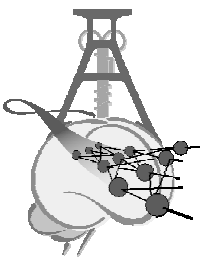
Diplomarbeit

Graphbasierte Navigation innerhalb von Gebäuden für einen autonomen Roboter

Stefan Haustein

Diplomarbeit
am Fachbereich Informatik
der Universität Dortmund
Lehrstuhl VII

7. Januar 1999



Betreuer:

Prof. Dr. Heinrich Müller
Fachbereich Informatik
Lehrstuhl VII
Universität Dortmund

Prof. Dr. Werner von Seelen
Institut für Neuroinformatik
Ruhr-Universität Bochum

Danksagung

An dieser Stelle möchte ich Herrn Prof. Dr. Heinrich Müller für seine Betreuung und konstruktive Kritik und Herrn Prof. Dr. Werner von Seelen für die Übernahme der Zweitbetreuung danken. Besonderer Dank gilt Dr. Frank Joublin für seine persönliche und fachliche Unterstützung. Weiterhin danke ich Jennifer Breitbarth, Christian Igel, Jörg Pleumann und meiner Schwester Sonja.

Meinen Eltern danke ich für ihre Unterstützung meines Studiums.

Inhalt

1 Einleitung 1

1.1 Problemstellung ... 1

1.2 Entwicklung ... 3

1.3 Gliederung ... 4

2 Systemumgebung 5

2.1 Der autonome Roboter Arnold ... 5

2.1.1 Das visuelle System ... 6

2.1.2 Die Plattform ... 6

2.2 Computer- und Betriebssystem ... 7

2.3 Kommunikationssystem PLANET ... 7

2.4 Systemmodule ... 7

2.4.1 Picserver ... 8

2.4.2 Plattformkontrolle ... 10

2.4.3 Depthmap ... 10

2.4.4 Lokalnavigation ... 11

2.4.5 Verhaltenskontrolle ... 11

3 Bestehende Navigationsverfahren 13

3.1 Bitmap-Repräsentation ... 13

3.2 Graph-Repräsentation ... 14

3.2.1 Geometrieinformationen ... 15

3.2.2 Topologieinformationen ... 15

4 Erste Umweltrepräsentation 17

4.1 Anforderungen ... 17

4.2 Vorarbeit ... 18

4.3 Entwurf ... 19

4.3.1 Dimensionalität 20
4.3.2 Koordinatensystem 20
4.3.3 Konvexe Zellen 21
4.3.4 Knoten und Kanten 22
4.4 Implementierung 23
4.4.1 Datenstrukturen 23
4.4.2 Positionsaktualisierung 24
4.4.3 Rekalibration 25
4.4.4 Einfügen neuer Daten 27
4.4.5 Sicherstellung der Konvexität 28

5 Erweiterte Repräsentation 33
5.1 Anforderungen 33
5.2 Entwurf 36
5.2.1 Geradendetektor 36
5.2.2 Kartenrepräsentation 38
5.2.3 Rekalibration 39
5.2.4 Globale Selbstlokalisation 40
5.3 Implementierung 40
5.3.1 Datenstrukturen 40
5.3.2 Änderungen im globalen Ablauf 42
5.3.3 Geradenextraktion 42
5.3.4 Rekalibration 47
5.3.5 Aktualisierung der Datenstruktur 51
5.3.6 Globale Selbstlokalisation 54

6 Integration der Repräsentation 57
6.1 Anforderungen 57
6.2 Entwurf 57
6.3 Implementierung 59
6.3.1 Datenstrukturen 59
6.3.2 Hauptschleife 60
6.3.3 Verhalten und Zielgenerierung 60
6.3.4 Nachrichtenverarbeitung 62

7 Visualisierung 63

7.1 Anforderungen 63
7.2 Entwurf 64
7.3 Implementierung 64
7.3.1 Visualisierungsserver 64
7.3.2 Visualisierungsclient 65

8 Ergebnisse **67**
8.1 Erste Umweltrepräsentation 67
8.2 Erweiterte Umweltrepräsentation 70

9 Diskussion **75**
9.1 Vergleich mit anderen Arbeiten 75
9.2 Bezug zur Biologie 76

10 Zusammenfassung und Ausblick **77**
10.1 Zusammenfassung 77
10.2 Ausblick 77
10.2.1 Wanddarstellung 78
10.2.2 Inverse Perspektive 78

A Positionsunsicherheit in der ersten Repräsentation **81**

B Belegung der Konstanten **83**

C Dokumentation der Visualisierungsschnittstelle **85**
C.1 Klasse Canvas 85
C.2 Klasse CanvasEvent 87
C.3 Klasse Point 87
C.4 Klasse Line 88
C.5 Template-Klasse Matrix 90

Literatur **93**

Kapitel 1

Einleitung

Roboter, von K. Capek geschaffene Bez. für Maschinenmenschen. [Bro83]

Der Traum von Maschinen, die sich selbständig bewegen und handeln um den Menschen zu dienen, fasziniert seit Anfang dieses Jahrhunderts Science-Fiction-Autoren und -Leser wie ernsthafte Wissenschaftler. Von der Vorstellung eines „Maschinenmenschen“ allerdings sind heutige Industrie- und Forschungsroboter noch weit entfernt. Ergibt sich in den Science-Fiction-Romanen die Spannung meist durch Probleme, die durch die Intelligenz der Roboter entstehen, so beschäftigt sich die Wissenschaft noch damit, den Robotern die nötige Intelligenz zu geben, um sich überhaupt in der Umwelt zurechtzufinden.

Motivation für den Bau von „Maschinenmenschen“ ist neben dieser Faszination auch, mehr über den Menschen selbst zu erfahren. Jeder, der sich mit dem Problem beschäftigt, einen Computer selbst einfache Dinge auf einem Bild erkennen zu lassen, bekommt großen Respekt vor der Leistung des menschlichen Sehsystems, die sonst eher als selbstverständlich hingenommen wird und erhält einen kleinen Einblick, welche Aufgaben im Gehirn gelöst werden müssen und welche Ansätze dazu beitragen könnten.

1.1 Problemstellung

Im Rahmen dieser Diplomarbeit soll ein System entwickelt werden, mit dessen Hilfe ein *autonomer Roboter* sich innerhalb eines Gebäudes zurechtfinden kann. Ein autonomer Roboter ist ein Roboter, der sich im Gegensatz zu fest montierten Industrierobotern weitgehend unabhängig bewegen kann.

Bei der Bewegung eines Roboters gibt es zwei wesentliche Aufgaben:

1. Der Roboter darf nicht mit Hindernissen kollidieren. Dazu muß er mit einem Hindernisvermeidungsprogramm ausgestattet werden.
2. Die Bewegung unter Kollisionsvermeidung ist kein Selbstzweck. Zur sinnvollen Bewegung braucht der Roboter eine interne Karte seiner Umwelt, mit der er den Weg zu seinem Zielort planen kann.

Einfache Möglichkeiten, eine solche Karte zu erstellen, sind sicherlich, vorab die Umgebung des Roboters zu vermessen oder eventuell vorhandene Gebäudepläne zu benutzen.

Eine flexiblere Alternative wäre, den Roboter eine solche Karte durch Erkundung seiner Umgebung selbst erstellen zu lassen. Die Realisierung dieses Zieles ist Gegenstand dieser Arbeit.

Zielsystem für das Navigationssystem ist der am Institut für Neuroinformatik der Ruhr-Universität Bochum entworfene und gebaute autonome Roboter *Arnold*, der in Abbildung 1.1 dargestellt ist. Arnold wurde für Arbeiten im Servicebereich in menschlicher Umgebung konzipiert und besitzt eine dem Menschen nachempfundene Anatomie.

Abbildung 1.1: Der Roboter des Institutes für Neuroinformatik „Arnold“

Andere existierende autonome Roboter besitzen meistens spezialisierte Sensoren wie hochsensible Laser-Abstandsmesser oder einfachere Infrarot- oder Sonarsensoren; besondere Anforderung an das zu entwickelnde Navigationssystem für den Roboter Arnold ist es jedoch, keine technischen Hilfsmittel zu benutzen, die der Mensch prinzipiell nicht besitzt. Für die Navigation beschränkt sich die Sensorik somit auf ein visuelles System und die über die Räder gemessenen Relativbewegungen. Dem Roboter stehen zur Kartenerstellung Daten eines Stereokamerasystems, das für senkrechte Bildkanten Tiefenmessungen vornehmen kann, zur Verfügung.

Bei der Erstellung einer Karte ergeben sich verschiedene Probleme:

- Die Bewegungsmessung des Roboters ist ungenau. Diese Ungenauigkeit soll durch einen Abgleich der gesehen Daten mit bereits in die Karte eingetragenen Daten ausgeglichen werden.
- Die Daten des Stereokamerasystems sind verrauscht.

- Die Daten des Stereokamerasystems sind mit einem überproportional zu der Entfernung steigendem Fehler behaftet.
- Da das Stereokamerasystem nur senkrechte Kanten „sehen“ kann, darf das Nichtvorhandensein von Tiefenmessungen nicht als freier Fahrraum interpretiert werden.

Ausgangspunkt für diese Arbeit ist die Diplomarbeit von Daniel Tepas [Tep97], in der bereits eine Graph-Struktur für die Karte entwickelt und in Teilen simuliert wurde.

Die von Daniel Tepas entwickelte Umweltrepräsentation und das Verhalten des Roboters zur Erkundung seiner Umwelt soll vervollständigt, auf den Roboter Arnold übertragen und an die mit Realdaten auftretenden Probleme angepaßt werden.

1.2 Entwicklung

Leider verlief die Entwicklung des Navigationssystems etwas anders als ursprünglich geplant, da die von der Stereoverarbeitung gelieferten Daten nicht der Erwartung entsprachen. An dieser Stelle wird daher auf die einzelnen Arbeitsschritte eingegangen, die auch die Gliederung dieses Textes bestimmen.

Zu Beginn der Arbeit war das in der Parallelarbeit von Martin Barkanowitz [Bar98] zu portierende und auszubauende Stereosystem noch nicht verfügbar. Daher wurde zunächst eine erneute Simulation entwickelt, die für eine vorgegebene Karte die Punkte generieren konnte, die von jeder vorgegebenen Position aus sichtbar wären. Im Unterschied zu der bereits von Daniel Tepas durchgeführten Simulation war diese in Bezug auf die Schnittstellen schon weitestgehend identisch zum Stereosystem und zur Ansteuerung der Plattform zur Bewegung des Roboters, um die anschließende Portierung des eigentlichen Navigationsprogrammes auf den Roboter zu erleichtern.

Bei der Umsetzung des eigentlichen Navigationssystems wurden einige Schwachpunkte deutlich; so war der Algorithmus zur Aufteilung der Karte in konvexe Teilräume nicht ganz korrekt und die ursprüngliche Simulation war nicht über einen Standpunkt des Roboters hinaus dokumentiert worden. Der Schwerpunkt wurde zunächst also auf die Vervollständigung des Navigationsverfahrens – insbesondere den korrekten Aufbau der Datenstruktur über einen Roboterstandpunkt hinaus – gelegt.

Zusätzlich wurde ein Programm entwickelt, über das die interne Karte des Roboters sowohl aus der Simulation heraus als auch von dem eigentlichen Roboter aus dargestellt werden konnte. Das Visualisierungsprogramm, das während der gesamten Zeit weiterentwickelt wurde, wird mittlerweile auch von anderen Modulen des Roboters genutzt.

Bis zur Verfügbarkeit von Realdaten konnte das Navigationsverfahren eine Simulationsumgebung vollständig und korrekt kartieren. Tests mit Realdaten verliefen jedoch enttäuschend. Insbesondere war beim Design des Navigationssystems zwar eine Fehlerabweichung der vom Stereoalgorithmus gelieferten Punkte berücksichtigt worden, nicht jedoch, daß durch Fehlzuordnungen der Kanten des rechten und linken Kamerabildes vollkommen falsche Tiefenwerte geliefert werden konnten.

Aber selbst ohne Fehlzuordnungen schien eine Wiedererkennung einzelner Punkte mit dem bestehenden Verfahren hoffnungslos.

Allerdings konnte man in den stark verrauschten Daten die meisten Wände relativ gut erkennen, was zu der Idee führte, daß es auch dem Roboter möglich sein müßte, diese Wände zu erkennen

und durch Einführung dieser zusätzlichen Abstraktionsebene insgesamt stabilere Merkmale zu erhalten.

