UNIVERSITÄT DORTMUND
FACHBEREICH INFORMATIK

PG 468

Vierbeinige fußballspielende Roboter -
Entwicklung eines dynamischen Weltmodells
basierend auf Zuverlässigkeitsinformation und
Schätzung

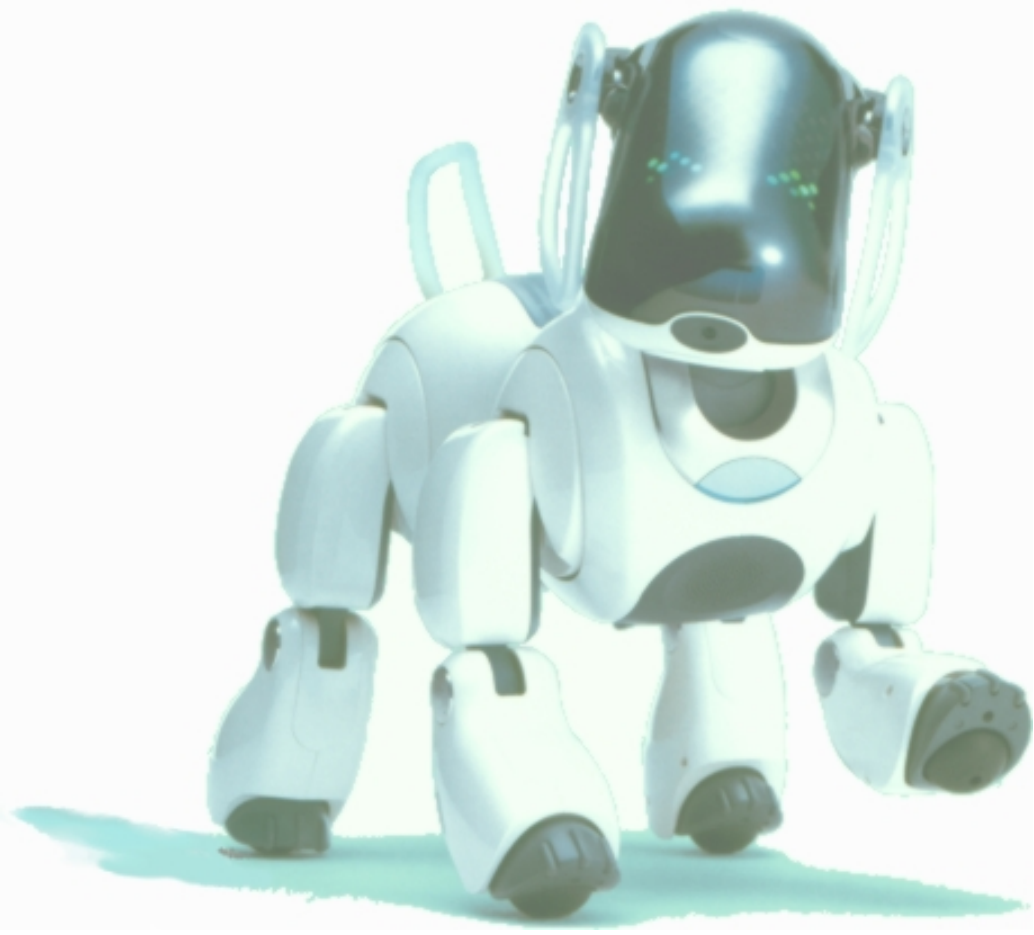
Studenten:

Markus Broszeit	Stefan Czarnetzki	Timo Diekmann
Denis Fisseler	Thorsten Kerkhof	Matthias Meyer
Bastian Schmitz	Carsten Rohde	Xuan-Anh Tran
Tobias Wegner	Judith Winter	Christine Zarges

Betreuer:

Ingo Dahm	Matthias Hebbel	Walter Nistico

Abschlussbericht




Microsoft®
Hellhounds

Inhaltsverzeichnis

1 Einleitung

Projektgruppen sind praxisorientierte Lehrveranstaltungen und ein fester Bestandteil des Informatikstudiums an der Universität Dortmund. Über einen Zeitraum von zwei Semestern bearbeiten bis zu zwölf Studenten selbständig ein vorgegebenes Thema. Dieser Bericht dokumentiert die Arbeiten der Projektgruppe 468.

Die Abwandlung des bekannten Fußballspiels, bei der intelligente Roboter die Rollen der Spieler übernehmen, wird *Roboterfußball* genannt. Im Rahmen des RoboCups[1] wird durch Wettbewerbe im Roboterfußball die Forschung an autonomen, mobilen Robotern in dynamischen Umgebungen motiviert. Erklärtes langfristiges Ziel ist es bis zum Jahr 2050 den Weltmeister im Fußball mit humanoiden Robotern zu schlagen.

Die Microsoft™ Hellhounds[2] sind ein Roboterfußballteam am Institut für Roboterforschung (IRF) der Universität Dortmund. Das Thema der Projektgruppe 468 ist die Entwicklung eines dynamischen Weltmodells basierend auf Zuverlässigkeitsinformationen und Schätzung für die Roboter der Microsoft™ Hellhounds.

1.1 Das Fußballspiel und die Roboter

In der Sony-4-legged-League treten zwei Mannschaften mit je vier Roboterspielern auf einem 4 m × 6 m großen Spielfeld gegeneinander an. Den Untergrund bildet grüner Kunstrasenteppich, auf dem weiße Feldlinien aufgebracht sind. An den Spielfeldenden befinden sich das gelbe und das blaue Tor. Als zusätzliche Orientierungshilfe stehen längs des Feldes vier farbig kodierte Zylinder, die sogenannten Landmarken. Es wird mit einem orangenem Ball von ca. 10 cm Durchmesser zwei Halbzeiten zu je zehn Minuten gegeneinander gespielt. Es gibt dabei drei Feldspieler und einen Torwart (Roboter mit der Nummer 1). Ein (menschlicher) Schiedsrichter überwacht während des Spiels die Einhaltung der Regeln. Bei Verstößen, wie zum Beispiel dem Verlassen des Spielfeldes oder dem Betreten des eigenen Strafraumes durch einen Feldspieler, wird der Roboter für 30 Sekunden aus dem Spiel genommen. Ebenso ist es nicht erlaubt, den Ball über längere Zeit zu verdecken oder in bestimmten Situationen andere Spieler anzurempeln. Rollt der Ball vom Spielfeld, wird er an einen von sechs Einwurfpunkten auf das Feld zurückgelegt. Die Roboter dürfen zwar untereinander kommunizieren, aber jede Form der Fernsteuerung ist regelwidrig, sie müssen also autonom handeln. Ziel des Spiels ist es natürlich, mehr Tore zu erzielen als die gegnerische Mannschaft. Abbildung 1.1 zeigt ein Spiel kurz vor dem Anstoß.

[1] `http://www.robocup.org`
[2] `http://fussballhunde.de`

Abbildung 1.1: Spielfeld der Sony-4-legged-League

Im Gegensatz zu anderen Ligen des Roboterfußballs, werden in der Sony-4-legged-League die Roboter nicht selbst entwickelt, sondern es darf nur mit verschiedene AIBO[3]-Robotermodellen gespielt werden – von praktischer Relevanz ist aber auf Grund der Rechenleistung nur der AIBO ERS7. Der Roboter ist mit einer 576 MHz Mips IV Prozessos ausgestattet und besitzt 64 MB Arbeitsspeicher. Die vier Beine des AIBOs haben je drei Freiheitsgrade. Mit ihnen kann er sich mit bis zu 50 cm/s fortbewegen oder den Ball schießen. Der auch mit 3 Freiheitsgraden bewegliche Kopf ist mit einer Kamera ausgestattet, die 30 Bilder pro Sekunde liefert. Abbildung 1.2 zeigt den Roboter und seine Sensoren und Aktoren. Nicht von außen sichtbar sind Beschleunigunssensoren entlang der 3 Raumachsen und Potenziometer zum Auslesen der Servowinkel. Außerdem verfügt der akkubetriebene Roboter über WLAN Hardware die zur Kommunikation mit anderen Robotern genutzt werden kann.

1.2 Ziele der Projektgruppe

Die Zielsetzung bestand darin, die bestehende Spielleistung zu verbessern. Da das Framework zur Entwicklung der Robotersoftware sehr modular aufgebaut ist, konnte parallel an folgenden Programmteilen gearbeitet werden:

- Die CeilingCam ist ein wichtiges Hilfsmittel bei der Entwicklung. Die über dem Spielfeld angebrachte Kamera nimmt Ball und Roboter auf und dient als Referenzdatenquelle für deren Positionen auf dem Spielfeld (siehe Kapitel 2).
- Die WalkingEngine (siehe Kapitel 3) setzt Bewegungsanweisungen für den Roboter in Ansteuerungen für die Beine um und ermöglicht so ein flüssiges Laufen in alle Richtungen.

[3] `http://www.aibo-europe.com`

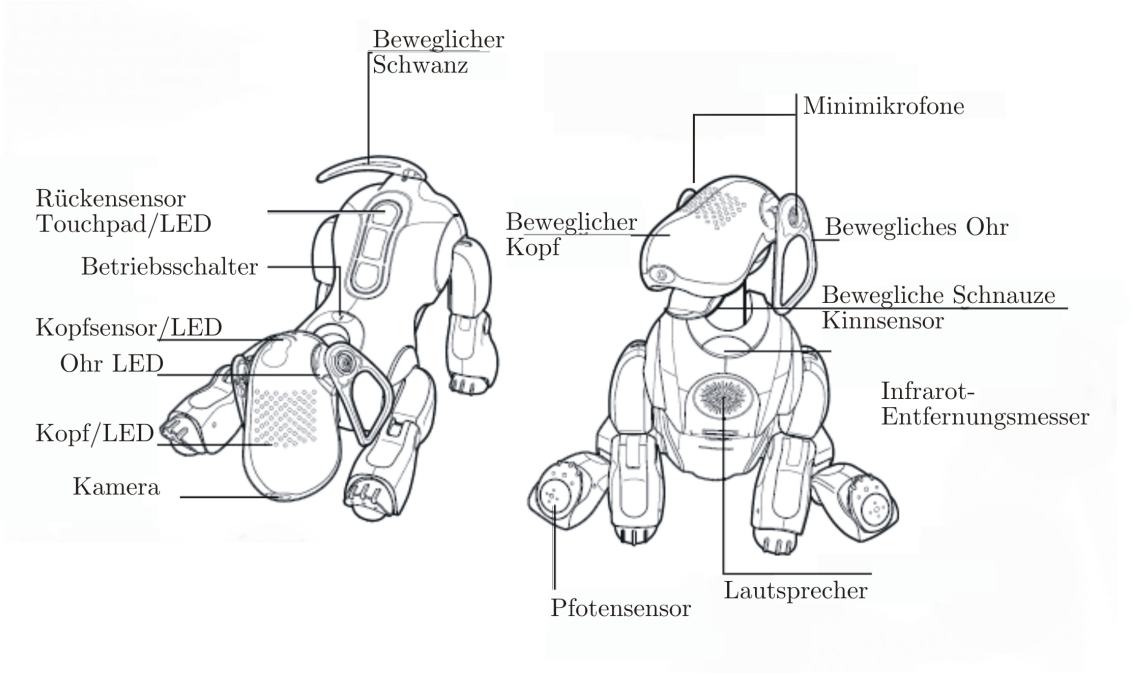


Abbildung 1.2: AIBO ERS7

- Der BallLocator (siehe Kapitel 4) modelliert mit Hilfe bestimmter Perzepte die Ballposition auf dem Feld, der TeamBallLocator (siehe Kapitel 5) nutzt dabei von Mitspielern übertragene Informationen.

- Der SelfLocator (siehe Kapitel 6) modelliert mit Hilfe bestimmter Perzepte die eigene Position auf dem Spielfeld.

- Der ImageProcessor (siehe Kapitel 7) extrahiert aus dem robotereigenen Kamerabildern Umgebungsinformationen (so genannte *Perzepte*) von bekannten Spielfeldteilen.

- Die Entwicklung der Programmiersprache XTC (siehe Kapitel 8) ermöglicht eine vereinfachte Verhaltensprogrammierung.

- Das Erstellen diverser DebugTools und Mechanismen um eine effiziente Entwicklung und Fehlersuche zu ermöglichen.

Das Hauptaugenmerk lag dabei auf der robusteren Modellierung der Umwelt des Roboters. Denn die genauere Kenntnis über Umgebung und Spielsituation bietet natürlich eine bessere Grundlage für Entscheidungen im Spiel. Ebenso grundlegend sind Verbesserungen bei der Bildverarbeitung, auf die ja die modellierenden Programmteile aufbauen. Es hat sich auch gezeigt, dass Veränderungen des Laufens sich durch höhere Geschwindigkeit und/oder größere Laufruhe signifikant auf das Spielgeschehen auswirken können.

Nach Fertigstellung dieser Entwicklungen musste das bisherige Spielverhalten an die neuen Softwareteile angepasst werden. Durch die Verwendung der neuen WalkingEngine änderten sich beispielsweise die experimentell ermittelten Zeitabstände und Ansteuerungsgeschwindigkeiten im Umgang mit dem Ball. Große Änderungen bei der Verhaltenprogrammierung waren auch auf Grund neuer Spielregeln notwendig.

1.3 Regeländerungen[4]

Die Spielregeln werden regelmäßig – überlicherweise jährlich – angepasst, um das Spielgeschehen in Richtung „echtes“ Fußballspiel zu dirigieren.

Das Spielfeld wurde von ursprünglich 3,1 m×4,6 m auf 4 m×6 m vergrößert. Gleichzeitg wurden auch die Tore und der Strafraum größer gemacht. 2004 wurde das Spielfeld noch durch eine 10 cm hohe weiße Bande begrenzt. Diese Bande wurde nun durch Begrenzungslinien auf dem Teppich ersetzt. Als direkte Folge davon können Ball und Roboter das Spielfeld leicht verlassen und die Erkennung der Spielfeldgrenzen gestaltet sich schwieriger. Die vier Landmarken zur Orientierung stehen an neuen Positionen. Ursprünglich befanden sie sich in den vier Spielfeldecken, jetzt stehen sie mittig am linken und rechten Rand jeder Spielfeldhälfte (siehe Abbildung 1.1).

Im den Finalspielen der Tuniere wird bei einem Unentschieden ein „Penalty-Shootout“, eine Art Elfmeterschießen veranstaltet. Dabei versucht ein Feldspieler gegen den gegnerischen Torwart innerhalb einer Minute ein Tor zu erzielen. Noch im Jahr zuvor wurde das Penalty-Shootout ohne den Torwart durchgeführt.

[4]Für eine umfassende Beschreibung der Regeländerungen vergleiche[17] und [19]

2 CeilingCam

Dieses Kapitel beschreibt die CeilingCam-Software, ein Programm, das mit Hilfe von zwei Deckenkameras Informationen darüber liefert, wo sich die Roboter und der Ball auf dem Spielfeld befinden. Dabei wird insbesondere auf folgende Punkte eingegangen:

- Motivation: Wofür die CeilingCam benötigt wird und was sie leisten soll
- Bildverarbeitung: Beschreibung von zwei Lösungen zur Bildanalyse
- Vergleich der Lösungen: Beschreibung einer integrierten Funktionalität zur Messung der Genauigkeit und die resultierenden Ergebnisse
- Verwendung von zwei Kameras: Besonderheiten, Probleme und ihre Lösung
- Übertragung der Daten: Wie die Daten zu den Robotern und anderen Nutzern kommen
- Ausblick: Verbesserungsmöglichkeiten

2.1 Motivation

In der Vergangenheit gab es kaum Möglichkeiten, vom Roboter berechnete Daten zu verifizieren (zum Beispiel Selbstlokalisierung beim Laufen oder die Ballmodellierung bei bewegtem Ball). Ebenso waren die Genauigkeiten dieser Werte unbekannt. Ferner sind selbstoptimierende Algorithmen möglich, wenn die genauen Daten bekannt sind (zum Beispiel die Evolution schnellerer Gangarten).

Ein unter der Decke angebrachtes Kamerasystem, welches das gesamte Spielfeld überblickt, kann zusammen mit einer Bildverarbeitungssoftware die nötigen Informationen liefern. In den Kamerabildern sollen dabei von der Software die Roboter und der Ball erkannt werden. Zur Kontrolle werden erkannte Objekte in den Bildern durch DebugDrawings markiert welche über die graphische Benutzeroberfläche (GUI) der CeilingCam-Software betrachtet werden können (siehe Abbildung 2.1). Durch die statische Anordnung der Kameras bezüglich des Feldes ermöglicht sich die einfache Berechnung der Positionen der erkannten Objekte. Der *WorldState* ist die Gesamtheit der von der CeilingCam ermittelten Daten und setzt sich wie folgt zusammen:

- Die Position, Orientierung und Geschwindigkeit für jeden Roboter

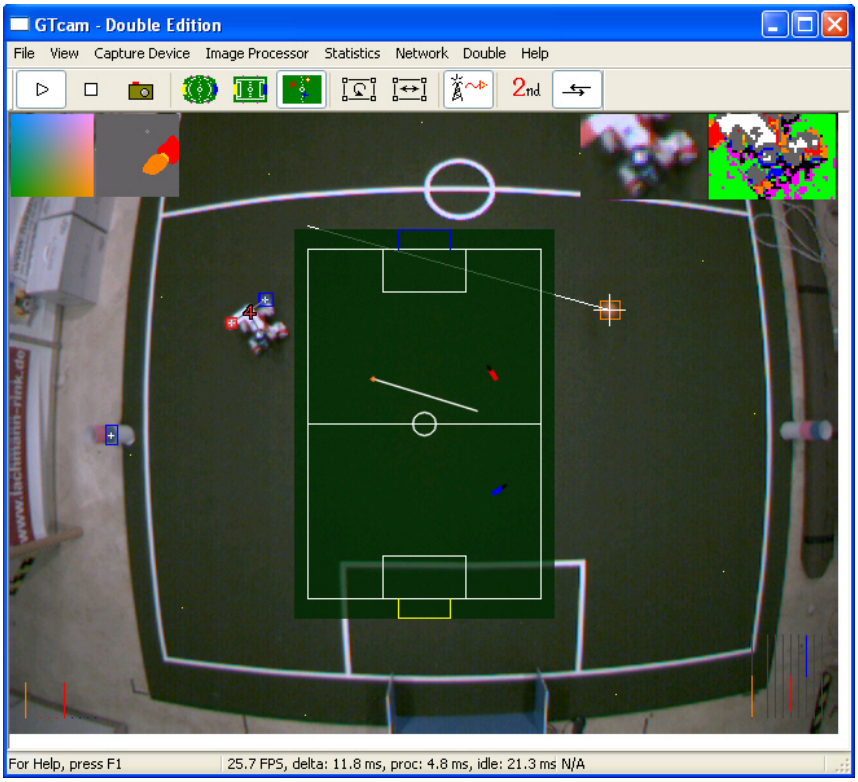


Abbildung 2.1: GUI der CeilingCam-Software: Ansicht des verarbeiteten Bildes mit DebugDrawings und zusätzlichen Informationen: Oben links ein Schnitt durch die verwendete Farbtabelle, unten links die Visualisierung der Qualitäten, mit denen die Objekte erkannt wurden, oben rechts die Bildschirmlupe mit farbklassifizierter Ansicht, unten rechts Visualisierung interner Werte des Algorithmus, der die Informationen der beiden Kameras vereinigt und in der Mitte eine schematische Ansicht des gesamten Spielfeldes mit den erkannten Objekten und ihren Geschwindigkeitsvektoren.

- Die Position und Geschwindigkeit für den Ball
- Ein Zeitstempel, der angibt, von wann die Daten sind
- Zusätzliche Informationen, welche Objekte erkannt worden sind und eine fortlaufende Nummer

Diese Daten werden über das vorhandene Netzwerk verbreitet und können so von den Robotern oder anderen Nutzern verwendet werden.

2.2 Bildverarbeitung

Der ImageProcessor extrahiert die nötigen Merkmale aus dem Kamerabild, um daraus die Positionen und gegebenenfalls die Orientierungen der Objekte zu berechnen.

Zusätzlich berechnet er die Geschwindigkeiten der Objekte aus den Änderungen der Positionen über der Zeit.

2.2.1 Grundlegende Probleme und Lösungsmöglichkeiten

Es gibt mehrere grundsätzliche Probleme, für die im folgenden Lösungsmöglichkeiten diskutiert werden.

2.2.1.1 Kamerabildverzerrung

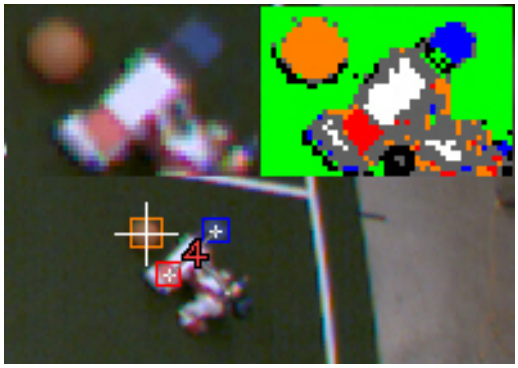
Die verwendeten Kameras sind mit einem extremen Weitwinkelobjektiv ausgestattet, um aus einer Höhe von 2,90 m etwas mehr als das halbe Spielfeld überblicken zu können. Dies führt zu einer starken Verzerrung (zu erkennen in Abbildung 2.1). Nun gibt es die Möglichkeit, zunächst das komplette Bild per Software zu entzerren und darauf die Bildverarbeitung durchzuführen. In diesem Fall vereinfacht sich die Bildverarbeitung, da sowohl der Ball als auch die Roboter im Randbereich des Bildes nicht mehr verkleinert erscheinen. Auch die Umrechnung von Bild- in Weltkoordinaten ist dann nur noch eine proportionale Zuordnung. Dieser Ansatz wurde in einem ersten ImageProcessor verwendet, dem SimpleImageProcessor. Allerdings benötigt das Entzerren des kompletten Bildes auf dem verwendeten Rechner[1] ca. 15 ms, was bei zwei Bildern schon 30 ms wären. Bei 25 Bildern pro Sekunde wären das schon 75% der zur Verfügung stehenden Zeit. Aus diesem Grund arbeiten die ImageProcessoren der CeilingCam seit der Verwendung von zwei Kameras nur noch auf dem Originalbild und entzerren erst die Positionen der im Bild gefundenen Merkmale.

2.2.1.2 Robotererkennung

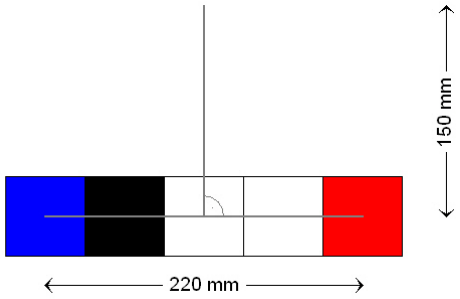
Im Gegensatz zur Erkennung des Balls, der eine eindeutige Farbe und von allen Richtungen eine einheitliche Form hat, ist die direkte Erkennung der Roboter nicht ganz unproblematisch. Aus diesem Grund haben wir die Roboter mit spezielen mehrfarbigen Markern versehen, die sowohl die Berechnung der Position und Orientierung, als auch eine eindeutige Identifizierung der Roboter gewährleisten. Die Aufgabe des ImageProcessors reduziert sich damit bezüglich der Roboter auf das Auffinden der Marker.

Abbildung 2.2 zeigt den Aufbau der Marker. Ein Marker besteht aus fünf aneinandergereihten farbigen Quadraten mit einer Kantenlänge von jeweils ca. 55 mm. Er wird quer zum Roboter auf dessen Rücken festgeklebt, so dass der Kopf keinen Teil des Markers verdecken kann. Das rechteste Quadrat ist immer rot, das Linkeste immer blau. Über diese beiden Farbflächen lässt sich die Position und Orientierung des Roboters bestimmen. Die drei mittleren Quadrate codieren die Spielernummern im Binärsystem. Ein schwarzes Quadrat steht dabei für eine 0, ein weißes Quadrat für eine 1. Das Rechte der drei mittleren Quadrate hat die Wertigkeit 1, das Mittlere

[1] Pentium® 4 CPU, 2,60 GHz, 1 GB RAM



(a) Roboter mit Marker und Ball aus Sicht der CeilingCam (oben rechts: farbklassifizierte Ansicht)



(b) Schematische Darstellung eines Markers. In grau: Konstruktion der Roboterposition

Abbildung 2.2: Aufbau der Marker

die Wertigkeit 2 und das Linke die Wertigkeit 4. Damit lassen sich die Nummern 0 bis 7 codieren, was den Spielernummern Rot 1 bis Blau 4 entspricht.

2.2.1.3 Bildanalyse

Da in der Sony Four-Legged Robot League die Objekte über Farben codiert werden, existieren gute Erfahrungen mit Methoden der Farbklassifizierung. Durch das Verwenden von Markern für die Roboter bietet es sich auch für die CeilingCam an, mittels Farbklassifizierung das Bild in Bereiche gleicher Farbe zu segmentieren. Dies kann zum Beispiel mit Methoden der Bibliothek CMVision (siehe [1]) geschehen. Da das ganze Bild jedesmal vollständig segmentiert wird, werden neu auftauchende Objekte sofort erkannt. Dieser Ansatz wurde ebenfalls im SimpleImageProcessor verwendet. Charakteristisch bei diesem Verfahren ist die nahezu konstante Laufzeit. Sie hängt kaum von der Anzahl der Objekte im Bild ab, ist dafür aber auch konstant hoch. Da sich die Objekte von einem Bild zum nächsten kaum bewegen (die Position eines Roboters mit 50 cm/s ändert sich bei 25 Bilder pro Sekunde und einer Auflösung von ca. 1 cm pro Pixel gerade mal um 2 Pixel von Bild zu Bild), reicht es für bereits gefundene Objekte aus, sie im nächsten Bild in der Umgebung der alten Position zu suchen. Allerdings muss dann zusätzlich nach neuen Objekten im übrigen Bild gesucht werden. Wenn diese Suche nicht zu intensiv betrieben wird, ist es möglich, Laufzeiteinsparungen gegenüber der obigen Methode zu realisieren. Ein weiterer Vorteil ist die direkte Kontrolle über den Algorithmus, der Bereiche gleicher Farbe findet. So hat die Methode von CMVision starke Probleme mit Objekten, die sich über den mit Farbsäumen behafteten Feldlinien befinden. Ein eigener Ansatz kann gezielt versuchen, dieses Problem in den Griff zu bekommen. Aus diesen Gründen verwenden die neueren ImageProcessoren der CeilingCam nicht CMVision, sondern einen eigenen Ansatz, der auf dem Tracking bekannter Objekte und randomisierter Suche von bisher unbekannten Objekten basiert.

2.2.2 ClusterImageProcessor

Der ClusterImageProcessor arbeitet auf dem nicht entzerrten Bild der Kamera. Jedes betrachtete Pixel wird über eine Farbtabelle klassifiziert, ob es sich um Orange (Ball), Rot (rechter Markerpunkt) oder Blau (linker Markerpunkt) handelt.

2.2.2.1 Die Cluster-Methode

Kern dieses Ansatzes ist eine Methode, die an gegebener Stelle überprüft, ob dort ein zusammengehöriger Bereich einer bestimmten Farbe und Größe existiert und gegebenenfalls subpixelgenau dessen Zentrum und eine Qualität zwischen Null und Eins berechnet. Mit dieser Methode wird sowohl die Position des Balls ermittelt, als auch die Positionen der rechten und linken Punkte der Marker. Dabei muss die Methode sowohl mit dem Farbsaum der Feldlinien klarkommen, als auch mit einzelnen Pixel innerhalb des Bereiches, die nicht entsprechend klassifiziert wurden (zum Beispiel Glanzlicht auf dem Ball, Pixelfehler im Sensor der Kamera usw.). Ferner erscheinen die Objekte durch die Kameraverzerrung am Rand des Bildes kleiner als im Zentrum. Durch eine mögliche Neigung der Marker auf dem Roboter wird dieses Problem im Randbereich noch vertärkt, wenn der Roboter den Marker von der Kamera wegkippt. Die Markerpunkte erreichen dadurch am Rand des Bildes zum Teil nicht mehr eine Größe von 3×3 Pixel. Um aber im Zentrum des Bildes die Genauigkeit zu erhöhen und die Wahrscheinlichkeit von fälschlicherweise als Markerpunkte erkannte Bereiche zu veringern, kann die Methode so parametrieiert werden, dass sie im Kern eines Farbbereiches mindestens 3×3 Pixel einer Farbe fordert oder nur 2×2 Pixel.

Funktionsweise Die Cluster-Methode benötigt die Informationen, nach welcher Farbe sie suchen soll, von welchem Pixel aus sie die Suche starten soll, die Mindestkantenlänge des Quadrates im Kern des Farbbereiches (zwei oder drei Pixel), die maximale Ausdehnung des Farbbereiches, sowie die maximale und minimale Anzahl von Pixeln der gesuchten Farbe des Farbbereiches.

Die Methode beginnt mit der Untersuchung, ob das Startpixel zu einem Quadrat der gegebenen Kantenlänge gehört, dessen Pixel alle die gesuchte Farbe haben müssen. Falls nein, wird mit einer Qualität von Null abgebrochen.

Falls ein Quadrat der gegebenen Farbe gefunden wurde, wird der Reihe nach versucht, eine der Seiten weiter nach außen zu schieben. Dabei wird eine Seite weiter geschoben, wenn die Reihe, auf die sie geschoben würde, mindestens soviele Pixel der gesuchten Farbe hat, wie die Kantenlänge des ursprünglichen Quadrats war (also zwei oder drei). Dabei werden aber auch die beiden Pixel berücksichtigt, die auf den verlängerten Diagonalen liegen. Dieses Bereichsvergrößerungsverfahren bricht ab, wenn es keinen Fortschritt mehr gibt, oder die maximale Größe überschritten wurde. Im letzteren Fall wird wieder eine Qualität von Null zurückgegeben.

Falls ein Bereich zulässiger Größe gefunden wurde, wird er auf sein Seitenverhältnis und seine Füllungsdichte überprüft. Seien $sigmaPixel$ die Anzahl der Pixel der gesuchten Farbe im gefundenen Bereich, $minPixel$ und $maxPixel$ die minimal bzw.

maximal zulässige Anzahl an Pixel der gesuchten Farbe, *height* und *width* die Höhe und Breite des gefundenen Bereiches, dann berechnet sich das Seitenverhältnis *ratio*, die Füllungsdichte $filling$ und die Qualität q durch:

$$ratio = \min\left\{\frac{width}{height},\ \frac{height}{width}\right\} \tag{2.1}$$

$$filling = \frac{sigmaPixel}{width \cdot height} \tag{2.2}$$

$$q = \max\left\{0;\ ratio \cdot \left(1 - \left(\frac{2 \cdot sigmaPixel - minPixel - maxPixel}{maxPixel - minPixel}\right)^2\right)\right\} \tag{2.3}$$

Die maximale Qualität von 1,0 wird dabei angenommen, wenn der gefundene Bereich quadratisch ist (das heißt $ratio = 1{,}0$) und $sigmaPixel$ gleich dem arithmethischen Mittel aus minPixel und maxPixel ist. Mit zunehmender Abweichung von diesen Idealwerten fällt die Qualität immer stärker bis sie für $sigmaPixel \leq minPixel$ oder $sigmaPixel \geq maxPixel$ gleich 0,0 wird. Eine Qualität von 0,0 bedeutet also, dass die Anzahl der Pixel der gesuchten Farbe im gefundenen Bereich nicht innerhalb des geforderten Intervalls von $minPixel$ bis $maxPixel$ liegt, der Bereich wird dann verworfen. Ein Verhältniss von kleiner als 0,25 bedeutet, dass der gefundene Farbbereich in einer Richtung vier mal so lang ist wie in der anderen. Es ist dann sehr unwahrscheinlich, dass es sich um einen Markerpunkt handelt, der eigentlich quadratisch sein sollte. Ebenso ist es unwahrscheinlich, dass es sich noch um einen Markerpunkt handelt, wenn weniger als 40% der Pixel innerhalb des gefundenen Bereiches die gesuchte Farbe haben. Daher wird der gefundene Farbbereich auch verworfen, wenn $ratio < 0{,}25$ oder $filling < 0{,}4$.

Zuletzt wird das Zentrum des Bereiches bestimmt. Für einen groben Ansatz könnte einfach für x und y der Duchschnitt der jeweiligen Begrenzungen gebildet werden und es ergäbe sich damit eine Auflösung von einem halben Pixel. Um das Quantisierungsrauschen weiter zu verringern, wird die Anzahl der Pixel mit der gesuchten Farbe in den Reihen in Betracht gezogen, die den gefundenen Bereich umgeben. Dadurch wird eine Auflösung von 1/6 Pixel bzw. 1/4 Pixel erreicht, je nachdem ob eine minimale Kantenlänge von drei oder zwei Pixeln gefordert wurde.

Bewertung Die Cluster-Methode des ClusterImageProcessors hat sich in monatelangem Einsatz als recht robust erwiesen. Durch die Forderung von zwei bzw. drei Pixel der gesuchten Farbe auf einer Line zur Vergrößerung des Bereiches, kommt dieser Ansatz mit Farbsäumen klar, wenn sie nicht breiter als ein bzw. zwei Pixel sind. Nachteil ist die Notwendigkeit einer guten Farbtabelle und das nicht zu vernachlässigende Quantisierungsrauschen. Eine Auflösung von 1/6 Pixel bedeutet bei 25 Bilder pro Sekunde und einer Auflösung von ca. 1 cm pro Pixel eine Quantisierung in der Geschwindigkeit von ca. 4 cm/s. Gerade im Randbereich, wo nur ein minimales Startquadrat von 2×2 Pixel als Startfläche für Markerpunkte gefordert wird,

verstärkt sich das Rauschen und es kann häufig zur fälschlicherweisen Erkennung von Markerpunkten kommen.

2.2.2.2 Ballverfolgung

Wurde in einem der letzten Bilder bereits ein Ball gefunden, ist es, wie bereits in Kapitel 2.2.1.3 erwähnt, nicht nötig, im ganzen Bild nach einem Ball zu suchen. Stattdessen wird von der letzten bekannten Position im Bild ausgehend, spiralförmig nach orangenen Pixeln gesucht. Wird eines gefunden, so wird an dieser Stelle die Cluster-Methode (siehe Kapitel 2.2.2.1) aufgerufen. Findet sie dort einen Ball, wird er akzeptiert, ansonsten wird auf der Spirale weiter nach dem nächsten orangenen Pixel gesucht. Kann auf diese Weise kein Ball gefunden werden, wird mit der im nächsten Abschnitt beschriebenen Methode nach dem Ball gesucht.

2.2.2.3 Ballfindung

Wenn noch keine Ballposition bekannt ist oder der ImageProcessor den Ball mit obiger Methode nicht mehr finden konnte, wird im gesamten Bild nach dem Ball gesucht. Dazu wird nacheinander zufällig ein Pixel aus dem Bild ausgewählt und, falls es orange ist, an dieser Stelle die Cluster-Methode (siehe Kapitel 2.2.2.1) aufgerufen. Dies wird so lange wiederholt, bis ein Ball mit einer Qualität von mehr als 0,2 gefunden wurde oder eine gewisse Anzahl an Versuchen überschritten ist.

2.2.2.4 Markerverfolgung

Ebenso wie beim Ball, werden bereits bekannte Marker von ihrer letzten bekannten Position aus gesucht. Das Verfahren von Kapitel 2.2.2.2 wird dafür sowohl für den rechten roten Markerpunkt, als auch für den linken blauen Markerpunkt ausgeführt.

2.2.2.5 Markerfindung

Nach bisher unbekannten Markern wird in jedem Bild gesucht, auch dann, wenn schon alle acht Marker verfolgt werden. Zunächst wird zufällig ein kleinerer quadratischer Unterbereich aus dem Bild ausgewählt, in dem nach Markern gesucht werden soll. Dann wird wie beim Ball (siehe Kapitel 2.2.2.3) nach jeweils bis zu zwei roten und blauen Markerpunkten gesucht. Die bis zu vier Kombinationen aus rotem und blauen Punkt werden auf ihre Möglichkeit als Marker hin überprüft. Dieses Verfahren wird für mehrere zufällige Bereiche wiederholt.

2.2.2.6 Markeridentifizierung

Wenn ein roter und ein blauer Bereich gefunden wurde, die zusammen einen Marker bilden könnten, werden ihre Koordinaten von Bild- in Weltkoordinaten umgerechnet und ihr Abstand bestimmt. Bewegt er sich in einem zulässigen Bereich, werden

die Weltkoordinaten der drei mittleren Markerquadrate bestimmt, indem die Strecke zwischen rotem und blauen Markerpunkt in vier gleich große Abschnitte unterteilt wird. Diese Weltkoordinaten werden in Bildkoordinaten zurück gerechnet und an diesen Stellen im Bild die Helligkeit ausgewertet. Über zwei Schwellwerte wird bestimmt, ob es sich um einen schwarzen oder weißen Punkt handelt. Mit diesen Informationen lässt sich die Spielernummer ermitteln (vergleiche Kapitel 2.2.1.2). Liegt der Helligkeitswert eines Punktes zwischen den Schwellwerten, wird davon ausgegangen, dass es sich nicht um einen Marker handelt.

2.2.3 EdgeImageProcessor

Ziel der Entwicklung des EdgeImageProcessors war die Reduzierung des Quantisierungsrauschen und eine geringere Abhängigkeit von der Farbtabelle. Er arbeitet wie der ClusterImageProcessor auf dem nicht entzerrten Bild der Kamera, nutzt aber nicht nur die Informationen des farbklassifizierten Bildes sondern verwendet zusätzlich die ursprünglichen Farbwerte zum farbtabellenunabhängigen Finden der Markerpunktkanten.

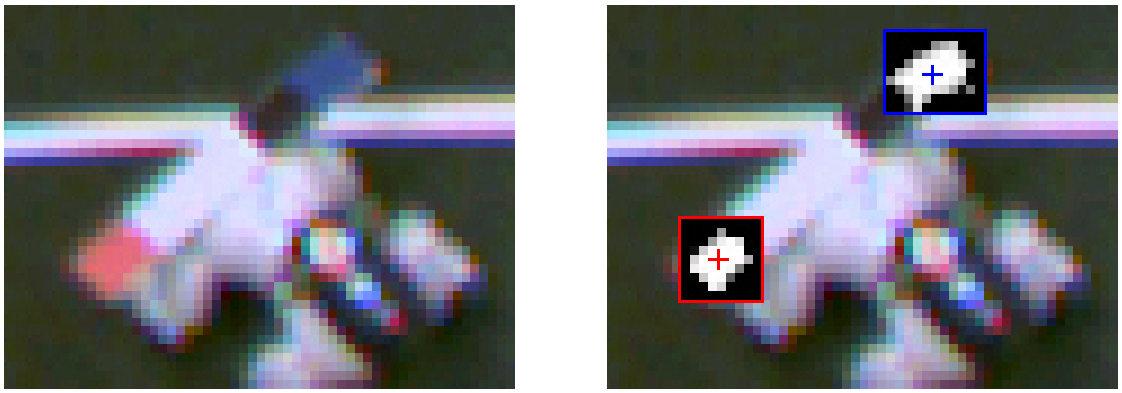
2.2.3.1 Die kantenbasierte Markerpunktfindung

Die kantenbasierte Markerpunktfindung soll wie die Cluster-Methode (siehe Kapitel 2.2.2.1) das Zentrum eines zusammengehörigen Farbbereiches und seine Qualität berechnen.

Funktionsweise Von einem gegebenen Pixel aus wird in einem ersten Schritt in alle vier Richtungen die nächste Kante gesucht. Dazu wird jeweils der Weg des geringsten Widerstandes gewählt, also nicht einfach geradeaus, sondern gegebenenfalls schräg nach links oder rechts, wenn in dieser Richtung der Farbunterschied geringer ist, solange bis der Farbunterschied einen gegebenen Schwellwert übersteigt.

Auf diese Weise lässt sich gut ein umhüllendes Rechteck für einen Markerpunkt finden, zur Zentrumsbestimmung ist diese Methode aber noch zu ungenau. Daher wird in einem zweiten Schritt die durchschnittliche Farbe auf den vier Wegen vom Startpunkt zu den vier Kantenpunkten genutzt, um das Zentrum des Markerpunktes als gewichteten Massenschwerpunkt über alle Pixel des umhüllenden Rechteckes zu berechnen. Sei $\vec{c}$ die im ersten Schritt ermittelte durchschnittliche Farbe im RGB-Raum und A die Menge der Pixel des umhüllenden Rechteckes, dann berechnet sich für jedes Pixel $\vec{p} \in A$ das Gewicht g durch Gleichung (2.4), wobei $\vec{f}$ die Farbfunktion des Bildes beschreibt und $MaxDist$ der Schwellwert für die Distanz im Farbraum ist, ab der ein Pixel nicht mehr berücksichtigt wird.

$$g\left(\vec{p}\right) = \max\left\{0;\ 1 - \left(\frac{\left\|\vec{f}\left(\vec{p}\right) - \vec{c}\right\|}{MaxDist}\right)^2\right\} \tag{2.4}$$



(a) Originalansicht (b) Mit Gewichtung und Zentrum

Abbildung 2.3: Visualisierung der Pixelgewichtung durch Graustufen

Das maximale Gewicht von 1,0 erhält ein Pixel dabei, wenn es exakt die zuvor ermittelte durchschnittliche Farbe hat. Mit zunehmender Abweichung von dieser Farbe wird das Pixel immer schwächer gewichtet, bis es ab einer Abweichung von $MaxDist$ nicht mehr ins Gewicht fällt. Abbildung 2.3 veranschaulicht die Gewichtung: Pixel mit einem Gewicht von 1,0 werden weiß dargestellt, Pixel mit einem Gewicht von 0,0 schwarz. Dazwischenliegende Gewichte werden durch Graustufen visualisiert.

Das Zentrum $\vec{z}$ berechnet sich dann durch:

$$\vec{z} = \frac{1}{\Delta} \sum_{\vec{p} \in A} \vec{p} \cdot \vec{g}\,(\vec{p}) \text{ mit } \Delta = \sum_{\vec{p} \in A} \vec{g}\,(\vec{p}) \tag{2.5}$$

Die Qualität berechnet sich ähnlich wie bei der Cluster-Methode durch:

$$q = \max \left\{ 0;\ 1 - \left(\frac{2 \cdot \Delta - minPixel - maxPixel}{maxPixel - minPixel} \right)^2 \right\} \tag{2.6}$$

Die maximale Qualität von 1,0 wird dabei angenommen, wenn Δ gleich dem arithmetischen Mittel aus $minPixel$ und $maxPixel$ ist. Mit zunehmender Abweichung von diesem Wert fällt die Qualität immer stärker, bis sie für $\Delta \leq minPixel$ oder $\Delta \geq maxPixel$ gleich 0,0 wird.

Bewertung Die kantenbasierte Markerpunktfindung bietet gegenüber der Cluster-Methode einige Vorteile. Sie hat konzeptionell bedingt (fast) kein Quantisierungsrauschen, sondern liefert fließende Positionswerte und damit gleichmäßigere Geschwindigkeitswerte. Durch die farbtabellenunabhängige Arbeitsweise wirkt sich die Qualität der Farbtabelle nicht auf die Genauigkeit aus. Ihre Laufzeit ist auf großen einfarbigen Flächen linear zum erwarteten Markerradius und nicht quadratisch, wie bei der Cluster-Methode. Das ist vorteilhaft, wenn etwa eine Person mit roter oder blauer Kleidung das Feld betritt.

Probleme bekommt die kantenbasierte Markerpunktfindung, wenn der Kontrast zwischen Markerpunkt und Umgebung zu klein wird, etwa am Spielfeldrand bei ungenügender Beleuchtung. Unter gewöhnlichen Spielbedingungen wurde dieses Problem aber nicht beobachtet. Allerdings neigt die Methode in ihrer jetzigen Form dazu, bei starken Kontrasten falsche Markerpunkte in Kanten zu finden.

2.2.3.2 Ballfindung und -verfolgung

Leider ließ sich die kantenbasierte Markerpunktfindung nicht so einfach auf den Ball übertragen, da der Ball durch seine räumliche Struktur keine so gleichmäßige Farbe aufweist, wie die Markerpunkte. Aus diesem Grund wurde für den Ball das selbe Verfahren beibehalten, wie beim ClusterImageProcessor.

2.2.3.3 Markerverfolgung

Für die Verfolgung der Markerpunkte wird die im ersten Schritt der kantenbasierten Markerpunktfindung ermittelte Farbe für jeden Markerpunkt gespeichert. Im nächsten Bild wird dann in der Umgebung der alten Position nach einem 3×3 Bereich gesucht, dessen durchschnittliche Farbe den geringsten euklidischen Abstand im RGB-Farbraum zur gespeicherten Farbe hat. Vom Mittelpunkt des gefundenen Bereiches aus wird dann die kantenbasierte Markerpunktfindung gestartet.

2.2.3.4 Markerfindung

Für das initiale Finden von Markerpunkten wird weiterhin die Farbtabelle verwendet. Wie beim ClusterImageProcessor wird zunächst zufällig ein quadratischer Unterbereich gewählt, in dem dann nach roten und blauen Pixel gesucht wird, von denen aus die kantenbasierte Markerpunktfindung gestartet wird.

2.2.3.5 Markeridentifizierung

Die Identifizierung der Spielernummer funktioniert wie beim ClusterImageProcessor. Eine Erweiterung liegt aber in der dynamischen Anpassung der Schwellwerte für schwarze und weiße Markerpunkte, indem das Wissen um die Helligkeit des roten und des blauen Markerpunktes genutzt wird.

2.3 Vergleich der Lösungen

In diesem Abschnitt wird eine in die CeilingCam-Software integrierte Funktionalität vorgestellt, um die Güte der ermittelten Daten bewerten zu können. Anschließend erfolgt eine Bewertung und ein Vergleich der beiden ImageProcessoren.

2.3.1 Das Analyse-Modul der CeilingCam-Software

Um das Rauschverhalten der ImageProcessoren der CeilingCam-Software zu untersuchen und zu vergleichen, war es zuvor nötig, die von der CeilingCam gesendeten Daten in eine Textdatei zu speichern und mit einem externen Tool auzuwerten. Das Analyse-Modul soll diese Funktionalität während des Betriebes der CeilingCam für ein frei zu wählendes Objekt (Ball, Roboter Rot 1, usw.) ermöglichen.

2.3.1.1 Ansätze für die Analyse

Die Größe des Rauschens läßt sich gut mit der Standardabweichung beschreiben. Sei n die Anzahl der Messungen und $\vec{p_i}$ die i-te gemessene Position, so berechnet sich die Standardabweichung $\vec{\sigma}$ durch:

$$\vec{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(\vec{p_i} - \vec{\bar{p}}\right)^2} \quad \text{mit} \quad \vec{\bar{p}} = \frac{1}{n}\sum_{i=1}^{n}\vec{p_i}. \tag{2.7}$$

Für diesen Ansatz wäre es allerding nötig, alle Messwerte zu speichern, da der Mittelwert $\vec{\bar{p}}$ nicht im Voraus bekannt ist. Durch Umformen mittels binomischer Formel und Einsetzten ergibt sich:

$$\vec{\sigma} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\vec{p_i}^2 - \frac{1}{n^2}\left(\sum_{i=1}^{n}\vec{p_i}\right)^2} \tag{2.8}$$

Bei diesem akkumulativen Ansatz ist es lediglich nötig, die Summe der Quadrate und die Summe der Messwerte mit zu führen und mit jeder Messung zu aktualisieren. Nachteilig bei dieser Methode sind nummerische Ungenauigkeiten, die durch die Quadrate großer Werte entstehen können.

2.3.1.2 Beschreibung der Analyse

Sobald das gewählte Objekt zweimal erkannt wurde, wird mit dem arithmetischen Mittel $\vec{\hat{p}}$ dieser beiden Positionen als Startposition die Analyse gestartet. Jede weitere Messung, bei der das Objekt erkannt wird und seine Position nach der Maximumsnorm um weniger als 100 mm von der Startposition abweicht, wird als korrekt erkannt gezählt und führt zur Aktualisierung der akkumulierten Quadrate und Positionen:

$$\overrightarrow{sdq} = \sum_{i=1}^{n}\left(\vec{p_i} - \vec{\hat{p}}\right)^2 \text{ und } \overrightarrow{sd} = \sum_{i=1}^{n}\left(\vec{p_i} - \vec{\hat{p}}\right). \tag{2.9}$$

Nach 1000 Messungen endet die Messung automatisch und es werden folgende Daten ausgegeben:

- die Startposition und gegebenenfalls die Startrotation

- die Anzahl der Messungen
- die Anzahl der Messungen ohne Objekterkennung
- die Anzahl der Messungen mit Objekterkennung außerhalb des Startbereiches
- die Anzahl der Messungen mit Objekterkennung innerhalb des Startbereiches n
- die mittlere Position innerhalb des Startbereiches $\vec{p} + \overrightarrow{sd}/n$
- die maximalen und minimalen innerhalb des Startbereiches gemessenen x- und y-Werte und ihre Differenzen
- die Standardabweichung innerhalb des Startbereiches $\vec{\sigma}$ nach Gleichung (2.10).

$$\vec{\sigma} = \sqrt{\frac{1}{n}\overrightarrow{sdq} - \frac{1}{n^2}\overrightarrow{sd^2}}. \tag{2.10}$$

2.3.1.3 Bewertung des Analyse-Tools

Das Analyse-Modul eignet sich gut zur Qualitätsüberprüfung der CeilingCam während ihres Einsatzes in Hinblick auf die Erkennungsrate von bewegten und ruhenden Objekten und auf das Rauschverhalten der Positionsdaten bei ruhenden Objekten. Durch die akkumulative Berechnung von Mittelwert und Standardabweichung ergibt sich ein minimaler konstanter Speicherbedarf und minimale konstante Laufzeit zur Endauswertung unabhängig von der Dauer der Analysephase. Die nummerischen Probleme, die dieses Verfahren gewöhnlich mit sich bringt, werden durch die auf 1000 Messungen beschränkte Analysephase, den beschränkten Bereich für als korrekt erkannte Messungen und die Translation der Positionen in Richtung Nullpunkt minimiert.

2.3.2 Genauigkeitsanalyse der ImageProcessoren

Um die Genauigkeit und das Rauschen der ImageProcessoren zu untersuchen, wurden mit jedem ImageProcessor sechs Messreihen von jeweils 1000 Messungen eines ruhenden Roboters durchgeführt. Untersucht wurden folgende Positionen:

- Roboter am Mittelkreis, Analyse der Daten beider Kameras
- Roboter am Schnittpunkt von Mittellinie und Außenlinie, Analyse der Daten beider Kameras
- Roboter senkrecht unter einer Kamera mit Ausrichtung in Längsrichtung des Feldes
- Roboter senkrecht unter einer Kamera mit diagonaler Ausrichtung

Die durchschnittliche Standardabweichung der Position betrug dabei beim ClusterImageProcessor 0,944 mm, beim EdgeImageProcessor mit 0,468 mm ca. die Hälfte. Dabei traten beim ClusterImageProcessor Ausreißer von ca. 1 cm und beim EdgeImageProcessor Ausreißer von ca. 0,5 cm auf.

An den beiden Positionen, an denen der Roboter von beiden Kameras erkannt wurde, betrug die größte Abweichung zwischen beiden Kameras weniger als 3 cm.

Für den Ball gelten für beide ImageProcessoren ähnliche Genauigkeiten wie für die Markererkennung beim ClusterImageProcessor.

2.3.3 Direkter Vergleich der ImageProcessoren

Um die ImageProcessoren direkt vergleichen zu können, ist es möglich, beide auf dem gleichen Bild laufen zu lassen. In diesem Test wurde die Position und Geschwindigkeit eines ruhenden Markers aufgezeichnet. In Abbildung 2.4 erkennt man deutlich die gröbere Quantisierung des ClusterImageProcessor und das geringere Rauschen des EdgeImageProcessor.

2.4 Verwendung von zwei Kameras

Auf Grund des großen Spielfeldes im Vergleich zur Deckenhöhe verwenden wir zwei Kameras, um das gesamte Spielfeld von oben überblicken zu können. Dies zieht einige Besonderheiten nach sich, um aus den zwei Kamerabildern einen gemeinsamen *WorldState* zu generieren.

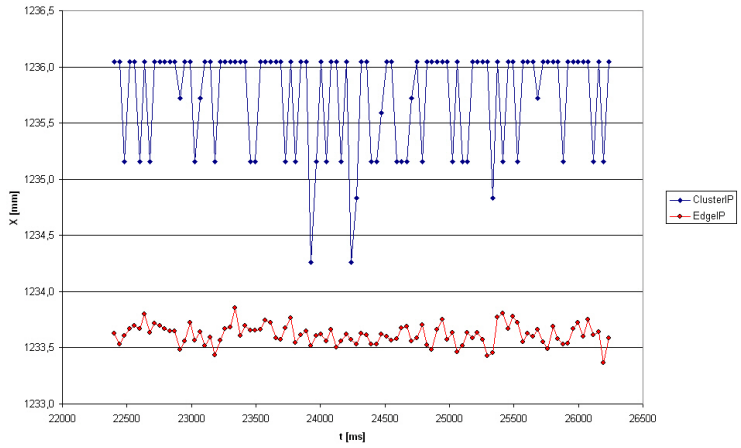
2.4.1 ImageCapturing

Zunächst werden die von den beiden Kameras aufgenommenen Bilder eingelesen, wobei darauf geachtet wird, dass diese Bilder einen möglichst kleinen Zeitversatz haben. Da die Kameras nicht mit der exakt gleichen Framerate arbeiten[2], sorgt ein Algorithmus dafür, dass gegebenenfalls ein Bild verworfen wird und sich so die Reihenfolge, in der die Bilder eingelesen werden, umdreht.

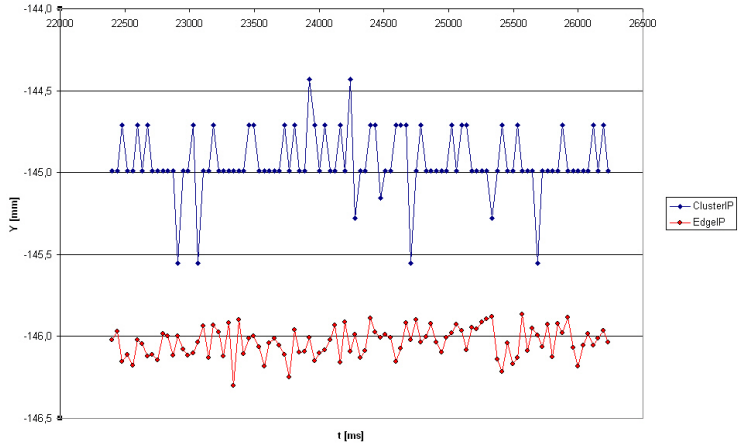
2.4.2 Bildverarbeitung

Sobald beide Bilder eingelesen sind, wird mit der Bildverarbeitung begonnen. Wie schon in Kapitel 2.2.1.1 beschrieben, sind die Bilder mit einer starken Verzerrung behaftet. Ein einfacher Ansatz wäre es, beide Kamerabilder zu entzerren, zu einem Bild zu vereinigen und darauf die Bildanalyse laufen zu lassen. Da die Roboter und der Ball aber eine unterschiedliche Höhe aufweisen und die Kameras ein Objekt aus unterschiedlichen Richtungen betrachten, ist es nicht möglich, die Bilder nahtlos zusammenzusetzen. Neben weiteren Nachteilen (hohe Laufzeit der Entzerrung und

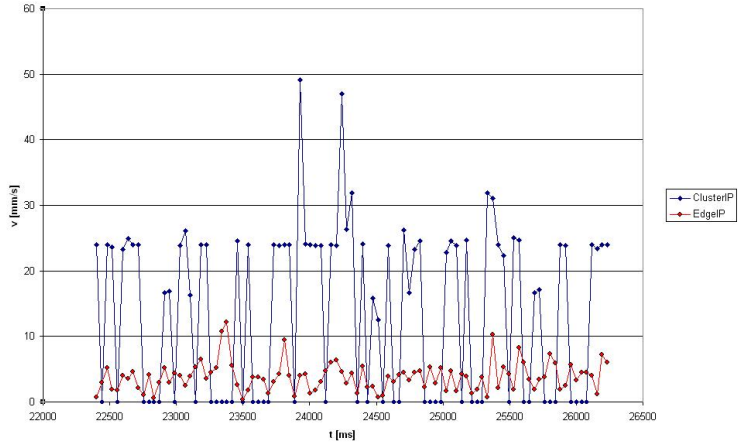
[2] der Unterschied beträgt ca. ein Bild in zwei Stunden



(a) x-Position



(b) y-Position



(c) Geschwindigkeit

Abbildung 2.4: Vergleich der ImageProcessoren: Position und Geschwindigkeit eines ruhenden Markers über der Zeit

unterschiedliche Aufnahmezeitpunkte der Bilder), war dies der Hauptgrund, diesen Ansatz zu verwerfen.

Stattdessen haben wir uns entschlossen, die Bilder getrennt zu analysieren. Zunächst wird das Bild verarbeitet, in dem der Ball erwartet wird. Dadurch ist es möglich, dem anderen ImageProcessor die Position des Balles mitzuteilen, woraufhin dieser gezielt dort nach dem Ball suchen kann. Dies ermöglicht eine nahtlose Erkennung des Balles auch dann, wenn er sich mit hoher Geschwindigkeit von einer auf die andere Spielfeldhälfte bewegt und somit von dem Erfassungsbereich der einen Kamera in den Erfassungsbereich der anderen übergeht. Da die Roboter sich nicht so schnell bewegen können wie der Ball, ist für sie keine Kommunikation zwischen den ImageProcessoren nötig, wenn sich die Bilder ausreichend überlappen.

In der graphischen Benutzeroberfläche (GUI) der CeilingCam-Software wird nur das Bild der vom Benutzer ausgewählten Kamera angezeigt. Der zugehörige ImageProcessor bekommt eine Kopie des Kamerabildes, analysiert es und markiert die gefundenen Objekte durch *DebugDrawings*. Dieses Bild wird anschließend in der GUI dargestellt. Der andere ImageProcessor arbeitet direkt auf dem Speicherbereich, in den der Kameratreiber das Bild abgelegt hat und wird angewiesen, das Bild nicht zu verändern.

2.4.3 Die Vereinigung der Daten

Nach der Bildverarbeitung liegen zwei *WorldStates* vor, von jeder Kamera einer. Diese müssen nun im Normalfall mit einem Algorithmus zu einem gemeinsamen *WorldState* vereinigt werden. Es gibt aber auch die Möglichkeit, beide Spielfeldhälften getrennt zu verwenden, dann entfällt dieser Schritt.

2.4.3.1 Funktionsweise

Um aus den *WorldStates* der beiden ImageProcessoren einen gemeinsamen *WorldState* zu generieren, muss zunächst ein gemeinsamer Zeitstempel gefunden werden, da die Bilder in aller Regel nicht zur gleichen Zeit aufgenommen wurden. Bei 25 Bildern pro Sekunde beträgt der maximale Zeitunterschied nicht wesentlich mehr als 20 ms, da anderenfalls die Reihenfolge gewechselt würde und sich eine kleinere Zeitdifferenz ergeben würde. In 20 ms bewegt sich der Ball bei 2 m/s aber bereits 4 cm. Wenn man als gemeinsamen Zeitstempel den Durchschnitt der Aufnahmezeitpunkte wählen würde, aber die Positionen unverändert übernähme, entstünde so ein Fehler von 2 cm.

Dieser Fehler lässt sich vermeiden, wenn beide *WorldStates* auf eine Zeit verrechnet werden. Da nicht bekannt ist, ob sich ein Objekt in Zukunft so weiterbewegt wie bisher (der Ball kann gegen einen Roboter prallen, oder ein Roboter seine Richtung ändern), ist es sicherer, den neueren *WorldState* auf die Zeit des älteren zurück zu interpolieren. Dann kann als gemeinsamer Zeitstempel der ältere der beiden genommen werden. Als Nebeneffekt wird das Rauschen der Daten aus dem neueren

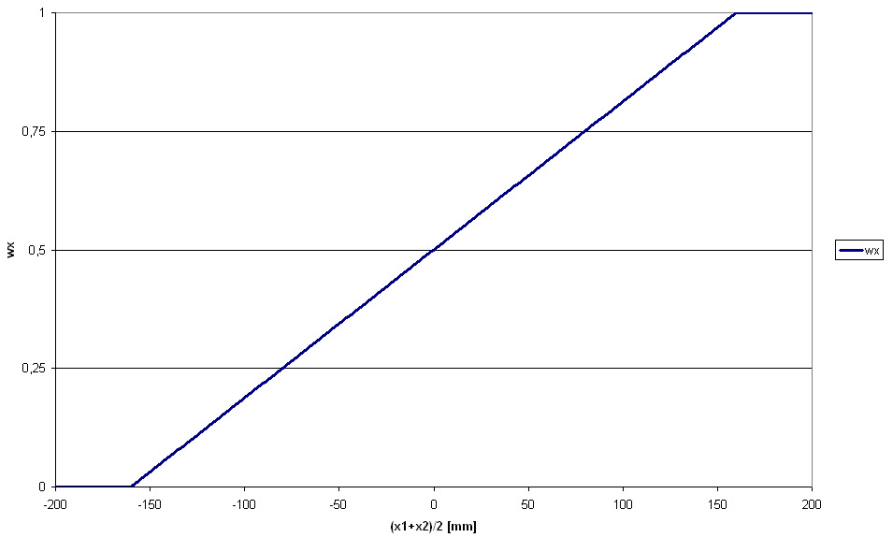


Abbildung 2.5: Verlauf von w_x in Abhängigkeit der mittleren x-Position eines Objektes bei einem Überlappungsbereich der Breite von 320 mm

WorldState gemildert, da die interpolierten Daten Konvexkombinationen der Originaldaten sind.

Nach dem zurück Interpolieren des neueren *WorldStates* basieren die Daten beider *WorldStates* auf dem selben Zeitpunkt und können nun vereinigt werden. Im einfachsten Fall wird ein Objekt von genau einer Kamera gesehen. Dann können dessen Daten direkt in den gemischten *WorldState* übernommen werden. Wenn ein Objekt im Überlappungsbereich der Kameras von beiden Kameras gesehen wird, wird ein gewichteter Durchschnitt berechnet. Dabei berechnet sich das verwendete Gewicht aus dem zuletzt verwendeten Gewicht und dem Sollgewicht, welches aus den Qualitäten, mit denen das Objekt erkannt wurde und seinen Positionen bestimmt wird. Seien q_1 und q_2 die Qualitäten, mit denen die Kameras das Objekt erkannt haben und x_1 und x_2 entsprechend die x-Koordinaten des Objektes. Ferner sei die Kamera mit der Nummer 1 über der Spielfeldhälfte mit den positiven x-Koordinaten montiert. Dann berechnet sich das Sollgewicht s_1 für die Daten der ersten Kamera durch Gleichung (2.11), wobei *width* die Breite des Überlappungsbereiches angibt.

$$s_1 = \frac{w_x \cdot q_1}{w_x \cdot q_1 + (1 - w_x) \cdot q_2} \text{ mit } w_x = \max\left\{0;\ \min\left\{1;\ \frac{x_1 + x_2}{2 \cdot width} + \frac{1}{2}\right\}\right\} \quad (2.11)$$

Dabei ist w_x die ortsabhängige Gewichtskomponente. Sie hängt von dem arithmethischen Durchschnitt der beiden x-Koordinaten ab und ist $1/2$, wenn sich das Objekt auf der Mittellinie befindet und fällt linear bis auf 0,0 ab, je weiter sich das Objekt im Bereich der zweiten Kamera befindet. Befindet sich das Objekt weiter im Bereich der ersten Kamera, steigt diese Gewichtskomponente linear bis auf 1,0 (siehe Abbildung 2.5). Die ortsabhängige Gewichtskomponente für die Daten der zweiten Kamera beträgt aufgrund der Symmetrie $1 - w_x$. Die Sollgewichte ergeben sich dann

aus dem Produkt ihrer ortsabhängigen Gewichtskomponente und der zugehörigen Qualität und werden so normiert, dass ihre Summe 1 beträgt.

