TUHH
Technische Universität Hamburg-Harburg

Institut
für
Zuverlässiges Rechnen

Erkennung und Lokalisation nicht statischer Objekte durch digitale Bildverarbeitung unter Verwendung der integrierten HD-Kameras des NAO-Robotiksystems

Bachelor Arbeit

vorgelegt von
Oliver Tretau

Hamburg, den 15. Oktober 2012

Erstprüfer: Prof. Dr.-Ing. Sven-Ole Voigt
Betreuer: Dipl.-Ing. Stefan Kaufmann

Erklärung
Hiermit versichere ich, Oliver Tretau, diese Arbeit im Rahmen der an der Technischen Universität Hamburg-Harburg üblichen Betreuung selbstständig angefertigt und keine anderen als die angegebenen Quellen und Hilfsmittel eingesetzt zu haben.

Oliver Tretau

Hamburg, den 15 Oktober 2012

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

Im Vordergrund der Arbeit mit dem *Nao*-Roboter an der TUHH steht der Aufbau einer Fußballmannschaft humanoider Roboter für eine Teilnahme am *RoboCup*. Hierbei handelt es sich um eine jährlich statt findende, internationale und interuniversitäre Meisterschaft im Fußball unter humanoiden Robotern. Der *Nao* wird bei dem *RoboCup* in der *Standard Platform League* verwendet, bei welcher sich die von den Teilnehmern verwendeten Fußballer nur in der Implementierung der Software unterscheiden. Die Hardware, also der Roboter, ist von allen Teams in derselben Ausführung zu verwenden.

Der *Nao*-Roboter nimmt den Großteil seiner Umwelt mit Kameras wahr. Damit er auf seine Umwelt adequat reagieren kann, muss der Prozess der Informationsaufnahme mit den Kameras über eine solide Bildverarbeitungsstruktur geregelt sein.

Abb. 1.1.: Der *Nao*-Roboter [Che12]

Da die Umwelt des *Nao*s Variationen unterworfen ist, beschäftigt sich diese Bachelorarbeit mit der Erkennung und der Lokalisation nicht statischer Objekte im Raum.
Für diese Bachelorarbeit steht die Entwicklung eines Moduls im Zentrum, welches dem *Nao* Bildverarbeitungsprozesse erlaubt, die ihm helfen ein nicht statisches Objekt, nämlich einen farbigen Ball auf einem Fußballspielfeld, zu detektieren. Dabei wird neben den allgemeinen Hardwareeigenschaften, die der *Nao* mit sich bringt, auch auf Methoden der Farb- und Formerkennung eingegangen. Das Modul wird in zwei Stufen erstellt: In der Simulation wird die Erkennung und Lokalisation in einer virtuellen Testumgebung vorentwickelt, um im Anschluss als Modul für den *Nao* implementiert zu werden.
Neben dem oben konkret gefasstem Ziel, soll mit dieser Bachelorarbeit die Forschung auf dem Gebiet der Humanoiden Robotor weitergeführt werden. Da ein großer Teil der Interaktion über Kameras geregelt werden kann, unterliegt der digitalen Bildverarbeitung ein besonders hoher Stellenwert in diesem Forschungbereich.

2. Nao-Robotiksystem

Der *Nao* ist ein programmierbarer humanoider Roboter des französischen Startup-Unternehmens *Aldebaran Robotics*. Zu Beginn des Projekts 2004 wurde das Ziel gesetzt, einen möglichst universell einsetzbaren Roboter zu entwickeln, daher findet der *Nao*-Roboter nicht nur Anwendung in Forschung und Entwicklung, sondern auch in der Lehre an Hochschulen und Universitäten. So wird er zum Beispiel seit 2008 als Nachfolger für den *Aibo* der Firma *Sony* in der *Standard Platform League* des *RoboCup*s verwendet.

2.1. Hardware

Der *Nao*-Roboter hat ein Gewicht von 5.2 kg, eine Höhe von 573 mm, eine Schulterbreite von 275 mm und eine Armspannweite von 311 mm. Für diese Arbeit verwendet wurde die Version H25 des *Nao*-Robotiksystems. Hierbei steht die 25 für die Anzahl der Freiheitsgrade. Zwei dieser Freiheitsgrade beziehen sich auch auf die Bewegungen, die durch den Kopf vollführbar sind. Außerdem befinden sich im Kopf des humanoiden Roboters zwei HD Kameras, die in der Stirn und in dem Mundbereich versteckt sind. Da die Kameras in unterschiedlichen Winkeln verbaut wurden, sind nicht nur Aufnahmen weiter entfernter Objekte möglich, sondern auch von Objekten die sich in unmittelbarer Nähe des Roboters befinden. Zudem sind eine Reihe weiterer Sensoren, wie Ultraschallsensoren und Berührungssensoren auch ein Beschleunigungssensor sowie ein Gyroskop, in dem *Nao*-Robotiksystem integriert, welche aber für die Ausarbeitungen dieser Arbeit keine Relevanz haben werden.
Da in dieser Bachelorarbeit die Balldetektion mit der im Kopf des *Nao*s eingebauten HD Kamera erarbeitet und untersucht wird, werden sonstige Beschreibungen der Hardware hier ausgelassen. Bei Interesse sei der Leser auf die Dokumentation des *Nao*-Robotiksystems [Rob12] verwiesen. Für die Kommunikation sind WLAN sowie Ethernet nutzbar.

2.1.1. Der Kopf

Die an der TUHH eingesetzten *Nao*-Roboter sind mit dem Kopf der Version 4.0 ausgerüstet, welcher mit einem ATOM Z530 1.6GHz Prozessor sowie 1GB RAM Arbeitsspeicher ausgestattet ist. Diese Ressourcen lassen daher gegenüber älteren Versionen auch die Bearbeitung von Bildern mit einer Verarbeitungsrate von bis zu 29fps in HD Auflösung zu.

2.1.2. Die Kamera

Hinter den beiden im Kopf des *Nao*s verbauten Kameras steckt ein *System on Chip* (SOC) Bildsensor mit der Modellnummer MT9M114. Eine maximale Auflösung für die Verarbeitung von Aufnahmen ist mit 1280 × 960 Pixel (also 1.22MP) gegeben, wobei die tatsächlichen aktiven Pixel einer Aufnahme einen geringfügig größeren Bereich mit 1288 × 968 Pixel abdecken. Dieser Umstand wird in der Berechnung der Position des Balles im Bildausschnitt eine Rolle spielen und kann zu eventuellen Ungenauigkeiten führen. Für einen Pixel ist bei der Betrachtung der Abmessung auf dem Bildsensor eine Fläche von 1.9 μm × 1.9 μm festgeschrieben.
Die Öffnungswinkel der Kameras betragen in horizontaler Achse 60.9° und in vertikaler 47.6°, woraus in der diagonalen Öffnung des sichtbaren Bereiches des *Nao*s ein Winkel von 72.6° resultiert. Die Reichweite des Sichtfeldes beginnt bei 30 cm und ist in die Ferne nur durch Unschärfe begrenzt.
Als Output liefert die Kamera Rohdaten im Bildatenformat *YUV 4:2:2*. Auf eine Erläuterung der Farbräume wird in Kapitel 3.1 eingegangen. Die Bilder werden mit der *Electric Rolling Shutter* (ERS) Methode aufgenommen. Bei diesem Verfahren wird nich nur ein einziger *Snapshot* pro Aufnahme gemacht, sondern das zu Grunde liegende Aufnahmefeld vertikal und horizontal gescannt. Da also nicht alle Bereichen des Aufnahmefeldes zur exakt selben Zeit gescannt werden, können schnelle Bewegungsabläufe zu Verzerrungen führen. Der Vorteil dieser Technik liegt darin, dass Bilder immer scharf sind.
Die Anordnung und Ausrichtung der beiden Kameras im Kopf des *Nao*-Roboters ergeben sich aus der Abbildung 2.1.

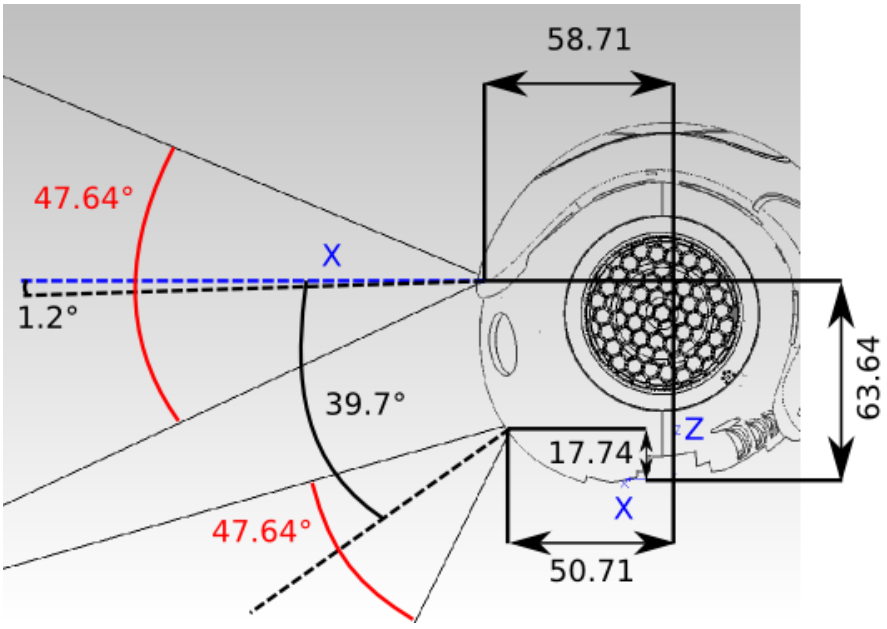

Abb. 2.1.: Anordnung der Kameras im Kopf des *Nao*-Roboters [Rob12]

2.2. Software

Der Hersteller *Aldebaran Robotics* stellt für den *Nao* neben dem Betriebssystem *OpenNAO* und dem Framework *NAOqi* auch eine graphische Entwicklungsumgebung *Choregraphe* (Kapitel 2.2.2) zur Verfügung, die schon über vielseitige Programmiermöglichkeiten verfügt. Unter anderem werden hier schon vorimplementierte Methoden der Bewegungsabläufe zur Verfügung gestellt sowie *text-to-speech* Algorithmen bereitgehalten [Rob12].
Dem Entwickler stehen jedoch nicht nur die *Aldebaran Robotics* eigenen Programme zur Entwicklung und Simulation zur Verfügung, sondern auch eine Vielzahl an Programme von Fremdanbietern zur Verfügung, wie zum Beispiel *Matlab* oder *Webots* (Kapitel 2.2.1). Außerdem können eigene Module in *Pyhton* oder *C++* erstellt werden. Mit dem grafischen Programmierwerkzeug *Choregraphe* können die mit einem der verfügbaren Software Development Kits (SDK) selbst entwickelten Programme auf dem *Nao* genutzt werden. Abbildung 2.2 zeigt eine kurze Übersicht der Einbindung externer Module in *Choregraphe*. Zum Testen bzw. zur Ausführung der

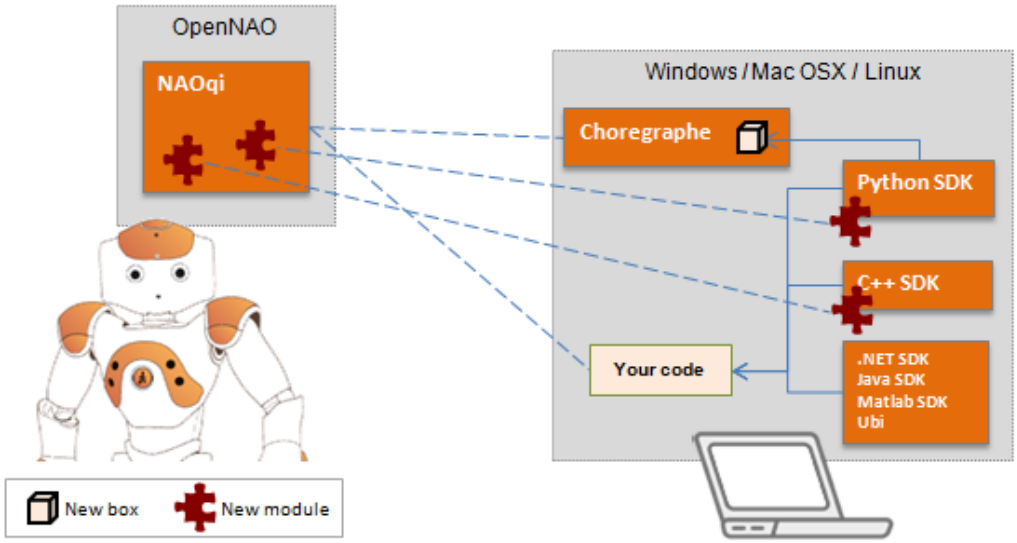

Abb. 2.2.: Einbindung externer Module mit *Choregraphe* [Rob12]

implementierten Module stehen dem Entwickler zwei Möglichkeiten zur Verfügung: die Ausführung des Programmes auf einem PC während die Daten der Sensoren und Motoren des *Nao*s übertragen werden (Remote) und die Ausführung des Programmes direkt auf dem *Nao* (Local).

Remote: Da bei dieser Form der Programmausführung der Prozessor des *Nao*-Roboters nicht für Berechnung genutzt wird, sondern lediglich die Daten der Sensoren übertragen werden müssen, kann diese Form der Programmausführung für Szenarien genutzt werden, in denen eine höhere Rechenleitung benötigt wird als die, die vom *Nao* bereitgestellt werden kann. Außerdem steht dem Entwickler hierbei offen, in die Ausführung des Programms einzugreifen bzw. bei Fehlern oder Ähnlichem zu reagieren.

Local: Die lokale Programmausführung ist letztendlich Ziel jeder Modulentwicklung, denn der *Nao*-Roboter soll als autonomer humanoider Roboter agieren und reagieren können. Hierbei besteht weiterhin die Möglichkeit der Überwachung der Sensor- und Motorenwerte für den Entwickler. Allerdings kann hierbei nicht mehr ins Programm eingegriffen werden. Des Weiteren können Datenströme auf lokaler Ebene schneller verarbeitet werden – so kann bei Bildern mit einer Auflösung von 1.22MP in der lokalen Ausführung mit Raten von bis zu 29fps gearbeitet werden, während die Rate beim entfernten Zugriff über Gigabit Ethernet bei dieser Auflösung auf bereits 10fps absinkt [Rob12].

Die in dieser Bachelorarbeit verwendeten Werkzeuge, Programme, Entwicklungsumgebungen und SDKs sind plattformunabhängig, können somit also unter Windows, Linux und Mac OS genutzt werden.

2.2.1. Simulation

Zur Simulation wurde *Webots* genutzt, eine Entwicklungsumgebung die dem Nutzer ermöglicht eine Reihe von Robotern zu gestalten, zu programmieren und zu simulieren, unter anderem auch den *Nao*-Roboter. Durch die Nutzung der *Education* Version von *Webots* kann die Entwicklungsumgebung mit *Matlab* verknüpft werden, so wie in dieser Ausarbeitung geschehen.

Webots EDU 6.4.4

Die Entwicklungsumgebung *Webots* (Abbildung 2.3) bietet eine ganze Reihe von Simulationen für den *Nao*. Kerngegenstände der Simulation sind hierbei die Welt (*World*), in welcher der *Nao* dargestellt wird, und die Gegenstände (*Protos*), welche in der Welt definiert sind. Über diese beiden Kernkomponenten lässt sich ein Spielfeld mit Robotern initialisieren. Ein weiterer wichtiger Kerngegenstand der Simulation ist die Einheit, welche die Verhaltensweisen des *Naos* definiert (*Controller*).