Obwohl die Arbeit aufgrund der späten Verfügbarkeit von Realdaten schon relativ weit fortgeschrittenen war, wurde das bestehende Konzept zu einem großen Teil verworfen und mit dem Design einer neuen Repräsentation begonnen, um diese – so weit wie in der verbleibenden Zeit möglich – umzusetzen.

Wie bei den Ergebnissen dokumentiert, konnten tatsächlich fehlerfreie Karten des Testraumes von beliebigen Positionen aus erstellt werden.

1.3 Gliederung

Das folgende Kapitel 2 „Systemumgebung“ enthält eine Beschreibung des eingesetzten Robotersystems einschließlich der Softwaremodule, auf die zurückgegriffen werden konnte.

Kapitel 3 beschreibt die für diese Arbeit relevanten Teile bestehender Roboternavigationsverfahren.

Aufgrund der besonderen Bedeutung der Repräsentation und der aufgetretenen Probleme mit Realdaten, die einen Neuentwurf erforderlich machten, werden die beiden Umweltrepräsentationen zuerst in Kapitel 4 und 5 einzeln behandelt, bevor in Kapitel 6 die Integration der Repräsentation in das Gesamtsystem und das Zusammenwirken mit den anderen Modulen des Roboters beschrieben werden.

Kapitel 7 beschreibt die entwickelte Visualisierung.

In Kapitel 8 werden die Ergebnisse dargestellt.

Ein Vergleich mit anderen Arbeiten und ein Bezug zur Biologie wird in Kapitel 9 beschrieben.

Schließlich stellt Kapitel 10 eine Zusammenfassung dieser Arbeit dar. Außerdem werden einige Erweiterungsmöglichkeiten aufgezeigt.

Kapitel 2

Systemumgebung

In diesem Kapitel wird die Systemumgebung beschrieben, die zur Entwicklung des Navigationssystems zur Verfügung stand.

Neben dem eingesetzten Roboter Arnold wird das Kommunikationssystem PLANET vorgestellt, über das die Module der Steuerungssoftware miteinander kommunizieren. Außerdem wird auf die einzelnen Softwaremodule eingegangen, die den Roboter kontrollieren oder die direkte Hardwareansteuerung übernehmen.

2.1 Der autonome Roboter Arnold

Der am Institut für Neuroinformatik der Ruhr-Universität Bochum entwickelte autonome Roboter Arnold besitzt eine dem Menschen nachempfundene Anatomie.

Anstelle von Beinen ist er allerdings mit einer Plattform mit Rädern zur Bewegung ausgestattet, die auch den Antrieb und zwei Autobatterien zur netzunabhängigen Energieversorgung enthält. Auf der Plattform sitzt der Rumpf, in dessen unterem Teil die CPU-Platinen und Erweiterungskarten untergebracht sind[1]. An dem oberen Teil des Rumpfes befinden sich Anschlüsse für Tastatur und Monitor. Arnolds Arm, der über dem Rumpf angebracht ist, besitzt sieben Freiheitsgrade. Diese dem Menschen nachempfundene Redundanz ermöglicht eine Hindernisvermeidung bei Greifbewegungen. Über der Schulter des Armes befindet sich der Kamerakopf, der mit den zwei Stereokamerapaaren ausgestattet ist.

Der für die Problemstellung wichtigste Unterschied zu anderen Forschungsrobotern ist, daß der Institutsroboter Arnold in Analogie zum Menschen nur mit visuelle Sensoren ausgestattet ist und keine Ultraschall-, Laser- oder sonstige Abstandsmesser besitzt.

Im weiteren wird detaillierter auf die Robotereigenschaften eingegangen, die für die Kartengenerierung und Orientierung von Bedeutung sind. Für eine ausführlichere Beschreibung des Roboters sei auf [BBD+97] verwiesen.

Beim Design der Navigation wird soweit wie möglich von den konkreten Eigenschaften des Institutsroboters abstrahiert.

---

[1] Arnold trifft also alle Entscheidungen aus dem Bauch heraus.

2.1.1 Das visuelle System

Arnolds optisches Sensorensystem verfügt über zwei hochauflösende Farbkameras und zwei Weitwinkel-Grauwertkameras, um wie beim Menschen eine Unterscheidung zwischen *fovealem* und *peripheren* Sehen zu ermöglichen: Der Mensch verfügt in der Fovea, die auf den Hauptaufmerksamkeitsbereich gerichtet wird, über eine extrem hohe Bildauflösung mit Sehzellen für farbiges Sehen (*Zapfen*). Der Rand der Netzhaut ist mit weniger Sehzellen ausgestattet, die den Blickbereich erweitern und zu einem Großteil auch keine Farbunterscheidung ermöglichen (*Stäbchen*; nach [BS96]). Bei Bedarf kann so ein Objekt im Peripheriebereich die Aufmerksamkeit erregen und „fovealisisert“ werden. Die Parameter der Kameratypen sind in Tabelle 2.1 zusammengefaßt.

Der Kamerakopf des Roboters, auf dem die beiden Kamerapaare montiert sind, läßt sich horizontal um 360° drehen und vertikal um 30° nach oben und unten neigen. Die Neigung der Kameras wird von dem Navigationssystem nicht genutzt.

	Fovea-Kamera	**Peripherie-Kamera**
Hersteller / Modell	Sony XC-999P	Sanyo VCK-465
Farbe	Ja	Nein
Physikalische Bildauflösung (CCD)	768×572	500×572
Logische Bildauflösung (Framegrabber)	752×582	752×582
Logische Pixelbreite	0,00833 mm	0,00638 mm
Logische Pixelhöhe	0,00825 mm	0,00618 mm
Brennweite	6 mm	3,6 mm
Horizontaler Kamerablickwinkel	54,7°	67°
Stereoabstand	300 mm	380 mm

Tabelle 2.1: Kameraparameter

Fehlerparameter des visuellen Systems

Die beschränkte horizontale Auflösung der Kameras führt zu einer Meßungenauigkeit bei der Winkelbestimmung, die dem minimal auflösbaren Winkel σ_α der Kamera entspricht. Für die Fovea-Kamera beträgt diese Meßungenauigkeit ca. 54,7° / 752 = 0,0727°; Für die Peripherie-Kamera 67° / 500 = 0,134°. Bei diesen Werten ist nicht berücksichtigt, daß der CCD-Chip, auf den das Bild projiziert wird, eine ebene Fläche darstellt.

2.1.2 Die Plattform

Die Roboterplattform ist mit zwei durch Elektromotoren einzeln angetriebenen Rädern ausgestattet und wird mit vier Laufrädern stabilisiert. Die am Institut eingesetzte Plattform kann in

einem *Punkt-zu-Punkt-Modus* oder in einem *kontinuierlichen Modus* angesteuert werden. In dem kontinuierlichen Modus werden eine Vorwärts- und eine Rotationsgeschwindigkeit gesetzt und bis zum nächsten Befehl aufrechterhalten.

Die Bewegung der Plattform wird anhand der Umdrehung der Räder berechnet. Die Abweichung der berechneten Bewegung von der tatsächlich zurückgelegten Strecke ist von vielen Faktoren abhängig, wie zum Beispiel der Beschaffenheit des Untergrundes, der Abnutzung der Räder, veränderter Reibungswerte der Räder bei Verschmutzungen des Untergrundes oder Feuchtigkeit.

Zusätzlich verfügt der Roboter über einen an der Plattform knapp über Bodenhöhe angebrachten umlaufenden Sensor („*Bumper*“), der eine Kollision mit einem Hindernis meldet.

2.2 Computer- und Betriebssystem

Der Roboter ist zur Zeit mit zwei Hauptplatinen, im folgenden als *Knoten* bezeichnet, ausgestattet, die jeweils mit einer Intel-Pentium CPU mit 200 MHz Taktfrequenz bestückt sind. Eine Erweiterung auf vier CPU-Platinen ist vorgesehen.

Einer der beiden Knoten ist mit zwei Framegrabber-Karten zum Einlesen der Kamerabilder ausgestattet.

Betriebssystem für den Institutsroboter ist das Echtzeit-Betriebssystem *QNX*, das sich stark an Unix anlehnt und den Posix-Standard implementiert.

2.3 Kommunikationssystem PLANET

Zur effizienten Kommunikation zwischen den Programmodulen sowohl auf einem als auch auf unterschiedlichen Knoten wurde im Rahmen einer Projektgruppe des Fachbereiches Informatik der Universität Dortmund das Kommunikationssystem PLANET für den Institutsroboter entwickelt [Alb+97].

PLANET stellt High-level-Funktionen zur effizienten Kommunikation über sogenannte Kanäle bereit, die von den verschiedenen Systemmodulen des Institutsroboters genutzt werden. PLANET-Kanäle können mehrere Sender und mehrere Empfänger haben.

PLANET startet und überwacht alle Prozesse, die den Roboter steuern, und stellt für diese einen Kommunikationspuffer zur Verfügung. PLANET startet einen Prozeß bei dessen „Absturz“ automatisch neu. Lokale Kommunikation auf einem Knoten wird automatisch über *shared memory* abgewickelt.

2.4 Systemmodule

Die Steuerungssoftware des Roboters ist nach dem Muster verhaltensgesteuerter Robotik [Bra84] modular aufgebaut. Grob können die Module in die Kategorien *Verhaltensmodule* und *Servicemodule* aufgeteilt werden. Die Verhaltensmodule sind für ein bestimmtes Verhalten des Roboters verantwortlich, die Servicemodule bekommen Aufgaben wie die Akquisition eines

Bildes oder die Fahrt zu einem Punkt von den Verhaltensmodulen zugeteilt und leisten die Sensordatenaufbereitung und direkte Hardwareansteuerung.

Eine Sonderstellung kommt der *Verhaltenskontrolle* zu, die die anderen Verhaltensmodule überwacht und das aktive Verhalten auswählt.

Alle Module sind eigenständige Prozesse; die Kommunikation der Module findet ausschließlich über PLANET-Kanäle statt.

Zu Beginn dieser Diplomarbeit waren die beiden Servicemodule *Picserver* (2.4.1) zur Akquirierung von Bildern und zur Kopfansteuerung und die *Plattformkontrolle* (2.4.2) zur Bewegung des Roboters vollständig verfügbar. Das Modul *Depthmap* (2.4.3) zur Stereobildverarbeitung wurde parallel zu dieser Diplomarbeit implementiert [Bar98].

Die Verhaltensmodule befinden sich in verschiedenen Stadien: Die *Lokalnavigation* zur Zielansteuerung unter lokaler Hindernisvermeidung steuert die vorgegebenen Ziele noch direkt ohne Hindernisvermeidung an. Um Verwechslungen mit der Lokalnavigation zu vermeiden, wird das im Rahmen dieser Diplomarbeit entwickelte Navigationsverfahren nachfolgend ausschließlich als *Globalnavigation* bezeichnet.

Die Verhaltenskontrolle befindet sich noch in der Planung, die Aufgaben der Verhaltenskontrolle werden zur Zeit von der Globalnavigation übernommen. Abbildung 2.1 veranschaulicht das aktuell implementierte Zusammenwirken der Module.

Im folgenden werden die anderen Module und ihre Aufgaben genauer beschrieben.

2.4.1 Picserver

Das Servicemodul Picserver steuert die Kameras und den Kopf des Roboters. Es nimmt von den anderen Modulen Bildanforderungen entgegen und bearbeitet sie in der Reihenfolge des Einganges.

Bildanfragen enthalten Parameter, um

- die Blickrichtung des Kopfes in Drehung und Neigung zu spezifizieren,
- den Typ der Kamera festzulegen (Fovea oder Peripherie),
- die Auflösung, den Ausschnitt und das Farbmodell der gewünschten Bilder zu bestimmen.

Ferner kann eine Bildanfrage eine *Weiterleitungsadresse* enthalten, die das Bild statt an den Absender der Anfrage an ein Servicemodul für weitere Bearbeitungsschritte lenkt. So können beispielsweise angeforderte Bilder ohne Umweg über das anfordernde Modul an das Stereoverfahren weitergeleitet werden, um einen breitbandigen Sendevorgang zu sparen.

