Laufplanung basierend auf realitätsnahen Umgebungsdaten für einen sechsbeinigen Laufroboter

Diplomarbeit

Institut für Robotik der Fakultät für Informatik
an der
Hochschule Mannheim

vorgelegt von
Uli Ruffler

Datum: 2. August 2006
Betreuer: Prof. Dr. Thomas Ihme
Zweitkorrektor: Prof. Dr. Steffen Rasenat

Eidesstattliche Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Stellen, die wörtlich oder sinngemäß aus veröffentlichten und nicht veröffentlichten Schriften entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit hat in dieser oder ähnlicher Form keiner anderen PrÜfungsbehörde vorgelegen.

Mannheim, den 2. August 2006

Uli Ruffler

Zusammenfassung

Der von André Herms für die Simulationsumgebung implementierte Laufplanungsalgorithmus plant die Fußauftrittspunkte für einen sechsbeinigen Roboter. Dazu verwendet er Geländeinformationen die in Form einer Höhenkarte vorliegen. Der Algorithmus plant dabei die gesamte Wegstrecke. Darüberhinaus steht ein Stereobildverarbeitungsalgorithmus zur Verfügung der in der Lage ist aus einem Bildpaar Tiefeninformationen zu rekonstruieren.

Um den Laufplanungsalgorithmus auf einem Roboter verwenden zu können, muss dieser aufeinanderfolgend ein kurzes Teilstück des gesamten Weges planen. In dieser Arbeit wurde der Laufplanungsalgorithmus auf eine inkrementelle Arbeitsweise modifiziert. Darüberhinaus wurde der Stereobildverarbeitungsalgorithmus in die Simulationsumgebung integriert, jedoch verwendet der Laufplaner die Tiefeninformationen bislang noch nicht.

Um ein inkrementelles Planen zu ermöglichen, müssen die geplanten Teilstrecken aneinandergehängt werden. In dieser Arbeit werden dazu die aufeinanderfolgenden Roboterkonfigurationen verkettet und der Bewegungsablauf des Roboters rekonstruiert. Der Stereobildverarbeitungsalgorithmus wurde als separates Modul integriert um diesen auslagerbar zu machen.

Die Geschwindigkeit des in dieser Arbeit modifizierten Algorithmus wurde gemessen, mit dem Ergebnis, dass im Mittel 6.25 Lösungen gefunden werden bis der Roboter in ein neues Teilstück eintritt. Dabei verbessert der Algorithmus seine Lösung im Schnitt 1.16 mal. Das Verfahren zeigte, dass es die Terrains Flachland, Stufe, Graben, Steg, Doppelsteg, große Löcher und zufälliges Gelände bewältigt, jedoch an den Szenarien Hindernis und kleine Löcher scheitert.

Danksagung

Ich möchte mich an dieser Stelle bei den Personen bedanken, die mich, im Rahmen der Entstehung meiner Diplomarbeit, mit Rat und Tat unterstützt haben.

Mein Dank gilt besonders Herrn Prof. Dr. Thomas Ihme für seine Unterstützung und Betreuung während der gesamten Dauer meiner Diplomarbeit. Danken möchte ich auch Herrn Stephan Platzek, der sich während meiner Diplomarbeit mit der Integration der Physics-Engine in die Simulationsumgebung beschäftigt hat. Dank gebührt auch Herrn Kai Wetzelsberger, der mir stehts bei Verständnisfragen zum Roboter und zum MCA Framework beiseite stand.

Bedanken möchte ich mich auch bei meiner Lebensgefährtin Susann Werner, für ihr Verständnis und ihre Hilfsbereitschaft in dieser Zeit. Außerdem danke ich meiner Schwester Simone Ruffler für das Korrekturlesen dieser Arbeit.

Widmen möchte ich diese Arbeit meinen Eltern, Elisabeth und Uwe Ruffler, die mich im Verlauf meines Studiums stets unterstützt haben.

Mannheim, den 2. August 2006

Uli Ruffler

Inhaltsverzeichnis

Kapitel 1

Einleitung

Mobile Roboter gewinnen immer mehr an Bedeutung, da sie auch in unwirtlichen Umgebungen, an denen ein Mensch nicht überleben kann, eingesetzt werden können. Ein Beispiel für solche Umgebungen sind Terrains nach Kernkraftwerkunglücken. Aber auch Orte an denen es für einen Menschen schlicht zu gefährlich ist, beispielsweise die Bergung von Opfern nach Erdbebenkatastrophen sind typische Einsatzgebiete für autonome Roboter. Doch nicht nur in gefährlichen Situationen, ja sogar in Alltagssituationen wie im Servicebereich sind Roboter von Interesse. So rollen in einigen Krankenhäusern schon heute autonome Wägelchen durch die Flure, um selbständig Transportaufgaben zu übernehmen.

Die schreitende Fortbewegung nimmt dabei eine Sonderstellung unter den Fortbewegungsarten ein. Sie ermöglicht das Manövrieren auch in zerklüftetem Gelände wo für rollende Fahrzeuge bereits kein Durchkommen mehr möglich wäre. Darüber hinaus ist die schreitende Fortbewegung wesentlich weniger belastend für den Untergrund. So werden für Forstarbeiten oftmals immer noch (oder besser wieder) Pferde eingesetzt, um die schweren Stämme aus dem Wald zu ziehen. Hier könnten in Zukunft Roboter die Arbeit der Pferde übernehmen und ebenso bodenschonend größere Lasten in kürzerer

Zeit bewegen. Dass es ein wirtschaftliches Interesse für solche Roboter gibt sieht man an Produkten wie der "Walking Forest Machine" (siehe Abbildung 1.1, des finnischen Unternehmens PlusTech[1].

Abbildung 1.1: Die "Walking Forest Machine" im Einsatz.

Wissenschaftler und Ingeneure haben aus diesem Interesse heraus die Gangart von Lebewesen untersucht. Die hexapode Gangart stellte sich als eine der am einfachsten zu kopierenden Arten heraus, da bei dieser das Balancieren entfallen kann solange mindestens drei Füße festen Halt haben. Mit mehr Beinpaaren steigt die Komplexität der Koordination rapide an.

Die indische Stabheuschrecke zählt zu einem der am besten auf ihre Fortbewegungsweise hin untersuchten Lebewesen, daher ist die Anatomie einiger

[1]PlusTech ist inzwischen ein Tochterunternehmen von John Deere.

Laufroboter ihrer Anatomie entlehnt. So auch die, des im Jahre 1992 am Forschungszentrum für Informatik (FZI) in Karlsruhe entwickelten LAURON [4] (siehe Abbildung 1.2). LAURON ist die Abkürzung für **LAU**fender **RO**boter **N**euronal gesteuert [12], bzw. im Englischen **L**egged **AU**tonomous **RO**bot **N**eural controlled [6], auch wenn seine Steuerung längst nicht mehr neuronal ist.

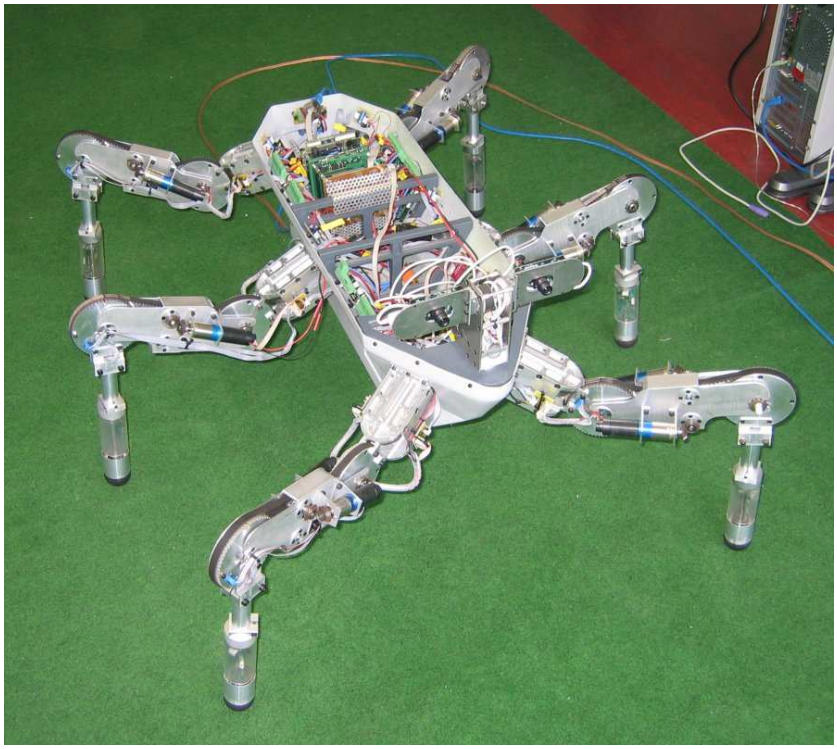

Abbildung 1.2: Der LAURON IVb des Instituts für Robotik der Hochschule Mannheim

1.1 Motivation

Das Institut für Robotik an der Hochschule Mannheim schaffte sich zur Forschung an Laufalgorithmen den Roboter LAURON in der Version IVb an. Im Zuge vorausgehender Forschungsarbeiten [13][14] wurden bereits Algorithmen für reaktives Laufverhalten entwickelt. In sehr unebenem Gelände birgt das reaktive Laufverhalten allerdings starke Nachteile. So kann es vorkommen, dass das Reagieren auf ein Hindernis die eigentliche Fortbewegung dominiert. Um dieses Verhalten zu verbessern wurde ein Laufplaner, zunächst in einer eigens entwickelten Simulationsumgebung, realisiert. Wie sich zeigte, kann das Laufverhalten dadurch positiv beeinflusst werden [15].

Damit der Algorithmus die Positionen der Fußauftrittspunkte während des Schreitens bestimmen kann, muss dieser das Gelände und die Position des Roboters in diesem kennen. In der Simulationsumgebung bereitet dies keine Schwierigkeiten, da das Gelände bereits als Höhenkarte vorliegt. In der Realität muss LAURON seine Welt mittels Sensoren selbst erkunden und seine Position selbst bestimmen.

Zu diesem Zweck ist der Roboter mit zwei Kameras ausgestattet die ihm ein räumliches Sehen ermöglichen. Ein entsprechender Stereobildverarbeitungsalgorithmus mit Echtzeitfähigkeiten wurde bereits in einer Diplomarbeit an der Universität Magdeburg [1] erstellt.

Um dem Ziel eines selbsterkundenden Laufroboters näher zu kommen sollen die Ergebnisse der beiden Arbeiten verknüpft werden. Dazu bedarf es allerdings noch einiger Vorarbeiten, Verbesserungen und Untersuchungen. Es musste der Laufplanungsalgorithmus, der bisher den kompletten Weg plant, auf eine inkrementelle Arbeitsweise angepasst werden. Aber auch eine geeignete Methode der Kartierung muss noch gefunden werden. Als Beispiele für Untersuchungen und Verbesserungen seien hier die Geschwindigkeitsmessun-

gen der Algorithmen und deren Verwendbarkeit in einer Echtzeitumgebung erwähnt.

1.2 Aufgabenstellung

In seiner Diplomarbeit [1] stellt Richard Bade einen Stereobildverarbeitungsalgorithmus vor, der ausgehend von einer minimalen Ausführungszeit jederzeit abbrechbar ist und dennoch ein Ergebnis liefert. Je länger dieser Algorithmus läuft um so besser ist das Ergebnis. Dieser Anytime-Algorithmus wurde für den Einsatz auf dem Laufroboter LAURON konzipiert, ist aber auch in der Lage, gespeicherte Bilder zu verarbeiten, was ihn dazu befähigt auch auf anderen Systemen wie zum Beispiel der Simulationsumgebung zu laufen.

Weiter hat André Herms [15] in seiner Diplomarbeit verschiedene heuristische Optimierungsverfahren auf ihre Tauglichkeit für den Einsatz in einem Laufplaner untersucht und dazu einen Laufplaner in der Simulationsumgebung erstellt. Dieser Laufplanungsalgorithmus generiert für vorgegebene Start- und Zielpunkte die Fußauftrittspunkte für die gesamte Strecke. Dazu nutzt er die Terraininformationen, die in der Simulationsumgebung bereits digital als Höhenkarte vorliegen.

Um dem langfristigen Ziel, einen funktionierenden Laufplanungsalgorithmus auf dem Roboter zu implementieren, der die nötigen Geländeinformationen selbst gewinnt, näher zu kommen, sollte der vorhandene Laufplanungsalgorithmus angepasst werden, so dass er inkrementell Fußauftrittspunkte generiert. Dies ist nötig, da eine weiter entfernte Geländeinformation unsicherer ist als eine naheliegende und somit eventuell Fehler enthält, die eine Neuplanung im Verlauf des Schreitens zum Ziel erzwingen. Außerdem soll der Laufplanungsalgorithmus während des Schreitens arbeiten, um einen flüssigen Lauf zu ermöglichen.

1.3 Ergebnisse

Als Ergebnis dieser Arbeit steht ein Laufplanungsalgorithmus zur Verfügung, der seine Fußauftrittspunkte sukzessiv während des Schreitens ermittelt. Dazu wurde der Algorithmus so angepasst, dass er von einer beliebigen Stellung des Roboters ausgehend eine Planung ansetzen kann. Dies befähigt den Algorithmus auch dazu, bei einem auftretendem, nicht erkanntem Hindernis, von der neuen Situation aus weitere Bewegungen zu planen.

Weiter wurde ein Fehler entdeckt, der aufeinanderfolgende Roboterkonfigurationen erzeugt, deren Übergang ungültige Positionen des Roboters zur Folge hat. In dieser Arbeit werden auch mögliche Fehlerbehebungsansätze besprochen und eine dieser Varianten implementiert.

Es konnte gezeigt werden, dass der Algorithmus verschiedene Terrains wie eine Stufe, Laufen auf einem schmalen Grat oder Schlaglöcher bewältigt. Die Lösungsgeschwindigkeit hängt dabei vom Schwierigkeitsgrad des Untergrundes ab. Die Laufrichtung ist nur durch die Bewertungsfunktion gesteuert, was die Lösungsgeschwindigkeit ebenfalls beeinflusst. Die Laufrichtung fällt in den Bereich der Routenplanung und wird in dieser Arbeit daher nicht berücksichtigt. Lediglich ein rudimentäres Routen durch Bewerten wurde implementiert, um die Funktionsweise ausreichend testen zu können.

Der Algorithmus wurde auf Geschwindigkeit untersucht mit dem Ergebnis, dass im Mittel 3.78 Lösungen pro Sekunde gefunden werden. Je Teilstück werden im Durchschnitt 6.25 Lösungen gefunden, bis der Roboter in ein neues Teilstück eintreten muss. Im Mittel verbessert der Algorithmus alle 14.65 Sekunden seine Lösung. Bezogen auf ein Teilstück erreicht der Algorithmus im Mittel 1.16 Verbesserungen.

Die Abbildung 1.3 veranschaulicht wie die Messergebnisse der Generierungen auf die unterschiedlichen Terrains verteilt ist.

