TECHNISCHE UNIVERSITÄT DRESDEN

FAKULTÄT ELEKTROTECHNIK UND INFORMATIONSTECHNIK

Institut für Automatisierungstechnik

# Studienarbeit

**Thema: Inbetriebnahme sowie Untersuchung einer kommerziellen Roboterplattform**

vorgelegt von Michael Voigt

Betreuer: Dipl-Ing. Sylvia Horn

Verantwortlicher Hochschullehrer: Prof. Dr. tech. Klaus Janschek

TECHNISCHE UNIVERSITÄT DRESDEN

**Fakultät Elektrotechnik und Informationstechnik**

Institut für Automatisierungstechnik

# Aufgabenstellung zur Studienarbeit

für

**Herrn Michael Voigt**

## Inbetriebnahme sowie Untersuchung einer kommerziellen Roboterplattform

Zielsetzung:

Um weiterführende Experimente am Institut vorzubereiten, soll innerhalb dieser Arbeit die kommerzielle Roboterplattform AIBO von Sony in Betrieb genommen werden und deren Nutzungsmöglichkeiten einer weit reichenden Prüfung unterzogen werden. Dies umfasst unter anderem die Einarbeitung in die Entwicklungsplattform AIBO SDE von Sony, die Programmierung eines Fernüberwachungstools sowie die eines Navigationsmoduls.

Das Fernüberwachungstool soll hierbei vor allem dazu beitragen, Aussagen über mögliche Stärken bzw. Schwächen der Plattform treffen zu können. Großes Interesse gilt hierbei vor allem den Eigenschaften der vorhanden Sensorik bzw. Aktorik. Das Navigationsmodul dient hierbei als Grundlage für spätere Implementierungen. So soll es die Plattform mit der Fähigkeit ausstatten, Hindernissen dynamisch auszuweichen sowie benutzerdefinierte graphische Landmarken aufzufinden.

Folgende Aufgaben sind zu lösen:

- Erstellen der Anforderungsdefinition
- Inbetriebnahme der Roboterplattform AIBO von Sony
- Literaturrecherche zu vorhandenen Steuerungsalgorithmen des AIBO
- Einarbeitung in die AIBO SDE Entwicklungsplattform von Sony
- Programmierung eines Fernüberwachungstools
- Test und Einschätzung der vorhandenen Sensorik/Aktorik
- Entwurf eines Navigationsmoduls
- Umsetzung der Algorithmen sowie Funktionstest
- Kritische Analyse möglicher Schwachstellen der Roboterplattform sowie der entstandenen Algorithmen

Prof. Dr.techn. K.Janschek
Verantwortlicher Hochschullehrer

Betreuer: Dipl.-Ing. S.Horn
Bearbeitungszeitraum: 21.03.2005 - 15.07.2005

# Thesen zur Studienarbeit

1. Für die Implementierung komplexer Steuerungsaufgaben sind Robotersysteme, die wie Aibo mit einem Multitasking-Betriebssystem ausgestattet sind, gegenüber rein sequentiell gesteuerten Robotern trotz der längeren Einarbeitungszeit klar im Vorteil. Ein Echtzeitscheduler in Aibos Betriebssystem ist aber wünschenswert.

2. Durch Experimente wurde bestimmt, dass Aibos Kopfabstandssensor für nahe Entfernungen in einem horizontalen Winkel von -3,4 Grad und sein Kopfabstandssensor für weite Entfernungen in einem Winkel von -3,6 Grad im mathematisch positiven Sinn abstrahlt.

3. Die für Aibos Kopfabstandssensoren bestimmten Verteilungsfunktionen der Sensorfehler können in guter Näherung nicht, wie einige andere Rauschgrößen, durch eine Normalverteilung modelliert werden.

4. Bei dem im Rahmen dieser Arbeit entworfenen Navigationsmodul wird angenommen, wenn beim Anlaufen einer gesichteten Landmarke durch Aibos Sensoren ein Hindernis festgestellt wird, dass es sich dabei um die Landmarke handelt. Wirklich sinnvoll könnte dieser Entwurfsfehler nur durch ein globales Lokalisierungsverfahren verhindert werden.

5. Beim Anlaufen einer gesichteten Landmarke wird die momentan zu rotierende Winkelgeschwindigkeit proportional zur aktuellen Winkelabweichung bestimmt. Eine Sicherung des Kreises im regelungstheoretischen Sinne konnte nicht realisiert werden, weil die Übertragungsfunktion des inneren Kreis, der Servomotoren von Aibo, nicht bekannt ist. Obwohl sich der verwendete Algorithmus als sehr funktionstüchtig herausgestellt hat, ist eine Bestimmung der Übertragungsfunktion von Aibos Servomotoren sehr wünschenswert.

6. Zur Modellierung des gewünschten Verhaltens des Navigationsmoduls wurden Zustandsdiagramme, die mit dem bei dieser Arbeit verwendeten Tekkotsu-Framework direkt implementiert werden können, benutzt. Sie stellen ein sehr brauchbares Instrumentarium für einen übersichtlichen und modularen Entwurf komplexer Steueralgorithmen dar.

7. Die Bestimmung einer Odometrie für Laufroboter wie Aibo ist um ein Vielfaches komplizierter als bei radgetriebenen Plattformen. Ein Rückgekoppelter Regelkreis zur Ansteuerung von Aibos Laufverhalten ist fast undenkbar.

TECHNISCHE UNIVERSITÄT DRESDEN

**Fakultät Elektrotechnik und Informationstechnik**

Institut für Automatisierungstechnik

# Inbetriebnahme sowie Untersuchung einer kommerziellen Roboterplattform

## Kurzfassung

Durch die Veröffentlichung der kostenlosen Software-Schnittstelle OPEN-R SDK hat sich der Unterhaltungsroboter Aibo von Sony in den letzten Jahren zu einer anerkannten Plattform im Bereich der Robotikforschung avanciert. Das Anliegen der Arbeit ist es, diese am Institut für Automatisierungstechnik neu angeschaffte kommerzielle Roboterplattform in Betrieb zu nehmen und einer umfassenden Untersuchung bezüglich ihrer verschiedenen Einsatzmöglichkeiten zu unterziehen.

Dazu werden zu Beginn der Arbeit die vielen verschiedenen Programmierumgebungen für Aibo detailliert vorgestellt und ihre konzeptionellen Besonderheiten herausgearbeitet. Im Anschluss daran wird begründet, warum bei dieser Arbeit die Wahl auf das Tekkotsu-Framework gefallen ist. Dieses bietet eine allgemeine Methodik zur Berechnung des Bewegungsverhaltens von Aibos Gliedern in Bezug zu einem Referenzpunkt in seiner Körpermitte. Es wird die Funktionsweise dieser Methodik erläutert.

Weiterhin wurden in Experimenten die zwei Infrarotabstandssensoren an Aibos Kopf auf ihre Genauigkeit hin in Abhängigkeit von der Position eines Gegenstands in Aibos Gesichtsfeld getestet. Die Ergebnisse dieser Untersuchungen werden dargestellt und ausgewertet.

Zur praktischen Demonstration der dargestellten theoretischen Sachverhalte wurde mit dem Tekkotsu-Framework ein Navigationsmodul entwickelt, dass Aibo mit der Fähigkeit ausstattet, farbige Landmarken im Raum aufzufinden und anzulaufen, ohne dabei mit Hindernissen in Kollision zu geraten.

Außerdem wurde mit Visual C++ 6.0 ein Fernüberwachungstool programmiert, das die momentan von Aibo gemessenen Sensorwerte am Bildschirm darstellt.

| | |
|---|---|
| Betreuer: | Dipl.-Ing. Sylvia Horn |
| Hochschullehrer: | Prof. Dr. tech. Janschek |
| Bearbeitungszeitraum: | 21.03.2005 bis 15.08.2005 |

TECHNISCHE UNIVERSITÄT DRESDEN

**Fakultät Elektrotechnik und Informationstechnik**

Institut für Automatisierungstechnik

# Inbetriebnahme sowie Untersuchung einer kommerziellen Roboterplattform

## Abstract

Due to the release of the free software interface OPEN-R SDK the entertainment robot Aibo by Sony has developed to a well-accepted platform in the field of robotics research. It is the purpose of this paper to investigate this robot platform with respect to different possibilities of use.

Therefore, at the beginning of the thesis, the many existing programming environments for Aibo are presented in detail with respect to their conceptional specialties. In the sequel we give reasons why we chose the Tekkotsu-Framework as a platform of development. This platform offers a general approach for the computation of the kinematical properties of Aibo's links referring to a reference point in the middle of its body. This approach is explained in detail.

Furthermore in experiments the two infrared distance sensors on Aibo's head had been tested with respect to their accuracy in dependency of the position of an obstacle in Aibo's field of view. The results of these measurements are presented and evaluated.

To demonstrate the presented theoretical aspects in a practical way a navigation module has been developed which gives the platform the ability to find colored landmarks in a room autonomously without clashing with obstacles.

In addition a remote supervision tool was programmed with Visual C++ 6.0 which shows the actual sensor values of Aibo on the screen.

# Inhaltsverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Einordnung der Arbeit

Anfang der neunziger Jahre begann eine Forschungsgruppe des Roboterentwicklungslabors *Sony Digital Creatures Lab* unter Leitung von Dr. Doi mit ersten Experimenten, ein für den Anwenderbereich geschaffenes Roboterspielzeug zu entwickeln, das sich autonom in seiner Umgebung bewegen und mit Menschen in Kontakt treten können soll. Nachdem von Sony bereits zwei Protoypen von vierbeinigen Roboterhaustieren vorgestellt worden sind, wurden im Jahr 1999 dann die ersten Modelle ERS-110 und ERS-111 des Roboterhundes *Aibo* veröffentlicht. *Aibo* ist eine Abkürzung für *A*rtifcial *I*ntelligence Ro*bo*t und bedeutet auf japanisch *Freund* oder *Gefährte*. In den Jahren 2000, 2001 und 2003 folgten dann die Veröffentlichungen der Modelle ERS-210, 220 sowie des Sony Aibo ERS-7.

Noch vor seiner offiziellen Veröffentlichung, ist im RoboCup – dem Wettkampf, bei dem verschiedene Universitäten gegeneinander im Roboterfussball antreten – im Jahr 1998 bereits eine eigene Liga für Aibo namens *Sony Four-Legged Robot League* eingeführt worden. In dieser Liga spielen zwei Teams von je vier Aibos gegeneinander mit einem orangefarbenen Ball Fussball. Bereits ab diesem Zeitpunkt konnten sich einige Universitäten mit Aibo als Roboterplattform beschäftigen; seine Programmierung war aber diesen wenigen Bildungseinrichtungen, die am RoboCup teilnehmen konnten, vorbehalten. Seit Sony aber im Jahr 2002 die Sotfware-Schnittstelle OPEN-R SDK kostenlos zur Verfügung gestellt hat, ist ein enormes Interesse der Robotikforschung an der Plattform Aibo entstanden und sie konnte sich so zu einer der kommerziellen Standardplattformen zum Einsatz im universitären Bereich avancieren.

Das Anliegen der vorliegenden Studienarbeit ist es, die eben beschriebene, am Institut für Automatisierungstechnik neu angeschaffte Roboterplattform Aibo ERS-7 in Betrieb zu nehmen und einer umfassenden Untersuchung hinsichtlich

ihrer verschiedenen Einsatzmöglichkeiten zu unterziehen. Eine ausführliche Beschreibung und Interpretation der Aufgabenstellung ist in Kapitel 2 dargelegt.

## 1.2 Aufbau der Arbeit

Es soll hier nun ergänzend als Bindeglied zwischen Inhaltsverzeichnis und Kurzfassung ein knapper Leitfaden zur Navigation durch die vorliegende Arbeit angegeben werden. Zu Beginn der Arbeit wird in Kapitel 2 zur Konkretisierung der Aufgabenstellung eine allgemeine Anforderungsanalyse durchgeführt. In Kapitel 3 folgt dann eine detaillierte Übersicht über die wichtigsten Programmierumgebungen, die momentan für den Roboterhund Aibo zur Verfügung stehen. Dieses Kapitel wird durch eine Schlussfolgerung und die Begründung, warum sich hier für das Tekkotsu-Framework entschieden wurde, abgeschlossen. Die im Rahmen dieser Arbeit durchgeführten Experimente werden in Kapitel 4 vorgestellt und ausgewertet. In Kapitel 5 wird dann die vom Tekkotsu-Framework zur Verfügung gestellte Methodik zur Berechnung des Bewegungsverhaltens von Aibos Gliedmaßen ausführlich erläutert, da sie auch für die Realisierung des in Kapitel 6 beschriebenen Navigationsmoduls zum Einsatz kam. Abgerundet wird die Arbeit durch den im letzten Teil des Kapitels 6 angegebenen Funktionstest des Navigationsmoduls.

# Kapitel 2

# Anforderungsdefinition

Um die Aufgabenstellung hinsichtlich ihrer konkreten Anforderungen zu analysieren und somit die Ziele der Arbeit unmissverständlich und detailliert darzulegen, wird in diesem Kapitel eine Anforderungsdefinition erstellt. Dazu werden zunächst im Abschnitt 2.1 die spezifischen Ziele der gesamten Arbeit in textueller Form erörtert. Da es sich bei dem zu entwerfenden Navigationsmodul um ein Softwareelement handelt, bietet es sich außerdem an, die Anforderungsanalyse dafür durch eine strukturierte Analyse nach Tom DeMarco [You89, DeM79] zu vertiefen.

Die strukturierte Analyse besteht prinzipiell aus einer Daten- und einer Steuerflussanalyse. Aufgrund der geringen Komplexität unserer Problemstellung beschränken wir uns im Rahmen dieser Arbeit allerdings auf die Datenflussanalyse. Der Ausgangspunkt dafür ist das im Unterpunkt 2.2.1 abgebildete Kontextdiagramm. Es beschreibt die Außensicht eines Nutzers auf das zu analysierende System und charakterisiert die gewünschte Interaktion des Systems mit seinen Terminatoren. Die in den Abschnitten 2.2.2, 2.2.3 und 2.2.4 dargestellten Datenflussdiagramme hingegen dienen dazu, die hierarchische Dekomposition des Systems in Subsysteme und somit die innere Struktur des System zu veranschaulichen. Die Ergebnisse der strukturierten Analyse bilden bereits eine solide Grundlage für den Entwurf des System.

## 2.1 Spezifische Anforderungen

Wie aus der Aufgabenstellung zu erkennen, ist es das wesentliche Ziel der Arbeit zu untersuchen, inwiefern die Roboterplattform Aibo für wissenschaftliche Experimente im Bereich der Robotik – insbesondere zum Test von Lokalisierungs- und Navigationsalgorithmen – geeignet ist.

Dieses Wissen soll einerseits durch eine Literaturrecherche zu Veröffentlichungen über Forschungen anderer Einrichtungen mit dem Aibo, andererseits

aber auch durch eigene Experimente zur Einschätzung der Sensorik und Aktorik gewonnen werden. Um diese Experimente zu ermöglichen, soll ein Fernüberwachungstool programmiert werden, das über die W-LAN-Schnittstelle die aktuellen Sensorinformationen und die Stellpositionen der Gelenke des Robotors am PC darstellt. Der Entwurf des Navigationsmoduls dient dazu, die gewonnenen Erkenntnisse zu erproben und die Roboterplattform einem weiteren Praxistest zu unterziehen.

Die mittlerweile große Vielfalt an unterschiedlichen Programmierumgebungen zur Steuerung und Überwachung des Aibo-Roboters bietet ein sehr breites Spektrum an Möglichkeiten, Software für ihn zu entwickeln. Es handelt sich dabei nicht nur um die von Sony veröffentlichten Plattformen wie das OPEN-R SDK, das Aibo Remote Framework oder die Skriptsprache R-CODE, sondern auch um Umgebungen anderer Anbieter, wie zum Beispiel das an der Carnegie Mellon University entwickelte Tekkotsu Framework. Eine weitere Aufgabe dieser Studienarbeit ist es, einen Überblick über die wichtigsten der verschiedenen Programmierumgebungen zu geben und sie bezüglich ihrer Einsatzfähigkeit in Gebieten der Robotik-Forschung zu beurteilen.

## 2.2 Strukturierte Analyse

### 2.2.1 Ebene 0: Kontextdiagramm

In Abbildung 2.1 erkennt man das Kontextdiagramm, das das zu entwerfende Navigationsmodul als einen einzigen Prozess F0 *navigiere Aibo* modelliert. Es

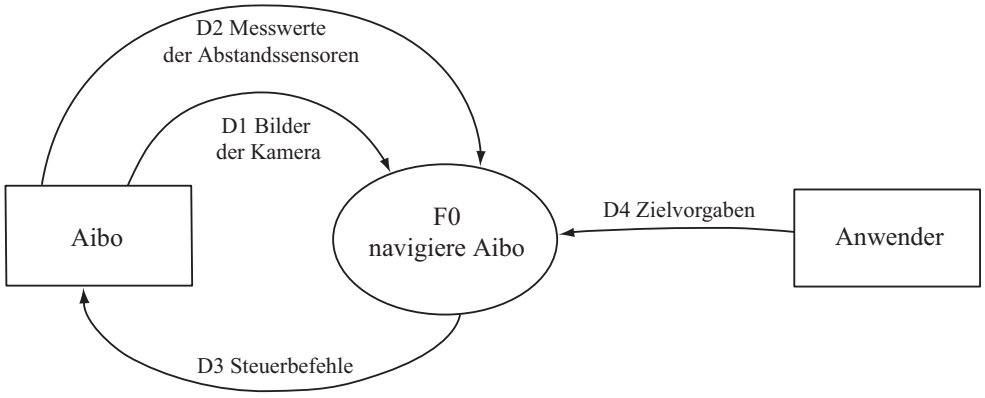



**Abbildung 2.1:** Kontextdiagramm des Navigationsmoduls.

folgt eine Erläuterung der im Kontextdiagramm dargestellten Datenflüsse:

**D1** Die CCD-Kamera des Aibo ERS-7 hat eine Auflösung von 416 (horizontal) x 320 (vertikal) Pixeln. Sie besitzt eine feste Brennweite von 3.27 mm und

eine Wiederholfrequenz von 30 Hz. Ihre Bilder werden zur Landmarkenerkennung verwendet.

**D2** Hierbei handelt es sich um die Werte der Infrarotsensoren. Davon befinden sich zwei am Kopf und einer im Brustbereich des Robotorhundes. Die Sensoren am Kopf haben einen Messbereich von 50 mm bis 500 mm beziehungsweise von 200 mm bis 1500 mm, der Brustabstandssensor hat eine Reichweite von 100 mm bis 900 mm.

**D3** Um Aibos Aktorik anzusteuern, müssen die gewünschten Werte für die Winkelstellung der Gelenke an das Betriebssystem des Roboters übergeben werden. Diese Werte werden dann durch das System als Sollwert der PID-Regelung des jeweiligen Gelenks eingestellt. In unserem Fall setzen sich die Steuerbefehle aus den Ansteuerbefehlen für den Kopf (D3.1) und aus den Befehlen zur Lauf- (D3.2) und Drehbewegung (D3.3) zusammen. Siehe dazu auch Abbildung 2.2.

**D4** Der Anwender gibt eine Folge von grafischen Landmarken vor, die in dieser Reihenfolge durch den Navigationsalgorithmus aufzufinden und anzulaufen sind.

### 2.2.2 Ebene 1: F0 *navigiere Aibo*

Die hierarchische Aufgliederung der Funktion F0 *navigiere Aibo* ist in Abbildung 2.2 als Datenflussdiagramm beschrieben. Es enthält folgende Funktionen:

**F0.1** Das aktuelle Bild der Kamera wird daraufhin untersucht, ob die zu findende Landmarke im Blickfeld des Aibo ist. Eine genauere Beschreibung der Bildauswertung befindet sich im Unterabschnitt 2.2.3.

**F0.2** Zur Hinderniserkennung werden die Daten der Abstandssensoren ausgewertet. Unterschreiten diese Werte eine bestimmte Schranke, so wird ein Hindernis vor dem entsprechenden Sensor angenommen.

**F0.3** Zum Auffinden der Landmarke ist vorgesehen, dass sich der Roboter immer wechselweise um einen zufällig ermittelten Weg geradeaus vorwärts bewegt, die Umgebung durch Drehen des Kopfes nach der Landmarke absucht und dann seinen Körper um einen zufällig bestimmten Winkel um seinen Mittelpunkt rotieren lässt. Ist die Landmarke gefunden, liefert die aktuelle Kopfneigung die Richtung, die auf dem Weg zu ihr einzuschlagen ist. Während der Roboter die Landmarke anläuft, wird periodisch erneut der Mittelpunkt des Blickfelds durch Drehung des Kopfes auf die Landmarke ausgerichtet.

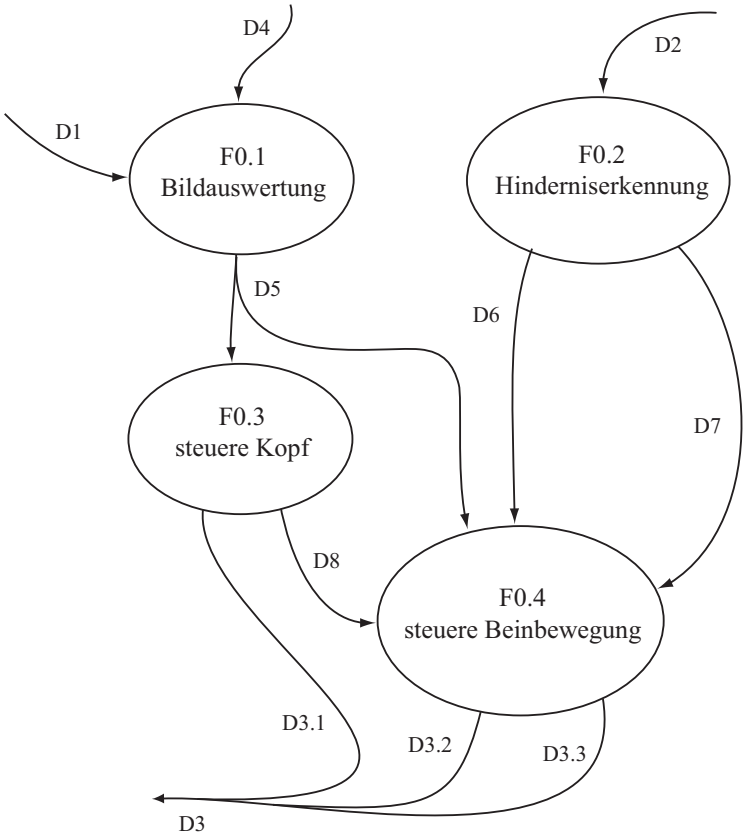



**Abbildung 2.2:** Ebene 1: F0 *navigiere Aibo.*

**F0.4** Die Bewegung der Beine setzt sich aus der Dreh- (D3.3) und aus der Laufbewegung (D2.3) zusammen. Diese Funktion ist genauer in Abschnitt 2.2.4 beschrieben.

Außerdem sind die folgenden Datenflüsse aufgeführt:

**D1-D4** Analog zu Abschnitt 2.2.1.

**D5** Es handelt sich hierbei um einen boolschen Wert, der angibt, ob die Landmarke im aktuellen Kamerabild entdeckt wurde oder nicht.

**D6/D7** geben an, ob ein Hindernis durch den Kopf-/Brustabstandssensor erkannt wurde. Befindet sich ein Hindernis vor dem Kopf und ist die Landmarke im Kamerabild erkannt worden, so wird angenommen, dass Aibo direkt vor ihr steht.

**D8** Aktueller Drehwinkel des Kopfes.

### 2.2.3 Ebene 2: F0.1 *Bildauswertung*

In Abbildung 2.3 wird die durch den Prozess F0.1 modellierte Bildauswertung de-

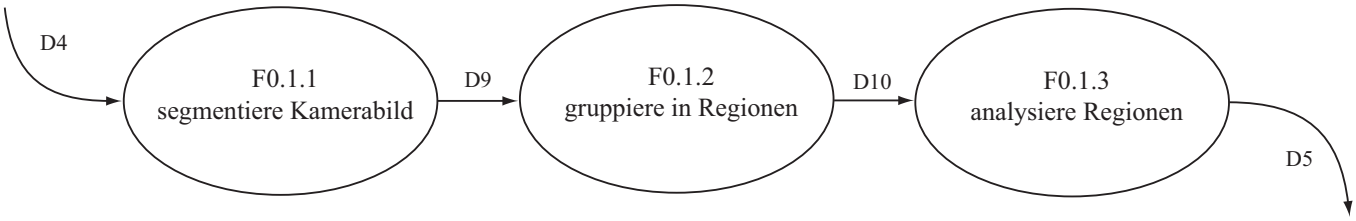



**Abbildung 2.3:** Ebene 2: F0.1 *Bildauswertung*.

taillierter charakterisiert. Sie besteht aus den Unterfunktionen F0.1.1 *segmentiere Kamerabild*, F0.1.2 *gruppiere in Regionen* und F0.1.3 *analysiere Regionen*. Dabei sind F0.1.1 und F0.1.2 bereits in dem hier verwendeten Tekkotsu-Framework implementiert, während die Funktion F0.1.3 im Rahmen dieser Arbeit realisiert werden soll. Die eben genannten Funktionen sind wie folgt beschrieben:

**F0.1.1** Die zu findenden Landmarken bestehen aus zwei sich übereinander befindenden, verschiedenfarbigen Rechtecken. Die Funktion F0.1.1 bestimmt für jedes Pixel des Kamerabilds, ob es einer der zu suchenden Farben zuzuordnen ist.

**F0.1.2** Durch den Prozess F0.1.2 werden die gefundenen Farbpixel in zusammenhängende Regionen gruppiert.

**F0.1.3** Der Prozess F0.1.3 untersucht die gefundenen Regionen daraufhin, ob es sich bei ihnen um die Hälfte einer Landmarke handeln kann und überprüft so, ob die zu findende Landmarke momentan in Aibos Kamerabild zu erkennen ist.

Das Diagramm in Abbildung 2.3 enthält die Datenflüsse D4, D5, D9 und D10:

**D4, D5** entsprechen den Datenflüssen D4 und D5 in Abbildung 2.2

**D9** enthält pro zu detektierender Farbe eine zweidimensionale binäre Matrix, die für jedes Pixel des Kamerabildes angibt, ob es der jeweiligen Farbe zuzuordnen ist.

**D10** besteht aus einer Menge von aus dem Kamerabild extrahierten Farbregionen.

### 2.2.4 Ebene 2: F0.4 *steuere Beinbewegung*

Es wird nun die Funktion F0.4 *steuere Beinbewegung* aus Abbildung 2.2 genauer spezifiziert. Sie spaltet sich, wie in Abbildung 2.4 zu erkennen, in die Funktionen F0.4.1 *steuere Laufbewegung* und F0.4.2 *steuere Drehbewegung* auf. Im Folgen-

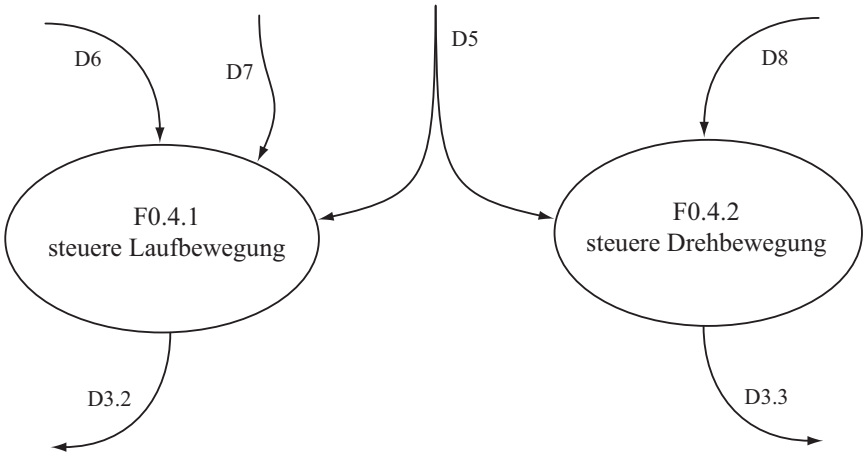



**Abbildung 2.4:** Ebene 2: F0.4 *steuere Beinbewegung*.

den sollen diese beschrieben werden (Die dargestellten Datenflüsse entsprechen Abschnitt 2.2.2):

**F0.4.1** Wenn die zu suchende Landmarke gefunden und von F0.3 das in dieser Arbeit nicht mitmodellierte *Fertig*-Steuersignal erhalten wurden, soll der Roboter solange in Richtung des Vektors, der auf die Landmarke deutet, laufen, bis er auf ein Hindernis trifft. Dann ist die Aufgabe beendet oder der Suchvorgang nach der nächsten Landmarke wird gestartet. In dem Fall, dass die Landmarke noch nicht gefunden und der *steuere Kopf*- und der *steuere Drehbewegung*- Vorgang beendet ist, soll die Plattform ein Stück geradeaus laufen. Dies wird allerdings abgebrochen, wenn ein Hindernis auftaucht.

