY; i++){
5      //Tiefpassfilter über alle Lanes anwenden
6      lowpass(roi, dt, i);
7      //Korrektur auf Basis der Spurbreitenprüfung
8      check_and_corrent_minmax(roi.regions[i],i);
9      check_and_correct_lowpass(roi.regions[i],i);
10
11     //Fahrspur pro Seite vertikal ausrichten
12     if(i < CI_ROIQUANTITY -1)
13       for(int j=0;j< CI_BOXQUANTITY; j++)
14         check_and_correct_vertically(roi,i,j};
15
16     //Merke Y-1 für Tiefpassfilter
17     Y_1_roi_bottom_roadwidth[i] = Y_roi_bottom_roadwidth[i];
18     Y_1_roi_top_roadwidth[i] = Y_roi_top_roadwidth[i];
19   }
20 }
```

3.4. Die Lane Region

Die Klasse LaneRegion ist zuständig für die neue Positionierung der ROI sowie die Transformation des Koordinatensystems der ROI, und sie implementiert das entworfene Modell zur dynamischen Anpassung aus dem Kapitel 2.3.2. Gemäß Klassendiagramm ist die Lane Region mittels Instanzvariable mit dem Lane Interpreter gekoppelt.

3.4.1. Transformation des Koordinatensystems

Das Nutzen der von OpenCV bereitgestellten Funktionen zum Setzen der ROI hat den Nachteil, dass das Koordinatensystem innerhalb einer ROI wieder bei 0/0 beginnt. Daraus folgt, dass man bei vier verschiedenen ROIs vier Lanes erhält, welche sich auf vier verschiedene Koordinatensysteme beziehen. Um dieses Problem zu beheben muss die gefundene Lane normalisiert, das heißt auf das globale Koordinatensystem transformiert werden. Dies kann einfach korrigiert werden, da die Positionen der ROIs pro Durchlauf feststehen. Folgender Quelltext in Listing 3.4 erfüllt diese Aufgabe. Zeile 4 zeigt die Korrektur des Startpunkts p durch Addition der Lage der zugehörigen ROI. Ebenfalls wird der Punkt p_i sowie der Punkt

Listing 3.4: Die Methode adjust()

```
1  void normalize_vector(const int boundary, Candidate& result) const{
2    assert(boundary == 0 || boundary == 1);
3    LaneRegionBoundary roi = get_lane_at(boundary);
4    result.set_p(result.get_p() + roi.get_x());
5    result.set_d(result.get_d();
6    result.set_pi(cvPoint(result.get_p(),
7      result.get_pi().y  + get_offset()));
8    result.set_qk(cvPoint(result.get_qk().x + roi.get_x() ,
9      result.get_pi().y  + get_height()));
10 }
```

q_k in Zeile 6 und 8 angepasst. Damit sind die Positionen der gefundenen Lanes im globalen Koordinatensystem bestimmt. Dies ist jedoch nicht das einzige Problem, da die Lanes nicht innerhalb der gesetzten ROIs gesucht werden. Es liegt, wie bereits gezeigt, noch ein Schritt dazwischen, nämlich das Resampling. Die ROI ist um den Resamplefaktor verkleinert worden, was nun herausgerechnet werden muss. Somit wird die Funktion ergänzt mit der Multiplikation durch den Resamplefaktor. Damit liegt nun eine vollständige Transformation auf das globale Koordinatensystem vor.

3.4.2. Dynamische Positionierung

Die Implementation folgt direkt aus dem Aktivitätsdiagramm in Abbildung 2.7. Die Aktion *Hole ROIpositionen* kann sehr einfach über lokale Variablen und getter-Methoden erfolgen. Die Aktion *Prüfe Fahrzustand* wird in diesem Prototypen über folgende Abfragen implementiert.

Dieser Quelltext weicht leicht von dem Entwurf bzw. dem Aktivitätsdiagramm ab und implementiert zwei Aktionen auf einmal. Einerseits wird auf Kurvenfahrt geprüft, andererseits gleich auf eine Überschneidung der ROIs. Für den Prototypen ist das nicht von Bedeutung, da sich an den ausgeführten Aktionen nichts ändert. Jedoch ist der Quelltext nicht gut lesbar. Für eine endgültige Version sei eine Implementation mit Hilfe der Fahrzustände, welche in eigene Klassen gekapselt werden können, empfohlen, um den Quelltext verständlich und wartbar zu halten.

Listing 3.5: Prüfe Fahrzustand

```
1  //linkskurve
2  if((l_width + l_x > r_x) && r_x < 325){
3  //löse Überdeckung links auf
4  ...
5  }
6  //rechtskurve
7  if((l_width + l_x > r_x) && (l_width + l_x) > 425){
8  //löse Überdeckung rechts auf
9  ...
10 }
11 //tue nichts
```

4. Test

Dem prototypisch entworfenen Lane Interpreter wird eine Sequenz von Testbildern eingespielt, mittels TFALDA findet der Interpreter darin Lanes und delegiert sie an das Lane Inference System. In diesem Kapitel wird untersucht, inwieweit das Lane Inference System Einfluss auf die gefundenen Lanes nimmt, und mit welcher Wahrscheinlichkeit das Gesamtsystem die Fahrbahnmarkierung erkannt hat. Dazu wird ein Testsystem entworfen, dass diese Analyse übernimmt und sowohl numerisch als auch grafisch in einem Diagramm darstellt, mit welcher Wahrscheinlichkeit die gefundenen Lanes korrekt waren. Für den Entwurf werden Testdaten sowie deren Sollwerte benötigt, aus denen dann ein Blackboxtest entwickelt wird. Zusätzlich zu dem Blackboxtest wird ein Visualisierungsmodul, der TFALDAVisualizer, geschrieben, welcher die gefundenen Lanes in die eingespielten Bilder einzeichnet und wieder ausgibt. So wird ein Film erzeugt, in dem live mitverfolgt werden kann, wie der TFALDA mit dem Lane Inference System arbeitet.

4.1. Testaufbau

Zur Verfügung steht ein Modellbaufahrzeug im Format 1:10. Auf diesem sind die für diese Arbeit wichtigen Teile, die Kamera sowie der PC im miniITX Format, montiert. Die Teststrecke erfüllt die Wettkampfbedingungen des Carolo-Cup 2008 (vgl. Braunschweig, 2008, S.6f).

4.1.1. Spezifizieren der Testfälle

Es werden die Testfälle Geradeausfahrt, Kurvenfahrt und Streckenfahrt festgelegt und wie folgt spezifiziert.

Folgende Merkmale bestimmen eine Geradeausfahrt:

- Das Fahrzeug steht bei Testbeginn auf der rechten Fahrbahnseite in Ruhe
- Das Fahrzeug steht bei Testbeginn parallel zur Fahrbahnmarkierung
- Es liegen keine Störungen durch Lichteinfall oder auf der Fahrbahn liegende Gegenstände vor

- Die Maximalgeschwindigkeit wird nicht überschritten
- Die abzufahrende Strecke führt durchgängig geradeaus

Folgende Merkmale bestimmen eine Kurvenfahrt:

- Es gelten die ersten vier Punkte der Geradeausfahrt
- Das Fahrzeug befindet sich bei Testbeginn mindestens anderthalb Meter vor der Kurve auf einer Geraden
- Das Fahrzeug fährt eine Kurvenart vollständig ab (keine S-Kurve)
- Nach Abfahren der Kurve hält das Fahrzeug auf der Geraden wieder an

Folgende Merkmale bestimmen eine Streckenfahrt:

- Es gelten die ersten vier Punkte der Geradeausfahrt
- Eine bestimmte Strecke wird komplett, jedoch nur einmal, abgefahren
- Das Fahrzeug startet und hält an festgelegten Punkten

Anhand dieser Spezifikation werden Testbilder aufgezeichnet und Sollwerte bestimmt.

4.1.2. Aufzeichnen der Testbilder

Zum Sammeln der Testbilder wird diejenige Kamera verwendet, die auch während der selbstständigen Fahrt auf dem Testfahrzeug angebracht ist. Sie wird in der korrekten Höhe montiert und ausgerichtet. Es wird ein Programm prototypisch entworfen, welches in einem definierten Zeitintervall Bilder aufnimmt und diese sequenziell abspeichert. Die einzelnen Bilder werden über eine fortlaufende Nummer im Dateinamen gekennzeichnet. Das Programm wird auf ein Notebook gespielt, und dieses mit der auf dem Fahrzeug montierten Kamera per USB verbunden. Anschließend kann das Fahrzeug per Fernsteuerung bedient oder von Hand über die Strecke geschoben werden, während auf dem Notebook das Aufzeichnungsprogramm läuft. Da das Programm etwa drei Bilder pro Sekunde aufzeichnet, muss das Fahrzeug entsprechend langsam geschoben werden. Auf diese Weise werden von der Teststrecke drei Bildreihen aufgenommen und abgespeichert. Damit stehen dem Testsystem verschiedene Bildreihen zur Verfügung, welche den zuvor spezifizierten Testfällen genügen.

4.1.3. Bestimmen der Sollwerte

Für einen Blackboxtest werden Sollwerte benötigt, die pro Bild die korrekte Fahrbahn markieren. Als Sollwert wird folgendes festgelegt. Der Sollwert für ein Bild der Sequenznummer X sei als Bild definiert, welches an den Stellen, an denen sich eine Fahrbahnmarkierung befindet, eine definierte Markierungsfarbe rot aufweist. Unterbrochene Fahrbahnmarkierungen, zum Beispiel der Mittelstreifen, sind als durchgängig zu kennzeichnen. Abbildung 4.1 zeigt ein solches Referenzbild. Für eine Bildsequenz werden zwischen 80 und 190 Bilder bearbeitet, pro Bild wird dafür etwa eine Minute benötigt.

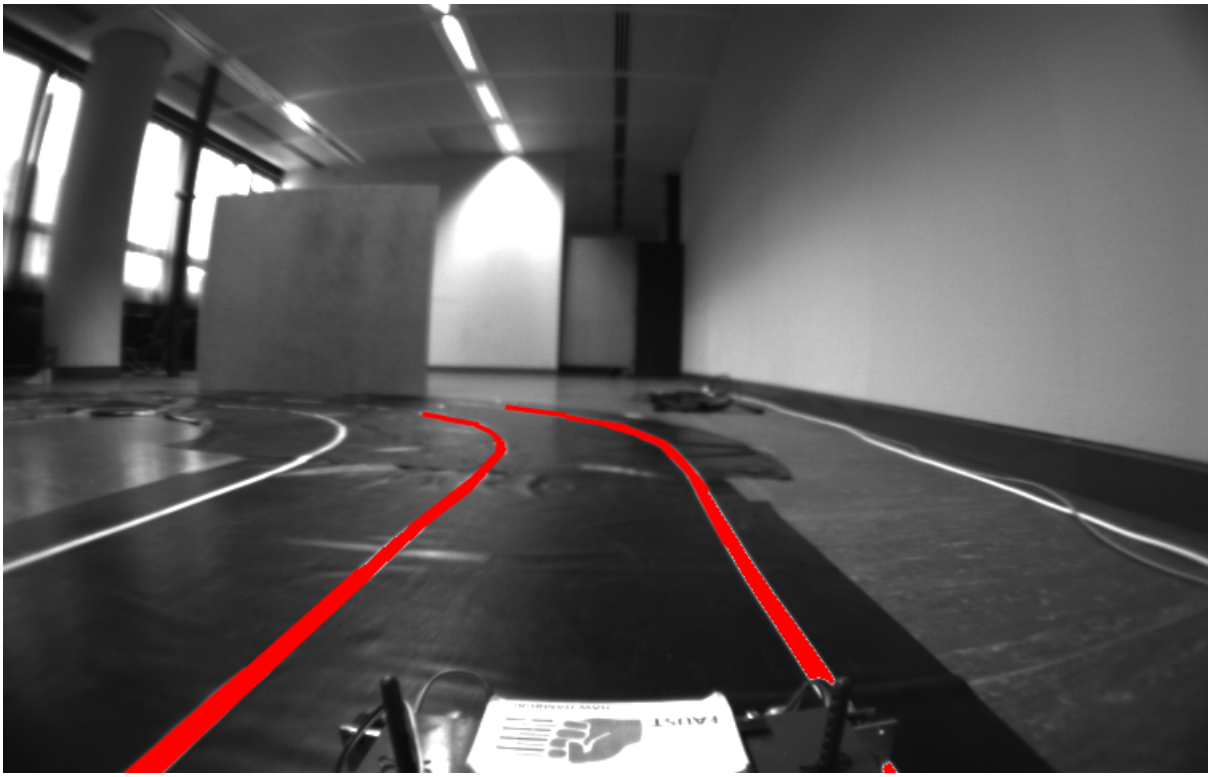

Abbildung 4.1.: Sollwertbild einer Kurvenfahrt

4.1.4. Das Testobjekt

Das Testobjekt selbst sind die gefundenen Fahrbahnmarkierungen des TFALDA, welche auf ihre Gültigkeit hin überprüft werden müssen. Zu diesem Zweck wird das Testobjekt über einen sogenannten Testtreiber angesprochen, dieser hat die Aufgabe, zu einer gegebenen Sequenznummer die gefundenen Lanes des TFALDA herauszusuchen und nach außen an

das Testsystem zu liefern. Die Sequenznummer entspricht dabei der Nummer der Bildes in der Sequenz, die vorher dem TFALDA über den Test Image Generator eingespielt wurde. Der Test Image Generator ist eine Klasse, die die Aufgabe hat, dem System statt der Kamerabilder eine Sequenz an vorher aufgezeichneten Bildern zur Verfügung zu stellen, ohne dass das System dabei einen Unterschied feststellen kann. Das Testobjekt wird somit hier nicht als Instanz während der Laufzeit angesprochen, sondern vielmehr als zustandsloses, serialisiertes Objekt betrachtet. Der Testtreiber übernimmt lediglich den Export des Testobjektes aus der Laufzeitumgebung und den Import in die Testumgebung.

4.1.5. Der Testtreiber im Detail

Der Testtreiber besteht aus zwei Teilen, dem Exportteil und dem Importteil. An den Exportteil werden folgende Anforderungen gestellt.

Das Export Modul

- enthält pro Bild die Sequenznummer, die Lanes mit ihrer Lage, Richtung sowie das Lambda der Distanzfunktion und einen Zeitstempel.
- speichert die Daten im XML-Format.
- ist ein eigenständiges, unabhängiges Modul.
- sollte den existierenden Code des Laufzeitsystems nicht unleserlich machen.
- sollte den Einfluss auf das Laufzeitsystem so gering wie möglich halten.

Dies deckt sich mit Anforderungen aus der Aspektorientierten Programmierung. Inbesondere der Begriff des Cross-Cutting Concern spielt hier eine Rolle. Dabei handelt es sich um einen *Belang*, der quer durch den Code verstreut ist. Als Belang kann hier das Exportieren der Fahrbahnmarkierung verstanden werden, welches in verschiedenen Modulen der Software geschieht und nicht durch herkömmliche Verfahren der Objektorientierung gekapselt werden kann. Dies entspricht einem Logging-Aspekt. Also wird für den Entwurf des Exportteils des Testtreibers ein Aspekt Exporter definiert. Dieser beinhaltet Methoden für den Dateizugriff sowie einfache Methoden für die Erstellung eines XML-Dokuments. Sobald die Lanes durch das Lane Inference System geprüft und korrigiert wurden, müssen sie nun als XML exportiert werden. Dafür wird ein Pointcut definiert, der im Aspekt Exporter das Protokollieren der Lanes an dieser Stelle übernimmt. Ein weiter Pointcut wird definiert, nachdem ein Bild vom TFALDA komplett verarbeitet wurde. Dieser Pointcut wird sowohl den exakten Zeitstempel aufnehmen als auch `flush()` auf das Dateisystem ausführen, so dass alle bisherigen Daten definitiv weggeschrieben werden. Ein letzter Pointcut ist am Ende der `main()` Methode nötig, um das XML-Dokument mit einem schließenden Tag versehen zu können. Eine Implementation mittels AspectC++ ist nicht Teil dieser Arbeit, sei aber empfohlen.

Die gefundenen Lanes liegen nach dem Export als XML serialisiert vor und können vom Importteil des Testtreibers der Testumgebung zur Verfügung gestellt werden. Durch den im JDK 1.6 bereits implementierten JAXB 2.0 Standard können Objekte aus XML komfortabel serialisiert werden. Der Testtreiber stellt Methoden zur Verfügung, um auf diese Objekte mittels der Sequenznummer zugreifen zu können. Damit ist eine Trennung zwischen Testobjekt und Testtreiber sauber vollzogen. Es ist dadurch ebenfalls verhindert, dass die Testumgebung ungewollt Einfluss auf den TFALDA nimmt, da die Fahrspurerkennung bereits abgeschlossen ist und das Testen ausschließlich auf den aufgezeichneten Daten erfolgt.

Listing 4.1: Serialisierte Lanes

