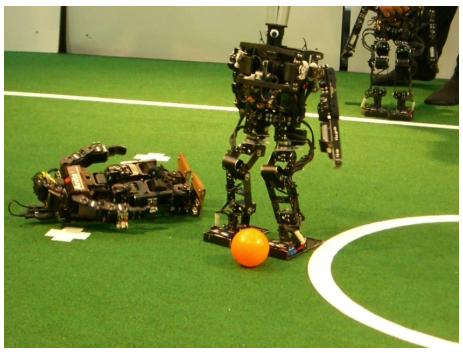
UNIVERSITÄT DORTMUND

ENDBERICHT DER PROJEKTGRUPPE 521

Zweibeinige Fußball spielende Roboter
Adaptive dynamische Bewegungssteuerung und kooperative Weltmodellierung



Institut für Roboterforschung

PG Teilnehmer:
Christian Büttner, Björn Grothe, Yang Huaiyoung, Thorsten Humberg,
Markus Küch, Rayan Naal, Tobias Schade, Markus Schmeing,
Christian Schoppmeyer, Marc Spohr, Oliver Urbann, Florian Wilmshöver

Betreuer:
Prof. Dr.-Ing. Uwe Schwiegelshohn
Stefan Czarnetzki
Sören Kerner

Date:
28. Oktober 2008

Inhaltsverzeichnis

1 Einleitung

Im Folgenden wird die Arbeit und die Ergebnisse der Projektgruppe 521 an der Technischen Universität Dortmund dargestellt. Ziel der Projektgruppe war die Programmierung eines zweibeinigen Roboters um am RoboCup 2008[1] in Suzhou teilzunehmen. Dieser Wettbewerb wird von der RoboCup Federation[2] organisiert, mit dem Ziel die Entwicklung in den Bereichen Künstliche Intelligenz und Robotik voran zu treiben. Bis zum Jahr 2050 soll ein Team vollkommen autonom agierender Roboter in der Lage sein, gegen den menschlichen Fußball-Weltmeister anzutreten.
Seit diesem Jahr wird der von Aldebaran Robotics[3] entwickelte Roboter *Nao* als neuer Hardware-Standard eingesetzt. In der entsprechenden Liga steht also die Software-Entwicklung, Künstliche Intelligenz und Verhalten im Vordergrund. Alle Teams verwenden dasselbe Robotermodell, dessen Hardware in keiner Weise modifiziert werden darf.
Einige Teile der Software wurden in früheren Projektgruppen entwickelt und teilweise unverändert, teilweise in modifizierter Form übernommen. Dazu gehören unter anderem Framework, Simulator und das Debug- und Remote Control Tool RoboControl XP. Da die ersten Roboter erst im April 2008 ausgeliefert wurden, musste die ersten Monate im Simulator entwickelt werden. Hardware-Entwicklung war hauptsächlich vonnöten um die Korrektheit und Genauigkeit von Sensordaten zu bestimmen.
Andere Teile der Software, wie zum Beispiel Module für die Hardware-Ansteuerung, mussten an das neue Robotermodell angepasst werden und wurden daher völlig neu entworfen. Eine zentrale neue Klasse stellt das Egomodell dar, welches alle nötigen Informationen über den Zustand des Roboters enthält und als Schnittstelle zwischen den einzelnen Modulen fungiert. Um bei dem Wettbewerb konkurrenzfähig zu sein, war vor allem die Entwicklung einer neuen Walking Engine notwendig, die dynamisch auf Sensordaten reagieren kann.

1 http://www.robocup-cn.org/

2 http://www.robocup.org/

3 http://www.aldebaran-robotics.com/

2 Sensorik

2.1 Sensoren im realen Roboter

2.1.1 Beschleunigungssensor

Beschleunigungssensor im Nao

Im Nao wurde ein Beschleunigungssensor Typ LIS3LV02DQ der Firma ST Microelectronics verbaut. Der Beschleungigungssensor ist Teil der Inertialen Einheit, die neben dem Beschleunigungssensor noch aus einem Zwei-Achsen-Gyroskop besteht. Er ist in der Lage Beschleunigungen auf einer linearen Skala von entweder $\pm 2G$ oder $\pm 6G$ mit einer Bandbreite von $640Hz$ und $12Bit$ Genauigkeit in drei Achsen zu messen. Messwerte werden über einen I2C-Bus ausgegeben. Der Sensor ist so im Roboter angebracht, dass die Achsen des Beschleunigungssensors ein rechtshändiges, kartesisches Koordinatensystem bilden. Die positive x-Richtung zeigt dabei nach vorne, die positive y-Richtung nach links und die positive z-Richtung nach oben.

Rausch- und Driftverhalten

Die Präzision des Sensors wird vom Hersteller mit 2% Abweichung in x-y-Richtung und 5% in z-Richtung für den $\pm 2G$ Bereich sowie 1% in x-y-Richtung und 2% in z-Richtung für den $\pm 6G$ Bereich des Sensors angegeben. Im Nao wird der Sensor im $2G$-Bereich betrieben. Während der ersten Messungen und Tests bewegten sich die gemessenen Werte im spezifizierten Bereich. Das rohe Signal wies ein weisses Rauschen im Bereich von ca. 1% - 2% auf. Im Gegensatz zur x- und y-Ebene rauschte die z-Richtung im Schnitt sogar mit mehr als 5%. Gemessene Werte in z-Richtung schwanken in der Spitze sogar bis zu $0,1G$. Dies ist eine Eigenheit des Sensors und ist auch nicht durch die durchgängig wirkende Erdbeschleunigung zu erklären, da, sobald der Roboter nicht mehr aufrecht steht, also auf der Seite liegt, den Oberkörper gebeugt hat oder umgefallen ist, die entsprechende Beschleunigung, in die die z-Achse des Roboters gerade zeigt, stärker verrauscht ist.

Kalibrierung

Die Rohdaten, die der Sensor ausgibt, repräsentieren noch keine physikalische Einheit, wie etwa G oder m/s^2. Die Werte bewegen sich zwischen -120 und 120 auf einer einheitenlosen Skala.
Unsere Messungen haben gezeigt, dass die z-Richtung bei geradem Stand des Roboters auf ebener Unterlage einen Wert zwischen 52 und 60 auf der numerischen Skala annimmt. Dies ist unser erster Ansatzpunkt für die Kalibrierung des Sensors gewesen. Unser Ziel

war es, die Beschleunigungswerte in G auszugeben. Um dies zu erreichen muss der Roboter zunächst aufrecht auf einer waagerechten Unterlage stehen. Der Oberkörper muss dabei soweit wie möglich senkrecht zum Untergrund stehen. Wenn der Roboter sich in dieser Position befindet, sollten die Rohwerte der Beschleunigung in z-Richtung ungefähr im oben genannten Bereich zwischen -53 und -60 auf der numerischen Skala liegen. Nun nimmt man sich den Mittelwert dieser Werte über einen angemessenen Zeitraum und trägt den Betrag des Kehrwerts des Mittelwerts als Gain-Parameter für den Beschleunigungssensor in die entsprechende Instanz des Frameworks ein. Mit dieser Maßnahme wird die numerische Skala auf G, also die Erdbeschleunigung, normiert. Die Beschleunigungen liegen damit in einer physikalischen Einheit vor und können weiter verwendet werden.
Erwähnt sei an dieser Stelle, dass bei den ersten Kalibrierungsversuchen auch Offsets für den Sensor eingestellt wurden. Wie aber festgestellt wurde, verfälschen Offsets die Messwerte des Sensors eher. Die Messwerte verhalten sich aufgrund der Temperaturabhängigkeit des Sensors und teilweise auch durch Verschleiß des Roboters nicht hundertprozentig linear, sodass ein Offset die Messwerte nach gewisser Betriebszeit des Roboters verfälscht. Je nach Wärmeentwicklung sollte der Gain-Wert der Sensorkalibrierung hierfür angepasst werden, indem er, wie oben beschrieben, neu kalibriert wird. Damit wird die leichte Nichtlinearität des Sensors besser angenähert.

Aussagekraft in verschiedenen Situationen

Sind die Sensoren einmal kalibriert lassen sich aus diesen Werten bereits erste Aussagen über den Zustand des Roboters treffen. Anhand der Beschleunigungswerte kann man bei vorheriger korrekter Kalibrierung des Sensors beispielsweise feststellen, ob der Roboter aufrecht steht. Im Wesentlichen sollte das dieselbe Situation darstellen, die auch für die Kalibrierung vorausgesetzt wird. Darauf aufbauend lässt sich auch auf die Neigung des Oberkörpers schließen. Die Positionsänderung muss allerdings eine Beschleunigsänderung bewirken, welche größer als das Sensorrauschen ist. Die ersten Tests haben gezeigt, dass der Roboter mit diesen ungefilterten Werten erkennen kann, ob er hingefallen ist bzw. auf dem Boden liegt oder nur schwankt aber noch steht.
Als Vorgriff sei hier erwähnt, dass auch die Walking Engine mit ungefilterten Werten bereits gute Ergebnisse beim Balancieren und Ausgleichen von unvorhergesehen Störungen lieferte.

Filterung

Es wurde ein erster Versuch unternommen, die Qualität der Rohdaten zu verbessern. Der naheliegendste Ansatz war zunächst eine einfache Tiefpassfilterung. Hierfür wurde eine vorgefertigte Tiefpassimplementierung angepasst, deren Parameter bereits ausgemessen und verifiziert wurden.
Bei dieser Filterimplementierung handelt es sich um eine Lösung, die mit Fixpunktarithmetik rechnet. Da unsere Werte aber als Fließkommazahlen vorliegen, muss für die Filterung eine Konvertierung vorgenommen werden.

k	Bandbreite (normalisiert auf $1kHz$)	Anstiegszeit (Samples)
1	0.1197	3
2	0.0466	8
3	0.0217	16
4	0.0104	34
5	0.0051	69
6	0.0026	140
7	0.0012	280
8	0.0007	561

Tabelle 2.1: Tiefpassfilter - Bandbreite, Anstiegszeit in Abhängigkeit vom Parameter k

Hierbei stellte es sich als sinnvoll heraus lediglich 3 Nachkommastellen des Rohsignals zu berücksichtigen. Alle weiteren Stellen werden durch Rauschen zu sehr beeinflusst, als sie brauchbare Informationen liefern könnten.
Während der Tests des Filters wurden mehrere Testreihen des Sensor mit unterschiedlichen Parametern K der Tabelle aufgenommen. Eine Testreihe sollte im wesentlichen alle Situationen enthalten die während eines Fussballspiels auftreten können und unterschiedliche Wertekombinationen und -änderungen des Beschleunigungssensors aufweisen. Hierzu wurden Werte im aufrechten Stand aufgenommen, Werte für die der Oberkörper über eine möglichst einheitliche Zeitspanne bis zur maximalen Auslenkung in positiver und negativer x- und y-Richtung des Roboterkoordinatensystems bewegt wurde, sowie Positionen in denen der Roboter auf dem Boden liegt. Diese Situationen ergeben nun unterschiedliche Verläufe der Beschleunigungskurven. Die Umfallsituation soll hier einen relativ harten Anstieg der Flanke aufweisen und dann möglichst konstant auf dem Wert der Erdbeschleunigung liegen je nachdem wie der Roboter gerade liegt. Die Auslenkungen des Oberkörpers sollen weichere Anstiege aufweisen, ähnlich denen wie sie bei einem stabilen Laufen auftreten würden.
Der Parameter K des Filters bestimmt nun die Bandbreite des Filters, also das Frequenzspektrum des Signals, welches durchgelassen wird. Alle Frequenzen die überhalb dieses Werts liegen werden unterdrückt. Da dieser Filter sich, wie alle anderen auch, nicht ideal verhält, nimmt der Filter auch Einfluss auf die Flanken des Signals und dadurch bedingt auch auf die Amplitude.
So wurde festgestellt, dass es keinen Sinn macht den Parameter K größer als 3 zu wählen. Bereits mit K=3 werden die Spitzen des Signals bereits um ca. ein Drittel ihrer Amplitude gedämpft. Die Unterdrückung des Rauschens wird natürlich besser, je höher der Parameter K gewählt wird.
Für den Einsatz im Nao hat es sich als sinnvoll erwiesen für K den Wert 2 zu wählen. Die war für uns der beste Kompromiss zwischen guter Rauschreduzierung und qualitativ gut erhaltenen Flanken. Quantitativ werden die Beschleunigungen in den Spitzen zwar etwas verfälscht, aber für den Einsatz mit unserer Walking Engine fallen diese leichten quantitativen Fehler praktisch nicht ins Gewicht.
Neben der beschriebenen Implementierung wurden auch noch andere Filtertypen ge-

testet, unter anderem ein Median-Filter. Median-Filter werden eigentlich eher in der Bildverarbeitung zur Reduzierung von Farbrauschen eingesetzt. Das Prinzip lässt sich aber auch auf eindimensionale Signale übertragen. Der Median-Filter merkt sich eine ungerade Anzahl von Werten, sortiert sie und gibt den Wert aus der Mitte dieses Wertefensters als Ausgabewert aus. Wenn ein neuer Wert hinzugefügt wird, wird der älteste Wert gelöscht und dann das ganze Fenster neu sortiert.
Um für die Beschleunigungswerte eine akzetable Rauschunterdrückung mit einem Median-Filter zu erhalten, muss man die Fenstergröße so groß wählen, dass das gefilterte Signal, im Gegensatz zu oben genannter Implementierung, auch qualitativ nicht mehr zu gebrauchen ist. Deswegen hat man Abstand davon genommen einen Median-Filter zu verwenden.

Generelle Probleme bei Inbetriebnahme

Wenn in den vorigen Abschnitten von Signalen die Rede ist, ist immer das Signal gemeint, das von unserem Framework geliefert wird. Es handelt sich dabei nicht um das mit $640Hz$ vom Sensor abgetastete Signal. Durch Vorgaben des Frameworks, soweit sei hier vorgegriffen, wird die Datenstruktur, die die aktuellen Beschleunigungswerte enthält, alle $20ms$ mit neuen Werten gefüllt. Das entspricht einer Samplingfrequenz von lediglich $50Hz$. Dies spiegelt natürlich nur zu einem gewissen Grad das eigentliche Signal wieder. Um bessere Aussagen über die anliegenden Beschleunigungen zu treffen wäre es wünschenswert den Beschleunigungswert mit einer Frequenz von mindesten $1,28kHz$ abzutasten. Dies würde sicherlich auch helfen das Rauschen des Signals zu verringern, da dann Effekte, wie etwa Alias-Effekte, die durch die Abtastung mit zu kleiner Frequenz entstehen, aus dem Signal verschwinden.

2.1.2 Gyroskop

Gyroskop im Nao

Den zweiten Teil der inertialen Einheit bildet ein Gyroskop der Firma InvenSense. Die hier verbaute Einheit des Typs IDG-300 misst Drehgeschwindigkeiten in x- und y-Drehrichtung. Dies ist möglich bis hin zu Geschwindigkeiten von 500 $^\circ/s$. Die Genauigkeit des Gyroskops über beide Achsen werden vom Hersteller im Bereich der vollen Skala von 500 $^\circ/s$ mit $\pm 2\%$ angegeben.
Auch die Achsen des Gyroskops sind entsprechend eines rechtshändigen Koordinatensystems angeordnet. Die x-Richtung mißt die seitlichen Drehbewegungen des Roboters, die y-Richtung die Vorwärts- und Rückwärtsdrehbewegung.

Rausch- und Driftverhalten

Bei ersten Messungen mit dem Gyroskop fiel bereits auf, dass die gemessenen Werte sich hochgradig nichtlinear verhalten. Dies macht es sehr schwierig fundierte Aussagen darüber zu treffen wie akkurat die gemessenen Werte tatsächlich sind und inwiefern sie den Umständen in der Realität entsprechen. Bereits im aufrechten Stand und absolut

bewegungslosen Oberkörper driftet die y-Richtung bereits um zehn bis zwanzig Einheiten der numerischen Skala. Das entspricht Geschwindigkeitsfehlern die sich etwa um 5 $^\circ/min$ erhöhen.

Aussagekraft in verschieden Situationen

Aufgrund der im vorigen Abschnitt genannten Unzulänglichkeiten wurden keine weitergehenden Versuche unternommen, die Sensorwerte des Gyroskops innerhalb des Frameworks zu verwenden. Um diese Rohdaten für weitere Zwecke, zum Beispiel eine akkuratere Berechnung des ZMPs einzusetzen, wird eine weitergehende Verarbeitung der Rohdaten von Nöten sein. Denkbar ist hier die Verwendung eines integrierenden Filters oder ähnlichem.

2.1.3 Fußdrucksensoren

Zusätzlich zu den bereits erwähnten Sensoren sind im Nao auch noch 4 Drucksensoren in jedem Fuß verbaut. Laut Herstellerangaben handelt es sich bei den messbaren Werten der einzelnen Sensoren um die Zeit die ein Kondensator braucht um sich über einen druckabhängigen Widerstand zu laden. Der funktionale Zusammenhang zwischen den Messwerten und dem ausgeübten Druck lässt sich in etwa mit $1/X$ beschreiben. Da dies die einzigen Angaben des Herstellers waren und eine Kalibrierung mit einem sehr großen Aufwand in Verbindung gestanden hätte wurde während der Projektgruppe davon abgesehen, die Fußsensoren des NAO zu verwenden.

2.1.4 Ultraschall

Die Ultraschallsensoren sind im Gegensatz zu allen anderen Sensoren nur im realen Roboter zu finden. Für den Simulator wurden sie nicht implementiert. Eine einfache Näherung stellen die Abstandssensoren dar. Diese können beliebig oft anhand eines Strahls Abstände zu einem Objekt in eine bestimmte Richtung berechnen. Die Ultraschallsensoren sollen der Erkennung eines Hindernisses im Raum dienen. Der Nao besitzt jeweils zwei Sender und zwei Empfänger. Dies soll dazu dienen die relative Position von Hindernissen genauer abzuschätzen. Die Zeit die zwischen zwei ausgehenden Wellen liegt, beträgt ca. 270ms. Dies ist ein zu langer Zeitraum, um Hindernisse akkurat lokalisieren zu können. Grund dafür sind zu starke Oberkörperschwanken bei nahezu allen Bewegungen. Ein Updateinterval von 300ms scheint hier die beständigsten Werte zu liefern. Abwechselnd wird mit einem der beiden Emitter eine Welle ausgesand, die mit dem auf der anderen Seite liegenden Empfänger aufgenommen wird. Die zurückgelieferten Abstände der Hindernisse werden auf einen Meter beschränkt. Alle Gegenstände in einer Entfernung von über einem Meter werden nicht als Hindernisse erkannt. Anhand des Abstandes und der Oberkörperorientierung wird bestimmt, wo das Hindernis erwartet wird. Hierbei wird davon ausgegangen, dass die Welle immer genau senkrecht von der Oberkörperorientierung aus auf ein Hindernis trifft. Zusätzlich wird aus der Höhe des Hindernisses noch eine Abschätzung für die Richtigkeit der Aussage, ob ein Hindernis existiert, berechnet. Dieser Kontrollwert ist wichtig, da sonst zu häufig der Boden als

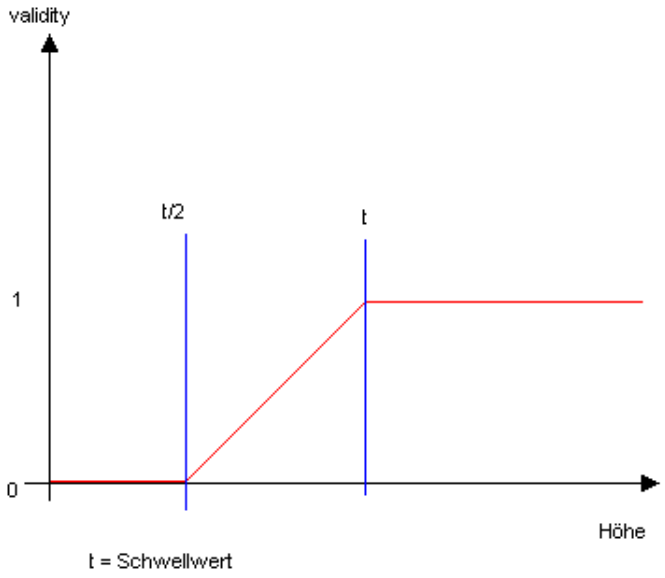


Abbildung 2.1: Abhängigkeit der Validität von der Höhe des detektierten Objektes durch einen Ultraschallsensor

Hindernis erkannt wird. Die Abhängigkeit des Kontrollwertes von der berechneten Höhe des Objektes ist in Abbildung 2.1 dargestellt. Softwaretechnisch wird NaoQI benutzt. NaoQi beschreibt die Schnittstelle an die Hardware. Es erzwingt eine Anfrage an den Actuator mit allen Parametern wie Zeit/Empfänger/Sender zu schicken, um anschliessend aus dem Speicher an der Stelle US/Sensor/Value den Abstand auszulesen. Im Spielbetrieb wurden die Sensoren nie genutzt. Zu beachten für weitere Tests bleibt, dass die Ultraschallsensoren auch die Arme des Naos detektieren können.

2.2 Sensoren im Simulator

Die Simulation von Robotern ist in Hinsicht auf die Roboterforschung und im Besonderen für Fussballroboter von essentieller Bedeutung. Damit die Algorithmen für das Walking (vergleiche Kapitel 4.1), das Egomodell (vergleiche Kapitel 3) und andere bereits vorab entwickelt und getestet werden können, ohne den realen Roboter eventuell zu beschädigen, benötigt man einen möglichst leistungsstarken Simulator. Dieser ermöglicht es, bereits auf der Simulationsebene, Fehler in der Programmierung zu entdecken und zu beseitigen. Ein Großteil der Entwicklungsarbeit innerhalb der Projektgruppe wurde am Simulator durchgeführt. Eingesetzt wurde der Simulator *SimRobot*, der vom Bremer Team B-Human entwickelt worden ist. Dieser hatte allerdings keinerlei Sensoren implementiert, wie sie für ein möglichst naturgetreues simulieren eines autonomen

Roboters notwendig sind. Für die Simulation von Sensoren benötigt man ein Physikmodell, sowie eine effiziente Physiksimulation, die die von den Sensoren zu messenden physikalischen Größen berechnet. In SimRobot übernimmt die *Open Dynamics Engine* (ODE) die Berechnung der Physik, was bedeutet, dass die Genauigkeit der Sensoren durch die Genauigkeit der Open Dynamics Engine begrenzt ist. In den folgenden Unterkapiteln werden die neuen Sensoren des SimRobots vorgestellt und Schwierigkeiten in der Entwicklung dieser erläutert.

2.2.1 Beschleunigungssensor

Die auf einen Roboter einwirkenden Kräfte können über einen Beschleunigungssensor in alle drei Dimensionen gemessen werden. Dieser wird benutzt, um Schwankungen des Roboters zu erfassen und auszugeben. Da die Position des realen im Nao verbauten Sensors nicht in Erfahrung zu bringen war, wurde der Sensor im Simulator zentral im Oberkörper nahe der z-Achse eingefügt. Die Positionierung nahe der Achse minimiert das Auftreten von Beschleunigungswerten durch Zentripetalkräfte, die während einer Längsrotation auf den Sensor wirken würden, wenn er sich nicht auf der Drehachse befinden würde. Technisch gesehen ist es möglich, die Zentripetalkräfte wieder heraus zu rechnen, allerdings bedeutet dies einen erheblichen Aufwand.
Zur Simulation des Sensors wurde ein kleiner fester physikalischer Block eingefügt, für den die in der ODE berechnete Geschwindigkeit abgefragt werden kann. Die Beschleunigung ergibt sich dann aus der Geschwindigkeitsdifferenz pro Zeitintervall, in dem beide Geschwindigkeitsabfragen stattgefunden haben. Es handelt sich also um die Bildung eines Differenzenquotienten. Das Zeitintervall ergibt sich aus der Abfragefrequenz der Sensoren. Je häufiger die Sensoren abgefragt werden, desto genauer werden die gelieferten Werte.
Um sicher zu stellen, dass die physikalischen Eigenschaften des simulierten Sensors denen eines echten Sensors entsprechen, wurden Versuche mit einem realen Beschleunigungssensor durchgeführt. Zu diesem Zweck wurde auf den triaxialen Beschleunigungssensor *Bosch SMB 380*, wie er in der Vorgänger Projektgruppe benutzt worden ist, zurück gegriffen ([PG407], Kapitel 1.7). Der Sensor verfügt über einen so genannten I^2C Datenbus, der die Kommunikation mit ihm ermöglicht. Um über diesen Datenbus mit dem Sensor kommunizieren zu können wurde mit den Materialien und Unterlagen der PG 499 ein I^2C zu *USB* Konverter aufgebaut. Zusätzlich wurde die schon vorhandene Konverter-Firmware optimiert und an den genutzten Spezialfall angeglichen. Zur Darstellung und Auswertung der vom Sensor gemessenen Werte wurde ein graphisches Tool geschrieben, welches die Sensorwerte sowohl zweidimensional in einem Graphen, als auch dreidimensional in einem Raum ausgeben kann.
Durch die Experimente mit diesem realen Sensor wurden zwei Schwachpunkte des virtuellen Sensors ersichtlich. Der Erste ist, dass man durch die Simulation einen perfekten Sensor erhält, der immer richtige Ergebnisse liefert und keinerlei Rauschen aufweist, wie es bei realen Sensoren aber der Fall ist. Ein künstliches Rauschen könnte hinzugefügt werden, indem man zufällige normalverteilte Werte auf den simulierten Sensorwert rechnet, allerdings müsste das so erzeugte Rauschen über einen Filter wieder geglättet werden.

Da über den im Nao verbauten Sensor zwar bekannt war, dass er seine Sensorwerte vor der Ausgabe sensorintern filtert und glättet, nicht aber wie das reale Rauschverhalten des Sensors aussieht, wurde auf ein künstliches Verrauschen der Werte beim simulierten Sensor verzichtet.
Das zweite Problem, das die Experimente aufzeigten war genereller Natur. Es zeigte sich, dass der reale Sensor Beschleunigungen in negativer Richtung der von außen wirkenden Beschleunigungen anzeigt. Dieser Effekt lässt sich am besten innerhalb einer Fahrstuhlkabine selbst erfahren. Angenommen die Fahrstuhlkabine stellt das zu beschleunigende System und die Person im Inneren den Sensor dar. Setzt sich die Kabine nach oben in Fahrt, so erfährt sie eine Beschleunigung in positive z-Richtung. Die Person im Inneren wird dabei stärker zu Boden gedrückt und nimmt dieses als eine vermeintliche Zunahme der Erdbeschleunigung in negative z-Richtung wahr. Gefühlte, bzw. gemessene Beschleunigungen zeigen also in entgegengesetzte Richtungen. Beschleunigt der Fahrstuhl in die umgekehrte Richtung, also nach unten, tritt der Effekt ebenso auf, nur mit umgekehrten Vorzeichen. Beschleunigungen werden also mit umgekehrten Vorzeichen wahrgenommen, wenn man ein Teil des beschleunigten Systems ist. Um diesen Effekt im Simulator nachzubilden werden alle berechneten Beschleunigungswerte negiert, anschließend wird die Erdbeschleunigung hinzu addiert.
Die aus den Geschwindigkeiten der physikalischen Objekte berechneten Beschleunigungswerte sind in Weltkoordinaten und müssen für eine korrekte Funktionsweise des Sensors in Roboterkoordinaten transformiert werden. Um dies zu erreichen ist es möglich, sich von der ODE eine 4×3 Matrix zurückgeben zu lassen. Diese enthält eine 3×3 Rotationsmatrix, sowie in der Zeile darunter die Translation des Objektes vom Ursprung des Weltkoordinatensystems bis zur aktuellen Position. Mit Hilfe dieser Matrix konnten die Beschleunigungswerte in das Roboterkoordinatensystem transformiert werden. Da der Roboter in der Simulation bereits eine Startrotation um die z-Achse des Weltkoordinatensystems besitzen kann, die in der Rotationsmatrix der ODE nicht mitgeliefert wird, musste für die Implementierung ein Rotationsoffset berechnet werden. Dies wurde sowohl für den Beschleunigungssensor, als auch für das Gyroskop durchgeführt.

2.2.2 Gyroskop

Ein Gyroskop wird in der Praxis unter anderem zur Lagebestimmung eines Objektes im Raum eingesetzt. Im Nao sind von der Firma Aldebaran Robotics Gyroskope in Chip Bauform verarbeitet, die Änderungen der Lage im Raum in Einheiten der Winkelgeschwindigkeit ($^\circ/s$) angeben. Genaue Angaben zu den eingebauten Gyroskopen können dem Kapitel 2.1 Sensoren im realen Roboter entnommen werden. Dabei können die Werte der Rotation um die x-, und y-Achse ausgelesen werden. Für den Simulator wurde ein Gyroskop implementiert das sowohl die Werte der Rotationsgeschwindigkeit um die X- und Y-Achse als auch die um die z-Achse berechnet und bezüglich des Roboterkoordinatensystems ausgibt. Es werden hier alle 3 Koordinatenachsen berücksichtigt, da zum Zeitpunkt der Implementierung des Gyroskopes noch keine Angaben über die tatsächlich benutzten Sensoren vorhanden waren. Im folgenden Absatz soll nun die Implementierung des Gyroskopes skizziert werden.

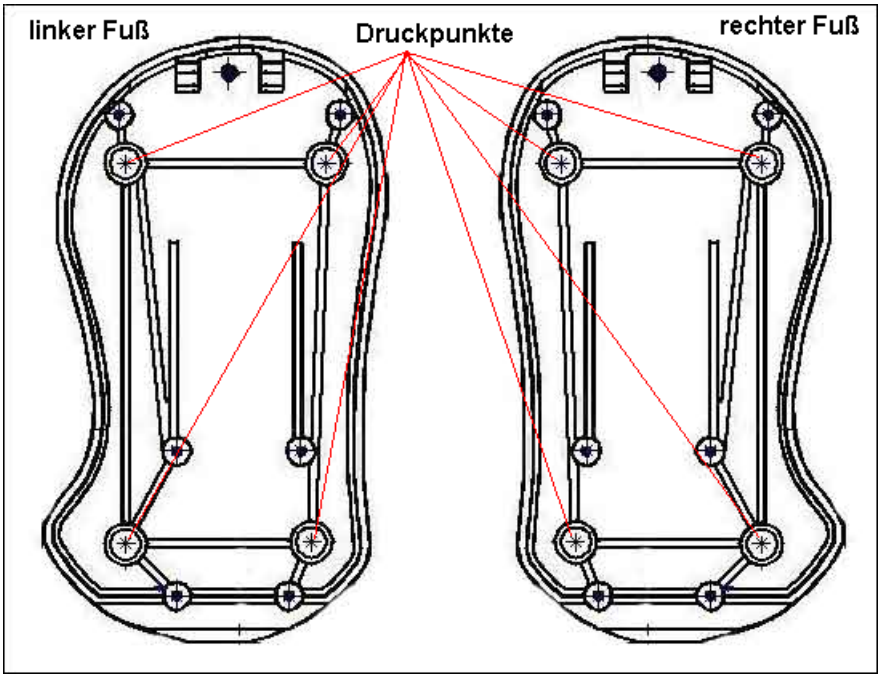


Abbildung 2.2: Schematische Darstellung eines Fusses des Nao

Über die Open Dynamics Engine im Simulator lassen sich die Winkelgeschwindigkeiten von simulierten Objekten direkt ablesen. Also wurde für den Gyroskopen stellvertretend eine Box mit geringen Dimensionen im Robotermodell platziert (siehe dazu Nao.rsi im config\scenes Verzeichnis des Projektes). Über eine eindeutige Identifikationsnummer der Body-ID, die in der Simulation jedes Objekt besitzt, werden in jedem Simulationsschritt die Rotationsgeschwindigkeiten um die x-, y- und z-Achse des Weltkoordinatensystems abgefragt. Die Werte der Rotationsgeschwindigkeiten wurden dann, wie im letzten Abschnitt zur Implementierung des Beschleunigungssensors bereits beschrieben, in das Koordinatensystem des Roboters übersetzt und ausgegeben. Die zentrale Methode in der Klasse Gyroscope der Nao Solution ist die computeValue(double*& value, int portId) Methode. Die Rotationsgeschwindigkeit einer Rotation um eine beliebige Achse entgegen des Uhrzeigersinns hat dabei ein positives Vorzeichen und die Rotationsgeschwindigkeit einer Rotation um eine beliebige Achse im Uhrzeigersinn ist mit negativen Vorzeichen definiert.

2.2.3 Fussdrucksensoren

Die Fussdrucksensoren des Naos bestehen aus vier Druckpunkten, die unter jeweils einem Fuss angebracht sind. Abbildung 2.2 zeigt eine schematische Zeichnung.

