Verbesserung der Selbstlokalisierung im Roboterfußball

mittels optischer Umgebungsmerkmale

Bachelorarbeit

im Fachgebiet Bildverarbeitung

vorgelegt von:	Samuel Eckermann
Studienbereich:	Informatik
Erstgutachter:	Dr. rer. nat. Johannes Waldmann

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Leipzig, den 12.09.2012

Samuel Eckermann

Zusammenfassung

Diese Arbeit beschäftigt sich mit einem wichtigen Thema aus der Standard Platform League des RoboCup: Orientierung. Der RoboCup ist eine Initiative zur Förderung der Robotik an der sich viele Universitäten und Fachhochschulen beteiligen. In der Standard Platform League spielen humanoide Roboter autonom Fußball.

Naturgemäß ist ein wichtiges Thema der Liga die Orientierung auf dem Fußballfeld. Bisher war dies mit den unterschiedlich farbigen Toren recht leicht zu realisieren. Seit einer Regeländerung von 2012, wonach nun mit gleichfarbigen Toren gespielt wird, steht die Liga vor einer neuen Herausforderung.

In dieser Arbeit habe ich eine innovative Lösung für dieses Problem entwickelt. Die Roboter extrahieren während dem Spiel markante Merkmale von außerhalb des Spielfeldes. Mit diesen bauen sie sich ein Model der Umgebung auf, welches sie, um es möglichst schnell zu vervollständigen, auch untereinander kommunizieren. So ist eine effektive Orientierung möglich.

Dieser Ansatz ist anderen Lösungen in einigen Punkten überlegen. Zum Beispiel ist die Nutzung eines Teamballes von der Anzahl der Roboter auf dem Feld abhängig, oder die Nutzung des Torwarts von dessen Vorhandensein auf dem Feld.

Inhaltsverzeichnis

1 Einleitung

Die Robotik ist ein Forschungsgebiet der Zukunft. Schon jetzt gibt es Roboter die menschliche Emotionen nachahmen oder komplexe industrielle Prozesse ausführen. Auch in der Raumfahrt und der Tiefseeforschung sind Roboter nicht mehr wegzudenken. Diese Entwicklung wird weiter fortschreiten und dazu trägt auch der RoboCup einen Teil bei. Das NAO-Team der HTWK Leipzig nimmt in der Standard-Plattform League am RoboCup teil. In dieser Liga spielen humanoide Roboter autonom Fußball.

1.1 Motivation

Die Grundvoraussetzung für ein erfolgreiches Spiel in der Standard Platform League ist die korrekte Lokalisierung. Dafür stehen dem Roboter mehrere Sensoren, wie zwei Kameras, Ultraschall und zwei Gyrometer zur Verfügung. Das NAO-Team HTWK Leipzig benutzt momentan für die Lokalisierung einen Algorithmus der die Feldlinien sowie die Torpfosten im Kamerabild erkennt, und damit im Idealfall die Anzahl der möglichen Positionen auf 2 einschränkt. Dabei lässt sich immer die Eine durch Drehung um 180° um den Mittelpunkt des Spielfeldes auf die andere Position abbilden.

Da der NAO kein GPS und keinen Kompass besitzt lag es bisher nahe die Farben der Tore zu betrachten, um herauszufinden, welche der beiden Positionen die korrekte ist. Durch die unterschiedliche Farben der Torpfosten, blau und gelb, konnte, wenn ein Pfosten im Kamerabild erkannt wurde entschieden werden auf welcher Seite sich der Roboter befand. Wenn kein Tor erkannt wurde lässt sich die richtige Position über Odometrie[1] bestimmen, das heißt der NAO merkt sich wie er sich von seiner letzten bekannten Position aus fortbewegt hat um damit zu errechnen, welcher der möglichen Standpunkte der wahrscheinlichste ist.

Da jedoch die Anforderungen in der SPL von Jahr zu Jahr erhöht werden, um sich dem Fußballspiel der Menschen immer weiter anzunähern, sind seit 2012 die Tore einfarbig. Damit fällt die Möglichkeit weg sich allein anhand des Spielfeldes zu lokalisieren, da dieses nun keine Asymmetrien mehr aufweist. Zwar ist eine fest definierte Startposition für jeden NAO auf dem Feld gegeben, doch die Vorstellung das mittels der Odometrie jeder Zeit die richtige Position herausgefunden werden könnte, ist weit von der Realität entfernt. Beispielsweise ist die Odometrie nicht

[1] Die Positionsbestimmung eines mobilen Systems anhand der Daten seiner Bewegung

störungsfrei und somit sind Delokalisierungen nicht ausgeschlossen und können nun auch nicht mehr erkannt werden. Der Roboter dreht sich zum Beispiel um 90°, denkt aber er hätte sich um 180° gedreht, oder aber er fällt um und dreht sich dabei, ohne dass es die Odometrie mitbekommt. Wenn er nun wieder aufsteht, steht er anders herum da, denkt aber dass er in die gleiche Richtung wie vorher blickt. Wenn dies nahe der Mittellinie passiert, vertauscht der NAO die Seiten und spielt von da an gegen sein eigenes Tor. Als Beispiel seien die meisten Spiele während der German Open 2012 in Magdeburg, wie im Semifinale Robo-Eireann gegen NAO-Team HTWK Leipzig, in dem mehrere Eigentore fielen, genannt.

1.2 Zielsetzung und Abgrenzung

Um den Verlust der Orientierung zu vermeiden braucht der Roboter also neue Methoden. Und genau hier setzt diese Bachelor Arbeit an: In Anlehnung an den Menschen sollen Merkmale in der Umgebung gefunden werden, die Hinweise auf die Blickrichtung und damit auch auf die Position liefern. Mein Ziel ist es diese Merkmale aus der Spielfeldumgebung zu extrahieren, um sie dann bei der Lokalisierung wieder korrekt zum zugehörigen Abschnitt der Umgebung zuzuordnen und damit die Informationen, welche bisher die Farbe des Tors geliefert hatte zu ersetzen. Dadurch soll das Umfallen in Nähe des Mittelkreises, und andere Schwächen der Lokalisierung ausgeglichen werden und selbst bei Delokalisierung nach beliebigem Umsetzen des Roboters eine korrekte Relokalisierung ermöglicht werden. Die Aussagekräftigkeit der gefundenen Merkmale, sowie deren korrekte Zuordnung wird mittels Testdatenbank optimiert.

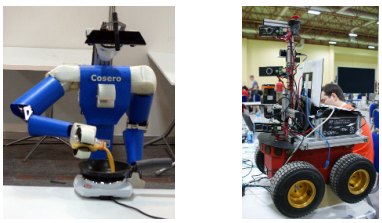
Die Erkennung der Feldlinien im Kamerabild und die Berechnung der möglichen Positionen anhand von diesen, wird nicht Bestandteil dieser Arbeit sein.

1.3 Aufbau der Arbeit

Die Arbeit gliedert sich in 4 große Bestandteile. Zunächst sage ich einiges zum RoboCup. Dann folgt ein Kapitel, in welchem die nötigen theoretischen Voraussetzungen zum Verstehen des Hauptteils erläutert werden. Der Hauptteil schließlich spezifiziert die vorhanden Daten und benötigten Algorithmen und geht dann auf deren Umsetzung ein. Danach wird das Kommunizieren (Sharing) der Daten zwischen den Robotern beschrieben.

2 Wissenschaftlicher Rahmen - Der RoboCup

Die Idee des RoboCups ist, bis zum Jahr 2050 gegen den dann amtierenden menschlichen Weltmeister im Fußball mit humanoiden Fußballrobotern zu gewinnen. Inzwischen sind noch Wettbewerbe in neuen Anwendungsbereichen hinzugekommen, wie zum Beispiel Haushalts oder Rettungsroboter[2].



(a) (b)

Abbildung 2.1: (a) zeigt einen Roboter, wie er in RoboCup@Home verwendet wird [RoboCup, 2012a]. (b) ist ein Rettungsroboter [RoboCup, 2012b].

Im Folgenden wird kurz auf die verschiedenen Ligen des RoboCup Soccer eingegangen und deren Lösungsansätze zur Lokalisierung vorgestellt:

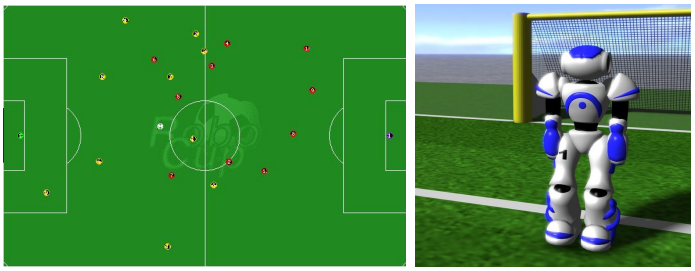
2.1 Die verschiedenen Ligen des RoboCup

Soccer Simulation League:
Die Simulation League benötigt keine Hardware und konzentriert sich so ausschließlich auf die Entwicklung von Algorithmen. Dies schließt künstliche Intellegenz und Team-Strategie ein. Die Simulation League wird weiter in die 2D und die 3D Liga unterteilt[3].

Ein Fußballspiel der 2D Simulation League findet zwischen 2 Teams mit je 12 Programmen, Agenten genannt, statt (siehe Abb. 2.2 (a)). Ein Server kennt die gesamte Situation und sendet den Agenten verrauschte Informationen über ihre Umgebung. Nun besteht die Aufgabe, diese Daten zu entrauschen und möglichst geschickte Aktionen auszuführen.

Die 3D Liga erhöht den Realismus der virtuellen Welt, durch eine extra Dimension und eine komplexere Physik, gegenüber der 2D League. Seit 2006 werden in dieser Liga humanoide Roboter simuliert (siehe Abb. 2.2 (b)).

[2] Vgl. . [Behnke, 2012]
[3] Vgl. [RoboCup Federation Wiki, 2012b]



(a) (b)

Abbildung 2.2: Die Abbildung zeigt Screenshots der Visualisierung für die 2D-Simulationsliga (a) [RoboCup Soccer Simulator, 2012] und die 3D-Simulationsliga (b) [SimSpark virtual Nao, 2012]

In beiden Ligen bekommen die Agenten verrauschte Positionsdaten, so dass die Lokalisierung aus der Filterung von diesen besteht.

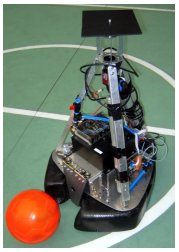
Small Size League:
In der Small Size League spielen 2 Teams, mit je 5 selbstgebauten Robotern auf Rädern gegeneinander (siehe Abb. 2.3(a)). Eine Kamera über dem Spielfeld nimmt die aktuelle Situation des Spiels auf und leitet sie an den zentralen Server weiter, auf dem die Programme der Teams Aktionen für die Roboter berechnen und per Funk an diese versenden. Die Hauptaufgabe in der Small Size League ist, neben dem Bau der Roboter, die Entwicklung von Taktiken. Die Lokalisierung übernehmen bildverarbeitende Algorithmen, welche die Roboter tracken und zudem das Spielfeld sowie den Ball erkennen. Dabei ist die Position der Roboter durch die fest angebrachte Kamera eindeutig zu bestimmen (Vgl. [RoboCup Federation Wiki, 2012a]).

Middle Size League:
Da die Roboter der Middle Size League (siehe Abb. 2.3(b)) gegenüber der Small Size League mehr als doppelt so groß sein können, sind nun alle Sensoren "onboard". Zudem gibt es keinen externen Server mehr, der die Berechnung für die Roboter übernimmt[4]. Gespielt wird mit 2 Teams zu je 5 Robotern auf einem $18m \times 12m$ großen Feld. Dabei sind die Regeln, bis auf kleine Anpassungen, mit dem Fifa-Regelwerk identisch. Die Lokalisierung erfolgt anhand der Feldlinien. Die Tore sind auch in dieser Liga gleichfarbig, so dass sie nicht für die Positionsfindung verwendet werden können. Jedoch ist es möglich die selbstgebauten Roboter mit einem Kompass auszurüsten. Zudem bewegen sich die Roboter auf Rädern fort, weswegen sie standfester sind. Dadurch kommt es bei einer guten Lokalisierung sowie einer guten Odometrie fast nicht dazu, dass Roboter die Seiten wechseln[5].

[4] [RoboCup Federation, 2012]
[5] Vgl. [MSL Technical Committee, 2011]

Humanoid League:
In der Humanoid League spielen selbstgebaute humanoide Roboter gegeneinander (siehe Abb. 2.3(c)). Die Roboter lokalisieren sich anhand der Feldlinien, wobei zur

(a) Small Size



(b) Middle Size

(c) Humanoid

Abbildung 2.3: Roboter der verschieden Ligen

Einschränkung der möglichen Positionen die Tore unterschiedlich gefärbt sind. Zudem sind unterschiedlich farbige Pfosten an den Endpunkten der Mittelline plaziert, welche die möglichen Positionen weiter einschränken.

Standard Platform League:
Die Standard Platform Liga (auch SPL) hat, als einzige Liga des RoboCup Soccer, den Schwerpunkt auf reiner Software-Entwicklung für ein Fußballspiel in realer Umgebung. Dies lässt sich nur dadurch erreichen, dass alle Teilnehmer die gleichen Hardware-Voraussetzungen erfüllen: Es ist alleine der NAO, ein Roboter der französischen Firma Aldebaran zugelassen. Zudem darf seine Hardware nicht verändert werden um ein faires Spiel zu gewährleisten. Somit entscheidet in der SPL allein die programmierte Software, welche auf dem Roboter läuft, über Sieg und Niederlage. (Etwas Glück gehört natürlich auch dazu.)

2.2 SPL

In der Standard-Plattform-Liga spielen also jeweils vier baugleiche humanoide NAOs auf einen 6×4 m^2 großen Feld gegeneinander:

2.2.1 Der NAO

Der NAO, von Aldebaran hergestellt, ist ein humanoider Roboter. Er ist mit 2 Kameras, einem Infrarot-Sensor, einem Sonar, mehreren Mikrophonen, Lautsprechern, Gyrometern und mit Beschleunigungssensoren ausgerüstet. Für diese Arbeit sind hauptsächlich die Kameras interessant, weshalb in Kapitel 3.2 noch einmal auf diese eingegangen wird.

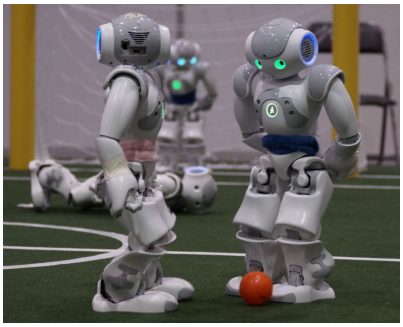


Abbildung 2.4: NAOs in einem Spiel auf der Weltmeisterschaft 2012 in Mexico

2.2.2 Das Spielfeld

Ein Spiel der SPL wird auf einem 6×4 Meter großen Feld ausgetragen. Das NAO-Team HTWK Leipzig hat sich für den Mittelpunkt des Feldes als Ursprung der Welt-Koordinaten entschieden. Die positive x-Koordinate zeigt dabei immer zum gegnerischen Tor. Abb. 2.5 zeigt das Welt-Koordinatensystem und wie das Spielfeld nach den Regeln auszusehen hat.

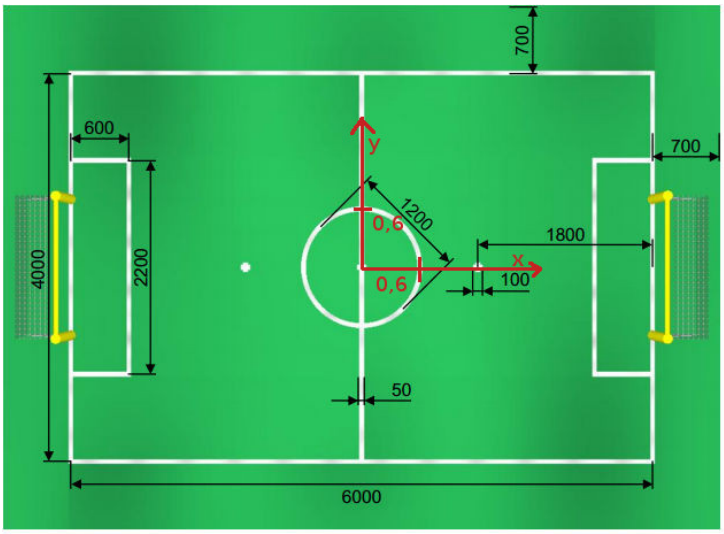


Abbildung 2.5: [Robocub Technical Committee, 2010, S. 1] Das offizielle Spielfeld mit rot eingezeichnetem Welt-Koordinatensystem für den Fall das rechts das Tor des gegnerischen Teams ist.

