Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Clemens Drauschke

Echtzeitfähige Startpunktalgorithmen für kamerabasierte Fahrspur-, Kreuzungs- und Hindernisidentifikation

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Clemens Drauschke

Echtzeitfähige Startpunktalgorithmen für kamerabasierte Fahrspur-, Kreuzungs- und Hindernisidentifikation

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stephan Pareigis
Zweitgutachter: Prof. Dr. Thomas Lehmann

Eingereicht am: 28.04.2016

Clemens Drauschke

Thema der Arbeit
Echtzeitfähige Startpunktalgorithmen für kamerabasierte Fahrspur-, Kreuzungs- und Hindernisidentifikation

Stichworte
Bildverarbeitung, Hinderniserkennung, Kreuzungserkennung, Schwellwertverfahren, Fahrspurerkennung, autonome Fahrzeuge, Carolo-Cup, Ausgleichsrechnung, Trajektorie, Inertiales Navigationssystem

Kurzzusammenfassung
Thema dieser Arbeit ist eine Fahrbahnerkennung für den Carolo-Cup. Es wird diskutiert wie aus dem Kamerabild Daten extrahiert werden, die für die Regelung der Geschwindigkeit und des Lenkwinkels eines autonomen Fahrzeuges benötigt werden.
Da in eingebetteten Systemen Rechenleistung eine kritische Komponente ist,
wird ein Startpunktalgorithmus präsentiert, der in unbearbeiteten Kamerabildern zur Spurfindung eingesetzt werden kann.
Ein einfaches regelbasiertes Verfahren für die Detektion der Mittellinie wurde realisiert. Das Prinzip aller dargestellten Ansätze besteht darin, soweit wie möglich, ohne einen vorhergehenden Zustand zu arbeiten.
Anschließend wird beschrieben, wie die extrahierten Informationen gefiltert und interpretiert werden.
Ein inertiales Navigationssystem soll die Robustheit der Bildverarbeitung erhöhen. Diese Arbeit kann als Anleitung für die Umsetzung einer kompletten Bildverarbeitung eines autonomen Fahrzeuges verstanden werden, das beim Carolo-Cup eingesetzt werden soll. Wichtige Transformationen werden in der Reihenfolge ihrer Anwendungen vorgestellt und deren Vorteile diskutiert. Das Ziel ist die Realisierung einer Bildverarbeitung die auf einem System-on-a-Chip Computer zum Einsatz kommen soll.
Clemens Drauschke

Title of the paper
Real-time capable starting point algorithms for camera-based lane, intersection and obstacle identification

Keywords
Image processing, obstacle identification, crossing identification, thresholding, lane detection, autonomous vehicles, Carolo-Cup, trajectory, Inertial Navigation System

Abstract
The topic of this work is a lane recognition software for the Carolo-Cup. Methods to obtain the requiered information for speed regulation and the steering angle estimation from camera image data will be presented.
Embedded systems have a limited computation power, therefore
a starting point algorithm, which can extract lane data from unprocessed camera images, will be presented.
A simple rule-based method for the detection of the center line will be realized. A description how the extracted information is filtered and interpreted will be given.
An inertial navigation system is used to increase the robustness of the image processing results. This work can be understood as a guide for the implementation of a complete image processing solution for an autonomous vehicle that can be used in the Carolo-Cup. Important transformations are presented in their order of execution and their advantages are discussed. The goal is the realization of a lane recognition software which runs on a System on a Chip computer.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Die in dieser Arbeit beschriebene Bilderkennung ist im Rahmen des Projekts Faust der HAW-Hamburg entstanden. Ziel des Projekts ist es ein autonomes Fahrzeug im Maßstab $1/10$ zu entwickeln. Dieses Fahrzeug darf beim Carolo-Cup in Braunschweig autonom auf einem Rundkurs fahren. Der Carolo-Cup ist ein studentischer, in mehrere Disziplinen gegliederter, Wettbewerb. Dieser beinhaltet statische und dynamische Disziplinen. Bei den dynamischen Disziplinen folgt das Fahrzeug autonom einer vorgegebenen Fahrbahn. Hindernissen auszuweichen und die Erkennung von Kreuzungen gehören zu den Aufgaben. Zudem gelten die Regeln der StVO. Eine Fachjury bewertet in der statischen Disziplin den verfolgten Ansatz unter den Aspekten der Innovation und weiteren Punkten wie der Rechtfertigung der entstandenen Kosten für den Aufbau des Fahrzeuges.

1.1 Ziel dieser Arbeit

Das Ziel dieser Arbeit besteht darin, eine robuste und echtzeitfähige Bilderkennung für den Carolo-Cup zu entwickeln, die auf einer günstigen Hardwareplattform zum Einsatz kommt. Eine Entkopplung zwischen der Bildverarbeitung und der Regelung soll durch ein Umgebungsmodell ermöglicht werden. In der Bildverarbeitung existieren mehrere traditionelle Ansätze für das Themengebiet der Linienerkennung. Diese Arbeit beschreibt deshalb eigene neue Konzepte, die speziell für den Carolo-Cup entwickelt und optimiert wurden. Durch den engen Verwendungskontext der Anwendung, der hier demonstrierten Bilderkennung, konnten spezielle und robuste Erkennungsmethoden entwickelt werden. Zudem wird versucht, mit den beschriebenen Ansätzen, einen der vorderen Plätze beim Carolo-Cup 2016 zu erreichen.

1.2 Anwendungsbereiche

Der Hauptanwendungsbereich der hier diskutierten Algorithmen ist der Carolo-Cup in Braunschweig. Die in dieser Arbeit vorgestellte Linienerkennung für unbearbeitete Kamerainformationen kann auch für andere Bereiche wie den Freescale Cup eingesetzt werden. Weitere generische Anwendungsszenarien der Linienerkennung sind denkbar. Die Mittellinienerkennung ist so spezialisiert auf den Carolo-Cup, dass diese nur mit weiteren Modifikationen für andere Einsatzgebiete fungieren kann.

1.3 Motivation

Wettbewerb ist die beste Motivation. Ein physikalisches Objekt unter der Beaufsichtigung von Prüfern per Software zu kontrollieren, die jeden Fahrfehler notieren und kein Fehlverhalten tolerieren, bringt die Entwicklung meiner Arbeit voran. Bildverarbeitung ist ein sehr weit gefächertes Themengebiet. Problemstellungen lassen sich auf unterschiedlichen wegen Lösen. Es wird gezeigt, wie eine Bildverarbeitung auf zeitintensive Operationen verzichten kann. Dies ist nur möglich, da sie in einem sehr engen Kontext zum Einsatz kommt. Hierdurch eröffnet sich die Möglichkeit Probleme mit einfachen Methoden zu lösen und eigene neue Ansätze zu entwickeln. Wenn die Komplexität der Algorithmen gering genug ist, um auf einer kostengünstigen Plattform die Echtzeitkriterien der Anwendung zu erfüllen, wird die Fachjury beim Wettbewerb dies positiv anerkennen. Es wird auf rechenintensive Operationen verzichtet, oder deren Anwendung durch tabellenbasierte Verfahren optimiert. Es gibt beim Carolo-Cup Teams die ihre Bildverarbeitung auf Grafikkarten durchführen, da zum Beispiel Partikelfilter eingesetzt werden. Es soll gezeigt werden, dass dieser Hardwareaufwand nicht notwendig ist.

1.4 Zu erfüllende Anforderungen

Eine wichtige selbst gewählte Anforderungen ist die Algorithmen der Bilderkennung weitestgehend zustandslos zu gestalten. Ältere Detektionsergebnisse sollen keinen Einfluss auf das aktuelle Detektionsergebnis haben. Jedes Ergebnis der Bildverarbeitung ist so weit wie möglich unabhängig von einem Vorzustand. Für diese Arbeit war diese Anforderung der grundlegende Gedanke bei der Entwicklung aller Algorithmen. Die vorgestellten Algorithmen müssen robust arbeiten. Unter diese Robustheit fallen Aspekte wie die Kompensation von Fehlstellen innerhalb des Fahrbahnverlaufs, unterschiedliche Beleuchtungsszenarien und das korrekte Erkennen von Hindernissen in Fehlstellen. Die Funktionalität der Software soll getestet werden. Eine visuelle Prüfung des Verhaltens der Algorithmen auf erzeugten Bildinformationen muss zu diesem Stand der Entwicklung zum Testen ausreichen. Beim Carolo-Cup gibt es unterschiedliche Typen von Linien die klassifiziert werden müssen. Diese zu differenzieren ist eine der grundlegenden Anforderungen. Auch die Erkennung von Hindernissen auf beiden Fahrspuren muss gewährleistet werden. Eine weitere Anforderung ist die rotationsinvariante Detektion von Kreuzungssituationen aus großen Distanzen, damit frühzeitig bei höheren Geschwindigkeiten ein Anhalten an diesen ermöglicht wird. Es reicht nicht aus Kreuzungen aus Distanzen unterhalb von einem Metern zu erkennen.
Fehlerhaft ausgewertete Bildinformation sollen durch eine hohe Framerate relativiert werden. Es ist von Vorteil, wenn die zurückgelegte Strecke zwischen zwei ausgewerteten Bildern gering ist. Die Zykluszeit δ_t der bildverarbeitenden Prozesskette muss deshalb niedrig sein.
δ_t darf nicht größer sein als 14 ms. Die 14 ms entstehen durch die gewählte Bildwiederholfrequenz der Kamera. Die Berechnungen der Bildverarbeitung innerhalb eines δ_t durchzuführen stellt die gewählte Echtzeitanforderung dar.
Die Eigenbewegung des Fahrzeugs zwischen Bildverarbeitung und der Ansteuerung der Aktorik muss für eine korrekte Längs- und Querführung berücksichtigt werden. Deshalb wird eine logische Abstraktionssicht zwischen Bildverarbeitung und Regelung eingeführt. Diese ermöglicht ein homogenes Verhalten bei unterschiedlichen Geschwindigkeiten des Fahrzeugs und Wiederholfrequenzen der Kamera.
Die Anforderung Schwellwerte dynamisch zu erstellen, sichert bei wechselnden Beleuchtungssituationen ein korrektes Verhalten zu.

1.5 Koordinatensystem

In der Bildverarbeitung liegt der Ursprung des Koordinatensystems oftmals oben links (0,0). Auch in dieser Arbeit werden die Postionen von Koordinaten so beschrieben.

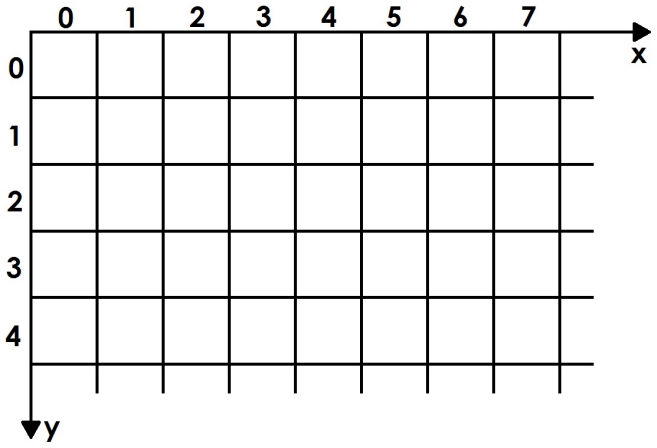

Abbildung 1.1: Koordinatensystem

Eine Zeile stellt horizontale Einträge dar. Spalten stellen vertikale Einträge dar.

1.6 Einbettung in die Software-Architektur des Fahrzeuges

Im Faust Projekt kommt derzeit eine selbst entwickelte Daten und Zeit-gesteuerte Architektur zum Einsatz. Die Bildererkennung wird innerhalb dieser Architektur periodisch asynchron ausgeführt. Sobald neue Ergebnisse der Bilderkennung bereit stehen, werden die Konsumenten dieser Informationen benachrichtigt. Als Datenquelle dient ein Shared Memory in dem die Kamera 8 Bit Pixelwerte ablegt. Vor dem Aufrufen der Bildverarbeitungskette wird eine lokale Kopie von diesem Shared Memory erzeugt. Die Algorithmen arbeiten dann auf dieser Kopie.

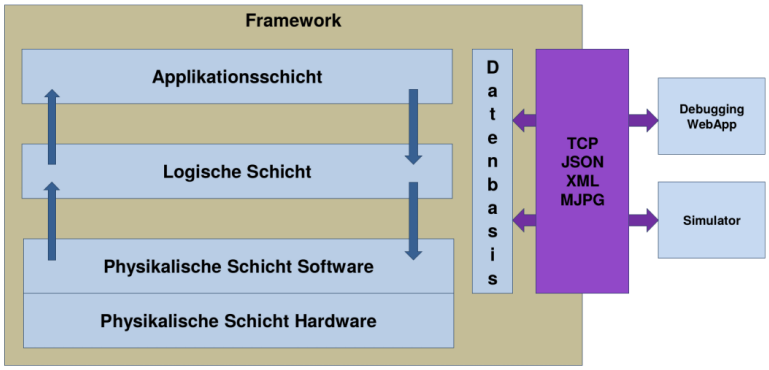

Abbildung 1.2: Software-Architektur

Die Bilderkennung befindet sich in der Applikationsschicht der Softwarearchitektur. Alle Ergebnisse der Auswertung stehen innerhalb der Datenbasis zur Verfügung. In der logischen Schicht wurde das Umgebungsmodell für die äquidistante Regelung des Fahrzeuges implementiert. Dieses Umgebungsmodell wird durch ein inertiales Navigationssystem realisiert (Kapitel 10).

1.7 Hardware

Die verwendete monochrome USB Industriekamera UI-1226LE-M von IDS stellt bei einer Auflösung von 752 mal 480 Pixeln 87,5 $^{\text{Frames}}/_{\text{Sekunde}}$ bereit. Ziel ist es mindestens 70 $^{\text{Frames}}/_{\text{Sekunde}}$ auszuwerten. Bei höheren Framerates ist die Belichtungszeit eventuell zu gering und das Bild verliert an Kontrast. 70 $^{\text{Frames}}/_{\text{Sekunde}}$ bedeutet, dass maximal 14,286 Millisekunden für die Auswertung eines jeden Frames benötigt werden dürfen. Da die Kamera durch ihren Neigungswinkel sehr viel von dem Fahrzeug betrachtet, werden lediglich die oberen 240 Zeilen der Kamera an die Rechenkomponente übertragen. Die Kamera ist in der Nähe der hinteren Fahrzeugachse platziert. Hierdurch wirkt sich ein Lenkwinkelwechsel unkritischer auf den Bildinhalt aus. Bei der aktuellen Hardwarekonfiguration kommt ein ODROID-XU4 Einplatinencomputer der Firma Hardkernel zum Einsatz. Dieser PC ist ARM basiert und verbraucht relativ wenig Energie. Die Aktor-Sensor-Schnittstelle wurde mit einem Teensy 3.1 Mikrocontroller realisiert. Die Zykluszeit der Kommunikation von Teensy und Odroid beträgt eine Millisekunde. Hall-Sensoren und ein Kreiselinstrument dienen der Eigenpositionsabschätzung. Die Informationen dieser Sensorik wird für eine logische Abstraktionssicht zwischen Bildverarbeitung und Ansteuerung der Aktorik des Fahrzeugs benötigt. Des Weiteren befindet sich seitlich ein Lichttaster auf dem Fahrzeug.

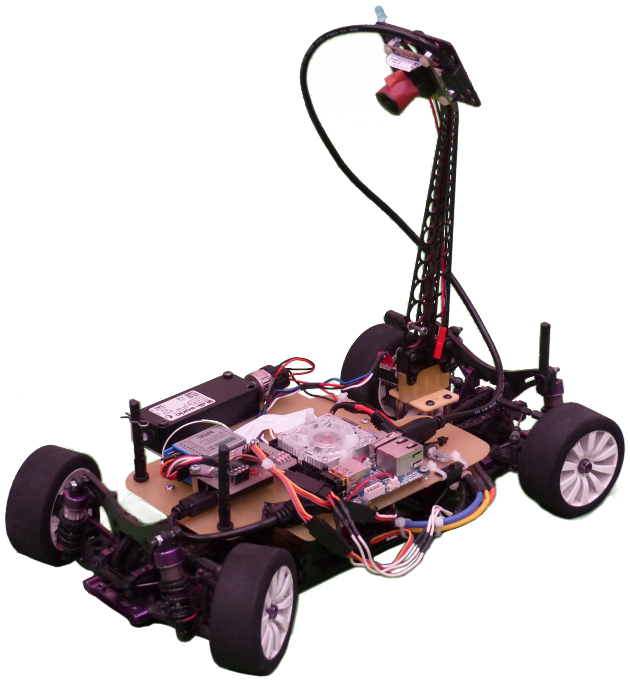

Abbildung 1.3: Hardware-Aufbau LaK-XU4000

2 Schwellwertberechnung

Damit die Bildverarbeitung in unterschiedlichen Beleuchtungssituationen eingesetzt werden kann, ist eine dynamische Erstellung von Schwellwerten unabdingbar. Auch unterschiedliche Verschlusszeiten der Kamera dürfen sich nicht negativ auf die Bildverarbeitung auswirken. Der Merkmalsraum der Detektion kann des Weiteren durch dynamische Schwellenwertverfahren reduziert werden. In der Arbeit wurde ein globales Schwellwertverfahren gewählt.

Abbildung 2.1: Unbearbeiteter Bildinhalt

In Abbildung 2.1 sind Reflexionen zu erkennen. Wählt man einen Schwellwert der zu niedrig ist, können weiße Inseln bei Verfahren wie der Binarisierung entstehen. Alle Algorithmen dieser Arbeit verwenden Otsu's Method (siehe Juan Pablo Balarini (2015)) für die Bestimmung von Schwellwerten.
Bei 8-Bit Graustufenbildern gibt es 256 mögliche Intensitäten die ein Pixelwert besitzen kann. I beschriebt den maximalen Wert dieser Intensitäten. Das Verfahren bestimmt einen Schwellwert, indem es die zu untersuchenden Pixelwerte in zwei Klassen aufteilt. Eine dieser Klassen beschreibt den Hintergrund $C1$ und die andere $C2$ den Vordergrund. Der berechnete Schwellwert t wird durch die Pixelintensität bestimmt die $C1$ und $C2$ separiert. Zu $C1$ gehören Pixelintensitäten von $[1, 2, 3, ..., t]$ und zu $C2$ die Pixelintensitäten von $[t + 1, t + 2, ..., I]$.
Für die Evaluation von t wird ein Histogramm (Code: 1 Zeilen 4-9) erstellt. Es wird gespeichert wie oft jede der möglichen Pixelintensitäten der zu untersuchenden Pixelwerte auftrat.

Eine Wahrscheinlichkeitsfunktion P wird zu jeder Pixelintensität gefunden. Jede Koordinate (x,y), eines Bildes M mit einer Anzahl von Pixeln N, besitzt einen Grauwert g = M(x,y) $\in$ [0,1,..,255]. Die Wahrscheinlichkeit, dass g $\in [0, 1, .., 255]$ ist, ist demnach:

$$P(g) = \frac{\text{Anzahl der Koordinaten, deren Grauwert g entspricht}}{N}$$

Durch eine Normalisierung dieses Histogramms wird die sichergestellt, dass die Wahrscheinlichkeitsverteilung eingehalten wird (Code: 1 Zeilen 11-13). Dies soll heißen, dass die Summe aller Wahrscheinlichkeiten die das Histogramm beinhaltet 1 ist.
Der Algorithmus minimiert die gewichtete Interklassenvarianz σ_w^2 (Code: 1 Zeilen 20-24). Diese wird durch den folgenden Ausdruck beschrieben:

$$\sigma_w^2 = q1(t)\sigma_1^2(t) + q2(t)\sigma_2^2(t)$$

Der Ausdruck $q1$ (Code: 1 Zeile 15) beschriebt die Wahrscheinlichkeiten, dass ein Pixelwert zu $C1$ gehört. Der Ausdruck $q2$ (Code: 1 Zeile 16) beschriebt die Wahrscheinlichkeiten, dass ein Pixelwert zu $C2$ gehört.

$$q1 = \sum_{i=1}^{t} P(i) \quad q2 = \sum_{i=t+1}^{I} P(i)$$

Die durchschnittlichen Intensitätswerte $\mu1$ und $\mu2$ beider Klassen werden bestimmt (Code: 1 Zeilen 18-19).

$$\mu1 = \sum_{i=1}^{t} \frac{iP(i)}{q1(t)} \quad \mu2 = \sum_{i=t+1}^{I} \frac{iP(i)}{q2(t)}$$

Es fehlen nun noch die Werte $\sigma_1^2(t)$ und $\sigma_2^2(t)$ für die Bestimmung der Innerklassenvarianz von $C1$ und $C2$. Durch die Minimierung dieser Werte wird anschließend der endgültige Schwellwert bestimmt.

$$\sigma_1^2(t) = \sum_{i=1}^{t} [i - \mu1(t)]^2 \frac{P(i)}{q1(t)} \quad \sigma_2^2(t) = \sum_{i=t+1}^{I} [i - \mu2(t)]^2 \frac{P(i)}{q2(t)}$$

Die Gesamtvarianz des Bildinhaltes σ^2 resultiert dann aus der Addition der beiden Klassenvarianzen. σ^2 beschreibt eine Konstante und ist unabhängig vom Schwellwert t.

$$\sigma^2 = \sigma_w^2(t) + \sigma_b^2(t) \quad \sigma_b^2 = q1(t)q2(t)[\mu1(t) - \mu2(t)]^2$$

Jetzt wird $\sigma_w^2(t)$ minimiert. Bei dieser Minimierung wird der endgültige Schwellwert evaluiert. Man könnte auch $\sigma_b^2(t)$ maximieren, was in der Implementation des Algorithmus realisiert wurde.