```
1 <root>
2 <image seq ="11" time="1211011323" >
3 <line>
4 <point>142,425</point>
5 <point>238,340</point>
6 <direction>-96</direction>
7 <lambda>-668</lambda>
8 </line>
9 ...
```

4.1.6. Der Testrahmen

Nach ANDREAS SPILLNER U.A. sei ein Testrahmen eine Sammlung aller Programme (u.a. Testtreiber, Testobjekte), die notwendig sind, um Testfälle auszuführen, auszuwerten und Testprotokolle aufzuzeichnen. Für den Test des TFALDA sowie des Lane Inference Systems wird der Testrahmen zusätzlich eingegrenzt und wie folgt spezifiziert. Der Testrahmen ist die Sammlung an Programmen, welche die erkannten Fahrspurmarkierungen, also das Testobjekt selbst, mittels Testfällen auf ihre Korrektheit überprüft. Der dabei verwendete Blackboxtest befragt ein Testorakel, welches auf definierte Sollwerte zurückgreift. Die gewonnenen Resultate sollen sowohl grafisch als auch numerisch vorliegen. Es wird eine Software entworfen, die die beschriebenen Anforderung erfüllen kann. Der Klassenentwurf sowie die Implementation können bei Interesse im Anhang nachgelesen werden.

4.1.7. Der Testlauf

Ein Testlauf sieht wie folgt aus. Für jedes Bild der Testreihe werden sequenziell die Lanes des Bildes mit den rot markierten Stellen des Referenzbildes verglichen. Die Vergleichsoperation

wird von dem Blackboxtest übernommen. Dieser stellt zu Beginn eine Summe von Pixeln zusammen, die sich auf der Lane befinden. Im mathematischen Sinne ist dies eine diskrete Approximation. Als Implementation wird eine optimierte Version des *Bresenham Algorithmus* gewählt, der von HERANS & BAKER in ihrer Arbeit Baker (2003) empfohlen wird. Mit der nach Durchlaufen des Algorithmus erhaltenen Pixelmenge wird gepfrüft, ob die Pixel, die sich auf der vom TFALDA gefundenen Lane befinden, einen rot markierten Pixel des Referenzbildes an der gleichen Stelle getroffen haben. Dabei wird eine Abweichung von $\epsilon = 5$ Pixel zugelassen. Die Anzahl der getroffenen Pixel wird in ein Verhältnis mit der gesamten Pixelmenge der Lane gesetzt, diese Wahrscheinlichkeit wird zurückgegeben und anschließend protokolliert.

4.2. Ergebnisse

Um den Einfluss des Lane Inference Systems auf die Fahrspurerkennung zu untersuchen, wird der Testlauf auf dieselbe Bildsequenz zweimal angewendet, bei dem einen wird das Lane Inference System dabei aktiviert, bei dem anderen nicht. Mit Hilfe der Testumgebung werden Ergebnisse und Verbesserungen durch das Lane Inference System gezeigt. Im Diagramm ist dies durch *mit LIS alles aktiviert* für den aktuellen Softwarestand mit allen entworfenen Verbesserungen, sowie durch *ohne LIS Stand Studienarbeit* für den Softwarestand nach der Studienarbeit Berger (2008) gekennzeichnet. Als Testlauf dienen die in Kapitel 4.1 gestellten Anforderung für Geradeaus-, Kurven- und Streckenfahrten.

4.2.1. Geradeausfahrt

Für die Geradeausfahrt wurden 80 Bilder aufgezeichnet und dem TFALDA mittels Test Image Generator eingespielt. In Abbildung 4.2 wird die Wahrscheinlichkeit, mit der pro Bild die vier Lanes eine gültige Fahrbahnmarkierung getroffen haben, gezeigt. Die Linien beider Testläufe verlaufen ähnlich, wobei zu sehen ist, dass das Lane Inference System eine leicht stabilisierende Wirkung hat. Die einzelnen nach unten gerichteten Spitzen in der blauen Linie sind die Phasen, in denen die ROI unten links keine weiße Markierung des Mittelstreifens enthält. Somit wird der TFALDA Algorithmus, im Hinblick auf die gesamte Fahrspur, keine gute neue Lane finden können. Das Lane Inference System kann diesen Missstand korrigieren, indem es die vorher gefundene bessere Lane verwendet. Dies ist jedoch nur dann möglich, wenn die vom TFALDA gefundene Lane schlecht genug war, so dass die Entscheidungssysteme der einzelnen Module des Lane Inference Systems greifen. Aus diesem Grund wird manchmal eine nicht völlig korrekte Lane nicht als solche erkannt. Somit verläuft selbst bei einer Geradeausfahrt die rote Linie nicht durchgängig bei 100 %.

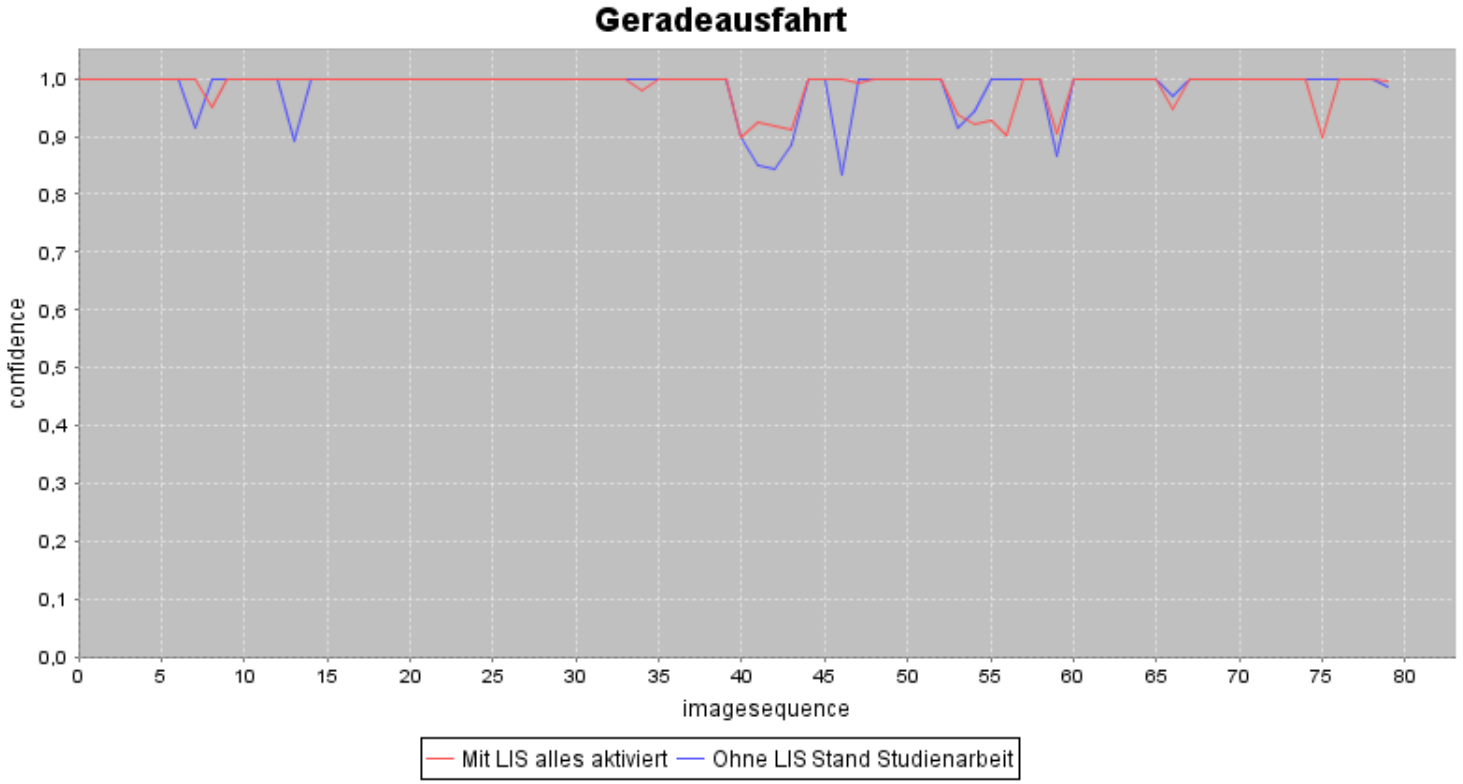

Abbildung 4.2.: Testfahrt Geradeaus

4.2.2. Kurvenfahrt

Für die Kurvenfahrt, es wurde exemplarisch eine Linkskurve gewählt, wurden 80 Bilder aufgezeichnet und dem TFALDA wie bei der Geradeausfahrt eingespielt. Es wird eine Untersequenz einer größeren Streckenfahrt verwendet, deshalb beginnt die Kurvenfahrt mit Sequenznummer 80 und endet mit Sequenznummer 160. Die Achsen des Diagramms beschreiben auch hier die Wahrscheinlichkeit, mit der die Lanes eine gültige Fahrbahnmarkierungen getroffen haben, zu sehen in Abbildung 4.3. Der Testlauf ohne Lane Inference System, repräsentiert durch die blaue Linie, zeigt während der Fahrt eine zunehmend geringere Wahrscheinlichkeit der Erkennung der Fahrbahnmarkierung, die gegen Ende der Kurvenfahrt nur noch zu ca 75% erkannt wird. Zur Anylyse dieses Phänomens werden die gefundenen Lanes in die Bildsequenz eingezeichnet. Dies übernimmt der TFALDA Visualizer, der im Rahmen dieser Arbeit zur Visualisierung der Fahrspurerkennung entwickelt wurde. Eine Betrachtung dieser Ergebnisse liefert die Erklärung. Sie zeigen, dass die ROI oben links innerhalb der Kurve bereits ab Bild 133 vom Mittelstreifen auf die äußere Fahrspurmarkierung der Gegenfahrbahn springt und dort bis zuletzt verweilt. Eine Weiterfahrt wäre nun nicht mehr möglich.

Die rote Linie, also der Testlauf mit aktiviertem Lane Inference System, zeigt eine insgesamt höhere Wahrscheinlichkeit eine korrekte Lane zu finden, und vor allem werden gegen Ende der Kurve alle Fahrbahnmarkierungen fast hundertprozentig richtig erkannt. Für den Teil der Sequenz von Bild 88 bis 95, wo die Linie an einem Punkt sogar unter 50% rutscht, besteht jedoch Klärungsbedarf. Die Analyse dieser Bilddaten hat folgendes Ergebnis zu Tage gebracht. Die oberen ROIs geben bereits die Richtung der Kurve gut vor, treffen jedoch die Fahrbahnmarkierung nicht exakt. Dies liegt unter anderem daran, dass eine Lane durch eine Strecke definiert ist, die Fahrbahnmarkierung in der Kurve jedoch ist gekrümmt. Somit versucht das System, eine Kurve durch eine Gerade darzustellen und trifft ähnlich einer Tangente nur einen Teil der Kurve. Dies wird besonders an der Abbildung 4.4 deutlich. Dort ist Sequenzbild 89 als Einzelbild mit eingezeichneten Lanes zu sehen. Dass die Wahrscheinlichkeit der Fahrspurerkennung in den nächsten folgenden Bildern noch weiter abnimmt, liegt an der Tatsache, dass die oberen ROIs die Richtung bestimmen und, gemäß Definition der vertikalen Korrektur, die unteren an diese angeglichen werden, so dass die untere linke Lane so korrigiert wird, dass sie dem korrekten Verlauf der Fahrbahn nicht mehr entspricht. Die Abbildungen 4.4 und 4.5 zeigen dieses Phänomen in den Bildern 89 und 93.

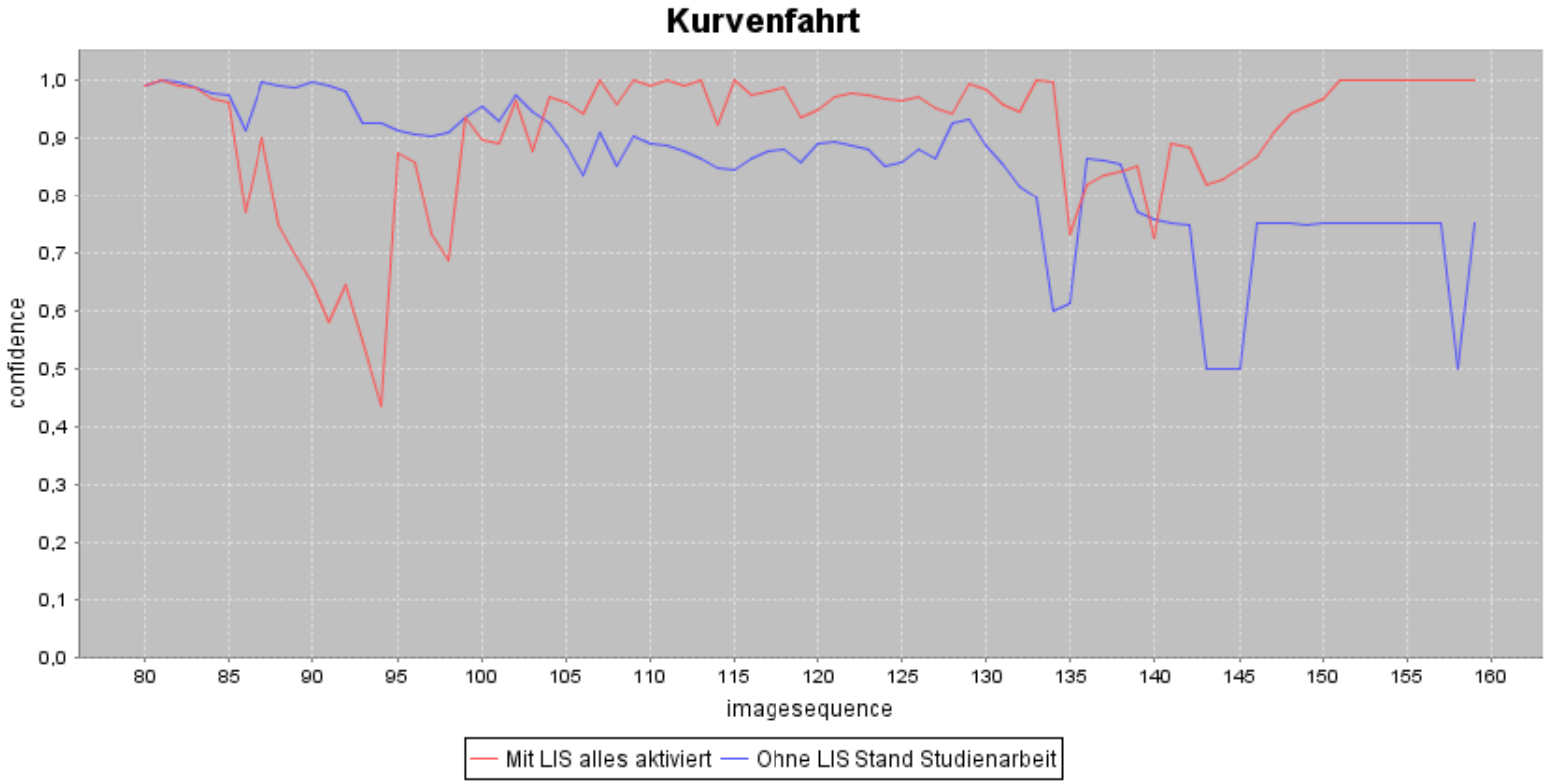

Abbildung 4.3.: Testfahrt Kurve

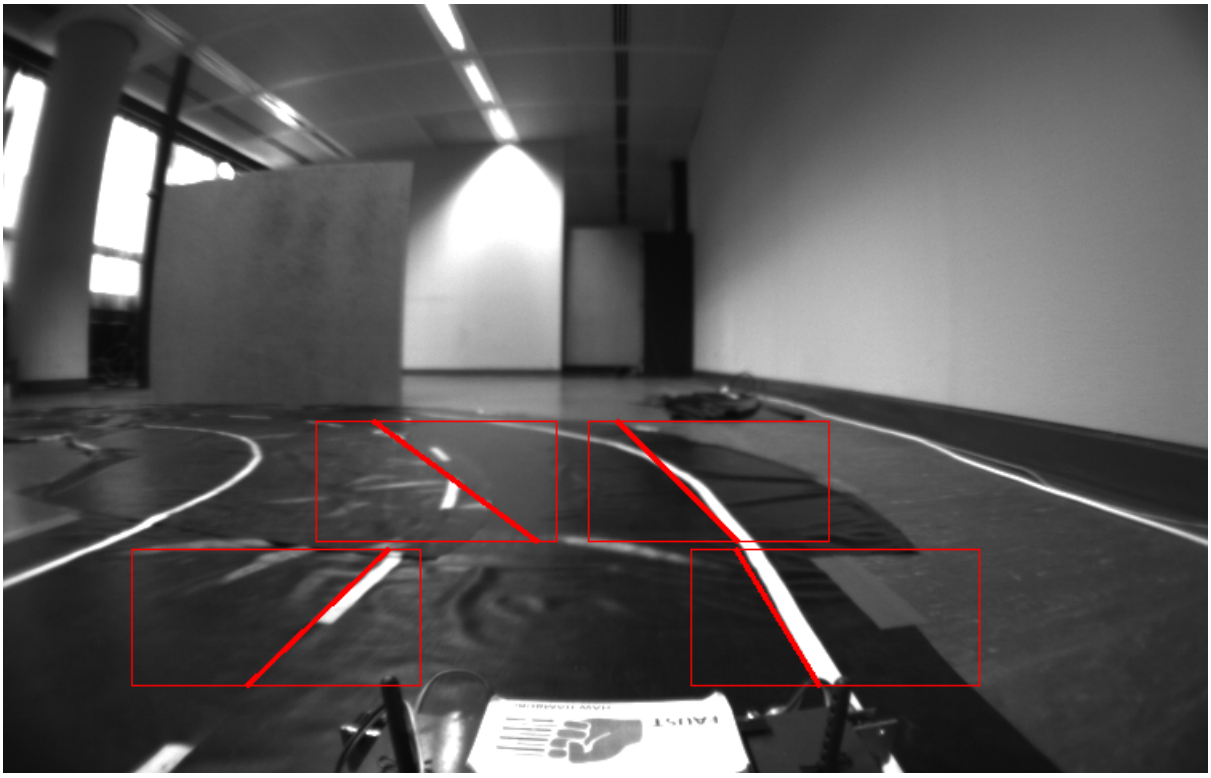

Abbildung 4.4.: Kurvenfahrt — Bild 89

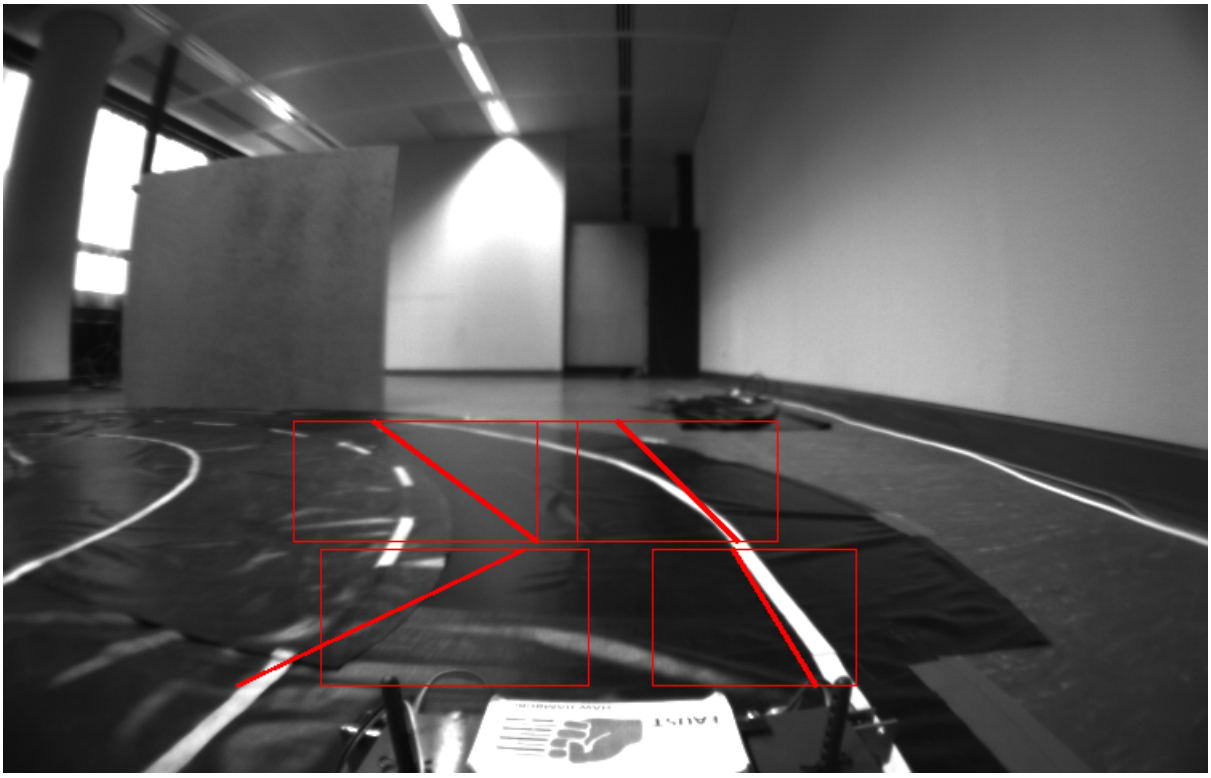

Abbildung 4.5.: Kurvenfahrt — Bild 93

4.2.3. Streckenfahrt