**F0.4.2** Wenn die Funktion F0.3 das *Fertig*-Steuersignal sendet und die Landmarke noch nicht erkannt wurde, D5 also *false* ist, dann soll, wie in 2.2.2 beschrieben, eine Drehung um einen zufälligen Winkel vorgenommen werden. Ist die Landmarke erkannt worden, soll eine Drehung in die von D8 bestimmte Richtung vorgenommen werden.

# Kapitel 3

# Programmierumgebungen

Bei den im Jahr 1999 veröffentlichten, ersten Modellen ERS110 und ERS111 des Aibo handelte es sich damals um reine Anwender-Produkte. Die Möglichkeit, die Roboter zu programmieren, war nur priviligierten Bildungsinstituten vorbehalten, die am damaligen RoboCup teilnehmen durften. Als Sony 2002 allerdings die kostenlos verfügbare Programmierumgebung OPEN-R SDK für den Aibo zur Verfügung stellte, entstand ein großes Interesse der Robotik-Forschung an der Plattform Aibo, da seitdem die Möglichkeit besteht, diese zu untersuchen und eigene Experimente mit ihr durchzuführen. Es folgten im Jahr 2004 die Veröffentlichungen der Skriptsprache R-CODE und des Aibo Remote Frameworks. Aber auch andere Einrichtungen, wie zum Beispiel die ENSTA[1] oder die Carnegie Mellon University, entwickelten – aufbauend auf den von Sony veröffentlichten Plattformen – eigene Frameworks zur Aibo-Programmierung. Aufgrund dieser enormen Vielfalt an Entwicklungsmöglichkeiten wird in diesem Kapitel eine Übersicht über die wichtigsten Programmierumgebungen für den Aibo gegeben. In den Abschnitten 3.1 bis 3.5 werden zunächst einige Umgebungen einzeln ausführlicher präsentiert, in Abschnitt 3.6 werden dann weitere, hier nicht näher beleuchtete Plattformen kurz vorgestellt. Abschließend wird in 3.7 begründet, warum sich für die in dieser Arbeit zu erledigende Aufgabe für das Tekkotsu-Framework entschieden wurde.

## 3.1 Aibo Remote Framework

Das Aibo Remote Framework (im Folgenden mit ARF abgekürzt) ist ein Entwicklungspaket, das es ermöglicht, mit Visual C++ 6.0 (oder höher) Windows-Applikationen zur Fernsteuerung und -überwachung über Wireless LAN für den Aibo ERS-7 zu programmieren. Es bietet unter anderem die Funktion, einige der rohen Sensorinformationen des Aibo zu empfangen, bereits vorhandene oder

[1] Ecole Nationale Supérieure de Techniques Avancées

durch den Motion Editor (siehe Abschnitt 3.2) selbst erzeugte Bewegungsabläufe abzuspielen oder ein dreidimensionales Modell der aktuellen Roboterstellung am Bildschirm darzustellen. Außerdem kann man mit ihm auf viele von den bereits durch Sony implementierten Funktionen, wie zum Beispiel die Visual Pattern Recognition, die Stimmerkennung, die Hinderniserkennung, die Abgrunderkennung oder die Positionsbestimmung des rosafarbenen Balls zurückgreifen. Die Flexibilität des ARF ist allerdings durch die Voraussetzung, dass zur Anwendung von ARF-Programmen der originale, von Sony mitgeliefte AIBO MIND 2 Memorystick in den Aibo eingelegt sein muss, eingeschränkt. Das Aibo Remote Framework ist dadurch nämlich nicht mit OPEN-R SDK oder R-CODE SDK interoperabel.

In Unterpunkt 3.1.1 wird nun zunächst die Grundarchitektur des ARF beschrieben. Da dessen Dokumentation nicht besonders ausführlich ist und um einen Eindruck von der Programmierung mit dem ARF zu bekommen, wird außerdem exemplarisch in Abschnitt 3.1.2 ausführlicher erläutert, wie mit ihm die rohen Sensorinformationen des Aibo ausgelesen werden. Für eine detaillierte Beschreibung des Aibo Remote Framework wird auf die Diplomarbeit von Michael Kreutzer [Kre05] verwiesen.

### 3.1.1 Grundarchitektur

Das Aibo Remote Framework besteht im Wesentlichen aus zwei Hauptelementen (siehe Abbildung 3.1): dem Virtual Aibo Server und den Aibo Remote Framework API. Der Virtual Aibo Server ist ein Windows-Servertask, der das Zwischenglied zwischen dem Aibo und der Windowsanwendung bildet. Er ist für die Kommunikation über die W-LAN-Schnittstelle mit einem oder mehreren Aibos zuständig und bietet dem Programmierer eine sehr angenehme Schnittstelle für den Zugriff seiner Anwendungen auf Funktionen der Roboterplattform Aibo. Dieser Zugriff geschieht mittels des zweiten Hauptbestandteils des ARF, der Aibo Remote Framework API. Die Aibo Remote Framework API sind eine Sammlung von fünf DLL-Dateien (Dynamically Linked Libraries) und stellen den für den Entwickler eigentlich interessanten Teil des Aibo Remote Frameworks dar, da er mit ihnen die Programmierung seiner Überwachungs- oder Steueranwendungen vornimmt. Eine detaillierte Veranschaulichnug der Grundstruktur des ARF und der Interaktion der Bibliotheksdateien mit dem Virtual Aibo Server findet sich in Abbildung 3.2. Es folgt des Weiteren eine Auflistung und kurze Beschreibung dieser fünf DLL-Dateien:

**VAIBOClient.DLL** Diese ist die wichtigste Bibliothek des ARF. Mit ihrer Hilfe können beispielsweise Sensordaten oder Kamerabilder, die der Virtual Aibo Server im Shared Memory bereitgestellt hat, ausgelesen werden, die Modi der Aibos gewechselt oder Steuerbefehle an sie gesendet werden. Außer-

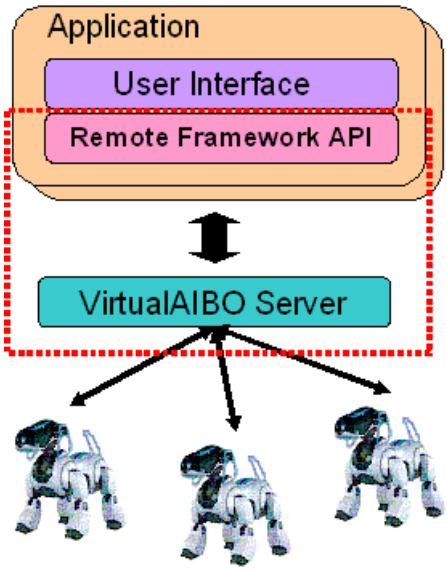



**Abbildung 3.1:** Schematische Darstellung der Grundbestandteile des ARF und ihrer Interaktion mit dem Aibo-Roboter und der Überwachungs- oder Steueranwendung. Das ARF ist in dieser Abbildung als Abgrenzung zu den anderen Elementen rot umrandet (Grafik entnommen von [Son04b]).

dem können über Windows Messages sogenannte *Semantics Information* vom Virtual Aibo Server empfangen werden. Hierbei handelt es sich um abstraktere Informationen als die rohen Sensordaten. Diese sind das Ergebnis der auf dem Aibo laufenden Auswertealgorithmen. Ein Beispiel ist die Semantic-Information *SEMID_ COLLISION*, die angibt, dass eine Kollision durch die Kollisionserkennung von Aibo entdeckt wurde.

**AIBO3D.DLL** Zur Darstellung eines dreidimensionalen Modell der aktuellen Stellung des Aibo über DirectX 7 kann AIBO3D.DLL verwendet werden.

**CPCInfo.DLL** Mit der CPCInfo.DLL lassen sich technische Daten des Aibo, wie zum Beispiel ein Array von Gelenk-IDs, auslesen. Diese Funktion ist insbesondere dann sinnvoll, wenn das ARF zukünftig einmal auf weitere Aibo-Modelle erweitert wird, um eine weitgehend vom Modell unabhänige Programmierung von ARF-Software erreichen zu können.

**VAIBOUPnP.DLL** Die VAIBOUPnP.DLL-Bibliothek gibt einem die Möglichkeit, ähnlich wie mit dem bei Aibo ERS-7 mitgelieferten Programm WLAN Manager 2 via Universal Plug and Play im Drahtlosnetzwerk nach Aibos suchen zu können.

**VAIBOTTS.DLL** Mit der VAIBOTTS.DLL hat man Zugriff auf die TTS(Text to Speech)-Engine, das Sprachsynthesemodul des Aibo.

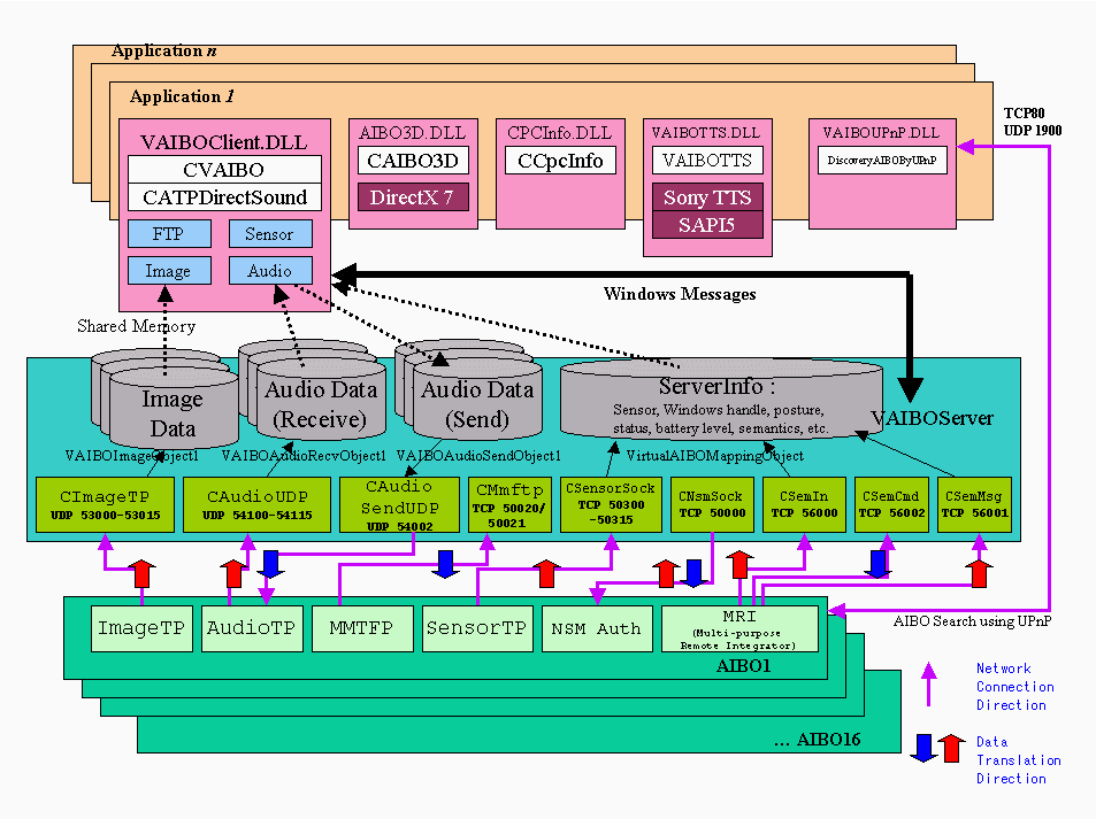



**Abbildung 3.2:** Detaillierte Darstellung der Struktur des Aibo Remote Frameworks. Wichtig ist hier insbesondere die Kommunikation der in orange abgebildeten Anwendungen mit dem Virtual Aibo Server (durch das große türkise Rechteck symbolisiert) über die in magenta dargestellten DLL-Dateien. Die Kommunikation geschieht entweder durch Windows Messages oder über Shared Memory (als graue Zylinderboxen dargestellt). Die Farbgebung der Abbildung korrespondiert mit der von Abbildung 3.1 (Grafik entnommen von [Son04b]).

### 3.1.2 Auslesen der Sensorinformationen

Die Sensordaten des Aibo werden durch den Virtual Aibo Server in regelmäßigen Abständen von der Roboterplattform empfangen und in einem geteilten Speicher-Bereich abgelegt (siehe Abbildung 3.2). Dieser wiederum kann mithilfe der Bibliothek VAIBOClient.DLL ausgelesen werden. Dies geschieht auf folgende Art und Weise: Wenn der Virtual Aibo Server die Sensordaten aktualisiert hat, sendet er eine *WM_VAIBO_SENSOR_DATA*-Nachricht an die Windowsapplikation. Diese Nachricht hat die beiden 32-Bit-Parameter wParam und lParam. Die ersten 16 Bit von wParam (HIWORD) geben in Integerkodierung die ID des jeweiligen Aibo an, dessen Sensorinformationen aktualisiert worden sind. Die darauffolgenden 16 Bit von wParam (LOWORD) enthalten – ebenfalls als Integer kodiert – die Anzahl der Elemente des Arrays der Sensorinformationen, das im Shared Memory abgelegt wurde. Der zweite Parameter lParam ist die

Windows-Speicheradresse des Arrays der Sensorinformationen (also ein Zeiger auf dieses Feld). Hierbei handelt es sich um ein Array vom Typ SensorRec, der in der Header-Datei VAIBODef.h des Aibo Remote Frameworks wie folgt definiert ist:

```
typedef struct SensorRec
{   int             sensorID;
    unsigned long   value;
    long            percentage_x;
    long            percentage_y;
} SensorRec, *SensorRecP;
```

Ein Element dieses Feldes enthält also die gesamte Information eines Sensors. Die Komponente *sensorID* charakterisiert dabei, um welchen Sensor es sich handelt und in der Teilvariablen *value* ist sein aktueller Wert abgespeichert. Die Attribute *percentage_x* und *percentage_y* sind nur für einige wenige Sensorwerte aktiv und sollen hier nicht erläutert werden. Eine Repräsentierung eines Sensors ist zwar sinnvoll, für den Menschen ist es allerdings sehr schwierig, eine Sensor-ID mit einem konkreten Sensor zu assoziieren. Daher wurde in der Datei CpcInfo.h die Enumeration *SensorID* definiert, die jeder möglichen Sensor-ID einen String-Identifier zuordnet:

```
// Primitive Locator ID
enum SensorID
{
        Acc1 = 0,
        Acc2, Acc3, HeadTilt, HeadPan, HeadRoll,
        Head1, Head2, Chin, PSD, LFJ1, LFJ2, LFJ3, LFSW,
        LRJ1, LRJ2, LRJ3, LRSW, RFJ1, RFJ2, RFJ3, RFSW,
        RRJ1, RRJ2, RRJ3, RRSW, BackSW,
        LEar, REar, Mouth, Tail1, Tail2, Temp,
        MultiSW1, MultiSW2, MultiSW3, PSDWithXY,
        HeadTilt2, BackSW2, BackSW3,
        PSDWithXY_FN, PSDWithXY_N, PSDWithXY_F,
        PrimitiveMax
};
```

Da es an ausführlicher Dokumentation zum Aibo Remote Framework momentan mangelt, musste durch Ausprobieren herausgefunden werden, welchem Sensor welcher Identifier zugeordnet ist. Die Ergebnisse dieser Untersuchung sind in Tabelle 3.1 aufgeführt; für eine genaue Beschreibung der Sensoren selbst sei auf das beim OPEN-R SDK mitgelieferte Datenblatt zum Aibo ERS-7 [Son04d] verwiesen. Außerdem bietet das im Rahmen dieser Studienarbeit entstandene Fernüberwachungstool eine gute Möglichkeit, neben der Programmierung mit dem ARF auch die vorhandene Sensorik und Aktorik des Aibo ERS-7 genauer kennenzulernen.

**Tabelle 3.1:** SensorIDs des ARFs und ihre Beschreibungen.

| **sensorID** | **Beschreibung** | **Dimension** |
|---|---|---|
| Acc1 | Beschleunigungssensor (x-Achse[a]) | $10^{-6}\,\frac{\mathrm{m}}{\mathrm{s}^2}$ |
| Acc2 | Beschleunigungssensor (y-Achse) | $10^{-6}\,\frac{\mathrm{m}}{\mathrm{s}^2}$ |
| Acc3 | Beschleunigungssensor (z-Achse) | $10^{-6}\,\frac{\mathrm{m}}{\mathrm{s}^2}$ |
| HeadTilt | Neigung des Kopfes | $10^{-6}$ (rad) |
| HeadTilt2 | Neigung des Hals | $10^{-6}$ (rad) |
| HeadPan | seitliche Drehung des Kopfes | $10^{-6}$ (rad) |
| HeadRoll | bei ERS-7 nicht existent | - |
| Head1 | Kopfberührungssensor | 1 |
| Head2 | momentan nicht aktiv | - |
| Chin | Kinnberührungssensor | 1 |
| PSD | momentan nicht aktiv | - |
| PSDWithXY(_N) | Kopfabstandssensor (nah) | $10^{-4}$ cm |
| PSDWithXY_F | Kopfabstandssensor (fern) | $10^{-4}$ cm |
| PSDWithXY_FN | Kopfabstandssensor (Kombination)[b] | $10^{-4}$ cm |
| LFJ1, LFJ2, ..., RRJ3 | Stellung der Gelenke[c] | $10^{-6}$ (rad) |
| LFSW, ..., RFSW | momentan nicht aktiv[d] | - |
| LEar, RAar | Stellung der Ohren | (n.u.)[e] |
| Mouth | Öffnungswinkel des Mundes | $10^{-6}$ (rad) |
| Tail1, Tail 2 | Stellung des Schwanz (2 Freiheitsgrade) | (n.u.) |
| Temp | momentan nicht aktiv | - |
| MultiSW1, ..., MultiSW3 | momentan nicht aktiv | - |
| BackSW, ..., BackSW3 | Tastsensoren am Rücken (von vorne nach hinten) | 1 |
| PrimitiveMax | nicht aktiv[f] | - |

[a] Aibos Referenzkoordinatensystem, siehe Kapitel 5

[b] Dieser Wert FN (far/near) ist eine Kombination aus dem Wert N (near) des Abstandsensors für sehr nahe Entfernungen und aus dem Wert F (far) des Abstandssensore für weitere Entfernungen. Sind N und W hoch, nimmt FN den Wert F an, sind beide Werte niedrig, so ist FN gleich N. Als Umschaltschwelle zwischen den beiden Werten F und N wurde eine Hysterese eingebaut: Sind F und N niedrig und steigen, so schaltet FN von N auf F, wenn F den Wert 22 cm überschreitet. Sind F und N hoch und sinken, so schaltet FN von F auf N, wenn N den Wert 20 cm unterschreitet.

[c] Der erste Buchstabe *L* oder *R* steht dabei für *left* respektive *right*, der zweite Buchstabe *F* oder *R* für *front* beziehungsweise *rear*. *J1* bis *J3* bezeichnen die Art des Gelenks. *J1* steht für das oberste Gelenk, das die Drehung der Beine verursacht, *J2* für das mittlere Gelenk, das zum Bewegen der Beine nach außen dient und *J3* für das unterste Gelenk, das „Kniegelenk“.

[d] Möglicherweise für die Zukunft als Werte der Tastsensoren der Pfoten angedacht.

[e] nicht untersucht

[f] Es ist üblicher Trick in C/C++, ein weiteres ungenutztes Element als letzten Bestandteil einer Enumeration zu definieren. Dieses Element kann dann zur Dimensionierung von Arrays verwendet werden.

## 3.2 Aibo Motion Editor

Der Aibo Motion Editor (siehe Abbildung 3.3) ist keine Programmierumgebung, sondern ein Windowsprogramm zum Erzeugen von Bewegungsabläufen für das Aibo-Modell ERS-7. Durch Positionieren des dreidimensionalen Modells mit der Maus oder durch Einstellen von Parameterwerten lassen sich sogenannte Posen erzeugen. Diese Posen können dann in einer für den Bewegungsablauf gewünschten Reihenfolge angeordnet werden. Für jeden Übergang von einer Pose in die nächste muss außerdem eine Interpolationszeit als Maß für die Übergangsgeschwindigkeit eingetragen werden. Auf diese Art und Weise lassen sich mit dem Aibo Motion Editor Bewegungsmuster erstellen, die in Verbindung mit OPEN-R SDK, R-CODE SDK und dem AIBO Remote Framework eingesetzt werden können. Eine Beschreibung zum Aibo Motion Editor findet sich in [Son04a].

## 3.3 OPEN-R SDK

*OPEN-R* steht für *Open* architecture for *R*obot Entertainment und ist ein allgemeines Architekturkonzept der Firma Sony für eine modulare Hard- und Softwarestruktur von Roboterunterhaltungssystemen. Bei dem *OPEN-R Software Development Kit* (OPEN-R SDK) hingegen handelt es sich um eine C++-Entwicklungsumgebung, die es ermöglicht, eigene auf dem Roboter laufende Steuersoftware für einen OPEN-R-Standard-konformen Unterhaltungsroboter – in diesem Fall den Aibo – zu entwickeln. Es enthält ein Basissystem[2] in kompilierter Form, welches gemeinsam mit der selbst entwickelten Software durch ein Skript auf einen leeren Memorystick kopiert wird. Außerdem ist das OPEN-R SDK mit einem C++-Cross-Compiler[3] sowie Bibliotheken für den Zugriff auf Funktionen dieses Basissystems ausgestattet. Da sie sehr hilfreich für die Programmierung des Aibo sind, werden zunächst in Unterpunkt 3.3.1 die Grundideen von Sonys OPEN-R vorgestellt. Anschließend wird in Abschnitt 3.3.2 ein Überblick gegeben, wie diese Konzepte konkret auf dem Aibo realisiert wurden und welche Besonderheiten das Betriebssystem Aperios und die Entwicklungsumgebung OPEN-R SDK charakterisieren. Zur Erlangung weitergehender Kenntnisse über die Konzepte von OPEN-R und die Programmierung mit dem OPEN-R SDK sei hier zu [Hol04, FK97, RGCCM04, SB03, Tél04a, Son04e, Son04c] geraten.

[2]Dieses besteht aus dem Aperios-Betriebssystemkern und notwendigen Systemdiensten.

[3]Als *Cross-Compiler* wird ein Compiler bezeichnet, der Maschinencode für einen anderen Prozessor erzeugt als den, von dem er ausgeführt wird. Den Cross-Compiler beim OPEN-R SDK bildet der C++-Compiler der GNU Compiler Collection (GCC) in Verbindung mit einem speziellen Backend für den MIPS-Prozessor des Aibos.

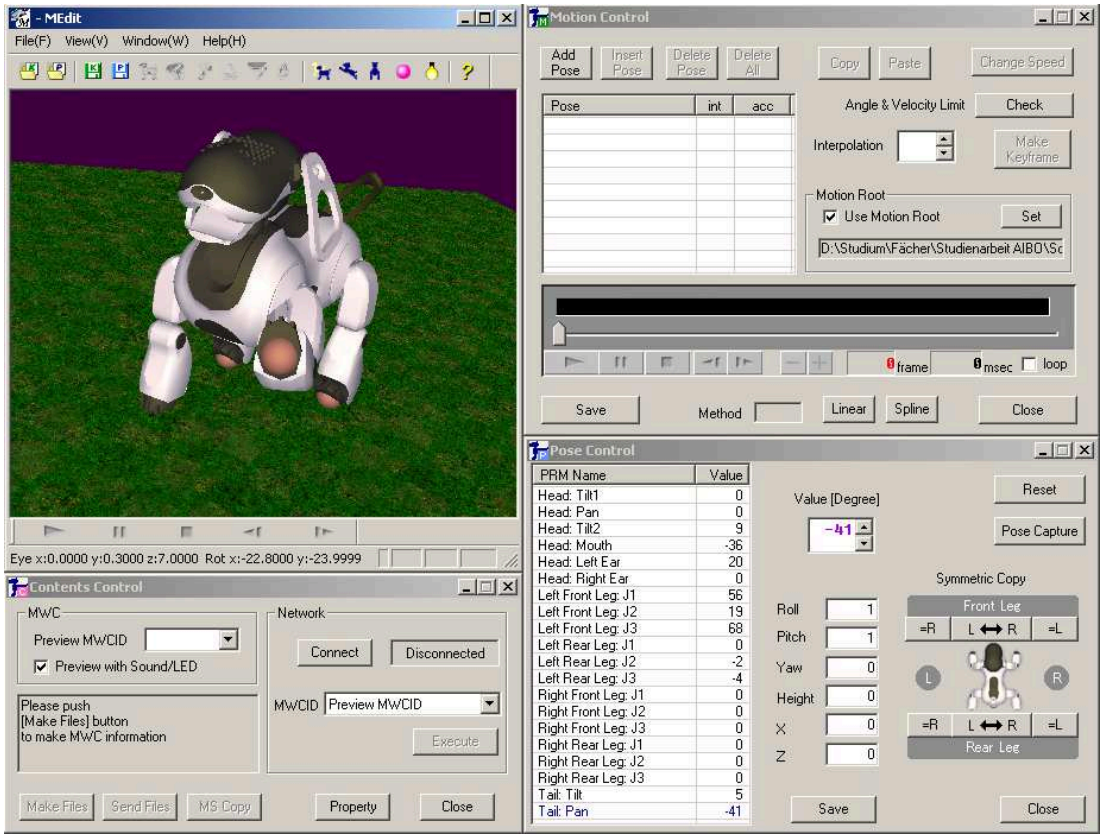



**Abbildung 3.3:** Mit dem Aibo Motion Editor lassen sich sehr leicht Bewegungsabläufe für den Aibo erzeugen.

### 3.3.1 Konzepte von OPEN-R

Zu Beginn dieses Unterpunktes soll das in der *Open architecture for Robot Entertainment* [FK97] niedergelegte *Allgemeine Referenzmodell* (Generic Reference Model) kurz vorgestellt werden. Es dient als Grundorientierung, um zu verstehen, was im OPEN-R-Standard unter einem Roboterunterhaltungssystem im Ganzen zu verstehen ist. Das Allgemeine Refernzmodell untergliedert ein Roboterunterhaltungssystem in die folgenden drei Komponenten:

1. **Basissystem (Basic System):** Das Basissystem bezeichnet die Roboterplattform selbst. Es besteht aus einem Systemkern – ein CPU-Board mit Hauptspeicher –, den Sensoren und Aktuatoren – in dem Modell durch sogenannte *Configurable Physical Components* (siehe unten) charakterisiert – und einem MPS (Media for Program Storage), einem herausnehmbaren, frei programmierbaren Speichermedium wie zum Beispiel einem Memorystick.

2. **Entwicklungsumgebung (Development Environment):** Die Entwicklungsumgebung ist ein auf einem PC oder einer Workstation installiertes

Softwarepaket. Sie soll dem Anwender eine komfortable Umgebung, gegebenenfalls auch mit graphischer Oberfläche, zur Programmierung von Applikationen für das Basissystem zur Verfügung stellen.

3. **Erweiterungssystem:** Hierbei handelt es sich um einen PC, ein Rechnernetz oder ein Multiprozessorsystem. Das Erweiterungssystem ist, beispielsweise über eine Drahtlosnetzwerk, mit dem Basissystem verbunden. Es kann zur Überwachung der Roboterplattform dienen, aber auch zur Auslagerung rechenintensiver Prozesse bis hin zur Fernsteuerung des Basissystems. Das Entwicklungssystem und das Erweiterungssystem können ein und dieselbe Plattform sein.