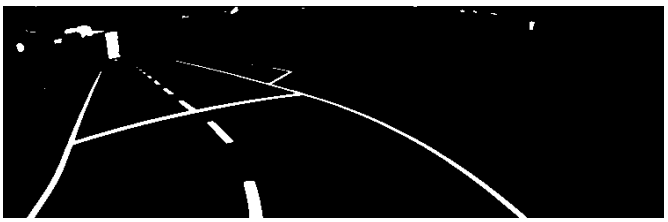

Abbildung 2.2: Otsu's Method

Mit Otsu's Method wurde ein Schwellwert T_{otsu} für die Grafik 2.1 erstellt. Für jeden Pixelwert in Abbildung 2.1 wurde geprüft, ob dieser über dem berechneten Schwellwert T_{otsu} liegt. Ist der Pixelwert $P_{original}$ einer Koordinate K größer als T_{otsu}, wird für die Postion von K ein Wert von 1 gespeichert. Man erhält das Binärbild das in Abbildung 2.2 zu sehen ist. Der Schwellwert liegt bei 108. Die Pixelwerte des Bildinhaltes haben eine Tiefe von acht Bit.

$$P_{binarisiert}(x,y) = \begin{cases} 1, & \text{if } P_{\text{original}}(\text{x,y}) > T_{\text{otsu}} \\ 0, & \text{if } P_{\text{original}}(\text{x,y}) \leq T_{\text{otsu}} \end{cases}$$

2.1 Pseudocode

Algorithm 1 Otsu's Method

```
1: max_intensity ← 256
2: image ← read(img)
3: N ← image.height * image.width
   {create Histogram}
4: for y = 0 y < image.height y++ do
5:    for x = 0 x < image.width x++ do
6:       value ← image(x, y)
7:       histogram[value] ← histogram[value] + 1
8:    end for
9: end for
10: threshold, var_max, sum, sumB, q1, q2, u1, u2 = 0 : init variables
11: for i = 1 i < max_intensity i++ do
12:    sum ← sum + i * histogram[i]
13: end for
14: for t = 1 i < max_intensity t++ do
15:    q1 ← q1 + histogram[t] {update q_i(t)}
16:    q2 ← N − q1
17:    sumB ← sumB + t * histogram[t] {update μ_i(t)}
18:    μ1 ← sumB/q1
19:    μ2 ← (sum − sumB)/q2
20:    VarianceBetweenClasses ← q1(t) * q2(t)[μ1(t) − μ2(t)]^2 {update μ_i(t)}
21:    if VarianceBetweenClasses > var_max then
22:       threshold ← t
23:       var_max ← VarianceBetweenClasses
24:    end if
25: end for
26: return threshold
```

3 Startpunkt Analyse

Die später in der Arbeit diskutierte Außenlinienerkennung (Natter 4.1) benötigt einen initialen Startpunkt $SP_{aktuell}$ und einen Winkel α. Mit $SP_{aktuell}$ ist eine Punkt auf einer der Außenlinien gemeint, der horizontal mittig auf dieser platziert ist (siehe Abbildung 4.2). Der Winkel α errechnet sich aus einem weiteren Punkt auf der Außenlinie, der über $SP_{aktuell}$ liegt. In Abbildung 4.2 bestimmt α die Ausrichtung des mittleren der fünf Zeiger, die direkt an $SP_{aktuell}$ grenzen. Alle Variablen der Form $X_{0,1,2}$ enthalten korrespondierende Elemente $0, 1, 2,$. Die initialen Tupel $SW_{0,1,2}$ aus Koordinaten $K_{0,1,2}$ und Winkeln $\alpha_{0,1,2}$ wird für jeden Startpunkt $SP_{0,1,2}$, beim Update der Bildinformation, neu bestimmt. Für die Extraktion der Informationen für jedes SW wird der Bildinhalt zyklisch mit einem Muster verglichen. Sobald die Kriterien des Musters erfüllt werden, wurden valide SW der Fahrbahn extrahiert. Drei horizontale eindimensionale Suchräume $SR_{0,1,2}$, mit einem definierten Zeilenabständen ZA1 und ZA2, werden in das Bild gelegt und mit den charakteristischen Eigenschaften des Musters verglichen. Jeder SR beinhalten den Inhalt einer kompletten Zeile Z des Bildmaterials. Es wird die Präsenz und Absenz von Informationen geprüft. Die Schwellwerte $T_{0,1,2}$ werden für $SR_{0,1,2}$ bestimmt. Anschließend wird eine eindimensionale Maske mit einer Breite von fünf Pixeln über $SR_{0,1,2}$ geschoben. Liegt ein Pixelwert P innerhalb der Maske über $T_{0,1,2}$ des zu untersuchenden $SR_{0,1,2}$, wird eine logische eins mit dem korrespondieren Index der aktuellen Maskenposition und des SR gespeichert. Ist $P \leq T_{0,1,2}$ wird eine null gespeichert. Diese binarisierten Ereignisse werden in $Events_{0,1,2}$ abgelegt.

Bei einem Schwellwert von $T_{Beispiel} = 100$ entsteht folgendes Ergebnis bei den in Abbildung 3.1 dargestellten Maskenpostionen 0 und 2.

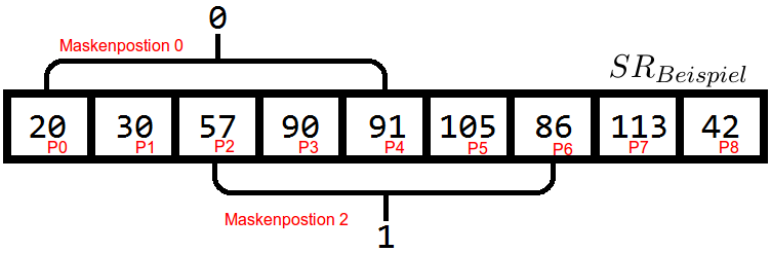

Abbildung 3.1: Beispielhafter Ausschnitt eines Suchraums

Maskenposition 0 erzeugt als Ergebnis eine null, da keiner der Werte (P0,P1,P2,P3,P4) über $T_{Beispiel}$ liegt. Das Ergebnis von Maskenposition 2 ist eine logische eins, da P5 einen Wert besitzt, der größer ist als $T_{Beispiel}$.

Im nächsten Schritt werden $Events_{0,1,2}$ auf konsekutive Ereignisse kE untersucht.
N_{min} wurde individuell für jeden SR aus Bildmaterial gewonnen.
$Event_1$ wird zum Veranschaulichen EV genannt. Eine Transition T besteht aus dem korrespondierenden Y-Wert der Zeile Z ihres SR und einem X-Wert der dem i, das die Bedingung aus 3.1 erfüllt, entspricht. i ist ein Index der binarisierten Ergebnisse aus $Events_{0,1,2}$.
$kE_{0,1,2}$ enthält diese Transitionen die aus $Events_{0,1,2}$ gebildet werden.

$$\sum_{j=0}^{j=N_{min}} EV_{i+j} = N_{min} \tag{3.1}$$

Aus den Transitionen kE_1 werden konsekutive dreier Tupel $T_{consecutive}$ gebildet.
Die Koordinate KM zwischen zwei Koordinaten K1 und K2 wird durch die Mittelwertbildung der X und Y Anteile von K1 und K2 berechnet.

$$KM(x) = \frac{K1(x) + K2(x)}{2}$$

$$KM(y) = \frac{K1(y) + K2(y)}{2}$$

Es wird ein δ_{Thresh} definiert. Druch Mittelwertbildung der Intensitäten $K1$ und $K2$ entsteht der Wert M. $K1$ und $K2$ sind benachbart, wenn ihre euklidische Distanz

$$ek = \sqrt{(K1(x) - K2(x))^2 + (K1(y) - K2(y))^2} \tag{3.2}$$

kleiner ist, als ein vorher definiertes δ_{ek}. Des Weiteren muss erfüllt werden, dass die Intensität KM_{val} von KM ähnlich zu M ist. Es muss $M - \delta_{Thresh} < KM_{val} \leq M + \delta_{Thresh}$ gelten.

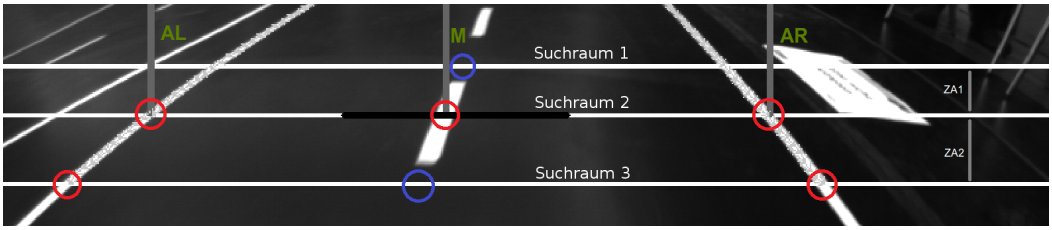

Abbildung 3.2: Startpunkt Analyse

Die rot markierten Bereiche in Abbildung 3.2 müssen Transitionen aufweisen. Blau markierte Bereiche in Abbildung 3.2 dürfen keine Informationen beinhalten. Sobald in Suchraum zwei aus Abbildung 3.2 (entspricht SR_1) mehr als zwei Transitionen gefunden werden wird geprüft, ob diese bestimmte Eigenschaften besitzen. Diese indizieren die Korrektheit der extrahierten Informationen.
Die drei Koordinaten K, L, M entsprechen einem T aus $T_{consecutive}$.

$$D_{left} = |K(x) - L(x)| \quad D_{right} = |L(x) - M(x)| \tag{3.3}$$

$$D_{max} = \max D_{left}, D_{right} \quad D_{min} = \min D_{left}, D_{right} \tag{3.4}$$

$$D_{rel} = \frac{D_{max}}{D_{min}} \tag{3.5}$$

D_{rel} muss innerhalb eines durch Testdaten definierten Bereiches liegen, sonst wird das Tupel verworfen.
Besitzt das mittlere Element eines Tupels T aus $T_{consecutive}$ keine direkten Nachbarn aus $kE_{0,2}$ (Abbildung 3.2 Suchraum 1,3) und die anderen Transitionen von T sind mit Transitionen aus kE_2 benachbart (Abbildung 3.2 Suchraum 3), stimmt das oben genannte Muster mit den Informationen der Fahrspur überein. Aus einem solchen benachbarten Tupel entstehen $SW_{0,1,2}$ für die linke, mittlere und rechte Fahrbahnmarkierung. Das jeweilige $\alpha_{0,1,2}$

des $SW_{0,1,2}$ stellt den Winkel zwischen benachbarten Transitionen dar. T beinhaltet die Koordinaten K, L, M. K ist benachbart mit K' und M mit M'. Dann wird das SW_0 für die linke Außenlinie aus K und dem Winkel zwischen K und K' gebildet. SW_2 für die rechte Außenlinie wird aus M und dem Winkel zwischen M und M' gebildet. Das SW_1 für die Mittellinie besteht aus L und dem arithmetischen Mittel der Winkel von SW_0 und SW_2.
Die Klassifikation von Startpunkten unterteilt die Fahrspur in drei Klassen von Linien. Danach können die Algorithmen für die jeweilige Linienklasse die Fahrspurinformation extrahieren. Nach der Startpunktextraktion werden die Startpunkte mittels Suchfenstern $F_{0,1,2}$ verfolgt. Die Startpunktextraktion legt die Bereiche für $F_{0,1,2}$ fest. $F_{0,1,2}$ werden auch durch eindimensionale Suchräume realisiert. Innerhalb der Fenster $F_{0,1,2}$ wird weiter nach Transitionen gesucht. Wurde eine Transition gefunden, bildet diese das neue Zentrum des Fensters für die nächste Suche. Die Suche nach einer Transition erfolgt analog zu dem oben beschriebenen Verfahren zur Erstellung von kE.
Sobald das hier beschriebene Muster der Startpunktextraktion erneut zutrifft, werden die Suchbereiche von $F_{0,1,2}$ neu gesetzt. Dies macht den Verlust eines Startpunkts, der durch das Verfolgen mittels der Fenster gefunden wurde unkritisch, da durch das Muster alle 20 zurückgelegten cm neue Bereiche für $F_{0,1,2}$ bestimmt werden können. Diese 20 cm entsprechen dem Abstand zweier Mittellinienelemente. Die Mustererkennung arbeitet zustandslos. Lediglich die Suchbereiche von $F_{0,1,2}$ besitzen einen Zustand. Die Gültigkeit dieser Zustände wurde durch die zustandslose Mustererkennung begrenzt, da $F_{0,1,2}$ im Durchschnitt nur für 20 cm Transitionen lokalisieren müssen. Der Quellcode für das beschriebene Verfahren liegt in StartPointAnalyzer.cpp.
Die Entscheidung eine Mustererkennung im originalen Bildinhalt der Kamera durchzuführen entstand durch die limitierten Möglichkeiten eines älteren Stands der diskutierten Arbeit. Dennoch ist es möglich Startpunkte aus dem originalen Bildinhalt zu extrahieren, da die perspektivischen Einflüsse in den Zeilen von $SR_{0,1,2}$ noch keinen zu starken Einfluss haben. Die vertikal eingezeichneten Linien (AL, M und AR) in Abbildung 3.2 sollen zeigen, dass das Muster gefunden wurde und die Linien nun in Außenlinie links (AL SP_0), Mittellinie (M SP_1) und rechte Außenlinie (AR SP_2) separiert werden können.

4 Fahrspurdetektion

4.1 Natter

Die Natter beschreibt das Verfahren der Außenlinienerkennung.
Die Natter ist ein iterativer Algorithmus. Für die initiale Iteration dienen die Informationen der Startpunktextraktion.
Die Idee hinter diesem Algorithmus besteht darin, von einem Startpunkt SP aus eine Schar von Zeigern Z in das Bild zu legen. Diese beschreiben eine Untermenge eines Kreissegments mit einem Öffnungswinkel von 144°. Jeder Zeiger Z' von Z weicht von seinem Nachbarzeiger um 28.6478 ° ab und hat eine Länge von 9 Pixeln. Die geometrische Anordnung Z besitzt einen Schwerpunkt STM. STM wird durch die von Z verdeckten Pixelintensitäten bestimmt.
Ein Aufruf der Natter mit SP_0 als SP erkennt die linke Außenlinie. Wird die Natter mit SP_2 aufgerufen, erkennt sie die rechte Außenlinie.
Der Quellcode zu diesem Algorithmus liegt in Natter.cpp und Natter.h.

4.2 Die Iteration im Detail

Z wird projiziert. Hierfür wird initial ein SP der Startpunktanalyse (siehe: Kapitel 3) verwendet. Zeilen 8 bis 15 des Pseudocodes Algorithm 2 zeigen wie Koordinaten von jedem Zeiger Z' von Z beschrieben werden. Die Koordinate aus SP setzt den Startpunkt $SP_{aktuell}$ der aktuellen Iteration. Das α aus SP gibt die Ausrichtung des mittleren Z' von Z an. Die von Z verdeckten Pixelwerte werden dafür benutzt ein Schwellwert S zu bilden (Algorithm 2 Zeile 16). Anschließend wird jeder Zeiger Z' von Z auf ein Tupel T abgebildet. T besteht aus einer Koordinaten K und einem akkumulierten Helligkeitswerten I. L entspricht, ausgehend von dem Startpunkt der jeweiligen Iteration, der ersten Koordinate des jeweiligen Z', deren Intensität unterhalb von S liegt. Die Koordinate auf Z' vor L wird K genannt. Die Intensitäten aller Koordinaten des jeweiligen Z' die vor L liegen werden akkumuliert. Das Ergebnis der Akkumulation I wird zusammen mit K als Tupel in TM abgelegt (Algorithm 2 Zeilen 20 bis 34). Liegt die letzte Koordinate von einem Z' oberhalb von S, determiniert diese K. Dieses Verfahren ist auch in Natter.cpp in der Methode polarScan implementiert. Der

Schwerpunkt STM von TM wird berechnet (siehe: 3). Innerhalb der Vierer-Nachbarschaft (siehe Abbildung 4.1) von STM wird der Pixelwert maximiert. Dies bedeutet, dass der Algorithmus sich für die hellste benachbarte Koordinate entscheiden kann.
Ist zum Beispiel die Intensität von Koordinate
$((STM(x), STM(y+1)) > (STM(x), STM(Y))$ wird STM auf
$(STM(x), STM(y+1))$ abgebildet.

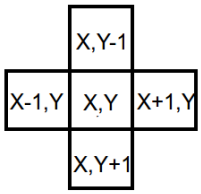

Abbildung 4.1: Vierer-Nachbarschaft

Das Ergebnis dieser Entscheidung nennen wir STM_MAX.
Der Winkel γ zwischen $SP_{aktuell}$ und STM_MAX wird berechnet.

$$\gamma = atan2(STM\_MAX(y) - SP_{aktuell}(y), STM\_MAX(x) - SP_{aktuell}(x)) \tag{4.1}$$

In $SP_{aktuell}$ wird STM_MAX gespeichert. Das α der nächsten Iteration ist das Ergebnis γ. Die Iteration startet erneut.
Sobald ein Ausgangspunkt stabil bleibt, eine maximale Anzahl von Iterationen erreicht worden ist, oder keine Pixelintensität unterhalb von Z über S liegt, terminiert der Algorithmus.
In Abbildung 4.2 sind die Zeiger länger als 9 Pixel. Die schematische Darstellung soll lediglich zeigen, wie der Algorithmus arbeitet.

Bei der Umsetzung des Algorithmus wurde versucht so maschinennah wie möglich zu arbeiten. Außerdem wurden verschiedene Parameter für die Länge der Zeiger und deren Anzahl getestet. So konnte die Laufzeit von unter $0,75$ Millisekunden für die Detektion einer Linie erreicht werden. Die Natter ist ein Anytime Algorithmus. Man könnte den Algorithmus deshalb durch eine obere Laufzeitgrenze vorzeitig terminieren lassen und die bis zu diesem Zeitpunkt gefunden Koordinaten auswerten.

Algorithm 2 polarScan : Calculate Threshold of Z and afterwards TM

```
1:  Z'Count ← 5
2:  range ← 72°
3:  step ← 28.6478°
4:  Z'Length ← 9
5:  x, y ← Starpunkt {this is SP_aktuell}
6:  direction ← α − range
7:  valuesForOtsu : array
8:  for i = 0 i < Z'Count i++ do
9:      for j = 0 j < Z'Length j++ do
10:         xNow, yNow ← polarCoordinate(x, y, j, direction)
11:         pixVal ← PixelValueAt(xNow, yNow)
12:         valuesForOtsu ← pixVal
13:     end for
14:     direction ← direction + step
15: end for
16: Threshold ← otsu(valuesForOtsu)
17: for i = 0 i < Z'Count i++ do
18:     accu ← 0
19:     wasSet ← false
20:     for j = 0 j < Z'Length j++ do
21:         xNow, yNow ← polarCoordinate(x, y, j, direction)
22:         pixVal ← PixelValueAt(xNow, yNow)
23:         if pixVal ≤ Threshold then
24:             xNow, yNow ← polarCoordinate(x, y, direction, j − 1)
25:             TM ← (xNow, yNow, accu)
26:             wasSet ← true
27:             break
28:         end if
29:         accu ← accu + pixVal
30:     end for
31:     if !wasSet then
32:         xNow, yNow ← polarCoordinate(x, y, direction, Z'Length − 1)
33:         TM ← (xNow, yNow, accu)
34:     end if
35:     direction ← direction + step
36: end for
```

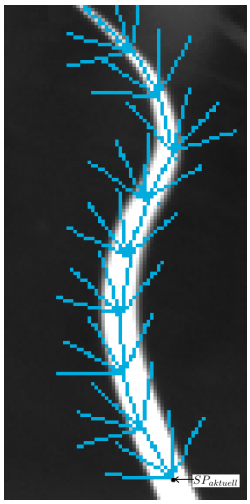

Abbildung 4.2: Schematische Darstellung der Funktionsweise

4.3 Geometrische Momente

Mittels geometrischer Momente kann der Schwerpunkt eines geometrischen Objekts ermittelt werden (siehe: Seite 48 Meisel (2012)). Dieses Objekt wird hier durch TM beschrieben. Der Schwerpunkt wird in der Methode filterResult berechnet.
Die Koordinate des Tupels aus TM mit dem größten akkumulierten Helligkeitswert wird insgesamt sechs mal stärker gewichtet als alle anderen Koordinaten der Tupel. Es hat sich gezeigt, dass die Laufzeit des Algorithmus so gesenkt werden kann, ohne gravierende Verluste der Güte der Erkennung zu riskieren..