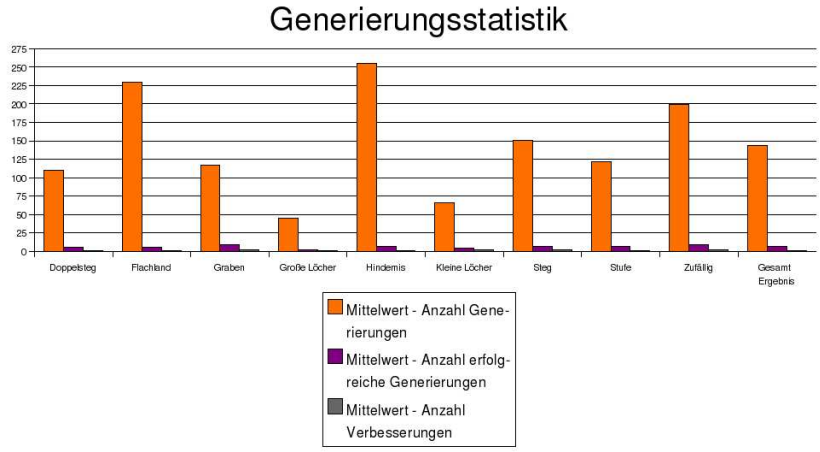

Abbildung 1.3: Verteilung der Generierungen auf die unterschiedlichen Terrains

1.4 Aufbau der Arbeit

In Kapitel 2 werden bisherige Veröffentlichungen im Umfeld der planenden Laufsteuerung, binokularem Sehen und Umweltmodellierung dargestellt.

Kapitel 3.1 beschreibt das biologische Vorbild des Roboters, geht auf die technische Umsetzung der Maschine ein und befasst sich mit dem Software Framework MCA2, das zur Programmierung des Roboters verwendet wurde. Anschließend wird in Kapitel 3.2 das Prinzip des stereoskopischen Sehens erläutert und im Speziellen auf den Algorithmus *Block Matching* eingegangen. In Kapitel 3.3 werden die nötigen Grundlagen der Laufplanung geschaffen. Hier wird die Aufgabe der Laufplanung erklärt und diese gegenüber der Pfadplanung abgegrenzt. Kapitel 3.4 erleutert die Bedeutung von Anytime-Algorithmen und zeigt in wie weit der verwendete Laufplanungsalgorithmus den Anytime-Anforderungen genügt.

In Kapitel 4 folgt die Analyse des Arbeitsstandes. Hier wird zunächst in 4.1 das Klassendesign mit den daraus resultierenden Konsequenzen besprochen und anschließend in Kapitel 4.2 der dieser Arbeit zugrundeliegende Laufplanungsalgorithmus erklärt. Danach wird in Kapitel 4.3 der verwendete Stereobildverarbeitungsalgorithmus erläutert.

In Kapitel 5 werden die möglichen Varianten zur Umsetzung der Forderungen erörtert. Kapitel 5.1 beschreibt die Möglichkeit, die im Algorithmus verwendeten Movements aneinander zu hängen und geht dabei auf die Schwierigkeiten und Vor- und Nachteile ein. In Kapitel 5.2 wird erklärt wie FootConfEvents verkettet werden können. Weiter werden auch hier die entstehenden Schwierigkeiten und Vor- und Nachteile aufgezeigt. Kapitel 5.3 bespricht einen im zugrundeliegenden Algorithmus liegenden Fehler und zeigt Möglichkeiten auf, wie dieser Fehler vermieden werden kann. Danach wird in Kapitel 5.4 die tatsächliche Umsetzung und die verwendeten Lösungsmöglichkeiten erläutert.

In Kapitel 6 werden die zum Testen des Algorithmus vorgenommenen Messungen vorgestellt. Kapitel 6.1 zeigt die Ereignisse der Geschwindigkeitsmessungen, die am Algorithmus vorgenommen wurden und in Kapitel 6.2 werden die bewältigten und nicht bewältigten Gälendeformen gezeigt.

Kapitel 7 fasst die Ergebnisse dieser Arbeit zusammen und geht auf mögliche Weiterentwicklungen ein.

Kapitel 2

Verwandte Arbeiten

Das Laufplanen wurde in der Wissenschaft schon häufiger untersucht und verschiedene Algorithmen wurden dafür entwickelt. Dabei wurden unterschiedliche Ansätze verfolgt, die vom simplen Anpassen der Hubhöhe eines Beines bis hin zum vollständigen Generieren der Bewegung reicht. Dabei unterscheiden sich die Verfahren in reaktive Verfahren, die auf eine mit den Sensoren wahrgenommene Situation mit immer der gleichen Aktion reagieren, deliberative Verfahren, die aus einer Menge möglicher Aktionen eine geeignete auswählen und reflexive Verfahren, bei denen der Roboter sich selbst im Zusammenhang zu seiner Umgebung erkennt. Nur mit den beiden zuletzt genannten Verfahren ist ein Planen möglich.

Zahlreiche Untersuchungen zu geeigneten Laufmustern wurden in der Vergangenheit durchgeführt. In [31] wurden verschiedene Laufmuster analysiert und hinsichtlich Geschwindigkeit und Stabilität bewertet. Durch Betrachten von Laufmustern als Optimierungsproblem und Lösen dieses mittels genetischer Algorithmen [22][24][25] und Backtracking [23] wurden diese Ergebnisse bestätigt. Laufmuster finden in verschiedenen Arbeiten Verwendung. Manchmal geschieht dies direkt, wie in [16][11][26], oder aber auch auf neuronale Netze übertragen, wie in [14][7][21]. Dabei wurden die Berechnungen des

Laufmusters und die der Gelenkwinkel zusammengefasst, was die Rechengeschwindigkeit verbesserte, jedoch zu Lasten der Genauigkeit ging.

In [9] wird ein Algorithmus vorgestellt, der die Fußauftrittspunkte für jedes einzelne Bein erzeugt. Darüberhinaus wird eine Methode gezeigt, die die Durchquerbarkeit eines Terrains feststellt, basierend auf einer diskreten Höhenkarte. Die Planung wird dazu in zwei Ebenen vorgenommen. In oberster Ebene wird der Algorithmus PRM[1] verwendet um die Körperbewegung zu planen. Dazu wird die Auflösung der Höhenkarte so skaliert, dass die kleinste Einheit gerade so groß wie die kleinste Fußauftrittsfläche des Roboters ist, um so die Rechenzeit zu verkürzen und den Speicherbedarf zu reduzieren. In der darunter liegenden Ebene werden Fußauftrittspunkte gesucht, die den Roboter entlang des zuvor gefundenen Pfades führen. Diese Ebene verwendet die Tiefensuche entlang eines Entscheidungsbaumes. Der Entscheidungsbaum wird aufgebaut über die Frage ob der Körper zur nächsten Position geschoben oder ein Fuß bewegt werden soll. Entwickelt wurde der Algorithmus für die PolyBot Plattform [37], wobei der Algorithmus allgemein auch auf andere Laufroboter übertragbar ist.

In [18] wird ein Verfahren vorgeschlagen, das das periodische Laufmuster des Roboters für das jeweilige Gelände in Echtzeit anpasst. Dazu werden einige Parameter der Gangart des betreffenden Beines angepasst. Die Besoderheit des Verfahrens liegt darin, dass der Algorithmus keine Kenntnisse über Standfestigkeit der Füße braucht. Eine Beispielimplementierung für einen Vierbeiner unter Verwendung des Wellengangs wird gezeigt und für Experimente herangezogen. Der präsentierte Algorithmus stellt ein reaktives Verfahren dar, wobei nach einem nötigen Anpassen des Laufmusters für ein Bein diese Information genutzt wird, um nachfolgende Beine möglichst gleich korrekt zu platzieren.

[1] Probabilistic road-map planner

Aber auch im Bereich der Umweltmodellierung wurde bereits geforscht. In [12] beschreibt Frommberger ein Weltmodell, das es einem Roboter erlaubt, sensorisches Wissen über seine Umgebung in geeigneter Form zu speichern, um dieses für eine Laufplanung auszuwerten. Dazu kartiert er die Umwelt in unterschiedlichen Abstraktionsstufen. Aus einer scrollenden 3D-Urkarte werden globale 2D-Karten und lokale Abbildungen der Fußumgebung erzeugt, um so Detailinformationen vor allem für das unmittelbare Umfeld der Maschine zu speichern. Mit einem auf Fuzzy-Logik basierenden Verfahren wird erreicht, dass Sensordaten mit vorhandenen Informationen abgeglichen werden können. Das von Frommberger vorgeschlagene Weltmodell ist nicht auf den Roboter LAURON beschränkt, jedoch wurden die Experimente und Messungen auf diesem durchgeführt.

Kapitel 3

Grundlagen

In diesem Kapitel werden die zum Verständnis der Arbeit nötigen Grundlagen besprochen. Darüber hinaus wird auf die Prinzipien, die zur Entscheidungsfindung für die verwendeten Algorithmen und zum Verständnis der Konzeption der Arbeit erforderlich sind, eingegangen.

Da die Simulationsumgebung nicht nur dem Studium der Machbarkeit von Algorithmen dient, sondern vielmehr in Richtung "Real Simulation" getrieben werden soll – also möglichst wirklichkeitsnah die tatsächlichen Bewegungsabläufe und Verhaltensweisen des LAURON simulieren soll – wird in Abschnitt 3.1 auf den Aufbau und die Software-Architektur des Laufroboters eingegangen.

3.1 Der Laufroboter LAURON

Wie eingangs bereits erwähnt, ist die Anatomie von LAURON der einer Stabheuschrecke (Carausius Morosus) nachempfunden, weil die Bewegungsabläufe dieser Tierart weitestgehend erforscht sind.

3.1.1 Das natürliche Vorbild - die Stabheuschrecke

Wie bei allen Insekten ist auch der Körper der indischen Stabheuschrecke dreigeteilt in Kopf (Caput), Brust (Thorax) und Hinterleib (Abdomen). Am Kopf fallen die beiden langen Fühler auf, die genauso lang oder länger als die vorderen Beinpaare sind. Auch der Thorax ist in drei Segmente geteilt von denen jedes ein Beinpaar trägt. Der Hinterleib ist in elf Segmente unterteilt. Jedes Bein hat drei Hauptsegmente (Coxa [Hüfte], Femur [Oberschenkel], Tibia [Unterschenkel]) und drei Hauptgelenke (Subcoxal [α], Coxa-Trochanter [β], Femur-Tibia [γ]), die ungefähr in einer Ebene liegen (vergleiche Abbildung 3.1.

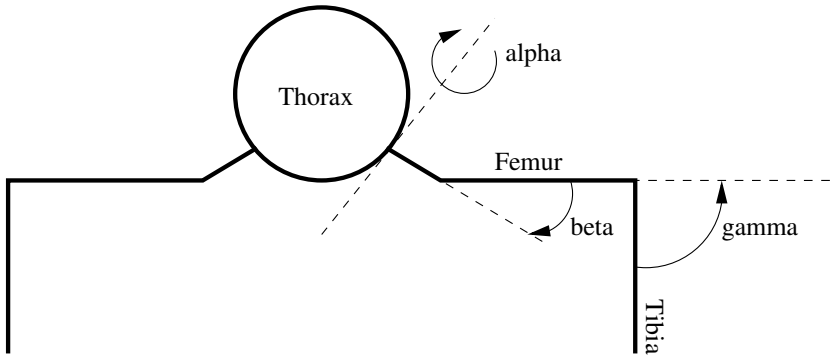

Abbildung 3.1: Beinsegmente und Hauptbewegungsachsen im vertikalen Schnitt.

Die beiden Gelenke β und γ sind Scharniergelenke und haben somit einen Freiheitsgrad. Sie sind für Beugen und Strecken des Beines verantwortlich. Das α-Gelenk verbindet Bein und Körper und besitzt zwei Freiheitsgrade. Allerdings entspricht die Hauptbewegung der eines Scharniergelenkes mit einer aus der Senkrechten geneigten Achse. Mit diesen drei Freiheitsgraden kann das Bein beliebig im dreidimensionalen Raum positioniert werden. Der zusätzliche Freiheitsgrad des α-Gelenks wird nur selten zum Laufen eingesetzt

und kann vernachlässigt werden. Hauptsächlich verwendet das Tier diesen Freiheitsgrad um zum Schutz die Beine an den Körper zu ziehen. [36]

Laufverhalten

Die Beine der Stabheuschrecke sind dezentral gesteuert. Jedes Bein besitzt einen sogenannten Schrittmustergenerator, der das Laufverhalten des einzelnen Beines steuert. Das Gesamtmuster entsteht aus dem Austausch der verschiedenen Zentren. Dazu findet ein Informationsaustausch zwischen banachbarten Beinen statt. Das jeweils vordere Bein gibt seine Information an das Folgende weiter; dieses "lernt" sozusagen von seinem Vorgänger. Wenn das Tier beispielsweise einen Graben überwinden will, muss nur das vorderste Beinpaar an den Rand herantreten und die Folgenden wissen bereits, dass sie einen großen Schritt machen müssen.

Die vom Schrittmustergenerator erzeugten zyklischen Bewegungen lassen sich in zwei Phasen unterteilen, die Stemmphase und die Schwingphase. In der Stemmphase stützt das Bein den Körper über dem Boden und schiebt ihn in die gewünschte Richtung. In der Schwingphase wird das Bein angehoben und in die vorgegebene Richtung bewegt, um an einem geeigneten Ort wieder in die Stemmphase eintreten zu können.

Über den Übergang zwischen den Phasen entscheidet der Schrittmustergenerator. Der Übergang von Stemm- zu Schwingphase ist hierbei besonders kritisch, da die stützende Funktion des Beines beendet wird. Ein Phasenwechsel im falschen Moment kann dazu führen, dass das Tier kippt. Um einen Wechsel im richtigen Moment vorzunehmen, muss das Insekt die Position des Beines, die Kraft mit der das Bein gerade belastet wird und die Phase im Schrittzyklus der anderen Beine in Betracht ziehen. [28] [36]

3.1.2 Der Aufbau des Roboters

LAURON wurde gebaut um Laufalgorithmen für unebenes Gelände zu erforschen. Er besteht aus einem Körper, in dem die Steuerungselektronik untergebracht ist, einem Kopf der drehbar auf den Körper montiert wurde und sechs identischen Beinen. Hinterleib und Segmentierung des Körpers wurde von seinem biologischen Vorbild nicht übernommen, da diese nur unwesentlich zum Laufverhalten beitragen. Jedes Bein hat drei Freiheitsgrade und somit einen Freiheitsgrad weniger als das Insekt (siehe Abbildung 3.2). Wie im vorangegangenen Kapitel erwähnt benutzt das Tier diesen Freiheitsgrad nicht oder nur selten zum Laufen, daher wurde dieser bei der Maschine zur Reduktion der Komplexität der Beinkoordination vernachlässigt. Damit kommt der Roboter auf 18 Beinfreiheitsgrade (sechs Beine mit je drei Freiheitsgraden).

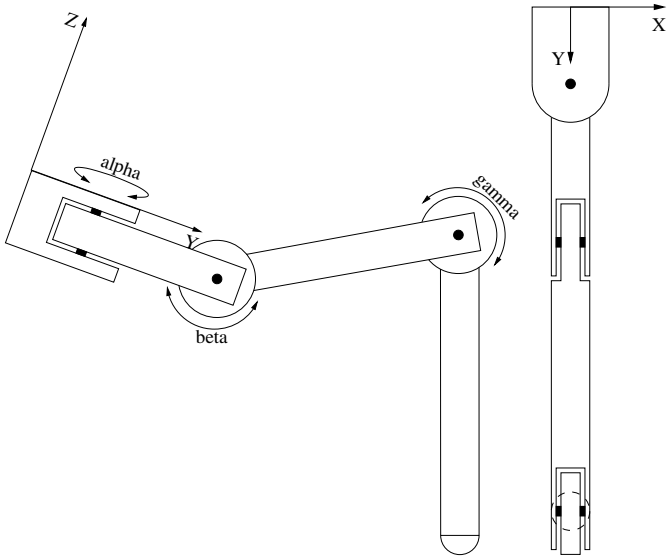

Abbildung 3.2: Bein des LAURON, Seitenansicht und Draufsicht.