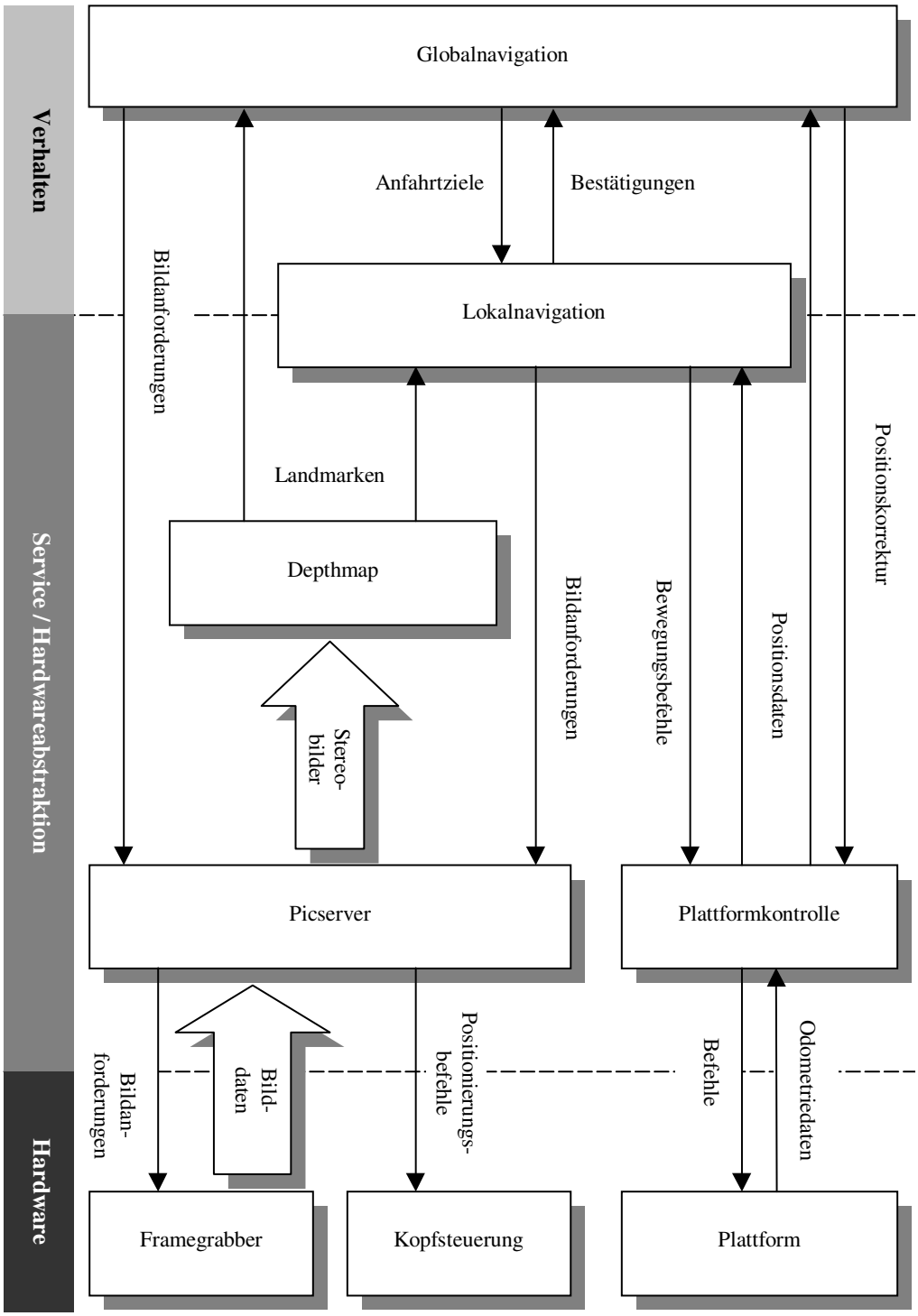


Abbildung 2.1: In dieser Diplomarbeit implementiertes Zusammenspiel der Module

2.4.2 Plattformkontrolle

Die Plattformkontrolle ist für den direkten Zugriff auf die Hardware der Plattform zuständig.

Sie akzeptiert

- Rotationsbefehle,
- Befehle zur Anfahrt an einen Punkt in Plattformkoordinaten sowie
- Befehle zum Setzen der Vorwärts- und Rotationsgeschwindigkeit.

Die Plattformkontrolle liefert ständig die Position der Plattform an alle Module, die als Empfänger in dem entsprechenden PLANET-Kanal eingetragen sind. Sie wird normalerweise von den Modulen nicht direkt angesteuert, sondern es werden Zielvorgaben an das Modul zur Lokalnavigation und Hindernisvermeidung übergeben, das die Ansteuerung der Plattform übernimmt.

2.4.3 Depthmap

Das Modul Depthmap zur Stereobildverarbeitung berechnet aus den Bildern der linken und der rechten Kamera eine Tiefenkarte.

Dieses Modul wird normalerweise nicht direkt angesteuert, sondern durch Setzen der Weiterleitungsadresse innerhalb der Bildanforderung an den Picserver. Auch die Parameter für das Stereoverfahren werden mit der Bildanforderung auf diesem Weg übermittelt.

Die Tiefenkarte wird durch das Stereoverfahren bestimmt, indem korrespondierende Kanten gesucht werden, die aufgrund der verschiedenen Positionen der Kameras auf horizontal unterschiedlichen Punkten der Bildebene abgebildet sind (wie in Abbildung 2.2 dargestellt).

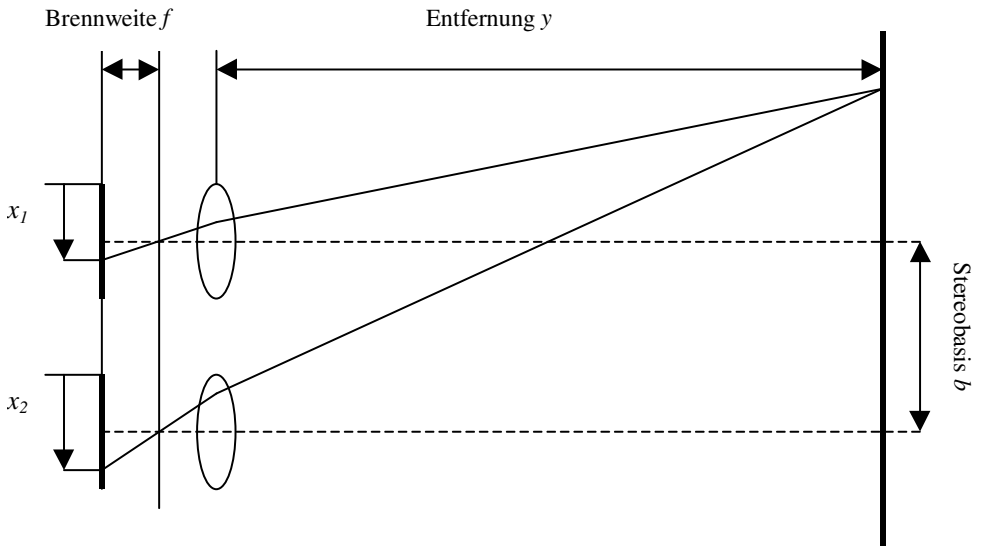


Abbildung 2.2: Schematische Darstellung des Stereosehens

Die Differenz der Positionen der Punkte in der Bildebene ist die *Disparität*, die umgekehrt proportional zur Entfernung des wahrgenommenen Merkmals ist.

Für ein Objekt, das auf beiden Kamerabildern um $d = x_2 - x_1$ Bildelemente der Breite p versetzt abgebildet wird, beträgt (nach [Jäh97]) bei im Abstand b parallel angebrachten Kameras mit Brennweite f die Entfernung y zum Objekt:

$$y = \frac{f \cdot b}{d \cdot p} \tag{2.1}$$

Die Entfernungsmessung ist mit einem überproportional mit der Entfernung steigenden Fehler behaftet.

Die vom Stereoverfahren gelieferten Punkte werden im folgenden als *Landmarken* oder *Merkmale* bezeichnet. Für eine detaillierte Beschreibung dieses Moduls sei auf [Bar98] verwiesen.

2.4.4 Lokalnavigation

Die Lokalnavigation übernimmt die Anfahrten zu einem Punkt und schickt entsprechende Bewegungskommandos an die Plattformkontrolle. Dabei berücksichtigt sie auftretende Hindernisse und fährt eventuell Umwege.

Die Lokalnavigation ist zur Zeit nicht vollständig implementiert, ein vereinfachtes Lokalnavigationsmodul steuert die vorgegebene Ziele ohne Hindernisvermeidung in dem Punkt-zu-Punkt-Modus der Plattform an.

Eine Zielvorgabe besteht aus:

- Einem Zielpunkt in dem Koordinatensystem der Plattform.
- Einer Distanz von dem Ziel, in der das anfordernde Modul benachrichtigt werden soll.
- Einer optionalen Vorgabe für die Richtung, in der das Ziel angefahren werden muß.

Zusätzlich kann zur Trajektorienplanung ein Zwischenziel mit den gleichen Parametern angegeben werden, das der Roboter zuerst ansteuert.

Die Lokalnavigation kann angewiesen werden, bei Erreichung eines Zielpunktes oder eines einstellbaren Abstandes zu einem der Zielpunkte eine Nachricht an das beauftragende Modul zu senden.

2.4.5 Verhaltenskontrolle

Die Verhaltenskontrolle soll bei Fertigstellung das Verhalten des Roboters steuern und entscheiden, welches oder welche Verhaltensmodule des Roboters zu einem Zeitpunkt aktiv sein sollen.

Zur Zeit steuert die Globalnavigation das Verhalten durch Erteilung von Aufträgen an die Bildverarbeitung und die Lokalnavigation.

Kapitel 3

Bestehende Navigationsverfahren

In diesem Kapitel wird ein Überblick über Navigationsverfahren gegeben, die von bestehenden Robotersystemen genutzt werden.

Roboternavigationssysteme lassen sich nach vielen Kriterien kategorisieren, zum Beispiel nach den verwendeten Sensoren oder danach, ob eine Karte vorgegeben wird oder nicht. An dieser Stelle wird – in Hinblick auf das Ziel der Arbeit – ein Schwerpunkt auf die Repräsentation und Selbstlokalisation gesetzt. Auf die verwendeten Sensoren und weitere Aspekte wird soweit eingegangen, wie sie eine besondere Bedeutung für das jeweils behandelte Verfahren besitzen.

Umweltrepräsentationen in bestehenden Roboternavigationssystemen lassen sich grob in graph- und bitmap-orientierte Repräsentationen unterteilen. Bei der Bitmap-Repräsentation werden in einer großen Matrix für diskretisierte Positionen in jede Zelle Umweltinformationen eingetragen. Die Graph-Repräsentationen lassen sich nach der Art der Repräsentation der Umgebung weiter in geometrische und topologische Karten unterscheiden.

Die Selbstlokalisation läßt sich in zwei Klassen unterteilen: Zum einen gibt es Verfahren zur *lokalen Rekalibration*, bei der geringe Positionsfehler des Roboters unter Ausnutzung des Wissens über die ungefähre Position korrigiert werden. Zum anderen existieren *globale Selbstlokalisationsverfahren*, bei denen kein Wissen über die Position genutzt wird („*kidnapped robot problem*“ [Eng94]). Da die Selbstlokalisationsverfahren sehr stark abhängig von der gewählten Umweltrepräsentation sind, wird im weiteren bei der entsprechenden Repräsentation auch auf die Selbstlokalisation eingegangen. Ist bei sehr abstrakten Karten meist eine genaue Rekalibration gar nicht erforderlich und eine globale Selbstlokalisation relativ einfach, so ist bei Bitmaprepräsentationen schon eine Rekalibration unter Nutzung von *Odometriedaten*[1] sehr aufwendig.

3.1 Bitmap-Repräsentation

Bitmaps sind die einfachste Repräsentation für die Umwelt des Roboters. „Zuverlässigkeitsgitter“ (Evidence Grids) gehen zurück auf Moravec und Elfes [ME85] und speichern in einer gro-

[1] Die Informationen, die der Roboter über seine Relativbewegung durch Messung der Umdrehung der Räder erhält. Nach dem griechischen Wort *hodometron*, aus *hodos* (Weg, Straße) und *metron* (messen) [Bla96], verwendet statt der wörtlichen Übersetzung des in diesem Zusammenhang im Englischen üblichen (eigentlich nautischen) Begriffes „*dead reckoning*“. Die wörtliche Übersetzung wäre „*koppeln*“ oder „*gegißtes Besteck*“: „*Bestimmung des geographischen Ortes (...) bei unklarem Wetter durch Berechnung aus der Fahrgeschwindigkeit und Kursrichtung (Koppelkurs) des Schiffs (gegißtes, geschätztes Besteck)*“ [Bro11]

ßen zweidimensionalen Matrix Wahrscheinlichkeiten, ob eine Zelle frei oder belegt ist. Diese Repräsentation wird bis heute ohne wesentliche Änderungen verwendet, Erweiterungen betreffen hauptsächlich die Operationen auf der Repräsentation [Yam96] und die Art der Einkopplung verschiedener Sensortypen zum Aufbau des Modells [Mor88].