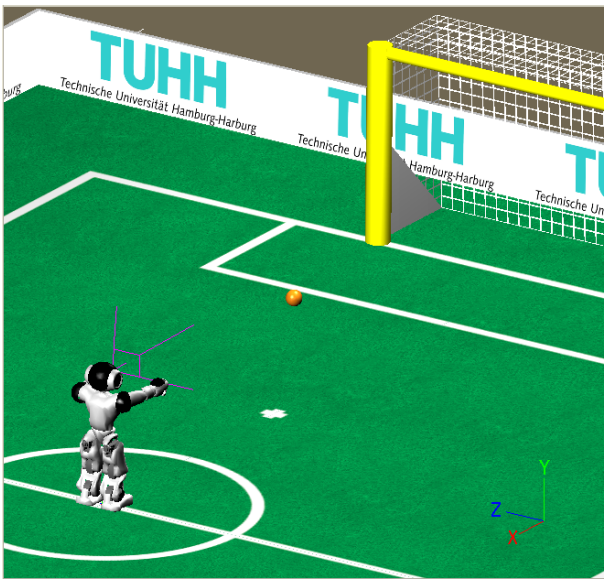

Abb. 2.3.: *Nao* in *Webots* [Cyb12]

Webots dient als sicherere Testumgebung für Module, die noch nicht direkt auf dem *Nao* ausprobiert werden können. Außerdem beinhaltet das Programm auch Tools, die dem Entwickler zum Beispiel ermöglichen, direkt während und auch außerhalb einer laufenden Simulation auf Bildmaterial des *Naos* zugreifen zu können.

World: Die oben bereits angesprochene Welt kann in *Webots* nach belieben modelliert werden und mit prinzipiell beliebig vielen Objekten (*Protos*) gefüllt werden. Das hier verwendete Spielfeld, sowie die genutzten *Protos* wurden für *Webots* von der Entwicklerfirma *Cyberbotics* implementiert und im Vorfeld dieser Bachelorabeit an der TUHH dem aktuellen Regelwerk des *RoboCup* angepasst.

Protos: In der Simulation können neben Robotern auch eine Vielzahl an Elementen zur Umgebungsgestaltung bzw. Hinderniserstellung eingebracht werden. Die *Protos* sind dabei Skripte, die in die *World* mit eingebunden sind. Sie defineren bei der Ausarbeitung unter Anderem die Tore auf dem Spielfeld, das Spielfeld als solches, den Ball usw. Sämtliche Elemente entsprechen hierbei auch den Vorgaben des *RoboCup*. Der hier verwendete *Proto* des *Nao*-Roboters ist in dem *Webots* Tool bereits in der Version 3.3 enthalten, wurde aber im Vorfeld an die während der Arbeit verwendete Kopfversion 4.0 angepasst.

Controller: Der Controller schreibt das Verhalten eines in die Welt integrierten *Nao*-Roboters vor und kann in den Hochsprachen *C*, *C++*, *Java*, *Python* oder *Matlab* implementiert werden. Hier war *Matlab* das Mittel der Wahl. Die Umgebung verfügt auch über einen eigenen Editor, der hier allerdings nicht verwendet wurde, da in der Entwicklung direkt auf dem Editor von *Matlab* zurückgegriffen wurde.

Matlab R2012a

Die Bildverarbeitung eines aus *Webots* oder der realen Welt erworbenen Bildes wurde zur Simulation und in Kooperation mit *Webots* vorerst nur unter *Matlab* implementiert, da das *Matrix Laboratory* schon sehr viele Möglichkeiten der Matrixmanipulation beisteuert und mit seiner *Image Processing Toolbox* auch noch Funktionen speziell zur digitalen Bildverarbeitung zur Verfügung stehen [TM12]. Da die Rohdaten eines *RGB* Bildes unter *Matlab* leicht in einer $M \times N \times 3$ Matrix dargestellt werden können, ergeben sich Manipulationen dieser Rohdaten in *Matlab* trivial über Zusammenhänge der linearen Algebra. Einige Funktionen, wie zum Beispiel die Konvertierung zwischen *RGB* und *YUV* Farbräumen, welche nicht in der *Image Processing Toolbox* verfügbar ist, wurden dazu noch selbst implementiert.

2.2.2. Modulentwicklung

Mit den oben angesprochenen Werkzeugen kann zwar relativ simpel getestet werden, allerdings lassen sich mit *Matlab* und *Webots* alleine keine Module entwickeln, die auch auf dem *Nao*-Roboter laufen würden. Ziel dieser Bachelorarbeit ist natürlich neben dem Testen der angedachten Algorithmen auch die Implementierung eines lauffähigen Moduls für den *Nao*, sodass getestete Umstände auch in der Praxis belegt werden können. Zur Modulentwicklung in *C++* und mit *OpenCV* muss auch die Entwicklungssoftware *Choregraphe* von *Aldebaran Robotics* sowie das herstellereigene Framework *NAOqi* genutzt werden.

Choregraphe

Die grafische Programmieroberfläche *Choregraphe* gestaltet die Programmierung des *Nao*s mit den von *Aldebaran Robotics* vorimplementierten Bausteinen sehr intuitiv indem Blöcke mit Bewegungs- bzw. Reaktionsabläufen per *drag-and-drop* miteinander verbunden werden können. Hierbei können Handlungsstränge sequentiell und/oder parallel laufen. Außerdem sind Kontrollstrukturen in Form von Schleifen sowie logischen Abfragen vorhanden, um eventbasierte Programmierung zu ermöglichen. Desweiteren können auch, wie eingangs schon erwähnt, selbst entwickelte Module in Hochsprachen wie unter anderem *C++* oder *Python* in die grafische Programmieroberfläche mit eingebunden werden.

Dem Entwickler steht auch die Möglichkeit offen, während der Programmausführung die Bewegungen des *Nao*s sowie die aktuelle Position im Programm und das aktuelle Kamerabild verfolgen zu können. Außerdem wurde in der Entwicklung des hierbei genutzten *tuhhSDK*, welches zuvor an der TUHH für den *Nao* entwickelt wurde und unter anderem Bewegungsabläufe bereitstellt, eine Log-Funktion implementiert, die in der Lage ist, Ausgaben an den *Debug* Monitor von *Choregraph* zu senden.

NAOqi

Das *NAOqi* Framework ist die von *Aldebaran Robotics* bereitgestellte Middleware, welche auf dem *Nao* läuft und die Kommunikation einer Vielzahl von bereits vorhandenen Modulen untereinander ermöglicht. Für die Zwecke dieser Modulentwicklung ist von Vorteil, dass auf Werte der Sensoren, Gelenkstellungen oder Motoren zugegriffen werden kann. Weiterhin kann das *NAOqi* Framework auch als SDK zur Ergänzung eigener Module mit den Vorgefertigten genutzt werden.
Für die Kommunikation mit der Kamera ist das *ALVideoDevice* ([Rob12]) entscheidend, welches mit der Klasse *ALVideoDeviceProxy* über die Möglichkeiten verfügt, auf die Bilder der Kamera zugreifen zu können und sie für die weitere Verarbeitung zugänglich zu machen.

C++ und OpenCV

Für die Nutzung von Methoden der digitalen Bildverarbeitung in der Hochsprache *C* oder *C++* bietet sich die frei verfügbare, plattformunabhängige Bibliothek *OpenCV* an, da mit den mehr als 2500 optimierten Algorithmen [Bra12] eine große Menge an Methoden für die digitale Verarbeitung von Bildern sowie das maschinelle Sehen zur Verfügung stehen. Da *Aldebaran Robotics* derzeit noch an dem Umstieg ihrer Software von *OpenCV 2.1* auf *OpenCV 2.3* arbeitet, wird in dieser Bachelorarbeit die Version 2.1 genutzt, damit die Module nicht nur *remote* sondern auch *local* auf dem *Nao* ausgeführt werden können.

3. Ballerkennung

Zur Erkennung seiner Umwelt, vor allem während des Fußballspielens beim *RoboCup*, setzt der *Nao*-Roboter seine beiden Kameras ein. In der *Standard Platform League* sind daher sämtliche zum Fußball spielen relevanten Objekte auf dem Feld farblich kodiert. Die Farbkodierungen ist im Detail dem Regelwerk [Inf12], dessen Schwierigkeitsgrad jährlich erhöht wird, zu entnehmen. Der hier zu detektierende Ball ist ein *Mylec* Hockey Ball der Farbe Orange. In seiner Form hat der Ball einen Durchmesser von 65 *mm* und ein Gewicht von 55 *g*.
Die beiden zu Testzwecken während der Simulation verwendeten Bilder sind in Abbildung 3.1 dargestellt. Das linke Bild entstammt einer Aufnahme des *Nao*s auf der *IdeenPark 2012* Messe[1], das rechte wurde in *Webots* vom simulierten *Nao-Proto* (siehe Kapitel 2.2.1) aufgenommen. In dieser Bachelorarbeit wird bei der Detetkion

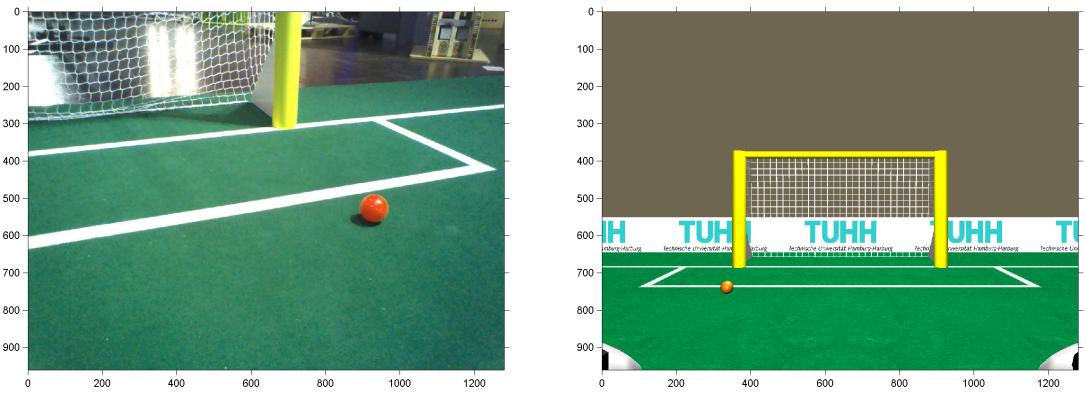

Abb. 3.1.: Originalbilder des *Nao* und aus *Webots*

des Balles zwischen der Farberkennung (Kapitel 3.2) und der Formerkennung (Kapitel 3.3) unterschieden. Zusätzlich zur Beschreibung des Vorgehens werden auch theoretische Aspekte der Farb- bzw. Formerkennung dargelegt.

3.1. Grundlagen zu Farbräumen

Nachfolgend werden grundlegende Annahmen für Farbräume in der digitalen Bildverarbeitung aufgezeigt, welche eine Basis für die erarbeiteten Berechnungen darstellen.

[1] Das Bild wurde aus dem nativen *YUV*-Farbraum des *Nao* in *RGB* Daten umgewandelt bevor es gespeichert wurde.

Der Mensch nimmt Farben als additive Kombination der drei Grundfarben Rot, Grün und Blau (*RGB*) wahr. Da dieses Farbmodell gerade für den Menschen besonders intuitiv ist, findet es nicht nur bei einigen Kameras oder Displays sondern auch in der digitalen Bildverarbeitung häufig Anwendung.

***RGB*-Farbraum:** Im *RGB*-Farbraum können die für den Menschen erkennbaren Farben in Kombination eines dreier Tupels mit Gewichtungen von 0 bis 255 der Grundfarben Rot, Grün und Blau zusammen gesetzt werden. In Abbildung 3.2 sind drei Visualisierungen dieses Farbraumes in drei Dimensionen dargestellt. Ein kurzer Überblick der dem Rot-Orange des Hockey Balles ähnlichen Farben in *RGB* Kombination ist außerdem in Tabelle 3.1 zu finden.

Farbe	**R**	**G**	**B**
Dunkles Orange	255	140	0
Orange-Rot	255	69	0
Tomate	255	99	71
Rot	255	0	0
Dunkel Rot	139	0	0

Tab. 3.1.: *RGB*-Rotwerte

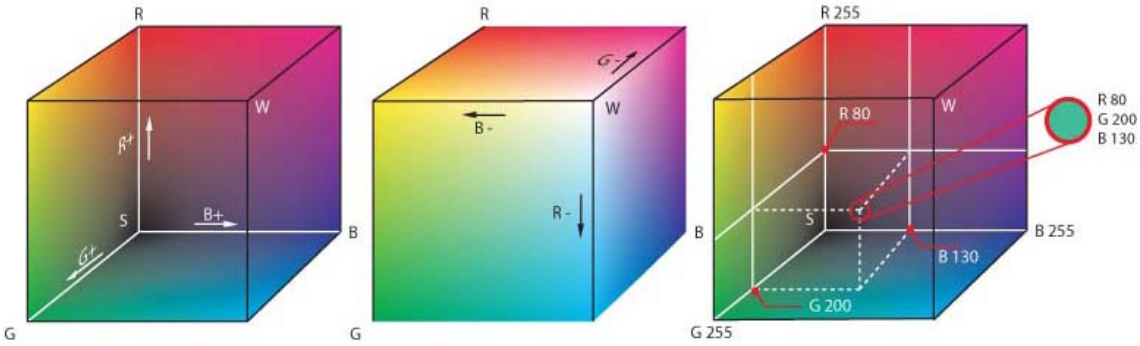

Abb. 3.2.: *RGB*-Farbraum dargestellt in drei Dimensionen[Inc12]

Allerdings ist der *RGB*-Farbraum nicht in jeder Hinsicht optimal zur Darstellung und Bearbeitung von Farben geeignet. Das *RGB*-Modell stößt, durch den oben erwähnten Aufbau aus drei farbigen Layern bedingt, schnell an seine Grenzen. So kann zum Beispiel der Farbton Orange leicht im *RGB*-Modell über die Kombination von $rgb = (255, 127, 0)$ definiert werden. Allerdings ergeben sich erste Probleme in der Wiedererkennung schon bei der Betrachtung zweier gleichfarbiger oranger Flächen mit unterschiedlichen Lichteinfällen und Schattenkompositionen, da die Helligkeit der Farbe in dem *RGB*-Modell nicht mit eingebunden ist.
Das Arbeiten und Modellieren in *RGB*-Farbräumen bringt also gerade für die Balldetektion beim *Nao* einige Hürden, die sich nicht umstoßen lassen. Die Lichteinfälle auf das *RoboCup*-Fußballfeld sind nicht kontrollierbar und daher kann eine Farbfilterung in *RGB* Darstellung bei sich ändernder Umgebung nicht eingesetzt werden. Ein

Farbraum, der nicht auf drei Farblayern aufgebaut ist, ist durch den *HSV*-Farbraum gegeben.

***HSV*-Farbraum:** Im *HSV*-Farbraum wird der Wert der darzustellenden Farbe mit Hilfe des Farbtons (*hue*), der Farbsättigung (*saturation*) und der Helligkeit (*value*) festgelegt. Der Farbton wird hierbei auf einer kreisförmigen Skala angegeben und besitzt Werte zwischen 0° und 360° (0° = Rot, 120° = Grün und 240° = Blau). Die Sättigung wird prozentual in einem Intervall von 0 bis 1 angegeben (0 = Neutralgrau, 0.5 = wenig gesättigte Farben und 1 = gesättigte Farben). Mit dem Hellwert kann zudem auch noch die Helligkeit in einem Intervall von 0 bis 1 eingerechnet werden (0 = keine Helligkeit und 1 = volle Helligkeit). Der *HSV*-Farbraum findet sich visualisiert in Abbildung 3.3. Außerdem verdeutlicht die Tabelle 3.2 die Darstellung von einigen ausgewählten Farben.

