Baccalaureatsarbeit

Follow-Right-Wall Algorithmus basierend auf der Nutzung von Verhaltensschichten

(Nach Rodney A. Brooks)

Sascha Jockel
Limosenweg 6 B, 22547 Hamburg
Tel.: 040/83210421, Mail: sasajo@web.de

Hamburg
26. Juli 2002

Durchgeführt im Rahmen des Projektes Navigationsunterstützung für Menschen und Roboter

Betreuer: Prof. Christian Freksa, PhD

Arbeitsbereich Wissens- und Sprachverarbeitung
Fachbereich Informatik
Universität Hamburg

Inhaltsverzeichnis

1 Einleitung **5**

2 Projektbeschreibung **6**

2.1 Allgemeine Zielsetzung des Projektes . . . 6
2.2 Arbeitsmittel . . . 6
2.2.1 Sensorik des Roboters . . . 7
2.2.2 Framework *mapsNrobots* . . . 8
2.3 Projektgruppen . . . 11
2.3.1 Exploration . . . 11
2.3.2 Kartierung . . . 15
2.3.3 Präsentation . . . 15

3 Verhaltensschichten gemäß des Ansatzes von R. A. Brooks **16**

3.1 Anforderungen an die Kontrollstruktur . . . 16
3.2 Traditionelle Architekturen . . . 17
3.3 Subsumption-Architektur & Kompetenzebenen . . . 18
3.4 Kontrollschichten . . . 20

4 Entwicklung des F-R-W Algorithmus **22**

4.1 Ziel . . . 22
4.2 Rahmenwerk . . . 22
4.3 Module . . . 24
4.3.1 Sense . . . 24
4.3.2 Collision Avoidance . . . 24
4.3.3 Wander . . . 25
4.3.4 Feelforce . . . 26
4.3.5 Attract to Wall . . . 27
4.3.6 Straight Walker . . . 28
4.3.7 Adjuster . . . 29
4.3.8 Corner . . . 30

4.4 Verhalten und Beziehungen der Module 32
4.5 Testlauf 35

5 Bewertung und Ausblick **39**

Abbildungsverzeichnis

2.1 Pioneer I Roboter mit Laser-Range-Finder 7
2.2 Laser-Range-Finder: LMS 200 der Firma SICK 8
2.3 Umgebungsrepräsentierender Scan des Labors 9
2.4 Kommunikator 9
2.5 Simulationsumgebung 10
2.6 Fernsteuerung des Simulators 10
2.7 Umfahr-o-mat mit den Zuständen Z_0 Start, Z_1 Facing, Z_2 Adjusted, Z_3 Advanced, Z_4 Stop und Z_5 Corner 12
2.8 Imaginärer Korridor neben Objekten 14

3.1 Traditionelle Zerlegung einer Kontrollstruktur mobiler Roboter (Quelle: [3]) 18
3.2 Zerlegung einer Kontrollstruktur mobiler Roboter in konkurrierende horizontale Verhaltensschichten (Quelle: [3]) 19
3.3 Verhaltensschichten (Layer of Competence) (Quelle: [3]) 21

4.1 Koordinatensystem des Pioneer I Roboters 24
4.2 Zu umfahrende Ecke 30
4.3 Abstoßung durch Feelforce von der Wand 32
4.4 Aufsicht des Feelforce-Verhaltens 33
4.5 Anziehung durch Attract to Wall zur Wand 34
4.6 Regelungssystem zur Wand 34
4.7 Verhaltensweisen des Roboters 36
4.8 Testlauf 1 37
4.9 Testlauf 2 38

Zusammenfassung

Ziel dieser Baccalaureatsarbeit war die Entwicklung eines Follow-Right-Wall Algorithmus, der es den mobilen Roboter ermöglicht, eine ihm unbekannte Umgebung entlang einer Wand zu seiner rechten abzufahren.

Schwerpunkt der Arbeit war die Implementation der Kontrollstruktur, die sich konzeptionell von den herkömmlichen sequentiellen Strukturen unterscheidet und einer verhaltensorientierten Kontrollstruktur entspricht, die in ihrer Erweiterungsmöglichkeit sehr flexibel ist.

Das Gesamtverhalten des Systems ist in einzelne Verhaltensschichten zerlegt, die jeweils ein eigenes Ziel verfolgen. Die einzelnen Verhaltensschichten konkurrieren untereinander und tragen unabhängig voneinander Befehle zur Bewegungssteuerung bei.

Kapitel 1

Einleitung

Das Studium autonomer Mobiler Roboter erfreut sich immer größerer Beliebtheit, insbesondere, da ihre praktische Anwendung in greifbare Nähe rückt. In den meisten Aufgabengebieten befindet sich ein mobile Roboter nicht in einer statischen Umgebung, sondern muss sich meist in einer Umgebung mit einer Vielzahl von dynamischen Objekten zurechtfinden. Die Umgebung kann z.B. bekannten Ebenen oder Gebäudestrukturen entsprechen, wie es etwa bei Büros, Werkhallen und Laboren der Fall wäre, von denen der Roboter eine interne Repräsentation seiner Welt hat. Es kann aber auch zu Einsätzen mobiler Roboter in unbekannten, schwer erreichbaren Umgebungen kommen, wie z.B. in Katastrophengebieten, dem Meeresgrund oder Planetenoberflächen die erst zu explorieren sind.

Eine wichtige Eigenschaft autonomer Roboter ist die Robustheit gegenüber der dynamischen Veränderung der Umgebung. Klassische Architekturen von Kontrollstrukturen tun sich dabei oft schwer und Abhilfe hat man in neuen Architekturen gesucht. Neue Architekturmodelle mit Verhaltensschichten erbrachten die nötige Abhilfe durch Ihre gegenseitige permanente Beeinflussung der Verhaltensschichten. Sie zeigen eine Art von Reflexverhalten wie es bei den Lebewesen auch funktioniert.

Um mit dieser Art von Architektur vertraut zu werden, wurde in dieser Arbeit ein Follow-Right-Wall Algorithmus entwickelt, basierend auf den Ansatz von Rodney A. Brooks vom MIT, der sich 1985 mit der Entwicklung einer solchen Architektur beschäftigt hat [3].

Kapitel 2

Projektbeschreibung

2.1 Allgemeine Zielsetzung des Projektes

Im Laufe des Projektes Navigationsunterstützung für Mensch und Roboter haben wir uns mit der Untersuchung geeigneter Navigations- und Explorationsverfahren für mobile Roboter beschäftigt. Die Verfahren der Navigation und Exploration sollten dabei den Umgang mit Karten erlauben, die auch für den Menschen verständlich sind. Als ein geeignetes Medium beziehen wir uns auf Schematische Karten, die sich im Gegensatz zu metrisch präzisen Karten durch ihre vereinfachenden, abstrahierenden Darstellung von relevanten Informationen und Merkmale zur Navigation besonders gut eignen. Ein wünschenswertes Ziel ist, dass Menschen und Roboter mittels schematischer Karten kommunizieren können, die Karten sollen also die Grundlage für die Kommunikation zwischen Menschen und Roboter schaffen. In diesem Projekt sollten daher verschiedene Fragestellungen untersucht werden:

- Wie lassen sich die sensoriell erfassten Umgebungsinformationen zu einer Karte kombinieren?
- Wie lässt sich erworbene Information repräsentieren?
- Wie kann ein Roboter mit ungenauer Information navigieren?
- Welche Raumrepräsentationen sind sowohl für Navigation als auch Kommunikation geeignet?

2.2 Arbeitsmittel

Im Laufe des Projektes wurde zur Verhaltensmodellierung eines mobilen Roboters ein Simulator genutzt, der Teil des uns zur Verfügung stehenden Fra-

meworks *mapsNrobots* ist, das in LISP programmiert ist. Natürlich wurden die entwickelten Algorithmen auch an dem mobilen Roboter direkt ausprobiert. Bei dem Roboter handelt es sich um einen Pioneer I Roboter mit Laser-Range-Finder als Sensor (Siehe Abbildung 2.1). Der Pioneer I Roboter verfügt desweiteren über Ultraschall-Sensoren.

Abbildung 2.1: Pioneer I Roboter mit Laser-Range-Finder

2.2.1 Sensorik des Roboters

Sensoren liefern dem Roboter Informationen über seine Umwelt (externe Sensoren). Da der Roboter kein Vorwissen z.B. in Form einer Karte über seine Einsatzumgebung erhält, sind sie die einzigen Informationsquellen zur Exploration seiner Umwelt. Der Roboter benötigt seine Sensoren im wesentlichen zur Erkennung von Hindernissen und Bestimmung deren Lage.

Sensoren zur Erkennung und Lagebestimmung von Hindernissen sind notwendig, damit der Roboter während der Fahrt eine Karte erstellen könnte und rechtzeitig auf Hindernisse reagieren kann. Während noch vor einigen Jahren die Ultraschallsensoren die dominierenden Entfernungssensoren für mobile Roboter waren, wird heutzutage jedoch in den meisten Laboren auf die Laser-Range-Finder zurückgegriffen. Laser-Range-Finder überlegen den um ein vielfaches billigeren Ultraschallsensoren weit in der Genauigkeit, Reichweite, Auflösung und

Abfragegeschwindigkeit. Dies ist auch der Grund, warum in unserem Labor auf die Nutzung der Ultraschallsensoren verzichtet wurde. Bei einem Laser-Range-Finder wird ein Laserstrahl ausgesendet, der von Hindernissen reflektiert wird, während der Ultraschallsensor mit Schallwellen und deren Reflektion arbeitet. Wenn nun eine Reflektion eines Laserstrahls wieder auf den Sensor trifft, wird anhand seiner Laufzeit und seiner Ausbreitungsgeschwindigkeit die Entfernung des Hindernisses in der entsprechenden Richtung berechnet. Der von uns verwendete Indoor Laser-Range-Finder (Abbildung 2.2) ist vom Typ LMS-200 der Firma SICK. Er hat einen Scanbereich von 180^0 und sendet die Laserstrahlen alle o,5^0 aus. Somit erhalten wir eine hohe Entfernungsauflösung bis zu einer Entfernung von 8 Metern mit 361 Entfernungswerten. Die Messungenauigkeit beträgt bei diesem Gerät ca. 2 Zentimeter. Der Scan, den wir von dem Scanner erhalten sind Listen mit Entfernungsvektoren und mit den Entfernungspunkten, die in eine bildliche Repräsentation der Umgebung zusammengefasst werden können (siehe Abbildung 2.3).

Abbildung 2.2: Laser-Range-Finder: LMS 200 der Firma SICK

2.2.2 Framework *mapsNrobots*

Das Framework *mapsNrobots* definiert mehrere Module, die die Schnittstelle zum Roboter, eine Fernsteuerung, eine graphische Benutzerschnittstelle, vorimplementiertes Bewegungsverhalten, Datenstrukturen und Algorithmen für geometrische Berechnungen, diverse Utilities sowie einen Echtzeitsimulator bereitstellen. Der Simulator empfängt und sendet mit den gleichen Datentypen

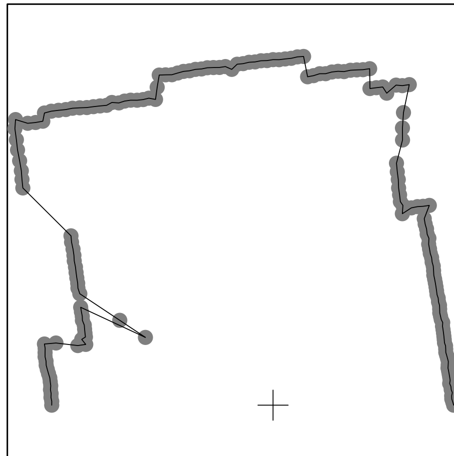

Abbildung 2.3: Umgebungsrepräsentierender Scan des Labors

wie der reale Roboter, so dass ein Steuerprogramm erst mit dem Simulator entwickelt sowie getestet und dann unverändert an der echten Maschine genutzt werden kann. Der Simulator verhält sich analog zum realen Roboter und führt die Bewegungen kontinuierlich aus. Sowohl der simulierte Laserscan der modellierten Umgebung als auch die Bewegungen werden verrauscht, da der reale Laserscan einem Messfehler unterliegt und die Bewegungen des Roboters ebenfalls nie exakt sind.

