Kalibrierungsfreie Spielfeldlinienerkennung im Roboterfußball

Bachelorarbeit

Lukas Malte Monnerjahn
Matrikelnummer: 4437178

5. März 2021

Fachbereich 3 Mathematik / Informatik
Studiengang Informatik

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbstständig angefertigt und nicht anderweitig zu Prüfungszwecken vorgelegt habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel verwendet. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen sind, sind als solche kenntlich gemacht.

Bremen, den 5. März 2021

.....
(Lukas Malte Monnerjahn)

Inhaltsverzeichnis

1. Einleitung

1.1 Motivation

Das Roboterfußballteam B-Human nimmt seit langem erfolgreich an RoboCup-Wettbewerben teil. B-Human strebt dabei das Ziel „Aus der Tasche auf das Feld“ an, also die Roboter unabhängig von Spielfeld und Gegner direkt einsatzbereit zu haben. Die von Menschen zu erledigenden Vorarbeiten, bis das Team tatsächlich konkurrenzfähig ist, sollen möglichst gering gehalten werden.

Eine der Aufgaben, die bisher dennoch von Hand erledigt werden muss, ist die Farbklassenkalibrierung der Kameras. Diese wird benötigt, da einige Bildverarbeitungsschritte auf der Unterscheidung vom Grün des Spielfelds und dem Weiß von Feldlinien, Ball und Roboterteilen gegenüber anderen Farben basieren. Damit diese Verarbeitungsschritte funktionieren, müssen vor jedem Spiel an einem neuen Ort oder unter anderen Beleuchtungsbedingungen die Parameter zur Unterscheidung der verschiedenen Farbklassen auf die Spielfeldfarbe und die Lichtverhältnisse eingestellt werden. [Röfer et al. , 2019, S. 18,22]

Aufgrund der Corona-Pandemie sind bei den nächsten RoboCup-Wettbewerben keine Spielaustragungen mit physischer Präsenz beider Teams möglich. Stattdessen ist es erforderlich zugeteilte Roboter ohne eigene Präsenz am Spielort mit dem eigenen Code spielbereit zu machen. Anweisungen an das Personal vor Ort und über das Internet übertragene Daten müssen dafür ausreichen. Aufwendige von Hand ausgeführte Kalibrierungsprozesse würden dadurch zusätzlich erschwert. In den gegen Ende des Bearbeitungszeitraums veröffentlichten Wettbewerbsregeln für das Jahr 2021 ist außerdem eine Challenge mit dem Ziel der vollständig autonomen Kalibrierung der Roboter enthalten, was manuelle Kalibrierung für diesen Teilwettbewerb komplett ausschließt. [RoboCup Technical Committee, S. 26]

Ein weiteres Problem bei der Verwendung von Farbklassen tritt bei natürlichen Beleuchtungsbedingungen zu Tage. Da das Spielfeld uneinheitlich beleuchtet sein kann, ist es manchmal unmöglich die Farbklassen so einzustellen, dass jedes Pixel korrekt klassifiziert wird. Die darauf aufbauenden Bildverarbeitungsschritte vertrauen dann der falschen Klassifikation und kommen gelegentlich zu falschen Ergebnissen.

Um die zeitaufwändige Farbklassenkalibrierung nicht mehr durchführen zu müssen, will B-Human die betroffenen Bildverarbeitungsschritte von den manuell kalibrierten Farbklassen unabhängig machen. Zu diesen Schritten gehört auch die Erkennung der Spielfeldlinien. Diese ist von großer Bedeutung für die Selbstlokalisierung der Roboter, welche für ein korrektes Weltmodel und einige Verhaltensentscheidungen benötigt wird.

1.2 Zielsetzung

Das Ziel dieser Arbeit ist es ein neues Linienerkennungsverfahren zu implementieren, welches keine Kalibrierung benötigt und den Anforderungen eines RoboCup Wettbewerbs gerecht wird. Die bestehende Linienerkennung wird dabei soweit möglich wiederverwendet, während die kalibrierungsabhängigen Teile ersetzt werden. Bei den zu ersetzenden Teilen handelt es sich einerseits um das Finden initialer Ansatzpunkte, die vermutlich zu Linien gehören und andererseits um die Validierung gefundener Linien, welche falsche Funde verringert.

Dazu werden in dieser Arbeit verschiedene Ansätze implementiert und getestet. Zum Finden der `LineSpot`s wird die Farbsegmentierung dynamisch je Bild erzeugt, ohne dass feste Grenzwerte zur Aufteilung der Farbklassen oder Kalibrierung durch Menschen benötigt werden. Dazu werden verschiedene Ansätze zur Einteilung des Bildes in homogene Regionen und zur Zuordnung von Farbklassen zu Bildregionen implementiert, die jeweils miteinander kombiniert werden können. Zusätzlich wird eine kalibrierungsfreie Methode zur Validierung gefundener Linien implementiert, die auf der relativen Helligkeits- und Farbsättigungsänderung von Linien gegenüber dem Feld beruht.

Das neu entwickelte Linienerkennungsverfahren muss auf den verwendeten Robotern echtzeitfähig sein, eine geringe Falschpositiv-Rate aufweisen und unter verschiedenen Beleuchtungssituationen funktionieren, um auf RoboCup Wettbewerben zu bestehen. Insbesondere ist das Vermeiden von Fehldetektionen wichtiger als möglichst viele Feldlinien zu detektieren, weil die fälschliche Detektion nicht vorhandener Linien zur Fehllokalisation des Roboters führen kann.

Die Ausgestaltung der Lösungsansätze wird im Kapitel 3 Implementierung genauer erläutert. Die durchgeführten Tests und ihre Ergebnisse sind in Kapitel 4 Evaluation beschrieben.

1.3 Hintergrund

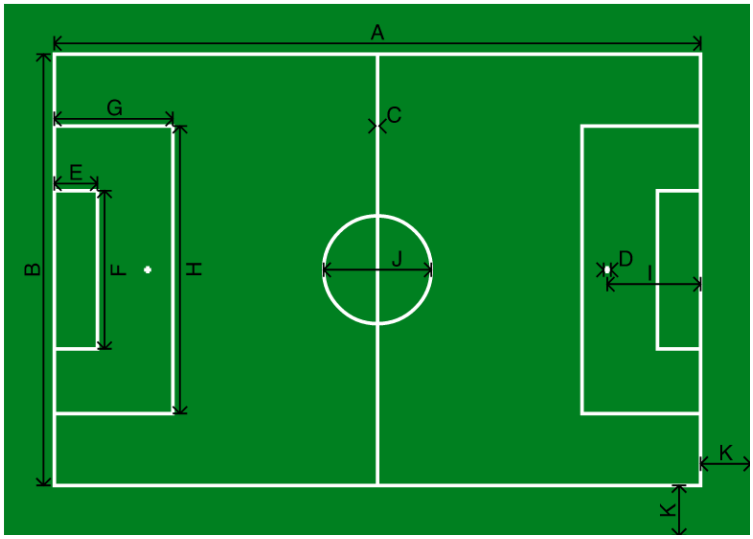
Diese Arbeit entstand im Kontext des RoboCup Teams B-Human, einem studentischen Projekt der Universität Bremen in Kooperation mit dem Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI).

Der RoboCup hat zum Ziel die Forschung in den Bereichen Robotik und künstliche Intelligenz zu fördern. Dazu wurde das Ziel ausgerufen bis Mitte des 21. Jahrhunderts mit einem vollständig autonomen Roboterteam gegen den dann amtierenden menschlichen Weltmeister zu gewinnen. Seit 1997 werden RoboCup Wettbewerbe durchgeführt, die verschiedenen Teams die Möglichkeit geben, ihr Können zu messen und Wissen auszutauschen. Die Regeln werden dabei von Jahr zu Jahr angepasst, um neue Herausforderungen anzugehen und sich nach und nach dem menschlichen Fußball anzunähern. Neben verschiedensten Roboterfußballligen umfasst der RoboCup inzwischen auch Rescue-, @Home- und Industrial-Ligen, sowie einige Juniorenwettbewerbe. [RoboCup Federation, 2016, Unterseiten /objective und /a_brief_history_of_robocup]

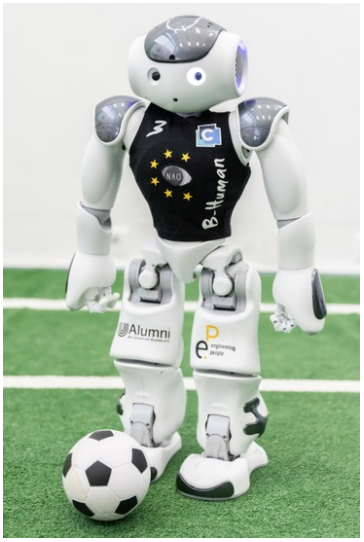
Das Team B-Human nimmt seit 2009 an RoboCup Wettbewerben in der Standard Platform League (SPL) teil. Dabei hat B-Human neun Siege auf der RoboCup German Open errungen und siebenmal die RoboCup World Championship gewonnen Team B-Human [2020]. Bei der SPL handelt es sich um eine Roboterfußballliga in der alle Teams baugleiche Roboter verwenden, die während eines Spiels autonom agieren RoboCup Federation [2020]. Dadurch ist die SPL ein Softwarewettbewerb.

In der SPL wird aktuell der NAOv6 Roboter von Softbank Robotics verwendet. Dieser ist ein 574mm großer humanoider Roboter, mit einer Vielzahl an Sensoren über die er seine Umgebung wahrnimmt. Für diese Arbeit hauptsächlich relevant sind die im Kopf des Roboters übereinander positionierten Kameras. In Abbildung 1.1b sind sie als kleine dunkle Öffnungen im Gehäuse des Roboterkopfes zu erkennen, dort wo bei einem Menschen Mund und Stirn wären. Zusätzlich verfügt der NAOv6 auch über Gelenkwinkelmesser, Mikrophone, eine Inertial Measurement Unit (Accelerometer und Gyroskop), Fußdrucksensoren, sowie Kontaktsensoren an Kopf, Händen und Füßen. SoftBank Robotics [2018]

In einem SPL Spiel treten 5 Roboter je Team auf einem 9×6 Meter großen Kunstrasenfeld gegeneinander an. Mit den SPL Regeln von 2020 entsprechen die Spielfeldlinien ungefähr denen der FIFA Regeln, lediglich die Kreisausschnitte um den Strafraum (Penalty Area im Roboterfußball) und die Eckballmarkierungen fehlen. Die vorhandenen Linien sind an die Größenverhältnisse im Roboterfußball angepasst. Die Linien müssen weiß und 5cm breit sein, es ist jedoch nicht vorgegeben wie das erreicht wird. Sie können beispielsweise aus weißem Kunstrasen bestehen, mit Klebeband aufgeklebt oder mit weißer Farbe aufgesprüht sein. [RoboCup Technical Committee, 2019, S. 1-3]



(a) SPL Spielfeld [RoboCup Technical Committee, 2019, S. 2]



(b) NAOv6 Roboter des Teams B-Human Popp [2019]

Abbildung 1.1

2. Grundlagen

Dieses Kapitel stellt die grundlegenden Methoden und Konzepte vor, auf welche in dieser Arbeit zurückgegriffen wird. Die ersten Unterabschnitte (2.1 bis 2.4) beschreiben die verwendeten Bildvorverarbeitungstechniken und Datenstrukturen. Die Abschnitte 2.5 B-Human Framework, 2.6 NAO Kameras und 2.7 Koordinatensysteme im B-Human System geben einen Überblick über die genutzte Infrastruktur. Anschließend gehe ich in den Abschnitten 2.8 Bisherige Linienerkennung bei B-Human und 2.9 Linienerkennung bei anderen Teams genauer auf Algorithmen zur Linienerkennung im RoboCup ein. Abschließend wird in 2.10 Verwandte Themengebiete auf Linienerkennung in Fußballanalyse- und Fahrassistenzsystemen eingegangen.

2.1 Lineare Filter

Lineare Filter beschreiben die lokale Anwendung einer linearen Verknüpfung, dass heißt gewichtete Summation, des Filters mit einem Eingangssignal. Das entspricht der mathematischen Operation der linearen Faltung, welche zwei Funktionen gleicher Dimensionalität miteinander verknüpft. Ein diskretes lineares Filter ist dabei gegeben durch eine Filtermatrix H, deren Einträge die Gewichtung der Samples des Eingangssignals bestimmen.

Im Kontext der Bildverarbeitung beschreiben lineare Filter Nachbarschaftsoperationen, bei denen der Wert eines Pixels nach Anwendung des Filters nur vom eigenen Wert und den Werten der Pixel in der Umgebung des betrachteten Pixels abhängt. Die Größe der betrachteten Umgebung hängt von der Größe der Filtermatrix ab. Viele nützliche Bildverarbeitungsoperationen wie Bildglättung oder Kantendetektion lassen sich als lineare Filter beschreiben.

Gegeben ein einkanaliges zweidimensionales Bild I und eine Filtermatrix H, ist die Operation des linearen Filters durch die diskrete lineare Faltung $(*)$ definiert als:

$$I'(u,v) = I * H := \sum_{i=-\infty}^{\infty} \sum_{j=-\infty}^{\infty} I(u-i, v-j) \cdot H(i,j) = \sum_{(i,j) \in R_H} I(u-i, v-j) \cdot H(i,j) \tag{2.1}$$

Werte außerhalb der Filtermatrix H werden dabei als Null angenommen. Um bei der Anwendung eines Filters am Rand des Bildes Werte für Pixel außerhalb des Bildes zu erhalten, werden je nach Anwendungskontext unterschiedliche Methoden verwendet. Typisch sind unter anderem die Verwendung des Wertes des nächstgelegenen Bildpixels oder die Spiegelung des Bildes an der Bildkante. [Jähne, 1997, S. 105-107, 117-119] [Burger u. Burge, 2015, S. 95-97, 105-108]

2.2 Kantenerkennung

In Graustufenbildern zeichnen sich Kanten durch deutliche Helligkeitsänderungen auf kleinem Raum in einheitlicher Richtung aus. Eine stärkere Helligkeitsänderung spricht dabei für eine schärfere Kante. In einem kontinuierlichen Helligkeitsverlauf entspricht die Kantenstärke daher seiner ersten Ableitung, wobei das Vorzeichen von der Richtung des Helligkeitsübergangs abhängt. In räumlich diskretisierten Bildern lässt sich die Ableitung an einem Pixel annähern,

indem die Steigung einer Geraden durch die beiden benachbarten Pixel ermittelt wird. Gibt $f(x)$ die Intensität von in einer Reihe liegenden Pixeln an, dann ist die Ableitung $f'(x)$:

$$f'(x) = \frac{df}{dx} \approx \frac{f(x+1) - f(x-1)}{2} \tag{2.2}$$

Da Bilder zweidimensionale Strukturen sind, müssen partielle Ableitungen einmal in horizontaler und einmal in vertikaler Richtung gebildet werden. Als lineares Filter notiert ergibt sich

$$H_x^D = \frac{1}{2}\begin{bmatrix}-1 & 0 & 1\end{bmatrix}\ ,\ H_y^D = \frac{1}{2}\begin{bmatrix}1\\0\\-1\end{bmatrix} \tag{2.3}$$

Die Ableitungsfilter sprechen stark auf horizontale und vertikale Kanten an, sind jedoch sehr anfällig gegenüber Bildrauschen. Um zuverlässigere Ergebnisse zu erhalten, werden zur Kantendetektion daher meist 3×3 Filtermatrizen verwendet, die sich in der verwendeten Glättung unterscheiden können. Die Filtermatrizen für horizontale Kanten der gängigen Kantenfilter von Prewitt, Sobel und Scharr sind in der nachfolgenden Tabelle aufgeführt. [Jähne, 1997, S. 348-355] [Burger u. Burge, 2015, S. 125-131]

Prewitt_x	Sobel_x	Scharr_x
$\begin{bmatrix}-1 & 0 & 1\\-1 & 0 & 1\\-1 & 0 & 1\end{bmatrix}$	$\begin{bmatrix}-1 & 0 & 1\\-2 & 0 & 2\\-1 & 0 & 1\end{bmatrix}$	$\begin{bmatrix}-3 & 0 & 3\\-10 & 0 & 10\\-3 & 0 & 3\end{bmatrix}$

Tabelle 2.1: Filtermatrizen gängiger Kantenfilter

Um die Richtung einer Kante festzustellen werden die horizontale und vertikale Kantenstärke in einem Gradientenvektor ∇I zusammengefasst. Dieser verläuft in Richtung der Normalen der Kante und sein Betrag ergibt rotationsinvariant die Kantenstärke.

$$\nabla I = \begin{pmatrix} I_x \\ I_y \end{pmatrix} = \begin{pmatrix} \frac{\partial I}{\partial x} \\ \frac{\partial I}{\partial y} \end{pmatrix} \tag{2.4}$$

Hierüber hinaus gibt es viele weitere Kantendetektionsverfahren, etwa den Laplacian-of-Gaussian Operator [Jähne, 1997, S. 359-362] [Burger u. Burge, 2015, S. 135] welcher Kantenpositionen anhand von Nulldurchgängen in der zweiten Ableitung feststellt oder den Canny-Algorithmus Canny [1986].

2.3 Bildglättung

Glättungs- oder Weichzeichnungsfilter glätten Unebenheiten im Bild und werden häufig zur Rauschverminderung verwendet. In der Umsetzung als lineare Filter berechnen sie einen gewichteten Durchschnitt über die Umgebung des weichgezeichneten Pixels.

Die simpelste Umsetzung eines Glättungsfilters ist das Boxfilter, bei dem alle umliegenden Pixel gleich gewichtet werden. Es wird selten verwendet, da es nicht drehungsinvariant ist und dem Zentrum keine höhere Gewichtung als den Rändern zukommen lässt.

Häufige Verwendung findet hingegen das Gaussfilter, welches sich aus der Diskretisierung der zweidimensionalen Normalverteilungsfunktion ergibt (vgl. Gleichung 2.5). Die Stärke der Glättung kann durch die Wahl des Parameters σ und der Größe der Filtermatrix beeinflusst werden. [Burger u. Burge, 2015, S. 103]

$$g_\sigma(x, y) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{x^2 y^2}{2\sigma^2}} \tag{2.5}$$

Boxfilter	Gaussfilter, $\sigma = 1$ (gerundet)
$\frac{1}{9}\begin{bmatrix}1 & 1 & 1\\ 1 & 1 & 1\\ 1 & 1 & 1\end{bmatrix}$	$\begin{bmatrix}0.07511 & 0.12384 & 0.07511\\ 0.12384 & 0.20418 & 0.12384\\ 0.07511 & 0.12384 & 0.07511\end{bmatrix}$

Tabelle 2.2: Filtermatrizen des Box- und des Gaussfilters

Nachteilig an linearen Glättungsfiltern ist, dass sie nicht nur Rauschen entfernen, sondern gleichzeitig auch Kanten verwischen. Da die Kameras der NAO Roboter insbesondere in lichtschwachen Umgebungen oder bei Verwendung kurzer Belichtungszeiten einiges Bildrauschen erzeugen, ist eine gute Bildglättung dennoch wichtig. Entsprechend hat sich auf Bildern der NAO Roboter Bildglättung mit einer 3×3 Gaussfiltermatrix als durchweg positiv erwiesen. Während Bildrauschen unterdrückt wird, bleiben echte Konturen weiterhin vorhanden, wovon die Kantenerkennung stark profitiert. [Thielke, 2016, S. 32]

2.4 Union-Find Datenstruktur

Eine Union-Find Datenstruktur, auch disjoint-set Datenstruktur genannt, verwaltet eine Partition einer Menge. Sie ist benannt nach den beiden Operationen, die sie bereitstellen muss, nämlich der Vereinigung von zwei Teilmengen (Union) und dem Finden eines Repräsentanten der Partition, zu welcher ein gegebenes Element gehört (Find) [Cormen et al. , 2009, S. 561-562]. In dieser Arbeit nutze ich eine Union-Find Datenstruktur zur effizienten Vereinigung ähnlichfarbener Scanlinienregionen (siehe dazu 2.8 Bisherige Linienerkennung bei B-Human und 3.3.2 Segmentierung des Spielfelds).

Eine einfache und effiziente Umsetzungsmöglichkeit ist die Implementierung der Union-Find Datenstruktur als Wald (disjoint-set forest). Dabei werden die einzelnen Teilmengen als Bäume repräsentiert, in denen jeder Knoten jeweils seinen Elternknoten kennt. Der Wurzelknoten, welcher keinen neuen Elternknoten mehr hat, fungiert als Repräsentant für die Partition. Die Union-Operation vereinigt zwei Teilmengen, indem der Wurzelknoten der einen Teilmenge

den Wurzelknoten der anderen Teilmenge als Elternknoten zugewiesen bekommt. Bei der Find-Operation werden von einem gegebenen Knoten aus rekursiv alle Elternknoten auf dem Weg zum Wurzelknoten durchlaufen. Ist der Wurzelknoten erreicht, wird dieser als Repräsentant der Partition des betrachteten Knotens zurückgegeben.

Zur Optimierung der amortisierten Laufzeit werden außerdem die Heuristiken Union-by-Rank und Pfadkompression bei Find-Operationen verwendet. Diese dienen dazu bei der Vereinigung die Tiefe des entstehenden Baums zu begrenzen und aktiv zu verringern. Union-by-Rank bedeutet dabei für jeden Knoten einen Rang, der die maximale Tiefe des Teilbaums unter dem Knoten angibt, zu speichern. Bei einer Vereinigungsoperation wird dann der Knoten mit dem kleinerem Rang als Elternknoten ausgewählt. Haben beide Knoten den gleichen Rang ist es egal welcher zum Elternknoten gemacht wird, aber der Rang des neuen Elternknoten wird um eins erhöht. Die Pfadkompression verringert bei Find-Operationen die Tiefe des Baums. Bei der Suche nach dem Wurzelknoten zu einem gegebenen Knoten, wird dabei allen auf dem Weg zur Wurzel liegenden Knoten die Wurzel als direkter Elternknoten zugeordnet. Damit müssen nachfolgende Find-Operationen möglichst wenig Elternknoten auf der Suche nach dem Wurzelknoten ihrer Menge durchlaufen. [Cormen et al. , 2009, S. 568-572]

Über eine Folge von m Union- und Find-Operationen bei einer Anzahl von n Elementen in der partitionierten Menge, also maximal $n-1$ Union-Operationen, liegt die Laufzeit dieser Umsetzung in der Größenordnung $\mathcal{O}(m \cdot \alpha(n))$. α ist dabei die inverse Ackermannfunktion. Diese wächst sehr langsam und kann für viele praktische Belange als konstant angesehen werden. Die Implementierung von Union-Find Datenstrukturen als disjoint-set forest, führt entsprechend zu einem Laufzeitverhalten nahezu linear proportional zur Anzahl der ausgeführten Union- und Find-Operationen. [Cormen et al. , 2009, S. 573-581]

2.5 B-Human Framework

Der in dieser Arbeit geschriebene Code ist in das B-Human Framework eingebettet und nur in diesem lauffähig. Mit dem B-Human Framework laufen auf den Robotern mehrere Threads mit unterschiedlichen Aufgaben. Während eines Wettbewerbsspiels sind vier Threads aktiv, die Upper, Lower, Cognition und Motion heißen. Außerhalb von Wettbewerbsspielen ist noch ein Debug-Thread aktiv, welcher bei Ausführung im Simulator zur Kommunikation mit dem Hostsystem dient.
Die Threads Lower und Upper verarbeiten die Bilder der beiden Kameras. Im Cognition-Thread werden der aktuelle Zustand auf dem Feld modelliert und Verhaltensentscheidungen getroffen. Der Motion-Thread liest Gelenkwinkel und andere Sensoren des Roboters aus und steuert die Bewegungen des Roboters. Die Threads laufen dabei nicht durchgängig, sondern werden erst durch das Eintreffen neuer Daten aktiviert. Die Bildverarbeitungs-Threads Upper und Lower werden also durch das Eintreffen neuer Kamerabilder aktiviert, was jeweils 30-mal pro Sekunde geschieht. Der Motion-Thread läuft auf der Frequenz der Roboterhardware, welche voraussetzt die Gelenke mit 83 Hz anzusteuern, und der Cognition-Thread wird immer dann aktiviert, wenn die anderen Threads einen Ausführungszyklus beendet haben und ihre berechneten Daten weitergeben. [Röfer et al. , 2019, S. 26]

Der in den Threads ausgeführte Code ist in Module und Repräsentationen gegliedert. Module erfüllen eine klar festgelegte Aufgabe, wie etwa Linien in den Kamerabildern zu erkennen. Die von einem Modul bereitgestellten Daten werden in Repräsentationen zusammengefasst. Eine Repräsentation kapselt zusammengehörige Informationen in einem Datentyp und stellt grundlegende Operationen zum Arbeiten mit diesen Daten bereit. Alle Repräsentationen

werden an einer zentralen Stelle, dem sogenannten Blackboard verwaltet. Somit existiert von jeder Repräsentation in einem Thread immer nur eine Instanz. Damit die Module ausgeführt werden können, benötigen sie Eingabedaten. Daher gibt jedes Modul an, welche Repräsentationen es als Eingabe benötigt und welche es bereitstellen kann. Daraus ergibt sich die Ausführungsreihenfolge der Module. Da die Module nur von den Repräsentationen jedoch nicht von anderen Modulen abhängen, können sie problemlos ausgetauscht werden. Nur wenn Repräsentationen geändert werden, müssen andere Codestellen angepasst werden. [Röfer et al. , 2019, S. 26-29]

Während die Roboter spielen, läuft neben dem zum Fußballspielen benötigten Code ein Logger, der nach jedem Durchlauf eines Threads den aktuellen Zustand wichtiger Repräsentationen in einer Log-Datei auf einem am Roboter eingesteckten USB-Stick festhält. Das beinhaltet standardmäßig alle Daten, die der Roboter über Kameras oder andere Sensoren aufnimmt, sowie wichtige Zwischenergebnisse und Verhaltensentscheidungen. Damit können alle Datenverarbeitungs- und Entscheidungsprozesse des Roboters im Nachhinein nachvollzogen werden. Wichtig zu beachten ist, dass die Kamerabilder nicht im von der Kamera gelieferten Rohformat abgespeichert werden können, da dies zu viel Speicherplatz und Datenbandbreite für die Übertragung auf den USB-Stick benötigen würde. Die Bilder werden daher erst mit JPEG komprimiert und dann abgespeichert. [Röfer et al. , 2019, S. 48]

Zusätzlich gehört zum B-Human Framework ein Simulator, welcher es erlaubt beliebige Spielsituationen als 3D-Simulation auszuführen. Ebenso ist es damit möglich aufgenommene Logdateien aus vergangenen Spielen abzuspielen und darauf geänderte Module erneut auszuführen oder auch eine Verbindung zu einem aktiven Roboter aufzubauen, um sich dessen Bildaufnahmen und Variablen-Werte in den Repräsentationen in Echtzeit ansehen zu können [Röfer et al. , 2019, S. 144-167]. Mittels des Simulators ist es auch möglich die umfangreichen Debug-Funktionen des Frameworks zu verwenden. An den Debug-Thread gestellte Debug Requests können zur Laufzeit entsprechend markierte Codestellen aktivieren, die ansonsten nicht ausgeführt werden. Darauf basieren noch einige weitere Debugging Funktionen, insbesondere die Debug Drawings. Dabei handelt es sich um dynamisch generierte Vektorgrafiken, die als Overlay über einem anderen Bild angezeigt werden können, beispielsweise um zu visualisieren, welche Linien in einem Kamerabild gesehen wurden [Röfer et al. , 2019, S. 42-47]. Für diese Arbeit sind darüber hinaus noch die Funktionen zum Messen der Ausführungszeiten wichtig, die im B-Human Framework Stopwatches genannt werden. Sie ermöglichen die Ausführungsdauer einzelner Module oder Codeabschnitte aufzuzeichnen und damit die Echtzeitfähigkeit des Systems zu testen [Röfer et al. , 2019, S. 48].