	Werte	**Weiß**	**Magenta**	**Rot**	**Grün**	**Blau**	**Schwarz**
H	$0° - 360°$	–	300°	0°	120°	240°	–
S	$0 - 1$	0	1	1	1	1	0
V	$0 - 1$	0.75	0.75	0.75	0.75	0.75	0

Tab. 3.2.: *HSV*-Farbwerte [Jac01]

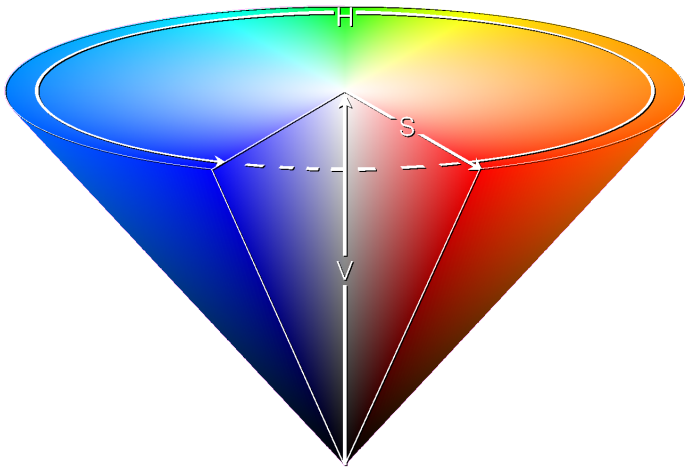

Abb. 3.3.: *HSV*-Farbraum dargestellt als Spitzkegel [Inc12]

Das *HSV*-System hat den deutlichen Vorteil gegenüber *RGB*, dass die Helligkeit in der Definition des Farbwertes mit integriert ist, sodass bei einer Filterung nach einer bestimmten Farbe ein Toleranzbereich für die Helligkeit angegeben und der Fehler durch Schatten in der Farberkennung eliminiert werden kann. Auch dadurch, dass die drei Grundfarben alle über den Wert der Farbtonkomponente angesprochen werden können, ergibt sich ein intuitiver Umgang mit dem Farbmodell.
Bei der Nutzung des *HSV*-Modells mit dem *Nao* ergibt sich der Nachteil, dass der

Nao ein anderes Farbsystem verwendet. Für eine Nutzung des *HSV*-Farbraumes auf dem *Nao* müsste vorerst, genau wie bei dem *RGB*-Farbmodell auch, eine Konvertierung von dem im *Nao* genutzten Farbraum *YUV* erfolgen. Diese Umwandlung ist in beiden Fällen *RGB* und *YUV* über Lineartransfomationen berechenbar, allerdings auch rechenintensiv. Das vom *Nao* genutzte *YUV*-Modell wird im folgenden Absatz diskutiert.

***YUV*-Farbraum:** Das *YUV*-Farbmodell wird zum Beispiel bei analogen Farbfernsehern und nach den Normen *PAL* und *NTSC* verwendet. Hier wird der Kanal Y (*luminance*) als Helligkeitswert für eine Farbe eingesetzt, die sich aus den beiden Farbkanälen U und V (*chrominance*) zusammensetzt. In der Abbildung 3.4 findet sich ein zweidimensionaler Plot, in dem V über U prozentual von -0.5 bis 0.5 aufgetragen ist bei einer Helligkeit Y von 0.5. Außerdem zeigt Gleichung 3.1 die lineare Konvertierung aus dem *RGB*-Farbraum in den *YUV*-Farbraum. Für *RGB*-Werte aus dem Bereich $0-255$ ergibt sich für Y ein Wertebereich von $16-235$, für U ein Bereich von $0-\pm112$ und für für V ein Bereich von $0-\pm157$ [Jac01].

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.147 & -0.289 & 0.436 \\ 0.615 & -0.515 & -0.100 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} \tag{3.1}$$

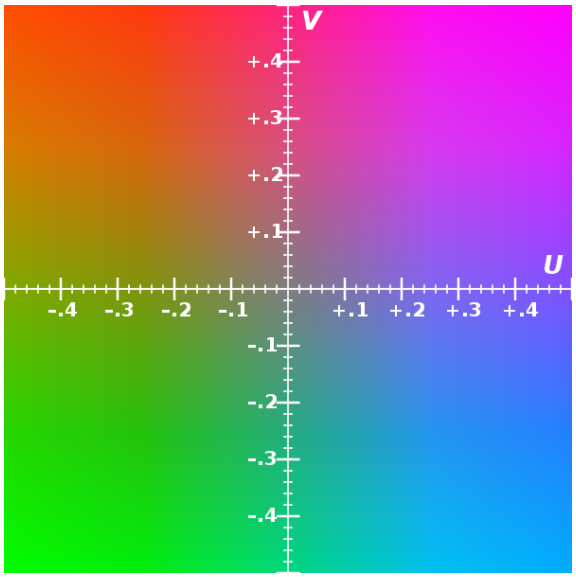

Abb. 3.4.: *YUV*-Farbraum als Plot von V über U bei $Y = 0.5$ [Inc12]

Anhand der Abbildung 3.4 ist auch zu erkennen, dass für eine Farbfilterung in dem *YUV*-Modell jetzt nicht mehr, wie im *RGB*-Farbsystem nötig, Toleranzen für einen dreidimensionalen Bereich angegeben werden müssen, sondern in zweidimensionalen Grenzen bestimmt werden kann, welcher Bereich gesucht ist. Diese Vereinfachung geschieht unter der Voraussetzung, dass die Helligkeit bei der Farbfilterung ein möglichst großes Interval abdeckt. Im Gegensatz zu *HSV* wird dieses Modell auch nativ in viele Eingabegeräten, vor allem Kameras genutzt. Daraus resultiert, dass bei der

Verarbeitung von Bildern auf dem *Nao* das *YUV*-Modell das ressourcenschonendste Modell sein wird, da Konvertierungen zu anderen Modellen nicht benötigt werden und somit Rechenzeit bei der Bildverarbeitung eingespart werden kann. Diese Annahme wird später in Tabelle 6.1 bestätigt werden.
Ein weiterer Farbraum, der für die Modulentwicklung in *OpenCV* von Bedeutung sein wird, ist der *YCbCr*-Farbraum, der eine normierte und skalierte Version des *YUV*-Farbraumes darstellt.

***YCbCr*-Farbraum:** Der *YCbCr*-Farbraum wurde während der Entwicklung des *digital component video standard*s eingeführt und stellt, wie bereits erwähnt, eine durch Skalierung und Normierung erzeugte Abwandlung des *YUV-Farbraumes* dar. Auch hier stellt Y die Helligkeit dar und U bzw. V stellen die Informationen für die Farwerte bereit, wobei Y einen Wert im Bereich von $16 - 235$ belegt und die Farbanteile U und V auf einem Bereich von $16 - 240$ definiert sind. Eine lineare Berechnung aus dem *RGB*-Farbraum ist auch hier möglich und findet sich in Gleichung 3.2. Des Weiteren sind in Tabelle 3.3 der Farbraum des *YCbCr*-Modells an einigen ausgewählten Farben erläutert.

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.172 & -0.339 & 0.511 \\ 0.511 & -0.428 & -0.083 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \tag{3.2}$$

	Werte	**Weiß**	**Magenta**	**Rot**	**Grün**	**Blau**	**Schwarz**
Y	16–235	180	84	65	112	35	16
Cb	16–240	128	184	100	72	212	128
Cr	16–240	128	198	212	58	114	128

Tab. 3.3.: *YCbCr*-Farbwerte aus [Jac01]

Im *YCbCr*-Farbraum sind außerdem mehrere Standards vorhanden. Das vom *Nao* verwendete Format ist *YCbCr 4:2:2*. Die Nummerierung *4:2:2* bezieht sich hierbei auf die Anordnung der *YCbCr*-Muster bei der Speicherung [Jac01]. Da während der Verarbeitung der Bilder nicht an dieser Anordnung editiert wird, geht diese Ausarbeitung nicht näher darauf ein.

3.2. Farberkennung

Als erster Schritt in der Segmentierung des gewünschten Bereiches im Bild steht die Erkennung der Farbe. Die Durchführung erfolgt aufgrund der oben dargelegten Umstände, im *YUV*-Farbraum. Zu Beginn wurde auch mit Filterung im *RGB*-Farbraum gearbeitet. Diese Methode wurde jedoch schnell revidiert, da wie oben schon diskutiert, keine Informationen über Helligekeit und Schatten im *RGB*-System integriert sind.

Während der Simulation in *Matlab* wurden die beiden in Abbildung 3.1 gezeigten Bilder verwendet, die im *RGB*-Farbraum vorliegen. Daher wurde zu Beginn der Farbfilterung zuerst ein Konvertierung des eigentlichen Bildes in den *YUV*-Farbraum vorgenommen. Die dafür genutzte lineare Transformation findet sich in der Gleichung 3.3. Des Weiteren findet sich in Tabelle 3.4 eine Auflistung der Farbwerte und Toleranzen (nach oben und unten) des orangen *Hockey Balles* im *YUV*-Farbraum. Die Farbwerte und Toleranzen liegen in dem von der Konvertierungsmatrix vorgeschreibenen Wertebereich des *YCbCr*-Farbraumes (Kapitel 3.1).

$$\begin{pmatrix} Y \\ U \\ V \end{pmatrix} = \begin{pmatrix} 0.299 & 0.587 & 0.114 \\ -0.169 & -0.332 & 0.500 \\ 0.500 & -0.419 & -0.0813 \end{pmatrix} \begin{pmatrix} R \\ G \\ B \end{pmatrix} + \begin{pmatrix} 0 \\ 128 \\ 128 \end{pmatrix} \tag{3.3}$$

	Y	**U**	**V**
Maximum	200	130	240
Minimum	16	16	170
Mittelwert	108	73	205
Toleranz	92	57	35

Tab. 3.4.: *YUV*-Farbfilterwerte in der *Matlab* Simulation

Im Anschluss an die Konvertierung folgt in der Simulation die Farbfilterung an Hand der in Tabelle 3.4 angegebenen Grenzen. Dazu wird das zu Grunde liegende Bild in den drei Layern des Farbraumes mit dem gültigen Wertebereich abgeglichen. Dieses Vorgehen der Farbfilterung, welches auch *Thresholding* genannt wird, ist im Folgenden erklärt.

Thresholding: In einer logischen Funktion werden die Wertebereiche des *Thresholdes* mit den Originalwerten aller Layer des Bildes verglichen. Sollten die Werte des Bildes an einem Punkt P in den Grenzen des gültigen Wertebereichs liegen – und zwar in allen drei Layern – dann wird der Punkt P in einem neuen Bild der selben Größe als logisch *true* gesetzt. Dieser Vorgang wird nun *Thresholding* genannt, da nur ein positives Ergebnis erzielt wird, sobald der gültige Bereich nicht überschritten wird. Resultat der Farbfilterung ist ein binäres Bild, in dem sämtliche im *Threshold* auftretende Flecken als logisch *true*, also als 1, markiert sind.
Der Prozess wird häufig auf Graustufenbilder angewandt, um besonders hoch oder tief liegende Grauwerte (*Single Thresholding*) oder beide (*Double Thresholding*) zu selektieren oder auszuschließen [McA04]. Der in dieser Bachelorarbeit durchgeführte Vorgang des *Thresholding* lässt sich mathetematisch in Gleichung 3.4 zusammen fassen:

Sei $A_{ij}(k)$ der Farbwert der k-ten Komponente im *YUV*-Farbraum an der Stelle (i, j) der Originalbildmatrix, dann ist der Wert an der Stelle (i, j) der Zielbildmatrix T gegeben durch

$$T_{ij} = \begin{cases} true & \text{falls } A_{ij}(k) \in \text{Toleranzbereich, mit } k \in \{1, 2, 3\} \\ false & \text{sonst} \end{cases} \tag{3.4}$$

Zusätzlich zur Farbfilterung wird das erhaltene Bild anschließend mit einem Filter gefaltet, damit Fehlfragmente bei dem Erkennen der Farbe eliminiert werden. Zur stärkeren Visualisierung der Ballform des erkannten Objektes wird der Fitler so gewählt, dass er im binären Bild die positiven Ergebnisse glättet.

Faltung: Das binäre Bild wird mit dem 5×5 Filter aus Gleichung 3.5 gefaltet. Dieser Filter wurde in der dargelegten Form verwendet, um die Runde Form des Balles zu modelieren. Bildpunkte, die sich vom momentan untersuchten Bildpunkt in horizontaler bzw. vertikaler Richtung weiter als eine Pixellänge entfernt aufhalten gehen schwächer in die Bewertung mit ein als andere. Bei der Faltung entstehen Werte die deutlich über dem Wertebereich – zwischen 0 und 1 – eines binären Bildes liegen. Deshalb wird das Ergebnis der Faltung anschließend mit einem Grenzwert (hier wurde 22 verwendet) verglichen. Nur Punkte, die genügend stark gewichtete Wertepaare um sich herum liegend hatten, werden ins Ergebnis der Faltung mit aufgenommen. Das Resultat ist ein geglättetes binäres Bild, in dem der *Hockey Ball* von den Punkten markiert wird, die eine 1-Wertigkeit besitzen.

$$F = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 2 & 6 & 2 & 1 \\ 1 & 2 & 2 & 2 & 1 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \tag{3.5}$$

Beispiel: Ein Beispiel zu den mathematischen Eigenschaften der hier beschreibenen Faltung ist nachfolgend angegeben [McA04]. Der Filter

$$\begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix} \tag{3.6}$$

angewandt auf die Grauwerte eines Bildausschnittes

	a	b	c	
	d	e	f	
	g	h	i	

ergibt für die grau unterlegte Position A_{ij} nach der Faltung

$$A_{ij} = a - 2b + c - 2d + 4e - 2f + g - 2h + i \tag{3.7}$$

Sollte im Anschluss ein *Threshold* auf $A_{i,j}$ angewendet und das Ergebnis zurück gespeichert werden, erhält man die hier verwendete Vorgehensweise.

3.3. Formerkennung

Nach der Segmentierung von Farben über die oben beschriebene Farbfilterung und Faltung gibt es häufig kein eindeutiges Ergebnis. Da im Vorherigen nur nach der Farbe gefiltert wurde, kann es vorkommen, dass mehrere Objekte derselben Farbe auf dem Bild als gefiltertes Objekt im binären Bild übrig bleiben. Deshalb muss nun festgestellt werden, welche Form das Objekt hat. Die Erkennung der Form erfolgt im allgemeinen über Kanten- bzw. Eckendetetktion im vorgefilterten Bild; in dieser Bachelorarbeit während der Simulation in *Matlab* über die Funktionen *corner()*, *edge()* und *bwboundaries()* [TM12]. Eine Kreisdetektion wurde nicht implementiert. Die Form des erkannten Objekts wurde allerdings zu Berechnung seiner Position genutzt.

3.3.1. Kantendetektion

Kanten stellen eines der markantesten Bildermerkmale dar, da sie aus Intensitätsunterschieden im Bild resultieren. Bei der Kantendetektion geht es also darum, diese Intensitätsunterschiede aufzuspüren. Dabei werden im wesentlichen drei Schritte durchgeführt.

1. Mit einer Maske werden auf dem Bild Intensitätsgradienten erzeugt.
2. Nennenswerte Gradienten werden mittels Thresholding selektiert.
3. Die errechneten Daten werden über Algorithmen zu Kanten zusammengefasst.

Je nach angewendetem Verfahren haben die einzelnen Schritte unterschiedliche Ausprägungen. Die Gradientenberechnung erfolgt meist über Differenzen und kann als linearer 3×3 Filter auf die einzelnen Bildpunkte mittels Faltung angewendet werden. Gleichung 3.8 verdeutlicht die *Sobel* Maske als Faltungsmatrix in x- bzw. y-Richtung [Pet06].

$$S = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{3.8}$$

Bei der *Sobel* Kantendetektion wird die Faltung mit der Matrix (Gleichung 3.8) in beiden Richtungen auf dem ursprünglichen Bildes durchgeführt, wobei eine Approximation der ersten Ableitung entsteht. Die Faltung in x-Richtung filtert die horizontalen Kanten und die Faltung in y-Richtung die vertikalen. Beim Zusammenfügen werden schließlich nur die Punkte übernommen, die ein Maximum darstellen. Die unten beschriebene *edge()* Funktion aus Matlab wird zur Kantendetektion mit der *Sobel*-Methode genutzt.

edge(): Die Funktion *edge()* extrahiert Kanten aus einem Graustufen- oder Schwarzweißbild und liefert ein binäres Bild selber Größe zurück, in welchem als Kanten detektierte Punkte als 1 und alle anderen Punkte als 0 kodiert sind [TM12]. Das Ergebnis der Kantendetetktion mit der *edge()* Funktion aus *Matlab* ist in Abbildung 3.5 zu finden.