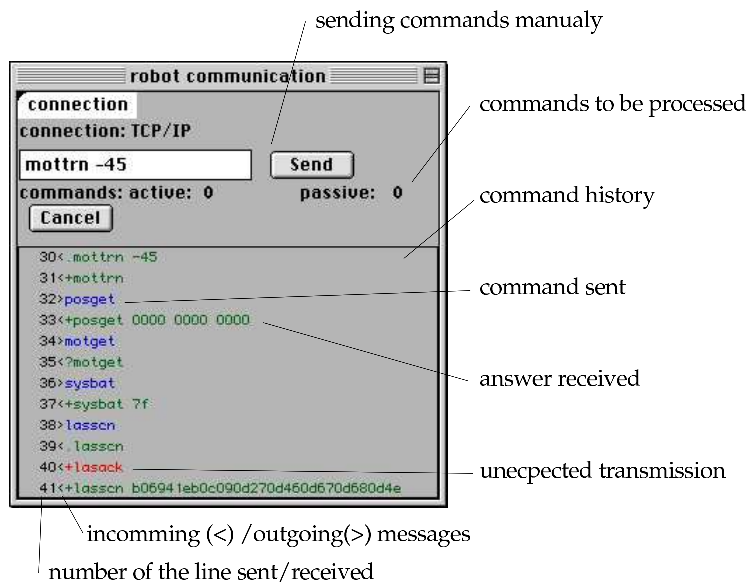

Abbildung 2.4: Kommunikator

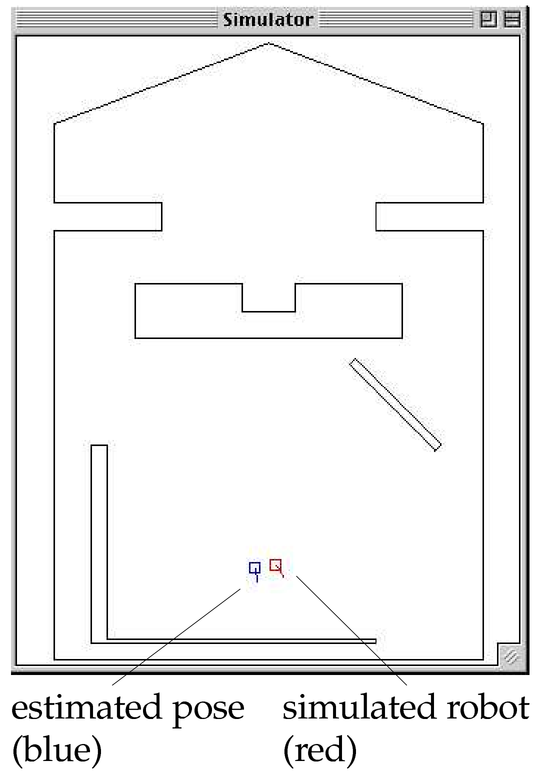

Abbildung 2.5: Simulationsumgebung

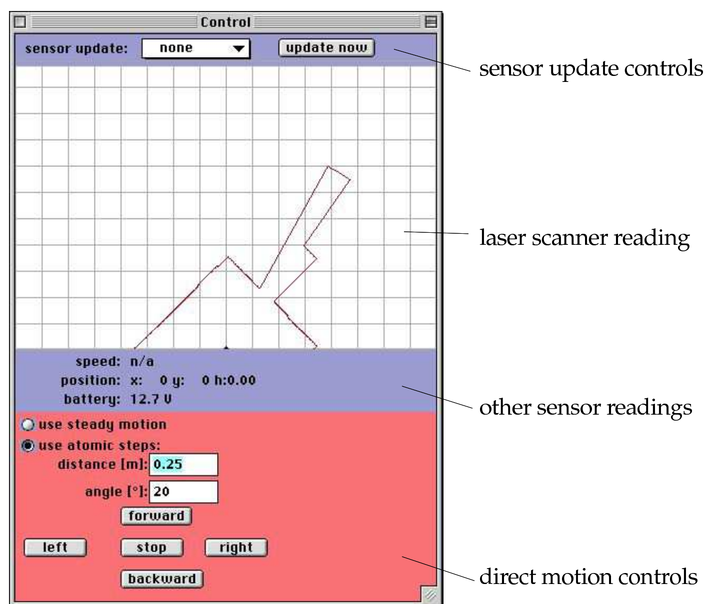

Abbildung 2.6: Fernsteuerung des Simulators

2.3 Projektgruppen

Um sich genauer mit den drei grundlegenden Kerngebieten in diesem Projekt in Bezug auf die vorangegangenen Fragestellungen zu beschäftigen, und den Fokus und die Konzentration aller Mitglieder auf die Kerngebiete zu verteilen, wurde die Projektgruppe in drei Gruppen unterteilt. In diesen konnten wir dann unsere Lösungsansätze für die genannten Problemstellungen erarbeiten und versuchen, die Aufgabe, ein Objekt von einem Roboter finden zu lassen, zu realisieren. Die Gruppen beschäftigen sich eingehend mit Ihrem Kerngebiet und stellten über Schnittstellen die entwickelten Funktionen für die anderen Gruppen zur Verfügung. Die Gruppen sind im folgenden aufgelistet. Hierbei habe ich mich der ersten Gruppe, der Exploration, verpflichtet. Diese Baccalaureatsarbeit ist somit in diesem Rahmen entstanden.

2.3.1 Exploration

Diese Gruppe hat als Hauptaugenmerk die direkte Ansteuerung des Roboters umzusetzen. Der mobile Roboter sollte in der Lage sein, eine unbekannte Umgebung erschöpfend zu explorieren. Hierbei mussten Teilprobleme gelöst werden: Der Roboter sollte sich in seinem Sichtfeld auf jeden Fall kollisionsfrei bewegen können. Er sollte Teilansichten von Umgebungen erstellen können, die aus der derzeitigen Position nicht einsehbar sind, durch Entwicklung eines Verfahrens, das den Roboter zu einem geeigneten „Aussichtspunkt“ fahren lässt, um diese Teilansicht wahrzunehmen. Hauptgegenstand war jedoch die Entwicklung eines Verfahrens zum Absuchen bzw. Abfahren von Hindernissen. Es wurde angenommen, das das gesuchte Objekt groß genug ist, um im Laser-Scan sichtbar zu sein. Ein sorgfältiges Umfahren aller Hindernisse sollte somit zum Erfolg führen.

Realisierung: Das zu entdeckende Objekt soll durch einen Infrarotsensor, der auf dem mobilen Roboter nach rechts gerichtet (und der Sender natürlich am Objekt), befestigt sein sollte. Durch die Entwicklung eines Follow-Right-Wall Algorithmus, der die Objekte rechterhand umfährt, müsste man also nach dem Abfahren aller Objekte das gesuchte finden müssen. Der Infrarotsensor reagiert auf den Sender wenn er sich in einem Umkreis von 1 Meter zum Sender befindet.

Wir wollten nun also Versuchen ein LISP-Programm zu Entwickeln und einen Automaten-Interpreter, der in anderen Formen schon vorhanden war, zu nutzen. Unser Programm sollte einem Mealy-Automaten entsprechen. Der

Interpreter konnte einen Automaten A, ein 4 Tupel, der folgenden Form verarbeiten:

$$A = (S, \Sigma, \Gamma, E)$$

wobei:

S	Menge der Startzustände $\subset \Sigma$
Σ	Menge der Zustände
Γ:$\Sigma \rightarrow \Sigma$	ist die Überführungsfunktion
E	Menge der Endzustände

Ein Zustand wurde in den nächsten überführt, wenn die Prämisse erfüllt ist und die Konklusion, die jeweils Funktionen sind, ausgeführt wurde. Unser Programm für den Follow-Right-Wall Algorithmus repräsentierte den Automaten aus Abbildung 2.7. In der Abbildung sind Prämisse und Konklusion an den Zustandsübergängen durch $\frac{Praemisse}{Konklusion}$ dargestellt.

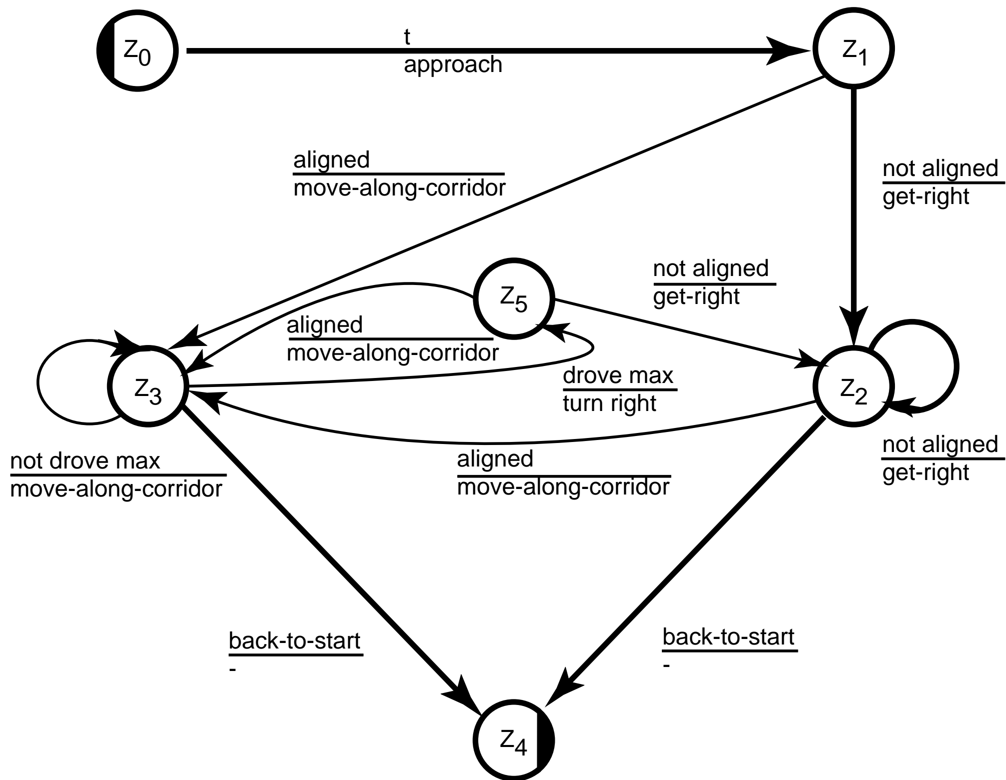

Abbildung 2.7: Umfahr-o-mat mit den Zuständen Z_0 Start, Z_1 Facing, Z_2 Adjusted, Z_3 Advanced, Z_4 Stop und Z_5 Corner

Erläuterung zu unserem Automaten Zuerst möchte ich ihnen die einzelnen Zustände erläutern:

Z_0 **Start:** Dieser Zustand ist der Startzustand des mobilen Roboters. Er befindet sich an einem beliebigen Ort im Raum auf dem Boden. Der Übergang vom Zustand Z_0 zu Z_1 hat als Prämisse immer TRUE und führt somit auf jeden Fall die Konklusion, also den Aufruf der Funktion `Approach` aus. Diese wiederum ruft einfach nur zwei Funktionen `Turn-to-closest-Objekt` und `move-as-far-as-can` auf, die den mobilen Roboter zu dem nächsten Objekt (mit dem geringsten Entfernungswert) in seinem Scan fährt.