Für die Streckenfahrt wurden 183 Bilder aufgezeichnet und dem TFALDA eingespielt. Abbildung 4.6 zeigt die Wahrscheinlichkeit der Fahrspurerkennung für eine Fahrt auf einer geraden Strecke, die dann durch eine Linkskurve wieder auf eine gerade Strecke führt. Die Bildsequenz 0 bis 80 zeigt den Verlauf der Geradeausfahrt, 81 bis 160 zeigt einen Kurvenverlauf, den Abschluss bildet Sequenz 161 bis 180 mit einer weiteren Geradeausfahrt. Deutlich ist zu sehen, dass der Algorithmus ohne das Lane Inference System zu Beginn der Kurve Probleme bei der Erkennung der Fahrspurmarkierung hat. Die Wahrscheinlichkeit liegt bei 75%, was in diesem Fall bedeutet, dass eine von vier Lanes die Fahrbahnmarkierung nicht trifft, was durch die Auswertung des Bildmaterials bestätigt wurde. Hier ist zu sehen, dass die Fault Lane sich innerhalb der ROI oben links befindet. Dort ist die Lane weiterhin nach rechts geneigt und zeigt immer noch eine Geradeausfahrt an, obgleich sich das Fahrzeug in der Linkskurve befindet. Mit dem Lane Inference System hingegen wird die Fahrspur korrekt erkannt. Im Bereich der Bildsequenz 130-150 zeigt der Blackboxtest zwar ein Absinken der Erkennungswahrscheinlichkeit an, das aber auf eine Besonderheit des Systems des TFALDA zurückzuführen ist. Nämlich darauf, dass eine Lane, die durch eine Gerade dargestellt wird, in einer Kurve nur einen Teil der Pixel der tatsächlichen Fahrspur, die ebenfalls stark gekrümmt ist, überdecken kann. Siehe dazu Abbildung 4.4. Der Schlussverlauf zeigt deutlich, dass der Algorithmus ohne Hilfe des Lane Inference Systems die Fahrspur durchgehend zu unter 75% erkennt. Dies deckt sich mit den Auswertungen des Bildmaterials, denn die ROI oben links erkennt die Spur der Gegenfahrbahn und bleibt auf dieser bis zum Schluss. Mit dem Lane Inference System wird dieser Fehler durch eine Überprüfung der Fahrspurbreite vermieden, nach Herausfahren aus der Kurve wird die Fahrbahn zu 100% korrekt erkannt.

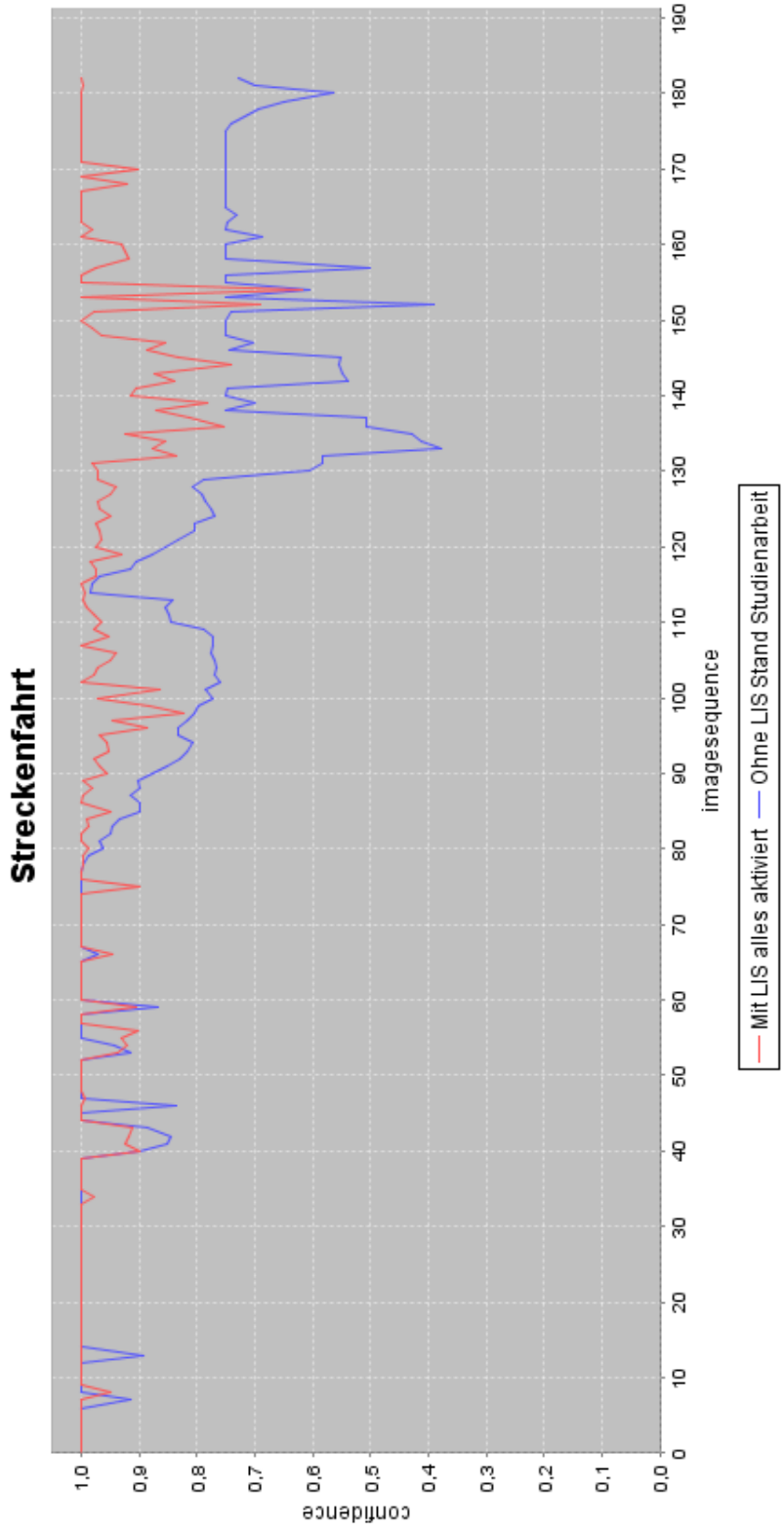

Abbildung 4.6.: Testfahrt Strecke

5. Fazit

Im Rahmen dieser Arbeit wurde für den TFALDA ein Lane Inference System entwickelt. Es soll als letztes Korrektursystem die Plausibilität der gefundenen Fahrspur überprüfen und ggf. korrigieren. Dafür sind drei Schritte vorgesehen, erstens die permanente Messung der Fahrbahnbreite mit Hilfe eines Tiefpassfilters, zweitens die Anwendung eines Prüfsystems für die vertikale Abhängigkeit der Lanes untereinander pro Seite. Die jeweils linken und rechten Lanes werden miteinander verbunden, so dass eine zusammenhängende Fahrbahnmarkierung entsteht. Als dritter Schritt sorgt ein Modul für die dynamische Positionierung der ROIs innerhalb des Bildes. Dies geschieht zum einen auf Basis der zuletzt gefunden Lanes sowie einer zustandsbasierten Komponente, so dass bei einer Kurvenfahrt das *in die Kurve Blicken* eines Menschen bei einer Autofahrt simuliert wird. Ein Resetsystem bringt, falls erforderlich, die ROIs wieder in ihre Ursprungsstellung.
Der Entwurf dieses Systems wurde in Kapitel 2 vorgestellt und im Kapitel 3 prototypisch in C++ implementiert.

Um Erkenntnis über die Effektivität des Lane Inference Systems zu erhalten, wurde im Kapitel 4 ein Testsystem entwickelt. Dieses ist in die vier Komponenten Testrahmen, Testfall (Blackboxtest), Testtreiber und das eigentliche Testobjekt unterteilt. Die äußere Testumgebung, auch Testrahmen genannt, startet die Testfälle, protokolliert die Ergebnisse und liefert sie visuell und nummerisch an den Benutzer zurück. Die Testfälle werden von einem Blackboxtest durchgeführt, welcher die Lanes mit den Sollwerten vergleicht, die im Voraus für jedes Bild einzeln ermittelt und eingetragen wurden. Im Kapitel 4.1.3 wird detailliert erläutert, wie dieser Prozess abläuft. Das eigentliche Testobjekt, nämlich die gefundenen Lanes, werden vom Testtreiber zur Verfügung gestellt. Dieser besteht aus einem Export- und einem Importteil, welche jeweils für das Exportieren der Lanes aus der Laufzeitumgebung, sowie für das Importieren in die Testumgebung zuständig sind.

In Kapitel 4.2 sind die durchgeführten Tests beschrieben, welche vor allem im Kurvenbereich und der gesamten Streckenfahrt eine Verbesserung der Fahrspurerkennung zeigen.

5.1. Kritik

In Kapitel 4.2 konnte dargestellt werden, dass die in Kapitel 2 gestellten Anforderungen erfüllt werden konnten, was durch die prototypische Implementation gezeigt wird. Das Lane Inference System beinhaltet Informationen über das Aussehen einer Fahrbahn und kann prüfen, ob die aktuellen Lanes insgesamt zu einer gültigen Fahrbahn führen. Gleichwohl fallen zwei Schwächen auf.

Erstens hat das vertikale Prüf- und Korrektursystem aus Kapitel 2.2 eine überschreibende Eigenschaft. Das bedeutet, dass falls eine Lane zu einer ungültigen Fahrbahn führt und somit als Fault Lane erkannt wird, so wird die Lane des Vorbildes als besser bewertet und wieder verwendet. Die aktuelle Lane wird in diesem Fall vom Lane Inference System komplett überschrieben. Dies kann bedeuten, dass eine Lane als schlechter bewertet wird als sie eigentlich ist, und dieses Korrektursystem so zu einer erkannten Fahrbahn führt, die nicht der Realität entspricht.
Besser wäre ein gewichteter Ansatz, der die vertikale Korrektur lediglich zu einem gewissen Teil, steuerbar über einen Faktor, Einfluss auf die aktuelle Lane nehmen lässt. Ein gewichteter, nicht überschreibender Lösungsansatz würde z. B. den Bezug zu einer gültigen Fahrbahn mit in die Distanzfunktion des TFALDA und damit in die gewichtete Berechnung einfließen lassen. Zur Erinnerung: Lambda ist das Ergebnis der Distanzfunktion des Automatic Lane Detector, welcher aus den Lane Candidates die Lane mit dem geringsten Lambda auswählt (Berger, 2008, S.8f; Yim und Oh, 2003, S.221).

λ berechnet sich folgendermaßen,

$$\lambda_i = K_P|P(C_i) - P(L_{n-1})| + K_D|D(C_i) - D(L_{n-1})| + K_I(I(L_{n-1}) - C_i) \tag{5.1}$$

wobei

P der Startpunkt, D die Richtung und I die Intensität der Lane Candidates C_i angibt. Da der TFALDA immer jeweils nur auf einer ROI arbeitet, müsste sich die Sicht des TFALDAs erweitern um die vertikale Korrektur in die Distanzfunktion einfließen zu lassen. Dabei stellt sich die Frage, ob der Versuch den TFALDA in dieses globale Modell zu zwängen noch sinnvoll ist, oder ob nicht ein alternatives Modell dann den Vorzug erhält. Das Problem der vertikalen Abhängigkeit wird u.a. im Kapitel 4.2.2 diskutiert und ist in Abbildung 4.5 gut zu sehen.

Dies führt direkt zur zweiten Schwäche des TFALDA, dieser ist nur dann stabil und stark, wenn die ROIs genügend groß sind. Das hängt damit zusammen, dass der Automatic Lane Detector genügend Pixel in der Höhe einer ROI braucht, um zuverlässig eine Lane darin erkennen zu können. In einer separaten Arbeit müsste der Zusammenhang zwischen Größe

der ROIs und Zuverlässigkeit des Automatic Lane Detector untersucht werden. Eine empirische Messung im Rahmen dieser Arbeit zeigte, dass eine Halbierung der Höhe der ROI dazu führt, dass das Fahrzeug im laufenden Betrieb bei der ersten Kurve die Fahrbahn verlässt. Gleichwohl lässt sich eine stark gekrümmte Kurve, wie sie durch die Spezifikation der verwendeten Teststrecke möglich ist, durch nur zwei ROIs schlecht approximieren und führt unweigerlich zu einem unbefriedigenden Ergebnis, wie in Abbildung 4.4 deutlich zu erkennen ist. Ebenfalls ist es problematisch, wenn, wie bei dem Mittelstreifen, Unterbrechungen in der Fahrbahnmarkierung vorliegen, und der TFALDA innerhalb einer kleinen ROI über mehrere Bilder einer Sequenz hinweg keine Fahrbahnmarkierung erkennt.

5.2. Ausblick

Die vertikale Prüfung zeigt noch zusätzliche Vorteile bei Überholmanövern oder generell Spurwechseln. In Abbildung 5.1 ist ein Spurwechsel zu sehen, der mit eingeschaltetem Lane Inference System durchgeführt wurde. Ohne zusätzlichen Aufwand könnten Spurwechsel möglich werden. Eine separate Untersuchung und Auswertungen mit Hilfe des Testsystems müssten noch durchgeführt werden, und sind für ein aussagekräftiges Ergebnis notwendig. Eventuell sollte ein Fahrzustand für Spurwechsel in die Klasse LaneRegion eingefügt werden, welche das Positionieren der ROIs übernimmt.

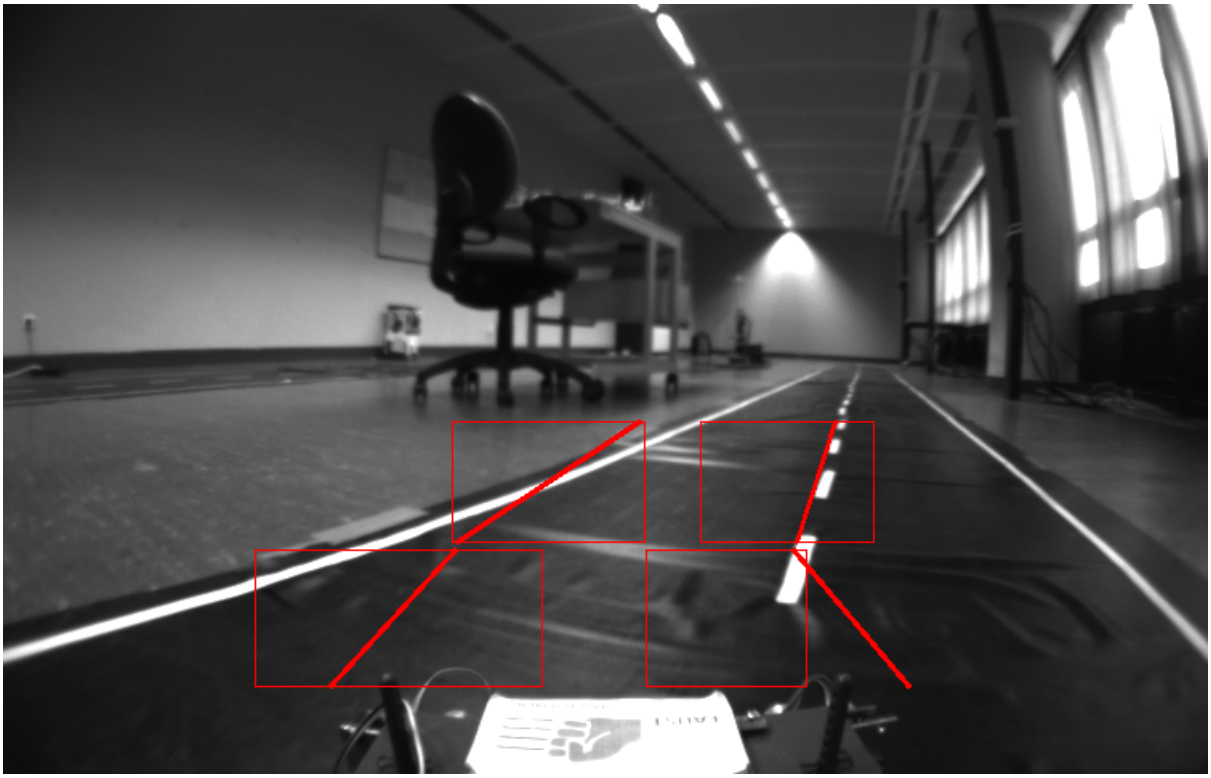

Abbildung 5.1.: Fahrspurwechsel auf einer Geraden

A. Quellcode Lane Inference System

In diesem Anhang wird der Quellcode zum Entwurf des Lane Inference Systems bereitgestellt. Dabei wird aufgrund des Umfangs darauf verzichtet, den gesamten Quellcode des Projekts hier anzuführen.

A.1. LaneInterpreter

Listing A.1: Laneinterpreter.h