2.6 NAO Kameras

Die Kameras des NAO Roboters bieten unterschiedliche Auflösungen und Bildraten an. B-Human verwendet bei der oberen Kamera eine Auflösung von 640×480 Pixel und bei der unteren Kamera 320×240 Pixel, welche die Kameras jeweils mit einer Bildrate von 30 Hz liefern. Bei der unteren Kamera wird die geringere Auflösung genutzt, da diese immer auf den Nahbereich vor dem Roboter gerichtet ist und daher auch bei geringer Auflösung alle Objekte im Bild erkannt werden können. Die Sichtbereiche der beiden Kameras überschneiden sich kaum und liefern ihre Bilder auch nicht synchron. Ihre Bilder werden daher getrennt verarbeitet. Die Kameras liefern ihre Bilder im YUV422-Format. Für zwei horizontal nebeneinander liegende Pixel gibt es zwei Helligkeitswerte (Y), aber jeweils nur einen Wert für die Blau- (U) und Rotchromazität (V). Die Kameras arbeiten mit einem Rolling Shutter, nehmen verschiedene Bildzeilen also nacheinander auf. [Röfer et al. , 2019, S. 54] [SoftBank Robotics, 2018, Unterseite: technical-overview/video-cameras]

2.7 Koordinatensysteme im B-Human System

Im B-Human System werden verschiedene Koordinatensysteme genutzt. In Kamerabildern liegt der Koordinatenursprung in der oberen linken Ecke, die x-Achse verläuft also nach rechts und die y-Achse nach unten. Demgegenüber stehen die Koordinatensysteme, die Positionen auf dem Feld angeben. Einerseits können Feldkoordinaten relativ zum Roboter angegeben werden. Der Koordinatenursprung befindet sich dabei im Torso des Roboters. Andererseits können sie als globale Feldkoordinaten, die für alle Roboter des Teams gleich sind, angegeben werden. In dieser Arbeit verwende ich nur die Feldkoordinaten relativ zum Roboter, welche ich nachfolgend als Feldkoordinaten bezeichne. [Röfer et al. , 2019, S. 55]

Um die Position von im Bild detektierten Objekten auf dem Feld feststellen zu können, ist es nötig vom Bildkoordinatensystem in ein Feldkoordinatensystem transformieren zu können. Als Zwischenschritt wird zunächst die Verzerrung, die sich aus der Kombination des Rolling Shutters der Kamera mit der Eigenbewegung des Roboters ergibt, kompensiert und der Koordinatenursprung in die Bildmitte auf Höhe des Horizonts verschoben. [Röfer et al. , 2019, S. 57]

Anschließend kann mit einer Transformationsmatrix von den korrigierten Bildkoordinaten zum roboterrelativen Feldkoordinatensystem transformiert werden. Damit das funktioniert, muss die Höhe des Objekts im Bild über dem Boden bekannt sein, da sich sonst aus einem zweidimensionalem Bild keine eindeutige dreidimensionale Position auf dem Feld bestimmen lässt. Sofern nicht anders angegeben, wird in dieser Arbeit immer von der Bodenhöhe ausgegangen, weil sich dort die Feldlinien befinden [Röfer et al. , 2019, S. 56]. Aller für die Transformationen benötigte Code ist bereits im B-Human System vorhanden, ebenso wie die Möglichkeit zur Rückprojektion von Feldkoordinaten in das Bildkoordinatensystem.

2.8 Bisherige Linienerkennung bei B-Human

Die Linienerkennung bei B-Human läuft in einem Modul namens `LinePerceptor` ab, welches die zwei Repräsentationen `LinesPercept` und `CirclePercept` erzeugt. Im `LinesPercept` sind alle gefundenen Linien in Feldkoordinaten enthalten, sowie die Koordinaten der Bildpunkte aus denen die Linien jeweils konstruiert wurden. Im CirclePercept wird gespeichert, ob der Mittelkreis gefunden wurde und falls er gefunden wurde, wo der Mittelpunkt in Feldkoordinaten liegt.

Die Grundlage dafür ist die Farbsegmentierung der Eingabebilder. Die von den Kameras im YUV422-Format gelieferten Bilder werden dafür zunächst in eine HSV-Darstellung umgewandelt, bei der für jedes Pixel ein Helligkeits- (Value), Farb- (Hue) und Sättigungswert (Saturation) vorliegt. Die Farbsegmentierung erfolgt dann pixelweise mit Schwellwerten für Mindestsättigung von Farben, Farbbereich des Feldes und Helligkeit in die Farbklassen Grün (Feld), Weiß, Schwarz und Anders. Für die Linienerkennung ist die Farbe Schwarz nicht relevant, sie wird jedoch in anderen Bildverarbeitungsschritten, die ebenfalls auf die Farbsegmentierung zurückgreifen, verwendet. [Röfer et al. , 2019, S. 60]

Um nicht das gesamte Kamerabild analysieren zu müssen, was mit der Rechenleistung der gegebenen Hardware auch nicht in Echtzeit möglich wäre, haben sich im RoboCup verschiedene Möglichkeiten zur Teilanalyse der Bilder bei möglichst geringem Informationsverlust etabliert [Reinhardt, 2011, S. 24-25]. B-Human nutzt hierfür ein dynamisches Scanliniengitter, das `ScanGrid`, dessen Auflösung an Position und Ausrichtung der Kamera angepasst ist. Dieses definiert horizontal und vertikal über das Bild verlaufende Scanlinien, deren zugehörige Bildpixel analysiert werden, während nicht auf den Scanlinien liegende Pixel für die Linienerkennung zunächst nicht verwendet werden. Wie in Abbildung 2.1) zu sehen ist, werden nah am Roboter auch innerhalb der Scanlinien Punkte übersprungen. In höherer Entfernung wird die Dichte der Scanlinien erhöht, wobei dafür in vertikaler Richtung zusätzliche kurze Scanlinien eingefügt werden, die nur den Fernbereich abdecken. Bereiche des Bildes, in denen der Roboter sich selbst sieht oder die über dem Horizont liegen, werden von den Scanlinien nicht abgedeckt. Haben zwei auf einer Scanlinie benachbarte Punkte unterschiedliche Farbklassen, wird die genaue Stelle, an der sich die Farbklasse ändert, ermittelt. Dadurch werden die Scanlinien in Regionen eingeteilt. Die Ergebnisse der Scanliniensegmentierung werden in den `ColorScanLineRegions` Repräsentationen gespeichert, wovon es je eine gibt für horizontale und vertikale Scanlinien. [Röfer et al. , 2019, S. 60]

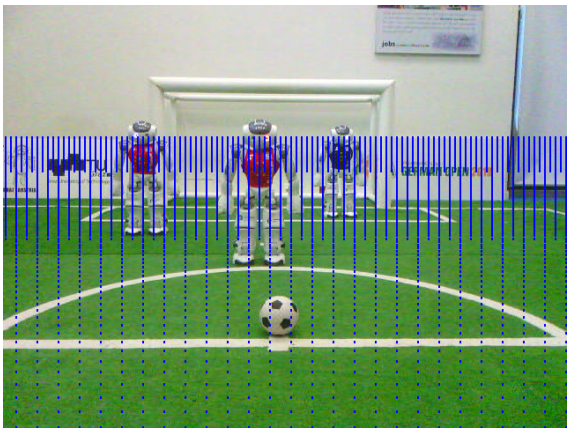


Abbildung 2.1: Scangitter der oberen Kamera

Für die Liniendetektion wird in den Scanlinien nach weißen Regionen passender Größe gesucht, die von grünen Regionen umgeben sind und sich nicht in einem zuvor erkannten Roboter befinden. Für Scanlinienregionen, die diese Kriterien erfüllen, wird der Mittelpunkt der Region im Bild berechnet und als sogenannter `LineSpot` abgespeichert. Zusätzlich wird er auf die Feldebene projiziert, was die Position des `LineSpot`s auf dem Feld relativ zum Roboter ergibt.

Dann wird der neu gefundene `LineSpot` mit allen bereits vorhandenen Linienkandidaten daraufhin abgeglichen, ob er die Linie verlängert. Als nächstes wird überprüft, ob nach dem Hinzunehmen des `LineSpot`s zum Linienkandidaten, die in das Kamerabild zurück projizierte Linie immer noch größtenteils aus weißen Pixeln besteht. Falls das zutrifft, wird der `LineSpot` in den Linienkandidaten aufgenommen. Ansonsten wird versucht mit anderen in der Nähe liegenden `LineSpot`s einen neuen Linienkandidaten zu beginnen. Zugleich werden, ebenso wie für die geraden Linien, `LineSpot`s die zusammen einen Bogen bilden zu Kandidaten für den Mittelkreis zusammengefasst. Nachdem alle Scanlinienregionen durchlaufen wurden, werden alle Linienkandidaten mit einer bestimmten Mindestlänge als Linien akzeptiert. Abschließend wird noch einmal versucht alle gefunden Linien in beide Richtungen zu verlängern, indem von den Endpunkten der Linien aus im Bild weitergegangen und überprüft wird, ob die so gefundenen Pixel ebenfalls als Weiß klassifiziert wurden. [Röfer et al. , 2019, S. 66]

Um nun den Mittelkreis zu finden, werden alle Mittelkreiskandidaten, die einen Mittelkreis in der richtigen Größe ergeben und aus einer Mindestanzahl an `LineSpot`s bestehen, in das Kamerabild zurückportiert und auf ihren Weißanteil überprüft. Ist damit noch kein Mittelkreis gefunden worden, werden aus den `LineSpot`s der gefundenen Linien mögliche Mittelpunkte des Mittelkreises berechnet. Dieses Vorgehen basiert darauf, dass Ausschnitte des Mittelkreises häufig als gerade Linie detektiert werden, die `LineSpot`s aber dennoch einen Bogen um den Mittelpunkt bilden. Aus allen Tripeln in einer Linie aufeinanderfolgender `LineSpot`s wird auf dieser Annahme die zugehörige Position des Mittelkreises errechnet. Bildet sich dabei ein großes Cluster, wird von dieser Position ausgehend der Verlauf des vermuteten Mittelkreises im Kamerabild berechnet und überprüft, ob er den Mindestanteil weißer Pixel enthält. Falls auch hiermit der Mittelkreis nicht gefunden wird, geht das Programm davon aus, dass der Mittelkreis nicht im Bild ist. [Röfer et al. , 2019, S. 66]

2.9 Linienerkennung bei anderen Teams

Die grundlegende Arbeit zum Thema kalibrierungsfreie Bildverarbeitung in der SPL stammt von Thomas Reinhardt (2011, Reinhardt [2011]) vom Nao-Team HTWK. Zur Linienerkennung verwendet Reinhardt ebenfalls ein Scanlinien-Verfahren, aber ohne Schwellwerte zur Farbsegmentierung. Um die Scanlinien in Bereiche einzuteilen, werden stattdessen Helligkeitsübergänge an den Objektgrenzen gesucht. Diese ergeben sich aus lokalen Extrempunkten der Helligkeitsänderung, die einen Schwellwert zum Ausfiltern von Bildrauschen übersteigen. Für jeden Bereich wird dann ein repräsentativer Farbwert ermittelt und anhand der aus einem vorherigen Schritt bereits bekannten Feldfarbe klassifiziert, ob dort das Spielfeld zu sehen ist oder nicht. Die übrigen Bereiche werden als Linienkandidaten klassifiziert, wenn sie zwischen zwei Feldbereichen liegen und ihre Helligkeit höher als die der angrenzenden Bereiche ist. [Reinhardt, 2011, S. 57-62]

Im Gegensatz zur Linienerkennung bei B-Human betrachtet Reinhardt die beiden Endpunkte (im Folgenden Kantenpunkte genannt) eines Linienkandidatens auf einer Scanlinie getrennt. Für beide Kantenpunkte berechnet er mit einem Sobelfilter den Helligkeitsgradienten, also einen Vektor orthogonal zur Verlaufsrichtung der vermuteten Linie. Die Genauigkeit des Helligkeitsgradienten wird anschließend noch einmal verbessert, indem aus allen nahe gelegenen Kantenpunkte derjenige mit der geringsten Winkelabweichung gesucht wird. Da beide wahrscheinlich auf der selben Linie liegen, können beide zu einem Paar zusammengefasst und der Gradient gemittelt werden. [Reinhardt, 2011, S. 70-78]

Daraufhin werden die Kantenpunktpaare miteinander zu Liniensegmenten zusammengefasst.

Dies erfolgt immer dann, wenn die mittleren Kantenpunkte einen geringen Abstand zu einer durch die beiden äußeren Kantenpunkte verlaufenden Gerade haben. Aus den Kantenpunkten eines Liniensegments wird eine Ausgleichsgerade berechnet, die den Verlauf der Linienkante im Bild widerspiegelt. Zum Finden des Mittelkreises werden zusätzlich Krümmungsanalyse und Ellipsenmatching angewendet, deren genaue Umsetzung allerdings nicht beschrieben ist [Reinhardt, 2011, S. 78-80]. Auch vier Jahre später wurde beim Nao-Team HTWK die gleiche Linienerkennung mit nur geringfügigen Änderungen weiterverwendet Schwarz et al. [2015].

Ein anderer Ansatz stammt aus der RoboCup Humanoid League von Team NimbRo (2015). Hier werden humanoide Roboter verwendet, die von den Teams selbst gebaut werden. Die verwendete Hardware ist in diesem Fall recht ähnlich zum NAO mit ungefähr der gleichen verfügbaren Rechenzeit und gleicher Kameraauflösung, wie B-Human sie für die obere NAO-Kamera verwendet. Der Öffnungswinkel der verwendeten Kamera ist hier allerdings deutlich höher, was zu einer starken Bildverzerrung führt. Farazi et al. [2015]

Zur Linienerkennung werden auf einem Graustufenbild mit einem Canny Kantendetektor Helligkeitsübergänge gesucht. Diese werden mittels einer progressiven probabilistischen Hough Transformation, einer auf Echtzeitanwendungen ausgelegten Variante der probabilistischen Hough Transformation Galambos et al. [2000], zu Liniensegmenten einer gewissen Mindestlänge zusammengefasst. Von den Liniensegmenten werden alle als echt akzeptiert, die über weiße Pixel verlaufen, sowie feldfarbene Pixel und Pixel mit Ausschlägen im Kantenbild zu beiden Seiten haben. Die Feldfarbe ist hier allerdings von Hand konfiguriert. Abschließend werden die akzeptierten Liniensegmente in roboterrelative Feldkoordinaten übertragen und zusammenpassende Liniensegmente zu längeren Linien zusammengefasst. Linien unterhalb einer bestimmten Länge werden außerdem daraufhin getestet, ob sie im Bild eine zum Mittelkreis passende Krümmung aufweisen. Farazi et al. [2015] Inzwischen verwendet NimbRo eine DeepLearning Architektur, die viele Bildverarbeitungsschritte vereint und ihre bisherigen Ergebnisse übertrifft. Allerdings steht ihnen inzwischen auch mehr Rechenleistung bereit, weshalb sich das neue Vorgehen nicht auf den NAO übertragen lässt Rodriguez et al. [2019].

2.10 Verwandte Themengebiete

Fußballanalysesysteme (zur Analyse von Spielen zwischen Menschen) dienen dazu interessante Spielszenen zu identifizieren oder Statistiken zu erstellen. Eine der dafür häufig benötigten Aufgaben ist, die Spielfeldlinien in TV-Aufzeichnungen zu finden. Dies ist insofern einfacher als die Linienerkennung im Roboterfußball, als dass die Kamera dabei fast immer von schräg oben an der Längsseite des Feldes filmt, wodurch mögliche Verlaufsrichtungen von Spielfeldlinien bereits im Vorhinein bekannt sind. Ebenso steht meistens mehr Rechenzeit je Bild zur Verfügung.

In für Fußballanalysesysteme publizierten Ansätzen zur Linienerkennung wird meistens vorausgesetzt, dass die Feldfarbe bereits bekannt ist. Sie wird etwa durch Bestimmung der dominanten Farbe über das gesamte Video ermittelt, oder für Echtzeitsysteme wird die Annahme gemacht, dass sie die dominante Farbe im aktuell analysierten Bild ist. Zur Linienerkennung wurden verschiedene Ansätze vorgeschlagen. In Cai u. Tai [2005] wird Segmentierung der Bilder in Grün und Weiß mit anschließender Verbesserung der Segmentierung durch Heuristiken vorgeschlagen. Eine weitere Methode ist nur die Bildregionen zu betrachten, die durch Öffnung (Erosion gefolgt von Dilatation) des Graustufenbildes entfernt werden würden. Daraus können, anhand von Feldfarbe und Spielererkennung, Spieler und die Umgebung des Feldes entfernt werden, was nur die Linien übrig lassen soll Sun u. Liu [2009]. Wie in Jiang

Bu et al. [2011] festgestellt, sind auch Kantenpixelerkennung als Eingabe in eine Hough Transformation oder den Random Sample Consensus Algorithmus gängige Vorgehensweisen, wobei sie selbst zur Kantendetektion zwei Laplacian-of-Gaussian Filter mit unterschiedlicher angesprochener Kantendicke verwenden und daraus ein zur Linienerkennung besser geeignetes Kantenbild zusammensetzen.

Eine recht ähnliche Aufgabe zur Spielfeldlinienerkennung ist auch die Fahrspurerkennung in Fahrassistenzsystemen oder beim autonomen Fahren. Zu den sich überschneidenden Herausforderungen gehören sich wechselnde Beleuchtungsbedingungen, Schatten, unterbrochene Linien (durch Verdeckung oder gestrichelte Spurlinien) und Echtzeitauswertung der Bilder. Im Unterschied zur Spielfeldlinienerkennung gibt es bei Straßenmarkierungen auch gekrümmte Linien, beispielsweise in Kurven. Diese gibt es in der SPL nur in Form des Mittelkreises, dessen Krümmung konstant und genau bekannt ist.

In Cario et al. [2017] werden als gängige Vorverarbeitungsschritte Erhöhung des Kontrast und Bildglättung zur Reduktion von Bildrauschen angeführt. Für die Detektion von Liniensegmenten werden mit dem Sobel- oder Prewittoperator Kanten im Bild identifiziert, die mit einer Kantenverteilungsfunktion oder der Hough Transformation verschiedenen Liniensegmenten zugeordnet oder wieder verworfen werden. Aus den zu einer Linie gehörenden Kantenpunkte wird dann mittels einer Fitting Methode der tatsächliche Linienverlauf berechnet. Aufgrund möglicher Linienkrümmung, wird dabei zwischen einem Nahbereich in welchen Fitting auf einen geraden Linienverlauf ausreicht und einem Fernbereich, wo beispielsweise ein linear parabolischer Verlauf angenommen wird, unterschieden.

Einen umfassenden Überblick über weitere Methoden und aktuelle Forschung liefern Bar Hillel et al. [2014] und Narotea et al. [2018].

3. Implementierung

Dieses Kapitel beschreibt die im Rahmen der Arbeit entwickelte Software. Abschnitt 3.1 Evaluationstool erklärt das für die Evaluation erstellte Testprogramm, die weiteren Abschnitte befassen sich mit dem für die Linienerkennung entwickeltem Code. In 3.2 Relative Feld- und Linienfarbenbestimmung werden Hilfsfunktionen beschrieben, welche nachfolgend in den Abschnitten über 3.3 Kalibrierungsfreie Farbsegmentierung und 3.4 Anpassungen am Linienerkennungsmodul mehrfach verwendet werden.

3.1 Evaluationstool

Um die nachfolgend entwickelten Änderungen am bisherigen Linienerkennungsverfahren mit möglichst geringem Aufwand testen zu können, habe ich zunächst ein Testprogramm entwickelt. Dieses ermöglicht häufiges teil-automatisiertes Testen, was die Entwicklung beschleunigt und die Reproduzierbarkeit der Evaluation verbessert. Das Testprogramm bewertet dabei nicht, ob die gefundenen Linien korrekt sind, sondern vergleicht die mit dem neuen Verfahren erkannten Linien mit den bisher gefundenen. Dadurch werden keine von Hand gelabelten Datensätze benötigt, welche derzeit nicht vorliegen und daher erst hätten erstellt werden müssen. Stattdessen verwendet das Programm Aufzeichnungen vergangener Test- und Turnierspiele. Da auch einige andere Teile der Bildverarbeitung bei B-Human demnächst verändert werden sollen, habe ich entsprechende Erweiterbarkeit bei der Entwicklung des Programms berücksichtigt.

Die ursprüngliche Idee war, die in den Logdateien ebenfalls enthaltenen Ergebnisse der Linienerkennung als Referenzdaten zu verwenden, da diese ebenfalls während des Spiels aufgezeichnet werden. Das hätte jedoch den Nachteil, dass alte Logdateien, in denen inzwischen geänderte Versionen der getesteten Module verwendet wurden, nicht zum Testen genutzt werden könnten. Zudem führt das Verwenden der im Spiel aufgezeichneten Linienerkennungsergebnisse noch zu einem weiteren Problem: Vor dem Abspeichern in der Logdatei werden die vom Roboter aufgenommenen Bilder im JPEG-Format verlustbehaftet komprimiert. Das führt teilweise dazu, dass der identische Programmcode bei erneuter Ausführung auf dem komprimierten Bild zu etwas anderen Ergebnissen kommt als bei der ursprünglichen Ausführung auf dem Roboter. Entsprechend stellt das Testprogramm in diesem Fall einen Unterschied fest, was die Auswertung verkomplizieren würde.

Daher wird stattdessen zunächst ein bewährtes Modul ausgeführt, welches Referenzdaten generiert – im Rahmen dieser Arbeit ist das immer der bisherige `LinePerceptor`. Anschließend wird das zu testende Modul ausgeführt und dessen Ergebnisse werden mit den Referenzdaten abgeglichen. Aufgrund von Beschränkungen im B-Human Framework ist es derzeit nicht möglich in einem Durchlauf einer Logdatei mehrere Versionen einer Repräsentation von verschiedenen Modulen gleichzeitig bereitstellen zu lassen. Es werden entsprechend zwei Durchläufe je Logdatei gemacht. Mit diesem Ansatz ist es weiterhin möglich die in der Aufzeichnung abgelegten Repräsentationen als Referenzdaten zu benutzen, indem als Referenzmodul der `LogDataProvider` genutzt wird, ein Modul das sonst beim Abspielen von Logdateien im Simulator genutzt wird.

Da die Spielaufzeichnungen mit 10 Minuten je Halbzeit plus der Ready und Set Phasen bis zum ersten Anstoß etwa 20.000 Bilder je Kamera enthalten, ist es nötig diese zuerst auf eine geeignete Größe zu reduzieren. Das Evaluationsprogramm enthält daher die Möglichkeit

in einem vom Nutzer vorgegebenen Abstand Bilder und die zugehörige Ergebnisse aus der Aufzeichnung zu filtern, während die übrigen Daten entfernt werden. Standardmäßig wird jedes 60-ste Bild einer Kamera verwendet, wodurch sich jeweils etwa 330 Testbilder je Kamera und Aufzeichnung ergeben.

Beim Vergleich der gefundenen Linien ist zu beachten, dass nicht auf exakte Gleichheit getestet werden kann, da es immer zu kleinen Unterschieden zwischen den erkannten Linien kommen wird, insbesondere bei der Länge der erkannten Linienstücke. Das im Test verwendete Gleichheitskriterium fordert daher zunächst nur eine ähnliche Verlaufsrichtung und geringe orthogonale Verschiebung gegenüber der Verlaufsrichtung ein. Dieses zweite Kriterium ist notwendig, um etwa Verwechselungen von Grundlinie und Goalbox zu verhindern. Die Abstände werden daher auch nicht in Bildpixeln, sondern in den tatsächlichen Entfernungen auf dem Spielfeld gemessen. Demgegenüber dürfen Abstände der Linienendpunkte zwischen aufgezeichneten und neu berechneten Linien recht groß sein. Es wird jedoch immer sichergestellt, dass die beiden Linien sich zu mindestens 50 % überschneiden müssen. Insgesamt werden zwei `LinesPercept`e immer dann vom Testprogramm als gleich angesehen, wenn es für jede ursprünglich gefundene Linie in der neuen Erkennung eine nach den obigen Kriterien gleiche Linie gibt und umgekehrt.

Wird mindestens ein Unterschied festgestellt, speichert das Programm ein Bild ab, auf dem die Unterschiede eingezeichnet sind. In den Referenzdaten vorhandene aber nicht erkannte Linien werden rot markiert, während zusätzlich gefundene Linien in blau dargestellt werden. Linien die sowohl in den Referenzdaten sind, als auch von der neuen Erkennung gefunden werden, sind aufgehellt in hellblau und rosa eingezeichnet. Zusätzlich ist der erkannte Spielfeldrand in orange eingezeichnet. Das ist beispielhaft in Abbildung 3.1 zu sehen.

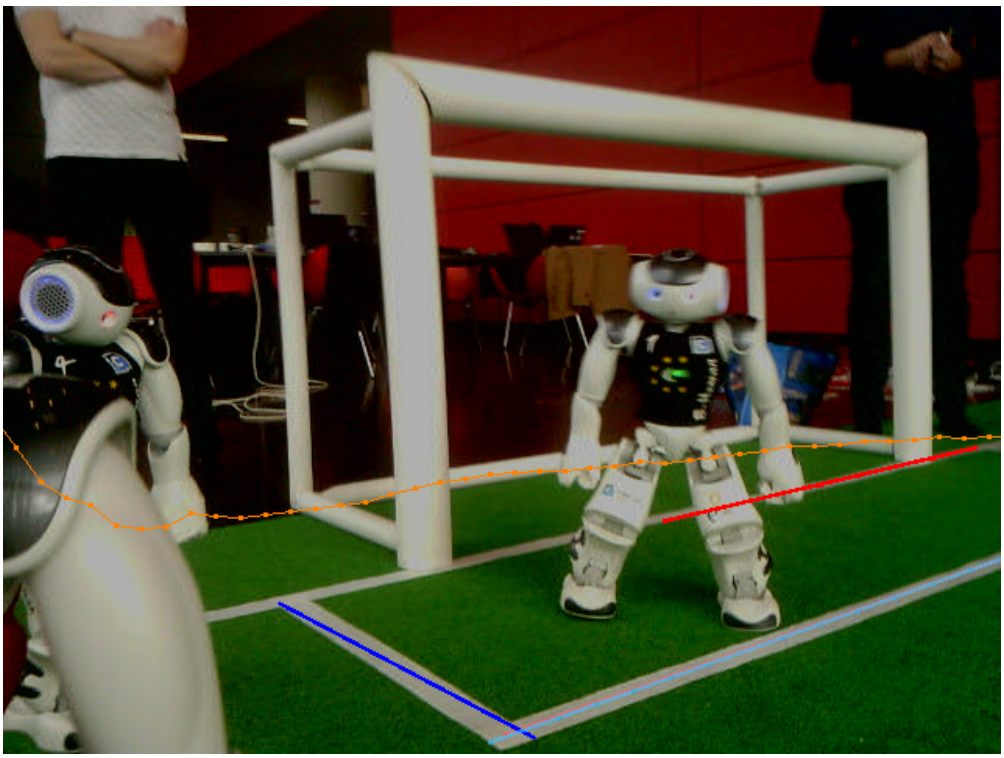


Abbildung 3.1: Kamerabild mit eingezeichneten unterschiedlich erkannten Linien

3.2 Relative Feld- und Linienfarbenbestimmung

Sowohl die Farbsegmentierung der Scanlinien (Abschnitt 3.3) als auch der neue Linienerkenner (Abschnitt 3.4) benötigen Funktionen, die Bildausschnitte, die das Spielfeld oder die Feldlinien zeigen, auseinander halten. Die mehrfach genutzten Funktionen und Parameter habe ich in einer neuen Repräsentation zusammengefasst, den `RelativeFieldColors`. Da andere Mitglieder des B-Human Teams diese Repräsentation mit entwickelt haben und an anderer Stelle einsetzen, ist der in diesem Abschnitt beschriebene Code nicht ausschließlich meine Leistung.