Jedes Bein verfügt über Kraftsensoren im letzten Glied, mit denen der Roboter entscheiden kann, ob das Bein den Untergrund berührt und wie-

viel Kraft das entsprechende Bein bei seiner Stützfunktion aufnimmt. Die Gelenke sind mit Winkelsensoren ausgestattet. Um Sensorsignale auswerten und Stellgrößen anlegen zu können, verarbeitet ein Digitaler-Signal-Prozessor (DSP) je Bein (und einer für den Kopf) die Ein- und Ausgangssignale. Diese DSPs sind mittels CAN-Bus vernetzt und an den Hauptrechner, einen gewöhnlichen Industrie-PC (PC104), angeschlossen. Der Hauptrechner übernimmt die gesamte Laufsteuerung und wird mit einem Linux (Debian) betrieben. Durch die Verwendung eines RT-Kernels erhält das Betriebssystem Echtzeitfähigkeiten.

Der drehbare Kopf des Roboters beherbergt zwei Digitalkameras, die per IEEE 1394[1] an den PC104 angeschlossen sind. Um die Blickrichtung zu ändern ist nicht nur der Kopf schwenkbar, sondern zusätzlich ist es möglich die Kameras zu neigen.

Angaben zu Abmessungen, Gewichten und technischen Daten können der Tabelle 3.1 entnommen werden.

3.1.3 Das Framework MCA2

Der Laufplaner wurde zwar zum Erforschen möglicher Algorithmen in einer Simulationsumgebung implementiert, da jedoch ein späterer Einsatz auf dem echten Roboter angedacht ist, erfolgt hier ein kurzer Überblick über das Framework MCA2. Sämtliche Laufalgorithmen des LAURON verwenden dieses Framework.

MCA2 steht für Modular Controler Architecture in der Version zwei. Die Architektur ist eine Entwicklung des Forschungszentrum für Informatik in Karlsruhe und wurde speziell für Entwicklungsarbeit an autonomen Robotern geschaffen. [30]

[1] auch bekannt als FireWire

Basis Parameter	
Grundform	Stabheuschreckenartig
Anzahl der Beine	6
Aktive Bein–Freiheitsgrade (gesamt)	$6 \cdot 3 = 18$
Max. theoretische Geschwindikeit ca.	0.5 km/h
Gewicht (Grundkonfiguration)	19 kg
Nutzlast ca.	10 kg
Körper	
Grundkörperabmessungen	
Höhe ca.	19 cm
Seitenlänge ca.	72 cm
Breite ca.	24 cm
Sensoren	
Neigungssensoren	1
Beine	
Aktive Freiheitsgrade	3
Arbeitsbereich der Gelenke	
α–Gelenk	$-60° .. +60°$
β–Gelenk	$-30° .. +60°$
γ–Gelenk	$-30° .. +60°$
Antriebe	Gleichstrommotoren (24V)
Abmessungen eines Beines	
Gesamt	64,5cm
A–Segment	7cm
B–Segment	7,5cm
C–Segment	20cm
D–Segment	30cm
Masse eines Beines ca.	2,2kg
Sensoren	
Gelenkwinkelsensoren	3 optische Inkrementalsensoren pro Beingelenk
Fußkraftsensoren	3 orthogonale Komponenten im Unterschenkel
Motorstromsensoren	3 pro Gelenkmotor
Kopf	
Aktive Freiheitsgrade	2
Sensoren	
Kameras	2 am Kopf angebrachte IEEE 1394 Stereokameras

Tabelle 3.1: Technische Daten LAURON IVb [36].

In MCA2 sind alle Methoden als einfache Module mit standardisierten Schnittstellen realisiert. Verbunden werden die Module mit Kanten, die die Daten transportieren, so ist die Kommunikation realisiert. Die Modulschnittstellen werden durch einfache Arrays von float-Werten repräsentiert. Kanten kopieren lediglich die Werte von der Ausgangs- zur Eingangsschnittstelle. Man kann Module zu Gruppen zusammenschließen, wobei eine Gruppe sich wie ein komplexeres Modul verhält (vergleiche Abbildung 3.3).

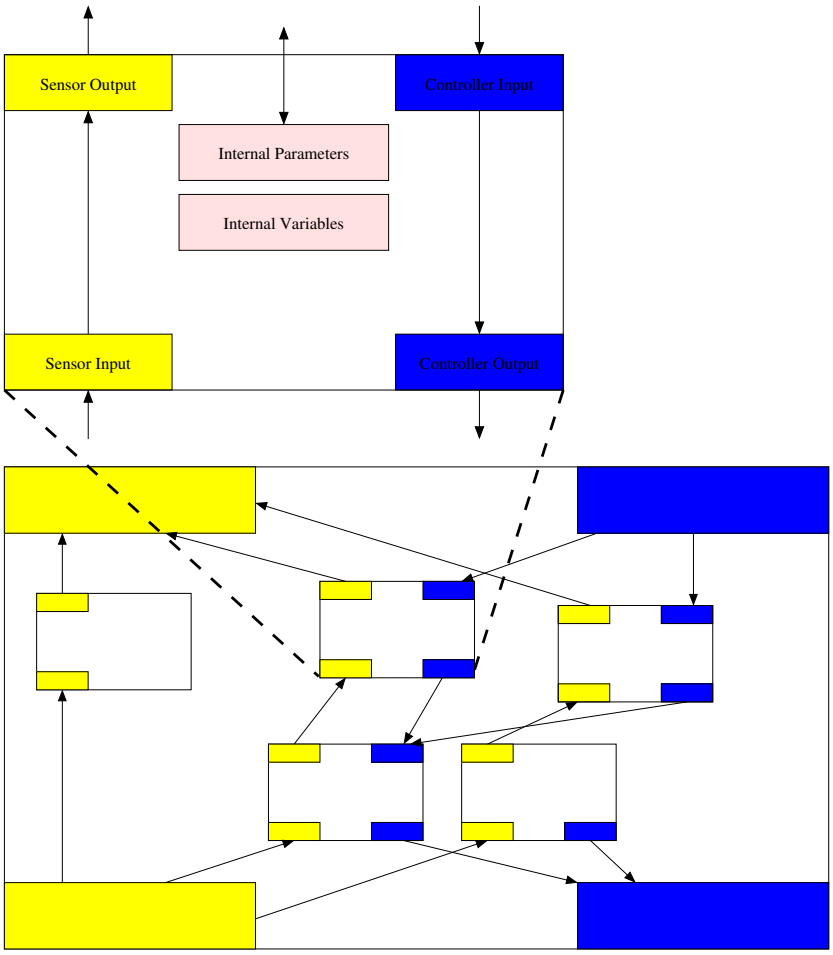

Abbildung 3.3: Schematische Darstellung eines MCA2 Moduls.

Alle Module sind gleich aufgebaut und jedes implementiert fünf Schnittstellen. Vier dieser Schnittstellen sind für die Kommunikation zuständig, die Fünfte um interne Parameter zu lesen oder zu schreiben.

Darüber hinaus implementiert jedes Modul die beiden Funktionen *Sense()* und *Control()*. Diese stellen die eigentliche Funktionalität des Modules zur Verfügung. Während *Sense()* dafür zuständig ist, die Sensordaten zu verarbeiten, ist *Control()* für die Stellgrößen zuständig. [29]

Innerhalb einer Gruppe sind die Module hierarchisch geordnet und werden nacheinander entsprechend ihrer Hierarchieebene abgearbeitet. Der Aufruf einer *Control()*-Funktion löst also alle nachfolgenden *Control()*-Funktionen entlang der absteigenden Kante aus. [36]

3.2 Informationsgewinnung

Um Informationen über das Gelände in dem sich LAURON bewegt zu gewinnen, stehen dem Roboter außer seinen taktilen Sensoren zwei Videokameras zur Verfügung. Bei den Optischen Systemen unterscheidet man zwischen aktiven und passiven Systemen. Passive Systeme stellen die Lichtintensitäts- oder Farbinformation der Umgebung in einem zweidimensionalen Bild fest. Aktive Systeme senden selbst ein strukturiertes Licht aus, das als Messsignal ausgewertet wird [17][19].

Da die Anschaffung eines Linienprojektors in Form eines Lasers in Erwägung gezogen wurde, wird hier auch ein aktives Messverfahren besprochen.

3.2.1 3D Informationen aus 2D Bildern

Bereits in zweidimensionalen Bildern sind Tiefeninformationen enthalten, dies wird beim Betrachten von Fotos erkennbar. Der Mensch empfindet auch bei diesen Bildern räumliche Tiefe. Dieser Eindruck wird durch das Gehirn

erzeugt, das bei der Rekonstruktion der gesehenen Szene die einzelnen Objekte erkennt und die Größenunterschiede mit bereits gelerntem Wissen vergleicht. Darüber hinaus wird aber auch der Verlauf von Licht und Schatten und Teilverdeckungen von Objekten ausgewertet.

Aber nicht nur der visuelle Eindruck liefert dem Gehirn Informationen zur räumlichen Tiefe. Auch die Fokussierung der Pupille und die Stellung der Augen liefern entscheidende Hinweise über den Abstand zum erfassten Objekt.

Diese Fähigkeiten hat man sich auch beim maschinellen Sehen zunutze gemacht. Auch hier gibt es Verfahren wie die *Verdeckungsanalyse* [32], mit dem entschieden werden kann, ob ein Objekt sich hinter oder vor einem anderen befindet, *Shape-From-Shading*[2] [33], das Licht und Schattenverläufe ausnutzt um Informationen zur Form des Objektes zu extrahieren oder *Tiefe-von-Fokus* [10], bei dem die Brennweiteneinstellung der Kamera verwendet wird um die Tiefe des fokussierten Objekts zu messen.

Triangulation mittels Linearprojektion

Diese Triangulationsmethode gehört zu den aktiven optischen Systemen und hat den Vorteil, dass sie unabhängig von Textur, Oberflächenbeschaffenheit, Form der Objekte und Licht- und Schattenverhältnissen funktioniert. Verfahren, die Kanten von Objekten nutzen um Tiefeninformationen zu gewinnen, oder auch stereoskopische Verfahren sind darauf angewiesen, dass gewisse Strukturen oder Objekte erkennbar sind. Stünde die Kamera beispielsweise senkrecht auf eine weiße Wand gerichtet, wäre das aufgenommene Bild lediglich eine weiße Fläche. Kantenverläufe wären nicht zu erkennen und auch ein Stereobildalgorithmus könnte ohne eine Textur keine Tiefeninformation gewinnen.

[2] Form durch Schattierung

Bei der Triangulationstechnik ist dies anders. Hier ist es möglich, durch geschickte Konfiguration von Projektor und Zeilenkamera den Abstand zum Objekt direkt an der Zeilennummer, auf die das ausgesandte Licht im Fotodetektor abgebildet wird, abzulesen. Die Objektentfernung und der Winkel, unter dem der Lichtpunkt am Objekt auf den CCD-Sensor abgebildet wird, stehen in geometrischem Verhältnis (vergleiche Abbildung 3.4). [19]

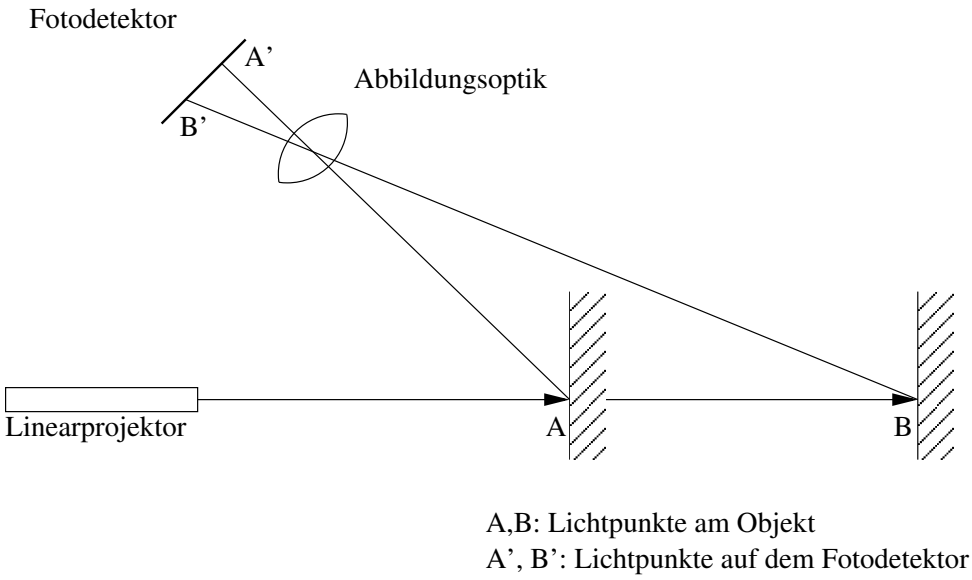

Abbildung 3.4: Laserentfernungsmessung nach dem Triangulationsprinzip.

Die Koordinaten des Punktes $\overrightarrow{P}$ (vergleiche Abbildung 3.5) lassen sich aber auch berechnen. Nehmen wir an, dass die beiden Koordinatensysteme Kamera- und Weltkoordinatensystem identisch sind. Der Ursprung sei in der Mitte der Linse und X- und Y-Achse verlaufen parallel zur X- und Y-Achse des Bildes. Die Z-Achse ist die optische Achse des Aufnahmesystems und verläuft senkrecht zum Bild durch den Linsenmittelpunkt. f sei der Abstand vom Linsenmittelpunkt zum Bild (vergleiche Abbildung 3.5).

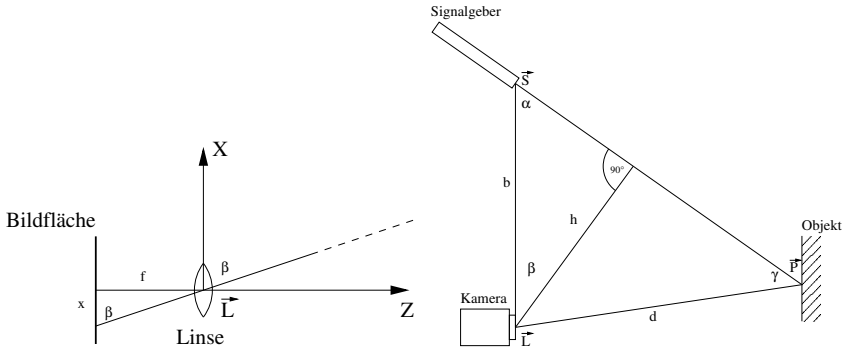

Abbildung 3.5: Lasertriangulation Kamerasystem

Unter der Einschränkung, dass der Punkt $\overrightarrow{P}$ auf $(x_P, 0, -f)$ abgebildet wurde lassen sich nun die Koordinaten von $\overrightarrow{P}$ bestimmen. Zunächst bestimmen wir das Dreieck mit den Eckpunkten $\overrightarrow{P}$, $\overrightarrow{L}$ und $\overrightarrow{S}$. Die erste Seite dieses Dreiecks ist die Linie $(0,0,0) - (0,0,b)$. Der erste Winkel ist damit $\frac{\pi}{2} - \alpha_x$. Da der Punkt auf $(x_P, 0, -f)$ abgebildet wird, muss $\alpha_y = 0$ sein und damit ist $\alpha_x = \alpha$.

β ist durch den Winkel zwischen der Linie vom Linsenmittelpunkt zum gesuchten Punkt $\overrightarrow{P}$ und der X-Achse gegeben. Dieser Winkel findet sich ein zweites Mal im Dreieck $(x_P, 0, -f)$, $(0,0,0)$ und $(0,0,-f)$. In diesem rechtwinkligen Dreieck sind alle Eckpunkte bekannt und damit ist $\tan\beta = \frac{f}{x_P}$.