```
1  #pragma once
2  #include "ISensorListener.h"
3  #include "StateReflector.h"
4  #include "FeatureSpace.h"
5  #include "TFALDALaneCandidateVector.h"
6  #include "TFALDARegionOfInterest.h"
7  #include "LaneInferenceSystem.h"
8
9  typedef struct i_result
10 {
11   CvPoint pi;
12   CvPoint qk;
13   int max;
14 } i_result;
15
16 class LaneInterpreter :
17   public ISensorListener
18 {
19 public:
20   TFALDARegionOfInterest _roi;
21
22   LaneInterpreter(void);
23   virtual ~LaneInterpreter(void);
24   virtual void notify();
25   int P(const CvPoint p);
26   void I(const CvPoint i, const IplImage* image, i_result& result);
```

```
27    int D(const CvPoint p, const CvPoint q);
28    void Ci(const int i, const IplImage* image,const int boundary,const
          LaneRegion& region, TFALDALaneCandidateVector& result);
29    void DistanceEvaluation(const FeatureSpace& featureSpace, const
          TFALDALaneCandidateVector& ln_p, TFALDALaneCandidateVector& result);
30    void getCV(const IplImage* image_roi,const int boundary, const
          LaneRegion& region, FeatureSpace& featureSpace);
31    void find_first_lane(const FeatureSpace& featurespace,
          TFALDALaneCandidateVector& result);
32    void reset();
33    IplImage* resample ( const IplImage* image, float ratio);
34    IplImage* sobel ( const IplImage* image, int direction);
35
36  private:
37    StateReflector& reflector;
38  };
39
40  template <class PEL>
41  class IplImageIterator {
42
43    int i, i0,j;
44    PEL* data;
45    PEL* pix;
46    int step;
47    int nl, nc;
48    int nch;
49
50   public:
51
52    /* constructor */
53    IplImageIterator(IplImage* image,
54       int x=0, int y=0, int dx= 0, int dy=0) :
55         i(x), j(y), i0(0) {
56
57      data= reinterpret_cast<PEL*>(image->imageData);
58      step= image->widthStep / sizeof(PEL);
59
60      nl= image->height;
61      if ((y+dy)>0 && (y+dy)<nl) nl= y+dy;
62      if (y<0) j=0;
63      data+= step*j;
64
65
66      nc= image->width ;
```