Algorithm 3 filterResult : Calculate Center of Gravity of Set TM

1: $moment00 \leftarrow 0.0$
2: $moment01 \leftarrow 0.0$
3: $moment10 \leftarrow 0.0$
4: **for** i = 0 i<TM'Length i++ **do**
5: $\quad intensity \leftarrow TM[i].pixelValue$
6: $\quad moment00 \leftarrow moment00 + intensitiy$
7: $\quad moment10 \leftarrow moment10 + pixelValue * TM[i].x$
8: $\quad moment01 \leftarrow moment01 + pixelValue * TM[i].y$
9: **end for**
10: $winnerX \leftarrow moment10/moment00$
11: $winnerY \leftarrow moment01/moment00$
12: changeToBrightestCoordianteWithinReach(winnerX,winnerY)
13: **return** $winnerX, winnerY$

Der Algorithmus 3 gewichtet das K mit der größten Intensität nicht stärker als andere Tupel. Der Code der Natter führt diese Operation dennoch aus.

4.4 Polarkoordinaten

Von einer Koordinate (x, y) aus kann, in einem Koordinatensystem das über zwei Dimension verfügt, jede andere Koordinate durch einen Abstand l und einem Winkel α berechnet werden. Der Pseudocode 4 zeigt die Umsetzung dieser Idee. Die Natter benutzt approximierte Winkelfunktionen für die Berechnung von Polarkoordinaten. Jede Pixelposition der Zeiger Z' kann durch einen Winkel, der aktuellen Länge des Zeigers und der Ausgangskoordinate der Iteration beschrieben werden. Bei einer Laufzeitanalyse auf der Zielplattform hat sich gezeigt, dass die Berechnung der Winkelfunktionen schneller ist als ein tabellenbasiertes Verfahren. Durch die Approximation der Winkelfunktionen konnte weitere Rechenzeit eingespart werden. Auf die Approximation möchte ich nicht weiter eingehen. Dies war lediglich ein Versuch die Ausführungszeit des Codes noch weiter zu reduzieren.

Algorithm 4 polarCoordinate : Parameters are x and y, angle α, and a pointer length l

Require: α is a radian value otherwise convert

1: $x \leftarrow x + l * cos(\alpha)$
2: $y \leftarrow y + l * sin(\alpha)$
3: **return** x,y

4.5 Ergebnisse

Die Ergebnisse zeigen wie viel Fahrbahn man, mit einer doch sehr niedrigen Auflösung der monochrome USB Industriekamera, erkennen kann. Durch das Schwerpunkt orientiere Verfahren sind auch Linien verfolgbar, die man nur noch mit dem menschlichen Auge erahnen kann. Dies in nicht bearbeiteten Bildern zu ermöglichen ist nicht einfach. Für die später gewählte Abstraktion der Fahrbahn werden sogar Informationen verworfen, da diese keine doppelt definierten Zeilen erlaubt.

In den nächsten Abbildungen wurden die Bereiche, die mittels der Natter detektiert worden sind, blau gekennzeichnet.

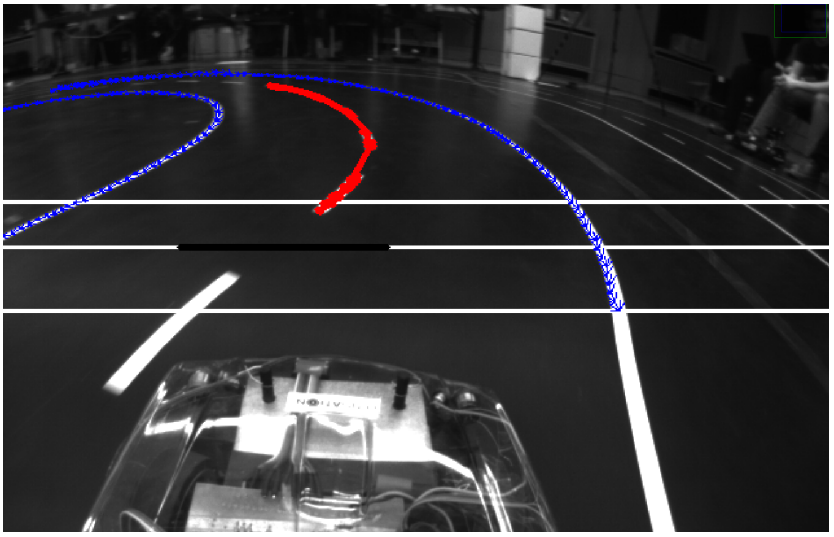

Abbildung 4.3: Die Natter erkennt eine Linkskurve

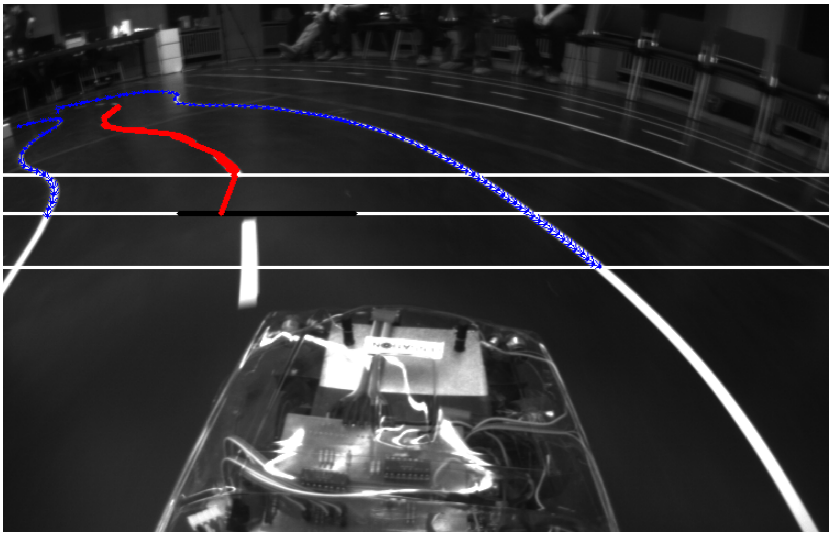

Abbildung 4.4: S-Kurve

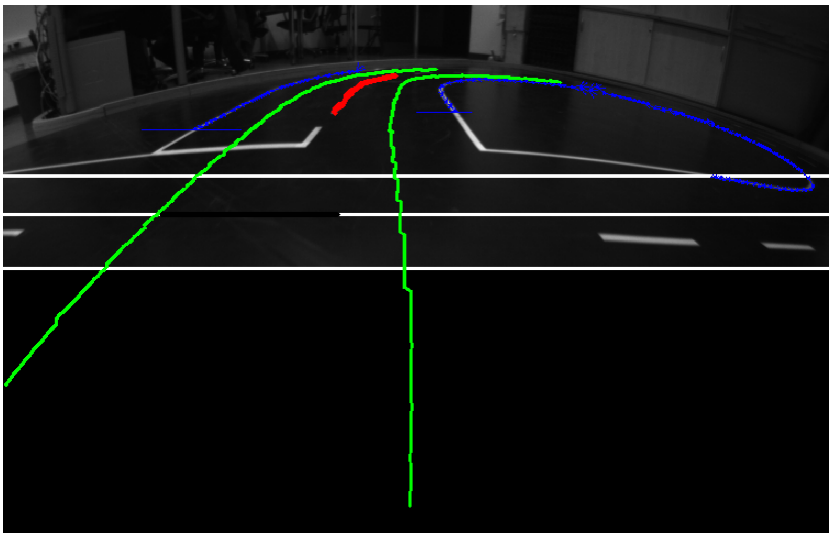

Abbildung 4.5: Weitere Startpunkte hinter der Kreuzungssituation

Abbildung 4.5 zeigt, dass die Natter mit weiteren Startpunkten, die hier beispielsweise durch die Kreuzungserkennung generiert wurden, auch Leerstellen überbrücken kann. Es zeigt sich jedoch, dass die Weite der Erkennung in einigen Situationen nicht von Vorteil ist. Da die Fahrbahn später mittels Polynomen abstrahiert wird, müssen alle detektierten Punkte verworfen werden die Bildzeilen doppelt definieren. Die Länge der in grün eingezeichneten Trajektorien verdeutlicht diesen notwendigen Schritt.

4.6 Nachteile der Natter

Der Algorithmus terminiert sobald eine zu verfolgende Linie endet. Leerstellen können deshalb nicht überbrückt werden, ohne neue Startpunkte aus anderen detektierten Linien zu gewinnen. Beim Carolo-Cup fehlen maximal zwei der drei Markierungen der Fahrspur. Da theoretisch immer Informationen vorhanden sind, können neue Startpunkte aus den Informationen anderer detektierter Linien erstellt werden. Somit können Leerstellen indirekt überbrückt werden. Ein Fehler der auftreten kann ist, dass die Natter sich in Haltelinien und Start-Stopp Markierungen iterieren kann, obwohl die ursprüngliche Linie hinter der Markierung eventuell weitere Informationen enthält.

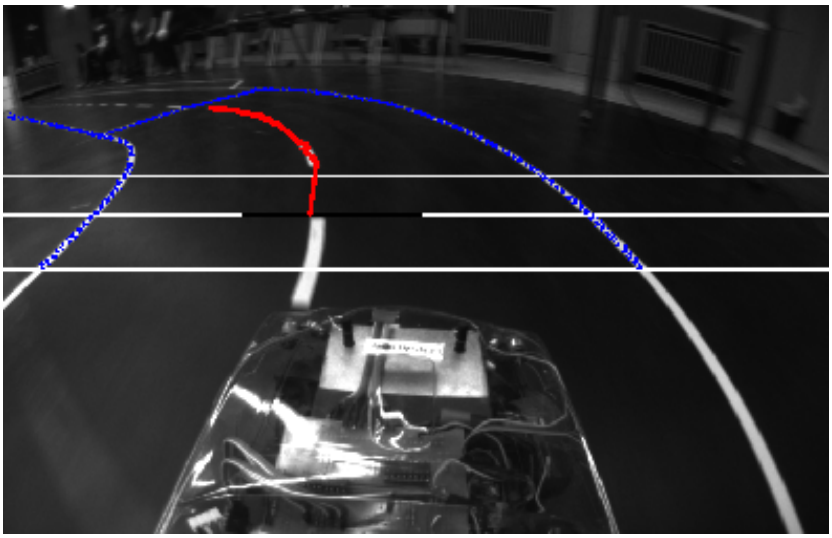

Abbildung 4.6: Die Natter macht beim Finden der rechten Außenlinie einen Fehler und iteriert in die Startmarkierung.

Diese Fehler müssen durch weitere Algorithmen gefunden werden. Der Natter dieses Fehlverhalten zu verbieten führt zu anderen nicht in Kauf zu nehmenden Einschränkungen. Es ist eine interessante Frage, ob diese Fehler wirkliche Fehler darstellen. Das Kapitel der Kreuzungserkennung wird sich näher mit dieser Thematik beschäftigen (Kapitel 11). Diese Korrekturen sind ohne eine Linsenkorrektur und eine perspektivische Transformation nur sehr schwer möglich und verkomplizieren den Code der Natter, ohne garantieren zu können diese Ereignisse aufzuspüren. Die Abstände zwischen den gefundenen Koordinaten, der durch die Natter detektierten Linien, sind nicht äquidistant. Auch dieser Punkt wird durch weitere Algorithmen entschärft.

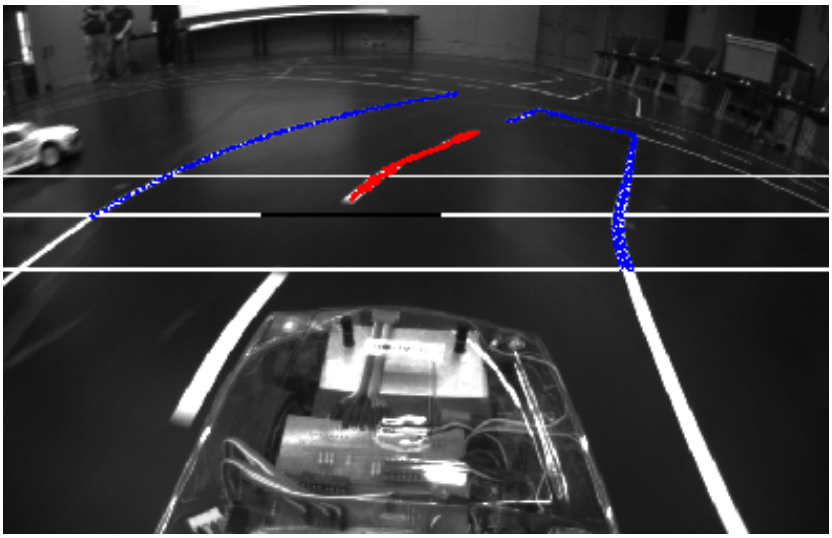

Abbildung 4.7: Die Natter macht beim Finden der rechten Außenlinie einen Fehler und iteriert in die Haltelinie der Kreuzung.

4.7 Vorteile der Natter

Durch die sehr große Erkennungstiefe entstehen in der Regel bei der Auswertung eines Frames valide Fahrspurinformationen für die nächsten vier Meter. Wenn man das nutzt, kann man die Längsregelung des Fahrzeugs optimieren und für eine gewisse Zeit ohne Kamerabild fahren. Das Kapitel über das inertiale Navigationssystem 10 wäre ohne die große Erkennungstiefe der Natter teilweise irrelevant. Da die Natter sehr ressourcenschonend ist, läuft sie auf eingebetteten Systemen die weniger Rechenleistung als die gewählte Zielplattform haben. Bei Tests mit einem Raspberry Pi 2 Model B wurden die gewählten Echtzeitanforderungen eingehalten.

5 Linsenkorrektur

5.1 Target-to-Source, Source-to-Target

In der Bildverarbeitung kommt es sehr oft vor, dass man Daten in eine andere Darstellung überführen muss. In dieser neuen Darstellung sind Operationen zulässig, die vorher nicht gültig waren. Dies erleichtert dem Entwickler die Programmierung. Hierfür müssen zwei Begriffe definiert werden. Diese Begrifflichkeiten drücken die Richtung der jeweiligen Transformation aus. Eine Funktion $f(x, y) = K_{abgebildet_{x,y}}$ wird gefunden, die der Abbildungsvorschrift der Darstellungsüberführung entspricht.
Die Formeln 6.1 und 6.2 sind ein Beispiel für $f(x, y)$.
Das Quellbild wird Q_{img} genannt. Das Zielbild wird Z_{img} genannt.
Unter dem Begriff Source-to-Traget versteht man eine Zuordnung jeder Pixelposition aus Q_{img} nach Z_{img}. Da Z_{img} unter Umständen mehr Pixelpositionen als Q_{img} besitzt, können Definitionslücken in Z_{img} entstehen.
Unter dem Begriff Target-to-Source versteht man eine äquivalente Zuordnung, deren Richtung umgekehrt ist. Da jeder Pixelposition von Z_{img} durch f ein Ursprung zugeordnet werden kann, entstehen keine Lücken in Z_{img}.

5.2 Korrektur der Verzerrung

Das Kamerabild ist stark verzerrt. Innerhalb dieses Bilds zu regeln, oder regelbasierte Operationen durchzuführen, ist fast nicht möglich. Für weitere Schritte der Bildverarbeitung muss die Verzerrung entfernt werden. Hierfür wurde ein einfacher Algorithmus für die Linsenkorrektur gewählt (siehe: Algorithm 5). Dieses Verfahren versucht die Eigenschaften der gewählten Linse abzubilden und zu entfernen. Die Idee hierfür stammt von Helland (2013a) und der Pseudocode für die Implementation von (Helland, 2013b).

Bei der Beschreibung von Linsenverzeichnung ist ein polares Koordinatensystem sinnvoll. Die Hauptachse der Linse ist dessen Ursprung.
Es wird die Annahme getroffen, dass die Linse ist ein symmetrisches Objekt ist.
Deshalb sind auch alle erzeugten Verzerrungen rotationssymmetrisch.
Diese Annahme kann getroffen werden, weil die optische Asche der Linse, an dem Schnittpunkt der einfallenden Lichtstrahlen, einen rechten Winkel zu dem optischen Sensor der Kamera bildet.
Die Koordinaten der Außenwelt werden mit den Großbuchstaben X,Y und die der Bildkoordinaten (Koordinaten auf dem Sensor der Kamera) mit den Kleinbuchstaben x,y bezeichnet.

$$X - X_0 = c1(\hat{x} - x_0) \quad Y - Y_0 = c2(\hat{y} - y_0) \tag{5.1}$$

Die Formeln 5.1 erlauben eine freie Definition der Ursprünge beider Koordinatensysteme (Außenwelt, Sensorfeld). Oft liegt der Ursprung der Bildkoordinaten bei (0,0) oben links im Bild.
Der Ursprung kann auch durch die Koordinate L beschrieben werden, die optimalerweise in der Mitte des Sensorfelds der Kamera (Bildzentrum) liegt.
c_1 und c_2 beschreiben Skalierungsfaktoren für die Richtungen von x und y. Diese liegen in diesem Beispiel beide bei eins. Die Formeln sehen anstatt von x und y ein $\hat{x}$ und $\hat{y}$ vor, um eine ideale koplanare Abbildung der Bildkoordinaten zu beschreiben, die durch Einflüsse der Herstellungstoleranzen der Optik nicht möglich ist. $\hat{x}$ und $\hat{y}$ sind Schätzungen, die aus den physikalischen Bildkoordinaten x und y abgeleitet werden müssen. Um die Linsenverzeichnung zu beschreiben wird eine Beziehung zwischen den physikalischen Koordinaten x,y und den idealen Koordinaten $\hat{x}$ $\hat{y}$ gesucht. $\hat{x}$ $\hat{y}$ sollen L entsprechen und x,y sollen dem wirklichen Mittelpunkt der Bildkoordinaten entsprechen. Die Abweichung zwischen x und $\hat{x}$ wird $\hat{x}_p$ genannt. Die Abweichung zwischen y und $\hat{y}$ wird äquivalent $\hat{y}_p$ genannt.
$\hat{x}_p$ und $\hat{y}_p$ sind davon Abhängig, ob die Linse mittig vor dem Sensorfeld der Kamera sitzt und ist auch von der Qualität des Herstellungsprozesses des Kamerasystems abhängig.
Linsenverzeichnung kann in zwei Bestandteile aufgeteilt werden. Diese sind die radiale Verzerrung und die tangentiale Verzerrung.
In Abbildung 5.2 kann man die Art der Verzerrung erkennen. Es handelt sich bei dem gewählten Weitwinkelobjektiv um eine Tonnenverzeichnung (starke radiale Verzerrung).
Da die radiale Linsenverzeichnung viel stärker ist, als die tangentiale Linsenverzeichnung, werden nur die radialen Verzerrungseigenschaften weiter betrachtet.

Beim Parametrisieren des Algorithmus 5 für die Entfernung der Verzerrung erhält man beste Ergebnisse, wenn man eine Parallelität zwischen den auf die Optik fallenden Strahlen und die der Strahlen die auf den Sensor treffen herstellt. Alle Abweichungen zu dieser Regel führen zu Linsenverzeichnungen.

5.3 Pseudocode

Der Algorithmus 5 modelliert die Verzerrung der Linse um L. Da L nicht perfekt in der Mitte des Sensors liegt (x_p y_p) wird halfWidth und halfHeight angepasst. Anschließend wird jede Koordinate K eines Z_{img} in den for-Schleifen aus einem kartesischen Koordinatensystem in eine Polardarstellung P überführt. Die Distanz von K zum korrigierten L wird berechnet (Zeile 9) und die Position von K mit einem Radius r und einem Winkel θ (Zeile 10,11) beschrieben. Der Radius wurde so modifiziert, dass die Entzerrung ermöglicht wird. Anschließend wird P zurück in eine kartesische Koordinaten überführt (Zeile 13,14), die als Quellkoordinate für die aktuellen Laufparameter der for-Schleifen dient.

Algorithm 5 Find Center Elements

1: $halfWidth \leftarrow imageWidth/2 - \hat{x}_p$ {L(x)}
2: $halfHeight \leftarrow imageHeight/2 - \hat{y}_p$ {L(y)}
3: $correctionRadius \leftarrow halfWidth$ {L(x) + halfWidth = imageWidth}
4: **for** y = 0 y < imageHeight y++ **do**
5: **for** x = 0 x < imageWidth x++ **do**
6: {x und y in in Relation zu L setzen}
7: $newX \leftarrow x - halfWidth$
8: $newY \leftarrow y - halfHeight$
9: $distance \leftarrow \sqrt{newX^2 + newY^2}$ {Distanz zu L berechnen}
10: $r \leftarrow distance/correctionRadius$ {r normieren}
11: $theta \leftarrow \arctan(r)/r$ {Winkel aus Koordinate}
12: {Quellkoordinate berechnen}
13: $sourceX \leftarrow halfWidth + theta * newX$
14: $sourceY \leftarrow halfHeight + theta * newY$
15: **end for**
16: **end for**

Da der Algorithmus 5 beschreibt eine Target-to-Sourcet Transformation. Die Richtung der Überführungsfunktion f, die durch 5 realisiert wurde, kann nicht durch Umstellen von f geändert werden. Deshalb wurde mit Matlab® numerisch eine Source-to-Target Transformationstabelle erstellt. Im Code beschreibt die Linsenkorrektur in Target-to-Source Form ein Pointer Array, dessen Einträge auf die Adressen der originalen Pixelwerte zeigen.