Die hauptsächliche Aufgabe der `RelativeFieldColors` besteht darin Funktionen bereitzustellen, die prüfen, ob ein bestimmter Bildpixel das Spielfeld oder eine Linie zeigt. Dafür wird ein gegebener HSV-Farbwert für den bekannt ist, ob es sich um Feld oder Linie handelt, mit einen neuen HSV-Farbwert, der noch klassifiziert werden muss, abgeglichen. Die erste Funktion `isWhiteNearField` beachtet dabei nur Helligkeit und Farbsättigung, da Linien beliebige Farbtöne annehmen können. Das betrachtete Helligkeit-, Sättigungspaar wird dann als Weiß deklariert, wenn es jeweils um einen festen Schwellwert heller und weniger gesättigt ist, als der Referenzfarbton. Zudem wird anhand von auf Erfahrung beruhenden Mindest- und Maximalwerten für Helligkeit und Sättigung geprüft, ob der vorgelegte Farbwert plausiblerweise Weiß sein kann. Analog dazu gibt es die Funktion `isFieldNearWhite`, die um den gleichen Schwellwert geringere Helligkeit und höhere Sättigung als die der Referenzwerte einfordert. Auch hier wird mit Erfahrungswerten für Maximalhelligkeit und Mindestsättigung geprüft, ob die Zuordnung als Feld plausibel ist. Zusätzlich gibt es eine Überladung der Funktion, die auch den zulässigen Farbtonbereich des Feldes einbezieht. Beide Funktionen setzen voraus, dass die Referenzfarbwerte von Bildpixeln nah am zuzuordnenden Pixel oder Bildbereich kommen, da die Farbvergleiche ähnliche Beleuchtung von Test- und Referenzfarbton benötigen.

Für die Plausibilitätsprüfungen habe ich hinzugefügt, dass auch die grundlegende Helligkeit und Sättigung der Bilder mit einbezogen wird. Diese werden unabhängig davon auch an anderen Stellen benötigt. Sie werden je aufgenommenem Bild vom `RelativeFieldColorsProvider` neu berechnet, dem Modul, das die `RelativeFieldColors` bereitstellt. Weil es unnötiger Rechenaufwand wäre dafür die ganzen Bilder zu betrachten, wende ich die Heuristik an, den Durchschnitt über drei horizontale Bildzeilen zu berechnen. Ich verwende dafür die Bildzeilen auf einem Viertel, der Hälfte und drei Vierteln der maximalen Höhe des Scanliniengitters. Diese Höhenbegrenzung ist sinnvoll, da sie die Berechnung der durchschnittlichen Helligkeit und Sättigung auf den Bereich des Spielfelds begrenzt.

Damit ein Farbton abschließend als Weiß akzeptiert wird, muss er heller sein, als die approximierte Durchschnittshelligkeit und gleichzeitig eine geringere Sättigung aufweisen als durchschnittlich im Bild vorhanden ist. Umgekehrt ist für eine Einstufung als Feld nötig, dass der Farbton nicht viel heller oder weniger gesättigt ist als die approximierten Durchschnittswerte des Bildes. Das beruht darauf, dass die approximierten Durchschnittswerte normalerweise ungefähr der Feldfarbe entsprechen und manchmal durch Linien oder Roboterteile etwas aufgehellt und entsättigt sind. Ausnahmen davon gibt es nur, wenn ein Großteil des Sichtfeldes von anderen Robotern oder dem Schiedsrichter eingenommen wird oder wenn der Roboter umfällt.

3.3 Kalibrierungsfreie Farbsegmentierung

B-Human's Algorithmus zur Linienerkennung verwendet als Grundlage seiner Berechnungen Scanlinien aus den farbsegmentierten Kamerabildern. Diese `ColorScanLineRegions` genannte Farbsegmentierung der Scanlinien bildet auch den Ausgangspunkt zum Identifizieren von Bildbereichen, in denen der Ball oder eine Strafstoßmarke zu sehen sein könnte. Wenn es gelingt die `ColorScanLineRegions` ohne manuelle Kalibrierung zu berechnen, ermöglicht das also nicht nur den bisherigen Linienerkennungsalgorithmus weiter zu verwenden, sondern ist auch ein wichtiger Schritt zur kalibrierungsfreien Erkennung von Ball und Strafstoßmarke. Ich habe daher ein Verfahren entwickelt, welches die `ColorScanLineRegions` nicht auf dem farbsegmentierten Kamerabildes berechnet, sondern dazu dessen HSV-Darstellung verwendet.

Das Ziel ist eine möglichst zuverlässige Farbsegmentierung der Scanlinien in die Klassen `Feld`, `Weiß` und `Anders` zu erzeugen, unter Berücksichtigung der Anforderungen an Echtzeitfähigkeit, Beleuchtungsunabhängigkeit und Kalibrierungsfreiheit. Die Evaluation der Farbsegmentierung als einzelnes Modul findet sich in Abschnitt 4.2.2, die zugehörigen Laufzeitmessungen in Abschnitt 4.3.2.

In Anlehnung an das beim Team HTWK Leipzig genutzte Verfahren Schwarz et al. [2015] habe ich den Algorithmus in vier Schritte geteilt:

1. Scanlinien in Regionen einteilen
2. Feldregionen erkennen
3. Weiße Regionen erkennen
4. Benachbarte gleichartige Regionen zusammenfassen

Die Implementierung der Schritte kann unabhängig voneinander ausgetauscht werden. Ich habe mehrere Varianten erstellt und miteinander kombiniert, um die für die Linienerkennung beste Kombination zu finden. Im Folgenden gehe ich detailliert auf die einzelnen Teilaufgaben ein und stelle die entwickelten Lösungsvarianten vor.

3.3.1 Regioneneinteilung der Scanlinien

Der erste Schritt zur Farbsegmentierung der Scanlinien ist die Einteilung in homogene Regionen. Für jede Scanlinie soll eine Liste von Bildregionen, angegeben durch Start- und Endpixel, erstellt und für jede Region repräsentative Helligkeits- und Farbwerte berechnet werden.

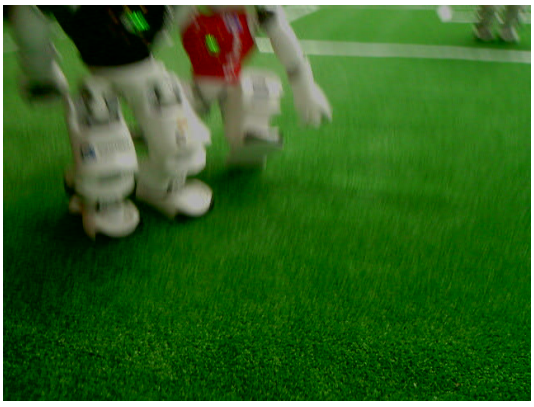


Abbildung 3.2: Kamerabild mit Bewegungsunschärfe und Rauschen

Es sollen alle Übergänge zwischen Feld, Linien, Ball und anderen Dingen gefunden werden. Wären beispielsweise Feld und Linie in der gleichen Region zu sehen, kann die Region in den folgenden Schritten nicht korrekt zugeordnet werden. Stattdessen würde sie als nur Feld oder nur Weiß markiert werden. Eine der beiden Informationen ginge verloren.

Andererseits ist auch die Einteilung gleichförmiger

Bildabschnitte in mehrere Regionen problematisch. Dies ist zwar weniger schlimm als nicht gefundene Regionsübergänge, da aufeinanderfolgende als gleich segmentierte Regionen wieder zusammengesetzt werden können, erfordert aber auch, dass die Algorithmen zum Zuordnen der Regionsklassen damit umgehen können. Zudem bedeutet eine höhere Anzahl an Regionen in den folgenden Schritten eine entsprechend höhere Laufzeit.

Das größte Problem bei der Regioneneinteilung ist die große Bandbreite der Bildschärfe. Abhängig davon, ob der Roboter sich fortbewegt und ob er den Kopf dreht, reichen die Bilder von sehr scharf bis zu vollkommen verwischt. Zusätzlich enthalten die scharfen Bilder häufig einiges Bildrauschen, einerseits durch die Aufnahmequalität der Kameras, andererseits auch durch den Untergrund auf dem gespielt wird. Der Kunstrasen sieht nämlich je nach Blickwinkel recht ungleichmäßig aus, was im Nahbereich zu starken Helligkeitsunterschieden zwischen benachbarten Pixeln führen kann, obwohl beide das Feld zeigen. Bei flacherem Blickwinkel und höherer Entfernung verschwindet dieser Effekt, wodurch weiter entfernte Spielfeldregionen weniger verrauscht sind. Wie in Abbildung 3.2 zu sehen ist es dadurch sogar möglich, dass beide Effekte gleichzeitig im selben Bild auftreten. Während der obere Teil des Bildes durch die Bewegung des Roboters verwischt ist, sind unten deutliche Helligkeitsunterschiede im Spielfeld zu sehen.

Für die Regioneneinteilung habe ich drei verschiedene Lösungen entwickelt, die im Folgenden beschrieben sind. Alle drei haben einstellbare Parameter, wie etwa Schwellwerte zur Kantendetektion. Diese müssen nur einmalig eingestellt werden und sind dann für alle Umgebungen gültig. Die letztendlich verwendeten Werte wurden als Teil der Evaluation experimentell ermittelt (siehe Abschnitt 4.2.1).

Regioneneinteilung mit Kantenfilter

Die Grundidee dieses Ansatzes besteht darin, Kanten im Graustufenbild zu identifizieren und diese als Regionsübergänge zu nutzen. Dazu wende ich auf die gesamte Scanlinie einen Kantenfilter an, welcher Kanten orthogonal zur Scanlinie findet. Als Filtermatrix verwende ich den Sobelfilter, da dieser sich auch in anderen Arbeiten zur Linienerkennung bewährt hat [Reinhardt, 2011, S. 73-74]. Entsprechend wende ich auf den horizontalen Scanlinien die Sobel_x-Matrix und auf den vertikalen Scanlinien die Sobel_y-Matrix an.

Die Laufzeit wird verbessert, indem die Filtermatrix in zwei eindimensionale Filter aufgeteilt wird. Zuerst wird der Glättungsanteil auf einer Pixelreihe quer zur Scanlinie berechnet und so lange wie nötig zwischengespeichert. Der Ableitungsanteil wird dann auf den Ausgaben des Glättungsfilters berechnet, wodurch auf kein Bildpixel mehrfach zugegriffen wird.

Glättung	Ableitung	Sobel_x
$\begin{bmatrix}1\\2\\1\end{bmatrix}$	$\begin{bmatrix}-1 & 0 & 1\end{bmatrix}$	$\begin{bmatrix}1\\2\\1\end{bmatrix} \cdot \begin{bmatrix}-1 & 0 & 1\end{bmatrix} = \begin{bmatrix}-1 & 0 & 1\\-2 & 0 & 2\\-1 & 0 & 1\end{bmatrix}$

Tabelle 3.1: Aufteilung des Sobelfilters in Ableitung und Glättung

Kanten identifiziere ich dann mittels eines Schwellwertverfahrens und einer Nichtmaximumsunterdrückung. Es werden also zunächst diejenigen Pixel als Kante akzeptiert, bei denen die Ausgabe des Sobelfilters über einem festgelegten Schwellwert liegt. Falls mehrere Pixel aufeinanderfolgend den Schwellwert übertreffen, ist davon auszugehen, dass es sich um die

selbe Kante handelt. In diesem Fall wird die Position der Kante als die des Pixels mit der höchsten Ausgabe des Sobelfilters ermittelt und die angrenzenden Pixel werden verworfen.

Erste Tests ergaben, dass das einfache Anwenden eines Sobelfilters nicht ausreicht, um zufriedenstellend mit der großen Bandbreite von Unschärfe und Rauschen umzugehen. Bei Bildern mit viel Bewegungsunschärfe erscheinen Feldlinien so verwaschen, dass der Helligkeitsanstieg den Schwellwert nicht erreicht. Innerhalb dieses Ansatzes ist es schwierig dieses Problem zu lösen. Ich konnte ihm zwar teilweise durch die Wahl eines eher niedrigen Kantenschwellwerts entgegenwirken, was die Methode jedoch anfälliger für Bildrauschen macht.

Zur Unterdrückung von Kanten, welche nur durch Bildrauschen entstehen gibt es allerdings einige Möglichkeiten. Zunächst habe ich hierfür eine Glättung des betrachteten Bildausschnitts mit einem Glättungsfilter implementiert. Um die Laufzeit zu optimieren, verwende ich ein ganzzahliges Glättungsfilter, das mit dem Sobelfilter in der gleichen Filtermatrix kombiniert ist. Die kombinierte Filtermatrix lässt sich auch wieder in zwei eindimensionale Filter aufteilen.

ganzzahliges Glättungsfilter	Sobel_x	kombiniertes Filter
$\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}$	$\begin{bmatrix} -1 & -2 & 0 & 2 & 1 \\ -4 & -8 & 0 & 8 & 4 \\ -6 & -12 & 0 & 12 & 6 \\ -4 & -8 & 0 & 8 & 4 \\ -1 & -2 & 0 & 2 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 4 \\ 6 \\ 4 \\ 1 \end{bmatrix} \cdot \begin{bmatrix} -1 & -2 & 0 & 2 & 1 \end{bmatrix}$

Tabelle 3.2: 5×5 Kantenfiltermatrix mit zusätzlicher Bildglättung

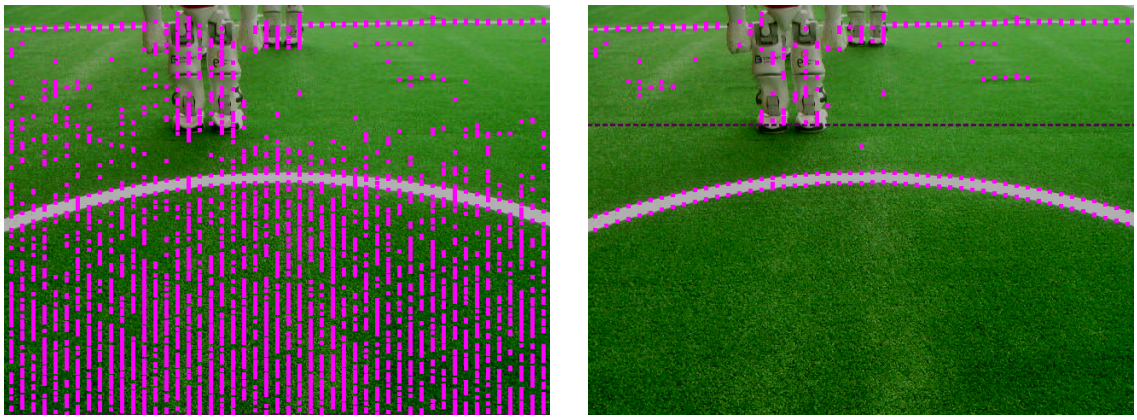
Da weiter entfernte Bildausschnitte weniger Rauschen enthalten und Linien allein dadurch, dass sie, wenn sie weit entfernt sind, nur wenige Bildpixel einnehmen, mit zusätzlicher Glättung seltener gefunden werden, ist es nicht sinnvoll die zusätzliche Glättung auf das ganze Bild anzuwenden. Stattdessen wechsele ich das verwendete Filter abhängig von der Entfernung der betrachteten Bildpixel zum Roboter.

Des Weiteren lässt sich ausnutzen, dass für die Regionenbildung hauptsächlich Übergänge zwischen dem Grün des Spielfelds und weißen Bildbereichen relevant sind. Da weiße Regionen eine geringe Farbsättigung haben, kann der Kantenschwellwert an die Sättigung der betrachteten Pixel angepasst werden. Bei geringer Farbsättigung wird der Schwellwert etwas abgesenkt, bei hoher Farbsättigung hingegen angehoben.

Ebenso lässt sich feststellen, dass Regionsübergänge in dunklen Bildpixeln meistens nur Rauschen oder anderweitig irrelevant sind und aussortiert werden können. Dafür lässt sich die in den `RelativeFieldColors` berechnete Grundhelligkeit wiederverwenden (siehe 3.2). Kanten auf Bildpixeln deren geglätteter Helligkeitswert deutlich unter der Grundhelligkeit liegt, werden dann nicht als Regionsübergang akzeptiert. Da Linien hell sind, können damit keine Regionsübergänge zwischen Feld und Linie herausgefiltert werden. Theoretisch können Regionsübergänge zwischen Feld und Ball verloren gehen, da die Unterseite des Balls meistens wenig Licht bekommt. Ebenso können die Regionsgrenzen zwischen Feld und dunklen Robotertrikots oder Kleidung von Schiedsrichtern verloren gehen. Falls diese einen größeren Anteil des Bildes ausmachen, kann das die ermittelte Feldfarbe verfälschen.

In der folgenden Abbildung 3.3 sind die gefundenen vertikalen Regionsgrenzen auf einem stark rauschbehaftetem Bild dargestellt. Links sind die Regionsgrenzen bei alleiniger Verwendung

eines Sobelfilters eingezeichnet, rechts die Regionsgrenzen mit den beschriebenen Techniken zur Rauschfilterung. Die gestrichelte dunkelviolette Linie zeigt dabei, bis wohin zusätzliche Bildglättung angewendet wird.



(a) Regionsübergänge ohne Rauschfilterung (b) Regionsübergänge mit Rauschfilterung

Abbildung 3.3

Die identifizierten Regionen lege ich in einer Datenstruktur zum Zwischenspeichern ab. Sie speichert die räumliche Anordnung durch Einsortierung der Regionen in eine zweidimensionale Array-Struktur und Festlegung der Start- und Endpixel der Region. Zusätzlich werden repräsentative HSV-Farbwerte für die Region ermittelt. Das geschieht mittels Durchschnittsbildung auf einem Viertel der Pixel der Region, wobei Regionen mit weniger als 20 Pixeln gesondert behandelt werden. Für diese geht ein höherer Anteil der Bildpixel in die HSV-Farbwerte ein, damit er auch für kleine Bildregionen repräsentativ ist.

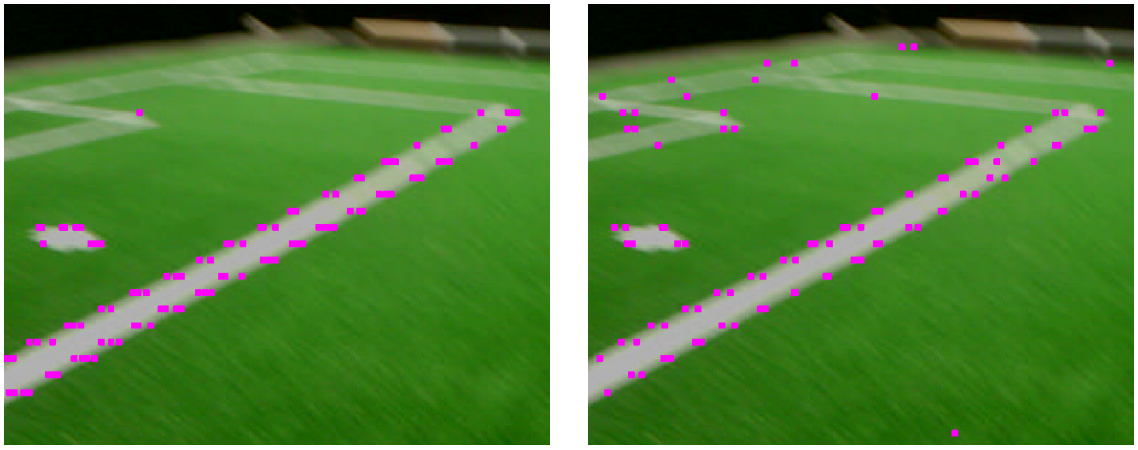
Regioneneinteilung mit Kantenfilter auf Scangitter

Der oben beschriebene Ansatz, welcher nur Kantenfilterung verwendet, hat große Probleme beim Finden sanfter Helligkeitsverläufe. Wenn die Roboter in Bewegung sind, kommen diese jedoch häufig vor, besonders bei schnellen Kopfbewegungen. Die durch die Eigenbewegung des Roboters hervorgerufene Bewegungsunschärfe in den Bilder verschmiert den Übergang zwischen Feld und Linie oft über mehrere Pixel. In der Folge wird je nach Höhe des Kantenschwellwerts der Übergang nicht gefunden oder es entstehen mehrere Regionsübergängen wo nur einer sein sollte, wie in Abbildung 3.4a.

Als Lösung dafür verwende ich das Scangitter aus der ursprünglichen Erstellung der Scanlinien wieder (siehe Abbildung 2.1 in Abschnitt 2.8). Ich berechne in diesem Ansatz die Bildglättung zunächst auf dem Scangitter und vergleiche dann die geglättete Helligkeit benachbarter Scangitterpunkte. Bei einem Helligkeitsunterschied über dem Schwellwert wird der höchste Ausschlag des Sobelfilters zwischen den beiden Scangitterpunkten zum Identifizieren des Übergangspunktes genutzt. Die einzelnen Scangitterpunkte liegen dabei eng genug beisammen, dass auf jeder Feldlinie mindestens ein Scangitterpunkt je Scanlinie liegt. Sie sind jedoch weit genug von einander entfernt, dass sich auch bei unscharfen Linien meistens ein klarer Helligkeitsanstieg ergibt. Mehrere dicht aufeinanderfolgende Regionsübergänge sind zwar auch damit nicht ausgeschlossen, ihre Anzahl ist aber durch das Scangitter begrenzt.

Wie in Abbildung 3.4b zu sehen ist, treten noch maximal zwei Regionsübergänge für eine echte Linienkante auf. Zusätzlich detektiert das Scangitterverfahren auch weitere Linienkanten im

Hintergrund. Aufgrund des geringen Helligkeitsanstiegs werden sie vom einfachen Kantenfilter nicht gefunden, während der Helligkeitsvergleich über die etwas höhere Distanz des Scangitters einen für die Detektion ausreichende Helligkeitsdifferenz bewirkt.



(a) Regionsübergänge des Kantenfilters

(b) Regionsübergänge des Kantenfilters auf dem Scangitter

Abbildung 3.4: Vergleich der horizontalen Regionsübergänge zwischen simplem Kantenfilter und Scangitterverfahren auf einem unscharfen Bildausschnitt, jeweils mit gleichem geringen Kantenschwellwert und Rauschfilterung

Als zusätzliche Verbesserung, mit der weitere Übergänge zwischen Feld und Linien gefunden werden, kann auch die Farbsättigung der Bilder mit einbezogen werden. Dazu wird die Glättung auch auf dem Sättigungsbild berechnen und dann von der geglätteten Helligkeit abgezogen. Da das Feld im Vergleich zu den Linien eher dunkel ist und eine hohe Sättigung aufweist, die Linien hingegen hell und niedrig gesättigt sind, erhöht das den Kontrast mit dem Linien sich abheben. Allerdings benötigt das Berechnen des Glättungsfilters für die Sättigung entsprechend zusätzliche Rechenzeit und erzeugt auch zusätzliche Übergänge innerhalb des Feldes durch Sprünge der Sättigung im Feld.

Das Einbeziehen der Sättigung kann in dieser Form auch im ersten Ansatz, der nur Kantenfilterung verwendet, angewendet werden. Dort wird dann der Sobelfilter von Helligkeit abzüglich Sättigung berechnet, mit den gleichen Vor- und Nachteilen, wie eben beschrieben.

Ich habe die Implementierung der am Scangitter orientierten Vorgehensweise hinsichtlich der Laufzeit nicht weiter optimiert. Wenngleich es möglich wäre den Sobelfilter nur dann zu berechnen, wenn zwischen zwei Scangitterpunkten ein Helligkeitsunterschied über dem Schwellwert festgestellt wird, berechne ich ihn, wie bei der ersten Methode, für alle Pixel der Scanlinie.

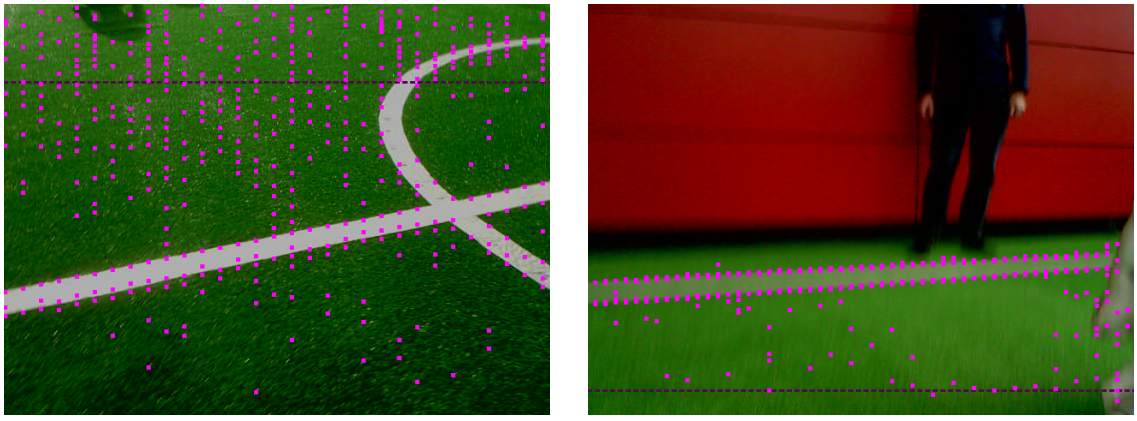
Inkrementelle Regionenbildung mit Farbähnlichkeitsschwellwerten

Eine weitere Möglichkeit die Regioneneinteilung zu erhalten ist sich direkt auf die Regionen selbst zu konzentrieren. In diesem Ansatz baue ich die Regionen Pixel für Pixel auf und beende eine Region dann, wenn der Farbunterschied eines neuen Pixels zur restlichen Region zu groß wird. Das hat auch zum Vorteil gegenüber der auf Regionsübergänge fokussierten Methoden, dass zu diesem Zeitpunkt die repräsentativen Farbwerte der Region bereits ermittelt sind und alle Pixel der Region darin eingehen.