Z_1 **Facing:** Wenn der Automat in diesem Zustand ist, befindet er sich vor dem Objekt. Nun ist die Frage, ob der mobile Roboter vor dem Objekt schon richtig Ausgerichtet, also parallel zu dem Objekt, ist. Die Abfrage, ob der Roboter `aligned` ist, also parallel zum Objekt steht, überführt bei FALSE zu Z_2 und bei TRUE zu Z_3. Wenn er nicht richtig zum Objekt ausgerichtet ist, so versucht er sich mit `get-right` so zu drehen, das sich das Objekt an seiner rechten Seite befindet.

Z_2 **Adjusted:** Der Roboter ist aus Z_1 zu diesem Zustand Z_2 gelangt. Der Roboter musste sich für den Zustandsübergang durch aufrufen von der Funktion `get-right` einmal zur Wand ausrichten. Da es noch nicht der Fall sein muss, das der mobile Roboter wirklich ganz parallel zur Wand steht, kann er bei Erfüllung der Prämisse `not aligned` wieder reflexiv von Z_2 zu Z_2 gelangen. Wenn er jedoch richtig ausgerichtet ist, erfolgt der Zustandsübergang zu Z_3 mit der Funktion `move-along-korridor`, bei der sich der Roboter in einem Korridor neben dem Objekt entlang bewegt (siehe Abbildung 2.8). Dieser imaginäre Korridor befindet sich in einer Entfernung von 40 cm bis 1 m zur Wand.

Z_3 **Advanced:** In diesem Zustand ist der Roboter den Korridor nun schon etwas abgefahren, soll nun mit `not drove max` prüfen, ob er noch weiter fahren kann, oder ob sich ein Objekt in seinem Korridor befindet, was ihn hindert, weiter zu fahren. Wenn `drove max` erfüllt ist, geschieht ein Zustandsübergang zum Zustand Z_5 Corner.

Z_4 **Stop:** In diesen Zustand gelang der mobile Roboter, wenn er ein Objekt umfahren hätte und sich wieder an der Ausgangsposition befinden würde. Hier war eine Schnittstelle mit der *Kartierung* angedacht. Sie sollte anhand der Scans ermitteln, ob sich der Roboter an einem Ort befindet, an dem wir schon einmal waren.

Z_5 **Corner:** Der mobile Roboter ist die Wand, bzw. das Objekt maximal entlang gefahren und soll nun um die Ecke fahren, wenn er vom Zustand

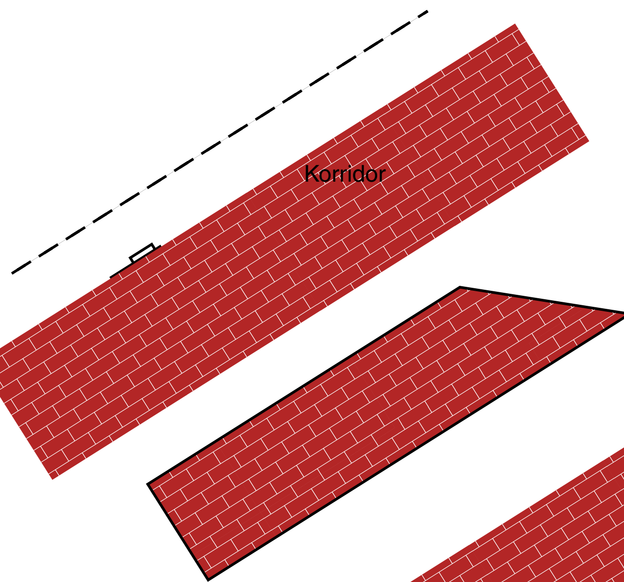

Abbildung 2.8: Imaginärer Korridor neben Objekten

Z_3 in diesen Zustand gelangt. Anhand der Funktion `turn-right` soll der Roboter um die Ecke fahren. Danach wird durch die `aligned`-Abfrage kontrolliert, ob er wieder das nächste Objekt ganz rechts hat und sich entweder neu ausrichten soll, oder den Korridor entlang fahren soll.

Probleme: Um im weiteren einen Navigations-Algorithmus zur Exploration eines Gesamten Raumes zu entwickeln wäre eine Verschmelzung des Start und Endzustandes hilfreich gewesen. Es wäre desweiteren noch eine Planungsstrategie nötig gewesen, die das nächste anzufahrende Objekt auswählt. Dann hätte noch die Routine zum anfahren eines Objektes so abgeändert werden müssen, das er sich als nächstes Objekt das aus der Planungsstrategie nimmt.

Probleme bei dem bisherigen Ansatz gab es vor allem bei dem umfahren einer Ecke, da der mobile Roboter die Ecke aus seinem Scanbereich verloren hat und somit wahllos einen Punkt voraus als neuen Fixpunkt genommen hat. Außerdem waren langsam die Aufgabenbereiche der einzelnen Funktionen ziemlich komplex und undurchsichtig geworden. Das hat mich dazu veranlasst, mich nach anderen Ansätzen umzuschauen. Dadurch kam ich auf den Ansatz von Rodney A. Brooks. Durch die modulare Layer-Architektur würde man die einzelnen Funktionen Aufgabenspezifischer und unabhängig voneinander erstellen und erweitern können. Somit fand sich mein Thema, über das diese Baccalaureatsarbeit berichtet: *Entwicklung eines Follow-Right-Wall Algorithmus basierend auf Verhaltensschichten.*

2.3.2 Kartierung

Aufgabe dieses Bereichs sollte das Zusammensetzen verschiedener lokaler Sensorbilder (Einzelbilder) zu einer konsistenten räumlichen Repräsentation (Karte) sein. Aus der Repräsentation müssen sich geometrische Merkmale, die für die Präsentation wichtig sind (wie z.B. Linien für Objektumrisse), entnehmen lassen. Die zu erstellende Repräsentation soll die Selbstlokalisation eines Roboters unterstützen, was bedeutet, dass sich aus einem gegebenen Sensorbild und der erstellten Karte eine Schätzung für den wirklichen Aufenthaltsort des Roboters bestimmt lassen soll. Wesentlich ist eine Bestimmung von miteinander korrespondierenden Teilansichten in Sensorbildern, das bedeutet, dass sich bestimmte Merkmale eines Sensorbildes in einem anderen wiederfinden lassen. Es ist zunächst zu entscheiden, welche Merkmale eines Sensorbildes dabei betrachtet werden sollen, wobei Merkmale wie Strecken, polygonale Linien, oder Eckpunkte sich hervorragend anbieten. Eine Extraktion dieser Merkmale aus den Daten des Laser-Scans ist somit zu implementieren. Auf Basis der extrahierten Merkmale ist dann eine Methode zu entwickeln, die bestimmen kann, welche Teilansichten zueinander passen. Hierbei handelt es sich um eine Suche, die zueinander passende Merkmale aus zwei Ansichten, wie z.B. zwei Sensorbildern oder ein Sensorbild und eine bereits erstellte Karte, herraussucht. Dabei sollen die korrespondierenden Merkmale gefunden werden, die in der Nähe der vom Roboter geschätzten Position liegen.

2.3.3 Präsentation

Aufgabe war es, aus einer Übersichtskarte (ein Aufriss) einer Büroumgebung und zwei Markierungen in dieser Übersichtskarte, die den Start und das Ziel darstellen, eine schematisierte, vereinfachte und zweckgebundene Instruktionskarte zu erstellen. Anhand dieser Instruktionskarte soll einem Menschen der Weg vom Start zum Ziel gewiesen werden. Hierfür musste erstmal der legitime Informationsgehalt einer Karte bestimmt werden. Außerdem wird eine Bestimmung des in der Instruktionskarte zu repräsentierenden Informationsgehaltes benötigt, und unter welchen Bedingungen, genauer, in welcher räumlichen Beziehung zu einem Pfad von Start zum Ziel die Objekte der Übersichtskarte in der Instruktionskarte zu repräsentieren sind. Dadurch soll für die Navigation relevante Information ausgewählt und so dargestellt werden, dass ein Weg vom Start zum Ziel entnehmbar ist.

Kapitel 3

Verhaltensschichten gemäß des Ansatzes von R. A. Brooks

3.1 Anforderungen an die Kontrollstruktur

Es gibt eine Vielzahl von Ansätzen für die Struktur eines Kontrollsystems eines komplett autonomen Roboters. Zuerst sollten die Anforderungen identifiziert werden, denen sie gerecht werden müssen. Die Informationsverarbeitung muss in Echtzeit erfolgen. Die Randbedingungen des Systems in dem sich der Roboter bewegt ändern sich typischerweise ständig; außerdem erfolgt die Bestimmung dieser Randbedingungen über sehr stark rauschende Kanäle da, es keine einfache Beziehung (Abbildungsvorschrift) zwischen den Sensordaten und der Form, wie sie für die Bestimmung der Randbedingungen benötigt werden, gibt. Anforderungen an das Kontrollsystem sind somit:

Konkurrierende Aufgabenziele Der Roboter verfolgt zu einem Zeitpunkt oft mehrere Ziele, die untereinander unter Umständen auch im Konflikt stehen. Beispielsweise soll er einen bestimmten Punkt erreichen, aber auch lokalen Hindernissen ausweichen. Dabei ist die relative Wichtigkeit der einzelnen Ziele kontextabhängig. Das Kontrollsystem muss sowohl Ziele mit höherer Priorität erreichen, andererseits auch die notwendigen Basisaufgaben erledigen.

Mehrere Sensoren Der Roboter hat wahrscheinlich mehrere Sensoren, durch die er seine Umwelt wahrnimmt. Jeder dieser Sensoren hat eine Fehlertoleranz, bzw. Abweichung. Außerdem gibt es meist keine einfache Beziehung zwischen den Sensordaten und den gewünschten physikalischen Größen. Einige der Sensoren werden sich bezüglich ihrem Aufnahmegebiet überlappen und es kommt unter Umständen zu inkonsistenten Messwerten, sei

es einfach durch Sensorfehler oder auch weil ein Sensor außerhalb des Bereiches misst, für den er eigentlich zuständig ist und dies nicht festgestellt werden kann.

Robustheit Eine weitere Anforderung an den Roboter ist seine Robustheit, einige Aspekte sind: Ein Sensorausfall muss erkannt werden und mit den noch funktionierenden weitergearbeitet werden. Eine drastische Umgebungsänderung darf sich nicht dadurch auswirken, dass der Roboter verwirrt stehen bleibt, sondern indem er sinnvoll auf die neue Situation reagiert. Weiter wäre es gut, dass der Roboter auch dann noch arbeitet wenn Teile der Kontrollstrukturen ausfallen.

Erweiterbarkeit Wenn der Roboter mehrere Sensoren oder Fähigkeiten, bekommen sollte, wird er mehr Rechenleistung benötigen. Die Erweiterung seiner Fähigkeiten würde sich in einer Erweiterung der Kontrollstruktur durch weitere Module kennzeichnen.

Die Architektur zur Kontrolle von mobilen Robotern durch Kontrollschichten wurde entworfen um Roboter in einem ständig wachsendem Kompetenzfeld einzusetzen. Verhaltensschichten bestehen aus Modulen die, jedes für sich, eine kleine Berechnungsfunktion verkörpern. Die Module können miteinander kommunizieren und ergeben alle zusammen ein flexibles Kontrollsystem für mobile Roboter.

3.2 Traditionelle Architekturen

Es gibt viele Ansätze einen autonomen mobilen Roboter zu bauen. Meist wird am Anfang das Problem in Teilprobleme zerlegt, diese Teilprobleme werden dann gelöst, und letztendlich werden die Lösungen wieder zusammengefügt.