Das tatsächliche Gewicht w_1 wird dem Sollgewicht langsam angenähert, um Sprünge zu vermeiden: Falls der Abstand zum Sollgewicht größer als 0,1 ist, wird w_1 dem Sollgewicht um 0,1 angenähert, ansonsten wird w_1 auf den Durchschnitt aus altem Gewicht und neuem Sollgewicht gesetzt. Eine zu schnelle Annäherung könnte zu Rauschen und Sprüngen in den resultierenden Daten aufgrund von schwankenden Qualitäten und unterschiedlichen Positionen in den zu vereinigenden *WorldStates* führen. Eine zu langsame Annäherung wäre eventuell mit der Überblendung nicht rechtzeitig fertig, wenn ein Objekt die Spielfeldhälfte wechselt, was wiederum zu einem Sprung in den Daten führen könnte. Das Gewicht für die Daten der zweiten Kamera w_2 berechnet sich schließlich durch $1 - w_1$.

2.4.3.2 Test des Vereinigungs-Algorithmus

In diesem Test werden die Geschwindigkeiten der beiden ImageProcessoren für den Ball in x-Richtung und die resultierende Größe nach dem Vereinigen sowie die verwendete Gewichtung aufgezeichnet und in Abbildung 2.6 visualisiert. Der Ball wird mit einer Anfangsgeschwindigkeit von über 1 m/s in Richtung steigender x-Koordinaten längs über das Feld geschossen. Zunächst befindet er sich im Bereich der zweiten Kamera, durchläuft dann den Überlappungsbereich und wird schließlich nur noch von der ersten Kamera gesehen. Die zweite Kamera hinkt der ersten Kamera zum Zeitpunkt des Tests um ca. 13 ms hinterher, das heißt, die von ihr gelieferten Daten sind aktueller als die der Ersten. Der *WorldState* des zweiten ImageProcessors wird also vor dem Vereinigen der Daten um ca. 13 ms zurück interpoliert. Deutlich erkennt man in der ersten Phase, dass dadurch das Rauschen der Geschwindigkeit gemildert wird (türkise Messreihe: Originalgeschwindigkeit des zweiten ImageProcessors, orangene Messreihe: resultierende Geschwindigkeit des Vereinigungs-Algorithmus).

Nach einer Sekunde tritt der Ball in den Sichtbarkeitsbereich der ersten Kamera ein und wird für ca. 700 ms von beiden ImageProcessoren erkannt. In dieser Zeit gewinnen die Daten des ersten ImageProcessors fließend mehr an Gewicht, bis sie schließlich alleine die vereinigte Größe bestimmen.

2.4.4 Zusätzliche DebugDrawings

Da aus Effizienzgründen nur ein Bild in der graphischen Benutzeroberfläche dargestellt wird, ist es nicht möglich, gleichzeitig die *DebugDrawings* beider ImageProcessoren zu beobachten. Aus diesem Grund können bei Bedarf die aktuellen Gewichte des Vereinigungs-Algorithmus und eine schematische Ansicht des gesamten Spielfeldes mit den gefundenen Objekten in das Bild eingeblendet werden (siehe Abbildung 2.1 rechts unten bzw. mitte).

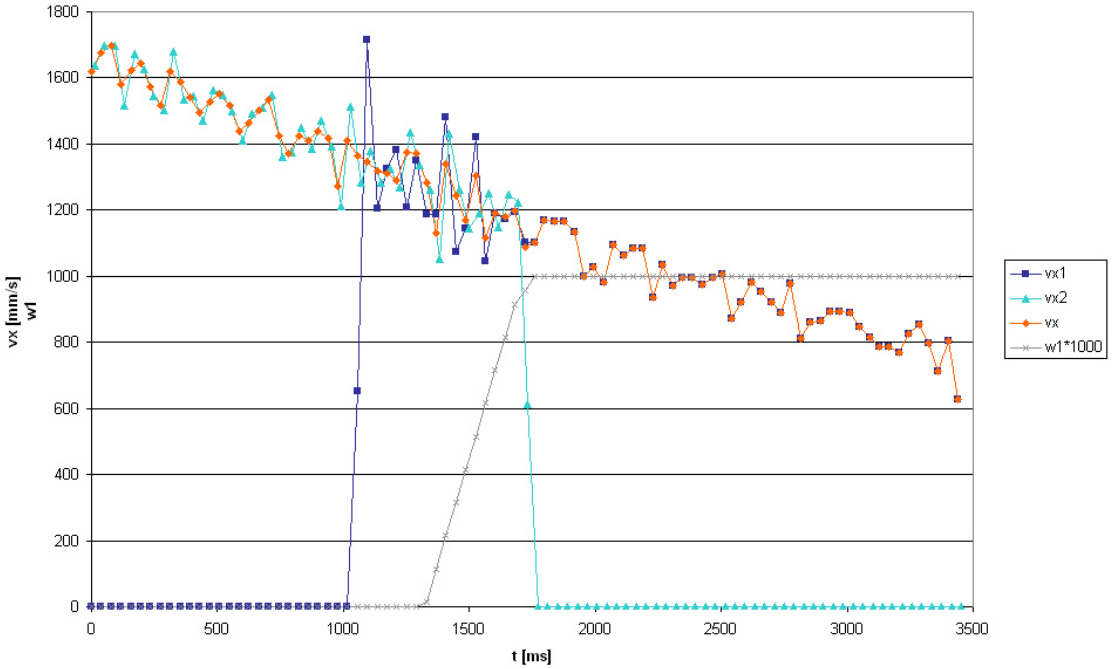


Abbildung 2.6: Darstellung der Ballgeschwindigkeit über der Zeit: dunkel blau die vom ersten ImageProcessor ermittelte Geschwindigkeit, türkis die vom zweiten ImageProcessor ermittelte Geschwindigkeit, orange die resultierende Geschwindigkeit des Vereinigungs-Algorithmus; In grau die verwendete Gewichtung für die Ballgeschwindigkeit des ersten ImageProcessor

2.5 Übertragung der CeilingCam Daten auf den Roboter

Um die Daten der CeilingCam auf dem Roboter zur Verfügung zu stellen, musste ein Medium gefunden werden, die Daten zu übertragen. Der verwendete Ansatz wird im folgenden beschrieben.

2.5.1 Allgemeines

Für die Übertragung der Daten der CeilingCam verwenden wir das WLAN an das die Roboter angeschlossen sind. Um die Daten an den Roboter zu versenden, gibt es verschiedene Protokolle, die gewählt werden können. Dies sind TCP, UDP Singlecast und UDP Broadcast. Dabei hat jedes dieser Protokolle seine Vor- und Nachteile.

2.5.1.1 TCP

Bei der Verwendung von TCP als Protokoll würde garantiert, dass die Pakete der CeilingCam zum Einen den Roboter überhaupt und zum Anderen auch noch in der richtigen Reihenfolge erreichen. Dies sind zwei Forderungen, die für unsere Verwendung der CeilingCam nicht notwendig sind. Pakete, die den Roboter nicht erreichen, haben keine Auswirkung auf die Verarbeitung, lediglich die Validität der auf dem Roboter zur Verfügung stehenden Daten sinkt, da diese auf einem Zeitpunkt weiter in der Vergangenheit beruhen.

TCP als Protokoll bringt auch Nachteile mit sich. So ist der Overhead des Protokolls im Vergleich zu UDP wesentlich größer. Desweiteren besteht das Problem, das jeder Roboter eine gesonderte Verbindung zum Server aufbauen müsste. Gleiches gilt für die Computer, auf denen die Daten der CeilingCam angezeigt werden sollen. Dies resultiert in einem hohen Verwaltungsaufwand auf dem Server. Außerdem besteht die Möglichkeit, dass der Server schneller Daten sendet, als der Roboter verarbeiten kann. Dies hätte zur Folge, dass der IP Stack des Roboters überlaufen würde und er abstürzt. Daher haben wir uns gegen diese Lösung entschieden.

2.5.1.2 UDP Singlecast

Die in Kapitel 2.5.1.1 beschriebenen Nachteile sind bei UDP Singlecast nicht vorhanden. Das Problem eines nicht ankommenden Paketes und dessen Auswirkungen ist immer noch vorhanden, doch kann der IP Stack des Roboters nicht volllaufen, da immer nur 1 Paket zwischengespeichert wird (falls ein Paket verloren geht, wird mit der Verarbeitung nicht gewartet, bis das Paket erneut übertragen wurde). Sollte ein weiteres Paket ankommen, bevor das Alte verarbeitet wurde, wird es einfach überschrieben. Wenn 2 Pakete in der falschen Reihenfolge ankommen, hat dies keine negativen Auswirkungen, da in jedem Paket ein exakter Zeitstempel enthalten ist. Der Nachteil, dass jeder Roboter eine seperate Verbindung zum Server öffnen

müsste, bestünde für UDP Singlecast wie für TCP weiterhin. Daher haben wir uns entschlossen, UDP Broadcast zu verwenden.

2.5.1.3 UDP Broadcast

Die von der CeilingCam gelieferten Daten werden per UDP Broadcast über das WLAN verteilt. Dies hat die gleichen Vorteile, wie eine Singlecast UDP Lösung, sowie den Vorteil, dass Daten nicht mehr explizit vom Server angefordert werden müssen, sondern kontinuierlich vom Server allen im Netz befindlichen Rechnern und Robotern zur Verfügung gestellt werden. Da im normalen Betrieb zumindest zwei Clients die Daten verarbeiten wollen, ist dies die effizienteste Übertragungsart. Daher haben wir uns letztendlich entschieden, diese Lösung zu verwenden.

2.5.2 Datenformat der versendeten CeilingCamDaten

Für jeden der 8 Roboter wird in jedem Datensatz, der von der CeilingCam versendet wird, jeweils ein zugehöriger Robotervektor $\vec{r_i}$ sowie der Ballvektor $\vec{b}$ zur Verfügung gestellt. Die Daten werden dabei mit doppelter Genauigkeit (doubles) übertragen. Dabei sieht der Robotervektor wie folgt aus:

$$\vec{r_i} = \begin{pmatrix} \mathrm{Pos}_x \\ \mathrm{Pos}_y \\ \mathrm{Rotation} \\ \mathrm{v}_x \\ \mathrm{v}_y \end{pmatrix} \tag{2.12}$$

Der Ballvektor beinhaltet folgende Einträge:

$$\vec{b} = \begin{pmatrix} \mathrm{Pos}_x \\ \mathrm{Pos}_y \\ \mathrm{v}_x \\ \mathrm{v}_y \end{pmatrix} \tag{2.13}$$

Zusätzlich zu diesen Informationen beinhaltet der Datensatz noch 2 Zeitstempel t und t_{sync}. Der Zeitstempel t wird benötigt, um die Zeit auf dem Roboter zu synchronisieren und somit eine Umrechnung von CeilingCamServerzeit in lokale Roboterzeit zu ermöglichen. Außerdem wird der t_{sync} Zeitstempel als Feld für Debug-Informationen genutzt. Er existiert aus Kompatibilitätsgründen mit dem von der Universität Darmstadt entwickelten Deckenkamera Protokoll. Als Debug-Informationen werden unter anderem versendet, welche Objekte vom Server tatsächlich erkannt wurden sowie eine eindeutige Bildnummer. Der komplette Datensatz

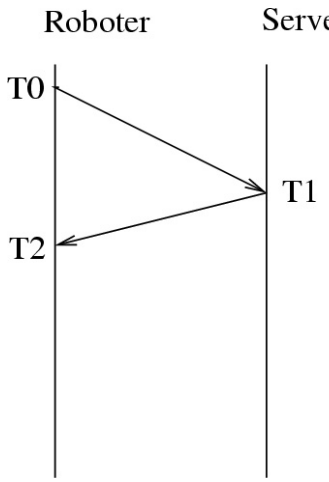


Abbildung 2.7: Überblick TimeSync Algorithmus

kann somit durch den Vektor $\vec{C}_{\text{data}}$ beschrieben werden.

$$\vec{C}_{\text{data}} = \begin{pmatrix} t \\ t_{\text{sync}} \\ \vec{r}_0 \\ \vec{r}_1 \\ \vdots \\ \vec{r}_7 \\ \vec{b} \end{pmatrix}$$

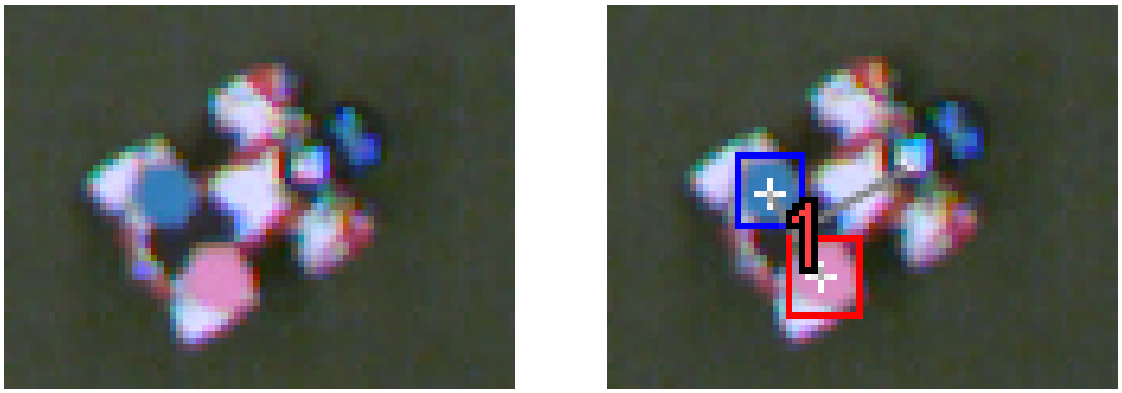
2.5.3 Zeitsynchronisation zwischen Roboter und CeilingCam-Server

Um die Daten auf dem Roboter auswerten zu können ist es nötig, die im *WorldState* übertragene Serverzeit in eine lokale Roboterzeit umzurechnen. Dazu verwenden wir einen Algorithmus, der in Abbildung 2.7 dargestellt ist und im Folgenden erläutert wird.

Um den Zeitstempel des *WorldStates* in Roboterzeit umrechnen zu können, besteht die Aufgabe in der Bestimmung der Zeitdifferenz zwischen Roboter und Server. Wie in Abbildung 2.7 dargestellt, sendet der Roboter zunächst ein Paket an den Server (Ping). Dabei speichert er die lokale Zeit t_0. Der Server sendet, sobald er das Paket empfangen hat, ein Paket mit der aktuellen Serverzeit t_1 zurück (Pong). Sobald der Roboter dieses Paket vom Server empfängt, kann er die Zeitdifferenz Δt berechnen.

$$\Delta t = t_1 - \frac{t_0}{2} - \frac{t_2}{2}$$

Dabei wird vorrausgesetzt, dass die Verzögerung durch das Netzwerk nahezu konstant ist und die Varianz vernachlässigbar klein. Um den Fehler von Ausreißern zu



(a) Originalansicht (b) Mit DebugDrawings

Abbildung 2.8: Prototyp eines neuen, schmaleren Markers aus Sicht der CeilingCam

minimieren, wird ein Ringpuffer verwendet. Dieser glättet Δt über die letzten 10 Messwerte und stabilisiert so die Zeitumrechnung.

Um stets ein aktuelles Δt zu haben, synchronisiert der Roboter alle 100 eingehenden *WorldStates* mit dem CeilingCam-Server. Dies hat sich als Wert erwiesen der einen guten Kompromiss zwischen Overhead und Synchronisation bietet.

2.6 Ausblick

Die bisher verwendeten Marker reichen seitlich weit über die Roboter hinaus und behindern die Roboter beim Fußballspielen. Schmalere Marker ermöglichen den Einsatz auch in Testspielen. Allerdings ist durch das Zusammenrücken von rotem und blauen Markerpunkt eine Verschlechterung der Rotationsbestimmung und damit auch der Positionsbestimmung in y-Richtung relativ zum Roboter zu erwarten.

Eine Verbesserung der Erkennungsgenauigkeit insbesondere in den Randbereichen der Spielfeldhälften könnte durch eine höhere Positionierung der Kameras mit entsprechend angepassten Objektiven erzielt werden. Zum Einen ist mit einer geringeren Verzerrung zu rechnen und zum Anderen ist der Blickwinkel auf die Marker steiler, so dass die Erkennung robuster gegenüber leichten Markerneigungen werden würde.

3 WalkingEngine

Eine sehr wichtige Komponente des Roboterfußballs ist eine effiziente Fortbewegung der Spieler. Da die von uns verwendeten AIBOs nicht mit Rädern, sondern mit vier Beinen ausgestattet sind, ist eine komplexe Ansteuerung der Servos in den Beinen nötig, um eine Gesamtbewegung zu generieren, die als Laufen bezeichnet werden kann und den Roboter in die Lage versetzt, jeden beliebigen Punkt des Spielfelds kontrolliert und schnell zu erreichen. Das in unserem Framework für diese Aufgabe zuständige Modul heißt WalkingEngine und hat die Aufgabe, eine Bewegungsanfrage des Verhaltens (im Folgenden WalkRequest genannt), bestehend aus je einer Translation in x- und y-Richtung sowie einer Rotation, in eine entsprechende Ansteuerung der Beinservos umzusetzen und auf diese Weise eine Laufbewegung zu generieren, die im Rahmen des technisch Möglichen am ehesten der angeforderten Bewegung entspricht. Auf Grund der Tatsache, dass die Orientierung des Roboters zum Ball, oder zu seinen Mitspielern eine wichtige Rolle dabei spielt, effizient ins Spielgeschehen eingreifen zu können, ist es wichtig, ohne die Orientierung des Roboters zu verändern, in der Lage zu sein, in alle Richtungen laufen zu können. Diese Fähigkeit wird auch als omnidirektionales Laufen bezeichnet.

3.1 Beschreibung des Problems

Für den Robocup 2004 wurde vom GermanTeam für diese Aufgabe eine auf inverser Kinematik basierende WalkingEngine, die GT2004WalkingEngine, verwendet, die nach dem so genannten Rädermodell (siehe [4]) arbeitet. Dabei werden die Füße des AIBOs auf einer in diesem Fall rechteckigen Bahnkurve um eine vorher festgelegte Nullposition bewegt und die zum Erreichen der so definierten Beinstellungen nötigen Gelenkwinkel der Beinservos über inverse Kinematik berechnet.

Da sich im Spielkontext gezeigt hatte, dass selbst kleine Verbesserungen der Laufgeschwindigkeit, der Manövrierfähigkeit oder des Ballhandlings großen Einfluss auf den Ausgang eines Spiels haben können, stellte sich die Frage, ob durch eine Überarbeitung oder Neuentwicklung der vorhandenen WalkingEngine eine signifikante Verbesserung der aktuellen Technik erreichbar war. Um mögliche Schwachstellen des alten Konzeptes aufzudecken, wurde die reale Bewegung der Beine des Roboters analysiert. Zu diesem Zweck wurden die während des Laufens erreichten Winkel der Beingelenke und die zur gleichen Zeit angesteuerten Gelenkwinkel aufgezeichnet. Nach der Rückrechnung der Fußbewegung aus den Gelenkwinkeln war zum Einen ein Phasenversatz zwischen den realen und den angesteuerten Werten zu erkennen und zum Anderen, dass die Form der real durchlaufenen Bahnkurven um den Null-

punkt keineswegs mit der angesteuerten rechteckigen Form übereinstimmte. Dabei waren die zu beobachtenden Abweichungen nicht nur zweidimensional auf die Ebene des Rechtecks beschränkt, sondern besaßen eine deutliche Ausdehnung in allen drei Raumdimensionen. Das ließ den Schluss zu, dass durch die Verwendung von zweidimensionalen, rechteckigen Bahnkurven zur Berechnung der Fußbewegungen nur eine unzureichende Ausnutzung der Freiheitsgrade der Beine des AIBOs gegeben war und wahrscheinlich effizientere Laufbewegungen durch die Verwendung von dreidimensionalen Bahnkurven möglich sind.

3.2 Lösungsansätze und Beschreibung der alten Lösung

Zur Überprüfung dieser Theorie bestand nun die Wahl, die bestehende WalkingEngine um eine entsprechende Unterstützung von dreidimensionalen Bahnkurven zu erweitern, oder eine vollkommen neue WalkingEngine zu programmieren, die diese Idee umsetzt.

Da eine Neuentwicklung in der Regel ein besseres Verständnis der Systemeigenheiten zur Folge hat, was einer möglichen Fehlerquelle vorbeugt und uns die GT2004-WalkingEngine an vielen Stellen implemetationstechnisch unflexibel erschien, haben wir uns für die zweite Lösung entschieden. Im Folgenden soll nun ein kurzer Überblick über die Funktionsweise der alten Lösung gegeben werden, da die neue WalkingEngine auf dem gleichen Grundprinzip basiert und so auch die Unterschiede zwischen den beiden Systemen deutlicher werden.

3.2.1 Generelle Definitionen

Für alle im folgenden Abschnitt zu findenden Betrachtungen werden Roboterkoordinaten verwendet. Das Roboterkoordinatensystem eines AIBOs ist relativ zu einem 2,5 mm hinter dem Drehzentrum des Nackengelenks befindlichen Punkt wie in Abbildung 3.1 definiert.

3.2.2 Rädermodell und Omnidirektionales Laufen

Eine Laufbewegung, bei der (nach dem oben beschriebenen Verfahren) die zeitliche Abfolge der angesteuerten Beinwinkel durch eine Bewegung der Roboterfüße entlang einer rechteckigen Bahnkurve und Rückrechnung der Beinwinkel mit inverser Kinematik ermittelt wird, erzeugt für den Fall, dass die Bahnkurve parallel zur xz-Ebene des Roboterkoordinatensystems liegt und die Bahnkurve im Uhrzeigersinn um die y-Achse des Roboterkoordinatensystems durchlaufen wird, eine effiziente Vorwärtsbewegung des AIBOs. Dabei werden zwei diagonal zueinander liegende Beine phasengleich, die beiden anderen mit einer um 180 ° verschobenen Phase angesteuert. Bei dieser dem Trab ähnelnden Gangart ist gewährleistet, dass sich immer zwei der Beine auf dem Boden befinden. Auf Grund der relativ schnellen Schrittfrequenz und

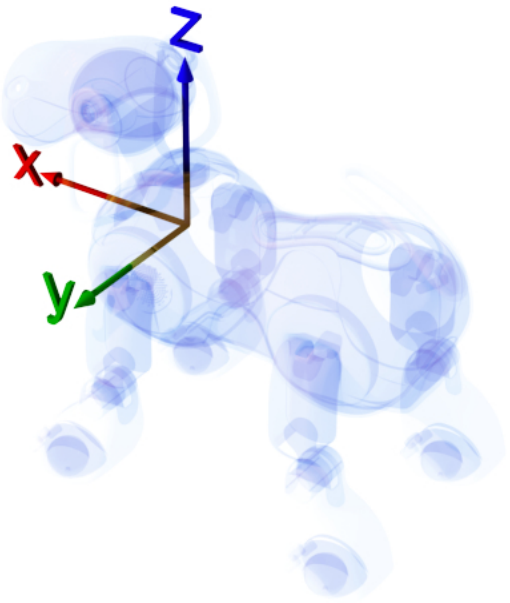


Abbildung 3.1: Das Koordinatensystem eines AIBOs

den nur wenig vom Boden abgehobenen Füßen wird eine dynamische, aber trotzdem sehr stabile Laufcharakteristik erreicht.

In die GT2004WalkingEngine war zwar bereits eine Unterstützung von unterschiedlich geformten, zweidimensionalen Bahnkurven, wie Rechtecken, Ellipsen, Halbellipsen und Ovalen integriert, allerdings hatte sich bei entsprechenden Tests gezeigt, dass die zusätzlichen Formen keinerlei Vorteile gegenüber den normalerweise verwendeten Rechtecken zu haben schienen. Aus diesem Grund waren Rechtecke die einzige in der finalen Version der GT2004WalkingEngine verwendete Bahnkurvenform.

Die eingesetzten Rechtecke werden durch ihre Höhe, ihre Länge und anhand von drei Parametern, die die Geschwindigkeit auf den vier Kanten festlegen, definiert. Die *Groundphase* bezeichnet die Zeitspanne der Fußbewegung, bei der angenommen wird, dass sich der Fuß des Roboters auf dem Boden befindet. Um eine Vorwärtsbewegung zu generieren, bewegt sich der Fuß in dieser Laufphase in negativer Richtung entlang der x-Achse des Roboterkoordinatensystems. Die beiden verbleibenden Parameter sind die *Liftphase* und die *Loweringphase*, die jeweils die für das Anheben bzw. das Absenken des Fußes benötigte Zeitspanne angeben. Das fehlende obere Segment des Rechtecks, bei dem der Fuß sich in positiver Richtung entlang der x-Achse bewegt, wird *Airphase* genannt und ergibt sich aus den drei vorangegangenen Phasen. Da sich die Länge der vorderen Beine des AIBOs von der der hinteren Beine unterscheidet, werden die an den Fußnullpositionen platzierten Rechtecke um einen Winkel um die y-Achse rotiert, damit die untere Kante parallel zum Boden verläuft.

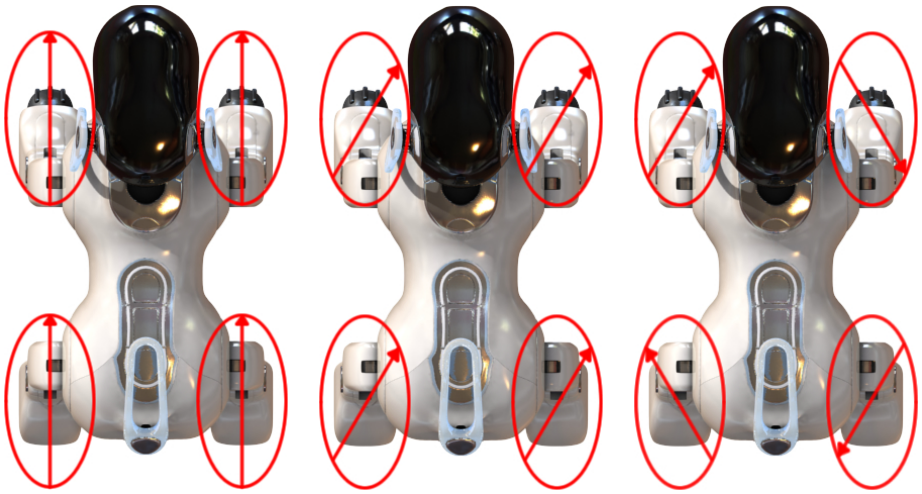
Setzt man einen Untergrund mit ausreichend hoher Haftreibung voraus und lässt man alle Effekte außer Acht, die durch eine Abrollbewegung der Füße auf dem Boden entstehen, so ist die Bewegungsgeschwindigkeit der Füße während der *Groundpha-*

Abbildung 3.2: Die vier Phasen beim Laufen

se ungefähr gleich der Fortbewegungsgeschwindigkeit des Roboters. Bei konstant bleibender Schrittfrequenz lässt sich eine Geschwindigkeitsänderung also durch eine Skalierung der Rechtecke in x-Richtung erreichen. Um die Rechtecke dabei in Position zu halten, wird diese Skalierung nicht um den Ursprung des Roboterkoordinatensystems, sondern um die Fußnullpositionen durchgeführt, die auch gleichzeitig die Ursprünge der Koordinatensysteme der Rechtecke angeben. Dabei ist zu beachten, dass die in der *Groundphase* erreichbare Geschwindigkeit der Füße durch Faktoren wie den Bewegungsspielraum der Beine, die Geschwindigkeiten der Beinservos und die Einstellung der PID-Regler, die die Ansteuerung der Servos kontrollieren begrenzt wird.

Echtes omnidirektionales Laufen kann nun dadurch erreicht werden, dass man die Rechtecke nach der Skalierung in x-Richtung zusätzlich um die z-Achse ihres Koordinatensystems dreht (siehe Abbildung 3.3). Auf Grund der Tatsache, dass der Bewegungsspielraum der Beine eines AIBOs in y-Richtung wesentlich geringer ist, als in x-Richtung muss dabei allerdings die Skalierung in Abhängigkeit vom Drehwinkel um die z-Achse angepasst werden. Aus zwei Parametern, die die maximalen in x- und y-Richtung möglichen Entfernungen der Füße vom Fußnullpunkt angeben, kann dann eine Skalierung berechnet werden, bei der die Füße nur innerhalb der tatsächlich möglichen Bewegungen angesteuert werden. Bei gleichzeitiger Skalierung und Drehung der Polygone aller vier Beine können auf diese Weise Bewegungen des Roboters in alle Richtungen erreicht werden, ohne dabei seine Orientierung zu verändern.

Für Drehbewegungen ist es erforderlich die Skalierungsfaktoren und Drehwinkel der Rechtecke getrennt zu betrachten. Werden die Skalierungsfaktoren und Drehwinkel so berechnet, dass die Bewegung aller Füße während der jeweiligen Groundphase tangential zu vier Kreisen mit gemeinsamem Mittelpunkt verlaufen und die Skalierung proportional zum Radius der Kreise ist, resultiert daraus eine Gesamtbewegung, die den Roboter um diesen Punkt rotieren lässt (siehe Abbildung 3.4). Die so erreichbaren Bewegungen decken alle möglichen Kombinationen aus Translations- und Rotationsbewegungen ab. Bei reinen Translationsbewegungen wird der Mittelpunkt faktisch ins Unendliche verlegt. Für eine reine Rotationsbewegung ist bei



(a) Vorwärtslaufen (b) Diagonallaufen (c) Rotation

Abbildung 3.3: Rotation der Bahnkurven beim Rädermodell für unterschiedliche Laufbewegungen

diesem Verfahren der Mittelpunkt des Roboters als Drehpunkt zu wählen.

3.2.3 Optimierte Parametersätze

Die Gesamtheit aller Parameter, wie Fußnullpositionen und Eigenschaften der Rechtecke, die die Eigenschaften eines Laufes bestimmen, wird als Parametersatz bezeichnet. Es ist zwar möglich einen Parametersatz zu entwickeln, der bei Läufen in alle Richtungen gute Geschwindigkeiten erreicht, es hat sich jedoch gezeigt, das mit für spezielle WalkRequests optimierten Parametersätzen oft wesentlich höhere Geschwindigkeiten möglich sind. Aus diesem Grund wurden für die GT2004WalkingEngine mit Hilfe eines automatisierten Lernverfahrens viele spezialisierte Parametersätze entwickelt, zwischen denen im laufenden Betrieb interpoliert wird, um alle WalkRequests mit optimaler Geschwindigkeit ausführen zu können. Da die GT2004WalkingEngine in hohem Maße parametrisierbar ist und sich praktisch alle Eigenschaften des Systems über die Parametersätze beeinflussen lassen, können auf diese Weise sehr unterschiedliche Läufe generiert und miteinander kombiniert werden.

3.2.4 Odometrie

Für eine effiziente Nutzung der zur Verfügung stehenden Läufe ist es wichtig sicherzustellen, dass die beim Ausführen eines WalkRequests generierte Bewegung möglichst genau den vom Verhalten angeforderten Translations- und Rotationsgeschwindigkeiten entspricht. Ist dies auf Grund technischer bzw. physikalischer Beschränkungen wie der Maximalgeschwindigkeit der Servos, des aktuellen Bewegungszustandes des AIBOs oder eines nicht erfüllbaren WalkRequests nicht möglich, muss die real ausgeführte Bewegung bekannt sein und das Verhalten und die Selbstlokali-

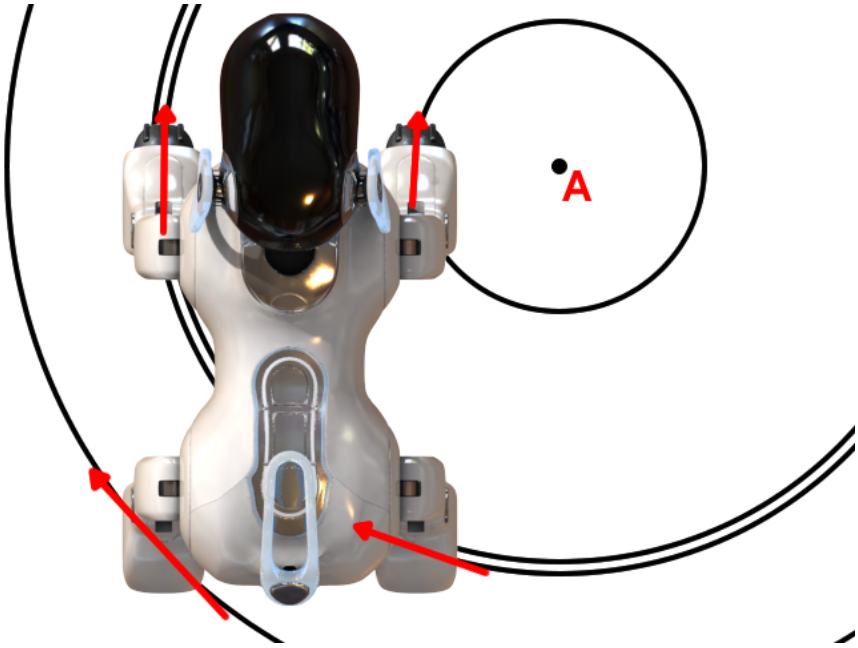


Abbildung 3.4: Virtuelles Rotationszentrum der rotierten Bahnkurven bei kombinierten Translations-/Rotationsbewegungen

sierung über die Abweichung informiert werden. Um das zu gewährleisten, wurden die mit den verschiedenen Parametersätzen maximal möglichen Geschwindigkeiten ausgemessen und daraus Korrekturfaktoren berechnet, mit denen die reale Gesamtbewegung des Roboters der angesteuerten Bewegung angepasst werden konnte.

3.3 Beschreibung der neuen Lösung

Es wurden verschiedene Ansätze diskutiert, um die Effizienz der WalkingEngine zu steigern. Die letztendlich gewählte Lösung soll im Folgenden genauer betrachtet werden.

3.3.1 Anforderungen

Die neu zu entwickelnde WalkingEngine sollte auf dem Konzept des Rädermodells aufbauen, da sich dieses als sehr effizient erwiesen hatte. Weiterhin sollten die bisher als Bahnkurven verwendeten Rechtecke durch eine nicht mehr auf eine zweidimensionale Ebene beschränkte Lösung ersetzt werden, um eine optimale Ausnutzung der Freiheitsgrade der Beine der Roboter zu gewährleisten. Außerdem sollte die Qualität der Odometriedaten der neuen WalkingEngine auf einem der alten Lösung äquivalenten oder besseren Niveau liegen.

3.3.2 Polygone als Bahnkurven

Ein möglicher Lösungsansatz war die Verwendung von dreidimensionalen Polygonen als Bahnkurven, anstelle der bisher verwendeten zweidimensionalen Rechtecke. Polygone erschienen zum Einen komplex genug, um den neuen Anforderungen an die Bahnkurven gerecht zu werden und zum Anderen nicht so komplex wie z.B. Bezier-Kurven, so dass ein zu hoher Verbrauch an Rechenzeit zu befürchten gewesen wäre.

Um eine größtmögliche Ausnutzung der Freiheitsgrade der Beine zu erreichen, sollte für jeden Lauf die Nutzung von unterschiedlichen Polygonen für jedes der vier Beine möglich sein. Die dafür einzusetzenden Polygone wurden wie folgt definiert:

Ein Polygon besteht aus einer frei wählbaren Anzahl von Punkten im $\mathbb{R}^3$ und einer zu jedem Punkt gehörenden Segmentlänge. Die Segmentlänge gibt den Bruchteil der Zeit im Verhältnis zu einem Komplettdurchlauf des Polygons an, der verwendet werden soll, um das entsprechende Segment zu durchlaufen. Für jedes Polygon wird außerdem eine Schrittfrequenz, in Form einer Zeitspanne für einen kompletten Durchlauf, angegeben.

Für den Fall, dass die zum omnidirektionalen Laufen nötigen Rotationen um die z-Achse nicht, wie in der GT2004WalkingEngine um den Nullpunkt des Koordinatensystems der Bahnkurve, also den jeweiligen Fußnullpunkt, sondern um das Massezentrum der Eckpunkte des Polygons ausgeführt werden, lassen sich alle Eigenschaften eines Laufes in den Parametern eines Satzes aus vier Polygonen kodieren. Die zusätzlichen Rotationen um die y-Achse zur Parallelisierung von Boden und unteren Bahnkurvenbereich entfallen, weil Rotationen dieser Art durch die nicht weiter eingeschränkte Form der Polygonpunkte praktisch überflüssig werden. Die Fußnullpositionen der Beine, an denen die Polygone platziert werden, sind nicht wie in der alten Lösung laufabhängige Parameter, sondern bleiben bei unterschiedlichen Läufen konstant, da eventuell nötige Verschiebungen der Bahnkurven durch eine relative Verschiebung der Koordinaten der Polygonpunkte realisiert werden können.

3.3.3 Parametersätze

Da auch bei der Verwendung von Polygonen kein Satz von Bahnkurven zu erwarten war, der für jeden denkbaren WalkRequest optimale Ergebnisse liefert, erschien die Verwendung von auf bestimmte WalkRequests spezialisierten Polygonsätzen zweckmäßig. Zwischen diesen spezialisierten Polygonsätzen sollte dann ein jeweils für den aktuellen WalkRequest zugeschnittener Polygonsatz durch Interpolation berechnet werden. Der primäre Auswahlfaktor für die spezialisierten WalkRequests war, die maximal mögliche Laufgeschwindigkeit in alle denkbaren Richtungen zu gewährleisten. Ein sekundärer Auswahlfaktor war es, in Bewegungsbereichen von Spielaktionen, die eine hohe Präzision erfordern, eine möglichst erschütterungsarme Bewegung einsetzen zu können, wodurch sich eine Verbesserung der Qualität der für andere Module zur Verfügung stehenden Informationen (z.B. weniger verwackelte Kamerabilder) ergeben sollte. Unter diesen Gesichtspunkten wurden die in Abbildung 3.5 dargestellten WalkRequests für eine gezielte Optimierung ausgewählt.

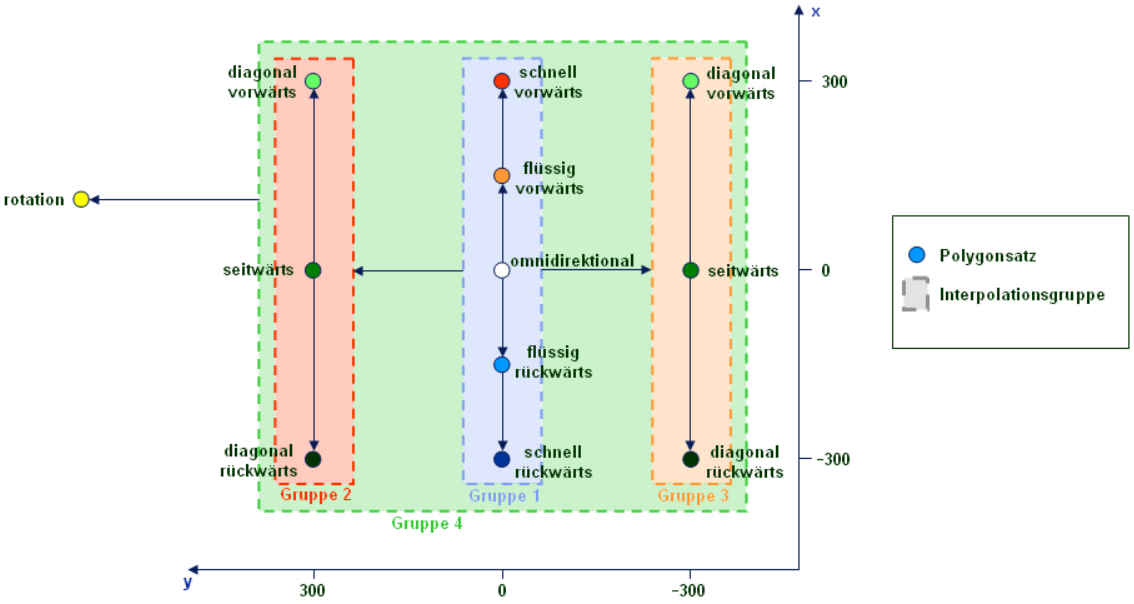


Abbildung 3.5: Interpolation und Lage der Polygonsätze im internen Ansteuerungsbereich

Neben einem optimierten Polygonsatz für langsames Laufen in alle Richtungen existieren acht Polygonsätze für maximale Geschwindigkeit in die acht Hauptrichtungen, zwei Polygonsätze für erschütterungsarmes Vor- und Rückwärtslaufen im mittleren Geschwindigkeitsbereich, sowie ein Polygonsatz für schnelle Rotation. Alle Polygonsätze, bis auf den letztgenannten, enthalten keine Rotationskomponente.

Sind für die gewählten WalkRequests optimierte Polygonsätze erzeugt worden, wird aus diesen ein für einen bestimmten WalkRequest zugeschnittener Polygonsatz durch eine dreistufige, lineare Interpolation berechnet. Das setzt voraus, dass alle Polygone in einem Polygonsatz die gleiche Anzahl von Polygonpunkten verwenden. Aus diesem Grund werden alle Polygonsätze, die in einer laufenden Instanz der WalkingEngine verwendet werden zu einem Parametersatz zusammengefasst, in dem neben der Anzahl der verwendeten Polygonpunkte auch die für alle Polygone gleichen Fußnullpositionen und der Phasenversatz für die Ansteuerung der Beine gespeichert werden.

Die Berechnung eines Polygonsatzes für einen bestimmten WalkRequest wird nun so realisiert, dass zunächst aus zweien der in Gruppe 1 (siehe Abbildung 3.5) enthaltenen Polygonsätze, entsprechend der x-Komponente des WalkRequests, ein neuer Polygonsatz durch lineare Interpolation aller enthaltenen Parameter erzeugt wird. Je nach Vorzeichen der y-Komponente des WalkRequests wird mit einer der Gruppen 2 oder 3 ebenso verfahren. Aus den zwei daraus resultierenden Polygonsätzen wird dann entsprechend der y-Komponente des WalkRequests ein Polygonsatz für Gruppe 4 interpoliert, der bereits optimal ist, falls der aktuelle WalkRequest keine Rotationskomponente enthält. Enthält der aktuelle WalkRequest jedoch eine Rotationskomponente, muss der so berechnete Polygonsatz noch mit dem für Ro-

tationen optimierten Satz gemischt werden. Diese letzte lineare Interpolation wird anhand des Betrages der Rotationskomponente durchgeführt, da der Polygonsatz für schnelle Rotationen systembedingt gleichermaßen für Links- und Rechtsrotationen geeignet ist. Zur Berechnung eines für den jeweiligen WalkRequest bestgeeigneten Polygonsatzes, aus den zur Verfügung stehenden optimierten Polygonsätzen, sind also insgesamt 4 Interpolationen zwischen Polygonsätzen nötig.

3.3.4 WalkRequests

Bei den WalkRequests wird zwischen einem internen und einem externen Ansteuerungsbereich der WalkingEngine unterschieden. Während die externen WalkRequests vom Verhalten verwendet werden und reale Geschwindigkeiten von ±450 mm/s für die Translation und ±200 °/s für die Rotation angeben, wird für die internen WalkRequests ein Ansteuerungsbereich von ±300 für alle drei Parameter verwendet. Die Optimierung der einzelnen Polygonsätze wird im internen Ansteuerungsbereich der WalkRequests vorgenommen, da im Vorhinein nicht bekannt ist, welche tatsächliche Geschwindigkeit bei einer bestimmten Ansteuerung erreicht wird.

3.3.5 Arbeitsweise der neuen WalkingEngine

Bei der Ausführung der WalkRequests wird in mehreren Schritten eine passende Ansteuerung der Beinmotoren generiert.

3.3.5.1 Glättung der WalkRequests

Bei jedem Aufruf der WalkingEngine wird der als Parameter übergebene WalkRequest zunächst auf den internen Ansteuerungsbereich umgerechnet und dann geglättet, um die maximalen Geschwindigkeitsänderungen kontrollieren zu können. Bei sich sprunghaft stark ändernden WalkRequests findet zwar bereits eine Glättung im Verhalten statt, um Zitterbewegungen bei schnellem, wiederholtem Umschalten zwischen zwei Zuständen zu verhindern, eine direkte Kontrolle dieser Werte in der WalkingEngine selbst, hat sich jedoch als vorteilhaft erwiesen. Bei der Einstellung der Glättungsparameter kann je nach Anforderung eine besonders flüssige Bewegung der Roboter erreicht werden, die allerdings durch eine hohe Ansteuerungsträgheit erkauft wird, oder eine sehr direkte Ansteuerung bei der bis zu 5 mal höhere Beschleunigungen als mit der GT2004WalkingEngine möglich sind. Als für unsere Anforderungen optimal haben sich Werte ergeben, die gerade hoch genug sind, um Zitterbewegungen bei sprunghafter Ansteuerung zu verhindern. Dadurch wird ein sehr agiles Verhalten erreicht, das in der Regel innerhalb von weniger als zwei Sekunden auf volle Geschwindigkeit in alle Richtungen beschleunigen kann und in gleicher Zeit aus jeder beliebigen Bewegung zum Stillstand kommt. Durch die hohen Beschleunigungen stieg zum Einen die Beanspruchung der Hardware und zum Anderen war eine starke Anpassung aller mit dem Ball interagierenden Verhaltensteile

notwendig, da dieses in den vergangenen Jahren auf die sehr flüssigen Bewegungen der alten WalkingEngine optimiert wurde.

3.3.5.2 Lauftypen

Nach der Glättung des WalkRequests wird ein dafür optimaler Polygonsatz berechnet. Dabei spielen nicht nur die Ansteuerungswerte der WalkRequests eine Rolle, sondern auch der aktuell gesetzte Lauftyp. Die Definition von Lauftypen soll spezielle Bewegungsanfragen vom Verhalten ermöglichen, wie zum Beispiel optimiertes Laufen mit dem Ball zwischen den Vorderbeinen. Zur Realisierung von Lauftypen wurde bis zu diesem Zeitpunkt für jeden Lauftyp eine speziell parameterisierte Instanz einer WalkingEngine erzeugt und diese dann nach vorheriger externer Prüfung des Lauftyps aufgerufen. Dabei verursachte das Umschalten der WalkingEngine stets eine kleine Verzögerung, da in der GT2004WalkingEngine nur gewisse Zeitpunkte eines Laufes als umschaltsicher markiert waren, um die verschiedenen Instanzen synchronisieren zu können. Die neue WalkingEngine unterstützt, im Gegensatz zur alten Lösung, die interne Verarbeitung von unterschiedlichen Lauftypen. Das hat nicht nur den Vorteil, dass nur noch eine Instanz der WalkingEngine ausgeführt werden muss, sondern auch, dass beim Umschalten zwischen den Lauftypen ein flüssiger, verzögerungsfreier Übergang möglich ist. Erreicht wird dies durch eine vom Lauftyp abhängige Verwendung von unterschiedlichen, für die jeweiligen Aufgaben optimierten Polygonsätzen.

3.3.5.3 Polygontransformationen

Ist ein für den aktuellen WalkRequest und Lauftyp optimaler Polygonsatz berechnet worden, werden die Polygone, wie oben beschrieben, entsprechend der Geschwindigkeit skaliert und gemäß gewünschter Laufrichtung und Rotation um die z-Achse ihres Massezentrums rotiert. Eine zusätzliche Skalierung wegen des geringeren Spielraums in y-Richtung wie bei der GT2004WalkingEngine ist nicht notwendig, weil für alle Laufrichtungen spezialisierte Polygonsätze verwendet werden. Da bei der Optimierung nur Polygonsätze entstehen, die innerhalb des Bewegungsfreiraums der Beine liegen und bei der Interpolation zwischen den spezialisierten Polygonsätzen die konvexe Hülle aller an einem Bein verwendeten Polygone nicht verlassen wird, funktionieren auch alle interpolierten Polygonsätze ohne zusätzliche Skalierung. Nach der korrekten Positionierung der Polygone wird dann die aktuelle Position auf den Polygonen aus der Laufphase bestimmt und daraus, zusammen mit den Fußnullpositionen per inverser Kinematik, die nötigen Ansteuerungswinkel für die Beingelenke berechnet.

Erschwerend kam hinzu, dass die zur Skalierung der Polygone benötigte Groundphase nicht mehr so gut wie bei der Verwendung von Rechtecken als Bahnkurven angenähert werden konnte. Durch die flexiblere Form der Polygone war es besonders bei der Verwendung von vielen Segmenten wahrscheinlich, dass die Groundphase aus mehreren Segmenten bestand. Da ebenso wenig die genauen Bodenkontaktpunkte der Füße der Roboter berechnet werden konnten, musste die Groundphase wie folgt

geschätzt werden: Zunächst werden die beiden in x-Richtung extremen Punkte des Polygons ermittelt und an diesen Stellen das Polygon in zwei Polygonzüge zertrennt. Dann wird berechnet, welcher dieser beiden Polygonzüge durchschnittlich näher am Boden liegt und die Summe der Segmentzeiten dieses Polygonzuges als Groundphase verwendet.

3.4 Optimierung von Polygonsätzen

Nach der Entwicklung eines Basissystems, mit dem Polygone als Bahnkurven verwendet werden konnten, war es erforderlich die Möglichkeiten dieses neuen Systems zu evaluieren. Es musste vor allem überprüft werden, ob, und in wie weit optimierte Polygone der bisherigen GT2004WalkingEngine überlegen waren.

Um die Möglichkeiten des neuen Systems voll auszuschöpfen, aber auf keinen Fall Probleme mit der zur Ausführung nötigen Rechenzeit zu bekommen, wurden zunächst Polygonsätze mit jeweils acht Segmenten für jedes der vier Polygone betrachtet. Diese sollten in der Lage sein, ausreichend abgerundete Ansteuerungskurven bei akzeptabler Rechenzeit zu generieren. Bei vier Parametern pro Segment und unter Hinzunahme der für jeden Polygonsatz gespeicherten Frequenz, führte dies zu einer Anzahl von 129 optimierbaren Parametern für einen Polygonsatz. An eine manuelle Optimierung der Polygone war also nicht zu denken. Da bei der Optimierung der Parameter für die alte WalkingEngine gute Erfahrungen mit evolutionären Algorithmen gemacht wurden, bot sich dieser Ansatz hier ebenfalls an.

3.4.1 Bewertbarkeit von Laufeigenschaften

Durch die parallele Entwicklung der CeilingCam stand ein hinreichend genaues, externes Instrument zur Messung der Geschwindigkeit eines AIBOs zur Verfügung. Bei einigen Probemessungen mit den internen Beschleunigungssensoren des AIBOs waren außerdem Unterschiede zwischen Läufen mit verschiedener Laufruhe zu erkennen. Damit waren alle Vorraussetzungen erfüllt, um die beiden für einen Lauf wichtigen Eigenschaften, Geschwindigkeit und Laufruhe, maschinell erfassen und bewerten zu können. Weitere wichtige Eigenschaften eines Laufes, wie Präzision beim Umgang mit dem Ball oder die Qualität des resultierenden Laufes bei der Interpolation mit anderen Polygonsätzen, schienen mit akzeptablem technischem und zeitlichem Aufwand nicht maschinell erfassbar zu sein. Also waren nur die Geschwindigkeit und die Werte der Beschleunigungssensoren als Parameter für die Fitnessfunktion verwertbar.

3.4.2 Evolutionsstrategie

Für die Optimierung der letztendlich verwendeten Polygonsätze wurde ein einfacher evolutionärer Algorithmus mit unbeschränkter Mutation, Rang-Selektion und einer $+$-Strategie verwendet, bei dem die Individuen eine maximale Lebensdauer von 2

Generationen besaßen. Dadurch sollten die Vorteile einer +-Strategie (schnelle Konvergenz), mit denen einer ,-Strategie (größere Resistenz gegenüber lokalen Maxima) kombiniert werden. Die Größe der Generationen wurde auf 24 festgelegt und die Anzahl der pro Generation selektierten Eltern lag bei 6. Die beiden letztgenannten Werte wurden durch einige Probeläufe experimentell ermittelt.

Zur Erzeugung von Individuen für eine neue Generation wurde eine Kombination aus Kreuzung und Mutation eingesetzt, wobei für die Kreuzungen jeweils eine 50%ige Überblendung verwendet wurde. Die für die Mutation benutzten Strategieparameter wurden je nach gewünschter Lokalität des Optimierungslaufes geschätzt und dann mit in die Kreuzungen und Mutationen einbezogen, so dass ein selbstadaptives Verhalten des Algorithmus erreicht wurde.

3.4.3 Wahl geeigneter Evolutionsstartpunkte

Vor dem Start jedes Optimierungslaufs wurde ein Polygonsatz mit Startwerten initialisiert und aus diesem die erste Generation erzeugt. Als geeignete Startwerte wurden dabei Polygone verwendet, die in Form und relativer Positionierung zu den Schultergelenken in etwa den von der alten WalkingEngine verwendeten Rechtecken entsprachen. Dadurch war für den Fall einer nicht zu groß gewählten Mutationsstärke sichergestellt, dass die ersten Generationen ausreichend viele funktionierende Läufe enthielten und nicht nur Individuen, bei denen keine effiziente Vorwärtsbewegung des Roboters mehr möglich war. Bestanden für einen Lauf spezielle Anforderungen wie vorgezogene Vorderbeine zum Laufen mit gegriffenem Ball oder eine möglichst hohe Kopfposition, wurde diese Standardinitialisierung auf der Basis geschätzter Änderungen, die diese Laufeigenschaften begünstigen müssten, manuell modifiziert.

3.4.4 Das Messverfahren zur Bewertung der Fitness

Zur Vermessung der einzelnen Läufe wurde ein Verhalten programmiert, das den Roboter wiederholt gerade Strecken unter der CeilingCam ablaufen ließ. Nach einer Zeit von 2 Sekunden nach dem Loslaufen wurde angenommen, dass alle Beschleunigungseffekte abgeklungen waren und mit einer ebenfalls zwei Sekunden dauernden Messphase begonnen. Die Laufgeschwindigkeit des Roboters wurde dabei durch eine einfache Startpunkt-Endpunkt Messung bestimmt und die Werte der Beschleunigungssensoren für jeden Frame zur Berechnung der Fitness nach Abschluss des Laufes aufgezeichnet. Die aufgezeichneten Werte konnten dabei nicht direkt auf den MemoryStick geschrieben werden, da Schreibzugriffen während des Laufens offenbar eine höhere Priorität, als der Ausführung des Motion-Prozesses zugeteilt wurde und deshalb zu einem stockenden Bewegungsablauf des AIBOs führten. Zur Gewährleistung von möglichst gleichbleibenden Bedingungen wurden die Beinverschalungen der verwendeten Roboter regelmäßig von Schmutz gereinigt und die Getriebe in den Beinen von eingedrungenen Fremdkörpern, wie kleinen Sandkörnern und Ähnlichem, befreit. Um einen negativen Einfluss der mit steigender Anzahl der durchgeführten Evolutionen zu beobachtenden Abnutzung des Teppichs zu minimieren, wurde auf

eine sich ständig ändernde Laufrichtung des Roboters relativ zum Teppich geachtet. Dadurch verteilten sich die Abnutzungserscheinungen auf einen größeren Bereich des Teppichs und verhinderten so ruckartige Geschwindigkeitsänderungen auf Grund sich plötzlich ändernder Bodenhaftung während der Messungen.

Da es in seltenen Fällen vorkommen konnte, dass die CeilingCam auf Grund falsch erkannter Marker eine nicht der Realität entsprechende Position des Roboters erkannte, wurde bei Läufen, deren Geschwindigkeit über dem Durchschnitt der aktuellen Elterngeneration lag, eine zweite Kontrollmessung durchgeführt. Von den beiden so ermittelten Geschwindigkeitswerten wurde stets der Niedrigere verwendet, da zu niedrig bewertete Individuen den Ablauf der Evolution nicht so negativ beeinflussen sollten wie fälschlicherweise zu hoch bewertete Individuen, die als Elemente einer neuen Elterngeneration einen negativen Einfluss auf die neu erzeugten Individuen haben können.

3.4.5 Finden eines Parameterraums geeigneter Größe und Entwicklung einer Fitnessfunktion

Erste Testläufe für reines Vorwärtslaufen, mit aus acht Segmenten bestehenden Polygonen, zeigten selbst nach 60 Generationen kein erkennbares Konvergenzverhalten und die dabei erreichten maximalen Geschwindigkeiten von etwa 33 cm/s waren noch weit von den erwarteten Geschwindigkeiten von über 40 cm/s entfernt.

Eine erste Vermutung für den Grund des Versagens der Evolution war die mit 129 recht hohe Anzahl der Parameter. Um diese Hypothese zu untersuchen, wurden weitere Testläufe mit Polygonen durchgeführt, die nur aus vier Segmenten bestanden, wodurch die Anzahl der Parameter halbiert wurde. Außerdem war durch die Art des zur Realisierung von omnidirektionalem Laufen verwendeten Verfahrens eine Spiegelsymmetrie der linken und rechten Bahnkurven gegeben, was eine weitere Reduktion der Anzahl der Parameter auf 33 ermöglichte. Ein konvergierendes Verhalten war auch hier nicht zu beobachten, allerdings enthielten einige der Generationen Individuen, die eine mit 40 cm/s bis 42 cm/s ungefähr im Zielbereich liegende Geschwindigkeit aufwiesen. Das ließ den Schluss zu, dass auch nur aus vier Segmenten bestehenden Polygonen bereits genug Potential innewohnte, um höhere Geschwindigkeiten als die GT2004WalkingEngine zu erreichen. Also wurde die reduzierte Segmentzahl bei weiteren Tests auf Grund des vorteilhaft kleineren Parameterraums beibehalten.

Bei der nun auf 33 reduzierten Anzahl der Parameter war selbst bei mehr als 50 Generationen keine Konvergenz bei der Evolution beobachtbar. Deshalb wurde die Fitnessfunktion auf mögliche Fehlerquellen untersucht. Um Fehler einer falschen Gewichtung der Werte der Beschleunigungssensoren auszuschließen, wurde die Fitnessfunktion zu Testzwecken auf den Geschwindigkeitswert reduziert. Mit dieser Modifikation war nun eine Konvergenz nach etwa 40 bis 50 Generationen zu beobachten, wobei Laufgeschwindigkeiten von bis zu 43 cm/s erreicht wurden. Durch eine Umstellung der anfänglich verwendeten reinen ,-Strategie des evolutionären Algorithmus auf eine +-Strategie mit einer maximalen Lebensdauer der Eltern von

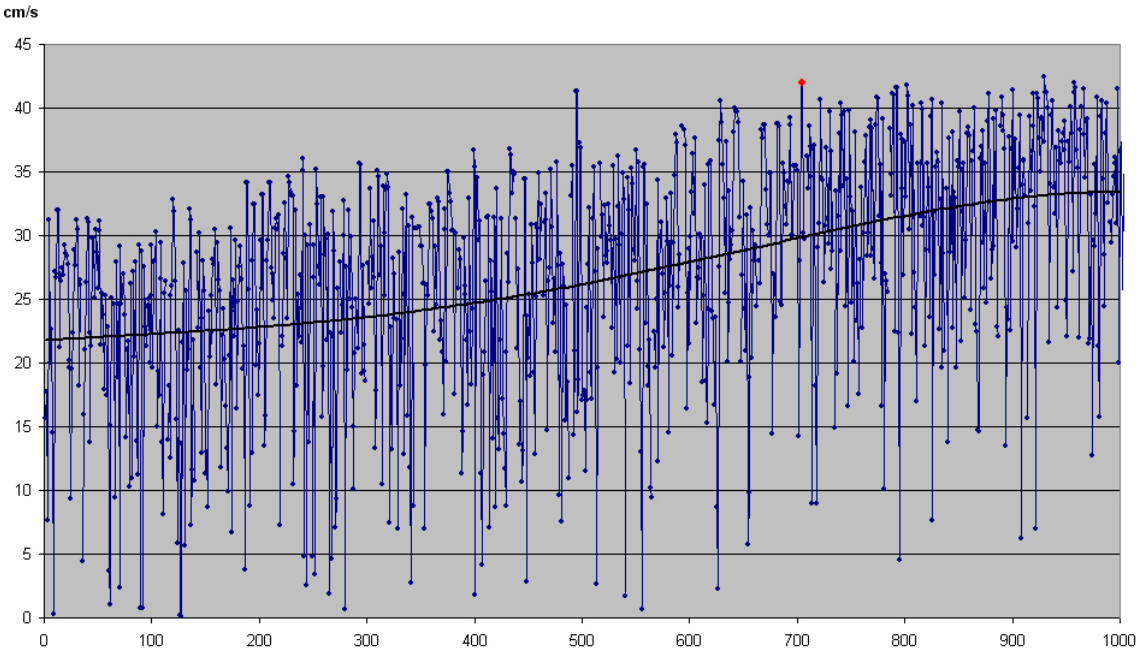


Abbildung 3.6: Bewertete Individuen eines Evolutionsdurchlaufes

zwei Generationen, konnte die Anzahl der zur Konvergenz benötigten Generationen weiter auf 30 bis 40 gesenkt werden.

3.4.6 Manuelle Bewertung im Zielgebiet

Mehrere Veränderungen der Gewichtung der Werte der Beschleunigungssensoren führten nicht zu dem erwünschten Ergebnis im Konvergenzbereich möglichst schnelle und erschütterungsarme Läufe vorzufinden. Stattdessen war entweder kein erkennbarer Effekt der in die Fitnessfunktion integrierten Beschleunigungswerte zu erkennen, oder schnelle Geschwindigkeiten und eine Konvergenz der Evolution blieben aus. Die bei Läufen mit ähnlicher Geschwindigkeit auftretenden Erschütterungen des Kopfes waren allerdings sehr gut mit bloßem Auge im Bezug auf die Laufruhe bewertbar. Deshalb wurde auf eine Integration der Beschleunigungswerte in die Fitnessfunktion verzichtet und im Konvergenzbereich der Evolution eine manuelle Bewertung der Laufruhe durchgeführt. Auf diese Weise konnte bei jeder Evolution eine Auswahl von ausreichend schnellen Individuen im Konvergenzbereich getroffen werden, die auf Grund ihrer Laufruhe potentiell für eine Verwendung im Spiel geeignet waren. Abbildung 3.6 zeigt beispielhaft die sich verbessernde Fitness des Evolutionsdurchlaufes, bei dem der Polygonsatz für schnelles Vorwärtslaufen(hier rot markiert) gewonnen wurde. Gut zu sehen ist die durchschnittlich steigende Fitness und die nach etwa 800 getesteten Individuen bzw. 33 Generationen auftretende Konvergenz. Der manuell selektierte Polygonsatz liegt nicht im Konvergenzbereich und ist auch nicht das schnellste Individuum des Durchlaufs, bietet jedoch das augenscheinlich bestgeeignetste Verhältnis aus Laufruhe und Geschwindigkeit.