Es wird über alle Pixel der Scanlinie iteriert. Bei jedem neuen Pixel wird zunächst geprüft, ob die Mindestregionsgröße erreicht ist. Falls nicht wird das Pixel vorbehaltlos in die Region aufgenommen und die Farbwerte der Region werden aktualisiert. Der Verzicht auf eine Zugehörigkeitsprüfung beim Beginn einer neuen Region dient dazu, ansonsten durch Bildrauschen entstehende Regionsübergänge zu vermeiden. Die Mindestregionsgröße ergibt sich dabei aus dem Abstand der Scangitterpunkte, über welche die Scanlinie verläuft.

Ist die Mindestregionsgröße erreicht wird überprüft, ob das Pixel ähnliche Farbwerte hat, wie die bisher aufgebaute Region. Dazu wird mit Schwellwerten für Helligkeit, Farbton und Sättigung geprüft, ob die Unterschiede innerhalb eines Toleranzbereiches bleiben. Bei geringer Helligkeit der bisherigen Region sind dabei höhere Sättigungsunterschiede erlaubt und analog dazu bei geringer Sättigung höhere Farbtonunterschiede zulässig. Falls die Unterschiede alle innerhalb des Toleranzbereichs liegen, wird das Pixel wie gehabt in die Region aufgenommen und zum nächsten Pixel gewechselt. Ansonsten wird die aktuelle Region beendet und das Pixel, welches außerhalb des Toleranzbereichs lag, beginnt eine neue Region.

Wie auch schon bei der Verwendung des Kantenfilters, bietet es sich an zur Rauschverminderung im Nahbereich zusätzlich zu glätten. Hier habe ich die Nahbereichsglättung mittels eines eindimensionalen ganzzahligen Gaussfilters implementiert, der quer zur Scanlinie angewendet wird. Die Technik des Unterdrückens von Regionsübergängen an dunkeln Pixeln deutlich unter der Grundhelligkeit des Bildes lässt sich hier ebenfalls wiederverwenden. Wie Abbildung 3.5a zeigt, bleibt aber eine hohe Anfälligkeit für helles Bildrauschen, durch Reflexion von Sonnenlicht am Kunstrasen. Dafür werden aber auch stark verwischte Linien mit einem geringen Helligkeitsanstieg gegenüber dem Feld gut erkannt, was in Abbildung 3.5b zu sehen ist. Die gestrichelte dunkelviolette Linie zeigt wieder bis wohin zusätzliche Glättung verwendet wird.



(a) Unnötige Regionsübergänge bei hellem Rauschen

(b) Regionsübergänge auf einem verwischten Bild

Abbildung 3.5: Durch inkrementelle Regionenbildung berechnete vertikale Regionsgrenzen

3.3.2 Segmentierung des Spielfelds

Die hauptsächlichen Erkennungsmerkmale des Spielfelds sind die grüne Farbe und die Tatsache, dass es als zusammenhängende Fläche sichtbar ist. Zudem nimmt es meistens einen Großteil des Sichtfeldes ein. Der genaue Bereich von Farbton und Sättigung kann je nach Material und Beleuchtung stark variieren.

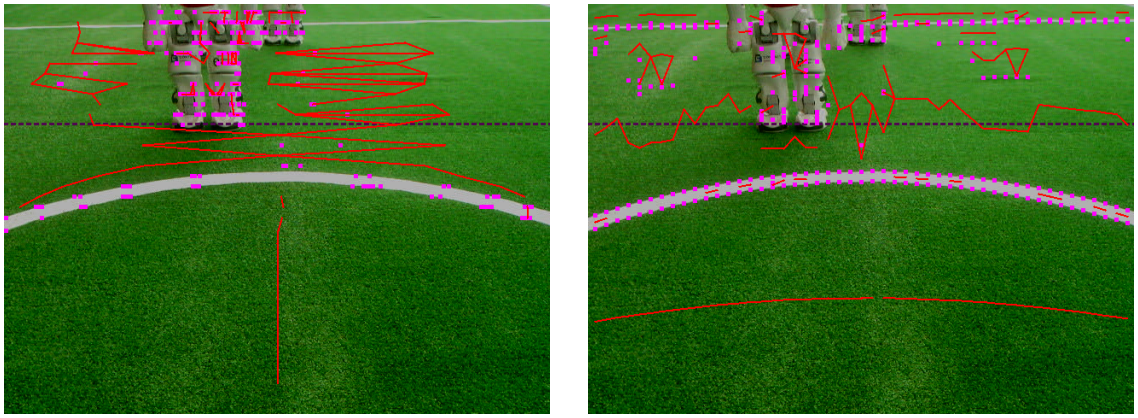
Um zuverlässig bestimmen zu können, in welchen Scanlinienregionen das Spielfeld zu sehen

ist, suche ich nach homogenen, zusammenhängenden Regionen im richtigen Farbbereich. Dafür habe ich die interne Repräsentation einer Scanlinienregion so erweitert, dass sie als Element eines Union-Find Waldes dienen kann. Beim Vereinigen zweier Regionen werden in dem Element, das als Elternknoten dient, die Farbwerte gewichtet nach den Regionsgrößen aktualisiert und die Gesamtregionsgröße gespeichert.

Damit überprüfe ich für alle benachbarten Scanlinienregionen, ob sie farblich ähnlich sind und vereinige sie gegebenenfalls. Bei diesem Ablauf muss zwischen den horizontalen und vertikalen Scanlinien unterschieden werden, da die vertikalen Scanlinien nicht alle gleich lang sind. In beiden Fällen wird durch alle Scanlinienregionen iteriert und überprüft, ob die aktuelle Region mit der nächsten Region auf der selben Scanlinie vereinigt werden soll. Anschließend wird versucht die aktuelle Region mit den benachbarten Regionen auf angrenzenden Scanlinien zu vereinigen. Bei den horizontalen Scanlinien ist dazu nur die direkt nächste Scanlinie relevant. Es werden alle Regionen der nächsten Scanlinie auf Ähnlichkeit überprüft und gegebenenfalls mit der aktuellen vereinigt, bis eine Region erreicht ist, deren Ende über das Ende der aktuellen Region hinausragt. Diese Region wird zwischengespeichert und als Startpunkt verwendet, wenn die nächste Region auf der selben Scanlinie mit Regionen auf der nächsten Scanlinie vereinigt wird.

Bei den vertikalen Scanlinien funktioniert dies nicht so einfach, da durch die unterschiedlichen Längen mehrere der folgenden Scanlinien benachbart sein können. Daher werden zuerst alle zur aktuellen Scanlinie benachbarten Regionen auf den nächsten vier Scanlinien identifiziert und nach Regionsende sortiert zwischengespeichert. Die zwischengespeicherten Regionen werden dann behandelt, als lägen sie auf der selben Scanlinie, wodurch danach das gleiche Vorgehen, wie bei den horizontalen Scanlinien genutzt werden kann.

Ich vereinige zwei benachbarte Regionen immer dann, wenn ihre Abweichung in Helligkeit, Farbton und Sättigung jeweils innerhalb festgelegter Toleranzbereiche bleibt. Auch diese Parameter wurden in 4.2.1 bei der Bestimmung optimaler Parameter der Scanliniensegmentierung experimentell bestimmt und gelten für alle Beleuchtungsbedingungen. In Abbildung 3.6 zeigen die roten Verbindungslinien welche Regionen vereinigt wurden. Die Verbindungslinien sind durch die Mittelpunkte der Scanlinienregionen gezeichnet.



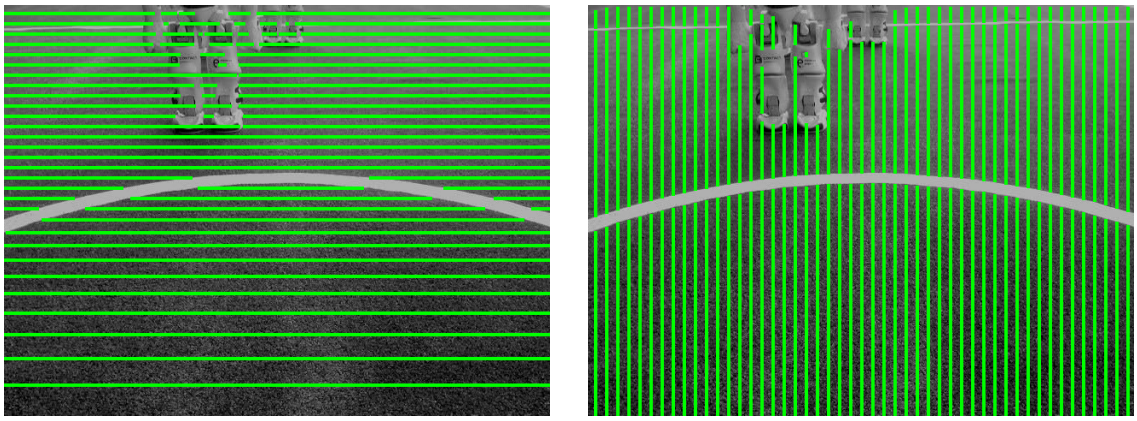
(a) Vereinigung horizontaler Regionen (b) Vereinigung vertikaler Regionen

Abbildung 3.6: Vereinigung benachbarter ähnlichfarbener Regionen

Nach Abschluss aller Vereinigungen deklariere ich alle vereinigten Regionen als `Feld`, die zusammen eine bestimmte Mindestgröße haben und deren Sättigung und Farbton im Bereich

der für das Spielfeld möglichen Werte liegen. Dabei wird auch der tatsächliche Farbbereich und die Mindestfarbsättigung des Spielfeldes im konkreten Bild festgestellt. Da es durch Schatten, ungleichmäßig geborstenen Rasen und ähnliches mehrere Feldfarbbereiche im selben Bild geben kann, verwende ich dazu die minimalen und maximalen Farbtonwerte und den minimalen Sättigungswert der gefundenen Feldregionen. Der Bereich der zulässigen Farb- und Sättigungswerte des Spielfelds erweitere ich dann noch einmal abhängig von der Anzahl der gefundenen Feldregionen. Je weniger verschiedene, nicht verbundene Feldregionen zu diesem Zeitpunkt bereits gefunden wurden, desto mehr wird der zulässige Bereich erweitert. Abschließend gleiche ich alle übrigen Scanlinienregionen mit den ermittelten Farbwerten des Spielfeldes ab und klassifiziere sie, falls sie übereinstimmen, ebenfalls als `Feld`.

Die Ergebnisse sind in Abbildung 3.7 dargestellt. Hier werden die Roboter zum Teil fälschlicherweise als Feld klassifiziert. Das geschieht einerseits wegen schlecht gesetzter Regionsgrenzen und andererseits weil der als Feld akzeptierte Farbbereich recht groß wird. Die Feldlinien werden hier hingegen nicht als Feld klassifiziert.



(a) horizontale Feldregionen (b) vertikale Feldregionen

Abbildung 3.7: Die als Feld erkannten Scanlinienregionen. Zur besseren Erkennbarkeit ist das Ursprungsbild in Graustufen dargestellt.

Für den Fall, dass einmal keine der vereinigten Region den Anforderungen an Mindestgröße und möglicher Farbwerte des Feldes entsprechen, speichere ich die zuletzt ermittelte Feldfarbe zusammen mit einem Zeitstempel zwischen. Ist seit dem letzten erfolgreichen Ermitteln der Feldfarbe wenig Zeit vergangen, kann die zuletzt ermittelte Feldfarbe wiederverwendet werden um alle Scanlinienregionen mit ihr abzugleichen. Ansonsten gehe ich davon aus, dass im Bild sehr wenig oder gar nichts vom Spielfeld zu sehen ist und klassifiziere keine Region als `Feld`.

3.3.3 Segmentierung weißer Bildregionen

Der dritte Schritt der Scanliniensegmentierung ist die Klassifizierung weißer Bildregionen. Wenngleich für die Linienerkennung nur das Weiß der Feldlinien relevant ist, werden die Scanlinien auch zum Finden von Strafstoßmarken und Ball genutzt, welche ebenfalls als Weiß segmentiert werden sollten. Zusätzlich sind die meisten Roboterteile ebenfalls weiß und können entsprechend klassifiziert werden, auch wenn die nachfolgenden Module davon derzeit keinen Nutzen haben.

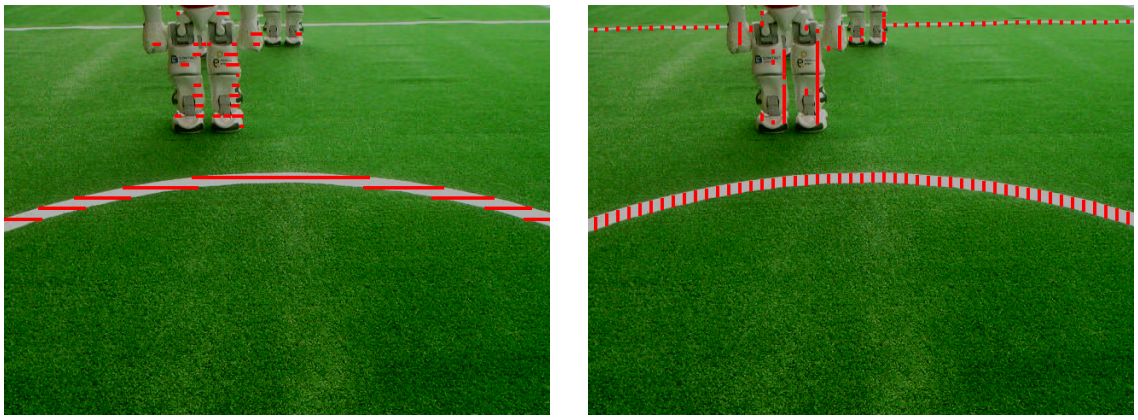
Weiße Regionen zeichnen sich dadurch aus, dass sie heller sind und eine geringere Farbsättigung haben als ihre Umgebung. Helligkeit und Sättigung können durch unterschiedliche Ausleuch-

tung und Schatten innerhalb eines Bildes variieren, was von den Algorithmen zur Weißklassifikation beachtet werden sollte. Da die Feld-Regionen zum Zeitpunkt der Weißklassifikation bereits klassifiziert sind, lassen sie sich bei der Weißklassifikation nutzen.

Weißklassifikation anhand von Feldregionen auf gleicher Scanlinie

In diesem Verfahren betrachte ich alle Scanlinien einzeln. Für jede noch nicht klassifizierte Region suche ich zu beiden Seiten die nächste `Feld`-Region auf der Scanlinie. Damit die betrachtete Region als `Weiß` klassifiziert wird, muss sie heller und weniger gesättigt sein, als beide `Feld`-Regionen. Zusätzlich wird zum einen überprüft, ob die Region heller ist, als die zu Beginn ermittelte Grundhelligkeit des Bildes, und zum anderen, ob ihre Sättigung kleiner gleich der maximal möglichen Sättigung einer weißen Region ist. Dazu nutze ich die in den `RelativeFieldColors` definierten Funktionen.

Falls zu einer Seite der betrachteten Region keine `Feld`-Region mehr liegt, klassifiziere ich sie nur anhand der einen `Feld`-Region auf der anderen Seite. Sollte eine Scanlinie einmal gar keine `Feld`-Regionen enthalten kann diese Methode auch keine weißen Regionen auf der Scanlinie finden. Für die nachfolgende Linienerkennung ist das allerdings irrelevant, da diese ohnehin nur weiße Regionen zwischen Feld-Region betrachtet. Um nicht für jede betrachtete Scanlinienregion neu nach benachbarten Feldregionen suchen zu müssen, habe ich diesen Ansatz als Zwei-Pass Verfahren umgesetzt. Jede Scanlinie wird zweimal durchlaufen, beim zweiten Mal aber in umgekehrter Richtung. Dabei wird jeweils die zuletzt angetroffene `Feld`-Region zwischengespeichert und alle nachfolgenden nicht klassifizierten Regionen werden mit ihr abgeglichen. Beim ersten Durchlaufen der Scanlinie, werden dabei Regionen welche die Kriterien zur Weißklassifikation gegenüber der `Feld`-Region bestehen vorläufig als `Weiß` markiert. Regionen die mit einer `Feld`-Region abgeglichen werden, aber den Test nicht bestehen werden sofort als `Anders` deklariert, also als weder `Feld` noch `Weiß`. Im zweiten Durchlauf in der entgegengesetzten Richtung, werden wiederum alle noch nicht endgültig klassifizierten Regionen anhand der zuletzt angetroffenen `Feld`-Region überprüft und endgültig als `Weiß` oder `Anders` klassifiziert. Im ersten Durchlauf vorläufig als `Weiß` klassifizierte Regionen, die im zweiten Durchlauf nicht noch einmal überprüft wurden, weil sie aus der anderen Richtung keine benachbarte `Feld`-Region haben, behalten ihre Klassifikation als `Weiß`. Die letztendlich als `Weiß` klassifizierten Regionen sind in Abbildung 3.8 zu sehen.



(a) horizontale weiße Regionen (b) vertikale weiße Regionen

Abbildung 3.8: Als weiß erkannte Scanlinienregionen (in rot eingezeichnet), komplementär zu den Feldregionen in Abbildung 3.7

Weißklassifikation anhand der Feldfarbe

Ein im Gegensatz dazu sehr simples Verfahren stützt sich komplett auf die zuvor ermittelte Feldfarbe, sowie Grundhelligkeit und -sättigung des Kamerabildes (siehe 3.2). Anhand dieser Daten lege ich in diesem Ansatz feste Schwellwerte für Mindesthelligkeit und Maximalsättigung weißer Regionen fest, die für das ganze Bild gelten. Ein klarer Vorteil dieses Verfahrens ist die sehr schnelle Ausführungszeit. Möglicherweise sind die ermittelten Schwellwerte jedoch unzureichend, um alle weißen Regionen in einem Bild zu identifizieren. Da das bisherige Verfahren allerdings ebenfalls mit festen Schwellwerten arbeitet und gute Resultate liefert, gehe ich davon aus, dass dieser Ansatz auch hier funktionieren kann.

Als Ausgangspunkt für einen Helligkeitsschwellwert bietet sich die Grundhelligkeit des Bildes an. Sofern das Bild nicht größtenteils weiß ist, was nur selten vorkommt, werden weiße Regionen immer eine deutlich höhere Helligkeit haben. Ich lege daher die Grundhelligkeit zuzüglich eines kleinen Sicherheitsabstands als Mindesthelligkeit fest, wobei ich diese noch auf die bei der Segmentierung des Spielfelds festgestellte maximale Feldhelligkeit begrenze.

Grundlage des Sättigungsschwellwerts ist die ermittelte minimale Feldsättigung. Da Feldlinien normalerweise eine deutlich niedrigere Sättigung als das Feld haben, reduziere ich diesen Wert. Abhängig von der ermittelten minimalen Feldsättigung, fällt die Reduktion allerdings unterschiedlich aus, da bei niedriger Feldsättigung die Sättigung der Linien deutlich näher liegen kann. Weil es bei manchen Beleuchtungsbedingungen vorkommen kann, dass die minimale Feldsättigung unter der maximalen Sättigung der Feldlinien liegt, verwende ich zusätzlich die Grundsättigung des Bildes. Diese wird wie auch die Grundhelligkeit einmalig zu Beginn anhand von drei horizontalen Scanlinien approximiert. Ausgehend davon, dass im Bild hauptsächlich das Feld zu sehen ist, entspricht die Grundsättigung ungefähr der durchschnittlichen Sättigung des Spielfeldes. Da die Sättigung weißer Bereiche deutlich niedriger sein muss und diese Option nur als Rückfall, für den Fall sehr niedrig gesättigter Feldregionen dient, verwende ich hier die halbierte Grundsättigung abzüglich eines kleinen Sicherheitsabstands. Von diesen beiden Sättigungsschwellwerten verwende ich den höheren, wobei ich den Schwellwert auf die über alle Bilder maximal erwartete Sättigung weißer Regionen begrenze.

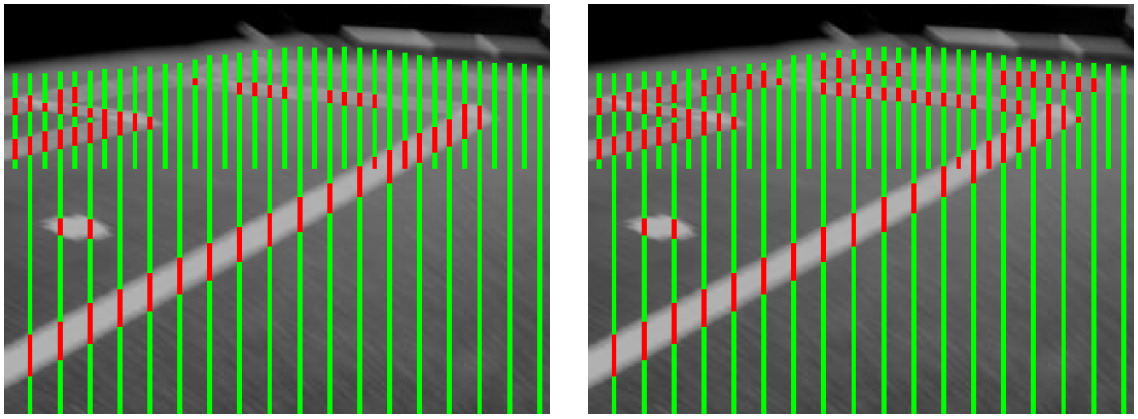
Ich deklariere dann alle Scanlinienregionen als `Weiß`, welche nicht schon zuvor als Feld erkannt wurden und die Schwellwerte für Helligkeit und Sättigung einhalten.

Weißsegmentierung bei der Regioneneinteilung

Da beide vorige Verfahren für sich genommen noch nicht optimal sind (siehe Abschnitt 4.2.2, Tabelle 4.10) habe ich ein dazu ergänzendes Verfahren entwickelt. Als hauptsächliches Problem hat sich herausgestellt, dass manchmal Regionen im vorigen Schritt, der Segmentierung des Spielfelds, bereits als `Feld` deklariert werden, obwohl sie eigentlich `Weiß` sein sollten. Das geschieht in weit entfernten Bildbereichen, in denen Feldlinien durch Bewegungsunschärfe mit den Feld vermischt werden und so einen Grünstich bekommen. Die zuvor implementierten Weißklassifikationen können jedoch keine falsch erkannten `Feld`-Regionen umdeklarieren, da sie beide direkt von der Felderkennung abhängig sind. Die erste Methode greift sogar direkt auf die `Feld`-Regionen als Fixpunkt für die Weißklassifikation zurück. Während das Umdeklarieren von `Feld`-Regionen im Schwellwertansatz prinzipiell möglich wäre, hängen dort die Schwellwerte aber auch vom ermittelten Feldfarbenbereich ab. Tests ergaben außerdem, dass dies zur fälschlichen Klassifizierung von durch Lichtreflexion sehr hellen Feldbereichen als weiße Regionen führen kann.

Die in der Folge entwickelte Weißsegmentierung bei der Regioneneinteilung ist daher eine

Ergänzung zu den anderen Verfahren. Das Ziel ist schon vor der Spielfeldklassifizierung erste Scanlinienregionen als weiß zu identifizieren, die ansonsten irrtümlicherweise dem Spielfeld zugeordnet würden. Da das nur in weit entfernten Bildteilen auftritt, begrenze ich dieses Vorgehen auf den relevanten Bereich, indem ich es nur auf den Bildteilen anwende in denen bei der Regionenbildung keine zusätzliche Glättung angewendet wird. Um hiermit tatsächlich nur Linienstücke als `Weiß` zu klassifizieren und beispielsweise großflächige Lichtreflexionen am Spielfeld auszuschließen, beschränke ich es außerdem auf Regionen mit einer für Linien typischen Länge von maximal 20 Pixeln.



(a) vertikale Regionen ohne Weißsegmentierung bei der Regioneneinteilung

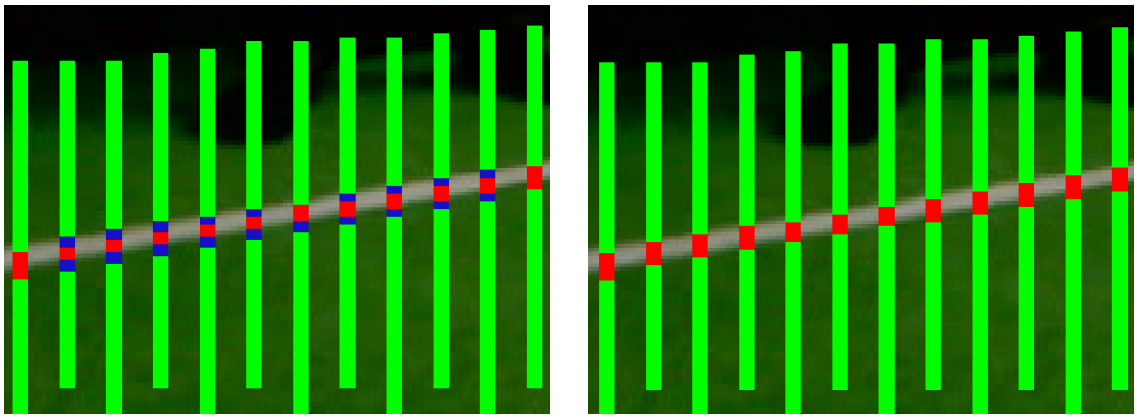
(b) vertikale Regionen mit Weißsegmentierung bei der Regioneneinteilung

Abbildung 3.9: Weißregionen (rot) auf einem unscharfen Bildausschnitt mit der Klassifikation per Schwellwert, mit und ohne Weißsegmentierung bei der Regioneneinteilung

Ist die Weißsegmentierung bei der Regioneneinteilung aktiviert, erhält diese einen zusätzlichen Teilschritt. Immer wenn eine Scanlinienregion beendet und in der zugehörigen Datenstruktur abgelegt wird, wird für die davor abgelegte Region diese Weißklassifikation durchgeführt, falls sie die oben beschriebenen Vorbedingungen erfüllt. Falls die betrachtete Region heller als die Grundhelligkeit und weniger gesättigt als die Grundsättigung des Bildes ist, wird geprüft, ob sie heller und weniger gesättigt ist als beide Nachbarregionen. Ist das beides erfüllt, wird die Region noch während der Regioneneinteilung als `Weiß` markiert und dann bei der Feldsegmentierung nicht mit einbezogen. Der Unterschied ist in Abbildung 3.9 deutlich zu erkennen.

3.3.4 Zusammenfassen benachbarter gleichartiger Regionen

Der letzte Schritt der neuen Scanliniensegmentierung ist das Zusammenfassen benachbarter gleichartiger Regionen. Wie bereits bei der Regionenbildung angemerkt, kann es vorkommen, dass am Übergang zwischen Feld und Linie mehrere Regionsgrenzen gefunden werden. Das führt dazu, dass der Übergangsbereich eine eigene Region bekommt, die sich schlecht weder dem Feld noch der Linie zuordnen lässt. Häufig sind sie stattdessen als `Anders` klassifiziert, was in Abbildung 3.10a in dunkelblau dargestellt ist. Da B-Human's Modul zur Linienerkennung nur solche `Weiß`-Regionen als Linienteilstücke akzeptiert, die direkt zwischen zwei `Feld`-Regionen liegen, wirkt sich das nachteilig auf die Erkennungsrate aus. Zusätzlich zum Zusammenführen gleichartiger Regionen entferne ich daher auch kleine als `Anders` klassifizierte Regionen, die zwischen `Feld` und `Weiß` liegen. Die an die entfernte Region angrenzenden Regionen werden entsprechend vergrößert, so dass sie jeweils eine Hälfte der entfernten Region einnehmen. Das Ergebnis ist in Abbildung 3.10b zu sehen.