Des Weiteren werden nun die signifikanten Konzepte von OPEN-R für die Architektur eines Unterhalungsroboters erläutert. Ein zentrales Ziel der offenen Roboterarchitektur ist ein möglichst hohes Maß an Modularisierung der Hard- und Software des Roboters. Um dies zu ermöglichen, werden klar definierte, standardisierte Schnittstellen benötigt. Die Menge dieser geforderten Schnittstellen enthält unter anderem die zwischen Hard- und Software und die zwischen System- und Anwendersoftware. Aus diesem Grund schlägt OPEN-R die in Abbildung 3.4 dargestellte, ebenenstrukturierte Architektur (Layered Architecture) vor. Sie

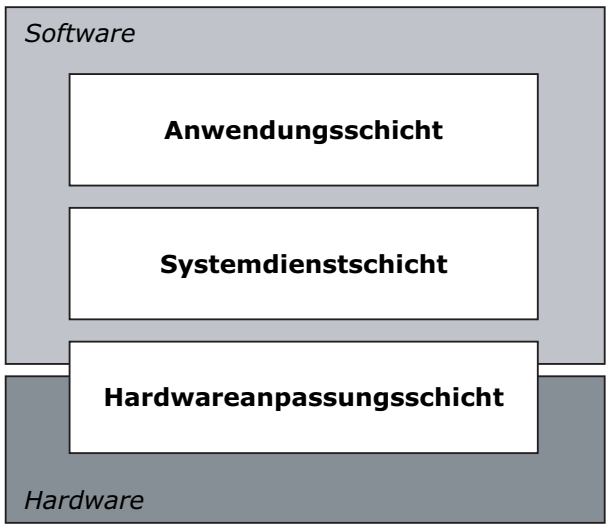



**Abbildung 3.4:** Ebenenstruktur der Open Architecture for Robot Entertainment.

besteht aus drei Schichten:

1. **Hardwareanpassungsschicht (Hardware Adaption Layer):** Die Hardwareanpassungsschicht abstrahiert die Details der Hardware und stellt somit der Systemdienstschicht ein universelles Interface zum Zugriff auf sie zur

Verfügung. Dies geschieht durch eine sinnvolle Strukturierung des Hardwareaufbaus, gegebenenfalls aber auch durch eine zusätzliche, möglichst klein gehaltene Softwareebene.

2. **Systemdienstschicht (System Service Layer):** In der Systemdienstschicht sind, wie der Name schon vermuten lässt, die Systemdienste des Betriebssystems der Roboterplattform angesiedelt. Sie ist das Bindeglied zwischen der Steuersoftware und der Hardware des Roboters und offeriert dem Anwender eine standardisierte Zugriffsmöglichkeit auf die Systemdienste und physikalischen Komponenten des Roboters.

3. **Anwendungsschicht (Application Layer):** Auf dieser Ebene kann der Anwender seine Applikationen für das Robotersystem entwickeln. Aufgrund des hohen Abstraktionsniveaus – erreicht durch die Schichtenstruktur – soll es leicht möglich sein, vorhandene Steueralgorithmen auf andere Robotorplattform zu portieren.

Es kommt ferner noch die Bestrebung nach klaren Schnittstellen zwischen den Hardware- und den Softwarekomponenten im Allgemeinen hinzu. Es besteht also die Forderung nach klar definierten Schnittstellen

- zwischen Hard- und Software,
- zwischen den Hardwarekomponenten untereinander und
- zwischen den Softwarekomponenten untereinander, im Besonderen zwischen Anwender- und Systemsoftware.

Um dies zu gewährleisten, werden von dem OPEN-R-Standard folgende Konzepte vorgeschlagen:

**Configurable Components** (für eine klar definierte Schnittstelle zwischen den Hardwarekomponenten untereinander und zwischen Hard- und Software). Um einen möglichst allgemeingültigen Standard der Hardwarestruktur einer Roboterplattform definieren zu können, wird als Modell für dessen physikalische Elemente, die von der Software aus zugfreifbar sein sollen (in der Regel sind dies die Sensoren und Aktoren), das Konzept der *Configurable Physical Component* (CPC) eingeführt.

Dieses sieht vor, dass sämtliche physikalische Elemente des Robotorsystems, eben diese CPCs, über eine Baumstruktur mit einem seriellen Bus verbunden sind (als Beispiel siehe Abbildung 3.5). Des Weiteren sollen die CPCs die Fähigkeit besitzen, sich selbstständig über eine speziell definierte Plug-and-Play-Schnittstelle gegenüber dem System zu identifizieren und ihm ihre spezifischen Eigenschaften mitzuteilen. Dadurch soll ermöglicht werden,

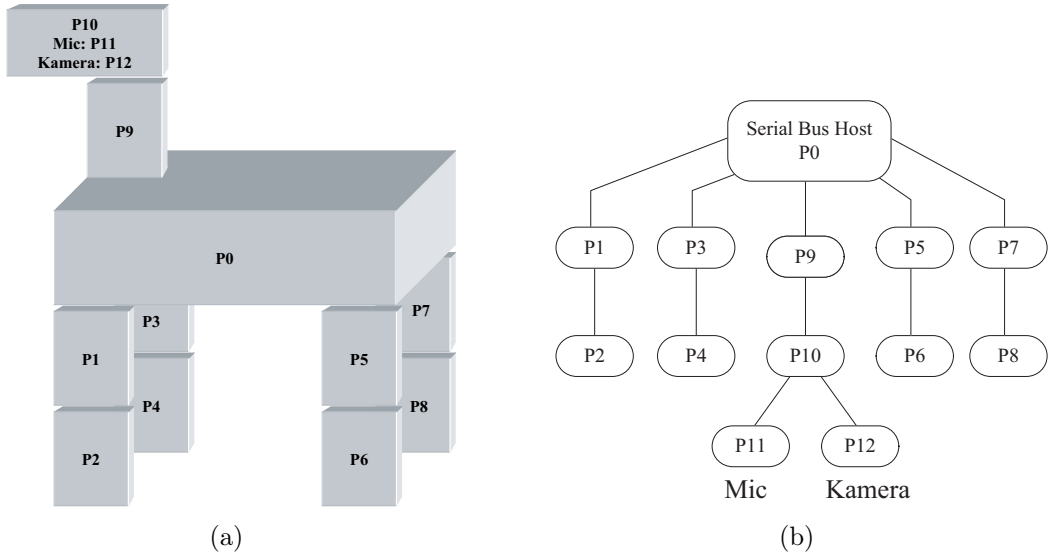



**Abbildung 3.5:** Der schematische physikalische Aufbau der CPCs eines einfachen, gedachten Roboterhundes (a) und dessen abstrakte Repräsentierung als CPC-Baumstruktur (b) (Darstellung angelehnt an Grafiken aus [FK97]).

dass die Software die Hardware des Roboters automatisch erkennen und konfigurieren kann, ohne dass eine für eine bestimmte Hardwarekonfiguration spezifische Hardwareanpassungsschicht auf Softwareebene nötig ist. Außerdem sind die physikalischen Elemente des Roboters dadurch beliebig austauschbar, solange sie dem CPC-Standard genügen.

Will man nun von der Software aus auf eine bestimmte CPC zugreifen, braucht man nur den CPC-Verbindungsaufbau des Robotors zu kennen und kann dann über die Baumstruktur die gewünschte Komponente adressieren. Die Adresse der Kamera des in Abbildung 3.5 dargestellten Roboters ist beispielsweise */P0/P9/P10/P12*.

**OPEN-R-Objekte** (für eine klar definierte Schnittstelle zwischen Anwendungs- und Systemdienstschicht und zwischen den Softwarekomponenten im Allgemeinen). Zur Modularisierung der Software präsentiert die offene Architektur für Unterhaltungsrobotersysteme das Konstrukt des *OPEN-R-Objektes*. Dieses ist eher an das Konzept des Prozesses eines Multitaskingbetriebssystems als an das des Objektes der objektorientierten Programmierung angelehnt, denn OPEN-R-Objekte sind Softwareeinheiten, die parallel und unabhängig voneinander auf einem Roboter ausgeführt werden. Vollständig unabhängig sind sie jedoch nicht, da sie über ein standardisiertes Interface miteinander kommunizieren können. In Abbildung 3.6 ist eine symbolische Veranschaulichung der Interobjektkommunikation der konkurrierend ablaufenden OPEN-R-Objekte zu sehen.

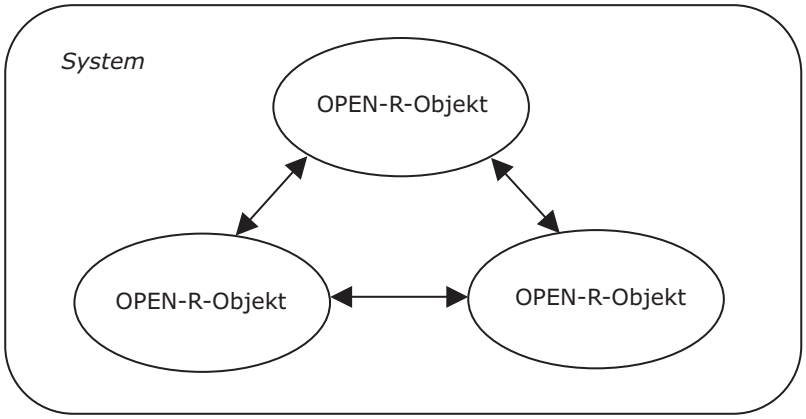



**Abbildung 3.6:** Interobjektkommunikation konkurrierend ablaufender OPEN-R-Objekte (Darstellung angelehnt an Grafik aus [Son04e]).

Es ist vorgesehen, dass alle Softwareeinheiten des Robotersystems auf sinnvolle Art und Weise in solche OPEN-R-Objekte gekapselt werden.

Die eben skizzierte OPEN-R-Architektur soll eine Struktur für Unterhaltungsrobotersysteme darstellen, die einen leichten und modularen Austausch von Hard- und Softwareentitäten erlaubt. Die Tatsache, dass die Aibo-Modelle ERS-7, ERS-210, ERS-220, ERS-210A und ERS-220A mit ein und derselben OPEN-R-Entwicklungsumgebung programmierbar sind und dass laut Ankündigiung auch Sonys neuer, humanoider Unterhaltungsroboter *QRIO* auf dem OPEN-R-Standard basiert, lässt darauf hindeuten, dass Sony dies – zumindest in Ansätzen – gelungen ist.

### 3.3.2 Überblick über Aperios und das OPEN-R SDK

**Prozessmodell:** Auch in der konkreten Implementierung des OPEN-R-Modells auf dem Aibo werden die Prozesse des Aibo-Betriebssystems Aperios als *Objekte* bezeichnet. Bei der Programmierung wird ein OPEN-R-Objekt durch seine sogenannte *Kernklasse* (Core Class), die von der Basisklasse *OObject* abgeleitet sein muss, charakterisiert.

Bei Aperios und dem OPEN-R-SDK handelt es sich um ein **ereignisorientiertes** System. Nach einer Initialisierungsphase befinden sich alle Objekte erst einmal im Ruhezustand (Idle State). Für ein OPEN-R-Objekt existiert also kein Einstiegspunkt wie eine *Main()*-Methode, die dann sequentiell abgearbeitet wird. Vielmehr hat ein OPEN-R-Objekt für jedes Ereignis, das für das Objekt interessant ist, einen eigenen *Einstiegspunkt* (Entry Point). Jeder Einstiegspunkt korrespondiert mit einer bestimmten Member-Funktion der Kernklasse des Objekts. Diese Member-Funktion enthält eine spezifische Behandlungsroutine für das jeweilige Ereignis. Ein Beispiel für

ein Ereignis ist das Erhalten einer bestimmten Nachricht von einem anderen OPEN-R-Objekt.

Ein OPEN-R-Objekt wird nach seiner Kompilierung in Form einer einzigen Datei mit der Endung **.bin* auf dem Memorystick des Aibo abgelegt. Die Objekte der Anwendungsschicht befinden sich alle im Verzeichnis `ms/open-r/mw/objs`. Welche Objekte beim Boot des Aibo initialisiert werden, ist in der Konfigurationsdatei *OBJECT.CFG* festgelegt.

**Interobjektkommunikation:** Zur Kommunikation mit anderen Objekten, benötigt ein OPEN-R-Objekt sogenannte *Subjects* und *Observers*. Ein *Subject* ist eine Art Sendeeinheit eines Objektes, die es ermöglicht, Daten an den Observer eines anderen Objektes zu übermitteln. Entsprechend ist ein *Observer* eine Empfangseinheit eines Objektes, die Daten von dem Subject eines anderen Objektes entgegennehmen kann. Es kann ein Subject nur mit genau einem Observer kommunizieren; ein OPEN-R-Objekt kann allerdings beliebig viele Subjects und Observer besitzen. Wie viele Subjects und Observer ein Objekt besitzt und welches Subject mit welchem Observer kommuniziert, wird in den Konfigurationsdateien *CONNECT.CFG* und *stub.cfg* beschrieben.

Die Interobjektkommunikation in Aperios geht wie folgt vonstatten. In der Initialisierungsphase eines Objektes, senden dessen Observer gewöhnlich an alle ihre Subjects eine *Ready*-Nachricht. Eine Ready-Nachricht teilt einem Subject mit, dass sein Observer empfangsbereit ist. Nun kann das entsprechende Subject Daten in einem **Shared-Memory-Bereich** für seinen Observer hinterlegen. Ist es damit fertig, sendet es eine *Notify*-Nachricht an seinen Observer. Dadurch weiß der Observer, dass er nun die für ihn bestimmten Daten aus dem Shared-Memory-Bereich auslesen kann. Hat er dies getan, sendet der Observer wiederrum eine Ready-Nachricht an sein Subject, das nun erneut Daten für den Observer bereitstellen kann.

**Scheduling:** Das Scheduling in Aperios ist preemtiv und erfolgt auf der Basis **statischer Prioritäten**. Zugewiesen werden diese Prioritäten manuell durch den Programmierer des jeweiligen Objektes. Sie umfassen 8 Bit. Die oberen 4 Bit davon ordnen das Objekt einer Schedulingklasse zu. Ein Objekt einer bestimmten Klasse kann niemals aktiv sein, wenn noch ein anderes Objekt einer höheren Schedulingklasse ausgeführt wird. Die unteren 4 Bit bestimmen, wie viel Rechenzeit in etwa dem Objekt relativ zu den anderen Objekten der gleichen Schedulingklasse zugeteilt wird.

Um **Wettlaufsituationen** (Race Conditions) eines Objektes mit sich selbst unmöglich zu machen, wird ein OPEN-R-Objekt immer nur durch einen einzigen Thread betrieben. Dies bedeutet, dass nur maximal eine Ereignisbehandlungsfunktion eines bestimmten Objektes zur gleichen Zeit aktiv

ist, Memberfunktionen verschiedener Objekte aber durchaus parallel[4] ausgeführt werden können. Erhält ein Objekt während der Behandlung eines Ereignisses ein weiteres Ereignis, so wird dieses gepuffert und zu einem späteren Zeitpunkt auf dem Objekt ausgelöst.

**Speicherverwaltung:** In Aperios spricht ein Objekt standardmäßig den Hauptspeicher des Aibo über einen **virtuellen Adressraum** an. Der virtuelle Speicher lässt sich lokal für einzelne OPEN-R-Objekte, aber auch global für das gesamte System deaktivieren. Wird global eine physikalische Adressierung des Speichers aktiviert, laufen sämtliche Objekte im Kernelmodus.

Außerdem sei noch darauf hingewiesen, dass man mit dem OPEN-R SDK die Verwendung **Prozessor-Caches** für ein bestimmtes OPEN-R-Objekt unterbinden kann. Dies erlaubt eine bessere Vorhersagbarkeit der Ausführungszeiten des Objekt-Codes.

**Zeitbasis:** Zeit spielt für Echtzeitsysteme eine entscheidende Rolle. Selbstverständlich kann für ein Digitalsystem wie den Aibo nur eine diskrete Repräsentierung der Zeit gewählt werden. Die Basiseinheit der Zeit des Aibo wird *Frame* genannt. Sie entspricht einer Spanne von acht Millisekunden.

**Systemdienstschicht:** Die Systemdienstschicht des Aibo-Roboters ist derzeit durch eine Menge von Objekten realisiert, die sich im Verzeichnis `ms/open-r/system/objs` des Memorysticks befinden. Leider ist durch Sony nur das Interface zu dieser Systemdienstschicht veröffentlicht; die Systemdienstschicht selbst, wie auch die Hardwaredetails des Aibo sind geheim.

## 3.4 R-CODE SDK

R-CODE ist eine interpretierte Skriptsprache zur autonomen Programmierung des Aibo ERS-7. Um R-CODE verwenden zu können, muss man das bei dem R-CODE-Paket mitgelieferte, vorkompilierte OPEN-R-Projekt auf seinen leeren Memorystick kopieren. Ein selbstgeschriebenes R-CODE-Programm muss dann nur noch unübersetzt in ein bestimmtes Verzeichnis (`/OPEN-R/APP/PC/AMS/`) mit einem bestimmten Namen (R-CODE.R) auf diesen Stick übertragen. Dieses Programm wird nach Einschalten des Aibo von einem auf dem fertigen Memorystick implementierten OPEN-R-Objekt zur Laufzeit interpretiert und ausgeführt.

R-CODE ist für den Anwenderbereich ausgelegt und somit sehr intuitiv und ohne Programmiervorkenntnisse verständlich. Man kann mit R-CODE beispielsweise mit wenigen Befehlen Aibo in eine seiner Basispositionen (Sitzen, Stehen, Schlafen) bringen, ihn laufen oder sich drehen lassen, vorgefertigte Methoden zur

[4] Da es sich bei Aibo um ein Einprozessorsystem handelt, müsste man hier eigentlich genauer von *quasiparallel* sprechen.

Gesichts- oder Stimmerkennung verwenden und eine der 600 mitgelieferten oder mit dem Aibo Motion Editor (siehe Abschnitt 3.2) selbst erstellte Bewegungsmuster ausführen. Der Preis, den man für diese Einfachheit zahlt, ist natürlich eine enorm geringe Fülle an Möglichkeiten wie auch eine sehr langsame Ausführung der selbstgeschriebenen Programme.

Für weitere Hinweise zu R-CODE sei auf [Tél04b] verwiesen. Um außerdem noch einen Eindruck von der Programmierung mit R-CODE zu bekommen wird hier abschließend ein kurzes Beispielprogramm angegeben, das Aibo dazu veranlasst, sich zehn mal hinzusetzen und wieder aufzustehen:

```
:DoSitStand
  FOR:i:1:10
    PLAY:ACTION:SIT
    WAIT
    PLAY:ACTION:STAND
    WAIT
  NEXT
RETURN
```

## 3.5 Tekkotsu-Framework

Da das OPEN-R SDK nur eine Basisschnittstelle zum Betriebssystem und der Sensorik und Aktorik des Aibos darstellt, es aber keine eigene Funktionalität mit einschließt, erfordert es tiefergehende Kenntnisse im Bereich der Robotik und einen erheblichen Entwicklungsaufwand, eigene Steuersoftware für Sonys Aibo mit ihm zu programmieren. Entwickelt von den RoboCup-Teams der *Sony Four-Legged Robot League* existiert inzwischen zwar eine sehr große Menge an frei verfügbarem Quellcode für den Aibo-Roboter, eine Wiederverwendung dieses Codes bedarf jedoch einer umfangreichen Portierung, da dieser Quelltext auf das sehr spezielle Feld des Roboterfußballs ausgelegt ist. Aus diesem Grund wurde an der *Carnegie Mellon University* in Pittsburgh ein objektorientiertes, ereignisbasiertes Open-Source-Framework namens *Tekkotsu*[5] entwickelt, das dem Programmierer neben einer sehr allgemein konstruierten vorgefertigten Software-Grundstruktur auch bereits implementierte Funktionalität wie zum Beispiel einen Laufalgorithmus, Methoden zur Berechnung der direkten und inversen Kinematik oder Algorithmen zur Bildverarbeitung bietet. Es wird in diesem Kapitel nun eine kurze Übersicht über den Aufbau und die Konzepte dieses Tekkotsu-Frameworks präsentiert.

Das Tekkotsu-Framework wurde mit dem OPEN-R SDK entwickelt und besteht, wie auch in Abbildung 3.8 verdeutlicht, aus vier separaten Objekten. Möch-

[5] *Tekkotsu* ist japanisch und bedeutet wörtlich übersetzt *eiserne Knochen.*

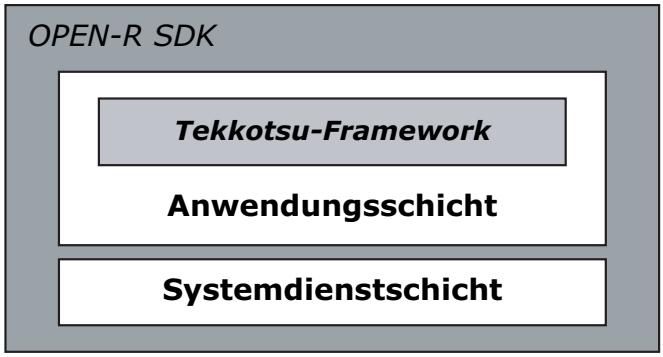



**Abbildung 3.7:** Das Tekkotsu-Framework ist ein OPEN-R-SDK-Projekt, das seinem Anwender eine allgemein gehaltene Softwaregrundstruktur, bereits vorimplementierte Algorithmen sowie eine bequeme Schnittstelle zum Zugriff auf die Funktion des Roboters zur Verfügung stellt.

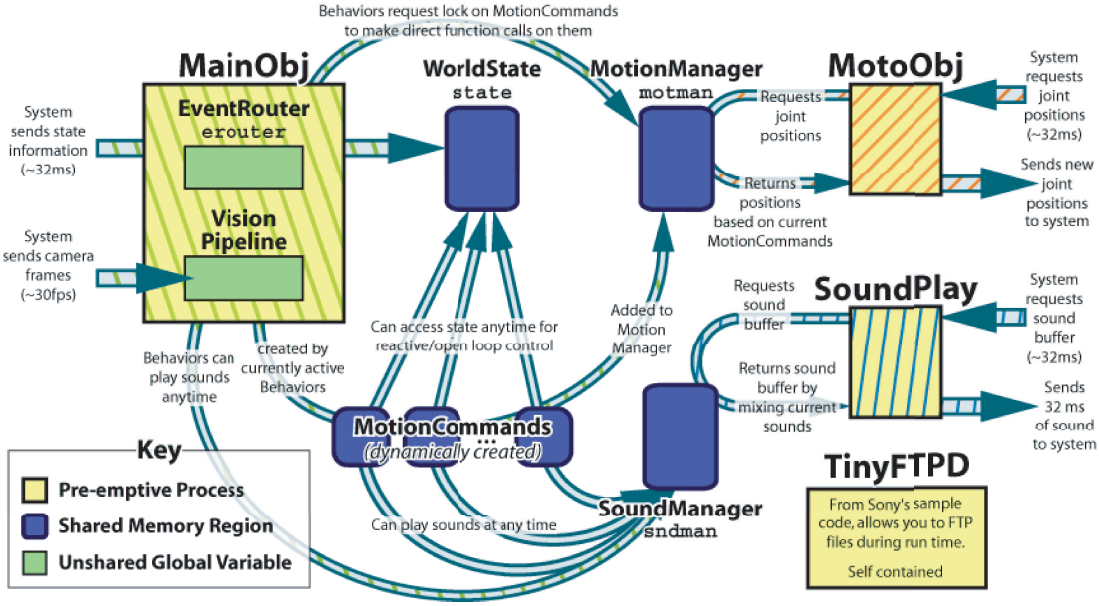



**Abbildung 3.8:** Architektur des Tekkotsu-Frameworks (Diagramm entnommen von [TTT05]).

te man Programme für einen Aibo mit ihm erstellen, ist es von Tekkotsu vorgesehen, dass man nicht seine eigenen OPEN-R-Objekte, sondern sogenannte *Behaviors* schreibt, deren Code vom Hauptprozess des Frameworks ausgeführt wird und deren Summe das Verhalten des Roboters bestimmt. Das Tekkotsu-Framework ist also aus technischer Sicht betrachtet ein OPEN-R-SDK-Projekt der Anwendungsschicht[6], welches sein Anwender auf das von ihm gewünsch-

[6]Die Systemdienstschicht des OPEN-R SDK ist wie auch Aibos Hardware proprietär und kann nur von Sony geändert werden.

te Verhalten des Roboters anpasst. Dieser Sachverhalt wurde in Abbildung 3.7 symbolisch veranschaulicht.

Ein *Behavior* ist dabei eine C++-Klasse, die von der in Tekkotsu implementierten abstrakten Basisklasse *BehaviorBase* abgeleitet sein muss. In welcher Art und Weise dann die Methoden der Behaviors aufgerufen werden, wird über eine im Framework definierte Menge von Ereignissen[7] bestimmt. Als Behandlungsroutine der Ereignisse ist hierzu für jedes Behavior zwingend eine Member-Funktion namens *processEvent(EvenBase & event)* vorgeschrieben. Diese enthält als Argument eine Referenz auf das zu behandelnde Ereignis. Wird am sogenannten *EventRouter* des Frameworks ein Ereignis ausgelöst, führt er diese Methode aller der Behaviors aus, die momentan für dieses Ereignis registriert sind. Ein Behavior kann sich zu jedem Zeitpunkt beim EventRouter für ein bestimmtes Ereignis registrieren und wieder abmelden lassen. Zur Illustration des ereignisgesteuerten Charakters der Behaviors sei auf ihren in Abbildung 3.9 dargestellten Lebenszyklus verwiesen.

Um eben erläutertes zu verdeutlichen und den Überblick über die Architektur des Frameworks zu vervollständigen folgt eine Beschreibung der in Abbildung 3.8 dargestellten vier Objekte des Tekkotsu-Frameworks:

**MainObj** MainObj ist der Hauptprozess des Frameworks. Alle Behaviors sind in seinem Adressraum instanziiert und werden von ihm ausgeführt. Außerdem enthält er eine globale Variable vom Typ *EventRouter* auf die der Pointer *erouter* zeigt. Wie bereits erwähnt ist der EventRouter für die Verwaltung der Ereignisse zuständig. Über seine Member-Funktion *postEvent(EventBase * e)* können von MainObj, beispielsweise wenn neue Sensordaten verfügbar sind, aber auch von Behaviors, Ereignisse ausgelöst werden. Wenn dies geschieht, sorgt der EventRouter dafür, dass die *processEvent(...)*-Methoden der für dieses Ereignis registrierten Behaviors der Reihe nach ausgeführt werden. Die sequentielle Ausführung der Ereignisbehandlungsmethoden macht Wettlaufsituationen auf Behaviorebene unmöglich.

Im Hauptprozess ist des Weiteren die *Vision Pipeline* angesiedelt. Diese empfängt von Aibos System das aktuelle Kamerabild und bearbeitet es in mehreren Schritten weiter. Behaviors können zwischen jeder Bearbeitungsstufe der Vision Pipeline auf das jeweilige Stadium des Bilds zugreifen.

Eine weitere Aufgabe des Objekts MainObj ist es, in regelmäßigen Abständen (circa 30 ms) von Aibos Systemobjekt *OVirtualRobotComm* neue Sensorwerte zu erhalten und in der in einem geteilten Speicherbereich angelegten Instanz der Klasse *WorldState* abzulegen. Aus dieser Variable können alle Behaviors über den Zeiger *state* die aktuellen Sensorinformationen ablesen.

[7] Es sei hier betont, dass diese Tekkotsu-internen Ereignisse nicht mit denen des Aperios-Betriebssystems zu verwechseln sind.