Durch diese Entscheidung entfällt ein ständiges Kopieren und Berechnen der korrigierten Bildinformationen.
Bei allen in dieser Arbeit vorgestellten Transformationen kommt diese Programmiertechnik zum Einsatz. Die Target-to-Source Transformation wird als Look Up Tabelle beim Programmstart erzeugt. Die Source-to-Target Transformation liegt als Header Datei vor. Die Linsenkorrektur wurde in LensCorrection.cpp realisiert.
Abbildung 5.1 zeigt das Ausgangsmaterial der Kamera. Dieses weist eine starke Weitwinkelverzerrung auf. Abbildung 5.2 stellt die Definitionslücken beim Source-to-Target Mapping dar.
In Abbildung 5.3 wird die Target To Source Variante der Linsenkorrektur dargestellt.

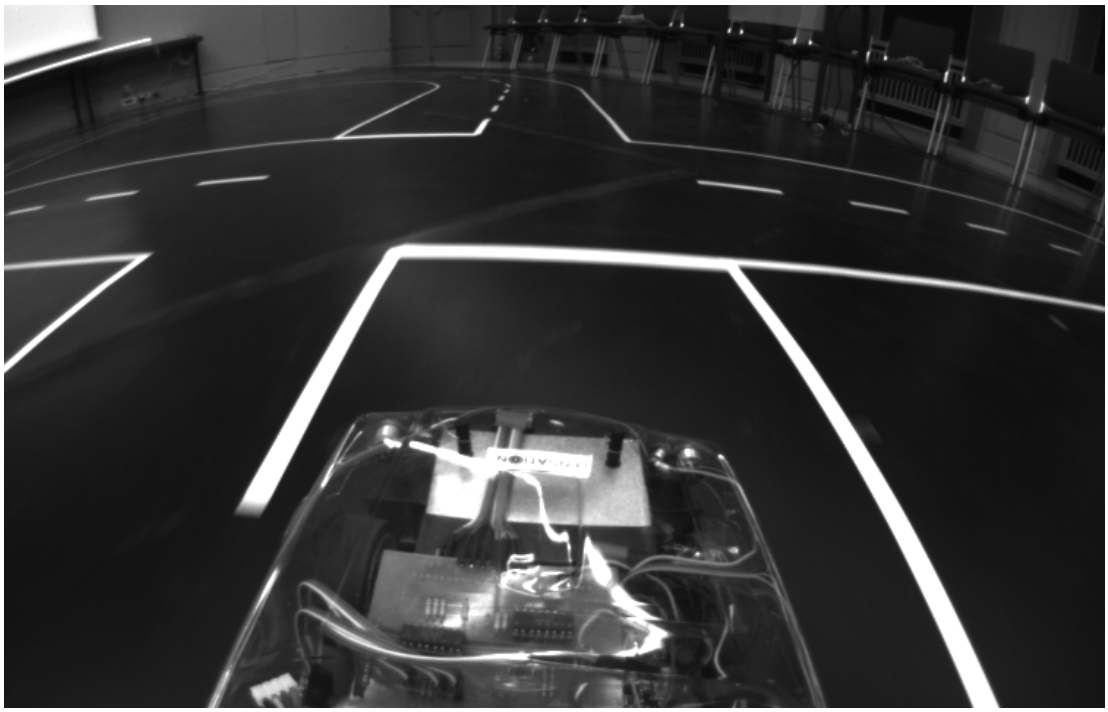

Abbildung 5.1: Die Verzerrung ist im original Bildmaterial deutlich erkennbar. Die Fahrbahnmarkierungen sollten später nicht mehr gewölbt sein, um weitere Transformationen zu ermöglichen.

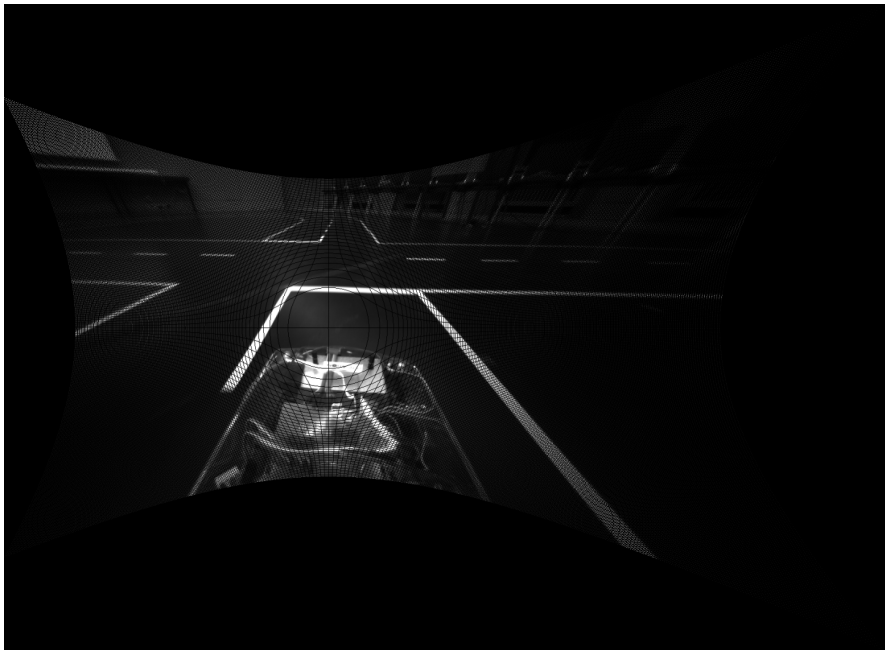

Abbildung 5.2: Bei der Source-to-Target Transformation sieht man die nicht behandelten Pixelpositionen. Der Bildinhalt ist deshalb deutlich dunkler als in Abbildung 5.3.

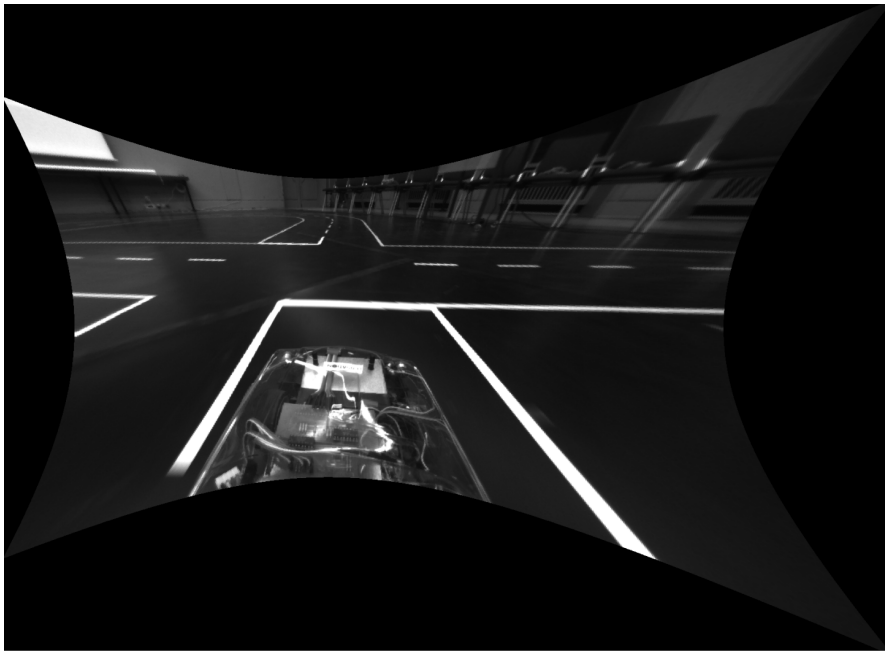

Abbildung 5.3: Bei der Target-to-Source Transformation treten keine nicht definierten Pixelwerte innerhalb des Bildinhaltes auf.

5.4 Realisierung des Source-to-Target Mapping

Mittels der fsolve Funktion Matlab® konnte die Lookup-Tabelle für die Source-to-Target Transformation berechnet werden. Die Formeln des Pseudocodes 5 lassen sich nicht umstellen. Deshalb war dieser Schritt notwendig.
Methode für die numerische Berechnung der Source-to-Target Transformation:

```
function [xVals, yVals] = STT_Nonlinear_Lens_Mapping()
    options = optimset('display','off');
    global x;
    global y;
    xVals = zeros(480,752);
    yVals = zeros(480,752);

    for yI = 0 : 479
        y = yI;
        for xI = 0 : 751
            x = xI;
            guess = [xI, yI];
            v = fsolve(@lens,guess,options);
            xVals(yI+1,xI+1) = int32(v(1)+0.5);
            yVals(yI+1,xI+1) = int32(v(2)+0.5);
        end;
    end;
end
```

Abbildung 5.4: Numerisches lösen der Formeln

In Abbildung 5.4 wird der Methode fsolve eine Funktion lens übergeben. Lens ist äquivalent zu dem Algorithmus 5. Fsolve benötigt einen initialen Startwert guess. Dieser hätte auf das Ergebnis des letzten fsolve Aufrufs gesetzt werden sollen, um die Berechnungsdauer zu senken. Ein globales x und y wird angelegt. Diese werden benötigt, um fsolve das Ergebnis vorzugeben, das numerisch berechnet werden soll. Die globalen Parameter x und y sind auch in der Funktion lens bekannt. Fsolve findet dann die Aufrufparameter x' und y' von lens, die x und y als Ergebnis haben. Für jedes $x, y \in Q_{img}$ wird das Ergebnis in xVals und yVals gespeichert. Diese Datenstrukturen enthalten am Ende die Source-to-Target Transformation. Die Datenstrukturen xVals und yVals wurden C++ konform exportiert und liegen im Quellcode als Header Datei vor. Die Arrays xVals und yVals werden in dieser Arbeit $lensXY_{STT_{x,y}}$ genannt.

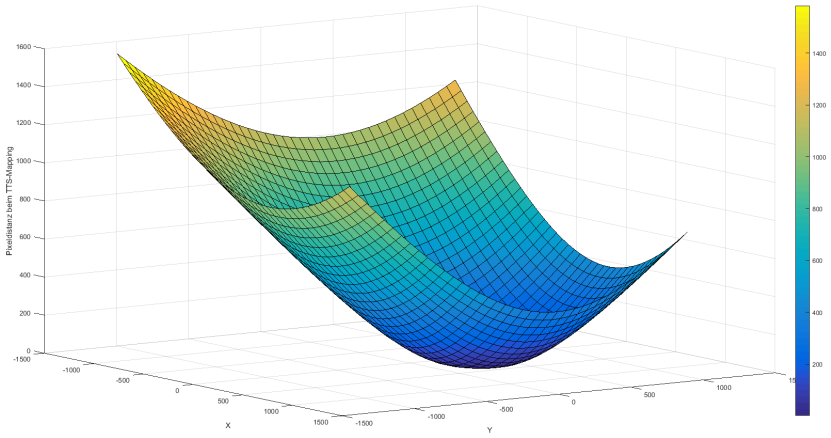

Abbildung 5.5: Surface Plot

Abbildung 5.6: Der Surface Plot dieser Berechnungen zeigt wie stark korrigiert werden musste, um die Verzerrung zu neutralisieren. Dargestellt wird die euklidische Distanz einer Zielkoordinate zu ihrer korrespondierenden Quellkoordinate.

6 Bird's Eye Transformation

Nach der Linsenkorrektur werden auch die Einflüsse der Perspektive entfernt. Diese Transformation kann nur auf Bildern durchgeführt werden deren Linsenverzeichnung kompensiert wurde.
Die Bird's Eye Transformation wurde als Lookup-Tabelle realisiert, deren Pointer auf die Pointer zu den Pixelwerten des originalen Bildinhaltes zeigen.
Für die Transformation wurden vier Pixelkoordinaten

$$Koordinaten_{Quelle} := \{\{xqrb, yqrb\}, \{xqlb, yqlb\}, \{xqrt, yqrt\}, \{xqlt, yqlt\}\}$$

des linsenkorrigierten Bildes gewählt. Durch iteratives Testen wurden die Positionen

$$Koordinaten_{Ziel} := \{\{xzrb, yzrb\}, \{xzlb, yzlb\}, \{xzrt, yzrt\}, \{xzlt, yzlt\}\}$$

gefunden, auf die $Koordinaten_{Quelle}$ abgebildet werden müssen, um die perspektivischen Einflüsse zu entfernen.
Hierfür müssen b Parameter berechnet werden, die mittels der GNU Scientific Library bestimmt wurden. Ein Gleichungssystem wird gelöst, welches $Koordinaten_{Quelle}$ und $Koordinaten_{Ziel}$ beinhaltet.
Die perspektivische Transformation wird durch folgende Formeln beschrieben (q stellt hierbei Quellkoordinaten dar und z Zielkoordinaten, die auf q abgebildet werden (Target-to-Source)):

$$x_q = \frac{b_{11}x_z + b_{12}y_z + b_{13}}{b_{31}x_z + b_{32}y_z + 1} \tag{6.1}$$

$$y_q = \frac{b_{21}x_z + b_{22}y_z + b_{23}}{b_{31}x_z + b_{32}y_z + 1} \tag{6.2}$$

Um das oben beschriebene Gleichungssystem zu erstellen, werden diese Formeln für x_q und y_q so umgestellt, dass die unbekannten Größen b von den bekannten gewählten Koordinaten separiert werden.

$$x_q = b_{11}x_z + b_{12}y_z + b_{13} - b_{31}x_z x_q - b_{32}y_z x_q \tag{6.3}$$

$$y_q = b_{21}x_z + b_{22}y_z + b_{23} - b_{31}x_z x_q - b_{32}y_z y_q \tag{6.4}$$

Setze man nun vier nicht kollineare Punktpaare (wie $Koordinaten_{Ziel}$ und $Koordinaten_{Quelle}$) in diese Formeln ein, erhält man acht Gleichungen.
Diese Formeln werden nun als Gleichungssystem dargestellt.

$$\begin{pmatrix} xzlb & yzlb & 1 & 0 & 0 & 0 & (-xzlb * xqlb) & (-yzlb * xqlb) \\ 0 & 0 & 0 & xzlb & yzlb & 1 & (-yzlb * xqlb) & (-yzlb * yqlb) \\ xzlt & yzlt & 1 & 0 & 0 & 0 & (-xzlt * xqlt) & (-yzlt * xqlt) \\ 0 & 0 & 0 & xzlt & yzlt & 1 & (-yzlt * xqlt) & (-yzlt * yqlt) \\ xzrb & yzrb & 1 & 0 & 0 & 0 & (-xzrb * xqrb) & (-yzrb * xqrb) \\ 0 & 0 & 0 & xzrb & yzrb & 1 & (-yzrb * xqrb) & (-yzrb * yqrb) \\ xzrt & yzrt & 1 & 0 & 0 & 0 & (-xzrt * xqrt) & (-yzrt * xqrt) \\ 0 & 0 & 0 & xzrt & yzrt & 1 & (-yzrt * xqrt) & (-yzrt * yqrt) \end{pmatrix} * \begin{pmatrix} b_{11} \\ b_{12} \\ b_{13} \\ b_{21} \\ b_{22} \\ b_{23} \\ b_{31} \\ b_{32} \end{pmatrix} = \begin{pmatrix} xqlb \\ yqlb \\ xqlt \\ yqlt \\ xqrb \\ yqrb \\ xqrt \\ yqrt \end{pmatrix} \tag{6.5}$$

Der Vektor der das Gleichungssystem löst (bVals_TTS[]) wird gesucht. Die linke Seite der Matrixmultiplikation wird in TTS_Matrix[] gespeichert. Nun wird in den Zeilen 150 bis 155 das Gleichungssystem mit der GNU Scientific Library gelöst.

```
134     int COEFFICIENT_COUNT = 8;
135     double b_coeff_TTS[] = { xqlb, yqlb, xqlt,yqlt, xqrb, yqrb, xqrt,yqrt};
136
137     double TTS_Matrix[] = {  xzlb, yzlb,    1,    0,    0,  0,  (-xzlb * xqlb), (-yzlb * xqlb),
138                                 0,    0,    0, xzlb, yzlb,  1,  (-yzlb * xqlb), (-yzlb * yqlb),
139
140                              xzlt, yzlt,    1,    0,    0,  0,  (-xzlt * xqlt), (-yzlt * xqlt),
141                                 0,    0,    0, xzlt, yzlt,  1,  (-yzlt * xqlt), (-yzlt * yqlt),
142
143                              xzrb, yzrb,    1,    0,    0,  0,  (-xzrb * xqrb), (-yzrb * xqrb),
144                                 0,    0,    0, xzrb, yzrb,  1,  (-yzrb * xqrb), (-yzrb * yqrb),
145
146                              xzrt, yzrt,    1,    0,    0,  0,  (-xzrt * xqrt), (-yzrt * xqrt),
147                                 0,    0,    0, xzrt, yzrt,  1,  (-yzrt * xqrt), (-yzrt * yqrt)};
148
149
150     gsl_matrix_view mTTS = gsl_matrix_view_array(TTS_Matrix, COEFFICIENT_COUNT, COEFFICIENT_COUNT);
151     gsl_vector_view bTTS = gsl_vector_view_array(b_coeff_TTS, COEFFICIENT_COUNT);
152     gsl_vector *TTS = gsl_vector_alloc (COEFFICIENT_COUNT);
153     gsl_permutation * pTTS = gsl_permutation_alloc (COEFFICIENT_COUNT);
154     gsl_linalg_LU_decomp (&mTTS.matrix, pTTS, &s);
155     gsl_linalg_LU_solve (&mTTS.matrix, pTTS, &bTTS.vector, TTS);
156
157     for(int i = 0; i<COEFFICIENT_COUNT;++i){
158         bVals_TTS[i] = gsl_vector_get(TTS,i);
159     }
160
161     gsl_permutation_free (pTTS);
162     gsl_vector_free (TTS);
```

Abbildung 6.1: Gleichungssystem

Die b Parameter werden in einem Array abgelegt (Zeile 158 der Abbildung 6.1).
Mit den Formeln 6.1 und 6.2 kann der Ursprung jeder Koordinate der transformierten Ansicht berechnet werden.
Der Quellcode für diesen Schritt liegt in Transformation.cpp. Nach einigen numerischen Problemen wurde ein Reverse-Lookup für die Source-to-Target Transformation präferiert. Hierbei wird versucht jede Koordinate Q aus dem originalen Bildinhalt der transformierten Ansicht zuzuordnen. Sobald für K durch 6.1 und 6.2 eine Quellkoordinate Q ermittelt wurde, kann Q auf K abgebildet werden. Ein Array $Trans_{STT_{x,y}}$ über $Q \rightarrow K$ beinhaltet diese Zuordnungen. Die Formeln für den Code stammen von Meisel (2012). Die Bird's Eye Transformation stellt einen Sonderfall der perspektivischen Transformation dar.

Für die Korrektheit der Transformation sollten, in einem späteren Softwarestand, mehrere Transformationstabellen angelegt werden. Bei Lastwechselreaktionen des Fahrzeugs ändert sich der Kamerawinkel und die Transformation leidet darunter. Deshalb könnte man mit den Informationen eines an der Kamera befestigten Kreiselinstruments zwischen mehreren Tabellen umschalten und so die Güte der Transformation erhöhen. Auch die Schräglage in Kurven könnte so kompensiert werden. In dieser transformierten Ansicht können rechte Winkel gefunden werden und weitere Annahmen getroffen werden. Die Regelung des Fahrzeugs wird sehr stark vereinfacht. Die so erstellte transformierte Ansicht wird Img_{trans} genannt. Abbildung 6.2 verdeutlicht diese Vorteile.