Typischerweise haben die Entwickler mobiler Roboter das Problem unterteilt in:

- Sensorik
- Überführen der Sensordaten in eine Representation
- Aktionsplanung
- lösen der Aufgaben
- Motorsteuerung
- Roboter-Hardware

Diese Zerlegung kann man als eine horizontale Zerlegung in vertikale Komponenten verstehen. Zu sehen ist das in der Abbildung 3.1. Durch die verketteten Komponenten fließt die Information der Umgebung des Roboters, via Sensorik, durch den Roboter und zurück zur Umgebung, via Aktuatoren. Eine Instanz jeder Komponente muss in Reihenfolge erstellt werden, es muss also jede Komponente vorhanden sein, damit der Roboter korrekt funktioniert. Spätere Änderungen an einem Abschnitt, etwa durch Erweiterung oder Änderung der Funktionalität, muss entweder so gemacht werden, das sich das Interface zu den anderen Abschnitten nicht ändert, oder sich die Effekte der Änderungen in benachbarten Komponenten fortpflanzen. Es müssen dann also auch die Funktionalitäten der nächsten Komponenten geändert und korrigiert werden.

Diese Zerlegung des Problems ist wirklich die klassische Zerlegung, die man sogar im Informatik Duden [5] finden kann. Hier ist die Aufteilung in *Eingabe* $\Rightarrow$ *Verarbeitung* $\Rightarrow$ *Ausgabe* gegliedert. An diesem Punkt verdeutlicht sich die Trennung der traditionellen Informatik zur KI, durch das nicht nutzen des traditionellen Ansatzes.

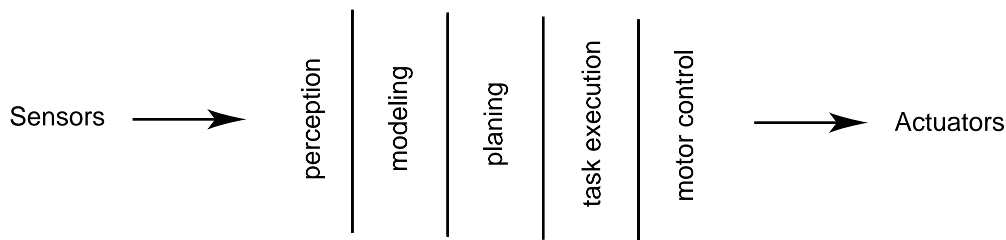

Abbildung 3.1: Traditionelle Zerlegung einer Kontrollstruktur mobiler Roboter (Quelle: [3])

3.3 Subsumption-Architektur & Kompetenzebenen

Brooks hat sich in „A Robust Layered Control System for a Mobile Robot“ [3] für die Zerlegung des Problems in horizontale Scheiben, d.h. Schichten, die vertikal angeordnet sind, entschieden (siehe Abbildung 3.2).

Bei dem *Subsumption*-Ansatz von Brooks wird das Gesamtverhalten des Systems in einzelne Verhaltensschichten zerlegt. Jede dieser Verhaltensschichten verfolgt ein eigenes Ziel und hat prinzipiell Zugriff auf alle Sensordaten. Subsumption heißt wörtlich übersetzt Ein-, bzw. Unterordnung, oder Zusammenfassung.

Die verschiedenen Verhaltensschichten werden laufend parallel berechnet, und aus dieser „Verhaltens-Menge“ wird anschließend das anzuwendende

Abbildung 3.2: Zerlegung einer Kontrollstruktur mobiler Roboter in konkurrierende horizontale Verhaltensschichten (Quelle: [3])

gewählt. Das erste Verhaltensmuster könnte beispielsweise festlegen, dass der Roboter in der Gegend herumfährt und dabei Hindernissen ausweicht. Ein anderes Verhalten könnte sein, den Roboter zu einer Steckdose zu bewegen, wenn seine Energie zur Neige geht. Hier ist es recht leicht einzusehen, wann welches Verhalten gezeigt werden soll: wenn die Batterien voll sind, werden die Steuerungsbefehle, die das erste Modul ausgibt, ausgeführt, das zweite ist nicht aktiv. Wird dieses aktiv, unterdrückt es die Befehle, die von der ersten Schicht ausgegeben werden. Verhalten höherer Schichten sind also in der Lage, Verhalten niedriger Schichten zu unterdrücken. Daher auch der Name Subsumption-Architecture. Die einzelnen Verhaltensschichten konkurrieren untereinander, so dass zu jeder Zeit also von allen Verhaltensweisen ein Vorschlag zur Steuerung der Aktuatoren vorliegen kann (siehe Abbildung 3.2). Einzelne Verhaltensweisen werden bei Brooks als endliche Automaten implementiert.

Bei dem Ausfall von höheren Schichten versagt nicht das gesamte System, sondern die unteren Schichten können in der Regel weiterhin arbeiten. Dadurch wird die Sicherheit des Systems erhöht, da zum Beispiel trotz Ausfalls eines anwendungsspezifischen Verhaltens, ein Verhalten zur Hindernisvermeidung noch arbeiten kann und so Kollisionen weiterhin vermieden werden.

Ein wesentlicher Unterschied zu anderen Ansätzen ist, dass der Roboter keine Repräsentation seiner Umwelt erstellt (es wird kein Weltenmodell entworfen, die einzige Information über die Umwelt zur Verfügung steht sind die aktuellen Sensordaten), sondern auf seine Sensordaten direkt reagiert, weshalb ein Vergleich mit Reflexen durchaus angebracht ist („Die Welt selbst ist das beste Modell“[3]).

Anstatt wie oben schon erwähnt das Problem entsprechenden der internen

Berechnungsmethode zu unterteilen (die Information fließt von einem Modul ins nächste), geschieht das hinsichtlich der gewünschten verschiedenen Verhaltensweisen. Das wird verwirklicht indem man Kompetenzebenen einführt, wobei in jeder dieser Ebenen immer eine Klasse von Verhaltensmustern definiert wird. Je höher die Kompetenzebene, desto spezifischer ist das in ihr beschriebene Verhalten. Dieses Konzept wird klarer, wenn man ein solches Kompetenz-System betrachtet:

1. Verhindern von Zusammenstößen mit sich bewegenden und stationären Objekten
2. Ziellos herumfahren unter Vermeidung von Zusammenstößen
3. Erforschen der Umwelt, indem man entfernte Orte findet die erreichbar erscheinen und diese Richtung einschlägt
4. Erstellen einer „Landkarte“ der Umgebung und Planen von Routen um von einem Ort zu einem anderen zu gelangen
5. Erkennen von Änderungen der statischen Umwelt
6. Über die Welt „nachdenken“, bezugnehmend auf identifizierbare Objekte und Aufgaben bezüglich dieser Objekte
7. Pläne, die die Umwelt in einer sinnvollen Weise verändern formulieren und ausführen
8. Über das Verhalten von Objekten in der Umwelt „nachdenken“ und Pläne entsprechend ändern

Jedes Verhalten, das von einer Kompetenzebene beschrieben wird, ist eine Untermenge derer der darunter liegenden, das heißt dass das Verhalten mit zunehmender Höhe der Ebene immer beschränkter wird.

3.4 Kontrollschichten

Man erstellt zu jeder der in vorigen Punkt beschriebenen Kompetenzebenen eine korrespondierende Kontrollschicht. Will man dem System eine neue Kompetenz hinzufügen, entwirft man einfach eine neue Kontrollschicht und fügt diese ein.

Man beginnt damit, das Kontrollsystem, das die Kompetenzebene 0 implementiert, zu bauen und nennen es entsprechend die Kontrollschicht 0. Dieses System wird nie mehr verändert. Wenn dieses System funktioniert, kann man die nächste Schicht hinzufügen, die für die 1. Kompetenzebene „zuständig“ ist.

Sie kann Daten aus der 0. Schicht verwenden, Daten in interne Schnittstellen der 0. Schicht eingeben, wobei der normale Datenstrom unterdrückt wird. Die 0. Schicht arbeitet weiter, ohne etwas von der oberen Schicht zu wissen; der einzige Einfluss ist die eventuelle Dateneingabe der 1. Schicht in interne Schnittstellen. Bei den weiteren Schichten wird gleich vorgegangen. Deshalb wird diese Architektur „Subsumption Architecture“ genannt.

Graphisch kann man sich diesen Aufbau wie in Abbildung 3.3vorstellen.

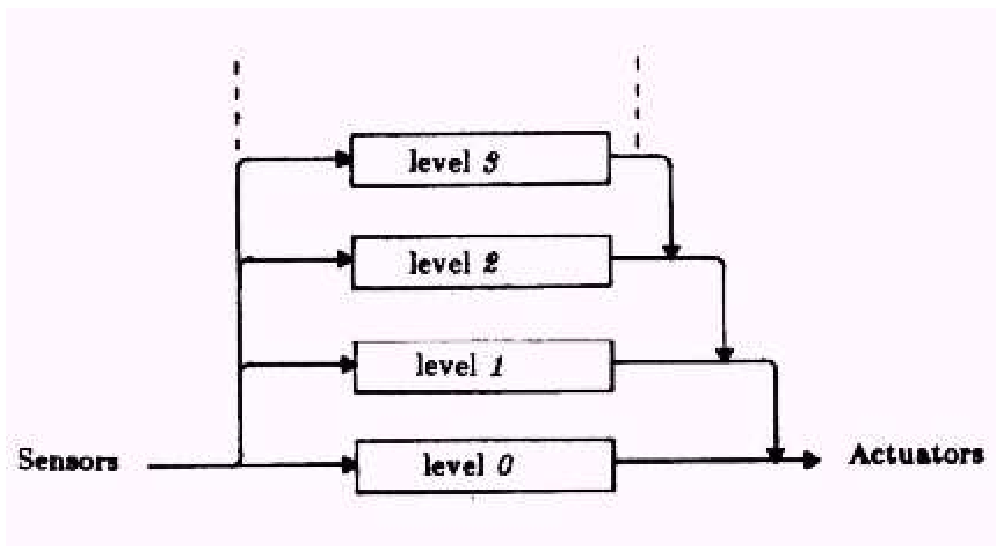

Abbildung 3.3: Verhaltensschichten (Layer of Competence) (Quelle: [3])

Kapitel 4

Entwicklung des F-R-W Algorithmus

4.1 Ziel

Ziel ist es bei meinem Algorithmus, dass der mobile Roboter, ausgesetzt in einer ihm unbekannten Umgebung, den Weg zu einem Objekt respektive Wand findet und diese dann rechter Hand abfährt. Er soll sich nicht durch dynamische Objekte verwirren lassen und zum Stillstand kommen, sondern einen Ausweg aus dieser neuen Situation finden.

Realisieren wollte ich das ganze basierend auf dem in den vorhergehenden Kapitel 3 erwähnten Architekturmodell der Verhaltensschichten.

4.2 Rahmenwerk

Zur Umsetzung meiner Arbeit in eine LISP Programm habe ich ein Rahmenwerk erstellen können, das mir die Programmierung der Robot motion behavior nach Brooks Ansatz ermöglichte. In einem Rahmenwerk `behavior-modules` werden neue Klassen zur Verfügung gestellt. Eine dieser Klassen repräsentiert die Module. Zur Erstellung von Modulen wurde ein Macro erstellt, da wir eine eigene Syntax für die Module nutzen möchten, das verschiedene Objekte und Funktionsdefinitionen automatisch erzeugt. Das Macro, mit dem ein Modul erstellt werden kann, lautet `defmodule`. Die Syntax der Module sieht in der EBNF (extended Backus-Naur-Form) folgendermaßen aus:

(`defmodule` *Modulname* `((:Inputs` $\{variable\}^*$`)(:Outputs`
$\{variable\}^*$`)(:Variables` $\{variable\}^*$`)))`

Mit `defmodule` wird eine Objekt der Klasse Modul angelegt und in gleichnamiger Variable gespeichert. In diesem Objekt werden Name, Inputs, Outputs und zwei Funktionen, eine zur Initialisierung und eine zur Ausführung des Moduls gespeichert.