(a) Kleine falsch zugeordnete Regionen am Übergang zwischen Feld und Linie

(b) Übergang zwischen Feld und Linie mit gefüllten Lücken

Abbildung 3.10: Füllung der Übergänge zwischen Feld und Linie

3.4 Anpassungen am Linienerkennungsmodul

Der bisherige Algorithmus zur Linienerkennung wie er im `LinePerceptor` Modul implementiert ist, nutzt als Grundlage seiner Berechnungen hauptsächlich die `ColorScanLineRegions`. Zu drei Zwecken greift er aber noch direkt auf das farbsegmentierte Kamerabild zu:

1. Beim Hinzufügen eines neuen Segments zu einem Linienkandidaten wird stichprobenweise überprüft, ob es durchgehend weiß ist.

2. Beim Erweitern von Linien über den durch die `ColorScanLineRegions` gefundenen Teil hinaus, wird als zusätzliche Validierung die Breite der Linie im Bild gemessen.

3. Die Position von Mittelkreiskandidaten wird anhand genauer Bestimmung von innerer und äußerer Kante der Mittelkreislinie präzisiert.

Für alle drei Fälle wurden kalibrierungsfreie Alternativlösungen benötigt. Die dafür entwickelten Lösungen ersetzen die Zugriffe auf das farbsegmentierten Bild durch die Nutzung der `RelativeFieldColors`.

3.4.1 Perspektivisch korrekte Vergleichspunktauswahl

Zur Validierung neuer Liniensegmente werden in regelmäßigen Abständen Pixel auf der vermuteten Linie ausgewählt. Ich überprüfe für jedes Testpixel, ob es weiß ist, anhand von Vergleichspunkten, die zu beiden Seiten des Liniensegments ausgewählt werden. Dafür wird zunächst einen Normalenvektor zum Liniensegment in Feldkoordinaten berechnet. Die Bildkoordinaten des Testpixels werden ebenfalls auf die Feldebene projiziert. Im Nahbereich wähle ich dann die Punkte im Abstand der doppelten Linienbreite in Richtung des positiven und negativen Normalenvektors als Vergleichspunkte aus. Bei weiter entfernten Testpixeln gehe ich bis zur dreifachen Linienbreite, da es sonst zu Problemen auf verwischten Bildern kommt. Diese Vergleichspunkte werden dann in die Bildkoordinaten zurück projiziert (siehe Abbildung 3.11). Mit der `isWhiteNearField` Funktion aus den `RelativeFieldColors` teste ich dann, ob das Testpixel tatsächlich heller und weniger gesättigt ist als die Bereiche neben der vermuteten Linie. Schlägt dieser Abgleich für mindestens ein Viertel der Testpixel fehl, wird das Liniensegment abgelehnt.

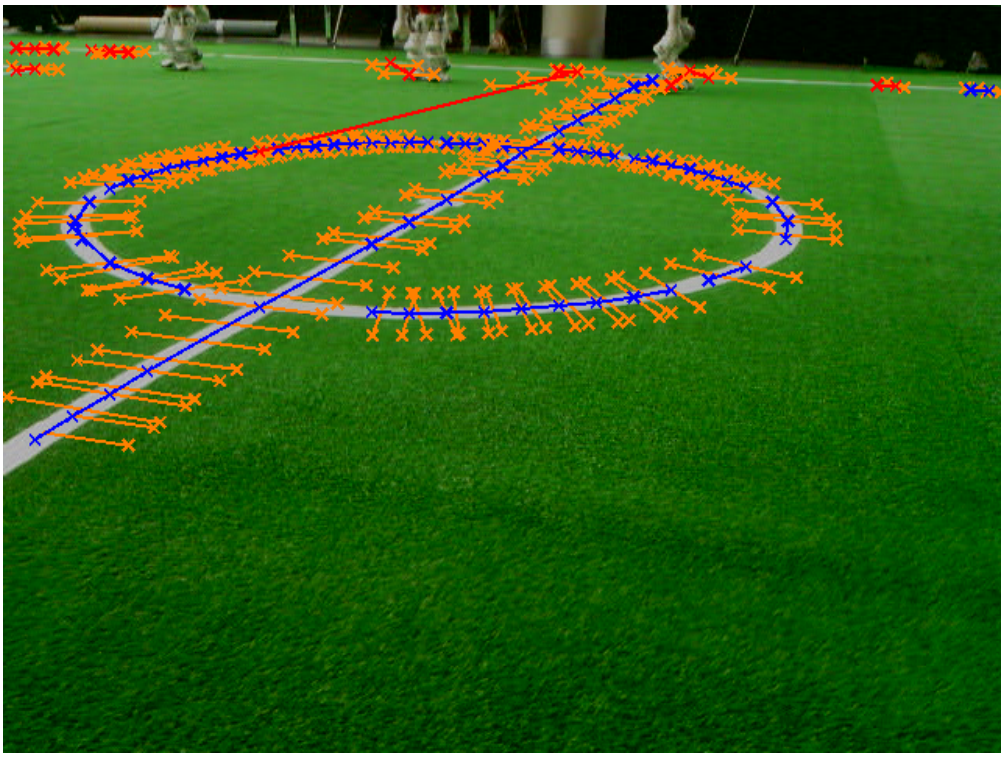
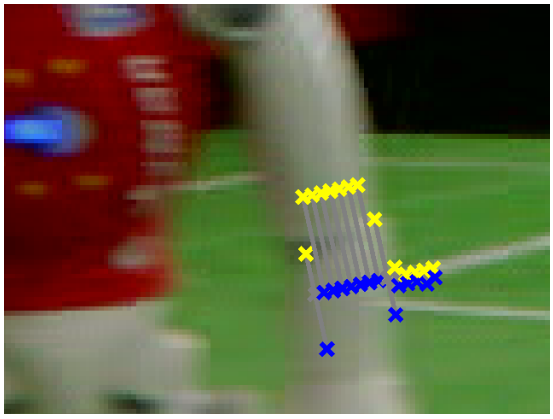


Abbildung 3.11: Perspektivisch korrekte Vergleichspunkte (orange Kreuze). Angenommene Liniensegmente in blau, abgelehnte in rot.

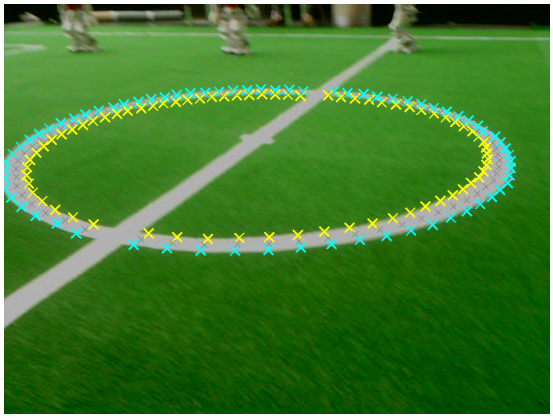
Zur Bestimmung der Linienbreite habe ich einen anderen Ansatz gewählt. Denn Referenzpunkte außerhalb des Bereichs, den eine echte Linie einnehmen kann, zu wählen und dann vom gegebenen Startpunkt aus solange in Richtung der Normale weiter zu suchen, bis `isWhiteNearField` den betrachteten Punkt nicht mehr als weiß ansieht, scheitert nämlich bei unscharfen Bildern. Dort ist der Übergang zwischen Linie und Feld verwischt und kann von den Funktionen in den `RelativeFieldColors` sowohl Feld als auch Linie zugeordnet werden. Bei der Verwendung von `isWhiteNearField` wird der Übergangsbereich, da er etwas heller ist als das normale Feld, komplett als weiß eingestuft. Das führt bei unscharfen Bildern zu sehr hohen festgestellten Linienbreiten und auch echte Linien werden dann zur Vermeidung von Fehlerkennungen aussortiert. Stattdessen verwende ich die Funktion `isFieldNearWhite` mit dem Ausgangspunkt auf der Linie als Referenzwert. Von dort aus gehe ich in pixelgroßen Schritten in Richtung der positiven und negativen Normale und überprüfe die entsprechenden Pixel mit `isFieldNearWhite`, ob sie als Feld eingestuft werden. Das stuft in unscharfen Bilden den vermischten Übergangsbereich als Feld ein und ist daher besser geeignet, um die Linienbreite zu bestimmen. Die detektierten Kanten der Linie sind die letzten Pixel, die noch nicht als Feld eingestuft werden. Für den Fall, dass einmal versucht wird die Linienbreite an einer Stelle zu bestimmen, wo sich gar keine Linie befindet, wird die Suche nach den Kantenpixeln abgebrochen sobald eine Breite erreicht ist, die unmöglich zu einer Linie gehören kann (siehe Abbildung 3.12a). Nach einer Transformation in Feldkoordinaten kann dann die Breite der Linie als der Abstand der beiden Kanten bestimmt werden.

Die Bestimmung der inneren und äußeren Kante der Mittelkreislinie (Abbildung 3.12b) funktioniert prinzipiell genau so, bereitet aber folgende zusätzliche Schwierigkeit: Initial kann die Position des Mittelpunkts für Mittelkreiskandidaten nur grob geschätzt werden. Zur genaueren Positionsbestimmung wird ausgehend von der Schätzung der Feldkoordinaten des

Mittelpunkts der erwartete Verlauf des Kreisbogens in das Bild projiziert. Dieser kann jedoch auch neben dem tatsächlichem Verlauf liegen. In diesem Fall befindet sich der Startpunkt nicht auf der Linie. Daher überprüfe ich zunächst, mit der gleichen Methode wie beim Validieren neuer Liniensegmente, ob der Ausgangspunkt tatsächlich weiß ist. Falls nicht, suche ich eine Linienbreite weit nach innen und außen nach einem Pixel, das als weiß eingestuft wird. Wird ein solcher Punkt gefunden, läuft die Bestimmung der inneren und äußeren Kante genauso wie zuvor ab. Ansonsten wird die betreffende Stelle des Kreisbogens verworfen und geht nicht in die Positionskorrektur ein. Müssen zu viele Stellen verworfen werden, wird der Mittelkreiskandidat als falsch angesehen und komplett verworfen.



(a) Festgestellte Linienbreite (Bildausschnitt). Im Roboterarm wird die Suche wegen unmöglich großer Linienbreite abgebrochen



(b) Kantenpunkte am Mittelkreis

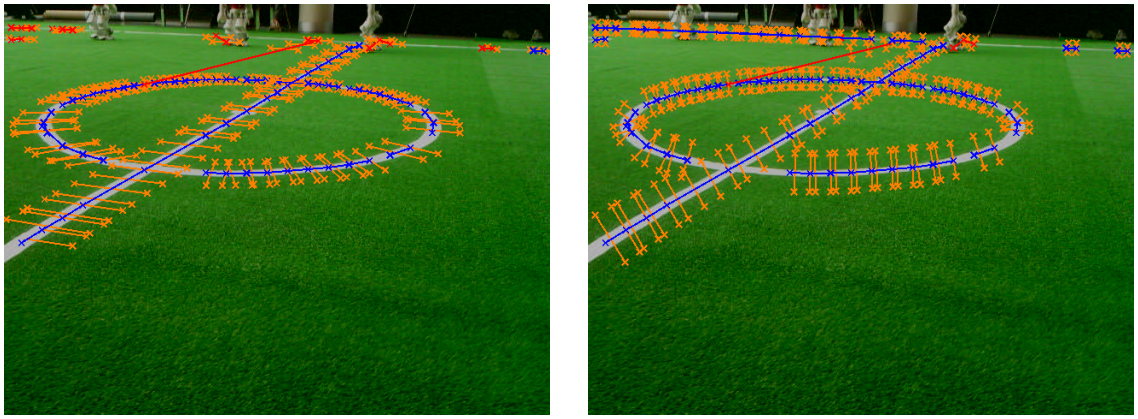
Abbildung 3.12: Pixelgenaue bestimmte Anfangs- und Endpunkte von Linien

3.4.2 Im Bildkoordinatensystem berechnete Vergleichspunkte

Zur Verbesserung der Laufzeit habe ich eine weitere Methode zur Auswahl der Vergleichspunkte bei der Validierung neuer Liniensegmente implementiert. Sie vermeidet es von Bildkoordinaten in die Feldebene und zurück transformieren zu müssen, sondern benötigt nur eine Transformation in die Feldebene, um die Entfernung des betrachteten Punktes zum Roboter abschätzen zu können. Die restlichen Berechnungen passieren in der Bildebene. Dadurch ist diese Methode nicht perspektivisch korrekt, was für eine potenziell schlechtere Vergleichspunktauswahl sorgt. Die unterschiedlichen Positionen der Vergleichspunkte werden in Abbildung 3.13 deutlich. Die Vergleichspunkte nur in Bildkoordinaten zu berechnen bringt außerdem nur für die Validierung neuer Liniensegmente einen nennenswerten Geschwindigkeitsvorteil, da das Ermitteln der Linienbreite und die Positionskorrektur von Mittelkreiskandidaten beide dennoch Berechnungen in der Feldebene mit anschließender Rücktransformation benötigen.

Da ich soweit möglich nur in Bildkoordinaten rechnen will, berechne ich zunächst einen Normalenvektor senkrecht zum untersuchten Linienstück im Bild. Um damit zu den Vergleichspunkten zu kommen, wird dann noch ein von der Entfernung abhängiger Skalierungsfaktor benötigt. Die Entfernung des untersuchten Punktes zum Roboter wird in Feldkoordinaten benötigt. Ihn vom Bild in die Feldebene zu projizieren, ist also unumgänglich. Für die Berechnung der exakten Länge eines Feldkoordinatenvektors muss eine Quadratwurzel berechnet werden, was eine aufwändige Operation ist. Da es nicht auf die pixelgenaue Position der Vergleichspunkte ankommt, ist eine Schätzung hier ausreichend. Entsprechend verwende ich stattdessen die vom Roboter aus nach vorne zeigende x-Komponente des Feldkoordinatenvek-

tors addiert auf einen variierenden Anteil der y-Komponente, der umso größer wird, je kleiner der x-Wert ist. Anhand dieser Abstandsschätzung berechne ich dann eine Pixeldistanz, die ein Vergleichspunkt zum Testpunkt haben sollte, um mit Sicherheit nicht auf der vermuteten Linie, aber möglichst nah daran zu liegen. Nach der Skalierung des Normalenvektors auf die berechnete Länge erhält man durch Addition und Subtraktion des Normalenvektors auf/vom Testpunkt die Bildkoordinaten von zwei Vergleichspunkten. Diese nutze ich dann wie gehabt als Eingaben in die `isWhiteNearField` Funktion der `RelativeFieldColors`, um den Testpunkt eines Liniensegmentes zu validieren.



(a) Perspektivisch korrekte Vergleichspunkte

(b) In Bildkoordinaten berechnete Vergleichspunkte

Abbildung 3.13: Positionen der Vergleichspunkte

3.4.3 Weitere Änderungen am Linienerkennungsalgorithmus

Im Zuge der Arbeiten am `LinePerceptor` sind zwei weitere Verbesserungsmöglichkeiten aufgefallen, die nicht mit der Farbkalibrierung zusammenhängen. Sie verbessern Probleme im alten Linienerkennungsalgorithmus, welche die korrekte Erkennung von Linien im Nahbereich teilweise verhinderten.

Erweiterte `LineSpot`-Auswahl

Das erste Problem trat bei der Auswahl der `LineSpots` auf, denjenigen Stellen auf den Scanlinien an denen Feldlinien vermutet werden. Damit möglichst keine Scanlinienregion, die zwar weiß ist aber etwas anderes als eine Feldlinie zeigt, als `LineSpot` ausgewählt wird, fordert der Algorithmus, dass zu beiden Seiten angrenzend Feld-Regionen einer gewissen Mindestgröße liegen müssen. Die Mindestgröße der Feld-Regionen beträgt das Vierfache der Linienbreite auf dem Feld, also 20cm. Die Größe in Bildpixeln wird dann berechnet durch Projektion des potenziellen `LineSpots` in die Feldebene, Addition eines 20cm Vektors in Richtung der Scanlinie und Rückprojektion dieses Punktes in das Kamerabild. Der Abstand dieses Punktes zum potenziellen `LineSpot` ergibt dann die Mindestgröße der benachbarten Feld-Regionen in Pixeln.

Da der alte `LinePerceptor` diese Mindestgröße auch für Regionen nah am Bildrand eingeforderte, wurden einige sinnvolle `LineSpots` wieder aussortiert. Bei Bildern der unteren Kamera passierte dies häufig, da diese nur eine kleine Fläche abdeckt und Linien sehr mittig im Bild platziert sein müssen, damit ausreichend Feld darum herum zu sehen ist. Das habe ich

geändert und akzeptiere einen `LineSpot` auch dann, wenn eine ihn umgebende Feld-Region kürzer ist als die normalerweise geforderte Mindestgröße, aber bis zum Bildrand reicht.

Möglicherweise problematisch ist allerdings, dass dadurch auch in das Bild hineinragende Roboterteile leichter fälschlich als Linie erkannt werden, wie in Abbildung 3.14 beispielhaft zu sehen ist. Das ist so, weil nur teilweise zu sehende Roboter von der Hinderniserkennung meistens nicht erkannt werden und daher auch nicht von der Linienerkennung ausgeschlossen werden. Bislang wurde im B-Human Team das Risiko von solchen Fehlerkennungen höher eingeschätzt, als der Nutzen der zusätzlichen Linienfunde. Allerdings wurde meines Wissens auch nie eine umfassende Analyse der Häufigkeit solcher Fehlerkennungen durchgeführt. Meine Evaluation dieser Änderung findet sich unter 4.2 Genauigkeit der Linienerkennung.

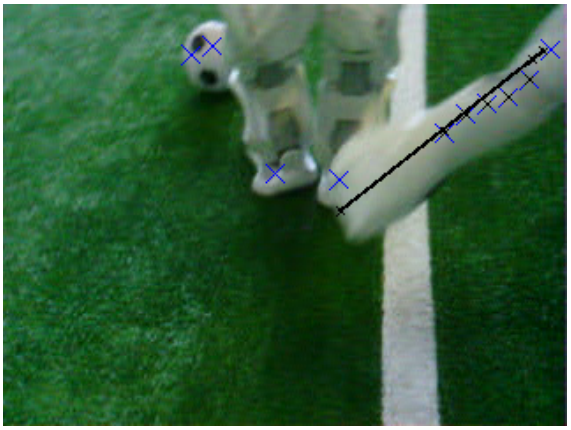
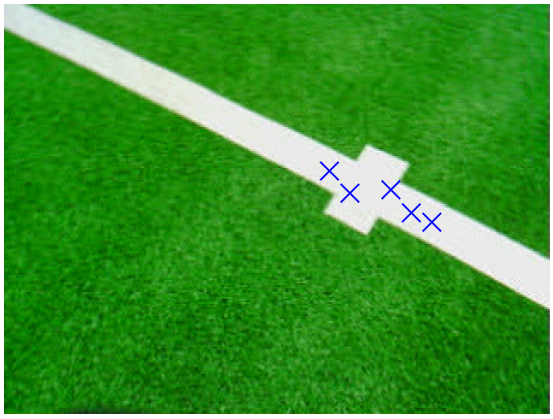
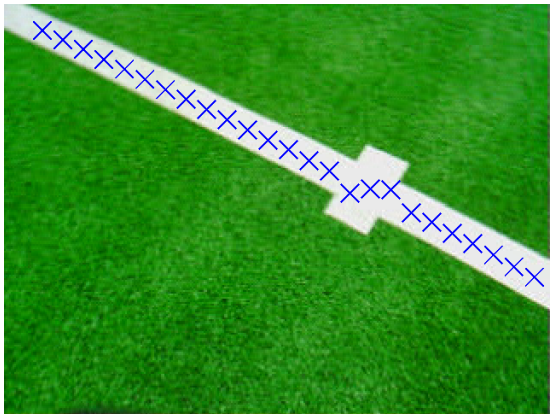


Abbildung 3.14: Fälschlich in einem Roboterarm erkannte Linie



(a) `LineSpot`s mit dem alten `LinePerceptor`



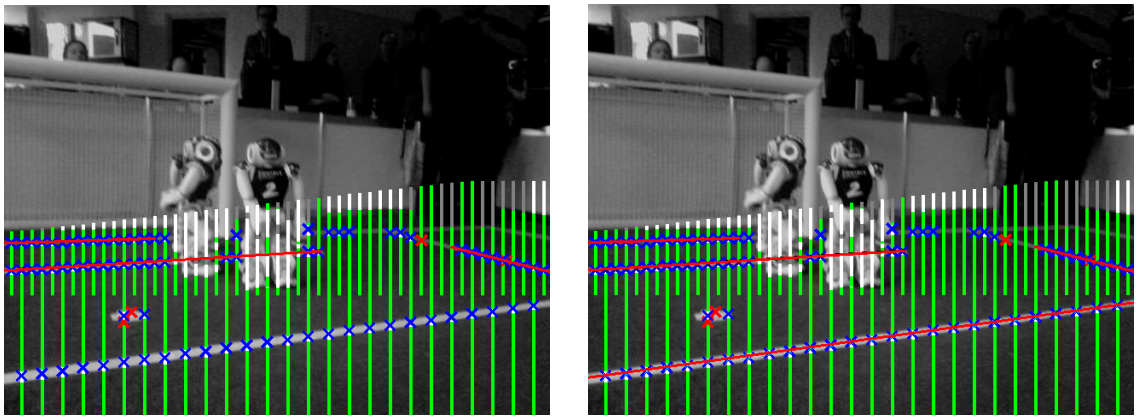
(b) `LineSpot`s mit der verbesserten `LineSpot`-Auswahl

Abbildung 3.15: Vergleich der `LineSpot`s in der unteren Kamera

Korrektur des Verbindens von `LineSpot`s auf vertikalen Scanlinien

Das zweite Problem betraf den Nahbereich der oberen Kamera. Dort wurde beim Verbinden der `LineSpot`s nicht korrekt beachtet, dass die vertikalen Scanlinien unterschiedlich lang sein können. Beim Verbinden der `LineSpot`s zu Liniensegmenten wird immer zunächst versucht neue `LineSpot`s einem bestehenden Liniensegment hinzuzufügen, sofern es bereits ein passend verlaufendes Segment gibt. Andernfalls probiert der `LinePerceptor` mit einem anderen `LineSpot` auf der vorherigen Scanlinie ein neues Liniensegment zu beginnen. Wenn die vertikalen Scanlinien unterschiedlich lang sind, führte das dazu, dass im unteren Bereich ungefähr horizontal verlaufende Linien nur gefunden werden konnten, wenn bereits ein angefangenes Liniensegment existierte. Sonst wurde zu den `LineSpot`s im nahen Bildbereich nämlich immer nur auf den kurzen Scanlinien, die nur den Fernbereich abdecken, nach `LineSpot`s zum Verbinden zu neuen Liniensegmenten gesucht. Dieser Fall ist in Abildung 3.16a zu sehen. Ich habe diesen Fehler behoben, indem ich die Höhe im Bild, auf welcher der aktuelle `LineSpot`

gefunden wurde, mit berücksichtige. Ich suche dann auf der letzten Scanlinie, die mindestens bis zur Höhe des zu verbindenden `LineSpot`s herunter reicht.



(a) Der alte `LinePerceptor` findet zwar die `LineSpot`s (blaue Kreuze), verbindet sie aber nicht. Die Linien im Hintergrund (rot) werden erkannt.

(b) Mit der Änderung wird auch die vordere Linie (rot) korrekt detektiert, obwohl die kurzen Scanlinien dort nicht hinreichen.

Abbildung 3.16: Vergleich der gefundenen Linien mit und ohne Korrektur der Verbindung von `LineSpot`s auf vertikalen Scanlinien

4. Evaluation

In diesem Kapitel werden für die entwickelten Vorgehensweisen zur kalibrierungsfreien Linienerkennung optimale Parameter in Hinblick auf die Erkennungsgenauigkeit bestimmt (Abschnitt 4.2.1) und die verschiedenen Implementierungen mit einander verglichen (Abschnitt 4.2.2). Die damit als optimal ermittelten Implementierungen werden dann als Gesamtsystem mit dem alten Linienerkennungssystem verglichen und ihre Einsatztauglichkeit auf RoboCup-Wettbewerben evaluiert. Betrachtet wird einerseits die Ausführungszeit des entwickelten Codes auf den Robotern (Abschnitt 4.3) und andererseits die Genauigkeit der Linienerkennung (Abschnitt 4.2.3) insbesondere im Hinblick auf die Falschpositivrate.

4.1 Methode

4.1.1 Genauigkeit der Linienerkennung

Die Tests der Genauigkeit der entwickelten Module zur kalibrierungsfreien Linienerkennung führe ich mit dem eigens dafür entwickelten Testtool durch. Da es nur die Unterschiede zur bisherigen Linienerkennung feststellt, überprüfe ich die detektierten Unterschiede von Hand. Dabei gibt es die Kategorien neu gefundener Linien und nicht mehr gefundener Linien, wobei jeweils zwischen korrekten Linienfunden und Falschdetektionen unterschieden wird. Daraus ergeben sich über die Anzahl der gefundenen Linien und Falschdetektionen Differenzbeträge, welche es erlauben die Zuverlässigkeit der neu implementierten Linienerkennung abzuschätzen. Geht man zusätzlich davon aus, dass die von beiden Verfahren gefundenen Linien immer korrekt sind, ergeben sich auch absolute Beträge zur Anzahl der detektierten Linien. Nicht erkannte Linien lassen sich mit dem Testtool hingegen nicht feststellen und gehen nicht in die Evaluation ein.

Als Datengrundlage der Evaluation dienen Spielaufzeichnungen der Roboter. Dadurch wird mit realen Spielsituationen getestet. Da sich unter den vorhandenen Spielaufzeichnungen keine von Spielen bei geringer Beleuchtung befinden, habe ich dafür eine Aufzeichnung hinzu genommen, die außerhalb eines Testspiels auf einem halben Feld aufgenommen wurde.

Nachfolgend eine Aufstellung aller verwendeten Logs. Die Größe der Logs wurde mit dem Testtool deutlich verringert auf ein handhabbares Maß. Die angegebene Anzahl der Testbilder entspricht nicht der Gesamtzahl der Bilder in den gefilterten Logs, sondern umfasst nur diejenigen Bilder, auf denen die Linienerkennung ausgeführt wird. Alle Bilder, bei denen wegen fehlendem Bodenkontakt der Füße des Roboters oder anderen Gründen die Linienerkennung nicht ausgeführt wird, wurden nicht mitgezählt. Die aufgeführten Anzahlen der gefundenen Linien und Mittelkreise wurden mit dem alten Linienerkennungssystem bei erneuter Ausführung im Testprogramm ermittelt.