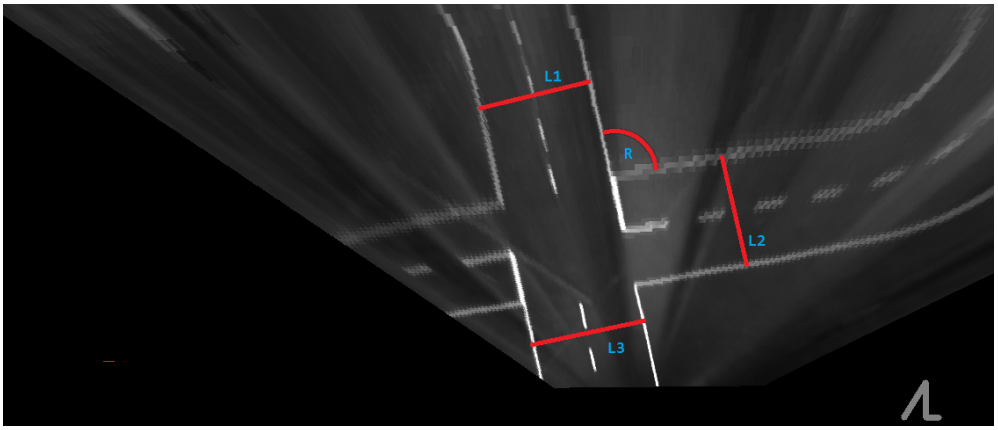

Abbildung 6.2: Transformierte Ansicht

In Abbildung 6.2 sieht man, dass die Linien L1, L2 und L3 ähnliche Längen aufweisen. Gut zu erkennen sind auch die rechten Winkel (Winkel R ist ein Beispiel dafür).

7 Regelbasierter Ansatz für die Erkennung der Mittellinie

Die Mittelliniendetektion wird Hummel genannt. Die Hummel funktioniert nur in Bird's Eye transformierten Bildmaterial (Img_{trans}).
Ein Element E der Mittellinie wird durch die Koordinaten MP beschrieben, deren Intensität über einem Schwellwert S liegen.
Des Weiteren muss für ein E gelten, dass auf zwei Kreisbahnen $r_{1,2}$, um das Zentrum C von E, keine Intensitäten KBI gefunden werden die größer sind als die von C.
Die Kreisbahnen Kreisbahnen $r_{1,2}$ werden mit dem Code aus Abbildung 7.1 erstellt. Hierfür werden Polarkoordinaten berechnet (siehe: Abschnitt 4.4). Die Zeilen 29 bis 35 hätten mit einem std::set realisiert werden sollen. Da diese Berechnung nur initial ausgeführt wird, interessiert der Aufwand nicht. Der Vektor scan entspricht r_1. Rescan entspricht r_2. Die Zeilen 22 bis 27 realisieren die Berechnung der Polarkoordinaten.

```
16 void Hummel::createStructuringElement(){
17
18     float start = M_PI;
19     float end = start+2*M_PI;
20
21     for(float angle = start; angle<end; angle+=0.002){
22         float sin_ = sin(angle);
23         float cos_ = cos(angle);
24         int xC = cos_*pointerLen+0.5;
25         int yC = sin_*pointerLen+0.5;
26         int xC2 = cos_*pointerLen2+0.5;
27         int yC2 = sin_*pointerLen2+0.5;
28
29         if (!(std::find(scan.begin(), scan.end(), pair<int,int>{xC,yC}) != scan.end())){
30             scan.push_back({xC,yC});
31         }
32
33         if (!(std::find(rescan.begin(), rescan.end(), pair<int,int>{xC2,yC2}) != rescan.end())){
34             rescan.push_back({xC2,yC2});
35         }
36
37     }
```

Abbildung 7.1: Berechnung von $r_{1,2}$

Abbildung 7.2 zeigt die Kantenlänge K der Mittellinienelemente.
Eine Variable fac beschriebt das Verhältnis von K zu den Radien von $r_{1,2}$.
Die Radien von $r_{1,2}$ entsprechen $\frac{K*fac}{2}$. r_2 ist ein Pixel länger als r_1.
Die Anzahl N der Koordinaten von MP eines E darf eine Obergrenze OG nicht überschreiten.

Abbildung 7.2: Zwei Mittellinienelemente $E1$ und $E2$ mit der Kantenlänge K in rot eingezeichnet

Die Beschreibung eines E ermöglichen eine translations- und rotationsinvariante Erkennung der Mittellinienelemente. Der Folgende Absatz beschreibt die Umsetzung dieser Idee.

Es wird der globaler Schwellwert S für Img_{trans} bestimmt. Anschließend wird für jede Koordinaten M aus Img_{trans} evaluiert, ob sie zu $MP_{candidates}$ gehört. Eine Koordinaten M muss als X-Wert ein Vielfaches von vier und als Y-Wert ein Vielfaches von drei besitzen. In diesen Schrittweiten Koordinaten zu inspizieren hat ähnliche Ergebnisse erzielt wie eine vollständige Analyse von Img_{trans} und senkt die Laufzeit.
Ein M gehört zu $MP_{candidates}$, wenn die Intensität von $M \geq S$ ist. Des Weiteren darf keine der Intensitäten KBI_M der Koordinaten auf den Kreisbahnen $r_{1,2}$ um M über der von M liegen.
Die Zeilen 7 bis 17 im Pseudocode 6 verdeutlichen diesen Schritt.
Die Radien $r_{1,2}$ sind größer als $\frac{K}{2}$.
Deshalb entstehen mehrere Detektionsergebnisse um ein C von einem möglichen E.
Um das C des jeweiligen E zu bestimmen, kommt ein sehr einfaches Clustering zum Einsatz.
$MP_{candidates}$ wird in die $Cluster_{1,..,n}$ aufgetrennt.
Eine Hash-Funktion h bildet die Koordinaten aus $MP_{candidates}$ auf vielfache Schrittweiten S_{width} in x und y Richtung ab. S_{width} entspricht K. Zu einem Cluster gehören alle Koordinaten aus $MP_{candidates}$, für die h das selbe Ergebnis erzeugt.

Um ein Zentrum ZC eines Clusters zu berechnen, werden die X Anteile der Koordinaten des jeweiligen Clusters summiert (X_{total}). Die Y Anteile werden entsprechend aufsummiert (Y_{total}). Die Anzahl der Koordinaten des Clusters wird NC genannt.

$$ZC(x) = \frac{X_{total}}{NC} \quad ZC(y) = \frac{Y_{total}}{NC}$$

Um die Zentren $ZC_{0,1,..,n}$ der Cluster werden zweidimensionale Suchräume $SR_{0,1,..,n}$ etabliert (siehe Abbildung 7.3 (mit B markiert)). Alle $SR_{0,1,..,n}$ haben eine Kantenlänge die K entspricht. Der Schnittpunkt der Diagonalen eines SR entspricht dem jeweiligen ZC.
Die Koordinatenserien $KSR_{0,1,..,n}$ werden durch die Pixelpositionen bestimmt, die durch ein korrespondierenden $SR_{0,1,..,n}$ verdeckt werden. Für jedes $KSR_{0,1,..,n}$ wird ein individueller Schwellwert $T_{0,1,..,n}$ bestimmt. Die Koordinaten eines KSR mit einer höheren Intensität als T bilden ein MP. Die Anzahl N der Koordinaten jedes MP muss unterhalb von OG liegen. Andernfalls wird das jeweilige MP verworfen. MP repräsentiert die Koordinaten eines E.

Abbildung 7.3: Darstellung der Vorschriften

In Abbildung 7.3 werden vier Postionen (P1, P2, P3 und P4) gezeigt, die Cs von Mittellinienelementen E darstellen sollen. Die graue Box B zeigt exemplarisch die oben genannten Suchräume SR. Die Situationen um P2, P3 und P4 sollen die Rotationsinvarianz des Verfahrens verdeutlichen.
Für jedes MP wird das Koordinatenpaar mit der größten Entfernung ermittelt. Dieses wurde in Abbildung 7.3 mit (D1,D2) gekennzeichnet.

Die Koordinatenpaare die wie D1 und D2 entstanden sind, werden nun zeilenweise sortiert. Jedes Mittellinienelement E erhält dadurch einen Startpunkt SP und einem Endpunkt EP. Die Mittellinienelemente $E1$ und $E2$ sind benachbart, wenn die euklidische Distanz eK von dem EP von $E1$ und dem SP von $E2 \leq K$ ist. eK wird analog zu Formel 3.2 berechnet. Da die Lücken in der Mittellinie die gleiche Länge wie K aufweisen, dient K als Maximaldistanz zweier benachbarter Elemente. Nun wird versucht zu jedem Element der Mittellinie E ein benachbartes E zu finden. Gerichtete Graphen aus diesen Nachbarschaften werden gebildet. Ist Element A mit Element B benachbart und B mit Element C können Nachbarschaften gebildet werden (A,B,C). Durch die Transitivität entstehen Graphen G. Jeder Graph G enthält die Koordinaten MP_G der MPs die für seine Bildung benötigt wurden.

In den nachfolgenden Abbildungen wurden alle erkannten Mittellinienelemente mit einer Box markiert. Diese Boxen entsprechen den Suchräumen $SR_{0,1,..,n}$.

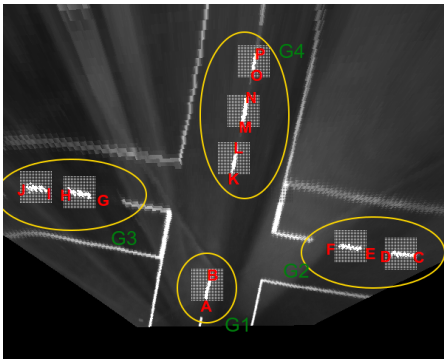

Abbildung 7.4: Darstellung der gefundenen Graphen
Das Verfahren konnte in diesem Beispiel vier Graphen (G1, G2, G3, G4) aus den Informationen der detektierten Mittellinienelemente extrahieren. Koordinate von B hat keine direkten Nachbarn und bildet daher einen Graphen mit einem Element. Betrachtet man G2,G3 und G4 erkennt man die oben beschriebene Transitivität.

Mittels eines modellbasierten Verfahrens wird die Anzahl der Koordinaten MP_G jedes Graphen G vergrößert. Als Modelle dienen Ausgleichsgeraden. Diese werden mit der GNU Scientific Library berechnet. In Kapitel 9.2 wird genauer beschrieben wie Koordinatenserien in Polynome überführt werden.
Für das modellbasierte Verfahren wird jeder gefundene Graph durch ein Polynom ersten Grades L approximiert. Wenn durch die Elemente MP_G, eines der Graphen G, eine Gerade L interpoliert wurde, kann geprüft werden welche Koordinaten aus anderen MPs in der Nähe von L liegen. Eine Koordinate CMP liegt in der Nähe zu L, wenn ihre Orthogonalprojektion (siehe: Formel 8.1) zu L eine kleinere Distanz aufweist, als ein vorher definiertes $\delta_{maxDist}$. Jedes CMP in der Nähe von L, geht in MP_G über. L wird für G neu berechnet. Dieses Verfahren wird solange wiederholt, bis keine weiteren Punkte zu MP_G des jeweiligen G hinzugefügt werden können, oder eine maximale Anzahl von Iterationen erreicht wird.

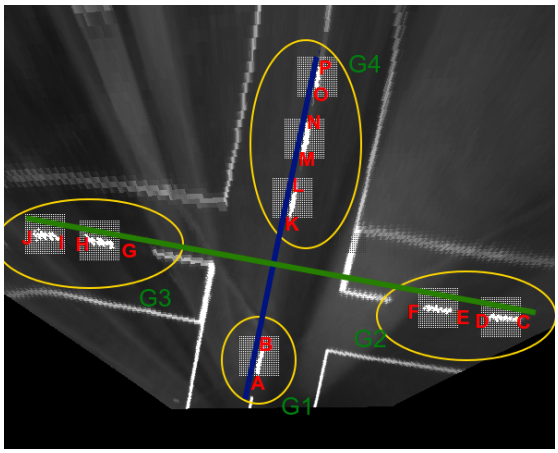

Abbildung 7.5: Ergebnis des modellbasierten Verfahrens

Anschließend kommt eine Heuristik zum Einsatz die entscheiden soll, welcher Graph die aktuelle Fahrbahn beschreibt. Diese Heuristik steckt noch in den Kinderschuhen und muss neu konzipiert werden. Der Quellcode der Mittelliniendetektion liegt in Hummel.cpp.

Bei der Programmierung wurde sehr stark auf die Maschinennähe des Quellcodes geachtet. Die Kreisbahnen $r_{1,2}$ werden deshalb nur initial berechnet und liegen als relative Offsets vor. Das Verfahren ist hochspezialisiert für diese Problemklasse der Mittelliniendetektion. Es ist sehr robust für das Auftreten von Leerstellen innerhalb der Mittellinie.

7.1 Pseudocode

Der nachfolgende Pseudocode abstrahiert die Hummel.

Algorithm 6 Find Center Elements

$r_{1,2} \leftarrow generateCircles$
$threshold \leftarrow otsu(Img_{trans})$
$MP_{candidates} \leftarrow List$
$Img_{trans\_less} \leftarrow reduceResolution(Img_{trans})$
$width \leftarrow Img_{trans\_less}.width$
$height \leftarrow Img_{trans\_less}.height$
for y = 0 y < height y++ **do**
  for x = 0 x < width x++ **do**
    $pixVal \leftarrow valueAt(x, y)$
    if pixVal $\geq$ threshold **then**
      $NothingWhiteAround\_M \leftarrow \forall P\ um\ (x, y)\ auf\ r_{1,2} : valueAt(P) < pixVal$
      if NothingWhiteAround_M **then**
        $MP_{candidates} \leftarrow (x, y)$
      end if
    end if
  end for
end for
$ZCs \leftarrow findCentersOfClusters(candidates)$ {Linear}
$SRs \leftarrow createSRs(ZCs)$ {$n * K^2$ Pixel}
$SRs\_filtered \leftarrow check\_OG(SRs)$
$ListOfMostDistantPairs \leftarrow findDistantPairs(SR\_filtered)$ {K^2}
$Graphs \leftarrow createNeighbourhoods(ListOfMostDistantPairs)$ {K^2}
$CreateModels \leftarrow ModelSearch(Graphs)$
return $BestModel$

7.2 Ergebnisse

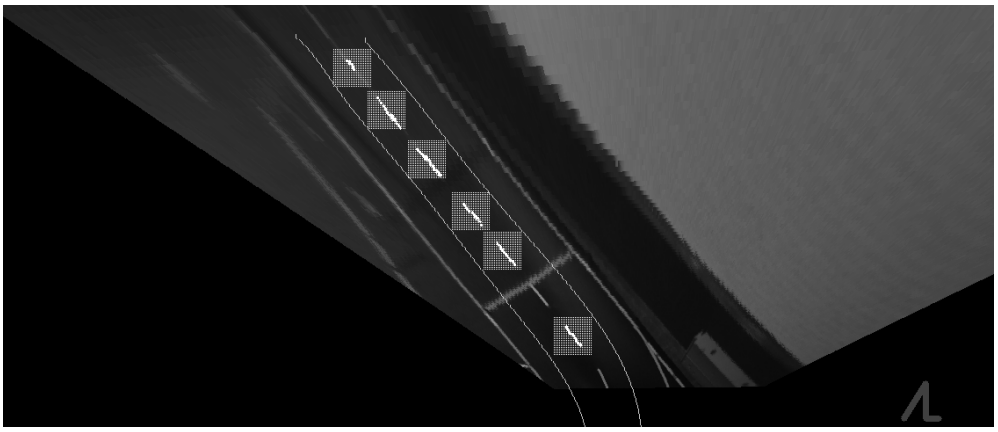

Abbildung 7.6: Bei der Startlinie konnte das Element nicht detektiert werden.

In Abbildung 7.6 liegt bei der Startlinie zwar ein Mittellinienelement, doch die Kriterien werden nicht erfüllt. Die Erkennungstiefe in diesem Beispiel liegt bei circa 3,2 Metern. Die Visualisierung der Ergebnisse soll lediglich die als Mittellinienelemente klassifizierten Koordinaten darstellen.

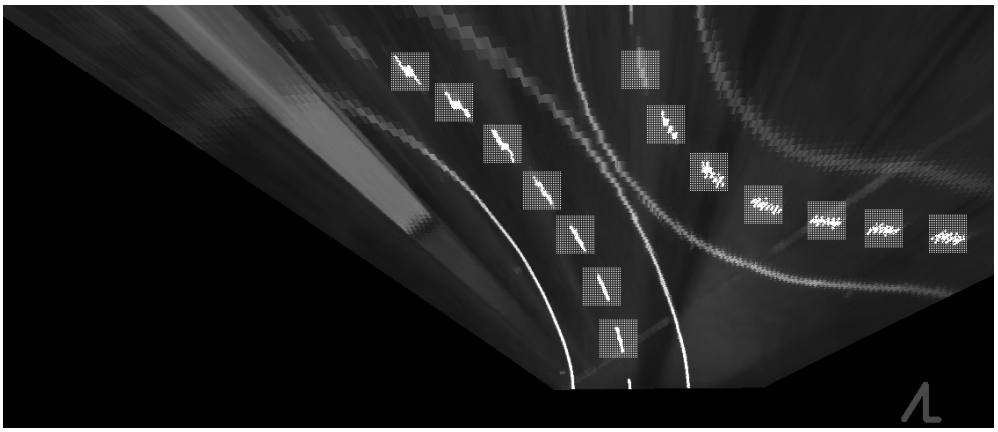

Abbildung 7.7: Zwei Straßen

Abbildung 7.7 ist ein Beispiel für zwei benachbarte Straßen. Hier ist es wichtig durch das modellbasierte Verfahren zwei Graphen zu finden, die diese repräsentieren und im Anschluss die richtige Wahl zu treffen.

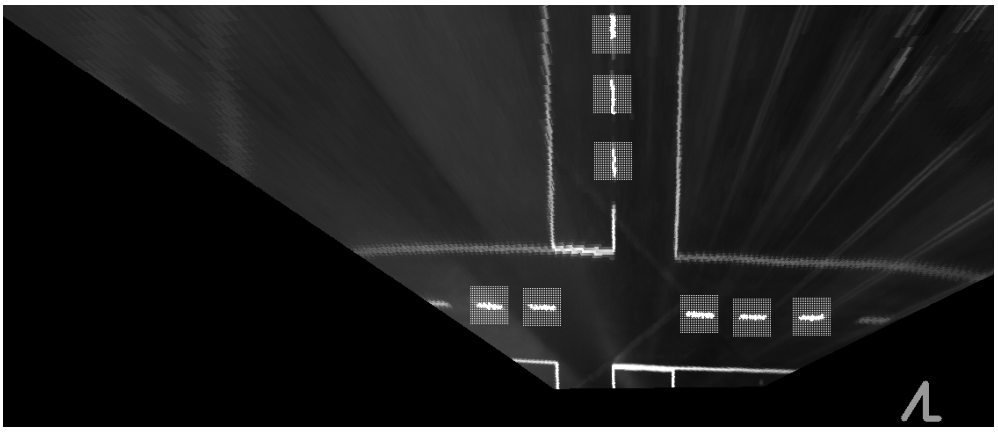

Abbildung 7.8: Kreuzung

Abbildung 7.8 zeigt die Kreuzungssituation aus einer anderen Ansicht als Abbildung 7.5.

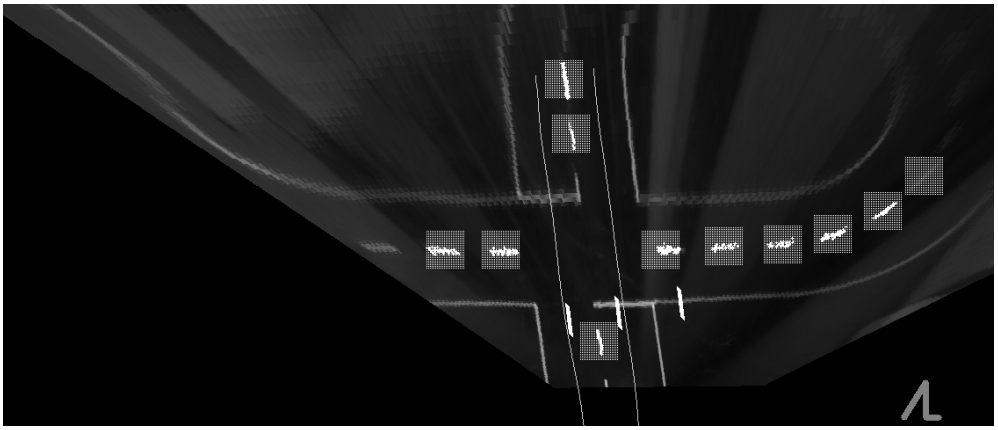

Abbildung 7.9: Kreuzung

Abbildung 7.9 ist im Teststreckenlabor der HAW entstanden.

8 Grundlegende Algorithmen

Dieses Kapitel nennt wichtige Algorithmen, die im nächsten Kapitel Verwendung finden. Die Intention dieses Kapitels besteht darin, die Algorithmen bekannt zu machen und deren Verwendungszweck zu erklären.