Ein Modul erhält einen beschreibenden Namen, muss eine Menge von Eingabewerte angeben, sowie eine Menge von Ausgabewerte. Diese Mengen können jedoch auch leer sein. Desweiteren müssen in dem Modul genutzte Variablen angegeben werden, wenn sie nicht als Eingabe- bzw. Ausgabewert fungieren. Mit der LISP-Funktion `send-output`, die im Modul-Körper definiert ist, senden wir den errechneten Ergebnisvektor dieses Moduls an ein außenstehendes Steuerprogramm, das die Motorsteuerung des mobilen Roboters übernimmt und die Ergebnisvektoren aller Module miteinander verrechnet. Diese Ausgabe durch `send-output` unterscheidet sich von den zuvor erwähnten `Ausgabewerten`, die anderen Funktionen zur Verfügung gestellt werden. Deshalb nutze ich im Folgenden den Begriff *Ausgabevektor* als die durch `send-output` erzeugte Ausgabe. Ein Modul wird ausgeführt, wenn alle seine Eingabewerte vorliegen. Durch die Ausführung werden die Ausgabewerte und der Ausgabevektor des Moduls berechnet.

Eine weitere Klasse, Namens `Layer`, stellt mit dem Macro `deflayer` eine Funktion zur Verfügung, mit der ein Objekt mit der Syntax:

(`deflayer` *Layername Prioritaet* ({*Module*}*))

erstellt werden kann. Mit `deflayer` wird ein Objekt der Klasse `layer` unter dem Namen *Layername* erzeugt. Bei den Modulen handelt es sich um eine Liste der Module, die sich in diesem Layer befinden. Verbindungen sind Wertübergaben zwischen Modulen. Bei der Ausführung eines Layers werden zuerst die ausführbaren Module gesucht. Das wären Module die keine Eingabewerte bekommen, aber u.U. Ausgabewerte produzieren. Das Sense Modul in 4.3.1 wäre solch ein Modul. Danach werden die Module ausgeführt, deren benötigten Eingabewerte schon zur Verfügung stehen. Nach der Berechnung aller Module wird die Motorsteuerung von dem Modul `move-robot-by-vector` generiert, die den mobilen Roboter atomar um diesen Vektor verschiebt. Dies ist somit eine Diskretisierung der Idee von Brooks.

Dieses Modul erzeugt aus den Bewegungsvektor die Motorbefehle. Der Roboter arbeitet mit dem Koordinatensystem aus Abbildung 4.1. Das Modul `move-robot-by-vektor` übernimmt die exakte Umrechnung des Bewegungsvektors in die `turn` und `move` Befehle des mobilen Roboters unter Einhaltung seiner Koordinaten.

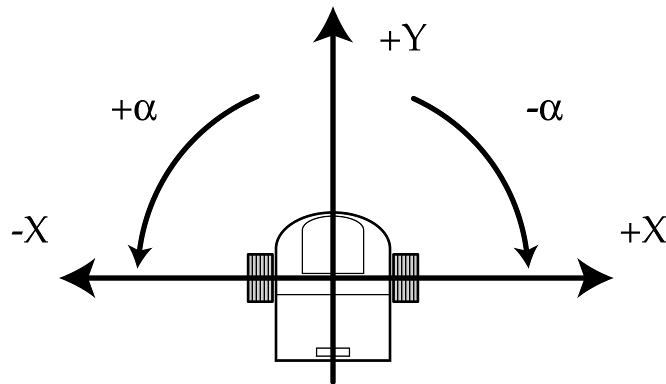

Abbildung 4.1: Koordinatensystem des Pioneer I Roboters

4.3 Module

Im diesem Abschnitt möchte ich nun die erstellten Module meiner Arbeit im Detail vorstellen und die Beziehungen der Module untereinander im darauf folgenden Abschnitt 4.4 beschreiben.

4.3.1 Sense

Das Sense Modul ist für das sensing des Roboters zuständig. Es holt sich einen Scan von dem Laser-Range-Finder (vgl. Abbildung 2.2). Dieser Scan darf nicht älter als 250 ms sein. Dann wird die Liste der Scan-Points, sowie der Scan-Vektoren gespeichert, die als Ausgabewerte dieses Moduls zur Verfügung stehen. Im Code sieht dieses Modul folgendermaßen aus:

```
(defmodule sense ((:outputs scan-points scan-vector))
  "The sense module sets the scan-points and scan-vector"
  (let ((scan (robbi:get-latest-sensor-reading
               robbi::*scan-server* 250)))
    (setq scan-points (robbi:get-local-scan-points scan)
          scan-vector (robbi:get-scan-vector scan))))
```

Da die Listen der Scan Points und des Scan Vektors nun als Ausgabewerte des Sense-Moduls angeboten werden, können die anderen Module diese als Eingabewerte nutzen und Ihre Funktionen darauf anwenden.

4.3.2 Collision Avoidance

Die Basisfunktion dieses Moduls ist die Vermeidung von Kollisionen mit Hindernissen in der dynamischen Umgebung. Die Vermeidung von Zusammenstößen ist

auch erfolgreich, wenn ein Objekt in den Scanbereich des mobilen Roboters läuft oder geschoben wird. Als Eingabewert bedient sich das `Collision-Avoidance` Modul des Scan-Vektors. Er schaut in der Liste der Scan-Vektoren nach, ob sich irgend ein Objekt in einer Entfernung von weniger als 35 cm befindet. Sollte das der Fall sein, so wird die Variable `halt` auf TRUE gesetzt. Das Modul sendet einen in seiner Priorität mit `nil` am höchsten eingestuften und nicht mehr änderbaren Wert `'halt`. Dieses Modul ist das einzige Modul, das als Ausgabevektor gar keinen Vektor schickt, sondern einen String. Das hat den Grund, das dieser Befehl in dem Verrechnungsmodul erkannt wird, und wir somit diesem Befehl eine höchste Priorität zuweisen konnten. Es ist also irrelevant, ob andere Module einen Ausgabevektor senden, sobald `'halt` auftritt, wird keine Bewegungsbefehl ausgeführt. Zur Behandlung des Befehls `'halt` erzeugt das Verrechnungsmodul jedoch einen negativen Y-Vektor, eine Rückwärtsbewegung von 20 cm. Damit der mobile Roboter wieder aus dem Gefahrenbereich heraus manövriert wird.

```
(defmodule collision-avoidance ((:inputs scan-vector)
                                (:outputs halt))
  "Avoidance of an obstacle collision, sends halt as output"
  (setq halt (some (lambda (x)
                     (< x 350))   ;Entfernung x < 350 mm
                   scan-vector))
  (if halt (send-output nil 'halt)))
```

4.3.3 Wander

Das Wander Modul erzeugt als Ausgabevektor einen Zufallsvektor. Beim durchlaufen einer inkrementellen Schleife, eines Zählers von 0 bis 10, wird dieser Vektor erstellt. Wenn dieser Zähler 10 erreicht hat, werden für die X- und Y-Koordinaten des Ausgabevektors jeweils zufällige Werte gewählt, die zwischen -100 und +100 liegen werden. Danach wird der Zähler wieder auf null gesetzt und die nächsten 10 Schleifendurchgänge sendet dieses Modul den erzeugten Zufallsvektor als Ausgabevektor und erst bei erneutem Erreichen der `if`-Bedingung, wird ein neuer Zufallsvektor erzeugt und die Prozedur beginnt wieder von vorne. Die Nutzung eines Zählers hatte einfach nur den einen Hintergrund, dass wir nicht immer wieder einen neuen Zufallsvektor erzeugen wollen, sondern das der mobile Roboter einige Durchgänge hat, in denen er sich in die Richtung des Zufallsvektors orientieren kann.

```
(defmodule wander ((:variables (loop-counter 10)
                                random-vector))
  "Creates a random-vector for moving every 10. Loop"
  (if (= 10 loop-counter)
    (setf random-vector (geo:make-2d-vector
                          (- (random 200) 100) (- (random 200) 100))
          loop-counter 0)
    (incf loop-counter))
  (send-output 1 random-vector))
```

Dieses `Wander` Modul ist Aufgrund der Nutzung des noch folgenden `straight-forward` Moduls jedoch nicht zum Einsatz gekommen. Allerdings hätten wir mit den drei vorgestellten Modulen `Sense`, `Collision-avoidance` und `Wander` ein lauffähiges Kontrollsystem, das den Roboter quer durch den Raum fahren würde und mit keinem Hindernis kollidieren lassen würde.

4.3.4 Feelforce

Dieses Modul erhält als Eingabe die `scan-points`. Das Feelforce Modul addiert die X-Werte der ersten beiden Elemente der Liste miteinander, dann die beiden Y-Werte ebenso. Das Ergebnis wird dann mit dem nächsten Element der Liste rekursiv weiter bis zum Ende der Liste aufaddiert. Daraus wird dann ein *Ergebnisvektor* „result“ erzeugt, der in die entgegengesetzte Richtung des aufaddierten Vektors zeigt. Dies ist nun die „Kraft“, die den mobilen Roboter von einer Wand, bzw von Objekten abstößt. Da die `scan-points` jedoch Vektoren enthält und wir mit mapcar keine Vektoren miteinander addieren können, sondern nur Listen, müssen wir mit `coerce scan-points 'list` die Vektoren in eine Liste umwandeln.

Da diese Kraft, die den mobilen Roboter von der Wand, bzw. dem Objekt, wegdrückt, stärker sein soll, je näher der Roboter der Wand kommt und nicht bei großer Entfernung genau so stark in die Gewichtung fallen sollte, musste eine Funktion entwickelt werden, die diese Kraft skaliert. Die Skalierung des *Ergebnisvektors* ist formal:

$$\vec{F} = (\frac{1}{||Ergebnisvektor||})^4 \cdot 75 \cdot 10^5$$

Bei der Entwicklung dieser Funktion dauerte es einige Zeit, bis die *richtigen* Werte gefunden wurden. Die Ausgabewerte des Feelforce Moduls sind der `Feelforce-Vektor` und der `scaled-Feelforce-Vektor`, wobei der `scaled-Feelforce-Vektor` einfach der Feelforce Vektor gestreckt um den

Faktor $75 \cdot 10^5$, also dem letzten Teil der obigen Formel ist. Die zur Verrechnung für das Motormodul gesendete Ausgabevektor ist jedoch nur der `scaled-Feelforce-Vektor`. Wie genau ich auf den *„richtigen“* Streckungsfaktor gekommen bin, wird in Abschnitt 4.4 anhand eines Versuchsaufbau erläutert.

```
(defmodule feelforce ((:inputs scan-points)
                      (:outputs feelforce-vector scaled-feelforce))
  "This module creates a Vector that reject the robot of the
  walls and obstacles."
  (setf feelforce-vector
    (reduce #'geo:add
      (mapcar
        (lambda (point)
          (let ((result (geo:make-2d-vector
                          (- (geo:x point)) (- (geo:y point)))))
            (geo:multiply
              (expt (/ 1 (geo:vector-length result)) 4) result)))
        (coerce scan-points 'list))))
  (setq scaled-feelforce (* 7500000 feelforce-vector))
  (send-output 1 scaled-feelforce))
```

4.3.5 Attract to Wall

Dieses Modul ist eine Art Abwandlung des Feelforce Moduls und erstellt einen Vektor, der den mobilen Roboter zu dem Objekt hinzieht, anstatt es von den Objekten wegzudrücken. Als Eingabewerte erhält dieses Modul auch wieder die `scan-points` und stellt als Ausgabewert den berechneten, jedoch nicht gestreckten `Attract-to-wall` Vektor zur Verfügung.

Desweiteren wird in die Berechnung nicht der gesamte Scan Bereich mit 361 Entfernungswerten in Betracht gezogen, sondern nur die ersten 91 Werte, die umgerechnet die ersten $45{,}5^0$ des Scan Bereichs auf der rechten Seite des mobilen Roboters ergeben. Der Scan beginnt mit dem ersten Messwert auf der rechten Seite des mobilen Roboters und geht bis zum Wert 361 auf der linken Seite. Das heißt also, das nur die Objekte innerhalb der ersten $45{,}5^0$, also nur Objekte rechts von dem mobilen Roboter, in die Gewichtung zum `Attract-to-wall` Vector einwirken.

$$\vec{A} = \left(\frac{1}{||Ergebnisvektor||}\right)^4 \cdot 2 \cdot 10^6$$

Die Formel zur Skalierung des `scaled-Attract-Vektors` war auch hier wieder eine Herausforderung, die ebenfalls in Abschnitt 4.4 genauer erklärt wird.