Den aktuellen Stand dieser Technik spiegelt wohl recht eindrucksvoll der an der Universität Bonn entwickelte Roboter Rhino [BBC+95] wieder, der seine Robustheit durch einen Einsatz als Museumsführer im Haus der Geschichte in Bonn eindrucksvoll demonstrieren konnte.

In Bitmap-Repräsentationen kann bei geringer Positionsunsicherheit eine aktuelle Karte der Umgebung mit gespeicherten Daten der vermuteten Position verglichen werden. Dazu wird die aktuelle Karte verschoben und rotiert und die beste Übereinstimmung mit den gespeicherten Daten gesucht [Yam96]. Die Positionskontrolle des Roboters Rhino kann die Orientierung von gespeicherten Wänden mit der Orientierung von Wänden in den aktuellen Sensordaten vergleichen und damit den Winkelfehler minimieren.

Größte Vorteile der Bitmap-Repräsentation sind die Einfachheit und die leichte Fusion von Daten verschiedener Sensortypen, Nachteile sind der hohe Speicherverbrauch[1] und der geringe Abstraktionsgrad sowie daraus resultierende Probleme bei der Selbstlokalisation und der Pfadplanung.

Abbildung 3.1: Bitmap-Umweltrepräsentation aus [BBC+95]

3.2 Graph-Repräsentation

Gestalten sich die Bitmap-Repräsentationen recht einheitlich, können die Graph-Repräsentationen nach der Art des repräsentierten Wissens weiter unterteilt werden. So gibt es Graphen, die ähnlich wie die Bitmaps Geometrieinformationen – jedoch auf einem höheren

[1] Viele Probleme sind vergleichbar mit den Vor- und Nachteilen von Bitmap- gegenüber Vektorgrafiken in der Bildverarbeitung.

Abstraktionsgrad – speichern und Graphen, die primär Erreichbarkeit zwischen zwei Knoten des Graphen durch Kanten darstellen.

3.2.1 Geometrieinformationen

Der Hauptnachteil bitmap-orientierter Repräsentationen ist, daß die Positionsrekalibration direkt auf den Kartendaten sehr aufwendig ist. So extrahieren Schiele und Crowley [SC94] (zitiert nach [VVX96]) Liniensegmente aus den Bitmaps, um sie mit einem vorab bekannten Gebäudeplan zu vergleichen, auch bei [BBC+95] wird – wie bereits erwähnt – versucht, Orientierungen von Wänden aus den Sensor- und Kartendaten zu extrahieren.

Mit der Verfügbarkeit relativ günstiger *Laser-Range-Finder* (LRF), die im Gegensatz zu bisher hauptsächlich eingesetzten Sonarsensoren sehr genaue Messungen ermöglichen, wurden auch Karten als Graph-Struktur direkt aus Liniensegmenten (z.B. bei Einsele [Ein97]) oder Liniensegmenten und Punktmengen (bei Vandorpe et al. [VVX96]) aufgebaut.

Dabei werden aus den Sensordaten mit Liniendetektoren Liniensegmente extrahiert und in einer Graph-Struktur gespeichert.

Zur Selbstlokalisation nutzten Vandorpe et al. einen Kalmanfilter-Algorithmus [Xu95, Bal97], Einsele nutzt dynamische Programmierung [Bal97], um von verschiedenen Rundblicken des LRF den passensten zu ermitteln und so die Position global zu bestimmen.

3.2.2 Topologieinformationen

Topologische[1] Umweltrepräsentationen versuchen nicht, ein exaktes geometrisches Modell der Umwelt aufzubauen. Sie speichern die Umwelt als eine Menge von „Plätzen“ und Verbindungen zwischen diesen Plätzen. Im Gegensatz zu anderen Repräsentationen speichern sie also die freie Fahrfläche anstelle von Hindernissen, was einen niedrigeren Speicherverbrauch zur Folge hat.

Diese „qualitativen“ Karten gehen zurück auf Kuipers und Byun [KB87]. Systeme, die mit diesen Karten arbeiten, lassen sich weiter aufteilen in mit explizit bekannten oder gelernten Landmarken arbeitende Systeme und Systeme, die eigenständig Sensoreindrücke speichern und daraus eine topologische Karte aufbauen.

So stellen beispielsweise Hanebeck und Schmidt [HS96] ein Verfahren vor, um aus den aktuellen Winkeln bekannter Landmarken die Position des Roboters zu ermitteln.

Einen „Sensoreindruck“, in den neben Sonardaten auch eine aus der Odometrie ermittelte ungefähre *xy*-Position mit eingeht, benutzt Uwe Zimmer [Zim95] als direkte Eingabe für ein künstliches neuronales Netz, das aus diesen „Eindrücken“ einen Topologiegraphen aufbaut.

Einige Verfahren nutzen keine rein topologische Karte, sondern eine Mischform aus topologischen und geometrischen Informationen, wie beispielsweise Edlinger und Weiß [EW95]. Auch bei der Repräsentation von Tobias Einsele (3.2.1) handelt es sich eigentlich um eine Mischform aus topologischer und geometrischer Karte.

---

[1] *„Topologie [grch.], Teilgebiet der Mathematik, das diejenigen Eigenschaften geometr. Gebilde behandelt, die bei ‚stetigen Veränderungen‘ erhalten bleiben, z.B. die Nachbarschaftsbeziehungen zw. Punkten.“* [Bro83].

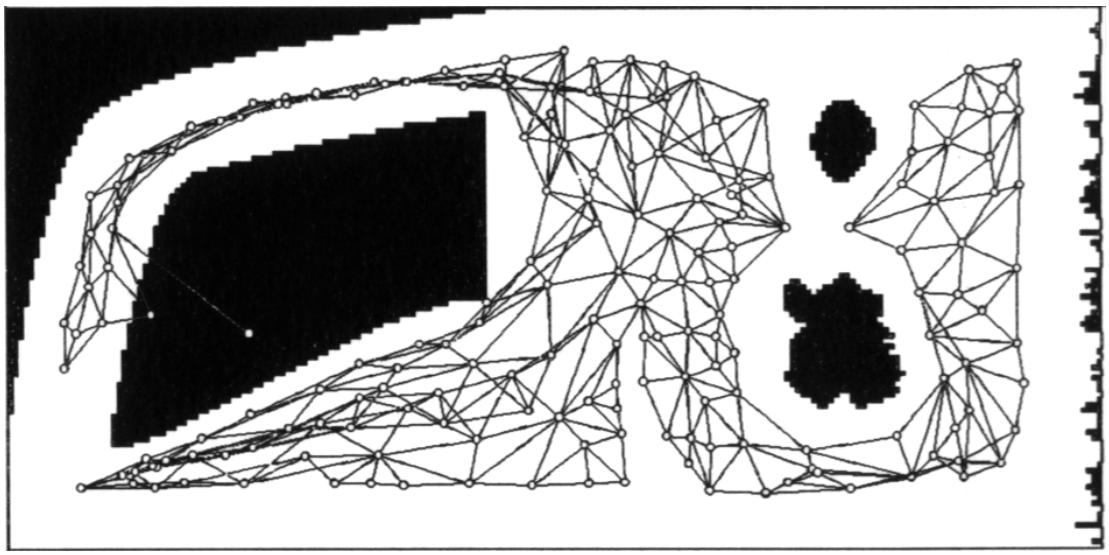


Abbildung 3.2: Topologiegraph aus [ZP94]; gespeichert werden nur die Knoten und Verbindungen.

Kapitel 4

Erste Umweltrepräsentation

Um sinnvolle Aufgaben ausführen zu können, braucht der Roboter eine Repräsentation seiner Umwelt. Die einfachste Darstellung der Umwelt ist eine Bitmap, in der gesehene Merkmale direkt eingetragen werden. Neben den in Kapitel 3 bereits dargestellten Gründen spricht hier jedoch ein weiterer Punkt gegen die Verwendung einer Bitmap als Karte:

Das in dem Modul Depthmap (2.4.3) verwendete Stereosystem liefert keine Merkmale auf Wänden ohne Textur. Diese sollen nicht als freie Fahrfläche angesehen werden, auch wenn sie in der Bitmap nicht direkt repräsentiert würden.

In einer Vorarbeit ist bereits eine Graph-Struktur für eine Umweltrepräsentation entwickelt worden, die auch in Teilen simuliert wurde. Im folgenden wird diese Umweltrepräsentation beschrieben, vervollständigt und auf dem Roboter umgesetzt.

4.1 Anforderungen

Ein sehr wesentliches Problem im Zusammenhang mit der Umweltrepräsentation ist die Rekalibration: Bei Bewegungen und Rotationen des Roboters können Abweichungen zwischen Soll- und Istwert der Roboterposition und -orientierung auftreten. Wenn dieser Fehler eine gewisse Größe überschreitet, macht es keinen Sinn mehr, die Karte zur Navigation zu benutzen oder neue Merkmale in die Karte einzutragen.

Bevor neue Merkmale in die Karte eingetragen werden oder wenn die geschätzte Ungenauigkeit eine Schwelle übersteigt, soll daher eine Rekalibration unter Ausnutzung des Wissens der ungefähren Position durchgeführt werden. Bei der Rekalibration wird versucht, aktuell gesehene und bereits gespeicherte Merkmale in Übereinstimmung zu bringen und aus der notwendigen Verschiebung und Rotation die tatsächliche Roboterposition zu bestimmen, wie in Abbildung 4.1 dargestellt.

Neben der Unterstützung der Pfadplanung muß also eine wesentliche Anforderung an die Kartenrepräsentation sein, effizient *Erwartungsbilder* zum Vergleich mit dem aktuell gesehenen Bild aus gespeicherten Daten mit lokal hoher Genauigkeit generieren zu können.

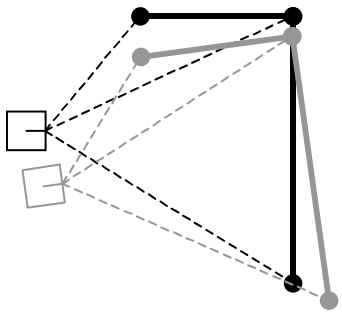


Abbildung 4.1: Anhand der vermuteten Position (grau) wird ein Erwartungsbild generiert. Durch Zuordnung der Punkte des Erwartungsbildes (grau) zu tatsächlich gesehenen Punkten (schwarz) kann die korrekte Position des Roboters (schwarz) bestimmt werden.

4.2 Vorarbeit

Die Grundlage für diese Repräsentation ist die Diplomarbeit von Daniel Tepas „*Abstraktion eines Umweltmodells für die Pfadplanung autonomer Roboter*“ [Tep97]. Daniel Tepas entwikkelt in seiner Arbeit ein Roboternavigationsverfahren und eine Simulation für Teile des Verfahrens. Die vom Stereokamerasystem gelieferten Punkte werden dabei durch Gauß-verteilte Ellipsen repräsentiert, wie in Abbildung 4.2 dargestellt. Die Ellipsen sollen eine Näherung für die Aufenthaltswahrscheinlichkeit der gemessenen Landmarken darstellen. Die Radien der Ellipse bestimmen das zweidimensionale *Vertrauensintervall* [AB74]; mit 95-prozentiger Wahrscheinlichkeit liegt die gesehene Landmarke innerhalb der Ellipse. Die Parameterisierung der Gauß-Ellipse ist im Anhang beschrieben.

Der eigentliche Aufbau der Karte läuft in den folgenden Schritten ab:

1. Der Roboter macht einen vollständigen Rundblick, das heißt, er erfaßt alle für das Stereosystem sichtbaren Punkte in seinem Umkreis.
2. Die Fehlerellipsen der erfaßten Punkte werden mit eventuell schon abgespeicherten Daten zur Übereinstimmung gebracht, um den Positionsfehler des Roboters zu minimieren.
3. Die Mittelpunkte der Ellipsen werden nach ihrem horizontalen Winkel relativ zum Roboter sortiert in eine Liste eingetragen.
4. In Bezug auf den Winkel benachbarte Ellipsenmittelpunkte werden durch eine Kante verbunden, wodurch sich ein Polygon ergibt.
5. Aus diesem Polygon werden Stützpunkte entfernt, bis es konvex ist.