Da die Druckpunkte auf einer Ebenen liegen, sind die Druckwerte somit auf einen zweidimensionalen Unterraum eingeschränkt. Diese Einschränkung ist so minimal, dass sie

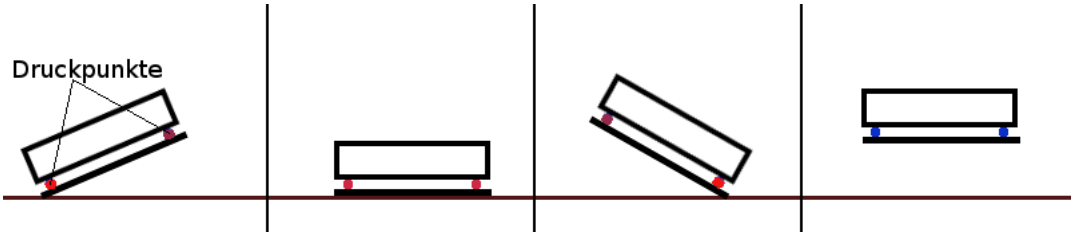


Abbildung 2.3: Abrollen des Fusses mit Bodenplatte

vernachlässigbar ist. Aus diesem Grund und aufgrund eines Geschwindigkeitsaspektes, wurden die Druckpunkte als physikalisches Objekt unter dem Fuss positioniert. Für die ODE bedeutet dies kleine Boxen oder Kugeln. Eine Einschränkung der ODE ist, dass Kräfte nicht von jedem physikalischem Objekt, sondern nur von Joints abgelesen werden können. Diese Einschränkung führte dazu, dass zwischen Fussplatte und Druckpunkten noch Slider-Joints eingebaut wurden. Diese Lösung ist jedoch nicht zufrieden stellend. Die Simulationsgeschwindigkeit nimmt drastisch ab und auch die Simulation selber weist häufiger Fehler auf. Nun kann man aber anstelle der Slider-Joints auch die Contact-Joints benutzen. Contact-Joints fügt man ein, wenn sich zwei Objekte berühren. In der ODE können Berührungen von Objekten durch Contact-Joints dargestellt werden. Diese haben nur den Nachteil, dass sie nur für einen Simulationsstep gültig sind. Dieses und die Tatsache, dass ODE die Feedbackvariablen nur füllt, sie aber weder allokiert noch deallokiert, haben zur Folge, dass man ein aufwändigeres Speichermanagement hat. Dazu überprüft man jeden Contact-Joint, ob er an einem Druckpunkt sitzt. Falls dies der Fall ist ordnet man ihn dem entsprechendem Druckpunkt zu und reserviert Speicher für die ODE (`dJointFeedback`). Nun kann es passieren, dass die Fussdruckpunkte gegeneinander stoßen. Da beiden Druckpunkten nun der gleiche Joint zugewiesen wird, muß der gleiche Feedback-Pointer verwendet werden. Hier muss darauf geachtet werden, dass der Pointer nicht überschrieben wird und auch nur einmal gelöscht wird. Das Nutzen der Contact-Joints löst zwar die Laufzeit und Physikgenauigkeitsprobleme, ein Problem bleibt jedoch bestehen. Die Druckpunkte können nicht hinreichend klein genug, gemacht werden, so dass, sollte der Fuss einmal nicht parallel zum Boden stehen, die Druckpunkte weitere Kippkanten bilden. Dies hat zur Folge, dass meistens nur zwei Sensorwerte korrekt sind. Man stelle sich vor der Roboter würde eine gehende Bewegung ausführen, dann würde er erst mit der hinteren Kante des Fusses den Boden berühren, so lange bis der Fuss parallel auf dem Boden steht. Danach würde sich die Fussplatte vom Boden wegbewegen, so dass nur die vordere Kante dabei den Boden Berührt.
Vergleicht man nun das Model mit Platte unter den Sensorpunkten (Abbdilung 2.3) und die ohne Platte (Abbdilung 2.4), so fällt auf, dass bei der Plattenversion immer alle vier Druckpunkte beansprucht werden. Die Version ohne Platte misst beim Auftreten nur an den hinteren Druckpunkten. Bei parallelem Stand werden plötzlich alle vier Druckpunkte einen Wert ausgeben, und beim Abrollen nach vorne hin werden nur die beiden vorderen Druckpunkte benutzt. Trotz dieses Nachteils können die simulierten Fussdrucksensoren genutzt werden. Die Frage welcher Fuss den Boden berührt lässt sich mit jedem Model

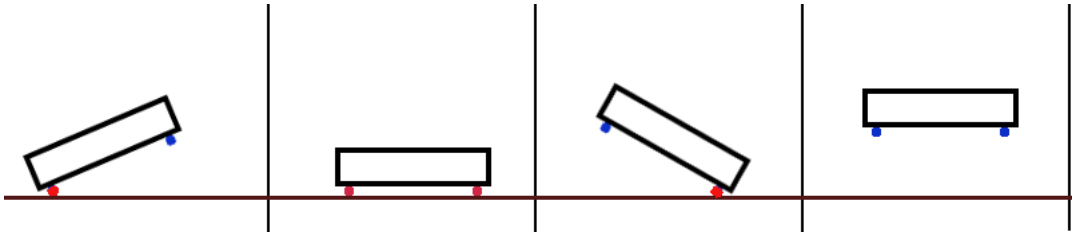


Abbildung 2.4: Abrollen des Fusses ohne Bodenplatte

leicht beantworten und wäre sicher für die Zukunft eine gute Erweiterung.

Implementierung

Der Simulator stellt mit dem Bumper bereits einen ähnlichen Sensor bereit. Der Bumper testet, ob er gedrückt wird. Der FootSensor ist somit eine Erweiterung des binären Falles drücken auf den analogen Fall, der verschiedene Stufen des Druckes kennt. Als logische Konsequenz wird der FootSensor also vom Bumper abgeleitet. Die zusätzliche Methode `void addJointId(dJointID *id, int n, dGeomID gid)` in der FootSensor Klasse dient dazu während der Physikberechnung die Contact-Joints an dem jeweiligen Fuss den Sensorpunkten zuzuordnen. An dieser Stelle bekommt auch der Contact-Joint einen Feedback-Pointer zugewiesen. Die Verwaltung des Feedback-Pointers bleibt jedoch in der FootSensor-Klasse, die dazu eine Liste von `IDCollectoren` hält. In der writeBack-Phase des Frameworks, bzw. damit auch des FootSensors, wird durch die Liste der `IDCollectoren` iteriert um die Kräfte der Contact-Joints auf die Sensorpunkte zu addieren. Ein `IDCollector` speichert die Adressen der Feedback-Pointers, Contact-Joints und die Anzahl die zu einer bestimmten `dBodyID` (Druckpunkt) gehören.

3 Egomodell

Nachdem im vorherigen Kapitel die Funktionsweise der Sensoren und deren Simulation näher erklärt wurde, ist noch ein weiterer wichtiger Entwicklungsschritt zu gehen, um die Basis für ein stabiles dynamisches Laufen zu schaffen.
Die Implementierung der Sensoren ermöglicht es, erstes Wissen über den aktuellen Bewegungszustand des Roboters zu erlangen. Neben den Daten, die Beschleunigungs-, Fußdrucksensoren oder Gyroskope liefern, ist es jedoch auch noch nötig, Informationen über die Gelenkstellungen, die kinematische Struktur, und die Schwerpunkte zu erhalten. Dazu wurde eine geeignete Repräsentation gesucht und im sogenannten Egomodell gefunden.
Erstes Ziel dieses Moduls zur Selbst-Repräsentation des Roboters ist dabei alle Daten zur Verfügung zu stellen, die die Walking Engine benötigt (Kapitel 4.1). Dieser dynamische Lauf verlangt ein Stabilitätskriterium, genannt Zero-Moment-Point, für welches wiederum der Gesamtschwerpunkt des Roboters benötigt wird. Dadurch ergeben sich die Hauptaufgaben des Egomodells zum einen in der dynamischen Berechnung des Gesamtschwerpunkts des Roboters zu jedem Zeitpunkt, und zum anderen in der Bereitstellung des Zero-Moment-Point.
Auf lange Sicht ist es hilfreich noch weitere Informationen vom Egomodell berechnen zu lassen. Somit sollen in Zukunft auch Drehmomente die auf den Roboter wirken an dieser Stelle berechnet und zur Verfügung gestellt werden.
Im Folgenden wird daher auf die Umsetzung der oben genannten Hauptaufgaben zur Unterstützung der Walking-Engine eingegangen. Dazu ist es zuallererst nötig, ein möglichst exaktes Abbild der kinematischen Struktur des Roboters bereitzustellen. Dies erfolgt durch das Parsen des Robotermodells, welches bereits im Simulator zum Einsatz kommt. Da dieses Modell viele für das Egomodell unnötige Informationen enthält, ist eine Reduzierung nötig. Daraus folgt die Möglichkeit die einzelnen Schwerpunkte zu berechnen, aus denen wiederum der Gesamtschwerpunkt resultiert. Letzterer ermöglicht die Berechnung des Zero-Moment-Points. Dieser kann auf unterschiedliche Weisen berechnet werden, was ebenfalls vom Egomodell berücksichtigt wird.

3.1 Modellerzeugung

Als Grundlage für das zu erstellende kinematische Modell dient das im Simulator benutzte Robotermodell. Dieses liegt in einem XML-Format vor, genannt RoSi und es enthält neben der gesamten Beschreibung der Szene auch die einzelnen Elemente des Roboters, aufgelistet in einer Baumstruktur. Die Definitionen beinhalten sowohl das äußere Erscheinungsbild als auch verschiedene physikalische Eigenschaften, woraus die kinematische Struktur abgeleitet werden kann. Im RoSi-Format können verschiedene Typen von Daten abgespeichert werden. Für die folgenden Arbeiten sind dabei die Rotations-

und Translationensangaben, die zu den Massenschwerpunkten der einzelnen Elemente als auch zu den Positionen der Gelenke führen, am bedeutendsten. Dabei werden mm-Entfernungen in x-, y-, z-Richtung angegeben, oder eine Rotation um diese Achsen in Grad. Jede dieser Angaben ist im eigenen Koordinatensystem, welche in das Roboterkoordinatensystem umgewandelt werden können, wenn man alle Transformationen entlang eines Pfades zurückgeht. Neben dem Roboter Bender (der bereits in diesem Format vorlag) konnte auch der Nao in diese Form exportiert werden, da ein Modell aus dem Webots-Simulator existiert, welches von der Universität Bremen in das RoSi-Format konvertiert wurde. Es zeigte sich jedoch mit der Zeit, dass dieses Modell nicht exakt mit dem echten Nao übereinstimmte, weswegen Änderungen daran vorgenommen werden mussten. Damit anhand der Informationen ein geeignetes Robotermodell für das EgoModell erzeugt werden kann, ist es nun nötig die vorhandene XML-Datei zu parsen. Zur Verwaltung der Informationen ist die neue Objektklasse `Parts` vorgesehen, die im Folgenden vorgestellt wird.

3.1.1 Die Objektklasse `Parts`

Die hier vorgestellte Objektklasse dient dazu die interne Repräsentation des gesamten Roboters und seiner kinematischen Eigenschaften zu realisieren. Dabei sind die `Parts`-Objekte als Knoten beziehungsweise Blätter eines Baumes organisiert, so dass ein einzelnes Objekt mehrere Kinder und einen Elter haben darf. Ziel ist, dass jedes `Parts`-Objekt je ein starres Glied der 5 kinematischen Ketten (Kopf, Arme, Beine), als auch die Basis (Körper) des Roboters darstellt, weshalb jedes Objekt höchstens ein Gelenk enthalten darf. Desweiteren werden auch Sensoren als `Parts` behandelt.
`Parts` enthalten daher Attribute für die Dimension, die Lage des Gelenks und seine Ausrichtung, die Position des Schwerpunkts und die Masse. Diese Informationen reichen zur Modellierung der kinematischen Struktur aus. An dieser Stelle kam zur Abspeicherung der Positionen eine bereits implementierte Klasse zum Einsatz, die alle gängigen Rechenoperationen für Lagetransformationen im dreidimensionellen Raum bereitstellt. Dessen ungeachtet werden noch neben einigen Hilfsvariablen die maximalen Achsauslenkungen, maximale Geschwindigkeiten und maximale Kräfte abgespeichert, da diese Daten vom Robotermodell geliefert werden. Sie werden weiter nicht beachtet, können sich aber in Zukunft als nützlich erweisen.
Die Lagebeschreibung erfolgt mit Hilfe von Rotationsmatrizen und Translationsvektoren, die bereits geeignet implementiert waren. Der Schwerpunkt liegt jeweils als Vektor im Koordinatensystem des zugehörigen `Parts`.
Jedes `Parts`-Objekt erhält dabei eine Variable für die Lagetransformation in die jegliche auftretende Rotationen und Translationen integriert werden, so dass eine Transformation von diesem zum folgenden Objekt möglich ist, als auch einen Vektor, der die Position des Schwerpunktes beschreibt. Dieser liegt im Koordinatensystems des Gelenks vor. Schließlich bietet so jedes Objekt Koordinaten des Schwerpunkts im Koordinatensystem des Gelenks, als auch die Transformation von diesem zum nächsten Objekt (Siehe Kapitel 3.1.3 und Kapitel 3.2).
Die Baumstruktur der `Parts` beschränkt nicht die Anzahl der Kinder, die ein Objekt

hat. Dadurch ist es auch denkbar Roboter mit anderen kinematischen Strukturen abzuspeichern. Das Ergebnis sollte eine möglichst universelle Datenstruktur sein, so dass die Arbeit auch für andere Projekte nutzbar sein kann. Im Folgenden wird gezeigt auf welche Weise die `Parts`-Objekte aus dem in XML vorliegendem Robotermodell geparst werden.

3.1.2 Der EgoModellparser

Um das EgoModell automatisch erstellen zu können ist es nötig, die entsprechenden Daten aus der Definitionsdatei des Roboters auszulesen. Hierfür wurde im wesentlichen die XML-Parser Implementierung, welche auch im Simulator selbst benutzt wird, adaptiert. Die Implementierung basiert auf der LibXML2 des Gnome Projekts und nutzt dessen 'Simple Api for XML' (SAX2). Das Auslesen der Datei wird dabei von der Bibliothek selbst übernommen. Auch die Validierung übernimmt die Bibliothek selbst. Die Anbindung an die Bibliothek sowie alle Aufrufe von Bibliotheksfunktionen sind in der Klasse LibXML2EgoParser zusammengefasst. Diese Klasse ruft für jedes XML-Element in der Datei jeweils eine 'start-' und 'end-Element()', sowie eine 'start-' und 'End-Document()'-Methode eines Handlers auf. Dieser Handler ist der eigentliche SAX2-Teil des Parsers und beinhaltet die Regeln und Funktionen, wie mit den jeweiligen XML-Daten zu verfahren ist.
Von Interesse für das EgoModell sind lediglich physikalische Definitionen wie Abmessungen, Reihenfolge der Gelenke, das Gewicht einzelner Teile und die Position der Teile und Gliedmaßen. Alle Elemente die Informationen über das Erscheinungsbild und auch über die Formen verschiedener Bauteile des Roboters enthalten, werden vom Parser ignoriert, da sie für das EgoModell keinerlei Relevanz haben.
Nun soll zunächst die Struktur einer Roboterdefinition im rsi-Format näher erläutert werden. Da es sich beim rsi-Format um ein XML-Format handelt, liegen alle Daten in einer verschachtelten Baumstruktur vor. Diese Baumstruktur soll auch für das Ausgabeformat des Parsers erhalten bleiben. Während des Parsens werden aus den XML-Daten `Part`-Objekte erstellt und entsprechend der Struktur im XML-Dokument verkettet. Die Wurzel des so entstandenen Baums aus `Parts` wird beim Beenden des Parsens an das EgoModell übergeben.
Grundlegendes Steuerelement während des Parsens des rsi-Formats ist das Element `<macro>`. Makroelemente werden im rsi-Format zum Strukturieren der physikalischen Attrribute einzelner Teile des Roboters, als auch des ganzen Roboters benutzt. Makroelemente enthalten zusätzlich zu den physikalischen Daten eines Bauteils, auch Informationen über das äußere Erscheinungsbild.

```
<Macro name="VIFS362">
  <ComplexShape name="VIFS36CS2">
    <Appearance ref="nao-white"/>
    <GraphicalRepresentation vertexList="VIFS36.vertices">
        <GraphicalAttributes>
...
      </GraphicalAttributes>
```

```
      </GraphicalRepresentation>
      <PhysicalRepresentation>
        <SimpleBox name="lFoot" length="0.09" width="0.16" height="0.02" canCollide
          <Translation z="-0.046" y="-0.03"/>
          <PhysicalAttributes>
            <Mass value="0.001"/>
          </PhysicalAttributes>
        </SimpleBox>
      </PhysicalRepresentation>
    </ComplexShape>
  </Macro>
```

Im Beispiel oben wird der linke Fuß beschrieben. Man sieht, dass ein `<ComplexShape>` sowohl ein `<GraphicalRepresentation>` als auch ein `<PhysicalRepresentation>` enthalten. Wie bereits erwähnt ist für das EgoModell nur die physikalische Beschreibung von Interesse. Aus diesem Grund werden nur die Elemente der `<PhysicalRepresentation>` vom Parser verarbeitet.
Makros für verschiedene Teile des Roboters werden in der Regel am Anfang der Datei definiert. Damit sie später in einem größeren Zusammenhang verwendet werden können. Dieser Mechanismus ermöglicht es auch Elemente mehrfach zu verwenden, etwa für symmetrische Teile wie die Oberschenkel des Roboters.
Über ein `<Use>` werden vorgefertigte Makros benutzt.

```
  <Use macroName="VIFS362" instanceName="lFoot">
    <Translation z="0.01"/>
      <Elements>
...
      </Elements>
  </Use>
```

Unterhalb des Element-Tags können wiederum `<Use>`-Elemente benutzt werden. Hier wird der vorgefertigte linke Fuß benutzt. Er besitzt bereits Abmessungen und Positionsangaben der Elemente aus denen er besteht. Diesem vorgefertigten Makro können auch noch zusätzliche Attribute hinzugefügt werden. Im Beispiel ist das die Translation vom vorangegangenen Element zum Fuß selbst.
Die erste auftretende Translation, die beim Nao im Torso auftritt, beschreibt dabei die Translation vom Koordinatenursprung des Roboters. Dieser Torso wird als Wurzel des Part-Baums als `Part* body` an das EgoModell übergeben wenn der Parser die XML-Datei abgearbeitet hat.
Es dürfen beliebig viele Makros in einer Datei definiert werden, jedoch niemals ein `<Macro>` innerhalb eines anderen. `<Macro>` darf aber beliebig oft `<Use>` enthalten.
Der Parser erstellt für jedes `<Macro>` ein neues Part-Objekt und setzt alle physikalischen Eigenschaften die er für dieses Bauteil findet. Im Beispiel der Makrodefinition oben etwa wären das die Abmessungen, die Translation und das Gewicht. Die vorgefertigten Part-Objekte werden in einer Liste abgelegt. Wenn nun ein `<Use>` ein Makro verwenden

will, durchsucht der Parser die Liste nach dem Makronamen und kopiert das gefundene Part-Objekt in ein neues mit dem Instanznamen der Use-Deklaration als Namen für das neue Part-Objekt. Das vorgefertigte Objekt bleibt in der Liste erhalten, falls diese Art Bauteil nochmal benötigt wird.
Ein weiteres wichtiges Steuerelement für den Parser ist `<Elements>` beziehungsweise `<Element>`. Diese Elemente benutzt der Parser um die Ebene festzulegen, auf der sich ein `Part`-Objekt im Baum befindet. Das bedeutet im Wesentlichen, dass ein Bauteil, welches eine Elemente-Definition enthält, weitere `Part`-Objekte als Kinder hat. Der Parser hängt alle Bauteile, die sich innerhalb der Element-Definition befinden, als Kinder des übergeordneten `Part`-Objekts in den Baum. Ihr Elter wird enstprechend das übergeordnete `Part`-Objekt, so dass der Baum immer doppelt verkettet ist.
Als letztes Steuerelement sei hier noch `<Movable>` erwähnt. `<Movable>` darf innerhalb der Roboterdefinitionsdatei nur genau einmal vorkommen. Wenn ein `<Macro>` ein `<Movable>` enthält, beinhaltet es die fertige Definition des Roboters. Dieses Makroelement wird nicht in die Liste der vorgefertigten Elemente eingefügt und kann innerhalb der Roboterdefinitionsdatei auch von keinem `<Use>` benutzt werden. Dieses `<Macro>` wird als `body`-Element an das EgoModell übergeben. In der Regel findet sich dieses Makroelement erst am Ende der Roboterdefinitionsdatei und das Parsen wird nach Abarbeiten dieses Bauteils beendet. Dies ist zwar nicht zwingend erforderlich, aber üblich.

3.1.3 Reduktion des geparsten Modells

Offensichtlich ist das in Kaptitel 3.1.2 erzeugte Modell zu groß. Es enthält viele Objekte, von denen einige keinerlei oder wenig Informationsgehalt besitzen. Diese Objekte beizubehalten würde die Laufzeit der Schwerpunktberechnung verschlechtern. Daher bietet es sich an das Modell soweit zu verkleinern, dass pro Glied des Roboters nur ein Objekt existiert. Im folgenden wird gezeigt wie die Reduzierung auf die nötigen Objekte abläuft. Die Reduzierung verläuft rekursiv nach dem Prinzip der Tiefensuche. Sie beginnt auf dem Basisobjekt. Von hier aus werden nun alle Kinder betrachtet, wobei immer jeweils zum tiefstmöglichen Kind traversiert wird. Es wird unterschieden zwischen Knoten, die beibehalten werden, und denjenigen, die mit den beizubehaltenden Knoten verschmolzen werden. Beibehalten wird dabei die Basis, als auch jeder Knoten, dessen Objekt ein Gelenk darstellt. Dies ist leicht unterscheidbar, da nur Gelenke auch Informationen über ihren Gelenkzustand enthalten. Erreicht der Lauf über den Objektbaum also einen Knoten, der ein Gelenk darstellt, wird er nicht verändert.
Sonst werden 2 Fälle behandelt:

1. Das betrachtete `Parts`-Objekt A ist ein Sensor, das heißt es wird eine neues Objekt einer Sensorklasse in eine geeignete Liste des Elter geschrieben und der Knoten beziehungsweise das Blatt aus dem Baum entfernt

2. Das betrachtete `Parts`-Objekt A ist weder Gelenk noch Sensor. In diesem Fall wird der Schwerpunkt von A mit dem Schwerpunkt des Elter zusammengefasst. Dies ist ohne weiteres möglich, da sich beide Schwerpunkte in einem starren physikalischen

Objekt befinden. Bevor jetzt das Kind-Objekt gelöscht werden kann muß noch beachtet werden, dass es sich bei dem Kind des Kindes A um ein Gelenk handeln kann. Sei dieses Kind des Kindes A das Objekt B, so muss die Transformation von dem Elter von A zu B erhalten bleiben. Die Lage von B ändert sich also, da als Bezugskoordinatensystem von B jetzt dasjenige von A in Frage kommt, was durch

$$T'_B = T_A \cdot T_B$$

wenn T als die jeweilige homogene (4x4)Transformationsmatrix wird. Nun kann A aus dem Baum entfernt werden und B wird das neue Kind des Elter von A.

Nach der Ausführung dieses Algorithmus enthält der `Parts`-Objektbaum jetzt je einen Knoten für jedes Gelenk des Roboters und einen Knoten für die Roboterbasis (dem Oberkörper bei einem humanoiden Roboter). Alle Objekte enthalten einen eindeutigen Schwerpunkt und die Transformation vom eigenen Koordinatensystem zu diesem Schwerpunkt. Desweiteren ist es im Laufe obiger Arbeitsschritte möglich den Gelenken noch ihre aktuellen realen beziehungsweise simulierte Gelenke zuzuordnen, so dass später Winkelstellungen auslesbar sind.
Die hier erläuterten Arbeitsschritte sind vorarbeiten und finden nicht zur Laufzeit im Simulator oder auf dem richtigen Roboter statt. Der Aufwand an dieser Stelle ermöglicht es, die Effizienz der Berechnungen, die zur Laufzeit stattfinden, zu erhöhen. An dieser Stelle stehen somit die statischen Schwerpunkte aller starren Teile des Roboters zur Verfügung. Im Folgenden wird gezeigt, wie aus diesen Daten der dynamische Gesamtschwerpunkt effizient erzeugt wird.

3.2 Dynamische Schwerpunktberechnung

3.2.1 Grundlagen

Wie bereits beschrieben existieren Objekte, die je ein Glied der kinematischen Ketten des Roboters darstellen. Diese besitzen einen statischen Schwerpunkt. Der Gesamtschwerpunkt mehrere statischer Objekte ermittelt sich dabei über:

$$COM_{gesamt} = \frac{\sum \vec{x_i} \cdot m_i}{\sum m_i}$$

Hierbei ist $\vec{x_i}$ der jeweilige Ortsvektor eines einzelnen Schwerpunkts und m_i seine Masse. Das Problem ist nun, dass die einzelnen Glieder des Roboters ihre Lage ständig ändern. Dadurch verändert sich auch der Schwerpunkt. Es ist also nötig, in jedem Berechnungsschritt die Gelenkstellungen mit einzubeziehen. Dies geschieht mit Hilfe der direkten *Vorwärtskinematik*.
In der Robotik beantwortet die Vorwärtskinematik die Frage, wie man aus den Gelenkwinkeln der Armelemente eines Roboters die Lage des Endeffektors[4] errechnet. Dabei

4 Der Endeffektor eines Roboters ist der Punkt des letzen Armes, an dem sich das Werkzeug befindet. Bei humanoiden Robotern könnte man diesen Punkt mit der Fingerspitze einer Hand veranschaulichen.

werden die Transformationsmatrizen einzelner Armelemente, die auf dem Weg zum Endeffektor liegen, in Reihenfolge ihres Auftretens multipliziert. Zusätzlich muss der aktuelle Winkel der Gelenke als Rotationsanteil in die Berechnung einfließen. Üblicherweise werden in der Industrie *Denavit-Hartenberg-Parameter* zur Modellierung der Vorwärtskinematik verwendet(DH-Verfahren) [Rie92].
Das DH-Verfahren basiert dabei auf Beschreibungsvorschriften für die kinematische Struktur. Beispielsweise ist die Drehachse des Gelenks immer die Z-Achse. Um von einem Gelenk zum nächsten zu gelangen sind nur Rotationen oder Translationen um die X- und die Z-Achsen erlaubt. Reihenfolge der Rotationen und der Translationen sind vorgeschrieben. Die daraus entstehenden Transformationsmatritzen ergeben nun, sofern sie nacheinander multipliziert werden, die Vorwärtsberechnung der Kinematik. Der Vorteil des DH-Verfahrens liegt darin, dass der Winkel eines Gelenks immer als Rotation um die Z-Achse zu den jeweiligen Transformationsmatrizen gerechnet werden kann.
Der Versuch dieses Verfahren zur Berechnung des Roboterschwerpunktes einzusetzen scheiterte jedoch. Es zeigte sich jedoch, dass eine automatisierte Modellierung dieser Parameter aufwendig ist. Da es nötig war schnell zu einem Ergebnis zu kommen und die vorliegenden Roboterstrukturen sehr speziell sind, erschien es ausreichend, einen einfacher umzusetzenden Ansatz zu wählen.

3.2.2 Die Vorwärtskinematik zur Schwerpunktberechnung

Die nun implementierte Vorwärtskinematik erwies sich als erwartet einfach. Da aus dem im RoSi-Format vorliegendem Robotermodell direkt hervorging, um welche Achse oder Orientierung das Gelenk rotiert, konnte der Rotationsteil in Form von einfachen Rotationsmatrizen zu den vorhandenen Transformationsmatrizen gerechnet werden. Dies geschieht durch einfache Matrixmultiplikation.

Daraus ergaben sich für die Vorwärtskinematik die folgenden Berechnungsschritte:
Sei T_i die jeweilige Transformationsmatrix um von Gelenk $i-1$ zu Gelenk i zu gelangen, R_i die Rotationsmatrix, die die Gelenkwinkelstellung von Gelenk i enthält und $\vec{X}_i$ der Ortsvektor des Schwerpunkts vom i-ten Teil, dann ist

$$COM_i = (T_i \cdot R_i) \cdot \vec{X}_i$$

der Schwerpunkt des i-ten Teils einer kinematischen Kette des Roboters.

In dieser Formel ist noch eine Abhängigkeit erkennbar, denn damit die Berechnung ausgeführt werden kann, muss die Transformation von Gelenk $i-1$ zu Gelenk i bereits bekannt sein. Um dies zu gewährleisten müssen, bevor der Schwerpunkt i berechnet werden kann, T_i, T_{i-1} , usw. berechnet sein.
Da in jedem Frame der aktuelle Gesamtschwerpunkt des Roboters berechnet werden soll, ist es immer erforderlich, dass alle Transformationen ausgerechnet werden. Daher werden die `Parts`-Objekte noch vor der Laufzeit gemäß ihrer kinematischen Abhängigkeiten sortiert, und nach dieser Sortierung in ein Array geschrieben. Nun wird während der

Laufzeit von jedem Arrayelement zuerst die Transformationsmatrix und anschliessend der Schwerpunkt berechnet. Da die Baumstruktur noch erhalten ist, kann in jedem folgenden Arrayelement die Positionsberechnung auf die Transformationsmatrix des Elter zurückgreifen. An dieser Stelle wird offensichtlich, dass keinerlei Berechnungen doppelt ausgeführt werden. Das Ergebnis ist eine effiziente, Redundanz vermeidende Berechnung, aus der die praktische Anwendbarkeit des Schwerpunkts für die niedrig getakteten Prozessoren der Naos folgt.

3.2.3 Ergebnis der Schwerpunktberechnung

Die Möglichkeit, den Schwerpunkt zur Berechnung des Zero-Moment-Points und so schliesslich als Teil des Stabilitätskriteriums der Walking Engine zu nutzen, erwies sich als sehr praktikabel. Allerdings wurde festgestellt, dass das aus dem Webots-Simulator stammende Modell nicht vollständig dem realen Modell entsprach. Es war also nötig, das exportierte RoSi-Modell noch entsprechend der Hardwaredokumentation von Aldebaran [Ald08b] zu modifizieren.
Dies verbesserte den Schwerpunkt, allerdings konnte auch hier festgestellt werden, dass die Spezifikationen nicht exakt der Realität entsprachen. Dies wurde durch Wiegen des gesamten Roboters festgestellt, was eine Differenz zwischen Realität und Dokumentation offenbarte. Dies Auszugleichen war nur näherungsweise möglich, da der Akku des Nao offenbar nicht im Simulationsmodell integriert war. Die Anpassung basiert jedoch nur auf Vermutungen, führt allerdings zu einem Schwerpunkt, der das Laufen sichtlich unterstützt und verbessert.
Genauere Ausmessungen des Naos sind leider im Rahmen der PG521 nicht möglich gewesen und gestalten sich auch als schwierig, da ein Garantieverlust drohen würde.
Die trotzdem erzielte Optimierung des Laufens durch die Bereitstellung eines angenäherten Schwerpunkts zeigt jedoch die Wichtigkeit dieser Berechnungen und, dass eine derartige Genauigkeit ausreicht um eine deutliche Verbesserung des Laufes zu erreichen.

3.3 Zero Moment Point

Aus den Daten die das Egomodell und die Sensoren liefern kann der Zero Moment Point berechnet werden. Dieser stellt ein wichtiges Kriterium für die momentane Stabilität des Roboters dar und lässt sich auf verschiedene Arten berechnen. So ist es mit seiner Hilfe mÃ¶glich zu entscheiden, ob der Roboter stabil steht beziehungsweise läuft, oder wie gegengesteuert werden muss um gerade dies zu erreichen. Zur Zeit werden diese Daten nur von der Walking Engine (Kapitel 4.1) genutzt, aber denkbar sind auch Anwendungen beispielsweise beim Schießen.

3.3.1 Physikalischer Hintergrund

Definiert ist der ZMP als der Punkt P_{ZMP} auf der den Roboter stützenden Oberfläche, an dem die Kippmomente um X- und Y- Achse Null werden. Die Kippmomente entstehen durch von Außen angreifende Kräfte. Für jeden Punkt P auf der Oberfläche lassen sich

die Kippmomente M_P wie folgt berechnen:

$$M_P = \sum_i \overrightarrow{PP_i} \times m_i \cdot (\vec{g} + \vec{a_i})$$

Für den ZMP P_{ZMP} muss nun gelten:

$$M_{ZMP,x} = 0 \qquad \text{und} \qquad M_{ZMP,y} = 0$$

Auf Grund der physikalischen Unabhängigkeit der Raumrichtungen kann man beide Bedingungen einzeln betrachten. Durch Auflösen in die einzelnen Komponenten ergeben sich folgende Beziehungen:

$$M_{ZMP,x} = \sum_i (P_{i,y} - P_{ZMP,y}) \cdot m_i(g_z + a_{i,z}) - (P_{i,z} - P_{ZMP,z}) \cdot m_i(g_y + a_{i,y}) = 0$$

$$M_{ZMP,y} = \sum_i (P_{i,x} - P_{ZMP,x}) \cdot m_i(g_z + a_{i,z}) - (P_{i,z} - P_{ZMP,z}) \cdot m_i(g_x + a_{i,x}) = 0$$

Als Lösung erhält man durch Umformen und Einsetzen der Bedingung $P_{ZMP,z} = 0$, da der ZMP definitionsgemäß in der X-Y-Ebene liegt:

$$P_{ZMP,x} = \frac{\sum_i m_i(P_{i,x}(g_z + a_{i,z}) - P_{i,z}(g_x + a_{i,x}))}{\sum_i m_i(g_z + a_{i,z})}$$

$$P_{ZMP,y} = \frac{\sum_i m_i(P_{i,y}(g_z + a_{i,z}) - P_{i,z}(g_y + a_{i,y}))}{\sum_i m_i(g_z + a_{i,z})}$$

Mit diesem Formel-Paar kann nun die Lage des ZMP berechnet werden, unter der Voraussetzung, dass die wirkenden Teilkräfte aus den Daten der Sensoren bestimmt werden können.
Befindet sich der ZMP innerhalb der konvexen Hülle der Auflagepunkte des Roboter steht der Roboter stabil. Befindet der ZMP sich außhalb dieser Fläche wirkt ein Kippmoment, das den Roboter in eine entsprechende Richtung verlagert. Auch dies kann erwünscht sein, z.B. beim Laufen.

3.3.2 Berechnung aus Beschleunigungssensor und Schwerpunkt

Wie bereits weiter oben ausgeführt wurde, verfügt der Roboter über einen 3-Achsigen Beschleunigungssensor. Zusammen mit der ebenfalls oben ausgeführten Berechnung des Roboter-Schwerpunktes aus dem internen Modell, kann der ZMP bestimmt werden.