Nachdem für alle dafür vorgesehenen WalkRequests eine Auswahl von geeigneten Kandidaten verfügbar war, wurden mögliche Kombinationen in einen Parametersatz geladen und das Verhalten des Roboters über den kompletten Ansteuerungsbereich bewertet. Dazu wurde der Roboter mit Hilfe eines Joysticks angesteuert und überprüft, ob sich in Bereichen, in denen eine starke Überblendung zwischen den spezialisierten Polygonsätzen stattfand, unerwünschte Effekte wie starke Wackelbewegungen oder starkes Rutschen der Beine auf dem Boden und damit eine drastisch reduzierten Geschwindigkeit zeigte. Dabei wurden zunächst die schnellsten der spezialisierten Polygonsätze ausgewählt und beim Auftreten von Problemen solange entsprechende Ersetzungen durchgeführt, bis ein insgesamt als akzeptabel zu bezeichnendes Laufverhalten in allen Richtungen erreicht war.

Bei längeren Lauftests wurde ebenfalls überprüft, ob die ausgewählten Polygonsätze unter Spielbedingungen, auf Grund ungünstiger Ansteuerungen oder zu hoher Beschleunigungen, eine zu hohe Stromaufnahme und damit eine automatische Abschaltung der Roboter verursachen konnten. Als akzeptabel wurden dabei Parametersätze betrachtet, die bei Einsatz eines voll geladenen Akkus mindestens zehn Minuten, also für den Zeitraum einer Halbzeit, absturzfrei liefen. Auch Polygonsätze, die bei Zusammenstößen mit anderen Robotern dazu neigten sich mit den Beinen im Gegner zu verhaken wurden aussortiert.

3.4.7 Ergebnisse

Die bei der Optimierung gewonnenen Polygonsätze ermöglichten in allen Richtungen deutlich höhere Geschwindigkeiten als die GT2004WalkingEngine wie in Abbildung 3.7 zu sehen ist. Der normale Laufmodus erreichte Geschwindigkeiten von 45,1 cm/s vorwärts, 40,5 cm/s rückwärts und 34,4 cm/s seitwärts. Diagonal waren bis zu 42,1 cm/s vorwärts und 43,5 cm/s rückwärts möglich. Die Geschwindigkeit reiner Rotationsbewegungen wurde auf 200 °/s beschränkt, da höhere Geschwindigkeiten auf Grund des dabei auftretenden starken Motionblurs nicht sinnvoll erschienen.

In Abbildung 3.7 ist ebenfalls eine Asymmetrie der Geschwindigkeiten in y-Richtung zu erkennen. Diese Asymmetrie trat bei allen zur Verfügung stehenden Robotern auf und ist, da andere Faktoren wie z.B. unterschiedliche Hardware der Beine, ausgeschlossen werden können, auf die nicht symmetrische Gewichtsverteilung der AIBOs zurückzuführen. Die bei der alten WalkingEngine dadurch auftretenden deutlichen Geschwindigkeitsunterschiede waren bei der neuen Lösung wesentlich schwächer ausgeprägt.

Da die neue WalkingEngine nicht mit Gegenrotationen zur Linearisierung der bei verschiedenen WalkRequests erzeugten Bewegungen arbeitet, treten auf Grund der ungleichen Gewichtsverteilung allerdings ausgeprägtere Rotationskomponenten bei diagonalem Laufen nach hinten auf.

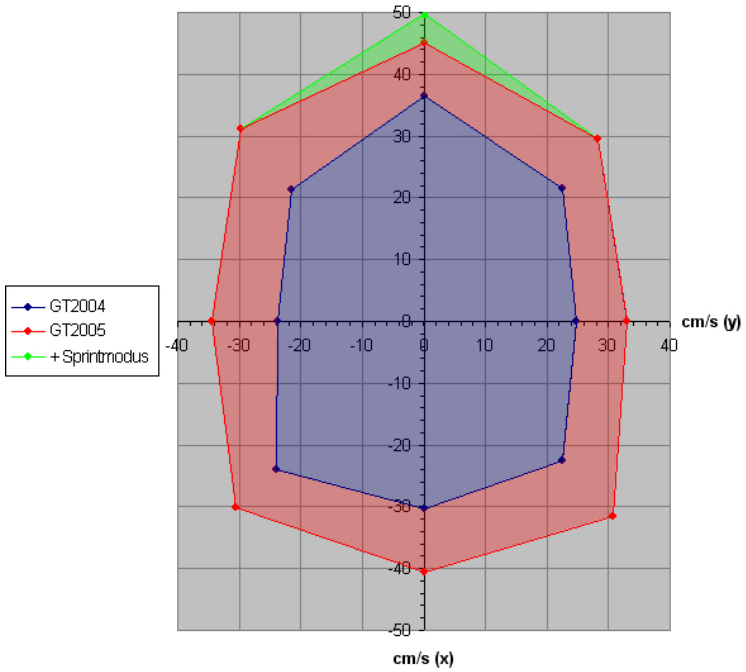


Abbildung 3.7: Vergleich der Laufgeschwindigkeiten GT2004/GT2005

3.4.7.1 Der Sprintmodus

Ein spezieller Polygonsatz, der über einen Lauftyp für sehr schnelles Vorwärtslaufen nutzbar gemacht wurde, wies zwar eine für normales Laufen mit Ball nicht akzeptable Laufruhe und Stabilität auf, war aber mit 50 cm/s der bisher schnellste auf einem AIBO entwickelte Lauf. Bei der Entwicklung dieses Laufes wurden die Polygone mit denen die Evolution initialisiert wurde so geschätzt, dass vermehrt Läufe produziert wurden, bei denen der AIBO, nicht wie mittlerweile bei allen am Robocup teilnehmenden Teams üblich, auf den unteren Verschalungen der Vorderbeine lief, sondern auf den Vorderpfoten. Dadurch wurde auch eine wesentlich höhere Kopfposition beim Laufen erreicht. Die höheren Geschwindigkeiten wurden durch die gestreckten Beine, durch die eine höhere Geschwindigkeit der den Boden berührenden Teile des AIBOs erreicht wurde, möglich. Gleichzeitig verringerte sich durch die längeren Hebel aber die mögliche Beschleunigung. Wegen der längeren Beschleunigungsphase war der Einsatz dieses Lauftyps also erst ab einer gewissen Distanz sinnvoll, besonders weil der verwendete Standardlauf nur etwa 5 cm/s langsamer war.

Der Polygonsatz wurde bei Aktivierung des entsprechenden Lauftyps so in die WalkingEngine integriert, dass bei der Polygonsatzinterpolation der Standardpolygonsatz für schnelles Vorwärtslaufen ersetzt wurde. Auf diese Weise blieben die guten Laufeigenschaften in alle anderen Richtungen beim Setzen dieses Laufmodus erhalten, während bei schnellem Vorwärtslaufen eine 5 cm/s höhere Geschwindigkeit erreicht wurde.

Der evolutionäre Algorithmus hatte ebenfalls einen Polygonsatz, der Geschwindigkeiten bis 53 cm/s ermöglichte erzeugt, jedoch war es bei diesem Polygonsatz nicht mehr möglich den Roboter bei ausreichender Kippsicherheit kontrolliert zu beschleunigen und abzubremsen, so dass eine Verwendung im Spiel nicht möglich war.

3.4.7.2 Laufen mit gegriffenem Ball

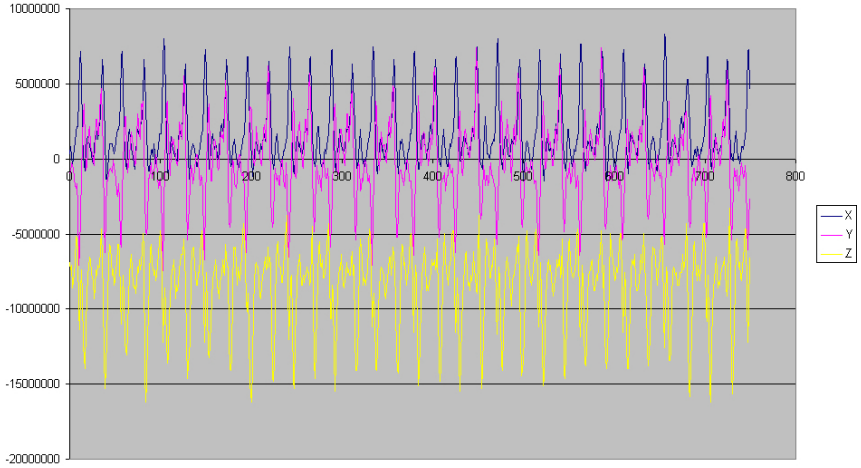
Ein weiterer spezieller Lauftyp wurde für Laufen mit gegriffenem Ball entwickelt. Dabei war eine hohe Schrittfrequenz und eine kurze Schrittlänge entscheidend, um den Ball nicht durch heftige Kollisionen mit den Vorderbeinen wieder zu verlieren und trotzdem eine ausreichende Geschwindigkeit zu erreichen. Für diesen Lauftyp wurde die komplette Polygonsatzinterpolation auf einen Polygonsatz reduziert, wodurch schnelles Laufen in alle Richtungen mit diesem Lauftyp unmöglich wurde. Die Vorteile, schnell mit gegriffenem Ball seitwärts laufen, sowie sich schnell mit und um den Ball drehen zu können, rechtfertigten diese Spezialisierung aber.

3.4.7.3 Laufruhe

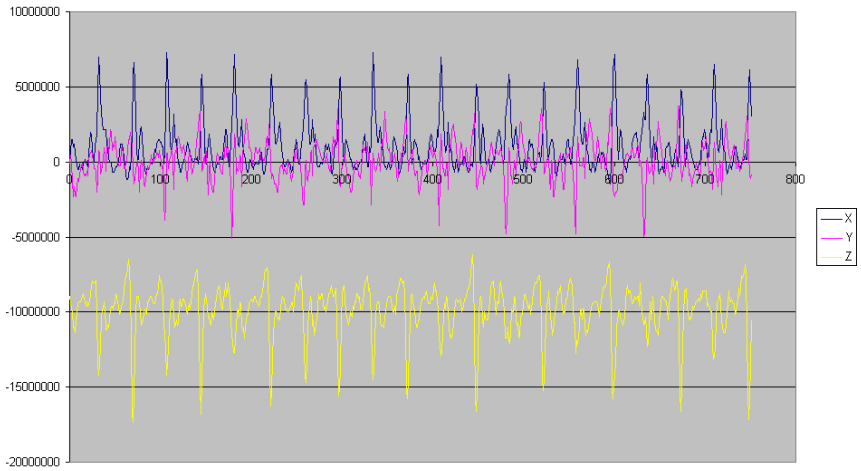
Eine nachträgliche Untersuchung der Werte der Beschleunigungssensoren für die im Einzelnen verwendeten Polygonsätze bestätigte die bei der manuellen Bewertung getroffenen Entscheidungen. Der Polygonsatz für erschütterungsarmes Vorwärtslaufen im mittleren Geschwindigkeitsbereich wies deutlich niedrigere Ausschläge der Beschleunigungssensoren auf, als der Polygonsatz für schnelles Vorwärtslaufen. Der Polygonsatz für den Sprintmodus war ebenfalls sichtbar unruhiger. Gut zu erkennen sind auch die unterschiedlichen Schrittfrequenzen. Ein Vergleich der für jeden Lauf charakteristischen Beschleunigungskurven mit denen eines gegen ein Hindernis laufenden AIBOs zeigte, dass eine Kollisionserkennung auf Basis der Beschleunigungsdaten möglich sein sollte, da insbesondere die y-Werte stark beeinflusst wurden. Da bis zu diesem Zeitpunkt aber nur unzureichende Tests für eine sinnvolle Integration von Kollisionen in das Spielverhalten gemacht wurden, wurde auf eine weitere Verfolgung dieses Ansatzes verzichtet.

3.5 Odometriekalibrierung

Bei den durch den Einsatz evolutionärer Strategien optimierten Polygonsätzen führte weder eine Ansteuerung im externen Ansteuerungsbereich der WalkRequests zu einer übereinstimmenden Bewegung des Roboters, noch war die WalkingEngine mit den bis zu diesem Zeitpunkt gewonnenen Daten in der Lage, genaue Informationen über die tatsächlich ausgeführten Bewegungen zu liefern. Eine gute Qualität der letztgenannten Informationen ist für eine gute Selbstlokalisierung eines sich in Bewegung befindenden Roboters unerlässlich, da diese beim Ausbleiben sonstiger Daten die einzige zuverlässige Informationsquelle über die sich verändernde Position und Orientierung des Roboters sind. Eine exakte Übereinstimmung von angesteuerter

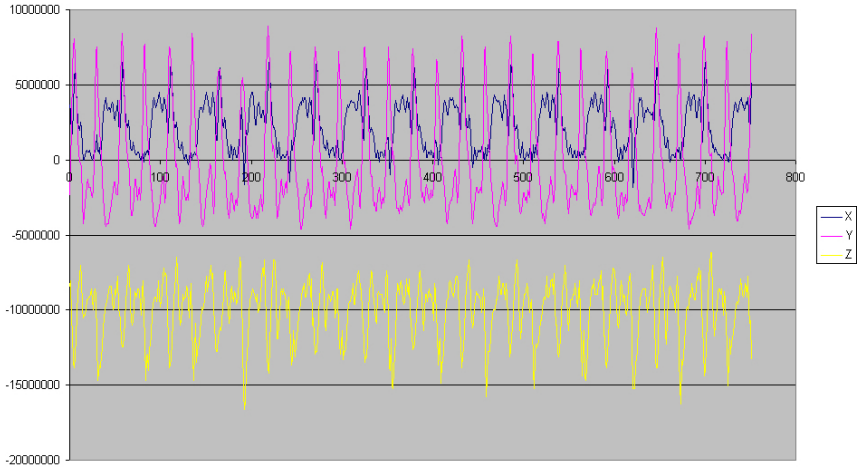


(a) schnelles Vorwärtslaufen

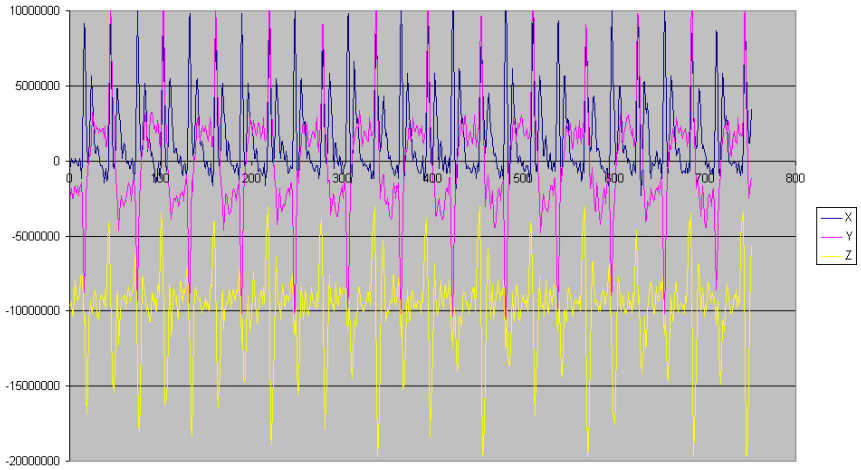


(b) erschütterungsarmes Vorwärtslaufen

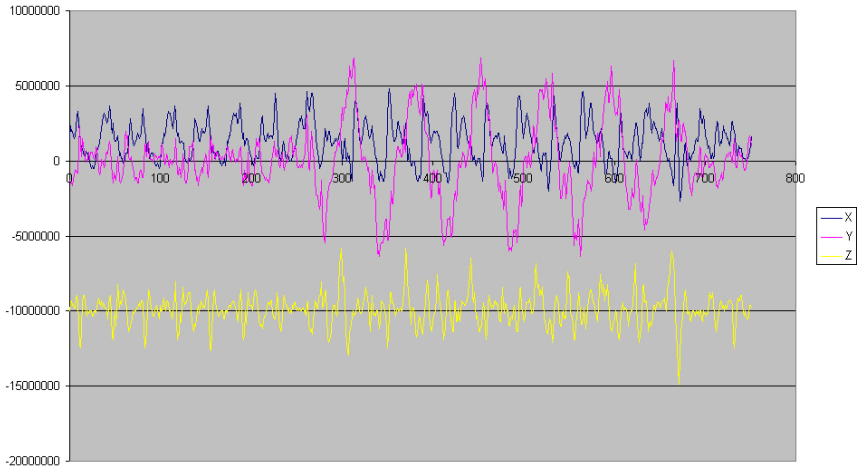
Abbildung 3.8: Werte der x-, y-, und z-Komponente der Beschleunigungssensoren



(a) Seitwärtslaufen



(b) Sprintmodus



(c) schnelles Vorwärtslaufen gegen ein Hindernis

Abbildung 3.9: Werte der x-, y-, und z-Komponente der Beschleunigungssensoren

und ausgeführter Bewegung, ist auf Grund der Tatsache, dass das im GT-Framework verwendete Verhalten praktisch alle Bewegungen des Roboters über positionsabhängig nachregelnde Mechanismen ansteuert von untergeordneter Bedeutung. Deshalb wurde die Dimensionierung des externen Ansteuerungsbereichs der WalkRequests ungefähr den für Translation und Rotation erreichten Maximalgeschwindigkeiten angepasst und nur eine lineare Umrechnung auf den internen Ansteuerungsbereich durchgeführt. Die durch die Nichtlinearität der intern angesteuerten Geschwindigkeiten auftretenden Ungenauigkeiten wurden dabei als vernachlässigbar angesehen.

Für die Realisierung eines Mechanismus, der ausreichend genaue Daten über die tatsächlich ausgeführte Bewegung liefern konnte, ist die in der GT2004WalkingEngine verwendete Methode, nämlich die Kalibrierung über die mit den jeweiligen Bahnkurven erreichbaren Höchstgeschwindigkeiten bei getrenntem Ausmessen der Linearität der Ansteuerung in x- und y-Richtung, unbrauchbar. Das war einerseits auf die komplexe Interpolation zwischen den Polygonsätzen zurückzuführen und andererseits darauf, dass wegen der ebenfalls komplexeren Form der Polygone bei der Berechung der Phase des Laufes, in der ein Bein des Roboters den Boden berührte, ein nicht kalkulierbarer Fehler entstand. Aus diesem Grund war es ebenfalls nicht möglich, die Bewegung des Roboters aus der Bewegung der Beine auf dem Boden zu errechnen. Eine dreidimensionale Kalibrierungstabelle schien deshalb die sinnvollste Lösung zu sein, um unter den gegebenen Umständen Odometrieinformationen mit ausreichender Genauigkeit liefern zu können, zumal in Form der CeilingCam ein gut geeignetes Instrument zur Gewinnung dieser Daten bereitgestellt wurde. Aus den in dieser Kalibrierungstabelle enthaltenen Daten sollten dann für alle möglichen WalkRequests passende Werte durch Interpolation berechnet werden.

Wegen der im Gegensatz zu den Messungen während der Evolutionsläufe aus Translations- und Rotationskomponente bestehenden Bewegungen war eine einfache Startpunkt-Endpunkt Messung hier ungeeignet, da eine eindeutige Trennung von Translation und Rotation ab einer Gesamtdrehung von etwa 180° nicht mehr möglich ist. Aus diesem Grund wurde die Vermessung eines WalkRequests in kleine Messintervalle zerlegt. Nach erfolgter Messung wurde jeweils über alle Messintervalle gemittelt, wobei die vier extremsten Messwerte von der Betrachtung ausgeschlossen wurden, um Ausreißern vorzubeugen. Die Mittelung über viele Messwerte sollte außerdem den Einfluss der zahlreichen Fehlerfaktoren so klein wie möglich halten. Dazu zählten neben den Messungenauigkeiten der CeilingCam auch Effekte, die durch eine schwankende Roboterbewegung, eine ungleichmäßige Bodenhaftung oder Resonanzeffekte während des Laufens entstanden. Im Gegensatz zu den letztgenannten Fehlerquellen waren die durch die CeilingCam verursachten Messfehler vernachlässigbar klein. Da der Einfluss der Fehlerfaktoren bei jeder einzelnen Messung schon durch die Verwendung anderer, interpolierter Polygone und einer anderen Laufbahn auf dem Teppich unterschiedlich ausfiel, ist es nicht möglich genaue Angaben über die Messungenauigkeiten des gesamten Verfahrens zu machen. Bei Wiederholungsmessungen mit gleicher Ansteuerung und identischen Polygonen waren allerdings maximale Abweichungen von 0,4 cm/s für die Translation und 1,5 °/s für die Rotation zu beobachten, wobei die durchschnittlichen Fehler deutlich darunter lagen. Die

Unterschiede zwischen den einzelnen Werten für die Messintervalle waren systembedingt größer, da beim Setzen der Beine naturgemäß kleine Beschleunigungsschübe entstehen, die eine Periodizität in der Schrittfrequenz aufweisen und deshalb zu einer periodischen Geschwindigkeitsänderung bei den Intervallmessungen führen.

Die verwendete Intervalllänge von 400ms wurde dabei so kurz gewählt, dass auch bei Läufen mit hoher Geschwindigkeit, bei denen sich Roboter nur verhältnismäßig kurz im Aufnahmebereich der CeillingCam befindet, eine ausreichende Anzahl von Intervallen vermessen werden konnte. Eine kürzere Intervalllänge war ebenfalls nicht möglich, weil dabei mitunter starke Messfehler auftraten, die mit der Bildrate der CeilingCam im Zusammenhang zu stehen schienen. Die Gesamtmessdauer wurde hingegen so groß wie möglich gewählt, um eine optimale Ausnutzung des unter der CeilingCam verfügbaren Platzes zu gewährleisten. Erschwerend kam hinzu, dass stets eine Anlaufphase von zwei Sekunden benötigt wurde, um sicherzustellen, dass sich der Roboter im gesamten Zeitraum der Messung mit der für die angesteuerten Werte charakteristischen Geschwindigkeit bewegte. Auf diese Weise konnte auch ein negativer Einfluss von Beschleunigungsphänomenen auf die Messung minimiert werden.

Zur Messung der Kalibrierungstabellen wurde ein Raster über den internen Ansteuerungsbereich der WalkRequests gelegt, das in Bereichen häufig verwendeter Ansteuerungen eine leicht erhöhte Dichte aufwies. Die Ausmessung dieser dreidimensionalen Tabelle wurde ebenenweise vorgenommen, d.h. es wurden bei konstant gehaltener Rotation in einem Messdurchgang jeweils alle in der Tabelle enthaltenen Translationskombinationen ausgemessen. Die Anzahl von 195 Messpunkten für jede dieser Translationsebenen war so dimensioniert worden, dass ein Messdurchlauf zur Erfassung dieser Daten mit einer einzigen Akkuladung eines AIBOs durchgeführt werden konnte.

Die Anzahl der nötigen Translationsebenen, also die Auflösung für die Rotationskomponente, wurde experimentell ermittelt. Da das entwickelte Odometrieinterpolationssystem dynamisch um Translationsebenen an beliebigen Stellen erweitert werden konnte, war es möglich, zunächst eine große Rasterweite für die Rotation zu wählen und das Raster dann dort zu verfeinern, wo bei Kontrollmessungen eine noch nicht ausreichend genaue Odometrie sichtbar wurde. Die dazu nötigen Kontrollmessungen wurden durch die Ansteuerung des Roboters per Joystick und manuelle Bewertung der Abweichungen zwischen realer und auf Basis der Odometrie berechneter Position durchgeführt. Dazu wurden die beiden berechneten Positionen auf dem Bildschirm visualisiert und die Abweichungen bei verschiedenen Bewegungsabfolgen kontrolliert. Dabei stellten sich 7 Translationsebenen als ausreichend heraus, um über den gesamten Rotationsbereich eine ausreichende Genauigkeit zu gewährleisten. Für reine Rotationsbewegungen wurde die Auflösung durch eine weitere Tabelle weiter auf insgesamt 21 Messwerte erhöht, da eine noch höhere Genauigkeit, speziell bei langsamen Rotationsbewegungen zum Anvisieren von Winkeln, vor Torschüssen sinnvoll erschien.

Für die zwei speziellen Lauftypen wurden ebenfalls zusätzliche Messungen durchgeführt. Diese Messungen wurden, um die Hardware zu schonen, nur für die für den

jeweiligen Einsatzzweck wichtigen WalkRequests durchgeführt. Für den Sprintmodus also für maximal schnelles Vorwärtslaufen und für das Laufen mit gegriffenem Ball für die Rotationsbewegungen mit und um den Ball.

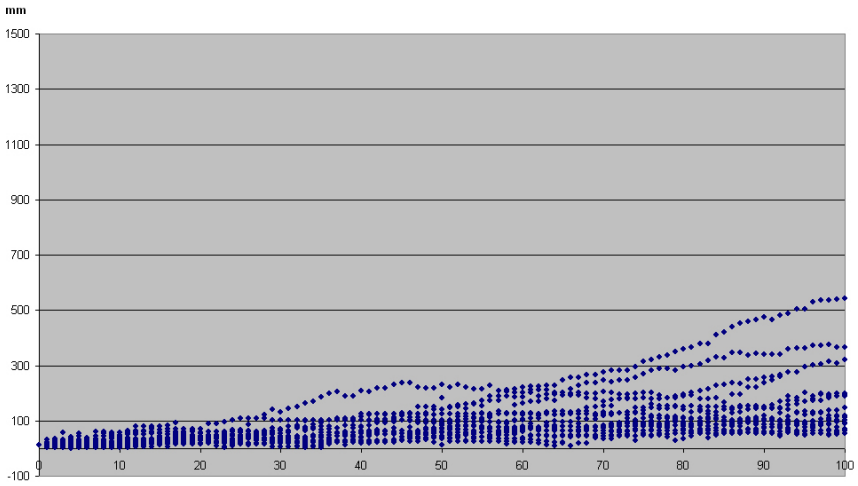
3.5.1 Teppich- und Roboterabhängigkeiten

Bei Odometriemessungen auf unterschiedlichen Teppichen waren bei der letztendlich verwendeten Kombination aus Polygonsätzen nur minimale Abweichungen in den erreichbaren Maximalgeschwindigkeiten für alle Richtungen zu beobachten. Insbesondere fiel auf, dass sich die Richtung der Abweichungen im Ansteuerungsraum praktisch nicht änderte, sondern nur der Betrag der Abweichungen. Es schien also möglich zu sein, einen Roboter durch manuelles Ausmessen weniger Kontrollpunkte ohne Zuhilfenahme der CeilingCam bei linearer Skalierung der Odometrietabelle eine Kalibrierung für verschiedene Untergrundtypen durchzuführen. Bei einigen der in früheren Entwicklungsstadien verwendeten Polygonsätze hatte sich allerdings gezeigt, dass Polygonsätze durchaus ein sehr unterschiedliches Verhalten auf verschiedenen Teppichen zeigen können. Deshalb sollte vor der Anwendung des oben beschriebenen Verfahrens überprüft werden, ob die eingesetzten Polygonsätze eine ausreichende Teppichinvarianz zeigen. Ist dies nicht der Fall, oder steht ein ausreichend großes Muster des zu verwendenden Teppichs zur Verfügung, ist eine komplette Neuvermessung der Odometrie mit Hilfe der CeilingCam oft die bessere Alternative.

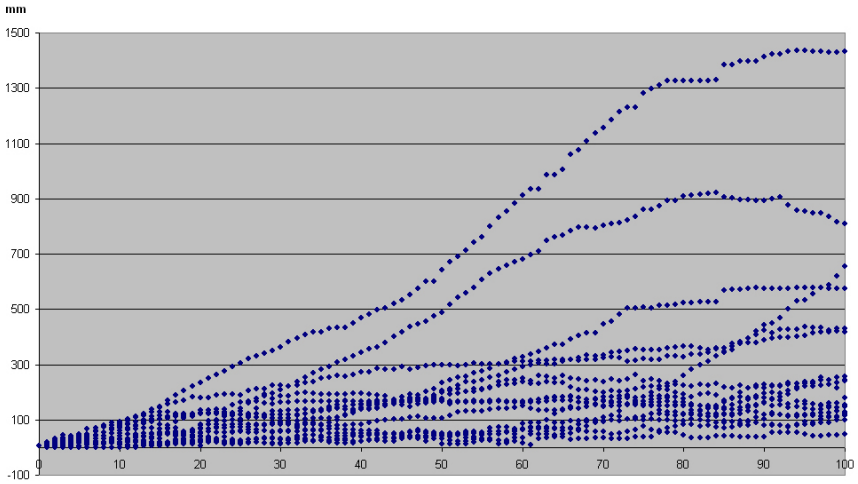
Eine Vermessung der Odometrie auf unterschiedlichen Robotern ergab Messwertabweichunge der gleichen Größenordnung wie bei unterschiedlichen Teppichen. Die Gründe für diese Abweichungen konnten in unterschiedlich abgenutzten Beinverschalungen der Roboter ausgemacht werden, durch die sich die Bodenhaftung offenbar entscheidend änderte. Außerdem schienen auch die sich bei längerem Laufen der Roboter auf dem Teppich auf den Beinverschalungen bildenden Ablagerungen einen nicht zu vernachlässigenden Einfluss auf die Laufeigenschaften zu haben. Der störende Einfluss dieser vermutlich aus Faserbeschichtungskomponenten des Teppichs bestehenden Ablagerungen wurde dadurch minimiert, dass die Roboter in regelmäßigen Abständen gereinigt wurden. Auch bei gereinigten Robotern mit identisch abgenutzten Beinverschalungen war ein Unterschied in der Odometrie, der höchstwahrscheinlich durch eine unterschiedliche Abnutzung der restlichen Hardware verursacht wurde, auszumachen. Da sich deren Abnutzungsgrad, insbesondere der der Gelenke, ständig änderte, war von einer sich ebenfalls mit der Zeit ändernden Odometrie der Roboter auszugehen.

Eine getrennte Messung der Odometrie für jeden Roboter war aus Zeitgründen, und um die Roboter nicht einer noch höheren Beanspruchung auszusetzen, nicht möglich. Deshalb wurden die wichtigsten Translationsebenen auf zwei verschiedenen Robotern unterschiedlicher Abnutzung ausgemessen und zwischen diesen Messwerten gemittelt. Die so gewonnenen Odometriedaten zeigten auf allen Robotern eine akzeptable Qualität, die der der alten Lösung ebenbürtig oder überlegen war.

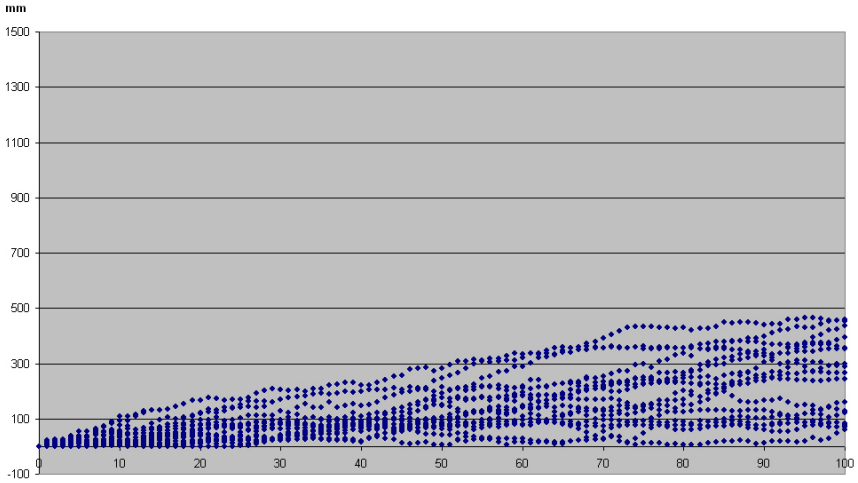
Für diverse Aktionen im Spiel, wie Präzisionsschüsse, war eine sehr genaue Odo-



(a) GT2004 WalkingEngine



(b) GT2005 WalkingEngine



(c) GT2005 WalkingEngine mit Beschleunigungsbeschränkung

Abbildung 3.10: Varianz der Positionsberechnung auf Odometriebasis im Spielbetrieb

metrie bei reinen Rotationsbewegungen von Vorteil. Da die roboterspezifischen Unterschiede bei reinen Rotationen am größten waren, wurde eine separate Ausmessung der Rotationsodometrie vorgenommen. Die dabei für die Roboter entstehende, zusätzliche Beanspruchung war bei 21 zusätzlichen Messwerten minimal, so dass für alle Roboter eine Zusatzmessung durchgeführt werden konnte. Zur Integration dieser roboterspezifischen Messdaten wurde ein System entworfen, das abhängig von den MAC-Adressen der WLAN-Karten Roboter das Vorhandensein von speziellen Odometriedaten auf dem MemoryStick kontrollierte und andernfalls die durch Mittelung gewonnene Standardodometrie verwendete. Dabei konnten sowohl spezialisierte Rotationsdaten, als auch spezialisierte Translationsebenen für jeden Roboter gespeichert werden, auch wenn letztere Möglichkeit von uns nicht genutzt wurde.

Abbildung 3.10 zeigt die Genauigkeit der neuen Odometriekalibrierung im Vergleich zur alten Lösung der GT2004 WalkingEngine. Zu sehen sind Positionsabweichungen zwischen realer und auf Basis der Odometrie berechneter Roboterposition in einem Bereich von 100 Frames(1 Frame entspricht 8ms) nach einer Positionssynchronisierung. Um die auftretenden Abweichungen nicht durch eine falsch konstruierte Testsituation zu verfälschen, wurden die Messungen auf einem Roboter im Spielbetrieb, d.h. mit aktiviertem Fussballverhalten durchgeführt.

Die einzelnen Roboter unterscheiden sich, aufgrund unterschiedlicher Hardwareabnutzung, im Bezug auf die bei einem bestimmten WalkRequest ausgeführte Bewegung. Die wesentlich höhere Anzahl der zur Kalibrierung nötigen Messwerte bei der neuen Lösung, führt dabei auch zu einer höheren Spezialisierung auf den zur Kalibrierung eingesetzten Roboter. Um die Spezialisierung auf einen Roboter zu minimieren, wurden Odometriemessungen auf zwei verschiedenen Robotern durchgeführt und zwischen den Ergebnissen gemittelt. Daraus resultiert bei identischen Beschleunigungsbegrenzungen eine höhere Varianz der Odometrie wie in Abbildung 3.10(a) und Abbildung 3.10(c) zu sehen ist.

Erlaubt man wesentlich höhere Beschleunigungen, steigen die Odometrieabweichungen in Phasen starker Beschleunigung deutlich an(siehe Abbildung 3.10(b)). Die meisten Messwerte bleiben dabei jedoch innerhalb akzeptabler Grenzen, so dass eine Erhöhung der Beschleunigungsfähigkeit insgesammt einen positiven Effekt hat.

3.5.2 Unterschiede zwischen ansteuerten und realen Bahnkurven

Beim Vergleich markanter Punkte der angesteuerten und realen Bahnkurven fiel ein Phasenversatz von 80 ms bis 100 ms zwischen den Datenreihen auf. Der entdeckte Phasenversatz bewegt sich in der Größenordnung von 11-12 Durchläufen der WalkingEngine und ist auf hardwarebedingte Verzögerungen wie die Trägheit der Ansteuerung der Beinmotoren zurückzuführen.

Ebenfalls auffällig war, dass die real durchlaufenen Bahnkurven, wie in Abbildung 3.11(a) zu sehen ist, nicht mit den angesteuerten Kurven übereinstimmten. Dabei war die Ähnlichkeit der Bahnkurven belastungs- und geschwindigkeitsabhängig. Bei Laufbewegungen ohne Bodenkontakt und bei langsamerer Ansteuerung nahm die

Ähnlichkeit stark zu (siehe Abbildung 3.11(b)). Das ließ den Schluss zu, dass bei der Berechnung der Ansteuerungsdaten der Motoren durch den AIBO nicht die Masse der Beine und die bei steigender Geschwindigkeit verstärkt auftretenden Trägheitseffekte berücksichtigt wurden. Basierend auf dieser Beobachtung wurde ein Ansatz verfolgt, bei dem die real duchlaufenen Bahnkurven ausgelesen und der AIBO mit diesen Daten wieder angesteuert wurde, um die realen und die angesteuerten Bahnkurven einander anzunähern. Durch dieses Verfahren sollte eine energieeffizientere Ansteuerung und somit eine möglicherweise weniger starke Beanspruchung der Hardware entstehen. Ebenfalls denkbar war eine Verwendung der Ähnlichkeit und damit der Energieeffizienz bei der Bewertung der Individuen in den Evolutionsläufen. Bei Experimenten mit mehreren Bahnkurven zeigte sich allerdings, dass bei der Ansteuerung mit ausgelesenen Bahnkurvendaten Bewegungen ausgeführt wurden, die wiederum von den angesteuerten Werten abwichen (siehe Abbildung 3.11(c)). Außerdem war bei der Verwendung von ausgelesenen Bahnkurvendaten eine erhebliche Verschlechterung der Laufleistung zu beobachten. Bei einigen Testpolygonsätzen war sogar gar keine Forbewegung mehr möglich. Auf Grund dieser Ergebnisse erschien eine weitere Verwendung solcher ausgelesener Bahnkurven zur Optimierung von Polygonsätzen nicht sinnvoll.

3.6 Datenanalyse und Tools

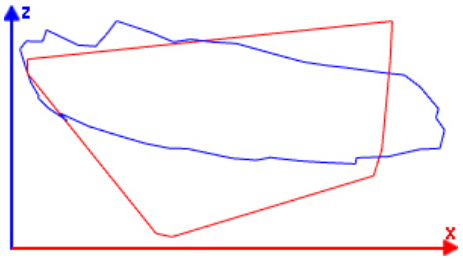
Zur Visualisierung und manuellen Kontrolle der Polygon- und Odometriedaten wurde ein spezielles Tool entwickelt, das drei verschiedene Datenmodi unterstützt.

3.6.1 Der Bahnkurvenmodus

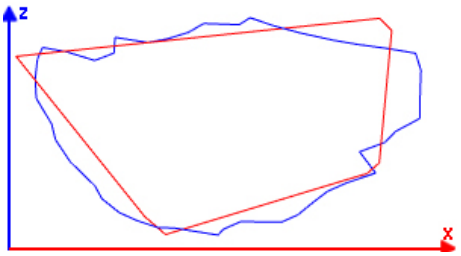
Im ersten Datenmodus können die im WalkingEngine eigenen Datenformat vorliegenden Polygonsätze in einer frei drehbaren 3D-Visualisierung eines sich passend bewegenden AIBOs dargestellt werden. Dieser Datemmodus erlaubt es, die Form der angesteuerten Bahnkurven im Kontext der beobachteten Laufeigenschaften zu betrachten und auf diese Weise bessere Startpolygonsätze für zukünftige Evolutionsdurchläufe schätzen zu können.

3.6.2 Der Debugdatenmodus

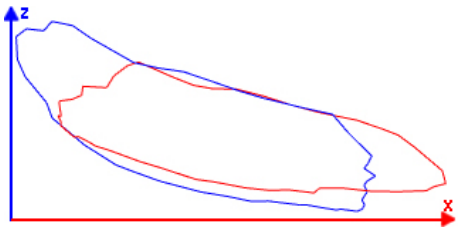
Der zweite Datenmodus erlaubt in derselben Visualisierungsumgebung die gleichzeitige Darstellung der angesteuerten Bahnkurven und der real ausgeführten Bewegungen. Zur Erzeugung der dafür nötigen Daten ist die WalkingEngine mit einem speziellen Debugmodus ausgestattet, nach dessen Aktivierung Ansteuerungs- und Gelenkdaten einer kompletten Schrittfolge aufgezeichnet und in einem für das Tool lesbaren Format abgespeichert werden. Dabei müssen die real durchlaufenen Bahnkurven erst aus den aufgezeichneten Gelenkdaten kinematisch errechnet werden und alle von der WalkingEngine auf den Polygonen durchgeführten Transformationen invertiert werden.



(a) schnelles Vorwärtslaufen auf dem Boden



(b) schnelles Vorwärtslaufen in der Luft



(c) versuchtes Vorwärtslaufen mit wieder ausgelesenen Bahnkurven

Abbildung 3.11: Angesteuerte (rot) und reale (blau) Bahnkurven

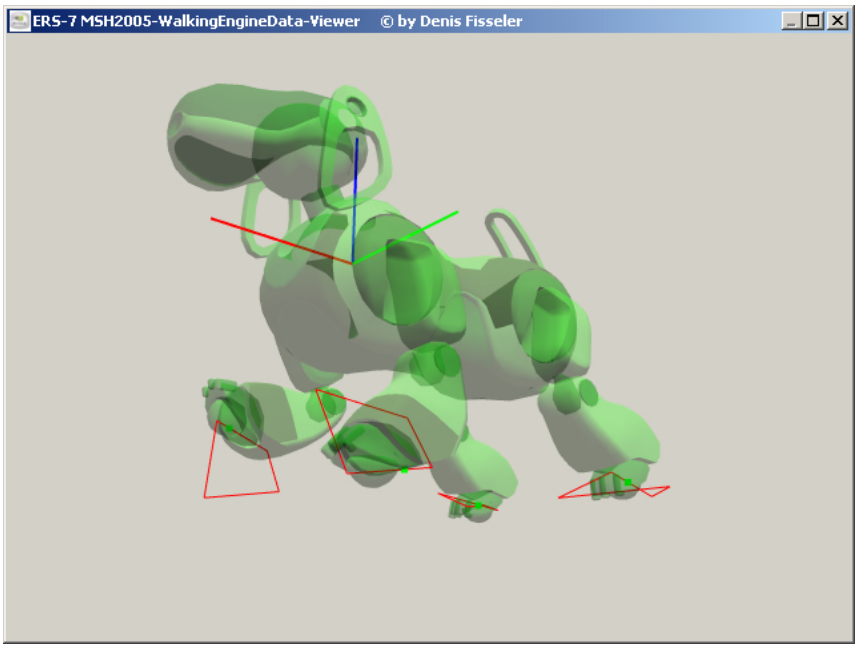


Abbildung 3.12: ERS-7 MSH2005 WalkingEngineData-Viewer (Polygonmodus)

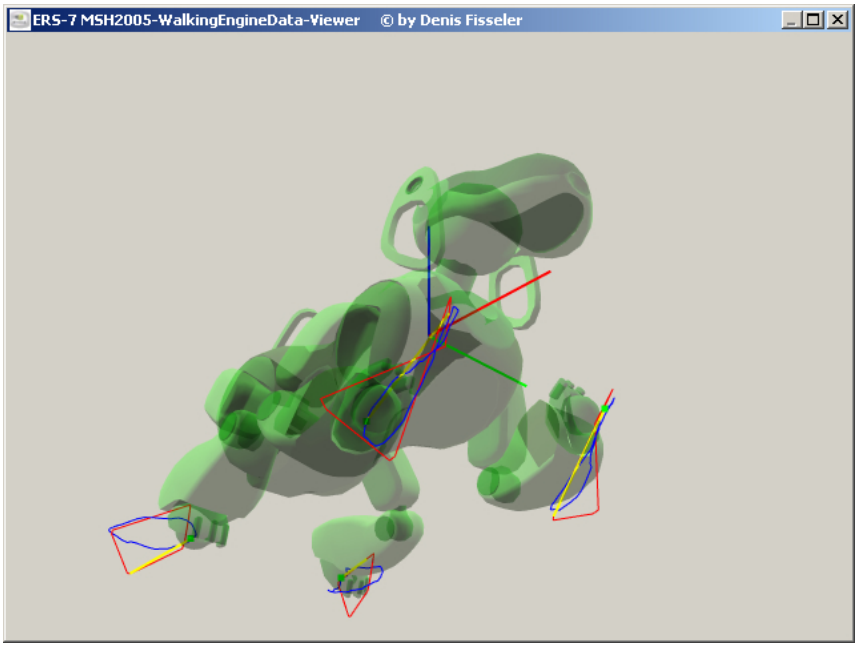
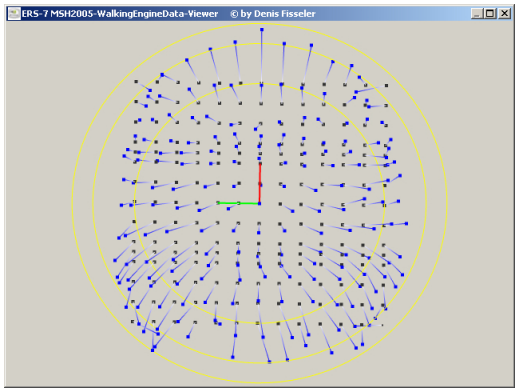


Abbildung 3.13: ERS-7 MSH2005 WalkingEngineData-Viewer (Debugdatenmodus)

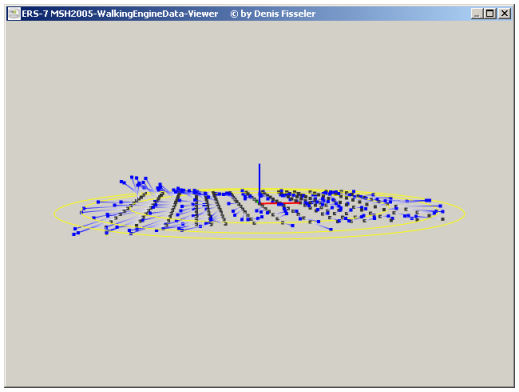
3.6.3 Der Odometriemodus

Ein dritter Datenmodus stellt die zur Berechnung der Odometrie verwendeten Kalibrierungsdaten als dreidimensionales Vektorfeld dar. Die Darstellung wurde aus Gründen der Übersichtlichkeit auf jeweils eine Translationsebene beschränkt, jedoch mit der Möglichkeit, schnell zwischen verschiedenen Ebenen umzuschalten. Es wird jeweils die Abweichung zwischen angesteuertem und real ausgeführtem WalkRequest als Vektor dargestellt, wobei die Rotationskomponente in z-Richtung aufgetragen wird. Dabei bilden die dunklen Punkte das Raster des internen Ansteuerungsbereiches der WalkingEngine. Die blauen Punkte zeigen dementsprechend die mit der CeilingCam gemessenen, real erreichten Werte. Als visuelles Hilfsmittel wurden drei Kreise für die Geschwindigkeitsbeträge 300 cm/s, 400 cm/s und 450 cm/s eingezeichet.

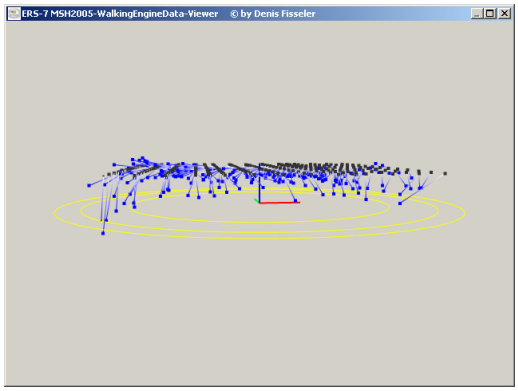
Dabei fällt auf, dass bei zunehmend größeren Rotationskomponenten ebenfalls die durchschnittliche Abweichung steigt, weil große Rotationskomponenten die gleichzeitig ausführbaren Translationen erheblich einschränken.



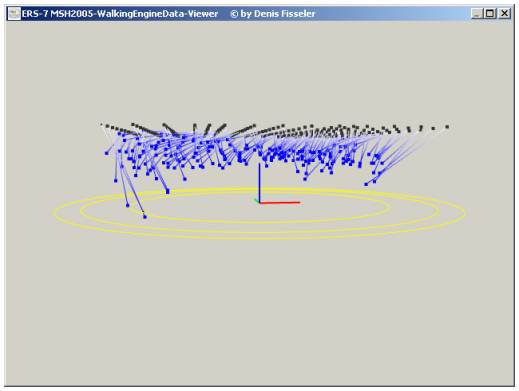
(a) Translationsebene ohne Rotationskomponente von oben



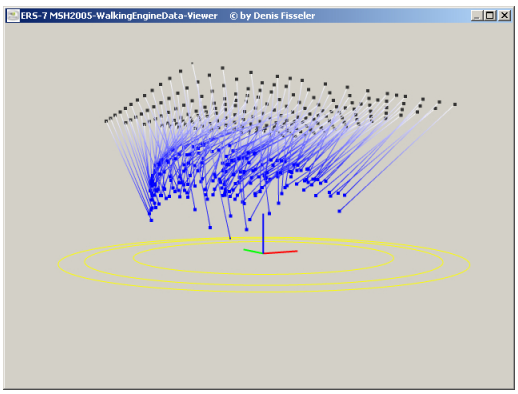
(b) Translationsebene ohne Rotationskomponente seitlich



(c) Translationsebene mit Rotationskomponente 50 seitlich



(d) Translationsebene mit Rotationskomponente 100 seitlich



(e) Translationsebene mit Rotationskomponente 200 seitlich

Abbildung 3.14: Darstellung unterschiedlicher Translationebenen im Odometriemodus

4 BallLocator

Eine der wichtigsten Informationen für einen Fußballspieler ist es zu wissen, wo sich der Ball befindet. Aufgabe des BallLocators ist es, diese Information aus den Daten des ImageProcessors zu berechnen.

4.1 Problembeschreibung

Der ImageProcessor berechnet so genannte *BallPercepte*. Das heißt, er analysiert das Bild der Kamera des Roboters und versucht in dem Bild Bälle zu finden. Dabei kann es vorkommen, dass in einem oder mehreren Bildern kein Ball erkannt wird. Es kann allerdings auch vorkommen, dass auf Grund von beispielsweise orangefarbenden Kleidungsstücken im Publikum mehrere bzw. falsche Bälle erkannt werden.

Die Aufgabe des BallLocators ist es, aus den möglicherweise fehlerbehafteten Informationen des ImageProcessors möglichst exakt die Position und Geschwindigkeit des Balls zu berechnen. Neben den schon erwähnten Problemen muss der BallLocator außerdem damit zurecht kommen, dass – selbst bei einem sich nicht bewegenden Roboter und einem sich nicht bewegendem Ball – die *BallPercepte* leicht springen. Dies ist vor allem für die Geschwindigkeitsberechnung ein großes Problem. Sobald sich der Roboter bewegt, wird dieser Effekt noch verstärkt.

4.2 Ansätze zur Lösung des Problems

Es wurden verschiedene Lösungsansätze für das Problem diskutiert, die im Folgenden näher erläutert werden.

4.2.1 Kalman-Filter

Ein Kalman-Filter (siehe Kapitel 6.2.1) ist ein iterativer Schätzer, der den nächsten Zustand eines dynamischen Systems mit einem linearen Prozessmodell berechnet [22]. Es wird die Annahme gemacht, dass das System mit gaußverteiltem Rauschen behaftet ist.

Der GT2004BallLocator implementiert einen Kalman-Filter [16]. Dabei wird der nächste Zustand in zwei Schritten berechnet. Im ersten Schritt (*TimeUpdate*) wird anhand der bisherigen Zustände der nächste Zustand vorhergesagt. Der zweite Schritt (*MeasurementUpdate*) wird nur ausgeführt, wenn ein *BallPercept* vorliegt. Anhand des Perzepts wird der vorhergesagte Zustand verbessert und verifiziert.

4.2.2 Partikel-Filter (Sequenzielle Monte-Carlo-Methode)

Partikel-Filter schätzen ebenso wie Kalman-Filter einen Zustand in einem dynamischen System [2]. Das Prozessmodell muss jedoch nicht linear sein. Außerdem wird keine konkrete Annahme über die Art des Rauschens getroffen.

Die Wahrscheinlichkeitsverteilung des aktuellen Systemzustands wird durch eine Menge von Partikeln dargestellt. Ein Partikel repräsentiert eine mögliche Ballposition und -geschwindigkeit.

Es gibt einerseits die Möglichkeit, mit einer konstanten Anzahl von Partikeln zu arbeiten und jedem Partikel eine Wahrscheinlichkeit zuzuordnen. Andererseits kann sich die Anzahl der Partikel auch dynamisch ändern, so dass die Wahrscheinlichkeitsdichte des zugehörigen Zustandsraums durch die jeweilige Partikeldichte bestimmt wird.

Da mehrere Ballpositionen auf einmal angenommen werden, können Partikel-Filter im Gegensatz zum Kalman-Filter auch mit mehrdeutigen Situationen umgehen, da verschiedene Kandidaten für Ballpositionen in Betracht gezogen werden. Dies kann zum Beispiel durch falsche Perzepte geschehen. Außerdem bieten Partikel-Filter die Möglichkeit mehr als ein *BallPercept* pro Durchlauf (siehe Kapitel 7.1.2) zu verarbeiten.

4.2.3 RAO-Blackwellised-Particle-Filter

Dieter Fox und Cody C. T. Kwok stellten 2004 im Rahmen des International RoboCup Symposiums ihren Ansatz des RAO-Blackwellised-Particle-Filters vor [9]. Die Idee dabei ist, die Vorteile eines Partikel-Filter-Systems mit denen eines Kalman-Filters zu vereinen. Der Ansatz wird auch von den „UW Huskies“[1] benutzt und hat auf Turnieren sehr gute Ergebnisse gezeigt.

Basis des RAO-Blackwellised-Particle-Filters ist ein Partikel-Filter-System, das die Nichlinearität des zu Grunde liegenden Systems repräsentiert. Zusätzlich wird jedem einzelnen Partikel ein Kalman-Filter zugeordnet, der die Position und Geschwindigkeit für das Partikel schätzt.

Um schneller auf Interaktionen mit der Umgebung reagieren zu können, wurde eine Zustandsmaschine entwickelt. In dieser werden für Zustände wie „ein Schuss wird ausgeführt“ oder „der Ball wird vom Roboter gegriffen“ zusätzliche Informationen in das Partikel-Filter-System mit einbezogen. Zum Beispiel werden die Partikel bei einem Schuss abhängig von der erwarteten Schussrichtung verschoben. Beim Greifen des Balls wird die Ballposition in der Nähe der Roboterposition angenommen.

[1] `http://www.cs.washington.edu/ai/Mobile_Robotics/Aibo/`

4.3 Beschreibung der gewählten Lösung (GT2005BallLocator)

Das zu Grunde liegende System für die Ballmodellierung, das heißt die Bewegung des Balls, ist nicht linear. Dies liegt zum Einen daran, dass die Bewegung des Balls auf dem Teppich gebremst wird. Zum Anderen kann der Ball durch Interaktion mit seiner Umgebung plötzlich seine Bewegungsrichtung oder Position ändern, zum Beispiel durch einen Schuss, einem Einwurf oder einer Kollision mit einem Hindernis. Der GT2004BallLocator hat demzufolge ein idealisiertes Modell verwendet.

Aus diesem Grund haben wir uns dafür entschieden, für die Positionsberechnungen einen neuen Ansatz auf Basis eines Partikelsystems zu entwickeln. Da der RAO-Blackwellised-Particle-Filter je nach Anzahl der Partikel einen relativ hohen Rechenaufwand erfordert, verzichten wir auf die zusätzlichen Kalman-Filter.

Die Berechnung der Ballgeschwindigkeit wird in jeder Iteration direkt auf Basis der letzten Perzepte durchgeführt. Hierfür wurde ein Ansatz (vergleiche [11]) aus der Middle-Size-League des RoboCups an unsere Bedürfnisse angepasst.

Position und Geschwindigkeit werden in Roboterkoordinaten gespeichert, damit Fehler in der Selbstlokalisierung keinen Einfluss auf die Güte der berechneten Ballposition haben. Dadurch wird die Genauigkeit der Ballposition jedoch abhängig von der Odometrie.

4.3.1 Positionsberechnung

Das Partikelsystem wurde an die speziellen Bedürfnisse der Ballmodellierung angepasst:

- Position und Geschwindigkeit können sich schnell ändern.
- Sensor Aliasing[2] ist in den meisten Situationen vernachlässigbar klein.
- Das Sensorrauschen ist abhängig von verschiedenen Faktoren hoch (zum Beispiel springende *BallPercepte* bei einem weit entfernten Ball), kann aber klein sein, besonders, wenn sich der Ball nah am Roboter befindet und der Roboter sich nicht bewegt.

Besonderer Wert wurde bei der Entwicklung auf eine gute Reaktivität gelegt. Außerdem sollte verhindert werden, dass viel Zeit für das Aktualisieren der Partikel verwendet wird, die sich weit weg von der realen Situation befinden und somit eine geringe Validität haben („filter degeneracy“). Dies kann zum Beispiel bei plötzlichen Positions- oder Bewegungsrichtungsänderungen passieren.

Der BallLocator arbeitet mit einer konstanten Anzahl von Partikeln. Eine Menge von 40 Partikeln hat sich in Experimenten als genügend groß herausgestellt. Eine kleinere Anzahl geht auf Kosten der Genauigkeit, eine größere Anzahl erhöht die

[2] Sensor-Daten sind nicht eindeutig, das heißt die Sensordaten können bei mehreren verschiedenen Situationen bzw. Umgebungszuständen dieselben sein.

Rechenzeit, ergibt jedoch keine signifikant besseren Ergebnisse. Für jedes Partikel wird die Position, sowie die zugehörige Validität berechnet.

Die Positionsberechnung ist in vier Schritte unterteilt. Der erste Schritt ist das *TimeUpdate*, welches immer ausgeführt wird und die Partikelposition anhand der Odometriedaten und der Geschwindigkeit ändert, sowie die Validität dekrementiert. Danach wird ein *MeasurementUpdate* durchgeführt – allerdings nur, wenn ein *Ball-Percept* vorliegt. Anschließend muss aus den Partikeln noch eine Gesamtposition samt Validität berechnet werden. Zum Schluss werden Partikel, welche eine vernachlässigbar kleine Validität haben, in die Nähe des letzten Perzepts gestreut (*Sensor-Resetting* [12]).

Im Folgenden bezeichnet:

- $\vec{s_i}$ die Partikelposition und $\mathrm{P}(s_i)$ die Validität des i-ten Partikels
- $\vec{s_i}'$ und $\mathrm{P}(s_i')$ die jeweiligen Werte aus dem vorherigen Durchlauf
- $\overrightarrow{ball}_{pos}$ die Gesamtposition und $\mathrm{P}(ball_{pos})$ die Validität
- ℓ die Anzahl der Partikel

4.3.1.1 TimeUpdate

Mit dem *TimeUpdate* wird der neue Partikelzustand abhängig von Roboter- und Ballbewegung propagiert. Als erstes wird die Partikelwolke anhand der Odometrie-Daten verschoben. Dies ist notwendig, da die Positionen relativ zum Roboter gespeichert werden. Sei $\vec{t}$ die Translationsänderung und α die Rotationsänderung des Roboters.

$$\vec{s_i} = \begin{pmatrix} \cos\alpha & -\sin\alpha \\ \sin\alpha & \cos\alpha \end{pmatrix} \cdot \vec{s_i}' - \vec{t} \tag{4.1}$$

In einem zweiten Schritt wird in Abhängigkeit von der im letzten Durchlauf berechneten Geschwindigkeit $\overrightarrow{ball}_{vel}$ und der Zeit Δt, die zwischen dem letzten und dem aktuellen Durchlauf vergangen ist, berechnet, wo sich das Partikel bei einer als konstant angenommenen Bewegung (*Constant-Speed-Model*) befinden würde (Gleichung (4.2)).

$$\vec{s_i} = \vec{s_i}' + \overrightarrow{ball}_{vel} \cdot \Delta t \tag{4.2}$$

Im *TimeUpdate* werden nur Informationen aus der Vergangenheit und keine aktuellen Messungen benutzt. Daher sind die Annahmen über die Partikelzustände nicht verifiziert. Die Validitäten werden abhängig vom Systemrauschen $\sigma = 0{,}025$ dekrementiert. Das Systemrauschen spiegelt Ungenauigkeiten in der Ballbewegung wider, die aus umgebungsbedingten Einflüssen (zum Beispiel Unebenheiten im Teppich, Reibung, Kollisionen etc.) resultieren. Ein größerer Wert für σ würde die Validitäten zu stark verkleinern (σ sollte nicht größer sein als der Wert, um den die Validität

im *MeasurementUpdate* (siehe Kapitel 4.3.1.2) erhöht wird, falls das Partikel in der Nähe eines Perzepts liegt). Ein kleinerer Wert ist nicht repräsentativ für die genannten umgebungsbedingten Einflüsse. Außerdem wird der Wert ω (siehe Gleichung (4.4)), der abhängig von der Anzahl der Frames $\Delta f_{Perzept}$ seit dem letzten *BallPercept* einen Dekrementierungswert bestimmt, mit einbezogen (siehe Gleichung (4.3)).

$$\mathrm{P}(s_i) = \mathrm{P}\left(s_i^{'}\right) \cdot (1 - \sigma) \cdot \omega \tag{4.3}$$

$$\omega = \max\left\{0{,}5;\ 1 - \frac{1}{100} \cdot \Delta f_{Perzept}\right\} \tag{4.4}$$

ω wurde so gewählt, dass nach 50 Frames (dies entspricht in etwa 2 s) der Validitätswert halbiert wird. Danach kann dem Perzept nur noch wenig vertraut werden, da sich die Position des Balls innerhalb dieser Zeit stark verändern kann. Damit die Validität jedoch nicht zu stark dekrementiert wird, werden Werte kleiner 0,5 auf 0,5 abgebildet.

4.3.1.2 MeasurementUpdate

Beim *MeasurementUpdate* werden die Partikeldaten anhand der Perzepte aktualisiert. Dabei werden die verschiedenen *BallPercept*e aus dem ImageProcessor (siehe Kapitel 7.1.2) der Reihe nach verarbeitet. Begonnen wird mit dem wahrscheinlichsten Perzept. Perzepte, deren Position weiter weg berechnet wurde als die Felddiagonale lang ist, werden verworfen. Solche Perzepte liegen mit Sicherheit außerhalb des Feldes.

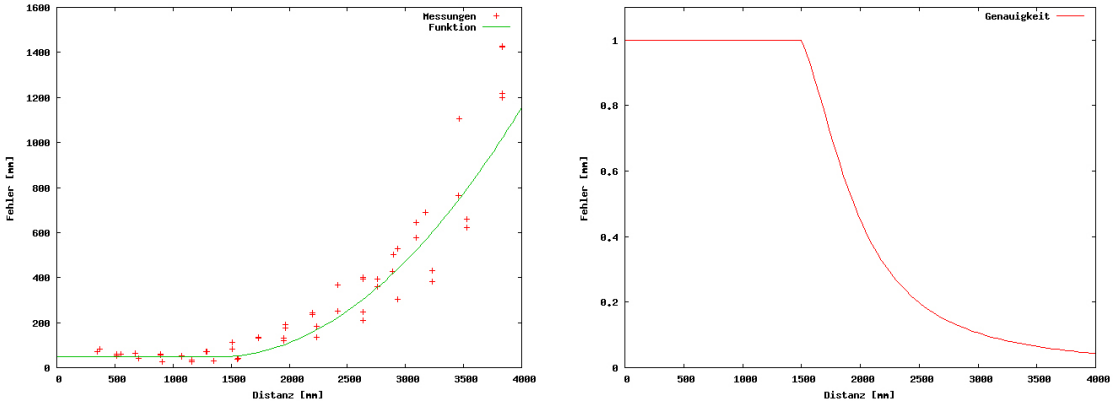
Zusätzlich wird in dem Fall, dass der Roboter den Ball gegriffen hat, die Roboterposition als zusätzliches Perzept benutzt. Dies ist wichtig, da der Roboter in diesem Fall den Ball nicht sehen kann. Die Information bekommt der BallLocator als Feedback aus dem Verhalten.