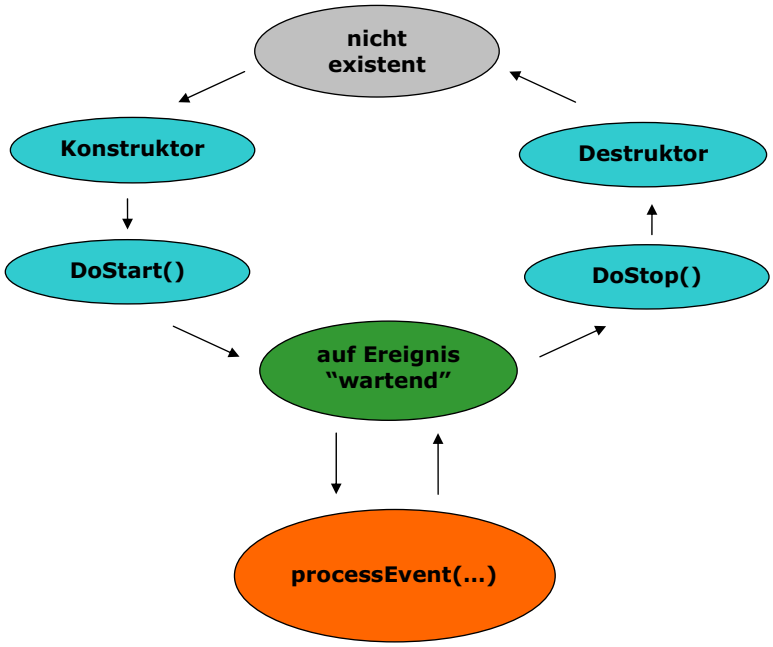



**Abbildung 3.9:** Lebenszyklus eines Behaviors als Zustandsdiagramm. In einem Behavior sind zwingend die Initialisierungsmethode *DoStart()*, die Deinitialisierungsmethode *DoStop()* und die Ereignisbehandlungsroutine *processEvent(...)* deklariert. Eine typische Verwendung für die DoStart()- und DoStop()-Funktionen ist beispielsweise das An- respektive Abmelden für verschiedene Ereignisse.

**MotoObj** Die Aufgabe des Objekts MotoObj ist das Ausführen der sogenannten *MotionCommands*, die zur Ansteuerung von Aibos Aktorik verwendet werden. MotionCommands erben von der gleichnamigen Klasse *MotionCommand* und werden in der Regel von Behaviors instanziiert, über ihrer Member-Funktionen wir gewünscht konfiguriert und über den Zeiger *motman* dem ebenfalls im Shared-Memory-Bereich liegenden *MotionManager* hinzugefügt.

MotoObj ist als Subject des Systemprozesses *OVirtualRobotComm* registriert, der es zyklisch (circa alle 30 ms) dazu auffordert, neue Sollwerte für die Gelenkstellungen an ihn zu senden. Wenn dies passiert, führt MotoObj die Member-Funktion *getOutputs(...)* des MotionManagers aus, die wiederrum der Reihe nach die Methode *updateOutputs()* aller sich momentan im MotionManger befindenden MotionCommands aufruft. In dieser Funktion *updateOutputs()* berechnet ein MotionCommand – in den meisten Fällen in Abhängigkeit von den aktuellen Sensorwerten – die von ihm gewünschten neuen Sollwerte der Aktuatoren und legt sie im MotionManager ab. MotoObj erhält schlussendlich von *getOutputs(...)* die Überlagerung der neu berechneten Stellwerte vom MotionManager und sendet sie an Aibos System. Bei der Programmierung sowie insbesondere der Verwendung von Motion-

Commands ist gründlich darauf zu achten, dass es zu keiner unerwünschten Interferenz von ihnen kommt.

Der Code von MotoObj wurde in diesem gesonderten Prozess untergebracht, um eine flüssige Ausgabe der Roboterbewegungen auch dann zu gewährleisten, wenn in den Behaviors gerade rechentechnisch aufwendige Algorithmen vollzogen werden. Es kommt allerdings häufig vor, dass Behaviors auch Methoden von MotionCommands dann ausführen möchte, nachdem sie sie bereits dem MotionManager hinzugefügt haben. Um möglicherweise fatale Folgen des gleichzeitigen Zugriffs von MainObj und MotoObj auf MotionCommands zu vermeiden, wurden dazu im Tekkotsu-Framework Sperren implementiert, die MotionCommands für den Zeitraum des Zugriffs der Behaviors dem MotionManager und somit MotoObj unzugänglich machen.

**SoundPlay** Der Prozess SoundPlay sorgt analog zum MotoObj parallel zu anderen Berechnungen für eine saubere Ausgabe der beim *SoundManager* in Auftrag gegebenen Klänge.

**TinyFTPD** Das Objekt TinyFTPD aus der Beispielsammlung von Sony implementiert einen rudimentären FTP-Server für Aibos Betriebssystem. Mit ihm lassen sich Dateien über die W-LAN-Schnittstelle auf den Memorystick übertragen.

Detaillierte Informationen über das Tekkotsu-Framework finden sich neben der Website auch in [Mül04].

## 3.6 Überblick über weitere Projekte

In diesem Abschnitt werden kurz einige weitere Programmierumgebungen für Aibo vorstellt, die hier nicht im Detail erläutert werden sollen.

### 3.6.1 URBI und Pyro

Bei der Entwicklung von Steueralgorithmen für Roboter kann es sehr nützlich sein, diese ohne umständliche Portierung mit verschiedenen Plattformen testen zu können. Um diesem Ziel näher zu kommen, wurden die zwei Projekte *URBI* und *Pyro* ins Leben gerufen, die in diesem Unterpunkt kurz erwähnt werden sollen.

*Py*thon *Ro*botics [BKMY05] ist ein Open-Source-Packet, das eine konfigurierbare, von der Hardware abstrahierende Schnittstelle zur Fernsteuerung einer großen Klasse von mobilen Robotern über die objektorientierte, interpretierte

Programmiersprache Python zur Verfügung stellt. Momentan bietet Pyro vorimplementierte Unterstützung zur Ansteuerung der Roboter *Pioneer*, *Pioneer2*, *PeopleBot*, *Khepera*, *Khepera 2*, *Hemisson* sowie des Sony Aibo. Der Aibo wurde dabei mithilfe des Tekkotsu-Frameworks auf Pyro angepasst. Außerdem ist Pyro bereits zur Zusammenarbeit mit den Roboter-Simulatoren *Player/Stage*, *Robocup Soccer Simulator*, *Gazebo* und dem *Khepera Simulator* konfiguriert. Da Pyros Quelltext offen zugänglich ist, ist es möglich, selbst weitere Schnittstellen zur Kooperation von Pyro mit anderen Plattformen und Simulatoren auszuarbeiten.

Ein ähnlicher Ansatz wie der von Pyro wurde am *Ecole Nationale Supérieure de Techniques Avancées* bei der Entwicklung des *U*niversal *R*obotic *B*ody *I*nterface[8] gewählt. Diese Schnittstelle *URBI* ist eine mächtige Low-Level-Skriptsprache, die zur Laufzeit von einem auf dem Roboter installierten Server in den nativen Prozessorcode der Plattform übersetzt wird. Die Steuerbefehle des Roboters können dabei direkt – als Programm auf dem Roboter abgelegt oder per Netzwerk an die Plattform gesendet – in der Sprache URBI erzeugt werden, oder über ein in einer Hochsprache geschriebenes Steuerprogramm, das sich zur Generierung der URBI-Befehle der Bibliothek *Liburbi* bedient. Dieses *Klient* genannte Steuerprogramm kann auf einem entfernten System oder der Plattform selbst ausgeführt werden. Liburbi existiert für C++ (Windows, Linux und MacOS) und für Java, Liburbi für OPEN-R und MATLAB sollen folgen. Den URBI-Server gibt es zur Zeit nur für Aibo, die Veröffentlichung des Quellcodes eines URBI-Servers für Linux ist aber angekündigt.

### 3.6.2 Kelb

Obwohl das auf Aibo laufende Aperios von Sony als Echtzeitbetriebssystem bezeichnet wird, ist diese Benennung im eigentlichen Sinn des Wortes nicht korrekt, da sich ein Echtzeitbetriebssystem dadurch auszeichnet, dass es für eine Menge von periodisch wiederkehrenden Aufgaben unter bestimmen Bedingungen gewährleisten kann, dass alle diese Aufgaben vor ihrer Deadline erledigt werden. Um eine bessere Vorsagbarkeit des zeitlichen Verhaltens der Steuerprogramme für Aibo erreichen zu können, wurde daher an der *Uppsala University* die Umgebung *Kelb* [CFJ+04] ins Leben gerufen. Da es mit erheblichen Schwierigkeiten verbunden ist, das properitäre Aperios-Betriebssystem zu substituieren, wurde in Kelb – ähnlich wie bei dem Tekkotsu-Framework – aufbauend auf dem OPEN-R SDK ein eigenes Prozessmodell realisiert. Im Gegensatz zu Tekkotsu ist Kelb aber mit einem Scheduler ausgestattet, der sich für Echtzeitschedulingverfahren wie *Earliest Deadline First* (EDF) oder auf statische Prioritäten basierende Algorithmen konfigurieren lässt.

[8] `http://www.urbiforge.com/`

### 3.6.3 Microsoft Hellhounds

Ein weiteres interessantes Projekt zur Softwareentwicklung für Aibo wurde durch das RoboCup-Team *Microsoft Hellhounds* der Universität Dortmund begonnen. Ihr Ziel ist es, Microsofts *.NET-Framework* [Voi04], das unter anderem zur komfortablen Vernetzung verschiedener eingebetteter Systeme entworfen wurde, auf den Aibo-Roboter zu übertragen. Dazu ist es ihnen bereits gelungen, Aperios durch einen angepassten Kernel des Betriebssystems *Microsoft Windows CE* auf Aibo zu ersetzen. Der weitere Erfolg des Projekts wird davon abhängen, ob es ihnen auch gelingen wird, einige im Moment nicht bekannte Hardwarespezifikationen Aibos von Sony zu bekommen, um Treiber für den CE-Kernel erstellen zu können.

## 3.7 Schlussfolgerung

Die gewaltige Palette an Möglichkeiten zur Programmierung von Aibo kann eine Entscheidung für eine bestimmte Umgebung zu einer komplizierteren Aufgabe werden lassen. Es wird deswegen hier kurz diskutiert, warum bei dieser Arbeit die Entscheidung für das Tekkotsu-Framework getroffen wurde. Zum Ersten war es wünschenswert, dass bei weiterführenden Projekten an diesem Institut flexibel entschieden werden kann, ob die Steuerprogramme vollständig autonom auf Aibo laufen oder ein entfernter Computer für rechenintensive Prozesse oder gar zur Fernsteuerung mit involviert ist. Es fallen daher das Aibo Remote Framework und die Pyro-Umgebung aus der Auswahl heraus, weil sie die eben genannte Flexibilität auf eine reine Fernsteuerung beschränken. Die Skriptsprache R-CODE ist für universitäre Zwecke in ihren Möglichkeiten zu eingeschränkt und kommt darum auch nicht in Betracht. Es mag anfangs vielleicht so scheinen, dass die am nächsten liegende Lösung ist, das OPEN-R SDK direkt zu verwenden; dies hätte jedoch den Rahmen dieser Arbeit bei weitem überstiegen. Außerdem ist es gerade im wissenschaftlichen Bereich in sehr vielen Fällen sinnvoller, öffentlich zugängliche Ergebnisse anderer zu analysieren und auf diese aufzubauen, anstatt das Rad vollständig selbst neu zu erfinden. Die Alternative zu Tekkotsu wäre die Verwendung von URBI gewesen, welches den Vorteil der Plattformunabhängigkeit gehabt hätte. Das URBI-Projekt ist aber derzeit in seiner Entwicklung noch nicht fortgeschritten genug, um diese Vorteile effektiv nutzen zu können und außerdem ist auch eine Linux-Version des Tekkotsu-Frameworks, die neben der Simulation von Aibo-Steuersoftware auch die Verwendung von Tekkotsu auf anderen Robotersystemen verspricht, in weiten Teilen bereits vollendet.

# Kapitel 4

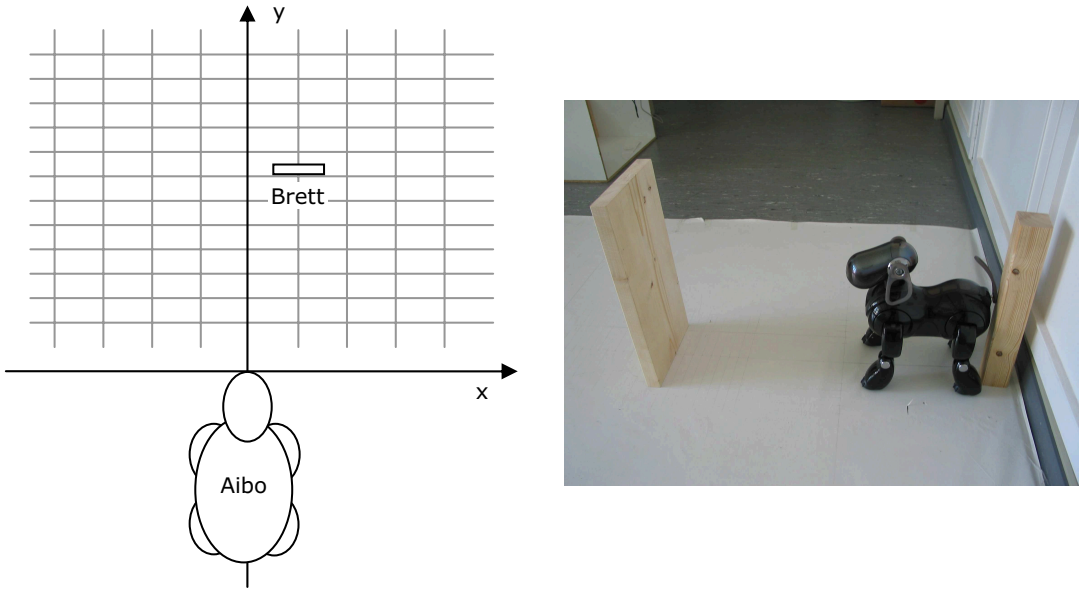
# Experimente

In diesem Kapitel werden die Ergebnisse der im Rahmen dieser Studienarbeit durchgeführten Experimente zu Aibos Sensorik präsentiert. Diese Untersuchungen waren auf die einfacher zu testenden Sensoren, seine zwei Infrarot-Kopfabstandssensoren, beschränkt. In Abschnitt 4.1 ist zunächst der Versuchsaufbau der durchgeführten Experimente beschrieben, in Abschnitt 4.2 werden ihre Ergebnisse präsentiert und ausgewertet.

## 4.1 Versuchsaufbau

Ziel der Untersuchungen zu den Kopfabstandssensoren war nicht nur, die Genauigkeit von ihnen zu bestimmen, sondern auch möglicherweise auftretende Abweichungen der Sensorwerte in Abhängigkeit von der Position eines Gegenstands in Aibos Gesichtsfeld zu ermitteln. Dazu wurde Papier einer etwa 1,50 m breiten Rolle am Boden des Raums angebracht und darauf ein Koordinatensystem gezeichnet. Das Experiment bestand dann darin, ein 20 cm breites Brett mit seinem horizontalen Mittelpunkt auf verschiedene Punkte des Koordinatensystems zu stellen und Aibo den Abstand zu diesem Brett messen zu lassen. Zum Festlegen der Messpunkte wurde ein Raster in das Koordinatensystem eingetragen. Aibo wurde für das Experiment auf dem Papier so positioniert, dass sein Kameramittelpunkt im Ursprung des Koordinatensystem liegt und er entlang der y-Achse geradeaus blickt. Der eben beschriebene Versuchsaufbau ist in Abbildung 4.1 dargestellt.

Die Positionierung von Aibo im Koordinatenursprung bereitete einige Schwierigkeiten. Während sich der Mittelpunkt der Kamera durch ein an einer Ecke des Koordinatenursprungs stehenden Bretts noch sehr genau ausrichten lies, gestaltete es sich schon um einiges problematischer, ihn auch genau in die gewünschte Orientierung zu bringen. Bei der Untersuchung des Kopfabstandssensors für weite Entfernungen wurde er vorn mit seiner Kamera an dem in der Ecke des




(a) Schematische Darstellung des Versuchsaufbaus.

(b) Fotografie des Aufbaus.

**Abbildung 4.1:** Versuchsaufbau zum Testen von Aibos Infrarotsensoren.

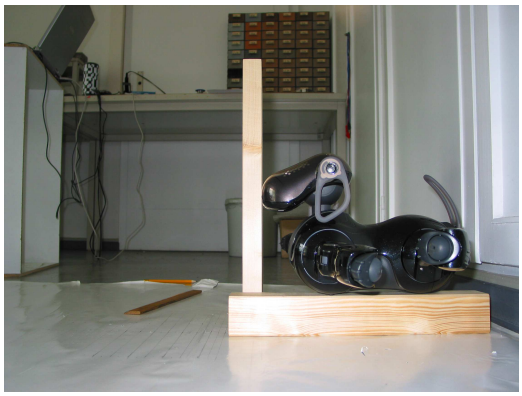
Koordinatenursprung stehenden Brett und hinten mit seiner Niete am Schwanz an einem an der y-Achse aufgestellten Holzpfahl ausgerichtet (siehe Abbildung 4.2(a)). Diese Methode stellte sich aber im Laufe des Experiments als sehr mühsam heraus. Deshalb wurde Aibo für die Untersuchung des Kopfabstandssensors für nahe Entfernungen mit abstehenden Beinen mit seinem Bauch auf einen Holzpfahl gelegt. Da dieser nur ein sehr kleines Stückchen breiter ist als Aibos Bauch, konnte er auf ihm sehr genau positioniert werden. Der Holzpfahl lies sich dann sehr leicht mit seiner horizontalen Mitte vorne und hinten an der y-Achse des Koordinatensystems ausrichten (siehe Abbildung 4.2(b)).

## 4.2 Auswertung

In Abbildung 4.4 sind die Ergebnisse der Tests der Kopfabstandssensoren dargestellt. Die dreidimensionalen Graphen geben an der z-Achse für einen Punkt $(x, y)$ an, welcher Wert von dem entsprechenden Sensor gemessen wurde, als das Brett mit seinem horizontalen Mittelpunkt an diesem Punkt positioniert wurde. Es handelt sich hierbei um Mittelwerte von je 200 Messungen, die mit einem Abtastintervall von 50 ms zueinander getätigt worden sind.

Der im Idealfall zu erwartende, also wünschenswerte Ausgang der Untersuchung des Kopfabstandssensors für weitere Entfernungen ist in Abbildung 4.5(a) dargestellt. Das entsprechende Diagramm für den Infrarotsensor für nahe Entfer-

(a) Für den Test des Kopfabstandssensors für weitere Entfernungen wurde Aibo an der Niete an seinem Schwanz und an seiner Kamera ausgerichtet.

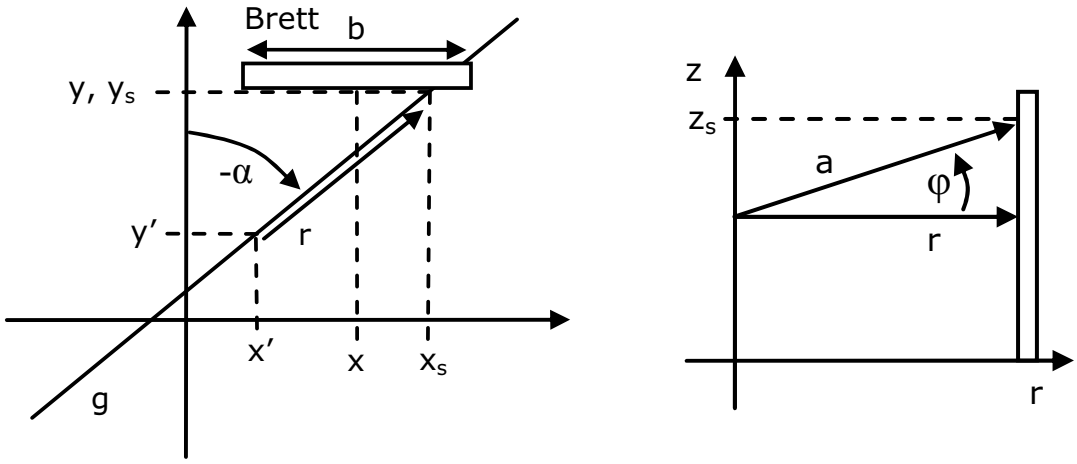


(b) Bei der Untersuchung des Kopfabstandssensors für nahe Entfernungen konnte Aibo durch eine geeignetere Vorgehensweise genauer positioniert werden.

**Abbildung 4.2:** Das genaue Positionieren von Aibo im Koordinatensystem brachte einige Schwierigkeiten mit sich.

nungen ergibt sich analog. Bei dieser Darstellung wurde von einem in Richtung der y-Achse zeigenden und im Urpsprung des Koordinatensystems liegenden Sensor ausgegangen; es wurde also nicht berücksichtigt, dass der Austrittspunkt von Aibos Infrarotstrahlen im Kopf nicht mit dem Mittelpunkt der Kamera übereinstimmt, sondern gegenüber diesem verschoben ist. Außerdem wurde nicht in Betracht gezogen, dass Aibos Sensoren möglicherweise nicht senkrecht nach vorne, sondern in einem Winkel $\alpha$ zur Seite und einem Winkel $\varphi$ nach oben abstrahlen könnten. Es soll deshalb für beide Kopfabstandssensoren eine Funktion $a_{\alpha,\varphi}(x, y)$ hergeleitet werden, die – wie der in Abbildung 4.5(a) dargestellte Graph – für einen Punkt $(x, y)$ den von Aibo zu erwartenden Messwert angibt, wenn ein Brett mit seinem Mittelpunkt auf diesen Punkt gestellt wird. Diese soll aber im Gegensatz zu der in 4.5(a) abgebildeten Funktion auch berücksichtigen, dass der jeweilige Infrarotsensor in einem Winkel $\alpha$ nach links und einem Winkel $\varphi$ nach oben abstrahlt und dass der Austrittspunkt des Sensorstrahls um einen Vektor $\begin{pmatrix} x' & y' \end{pmatrix}^T$ gegenüber dem Koordinatenursprung verschoben ist. Die Funktion $a_{\alpha,\varphi}(x, y)$ wird benötigt, um durch Korrelationsanalyse mit ihr und den gemessenen Werten die Parameter $\alpha$ und $\varphi$ bestimmen und darauf aufbauend sinnvolle Sollwerte der Messung definieren zu können. Daraus können dann die Messwerte der Sensoren in Abweichungen von gewünschten Sollwerten an der Stelle $(x, y)$ transformiert werden und so die Verteilungsfunktion der Zufallsgröße geschätzt werden, welche die zu erwartende Sensorabweichung angibt.

Das Ziel beim Herleiten der Funktion $a_{\alpha,\varphi}(x, y)$ ist es, allgemein für beide Kopfabstandssensoren den von Aibo zu messenden Abstand $a$ (siehe Abbildung 4.3(b)) in Abhängigkeit von der momentanen Position $(x, y)$ des Mittelpunktes




(a) (b)

**Abbildung 4.3:** Über die Parameter $\alpha$, $\varphi$, $x'$, $y'$ kann der von Aibo zu messende Abstand in Abhängigkeit von der momentanen Position $(x, y)$ des Mittelpunkts eines Bretts der Breite $b$ bestimmt werden.

des Bretts und der Parameter $\alpha$, $\varphi$, $x'$ und $y'$ (siehe Abbildung 4.3) auszudrücken. Dazu wird im ersten Schritt der Vektor $\vec{a}$ des Infrarotstrahls, der ausgehend vom Austrittspunkt $(x', y')$ des Sensorstrahls auf den Punkt $(x_s, y_s, z_s)$ des Bretts zeigt, in die x-y-Ebene auf den Vektor $\vec{r}$ (siehe Abbildung 4.3(b)) projiziert. Die zu bestimmende Länge $a$ des Vektors ergibt sich in Abhängigkeit des Betrags von $\vec{r}$ und des Winkels $\varphi$ zu

$$a = \frac{r}{\cos\varphi}. \tag{4.1}$$

Alle weiteren Betrachtungen können nun in der in Abbildung 4.3(b) skizzierten x-y-Ebene erfolgen. Zur Bestimmung des Abstands $r$ wird die Gerade $g$, auf der Vektor $\vec{r}$ liegt, ermittelt. Diese Gerade hat den Anstieg $\tan(\alpha + \frac{\pi}{2}) = \tan^{-1}\alpha$ und schneidet den Sensorstrahlaustrittspunkt $(x', y')$. Ihre Geradengleichung ergibt sich somit zu

$$y = \frac{(x - x')}{\tan\alpha} + y'. \tag{4.2}$$

Diese Gerade charakterisiert den Verlauf des auf die x-y-Ebene projizierten Sensorstrahls. Über ihren Schnittpunkt $(x_s, y_s)$ mit der Geraden $y$ – wobei $y$ in diesem Fall die momentane Position des Bretts in vertikaler Richtung ist – kann der Abstand $r$ mit dem Satz des Pythagoras als

$$r = \sqrt{(y_s - y')^2 + (x_s - x')^2} \tag{4.3}$$

angegeben werden. Wie aus Abbildung 4.3(a) zu erkennen, ergibt sich $y_s$ dabei aus der momentanen vertikalen Position des Bretts zu $y_s{=}y$ und der Wert $x_s$ über

die Geradengleichung von $g$ zu

$$x_s = (y - y') \tan \alpha + x'. \tag{4.4}$$

Es kann nun die Funktion $a_{\alpha,\varphi}(x, y)$ angegeben werden. Dabei ist jedoch noch zu beachten, dass der Sensor seinen Messbereich nicht überschreiten kann und sie daher ab einem Wert $a_{\max}$ abgeschnitten werden muss. Sie nimmt außerdem den Wert $a_{\max}$ an, wenn $|x - x_s| > \frac{1}{2}b$ gilt, Aibos Infrarotsensorstrahl also nicht auf das Brett, sondern daran vorbei zeigt:

$$\begin{aligned} a_{\alpha,\varphi}(x, y) = &\begin{cases} a'_{\alpha,\varphi}(x, y) & \text{wenn } |x - x_s| \leq \frac{1}{2}b \text{ und } a'_{\alpha,\varphi}(x, y) \leq a_{\max} \\ a_{\max} & \text{sonst} \end{cases} \\ &\text{mit } a'_{\alpha,\varphi}(x, y) = \frac{\sqrt{(y - y')^2 + (x_s - x')^2}}{\cos \varphi} \\ &\text{und } x_s = (y - y') \tan \alpha + x'. \end{aligned} \tag{4.5}$$

Die Parameter $x'$ und $y'$ konnten für die beiden Sensoren direkt aus dem Datenblatt von Sony zum Aibo ERS-7 [Son04d] entnommen werden; die Winkel $\alpha$ und $\varphi$ wurden, wie bereits erwähnt, über eine Korrelationsanlyse ermittelt. Diese bestand darin, die Funktion $a_{\alpha,\varphi}(x, y)$ für verschiedene Winkel $\alpha$ und $\varphi$ über den diskreten Korrelationsindex

$$\rho_{\alpha,\varphi} = \sum_x \sum_y \left(a_{\alpha,\varphi}(x, y) - \overline{a_{\alpha,\varphi}}\right) \left(I(x, y) - \bar{I}\right) \tag{4.6}$$

mit den gemessenen Werten $I(x, y)$ zu vergleichen. Die Funktionen $a_{\alpha,\varphi}(x, y)$ und $I(x, y)$ mussten dafür von ihrem jeweiligen Mittelwert $\overline{a_{\alpha,\varphi}}$ beziehungsweise $\bar{I}$ befreit werden. Als Lösung des Verfahrens wurde dann das Wertepaar $(\alpha, \varphi)$ angenommen, für das $\rho_{\alpha,\varphi}$ maximal ist. In Tabelle 4.1 sind für beide Kopfabstandssensoren *NEAR IR* und *FAR IR* die ermittelten Parameter aufgeführt. Die Funktion $a_{\alpha,\varphi}(x, y)$ ist in Abbildung 4.5(b) mit den Paramtern für den Abstandssensor für weite Entfernungen *FAR IR* dargestellt.