- B-Human Testspiellogs von Feldspielern und Torwart, aufgenommen in einem Raum mit großer Glasfront an einer Längs- und einer Querseite des Feldes. Zu einem kleinen Anteil gibt es direktes Sonnenlicht.

Kamera	Logs	Testbilder	Linien	Mittelkreise
Upper	3	605	1515	68
Lower	3	608	119	9
Gesamt	3	1213	1634	77

Tabelle 4.1: Ergebnisse des alten Linienerkennungssystems auf hellen Testspiellogs

- Feldspieler Logs aus einem Testspiel gegen HTWK Leipzig.

Kamera	Logs	Testbilder	Linien	Mittelkreise
Upper	2	487	994	75
Lower	2	485	125	20
Gesamt	2	972	1119	95

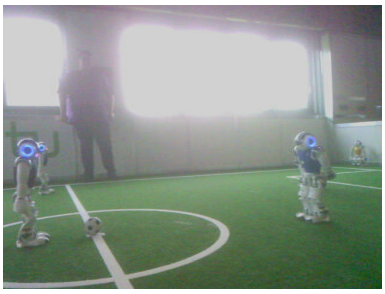


Tabelle 4.2: Ergebnisse des alten Linienerkennungssystems auf Testspiellogs auf einem gleichmäßig beleuchteten Feld

- Sehr dunkles zu Testzwecken auf einem halben Feld aufgenommenes Log.

Kamera	Logs	Testbilder	Linien	Mittelkreise
Upper	1	324	364	9
Lower	1	322	12	0
Gesamt	1	646	376	9

Tabelle 4.3: Ergebnisse des alten Linienerkennungssystems auf einem sehr dunklen Log

- Testspiel-Log mit versehentlich falsch kalibrierter Kamera. Alle Bilder der oberen Kamera in diesem Log sind unscharf.

Kamera	Logs	Testbilder	Linien	Mittelkreise
Upper	1	382	563	0

Tabelle 4.4: Ergebnisse des alten Linienerkennungssystems auf einem sehr unscharfen Log

Die Logs aus den ersten beiden Quellen sind so ausgewählt, dass zusammen alle Spielerrollen (Torwart, Verteidiger, Angreifer) in der Zusammensetzung, wie sie in echten Spielen vorkommen, abgedeckt sind. Ich habe dafür Logs aus zwei verschiedenen Testspielen ausgewählt, um verschiedene Beleuchtungssituationen und Feldumgebungen abzudecken.

Ich betrachte die Ergebnisse auf den Logs sowohl getrennt von den anderen, als auch gemeinsam mit allen zusammen in Summe. Dadurch entsteht ein aussagekräftiges Gesamtergebnis auf einer breiten Datenbasis, während auch die Möglichkeit genutzt wird, die entwickelten Linienerkennungsverfahren auf Schwächen in bestimmten Situationen zu testen.

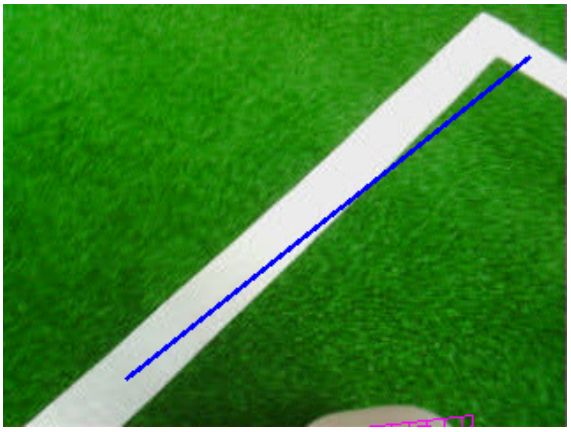


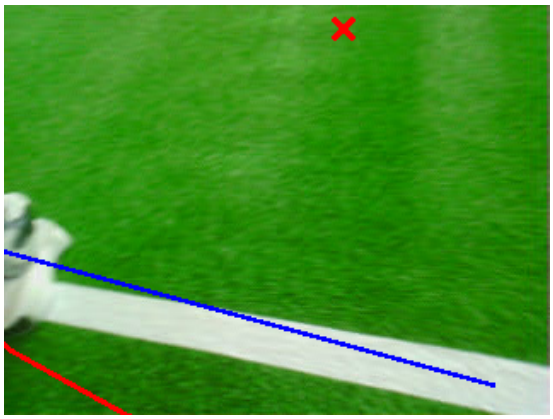
Abbildung 4.1: falsch positiver Linienfund

Immer wenn das Testprogramm zwischen neuem und altem Linienerkenner übereinstimmende Linien und Mittelkreise findet, werden diese automatisch als korrekt angesehen. Linienfunde die vom Testprogramm nicht als äquivalent zu einer auch vom alten Linienerkenner gefundenen Linie angesehen werden, sowie Linien, die nicht mehr gefunden werden, müssen von Hand überprüft werden. Um diesen Prozess so objektiv und wiederholbar wie möglich zu machen, habe ich mich dabei auf die folgenden Regeln festgelegt:

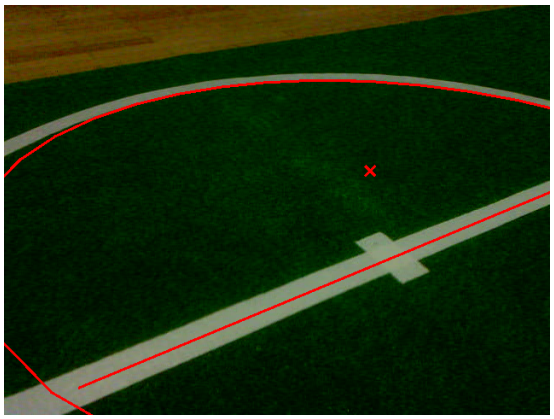
- Werden mehrere kürzere Teilstücke einer einzelnen Feldlinie separat gefunden, werden alle Teilstücke zusammen als ein korrekter Linienfund gezählt. Der obere und untere Halbkreis des Mittelkreises zählen dabei jeweils als eine Linie.
- Linienfunde, die über Eck verlaufen und dabei durch das Spielfeld hindurch gehen, werden als falsch positiv gezählt (siehe Abbildung 4.1). Ebenso werden Linienfunde, die über das Ende der echten Linie hinaus in das Feld hineinragen als falsch positiv gezählt.
- Gerade Linien, die im Mittelkreis gefunden wurden, dürfen über das Feld verlaufen, sofern sie im Mittelkreis beginnen und enden. Außerdem müssen alle `LineSpot`s, aus denen die Linie konstruiert wurde, auf dem Mittelkreis liegen. Ansonsten können solche Linie nicht zum Finden des Mittelkreises beitragen und werden als falsch positiv gezählt.
- In Unterstützungsbalken des Tores gefundene Linien werden als korrekte Linienfunde

behandelt, da diese im Linienmodell vorhanden sind. Die Torpfosten und die Latte zählen allerdings nicht dazu, weil nur Objekte auf Bodenhöhe berücksichtigt werden können.

- Gefundene Mittelkreise, deren Mittelpunkt vom tatsächlichen Mittelpunkt im Bild abweicht (Abbildung 4.2b), die aber eindeutig zum echten Mittelkreis gehören, sind meistens auf eine schlechte Transformationsmatrix zwischen Bild und Feldebene zurückzuführen. Sie werden daher als korrekte Mittelkreise gezählt.



(a) Falsch positiver Mittelkreis (rot) und falsch positiver Linienfund (blau)



(b) Schlecht positionierter aber korrekter Mittelkreis

Abbildung 4.2

Diese Ergebnisse sind hier in Form von Ergebnistabellen dargestellt, deren Aufbau in Tabelle 4.5 zu sehen ist. Darin sind jeweils die Teilergebnisse auf den unterschiedlichen Logs, getrennt nach verwendeter Kamera, aufgelistet, sowie die aufsummierten Ergebnisse je Kamera und das Gesamtergebnis. Die Aufstellung ist dabei nach geraden Feldlinien und Mittelkreisen unterteilt, da diese im Linienerkenner separat behandelt werden und ich sie daher auch getrennt auswerte. Für Linien und Mittelkreise habe ich jeweils die absolute Anzahl der korrekten Funde (tp_n, true positives), das Verhältnis der neuen true positives zur Anzahl der bisherigen Linienfunde ($\mathrm{tp}_n/\mathrm{p}_a$), die Anzahl der falsch positiven Funde (fp_n, neue false positives) und die Anzahl der nicht mehr gemachten Fehler (tn_n, neue true negatives) angegeben.

Genaue Angaben zu den üblichen Beurteilungsmaßen Precision und Recall (im Deutschen auch als Relevanz und Sensitivität bezeichnet) sind leider nicht möglich. Zur Berechnung der Precision ($\frac{tp}{p} = \frac{tp}{tp+fp}$) ist die Annahme notwendig, dass es keine falschen Funde gibt, die gleichzeitig vom alten und vom neuen Modul gemacht werden. Denn in diesem Fall, dass beide Module den gleichen Fehler machen, kann das Testprogramm keinen Unterschied feststellen und wird sie automatisch als korrekte Funde einstufen. Der Recall ($\frac{tp}{tp+fn}$) lässt sich gar nicht berechnen, da die Gesamtanzahl der Linien in den Testbildern unbekannt ist. [Müller u. Guido, 2017, S. 267]

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	$\mathrm{tp}_n/\mathrm{p}_a$	fp_n	tn_n	tp_n	$\mathrm{tp}_n/\mathrm{p}_a$	fp_n	tn_n
Upper	hell								
	⋮		⋮				⋮		
Gesamt	**Gesamt**	Σ_u				Σ_u			
Lower	hell								
	⋮		⋮				⋮		
Gesamt	**Gesamt**	Σ_l				Σ_l			
Gesamt	**Gesamt**	Σ_g				Σ_g			

Tabelle 4.5: Aufbau der Ergebnistabellen

4.1.2 Laufzeit

Die Laufzeit der neu entwickelten Module teste ich durch Ausführung auf einem physischen Roboter. Ich verwende dazu die im B-Human Framework enthaltenen Stopwatch Funktionalitäten, die für alle Module automatisch die Ausführungszeit aufzeichnen. Alle Laufzeittests führe ich im B-Human RoboCup Labor auf einem halben Kleinfeld durch.

Für alle Modulkonfigurationen teste ich drei verschiedene Verhaltensweisen des Roboter einzeln:

1. Stehender Roboter, der keine Bewegungen ausführt. Der Roboter steht in der eigenen Hälfte am Mittelkreis und schaut Richtung gegnerischem Tor. Die untere Hälfte des Mittelkreises ist in der unteren Kamera zu erkennen, die Bilder der oberen Kamera werden nahezu komplett vom Feld eingenommen, siehe Abbildung 4.3. Das ist der erwartete Laufzeit Worst Case, da die kompletten Bilder analysiert werden müssen und viele Linien zu sehen sind.
2. Stehender Roboter, der den Kopf nach links und rechts hin und her dreht. Die dabei zu sehende Umgebung ist in Abbildung 4.4 dargestellt, wobei sie einen geringen Einfluss auf die Laufzeit haben sollte, da der Bereich außerhalb der festgestellten Feldbegrenzung bei der Linienerkennung ignoriert wird.
3. Laufender Roboter, der versucht den Ball im Blick zu behalten und in das Tor zu befördern.

Für alle drei Fälle ermittele ich die maximale und durchschnittliche Laufzeit über 500 Bilder je Kamera.

Abbildung 4.3: Sicht des Roboters im ersten Test

(a) Umgebung zur linken Seite (b) Umgebung zur rechten Seite

Abbildung 4.4: Umgebung bei der Durchführung der Laufzeittests

4.2 Genauigkeit der Linienerkennung

4.2.1 Bestimmung optimaler Parameter der Scanliniensegmentierung

Die neu entwickelte Scanliniensegmentierung (Abschnitt 3.3) benötigt einige fest eingestellte Parameter. Welche Parameter das im einzelnen sind, ist bei den verschieden implementierten Ansätzen teilweise unterschiedlich.

Die meisten einstellbaren Parameter gibt es bei der Regioneneinteilung der Scanlinien. Für die beiden Ansätze, die Kantenfilter verwenden, muss ein Kantenschwellwert festgelegt werden. Zudem muss evaluiert werden, ob die Sättigung in das Kantenfilter einbezogen werden soll, was Einfluss auf den optimalen Schwellwert hat. Beim Ansatz der inkrementellen Regionenbildung werden stattdessen Ähnlichkeitsschwellwerte für Helligkeit, Farbton und Sättigung benötigt. Solche Ähnlichkeitsschwellwerte werden auch bei der Segmentierung des Spielfelds benötigt. Diese sind von denen bei der inkrementellen Regioneneinteilung unabhängig. Für die Segmentierung weißer Bildregionen müssen keine Parameter eingestellt werden, da diese die Funktionen der `RelativeFieldColors` oder die zuvor ermittelten Feldfarben und die Grundhelligkeit und -sättigung des Kamerabildes verwendet.

Ich habe die für die weitere Evaluation verwendeten Parameter experimentell bestimmt. Dafür habe ich eine Teilmenge der zur abschließenden Evaluation verwendeten Logs genutzt. Überschneidungen mit dem Testdatensatz wurden verhindert, indem die Originallogs mit der gleichen Filterung, wie für die Testdaten, aber um ein paar Bilder versetzt gefiltert wurden. Um für die Parameteroptimierung die Vergleichsbilder des Testprogramms nicht von Hand auswerten zu müssen, habe ich auf eine möglichst geringe Anzahl nicht mehr wiedergefunden Linien hin optimiert. Die Anzahl nicht mehr gefundener Linien, die vom alten System gefunden wurden, wird dafür vom Testprogramm in einer Ergebnisdatei ausgegebene. Dabei habe ich in Testläufen die neue Scanliniensegmentierung zusammen mit dem neuen `LinePerceptor` in der in Abschnitt 4.2.2 Ergebnisse des kalibrierungsfreien `LinePerceptors` mit alten `ColorScanLineRegions` als am Besten erachteten Konfiguration (perspektivisch korrekte Vergleichspunkte und erweiterte `LineSpot`-Auswahl) benutzt und dessen Ergebnisse mit denen des alten Linienerkennungssystems verglichen. Die unter diesen Bedingungen optimalen Parameter habe ich auch für die gesamte restliche Evaluation genutzt.

4.2.2 Genauigkeit der einzelnen Module

Ergebnisse des alten Linienerkenners mit kalibrierungsfreier Scanliniensegmentierung

In diesem Abschnitt habe ich die Ergebnisse des alten `LinePerceptor` Moduls unter Verwendung der kalibrierungsfrei erstellten `ColorScanLineRegions` als Eingabedaten getestet. Das ermöglicht die Auswirkungen der geänderten Scanliniensegmentierung getrennt von den Änderungen am Linienerkenner zu bewerten.

Zur Segmentierung der Scanlinien habe ich einige verschiedene Ansätze und Ausführungsoptionen implementiert. Von diesen habe ich zunächst die drei Ansätze zur Regioneneinteilung (3.3.1) getestet, jeweils mit beiden Ansätzen zur Erkennung weißer Regionen. Da die Auswirkungen der unterschiedlichen Methoden zur Weißerkennung unabhängig von der verwendeten

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1332	89%	7	4	59	87%	0	0
	gleichmäßig	950	96%	2	1	85	113%	0	0
	dunkel	332	91%	0	0	9	100%	0	0
	unscharf	472	84%	4	1	0	0/0	0	0
Upper	**Gesamt**	3086	90%	13	6	153	101%	0	0
Lower	hell	122	103%	0	1	11	122%	0	0
	gleichmäßig	120	96%	0	0	18	90%	0	0
	dunkel	14	117%	0	0	0	0/0	0	0
Lower	**Gesamt**	256	100%	0	1	29	100%	0	0
Gesamt	**Gesamt**	3342	91%	13	7	182	101%	0	0

Tabelle 4.6: Regionenbildung mit Kantenfilter, schwellwertbasierte Weißerkennung

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1314	87%	1	5	64	94%	0	0
	gleichmäßig	1042	105%	2	1	80	107%	0	0
	dunkel	336	92%	0	0	9	100%	0	0
	unscharf	416	74%	1	1	0	0/0	0	0
Upper	**Gesamt**	3108	90%	4	7	153	101%	0	0
Lower	hell	123	103%	0	1	8	89%	0	0
	gleichmäßig	119	95%	0	0	17	85%	0	0
	dunkel	14	117%	0	0	0	0/0	0	0
Lower	**Gesamt**	256	100%	0	1	25	86%	0	0
Gesamt	**Gesamt**	3364	91%	4	8	178	98%	0	0

Tabelle 4.7: Regionenbildung mit am Scangitter ausgerichtetem Kantenfilter, schwellwertbasierte Weißerkennung

Regioneneinteilung ähnlich sind, habe ich mich hier darauf beschränkt, nur die Ergebnisse unter Verwendung der schwellwertbasierten Weißerkennung zu zeigen.

Beim Vergleich der in den Tabellen 4.6 bis 4.8 aufgestellten Ergebnisse der unterschiedlichen Ansätze zur Regioneneinteilung, fällt zunächst auf, dass mit keinem davon so viele Linien erkannt werden, wie mit den alten `ColorScanLineRegions`. Das betrifft vor allem weit entfernte Linien, die in den Bildern nur wenige Pixel breit sind und durch Bewegungsunschärfe schwer auszumachen sind. Häufig bekommen sie gar keine eigene Scanlinienregion, sondern werden direkt dem Feld mit zugeordnet. Wenn dieser Fall eintritt, wird es für die nachfolgende Linienerkennung unmöglich die weit entfernten Linien zu erkennen. Im Nahbereich funktionieren alle drei Ansätze zur Regioneneinteilung gut, wie sich an den Ergebnissen auf den Bildern der unteren Kamera ablesen lässt. Dort wird die gleiche Linienerkennungsrate erreicht ohne neue Falscherkennungen zu produzieren.

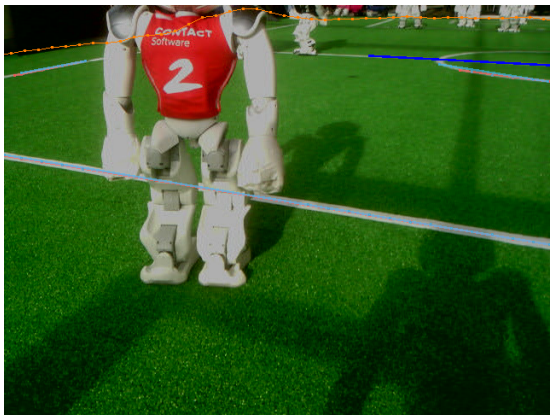
Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1263	83%	11	6	51	75%	0	0
	gleichmäßig	841	85%	2	2	65	87%	0	0
	dunkel	320	88%	0	2	7	78%	0	0
	unscharf	475	84%	3	1	0	0/0	0	0
Upper	**Gesamt**	2899	84%	16	11	123	81%	0	0
Lower	hell	128	108%	0	1	7	78%	0	0
	gleichmäßig	117	94%	0	0	15	75%	0	0
	dunkel	13	108%	0	0	0	0/0	0	0
Lower	**Gesamt**	258	101%	0	1	22	76%	0	0
Gesamt	**Gesamt**	3157	86%	16	12	145	80%	0	0

Tabelle 4.8: inkrementelle Regionenbildung, schwellwertbasierte Weißerkennung

Bei der oberen Kamera gibt es hingegen einige neue Fehlerkennungen und hier liegt auch der hauptsächliche Unterschied zwischen den drei Ansätzen zur Regioneneinteilung. Die Regionenbildung per Kantenfilter und die inkrementelle Regionenbildung erzeugen beide mehr neue falsche Linienfunde, als dass sie Fehler, die bei den alten `ColorScanLineRegions` auftreten, vermeiden. Insgesamt verursacht die Regionenbildung per Kantenfilter sechs und die inkrementelle Regionenbildung vier zusätzliche falsch positive Linienfunde. Im Gegensatz dazu reduziert die Regionenbildung mit einem am Scangitter orientierten Kantenfilter die Anzahl der Fehlerkennungen um vier.

Die falsch positiven Linienfunde lassen sich auf zwei hauptsächliche Problemquellen zurückführen. Einerseits werden häufig Linien in Roboterarmen erkannt oder echte Linien durch Roboterteile künstlich verlängert. Andererseits gibt es Probleme bei Lichtreflexionen am Kunstrasen des Spielfelds. Meistens werden durch Lichtreflexionen nur Linienfunde verhindert, da die Linien nicht mehr genug Kontrast haben, wie in Abbildung 4.5a zu sehen ist können sie aber auch zu Fehlerkennungen führen. Das erklärt auch die durchweg etwas niedrigere Linienerkennungsrate bei den hellen Logs, da es bei diesen während der Aufnahme teilweise direktes Sonnenlicht gab. Schatten, beispielsweise durch Sonne von hinten wie in 4.5b, verursachen bei keiner der drei Methoden Probleme.

(a) Reflexion des Sonnenlichts am Kunstrasen kann zu Fehlerkennungen führen



(b) Schatten bei Sonne von hinten verursachen keine Probleme

Abbildung 4.5: Ergebnisse bei Sonne und Schatten

Bei der Verwendung der Regionenbildung mit am Scangitter ausgerichtetem Kantenfilter, konnte ich Fehlerkennungen durch direktes Sonnenlicht nicht feststellen. Damit eignet sich dieser Ansatz tatsächlich für alle Beleuchtungsbedingungen. Als letzte Auffälligkeit bleibt noch zu bemerken, dass dieser Ansatz eine geringere Erkennungsrate auf dem unscharfen Log aufweist als die anderen beiden, obwohl er ursprünglich als Verbesserung des einfachen Kantenfilteransatzes zum Finden sanfter Helligkeitsverläufe gedacht war. Das liegt möglicherweise daran, dass mit dem einfachen Kantenfilter, wenn er denn auf die Linien anspringt nur der eindeutig weiße Kernbereich als weiß erkannt wird, während der Scangitter Ansatz häufig die Randbereiche, in denen Feld und Linie vermischt zu sehen sind, mit hinzu nimmt. Das ergibt dann gelegentlich weiße Bereiche, die breiter sind als das, was der `LinePerceptor` als Linie akzeptiert.

Insgesamt hat sich damit die Regionenbildung mit am Scangitter ausgerichtetem Kantenfilter als eindeutig am Besten geeignet herausgestellt. Deshalb verwende ich in der weiteren Auswertung nur diese Variante.

Der nächste Schritt in der Segmentierung, für den ich mehrere Ansätze implementiert habe, ist die Klassifikation weißer Regionen (Abschnitt 3.3.3). Die zwei Hauptmethoden sind dabei die Klassifikation anhand der Feldregionen auf der selben Scanlinie einerseits und andererseits die Klassifikation mittels Schwellwerten, die von der ermittelten Feldfarbe abhängen. Ich habe beide Ansätze mit allen Ansätzen zur Regionenbildung getestet. Ergänzend dazu habe ich noch die Weißsegmentierung bei der Regioneneinteilung genutzt, welche bei allen Testläufen aktiv war. Da die Regionenbildung mit Kantenfilter auf dem Scangitter eindeutig die effektivste ist, habe ich in Tabelle 4.9 nur die Ergebnisse der Weißklassifikation anhand von Feldregionen auf der gleichen Scanlinie bei Verwendung der Regionen der Scangittermethode aufgeführt.

Kamera	Logs	Linien				Mittelkreise			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1216	80%	1	5	63	93%	0	0
	gleichmäßig	997	100%	2	1	82	109%	0	0
	dunkel	321	88%	0	0	10	111%	0	0
	unscharf	394	70%	1	1	0	0/0	0	0
Upper	**Gesamt**	2928	85%	4	7	155	102%	0	0
Lower	hell	123	103%	0	1	9	100%	0	0
	gleichmäßig	119	95%	0	0	17	85%	0	0
	dunkel	14	117%			0	0/0	0	0
Lower	**Gesamt**	256	100%	0	1	26	90%	0	0
Gesamt	**Gesamt**	3184	86%	4	8	181	100%	0	0

Tabelle 4.9: Weißklassifikation anhand von Feldregionen auf gleicher Scanlinie

Der Vergleich mit Tabelle 4.7 zeigt, dass die gleiche Menge falsch positiver Linienfunde auftritt, insgesamt aber deutlich weniger Linien gefunden werden. Bei den anderen Methoden zur Regionenbildung bewirkt die Weißklassifikation anhand der Feldregionen auf der gleichen Scanlinie zusätzlich eine Reduktion der falschen Linienfunde. Jedoch nicht auf das Maß herunter, das allein durch die Verwendung der Scangitter Methode bereits erreicht ist. Ebenso verringert diese Art der Weißklassifikation auch bei den anderen Regioneneinteilungen die Gesamtzahl der gefundenen Linien deutlich.

Kamera	Logs	Linien				Mittelkreise			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1134	75%	2	5	63	93%	0	0
	gleichmäßig	980	99%	1	1	79	105%	0	0
	dunkel	313	86%	0	0	9	100%	0	0
	unscharf	403	72%	1	1	0	0/0	0	0
Upper	**Gesamt**	2818	82%	4	7	151	99%	0	0

Tabelle 4.10: Weißklassifikation mit Schwellwerten, ohne Weißsegmentierung bei der Regioneneinteilung

Zum Test der Auswirkungen der Weißklassifikation bei der Regioneneinteilung habe ich einen Testlauf mit den zuvor als am Besten identifizierten Methoden gemacht, bei dem die Weißklassifikation bei der Regioneneinteilung ausgeschaltet war. Die Ergebnisse auf den Bildern der oberen Kamera sind in Tabelle 4.10 aufgeführt. Gegenüber der Ausführung mit der ergänzenden Weißklassifikation werden 290 Linien weniger gefunden. Das betrifft hauptsächlich helle und unscharfe Bilder, da dort weit entfernte Linien sonst von der Felderkennung häufig dem Feld zugeordnet werden. Es gibt keine Auswirkung auf die Falschpositivrate. Auf die Segmentierung der Bilder der unteren Kamera hat diese Änderung keine Auswirkungen.

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1348	89%	2	4	63	93%	0	0
	gleichmäßig	991	100%	2	1	80	107%	0	0
	dunkel	329	90%	0	1	7	78%	0	0
	unscharf	469	83%	1	1	0	0/0	0	0
Upper	**Gesamt**	3137	91%	5	7	150	99%	0	0
Lower	hell	124	104%	0	1	8	89%	0	0
	gleichmäßig	114	91%	0	0	17	85%	0	0
	dunkel	14	117%	0	0	0	0/0	0	0
Lower	**Gesamt**	252	98%	0	1	25	86%	0	0
Gesamt	**Gesamt**	3389	92%	5	8	175	97%	0	0

Tabelle 4.11: Regionenbildung mit am Scangitter ausgerichtetem Kantenfilter auf Differenz von Helligkeit und Sättigung

Weniger hilfreich ist die Verwendung der Sättigung im Kantenfilter der Regioneneinteilung. Die Idee war den Kantenfilter auf der Differenz aus Helligkeit und Farbsättigung zu berechnen, da Feldlinien sich sowohl durch hohe Helligkeit als auch niedrige Sättigung auszeichnen. Wie Tabelle 4.11 zeigt, erhöht sich die Anzahl der gefundenen Linien lediglich um 25 gegenüber der Nutzung eines Kantenfilter, der nur auf die Helligkeit angewendet wird. Die zur Ausführung benötigte Laufzeit ist hingegen deutlich erhöht, was in Abschnitt 4.3.2 festgestellt wird.