Nun ermitteln wir den Abstand d. Für h gelten offensichtlich folgende Beziehungen:

$$h = d \cdot \sin\gamma = b \cdot \sin\alpha \tag{3.1}$$

Durch Gleichsetzung erhält man:

$$d = b \cdot \frac{\sin\alpha}{\sin\gamma} \tag{3.2}$$

Da aber $\alpha + \beta + \gamma = \pi$ und $\sin \pi - \alpha - \beta = -\sin \alpha + \beta$ sind ergibt sich:

$$d = -b \cdot \frac{\sin \alpha}{\sin \alpha + \beta} \tag{3.3}$$

Nun, da d bekannt ist, kann $\overrightarrow{P}$ berechnet werden:

$$\overrightarrow{P} = (d \cdot \cos \beta, 0, d \cdot \sin \beta) \tag{3.4}$$

Im Falle eines Linienprojektors liegt $\overrightarrow{P} = (x_P, y_P, -f)$ nicht immer in der XZ-Ebene. Dann wird zunächst die gleiche Berechnung für einen in die XZ-Ebene projizierten Punkt $\overrightarrow{P}'$ durchgeführt. β wird wieder als $\beta = \tan^{-1} \frac{f}{x_P}$ berechnet. Der erste Winkel muss jetzt der Winkel α_x sein. Da $\overrightarrow{P}'$ der in XZ-Ebene projizierte Punkt $\overrightarrow{P}$ ist, sind X- und Z-Koordinate der beiden Punkte gleich. Die Y-Koordinate ergibt sich durch Anwendung der Strahlensätze. Danach ist

$$\frac{y_P}{f} = \frac{Y}{d} \Leftrightarrow Y = \frac{d}{f} \cdot y_P$$

Auf diese Weise kann nun jeder Punkt der projizierten Linie berechnet werden. [33]

Die Triangulationstechnik hat aber auch Grenzen. So kann es in einem Terrain vorkommen, dass ein Vorsprung einer Anhöhe den Laserstrahl komplett abschattet (vergleiche Abbildung 3.6). In einem solchen Fall lässt sich über das zu vermessende Gebiet keine Aussage machen.

3.2.2 Räumliches Sehen

Beim stereoskopischen oder auch binokularen Sehen werden von einem Augenpaar, das auf die gleiche Szene gerichtet ist, 2 Bilder dieser Szene aus unterschiedlichen Perspektiven erstellt. Durch Auswerten der Disparität und der perspektivischen Verzerrung gewinnt das Gehirn einen Tiefeneindruck der Szene.

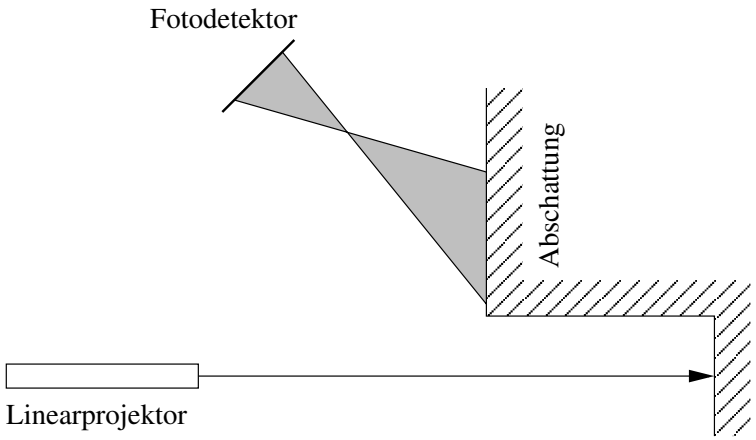

Abbildung 3.6: Abschattungsproblem bei der Lasertriangulation

Über Disparität und Perspektive hinaus wertet das Gehirn auch Bewegungen von Objekten in aufeinanderfolgenden Bildern aus, wodurch weitere Tiefeninformationen gewonnen werden können.

Beim maschinellen räumlichen Sehen wird meist die Disparität ausgewertet. Ein solches Verfahren ist das Blockmatching, das auch im Stereobildverarbeitungsalgorithmus von Richard Bade Anwendung findet.

Anytime-Block-Matching

Beim Block-Matching unterteilt man eines der beiden Bilder in $m \times n$ Blöcke und sucht im anderen Bild nach einem möglichst ähnlichen Block gleicher Größe. Als Ähnlichkeitsmaß wird meist der quadratische Fehler (MSE) zwischen den Pixelwerten der Blöcke verwendet.

Das linke Bild I_L und das rechte Bild I_R sind mit Grauwerten G definiert als

$$I_L(i,j) = G_L(i,j) \tag{3.5}$$

$$I_R(i,j) = G_R(i,j) \tag{3.6}$$

Der MSE ist dann definiert mit $n = m = 2k + 1$ als

$$MSE(x, y, \Delta) = \frac{1}{n \cdot m} \sum_{i=-k}^{k} \sum_{j=-k}^{k} ((G_L(x + i, y + j) - G_R(x + i + \Delta, y + j))^2) \tag{3.7}$$

Δ ist hierbei der Pixelabstand $(x_L - x_R)$ der beiden Blöcke. Bei einer kanonischen Kamerakonfiguration, wie es bei LAURON der Fall ist, entsprechen die Epipolarlinien den Bildzeilen. Dadurch kann die Suche auf den horizontalen Suchraum eingeschränkt werden.

Die Disparität D zweier Blöcke ist definiert als die horizontale Distanz, bei der der MSE minimal ist. Weiter kann der Suchraum durch die Vorgabe einer maximalen Disparität d_{max} eingeschränkt werden. Diese Einschränkung entspricht also einem Mindestabstand vom Objekt zur Kamera. [2]

$$D = \min_{|\Delta| <= d_{max}} MSE(x, y, \Delta) \tag{3.8}$$

Dieses Verfahren muss, um ein Ergebnis zu liefern, einmal über das komplette Bild laufen. Das heißt, wird das Verfahren vorzeitig unterbrochen, steht lediglich ein Teilergebnis zur Verfügung und kein Gesamtergebnis von minderer Qualität. Damit genügt dieses Verfahren nicht den Anytime-Anforderungen.

Um aus diesem Verfahren einen Anytime-Algorithmus zu machen, verwendet Bade das Konzept der Gaußpyramiden [5]. Das heißt, dass das Bildpaar zunächst auf eine geringere Auflösung heruntergerechnet wird. Das so skalierte Bildpaar kann der Algorithmus binnen kürzester Zeit lösen, so dass nach einer kurzen Mindestlaufzeit ein Ergebnis – wenn auch ein schlechtes – vorliegt. Nun werden nacheinander die darunterliegenden Ebenen (vergleiche Abbildung 3.7) durch das Verfahren bearbeitet, bis zuletzt das Originalbild ausgewertet wird. Auf diese Weise verbessert sich das Ergebnis stetig mit der Laufzeit und das Verfahren ist durch ein geeignetes Scheduling abbrechbar.

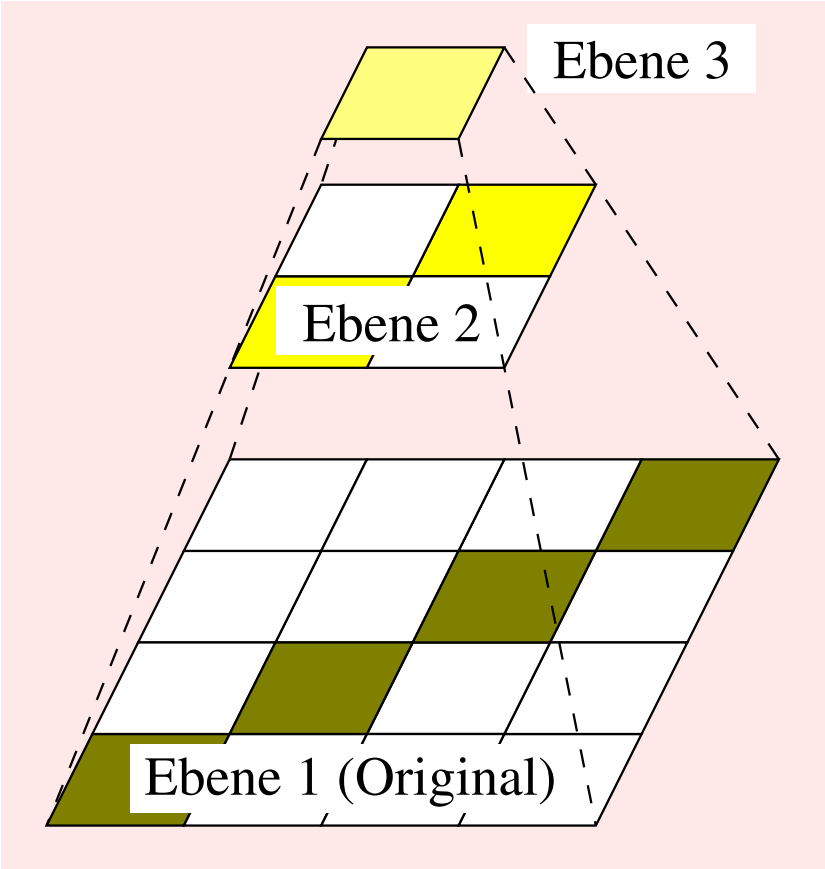

Abbildung 3.7: Schema der Gaußpyramide

Um das Verfahren möglichst effektiv zu halten, liegt es nahe, die Ergebnisse aus höheren Ebenen zu verwenden. Dies ist möglich, indem die Disparitätmatrix mit diesen Ergebnissen vorinitialisiert wird, dabei muss die Matrix selbstverständlich auf die nötige Auflösung skaliert werden.

3.3 Laufplanung

Um eine schreitende Fortbewegung eines Roboters zu implementieren hat man sich an der Natur angelehnt und die Gangart verschiedenster Tiere analysiert. Darunter auch die Gangart der indischen Stabheuschrecke (Casausius morosus), deren Fortbewegungsart sehr gut untersucht wurde, wobei sich der Dreibeingang als einer der stabilsten herausgestellt hat.

Um diese Gangart mit einem Roboter zu reproduzieren, werden statische Verfahren eingesetzt. Im LAURON wurde hierzu ein Verfahren gewählt in dem die Punkte für die Übergänge von Schwing- in Stemmphase und von Stemm- in Schwingphase in Abängigkeit der benachbarten Beine definiert werden. Dazu wurden sechs einfache Regeln definiert, die in Form eines zellulären Automaten implementiert wurden.

Statische Verfahren wurden für ein ganz bestimmtes Gelände entwickelt und können auch nur in diesem verwendet werden. Beispielsweise kann ein Roboter der für ebenes Terrain programmiert wurde keine Treppen steigen und umgekehrt. Das hat zur Folge, dass ein tiefes Schlagloch im Boden unter Umständen ein unüberwindbares Hindernis darstellt.

Abhilfe schaffen hier die reaktiven Verfahren. Diese versuchen mittels Kraftsensoren festzustellen ob ein Bein zu früh oder zu spät in die Stemmphase eintritt oder ein Bein in seiner Bewegung blockiert wird. Tritt ein solcher Fall ein, wird durch "Tasten" eine geeignete Auftrittstelle in der direkten Umgebung gesucht, um danach wieder in die statische Bewegung überzugehen.

In stark zerklüftetem Gelände hat aber auch dieses Verfahren seine Grenzen. Hat der Boden zuviele Unebenheiten, überlagert das reaktive Verhalten die eigentliche Fortbewegung. Es hat sich gezeigt, dass mit Kenntnis des Geländes, eine Laufplanung das Schreiten verbessern kann.

3.3.1 Aufgabe der Laufplanung

Die Laufplanung soll im Vorfeld, also vor dem Eintreten der Störung, geeignete Fußauftrittpunkte finden. Anders als bei reaktiven Verfahren muss auf diese Weise die Fortbewegung nicht unterbrochen werden.

Ein "Nicht-Unterbrechen" bedeutet aber auch, dass die Laufplanung während des Schreitens berechnet werden muss. Dies wiederum stellt eine Echtzeitanforderung dar. Genauer handelt es sich um weiche Echtzeitanforderungen, da eine zu lange Laufzeit des Algorithmus lediglich ein Unterbrechen der Fortbewegung zur Folge hat.

Die Planung selbst kann auf unterschiedlichen Ebenen erfolgen. Martin Guddat beschreibt in seiner Dissertation [14] eine Möglichkeit den Elevator-Reflex[3] zu vermeiden, indem Sensorinformationen dazu genutzt werden, vorausschauend das Bein auf die erforderliche Höhe anzuheben.

Craig Eldershaw hingegen beschreibt in seiner D.Phil.[4] Thesis [8] wie mit heuristischen Verfahren die gesamte Roboterbewegung geplant werden kann. Er verwendet hierzu ein mehrstufiges Verfahren das auf genetischen Algorithmen und Backtracking beruht.

3.3.2 Abgrenzung

Laufplanungsalgorithmen passen im Vorfeld das statische Laufmuster an, suchen nach geeigneten Auftrittspunkten oder planen gleich den gesamten Bewegungsablauf. Die Laufplanung beschäftigt sich nicht mit der Suche nach möglichen Wegen um ein Hindernis herum. Das Suchen geeigneter Wege fällt in den Bereich der Routen- oder auch Pfadplanung. Es ist wichtig dies zu unterscheiden, da bei der Routenplanung von den Sensordaten eine deutlich

[3] Reflex der das Bein anheben lässt, um so Hindernisse überwinden zu können.

[4] An der Universität Oxford wird der bekannte PhD so abgekürzt. Enspricht dem deutschen Titel "Doktor der Wissenschaften"

weitere “Sichtweite” gefordert wird. Im Gegensatz dazu fordert die Laufplanung eine höhere Detailtreue in der direkten Umgebung.

3.4 Anytime-Algorithmen

Der Begriff Anytime-Algorithmen spielt eine Rolle im Zusammenhang mit zeitbeschränkten Verfahren, die bereits vor ihrer vorgesehenen Terminierung ein approximatives Resultat liefern. Dean und Boddy führten den Begriff als Spezialfall unterbrechbarer Algorithmen ein [34][3]:

1. Anytime-Algorithmen können mit vernachlässigbarem Verwaltungsaufwand jederzeit unterbrochen und wiederaufgenommen werden.
2. Dabei geben sie auf jeden Fall ein Ergebnis zurück.
3. Die Qualität des Ergebnisses steigt monoton mit der investierten Rechenzeit.

Die letzten beiden Punkte machen hierbei den Hauptunterschied gegenüber normalen Algorithmen aus. Aufgrund dieser Eigenschaften eignen sich solche Algorithmen besonders für Systeme mit variabler Ressourcenbeschränkung, wie zum Beispiel Roboter, bei denen je nach Geschwindigkeit mehr oder weniger Rechenzeit je Schreitzyklus zur Verfügung steht.

Bei dem verwendeten Laufplanungsalgorithmus kann es vorkommen, dass nach Durchschreiten eines Teilstückes noch kein gültiger Bewegungsplan vorliegt. In diesem Fall muss die Bewegung unterbrochen werden, um so dem Algorithmus weitere Rechenzeit zu verschaffen. Dies widerspricht nicht der Definition eines Anytime-Algorithmus, da der Laufplaner jederzeit ein Ergebnis liefert, dieses jedoch in einem solchen Fall keine gültige Bewegung darstellt, da die Bewertungsfunktion den Wert “unbekannt” liefert.

Kapitel 4

Analyse des Arbeitsstandes

4.1 Die Simulationsumgebung

Die Simulationsumgebung ist in der objektorientierten Programmiersprache C++ geschrieben. Zur Darstellung wird das Toolkit OpenInventor verwendet, das auf OpenGL aufsetzt und ebenfalls objektorientiert in C++ implementiert ist. OpenInventor bietet bereits eine komplette grafische Benutzeroberfläche, die grundlegende Manipulationen am Darstellungsobjekt zulässt (z. B. Vergrößern oder Verkleinern, Kameraflug um die Szene oder Bewegen von Objekten mittels Dragger).