```
67      if ((x+dx)>0 && (x+dx)<nc) nc= x+dx;
68      nc*= image->nChannels;
69      if (x>0) i0= x*image->nChannels;
70      i= i0;
71
72      nch= image->nChannels;
73      pix= new PEL[nch];}
74
75
76    /* has next ? */
77    bool operator!() const { return j < nl; }
78
79    /* next pixel */
80    IplImageIterator& operator++() {i++;
81      if (i >= nc) { i=i0; j++; data+= step; }
82      return *this;}
83    IplImageIterator& operator+=(int s) {i+=s;
84      if (i >= nc) { i=i0; j++; data+= step; }
85      return *this;}
86    /* pixel access */
87    PEL& operator*() { return data[i]; }
88    const PEL operator*() const { return data[i]; }
89    const PEL neighbor(int dx, int dy) const
90      { return *(data+dy*step+i+dx); }
91    PEL* operator&() const { return data+i; }
92
93
94    /* current pixel coordinates */
95    int column() const { return i/nch; }
96    int line() const { return j; }
97  };
```

Listing A.2: Laneinterpreter.cpp

```
1  #include "StdAfx.h"
2  #include "LaneInterpreter.h"
3  #include "LaneRegion.h"
4  #include "TFALDASobelManager.h"
5  #include "TFALDAVisualizer.h"
6  #include "TFALDAlogger.h"
7  #include <boost/foreach.hpp>
8  #include <boost/lexical_cast.hpp>
9  #include "mmsystem.h"
10 #include <highgui.h>
11
```

```
12 /*
13   Zur initialisierung der ersten Runde
14  */
15 static bool firstround = true;
16 static LaneRegion previous_regions[CI_ROIQUANTITY];
17 LaneInferenceSystem lis;
18 static TFALDAlogger logger("e:\\logfiles_kurvenfahrt\\TMP");
19 static int counter;
20 static int bilder;
21 static float zeit;
22
23 LaneInterpreter::LaneInterpreter(void):
24 reflector(StateReflector::getInstance())
25 {
26   // lets listen to camera images
27   reflector.joinCameraImage(*this);
28   firstround = true;
29   counter = 0;
30   bilder=0;
31   zeit=0.0;
32   lis.init(previous_regions,_roi);
33 }
34
35 LaneInterpreter::~LaneInterpreter(void)
36 {
37 }
38
39 void LaneInterpreter::notify()
40 {
41   cout << this;
42   /* Bilder / Sek anzeigen
43   LONGLONG Frequency, CurrentTime, LastTime;
44   double TimeElapsed, TimeScale;
45   bool CounterAvailable = false;
46   if (QueryPerformanceFrequency( (LARGE_INTEGER*) &Frequency))
47   {
48     CounterAvailable = true;
49     TimeScale = 1.0/Frequency;
50     QueryPerformanceCounter( (LARGE_INTEGER*) &LastTime);
51   }
52   */
53
54   const CameraImage& img(reflector.getLastCameraImage());
55
```

```
56    //// evaluate the pushbuttons
57    //const PushButtonData& pbData = reflector.getLastPushButtonData();
58    //
59    //// yellow button -> change back to ApplicationStart
60    //if (pbData.isYellowButtonPressed())
61    //  reset();
62
63
64    // generate lane data somehow -> use tfalda
65    LaneData data;
66
67    //logger
68    logger.add_image(counter);
69
70    //Testausgabe der Bilder:
71    TFALDAVisualizer tv(img.getImage());
72
73    //Falls zu viele Fehler auftreten wird resettet
74
75    if(lis.fault_counter > 10)
76      reset();
77
78    //ALL ROIs
79    for(int i=0;i<CI_ROIQUANTITY;i++){
80        // left and right
81        for(int j=0;j<CI_BOXQUANTITY;j++){
82          const IplImage* image;
83          IplImage* image_resampled;
84          IplImage* image_sobeled;
85          FeatureSpace _featurespace;
86
87          int sobeloperator;
88          TFALDASobelManager sbm;
89
90          // Horizontal oder Vertikaler Sobel
91          sobeloperator = sbm.decide(previous_regions[i].get_lane_at(j).
                get_lane());
92          // kleines ROIrect holen
93          image = img.getRegionOfInterest(_roi.regions[i].get_lane_at(j).
                get_window());
94          image_resampled = resample(image,_roi.regions[i].
                get_resample_factor());
95          image_sobeled = sobel(image_resampled,sobeloperator);
96
```

```
97          img.resetROI();
98
99          getCV(image_sobeled, j, _roi.regions[i], _featurespace);
100
101         // In der ersten Runde haben wir keinen Vorvektor daher ist
                previous_regions immer Null und der
102         // Distanz algorithmus kann nicht angewendet werden.
103         if(firstround)
104           find_first_lane(_featurespace,_roi.regions[i].get_lane_at_w(j).
                get_lane_w());
105         else
106
107           DistanceEvaluation(_featurespace,previous_regions[i].
                get_lane_at(j).get_lane(),_roi.regions[i].get_lane_at_w(j).
                get_lane_w());
108
109         //free images
110         cvReleaseImage(&image_sobeled);
111         cvReleaseImage(&image_resampled);
112         //save for next round
113       }
114   }
115
116
117   int count=0;
118
119   // Anpassen der ROIs
120   if(!firstround)
121     lis.infernewlane(_roi,14); // 0.14 als alpha für lowpass, RC is 85!!!
122
123   if(!firstround)
124     // _roi.adjust(*previous_regions);
125     for(int i=0;i<CI_ROIQUANTITY;i++)
126         // _roi.regions[i].adjust(previous_regions[i]);
127         _roi.regions[i].adjust_total(previous_regions[i]);
128
129   //Vorvektor setzen
130   for (int i=0;i<CI_ROIQUANTITY;i++)
131     previous_regions[i] = _roi.regions[i];
132
133   /*
134   Ich zwinge meine Struktur in die Lanedata. Folgendes ist zu beachten.
        In jedem Region of interest gibt einen Vektor.
```

```
135   Dieser wird durch zwei Punkte beschrieben und einem Winkel der zwischen
          beiden entsteht
136   */
137   int c=0;
138   for(int i=0;i<CI_ROIQUANTITY;i++)
139    for(int j=0;j<CI_BOXQUANTITY;j++){
140      //linker Fall
141     if(j==0){
142       c=2*i;
143       //untere Kante des Vektors
144       data.hData[c].left.x = _roi.regions[i].get_lane_at(j).get_lane().
              get_qk().x - CI_LEAD ;
145       data.hData[c].left.y = _roi.regions[i].get_lane_at(j).get_lane().
              get_qk().y;
146       data.hData[c].left.d = _roi.regions[i].get_lane_at(j).get_lane().
              get_d();
147       /*logging*/
148       logger.add_tree("line");
149       logger.add_tag("point", boost::lexical_cast<std::string>(_roi.
              regions[i].get_lane_at(j).get_lane().get_qk().x) + "," +
150             boost::lexical_cast<std::string>(_roi.regions[i].
                  get_lane_at(j).get_lane().get_qk().y));
151       c++;
152
153       //obere Kante des Vektors
154       data.hData[c].left.x = _roi.regions[i].get_lane_at(j).get_lane().
              get_pi().x - CI_LEAD ;
155       data.hData[c].left.y = _roi.regions[i].get_lane_at(j).get_lane().
              get_pi().y;
156       data.hData[c].left.d = _roi.regions[i].get_lane_at(j).get_lane().
              get_d();
157       logger.add_tag("point", boost::lexical_cast<std::string>(_roi.
              regions[i].get_lane_at(j).get_lane().get_pi().x) + "," +
158             boost::lexical_cast<std::string>(_roi.regions[i].
                  get_lane_at(j).get_lane().get_pi().y));
159       logger.add_tag("direction",boost::lexical_cast<std::string>(data.
              hData[c].left.d));
160       logger.add_tag("lambda",boost::lexical_cast<std::string>(_roi.
              regions[i].get_lane_at(j).get_lane().get_L()));
161       logger.close_tree();
162     }
163     //rechter Fall
164     else {
165       c=2*i;
```

```
166         //untere Kante des Vektors
167         data.hData[c].right.x = _roi.regions[i].get_lane_at(j).get_lane().
                get_qk().x - CI_LEAD ;
168         data.hData[c].right.y = _roi.regions[i].get_lane_at(j).get_lane().
                get_qk().y;
169         data.hData[c].right.d = _roi.regions[i].get_lane_at(j).get_lane().
                get_d();
170         logger.add_tree("line");
171         logger.add_tag("point", boost::lexical_cast<std::string>(_roi.
                regions[i].get_lane_at(j).get_lane().get_qk().x) + "," +
172                 boost::lexical_cast<std::string>(_roi.regions[i].
                        get_lane_at(j).get_lane().get_qk().y));
173
174         c++;
175         //obere Kante des Vektors
176         data.hData[c].right.x = _roi.regions[i].get_lane_at(j).get_lane().
                get_pi().x - CI_LEAD ;
177         data.hData[c].right.y = _roi.regions[i].get_lane_at(j).get_lane().
                get_pi().y;
178         data.hData[c].right.d = _roi.regions[i].get_lane_at(j).get_lane().
                get_d();
179         logger.add_tag("point", boost::lexical_cast<std::string>(_roi.
                regions[i].get_lane_at(j).get_lane().get_pi().x) + "," +
180                 boost::lexical_cast<std::string>(_roi.regions[i].
                        get_lane_at(j).get_lane().get_pi().y));
181         logger.add_tag("direction",boost::lexical_cast<std::string>(data.
                hData[c].right.d));
182         logger.add_tag("lambda",boost::lexical_cast<std::string>(_roi.
                regions[i].get_lane_at(j).get_lane().get_L()));
183         logger.close_tree();
184
185       }
186     }
187     firstround = false;
188     counter++;
189     logger.close_tree();
190
191     tv.drawlanes(_roi);
192     tv.save(counter);
193     tv.display();
194
195     // store the lane data -> this will call all listeners
196     /* Bilder / sek
197     QueryPerformanceCounter((LARGE_INTEGER*) &CurrentTime);
```

```
198       TimeElapsed = (CurrentTime - LastTime)* TimeScale;
199     counter++;
200     zeit += TimeElapsed;
201     bilder = counter/zeit;
202     cout << "Bilder/Sek "<< bilder << endl;
203     */
204     reflector.updateDB(data);
205   }
206
207
208   //Caller must free!!!
209   IplImage* LaneInterpreter::resample( const IplImage* image, float ratio)
        {
210     IplImage* resample = cvCreateImage( cvSize((int)(cvGetSize(image).width
          /ratio), (int)(cvGetSize(image).height/ratio)) , 8, 1 );
211     cvResize(image,resample);
212     return resample;
213   }
214
215   //Caller must free!!!
216   IplImage* LaneInterpreter::sobel( const IplImage* image, int direction) {
217
218     IplImage* result = cvCreateImage( cvGetSize(image), 8, 1 );
219
220     assert(direction > -1 && direction < 2);
221     if(direction == 1)
222       cvSobel(image,result, 1, 0, 3);
223     if(direction == 0)
224       cvSobel(image,result, 0, 1, 3);
225     return result;
226
227     cvReleaseImage(&dest_dx);
228     cvReleaseImage(&dest_dy);
229     return result;
230     */
231   }
232
233   void LaneInterpreter::l(const CvPoint i, const IplImage* image, i_result&
        result)
234   {
235     int j;
236     int max=0;
237     CvPoint pj, qj;
238     uchar* dst_buf = 0;
```

```
239
240   // Maximale Laenge einer Linie kann nach Pythagoras c = sqrt(a^2 + b^2)
          sein. Es gilt jedoch immer
241   // folgende Ungleichung a+b > c. Somit vermeiden wir auf kosten des
          speicherbedarf swrt() und pow()
242   // operation.
243   dst_buf = (uchar*)cvAlloc( image->width + image->height);
244
245   //lane aufspannen
246   //startpunkt setzen
247   pj.x = i.x;
248   pj.y = i.y;
249   result.pi.x = i.x;
250   result.pi.y = i.y;
251
252   //Zielpunkt ganz unten ansetzen und über alle X-cood iterieren.
253   qj.y = cvGetSize(image).height-1;
254
255   // Iteration über die volle Breite.
256   for(j=0;j<cvGetSize(image).width;j++){
257     int count,sum;
258     sum=0;
259
260     //candidate Lanes aufspannen
261     qj.x = j;
262     count = cvSampleLine( image, pj, qj, dst_buf );
263
264     //Über alle punkte auf der Linie iterieren und Summe bilden
265     for(int k=0;k<count;k++){
266       sum += dst_buf[k];
267     }
268
269     if(sum > max){
270       max = sum;
271       result.max = max;
272       result.qk.x = qj.x;
273       result.qk.y = qj.y;
274     }
275   }
276   cvFree(&dst_buf);
277
278 }
279
280
```

```
281 int LaneInterpreter::P(const CvPoint p){
282   return p.x;
283 }
284
285 int LaneInterpreter::D(const CvPoint p, const CvPoint q){
286   /*
287    Richtung wird auf eine Dimension spezialisiert. Dies geht, da die
         Laenge irrelevant ist.
288    daher ist nur X-Abstand relevant.
289   */
290   return q.x - p.x; // - q.x;
291 }
292
293 void LaneInterpreter::Ci(const int i, const IplImage* image, const int
     boundary, const LaneRegion& region, TFALDALaneCandidateVector& result)
     {
294   CvPoint startl;
295   i_result _i_result = {0};
296   startl.x = i;
297   startl.y = 0;
298
299   //P()
300   result.set_p(P(startl));
301   //I()
302   I(startl,image,_i_result);
303   result.set_i(_i_result.max);
304   //D()
305   result.set_d(D(_i_result.pi,_i_result.qk));
306   //koordinaten speichern
307   result.set_pi(_i_result.pi);
308   result.set_qk(_i_result.qk);
309   region.normalize_vector(boundary, result);
310 }
311
312
313 void LaneInterpreter::DistanceEvaluation
314   (const FeatureSpace& featurespace, const TFALDALaneCandidateVector&
       In_p, TFALDALaneCandidateVector& result )
315 {
316 float Kp=7.33f;
317 // float Kp=3.33f;
318 float Ki=1.0f;
319 float Kd=10.67f;
320 // float Kd=2.67f;
```