Ähnlich zu der Middle Size League sind die gleichfarbigen Tore und die Selbstlokalisierung der Roboter anhand der Feldlinien. Da die NAOs jedoch keinen Kompass besitzen und als Zweibeiner häufig stürzen, werden zusätzliche Methoden benötigt um immer die korrekte Position zu finden. Hier setzt diese Bachelorarbeit an. Zuvor soll jedoch die theoretische Basis erläutert werden.

3 Theoretische Basis

In diesem Kapitel sollen einige Grundlagen geklärt werden, welche für die von mir entwickelten Algorithmen benötigt werden. Es werden die verwendeten Koordinatensysteme erklärt, das Model der Kamera, die zugehörigen Koordinatentransformationen, sowie das YCbCr-Model und die vom NAO-Team HTWK Leipzig verwendete Lokalisierung.

3.1 Verwendete Koordinatensysteme

Das NAO-Team HTWK Leipzig hat folgende Koordinatensysteme als internen Standard festgelegt:

3.1.1 Pixel-Koordinaten

Pixel-Koordinaten $PK \subset \mathbb{N}^2$ sind 2-dimensionale Koordinaten $(x, y) \in PK$, denen in einem Bild Farbwerte zugeordnet werden. Hierbei liegt der Ursprung des Koordinatensystems in der linken oberen Ecke eines Bildes (Vgl. Abb.3.1).

3.1.2 Roboter-Koordinaten

Die Roboter-Koordinaten $RK \subset \mathbb{R}^2$ sind relative Koordinaten, welche den Abstand von Dingen bezüglich eines Roboters angeben (Vgl. Abb.3.1).

3.1.3 Welt-Koordinaten

Unter Welt-Koordinaten $WK \subset \mathbb{R}^2$ sind absolute Koordinaten bezüglich des Feld-Mittelpunktes in Richtung des gegnerischen Tors zu verstehen (Vgl. Abb.2.5).

3.2 Die Kamera

Der NAO besitzt im Kopf 2 Kameras. Diese unterstützen jeweils eine Auflösung von 1288×968 Pixel. Ein Bild das eine solche Kamera liefert ist als Abbildung von Pixelkoordinaten nach Farben (siehe Kap. 3.3) definiert: kamerabild : $PK \to F$

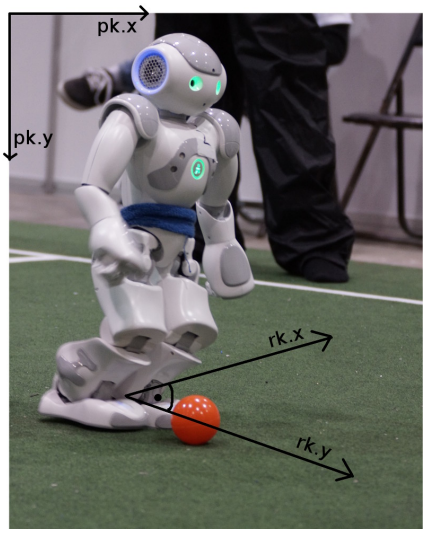


Abbildung 3.1: In dem Bild ist das Pixel-Koordinatensystem, sowie das Roboter-Koordinatensystem eingezeichnet (Punktopterator siehe S. 21)

Zur Zeit wird im NAO-Team HTWK Leipzig, aufgrund des Geschwindigkeitszuwachses bei der Bildverarbeitung von kleineren Auflösungen, jedoch nur 640×480 Pixel verwendet. Um Informationen aus den Kamerabildern verwenden zu können, müssen die Pixel auf Positionen in der "realen Welt" abgebildet werden. Am besten lässt sich dies an einem vereinfachten Kameramodel erklären.

3.2.1 Das Lochkamera-Modell

Ein einfaches Modell einer Kamera ist das "pinhole"-Kameramodel:

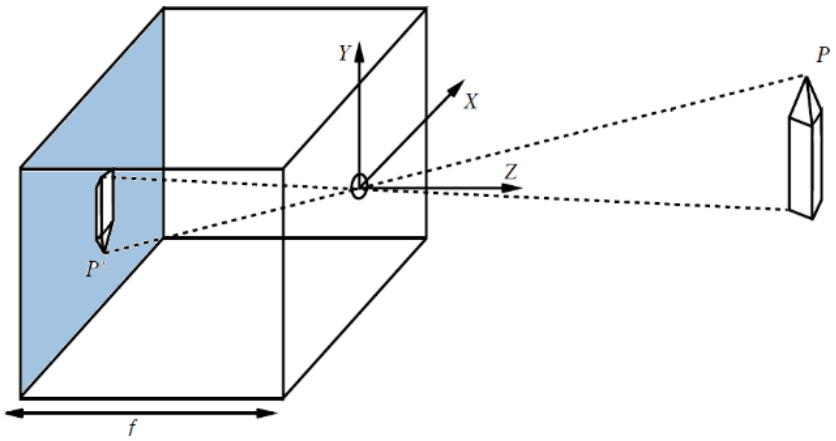


Abbildung 3.2: Skizze einer Lochkamera mit blau eingefärbter Bildebene

Dabei wird ein Punkt (X, Y, Z) der Realen Welt auf den Punkt $\left(x = \frac{X \cdot f}{Z}, y = \frac{Y \cdot f}{Z}\right)$ in der Bildebene abgebildet. Dies lässt sich mit dem Strahlensatz leicht beweisen (Vgl. [Gohout u. Reimer, 2005, S. 120]):

$$\frac{f}{-y} = \frac{Z}{Y} \quad \Leftrightarrow \quad y = \frac{-f \cdot Y}{Z} \tag{3.1}$$

$$\frac{f}{-x} = \frac{Z}{X} \quad \Leftrightarrow \quad x = \frac{-f \cdot X}{Z} \tag{3.2}$$

Dieser Ausdruck lässt sich noch vereinfachen, indem man durch einen "Trick" das Minus weglassen kann:
Statt die Bildebene hinter das Kameraloch zu legen, kann man sich diese einfach zwischen Kameraloch und Objekt wünschen (Abb. 3.3). Dabei steht das abgebildete Objekt nicht mehr auf dem Kopf, das Minuszeichen fällt weg und die anderen Eigenschaften bleiben erhalten (Vgl. [Siciliano u. a., 2010, S. 226,227]).

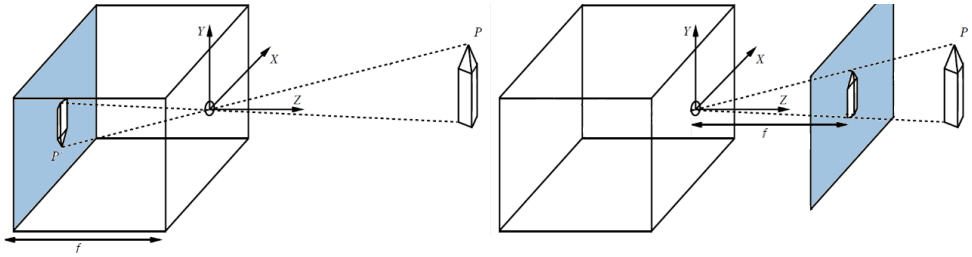


Abbildung 3.3: Veranschaulichung der Bildebene (blau) einmal hinter dem Pinhole und einmal vorne zwischen Pinhole und Objekt dargestellt

Da nun auch häufig von Interesse ist, wo Punkte auf der Bildebene in der realen Welt liegen, muss obiges Verfahren umgekehrt werden. Dafür fehlt allerdings die Tiefe Z, da die Bildebene nur zweidimensional ist. Dadurch lässt sich die Umkehrung nicht ohne weiteres bewerkstelligen. Wenn man das Problem allerdings vereinfacht und unter der Annahme betrachtet, dass sämtliche Bildpunkte nach der Projektion in Welt-Koordinaten auf dem Boden liegen müssen (also $Y = 0$ ist), lässt sich das Problem mit dem Strahlensatz (Vgl. Abb. 3.4) lösen:

1. Erweiterung des Bildpunktes um die 3. Dimension: aus (x, y) wird (x, y, f)
2. Projektion auf den Boden mittels: $\left(X = \frac{x \cdot h}{y}, Y = 0, Z = \frac{f \cdot h}{y}\right)$, wobei h der Abstand von Kameraloch (Pinhole) und Boden ist.

Mit diesem Ansatz lässt sich (unter der Annahme dass die Kamera nicht zusätzlich noch verdreht ist) zum Beispiel der Standort von erkannten Robotern und Torpfosten relativ zur Kamera feststellen. Dazu wird deren Fußpunkt (zum Objekt gehörender

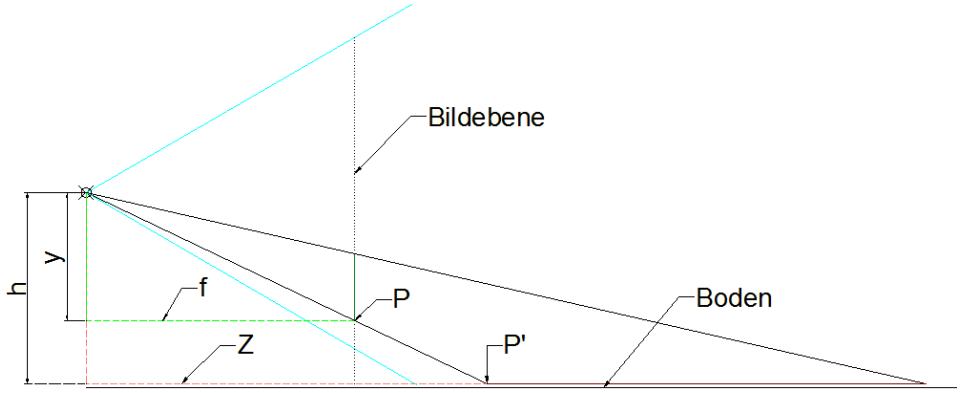


Abbildung 3.4: Projektion auf den Boden

Pixel, welcher direkt auf dem Boden liegt) mit der eben beschriebenen Technik projiziert. Dabei ist zu beachten, dass nur die Fußpunkte verwendet werden dürfen, da nur für sie die obige Annahme, dass diese auf dem Boden liegen, zutrifft.

Für die Bestimmung der notwendigen Parameter (hier f) muss man die Eigenschaften der Kameras des NAOs betrachten.

3.2.2 Die Bildebene

Die Bildebene des NAOs ist 640×480 Pixel groß. Dabei hat ein Kamerabild in der linken oberen Ecke die Koordinate $(0, 0)$, in der rechten unteren Ecke die Koordinate $(640, 480)$ (Vgl. Kap. 3.1.1). Um den Abstand f von Bildebene und Kamera-Pinhole zu berechnen benötigt man das Field of View ($FOV \in \mathbb{R}$). Dieses gibt die Winkel im Gegenstandsraum (der realen Welt) an, die durch die Ränder des Aufnahmeformats begrenzt werden. Die Kameras des NAOs haben einen horizontalen Field of View (FOV_H) von 60.9°. Damit lässt sich mit Hilfe von trigonometrischen Funktionen die Bildweite f bestimmen (Vgl. [Szeliski, 2010, S. 53]):

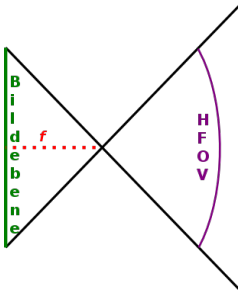


Abbildung 3.5: Draufsicht auf das Modell der Kamera

$$\tan\left(\frac{FOV_H}{2}\right) = \frac{width/2}{f} \quad \Leftrightarrow \quad f = \frac{width/2}{\tan\left(FOV_H/2\right)} \tag{3.3}$$

Dabei ist unter *width* die Breite der Bildebene zu verstehen (hier 640 Pixel).

3.2.3 Transformation von Pixel nach Roboter-Koordinaten unter Berücksichtigung von Verdrehungen der Kamera

Damit auch im richtigen Spiel bestimmt werden kann, wo sich Roboter und andere Dinge befinden, darf die Drehung der Kamera nicht vernachlässigt werden. Diese tritt auf, sobald der Roboter seinen Kopf neigt, oder durch Schwanken bei der Fortbewegung.

Es gibt 3 Achsen um die die Kamera verdreht sein kann,

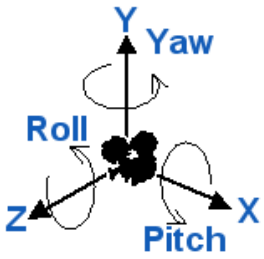


Abbildung 3.6: Drehachsen der Kamera, wobei das Kamera-Icon die Blickrichtung des NAOs angibt

wobei die Drehung um Yaw ignoriert werden kann, indem man diese mit der Drehung des Roboters verrechnet.

Um nun die relative Position von Gegenständen bezüglich der Füße des Roboters zu berechnen, muss die Neigung der Kamera zurückgerechnet werden, so dass die Bildebene wieder senkrecht auf dem Boden steht und deren x-Achse parallel zum Boden ist. Dafür muss sie erst um Roll und dann um Pitch zurückgedreht werden. Dies lässt sich mit Hilfe von Dreh-Matrizen erreichen (Vgl. [Meyberg u. Vachenauer, 2003, S. 317-319]). Für die gesamte Projektion ergeben sich dann folgende Haskell Funktionen:

Listing 3.1: Drehung um Roll

```
1 rotateRoll :: Double -> (Double, Double) -> (Double, Double)
2 rotateRoll roll (x, y) = (rx,ry)
3   where cosRoll = cos roll
4         sinRoll = sin roll
5         rx = x*cosRoll-y*sinRoll
6         ry = x*sinRoll+y*cosRoll
```

Listing 3.2: Drehung um Pitch

```
rotatePitch :: Double -> (Double, Double) -> (Double, Double, Double)
rotatePitch pitch (x, y) = (x, ry , rz)
       where cosPitch = cos pitch
             sinPitch = sin pitch
             ry = cosPitch*f-sinPitch*y
             rz = sinPitch*f+cosPitch*y
```

Listing 3.3: Projektion auf den Boden

```
project :: (Double, Double, Double ) -> (Double, Double)
project (x, y , z) = (x*fac, y*fac)
       where fac = botHeight/z
```

Listing 3.4: Pixel-Koordinaten nach Roboter-Koordinaten

```
data Position = Position{
  pos   :: (Double, Double), yaw :: Double,
  roll  :: Double          , pitch :: Double }

pixelToRoboter :: (Int, Int) -> Position -> (Double, Double)
pixelToRoboter (px, py) (Position _ _ roll pitch) = rk
       where pk = (fromIntegral (px - camWidth `div` 2),
                   fromIntegral ( py - camHeight `div` 2))
             rk = project $ rotatePitch pitch $ rotateRoll roll pk
```

3.2.4 Transformation von Roboter-Koordinaten nach Welt-Koordinaten

Um den Standort von Dingen unabhängig von einem Roboter zu erhalten (also in Welt-Koordinaten), müssen die Roboter-Koordinaten mit der Roboterposition verrechnet werden.

Listing 3.5: Roboter-Koordinaten nach Welt-Koordinaten

```
    (Double, Double) -> Position -> (Double, Double)
    roboterToWorld (rx, ry) (Position (robx, roby) yaw _ _) = (x,
    y) where x = rx * cos (-yaw) - ry * sin (-yaw) + robx y = rx *
    sin (-yaw) + ry * cos (-yaw) + roby
```

3.3 Das YCbCr-Farbmodel

Die Kameras des NAOs nehmen im YCbCr-Format auf. Dabei werden die Farbinformationen, anders als beim RGB-Model nicht auf die drei Grundfarben aufgeteilt, sondern auf die Luminance Y und die zwei Farbkomponenten C_b und C_r (Vgl. Abb. 3.7).
Die Luminance gibt dabei die Grundhelligkeit an. C_b entspricht der Farbabweichung in Richtung Blau-Gelb, C_r steht für die Farbabweichung in Richtung Rot-Türkis.