4.1.1 Aufbau

In der obligatorischen Funktion *main* werden die globalen Objekte *World*, *Movement* und *Terrain* angelegt, wobei die abhängigen Objekte *Eyes* und *Robot* durch die Klasse *World* bei deren Instanziierung angelegt werden. Die Kompositionsklasse *Robot* erzeugt seinerseits wiederum die Abhängigen *Head* und *Leg* Objekte. Die Klasse *Eyes* stellt dabei anders als der Name vermuten lässt nicht das Augenpaar des Roboters dar, sondern die in OpenInventor

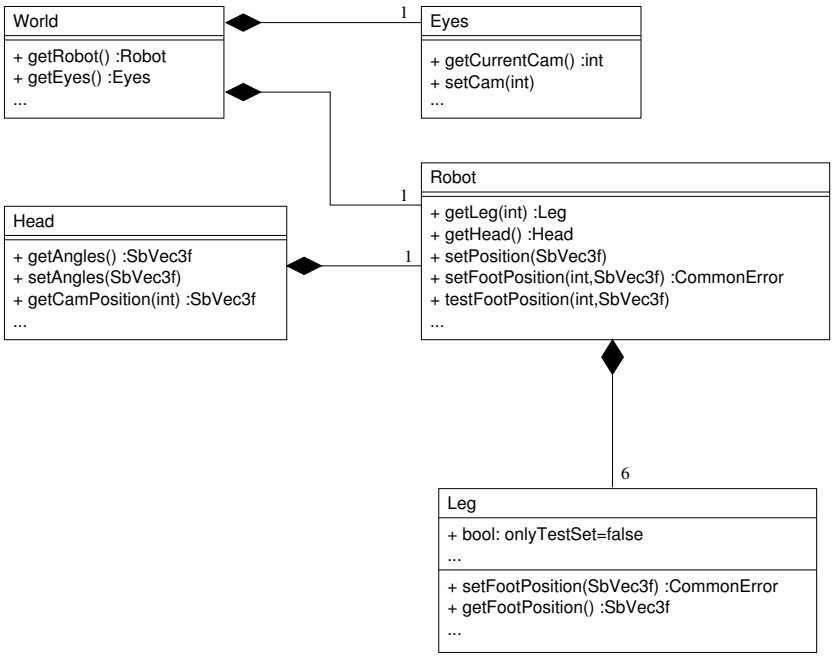

Abbildung 4.1: UML Klassendiagramm der Simulationsumgebung (auszugsweise Darstellung).

definierten Kameras, die den Blick auf die Szene ermöglichen (vergleiche Abbildung 4.1).

Alle Klassen, die den Roboter in OpenInventor darstellen, haben eine Member-Variable vom Typ *SoSeperator*. Diese ist das Darstellungsobjekt in OpenInventor und muss in den Objektbaum mit einem eindeutigen Namen eingehängt sein. Durch den eindeutigen Namen und aufgrund der Tatsache, dass dieser an verschiedenen Stellen im Quelltext hart kodiert ist, sind diese Klassen de facto nur einmalig verwendbar. Dies ist allerdings nicht durch das Design-Pattern Singleton abgesichert. Besser als die Objekte einmalig zu machen, wäre es allerdings diese Beschränkung aufzugeben und ein Mehrfachinstanziieren zu ermöglichen. So wären später Erweiterungen denkbar, in denen mehrere Roboter gleichzeitig laufen.

Die Simulationssoftware selbst, ist beziehungsweise war nicht auf den Roboter LAURON beschränkt. So wurden Form, Abmessungen und sogar die Beinanzahl in eine Konfigurationsdatei ausgelagert oder parametrisiert via Compiler-Makro. Bei einigen Erweiterungen, darunter auch der Laufplanungsalgorithmus, wurde diese Funktionalität allerdings fallen gelassen. So sind nicht nur die Anzahl der Beine hart kodiert, sondern auch implizit Annahmen über Proportionen und Bewegungsfreiheiten des Roboters im Algorithmus getroffen.

4.1.2 Probleme der Klasse Movement

In der Klasse *Movement* werden globale Objekte und Methoden statischer Klassen verwendet, die so nur in der Simulationsumgebung existieren können. Beispielsweise wird die Körperposition in Richtung der Z-Achse der Klasse *Control* entnommen, die eigentlich zur Ablaufsteuerung der Simulationsumgebung zuständig ist. *Control* verwendet hierzu auch die Höhenangaben, die *Terrain* liefert. *Terrain* stellt hierbei aber nicht das dem Roboter zur Verfügung stehende Kartenmaterial dar, sondern die Höhenkarte die zur Darstellung der Landschaft in OpenInventor verwendet wird. Dies erschwert ein späteres Portieren des Algorithmus auf den echten Roboter.

Der Algorithmus wurde in der Klasse *Movement* implementiert, wodurch dieser nur schwer austauschbar ist. Sinnvoller wäre es, die Klasse *Movement* lediglich zur Repräsentation einer Bewegung zu verwenden und den Laufplanungsalgorithmus in eine neue Klasse auszulagern.

4.2 Der Laufplanungsalgorithmus

André Herms untersuchte in seiner Diplomarbeit [15] verschiedene Laufplanungsalgorithmen, die auf heuristischen Suchverfahren basieren, auf Effizienz

und Verwendbarkeit. Er fasst dazu das Laufplanen als Optimierungsproblem auf. Als geeignetes Verfahren schlägt er das *random sampling* vor. Der von ihm erstellte Algorithmus wurde für die Simulationsumgebung geschrieben, da der Algorithmus Geländeinformationen benötigt, die der echte Roboter noch nicht liefert. In der vorhandenen Form generiert der Algorithmus alle notwendigen Fußaufsetzpunkte und Körperbewegungen, um von einem Startpunkt zu einem Zielpunkt zu gelangen und speichert diese in einer *XML-Datei*, dem sogenannten *Movement*, ab.

Der Algorithmus geht von einer Roboterkonfiguration aus, in der alle Beine festen Bodenkontakt haben. Ebenso haben am Zielort alle Beine des Roboters Bodenkontakt, was durch die Definition des *Movement*s bedingt ist.

Da beim Generieren auch Bewegungen des Körpers (*centerMove*) erzeugt werden, erzeugt der Algorithmus für große Planstrecken sogar Bewegungen, die eigentlich in den Aufgabenbereich der Routenplanung fallen.

Die Erzeugung der Bewegung geschieht (grob gegliedert) in zwei Schritten. Der Algorithmus erzeugt zunächst eine *FootConfEvent* Liste (also eine Liste von gültigen Roboterkonfigurationen wobei beim Konfigurationsübergang keine ungültigen Zustände entstehen dürfen), die ohne Berücksichtigung von Zeiten und Geschwindigkeiten definiert ist. Diese Liste wird nach Erstellen der geplanten Bewegung in ein *Movement* überführt, wobei die nötigen Zeiten (und damit auch die Geschwindigkeiten) berechnet werden.

Der Algorithmus generiert lediglich geeignete Fußaufsetzpunkte und beachtet dabei keine Hindernisse, die eventuell dem Körper oder einem Bein im Wege stehen. Dies kann bei bestimmten Landschaftsformen zu ungewünschten Effekten, wie zum Beispiel Durchdringungen, führen. Bezogen auf den realen Roboter bedeutet dies lediglich ein Anstoßen des Elevator-Reflexes.

4.2.1 Stabilität des Laufmusters

Damit der Roboter wärend des Reproduzieren des erzeugten Laufmuster nicht kippen kann, muss das Muster bestimmten Anforderungen genügen. Um ein Balancieren zu umgehen, wurde die minimale Anzahl der Beine, die sich in der Stemmphase befinden, auf drei festgelegt. Darüberhinaus wurde festgelegt, dass beim Übergang zwischen zwei aufeinander folgenden Konfigurationen maximal ein Bein seine Phase wechseln darf.

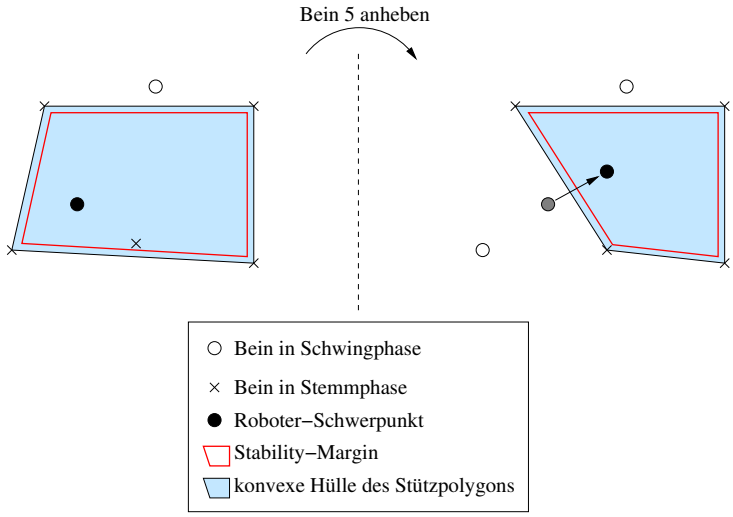

Abbildung 4.2: Verlagerung des Massenschwerpunktes beim Konfigurationsübergang

Um einen stabilen Stand zu gewährleisten, muss der Algorithmus den Massenschwerpunkt des Roboter stets innerhalb des durch die in der Stemmphase befindlichen Beine aufgespannten und auf die XY-Ebene projizierten Stützpolygons halten (Vergleiche Abbildung 4.2). Genaugenommen genügt es, den Massenschwerpunkt innerhalb der konvexen Hülle, die durch die genannten Beine entsteht, zu halten. Wird nun ein Bein angehoben, muss der Algorithmus vorher seinen Massenschwerpunkt verlagern, sodass dieser in

der neuen konvexen Hülle, die durch das Anheben des Beines entsteht, liegt. Da sowohl Anfangs- als auch Endpunkt innerhalb der konvexen Hülle liegen, befinden sich per Definition dieser Hülle auch alle Punkte der Verbindungsstrecke dieser Punkte innerhalb der konvexen Hülle. Dies garantiert einen sicheren Stand auch beim Übergang zwischen den Konfigurationszuständen.

4.2.2 Modellierung von Zuständen – Die FootConfEvent Liste

Ein *FootConfEvent* (vergleiche Abbildung 4.3) besteht aus

- der Beinkonfiguration - footConf, die bitkodierte Information, ob ein Bein den Boden berührt,
- den Beinpositionen - footPos, die Koordinaten des Aufsetzpunktes eines jeden Beines im zweidimensionalen Koordinatensystem relativ zum Körper und
- der Körperposition - center, die linearisierte Positionsangabe im Weltkoordinatensystem.

Die beiden Variablen *centerStart* und *centerEnd* (vergleiche 4.3) markieren den zulässigen Bereich für Körperbewegungen entlang des vorgegebenen Pfades.

Die für die Darstellung der Beinkonfiguration gewählte Bitkodierung hat den einzigen Vorteil, dass dadurch Speicherplatz gespart wird, was in Anbetracht des angedachten Einsatzortes eine nicht ganz unwichtige Eigenschaft ist. Programmiertechnisch ergeben sich daraus allerdings auch Nachteile. Die Gefahr von Programmierfehlern steigt, da der Variableninhalt nicht mehr einsichtig ist. Dies macht auch den Einsatz eines Debuggers schwierig.

```
struct FootConfEvent
{
  signed char footConf;
  Position footPos[6];
  float centerStart;
  float centerEnd;
  float center;

  FootConfEvent(
      signed char _footConf,
      const Position _footPos[],
      float _centerStart,
      float _centerEnd)
      :
      footConf(_footConf),
      centerStart(_centerStart),
      centerEnd(_centerEnd)
  {
    for (int i = 0; i < 6; i++)
    this->footPos[i] = _footPos[i];
  }
};
```

Abbildung 4.3: Deklaration FootConfEvent.

Für die Speicherung der Fußauftrittspunkte wurde die zweidimensionale kartesische Form im RKS[1] gewählt. Die Körperposition (*center*) hingegen

[1]Das relative Koordinatensystem (RKS) hat die gleiche Orientierung wie das Weltkoordinatensystem, jedoch liegt der Ursprung in der Körpermitte des Roboters.

steht während der Generierung der *FootConfEvent*-Liste noch nicht eindeutig fest. Lediglich die Grenzen, in denen sich der Massenschwerpunkt befinden muss (*centerStart* und *centerEnd*), sind zu diesem Zeitpunkt sicher. Nach erfolgreicher Generierung der *FootConfEvent*-Liste wird durch diese iteriert und eine geeignete Körperposition bestimmt. Auf diese Weise können nachfolgende Konfigurationen auf die Körperposition der aktuellen Konfiguration Einfluss nehmen.

4.2.3 Movement

Ein *Movement* ist die Beschreibung eines kompletten Bewegungsablaufes des Roboters über die Zeit. Die zeitkontinuierliche Darstellung ist zur programminternen Repräsentation jedoch ungeeignet, daher muss der Bewegungsablauf diskretisiert werden. Dazu wird der Bewegungsablauf an den Ereignisgrenzen erfasst, eine zugehörige Körperbewegung generiert und die entsprechenden Geschwindigkeiten bestimmt. Alle Bein- und Körperpositionen zwischen den Ereignissen können interpoliert werden.

Ein Movement besteht aus sieben *Move*-Vektoren. Sechs davon sind als *Array* angelegt und für die Beine zuständig. Der siebte Vektor beschreibt die Bewegung des Körpers.

Ein *Move* besteht aus dem *Startzeitpunkt*, der *Dauer* und dem *Ziel*, wobei das Ziel als zweidimensionale kartesische Koordinate gegeben ist. Für die Beine wird das Ziel im RKS und für den Körper im WKS[2] gespeichert. Zu Beginn eines Fuß-*Move*s hat dieser also Bodenkontakt. Nach Verstreichen der *Startzeit* wird das Bein angehoben und nach der *Dauer* am *Ziel* abgesetzt (vergleiche Abbildung 4.4). Das bedeutet, dass nach Beendigung eines *Move*s das entsprechende Bein Bodenkontakt hat. Als Konsequenz berührt

[2]Das Weltkoordinatensystem (WKS) hat seinen Ursprung in der Mitte des Terrains und ist rechtshändig. Die Z-Achse zeigt entgegen der gedachten Schwerkraft.

der Roboter nach Ende eines *Movement*s immer mit allen sechs Füßen den Boden. Bewegungen des Körpers werden ebenfalls in einem *Move* erfasst.

Abbildung 4.4: Startzeit und Dauer eines Moves

Um die *FootConfEvent*-Liste in ein *Movement* zu überführen wird für jedes Fußereignis eine Körperbewegung generiert. Auf diese Weise ist sichergestellt, dass sich der Körper beim Eintreten des Ereignisses an einer zulässigen Position befindet.

4.2.4 Random Sampling

Das *Random Sampling* erzeugt – wie der Name schon sagt – zufällig Fußauftrittspunkte und Körperpositionen, wobei bereits beim Generieren darauf geachtet wird, dass nur gültige Konfigurationen erzeugt werden, also:

- keine Füße angehoben werden, die ein Kippen des Roboters zur Folge hätten,
- keine Punkte außerhalb des Arbeitsbereiches angesteuert werden,
- kein Untergrund mit zu starkem Gefälle betreten wird
- oder Füße zusammenstoßen werden.

Der Algorithmus legt zunächst einen Pfad zum Ziel fest. Dazu erzeugt er eine zufällige Anzahl zufälliger Körperpositionen. Diese werden in eine Liste *cList*[3] eingetragen. Für den weiteren Verlauf des Algorithmus sind nur noch

[3] cList ist ein Vektor von zweidimensionalen Koordinaten.