Es konnten nun von den gemessenen Werten der jeweils entsprechende über die Funktion $a_{\alpha,\varphi}(x, y)$ definierte Sollwert subtrahiert werden. Daraus wurden mit der Funktion *cdfplot(..)* von MATLAB kumulative empirische Verteilungsfunktionen der Zufallsgröße des Sensorfehlers erzeugt. Diese sind für den Sensor *FAR IR* in Abbildung 4.6 und für den Sensor *NEAR IR* in Abbildung 4.7 dargestellt. Die Unterabbildung (a) gibt dabei jeweils die bei Einbeziehung aller ermittelten Messwerte bestimmte Verteilungsfunktion an, die Unterabbildung (c) eine Verteilungsfunktion, bei deren Bestimmung nur die Werte für $x = 0$ berücksichtigt worden sind. Der von der Funktion aus Unterabbildung (c) angegebene Wert kann also als die Wahrscheinlichkeit eines bestimmten Sensorfehlers unter der Bedingung, dass sich auf der y-Achse, also geradeaus vor Aibo, ein Gegenstand

**Tabelle 4.1:** Parameterwerte der in 4.2 hergeleiteten Funktion $a_{\alpha,\varphi}(x, y)$ für den Kopfabstandssensor *NEAR IR* für nahe Entfernungen und für *FAR IR*, den Kopfabstandssensor für weitere Entfernungen.

| **Parameter** | **Wert für Sensor *NEAR IR*** | **Wert für Sensor *FAR IR*** |
|---|---|---|
| $x'$ | $0,325\,\text{cm}$ | $-0,695\,\text{cm}$ |
| $y'$ | $-0,416\,\text{cm}$ | $-0,416\,\text{cm}$ |
| $\alpha$ | $-3,4\,\text{Grad}$ | $-3,6\,\text{Grad}$ |
| $\varphi$ | $0\,\text{Grad}$ | $0\,\text{Grad}$ |
| $a_{\max}$ | $50\,\text{cm}$ | $150\,\text{cm}$ |
| $b$ | $20\,\text{cm}$ | $20\,\text{cm}$ |

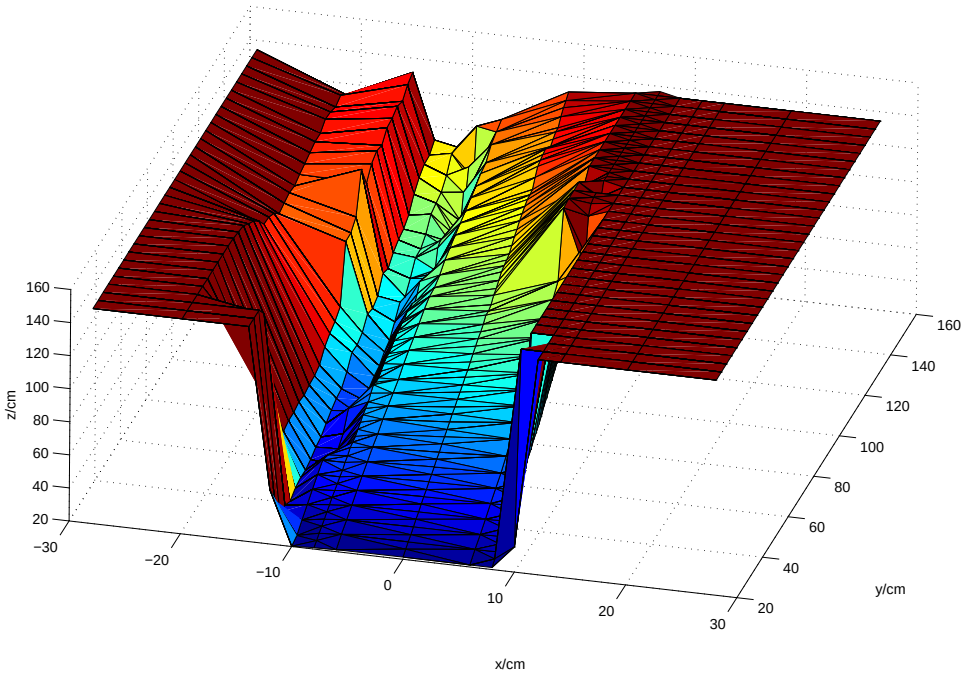
im Messbereich des Sensors befindet, interpretiert werden. In den Unterabbildungen (b) und (d) sind die gleichen Verteilungsfunktionen noch einmal dargestellt, aber diesmal mit der Funktion *normplot(...)* von MATLAB, die die y-Achse in der Art und Weise verzerrt, dass eine Normalverteilung in dieser Darstellung eine Gerade ergeben würde. Diese Diagramme wurden angegeben, weil es in vielen Fällen üblich ist, Sensorfehler durch eine gaußsche Zufallsgröße mit Erwartungswert $\mu = 0$ zu modellieren. Wie aus den Abbildungen zu erkennen ist, wäre das aber in diesem Fall bei Forderung nach einer guten Näherung nicht zu vertreten.

Es ist hier nun die Frage zu diskutieren, inwieweit den soeben präsentierten Ergebnissen Vertrauen geschenkt werden kann und wie diese zu deuten sind. Es wird dazu zuerst auf die rohen Messwerte eingegangen. Die Glattheit der in Abbildung 4.4(b) dargestellten Funktion lässt darauf hindeuten, dass zumindest bei diesem Experiment hinreichend genau genug bei der Bestimmung der Messwerte vorgegangen wurde. Dass die in Abbildung 4.4(a) dargestellten Messwerte des Sensors für längere Abstände deutlich stärker verrauscht sind, kann zwar auf die bei diesem Experiment angewandte schlechtere Methode zur Positionierung von Aibo zurückgeführt werden; es wird hier aber auch von einem stärkeren dem Sensor innewohnenden Rauschen ausgegangen. Unüberschaubare Streu- und Interferenzeffekte bei Auftreffen des Inrarotstrahls auf eine der Kanten des Bretts werden als eine weitere mögliche Fehlerquelle mit in Betracht gezogen. Es können also insgesamt die rohen Messwerte als aussagekräftig angenommen werden.

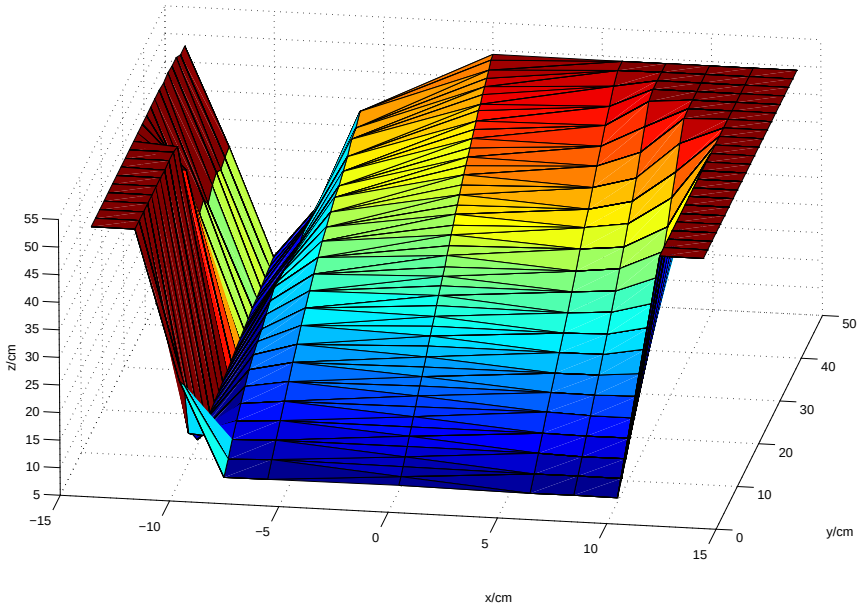
Die Frage, inwiefern den in Abbildung 4.6 und 4.7 dargestellten Verteilungsfunktionen Vertrauen zu schenken ist, ist schon mit mehr Vorsicht zu beantworten. Aber obwohl sie mit ihren zum Teil stark von Null abweichenden Mittelwerten auf den ersten Eindruck absurd erscheinen mögen, so kann es dennoch Ursachen geben, die ihren Verlauf rechtfertigen. Die enorme Ähnlichkeit der sich in den beiden Abbildungen entsprechenden Funktionen deutet zumindest nochmals darauf hin, dass hier nicht von einem Fehler bei der Bestimmung der rohen Messdaten ausgegangen werden kann. Es sollen hier zwei mögliche Denkansätze für Ursachen eines solchen ungleichmäßigen Verlaufs der Verteilungsfunktion,

wie er in den Abbildungen 4.6 und 4.7 zu finden ist, diskutiert werden. Als erste mögliche Ursache wird in Erwägung gezogen, dass beim Aufstellen der Funktion $a_{\alpha,\varphi}(x, y)$ ein bestimmbarer und korrigierbarer Faktor nicht berücksichtigt worden ist und somit einige Messwerte zu unrecht als falsch eingestuft wurden. In diesem Fall würde eine sinnvollere Wahl der Funktion $a_{\alpha,\varphi}(x, y)$ zu Verteilungsfunktionen mit einer geringeren Standardabweichung und vor allem Mittelwerten näher bei Null führen. Der zweite Erklärungsversuch erwägt, dass diese Veläufe durch eine dem Messverfahren des Sensors innewohnende, nicht korrigierbare Eigenschaft begründet sind. Ist beispielsweise die Abbildung der vom Sensor gemessenen Refernzgröße – die im Regelfall eine Spannung ist – auf die vom Sensor anzugebende Größe nichtlinear, so kann eine Auslenkung der Referenzgröße in die eine Richtung eine große Auslenkung der gemessenen Größe bedingen, während die gleiche Auslenkung der Referenzgröße in die andere Richtung eine kleinere Auslenkung der gemessenen Größe bewirkt.

In jedem Fall können die Verteilungsfunktionen unter der Bedingung, dass die Funktion $a_{\alpha,\varphi}(x, y)$ mit den in Tabelle 4.1 abgebildeten Parametern die gewünschten Sollwerte repräsentiert, als aussagekräftig angesehen werden.

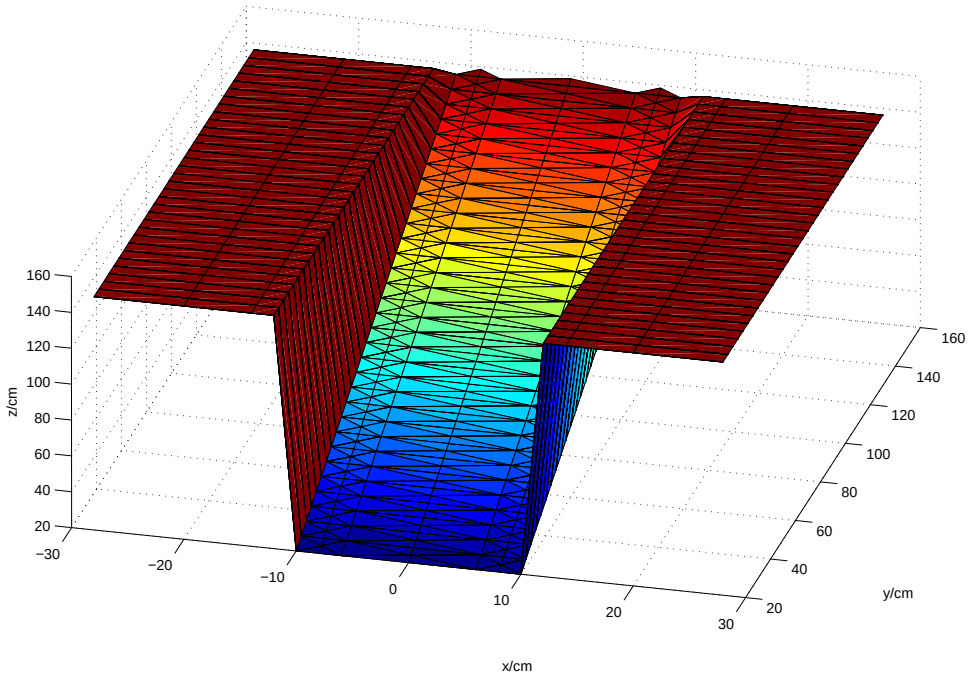



(a) Ergebnisse vom Test des Infrarotabstandssensors für weite Entfernungen. Er hat einen Messbereich von 20 cm bis 150 cm.

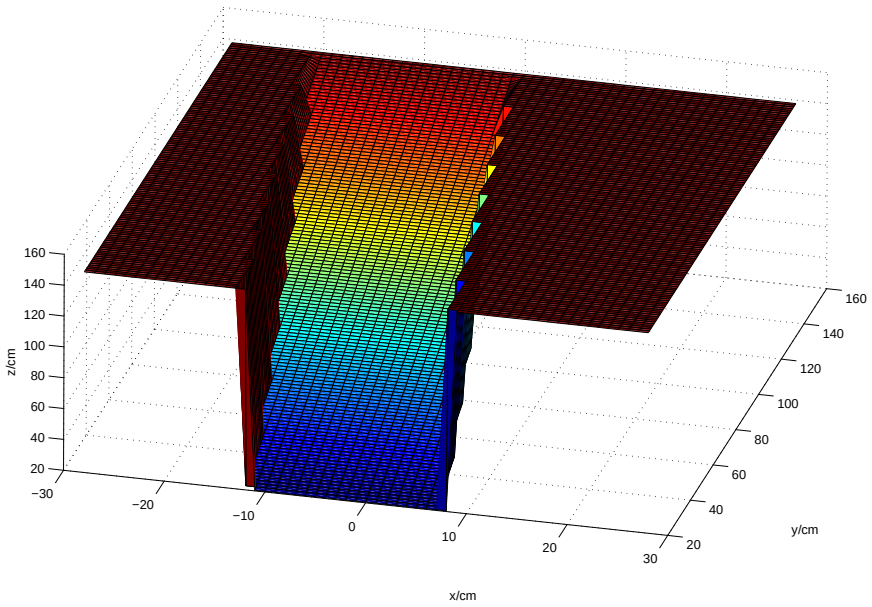



(b) Ergebnisse vom Test des Infrarotabstandssensors für nahe Entfernungen. Er hat einen Messbereich von 5 cm bis 50 cm.

**Abbildung 4.4:** Ergebnisse der in Abschnitt 4.1 vorgestellten Experimente zu den Infrarot-Kopfabstandssensoren von Aibo. Die gemessenen Punkte sind dabei als Gitterpunkte im Graphen zu erkennen. Diese Gitterpunkte wurden durch Linien verbunden.

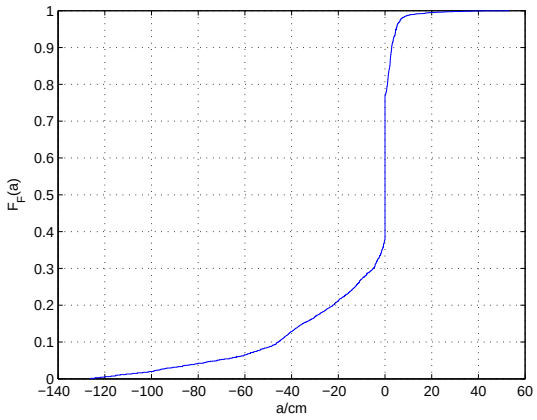



(a)

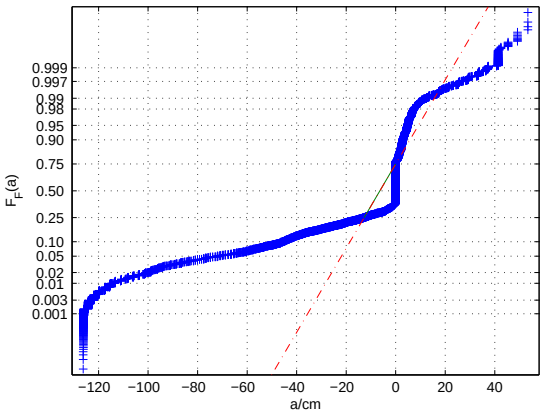



(b)

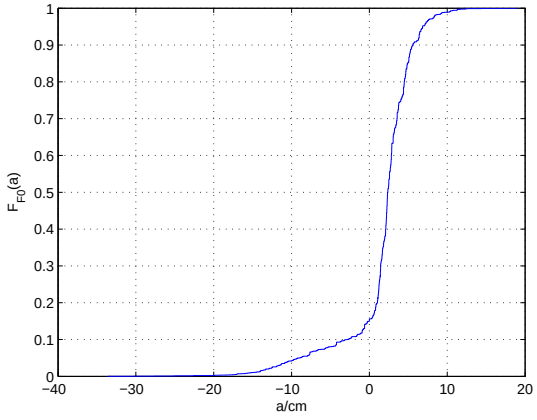
**Abbildung 4.5:** Die im Idealfall zu erwartenden Ergebnisse der in Abschnitt 4.1 vorgestellten Experimente mit Aibos Kopfabstandssensor *FAR IR* für weitere Entfernungen unter Berücksichtigung, dass der Sensoraustrittsstrahl verschoben ist und er in einem Winkel von $3,6$ Grad schräg nach rechts abstrahlt (b) und bei Vernachlässigung dieser Tatsachen (a).



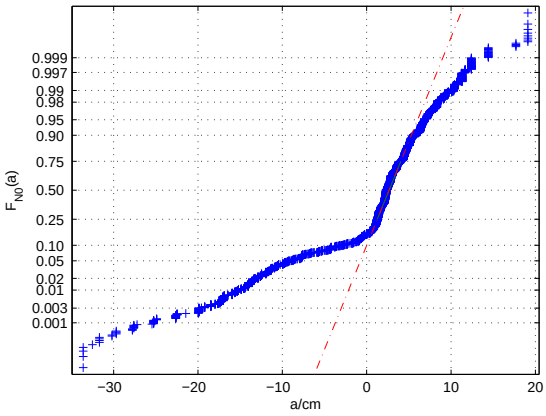

(a) Verteilungsfunktion bei Einbeziehung aller gemessenen Werte. Anzahl der einbezezogenen Werte = 227231. Erwartungswert $\mu = -11,7\,\text{cm}$, Standardabweichung $\sigma = 25,7\,\text{cm}$.




(b) Verteilungsfunktion bei Einbeziehung aller gemessenen Werte. Die y-Achse ist allerdings so verzerrt, dass eine Normalverteilung eine Gerade ergibt.

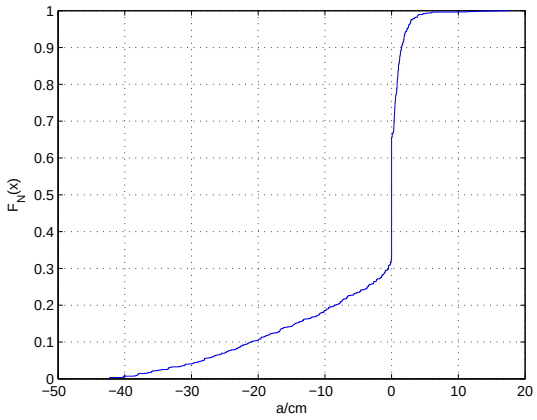



(c) Verteilungsfunktion bei Einbeziehung nur der Werte mit $x = 0$. Anzahl der einbezezogenen Werte = 28281. Erwartungswert $\mu = 1,8\,\text{cm}$, Standardabweichung $\sigma = 4,6\,\text{cm}$.

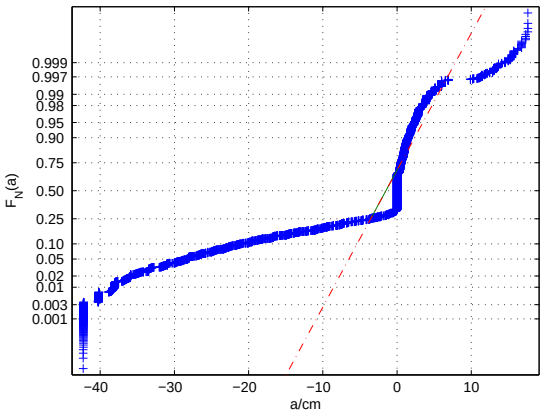



(d) Verteilungsfunktion bei Einbeziehung nur der Werte mit $x = 0$. Die y-Achse ist allerdings so verzerrt, dass eine Normalverteilung eine Gerade ergibt.

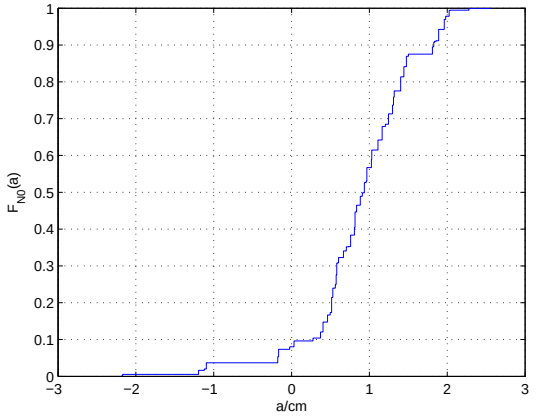
**Abbildung 4.6:** Verteilungsfunktionen der Zufallsgröße der Abweichung des Sensors *FAR IR* für weite Abstände.



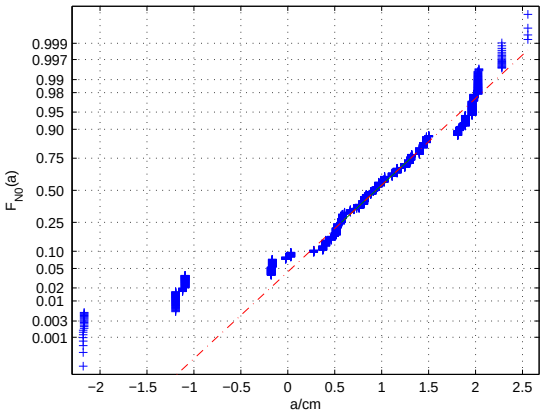

(a) Verteilungsfunktion bei Einbeziehung aller gemessenen Werte. Anzahl der einbezogenen Werte = 55200. Erwartungswert $\mu = -4,3\,\text{cm}$, Standardabweichung $\sigma = 9,9\,\text{cm}$




(b) Verteilungsfunktion bei Einbeziehung aller gemessenen Werte. Die y-Achse ist allerdings so verzerrt, dass eine Normalverteilung eine Gerade ergibt.




(c) Verteilungsfunktion bei Einbeziehung nur der Werte mit $x = 0$. Anzahl der einbezogenen Werte = 4600. Erwartungswert $\mu = 0,7\,\text{cm}$, Standardabweichung $\sigma = 0,9\,\text{cm}$.




(d) Verteilungsfunktion bei Einbeziehung nur der Werte mit $x = 0$. Die y-Achse ist allerdings so verzerrt, dass eine Normalverteilung eine Gerade ergibt.

**Abbildung 4.7:** Verteilungsfunktionen der Zufallsgröße der Abweichung des Sensors *NEAR IR* für nahe Abstände.

# Kapitel 5

# Kinematische Berechnungen mit dem Tekkotsu-Framework

Nach einer überblicksartigen Analyse der vorhandenen Veröffentlichungen zu Ergebnissen der Forschung und Entwicklung anderer Universitäten mit Aibo, musste festgestellt werden, dass eine umfassende Präsentation einer Übersicht der zu Aibo vorhandenen Literatur den Rahmen dieser Arbeit um ein Vielfaches überstiegen hätte. Es wird daher in diesem Kapitel exemplarisch die vom Tekkotsu-Framework zur Verfügung gestellte Funktionalität zur Untersuchung des Bewegungsverhaltens der Glieder von Aibo in Bezug zu seinem Körpermittelpunkt beschrieben. Im Framework wurde zur Realisierung dieser Funktionalität das zur Berechnung von Manipulator-Kinematiken entworfene Paket *ROBOOP* eingebunden. Dieses soll in Abschnitt 5.1 vorgestellt werden. Das Paket ROBOOP basiert auf der *Denavit-Hartenberg-Konvention* und der auf ihr aufbauenden *Denavit-Hartenberg-Transformation*, die in Abschnitt 5.2 respektive 5.4 beschrieben werden. Die Denavit-Hartenberg-Transformation verwendet wiederum sogenannte *homogene Matrizen*, welche in Unterpunkt 5.3 erläutert sind.

## 5.1 Das Paket ROBOOP

Wie in der Einleitung des Kapitels schon angeklungen, bedient sich das Tekkotsu-Framework zur Berechnung des Bewegungsverhaltens von Aibos Gliedern relativ zu einem Bezugspunkt in seiner Körpermitte des von Richard Gourdeau am *Ècole Polytechnique de Montrèaldes* entwickelten Pakets *ROBOOP*[1] [Gou97]. Es wurde zur Kalkulationen der Kinematik von Roboterarmen entworfen, die an einer ihrer Seiten starr an einer Basis befestigt sind. Durch den objektorientierten Ansatz dieser Bibliothek lässt sich eine sehr große Klasse von solchen Manipulatoren mit ihr definieren und berechnen.

[1] A *R*obotics *O*bject *O*riented *P*ackage in C++

Ein *Roboter* ist nach dem Modell der *Denavit-Hartenberg-Konvention*, welches von ROBOOP zur Symbolisierung von Manipulatoren verwendet wird, eine unverzweigte Kette von $n+1$ starren Gliedern, die über $n$ Gelenke miteinander verbunden sind. Ein Gelenk hat ohne Beschränkung der Allgemeinheit[2] nur einen Freiheitsgrad und ist entweder rotatorischer oder translatorischer Natur. Dieses Modell ist in Abbildung 5.1 symbolisch veranschaulicht. Da Aibo aber nicht als unverzweigte Kette von Gliedern darstellbar ist, wird er durch fünf solche *Roboter* modelliert, wobei vier von ihnen seine Beine und einer seinen Kopf repräsentieren. Sie haben dabei alle eine gemeinsame Basis in seinem Körpermittelpunkt. Die Gelenke von Aibo sind alle ausnahmslos rotatorischer Natur. Das von den Entwicklern des Tekkotsu-Frameworks aufgestellte ROBOOP-Modell von Aibo ist in Abbildung 5.2 grafisch dargestellt.

Zur Berechnung der Kinematik dieser Roboter wird in ROBOOP nach der *Denavit-Hartenberg-Konvention*, die in Unterpunkt 5.2 vorgestellt wird, für jedes Glied des Roboters ein Koordinatensystem festgelegt, dass sich bei Veränderung der Gelenkstellungen mitbewegt. Ein Punkt eines Gliedes ist also in dessen Bezugssystem stets der gleiche, da das Glied als unverformbar angenommen wird. Für eine Darstellung der für das ROBOOP-Modell des Aibo festgelegten Koordinatensysteme sei auf die von der Tekkotsu-Homepage entnommenen und in Anhang C abgelegten Diagramme verwiesen. In diesen sind zusätzlich zu den Koordinatensystemen der Glieder noch sogenannte, im Tekkotsu-Framework bereits vordefinierte *Interest Points* eingezeichnet, deren momentane Lage relativ zu Aibos Bezugssystem über ROBOOP bestimmt werden kann.

Nach Aufstellen des Robotermodells kann nun mit ROBOOP über die sogenannte *Denavit-Hartenberg-Transformation*, welche in Abschnitt 5.4 beschrieben ist, für eine bestimmte Gelenkstellung ein Punkt im Koordinatensystems eines Gliedes in ein anderes, zum Beispiel das der Basis des Roboters, transformiert werden. Für den Fall des Aibo bedeutet dies, dass beispielsweise der Punkt einer seiner Pfotenspitzen in Abhängigkeit von einer bestimmten Stellung seiner Gelenke ermittelt und somit ihr Bewegungsverhalten beim Einstellen verschiedener

[2]Reale Gelenke mit Freiheitsgraden höherer Ordnung lassen sich durch eine Verkettung von Gelenken eines Freiheitsgrades modellieren. Die Zwischenglieder dieser Modellgelenke sind in diesem Fall unendlich kleine Punkte.

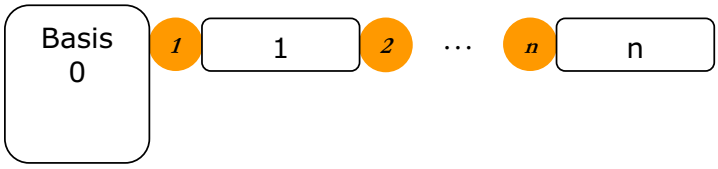



**Abbildung 5.1:** Durch die Denavit-Hartenberg-Konvention wird ein *Roboter* als unverzweigte Kette von $n+1$ Gliedern modelliert, die durch $n$ Gelenke miteinander verbunden sind. Das $i$te Gelenk verbindet dabei das Glied $(i-1)$ mit dem $i$ten.