Sie hat eine geringere Streckung als der Feelforce Vektor. Dieser skalierte Vektor wird dann zur Verrechnung als Ausgabevektor verschickt.

```
(defmodule attract-to-wall ((:inputs scan-points)
                            (:outputs attract-vector))
  "This module creates a Vector that attract the robot to the
  walls righthanded."
  (setf attract-vector
    (reduce #'geo:add
      (mapcar
        (lambda (point)
          (let ((result (geo:make-2d-vector
                            (geo:x point) (geo:y point))))
            (geo:multiply
              (expt (/ 1 (geo:vector-length result)) 4) result)))
        (coerce (subseq scan-points 0 90) 'list))))
  (let ((scaled-attract (* 2000000 attract-vector)))
    (send-output 1 scaled-attract)))
```

4.3.6 Straight Walker

Das `Straight-walker`-Modul erzeugt eine Vorwärts-Bewegung, die durch einen Vector der Länge 200 in Y-Richtung (siehe Abbildung 4.1), also vorwärts, des mobilen Roboters verkörpert wird.

```
(defmodule straight-walker ((:inputs scan-vector)
                            (:outputs ahead-free))
  "Creates a straight-vector for moving straight forward"
  (setq ahead-free
        (not (find-if (lambda (distance)
                   (< distance 500))
                 scan-vector :start 121 :end 240)))
  (if ahead-free (send-output 1 #v(0 200))))
```

Dieser Vektor wird aber nur zur Verrechnung verschickt, wenn die Variable `ahead-free` TRUE ist. Dies ist der Fall, wenn sich in dem Bereich von 61^0 bis $120{,}5^0$, also direkt vor dem mobilen Roboter, nichts näher als 50 cm befindet. Als Ausgabewert steht `ahead-free` für weitere Module zur Verfügung.

4.3.7 Adjuster

Mit dem Adjuster Modul besteht die Möglichkeit, die Ausrichtung des mobilen Roboters zu korrigieren, wenn er nicht mehr exakt ausgerichtet zur Wand steht oder in eine Ecke, bzw. trichterförmigen Sackgasse fährt. Das Modul hat zwei Variablenfunktionen. Es handelt sich bei den Variablen um `left-free` & `right-free`. Bei der 1. Funktion wird `left-free` bestimmt, indem der Bereich von 121^0 bis 181^0 auf Objekte in einer Entfernung näher als 70 cm kontrolliert wird. Wenn sich kein Objekt in diesem Bereich befindet wird left-free auf TRUE gesetzt. Gleichermaßen geht es bei der 2. Funktion vor. Hier wird jedoch der Bereich von 0^0 bis $70{,}5^0$ kontrolliert. Die Entfernung, in der sich keine Objekte befinden dürfen beträgt 90 cm.

```
(defmodule adjuster ((:inputs scan-vector corner ahead-free)
                     (:outputs left-free)
                     (:variables right-free))
  "Adjusts the robot, if at the right side and the front are walls
   less than 90 cm & 70 cm, to the left if this side is free."
  (setq left-free
        (not (find-if (lambda (distance)
                        (< distance 700))
                      scan-vector :start 241 :end 361)))
  (setq right-free
        (not (find-if (lambda (distance)
                        (< distance 900))
                      scan-vector :start 0 :end 140)))
  (if (and (not right-free) (not ahead-free) left-free)
    (send-output 1 #c(-40 100)))
  (if (and (not corner) (not ahead-free) (not left-free))
    (send-output 1 #c(-60 0))))
```

Anhand zweier `if`-Abfragen wird entschieden, ob der Ausgabevektor ein Vektor ist, der eine reine Drehbewegung ausgibt, oder einen Bewegungsvektor der einen Vorwärtsimpuls mit einem kleinen Drehimpuls gibt. Für diese Abfragen werden nun Eingabewerte von Variablen anderer Module benötigt. Eingabewerte dieses Moduls sind: `scan-vector`, `corner` und `ahead-free`. Wenn der Roboter sich nun z.B. in einer Ecke befindet, in der sich rechts und vor ihm eine Wand befindet, links jedoch frei ist, so soll er eine gemächliche Kurve nach Links fahren. Es wird also ein Vektor (-40 100) ausgegeben. Befindet sich der Roboter jedoch in einer Sackgasse, in der es sich nicht um eine Ecke handelt,

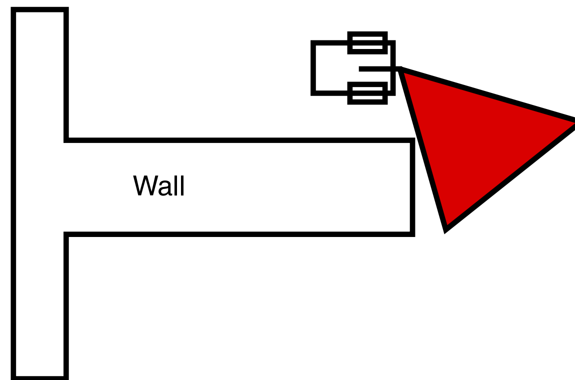

Abbildung 4.2: Zu umfahrende Ecke

vorne, sowie links nicht frei ist, so soll er sich nur drehen, ohne einen Vorwärtsimpuls zu erhalten. Wir haben dann also als Ausgabevektor (-60 0) mit dem wir eine Drehung des mobilen Roboters veranlassen.

4.3.8 Corner

Dieses Modul überprüft, ob sich rechts vor dem mobilen Roboter ein Ende eines Hindernisses befindet. Diese „Ecke“ soll der mobile Roboter wahrnehmen und sie umfahren. Das `Corner` Modul überprüft in einer Abfrage, ob sich in dem Scan Bereich von $10{,}5^0$ bis $70{,}5^0$ kein Entfernungsvektor mit einer Entfernung geringer als 70 cm befindet. Wenn der Roboter nun an einer Wand entlang fährt, hat er zur Wand die ganze Zeit eine Entfernung von etwa 40 cm bis 60 cm, bedingt durch das Feelforce- und Attract to Wall Modul. Sollte er sich zu weit von der Wand entfernen, oder die Wand aufhören, wird durch die Abfrage die Variable `corner` auf TRUE gesetzt. Hierbei wird ein Vektor für das befahren einer Rechtskurve erzeugt, dieser Vektor ist (50 20). Sollte der Roboter nach einem Durchlauf in seinem eben erwähnten Scanbereich immer noch kein Objekt haben, so wird er wieder den Vektor (50 20) als Ausgabevektor senden. Ich habe bei den Werten dieses Vektors bewusst eine leichte Vorwärtsbewegung mit einfließen lassen, damit er sich in einem Parabelbogen um die Ecke bewegt. Außerdem wäre die Nutzung eines reinen Drehungsvektors ohne Vorwärtsbewegung fatal, da der Roboter die Ecke ja schon erkennt, bevor er sie wirklich erreicht (siehe Abbildung 4.2), denn er überprüft ja erst den 21. Entfernungswert, also ab $10{,}5^0$, ob kein Objekt im Sichtfeld ist.

Da mit diesem Modul in der ersten Entwicklungsstufe Probleme entstanden sind, wenn sich der Roboter in seiner Ausgangsposition in der Mitte eines Raumes aufgehalten hat, musste dieses Modul noch etwas erweitert werden.

Der Roboter hat in der Mitte des Raumes kein Objekt in seinem Scanbereich, wenn das nächste Objekt weiter entfernt war als 70 cm, berechnete das Modul Corner, dass es sich hierbei um eine Ecke handelte. Im nächsten Schritt war dann wieder kein Objekt in der nähe und er drehte sich wieder. Das ging dann so weiter, bis der mobile Roboter seine Akkuleistung verbraucht hat, und er dabei immer wieder in Kreis fuhr.

Ich wollte dem Roboter aber nicht die Möglichkeit nehmen, sich einmal im Kreis zu Orientieren, damit er kontrollieren kann, ob sich hinter ihm evtl. ein Objekt findet, an dem er sein Follow-Right-Wall Verhalten beginnen kann. Ich habe nun also noch einen Zähler eingebunden, der synchron mit der Ausgabe des Drehvektors erhöht wird. Wenn er 28 erreicht hat, setzt er die Variable `inhibit` auf TRUE und führt nicht mehr die Abfrage Corner aus. Diese Unterdrückung der Corner-Abfrage führt dazu, dass sich der Roboter nur noch gerade bewegt (Ausgenommen der Ablenkungen durch Feelforce und Attract to Wall), und zwar so lange, bis der Absolutbetrag des `Feelforce-Vektors` größer als 5 ist. Wenn dieser Betrag wächst, kommt der Roboter also einem Objekt nahe. Dieses Verhalten der Kreisbahn können sie zu Beginn der beiden Testläufen aus Abbildung 4.8 & 4.9 deutlich erkennen.

Die Schlussfassung des Corner Moduls sah nun folgendermaßen aus:

```
(defmodule corner ((:inputs scan-vector scaled-feelforce)
                   (:outputs corner)
                   (:variables (circle-counter 0)
                               (inhibit nil)))
  "This function will guide the robot around a corner"
  (if (> 28 circle-counter)
    (progn (setq corner
                 (not (find-if (lambda (distance)
                                 (< distance 700))
                               scan-vector :start 20 :end 140)))
           (setq inhibit nil)
           (when (not corner)
             (setq circle-counter 0)))
    (let ((new-inhibit (< (abs scaled-feelforce) 5)))
      (when (and inhibit (not new-inhibit))
        (setq circle-counter 0))
      (setq inhibit new-inhibit)))
  (when (and corner (not inhibit))
    (send-output 1 #c(50 20))(incf circle-counter)))
```

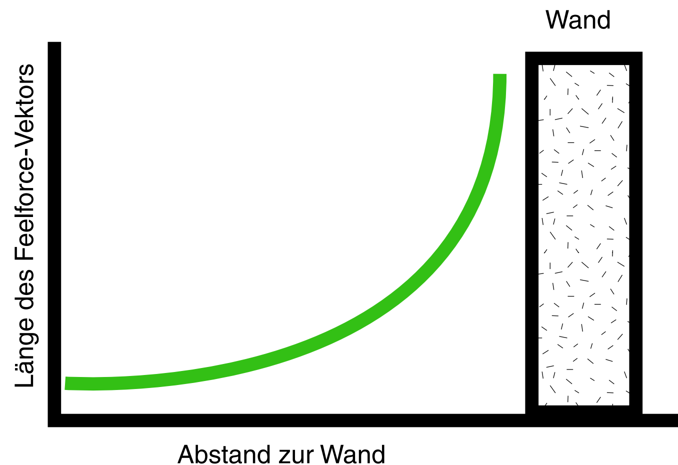

Abbildung 4.3: Abstoßung durch Feelforce von der Wand

4.4 Verhalten und Beziehungen der Module

Wie schon im Abschnitt 4.3.3 erwähnt kann man also schon mit drei Modulen (`Sense`, `Collision-avoidance` und `Wander`) ein funtionsfähiges Kontrollsystem entwerfen. Das System ist zwar für sich genommen simple und entspricht keineswegs einem Follow-Right-Wall Verhalten, welches ja in dieser Arbeit eigentlich entwickelt werden sollte. Es ist jedoch nicht zu verachten, das sich das gezeigte Verhalten, kreuz und quer durch den Raum zu wandern ohne anzustoßen, mit dem chaotischen Laufverhalten einiger Insekten vergleichen läßt.