Theorie

Bei dieser Bestimmungsmethode wird ein recht einfaches physikalisches Modell zugrunde gelegt, bei dem der Roboter als ein kompakter Körper angenommen wird, dessen gesamte Masse im Schwerpunkt vereint ist. Weiterhin nimmt man an, dass sich auch

der Beschleunigungs-Sensor zumindest näherungsweiße an diesem Punkt befindet. Auf Grund des sehr einfachen Modells reduzieren sich die obigen Formeln nochmals:

$$P_{ZMP,x} = P_{COM,x} - P_{COM,z}\frac{g_x + a_x}{g_z + a_z}$$

$$P_{ZMP,y} = P_{COM,y} - P_{COM,z}\frac{g_y + a_y}{g_z + a_z}$$

Die Werte für $P_{COM,x}$, $P_{COM,y}$ und $P_{COM,z}$ sind bekannt und der Beschleunigungs-Sensor liefert genau die Werte für $g_x + a_x$, $g_x + a_x$ und $g_z + a_z$. Der ZMP ist also in diesem Modell nichts anderes als der, entlang des vom Beschleunigungssensors gelieferten Vektor, auf den Boden projezierte Massenschwerpunkt.
Das Modell läßt sich noch verfeinern, wenn die Position des Sensors genau bekannt ist, sowie verlÃ¤ssliche Annahmen über Stärke und Richtung der Schwerkraft getroffen werden köennen.

Umsetzung auf dem Roboter

Auf dem Roboter wurde dieses Verfahren exakt so umgesetzt. Es bestand aber das Problem der Koordinaten Umrechnung, da für die ZMP-Berechnung das Koordinaten-System über den Boden bestimmt war, und nicht über einen festen Punkt am Roboter. Es musste also eine Annahme über die Position und Lage des Bodens getroffen werden. Man entschied sich hierbei hinsichtlich der Position für den jeweils tieferen Fuss. Bei der Lage wurde zunächst versucht, diese aus den Gyroskopen und der vom Beschleunigungssensor gemessenen Erdbeschleunigung zu bestimmen, aber dies erwies sich als nicht zuverlässig im Einsatz auf der realen Roboterhardware. Somit wurde dann einfach die X-Y-Ebene angenommen.
Dieses Modell wurde für die dynamische Laufsteuerung verwendet und erzielte zufriedenstellende Ergebnisse, obwohl leider keine Vergleichsdaten zu anderen Modellen vorliegen.

3.3.3 Berechnung aus dem internen Kinematischen Modell

Ein weiterer Ansatz zur Bestimmung des ZMP war die Berechnung aus dem internen Modell des Roboters. Aus der Motor-Ansteuerung und dem Wissen um den Roboteraufbau köennen ebenfalls die nötigen Daten ermittelt werden.

Theorie

Betrachtet man die kinematischen Ketten, aus denen der Roboter besteht, kann man alle Teile zwischen den Gelenken zusammenfassen und erhält so ein Modell, aus dem sich der ZMP mit der bekannten Formel direkt bestimmen lässt. Position und Masse der einzelnen Teile sind direkt im Modell enthalten, die Beschleunigungen kann aus den Motoransteuerungen ermittelt werden. Für die Gravitation muss eine passende Annahme über Richtung (in der Regel direkt nach unten) und StÃ¤rke getroffen werden.
Damit sind alle nÃ¶tigen Daten vorhanden um mit der oben aufgefÃ¼hrten Formel den ZMP zu berechnen.

Bei diesem Modell ist zu beachten, dass es nur mit den selbst verursachten Kräften rechnet, und somit äußere Einflüsse nicht berücksichtigen kann. Es kann aber sehr gut z.B. durch Armbewegungen verursachte Änderungen des ZMP berechnen bzw. sogar vorraussagen.

Umsetzung auf dem Roboter

Auf dem Roboter wurde dieses Modell nicht eingesetzt, da die Beschleunigungen nicht im internen Modell berechnet wurden, wie zunächst einmal geplant war. Der bereits angelegte Code wurde dann später wieder entfernt.

3.3.4 Berechnung aus Drucksensoren in den Füßen

Eine weitere Möglichkeit zur Bestimmung des ZMP erhält man durch die Drucksensoren. Auch dieses Verfahren hat seine Nachteile, ist aber sonst sehr einfach.

Theorie

Die Sensoren befinden sich direkt in der X-Y-Ebene und messen nur Kräfte entlang der Z-Achse. Daher vereinfacht sich die Formel auf folgende zwei Gleichungen:

$$P_{ZMP,x} = \frac{\sum_i P_{i,x} \cdot m_i(g_z + a_{i,z})}{\sum_i m_i(g_z + a_{i,z})}$$

$$P_{ZMP,y} = \frac{\sum_i P_{i,y} \cdot m_i(g_z + a_{i,z})}{\sum_i m_i(g_z + a_{i,z})}$$

Der ZMP nach diesem Modell ergibt sich also, als das mit den Messwerten gewichtete Mittel der Sensorpositionen.
Zu beachten ist allerdings, dass dieses Modell nur verlässliche Werte liefert, wenn sich mindestens ein Fuß fest auf dem Boden befindet, da er nicht mehr messen kann, wenn sich der Roboter bereits in einer Kippbewegung befindet.

Umsetzung auf dem Roboter

Die Umsetzung auf dem Roboter erwies sich als nicht unproblematisch. Zum einen waren die in der Dokumentation angegebenen Positionen nicht korrekt, so dass diese aus einer Risszeichnung des Fußes per Hand bestimmt werden mussten. Weiterhin muss bei der Umsetzung beachtet werden, dass die Sensoren, im sich gerade nicht am Boden befindlichen Fuß, nicht berücksichtigt werden, da diese sonst auf Grund des Sensorrauschens das Ergebniss stark verfälschen.
Letztlich wurde dieses Verfahren nur im Simulator getestet, da die Sensoren auf dem realen Roboter, nicht kalibriert wurden. Im Simulator zeigten sich allerdings qualitativ plausible Werte, wobei die quantitative Verlässlichkeit fraglich ist, da es Probleme mit der Modellierung der Sensoren im Simulator gab. (vgl. hier zu 2.2.3)

3.3.5 Weitere Betrachtungen

Es wurden drei Verfahren untersucht, die alle mit akzeptablen Einschränkungen und Ungenauigkeiten den Zero Moment Point bestimmen, können. In der Praxis wurde aber aus den oben genannten Gründen nur ein Verfahren eingesetzt. Bessere Ergebnisse kann man erhalten, wenn man auf die Fusion mehrerer Verfahren zurückgreift, sobald diese implementiert sind. Auch ist eine Verfeinerung der Modelle möglich, wenn mehr verlässliche Information über den aktuellen Roboterzustand vorliegen. So ist eine Kompensation für das Kippen des Roboters gegenüber dem Boden sicherlich nützlich, wenn hier verlässliche Informationen vorliegen.

3.4 Drehmomente

Zur Verbesserung der Genauigkeit beim Ansteuern von Gelenkstellungen sind die auf das Gelenk wirkenden Drehmomente von entscheidender Bedeutung. Sind diese Drehmomente bekannt, kann eine mögliche Fehlpositionierung des Gelenks bereits im Voraus erkannt und ausgeglichen werden. Da von den Sensoren hierfür nur spärliche Informationen (die angefahrenen Gelenkwinkel) ausgelesen werden können, sind komplexe Berechnungen notwendig. Zunächst werden im Egomodell bereits die Positionen der Schwerpunkte aller Parts berechnet und gespeichert, ebenso ist die Masse eines jeden Parts bekannt. Mittels dieser Informationen berechnen die Methoden *calculateTreeCOM* und *getMassOfTree* der Klasse *Part* den Schwerpunkt und die Masse des gesamten Parts-Teilbaums, der mit dem aktuellen Part beginnt. Durch Festhalten von drei Positionen $(\vec{p_0}, \vec{p_1}, \vec{p_2})$ und Zeitpunkten (t_0, t_1, t_2) der entsprechenden Schwerpunkte lassen sich Geschwindigkeiten und eine Beschleunigung bestimmen:

$$\begin{aligned} \vec{v_{01}} &= \frac{\vec{p_1} - \vec{p_0}}{t_1 - t_0} \\ \vec{v_{12}} &= \frac{\vec{p_2} - \vec{p_1}}{t_2 - t_1} \\ \vec{a} &= \frac{\vec{v_{12}} - \vec{v_{01}}}{\frac{t_2 - t_0}{2}} - \vec{a_g} \end{aligned}$$

Die Erdbeschleunigung $\vec{a_g}$ lässt sich mittels der vom *OrientationInfoProvider* zur Verfügung gestellten Eigenschaft *downDirection* berücksichtigen.
Von der so ermittelten Beschleunigung ist nur die Komponente von Interesse, die parallel zur Drehrichtung $\vec{d_r}$ des Gelenkes steht. Um diese zu bestimmen, wird zuerst mit Hilfe der Ausrichtung der Drehachse $\vec{d_a}$ und der Position des Ankerpunktes des Gelenks $\vec{p_a}$ die Drehrichtung $\vec{d_r}$ ermittelt und anschließend damit der Betrag der Beschleunigung errechnet:

$$\begin{aligned} \vec{d_r} &= \vec{d_a} \times (\vec{p_2} - \vec{p_a}) \\ a &= \frac{\left\langle \vec{d_r}, \vec{a} \right\rangle}{\left| \vec{d_r} \right|} \end{aligned}$$

Zur endgültigen Berechnung des Drehmomentes M ist weiterhin die Bestimmung der Richtung $\vec{d_c}$ und Länge l_a des Hebelarms notwenig. Auch hier ist wieder nur die Komponente des Hebelarms, die senkrecht zur Drehachse steht von Interesse. Da die Beschleunigung im Schwerpunkt des Part wirkt, ist hierfür offensichtlich die Position des Ankerpunktes $\vec{p_a}$ des Part sowie der aktuelle Schwerpunkt $\vec{p_2}$ von Bedeutung. Es ergeben sich die Folgenden weiteren Gleichungen:

$$\begin{aligned} \vec{d_c} &= \vec{d_a} \times \vec{d_r} \\ l_a &= \frac{\left\langle (\vec{p_2} - \vec{p_a}), \vec{d_c} \right\rangle}{\left| \vec{d_c} \right|} \\ M &= F * l_a = m * a * l_a \end{aligned}$$

Diese Berechnungen berücksichtigen bisher leider nur die durch die eigene Bewegung und Massenträgheit auftretenden Kräfte und Drehmomente. Die Belastung, die aufgrund des Körpergewichtes des Roboters beim Stehen und Laufen zusätzlich auf die Beingelenke wirkt muss noch eingearbeitet werden (s. auch 4.1.5).

4 Bewegungssteuerung

4.1 Walking Engine

Grundsätzlich haben sich zwei Ansätze in der Wissenschaft besonders hervorgetan, um zweibeinigen Robotern das Laufen zu ermöglichen. Der anschaulichere Ansatz basiert auf festgelegten Trajektorien, die die Füße abfahren, um die menschliche Laufbewegung nachzuahmen. Verschiedene Parameter bestimmen dabei die Dauer eines Schrittes, die Bewegung des Oberkörpers und der Arme, usw. Die Bewegung wird dabei Hauptsächlich von diesen Parametern festgelegt [Kos06]. Evolutionäre Algorithmen sind daher ein Mittel, um optimale Laufparameter zu finden.
Ein anderer Ansatz basiert zur Generierung der Laufbewegung auf der Theorie des invertierenden Pendels und des ZMPs [CKU08]. Konzepte wie dynamisches Kicken, omnidirektionales Laufen, Sensorkontrollen usw. erfordern eine Umsetzung, die verschiedene physikalische Prinzipien beachten, um deterministisch eine stabile Laufbewegung entwerfen zu können.
Bisherige Konzepte wie die der festen Trajektorien, erlauben eine Optimierung durch selbstlernende Algorithmen, sind einfach umzusetzen, und sind durch vielerlei Parameter an die reale Welt anpassbar. Durch diese selbsterlernten Parameter können sie mit Ungenauigkeiten im Roboter (durch z.B. mechanische Ungenauigkeiten), der Dämpfung durch den Teppich und anderen physikalischen Effekten umgehen, da die Parameter in der realen Welt erlernt werden. Das ist aber auch gleichzeitig der entscheidende Nachteil, weshalb namhafte Teams zugeben, dass diese Art der Optimierung nicht für Einsätze, wie beim Robocup, geeignet sind, ohne sie zudem noch von Hand auf Stabilität hin zu optimieren. Allerdings ist es auch nicht ausreichend nur auf physikalische Prinzipen zu achten, Probleme wie ungenaue Gelenke können nur durch selbstlernen Algorithmen gut angegangen werden, was ein Nachteil des hier vorgestellten Ansatzes ist.
Das physikalische Grundprinzip beim Laufen ist das des Invertierenden Pendels [KKK+03]. Die Theorie des Invertierenden Pendels besagt, dass sich der Schwerpunkt des laufenden Systems auf einer bestimmten Bahn bewegen muss, damit der Oberkörper stabil bleibt. Dieser Schwerpunktverlauf wird bei Walking Engines mit festen Trajektorien z.B. durch Parameter erreicht, die die Federung des Teppichs nutzen. So funktionieren einmal erlernte Parameter nicht mehr auf anderen Untergründen (insbesondere anderen Teppichen), auch Verschleiss und sich erwärmende Servos können den Lauf so verändern, dass er nicht mehr stabil ist.
Um den oben genannten Erfordernissen gerecht zu werden, und unabhäniger von nicht steuerbaren Faktoren zu werden, wurden die im folgenden beschriebenen Module entwickelt, die zusammen die Walking Engine darstellen.

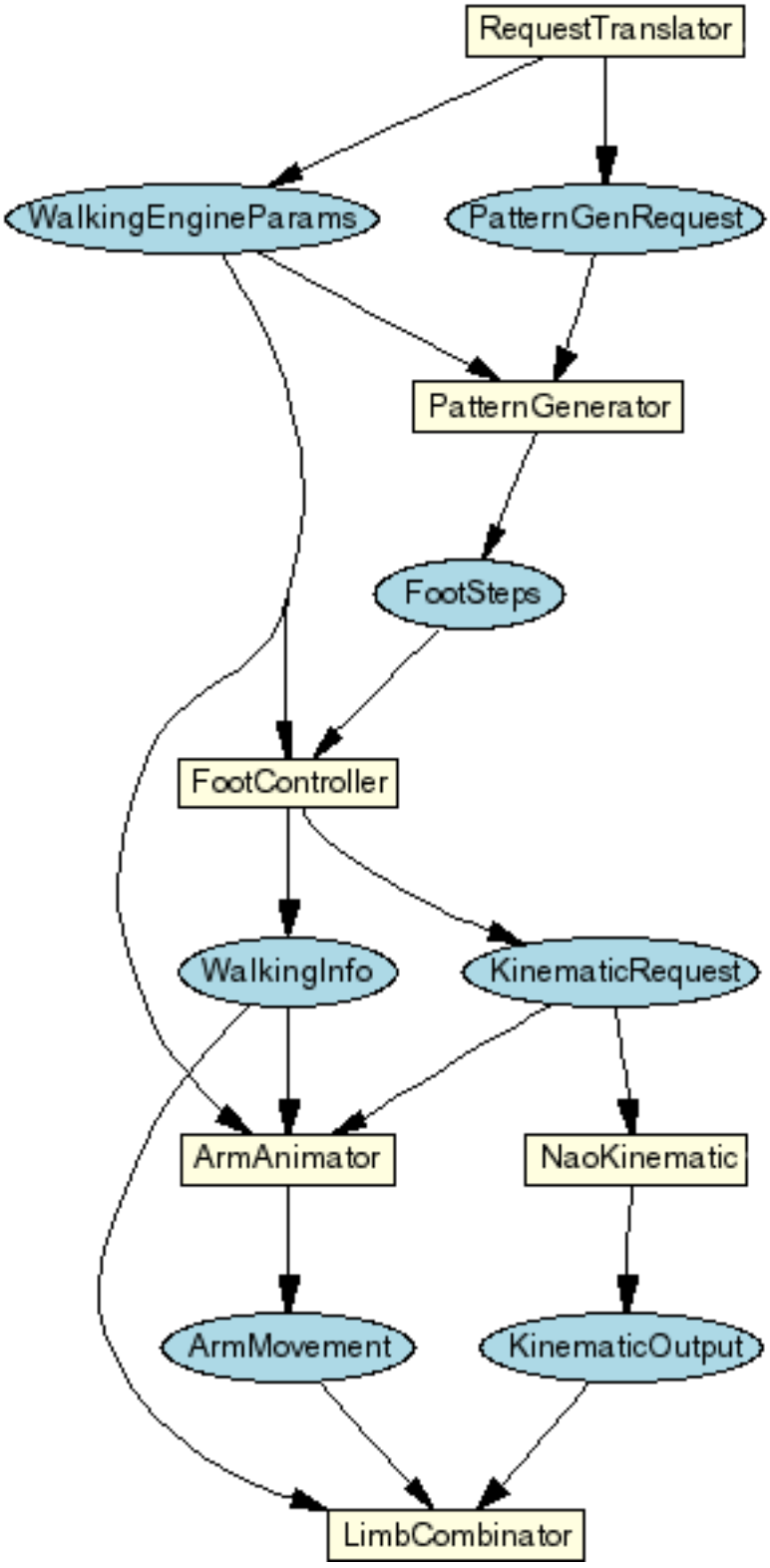


Abbildung 4.1: Struktur der Walking Engine

4.1.1 Struktur der Walking Engine

Abbildung 4.1 zeigt die derzeitige Struktur der Walking Engine innerhalb des Frameworks. Auf die Modulstruktur des Frameworks wird später noch im Detail eingegangen. An dieser Stelle wird nur abstrakt auf die Struktur eingegangen, die sich übersichtlich in einzelne Klassen unterteilt, die untereinander nur durch festgelegte Nachrichten kommunizieren.
Ein Walk beginnt mit einem Kommando, dem MotionRequest, an den RequestTranslator. Dort enthalten sind die gewünschte Geschwindigkeit (Translation x und y, sowie Rotation) und der Pitch (gewünscht Neigung des Oberkörpers), sowie die Information, dass der Roboter laufen soll, statt sonstige Bewegungen auszuführen. Der RequestTranslator lädt die Walking Parameter für die gesamte Engine und übersetzt das eingegangene Kommando in eine Anweisung für den PatternGenerator 4.1.2. Darin enthalten sind wieder die Geschwindigkeiten, die auf erlaubte Werte beschränkt wurden. Zudem ist eine Variable enthalten, die den neuen Zustand angibt, den der PatternGenerator annehmen soll. Der PatternGenerator ist eine Zustandsmaschine, und wechselt seine Zustände entweder automatisch, oder reagiert auf Kommandos vom RequestTranslator. Mehr dazu im nächsten Kapitel.
Der PatternGenerator nimmt die Anfrage des RequestTranslators entgegen und erzeugt daraus Fußstapfen in einem selbst erzeugten, nur in der Walking Engine verwendetem Weltkoordinatensystem, das seinen Ursprung dort hat, wo die Walking Engine gestartet worden ist. Die Fußstapfen kann man sich in diesem Koordinatensystem vorstellen, wie die Fußstapfen in einem unendlich großen Schneefeld. Diese sind die Positionen, die an den Footcontroller gesendet werden.
Der Footcontroller, besser gesagt, der darunter liegende ZMP/IP-Controller 4.1.3, berechnet aus den Fußstapfen in Weltkoordinaten die Fußpositionen im Roboterkoordinatensystem. Daraus ergeben sich einige Informationen über den aktuellen Zustand des Laufs, die in der Datenstruktur WalkingInfo gespeichert werden, darunter die Odometrie, die aktuellen Fußpositionen und die Information, ob man die Walking Engine beenden kann. Diese Informationen sind auch teilweise in RobotControlXP (siehe Kapitel 7.1.2) grafisch darstellbar.
Der Hauptstrang des Datenstroms teilt sich nun in zwei Teile. Zum einen in die Armbewegung, die direkt die Winkel an den LimbCombinator gibt, zum anderen gehen die Fußpositionen in die Inverse Kinematik (siehe Kapitel 4.1.4), die ebenfalls ihre Winkel an den LimbCombinator weitergibt.
Als Letztes fügt der LimbCombinator die Winkel, die er für die Arme und Beine bekommt, zusammen und erzeugt daraus den WalkingEngineOutput.

4.1.2 Der PatternGenerator

Wie bereits angesprochen, werden die Fußpositionen auf dem Boden vorgegeben. Zur Erzeugung der Fußstapfen werden zwei Zustandsmaschinen verwendet. Die eine beschreibt den aktuellen Zustand der Engine und die andere die aktuelle Phase des Laufs. Bei dem aktuellen Zustand der Engine muss zwischen den direkt auswählbaren Zuständen und

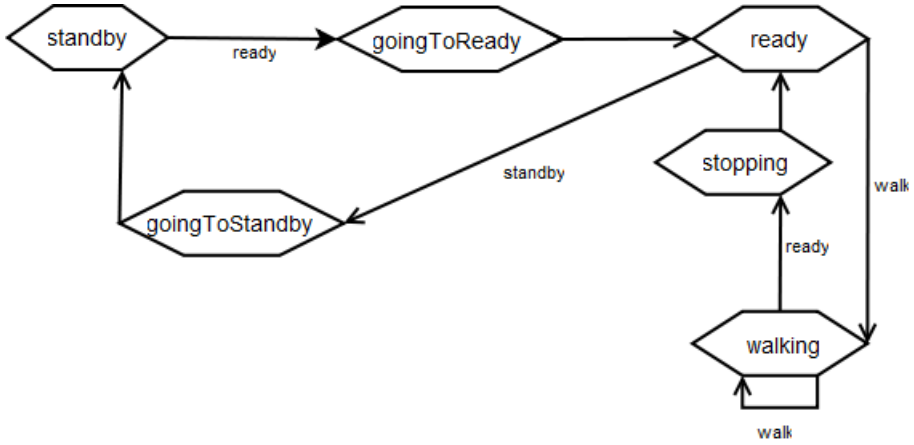


Abbildung 4.2: Zustandsmaschine des PatternGenerators

den internen Zuständen unterschieden werden. Intern kann der PatternGenerator die Zustände „standby“, „ready“, „walking“, „stopping“, „goingToReady“ und „goingToStandby“ annehmen. Abbildung 4.2 zeigt den Automaten. Zur Ansteuerung dienen die Anfragen vom RequestTranslator. In der Abbildung sind die Übergänge mit der Anforderung markiert, die zu diesem Übergang führen. Speziell zu erwähnen sei der „walk“ Übergang, der zu jeder Zeit im Zustand „walk“ und „ready“ angefordert werden kann, um die Geschwindigkeit zu ändern, bzw. zum Starten und Stoppen des Laufens.
Nicht zu verwechseln sind diese Anforderungen mit dem MotionRequest, der an den RequestTranslator geschickt wird, und der die einzige Möglichkeit ist, die Walking Engine von Außen zu steuern. Im MotionRequest wird grundsätzlich ein „walk“ angefordert, um die Engine zu nutzen. Die andere Möglichkeit im MotionRequest sind nur SpecialActions. Wählt man im MotionRequest „walk“ zusammen mit der Geschwindigkeit null wird die Engine in den Zustand „ready“ gehen. Eine Geschwindigkeit ungleich null versetzt die Engine in den Zustand „walk“.
Zur Erzeugung der Fußstapfen dient folgender Algorithmus. Da die Soll-Position des Schwerpunktes im Weltkoordinatensystem erst nach der Berechnung durch den ZMP/IP-Controllers bekannt ist, wird im PatternGenerator eine fiktive Position des Oberkörpers im Weltkoordinatensystem angenommen. Zur Berechnung dieser Position wird die vorgegebene Geschwindigkeit benutzt, wobei die Position in diskreten Abständen geupdated wird. Dazu wird der Lauf in 4 Phasen unterteilt:

1. erste single support[5]
2. erste double support
3. zweite single support

5 single support Phase ist die Phase beim Laufen, wo nur ein Fuß den Boden berührt. In der double support Phase berühren beide Füsse den Boden.

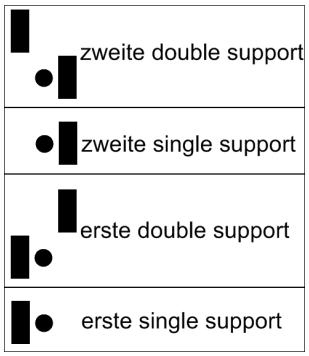


Abbildung 4.3: Prinzip der Fußstapfenerzeugung

4. zweite double support

Die Dauer der 4 Phasen zusammen ist in der entsprechenden Konfigurationsdatei einstellbar (t_{ges}). Zudem ist die „*double support ratio*“ in Prozent (r, z.B. $r = \frac{1}{4} = 25\%$) einstellbar. Die Länge der Phasen ist dann:

$$t_{ds} = \frac{r \cdot t_{ges}}{2}$$

$$t_{ss} = \frac{(1-r) \cdot t_{ges}}{2}$$

Abbildung 4.3 zeigt beispielhaft die 4 Phasen. Der Kreis stellt dabei die fiktive Position des Roboters im Weltkoordinatensystem dar. Die Position wird jeweils nach der ersten double support und nach der zweiten double support Phase entsprechend der gewählten Geschwindigkeit verändert. Die Fußpositionen werden dann relativ, in der Form eines Offsets, zu der Roboterposition festgelegt. In der ersten single support Phase ist dabei der linke Fuß am Boden. In der beispielhaften Abbildung wäre der Offset des linken Fußes in der ersten single support Phase $x_{\text{offset}} = 0$ und $y_{\text{offset}} > 0$.
Die Abbildung ist deswegen nur als ein Beispiel anzusehen, da das Update der Roboterposition und der Offset der Füße innerhalb des Codes frei einstellbar ist. In der Klasse `PatternGenerator` wird in der Funktion `setStepLength()` die zweidimensionale Datenstruktur `footModifier[][]` mit den entsprechenden Werten gefüllt. Dieses Verfahren macht den PatternGenerator flexibel und leicht anpassbar.

4.1.3 Der ZMP/IP-Controller

Der ZMP/IP-Controller bildet das Herzstück der gesamten Walking Engine. Die Fußstapfen vom PatternGenerator werden zunächst in Soll-ZMPs verwandelt. Dabei liegt der Soll-ZMP (auch Referenz-ZMP genannt) in einer single support Phase in der Mitte des Fußes, und wird in der doubble support Phase von einem Fuß zum anderen per B-Spline-Funktion interpoliert[PBP02].
Nachdem der Soll-ZMP feststeht wird er dem eigentlichen Controller übergeben. Dabei handelt es sich um einen beobachterbasierten Preview-Controller, der anhand der

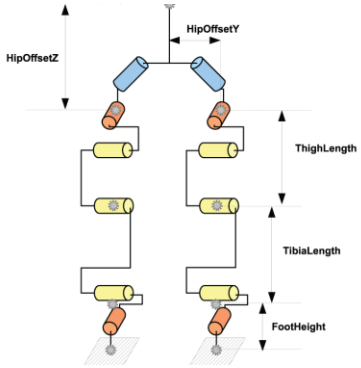


Abbildung 4.4: Logischer Aufbau der Beine des Nao. Der Stern stellt den Punkt dar, in welchem sich die Achsen schneiden, jeweils für Hüfte, Knie und Fuß.

Theorie des invertierten Pendels [KKK+03] aus den ZMPs eine Position für den Schwerpunkt errechnet. Zudem werden vom Controller Sensordaten verwendet. Derzeit ist das der ZMP, der aus dem Beschleunigungssensor berechnet wird. Details zu dem Verfahren finden sich in [CKU08].
Nachdem die Soll-Position des Schwerpunktes in Weltkoordinaten errechnet wurde, wird er von den Soll-Fußpositionen, die vom PatternGenerator übergeben wurden, subtrahiert und passend rotiert. Danach wird die Ist-Schwerpunkt-Position im Roboterkoordinatensystem, die vom EgoModell geliefert wird, addiert, und man erhält die Fußpositionen im Roboterkoordinatensystem, die nun an die Kinematik übergeben werden.

4.1.4 Die Inverse Kinematik

Der Aufbau der Beine des Nao hat eine Besonderheit, welche in Abbildung 4.4 zu erkennen ist. Dabei sind die beiden obersten Gelenke, nur kombiniert drehbar. Dieses bedeutet, man kann für beide eingezeichneten Gelenke nur einen Winkel gleichzeitig einstellen (physisch handelt es sich dabei um ein Gelenk). Das daraus resultierende Problem ist, dass man nur für einen Fuß die z-Rotation (bezüglich der Koordinatensystem des Fußes) angeben kann. Beim anderen Fuß ist damit dessen z-Rotation in seinem lokalen Koordinatensystem bereits festgelegt. Daher wird die inverse Kinematik für den Nao zweigeteilt. Der eine Teil berechnet, in geschlossener Form, die Winkel für ein Bein (links oder rechts) für eine beliebige Position und Rotation im Raum. Der andere Teil der Kinematik bekommt dann als Vorgabe die Position für x, y und z, die Rotation allerdings nur für x und y, nicht aber für z. Dafür erhält dieser Teil der Kinematik den Winkel von dem kombinierten Gelenk vorgegeben und errechnet anhand dieser Daten die Winkel. Die folgende Formel verdeutlich dieses Vorgehen. f_{norm} steht dabei für die normale inverse Kinematik, und $f_{special}$ für die spezielle inverse Kinematik, die jeweils die Winkel für die Gelenke 0 bis 5 liefern:

$$(\varphi_0, ..., \varphi_5) = f_{norm}\,(t_x, t_y, t_z, r_x, r_y, r_z, s)$$
$$(\varphi'_0, ..., \varphi'_5) = f_{special}\,(t'_x, t'_y, t'_z, r'_x, r'_y, \varphi_0, 1-s)$$

$$s = \begin{cases} 0 & \text{falls rechtes Bein} \\ 1 & \text{sonst} \end{cases}$$

Die Entscheidung, wie s gewählt wird, hängt von der Position des Schwerpunktes ab. Näherungsweise kann man davon ausgehen, dass der Fuß, der (im Roboterkoordinatensystem) näher am Schwerpunkt ist, auch mehr Gewicht tragen muss, und damit nicht den Fehler in der z-Rotation haben sollte. Daher wird s während des Laufens ständig anhand dieses Kriteriums neu ausgewählt.

4.1.5 Damping Controller

Ziel des Damping Controllers ist es, die idealisierten Annahmen der inversen Kinematik um das reale Nachgeben der Beine bei Lasteinwirkung zu erweitern. Diese Ungenauigkeiten können mehrere verschieden Ursachen haben. Zum einen können sich die Bauteile verformen, zum anderen geben die Servos je nach Krafteinwirkung nach. Zusätzlich haben die Gelenke selbst ein gewisses Spiel, das schon bei minimaler Kraft die angefahrene Position verfälscht.
Eine grobe Abschätzung der zu erwartenden Verformungen der einzelnen Teile ergab, dass diese Ungenauigkeit praktisch zu vernachlässigen ist, in jedem Fall aber deutlich unter dem durch die Gelenke verursachten Fehler liegt. Der Dampingcontroller beachtet daher in der jetzigen Version nur die Gelenke.
Die Arbeitsweise des Controllers besteht aus zwei Stufen. Zuerst wird für jedes Gelenk das Drehmoment ermittelt, das auf diesen bei der Ansteuerung im aktuellen Frame zu erwarten ist. Da keine Sensoren vorhanden sind, die dieses direkt erfassen könnten, wird das Drehmoment indirekt aus der aktuellen Ausrichtung im Raum (Orientation Info), der Kinematik der Beine und der Kraft auf die einzelnen Fußdrucksensoren des jeweiligen Beines berechnet.
Als nächstes wird aus der auf das Gelenk wirkenden Kraft der Winkel berechnet, den das Gelenk von seiner Sollposition abweichen wird. Die momentane Modellierung geht davon aus, dass dieser Winkel eine Funktion des wirkenden Drehmomentes und des Motortyps ist. Um diesen ermittelten Winkel wird dann der von der inversen Kinematik gelieferte Wert korrigiert.
Die Entwicklung des Damping Controllers wurde begonnen, bevor ein realer Nao zur Verfügung stand und war durch die Erfahrungen mit den zuvor verwendeten DohBots ([PG407]) motiviert, deren Gelenke eine hohe Abweichung zeigten. Der Damping Controller wurde soweit angelegt, um das beschriebene Vorgehen zu ermöglichen, kam aber nie zum Einsatz. Die Gründe hierfür liegen darin, dass die benötigten Werte der Fußdrucksensoren (s. 2.2.3) auf dem Nao nicht zur Verfügung standen und darin, dass das Ermitteln der zu erwartenden Abweichung der einzelnen Gelenke eine sehr aufwändige Aufgabe ist. Zudem fahren die Servos der Naos die Positionen auch unter Last deutlich präziser an als die der DohBots, so dass die Verwendung des Controllers als nicht dringend erschien.

4.1.6 Armbewegungen

Die Armbewegungen dienen dazu, das beim Laufen auftretende Drehmoment um die z-Achse zu verringern. Dies entsteht dadurch, dass die Schwerpunkte der Beine parallel zur x-Achse bewegt werden, zu dieser aber nicht symmetrisch sind.
Die von den Armen ausgeführte Bewegung ist sehr einfach aufgebaut. Die Ellbogen bleiben konstant in 0-Stellung (B.1) und die Arme werden mit den ShoulderRoll-Joints auf einen konstanten Abstand zum Körper gebracht, die einzige Bewegung kommt aus den ShoulderPitch Servos. Diese fahren Winkel an, die proportional zur x-Position des gegenüberliegenden Fußes sind. Der Winkel ist mit einer Proportionalitätskonstante skalierbar und in jeder Richtung bei einem Maximalwert begrenzt.
Ein Problem dieses einfachen Ansatzes liegt darin, dass die Schrittweite von der Laufgeschwindigkeit abhängt. Dieses macht die Armbewegungen gerade bei schnellerem Laufen ruckhaft. Hier wäre eine Anpassung der Konstante an die aktuelle Geschwindigkeit denkbar.
Generell ist bei den Naos, mindestens bis Version 2, auf zu starke Bewegungen in den Schultergelenken zu verzichten, da diese die Kabelverbindungen zu den Armboards beschädigen.