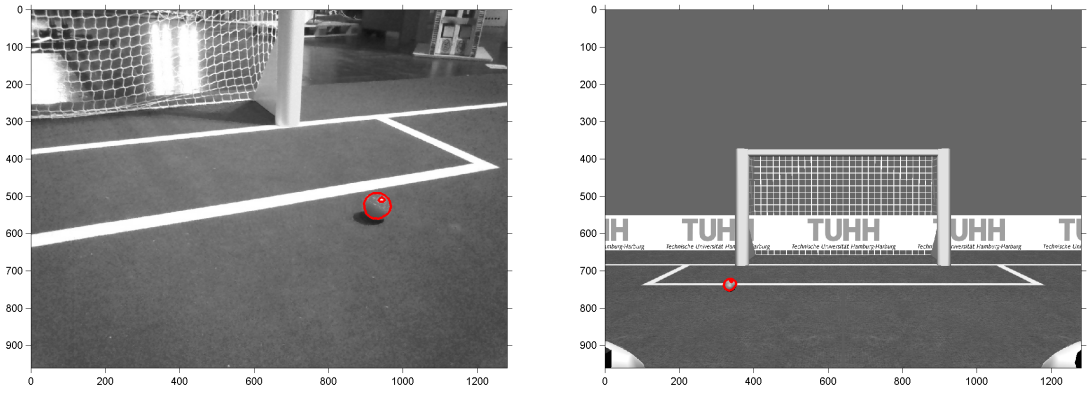

Abb. 3.5.: Kantendetektion

3.3.2. Eckendetektion

Neben den Kanten sind auch die Ecken Bildmerkmale, die bei der Ballsuche von entscheidender Bedeutung sein können, hierbei wird ähnlich zu der Kantendetektion nach Gradienten gesucht. Die in der Simulation genutzte *corner()* Funktion verwendet den *Harris corner detector*. Die *Harris* Methode baut auf dem *Moravec's corner detector* auf, dessen Funktion sich auch aus 3 Schritten zusammensetzt und auf Messungen der Intensitäten in kleinen Bildausschnitten bei Verschiebungen basiert.

1. Alle Verschiebungen ergeben kleine Unterschiede → einfarbige Region.
2. Alle Verschiebungen entlang der Kante ergeben geringe und alle senkrecht zur Kante ergeben hohe Intensitätsunterschiede → Kante.
3. Alle Verschiebungen ergeben hohe Intensitätsunterschiede → Ecke.

Die *Harris* Methode fängt eventuell auftretende Fehler in der *Moravec* Methode ab. Sie wurde als Erweiterung entwickelt und liefert mehr Sicherheit bei der Detektion von Ecken. Auf eine ausführliche Erklärung der mathematischen Zusammenhänge der *Harris* Methode wird an dieser Stelle verzichtet und auf tiefergehende Literatur (z.B. aus [Gri08]) verwiesen.

corner(): Die von *Matlab* durch die *Image Processing Toolbox* bereitgestellte Funktion *corner()* untersucht ein zu Grunde liegendes schwarzweißes oder binäres Bild nach Ecken und gibt dieses in der $M \times 2$ Matrix mit Zeilen und Spalten Koordinaten der Eckenpixel zurück, wobei M der Anzahl der detektieren Ecken entspricht [TM12].

Der Funktion kann außerdem eine preferierte Methode der Eckendetektion übergeben werden. Hier wurde die per *default* genutzte und oben beschriebene *Harris* Eckenerkennung verwendet. Die Ergebnisse der *corner()* Funktion sind zwar gut, liefern jedoch bei schwankenden Eingabeparametern, in Form von weniger gut farbgefilterten Bildern, eine geringere Sicherheit bei der Eckendetektion, als die in Kapitel 3.3.3

beschriebene Grenzdetektion. Das Ergebnis der Eckendetektion ist in Abbildung 3.6 festgehalten.

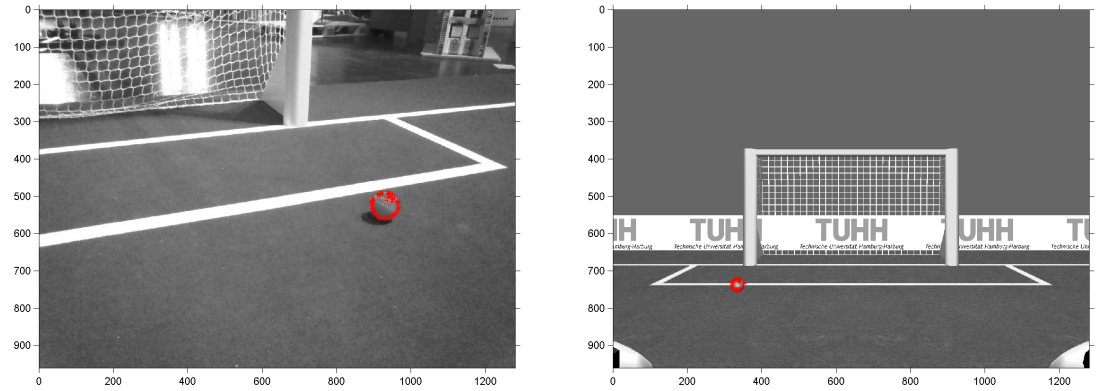

Abb. 3.6.: Eckendetektion

3.3.3. Begrenzungen

Neben Ecken und Kanten gibt es in *Matlab* auch noch die Möglichkeit Begrenzungen zu verfolgen. Eine Funktion, die Begrenzungen von Objekten detektieren kann, ist im folgenden vorgestellt.

bwboundaries(): Die ebenfalls in der *Image Processing Toolbox* enthaltene Funktion *bwboundaries()* ist dazu in der Lage, Begrenzungen in binären Bildern zu detektieren. Es werden sowohl äußere als auch innere Begrenzungen erkannt. Zurückgegeben wird ein $P \times 1$ Zellen Array, wobei P für die Anzahl der erkannten Grenzen steht, in welchem jede Zelle eine $Q \times 2$ Matrix mit Zeilen und Spalten Koordinaten der Grenzpixel enthält, wobei Q der Anzahl der erkannten Pixelpaare der zugehörigen Region entspricht [TM12].

Die Funktion kann bei der Erkennung von Grenzen auch zwischen äußeren (*Parent Objects*) und inneren (*Child Objects*) Grenzen unterscheiden. Da hier allerdings nicht näher auf eventuell auftretende Projektionen von Licht oder ähnlichem eingegangen wird und dieses im *YUV* Farbraum eine untergeordnetete Rolle spielt (vgl. Kapitel 3.1), wurden Grenzübergänge im Inneren des als Ball erkannten Objektes einfach mit einem *Hole Fill*, bei dem nur noch die äußere Grenze als Markierung gesetzt wird und der innerer Bereich gleichwertig gekennzeichnet wird, überschrieben. Resultat ist eine Begrenzung, die sich als äußeren Ring auf dem Ball visuell ausgeben lässt, dargestellt in Abbildung 3.7. Mit denen aus der *bwboundaries()* Funktion erhaltenen Daten wurde in der Simulation der Ballmittelpunkt berechnet.

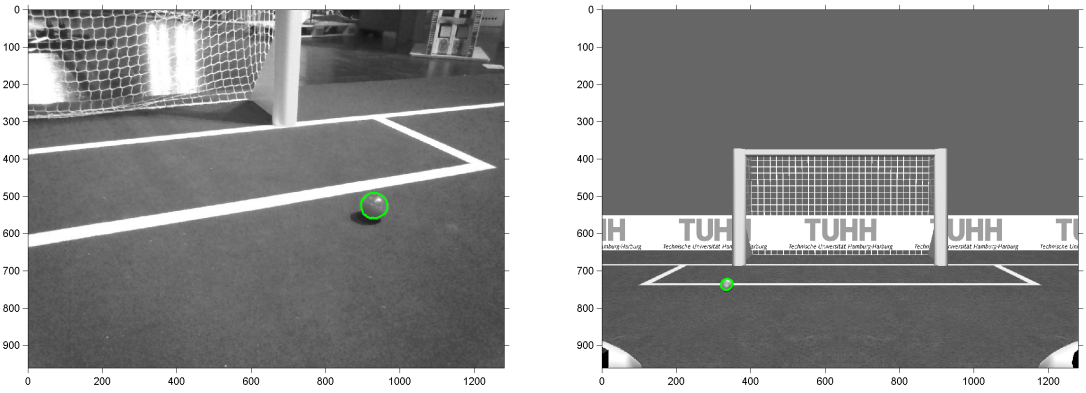

Abb. 3.7.: *bwbounderies()* liefert gewünschte Kantendetektion

3.4. Ergebnis der Ballerkennung

Zur Erzielung des nachfoglenden Ergebnisses wurden nur die Grenzübergangsdetektionen der *bwboundaries()* Methode eingesetzt, da die Evaluation des Ballmittelpunkts mit dem Rückgabeformat dieser Funktion am schnellsten durchzuführen war. Die Qualität der anderen Algorithmen *edge()* und *corner* differenziert sich nicht von dem Ergebnis der *bwboundaries()* Methode, allerdings wird die bei der Simulation verwendete Möglichkeit der Füllung innerer Reflektionen auf dem Ball, die von der Filterung nicht beachtet wurden, nicht bereitgestellt.

Das Ergebnis der Ballerkennung ist eine solide Simulation, die sowohl mit Bildmaterial aus *Webots* als auch aus der realen Welt eine Ballerkennung ausführen kann. Das Ergebnis ist noch einmal in Abbildung 3.8 dargestellt. Hierbei markiert die

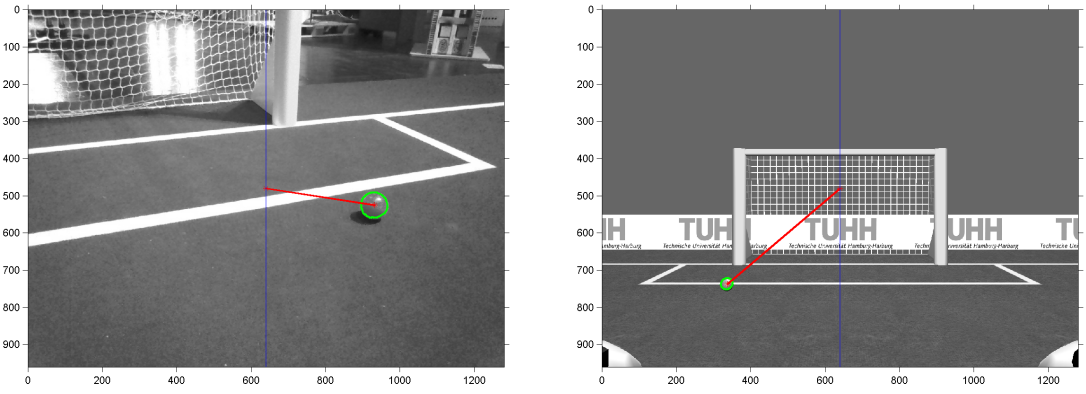

Abb. 3.8.: Ergebnis der Ballfilterung mit Darstellung des Richtungsvektors

blaue Linie die Bildmitte in vertikaler Achse, der roten Punkt auf ihr markiert die Bildmitte. Der rote Punkt auf dem erkannten Ball im Bild markiert die Mitte des Balles, dessen Begrenzungen mittels der *bwboundaries()* Methode gefunden wurden und in grün dargestellt sind, und die rote Linie markiert einen Vektor von der Bild-

mitte zur Position des Balles. Dieser Vektor wird entscheidend für die Berechnung der Position des Balles relativ zum *Nao* sein.

Als Resultat des ersten Teils der Objekterkennung ergibt sich ein nach Farben gefiltertes binäres Bild, ein Vektor, der die Richtungsangaben in x- und y-Koordinaten von dem Bildmittelpunkt zum Ballmittelpunkt im Bild beinhaltet, sowie einen Vektor, der die Winkel zur Projektion des Balles bezüglich des Mittelpunktes des Bildes, wieder in x- und y-Koordinaten, enthält. Auf den Hintergrund der Verwendung und der Berechnung dieser Winkel wird in Kapitel 4 eingegangen. Die vollständige Auswertung der im Rahmen dieser Bachelorarbeit angefertigeten Berechnungen findet sich in der Evaluation (siehe Kapitel 6).

4. Balllokalisation

Nachdem im ersten Teil der Objekterkennung und Lokalisation des nicht statischen Objektes *Hockey Ball* dieser auf dem vom *Nao* verarbeitetem Bild erkannt wurde und die Richtungsangaben, sowie Winkel, von der Bildmitte zur Ballprojektion auf dem Bild errechnet wurden, ergibt sich nun die Fragestellung, wie von diesen Koordinaten im Bild auf die realen Koordinaten im Spielfeld geschlossen werden kann. Hierbei wird sich zeigen, dass vor allem geometrische Modelle und lineare Transformationen bei der Lokalisation von Objekten, die von einer dreidimensionalen Ebene in eine zweidimensionale projiziert worden sind, zu einer Lösung des Problems führen werden.

Nachfolgend werden die der hier verwendeten Objektlokalisation über das *Pinhole* Modell (Kapitel 4.2) zu Grunde liegenden Annahmen sowie die Durchführung der Berechnung erläutert. Da sowohl für die Arbeit mit dem *Pinhole* Modell als auch für die Berechnung und Nutzung der Transformationsmatrizen, die bei der Lokalisation zum Einsatz kommen werden, die Brennweite der Linse der Kamera von Bedeutung ist, wird im ersten Abschnitt dieses Kapitel detailierter auf die Hardware des *Nao* eingegangen und die Methodik der Brennweitenberechnung, wie sie hier verwendet worden ist, dargelegt.

4.1. Hardware des *Nao*

Wie schon in Kapitel 2.1 beschrieben, sind in dem Kopf des *Nao*-Roboters zwei HD-Kameras verbaut. Außerdem konnte der Abbildung 2.1 bereits entnommen werden, dass das dem Menschen mögliche stereoskopische Sehen beim *Nao* nicht möglich ist. Die Ausrichtung der Kameras in Stirn und Kinn ist bei der Kopf Version 4.0 zwar so gewählt worden, dass die vertikalen Öffnungwinkel sich überlagern können, dennoch kann die Nutzung der beiden Kameras beim Erkennen von Objekten dadurch nur in einem minimalen Sichtbereich erfolgen. Bei der Lokalisation von Punkten auf zuvor ausgewerteten Bildern muss also auf andere Techniken zurückgegriffen werden. Der Vollständigkeit halber ist noch anzumerken, dass der verbauten Kamera keine Form des Zoomens offen stehen, weshalb auch eine stereoskopische Umgebungseinschätzung basierend auf Zoomen entfällt.

Um mit der Kamera des *Nao*s arbeiten zu können, muss bekannt sein, welche inneren Parameter zu Grunde liegen. Wichtig wird bei der Verarbeitung der Ballprojektion im Bild unter anderem die Brennweite der Kamera und die Größe der Einzelschrittwinkel im Kameraöffnungswinkel sein (vgl Kapitel 4.3 und 4.4), daher folgt im anschließenden Kapitel die Herleitung und Berechnung dieser beiden Parameter.

4.1.1. Berechnung der Brennweite

Die Abbildung 4.1 beschreibt grafisch den geometrischen Zusammenhang zwischen der Brennweite *f*, der Projektionsfläche D der realen Aufnahme, der Projektionsfläche d im Inneren der Kamera sowie dem Abstand F des Brennpunktes zur Projektion der realen Aufnahme. Die Einfallswinkel α und β entsprechen in beiden Fällen ihren Ausfallswinkeln und wurden deshalb selbig betitelt.

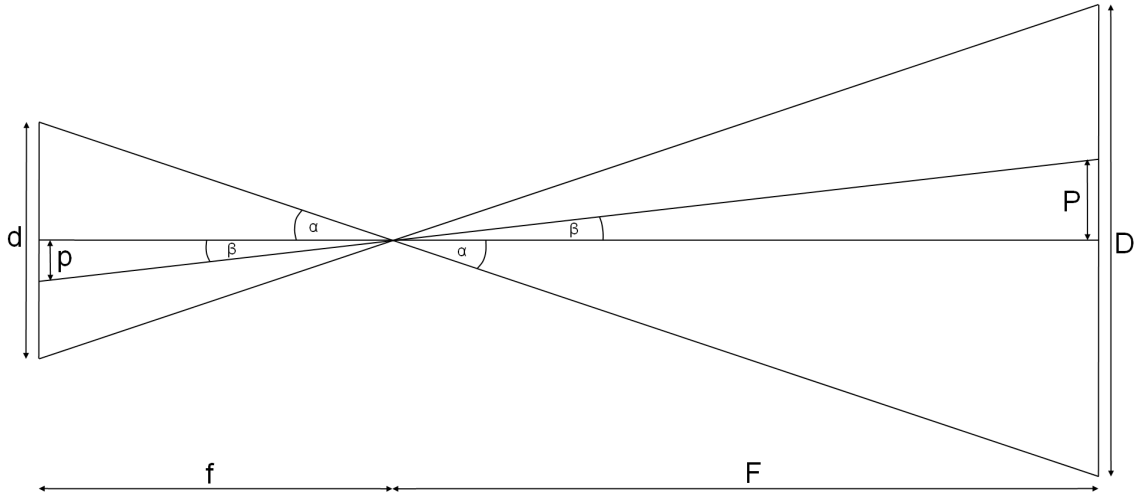

Abb. 4.1.: Brennweitengeometrie graphisch dargestellt

Ein gleichbleibender Öffnungswinkel der Größe 2α, der durch die Hardwareeigenschaften der Kamera zu Grunde liegt (der horizontale Öffnungswinkel der Kamera im *Nao* beträgt 60.9° und der vertikale 47.6°, vgl. Kapitel 2.1.2), und eine in der Länge nicht variierbare Projektionsfläche d in der Kamera sorgen nun dafür, dass sich die Brennweite der Kamera über den Zusammenhang des *Tangens* in einem rechtwinkligen Dreieck, der in Gleichung 4.1 dargestellt ist, berechnen lässt.