Jede Komponente kann hierbei einen Wert im Intervall $[0; 255]$ annehmen. Im Übrigen kann eine Farbe $f = (Y, C_b, C_r) \in F$ als Vektor betrachtet werden, für den die Multiplikation mit Skalaren definiert ist. Des Weiteren ist es wichtig für den Algorithmus updateUmgebungsmodel, dass F nicht nur $\subset \mathbb{N}^3$ sondern $\subset \mathbb{R}^3$ ist (siehe Kap. 4.6.1).

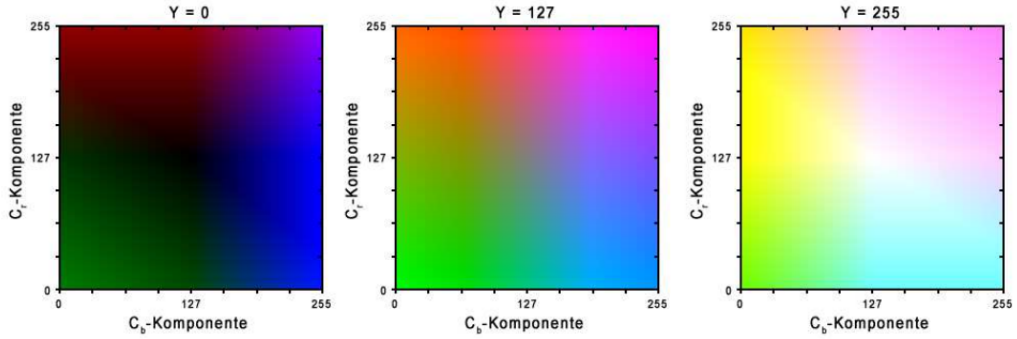


Abbildung 3.7: [Reinhardt, 2011, S. 13] Darstellung der C_b/C_r-Ebene für drei verschiedene Helligkeitsstufen

3.4 Lokalisierung

Die Lokalisierung lokalisierung : kamerabild $\mapsto [P]$, die im NAO-Team HTWK Leipzig bereits verwendet wird, ist Grundlage für die von mir geschriebene Positionsfindung.

Die Lokalisierung berechnet anhand eines Kamerabildes kamerabild Positionen $[P]$, an denen der Roboter möglicherweise steht. Diese möglichen Positionen werden dann durch meine Algorithmen mit Wahrscheinlichkeiten versehen und danach bewertet. Eine Position $p \in P$ ist hierbei ein 6-Tupel: $p = (x, y, roll, pitch, yaw, qual)$. Dabei ist $(x, y) \in WK$ die Weltkoordinate, $(roll, pitch, yaw) \in \mathbb{R}^3$ gibt die Kamerawinkel des Roboters an (Vgl. Abb. 3.6), und $qual \in \mathbb{R}$ die Qualität der Position. Je höher die Qualität einer Position, desto wahrscheinlicher steht der Roboter dort.

In der Lokalisierung wird zuerst das Bild segmentiert um Objekte zu erkennen. Dabei sind Feldlinien und Torpfosten von größter Bedeutung.
Da über das Spielfeld Informationen bekannt sind, wie zum Beispiel Linienbreite und Abstand und Winkel der Linien zueinander, kann man diese benutzen um aus den erkannten Linien auf die Position zu schließen. Dabei wird folgendermaßen vorgegangen:

Die Linien werden so auf den Boden projiziert, dass die Winkel, Abstände und Linienbreiten mit denen des Spielfeld-Models übereinstimmen. Um dann die richtige

Projektion zu finden, benötigt man die Kamerawinkel. Diese werden bisweilen mit einem Hill-Climbing Algorithmus bestimmt:

Von 2 Ausgangswinkeln her (für Roll und Pitch) werden die Linien projiziert, die Abweichung zum Model berechnet, der Ausgangswinkel so lange verändert, dass die Abweichung geringer wird, und dann wieder von vorne begonnen. Wenn die Abweichung zum Model gering ist, sind die Kamerawinkel meist sehr genau bestimmt.

Der nächste Schritt ist die projizierten Linien so auf das Model des Feldes zu legen, dass sie deckungsgleich sind. Erkannte Torpfosten vereinfachen dieses Problem, da sie die Anzahl der möglichen Matches stark reduzieren. Dabei lassen sich die Linien immer auf mindestens zwei Arten über das Model legen. Damit hat man die Kamerawinkel und weiß wo die erkannten Linien auf dem Spielfeld liegen können.

Zuletzt werden die relativen Positionen der erkannten Linien zu dem Roboter mit Hilfe der Kamerawinkel berechnet. Mit diesen lassen sich nun für jede der verschiedenen Annahmen, welche Feldlinien die erkannten Linien sind, eine absolute Roboterposition bestimmen.

Die Lokalisierung berechnet also für ein Kamerabild die möglichen Roboterpositionen, wobei die Zahl dieser schrumpft, je mehr Linien und Torpfosten auf dem Kamerabild erkannt worden sind. Da das Spielfeld symmetrisch ist kommt dabei jedoch immer eine gerade Anzahl von möglichen Positionen zustande, da es für jede Position eine zweite auf der gegenüberliegenden Seite des Spielfeldes gibt.

3.5 Verfügbare Daten

Damit und mit der Software des NAO-Teams HTWK Leipzig werden folgende, für die Nutzung der Umgebung als Orientierungshilfe relevanten, Daten zur Verfügung gestellt:

- Die Kamerabilder (640×480 Pixel), um Informationen aus der Umgebung zu extrahieren.
- Die Lokalisierung, welche aus dem Kamerabild 0, 2 oder mehrere mögliche Positionen bestimmt, an denen sich der Roboter befinden könnte.
- Die Odometrie, welche anhand von Bewegungsinformationen ausgehend von einer Position, eine Folgeposition berechnet.
- Die Farbe des Spielfeldes, welche für jedes Bild neu bestimmt wird. Sie wird unter anderem für die Lokalisierung benötigt und bei der Segmentierung der Bilder bestimmt (Vgl. [Reinhardt, 2011, S. 31-47]).

- istFeldfarbe : $F \rightarrow \mathit{BOOLEAN}$ ist eine Abbildung die für einen übergebenen Farbwert $f \in F$ bestimmt, ob er als Feldfarbe angesehen werden kann. Dies kann für mehr als ein $f \in F$ zutreffen.
- Ein Algorithmus erkenneFeldgrenze : kamerabild $\mapsto \mathit{EFG}$ mit $\mathit{EFG} \subset \mathit{PK}$ welcher die Begrenzungen des Spielfeldes in einem Kamerabild berechnet (Vgl. [Reinhardt, 2011, S. 62-70]). Dabei ist EFG die Menge der Pixel-Koordinaten eines Kamerabildes, an deren Stelle die Feldbegrenzung im Kamerabild ist (siehe weiße Linie in Abb. 4.3).
- Die letzte Position an der der Roboter denkt, dass er sich befand. Mit dieser und der Odometrie wird bestimmt, welche der aktuell möglichen Positionen die vermeintlich Richtige ist.
- Eine Abstands-Funktion abstandPos : $(p_1, p_2) \mapsto dis$, welche die zurückgelegte Distanz dis von einer Position $p_1 \in P$ zu einer anderen Position $p_2 \in P$ modelliert. Dabei haben unwahrscheinliche, bzw untypische Bewegungen von p_1 nach p_2, wie Rückwärtsbewegungen, eine höhere Distanz, als die für den Roboter typischen Bewegungen, wie zum Beispiel in einer Kurve nach schräg vorne laufen.
- Eine Funktion pixelToRobot : $(\mathit{PK}, P) \rightarrow \mathit{RK}$, welche eine Pixel-Koordinate $pk \in \mathit{PK}$ in eine relative Koordinate $rk \in \mathit{RK}$ bezüglich einer Roboterposition $p \in P$ umrechnet.
- Eine Funktion robotToWorld : $(\mathit{RK}, P) \rightarrow \mathit{WK}$, welche eine zur Roboterposition $p \in P$ relative Koordinate RK in eine Welt-Koordinate $wk \in \mathit{WK}$ umrechnet.

4 Orientierung mit Hilfe der Umgebung

In diesem Kapitel wird nach einem kurzen Überblick über andere Orientierungsverfahren das von mir verwendete Umgebungsmodel und die Algorithmen spezifiziert. Bei den Algorithmen wird die Beschreibung durch Tests ergänzt.

4.1 Andere Ansätze zur Orientierung

Warum sollte man ausgerechnet die Umgebung zur Orientierung benutzen? Um diese Frage zu beantworten, werden im Folgenden zuerst verschieden Möglichkeiten aufgelistet, welche möglicherweise ebenfalls für die Problemlösung geeignet wären und deren Vor- und Nachteile beleuchtet:

4.1.1 Teamball

Der Ball bringt eine Asymmetrie ins Spiel, die man benutzen könnte. Wenn ein Roboter, im Vergleich zu seinen Mitstreitern, den Ball auf der gespiegelten Position sieht, ist dieser vermutlich delokalisiert. Bei einem gut modellierten Teamball, ist dies eine einfache und recht sichere Möglichkeit der Positionsfindung. Wenn allerdings nur 2 Spieler im Spiel sind, ist der Teamball für die Orientierung nutzlos, da nicht mehr entschieden werden kann, wer delokalisiert ist. Und bei einem Roboter gibt es gar keinen Teamball (Vgl. [Brose, 2008]).

4.1.2 Torwart

Da der Torwart meist im eigenen Tor steht und daher kaum die Möglichkeit hat sich zu delokalisieren, kann dieser Signale versenden, welche wiederum die anderen Roboter zur Lokalisierung benutzen können. Dabei bieten sich unter anderem Ton- und Infrarotsignale an. Der große Nachteil ist offensichtlich: Alles hängt vom Torwart ab. Wenn dieser wegen Inaktivität oder Verlassen des Spielfeldes herausgenommen wird, kann kein Spieler mehr seine Signale benutzen. Im schlimmsten Fall wechselt der Torwart selbst die Seite, zum Beispiel beim heraus dribbeln des Balles aus dem Tor, was zur Folge hat, dass das restliche Team mitzieht und auch die Seite wechselt.

4.1.3 W-LAN

Eine weitere Möglichkeit der Positionsfindung ist, die W-LAN Stärke zu benutzen. Dies wäre unabhängig von der Anzahl der Roboter möglich. Der große Nachteil daran ist, dass bei Veranstaltungen wie dem RoboCup, es aufgrund der zahlreichen Netze stets W-LAN Probleme gibt, was die Verwendung dieses Verfahrens meist nicht möglich macht.

4.1.4 Resümee

All diesen Verfahren gegenübergestellt, hat die Orientierung anhand der Umgebung Vorteile. Sie ist unabhängig von der Anzahl der Roboter, der WLAN-Stärke oder dem Vorhandensein des Torwarts. Im Zusammenspiel mit der Odometrie ermöglicht sie eine äußerst sichere Lokalisierung, welche durch Nutzung des Teamballs noch weiter verbessert werden kann.

4.2 Übersicht über die Algorithmen

Um die Umgebung, also Farbinformationen außerhalb des grünen Spielteppichs, als Orientierungshilfe zu verwenden, müssen zuerst bei stabiler Lokalisierung Informationen über die Umgebung gesammelt werden. Diese werden mit Algorithmus 1 aus den Kamerabildern extrahiert und in ein Model der Umgebung gespeichert.

Algorithmus 2 verrechnet dabei jeweils das letzte bekannte Umgebungsmodel mit dem des aktuellen Kamerabildes zu einem Gesamt-Model.

Sobald das Gesamt-Model Informationen gespeichert hat, wird dieses verwendet um etwas über die Korrektheit von Roboterposition auszusagen: Algorithmus 3 berechnet für jede Position, die die Lokalisierung liefert, ein Umgebungsmodel und vergleicht dieses mit dem Gesamt-Model. Positionen deren Modelle besser mit dem Gesamt-Model übereinstimmen werden dabei besser bewertet.

Die Bewertungen werden mit Algorithmus 4 über die Zeit hinweg betrachtet. Falls dabei eine starke Tendenz in Richtung einer anderen Position p_2 herauskommt, wird die aktuelle Position p_1 verworfen und der Roboter nimmt p_2 als korrekt an. Sonst wird weiterhin p_1, welche von der Odometrie bestimmt wurde, benutzt.
Algorithmus 4 dient vor allem dazu die Odometrie und die hier vorgestellten Algorithmen gleichzeitig verwendbar zu machen.

In Abb. 4.1 ist das Zusammenspiel der Algorithmen veranschaulicht. Da dies ein iterativer Prozess ist, werden teilweise Ausgaben im nächsten Iterationsschritt als

Eingaben verwendet. Solch geartete Zusammenhänge sind in der Abbildung mit blauen Pfeilen gekennzeichnet.

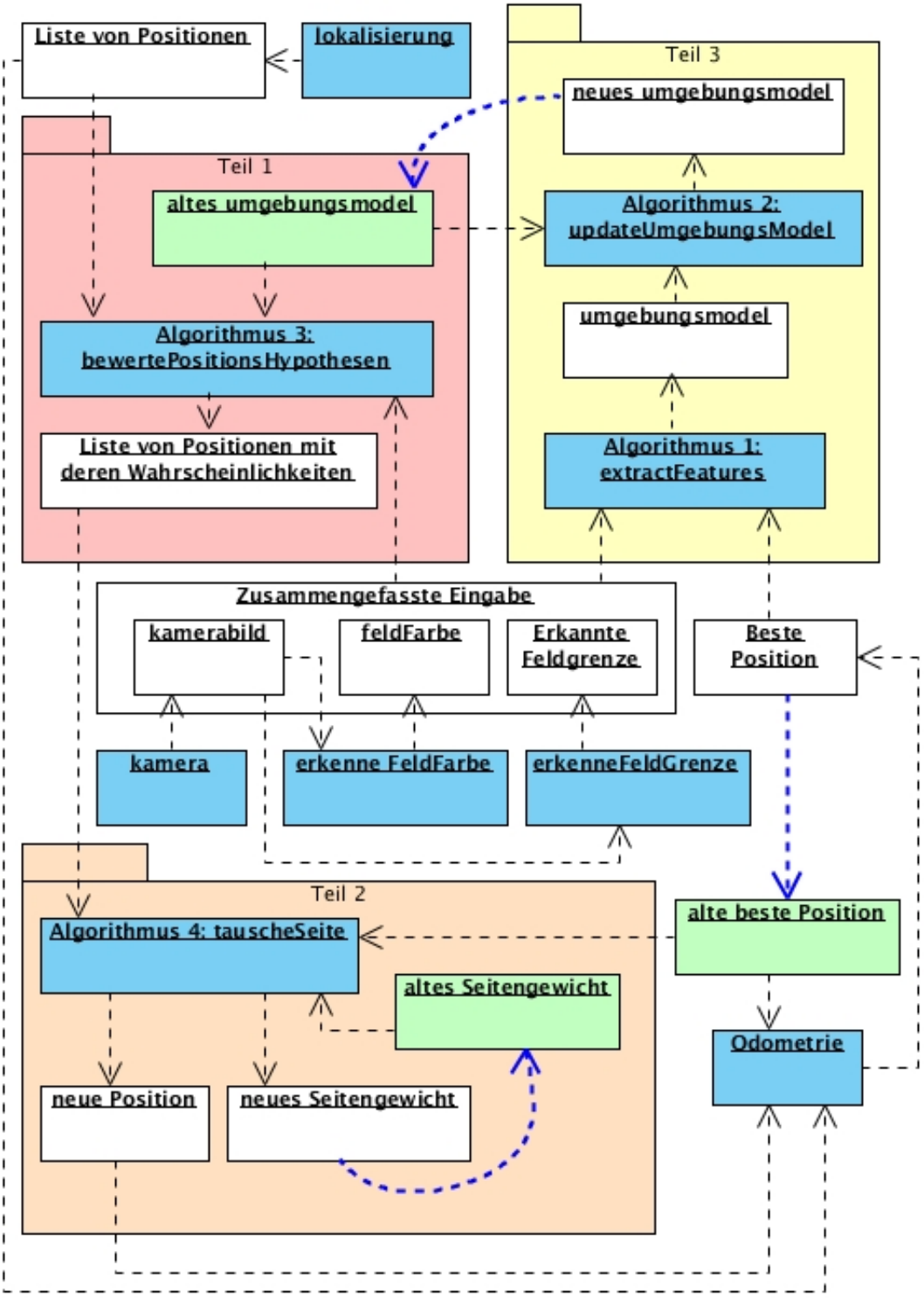


Abbildung 4.1: Die Abbildung zeigt den Ablauf der Algorithmen

4.3 Das Umgebungsmodel

Das Umgebungsmodel ist die Grundlage der von mir entwickelten Algorithmen. Es wird im Folgendem erklärt.