Körperpositionen auf dem aus der *cList* gebildeten Pfad zulässig. Jetzt wird für die aktuelle Fußstellung der Bereich ermittelt, in dem sich der Körpermittelpunkt befinden kann. Dann wird zufällig ein Fuß ausgewählt, der angehoben bzw. abgesetzt wird. Soll ein Fuß abgesetzt werden, so wird zufällig eine neue gültige Fußposition bestimmt. Danach wird eine neue Körperposition innerhalb des zuvor bestimmten Bereiches festgelegt.

Da die möglichen Körperpositionen von der Fußstellung abhängen, wird der mögliche Bereich und die tatsächliche Körperposition zusammen mit den Fußkonfigurationen in der *fList*[4] gespeichert. Es genügt hier eine linearisierte Form, da der Pfad bereits durch die *cList* festgelegt ist. Das heißt, es wird lediglich gespeichert, wie weit der Roboter auf seinem Pfad bereits vorangeschritten ist.

Mit der *fList* und dem zugehörigen Pfad *cList* ist die Bewegung des Roboters vollständig beschrieben, jedoch noch ohne Angabe von Geschwindigkeiten beziehungsweise Dauer der einzelnen Bewegungen. Diese werden anschließend ermittelt, bei der Konstruktion des eigentlichen Movements.

4.3 Der Stereobildverarbeitungsalgorithmus

Der von Richard Bade an eine Echtzeitumgebung angepasste Stereobildverarbeitungsalgorithmus [1] arbeitet in drei Schritten.

Zunächst werden die beiden auszuwertenden Bilder vorverarbeitet, um das Ergebnis des Algorithmus zu verbessern. Dabei kommt zum Einen ein Medianfilter zum Einsatz, da dieser zuverlässig sogenannte Ausreißer eliminiert (beispielsweise tote Pixel im CCD Sensor), ohne das Messergebnis zu verfälschen (also das aufgenommene Bild unscharf zu machen). Prinzipiell arbeitet der Medianfilter in dem er die Intensitätswerte des Pixels und die der benachbarten Pixel sortiert und sich für das mittlere Pixel, den Me-

[4] fList ist ein Vector von FootConfEvent.

dian entscheidet. Zum Anderen wird ein Binomialfilter eingesetzt um das Bildrauschen zu reduzieren. Binomialfilter (auch Gaußfilter genannt) sind Konvolutionsfilter deren Faltungsmatrix auf einer diskreten Näherung der Gaußfunktion beruht, den Binomialzahlen. Das bedeutet, dass ein gewichteter Mittelwert eines Pixels und seiner benachbarten Pixel gebildet wird.

Der verwendete Stereobildverarbeitungsalgorithmus beruht auf dem Verfahren *Blockmatching*. Richard Bade hat diesen Algorithmus angepasst, so dass dieser nun Anytime-Anforderungen genügt. Dies wird erreicht, indem die beiden auszuwertenden Bilder zunächst verkleinert werden. So liefert das Blockmatching schnell ein Ergebnis, das allerdings von minderer Qualität ist. Der Algorithmus wird wiederholt auf die Bildpaare angewendet, wobei deren Auflösung immer höher gewählt wird. Auf diese Weise ist das Verfahren nach einer kurzen Mindestlaufzeit jederzeit abbrechbar und verbessert sein Ergebnis mit zunehmender Laufzeit.

Kapitel 5

Vorgehensweise zur Lösung

Da der von André Herms beschriebene Laufplaner [15] bereits in der Simulationsumgebung lauffähig ist, und der von Richard Bade angepasste Stereobildverarbeitungsalgorithmus [1] ein eigenständig arbeitendes Programm ist, wurde die Entscheidung getroffen zunächst in der Simulationsumgebung die Fusion zu beginnen. Um später eine Portierung auf den realen Laufroboter zu erleichtern, wurde bei der Konzeption und Implementierung bereits auf die Gegebenheiten des MCA2 geachtet.

Um den Laufplanungsalgorithmus hin zu einer inkrementellen Arbeitsweise zu modifizieren, muss die Planungsweite auf eine Teilstücklänge verkürzt werden. Die so generierten Teilstücke können dann nacheinander wiedergegeben werden. Im Algorithmus bedeutet dies, dass aus den einzelnen Movements ein Ganzes entstehen muss, das den Anforderungen nach Laufstabilität genügt. Weiter darf ein so erweitertes Movement sich im bereits vergangenen Teilbereich nicht mehr ändern.

Grundsätzlich gibt es zwei Möglichkeiten, den Algorithmus für eine inkrementelle Arbeitsweise zu modifizieren. Zum Einen ist es möglich, die *move*-Listen aneinanderzuhängen, zum Anderen können die beim *random sampling* erzeugten Ereignisse aneinandergereiht werden, um anschließend ein neues

Movement zu generieren. Beide Varianten sind mögliche Lösungen, jedoch mit Vor- und Nachteilen die abzuwägen sind und im Folgenden besprochen werden.

5.1 Verketten von Movements

Um *Movement*s zu verketten, müssen die Beinvektoren des *Movement*s, an das ein weiteres Movement angehänget werden soll, mit geeigneten *Move*s aufgefüllt werden, um am Ende einen zeitlich einheitlichen Schluss zu bekommen. Ohne dieses würden die *Move*s direkt angehängt werden, was einen unterschiedlichen Anfangszeitpunkt für die einzelnen Bein-Vektoren zur Folge hätte und unkontrollierte Roboterkonfigurationen verursacht.

Die zum Auffüllen nötigen "leeren" *Move*s müssen eine *Dauer* von 0 Sekunden haben, da dies die Zeitspanne ist, in der das Bein angehoben ist. Eine längere Ereignisdauer hätte ein gleichzeitiges Anheben aller Beine zur Folge und resultiert damit zwangsläufig in einem Fehler.

Ein Verketten von *Movement*s ist leicht zu implementieren, da lediglich die Bein-Vektoren mit "leeren" *Move*s aufgefüllt werden müssen und die Zielpositionen bereits bekannt sind. Das eigentliche Verketten ist dann ein simples Verknüpfen zweier Vektoren (vergleiche Abbildung 5.1).

Nachteil dieses Verfahrens ist, dass zwischen den einzelnen *Movement*s der Roboter immer wieder kurze Bewegungspausen einlegen muss, um wieder mit allen sechs Beinen Bodenkontakt zu bekommen. Dies verhindert eine flüssige Bewegung und der Bewegungsablauf wirkt abgehackt, besonders für kurze Planungslängen. Dies kann bis zum pentapoden Gang (es berühren also immer fünf Beine den Boden) ausarten. Ein weiterer Nachteil ist, dass es auf diese Weise nicht möglich ist, bei einem unerwarteten Hindernis mit der aktuellen Roboterkonfiguration weiter zu planen. Wird beispielsweise ein Hindernis erkannt während ein Bein den Boden nicht berührt, kann aus dieser

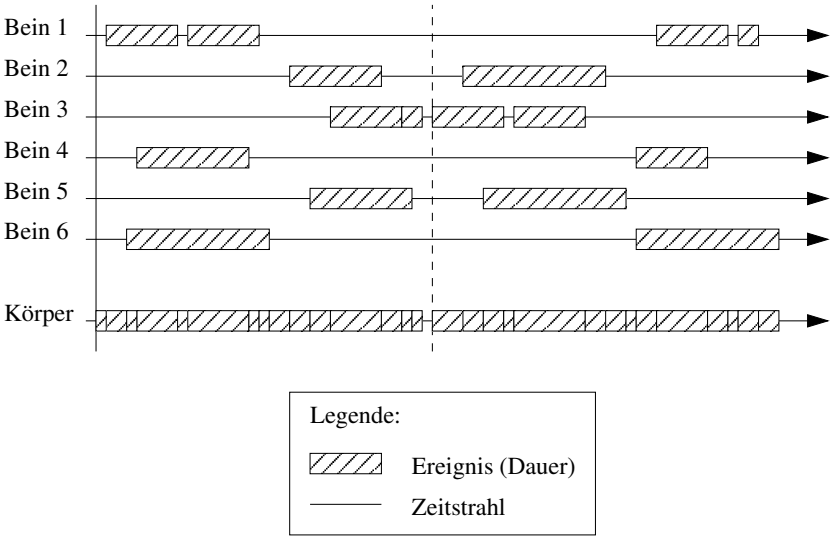

Abbildung 5.1: Verkettete Movements.

Position keine weitere Planung erfolgen, da der Algorithmus erwartet, dass alle Beine den Boden berühren.

5.2 Verketten von FootConfEvents

Um nahtlos eine weitere Planung ansetzen zu können, muss der Algorithmus so angepasst werden, dass er mit beliebigen Roboterkonfigurationen zu Beginn umgehen kann. Der so modifizierte Algorithmus kann nun mit der Konfiguration, die der Roboter am Ende des vorigen *Movements* hat, als Initialkonfiguration angestoßen werden. Das so erzeugte *Movement* enthält neben den *Move*-Vektoren auch die Liste der *FootConfEvents*.

Soll Movement1 um Movement2 erweitert werden, überprüft man ob das letzte *FootConfEvent* von Movement1 gleich der Initialposition von Movement2 ist und verkettet die beiden *fList*en. Die beiden *cList*en, die den Pfad zum Ziel beschreiben müssen ebenfalls aneinandergehängt werden. Da dies

den Pfad verlängert, müssen alle linearisiert gespeicherten Körperpositionen und Bereiche in der *fList* des Movement2 zuvor berichtigt werden. Nun kann das Movement neu konstruiert werden.

Auf diese Weise wird ein Movement erzeugt, das einen flüssigen Bewegungsablauf zulässt. Auch ein unerwartetes Hindernis stellt kein Problem dar, da der Algorithmus mit beliebigen Fußstellungen als Initialisierung umgehen kann. Dieses Verfahren ist jedoch rechenintensiver als das schlichte Aneinanderreihen von fertigen Movements.

5.3 Fehler im Algorithmus Random Sampling

Das *Random Sampling* erzeugt nur gültige Roboterkonfigurationen und ist so angelegt, dass beim Übergang der aufeinanderfolgenden Konfigurationen keine ungültigen Stellungen des Roboters entstehen sollen. Um dies zu gewährleisten, wird die Kippstabilität des Roboters gesichert, indem der Körperschwerpunkt in die konvexe Hülle der Fußauftrittspunkte der Beine, die sich in der Stemmphase befinden, gelegt wird. Die folgende Roboterkonfiguration resultiert aus dem Bewegungsradius, der sich aus der konvexen Hülle der vorherigen Konfiguration ergibt. Dadurch ist sichergestellt, dass der Geradenausschnitt zwischen aufeinanderfolgenden Körperpositionen immer innerhalb der zugehörigen konvexen Hülle liegt (wie in 4.2.1 beschrieben).

Nicht alle in der konvexen Hülle liegenden Punkte sind mit dem Massenschwerpunkt erreichbar. Die erreichbaren Positionen werden durch den Arbeitsbereich der Beine weiter eingeschränkt. Spreizt der Roboter beispielsweise alle Beine soweit wie möglich von sich, kann der Roboter seinen Schwerpunkt nicht mehr verlagern. Dieser Grenzfall zeigt, dass durch den Beinarbeitsbereich die Erreichbarkeit innerhalb der konvexen Hülle weiter eingeschränkt sein kann. Dieser Fall wird durch die Funktion *calcCenterRegion()* abgefangen.

Darüber hinaus besteht aus dem gleichen Grund, also dem Beinarbeitsbereich, eine Einschränkung der Fußauftrittspunkte. Auch dies wird im Algorithmus beachtet und verhindert, indem nur Fußauftrittspunkte innerhalb des jeweiligen Beinarbeitsbereich generiert werden.

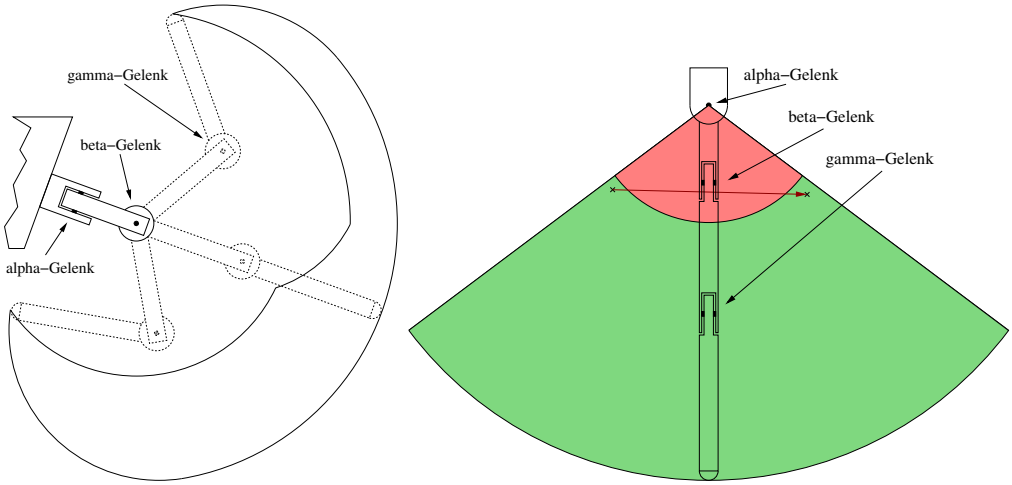

Abbildung 5.2: Arbeitsbereich eines Beines in XZ-Ebene und XY-Ebene.

Jedoch ist der Beinarbeitsbereich durch die maximalen Winkelstellungen der Gelenke weiter eingeschränkt (vergleiche Abbildung 5.2). Der tatsächliche Beinarbeitsbereich entsteht durch Überlagerung der erreichbaren Punkte je Gelenk. Der so gebildete Bereich ist jedoch nicht konvex und somit kann für zwei gültige Beinpositionen der Übergang von der einen zur anderen Beinposition unzulässige Zustände herbeiführen.

Für ein Bein in der Schwingphase bedeutet dies lediglich, dass die Beintrajektorie, um das Bein vom einen zum anderen Punkt zu bewegen, so bestimmt werden muss, dass diese innerhalb des Arbeitsraumes des Beines liegt.

Für ein Bein in der Stemmphase bedeutet dies allerdings, dass der Konfigurationsübergang tatsächlich ungültige Positionen erzwingt. In diesem Fall bedeutet ein Konfigurationsübergang, dass der Roboter seinen Körper an ei-

ne nicht erreichbare Stelle schieben soll. Beim realen Roboter verhindern dies die Anschläge der Gelenke. Für den Algorithmus bedeutet dies, dass aus dieser Position nicht weiter geplant werden kann, da bereits die Initialposition ungültig ist.

5.4 Implementierung

Für die Implementierung wurde das Konkatenieren der *FootConfEvent*-Listen mit anschließender Rekonstruktion des *Movement*s gewählt, da nur so ein flüssiger Bewegungsablauf erzeugt werden kann.

5.4.1 Erzeugen des Movements

Umgehen des Fehlers Um dem in 5.3 aufgezeigten Problem zu begegnen, kann die Beintrajektorie in der Schwingphase angepasst werden. Die Körperbewegung jedoch muss eingeschränkt werden. Eine der einfachsten Möglichkeiten dies zu umgehen ist, den Beinarbeitsbereich weiter einzuschränken, um somit wieder eine konvexe Hülle als Schnitt des Stützpolygons und des Bewegungsbereiches zu erreichen (vergleiche Abbildung 5.3).