Liegen hinter einer Kante des nun konvexen Polygons Punkte, so wird die Kante als *Diskontinuität*[1] markiert.

Kanten, die länger als die Roboterbreite und noch nicht als Diskontinuität markiert sind, werden auf ihre Passierbarkeit genauer untersucht, indem der Roboter eine Position in der Nähe dieser Kanten ansteuert und dort weitere Daten vom Stereosystem anfordert.

[1] Der Begriff „Diskontinuität“ wird zusammenfassend für Türen und künstliche Raumunterteilungen benutzt.

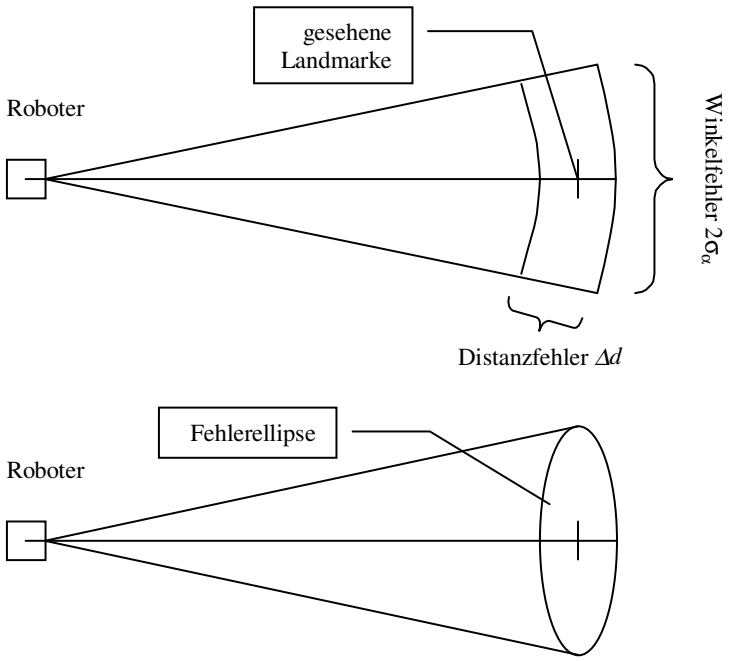


Abbildung 4.2: Näherung der Aufenthaltswahrscheinlichkeit der Landmarke durch eine Gauß-verteilte Ellipse. Die genaue Parameterisierung der Ellipse ist im Anhang beschrieben.

Die Arbeit von Daniel Tepas greift in wesentlichen Punkten auf eine Arbeit von Akio Kosaka und Avinash C. Kak zurück [KK92]. Das von Kosaka und Kak entwickelte Navigationsverfahren basiert auf monokularen Bildern, die zunächst auf Kanten untersucht werden. Die End- und Eck- oder Schnittpunkte der gefundenen Kanten werden mit einem aus einem dreidimensionalen CAD-Modell der Umgebung gewonnenen Erwartungsbild verglichen. Der Roboter kann sich durch den Vergleich der Bilder relativ schnell und genau rekalibrieren.

Der Vergleich der Punkte wurde realisiert, indem um jeden extrahierten Punkt eine aus der expliziten Fehlerrepräsentation des Roboters berechnete Gauß-Ellipse gelegt wurde. Mittels der Gauß-Ellipsen wurde dann die wahrscheinlichste Punktzuordnung ermittelt.

Daniel Tepas übernimmt in seiner Arbeit die explizite Positionsfehlerspeicherung des Roboters und einen Vergleich von Punkten mittels Gauß-Ellipsen zur Rekalibration.

Eine wesentliche Erweiterung des Verfahrens in der Arbeit von Daniel Tepas ist der Verzicht auf ein vorgegebenes Umgebungsmodell und die Generierung des Umweltmodells aus den Sensordaten des Roboters.

Weiterhin wird der Vergleich von monokularen Bildern ersetzt durch eine Tiefenkarte, wie sie vom Stereoverfahren des Institutsroboters Arnold generiert werden kann. Das Konzept der Erwartungsbilder und die explizite Fehlerrepräsentation bleiben erhalten.

4.3 Entwurf

Im folgenden werden nur die in der Arbeit von Daniel Tepas noch nicht oder nur unvollständig bearbeiteten Punkt ausführlich behandelt, insbesondere die Punkte

- Motivation für eine zweidimensionale Graphstruktur,
- Motivation für die Zerlegung des Graphen in konvexe Zellen und

- Erstellung von Erwartungsbildern aus der Karte.

Diese Aspekte behalten auch bei der erweiterten Repräsentation zum größten Teil ihre Gültigkeit.

Für den Entwurf einer Kartenrepräsentation gibt es verschiedene Kriterien wie Speicherbedarf, Fehlertoleranz oder Komplexität der auf der Karte benötigten Operationen zur Navigation. So ist beispielsweise zu entscheiden, ob eine zweidimensionale Repräsentation ausreicht oder eine dreidimensionale Karte aufgebaut werden muß und welches Koordinatensystem verwendet werden soll. Diese Fragen werden im folgenden betrachtet.

4.3.1 Dimensionalität

Für die Pfadplanung ist nur von Bedeutung, ob ein Punkt im Raum befahrbar ist oder nicht. Die Pfadplanung erfordert also nur eine zweidimensionale Karte. Soll der Roboter Fahrstühle benutzen können, ist immer noch ein Satz von zweidimensionalen Karten ausreichend. Treppen oder Rampen müßten gesondert repräsentiert werden, werden aber hier nicht betrachtet.

Für die Orientierung soll die Karte nur sehr stabile Umweltmerkmale aufnehmen, insbesondere Kanten von Wänden, Schränken oder Türen. Da diese Merkmale die gesamte Höhe einnehmen, die für den Roboter von Bedeutung ist, ist auch für die Orientierung eine zweidimensionale Karte ausreichend.

Führen im praktischen Einsatz überblickbare Hindernisse wie Tische zu Problemen bei der Orientierung, die sich mit der einfachen zweidimensionalen Karte oder einer zusätzlich zu implementierenden Objekterkennung nicht lösen lassen, muß das System auf eine zweieinhalbdimensionale Repräsentation erweitert werden. Bei einer zweieinhalbdimensionalen Repräsentation wird zu den Merkmalen noch eine maximale Höhe gespeichert, so daß auch überblickbare Merkmale repräsentiert werden könnten, ohne Merkmale im Hintergrund vollständig zu verdekken.

4.3.2 Koordinatensystem

Die einfachste Möglichkeit eines Koordinatensystems ist ein globales Koordinatensystem, das einen beliebigen Ursprung, beispielsweise den ersten Roboterstandpunkt, hat.

Allerdings ist in einem globalen Koordinatensystem die Fehlerrepräsentation problematisch: Eine Protokollierung des maximalen summierten Fehlers würde zu einem Fehler proportional zum Abstand vom Ursprung führen.

Sinnvoll wäre eine jeweils lokal hohe Genauigkeit unabhängig vom Standort, um beispielsweise die Breite und damit Passierbarkeit von Türen sicher eintragen zu können; für die Pfadplanung dagegen ist eine höhere Abweichung tolerierbar.

Die Karte wird daher in Zellen zerlegt, die ein lokales Koordinatensystem besitzen und untereinander zur Gesamtkarte verbunden sind. Da Verbindungsdaten sich immer nur auf Nachbarzellen beziehen, erhält man lokal eine hohe Genauigkeit; erst über mehrere Zellen hinweg summiert sich der Fehler.

Werden intern Daten aus Nachbarzellen benötigt, wird durch Breitensuche sichergestellt, daß die Genauigkeit so hoch wie möglich bleibt: Mit einer Breitensuche werden zu jeder Zelle so

wenig Zellübergänge wie möglich benutzt. Bei Verwendung eines Tiefendurchlaufs würde der gesamte Fehler auf einem eventuell indirekten Weg die Genauigkeit vermindern.

4.3.3 Konvexe Zellen

Natürlich muß es ein Kriterium für die Aufteilung der Karte in die geforderten Zellen geben. Sinnvoll ist ein Kriterium, das auch die anderen Operationen auf der Karte erleichtert.

Daniel Tepas verwendet in seiner Simulation eine Aufteilung in konvexe[1] Teilräume. Er begründet die Zerlegung mit der Notwendigkeit, in einer Zelle möglichst Verbindungen zu allen sichtbaren Landmarken speichern zu können und den Verwaltungsaufwand in Grenzen zu halten.

Die Benutzung konvexer Zellen bietet jedoch noch weitere Vorteile:

- Das Sichtbarkeitsproblem für Merkmale in Nachbarzellen gestaltet sich relativ einfach, da innerhalb einer konvexen Zelle keine Landmarken verdeckt werden können, wie in Abbildung 4.3 dargestellt.
- Der Algorithmus zur Pfadbestimmung kann zur Ermittlung der Passierbarkeit und Abstandsermittlung die Konvexität der Zellen ausnutzen.

Bei Generierung wird die Karte also so aufgeteilt, daß sie ausschließlich aus konvexen Zellen besteht.

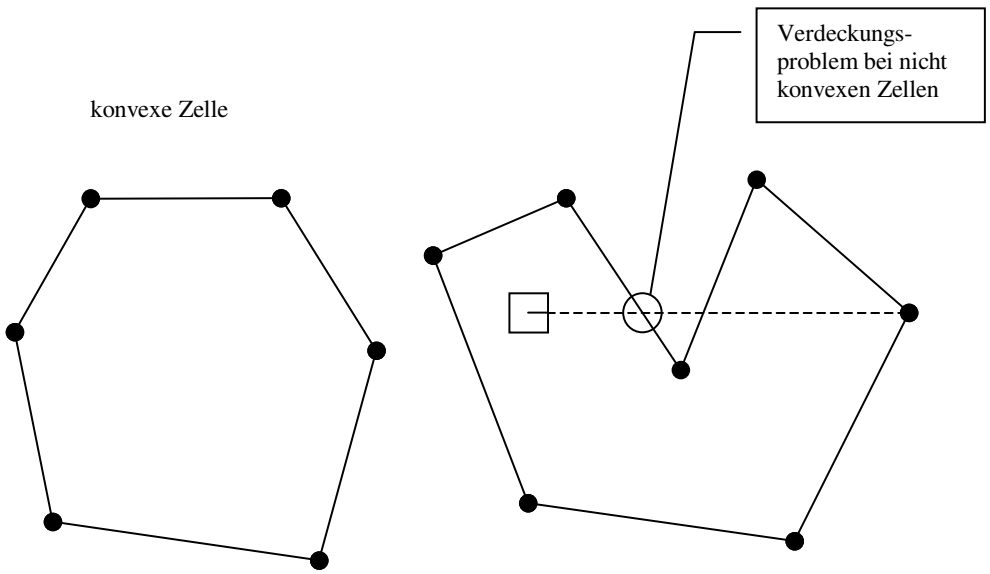


Abbildung 4.3: In konvexen Zellen können keine Merkmale verdeckt werden

[1] Ein Polygon ist konvex, wenn alle Winkel (gemessen als Änderung der Kantenrichtung, wie in Abbildung 4.7 dargestellt) positiv sind.

4.3.4 Knoten und Kanten

Der Stereoalgorithmus liefert eine Menge von Landmarken in der Umgebung des Roboters. Diese Punkte werden im Graph als Knoten (im folgenden auch als Ecken bezeichnet) gespeichert.

Ordnet man diese Knoten nach dem Winkel, so ergeben sich Kanten implizit als Verbindung zweier benachbarter Knoten.

Die Kanten können dabei zunächst eine Wand darstellen oder ein Durchgang zu einer anderen Zelle, in der von dem aktuellen Standpunkt noch keine Landmarken sichtbar waren.

Nach dem Vergleich mit dem Erwartungsbild und Sicherstellung der Konvexität können die Kanten auch „richtige“ Verbindungen zu Nachbarzellen sein.

Abbildung 4.4 illustriert das entwickelte Graphmodell.