4.3.1 Spezifikation

Um ein Umgebungsmodel zu definieren wird zunächst die Linie zwischen dem grünen Spielteppich und der Umwelt (Feldgrenze) in Randbereiche *RB* unterteilt. Die *RB* sind also abstrakte Stücke der Feldgrenze, die durch eine fortlaufende Randbereichsnummer $rn \in RN \subset \mathbb{N}$ gekennzeichnet sind.
Dann ordnet die Funktion getCoord : $RN \to BK$ den Nummern der *RB* die Koordinaten $BK \subseteq WK$ der Mittelpunkte der jeweiligen *RB* zu.

Ein Umgebungsmodel umgebungsmodel : $RN \hookrightarrow EM$ ist eine partielle Abbildung von den Nummern der Randbereiche auf Erkennungsmerkmale. Es ist zu klären was wir unter einer partiellen Abbildung und einem Erkennungsmerkmal verstehen wollen.

Zunächst die partielle Abbildung $\hookrightarrow$:
Eine partielle Abbildung $f : A \hookrightarrow B$ ist eine Relation $Rel \subseteq A \times B$, deren Definitionsbereich dom folgendermaßen definiert ist: $\operatorname{dom} f = \{x \mid \exists y : (x, y) \in f\}$
Für f gilt:

$$f(x) = \begin{cases} y, & \text{falls } x \in \operatorname{dom} f \\ \perp & \text{sonst} \end{cases},$$

Dabei ist unter $\perp$ die leere Ausgabe zu verstehen.

Was ist nun unter einem Erkennungsmerkmal *em* zu verstehen? Es gilt:
$em = (f, counter) \in EM$, wobei $f \in F$ und $counter \in \mathbb{N}$ ist.

4.3.2 Wahl der Unterteilung

Die Anzahl der Randbereiche *RB* gibt vor wie genau das Model sein kann: Je größer sie ist, desto genauer kann das Model die Umgebung abbilden.

Da das Kamerabild allerdings eine nicht allzu hohe Auflösung hat (640×480 Pixel), machen zu feine Unterteilungen nur bedingt Sinn: Wenn eine zu feine Unterteilung gewählt wird, so sind die Informationen, welche das Kamerabild liefert, ungenauer als das Model, wodurch das Model nur teilweise gefüllt werden könnte.

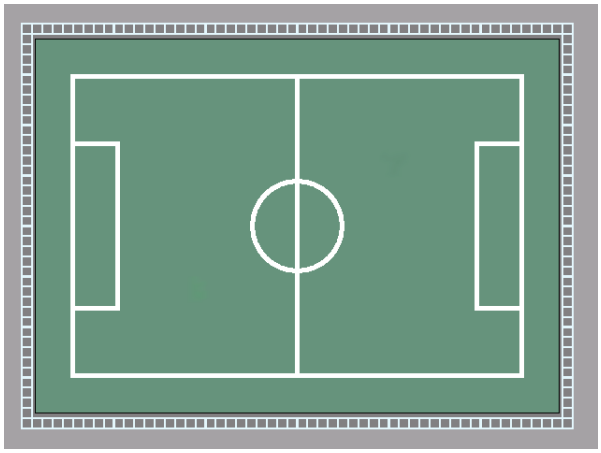


Abbildung 4.2: Die Abbildung zeigt eine Skizze des Spielfeldes mit der Unterteilung der Feldgrenze in Randbereiche

Andererseits führt eine zu grobe Unterteilung zu einem ungenauen Model der Umgebung: Unterschiedliche Informationen, welche in der realen Welt nahe zusammenliegen, würden, nach deren Extraktion, demselben Randgebiet zugeordnet.
Die optimale Unterteilung wäre also, dass möglichst jede aus einem Kamerabild extrahierte Information in eine eigene Randregion des Umgebungsmodels fällt und zugleich Lücken zwischen den "gefüllten" Randbereiche vermieden werden. Im Kapitel 4.5.5 wird eine mögliche Unterteilung, welche diese Richtlinien erfüllt, bestimmt.

Nachdem das Umgebungsmodel festgelegt ist, sollen nun die verwendeten Algorithmen spezifiziert werden.

4.4 Spezifikation der Algorithmentypen

Bevor sich die nächsten Kapitel der Umsetzung der Algorithmen widmen, werden hier die Typen festgelegt.

4.4.1 Algorithmus 1: extractFeatures

extractFeatures(p, kamerabild, istFeldfarbe, *EFG*) = umgebungsmodel,
wobei $p \in P$

4.4.2 Algorithmus 2: updateUmgebungsmodel

updateUmgebungsmodel$(u_1, u_2) = u_3$,
wobei u_1, u_2 und u_3 vom Typ umgebungsmodel sind. Dabei gilt:

$$\operatorname{dom} u_3 = \operatorname{dom} u_1 \cup \operatorname{dom} u_2$$

Für die x die Element von beiden Definitionsbereichen $\operatorname{dom} u_1$ und $\operatorname{dom} u_2$ sind, gilt: $u_3(x) = \text{verrechnen}(u_1(x), u_2(x))$, wobei verrechnen in Kapitel 4.6.1 erläutert wird. Für alle anderen x hat u_3 folgendes Ergebnis:

$$u_3(x) = \begin{cases} u_1(x) & \text{falls } x \in \operatorname{dom} u_1 \\ u_2(x) & \text{falls } x \in \operatorname{dom} u_2 \\ \bot & \text{sonst} \end{cases}$$

4.4.3 Algorithmus 3: bewertePositionsHypothesen

$$\begin{aligned} &\text{bewertePositionsHypothesen}([P], \text{umgebungsmodel}, \\ &\qquad \text{kamerabild}, \text{istFeldfarbe}, \mathit{EFG}) = [(P, B)] \end{aligned}$$

Dabei ist $B \in \mathbb{R}$ eine Bewertung.

4.4.4 Algorithmus 4: tauscheSeite

tauscheSeite$(w_1, w_2, p_a, sw_a) = (sw_n, p_n)$, wobei:
w_1, $w_2 \in \{(P, \text{Wahrscheinlichkeit } wk \in \mathbb{R} \text{ von } P)\}$; p_a, $p_n \in P$; sw_a, $sw_n \in \mathit{SW}$,
und es gilt: $(w_1.p = p_n \vee w_2.p = p_n)$

Der . Operator steht hierbei für den Zugriff auf eine Komponente. Beispiel:
Falls $w_1 = ((2,3,4,3,5,7),1)$ ist, wäre $w_1.wk = 1$, $w_1.p = (2,3,4,3,5,7)$ und $w_1.p.x = 2$.

4.5 extractFeatures

Damit der Roboter die Spielfeldumgebung als Orientierungshilfe nutzen kann, müssen aus den Kamera-Bildern markante Daten über die Umgebung gewonnen werden und in das Umgebungsmodel gespeichert werden.

4.5.1 Extraktion der relevanten Daten aus den Kamera-Bildern

Für die Entscheidung ob ein Pixel $pk \in PK$ im Kamerabild als Information über die Spielfeldumgebung dienen kann, muss entschieden werden ob er außerhalb des Spielfeldteppichs liegt. Dafür wird die durch unsere Software bereits erkannte Feldgrenze des grünen Spielteppichs (*EFG*) genutzt (siehe Abb. 4.3). Für die Menge der relevanten Pixel $RELPK \subset PK$ gilt:
$\forall relpk \in RELPK : \exists efg \in EFG : relpk.y < efg.y \wedge relpk.x = efg.x$

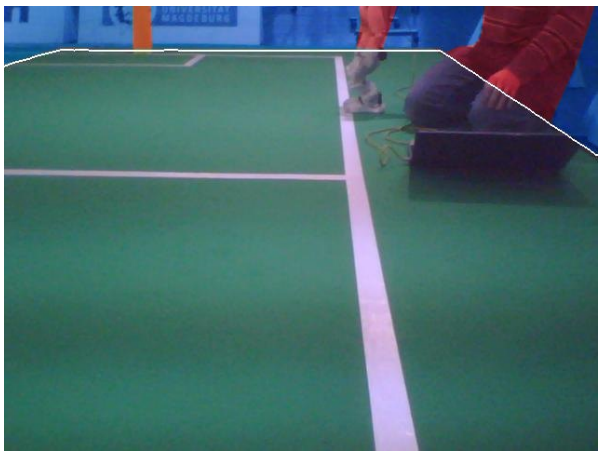


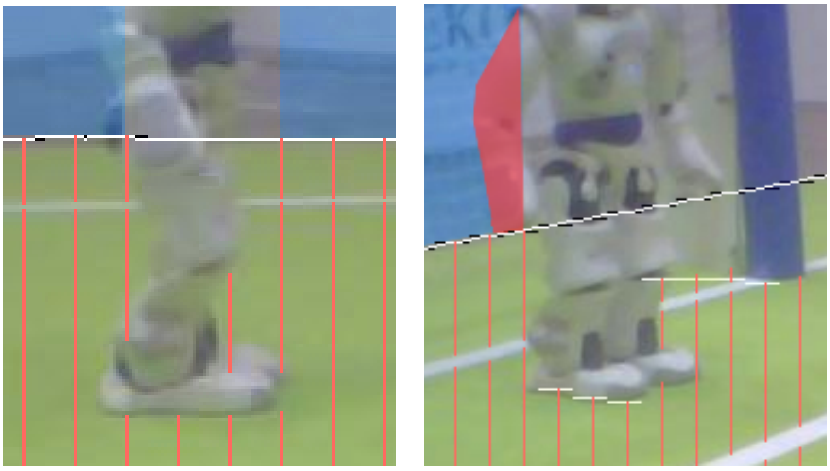
Abbildung 4.3: Die relevanten Pixel sind blau eingefärbt und die erkannte Feldgrenze *EFG* durch eine weiße Linie gekennzeichnet

Wie man jedoch auch auf Abb. 4.3 sieht, kann es passieren, dass temporäre Hindernisse, wie andere Roboter oder Schiedsrichter, die Eigenschaften der für die Orientierung relevanten Pixel *RELPK* bestimmen, und so das Ergebnis verfälschen (dunkelrot gekennzeichnet). Um dies zu verhindern, muss überprüft werden, ob der Pixel $pu \in PK$, der senkrecht unter dem betrachteten Pixel und zudem direkt unter dem erkannten Rand des Spielteppichs liegt, grün ist:

$\forall relpk \in RELPK : \exists efg \in EFG : relpk.y < efg.y \wedge relpk.x = efg.x \wedge$
$\text{istFeldfarbe}(\text{kamerabild}(efg.x, efg.y + 1)) = true$

Wenn diese Bedingung erfüllt ist, sind die relevanten Pixel *RELPK* mit hoher Wahrscheinlichkeit nicht Teil eines temporären Hindernisses (siehe Abb. 4.4(a)).

Falls das Hindernis jedoch zu nahe am Rand steht, kann es sein, dass obwohl die Bedingung erfüllt ist, freischwebende Arme oder andere Dinge mit einfließen (siehe Abb. 4.4(b)).



(a) (b)

Abbildung 4.4: Das Erkannte grüne Spielfeld ist mit roten Linien gekennzeichnet. Nur jede 16te Bildspalte wurde betrachtet. Die *RELPK* sind blau gekennzeichnet

4.5.2 Zuordnung auf die Randbereiche des Umgebungsmodels

Nun wollen wir mit bereichsfindung den aus dem Bild erkannten Pixelkoordinaten der Feldgrenze und einer zugehörigen Roboterposition eine Nummer eines Randbereichs zuordnen:
bereichsfindung : $(EFG, P) \rightarrow RN$.

Diese berechnet zuerst für den übergebenen Pixel $efg \in EFG$ dessen Weltkoordinate $wk \in WK$:
robotToWorld(pixelToRobot$(efg, p), p) = wk$, wobei $p \in P$. Solche Projektionen sind in Abb. 4.5 dargestellt.

Abbildung 4.5: Randpixel und deren Projektion nach Welt-Koordinaten

Der berechneten Weltkoordinate wk wird nun die Nummer $rn' \in RN$ des Randbereichs mit dem kleinsten Abstand zugeordnet. Dies geschieht mittels
zuordnung : $WK \rightarrow RN$. Dabei gilt:

$$\forall rn \in RN : \text{abstand}(wk, \text{getCoord}(rn)) \geq \text{abstand}(wk, \text{getCoord}(\text{zuordnung}(wk)))$$

4.5.3 Fehlererkennung durch Projektion

Nun kann es durch vorhandene Ungenauigkeiten der Lokalisierung und der Feldlinienerkennung zu einer falschen Erkennung der Feldgrenze kommen. Um diese Fälle auszusondern, wird folgender Plausibilitätstest durchgeführt:
Die auf Welt-Koordinaten abgebildeten Pixelkoordinaten der erkannten Feldgrenze *EFG* dürfen einen bestimmten Abstand a von den Welt-Koordinaten der tatsächlichen Feldgrenze nicht überschreiten (siehe Abb. 4.6). Dadurch wird aus der Funktion bereichsfindung eine partielle Funktion: bereichsfindung : $(EFG, P) \hookrightarrow RN$
Diese liefert nur ein Ergebnis, falls:
$\text{abstand}(wk, \text{getCoord}(\text{zuordnung}(wk))) < a$ ist.

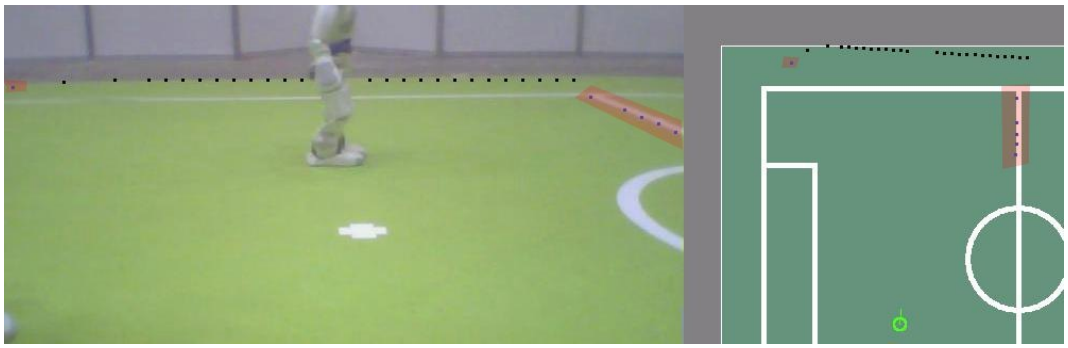


Abbildung 4.6: Die blau-rot gekennzeichneten Pixel liegen nach der Projektion in Welt-Koordinaten innerhalb des Spielfeldes und werden deswegen aussortiert

4.5.4 Erkennungsmerkmale

Damit sind sowohl die relevanten Pixelkoordinaten *RELPK* eines Bildes, als auch die Zugehörigkeit der Pixelkoordinaten der erkannten Feldgrenze zu einer Randbereichs-Nummer *RN* bekannt. Jetzt bleibt die Frage offen, wie aus dem Kamerabild mittels den relevanten Pixelkoordinaten *RELPK* Merkmale gewonnen werden können.

Dafür müssen wir zunächst festlegen, was wir unter einem Erkennungsmerkmal verstehen wollen:

Als Erkennungsmerkmal, das einen Randbereich charakterisiert, bietet sich ein Histogramm an, das die Farbgegebenheiten der Region oberhalb von diesem speichert (Vgl. [Kangaroos u. a., 2012, S. 4]). Dies würde allerdings eine große Menge an Speicherplatz benötigen, was den geplanten Austausch der Daten zwischen den Robotern problematisch machen würde. Da zudem für die Extraktion und Verarbeitung ein hoher Rechenaufwand nötig ist, jedoch eine echtzeitfähige Lösung gesucht ist, muss etwas anderes in Betracht gezogen werden.
Die Idee ist, ein Umgebungsmodel zu erstellen, welches zu den Randbereichen eine einzige charakteristische Farbe, mit einer Gewichtung speichert. Es gilt also für EM (Menge der Erkennungsmerkmale): $EM \subseteq (F, \mathbb{N})$
Die Erkennungsmerkmale können dann zur Bewertung der Wahrscheinlichkeit verschiedener Roboterpositionen P verwendet werden. Eine einzige für einen Bereich charakteristische Farbe, statt die Häufigkeiten der Farben, als Merkmal zu verwenden, verringert die Rechenzeit deutlich. Außerdem kann dadurch der benötigte Speicherplatz erheblich reduziert werden und auch das Versenden der gespeicherten Informationen zwischen den Robotern ist durch die kleine Datenmenge einfach zu realisieren.