8.1 Ramer-Douglas-Peucker

Der Ramer-Douglas-Peucker Algorithmus reduziert die Anzahl von Punkten einer Kurve, die durch eine Serie von Koordinaten beschrieben wird. Man stelle sich vor eine Linie L wird in einer Teilmenge dieser Serie von Punkten gebildet. L wird durch den ersten und den letzten Punkt der Menge beschrieben. Nun wird der Punkt P $\{x3, y3\}$ gesucht, dessen Orthogonalprojektion OP den größten Abstand $dist$ zu L aufweist. L ist definiert durch zwei Koordinaten $\{x1, y1\}$ und $\{x2, y2\}$.
Die Funktion $f(x1, y1, x2, y2, x3, y3)$ soll OP beschreiben.

$$abdx := x2 - x1 \quad abdy := y2 - y1$$

$$len := \sqrt{abdx^2 + abdx^2}$$

$$OP := f(x1, y1, x2, y2, x3, y3) = |\frac{(x3 - x1)abdy - (y3 - y1)abdx}{len}| \tag{8.1}$$

Ist $dist$ kleiner als ein vordefiniertes ϵ_{dist}, wird P entfernt. Alle anderen Punkte der Teilmenge sind deshalb auch zu entfernen, da die Abstände ihrer Orthogonalprojektionen auch unterhalb von ϵ_{dist} liegen. Ist der Abstand der OP von P zu L größer als ϵ_{dist}, wird die Kurve in zwei neue Teile gespalten. Der erste Teil besteht aus dem ersten bis einschließlich des entferntesten Punkts. Der zweite Teil wird durch den entferntesten Punkt und der Restpunktmenge beschrieben. Der Algorithmus wird nun rekursiv auf beide neu entstandenen Teilmengen angewendet. Die Ergebnisse der rekursiven Aufrufe bilden die geglättete Kurve.

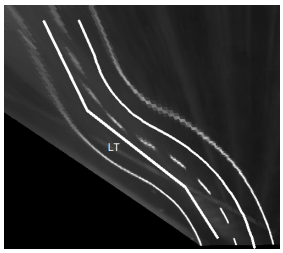

Abbildung 8.1: Die linke Trajektorie LT wurde zur Illustration mittels Ramer-Douglas-Peucker unter einem ϵ_{dist} von 23 vereinfacht. LT besteht nur noch aus vier Punkten.

Die ursprüngliche Implementation stammt von Dabrowski (2014) und wurde für dieses Projekt portiert. Da der Algorithmus ursprünglich rekursiv ist, wurde nach einer iterativen Lösung gesucht. Beim Code des Fahrzeugs sollten Rekursionen vermieden werden.

8.2 Bresenham Algorithmus

Jede der gefundenen Fahrbahnmarkierungen wird durch eine Reihe von Koordinaten repräsentiert. Diese Koordinaten liegen teilweise weit voneinander entfernt. Möchte man nun einen Zeilenweisen Zugriff ermöglichen, müssen weitere Punkte inferiert werden. Dies könnte man durch die Berechnung von Steigungsdreiecken realisieren. Hierdurch wären massiv viele Floating Point Operationen notwendig und man müsste sicher stellen, dass für jeden Punkt zwischen zwei zu interpolierenden Punkten ein Wert gefunden wurde. In dieser Arbeit wird deshalb der Bresenham Algorithmus verwendet (siehe Hans-Joachim Bungartz (2002)).
Von einer Startkoordiante aus soll zu einer Zielkoordinate eine Linie konstruiert werden, die durch die dazwischenliegenden Pixel beschrieben wird. Zuerst wird das δ_x und δ_x berechnet. δ_x ist die Differenz aus dem X-Wert der Endkoordinate und dem X-Wert der Startkoordinate. δ_y wird entsprechend δ_x berechnet.
Nun wird das Maximum δ_{max} von $\delta_{x,y}$ bestimmt. Das Minimum von $\delta_{x,y}$ nennen wir im Folgenden δ_{min}. Der Algorithmus benötigt noch eine Variable err die einen Fehlerwert beinhaltet. Von err wird abhängig gemacht, ob ein Schritt in die Richtung gemacht wird die δ_{min} bestimmt hat. Mit Richtung ist ein Schritt in X, oder Y von der Startkoordinate aus gemeint. Hierbei wird die Richtung dessen δ_{max} setzt bei jedem Durchlauf inkrementiert.

$$\text{Der initiale Fehlerwert von err ist: } err = \frac{\delta_{max}}{2}$$

Bei jeder Iteration wird vom Fehler δ_{min} subtrahiert. Ob ein Schritt in die Richtung gemacht wird, die δ_{min} setzt ist davon abhängig, ob $err < 0$ geworden ist. Ist dies der Fall wird err um δ_{max} erhöht. Das Ergebnis des Algorithmus sind die beiden übergebenen Koordinaten und die, wie oben beschrieben entstandenen, dazwischenliegenden Koordinaten.

8.3 Moving Average Filter

Der Moving Average Filter reduziert die Frequenzanteile eines Signals, das durch eine Serie von Werten beschrieben wird. Das Signal in diesem Kontext ist eine Serie von Koordinaten. Der Filter wird auf den X- und Y-Anteil der Koordinatenserie angewendet. Durch diese Glättung können anschließend markante Wendepunkte oder ähnliches registriert werden. Ein Fenster F wird über die Serie von Werten W geschoben. Bei jeder Verschiebung wird die Summe S aller Werte unterhalb von F gebildet und durch die Fensterbreite FN geteilt. Dieses Ergebnis bildet den geglätteten Wert. FN impliziert einen Datenverlust in den Randbereichen der zu glättenden Daten. Diese Operation anzuwenden ergibt nur Sinn, wenn die Abstände zwischen den Koordinaten äquidistant sind. Diese Äquidistanz sichert eine gleichbleibende Abtastrate zu. Im Quellcode beschriebt der Moving Average Filter einen Online-Algorithmus, um die Anzahl der Berechnungen zu minimieren.
W_{smooth} soll das Ergebnis beinhalten, das durch die Verwendung des Filters aus W entsteht. W enthält N Werte.

$$\forall_i \in \{0, ..(N - FN)\}\, W_{smooth_i} = \frac{\sum_{j=0}^{FN-1} W_{i+j}}{FN} \tag{8.2}$$

8.4 Taubin Fit - Ausgleichskreise

Beim Carolo-Cup gibt es keine Traktrizes. Eine Kurve kann deshalb durch Kreise repräsentiert werden. Taubin Fit (siehe: Taubin (1991)) wird dazu genutzt, eine Serie von Koordinaten in einen Ausgleichskreis zu überführen. Der Quellcode für die Berechnung der Ausgleichskreise stammt von Chernov (2012). Diese Methode gehört zu den algebraischen Ansätzen Ausgleichskreise zu berechnen. Die Idee die Fahrbahn als Kreise zu behandeln hat einen extrem positiven Einfluss auf die Möglichkeiten die Daten zu interpretieren. Mehrere Meter Fahrbahn können durch wenige Zentren und Radien akkurat beschrieben werden.

9 Aufwertung der extrahierten Fahrspurinformationen

Die Koordinatenserien der Erkennungsverfahren (Natter und Hummel) sollen gewisse Eigenschaften aufweisen.
Die Ergebnisse der Hummel H_{res} und der Natter $N_{links,rechts}$ werden deshalb aufbereitet.
Die Ergebnisse der Außenlinienerkennung werden entzerrt und in Bird's Eye Ansicht transformiert.
Die Koordinatenserien $N_{links,rechts}$ werden zuerst von der Linsenverzeichnung befreit. Die Datenstruktur $lensXY_{STT_{x,y}}$ der Linsenkorrektur wird hierfür verwendet.

$$N_{lens_{links}} := \forall K \in N_{links} K \rightarrow lensXY_{STT_{[K(x)],[K(y)]}}$$

$$N_{lens_{rechts}} := \forall K \in N_{rechts} K \rightarrow lensXY_{STT_{[K(x)],[K(y)]}}$$

Nachdem in $N_{lens_{links,rechts}}$ die entzerrten Koordinatenserien vorliegen, wird die Bird's Eye Transformation in Source-To-Target Richtung durchgeführt (siehe: Kapitel 6). Die Datenstruktur $Trans_{STT_{x,y}}$ wird hierfür verwendet.

$$N_{trans_{links}} := \forall K \in N_{lens_{links}} K \rightarrow Trans_{STT_{[K(x)][K(y)]}}$$

$$N_{trans_{rechts}} := \forall K \in N_{lens_{rechts}} K \rightarrow Trans_{STT_{[K(x)][K(y)]}}$$

Als Abkürzung für die Koordinatenserien wird $KS_{l,c,r}$ eingeführt.

$$l := N_{trans_{links}} \quad c := H_{res} \quad r := N_{trans_{rechts}}$$

Für zwei konsekutive Koordinaten K_i und K_{i+1} einer der Koordinatenserien KS_i aus $KS_{l,c,r}$ soll gelten, dass ihre euklidische Distanz $eK \leq \sqrt{2}$ ist und $K_i(y) \geq K_{i+1}(y)$. eK wird analog zu der Formel 3.2 berechnet.
Deshalb werden die KS_i von $KS_{l,c,r}$ sortiert und anschließend dem Bresenham Algorithmus (siehe: Abschnitt 8.2) übergeben. Da die Ergebnisse den Eindruck machten zu rauschen,

werden $KS_{l,c,r}$ mittels des Moving Average Filters geglättet. Die Fenstergröße FN wurde so gewählt, dass das Ergebnis stark genug geglättet wurde (siehe: Abschnitt 8.3).
Nachdem durch den Moving Average Filter irrelevante Frequenzanteile entfernt wurden, werden die einzelnen Koordinatenserien KS_i aus $KS_{l,c,r}$ in Kurvensegmente unterteilt.

Eine Fahrbahnmarkierung einer S-Kurve, wie in Abbildung 9.1, kann mit zwei Kreisen dargestellt werden. Der folgende Absatz beschreibt wie diese Kreise gefunden werden. Durch die Überführung von Koordinatenserien in Kreissegmente eröffnen sich neue Möglichkeiten.

Kurvensegmente werden durch die Wendepunkte innerhalb jedes KS_i bestimmt. Durch die Verwendung der numerischen zweiten Ableitung werden diese Wendepunkte gefunden. Hierbei interessiert lediglich das Vorzeichen des Wendepunkts (Formel 9.4 enthält keine Schrittweite h). Durch das Vorzeichen kann nun bestimmt werden, ob ein Kurvensegment eine Rechts- oder Linkskurve repräsentiert. In dieser Ansicht stellen Geraden Kurvensegmente mit einem unendlichen Radius dar. Bei jedem Vorzeichenwechsel wird von einem neuen Kurvensegment gesprochen.
Die zweite Ableitung wird beschrieben durch:

$$a(x1, x2, x3, y1) = \frac{2y1}{(x2 - x1)(x3 - x1)} \tag{9.1}$$

$$b(x1, x2, x3, y2) = \frac{2y2}{(x3 - x2)(x2 - x1)} \tag{9.2}$$

$$c(x1, x2, x3, y3) = \frac{2y3}{(x3 - x2)(x3 - x1)} \tag{9.3}$$

$$\begin{aligned} sign(K1, K2, K3) = a(K1(x), K2(x), K3(x), K1(y)) - \\ b(K1(x), K2(x), K3(x), K2(y)) + \\ c(K1(x), K2(x), K3(x), K3(y)) \end{aligned} \tag{9.4}$$

Listen $LS_{l,c,r}$ über die Vorzeichen werden erstellt.
Eine dieser Listen soll LS heißen.
Eine Koordinatenserie aus $KS_{l,c,r}$ soll KS heißen. KS besitzt N Koordinaten.

$$\forall_i \in \{0, ..(N-2)\}\, LS_i = sign(KS_i, KS_{i+1}, KS_{i+2})$$

Für jede Koordinatenserie KS_i aus $KS_{l,c,r}$ wird eine LS erstellt ($LS_{l,c,r}$). Anschließend wird jede Koordinatenserie KS_i bei den Indizes, die korrespondierenden Vorzeichenwechsel in $LS_{l,c,r}$ entsprechen, gespalten. Die Untermengen der Koordinatenserien KS_i werden $U_{l_{0,.n1},c_{0,.n2},r_{0,.n3}}$ bezeichnet.
Der Quellcode für die Ausgleichskreisberechnung von Chernov (2012) besteht aus den Klassen Data.cpp, Circle.cpp und CircleFitByTaubin.cpp. Für jede Koordinatenserienuntermenge U aus $U_{l_{0,.n1},c_{0,.n2},r_{0,.n3}}$ wird ein Data Objekt erstellt. Das Data Objekt wird der Methode CircleFitByTaubin aus CircleFitByTaubin.cpp übergeben. Durch diesen Aufruf wird ein Ausgleichskreis C berechnet. Die erste Koordinate von U wird Ke genannt. Kl entspricht der letzten Koordinate. Jedem C werden zwei Attribute $Start$ und End zugeordnet. $Start$ entspricht dem Winkel zwischen Ke und dem Zentrum von C. End ist der Winkel zwischen Kl und dem Zentrum von C.
Ein Kurvensegment wird durch einen Ausgleichskreis, einen Definitionsbereich und einer Rotationsrichtung Rot beschrieben. Eine Rotationsrichtung Rot wird für alle C gefunden. Diese wird durch das Vorzeichen des Wendepunkts aus $LS_{l,c,r}$ bestimmt.
Nun wird geprüft, ob Rot korrekt ermittelt wurde. Hierfür wird eine Koordinate Z aus den Kreisparametern berechnet, die in der Mitte des Definitionsbereiches liegen sollte. Der Definitionsbereich liegt zwischen $Start$ und End.
Der Winkel $\gamma = \frac{Start+End}{2}$ und das Zentrum von C wird dafür genutzt eine Koordinate $K_{between}$ zu berechnen. Aus γ und $K_{between}$ wird analog zu Pseudocode 4 $K_{between}$ erstellt. Wird eine Koordinate innerhalb der dem Kreis zugrundeliegenden Koordinatenserie U gefunden, deren euklidische Distanz (siehe: 3.2) zu $K_{between}$ größer ist als der Radius des Kreises, muss Rot geändert werden. Bei dieser Operation wird eine kleine Toleranz akzeptiert, bevor Änderungen getroffen werden, da die Kreisbahn lediglich eine Annäherung ist.
Ein Radius, der unterhalb des minimalen Kurvenradius der Carolo-Cup Strecke liegt, indiziert eine Fehlerkennung. Der Kreis wird dann verworfen.
Die approximierten Kreissegmente werden als Listen für jede der drei Fahrbahnmarkierung (l, c, r) gespeichert. Diese Listen werden $Circles_{l,r,c}$ genannt.

9.1 Erstellung einer Trajektorie

Da die abstrahierten Fahrbahninformationen $Circles_{l,r,c}$ nicht den geplanten Weg des Fahrzeuges beschreiben, sondern die einzelnen Fahrbahnmarkierungen (l, r, c) , müssen weitere Anpassungen getroffen werden. Durch das Modifizieren der Radien der Kreissegmente aus den Listen von $Circles_{l,r,c}$ werden die Trajektorien $T_{left,right}$ erzeugt. Aus jedem Kreissegment kann der linke und rechte befahrbare Weg gewonnen werden. Die Radien der Kreissegmente müssen entsprechend ihrer Produzenten $KS_{l,r,c}$ modifiziert werden.
Als Beispiel wird schematisch die Generierung einer Trajektorie aus Kreissegmenten, die aus der linken Außenlinienerkennung l hervor gegangen sind, behandelt. Bei Linkskurven ist die Rotationsrichtung $R = 0$ und bei Rechtskurven 1. Die Radien der Kreissegmente die Linkskurven abbilden müssen in diesem Fall vergrößert werden. Diese Modifikation entspricht $\frac{1}{4}$ der kompletten Fahrbahnbreite B für den linken befahrbaren Weg T_{left} und $\frac{3B}{4}$ für den rechten befahrbaren Weg T_{right}.
Bei Rechtskurven werden die Radien verkleinert. Für Kreissegmente die aus den anderen Linienerkennung entstanden sind (c,r), müssen die Radien jeweils entsprechend logisch so modifiziert werden, dass sie die korrekte Fahrbahnhälfte (Trajektorie) $T_{left,right}$ beschreiben. Aus jedem Kreissegment entstehen durch die Radienmodifikation zwei Kreissegmente, denen ihre zu abstrahierende Fahrbahnhälfte $T_{left,right}$ zugeordnet wird.

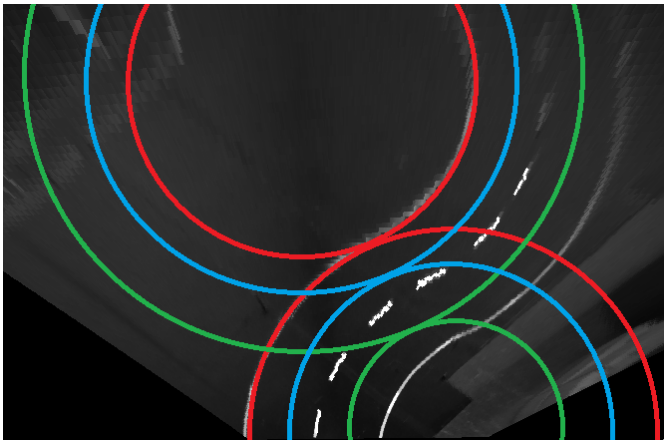

Abbildung 9.1: Darstellung der einzelnen Kreise die durch die linke Außenlinie entstehen (ohne Definitionsbereiche)

Die modifizierten Kreissegmente werden in Koordinatenserien überführt.
Die Überführung in Koordinatenserien findet nur innerhalb der Definitionsbereiche der Kreissegmente statt. Hierfür werden 32 Koordinaten aus dem Definitionsbereich eines Kreissegments berechnet. Die nicht dargestellten Koordinaten des Definitionsbereichs werden mit dem Bresenham Algorithmus (siehe: Abschnitt 8.2) inferiert.
Wurden von der Bilderkennung alle Linien detektiert, entstehen durch die Zugehörigkeit eines Kreissegments zu einem $T_{left,right}$ unter der Berücksichtigung seines Produzenten l, r, c jeweils drei Trajektorien für $T_{left,right}$. $T_{left_{l,r,c}}$ und $T_{right_{l,r,c}}$ enthalten die rücktransformierten Koordinatenserien.
$T_{left_{l,r,c}}$ und $T_{right_{l,r,c}}$ werden heuristisch in jeweils eine Trajektorie verschmolzen.
Um dies zu ermöglichen wird jede Koordinatenserie aus $T_{left_{l,r,c}}$ und $T_{right_{l,r,c}}$ zeilenweise vergleichbar gemacht. In einem definierten Zeilenabstand werden die X-Anteile aus $T_{left_{l,r,c}}$ und $T_{right_{l,r,c}}$ verfügbar gemacht. So können, bei drei erkannten Linien, pro Zeile ZN drei Werte leftX, centerX und rightX, der linken und rechten zu erstellenden Trajektorie, miteinander verglichen werden. Die Werte leftX, centerX und rightX werden aufsteigend ihrer X-Anteile sortiert und bilden das Tupel T. Ein $\epsilon_{xMaxDiff}$ wird definiert. Die Elemente von T die innerhalb der Toleranz von $\epsilon_{xMaxDiff}$ zu dem mittleren Element TC von T liegen, werden dafür genutzt ein X-Wert XA für die aktuelle Zeile zu bestimmen. Liegt ein äußeres Element TLR von T nicht innerhalb des Toleranzbereiches $\epsilon_{xMaxDiff}$ zu TC, wird TLR nicht für die Mittelwertbildung von XA genutzt. Die Koordinate aus XA und ZN wird in dem entsprechenden T_{right}, oder T_{left} gespeichert.
In dem nächsten Kapitel 9.2 wird beschrieben wie T_{right} und T_{left} durch Polynome abstrahiert werden.

9.2 Repräsentation der Trajektorien durch Polynome

Ein Polynom P der Form $f(y) = y^i a_i + y^{i-1} a_{i-1} +, ..., +y^0 a_0$ bildet ein y auf ein x ab.

Deshalb müssen alle Koordinatenserien injektiv sein, bevor sie als Polynome dargestellt werden können (keine doppelt definierten Zeilen). Die Injektivität der Koordinatenserien T_{right} und T_{left} wird erzeugt. Zwei Datenstrukturen die keine Duplikate aufnehmen werden angelegt $hm_{left,right}$. Jede Koordinatenserie KS von $T_{left,right}$ wird durchlaufen. Das y einer Koordinate K aus KS wird, wenn es nicht in dem korrespondierenden $hm_{left,right}$ enthalten ist, in $hm_{left,right}$ eingefügt. Beinhaltet das entsprechende $hm_{left,right}$ bereits y, wird die Koordinate K aus KS verworfen.

Aus historischen Gründen werden die Ergebnisse der Bildverarbeitung als Polynom für die Längs- und Querregelung verfügbar gemacht. Diese Abstraktion bringt Vorteile mit sich. Der Fahrbahnverlauf kann nun zum Beispiel integriert und differenziert werden. Hierdurch kann man die Längs- und Querregelung vereinfachen. In der Software können durch Polynome X-Werte für Bildzeilen Y aus Img_{trans} berechnet werden. Bei der Überführung der Koordinaten in Polynome wurde auf die Gnu Scientific Library zurückgegriffen. Beim Anlegen und beim Zugriff auf Polynome werden viele Potenzen gebildet, deshalb liegen diese Potenzen als Tabelle vor. Da der Fahrbahnverlauf sehr genau dargestellt werden sollte, kommen Polynome zehnten Grades zum Einsatz.