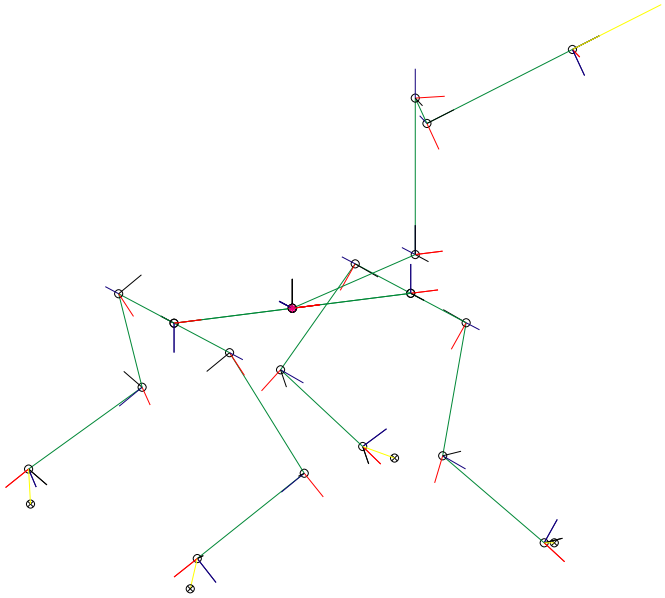


**Abbildung 5.2:** Im Paket ROBOOP ist Aibo durch fünf *Roboterarme* modelliert, die alle ihre Basis in seiner Körpermitte haben.

Stellwerte untersucht werden kann. Die Umwandlung eines Vektors in ein anderes Bezugssystem erfolgt in der Denavit-Hartenberg-Transformation durch Multiplikation mit sogenannten *homogenen Matrizen*, die in Abschnitt 5.3 erklärt sind. Um diese Rechnungen angenehm handhaben zu können, verwendet ROBOOP Robert Davies *Newmat C++ matrix library*, deren Ziel es ist, eine MATLAB nachempfundene Umgebung für C++ zur Verfügung zu stellen.

Zur Berechnung der inversen Kinematik verwendet ROBOOP im Allgemeinfall numerische Algorithmen wie das Newton-Raphson-Verfahren, die hier nicht beschrieben sein sollen.

## 5.2 Denavit-Hartenberg-Konvention

Das in Abbildung 5.1 skizzierte Modell der Denavit-Hartenberg-Konvention eines Roboters ist bereits in Abschnitt 5.1 geschildert worden. Es sollen nun die Regeln angegeben werden, die beim Aufstellen der Koordinatensysteme der Glieder bei Einhaltung der Konvention beachtet werden müssen:

1. Zu Beginn werden für die $n+1$ Glieder des Roboters die z-Achsen $\mathbf{z_i}$ (mit $0 \leq i \leq n$) der Koordinatensysteme festgelegt. Diese müssen für $0 \leq i < n$ die Bewegungsachse des $(i+1)$-ten Gelenks sein; die Achse $\mathbf{z_n}$ kann beliebig gewählt werden.

2. Die Achse $\mathbf{x_i}$ ist für $0 < i \leq n$ so zu bestimmen, dass sie auf der Geraden

liegt, die senkrecht zu den Achsen $\mathbf{z_{i-1}}$ und $\mathbf{z_i}$ steht und zusätzlich diese beiden schneidet. Sind $\mathbf{z_{i-1}}$ und $\mathbf{z_i}$ parallel, gibt es dafür unendlich viele Lösungen; man suche eine sinnvolle davon aus. Die Richtung der $\mathbf{x_i}$-Achse sei immer so bestimmt, dass sie von der $\mathbf{z_{i-1}}$-Achse wegzeigt. Die Achse $\mathbf{x_0}$ ist frei zu wählen.

Diese zweite Forderung mag auf den ersten Blick als nicht besonders sinnvoll erscheinen; sie ist aber sehr brauchbar, weil – wie wir in 5.4 noch sehen werden – in dieser Anordnung die Achsen $\mathbf{z_{i-1}}$ und $\mathbf{z_i}$ durch eine einzige Translation $a_i$ entlang der $\mathbf{x_n}$-Achse und eine Rotation $\alpha_i$ um $\mathbf{x_n}$ auf eine Gerade gebracht werden können.

3. Die Koordinatensysteme aller Glieder $i$ werden für $0 \leq i \leq n$ durch geeignete Wahl der $\mathbf{y_i}$-Achse zu orthonormalen Rechtssystemen ergänzt.

## 5.3 Homogene Koordinaten

Die Transformation eines kartesischen Koordinatensystems in ein anderes lässt sich sehr schön durch die Multiplikation des zu transformierenden Ortsvektors mit der sogenannten Transformationsmatrix beschreiben. Dies setzt aber voraus, dass sämtliche Operationen der Transformation rotatorischer und nicht translatorischer Natur sind; denn die Addition mit einem Vektor im $\mathbb{R}^n$ ist keine lineare Operation. Es ist also wünschenswert, rotatorische und translatorische Transformationen einheitlich durch eine Matrix $\mathbf{T}$ beschreiben zu können.

Dies wird durch sogenannte *homogene Transformationsmatrizen*, für deren Anwendung die zu transformierenden Vektoren durch einen Trick zeitweilig um eine weitere Dimension in *homogenen Koordinaten* erweitert werden, ermöglicht. Wie dies geschieht, wird nun anhand des dreidimensonalen kartesischen Koordinatensystems exemplarisch beschrieben. Zur Hintransformation eines Vektors

$$\vec{v} = \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

in homogene Koordinaten wird die folgende Abbildungsvorschrift verwendet:

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} \multimap \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} \tag{5.1}$$

Die Bedeutung der vierten Koordinate von homogenen Koordinaten lässt sich

erschließen, wenn man die Vorschrift zur Rücktransformation eines Vektors

$$\vec{v}' = \begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix}$$

in homogenen Koordinaten zurück in gewöhnliche kartesische Koordinaten betrachtet:

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} \bullet\!\!-\!\!\circ \begin{pmatrix} \frac{x'}{w'} \\ \frac{y'}{w'} \\ \frac{z'}{w'} \end{pmatrix} \tag{5.2}$$

Die vierte Komponente $w'$ eines homogenen Vektors ist also eine reele Zahl, die den zugehörigen Vektor in gewöhnlichen kartesischen Koordinaten indirekt proportional skaliert. Wendet man homogene Transformationsmatrizen an, die nur rotatorische und translatorische Anteile haben, bleibt diese Komponente stets Eins. In einer homogenen Transformationsmatrix sind nämlich zusätzlich zur Rotation und Translation auch eine lineare Streckung und Stauchung der einzelnen Achsen sowie eine achsenunabhängige Skalierung des dreidimensionalen Vektorraums beschreibbar.

Zur Veranschaulichung der Funktionsweise einer homogenen Transformationsmatrix $\mathbf{T}$ werden nun die Bedeutungen ihrer einzelnen Elemente erläutert. Sie kann in folgender allgemeiner Form geschrieben werden:

$$\mathbf{T} = \begin{pmatrix} \mathbf{R} & \vec{t} \\ \vec{d} & s \end{pmatrix} \tag{5.3}$$

Die Komponente $\mathbf{R}$ gibt dabei ihren rotatorischen Anteil, wie von nichthomogenen Koordinaten gewohnt, als $3 \times 3$-Matrix an; der Spaltenvektor

$$\vec{t} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}$$

beschreibt den translatorischen Anteil der Transformation in ihren einzelnen Koordinaten; der Zeilenvektor

$$\vec{d} = \begin{pmatrix} d_x & d_y & d_z \end{pmatrix}$$

gibt die oben genannte lineare Verzerrung der jeweiligen Achsen und die reele Zahl $s$ die ebenfalls bereits genannte indirekt proportionale Skalierung des Vektorraums an. Bei der für diese Arbeit verwendete Klasse von Transformationsmatrizen sind allerdings die Elemente $d_x$, $d_y$ und $d_z$ stets gleich Null, sowie der Paramter $s$ stets gleich Eins.

Um die gerade beschriebenen Bedeutungen der einzelnen Komponenten einer homogenen Transformationsmatrix verinnerlichen zu können, empfiehlt es sich, die homogene Transformation selbstständig anhand von Beispielen nachzuvollziehen. Aus diesem Grund seien als Abschluss dieser knappen Einführung in homogene Koordinaten noch vier Basistypen von oft auftretenden homogenen Transformationsmatrizen angegeben:

1. Translation:
$$\mathbf{T}(t_x, t_y, t_z) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.4}$$

2. Rotation um die x-Achse:
$$\mathbf{R_x}(\alpha) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & -\sin\alpha & 0 \\ 0 & \sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.5}$$

3. Rotation um die y-Achse:
$$\mathbf{R_y}(\alpha) = \begin{pmatrix} \cos\alpha & 0 & \sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.6}$$

4. Rotation um die z-Achse:
$$\mathbf{R_z}(\alpha) = \begin{pmatrix} \cos\alpha & -\sin\alpha & 0 & 0 \\ \sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.7}$$

## 5.4 Denavit-Hartenberg-Transformation

Es wird jetzt die homogene Matrix $\mathbf{T}_{i-1}^{i}$ bestimmt, die das Bezugssystem des Glieds $i-1$ in das des Glieds $i$ eines *Roboters* im Sinne der Denavit-Hartenberg-Konvention transformiert. Als Transformation der Koordinatensysteme sei hierbei verstanden, dass man bei Multiplikation der Transformationsmatrix $\mathbf{T}_{i-1}^{i}$ mit einem Vektor im Koordinatensystem $i$ den entsprechenden Vektor im Koordinatensystem $i-1$ erhält.

Die Denavit-Hartenberg-Transformation $\mathbf{T}_{i-1}^{i}$ zur Umwandlung des Bezugssystems $i-1$ in das Koordinatensystem $i$ kann in vier einzelne Operationen untergliedert werden:

1. Eine Rotation des Gelenkwinkels $\theta_i$ um die $\mathbf{z_{i-1}}$-Achse:

$$\mathbf{R}_{\mathbf{z_{i-1}}}(\theta_i) = \begin{pmatrix} \cos\theta_i & -\sin\theta_i & 0 & 0 \\ \sin\theta_i & \cos\theta_i & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.8}$$

2. Eine Translation der Gelenkverschiebung $d_i$ entlang der $\mathbf{z_{i-1}}$-Achse:

$$\mathbf{T}_{\mathbf{z_{i-1}}}(d_i) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.9}$$

3. Eine Translation der Lageverschiebung $a_i$ entlang der durch die Schritte 1 und 2 entstandene neuen x-Achse, der Achse $\mathbf{x_i}$:

$$\mathbf{T}_{\mathbf{x_i}}(a_i) = \begin{pmatrix} 1 & 0 & 0 & a_i \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.10}$$

4. Eine Rotation des Lagewinkels $\alpha_i$ um die $\mathbf{x_i}$-Achse:

$$\mathbf{R}_{\mathbf{x_i}}(\alpha_i) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha_i & -\sin\alpha_i & 0 \\ 0 & \sin\alpha_i & \cos\alpha_i & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{5.11}$$

Die Transformationsmatrix $\mathbf{T}_{i-1}^{i}$ ergibt sich damit zu

$$\begin{aligned} \mathbf{T}_{i-1}^{i} &= \mathbf{R}_{\mathbf{z_{i-1}}}(\theta_i) \cdot \mathbf{T}_{\mathbf{z_{i-1}}}(d_i) \cdot \mathbf{T}_{\mathbf{x_i}}(a_i) \cdot \mathbf{R}_{\mathbf{x_i}}(\alpha_i) \\ &= \begin{pmatrix} \cos\theta_i & -\sin\theta_i\cos\alpha_i & \sin\theta_i\sin\alpha_i & a_i\cos\theta_i \\ \sin\theta_i & \cos\theta_i\cos\alpha_i & -\cos\theta_i\sin\alpha_i & a_i\sin\theta_i \\ 0 & \sin\alpha_i & \cos\alpha_i & d_i \\ 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{5.12}$$

In den ersten beiden Schritten der Transformation wird das Koordinatensystem des Glieds $i-1$ um den aktuellen Drehwinkel $\theta_i$ des Gelenks rotiert und um die aktuelle Gelenkverschiebung $d_i$ bewegt. Für ein Rotationsgelenk ist dabei der Parameter $\theta_i$, für ein Translationsgelenk $d_i$ variabel. Der jeweils andere feste Parameter ist im allgemeinen Fall notwendig, um bei der Transformation die Ursprünge der Koordinatensysteme in Übereinstimmung bringen zu können. Diese Notwendigkeit ist dadurch begründet, dass zwar die Bewegungsachsen $\mathbf{z_{i-1}}$ der $n$

Gelenke frei wählbar, die Ursprünge der Koordinatensysteme der Glieder 1 bis $n$ aber dann durch die Denavit-Hartenberg-Konvention – außer bei parallelen Achsen $\mathbf{z_{i-1}}$ und $\mathbf{z_i}$ – eindeutig bestimmt sind. Daraus resultiert auch, dass der Nullwert des variablen Parameters im Allgemeinfall durch die Denavit-Hartenberg-Konvention vorgegeben ist. Zur besseren Handhabung kann hier in einigen Fällen die Einführung eines festen Offset-Wertes für den variablen Parameter sinnvoll sein.

In den Schritten drei und vier wird der Lageunterschied der Achse $\mathbf{z_{i-1}}$ zur Achse $\mathbf{z_i}$ ausgeglichen. Da nach den Schritten eins und zwei die neue x-Achse unseres zur Hälfte transformierten Koordinatensystems bereits auf der des Gliedes $i$ liegt und diese Achse $\mathbf{x_i}$ wie in Abschnitt 5.4 gefordert gelegt wurde, kann dies durch eine Translation $a_i$ entlang von $\mathbf{x_i}$ und eine Rotation $\alpha_i$ um $\mathbf{x_i}$ vollzogen werden.

Es wurde nun die homogene Matrix $\mathbf{T}_{i-1}^{i}$ zur Transformation eines Koordinatensystems des Glieds $i-1$ in das des Glieds $i$ bestimmt. Durch geeignete Multiplikation dieser Basistransformationsmatrizen kann eine Matrix zur Transformation zwischen Koordinatensystemen beliebiger Glieder erzeugt werden. Weitere Informationen zur Berechnung des Bewegungsverhaltens von Manipulatoren sind unter anderem in [Sch97] zu finden. Als Tipp zum Nachvollziehen der Denavit-Hartenberg-Transformation sei außerdem abschließend noch erwähnt, dass sie auch rückwärts gelesen werden kann; also nicht als eine Transformation des Koordinatensystems des Glieds $i-1$ in das des Glieds $i$ betrachtet wird, sondern als eine Umwandlung eines Ortsvektors im Koordinatensystem des Glieds $i$ in den ihm entsprechenden Punkt im Bezugsystem $i-1$.

# Kapitel 6

# Navigationsmodul

In diesem Kapitel wird das bei dieser Arbeit entstandene Navigationsmodul beschrieben und hinsichtlich seiner Fähigkeiten kritisch beleuchtet. Es wird dafür zuerst in Abschnitt 6.1 der Algorithmus zur Navigation des Roboters allgemein skizziert. Darauf folgend wird in Unterpunkt 6.2 die für diesen Algorithmus verwendete Vorgehensweise zur Landmarkenerkennung erläutert und in Abschnitt 6.3 ist dann die Implementierung des Zustands *Track Landmark* (siehe Abbildung 6.1), in dem Aibo die gefundene Landmarke anläuft, genauer charakterisiert. Abgeschlossen wird das Kapitel durch die Ergebnisse vom Funktionstest des Navigationsmoduls. Das UML-Klassendiagramm des Navigationsmoduls ist in Anhang B.2 zu finden.

## 6.1 Algorithmus

Um auch bei der Entwicklung von komplizierteren Roboter-Steuerkonzepten die Übersicht wahren zu können, bietet sich in vielen Fällen zur Modellierung des zu entwerfenden Algorithmus ein Zustandsdiagramm an. Aus diesem Grund sind dazu im Tekkotsu-Framework die von *BehaviorBase* abgeleiteten Klassen *StateNode* und *Transition* vorgesehen, welche eine direkte Implementierung einer hierarchischen Zustandsmaschine mit ihm gestatten. *Hierarchisch* bedeutet hierbei, dass ein Zustand selbst wieder durch eine Zustandsmaschine bestimmt, also er selbst eine Zustandsmaschine sein kann.

Der hier beschriebene Navigationsalgorithmus wurde mit der von Tekkotsu vorgesehenen Methodik zur Programmierung von Zustandsmaschinen verwirklicht. Deshalb soll diese Methodik hier umrißhaft erläutert werden. Eine von *StateNode* abgeleitete Klasse ist ein Behavior, das das Verhalten des Roboters in dem durch diese Klasse repräsentierten Zustand charakterisiert. Aber auch die Transitionen sind in Tekkotsu Behaviors, welche bestimmte Ereignisse überwachen und dabei prüfen, ob ihre Schaltbedingung erfüllt ist. Will der Programmie-

rer einer Transition diese zu einem bestimmten Zeitpunkt schalten lassen, muss er die in ihrer Basisklasse *Transition* implementierte Funktion *fire()*, die ihre Quellzustände über die Methode *DoStop()* beendet sowie all ihre Zielzustände über *DoStart()* aktiviert, aufrufen. Es sind dazu Zeiger auf all ihre Quell- und Zielknoten in der Transition gespeichert. Ein Zustandsautomat ist vollständig durch seine Zustände, seine Transitionen und die Menge der gerichteten Kanten zwischen Zuständen und Transitionen gekennzeichnet. Dennoch enthält im Tekkostu-Framework auch ein Zustand Pointer auf die von ihm ausgehenden Transitionen, um sie ebenfalls bei seinem Start über ihre Funktion *DoStart()* aktivieren und bei seiner Deaktivierung über *DoStop()* beenden zu können. Es sei hier abschließend noch einmal betont, dass eine Transition nach dem Modell von Tekkotsu **mehrere** Ziel- und Ausgangszustände haben kann und sich somit auch Nebenläufigkeiten modellierende Zustandsautomaten wie etwa steuerungstechnisch interpretierte Petri-Netze unmittelbar mit dem Tekkotsu-Framework realisieren lassen.

Zur Schilderung des hier verwendeten Algorithmus wird nun zunächst in Unterabschnitt 6.1.1 der Knoten *Navigation Behavior*, der die oberste Ebene des hierarchischen Zustandsmodells repräsentiert, beschrieben. Es folgt darauf in 6.1.2 die Darstellung seiner Unterzustandsmaschine *Search For Landmark*, deren Knoten *Scan Environment* abschließend in Unterpunkt 6.1.3 wiederum als Zustandsautomat ausführlicher charakterisiert ist. Als Bezeichnungen der Knoten und Transitionen wurden direkt die Original-Begriffe des Quelltextes gewählt, der vollständig in Englisch verfasst wurde, um eine Mischung von Deutsch und Englisch in ihm zu vermeiden.

### 6.1.1 Navigation Behavior

Wie aus Abbildung 6.1 zu ersehen, besteht das *Navigation Behavior* aus den drei Knoten *Configure*, *Search For Landmark* und *Track Landmark*. Der Algorithmus startet dabei im Zustand *Configure*, in dem die Folge der zu findenden Landmarken festgelegt wird. Diese ist momentan direkt durch das Programm vorgeschrieben, welches aber so erweitert werden kann, dass sie durch Interaktion mit dem Benutzer bestimmt wird. Ist die Konfiguration beendet, geht das *Navigation Behavior* in den Zustand *Search For Landmark* über, der in 6.1.2 vorgestellt wird. Wenn Aibo die Landmarke dann gefunden hat, läuft er im Zustand *Track Landmark* auf sie zu; verliert er sie indes, beginnt er die Suche nach ihr erneut. Das Verfahren zum Ansteuern der Landmarke ist in Abschnitt 6.3 gesondert erläutert. Aibos Aufgabe wird für eine Landmark als absolviert angesehen, wenn beim Anlaufen von ihr sein Kopfabstandssensor *NEAR IR* für nahe Entfernungen einen Wert $< 70\,\text{mm}$ oder sein Brustabsstandssensor *CHEST IR* einen Wert $\leq 100\,\text{mm}$ oder $> 270\,\text{mm}$ meldet. Dann kehrt der Algorithmus wieder in

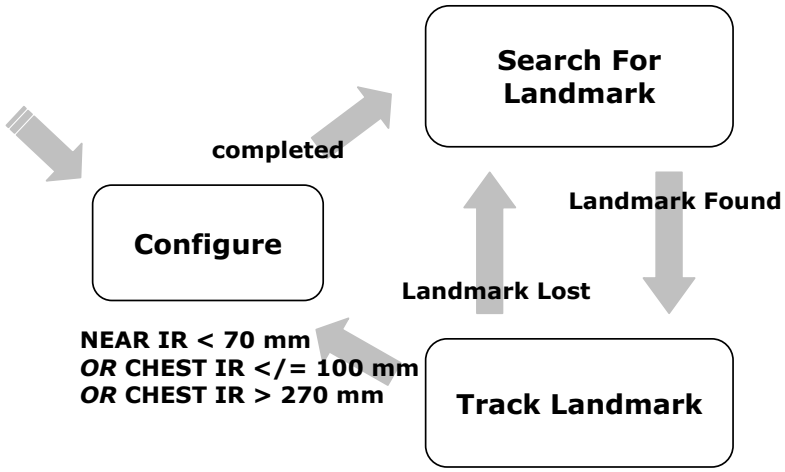



**Abbildung 6.1:** Der Zustand *Navigation Behavior*, der die oberste Ebene der hierarchischen Zustandsmaschine des Navigationsmoduls umspannt. Es wurde in diesem Kapitel eine abstrakte, intuitiv zu verstehende Notation zur Darstellung der Zustandsmodelle gewählt. Der gesamte Automat ist in Anhang B.1 als UML-Zustandsmaschine abgebildet.

den Zustand *Configure* zurück, in dem eine neue aktuell zu findende Landmark gesetzt oder das Programm bei der letzten Landmarke beendet wird.

### 6.1.2 Search For Landmark

Das in Abbildung 6.2 illustrierte Suchen nach der momentan zu findenden Landmarke beginnt damit, dass Aibo wie in 6.1.3 beschrieben die Gegend untersucht. Ist er damit fertig, läuft er eine Strecke geradeaus, deren Weg über eine zwischen 200 mm und 2000 mm gleichverteilte Zufallsvariable bestimmt wird. Dieser Vorgang wird vorzeitig abgebrochen, wenn er ein Hindernis oder einen Abgrund erkennt, das heißt sein Kopfabstanssensor den Wert 120 mm unterschreitet oder sein Brustabstandssensor den Wert 115 mm unter- oder den Wert 210 mm überschreitet. Es beginnt dann der eben dargelegte Zyklus von neuem.

### 6.1.3 Scan Environment

Zum Zeitpunkt des Algorithmusentwurfs konnte noch nicht genau bestimmt werden, wie das Absuchen der Gegend am besten zu konzipieren ist. Infolgedessen ist dieser Teil des Programms als eine gesonderte Zustandsmaschine im Knoten *Scan Environment* vorgesehen, um einen modularen nachträglichen Austasch von ihm zu ermöglichen. Es hat sich Laufe der Entwicklung herausgestellt, dass sich für den *Scan Environment* die in Abbildung 6.3 zu erkennende Zustandsmaschine am besten eignet. Diese veranlasst, dass Aibo zuerst langsam seinen Kopf um jeweils 90 Grad nach links und rechts bewegt und dann eine Drehung seines Kör-

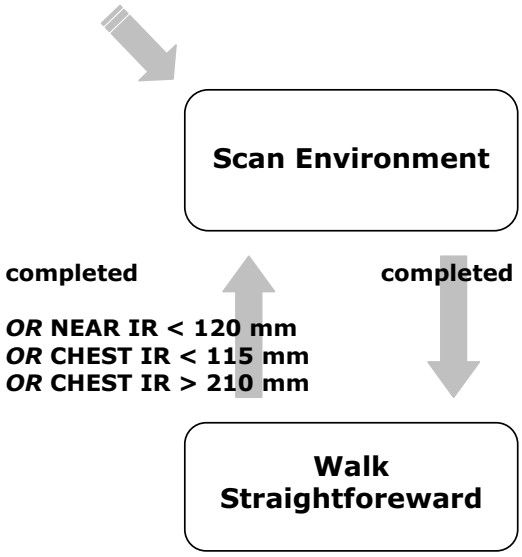



**Abbildung 6.2:** Zustandsdiagramm des Knotens *Search For Landmark*.

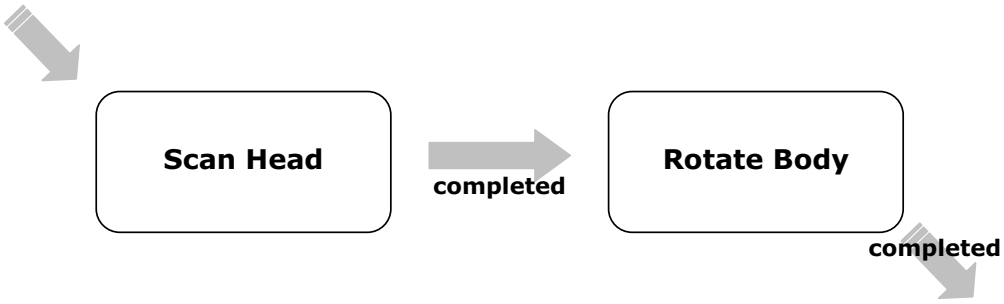



**Abbildung 6.3:** Die unterste Ebene des Automaten, der Zustand *Scan Environment*.

pers um einen zufällig ermittelten Winkel ausführt. Der Betrag des Winkels ist hierbei eine zwischen 15 Grad und 180 Grad gleichverteilte Zufallsgröße. Für die Richtung der Drehung war vorerst eine für positives und negatives Vorzeichen äquivalente Wahrscheinlichkeit von $\frac{1}{2}$ vorgesehen; es hat sich aber für ein zielgerichteteres Suchen der Landmarke als vorteilhafter herausgestellt, eine bestimmte Rotationsrichtung zu bevorzugen und so wurde für eine Drehung im mathematisch positiven Richtungssinn eine Wahrscheinlichkeit von $\frac{3}{4}$ und demzufolge eine Wahrscheinlichkeit von $\frac{1}{4}$ für eine Drehung im Uhrzeigersinn gewählt.

## 6.2 Landmarkenerkennung

Als Landmarken werden DIN-A3-Blätter verwendet, die mit zwei vertikalen Farbstreifen bedruckt und dann an ihrer Ober- und Unterkante mit Büroklammern zu zylinderförmigen Türmchen verbunden worden sind (siehe Abbildung 6.4).

Farbige Landmarken zu verwenden bietet sich an, da Tekkotsus *Vision Pipeline* – auf die das Navigationsmodul zur Erkennung der Landmarken zugreift – mit der an der Carnegie Mellon University entstandenen Bibliothek *CMVision* [BBV00] zur Segmentierung von Farbkamerabildern realisiert wurde. Die Funktionsweise dieser Bibliothek CMVision wird in Abschnitt 6.2.1 erklärt. Im Anschluss daran wird in 6.2.2 beschrieben, wie darauf aufbauend der *LandmarkDetectionGenerator* des Navigationsmoduls die von der Vision Pipeline aus dem Bild gewonnenen Informationen nutzt um festzustellen, ob sich die zu findende Landmarke im Kamerabild befindet oder nicht.

### 6.2.1 Die Bibliothek CMVision

Die Extraktion der Farbinformationen aus dem Kamerbild erfolgt, wie in Abbildung 2.3 des Abschnitts 2.2.3 bereits zu erkennen, in zwei gesonderten Stufen. In der ersten Stufe wird für jedes Pixel des Kamerabilds festgestellt, in welcher der benutzerdefinierten Farbklassen es enthalten ist. Als *Farbklasse* sei hier eine Menge von Punkten eines in den meisten Fällen dreidimensionalen euklidischen Raums, der als *Farbraum* bezeichnet wird, verstanden. Ein Farbraum wird dazu verwendet, um das Frequenzspektrum des Lichts, das von einem bestimmten Punkt im Raum auf die Netzhaut des Auges respektive die Bildfläche der Kamera trifft, sinnvoll parameterisieren und kategorisieren zu können. Als typisches Beispiel für einen solchen Raum sei der an die Farbwahrnehmung des menschlichen Auges angelehnte RGB-Farbraum genannt. Dessen Achsen bezeichnen für eine bestimmte Farbe, welchen Anteil ihr Spektrum an den Frequenzen der Spektralfarben *Rot*, *Grün* und *Blau* hat. In Abbildung 6.5 ist ein Beispiel einer solchen Farbklasse, hier des RGB-Farbraums, als Körper skizziert. Die Bibliothek CMVision speichert eine solche Farbklasse als dreidimensionales binäres Array, das für jeden Farbwert angibt, ob er zu der jeweiligen Farbklasse gehört oder nicht. Es wird dadurch eine sehr schnelle Ausführung des Segmentierungsalgorithmus gewährleistet, weil so für ein Pixel durch einfaches Einsetzen seiner Farbwerte in das Array die Zugehörigkeit zu einer bestimmten Farbklasse analysiert werden kann. Ob die Wahl der Farbklasse erfolgreich war, kann man sich mit Tools des Tekkotsu-Frameworks anzeigen lassen (siehe Abbildung 6.6).