Genauigkeitsabschätzung der Perzepte Der Einfluss der Perzepte auf die Veränderung der Partikeldaten wird abhängig von der erwarteten Genauigkeit des Perzepts berechnet. Die Genauigkeit $\vec{\vartheta}$ des Perzepts $\vec{b}$ wird von folgenden Faktoren beeinflusst:

1. Perzept-Validität $\mathrm{P}(b)$ aus dem ImageProcessor (siehe Kapitel 7.1.2)

Genauigkeit abhängig von:

2. Distanz zum Perzept: f_{dist}
3. Geschwindigkeit der Kopfbewegung des Roboters: $f_{panningVel}$
4. Laufgeschwindigkeit in x-Richtung: $f_{robotSpeed_x}$
5. Laufgeschwindigkeit in y-Richtung: $f_{robotSpeed_y}$
6. Rotation des Roboters: $f_{robotSpeed_{rot}}$



(a) Fehlerfunktion (b) Resultierende Genauigkeitsfunktion

Abbildung 4.1: Ergebnisse für die Genauigkeit am Beispiel der Distanz zum Perzept

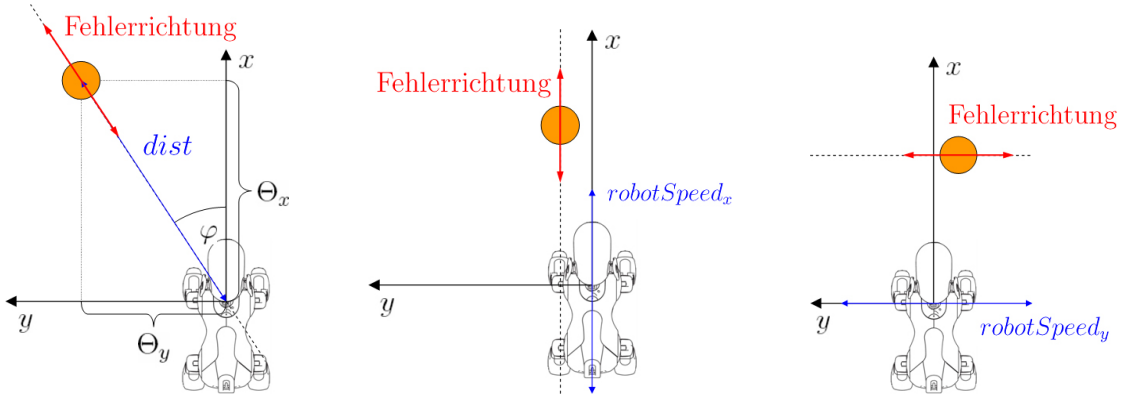
Um die f-Funktionen zu bestimmen, wurde ein Testverhalten implementiert, mit dessen Hilfe Fehlerwerte in Abhängigkeit der verschiedenen Faktoren bestimmt werden konnten. Beispielhaft ist in Abbildung 4.1(a) das Ergebnis der Auswertungen für die Distanz zum Perzept zu sehen. Die f-Funktionen sind umgekehrt proportional zu den erhaltenen Fehlerfunktionen. Außerdem werden die Funktionen durch Multiplikation mit dem minimalen Fehlerwert so gestreckt, dass das Funktionsmaximum bei 1 liegt. Jeder Funktionswert wird weiterhin bei einem Wert von kleiner als 0,01 auf 0,01 abgebildet. Abbildung 4.1(b) zeigt die resultierende Genauigkeitsfunktion f_{dist}.

Da der Einfluss $\vec{\vartheta}$ des Fehlers in x- und y-Richtung unterschiedlich ist, müssen jeweils die Anteile in die beiden Richtungen bestimmt werden.

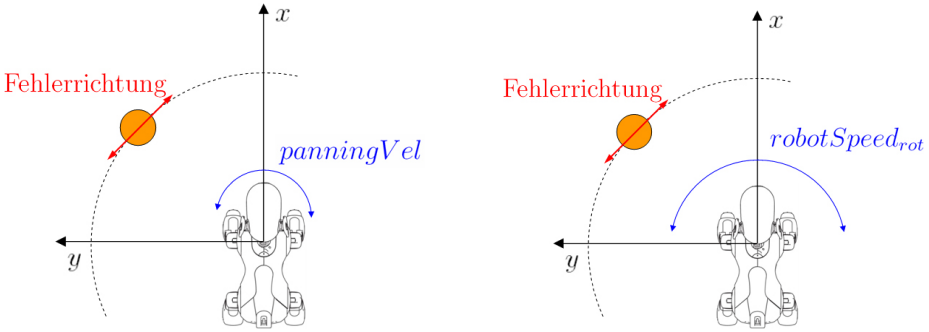
Bei der Distanz ergibt sich dieses durch Einsetzen der x- bzw. y-Entfernung in f_{dist} (siehe Abbildung 4.2(a)). Bei der Kopfbewegungsgeschwindigkeit und der Rotation liegt die Fehlerrichtung tangential zur Rotationsbewegung (vergleiche Abbildung 4.2(d) und Abbildung 4.2(e)). Der Anteil kann über den Winkel zum Perzept bestimmt werden. Für die Laufgeschwindigkeit des Roboters ist die Fehlerrichtung jeweils parallel zur Bewegungsrichtung (siehe Abbildung 4.2(b) und Abbildung 4.2(c)). Daher geht $f_{robotSpeed_x}$ nur in die Genauigkeit in x-Richtung und $f_{robotSpeed_y}$ nur in die Genauigkeit in y-Richtung ein.

Die Genauigeit $\vec{\vartheta}$ ergibt sich schließlich durch Multiplikation der Perzeptvalidität aus dem ImageProcessor und der einzelnen erhaltenen Genauigkeitsfunktionen.

Für das zusätzliche Perzept beim Greifen des Balles wird die Genauigkeit fest auf 0,6 gesetzt. Dabei sind die Genauigkeitsfunktionen konstant 1, da die Genauigkeit von keinem der genannten Faktoren abhängt. Die Validität des Perzepts wird nicht auf 1 gesetzt, da es auch ein fehlerhaftes Feedback aus dem Verhalten geben kann (zum Beispiel, wenn das Greifen schief geht). Erfahrungsgemäß ist die Information in 60% der Fälle korrekt. In diesem Fall ist die Ballposition sehr genau, da der Roboter den Ball zwischen Kinn und Brust eingeklemmt hat und die Berechnung relativ zum Roboter durchgeführt wird.



(a) Distanz (b) x-Laufrichtung (c) y-Laufrichtung



(d) Kopfbewegung (e) Rotation

Abbildung 4.2: Fehlerrichtungen für die verschiedenen Ungenauigkeitsfaktoren

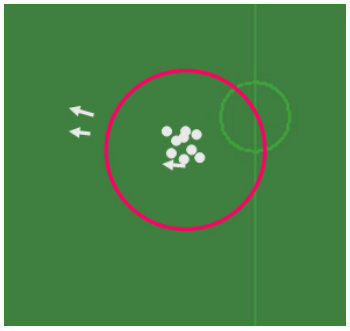


Abbildung 4.3: Der rote Kreis zeigt, welche Partikel in der Nähe des Perzepts liegen

Aktualisieren der Partikeldaten Beim Aktualisieren eines Partikels wird unterschieden, ob die Position in der Nähe des Perzepts $\vec{b}$ liegt oder ob sie weiter weg ist (Abbildung 4.3). In der Nähe des Perzepts bedeutet, dass die Distanz kleiner ist als $10 \cdot$ Ballradius. Dies entspricht 43 cm. Ist die Distanz größer als dieser Grenzwert, kann man sich sicher sein, dass es sich entweder um ein Fehlperzept handelt oder die angenommene Partikelposition falsch ist.

Für Partikel, die nah am Perzept liegen, wird die neue Partikelposition $\vec{s}_i$ durch ein gewichtetes Mittel aus alter Partikelposition und Perzept berechnet (Gleichung (4.5) und Gleichung (4.6)). Dabei wird die Partikelposition vierfach gewichtet, um die Position stabiler zu halten. Größere Werte würden die Reaktivität, kleinere Werte die Stabilität der Ballposition verschlechtern. Weiterhin erhält sie als Gewicht die Partikelvalidität $\mathrm{P}\left(s_i^{'}\right)$. Das Perzept wird mit der Genauigkeit $\vec{\vartheta}$ gewichtet.

$$s_{i,x} = \frac{4 \cdot \mathrm{P}\left(s_i^{'}\right) \cdot s_{i,x}^{'} + \vartheta_x \cdot b_x}{4 \cdot \mathrm{P}\left(s_i^{'}\right) + \vartheta_x} \tag{4.5}$$

$$s_{i,y} = \frac{4 \cdot \mathrm{P}\left(s_i^{'}\right) \cdot s_{i,y}^{'} + \vartheta_y \cdot b_y}{4 \cdot \mathrm{P}\left(s_i^{'}\right) + \vartheta_y} \tag{4.6}$$

Die Position des Partikels wurde verifiziert und verbessert. Daher wird die Validität abhängig von der bisherigen Partikelvalidität und der Genauigkeit des Perzepts um einen Wert $\delta_i \in \left]0; \frac{1}{4} \cdot \mathrm{P}\left(s_i^{'}\right)\right]$ erhöht (siehe Gleichung (4.7)). Validitätswerte größer 1 werden auf 1 abgebildet.

$$\mathrm{P}(s_i) = \min\left\{1;\ \mathrm{P}\left(s_i^{'}\right) + \underbrace{\frac{1}{4} \cdot \mathrm{P}\left(s_i^{'}\right) \cdot \frac{\vartheta_x + \vartheta_y}{2}}_{\delta_i}\right\} \tag{4.7}$$

Damit Fehlperzepte nicht zu viel Einfluss auf die Partikelwolke haben, wird die Position von Partikeln außerhalb des Bereichs $10 \cdot$ Ballradius nicht verändert. Die Validität wird abhängig von der Genauigkeit des Perzepts dekrementiert (siehe Gleichung (4.8)). Ergibt sich dabei eine Validität kleiner als 0, wird diese auf 0 abgebildet.

$$\mathrm{P}(s_i) = \max\left\{0;\ \mathrm{P}\left(s_i^{'}\right) - \frac{1}{10} \cdot \mathrm{P}\left(s_i^{'}\right) \cdot \frac{\vartheta_x + \vartheta_y}{2}\right\} \tag{4.8}$$

4.3.1.3 Berechnen der Gesamtposition

Um aus den Partikeln des BallLocators eine Gesamtposition zu berechnen, benutzen wir die Idee des Clusterns aus dem SelfLocator (siehe Kapitel 6.2.4) und passen sie an unser Problem an. Die Idee ist, das Feld in ein Netz von 10×10 Zellen zu unterteilen. Es wird dann für jede Zelle die Summe der Validitäten der in ihr enthaltenen Partikel berechnet, um das 2×2-Netz zu berechnen, welches die größte aufsummierte Positionsvalidität hat (siehe Abbildung 4.4).

Für die k Partikel in diesem 2×2-Netz $(\vec{s}_1, \ldots, \vec{s}_k)$ wird anschließend ein gewichtetes Mittel für die Position berechnet. Bei der Berechnung der Positionsvalidität

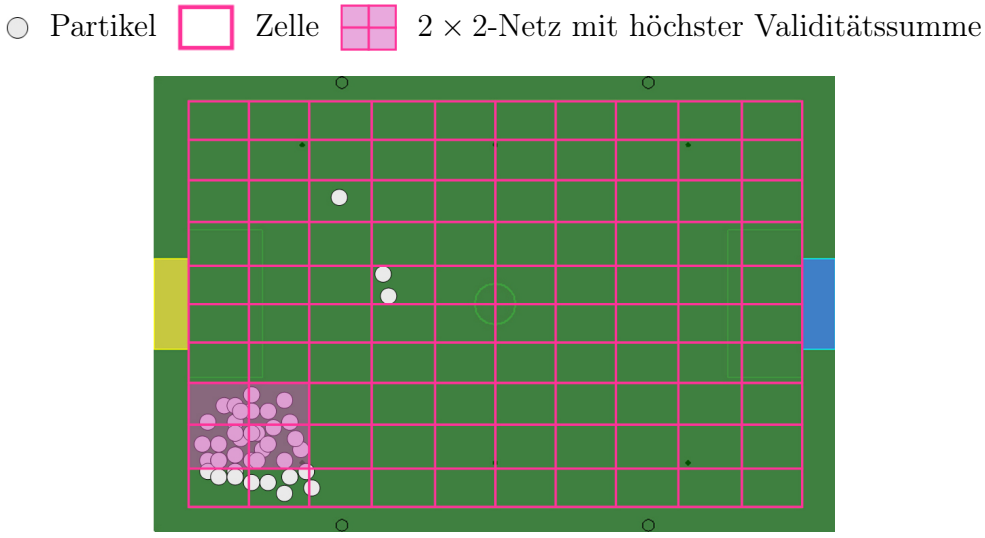


Abbildung 4.4: Einteilung des Feldes in Zellen und Berechnung des 2×2-Netzes mit der höchsten Validitätssumme

wird ein Mittel der Validitäten aller Partikel berechnet. Jedes Partikel wird mit dem Kehrwert der Distanz zur berechneten Gesamtballposition gewichtet. Gesamtposition und deren Validität berechnen sich dann durch Gleichung (4.9).

$$\overrightarrow{ball}_{pos} = \frac{\sum_{i=1}^{k} \vec{s}_i \cdot \mathrm{P}(s_i)}{\sum_{i=1}^{k} \mathrm{P}(s_i)} \qquad \mathrm{P}(ball_{pos}) = \frac{\sum_{i=1}^{\ell} \frac{\mathrm{P}(s_i)}{\mathrm{dist}\left(\overrightarrow{ball}_{pos},\ \vec{s}_i\right)}}{\sum_{i=1}^{\ell} \frac{1}{\mathrm{dist}\left(\overrightarrow{ball}_{pos},\ \vec{s}_i\right)}} \tag{4.9}$$

4.3.1.4 Sensor-Resetting

Jedes Partikel, das eine zu kleine Validität besitzt, wird entfernt und neu in der Nähe des letzten Perzepts eingefügt. Ein Wert von 0,05 hat sich als Kriterium günstig erwiesen, da kleinere Validitäten nur vorkommen, wenn das Partikel schon lange nicht mehr in der Nähe eines Perzepts lag. Bei einer Validität von größer als 0,05 kann die Partikelposition und Validität aber auch noch schnell genug verbessert werden, falls das Partikel doch wieder nah an einem Perzept liegen sollte.

Außerdem wird sichergestellt, dass in der Nähe des Perzepts immer mindestens 5% der Partikel liegen. Diese Maßnahme ist wichtig, weil nur die Position von Partikeln in der Nähe des Perzepts aktualisiert werden. Liegen mindestens 5% der Partikel in der Nähe, wird die Validität der Partikel schnell groß genug, um die Gesamtposition in die Nähe des Perzepts zu verbessern. Falls weniger als 5% der Partikel in der Nähe des Perzepts liegen, werden so lange die Partikel mit der geringsten Validität in die Nähe des Perzepts neu eingefügt, bis dieser Grenzwert überschritten ist.

Für das Einfügen wird eine Gaußverteilung benutzt. Je älter das Perzept ist, desto größer ist die Standardabweichung.

Das eingefügte Partikel bekommt eine Validität von 0,20. Bei einer Validität von

0,20 und falls lange kein Ball gesehen wurde, überlebt das Partikel nur zwei Frames (vergleiche Gleichung (4.3)), bis es wieder neu eingefügt wird. Falls es in der Nähe eines Perzepts liegt, wird die Wahrscheinlichkeit aber schnell erhöht.

4.3.2 Geschwindigkeitsberechnung

Die zu Grunde liegende Idee ist, die Geschwindigkeit $\overrightarrow{ball}_{vel}$ so aus den letzten Beobachtungen zu schätzen, dass der quadratische Fehler zwischen beobachteter und erwarteter Position minimiert wird. Beobachtungen sind in unserem Fall die Perzepte. Die erwartete Position ist die berechnete Gesamtposition.

Es wird angenommen, dass die Ballbewegung auf dem Spielfeld linear ist. Das heißt, die Ballposition $\overrightarrow{ball}_{pos}^{t_n}$ zum Zeitpunkt t_n ist durch Gleichung (4.10) gegeben.

$$\overrightarrow{ball}_{pos}^{t_n} = \overrightarrow{ball}_{pos}^{t_{n-1}} + \overrightarrow{ball}_{vel}^{t_{n-1}} \cdot (t_n - t_{n-1}) \tag{4.10}$$

Für Beobachtungen $\vec{b}^{t_1}, \vec{b}^{t_2}, \ldots, \vec{b}^{t_n}$ zu Zeitpunkten $t_1, t_2, \ldots, t_n$ ist also die Minimierungsfunktion (4.11) zu lösen.

$$\min_{\overrightarrow{ball}_{pos}^{t_n}, \overrightarrow{ball}_{vel}^{t_n}} \left[\frac{1}{2} \cdot \sum_{k=1}^{n} \left\| \overrightarrow{ball}_{pos}^{t_n} + \overrightarrow{ball}_{vel}^{t_k} \cdot (t_k - t_n) - \vec{b}^{t_k} \right\|^2 \right] \tag{4.11}$$

Diese Schätzung liefert robuste Werte für Position und Geschwindigkeit, ohne Annahmen über die Art des Rauschens der Messungen zu machen. Allerdings löst sie nicht das Problem, dass die erwartete Geschwindigkeit bei einem liegendem Ball auf Grund von springenden Perzepten größer als 0 ist. Um diesen Effekt zu minimieren, wird die Optimierungsfunktion um einen weiteren Term – dem gewichteten Quadrat der geschätzten Ballgeschwindigkeit – erweitert (Gleichung (4.12)).

$$\min_{\overrightarrow{ball}_{pos}^{t_n}, \overrightarrow{ball}_{vel}^{t_n}} \left[\frac{1}{2} \cdot \sum_{k=1}^{n} \left\| \overrightarrow{ball}_{pos}^{t_n} + \overrightarrow{ball}_{vel}^{t_k} \cdot (t_k - t_n) - \vec{b}^{t_k} \right\|^2 + \frac{\lambda}{2} \cdot \left\| \overrightarrow{ball}_{vel}^{t_n} \right\|^2 \right] \tag{4.12}$$

Durch Lösen der Minimierungsfunktion (4.12) erhält man Gleichung (4.13) als Schätzung der aktuellen Geschwindigkeit.

$$\overrightarrow{ball}_{vel}^{t_n} = \frac{n \cdot \sum_{k=1}^{n} \left((t_k - t_n) \cdot \vec{b}^{t_k} \right) - \sum_{k=1}^{n} (t_k - t_n) \cdot \sum_{k=1}^{n} \vec{b}^{t_k}}{n \cdot \left(\lambda + \sum_{k=1}^{n} (t_k - t_n)^2 \right) - \left(\sum_{k=1}^{n} (t_k - t_n) \right)^2} \tag{4.13}$$

Da die Genauigkeit des Perzepts in x-Richtung verschieden von der Genauigkeit in y-Richtung ist, werden wie schon bei der Positionsberechnung (vergleiche Kapitel 4.3.1.2) verschiedene λ-Werte für die jeweiligen Richtungen bestimmt (siehe Gleichung (4.14)). Je größer λ_x bzw. λ_y wird, desto stärker werden kleinere Geschwindigkeiten in die betroffene Richtung bevorzugt.

$$\begin{pmatrix} \lambda_x \\ \lambda_y \end{pmatrix} = \left(1 - \begin{pmatrix} \vartheta_x \\ \vartheta_y \end{pmatrix} \right) \cdot 10 \tag{4.14}$$

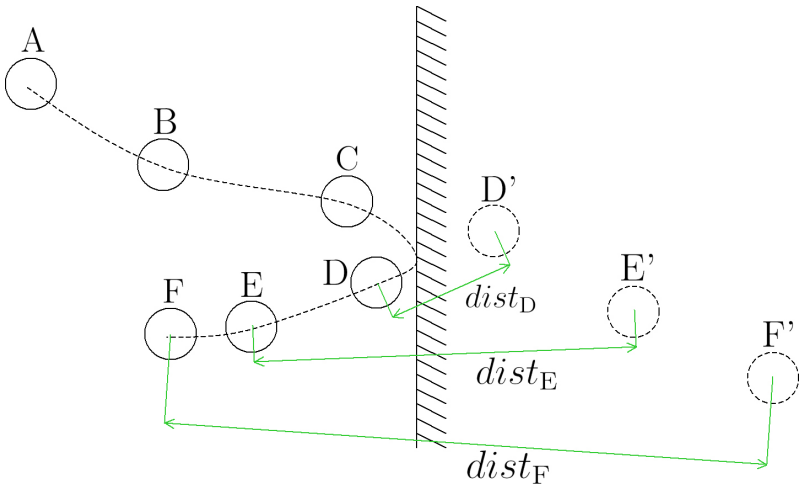


Abbildung 4.5: Plötzliche Richtungsänderung des Balls

Die Güte der Geschwindigkeitsschätzung hängt stark von der Anzahl n der Beobachtungen ab, die in die Berechnung einbezogen werden. Bei kleinem n ist der Einfluss des Rauschens auf die Schätzung hoch. Zu große Werte für n schränken jedoch die Reaktivität ein, falls der Ball plötzlich seine Bewegungsrichtung ändert. Daher wird die Anzahl der einbezogenen Beobachtungen dynamisch verändert.

Die letzten Perzepte und deren Genauigkeiten $\vec{\vartheta}$ werden in einem Puffer der maximalen Größe 15 gespeichert. Bei jeder Einfügung wird überprüft, ob die Position in das bisherige Bewegungsmodell passt, das heißt es wird getestet, ob die Distanz des Partikels zur momentan geschätzten Ballposition einen bestimmten Wert δ überschreitet. In diesem Fall wird der Wert markiert. Falls zwei aufeinanderfolgende Werte markiert sind, wird das bisherige Bewegungsmodell durch Reduzierung des Puffers auf eine Minimalgröße von 3 verworfen. Auf diese Weise werden alte Beobachtungen eliminiert und die Schätzung nur auf Basis der neuesten Werte fortgesetzt.

Abbildung 4.5 zeigt ein Beispiel, bei dem der Ball plötzlich seine Bewegungsrichtung ändert. A – F sind die beobachteten Ballpositionen, D' – F' die aus A, B und C geschätzten Positionen unter der Annahme, dass sich der Ball mit einer konstanten Geschwindigkeit bewegt. Da $dist_{\mathrm{D}} < \delta$ ist, wird D' nicht markiert. Es gilt aber $dist_{\mathrm{E}} > \delta$ und $dist_{\mathrm{F}} > \delta$. E und F werden markiert und das bisherige Bewegungsmodell verworfen.

Das Vertrauen $\mathrm{P}(ball_{vel})$ in die Berechnung der Geschwindigkeit hängt von der Validität der Beobachtungen ab. Dies ist in unserem Fall die Genauigkeit $\vec{\vartheta}$ (vergleiche Kapitel 4.3.1.2) der betrachteten Perzepte. Des Weiteren geht die Anzahl der Beobachtungen n und deren Alter in die Berechnung ein (siehe Gleichung (4.15)).

$$\mathrm{P}(ball_{vel}) = \frac{\left(1 - \frac{1}{n}\right) \cdot \sum_{k=1}^{n} \frac{\vec{\vartheta}^{\,t_k}}{t_n - t_k + 1}}{\sum_{k=1}^{n} \frac{1}{t_n - t_k + 1}} \tag{4.15}$$

4.3.3 Diskussion der Güte des neuen BallLocators

Um die Güte des neuen BallLocators zu analysieren, wurde das Modul mit der bestehenden Lösung – dem GT2004BallLocator – verglichen. Als Referenz wurde hierfür die CeilingCam herangezogen. Für die Messungen wurde ein so genannter „ComboBallLocator“ benutzt. In diesem wurden in jedem Durchlauf unabhängig voneinander sowohl der GT2004BallLocator, als auch der GT2005BallLocator ausgeführt. Die jeweils berechneten Ballpositionen und -geschwindigkeiten wurden mit denen der CeilingCam verglichen und ein Fehlerwert berechnet. Der Verlauf der Fehlerwerte wurde anschließend mit Hilfe eines Statistik-Tools[3] visualisiert.

Frames, in denen die CeilingCam keinen Ball erkannt hat, wurden für die Fehlermessung ignoriert, da kein Referenzwert zur Verfügung stand. Dies kann zum Beispiel passieren, wenn ein Roboter den Ball gegriffen hat. Zu diesen Zeitpunkten sind Lücken in den Diagrammen gelassen worden.

Im Spiel kann es häufiger vorkommen, dass der ImageProcessor fälschlicherweise einen Ball erkennt. Dies kann aus einer schlechten Farbtabelle mit orangenen Pixeln in roten Trikots oder im gelben Tor resultieren. Andererseits kann man auch nicht ausschließen, dass sich orangene Kleidung oder Gegenstände im Publikum befinden (siehe Kapitel 4.3.3.1). Problematisch wird diese Situation, wenn sich zusätzlich der richtige Ball im Bild befindet und das Fehlperzept vom ImageProcessor als wahrscheinlicher eingestuft wird. Der Kalman-Filter des GT2004BallLocators betrachtet in diesem Fall immer nur das wahrscheinlichste Perzept und verwirft das Korrekte. Der neue Ansatz hingegen benutzt alle Perzepte, die vom ImageProcessor erkannt werden. Aus diesem Grund wurde in diesen speziellen Situationen untersucht, ob der neue BallLocator hier tatsächlich eine Verbesserung erzielen konnte. Die Ergebnisse einer dieser Situationen ist in Kapitel 4.3.3.1 näher beschrieben.

Außerdem wurden verschiedene Spielsituationen (1-gegen-1, Kapitel 4.3.3.2, und 2-gegen-2, Kapitel 4.3.3.3) betrachtet, um zu zeigen, dass der entwickelte BallLocator auch in reellen Situationen besser ist.

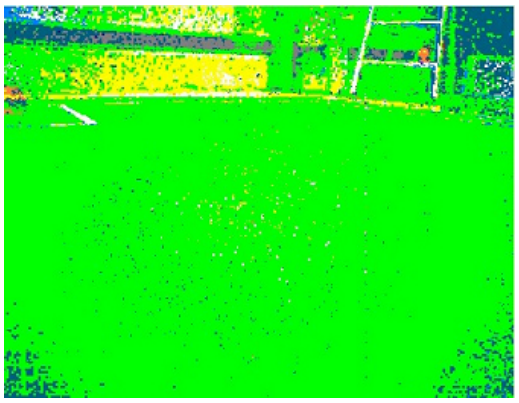
4.3.3.1 Ball neben dem Spielfeld

Der Versuchsaufbau ist in Abbildung 4.6(c) dargestellt. Neben dem Spielfeld wurde auf geringer Höhe ein Ball platziert. Zusätzlich zu diesem Ball neben dem Spielfeld befindet sich ein Ball auf dem Spielfeld. Im optimalen Fall sollte nur der Ball auf dem Spielfeld erkannt werden und der neben dem Spielfeld ignoriert werden. Abbildung 4.6(a) zeigt das zugehörige Kamerabild des Roboters, Abbildung 4.6(b) das farbsegmentierte Bild, welches die Güte der Farbtabelle zeigt.

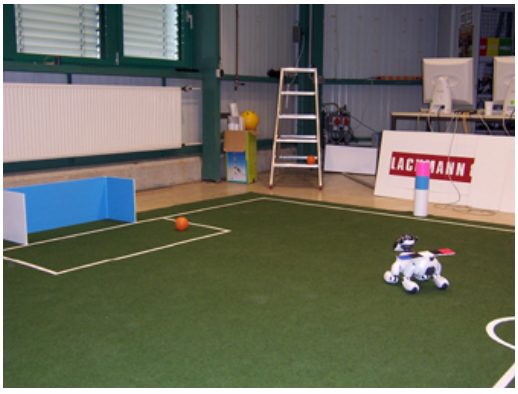
In Abbildung 4.7(a) ist das Ergebnis dargestellt, wenn der Roboter als Kopfbewegung „*search-for-landmarks*“ hat. Bei dieser Kopfbewegung schwenkt der Kopf kontinuierlich von links nach rechts und wieder zurück. Dies führt dazu, dass der Roboter in einigen Frames nur einen der Bälle sieht. Ist dies der Ball außerhalb des Spielfeldes, springt die berechnete Position vom GT2004BallLocator sehr schnell dorthin.

[3] `http://www.gnuplot.info/`

(a) Kamera-Bild des Roboters

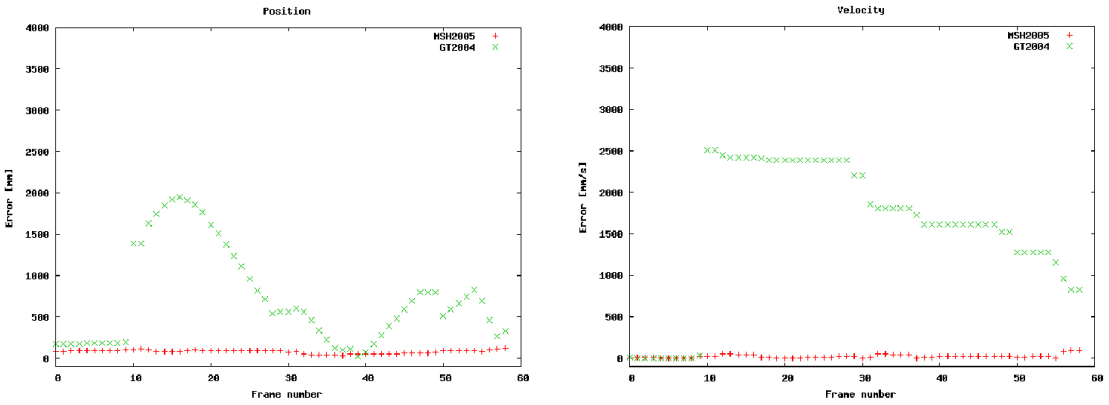


(b) Farbsegmentiertes Bild



(c) Im Strafraum liegt der Spielball, neben dem Spielfeld auf einer Leitersprosse liegt ein zweiter Ball.

Abbildung 4.6: Testszenario für eine spezielle Spielsituation



(a) Positionsberechnung (b) Geschwindigkeitsberechnung

Abbildung 4.7: Ergebnisse für das Testszenario (Ball im Publikum)

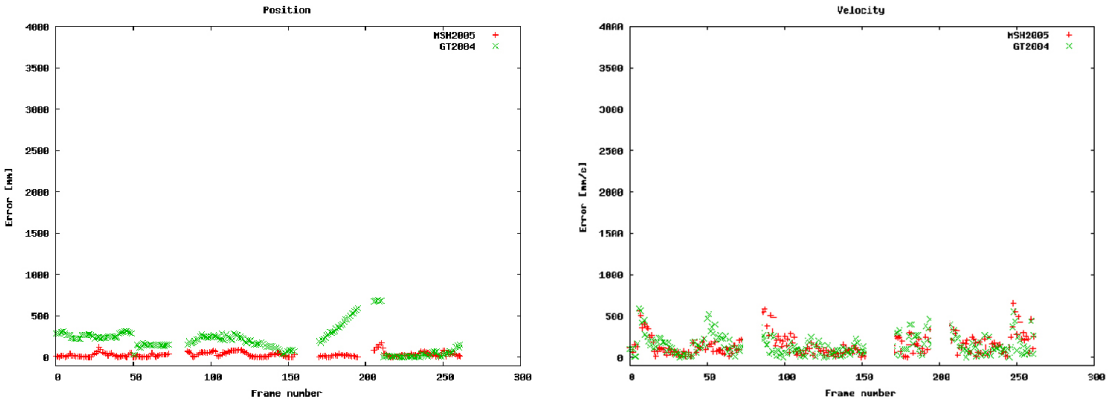
Der GT2005BallLocator hingegen berechnet weiterhin die korrekte Position, da das Partikel-Filter-System wenige Fehlperzepte ausgleichen kann. Der GT2004BallLocator berechnet immer dann, wenn er nur den Ball außerhalb des Spielfeldes sieht, dort die Ballposition. Sobald er aber auch – oder nur – den Ball auf dem Spielfeld sieht, berechnet er die Ballposition dort. Dies führt dazu, dass die Position stark springt und eine sehr große Geschwindigkeit berechnet wird, obwohl sich der Ball nicht bewegt (siehe Abbildung 4.7(b)).

4.3.3.2 1-gegen-1-Spielsituation

Um zu untersuchen, ob auch in einer typischen Spielsituation, und nicht nur in künstlich provozierten Situationen, der GT2005BallLocator bessere Ergebnisse liefert, wurden zwei Situationen untersucht. Als erstes wurde ein Spiel 1-gegen-1 durchgeführt. Der Roboter, dessen Ergebnisse mit denen der CeilingCam verglichen werden, ist ein Striker. Er spielt auf das blaue Tor gegen einen Goalie. Es wurden verschiedene Durchläufe getestet. In Abbildung 4.8 sind die Ergebnisse einer ausgewählten, repräsentativen Situation abgebildet.

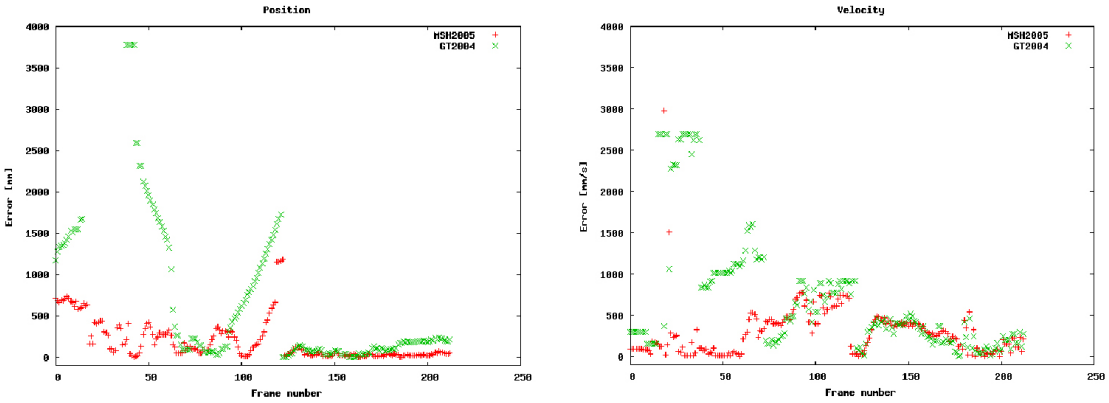
Zu Beginn dribbelt der Roboter den Ball in Richtung des Tors. Dabei verdeckt er zwischenzeitlich den Ball mit dem Kopf, so dass die CeilingCam ihn nicht mehr sehen kann. Irgendwann erreicht er mit dem Ball das Tor, so dass der Ball am Goalie abprallt. Danach dreht sich der Striker zum Ball, greift ihn und dreht sich mit ihm, um ihn dann in Richtung des Tores loszulassen. Während des Greifens kann die CeilingCam den Ball wieder nicht sehen. Es ist aber interessant, in den Ergebnissen zu sehen, dass nach dem Loslassen des Balls, wenn er vom Roboter wegrollt, der neue BallLocator eindeutig bessere Ergebnisse liefert.

In der gesamtem Spielsituation ist die Positionsberechnung des GT2005BallLocators besser. Bei der Geschwindigkeitsberechnung ist das Ergebnis ausgewogen: GT2004BallLocator und GT2005BallLocator sind meist in etwa gleich gut bzw. die Anzahl schlechterer und besserer Ergebnisse gleichen sich aus.



(a) Positionsberechnung (b) Geschwindigkeitsberechnung

Abbildung 4.8: Ergebnisse Spielsituation (1 gegen 1)



(a) Positionsberechnung (b) Geschwindigkeitsberechnung

Abbildung 4.9: Ergebnisse Spielsituation (2 gegen 2)

4.3.3.3 2-gegen-2-Spielsituation

In der zweiten Spielsituation spielen ein Goalie und ein Verteidiger gegen einen Striker und einen Supporter. Auf dem Striker findet der Vergleich statt. Auch hier wurden mehrere Tests gemacht. In Abbildung 4.9 sind wieder die Ergebnisse eines repräsentativen Vergleichs dargestellt.

In dieser Situation läßt sich gut vergleichen, wie sich die BallLocator verhalten, wenn es Kollisionen gibt. In der Situation wollen sowohl der Striker, als auch der Verteidiger zum Ball gehen und verhaken sich dabei. Weiterhin verdeckt der Supporter kurzzeitig den Ball. Bei den Ergebnissen ist ein deutlicher Unterschied zwischen GT2005BallLocator und GT2004BallLocator zu erkennen. Die Position wird deutlich besser berechnet. Auch bei der Geschwindigkeit ist eine Überlegenheit des neuen BallLocator auszumachen.

4.3.4 Ausblick

Im *MeasurementUpdate* wird zwischen Partikeln unterschieden, die nah am Perzept liegen und solchen, die weiter weg sind. Die Entscheidung, ob ein Perzept nah ist, ist im Moment eine statische (Entfernung $< 10 \cdot$ Ballradius). Es hat sich als sinnvoll erwiesen, die Güte des Perzepts anhand verschiedener Einflussgrößen wie die Distanz zum Perzept und die Kopfbewegungsgeschwindigkeit zu beurteilen. Daher soll in der Zukunft untersucht werden, ob die Entscheidung, was in der Nähe des Perzepts bedeutet, auch abhängig von diesen Größen gemacht werden soll.

5 TeamBallLocator

Je nachdem, wo sich der Roboter und der Ball auf dem Spielfeld befinden, ist es für den Roboter nicht möglich, den Ball zu sehen. In solchen Situationen ist es aber trotzdem notwendig zu wissen, wo sich der Ball gerade befindet. Die Aufgabe des TeamBallLocators besteht darin, diese Information, die so genannte *CommunicatedBallPosition*, zu berechnen.

5.1 Problembeschreibung

Beim TeamBallLocator werden, im Gegensatz zum BallLocator, neben den Informationen, die von den eigenen Sensoren zur Verfügung gestellt werden, auch Informationen der Mitspieler benutzt. Diese werden über das WLAN kommuniziert.

Während der BallLocator unabhängig von der Lokalisierung des Roboters ist, ist dies beim TeamBallLocator nicht der Fall, da der Ball in einem Koordinatensystem modelliert werden muss, das für alle Roboter gleich ist. Die Hauptaufgabe ist es, trotz fehlerbehafteter Ballposition und Lokalisierung eine möglichst genaue Information bereitzustellen, wo sich der Ball auf dem Spielfeld befindet. Diese Information soll auch dann noch gegeben sein, wenn der Roboter selber den Ball nicht sieht.

Ein weiteres Problem besteht darin, dass die empfangenen Daten widersprüchlich sein können oder in manchen TeamBallLocator-Durchläufen keine Informationen empfangen werden. Mit dieser Problematik muss der TeamBallLocator zurecht kommen. Die berechnete Ballposition soll dennoch möglichst stabil sein.

Die Informationen des TeamBallLocators werden z.B. für die Positionierung der Supporter benötigt. Hierfür ist es wichtig, dass die *CommunicatedBallPosition* aller Roboter in einem Team identisch ist, da ansonsten die taktischen Entscheidungen nicht zueinander passen. Zusätzlich gibt die *CommunicatedBallPosition* einen Anhaltspunkt, wo ein Roboter am besten nach dem Ball suchen sollte, wenn er ihn für lange Zeit nicht gesehen hat. Da die Ballgeschwindigkeit bei beiden Anwendungen nicht benötigt wird, kann der TeamBallLocator auf deren Berechnung verzichten. Wir konzentrieren uns im Folgenden deshalb auf eine möglichst exakte Positionsberechnung.

5.2 Ansätze zur Lösung des Problems

Der Ansatz des GT2004TeamBallLocators war es, die berechnete Ballposition samt Roboterposition und Orientierung (*RobotPose*, siehe Kapitel 6) zu kommunizieren [16]. Aus den empfangenen Informationen wurde zuerst anhand der Validität der

	Bytes
Maximale Größe eines Ethernet-Pakets [5]	1518
Header [5]	18
IP Header [6]	20
UDP-Header [7]	16
Insgesamt	1444

Tabelle 5.1: Maximale Größe eines UDP-Pakets

Roboterposition bestimmt, welcher Roboter am besten lokalisiert ist. Dessen Ballposition wurde in Feldkoordinaten umgerechnet und als *CommunicatedBallPosition* ausgegeben. Dieser Ansatz ist sehr simpel und effizient. Der große Nachteil ist aber die Abhängigkeit von der Güte der Validitätsberechnung der *RobotPose*. Außerdem wurden immer nur die gesendeten Informationen von einem Roboter benutzt, obwohl mehr Informationen zur Verfügung standen.

Die Idee des GT2005TeamBallLocators ist es, die Partikel des BallLocators aller vier Roboter in einem Team zu benutzen, um aus diesen die *CommunicatedBallPosition* zu berechnen. Der Vorteil hierbei ist, dass durch die Benutzung der Partikel fehlerhaft erkannte Bälle oder durch schlechte Selbstlokalisierung verfälschte Ballpositionen kompensiert werden können. Auf diese Weise können die Informationen aller Roboter in einem Team kombiniert werden.

Ein simpler Ansatz, um diese Idee zu verwirklichen, ist Algorithmus 5.1.

Algorithmus 5.1 (Simpler Ansatz)

1. Wandle die Partikel des BallLocators in Feldkoordinaten um.
2. Versende die Partikel an alle Mitspieler.
3. Empfange die Partikel aller Mitspieler.
4. Berechne aus den empfangenen und den eigenen, gesendeten Partikeln mit Hilfe des Algorithmus aus dem BallLocator eine Gesamtballposition und -wahrscheinlichkeit.

In der Realität ist Algorithmus 5.1 jedoch nicht zweckmäßig. Der Grund dafür ist, dass es zu aufwändig ist, 40 Partikel zu versenden. Die Daten eines Partikels sind 24 Byte groß (2 doubles für die Position und 1 double für die Validität, 1 double $\hat{=}$ 8 Byte). Damit müssten 960 Byte zusätzlich zu den sonstigen Daten, die ein Roboter über das WLAN überträgt, versendet werden. Ein UDP-Paket darf maximal eine Größe von 1444 Byte haben (siehe Tabelle 5.1). Mehrere Pakete zu versenden erhöht den Aufwand der Verarbeitung und erhöht die Wahrscheinlichkeit, dass Pakete verloren gehen (siehe Kapitel 5.3). Daher ist es notwendig, die Pakete so klein wie möglich zu halten.

Der GT2005TeamBallLocator (siehe Algorithmus 5.2) stellt deshalb die Informationen der 40 Partikel in maximal 12 so genannten repräsentativen Partikeln dar.

Damit reduziert sich die Größe pro Roboter auf nur 288 Byte. Der Ansatz ermöglicht es, die Anzahl der Partikel des BallLocators beliebig zu verändern, ohne sich Gedanken um den TeamBallLocator machen zu müssen.

Algorithmus 5.2 (Verbesserter Ansatz)

1. Wandel die Partikel des BallLocators in Feldkoordinaten um.
2. Berechne repräsentative Partikel.
3. Versende die repräsentativen Partikel an alle Mannschaftsmitglieder.
4. Empfange die repräsentativen Partikel aller Mannschaftsmitglieder.
5. Berechne aus den empfangenen und den eigenen, gesendeten Partikeln mit Hilfe des Algorithmus aus dem BallLocator eine Gesamtballposition und -wahrscheinlichkeit.

Dieser Ansatz wird in Kapitel 5.3 beschrieben.

5.3 Beschreibung der gewählten Lösung

Die Hauptaufgabe ist es, die repräsentativen Partikel zu berechnen, die dann an die anderen Teammitglieder geschickt werden. Bevor aus den empfangenen und den eigenen Partikeln eine Gesamtposition berechnet wird, wird die Validität der empfangenen Partikel abhängig von der Zeit zwischen dem Senden und Empfangen der Partikel runtergesetzt. Die Berechnung der Gesamtposition funktioniert genauso wie im BallLocator (siehe Kapitel 4.3.1.3) – daher wird hier nicht näher auf den Algorithmus eingegangen.

Ein Problem ist, dass auf Grund von WLAN-Ausfällen in manchen TeamBallLocator-Iterationen keine Partikel empfangen werden können. Besonders auf großen Turnieren tritt dieses Problem häufig auf, da dort sehr viele Access-Points gleichzeitig angeschaltet sind und sich oft gegenseitig stören. Da es aber dennoch notwendig ist, zumindest eine grobe Richtung anzugeben, in der sich der Ball befindet, sollte man in solchen Fällen nicht die alten Partikel verwerfen, sondern diese besonders behandeln. Es ist besser, eine ungenaue Angabe über die Ballposition zu haben als gar keine.

5.3.1 Berechnen von repräsentativen Partikeln

Im Folgenden wird erläutert, wie aus den Partikeln des BallLocators die erwähnten repräsentativen Partikel berechnet werden.

5.3.1.1 Umwandeln von Roboterkoordinaten in Feldkoordinaten

Der GT2005SelfLocator wurde so angepasst, dass er für den TeamBallLocator nicht nur eine Roboterposition, sondern bis zu drei (in der so genannten *RobotPose-Collection* (siehe Kapitel 6.4.5)) ausgibt. In typischen Spielsituationen hat sich gezeigt, dass 90% der Gesamtwahrscheinlichkeit der *RobotPose* durch drei SelfLocator-Partikel-Cluster repräsentiert wird.

Die Benutzung von drei Roboterpositionen, anstatt von einer, hat den Vorteil, dass es möglich ist, eine gute Ballposition zu berechnen, auch wenn ein Roboter nicht perfekt lokalisiert ist. Es kann nämlich vorkommen, dass er nur Informationen sieht, die auf dem Feld spiegelsymmetrische Positionen gleichwahrscheinlich machen (Linien sind spiegelsymmetrisch, Landmarken nicht) und somit die *RobotPose* zwischen diesen Positionen springt. Die Verwendung mehrerer möglicher Roboterpositionen bringt hier den Vorteil, dass die richtige Position nicht verworfen wird, wenn der SelfLocator die falsche Position als *RobotPose* ausgeben würde.

Der TeamBallLocator reagiert auf die verschiedenen *RobotPose*s, indem er aus den 40 Partikeln in Roboterkoordinaten 120 Partikel in Feldkoordinaten berechnet – für jedes Partikel in Roboterkoordinaten drei Partikel in Feldkoordinaten. Bei der Berechnung der Partikelvaliditäten werden die Validitäten für die speziellen Roboterpositionen P($RobotPose_1$), P($RobotPose_2$) und P($RobotPose_3$) mit der Validität des Partikels P($Partikel$) verrechnet. Es ergeben sich die Validitäten P($Partikel_1$), P($Partikel_2$) und P($Partikel_3$) für die drei Partikel in Feldkoordinaten durch Gleichung (5.1).

$$\mathrm{P}(Partikel_i) = \mathrm{P}(RobotPose_i) \cdot \mathrm{P}(Partikel) \qquad i \in \{1, 2, 3\} \tag{5.1}$$

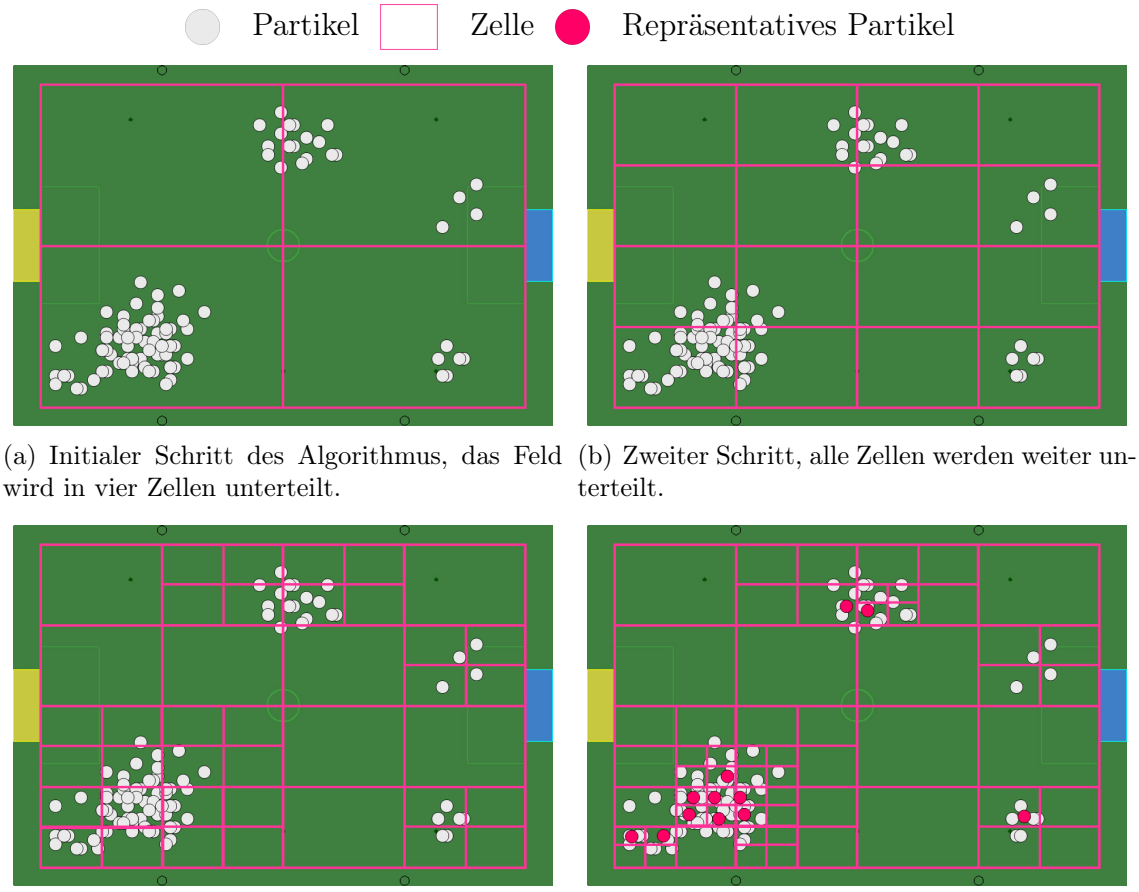
5.3.1.2 Zusammenfassen von Partikeln

Aus den 120 Partikeln in Feldkoordinaten werden bis zu 12 repräsentative Partikel berechnet. Dazu wird ein rekursiver Algorithmus angewendet. Zur Modellierung der zugehörigen Datenstruktur benutzen wir das Entwurfsmuster „Kompositum“ [3], welches in Abbildung 5.3 dargestellt ist. Das Feld wird in Zellen unterteilt. Dabei unterscheiden wir zwischen Basiszellen und zusammengesetzten Zellen:

- *Basiszellen:* Eine Basiszelle beinhaltet Partikel.
- *Zusammengesetzte Zellen:* Eine zusammengesetzte Zelle beinhaltet vier Basiszellen.

Abbildung 5.1 zeigt ein Beispiel, wie der Algorithmus schrittweise das Feld unterteilt. Abbildung 5.2 visualisiert den zugehörigen Baum der verwendeten Datenstruktur am Ende des Algorithmus.

Der Algorithmus beginnt mit einer zusammengesetzten Zelle und fügt jedes Partikel in die *root*-Zelle ein. Dabei wird anhand der Position des Partikels entschieden, in welche der Basiszellen das Partikel gehört. Anschließend wird der rekursive Algorithmus 5.3 angewendet.



(a) Initialer Schritt des Algorithmus, das Feld wird in vier Zellen unterteilt.

(b) Zweiter Schritt, alle Zellen werden weiter unterteilt.

(c) Dritter Schritt, einige Zellen haben eine zu kleine aufsummierte Validität und werden nicht weiter unterteilt, andere hingegen werden weiter aufgeteilt.

(d) Letzter Schritt, ein paar Zellen werden nochmal weiter unterteilt und am Ende wird aus den Zellen mit größtem Γ die repräsentativen Partikel berechnet.

Abbildung 5.1: Darstellung des Algorithmus zum Berechnen von repräsentativen Partikeln anhand eines Beispiels

Zusammengesetzte Zelle
Basiszelle mit zu geringem Γ
Basiszelle, für die ein repräsentatives Partikel berechnet wird

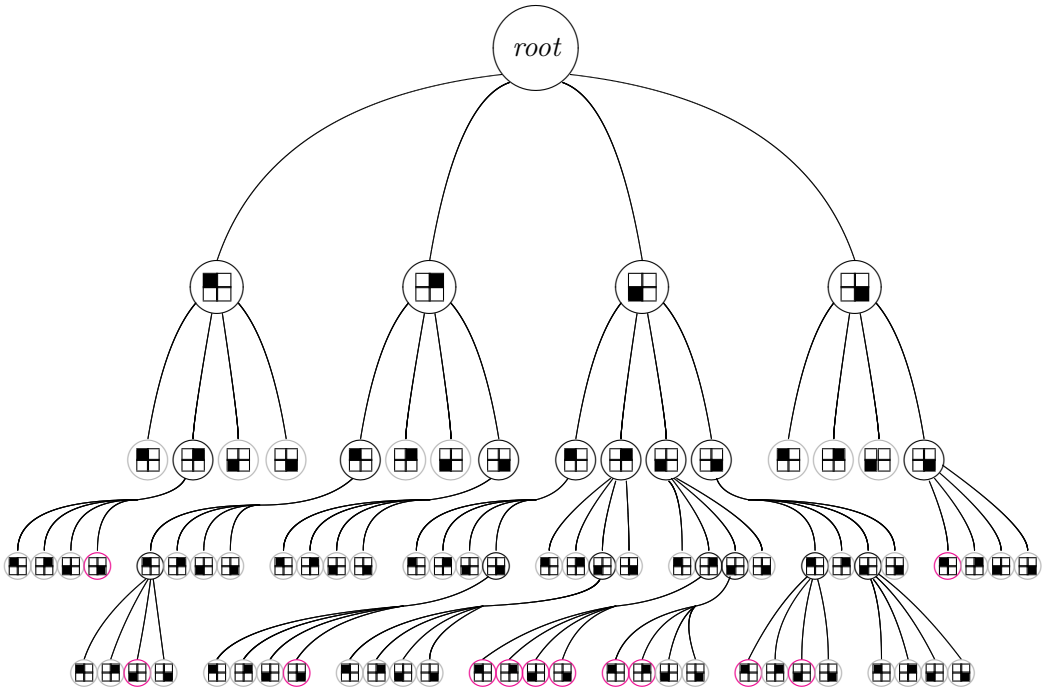


Abbildung 5.2: Datenstruktur für die Zellen am Ende des Beispiels, welches in Abbildung 5.1 dargestellt ist.

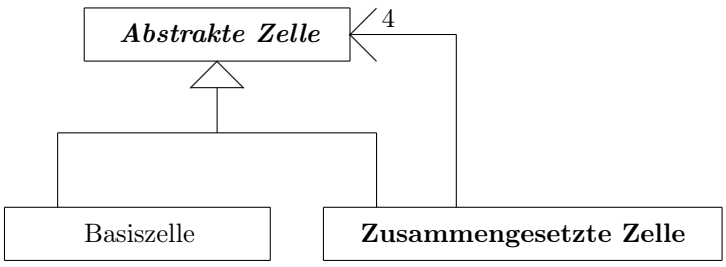


Abbildung 5.3: UML-Klassendiagramm des Kompositums

Algorithmus 5.3 (Berechnen von repräsentativen Partikeln)

1. Berechne für jede in einer zusammengesetzten Zelle enthaltenen Basiszelle i mit $i = 1, \ldots, 4$ die Summe x_i der Partikelvaliditäten in dieser Zelle. Es sei s die Summe der Validitäten aller Partikel und r die Rekursionstiefe. Falls

   - $x_i \leq 0,05 \cdot s$ oder $r = 4$: Mache nichts für diese Basiszelle.
   - $x_i > 0,05 \cdot s$ und $r < 4$: Die Basiszelle wird in vier Basiszellen aufgeteilt, welche von einer neu erzeugten zusammengesetzten Zelle verwaltet werden. Die neue zusammengesetzte Zelle wird an die Stelle der alten Basiszelle eingefügt. Alle Partikel der alten Basiszelle werden entsprechend ihrer Position auf dem Feld einer der neuen Basiszellen zugeordnet. Für jede der neu entstandenen Basiszellen führe Schritt 1 aus.

2. Berechne ein repräsentatives Partikel für die 12 Zellen mit dem höchsten Wert Γ durch ein gewichtetes Mittel der in ihr enthaltenen Partikel. Γ berechnet sich aus der Summe $\sum \mathrm{P}$ der Validitäten der Partikel in der Zelle und der Tiefe der Zelle im Datenstrukturbaum r: $\Gamma = \sum \mathrm{P} \cdot r^2$.

5.3.2 Reagieren auf den Ausfall von neuen Informationen

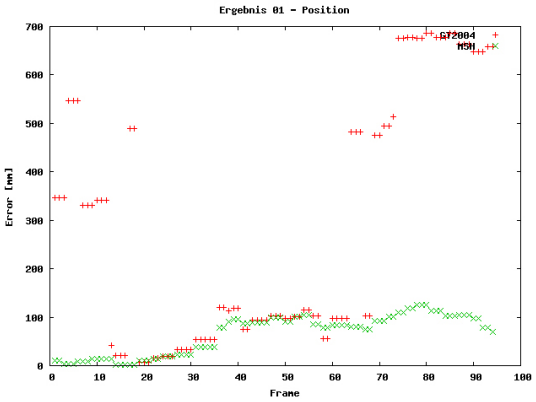
Der GT2005TeamBallLocator wurde anfänglich so konzipiert, dass es notwendig war, in jedem Durchlauf neue Daten von Mitspielern zu empfangen. Geschah dies nicht, so wurden alle Informationen über die Ballposition, die man von einem Mitspieler im vorherigen Durchlauf bekommen hatte, verworfen. Insbesondere, wenn der WLAN-Empfang kurzzeitig nicht verfügbar ist und der Roboter den Ball selber lange nicht gesehen hat, bereitet dies große Probleme. Um dieses zu umgehen, wurde folgende Reaktion auf den Ausfall des WLANs implementiert.

Die Partikel werden so gespeichert, dass man sie jederzeit einem Roboter zuordnen kann (anhand der Spielernummer). Werden von einem Roboter keine Partikel empfangen, so werden die alten Partikel zuerst einem Resampling unterzogen. Das heißt, es wird aus jedem Partikel zwei Partikel mit jeweils der Hälfte der ursprünglichen Validität gemacht, so lange, bis man 12 oder mehr Partikel hat. Hat man mehr als 12 werden die Partikel mit geringster Validität entfernt. Danach werden alle Partikel gaußverteilt gestreut. Die Standardabweichung für die Streuung hängt von der Anzahl der TeamBallLocator-Durchläufe ab, in denen keine Partikel von diesem Roboter empfangen wurden. Jedes Mal, wenn ein Partikel gestreut wird, wird die Validität verkleinert.

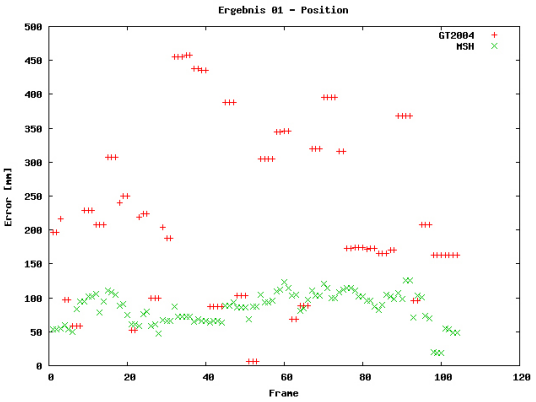
Diese Maßnahme führt dazu, dass die *CommunicatedBallPosition* immer zumindest in der Nähe der tatsächlichen Ballposition bleibt.

5.3.3 Diskussion der Güte des neuen TeamBallLocators

In Testspielen hat sich gezeigt, dass sich die Roboter mit dem GT2005TeamBallLocator viel besser auf der Höhe des Balls positionieren, als mit dem GT2004TeamBallLocator. Beim Visualisieren der *CommunicatedBallPosition* fällt auch auf, dass



(a) Der Roboter, von dem die Messdaten stammen, steht so, dass er den Ball nicht sehen kann. Die anderen Roboter können den Ball sehen, um sich zu lokalisieren gucken sie aber teilweise zu Landmarken. Dann sehen sie den Ball nicht.



(b) Der Roboter, von dem die Messdaten stammen, steht so, dass er den Ball nicht sehen kann. Ein anderer Roboter hat eine schlechte Farbtabelle und erkennt im gelben Tor fälschlicherweise einen Ball. Die anderen beiden Roboter sehen den Ball korrekt.

Abbildung 5.4: *CommunicatedBallPosition* bei künstlichen Situationen

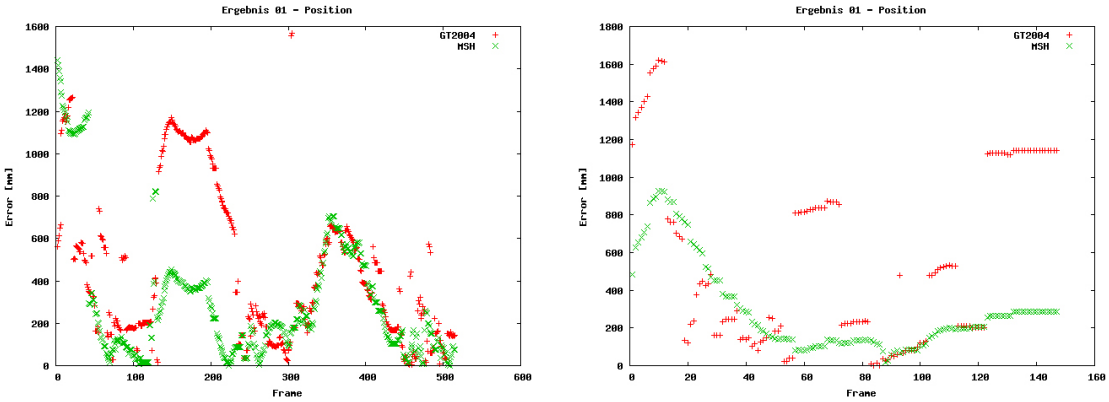
die Position stabiler ist. Das heißt, die Position ändert sich nicht in einem Durchlauf um eine große Distanz und ist im nächsten Durchlauf wieder auf der alten Position.

Ähnlich wie im BallLocator (vergleiche Kapitel 4.3.3) wurden auch für den neu entwickelten TeamBallLocator Vergleiche zur alten Lösung gemacht. Auch hier galt die CeilingCam als Referenzsystem und es wurde ein „ComboTeamBallLocator“ benutzt, welcher gleichzeitig den GT2004- und GT2005TeamBallLocator ausführt und die Ergebnisse mit denen der CeilingCam vergleicht.

In Abbildung 5.4 sind die Ergebnisse zweier Tests zu sehen, in denen alle Roboter einer Mannschaft und der Ball speziell positioniert wurden. In diesen künstlich erzeugten Situationen zeigt sich der Vorteil des neuen TeamBallLocators deutlich. Vor allem zeigt sich die Stabilität der Ballposition. Während beim GT2004TeamBallLocator die Position stark springt, ist die Position beim GT2005TeamBallLocator stabil und nur in wenigen Frames vernachlässigbar schlechter.

In weiteren Tests musste der neue TeamBallLocator seine Güte in typischen Spielsituationen unter Beweis stellen. In Abbildung 5.5 sind die Ergebnisse von zwei Tests dargestellt. Es wurde jeweils ein ganz normales Spiel ausgeführt, also vier-gegen-vier. In Abbildung 5.5(a) lässt sich um den 200sten Frame der Vorteil der *RobotPoseCollection* erkennen. In dieser Situation konnte nur ein einziger Roboter aus der Mannschaft den Ball gut sehen. Der Roboter rennt zum Ball und verhakt sich, dabei sieht er lange keine Landmarken und seine Lokalisierung wird ungenau. Ein zweiter Roboter der Mannschaft kann nur einen Teil des Balls erkennen, da ein Gegner die Sicht teils verdeckt. Die Informationen beider Roboter zusammen genutzt – dies entspricht dem Ansatz des GT2005TeamBallLocator – ergibt eine deutlich genauere *CommunicatedBallPosition* als der Ansatz des GT2004TeamBallLocators.

In Abbildung 5.5(b) zeigt sich, dass auch im Spiel beim neuen Ansatz die Position



(a) Eine typische Spielsituation. Die Messdaten stammen vom Torwart.