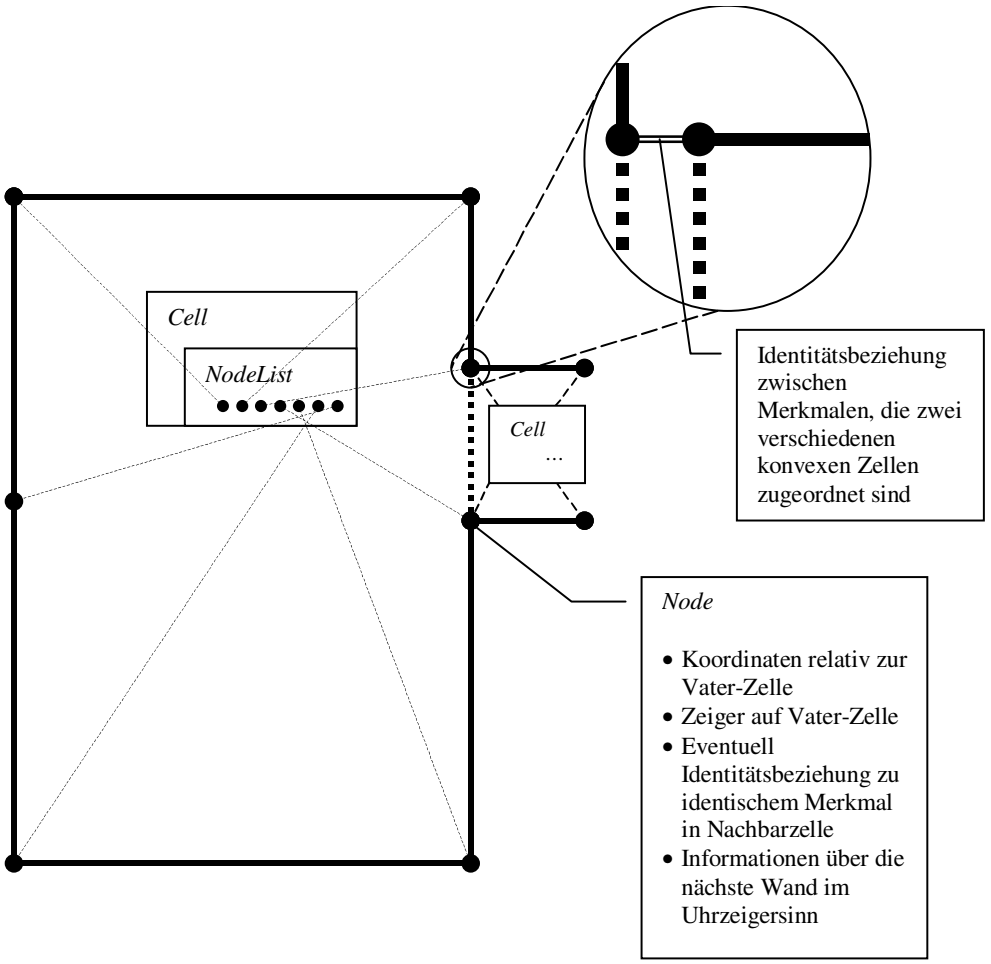


Abbildung 4.4: Darstellung des Graphmodells anhand eines einfachen Raumes, der rechts mit einem Gang verbunden ist.

4.4 Implementierung

Besonderer Wert bei der Implementierung dieses Moduls und der anderen Programmteile wurde auf einen sauberen objektorientierten Entwurf gelegt. Dabei stand eine übersichtliche Programmstruktur im Vordergrund, insbesondere eine klare Zuordnung der Methoden zu den entsprechenden Datenstrukturen.

Dieses und die anderen Module der Globalnavigation bis auf die Visualisierung sind in C++ implementiert, im folgenden wird jedoch von der verwendeten Programmiersprache abstrahiert, nicht jedoch von der Verwendung einer objektorientierten prozeduralen Sprache. Die Darstellung von Pseudocodes orientiert sich dabei an [Güt92] mit der Erweiterung, daß „Records“ neben Variablen auch Methoden enthalten können. In den Pseudocodes und im Text werden Bezeichner kursiv dargestellt; Bezeichner für Klassen und Konstanten werden groß geschrieben. Die Belegung der Konstanten ist im Anhang B aufgeführt.

4.4.1 Datenstrukturen

Die Einteilung der Klassen orientiert sich direkt an dem entworfenen Graphmodell für die Karte: Ein Objekt vom Typ *Graph,* das eine Liste aller Zellen vom Typ *Cell* enthält, speichert die Karte der Globalnavigation. Die Zellen wiederum speichern in einem Objekt vom Typ *NodeList* eine nach Winkeln sortierte Liste der Ecken, die diese Zelle definieren. Wände und Verbindungen zwischen den Zellen werden nicht explizit gespeichert, sondern als Merkmale bei den Ekken. Die Punkte des Erwartungsbildes werden in einer speziellen Klasse *Polar* gespeichert, die die Polardarstellung des Merkmales relativ zum Roboter und eine Referenz zum Merkmal selbst enthält.

Zusätzlich werden die Hilfsklassen *Point*, *Polar* und *Line* definiert, die den Umgang mit Punkten und Strecken im zweidimensionalen Raum erleichtern.

Klasse Point

Die Klasse *Point* dient dem einfacheren Umgang mit Punkten im zweidimensionalen Raum. Sie speichert die beiden Koordinaten des Punktes im kartesischen Koordinatensystem und stellt Methoden zur Verfügung, um Punkte zu verschieben, Winkel und Abstände zwischen Punkten zu berechnen, Punkte zu skalieren und um beliebige andere Punkte zu rotieren.

Klasse Line

Die Klasse *Line* definiert durch zwei Variablen *a* und *b* vom Typ *Point* eine (gerichtete) Strecke und stellt Methoden zur Verfügung, um Schnittpunkte von Strecken zu ermitteln, Strecken zu verschieben und Abstände zwischen Punkten und Strecken zu bestimmen. Außerdem kann für einen Punkt ermittelt werden, auf welcher Seite einer gerichteten Strecke er liegt.

Klasse Graph

In einem Objekt vom Typ Graph werden alle bekannten Zellen gespeichert.

Klasse Cell

Die Klasse *Cell* speichert alle Informationen einer konvexen Zelle, insbesondere in der Variablen *nodes* vom Typ *NodeList* die Liste der Knoten der Zelle. Bei der Konstruktion einer Zelle kann diese solange nicht konvex sein, bis der Roboter sich bewegt. Die Zelle speichert ihre zuletzt ermittelte globale Position in der Variablen *origin*.

Klasse NodeList

Die Klasse *NodeList* speichert einen Vektor von Zeigern auf *Nodes*.

Um einen Durchlauf durch alle nach Winkeln zum Zellenursprung sortierten Knoten von einem beliebigen Startknoten zu erleichtern, führt ein Zugriff über die Vektorgrenzen hinaus nicht zu einem Fehler, sondern der Index wird modulo Vektorgröße auf einen gültigen Wert umgerechnet.

Mit dem gültigen Index wird dann die entsprechende Funktion der Superklasse aufgerufen.

Klasse Node

Die Klasse *Node* ist ein Erbe der Klasse *Point* und speichert die vom Modul Depthmap gelieferten Stützpunkte der konvexen Zellen. Die Koordinaten des Punktes bestimmen dabei die Position relativ zum Ursprung der Zelle.

Zwei benachbarte *Nodes* speichern implizit eine Wand, eine unsicher Kante oder eine Verbindung zu einer Nachbarzelle. Bei dem ersten der beiden Knoten einer Wand im Uhrzeigersinn werden Informationen abgelegt, wie weit die Strecke zwischen den beiden Knoten untersucht wurde und ob diese bereits Blick- oder Bewegungsziel war. Diese Informationen werden später für das Verhalten des Roboters benötigt (siehe Kapitel 1).

Verbindungen werden explizit nur als Identitätsbeziehung zwischen den Ecken verschiedener Zellen gespeichert. Zwei Zellen sind genau dann verbunden, wenn zwei benachbarte Ecken eine Identitätsverbindung zu zwei Ecken der gleichen Nachbarzelle haben.

Klasse Polar

Die Klasse *Polar* wird bei der Generierung von Erwartungsbildern und dem anschließenden Vergleich verwendet. Sie speichert die vermutete Position der Landmarken relativ zur aktuellen Roboterposition und -richtung in Polarkoordinaten sowie eine Referenz auf die Landmarke selbst.

4.4.2 Positionsaktualisierung

Die Globalnavigation besitzt ein eigenes kalibriertes Koordinatensystem, das im Gegensatz zum internen Koordinatensystem der Plattform bei jeder Rekalibration korrigiert wird.

Zwischen den Rekalibrationen wird die Information über die interne Plattformposition genutzt, um aus Relativbewegungen von der letzten Plattformposition die aktuelle Position im kalibrierten Koordinatensystem der Globalnavigation zu berechnen.

Weicht die Position von der letzten Position ab, so wird vor der Änderung der internen Position ein „Bewegungsmelder“ aufgerufen:

Da die Roboterbewegung nicht nur von der Globalnavigation, sondern auch von anderen Modulen kontrolliert werden soll, kann sich die Roboterposition jederzeit ändern.

Bei einer Bewegung muß die Konvexität aller Zellen sichergestellt werden, da von einem geänderten Standpunkt des Roboters aus die Winkelsortierung der Landmarken nur für konvexe Zellen erhalten bleibt. So wird verhindert, daß Landmarken zu Indikatoren für eine falsche Position einer Verbindung („Tür“) zwischen zwei Zellen werden können, wie in Abbildung 4.5 dargestellt.

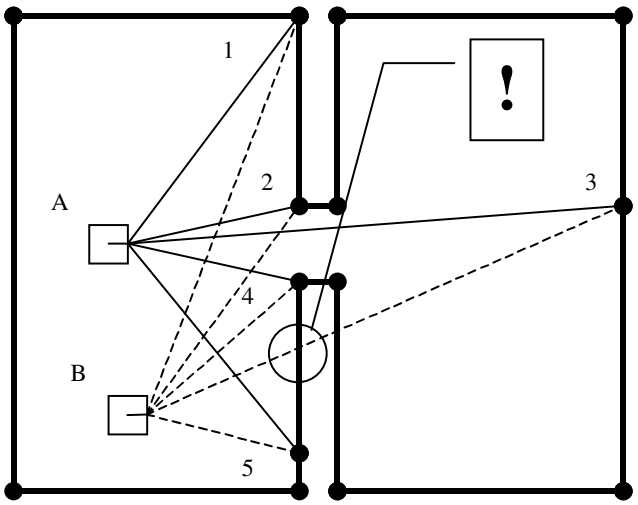


Abbildung 4.5: Vor der Bewegung impliziert die Sichtbarkeit der Landmarke 3 von Standpunkt A aus eine korrekte Diskontinuität zwischen den Landmarken 4 und 5. Wird vor der Bewegung die Konvexität nicht hergestellt, so würde nach der Bewegung die Sichtbarkeit der Landmarke 3 eine falsche Diskontinuität zwischen den Landmarken 4 und 5 implizieren.

4.4.3 Rekalibration

Die Rekalibration wird in zwei Schritten durchgeführt: Zuerst wird ein Erwartungsbild generiert, dann wird das Erwartungsbild mit den aktuell gesehenen Merkmalen verglichen und die Position anhand der Abweichung kalibriert.

Generierung eines Erwartungsbildes

Zur Rekalibration bei geringen Positionsfehlern vor dem Einfügen neuer Daten wird das Bild aus dem Graphen generiert, das der Roboter sehen würde, wenn er genau an der vermuteten Position in dem entsprechenden Winkel stünde.

Dieses Erwartungsbild wird dann in der Rekalibration mit dem Ergebnis der Bilddatenanalyse verglichen, und die Position wird entsprechend korrigiert.

Zur Generierung der Erwartungsbilder werden alle Merkmale der aktuellen Zelle für den gesehenen Winkelbereich in eine Polardarstellung transferiert. In dem polaren Koordinatensystem für den Vergleich ist die vermutete Roboterposition der Ursprung und die aktuelle Roboterrichtung auf null Grad normiert. Diese transferierten Merkmale werden dann in die Datenstruk-

tur für das Erwartungsbild übernommen. Da die Zellen im Graphen konvex sind, sind von jedem Punkt aus alle Merkmale sichtbar, die in dem Winkelbereich liegen; es gibt keine verdeckten Kanten.