4.2 Special Action

Special Actions bezeichnen statische Bewegungen wie das Aufstehen nachdem der Roboter umgefallen ist oder Bewegungen zum Schießen des Balls. Diese Bewegungen oder auch Aktionen werden immer zu einem fest definierten Zeitpunkt aufgerufen und können dann jedes mal statisch nach Erreichen eines bestimmten Zustandes ausgelöst werden. Zum Beispiel kann der Ball mit einem fest definierten Bewegungsablauf geschossen werden, nachdem die Positionierung am Ball und die Ausrichtung zum Tor abgeschlossen wurden. Der Nachteil solcher fest definierten Bewegungsabläufe besteht darin, dass diese nur durch vorher festgelegte Transitionen unterbrochen werden können. Solche Transitionen sind jedoch zu starr, wenn während einer Schussbewegung der Ball verschoben wird oder der Roboter zu fallen beginnt. In solchen Fällen ist es wünschenswert die Aktion zu einem beliebigen Zeitpunkt zu unterbrechen und sich erneut am Ball auszurichten bzw. sich vor einem Sturz zu bewahren.Im Folgenden soll die Entwicklung von Special Action Schritt für Schritt beschrieben werden.

4.2.1 Entwicklungsprozess

Die Erstellung einer neuen Special Action und die Bekanntmachung dieser neuen Special Action im Framework läuft in 5 Entwicklungsschritten ab. Als erstes muss eine Datei mit dem Namen [Name der Special Action].mof im Ordner *mof* im Verzeichnis *Src\Modules\MotionControl* des Projektes angelegt werden. In dieser Datei werden im wesentlichen die Gelenkwinkel für jedes einzelne Gelenk des Nao definiert, die während der Bewegung angesteuert werden sollen. In Abbildung 4.5 ist der Inhalt einer mof-Datei abgebildet. Nachdem die mof-Datei erstellt ist, muss der Name der Special Action in die

pitch.mof - Editor

Datei Bearbeiten Format Ansicht ?

```
motion_id = pitch
"auf dem Nao schon getestet
label start

"HP HT   AL1 AL2 AL3 AL4 AR1 AR2 AR3 AR4 LL0 LL1 LL2 LL3 LL4 LL5 LR0 LR1 LR2 LR3 LR4 LR5 Int Dur
 *  *    -90 30  0   0   -90 30  0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   500

 *  *    -90 30  0   0   -90 30  0   0   0   0   -30 0   10  0   0   0   -30 0   10  0   1   1000
label stay
 *  *    -90 30  0   0   -90 30  0   0   0   0   -30 0   10  0   0   0   -30 0   10  0   1   500
transition pitch pitch stay
 *  *    -90 30  0   0   -90 30  0   0   0   0   0   0   0   0   0   0   0   0   0   0   1   1000

transition allMotions extern start
```

Abbildung 4.5: pitch.mof

extern.mof eingetragen werden. Die Datei extern.mof spielt eine besondere Rolle unter den mof-Dateien. In ihr werden alle Special Action deklariert. Neue Special Action werden immer in der Special Action Liste unten eingefügt mit der Syntax *transition [Special Action Name] [Special Action Name] start*, jedoch vor der *transition extern extern start* Anweisung. Die Anweisung *transition extern extern start* muss die letzte Anweisung in der Datei extern.mof sein. Im 3. Schritt wird die neu erstellte Special Action in die *odometry.cfg* im Verzeichnis *Config* eingetragen. Durch die odometry.cfg sollen dabei Bewegungen von Special Actions berücksichtigt werden, die eine Änderung der aktuellen Position bewirken. Diese Positionsänderung kann einmal durch die Integration über die Dauer der Special Action oder aber als fester Offset für die Special Action selbst angegeben werden. Ist die Special Action der odometry.cfg hinzugefügt, müssen noch Datenstrukturen der *SpecialActionRequest.h* im Framework angepasst werden. Dazu muss der Enumeration *SpecialActionID* der Name der Special Action hinzugefügt werden. Des Weiteren ist in der Methode *static const char* getSpecialActionName(SpecialActionID id)* eine zusätzliche case Anweisung zu ergänzen. Nachdem alle Schritte durchgeführt sind, muss als letzter Schritt das Projekt URC und das Projekt Special Action neu kompiliert werden. Bei der Kompilierung des Projektes Special Action wird im config-Verzeichnis des Framework die Datei *specialaction.dat* erzeugt, die nun alle Special Actions enthält und im Verzeichnis *Config\Robots\Naoxx* auf der realen Roboterhardware platziert werden kann. Damit steht den Robotern nun die neu entwickelte Special Action zur Verfügung. Eine Übersicht der während der PG 521 entwickelten Special Action ist im Anhang C zu finden.

5 Framework

Als Framework wurde das ursprünglich vom GermanTeam entwickelte Framework verwendet und angepasst bzw. erweitert. Das Framework ist speziell für die Verwendung im Roboterfußball entwickelt und in der verwendeten Version auf Humanoide ausgelegt. Die ursprüngliche Version wurde zuvor erfolgreich in der *4-legged League* auf den *AIBO* Roboterhunden eingesetzt.

5.1 Einleitung

Die zwei grundlegenden Konzepte des Frameworks sind *Representations* und *Provider*. Eine Representation ist eine Datenstruktur, die dazu dient, Informationen zwischen verschiedenen Stufen der Datenverarbeitung weiterzureichen. Das eigentliche Berechnen dieser Informationen geschieht in den Providern. Diese sind Klassen, die die Informationen für eine oder auch mehrere Representations berechnen können und dafür wiederum eine bestimmte Menge an Representations voraussetzen. Diese Abhängigkeiten bestimmen, in welcher Reihenfolge die Provider ausgeführt werden müssen, damit jeder mit den Informationen des aktuellen Frames arbeiten kann. Hierfür ist eine topologische Sortierung der Provider notwendig, was beim Start des Frameworks vom *Modulmanager* erledigt wird. Es ist darauf zu achten, dass keine zirkulären Abhängigkeiten entstehen, da dies die korrekte Sortierung verhindert. Welcher Provider zur Berechnung einer bestimmten Representation tatsächlich benutzt wird, kann über die Konfigurationsdatei *modules.cfg* eingestellt werden, die ebenfalls vom *Modulmanager* ausgewertet wird.
Die Provider lassen sich dabei in zwei Mengen aufteilen, dies sind *Cognition* und *Motion*. Diese Trennung zeichnet sich im Framework dadurch ab, dass die Provider aus beiden Mengen in zwei verschiedenen Threads ausgeführt werden. Dies ermöglicht es, die Daten in beiden Mengen mit unterschiedlicher Frequenz zu aktualisieren. Dies wird benutzt, um insbesondere Daten in der Bildverarbeitung oder der Verhaltenssteuerung seltener zu berechnen und so Rechenleistung zu sparen, während zeitkritische Daten wie die der Bewegungssteuerung mit einer höheren Frequenz berechnet werden. Der Modulmanager verfolgt dabei, wie lang der zeitliche Abstand zwischen zwei Ausführungen war und passt darauf basierend die Wartezeit bis zur nächsten Ausführung an, um so auf Dauer die gewünschte Frequenz möglichst genau einzuhalten. Hier ist darauf zu achten, ob es Provider gibt, die unter Umständen für einzelne Frames eine erheblich längere Laufzeit haben als die angestrebte Zeit zwischen zwei Ausführungen. Dies stört das Timing massiv, da zum einen die Daten für diese Frames zu spät verfügbar sind, zum anderen aber die folgenden Frames zu schnell aufgerufen werden, da der *Modulmanager* versucht, den Rückstand zu kompensieren.
Der Cognition Thread erledigt vor allem das Image Processing und die darauf basierenden Modellierungen bspw. für den Ball oder die Tore. Diese dienen dann als Ein-

gabesymbole für die Verhaltensmodellierung (s. 6). Die Aufruffrequenz ist in der Regel deutlich niedriger gewählt als beim Motion Thread. Da ein Hauptteil der Daten aus der Bildverarbeitung gewonnen wird, stellt hier insbesondere die maximale Framerate der Kamera (30 FPS) eine Obergrenze dar.
Der Motion Thread behandelt alle Aspekte der vom Roboter auszuführenden Bewegungen. Dies kann entweder eine Laufbewegung oder eine Special Action sein (bestimmt durch den aktuellen *MotionRequest*). Die Bewegungen werden mit den entsprechenden Gruppen von Modulen generiert, für das Laufen sind dies die *DortmundWalkingEngine* und die inverse Kinematik, für eine Special Action liest das Modul *SpecialAction* einen festen Bewegungsablauf ein (4.2). Die aktuelle Umsetzung des Frameworks bedingt, dass immer für beide Möglichkeiten JointRequests berechnet werden. Welche dann tatsächlich über den *(Nao)JointDataProvider* an den Roboter weitergegeben werden bestimmt anschließend das Modul *MotionCombinator* anhand des MotionRequest.
Der Motion-Thread läuft normalerweise mit einer Frequenz von 50 Hz.

5.2 Software auf dem Nao

Als Betriebssystem läuft auf dem Nao ein Linuxkernel der Reihe 2.6, der mittels eines Patches um Echzeitfähigkeit erweitert wurde ([Ald08b]). Ein großer Teil der im Nao enthaltenen Hardware sind Standardkomponenten, die direkt vom Kernel unterstützt werden, wie beispielsweise die Kamera (s. 5.4.3) oder die Soundausgabe (s. 5.4.5).
Zur Ansteuerung der Roboterhardware, die nicht vom Kernel verwaltet wird dient die Software *NaoQi* von Aldebaran.
NaoQi ist modular aufgebaut und ermöglicht es, neue Module entweder direkt als Shared-Library zu integrieren oder als Broker, ein externes Programm, das über eine SOAP Verbindung mit *NaoQi* kommuniziert, zu benutzen. Die einzelnen Module ermöglichen den Zugriff auf die Hardware auf unterschiedlichen Abstraktionsniveaus. Auf höherer Ebene wird z.B. eine Laufbewegung angeboten, die direkt die Angabe der gewünschten Laufgeschwindigkeit und Entfernung zulässt, jedoch besteht auch die Möglichkeit eines relativ direkten Zugriffs auf die Hardware ([Ald08b]).
Die Kommunikation mit der Hardware erfolgt dabei immer über die Hardware-Komponente *DCM* (s. 5.4), entweder direkt über das NaoQi-Modul *devicecommunicationmanager* oder über ein anderes Modul, welches den Zugriff auf einem höheren Abstaktionsniveau erlaubt, wie bspw. *ALMotion*.

5.3 Portierung

5.3.1 Tor-Modellierung

Für das Verhalten des Roboter werden einige Daten über die Beschaffenheit des Spielfelds, besonders über die nähere Umgebung des Roboters benötigt. Dazu zählt unter anderem die Erkennung der Tore. Sowohl das eigene, als auch das gegnerische Tor, muss als Tor erkannt werden und dann entsprechend der Farbe klassifiziert werden. Hierfür wird auf das Imageprocessing zurückgegriffen, da alle Dinge auf dem Spielfeld nur über

die Kamera erkannt werden können und nicht a priori vorgegeben sind. Der Imageprocessor liefert hierfür über die Bilderkennung die Rohdaten, bestehend aus gesehenen Torpfosten und deren Eigenschaften, wie *linker Pfosten*, *rechter Pfosten*, *gegnerisch* oder *eigener*.
Um aus diesen Rohdaten verwendbare Daten zu erstellen, wird auf eine alte Tormodellierung zurückgegriffen. Diese bestand *pro Tor* aus dem *Winkel vom Roboter zur Mitte des freien Teils* des erkannten Tores. Der *Größe des Öffnungswinkels des freien Teils* des Tores und einem *Zeitstempel*, wann zuletzt dieser freie Winkel des Tores gesehen wurde. Immer wenn beide Torpfosten eines Tores erkannt wurden, wurden die genannten Daten aktualisiert. Wurde nur ein Torpfosten erkannt, konnte diese Modellierung nicht aktualisiert werden.
Diese Tor-Modellierung wurde ergänzt und verbessert, indem nun zunächst die klar erkannten Goal-Post-Percepts verarbeitet werden, die sowohl in der *Farbe*, *eigenes* oder *gegnerisches* Tor, als auch in der Orientierung, *linker* oder *rechter* Pfosten, eindeutig erkannt werden. Sind sowohl *rechter*, als auch *linker* Pfosten eines Tores erkannt, so wird der *Winkel zum Tor*, der *Öffnungswinkel* und die *Distanz zum Tor* berechnet. Für den Winkel wird der Mittelpunkt zwischen den Pfosten als Bezugspunkt verarbeitet. Um die Genauigkeit der Erkennung zu dokumentieren wird die Zeit des letztmalig eindeutig erkannten Pfostens gespeichert. Hierbei muss nicht beachtet werden, wann zum letzten Mal beide Pfosten gesehen wurden, wenn ein linker oder rechter Pfosten eindeutig identifiziert wird.
Falls nur einer der beiden Torpfosten richtig, *Farbe* und *rechts* bzw. *links*, erkannt wurde und vorher nicht beide Pfosten erkannt wurden, konnte mit der alten Modellierung kein bestehendes Tor aktualisiert werden. Daher wurde für jeden der vier Torpfosten die Repräsentation im Obstaclelocator erweitert, um die *Winkel zu den einzelnen Pfosten* und die *Zeit wann der einzelne Pfosten zuletzt gesehen wurde*. Wird nun ein Pfosten einzeln erkannt, wird der *Winkel zum Tor* und der *Winkel zum Pfosten* auf diesen Pfosten gesetzt. Außerdem werden die *Zeiten* für beide Werte aktualisiert. Von der Möglichkeit den Winkel zum Tor zwischen dem gesehenen Pfosten und dem Bildrand zu setzen wird abgesehen. Zwar stimmt die Richtung für die Toröffnung, aber z.B. ein anderer Roboter könnte den zweiten Pfosten abdecken, der läge damit im toten Winkel und der berechnete Winkel zwischen Pfosten und Bildrand wäre nicht korrekt. Aus dem selben Grund wird auch der Öffnungswinkel nicht auf den Winkel zwischen Pfosten und Bildrand gesetzt. Hier wird dem Verhalten die Möglichkeit überlassen zu entscheiden, auf welchen Winkel neben dem Pfosten der Roboter zum Schuss ausgerichtet werden soll.
Alle gesammelten Daten über die *Winkel zu beiden Toren*, die *Öffnungswinkel* und die *Winkel zu den einzelnen Pfosten* werden über die *Sichtungszeiten* mit einem Alter versehen. Zu alte Werte werden herrausgestrichen und die Winkel auf den Ausgangswert 0 zurückgesetzt. Außerdem wird auf alle Werte noch die Odometrie-Verschiebung und -Rotation angewand, um bei einer Drehung des Roboters die Werte der Winkel stets aktuell zu halten.

5.4 Anbindung an NaoQi

Um das bestehende Framework einfach zu integrieren, wurde ein Wrapper geschrieben, so dass das Framework als *NaoQi-Modul* eingebunden werden kann. Im Regelbetrieb wird das Framework als Library direkt benutzt und alle anderen Module bis auf den *Device Communication Manager* werden deaktiviert, da das Framework alles nötige enthält. Ein Betrieb als Broker ist ebenfalls möglich, aber es können dann nicht alle Teile getestet werden, da z.B. beim Laufen zu große Verzögerungen durch die Kommunikation auftreten.

5.4.1 Joints

Es wurden zwei verschiedene Möglichkeiten der Ansteuerung der Servos implementiert, dies sind die Provider *NaoJointControl* und *NaoDirectJointControl*.

NaoJointControl

Diese Art der Ansteuerung setzt voraus, dass das NaoQi-Modul *Motion* geladen ist. Dann ist es möglich, alle zu setzenden Winkel in Form einer Liste an NaoQi zu übergeben, welches diese dann innerhalb einer vorgegebenen Zeit anfährt. Ein Problem besteht darin, dass NaoQi ermittelt, ob die Winkel in der angegebenen Zeit überhaupt erreichbar sind und den kompletten Satz Winkel verwirft, falls mindestens einer davon nicht möglich ist. Es ist daher schon beim Zusammenstellen der Anforderung darauf zu achten, dass die Zielwinkel nicht mehr als einen bestimmten Betrag vom aktuellen Winkel abweichen.

NaoDirectJointControl

Ab der NaoQi-Version 0.1.13 ist es möglich, direkt den *Device Communication Manager* (DCM) anzusprechen ([Ald08a]). Dies ist ein Modul, das eine Zwischenschicht zwischen NaoQi und den eigentlichen Sensoren und Servos darstellt. Der Vorteil gegenüber dem vorher verwendeten Provider NaoJointControl besteht darin, dass auf das Laden des Motion-Moduls in NaoQi verzichtet werden kann. Ebenfalls ist hiermit eine bessere Kontrolle über die ausgeführten Bewegungen möglich.
Der DCM erlaubt es insbesondere, die Zielpositionen für alle oder auch einzelne Joints mit einer absoluten Zeitangabe zu versehen.

5.4.2 Sensors

NaoSensorDataProvider

Ähnlich wie beim NaoJointControl wird zum Betrieb des Moduls, das die Sensorwerte bereitstellt, das NaoQi-Modul *Sensors* benötigt. Der NaoSensorDataProvider selbst fragt im Wesentlichen die Werte des Beschleunigungssensors, des Gyroskopen, die Positionen und Temperatur aller Gelenke und die aktuelle Batteriespannung ab. Diese werden dann an entsprechende *Representations* des Frameworks weitergereicht. Bei verschiedenen Laufzeittests hat sich jedoch herausgestellt, dass die Aufrufe von `update()`

des NaoSensorDataProvider im Schnitt etwa $14ms$ brauchen. Dies ließ sich nur dadurch erklären, dass die Sensorwerte bei dieser Art der Implementierung den Umweg über NaoQi nehmen müssen und damit die Laufzeit enorm erhöht wird. Dieses grundsätzliche Problem hatte die Implementierung des NaoDirectSensorDataProviders zur Folge. Der NaoSensorDataProvider wird nicht mehr benutzt.

NaoDirectSensorDataProvider

Im Unterschied zum NaoSensorDataProvider benötigt der NaoDirectSensorDataProvider kein zusätzliches NaoQi-Modul mehr. Die API für den Sensorzugriff dieses Moduls wird in *ALMemoryFastAccess.h* definiert und implementiert den Zugriff auf die Sensorwerte direkt über den DCM des Roboters. Durch diesen neuen Ansatz ließ sich die Laufzeit für das Auslesen der Sensorwerte von ca. $14ms$ auf unter $1ms$ verringern.
Der NaoDirectSensorDataProvider liest die selben Werte der Sensoren des Roboters wie der NaoSensorDataProvider, enthält aber zusätzlich noch Filterimplementierungen für einzelne Sensorwerte wie etwa die gemessenen Beschleunigungen.

5.4.3 CameraProvider

Die Kameraanbindung besteht aus zwei Teilen. Ein Teil liefert die Kamerabilder für das Framework auf dem Nao, der andere Teil ist für die Ausführung auf einem entfernten Windows-Rechner gedacht. Die beiden Teile unterscheiden sich im Auslesen der Bilder.

CameraProvider auf dem Nao

Auf dem Nao wird NaoQi umgangen, und direkt aus dem Stream `/dev/video0` gelesen, unter Verwendung des Video4Linux-Treibers, der auf dem Nao vorinstalliert ist. Zum Lesen wird die `read()` Methode von Linux verwendet, die allerdings einen Nachteil hat. Die Daten werden dabei aus dem Puffer in einen lokal angelegten Speichern kopiert. Es wäre allerdings auch möglich, dem Treiber einen Pointer auf einen Speicherbereich zu übergeben, in den der Treiber die Daten von der Kamera direkt schreibt. Leider ist diese und eine ähnliche andere Methode nicht einsetzbar, da sich in dem Video4Linux-Treiber von AMD, der auf dem Nao eingesetzt wird, verschiedene Bugs befinden, die den Kernel abstürzen lassen. Da der Sourcecode für den Treiber nicht zu Verfügung gestellt wurde, können die Fehler nur durch ein Update von Aldebaran behoben werden.
Neben dem Auslesen der Bilddaten kann die Anbindung auch die Kameraeinstellungen setzen, darunter z.B. „exposure“, „red/white balance“ und „gain“. Diese Daten werden im Normalfall aus der entsprechenden Konfigurationsdatei gelesen.

CameraProvider auf einem entfernten Windows-Rechner

Eine sinnvolle Methode zum Debuggen ist die Möglichkeit, das Framework auf dem eigenen Windows-Rechner auszuführen und sich über ein Netzwerk mit dem Nao zu verbin-

den. Um so auch Kamerabilder vom echten Nao erhalten zu können gibt es Abschnitte im CameraProvider, die das Bild nicht von `/dev/video0` lesen, sondern NaoQi-Methoden zum Auslesen verwenden. Das Auslesen mit NaoQi-Funktionen ist zwar langsamer, da das Bild, unter anderem, konvertiert wird, ist dafür aber ohne großen Programmieraufwand auch über Netzwerk möglich, da NaoQi sich bereits um die Kommunikation der Windows-Komponente mit dem Nao kümmert. Natürlich lassen sich auch hier Einstellungen der Kamera setzen.

5.4.4 LED Output

Der Nao besitzt acht RGB LEDs (Vollfarb-LEDs) in jedem Auge und jeweils eine RGB LED pro Fuß und im Torsobutton. Dazu kommen zehn blaue LEDs in jedem Ohr. Die LEDs können und sollten als Debugmöglichkeit dienen. Die Ohr-LEDs, welche ringförmig an den Ohren angebracht sind, könnten zur Signalisierung des WLAN-Empfangs oder auch des Akku-Ladestandes verwendet werden.
Die derzeitige Version des Naos bzw. seiner Software lässt ein Ansteuern der LEDs im Kopf nicht zu. Der I2C-Bus, der die LEDs ansteuert, kann zwar durch eine Konfigurationsdatei eingeschaltet werden, jedoch rät Aldebaran davon ab. Ein Fehlverhalten in der I2C-Buskommunikation sorgt dafür, dass der Bus scheinbar einfriert oder falsche Pakete verschickt. Dies wäre in Kauf zu nehmen, wenn die Problematik lediglich die LEDs beträfe. Da aber die Kamera ebenfalls von diesen Fehlern betroffen ist, wird derzeit auf alle LEDs im Kopf verzichtet. Ausserdem können auch in seltenen Fällen die restlichen Bussysteme von den Fehlern betroffen sein, so dass der Nao gar nicht mehr steuerbar ist. Ist dies einmal passiert, muss der Nao neu gebootet werden. Für den Spielbetrieb ist solch ein fehleranfälliges Verhalten nicht akzeptabel. So ist eine Einschränkung auf die Fuß und Torso-LEDs gerechtfertigt. Diese müssen auch genutzt werden, da die Spielregeln ein bestimmtes Verhalten dieser LEDs vorschreiben.
Die LEDs werden von NaoQi genutzt um diverse Einstellungen zu visualisieren. Dieses Verhalten wird durch das NaoQi-Modul ALLeds ausgelöst, welches entweder nicht gestartet oder schnellst möglich beendet werden sollte. ALLeds wird zusätzlich noch durch ALInit gestartet. Entfernt man aus der AUTOLOAD.INI sowohl ALLeds als auch ALInit hat man mit dem DCM die komplette Kontrolle über die LEDs. Jede RGB LED besitzt jeweils einen Actuator für rot, grün und blau. Diese werden analog zu Joint-Actuator angesteuert. Es wäre so zwar möglich das komplette Farbspektrum der LEDs zu nutzen, jedoch macht dies nur wenig Sinn, da das menschliche Auge zwischen vielen Farben den Unterschied nicht erkennen würde. Im Framework gibt es für jeden Farbkanal einer LED drei verschiedene Zustände: an, aus und blinkend. Somit kann man pro RGB LED $3^3 = 27$ unterschiedliche Farb-Zustandskombinationen wählen.
Analog zur Anbindung Framework-Hardware ist die Anbindung an das Behavior gestaltet. Der Behaviorentwickler kann jeden Farbkanal einer LED ein/aus oder blinkend schalten. Die LEDs sind neben dem Sound Output eine gute Debugvariante.

5.4.5 Sound Output

Neben den LED-Anzeigen bieten Sound-Ausgaben eine weitere Möglichkeit, Fehler im Behavior zu erkennen. Beim Erreichen eines Zustandes wird dazu ein *Sound-Request* gesetzt, welches von einem *SoundOutput Provider* weiter verarbeitet wird. Zuerst wurde der Provider *NaoSoundControl* implementiert. Dieser setzte auf das von Aldebaran bereitgestellte Text-to-Speech Modul auf, wodurch die Ausgabe fast beliebiger Status Meldungen ermöglicht wurde.
Aus Gründen der Performance-Optimierung wurde entschieden, dass diese Lösung auf Dauer nicht beibehalten werden sollte. Eine sinnvolle Alternative erschien hier die Nutzung der *ALSA-API*. Diese ermöglicht es *Wave*-Dateien über die Sound Hardware auszugeben ohne diese für andere Prozesse zu blockieren. Da Speicherplatz hier kein größeres Problem darstellt, wird in dem dafür zuständigen Provider *NaoAlsaSoundControl* bewusst auf die Möglichkeit MP3-Dateien zu dekodieren und abzuspielen verzichtet. Zudem wäre die für die Dekodierung benötigte Rechnerleistung voraussichtlich zur Laufzeit nicht abrufbar ohne die Ausführung der übrigen Funktionen des Roboters erheblich zu beeinträchtigen.
Bei der Initialisierung des Providers NaoAlsaSoundControl wird zunächst geprüft, für welche Sound-Requests Wave Dateien bekannt sind und ob auf diese zugegriffen werden kann. Anschließend werden aus den ersten 100 Byte der Datei die benötigten Wave-Header Informationen ausgelesen. Hierzu gehören vor allem die Startposition und Länge des eigentlichen Audio-Streams und die Anzahl der Wiedergabekanäle (üblicherweise werden hier nur mono Dateien verwendet). Zudem wird die Anzahl der Bits, die für ein Sample verwendet werden, und die Sample Rate, also die Anzahl Samples pro Sekunde ausgelesen. Diese Informationen werden zusammen mit dem kompletten Audio-Stream für alle Sound-Requests in einem Array abgelegt.
Als Teil des *Cognition* Moduls prüft NaoAlsaSoundControl jetzt in regelmäßigen Abständen beim Aufruf der *update*-Methode, ob ein Sound-Request gesetzt wurde. Ist dies der Fall werden die zuvor ausgelesenen Header Informationen an die ALSA-Schnittstelle übergeben und damit die Audio-Hardware für die Wiedergabe der spezifizierten Sound Datei vorbereitet. Im Folgenden werden bei jedem Aufruf der update-Methode so viele Daten des Audio-Streams wie möglich in den ALSA Ringbuffer geschrieben, bis die Audiodatei komplett wiedergegeben wurde.
Probleme gibt es hierbei, wenn in zwei aufeinanderfolgenden Zuständen des Behaviors dieselbe Sound Datei abgespielt werden soll. Es gibt für den SoundOutput Provider keine Möglichkeit zu erkennen, ob das Behavior den Zustand gewechselt hat. Dadurch wird es nötig in einen Zustand zu wechseln, der keinen Sound-Request oder aber einen vom aktuellen verschiedenen Sound-Request setzt.

6 Verhalten

Als Verhalten des Roboters bezeichnet man die Entscheidungen die der Roboter auf Grund seiner Sensoreingaben fällt. Die daraus resultierenden Aktionen, wie laufen, schießen, sich umsehen oder ähnliches sind das Ergebnis fest vorprogrammierter Entscheidungsmöglichkeiten. Die Erstellung eines funktionierenden und schlüssigen Verhaltens ist daher ein wichtiger Aspekt bei der Entwicklung eines Fußball spielenden Roboters.
Die Entscheidungsfindung basiert auf Statemachines. Diese beinhalten mehrere Zustände, auch States genannt, die durch Zustandsübergänge miteinander verbunden sind. Je nachdem welche Sensorwerte der Roboter bekommt, wechselt das Verhalten von einem Zustand in einen anderen. Jeder Zustand bietet dann die Möglichkeit, weiterer Entscheidungen zu treffen oder Aktionen auszuführen.
Zur Programmierung des Verhaltens wird die für das *German Team*[Ger08] entwickelte Programmiersprache XABSL [LJRK08] benutzt. XABSL steht für *Extensible Agent Behavior Specification Language* und ist eine Sprache, um das Verhalten von autonomen Robotern mit Hilfe von Statemachines, in XABSL Options genannt, zu beschreiben. Der XABSL Kompiler kompiliert die in einer C Syntax geschriebenen XABSL Dateien in einen Intermediate Code, der durch die XABSL Engine, die auf dem Roboter läuft, ausgeführt wird. Der Vorteil des Intermediate Code liegt darin, dass nach Veränderungen am Verhalten nicht der gesamte Robotersourcecode neu kompiliert werden muss, sondern lediglich der XABSL Code, welcher dann als Intermediate Code auf den Roboter kopiert wird.
Die auf den Roboter kopierte Datei mit dem Intermediate Code des Verhaltens enthält nicht nur ein bestimmtes Verhalten, sondern kann beliebig viele enthalten. Welches der im Intermediate Code enthaltenen Verhalten, auch Agents genannt, gestartet werden soll, kann in der Datei Behavior.cfg eingestellt werden. Der aktuelle Nao Behavior Code beinhaltet mehrere Agents. Einen Agent für das Fußball spielen, einen anderen für den Demonstrationsmodus, sowie mehrere Agents zum Testen von Unterstrukturen eines Verhaltens. Ein jeder Agent stellt dabei den Einstiegspunkt, bzw. die Wurzel des entsprechenden Entscheidungsbaum dar. Weil das Verhalten hierarchisch aufgebaut ist, ist die Wurzel gleichzeitig die höchste Ebene des Verhaltens. In der Statemachine auf dieser Ebene werden die wichtigsten und relevantesten Entscheidungen getroffen und eine oder mehrere Options der nächst tiefere Ebene aufgerufen. Theoretisch ist es möglich beliebig viele Verhaltensebenen anzulegen.
Beim Nao Agent, der zum Fußball spielen aufgerufen wird, befinden sich beispielsweise auf den ersten beiden Verhaltensebenen die Wiederaufstehkontrollen, die den Nao bei einem Sturz wieder aufstehen lassen, einige Optionen für Debuggingausgaben, das Button Interface, mit welchem man den Nao für ein Spiel konfigurieren kann, der Gamecontroller, der die Anweisungen des Schiedsrichters umsetzt, sowie parallel dazu eine Head Control Option, die die Steuerung der Kopfbewegungen übernimmt. Ab der dritten Ver-

haltensebene beginnt das eigentlich Spielverhalten, mit der Vergabe der Spielerposition, der Teamtaktik und der Spielertaktik.
Das Verhalten des Roboters, bzw. sein Agent wird in jedem Frame einmal aufgerufen und der zugehörige Entscheidungsbaum von der obersten Entscheidungsebene bis zur untersten durchlaufen. Jede Entscheidungsebene besteht dabei aus mindestens einer Option, die aus mindestens einem State besteht. Innerhalb eines States werden die Entscheidungen getroffen, ob der aktuelle State gewechselt werden soll, in eine andere Option gesprungen werden soll, oder eine Aktion ausgeführt werden soll. Die meisten States eines Verhaltens beinhalten nur Entscheidungen, führen aber keine Aktionen aus. Aktionen werden hauptsächlich von den States in den Options der untersten Verhaltensebene des Entscheidungsbaumes, man kann auch sagen in den Blättern, ausgeführt. Die hier vorhandenen States beinhalten Aktionen, wie einen Walkrequest starten, einen Kick ausführen, oder sich umsehen. Da diese grundlegenden Aktionen häufiger gebraucht werden, werden für sie eigene kleine Statemachines angelegt, die man im Verhalten wiederverwenden kann. Somit muss man nicht jeden Zweig des Entscheidungsbaumes bis zur untersten Ebene programmieren, sondern kann von jeder Statemachine aus, aus jeder Ebene auf dieses Grundlagenverhalten zugreifen.