In dieser Arbeit wurde der Beinarbeitsbereich beschnitten, indem alle Fußpositionen, die sich näher am Körper als eine geeignete Schranke befinden, schlicht verboten wurden. Konkret heißt das, der Beinarbeitsbereich wurde in Y-Ritung auf einen Abstand von mindestens 320[1] zur Körpermitte beschnitten.

Anpassung des Zieles Eine weitere Anforderung des Ausgangsalgorithmus ist, dass er das vorgegebene Ziel immer erreichen soll. Dazu wurde der Liste der Körperpositionen (*cList*) nach deren Erzeugung immer auch die

[1] Ohne Einheit, da es sich um Koordinaten im OpenInventor handelt.

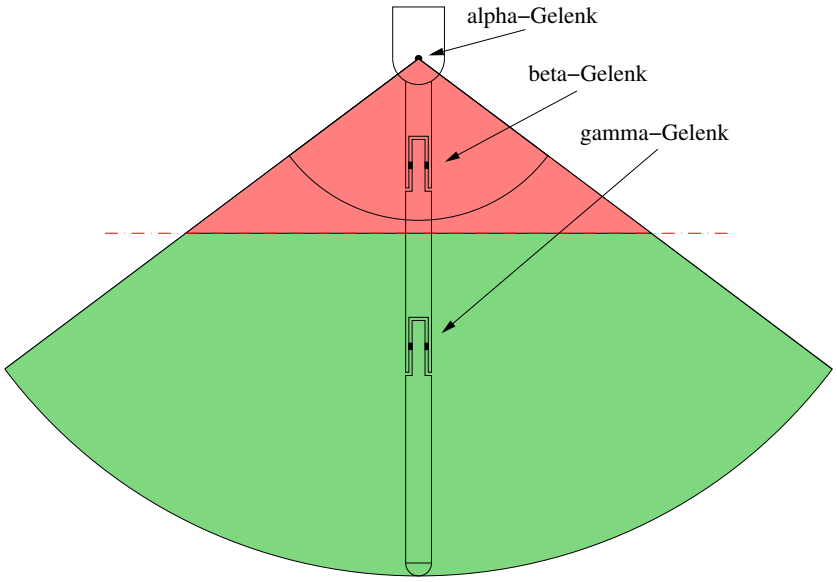

Abbildung 5.3: Beschneidung des Beinarbeitsbereiches.

Zielposition hinzugefügt. Dies sorgt dafür, dass der Algorithmus den Roboter auf jeden Fall zum Ziel bringt. Jedoch ist das in der inkrementellen Planung nicht immer wünschenswert. Soll der Roboter beispielsweise einen Graben überwinden, zwingt der Algorithmus den Roboter auf dem letzten Teilstück des Pfades, den Graben auf jeden Fall zu überwinden. In einem solchen Fall wäre die gewünschte Vorgehensweise, zunächst möglichst nahe an den Graben zu kommen und dann im nächsten Planungsschritt den Graben schließlich zu überwinden.

Daher wurde die Vorgabe, das Ziel zu erreichen, fallengelassen und mittels Bewertungsfunktion ein zielgerichtetes Laufen ermöglicht. Dazu wird der Bewertung $\frac{verbleibendeWegstrecke}{Planungslnge} \cdot Gewichtungsfaktor$ abgezogen, wobei bei einer verbleibenden Wegstrecke, die größer ist als die Planungslänge die Lösung für ungültig erklärt wird.

5.4.2 Stabilität des Movements

Bei der Konstruktion eines Movements aus den Listen *fList* und *cList* heraus wird für jedes Ereignis in der *fList* ein dazugehöriges *centerMove* erzeugt, um den Körper in die für das Ereignis nötige Position zu bewegen. Das heißt, es werden *centerMove*s sowohl beim Eintritt eines Beines in eine Stemmphase, als auch beim Eintritt in eine Schwingphase erzeugt. Die eigentlichen Beinbewegungen können allerdings erst beim Eintreten in die Stemmphase erzeugt werden, da erst zu diesem Zeitpunkt die Bewegung vollständig ist (vergleiche Kapitel 4.2.3).

Dies bedeutet, dass ein Movement, an dessen Ende der Roboter nicht mit allen Beinen den Boden berührt, nicht bis zum Schluss einen korrekten Bewegungsablauf beschreibt. Wird nun an ein solches Movement ein passendes folgendes Movement angehängt, werden die Fußbewegungen konstruiert, wodurch sich zum Einen völlig andere Roboterstellungen ergeben können, zum Anderen sich die Zeiten der Körperbewegungen noch einmal ändern können, da diese von den Fußbewegungen abhängen.

5.4.3 Die Klasse GaitPlanner

Da beim inkrementellen Generieren von Laufmustern nach der beschriebenen Methode das erzeugte *Movement* nicht bis zum Ende stabil ist, kann die in der Klasse *Controll* implementierte Funktion *update()* nicht zum Aktualisieren der Darstellung verwendet werden. Darüberhinaus ist nicht garantiert, dass der Algorithmus rechtzeitig eine gültige Lösung findet. In einem solchen Fall muss der interne Zeitzähler angehalten werden (bzw. er darf nicht erhöht werden) um ein Pausieren zu simulieren. Daher wurde in der Klasse GaitPlanner auch das Aktualisieren der Anzeige implementiert.

Beim Konstruktoraufruf der Klasse GaitPlanner werden zwei weitere *Movement*s angelegt, das aktuell beste *bestMovement* und das temporäre mit

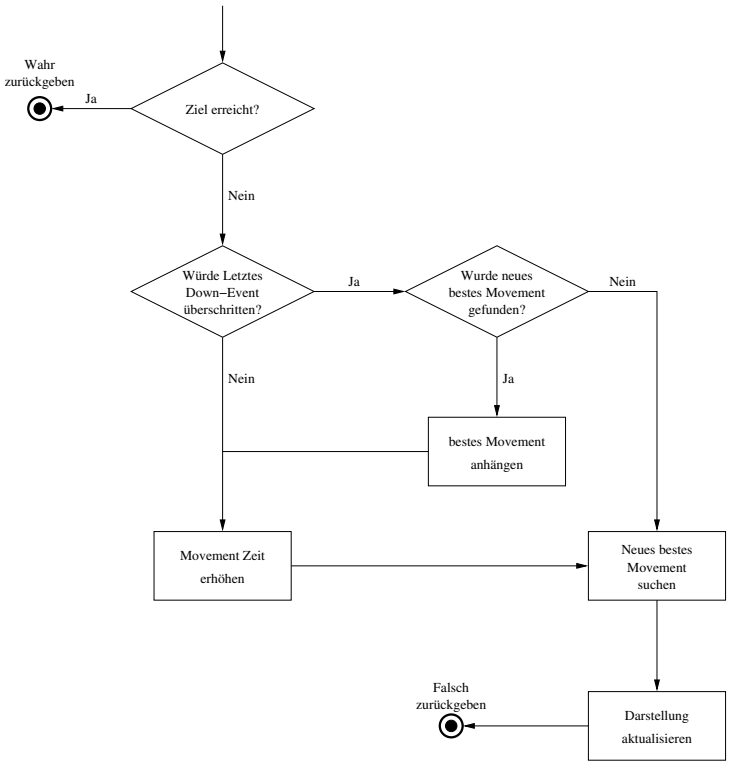

Abbildung 5.4: Flussdiagramm der Funktion GaitPlanner::update().

dem gearbeitet wird *tmpMovement*. Beide werden mit *-Movement::Mega*, dem Wert für ein ungültiges *Movement*, initialisiert.

Um den Schreitprozess in Gang zu setzen wird die Funktion *update()* der Klasse *Controll* in OpenInventor eingeplant. Diese Funktion plant sich selbst immer wieder neu ein, solange das Ziel nicht erreicht wurde. Im Falle des sukzessiven Planens wird das Aktualisieren der Darstellung umgangen und stattdessen die *update()* Funktion des *GaitPlanners* aufgerufen (siehe Abbildung 5.4).

Dieser überprüft zunächst ob das Ziel bereits erreicht wurde. Ist dies der Fall, wird die Planung beendet. Danach wird getestet, ob im nächsten Schritt

die Stabilitätsgrenze des *Movement*s überschritten würde. Bei einem Überschreiten dieser Grenze muss zuvor das aktuell beste *Movement* angehängt werden. Ist kein gültiges *bestMovement* vorhanden, wird der Schreitzyklus unterbrochen. Würde die Stabilitätsgrenze nicht überschritten, kann im alten *Movement* fortgefahren werden. So hat der Laufplanungsalgorithmus weiter Zeit, sein Ergebnis zu verbessern.

5.4.4 Integration der Stereobildverarbeitung

Um der Stereobildverarbeitung auch in der Simulationsumgebung zwei Bildpaare liefern zu können, müssen die in OpenInventor angelegten Kameras umgeschaltet werden. Dies ist nötig, da das eigentliche Rendering der Szene durch die openGL fähige Grafikkarte geschieht. Sobald auf den Kamera-Viewport umgeschaltet wurde, kann ein Screenshot gemacht werden. Dazu wird direkt mittels openGL auf das generierte Bild zugegriffen, und die so gewonnenen Daten in eine Datei gespeichert. Nachdem auch die zweite Kamera ein Bild geliefert hat, kann wieder zur Szenen-Kamera zurückgeschaltet werden.

Um aber ein ständiges Umschalten zwischen den Kameras und somit ein Flimmern des Bildes zu vermeiden, werden diese Schaltvorgänge innerhalb der gleichen in OpenInventor eingeplanten Funktion abgearbeitet. Das heißt, man kann diese nicht in einen Thread auslagern. Auf diese Weise wird OpenInventor davon abgehalten in die Main-Loop einzutreten und somit wird die Darstellung unterbunden.

Da die generierten Bilder anders als tatsächliche Aufnahmen keine Textur erkennen ließen, wurde ein Texturmapping implementiert. OpenInventor stellt hierzu die Klasse SoTexture2 zur Verfügung, die eine Textur aus einem JPEG Bild laden kann. Nach dem instanziieren eines Objektes dieser Klasse, muss die Textur noch skaliert werden. Dazu wird das Objekt in

ein SoTexture2Transform Objekt gehüllt und dieses kann schließlich in den Objektbaum als Kind des mit Textur zuversehenden Objektes eingehängt werden.

Um mit dem Stereobildverarbeitungsalgorithmus möglichst gute Resultate zu erzielen, wählt man eine Textur mit möglichst hoher Auflösung. Eine hohe Auflösung hat zur Folge, dass sich die Textur nicht zu häufig wiederholt, somit sinkt die Wahrscheinlichkeit, dass der Roboter mehrmals die gleiche Oberfläche sieht.

Dadurch, dass das Speichern der Kamerabilder in eine Datei erfolgt, ist es möglich, den Stereobildverarbeitungsalgorithmus auszulagern. Auf dem realen Roboter ist dazu ein Video-Streaming, wie es in [35] vorgestellt wird, nötig.

Kapitel 6

Ergebnisse und Messungen

Da die Qualität der Lösung bei dem verwendeten Anytime-Algorithmus von der Laufzeit desselben abhängt, stellt sich die Frage in welchem funktionellen Zusammenhang Laufzeit und Qualität stehen. Anhand dieses Zusammenhangs lässt sich auch abschätzen ob beziehungsweise wieviel Rechenzeit für die Auswertung der Stereobildverarbeitung zur Verfügung steht.

Aber nicht nur diese Frage soll in diesem Kapitel untersucht werden. Eine weitere zu untersuchende Frage ist, welche Terrains von dem Algorithmus bewältigt werden. Weiter wird außerdem untersucht, in welchem Zusammenhang die Geländeform und die Laufzeit des Algorithmus stehen.

Die im Weiteren verwendeten Geländeformen wurden zum besseren Verständnis, wie aus Abbildung 6.1 zu entnehmen ist, benannt.

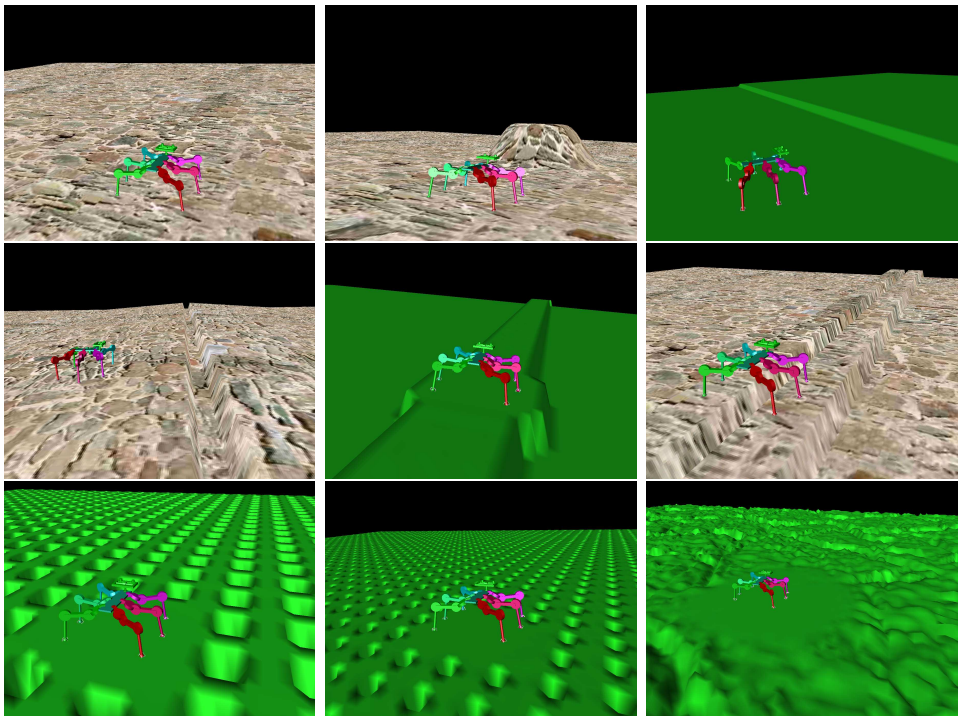

Abbildung 6.1: Namen der Geländeformen (oben links beginnend, zeilenweise): Flachland, Hindernis, Stufe, Graben, Steg, Doppelsteg, große Löcher, kleine Löcher, zufällig

6.1 Geschwindigkeit des Algorithmus

Alle Messungen wurden mit einer Planungslänge von 700 Einheiten[1] vorgenommen. Es werden die ersten 10 Teilstücke als Messgrundlage verwendet. Das Hindernis (bei den Terrains Graben, Hindernis und Stufe) befindet sich gerade an der Grenze zur Planungslänge. Für die Messungen wurde ein 3 GHz Rechner (Intel© Pentium© 4) verwendet. Die verwendete Grafikkarte ist eine nVidia GeForce4 Ti 4600.

Die in den folgenden Tabellen aufgelisteten Werte, sind Mittelwerte des jeweiligen Teilstücks, basierend auf 10 Messungen.