```
321
322 double lambda=987654321.0;
323 double max=100000000;
324 double term1,term2,term3;
325
326 ln_p.get_p();
327
328 BOOST_FOREACH(TFALDALaneCandidateVector candidate, featurespace.
      _candidates ){
329   term1 = Kp * abs(candidate.get_p() - ln_p.get_p());
330   term2 = Kd * abs(candidate.get_d() - ln_p.get_d());
331   term3 = Ki * (ln_p.get_i() - candidate.get_i());
332   lambda = term1 + term2 + term3;
333
334   if( lambda < max){
335     max = lambda;
336     result = candidate;
337     result.set_L(lambda);
338   }
339 }
340 }
341
342 void LaneInterpreter::getCV(const IplImage* image_roi,const int boundary,
      const LaneRegion& region, FeatureSpace& featurespace){
343   int N;
344
345   N  = cvGetSize(image_roi).width;
346   for(int i=0;i<N;i++){
347     TFALDALaneCandidateVector lc;
348     Ci(i,image_roi,boundary,region,lc);
349     featurespace._candidates.push_back(lc);
350   }
351 }
352
353 void LaneInterpreter::find_first_lane(const FeatureSpace& featurespace,
      TFALDALaneCandidateVector& result){
354   int max = 0;
355   BOOST_FOREACH(TFALDALaneCandidateVector lc, featurespace._candidates) {
356     if(max < lc.get_i()){
357       max = lc.get_i();
358       result = lc;
359     }
360   }
361 }
```

```
362
363 void LaneInterpreter::reset() {
364   _roi = TFALDARegionOfInterest();
365   firstround=true;
366   lis.fault_counter=0;
367 }
```

A.2. LaneInferenceSystem

Listing A.3: LaneInferenceSystem.h

```
1  #pragma once
2  #include "TFALDALaneCandidateVector.h"
3  #include "TFALDARegionOfInterest.h"
4  #include "LaneRegion.h"
5
6  class LaneInferenceSystem
7  {
8  public:
9
10   const LaneRegion* _previous_regions;
11   int l_threshhold;
12   int fault_counter;
13   double RC;
14   int Y_roi_bottom_roadwidth[CI_ROIQUANTITY];
15   int Y_roi_top_roadwidth[CI_ROIQUANTITY];
16   int Y_1_roi_bottom_roadwidth[CI_ROIQUANTITY];
17   int Y_1_roi_top_roadwidth[CI_ROIQUANTITY];
18   int X_roi_bottom_roadwidth[CI_ROIQUANTITY];
19   int X_roi_top_roadwidth[CI_ROIQUANTITY];
20   int roi_roadwidth_constraint[CI_ROIQUANTITY];
21
22   bool lets_go;
23   // default constructor
24   LaneInferenceSystem();
25   ~LaneInferenceSystem(void);
26   void LaneInferenceSystem::infernewlane(TFALDARegionOfInterest&
       actual_lanes,const int dt);
27   void LaneInferenceSystem::init(const LaneRegion* previous_regions,const
        TFALDARegionOfInterest& roi);
28
29
```

```
30 private:
31   const bool LaneInferenceSystem::roadwidth_is_ok(const int i);
32   const bool LaneInferenceSystem::is_a_valid_road(const int i);
33   void LaneInferenceSystem::take_old_vector( LaneRegion& region, const
         int i);
34   void LaneInferenceSystem::check_and_correct(LaneRegion& region, const
         int i);
35   void LaneInferenceSystem::lowpass(const TFALDARegionOfInterest& roi,
         const double dt, const int i );
36   void LaneInferenceSystem::check_and_correct_vertically(
         TFALDARegionOfInterest& roi, const int i, const int j, const double
         toleranz);
37 };
```

Listing A.4: LaneInferenceSystem.cpp

```
1  #include "StdAfx.h"
2  #include "LaneInferenceSystem.h"
3  #include "TFALDARegionOfInterest.h"
4  #include "LaneRegion.h"
5
6
7  /* Lowpass pseudocode
8   // Return RC low-pass filter output samples, given input samples,
9   // time interval dt, and time constant RC
10  function lowpass(real[0..n] x, real dt, real RC)
11    var real[0..n] y
12    var real alpha := dt / (RC + dt)
13        y[0] := x[0]
14    for i from 1 to n
15        y[i] := alpha * x[i] + (1-alpha) * y[i-1]
16    return y
17 */
18
19 /*
20 Wir schaffen 50 Bilder pro Sekunde und legen als Minimalgeschwindigkeit
       einen Meter
21 pro Sekunde fest.
22 So erhalten wir ein Zeitintervall von dt = 14 und RC = 85 für 1 Meter pro
       Sekunde.
23 */
24
25 LaneInferenceSystem::LaneInferenceSystem():
26 l_threshhold(140),
27 lets_go(false),
```

```
28 fault_counter(0),
29 RC(85){}
30
31 LaneInferenceSystem::~LaneInferenceSystem(void)
32 {
33 }
34
35 void LaneInferenceSystem::init(const LaneRegion* previous_regions,const
       TFALDARegionOfInterest& roi) {
36 for(int i=0; i< CI_ROIQUANTITY; i++){
37     Y_roi_bottom_roadwidth[i] = roi.regions[i].get_lane_at(1).get_lane().
         get_qk().x - roi.regions[i].get_lane_at(0).get_lane().get_qk().x;
38     Y_roi_top_roadwidth[i] = roi.regions[i].get_lane_at(1).get_lane().
         get_pi().x - roi.regions[i].get_lane_at(0).get_lane().get_pi().x;
39     roi_roadwidth_constraint[i] = roi.regions[i].get_tolerance();
40   }
41 _previous_regions = previous_regions;
42 }
43
44
45 void LaneInferenceSystem::infernewlane(TFALDARegionOfInterest& roi, const
       int dt){
46
47   for(int i=0; i< CI_ROIQUANTITY; i++){
48     //Tiefpassfilter der Spurbreite
49     lowpass(roi, dt, i);
50     //Korrektur auf Basis der Spurbreitenprüfung!!!
51     check_and_correct(roi.regions[i],i);
52
53     //Fahrspur pro Seite Vertikal ausrichten (einzelne Lanes aneinander
         Knoten)
54
55     if(i < CI_ROIQUANTITY -1)
56       for(int j=0;j< CI_BOXQUANTITY; j++)
57         check_and_correct_vertically(roi,i,j,5);
58
59     //merke y-1
60     Y_1_roi_bottom_roadwidth[i] = Y_roi_bottom_roadwidth[i];
61     Y_1_roi_top_roadwidth[i] = Y_roi_top_roadwidth[i];
62   }
63 }
64 /*
65 * Validierungs und Verifikationsprüfungen der Fahrspurbreite mit
     Korrektur.
```

```
66 * Zunächst wird geprüft ob es sich überhaupt um eine Fahrspur handelt. Ob die mindest
67 * und maximale Breite nicht über- bzw unterschritten wurde. Danach wird geprüft ob sich die Breite
68 * nicht um einen Faktor (in tfaldaregionofinterest TOLERANCE genannt) ändert.
69 */
70
71 void LaneInferenceSystem::check_and_correct(LaneRegion& region, const int i) {
72
73   if( ! is_a_valid_road(i)){
74     take_old_vector(region,i);
75     fault_counter++;
76   }
77
78   else {
79     if ( ! roadwidth_is_ok(i) ){
80       take_old_vector(region,i);
81       fault_counter++;
82     }
83     else
84       if(fault_counter > 0)
85         fault_counter--;
86
87   }
88
89 }
90
91 void LaneInferenceSystem::take_old_vector(LaneRegion& region, const int i){
92   region.set_lane_at(0,_previous_regions[i].get_lane_at(0));
93   region.set_lane_at(1,_previous_regions[i].get_lane_at(1));
94 }
95
96 //AK1 entspricht Äquivalenzklasse 1!
     [-----------------|------|----------|------]
97 //                                  min                 0       50      150     max
98 // Straßenbreite z.B zwischen 50 und 150 pixel!!
99 const bool LaneInferenceSystem::is_a_valid_road(const int i){
100  bool result=true;
101  if(i==1){
```

```
102     result = result && X_roi_top_roadwidth[i] > 40 && X_roi_top_roadwidth
            [i] < 240;
103     result = result && X_roi_bottom_roadwidth[i] > 180 &&
            X_roi_bottom_roadwidth[i] < 292;
104   }
105   if(i==0){
106     result = result && X_roi_top_roadwidth[i] > 200 &&
            X_roi_top_roadwidth[i] < 272;
107     result = result && X_roi_bottom_roadwidth[i] > 320 &&
            X_roi_bottom_roadwidth[i] < 520;
108   }
109   return result;
110 }
111
112 const bool LaneInferenceSystem::roadwidth_is_ok(const int i){
113   if(abs(Y_roi_top_roadwidth[i] - Y_1_roi_top_roadwidth[i]) >
        roi_roadwidth_constraint[i])
114     return false;
115   if(abs(Y_roi_bottom_roadwidth[i] - Y_1_roi_bottom_roadwidth[i]) >
        roi_roadwidth_constraint[i])
116     return false;
117   return true;
118 }
119
120 void LaneInferenceSystem::check_and_correct_vertically(
      TFALDARegionOfInterest& roi, const int i, const int j, const double
      toleranz){
121   assert(i < CI_ROIQUANTITY && i >= 0);
122   int x1 = roi.regions[i+1].get_lane_at(j).get_lane().get_qk().x;
123   int x2 = roi.regions[i+1].get_lane_at(j).get_lane().get_pi().x;
124   int y1 = roi.regions[i+1].get_lane_at(j).get_lane().get_qk().y;
125   int y2 = roi.regions[i+1].get_lane_at(j).get_lane().get_pi().y;
126   //int y  = roi.regions[i].get_lane_at(j).get_lane().get_pi().y;
127   int y  = (roi.regions[i].get_lane_at(j).get_lane().get_pi().y + y1) /
        2;
128   int x  = x1 - (((x2 - x1) / (y2 - y1)) + (y - y1));
129
130   //korrigieren!!!
131   if(abs(x - roi.regions[i].get_lane_at(j).get_lane().get_pi().x) >
        toleranz){
132     roi.regions[i].get_lane_at_w(j).get_lane_w().get_pi_w().x = x;
133     roi.regions[i].get_lane_at_w(j).get_lane_w().set_p(x);
134   }
135 }
```

```
136
137 void LaneInferenceSystem::lowpass(const TFALDARegionOfInterest& roi,
        const double dt,const int i){
138
139   double alpha=dt/(RC + dt);
140   X_roi_bottom_roadwidth[i] = roi.regions[i].get_lane_at(1).get_lane().
        get_qk().x - roi.regions[i].get_lane_at(0).get_lane().get_qk().x;
141   X_roi_top_roadwidth[i] = roi.regions[i].get_lane_at(1).get_lane().
        get_pi().x - roi.regions[i].get_lane_at(0).get_lane().get_pi().x;
142
143   Y_roi_bottom_roadwidth[i] = alpha * X_roi_bottom_roadwidth[i] + (1-
        alpha) * Y_1_roi_bottom_roadwidth[i];
144   Y_roi_top_roadwidth[i] = alpha * X_roi_top_roadwidth[i] + (1-alpha) *
        Y_1_roi_top_roadwidth[i];
145
146
147 }
```

A.3. LaneRegion

Listing A.5: LaneRegion.h

```
1  #pragma once
2  #include "LaneRegionBoundary.h"
3  #include "TFALDALaneCandidateVector.h"
4
5  class LaneRegion
6  {
7  public:
8
9    const float get_resample_factor() const { return (_resample_factor); }
10   const int get_tolerance() const { return (_tolerance); }
11   const int get_offset() const { return (_offset); }
12   const int get_height() const { return (_height); }
13   const int get_roadwidth_top() const { return (_roadwidth_top); }
14   const int get_roadwidth_bottom() const { return (_roadwidth_bottom); }
15   // _lane[0] ist links _lane[1] ist rechts
16   const LaneRegionBoundary &get_lane_at(const int &pos) const
17   {
18     assert(pos >= 0 && pos < sizeof(_lanes));
19     return _lanes[pos];
20   }
```

```
21    // _lane[0] ist links _lane[1] ist rechts
22    LaneRegionBoundary &get_lane_at_w(const int &pos)
23    {
24      assert(pos >= 0 && pos < sizeof(_lanes));
25      return _lanes[pos];
26    }
27
28    void set_resample_factor(const float &resample_factor) {
          _resample_factor = resample_factor; }
29    void set_offset(const int &offset) {_offset = offset; }
30    void set_tolerance(const int &tolerance) {_tolerance = tolerance; }
31    void set_height(const int &height) {_height = height; }
32    void set_roadwidth_top(const int &width) {_roadwidth_top = width; }
33    void set_roadwidth_bottom(const int &width) {_roadwidth_bottom = width;
          }
34    void set_lane_at(const int &pos, const LaneRegionBoundary &lanebound)
35    {
36      assert(pos >= 0 && pos < sizeof(_lanes));
37      _lanes[pos] = lanebound;
38    }
39    void normalize_vector(const int boundary, TFALDALaneCandidateVector&
          result) const;
40    void LaneRegion::adjust(const LaneRegion& previous_region);
41    void LaneRegion::adjust_total(const LaneRegion& previous_region);
42    void LaneRegion::save_state(int i);
43    void LaneRegion::reset(int i);
44
45    // default constructor
46
47    LaneRegion();
48    ~LaneRegion(void);
49
50  private:
51
52    float _resample_factor;
53    int _offset;
54    int _height;
55    int _tolerance;
56    int _roadwidth_top;
57    int _roadwidth_bottom;
58    // _lane[0] ist links _lane[1] ist rechts
59    LaneRegionBoundary _lanes[2];
60
61  };
```

Listing A.6: LaneRegion.cpp