6.1 Entwicklungsprozess

Die Entwicklung eines komplett neuen Verhaltens ist sehr aufwendig und kann, wie jeder Entwicklungsprozess, Fehler enthalten. Um die Anzahl dieser zu minimieren und möglichst schnell ein funktionierendes Nao Verhalten zu bekommen wurde als Entwicklungsgrundlage des neuen Nao Verhaltens auf ein bereits vorhandenes Verhalten des Bremer Humanoid League Roboters zurückgegriffen. Dieses bildetet eine Programmier- und Lernbasis, die an die Bedürfnisse der neuen Nao Liga und an unseren Roboter angepasst wurde. Übernommen wurde mit einigen Anpassungen die oberste Verhaltensebene mit Initialisierungsphase, Buttoninterface, Gamecontroller und Head Motion Behavior. Alle darunter liegenden Schichten wurden von Grund auf neu designt, so dass vom Bremer Verhalten nur noch das Grundgerüst mit der höherwertigen Verhaltenslogik übrig blieb.
Die erste wichtige Änderung war das Hinzufügen der Umfallkontrolle. Diese wurde in die erste Verhaltensebene eingebaut, da die Aufstehbewegung absolut vorrangig vor allen anderen Bewegungen und Entscheidungen ausgeführt werden muss, denn der Roboter kann nicht Fußball spielen, wenn er liegt. Das Verhalten wechselt in die *stand_up* Option, wenn es merkt, dass der Nao gefallen ist und entscheidet dann, je nachdem ob der Nao auf dem Bauch oder Rücken liegt, welche Special Action (vergleiche Kapitel 4.2) zum Aufstehen genutzt werden soll. Steht der Nao wieder, läuft das normale Behavior weiter. Diese Option wurde allerdings während der regulären Robocupspiele deaktiviert, da die Roboter nach dem aktuellsten Reglement nicht selbständig aufstehen müssen, sondern per Hand wieder aufgestellt werden dürfen.
Damit sich der Nao nicht weiter bewegt, wenn er wieder aufgestellt wird, muss er während des Spiels aus seinem aktiven Spielmodus in einen neutralen Modus versetzt werden können. Diese Modi werden Gamestate genannt und können während eines Spiels geän-

dert werden. Zu diesem Zweck wurde der Gamecontroler und ein modifiziertes Buttoninterface übernommen. Beide ermöglichen Gamestatewechsel des Behaviors während des Spiels. Beim Gamecontroler geschieht dies über Kommandos, die per WLAN gesendet werden, beim Buttoninterface über den Hauptknopf des Naos auf seiner Brust. Während man beim alten Bremer Verhalten über das Buttoninterface das gesamte Setup des Roboters, mit der Auswahl der Teamfarbe, der Spielerposition, den Anstossoptionen usw., auswählen konnte, wurde das Nao Buttoninterface so weit vereinfacht, dass man mit ihm nur die Gamestates durchschalten kann. Das Setup des Naos wird vorher am Computer erledigt. Grund für diese Änderung war der Mangel an Tastern am Nao, mit denen man eine ordentliche Menüführung hätte programmieren können. Weil das gesamte Buttoninterface über den Hauptknopf auf der Brust des Naos gesteuert wird und dieser Knopf mehrfach belegt ist, muss man beim Durchschalten der Gamestates darauf achten, den Knopf nicht zu schnell hintereinander zu drücken, da das Nao Betriebssystem den Nao sonst herunter fährt. Neben den oben erwähnten Gamestates *playing* und *penalized* gibt es noch die states *initial*, *ready* und *set* die alle am Anfang eines Spieles benutzt werden und den Nao auf den Spielstart vorbereiten. So steht der Nao beispielsweise im state *set* aufrecht da und beginnt mit seinen Kopfbewegungen, um sich zu lokalisieren und begibt sich in seine Startposition.
Diese Kopfbewegungen sind ein Teil der *head_control* Option, die zur obersten Verhaltensebene gehört. Sie koordiniert die verschieden zur Verfügung stehenden Kopfbewegungen. Für den Nao wurden einige der schon vorhandenen Bewegungen angepasst, und so modifiziert, dass sie dem Nao die Möglichkeit bieten den Ball bzw. das Tor schneller als zuvor zu finden. Dieses wurde größtenteils durch Geschwindigkeitsanpassungen und Optimierungen der Trajektorien der Kopfbewegung erreicht.
Hat ein Nao den Ball gesehen, so teilt er dies seinen Teamkollegen über die neu hinzugefügte Teamkommunikation mit. Dabei broadcasted ein Nao seine Informationen, über den Ball, wie beispielsweise Position und Entfernung an alle anderen Naos im Team. Diese Informationen werden direkt per Input Symbol in das Behavior weiter gereicht und können dort die Entscheidungsfindung der anderen Naos im Team beeinflussen. Besonders wichtig ist dieses bei der dynamischen Rollenverteilung der Naos. War das alte Bremer Verhalten noch auf zwei Spieler ausgelegt, wurde das neue Verhalten um eine dynamische Rollenverteilung erweitert. Dabei kann jeder Nao auf dem Spielfeld, abgesehen vom Torwart, jederzeit zum Striker oder Supporter werden. Dieses Roleswitching ist unabhängig von der Anzahl der Spieler auf dem Spielfeld und ermöglicht somit beliebig große Teams. Die Teamgröße während des Robocups 2008 wurde auf zwei spielenden Naos begrenzt. Dieser Umstand und die Tatsache, dass sich das BreDoBrothers Team dazu entschloss mit zwei Strikern zu spielen führten dazu, dass während der Spiele beim Robocup keine aktives Role-Switching benutzt worden ist und somit auch nur eine einfache Spieltaktik eingesetzt worden ist.

6.1.1 Spieltaktik

Die Entwicklung einer passenden Spieltaktik zögerte sich auf Grund der hohen Ausfallrate der Naos, die ein Testen des Verhaltens erschwerte, und nicht endgültigen Spielregeln

bis auf wenige Wochen vor dem Robocup 2008 hinaus. Durch die hohe Ausfallrate der Naos war schnell klar, dass ein Spiel 4 gegen 4 nicht stattfinden würde. Daher wurde sowohl eine Spieltaktik für drei Naos als auch für zwei Naos auf dem Spielfeld entwickelt, wovon letztere zum Einsatz kam.

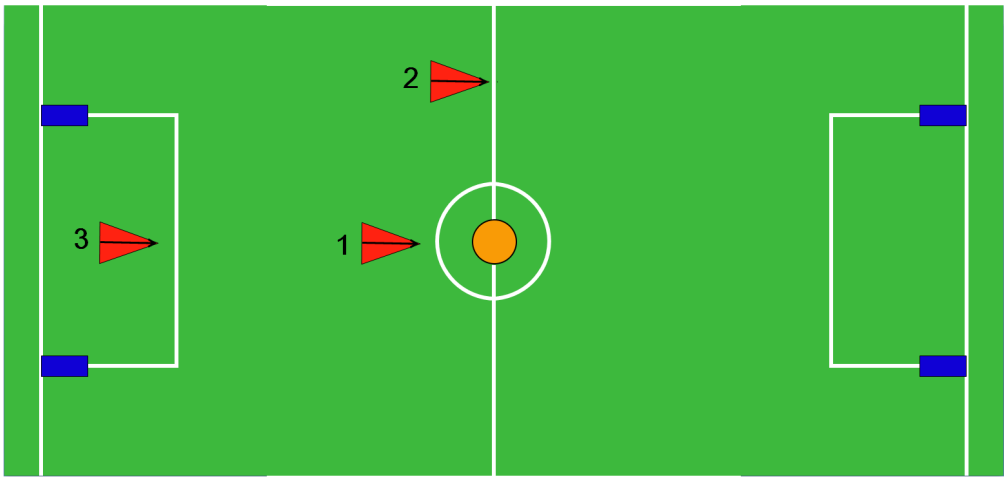


Abbildung 6.1: Anstoß mit drei Naos

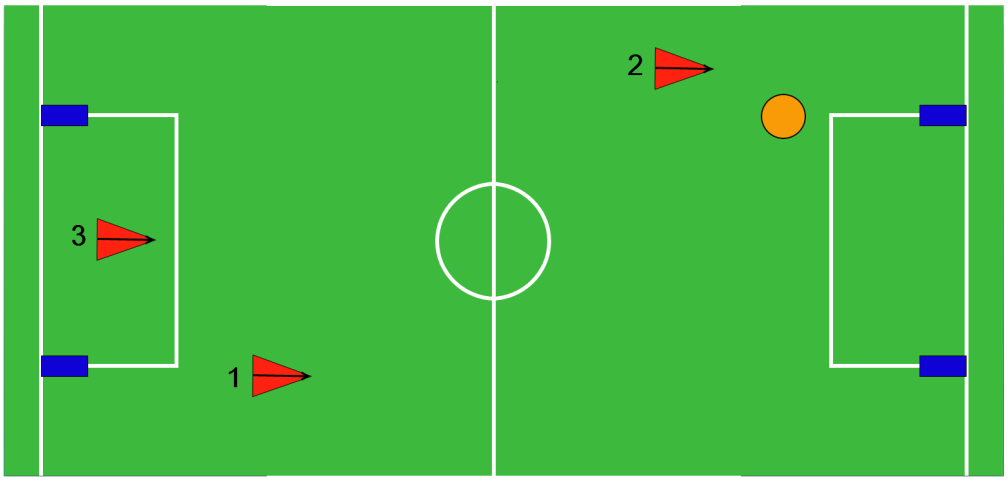


Abbildung 6.2: Spieltaktik mit drei Naos

Die Variante mit drei Spielern, zu sehen in Abbildung 6.1, sollte einen festen Goalie (Spieler 3) haben, der immer im Tor bleibt und nie seine Rolle wechselt, sowie einen Striker (Spieler 2) und einen Defensive Supporter (Spieler 1) haben. Letztere sollten ihre Rollen tauschen könnten. Der Supporter sollte direkt am Mittelkreis stehen und nach dem Anpfiff den Ball aus diesem heraus schießen. Währenddessen wäre der Striker, der

an der Mittellinie gestanden hätte in die gegnerische Spielfeldhälfte gelaufen und hätte den Pass des anstoßenden Naos angenommen und versucht ein Tor zu schießen. Während des Spiels sollte sich der Striker, wie in Abbildung 6.2 zu sehen ist, in der gegnerischen Spielfeldhälfte aufhalten, versuchen den Ball zu bekommen und auf das Tor zu schießen, während sich der Supporter in der eigenen Hälfte aufhalten und versuchen sollte gegnerische Schüsse abzuwehren. Um sich nicht gegenseitig die Sicht zu nehmen sollten sich beider Naos während des Spiels Punkt gespiegelt zum Mittelpunkt des Spielfeldes bewegen. Der Supporter sollte also auf der linken Seite stehen, wenn der Striker auf der Rechten gewesen wäre und umgekehrt.
Das gesamte Spielverhalten wurde vor dem Robocup genauestens geplant, allerdings nie programmiert, geschweige denn eingesetzt. Schwierigkeiten bereiteten dabei nicht nur die benötigte Lokalisation, die nicht funktionierte, sondern auch der Zeitmangel und daraus resultierend die Möglichkeit das Verhalten zu testen. Somit schied auch die Variante aus, bei einem Spiel zwei gegen zwei diese Spieltaktik ohne den Goalie zu benutzen.
Letztendlich waren die taktischen Möglichkeiten, die beim Robocup übrig geblieben waren, mit einem Striker und einem Goalie zu Spielen, mit einem Striker und einem Supporter oder aber mit zwei Strikern. Sowohl die Tatsche, dass keines der teilnehmenden Teams in den ersten Spielen der Vorrunde gezielt auf das Tor schießen konnte, sowie die nicht funktionierende Teamkommunikation und das damit fehlende Role-Switching, ließ die Entscheidung auf ein Spiel mit zwei Strikern fallen.
Das frühe Entwicklungsstadium des Strikerverhaltens ermöglichte in den ersten Vorrundenspielen kein gezieltes Schießen auf das gegnerische Tor. Die Striker waren lediglich in der Lage zum Ball zu gehen und diesem irgendwo hin zu schießen. Eine Absprache zwischen den Naos gab es nicht, weswegen grundsätzlich beide zum Ball liefen und der Roboter der zu erst in Schussposition war schießen durfte. Trotz dieses simplen Verhaltens gelang den BreDoBrothers das erste Tor der Nao Liga. Ein gezieltes Ausrichten zum gegnerischen Tor wurde erst in den letzten Spielen eingebaut, änderte aber an der grundlegenden Taktik, mit zwei Strikern zu spielen, nichts.

6.1.2 Striker

Das Verhalten des Strikers setzt sich aus drei Elementen zusammen. Diese sind in der Reihenfolge ihrer Aufrufe die Suche nach dem Ball, das Laufen zum Ball und das Schießen des Balls. Jedes Einzelne dieser Handlungselemente ist eine eigene Option, die zum Teil in anderen Spielerverhalten, wie dem des Supporters, wiederverwendet werden. Die Suche nach dem übernimmt die Option *start_searching_for_ball*, die aufgerufen wird, wenn der Nao den Ball für eine vorgegebene Zeitspanne nicht mehr gesehen hat. Jede der für den Striker benutzten Options wurde von einer Grundversion in mehreren Schritten Version für Version weiterentwickelt. In der ersten Version dieser Option bleibt der Nao gerade stehen und beginnt durch Kopfbewegungen in seinem Sichtbereich den Ball zu suchen. Kann dieser beim Umsehen nicht gefunden werden, so beginnt sich der Nao mit maximaler Drehgeschwindigkeit umzudrehen und dabei weiterhin durch Kopfbewegungen den Ball zu suchen. Diese sehr einfache Art der Suche erweist sich allerdings als unzureichend, um in jeder Situation den Ball zu finden. Vor allem, wenn der Ball näher

als vierzig Zentimeter am Nao liegt, kann dieser den Ball nicht sehen, weil er nicht weit genug nach unten sehen kann. Die Sichtgrenze des Naos wurde experimentell auf etwa vierzig Zentimeter bestimmt. Es wurde eine Special Action (vergleiche Kapitel 4.2) bzw. später eine Erweiterung der Walking Engine (vergleiche Kapitel 4.1) verwendet, die es dem Nao ermöglichte direkt vor seine Füße sehen zu können.
Ein weiteres Problem wurde erst beim Robocup ersichtlich. In einigen Spielen kam es vor, dass der Nao beim Ballsuchen den Ball zwar sah, sich aber bei der Drehung so weit weiter bewegte, dass er ihn sofort wieder verlor und erneut suchen musste. Es zeigte, dass die Drehgeschwindigkeit so schnell war, dass der Nao den Ball schon wieder verloren hatte, bis er zum Stehen gekommen war. Auch dieses wurde korrigiert.
Die aktuelle Version der Option *start_searching_for_ball* funktioniert nun so, dass der Nao zuerst vor seinen Füßen nach dem Ball sucht, da sich der Ball erfahrungsgemäß meistens dort befindet, wenn er ihn aus der Sicht verliert. Danach richtet sich der Nao auf und beginnt seine Umgebung vor sich abzusuchen. Sollte er auch dort keinen Ball sehen, beginnt er sich zu drehen, in die Richtung in der er den Ball zuletzt gesehen hat, bevor er ihn verloren hat. Nach jeweils 120 Grad Drehung schaut er wieder nach unten vor seine Füße. Dieses ist wichtig, weil es vorkommen kann, dass sich der Ball anfangs direkt neben oder hinter dem Nao befindet.

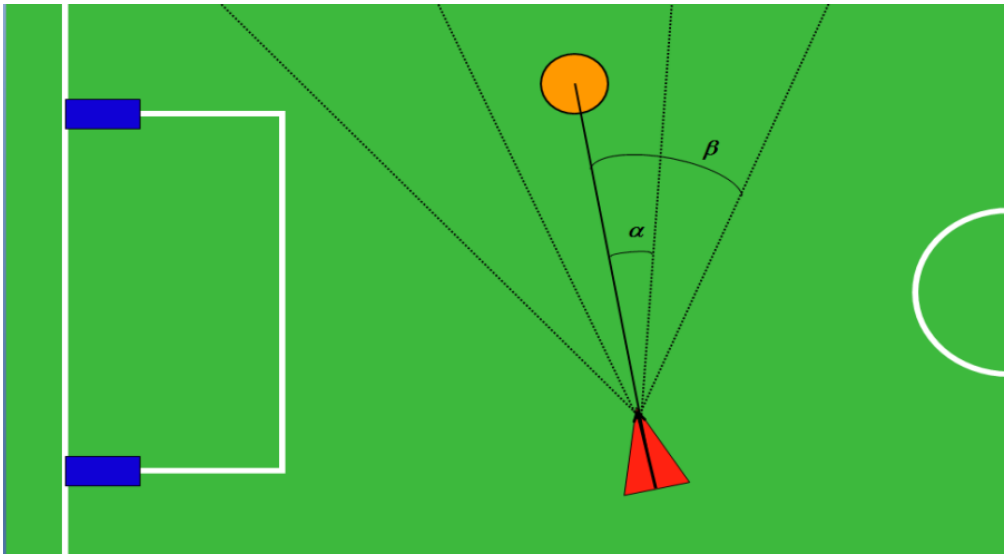


Abbildung 6.3: Regelungsparameter zur Ballannäherung

Hat der Striker den Ball gesehen, beginnt er darauf zu zu laufen. Dieses Verhalten ist das zweite wichtige Element des Strikerverhaltens. In den ersten Versionen dieser Option wurde versucht den Roboter möglichst geradlinig auf den Ball zu bewegen zu lassen. Zu diesem Zweck wurde eine fünf Punkt Regelung in XABSL implementiert. Das bedeutet, die Regelgröße, also die Abweichung vom idealen Weg zum Ball, wurde in fünf Bereiche aufgeteilt. Je nachdem in welchem Bereich sich die Abweichung befand, wurden mit fest

vorgegebenen Stellwerten, in Form von Drehbewegungen des Naos, die Abweichung zum idealen Weg korrigiert. Wie im Detail in Abbildung 6.3 zu sehen ist, ist die Regelung so aufgebaut, dass der Nao ab einem Winkel zwischen Roboter und Ball, der größer als β ist, stehen bleibt und sich auf der Stelle mit einer vorgegeben Drehgeschwindigkeit Richtung Ball dreht. Ist der Winkel kleiner als α, beginnt er sich auf den Ball zu zu bewegen. Überschreitet der Winkel zwischen Roboter und Ball dabei wieder den kleineren Grenzwinkel α, so führt der Nao zusätzlich zur Vorwärtsbewegung eine überlagerte Drehbewegung mit fester Drehgeschwindigkeit durch. Somit bewegt er sich auf einer leichten Schlangenlinie zum Ball. Schnell war klar, dass dieses nicht die optimale Lösung sein konnte. Für die zweite Version der *go_to_ball* Option wurde ein Proportionalregler implementiert. Es wurden keine festen Drehgeschwindigkeiten mehr vorgegeben. Beim P-Regler hängt die dem Laufen überlagerte Drehgeschwindigkeit vom Winkel des Roboters zum Ball ab. Somit können leichte Abweichungen von der direkten Bahn zum Ball durch leichte Drehungen korrigiert werden.
Es wurden zusätzliche Versuche durchgeführt, die Laufgeschwindigkeit des Naos durch einen P-Regler an den Abstand zum Ball zu koppeln, so dass der Nao langsamer wird, wenn er dem Ball auf den letzten Dezimetern näher kommt. Diese Versuche führten aber zu keinem zufriedenstellenden Ergebnis und wurden auf Grund von Zeitmangel eingestellt.
Die aktuelle Version der *go_to_ball* Option funktioniert nun so, dass der Roboter bis zu einem bestimmten Winkel auf der Stelle stehen bleibt und sich mit maximaler Drehgeschwindigkeit Richtung Ball dreht. Ist dieser Winkel überschritten läuft der Roboter auf den Ball zu und der P-Regler übernimmt die Bahnkorrektur. Wird der Ball, während der Nao auf ihn zugeht, so weit bewegt, dass der Winkel zwischen Roboter und Ball größer als oben erwähnter Grenzwinkel wird, bleibt der Roboter stehen und die Option beginnt von Neuem.
Hat sich der Roboter dem Ball auf 40 cm genähert, wechselt das Strikerverhalten in die Option *approach_ball_and_kick*. Diese Option übernimmt das Positionieren des Roboters vor dem Ball. Dabei wird keine Korrektur mehr der Schussposition zum Tor durchgeführt. Dieses ist die Aufgabe der übergeordneten Option, die ausgeführt worden ist, bevor das Verhalten in die Option *approach_ball_and_kick* gewechselt ist. Wie bei allen oben beschriebenen Verhaltenselementen hat auch dieser Teil des Verhaltens eine Entwicklung durchlaufen, die zu der Version geführt hat, die beim Robocup eingesetzt worden ist. In der ersten Version dieser Option ging es lediglich darum den Roboter so dicht vor dem Ball stoppen zu lassen, dass der Nao den Ball schießen konnte. Die Schwierigkeit dabei war, dass der Nao den Ball trotz gesenktem Kopf nur bis zu einer Entfernung von etwas unter 40 cm sehen konnte. Für alle Abstände unter diesem Wert wurde der Ball im Stehen nicht mehr wahrgenommen. Gelöst wurde das Problem, wie bereits erwähnt, durch eine Special Action (vergleiche Kapitel 4.2), die den Nao so weit nach vorne beugen ließ, dass er seine Fussspitzen sehen konnte. Die Option funktionierte seit diesem Zeitpunkt so, dass der Nao ab 40 cm Ballabstand blind auf den Ball zu lief und dann stehen blieb, wenn die Odometriedaten dem Behavior sagten, dass er nun 40 cm gelaufen sein müsste und vor dem Ball stehen müsste. Danach beugte sich der Nao vor und schaute, ob der Ball wirklich vor seinen Füssen lag. War er zu weit weg, richtete

er sich wieder auf und lief das fehlende Stück. Stand der Nao nah genug am Ball wurde über die Y-Position des Balls der Fuss ausgewählt mit dem geschossen werden sollte und die Schuss Special Action aufgerufen.
Die fortgeschrittene Version der Walking Engine (vergleiche Kapitel 4.1) führte dazu, dass der Nao vorn über gebeugt laufen konnte. Damit war es möglich den Ball während des gesamtes Annäherungsprozesses im Blick zu behalten. Auf die Spezial Action zum Vorbeugen wurde von da an verzichtet. Allerdings lief der Nao nach vorne geneigt nicht immer hundertprozentig geradeaus, wodurch es vorkam, dass er seitlich vom Ball stand und diesen nicht mehr schießen konnte. Die Option wurde daraufhin so erweitert, dass der Nao auch seine seitliche Position vor dem Ball korrigieren konnte. Diese Korrektur wurde dann durchgeführt, wenn er direkt vor dem Ball stand, der X-Abstand zum Ball schon stimmte, aber der Y-Abstand noch zu groß oder zu klein war. Dass der Y-Abstand zu klein war kam dann vor, wenn der Ball genau zwischen den Füßen lag. Leider zeigte sich, dass das Seitwärtsgehen keine Stärke des Naos war. Zusätzlich erzeugte eine nach vorne gebeugte Seitwärtsbewegung eine leichte Drehung des Nao, so dass diese Korrekturbewegung nicht optimal funktionierte.
Eine weitere Verbesserung der Walking Engine ergab eine wesentlich bessere Odometrie, während sich der Nao bewegte. Somit war es möglich sich dem Ball anzunähern ohne diesen ständig im Blick zu haben. Das ungenaue vorgebeugte Laufen aus der Option wurde entfernt und durch wenige notwendige Bewegungen zum Herunterschauen ersetzt. Daraus entstand mit einigen Optimierungen die jetzige Version der Option.
Die aktuelle Version der *approach_ball_and_kick* Option funktioniert nun so, dass sich der Nao nur noch mit Hilfe der Pitch Fähigkeit der Walking Engine beim Stehen nach vorne beugt und die Position des Balls bezüglich seiner eigenen bestimmt. Daraufhin richtet sich der Nao auf und läuft blind, mittels Odometrie, auf den Ball zu. Gleichzeitig wird durch eine überlagerte Seitwärtsbewegung die Y-Position zum Ball korrigiert. Diese Seitwärtsbewegung ist stärker, je größer die seitliche Abweichung des Balls zum Nao ist. Hat der Nao durch die Daten der Odometrie erkannt, dass er dicht genug am Ball ist, bleibt er stehen und beugt sich wieder nach Vorne. Ist der Ball noch immer zu weit von der optimalen Schussdistanz zum Fuss entfernt, so richtet sich der Nao wieder auf und nähert sich dem Ball weiter. Sollte der Ball zu weit seitlich liegen, so läuft der Nao rückwärts vom Ball weg, überlagert von einer Seitwärtsbewegung Richtung Ball. Im Anschluss nähert sich der Nao dem Ball wieder ganz normal. Die Überlagerung der Seitwärtsbewegung mit einer Vorwärts- und Rückwärtsbewegung hat sich als besser erwiesen, als nur seitlich zu gehen. Hat der Nao es geschafft sich nach vielen Korrekturen passend vor dem Ball auszurichten, wird anhand der Y-Position des Balls entschieden, welcher Fuß zum Schießen benutzt werden soll und die entsprechende Special Action aufgerufen.
Um ein echtes Fussballspiel zu absolvieren ist es nicht nur notwendig, dass der Striker den Ball schießen kann, sondern dieses sollte möglichst Richtung Tor geschehen. Dazu muss sich der Roboter so ausrichten, dass sich der Ball zwischen ihm und dem Tor befindet. Die Entwicklung dieses Teils des Verhalten wurde erst während des Robocups durchgeführt. Dabei wurden zwei unterschiedliche Ansätze verfolgt, die letztendlich aber nur unzureichend funktionierten. Der erste Ansatz verfolgte die Idee, den Roboter mit

einem konstanten Abstand auf den Ball ausgerichtet zu lassen und mit Seitwärtsbewegungen so lange um den Ball herum zu bewegen, bis das Tor vor dem Roboter liegt. Diese Ausrichtung dauert allerdings selbst im Simulator und erst recht auf dem realen Roboter sehr lange. Hinsichtlich der Schwierigkeiten, die der Nao beim Seitwärtslaufen hatte, wurde dieser Ansatz nie im Spielverhalten eingesetzt.
Der andere Ansatz versuchte den Roboter bereits während er sich auf den Ball zu bewegte passend zum Tor auszurichten und somit die Aufgabe der *go_to_ball* Option zu übernehmen. Die Option setzt keine eigenen Walkrequests, sondern benutzt die Option *go_to_point_rel*, die es ermöglicht zu einem Punkt relativ zur aktuellen Nao Position zu laufen. Dabei wurde bei diesem Ansatz versucht mit einfachen Winkelsätzen und dem Winkel zum Tor eine Position 60 cm hinter dem Ball zu berechnen und diese direkt anzusteuern. Somit sollte sich der Ball genau zwischen Nao und Tor befinden. Allerdings bereitete vor allem die Abbruchbedingung, wann denn der richtig Punkt hinter dem Ball erreicht ist, Schwierigkeiten. Ein Weiterentwickeln dieses Ansatzes könnte sich aber für die Zukunft lohnen.
Der erste Ansatz positioniert den Roboter sehr genau, hat aber den entscheidenden Nachteil, dass die Positionierung durch die Seitwärtsbewegungen sehr langsam geschieht. Beim zweiten Ansatz ist dies nicht der Fall. Die Positionierung erfolgt sehr viel schneller, kann dafür aber auch wesentlich schlechter ausfallen. Ein Designfehler in beiden Ansätzen ist jedoch erst zum Ende des Robcups 2008 aufgefallen. Beide Verhalten versuchen sich so auszurichten, dass sie auf die Mitte des Tores schießen können, dabei würde es reichen, wenn man nur etwas neben die Pfosten schießen könnte. das heißt man könnte einen größeren Schusswinkel nutzen und wesentlich öfter schießen, da man zum Ausrichten vor dem Tor einen größeren Winkel zur Verfügung hat.
Insgesamt hat das Strikerverhalten für seine kurze Entwicklungs- und Testzeit so gut funktioniert, dass zum Robocup 2008 grundsätzlich 2 Striker auf dem Spielfeld standen. Trotzdem wurde die Entwicklung des Supporterverhaltens genauso voran getrieben, wie die des Strikers und des Goalies.

6.1.3 Supporter

Der Supporter hat, wie der Name schon sagt, eine rein unterstützende Funktion. Seine Verhaltensprogrammierung ist so ausgelegt, dass er sich geschickt auf dem Spielfeld positioniert. Bei einer Positionierung des Naos zwischen Ball und eigenem Tor spricht man von einem Defensive Supporter, da er einen Torschuss für den Gegner erschwert. Im Gegensatz dazu positioniert sich der Offensive Supporter so, dass er einen vom Striker verschossenen, oder vom Torpfosten abgeprallten Ball schnell abfangen kann und so selbst zum Striker werden kann.
Auf Grund des Zeitmangels vor dem Robocup 2008 wurde kein Verhalten für einen Offensive Supporter geschrieben. Stattdessen wurden mehrere Varianten eines Defensive Supporters implementiert. Dieser sollte Bälle in der eigenen Spielfeldhälfte abfangen und wieder nach vorn zum Striker spielen. Dabei wurden zwei Ansätze verfolgt, die abhängig von der Funktionsfähigkeit der Lokalisierung eingesetzt werden sollten.

Mit Lokalisierung

Sollte eine Lokalisierung vorhanden sein, so kann der Roboter einen beliebigen Punkt im Feld-Koordinatensystem direkt ansteuern. Das Verhalten sieht in diesem Falle vor, auf den am nächsten gelegenen Punkt der Verbindungslinie zwischen Ball und eigenem Tor zu laufen, um möglichst schnell einen direkten Schuss zu verhindern (s. Abb. 6.4). Es sind hierbei einige Spezialfälle abzufangen, so etwa wenn der Roboter nicht seitlich der Verbindungslinie steht, oder er erst noch um den Ball herumlaufen muss oder der Ball zu nah am eigenen Strafraum liegt.
Als nächstes versucht der Roboter die eigene Positionierung zu verbessern, ohne dabei die direkte Linie zu verlassen. Dies betrifft daher vor allem den Abstand zum Ball, der so klein wie möglich gewählt werden sollte, um Schüsse in einem möglichst großen Winkel zu verhindern. Es wird dabei vom ungünstigsten Fall ausgegangen, nämlich dem, dass ein gegnerischer Spieler gerade in Ballbesitz ist und sich vom Supporter aus gesehen direkt hinter dem Ball befindet. In diesem Fall sehen die Regeln vor, dass sich der Supporter nicht auf weniger als 50 cm dem gegnerischen Spieler nähern darf, ohne vom Feld genommen zu werden. Dies ist daher auch die Entfernung, die der Supporter anstrebt, wobei er sich trotzdem nur im Bereich zwischen eigenem Strafraum und Mittellinie bewegt. Die Ausrichtung auf den Ball wird dabei beibehalten, um diesen im Blick zu behalten und die eigene Zielposition möglichst schnell korrigieren zu können, der Nao läuft dabei also ggf. rückwärts. Nähert sich der Ball, so wird die Position trotzdem beibehalten, da in diesem Fall der Ballbesitz auf den Supporter übergeht.
Das Beschriebene ist das Standardverhalten des defensiven Supporters. Um einen Stillstand des Spiels z.B. bei Ausfall des eigenen Strikers zu verhindern, gibt es jedoch auch die Möglichkeit, dass der Supporter das Annäherungs- und Schussverhalten des Strikers ausführt, dies geschieht insbesondere auch, falls keine Teamkommunikation genutzt wird. Auf der eigenen Seite wird die Angriffsoption genutzt, falls der Supporter nach der Positionierung näher am Ball ist als der Striker (bei Ausfall der Teamkommunikation also in jedem Fall). Liegt der Ball in der gegnerischen Hälfte gibt es zusätzlich noch die Bedingung, dass der Supporter schon länger als 30 Sekunden auf der Verteidigungsposition sein muss.

Ohne Lokalisierung

Im Falle einer fehlenden Lokalisierung, sollte der Supporter anhand des Ballpercepts und des Goalpercepts versuchen, sich zwischen eigenem Tor und Ball zu positionieren. Dazu muss der Roboter verständlicherweise erst den Ball finden. Daher wird im Startzustand nach dem Ball gesucht bis dieser gefunden wurde. Ist dies geschehen, muss das eigene Tor gefunden werden. Sobald der Roboter auch dies gefunden hat, startet der Vorgang der Positionierung. Dazu dreht der Roboter sich erst in Richtung der Mitte zwischen Tor und Ball, bis der Winkel des Roboters zum Ball dem Winkel des Roboters zur Mitte des Tors entspricht. Daraufhin läuft der Roboter zur ungefähren Position des Mittelpunktes der Linie die den Ball und die Mitte des Tors verbindet. Die Entfernung x die er dafür

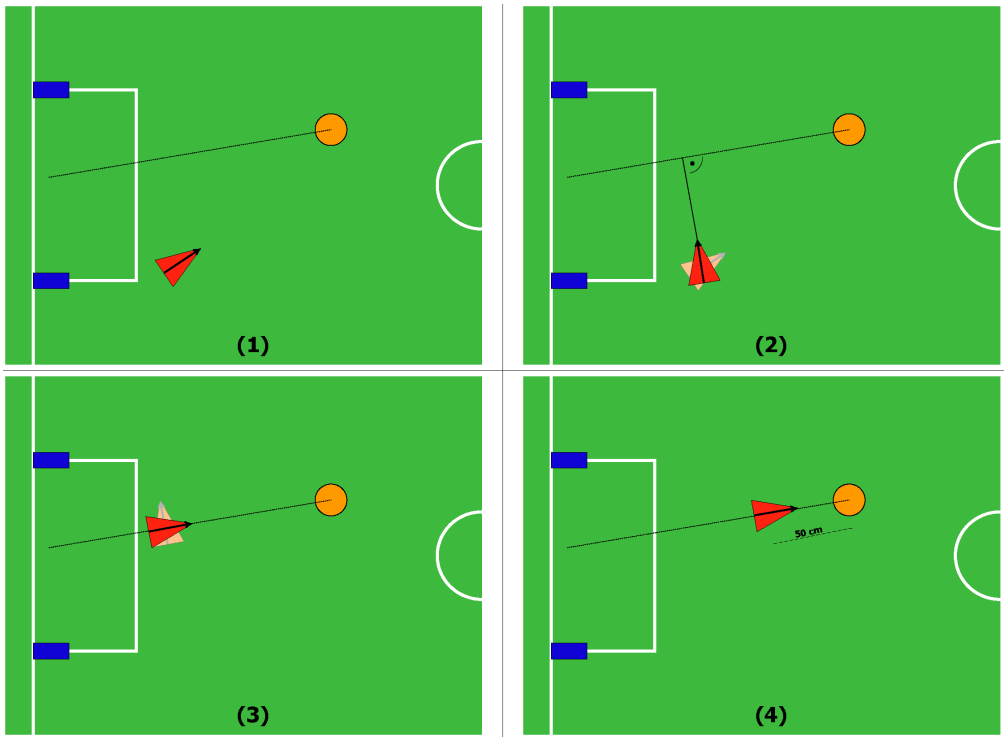


Abbildung 6.4: Supporter Verhalten mit Lokalisierung

läuft wird wie folgt berechnet.

$$x = cos(a) * r$$

Dabei ist a der Winkel vom Roboter zum Ball und r die Distanz zum Ball (Abbildung 6.5). Bei diesem Verhalten muss stets darauf geachtet werden, dass der Roboter sich nicht hinter dem Ball positioniert und der Ball zwischen Roboter und eigenem Tor liegt. In diesem Fall könnte es schnell zum Eigentor kommen, sollte der Roboter sich vorwärts bewegen, oder den Ball schießen. Falls dieser Fall auftritt, muss der Roboter sich erst um den Ball drehen. Dies wird weiter unten in der Funktion *Walk around Ball* beschrieben. Zwischen Tor und Ball angekommen, dreht sich der Roboter zu letzterem um sich davor zu positionieren. Ist der Roboter so ausgerichtet, dass er den Ball Richtung gegnerisches Tor schießen kann, so schießt er den Ball. Andernfalls richtet er sich so aus, dass der Winkel zum Ball ungefähr dem Winkel zum gegnerischen Tor ist.