Liegt in dem Winkelbereich ein Übergang zu einer anderen Zelle, so wird auf diese Zelle der Algorithmus rekursiv angewendet, allerdings mit dem Winkelbereich der Verbindung der Zellen, falls dieser geringer ist als der gesehene Winkelbereich.

Abbildung 4.6 veranschaulicht das Vorgehen des Algorithmus an einem Beispiel.

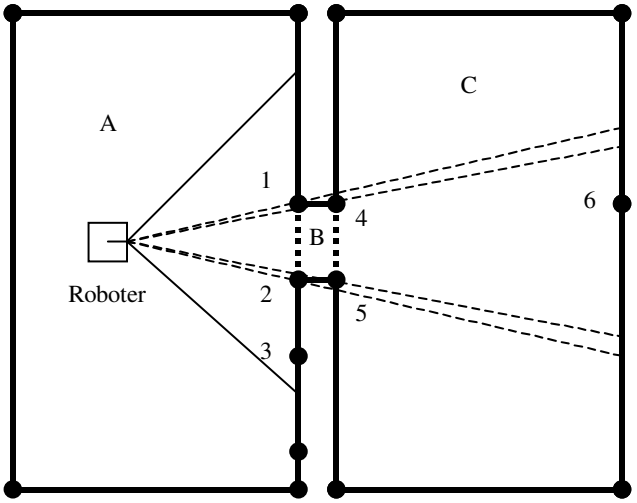


Abbildung 4.6: In der Zelle A, in der sich der Roboter befindet, werden die Landmarken 1, 2 und 3 gesehen. Aufgrund der Diskontinuitätsverbindung zu Zelle B wird die Methode rekursiv mit dem durch die Verbindung beschränkten Winkelbereich auf Zelle B angewendet. Landmarken 4 und 5 werden hinzugefügt, Landmarken 1 und 2 sind schon im Erwartungsbild. Aufgrund der Diskontinuität zwischen 4 und 5 wird der Algorithmus mit wiederum verkleinertem Winkelbereich auf Zelle C angewendet, wo schließlich Landmarke 6 gefunden wird.

---

procedure *Cell.getExpectation* (*position*: *Point*; *viewAngle*, *minAngle*, *maxAngle*: *Real*;
var *Expectation*: **Set of** *Polar*);

// sichtbare Punkte dieser Zelle ermitteln

for *i* := 0 **to** *nodes.count* – 1 **do**

// Polarkoordinaten relativ zum Roboter und zur Blickrichtung bestimmen

p := *Polar.create* (*nodes* [*i*], *pos*);
p.angle := *normAngle* (*p.angle* – *viewAngle*);

// Falls Knoten sichtbar ist, in das Erwartungsbild aufnehmen

if *p.angle* ≥ *minAngle* ∧ *p.angle* ≤ *maxAngle* then
expectation := *expectation* ∪ {*p*};
fi

od

// Verbindungen zu anderen Zellen im Sichtfeld untersuchen:
// Alle Knoten der Zelle durchlaufen

for *i* := 0 **to** *nodes.count* – 1 **do**

// eingeschränkten Winkelbereich zwischen Knoten *i* und *i*+1 berechnen

left := max (*normAngle* (*position.angleTo* (*nodes* [*i*]) – *viewAngle*), *minAngle*);
right := min (*normAngle* (*position.angleTo* (*nodes* [*i*+1])) – *viewAngle*), *maxAngle*);

```
        // Nachbarzelle c2 und deren relativen Ursprung aDelta ermitteln
        c2 := getNeighbour (i, aDelta);
        // rekursiver Aufruf, falls Nachbar existiert und Winkelbereich nicht leer
        if (c2 ≠ NIL) ∧ (right > left) then
            c2.getExpectation (position – aDelta, viewAngle, left, right, expectation);
        fi;
    od;
    return expectation.
```

Algorithmus 4.1: Bestimmung des Erwartungsbildes

Das in Algorithmus 4.1 verwendete Einfügen eines Elementes in die Menge kann in $O(1)$ realisiert werden, wenn bei dem ursprünglichen Punkt selbst vermerkt wird, ob er schon in der Menge enthalten ist oder nicht. Pro Zelle beträgt der Gesamtaufwand somit für n Knoten $O(n)$.

Da weitere Zellen rekursiv besucht werden, falls Diskontinuitäten im Sichtbereich bestehen, muß auch die Anzahl der besuchten Zellen betrachtet werden.

Diese Anzahl der insgesamt besuchten Zellen kann auf verschiedene Weise beschränkt werden, so macht es keinen Sinn, Punkte aufzunehmen, die mehr als 10 oder 20 m entfernt sind.

Eine für die Größenordnung der maximalen Gesamtlaufzeit relevante Beschränkung wäre, einen minimalen Blickbereich festzusetzen. Die Laufzeit für solch einen Blickbereich wäre beschränkt durch die Gesamtanzahl der Punkte im Graphen: Der minimale Blickbereich kann nicht weiter aufgeteilt werden, *getExpectation* kann jeweils maximal einen weiteren rekursiven Aufruf von *getExpectation* nach sich ziehen. Durch die Geometrie ist sichergestellt, daß keine Zelle doppelt betrachtet wird. Der gesamte Blickbereich ist maximal 360°, es käme also insgesamt maximal der konstante Faktor 360° / minimaler Winkel hinzu, wodurch die Laufzeit linear zur Anzahl der Punkte im Graphen insgesamt wäre.

Vergleich

Zur Rekalibration wird das Erwartungsbild mit der ebenfalls in entsprechende Polarkoordinaten transferierten Ausgabe der Bildanalyse verglichen.

Übereinstimmende Kanten werden markiert, um zu verhindern, daß Kanten doppelt in die Karte eingetragen werden. Die vermutete Position des Roboters wird dann um die Abweichung der Bilder korrigiert.

Das Verfahren zur Korrektur der Position ist ausführlich in [Tep97] beschrieben.

4.4.4 Einfügen neuer Daten

Erst nach der Rekalibration werden eventuell neue oder korrigierte Daten aus der Bildanalyse in den Graphen übernommen.

In die Datenstruktur müssen nur Landmarken eingefügt werden, die nicht bei der Rekalibration einer bereits gespeicherten Landmarke zugeordnet wurden.

Bei dem Einfügen neuer Daten ist erlaubt, daß die Konvexitätsbedingung kurzfristig verletzt wird, die Konvexität wird erst bei einer Bewegung des Roboters sichergestellt.

4.4.5 Sicherstellung der Konvexität

Daniel Tepas [Tep97] verwendet in seiner Navigationssimulation zur Sicherstellung der Konvexität eine Heuristik zur Zerlegung in konvexe Teilräume. Der Algorithmus klassifiziert alle Winkel größer als 45° als *innere* und alle Winkel kleiner –35° als *äußere Winkel* (siehe Abbildung 4.7).

Die Unterteilung in konvexe Teilräume wird vorgenommen, indem zu einem äußeren Winkel eine Ecke gesucht wird, die eine möglichst „schöne“ Zerteilung des Raumes ermöglicht. Diese Heuristik hat zwei Probleme:

1. Es lassen sich mit Winkeln zwischen 0 und –35° nicht konvexe Räume konstruieren, die die Heuristik nicht als konvex erkennt und folglich nicht unterteilt. Dieses Problem läßt sich lösen, indem man alle negativen Winkel als äußere Ecken betrachtet.
2. Das Kriterium zur Auswahl der zweiten Ecke kann bei komplexeren als in der Simulation verwendeten Räumen zu Überschneidungen der Zellgrenzen führen.

Ein deterministischer Algorithmus zur Sicherstellung der Konvexität besteht darin, von der aktuellen Zelle solange geeignete Dreiecke abzuspalten, bis die Zelle selbst konvex ist. Die abgespaltenen Bereiche sind automatisch konvex, da Dreiecke immer konvex sind. Die Zelle selbst ist spätestens dann konvex, wenn sie selbst ein Dreieck ist. Der Algorithmus ist somit korrekt und vollständig.

Um größere Zellen zu erhalten, wird zunächst versucht, eine Heuristik zu verwenden, die im wesentlichen eine Verbesserung des Verfahrens von Daniel Tepas darstellt. Wird dabei eine Überschneidung von Zellgrenzen erkannt, so wird auf den deterministischen Algorithmus zurückgegriffen.

Der verwendete Algorithmus läßt sich in vier Teile gliedern, die im Folgenden betrachtet werden:

1. Entdeckung von Verletzungen der Konvexitätsbedingung
2. Eliminierung einer „nicht konvexen“ Ecke durch Suche einer geeigneten Zerlegung der Zelle
3. Zerlegung der Zelle in zwei Teilzellen

Abschließend wird noch auf die Laufzeit des Gesamtalgorithmus eingegangen.

Verletzungen der Konvexitätsbedingung

Der Algorithmus überprüft zuerst, ob eine Veränderung an der Zelle seit der letzten Konvexitätsüberprüfung vorgenommen worden ist; falls ja, wird nach einer Verletzung der Konvexitätsbedingung, also einem in die Zelle hineinzeigenden Winkel gesucht. Dieser Winkel wird dann von der Methode *eliminate* (Algorithmus 4.2) eliminiert. Da *eliminate* beide resultierenden Teilräume überprüft, kann die Schleife nach dem Aufruf von *eliminate* verlassen werden.

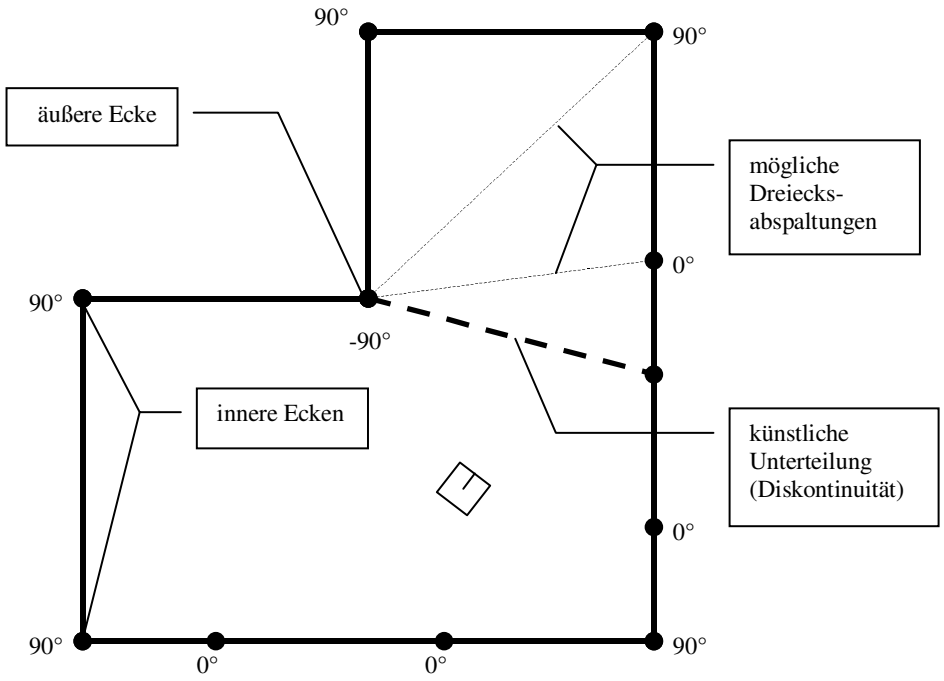


Abbildung 4.7: Künstliche Unterteilung eines Raumes in konvexe Zellen

Suche einer geeigneten Zerlegung der Zelle

Ist eine Ecke gefunden worden, die die Konvexitätsbedingung verletzt, sucht die aufgerufene Methode *eliminate* einen zweiten Knoten zur Zerlegung der Zelle an der Verbindungsstrecke der beiden Knoten.. Der zweite Knoten wird dabei so gewählt, daß die Verbindungsstrecke möglichst kurz ist und der Winkel an der bisher nicht konvexen Ecke so zerlegt wird, daß die beiden entstehenden Winkel die Konvexitätsbedingung erfüllen.

Die Methode *split*, die die eigentliche Zerlegung vornimmt, überprüft, ob die Verbindungsstrekke der beiden Ecken eine andere Strecke der Zelle schneidet. In diesem Fall wird keine Zerlegung vorgenommen und *false* als Rückgabewert an *eliminate* übergeben. *eliminate* spaltet dann mit *split* einfach nur ein Dreieck ab.

Die beiden resultierenden Zellen werden innerhalb von *split* erneut durch die Methode *checkConvexity* auf ihre Konvexität überprüft.