In einem rechtwinkligem Dreick gilt für einen Winkel α, der gegenüber der *Gegenkathete* und an der *Ankathete* liegt, allgemein:

$$\tan(\alpha) = \frac{Gegenkathete}{Ankathete} \tag{4.1}$$

Mit den aus den Literaturwerten ([Rob12]) erhaltenen Werte für die Pixelbreite *wd* und die Pixelhöhe *ht* des aufgenommenen Bildes, sowie mit dem Wert des horizontalen *Field Of Views (*FOV*) horFOV*, kann über den in Gleichung 4.2 angegebenen Zusammenhang das vertikale *FOV*, *verFOV*, berechnet werden. Im Anschluss erfolgt die Berechnung der Brennweiten in horizontaler und vertikaler Achse nach Gleichung 4.3.

$$verFOV = horFOV \cdot \left(\frac{ht}{wd}\right) \tag{4.2}$$

$$flx = \frac{\left(\frac{wd}{2}\right)}{\tan\left(\frac{horFOV}{2}\right)} \quad \text{bzw.} \quad fly = \frac{\left(\frac{ht}{2}\right)}{\tan\left(\frac{verFOV}{2}\right)} \tag{4.3}$$

4.1.2. Berechnung der Einzelschrittwinkel

Nachfolgend wird die Berechnung der Einzelschrittwinkel des Öffnungswinkels der Kamera aus der Brennweite und den äquidistanten Pixelabständen auf der Projektionsfläche der Kamera sowie die Berechnung der Schrittweiten auf der Projektionsfläche des realen Bildes mathematisch und mit Implementierung in *Matlab* 4.3 dargestellt.

Die Projektionsfläche d der Kamera ist, wie oben angegeben, nicht in der Länge variierbar, kann also, da die Auflösung der Kamera in horizontaler sowie vertikaler Achse auch nicht geändert werden kann, in äquidistanten Abständen der Pixelmenge der horizontal bzw. vertiakal verfügbaren Bildpunkte eingeteilt werden. Bei der Einteilung der Abstände in der Projektion des realen Bildes kann nicht davon ausgegangen werden, dass die Abstände der Bildpunkte gleichverteilt sind. Aus Abbildung 4.2 geht hervor, dass zwischen den projizierten Bildpunkten (p0,p1,p2,p3,...,pN) in der Kamera und den aus der realen Welt projizierten Bildpunkten (P0,P1,P2,P3,...,PN) ein nicht linearer Zusammenhang, verursacht durch den *Tangens*, zu bestehen scheint.

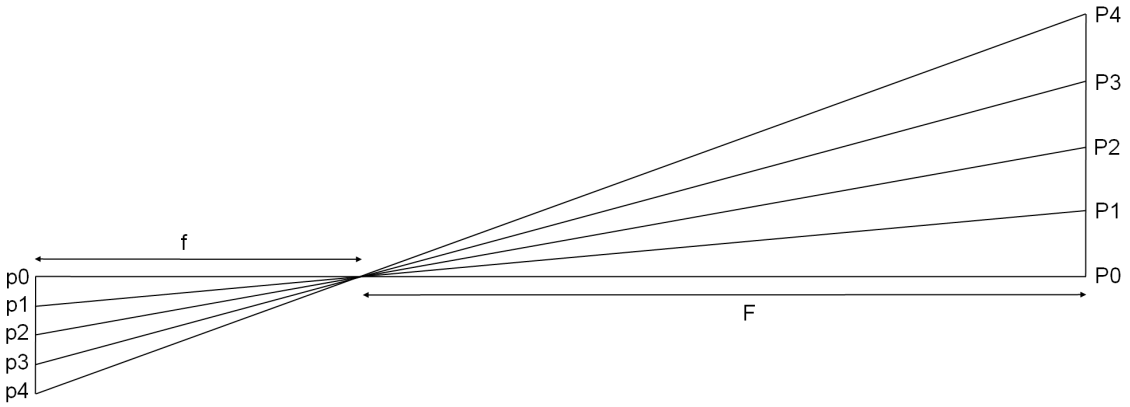

Abb. 4.2.: Einzelschrittprojektionen grafisch dargestellt

Und in der Tat ergibt sich die Berechnung der Einzelschrittwinkel α_i nur in Abhängigkeit des *Tangens*. Der im Index stehende *offset* bewirkt eine Verschiebung des Startpunktes zu dem Bildpunkt $p0$. Ob der Laufindex i nach oben oder unten läuft ist nicht von belangen, da die erhaltenen Winkel für einen Schritt i mit negativem Vorzeichen gespiegelt werden. Gleichung 4.4 verdeutlicht den mathematischen Zusammenhang.

$$\alpha_{i+offset} = \left(\frac{\arctan(i)}{fl}\right) \cdot \left(\frac{180}{\pi}\right) \tag{4.4}$$

Nachfolgend findet sich ein Ausschnitt (Abbildung 4.3) aus dem *Matlab* Quellcode für die oben hergeleiteten Berechnungen der Brennweite und der Einzelschrittwinkel. In den Kommentaren fällt auf, dass die Ergebnisse der berechneten Brennweiten in der horizontalen und vertikalen Richtung bereits in der ersten Nachkommastelle voneinander abweichen. Dieser Fehler tritt auf, da während der Berechnung nur

mit den aktiven Pixeln des Kamerabereichs gerechnet wird. In der Praxis liegen die Grenzen der Öffnungswinkel bei 1288×968 Pixel, wie auch schon eingangs in Kapitel 2.1.2 erwähnt.

Da während der Berechnung mit Pixeln gearbeitet wird und somit auch die resultierende Einheit der Brennweite in Pixel angegeben ist, wirkt der Wert der Brennweite sehr groß, hier ist allerdings noch nicht berücksichtigt, dass ein Pixel nur eine ideale Größe von 1.9 μm belegt. Mit dieser Berücksichtigung kann die Differenz der beiden Werte für die hier durchgeführten Berechnungen vernachlässigt werden. Außerdem wird in den weiteren Berechnungen der Vektoren und Winkel für die x- bzw y-Koordinate die jeweilige über das horizontale bzw. vertikale FOV berechnete Brennweite verwendet.

```
wd = 1280;
ht = 960;
horFOV = 1.064;
verFOV = horFOV*(ht/wd);
flx = (wd/2)/tan(horFOV/2); % Brennweite horizontal = 1.0873e+003
fly = (ht/2)/tan(verFOV/2); % Brennweite vertikal  = 1.1385e+003

alphah = zeros(wd+1,2);
alphav = zeros(ht+1,2);
% calculate angles for image width and horizontal apex angle
for i=1:1:wd/2
    alphah(i+(wd/2)+1,:) = ([atan((i)/flx)*(360/(2*pi)) i]);
    alphah(((wd/2)+1-i),:) = -alphah(i+(wd/2)+1,:);
end
% calculate angles for image heigh and vertical apex angle
for i=1:1:ht/2
    alphav(i+(ht/2)+1,:) = ([atan((i)/fly)*(360/(2*pi)) i]);
    alphav(((ht/2)+1-i),:) = -alphav(i+(ht/2)+1,:);
end
```

Abb. 4.3.: Berechnung der Einzelschrittwinkel in *Matlab*

4.2. Pinhole Modell

Die Projektion eines Objektes aus einer dreidimensionalen Umgebung in eine zweidimensionale Fläche kann im Wesentlichen als zentral perspektivische oder parallele Projektion beschrieben werden. Der Zusammenhang der beiden Koordinatensysteme kann sogar linear dargestellt werden, zum Beispiel mit dem *Pinhole Modell*.

Zunächst wird eine Projektionsmitte (*Center Of Projection* (*COP*)) als Ursprung des Koordinatensystems definitiert. In diesem Modell dient die Kameralinse des *Nao*-Roboters als Ausgangspunkt. Aus diesem Ursprung kann ein Punkt P in der Projektionsebene der realen Aufnahme dargestellt werden als $P = (x_1, x_2, x_3)$ und in eine zweidimensionale Ebene zum Punkt Q, in der Projektionsebene der Kamera,

mit $Q = (y_1, y_2)$ projiziert werden. Mathematisch ist also eine Projektion aus dem $\mathbb{R}^3$ in den $\mathbb{R}^2$ gesucht (vgl. Abbildung 4.4).

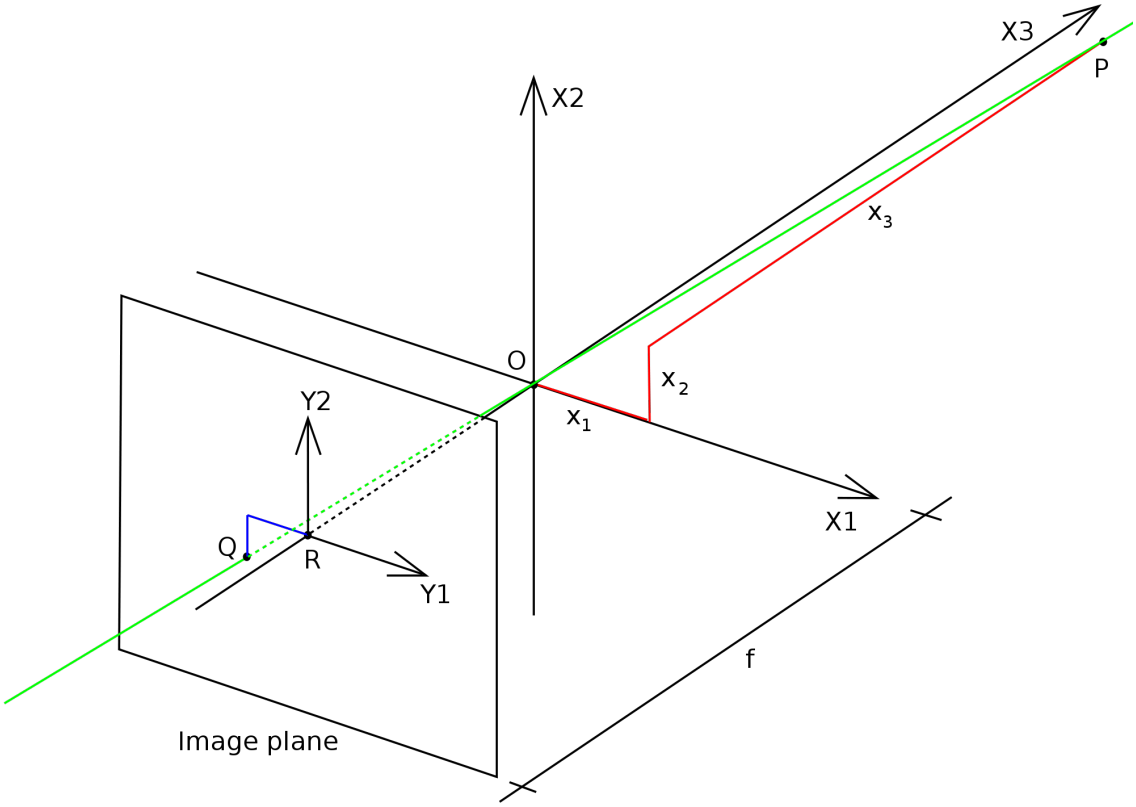

Abb. 4.4.: Pinhole Modell in der Gesamtübersicht [Inc12]

In der Abbildung ist zu erkennen, wie ein Punkt P mit den Koordinaten (x_1, x_2, x_3) aus dem Raum $X \in \mathbb{R}^3$ über den *COP* auf die Projektionsfläche in der Kamera (hier als *image plane* bezeichnet) in den Raum $Y \in \mathbb{R}^2$ projiziert wird. Dabei erhält der in der Ebene Y enstandene Projektionspunkt Q die Koordinaten (q_1, q_2). Die Brennweite ist gegeben durch f. Der Punkt R markiert den Mittelpunkt der *image plane* Projektionsfläche. Trigonometrisch ergibt sich nach der Abbildung 4.5 die Projektion des realen Punktes (x_1, x_2, x_3) auf den Punkt $\left(f \cdot \frac{x1}{x3}, f \cdot \frac{x2}{x3}, f\right)$ im *COP*.

Vom Koordinatensystem der Kamera (in Abbildung 4.4 mit der Achsenmarkierung (X1,X2,X3) bezeichnet) zur Projektion im Kamerainneren ergibt sich also die Gleichung

$$(x_1, x_2, x_3)^T \rightarrow \left(f \cdot \frac{x_1}{x_3}, f \cdot \frac{x_2}{x_3}\right) \tag{4.5}$$

Im Nachfolgenden wird die Herleitung der Kamerakalibirerungsmatrix aufgezeigt ([Fis97]), mit der im *Pinhole* Modell die Transformation von der Projektionsebene in der Kamera zum äußeren Koordinatensystem der Kamera durchgeführt werden kann.

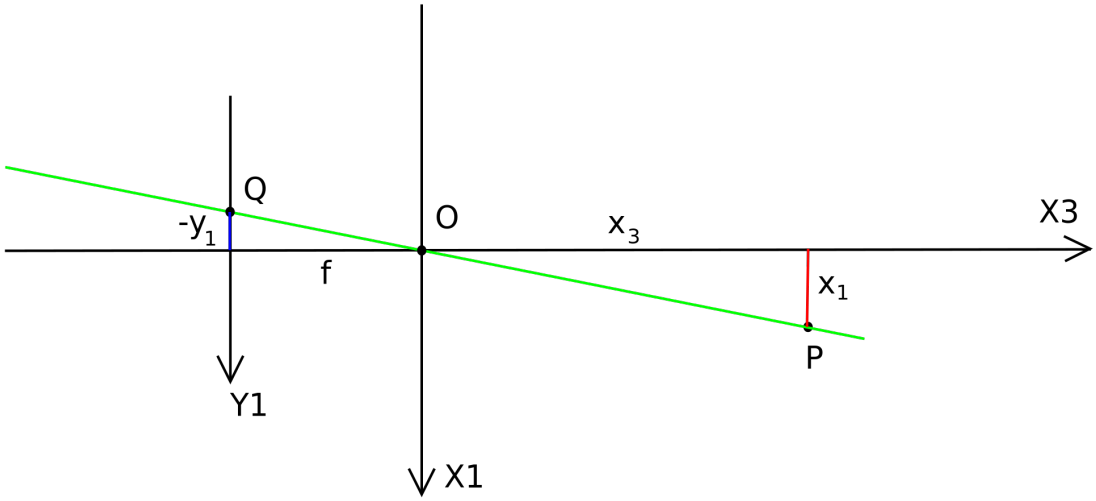

Abb. 4.5.: Pinhole Modell aus der Sicht der $x2$ Achse [Inc12]

Kamera: Zur Aufstellung einer Kameramatrix, mit der es möglich ist linear zwischen den beiden Ebenen untereinander zu transformieren, wird der Term umgeschrieben. Anstelle des zweidimensionalen Zielvektors wird ein dreidimensionaler Vektor $Y = (y_1, y_2, 1)$ gesetzt durch Hinzufügen einer dritten Gleichung. Es ergibt sich

$$\begin{pmatrix} y_1 \\ y_2 \\ f \end{pmatrix} = \frac{f}{x_3} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} \tag{4.6}$$

Bild: Für interne Kameraparameter kann diese Gleichung als Abbildung von Homogenen Koordinaten (4.7) geschrieben werden, hierbei beschreibt die resultierende 3×4 Matrix eine Abbildung vom $\mathbb{R}^3$ in den $\mathbb{R}^2$, kann jetzt aber auch für die Rücktransformation verwendet werden.

$$\begin{pmatrix} y_1 \\ y_2 \\ f \end{pmatrix} = \frac{f}{x_3} \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix} \tag{4.7}$$

Im nächsten Schritt wird die Kalibirerungsmatrix der Kamera ermittelt. Dazu werden die Bildachsen der kamerainternen Projektion links oben beginnend in horizontaler Richtung mit u und in vertikaler mit v betitelt. Dann ergibt sich mit $R = (u_0, v_0)$ der Punkt Q aus Abbildung 4.4 zu Gleichung 4.8. Der Faktor k ist in der Einheit [Pixel/Länge] aufgestellt.

$$Q = \begin{pmatrix} q_1 \\ q_2 \end{pmatrix} = \begin{pmatrix} k_u y_1 \\ k_v y_2 \end{pmatrix} = \begin{pmatrix} u - u_0 \\ v_0 - v \end{pmatrix} \tag{4.8}$$

Ein projezierter Punkt kann jetzt über die Gleichung 4.9 direkt in den Bildkoordinaten errechnet werden. Die entstehende 3×3 obere Dreiecksmatrix C wird Kamerakalibrierungsmatrix genannt. Eine Kamera gilt als kalibriert, wenn C bestimmt wurde.

$$\begin{pmatrix} u \\ v \\ 1 \end{pmatrix} = \begin{pmatrix} fk_u & 0 & u_0 \\ 0 & -fk_v & v_0 \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ f \end{pmatrix} = C \begin{pmatrix} x_1 \\ x_2 \\ f \end{pmatrix} \tag{4.9}$$

Damit ist ein System entwickelt, mit welchem von der Bildprojektion in der Kamera in das Koordinatensystem des *COP* transformiert werden kann. Zur Detektion des Balles auf dem *RoboCup* Fußballfeld muss anschließend noch über die Motorenstellungen des *Nao*-Roboters von dem im Kamerakoordinatensystem des *Nao* erkannten Balls auf die Fußpunkte des Roboters zurückgerechnet werden, um den Ball vollständig zu lokalisieren.