Dabei wird die Farbe eines Erkennungsmerkmal $em \in EM$ aus dem komponentenweisen Median im Kamerabild übereinander liegenden Farben gebildet:
$em.f = \text{median}(\text{kamerabild}(\mathit{RELPK}'))$, mit:
$\exists t \in \mathbb{R} : \forall relpk' \in \mathit{RELPK}' : relpk'.x = t$
Hier wird kamerabild eine Menge von Pixel-Koordinaten gegeben, statt nur eine Pixel-Koordinate, weswegen hier kamerabild auch eine Menge von Farben liefert.
Für die Gewichtung der Erkennungsmerkmale gilt dabei immer: $em.counter = 1$

Bestimmung des Umgebungsmodels

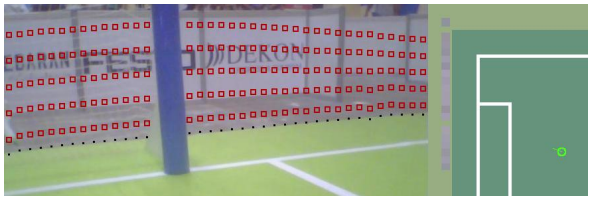
Die partielle Abbildung umgebungsmodel setzt sich nun so zusammen:
Oberhalb aller $efg \in EFG$, für die bereichsfindung ein Ergebnis rn liefert, wird das Erkennungsmerkmal em extrahiert. umgebungsmodel bildet nun rn auf em ab.

Um extractFeatures zu beschleunigen, werden 2 zusätzliche Bedingungen aufgestellt:

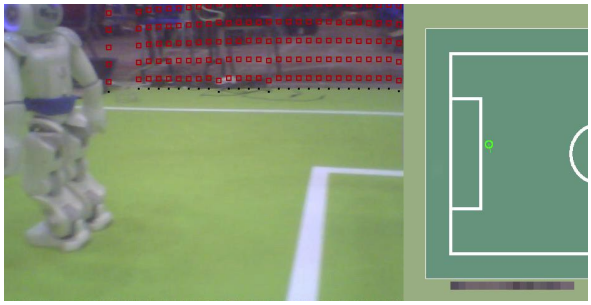
1. Es wird nur jede 16te Bildspalte betrachtet, so dass bereichsfindung nur ein Ergebnis liefert für:
   $\forall efg \in EFG : efg.x \,\text{modulo}\, 16 = 0$
2. erkennungsmerkmal(kamerabild(RELPK')) $= em$ wählt 5 Pixel aus seiner Eingabe aus und berechnet nur anhand von diesen den komponentenweisen Me-

dian. Dieser wird mit einem $counter = 1$ kombiniert als $em \in EM$ zurückgegeben.

Auf Abb. 4.7 lassen sich die Unterschiede in der Umgebung nach der Komprimierung zu Erkennungsmerkmalen und Zuordnung zu den Randbereichen weiterhin gut erkennen: In Abb. 4.7(a) ist der Hintergrund gräulich und hell, und auch die im rechten Teil des Bildes visualisierten Erkennungsmerkmale teilen diese Eigenschaft. Hingegen ist in Abb. 4.7(b) die Umgebung, wie auch die gewonnenen Erkennungsmerkmale, dunkler und bräunlich.



(a)



(b)

Abbildung 4.7: Die 5 Pixel welche zur Erkennungsmerkmals-Bildung verwendet werden sind jeweils rot gekennzeichnet. Die Erkennungsmerkmale werden den Randbereichen zugeordnet und sind im rechten Teil der Bilder visualisiert.

Bestimmung des Abstandes zwischen den Pixeln

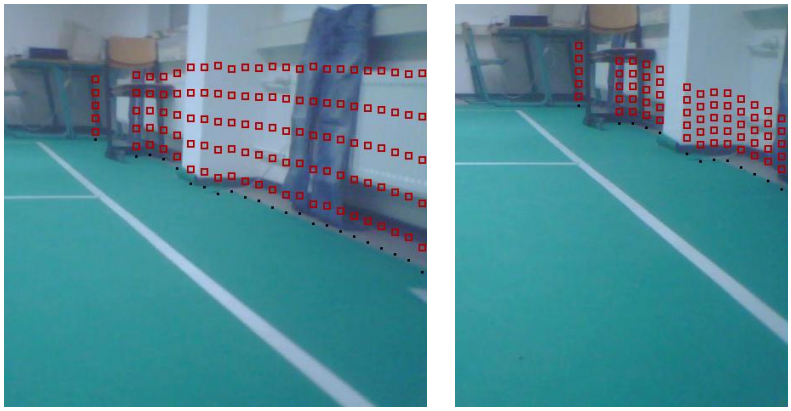
Der Abstand zwischen den 5 Pixeln ist, wie in 4.7(a) zu sehen, nicht konstant. Wenn dies so wäre, würde ein Erkennungsmerkmal, je nach Abstand des Roboters zum Rand, von Pixeln unterschiedlicher Höhe gebildet werden. Wenn der Roboter nahe am gesehenen Spielfeldrand steht, würde viel weniger weit nach oben nach relevanten Pixel gesucht werden, als wenn er weiter weg vom Rand steht. Somit würden die Informationen sehr stark von dem Standort des Roboters abhängen, wodurch eine Verwendung dieser dann auch nur abhängig von der Roboterposition geschehen könnte. Dies habe ich vermieden, indem der Abstand 5 Pixel in Abhängigkeit von der Entfernung zwischen Roboter und Randbereich gewählt wurde:

Listing 4.1: getPoints

```
1      for(int i = 0; i<5; i++ ){
2        int nextY = (int)(P.y-i*(abstand*fac));
3        p[i]=getColor(image, P.x, nextY);
4      }
```

Dabei ist $P \in EFG$ ein Punkt direkt auf der erkannten Feldgrenze, *abstand* der Abstand vom Roboter zur gesehenen Randregion, $p[i]$ der $i$te von den 5 gesuchten Pixeln und fac eine Konstante, die es geschickt zu wählen gilt.

Die 2 Bilder in Abb. 4.8 veranschaulichen nochmal den Unterschied zwischen konstantem und variablem Abstand.



(a) variabler Abstand (b) konstanter Abstand

Abbildung 4.8: In (a) werden die Daten bis knapp unter dem Fensterbrett extrahiert, vollkommen unabhängig von der Entfernung. In (b) hingegen wird, in der Nähe des Roboters, das Bild nur bis halber Höhe betrachtet.

4.5.5 Wahl der Konstanten und Tests

Anzahl der Randregionen

Wie in Kapitel 4.3 beschrieben, ist die Unterteilung der Umgebung am besten so zu wählen, dass die aus den Kamerabild extrahierten Informationen in eigene, naheliegende Randregionen des Umgebungsmodels fallen. Damit alle gewonnen Features des Kamerabildes in eine eigene Randregion fallen können, muss die auf den Bild zu sehende Umgebung also mindestens in gleichviele Regionen aufgeteilt werden. Wenn bei einer Auflösung von 640×480 aus jeder 16ten Spalte ein Feature gewonnen wird, ergeben sich 40 Merkmale pro Bild. Da in einem Kamerabild meistens eine Seite des Spielfeldes komplett zu sehen ist, bietet es sich an die Seiten je in ca. 40 Regionen zu unterteilen. Um dabei gleichgroße Regionen zu erhalten, sind die Seitenverhältnisse des Spielfeldes zu beachten: Bei einem Verhältnis von 7 zu 5 und der Unterteilung der

kurzen Seiten in je 40 Regionen ergibt sich für die langen Seiten eine Unterteilung in je 56 Regionen.

Es bleibt allerdings die Frage offen, was geschehen soll, wenn in einem Bild doch mehrere Mediane in die gleiche Randregion fallen, also die Auflösung des Bildes feiner ist, als die Unterteilung der Umgebung in Randregionen. Es gibt nun verschiedene Möglichkeiten mit diesem Problem umzugehen. Einerseits könnte nur der Erste gefundene Median verwendet werden, andererseits könnte man auch den Durchschnitt aller, in die gleiche Randregion fallenden, Mediane bilden und ihn als charakteristische Farbe verwenden. Dadurch würde, bei starkem Farbunterschied innerhalb der gleichen Region, ein Mittel verwendet, welches im Gegensatz zum einfachen Median die Randregion nicht so einseitig darstellt. Bei den gewählten Parametern kommt dies allerdings nur selten vor (siehe Tabelle 4.1).

Tabelle 4.1: Die Daten wurden mit einer Folge von 890 Bildern erstellt

Anzahl der Regionen		... Mediane, fallen in die gleiche Region							
lange Kante	kurze Kante	1	2	3	4	5	6	7	8
60	84	15181	6148	736	90	39	10	3	0
50	70	11543	6675	1469	159	49	17	5	0
40	56	7869	5889	2709	473	67	26	9	2
30	42	5086	4486	2844	1451	306	41	12	6
20	28	2665	2662	1978	1582	995	605	143	31

Auswertung

Die Tests ergaben, dass ab 50×70 Regionen eine starke Abnahme der Fälle gibt, bei denen mehr als 2 Mediane in eine Randregion fallen. Deswegen werden im Folgenden alle Daten in einem Umgebungsmodel gespeichert, welches den Rand in 50×70 Regionen unterteilt.

Weiterhin zeigten die Tests das es vereinzelt vorkommt, dass viele Mediane in die gleiche Region fallen. Bei der genaueren Betrachtung dieser Fälle ist aufgefallen, dass der Roboter immer delokalisiert war und die dadurch verursachte falsche Projektion Auslöser für die gehäuften Mediane pro Region war. Als Folge dessen wurde eine Grenze eingeführt, welche besagt dass wenn mehr als 4 Mediane in eine Randregion fallen, die Informationen nicht weiter betrachtet werden sollen, da sie mit hoher Wahrscheinlichkeit auf einer falschen Positions-Annahme basieren.

4.6 updateUmgebungsmodel

Für die Aktualisierung eines bestehenden Umgebungsmodels und seiner Erkennungsmerkmale ist updateUmgebungsmodel zuständig. Dies ist nötig, um möglichst viele, sowie aktuelle Erkennungsmerkmale in dem bestehenden Umgebungsmodel zu haben. Nur so kann der Roboter dieses als Orientierungshilfe verwenden.
Der Algorithmus bekommt das alte Model u_1 und ein neues u_2. Als Ergebnis liefert er ein weiteres Umgebungsmodel u_3, welches zu exakt den Bereichs-Nummern Informationen gespeichert hat, zu denen u_1 und/oder u_2 auch Informationen hatte. Falls nur in einem der beiden Modelle eine Zuordnung von einer Randbereichs-Nummer zu einem Erkennungsmerkmal existiert, wird diese übernommen. Für die Nummern der Bereiche zu denen u_1 und u_2 jeweils ein Erkennungsmerkmal liefern, müssen diese 2 Erkennungsmerkmale für u_3 zu einem neuen mittels verrechnen verrechnet werden.

$$u_3(x) = \begin{cases} u_1(x) & \text{falls } x \in \operatorname{dom} u_1 \wedge x \notin \operatorname{dom} u_2 \\ u_2(x) & \text{falls } x \notin \operatorname{dom} u_1 \wedge x \in \operatorname{dom} u_2 \\ \text{verrechnen}(u_1(x), u_2(x)) & \text{falls } x \in \operatorname{dom} u_1 \wedge x \in \operatorname{dom} u_2 \\ \bot & \text{sonst} \end{cases}$$

Für verrechnen wird folgender Algorithmus verwendet:

4.6.1 Verrechnen der Merkmale

Für verrechnen$(em_1, em_2) \mapsto em_3$, mit em_1, em_2 und $em_3 \in EM$, wird das "Farbtopf-Prinzip" verwendet:
Man stelle sich einen Topf t vor, der maximal $t.x$ Tropfen Farbe fasst, und mit $em_1.counter$ Tropfen der Farbe $em_1.f$ gefüllt ist, wobei $em_1.counter < t.x$. Nun tropft $em_2.counter$ mal die Farbe $em_2.f$ in den Topf.
Falls $em_1.counter + em_2.counter \leq t.x$ sind, ergibt sich für den Inhalt des Topfes t:

$$\begin{aligned} t.counter &= em_1.counter + em_2.counter \\ t.f &= \frac{em_1.counter * em_1.farbe + em_2.counter * em_2.farbe}{t.counter} \end{aligned}$$

Falls jedoch $em_1.counter + em_2.counter > x$ ist und dabei der Topf noch nicht voll ist, also $t.counter < t.x$ gilt, geht der Teil Farbe von $em_2.f$ verloren, der nicht mehr in den Topf passt. In diesem Fall ergibt sich für den Inhalt des Topfes t:

$$\begin{aligned} t.counter &= t.x \\ t.f \qquad &= \frac{em_1.counter * em_1.farbe + (t.x - em_1.counter) * em_2.farbe}{t.x} \end{aligned}$$

Wenn der Topf jedoch bereits mit em_1 gefüllt ist, also $em_1.counter = t.x$, geht fast ganz em_2 verloren, so dass für den Inhalt des Topfes t diesmal gilt:

$$\begin{aligned} t.counter &= t.x \\ t.f \qquad &= \frac{(t.x - 1) * em_1.farbe + em_2.farbe}{t.x} \end{aligned}$$

Also kann sich die Farbe $t.f$ eines nicht vollen Topfes t umso schneller ändern, je größer der Topf ist. Mit einem großen Topf wird gewährleistet, das anfänglich falsche Farbinformationen ausgeglichen werden. Erst wenn eine solide Basis von $t.x$ Informationen vorhanden ist, wird der Farbtopf langsamer aktualisiert. Damit auch Änderungen im Nachkommabereich von $t.f$ möglich sind, wurden die Farben F als $\subset \mathbb{R}^3$ definiert und nicht wie sonst üblich als $\subset \mathbb{N}^3$.

Das Ergebnis em_3 von verrechnen ist somit der anfänglich leere Farbtopf t, der zuerst mit em_1 befüllt und dem darauf em_2 zugegeben worden ist.

4.6.2 Andere Möglichkeiten der Verrechnung

Andere Algorithmen wie Durchschnittsbildung oder gewichtete Durchschnittsbildung für verrechnen zu verwenden bietet sich nicht an, da sie Nachteile haben:

- Durchschnittsbildung: Dabei wird em_3 zu gleichen teilen aus em_1 und em_2 gebildet. Der Nachteil dabei ist, dass das Umgebungsmodel welches geupdated wird, ständigen starken Schwankungen ausgesetzt wird. Falls das zum Update verwendete Model Fehler enthält, also die Zuordnung von Bereichsnummern zu Erkennungsmerkmalen nicht der Welt “entspricht”, wirken sich diese sofort und stark auf das Ergebnis aus.
- Gewichtete Durchschnittsbildung: $em_3 = em_1 * f + em_2 * (1 - f)$ mit $f \in [0..1]$ Hierbei werden je nach Faktor f die Erkennungsmerkmale unterschiedlich gewichtet. Bei $f = 0.5$ erhält man wieder die Durchschnittsbildung. Damit die Fehler im Model sich nicht so schnell auswirken, muss $f > 0.5$ gewählt werden. Damit fließt em_2 nur noch langsam ein. Bei einen Faktor $f = 0.99$ ergibt sich,

das em_3 erst nach rund 70 Aktualisierungen (b) zu gleichen Teilen aus em_1 und em_2 besteht, also $x * f^{70} + y * (1-f)^{70} \approx x/2 + y/2$ gilt:

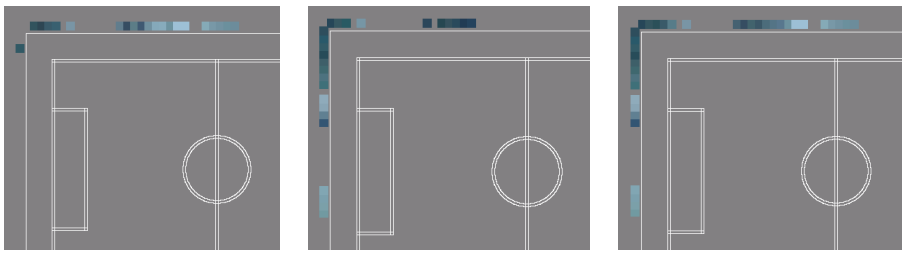
$$\begin{aligned} x \cdot f^b &= \frac{x}{2} & \Leftrightarrow \quad f^b &= \frac{1}{2} \\ \Leftrightarrow \ln\left(f^b\right) &= \ln\left(\frac{1}{2}\right) & \Leftrightarrow \quad b \cdot \ln(f) &= \ln\left(\frac{1}{2}\right) \\ \Leftrightarrow b &= \frac{\ln\left(\frac{1}{2}\right)}{\ln(f)} \quad \text{und damit für } f = 0.99\text{:} & \frac{\ln\left(\frac{1}{2}\right)}{\ln(0.99)} &\approx 70 \end{aligned}$$

Der große Nachteil der gewichteten Durchschnittsbildung ist, dass wenn em_1 am Anfang fehlerhaft war, dieser Fehler nur sehr langsam korrigiert wird (siehe Kapitel 4.6.3).