Im weiteren blieb mir leider nicht erspart, dass sich einige Module untereinander beeinflussen. Diese Beeinflussung wollte ich eigentlich unterbinden, um den Ansatz von Brooks gerecht zu werden. Jedoch würde eine strikte Trennung der Module voneinander bei meinem Programm eine Menge redundanten Lisp-Code und eine höhere Berechnungszeit hervorrufen. Somit habe ich mich dazu entschieden, einige Module in Abhängigkeiten zu setzen. Das `Feelforce` Modul und das `Attract to Wall` Modul bilden zusammen, auch wenn es nicht so erscheinen mag, da ja keine globalen Variablen des jeweils anderen Moduls benutzt werden, einen Regelkreislauf. Das Feelforce Modul erzeugt, wie schon in 4.3.4 erwähnt, einen Vektor, einen Absolutbetrag der einzelnen negativen Scan-Vektoren, der von den Objekten weg zeigt. Je näher der mobile Roboter der Wand kommt, desto stärker wirkt der Vektor auf ihn und drückt ihn weg. In Abbildung 4.3 versuche ich das Verhalten grafisch Darzustellen. Die exponentielle Kurve wird durch die in Abschnitt 4.3.4 vorgestellte Formel erreicht.

Der Versuchsaufbau zur Konstruktion der Formel sah so aus, dass der mobile Roboter aus einer Entfernung von etwa 1,5 Metern und einem Öffnungswinkel

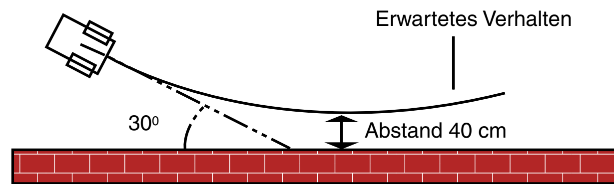

Abbildung 4.4: Aufsicht des Feelforce-Verhaltens

von bis zu 30^0 zur Wand folgendes Verhalten zeigen sollte. Er sollte sich Anfangs der Wand in seinem Winkel von 30^0 nähern und sich langsam parallel zu Ihr ausrichten, indem er den Winkel zur Wand immer mehr verringert. Dabei sollte er der Wand nicht näher als 40 cm bis 50 cm kommen. Der mobile Roboter würde jedoch nicht parallel zur Wand ausgerichtet bleiben sondern sich wieder von Ihr entfernen, da er durch den Vektor noch zu sehr weggedrückt wurde (vgl. Abbildung 4.4). Das hätte man jedoch noch so regeln können, das er sich zwar parallel zur Wand ausrichtet, und nicht wieder von der Wand weg bewegt, indem man einen geringeren Streckungsfaktor als $75 \cdot 10^5$ gewählt hätte. Dabei hätte sich aber auch der mögliche Öffnungswinkel von 30^0 verringert, da der Roboter es mit diesem Winkel nicht mehr rechtzeitig geschafft hätte, sich vor erreichen des Mindestabstandes parallel zur Wand auszurichten. Allerdings wollten wir den mobilen Roboter ja auch zu einer Wand hinziehen, wenn er sich von Ihr wegbewegen würde, und nicht nur parallel ausrichten, wenn er zufällig auf eine Wand zu fährt.

Als Abhilfe kam also die Einführung des `Attract-to-Wall` Moduls. Dieses Modul erzeugt eine negative exponentielle Anziehungskurve wie in 4.5 dargestellt. Dieses Modul für sich würde den mobilen Roboter so lange weiter zur Wand hin ziehen, bis er in die Sperre für den Mindestabstand von Objekten läuft, was durch das Modul `Collision-avoidance` kontrolliert wird.

Das Ziel war jedoch nicht, den mobilen Roboter permanent zur Wand zu ziehen oder von ihr abzustoßen, sondern ein Regelungssystem zu finden, das den Roboter in einem ca. 40 cm breitem Korridor in einem sicheren Abstand zur Wand hält. Fährt der Roboter zu weit von der Wand weg, so greift das `Attract-to-Wall` Modul viel stärker ein als das `Feelforce` Modul und zieht es wieder zur Wand und umgekehrt. Die parallele Nutzung der beiden Module führt also dazu, den Roboter in der Nähe des Schnittpunktes der beiden Überlagernden exponentiellen Kurven zu halten (siehe 4.6). Daher war es auch sehr schwer die beiden Formeln in den Modulen zu erstellen. Die genaue Abstimmung der Größen der Streckungsfaktoren beider Formeln war somit von großer Bedeutung und erforderten eine Menge Testläufe bis zur zufrieden stellenden

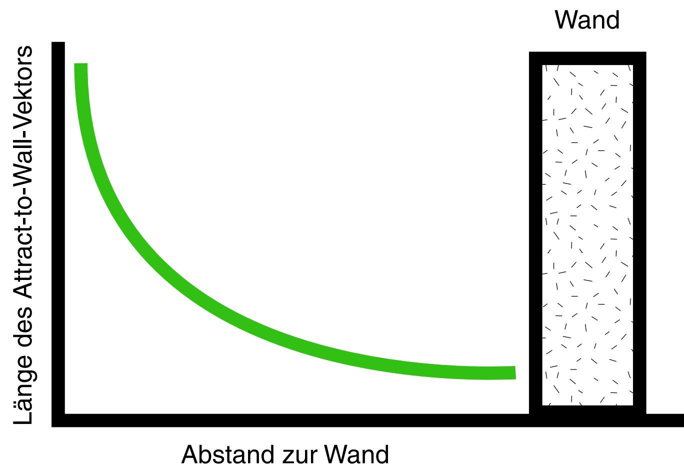

Abbildung 4.5: Anziehung durch Attract to Wall zur Wand

Version.

Problem hierbei ist aber, das bei dem Ausfall eines dieser beiden Module die Funktionsweise des mobilen Roboters stark beeinträchtigt wäre und er möglicherweise nicht mehr seine Ziele erfüllen kann.

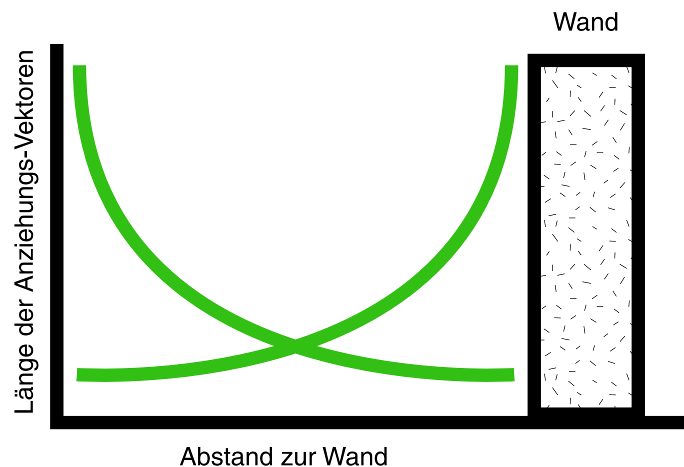

Abbildung 4.6: Regelungssystem zur Wand

Diese Module haben es nun also geschafft, den mobilen Roboter bis zu einem Winkel von 30^0 auszurichten. Doch was passiert, wenn der Roboter senkrecht auf eine Wand oder ein Objekt zufährt? Die Ausrichtung wird dann von dem `Corner` und `Adjuster` Modulen vorgenommen.

Ein weiteres Modul das sich das `Feelforce` Modul zu Hilfe nimmt ist das `Corner` Modul. Wie ja schon in Abschnitt 4.3.8 dargestellt wurde, unterdrückt

er die Ausgabe eines Drehvektors nach erreichen der Grenze des Zählers so lange, bis er wieder in die nähe eines Objektes gelangt und der Absolutbetrag des skalierten Feelforce-Vektors größer als 5 ist. Sollte das Feelforce Modul ausfallen, so kann auch dieses Modul nicht mehr einwandfrei arbeiten.

Das `Corner` Modul korreliert außerdem noch mit dem `Adjuster` Modul. Die Namenswahl für das letztere Modul war vielleicht nicht ganz treffend, denn die Funktionalität der richtigen Ausrichtung des mobilen Roboters wird durch die Kombination dieser beiden Module und dem `Straight-Walker` Modul erreicht. Ich wollte das `Corner` Modul Aufgrund seiner Komplexität nicht mehr erweitern. Ebenso wenig wollte ich in das `Adjuster` Modul zusätzlich noch eine Funktion einbauen, die zur Überprüfung des Vorhandenseins einer Ecke dient, und redundant zur Cornerfunktion wäre. Das `Adjuster` Modul interagiert jedoch nicht nur mit dem `Corner`, sondern auch mit dem `Straight-Walker` Modul. Es bekommt den boolschen Wert `ahead-free` von dem `Straight-Walker` Modul und bedient sich desweiteren der ebenfalls boolschen Variable `Corner` aus dem gleichnamigen Modul. Außerdem erhält er natürlich noch die Liste des `Scan-Vektors`.

Er nutzt die Eingaben `Corner` und `ahead-free` aus den anderen Modulen bei den letzten `if`-Abfragen um zu entscheiden, ob die Ausgabe ein Vektor in -X–Richtung sein soll, also eine Bewegung nach links ausüben soll, oder ob der auszugebende Vektor in Y– & -X–Richtung zeigen soll, also eine Linkskurve mit Vorwärtsbewegung ausüben soll. Bei diesem Modul sei auch wieder erwähnt das es nicht korrekt arbeiten würde, wenn eines der anderen Module Fehler produzieren würde.

4.5 Testlauf

Ich möchte Ihnen natürlich nicht die Grafiken meiner Testläufe vorenthalten. Doch zuerst möchte ich mit Abbildung 4.7 kurz zeigen, wie sich der Roboter gewünschter Weise verhalten sollte. Diese Abbildung können man auch als Detaillierte Darstellung der Verhaltensweisen an den Ecken und Problemstellen der Testläufe betrachten.

In den Abbildungen 4.8 und 4.9 sehen sie jeweils Testläufe ausgehend vom gleichen Startpunkt. Diese beiden Testläufe habe ich deshalb gewählt, weil sie ein unterschiedliches Verhalten aufweisen. Im beiden Testläufen beginnt der Roboter mit dem geschilderten Verhalten, dass er eine Ecke berechnet und erstmal so lange im Kreis fährt, bis die `corner`-Abfrage eingestellt wird. Sie wird erst wieder aktiviert, als der Roboter nach langer geradeaus Fahrt eine Wand erreicht. Dort beginnt nun sein Follow-Right-Wall Verhalten. Als er die erste Ecke

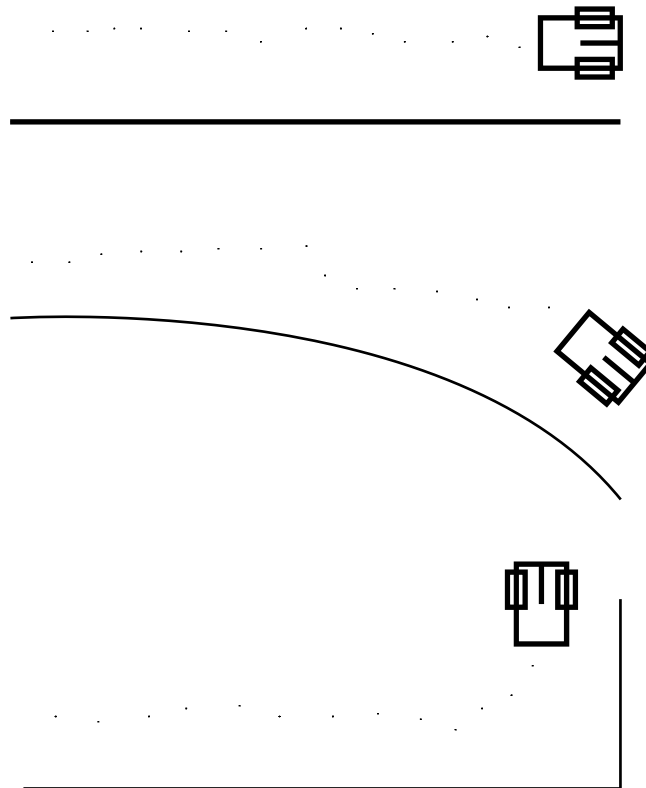

Abbildung 4.7: Verhaltensweisen des Roboters

des Raumes (rechts unten in der Abbildung 4.8) erfolgreich passiert hat und auf dem Weg entlang der rechten Wand ist, schafft er es in dem ersten Testlauf zwischen der Wand und dem Objekt hindurch zu fahren. Der Abstand zwischen Wand und Objekt ist an der Stelle nach Abzug der Mindestabstände von jeweils 35 cm so gering, dass dem mobilen Roboter ein nur etwa 5 cm schmaler Korridor bleibt, durch den er diese Enge passieren kann. Bei der Zweiten Runde richtet er sich sogar nochmal richtig zu diesem schmalen Korridor aus und schafft es erneut hindurch.