In der Praxis wurde die Kamerakalibrierungsmatrix nicht implementiert, daher finden sich um Quellcode keine identischen Anwedung des *Pinhole* Modells in der Form, in der es hier beschrieben wurde. Dennoch ist die Herleitung der Kamerakalibrierungsmatrix auf dem hier gezeigtem Weg realisierbar. In den nächsten beiden Unterkapitel erfolgt eine Erklärung der Berechnungen über die Öffnunungswinkel der Kamera (Kapitel 4.3) und die Beschreibung der tatsächlich genutzten Herleitung des Kamerakoordinatensystems des *Nao*s über eine Berechnung mit Vektoren (Kapitel 4.4), welche Parallelen zum *Pinhole* Modell aufweist.

4.3. Berechnung mit Winkeln

Die Methodik der Berechnung der Koordinatentransformation über die im Bild errechneten Winkel wurde bisher unter Verwendung des *tuhhSDK* genutzt, allerdings wurde bei bisherigen Implementierungen nicht mit der korrekten Berechnung der Einzelschrittwunkel gearbeitet. Prinzipiell kann sofern die Winkel in x- bzw y-Richtung einer Projektion vom Kamerakoordinatensystem in die reale Welt bekannt sind, diese Winkelauslenkung über die Motoren im Kopf des *Nao* virtuell nachvollzogen und rücktransformiert werden. Danach muss nur über die auch für die Vektorenmethode nötigen Transformationsmatrizen der Gelenke des *Nao*s in das Koordinatensystem der Fußpunkte abgebildet werden und die Winkel zum Ball von den Fußpunkten aus sind berechnet.

Bei der Umsetzung der Balldetektion wurde dieser Weg jedoch nicht eingeschlagen, da die Berechnung über Vektoren, welche im nachfolgenden Kapitel erläutert wird, einfacher ist.

4.4. Berechnung mit Vektoren

Das Kamerakoordinatensystem im Kopf des *Nao*s entspricht dem aus Abbildung 4.4, wenn die Ausrichtung der $x2$-Achse invertiert wird. Unter der Annahme, dass an dieser Stelle von der Bildverarbeitung, die im Vorfeld den Ball im Bild erkannt hat, nur die Abstände der x- und y-Koordinaten auf der internen Projektion vom Mittelpunkt des Bildes (y_1, y_2) in Pixeln bekannt sind, ergibt sich unter Verwendung der Gleichung 4.7 folgendes lineares Gleichungssystem in Gleichung 4.10.

$$\begin{pmatrix} \frac{f}{x_3} & 0 & 0 & 0 \\ 0 & \frac{f}{x_3} & 0 & 0 \\ 0 & 0 & \frac{f}{x_3} & 0 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ 1 \end{pmatrix} - \begin{pmatrix} y_1 \\ y_2 \\ f \end{pmatrix} = 0 \tag{4.10}$$

Da sich ein Lösungsraum ergibt, der unendlich viele Lösungen beherbergt, wurde dieser Weg in der Umsetzung umgangen. Stattdessen werden die Abstände (y_1, y_2) direkt (siehe Gleichung 4.11) in das Koordinatensystem des *Nao*s umgeschrieben (ähnlich Gleichung 4.6) und anschließend die in dem *tuhhSDK* bereits implementierten Methoden, die Kopf-, Torso- oder Fußmatrizen über Rotations- und Translationmatrizen ineinander umzurechnen, genutzt, um das enstehende Koordinatensystem auf die Mitte der Fußpunkte zurückzurechnen. In der nachfolgenden Gleichung beschreibt *cdv* die Kamerablickrichtung. Das Tupel (x_1, x_2, x_3) beschreibt das Koordinatensystem der Kamera und das Tupel (y_1, y_2) das des projizierten Bildes innerhalb der Kamera.

$$cdv = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \frac{1}{f} \begin{pmatrix} f \\ -y_1 \\ -y_2 \end{pmatrix} = \begin{pmatrix} 1 \\ -\frac{y_1}{f} \\ -\frac{y_2}{f} \end{pmatrix} \tag{4.11}$$

Im weiteren Prozess wird nun von der Kamera zum linken Fuß transformiert, damit zusammen mit der Ausgangsposition der Kamera des *Nao*s bezüglich des linken Fußes ein Vektor errechnet werden kann, der dann zur Suche eines erhöhten Schnittpunkt mit der Fußbodenebene verwendet wird. Die Erhöhung der x_3-Achse des *Nao*-Koordinatensystems fließt bei der Berechnung des Schnittpunktes über den Parameter λ mit ein (siehe Quellcodeausschnitt 4.6) und simuliert den Radius des zu suchenden Balles (Durchmesser 65 mm), damit Ungenauigkeiten bei zu geringem Abstand keine Berechnungsfehler hervorrufen können. Anschließend wird die Position über Verlängerung des Vektors mit dem errechneten λ durchgeführt und der Koordinatenursprung in die Mitte der Füße gesetzt. Damit ist die Detektion des Balles abgeschlossen.

4.5. Ergebnis der Transformation

Der in der Simulation mit *Webots* und *Matlab* entwickelte Algorithmus kann Abstände zum *Hockey Ball* aus Bildern, die vom simulierten *Nao-Proto* gemacht worden sind, über die Motorenstellung des *Nao*s berechnen. Das Resultat ist hier ein 4×1

```
% Berechnung der Blickrichtung
CamDirection = ([1 -y(1)/f -y(2)/f 0])
% Bestimmung von Kamera bis Fuß
HeadPitch2Torso = C_call('BB_getKinematicMatrix',1)
Cam2Torso = HeadPitch2Torso * TransZ(63.64) * TransX(58.71) * RotY
    (1.2*TO_RAD)
Cam2LFoot = C_call('BB_getKinematicMatrix',26) \ Cam2Torso
CamPosition = Cam2LFoot(1:4,4)
% Schnittpunktberechnung unter Berücksichtigung des Ballradius
lambda = -((CamPosition(3)-32.4752)/CamDirection(3))
Ball2LFoot = CamPosition + lambda*CamDirection'
% Verschiebung zur Beinmitte
Ball2FootCenter = Ball2LFoot + ([0 50 0 0])'
```

Abb. 4.6.: Berechnung der Ballposition in *Matlab*

Vektor, der in den ersten 3 Zeilen die x-, y- und z-Koordinaten des gefilterten Balles bezüglich des Koordinatensystems des *Nao*-Roboters enthält; dieses entspricht in der Simulation dem Koordinatensystem des Spielfeldes. Für die Auswertung der Qualität dieser ermittelten Punkte auf dem Spielfeld sei auf die Evaluation, in der auch weitere Auswertungen diskutiert werden, in Kapitel 6 verwiesen

5. Implementierung für den Nao

Der in der Simualiton mit *Matlab* enthaltene Algorithmus muss für die Implementierung in *C++* an einigen Stellen umgeschrieben werden. In *Matlab* genutzte Funktionen wie zum Beispiel die *bwboundaries()* Methode aus Kapitel 3.3.3 sind durch *OpenCV* nicht vorimplementiert und benötigen daher eigene Ersatzalgorithmen. In dem folgenden Kapitel sind die Änderungen bei der Modulimplementierung für den *Nao* dargelegt.

5.1. Ballerkennung

Auch hier liegt, genau wie in der Simulation, das von *Nao* aufgenommene Bild als Eingabe vor. Die Bilddaten werden vom *ALVideoDevice* des *ALVisionModuls* bereitgestellt und liegen als *IplImage* (von *OpenCV* bereitgestellte Klasse) im Farbraum *YUV4:2:2* (siehe Kapitel 3.1) vor. Der von diesem Farbraum benötigte Bereich (zwischen 16 und 240 mögliche Werte) ist mit der *IplImage* Struktur, welche Farbräume in Wertebereichen von 0 bis 255 darstellen kann, abgedeckt. Wie schon angesprochen wird an dieser Stelle keine Konvertierung in einen anderen Farbraum unternommen, da das *YUV4:2:2* Format bereits in der Simulation zu positiven Ergebnissen geführt hat. Die zur Bildverarbeitung genutzte Struktur ist die *cv::Mat* Klasse aus *OpenCV*. Da eine Anpassung der Auflösung auch nicht stattfindet wird in der nativ verwendeten HD Auflösung von 1280×960 Pixel gearbeitet.

In der Abbildung 5.1 findet sich eine Aufnahme mit der Kamera des *Nao*. Das Bild enthält die in *RGB* Komponenten dargestellten Farbwerte des *YUV*-Farbraumes.

Abb. 5.1.: Bildverarbeitung auf dem *Nao* – Originalbild

Für die Filterung in ein binäres Format werden die Farbwerte und Toleranzen aus Tabelle 5.1 genutzt. Diese Werte entsprechen denen aus der Simulation. Die in *Matlab* genutzte Implementierung der Filterung wird an dieser Stelle ähnlich verwendet, allerdings liegt mit der *cv::Mat* Struktur eine andere Bildrepräsentation zu Grunde, welche die Layer des *YUV*-Farbraumes hintereinander anordnet. Das bedeutet, dass die Bildmatrix eine Proportion von $(3 \cdot 1280) \times 960$ besitzt. Mit *Pointern* wird diese Bildmatrix durchlaufen und jeder Punkt (i, j) in allen drei Farblayern mit seinen Toleranzwerten abgeglichen. Befinden sich Werte im akzeptierten Bereich wird der Punkt (i, j) einer Zielmatrix des Formats *cv::Mat* mit nur einem Layer (binäres Bild) mit dem Wert *true* markiert. In Abbildung 5.2 ist das gefilterte Bild des *Nao* zu sehen.

	Y	**U**	**V**
Maximum	200	130	240
Minimum	16	16	170
Mittelwert	108	73	205
Toleranz	92	57	35

Tab. 5.1.: *YUV*-Farbfilterwerte in der *C++* Implementierung

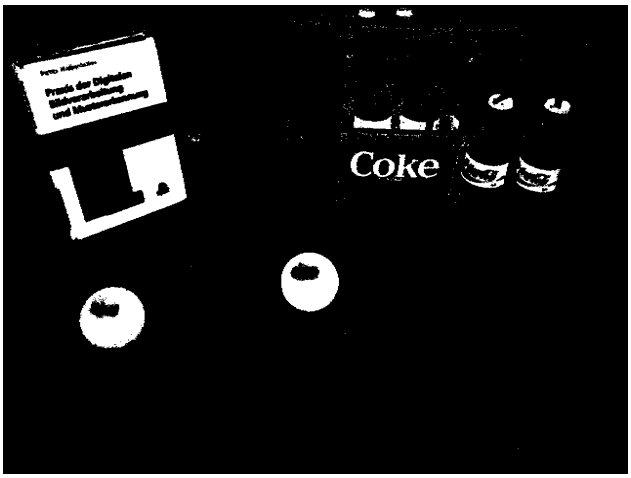

Abb. 5.2.: Bildverarbeitung auf dem *Nao* – Bild gefiltert

Nach der Filterung folgt die Faltung zur Glättung der Objekte im erhaltenen binären Bild. Dieser Prozess wird wieder bedingt durch die von *OpenCV* vorgegebene Struktur per Hand ausgeführt. Via Schleifenstruktur und mit *Pointerarithmetik* wird jeder Bildpunkt des binären Bildes aufgerufen, der neue Wert mit der in der Simulation genutzten Faltungsmatrix aus Gleichung 3.5 berechnet und der logische Wert, welcher aus einem *Thresholding* resultiert, in ein neues binäres Bild vom Datentyp *cv::Mat* zurückgeschrieben. Als Grenze wurde hier, wie in der Simulation, ein Wert von 22 verwendet. In Abbildung 5.3 findet sich das Ergebnis der Faltung.
Der Vorgang besitzt also bis hierhin noch dieselbe Struktur wie in der Simulation. Die Formerkennung, welche an die Filterung und Faltung anschließen soll, wurde noch

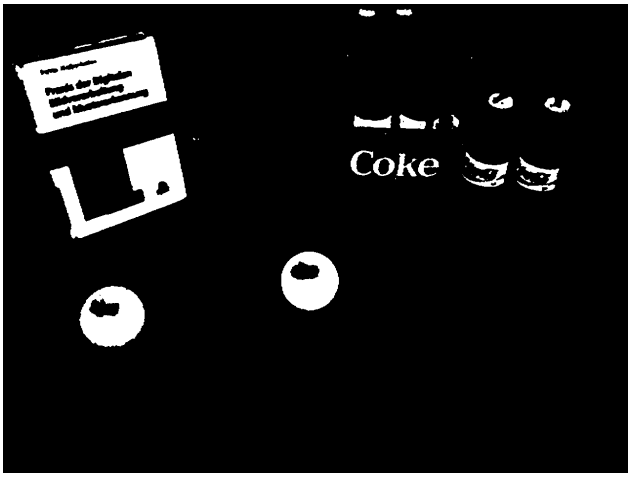

Abb. 5.3.: Bildverarbeitung auf dem *Nao* – Bild gefaltet

nicht vollständig implementiert, ist aber lauffähig. Im nächsten Schritt wird bei der Erkennung des möglichen Ballkandidaten im gefalteten binären Bild nur der Punkt verwendet, der dem unteren Rand des Bildes am nächsten ist. Prinzipiell soll in dem Schritt der Ecken- bzw. Kantenerkennung eine Möglichkeit der Formüberprüfung bereitgestellt werden. Diese Überprüfungsmethode ist bei der Implementierung auf dem *Nao* aus Zeitgründen noch nicht erfolgt.

5.2. Balllokalisation

Zur Koordinatentransformation in C++ für den *Nao* konnte der verwendete Algortihmus komplett übernommen werden. Neben den Synthaxanpassungen mussten lediglich während der Simulation verwendete *mex*-Aufrufe (eine Möglichkeit der Verwendung von *C++* Quellcode des *tuhhSDK* unter *Matlab*) durch equivalente Aufrufe in *C++* ersetzt werden.

6. Evaluation

Mit den bisherigen Ergebnissen ist – in der Simulation mit *Matlab* und *Webots* vollständig, in der praktischen Implementierung mit dem *Nao* unvollständig, aber mit den richtigen äußeren Umständen funktionierend – eine Lösung des Problems der Erkennung und Lokalisation des *Hockey Ball*s entwickelt worden. In diesem Kapitel werden die durch den entwickelten Algorithmus erzielten Ergebnisse untersucht und bewertet werden. Dabei wird zwischen der Simulation in Abschnitt 6.1 und der Implementierung in Abschnitt 6.2 unterschieden.