(b) Eine weitere typische Spielsituation. Auch hier stammen die Messdaten vom Torwart.

Abbildung 5.5: *CommunicatedBallPosition* bei Spielsituationen

des Balls stabiler berechnet wird.

5.4 Ausblick

Die Nutzung des TeamBallLocators im Verhalten ist im Moment noch gering. Ein Grund dafür ist, dass die Entscheidung, wann man die Informationen des BallLocators und wann die des TeamBallLocators nutzen sollte, nicht sehr einfach ist. Der TeamBallLocator ist in den meisten Fällen auf Grund der Abhängigkeit zur korrekten Berechnung der Roboterposition ungenauer. Daher wird erst dann, wenn der Roboter längere Zeit den Ball nicht gesehen hat, auf die *CommunicatedBallPosition* zurückgegriffen.

Das Ziel ist es, den TeamBallLocator so zu gestalten, dass er BallLocator und TeamBallLocator vereint. Man muss dann nur noch eine Ballposition abfragen. Es ist nicht wichtig zu wissen, ob sie nur vom Roboter selber berechnet oder ob die Informationen von anderen Robotern mitbenutzt wurden.

Um dies zu erreichen, wird der Ansatz verfolgt, für den TeamBallLocator ein eigenes Partikel-Filter-System zu implementieren. Dieses basiert auf gesendeten Perzepten, Odometrie-Daten und Roboterpositionen. Mit diesen Informationen werden dann auf jedem Roboter die Partikel aktualisiert. Dieser Ansatz sollte noch stabiler und genauer als der Ansatz des GT2005TeamBallLocators die Ballposition berechnen. Er ist allerdings auch rechenzeitaufwändiger. Das wird jedoch dadurch kompensiert, dass anstelle von zwei Modulen (BallLocator und TeamBallLocator) nur noch ein Modul vorhanden ist.

Weiterhin kann auch alleine auf der Grundlage der Daten von anderen Robotern eine Geschwindigkeit des Balls berechnet werden. Beim GT2005TeamBallLocator war es nicht nötig, diese zu berechnen. Die Benutzung der übertragenden Geschwindigkeiten aus dem BallLocator hatte zudem für den TeamBallLocator keine brauch-

baren Ergebnisse geliefert. Ein Grund dafür ist die Netzwerklatenz, die dazu führt, dass die empfangenen Daten nicht denselben Bezugszeitpunkt haben. Die Geschwindigkeitsberechnung reagiert im Gegensatz zur Positionsberechnung sehr empfindlich auf derartige Ungenauigkeiten, so dass an dieser Stelle noch Verbesserungen nötig sind.

Das Partikel-Filter-System soll ähnlich wie im GT2005BallLocator gestaltet werden. Das Umrechnen der Perzepte anderer Roboter von Roboterkoordinaten in Feldkoordinaten wird von der Berechnung für Partikel im GT2005TeamBallLocator (siehe Kapitel 5.3.1.1) adaptiert.

6 Selbstlokalisierung

Das Wissen über die Position (x, y) und Ausrichtung (θ) der Roboter auf dem Spielfeld (*RobotPose* (x, y, θ)) ist grundlegend für koordiniertes Handeln und damit für die Entwicklung eines konkurrenzfähigen Verhaltens. Aufgabe des SelfLocators ist also die Bestimmung der *RobotPose* in den unterschiedlichen Spielsituationen (Selbstlokalisierung). Hierfür stehen die durch den ImageProcessor erkannten Feldlinien, Landmarken und Tore (im Folgenden Perzepte genannt), sowie das Wissen über die ausgeführte Bewegung des Roboters (Odometrie) zur Verfügung. Im Gegensatz zu den Feldlinien lassen sich die Tore und Landmarken auf Grund ihrer Farbe eindeutig identifizieren (siehe Kapitel 1.3).

6.1 Problembeschreibung

Die aktuellen Regeländerungen (siehe Kapitel 1.3) stellen bezüglich der Selbstlokalisierung eine neue Herausforderung dar, da auf Grund der neuen Positionen der Landmarken, der Vergrößerung des Spielfeldes und des Wegfalls der Banden zumeist weniger Informationen verfügbar sind. Zusätzlich kommt einer präzisen Selbstlokalisierung wegen der Einführung der Auslinie sowie der „leaving field"-Regel eine noch größere Bedeutung zu. Daher ist das Ziel eine Verbesserung der existierenden Lösung (GT2004SelfLocator).

Allgemein soll der SelfLocator in möglichst allen Spielsituationen eine akzeptable Selbstlokalisierung gewährleisten, was zu teilweise widersprüchlichen Anforderungen führt:

- **Präzision:** Sicheres Handeln an den Auslinien, Vermeidung von „illegal defender" Situationen und die Modellierung teamweiter Positionsinformationen (z.B. TeamBallLocator Kapitel 5.1) erfordern eine hohe Präzision.
- **RobotPose Stabilität:** Eine sich schnell ändernde Hypothese der *RobotPose* kann durchaus durch die erzeugten Perzepte gerechtfertigt sein, ist aber für die Verhaltensplanung hinderlich. Eine stabile, aber im Durchschnitt etwas ungenauere Hypothese ist hier vorzuziehen.
- **schnelle Relokalisierung:** Im Spiel werden Roboterpositionen durch den Schiedsrichter verändert und auf dem Spielfeld existieren symmetrische Positionen, die nicht unterscheidbar sind. Es ist also wichtig, dass der SelfLocator in der Lage ist bei widersprüchlichen Perzepten schnell eine neue, bessere Hypothese für die *RobotPose* zu bestimmen.

- **geringe Laufzeit:** Da die Rechenleistung der AIBO Roboter beschränkt ist, muss jede Lösung entsprechende Laufzeitbeschränkungen berücksichtigen.

6.2 Die sequenzielle Monte-Carlo-Methode zur Lokalisierung

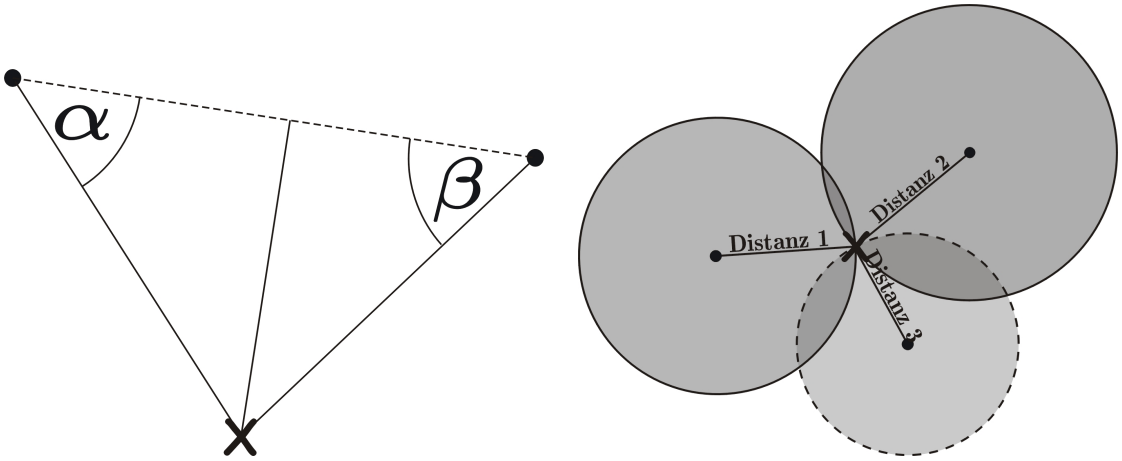
Die bestehende Lösung, der GT2004SelfLocator basiert auf einem speziellen Verfahren um die *RobotPose* möglichst unabhängig von Störeinflüssen zu bestimmen. Dieses soll hier kurz beschrieben und gegen andere Verfahren abgegrenzt werden.

6.2.1 Vorteile gegenüber anderen Verfahren

Zur Bestimmung der Position des Roboters auf dem Spielfeld stehen verschiedene Ansätze zur Verfügung, beispielsweise geometrische Verfahren (Abbildung 6.2.1), Kalman-Filter (siehe [22]) oder die sequenzielle Monte-Carlo-Methode (SMC), auch Partikelfilter genannt. Um geometrische Verfahren wie Triangulation und Trilateration nutzen zu können, sind stets mehrere Richtungen bzw. Entfernungen nötig. Dies ist mit der im AIBO verfügbaren Kamera (Öffnungswinkel ca. 57°) in einem Frame im Allgemeinen aber nicht möglich. Eine Korrektur der Perzepte auf Basis der Odometrie über mehrere Frames ermöglicht es zwar für die Triangulierung hinreichend viele Informationen zu sammeln, führt aber auch zu einem großen Einfluss der Odometrie auf die Genauigkeit der *RobotPose*. Da die Odometrie und Wahrnehmung im Spiel (z.B. durch Kollisionen mit anderen Robotern) häufig stark mit Rauschen behaftet ist und darüber hinaus die Triangulierung nicht von uneindeutigen Informationen (Linien) profitieren kann, eignet sich dieses Verfahren nur schlecht für die Selbstlokalisierung in der „Sony Four-Legged League“.

Weniger durch Rauschen beeinflusst wird der Kalman-Filter, der auch eine Verwendung uneindeutiger Informationen zulässt. Der Kalman-Filter ist ein iteratives Verfahren, das auf Grund aktueller Eingangsdaten einen zu erwartenden Ausgangswert ermittelt, diesen mit dem tatsächlichen (gemessenen) Ausgangswert vergleicht und anhand der Differenz die Ausgangswertabschätzung verbessert. Hierbei wird die Annahme gemacht, dass das Rauschen des beobachteten Prozesses gaußverteilt ist. Dies ist ein Nachteil des Kalman-Filters in der Anwendung zur Selbstlokalisierung, da die Verteilung der Aufenthaltswahrscheinlichkeit zumeist nicht gaußverteilt ist. Abbildung 6.2 zeigt dies am Beispiel einer gesehenen Landmarke. Ein weiterer Nachteil ist die Tatsache, dass der Kalman-Filter keine multimodale[1] Verteilungen abschätzen kann, die bei der Selbstlokalisierung häufig auftreten können (Abbildung 6.9). Die SMC ist ein Verfahren, das bei bezüglich der Rauschempfindlichkeit ähnlichen Vorteilen, wie denen des Kalman-Filters, nicht auf gaußsche, unimodale Verteilungen eingeschränkt ist.

[1] mehrere Maxima besitzend



(a) Triangulation (b) Trilateration

Abbildung 6.1: Geometrische Verfahren zur Positionsbestimmung

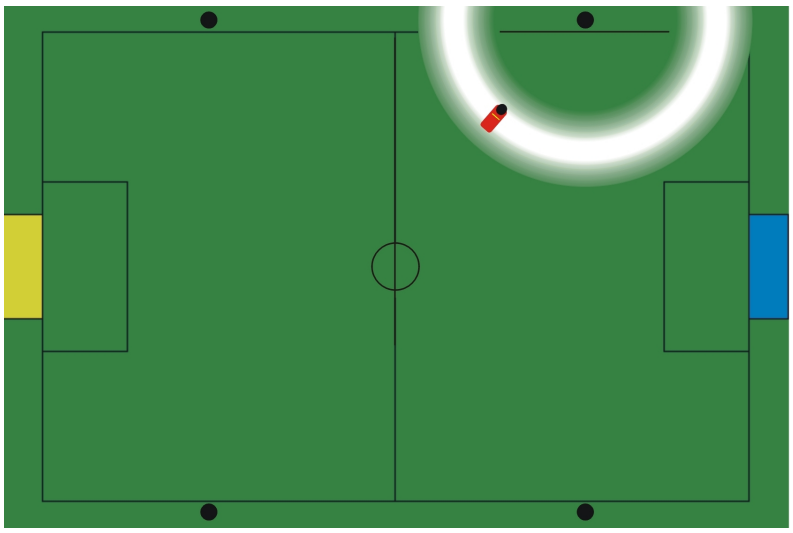


Abbildung 6.2: Aufenthaltswahrscheinlichkeit auf Grund einer Landmarke

6.2.2 Grundlagen

Aus den Beobachtungen zu einem beliebigen Zeitpunkt lässt sich für einen Bereich um einen Punkt l auf dem Spielfeld eine Aufenthaltswahrscheinlichkeit $P(L = l, l \in [l_0, l_1])$ bestimmen. Die Abhängigkeit der Aufenthaltswahrscheinlichkeiten von kleinen Umgebungen um den Ort l wird von der Wahrscheinlichkeitsdichtefunktion $p(l)$ beschrieben, deren Maximum den Ort beschreibt, an dem sich der Roboter mit der höchsten Wahrscheinlichkeit aufhält (siehe Gleichung (6.1)).

$$P(L = l, l \in [l_0, l_1]) = \int_{l=l_0}^{l=l_1} p(l)\mathrm{d}l \tag{6.1}$$

Da es auf Grund der im AIBO verbauten Kamera in der „Four-Legged-League", im Allgemeinen nicht möglich ist hieraus die Position des Roboters abzuleiten, werden frühere Informationen benötigt. Der *BeliefState* $B(L^t)$ vereint, ausgehend von allen möglichen Startpositionen, die Wahrscheinlichkeitsdichten, die unter Berücksichtigung aller vergangenen Beobachtungen o und Roboterbewegungen u für den Zeitpunkt t existieren (Gleichung (6.2)).

$$B(L^t) = \int_{l^0} p\left(L^t, L^0 | o^t, u^{t-1}, o^{t-1}, u^{t-2}, \ldots, o^1, u^0\right) \mathrm{d}l \tag{6.2}$$

Der *BeliefState* $B(L^t)$ ist also eine Wahrscheinlichkeitsdichtefunktion, die sich aus den Wahrscheinlichkeiten für jeden Ort und jeden Startzustand ergibt. Um das Maximum dieser Funktion berechnen zu können, muss zunächst der kontinuierliche Fall auf den diskreten Fall übertragen werden (Gleichung (6.3)).

$$\int_{a=a_0}^{a_1} P(A = a)\mathrm{d}a \overset{\text{geht über in}}{\longrightarrow} \sum_{a=a_0}^{a_1} \frac{P(A = a)}{n} \tag{6.3}$$

Allerdings ist auch die Berücksichtigung der kompletten Beobachtungs- und Bewegungshistorie nicht praktikabel, so dass man, eine statische Umgebung voraussetzend, die Markov-Eigenschaft[2] (Gedächtnislosigkeit) für den Prozess annimmt. Die Wahrscheinlichkeitsdichte des Ortes L hängt somit nur von der letzten Wahrscheinlichkeitsdichte und der letzten Bewegung ab, damit aber auch rekursiv von allen vergangenen Wahrscheinlichkeitsdichten und Bewegungen. Abbildung 6.3 demonstriert die Veränderung des *BeliefState* eines Roboters, der sich mit den bekannten Positionen dreier Türen entlang einer Linie lokalisiert. Zunächst wurden keine Beobachtungen gemacht, es gibt keine Maxima in der Wahrscheinlichkeitsdichtefunktion. Erkennt der Roboter die erste Tür, so ergibt sich an jeder Tür ein Maximum, da alle Türen mit gleicher Wahrscheinlichkeit unter den gegebenen Voraussetzungen (vorherige Dichte, Bewegung) gesehen werden können. Während der Bewegung des

[2] siehe Anhang D

Abbildung 6.3: Veränderung der Aufenthaltswahrscheinlichkeitsdichte

Roboters zur nächsten Tür, werden keine weitere Beobachtungen gemacht, aber die Ungenauigkeit der Bewegung führt zu breiteren Maxima. Die Erkennung der zweiten Tür schließlich lässt nur noch ein Maximum zu, denn nach der vorherigen Dichteverteilung und der absolvierten Bewegung sind die restlichen Positionen ausgeschlossen. Auf dieser Basis approximiert die SMC den *BeliefState*.

6.2.3 Funktionsweise

Die SMC zur Selbstlokalisierung nutzt zur Approximierung des *BeliefState* so genannte Samples oder Partikel. Jedes Partikel repräsentiert eine hypothetische *RobotPose* und besitzt eine Wahrscheinlichkeit, die Aufenthaltswahrscheinlichkeit. Die Dichte der Partikel mit ihren Wahrscheinlichkeiten in einem Bereich entspricht der Wahrscheinlichkeitsdichte an dieser Stelle, die gesamte Partikelverteilung repräsentiert den *BeliefState* (Abbildung 6.4).

Die Bestimmung der *RobotPose* mit Hilfe der SMC erfolgt schrittweise, wobei zu Beginn die Partikel gleichmäßig in Ort und Rotation verteilt sind. In jedem Frame werden die Partikel gemäß der Odometrie bewegt und bewegungs- sowie partikelwahrscheinlichkeitsabhängig gestreut; dann wird jedem Partikel seine Aufenthaltswahrscheinlichkeit zugewiesen, indem die von der Partikelposition und Rotation zu erwartenden Beobachtungen mit den Tatsächlichen verglichen werden.

In einem Resampling genannten Vorgang erhalten alle Partikel die gleiche Wahrscheinlichkeit, indem überdurchschnittlich wahrscheinliche Partikel vervielfältigt und entsprechend ein Teil der unterdurchschnittlich wahrscheinlichen Partikel entfernt

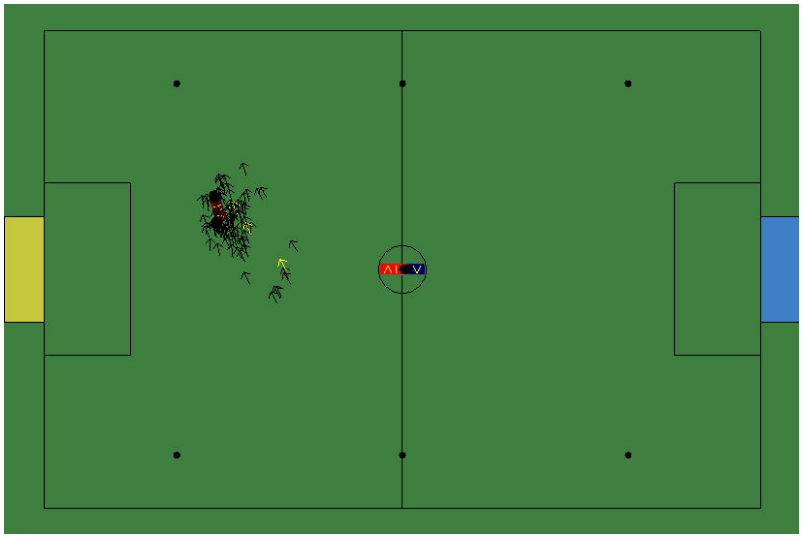


Abbildung 6.4: *BeliefState*, repräsentiert durch Partikel

wird. Des Weiteren wird ein Teil der unterdurchschnittlich wahrscheinlichen Partikel durch zufällig Positionierte ersetzt. Dieses Ersetzen durch zufällige Positionen ist eine zufällige (blinde) Suche nach weiteren Maxima im Suchraum, die parallel zur Verfolgung der existierernden Maxima stattfindet. Auf diese Weise versammeln sich nach wenigen Durchläufen die Partikel vor allem in der Nähe der Maxima der Wahrscheinlichkeitsdichtefunktion. Auf Grund der gleichen Wahrscheinlichkeit der Partikel nach dem Resampling entspricht nun die Dichte der Partikel der Wahrscheinlichkeitsdichte an den jeweiligen Stellen. Ähnlich wie auch der Kalman-Filter verringert die SMC den Einfluss von Rauschen durch einen Abgleich von zu erwartenden und tatsächlichen Beobachtungen, darüber hinaus ist es aber möglich nicht-gaußsche und multimodale Verteilungen anzunähern, da sich die Partikel frei entsprechend der Aufenthaltswahrscheinlichkeit verteilen können. (Abbildung 6.5). Abschließend wird die *RobotPose* bestimmt, indem der Punkt der höchsten Wahrscheinlichkeitsdichte mithilfe eines Clustering-Verfahrens approximiert wird, das den Bereich der höchsten Partikeldichte bestimmt. Weitere Informationen zu Partikelfilter finden sich beispielsweise bei [10].

6.2.4 Partikelfilter zur Selbstlokalisierung: der GT2004SelfLocator

Der GT2004SelfLocator (siehe [16]) implementiert das oben beschriebene Verfahren mit folgender Abfolge:

1. Aktualisierung der Partikel mittels Odometrie (Motion-Update)
2. Wahrscheinlichkeitsberechnung anhand der Perzepte (Observation-Update)
3. Resampling

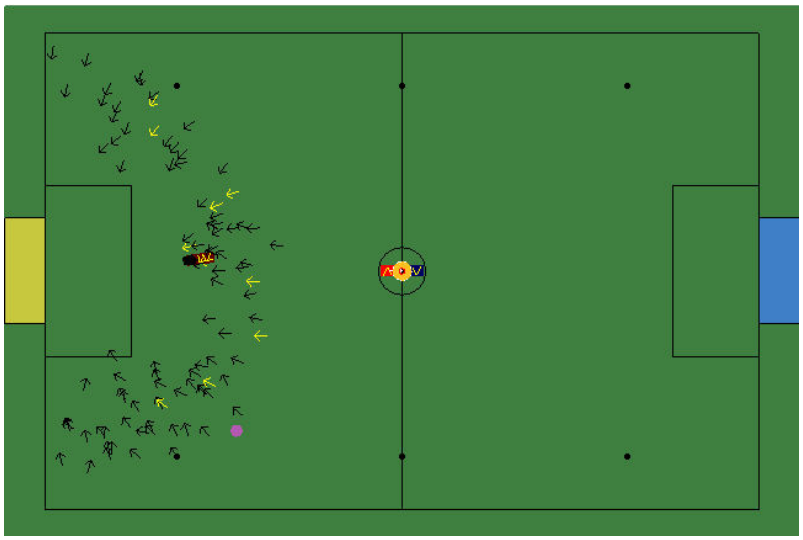


Abbildung 6.5: Die Partikel verteilen sich gemäß der Aufenthaltswahrscheinlichkeit

4. Bestimmung der *RobotPose* durch Clustering

Das Motion-Update vereint hierbei sowohl die Anwendung des Odometrie bedingten Rauschens als auch die zufällige Streuung abhängig von der Partikelwahrscheinlichkeit. Der GT2004SelfLocator nutzt eine „Sensor Resetting“ genannte Methode (siehe [12]), um die Anzahl der benötigten Partikel zu reduzieren. Hierbei werden für die zu ersetzenden Partikel keine zufälligen Positionen gewählt, sondern solche, die sich aus den Beobachtungen errechnen lassen (*Templates*). So können für eine einzelne Landmarke die möglichen Positionen auf den in Abbildung 6.2 gezeigten Bereich beschränkt werden. Dies führt zusammen mit dem Resampling dazu, dass im Fall einer Fehllokalisierung die (wenigen) Partikel sich schneller im Bereich der tatsächlichen *RobotPose* häufen, als es bei einer blinden (zufälligen) Suche der Fall wäre. Ein Nachteil bei zufällig auftretenden falschen Beobachtungen entsteht durch dieses Verfahren nicht, da das Resampling sicherstellt, dass falsch eingefügte Partikel auf Grund der geringen Wahrscheinlichkeit schnell wieder entfernt werden.

Eine weitere Besonderheit des GT2004SelfLocators ist die Modellierung der einzelnen Partikel. Anstatt eine einzelne Wahrscheinlichkeit zu verwalten, wird für jeden Perzepttyp eine Wahrscheinlickeit verwaltet, deren Wert sich für jedes ausgewertete Perzept nur um einen bestimmten Wert erhöhen bzw. erniedrigen kann. Am Ende eines jeden Frames wird die Wahrscheinlichkeit des Partikels durch Multiplikation der Einzelwahrscheinlichkeiten bestimmt. Dieses Filtern der Partikelwahrscheinlichkeit reduziert den Einfluss falscher Perzepte bei geringer Partikelanzahl.

Zur Berechnung der Wahrscheinlichkeit p der Partikel (*Observation-Update*) werden im GT2004SelfLocator stets Winkeldifferenzen benutzt. Hierbei wird der Winkel zu einem Perzept mit dem, ausgehend von der *RobotPose* des jeweiligen Partikels, erwarteten Winkel verglichen (Abbildung 6.6).

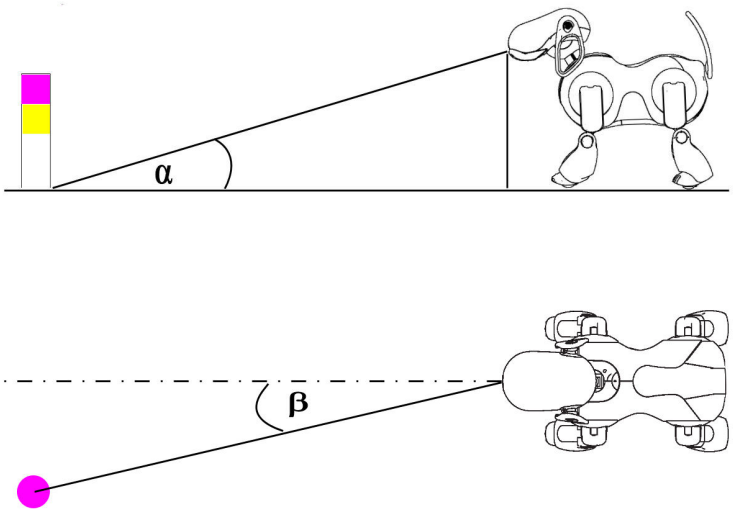


Abbildung 6.6: Perzepte sind im SelfLocator winkelbasiert

Die unterschiedlichen Perzepte können unterschiedliche Genauigkeiten aufweisen und zudem von unterschiedlicher Bedeutung für das Spiel sein (die Ausrichtung zum Tor ist zum Beispiel von besonders hoher Wichtigkeit). Darüber hinaus können Perzepte unterschiedlich viele Winkel beinhalten, welche auf Grund der Beschaffenheit des beobachteten Objekts unterschiedlich genau berechnet werden. Daher wird für jede benutzte Winkeldifferenz δ_i ein Gewicht w_i vergeben (Gleichung (6.4)).

$$\begin{aligned} \delta_i &= \alpha_{gemessen} - \alpha_{berechnet} \\ p &= e^{w_1 \cdot (-\delta)_1^2} \cdot e^{w_2 \cdot (-\delta)_2^2} \cdot \ldots \end{aligned} \tag{6.4}$$

Im Resampling-Schritt werden die Partikel zur Ersetzung durch *Templates* ausgewählt, deren Wahrscheinlichkeit einen zufällig gewählten Bruchteil der Durchschnittswahrscheinlichkeit der Partikelmenge unterschreitet.

Das Clustering unterteilt den Suchraum in $10 \times 10 \times 10$ Würfel. Dieser wird nach dem $2 \times 2 \times 2$ Unterwürfel mit den meisten Partikeln durchsucht, da nach dem Resampling alle Partikel gleichwahrscheinlich sind. Abschließend wird im Bereich der höchsten Partikeldichte die *RobotPose* durch Mitteln der Partikel bestimmt, was für die Winkelachse auf Grund ihrer Zirkularität gemäß Gleichung (6.5) erfolgt[3].

$$\theta_{mittel} = \text{atan2}\left(\sum_i \sin(\theta_i),\ \sum_i \cos(\theta_i)\right) \tag{6.5}$$

[3]atan2(y,x) ist der Winkel des Vektors (x,y)

Da der GT2004SelfLocator und seine Umgebung aus der Vereinigung mehrerer Ideen entstanden ist, ist seine Struktur aus softwaretechnischer Sicht nicht zufriedenstellend. Alle logischen Module befinden sich in einer Datei und die Benennung der Perzepte ist nicht eindeutig; mitunter werden verschiedene Perzepte aufeinander abgebildet. Beispielsweise wird eine Feldlinie in y-Richtung und ein blauer Roboter in einer Klasse durch den gleichen Wert repräsentiert.

6.3 Diskussion von Verbesserungsmöglichkeiten

Nach den Vorüberlegungen stand schnell fest, dass ein Partikelfilter ähnlich dem GT2004SelfLocator zum Einsatz kommen würde. Es musste allerdings entschieden werden, ob die existierende Lösung verbessert oder eine neue Lösung implementiert werden sollte. Für eine Neuimplementierung spricht die schlechte Struktur des GT2004SelfLocators, die die Erweiterung und Pflege des Moduls deutlich erschwert. Nachteilig ist, dass so Verbesserungen erst relativ spät nutzbar werden. Daher haben wir uns entschieden, ein neues Modul auf Basis des GT2004SelfLocators durch schrittweise Umstrukturierung (Refaktorierung) zu erzeugen. So können sowohl Verbesserungsansätze sofort implementiert und getestet werden, als auch langfristig ein Modul implementiert werden, das sich zur weiteren Optimierung und Pflege besser eignet, als das Existierende.

6.3.1 Ziel der Refaktorierung

Das Ergebnis der Refaktorierung soll ein Modul sein, das sich leicht warten und auf Änderungen in anderen Modulen anpassen lässt. Dafür ist es wünschenswert, die Zahl der Parameter zu verringern und die Größe der Klassen zu beschränken. Es soll gewährleistet werden, dass jederzeit eine Version des SelfLocators existiert, die im Vergleich zum GT2004SelfLocator zumindest keine Verschlechterung darstellt.

6.3.2 Vorgehen zur Verbesserung der Lokalisierung

Da die zuvor beschriebenen neuen Anforderungen an die Lokalisierung auf einem Mangel an verfügbaren Informationen beruhen, stützte sich die Verbesserung des SelfLocators zunächst auf die Nutzung bisher nicht berücksichtigter Informationen. Auch hierbei ist zu gewährleisten, dass keine Verschlechterung im Vergleich zum GT2004SelfLocator eintritt. Daher soll die Integration neuer Informationen schrittweise erfolgen und regelmäßig durch Tests überprüft werden. Die Verbesserung primär auf die Optimierung der Parameter oder des Partikelfilters zu stützen erscheint zunächst nicht sinnvoll, da der GT2004SelfLocator bereits in hohem Maße angepasst ist und mit den verfügbaren Informationen gute Ergebnisse liefert (siehe [16]).

6.4 Umsetzung

Da wir uns entschieden haben, die neue Lösung auf Basis des GT2004SelfLocators zu entwickeln, erfolgte die Umsetzung in mehreren Phasen, in denen jeweils ein Ansatz zur Verbesserung verfolgt und bewertet wurde.

6.4.1 Phase 1: Refaktorierung des SelfLocators

Zur Reduktion der Modulgröße wurden folgende Teile des SelfLocators identifiziert: der Partikelfilter, das Sample, die *Template*-Erzeugung sowie das Bewegungs- und Beobachtungsmodell. Als besonders leicht zu trennen stellten sich dabei das Sample und die *Template*-Erzeugung heraus. Auf die Trennung der weiteren Module wurde zunächst verzichtet, da die Modulgröße nach der Trennung besagter Teile hinreichend klein war. Weiterhin ist die Zuordnung der neuen Module nicht vollkommen klar, denn Teile des Beobachtungsmodells lassen sich dem ImageProcessor zuordnen und Teile des Bewegungsmodells der WalkingEngine. Die Klasse zur Kapselung der linienartigen Beobachtungen, das LinienPerzept, wurde, wie bereits angedeutet, in wenig nachvollziehbarer Weise gebraucht, um zusätzliche Perzept-Typen ohne Anpassung der Klasse zu verwenden. Da dies zu ändern aber tiefgreifende Veränderung mehrerer Module zur Folge hätte, wurde der SelfLocator durch eine Abbildung des LinienPerzept auf eine eigene Repräsentation gegen das LinienPerzept gekapselt. Die Parameter des SelfLocators (siehe Kapitel 6.2.4) sind im GT2004SelfLocator nicht zentral angeordnet und müssen bei Änderungen der Perzepte neu geschätzt werden, da sie nicht aus Messungen ermittelt werden können. Die neue Lösung hierfür sammelt die Parameter w zentral und spaltet diese in einen messbaren (Varianz v) und einen zu schätzenden (Gewicht g) Teil auf (Gleichung (6.6)).

$$w = \frac{g}{v} \tag{6.6}$$

Hierdurch sinkt die Zahl der zu schätzenden Parameter auf die Anzahl der genutzten Perzepte, während zuvor für jede Dimension eines Perzepts ein einzelner Parameter zu schätzen war. Anpassungen, die auf Grund einer Veränderung im Messverfahren (zum Beispiel eine Verbesserung der Genauigkeit) notwendig werden, können nun duch Messungen vorgenommen werden und die zu schätzenden Parameter entsprechen stärker der Idee eines Gewichts für ein Perzept. Allerdings wird hierbei auch das Wissen über die Anzahl der Dimensionen eines Perzepts benötigt, da die durchschnittliche Wahrscheinlichkeit, die für ein Perzept vergeben wird, bei einer Erhöhung der Anzahl der Dimensionen sinkt. Die so erreichte Verringerung der Zahl der nicht messbaren Parameter kann in Zukunft bei einer automatischen Approximation dieser hilfreich sein. Für den SelfLocator wurden darüber hinaus parallel mehrere Versionen angelegt: eine sicher funktionierende Basisversion, eine getestete verbesserte Version und eine experimentelle Version. So konnte sichergestellt werden, dass Fortschritte stets verifiziert werden konnten, und gleichzeitig immer (mindestens) eine funktionsfähige Version existiert. Zur Verifikation wurde

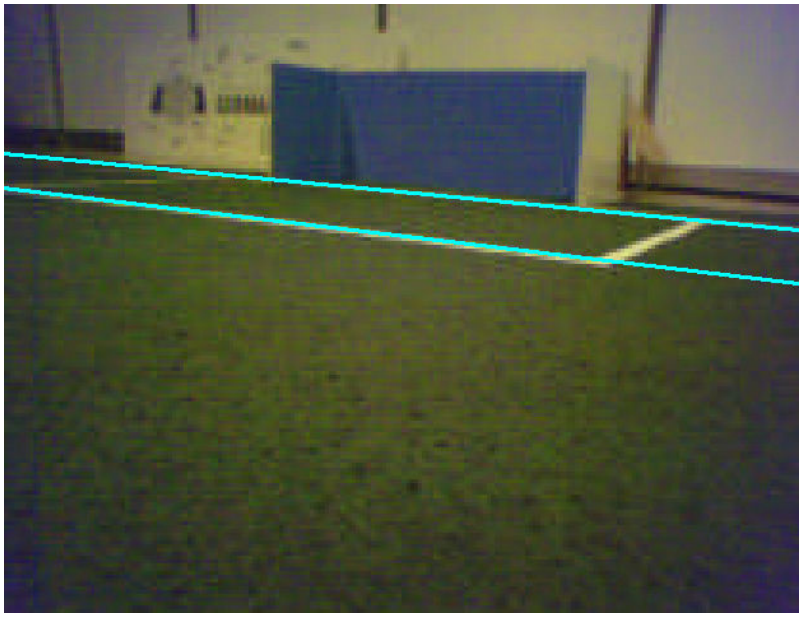
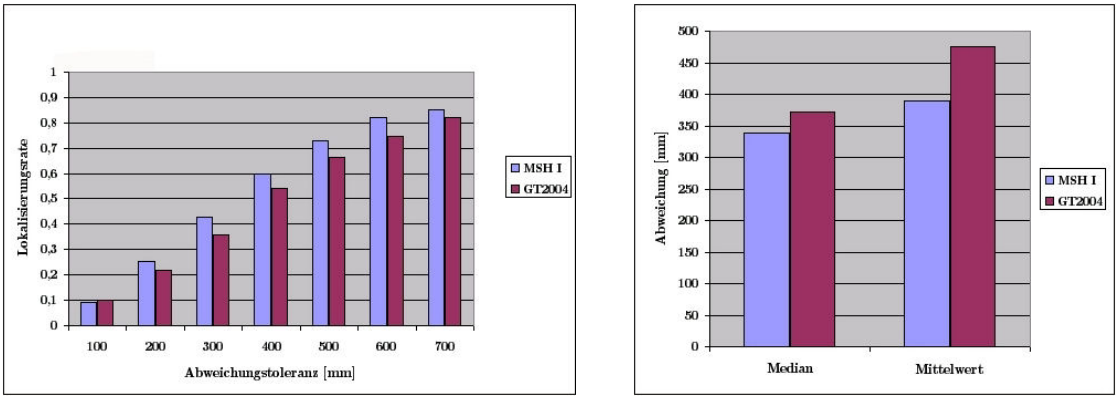


Abbildung 6.7: Die Linien des Strafraums werden nicht alle erkannt

ein so genannter „ComboSelfLocator“ implementiert, der in jedem Frame mehrere (unterschiedliche) SelfLocator-Instanzen ausführt und deren Ergebnisse mit denen der CeilingCam vergleicht. Auf diese Weise kann festgestellt werden, welche Instanz die besseren Ergebnisse liefert, indem man den Roboter einige Zeit unterschiedliche Positionen auf dem Spielfeld anlaufen lässt.

6.4.2 Phase 2: Neue Perzepte: Linienkreuzungen und der Mittelkreis

Bisher wurden zur Selbstlokalisierung neben den Toren und Landmarken lediglich Punkte auf Linien verwendet. In jedem Frame wurden bis zu 3 Punkte ausgewertet und Punkten einer Linien-Tabelle zugeordnet. Als zusätzliche Informationen bieten sich auffällige Konstellationen von Linien an, wobei hier nur relativ kleine Bereiche in Frage kommen, da beispielsweise schon der Strafraum nicht sicher mit allen Linien erkannt wird (Abbildung 6.7). Übrig bleiben daher Kreuzungspunkte von Linien und der Mittelkreis. Der Mittelkreis ist dabei ein besonders aussagekräftiges Merkmal, da er durch die Richtung der Mittellinie die möglichen Positionen für den Roboter auf 2 einschränkt (Abbildung 6.9). Allerdings ist bei der Erkennung des Mittelkreises die Rate der Fehlerkennungen aus diesem Grund besonders kritisch. Gerade solche Fehlerkennungen sind aber sehr wahrscheinlich, da Bereiche wie die Toraußenwand mit dem Feldende und der Auslinie leicht als Kreis identifiziert werden können. Daher fokussierte sich die Entwicklung zunächst auf die Integration von Linienkreuzungspunkten. Hierbei entsteht der Vorteil für die Lokalisierung aus der geringeren Anzahl der Linienkreuzungen im Vergleich zu Linienpunkten. Eine Linienkreuzung vereint also die Informationen vieler Linienpunkte, so dass pro Frame nun mehr Information



(a) Häufigkeit einer Lokalisierung für eine gegebene maximale Abweichungstoleranz

(b) Median und Mittelwert der Positionsabweichung

Abbildung 6.8: Verbesserung durch Linienkreuzungen (MSH I)

ausgewertet werden kann als zuvor. Zusammen mit einer entsprechenden Anpassung der Parameter konnte die in Abbildung 6.8 gezeigte Verbesserung erreicht werden. Hierbei wurde untersucht, in welchem Anteil der Frames der Fehler der Selbstlokalisierung kleiner oder gleich einer gegebenen Abweichungstoleranz war und wie groß der Durchschnitt bzw. Median der Abweichung ist.

Da Linienkreuzungen lediglich die Informationen vieler Linienenpunkte zusammenfassen, ergibt sich für den MSH I SelfLocator nur eine geringfügige Verbesserung. Die Laufzeit des SelfLocators steigt durch die Auswertung von Linienkreuzungen zusätzlich zu den Linienpunkten von 4,1 ms auf 5,3 ms (10.000 Ausführungen der beiden SelfLocator-Instanzen im Combo-Locator).

6.4.3 Phase 3: Klassifizierte Linienkreuzungen

Generell ist bei der Bewertung von Perzepten ein Abwägen von Perzept-Rate, Aussagekraft der Perzepte und Fehlerquote erforderlich. Perzepte mit besonders hoher Aussagekraft haben typischerweise eine relativ geringe Perzept-Rate, denn es sind seltene Merkmale auf dem Spielfeld. Daher ist hier eine geringe Fehlerquote besonders wichtig, um einen Vorteil aus der Aussagekraft des Perzepts zu ziehen. Allerdings führt auf Seiten des ImageProcessors (siehe Kapitel 7.3.3) meist eine Senkung der Fehlerquote zu einer Senkung der Perzept-Rate, was dazu führen kann, dass das Perzept auf Grund zu geringer Häufigkeit praktisch keine Bedeutung für die Lokalisierung hat. Linienkreuzungen existieren auf dem Feld in L- oder T-Form, zudem lassen sich virtuelle Kreuzungspunkte aus Linienverlängerungen ermitteln (Abbildung 6.10). Die Linienverlängerungen sind in Abbildung 6.10 als gepunktete Linien dargestellt. Der Schnittpunkt einer Linienverlängerung und einer Feldlinie wird virtueller Kreuzungspunkt genannt, solange es sich nicht um eine L- oder T-Kreuzung handelt.

Bei der Erkennung der Kreuzungspunkte und deren Integration in den SelfLoca-

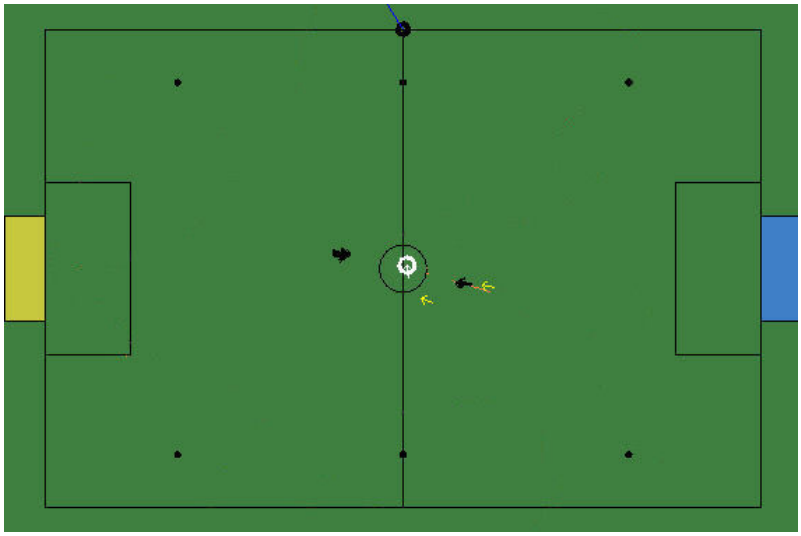


Abbildung 6.9: Der Mittelkreis schränkt die möglichen Positionen auf 2 ein

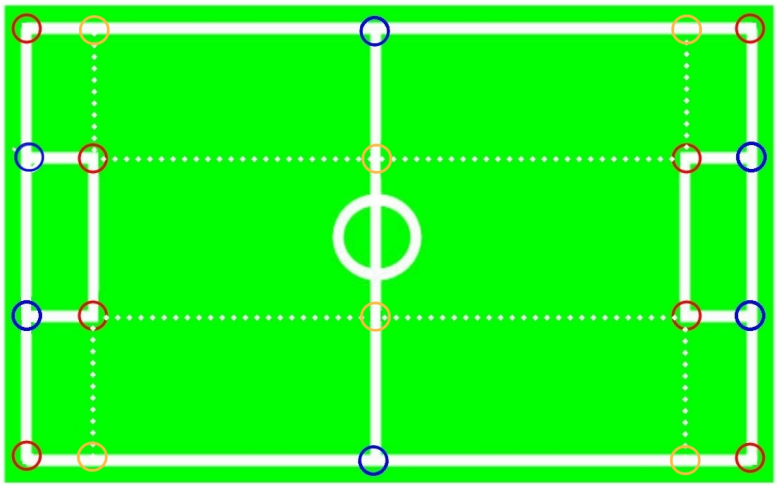


Abbildung 6.10: Linienkreuzungen in L- (rot) und T- (blau) Form, virtuelle Linienkreuzungen (gelb)

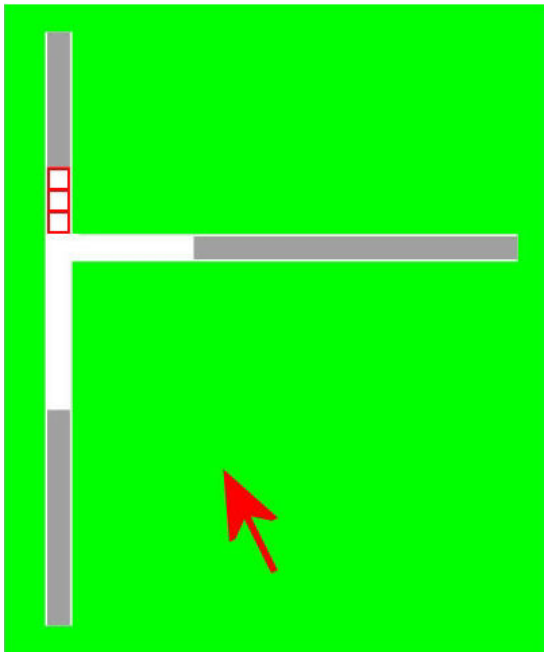
tor war oben beschriebene Abwägung vorzunehmen. Sich auf wenige genau erkannte Kreuzungspunkte zu beschränken, führt bei bewegtem Roboter und einer Reihe weiterer Roboter auf dem Spielfeld schnell dazu, dass nahezu keine Kreuzungspunkte erkannt werden und damit zu keiner messbaren Verbesserung der Lokalisierung führt. Allerdings verbessert die Nutzung aller möglichen Kreuzungspunkte (also auch in ihrer Form nicht Erkannter) die Lokalisierung nur wenig (siehe Kapitel 6.4.2).

Als Lösung wurde eine Klassifizierung von Linienkreuzungen entwickelt. Neben der Erkennung als L-, T- oder virtuellem Kreuzungspunkt existieren noch unbekannte Kreuzungspunkte, bei denen nicht entschieden werden kann, zu welcher der Kreuzungen der Punkt zuzuordnen ist. Auf diese Weise kann der größere Vorteil aus der Nutzung aussagekräftiger (klassifizierter) Kreuzungspunkte gezogen werden, ohne auf die geringfügige Verbesserung zu verzichten, die nicht klassifizierte (unbekannte) Linienkreuzungen bieten können.

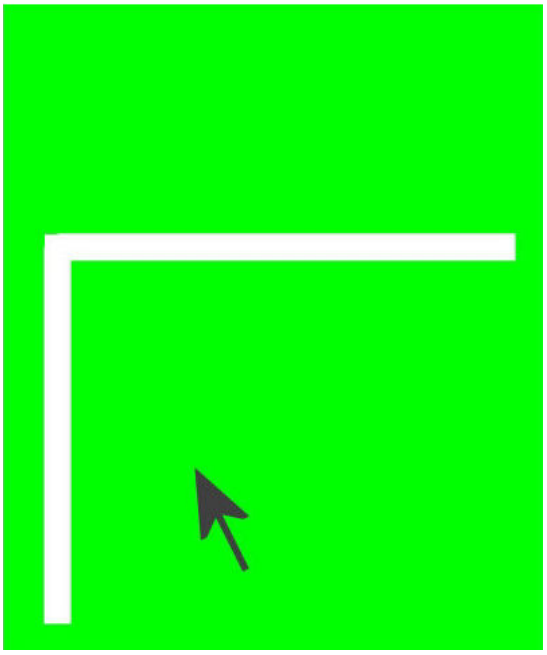
Die so entstandene Verbesserung lässt sich analog zu Kapitel 6.4.2 in Abbildung 6.12 ablesen und resultiert aus einer weiteren Erhöhung des Informationsgehalts der Linienkreuzungen.

Klassifizierte Linienkreuzungen bieten darüber hinaus aber einen weiteren Vorteil, denn für den Aktualisierungsvorgang (siehe Kapitel 6.2.4) kann ein klassifizierter Linienkreuzungspunkt mehr Informationen liefern, als die in seiner Erkennung beteiligten Linienpunkte. Abbildung 6.11 verdeutlicht dies am Beispiel eines Roboters (roter Pfeil) in der Nähe einer Linienkreuzung in T-Form. Die Entfernung, in der der Roboter Linien(punkte) erkennen kann, ist begrenzt (graue Linien). Daher werden die entfernten Linien der Linienkreuzung nur zu einem Teil genutzt und gehen in eine Bewertung einer Positionshypothese nur zu einem entsprechend kleinen Teil ein. Die durch den grauen Pfeil symbolisierte falsche Positionshypothese kann ohne die Nutzung von klassifizierten Linienkreuzungen nur durch die wenigen, rot markierten Linienpunkte falsifiziert werden. Die Aktualisierung mit allen in der dargestellten Situation verfügbaren Linienpunkten führt auf Grund des geringen Abstands der rot markierten Punkte von den aus Sicht der grau dargestellten Hypothese zu erwartenden Punkten nur zu einer geringfügig schlechteren Bewertung, als die der korrekten Position. Die Information über die Art der erkannten Linienkreuzung liefert dagegen eine viel stärkere Abweichung für die falsche Hypothese, denn die Entfernung zur nächsten möglichen Position einer entsprechenden Linienkreuzung ist wesentlich größer.

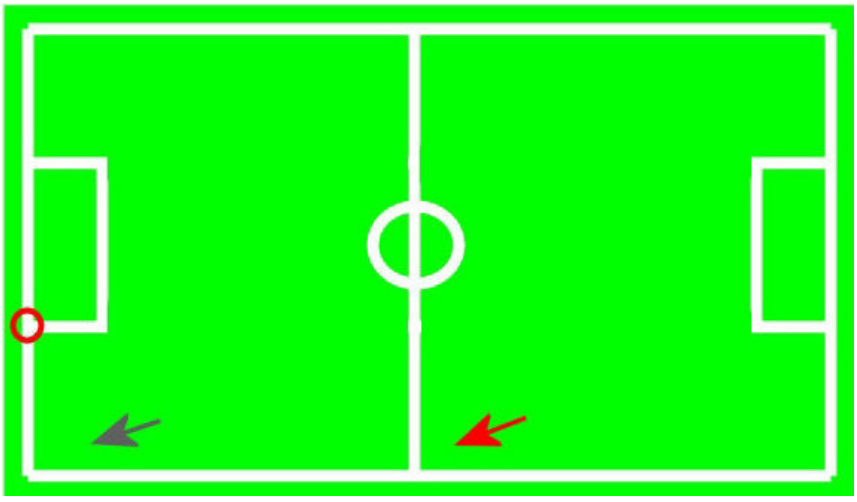
Nebenbei wurde die Nutzung des Mittelkreises in der experimentellen Version implementiert, um hierfür eine Testumgebung zur Verfügung zu haben. Bis zum Ende der Projektgruppe wurde der Mittelkreis zur Lokalisierung im Spiel allerdings nicht eingesetzt, da auf Grund der mittlerweile guten Selbstlokalisierung, das Risiko zu hoch erschien. Im Rahmen der „(almost)SLAM Challenge“ (Kapitel 10.1) zur Lokalisierung mit vorher unbekannten Landmarken wurde auf den GermanOpen eine angepasste Version des SelfLocators präsentiert, der – eine korrekte Anfangslokalisierung vorausgesetzt – nur auf Basis der Linieninformationen (hier mit dem Mittelkreis) die Position des Roboters verfolgen und Lokalisierungsabweichungen korrigieren kann.



(a) Linienpunkte werden nur im weiß dargestellten Bereich erkannt, daher ist der Unterschied zu einer Kreuzung in L-Form gering (rote Punkte)

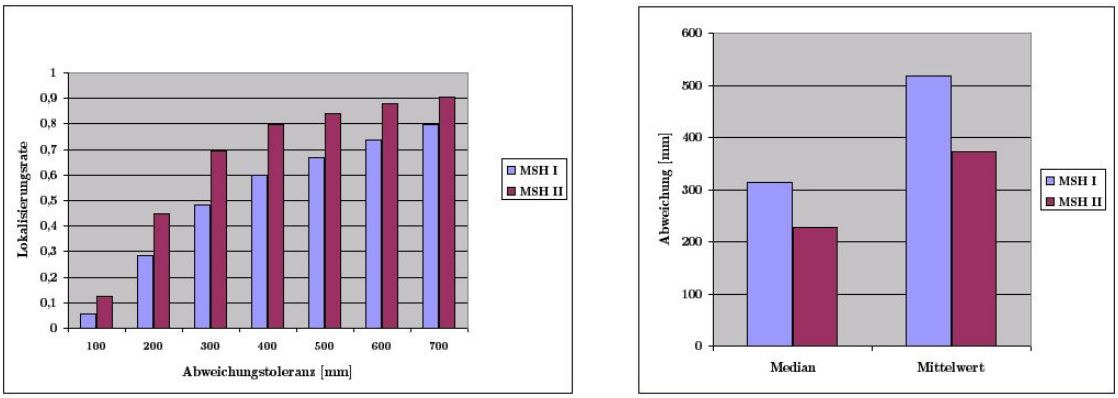


(b) hier wird auf das Verwenden von Linienkreuzungen vezichtet



(c) die grau markierte Position unterscheidet sich, auf Basis von Linienpunkten, von der Roten nur geringfügig

Abbildung 6.11: Die Klassifizierung von Linienkreuzungen birgt Vorteile



(a) Häufigkeit einer Lokalisierung für eine gegebene maximale Abweichungstoleranz

(b) Median und Mittelwert der Positionsabweichung

Abbildung 6.12: Verbesserung durch klassifizierte Linienkreuzungen (MSH II)

6.4.4 Phase 4: Optimierung nicht messbarer Parameter

Die Integration neuer Perzepte machte auch eine Anpassung der nicht messbaren Parameter des SelfLocators nötig, die zunächst von Hand (Schätzung) vorgenommen wurde. Um die so entstandenen Parameter zu optimieren und den Prozess der Parameteranpassung bei der Integration neuer Perzepte zu beschleunigen, wurde eine evolutionäre Strategie (siehe Anhang C) erprobt. Dabei ergaben sich vor allem zwei Probleme: zum Einen die Wahl der Bewertungsfunktion, zum Anderen die Menge der verfügbaren Trainingsdaten und die Anzahl der durchführbaren Iterationen. Eine Bewertungsfunktion zu finden, die die in Kapitel 6.1 genannten Kriterien geeignet berücksichtigt, erscheint schwierig, daher wurde zur Erprobung dieses Verfahrens zur Parameteroptimierung lediglich die Abweichung der Roboterposition benutzt. Die Hoffnung war, so unter vereinfachten Bedingungen zeigen zu können, dass das Verfahren prinzipiell zur Optimierung der Parameter geeignet ist.

Da wir mit Hilfe unserer Debug-Werkzeuge nicht in der Lage waren, die Eingangsdaten des SelfLocators so zu speichern, dass das Verhalten des SelfLocators mit diesen Daten reproduzierbar war, wurde versucht die Optimierung in Echtzeit auf dem Roboter vorzunehmen. Auf Grund der daraus resultierenden Laufzeitbeschränkungen ist es nicht möglich, einen Evolutionären Algorithmus mit einer größeren Population zu nutzen. Des Weiteren ist es schwierig zu entscheiden, wann hinreichend viele Daten eingegangen sind, um eine Aussage über die Güte einer Parameterwahl zu treffen. Dadurch, dass sich die verschiedenen Positionen auf dem Spielfeld bezüglich der verfügbaren Perzepte stark unterscheiden, lässt sich auch kein kurzer Ablauf finden, der für viele Situationen repräsentativ ist. Um diese Probleme zu umgehen, wurde eine 1+1 ES gewählt, die zur Bewertung der Güte der mutierten Parameter die Güte des gleichen Algorithmus mit bekannt guten Parametern auf den gleichen Daten heranzieht (ähnlich der Auswertung durch den Combo-SelfLocator). Gelingt es, einen Satz Parameter zu entwickeln, der diese bekannt guten Parameter bezüglich Positionsgenauigkeit übertrifft, werden die besseren Parameter die Referenz für

die nächste Mutation. Dazu müssen sie aber auch einem weiteren Vergleich mit den alten Parametern standhalten. Dieses Turnierverfahren hat den Vorteil, dass Bewertungsdurchläufe nicht repräsentativ sein müssen, um einen Vergleich zuzulassen. Allerdings werden hier auch nur nicht repräsentative Situationen optimiert, was für das Ergebnis nach hinreichend vielen Situationen kein Problem sein sollte, da das Ziel ohnehin eine im Durchschnitt verbesserte Positionsgenauigkeit ist und genau dies durch solche Parameter erfüllt wird, die viele Iterationen „überlebt" haben. Um eine schnellere Konvergenz der Evolution zu erreichen und um ein Maß zu erhalten, das eine Bewertung des Fortschritts der Evolution zulässt, adaptiert der eingesetzte evolutionäre Algorithmus die Schrittweite der Mutationen. Eine kleine Schrittweite ist hierbei ein Indiz für das Erreichen eines Maximums (siehe [21], Kapitel 5.1.3).

Für die Positionsänderungen des Roboters wurde hierbei ein spezielles Verhalten entwickelt, das zufällig gewählte Punkte mit zufälligen Rotationen des Roboters auf Basis der von der CeilingCam angegebenen Position anläuft. In bestimmten Zeitabständen wird hierbei eine Bewertung der Parameter vorgenommen und ein neuer Evolutionsschritt durchgeführt. Für jeden Evolutionsschritt werden die Partikel der parallel laufenden SelfLocator-Module auf Zufallspositionen zurücksetzt, um Einflüsse alter Parameter auf neue zu verhindern. Mit diesem Verfahren konnten allerdings keine Parameter erzeugt werden, die die auf Grund von Erfahrungen eingestellten Parameter übertreffen.

Als mögliche Gründe wurden das Entscheidungskriterium und die Länge der Durchgänge untersucht. Dabei zeigte sich, dass weder strenger (Forderung einer stärkeren Verbesserung oder Übertreffen der handoptimierten Parameter) noch weniger streng formulierte Entscheidungskriterien den gewünschten Erfolg brachten. Im Rahmen sinnvoller Laufzeiten (einige Stunden) konnte auch die Länge der Testläufe nicht so weit erhöht werden, dass verbesserte Parameter erzeugt werden konnten. Ebenso brachte eine Reduktion der Anforderung auf das Torhüter-Verhalten anstelle des Zufallslaufs keine Verbesserung. Nachdem auch dieser Versuch gescheitert war, ist der wahrscheinlichste Grund für das Scheitern der Parameteroptimierung die zu geringe Zahl der Evolutionsschritte. Selbst mit dem beschriebenen Turnierverfahren lassen sich kaum mehr als zwei Mutationen pro Minute realisieren. In Zukunft sollte versucht werden, das Speichern der Eingangsdaten des SelfLocators soweit zu verbessern, dass mit langen, gespeicherten Durchläufen auf leistungsstarken Rechnern und mit evolutionären Strategien mit größeren Populationen eine große Zahl von Evolutionsschritten realisiert werden kann.

6.4.5 Phase 5: Verbesserte Abschätzung der Lokalisierungsgüte

Mit der Entwicklung des TeamBallLocators (Kapitel 5) entstand ein Modul, das eine möglichst gute Abschätzung der Lokalisierungsgenauigkeit benötigt, um Rückschlüsse auf die Genauigkeit der Ballposition in absoluten Koordinaten zu ziehen. Dafür wird durch den SelfLocator eine Validität v zwischen 0 und 1 berechnet. Diese wird im GT2004SelfLocator bei Abschluss des Clusterings aus dem Verhältnis der

Zahl der Partikel innerhalb des Würfels mit der höchsten Partikeldichte und der Zahl der Partikel außerhalb dieses Würfels sowie aus der durchschnittlichen Wahrscheinlichkeit der Partikel innerhalb des Würfels bestimmt. Dieses Verfahren liefert allerdings häufig keine guten Ergebnisse. Um die Güte der Validitätsberechnung zu messen, wurde in verschiedenen Situationen auf dem Spielfeld die Validität v und die Distanz d der durch den SelfLocator errechneten Position von der realen Position protokolliert und hieraus eine Güte mittels der Bewertungsfunktion $q(d, v) = v \cdot d$ abgeleitet. Die kritischen Validitäten sind die, die trotz einer schlechten Lokalisierung einen großen Wert haben. Diese Bewertungsfunktion wird besonders von diesen Werten dominiert. Anhand dieser Funktion können bezüglich der besagten Kriterien Methoden zur Berechnung der Validität verglichen werden. Der Ansatz zur Verbesserung der Validitätsberechnung sollte möglichst kompatibel zur bisherigen Berechnung sein, d.h. einen ähnlichen Verlauf zeigen, aber weniger häufig eine fehlerhaft hohe Validität berechnen. Hierfür wurden zunächst die Varianzen v_x, v_y, v_θ der Partikelverteilung in x- und y-Richtung und Rotation θ berechnet, aus der dann mittels $e^{-(\sqrt{v_x+v_y+v_\theta}\cdot c)^2}$ eine neue Validität abgeschätzt wurde, die im Durchschnitt bessere Ergebnisse lieferte, als die bisher benutzte (Tabelle 6.1).

q_{alt}	118,84
q_{neu}	109,72

Tabelle 6.1: Verbesserung der Validitätsberechnung

Allerdings beschreibt diese Validität generell nur schlecht die Ausdehnung der Partikelverteilung. Der Grund hierfür ist, dass die Partikel mit den gewählten Parametern sich zumeist nicht gemäß einer gaußschen Verteilung bewegen, sondern in mehreren kleinen Clustern. Daher ist auch eine Berechnung einer Varianz/Covarianz-Matrix[4] zur Bestimmung einer Aufenthaltswahrscheinlichkeitsellipse nicht erfolgversprechend.

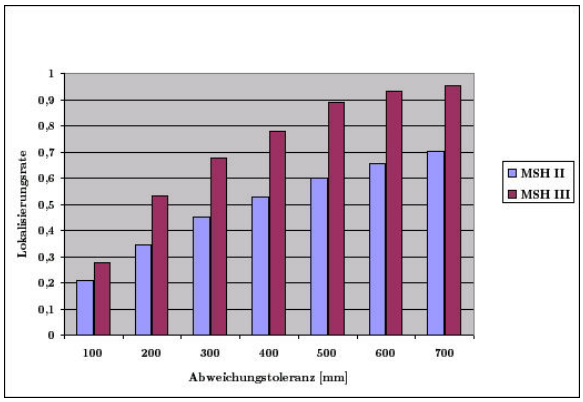
Diese Beobachtung legte nahe, dem TeamBallLocator anstatt eines einzigen Zuverlässigkeitswertes oder einer Varianz/Covarianz-Matrix zusätzlich zur Gesamtvalidität die Positionen und jeweils eine eigene Validität dieser Cluster mitzuteilen. In der *RobotPoseCollection* werden die jeweils 3 umfangreichsten Cluster gespeichert.

Dies ermöglicht dem TeamBallLocator, ausgehend von diesen *RobotPose*-Hypothesen, verschiedene Ballpositionen abzuleiten.