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1315	87%	2	4	63	93%	0	0
	gleichmäßig	1016	102%	2	1	80	107%	0	0
	dunkel	347	95%	0	0	9	100%	0	0
	unscharf	419	74%	1	1	0	0/0	0	0
Upper	**Gesamt**	3097	90%	5	6	152	100%	0	0

Tabelle 4.12: Keine Bildglättung im Nahbereich der oberen Kamera

Abschließend habe ich noch die Auswirkungen der Aufteilung der Bilder der oberen Kamera in einen Nah- und einen Fernbereich getestet. Dabei wird im Nahbereich zusätzliche Bildglättung angewendet, was Bildrauschen und damit Fehlerkennungen verhindern soll. Während bei allen vorherigen Tests die zusätzliche Bildglättung im Nahbereich der oberen Kamera aktiv war, habe ich sie für diesen Test deaktiviert. Ansonsten habe ich die Regionenbildung per Kantenfilter auf dem Scangitter und die schwellwertbasierte Weißklassifikation genutzt. Die Ergebnisse in Tabelle 4.12 zeigen wie erwartet eine ähnliche Anzahl erkannter Linien und Kreise. Der Vergleich mit Tabelle 4.7 macht außerdem deutlich, dass die zusätzliche Glättung im Nahbereich drei falsche Linienfunde verhindert.

Ergebnisse des kalibrierungsfreien `LinePerceptors` mit alten `ColorScanLineRegions`

In diesem Abschnitt habe ich den veränderten `LinePerceptor` mit den alten von der manuellen Farbkalibrierung abhängigen `ColorScanLineRegions` als Eingabedaten getestet. Auch hier ist wieder das Ziel die Leistung des Moduls getrennt von den veränderten Eingabedaten zu betrachten. Ich habe vier Konfigurationen des neuen `LinePerceptor` Moduls getestet. Sie unterscheiden sich einerseits darin, ob die Vergleichspunktauswahl zur Validierung neuer Liniensegmente perspektivisch korrekt (Abschnitt 3.4.1) oder rein in Bildkoordinaten (3.4.2) ausgeführt wird, und andererseits, ob die erweiterte `LineSpot`-Auswahl (Abschnitt 3.4.3) genutzt wird.

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1455	96%	0	5	73	107%	0	0
	gleichmäßig	1067	107%	0	1	77	103%	0	0
	dunkel	353	97%	0	2	11	122%	0	0
	unscharf	444	79%	0	2	3	3/0	0	0
Upper	**Gesamt**	3319	97%	0	10	164	108%	0	0
Lower	hell	120	101%	0	0	10	111%	0	0
	gleichmäßig	126	101%	0	0	17	85%	0	0
	dunkel	12	100%	0	0	0	0/0	0	0
Lower	**Gesamt**	258	101%	0	0	27	93%	0	0
Gesamt	**Gesamt**	3577	97%	0	10	191	106%	0	0

Tabelle 4.13: Perspektivisch korrekte Vergleichspunkte, alte `LineSpot` Auswahl

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1537	101%	1	5	73	107%	0	0
	gleichmäßig	1080	109%	0	1	76	101%	0	0
	dunkel	351	96%	0	2	11	122%	0	0
	unscharf	548	97%	0	1	3	3/0	0	0
Upper	**Gesamt**	3516	102%	1	9	163	107%	0	0
Lower	hell	120	101%	0	0	10	111%	0	0
	gleichmäßig	126	101%	0	0	18	90%	0	0
	dunkel	12	100%	0	0	0	0/0	0	0
Lower	**Gesamt**	258	101%	0	0	28	97%	0	0
Gesamt	**Gesamt**	3774	102%	1	9	191	106%	0	0

Tabelle 4.14: In Bildkoordinaten berechnete Vergleichspunkte, alte `LineSpot` Auswahl

In den Tabellen 4.13 und 4.14 sind die Ergebnisse unter Verwendung der alten `LineSpot` Auswahl aufgezeichnet. Der neue Linienerkenner erreicht nahezu die gleiche Anzahl korrekt identifizierter Linien, wie der bisherige und sogar eine etwas höhere Rate erkannter Mittelkreise.

Zusätzlich werden einige falsche Linienerkennungen des alten Moduls vermieden. Ebenso erfreulich ist, dass die Erkennungsraten über alle Beleuchtungsbedingungen hinweg ähnlich sind. Auffällig ist, dass bei der perspektivisch korrekten Vergleichspunktauswahl die Linienerkennung auf unscharfen Bildern deutlich schlechter abschneidet als zuvor.

Die ursprünglich als Möglichkeit zur Laufzeitverbesserung gedachte Berechnung der Vergleichspunkte in Bildkoordinaten sorgt interessanterweise für eine deutliche Verbesserung auf unscharfen Bildern. Ebenso ist die Erkennungsrate weit entfernter Linien in den übrigen Logs etwas höher. Das liegt vermutlich daran, dass weit entfernte Linien durch Bewegungsunschärfe stärker verwischt werden und damit mehr Fläche einnehmen, wodurch die perspektivisch korrekt gesetzten Vergleichspunkte im verwischten Übergangsbereich zwischen Feld und Linie landen können. Die Berechnung der Vergleichspunkte in Bildkoordinaten sorgt tendenziell für etwas größere Abstände zwischen Test- und Vergleichspunkt, was bei verwischten Linien vorteilhaft ist. Allerdings sorgt sie auch für zwei falsche Linienfunde mehr gegenüber der perspektivisch korrekten Vergleichspunktberechnung.

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	$\mathrm{tp}_n/\mathrm{p}_a$	fp_n	tn_n	tp_n	$\mathrm{tp}_n/\mathrm{p}_a$	fp_n	tn_n
Upper	hell	1501	99%	0	5	72	105%	0	0
	gleichmäßig	1100	111%	0	1	79	105%	0	0
	dunkel	371	102%	0	2	12	133%	0	0
	unscharf	461	82%	0	1	4	4/0	0	0
Upper	**Gesamt**	3433	100%	0	10	167	110%	0	0
Lower	hell	299	251%	1	0	22	244%	1	0
	gleichmäßig	219	175%	0	0	26	130%	0	0
	dunkel	36	300%	0	0	0	0/0	0	0
Lower	**Gesamt**	554	216%	1	0	48	166%	1	0
Gesamt	**Gesamt**	3987	108%	1	10	215	119%	1	0

Tabelle 4.15: Perspektivisch korrekte Vergleichspunkte, erweiterte `LineSpot` Auswahl

Im Vergleich dazu sorgt die erweiterte `LineSpot` Auswahl (Tabelle 4.15 und 4.16) wie erwartet für eine deutliche Zunahme der mit der unteren Kamera detektierten Feldlinien und Mittelkreise. Auch einige Linien am unteren Rand der oberen Kamera werden damit zusätzlich detektiert. Die Befürchtung einer erhöhten Anzahl falsch positiver Linienfunde durch als Linien erkannte Roboterteile hat sich nicht bestätigt. Es gibt zwar bei Verwendung der perspektivisch korrekten Vorgehensweise einen – bei der perspektivisch inkorrekten Vorgehensweise zwei – falschen Linienfund und einen falschen Mittelkreis mehr. Allerdings ist ein zusätzlicher falscher Linienfund bei 420 neuen korrekt detektierten Linien aus meiner Sicht vollkommen akzeptabel. Insgesamt spricht das meines Erachtens dafür, die erweiterte `LineSpot` Auswahl beizubehalten.

Bei den in Bildkoordinaten berechneten Vergleichspunkten zur Liniensegmentvalidierung ist die Quote hingegen deutlich schlechter mit zwei zusätzlichen Falschdetektionen für nur etwa 200 zusätzlich erkannte Linien. Aufgrund der hohen Priorität der Vermeidung von Fehlerkennungen spricht das gegen die vorbehaltlose Nutzung dieser Methode. Die guten Ergebnisse auf unscharfen Bildern und bei weit entfernten Linien lassen darauf schließen, dass

die Anwendung dieser Methode exklusiv auf den Fernsichtbereich die Erkennungsgenauigkeit insgesamt verbessern könnte. Diesen Ansatz habe ich jedoch nicht getestet.

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1582	104%	0	4	72	105%	0	0
	gleichmäßig	1113	112%	0	1	79	105%	0	0
	dunkel	375	103%	0	2	12	133%	0	0
	unscharf	570	101%	0	1	4	4/0	0	0
Upper	**Gesamt**	3640	106%	0	8	167	110%	0	0
Lower	hell	298	250%	1	0	22	244%	1	0
	gleichmäßig	220	176%	0	0	26	130%	0	0
	dunkel	36	300%	1	0	0	0/0	0	0
Lower	**Gesamt**	554	216%	2	0	48	166%	1	0
Gesamt	**Gesamt**	4194	114%	2	8	215	119%	1	0

Tabelle 4.16: In Bildkoordinaten berechnete Vergleichspunkte, erweiterte `LineSpot` Auswahl

Die zweite Änderung am `LinePerceptor`, die nicht direkt mit der Umstellung auf kalibrierungsfreie Linienerkennung zu tun hat, habe ich erst in Reaktion auf die Ergebnisse des Gesamtsystems vorgenommen (siehe Abschnitte 3.4.3 und 4.2.3). Es handelt sich um das korrigierte Vorgehen beim Verbinden der LineSpots auf vertikalen Scanlinien, was nur die obere Kamera betrifft. Ich habe deshalb noch die Ergebnisse des neuen `LinePerceptor`s mit korrigiertem Verbindungsverhalten, aber mit der alten LineSpot-Auswahl und perspektivisch korrekter Vergleichspunktauswahl in Kombination mit den alten `ColorScanLineRegions` auf der oberen Kamera in Tabelle 4.17 festgehalten.

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	tp_n/p_a	fp_n	tn_n	tp_n	tp_n/p_a	fp_n	tn_n
Upper	hell	1428	94%	0	5	71	104%	0	0
	gleichmäßig	1039	105%	0	1	77	103%	0	0
	dunkel	345	95%	0	2	11	122%	0	0
	unscharf	420	75%	0	3	3	3/0	0	0
Upper	**Gesamt**	3232	94%	0	11	162	107%	0	0

Tabelle 4.17: Korrigiertes Vorgehen beim Verbinden der LineSpots auf vertikalen Scanlinien

Interessanterweise hat diese Änderung unter Verwendung der alten Scanliniensegmentierung kaum eine Auswirkung auf die Genauigkeit der Linienerkennung. Der Vergleich mit Tabelle 4.13 ergibt tatsächlich eine kleine Verringerung der Anzahl der gefundenen Linien und Mittelkreise, sowie einen falsch positiven Linienfund weniger.

4.2.3 Genauigkeit des Gesamtsystems

Bei der Evaluation der entwickelten Linienerkennung im Gesamtsystem habe ich jeweils die in den Tests der Einzelmodule als optimale ermittelte Konfiguration genutzt. Für die Scanliniensegmentierung bedeutet das die Nutzung der Regioneneinteilung mit Kantenfilter auf dem Scangitter und Weißklassifikation per aus der Feldfarbe berechneten Schwellwerten, sowie die zusätzliche Nutzung der Weißklassifikation bei der Regioneneinteilung. Für das eigentliche Linienerkennungsmodul muss beachtet werden, dass ich bestehenden Code weiterentwickelt und dabei zwei Änderungen vorgenommen habe, die nicht direkt mit der Kalibrierungsfreiheit in Verbindung stehen. Ich habe daher zwei Evaluationen durchgeführt. Einmal nur mit den Änderungen, die zur kalibrierungsfreien Linienerkennung nötig sind, unter Verwendung perspektivisch korrekt berechneter Vergleichspunkte beim Validieren neuer Liniensegmente (Abschnitt 3.4.1). Diese Ergebnisse sind in Tabelle 4.18 dargestellt. Im zweiten Evaluationsdurchgang wurden zusätzlich die Erweiterungen für erhöhte Linienerkennungsraten verwendet, die auf Wettbewerben auch mit eingesetzt werden sollen, zur kalibrierungsfreien Linienerkennung jedoch nicht zwingend nötig sind (Abschnitt 3.4.3, Tabelle 4.19).

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	$\mathrm{tp}_n/\mathrm{p}_a$	fp_n	tn_n	tp_n	$\mathrm{tp}_n/\mathrm{p}_a$	fp_n	tn_n
Upper	hell	1285	85%	1	8	44	65%	0	0
	gleichmäßig	1177	118%	1	2	54	72%	0	0
	dunkel	295	81%	0	2	11	122%	0	0
	unscharf	180	32%	2	2	1	1/0	0	0
Upper	**Gesamt**	2918	85%	4	14	110	72%	0	0
Lower	hell	119	100%	0	1	8	89%	0	0
	gleichmäßig	118	94%	0	0	16	80%	0	0
	dunkel	14	117%	0	0	0	0/0	0	0
Lower	**Gesamt**	251	98%	0	1	24	83%	0	0
Gesamt	**Gesamt**	3169	86%	4	15	134	74%	0	0

Tabelle 4.18: Ergebnisse des Gesamtsystems ohne die weiteren Änderungen am `LinePerceptor`

Wie sich aus den Ergebnissen der Genauigkeit der einzelnen Module schon erahnen ließ (vgl. Tabellen 4.7 und 4.13), findet das kalibrierungsfreie Linienerkennungssystem auf hellen und dunklen Bilder der oberen Kamera weniger Linien als das alte System, auf gleichmäßig ausgeleuchteten Bildern hingegen mehr als zuvor. Auf den Bildern der unteren Kamera findet das neue System unabhängig von der Beleuchtung ungefähr so viele Linien wie zuvor. Insgesamt findet es über die 3692 Testbilder 3169 Linien, was 0.86 mal so viele Linien, wie die des alten Linienerkenners sind. Dabei vermeidet es 15 Fehldetektionen, während es nur vier neue falsche Linienfunde einführt, seine gefundenen Linien sind also sehr zuverlässig richtig.

Die hauptsächlichen Probleme, die zu nicht erkannten Linien führen, sind Bewegungsunschärfe und Verdeckung von Teilen der Linien durch Roboter. Unscharfe oder verwischte Linien verursachen sowohl bei der Regioneneinteilung, als auch bei Farbklassifikation in der Scanliniensegmentierung und bei der Validierung vermuteter Liniensegmente, zu Problemen. Bei den Tests der einzelnen Module erreichen beide in Kombination mit dem jeweiligen alten Modul eine Linienerkennungsquote von etwa drei Vierteln des alten Linienerkenners. In

Kombination bleibt auf dem Log mit ausschließlich unscharfen Bildern gerade mal knapp ein Drittel der ursprünglichen Linienfunde über. In Teilen verdeckte Linien werden vom kalibrierungsfreien Linienerkenner meistens in mehreren Teilstücken erkannt, manchmal wird auch nur ein Linienstück auf einer Seite des verdeckenden Roboters gefunden und das auf der anderen Seite nicht. Das liegt daran, dass die Validierung neuer Liniensegmente davon ausgeht, dass seine Vergleichspunkte bei echten Linien immer im Spielfeld liegen. Wird nun ein Punkt im die Linie verdeckenden Roboter überprüft, liegen die Vergleichspunkte im Roboter und das Liniensegment wird abgelehnt. Der alte `LinePerceptor` verließ sich dafür stattdessen auf das farbsegmentierte Bild, in welchem die meisten Roboterteile als weiß klassifiziert sind, weshalb die hinter dem Roboter verlaufende Linie meist akzeptiert wurde.

Eine für mich zum Zeitpunkt dieser Evaluation völlig unerwartete Auffälligkeit ist, dass viele Linien und Mittelkreise im Nahbereich der oberen Kamera nicht gefunden werden. Nicht gefundene Linien im Nahbereich traten beim Test der einzelnen Module kaum auf. Ebenso findet das Gesamtsystem deutlich weniger Mittelkreise als zuvor. Das zugrunde liegende Problem steht im Zusammenhang mit den variierenden Längen der vertikalen Scanlinien und ist in Abschnitt 3.4.3 genauer erläutert. Bei der Evaluation der einzelnen Module trat es noch nicht auf, sondern erst durch die Kombination beider. Den genauen Grund dafür konnte ich leider nicht herausfinden.

(a) Nicht gefunden Linie im Nahbereich (rot). Weiter entfernte Linien werden gefunden (blau).

(b) Nicht mehr gefundene Linien im Mittelkreis, die auch zum Finden des Mittelkreises benötigt wurden.

Abbildung 4.6

In der nachfolgenden Tabelle sind die Ergebnisse des Gesamtsystems mit den zusätzlichen Veränderungen dargestellt. Ich habe gegenüber der obigen Evaluation des Gesamtsystems keine Änderungen an der Konfiguration der Scanliniensegmentierung vorgenommen, aber für die Linienerkennung die nicht die Kalibrierungsfreiheit betreffenden Änderungen aktiviert. Die erweitere LineSpot-Auswahl an Bildrändern und die korrigierte Verbindungsweise von LineSpots auf vertikalen Scanlinien waren beide aktiv.

Aus der Tabelle ist sofort ersichtlich, dass diese Änderungen für eine hohe Zunahme an gefundenen Linien und Mittelkreisen in beiden Kameras sorgen, allerdings auch für einige neue Fehldetektionen. Auf unscharfen Bildern schneidet der neue Linienerkenner nach wie vor schlecht ab, mit nur 78% der Linienfunde des alten Moduls und zwei zusätzlichen Fehlerkennungen. Insgesamt werden deutlich mehr Linien und Mittelkreise detektiert, bei einer gleichzeitig etwas geringeren Gesamtzahl an Fehldetektionen.

Kamera	**Logs**	**Linien**				**Mittelkreise**			
		tp_n	$\mathrm{tp}_n/\mathrm{p}_a$	fp_n	tn_n	tp_n	$\mathrm{tp}_n/\mathrm{p}_a$	fp_n	tn_n
Upper	hell	1782	118%	1	6	74	109%	0	0
	gleichmäßig	1574	158%	1	2	73	97%	0	0
	dunkel	428	118%	0	2	16	178%	0	0
	unscharf	440	78%	3	1	3	3/0	0	0
Upper	**Gesamt**	4224	123%	5	11	166	109%	0	0
Lower	hell	332	279%	1	1	25	278%	0	0
	gleichmäßig	220	176%	1	0	25	125%	0	0
	dunkel	68	567%	0	0	0	0/0	0	0
Lower	**Gesamt**	620	242%	2	1	50	172%	0	0
Gesamt	**Gesamt**	4844	131%	7	12	216	119%	0	0

Tabelle 4.19: Ergebnisse des Gesamtsystems mit den weiteren Änderungen am `LinePerceptor`

Die zwei wesentlichen Punkte, die ich nachfolgend genauer analysiere, sind die Fehlerkennungen und die (Un-)Abhängigkeit von der Beleuchtung. Aus Tabelle 4.18 ist bekannt, dass das alte Linienerkennungssystem auf dem Testdatensatz mindestens 15 falsche Linienfunde macht. In Kombination mit Tabelle 4.17, wo ein zusätzlicher Fehler des alten Systems auf den unscharfen Bilder aufgedeckt wurde, sind damit 16 falsch positive Detektionen des alten Linienerkennungssystems bekannt. Ob es noch weitere Fehler macht, die der neue Linienerkenner wiederholt, ist durch die gewählte Evaluationsmethode nicht bekannt. Ich gehe allerdings davon aus, dass es, falls es sie gibt, nur wenige Fälle sind. Unter dieser Annahme kann nun die Precision, also der Anteil der korrekten Linienfunde an der Gesamtheit aller Positivangaben, berechnet werden. In Abbildung 4.7 ist die errechnete Precision über alle Testbilder unter Annahme verschiedener falsch positiv Anzahlen des alten Moduls dargestellt. Die Mittelkreise habe ich dabei nicht mit eingehen lassen, da diese separat von den Linien detektiert werden. Zudem konnte ich bei keinem der drei Systeme Fehler bei der Mittelkreiserkennung feststellen. In Bezug darauf ist ihre Precision also gleich.

Wie auch bereits aus den Tabellen eindeutig hervorging, ist das neu entwickelte Linienerkennungsverfahren etwas sicherer in der Vermeidung von Fehlerkennungen. Dass die erweiterte LineSpot-Auswahl und das korrigierte Verbinden vertikaler LineSpots kaum Auswirkungen auf die Falschpositivrate haben, wird ebenfalls deutlich. Die neuen Fehler in der Variante des Moduls mit diesen Änderungen, liegen hoch wahrscheinlich an der erhöhten Anzahl detektierter Linien. Die Wahrscheinlichkeit, dass eine detektierte Linie korrekt ist, bleibt hingegen nahezu unverändert. Geht man von einer hohen Anzahl unentdeckt gebliebener Fehler des alten Moduls aus, könnte sie mit den Änderungen sogar besser sein als ohne.

Nun noch zur Analyse wie gut das kalibrierungsunabhängige System unter verschiedenen Beleuchtungsbedingungen funktioniert. Dazu vergleiche ich die durchschnittliche Anzahl der gefundenen Linien je Bild, unterteilt nach den verschiedenen Beleuchtungsbedingungen. Da die Anzahl der Mittelkreise zu gering für aussagekräftige Schlussfolgerungen ist, habe ich mich auch hier wieder auf die geraden Feldlinien beschränkt. Das Log der dunklen Testbilder ist ebenfalls problematisch, da es auf einem halben Feld aufgenommen wurde und nicht während eines regulären Spiels, weshalb dort deutlich weniger Linien sichtbar sind. Der Vollständigkeit

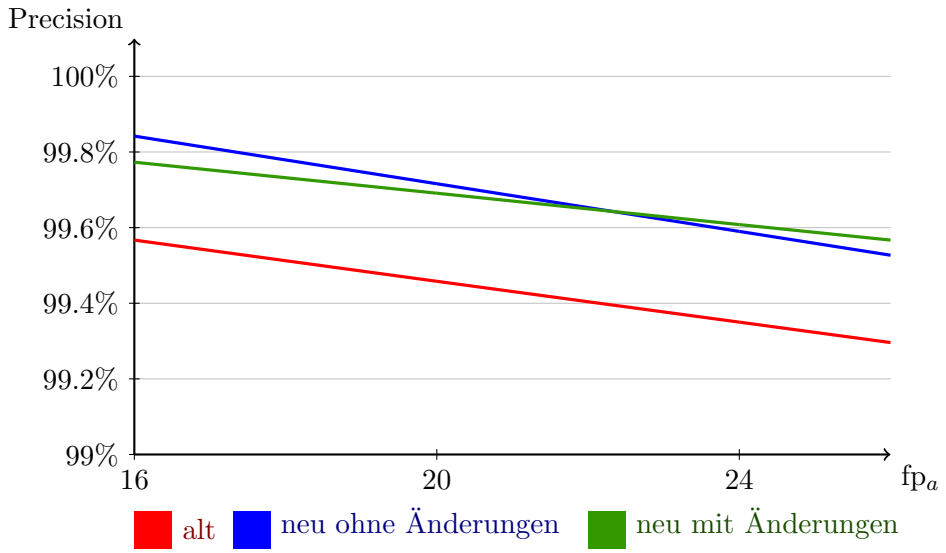
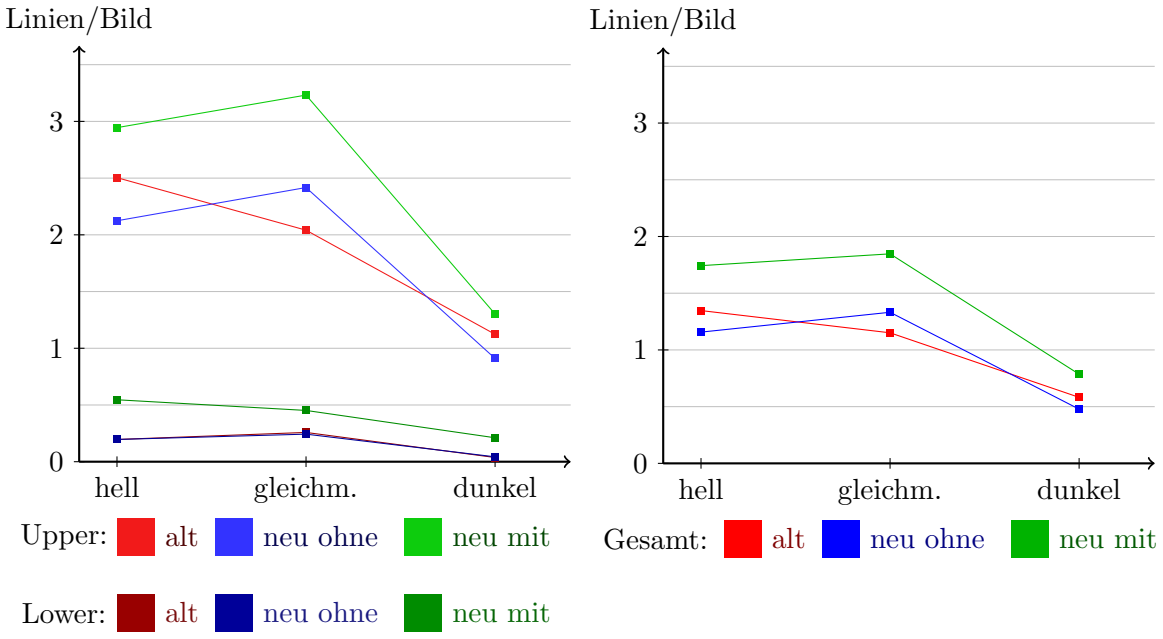


Abbildung 4.7: Precision der Systeme unter Annahme verschiedener falsch positiv Anzahlen des alten `LinePerceptors`

halber habe ich es in den Diagrammen in Abbildung 4.8 dennoch aufgeführt.

Anhand der deutlich vermehrten Linienfunde auf gleichmäßig beleuchteten Bildern gegenüber dem alten Linienerkenner, könnte man zunächst davon ausgehen, dass der neue Linienerkenner dort bei weitem besser funktioniert, als bei starker oder geringer Beleuchtung. Tatsächlich hat aber der alte Linienerkenner auf den verwendeten ausgeglichen beleuchteten Bildern weniger Linien erkannt, als bei den stark beleuchteten Bildern. Beim neuen Modul ergibt sich daher nur eine moderat höhere Quote von Linien je Bild, was auch ausschließlich auf die obere Kamera zutrifft. Das ist nicht verwunderlich, da bei manchen der Bilder mit direkter Sonneneinstrahlung die weit entfernten Linien auch für Menschen schwer auszumachen sind. Die geringe Quote gefundener Linien auf den gleichmäßig beleuchteten Bildern beim alten Linienerkenner könnte an einer suboptimal durchgeführten Farbkalibrierung liegen.