Das verwendete Farbtopf-Prinzip hingegen vereint die Vorteile beider Algorithmen und ist damit das Mittel der Wahl.

4.6.3 Auswertung und Tests

Nach Anwendung des Algorithmus updateUmgebungsmodel auf 2 Umgebungsmodelle entsteht ein neues (Vgl. Abb. 4.9).



(a) U_a (b) U_n (c) U_u

Abbildung 4.9: Umgebungsmodel U_u als Ergebnis von updateUmgebungsmodel mit den 2 Modellen U_a und U_n als Eingabe

Bei regelmäßigem Update des Models ist die Umgebung um das komplette Spielfeld herum gespeichert. Auf Abbildung 4.10 ist ein Umgebungsmodel zu sehen, dass bei einer Folge von ca. 900 Bildern entstanden ist.

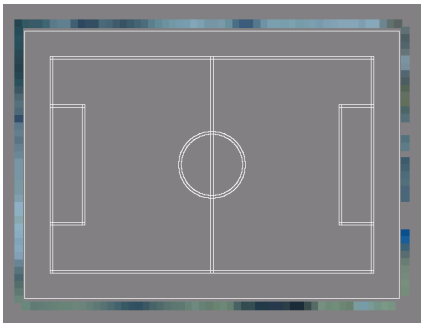
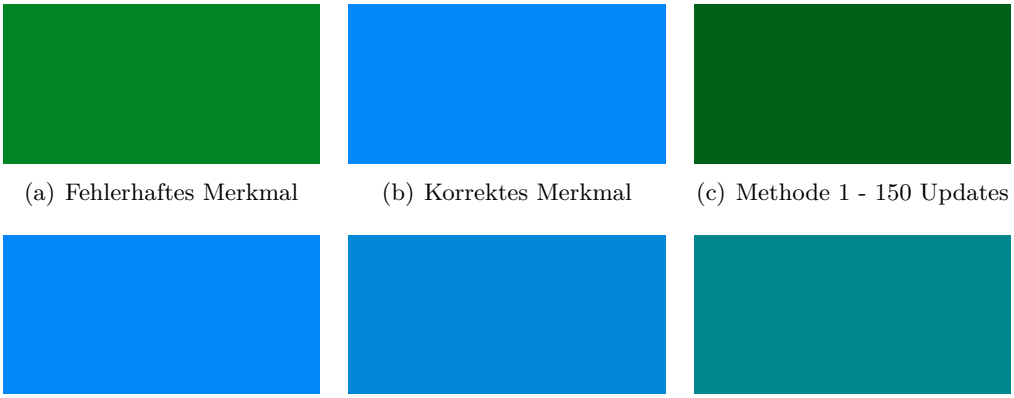


Abbildung 4.10: Ein Umgebungsmodel, welches Informationen über fast alle Randregionen gespeichert hält

In der Abbildung 4.11 soll noch einmal die unterschiedliche Entwicklung des Umgebungsmodels, bei Anwendung der verschiedenen Methoden gezeigt werden, für den Fall, dass der Erste gewonnene Wert fehlerhaft war.



(a) Fehlerhaftes Merkmal (b) Korrektes Merkmal (c) Methode 1 - 150 Updates

(d) Methode 2 - 150 Updates (e) Methode 2 - 5 Updates (f) Methode 2 - 1 Update

Abbildung 4.11: Vergleich von Methode 1: gewichtete Durchschnittsbildung und Methode 2: Das Farbtopf-Prinzip

Hierbei zeigt Abb. 4.11(a) ein Erkennungsmerkmal welches fälschlicherweise vom grünen Spielteppich statt aus der Umgebung gewonnen wurde, weil die Rand-Erkennung falsch war. Abbildung 4.11(b) zeigt wie das Merkmal eigentlich aussehen sollte - hier blau, vielleicht von einem bläulichen Werbeschild. Abbildung 4.11(c) bis 4.11(f) zeigen wie sich das fehlerhafte Merkmal über die Zeit hinweg verändert. Hierbei wurde für die Updates das korrekte Bild verwendet. Bei Methode 1 lässt sich sehr gut erkennen, dass selbst nach 150 Updates, also nach über 3 Sekunden das Merkmal immer noch mehr grün als blau ist. Bei dem Farbtopf-Prinzip, lässt sich jedoch bereits nach einem Update eine deutliche Verbesserung erkennen und nach 5 Updates ist der Fehler kaum mehr zu erkennen.

4.6.4 Wahl von Parametern: Die Größe des Farbtopfes

Die richtige Wahl der Größe des Farbtopfes kommt stark auf die Anzahl der verarbeiteten Bilder pro Sekunde an. Je schneller die Bilder verarbeitet werden, desto größer muss dieser sein, damit die gespeicherten Werte sich nicht zu schnell zu stark ändern. Bei einer Größe von 255 ensteht eine solide Basis, welche aus dem Durchschnitt von 255 gesehenen Werten besteht und welche sich daraufhin nur langsam mit dem Faktor $254/255 = 0.99608$ ändert. Dieser sagt aus, dass die Basis, bei 30 Bildern pro Sekunde, nach ca. 6s nur noch zur Hälfte in den aktuellen Wert einfließt (Vgl. Kapitel 4.6.2).

4.6.5 Zusätzliche Bedingungen für den Aufruf von updateUmgebungsmodel

Um zu verhindern, dass das gespeicherte Umgebungsmodel u_1 mit einem neuen "falschen" Umgebungsmodel u_2 aktualisiert wird, darf updateUmgebungsmodel(u_1, u_2) nur ausgeführt werden, wenn für die Roboterposition p, für die u_2 mit extractFeatures$(p, \cdots) = u_2$ bestimmt worden ist, zusätzliche Bedingungen erfüllt sind:

- $p.qual > 0.9$: Die Qualität der Position muss sehr hoch sein.
- falls die Position p mit bewertePositionsHypothesen bewertet worden ist, muss die dazugehörige Bewertung b größer als 0.5 sein.
- $sw \geq 0$: Die Tendenz geht nicht in Richtung der anderen Seite (Vgl. Kap. 4.8).

Hiermit ist gewährleistet, dass nur Umgebungsmodelle verwendet werden, bei denen relativ sicher ist das sie von der Position aus gewonnen worden sind, auf der sich der Roboter auch tatsächlich befand.

4.7 bewertePositionsHypothesen

Nachdem die Roboter nun mit dem Umgebungsmodel eine Vorstellung über ihre Umwelt haben, können sie das Model benutzen um mögliche Positionen zu bewerten. Die dafür in Frage kommenden Positionen vom Typ P werden durch die Lokalisierung mittels der Feldlinien berechnet (Vgl. Kapitel 3.4).
Für jede dieser Positionen kann mit extractFeatures ein umgebungsmodel u_p berechnet werden, dass das Gesehene auf die Umgebung bezüglich der vermuteten Position $p \in P$ abbildet. Diese Umgebungsmodelle können dann mit dem bereits vorhandenen umgebungsmodel u_v mit vergleichen$(u_p, u_v) = r_p$ verglichen werden.

Als Ergebnis wird den möglichen Positionen eine Wahrscheinlichkeit $r_p \in \mathbb{R}$ ihrer Korrektheit zugeordnet.

Für vergleichen(u_p, u_v), kann auf Algorithmen zurückgegriffen werden, welche für die Bestimmung der Ähnlichkeit von zwei Bildern verwendet werden.
Dafür muss nur der Input $i \in RN$ von umgebungsmodel als Pixelkoordinate k einer eindimensionalen Bildfunktion bild aufgefasst werden und der Output $em \in EM$ von umgebungsmodel als Farbe $f \in F$ interpretiert werden. Das heißt $em.counter$ wird nicht beachtet. Dabei gilt für die Bildfunktion: bild$(k) = f$, wobei $k \in \mathbb{N}$ und $f \in F$ ist.

Für vergleichen wird der Korrelationskoeffizient (siehe Kap. 4.7.4) verwendet. Dabei liefert vergleichen(u_p, u_v) nur ein Ergebnis, falls u_v für alle r ein Ergebnis liefert, für dass auch u_p ein Ergebnis liefert.
Im den folgenden Abschnitten werden die Algorithmen erläutert, auf denen der Korrelationskoeffizient basiert.

4.7.1 Der N-dimensionale euklidische Abstand

Die Summe d_E der quadratischen Abstände, auch N-dimensionaler euklidischer Abstand genannt, eignet sich gut als Maß für den Abstand zwischen zweidimensionalen Bildfunktionen $I(u, v)$ und $R(u, v)$.

$$d_E(r, s) = \sqrt{\sum_{(i,j)\in R} (I(r + i, s + j) - R(i, j))^2} \tag{4.1}$$

Um die beste Übereinstimmung zwischen dem Referenzbild $R(u, v)$ und dem Zielbild $I(u, v)$ zu finden, reicht es, da die Wurzel-Funktion streng monoton steigend ist, das Quadrat von d_E zu minimieren:

$$\begin{aligned} d_E(r, s) &= \sum_{(i,j)\in R} (I(r + i, s + j) - R(i, j))^2 \\ &= \underbrace{\sum_{(i,j)\in R} (I(r + i, s + j))^2}_{A(r,s)} + \underbrace{\sum_{(i,j)\in R} (R(i, j))^2}_{B} - 2 \underbrace{\sum_{(i,j)\in R} I(r + i, s + j) \cdot R(i, j)}_{C(r,s)} \end{aligned} \tag{4.2}$$

Da der Ausdruck B in (4.2) eine von r, s unabhängige Konstante ist, muss er zum minimieren nicht betrachtet werden. $A(r, s)$ ist die quadratische Summe der Werte des entsprechenden Bildausschnitts in I, auch "Signal-Energie" genannt. $C(r, s)$ entspricht der linearen Kreuzkorrelation zwischen I und R[6].

[6] Vgl. [Burger u. Burge, 2005, S. 427-428]

4.7.2 Lineare Kreuzkorrelation

Wenn $A(r,s)$ in (4.2) innerhalb des Bildes I weitgehend konstant ist, dann stimmt I mit dem Offset (r,s), an dem die lineare Kreuzkorrelation

$$C(r,s) = \sum_{(i,j)\in R} I(r+i, s+j) \cdot R(i,j) \tag{4.3}$$

ihren maximal Wert erreicht, mit dem Referenzbild R am besten überein[7].

4.7.3 Normalisierte Kreuzkorrelation

Da in der Praxis die Annahme, dass $A(r,s)$ über das Bild hinweg konstant ist, meist nicht zu trifft, muss diese Abhängigkeit kompensiert werden. Dies macht die normalisierte Kreuzkorrelation C_N, welche die Energie des aktuellen Bildausschnittes berücksichtigt:

$$C_N(r,s) = \frac{C(r,s)}{\sqrt{A(r,s)\cdot B}} = \frac{C(r,s)}{\sqrt{A(r,s)}\cdot\sqrt{B}}$$

$$= \frac{\sum\limits_{(i,j)\in R} I(r+i, s+j)\cdot R(i,j)}{\sqrt{\sum\limits_{(i,j)\in R}(I(r+i,s+j)^2}\cdot\sqrt{\sum\limits_{(i,j)\in R}(R(i,j))^2}} \tag{4.4}$$

Weisen die Bilder ausschließlich positive Werte auf, dann ist das Ergebnis $C_N(r,s))$ immer im Bereich $[0,1]$. Ein Wert $C_N(r,s) = 1$ steht für eine maximale Übereinstimmung zwischen R und I bei einem Offset (r,s). Ein weiterer Vorteil der normalisierten Korrelation ist, dass sie ein standardisiertes Maß für den Grad der Übereinstimmung liefert, und deshalb direkt für die Entscheidung der Akzeptanz der entsprechenden Position verwendet werden kann.

Die Normierung in (4.4) gibt also im Unterschied zu Gl. (4.3) ein lokales Abstandsmaß an. Allerdings hat sie noch das Problem, das die absolute Distanz zwischen den Bildern gemessen wird. Wenn sich beispielsweise die Gesamthelligkeit des Bildes I erhöht, wird sich auch das Ergebnis der normalisierten Korrelation $C_N(r,s)$ deutlich verändern[8].

[7] Vgl. [Burger u. Burge, 2005, S. 428]
[8] Vgl. [Burger u. Burge, 2005, S. 428-429]

4.7.4 Korrelationskoeffizient

Statt der absoluten Distanz, bietet es sich an die Differenz in Bezug auf die lokalen Durchschnittswerte in R ($\bar{R}$) und die des zugehörigen Bildausschnitts von I ($\bar{I}$) zu vergleichen. Man subtrahiert also $\bar{J}$ und $\bar{R}$ und erhält Gl. (4.4).

$$C_L(r,s) = \frac{\sum\limits_{(i,j)\in R} \Big(I(r+i,s+j) - \bar{I}(r,s)\Big) \cdot \Big(R(i,j) - \bar{R}\Big)}{\sqrt{\sum\limits_{(i,j)\in R} \Big(I(r+i,s+j) - \bar{I}(r,s)\Big)^2} \cdot \sqrt{\sum\limits_{(i,j)\in R} \Big(R(i,j) - \bar{R}\Big)^2}}\,, \tag{4.5}$$

wobei die Durchschnittswerte $\bar{I}(r,s)$ und $\bar{R}$ definiert sind als

$$\bar{I}(r,s) = \frac{1}{K} \sum_{(i,j)\in R} I(r+i,s+j) \qquad\qquad \bar{R} = \frac{1}{K} \sum_{(i,j)\in R} R(i,j) \tag{4.6}$$

und $K = |R|$, also die Anzahl der Elemente in R ist. Die Ergebniswerte von $C_L(r,s)$ liegen im Unterschied zur normalisierten Kreuzkorrelation im Intervall $[-1,1]$, wobei der Wert 1 der maximale Übereinstimmung und -1 der maximalen Abweichung entspricht[9].

4.7.5 Vergleiche der unterschiedlichen Verfahren anhand von Tests

Lineare Kreuzkorrelation

Die Lineare Kreuzkorrelation liefert an sich keine brauchbare Ergebnisse:
Die Ergebnisse für die verschiedenen möglichen Positionen unterscheiden sich nur selten stark. Bei Insgesamt 10 Testsequenzen je 1000 Bilder, ist der Unterschied zwischen der am besten und der am schlechtesten bewerteten Position im Schnitt bei 0.05. Dabei wurden die Ergebniswerte durch ihre Summe geteilt, damit diese zwischen 0 und 1 liegen und besser zu vergleichen sind. Durch diesen geringen Unterschied lässt sich nur schwer eine Aussage treffen, welche von den Positionen die vermeintlich Richtige ist.

Normalisierte Kreuzkorrelation

Durch die Normalisierung liegen die Ergebnisse der Normalisierten Kreuzkorrelation bereits zwischen 0 und 1, so dass die Ergebnisse gut miteinander auch über mehrere Bilder hinweg verglichen werden können.
Doch auch hier ist der Unterschied zwischen den einzelnen Positionen verschwindend

[9] Vgl. [Burger u. Burge, 2005, S. 429-430]

gering: Bei den gleichen Testdaten ist dieser im Schnitt sogar nur bei 0.025. Deshalb sind auch hier die Ergebnisse verschiedener Bilder nicht gut zu vergleichen.