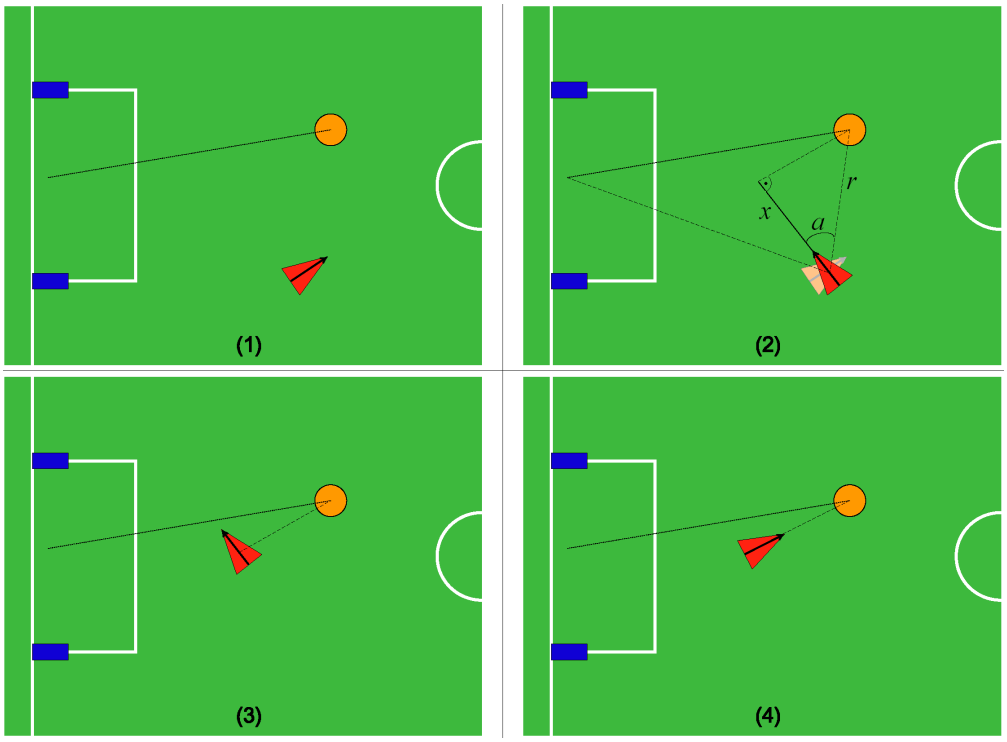


Abbildung 6.5: Supporter Verhalten ohne Lokalisierung

Walk around Ball

Diese Funktion wurde implementiert, um dem Roboter die Möglichkeit zu geben sich um den Ball zu drehen und sich auf der gegenüber liegenden Seite zu positionieren. Diese Funktion ist sehr nützlich, sollte der Supporter so stehen, dass der Ball zwischen ihm und dem eigenen Tor liegt. In diesem Fall ist der Winkel des Roboters zum Ball ungefähr gleich dem Winkel zur Mitte des eigenen Tores. Um den Ball nicht in die Richtung des

eigenen Tors zu bewegen, muss sich der Supporter mit der Walk around Ball Funktion um den Ball drehen, um zwischen eigenem Tor und Ball zu stehen. Im Falle einer nicht vorhandenen Lokalisierung geschieht dies nur mithilfe des Ballpercepts. Daher muss der Roboter den Ball ständig im Blick behalten um sich um ihn drehen zu können. Da sich der Kopf des Roboters nur bis zu einem bestimmten Grad drehen kann, darf der Winkel des Roboters zum Ball nicht größer als 80° sein. Der dabei verfolgte Ansatz funktioniert wie folgt (siehe Abbildung 6.6):

1. Der Roboter läuft schräg nach vorne rechts bis der Winkel zum Ball $> 70°$ ist
2. Nun läuft der Roboter nach vorne bis der Winkel zum Ball $< 80°$ ist und dreht sich dann zum Ball
3. Der Roboter läuft erneut schräg nach vorne rechts bis der Winkel zum Ball $> 70°$ ist
4. Nun sollte der Roboter zwischen eigenem Tor und Ball stehen und muss sich nur noch zum Ball ausrichten

Dieser eher ungewöhnliche Ansatz wurde gewählt, um dem Roboter die Möglichkeit zu geben den Ball zu sehen während er sich um diesen bewegt.

Das Supporter-Verhalten wurde beim RoboCup 2008 nicht eingesetzt. Aufgrund der geringen Anzahl an Robotern auf dem Spielfeld wurden beide Spieler mit einem Striker-Verhalten ausgestattet, das mehr Tormöglichkeiten versprach. Außerdem war das Supporter-Verhalten noch unausgereift und wurde in der Praxis nicht getestet.

6.1.4 Goalie

Der Goalie ist, wie der Name schon sagt, der Torwart. Er ist der einzige Roboter der eigenen Mannschaft, der sich im 600x1400 mm großen Strafraum vor dem eigenen Tor aufhalten darf. Das Verhalten des Torwarts setzt sich daher aus drei Elementen zusammen:

1. Die Verteidigung im Tor
2. Das Herausschlagen des Balls aus dem Strafraum
3. Die Zurückorientierung ins Tor nach Punkt 2

Dabei besitzt der Goalie eine übergeordnete Steuerungsoption, die entscheidet in welchen der drei oben genannten Verhaltensoptionen er sich befinden soll. Diese Steuerungsoption kann ihn jederzeit aus jeder der aufgerufenen Optionen wieder herausnehmen und ihn ein anderes Verhalten ausführen lassen.
Im folgenden werden die konzeptionellen Ideen vorgestellt, die diese drei Elemente realisieren sollen. Da diese jedoch nicht auf dem realen Roboter getestet wurden, werden keine konkreten Umsetzungen beschrieben. Eine mögliche Funktionsweise gemäß dieser Konzepte wurde im Zuge der PG521 implementiert und im Simulator unter idealisierten Bedingungen getestet.

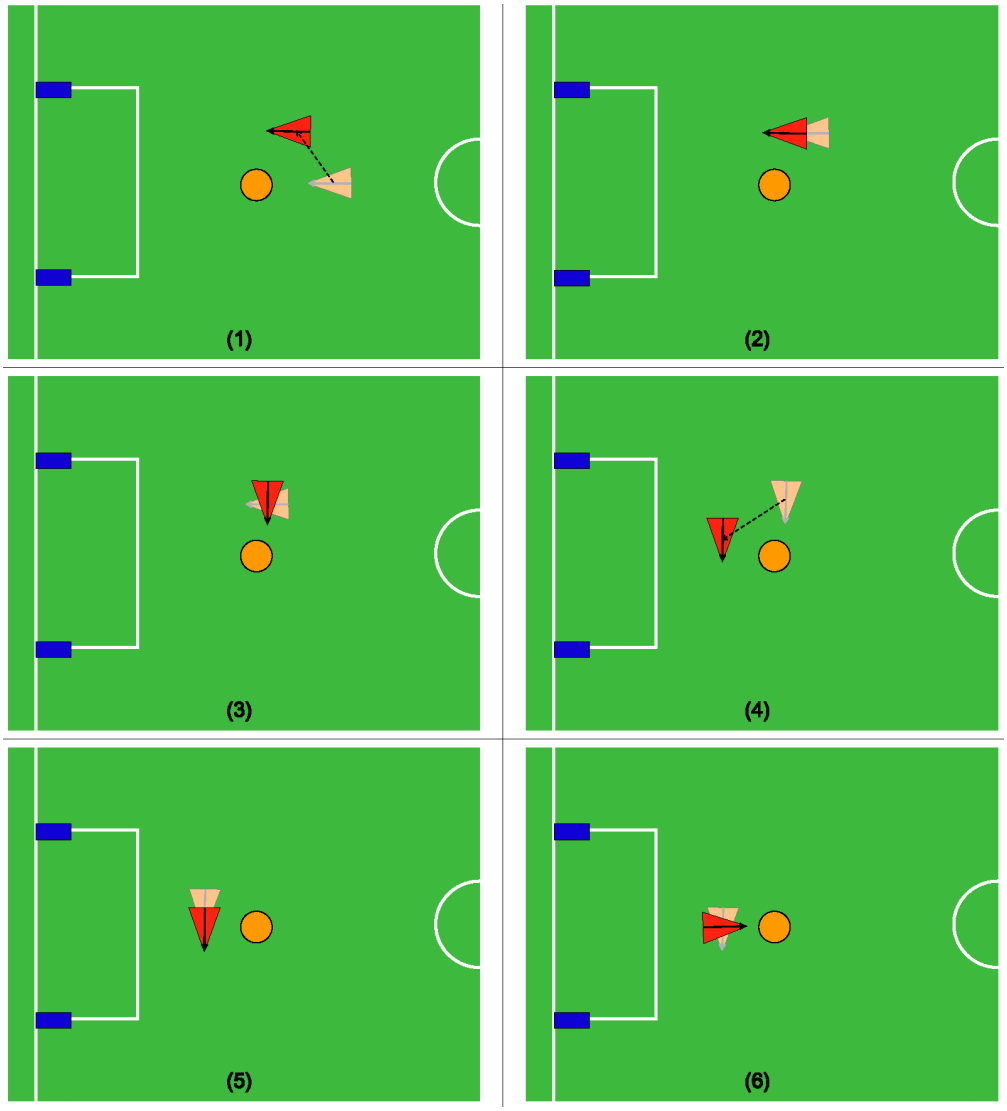


Abbildung 6.6: Walk around Ball

Die Verteidigung im Tor

Während des Spiels liegt der Ball im Normalfall auf einem Punkt in der eigenen oder der gegnerischen Spielfeldhälfte. Des weiteren kann angenommen werden dass sowohl gegnerische als auch eigene Feldspieler versuchen sich vor diesem Ball zu positionieren und zu schießen. Während dieses Spielvorgangs ist es die Aufgabe des Goalies auf der Linie zu stehen um den Ball zu beobachten. Sobald der Ball nach einem Schuss auf ihn zurollt, sollte er eine geeignete Abwehrbewegung ausführen (Siehe Kapitel 4.2).
Weiterhin ist es sinnvoll, dass der Goalie das aus dem echten Fußball bekannte Positionierungsverhalten nachahmt. Bei Zunahme des Winkels vom Ball zum Tor, positioniert er sich so in der entsprechenden Torhälfte, dass die ungeschützte Torfläche möglichst gering bleibt. Die Positionierung hängt dabei sowohl vom Winkel, als auch von der Entfernung des Balls ab. Dabei ist bei kurzen Distanzen eine stärkere Positionierung nötig als bei größeren.
Sollte der Ball sich sehr nah am Torwart befinden, ist ein Wechsel des Verhaltens nötig.

Das Herausschlagen des Balls

Eine mögliche Spielsituation in der die übergeordnete Steuerungsoption des Goalies ihn dazu bringen sollte sein normales Abwehrverhalten zu verlassen, würde eintreffen wenn der Ball im Strafraum liegt. Kein anderer Mitspieler dürfte in dieser Situation zum Ball gehen, da er unweigerlich den Strafraum betreten würde. An dieser Stelle kann der Goalie vereinfachend auf das Schussverhalten für den Striker (siehe Kapitel 6.1.2) zurückgreifen. Das Ende dieses Verhaltens kann entweder der erfolgreiche Schuss sein, der an der Entfernung von ihm zum Ball gemessen wird[6], oder die eigene Position, sofern er zu weit den Strafraum verlassen hat. Was sich im Spielbetrieb als nützlicher erweisen wird, kann an dieser Stelle nur vermutet werden.

Rückkehr zum Tor

Nach einem solchen möglichen Verlassen des Tores besteht die Notwendigkeit, dass der Torwart wieder seine angestammte Position zwischen den Torpfosten einnimmt. Dazu hat er lediglich zu entscheiden, ob es aufgrund der Entfernung zum eigenen Tor sinnvoll ist sich umzudrehen und vorwärts zurückzulaufen, oder dies rückwärts zu tun.
Eine weitere Situation, in der die Rückkehr zum Tor nötig sein kann, wäre wenn der Torwart während er im Tor verteidigt, sich aufgrund fehlerhafter Selbstlokalisierung oder unsauberen Bewegens in y-Richtung, zu sehr von seiner Position entfernen würde. Hierbei ist jedoch wichtig, dass die Parameter nicht zu scharf gestellt werden, da sonst bei falschen Lokalisierungsergebnissen ein Pendeln zwischen der Rückkehr- und der Verteidigungsoption entstehen könnte.

6 In diesem Fall wäre möglich, dass der Schuss misslingt und so der Torwart durch leichte aufeinanderfolgende oder missglückte Schüsse den Strafraum verlässt und sogar bis zum gegnerischen Tor läuft

Wichtig ist auch in der übergeordneten Steuerungsoption zu unterscheiden, ob der Torwart wegen eines Herausschlagens oder wegen einer Postionsverletzung während der Verteidigung, den Strafraum bzw. den erlaubten Wirkungsbereich verlässt. Ansonsten könnte es sein, dass ein Herausschlagen ungewollt abgebrochen wird.

6.2 Probleme

Bei der Entwicklung des Nao Behaviors kristallisierten sich drei Probleme heraus, die aber eher allgemeiner Natur sind. Sie charakterisieren die generellen Schwierigkeiten beim Erstellen eines Behaviors.
Die erste Schwierigkeit lag darin, die Idee für ein ausgearbeitetes Verhalten in eine Statemachine zu übertragen. Dabei wurde zu Beginn häufig der Fehler gemacht, zu große und aufwändige Statemachines zu implementieren, die jeden kleinen Sonderfall berücksichtigten und in einem eigenem State behandelten. Dieses führte dazu, dass die Statemachines zu unübersichtlich wurden und das resultierende Verhalten häufig nicht mehr nachvollziehbar war. Es stellte sich heraus, dass in vielen Fällen eine Abstraktion von vielen Sonderfällen auf einige wenige allgemeine Fälle zu einer wesentlich robusteren Statemachine mit weniger Zuständen und Übergängen führte, die aber trotzdem bessere Ergebnisse lieferte. Der Grund dafür liegt vor allem in der Wahl passender Parameter für die Zustandsübergänge.
Dieses ist das zweite Problem mit dem man beim Erstellen eines Behaviors konfrontiert wird. Oft kann man nur schätzen, bei welchem Parameterwert ein Zustandswechsel am sinnvollsten ist. Gut funktionierende Parameter lassen sich meist nur experimentell bestimmen. Jener Umstand macht den Unterschied zwischen einem guten und einem weniger guten Behavior aus. Während des Robocups 2008 wurden daher Konstanten in das Behavior eingebaut. Diese hatten nicht nur den Vorteil, dass mehrere Behaviorprogrammierer den gleichen Parametersatz verwendeten und damit eine bessere Zusammenarbeit zustande kam, sondern auch die Möglichkeit in kurzer Zeit ein Behavior mit mehreren unterschiedlichen Parametern zu testen. Dazu konnten die Konstanten zur Laufzeit des Frameworks verändert werden. Vor allem bei der Optimierung der Schussposition des Naos vor dem Ball kam diese Methode zum Einsatz.
Gerade bei der zentimetergenauen Positionierung vor dem Ball wurde eine weitere Schwierigkeit besonders deutlich. Durch das unvermeidbare Rauschen der Sensoren, d.h. das Schwanken der Sensorwerte um den wahren zu messenden Wert, kann man einen Zustandsübergang in einem State nicht auf einen bestimmten Wert festlegen. Dieser wird wegen des Rauschens nie exakt erreicht, wodurch es zu ungewollten Zustandsübergängen kommen kann. Im schlimmsten Fall erhällt man oszillierende Zustände, die ständig hin und her wechseln. Beim Positionieren des Naos äußerte sich dies dadurch, dass das Behavior die Position des Naos zum Ball als zu weit links bewertete, dann nach der Korrektur, zu weit rechts, um nach einer weiteren Korrektur wieder zu weit links zu stehen. Dieses Problem wurde gelöst, indem die Hystereseparameter erhöht wurden. Unter Hysterese versteht man Folgendes. Will man für einen bestimmten Wert in einem State bleiben, so testet man nicht auf genau diesen Wert, sondern auf einen Wertebereich, beispielsweise 10% mehr und weniger. Erst wenn dieser Wertebereich durch einen Sensorwert verlassen

wird, soll es zu einem Zustandswechsel kommen. Somit verhindert man, dass es schon für minimale Änderungen der zu überprüfenden Werte zu einem Zustandswechsel kommt. Gute Hystereseparameter zu finden ist meistens nur durch Experimente und viel Testen möglich, wie weiter oben schon einmal erwähnt. Diese Tests sind sehr zeitaufwändig und setzen gute Debugmöglichkeiten für ein Behavior voraus.

6.3 Debugging

Die Debugmöglichkeiten von XABSL sind zahlreich, wenn auch nicht immer ausreichend. Um sich einen ersten Eindruck von der Struktur des geschrieben Verhaltens machen zu können wurde der XABSL Editor verwendet. Dieser bietet nicht nur Syntax Highlighting und Autovervollständigung an, um häufig auftretende Tippfehler zu vermeiden, sondern visualisiert auch das aktuelle Verhalten, bzw. eine einzelne Option des Verhaltens als Graphen in einem extra Fenster. Somit bekam man schon während der Programmierung die Möglichkeit, falsche Zustandsübergänge zu erkennen und zu vermeiden. Um diese Funktionalität nutzen zu können muss das Adobe SVG Plugin auf dem Computer installiert sein, was aber für Windows Vista nicht mehr zur Verfügung steht. Eine zukünftige Nutzung des XABSL Editors könnte daher schwierig werden. Zusätzlich zur Editor Visualisierung wurde in regelmäßigen Abständen die *Behavior Documentation* kompiliert. Dieses erzeugt mehrere html Dateien, die alle Agents, Options, States, Symbols und Graphen enthalten, die so untereinander verlinkt sind, dass man sich eine gute Übersicht des Verhaltens ansehen kann.
War ein Verhalten fertig programmiert und kompiliert wurde auf eines der zwei Möglichkeiten sich den Ablauf des Verhaltens live anzeigen zu lassen zurückgegriffen. So wurde in SimRobot die Option genutzt sich zur Laufzeit den XABSL Dialog anzeigen zu lassen. Dieser kann über den TreeView geöffnet werden und enthält aktuelle Options und States, zusammen mit den darin jeweils gesetzten Output Symbols und genutzten Input Symbols. Zusätzlich wurden mit dem Befehl *xis InputVariablenName* oder *xos OutputVariablenName* weitere hilfreiche Variablen eingeblendet, oder mit dem Befehl *xo OptionName* eine bestimmte Option des Behaviors zu Testzwecken gestartet. Als hilfreich hat es sich erwiesen eine eigene Config Datei, die die Dateiendung .con hat, für den Simulator zu erstellen, die alle häufig benutzen Variablenaufrufe, wie den Abstand zum Ball oder den Winkel zum Tor, enthält und zu Debugzwecken geöffnet werden kann. Nachteilig an dieser Art des Debuggen war die Tatsache, dass sich die Zustände meist so schnell änderten, dass es schwierig war mitzubekommen, warum gerade zu einem bestimmten Zeitpunkt ein Zustandswechsel stattfand. Um dem entgegen zu wirken wurde versucht, bestimmte Situationen im Simulator nachzustellen und diese im Einzelschrittmodus anzusehen. Hilfreich wäre an dieser Stelle gewesen, wenn man eine log Datei des Verhaltens zur Laufzeit hätte aufnehmen können um sich diese später mit einem log aller Sensorwerte ansehen zu können. Leider kam diese Idee erst während des Robocups 2008 auf und wurde nicht mehr umgesetzt.
Eine weitere Debugmöglichkeit die benutzt worden ist, ist die XABSL Ansicht in RCXP. Nachdem man sich mit dem Roboter verbunden hat, sei es der echte Nao, oder ein Nao im Simulator, kann man sich im XABSL Viewer den Ablauf des gerade ausgewählten

Verhaltens ansehen. Wie in SimRobot werden die aktuellen States inklusive der Verweildauer in diesen angezeigt. Ein Vorteil von RCXP lag drin, dass bestimmte Variablen zur Laufzeit verändert werden konnten und somit das Verhalten des Roboters kurzzeitig zu Testzwecken verändert werden konnte. Dieser Umstand wurde vor allem beim Robocup 2008 genutzt, um auf schnelle Art den perfekten Schussabstand des Naos zum Ball experimentell zu bestimmen. Anstatt für verschiedene Abstandswerte das Verhalten jedes Mal abzuändern und neu zu kompilieren, wurden in RCXP die bereits oben erwähnten Debugsymbols für Konstanten in XABSL geändert und dadurch die Abschussentfernung variiert. In Kombination mit sehr gut erstellten Schuss Special Actions ergaben sich so die besten Torschüsse der Nao Liga.
Weitere Debugmöglichkeiten wurden kurz vor, bzw. beim Robocup hinzugefügt. Zum Einsatz kam in erste Linie die Soundausgabe für den realen Nao. Diese vereinfachte das Debuggen auf dem realen Nao erheblich, da es sehr schwierig war den Nao zu beobachten und gleichzeitig auf einen Bildschirm mit dem aktuellen Zustandswechsel zu achten. Durch die Soundausgabe konnte man sich auf den Nao konzentrieren und gleichzeitig mitbekommen, was das Behavior gerade tat. Zu diesem Zweck wurden Debugoptions in die oberste Verhaltensebene eingefügt, die allgemeine Ausgaben machten, wie den aktuellen Batteriestand anzusagen, aber auch speziellere, wie das Sehen des Balls zu melden. Zusätzlich wurden Soundausgaben in allen wichtigen States eingefügt. Eine Ansteuerung der LEDs des Naos wurde ebenfalls implementiert, um zusätzliches Feedback zu liefern. Alles in allem erleichterten die zusätzlichen Debugmöglichkeiten das Erstellen und Testen des Behaviors sehr. Zusammen mit den hervorragenden Special Actions ergab sich ein schönes und ausgewogenes Spielverhalten.

7 Debugging

Die entwickelte Debugsoftware soll das Arbeiten mit dem Roboter und die Entwicklung der Software erleichtern. Die Debugsoftware dient dazu den Roboter zu visualisieren und zu steuern. Zusätzlich können Daten und Informationen angezeigt werden. Dadurch kann die Funktionsweise und das Verhalten des Roboters analysiert werden. Dies hilft dabei die entwickelte Robotersoftware zu testen und Fehler zu finden. In diesem Kapitel werden die beiden Programme Robot Control XP und SimRobot vorgestellt.

7.1 RCXP

Robot Control [Spr06] ist ein Programm, das zur Datenvisualisierung, zum Debuggen und zum Steuern von Robotern eingesetzt wird. Es wurde vom *German Team* [Ger08] entwickelt und anfangs für Fußball spielende *Aibo* Roboter von *Sony* eingesetzt. Seit der ersten Version wurde Robot Control immer weiter ausgebaut und für nachfolgende Roboter modifiziert. Robot Control XP (RCXP) ist die aktuellste Version dieses Programms, das von der Projektgruppe 521 eingesetzt und erweitert wurde. RCXP besitzt eine grafische Benutzeroberfläche in der zahlreiche Tools geladen werden können. Diese geben dem User bzw. Programmierer hilfreiche Funktionen und Debug-Möglichkeiten, die nicht erst neu programmiert werden müssen. Wie z.B.:

- die Verbindung mit dem Roboter herstellen und überwachen
- Kamera Bilder und Debug-Images anzeigen
- den Roboter fernsteuern
- Bewegungen für den Roboter erstellen
- das Fußballfeld und Positionen von Roboter und Ball visualisieren
- Color Tables erstellen

Zur Implementierung dieser Funktionen werden Debug-Makros eingesetzt, die Daten und Nachrichten an das Framework schicken bzw. empfangen und Debug Informationen ausgeben. Dies ist eine Aufzählung einiger Makros:

- `OUTPUT(type, format, expression)`

  Die Daten aus dem Framework werden mit einem OUTPUT geschickt. Der Typ muss eindeutig sein und wird als „id“ verwendet um eine Nachricht zu finden und zu identifizieren. Als Format kann bin (Binärdaten) oder Text gewählt werden. Der letzte Parameter übergibt schließlich das Objekt, das verschickt werden soll.

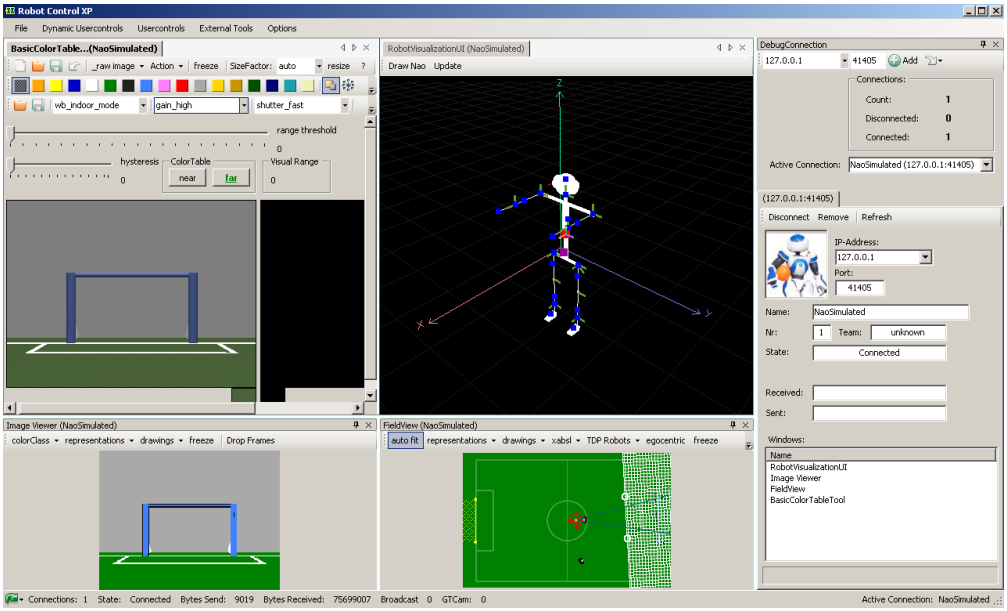


Abbildung 7.1: Benutzeroberfläche von Robot Control XP

- `DEBUG_RESPONSE(id, expression)`

  Damit das Framework auf eine Nachricht von RCXP reagieren kann, muss eine Debug Response definiert werden. Dadurch wird bei Empfang einer RCXP-Anfrage mit derselben „id“, der Code in „expression“ ausgeführt. Dies kann ein Makro wie z.B. das Output sein (siehe oben), oder beliebiger ausführbarer Code.

- `DebugRequests.Enable(requestString, bool enabled)`
  `DebugRequests.EnableOnce(requestString)`
  `DebugRequests.Disable(requestString)`

  In Robot Control muss ein Debug Request geschickt werden um Daten vom Framework zu empfangen. Es stehen Methoden zur Verfügung, um sich die Daten mehrmals oder nur ein einziges mal senden zu lassen, oder um das Senden zu deaktivieren. Hier dient der requestString als „id“ um die Nachricht im Framework zu finden.

- `DECLARE_DEBUGDRAWING(id, type, description)`

  Debug Drawings müssen über dieses Makro deklariert werden. Dazu wird eine „id“, ein „type“ (z.B. „drawOnFrontImage“ oder „drawOnField“ um zu bestimmen ob auf dem Kamerabild oder dem Feld gezeichnet wird) und eine Beschreibung angegeben.

- `CIRCLE(id, x, y, radius, penWidth, penStyle, penColor)`

  Dieses Makro zeichnet einen Kreis um den Ursprung (x, y) mit den Parametern für

den Radius, der Breite des Striches, dessen Typ und Farbe. Weitere geometrische Objekte wie Linien und Ellipsen können mit ähnlichen Makros gezeichnet werden.

Da die Entwicklung der Robotersoftware stets neue Debuggingmöglichkeiten erfordert, wurden im Laufe der PG521 neue Funktionen in RCXP eingefügt. Bei der Nutzung von RCXP wurden auch Fehler gefunden die behoben werden mussten. Einige bereits existierende Tools mussten zudem aktualisiert werden um mit der neuen Roboter Hardware und den Spielregeln des Robocup kompatibel zu sein. Im folgenden werden erst einige Fehlerkorrekturen erwähnt und dann die neuen Funktionen und Erweiterungen vorgestellt.

7.1.1 Fehlerbeseitigung in RCXP

YUYV-Image

Da sowohl die Kamera auf dem richtigen Roboter, als auch die „Kamera“ im Simulator ihre Bilder als YUYV-Bild liefern, müssen diese auch im gleichen Format zum *ImageViewer* und *BasicColorTableTool* geleitet werden. Außerdem wird in RCXP noch mit der Darstellung als aRGB-Bild gearbeitet, besonders für die Repräsentation von Farbklassen. Vor allem die Oberflächen in C# benutzen diese Darstellung standardmäßig, weshalb im *ImageManager* alle eingehenden Bilddaten in das aRGB-Bild umgerechnet wurden. Im *BasicColorTableTool* wiederum werden die *ColorTables* auf Basis von YUYV-Bildern erstellt, nur für Darstellung der segmentierten Bilder mit der Einteilung in die verschiedenen *ColorClasses* müssen diese YUYV-Bilder zu aRGB-Bildern konvertiert werden.
Da jedoch die Bilder im *BasicColorTableTool* nicht als YUYV-Bilder ankamen, sondern schon in aRGB umgerechnet waren, kam es hier zu Fehlern bei der Segmentierung, die *ColorClasses* wurden nicht richtig gesetzt und erkannt.
Durch Einführung einer doppelten Datenhaltung im *ImageManager*, also YUYV-Bild und aRGB-Bild wird dieses Problem gelöst. Zur Anzeige auf den Oberflächen in RCXP müssen die Bilder immernoch in die aRGB Darstellung umgerechnet werden, aber die Bilder auf denen die Segmentierung stattfindet sind im YUYV-Format. Sowohl die Umrechnungsfehler, als auch das falsche Bildformat sind behoben.

BasicColorTableTool

Beim Überprüfen der YUYV-Images im *BasicColorTableTool* fiel auf, dass zwar die Daten von der Kamera jetzt richtig in RCXP verarbeitet und auch gespeichert werden, aber dass die Interpretation der Farben als *ColorClasses* nicht richtig funktioniert. Auf der Benutzeroberfläche wurden viele Farben angeboten, welche dann in der Datenhaltung auf einige wenige Farben reduziert wurden. Außerdem waren einige Farben falsch gemappt. Diese Inkonsistenzen wurden durch das Einführen neuer *ColorClasses* in der Datenhaltung sowie das richtige Mappen der Buttons der Oberfläche auf die jeweilige *ColorClass* gelöst. Damit ist das Erstellen von korrekten *ColorTables* nun möglich.
Für die schnelle und komfortable Erstellung einer *ColorTable* sind die Hotkeys des *BasicColorTableTools* unabdingbar. Bei einer Funktionsüberprüfung wurde festgestellt, dass

die Hotkeys um die Frames des *Logplayers* einzeln durchzuschalten nicht richtig funktionieren. Diese Hotkeys wurden mit den richtigen Funktionen hinterlegt. Außerdem wurde die Funktionsweise so ergänzt, dass der *LogPlayer* jetzt auch das Abspielen der Logdatei stoppt, wenn man ein Bild *vor* oder *zurück* schaltet und nicht wie früher endlos weiterläuft. Auch wurden neue Hotkeys eingefügt, mit denen es möglich ist, den *Logplayer* ganz zu Stoppen oder Weiterlaufen zu lassen. Für die Undo-Funktion des *BasicColorTableTools* wurde auch ein weiterer Hotkey ergänzt.
Die Undo-Funktion des *BaisColorTableTools* wies Fehler auf. Zum einen wurde die *ColorTable* nicht auf den richtigen Ausgangspunkt vor der letzten Änderung zurückgesetzt, zum anderen wurde die Anzeige des *ColorClassImages* nicht richtig aktualisiert. Beide Fehler wurden behoben, außerdem wurde noch eine Überprüfung eingebaut, ob überhaupt Arbeitsschritte zurückgenommen werden können, weil das Fehlen selbiger in Ausnahmefälle zu einem Fehler geführt hatte.
Weiter wurde auch der DropFrames-Button auf der Oberfläche mit einer Funktion versehen, welche die gleiche Funktionsweise hat wie im ImageViewer.

ucDebugData

Nach einer Umstellung der Repräsentation der *JointAngles* von *double* auf *float* im Framework wurde festgestellt, dass *ucDebugData* diese Werte nicht interpretieren kann. Daher musste der *Tokenizer*, der die Nachrichten, welche vom Framework zu RXCP geschickt werden, als Byte-Stream ausließt, erweitert werden, damit sowohl einfache *float*-Daten, als auch komplexe Datenstrukturen vom Typ *float* im Byte-Stream erkannt werden und richtig zusammengesetzt werden. Beim Einfügen des Typs *float* wurde festgestellt, dass auch weitere Datentypen wie u.a. *char* nicht vom *Tokenizer* bearbeitet werden können. Um all diese fehlenden Datentypen wurde daher der *Tokenizer* umfangreich ergänzt. Um den fehlerfreien Nachrichtenverkehr nicht nur in der Richtung Framework zu RCXP sicherzustellen, musste auch der *OutputTokenizer* um die gesammte Menge der neuen Datentypen ergänzt werden.