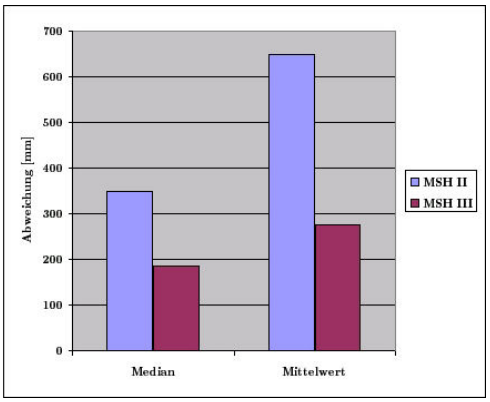
6.4.6 Phase 6: Richtung von Linienkreuzungen

Bei der Erkennung von Linienkreuzungen lässt sich auch eine Richtung dieser ableiten, was die möglichen Aufenthaltspositionen, die sich aus einer Linienkreuzung ergeben, ähnlich zum Mittelkreis, auf 2 reduziert. Die Nutzung dieser Zusatzinformation wurde in der experimentellen Version des SelfLocators erprobt. Allerdings konnte diese Version bis zum Ende der Projektgruppenzeit nicht hinreichend gut

[4] siehe Anhang D



(a) Häufigkeit einer Lokalisierung für eine gegebene maximale Abweichungstoleranz



(b) Median und Mittelwert der Positionsabweichung

Abbildung 6.13: Verbesserung durch Richtungen von Linienkreuzungen

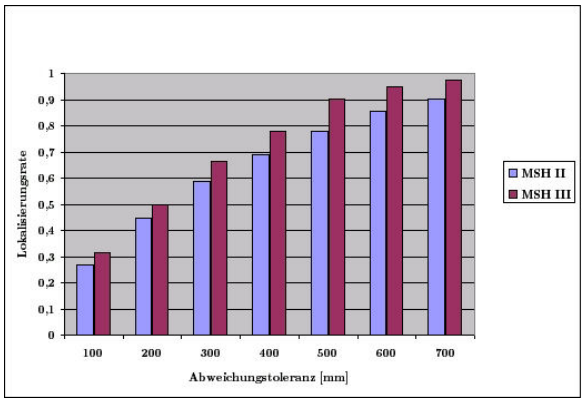
getestet werden, so dass eine Nutzung in den Wettkämpfen nicht sinnvoll war. Erste Messungen bestätigten allerdings die Vermutung, dass mit der Richtungsinformation an Linenkreuzungen eine deutliche Verbesserung möglich ist (Abbildung 6.13). Bei dieser Abbildung ist zu beachten, dass während der Messung eine Delokalisierung provoziert wurde, indem der Roboter mehrfach in ein Hindernis gesteuert wurde. Berücksichtigt man diesen Abschnitt der stärksten Delokalisierung in der Auswertung nicht, ist die Differenz der beiden SelfLocator Varianten geringer (Abbildung 6.14).

6.4.7 Phase 7: Optimierung messbarer Parameter

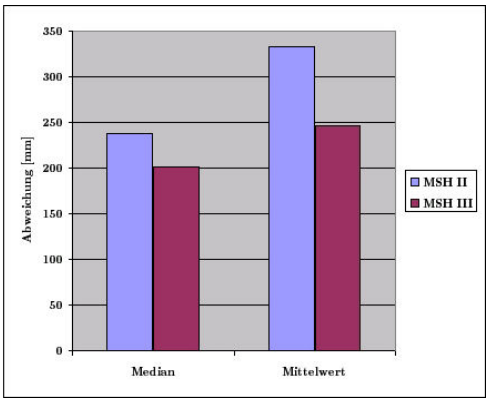
Der Wert der messbaren Parameter des SelfLocators (Kapitel 6.4.1) repräsentiert in der gegenwärtigen Form nur einen Durchschnittswert für die Varianzen der einzelnen Perzepte. Wenn der Roboter aber im Verlauf eines Spiels unterschiedliche Laufbewegungen absolviert, wird die Kamera mit verschiedenen Frequenzen bewegt, so dass die Genauigkeit der Perzepte über kurze Zeiträume nicht diesem Durchschnittswert entspricht. Um die Genauigkeit der Selbstlokalisierung weiter zu verbessern wurde daher versucht, die Funktion der Genauigkeit einzelner Perzepte in Abhängigkeit des Bewegungszustands des Roboters mithilfe eines Neuronalen Netzes (Anhang B) anzunähern. Der Versuch hierfür eine eigene Applikation zu entwickeln wurde abgebrochen, da mit „SNNS“ (Stuttgart Neural Network Simulator[5]) ein Werkzeug zu Verfügung steht, mit dem Neuronale Netze schnell erstellt und trainiert werden können.

Hierfür müssen erst alle benötigten Neuronen erzeugt und angeordnet werden. Jedem Neuron wird eine Nummer, eine Aktivierungsfunktion, eine Ausgabefunktion, eine Schicht und damit auch ein Typ zugeordnet (siehe Abbildung 6.15 unten rechts). Die Verbindungen zwischen den Neuonen werden je nach Netztyp automatisch er-

[5] http://www-ra.informatik.uni-tuebingen.de/SNNS



(a) Häufigkeit einer Lokalisierung für eine gegebene maximale Abweichungstoleranz



(b) Median und Mittelwert der Positionsabweichung

Abbildung 6.14: Verbesserung durch Richtungen von Linienkreuzungen - Messung ohne Delokalisierung

stellt. Der Fehlergraph (siehe Abbildung 6.15 unten links) stellt die quadratische Abweichung zwischen der Soll- und der Ist-Ausgabe aller Trainingsdaten dar. Durch Veränderungen der Aktivierungsfunktionen, Ausgabefunktionen und spezieller Parameter des Backpropagation-Algorithmus - wie der Lernrate und dem maximalen Fehler - kann der Gesamtfehler minimiert werden.

6.4.7.1 Umsetzung

Wir haben uns dafür entschieden, zuerst die Abweichungen der Landmarkenperzepte mithilfe des Neuronalen Netzes durch eine Funktion zu approximieren. Der Ablauf für die Torperzepte sähe genauso aus.

Als 3 Eingangsneuronen des Neuronalen Netzes haben wir die Veränderung der Odometriedaten des Roboters gewählt, also die Fortbewegung des Roboters in x-Richtung, die Fortbewegung in y-Richtung und die Drehung des Roboters seit dem letzten Frame. Das Neuronale Netz besitzt genau 1 Ausgabeneuron für den erwarteten Sichtwinkel zur Landmarke. Wir haben den SelfLocator so verändert, dass in jedem Durchlauf die für das Neuronale Netz benötigten Werte berechnet werden. Mit dieser Wertetabelle können wir dann das Neuronale Netzwerk in SNNS trainieren.

6.4.7.2 Schwierigkeiten

Wenn der Roboter steht, die Werte der Eingangsneuronen also $(0, 0, 0)$ sind, ist der Ausgabewert auf Grund der ungenauen Kameradaten des Roboters kein konstanter Wert. In einem solchen Fall ist das Netzwerk nicht in der Lage, seine Gewichte zu erlernen. Falls identische Inputwerte zu unterschiedlichen Ausgabewerten führen, wird der Lernalgorithmus des Netzwerks gestört. Aus dem Grund werden die Input- und Outputwerte im SelfLocator nur dann ausgegeben, wenn sich der Roboter auch

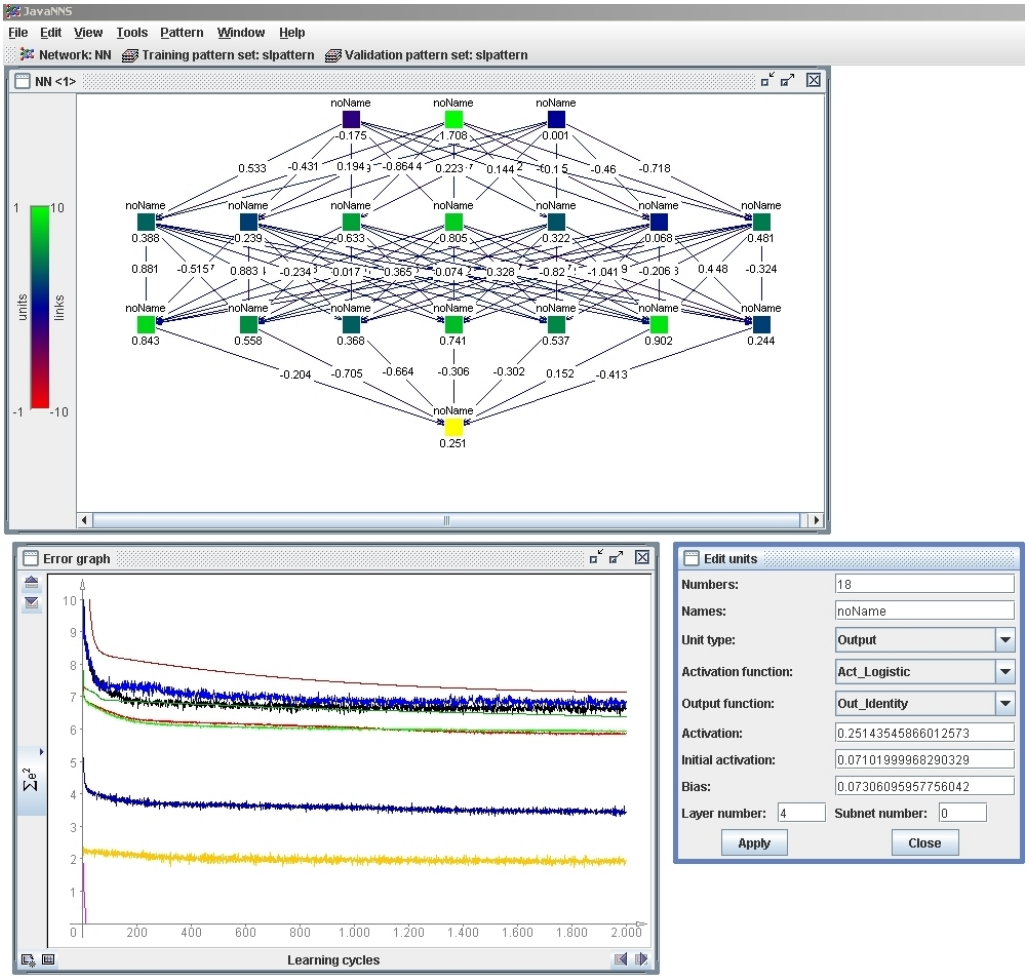


Abbildung 6.15: SNNS, ein Werkzeug für Neuronale Netze

wirklich bewegt hat, die CeilingCam also eine Bewegung des Roboters festgestellt hat.

6.4.7.3 Ergebnisse

Durch Variieren der Netzgröße, der Aktivierungs- und Ausgabefunktionen konnten wir den Gesamtfehler des Neuronalen Netzes minimieren. Allerdings sind die Ergebnisse nicht zufriedenstellend, da die Gewichte in der Tainingsphase so angepasst wurden, dass das Neuronale Netz immer einen ähnlichen Ausgabewert liefert, im Endeffekt also nur einen Durchschnitt über die erwarteten Ausgabewerte liefert (siehe blaue Kurve in Abbildung 6.16). Für viele Testdaten führt dies zu einer geringen Differenz zwischen der berechneten und erwarteten Ausgabe des Netzwerks, im Durchschnitt beträgt die Abweichung 7°. Der Median der prozentualen Abweichung beträgt 33,89%.

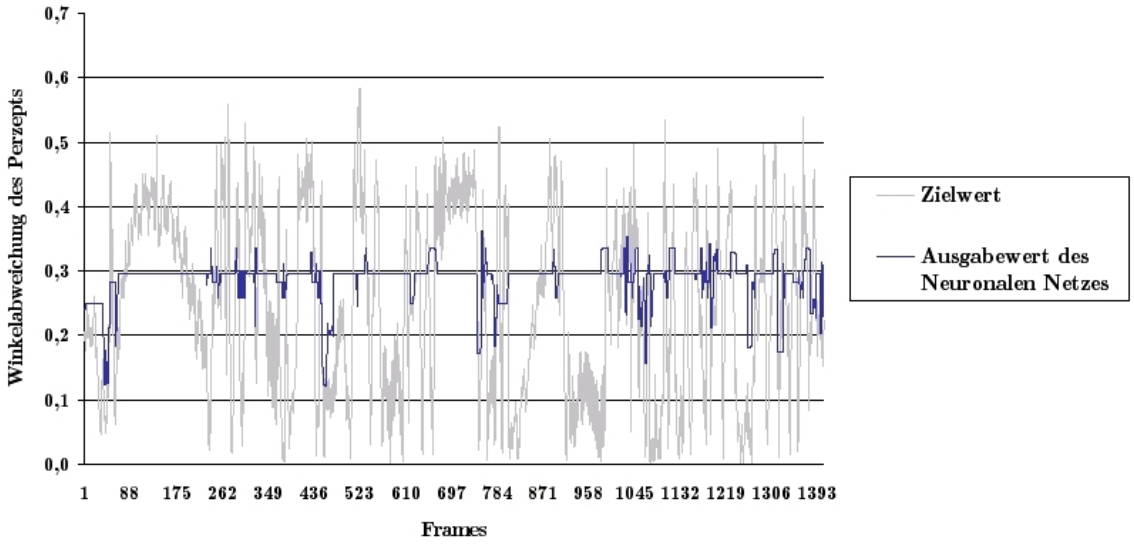


Abbildung 6.16: Ergebnisse eines Trainigsdurchlaufs

6.4.7.4 Diskussion

Wenn der Roboter geradeaus läuft, ändern sich die Werte der Eingangsneuronen kaum, da sich die Laufgeschwindigkeit und die Laufrichtung des Roboters nicht ändert. Dadurch ähneln sich viele Trainingsdaten. Möglicherweise ist das ein Grund dafür, dass das Neuronale Netz für viele Datenmengen gute Ergebnisse, hingegen für die Datenmengen, die von den üblichen Werten abweichen, schlechte Ergebnisse liefert. Dieses Problem ist auch als „Overfitting“ bekannt. Mit Overfitting beschreibt man das Problem der Überanpassung eines Modells an einen Datensatz. Das Modell passt sich sehr gut an diese Datenmenge an, kann aber wegen der fehlenden Generalisierung nur schlecht auf neue Daten reagieren (siehe [20], Kapitel 6.3.4).

Durch die schnelle Kopfbewegung des Roboters sind die Daten der Kamera sehr verrauscht, für ähnliche Eingangswerte werden unterschiedliche Ausgabewerte berechnet. Daher haben wir uns entschieden, als viertes Eingangsneuron die Kopfbewegungsgeschwindigkeit des Roboters zu verwenden. Diese Idee konnten wir aber aus Zeitmangel nicht mehr zu Ende verfolgen.

6.5 Ausblick: Lokalisierung mit Liniensegmenten statt Linienpunkten

Werden statt der Linienpunkte Teile erkannter Linien genutzt, kann die Informationsausbeute pro Perzept weiter gesteigert werden. Liniensegmente vereinen die Informationen mehrerer Linienpunkte in einem Perzept. Da die Bestimmung von Liniensegmenten ohnehin für die Linienkreuzungserkennung notwendig ist, entsteht hierdurch kein weiterer zusätzlicher Rechenaufwand. Bei gleicher Anzahl benutzter Perzepte und damit gleicher Laufzeit ist also eine Verbesserung der Lokalisierung

möglich. Zusätzlich zur Vereinigung der Positionsinformation mehrerer Linienpunkte lässt sich für Linien die Ausrichtung bestimmen, wodurch für ein bestimmtes Liniensegment nur noch zwei mögliche Roboterrotationen verbleiben.

7 Bildverarbeitung

Die Bildverarbeitung stellt für den Roboter nahezu die einzige Quelle von Informationen über die ihn umgebende Welt dar. Erkannt werden müssen sowohl die statischen Elemente der Umwelt wie Landmarken und Feldmarkierungen, als auch die dynamischen Elemente wie der Ball und idealer Weise auch andere Roboter. Die gesammelten Informationen über die erkannten Objekte in der Umgebung, wie zum Beispiel der Abstand der Objekte zu dem AIBO, werden als Perzept (engl. die Empfindung) abgespeichert. Ein Perzept ist also ein Set von Informationen über ein erkanntes Objekt in der Umgebung. Die nun generierten Perzepte werden als Eingabedaten an SelfLocator , BallLocator und PlayersLocator weitergegeben. Dabei stehen nur die Informationen über die räumliche Lage und Orientierung der Kamera und das eigentliche Bild zur Verfügung, ein Rückfluss aus der Modellierung ist nicht vorgesehen.

Um die verschiedenen Aufgaben zu bewältigen, ist der ImageProcessor modular aufgebaut. Zunächst gibt es einen Hauptteil, der durch Scannen des Kamerabildes Linienpunkte, Hindernisse und orange Bildabschnitte findet. Das Scannen, siehe Abbildung 7.1, wird sowohl senkrecht, als auch parallel zur Horizontlinie, die im Bild die „Augenhöhe“ des AIBOs anzeigt, ausgeführt. In Richtung senkrecht zur Horizontallinie wird mit drei Typen von Scanlinien, die sich in der Länge unterscheiden, abgetastet und in paralleler Richtung verlaufen die Scanlinien über das gesamte Bild.

Nach dem Scanvorgang werden die gesammelten Informationen an die Spezialisten weitergegeben. Diese Spezialisten sorgen dann für die exakte Erkennung bzw. Charakterisierung der gefundenen Objekte und das Füllen der dafür vorgesehenen

Abbildung 7.1: Die Scanlinien des ImageProcessors

Datenstrukturen.

Um den veränderten Bedingungen gerecht zu werden und neuen Raum für Verbesserungen zu geben, wurden einige dieser Spezialisten verbessert oder auch neu entwickelt.

7.1 Ballerkennung

In diesem Kapitel wird die Anpassung der bereits vorhandenen Ballerkennung an die neuen Feldbedingungen beschrieben.

7.1.1 Problembeschreibung

Das Spielfeld wird seit 2005 nicht mehr durch eine 10 cm hohe Bande abgegrenzt, so dass Gegenstände aus der Umgebung des Feldes nun für den AIBO sichtbar sind. Es ergibt sich das Problem, dass orange Gegenstände als Ball erkannt werden können.

Im Folgenden wird der Beitrag des ImageProcessors zur Lösung dieses Problems beschrieben.

7.1.2 Ansatz zur Lösung des Problems

Zur Lösung des Problems müssen zuerst alle orangen Kandidaten im Bild erfasst werden. Einen Kandidaten nimmt der ImageProcessor in Form von vielen orangen Scanlinienläufen wahr, siehe Abbildung 7.2, die als Cluster[1] zusammengefasst werden müssen. Für das Zusammenfassen der Scanlinienläufe und die Verwaltung der Cluster wird die Datenstruktur einer doppelt verknüpften Liste verwendet (siehe Abbildung 7.3). Sie besteht aus drei Listen, *cList*, *vNavi* und *hNavi*. Die *cList* ist für die Verwaltung von Clustern zuständig. Ein Cluster besteht aus den Extremwerten y_{min}, y_{max}, x_{min} und x_{max} und der aus den Extremwerten berechnete Mittelpunkt, siehe Abbildung 7.2. Die Listen *vNavi* und *hNavi* sind Listen von Zeigern, die für die Sortierung der Cluster zuständig sind. Die Zeiger von *vNavi* referenzieren so auf die Cluster der *cList*, dass das Durchlaufen der Liste *vNavi* eine nach x_{min} aufsteigend geordnete Ausgabe der Cluster ergibt. Dementsprechend ergibt das Durchlaufen der Liste *hNavi* eine nach y_{min} geordnete Ausgabe. Die Cluster in *cList*, auf die referenziert wird, speichern selber auch eine Referenz auf die Elemente in den Zeigerlisten ab, die auf sie zeigen.

Die Zeigerlisten sind so konzipiert, dass eine Änderung in einer der beiden Zeigerlisten, wie zum Beispiel dem Löschen oder dem Einfügen eines Clusters, automatisch in der anderen Zeigerliste aktualisiert wird.

Um nun die Scanlinienläufe, die jeweils mindestens drei Bildpunkte lang sind, zusammenzufassen, werden sie nacheinander in Form von Start- und Endpunkt einer

[1] Mit Cluster (engl. Traube, Bündel, Schwarm, Haufen) wird ein Verbund der orangen Scanlinienläufe bezeichnet, die sich an einer Stelle anhäufen.

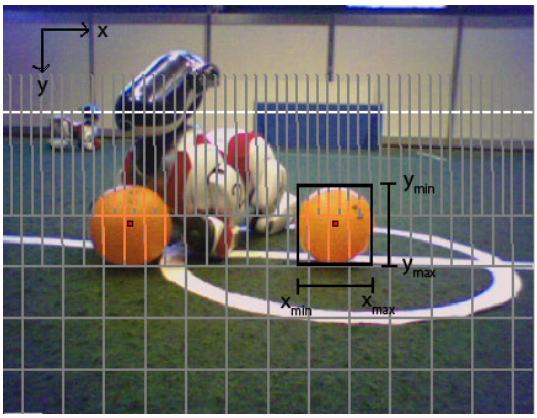


Abbildung 7.2: Der ImageProcessor nimmt ein oranges Objekt in Form von orangen Scanlinienläufen wahr. Ein Cluster wird durch die Extremwerte, die die Bounding-Box definieren, und dessen Mittelpunkt definiert.

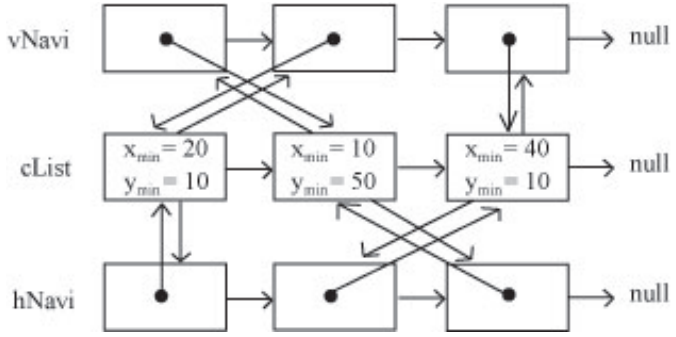


Abbildung 7.3: Die Datenstruktur für die Clusterung

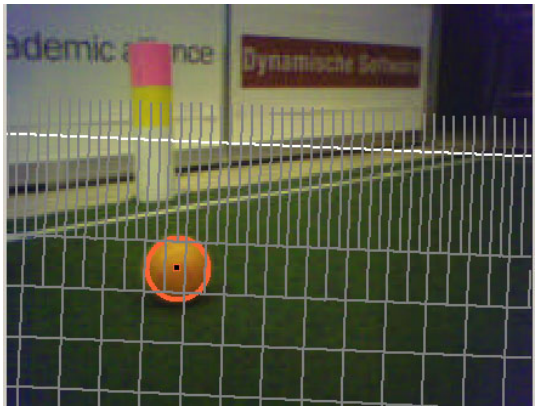


Abbildung 7.4: Die Scanlinien verlaufen bedingt durch die Kopfbewegung nicht genau vertikal bzw. horizontal

Instanz von *Clustering* übergeben. Da die Scanlinienläufe bedingt durch die Kopfbewegung des AIBO nicht genau vertikal bzw. horizontal verlaufen, siehe Abbildung 7.4, werden die einzelnen Scanlinienläufe zur Vereinfachung durch eine Bounding-Box[2] repräsentiert. Zu Beginn von jedem Frame ist die Datenstruktur leer und der erste Scanlinienlauf wird mit ihren Extremwerten als Cluster in die Clusterliste eingefügt. Beinhaltet die *cList* jedoch bereits Elemente, so muss überprüft werden, ob Cluster vorhanden sind, die nah genug der Bounding-Box liegen, um mit der Bounding-Box zu einem Cluster zusammengefasst zu werden. Die maximale Entfernung eines Clusters zu einer Bounding-Box, die noch zu einer Fusion führt, muss so gewählt werden, dass alle Scanlinienläufe eines Objektes trotz kleinerer Störungen, wie beispielsweise die Reflektion des Lichtes im Ball, miteinander verschmolzen werden. Die Maximalentfernung muss also mindestens so groß sein, wie der Abstand der einzelnen Scanlinien zueinander. Da das Bild vertikal durch drei unterschiedliche Typen von Scanlinien, die sich in der Länge unterscheiden, abgetastet wird, kann es in drei Bereiche mit jeweils unterschiedlichen Abständen der Scanlinien unterteilt werden. Die horizontalen Scanlinien verlaufen alle durch das gesamte Bild und besitzen den gleichen Abstand zueinander. Nach mehrmaligem Testen haben sich folgende Maximalabstände (siehe Abbildung 7.5) bewährt:

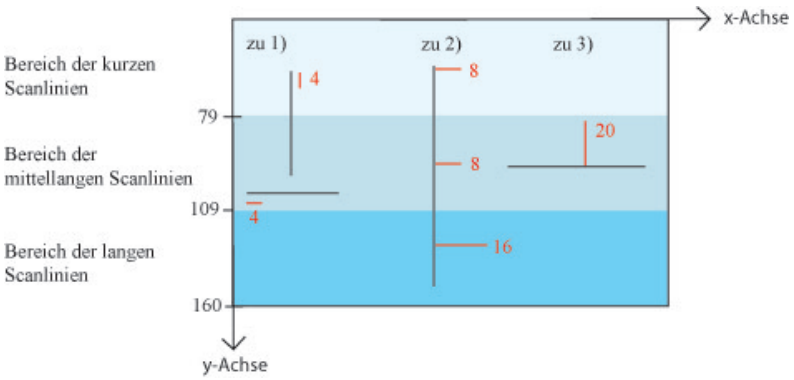
Die hier genannten x- und y-Werte sind Koordinaten in der Bildebene:

1. Bei der Überprüfung der Cluster in Richtung der Abtastung, ist der maximale Abstand 4 Bildpunkte.

2. Bei der Überprüfung von Nachbarn in der Horizontalen, während die Abtastung in der Vertikalen geschieht, ist der maximale Abstand:

   a) Im Bereich der kurzen Scanlinien, das heißt von $y = 0$ bis $y = 79, 8$ Bildpunkte.

[2]Eine Bounding-Box ist das kleinste Rechteck, das den Scanlinienlauf noch komplett umfasst



zu 1) In Richtung der Abtastung.

zu 2) Abstand zu horizontalen Nachbarn, während die Abtastung in die vertikale Richtung verläuft.

zu 3) Abstand zu vertikalen Nachbarn, während die Abtastung in die horizontale Richtung verläuft.

Abbildung 7.5: In der Abbildung werden die zu wählenden Maximalabstände (rot) in Abhängigkeit von der Abtastrichtung (schwarz) dargestellt.

b) Im Bereich der mittleren Scanlinien ohne den Bereich der kurzen Scanlinien, das heißt von $y = 80$ bis $y = 109$, 8 Bildpunkte.

c) Im Bereich der längsten Scanlinien ohne den Bereich der mittleren Scanlinien, das heißt von $y = 110$ bis $y = 160$, 16 Bildpunkte.

3. Bei der Überprüfung von Nachbarn in der Vertikalen, während die Abtastung in der Horizontalen geschieht, beträgt der maximale Abstand 20 Bildpunkte.

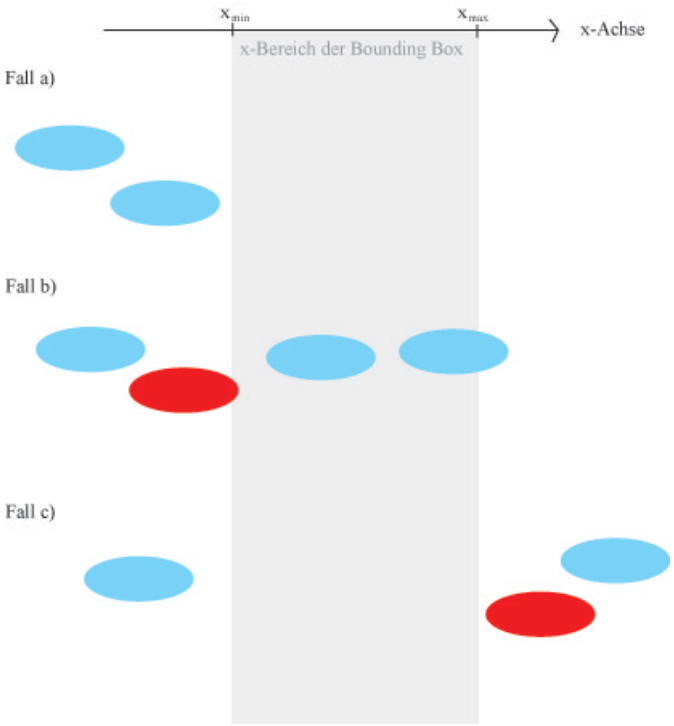
Der genaue Ablauf der Clusterung wird im Folgenden für den Fall des vertikalen Abtastens erläutert.

Für das vertikale Clustern stehen 3 Methoden zur Verfügung: *prepV*, *markV* und *clusterFinal* (siehe Abbildung 7.7). Die Methode *prepV* durchläuft die Liste *vNavi* und sucht das erste Cluster heraus, das in den x-Bereich der Bounding-Box tritt, wie in Abbildung 7.6, Fall b) dargestellt. Falls das Ende der Liste erreicht wurde (vergleiche Abbildung 7.6, Fall a)), sind alle Cluster vor der Bounding-Box platziert. Wird die Suche angehalten (vergleiche Abbildung 7.6, Fall c)), dann gibt es kein Cluster innerhalb des x-Bereiches der Bounding-Box und das erste Cluster nach der Bounding-Box wird durch *prepV* zurückgegeben.

Im Falle, dass sich keine Cluster innerhalb des x-Bereiches befinden, wird ein neues Cluster mit den Extremwerten der Bounding-Box in die entsprechende Stelle eingefügt:

- Wenn das Ende der Liste erreicht wurde, wird das Cluster an das Ende der Liste eingefügt.
- Wenn ein Cluster gefunden wurde, das den Bereich der Bounding Box in x-Richtung bereits übersprungen hat, so wird das neue Cluster davor eingefügt.

Wenn sich Cluster innerhalb des Bereiches der Bounding-Box in x-Richtung befinden, so wird die Methode *markV* aufgerufen, die zu fusionierende Kandidaten



a) Die Cluster liegen vor der Bounding-Box.
b) Es liegen Cluster im Bereich der Bounding-Box. Das erste Cluster (rot), das die Box schneidet wird von *prepV* zurückgegeben.
c) Es liegen keine Cluster im Bereich der Bounding-Box. Das erste Cluster nach der Bounding-Box wird von *prepV* zurückgegeben.

Abbildung 7.6: Die Methode *prepV*

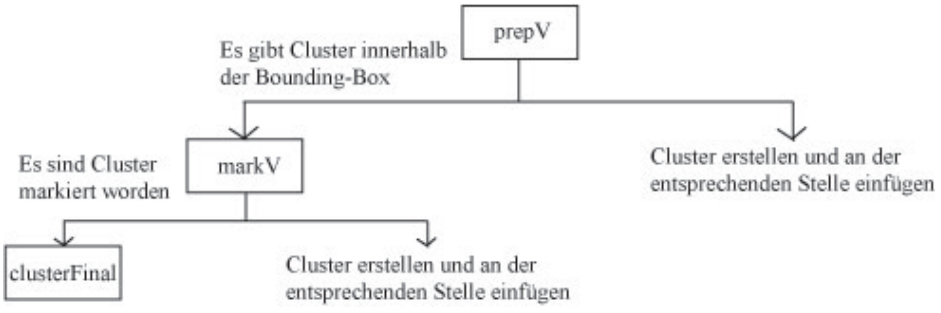


Abbildung 7.7: Der Ablauf für das vertikale Clustern

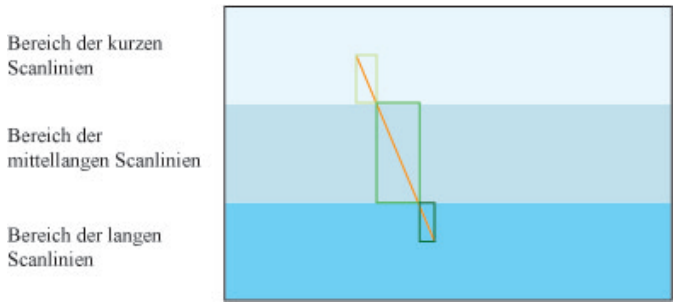


Abbildung 7.8: Unterteilung der Bounding-Box

markiert. Dafür wird die Liste *vNavi* beginnend mit dem durch *prepV* gefundenen Cluster durchlaufen, wobei jedes Cluster auf den vertikalen Abstand zu der Bounding-Box überprüft wird. Markiert werden diejenigen Cluster, die den Bereich $x_{min} - 4$ (Maximalabstand bei Überprüfung der Cluster in Richtung der Abtastung in vertikaler Richtung) bis $x_{max} + 4$ schneiden. Die Überprüfung wird beendet sobald ein Cluster gefunden wird, das den Bereich der Bounding-Box in x-Richtung übersprungen hat, oder das Ende der Liste erreicht wird. Durch Zeiger wurde jeweils das erste und das letzte Cluster vermerkt, das markiert wurde. Falls überhaupt keine Cluster markiert worden sind, so wird ein neues Cluster mit den Extremwerten der Bounding-Box entsprechend von x_{min} eingefügt. Falls x_{min} kleiner als die minimale Grenze in x-Richtung des durch *prepV* gefundenen Clusters ist, dann wird die Bounding-Box davor eingefügt, ansonsten dahinter. Im Falle, dass Cluster markiert worden sind, werden diese durch die Methode *clusterFinal* mit der Bounding-Box zu einem Cluster zusammengefasst und an der Stelle eingefügt, an der sich das erste markierte Cluster befindet. Die Aktualisierung der anderen Zeigerliste erfolgt bei Änderung von *vNavi* automatisch.

Nachdem die Nachbarn in Abtastrichtung überprüft worden sind, müssen auch noch die Nachbarn in horizontaler Richtung überprüft werden. Hierbei muss beachtet werden, dass auf Grund der unterschiedlich langen Scanlinien drei Bildbereiche mit unterschiedlichen Maximalentfernungen existieren, indem die Bounding-Box entsprechend der Anzahl der durchlaufenden Bildbereiche in Bounding-Boxen unterteilt wird, siehe Abbildung 7.8. Jede einzelne Bounding-Box wird wie vorher beschrieben entsprechend der Abbildung 7.7 untersucht, mit dem Unterschied, dass statt *prepV* *prepH* und statt *markV* *markH* benutzt wird.

Nachdem die Kandidaten erfasst worden sind, werden sie alle dem Ballspezialisten übergeben. Für die nun vom Ballspezialisten als Ball empfundenen Kandidaten wird eine Validität berechnet, die abhängig von 4 Faktoren ist, nämlich den Prozentzahlen *percentOfOrange*, *distanceReliability*, *durabilityOfBallPoints* und *deviationOfBallPoints*. Die Zahl *percentOfOrange* gibt das Verhältnis von orangen Punkten zu der Gesamtzahl an abgetasteten Punkten an. Die Zahl *distanceReliability* gibt die Übereinstimmung zweier Abstände, die auf unterschiedliche Weise nach [8] berechnet wurden, an. Eine Abstandsberechnung benutzt den durch den Ballspezialisten angegebenen Radius im Bild und berechnet aus der Kenntnis des tatsächlichen Ball-

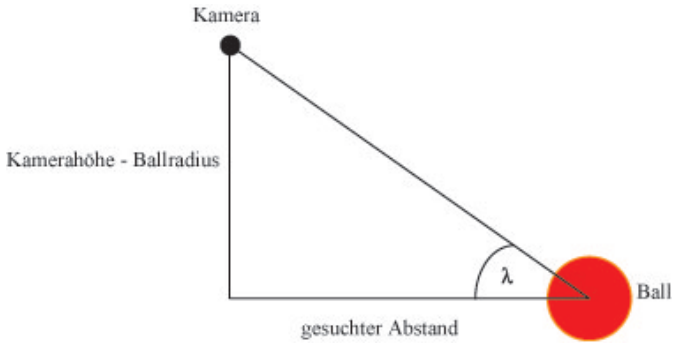


Abbildung 7.9: Abstandsberechnung mithilfe des Winkels γ

radius den Abstand, während die andere Berechnung den Winkel γ, siehe Abbildung 7.9, nutzt. Die Zahl *durabilityOfBallPoints* berücksichtigt den Anteil an Punkten, die in der Nähe von grün bzw. gelb liegen. Die Zahl *deviationOfBallPoints* ist der Durchschnittsabstand der Ballpunkte zu dem Mittelpunkt des Balles.

Die Kandidaten werden dann der Größe der Reliabilty nach geordnet in einer Liste abgespeichert und an das *BallPercept* als *MultipleBallPercept* zur weiteren Auswertung übergeben.

7.1.3 Diskussion der Güte der Lösung

Getestet wurde der Ansatz mit zwei Bällen und einer orangen Schranktür, die durch drei Scanlinienbereiche durchläuft. Alle Objekte im Bild wurden richtig erfasst, d.h. der korrekte Mittelpunkt der Objekte wurde angezeigt. Die dafür benötigte Laufzeit beträgt 1 ms. Die Laufzeit dieses Ansatzes ist vor allem davon abhängig, wieviele orangefarbene Objekte sich im Bild befinden bzw. wie weit es sich über das Bild in vertikaler Richtung erstreckt, wie zum Beispiel die Schranktür. Durchläuft das orangefarbene Objekt mehrere Scanlinienbereiche, siehe Abbildung 7.5, so wird das Objekt entsprechend der durchlaufenden Bereiche unterteilt, die einzeln untersucht werden. Die Laufzeit wird dementsprechend für das eine Objekt entsprechend der Anzahl der durchlaufenden Bereiche vervielfacht, also im schlimmsten Falle verdreifacht.

7.2 Linien

Feldlinien, genauso wie der Mittelkreis, sind ein Teil jedes Fußballfeldes, sei es Roboterfußball oder der „richtige Fußball“. Daher werden diese auch in Zukunft immer zur Verfügung stehen, was bei den momentan noch üblichen künstlichen Landmarken sicher nicht mehr lange der Fall ist. Ganz allgemein lässt sich daher das Erkennen von Linien und die Orientierung mit ihrer Hilfe als ein Schritt in die richtige Richtung bezeichnen.

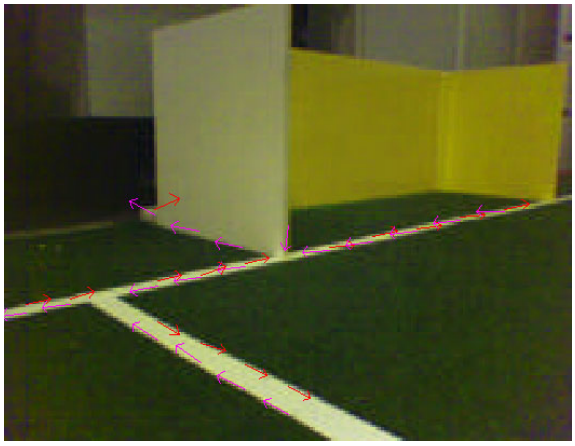


Abbildung 7.10: Linienpunkte und aus den Gradienten abgeleitete Richtungen

7.2.1 Problembeschreibung

Linien bieten neben Landmarken und Toren eine weitere Möglichkeit, sich auf dem Feld zurechtzufinden. Im Gegensatz zu diesen werden sie aber viel häufiger gesehen und besonders auch, ohne extra den Kopf heben zu müssen und dabei eventuell den Ball aus dem Blick zu verlieren.

Informationen über Linien wurden auch bisher schon genutzt, allerdings nur recht spärlich in Form von Linienpunkten. Diese Linienpunkte bezeichnen Koordinaten von Punkten, über die beim Scannen herausgefunden wurde, dass sie auf einer Feldlinie liegen.

Im Folgenden wird nun eine Möglichkeit beschrieben, aus diesen Punkten auf effiziente Weise die zugehörigen Linien und deren Schnittpunkte zu berechnen.

7.2.2 Ansätze zur Lösung des Problems

Wie schon beschrieben wurde, werden einfache Linienpunkte schon während des Scanvorgangs im ImageProcessor-Hauptteil gefunden, somit auch nur entlang der dort benutzen Scanlinien. Kriterien für einen solchen Linienpunkt sind ein entsprechender Gradient am Rand der Linie, sowie ein entsprechend klassifizierter Weiß/Grün-Übergang. Zusätzlich wird noch überprüft, ob die gefundene Breite der Linie mit einer gewissen Toleranz der erwarteten Breite an dieser Stelle im Bild entspricht. Die gefundenen Linienpunkte werden weiterhin für die im Folgenden beschriebenen Algorithmen als Eingabe aufgefasst. Zusätzlich kann eine Schätzung der Richtung der zugehörigen Linie aus dem lokalen Gradienten des Helligkeitskanals um den Linienpunkt herum berechnet werden, wie in Abbildung 7.10 zu sehen ist.

7.2.2.1 Hough-Transformation

Eine Möglichkeit, nur aus den Punkten, also ohne die zusätzliche Richtungsinformation, Linien zu finden, ist durch die Hough-Transformation gegeben.

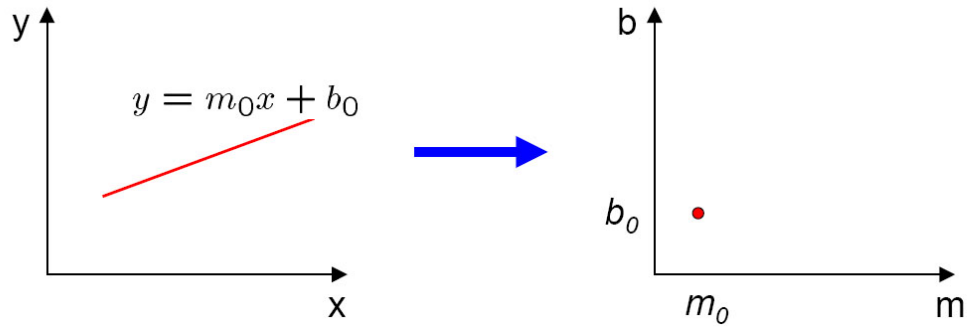


Abbildung 7.11: Eine Gerade in der Bildebene kann eindeutig beschrieben werden durch einen Punkt in der Parameterebene

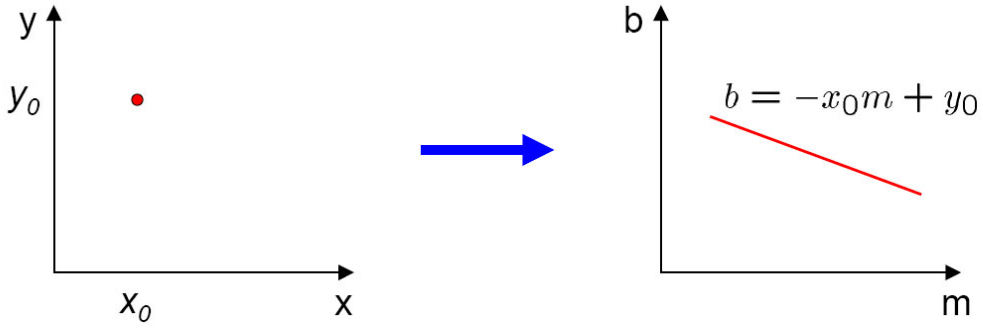


Abbildung 7.12: Ein einzelner Punkt in der Bildebene ergibt eine Gerade verschiedener Parameterpaare im Parameterraum

Bei dieser wird der Dualismus zwischen der Bildebene des Koordinatenraums und der Parameterebene des Raums aller Geraden in Parameterform ausgenutzt: Eine bekannte Gerade $y = m_0 \cdot x + b_0$ in der durch die x-Achse und y-Achse aufgespannte Bildebene ergibt einen Punkt (m_0, b_0) auf der durch die m-Achse und b-Achse der Parameter aufgespannte Parameterebene, wie in Abbildung 7.11 dargestellt.

Ein einzelner Punkt (x_0, y_0) in der Bildebene kann nun aber zu vielen verschiedenen Geraden $y = m_i \cdot x + b_i$ gehören, die alle (x_0, y_0) berühren. Also ergibt dieser Punkt auf der Bildebene wiederum eine Gerade $b = -x_0 \cdot m + y_0$ in der Parameterebene, nämlich die Menge aller Parameterpaare (m_i, b_i), die die möglichen Geraden in der Bildebene beschreiben, wie in Abbildung 7.12 illustriert. Dies gilt, wenn die Gerade in der Form $y = m \cdot x + b$ gegeben ist.

Sind mehrere Punkte gegeben, die auf einer Geraden liegen, so ergeben sich die Parameter der Geraden aus dem Schnittpunkt der zugehörigen Geraden in der Parameterebene.

In realen Anwendungen sind aber nicht mehrere Punkte von nur einer Geraden die Eingabe, sondern Punkte, die zum Teil auf unterschiedlichen Geraden liegen, zum Teil aber auch nur auf Bildrauschen und unsichere Eingabedaten zurückzuführen sind. Es ist also ein Bewertungsschema nötig, um zu entscheiden, welche Geraden in

dem Bild wirklich vorkommen. Dazu muss zuerst einmal der Parameterraum diskretisiert, also in Zellen aufgeteilt werden, da sonst unendliche viele mögliche Parameterpaare bewertet werden müßten. Diese sogenannten Akkumulatorzellen enthalten ganzzahlige Werte, die zu Beginn mit 0 initialisiert sind.

Jeder Punkt p_i aus der Eingabemenge ist ein Punkt in der Bildebene und besitzt somit eine entsprechende Gerade g_i in der Parameterebene. Für jeden dieser Punkte p_i werden nun die Werte derjenigen Zellen um einen erhöht, die von g_i berührt werden.

Anschließend liefern die Zellen mit den höchsten Werten die wahrscheinlichsten Geraden. Denn jede Zelle steht stellvertretend für ein Parameterpaar, also für eine mögliche Gerade in der Bildebene, und der Wert der Zelle entspricht der Anzahl der auf dieser Geraden liegenden Punkte der Eingabemenge.

Ein Problem ergibt sich allerdings bei sehr steilen Geraden, das heißt bei solchen, die annähernd parallel zur y-Achse liegen. Hier ergeben sich bei einer Darstellung der Geraden in der Form $y = a \cdot x + b$ sehr hohe Werte sowohl für den Parameter a als auch für b. Um dies zu vermeiden, wird die Hessesche Normalform $x \cdot \cos(\Theta) + y \cdot \sin(\Theta) = d$ benutzt. Hier beschreibt Θ die Neigung der Geraden und d den Abstand vom Koordinatenursprung. Θ ist also ein Winkel und d durch die Länge der Bilddiagonalen beschränkt. Allerdings ergibt sich nun für einen Punkt in der Bildebene keine Gerade in der Parameterebene, sondern eine Sinuskurve. Das Prinzip sowie das algorithmische Bewertungsverfahren bleibt aber gleich.

Bei diesem Verfahren lässt sich nun die Rechenzeit sehr einfach über die benutzte Auflösung des Akkumulatorraumes skalieren. Da allerdings bereits kleine Abweichungen der Geraden von den wirklich Abgebildeten bei der Rückprojektion auf das Feld große Fehler ergeben können, bringt diese Möglichkeit der Skalierung keinen großen Vorteil, da die Auflösung dementsprechend hoch zu wählen ist.

7.2.2.2 Partikelfilter im Hough-Raum

Eine Möglichkeit, die Laufzeit des in Kapitel 7.2.2.1 beschriebenen Ansatzes weiter zu reduzieren, ohne drastische Auflösungsverschlechterung hinnehmen zu müssen, besteht darin, den Parameterraum nicht zu quantisieren, sondern stattdessen einen Partikelfilter darauf anzusetzen. Das zugrunde liegende Prinzip von Partikelfiltern wurde in Kapitel 4.2.2 und Kapitel 6.2 schon hinreichend erläutert.

Hierbei bietet sich einem nun auch die Möglichkeit, die zusätzliche Information über die mögliche Richtung von Linienpunkten in das Verfahren einzubringen. Diese Richtungsinformation einer zu einem Linienpunkt gehörenden Linie kann aus dem lokalen Gradienten um den Punkt herum gewonnen werden. So kann der Partikelfilter zum Beispiel statt mit der ganzen sich aus dem Punkt im Bildbereich ergebenden Kurve mit nur einem Teil dieser Kurve geupdatet werden, oder sogar nur mit dem einen Punkt im Parameterraum, der sich aus dem Punkt im Bildraum und der Richtung ergibt.

Die Laufzeit lässt sich bei diesem Verfahren nun über die Anzahl der Partikel skalieren, wobei bei sehr wenigen Partikeln auch die Wahrscheinlichkeit steigt, Linien

zu verpassen und gar nicht zu erkennen.

7.2.2.3 Clusterverfahren im Hough-Raum

Der Versuch, die Richtungsinformation in das klassische Hough-Verfahren einzubringen, führt zu einem Verfahren, bei dem der Parameterraum wie in Kapitel 7.2.2.1 in Zellen unterteilt ist und iterativ mit den Informationen aus den Linienpunkten gefüllt wird. Allerdings werden nun nicht mehr die Werte in Zellen nicht entlang der ganzen Kurve inkrementiert, sondern nur jeweils die in genau der Zelle, die durch Linienpunkt und Richtung genau bestimmt ist.

Da nun allerdings die Richtungsinformation nicht perfekt, sondern verrauscht, also mit einem zufälligen Fehler versehen ist, geben auf derselben Linie liegende Punkte keinen einzigen Punkt im Parameterraum, sondern eher eine um die richtigen Parameter verdichtete Punktwolke. Dies macht ein Clusterverfahren nötig, zum Beispiel ähnlich den in Kapitel 6.2.4 und Kapitel 5.3.1.2 Beschriebenen.

Wichtig ist in diesem Falle, dass nicht nur eine Zelle, sondern auch ihre Umgebung betrachtet wird. Dadurch kann durch Interpolation ein noch genaueres Ergebnis erzielt werden, als die Auflösung der Akkumulatorzellen liefern würde. Weiterhin wird dadurch verhindert, dass mehrere nahezu parallele Linien erkannt werden, wo eigentlich nur eine liegt. Weiterhin ist wichtig, dass das Clusterverfahren multimodal arbeitet, also in der Lage ist, mehr als nur die größte Ballung zu erkennen. Dies ist unerlässlich, da das Ziel, wie in Kapitel 7.2.1 beschrieben, ja darin besteht, möglichst alle im Bild enthaltenen Linien zu erkennen, um darüber hinaus deren Schnittpunkte liefern zu können.

Vorausgesetzt, die Richtungsinformationen wären nur mit einem nicht allzu starken, rein zufälligen Fehler versehen, wäre dieses Verfahren eine gelungene Verbesserung des Verfahrens der klassischen Hough-Transformation.

Die Effizienz dieses Verfahrens hängt allerdings stark von dem benutzten Clusteralgorithmus ab.

7.2.2.4 Heuristisches Clusterverfahren

Im Gegensatz zu den bisher beschriebenen globalen Ansätzen nutzt das im Folgenden beschriebene Verfahren eine andere Strategie. In mehreren Schritten werden erst die Linienpunkte nach lokaler Ähnlichkeit beurteilt, dann zu Linienfragmenten zusammengesetzt und diese anschließend zu den endgültigen Linien verschmolzen.

Das erste Kriterium für die Beurteilung der Ähnlichkeit ist die gleiche Richtung des Gradienten. Hier wird die Richtung des Gradientenvektors verwendet und Richtungen, die zwar dieselbe Gerade beschreiben würden, aber entgegengesetzt orientiert sind, werden als nicht ähnlich betrachtet. Dies hat den Grund, dass im nächsten Schritt zwischen den beiden Kanten derselben Feldlinie unterschieden werden kann. Auf die Notwendigkeit dessen wird später eingegangen.

Weiterhin wird bei jedem Punktepaar p_1 und p_2 und ihren zugehörigen Richtungen r_1 und r_2 getestet, wie weit p_1 von der durch p_2 und r_2 induzierten Gerade entfernt

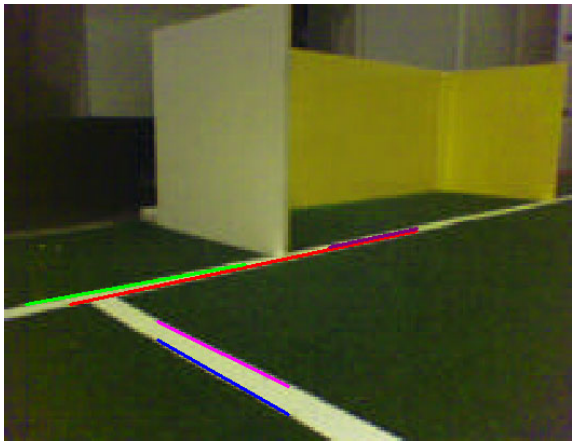


Abbildung 7.13: Gefundene Linienfragmente

ist und umgekehrt. Wird bei einem dieser Tests eine zu große Abweichung erkannt, werden die beiden Punkte als nicht ähnlich betrachtet.

Anschließend werden aus den so entstandenen Clustern gleicher Punkte, beginnend mit dem Größten, Linienfragmente gebildet. Hier zeigt sich, dass ein Mittel aller aus den Gradienten gewonnenen Richtungen eine sichtbar schlechte Annäherung der wirklichen Gerade ergibt. Statt dessen werden aus dem Cluster jeweils die äußersten beiden Punkte gewählt, also der Anfangs- und Endpunkt des Linienfragments, und daraus dessen Richtung bestimmt.

Diese Art der Richtungsberechnung ist auch der oben angedeutete Grund, weshalb die beiden Seiten einer Feldlinie unterschieden werden. Würde diese Unterscheidung nicht vorgenommen werden, könnten die beiden am weitesten voneinander entfernten Punkte auf verschiedenen Seiten der Feldlinie liegen, die eine endliche Breite hat. In dem Falle würde dann aber die gefundene Richtung derjenigen der Diagonalen durch den entsprechenden Feldlinienausschnitt entsprechen, und nicht der wirklichen Richtung der Feldlinie.

Auf Auswirkungen auf die anderen Verfahren durch die häufig starken Abweichungen des Mittels der Gradientenrichtungen zu der wirklichen Linienrichtung wird in Kapitel 7.2.3 weiter eingegangen.

Die nun vorliegenden Linienfragmente beschreiben zum Teil dieselben Feldlinien, und zwar sowohl die unterschiedlichen Seiten als auch einzelne Teile derselben Seite einer Linie, wie auch in Abbildung 7.13 gut zu erkennen ist. Allerdings liegen diese nun mit weit zuverlässigeren Richtungsinformationen vor. In einem weiteren Schritt werden diese zusammengehörigen Linienfragmente nun zu Linien verschmolzen. Dies geschieht nach ähnlichen Kriterien wie bei der Ähnlichkeitsbestimmung der Punkte. Es wird nur bei Richtungsvergleichen noch Rücksicht auf die je nach Lage und Orientierung im Bild unterschiedlich starke Auswirkung der perspektivischen Verzerrung genommen. Weiterhin wird bei der Abstandsberechnung die erwartete Breite einer Feldlinie an der Position im Bild mit einbezogen. Beim Verschmelzen der Linienfragmente kann nun sowohl für die Lage als auch für die Richtung gemittelt

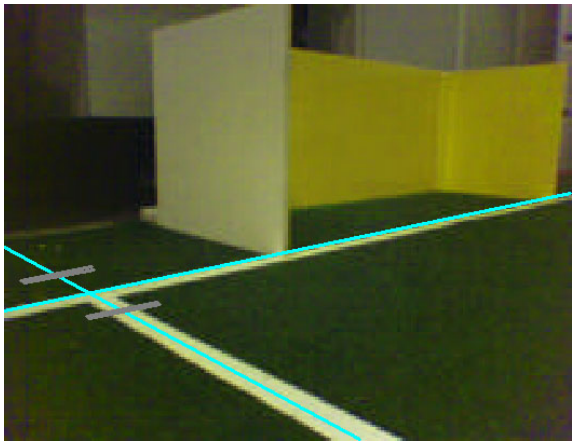


Abbildung 7.14: Die resultierenden Linien und deren Schnittpunkt

werden.

Als Resultat dieses Algorithmus stehen die erkannten Linien wie in Abbildung 7.14 für weitere Berechnungen zur Verfügung. Allerdings kann auch die in den Linienfragmenten enthaltene Information noch an mehreren Stellen benutzt werden, wie in Kapitel 7.3.2.2 und Kapitel 6.5 beschrieben.

7.2.3 Beschreibung der gewählten Lösung

Die ersten drei in Kapitel 7.2.2.1 bis Kapitel 7.2.2.3 beschriebenen Verfahren basieren auf globalen Ansätzen und dem Versuch, die Komplexität in Hinblick auf die Anzahl der Linienpunkte von quadratisch auf linear zu drücken. Dies ist in den meisten Anwendungen ein gutes Vorgehen, in denen zum Beispiel die Linienpunkte gewonnen werden, indem ein Gradientenbild erzeugt wird, und anschließend alle Punkte mit hohem Gradientenwert als Eingabe für die Liniensuche verwendet werden. In unserem Fall allerdings ergeben sich die Linienpunkte aus den in Kapitel 7.2.2 genannten Kriterien und sowieso nur auf den aus Laufzeitgründen wenigen nach ihnen abgesuchten Scanlinien. Weiterhin machen diese Kriterien das Verfahren sehr stabil. Zum Beispiel werden weder an andere Robotern, noch an sehr hellen und somit als Weiß klassifizierten Stellen im Teppich Linienpunkte gefunden.

Wie zu erwarten, sind die gefundenen Punkte also nicht allzu zahlreich, dafür aber einigermaßen zuverlässig. Im Allgemeinen ergeben sich in Spielsituationen durchschnittlich zwischen 15 und 40 Linienpunkte pro Bild. Somit ergibt sich durch obige Verfahren nicht der gewünschte Laufzeitvorteil.

Weiterhin sind die Richtungsinformationen aus den Gradienten der Linienpunkte, bedingt durch die schlechte Bildgewinnung, für diese globalen Verfahren zu stark verrauscht. Wie in Kapitel 7.2.2.4 beschrieben, ist der Fehler oft so groß, dass auch nach Mitteln der Richtungsinformation über mehrere Punkte einer Linie diese gemittelte Richtung noch signifikant von der wirklichen abweicht. Dies beeinflusst besonders das in Kapitel 7.2.2.3 beschriebene Verfahren des Clusterns im Hough-Raum, da

dabei davon ausgegangen wird, dass die Linienpunkte zusammen mit ihren Richtungen Punkte im Parameterraum ergeben, die sich annähernd normalverteilt und nicht allzu weit gestreut um das wirkliche Parametertupel aufhalten. Dies ist nach der Hough-Transformation der Linienpunkte mit ihrer stark verrauschten Richtungsinformation allerdings nicht mehr der Fall. So führte dieser Ansatz auch zu keinen guten Ergebnissen.

Ebenso beeinträchtigt wird dadurch der in Kapitel 7.2.2.2 beschriebene Partikelfilter. Bei einer Partikelanzahl von 30 wurde nie mehr als eine der Linien im Bild mit unter 10° Richtungsabweichung gefunden. Bei drastisch höherer Partikelzahl verbesserte sich dies zwar, allerdings auf Kosten der Laufzeit, die dann nicht mehr akzeptabel war.

Letztlich erweist sich der in Kapitel 7.2.2.4 beschriebene Algorithmus den anderen vorgestellten Verfahren sowohl in Laufzeit als auch in Genauigkeit als weit überlegen. Der enthaltene Teil mit zur Anzahl der Linienpunkte quadratischer Komplexität fällt, wie bereits erwähnt, durch deren geringe Anzahl nicht stark ins Gewicht. Besonders nach Laufzeitoptimierung der entsprechenden Schleifen ist dies ohne weiteres akzeptabel. Auf einem AIBO werden mit dem Algorithmus in unter 2 ms meist sowohl alle Linien gefunden als auch deren Schnittpunkte berechnet. Außerdem lässt sich aufbauend auf diesem Algorithmus sehr effizient der Mittelkreis finden, wie in Kapitel 7.3.2.2 näher beschrieben. Auch dies ist in den 2 ms Laufzeit bereits enthalten.

7.2.4 Finden von Linienkreuzungen

Sind die im Bild zu sehenden Linien erstmal erkannt und in mathematischer Form gegeben, so können die entsprechenden Kreuzungen einfach berechnet und durch weiteres Scannen charakterisiert werden, wie auch in Abbildung 7.14 zu sehen ist. Die Charakterisierung besteht aus der Feststellung, auf welchen Seiten der Linienkreuzung sich die Feldlinien fortsetzen. Diese Unterscheidung liefert allerdings nur eindeutige Ergebnisse, wenn die Linienkreuzungen gut sichtbar sind. Manchmal sind diese durch weite Entfernung und dementsprechende Nähe zu Objekten im Hintergrund schwierig zu unterscheiden, von Robotern oder dem Ball teilweise oder auch ganz verdeckt, oder liegen auch einfach gar nicht im Bild. In letzterem Fall wird der Kreuzungspunkt dementsprechend markiert. In den anderen Fällen wird die entsprechende Richtung als nicht bekannt gewertet.

Auf Grund dieser mangelnden Eindeutigkeit wird die weitere Interpretation, etwa als L-Kreuzung wie in den Spielfeld- oder Strafraumecken, oder als T-Kreuzung wie an der Strafraumbasis oder dem Schnitt von Mittel- und Außenlinie, vom ImageProcessor nicht vorgenommen und dem SelfLocator überlassen.

7.3 Mittelkreis

Das meiste, was Anfang Kapitel 7.2 über die Linien gesagt wurde, gilt auch für den Mittelkreis. Auch er stellt eine feste Größe auf allen Fußballfeldern dar und wird auch in Zukunft zur Verfügung stehen. Genau wie für die Linien braucht der Roboter auch hier nicht vom Ball aufzuschauen. Zwar wird der Mittelkreis weit seltener im wirklichen Spiel überhaupt im Bild zu sehen sein, dann allerdings können im Falle einer guten Erkennung auch viel mehr Informationen gewonnen werden. Wie in Kapitel 6.4.2 und Kapitel 10.1.3.1 genauer beschrieben, kann ein gut erkannter Mittelkreis die möglichen Aufenthaltsorte auf dem Spielfeld auf zwei punktsymmetrische Positionen reduzieren.

7.3.1 Problembeschreibung

Der Mittelkreis ist wohl eines der am schwierigsten zu erkennenden statischen Objekte auf dem Spielfeld. Dies kommt zum Einen daher, dass mit der derzeitigen Kamera des AIBO schon für das menschliche Auge bei einer Entfernung von mehr als 2 m auf dem Kamerabild nur noch ein Farbfleck zu erkennen ist, zum Anderen auch an der auf Scanlinien basierenden Scanstrategie, durch die letztlich einerseits nur ein Bruchteil der Pixel bearbeitet werden muss, was Rechenzeit spart, andererseits aber auch aufschlussreiche Strukturen durch das grobmaschige Netz der Scanlinien rutschen und unbeachtet bleiben können.

Außerdem ergeben sich allein schon durch die Form weitere Schwierigkeiten. Nach der Projektion wird der Mittelkreis nämlich zu einer Ellipse, deren Größe von der Entfernung zur Kamera abhängt. Diese Ellipse wird weiterhin noch mit je nach Blickrichtung auf den Mittelkreis anderem Winkel von der Mittellinie geschnitten.

7.3.2 Ansätze zur Lösung des Problems

Bei beiden im Folgenden beschriebenen Verfahren zur Erkennung des Mittelkreises wurde auf vorheriges intensives Scannen des Bildes aus Laufzeitgründen verzichtet. Stattdessen bestehen die als Eingabe nötigen Informationen größtenteils aus den im ImageProcessor-Hauptteil gefundenen Linienpunkten, wie schon in Kapitel 7.2.2 genauer beschrieben. Zusätzliche Scanvorgänge werden zur späteren Verifikation des Ergebnisses durchgeführt.

7.3.2.1 Suche im Hough-Raum

Die Hough-Transformation wird angewendet, um Kandidaten für den Mittelpunkt des Mittelkreises zu ermitteln. Schwierigerweise müssen auch Kreise erkannt werden, die nur teilweise im Bild sichtbar sind, wodurch zum Beispiel Feldecken fälschlicherweise als Kreiskandidaten angenommen werden können. Daher müssen die ermittelten Kandidaten auf weitere Bedingungen geprüft werden, die im Folgenden noch erläutert werden.