```
1  #include "StdAfx.h"
2  #include "LaneRegion.h"
3
4  static LaneRegion savestate[3];
5
6  LaneRegion::LaneRegion(void)
7  {
8  }
9
10 LaneRegion::~LaneRegion(void)
11 {
12 }
13
14 void LaneRegion::normalize_vector(const int boundary,
       TFALDALaneCandidateVector& result) const{
15   assert(boundary == 0 || boundary == 1);
16   LaneRegionBoundary roi = get_lane_at(boundary);
17   result.set_p(result.get_p() * get_resample_factor() + roi.get_x());
18   result.set_d(result.get_d() * get_resample_factor());
19   result.set_pi(cvPoint(result.get_p(),result.get_pi().y *
       get_resample_factor() + get_offset()));
20   result.set_qk(cvPoint(result.get_qk().x * get_resample_factor() + roi.
       get_x() ,result.get_pi().y  + get_height()));
21 }
22
23 void LaneRegion::adjust(const LaneRegion& previous_region)
24 {
25   int newleft;
26   int newright;
27
28   int leftcase = previous_region.get_lane_at(0).get_lane().get_p() - (
       previous_region.get_lane_at(0).get_x() + previous_region.get_lane_at
       (0).get_width()/2);
29   int rightcase = previous_region.get_lane_at(1).get_lane().get_p() - (
       previous_region.get_lane_at(1).get_x() + previous_region.get_lane_at
       (1).get_width()/2);
30
31   int relative = (int)((leftcase + rightcase) / 2);
32
33   if(relative > previous_region.get_tolerance())
34     relative = previous_region.get_tolerance();
35   if(relative < -previous_region.get_tolerance())
36     relative = -previous_region.get_tolerance();
```

```
37
38
39    newleft = get_lane_at(0).get_x() + relative;
40    newright = get_lane_at(1).get_x() + relative;
41
42    if(newleft < 25 || newright > 700)
43      return;
44    get_lane_at_w(0).set_x(newleft);
45    get_lane_at_w(1).set_x(newright);
46  }
47
48  //linke kante >0 && 752-breite -1
49  void LaneRegion::adjust_total(const LaneRegion& previous_region)
50  {
51    int p1;
52    int p2;
53    p1 = previous_region.get_lane_at(0).get_lane().get_p();
54    p2 = previous_region.get_lane_at(0).get_lane().get_p() +
        previous_region.get_lane_at(0).get_lane().get_d();
55    int l_x = min(p1,p2) - 50;
56    int l_width = 75+abs(previous_region.get_lane_at(0).get_lane().get_d())
        ;
57
58    //check links
59    l_width = min(275,l_width); // maximal 375
60    l_width = max(100,l_width); // minimal 100
61    l_x = min(752-l_width-1,l_x); // maximal x<breite -1
62    l_x = max(1,l_x); // x>0
63
64    p1 = previous_region.get_lane_at(1).get_lane().get_p();
65    p2 = previous_region.get_lane_at(1).get_lane().get_p() +
        previous_region.get_lane_at(1).get_lane().get_d();
66    int r_x = min(p1,p2) - 50;
67    int r_width = 75+abs(previous_region.get_lane_at(1).get_lane().get_d())
        ;
68
69    //check rechts
70    r_width = min(275,r_width); // maximal 375
71    r_width = max(100,r_width); // minimal 100
72    r_x = min(752-r_width-1,r_x); // maximal x<breite -1
73    r_x = max(1,r_x); // x>0
74
75    /*
```

```
76    * In der Kurve schauen wir Menschen nach links oder rechts in die Kurve
          .
77    * Diese Vorgang den das Gehirn vollzieht müssen wir in logik nachbauen
78    */
79
80    //linkskurve
81    if((l_width + l_x > r_x) && r_x < 325){
82      int k = l_width + l_x;
83      k = (k-r_x)/2;
84      l_x-=k;
85      l_width-=k;
86      //dennoch nicht über den Rand hinaus!!!
87      l_x=max(1,l_x);
88      l_width=max(100,l_width);
89    }
90
91    //rechtskurve
92    if((l_width + l_x > r_x) && (l_width + l_x) > 425){
93      int k = r_width + r_x;
94      k = (k-l_x)/2;
95      r_x+=k;
96      r_width-=k;
97      //dennoch nicht über den rechten Rand hinaus!!!
98      if(r_width+r_x>752){
99        int m = r_width+r_x-752;
100       r_width -= m;
101     }
102   }
103
104   get_lane_at_w(0).set_x(l_x);
105   get_lane_at_w(0).set_width(l_width);
106   get_lane_at_w(1).set_x(r_x);
107   get_lane_at_w(1).set_width(r_width);
108
109 }
110
111 void LaneRegion::save_state(int i){
112   savestate[i] = LaneRegion(*this);
113 }
114
115 void LaneRegion::reset(int i){
116   _resample_factor = savestate[i]._resample_factor;
117   _offset = savestate[i]._offset;
118   _height = savestate[i]._height;
```

```
119   _tolerance = savestate[i]._tolerance;
120 }
```

A.4. TFALDALogger

Listing A.7: TFALDALogger.h

```
1  #pragma once
2  #include <ostream>
3  #include <string>
4  #include <vector>
5  #include <boost/iostreams/device/file.hpp>
6  #include <boost/iostreams/stream.hpp>
7
8  namespace io = boost::iostreams;
9
10 class TFALDAlogger
11 {
12
13 public:
14   // default constructor
15   TFALDAlogger::TFALDAlogger();
16   TFALDAlogger::TFALDAlogger(const std::string &filename);
17   TFALDAlogger::~TFALDAlogger();
18   void TFALDAlogger::add_tree(const std::string & name);
19   void TFALDAlogger::add_image(int number);
20   void TFALDAlogger::close_tree();
21   void TFALDAlogger::close_image();
22   void TFALDAlogger::flush();
23   template <typename T>
24   void add_tag(std::string const & name, T const & data){
25       logfile << '<' << name << '>';
26       logfile << data;
27       logfile << "</" << name << ">\n";
28     }
29
30 private:
31   typedef io::stream<io::file_sink> ofstream;
32   ofstream logfile;
33   std::vector<std::string> tags;
34   std::size_t depth;
35
```

```
36 };
```

Listing A.8: TFALDALogger.cpp

```
1  #include "StdAfx.h"
2  #include "TFALDAlogger.h"
3  #include <boost/lexical_cast.hpp>
4
5  TFALDAlogger::TFALDAlogger(std::string const & t_filename):logfile(
       t_filename)
6  {
7    logfile << "<?xml␣version=\"1.0\"␣encoding=\"iso-8859-1\"␣?>\n";
8    add_tree("root");
9  }
10
11 TFALDAlogger::TFALDAlogger()
12 {
13   TFALDAlogger("e:\\logfiles\\defaultlog.xml");
14 }
15
16 TFALDAlogger::~TFALDAlogger(void)
17 {
18   while (tags.empty() == false)
19     {
20       close_tree();
21     }
22     logfile.close();
23 }
24
25 void TFALDAlogger::add_image(int imageseq){
26   logfile << "<image␣" << "seq␣=\"" << boost::lexical_cast<std::string>(
       imageseq) << '"' << ">␣\n";
27   tags.push_back("image");
28   ++depth;
29 }
30
31 void TFALDAlogger::add_tree(std::string const & name){
32   logfile << '<' << name << ">␣\n";
33   tags.push_back(name);
34   ++depth;
35 }
36
37 void TFALDAlogger::close_tree(){
38     if (tags.empty() == true || depth == 0) return; // we are at level 0
39     --depth;
```

```
40      logfile << "</" + tags[tags.size() - 1] + ">\n" ;
41      tags.pop_back();
42      flush();
43  }
44
45  void TFALDAlogger::close_image(){
46    TFALDAlogger::close_tree();
47  }
48
49  void TFALDAlogger::flush(){
50    logfile.flush();
51  }
```

B. Testsystem

In diesem Anhang wird der Quellcode zum Entwurf des Testsystems bereitgestellt. Dabei wird aufgrund des Umfangs darauf verzichtet des gesamten Quellcode des Projekts hier anzuführen und lediglich die relevanten Teile ausgewählt.

B.1. TestManager.java

Listing B.1: TestManager

```
1  package tfalda.test;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.io.FileReader;
6  import java.io.IOException;
7  import java.util.ArrayList;
8  import java.util.List;
9
10 import javax.xml.bind.JAXBException;
11
12 import org.jfree.ui.RefineryUtilities;
13 import org.w3c.dom.Document;
14 import org.w3c.dom.Element;
15 import org.w3c.dom.Node;
16 import org.w3c.dom.NodeList;
17 import org.xml.sax.InputSource;
18 import org.xml.sax.SAXException;
19
20 import com.sun.org.apache.xerces.internal.parsers.DOMParser;
21
22 public class TestManager {
23
24   private Document logDocument;
25   private String logfile;
26   private List<Image> images;
```

```
27    private ProvenImageManager pm;
28    private List<Datalist> datasets;
29
30    public TestManager() {
31      images = new ArrayList<Image>(10);
32      pm = new ProvenImageManager();
33      datasets = new ArrayList<Datalist>(5);
34    }
35
36    public static void main(String[] args) {
37      TestManager t = new TestManager();
38      /*
39       * Logfiles auswerten
40       */
41      for( File file: new File("e:\\logfiles_kurvenfahrt").listFiles()){
42        t.parseLogfile(file,0,80);
43        StatList sl = t.testImages();
44        t.addResults(sl,file.getName());
45        t.clearImages();
46      }
47      /*
48       * Nur ausgeben
49       */
50
51      t.showResults();
52    }
53
54    private void clearImages() {
55      this.images.clear();
56
57    }
58
59    private void addResults(StatList sl, String description) {
60      DataManager dm = new DataManager();
61      try {
62        Datalist dl = dm.convert(sl, description);
63        datasets.add(dl);
64      } catch (JAXBException e) {
65        // TODO Auto-generated catch block
66        e.printStackTrace();
67      }
68    }
69
70    private void showResults() {
```

```
71        Spliner demo = new Spliner("Spline_Demo");
72        for(Datalist d : datasets){
73          demo.add(d, d.getDescription());
74        }
75          demo.pack();
76           RefineryUtilities.centerFrameOnScreen(demo);
77          demo.setVisible(true);
78          demo.display("Kurvenfahrt");
79    }
80
81    private StatList testImages() {
82      StatManager sm = new StatManager();
83      for (Image i : images) {
84        ProvenImage pi = pm.getProvenImage(new Integer(i.getSequence()));
85        if (pi == null)
86          continue;
87        ImageTester t = new ImageTester(pi, i);
88        t.run();
89        sm.addStat(t.stats());
90      }
91
92      return sm.getStats();
93    }
94
95    private void parseLogfile(File file,Integer from, Integer to) {
96      DOMParser parser = new DOMParser();
97      try {
98        parser.parse(new InputSource(new FileReader(file)));
99      } catch (FileNotFoundException e) {
100       // TODO Auto-generated catch block
101       e.printStackTrace();
102     } catch (SAXException e) {
103       // TODO Auto-generated catch block
104       e.printStackTrace();
105     } catch (IOException e) {
106       // TODO Auto-generated catch block
107       e.printStackTrace();
108     }
109     logDocument = parser.getDocument();
110     Node rootnode = logDocument.getElementsByTagName(
111         Messages.getString("TestManager.node_root")).item(0); //$NON-NLS
              -1$
112     NodeList nl = ((Element) rootnode).getElementsByTagName(Messages
113         .getString("TestManager.node_image")); //$NON-NLS-1$
```

```
114
115     to = Math.min(nl.getLength(),to);
116     from = Math.max(0, from);
117
118     // Images
119     //for (int i = 0; i < nl.getLength(); i++) {
120     for (int i = from; i < to; i++) {
121       Node n = nl.item(i);
122       Image image = new Image();
123       NodeList lines = ((Element) n).getElementsByTagName(Messages
124           .getString("TestManager.node_line")); //$NON-NLS-1$
125       String imagenumber = "0"; //$NON-NLS-1$
126
127       try {
128         imagenumber = n.getAttributes().getNamedItem(
129             Messages.getString("TestManager.attr_sequence")) //$NON-NLS-1
                $
130             .getNodeValue();
131
132         imagenumber = Integer.toString(new Integer(imagenumber)+80);
133         image.setSequence(imagenumber);
134       } catch (NullPointerException e) {
135       }
136
137       // Lines
138       for (int j = 0; j < lines.getLength(); j++) {
139         Node linenode = lines.item(j);
140         NodeList points = ((Element) linenode)
141             .getElementsByTagName(Messages
142                 .getString("TestManager.node_point")); //$NON-NLS-1$
143         NodeList lambdas = ((Element) linenode)
144         .getElementsByTagName(Messages
145             .getString("TestManager.node_lambda")); //$NON-NLS-1$
146         Node lambdanode = lambdas.item(0);
147         Line l = new Line();
148         l.setLambda(LineHelper.parseLambda(lambdanode));
149         l.setDirection(LineHelper.parseDirection(linenode));
150         // Points
151         for (int k = 0; k < points.getLength(); k++) {
152           Node pointnode = points.item(k);
153           Point p = PointHelper.parsePoint(pointnode);
154           l.set(k, p);
155         }
156         image.addLine(l);
```

```
157         }
158         images.add(image);
159       }
160     }
161
162 }
163
164 \end{lstlisting
165 \section{ImageTester.java}
166 \begin{lstlisting}[caption=ImageTester,label=lst:app:imagetester]
167 package tfalda.test;
168
169 public class ImageTester {
170
171   private ProvenImage pi;
172   private Image i;
173   private boolean abort = false;
174   public Stat stats;
175
176   public ImageTester(ProvenImage provenImage, Image i) {
177     pi= provenImage;
178     this.i = i;
179     if(provenImage == null)
180       abort = true;
181     stats = new Stat();
182     stats.setImage(i);
183   }
184
185   public void run() {
186     if(abort)
187       return;
188     for(Line l : i.get_lines()){
189       int i=0,j=0;
190       Point[] points_to_check = LineHelper.lineFast(l);
191       for(Point p : points_to_check){
192         if(p == null)
193           continue;
194         i++;
195         if(checkPoint(p,pi))
196           j++;
197       }
198       stats.put(l,(float)j/i);
199     }
200   }
```

```
201
202   private boolean checkPoint(Point p, ProvenImage pi2) {
203     for(Point validpoint : pi2.getProvenPoints()){
204       if(checkPoint(p, validpoint,10))
205         return true;
206     }
207     return false;
208   }
209
210   private boolean checkPoint(Point p1, Point p2, int epsilon){
211     return (Math.abs(p1.getX() - p2.getX()) < epsilon) && (Math.abs(p1.
          getY() - p2.getY()) < epsilon) ;
212   }
213
214   public Stat stats() {
215     return this.stats;
216   }
217
218
219 }
```

B.2. LineHelper.java

Listing B.2: LineHelper