Teilstück Nummer	Zeit zwischen Ereignissen (sec)	Anzahl Durchläufe	Anzahl korrekter Lösungen	Anzahl Verbesserungen
1	0,191659	1	1	1
2	6,908941	41,7	22,4	2
3	18,275197	110,8	22,9	2,9
4	25,45305	156,1	4,5	1,4
5	18,214673	111,1	1	1
6	14,503972	89,1	1,7	1,5
7	12,5979048	76,5	1,8	1,2
8	20,099289	122,6	1,1	1
9	41,319759	252	1	1
10	217,553118	1331,7	1,2	1
Gesamt:	37,51175628	229,26	5,86	1,4

Tabelle 6.1: Messung “Flachland”

[1] im Koordinatensystem des OpenInventors

Teilstück Nummer	Zeit zwischen Ereignissen (sec)	Anzahl Durchläufe	Anzahl korrekter Lösungen	Anzahl Verbesserungen
1	0,1902644	1	1	1
2	7,139572	43,5	25	2
3	19,523684	119,7	28,9	4
4	18,23221	110,7	3,8	1,5
5	11,569948	71	1	1
6	8,7424785	53,6	2,8	1,3
7	32,59528	199,6	2,8	1,1
8	76,709263	469,2	1,1	1
9	47,688364	291,9	1	1
10	194,8602313	1194	1,1	1
Gesamt:	41,72512952	255,42	6,85	1,49

Tabelle 6.2: Messung “Hindernis”

Teilstück Nummer	Zeit zwischen Ereignissen (sec)	Anzahl Durchläufe	Anzahl korrekter Lösungen	Anzahl Verbesserungen
1	0,1943951	1	1	1
2	7,0205	42,8	23,2	2
3	18,385215	111,1	23,6	3,8
4	18,58373	113,2	5,8	1,8
5	9,3806636	57,4	1,2	1
6	6,7191645	39,3	2,5	1,4
7	7,3173796	40,8	2,4	1,2
8	38,9612063	234,9	3,3	1,4
9	56,6733394	346,6	2,2	1,1
10	38,0312837	232,8	1,2	1,1
Gesamt:	20,12668772	121,99	6,64	1,58

Tabelle 6.3: Messung “Stufe”

Teilstück Nummer	Zeit zwischen Ereignissen (sec)	Anzahl Durchläufe	Anzahl korrekter Lösungen	Anzahl Verbesserungen
1	0,1920221	1	1	1
2	5,928805	36,3	18,1	3,2
3	18,279318	112,3	25,6	3,2
4	11,315255	69,1	9,2	2,4
5	15,111072	91,8	4,8	1,7
6	17,0130366	103,7	5,8	2
7	16,18352	98,5	3,6	1,4
8	30,614138	186,9	4,4	1,2
9	33,10644	201,3	6,9	1,7
10	44,288116	271,3	7,4	2,4
Gesamt:	19,20317227	117,22	8,68	2,02

Tabelle 6.4: Messung “Graben”

Teilstück Nummer	Zeit zwischen Ereignissen (sec)	Anzahl Durchläufe	Anzahl korrekter Lösungen	Anzahl Verbesserungen
1	0,1932374	1	1	1
2	6,855194	41,6	22,7	2
3	18,007974	110,5	22,5	3,9
4	17,21371	104,9	6,9	2,2
5	29,6168727	179,8	4,1	1,4
6	13,6029793	83,7	2,7	1,4
7	6,1899758	37,8	3	1,2
8	25,8031751	157	1,1	1,1
9	24,7133383	151,3	1	1
10	105,3338587	643,4	2,2	1,5
Gesamt:	24,75303153	151,1	6,72	1,67

Tabelle 6.5: Messung “Steg”

Teilstück Nummer	Zeit zwischen Ereignissen (sec)	Anzahl Durchläufe	Anzahl korrekter Lösungen	Anzahl Verbesserungen
1	0,1926884	1	1	1
2	6,768067	40,9	18,5	2
3	17,26101	105,8	21,6	2,2
4	22,23374	136,4	1,2	1,2
5	3,3999833	20,4	2,1	1,3
6	11,6613218	70,8	2,4	1,2
7	11,992971	73,2	1,7	1
8	75,7904097	456,6	2,9	1,4
9	10,5209023	64,1	3,7	1,7
10	22,482344	136,5	1,8	1,5
Gesamt:	18,23034375	110,57	5,69	1,45

Tabelle 6.6: Messung “Doppelsteg”

Teilstück Nummer	Zeit zwischen Ereignissen (sec)	Anzahl Durchläufe	Anzahl korrekter Lösungen	Anzahl Verbesserungen
1	0,2045047	1	1	1
2	5,262266	31,3	3	2
3	9,312682	57,1	1,1	1,1
4	6,968475	42,1	2,2	1,3
5	14,274245	86,9	1,7	1
6	4,6398156	28	1,7	1,1
7	5,3475447	31,6	1,8	1,3
8	9,0047336	54	2,4	1,1
9	11,749372	69,1	4,5	1,9
10	7,4983164	44,8	2,8	1,7
Gesamt:	7,4261955	44,59	2,22	1,35

Tabelle 6.7: Messung “große Löcher”

Teilstück Nummer	Zeit zwischen Ereignissen (sec)	Anzahl Durchläufe	Anzahl korrekter Lösungen	Anzahl Verbesserungen
1	0,1979399	1	1	1
2	5,759706	32,5	4,6	2,4
3	10,565459	64	4,3	1,8
4	6,1426564	37,8	6,1	2,1
5	17,636971	107,2	9,5	1,9
6	6,4101	39,4	7	3
7	9,346303	57	4,4	1,6
8	9,490688	56,6	6,4	1,6
9	14,251228	86,8	1,6	1,2
10	29,4780835	179,2	1,1	1,1
Gesamt:	10,92791348	66,15	4,6	1,77

Tabelle 6.8: Messung “kleine Löcher”

Teilstück Nummer	Zeit zwischen Ereignissen (sec)	Anzahl Durchläufe	Anzahl korrekter Lösungen	Anzahl Verbesserungen
1	0,1886461	1	1	1
2	7,219636	43,7	27	2
3	20,19502	123,4	34,2	4,1
4	25,256639	154,6	13,4	2,7
5	10,760986	65,7	5,1	1,9
6	14,918187	91,3	2,5	1,1
7	20,8272004	127	3,1	1,7
8	43,2586598	265,2	1,3	1,1
9	44,2570211	269,9	1,5	1,2
10	140,0805432	855,1	1	1
Gesamt:	32,69625386	199,69	9,01	1,78

Tabelle 6.9: Messung “Zufällig”

6.2 Geländeformen

Da das Pfadsuchen nicht in den Verantwortungsbereich dieses Laufplaners fällt, und zu Testzwecken lediglich ein rudimentäres Routen vorgenommen wird, stellt ein echtes Hindernis, das der Roboter nicht überwinden kann, eine unlösbare Aufgabe dar. Wie die Tests zeigten hat der Roboter dieses Szenario auch nicht bewältigt. In der von André Herms implementierten Version des Laufplanungsalgorithmus war dies noch möglich, da dieser den kompletten Weg plant. Wird der Weg zum Ziel aber geradlinig in Teilstücke aufgeteilt, kommt es zu der Situation, dass der Roboter als Teilziel das Hindernis besteigen müsste, was jedoch nicht möglich ist, da die Anhöhe absichtlich zu steil gewählt wurde.

Weiter wurde auch das Szenario "kleine Löcher" nicht bewältigt. Der Grund hierfür liegt im Algorithmus. Dieser interpoliert die Steigung an einer Position anhand der Höhenkarte, die die diskrete Information in einem Raster liefert. Im Terrain "kleine Löcher" liegen Höhen und Tiefen so eng beisammen, so dass die Interpolation ein zu steiles Gefälle ergibt, als dass der Algorithmus diese Position als sicheren Fußauftrittspunkt erkennt.

Alle anderen Szenarien bewältigt der inkrementelle Algorithmus, jedoch zum Teil mit beträchtlichem Zeitaufwand.

Kapitel 7

Zusammenfassung und Ausblick

In dieser Arbeit wurde der von André Herms [15] vorgestellte Algorithmus auf eine inkrementelle Arbeitsweise angepasst, um ihn so für einen späteren Einsatz im realen Laufroboter vorzubereiten. Für die Integration der Stereobildverarbeitung wurde der Grundstein gelegt. Es ist nun möglich, während der Laufplanung den Algorithmus anzustoßen, jedoch verwendet der Laufplaner noch nicht die Ergebnisse des Algorithmus.

Der in dieser Arbeit vorgestellte inkrementell arbeitende Laufplaner hat gezeigt, dass er in der Lage ist, auch schwierige Geländeformen wie das Laufen auf einem schmalen Steg, Löcher, und sogar zufällig strukturiertes Gelände zu bewältigen. Die Laufzeiten, bis eine gültige Lösung gefunden wird, sind dabei aber deutlich zu lange. Da der Algorithmus aber gut parallelisierbar ist, wäre hier eine Auslagerung auf ein externes Cluster denkbar. Eine weitere Herangehensweise könnte darin bestehen, das *Random Sampling* zu verbessern, um bereits im Voraus ungültige Lösungen auszuschließen.

Da das entstehende Laufmuster nicht optimal ist, sollte es mit statischen und reaktiven Verfahren kombiniert werden. Dies wäre möglich, indem man mit den aus der Stereobildverarbeitung gewonnenen Daten während des Schreitens nach statischem Laufmuster unstrukturierten Untergrund oder

Hindernisse erkennt und nur für diesen Fall das plandende Laufen verwendet. Tritt wärend der geplanten Fortbewegung nun ein "übersehenes" Hindernis auf, kann ein reaktives Verfahren eingesetzt werden.

Abbildungsverzeichnis

Literaturverzeichnis

[1] Bade, Richard. *Modifikation einer geeigneten Stereobildverarbeitungsmethode für die Anwendung in einer Echtzeitumgebung.* http://ivs.cs.uni-magdeburg.de/EuK/mitarbeiter/ribade/diplomarbeit_rbj%.pdf 2003

[2] Bade, Richard ; Ihme, Thomas. *Verfahren zur hierarchischen Stereobildverarbeitung zur Umgebungsmodellierung unterstützt durch Bewegungsinformationen.* http://www-ivs.cs.uni-magdeburg.de/EuK/mitarbeiter/ribade/robotik2004_%bade_ihme.pdf 2004

[3] Beckert, Axel. *Kompilierung von Anytime-Algorithmen.* http://fsinfo.cs.uni-sb.de/~abe/w5/ORCAN2/Kompilation%20von%20Anytime-%Algorithmen%3A%20Konzeption,%20Implementation%20und%20Analyse.ps.bz2 2000

[4] Berns, Karsten. *Walking Machine Catalogue.* http://www.walking-machines.org 2006

[5] Burt, P. J.: The pyramid as a structure for efficient computation. In: *Springer Series in Information Sciences* 12 (1984)

[6] Cordes, Stefan ; Berns, Karsten. *A Flexible Hardware Architecture for the Adaptive Control of Mobile Robots.* http://www.fzi.de/KCMS/kcms_file.php?action=link&id=252

[7] Cruse, Holk [u. a.: Walking: A complex Behaviour Controlled by Simple Networks. In: *Adaptive Behaviour* 3 (1995), S. 385–418

[8] Eldershaw, Craig. *Heuristic algorithms for motion planning.* `http://www2.parc.com/hsl/members/celdersh/papers/thesis.pdf` 2001

[9] Eldershow, Craig ; Yim, Mark: Motion planning of legged vehicles in an unstructured environment. In: *IEEE International Conference on Robotics and Automation* (2001)

[10] Ens, John ; Lawrence, Peter: An Investigation of Methods for Determining Depth from Focus. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 15 (1993), Nr. 2, S. 97–108

[11] Ferrell, Cynthia: A comparison of three insect-inspired locomotion controllers. In: *Robotics and Autonomous Systems* 16 (1995), S. 135–159

[12] Frommberger, Lutz. *3D-Umweltmodellierung zur Navigationsunterstützung von Laufmaschinen im Gelände.* `http://www.aussagekraft.de/files/Diplomarbeit_Frommberger.pdf` 2002

[13] Gassmann, Bernd. *Erweiterung einer modularen Laufmaschinensteuerung für unstrukturiertes Gelände.* 2000

[14] Guddat, Martin. *Autonome, adaptive Bewegungskoordination von Gehmaschinen in komplexer Umgebung.* `http://deposit.ddb.de/cgi-bin/dokserv?idn=966303652&dok_var=d1&dok_ext%=pdf&filename=966303652.pdf` 2002

[15] Herms, André. *Entwicklung eines verteilten Laufplaners basierend auf heuristischen Optimierungsverfahren*

[16] Ihme, Thomas. *Steuerung von sechsbeinigen Laufrobotern unter dem Aspekt technischer Anwendungen.* `http://diglib.uni-magdeburg.de/Dissertationen/2002/thoihme.pdf` 2002

[17] Jiang, Xiaoyi ; Bunke, Horst: *Dreidimensionales Computersehen.* Berlin : Springer-Verlag, 1995. – ISBN 3–540–60797–8

[18] Jiménez, María A. ; Santos, P. González d.: Terrain-Adaptive Gait for Walking Machine. In: *The International Journal of Robotics Research* 16 (1997), Nr. 3, S. 320–339

[19] Knieriemen, Thomas: *Autonome mobile Roboter.* B.I. Wissenschaftsverlag, 1991. – ISBN 3–411–15031–9

[20] Kopka, Helmut: *LaTeX Band 1: Einführung.* Bd. 3. Pearson Studium, 2000. – ISBN 3–8273–7038–8

[21] Kubow, T. M. ; Full, R. J.: The role of the mechanical system in control: a hypothesis of self-stabilization in hexapedal runners. In: *Royal Society London* 354 (1999), S. 849–861

[22] Lewis, M. A. ; Fagg, Andrew H. ; Beckey, George A. *Genetic Algorithms for Gait Synthesis in a Hexapod Robot.* 1994

[23] Pal, Prabir K. ; Kar, Dayal C.: Gait Optimization through Search. In: *The International Journal of Robotic Research* 19 (2000), S. 394–408

[24] Parker, Garry B. ; Braun, David W. ; Cyliax, Ingo. *Learning Geits from the Stiquito.* 1997

[25] Parker, Garry B. ; Braun, David W. ; Cyliax, Ingo. *Evolving Hexapod Gaits using a cyclic Genetic Algorithm.* 2000

[26] Pfeiffer, Friedrich ; Eltze, Jürgen ; Weidemann, Hans-Jürgen: Six-legged technical walking considering biological principles. In: *Robotics and Autonomous Systems* 14 (1995), S. 223–232

[27] PlusTech. *Walking Forest Machine.* http://www.robotory.com/pics/plustech/6x6_02.jpg

[28] Schamburek, Jan-Ullrich. *Bewegungssteuerung bei Insekten.* http://goethe.ira.uka.de/~feldbus/Seminar-SS03/ausarbeitungen/Bewegung%ssteuerung_bei_Insekten.pdf 2003

[29] Scholl, K.-U. ; Albiez, J. ; Gassmann, B. ; Zöllner, J. M.: *VDI-Berichte: MCA - Modular Controller Architecture.* Düsseldorf : VDI Verlag, 2002. – ISBN 3–18–091679–6

[30] Scholl, Kay-Ulrich. *MCA2 Introduction.* http://mca2.org/introduction.html

[31] Song, Shin-Min ; Waldron, Kenneth J. *An Anaytical Approach for Gait Study and Its Applications on Wave Gaits.* 1987

[32] Thiel, Christian. *Verdeckungsanalyse in Stereobildern unter der Verwendung von Disparity-Space-Images und dynabooklether Programmierung.* http://www.informatik.uni-mannheim.de/pi4/lectures/ws0203/seminar_data%/ChristianThiel.pdf 2003

[33] Tönnies, Klaus D.: *Grundlagen der Bildverarbeitung.* München : Pearson Studium, 2005. – ISBN 3–8273–7155–4

[34] Wahlster, Wolfgang ; Tack, Werner: Ressourceenadaptive Kognitive Prozesse. In: *Informatik als Innovationsmotor* 27 (1997), S. 51–57

[35] Weber, Marc ; Herbert, David ; Jochim, Peter. *Videoübertragung über eine WLAN-Funkstrecke.* 2006

[36] Wetzelsberger, Kai. *Konzeption und Implementierung von sensorbasierten Bewegungsalgorithmen für einen Laufroboter.* 2005

[37] Yim, M. ; Duff, D. G. ; Roufas, K. D.: PolyBot: a modular reconfigurable robot. San Francisco, California, USA, 2000