Korrelationskoeffizient

Mit dem Korrelationskoeffizienten lässt sich ohne weitere Normierung der Unterschied auf ganze 0.9 erhöhen! Hierbei sei dazu gesagt, dass die Ergebnisse auch in einem größeren Bereich (von -1 bis 1) liegen. Mit diesen Werten ist es einfach die vermeintlich beste Position auszuwählen.

Hierbei stellt sich jedoch noch die Frage ob die Ergebnisse auch korrekt sind:
Wird die richtige Position auch besser bewertet?

Überprüfung der Ergebnisse

Da sich die Korrektheit der Ergebnisse nur schwer beweisen lässt, wurde für die Überprüfung der Ergebnisse eine Test-Datenbank erstellt, welche von einem NAO aufgenommene Bildsequenzen mit deren Aufnahme-Positionen enthält.
Mittels eines, für diese Arbeit entwickelten, Testprogrammes wurden dann alle Bildsequenzen durchlaufen. Das Testprogramm führt zuerst die Lokalisierung, wie unter Kapitel 3.4 beschrieben, durch um danach die verschiedenen Positionen mit den Algorithmen 1-4 zu bewerten:
Hierbei war (falls eine Bewertung der Positionen überhaupt möglich war), bei dem Vergleich des gespeicherten Umgebungsmodels mit dem aus dem aktuellen Kamerabild gewonnenen, zu 92% die korrekte Position auch am besten bewertet. Die Fehler der 8%, die falsch bewertet wurden, waren einerseits durch falsche Ausgangsdaten verursacht worden, welche die Lokalisierung lieferte, andererseits durch Ungenauigkeiten der entwickelten Algorithmen. Eine Ungenauigkeit wäre zum Beispiel die Projektion der Merkmale auf den Boden oder auch die komprimierte Speicherung der Merkmale im Umgebungsmodel als Mediane.

4.8 tauscheSeite

Mit bewertePositionsHypothesen erhält man für jede mögliche Position, welche die Lokalisierung berechnet hat (durch den Vergleich der Umgebungsmodelle), eine Wahrscheinlichkeit. Diese kann zusätzlich zur Odometrie verwendet werden um möglichst erfolgreich die tatsächliche Position herauszufinden. Da im richtigen Spiel der Roboter sehr viele Bilder pro Sekunde verarbeitet, könnten Fehlentscheidungen den Roboter häufig zu kurzzeitigem Wechseln der Seite bringen. Um dies zu verhindern

müssen die Ergebnisse von bewertePositionsHypothesen über die Zeit hinweg betrachtet werden. Erst wenn die Tendenz über längere Zeit hinweg in Richtung einer anderen Position geht, soll die Jetzige verworfen werden und stattdessen die Andere benutzt werden.

4.8.1 Umsetzung

Die Speicherung und Auswertung der Wahrscheinlichkeiten der möglichen Positionen über die Zeit hinweg übernimmt tauscheSeite.
Im Allgemeinen bietet es sich an mit bewertePositionsHypothesen$([P], \cdots) = [(P, B)]$ nur dann die Positionen auszuwerten, wenn genau 2 Positionen gefunden wurden, also $[P].size = 2$ gilt. Einerseits kommt dies bei der Bildverarbeitungsgeschwindigkeit von 30 Bildern pro Sekunde sehr häufig vor, andererseits vereinfacht es das Problem ungemein:

Um nun nicht ständig bei Fehlerkennnung zwischen 2 Position zu toggeln, wird mit tauscheSeite : $(w_1, w_2, p_a, sw_a) \mapsto (sw_n, p_n)$ ein Filter implementiert:
Es wird ein Schwellwert $S \in \mathbb{R}$ definiert, welcher unterschritten werden muss, um die von der Odometrie (siehe Kap. 3.5) ausgewählte Position zu verwerfen und stattdessen die andere zu verwenden. Dafür wird zusätzlich zu den Wahrscheinlichkeiten auf Basis der Umgebungs-Informationen, noch die Nähe zur letzten gewählten Position p_a mittels abstandPos : $(P, P) \to \mathbb{R}$ berechnet (siehe Kap. 3.5). Die Wahrscheinlichkeit der zu p_a nächsten Position, wird auf den Wert sw_a hinzuaddiert und die Wahrscheinlichkeit der anderen Position subtrahiert, um somit sw_n zu erhalten. Wenn die Position, welche näher an p_a liegt die höhere Wahrscheinlichkeit besitzt, dann ist $sw_n > sw_a$. Wenn andererseits deren Wahrscheinlichkeit geringer ist, als die der weiter weg liegenden Position, schrumpft der Wert, so dass $sw_n < sw_a$. Bei Unterschreiten des festgelegten Schwellwertes, also $sw_n < S$, hat in den letzen Positionsbestimmungen die weiter weg liegende Position gehäuft eine höhere Wahrscheinlichkeit gehabt. Dies wiederum heißt, dass der Roboter sich mit hoher Sicherheit delokalisiert hat und sich eigentlich auf der anderen Position befindet.

Deswegen wird daraufhin die Position des Roboters gespiegelt und sw_n wieder auf 0 zurückgesetzt. Damit ergibt sich für sw_n und p_n:

$$
\begin{aligned}
a_1 &= \text{abstandPos}(w_1.p, p_a) \\
a_2 &= \text{abstandPos}(w_2.p, p_a) \\
sw'_n &= \begin{cases} sw_a + w_1.wk - w_2.wk & \text{falls } a_1 < a_2 \\ sw_a - w_1.wk + w_2.wk & \text{falls } a_1 > a_2 \\ sw_n & \text{sonst} \end{cases} \\
sw_n &= \begin{cases} sw'_n & \text{falls } sw'_n \geq S \\ 0 & \text{falls } sw'_n < S \end{cases} \\
p_n &= \begin{cases} w_1.p & \text{falls } a1 < a2 \wedge sw'_n \geq S \vee a1 > a2 \wedge sw'_n < S \\ w_2.p & \text{sonst} \end{cases}
\end{aligned}
$$

Um zu verhindern, dass sw_n nicht zu groß wird, solange der Roboter seine Position korrekt erkannt hat, lässt man ihn mit der Zeit schrumpfen, zum Beispiel durch Multiplikation mit einem Faktor $F \in]0, 1[$. Dadurch bleibt der Wert in einem kleinen Rahmen, und hat damit die Möglichkeit bei Delokalisierung auch den Schwellwert S in kurzer Zeit zu unterschreiten.

4.8.2 Tests

Ohne Verwendung von tauscheSeite hatte der Roboter sich alle paar Sekunden kurzzeitig für die gespiegelte Position entschieden und seine Laufrichtung geändert. Erst mit tauscheSeite wurde dieses Problem behoben.
Um dies zu testen, wurde ein Roboter so programmiert, dass er ständig zu der selben Position läuft. Wenn er diese das erste Mal erreicht hat, wird er von Hand um seine eigene Achse gedreht um das intern gespeicherte Umgebungsmodel aufzubauen. Für den eigentlichen Test wird der Roboter dann auf verschiedene Positionen auf dem Spielfeld gesetzt. Dann wird beobachtet, ob und nach welcher Zeit er wieder auf die gegebene Position zurückläuft, oder aber ob er sich für die gegenüberliegende, falsche, Position entscheidet (Vgl. Tabelle 4.2).

Durch die Wahl der richtigen Parameter traten in den Testfällen keine falschen Positionswechsel mehr auf.
Je größer S gewählt worden ist, desto schneller trat ein Wechsel bei falscher Positionierung auf. Bei zu großem S wechselte der Roboter allerdings auch manchmal die Seite, während er auf der richtigen Position stand (siehe Testfall 15).
F hingegen hält sw nahe 0. Ein zu kleiner Faktor F bewirkt dass sw sich kaum

Tabelle 4.2: Durchführung von Tests um passende Werte für S und F herauszufinden

Testfall	S	F	Zeit	Kommentar
1	-10	0.8	-	fast keine Veränderung von sw
4	-10	0.9999	-	Abbruch da sw anfangs zu groß geworden ist
8	-10	0.995	8	in den Tests blieb $sw < 20$
11	-7	0.995	6	Wechsel dauert zulange
15	-1	0.995	< 1	manchmal zur falschen Seite gewechselt
17	-1.5	0.995	2	Wechsel klappt gut

verändert, und so nie S unterschreitet. Ein zu großes F lässt allerdings sw beliebig groß werden, solange der Roboter auf der korrekten Seite steht.
Bei der richtigen Wahl von S und F bleibt sw relativ nahe an 0, kann dabei aber auch schnell kleiner als S werden. Dadurch wechselt der Roboter schnell und zuverlässig die Seite, wenn er "merkt" das er sich auf der falschen Seite befindet.

4.9 Verwendung im Spiel

Das bis hierher vorgestellte Verfahren wurde ausgiebig in Spielen der Weltmeisterschaft 2012 in Mexico-City getestet:

Anfangs wurde es in einigen Testspielen verwendet. Dabei kamen mehrere kleine Programmierfehler ans Licht, welche sich schnell beheben ließen.
Daraufhin wurde das Verfahren erfolgreich in mehreren Meisterschaftsspielen eingesetzt, wie auch im Viertelfinale gegen B-Human. In diesen Spielen traten fast keine Orientierungsverluste mehr auf. In den wenigen Fällen in denen ein Roboter dennoch die Orientierung verlor, ließ es sich auf ein noch fast leeres Umgebungsmodel zurückführen, da es immer kurz nach nach dem Einlaufen passierte.

Um diese Fälle zu verhindern, muss das Umgebungsmodel schneller aufgebaut werden. Dafür bietet es sich an die Umgebungsmodelle zwischen den Robotern zu teilen. Vor allem beim Einlaufen der Roboter, wenn die Umgebungsmodelle noch recht leer sind, werden diese schneller gefüllt:
Die Roboter laufen von beiden Seiten des Spielfeldes ein und können Informationen über unterschiedliche Randbereiche sammeln und austauschen. Das Teilen der Umgebungsmodelle untereinander wird im folgendem Kapitel behandelt.

5 Sharing der Daten

Um möglichst schnell an aktuelle Informationen über den kompletten Rand zu gelangen, bietet es sich an, die Umgebungsmodelle zwischen den Robotern zu teilen. Je vollständiger das Umgebungsmodel eines Roboters ist, desto besser kann er sich orientieren. Das Model zu füllen, kann je nach Spieler unterschiedlich lange dauern. Der Torwart zum Beispiel, blickt meistens nur in Richtung gegnerisches Tor, so dass er nicht weiß wie es hinter ihm aussieht und darüber keine Informationen gespeichert hat.
Für den Austausch der Umgebungsmodelle zwischen den Robotern kommen zwei Möglichkeiten in Frage:
Entweder der komplette Austausch des gespeicherten Umgebungsmodels, oder nur das Versenden der neu gesehenen Modelle.

5.1 Verschicken der neu gesehenen Randmerkmale

Ein Vorteil daran nur die neu gesehenen Umgebungsmodelle bei allen Robotern zum Update des gespeicherten Models zu verwenden, ist, dass die Verrechnung der neuen Daten bei allen gleich ablaufen kann:
Es wird dafür einfach updateUmgebungsmodel verwendet. Zudem wird die Größe des zu versendenden Datenpakets klein gehalten. Allerdings ist zu beachten, das die WLAN Verbindung der NAOs sehr störungsanfällig ist. So pendelte die Ping-Zeit während der German Open 2012 meist zwischen 10ms und 13 Sekunden. Dadurch kommen die versendete Pakete in anderer Reihenfolge an, und die Informationen die jeder Roboter über den Rand hat können sehr unterschiedlich sein. Ein noch größeres Problem ist wenn (wie während der Weltmeisterschaft 2012 in Mexico) so gut wie keine über WLAN verschickten Pakete ankommen, denn dann gehen die gesendeten Informationen für die anderen Roboter endgültig verloren. Dies wäre bei dem regelmäßigen Versenden aller gespeicherten Informationen nicht der Fall.

5.2 Getaktetes Verschicken aller gespeicherten Randmerkmale

Wenn in regelmäßigen Abständen alle gespeicherten Merkmale versendet werden, gehen keine Informationen endgültig verloren. Die Empfänger können das empfangenen

Umgebungsmodel mit ihren gespeicherten Model, zum Beispiel mit Durchschnittsbildung, verrechnen. Bei Verlust von versendeten Paketen, kann zwar genauso eine unterschiedliche Vorstellung der Roboter über den Rand entstehen, doch gehen keine Informationen endgültig verloren, da diese im nächsten Takt wieder versendet werden.

Bei dem Versendung der Daten ist es wichtig die Größe einer MTU[10] (1500 Bytes bei dem von uns verwendetem Ethernet-Protokoll) nicht zu überschreiten. Bei 240 gespeicherten Farbfeatures, zu je 3 Float Werten a 4 Bytes, kommen über 2800Byte zusammen. Da der Wertebereich jedes Farbkanals der Features zwischen 0 und 255 liegt lässt sich jedoch beim Versenden, durch Verwendung des Datentyps Char, die Größe auf 720Byte senken.

Durch die sekündliche Versendung des gesamten gespeicherten Umgebungsmodels, ist es möglich eine gute und jedem Roboter bekannte Vorstellung über die Randbereiche zu haben, auch wenn Datenpakete verloren gehen sollten.

5.3 Normalisieren der Daten

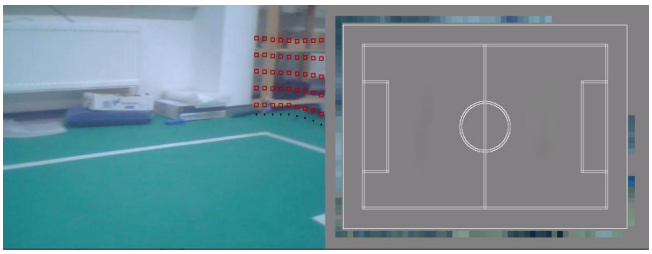
Durch unterschiedliche Kameraeinstellungen unter den Robotern, wie auch durch wechselnde Lichtverhältnisse, ist die Helligkeit der Kamerabilder teilweise sehr verschieden. Daraus resultiert, dass bei dem Verrechnen der Daten, das Ergebnis stark schwanken kann, auch wenn die zu Grunde liegenden Daten die gleichen sind. Deshalb stellt sich die Frage ob die Umgebungsmodelle vor dem Versenden normalisiert werden müssen.

5.3.1 Abgleich mittels der Feldfarbe

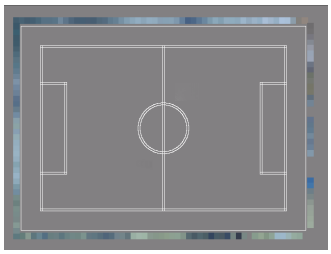
Für die Normalisierung würde sich die Feldfarbe eignen. Sie verändert sich während des Spieles nicht und wird zudem während der Bildsegmentierung bereits bestimmt (Vgl. [Reinhardt, 2011, S. 31-49]). Durch Subtraktion der Feldfarbe eines Bildes von den für die Feature verwendeten Pixeln, erhält man bei zuvor unterschiedlich beleuchteten Bildern ähnliche Ergebnisse. Für die Darstellung der folgenden Beispiele (Abb. 5.1) wurde ein Grünton[11] auf die Merkmale hinzuaddiert, um einerseits eine gültige Farbe zur Darstellung und andererseits eine natürlichere Farbe zu erhalten.