Die Hough-Transformation Zur Ermittlung des Punktes im Bild, der die meisten Eingabepunkte als Kreispunkte besitzt, wird die Kreisformel $(x-x_m)^2+(y-y_m)^2 = r^2$ benutzt. In der Formel sind die Eingabepunkte (x, y) und der Radius r gegeben. Die beiden Variablen des Mittelpunktes (x_m, y_m) sind unbekannt. Für „jedes“ y_m wird nun für jeden Eingabepunkt das dazugehörige x_m berechnet. Die Anzahl der zu überprüfenden y_m reduziert sich auf die Anzahl der Zeilen des Gitters, das auf den sichtbaren Bereich des AIBOs gelegt wird. Für jede Zeile des Gitters, wird der y-Mittelwert verwendet. x_m berechnet sich dann folgendermaßen:

$$\pm(x - x_m) = +\sqrt{r^2 - (y - y_m)^2} \tag{7.1}$$

$$x_{m1} = x - \sqrt{r^2 - (y - y_m)^2} \tag{7.2}$$

$$x_{m2} = \sqrt{r^2 - (y - y_m)^2} - x \tag{7.3}$$

Falls das Ergebnis der Wurzel in Gleichung (7.1) negativ ist, dann ist der Eingabepunkt zu weit von y_m platziert und ist damit kein möglicher Kreispunkt. Ansonsten ist der Wert der Wurzel positiv und die zu y_m gehörigen x_m werden ausgerechnet. Die berechneten Koordinatenpaare werden vermerkt, indem die Zellen des Gitters die Anzahl der Koordinatenpaare zählt, die in der entsprechenden Zelle liegen.

Nachdem alle Koordinatenpaare vermerkt worden sind, kann nun im Gitter abgelesen werden, welche Zelle bzw. Zellen die meisten Kreispunkte zählt bzw. zählen. Falls es weniger als 4 Kreispunkte sind, dann wird an dieser Stelle abgebrochen, da diese Punkte wahrscheinlich keine Kreispunkte, sondern Linienpunkte sind. Ansonsten wird abhängig von der ermittelten Zellenanzahl unterschiedlich fortgefahren.

Die in den folgenden zwei Abschnitten erwähnten Parameter wurden durch mehrere Tests auf dem Feld bestimmt.

Eine Zelle Bei einer Zelle gibt es nur einen zu prüfenden Mittelpunktskandidaten, der die Koordinaten des Mittelpunktes der Zelle besitzt.

Im Falle, dass mehr als 35 Kreispunkte vorhanden sind, gilt der Kreis als gefunden. Ansonsten wird gezählt, wieviele Eingabepunkte innerhalb einer ϵ-Umgebung um die Kreislinie des Mittelpunktkandidaten liegen. Die ϵ-Umgebung ist dabei so groß wie die maximale Zellenlänge des Gitters. Falls mehr als 35 Kreispunkte gefunden werden, gilt der Kreis als gefunden. Andernfalls wird sternförmig nach Grün-Weiß-Grün-Übergängen gesucht. Dabei müssen 6 Übergänge gefunden werden, damit der Kreis als gefunden gilt.

Mehrere Zellen Es gibt mehrere Zellen, die die gleiche maximale Anzahl an Kreispunkten gezählt haben. Für den Fall, dass mehr als 15 Kreispunkte gefunden worden sind, ist es wahrscheinlich, dass alle Kandidaten innerhalb des Mittelkreises, konzentriert an der Stelle des Mittelpunktes, platziert sind. Zur Überprüfung wird die durchschnittliche Entfernung jedes Kandidaten zu den anderen berechnet. Beträgt die kleinste durchschnittliche Entfernung 20 cm so gilt der Kreis als gefunden und der Kandidat mit der kleinsten durchschnittlichen Entfernung ist Mittelpunkt.

Abbildung 7.15: Linienfragmente am Mittelkreis

Wurden weniger als 15 Kreispunkte gefunden, so wird für jeden Kandidat die Anzahl der Eingabepunkte gezählt, die sich innerhalb der ϵ -Umgebung derer Kreislinien befinden. Geht nach der Abzählung ein Kandidat mit der maximalen Anzahl an Kreispunkte hervor, so gilt dieser als Kreis, wenn mehr als 35 Kreispunkte gefunden worden sind, ansonsten wird nach Grün-Weiß-Grün-Übergängen gesucht. Bleiben nach der Abzählung noch mehrere Kandidaten übrig und die Anzahl der gezählten Kreispunkte ist größer als 11, dann wird jeder einzelne Kandidat nach Grün-Weiß-Grün-Übergängen überprüft. Der Kreis gilt als gefunden, wenn ein Kandidat vorhanden ist, der 6 Grün-Weiß-Grün-Übergänge besitzt.

7.3.2.2 Tangenten-Heuristik

Bei diesem Verfahren wird die bereits vom Linienerkennungsalgorithmus geleistete Vorarbeit ausgenutzt. Wie in Kapitel 7.2.2.4 beschrieben, wird dort nicht sofort auf komplette Linien, sondern aus dem Grund der Ungenauigkeit der Richtungsinformationen der Gradienten erst einmal auf Linienfragmente geschlossen. Ist eine geschwungene Linie im Bild zu sehen, so bilden diese Linienfragmente näherungsweise Tangenten an diese Kurve.

Diese Eigenschaft kann nun ausgenutzt werden, indem in einem zusätzlichen Zwischenschritt der Linienerkennung alle vorhandenen Linienfragmente als potentielle Tangenten eines auf die Bildebene projezierten Kreises aufgefaßt werden. Abbildung 7.15 zeigt die Linienfragmente, die sich bei einem Mittelkreis im Bild ergeben können.

Als erstes müssen nun die Linienfragmente wieder auf das Feld projeziert werden. In diesem Schritt ist eine genaue Kenntnis der internen und externen Kameraparameter unerlässlich, da ein falsch kalibrierter Horizont zu einer falsch skalierten Projektion führt und somit zu einer deutlich schlechteren Erkennungsrate.

Von diesen in Feldkoordinaten vorliegenden Strecken können nun je paarweise die Mittelsenkrechten geschnitten und daraufhin geprüft werden, ob der sich ergebende Schnittpunkt zu beiden Strecken den richtigen Abstand besitzt, also mit einer

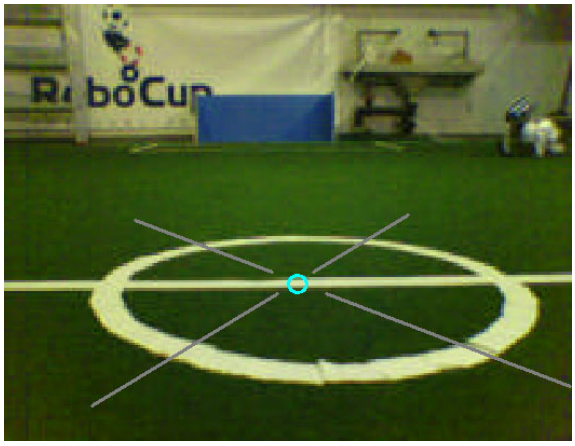


Abbildung 7.16: Abschließende Überprüfung des Mittelpunktkandidaten

gewissen zulässigen Abweichung dem bekannten Radius des Mittelkreises entspricht.

Ist in dem Bild wirklich der Mittelkreis zu sehen, so ergeben sich meist gleich mehrere solcher Kandidatenpositionen, die zusätzlich auch sehr nah beieinander liegen. Aber auch in anderen Situationen können durch einfachen Zufall und verrauschte Daten Linienfragmente ein- und derselben Linie nicht ganz parallel liegen und Schnittpunkte ergeben, die dieses erste Kriterium erfüllen.

In einem ersten Filterschritt kann von diesen Kandidaten deren Abweichung voneinander und vom erwarteten Radius überprüft werden. Sollten diese Abweichungen einen bestimmten Wert überschreiten, wird die Wahrscheinlichkeit, wirklich einen Mittelkreis im Bild zu haben, deutlich niedrig sein.

Besonders in dem Falle, dass sich nur ein einziger Kandidat ergibt, greift dieses Kriterium natürlich nicht. Ausgehend davon, dass der Mittelkreis im Bild enthalten ist, kann angenommen werden, dass mindestens eines der Linienfragmente keine Tangente zum Mittelkreis ist, sondern stattdessen zu der Mittellinie gehört. Dementsprechend kann der Mittelpunktkandidat damit verifiziert werden, dass er sehr nahe bei der Mittellinie liegen muss.

Eine anschließende Projektion auf diese Mittellinie, wie in Abbildung 7.16 geschehen, kann in den meisten Fällen auch die Genauigkeit des Ergebnisses verbessern. Zusätzlich kann die Information über diese Mittellinie als Ausrichtung für den gefundenen Mittelkreis hergenommen werden, wie man in Abbildung 7.17 sehen kann.

Ein abschließender Verifikationsscan wie in Abbildung 7.16 sorgt letztlich dafür, dass eventuell noch vorhandene falsche Kandidaten ausgeschlossen werden, sowie solche, die versehentlich auf die Kreislinie selbst projiziert wurden.

7.3.3 Beschreibung der gewählten Lösung

Beide Ansätze wurden auf dem Feld unter unterschiedlichen Bedingungen, wie Abstand zum Mittelkreis, sichtbarer Anteil des Kreises, mit bzw. ohne Kopfbewegung und stehender bzw. sich fortbewegender AIBO getestet. Es ergab, dass die Erken-

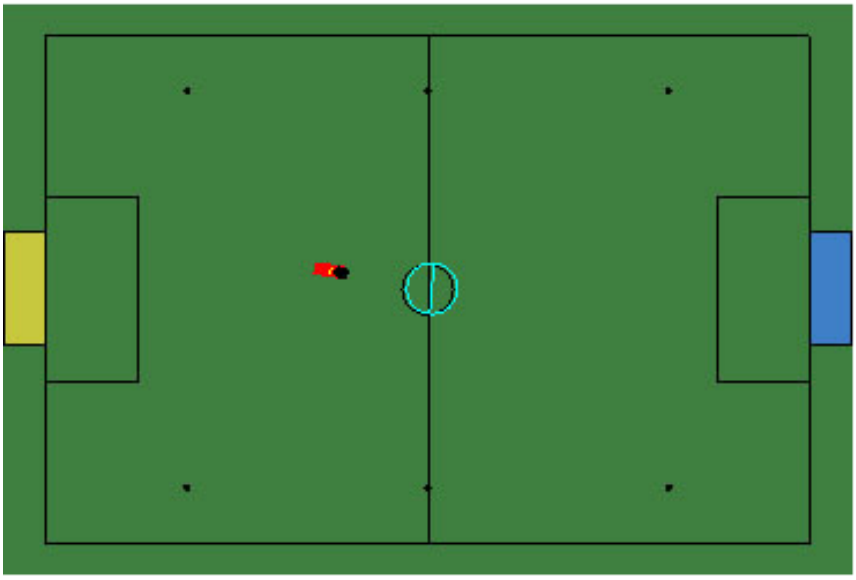


Abbildung 7.17: Der erkannte Mittelkreis mit Ausrichtung

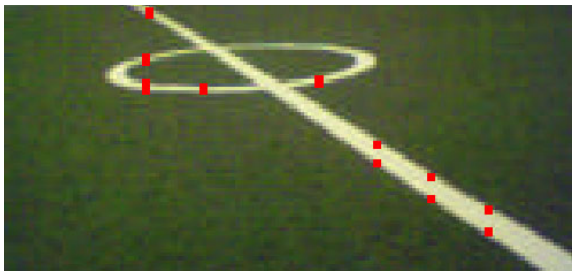


Abbildung 7.18: Ab einer Entfernung von 1 m stehen zu wenige Linienpunkte für die Kreiserkennung zur Verfügung.

nung des Mittelkreises bei beiden Ansätzen auf 1 m Abstand eingeschränkt ist. Der Grund liegt darin, dass beide Ansätze dieselben Eingabepunkte erhalten, die ab einem Abstand von 1 m auf Grund der beschränkten Scanlinienanzahl, zu wenig Kreispunkte enthalten, siehe Abbildung 7.18. Die durchschnittliche Abweichung der dabei erkannten Mittelpunkte beträgt beim Hough-Ansatz 5 cm und beim Tangenten-Ansatz 3 cm. Die Erkennung von Kreisen, die nicht vollständig im Bild sind, ist nur im Tangenten-Ansatz, unter der Vorraussetzung, dass die Mittellinie innerhalb des Kreises sichtbar ist, möglich. Der Tangenten-Ansatz beansprucht kaum Zeit, während der Hough-Ansatz 1 ms benötigt. Das Erkennen von falschen Kreisen ist in beiden Ansätzen äußerst selten, wofür aber die Erkennungsrate einbüßen musste.

Aus den vorher genannten Testkriterien hat der Tangenten-Ansatz bessere Ergebnisse geliefert und wird damit für die Kreiserkennung verwendet.

7.4 Robotererkennung

Im modernen Roboterspiel wird es immer wichtiger zu wissen, wo die einzelnen Gegner, aber auch die eigenen Spieler, relativ zum eigenen AIBO stehen. Gerade bei dem für die Zukunft angestrebten Passspiel ist es zu vermeiden, dass ein Ball verloren geht, weil ein Gegner im Weg steht, oder der angespielte AIBO nicht an der von ihm kommunizierten Position steht. Weiterhin hat man mit einem zuverlässigen Modell der Roboterpositionen ganz neue taktische Möglichkeiten. Es können Verteidiger umspielt und gezielt am Torwart vorbei geschossen werden.

7.4.1 Problembeschreibung

Die Erkennung von AIBOs unterscheidet sich auf Grund der geometrischen Komplexität deutlich von der Erkennung anderer Objekte auf dem Spielfeld. Im Gegensatz zu Toren, Bällen und Landmarken sieht das Trikot von jeder Richtung anders aus. Leider sind rot und auch blau zwei Farben, die häufig im Hintergrund zu finden sind seit die weißen Begrenzungsbanden weggefallen sind und der Blick in die Zuschauer frei ist. Zusätzlich erschwerend kommt bei den blauen Trikots hinzu, dass auf Grund der schlechten Kamera mit ihrer Blauverschiebung im Randbereich das blaue Trikot schlecht von dunklen Passagen im Hintergrund zu unterscheiden ist.

7.4.2 Ansätze zur Lösung des Problems

Im Laufe der Entwicklung der Robotererkennung wurde schnell klar, dass sich die gesamte Aufgabe grob in folgende Teilprobleme zerlegen lässt:

7.4.2.1 Suchen von Trikotstücken

Zum Suchen von Trikotstücken wird zunächst, wie bei anderen Objekten auch, nach der spezifischen Farbe gesucht. Um falsche Erkennung gleich im ersten Schritt zu vermindern und möglichst wenig Zeit in den folgenden Schritten zu verschwenden,

Abbildung 7.19: Beim Clustern werden einzelne Scanlinien verbunden und Farben zwischen ihnen untersucht.

werden bestimmte Bedingungen abgeprüft. Als praktikabel erwies es sich zu fordern, dass eine gewisse Anzahl aufeinander folgende Bildpunkte die Farben Blau bzw. Rot aufweisen. Allerdings existiert dabei ein Trade-Off zwischen hohem Rechenaufwand und einer geringen Anzahl an Perzepten.

7.4.2.2 Clustern zu Trikots

Nachdem einzelne Teile von einem Trikot gefunden worden sind, ist der nächste Schritt der, sie zu einem Stück zusammenzufügen. Dazu wird eine Verbindungslinie zwischen zwei gefundenen Trikotstücken errechnet und auf ihr eine Farbverteilung generiert. Wenn es sich wirklich um einen zusammenhängenden AIBO handelt, darf sich zwischen den einzelnen Stücken allenfalls die Farbe des Trikots und die Farbe Weiß in größerer Anzahl finden lassen. Ein längeres Stück Grün legt zum Beispiel nahe, dass es sich um zwei verschiedene AIBOs handelt (siehe Abbildung 7.19).

7.4.2.3 Filtern

Das abschließende Filtern soll verhindern, dass Gegenstände oder Personen im Hintergrund oder Teile des Feldes fälschlich als AIBO erkannt werden.

Finden von Fußpunkten: Ein erster Schritt im Zuge des Filters ist das Finden der zu den Trikotstücken korrespondierenden Fußpunkte[3]. Dazu wird auf einer Scanlinie senkrecht zum Horizont und ausgehend vom Trikotpunkt nach grün gesucht. Wenn eine Mindestanzahl grüner Pixel gefunden wurde, werden die Bildkoordinaten in lokale Roboterkoordinaten (siehe Anhang A) umgerechnet (siehe Abbildung 7.20).

Finden von Grün Übergängen Gleichzeitig werden in der Hauptscanmethode des ImageProcessors Farbübergänge von Grün oder nach Grün gesucht. Diese wer-

[3]Ein Fußpunkt eines beliebigen Punktes A in einem Bild, ist seine Projektion auf die xy-Ebene

Abbildung 7.20: Beim Scannen werde Trikotkandidaten ab einer gewissen Länge gespeichert und gemerkt.

Abbildung 7.21: Jeder Anfang und jedes Ende eines grünen Streifens wird gespeichert.

den dann von links nach rechts in einer Liste gespeichert (siehe Abbildung 7.21). Mit diesen Infomationen können später Abstandsmaxima und -minima gefunden werden.

Finden von U-Shapes: Roboter haben im Gegensatz zu Störungen im Hintergrund den entscheidenden Unterschied, dass sie auf Füßen stehen, die sich mehr oder weniger an den Ecken der Roboters befinden und eine charakteristische Form besitzen. Diese kann man versuchen zu erkennen. Ausgehend von der Liste der Grün-Übergänge wird nach einem Maximum im Abstand der Übergänge vom Horizont gesucht. Dann kann verglichen werden, ob das bisher erkannte Perzept sich über oder zumindest nahe eines solchen Maximas befindet und es wird im Zweifelsfall die Bounding Box[4] um den gefundenen AIBO mit Hilfe

[4]Eine Bounding Box ist das kleinste Rechteck, das die maximalen Ausmaße des AIBOs noch komplett umfasst

Abbildung 7.22: Erkannte Fußpunkte.

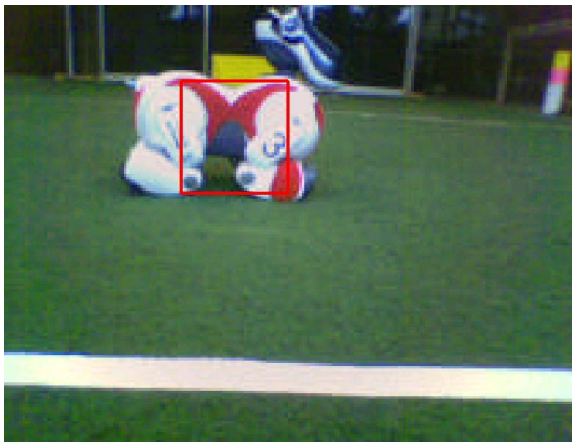


Abbildung 7.23: Die erkannten Ausmaße des Roboters.

der gefundenen Füße angepasst (siehe Abbildung 7.22).

Geometrische Plausibilität: Da die maximalen Ausmaße eines Roboters bekannt sind, kann man alle gefundenen Perzepte ausschließen, deren Bounding Box wesentlich größer oder kleiner als ein AIBO sind (siehe Abbildung 7.23).

7.4.3 Beschreibung der gewählten Lösung

Trotz Einsatz aller oben genannten Ideen, war es in der relativ kurzen verbleibenden Zeit nicht möglich, eine Erkennungsrate der blauen Roboter über 50% zu liefern. Rote Roboter konnte mit einer Wahrscheinlichkeit von ca. 70% erkannt werden, letztendlich wurde der Code aber – unter anderem auch mangels passender Modellierung – nie in einem Spiel benutzt.

8 XTC

Dieses Kapitel beschreibt die Verhaltensarchitektur und gibt einen Einblick in den Übersetzer der Sprache *XTC*.

8.1 Einleitung

Das Softwaremodul, das den AIBO während eines Fußballspiels steuert, heißt *BehaviorControl*. Es verwendet die von den übrigen Modulen generierten Informationen (zum Beispiel die eigene Position, die Ballposition oder den Winkel zum gegnerischen Tor) als potenzielle Eingaben und erzeugt als Ausgabe *WalkRequests* (siehe Kapitel 3.3.4). Auch die Kopfbewegung und das Schießen des Balls werden vom BehaviorControl veranlasst.

Es hat sich gezeigt, dass die Implementierung des Verhaltensprogramms in C++ zu stark undurchsichtigem und schlecht wartbaren Quelltexten führt. Es ist schwierig mit einer objektorientierten Sprache diese Aufgabe zu bewältigen, da der enorme Freiheitsgrad, den eine Hochsprache bietet, auch ein große potenzielle Fehlerquelle darstellt. Die Einschränkung auf einen endlichen Automaten zur Kodierung der Spielsituation und Verhaltensweisen hat sich als sinnvoll und praktikabel erwiesen, wenn man eine Schachtelung von Zuständen in Unterzustände zulässt. Dies ermöglicht dann goße Teile des Automaten mehrfach zu verwenden.

XABSL ist eine für autonome Agentensysteme entwickelte Verhaltensbeschreibungssprache von Martin Lötzsch und realisiert genau eine solche Automatenstruktur (siehe [13]). Zustände, die andere enthalten, heißen *Options*. Jene, die sich nicht weiter schachteln lassen, werden *BasicBehaviors* genannt. Abbildung 8.1 zeigt ein Beispiel für eine *Option*. Die Pfeile zu Rechtecken stellen Übergänge zu anderen *Options* dar, die Pfeile zu Ellipsen die Übergänge zu *BasicBehaviors*. Diese *BasicBehaviors* sind wiederum in C++ implementiert. So können darin Berechnungen für die Generierung eines *WalkRequests* leicht durchgeführt werden.

Die Sprache und die damit beschriebenen Automaten erzwingen eine hierarchische Gliederung und einen strukturierten Zugriff auf die von den überigen Modulen bereitgestellten Daten. XABSL übersetzt das aus mehreren XML-Dateien bestehende Verhaltensprogamm mit einer XSL-Transformation in den *Intermediate Cod*e, aus dem zur Laufzeit auf dem Hund die Datenstruktur für die *State Machine* erzeugt wird. Ausserdem wird eine detaillierte Dokumentation des Programms im HTML-Format erzeugt.

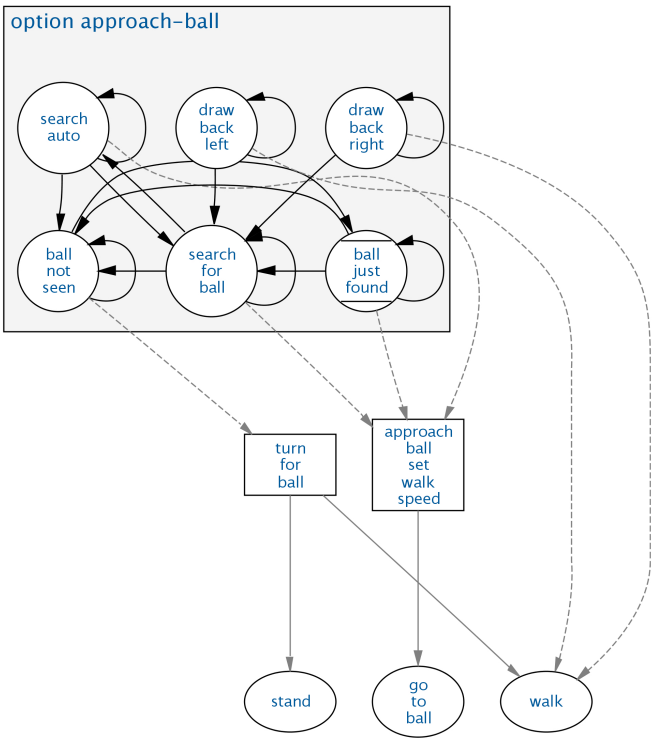


Abbildung 8.1: Beispiel einer XABSL-*Option*

8.2 Motivation

Da die Programmierung jedoch in XML erfolgt, sind die Quelltexte ohne die generierte Dokumentation nur schwer zu durchschauen und verhältnismäßig lang. Dies führte zu der Idee, eine Eingabesprache für XABSL zu entwerfen, die verbreiteten Programmiersprachen mehr gleicht, also eine einfachere Syntax besitzt. Das bestehende Verhaltensprogramm soll außerdem in die neue Sprache XTC übersetzt werden.

XTC ähnelt C++ oder Java Quelltext, hat aber wesentlich weniger Sprachelemente. Es hat gegenüber XABSL den Vorteil, dass es weniger redundant ist, also beispielsweise keine separaten Deklarationen und Definitionen benötigt und wesentlich weniger syntaktischen (XML-Tags) und semantischen (Präfixe von Variablennamen) Ballast mit sich rumschleppt. Das macht besonders lange Bedingungen wesentlich einfacher zu begreifen:

```
/** kick possible */
if (ball.just_seen && abs(value = ball.seen.relative_speed.y) < 150 &&
    ball.seen.relative_speed.x < 150 &&
    retrieve_kick(angle = @angle, table_id = @table_id) != action.nothing)
```

Im Vergleich dazu das XABSL Äquivalent:

```
<condition description="kick possible">
```

```
  <and>
    <boolean-input-symbol-ref ref="ball.just-seen"/>
    <less-than>
      <decimal-input-function-call ref="abs">
        <with-parameter ref="abs.value">
          <decimal-input-symbol-ref ref="ball.seen.relative-speed.y"/>
        </with-parameter>
      </decimal-input-function-call>
      <decimal-value value="150"/>
    </less-than>
    <less-than>
      <decimal-input-symbol-ref ref="ball.seen.relative-speed.x"/>
      <decimal-value value="150"/>
    </less-than>
    <not-equal-to>
      <decimal-input-function-call ref="retrieve-kick">
        <with-parameter ref="retrieve-kick.angle">
          <option-parameter-ref ref="approach-and-kick.angle"/>
        </with-parameter>
        <with-parameter ref="retrieve-kick.table-id">
          <option-parameter-ref ref="approach-and-kick.table-id"/>
        </with-parameter>
      </decimal-input-function-call>
      <constant-ref ref="action.nothing"/>
    </not-equal-to>
  </and>
</condition>
```

8.3 Aufbau

Der XTC Compiler ist in Ruby[1] implementiert, was sich auf Grund des eleganten Umgangs mit Zeichenketten und dem hohen Maß an Flexibilität und Objektorientiertheit anbietet. Nachteilig ist die Geschwindigkeit der interpretierten Sprache beim Kompilieren und die geringe Verbreitung von Ruby für die Wartbarkeit des Übersetzers.

Als Ausgabeformate gibt es

- *Intermediate Code*, der von der XABSL-Engine ausgeführt werden kann.
- *Debug Symbols*, die von RobotControl und RobotControl 2 zum Debuggen von XABSL benötigt werden.
- XABSL XML-Dateien als Austauschformat und zur Nutzung des bestehenden Dokumentationsverfahrens
- eine Symboldatei, die von SciTE[2] und Microsoft™ Visual Studio 2003[3] zum Syntax-Highlighting von XTC Code verwendet wird

[1] http://www.ruby-lang.org
[2] http://www.scintilla.org/SciTE.html
[3] http://msdn.microsoft.com/vstudio

Ein Compiler besteht im Allgemeinen aus zwei Phasen: der Analyse, in der der Quelltext eingelesen und eine Baumstruktur daraus erzeugt wird, und der Synthese, die diesen Baum in ein Ausgabeformat umwandelt. In der Analysephase kommt als erstes ein Scanner zum Einsatz, der den Quelltext in Zeichenklassen, sogenannte Tokens einteilt. Das sind beispielsweise Zahlen, Bezeichner und die verschiedenen Operatoren und Schlüsselwörter. Diese Kette von Tokens wird dann weitergereicht an den Parser, der anhand einer gegebenen Grammatik für eine Sprache diese auf syntaktische Korrektheit überprüft. Üblicherweise findet gleichzeitig auch die semantische Analyse statt, da der Parser, während er den Token-Stream auf grammatikalische Regeln abbildet, die Möglichkeit bietet Einsprungpunkte in der Grammatik zu definieren. So können alle Sprachkonstrukte einzeln behandelt werden und zum Beispiel Deklarationen erkannt und in Tabellen abgespeichert werden. Für andere Sprachkonstrukte, wie Berechnungen und Funktionsaufrufe wird eine Baumstruktur angelegt – der sogenannte Syntaxbaum – in der sich viele aus dem Quelltext extrahierten Informationen wiederfinden. In der Synthesephase wird aus diesem Syntaxbaum ein Ausgabeformat erzeugt. Diese Phase kann je nach Anwendungsfall sehr variieren, zum Beispiel die Erzeugung von Zwischencode oder Codeoptimierung, bevor ein Maschinencode erzeugt wird, umfassen.

Die Pakete REXML[4] und RACC[5] werden zum Parsen von XML-Dateien und zur maschinellen Generierung eines Parsers verwendet.

Der Compiler besteht aus diesem, einem Scanner und vor allem aus dem Teil des Parsers, der aus den Aufrufen für einzelne Grammatikregeln einen Teil vom Syntaxbaum erstellt. Dies passiert unter Zuhilfenahme von kontextabhängig agierenden Spezialisten, die zum Beispiel semantische Daten zwischenspeichern oder nur lokal definierte Symbole zugreifbar machen. Der Compiler parst die XTC-Quelltexte von einer Hauptdatei ausgehend und löst dabei Referenzen auf Options-, Funktions-, State- und Variablennamen auf. In einem zweiten Schritt wird für jeden Quelltext ein Syntaxbaum erstellt, der einer Darstellung in XML strukturell entspricht. Im Fall von XTC stellt der Syntaxbaum beinahe die XML-Repräsentation von XABSL dar. Daher beschränkt sich der Syntheseaufwand bei der Erstellung von XML auf das Traversieren des Syntaxbaums. Ähnliches gilt für den *Intermediate Code*.

Weiterhin unterstützt der Compiler ein inkrementelles Kompilieren der XTC Quelltexte. Dies geschieht durch das Erstellen eines Abhängigkeitsgraphen für alle Symbole während des Kompilierens. Wenn eine Datei zwischen zwei aufeinanderfolgenden Kompilieraufrufen geändert wurde, werden alle in dieser Datei definierten Symbole als geändert betrachtet und auch alle Dateien neu kompiliert, die diese verwenden. Auf Grund der Strukturierung von XABSL werden in der Regel aber nur wenige Dateien neu übersetzt.

Zum Testen der Korrektheit des erstellten Übersetzers wurde über einen längeren Entwicklungszeitraum der bestehende XML- XABSL-Code nach XTC übersetzt und dann in XML zurückübersetzt und verglichen. Ebenso wurde der *Intermediate Code* direkt aus XABSL und über den „Umweg“ XTC generiert und verglichen. So

[4] `http://www.germane-software.com/software/rexml`
[5] `http://i.loveruby.net/en/prog/racc.html`

ließ sich die Korrektheit an einem komplexen Beispiel verifizieren und mögliche Entwicklungsfehler vermeiden. Zusätzlich gibt es noch Testfälle, bei denen gezielt das Auftreten von Fehlern geprüft wird.

Um die Programmierung einfach zu gestalten, wird für den Text-Editor SciTE eine Datei mit Schlüsselwörtern der Sprache und Funktions- und Variablennamen generiert. Dieser bietet dann mögliche Wortvervollständigungen zur Wahl. Für die Integration in den Editor von MSVisualStudio wurde ein *Language-Service* für XTC erstellt, der Syntax-Highlighting für die Quelltexte bietet und eine Vervollständigung ähnlich wie SciTE auf der Basis einer Wortdatei realisiert.

9 DebugTools

Um effizient auf dem AIBO entwickeln zu können, ist es unumgänglich Werkzeuge zu besitzen, die den aktuellen Status des Roboters darstellen können.

9.1 StatusBroadcast

Der StatusBroadcast ermöglicht es, grundlegende Informationen über den aktuellen Roboterzustand zu versenden, ohne vorher mit dem Roboter per WLAN verbunden zu sein.

9.1.1 Motivation

Bisher war es immer nur möglich, Statusinformationen des Roboters zu erhalten, solange eine Verbindung über das WLAN zu ihm bestand. Dies ist für gewisse Informationen, die beispielsweise laufzeitkritisch sind, nicht unbedingt gewünscht, da selbst das Verbinden auf den Roboter bereits dazu beiträgt, die Ergebnisse zu verfälschen. Außerdem gibt es Informationen, die bereits beim Booten des Roboters zur Verfügung stehen sollten, und nicht erst explizit aktiviert werden müssen. Dazu gehört zum Beispiel die aktuelle Position des Roboters auf dem Spielfeld sowie seine Rotation.

9.1.2 Aufbau des StatusBroadcast

Der StatusBroadcast sendet seine Daten per UDP-Broadcast in das Netzwerk. Somit ist jeder angeschlossene Rechner in der Lage, diese Daten zu empfangen und zu interpretieren. Um die Netzwerkressourcen zu schonen und nicht eine unnötig hohe Datenrate zu erzeugen, wird der StatusBroadcast nur alle 30 Roboterframes (entspricht ca. 1 Sekunde) gesendet. Dies reicht für die meisten Informationen vollkommen aus und belastet das Netz nur unwesentlich.

9.1.3 Verwendetes Datenformat des StatusBroadcast

Die Daten des StatusBroadcast werden als StatusBroadcast-Messages versendet. Jede dieser Nachrichten enthält eine vorangestellte Größenangabe der Nachricht, einen eindeutigen Typ, sowie ein Nutzdatenfeld. Anhand der Typsignatur kann RobotControl 2 die Daten entsprechend interpretieren. Dafür wurde der *StatusBroadcastManager* implementiert.

9.2 RobotControl 2

RobotControl 2 ist eines der verwendeten DebugTools . Es ist der Nachfolger des bisher verwendeten RobotControl das ein neues Framework für das Debuggen zur Verfügung stellt.

9.2.1 Motivation

In der Vergangenheit wurde für das Debugging des Roboters das Programm RobotControl verwendet. RobotControl stellt dem Benutzer alle wichtigen Zustandsinformationen des Roboters zur Verfügung und bietet die Möglichkeit den Roboter zu steuern, nachdem es mittels WLAN zu ihm verbunden ist. Nachdem RobotControl nach und nach durch das Hinzufügen neuer Funktionalität immer unübersichtlicher wurde, wurde beschlossen, einen klar strukturierten Nachfolger zu entwickeln. RobotControl 2 sollte eine bessere Struktur beinhalten und eine Trennung zwischen Roboter-Code und DebugTools-Code vollziehen. Hierzu wird in RobotControl 2 kein Roboter-Code eingebunden sondern die wichtigen Teile für die Kommuniktation erneut implementiert. RobotControl 2 wurde in C# implementiert, um unter anderem die Übersetzungszeiten zu reduzieren und die neuen Möglichkeiten des .Net Frameworks (siehe [14]) ausnutzen zu können.

9.2.2 Aufbau

RobotControl 2 hat eine klare Gliederung in 3 Teile (Framework, Manager, UserControls).

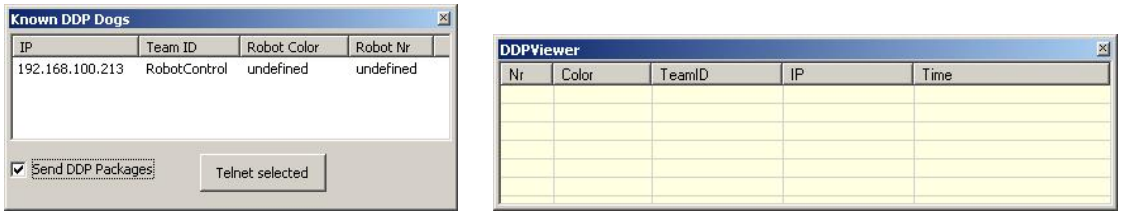
9.2.2.1 Framework

Das Framework hat die Aufgabe, sich um die Darstellung der Usercontrol (vergleiche Kapitel 9.2.2.2) in RobotControl 2 zu kümmern. UserControls sind dabei die Dialoge die in RobotControl 2 verwendet werden. Dabei stellt das Framework Funktionalität für das Anlegen und Verwalten von Dialogen, das Drag & Drop sowie die *MessageQueue* zur Verfügung. Das Anlegen der Manager und der Dialoge findet hier statt.

9.2.2.2 UserControls

UserControls sind die Anzeigeelemente in RobotControl 2. Sie können beliebige Inhalte darstellen und dienen zur Interaktion mit dem Benutzer. UserControls erben alle von einer Basisklasse[1], die sich um die Integration in das Framework kümmert. Außerdem stellt sie die Möglichkeit des Dockings zur Verfügung. Wir haben RobotControl um verschiedene UserControls erweitert.

[1] RobotControl.UserControls.RobotControlUserControl



(a) DDP-Dialog in RobotControl (b) DDP-Dialog in RobotControl 2

Abbildung 9.1: DDP Dialoge

9.2.2.3 Manager

In RobotControl 2 stellen die Manager das Verbindungsglied zwischen den Roboter Daten und den UserControls dar. Die Daten, die vom Roboter über das Netzwerk kommen, werden hier aufbereitet und stehen dann den UserControls zur Anzeige und Auswertung zur Verfügung. Ziel sollte es sein, UserControls beliebig hinzufügen und entfernen zu können, ohne dass evtl. Anpassungen an anderen UserControls notwendig ist. Daher verarbeiten die UserControls die Daten der Roboter nicht eigenständig, sondern verwenden die Manager.

9.2.3 Neue und erweiterte Dialoge

Im Rahmen der Umstellung von RobotControl auf RobotControl 2 wurden einige neue Dialoge implementiert.

9.2.3.1 Known DDP Dialog

Sowohl RobotControl als auch RobotControl 2 wurden um einen Dialog zur Anzeige von DDP Paketen (vergleiche [16], Seite 31) erweitert (siehe Abbildung 9.1). DDP Pakete sind Pakete die der Roboter über das WLAN sendet und die es ihm ermöglichen selbständig seine Mitspieler im WLAN zu finden. In beiden Dialogen werden die vom Roboter versendeten DDP-Pakete im Klartext angezeigt. Mittels eines Doppelklick auf einen Eintrag verbindet sich RobotControl zu dem Roboter und die Debug-Informationen werden übertragen.

9.2.3.2 AIBOVision nach RobotControl 2 Portierung

AiboVision ist ein Java-Programm von Walter Nistico zur Manipulation von Farbtabellen. Außer den bereits in RobotControl bzw. RobotControl 2 bereitgestellten Funktionen zur manuellen Bearbeitung von Farbtabellen, bietet AiboVision noch automatisierte Filter- und Optimierungsfunktionen. Um nicht mit unterschiedlichen Tools arbeiten zu müssen, wurden beliebte Teile von AiboVision in RobotControl 2 integriert und der entsprechende Java-Code nach C# konvertiert.

ColorTableTool Das *ColorTableTool* in RobotControl 2 wurde um die Möglichkeit, Farbtabellen zu generalisieren, erweitert. Initial werden die Farben anhand von wenigen Bildern grob klassifiziert. Dann werden automatisch die Ausdehnungen der Farben im Farbraum vergrößert. Dadurch wird eine höhere Toleranz gegenüber variierenden Lichtverhältnissen während eines Spiels erreicht. Es stehen verschieden starke Filterfunktionen zur Auswahl, um ein passendes Ergebnis zu erhalten.

ColorCorrectorCalibration Der bereits im GT-Framework vorhandene *ColorCorrector* wurde zur Verwendung in RobotControl 2 nach C# konvertiert. Die vom AIBO aufgenommenen Bilder werden zum Rand hin dunkler (Vignettierung). Um diesen Effekt auszugleichen, wird der *ColorCorrector* eingesetzt. Er benötigt jedoch Koeffizienten, um die Korrektur ausführen zu können. Um diese Koeffizienten berechnen zu können, wurde in RobotControl 2 der Dialog *ColorCorrectorCalibration* hinzugefügt.

Zunächst muss ein Log-File vom Roboter aufgezeichnet, im Log-Player abgespielt oder aus einer Datei geladen werden. Es muss einfarbige Kamerabilder des Roboters enthalten, wobei weiße, gelbe und blaue Bilder erforderlich sind. Dann kann die Qualität und der gewünschte Farbkanal für die Koeffizientenberechnung ausgewählt werden. Im rechten Teil des Dialogs wird das zuletzt (vom Roboter oder aus dem Log-Player) empfangene Bild angezeigt. Dieses kann zu Vorschauzwecken auch farbkorrigiert angezeigt werden, nachdem die Koeffizienten-Dateien geladen wurden.

9.3 CeilingCam

Die CeilingCam stellt ein zentrales Messwerkzeug für die Güte der auf dem Roboter verwendeten Algorithmen zur Verfügung. Neben dem CeilingCam-Server wurden auch in den DebugTools nötige Änderungen durchgeführt, um mit der CeilingCam arbeiten zu können.

9.3.1 Motivation

Durch die Verwendung der CeilingCam bestehen neue Möglichkeiten des Debuggings. So ist es nun möglich, SelfLocator oder BallLocator gegenüber einem Referenzsystem zu vergleichen. Dies wird durch verschiedene Tools und Module unterstützt.

9.3.2 Module auf dem Roboter

Für den Roboter wurden für die meisten Module Platzhalter programmiert, die auf den Daten der CeilingCam arbeiten und somit quasi Orakeldaten zur Verfügung stellen. So ist es möglich, einen BallLocator zu testen, ohne notwendigerweise einen SelfLocator zu besitzen. Dies bietet außerdem den Vorteil, dass sich die Messfehler

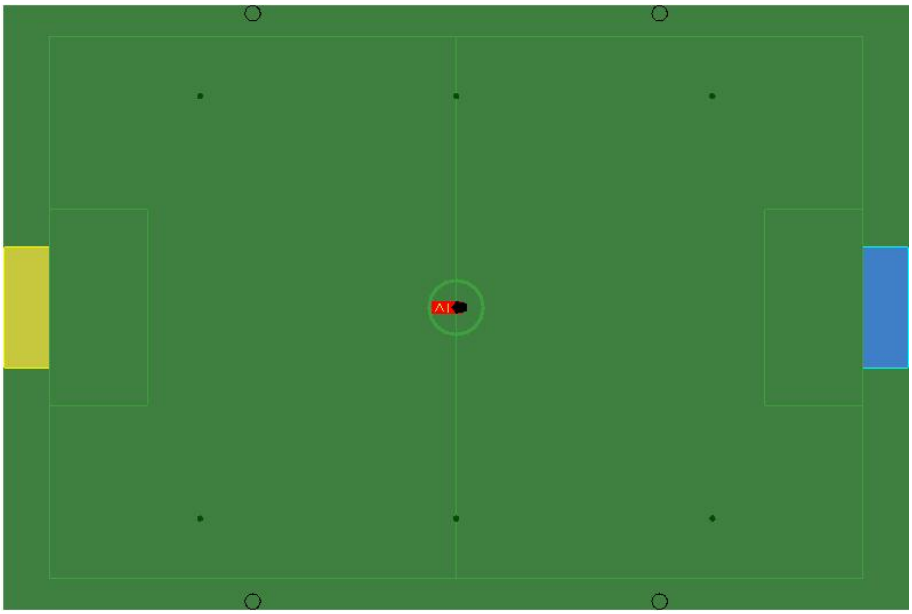


Abbildung 9.2: *FieldView* mit CeilingCam Daten in RobotControl

und Messungenauigkeiten, die zum Beispiel auf einer schlechten Lokalisierung beruhen, nicht mehr auftreten und sich somit die Fehler der einzelnen Module nicht unbedingt akkumulieren.

9.3.3 Anzeige des WorldState

Um die Daten der CeilingCam zu visualisieren, wurde in den DebugTools RobotControl und RobotControl 2 die Möglichkeit implementiert, den *WorldState*, der von der CeilingCam versendet wird, anzuzeigen. Hierfür wird der schon bisher vorhandene *FieldView* verwendet und die Daten der CeilingCam als Overlay hinzugefügt (Abbildung 9.2). Die Daten der CeilingCam werden in anderen Farben dargestellt als die normalen Debug-Informationen. Zusätzlich ist jeder Roboter der CeilingCam mit einer eindeutigen Rückennummer gekennzeichnet.

Neben dem *FieldView* wurde der GTCam-WorldStateViewer-Dialog (siehe Abbildung 9.3) implementiert, der die Daten der CeilingCam textuell darstellt. Zusätzlich können in ihm die GTCam-Daten aufgezeichnet und in verschiedene mit GNUPlot[2] verarbeitbare Datenformate exportiert werden. Dies wurde unter anderem bei der Auswertung des BallLocator (siehe Kapitel 4) verwendet.

[2] `http://www.gnuplot.info`

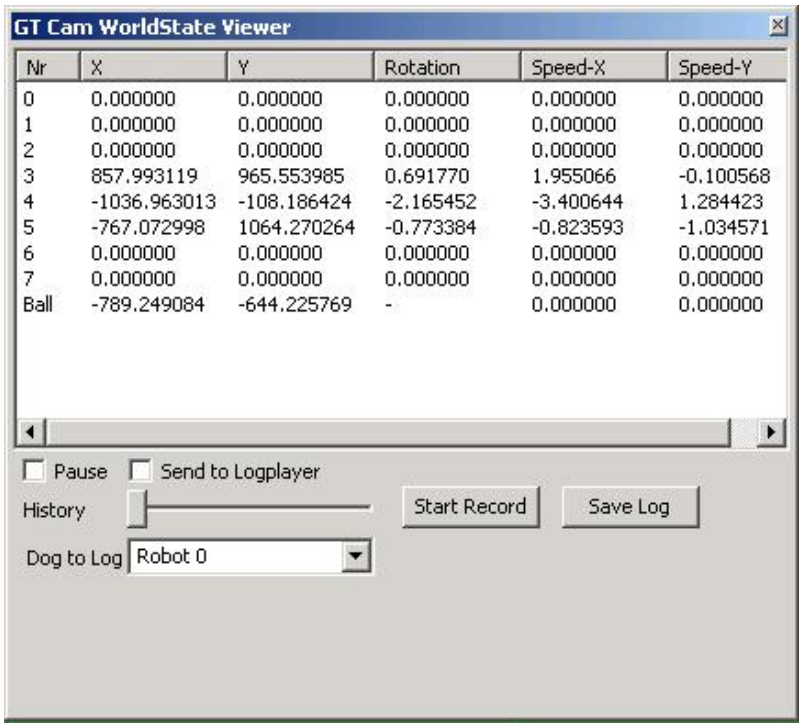


Abbildung 9.3: GTCam-WorldStateViewer-Dialog

10 Challenges

Neben dem Roboterfußball finden bei den Weltmeisterschaften auch die so genannten Challenges statt. Hierbei gilt es, Aufgaben zu lösen, die am Ziel, den Roboterfußball weiter an die Bedingungen eines richtigen Fußballspiels anzunähern, ausgerichtet sind. Für die Weltmeisterschaften 2005 in Osaka haben wir Lösungen für zwei Challenges vorbereitet.

10.1 Die (almost) SLAM[1] -Challenge

Bisher werden in der Liga der vierbeinigen fußballspielenden Roboter zur besseren Lokalisierung Hilfsmarkierungen mit bekanntem Standort und festgelegter Farbmarkierung verwendet. Zur Förderung der weiteren Forschung und in Hinblick auf das Fernziel, ein normales Fußballspiel auf einem normalen Feld bestreiten zu können, soll sich der AIBO in Zukunft eigenständig an dem vorhandenen Hintergrund orientieren. Die „(almost) SLAM-Challenge“ beinhaltet als zusätzliche Hilfestellung eine Initialisierungsphase, in der die normalen Landmarken auf dem Feld aufgestellt sind und sich der AIBO zusätzliche markante Punkte merken kann. Später muss sich der AIBO anhand der gemerkten natürlichen, also im Hintergrund gefundenen, markanten Punkte orientieren.

10.1.1 Problembeschreibung

Die Aufgabe in der „(almost) SLAM-Challenge“ ist es zunächst, in der Lernphase innerhalb einer Minute neue markante Punkte zu lernen, anhand derer der AIBO sich in der zweiten Phase orientieren kann. In dieser Lernphase sind die bekannten Landmarken und Tore zunächst noch sichtbar, damit der AIBO seine Position ermitteln kann. Um die Anwesenheit markanter fixer Punkte zu garantieren, geben die Regeln desweiteren vor, dass neue Landmarken mit bekannten Spezifikationen bezüglich Größe und erlaubten Farben aufgestellt werden (siehe [18]). In der zweiten Phase muss der AIBO innerhalb von zwei Minuten, von einem unbekannten Startpunkt aus, fünf Punkte auf dem Feld möglichst genau anlaufen. Insgesamt soll der AIBO sich also zusätzliche Orientierungspunkte suchen und merken können, um sich später anhand der gespeicherten Informationen über die Umgebung seine Position finden und auch beim Anlaufen der Zielpunkte nicht verlieren können.

[1] SLAM = simultanous location and mapping, gleichzeitige Lokalisierung und Kartierung

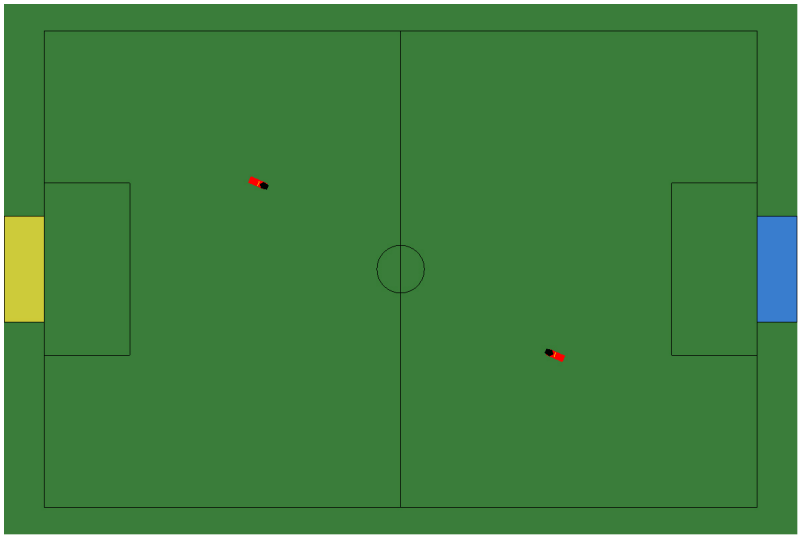


Abbildung 10.1: Ohne Tore kann der AIBO die beiden gezeigten Positionen zunächst nicht unterscheiden.

10.1.2 Problemanalyse

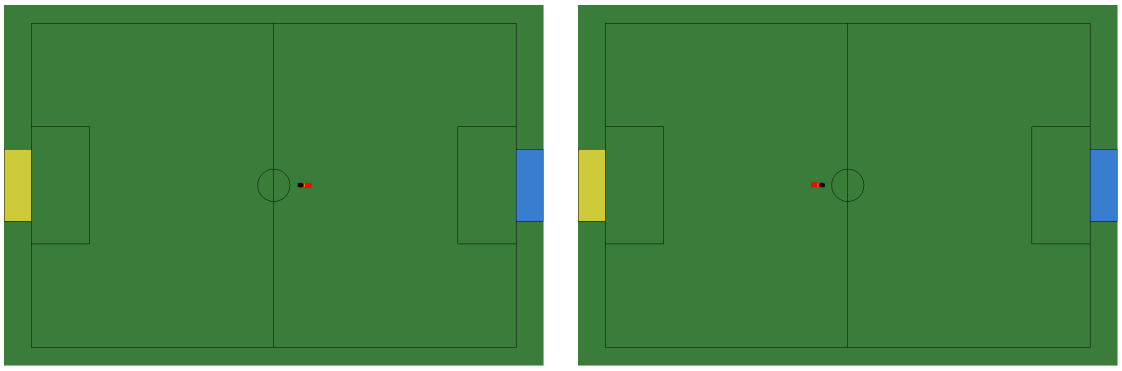
Die beschriebene Aufgabe lässt sich in drei Aufgabenteile zerlegen:

- Erkennung neuer Landmarken
- Kartierung neuer Landmarken
- Nutzung der neuen Landmarken

Da die in der zweiten Phase der „(almost) SLAM-Challenge“ neben den neuen Landmarken lediglich die Feldlinien verbleiben und diese durch ihre Symmetrie keine eindeutige Lokalisierung erlauben (Abbildung 10.1), ist eine Erkennung und Kartierung der neuen Landmarken nötig. Hierbei ist es sinnvoll die Vorgaben für Aussehen und Positionierung dieser zu berücksichtigen, um das Problem zu vereinfachen. Da nicht davon ausgegangen werden kann, dass die zusätzlichen Landmarken zuverlässige Informationen liefern, muss der SelfLocator dahingehend optimiert werden, in besonderem Maße die Feldlinien und den Mittelkreis zu nutzen. Desweiteren kann die Selbstlokalisierung stärker auf die Odometrie vertrauen, da Kollisionen nicht zu erwarten sind. Da der Mittelkreis auch ohne weitere Landmarken nur zwei symmetrische Positionen zulässt (Abbildung 6.9), ist es in seiner Nähe besonders einfach, die *RobotPose* auf Grund weniger zusätzlicher Informationen zu bestimmen.

10.1.3 Ansätze zur Lösung des Problems

Auf Grundlage der Vorüberlegungen musste eine Vorgehensweise entwickelt werden, um die Aufgabe möglichst robust zu lösen; es mussten Verfahren getestet werden, die die Bestimmung der *RobotPose* in der zweiten Phase ermöglichen und für den



(a) Der Roboter positioniert sich derart, dass er genau eine Feldhälfte im Blick hat...

(b) ...und daraufhin die andere Feldhälfte.

Abbildung 10.2: Positionierung am Mittelkreis

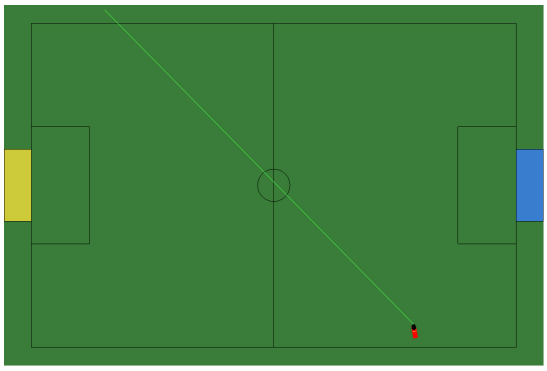
SelfLocator mussten Anpassungen an die Regeln der „(almost) SLAM-Challenge“ vorgenommen werden.

10.1.3.1 Vorgehen (Verhalten)

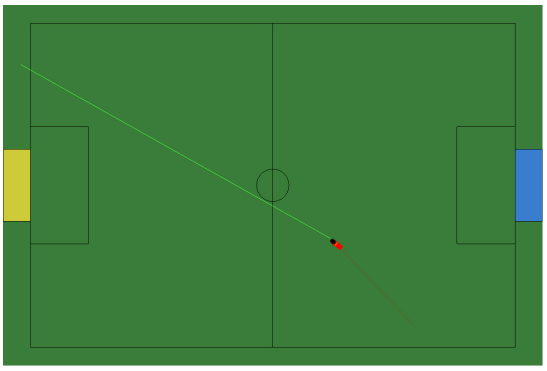
Nach den Vorüberlegungen haben wir uns für ein Vorgehen entschieden, das oben beschriebene Vereinfachungen so weit wie möglich nutzt und geringstmögliche Änderungen an den bestehenden Modulen erfordert. In der ersten Phase wird der Mittelkreis wie in Abbildung 10.2 gezeigt angelaufen. In beiden abgebildeten Positionen wird der Bereich vor dem Roboter, der durch Kopfbewegung sichtbar ist, nach neuen Landmarken untersucht, die in eine Karte eingetragen werden. In der zweiten Phase ist die Position des AIBO zunächst unbekannt und es soll eine Richtung ermittelt werden, die den Roboter mit hoher Wahrscheinlichkeit auf den Mittelkreis zugehen lässt. Ist dieser sichtbar, wird eine der beiden in Abbildung 10.2 gezeigen Positionen angelaufen, was mit gesehenem Mittelkreis sehr präzise möglich ist (Abbildung 6.9). Allerdings ist an dieser Stelle nicht klar, welche der beiden Positionen die Aktuelle ist. Um dies zu entscheiden, werden die von dieser Position aus erkannten neuen Landmarken mit der in der ersten Phase entstandenen Karte verglichen. Sobald die Rotation der *RobotPose* auf diese Weise entschieden wurde, sollen nur auf Basis der beobachteten Linien die fünf Zielpositionen angelaufen werden.

10.1.3.2 Richtungsbestimmung ohne Lokalisierung (Max-Green-Detection und das Feldende)

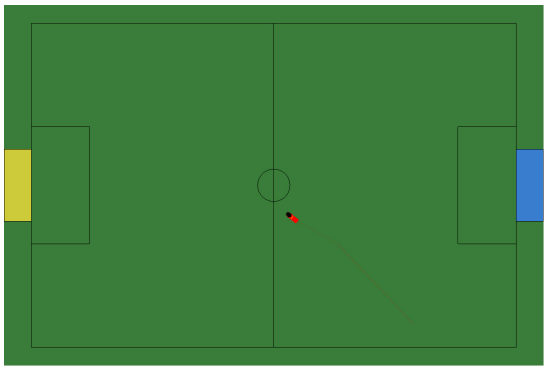
Generell weist die Richtung, in der vom Roboter aus der längste grüne Bereich zu erkennen ist, in etwa auf den Mittelkreis. Dies soll genutzt werden, um das in der zweiten Phase erforderliche Auffinden des Mittelkreises zu realisieren (Abbildung 10.3). Hierzu dreht sich der AIBO auf einem Punkt um 360° und speichert in jedem Bild den Abstand zum weit entferntesten Punkt, der gerade noch auf dem Feld liegt. Währenddessen sucht der AIBO auf senkrecht zum Horizont verlaufenden Scanlinien



(a) Der Roboter sucht die Richtung, in der er am weitesten Grün sieht und geht in etwa in die Richtung.



(b) Nach einer bestimmten Zeit, sucht der Roboter erneut die Richtung, in die sich das Feld am weitesten erstreckt und läuft hin.



(c) Wenn der Roboter nahe genug am Mittelkreis ist, kann er ihn erkennen.

Abbildung 10.3: Finden des Mittelkreises

längere, zusammenhängende grüne Pixelstreifen und projeziert den obersten Pixel aus der Bildebene auf das Spielfeld in lokalen Roboterkoordinaten ($z \stackrel{!}{=} 0$). Von allen projezierten Pixeln wird der kartesische Abstand vom Roboter (in lokalen Koordinaten $x = 0, y = 0, z \stackrel{!}{=} 0$) berechnet und der maximale Abstand bestimmt. Dieser maximale Abstand wird für jedes Frame mit der zugehörigen Richtung gespeichert und nach Vollendung der Drehung wird das Maximum und dessen Richtung über alle Frames bestimmt. Verbessert werden konnte diese Information vor Ort bei der WM in Osaka, da das Feld durch einen hellgrauen Teppich begrenzt wurde, der in einem Abstand von bis zu ca. 2 m analog zur Liniendetektion in Kapitel 7.2.3 erkannt werden konnte. Die so ermittelte Richtung wurde genutzt, um den Roboter an den Mittelkreis anzunähern und war durch die Erkennung der Feldbegrenzung besonders robust gegen das Verlassen des Feldes.

10.1.3.3 Erkennung neuer Landmarken

Da für einen Teil der Landmarken sichergestellt ist, dass sie einen pink markierten Bereich enthalten, wird nach solchen Flächen gesucht und von diesen aus überprüft, ob sie von anderen bekannten Farben umgeben sind. Die dabei mögliche Klassifizierung wurde in den „(almost) SLAM-Challenge“ allerdings nicht genutzt.

10.1.3.4 Kartierung neuer Landmarken

Bei der Kartierung der neuen Landmarken wurde auf eine Verfolgung erkannter Landmarken verzichtet. Stattdessen wurde der Bereich, in dem neue Landmarken nach den Regeln zu erwarten waren, als Wahrscheinlichkeitsfeld modelliert, wobei bei jeder Erkennung einer Landmarke auf einer Linie, die in Richtung der Landmarke durch das Wahrscheinlichkeitsfeld gezogen wird, der Wahrscheinlichkeitswert inkrementiert wird (Abbildung 10.4). Obwohl es möglich ist, so unterschiedlich charakterisierte Landmarken getrennt zu kartieren, wurde hierauf aus Zeitmangel verzichtet, und gehofft, dass die Landmarken mit einem pinkfarbenen Bereich hinreichend unsymmetrisch am Feld verteilt würden. Beobachtet man nun das Feld in alle Richtungen gleichmäßig, so wie es im beschriebenen Verhalten der Fall ist, entstehen entlang der Richtungen der tatsächlichen Landmarken deutliche Maxima, während zufällige Fehlerkennungen nur in Form eines Rauschens kleine Werte im Wahrscheinlichkeitsfeld hinterlassen. Mit dem Wissen, dass die zusätzlichen Landmarken nur in einem schmalen Bereich aufgestellt werden, können so die Positionen dieser Landmarken angenähert werden. Allerdings ist das beschriebene Verfahren auch geeignet, Landmarkenpositionen unabhängig von ihrer Distanz zum Feld zu bestimmen, wenn man die Position des Roboters während der Beobachtungsphase ändert (Abbildung 10.5). Problematisch bei der Kartierung war noch die Behandlung der bekannten Landmarken und Tore. Vor allem die Tore können zusammen mit einer neuen pinken Landmarke leicht als bekannte Landmarke identifiziert werden. Daher wurden alle Bereiche im Bild, die als bekannte Landmarke identifiziert wurden, mit Ausnahme solcher in Tornähe, von der Betrachtung ausgeschlossen. Da die bekannten Landmarken allerdings seltener als solche identifiziert wurden als neue, wurde zusätzlich der Winkelbereich, der zu diesen gehört, ausgeschlossen.