Wie zuvor bereits angemerkt, deutet die geringe Anzahl der Linienfunde in den dunklen Bildern zwar auf eine schlechtere Leistung bei geringer Beleuchtung hin, ist aufgrund der veränderten Umstände aber kein eindeutiges Indiz. In den Bildern der oberen Kamera sind durch die Verwendung eines halben Spielfeldes überhaupt weniger Linien sichtbar. Auf die untere Kamera hat das allerdings fast gar keinen Einfluss, da sie nur den Nahbereich vor den eigenen Füßen abdeckt. Es muss allerdings auch beachtet werden, dass das verwendete dunkle Log nicht während eines Testspiels aufgenommen wurde. Stattdessen besteht es darin, dass der verwendete Roboter dem durch einen Menschen bewegten Ball folgt. Dadurch sind nur gelegentlich Linien in den Bildern der unteren Kamera sichtbar. In den Testspielen stehen Roboter hingegen häufig für längere Zeit so, dass eine Linie über die untere Kamera sichtbar ist. Das trifft insbesondere auf Roboter in der Torwartrolle oder beim Warten auf die Freigabe für den eigenen Anstoß zu. Es ist daher nicht abschließend klar, ob das neu entwickelte Linienerkennungssystem bei geringer Beleuchtung ebenso gut funktioniert wie auf einem gut ausgeleuchteten Spielfeld. Für eine aussagekräftige Beurteilung wären Aufzeichnungen eines kompletten Testspiels bei geringer Beleuchtung auf einem vollständigen Feld oder Datensätze für alle Beleuchtungsbedingungen, zu denen alle Linienverläufe genau bekannt sind, nötig.



(a) Obere und untere Kamera getrennt

(b) Durchschnitt über beide Kameras

Abbildung 4.8: Gefundene Linien je Bild

4.3 Laufzeit

4.3.1 Gesamtsystem

Da die Information, ob ein bestimmtes Bildpixel weiß, grün oder etwas anderes ist, für das alte Linienerkennungssystem durch die Farbkalibrierung vorgegeben war, ließ es sich mit sehr geringer Laufzeit ausführen (siehe Tabelle 4.20). Einzig die Maximallaufzeiten des Mittelkreisdetektor stechen heraus, da dieser recht lange braucht, wenn er mehrere plausible Kandidaten überprüfen muss. Die Durchschnittslaufzeit der Mittelkreisdetektion ist hingegen geringer als die der Linienerkennung, da häufig gar kein Mittelkreiskandidat überprüft werden muss.

Kamera	**Verhalten**	**Maximum**				**Durchschnitt**			
		Hor.	Vert.	Linien	Kreise	Hor.	Vert.	Linien	Kreise
Upper	**stehend**	0.285	0.152	0.153	0.491	0.244	0.122	0.102	0.021
	Kopfdrehung	0.257	0.165	0.193	0.515	0.224	0.114	0.133	0.044
	laufend	0.277	0.179	0.130	0.114	0.206	0.083	0.043	0.007
Lower	**stehend**	0.073	0.052	0.039	0.017	0.052	0.036	0.019	0.002
	Kopfdrehung	0.088	0.051	0.036	0.019	0.065	0.034	0.001	0.002
	laufend	0.127	0.057	0.099	0.075	0.059	0.032	0.029	0.004

Tabelle 4.20: Laufzeiten der alten Module in Millisekunden. „Hor.“ bezeichnet die Laufzeit zum Berechnen der horizontalen Scanliniensegmentierung, „Vert.“ jene für die vertikale

Die Laufzeiten des neuen Systems sind in Tabelle 4.21 aufgezeichnet. Dabei waren die Regioneneinteilung mit Kantenfilter auf dem Scangitter, Weißsegmentierung per Schwellwert und die perspektivisch korrekte Vergleichspunktauswahl beim Validieren neuer Liniensegmente

aktiv. Aufgrund der Corona-Pandemie konnte ich die Laufzeitmessungen nur an einem einzigen Tag ausführen. Da ich an der erweiterten LineSpot-Auswahl und der Verbindung von LineSpots auf vertikalen Scanlinien nach diesem Tag noch Änderungen vorgenommen habe, konnte dies nicht auf einem Roboter getestet werden. Das betrifft nur die Laufzeiten zur Erkennung von Linien und Mittelkreisen. Auf dem zum entwickeln genutzten Rechner konnte ich mit diesen Änderungen einen geringen Laufzeitanstieg messen.

Kamera	**Verhalten**	**Maximum**				**Durchschnitt**			
		Hor.	Vert.	Linien	Kreise	Hor.	Vert.	Linien	Kreise
Upper	**stehend**	0.955	1.508	0.776	1.788	0.840	1.267	0.515	0.124
	Kopfdrehung	0.875	1.426	0.741	1.374	0.706	1.046	0.172	0.495
	laufend	0.934	1.266	0.691	1.039	0.399	0.523	0.110	0.032
Lower	**stehend**	0.215	0.478	0.045	0.010	0.179	0.400	0.003	0.015
	Kopfdrehung	0.323	0.468	0.224	0.159	0.230	0.382	0.071	0.058
	laufend	0.319	0.532	0.099	0.374	0.214	0.374	0.141	0.035

Tabelle 4.21: Laufzeiten der neuen Module in Millisekunden

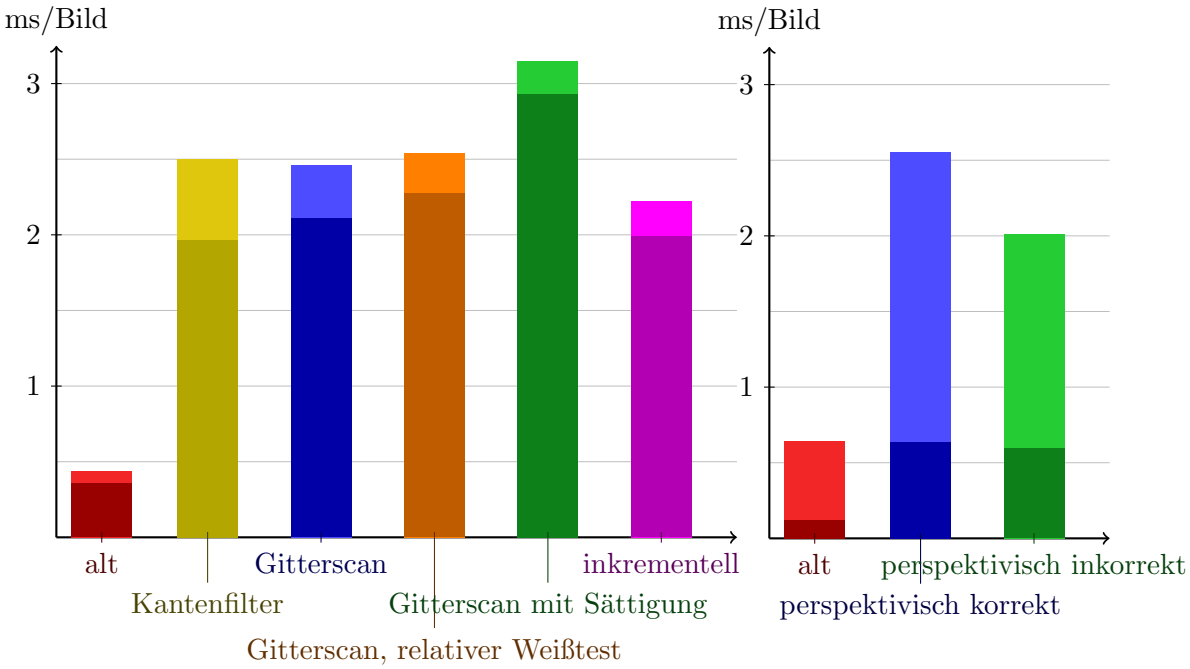
Mit durchschnittlich 2,1 Millisekunden zur Berechnung der Scanliniensegmentierung und 2,5 Millisekunden im Maximum auf scharfen Bildern der oberen Kamera benötigt sie nun deutlich länger. Auf durch Laufbewegungen unscharfen Bilder liegt der Durchschnitt mit 0.9 Millisekunden deutlich niedriger, weil dort weniger unterschiedliche Regionen erkannt werden. Das beschleunigt die Regioneneinteilung etwas und alle nachfolgenden Schritte deutlich. Die Laufzeit des Linienerkenners ist auf scharfen Bildern der oberen Kamera etwa verfünffacht. Seine Durchschnittslaufzeit auf während Bewegungen aufgenommenen Bildern ist gegenüber dem alten Modul nur ein wenig erhöht. Auch die hohen Maximallaufzeiten der Mittelkreiserkennung treten durch die Möglichkeit mehrerer Mittelkreiskandidaten weiterhin auf.

Aus der Summe der Maxima ergibt sich eine Worst Case Laufzeit von fünf Millisekunden je Bild der oberen Kamera. Aufgrund der Bildrate von 30 Bildern pro Sekunde, dürfen pro Bild maximal 33 Millisekunden aufgewendet werden, um die Linienerkennung und alle anderen Verarbeitungsschritte wie Ball- und Hindernisdetektion auszuführen. Mit dem aktuellen Entwicklungsstand ist das neue Linienerkennungssystem schnell genug, um diese Beschränkung einzuhalten. Zur Verringerung der Latenz zwischen Bildaufnahme und Beendung der Informationsextraktion ist es dennoch sinnvoll, zu versuchen die Laufzeit wieder zu verringern. Zwei Möglichkeiten dafür sind, bei der Regioneneinteilung den Kantenfilter nur dann zu berechnen, wenn er tatsächlich benötigt wird (vgl. Abschnitt 3.3.1), sowie die Beschränkung der Mittelkreisdetektion auf den vielversprechendsten Kandidaten, so dass nicht mehrere Mittelkreiskandidaten überprüft werden.

Aufgrund der geringeren Auflösung der Bilder der unteren Kamera, sind die benötigten Laufzeiten dort deutlich niedriger, während mehr Laufzeit zur Bildanalyse frei ist. Für die untere Kamera sind daher keine Probleme zu erwarten.

4.3.2 Vergleich der verschiedenen Ansätze

Ich habe auch einen Vergleich mit den anderen Ansätzen durchgeführt. Die Ergebnisse für Bilder der oberen Kamera, während der Roboter still steht, sind in Abbildung 4.9 dargestellt.



(a) Summe der Laufzeiten für horizontale und vertikale `ColorScanLineRegions`

(b) Laufzeit des `LinePerceptors` (Linien und Kreise)

Abbildung 4.9: Laufzeiten der verschiedenen Implementierungsvarianten auf scharfen Bildern der oberen Kamera. Dunkler Balken: Durchschnitt über 500 Bilder, Heller Balken: Maximallaufzeit

Das linke Diagramm zeigt die summierten Laufzeiten der horizontalen und vertikalen Scanliniensegmentierung. Die drei Varianten der Regioneneinteilung (Kantenfilter, Gitterscan und inkrementell) sind etwa gleich schnell, mit einem etwas höheren Durchschnitt beim Kantenfilter auf dem Scangitter (blau) und einem etwas niedrigerem Maximum beim inkrementellen Vorgehen. Außer dort wo es explizit angegeben ist, habe ich immer die Weißklassifikation anhand der Feldfarbe genutzt. Die Weißsegmentierung bei der Regioneneinteilung war zusätzlich immer aktiv. Die Weißklassifikation anhand der Feldregionen auf der gleichen Scanlinien (orange) ist etwas langsamer als die Weißklassifikation anhand der Feldfarbe (vgl. blau). Die Sättigung in das Kantenfilter mit einzubeziehen (grün) benötigt erwartungsgemäß viel zusätzliche Laufzeit und ist durchschnittlich 0.8 Millisekunden langsamer.

Im rechten Diagramm ist die summierte Laufzeit für Linien- und Mittelkreiserkennung dargestellt. Die perspektivisch inkorrekt arbeitende Variante des neuen `LinePerceptors`, die wo möglich nur in den Bildkoordinaten arbeitet, ist im Mittel minimal schneller als perspektivisch korrekte Variante. Das Maximum liegt deutlich niedriger, aber da es sehr stark von Ausreißern bei der Mittelkreiserkennung abhängt, hat das nur begrenzte Aussagekraft.

Keine der anderen neuen Varianten bietet einen so signifikanten Laufzeitvorteil gegenüber der für ihre guten Linienerkennungsergebnisse ausgewählten Implementierung (jeweils blau), dass es sich lohnen würde zu dieser zu wechseln.

5. Fazit

Mit dieser Arbeit ist es gelungen eine neue kalibrierungsfreie Linienerkennung zu implementieren. Sowohl für die Segmentierung der Scanlinien als auch für die Linien- und Mittelkreiserkennung wurden neue Lösungen entwickelt. Die Scanliniensegmentierung, welche ein wichtiger Vorverarbeitungsschritt für die Linien-, Strafstoßmarken- und auch Ballerkennung, wurde als mehrstufiges Verfahren mit weitgehend voneinander unabhängigen Teilschritten umgesetzt. Für die Teilschritte der Regioneneinteilung und Weißklassifikation wurden mehrere unterschiedliche Ansätze implementiert und verglichen. Zusätzlich wurden weiteren Variationen entwickelt und ausgewertet. Das vorher bereits vorhandene Modul zur Linienerkennung könnte ebenfalls von der Farbkalibrierung unabhängig gemacht werden. Dafür wurden die Validierung gefundener Liniensegmente, Messung der Linienbreite und die Positionsbestimmung des Mittelkreises neu gelöst.

Die Evaluation ergibt, dass mit den Maßnahmen zur Kalibrierungsfreiheit allein weniger Linien gefunden werden als mit dem manuell kalibriertem System. Es wären aber immer noch genug, um in den meisten Situationen die Position des Roboters bestimmen zu können. Die detektierten Linien und Mittelkreise sind dabei mit enormer Zuverlässigkeit richtig, was die schon zuvor sehr niedrige Menge an Fehlerkennungen weiter absenkt. Weitere Verbesserungen und Fehlerbehebungen am Linienerkenner sorgen dafür, dass jetzt insgesamt mehr Linien und Mittelkreise erkannt werden als zuvor, ohne das es Beeinträchtigungen bei der Fehlerrate gibt.

Auf gleichmäßig beleuchten Spielfeldern und bei direktem Sonnenlicht funktioniert die neue Linienerkennung gleichermaßen gut. Ob sie von der Beleuchtung vollständig unabhängig ist, konnte nicht eindeutig festgestellt werden, da für dunkle Umgebungen keine ausreichenden Testdaten vorlagen. Die durchgeführten Tests weisen allerdings auf eine etwas schlechtere Quote der in dunklen Bildern erkannten Linien hin.

Das neu entwickelte Linienerkennungssystem hat einen stark erhöhten Laufzeitbedarf, ist aber immer noch echtzeitfähig. Trotzdem wären Laufzeitverbesserungen sinnvoll, damit die Latenz zwischen Bildaufnahme und fertiger Informationsextraktion wieder abgesenkt wird und mehr Laufzeit für potenzielle Neuentwicklungen frei bleibt. Bei der in Bezug auf die Erkennungsrate optimalen Implementierung, nämlich der Regioneneinteilung durch Anwendung eines Kantenfilters auf das Scangitter, sind noch Laufzeitoptimierungen möglich ohne die Arbeitsweise des Algorithmus zu verändern. Das ist daher ein logischer nächster Schritt.

Insgesamt ist mit dieser Arbeit ein wichtiger Schritt in Richtung einer Bildverarbeitung ohne manuelle Kalibrierung getan. Daran wird derzeit auch von einigen anderen Mitgliedern des Team's B-Human gearbeitet. Die Erprobung des kalibrierungsfreien Systems in Wettbewerben bleibt indes noch abzuwarten.

Literaturverzeichnis

[Bar Hillel et al. 2014] Bar Hillel, Aharon ; Lerner, Ronen ; Levi, Dan ; Raz, Guy: Recent progress in road and lane detection: a survey. In: *Machine Vision and Applications* 25 (2014), S. 727–745. – https://doi.org/10.1007/s00138-011-0404-2 [Abruf: 9.11.2020]

[Burger u. Burge 2015] Burger, Wilhelm ; Burge, Mark J.: *Digitale Bildverarbeitung - Eine algorithmische Einführung mit Java.* 3., vollständig überarbeitete und erweiterte Auflage. Springer Vieweg, Berlin, Heidelberg, 2015. – https://doi.org/10.1007/978-3-642-04604-9 [Abruf: 10.11.2020]

[Cai u. Tai 2005] Cai, Z. Q. ; Tai, J.: Line detection in Soccer Video. In: *2005 5th International Conference on Information Communications Signal Processing*, 2005, S. 538–541. – https://doi.org/10.1109/ICICS.2005.1689104 [Abruf: 9.11.2020]

[Canny 1986] Canny, John: A Computational Approach to Edge Detection. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8 (1986), Nr. 6, S. 679–698. – https://doi.org/10.1109/TPAMI.1986.4767851 [Abruf: 10.11.2020]

[Cario et al. 2017] Cario, Gianni ; Casavola, Alessandro ; Lupia, Marco: Lane Detection and Tracking Problems in Lane Departure Warning Systems. In: Loce, Robert P. (Hrsg.) ; Bala, Raja (Hrsg.) ; Trivedi, Mohan (Hrsg.): *Computer Vision and Imaging in Intelligent Transportation Systems*, 2017, S. 283–304. – https://doi.org/10.1002/9781118971666.ch1 [Abruf: 9.11.2020]

[Cormen et al. 2009] Cormen, Thomas ; Leiserson, Charles E. ; Rivest, Ronald L. ; Stein, Clifford: Data Structures for Disjoint Sets. In: *Introduction to Algorithms, Third Edition*, MIT Press, 2009, S. 561–585. – https://ebookcentral.proquest.com/lib/suub-shib/detail.action?docID=3339142 [Abruf: 11.11.2020]

[Farazi et al. 2015] Farazi, Hafez ; Allgeuer, Philipp ; Behnke, Sven: A Monocular Vision System for Playing Soccer in Low Color Information Environments. In: *The 10th Workshop on Humanoid Soccer Robots 15th IEEE-RAS International Conference on Humanoid Robots, Seoul, Korea, November 2015*, 2015. – http://ais.uni-bonn.de/papers/HSR_2015_Farazi.pdf [Abruf: 8.11.2020]

[Galambos et al. 2000] Galambos, C. ; Matas, J. ; Kittler, J.: Progressive Probabilistic Hough Transform for line detection. In: *Computer Vision and Image Understanding* 78 (2000), Nr. 1, S. 119–137. – https://dspace.cvut.cz/bitstream/handle/10467/9451/1999-Progressive-probabilistic-Hough-Transform-for-line-detection.pdf [Abruf:7.11.2020]

[Jiang Bu et al. 2011] Jiang Bu ; Songyang Lao ; Liang Bai: Automatic line mark recognition and its application in camera calibration in soccer video. In: *2011 IEEE International Conference on Multimedia and Expo*, 2011, S. 1–6. – https://doi.org/10.1109/ICME.2011.6012137 [Abruf: 9.11.2020]

[Jähne 1997] Jähne, Bernd: *Digitale Bildverarbeitung.* Vierte, vollständig überarbeitete und erweiterte Auflage. Springer, Berlin, Heidelberg, 1997. – https://doi.org/10.1007/978-3-662-06733-8 [Abruf: 10.11.2020]

[Müller u. Guido 2017] Müller, Andreas C. ; Guido, Sarah: *Einführung in Machine Learning mit Python*. O'Reilly, 2017

[Narotea et al. 2018] Narotea, Sandipann P. ; Bhujbalb, Pradnya N. ; Narotec, Abhilasha S. ; Dhane, Dhiraj M.: A review of recent advances in lane detection and departure warning system. In: *Pattern Recognition* 73 (2018), S. 216–234. – `https://doi.org/10.1016/j.patcog.2017.08.014` [Abruf: 9.11.2020]

[Popp 2019] Popp, Annemarie: *B-Human Robot*. 2019. – Veröffentlicht durch die DFKI GmbH mit CC BY-NC-ND 4.0 Lizenz – `https://b-log.b-human.de/galleries/b-human-team-and-robots-2019/20190215_B-Human_ap-0334.html` [Abruf: 4.11.2020]

[Reinhardt 2011] Reinhardt, Thomas: *Kalibrierungsfreie Bildverarbeitungsalgorithmen zur echtzeitfähigen Objekterkennung im Roboterfußball*, Hochschule für Technik, Wirtschaft und Kultur Leipzig, mathesis, 2011. – `http://www.thomas-reinhardt.de/Vision/masterthesis_thomas_reinhardt.pdf` [Abruf: 9.9.2020]

[RoboCup Federation 2016] RoboCup Federation: *RoboCup Webseite*. 2016. – `https://www.robocup.org` [Abruf: 3.11.2020]

[RoboCup Federation 2020] RoboCup Federation: *RoboCup Standard Platform League*. 2020. – `https://spl.robocup.org/` [Abruf: 3.11.2020]

[RoboCup Technical Committee] RoboCup Technical Committee: *RoboCup Standard Platform League (NAO) Challenges & Rule Book (DRAFT 2021 rules, as of January 29, 2021)*. – `https://cdn.robocup.org/spl/wp/2021/01/SPL-Rules-2021.pdf` [Abruf: 18.2.2021]

[RoboCup Technical Committee 2019] RoboCup Technical Committee: *RoboCup Standard Platform League (NAO) Rule Book, DRAFT 2020 rules*. 2019. – `https://collaborating.tuhh.de/HULKs/robocup_tc_public/raw/master/SPL-Rules_2020.pdf` [Abruf: 4.11.2020]

[Rodriguez et al. 2019] Rodriguez, Diego ; Farazi, Hafez ; Ficht, Grzegorz ; Pavlichenko, Dmytro ; Brandenburger, André ; Hosseini, Mojtaba ; Kosenko, Oleg ; Schreiber, Michael ; Missura, Marcel ; Behnke, Sven: RoboCup 2019 AdultSize Winner NimbRo: Deep Learning Perception, In-Walk Kick, Push Recovery, and Team Play Capabilities. In: Chalup, Stephan (Hrsg.) ; Niemueller, Tim (Hrsg.) ; Suthakorn, Jackrit (Hrsg.) ; Williams, Mary-Anne (Hrsg.): *RoboCup 2019: Robot World Cup XXIII*. Cham : Springer International Publishing, 2019. – ISBN 978–3–030–35699–6, S. 631–645. – `https://doi.org/10.1007/978-3-030-35699-6_51` [Abruf: 8.11.2020]

[Röfer et al. 2019] Röfer, Thomas ; Laue, Tim ; Baude, Andreas ; Blumenkamp, Jan ; Felsch, Gerrit ; Fiedler, Jan ; Hasselbring, Arne ; Hass, Tim ; Oppermann, Jan ; Reichenberg, Philip ; Schrader, Nicole ; Weiss, Dennis: *B-Human Team Report and Code Release 2019*. 2019. – `https://www.b-human.de/downloads/publications/2019/CodeRelease2019.pdf` [Abruf: 4.9.2020]

[Schwarz et al. 2015] Schwarz, Ingmar ; Hofmann, Matthias ; Urbann, Oliver ; Tasse, Stefan: A Robust and Calibration-Free Vision System for Humanoid Soccer Robots. In: Almeida, Luis (Hrsg.) ; Ji, Jianmin (Hrsg.) ; Steinbauer, Gerald (Hrsg.) ; Luke, Sean (Hrsg.): *RoboCup 2015: Robot World Cup XIX*. Cham : Springer International

Publishing, 2015. – ISBN 978–3–319–29339–4, S. 239–250. – https://doi.org/10.1007/978-3-319-29339-4_20 [Abruf:8.11.2020]

[SoftBank Robotics 2018] SoftBank Robotics: *NAO - Developer Guide*. 2018. – https://developer.softbankrobotics.com/nao6/nao-documentation/nao-developer-guide [Abruf: 3.11.2020]

[Sun u. Liu 2009] Sun, L. ; Liu, G.: Field lines and players detection and recognition in soccer video. In: *2009 IEEE International Conference on Acoustics, Speech and Signal Processing*, 2009, S. 1237–1240. – https://doi.org/10.1109/ICASSP.2009.4959814 [Abruf: 9.11.2020]

[Team B-Human 2020] Team B-Human: *B-Human Team Webseite*. 2020. – https://www.b-human.de/ [Abruf: 2.11.2020]

[Thielke 2016] Thielke, Felix: *Erkennung beliebiger Bälle durch den humanoiden Roboter NAO in der RoboCup Standard Platform League*, Universität Bremen, bathesis, 2016. – https://b-human.de/downloads/theses/bachelor-thesis-fthielke.pdf [Abruf: 28.9.2020]

A. Inhalt des beiliegenden Datenträgers

Dies ist eine Aufstellung der Inhalte des beiliegenden Datenträgers. Alternativ sind die aufgeführten Dateien auch über folgenden Link erhältlich:
`https://seafile.zfn.uni-bremen.de/d/68b1032ecbef4f509b15/`

Inhalt des Datenträgers:

PDF-Version dieser Arbeit als Datei: BA-Monnerjahn.pdf

ReadMe.txt Datei zum Umgang mit dem für diese Arbeit entwickelten Code.

Konfigurationsdateien für den entwickelten Code im Verzechnis Code/Config:

- Scenarios/PerceptionTest/threads.cfg

Build-Dateien für den entwickelten Code im Verzeichnis Code/Make:

- CMake/PerceptionTester.cmake
- Common/CMakeLists.txt

Der für diese Arbeit entwickelte und veränderte Code im Verzeichnis Code/Src:

- Controller
  - Visualization
    - PaintMethods.h
  - LogPlayer.cpp
  - LogPlayer.h
- Modules
  - Configuration
    - AutoExposureWeightTableProvider
      - AutoExposureWeightTableProvider.cpp
    - ConfigurationDataProvider
      - ConfigurationDataProvider.cpp
      - ConfigurationDataProvider.h
  - Perception
    - FieldPerceptors
      - ExpLinePerceptor.cpp
      - ExpLinePerceptor.h
    - ImagePreprocessores

            - RelativeFieldColorsProvider.cpp
            - RelativeFieldColorsProvider.h
        * Scanlines
            - ExpScanLineRegionizer.cpp
            - ExpScanLineRegionizer.h
- Platform
    - Linux
        * SystemCall.cpp
    - macOS
        * SystemCall.cpp
    - Nao
        * SystemCall.cpp
    - Windows
        * SystemCall.cpp
    - File.cpp
    - SystemCall.cpp
    - SystemCall.h
- Representations
    - Configuration
        * RelativeFieldColorsParameters.h
    - Perception
        * ImagePreprocessing
            - BodyContour.cpp
            - BodyContour.h
            - FieldBoundary.cpp
            - FieldBoundary.h
            - RelativeFieldColors.h
- Threads
    - Debug.h
    - Motion.cpp
- Tools

  - Debugging
    - DebugDrawings.h
    - Debugging.h
    - DebugRequest.h
  - Framework
    - Robot.cpp
    - Robot.h
  - ImageProcessing
    - Image.h
  - Math
    - EigenMatrixBaseExtensions.h
  - Settings.cpp
  - Settings.h
- Utils
  - PerceptionTester
    - Controller
      - PerceptionTestRobotConsole.cpp
      - PerceptionTestRobotConsole.h
    - Helper
      - ArgumentParser.h
      - ImagePainter.cpp
      - ImagePainter.h
      - PerceptionLogFilter.h
      - TestModuleFactory.h
    - TestModules
      - BallSpotsTest.h
      - CirclePerceptTest.h
      - LinesPerceptTest.h
      - PenaltyMarkPerceptTest.h
      - TestModuleBase.h
    - Main.cpp