[10] Die Maximum Transmission Unit beschreibt die maximale Paketgröße, die auf einmal versendet werden kann

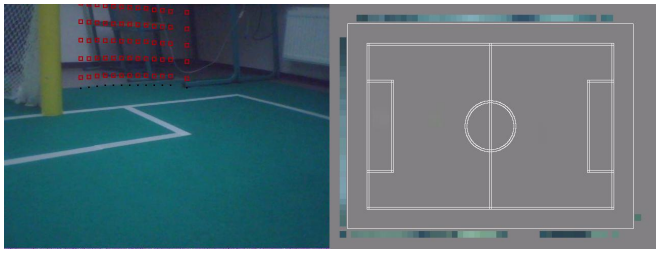
[11] hier: $cy = 116$, $cb = 147$ und $cr = 75$



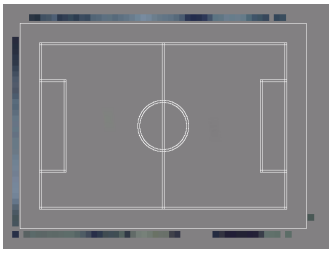
(a) normalisiert



(b) nicht normalisiert



(c) normalisiert



(d) nicht normalisiert

Abbildung 5.1: 5.1(a) und 5.1(b), sowie 5.1(c) und 5.1(d) wurden jeweils mit gleichem Bildmaterial erstellt. Dabei ähneln sich in (a) und (c) die, bereits vom Roboter gesammelten, Randmerkmale (um die Feldgrenze herum eingezeichnet). In (b) und (d) hingegen gibt es große Unterschiede

5.3.2 Ähnlichkeit vor und nach der Normalisierung

Um den Nutzen der Normalisierung zu testen, werden im folgenden Umgebungsmodelle miteinander verglichen. Dabei wurden diese unter verschiedenen Lichtverhältnissen aufgenommen. Das Ziel hierbei ist zu bestimmen, ob dabei die normalisierten Modelle eine höhere Ähnlichkeit aufweisen, als die nicht normalisierten. Zur Feststellung der Ähnlichkeit wird der Korrelationskoeffizient, also der unter dem Kapitel 4.7.4 vorgestellte Algorithmus, verwendet. Da bei der Aufnahme der Umgebungsmodelle nicht über jede Region Daten bezogen werden konnten wurden diese Regionen in den zu vergleichenden Modellen entfernt. Dies lässt sich damit rechtfertigen, das beim eigentlichen Vergleich mit bewertePositionsHypothesen auch nur die Teile der Modelle verglichen werden, in denen beide Modelle Informationen gespeichert haben.

Der Vergleich der normalisierten Bespiele ergab im Schnitt eine Ähnlichkeit von 0.68, wobei 1 heißt, dass die Modelle übereinstimmen, 0 dass sie “zufällig” anders sind und -1 dass sie gegensätzlich sind. Der Vergleich der nicht normalisierten Umgebungsmodelle ergab hingegen im Schnitt eine Ähnlichkeit von 0.75.

Tabelle 5.1: Die Daten wurden mit einer Folge von 890 Bildern erstellt

Testfall	ohne Normalisierung	mit Normalisierung	unterschied
Test 1	0.73277	0.68606	4.67%
Test 3	0.80035	0.75353	4.68%
Test 7	0.69785	0.64568	5.22%
Test 10	0.71558	0.66912	4.65%

Nun stellt sich natürlich die Frage, warum das Ergebnis ohne Normalisierung so gut ist und warum die Normalisierung es verschlechtert.
Dies lässt sich damit erklären, dass der Korrelationskoeffizient bereits eine Normierung vornimmt und Unterschiede in der Helligkeit heraus rechnet. Die zusätzliche Normierung mit der Farbe des Spielfeldes, verfälscht dabei das Ergebnis eher, da die Erkennung der grünen Feldfarbe ebenfalls nicht Fehlerfrei ist:
Sie wird für jedes Bild neu berechnet. Dabei wird ein Durchschnitt über die Farben des erkannten Spielfeldes berechnet und dieser als Feldfarbe gespeichert.

6 Abschießende Gedanken

In diesem viertel Jahr intensiver Arbeit wurden Algorithmen zur Verbesserung der Selbstlokalisierung im Roboterfußball entwickelt. Die einzelnen Algorithmen arbeiten fehlerfrei und entsprechend ihrer Spezifikation zusammen. Dies lässt sich an den fast nicht statt gefundenen Delokalisierungen während der Spiele des RoboCups 2012 und der Tests in Kapitel 4.8.2 quantifizieren.
Die einzelnen Algorithmen an sich konnten in dieser Arbeit leider nicht vollständig getestet werden, da dafür ein Simulator vonnöten gewesen wäre, welcher ein unverrauschtes ideales Umgebungsmodel erzeugt. Dieses hätte als Maßstab verwendet werden können, um die Ergebnisse der Algorithmen in einer, wiederum vom Simulator erzeugten, verrauschten Umgebung zu testen. Einen solchen Simulator zu entwickeln hätte den Rahmen dieser Bachelorarbeit gesprengt (siehe ein, im Rahmen eines Bachelor-Praktikums beim NAO-Team HTWK Leipzig erstellter, Simulator "Spielwiese" [Bellersen, 2011, S. 11]).

Das Ziel wurde erreicht und die Selbstlokalisierung verbessert. Trotzdem gibt es natürlich noch Verbesserungsmöglichkeiten:

Man könnte die Gewichte der im Umgebungsmodel gespeicherten Merkmale *EM*, falls sie länger nicht aktualisiert wurden, mit der Zeit abnehmen lassen. Dadurch können sich länger nicht aktualisierte Merkmale schneller ändern und so flexibler auf Änderungen in der Umgebung eingegangen werden. Allerdings muss hierbei mit gründlichen Tests herausgefunden werden, wie die Gewichte zu ändern sind.

Weiterhin könnte man nach dem Versenden von umgebungsmodel die Gewichte der gespeicherten Merkmale *EM* in die Verrechnung mit einbeziehen. Statt bei der Verrechnung des eigenen mit dem empfangenen Model einfach den Durchschnitt zu bilden, könnte man einen gewichteten Durchschnitt verwenden. Dies hätte immer dann vorteilhafte Auswirkungen, wenn sich die Umgebung in dem Randbereich bereits stark verändert hat, da länger nicht aktualisierte Merkmale nicht so stark in das eigene Umgebungsmodel einfließen würden.

Abschließend möchte ich bemerken, dass sich das gesamte Projekt auf der beigelegten CD befindet. Dies umfasst den C++ Quelltext der auf dem Roboter läuft, sowie den Java Code, der zu eingegebenen Bildern die Selbstlokalisierung visualisiert (siehe zum Beispiel S. 26). Des Weiteren befinden sich auch die verwendete Quellen, welche als PDF verfügbar waren, auf der CD.

Abkürzungsverzeichnis

B ..... Menge der Bewertungen – $b \in B \subset \mathbb{R}$ – S. 21

BK ..... Menge der Koordinaten der Mittelpunkte der Randbereiche – $(x, y) \in BK \subset \mathbb{R}^2$ – S. 19

BOOLEAN ..... Menge der Wahrheitswerte $= \{true, false\}$ – S. 15

EFG ..... Menge der Pixel-Koordinaten eines Bildes, die zu der erkannten Feldgrenze gehöhren – $EFG \subseteq PK$ – S. 15

EM ..... Menge der Erkennungsmerkmale – $(f, counter) \in EM$, wobei $f \in F$ und $counter \in \mathbb{N}$ – S. 19

F ..... Menge der Farben im YCbCr-Kodierung – $(Y, C_b, C_r) \in F \subset \mathbb{R}^3$ – S. 12

FOV ..... Das Field of View einer Kamera – $FOV \in \mathbb{R}$ – S. 10

P ..... Menge der Roboterpositionen mit zugehörigen Kamerawinkeln – $(x, y, roll, pitch, yaw, qualtitaet) \in P \subset \mathbb{R}^6$ – S. 13

PK ..... Menge der Pixel-Koordinaten – $(x, y) \in PK \subset \mathbb{N}^2$ – S. 7

RB ..... Menge der Randbereiche – S. 19

RELPK ..... Menge der für die Bildung von Erkennugsmerkmalen relevanten Pixelkoordinaten – $RELPK \subseteq PK$ – S. 22

RK ..... Menge der Roboter-Koordinaten – $(x, y) \in RK \subset \mathbb{R}^2$ – S. 7

RN ..... Menge der Nummern der Randbereiche – $rn \in RN \subset \mathbb{N}$ – S. 19

SW ..... Menge der Gewichte – $sw \in SW \subset \mathbb{R}$ – S. 21

WK ..... Menge der Welt-Koordinaten – $(x, y) \in WK \subset \mathbb{R}^2$ – S. 7

$x.a$ ..... Zugriffsoperator: falls $x = (a, b)$ mit $a = 7$ und $b = 9$ gilt: $x.a = 7$ und $x.b$=9 – S. 21

$[X]$ ..... Liste mit Elementen vom Typ X – S. 13

dom f ..... Ist der Definitionsbereich einer Funtkion f – S. 19

abstand ..... Bildet (wk_1, wk_2) auf dis ab – $wk_1 \in WK$, $wk_2 \in WK$ und $dis \in \mathbb{R}$ – S. 24

abstandPos ..... Bildet (p_1, p_2) auf dis ab – $p_1 \in P$, $p_2 \in P$ und $dis \in \mathbb{R}$ – S. 15

bereichsfindung ..... Bildet (efg, p) auf rn ab – $efg \in EFG$, $p \in P$ und $rn \in RN$ – S. 23

bild ..... Bildet k auf f ab – $k \in \mathbb{N}$, $f \in F$ – S. 34

erkenneFeldgrenze ..... Bildet kamerabild auf EFG ab – S. 15

erkennungsmerkmal ..... Bildet T auf em ab – $T \subseteq F$ und $em \in EM$ – S. 26

extractFeatures ..... Bildet $(p, \text{kamerabild}, \text{istFeldfarbe}, EFG)$ auf umgebungsmodel ab – $p \in P$ – S. 20

getCoord ..... Bildet rn auf bk ab – $rn \in RN$ und $bk \in BK$ – S. 19

istFeldfarbe ..... Bildet f auf $boolean$ ab – $f \in F$ und $boolean \in BOOLEAN$ – S. 15

kamerabild ..... Bildet pk auf f ab – $pk \in PK$ und $f \in F$ – S. 7

lokalisierung ..... Bildet kamerabild auf $[P]$ ab – S. 13

median ..... Bildet $[F]$ auf f ab – $f \in F$ – S. 25

modulo ..... Rechnet den Rest aus: $5 \,\text{modulo}\, 3 = 2$ – S. 25

pixelToRobot ..... Bildet (pk, p) auf rk ab – $pk \in PK$, $p \in P$ und $rk \in RK$ – S. 11

bewertePositionsHypothesen . Bildet $([P], \text{umgebungsmodel}, \text{kamerabild}, \text{istFeldfarbe}, EFG)$ auf $[(P, B)]$ ab – S. 21

robotToWorld ..... Bildet (rk, p) auf wk ab – $rk \in RK$, $p \in P$ und $wk \in WK$ – S. 12

tauscheSeite ..... Bildet $((p_1, b_1), (p_2, b_2), p_a, sw_a)$ auf (sw_n, p_n) ab – $p_1 \in P$, $p_2 \in P$, $p_a \in P$, $p_n \in P$, $b_1 \in B$, $b_2 \in B$, $sw_a \in SW$ und $sw_n \in SW$ – S. 21

umgebungsmodel ..... Bildet rn auf em ab – $rn \in RN$ und $em \in EM$ – S. 19

updateUmgebungsmodel ..... Bildet (umgebungsmodel, umgebungsmodel) auf umgebungsmodel ab – S. 21

vergleichen ..... Bildet (umgebungsmodel, umgebungsmodel) auf b ab – $b \in B$ – S. 33

verrechnen ..... Bildet (em_1, em_2) auf em_3 ab – $em_1 \in EM$, $em_2 \in EM$ und $em_3 \in EM$ – S. 21

zuordnung ..... Bildet wk auf rn ab – $wk \in WK$ und $rn \in RN$ – S. 24

Abbildungsverzeichnis

Listings

Literaturverzeichnis

[Behnke 2012] Behnke, Prof. Dr. S.: *RoboCup*. http://www.ais.uni-bonn.de/robocup.de/, August 2012. – [Online; accessed 05.08.2012] 2

[Bellersen 2011] Bellersen, Manuel: *Austauschbare Module zur Integrierung von KI in den Nao-Simulator "Spielwiese"*, HTWK-Leipzig, Bachelorarbeit, Oktober 2011. – manuel-bellersen.googlecode.com/files/Bachelorarbeit_fertig.pdf [Online; accessed 11.09.2012] 6

[Brose 2008] Brose, Jörg: *Partikelbasierte Ballmodelliernug in einem Team autonomer, vierbeiniger Roboter*, Technische Universität Darmstadt, Diplomarbeit, März 2008. – http://www.sim.informatik.tu-darmstadt.de/publ/da/2008-Brose.pdf [Online; accessed 09.08.2012] 4.1.1

[Burger u. Burge 2005] Burger, Wilhelm ; Burge, Mark J.: *Digitale Bildverarbeitung*. Springer Verlag, 2005 6, 7, 8, 9

[Gohout u. Reimer 2005] Gohout, Professor Dr. W. ; Reimer, Dr. D.: *Formelsammlung Mathematix für Wirtschaft und Technik*. Verlag Harri Deutsch, 2005 3.2.1

[Kangaroos u. a. 2012] Kangaroos, Austrian ; Bader, Markus ; Hofmann, Alexander ; Knoop, Jens ; Schreiner, Dietmar ; Vincze, Markus: *Team Description Paper*. http://www.cse.unsw.edu.au/~bradh/RoboCup/tdp2012_austrian-kangaroos.pdf, 2012. – [Online; accessed 07.05.2012] 4.5.4

[Meyberg u. Vachenauer 2003] Meyberg, Professor Dr. K. ; Vachenauer, Dr. P.: *Höhere Mathematik 1*. Springer Verlag, 2003 3.2.3

[MSL Technical Committee 2011] MSL Technical Committee: *Middle Size Robot League Rules and Regulations for 2012*. https://docs.google.com/file/d/0B6cdZrEa8IsWZWU2M2M3YzctYjAxMy00ZGRjLTk0MjctMTJmOGY1ZDA0ZmVh/edit?pli=1, Dezember 2011. – [Online; accessed 08.05.2012] 5

[Reinhardt 2011] Reinhardt, Thomas: *Kalibrierungsfreie Bildverarbeitungsalgorithmen zur echtzeitfähigen Objekterkennung im Roboterfußball*, HTWK-Leipzig, Masterarbeit, Oktober 2011. – http://robocup.imn.htwk-leipzig.de/documents/thesis_thomas_reinhardt.pdf?lang=en [Online; accessed 09.08.2012] 3.7, 3.5, 5.3.1, 6

[Robocub Technical Committee 2010] Robocub Technical Committee: *RoboCub Standard Platform League (Nao) Rule Book*. http://www.tzi.de/spl/pub/

Website/Downloads/Rules2010.pdf, Mai 2010. – [Online; accessed 07.05.2012] 2.5, 6

[RoboCup 2012a] RoboCup: *home Robot*. http://www.ais.uni-bonn.de/robocup.de/images/RC11/RC11_Home_Final_NimbRo_Cosero_making_Omelet.jpg, August 2012. – [Online; accessed 05.08.2012] 2.1, 6

[RoboCup 2012b] RoboCup: *rescue Robot*. http://www.ais.uni-bonn.de/robocup.de/images/RC11/RC11_Rescue_Koblenz.jpg, August 2012. – [Online; accessed 05.08.2012] 2.1, 6

[RoboCup Federation 2012] RoboCup Federation: *Middle Size League*. http://www.robocup.org/robocup-soccer/middle-size/, Mai 2012. – [Online; accessed 08.05.2012] 4

[RoboCup Federation Wiki 2012a] RoboCup Federation Wiki: *Small Size League*. http://wiki.robocup.org/wiki/Small_Size_League, April 2012. – [Online; accessed 08.05.2012] 2.1

[RoboCup Federation Wiki 2012b] RoboCup Federation Wiki: *Soccer Simulation League*. http://wiki.robocup.org/wiki/Soccer_Simulation_League, April 2012. – [Online; accessed 07.05.2012] 3

[RoboCup Soccer Simulator 2012] RoboCup Soccer Simulator: *Visualisierer*. http://robocup.mty.itesm.mx/app/webroot/img/robocup2d.png, Mai 2012. – [Online; accessed 07.05.2012] 2.2, 6

[Siciliano u. a. 2010] Siciliano, Bruno ; Sciavicco, Lorenzo ; Villani, Luigi ; Oriolo, Giuseppe: *Robotics: Modelling, Planning and Control*. Springer Verlag, 2010 3.2.1

[SimSpark virtual Nao 2012] SimSpark virtual Nao: *virtual Nao*. http://simspark.sourceforge.net/wiki/index.php/File:Models-nao.jpg, Mai 2012. – [Online; accessed 07.05.2012] 2.2, 6

[Szeliski 2010] Szeliski, Richard: *Computer Vision: Algorithms and Applications*. Springer Verlag, 2010 3.2.2