LogPlayer

Der LogPlayer in RCXP wird unter anderem dazu benutzt das Kamerabild des Roboters zu loggen und die Bilder zu speichern. Dazu werden die einzelnen Frames in eine Logdatei gespeichert, aus der einzelne BMP Dateien extrahiert werden können. Das extrahieren dieser BMP Dateien funktionierte jedoch nicht, da das Umwandeln von YUV zu RGB fehlerhaft war. Durch die Implementierung einer neuen Funktion zur Umwandlung von YUYV in RGB innerhalb des LogManagers, ist der Fehler behoben. Diese Umwandlungen sind identisch zu denen die auch im ImageViewer vorhanden sind. Dadurch ist es wieder möglich die einzelnen Frames aus der log Datei zu extrahieren und als BMPs zu speichern.

FrameInfo

Einige Funktionen in RCXP benötigen zu jeder empfangenen Nachricht eine Frame-Nummer um diese Nachricht zeitlich einordnen zu können (siehe Value History Viewer in Unterkapitel 7.1.2). Diese Frame-Nummer wurde mittels einer eigenen FrameInfo-Message vom Framework geschickt. Tools die diese Info benötigen hatten jedoch nicht richtig funktioniert. Dies lag daran, dass die FrameInfo-Nachricht nicht korrekt gesendet wurde. Der Fehler lag daher nicht an RCXP sondern am Framework. Dort hat die Modifikation der FrameInfo-Representation das Problem gelöst.

ImageViewer

Die fehlende Funktion des DropFrames-Button wurde im *ImageViewer* ergänzt. Da es bei großen Logdateien oder bei der permanenten Betrachtung der Livebilder der Roboterkamera zu einer asynchronen Darstellung der Bilder auf der Oberfläche gekommen ist. Da die Bilder Zeile für Zeile ausgelesen werden und dieses Verfahren sehr langsam ist, wurde eine Funktion eingefügt um die Bilder, die zwischen dem aktuell auf der Oberfläche angezeigten Bild und dem wirklich aktuellen Bild im Puffer sind, zu überspringen. Hierzu wurde auf die *FrameInfo* des aktuellen Bildes zurückgegriffen und jedes Bild mit einer alten Nummer einfach aus dem Puffer geholt und nicht weiter bearbeitet, also verworfen. Durch dieses Auslassen der nicht „live“ anzeigbaren Bilder wurde die flüssige Darstellung der Logdateien und der Kamerabilder verbessert.

7.1.2 Erweiterungen für RCXP

Robot Visualization UI

Wie in Kapitel 3 beschrieben, wurde ein Modell zur Selbstrepräsentation des Roboters erstellt. Dieses sogenannte Egomodell stellt Daten zur Verfügung die vor allem von der Walking-Engine benötigte werden. Diese sind neben dem Zero Moment Point (ZMP) und den Center of Mass (COM) der einzelnen Roboter Parts, auch Positionen und Rotationsachsen der Gelenke.
Da die Berechnung dieser Werte mathematisch anspruchsvoll sein kann, und sich dabei schnell Fehler einschleichen können, war es nötig eine Visualisierung dieser Daten zu entwickeln. Solch eine Visualisierung sollte es ermöglichen das Egomodell bzw. die in ihm gespeicherten Werte, auf einfache und schnelle Art anzeigen und überprüfen zu lassen. Dazu wird eine dreidimensionale Visualisierung des Roboters erstellt die auf einem 3d Roboterskelett basiert. Anhand dieses Modells sollten die im Egomodell gespeicherten Positionen der Gelenke und die COMs der Parts angezeigt werden.
Dazu wird in RCXP ein EgoModellManager benutzt, der die Daten vom Egomodell im Framework abfragt, speichert und aktualisiert. Diese Daten werden von der RobotVisualisation Klasse gelesen und visualisiert um in RCXP angezeigt zu werden. Hierbei wird die OpenGL-API eingesetzt um das 3d-Modell in Echtzeit darzustellen. Zur Zeichnung des Modells, werden in einem 3d-Koordinatensystem die Positionen der Gelenke, die vom Egomodell stammen, eingezeichnet und verbunden. Dadurch erhält man ein

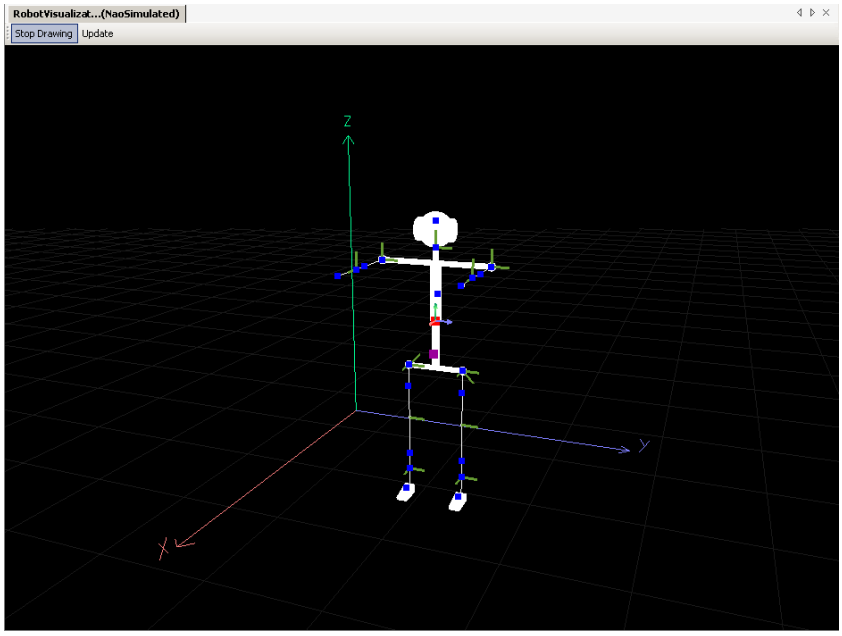


Abbildung 7.2: Benutzeroberfläche der Robot Visualization UI

Roboterskelett das dem realen Roboter ähneln muss, vorasugesetzt die Werte im Egomodell sind richtig. Außerdem werden die Massenschwerpunkte der Roboter Parts und die Rotationsachsen der Gelenke auf diesem Skelett eingezeichnet.

Anhand dieser Visualisierung können Fehler in den Daten des Egomodells gefunden werden. Da das Model bei Veränderung der Daten im Egomodell aktualisiert wird, kann auch die Bewegung des Roboters verfolgt werden. Bei der Arbeit mit dem realen Roboter hilft die Robot Visualisation UI dabei, Fehler in der Berechnung einiger Gelenkrotationen zu finden. Auch defekte Gelenke des Roboters können identifiziert werden, da die Werte aus dem EgoModell, nicht mit denen des realen Roboters übereinstimmen würden.

In Zukunft kann dieses Tool erweitert werden um weitere Informationen wie z.B. Drehgeschwindigkeiten und Bewegungsrichtungen zu visualisieren.

Footsensorviewer UI

Da der Nao über Fussdrucksensoren verfügt, wurde auch hierfür eine Visualisierung in RCXP erstellt. Diese Visualisierung stellt die einzelnen Sensorwerte der acht Fussdrucksensoren der Nao-Füße dar. Die Darstellung der Druckstärke, die von den Sensoren jeweils gemessen wird, wurde in OpenGL vorgenommen, um ein „Flackern“ zu vermeiden, welches bei einer Darstellung auf der normalen Oberfläche durch die Veränderung der Werte der Druckstärken in hohen Frequenzen auftreten würde. Der jeweilige Druck pro Sensor wird durch einen Kreis dargestellt. Je höher der Druck an einem Messpunkt ist, desto größer ist der Radius des jeweiligen Kreises. Die einzelnen Kreismittelpunkte

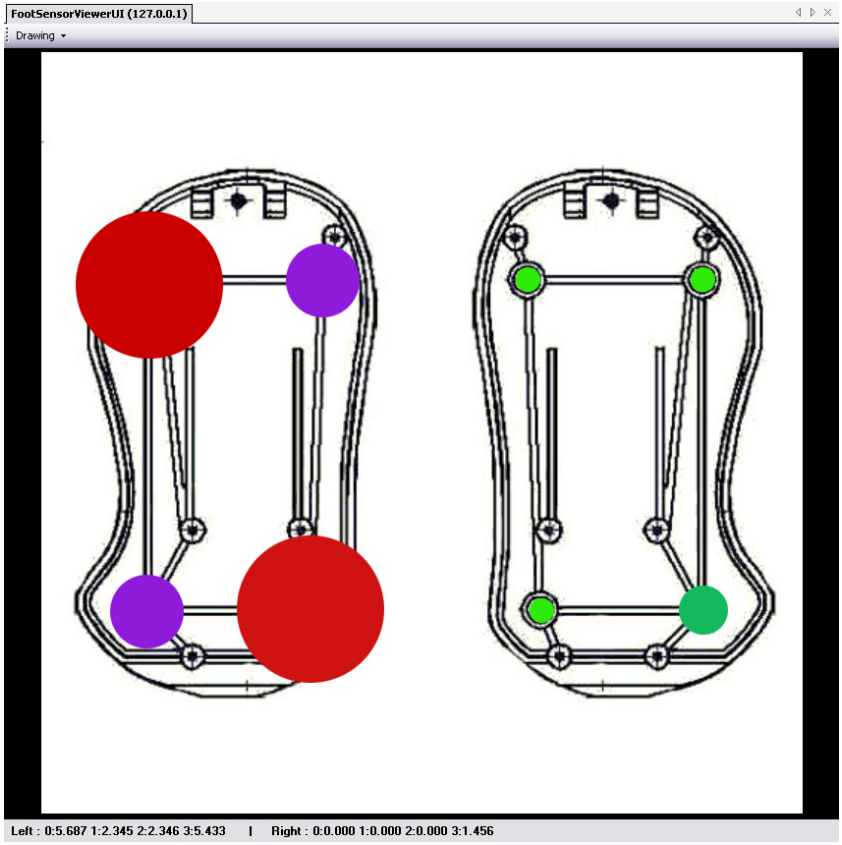


Abbildung 7.3: Benutzeroberfläche des FootSensor Viewer

wurden auf einem Querschnitt, welcher die beiden Füsse des Naos von oben zeigt, exakt auf den realen Einbaupositionen der Sensoren positioniert. Es wurde sichergestellt, dass selbst bei maximalem Druck die Kreise für die einzelnen Sensoren nicht überlappen. Weiter wurde durch eine geeignete Wahl der farblichen Füllung der Kreise, von *grün - niedriger Druck, kleiner Radius* zu *rot - hoher Druck, großer Radius*, die jeweilige Druckstärke verdeutlicht.
Als Ergänzung wurde eine Statuszeile eingefügt, in der die jeweiligen absoluten Werte alle acht Sensoren als Zahlen dargestellt werden.

Footstep Viewer

Um die Fussschritte des Roboters zu visualisieren wurde der Footstep Viewer erstellt. Dieser stellt die einzelnen Schritte der Füsse als eine Projektion auf eine 2D Ebene dar. Dazu musste erstmal im Framework der *PatternGeneratorOutput* erstellt werden, der die nächste Position des *rechten* und *linken* Fusses mit der *x- und y-Koordinate* und der jeweiligen *Rotation* ausgibt, welche im nächsten Schritt angefahren werden soll. Außerdem wird der dafür im PatternGenerator *verwendete ZMP* mit *x- und y-Koordinate*

ausgegeben. Anhand dieser Daten wurde in RCXP eine Repräsentation erstellt, der PatternGenManager, welche die letzten zwanzig Schritte, zehn Schritte pro Fuss, aufzeichnet und die letzten hundert Schritte des verwendeten ZMP, da dieser schnelleren Veränderungen ausgesetzt ist. Die einzelnen Schritte bestehen aus den Koordinaten im Roboterkoordinatensystem mit der jeweiligen Rotation, ebenso die Koordinaten und die Rotation für den ZMP.
Diese Schritte der Füße werden, ausgehend vom ersten in RCXP empfangenen Schritt, auf die 2D Ebene gezeichnet. Der erste Schritt wird hierbei als zentraler Punkt benutzt und genau in die Mitte der Ebene gezeichnet, von diesem Punkt aus werden alle anderen Schritte relativ dazu gezeichnet. Der verwendete ZMP wird auf die gleiche Weise eingezeichnet, also auch relativ zum ersten Schritt. Da sich der Wert des ZMP sehr schnell verändert, wurde hier auf OpenGL 2D gesetzt, um ein „flackern“ der Darstellung bei schnell springenden Werten zuvermeiden. Die einzelnen ZMP Werte werden der Reihe nach durch eine Linie verbunden, um den Verlauf der Werte geeignet darzustellen.
Weiter wird ein Raster über die gesamte Ebene gelegt, um Entfernungen und „gerades“ Laufen besser kenntlich zu machen. Zur besseren Unterscheidung der einzelnen Schritte der beiden Füsse, werden diese in zwei verschiedenen Farben, *blau* und *orange*, dargestellt. Auch der ZMP hat eine eigene Farbe, *grün*. Da es bei langsamer Schrittfolge zu einer Überlappung der Auftrittflächen der einzelnen Füsse kommen kann und diese dadurch kaum unterscheidbar sind, wurde die Möglichkeit ergänzt, von einer rechteckigen Darstellung der Füsse auf eine Darstellung als Kreuz, oder eine Darstellung als Vektoren umzustellen. Dadurch sind auch kleine Schritte gut differenzierbar.
Zur Verbesserung der Übersichtlichkeit wurde eine Zoom-Funktion per Mausrad eingeführt, damit man zwischen detailierter oder genereller Übersicht wechseln kann. Auch kann über die Maus die gesamte Ebene in x- oder y-Richtung verschoben werden und damit ein anderer Ausschnitt ins Sichtfeld bewegt werden. Da es dabei passieren kann, dass die aktuellen Schritte aus dem Sichtfeld „rutschen“ existiert ein Menübutton, der den Bildausschnitt auf die aktuellen Fussschritte, die aktuell eingefügt werden, zentriert. Um die Fussschritte nicht aus den Augen zu verlieren wurde auch ein Button eingefügt, durch den der Bildausschnitt immer auf die aktuellen Veränderungen zentriert wird, also mit diesen mit aktualisiert wird.

Um die korrekte Auslenkung des verwendeten ZMPs zu überprüfen, wurde die Visualisierung um die Möglichkeit erweitert, die einzelnen ZMPs aus dem Egomodell in die gleiche Ebene einzublenden. Dafür musste auch hier eine kleine neue Repräsentation der Daten aus dem Egomodell eingefügt werden, welche die jeweiligen letzten 100 Werte der einzelnen ZMPs des Egomodells zwischenspeichert. Diese ZMPs kann man einzeln über das Menü zu- und wegschalten. Auch gibt es die Möglichkeit die Füsse oder den verwendeten ZMP über dieses Menü zu- und wegzuschalten, falls dieses gewünscht wird. Zur Unterscheidung der ZMPs wurden auch hier verschiedene Farben für die jeweilige Darstellung benutzt.

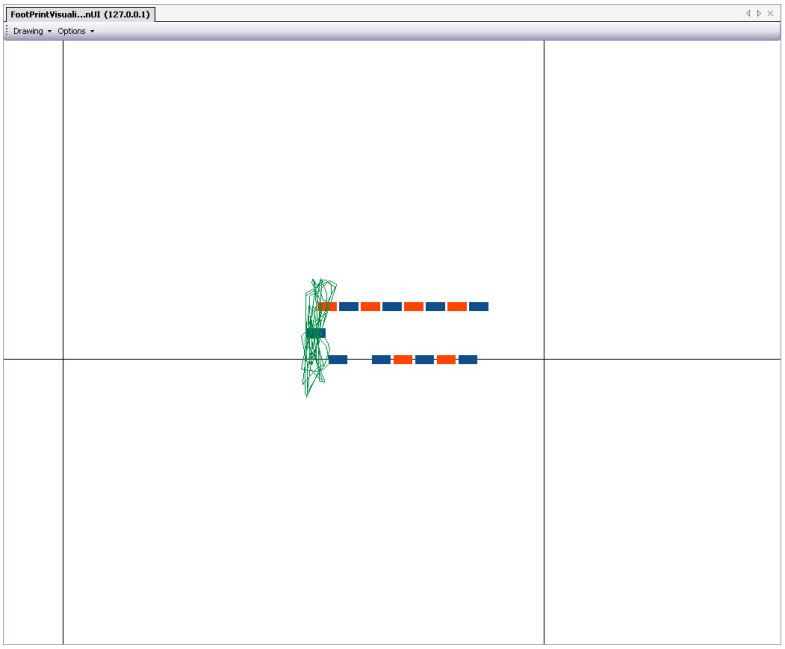


Abbildung 7.4: Benutzeroberfläche des FootPrint Viewer

Xabsl Viewer

Um das Debuggen des Verhalten zu vereinfachen, wurde eine Visualisierung für die Xabsl-Zustandsdiagramme in RCXP integriert. Diese Visualisierung zeigt eine Liste alle vorhandenen Xabsl-Dateien an. Hierfür wird der Config-Ordner des Frameworks mit den Xabsl-Dateien nach den richtigen Dateien durchsucht und diese werden in die Liste eingefügt. Von dieser Ebene des Dateisystems wird jeder Unterordner rekursiv durchlaufen und auch hier werden die enthaltenen Xabsl-Dateien gesucht und unter Ergänzung des Unterordnernamens in die Liste eingefügt.

Wird eine dieser Xabsl-Dateien aus der Liste ausgewählt, so wird das Zustandsdiagramm dafür erstellt und in die Oberfläche gezeichnet. Hierbei werden alle Daten aus der gewählten Xabsl-Datei geparst. Die Zustände werden als Boxen (Knoten) dargestellt. In diesen Boxen werden die Entscheidungen der einzelnen Zustände, die Aktionen und weitere Informationen angezeigt. Die Zustandsübergänge werden als gerichtete Kanten dargestellt. Weiter wurde ein Button eingefügt, der es erlaubt auf eine komplexere Ansicht des Diagramms zu wechseln. Diese enthält alle Informationen in einer vollen Darstellung. Außerdem werden die Kanten zwischen den Boxen an der jeweiligen Anzahl ausgerichtet, damit keine Überlappungen mehr entstehen, welche bei sehr großen Dateien vorkommen können. Diese Darstellung braucht mehr Vorberechnungen und ist daher langsamer.
In beiden Darstellungsmöglichkeiten werden für die Kanten verschiedene Farben verwendet um die Übersichtlichkeit zu erhöhen. Auch die Informationen in den einzelnen Zuständen werden durch verschiedene Farben und durch ein strukturiertes Layout her-

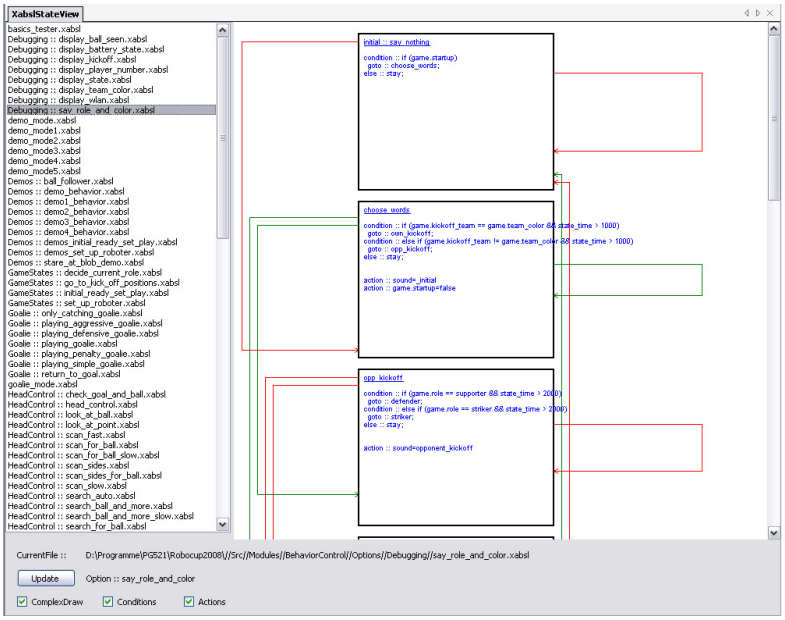


Abbildung 7.5: Benutzeroberfläche des Xabsl State View

vorgehoben.

Value History Viewer

Der Value History Viewer dient dazu Debug Daten, die vom Framework über `DebugRequests` angefordert werden, in einem Diagramm zu visualisieren. Die Debug Werte (Y-Achse) werden dem Zeitpunkt ihres Auftretens (X-Achse) gegenübergestellt. Damit die Visualisierung sowohl optisch ansprechend ist, als auch Funktionen zur Handhabung der geplotteten Daten einfacher zu implementieren waren, wurde die API „ZedGraph“ benutzt. „ZedGraph“ ist eine umfangreiche Programmbibliothek, die bereits Methoden zur Skalierung von Graphen, das Neuzeichnen der Achsen nach dem Einfügen von neuen Daten, Zoom Funktionalität und weitere wünschenswerte Eigenschaften zur Verfügung stellt. Sobald mit dem Roboter oder mit dem Simulator eine Verbindung aufgebaut wurde kann der Value History Viewer aufgerufen werden. Es wird nun der Server der Debug Daten dazu aufgefordert alle verfügbaren Debug Daten dem Client (RCXP) zu senden.

Aus dieser ersten Anforderung wird nun eine Baumstruktur im Value History Viewer erstellt die zur Auswahl der gewünschten Debug Daten dient. Die gewünschten Daten werden visualisiert, sobald die entsprechende Checkbox in der Baumstruktur ausgewählt wurde. Abbildung 7.6 zeigt das entsprechende Szenario. Für das Debugging wurden zwei Werkzeuge implementiert, die zur leichteren Untersuchung der Daten dienen. Zum Ersten kann man das Update der Daten pausieren. Zum Zweiten können die erhaltenen Daten über den Save Schalter in eine csv Datei exportiert werden. Vom Value History Viewer

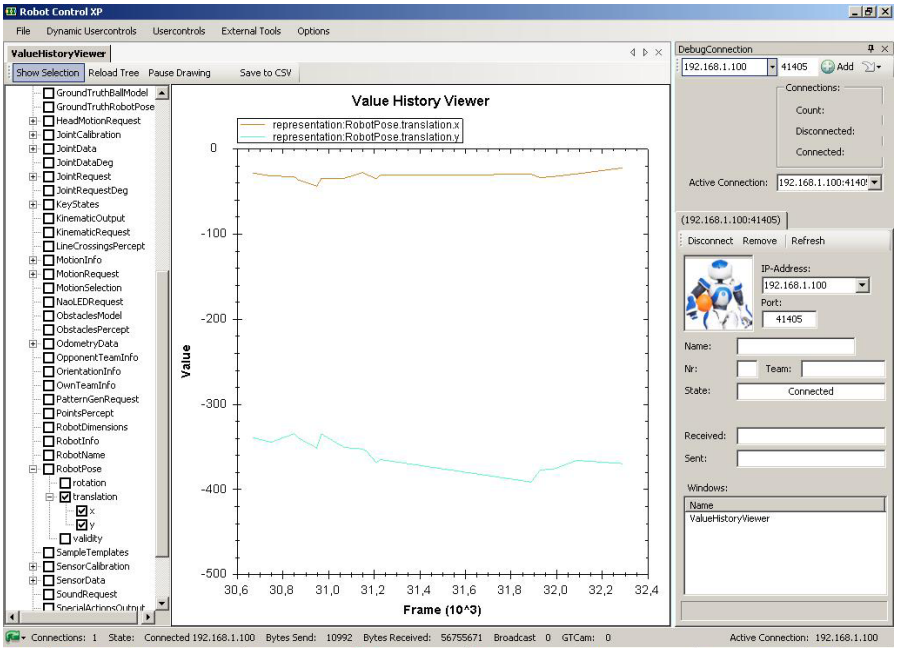


Abbildung 7.6: Benutzeroberfläche des Value History Viewer

werden zum plotten Fließkommazahlen einfacher und doppelter Genauigkeit, ganzzahlige vorzeichenbehaftete sowie vorzeichenlose Werte und Boolean unterstützt. Entspricht ein angeforderter Debug Wert nicht diesem Typus erscheint eine Fehlermeldung, die den Benutzer auf diesen Umstand hinweist („ungültiger Datentyp“).

7.2 Erweiterungen am Simulator

Der verwendetet Simulator wurde vom Bremer Team B-Human entwickelt und von uns zusammen mit dem Framework übernommen. Die grundlegenden Funktionen wurden beibehalten, es wurden aber auch einige Erweiterungen implementiert.

7.2.1 Sensoren

In der ursprünglichen Version waren keine/wenige simulierte Sensoren verfügbar. Daher wurden von uns zu Anfang Beschleunigungs- und Gyroskopsensoren eingebaut, später auch die Fußdrucksensoren (s. 2.2). Eine Sensorart ohne Entsprechung im realen Roboter sind die COM-Sensoren (Center of Mass), mit denen zu Debugzwecken die absolute Position im Raum gemessen werden kann.

7.2.2 Erweiterung der Oberfläche

Hier wurde vor allem die Möglichkeit geschaffen, die Szene in mehreren 3D-Fenster aus unterschiedlichen Perspektiven gleichzeitig zu betrachten. Dies hat sich bei Arbeiten in

verschiedenen Bereichen als nützlich erwiesen, zumal dies auch die Möglichkeit bietet, die Szene mit verschiedenen Darstellungseigenschaften gleichzeitig zu betrachten, z.B. als physikalisches Wireframe-Modell und mit Oberflächendarstellung.

7.2.3 Videoaufzeichnung

Da die Simulationsgeschwindigkeit je nach Rechner und Genauigkeit der Physiksimulation sehr niedrig werden kann und dadurch eine Abschätzung der tatsächlichen Bewegung sehr erschwert wird, wurde die Möglichkeit eingebaut, den simulierten Ablauf als Videodatei aufzuzeichnen, um diese später in Realzeit betrachten zu können.
Wurde die Aufzeichnung gestartet, so wird in jedem Frame der Inhalt des 3D-Fensters als Bild ausgelesen. Diese Bilder werden dann mit Hilfe der Bibliothek *Revel* ([Str04]) zu einem Video zusammengesetzt.

8 Fazit

Die wichtigsten Ziele in der Projektgruppe 521 waren das Thema Laufen und die Teilnahme am Robocup 2008 in Suzhou. Für das Laufen wurde eine Walking Engine entwickelt, die sowohl auf die geforderte Dynamik aus den Sensoren eingehen kann, als auch hinreichend schnell und dennoch stabil ist, um damit Fussball zu spielen. Mit dieser Walking Engine wurde auch der Robocup 2008 bestritten, wo im Vergleich zu den anderen Teams, wovon die meisten auf einen parameteroptimierten Standard Walk von Aldebaran gesetzt haben, eine sicherer aber dennoch schneller Lauf Vorteile verschaffte. Leider blieb man hierbei unter den Möglichkeiten, da zum Beispiel die Arme durch wiederkehrende Defekte außer Funktion gesetzt wurden. Auch andere Probleme, wie fehlerhafte Colortables oder sich ändernde Lichtverhältnisse durch neu geöffene Fenster über dem Spielfeld, aber vor allem Probleme beim Verhalten haben ein besseres Abschneiden verhindert. Insgesamt aber war der Robocup auch ohne Titelgewinn ein Erfolg, da immerhin das erste Tor der neuen Standard-Plattform-Liga geschossen wurde und auch der Achtelfinaleinzug.
Mit dem Ego Modell wurde eine neue Repräsentationsgrundlage für die aktuellen Parameter des Roboters und seiner Sensoren gefunden und diese effizient umgesetzt. Diese Repräsentation ermöglicht den einfachen Zugriff auf alle wichtigen Daten, die zur Steuerung und Kontrolle des Roboters nötig sind. Außerdem dient sie als universelle Schnittstelle zwischen den Modulen.
Außerdem wurde das Framework an die neue Hardwaregrundlage, den NAO, angepasst. Hierbei hat sich herausgestellt, dass sich die einzelnen Roboter durchaus unterscheiden und eventuell die Sensorwerte bei jedem Roboter einzeln überprüft und eingestellt werden müssen.
Die Erweiterungen am Simulator und vor allem an RCXP eröffnen neue Möglichkeiten der Kontrolle und des Debuggings. So können über RCXP alle aktuellen Daten des Roboters ausgelesen und geeignet visualisiert werden, was die Fehlersuche und die Überprüfung von Daten erleichtert. Die Beseitigung der Fehler hat dazu beigetragen, dass sinnvolle alte Funktionen des Tools wieder nutzbar sind, vor allem verlässlich fehlerfrei nutzbar.
Leider haben wir erst spät mit der Entwicklung des Verhaltens für die Roboter angefangen. Gerade bei diesem testintensiven Thema wurden wir sehr oft durch defekte Roboter zurückgeworfen, vor dem Robocup hatten wir nie die Möglichkeit ein richtiges Testspiel durchzuführen. Auch haben komplexe Verhaltensstrukturen keine Test oder gar Anwendungsmöglichkeit gefunden, da die Zeit vor und auf dem Robocup dafür zu knapp war um Fehler durch komplexes Testen zu beseitigen.
Für die SpecialActions haben wir diese Zeit zum Glück noch zwischendurch irgendwie, besonders nachts auf dem Robocup, gefunden. Wodurch wir zu einem sehr guten, auf den jeweiligen Roboter optimierten Schuss gekommen sind.

Insgesamt wurde mit dem dynamischen Lauf, dem Schuss und den weiteren Entwicklungen, wie das EgoModel, wie die Sensoranbindung und Kalibrierung und den Erweiterungen und Fehlerbeseitigungen im Framework und in den Tools wie RCXP, eine gute Grundlage geschaffen um mit dem NAO Fussball zu spielen. Verbesserungspotential gibt es in allen Bereichen, da die anderen Teams bis nächstes Jahr bestimmt auch einen dynamischen Lauf hinbekommen, aber vor allem beim Roboter selber, da einfach zu viele Ausfälle der Hardware zu verzeichnen waren, was das Testen und das Debuggen sehr erschwert hat.

9 Ausblick

Die zurückliegende Entwicklung hat mehrere Möglichkeiten zur weiteren Verbesserung offenbart. Zunächst kann das Egomodell erweitert werden. Hier wären die Optimierung der Drehmomentberechnung oder die Verbesserung der ZMP-Berechnung durch Fusion der Modelle oder deren Verfeinerung zu nennen. Weiterhin bietet die Bewegungssteuerung noch Potential. Mit Hilfe eines Damping Controllers können Abweichungen der Gelenkansteuerung kompensiert werden. Auch wird eine dynamische Berechnung die Stabilität und Genauigkeit der Kickbewegung erhöhen. Beim Verhalten ist eine Verbesserung der Ausrichtung zum Tor und zum Ball nötig. In Zukunft wird mit erhöhter Spielerzahl eine stärkere Kooperation zwischen den Spielern und die Umsetzung sowie Verbesserung der weiteren Rollen nötig sein. Hierzu werden sicherlich auch Verbesserungen an der Lokalisierung und der Bildverarbeitung erforderlich sein.

A Nao Remote System Control

A.1 Einleitung

Um die Arbeit mit den Nao-Robotern zu vereinfachen und Fehler durch Hektik und Stress zu vermeiden wurde Nao Remote System Control (kurz Nao-RSC) geschrieben. Nao-RSC ist ein kleines Java-Programm, das auf normale Shell-Scripte zurückgreift um Standard-Aufgaben bei der Arbeit mit dem Roboter zu automatisieren. Es bietet ein einfache Oberfläche um viele, mit dem Linux-System auf dem Roboter zusammenhängende Vorgänge, per Kopfdruck zu erledigen. Zum Beispiel sind dies: Transfer einzelner Dateien oder ganzer Config-Teile auf den Roboter oder von diesem zurück, Bearbeiten von Einstellungen (Roboternummer, Location, etc.) oder Start, Stop und Überwachung von NaoQi.
Nao-RSC ist modular aufgebaut und kann so problemlos erweitert werden. Hierbei teilen sich alle Teilkomponenten die zentrale Verbindung zum Roboter über SSH.

A.2 Installation

Unter Windows müssen zunächst die verwendeten Kommandozeilen-Tools installiert werden. Dies geschieht am einfachsten durch Installation des Nao-RSC Paketes aus dem Cygwin-Repository, das alle nötigen Unix-Tools, die von Nao-RSC verwendet werden, als Abhängigkeiten enthält. Unter Linux gehören die Programme ssh, bash, sed, awk, dos2unix, find auf den meisten Systemen zur Standart Konfiguration. Sollten sie nicht vorhanden sein, müssen sie über den jeweiligen Paketmanager nachinstalliert werden.
Weiterhin muss sichergestellt werden, dass sich Nao-RSC ohne Benutzereingabe mit dem Roboter verbinden kann. Hierzu ist es nötig sich einmal mittels SSH auf Kommandozeile mit dem Roboter zu verbinden und den Host-Key zu bestätigen, damit während der Verwendung von Nao-RSC diese Abfrage nicht sichtbar die Ausführung blockiert. Es ist zu beachten, dass dies für jeden Roboter und jede IP-Addresse des Roboters einmal getan werden muss.
Danach kann Nao-RSC über die Scripte startNao-RSC.bat bzw. unter Linux startNao-RSC.sh gestartet werden.

A.3 Bedienung

Nach dem Programmstart öffnet sich ein Fenster, das in mehrere Tabs für die einzelnen Teilbereiche unterteilt ist. In der aktuellen Version sind dies die 4 folgenden:

- Verbindung
- Console

- Einstellung
- NaoQi

Nach dem Start ist immer das Verbindungs-Tab zu sehen, da Nao-RSC erst mit einem Roboter verbunden werden muss, um die anderen Funktionen nutzen zu können. Im Folgenden wird die Arbeit mit den einzelnen Teilen kurz erläutert.