In der zweiten Stufe werden die gefundenen Pixel einer Farbklasse in zusammenhängende Regionen gruppiert. Dazu wird im ersten Durchgang dieser Stufe eine Lauflängenkodierung durchgeführt; das heißt alle horizontal zusammenhängenden Pixelgruppen einer Reihe werden in sogenannten *Läufen (runs)* zusammengefasst. Jeder dieser Läufe soll nun im zweiten Durchgang der Stufe seiner Region zugeordnet werden. Ziel ist es hierbei, dass jeder Lauf eine Referenz auf den am weitesten links oben liegenden Lauf seiner Region innehält. Dieser Lauf ist in CMVision der Repräsentant der Region. Der zweite Durchgang wird so initialisiert, dass jeder Lauf zu Beginn sich selbst als Region referenziert. Im ersten

**Abbildung 6.4:** Die für das Navigationsmodul verwendeten Landmarken. Eine Landmarke enthält eine Auswahl aus den Farben magenta, cyan und gelb.

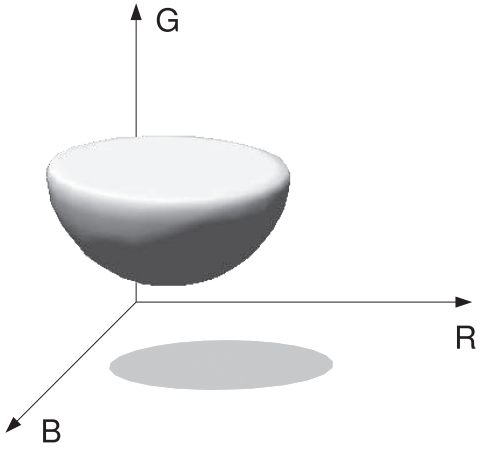
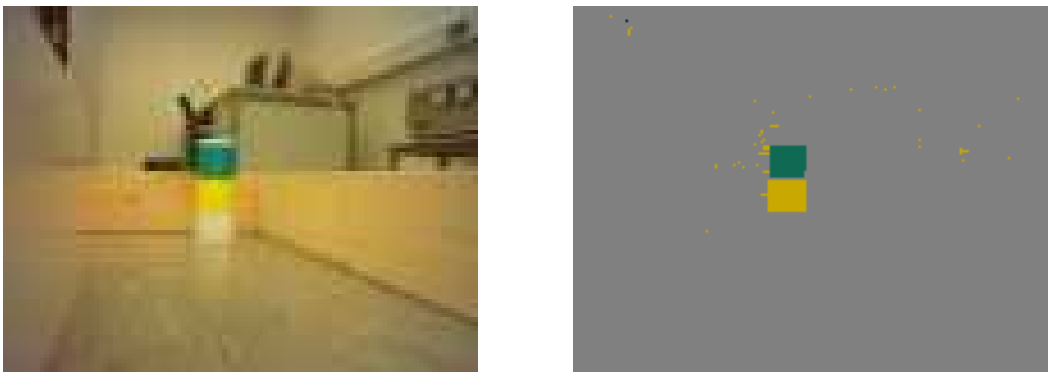



**Abbildung 6.5:** Beispiel eines eine Farbklasse repräsentierenden Körpers im RGB-Farbraum.



(a) Originales Kamerabild des Aibo. (b) Ergebnis der Segmentierung.

**Abbildung 6.6:** Farbsegmentierung eines Kamerabilds.

Schritt dieses Durchgangs werden alle in benachbarten Zeilen liegenden Läufe untersucht; überlappen sich Läufe vertikal, wird der am weitesten links liegende der oberen Läufe als Regionsreferenz der unteren gesetzt. Im zweiten Schritt bildet der Algorithmus die transitive Hülle der nun über die Regionsreferenz vernetzten Läufe; das bedeutet er setzt als neue Regionsrefernz den Lauf, der über eine maximale Anzahl von Zwischenreferenzen erreichbar ist. Ist der Schritt beendet, referenziert jeder Lauf jetzt wirklich den Repräsentanten seiner Farbregion. Während dieser Vorgänge werden nebenbei iterativ verschiedene Parameter der Region – wie etwa der Massenmittelpunkt, die Anzahl der Pixel oder das kleinste die Region umschließende Rechteck (die sogenannte *Bounding Box*) – berechnet.

### 6.2.2 LandmarkDetectionGenerator

Die an Tekkotsus *BallDetectionGenerator* angelehnte Klasse *LandmarkDetectionGenerator* ist ein Behavior, welches auf das periodisch vom *RegionGenerator* der Vision Pipeline generierte Ereignis *SegmentedColorFilterBankEvent* lauscht, die von ihm gelieferten Farbregionsinformationen untersucht und anhand dessen entsprechende *LandmarkVisionObjectEvents* auslöst. Ein Instanz von *LandmarkDetectionGenerator* ist dabei für die Erkennung von genau den zwei Landmarken zuständig, die beide die gleichen zwei Farben enthalten und sich nur dadurch unterscheiden, dass diese beiden Farben untereinander vertauscht sind.

Der erste Schritt des LandmarkDetectionGenerators bei Erhalt des SegmentedColorFilterBankEvent ist, dass er für die zehn größten Regionen seiner beiden Farben einen einer Wahrscheinlichkeit nachempfundenen Konfidenzwert errechnet, der ausdrückt, wie sehr diese Regionen der einfarbigen rechteckigen Hälfte der zu erkennenden Landmarke ähneln. Dazu verwendet er wie Tekkotsus BallDetectionGenerator die an die Dichte der Gaußverteilung angelehnte Funktion

$$f_{d,m}(a,b) = \begin{cases} g_d(a,b) & \text{wenn } g_d(a,b) \geq m \\ 0 & \text{sonst} \end{cases} \quad \text{mit } g_d(a,b) = e^{-\frac{\left(d \cdot \left|\frac{a-b}{a+b}\right|\right)^2}{2}}. \tag{6.1}$$

Diese ab einem Minimumswert $m$ abgeschnittene Funktion $f$ gibt ein normiertes Maß dafür an, wie gering die Differenz ihrer beiden Veränderlichen $a$ und $b$ voneinander ist. Sie wird proportional zu ihrem Parameter $d$ gestreckt. Zur Veranschaulichung ist sie für $m = 0$ und $d = \frac{5}{3}$ in Abbildung 6.7 grafisch dargestellt. Der oben genannte Konfidenzwert $c_{ij}$ der $i$ten Region (mit $1 \leq i \leq 10$) der Farbklasse $j$ (mit $1 \leq j \leq 2$) ergibt sich zu

$$\begin{aligned} c_{ij} &= \begin{cases} t_{ij} & \text{wenn } t_{ij} \leq 1 \\ 1 & \text{sonst} \end{cases} \\ \text{mit } t_{ij} &= \underbrace{f_{d,e^{-3}}(w_{ij} \cdot \tfrac{10}{12}, h_{ij})}_{I} \cdot \underbrace{f_{\frac{5}{3},e^{-3}}(w_{ij} \cdot h_{ij}, A_{ij})}_{II} \cdot \underbrace{\frac{A_{ij}}{1000}}_{III}. \end{aligned} \tag{6.2}$$

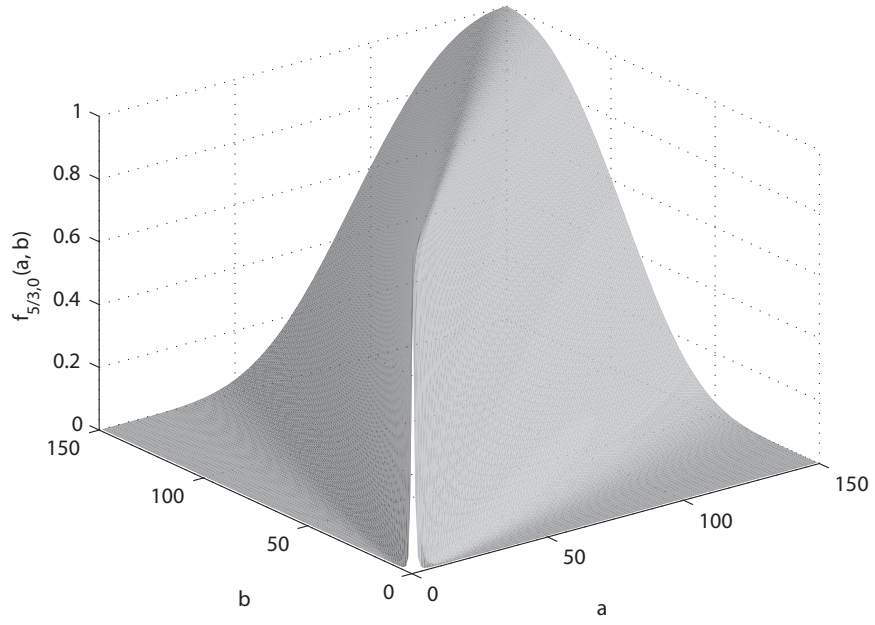



**Abbildung 6.7:** Grafische Darstellung der Funktion $f_{\frac{5}{3},0}(a, b)$ aus Gleichung 6.1.

Teil *I* von Gleichung 6.2 gibt dabei an, wie genau das Seitenverhältnis von der in Pixel gegebenen Höhe $h_{ij}$ und Breite $w_{ij}$ der Bounding Box mit $\frac{10}{12}$, dem Seitenverhältnis des rechteckigen Teils der zu findenden Landmarke, übereinstimmt; Teil *II* bestimmt, wie sehr die rechteckige Bounding Box mit Pixeln der jeweiligen Farbklasse gefüllt ist, also wie gering die Abweichung der ebenfalls in Pixel gegebenen Fläche $A_{ij}$ der Region von $w_{ij} \cdot h_{ij}$ ist; und in Teil *III* erhält der Konfidenzwert der Region abschließend einen Bonus für besonders große Flächen. Der Paramter $d$ ist gleich 5 für Regionen, die den Rand der Bildfläche berühren und gleich $\frac{5}{3}$ für Regionen, die sich ausschließlich im inneren Bereich der Bildfläche befinden. Es werden also die Regionen, die sich am Rand der Bildfläche befinden „weniger streng bewertet“, weil angenommen wird, dass ein Teil von ihnen außerhalb des Bildes ist und so ihr Seitenverhältnis nicht so genau stimmen kann wie von denen, die sich vollständig im Bild befinden.

Es wird dann für die beiden Farben 1 und 2 die Region $r_{k1}$ beziehungsweise $r_{l2}$ (mit $1 \leq k, l \leq 10$) mit dem maximalen Konfidenzwert $c_{k1}$ beziehungsweise $c_{l2}$ bestimmt. Der Konfidenzwert $c$, der aussagt, wie sicher es ist, dass sich eine der zu findenden zwei Landmarken im Bild befindet, errechnet sich dann aus dem Produkt der maximalen Konfidenzwerte der beiden Farben:

$$c = c_{k1} \cdot c_{l2} \tag{6.3}$$

Wenn dieser einen Wert $c_{\text{thresh}} = 0,8$ überschreitet, nimmt der LandmarkDetec-

tionGenerator an, dass sich eine der zwei Landmarken im Kamerabild befindet und löst ein LandmarkVisionObjectEvent am EventRouter aus. Dieses Ereignis enthält eine boolsche Variable *orientation*, die über seine Funktion *getOrientation()* abgefragt werden kann. Sie ist *true*, wenn die als Teil der Landmarke angenommene Region der Farbe 1 über der von Farbe 2 ist, andernfalls ist sie *false*.

## 6.3 Anlaufen der gesichteten Landmarke

Es war ursprünglich vorgesehen gewesen, die Trajektorie zum Anlaufen der Landmarke über einen Kaskadenregelkreis zu bestimmen. Da aber weder die Übertragungsfunktion der Steuerstrecke noch die des Gesamtsystems von Aibos PID-geregelten Servomotoren bekannt sind, musste hier auf eine im theoretischen Sinne unsichere Regelung zurückgegriffen werden. Diese richtet periodisch zuerst Aibos Kopf auf die gefundene Landmarke aus, bestimmt über den Infrarot-Kopfabstandssensor den Punkt der Landmarke relativ zu seinem Körpermittelpunkt und errechnet über diesen dann den zum Ansteuern der Landmarke momentan zu laufenden Geschwindigkeitsvektor. Dass die Stabilität dieses äußeren Kreis nicht im regelungstechnischen Sinne gesichert ist, stellt aber kein Problem dar, da der innere Regelkreis der Motoren auf jegliche Art von Sprung des Sollwerts am Eingang gefeit sein sollte.

Zum Ausrichten des Kopfes wird als erstes aus dem Landmarkenmittelpunkt im Kamerbild der Vektor $\overrightarrow{ray}$ (siehe Abbildung 6.8) berechnet, der ausgehend von Aibos Kamera-Bezugssystem die Richtung angibt, in der sich die Landmarke befindet. Dies geschieht in der im Tekkotsu-Framework implementierten Funktion *config->vision.computeRay(...)*, die sich dazu einfacher Winkel- und Strahlensatzbeziehungen, wie sie beispielsweise in [Bec04] auf Seite 40 beschrieben sind, bedient. Es wurde hier diese Funktion verwendet und nicht die Berechung direkt durchgeführt, weil von Tekkotsus Betreibern für eine spätere Version des Frameworks vorgesehen ist, die in [TT05] beschriebenen Ergebnisse eines Projekts zur Kalibrierung von Aibos Kamera mit in diese Funktion einfließen zu lassen, um so eine sehr viel exaktere Berechnung der Abbildung erreichen zu können. Um nun den auszurichtenden Drehwinkel des Kopfs genau bestimmen zu können, müsste eigentlich auch die Entfernung zur Landmarke und nicht nur ihre Richtung $\overrightarrow{ray}$ bekannt sein. Da dies aber nicht der Fall ist, wird angenommen, dass sie 500 mm vom Nullpunkt des Kamera-Bezugssystems entfernt ist:

$$\overrightarrow{ray}' = \frac{\overrightarrow{ray}}{|\overrightarrow{ray}|} \cdot 500\,\text{mm}. \tag{6.4}$$

Dieser Punkt $\overrightarrow{ray}'$ wird dann über die kinematischen Berechnungsmethoden des im Tekkotsu-Framework eingebundenen Pakets ROBOOP (siehe Abschnitt 5.1)

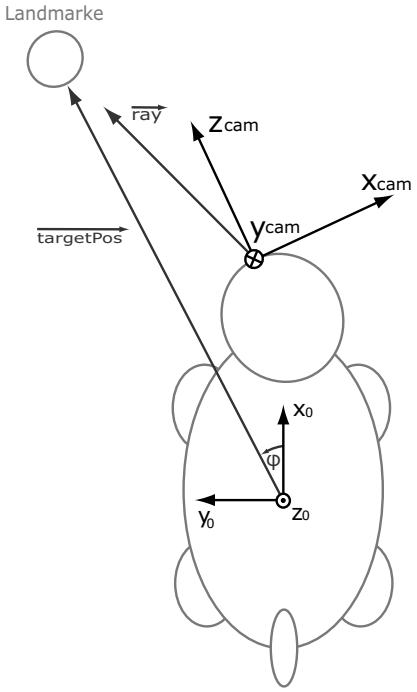



**Abbildung 6.8:** Skizze des Aibos mit seinem Bezugssystem in der Körpermitte und dem Koordinatensystem seiner Kamera.

in das Referenzkoordinatensystem in Aibos Körpermitte (siehe Abbildung 6.8) umgerechnet und dann in die Funktion *lookAtPoint(...)* des MotionCommands *HeadPointerMC* eingesetzt. Diese Methode *lookAtPoint(...)* verwendet die inverse Kinematik von ROBOOP, um den Punkt im Raum, auf den sie den Bildmittelpunkt von Aibos Kamera richten soll, in Stellwinkel seiner Kopfgelenke umrechnen und als neue Sollwerte setzen zu können.

Die momentan zu laufenden Geschwindigkeitssollwerte werden über den Vektor $\overrightarrow{targetPos}$ (siehe Abbildung 6.8) bestimmt, der relativ zu Aibos Körpermitte den Punkt der gesichteten Landmarke im Raum angibt. Bei seiner Berechnung wird angenommen, dass das Ausrichten von Aibos Kopfs bereits vollzogen ist und Aibo senkrecht auf die Landmarke blickt. Der Vektor $\overrightarrow{targetPos}$ kann daher über den im Bezugssystem des Kopfabstandssensor auf die Landmarke zeigenden Vektor

$$\vec{l} = \begin{pmatrix} 0 \\ 0 \\ a \end{pmatrix},$$

dessen dritte Komponente $a$ der momentan vom Infrarotabstandssensor gemessene Wert ist, ermittelt werden. Dies passiert in den im folgenden abgedruckten Zeilen Quelltext, die abermals die in Tekkotsu implementierte Funktionalität zur Kinematikberechnung verwenden:

```
NEWMAT::Matrix TIR;
NEWMAT::ColumnVector targetPos;

if (state->sensors[FarIRDistOffset] < 30)
{
   TIR = kine->jointToBase(NearIRFrameOffset);
   targetPos = TIR *
     Kinematics::pack(0, 0, state->sensors[NearIRDistOffset]);
}
else
{
   TIR = kine->jointToBase(FarIRFrameOffset);
   targetPos = TIR *
     Kinematics::pack(0, 0, state->sensors[FarIRDistOffset]);
}
```

Aus dem eben berechneten Vektor $\overrightarrow{targetPos}$ ergibt sich der in Skizze 6.8 eingezeichnete Winkel $\varphi$ dann zu

$$\varphi = \arctan(\tfrac{t_y}{t_x}) \text{ mit } \overrightarrow{targetPos} = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix}. \tag{6.5}$$

Anhand dieses Winkels $\varphi$ können nun die drei Komponenten $\dot{x_0}$, $\dot{y_0}$ und $\dot{\varphi}$ (korrespondierend zu Abbildung 6.8) des zu laufenden Geschwindigkeitsvektors $\vec{v}$ mit

$$\vec{v} = \begin{pmatrix} \dot{x_0} \\ \dot{y_0} \\ \dot{\varphi} \end{pmatrix}$$

zu

$$\vec{v} = \begin{pmatrix} walkSpeed \cdot \cos\varphi \\ walkSpeed \cdot \sin\varphi \\ bangOn \cdot maxAngularSpeed \cdot \frac{\varphi}{\frac{\pi}{2}} \end{pmatrix} \text{ mit } bangOn \in \{0, 1\} \tag{6.6}$$

festgelegt werden. Der translatorische Bestandteil $\dot{x_0}$ und $\dot{y_0}$ von $\vec{v}$ entspricht einem Vektor der Richtung $\overrightarrow{targetPos}$ und des Betrags *walkSpeed*. Die dritte Komponente, die Rotationsgeschwindigkeit, wird proportional zur Winkelabweichung $\varphi$ bestimmt. Sie wird außerdem zusätzlich über die binäre Variable *bangOn* durch einen Zweipunktregler mit den Schaltschwellen $\varphi_{\text{Low}} = 1 \cdot \frac{\pi}{180}$ und $\varphi_{\text{High}} = 4 \cdot \frac{\pi}{180}$ gesteuert.

Die ermittelten Komponenten des Geschwindigkeitsvektors $\vec{v}$ werden über die Memberfunktion *setTargetVelocity(...)* an den MotionCommand *WalkMC* übergeben. WalkMC ist der in das Tekkotsu-Framework portierte Laufalgorithmus von *CMPack'02* [VLV+02, LBV01], der Quelltextsammlung des Aibo-Teams der Carnegie Mellon University vom RoboCup 2002.

## 6.4 Funktionstest und Schlussfolgerung

Es stellte sich beim Test als ein sehr großes Problem des Navigationsmoduls heraus, dass die momentane Implementierung der Landmarkenerkennung sehr empfindlich auf Schwankungen der Lichtverhältnisse reagiert. Der Grund dafür ist zum ersten, dass Aibos Visionssystem nicht durch ständige automatische Rekalibrierung den aktuellen Lichtverhältnissen angepasst wird und zum zweiten, dass die zu findenden Farbklassen manuell gewählt und nicht durch ein sich ebenfalls anpassendes Lernverfahren selbstständig vom Robotor bestimmt werden. Erst durch ein adaptives Verfahren, wie es beispielsweise in [Jün05] und [JHL04] beschrieben ist, ließe sich es vermeiden, dass vor jeder erneuten Anwendung des Navigationsmoduls nach einem längeren Zeitraum eine Kalibrierung der Kamera von Hand, wie auch eine erneute manuelle Wahl der zu segmentierenden Farbklassen nötig ist.

Weitere Schwierigkeiten bereitete aufgrund ihrer Ineffizienz die zur Vermeidung von Zyklen gewählte Zufallsstrategie zur Navigation von Aibo. Wirklich Abhilfe kann hier nur ein globales Lokalisierungsverfahren schaffen, da erst dadurch eine zielgerichtete Fortbewegung im Raum ermöglicht wird. Als vorteilhaft hat sich aber die Wahl einer Vorzugsrichtung beim zufälligen Drehen des Körpers im Zustand *Rotate Body* erwiesen, weil sie Hin- und Herdrehen auf der Stelle beim Versuch Hindernissen auszuweichen verringert.

Es ist eine grundsätzlich fragwürdige Entscheidung, dass Aibo im Zustand *Track Landmark* die Aufgabe als beendet ansieht, wenn er ein Hindernis mit seinen Sensoren erkennt. Da das Navigationsmodul aber nur ein Demonstrationsprogramm ist und außerdem eine wirklich zufriedenstellende Lösung des Problems ebenfalls nur durch ein globales Lokalisierungsverfahren zu erreichen wäre, wurde hier darauf verzichtet, diesen Makel zu beseitigen.

Als letztes Problem sei abschließend noch genannt, dass beim Anlaufen der Landmarke bei großen Winkelabweichungen die Rotation des Körpers zu schnell erfolgte, um ein Nachregeln der Kopfposition gewährleisten zu können. Dieses ungewünschte Verhalten ließ sich durch Verringerung des Parameters *maxAngularSpeed* unterdrücken. Besonders das Verfahren zum Anlaufen der Landmarke hat sich aber ansonsten als sehr funktionstüchtig und resistent gegenüber Störungen erwiesen.

# Anhang A

# Technische Hinweise

## A.1 Einrichten der W-LAN-Verbindung

Die Wireless-LAN-Verbindung zwischen dem Aibo ERS-7 und dem PC wurde bei dieser Arbeit entweder mit der im *Asus L5000-Notebook* eingebauten 802.11g-W-LAN-Karte oder mit der externen Karte *DWL-G120 Wireless USB* der Firma D-Link im Ad-Hoc-Modus hergestellt. Da es dabei einige technische Schwierigkeiten gab, sollen hier kurz einige Hinweise dazu gegeben werden:

1. Beim Ad-Hoc-Modus kann es sein, dass Windows eine Verbindung des Rechners zu sich selbst auch schon als Verbindung wertet. Außerdem treten hier öfters einige Ungereimtheiten bei der Verbindungs-Anzeige in der Task-Leiste von Windows auf. Als sichere Informationsquelle, ob eine funktionierende Verbindung zum Aibo-Robotor besteht, hat sich die Verwendung von „ping *<IP-Adresse des Aibo>*“ in der Windows-Kommandozeile bewährt.

2. Auf die Verwendung der *SSDP*-Funktion sollte verzichtet werden. Sie dient lediglich dazu, nach Aibos im Drahtlosnetzwerk suchen zu können und führt leicht zum Nichtfunktionieren einer Verbindung.

3. Es ist sehr sinnvoll, die WEP-Verschlüsselung zu aktivieren. Denn dies verhindert, dass Aibo sich mit anderen im Raum vorhandenen W-LAN-Netzwerken verbinden kann. Ist beispielsweise der *Operation Mode* auf „Auto“ gesetzt, so verbindet sich Aibo bevorzugt mit einem Infrastructure-Netzwerk (Netzwerk mit Access Point), welches an vielen Orten im Bereich des Campus der TU Dresden vorhanden ist.

4. Die ESSID sollte in Großbuchstaben gegeben werden. Durch Mischen von Groß- und Kleinschreibung kann es hier zu Problemen kommen.

5. Durch Beobachtung wurde folgende Vermutung aufgestellt: Wenn das Betriebssystem des Aibo bootet, versucht es einmal (oder eine begrenzte An-

zahl von Malen), eine W-LAN-Verbindung zum PC herzustellen. Treten dabei Fehler auf, deaktiviert das System das Wireless LAN. Auf jeden Fall wurde festgestellt, dass ein einfaches Aus- und wieder Einstellen des Aibo zum Herstellen einer funktionierenden Verbindung geführt hat.

6. Es hat sich allgemein als günstig erwiesen, für Hardwareeinheiten immer die neuste Treiberversion aus dem Internet zu verwenden.

7. Eine alte Windows-Weisheit besagt: Auch wenn es keine Erklärung dafür gibt, ein Neustart löst so manches Problem. Zumindest funktionierte schon oft eine nichtfunktionierende W-LAN-Verbindung nach einem Windows-Neustart plötzlich tadellos. Bei Verwendung der *DWL-G120*-Karte wird empfohlen, diese bereits *vor* Einschalten des Computers einzustecken.

Abschließend folgt noch die Angabe einer Konfiguration der *wlanconf.txt*-Datei (bei Verwendung des AIBO MIND 2 Memory Sticks) beziehungsweise der *wlandflt.txt*-Datei (bei Verwendung vom OPEN-R SDK oder des Tekkotsu-Frameworks), die sich im Rahmen dieser Arbeit bewährt hat:

```
#
# WLAN
#
HOSTNAME=OKAMI
ESSID=AIBONET
WEPENABLE=1
WEPKEY=asdfg
APMODE=2

#
# IP network
#
#USE_DHCP=1
#
# If DHCP is not used (USE_DHCP=0), you need to specify IP
# network configuration.
#
USE_DHCP=0
ETHER_IP=192.168.0.10
ETHER_NETMASK=255.255.255.0
IP_GATEWAY=192.168.0.1
DNS_SERVER_1=192.168.0.1

#
# SSDP
#
SSDP_ENABLE=0
```

## A.2 Installation des Aibo Remote Frameworks

**Hinweis:** Bei dem in dieser Arbeit beschriebenen ARF handelt es sich um die Version „Final R1“ (`AIBORemoteFramework_r1.zip` von `http://openr.aibo.com/openr/eng/index.php4` (Stand: 04.05.2005))

Die Installation des ersten Teils des ARFs geschieht automatisch durch Installieren des Programms „Aibo Entertainment Player“. Der zweite Teil, die Aibo Remote Framework API, können von `http://openr.aibo.com/openr/eng/index.php4` (Stand 04.05.2005) als `AIBORemoteFramework_r1.zip` heruntergeladen werden. Die Bibliotheken müssen nicht installiert werden. Es genügt, sie zu entpacken und in das entsprechende Visual-C++-Projekt einzubinden. Da das Einbinden der Aibo Remote Framework API unangenehm ist, empfiehlt es sich, außerdem die von Sony erhältlichen Beispielprogramme `RFW_Sample_src_r1.zip` (Stand 04.05.2005) herunterzuladen und in ein Unterverzeichnis (beispielsweise „Samples“) des zuvor entpackten Verzeichnis „Aibo Remote Framework“ zu entpacken. Diese Beispielprogramme können dann nach Belieben geändert werden. Im Rahmen dieser Arbeit wurde das Beispielprogramm „BaseClient“ bis auf das Nötigste reduziert und in dem Verzeichnis „Ausgangsprogramm“ der dieser Arbeit zugehörigen CD-ROM abgelegt. Es dient als guter Ausgangspunkt für die Entwicklung eines eigenen ARF-Programms.