```
1  package tfalda.test;
2
3  import org.w3c.dom.Element;
4  import org.w3c.dom.Node;
5  import org.w3c.dom.NodeList;
6
7  public class LineHelper {
8
9    public static int parseDirection(Node xmlLinenode) {
10     Integer result;
11     String directionString;
12     NodeList nl  = ((Element)xmlLinenode).getElementsByTagName(Messages.
         getString("LineHelper.node_direction")); //$NON-NLS-1$
13     Node directionnode = nl.item(0);
14     directionString = directionnode.getTextContent();
15     result = new Integer(directionString);
16     return result;
```

```
17  }
18
19  // as suggested by http://www.cs.unc.edu/~mcmillan/comp136/Lecture6/
        Lines.html!!
20  public static Point[] lineFast(Line l) {
21    Point[] p = new Point[1024];
22    int x0 = l.get(0).getX();
23    int x1 = l.get(1).getX();
24    int y0 = l.get(0).getY();
25    int y1 = l.get(1).getY();
26    int dy = y1 - y0;
27    int dx = x1 - x0;
28    int stepx, stepy;
29
30    if (dy < 0) {
31      dy = -dy;
32      stepy = -1;
33    } else {
34      stepy = 1;
35    }
36    if (dx < 0) {
37      dx = -dx;
38      stepx = -1;
39    } else {
40      stepx = 1;
41    }
42    dy <<= 1; // dy is now 2*dy
43    dx <<= 1; // dx is now 2*dx
44
45    p[0] = new Point(x0, y0);
46    if (dx > dy) {
47      int i = 1;
48      int fraction = dy - (dx >> 1); // same as 2*dy - dx
49      while (x0 != x1) {
50
51        if (fraction >= 0) {
52          y0 += stepy;
53          fraction -= dx; // same as fraction -= 2*dx
54        }
55        x0 += stepx;
56        fraction += dy; // same as fraction -= 2*dy
57        p[i] = new Point(x0, y0);
58        i++;
59      }
```

```
60      } else {
61        int i = 1;
62        int fraction = dx - (dy >> 1);
63        while (y0 != y1) {
64          if (fraction >= 0) {
65            x0 += stepx;
66            fraction -= dy;
67          }
68          y0 += stepy;
69          fraction += dx;
70          p[i] = new Point(x0, y0);
71          i++;
72        }
73      }
74
75      return p;
76    }
77
78    public static int parseLambda(Node lambdanode) {
79      Integer result;
80      String directionString;
81      directionString = lambdanode.getTextContent();
82      result = new Integer(directionString);
83      return result;
84    }
85  }
```

B.3. DataManager.java

Listing B.3: DataManager

```
1  package tfalda.test;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.io.FileOutputStream;
6
7  import javax.sound.sampled.DataLine;
8  import javax.xml.bind.JAXBContext;
9  import javax.xml.bind.JAXBException;
10 import javax.xml.bind.Marshaller;
11 import javax.xml.bind.Unmarshaller;
```

```
12
13
14 public class DataManager {
15
16   public Datalist convert(StatList sl, String description) throws
         JAXBException{
17     Datalist dl = new Datalist();
18     dl.setDescription(description);
19       for(Stat s : sl.getStats()){
20             int i=0;
21             double conf=0.0;
22             i = s.getLine_confidence().size();
23             for(Line l : s.getLine_confidence().keySet()){
24               conf+=s.getLine_confidence().get(l);
25             }
26             conf/=i;
27             dl.add(new Double(s.getImage().getSequence()).doubleValue(),
                   conf);
28           }
29       return dl;
30   }
31
32   public Datalist importz(File file) throws FileNotFoundException,
         JAXBException{
33     return unmarshall(file);
34   }
35
36   public void marshall(Datalist dl, File file) throws JAXBException,
         FileNotFoundException {
37
38     JAXBContext context;
39     try {
40       context = JAXBContext.newInstance(StatList.class);
41       Marshaller m = context.createMarshaller();
42       m.marshal(dl, new FileOutputStream(file));
43     } catch (JAXBException e) {
44       // TODO Auto-generated catch block
45       e.printStackTrace();
46     }
47   }
48
49   public Datalist unmarshall(File file) {
50     JAXBContext context;
51     Datalist dl=new Datalist();
```

```
52      try {
53        context = JAXBContext.newInstance(Datalist.class);
54        Unmarshaller unm = context.createUnmarshaller();
55        dl = ((Datalist) unm.unmarshal(file));
56      } catch (JAXBException e) {
57        // TODO Auto-generated catch block
58        e.printStackTrace();
59      }
60      return dl;
61    }
62
63  }
```

B.4. Spliner.java / draw diagram

Listing B.4: Spliner

```
1
2   package tfalda.test;
3
4   import java.awt.Color;
5   import java.awt.Dimension;
6
7   import org.jfree.chart.ChartFactory;
8   import org.jfree.chart.ChartPanel;
9   import org.jfree.chart.JFreeChart;
10  import org.jfree.chart.plot.PlotOrientation;
11  import org.jfree.chart.plot.XYPlot;
12  import org.jfree.chart.renderer.xy.XYLineAndShapeRenderer;
13  import org.jfree.data.general.Dataset;
14  import org.jfree.data.xy.XYDataset;
15  import org.jfree.data.xy.XYSeries;
16  import org.jfree.data.xy.XYSeriesCollection;
17  import org.jfree.ui.ApplicationFrame;
18
19    private static final long serialVersionUID = 1L;
20    private XYSeriesCollection xsc = new XYSeriesCollection();
21
22    /**
23       * Creates a new demo instance.
24       *
25       * @param title  the frame title.
```

```
26          */
27         public Spliner(String title){
28             super(title);
29         }
30
31         /**
32          * Returns a sample dataset.
33          *
34          * @return The dataset.
35          */
36
37         private XYSeries converDatalistToSeries(Datalist dl, String key){
38           XYSeries series = new XYSeries(key);
39             for(Data d : dl.getDatalist()){
40               series.add(d.getSeq(),d.getConf());
41             }
42             return series;
43         }
44
45         private void addSeries(XYSeries series){
46           xsc.addSeries(series);
47         }
48
49         private Dataset getDataset() {
50             return (Dataset)xsc;
51         }
52
53         public void add(Datalist dl, String key){
54           addSeries(converDatalistToSeries(dl, key));
55         }
56
57         /**
58          * Creates a sample chart.
59          *
60          * @param dataset  the dataset.
61          *
62          * @return The chart.
63          */
64         private JFreeChart createChart(String title) {
65
66             // create the chart...
67             JFreeChart chart = ChartFactory.createXYLineChart(
68               title,                  // chart title
69                 "imagesequence",          // domain axis label
```

```
70              "confidence",               // range axis label
71              (XYDataset)getDataset(),    // data
72              PlotOrientation.VERTICAL,   // orientation
73              true,                       // include legend
74              true,                       // tooltips?
75              false                       // URLs?
76          );
77
78
79
80          // set the background color for the chart...
81          chart.setBackgroundPaint(Color.white);
82
83          // get a reference to the plot for further customisation...
84          XYPlot plot = (XYPlot) chart.getPlot();
85          plot.setBackgroundPaint(Color.lightGray);
86          plot.setDomainGridlinePaint(Color.white);
87          plot.setDomainGridlinesVisible(true);
88          plot.setRangeGridlinePaint(Color.white);
89
90          XYLineAndShapeRenderer renderer = new XYLineAndShapeRenderer();
91
92          renderer.setBaseShapesVisible(false);
93          plot.setRenderer( renderer );
94
95
96          return chart;
97
98      }
99
100   public void display(String title) {
101     JFreeChart chart = createChart(title);
102     ChartPanel chartPanel = new ChartPanel(chart, false);
103     chartPanel.setPreferredSize(new Dimension(300, 600));
104     setContentPane(chartPanel);
105   }
106
107 }
```

B.5. messages.properties

Listing B.5: messages.properties

```
1  ProvenImageHelper.path_to_provenImages=E:\\uni\\FTestbilder
2  ProvenImageHelper.regex_for_sequence=([0-9]*)\\.bmp
3  LineHelper.node_direction=direction
4  LineHelper.node_lambda=lambda
5  TestManager.logfile=e:\\testsystem.xml
6  TestManager.node_root=root
7  TestManager.node_image=image
8  TestManager.node_line=line
9  TestManager.attr_sequence=seq
10 TestManager.node_point=point
11 TestManager.node_lambda=lambda
12 StatManager.statisticfile=e:\\stats.xml
```

Literaturverzeichnis

[Ambler 2004] AMBLER, Scott W.: *The Object Primer: Agile Model-Driven Development with UML 2.0*. Cambridge University Press; 3 edition, 2004. – ISBN 0-5215-4018-6

[Ambler 2006] AMBLER, Scott W.: *UML 2 State Machine Diagrams*. 2006. – URL http://www.agilemodeling.com/artifacts/stateMachineDiagram.htm

[Baker 2003] BAKER, Herans ⋮ *Line Drawing - A study in optimization*. 2003. – URL http://www.cs.unc.edu/~mcmillan/comp136/Lecture6/index.html

[Berger 2008] BERGER, Dennis: *Fahrspurerkennung mit Three Feature Based Lane Detection Algorithm (TFALDA)*, Hochschule für Angewandte Wissenschaften Hamburg, Studienarbeit, 2008. – URL http://users.informatik.haw-hamburg.de/~ubicomp/arbeiten/studien/butrynowski.pdf

[Braunschweig 2008] BRAUNSCHWEIG, TU: *Carolo-Cup Regelwerk*. Dezember 2008. – URL http://www.carolo-cup.de/uploads/media/20080130_Carolo-Cup_Regelwerk.pdf

[Dawson 2003] DAWSON, Christian W.: *Computerprojekte im Klartext*. Pearson Studium, 2003. – ISBN 3-8273-7067-1

[Drosdowski u. a. 1997] DROSDOWSKI, Prof. Dr. Dr. h.c. Günther (Hrsg.) ; SCHOLZE-STUBENRECHT, Dr. W. (Hrsg.) ; WERMKE, Dr. M. (Hrsg.): *Das FremdWörterbuch*. Duden, 1997. – ISBN 3-411-04056-4

[H. Nickel 1990] H. NICKEL, H. Beinhoff W. Pauli H. Kreul W.: *Algebra und Geometrie für Ingenieure*. Harri Deutsch Thun, 1990. – ISBN 3-87-144-107-7

[Haberäcker 1995] HABERÄCKER, Peter: *Praxis der Digitalen Bildverarbeitung und Mustererkennung*. Carl Hanser Verlag München Wien, 1995. – ISBN 3-446-15517-1

[Hunt und Thomas 2003] HUNT, Andrew ; THOMAS, David: *Der Pragmatische Programmierer*. Hanser, 2003. – ISBN 3-446-22309-6

[Kemnitz 1999] KEMNITZ, Arnfried: *Mathematik zum Studienbeginn*. 2., verbesserte Auflage. Vieweg, 1999. – ISBN 3-528-16990-7

[Kohm und Morawksi 2003] KOHM, Markus ; MORAWKSI, Jens-Uwe: *KOMA-Script*. dante, 2003. – ISBN 3-936427-45-3

[Kubat 2007] KUBAT, Kordian: *Eine hierarchisches Steuerungsarchitektur mit Fuzzy-Regelung zur positionsmarkenbasierten Navigation eines autonomen Modellfahrzeugs*, Hochschule für Angewandte Wissenschaften Hamburg; Department Informatik, Diplomarbeit, 2007

[Labayrade u. a. 2006] LABAYRADE, Raphaël ; DOURET, Jerome ; LANEURIT, Jean ; CHAPUIS, Roland: A Reliable and Robust Lane Detection System Based on the Parallel Use of Three Algorithms for Driving Safety Assistance. In: *IEICE - Trans. Inf. Syst.* E89-D (2006), Nr. 7, S. 2092–2100. – ISSN 0916-8532

[Meyers 2008] MEYERS, Scott: *Effektiv C++ programmieren*. Addison-Wesley, München; Auflage: 3. Aufl., 2008. – ISBN 978-3827326904

[Reimers 2008] REIMERS, Hannes: *Fahrzeugsteuerung mit Fuzzy*, Hochschule für Angewandte Wissenschaften Hamburg; Department Informatik, Diplomarbeit, 2008

[Yim und Oh 2003] YIM, Young U. ; OH, Se-Young: Three-feature based automatic lane detection algorithm (TFALDA) for autonomous driving. In: *IEEE Transactions on Intelligent Transportation Systems* 4 (2003), Dezember, Nr. 4, S. 219–225. – ISSN 1524-9050

Glossar

Automatic Lane Detection Zweite Phase des TFALDA; beschreibt die Erkennung der Kanten mit Hilfe einer Distanzfunktion, welche Richtung, Startposition und Intensität durch Gewichte berücksichtigt

Faltungsmatrix Eine 3x3 Matrix, die der Sobel-Operator verwendet, um das kantenverstärkte Gradientenbild zu erhalten

Fault Lane Die durch das Lane Inference System deklarierte Lane, die zu einer ungültigen Fahrbahn führt

Lane Die durch die Distanzfunktion des TFALDA gefundene Fahrbahnmarkierung innerhalb einer ROI

Lane Inference System Dritte Phase des TFALDA; Korrektursystem, welches die Fahrbahnbreite sukzessive misst und überprüft, ob die Lanes zu einer gültigen Fahrbahn führen

Orakel, Testorakel Abstaktes System, welches befragt werden kann und nur richtig oder falsch antwortet

Orakel, Testorakel weiss ich jetzt nicht

Resampling Reduzierung der Pixelmenge; wird vom TFALDA in der ersten Phase verwendet

ROI Region Of Interest; ein Bereich, der für ein Bild definiert wird und in dem Algorithmen arbeiten können

Sobel-Operator Ein Kantenverstärkunsfilter, das in der ersten Phase des TFALDA verwendet wird

TFALDA Three Feature Based Lane Detection Algorithm

Tiefpassfilter Digitalisierte Simulation des analogen Filters, dient der Glättung der Fahrspurbreitenmessung

Zustandsautomat Eine Bezeichnung aus der formalen Informatik für einen endlichen Automaten; englisch: state machine

Versicherung über Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §25(4) ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 26. August 2008	
Ort, Datum	Unterschrift