6.1. Simulation

Die Simulation wurde mit Bildern aus *Webots* und der realen Welt durchgeführt. Nachfolgend werden die Ergebnisse aufgezeigt und anschließend bewertet.

6.1.1. Ergebnisse

Bei beiden Bilder aus der realen Welt und der *Webots*-Welt, dargestellt in Abbildung 6.1, wurden nach Einflüssen der Auflösung auf die Ausführungszeit untersucht.

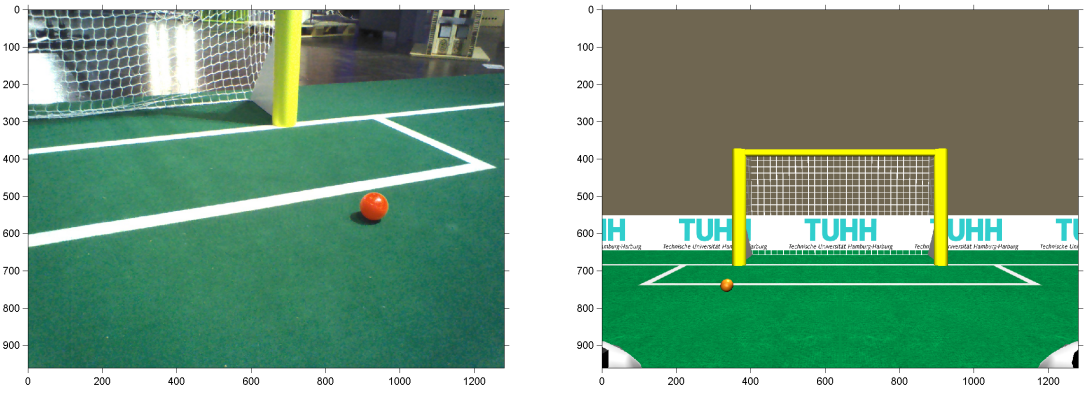

Abb. 6.1.: Originalbilder des *Nao* und aus *Webots*

In der folgenden Tabelle 6.1 findet sich eine Auflistung der von den einzelnen Stufen der Bildverarbeitung benötigten Ausführungszeit, dabei wurde auch die Ausführungzeit der Konvertierung, welche linear berechenbar ist, von RBG nach YUV mit angegeben. Alle Verarbeitungsschritte wurden im YUV-Farbraum ausgeführt und die Zeitangaben sind in Sekunden notiert. In der Tabelle wurden nur die in Kapitel 3 beschriebenen Schritte durchgeführt. Als Resultat ist die Position des Balles auf dem Bild, sowie die Winkel vom Bildmittelpunkt zum Ball berechnet worden.

Zur Bestätigung des Ergebnisses wurden die Winkel zum Ball ausgehend von der Bildmitte mit angegeben. An ihnen lässt sich auch bestätigen, dass die Auflösung der verarbeiteten Bilder keinen Einfluss auf die Qualität der Filterung hat, da der Winkel bei unterschiedlichen Auflösungen identisch bleiben muss.

	Reales Bild		**Bild aus *Webots***	
Auflösung	640×480	1280×960	640×480	1280×960
Konvertierung [s]	0.884	3.437	0.836	3.387
Filter und Faltung [s]	0.087	0.370	0.085	0.365
Kanten und Ecken [s]	0.175	0.629	0.167	0.602
Position und Winkel [s]	0.016	0.021	0.012	0.023
Horizontaler Winkel [°]	15.03	15.03	−15.52	−15.57
Vertiakler Winkel [°]	2.31	2.26	12.77	12.72

Tab. 6.1.: Ausführungszeiten realer und simulierter Bilder in *Matlab*

Das Ergebnis der Koordinatentransformation soll am Beispiel des in Abbildung 6.2 gezeigten Bildes ermittelt werden. Das Bild entstammt einer Aufnahme der *Nao* Kamera in *Webots*. Es wird der in *Matlab* implementierten Funktion, welche die Bildverarbeitung und die anschließende Transformation ins Fußkoordinatensystem übernimmt, übergeben. Aus der Transformation ergibt sich der Abstand vom Mittelpunkt der Fußpunkte des *Nao*s zur Position des Balles auf dem Spielfeld. Die Position des Balles wird in *Webots* durch die Anpassung der *Proto*-Datei geändert. Zur vereinfachten Evaluation wurde die Spielfeldmitte als Fußmittelpunkt gewählt, damit Nullpunkt des Spielfeldkoordinatensystems und Fußkoordinatensystem übereinstimmen.

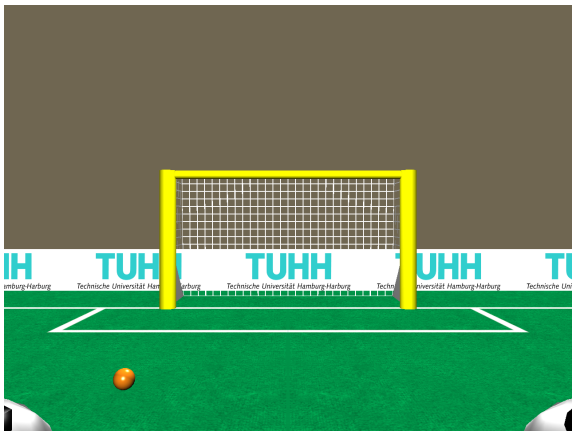

Abb. 6.2.: Das für die Positionsbestimmung in *Webots* genutzte Bild

Auf dem Bild wurden vom Simulationsalgorithmus die in Tabelle 6.2 dargestellten Werte ermittelt.

	x	y
Bildmitte nach Ballmitte [px]	−368	362
Bildmitte nach Ballmitte [°]	−18.70	−17.64
Brennweite [px]	1087.31	1138.48

Tab. 6.2.: Ergebnis der Bildkoordinatenberechnung in *Matlab*

Die vorher im Proto definierten Werte für die Position des Balles sind in Tabelle 6.3 aufgeführt. Hierbei ist die x-Achse vom Tor in die Richtung des *Nao*, die y-Achse vom Spielfeld nach links und die z-Achse vom *Nao* nach oben ausgerichtet.

Koordinatensystem des Spielfeldes	x	y	z
Proto-Werte des Ballmittelpunktes	$-1,600$	$0,500$	$0,032$
Berechnete Werte des Testalgorithmus in *Matlab*	$-1,601$	$0,522$	$0,032$

Tab. 6.3.: Ergebnis der Positionsberechnung in *Matlab*

6.1.2. Bewertung

Die Farberkennung und die Verarbeitung der binären Bilddaten stellt in der Simulation eine zuverlässige Methode dar, den Ballmittelpunkt auf einem Bild zu finden. Die horizontalen bzw vertikelen Winkel von dem Bildmittelpunkt zur Ballmitte werden sowohl rohdatenunabhängig als auch auflösungunabhängig von dem Algorithmus ermittelt. Wie aus Tabelle 6.1 zu entnehmen ist, schwankt die Bearbeitungszeit stark mit der Größe der Rohdaten, hauptsächlich verursacht durch die Konvertierung der Bildaten in den YUV-Farbraum. Da die Konvertierung der Bilder bei der Implementierung des Moduls für den *Nao* wegfällt, sollte die Ballerkennung in vollem Umfang, so wie die in der Simulation durchgeführt worden ist, auf dem *Nao* in unter einer Sekunde auszuführen sein.

Da Fehler in der Koordinatentransformation oftmals Folgefehler einer schlechten Ballerkennung sein können, kann über die Sicherheit der simulierten Transformation nur vernünftig geurteilt werden, wenn adequate Grunddaten vorliegen. Hier ist das der Fall und wie der Tabelle 6.3 zu entnehmen ist, fallen die Differenzen zwischen errechneten und vorgegebenen Daten sehr gering aus. Die Ungenauigkeit von 2.2 cm in der y-Koordinate bei einer Entfernung von 1.6 m in der x-Koordinate ist durchaus annehmbar.

6.2. Implementierung

Die Implementierung eines Moduls, welches in der Lage ist, einen roten Ball aus einem, mit der Stirnkamera des *Nao* aufgenommenen, Bild nach der Farbe zu filtern und mit einer Faltung zu glätten, ist gelungen. Dem Modul ist außerdem möglich, Bildkoordinaten, sobald diese erhalten worden sind, in das dreidimensionale Spielfeld

umzurechnen. Eine Abschätzung der Form des erkannten roten Gegenstandes wurde noch nicht implementiert. Im Nachfolgenden wird nun an einigen Beispielen gezeigt, dass der entwickelte Algorithmus dennoch durchaus in der Lage ist Ballpositionen auf dem Spielfeld zu ermitteln. Den Berechnungen zu Grunde liegen werden mehrere, mit der Kamera des *Nao* und unter realen Bedingungen aufgezeichnete, Bilder im *YUV*-Farbraum.

6.2.1. Ergebnisse

Zur Berechnung der in den folgenden Abbildungen gezeigten Originalbildern und den dazu gehörigen, nach der Filterung und Faltung entstandenen, binären Bildern wurde eine Testumgebung genutzt, in der auf einem minimalistisch nachgebautem Spielfeld, Koordinatenpunkte festgelegt worden sind. Die Bilder wurden, in der Regel, vom Koordinatenursprung (*NN*) aus aufgenommen, Ausnahme ist Abbildung 6.6, bei der der *Nao* auf der y-Achse um 600 *mm* in positiver Richtung verschoben wurde. Das genutzte Koordinatensystem ist genauso aufgebaut, wie das in der Testumgebung *Webots* genutzte: Die x-Achse zeigt vom Tor zum *Nao*, die y-Achse ist nach Links gerichtet und die z-Achse nach oben.
Abbildung 6.3 zeigt das erste Beispiel eines gefilterten und gefalteten Bildes aus einer realen Umgebung bei einem Abstand von 1.4 m in x-Richtung. Es fällt direkt auf, dass nicht nur der Ball erkannt wird, sondern auch sämtliche anderen Rotwerte.

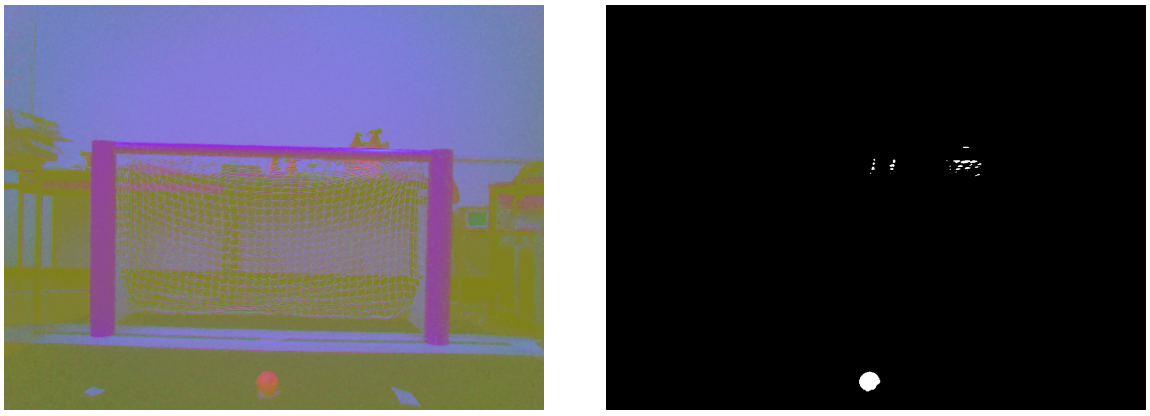

Abb. 6.3.: Implementierung – Ball mittig

Die mitgefilterten roten Elemente in den aufgenommenen Bildern hätten in der Theorie keinen Einfluss auf die Filterung und Faltung, sobald eine Formerkennung mit implementiert ist. Praktisch ergibt sich auch kein Nachteil, solange der Ball nahe am *Nao* positioniert ist. In Abbildung 6.4 ist ein weiteres Beispiel einer Filterung und Faltung mit einem realen Bild dargestellt. Hier beträgt der Abstand von dem *Nao*, also von *NN*, zum Ball, wie auch in der Tabelle 6.4 aufgelistet, in Richtung der x-Achse 1.4 m und in Richtung der y-Achse 0.5 m.

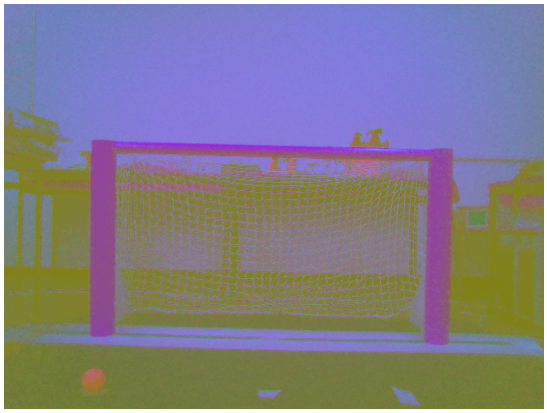
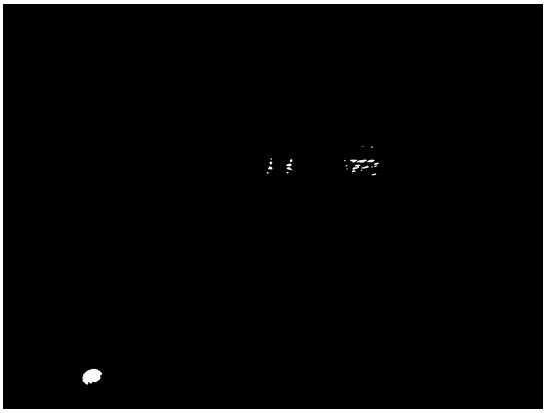

Abb. 6.4.: Implementierung – Ball vorne links

In Abbildung 6.5 ist noch einmal eines der gefilterten und gefalteten Bilder gezeigt. Dieses Mal ist der Ball in der x-Koordinate um 1.9 m und in der y-Koordinate um 0.46 m verschoben. An dieser Stelle ist im Hintergrund die Fehlfilterung eines Objektes, welches nicht dem Ball entspricht, zum dritten Mal wiederholt aufgetreten (siehe Kapitel 7.2.2).

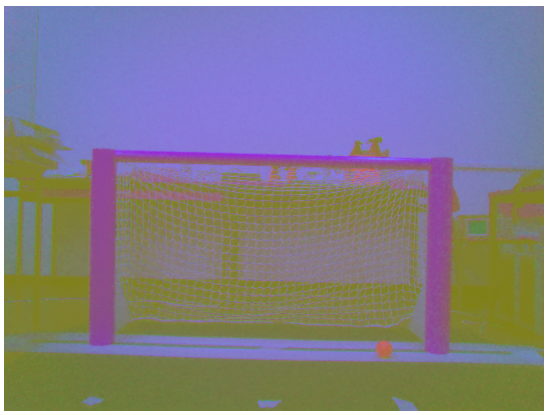
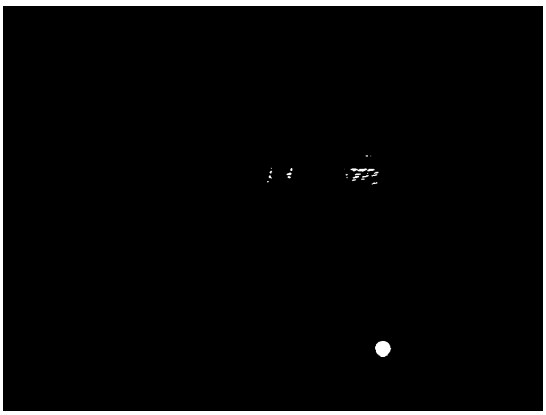

Abb. 6.5.: Implementierung – Ball hinten rechts

Die äußere Einflussnahme auf die Position des *Nao*s von dem Koordinatenursprung weg entlang der y-Achse hat auf die Berechnung der relativen Position des Balles zum *Nao* selbstverständlich keinen Einfluss. Tortzdem wird in Abbildung 6.6 deutlich, wie stark der Einfluss eines roten Objektes werden kann. Bei der Entwicklung eines Moduls zur Ballfilterung nach Farben muss nicht nur die Objekterkennung eines Kreises sondern auch die Stellen, an denen gesucht wird, beachtet werden. Abhängig von der Neigung des Kopfes muss langfristig eine obere Schranke für das Erkennen von Objekten, die als Ball gelten könnten, festgesetzt werden (siehe Kapitel 7.2.2). In diesem Bild ist eine Distanz vom *Nao* zum Ball von 0.6 m in y-Richtung und von 1.7 m in x-Richtung Grundlage der Berechnung. Die Auswertung der Berechnungen erfolgt wie bei den Abbildungen zuvor auch in der Tabelle 6.4.