Im 2. Testlauf hingegen, wird er gleich so korrigiert, dass er an dem Objekt entlangfährt und nicht weiter durch den Korridor an der Wand entlang. Sogar als er von oben an die Enge kommt, schafft er es nicht hindurch. Dies hätte jedoch passieren können, wenn er sich wie schon erwähnt, im richtigen Abstand zu der Wand und dem Objekt befunden hätte. Somit wäre er nochmals die Objekte im Raum abgefahren und womöglich wieder und wieder.

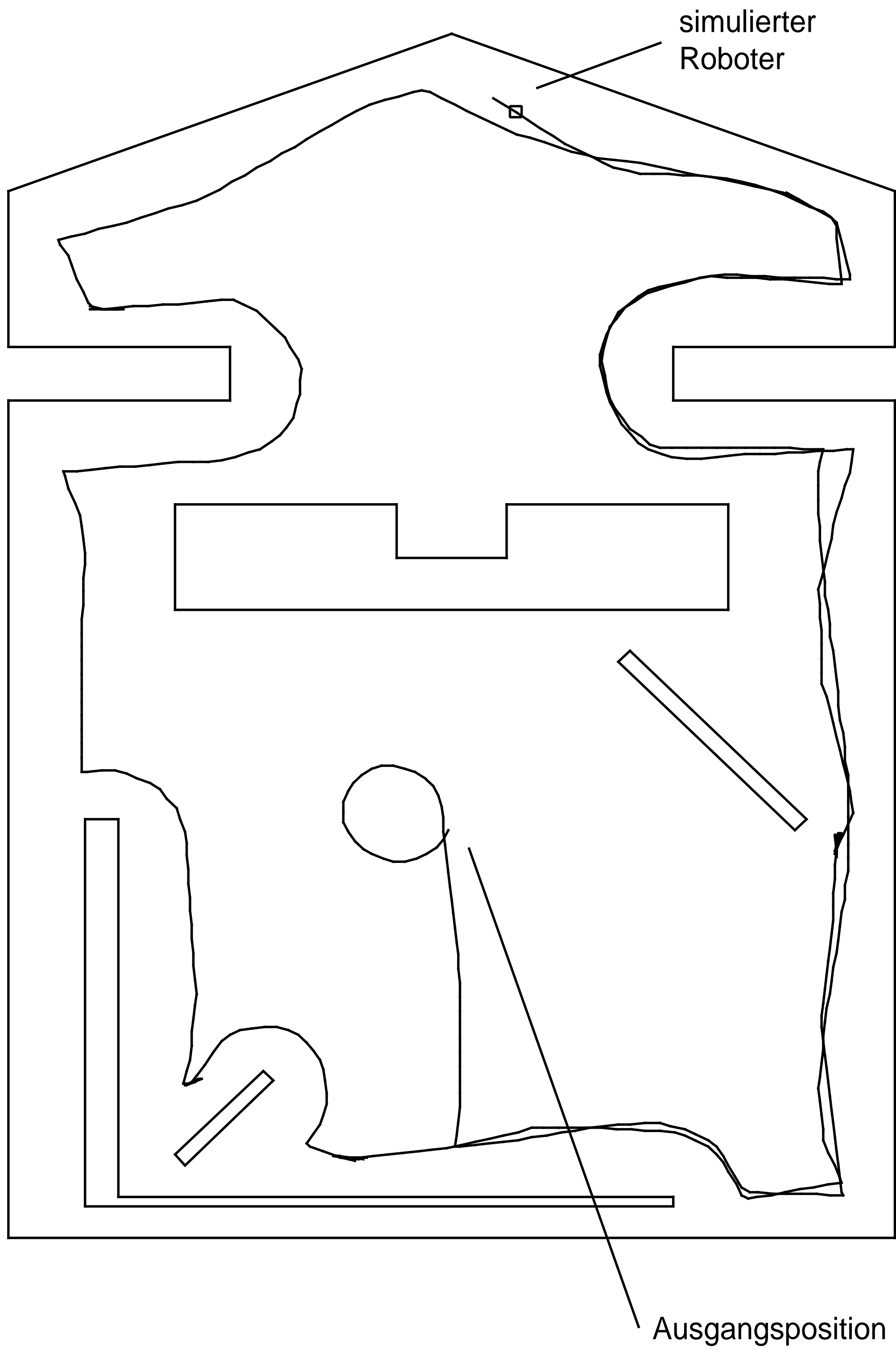

Abbildung 4.8: Testlauf 1

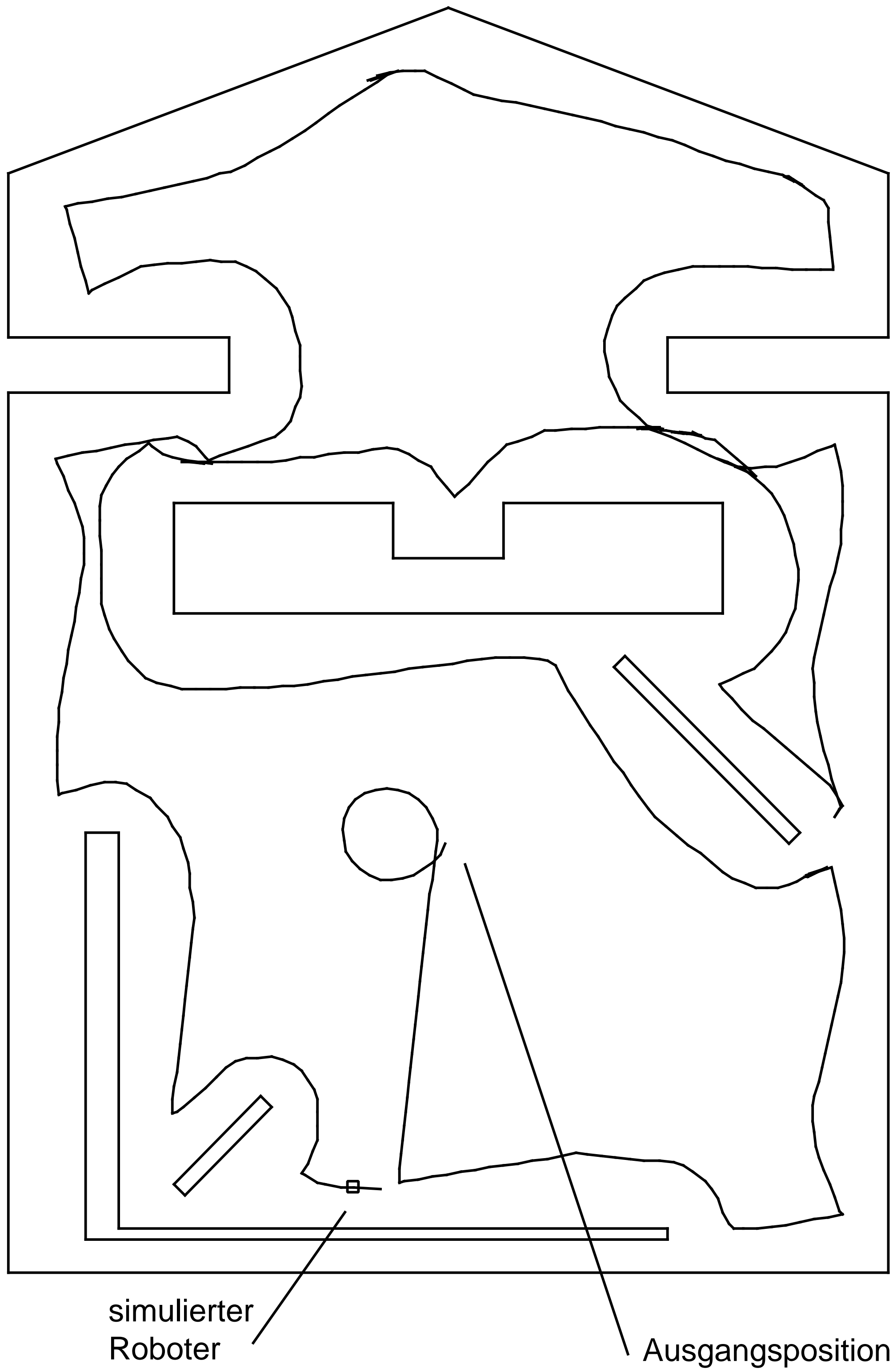

Abbildung 4.9: Testlauf 2

Kapitel 5

Bewertung und Ausblick

Die in dieser Arbeit erstellte Kontrollstruktur erfüllt zwar nicht optimal einen Follow-Right-Wall Algorithmus, aber er tut seine Arbeit zumindest hinreichend. Meine Arbeit sollte sich an den Ideen von Rodney A. Brooks orientieren, hat aber seine eigene Form angenommen. Die Realisierung der Kompetenzebenen von Brooks ist in meiner Arbeit nicht ganz verwirklicht worden. Dies hat aber den Grund, das es sich bei dieser Kontrollstruktur noch um einen Algorithmus handelt, den man ohne die Nutzung von Kompetenzebenen erstellen kann. Schwieriger würde es nun werden, wenn der mobile Roboter zusätzlich auch noch eine Wegplanung im Sinne der anzufahrenden Objekte machen müsste. Die Idee bei Brooks Kompetenzebenen war ja, dass die Ausgaben höherer Schichten, beim erreichen eines Schwellwertes, die Ausgaben niedriger schichten unterdrücken können. In meinem Programm war es jedoch nicht zwingend notwendig, diese Funktionalität zu haben.

Doch die Idee dieser Arbeit sollte die Zerlegung eines Kontrollproblems für mobile Roboter in Module sein, sowie die inkrementelle Erweiterbarkeit an Funktionalitäten. Die Idee der Kontrollstrukturen basierend auf Verhaltensweisen ist ein bedeutender Ansatz des KI Bereichs und wird sicher noch einiges an Potential bieten. Nicht nur in der Forschung sondern auch in der Anwendung wird diese Idee der Kontrollstruktur sicher noch größere Beliebtheit finden. Wenn es mehr verhaltensbasierte Systeme gibt, werden viel höhere Stufen der integration benötigt werden, aber das in dieser Arbeit behandelte low-level Verhalten ist zum grundsätzlichen Bestehen und Navigieren notwendig.

Literaturverzeichnis

[1] Graham, Paul. *ANSI Common Lisp*, Prentice-Hall, 1996

[2] Brooks, Rodney A. *Integrated Systems Based on Behaviours*, SIGART Bulletin (2:4), August 1991, pp. 46-50. (http://www.ai.mit.edu/people/brooks/papers/ssymp.pdf)

[3] Brooks, Rodney A. *A Robust Layered Control System for a Mobile Robot*, IEEE Journal of Robotics and Automation, Vol. 2, No. 1, pp. 14-23; March 1986; also MIT AI Memo 864, September 1985. (http://www.ai.mit.edu/people/brooks/papers/AIM-864.pdf)

[4] Brooks, Rodney A. *Intelligence Without Representation*, Artificial Intelligence Journal (47), 1991, pp. 139-159. (http://www.ai.mit.edu/people/brooks/papers/representation.pdf)

[5] Hermann Engesser [Hrsg.], *DUDEN Informatik*, 2. Auflage, DUDENVERLAG, 1993, Mannheim·Leipzig·Wien·Wien.