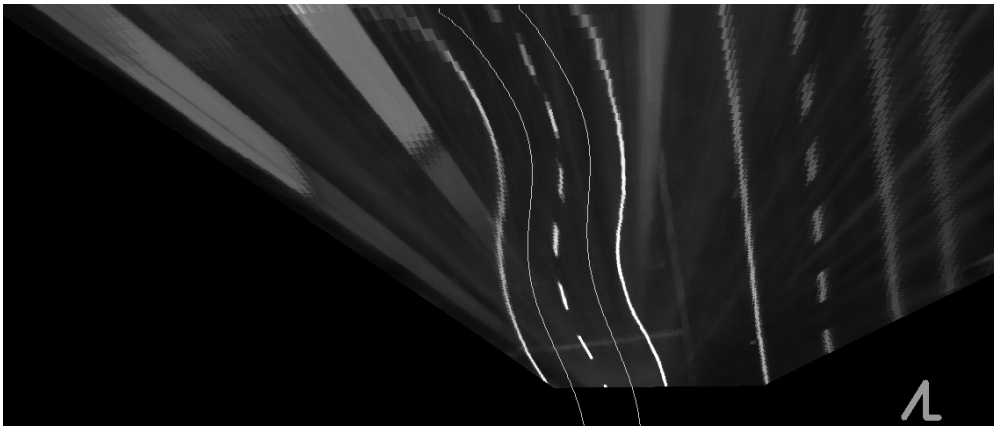

Abbildung 9.2: Darstellung der linken und rechten Trajektorie durch Polynome

```
263 void Polynom::calcPolynom(vp &points){
264     if((int)points.size()<degree){
265         valid = false;
266         return;
267     }
268     int obs = points.size();
269     gsl_multifit_linear_workspace *ws;
270     gsl_matrix *cov, *X;
271     gsl_vector *y, *c;
272     double chisq;
273 
274     int i, j;
275 
276     X = gsl_matrix_alloc(obs, degree);
277     y = gsl_vector_alloc(obs);
278     c = gsl_vector_alloc(degree);
279     cov = gsl_matrix_alloc(degree, degree);
280 
281     for(i=0; i < obs; ++i){
282         gsl_matrix_set(X, i, 0, 1.0);
283         for(j=0; j < degree; j++){
284             gsl_matrix_set(X, i, j, PowTable::pow((int)(points[i].second+0.5), j));
285         }
286         gsl_vector_set(y, i, points[i].first);
287     }
288 
289     ws = gsl_multifit_linear_alloc(obs, degree);
290     gsl_multifit_linear(X, y, c, cov, &chisq, ws);
291 
292     // store result
293     for(i=0; i < degree; ++i)
294     {
295         mdPolynom[i] = gsl_vector_get(c, i);
296     }
297 
298     gsl_multifit_linear_free(ws);
299     gsl_matrix_free(X);
300     gsl_matrix_free(cov);
301     gsl_vector_free(y);
302     gsl_vector_free(c);
303 
304 }
```

Abbildung 9.3: Polynominterpolation

Abbildung 9.3 zeigt den Quellcode für die Polynominterpolation mittels der GNU Scientific Library. Ein Vektor vp von Koordinaten wird an die Methode übergeben. Vorher wurde der Grad des Polynoms (degree) festgelegt. Die Methode gsl_multifit_linear in Zeile 290 berechnet die am besten passenden Koeffizienten c, für ein Modell aus Beobachtungen y und Prädiktorvariablen X. Das Modell hat die Form $y = Xc$. Die Beobachtungen sind in diesem Fall die X-Werte der zu interpolierenden Koordinaten. Die Prädiktorvariablen X werden in den Zeilen 281 bis 287 in einer Matrix abgelegt. Für die Berechnung wird die vorher angelegte Datenstruktur gsl_multifit_linear_workspace benutzt (siehe: GNU-Scientific-Library (2016)). Der Code entspricht dem Verwendungsbeispiel aus Rosettacode (2016)
Nachdem die Library die Ausgleichsrechnung durchgeführt hat, liegen die Polynomkoeffizienten in mdPolynom vor (Zeile 295).
Anschließend können die Werte eines Polynoms berechnet werden. Abbildung 9.4 zeigt die Berechnung eines X-Werts für eine Zeile Y mit Polynom.cpp.

```
251 float Polynom::solveXHighPrecision(int y){
252     double x = mdPolynom[0];
253     for(int i = degree-1; i>0; --i){
254         x+= mdPolynom[i] * PowTable::pow(y,i);
255     }
256     return x;
257 }
```

Abbildung 9.4: f(y) = x

Polynome zehnten Grades fangen sehr leicht an ein komisches Verhalten, in nicht definierten Bereichen der Koordinatenserien, zu entwickeln. Die Koordinatenserien T_{right} und T_{left} werden stabilisiert bevor sie in die Polynome $P_{left,right}$ überführt werden. Nicht definierte Bildzeilen von $T_{left,right}$ aus Img_{trans} werden hierdurch inferiert. Koordinatenserien werden nur unterhalb der Bildzeile Y_{max} stabilisiert. Dieser Wert wurde durch Testen gefunden. Koordinaten die durch eine Translation in Y-Richtung um δ_Y unterhalb dieser Zeile liegen werden erzeugt.

1: $ContainerForCoordinates \leftarrow List$
2: **for** $\forall Points \in toBeStabilized$ **do**
3: $\quad newY \leftarrow P(y) + \delta_Y$
4: $\quad$ **if** $P(y) > Y_{max}$ **then**
5: $\quad\quad ContainerForCoordinates \leftarrow insert\ \{P(x), newY\}$
6: $\quad$ **end if**
7: **end for**
8: $P2 \leftarrow calculatePolynomDegree2(ContainerForCoordinates)$

Jetzt werden Koordinaten mit dem Polynom $P2$ bestimmt. P2 ist ein Polynom zweiten Grades und wird aus den Koordinaten die in den Zeilen 2 bis 6 entstanden sind gebildet. Wenn T_{right} zum Beispiel Koordinaten mit Y-Werten zwischen 350 und 200 enthält, wird mit $P2$ eine Koordinatenserie KP_{right} generiert, deren Y-Werte zwischen der untersten Zeile von Img_{trans} und 350 liegen. Die Polynome $P_{left,right}$ werden aus $KP_{left,right}$ und dem jeweiligen $T_{left,right}$ approximiert. Hierdurch gibt es Punkte aus $KP_{left,right}$ die das Polynom $P_{left,right}$, in nicht definierten Bereichen, dazu zwingen stabil zu bleiben.

Es gibt Methoden für die Translation und Rotation von Polynomen. Diese Operationen erstellen lediglich eine Matrix die für ein Mapping benutzt wird. Die Koeffizienten des Polynoms werden hierfür nicht manipuliert. Die Formeln für das Rotieren und Verschieben stammen von Meisel (2012).
Der Quellcode zeigt was passieren muss, um die Matrix zu erstellen (rTMatrix).

```
616 //rotates counterclokwise
617 void Polynom::generateRotationAndTranslationMatrix(float angleInRadiansClockise, float drivenDistanceInMM){
618     useMatrix = true;
619     auto const centerOfRotation = calcCoordinateAtArcLength(drivenDistanceInMM);
620
621     // first calc coor at front wheels of the car (where drivable way starts)
622
623     //auto const frontOfCar = isRight?lastStabilizationRight:lastStabilizationLeft;
624
625     // now calc xDiff and yDiff
626
627     offXToOriginOfCoorSystem = centerOfRotation.first;
628     offYToOriginOfCoorSystem = centerOfRotation.second;
629
630     float tmp[3][3] = {{0,0,0},{0,0,0},{0,0,0}};
631
632     float transCenter[3][3] = {{1,0,offXToOriginOfCoorSystem},{0,1,offYToOriginOfCoorSystem},{0,0,1}};
633
634     float cosTheta = cos(angleInRadiansClockise);
635     float sinTheta = sin(angleInRadiansClockise);
636
637     float rot[3][3] = {{cosTheta,-sinTheta,0},{sinTheta,cosTheta,0},{0,0,1}};
638
639     float transBack[3][3] = {{1,0,-offXToOriginOfCoorSystem},{0,1,-offYToOriginOfCoorSystem},{0,0,1}};
640
641     float sum;
642     //transCenter x rot -> combo1 in tmp
643     for(int row = 0; row<3; ++row){
644         for(int col = 0; col<3; ++col){
645             sum=0;
646             for (int inner = 0; inner < 3; ++inner) {
647                 sum += transCenter[row][inner] * rot[inner][col];
648             }
649             tmp[row][col] = sum;
650         }
651     }
652
653     //tmp x transBack -> combo2
654     for(int row = 0; row<3; ++row){
655         for(int col = 0; col<3; ++col){
656             sum=0;
657             for (int inner = 0; inner < 3; ++inner) {
658                 sum += tmp[row][inner] * transBack[inner][col];
659             }
660             rTMatrix[row][col] = sum;
661         }
662     }
663 }
```

Abbildung 9.5: Quellcode zum Erstellen der kombinierten Matrix

Die Zeile in der die Frontachse des Fahrzeugs in Img_{trans} liegt wird ZFF genannt. In Zeile 619 in Abbildung 9.5 wird die Koordinaten COR des Polynoms P bestimmt, deren Distanz der Bogenlänge von P ausgehend ZFF entspricht. Zuerst findet eine Translation von COR zum Ursprung der Koordinatensystems (0,0) statt (Zeilen 627,628)(siehe: $transCenter$). Für den Winkel angleInRadiansClockwise wird ein Cosinus und Sinus Wert berechnet. Diese sind wichtig für die Rotation um den Ursprung. Zeile 637 definiert die Rotationsmatrix rot. Die Matrizen $transCenter$ und rot werden kombiniert zu einer Matrix rT (Zeile 645 - 662).

Um die Rotation und Translation durchzuführen wurde die folgende Methode entwickelt.

```
586 void Polynom::useRotationAndTranslationMatrix(float &x, float &y){
587     float sum;
588     float outVec[3] = {0,0,0};
589     float inVec[3] = {x,y,1};
590     //transCenter x rot -> combo1 in tmp
591     for(int row = 0; row<3; ++row){
592         sum = 0;
593         for (int inner = 0; inner < 3; ++inner) {
594             sum += rTMatrix[row][inner] * inVec[inner];
595         }
596         outVec[row] = sum;
597     }
598
599     if(isRight){
600         translationX = lastStabilizationRight.first-offXToOriginOfCoorSystem;
601         translationY = lastStabilizationRight.second-offYToOriginOfCoorSystem;
602     }else{
603         translationX = lastStabilizationLeft.first-offXToOriginOfCoorSystem;
604         translationY = lastStabilizationLeft.second-offYToOriginOfCoorSystem;
605     }
606
607     x = outVec[0]+translationX;
608     y = outVec[1]+translationY;
609 }
```

Abbildung 9.6: $P \rightarrow P_{rt}$

Abbildung 9.6 enthält zwei Variablen (translationX,translationX). Diese beinhalten die Koordinaten $\{ZZT, PL.solveX(ZZT)\}$ des zuletzt interpolierten Polynoms $PLleft, right$ (Codezeile: 600-604). Diese Variablen werden für die Translation von COR zur Fahrzeugposition genutzt. isRight ist bei einem Polynom das die rechte Trajektorie beschreibt true. Hiervon wird abhängig gemacht, zu welcher Koordinate vor dem Fahrzeug COR verschoben wird. In den Zeilen 591 bis 597 wird die Matrixmultiplikation vorgenommen. Das Ergebnis der Matrixmultiplikation wird zur Fahrzeugposition verschoben (Zeilen 607,608).

Da das Polynom nicht wirklich modifiziert wird muss etwas Aufwand betreiben werden, um an den Wert zu kommen, der nach der Rotation und Verschiebung auf eine bestimmte Zeile Z_{rt} abbildet. Dieses Verfahren wurde mit einer binären Suche realisiert. Es ist deshalb nicht notwendig Polynome durch die Modifikation ihrer Faktoren zu rotieren, oder zu verschieben. Eine obere Grenze von acht Iterationen wurde für die binäre Suche festgelegt.

10 Inertiales Navigationssystem

Es sind Situationen aufgetreten, die durch eine komplett kamerabasierte Längs- und Querregelung nicht behandelt werden konnten. Es ist von Vorteil bis zu vier Meter Erkennungstiefe des Fahrbahnverlaufes zu erreichen, aber eine alleinige Nutzung dieser Daten für den aktuellen Regelzyklus ist sehr verschwenderisch. Die Überlegung bestand nun darin, für eine bestimmte Strecke S ohne Kamerabild fahren zu können. S entspricht der Trajektorielänge die durch ein Polynom repräsentiert wird. Die Umsetzung dieser Idee sollte keinen großen Rechenaufwand besitzen und schnell umgesetzt werden. Deshalb wurde ein sehr einfaches Verfahren hierfür entwickelt. Wird ein Frame ausgewertet, sind seine Fahrbahninformationen beim Setzen des Lenkwinkels schon veraltet, da die Prozesskette der Bildverarbeitung und die Aktorik eine gewisse Latenz besitzen. Der Lenkwinkel stimmt deshalb ohne eine Kompensation nicht für die aktuelle Fahrzeugposition. Die Schwere dieses Fehlers hängt von der Framerate, der Trägheit der Aktorik und der Geschwindigkeit des Fahrzeugs ab. Der Fehler der durch diese Eigenbewegung entsteht muss kompensiert werden. Beim Carolo-Cup ist mindestens eine Fahrbahnmarkierung vorhanden. Diese muss aber nicht unbedingt im Sichtfeld der Kamera liegen. Ist dies der Fall liegen ohne ein inertiales Navigationssystem keine Streckeninformationen für den jeweiligen Regelzyklus vor. Die Regelung des Fahrzeuges muss deshalb unabhängig von der Bilderarbeitung sein. Durch diese Unabhängigkeit erreicht man ein definiertes Fahrzeugverhalten bei unterschiedlichen Geschwindigkeiten, da äquidistant von der Regelung Fahrbahninformationen ausgewertet werden können. Bei Tests hat sich gezeigt, dass mit dem System bei einer Geschwindigkeit von 1,5 $^{m}/_{s}$ 2 $^{Frames}/_{s}$ für die korrekte Regelung des Fahrzeugs ausreichend sind. Das System nutzt die Daten der Hallsensorik des Fahrzeugsmotors für die Bestimmung der zurückgelegten Distanz δ_S und ein Kreiselinstrument für die Bestimmung der momentanen Absolutausrichtung $\alpha_{absolute}$ des Fahrzeugs. Nachdem die Prozesskette der Bildverarbeitung durchlaufen wurde, wird das Ergebnis mit der aktuell zurückgelegten Distanz D_{now} und der momentanen Absolutausrichtung $\alpha_{absolute}$ des Fahrzeugs von dem System gespeichert. Das Ergebnis der Bildverarbeitung sind die Polynome $P_{left,right}$ (siehe: 9.2). Auch eine Information darüber wann das mit diesen Werten gespeicherte Polynom seine Gültigkeit verliert ist von

Interesse. Das Polynom P ist nicht mehr gültig und wird verworfen, sobald es keine aktuellen Fahrbahninformationen repräsentiert. Wird durch die Regelung nach einem Wert verlangt, werden die Rotations- und Translationsmatrizen der gespeicherten Polynome neu berechnet, um ihre Informationen für die aktuelle Fahrzeugposition zu modifizieren. Hierbei wird das $\delta_{Rotation}$ der Absolutausrichtungen α zwischen Aufnahme und Abfrage des jeweiligen Polynoms gebildet. Im Anschluss wird das δ_S zwischen Aufnahme und Abfrage des Polynoms berechnet und jedes Polynom an dem entsprechenden Wegpunkt COR seiner Bogenlänge um $\delta_{Rotation}$ verkippt. Der Wegpunkt COR beschreibt die theoretische momentane Position des Fahrzeugs auf der Fahrbahn. Im Anschluss werden die Polynome an diesem Punkt vor die Fahrzeugposition verschoben, um neue valide Fahrbahninformationen zu erzeugen. Da durch dieses Verfahren mehrere Ergebnisse der Bildverarbeitung die momentane Position des Fahrzeuges beschreiben, können Mehrheitsentscheidungen über zum Beispiel den Lenkwinkel getroffen werden. So werden Fehlerkennungen unkritischer. Ein wichtiger Punkt dieses Systems ist, dass keine aufwendige Selbstlokalisation durchgeführt werden muss und fast keine Rechenzeit für einen extremen Mehrgewinn an Information anfällt. In dem Kapitel 9.2 wird die Rotation und Translation von Polynome näher beschrieben.

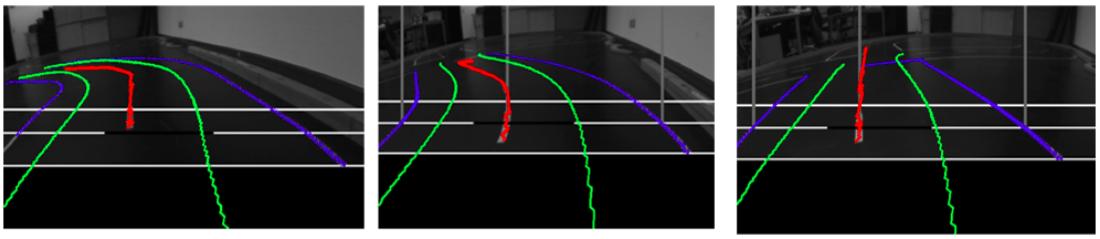

Abbildung 10.1: Visualisierung durch gespeicherte Sensorwerte
Nach der Auswertung des linken Frames wurden die Trajektorien (grün) durch das inertiale Navigationssystem erzeugt und sind unabhängig von der Bildverarbeitung. Weitere Ergebnisse der Bilderkennung wurden nicht in das System gespeist. Die Bilderkennung läuft weiter, es wurde dennoch ein Ausfall der Kamera simuliert. Die Information des Systems wurde aus der Bird's Eye Ansicht für eine humanere Darstellung zurück in eine Ansicht mit Perspektive transformiert.

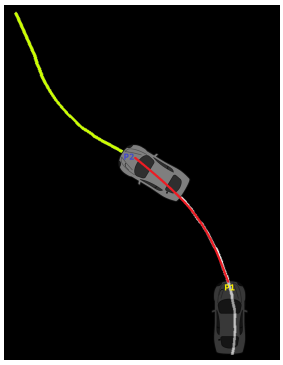

Abbildung 10.2: Bestimmung der Bogenlänge
Die Trajektorie wurde von dem System mit der Absolutausrichtung des Fahrzeugs und dem aktuellen Kilometerstand bei Poistion P1 gespeichert. Das Fahrzeug befindet sich im Bild immer an Punkt P1. Nun fährt das Fahrzeug die in rot markierte Strecke ab. Das System sucht auf dem gespeicherten Polynom die Koordinate, die der gefahrenen Distanz entspricht. Hierfür berechnet es die Bogenlänge von P1 bis zum Distanz-Delta woraus P2 hervorgeht.

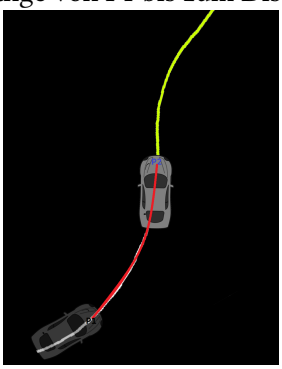

Abbildung 10.3: Rotation
Das gespeicherte Polynom wird an Punkt P2 um die Differenz der Absolutausrichtung des Fahrzeugs zwischen Aufnahme in das System und der Anforderung verkippt.

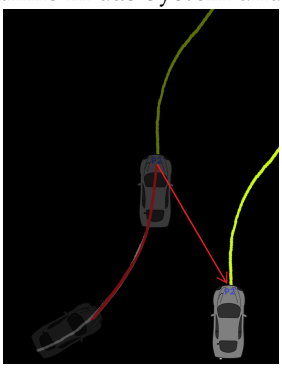

Abbildung 10.4: Translation
Da die Regelung des Fahrzeuges davon ausgeht, dass das Fahrzeug immer an einem bestimmten Punkt im Bild positioniert ist, wird das rotierte Polynom um den Vektor aus dem ursprünglichen Punkt P1 aus Grafik 10.2 und P2 verschoben.

11 Rotationsinvariante Detektion von Kreuzungen

Im Wettbewerb sehen Kreuzungen immer gleich aus. Hierdurch kann eine Regel für die Detektion formuliert werden. Betrachtet man Abbildung 11.1 fällt auf, dass die rechte Außenlinie T-förmig ist. Das Ziel ist es aus jedem Winkel und großer Distanz dieses T zu detektieren.