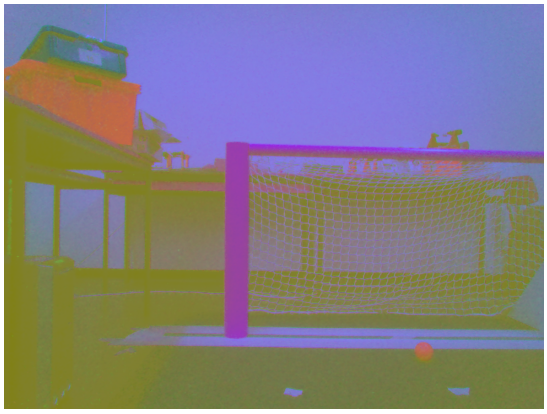


Abb. 6.6.: Implementierung – störende Objekte

In dem letzten Beispiel wurde das Spielfeld mit drei *Hockey Bällen* überladen. Die Positionen hier sind in (x/y)-Koordinaten jeweils in dem Bild von links nach rechts in m vorgegeben mit $(-1.86/0.4)$, $(-2.0/0.0)$ und $(-1.4/-0.4)$. Die Koordinaten beziehen sich auf das Koordinatensystem des Spielfeldes. Welcher Ball als Objekt für die Positionsbestimmung und Koordinatentransformation genutzt wird, bleibt jedoch nicht dem Zufall überlassen. Wie oben schon vermerkt, wird das Objekt als Ziel erkannt, welches am nähesten zum unteren Bildrand positioniert ist.

Dies ist nicht die optimalste Lösung, kommt aber der Methode, wie sie momentan an der TUHH verwendet wird, erst in unmittelbarer Nähe nach dem Ball zu suchen, am nähesten. Bei mehreren, als gleichwertige Bälle erkannten, Objekte auf dem Spielfeld ist die Nachvollziehbarkeit einer Suche nach einem nahen Objekt trivial.

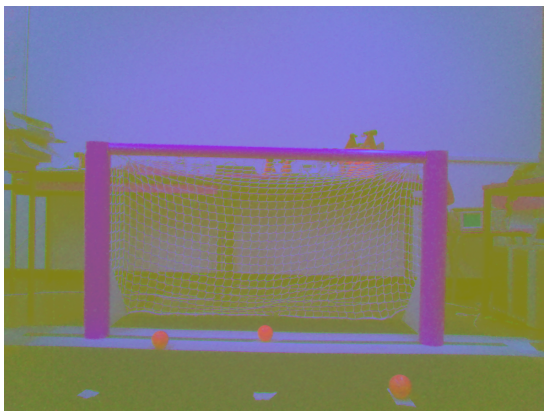
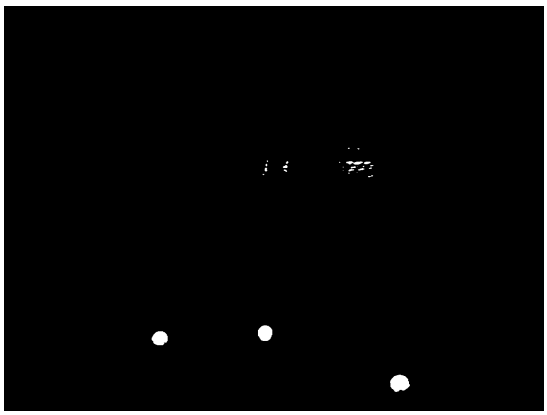

Abb. 6.7.: Implementierung – mehrere *Hockey Bälle*

In Tabelle 6.4 sind die errechneten Abstände in x- und y-Koordinaten vom *Nao*-Roboter zum Ball auf dem Spielfeld aufgelistet. Die Wege sind im Koordinatensystem des Spielfeldes angegeben. Der Ballmittelpunkt, also die Höhe der z-Koordinate, wurde in jeder Berechnung mit dem korrekten Wert 32.47 *mm* angegeben und ist daher nicht in der Tabelle mit angeführt. Die Bildkoordinaten wurden zur manuellen Überprüfung des Ergebnisses der Filterung und Faltung mit angegeben, da von

Ungenauigkeiten der Bildkoordinaten auf Fehler in der Genauigkeit der Ballpositionsbestimmung geschlossen werden kann.

	Bildkoordinaten		**Weg berechnet**		**Abstand real**	
	x in $[px]$	y in $[px]$	x in $[mm]$	y in $[mm]$	x in $[mm]$	y in $[mm]$
Abb. 6.3	622	915	−1526.3	85.5	−1400.0	0.0
Abb. 6.4	−440	421	−1573.9	−483.1	−1400.0	−500.0
Abb. 6.5	264	351	−1872.5	543.4	−1900.0	460.0
Abb. 6.6	353	355	−1851.2	684.3	−1700.0	600.0
Abb. 6.7	290	434	−1529.7	445.3	−1400.0	400.0

Tab. 6.4.: Ergebnis der Positionsberechnung auf dem *Nao*

Aus der Tabelle 6.4 ergeben sich für die angestellten Messungen die in Tabelle 6.5 die Beträge der absoluten und relativen Fehler.

	Fehler absolut		**Fehler relativ**	
	x in $[mm]$	y in $[mm]$	x	y
Abb. 6.3	126.3	85.5	0.090	–
Abb. 6.4	173.9	16.9	0.124	0.034
Abb. 6.5	27.5	83.4	0.014	0.181
Abb. 6.6	151.2	84.3	0.089	0.141
Abb. 6.7	129.7	45.3	0.093	0.113

Tab. 6.5.: Beträge der absoluten und relative Fehler bei der Positionsberechnung

6.2.2. Bewertung

Wie die Ergebnisse zeigen funktioniert die Erkennung von Rotwerten über den *YUV*-Farbraum auch in der realen Testumgebung gut. Die Faltung bringt eine geringe Glättung, nimmt aber keinen großen Einfluss auf das gefilterte Bild. Der Hintergrund dieses Umstandes ist vermutlich die Form der Faltungsmatrix, die verwendet wurde. Der Gedanke einer Glättung und Abrundung des als Ball erkannten Objekts bleibt ein möglicher und korrekter Weg, allerdings muss die Form der Matrix zur Faltung angepasst werden.
Da noch keine Form der Kreiserkennung mit eingebunden wurde, kann an dieser Stelle auch diesbezüglich keine Bewertung erfolgen. Allerdings bleibt die Notwendigkeit dieser Implementierung auf langfristige Sicht, gerade im Hinblick auf die Abbildung 6.6, bestehend.

Die im Rahmen der oben aufgelisteten Beispiele durchgeführten Berechnungen liefern, wie letztendlich der Tabelle 6.4 auch zu entnehmen ist, ein positives Ergebnis. Die teilweise bis zu 17 *cm* vom Originalwert abweichenden berechneten Werte in der x-Koordinate lassen sich mit großer Wahrscheinlichkeit auf kleinere Ungenauigkeiten in den acht Motoren, welche bei der Koordinatentransformation durchlaufen

werden, zurückführen. Bei dem Testen mit dem *Nao* führen auch kleinere Regelungen an der Neigung des Kopfes zu Ausreißern in der Berechnung. Ob Änderungen der Kopfneigung stärker oder schwächer in die Variation der Ergebnisse eingehen, wurde nicht untersucht.
Die Abweichungen in der y-Koordinate sind mit Werten in maximalen Bereichen von bis zu 9 *cm* zwar groß, aber auch erklärbar. Eine mögliche Fehlerquelle ist hier die nicht genau ausrichtbare Position des *Nao*-Roboters zum Koordinatensystem des Spielfeldes. Außerdem kann, wie bei Fehlern in der x-Koordinate auch, die Rückgabe der Ballfilterung eine Rolle spielen, da in dem verwendeten Algorithmus nur die rechte unteren Ecke des Objektes, welches am unteren Rand des Bildes als roter Ball erfasst wurde, als Ballmitte behandelt wird.

Die Algorithmen für den *Nao*, mit denen in der Evaluation gearbeitet wurde und wie sie im Model für die Erkennung in Lokalisation des Balles in *C++* entwickelt wurden, sind optimierugnsfähig und stellen keine endgültige Implementierung dar. Außerdem wurde bei den Messungen bis hierhin nur der entfernte Bereich des Fußballfeldes untersucht. Während den Erarbeitung der Ergebnisse wurden auch Bilder mit der unteren Kinnkamera des *Nao* unternommen, da allerdings in dem Modul noch keine Methoden für die Koordinatentransformationen der Kinnkamera implementiert ist, fielen die Ergenisse der Berechnungen mit der unteren Kamera natürlich falsch aus und wurden daher nicht in die Evaluation aufgenommen.

7. Zusammenfassung und Ausblick

In diesem Kapitel findet sich nach der Zusammenfassung in 7.1 ein abschließender Ausblick auf weitere Entwicklungsmöglichkeiten im Bereich der Erkennung und Lokalisation eines nicht statischen Objektes gegeben in 7.2.

7.1. Zusammenfassung

Der Grundstein für die stetige Weiterentwicklung der Qualität der *Standard Platform League* des *RoboCup* hängt mit den Reaktionen der *Nao*s auf ihre sich meist bewegende Umwelt zusammen. Und da die meisten Informationen über das Umfeld des *Nao*s über die Kameras ermittelt werden, stellt die Weiterentwicklung der Bildverarbeitung einen entscheidenden Grundpfeiler dar.

Dieser Gedanke wurde mit dem Ziel dieser Bachelorarbeit aufgegriffen, ein Modul zu entwickeln, welches dem *Nao* erlaubt in seinem Umfeld ein nicht statisches Objekt mit seiner Kamera aufzunehmen, um im Anschluss auf die Präsenz eines roten *Hockey Balles* zu filtern und aus dem Ergebnis die Position des Balles im Raum realtiv zu sich selbst zu berechnen. In diesem Bestreben wurde zunächst eine Simulation entworfen, in welcher die wesentlichen Schritte des Algorithmus getestet werden können. Im weiteren Verlauf wurde das Modul schließlich für den *Nao* entwickelt, untersucht sowie ausgewertet und stellt nun eine erste Implementierung des Moduls einer Ballerkennung und Lokalisation dar, die unter Einfluss der richtigen äußeren Umstände zufriedenstellende Ergebnisse liefert.

7.2. Ausblick

Ansätze, an denen die Entwicklung eines sicheren Modules für den *Nao* weitergeführt werden kann, sind unter Anderem die Steigerung der Performance durch *Downsizing* der Auflösungen der verarbeiteten Daten oder die Entwicklung einer soliden Formerkennung in *C++* unter Nutzung von *OpenCV*.

7.2.1. Variation der Auflösung

In dieser Ausarbeitung wurde unter *Webots* und *Matlab* sowie dem *Nao* selbst größtenteils nur mit HD Bildern gearbeitet. Wie aber schon ein der Simulation evaluiert, gibt es deutliche Möglichkeiten der Performancesteigerung bei Variation der Auflösung der als Bild vorliegenden Grunddaten, ohne dass Qualitäten in der weiteren

Verarbeitung eingebüßt werden müssen. Die Ergebnisse werden auch bei einer Auflösung von 640 × 480 Pixel noch sicher berechnet (vgl. Tabelle 6.1 aus Kapitel 6.1.1).

7.2.2. Formerkennung in C++

Da die Formerkennung des Balles in seiner Rolle als rundes Objekt weder in der Simulation noch in dem lauffähigem Modul eingebunden wurde, besteht auch hier Entwicklungspotential, an welchem weiterentwickelt werden kann. Der entworfene Algorithmus funktioniert unter gegebenen Voraussetzungen bis hierhin gut, fängt allerdings keine Sonderfälle, wie zum Beispiel mehrere *Hockey Bälle* auf dem Spielfeld, ab. Außerdem können als rot erkannte Flächen störende Einflüsse auf die Bildverarbeitung haben.
Des Weiteren können Methoden zur Sicherheitssteigerungen, z.B. im Bereich Eckenbzw. Kantendetektion eingebunden werden, damit nach der Faltung und der resultierenden glättenden Wirkung, kantige Objekte als potentielle Ballkandidaten ausgeschlossen werden können. Eine große Herausforderung ergibt sich auch noch nach der Formerkennung, bei der Überlegung, die Formerkennung mit in die Distanzabschätzung zum Ball miteinfließen zu lassen. Da die Kamera des *Nao*s nicht in der Lage ist, zu Zoomen, kann ein erkannter und in seiner realen Größe zurückgerechneter Durchmesser eines *Hockey Balles* hilfreich bei der Berechnung der Entfernung zum Ball sein.

7.2.3. Bewegte Objekte

Bisher wurden bei der Objekterkennung nur nicht statische Objekte in fixen Positionen verwendet. Vor allem beim Fußballspielen kommt es nur selten vor, dass der Ball in einer ruhenden Position verweilt. Daher muss auf längere Sicht eine Methode entwickelt werden, mit der auch Bewegungen wahrgenommen werden können. Ein erster Schritt in diese Richtung wäre eine temporäre Speicherung mehrerer Positionspunkte, mit denen bei Änderung der Position dann sogar abgeschätzt werden könnte, wie schnell bzw. wohin der Ball sich bewegen würde. Ein planbare Positionierungsbewegung des Balles könnte dann als Grundstein für eine bis jetzt in der *Standard Platform League* noch als utopisch geltende Passspielsituation der *Nao*s untereinander verwenden werden.

A. Inhalte der CD

- Bericht,
- Literatur,
- Quellcode.

Literaturverzeichnis

[Bra12] Bradski, Gary: *OpenCV Wiki*. http://opencv.willowgarage.com/wiki. Version: 2012. – [Online; zugegriffen 05-Okt-2012]

[Che12] Chemnitz, TU: *Technische Universität Chemnitz*. http://www.tu-chemnitz.de/. Version: 2012. – [Online; zugegriffen 13-Okt-2012]

[Cyb12] Cyberbotics: *Webots Reference Manual*. http://www.cyberbotics.com/. Version: 6.4.4, 2012. – [Online; zugegriffen 02-Okt-2012]

[Fis97] Fisher, Bob: *3x4 Projection Matrix*. http://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/EPSRC_SSAZ/node3.html. Version: 1997. – [Online; zugegriffen 10-Okt-2012]

[Gri08] *Kapitel* 5. In: Grigat, Prof. Dr.-Ing. Rolf-Rainer: *Digital Image Processing*. TU Vision, Technische Universität Hamburg-Harburg, 2008, S. 154ff

[Inc12] Inc., Wikimedia F.: *Wikipedia, Die freie Enzyklopädie*. http://de.wikipedia.org/wiki/. Version: 2012. – [Online; zugegriffen 08-Okt-2012]

[Inf12] Informatik, FB3 M.: *RoboCup Stadard Platform League*. http://www.tzi.de/spl/bin/view/Website/WebHome. Version: 2012. – [Online; zugegriffen 08-Okt-2012]

[Jac01] *Kapitel* 3. In: Jack, Keith: *Video demystified: a handbook for the digital engineer*. LLH Technology Publishing, 2001, S. 15ff

[McA04] *Kapitel* 7. In: McAndrew, Alisdair: *An Introduction to Ditigal Image Processing with MATLAB*. School of Computer Science and Mathematics, Victoria University of Technology, 2004, S. 62

[Pet06] *Kapitel* 3. In: Peterwitz, Julia: *Grundlagen Bildverarbeitung und Objekterkennung*. Technische Universität München, 2006, S. 7ff

[Rob12] Robotics, Aldebaran: *Nao User Guide*. http://www.aldebaran-robotics.com/documentation. Version: 1.12.5, 2012. – [Online; zugegriffen 01-Okt-2012]

[SÓ9] Sánchez, Tomás G.: *Artificial Vision in the Nao Humanoid Robot*, Universitat Rovira i Virgili, master thesis, 2009

[TM12] The MathWorks, Inc.: *Image Processing Toolbox Users Guide*. `http://www.mathworks.de/help/pdf_doc/images/images_tb.pdf`. Version: R2012b, 2012. – [Online; zugegriffen 04-Okt-2012]