10.1.3.5 Schätzung der Roboterrichtung mittels WLAN-Feldstärkenmessungen

Neben der Auswertung neuer Landmarken zur Wahl der richtigen Feldhälfte wurde erprobt, die WLAN Feldstärke im Verlauf über die Zeit zu messen und auszuwerten. Da die eingebaute Antenne eine gewisse Richtwirkung hat, kann man theoretisch das Ansteigen und Abfallen der Empfangsstärke beim hin- bzw. wegdrehen, zu bzw. von dem Access-Point, die Richtung, in der der Access-Point steht, anpeilen. Bei bekannter Position, lässt sich idealer Weise dann so mit einem Access-Point ganz einfach die Orientierung bestimmen, mit Hilfe eines zweiten Access-Point sogar schon die Position und die Orientierung. In der Praxis tritt allerdings durch Mehrfachempfang

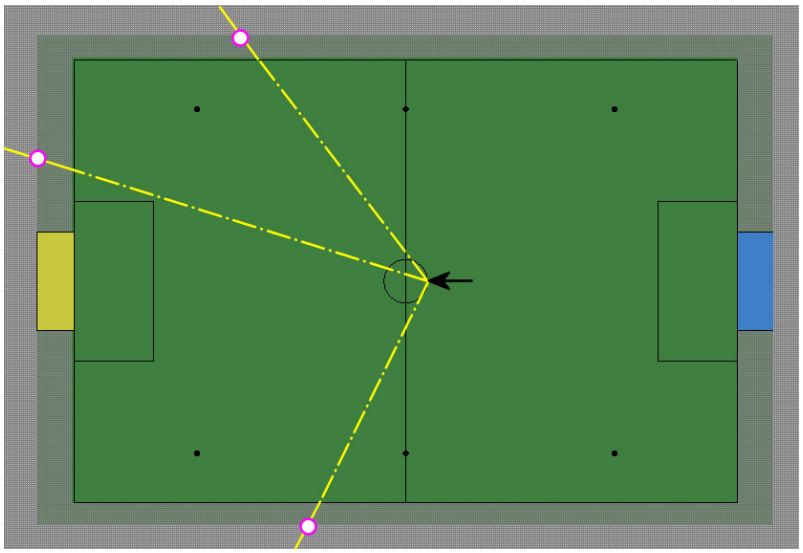


Abbildung 10.4: Der Wahrscheinlichkeitswert wird entlang von Linien inkrementiert

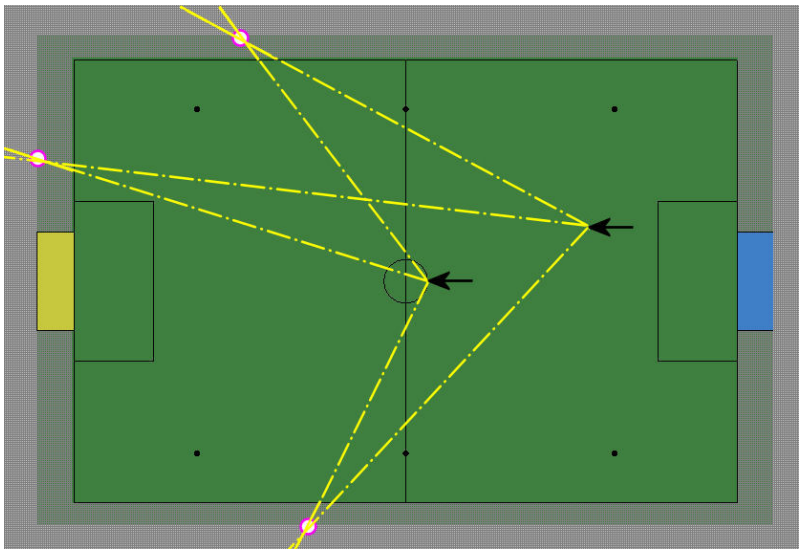


Abbildung 10.5: Durch Bewegen des Roboters können die Landmarkenpositionen genau bestimmt werden

und Verdeckungen sowohl ein Rauschen, als auch Fehler auf. Um dem Rechnung zu tragen, wird die Messung über einen kurzen Zeitraum während einer oder mehrerer vollständiger Drehungen aufgenommen. Damit wird vermieden, dass lokale Feldstärkenunterschiede Einfluss auf die Messung nehmen und durch möglichst kurze Messungen werden auch die temporären Schwankungen minimiert. Trotzdem ergab sich leider keine ausreichende Genauigkeit.

10.1.3.6 Auswertung neuer Informationen

Sobald der Roboter in der zweiten Phase eine Position vor dem Mittelkreis erreicht hat, soll anhand der entsprechend Kapitel 10.1.3.4 kartierten Landmarken und ggf. der WLAN-Messung (Kapitel 10.1.3.5) entschieden werden, auf welcher Seite des Mittelkreises sich der Roboter befindet. Hierfür ist die Karte der Landmarken in je einen Teil für eine Feldhälfte geteilt. Für die Beobachtungen in der zweiten Phase werden dann die Winkeldifferenzen zu den nächstmöglichen Landmarken für beide Hälften getrennt summiert, so dass nach einigen Beobachtungen für jede Hälfte eine Bewertung vorliegt. Für die Richtung des Roboters wird schließlich die Hälfte mit der niedrigeren Winkeldifferenzsumme zugrundegelegt. Sollen weitere Kriterien zur Entscheidung der Rotation genutzt werden, muss eine Bewertungsfunktion gewählt werden, die diese Kriterien vergleichbar macht. Das beschriebene Vorgehen hat den Vorteil, dass, sollte kein Kriterium eine Unsymmetrie aufweisen, die initiale Position des Roboters noch mit 50% Wahrscheinlichkeit richtig bestimmt wird.

10.1.3.7 Anpassung des SelfLocators

Um beiden Phasen der Challenge gerecht zu werden, wurden zwei Sätze von Parametern optimiert. Einer, der eine möglichst genaue Selbstlokalisierung mit den in der ersten Phase verfügbaren Informationen erlaubt, und einer, mit dem eine anfänglich bekannte Position mithilfe der Linien und der Odometrie weiterverfolgt werden kann. Da dem Mittelkreis in der zweiten Phase eine besonders hohe Bedeutung zukommt, werden 3% der Partikel durch *Templates* ersetzt, deren Position sich aus dem Mittelkreis ableitet. Um die Genauigkeit des SelfLocators in der 2. Phase weiter zu verbessern, wurde die Modellierung des Spielfeldes de facto auf eine Hälfte eingeschränkt (dies ist auf Grund der Symmetrie auch hinreichend), indem Partikel auf der zur aktuellen *RobotPose* symmetrischen Position auf die andere Feldhälfte übertragen werden (Abbildung 10.6 und Gleichung (10.1)).

$$(x', y', \theta') = (-x, -y, \theta + \pi) \tag{10.1}$$

10.1.4 Beschreibung der gewählten Lösung und Diskussion

Auf der Weltmeisterschaft in Osaka wurden alle Lösungsansätze mit Ausnahme der WLAN-Feldstärkenmessungen benutzt, da diese auf Grund der niedrigen Genauigkeit keinen Informationszuwachs brachte. Im Wettkampf konnten 4 von 5 Punkten

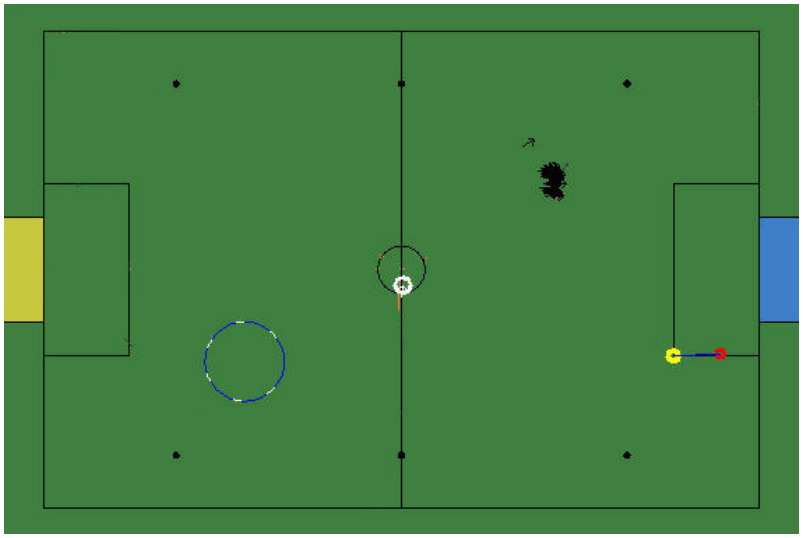


Abbildung 10.6: Alle Partikel, die sich in einem gewissen Radius um die symmetrische Position befinden, werden auf die andere Feldhälfte übertragen.

auf dem Feld mit hoher Genauigkeit angelaufen werden, was uns mit Abstand den Sieg sicherte. Wie bereits erwähnt, lässt sich der hier implementierte Ansatz für die Challenge durchaus erweitern. So lassen sich mit kleinen Änderungen unterschiedliche neue Landmarken kartieren und in der zweiten Phase ist das Finden des Mittelkreises nicht unbedingt vonnöten, wenn man die kartierten Landmarken ständig zur Lokalisierung nutzt.

10.2 Die Variable Lighting Challenge

10.2.1 Motivation

Im aktuellen Stand der Sony-4-legged-League und auch der meisten anderen Ligen benuzten alle Teams für die Bildverarbeitung sogenannte Farbtabellen. Das sind generell Nachschlagewerke, mit deren Hilfe die einzelnen Pixel eines Bildes einer Farbklasse zugeordnet werden. Dabei besitzt jeder Pixel drei Werte, je einen für jeden der drei Farbkanäle. Eine solche Farbtabelle stellt also eine Aufteilung des dreidimensionalen Farbraumes in verschiedene Farbregionen dar. Je nach Vorliebe und Erfahrung sind diese unterschiedlich kompliziert aufgebaut: Von einfachen oberen und unteren Grenzwerten je Klasse auf allen Farbkanälen, was zu würfelförmigen Farbregionen führt, bis hin zu der allgemeinsten Form, bei der einfach für jedes Wertetriplet die zugehörige Klasse gespeichert wird.

Wie in Kapitel 7 beschrieben, stellt die Bildverarbeitung die nahezu einzige Informationsquelle über die umgebende Welt dar. Da die Qualität der beschriebenen Farbzuordnungen grundlegend für die Bildverarbeitung und somit auch für das komplette Verhalten des Roboters ist, stellen diese auch ein wichtiges Kriterium beim

Abschneiden in den Wettbewerben dar. In den meisten Teams ist es üblich, vor jedem Spiel diese Farbtabellen zu überarbeiten oder gar neu zu erstellen, teilweise sogar individuell für jeden Roboter.

Diese extreme Spezialisierung widerspricht allerdings dem letztlich gesetzten Ziel eines möglichst flexibel einsetzbaren Roboters, der ohne aufwendige Kalibrierungsvorgänge einfach auf einem beliebigen Feld antreten kann.

Um diese starke Abhängigkeit von Beleuchtungsveränderungen abzuschütteln, wurde die Variable Lighting Challenge ausgeschrieben.

10.2.2 Problembeschreibung

Die Variable Lighting Challenge ist dafür vorgesehen, Teams dazu anzuhalten, die Robustheit ihrer Bildverarbeitung gegen Beleuchtungsveränderungen zu erhöhen. Sie basiert auf einer Strafschussaufgabe. Das antretende Team platziert einen einzelnen blauen Roboter auf dem Feld, welcher dann innerhalb von drei Minuten den Ball so oft wie möglich in das gelbe Tor schießen muss. Das Team mit den meisten Toren gewinnt, gleiche Anzahl gilt als Unentschieden. Bei den GermanOpen wurde zusätzlich als weiteres Kriterium die benötigte Gesamtzeit verwendet. Bei gleicher Anzahl an Toren gewann das Team, dessen Roboter das letzte Tor früher schoss.

Zusätzlich zu dem blauen Roboter werden zwei gegnerische rote Roboter im Torraum bzw. knapp außerhalb des Strafraums platziert. Die exakte Position wird erst zu Beginn vom Schiedsrichter festgelegt. Die roten Roboter sind pausiert und dürfen sich währenddessen nicht bewegen.

Zu Beginn und nach jedem Tor oder nach jedem Schuss ins Aus wird der Ball wieder am Mittelpunkt platziert.

Die größte Herausforderung ist nun, dass sich während dieser drei Minuten die Beleuchtung ständig verändert, wobei auch zusätzliche Lichtquellen unvorhersehbarer Farbwärme eingebracht werden können.

Vor Beginn wird von dem Schiedsrichter ein Ablaufplan der Beleuchtungsveränderungen aufgestellt, der sowohl Zeiträume konstanter Beleuchtung enthalten wird, als auch solche langsamer oder auch rapider Veränderungen. Hierbei werden allerdings die Lichtquellen nicht gleichmäßig verteilt, und somit die von ihnen hervorgerufenen Veränderungen ebensowenig gleichförmig sein.

10.2.3 Ansätze zur Lösung des Problems

Grundlegend ist die Aufgabenstellung der Variable Lighting Challenge, die von der Kamera des AIBOs aufgenommenen Bilder bei wechselnden Lichtverhältnissen zu jedem Zeitpunkt möglichst korrekt zu analysieren. Das Problem besteht darin, dass sich die Farbklassen bei unterschiedlichen Beleuchtungsbedingungen innerhalb des in Kapitel 10.2.1 beschriebenen Farbraumes in nicht-linearer Weise verschieben.

Besondere Schwierigkeit macht das Auseinanderhalten von gelb, orange und rot, somit also die Unterscheidung zwischen gegnerischem Trikot, Ball und dem gelben Tor. Unter einem hellen Scheinwerfer kann bei der schlechten Qualität der Kamera

des AIBOs ein rotes Trikot orange aussehen, ebenso wie ein im Schatten liegendes Tor. Beides ist für das Spielverhalten verhehrend, da der Roboter in beiden Fällen versuchen würde, das statische Objekt wie einen Ball zu schießen. Neben der Aussichtslosigkeit dieses Versuches kommt auch eine besonders bei den üblichen mit dem Kopf ausgeführten Schüssen nicht zu unterschätzende Beschädigungsgefahr des strukturell schwachen Halsgelenkes des Roboters hinzu.

Diese Aufgabenstellung der Variable Lighting Challenge wurde in der Vergangenheit auf verschiedene Arten mit zum Teil sehr komplexen Ansätzen und viel Arbeitsaufwand angegangen, wobei zum Teil ein ImageProcessor auf Basis dynamischer Farbklassifizierung komplett neu implementiert wurde. Im Folgenden werden nun zwei einfachere Herangehensweisen geschildert.

10.2.3.1 Übergeneralisierte Farbtabelle

Bei den Microsoft™ Hellhounds werden seit einiger Zeit die Farbtabellen nicht nur einfach per Hand erstellt, sondern anschließend noch automatisiert nachbearbeitet [15]. Diese allgemein als „Generalisierung“ bezeichnete Nachbearbeitung ist ein auf Simulation von Strahlungsprozessen basierendes Filterverfahren, das bei der Farbklassifikation sowohl falsche Zuordnungen minimiert, als auch gegebenenfalls den einer Klasse zugeordneten Teilraum nach vorgegebenen Heuristiken expandiert, so dass eventuell mehr korrekte Zuordnungen entstehen.

Im Falle eines normalen Spieles werden die Parameter dieser Generalisierungsheuristik so gewählt, dass nur diese stabilisierende Wirkung auftritt. Allerdings können diese Parameter auch so eingestellt werden, dass sich die Klassifikationsteilräume so ausdehnen, dass sie letztlich Partitionen des gesamten Farbraumes darstellen, was allgemein als „Übergeneralisierung“ bezeichnet wird.

In diesem Falle ist die Farbzuordnung deutlich stabiler gegen leichte Verschiebungen im Farbraum, allerdings ist die Fehlklassifikationsrate auch erheblich höher.

10.2.3.2 Dynamische Auswahl verschiedener Farbtabellen

Die im Folgenden beschriebene Herangehensweise stellt über die eigentliche Farbklassifizierung erst einmal eine Klassifizierung der gegebenen Beleuchtungssituation.

Hierbei wird zuerst versucht, auf Grund einiger in den zuletzt verarbeiteten Bildern gesammelter Merkmale, die optimale Farbtabelle auszuwählen und diese dann für die Verarbeitung des nächsten anstehenden Bildes zu benutzen.

Mit einem einfachen Kriterium, wie dem über die letzten fünf Bilder geglätteten Durchschnittswert eines Farbkanals, der ohne große Mehrkosten während des normalen Scanvorganges gewonnen wird, können so robust und reaktionsschnell Beleuchtungssituationen innerhalb einer breiten Helligkeitsspanne unterscheiden und diesen jeweils die passende Farbtabelle zuweisen werden.

Eine auch unter normalen Bedingungen auftretende Klassifizierungsschwierigkeit ist es, den Ball sowohl im hellen Scheinwerferlicht zu erkennen, als auch dann, wenn der Roboter sehr nahe steht, so dass der eigene Schatten auf dem Ball liegt. In beiden Fällen muss den Ballpixeln noch genügend Orange zugewiesen werden, um eine

Erkennung als solchen zuzulassen, ohne jedoch in Teilen vom gelben Tor oder den roten Trikots Orange zu sehen. Dieses Problem wird mit der beschriebenen Methode auf natürliche Weise aufgefangen, wie in Abbildung 10.7 dargestellt. Sobald nämlich der Schatten einen Teil des Bildes ausmacht, wird automatisch zu einer dunkleren Farbtabelle umgeschaltet.

Werden diese Farbtabellen zusätzlich noch durch die in [15] beschriebene Methode generalisiert, so sind die Farbzuordnungen auch in den Übergangsbereichen zufriedenstellend.

Diese Methode benötigt somit zwar mehrere Farbtabellen, allerdings dann auch keine Neueinstellung bei großen Helligkeitsänderungen.

10.2.4 Beschreibung der gewählten Lösung

Beide Lösungsansätze beschreiben Möglichkeiten, die gestellte Aufgabe anzugehen, ohne den kompletten ImageProcessor nach einem völlig anderen Konzept neu zu implementieren. Dies stellt ein wichtiges Kriterium dar, da für eine Lösung der Variable Lighting Challenge weder viel Zeit noch viele Ressourcen investiert werden sollten.

Der in Kapitel 10.2.3.1 beschriebene Ansatz wurde schon bei der Variable Lighting Challenge im letzten Jahr eingesetzt, allerdings nur mit mäßigem Erfolg. Während kleinere Helligkeitsschwankungen gut ausgeglichen werden, können größere Verschiebungen im Farbraum nur sehr begrenzt aufgefangen werden.

Der Ansatz aus Kapitel 10.2.3.2 hingegen lieferte unter Laborbedingungen durchweg gute Ergebnisse und konnte bei einer großen Helligkeitsspanne eingesetzt werden.

Allerdings orientierten sich diese Tests stark an den in den letzten Jahren abgehaltenen Durchführungen der Variable Lighting Challenge.

10.2.5 Abschneiden im Wettbewerb

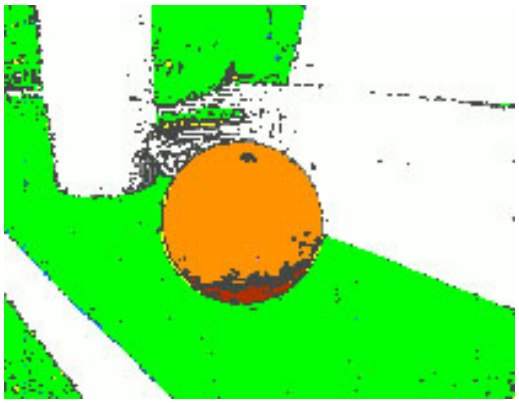
Die Bildverarbeitung mit der beschriebenen Lösung lieferte auf den GermanOpen in Paderborn gute Ergebnisse. Der Roboter verlor nie den Ball aus dem Blick und fand ihn auch nach erzielten Toren selbstständig an der Mittellinie wieder. Weiterhin wurde zu keinem Zeitpunkt der Ball an einer Stelle gesehen, wo er nicht war. Ballerkennung und Lokalisierung schienen gute Ergebnisse zu liefern.

Leider war das Verhalten mehr auf Genauigkeit und Sicherheit, als auf Geschwindigkeit ausgerichtet. So wurde der Ball oft zwar in Richtung Tor gedribbelt, allerdings auch direkt gegen den Verteidiger. Da dieser Fall ohne funktionierende Robotererkennung nicht sinnvoll abgefangen werden kann, kosteten solche Missgeschicke wertvolle Zeit.

Letztlich erlangten wir mit gleicher Anzahl Tore in etwas mehr Zeit den zweiten Platz hinter den Hamburg DogBots[2].

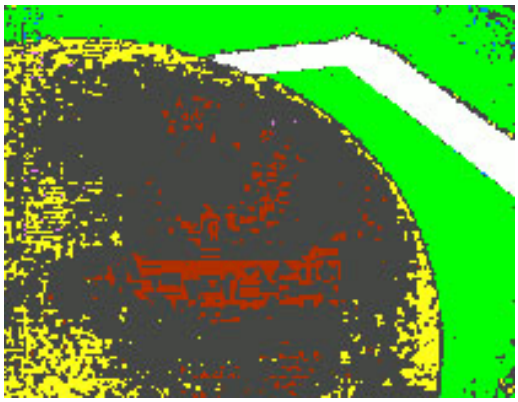
[2] `http://www.informatik.uni-hamburg.de/robocup/`

(a) Ball unter normalen Lichtbedingungen

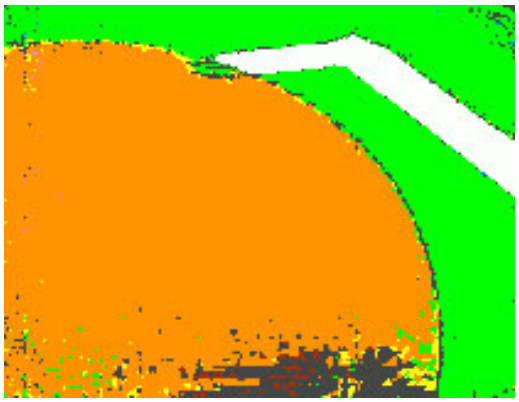


(b) Farbsegmentiertes Bild

(c) Ball im Schatten vom Roboter



(d) Segmentiert mit der selben Farbtabelle wie in Abbildung 10.7(b)



(e) Segmentiert mit Farbtabelle für eine dunklere Beleuchtungssituation

Abbildung 10.7: Verschiedene Situationen mit unterschiedlich beleuchtetem Ball

Bis zu den Weltmeisterschaften in Osaka wurde konzeptionell nicht mehr viel an der Bildverarbeitung geändert, das Verhalten allerdings komplett neu geschrieben. Da die Prioritäten in der Woche vor und während den Meisterschaften anders gelegt wurden, wurde in dieser Zeit an der Variable Lighting Challenge weder entwickelt noch getestet.

Durch Pannen in der Abgabehektik wurde letztlich ein Speicherstick mit falscher Kamerakonfiguration abgegeben, wodurch der Roboter natürlich keinerlei sinnvolle Erkennung seiner Umgebung zustande brachte.

Vermutlich wäre auch bei korrekten Einstellungen ein Erfolg nicht sicher gewesen, da sich der Wettbewerbsaufbau aus organisatorischen Gründen grundsätzlich von dem der Vorjahre unterschied. Um die Vorbereitungen auf dem nahegelegenen, für das Endspiel vorgesehene, Feld nicht zu behindern, wurde fast keine Abdunklung vorgenommen. Stattdessen wurde durch zwei gelbliche Strahler ein kleiner, scharf abgegrenzter Bereich auf dem Feld geschaffen, in dem sich die wahrgenommenen Farben deutlich ins Gelbe hinein verschoben.

Da ein Testlauf außer Konkurrenz zu einem späteren Zeitpunkt nicht möglich war, wurde der Ansatz nicht mehr unter Wettkampfbedingungen getestet.

11 Ergebnisse und Wettbewerbe

Wir, die Microsoft™ Hellhounds, nahmen an verschiedenen Wettbewerben im Roboter-Fußball teil. Das Ziel dabei ist, die Ergebnisse unserer eigenen Arbeit, mit den Ideen der anderen Teams zu vergleichen und Erfahrungen auszutauschen. Selbstverständlich kommt auch der Spaß am Spiel nicht zu kurz und es wird im (menschlichen) Team heftig mitgefiebert.

11.1 RoboGames, San Francisco

Die RoboGames waren der erste Wettbewerb, bei dem nach den neuen Regeln [19] gespielt wurde. Deshalb waren im Vorfeld Anpassungen auch am Verhalten notwendig. Ein Ziel war der Test aller neuen Module, die seit dem letzten Wettbewerb erstellt wurden. Bei einem ersten Testspiel gegen das Austin Villa Robot Soccer Team fiel auf, dass es Probleme mit dem neu entwickelten Lauf gab. Die Beschaffenheit des Teppichs war anders als die unseres Exemplars, weshalb die Roboter ins Rutschen gerieten. Trotz dieser Probleme ging das Spiel unentschieden 2:2 zu Ende.

Das erste „richtige“ Spiel gegen SPQR haben wir überlegen gewonnen. Nachdem es zur Halbzeitpause schon 3:0 stand, wurde der Vorsprung in der zweiten Halbzeit auf 5:0 ausgebaut. Das zweite Spiel konnte Austin Villa gegen SPQR 4:0 gewinnen. Im Finale hielten unsere Roboter den Ball kontrolliert in der gegnerischen Hälfte und erreichten einen Halbzeitstand von 2:0. In der zweiten Halbzeit konnte Austin Villa einmal den Ball in unsere Hälfte bringen und unser Torwart musste sein Können zeigen. Trotzdem dominierten wir weiterhin das Spiel und konnten noch 3 weitere Tore erzielen, so dass es am Ende 5:0 für uns stand.

Die RoboGames gewannen wir ohne Gegentore!

11.2 GermanOpen, Paderborn

Zu den GermanOpen konnten endlich alle „Trainer“ der Microsoft™ Hellhounds mitkommen, da Paderborn nicht so weit weg ist. Eine kleine Gruppe fuhr schon einen Tag eher, als die restlichen Leute, um das Feld in der Mitte des Auditoriums aufzubauen und genügend Tische und Stühle für alle in der Team-Area zu organisieren. Am nächsten Tag trafen dann morgens alle Team-Mitglieder im Heinz-Nixdorf-Forum zusammen. Sofort wurde noch an den letzten Verbesserungen gearbeitet und ein kleines Team organisiert, das für das leibliche Wohl sorgte. So gut gerüstet, konnten die Spiele beginnen.

Unser erstes Spiel fand gegen den (noch) amtierenden Deutschen Meister, das Aibo Team Humboldt (ATH) statt. Der Spielverlauf, wie auch das Ergebnis (2:2), waren ausgeglichen. Wir gingen zwar mit 1:0 in Führung, aber ATH konnte noch vor der Halbzeitpause ausgleichen. In der zweiten Halbzeit dann umgekehrt: ATH ging mit 2:1 in Führung und wir glichen aus.

Weiter ging es mit dem Spiel gegen SPQR Sicilia. Die Roboter von SPQR sind zu Spielbeginn zwar einmal in unseren Strafraum vorgedrungen, konnten aber kein Tor erzielen. Den Rest des Spiels haben unsere Roboter klar dominiert und 4 Tore geschossen. Zum Endstand von 5:0 verhalf uns dann noch ein AIBO von SPQR, indem er ein Eigentor schoss. Durch das hervorragende Torverhältnis wurden wir Gruppenbester.

Im Viertelfinale traten wir gegen Les 3 Mousquetaires an. Unsere Roboter konnten den Ball fast ausschließlich in der gegnerischen Spielfeldhälfte halten und nebenbei noch 5 Tore schießen. Obwohl sich in der zweiten Halbzeit die Lichtverhältnisse änderten, weil zwei Strahler ausfielen, verloren unsere Roboter nicht die Orientierung. Im Gegenteil, sie schossen noch 4 weitere Tore. Wir haben also 9:0 gewonnen und sind damit ins Halbfinale eingezogen.

Unser Gegner im Halbfinale waren die Bremen Byters. Wir gingen, nach Torchancen auf beiden Seiten, schnell in Führung. Beim zweiten Tor half uns der Bremer Torwart, indem er den Ball, den wir nur bis zur Torlinie befördert hatten, rückwärts ins Tor schob. Bald darauf konnten die Bremer eine unübersichtliche Situation für sich ausnutzen und ein Tor schießen. Ein sofort gestarteter Gegenangriff wurde wieder vom Bremer Torwart unterstützt, der diesmal mit dem Kopf ein Eigentor erzielte. Am Halbzeitstand von 3:1 änderte sich in der zweiten Halbzeit nichts mehr. Einmal wurde unser Goalie von einem Bremer Angreifer bedrängt. Der Schiedsrichter entschied auf „Goalie-pushing“[1], der Hilfsschiedsrichter entfernte jedoch fälschlicherweise unseren Goalie aus dem Tor. Unsere Proteste wurden aber schnell berücksichtigt und der Angreifer bestraft. Da der pushende Bremer keinen Ball hatte blieb die Aktion ansonsten zum Glück folgenlos. Später rollte ein orangefarbener Golfball, mit dem auf dem Nachbarfeld gespielt wurde und dort ins Aus ging, über unser Feld. Kurzum: „Finale, wir kommen!“

Unseren Finalgegner kannten wir schon aus dem ersten Spiel, es war ATH. Berlin ging zwar in Führung, aber wir konnten nicht nur ausgleichen, sondern sogar ein 2:1 mit in die Halbzeitpause nehmen. Die zweite Halbzeit war ebenso spannend wie die erste und es fielen auf beiden Seiten weitere Tore: genau wie in der ersten Halbzeit schossen die Berliner eines und wir zwei, so dass wir mit 4:2 aus dem Finale gingen.

Damit haben wir die GermanOpen 2005 gewonnen und sind Deutscher Meister. Eigentlich sogar Europa-Meister, da auch einige ausländische Teams an den GermanOpen teilnahmen und keine gesonderte EM stattfand. Unser erstes hochgestecktes Ziel ist damit erreicht! Und weiter geht's!

[1] ein angreifender Roboter darf den (gegnerischen) Torwart höchstens drei Sekunden lang berühren/schieben, sonst wird er wegen Goalie-pushing bestraft

11.3 US-Open, Atlanta

Bei den US-Open durften wir offiziell nicht teilnehmen. Aber wir durften vier Exhibition-Matches gegen die amerikanischen Teams spielen. Im so genannten Finale traten wir dann gegen den amtierenden amerikanischen Meister an. Die Carnegie Mellon University (CMU) konnte sich gegen die University of Pennsylvania (UPenn) durchsetzen, so dass wir gegen CMU antraten.

Gleich bei unserem Anstoß ging der Ball ins Tor. (Die Regeln besagen, dass der Ball vom Team, das Anstoß hat, außerhalb des Mittelkreises berührt werden muss, damit ein Tor gewertet wird. Leider war das hier nicht der Fall.) Die erste Halbzeit konnten wir dann mit 2:0 beenden. In der zweiten Halbzeit war der Gegner etwas stärker, aber es fielen keine weiteren Tore. Damit tragen wir den Titel „Interkontinentalmeister“.

11.4 Weltmeisterschaft, Osaka

Bei der Weltmeisterschaft waren wir ein Teil des GermanTeam. Zum Zeitpunkt der Weltmeisterschaft bestand es aus dem Aibo Team Humboldt, den Bremen Byters, den Darmstadt Dribbling Dackels und den Microsoft™ Hellhounds. Nach der WM haben wir das GermanTeam verlassen und treten bei der nächsten WM als eigenständiges Team auf.

Die Vorrunden- und Qualifizierungs-Spiele fanden gegen ARAIBO (Japan) und MiPal/Griffith (Australien) statt. Beide Gegner waren nicht wirklich gefährlich; gegen ARAIBO spielten wir 4:0, gegen MiPal/Griffith 5:1.

Wright Eagle aus China war schon bedeutend stärker. Schließlich setzt deren Software auf unserem vorjährigen Weltmeister-Code auf. Das Ergebnis war am Ende ein 1:1 Gleichstand.

Dann ging es um den Einzug ins Viertelfinale. Beide Gegner benutzen ebenfalls das Framework vom GermanTeam. Zuerst spielten wir gegen das Dutch Aibo Team. Dabei konnten wir schon in der ersten Halbzeit vier Tore erzielen und in der zweiten Halbzeit fiel noch ein weiteres für uns. Als nächstes spielten wir gegen die Hamburg Dog Bots und mussten außer einem Sieg auch eine gute Tordifferenz erreichen. Das gelang uns mit einem 3:0. Weil nun noch Wright Eagle gegen das Dutch Aibo Team 3:3 spielte, wurden wir Gruppenerster.

Unser Viertelfinalspiel fand gegen Jolly Pochie (Japan) statt. Wir konnten problemlos ein 5:0 erreichen und zogen in das Halbfinale ein.

Endlich trafen wir auf unseren Lieblingsgegner, die Carnegie Mellon University (CMU) aus den USA. Weil unsere Roboter häufiger am Ball waren und den Robotern von CMU häufig sogar den Ball entreißen konnten, schossen wir schon in der ersten Halbzeit drei Tore. Diese wurden in der zweiten Halbzeit durch zwei Weitere ergänzt.

Das Finale war eines der spannendsten Spiele. Die NuBots (Australien) konnten zwischenzeitlich sogar in Führung gehen, aber unsere Roboter kämpften tapfer und

konnten zum 2:2 ausgleichen. Deshalb wurde ein Penalty-Shootout[2] notwendig, um das Finale entscheiden zu können. Der Schiedsrichter musste nun erst noch die Regeln lesen, was etwas Zeit benötigte, uns jedoch endlos erschien und unsere Nerven bis zum Zerreißen spannte!

Wir durften zuerst angreifen, verfehlten aber das Tor ebenso, wie der Angreifer der NuBots. Beim zweiten Versuch trafen sowohl wir, als auch die NuBots, also immer noch Gleichstand. Dass es so nicht bleiben konnte, sahen sowohl unser Angreifer, als auch der Angreifer der NuBots ein: unser Angreifer schoss noch ein Tor! Der Angreifer der NuBots daraufhin nicht. Bei dem nächsten Angriff der NuBots verlies unser Torwart das Tor, um an den Ball zu kommen. Nach einer kurzen Rangelei sah er den Ball nicht mehr, da der Ball hinter ihm lag, und unser Torwart schoss den Ball rückwärts bis auf die Torlinie, wo der Ball dann liegen blieb. Damit hatten wir verdammt viel Glück. Nun sah unser Torwart den Ball wieder und konnte ihn sogar vor dem Angreifer der NuBots erreichen und vom Tor weg schießen. Jedoch rollte der Ball nur bis zur seitlichen „Aus“-Linie und blieb dort liegen. Und nochmal stellte unser Torwart seine Schnelligkeit unter Beweis, erreichte den Ball deutlich vor dem Angreifer der NuBots und spielte den Ball mit einem wunderschönen Kopfstoß ins gegnerische Tor! Das war zwar dann kein Tor für uns, aber der Angriff war endlich erfolgreich abgewehrt. Unser Angreifer schoss keine weiteren Tore, so dass die „Trainer“ beim letzten Angriff der NuBots nochmal heftig zitterten. Aber die NuBots trafen das Tor ebenfalls nicht!

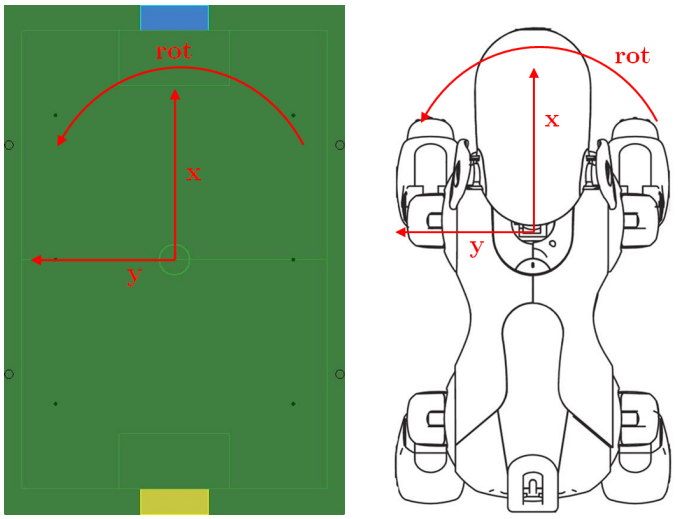
WELTMEISTER! Wir sind Weltmeister! Die Microsoft™ Hellhounds (als Teil des GermanTeam) sind WELTMEISTER!

Damit haben wir sämtliche Wettbewerbe gewonnen, an denen wir teilgenommen haben! 2005 sind die Microsoft™ Hellhounds ungeschlagen!

[2]Beim Penalty-Schootout wird abwechselnd je fünf Mal auf ein Tor geschossen. Es stehen sich dabei ein Angreifer und ein Torwart gegenüber. Wenn ein Tor fällt, oder wenn der Ball ins Aus geht, ist der Versuch vorüber und die andere Mannschaft darf angreifen.

A Koordinatensysteme

Es gibt verschiedene Koordinatensysteme, die benutzt werden. Beispielsweise kann der Ball in relativen Koordinaten zum Roboter (siehe Abbildung A.1 b)) modelliert werden. Die *RobotPose* muss natürlich in Feldkoordinaten (siehe Abbildung A.1 a)) angegeben werden.



a) Feldkoordinaten b) Roboterkoordinaten

Abbildung A.1: Koordinatensysteme

B Neuronale Netze

B.1 Aufbau Neuronaler Netze

Ein Neuronales Netz besteht aus einer Menge von Neuronen, die mittels Verbindungen miteinander verknüpft sind. Die Neuronen sind schichtenweise angeordnet. Es gibt drei Typen von Schichten:

- die Eingabeschicht (Input layer)
- die Ausgabeschicht (Output layer)
- versteckte Schichten (Hidden layers), wobei die Anzahl der versteckten Schichten nicht beschränkt ist

Die Verbindungen zwischen den Neuronen bestimmen die Netzwerkstruktur des Neuronalen Netzes. In Feed-Forward-Netzwerken ist jedes Neuron mit allen Neuronen der nächsten Schicht verbunden, es gibt keine Verbindungen innerhalb einer Schicht oder Verbindungen zu früheren Schichten. Das Neuronale Netz in Abbildung B.1 stellt ein Feed-Forward-Netzwerk dar. In rekurrenten Netzwerken dagegen sind auch Verbindungskanten zu früheren Schichten möglich.

Jede Verbindung besitzt ein Gewicht w_{ij}, wobei i den Startknoten und j den Endknoten der Verbindung bezeichnet. Die Gewichte werden während des Lernvorgangs eines Neuronalen Netzes geändert.

Jedes Neuron besitzt Eingangs- und Ausgangskanten, den Netinput, eine Aktivierungsfunktion, einen Aktivierungszustand und eine Ausgabefunktion. Der Netinput jedes Neurons berechnet sich aus den Outputwerten der Neuronen, mit denen es

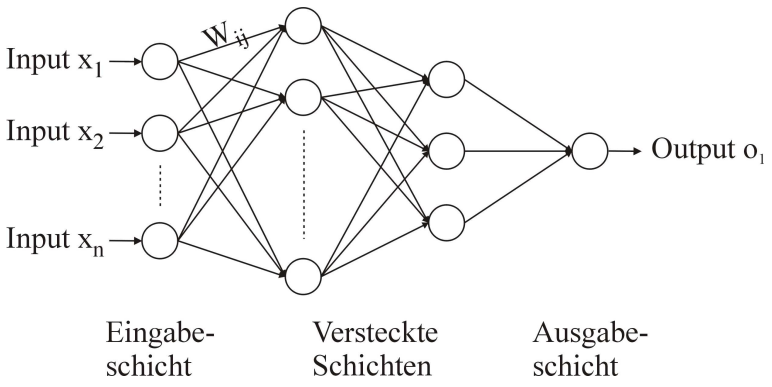


Abbildung B.1: Neuronales Netz - Feed-Forward-Netzwerk

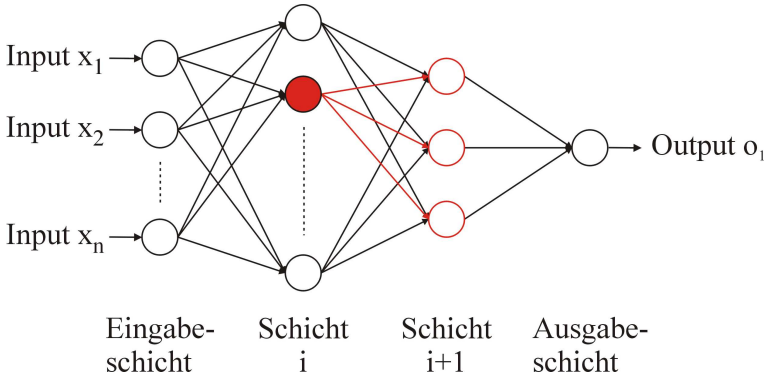


Abbildung B.2: Backpropagation

durch die Eingangskanten verbunden ist und den Gewichten dieser Verbindungskanten. Die Aktivierungsfunktion berechnet aus dem Netinput den Aktivierungszustand eines Neurons. Die Ausgabefunktion berechnet dann aus dem Aktivierungszustand die Ausgabe des Neurons, dieser Ausgabewert wird über die Ausgangskanten an die nächsten Neuronen weitergeleitet.

B.2 Lernalgorithmen Neuronaler Netze

Die Aufgabe von Neuronalen Netzen ist es, mit Hilfe einer Menge von vorgegebenen Input- und Outputdaten die Gewichte in dem Netzwerk so anzupassen, dass zu einem beliebigen Input der richtige Output berechnet wird. Ein möglicher Algorithmus zum Erlernen der Gewichte ist der Backpropagation-Algorithmus. Zuerst wird die Ausgabe des Neuronalen Netzes berechnet. Die Idee des Backpropagation-Algorithmus besteht jetzt darin, dass für den Fehler eines Neurons j, das heißt für die Differenz zwischen der berechneten Ausgabe und der erwarteten Ausgabe des Neurons j, all diejenigen Neuronen mitverantwortlich sind, die mit j verbunden sind, deren Output in die Berechnung des Netinputs von Neuron j einfließt. Je größer der Fehler ist, desto stärker müssen die Gewichte der Verbindungskanten verändert werden. Für die Ausgabeschicht kann dieser Fehler direkt berechnet werden, da sowohl der tatsächliche Ausgabewert als auch der erwartete Ausgabewert bekannt sind. Zur Anpassung der Gewichte früherer Schichten muss der Fehler dagegen zurückpropagiert werden. Um den Fehler des Neurons in Schicht i zu berechnen, wird eine gewichtete Summe der Fehlerterme aller Neuronen in Schicht $i+1$ genommen, mit denen das Neuron in Schicht i verbunden ist. So wird der Fehler Schicht für Schicht in Richtung Eingabeschicht weitergereicht, wobei jeweils Korrekturen an den entsprechenden Gewichten vorgenommen werden.

C Evolutionäre Algorithmen

Evolutionäre Algorithmen sind randomisierte Optimierungsverfahren. Zu Beginn existiert eine Population von n Individuen, jedes Individuum stellt eine potentielle Lösung für das gegebene Problem dar. Neue Nachkommen werden geschaffen durch:

- Mutation : Variation der Individuen
  Ein Individuum wird um einen Wert x verändert, wobei die Mutationsschrittweite x eine normalverteilte Zufallsvariable mit dem Mittelwert 0 ist, d.h. kleine Änderungen um 0 herum sind wahrscheinlicher als große Mutationsschrittweiten.

- Rekombination : Kombination der Information mehrerer Elternindividuen

- Selektion : survival of the fittest

  Es werden die n besten Individuen ausgewählt, diese bilden die Population für die nächste Evolutionsschleife. Die Verbesserung eines Individuums wird durch die Fitnessfunktion/Bewertungsfunktion gemessen.

Diese 3 Schritte werden so lange wiederholt, bis ein Abbruchkriterium erfüllt ist.

Es gibt verschiedene Evolutionsstrategien, die sich in der Anzahl der Eltern, der Nachkommen und der Art der Selektion unterscheiden. Für die Optimierung der Parameter des SelfLocators nutzen wir die 1+1 Evolutionsstrategie.

1+1 ES: Die erste Eins bezieht sich auf die Anzahl der Eltern, die zweite Eins auf die Anzahl der Nachkommen. Wir starten also mit 1 Individuum, durch die Mutation entsteht 1 neuer Nachkomme. Das „+“ bedeutet, dass bei der Selektion sowohl Elter als auch Nachkomme verglichen werden und das Beste der zwei Individuen überlebt. Rekombination ist bei einer Population von nur einem Individuum nicht möglich.

D Stochastik

Markov-Kette und -Eigenschaft Gleichung (D.1) heisst Markov-Kette n-ter Ordnung. Die Tatsache, dass ein solcher Prozess nur von einer begrenzten Zahl vorhergehender Zustände abhängt, nennt man Markov-Eigenschaft. In Fall des SelfLocators geht man von einer Markov-Kette erster Ordnung aus.

$$P(X_{t+1} = x_{t+1}|X_t = x_t, \ldots, X_0 = x_0) = \\ P(X_{t+1} = x_{t+1}|X_t = x_t, \ldots, X_{t-n+1} = x_{t-n+1}) \tag{D.1}$$

Varianz/Covarianz-Matrix Die Varianz einer Verteilung $V(X)$ einer Zufallsvariable X ist die mittlere quadratische Abweichung vom Erwartungswert E: $V(X) = \frac{1}{n}\sum_i (x_i - E)^2$, ihre Wurzel ist die Standardabweichung σ. Ist die Verteilung mehrdimensional, können neben den Varianzen für die einzelnen Dimensionen noch Zusammenhänge zwischen diesen existieren, die Covarianzen: $C_{x,y} = \sigma_x \cdot \sigma_y$.

Die Varianz/Covarianz-Matrix (Abbildung D.2) einer Verteilung beinhaltet für die Dimensionen die Varianzen (auf der Hauptdiagonale) und Covarianzen. Ihre Eigenvektoren weisen in die Richtung einer Ausdehnung der Verteilung und der zugehörige Eigenwert beschreibt die Varianz der Verteilung in die jeweilige Ausdehnungsrichtung. Im zweidimensionalen Fall wird so über ihre Haupt- und Nebenachse eine Ellipse beschrieben.

$$\begin{pmatrix} \sigma_x^2 & \sigma_x\sigma_y \\ \sigma_y\sigma_x & \sigma_y^2 \end{pmatrix} \tag{D.2}$$

E XTC Referenz

E.1 Aufbau von XABSL

In jedem Durchlauf der XABSL-Engine wird genau ein *Basic Behavior* bestimmt, das der Roboter ausführt. Ein Basic Behavior ist im wesentlichen eine C++ Funktion, die mit bestimmten Parametern ausgeführt wird. Beispiele für Basic Behaviors sind `stand()` oder `walk(type, speed_x, speed_y, rotation_speed)`.

Wichtigstes strukturelles Element von XTC ist die *Option*. Eine Option kapselt einen endlichen Automaten und enthält deshalb sinnvollerweise mehrere Zustände (*States*). Für diese Zustände kann man eine Übergangsfunktion in Abbhängikeit von bereitgestellten Eingabesymbolen und Funktionen definieren. Dies passiert in XABSL mit den sogenannten *decision-trees*, in XTC in geschachtelten `if`-Anweisungen. Die XABSL-Engine welchselt so lange die Zustände, bis ein Zustand das zweite Mal angelaufen wird. Dann erst werden die mit dem ausgewählten State verknüpften Aktionen ausgeführt. Diese Umfassen das Setzen von so genannten *output symbols* und die Verzweigung zu einer weiteren Option oder zu einem Basic Behavior. Sowohl dem Basic Behavior als auch der Folgeoption können Parameter übergeben werden.

Die Ausführung beginnt immer bei einer festgelegten Option, der so genannten *Root Option*. Die XABSL-Engine besitzt ein Gedächtnis für die letzten ausgewählten States einer Option. Wenn im nächsten Durchlauf über den gleichen Pfad eine Option aktiviert wird, dann beginnt die Ausführung ihrer State Machine beim letzten ausgewählten State. So lassen sich über mehre Options hinweg sequenzielle Abläufe realisieren, wie zum Beispiel zum Ball gehen, um den Ball in Richtung Tor drehen und dann schießen.

E.2 Bezeichner

Bezeichner sind die Namen, mit denen die Sprachelemenete wie Optionen, States, Variablen und Funktionen identifiziert werden. Bezeichner beginnen im Allgemeinen mit einem Buchstaben und dürfen danach aus Buchstaben, Ziffern, dem Unterstrich und dem Punkt bestehen.

Innerhalb von XABSL-Bezeichnern wird durchgängig das Minuszeichen als Trennzeichen benutzt. Dies führt allerdings zu Konflikten, wenn man versuchen würde in XTC Variablen oder Funktionen voneinander zu subtrahieren. Deshalb werden bijektiv alle Minuszeichen in XABSL-Bezeichnern auf Unterstriche in XTC-Bezeichnern abgebildet. Weil Funktions- und Behaviordeklarationen auch in C++ Quelltext vorgenommen werden müssen, stehen dort weiterhin die Bezeichner mit Unterstrichen.

Ebenso in der generierten Dokumentation und in den Dialogen von RobotControl und RobotControl 2. Dies hat sich als häufige Fehlerquelle im Umgang mit XTC erwiesen.

Eine zweite Besonderheit im Umgang mit Bezeichnern ist die Möglichkeit, XTC-Schlüsselwörter in XABSL-Bezeichnern zu verwenden. Beispielsweise heißen States oft „initial“, dies ist aber ein reserviertes Wort in XTC. Als Lösung stellt man solchen Bezeichnern in XTC zwei Unterstriche voran.

In XABSL sind die Bezeichner von Enumerationen den Enumerationswerten oft vorangestellt, aber diese Schreibweise wird nicht durchgängig verwendet. Da in XTC grundsätzlich auf eine Wiederholung dieser Präfixe verzichtet werden soll, fehlt dem Übersetzer die Information, wie die vollständigen Bezeicher zu generieren sind. Gleiches gilt für die Parameter von Funktionen und Basic Behaviors. Daher kann man bei deren Deklaration durch Angabe des Schlüsselwortes `unqualified` das Voranstellen des Enumerations, Funktions- oder Behaviornamens verhindern.

E.3 Schlüsselwörter

Die folgenden Wörter dürfen nicht als Bezeichner für Variablen, Optionen, etc. eingesetzt werden, da sie eine spezielle Bedeutung für den Übersetzer haben:
`action`, `action_done`, `agent`, `behavior`, `bool`, `const`, `decision`, `else`, `enum`, `float`, `function`, `goto`, `if`, `include`, `initial`, `input`, `namespace`, `option_time`, `option`, `output`, `state_time`, `state`, `stay`, `target`, `unqualified`.

E.4 Typen

Es gibt nur drei verschiedene Grundtypen in XTC.

- Enumerationstypen: *Enumerated Input Symbols* können mit Hilfe des == Operators mit einem Enumerationswert verglichen werden. *Enumerated Output Symbols* kann mit dem = Operator ein Enumerationswert zugewiesen werden.
- Boolsche Werte entstehen als Ergebnisse von Vergleichen oder durch das Referenzieren von *Boolean Input Symbols*.
- Fließkommazahlen kommen als Zahlkonstanten vor, als symbolische Konstanten, als *Decimal Input Symbols* und Rückgabewert von Funktionen. Die Parameter von Funktionen, Optionen und Basic Behaviors sind immer Fießkommazahlen, niemals Enumerations- oder Boolsche Werte.

E.5 Deklarationen

Basic Behaviors, Funktionen und Symbole müssen in XTC deklariert werden. Das liegt daran, dass diese Sprachelemente eigentlich im C++ Quelltext definiert sind

und dem XTC Compiler vor der Benutzung bekannt gemacht werden müssen. Die Deklarationen von Basic Behaviors sieht folgendermaßen aus:

```
/** Common basic behaviors that are shared by
    all Xabsl2 BehaviorControl solutions. */
namespace common_basic_behaviors("Common Basic Behaviors") {

  /** Executes a normal walk with the given parameters. */
  behavior walk {
    /** Specifies the used walking type. */
    type [integer value] "integer value";
    /** X Speed. */
    speed_x [-450..450] "mm/s";
    /** Y Speed. */
    speed_y [-450..450] "mm/s";
    /** Rotational speed. */
    rotation_speed [-200..200] "deg/s";
  }
}
```

Hier wird zuerst ein Namespace deklariert und darin ein Basic Behavior mit dem Bezeichner `walk` und den vier Parametern `type`, `speed_x`, `speed_y` und `rotation_speed`. Alle weiteren Angaben wie die Kommentare (/** ... */), die Bereichsangaben in eckigen Klammern und die Einheiten in Ausführunszeichen werden nur für die Generierung der Dokumentation herangezogen und haben keinen Einfluss auf den Ablauf eines XTC Programms. Der Namespace in dem etwas deklariert wird, hat in XTC auch keine praktische Bedeutung, diese Angabe wird auch nur für Strukturierung der Dokumentation genutzt. Die Parameter sind immer auf den *double* Fließkommazahlen festgelegt. Funtionen berechnen aus Parametern einen Rückgabewert. Sie werden ganz ähnlich deklariert:

```
/** Some common math functions. */
namespace math_functions("Math functions") {

  /** Calculates the distance to a specified point on the field */
  function distance_to "mm" {
    /** The x position on the field */
    x [-3000..3000] "mm";
    /** The y position on the field */
    y [-2000..2000] "mm";
}
```

Es gibt noch fünf andere Arten von Deklarationen, nämlich

- Konstanten

```
/** Symbols for the special actions that can be performed. */
namespace special_action_symbols("Special Action Symbols") {

  /** The number of a special action used by the special-action skill */
  float const special_action.any_left = 0;

  /** The number of a special action used by the special-action skill */
  float const special_action.arm_left = 1;

...
```

- enumerierte Ausgabesymbole

```
/** a mode for the head control. */
  enum output head_control_mode {
    none,
    search_for_ball,
    search_auto,
    search_for_landmarks,
    look_between_feet,
    look_left,
    look_right
  }
```

- enumerierte Eingabesymbole

```
 /** The state of the mouth */
  enum input robot_state.mouth_state {
    unqualified mouth_open,
    unqualified mouth_closed
  }
```

- boolsche Eingabesymbole

```
  /** the ball was just seen */
  bool input ball.just_seen;
```

- Eingabesymbole die eine Fließkommazahl zurückliefern

```
  /** disctance to the seen ball */
  float input ball.seen.distance "mm";
```

Eingabesymbole sind nur-lesbare Variablen, deren Werte in C++ berechnet werden und auf denen die Auswahl des Verhaltens beruht. Mit Ausgabesymbolen wird beispielsweise die Kopfbewegung oder die LEDs des Roboters gesteuert.

E.6 Options und States

Durch das Schlüsselwort `option` wird eine Option eingeleitet. Darauf folgt optional die Deklaration der Parameter, die man an die Option übergeben kann. Anschliessend werden die in der Option enthaltenen States definiert. Das Schlüsselwort `initial` vor der Definition eines States markiert diesen als Startzustand. Für jede Option muss es genau einen Startzustand geben. `target` vor dem Statenamen gibt an, dass es sich bei dem State um einen Zielzustand handelt. Das Ereichen von diesen Zuständen kann von der aufrufenden Option mit der `action_done`-Anweisung abgefragt werden.

```
/** Dribbles the ball */
option dribble_ball {

  /** The angle that the ball should be dribbled */
  @destination_angle [-180..180] "deg";

  initial state go_to_ball_far
  {
  }

  state rotate_to_angle
  {
   ...
  }

  target state dribble
  {
   ...
  }
  ...
}
```

@destination_ angle wird hier als Parameter der Option *dribble_ ball* deklariert und kann somit innerhalb dieser Option verwendet werden. Solche Parameter haben innerhalb der Option ein `@` vorangestellt, um sie als Parameter zu kennzeichnen und um Namenskonflikte mit Input Symbols zu vermeiden.

Ein State besteht wiederum aus genau zwei Teilen. Einem `decision`-Block der eine Folge von `if`-Anweisungen enthält und dem `action`-Block der angibt, was passieren soll, falls dieser Zustand ausgewählt wird. Darin befinden sich optional mehrere Zuweisungen für Ouput Symbols und ein Verweis auf eine weitere Option oder ein Basic Behavior.

```
  ...
  initial state go_to_ball_far
```

```
  {
    decision
    {
      if(ball.seen.distance < 500)
      {
        goto(go_to_ball_near);
      }
      else
      {
        stay();
      }
    }
    action
    {
      head_control_mode = search_auto;
      go_to_ball_propagated(distance = 0, max_speed = 450,
                            walk_type = walk_type.gt2004mode);
    }
}
...
```

Hier wird im `decision`-Block zum State *go_ to_ ball_ near* verzweigt, falls der Abstand zum Ball weniger als 500 mm beträgt. Andernfalls wird dieser Zustand ausgewählt (`stay();`) und der `action`-Block ausgeführt. In diesem Beispiel bedeutet das das setzen des Output Symbols *head_ control_ mode* auf den Wert *search_ auto* und das die Ausführung der XABSL-Engine bei der Option *go_ to_ ball_ propagated* fortgsetzt wird. Dieser Option werden Werte für die drei Parameter *distance*, *max_ speed* und *walk_ type* übergeben.

`if`-Anweisungen lassen sich auch schachteln, so dass auch komplexer Bedindungen elegant möglich sind. Abgesehen daon darf in den geschweiften Klammern ein `goto(...);` stehen, mit dem ein Übergang zu einem anderen State der Option erzeugt wird oder ein `stay();`, was eine Auswahl dieses Zustandes bewirkt. Wichtig ist, dass der äußerste `if`-Block immer einen `else`-Zweig besitzt. Damit wird sichergestellt, dass in jedem Fall eine Entscheidung getroffen werden kann.

Es besteht auch die Möglichkeit einen gemeinsamen `decision`-Block für alle States zu definieren, dieser steht dann zwischen Parameterdeklarationen und erstem State in der Option und verhält sich so, als würde man diesen Block vor jeden `decision`-Block der einzelnen States einfügen.

E.7 Ausdrücke

Ausdrücke sind die Codeteile, die bei einer `if`-Anweisung innerhalb der runden Klammern stehen und bei Funktions-, Option- oder Behavioraufrufen für jeden Pa-

rameter hinter den Gleichheitszeichen steht. Ausdrücke können entweder zu Wahrheitswerten oder Fließkommazahlen ausgewertet werden. Ausdrücke sind primär

- Zahlen und Konstanten
- Fließkomma und boolsche Eingabesymbole
- Vergleiche von enumerierten Eingabesymbolen mit zugehörigen Enmuerationswerten
- die Schlüsselwörter `state_time`, `option_time`, `action_done`

Darüberhinaus lassen sich Ausdrücke mit den aus C++ bekannten Operatoren `<`, `<=`, `==`, `!=`, `>=` ,`>`, `+`, `-`, `*`, `/`, `%`, `!`, `&&`, `||`, `? (...) : (...)` und aus der Klammerung von Ausdrücken zusammensetzen.

E.8 Agents

Agentdeklarationen bilden die Einstiegspunkte für die Ausführung der Optionen. Mehrere Agents ermöglichen ein leichtes Umschalten des Verhaltens.

```
/** The soccer agent */
agent soccer("Soccer", play_soccer);
```

Dieses Beispiel definiert einen Agent mit dem Bezeichner *soccer* und der Startoption *play_ soccer*

E.9 Dateiaufteilung

XABSL erzwingt mit seiner XML Struktur eine bestimmte Aufteilung auf Dateien. Um weiterhin die XABSL-Dokumentation generieren zu können, müssen einige Regeln eingehalten werden.

- Es gibt nur eine Hauptdatei, in der Agents definiert werden und mit `include`-Anweisungen alle anderen Quelldateien referenziert werden. Diese sollte `agents.xtc` heißen.
- Jede andere XTC-Datei enthält entweder genau einen `namespace` mit Deklarationen oder eine Option.
- In einem `namespace` dürfen entweder nur Basic Behaviors oder Funktionen und Symbole deklariert werden.

Literaturverzeichnis

[1] J. Bruce, „CMVision - Realtime Color Vision“. http://www.cs.cmu.edu/~jbruce/cmvision/, 2004.

[2] A. Doucet, N. de Freitas und N. Gordon, Hrsg., *Sequential Monte Carlo Methods in Practice*. New York: Springer-Verlag, 2001. Online: http://www-sigproc.eng.cam.ac.uk/~ad2/book.html.

[3] E. Gamma, R. Helm, R. Johnson und J. Vlissides, *Entwurfsmuster - Elemente wiederverwendbarer objektorientierter Software*, S. 239–253. Addison-Wesley-Longman, 2004.

[4] B. Hengst, D. Ibbotson, S. B. Pham und C. Sammut, „Omnidirectional locomotion for quadruped robots. in robocup 2001 robot soccer world cup v“, in *Lecture Notes in Computer Science*, S. 368–373, Springer, 2002.

[5] IEEE Computer Society, „Carrier sense multiple access with collision detection (CSMA/CD) access method and physical layer specifications“. http://standards.ieee.org/getieee802/download/802.3-2002.pdf, 2002.

[6] Information Sciences Institute, „RFC 791 - Internet Protocol Darpa Internet Program Protocol Specification“. http://www.ietf.org/rfc/rfc791.txt, 1981.

[7] J. Postel, „RFC 768 - User Datagram Protocol“. http://www.ietf.org/rfc/rfc768.txt, 1980.

[8] M. Jüngel, „A Vision System for RoboCup“, Diplomarbeit, Humboldt-Universität zu Berlin, 2004. http://www.informatik.hu-berlin.de/~juengel/papers/juengel-diploma-thesis.pdf.

[9] C. C. T. Kwok und D. Fox, „Map-Based Multiple Model Tracking of a Moving Object“, in *RoboCup* (D. Nardi, M. Riedmiller, C. Sammut und J. Santos-Victor, Hrsg.), Bd. 3276 der Reihe *Lecture Notes in Computer Science*, S. 18–33, Springer, 2004. Online: http://www.cs.washington.edu/ai/Mobile_Robotics/projects/postscripts/tracking-robocup-04.pdf.

[10] C. C. T. Kwok, D. Fox und M. Meila, „Real-Time Particle Filters“, in *NIPS* (S. Becker, S. Thrun und K. Obermayer, Hrsg.), S. 1057–1064, MIT Press, 2002. http://books.nips.cc/papers/files/nips15/AA73.pdf.

[11] M. Lauer, S. Lange und M. Riedmiller, „Modeling Moving Objects in a Dynamically Changing Robot Application“, in *KI* (U. Furbach, Hrsg.), Bd. 3698 der Reihe *Lecture Notes in Computer Science*, Springer, 2005. Erscheint demnächst.

[12] S. Lenser und M. M. Veloso, „Sensor Resetting Localization for Poorly Modelled Mobile Robots“, in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, S. 1225–1232, IEEE, 2000. Online: `http://www.ri.cmu.edu/pub_files/pub2/lenser_scott_2000_1/lenser_scott_2000_1.pdf`.

[13] M. Lötzsch, „XABSL - A Behavior Engineering System for Autonomous Agents“, Diplomarbeit, Humboldt-Universität zu Berlin, 2004. `http://www.martin-loetzsch.de/papers/diploma-thesis.pdf`.

[14] MSDN Developer Center Deutschland, „.NET Framework - Anwendungsentwicklung mit der .NET-Klassenbibliothek“. `http://www.microsoft.com/germany/msdn/netframework/default.mspx`, 2005.

[15] W. Nistico und T. Röfer, „Improving Percept Reliability in the Sony Four-Legged League“, in *RoboCup 2005: Robot Soccer World Cup IX*, Lecture Notes in Artificial Intelligence, Springer, 2006. Erscheint demnächst. Online: `http://www.tzi.de/kogrob/papers/rc06-vision.pdf`.

[16] T. Röfer, B. Altmeyer, R. Brunn, H.-D. Burkhard, I. Dahm, M. Dassler, U. Düffert, D. Göhring, V. Goetzke, M. Hebbel, J. Hoffmann, M. Jüngel, M. Kunz, T. Laue, M. Lötzsch, W. Nisticó, M. Risler, C. Schumann, U. Schwiegelshohn, M. Spranger, M. Stelzer, O. von Stryk, D. Thomas, S. Uhrig und M. Wachter, „GermanTeam RoboCup 2004“, Technical Report, 2004. Online: `http://www.germanteam.org/GT2004.pdf`.

[17] RoboCup Technical Committee, „Sony Four Legged Robot Football League Rule Book“. `http://www.tzi.de/4legged/pub/Website/Downloads/Rules2004.pdf`, 2004.

[18] RoboCup Technical Committee, „Sony Four Legged Robot Football League Challenges Rule Book“. `http://www.tzi.de/4legged/pub/Website/Downloads/Challenges2005.pdf`, 2005.

[19] RoboCup Technical Committee, „Sony Four Legged Robot Football League Rule Book“. `http://www.tzi.de/4legged/pub/Website/Downloads/Rules2005.pdf`, 2005.

[20] R. Rojas, *Neural networks: a systematic introduction.* New York, NY, USA: Springer-Verlag New York, Inc., 1996.

[21] H.-P. Schwefel, *Evolution and Optimum Seeking.* Sixth-Generation Computer Technology, New York: Wiley Interscience, 1995.

[22] G. Welch und G. Bishop, „An Introduction to the Kalman Filter“, Technical Report TR 95-041, University of North Carolina at Chapel Hill, 1995. Online: http://www.cs.unc.edu/~welch/media/pdf/kalman_intro.pdf.

Abbildungsverzeichnis

Index