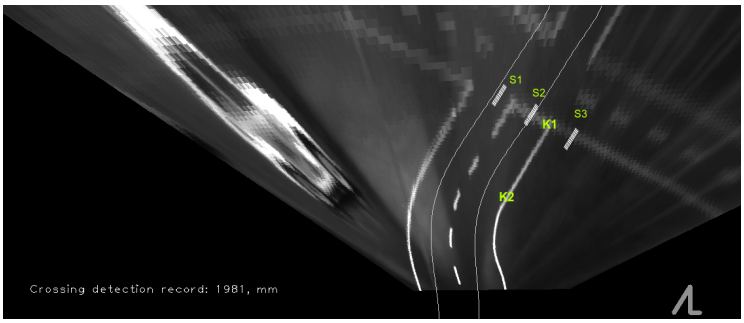

Abbildung 11.1: Transformierte Ansicht der Kreuzung

Die Aufruf der Natter für die rechte Außenlinie erzeugt durch das Fehlverhalten, das in Abbildung 4.7 gezeigt wird, die notwendigen Informationen dafür.
Wie in Kapitel 9 beschrieben wurde, wird die Koordinatenserie $N_{trans_{rechts}}$ erstellt. $N_{trans_{rechts}}$ wird für eine Analyse der Winkel zwischen konsekutiven Koordinaten $K_{1,2,3}$ verwendet. Es wird nach rechten Winkeln gesucht. $N_{trans_{rechts}}$ enthält nicht signifikante Winkel und wird deshalb mittels Ramer-Douglas-Peucker (siehe: Abschnitt 8.1) in eine Koordinatenserie L überführt. Wie in Abbildung 8.1 zu erkennen ist, wird die Koordinatenserie $N_{trans_{rechts}}$ dadurch reduziert. RA entspricht den korrespondierenden Indizes der rechten Winkel in L. Es wird eine maximale Abweichung zu einem rechten Winkel ϵ_{diff} definiert. ϵ_{diff} sagt aus, wie exakt ein Winkel von RA einem Vielfachen von $90°$ entsprechen muss. Die Anzahl der Koordinaten von L heißt A.

$$RA := \{i \in \mathbb{N} \quad |i < (A-2),\ |atan2(L_i, L_{i+1}) - atan2(L_{i+1}, L_{i+2})| \equiv n * 90°\} \quad (11.1)$$

Jeder Index i aus RA wird genutzt, um eine Orthogonale Q zu den Koordinaten $K1 = L_{i+1}, K2 = L_i$ zu bilden. Der Winkel α zwischen $K2$ und $K1$ wird berechnet. Die Breite der Straße in Abbildung 11.1 Pixeln beträgt B. Das $delta_{x,y}$ von $K1$ und $K2$ wird normiert ($normLength = \frac{B}{8}$).

$$dX = \frac{K1(x) - K2(x)}{2}$$

$$dY = \frac{K1(y) - K2(y)}{2}$$

$$fac = \frac{\sqrt{dX * dX + dY * dY}}{normLength}$$

$$K'(x) = \frac{dX}{fac}$$

$$K'(y) = \frac{dY}{fac}$$

Drei Suchbereiche $S_{1,2,3}$ werden entlang Q orthogonal (siehe Abbildung 11.1) etabliert. Die Polarkoordinaten $C_{1,2}$ werden um $K1$ konstruiert. C_1 hat eine Distanz von $\frac{B}{4}$ zu $K1$. C_2 hat eine Distanz von $\frac{3B}{4}$ zu $K1$. Ihre Winkel zu $K1$ sind $W_{1,2}$ = $\{\alpha + \pi/2, \alpha - \pi/2\}$.
Jetzt werden aus $C_{1,2}$ und $K1$, durch komponentenweise Addition und Subtraktion von K', drei Koordinatenpaare berechnet $COff_{1,2,3}$. Für jedes dieser Paare werden die Punktmengen $P_{1,2,3}$ zwischen ihnen bestimmt. Hierfür wird der Bresenham Algorithmus verwendet (siehe: 8.2).
Ein gemeinsamer Schwellwert S wird für die Intensitäten der Koordinaten von $P_{1,2,3}$ gefunden. Im Anschluss muss gelten, dass keine der Intensitäten von P_1 über S liegt. Die Intensitäten von $P_{2,3}$ müssen mindestens einen Wert aufweisen, der über S liegt. Ist diese Bedingung erfüllt, werden die euklidischen Distanzen von L_0 bis L_{i+1} akkumuliert. Dieser Wert entspricht der Distanz zur Kreuzung.

12 Hindernisdetektion

Beim Carolo-Cup können bei einer der dynamischen Disziplinen Hindernisse auf der Fahrbahn stehen. Ist dies der Fall muss das Fahrzeug diese erkennen und eine Ausweichbewegung einleiten.

Der Gedanke hinter der Hindernisdetektion besteht darin, zu bewerten, ob die jeweilige Trajektorie, die durch $P_{left,right}$ beschrieben wird, befahrbar ist. Unterscheiden sich die Pixelwerte des Fahrbahnverlaufes nicht stark voneinander, ist die Wahrscheinlichkeit gering, dass ein Hindernis detektiert werden wird.
Der Algorithmus muss für das jeweilige Polynom P von $P_{left,right}$ aufgerufen werden.
Im ersten Schritt werden annähernd äquidistante Koordinatentupel $T_{low,top}$ aus P erzeugt und in KS abgelegt. Die Datenstruktur PD, die P repräsentiert, besitzt einen Wert $PD'lastDefinedY$. Dieser indiziert bis zu welcher Bildzeile von Img_{trans} Koordinaten für die Approximation von P vorlagen. Die Methode $getX(y)$ von PD ermöglicht es eine Bildzeile aus Img_{trans} Y auf ein X aus Img_{trans} abzubilden. Der Pseudocode 7 erzeugt KS.
Die Variablen scanResultCount = 5, pointerLength = $\frac{SpurbreiteFahrzeug}{2}$, range = $\frac{\pi}{4}$ und $\gamma = \frac{2*range}{scanResultCount}$ werden definiert.
Ein Tupel aus zwei Koordinaten K1 und K2 wird auf ein Tupel aus Koordinate $K := K1$ und Winkel $\gamma = (atan2(K1, K2))$ abgebildet. Die Tupel aus KS werden so abgebildet und repräsentieren KW. Die Reihenfolge bleibt erhalten.
Das erste Element der Tupel von KW wird KW_{coor} genannt. Das zweite Element der Tupel von KW wird KW_{angle} genannt.
N ist die Anzahl der Tupel von KW. Im nächsten Schritt werden halbkreisförmige Strukturen $HS_{0,1,..,N-1}$ erstellt. Jedes HS besitzt scanResultCount Koordinaten.
KW wird nun genutzt um die Einträge von jedem HS zu erstellen.
Hierfür werden Polarkoordinaten berechnet. Die Funktion $p(K, \alpha)$ abstrahiert das in 4 beschriebene Verfahren. U ist $\frac{scanResultCount}{2}$.

$$\forall_i \in \{0, 1, ..N-1\} \forall_j \in \{-U/2, ..U/2\}\, HS_{i_j} = p(KW_{i_{coor}}, KW_{i_{angle}} + j * \gamma)) \quad (12.1)$$

Für jedes HS wird der Schwerpunkt, analog zu dem Pseudocode 3, berechnet. C enthält die Koordinaten der Schwerpunkte in korrespondierender Reihenfolge zu KW. Die Anzahl von Koordinaten aus C heißt NC. Die Standardabweichung SD der Pixelwerte der Koordinaten von C wird ermittelt.
Die durchschnittliche Intensität (mean) der Koordinaten von C wird berechnet.

$$mean = \frac{\sum_{i=0}^{NC-1} intensity(C_i)}{NC} \tag{12.2}$$

$$sumOfSquares = \frac{\sum_{i=0}^{NC-1} (intensity(C_i) - mean)^2}{NC - 1} \tag{12.3}$$

$$SD = \sqrt{sumOfSquares} \tag{12.4}$$

Eine Variable $MinSD$ wird definiert. $MinSD$ stellt den Mindestwert dar, bei dem die Annahme getroffen werden kann, dass Hindernisse im Fahrbahnverlauf gefunden werden. Ist $SD < MinSD$ handelt es sich um eine großflächige Reflexion, oder die Fahrbahn enthält keine Hindernisse.
Ist $SD \geq MinSD$ wird eine Variable $filter = mean + fac * SD$ eingeführt. Der Faktor fac stellt die Empfindlichkeit für nachfolgende Auswertung dar.
Nun wird eine Maske M mit einer Größe G von sieben über die Elemente dieser Liste geschoben. Werden mindestens V = fünf Pixelwerten innerhalb der aktuellen Maskenposition gefunden die über filter liegen, wurde ein Hindernis detektiert.
Die Intensitäten der Koordinaten von C werden binarisiert. Hierfür wird $filter$ als Schwellwert verwendet.

$$\sum_{j=0}^{G} intensity(C_{i+j}) \geq V \tag{12.5}$$

Die erste Koordinate KH aus C mit einem Index der 12.5 erfüllt stellt ein detektiertes Hindernis dar. Mit PD werden Koordinaten von dem y Wert ZFF (y-Fahrzeugfrontachse aus Img_{trans}) bis zu dem y aus von KH berechnet. Die Akkumulation der euklidischen Distanzen zwischen diesen Koordinaten ist die Distanz zum erkannten Hindernis. Die Erkennungstiefe der Hindernisdetektion wurde auf 1,5 Meter beschränkt.

Algorithm 7 KS generieren

1: $KS \leftarrow ContainerForCoordiantes$
2: $equidistant \leftarrow 3$
3: $currentDistanceReached \leftarrow 0$;
4: $setLowPoint \leftarrow true$
5: **for** y = $bottomYOf(Img_{trans})$ y>P'lastDefinedY y-=1 **do**
6: **if** setLowPoint **then**
7: $low \leftarrow (PD.getX(y), y)$
8: $setLowPoint \leftarrow false$
9: $continue$
10: **end if**
11: $top \leftarrow (P.getX(y), y)$
12: $currentDistanceReached \leftarrow euclideanDistance(low, top)$
13: **if** $currentDistanceReached < equidistant$ **then**
14: $continue$
15: **else**
16: $distace \leftarrow distance + currentDistanceReached$
17: $currentDistanceReached \leftarrow 0$
18: $setLowPoint \leftarrow true$
19: **end if**
20: **if** $distace > distanceMax$ **then**
21: $break$ {distanceMax entspricht 1.5 Meter}
22: **end if**
23: $T \leftarrow (low, top)$
24: $KS \leftarrow insert\, T$
25: **end for**
26: **return** KS

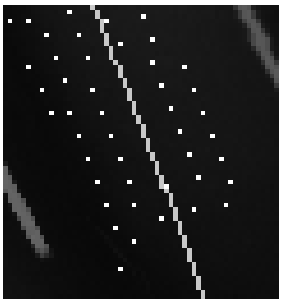

Abbildung 12.1: Gestreute Suchelemtkoordinaten der Hindernisdetektion

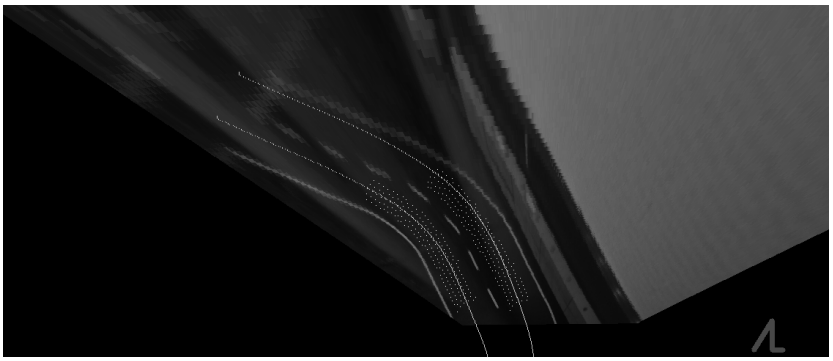

Abbildung 12.2: Suchelemte der Hindernisdetektion
Die Standardabweichung der Pixelwerte ist so gering, dass der Algorithmus vorzeitig terminiert. Es wird nach der Berechnung der Standardabweichung die Annahme getroffen, dass es sich um eine befahrbare Trajektorie handelt.

12.1 Ergebnisse

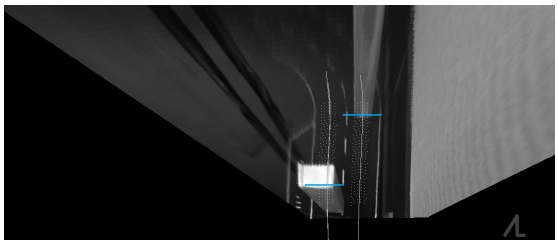

Abbildung 12.3: Links wurde in 630 mm Entfernung ein Hindernis erkannt. Auf der rechten Fahrbahnhälfte wurde ein Hindernis in 1480 mm detektiert.

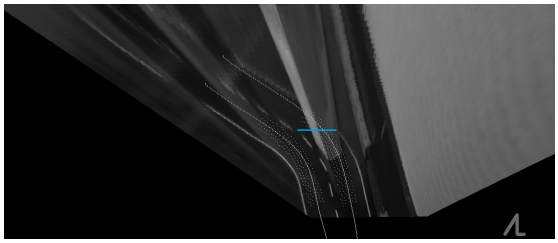

Abbildung 12.4: Auf der rechten Fahrbahnhälfte wurde in 1390 mm Entfernung ein Hindernis erkannt.

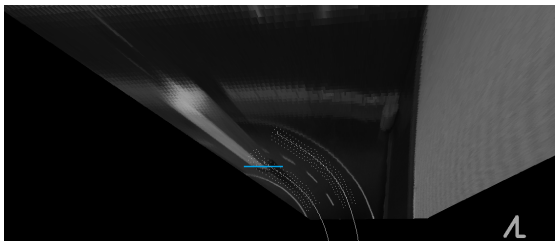

Abbildung 12.5: Auf der linken Fahrbahnhälfte wurde innerhalb einer Kurve in 1170 mm Entfernung ein Hindernis erkannt.

In Abbildung 12.3 und 12.4 sieht man, dass die Distanzen zu den Hindernissen nicht komplett korrekt bestimmt wurden. Der Algorithmus hat sich in diesen Fällen dafür entschieden, die obere Kante der Hindernisse für die Berechnung der Distanz zu nutzen. Diese Fehler entstehen durch die Beleuchtungssituation. Die Standardabweichung ist durch die stark reflektierende Kartonoberseite so hoch, dass die Vorderseite als befahrbar bewertet wird. Die korrekte Distanz ist in der Praxis aber kein sehr kritischer Wert.

13 Test der Software

Die Software wurde im Teststreckenlabor der HAW-Hamburg getestet. Zudem gelang es beim Carolo-Cup 2016 eine sehr geringe Gesamtstrafe von fünf Metern zu erreichen. Innerhalb von drei Minuten legte das Fahrzeug 465 Meter zurück. Hierbei wurde eine durchschnittliche Geschwindigkeit von $9,3 \mathrm{km/h}$ erreicht. Es wurden mehrere Leerstellen-Szenarien geprobt. Keine der Situationen stellte ein Problem dar. Dies zeigte sich auch beim Carolo-Cup 2016, der den eigentlichen Softwaretest darstellt. Beim Rundkurs ohne Hindernisse konnte Platz 4 von insgesamt 15 Teams erreicht werden. Der fünfte Platz in der Gesamtwertung ist auch kein schlechtes Ergebnis.
Das inertiale Navigationssystem konnte visuell getestet werden. Sensorwerte und Bilder der Kamera wurden paarweise gespeichert. Diese Paare konnten dann genutzt werden, um das inertiale Navigationssystem unabhängig von der Zielplattform zu testen. In einer Testbench wurde ein Kameraausfall simuliert und visuell die Validität neuer Trajektorien aus alten Informationen geprüft. Im Teststreckenlabor der HAW Hamburg wurde jede Veränderung der Software ausgiebig getestet.
Es kann unter Umständen zu Problemen kommen, wenn eine Ausweichbewegung ausgeführt wird. Liegt der eigene Straßenverlauf nicht mehr im Kamerabild und die angrenzende Fahrbahn wird detektiert, muss die Regelung dies mit den Daten des inertialen Navigationssystems kompensieren. Diese Situationen können durch eine verbesserte Ausweichbewegung entschärft werden, bei der mehr von der eigenen Fahrbahn im Kamerabild zu sehen ist. Beim Carolo-Cup 2016 hat die Hindernisdetektion sehr gut funktioniert. Doch die Ausweichbewegung war leider nicht optimal. Durch eine bessere Auswertung der Informationen des inertialen Navigationssystems und eine weichere Ausweichbewegung hätte man beim Wettbewerb deutlich mehr erreichen können.

14 Fazit

Es ist gelungen eine sehr robuste Bilderkennung auf einer günstigen Hardware-Plattform zu realisieren und einen der vorderen Plätze beim Carolo-Cup 2016 zu erreichen. Nach einem Refactoring konnten die Lines of Code auf knapp unter 7000 Zeilen reduziert werden. Die Umsetzung der Ideen hat oftmals sehr viel Zeit in Anspruch genommen. Es hat sehr viel Spaß gemacht eine komplett eigene Bildverarbeitung zu entwickeln und beim Carolo-Cup 2016 teilzunehmen. Durch den sehr engen Verwendungskontext konnten neue Algorithmen entwickelt werden, die sich stark von den klassischen Ansätzen der Linienerkennung unterscheiden. Das Beispiel der Detektion von Kreuzungen zeigt, wie man ohne einen Mehraufwand durch die Analyse der durch die Natter gewonnenen Daten eine einfache Kreuzungserkennung realisieren kann. Durch das inertiale Navigationssystem konnten die formulierten Eigenschaften der Robustheit erreicht werden. Auch das regelbasierte Verfahren der Mittellinienerkennung trägt dieser Robustheit bei. Auf dem Zielsystem traten Zykluszeiten δ_t von unter 5 ms durch die Bildverarbeitung auf. Hierdurch können mehr als 70 $^{\text{Frames}}/_{\text{Sekunde}}$ ausgewertet werden.
Wären 14ms überschritten worden, hätte das Framework dies in einem Log-File protokolliert. Das Log-File hat nie einen Eintrag enthalten. Die Echtzeitanforderung konnte so durch Testen erfüllt werden. Hätte das Log-File einen Eintrag enthalten, wäre die Anytime Eigenschaft der Natter aktiviert worden. Des Weiteren wäre die maximal zu erkennenden Anzahl von Mittellinienelementen auf eine Obergrenze beschränkt worden.
Der limitierende Faktor der maximalen Fahrzeuggeschwindigkeit wird durch die Physik festgelegt. Durch die Umsetzung besserer Regelalgorithmen könnte man deutlich schneller fahren. Für das inertiale Navigationssystem muss weitere Sensorik verbaut werden, um die zurückgelegte Distanz beim Blockieren der Räder bestimmen zu können. Dennoch konnte das System sehr gut beim Carolo-Cup eingesetzt werden.

Literaturverzeichnis

[Chernov 2012] Chernov, Nikolai: "C++ code for circle fitting algorithms". (2012). – URL http://people.cas.uab.edu/~mosya/cl/CPPcircle.html. – letzter Zugriff: 14.04.2016

[Dabrowski 2014] Dabrowski, Kamil: Ramer-Douglas-Peucker. (2014). – URL https://www.namekdev.net/2014/06/. – letzter Zugriff: 14.04.2016

[GNU-Scientific-Library 2016] GNU-Scientific-Library: *Multi-parameter fitting*. 2016. – URL https://www.gnu.org/software/gsl/manual/html_node/Multi_002dparameter-fitting.html. – letzter Zugriff: 14.04.2016

[Hans-Joachim Bungartz 2002] Hans-Joachim Bungartz, Michael G.: *Einüfhrung in die Computergraphik: Grundlagen, Geometrische Modellierung, Algorithmen (Mathematische Grundlagen Der Informatik) (German Edition)*. Vieweg+Teubner Verlag, 2002. – ISBN 3528167696

[Helland 2013a] Helland, Tanner: Linsenkorrektur. (2013). – URL http://www.imagemagick.org/Usage/lens/correcting_lens_distortions.pdf. – letzter Zugriff: 14.04.2016

[Helland 2013b] Helland, Tanner: LinsenkorrekturAlgorithmus. (2013). – URL http://www.tannerhelland.com/4743/simple-algorithm-correcting-lens-distortion/. – letzter Zugriff: 14.04.2016

[Juan Pablo Balarini 2015] Juan Pablo Balarini, Sergio N.: A C++ Implementation of Otsu's Image Segmentation Method. (2015). – URL http://www.ipol.im/pub/pre/158/preprint.pdf. – letzter Zugriff: 14.04.2016

[Meisel 2012] Meisel, Prof. Dr.-Ing A.: RV Robot Vision. (2012)

[Rosettacode 2016] Rosettacode: *Polynomial Regression*. 2016. – URL https://rosettacode.org/wiki/Polynomial_regression#C. – letzter Zugriff: 14.04.2016

[Taubin 1991] Taubin, G.: *Estimation Of Planar Curves, Surfaces And Nonplanar Space Curves Defined By Implicit Equations, With Applications To Edge And Range Image Segmentation*. IEEE Trans. PAMI, Vol. 13, pages 1115-1138, 1991

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 28.04.2016 Clemens Drauschke