## A.3 Installation des Tekkotsu-Frameworks

**Hinweis:** Bei dem in dieser Arbeit beschriebenen Tekkotsu-Framework handelt es sich um die Version 2.3 und bei OPEN-R SDK um die Version 1.1.5 R3 (`OPEN_R_SDK-1.1.5-r3.tar.gz`).

Es wird hier nun kurz die Installation des Tekkotsu-Frameworks unter Windows XP beschrieben.

### A.3.1 Cygwin

*Cygwin* ist ein Emulator einer Linux/UNIX-Kommandozeile für Windows. Dieser wird benötigt, da OPEN-R, das Tekkotsu-Framework und die zugehörigen Skripte auf Linux ausgelegt sind (beispielsweise wird ein Aperios-Backend für den GNU-C++-Compiler verwendet). Am günstigsten installiert man vorerst die auf der OPEN-SDE-Website verfügbare Version von Cygwin. Dazu lädt man sich die Datei `cygwin-packages-1.5.5-bin.exe` von `http://openr.aibo.com/openr/eng/index.php4` (Stand 05.04.2005) runter und extrahiert sie (Selbstextrahierendes Archiv, entpacken durch Ausführen). Dann führt man die Datei *setup.exe*

des entstandenen Verzeichnisses aus und wählt „Install from Local Directory“. Nach Wahl des Installationsverzeichnisses (`C:\Programme\Cygwin`) und Wahl des *Default Text File Type* auf Unix, soll das *Local Package Directory* angegeben werden. Hierfür wähle man das entpackte Verzeichnis (sollte automatisch eingestellt sein). Dann fährt man mit der Installation fort.

Es wurden nun die von Sony mitgelieferten Packages von Cygwin installiert. Diese sind allerdings nur für OPEN-R alleine ausreichend. Für das Tekkotsu-Framework müssen weitere Packages installiert werden. Dazu lädt man sich die Datei *setup.exe* von der Cygwin-Website herunter (`http://www.cygwin.de`, Stand 04.05.2005) und fährt ähnlich wie mit der von Sony downgeloadeten Version fort. Man wähle hier jedoch „Install from Internet“. Folgende Packete müssen auf jeden Fall zusätzlich noch installiert werden:

- Alle Packete der Kategorie *Devel*
- GNU *make*
- GNU *binutils*
- *rsync* der Kategorie *Net*
- *more* der Kategorie *Text*

Eine aktuelle Liste der über OPEN-R hinaus benötigen Packete befindet sich auf der Tekkotsu-Hompage (`http://www-2.cs.cmu.edu/~tekkotsu/`, Stand 05.05.2005). Nun hat man eine Linux-Shell unter Windows. Die Linux-Verzeichnisstruktur ist unter dem Cygwin-Verzeichnis nachgebildet. Hinweise für Linux-Anfänger: Eine Liste der im aktuellen Verzeichnis vorhandenen Dateien und Ordner erhält man durch den Befehl *ls* und der Verzeichniswechsel geschieht ähnlich wie in DOS mit *cd* (statt *cd..* jedoch *cd ..*).

### A.3.2 OPEN-R SDK

Für die Installation des OPEN-R SDK lade man sich von der Sony-SDE-Website die Dateien `OPEN_R_SDK-1.1.5-r3.tar.gz` und `mipsel-devtools-3.3.2-bin-r1.tar.gz` herunter und entpacke das in beiden der Dateien enthaltene Verzeichnis `OPEN_R_SDK` in das Unterverzeichnis `/usr/local/OPEN_R_SDK` der Cygwin-Installation (z.B. `C:\Programme\Cygwin\usr\local\OPEN_R_SDK`).

### A.3.3 Java SDK

Um die PC-seitigen Tools des Tekkotsu-Frameworks nutzen zu können, benötigt man Java. Für die Java-Installation sei auf [Mül04] verwiesen. Bemerkung: Um

das dreidimensionale Modell des Aibos zu verwenden muss zusätzlich zur Java-Standard-Installation noch *java3D* (`http://java.sun.com/products/java-media/3D/`, Stand 06.05.2005) installiert sein.

### A.3.4 Tekkotsu-Framework

Auch die Installation des Tekkotsu-Frameworks selbst ist in [Mül04] sehr gut beschrieben. Allerdings ein paar zusätzliche Hinweise:

1. Es wird empfohlen, das Tekkotsu-Framework nicht nach `C:\`, sondern direkt in das cygwin-Verzeichnis zu entpacken (`C:\Programme\cygwin\usr\local\Tekkotsu`).

2. Anstelle des Setzens des Links auf `/mnt/memstick` wird hier empfohlen, den Laufwerksbuchstaben des Memory-Sticks direkt in die `Environment.conf`-Datei des jeweiligen Projektes einzutragen.

3. Da die Dateien im Cygwin-Verzeichnis nicht im unter Windows üblichen ANSI-Format kodiert sind, ist der Standard-Windows-Editor *notepad* zum editieren dieser Dateien ungeeignet. Man nehme einen komplexeren Editor wie zum Beispiel den von MATLAB oder den des Visual Studios.

4. Entgegen der Behauptung von Martin Müller in seiner Diplomarbeit [Mül04] lief der FTP-Server TinyFTPD des Tekkotsu-Frameworks sehr stabil. Seine Verwendung wird empfohlen, da sie sehr viel Zeit spart. Im Tekkotsu-Verzeichnis *tools* wurde dafür von den Entwicklern des Frameworks ein Skript *ftpupdate* angelegt, das den Memorystick des Aibo via FTP aktualisiert und den Aibo neu bootet. Um den Entwicklungsprozess zu beschleunigen, wurde das *Makefile* aus dem *project*-Verzeichnis um folgende Zeilen erweitert:

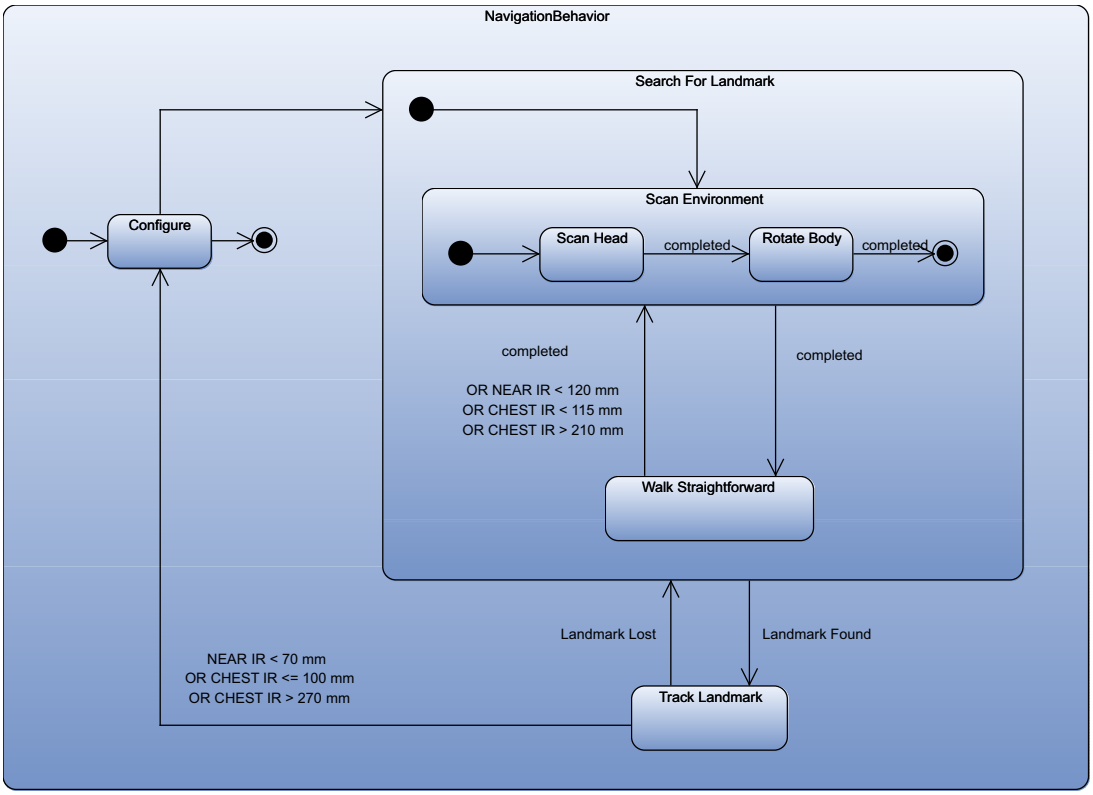
   ```
   ftpupdate: compile $(TEKKOTSU_ROOT)/tools/evenmodtime/evenmodtime
           @echo "Updating the memorystick via ftp using ftpupdate
                                   with the IP address $(IPADDRESS)"
           @$(TEKKOTSU_ROOT)/tools/ftpupdate $(IPADDRESS) ms;
   ```

   Dadurch wird durch Eingabe von *make ftpupdate IPADDRESS=*<IP-Adresse des Aibo> im aktuellen Projektverzeichnis das Projekt neu kompiliert und via FTP auf den Aibo übertragen. Das geänderte Makefile liegt der CD-Rom dieser Arbeit bei.

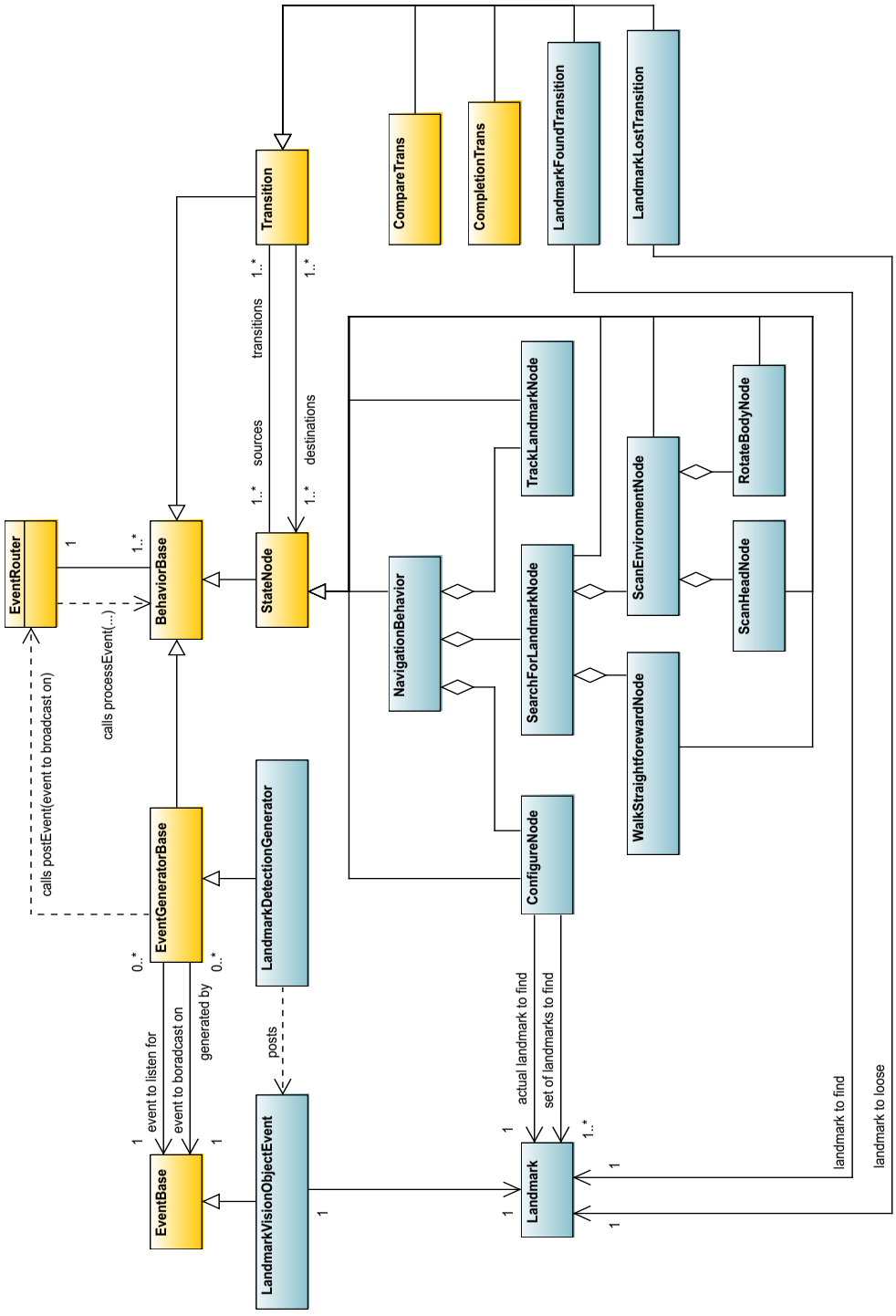
# Anhang B

# Software-Dokumentation

## B.1 UML-Zustandsdiagramm des Navigationsmoduls

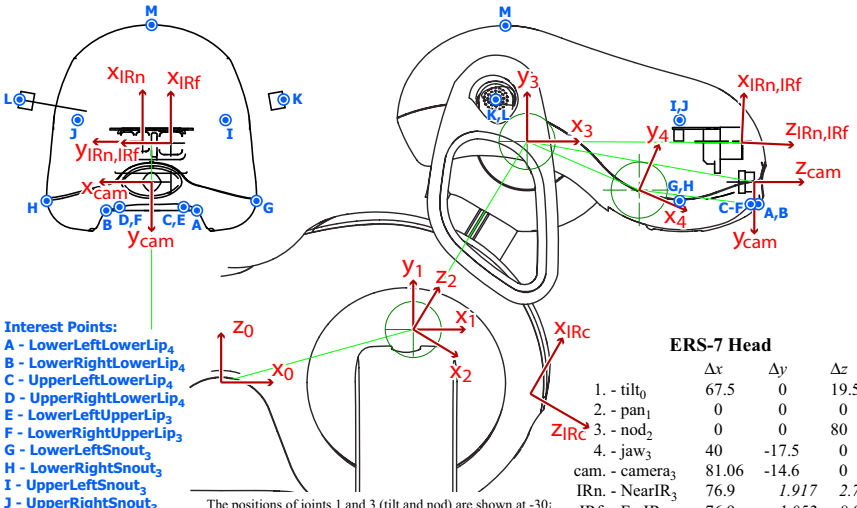


## B.2 UML-Klassendiagramm des Navigationsmoduls



# Anhang C

# Diagramme für das Paket ROBOOP




The positions of joints 1 and 3 (tilt and nod) are shown at $-30_i$ and $30_i$ respectively. When both are at $0_i$, the neck is vertical, and the head is in the same orientation (level horizontally)

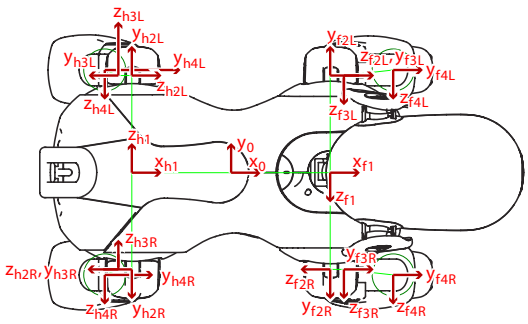
The near and far IR beam has been measured from calibration on an actual ERS-7, chest IR is only from diagrams

**ERS-7 Head**

| | $\Delta x$ | $\Delta y$ | $\Delta z$ |
|---|---|---|---|
| 1. - $tilt_0$ | 67.5 | 0 | 19.5 |
| 2. - $pan_1$ | 0 | 0 | 0 |
| 3. - $nod_2$ | 0 | 0 | 80 |
| 4. - $jaw_3$ | 40 | -17.5 | 0 |
| cam. - $camera_3$ | 81.06 | -14.6 | 0 |
| IRn. - $NearIR_3$ | 76.9 | *1.917* | *2.795* |
| IRf. - $FarIR_3$ | 76.9 | *1.052* | *-8.047* |
| IRc. - $ChestIR_0$ | 109.136 | -3.384 | 0 |

$x_3 \angle x_4 = -23.6294_i$

$x_3 \angle E,F = -7.3301_i$

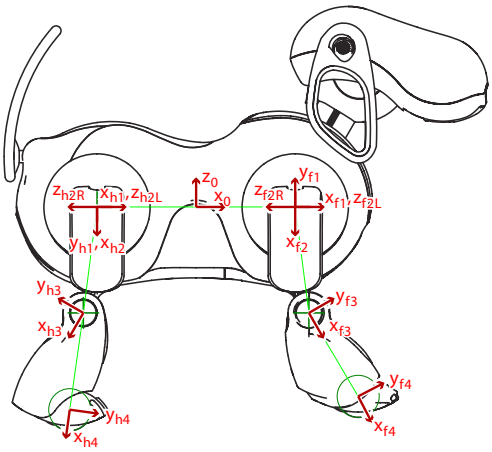


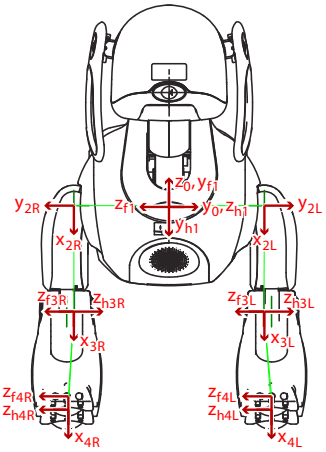

**ERS-7 Legs**

| | $\Delta x$ | $\Delta y$ | $\Delta z$ |
|---|---|---|---|
| 1. - shoulder | 65 | 0 | 0 |
| 2. - elevator | 0 | 0 | 62.5 |
| 3. - knee | 69.5 | 0 | 9 |
| f4. - ball | *69.987* | *-4.993* | 4.7 |
| h4. - ball | *67.681* | *-18.503* | 4.7 |

Diameter of ball of foot is *23.433*mm
Each link offset is relative to previous link

The shins shown in this diagram appear to be slightly distorted compared to a real robot. Corresponding measurements have been taken from actual models.



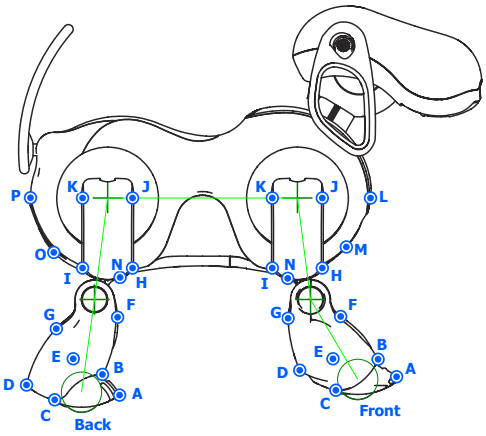

**Interest Points:**
**A - Toe{L,R}{Fr,Bk}Paw$_4$**
**B - Lower{Inner,Outer}Front{L,R}{Fr,Bk}Shin$_3$**
**C - Lower{Inner,Outer}Middle{L,R}{Fr,Bk}Shin$_3$**
**D - Lower{Inner,Outer}Back{L,R}{Fr,Bk}Shin$_3$**
**E - Middle{Inner,Outer}Middle{L,R}{Fr,Bk}Shin$_3$**
**F - Upper{Inner,Outer}Front{L,R}{Fr,Bk}Shin$_3$**
**G - Upper{Inner,Outer}Back{L,R}{Fr,Bk}Shin$_3$**
**H - Lower{Inner,Outer}Front{L,R}{Fr,Bk}Thigh$_2$**
**I - Lower{Inner,Outer}Back{L,R}{Fr,Bk}Thigh$_2$**
**J - Upper{Inner,Outer}Front{L,R}{Fr,Bk}Thigh$_2$**
**K - Upper{Inner,Outer}Back{L,R}{Fr,Bk}Thigh$_2$**
**L - Upper{L,R}Chest$_0$**
**M - Lower{L,R}Chest$_0$**
**N - {L,R}{Fr,Bk}Belly$_0$**
**O - Lower{L,R}Rump$_0$**
**P - Upper{L,R}Rump$_0$**

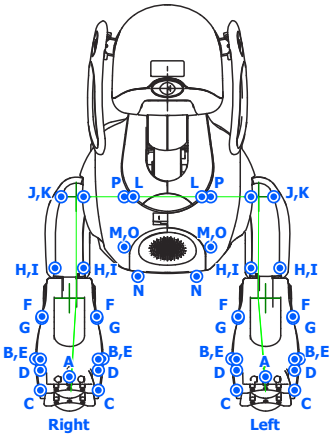
**ERS-7 Legs**

| | *Æx* | *Æy* | *Æz* |
|---|---|---|---|
| 1. - shoulder | 65 | 0 | 0 |
| 2. - elevator | 0 | 0 | 62.5 |
| 3. - knee | 69.5 | 0 | 9 |
| f4. - ball | *69.987* | *-4.993* | 4.7 |
| h4. - ball | *67.681* | *-18.503* | 4.7 |

Diameter of ball of foot is *23.433*mm
Each link offset is relative to previous link

The shins shown in this diagram appear to be slightly distorted compared to a real robot. Corresponding measurements have been taken from actual models.



# Literaturverzeichnis


[BBV00] Bruce, J.; Balch, T. ; Veloso, M.: Fast and Inexpensive Color Image Segmentation for Interactive Robots. In: *Proceedings of the 2000 IEEE/RSJ International Conference on intelligent Robots and Systems* 3 (2000), Oktober, S. 2061–2066

[Bec04] Beck, M.: *Ein Navigationsverfahren für mobile Roboter unter Nutzung des optischen Flusses.* Fakultät Elektrotechnik und Informationstechnik, Technische Universität Dresden, Diplomarbeit, 2004

[BKMY05] Blank, D.; Kumar, D.; Meeden, L. ; Yanco, H.: The Pyro toolkit for AI and robotics. In: *derzeit (20.07.2005) noch nicht veröffentlicht, eingereicht bei dem Journal:* AI Magzine (2005)

[CFJ+04] Cedheim, E.; Ferchichi, R.; Jonsson, A.; Lind, D.; Nyman, H.; Sivertsson, O.; Widenfalk, A.; Åkerlund, J.; Mokrushin, L. ; Pettersson, P.: Kelb - A Real-Time Programming Environment for the Sony Aibo / Uppsala University. Department of Information Technology, Oktober 2004 ( 2004-044). – Forschungsbericht. `http://www.it.uu.se/research/reports/2004-044/`

[DeM79] DeMarco, T.: *Structured Analysis and System Specification.* 1. Auflage. Prentice Hall, 1979

[FK97] Fujita, M.; Kageyama, K.: An Open Architecture for Robot Entertainment. In: *Proceedings of the First International Conference on Autonomous Agents* (1997), S. 435–442

[Gou97] Gourdeau, R.: Object-Oriented Programming for Robotic Manipulator Simulation. In: *IEEE Robotics & Automation Magazine* (1997), September, S. 21–29

[Hol04] Holinski, M.: *Grundlagen der Programmierung von autonomen Systemen am Beispiel des Sony Roboters AIBO ERS-7 unter Verwendung des OPEN-R SDK.* Fachbereich Mathematik, Naturwissenschaften und Informatik, Fachhochschule Gießen-Friedberg, Diplomarbeit, 2004

[JHL04] Jüngel, M.; Hoffmann, J. ; Lötzsch, M.: A Real-Time Auto-Adjusting Vision System for Robotic Soccer. In: *7th International Workshop on RoboCup 2003* (2004)

[Jün05] Jüngel, M.: Using Layered Color Precision for a Self-Calibrating Vision System. In: *8th International Workshop on RoboCup 2004* (2005)

[Kre05] Kreutzer, M.: *Entwicklung eines Stereokamerasystems und eines Fernsteuersystems für den Sony Roboter AIBO ERS-7*. Fachbereich Mathematik, Naturwissenschaften und Informatik, Fachhochschule Gießen-Friedberg, Diplomarbeit, 2005

[LBV01] Lenser, S.; Bruce, J. ; Veloso, M.: CMPack: A Complete Software System for Autonomous Legged Soccer Robots. In: *Proceedings of the fifth international conference on Autonomous agents* (2001), S. 204–211

[Mül04] Müller, M.: *Einführung in die Programmierung autonomer Systeme am Beispiel des Sony Roboters AIBO ERS-7 unter Verwendung des Tekkotsu Frameworks*. Fachbereich Mathematik, Naturwissenschaften und Informatik, Fachhochschule Gießen-Friedberg, Diplomarbeit, 2004

[RGCCM04] Rico, F. M.; González-Careaga, R.; Cañas, J. M. ; Matellán, V.: Programming Model Based on Concurrent Objects for the AIBO Robot. In: *Actas de las XII Jornadas de Concurrencia y Sistemas Distribuídos* (2004), Juni, S. 367–379

[SB03] Serra, F.; Baillie, J.-C.: *OPEN-R SDK Tutorial*. Version 1.0. ENSTA, Juni 2003. – von `http://uei.ensta.fr/baillie/eng/openr_tutorial.html` (Stand: 04.04.2005)

[Sch97] Schairer, A.: *Verfahren zur aufgabenbezogenen Auslegung von Roboterkinematiken*. Institut für Industrielle Fertigung und Fabrikbetrieb, Universität Stuttgart, Studienarbeit, 1997

[Son04a] Sony: *AIBO Motion Editor. User's Manual and Reference*. Version 115-01. 2004. – aus `MEdit_Manual_E_Ver1.zip` von `http://openr.aibo.com/openr/eng/index.php4` (Stand: 12.03.2005)

[Son04b] Sony: *AIBO Remote Framework (for ERS-7) Specification*. Version vom 21.10.2004. 2004. – aus `AIBORemoteFramework_r1.zip` von `http://openr.aibo.com/openr/eng/index.php4` (Stand: 04.04.2005)

[Son04c] Sony: *OPEN-R SDK. Level2 Reference Guide*. Version 115-01. 2004. – aus `OPEN_R_SDK-docE-1.1.5-r1` von `http://openr.aibo.com/openr/eng/index.php4` (Stand: 12.03.2005)

[Son04d] Sony: *OPEN-R SDK. Model Information for ERS-7*. Version 115-01. 2004. – aus `OPEN_R_SDK-docE-1.1.5-r1` von `http://openr.aibo.com/openr/eng/index.php4` (Stand: 12.03.2005)

[Son04e] Sony: *OPEN-R SDK. Programmer's Guide*. Version 115-01. 2004. – aus `OPEN_R_SDK-docE-1.1.5-r1` von `http://openr.aibo.com/openr/eng/index.php4` (Stand: 12.03.2005)

[Tél04a] Téllez, R. A.: *OPEN-R Essentials*. Version 1.0. Mediterranean University, September 2004. – von `http://www.ouroboros.org/aibo.html` (Stand: 01.04.2005)

[Tél04b] Téllez, R. A.: *R-Code SDK Tutorial*. Version 1.2. Mediterranean University, September 2004. – von `http://www.ouroboros.org/aibo.html` (Stand: 01.04.2005)

[TT05] Tira-Thompson, E.: 16-721 Project / Carnegie Mellon University. 2005. – Forschungsbericht. `http://ejt.no-ip.org/stuff/16-721/` (Stand: 04.08.2005)

[TTT05] Touretzky, D. S.; Tira-Thompson, E. J.: *Exploring Tekkotsu Programming on the Sony AIBO*. Draft version (incomplete) as of January 14, 2005, Januar 2005. – `http://www-2.cs.cmu.edu/%7Edst/Tekkotsu/Tutorial/` (Stand: 19.07.2005)

[VLV+02] Veloso, M.; Lenser, S.; Vail, D.; Roth, M.; Stroupe, A. ; Chernova, S.: *CMPack-02: CMU's Legged Robot Soccer Team*. RoboCup 2002 report: Carnegie Mellon University, 2002. – `http://www.cs.cmu.edu/~coral-downloads/legged/papers/cmpack_2002_teamdesc.pdf` (Stand: 13.08.2005)

[Voi04] Voigt, M.: *Konzepte hinter Microsofts .NET*. Fakultät Informatik, Technische Universität Dresden, Seminararbeit, 2004. – `http://os.inf.tu-dresden.de/EZAG/old/ws2004/abstracts/abstract_20050107.xml` (Stand: 07.01.2005)

[You89] Yourdon, E.: *Modern Structured Analysis*. 13. Auflage. Yourdon Press, 1989

## Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die von mir am heutigen Tag eingereichte Studienarbeit zum Thema *Inbetriebnahme sowie Untersuchung einer kommerziellen Roboterplattform* vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Dresden, den

.....