A.3.1 Verbindung

Nao-RSC öffnet im Hintergrund zwei Bash-Shells, eine auf dem lokalen Rechner und eine über ssh auf dem Roboter, um Kommandos auszuführen. Diese können mit den Feldern Lokale Verbindung und Roboter Verbindung beinflusst werden. Diese sind Abbildung A.1 zu erkennen.

Abbildung A.1: Verbindungs-Tab von Nao-RSC

Der aktuelle Status der jeweiligen Shell wird in Textform angezeigt. Über die Connect-Knöpfe kann dieser geändert werden. Es zu beachten, dass einige Funktionen nur in bestimmten Zuständen zur Verfügung stehen. Im nicht verbundenen Zustand erreicht man mit dem jeweiligen Setup-Knopf einen Dialog, in dem bestimmt werden kann, welche Verzeichnisse Nao-RSC verwenden soll und über welche Addresse es sich mit dem Roboter verbindet (vgl. Abb. A.2).
Als Basis-Pfad muss der vollständige Pfad zu dem Verzeichniss, in dem auch das Config-Verzeichniss usw. liegt eingetragen werden. Beim dem Hostnamen muss häufig auch der Benutzer mit eingetragen werden, da sonst versucht wird die Verbindung mit dem aktuellen Benutzer aufzubauen, der auf den Naos nicht existiert. Beispielsweise: root@nao07.

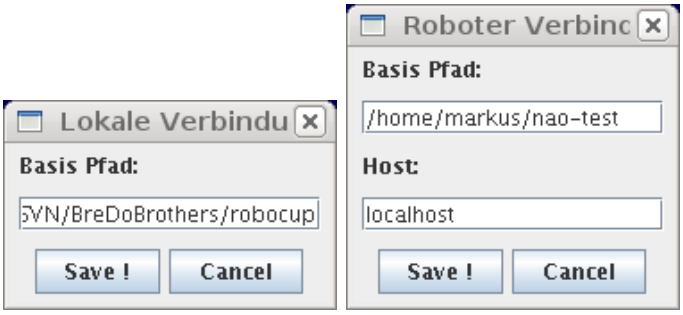


Abbildung A.2: Verbindungseinstellungen von Nao-RSC

A.3.2 Console

Der Consolen-Tab ermöglich es selbst Befehle einzugeben und die Meldungen der ausbeführten Befehle zu sehen.

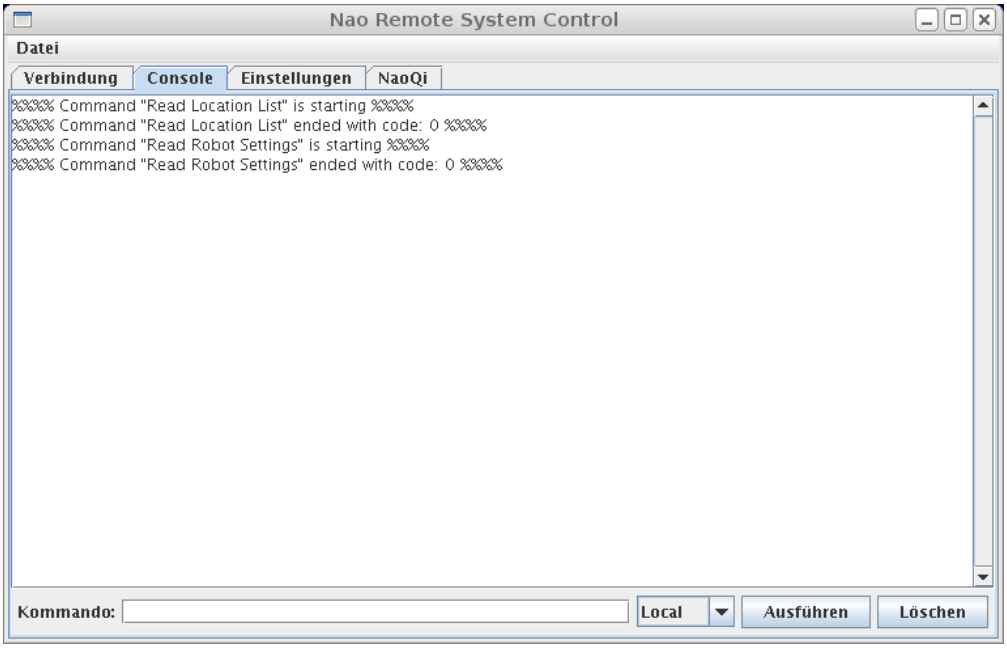


Abbildung A.3: Console-Tab von Nao-RSC

Wie auf ABbildung A.3 zu erkennen ist, nimmt die Ausgabe den oberen Teil ein, während in das untere Eingabefeld Befehle eingegeben werden können, die mittels Eingabe-Taste oder Betätigen des Äusführen Knopfes ausgeführt werden.

A.3.3 Einstellungen

Dieser Tab ermöglicht es Einstellungen des Framework direkt zu bearbeiten bzw. sie zwischen dem lokalen Rechner und dem Roboter zu transferieren.

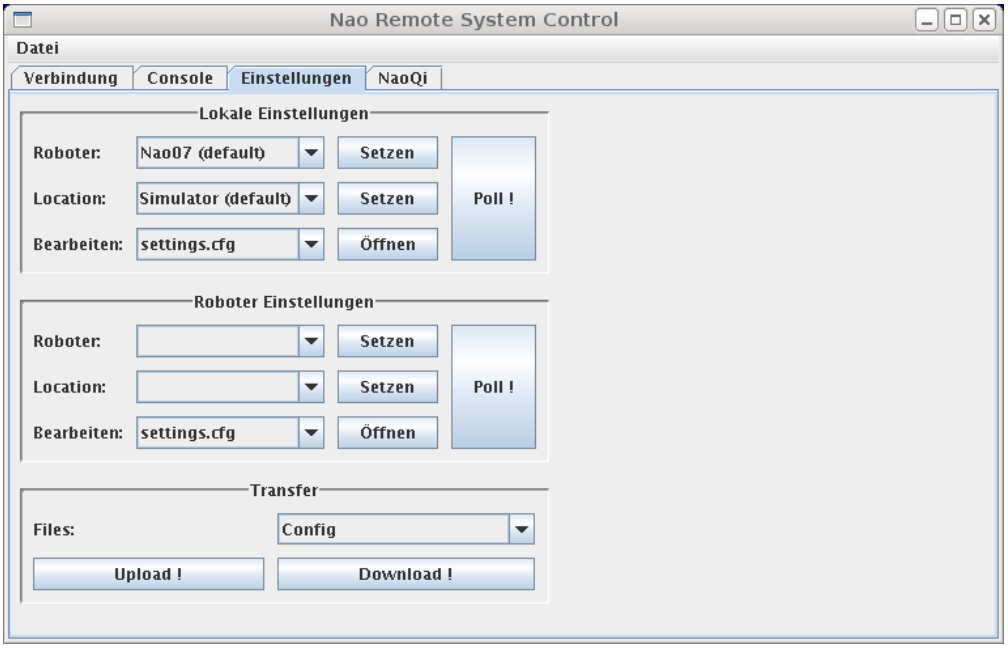


Abbildung A.4: Einstellungen-Tab von Nao-RSC

Abbildung A.4 stellt die verschiedenen Bereiche dieses Programmteils dar. Es existieren jeweils Bereiche für die Dateien auf dem lokalen Rechner und dem Roboter sowie ein Transfer Bereich. Die Auswahlfelder werden zu Anfang leer sein, da die nötigen Informationen erst nach betätigen des "Poll Knopfes eingelesen werden, da dies einen Moment benötigen kann, je nach Geschwindigkeit des verwendeten Rechners. Die Location und der verwendete Roboter können direkt ausgewählt und in die Config übernommen werden. Für die anderen aufwendigeren Optionen öffnet sich jeweils nach Auswählen der entsprechen Datei und betätigen des "Öffnen Knopfes ein angepasster Editordialog. In Abbildung A.5 ist beispielsweise der Dialog für die settings.cfg dargestellt.
Um Einstellungsdateien zu übertragen muss im Transferbereich ausgewählt werden, in welchem Umfang dies gesehen soll und danach der Upload- oder Download-Knopf betätigt werden. Gegebenenfalls öffnet sich dann ein Dialog, der weitere benötigte Informationen abfragt.

A.3.4 NaoQi

Hier sind alle Funktionen gesammelt, die mit NaoQi zu tun hab. Starten und Stoppen des selbigen, Statusabfrage, Änderung der verwendeten Module etc. Die Möglichkeiten sind in Abbildung A.6 gut zu erkennen.

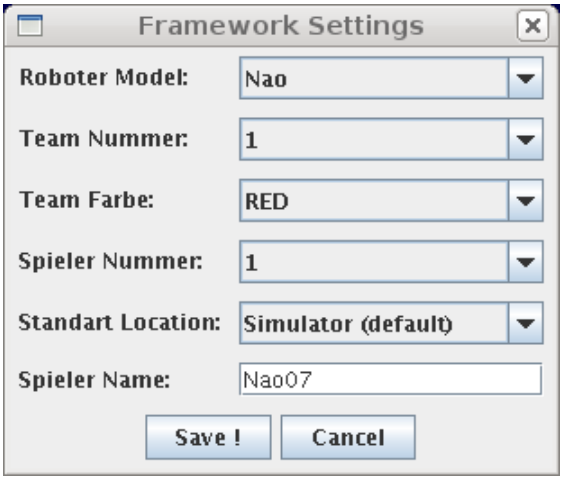


Abbildung A.5: settings.cfg Editor von Nao-RSC

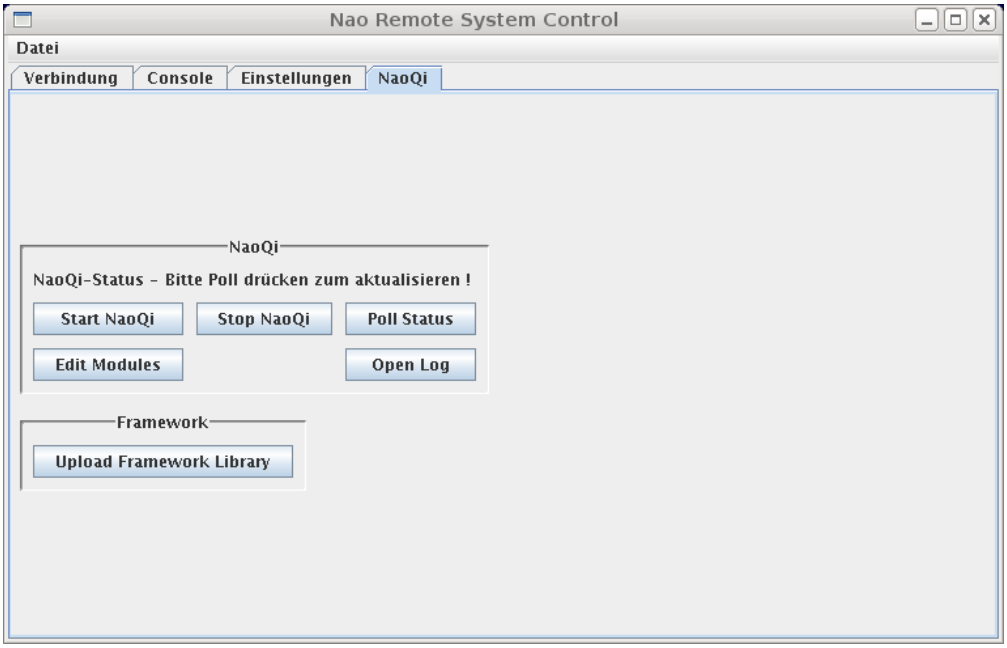


Abbildung A.6: NaoQi-Tab von Nao-RSC

Der untere Teil ermöglicht es ein selbst kompiliertes Library-Modul für NaoQi auf den Roboter zu laden. In einem Dialog werden dann die gefunden Module gezeigt und es kann ein Name festgelegt werden, bevor die Übertragung erfolgt.
Im oberen Teil ist die Statusanzeige zu finden, die über den Poll Status-Knopf aktualisiert werden kann. Möchte man die verwendeten Module bearbeiten oder betrachteten, öffnet sich ein Dialog (Abb. A.7) in dem aus den zur Verfügung stehenden ausgewählt werden

kann.

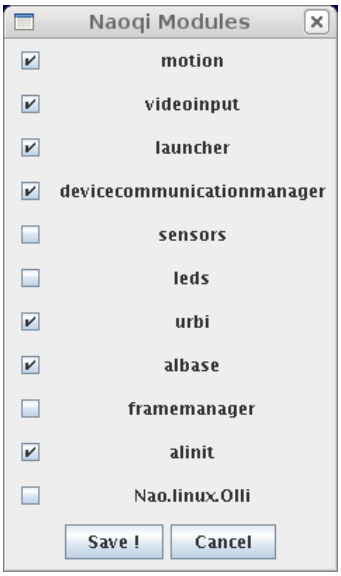


Abbildung A.7: NaoQi-Modules in Nao-RSC

Der Logviewer zeigt jeweils die aktuelle Logdatei von NaoQi an, kann sie aber nicht laufend aktualisieren, so dass hier ein neues Öffnen des Viewers nötig ist.

B Kinematik des Nao

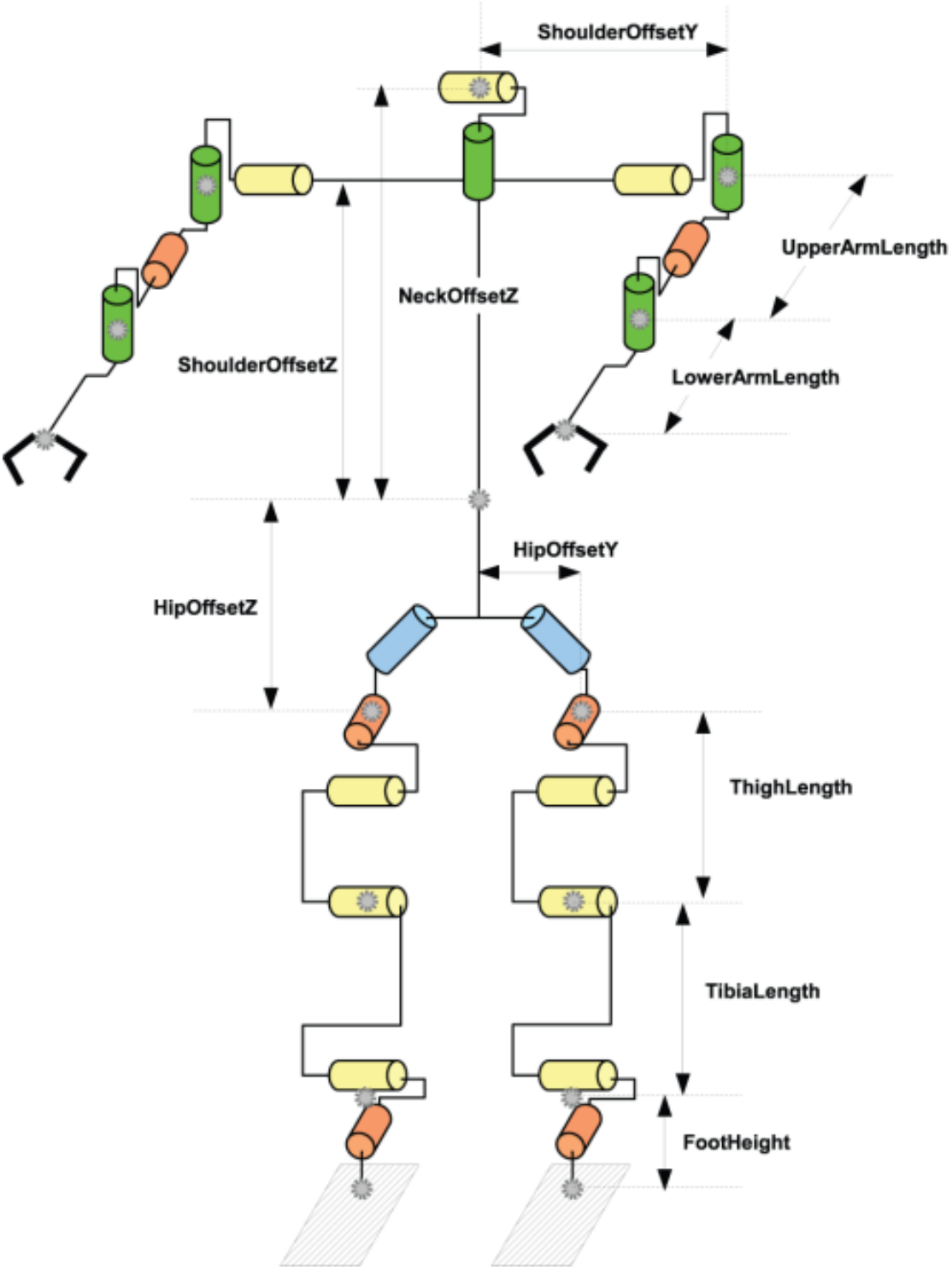


Abbildung B.1: Kinematischer Aufbau des Nao. Quelle: [Ald08b]

C Special Actions im Überblick

Da in der Zeit der PG521 zu verschiedenen Entwicklungszeitpunkten des Projektes Special Actions entwickelt wurden, kann es auf Grund von inzwischen geänderten Gelenkkonfigurationen dazu kommen, dass einige Special Action nicht mehr ausgeführt werden können. Deshalb sind getestete oder sich im Einsatz befindende Special Action in dieser Übersicht mit * gekennzeichnet.

Special Action	**Beschreibung**
stand*:	Roboter geht in einen aufrechten Stand.
pitch*:	Roboter geht in die Beuge, um den Boden vor seinen Füßen zu sehen.
standUpBack:	Aufstehen des Roboters aus der Rückenlage.
standUpFront:	Aufstehen des Roboters aus der Bauchlage.
Cheering*:	Jubelpose
goalkeeperDefend:	Roboter geht in die Knie und breitet die Arme neben den Beinen aus. Special Action für den Torwart.
goalkeeperDefendLeft:	Roboter geht in die Knie und beugt sich dabei nach links. Der linke Arm steht dabei senkrecht zum Boden. Special Action für den Torwart
goalkeeperDefendRight:	Roboter geht in die Knie und beugt sich dabei nach rechts. Der rechte Arm steht dabei senkrecht zum Boden. Special Action für den Torwart
goToLeft:	Roboter führt seitlichen Schritt nach links aus.
goToRight:	Roboter führt seitlichen Schritt nach rechts aus.
kickBallToLeft:	Der Roboter legt den Ball, mit dem rechten Bein, nach links.

Tabelle C.1: entworfene Special Action im Zeitraum 2007/08

Special Action	Beschreibung
kickBallToRight:	Der Roboter legt den Ball, mit dem linken Bein, nach rechts.
kickLeft:	Der Roboter führt mit dem linken Bein eine Schußbewegung nach vorn aus.
kickLeftFast:	Der Roboter führt nur unter Verwendung des Unterschenkels mit dem linken Bein eine Schußbewegung nach vorn aus.
kickRight:	Der Roboter führt mit dem rechten Bein eine Schußbewegung nach vorn aus.
kickRightFast:	Der Roboter führt nur unter Verwendung des Unterschenkels mit dem rechten Bein eine Schußbewegung nach vorn aus.
turnInPlaceToLeft:	Der Roboter dreht sich auf der Stelle um etwa 30 Grad nach links.
turnInPlaceToRight:	Der Roboter dreht sich auf der Stelle um etwa 30 Grad nach rechts.
turnToLeft90:	Der Roboter dreht sich auf der Stelle um etwa 90 Grad nach links.
turnToRight90:	Der Roboter dreht sich auf der Stelle um etwa 90 Grad nach rechts.
sitBackAndWait:	Der Roboter geht in die Knie und bleibt dort.
falling*:	Alle Gelenke werden abgeschaltet.
kickLeft_V2:	Der Roboter führt mit dem linken Bein eine Schußbewegung nach vorn aus. Das Schusstiming für den Unterschenkel ist hier auf 200ms gesetzt.

Tabelle C.1: entworfene Special Action im Zeitraum 2007/08

Special Action	**Beschreibung**
kickLeft_V25:	Der Roboter führt mit dem linken Bein eine Schußbewegung nach vorn aus. Das Schusstiming für den Unterschenkel ist hier auf 250ms gesetzt.
kickLeft_V5:	Der Roboter führt mit dem linken Bein eine Schußbewegung nach vorn aus. Das Schusstiming für den Unterschenkel ist hier auf 500ms gesetzt.
kickLeft_No07*:	Der Roboter führt mit dem linken Bein eine Schußbewegung nach vorn mit maximaler Geschwindigkeit aus.
kickLeft_No31*:	Der Roboter führt mit dem linken Bein eine Schußbewegung nach vorn mit maximaler Geschwindigkeit aus. (bremer Roboter)
kickLeft_No33*:	Der Roboter führt mit dem linken Bein eine Schußbewegung nach vorn mit maximaler Geschwindigkeit aus.
kickLeft_No52*:	Der Roboter führt mit dem linken Bein eine Schußbewegung nach vorn mit maximaler Geschwindigkeit aus. (bremer Roboter)
kickRight_V5:	Der Roboter führt mit dem rechten Bein eine Schußbewegung nach vorn aus. Das Schusstiming für den Unterschenkel ist hier auf 500ms gesetzt.
kickRight_No07*:	Der Roboter führt mit dem rechten Bein eine Schußbewegung nach vorn mit maximaler Geschwindigkeit aus.
kickRight_No31*:	Der Roboter führt mit dem rechten Bein eine Schußbewegung nach vorn mit maximaler Geschwindigkeit aus. (bremer Roboter)
kickRight_No33*:	Der Roboter führt mit dem rechten Bein eine Schußbewegung nach vorn mit maximaler Geschwindigkeit aus.
kickRight_No52*:	Der Roboter führt mit dem rechten Bein eine Schußbewegung nach vorn mit maximaler Geschwindigkeit aus. (bremer Roboter)

Tabelle C.1: entworfene Special Action im Zeitraum 2007/08

D Technische Daten

ALDEBARAN Robotics

Nao®, The new Robocup standard league platform.

- Compact and lightweight humanoids
- Fully programmable & easy to operate
- Bipedal architecture with 21 DOF
- Multiple sensors
- Programming capacities and remote control
- x86 AMD Geode 500 Mhz CPU
- 256 Mo SDRAM / 1 Gb Flash memory
- Wifi 802.11g and ethernet port
- 30 FPS CMOS videocam res. 640x480
- Vision processing capacities
- Two loudspeakers and english vocal synthesis
- Friendly design and optional colours

NAO Robocup Edition

Technical specification for Robocup edition only

Non contractual photo / Aldebaran Robotics

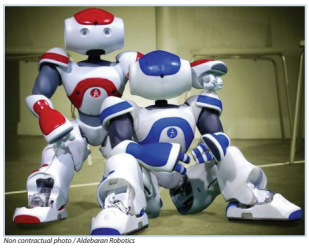
A high performance biped robot.

Nao® stands tall in all points amongst its robotic brethren : this unique robot can be controlled and programmed using linux, windows and Mac OS. Nao® can produce very precise movements with fluidity and dynamism. This concentration of high technology can be throughly customised to your specific needs. With its fonctionnality, its flexibility, and its communication system, this robot will satisfy both professional and home users

Non contractual photo / Aldebaran Robotics

With Nao®, a new stage has been reached

This robot is the fruit of Aldebaran Robotics ingeneering team's intensive research, and goes on the spirit of the objectives defined by the Dr Hiroaki Kitano, funder of the Robocup competition.
With the planned confrontation between human and robots set for 2050, Nao®



Non contractual photo / Aldebaran Robotics

brings a biped to the Standard League for the first time.

Nao® will allow teams to program multiple strategies in order to make passes, to find good attacks and to choose the right speed and angle needed to score.

Movements

The 21 degrees of freedom allow Nao to have a great mobility in its environnement.

The inertial unit combined with the FSRs located under each foot gives the robot great stability.

The sonar sensors enables Nao® to avoid obstacles.

His revolutionary proprietary actuators, gives Nao extreme precision in its movements.

Programming capacities

The computer architecture is based on Linux (Linux 32 bit x86 ELF; using a custom OpenEmbedded based distribution with a real-time patched 2.6.22.9 kernel, and gcc-4) and a modular system allowing the user, depending on their expertise, to control Nao® at different levels : either programming in C++ or URBI (or other script languages), or optionaly through Choregraphe, our user friendly motion editor, simple simulator and graphical programming interface.

Besides these possibilities, robocuppers will get an API with low level access to sensors and actuators, and can, if they wish, replace our code with custom adaptations.

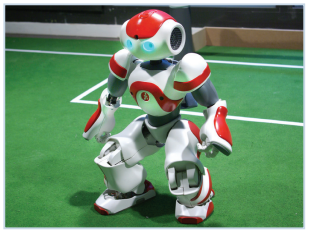
Finally, a free simulator based on Webots/Urbi is provided, specifically adapted to robocuppers.

Interactions

Nao® has embeded software modules allowing text to speech, sound localization, visual pattern and colouredshape detection, obstacle detection (based on the two chanel sonar system) and visual effects or communication through the many LEDs.

Body and multimedia

Nao® can be personnalised with different colors (red, blue or customized colours...). Integrated multimedia components (Hi-Fi speaker system, microphone, video camera) allows many different possibilities like adding voice and face recognition programs or playing music.

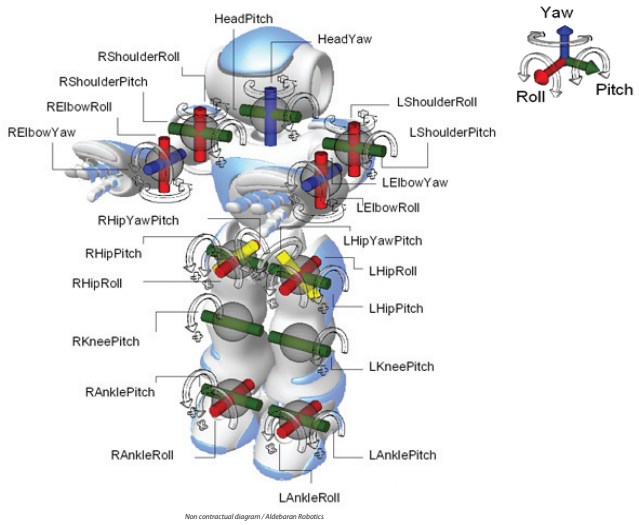


Non contractual photo / Aldebaran Robotics

The design of Nao® is the property of Aldebaran robotics.

Kinematics

The scheme underneath presents all the robot's axes. Together, these axes allow 21 degrees of freedom, which when coupled with the inertial sensor, the force sensitive resistors, the hall effect sensors, the infrared receiver and the sonar sensors, allows Nao® a high level of stability and fluidity in its displacements.



Motion range

PART	JOINT NAME	MOTION	RANGE (degrees)
Head	HeadYaw	Head joint twist (Z)	-120 to 120
	HeadPitch	Head joint front & back (Y)	-45 to 45
Left arm	LShoulderPitch	Left shoulder joint front & back (Y)	-120 to 120
	LShoulderRoll	Left shoulder joint right & left (Z)	0 to 95
	LElbowRoll	Left shoulder joint twist (X)	-120 to 120
	LElbowYaw	Left elbow joint (Z)	0 to 90
Left leg	LHipYawPitch	Left hip joint twist (Z45°)	-90 to 0
	LHipPitch	Left hip joint front & back (Y)	-100 to 25
	LHipRoll	Left hip joint right and left (X)	-25 to 45
	LKneePitch	Left knee joint (Y)	0 to 130
	LAnklePitch	Left ankle joint front & back (Y)	-75 to 45
	LAnkleRoll	Left ankle joint right & left (X)	-45 to 25
Right leg	RHipYawPitch	Right hip joint twist (Z45°)	-90 to 0
	RHipPitch	Right hip joint front and back (Y)	-100 to 25
	RHipRoll	Right hip joint right & left (X)	-45 to 25
	RKneePitch	Right knee joint (Y)	0 to 130
	RAnklePitch	Right ankle joint front & back (Y)	-75 to 45
	RAnkleRoll	Right ankle right & left (X)	-25 to 45
Right arm	RShoulderPitch	Right shoulder joint front & back (Y)	-120 to 120
	RShoulderRoll	Right shoulder joint right & left (Z)	-95 to 0
	RElbowRoll	Right shoulder joint twist (X)	-120 to 120
	RElbowYaw	Right elbow joint (Z)	-90 to 0

General caracteristics

Body caracteristics

Height	~ 57 cm
Weight	~ 4.5 Kgs
Body type	Technical plastic

Energy

Battery type	Lithium-ion
Charger	AC 90-230 volts/DC 24 volts adapter
Battery capacity	55 Wh (~45 min. autonomy)

Degrees of freedom

Head	2 DOF
Arms	4 DOF in each arm
Pelvis	1 DOF
Leg	5 DOF in each leg

Audio

Speakers	2 Loudspeakers
Microphones	2 Microphones

Network access

Connections type	- Wifi (IEE 802.11g) - Ethernet connection

Actuators

Aldebaran Robotics original design based on :
- Hall Effect sensors
- dsPICS microcontrollers
- Coreless MAXON DC motors

Sensors

Different type	32 x hall effect sensors 8 x FSR 2 x gyro meters 3 x accelerometers 2 x bumpers 2 channels sonar

LED

Eyes	2 x 8 LED RGB Fullcolor
Ears	2 x 10 LED 16 Blue levels
Torso	1 LED RGB Fullcolour
Feet	2 x 1 LED RGB Fullcolour

Mother board

- x86 AMD GEODE 500MHz CPU
- 256 Mo SDRAM / 1 Go flash memory

Software

OS	- Embedded Linux (32 bit x86 ELF) using custom OpenEmbedded based distribution
Programming	- URBI, C, C++

Motors data sheet

Nao® is equiped with two different motors types with the followings caracteristics :

Motor Type 1

No Load Speed	8000 RPM
Stall Torque	59.5 mNm
Nominal Speed	6330 RPM
Nominal Torque	12.3 mNm
Reduction ratio type 1	201.3
No Load Speed	238.45 °/s (4.76°/20ms)
Stall Torque	11.97 Nm (without the ratio efficiency)
Nominal Speed	188.67 °/s (3.77°/20ms)
Nominal Torque	2.47 Nm (without the ratio efficiency)
Reduction ratio type 2	130.85
No Load Speed	366.83 °/s (7.33°/20ms)
Stall Torque	7.78 Nm (without the ratio efficiency)
Nominal Speed	290.25 °/s (5.80°/20ms)
Nominal Torque	1.61 Nm (without the ratio efficiency)

Motor Type 2

NoLoad Speed	11900 RPM
Stall Torque	15.1 mNm
Nominal Speed	8810 RPM
Nominal Torque	3.84 mNm
Reduction ratio Type 1	150.27
No Load Speed	473.72 °/s (9.47°/20ms)
Stall Torque	2.27 Nm (without the ratio efficiency)
Nominal Speed	351.77 °/s (7.03°/20ms)
Nominal Torque	0.57 Nm (without the ratio efficiency)
Reduction ratio Type 2	173.22
No Load Speed	412.19 °/s (8.24°/20ms)
Stall Torque	2.61 Nm (without the ratio efficiency)
Nominal Speed	305.16 °/s (6.10°/20ms)
Nominal Torque	0.66 Nm (without the ratio efficiency)

All specifications are not contractual and are subject to change.

Copyright

Aldebaran Robotics
robocup@aldebaran-robotics.com
Tel : + 33 1 77 371 751
Fax : +33 1 77 352 268

ALDEBARAN Robotics - SAS au capital de 1 080 000 € ; Siège social : 126, boulevard Saint Germain - 75006 Paris
RCS Paris B 483 185 807 - TVA intracommunautaire FR06483185807

Ref. 261107 - cfRCup vers. 2.0

Abbildungsverzeichnis

Literaturverzeichnis

[Ald08a] Aldebaran Robotics: *DCM Documentation.* 2007/2008

[Ald08b] Aldebaran Robotics: *Nao Ware Documentation.* 2007/2008

[CKU08] Czarnetzki, Stefan ; Kerner, Soeren ; Urbann, Oliver. *Observer-based Dynamic Walking Control for Biped Robots.* 2008

[Ger08] GermanTeam: GermanTeam. In: *http://www.germanteam.org/* (2008)

[KKK+03] Kajita, Shuuji ; Kanehiro, Fumio ; Kaneko, Kenji ; Fujiwara, Kiyoshi ; Yokoi, Kazuhito ; Hirukawa, Hirohisa: Biped walking pattern generation by a simple three-dimensional inverted pendulum model. In: *Advanced Robotics* 17 (2003), Nr. 2, S. 131–147

[Kos06] Kosse, Ralf: *Planung und Implementierung eines evolutionären Ansatzes zur Steuerung eines zweibeinigen Roboters.* Dortmund, Technische Universität Dortmund, Diplomarbeit, September 2006

[LJRK08] Loetzsch, Martin ; Jüngel, Matthias ; Risler, Max ; Krause, Thomas: XABSL The Extensible Agent Behavior Specification Language. In: *http://www2.informatik.hu-berlin.de/ki/XABSL/index.html* (2008)

[PBP02] Prautzsch, Hartmut ; Boehm, Wolfgang ; Paluszny, Marco: *Bezier and B-spline techniques.* Springer, 2002

[PG407] PG499: Biped soccer robots: Development of an universal robotic software and hardware architecture / IRF. 2007. – Forschungsbericht

[Rie92] Rieseler, Harald: *Roboterkinematik - Grundlagen, Invertierung und symbolische Berechnung.* Verlag Vieweg, 1992 (Fortschritte der Robotik 16)

[Spr06] Spranger, Michael: *An Architecture supporting Research and Education in Robotics*, Humboldt-Universität Berlin, Diplomarbeit, 2006

[Str04] Stratton, Cort. *http://revel.sourceforge.net/.* 2004