---

procedure *Cell.eliminate* (*index*: *Integer*; *robotPosition*: *Point*);

// zwei Indexvariablen initialisieren, die die Knoten der Zelle im und gegen den
// Uhrzeigersinn durchlaufen

k := (*index* + 2) modulo *nodes.count*;
i := (*index* – 2) modulo *nodes.count*;

// speichern, an welchen Positionen der Durchlauf der Knoten vollständig ist

borderK := *i*;
borderI := *k*;

// vom übergebenen Index im Uhrzeigersinn Ecke suchen, so daß der Winkel OK ist

```
while k ≠ borderK ∧ winkel (index – 1, index, k) < 0 do
    k := (k + 1) modulo nodes.count;
od;

// vom übergebenen Index gegen den Uhrzeigersinn Ecke suchen,
// so daß der Winkel OK ist

while i ≠ borderI ∧ getAngle (i, index, index + 1) < 0 do
    i := (i – 1) modulo nodes.count;
od;

// falls ein gültiger Bereich gefunden wurde, kürzeste Verbindung suchen

if i ≠ borderI ∧ k ≠ borderK then
    best := ∞;
    while k modulo nodes.count ≠ (i + 1) modulo nodes.count do
        d := nodes [index].getDistance (nodes [k]);
        if (d < best) then
            splitIndex := k;
            best := d;
        fi
        k := k + 1;
    od

// Teilungsversuch; falls Zerlegung ungültig, nur ein Dreieck abspalten

if ¬split (pos, index, splitIndex) then

    // weiter entfernten Nachbarn der übergebenen Ecke ermitteln
    // und das entsprechende Dreieck abspalten

    if nodes [index + 1].getDistance > nodes [index – 1].getDistance then
        split (roboterposition, index, index + 2);
    else
        split (roboterposition, index – 2, index);
    fi
fi.
```

Algorithmus 4.2: Abspaltung einer „nicht konvexen“ Ecke

Zerlegung der Zelle

Die Methode *split* zur Zerlegung der Zelle überprüft die Verbindungsstrecke der beiden Ecken, an denen die Zelle geteilt werden soll, auf Schnittpunkte mit anderen Strecken der ursprünglichen Zelle.

Wird ein Schnittpunkt gefunden, so liefert die Methode *false* zurück und an der Zelle wird keine Veränderung vorgenommen. Wird kein Schnittpunkt gefunden, liefert die Methode *true* zurück, und die Zelle wird zwischen den beiden übergebenen Ecken in zwei Zellen geteilt, so daß die bisherige Zelle weiterhin den Punkt *robotPos* enthält.

```
function Cell.split (robotPos: Point; i, j: Integer): Boolean;

// Zuerst Indizes normieren

i := i modulo nodes.count;
j := j modulo nodes.count;
```

```
// auf Überschneidungen prüfen, dazu die Verbindungsstrecke erzeugen
test := Line.create (nodes [i], nodes [j]);
// Vergleich mit allen Kanten der aktuellen Zelle
for k := 0 to nodes.count – 1 do
    kk := (k + 1) modulo nodes.count;
    if k ≠ j ∧ k ≠ i ∧ kk ≠ i ∧ kk ≠ j then
        // Abbruch, falls die Zerlegungsstrecke eine Kante der Zelle schneidet
        if test.intersection (Line.create (nodes [k], nodes [kk])) then
            return false;
        fi;
    fi;
od;
// keine Überscheidung; Indizes ordnen
if i > j then
    swap (i, j);
fi;
// nun neue Zelle erstellen
c2 := Cell.create (pGraph);
c2.origin := origin;
// anhand der Strecke i–j kann entschieden werden, welche Zell robotPos enthält;
// anschließend werden in die neue Zelle alle Knoten jenseits dieser Strecke übertragen
if normAngle (nodes [i].angleTo (nodes [j]) – nodes [i].angleTo (robotPos)) > 0 then
    c2.addNodes (nodes [0 .. i, j .. nodes.count – 1])
    nodes := nodes [i .. j];
    link (c2.nodes [i], nodes [0]);
    link (c2.nodes [i + 1], nodes [nodes.count]);
else
    c2.addNodes (nodes [i .. j]);
    nodes := nodes [0 .. i, j .. nodes.count – 1];
    link (c2.nodes [0], nodes [i]);
    link (c2.nodes [c2.nodes.count], (nodes [i + 1]);
fi;
c2.reOrigin;
// für beide Teilzellen Konvexität durch rekursiven Aufruf von
// checkConvexity sicherstellen
c2.checkConvexity (Point.create (0,0));
checkConvexity (robotPos);
return true.
```

Algorithmus 4.3: Zerlegung einer Zelle zwischen Knoten i und j

Laufzeit

Die Laufzeiten für die Suche eines Knotens, der die Konvexitätsbedingung verletzt *(checkConvexity)* und für *eliminate* sind insgesamt linear: Beide Methoden führen hintereinander eine konstante Anzahl von Durchläufen der Knotenliste aus.

Auch die Methode *split* hat bis auf den rekursiven Aufruf von *checkConvexity* eine zur Anzahl n der Knoten der Zelle lineare Laufzeit. Im schlimmsten Fall, wenn pro Aufruf von *split* nur ein Dreieck abgespalten wird, hat einer der beiden rekursive Aufruf von *checkConvexity* noch n–1 Knoten zu überprüfen.

Somit ergibt sich insgesamt maximal eine quadratische Laufzeit für jede Zelle, in die neue Punkte eingetragen worden sind.

Kapitel 5

Erweiterte Repräsentation

Wie in Abbildung 8.1 dargestellt, konnte der Roboter in einer Simulation eine Testumgebung mit der vorgestellten Kartenimplementierung erkunden und korrekt kartieren. Bei Verfügbarkeit des Moduls Depthmap (2.4.3) führten Tests der implementierten Umweltrepräsentation mit realen Daten allerdings zunächst zu erheblichen Problemen.

Die Umweltdaten, die der Stereoalgorithmus liefert, weisen verschiedene Fehler und Ungenauigkeiten auf:

- **Fehlzuordnungen**: Durch Spiegelungen oder sich wiederholende Strukturen können Punkte des rechten und linken Kamerabildes falsch zugeordnet werden und so einen vollkommen falschen Tiefenwert für den Punkt liefern.
- **Auflösungsbedingte Ungenauigkeit**: Die Korrespondenz der Punkte wird in Pixeln gemessen, somit ist die Auflösung implizit eine Schranke für die Genauigkeit der Daten. Die Ungenauigkeit steigt dabei überproportional mit der Entfernung. Die Toleranz vertikal zusammengefaßter Punkte wird in der Datenstruktur, die der Stereoalgorithmus liefert, für jeden Punkt mit angegeben.

Diese beiden Probleme führten zu einer sehr großen Varianz der gemessenen Distanzen der Punkte in einer Wand und zur Zersplitterung der Räume in kleine Zellen, selbst wenn die Punkte innerhalb der auflösungsbedingten Ungenauigkeit so verschoben wurden, daß nach innen gerichtete Ecken möglichst vermieden wurden (siehe Abbildung 5.2). Ferner erzeugten einzelne Fehlzuordnungen falsche Kanten in der aus dem Rundblick erstellten Karte.

Allein der Vergleich der Ausdrucke zweier Rundblicke ließ zudem starke Zweifel aufkommen, daß eine Rekalibration und Kartierung basierend auf einzelnen Merkmalen möglich ist (siehe Abbildung 5.3).

5.1 Anforderungen

Die bestehende Repräsentation soll derart erweitert werden, daß sie mit den Realdaten möglichst robust und effizient funktioniert. Die aufgezeigten Vorteile einer Graph-Repräsentation sollen dabei möglichst vollständig erhalten bleiben.

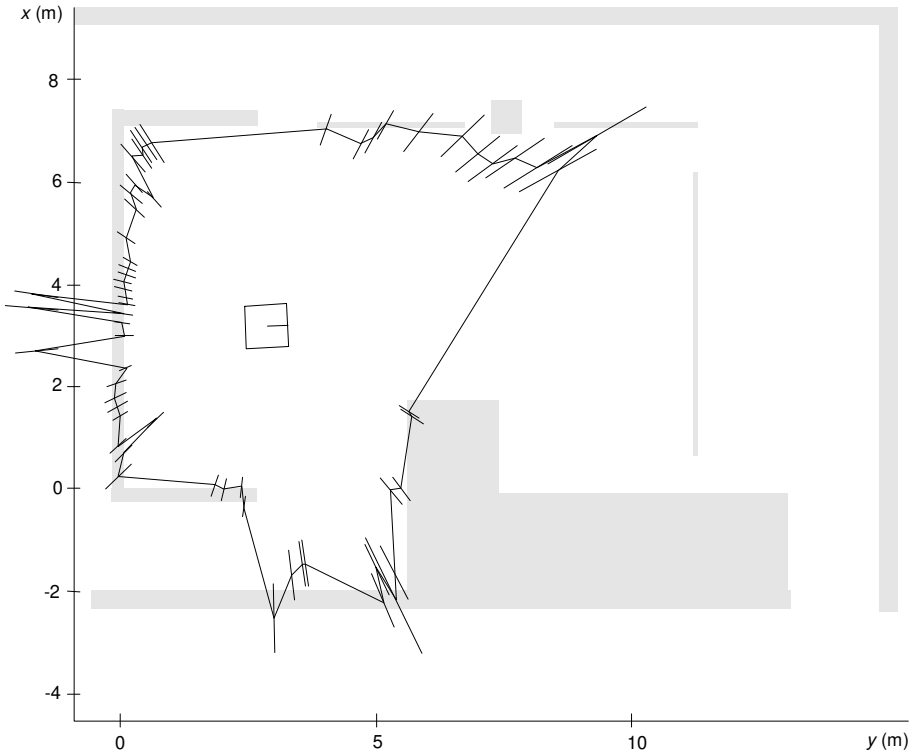


Abbildung 5.1: Realdaten vor der Zerlegung in konvexe Zellen mit eingezeichneter Abstandsunsicherheit.

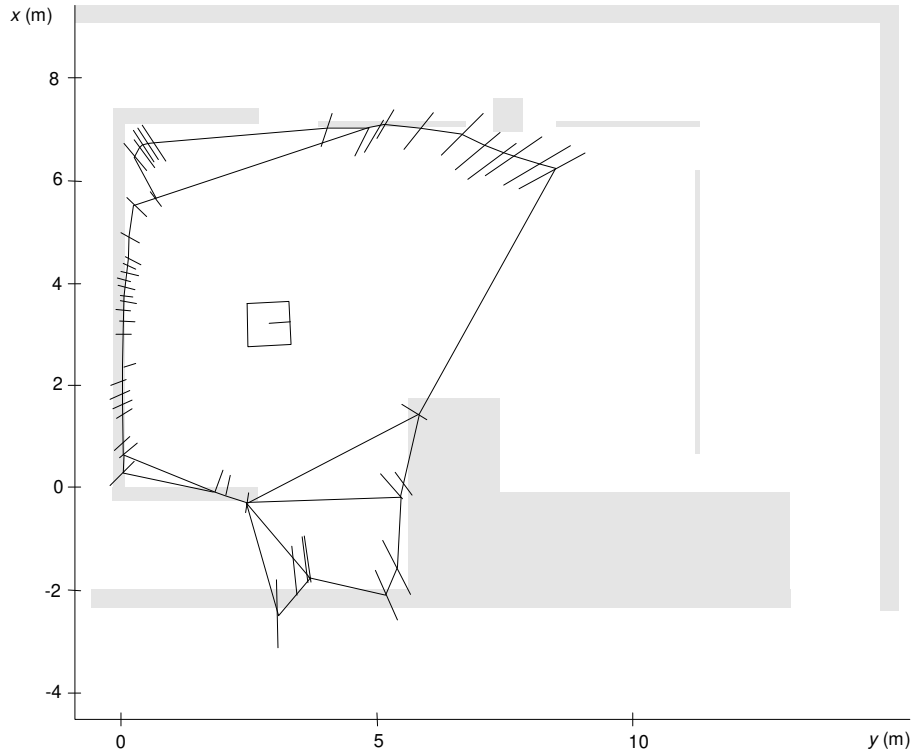


Abbildung 5.2: Realdaten nach der Zerlegung in konvexe Zellen unter Ausnutzung der Abstandsunsicherheit und Löschung von „Ausreißern“ zur Verminderung der Fragmentierung.

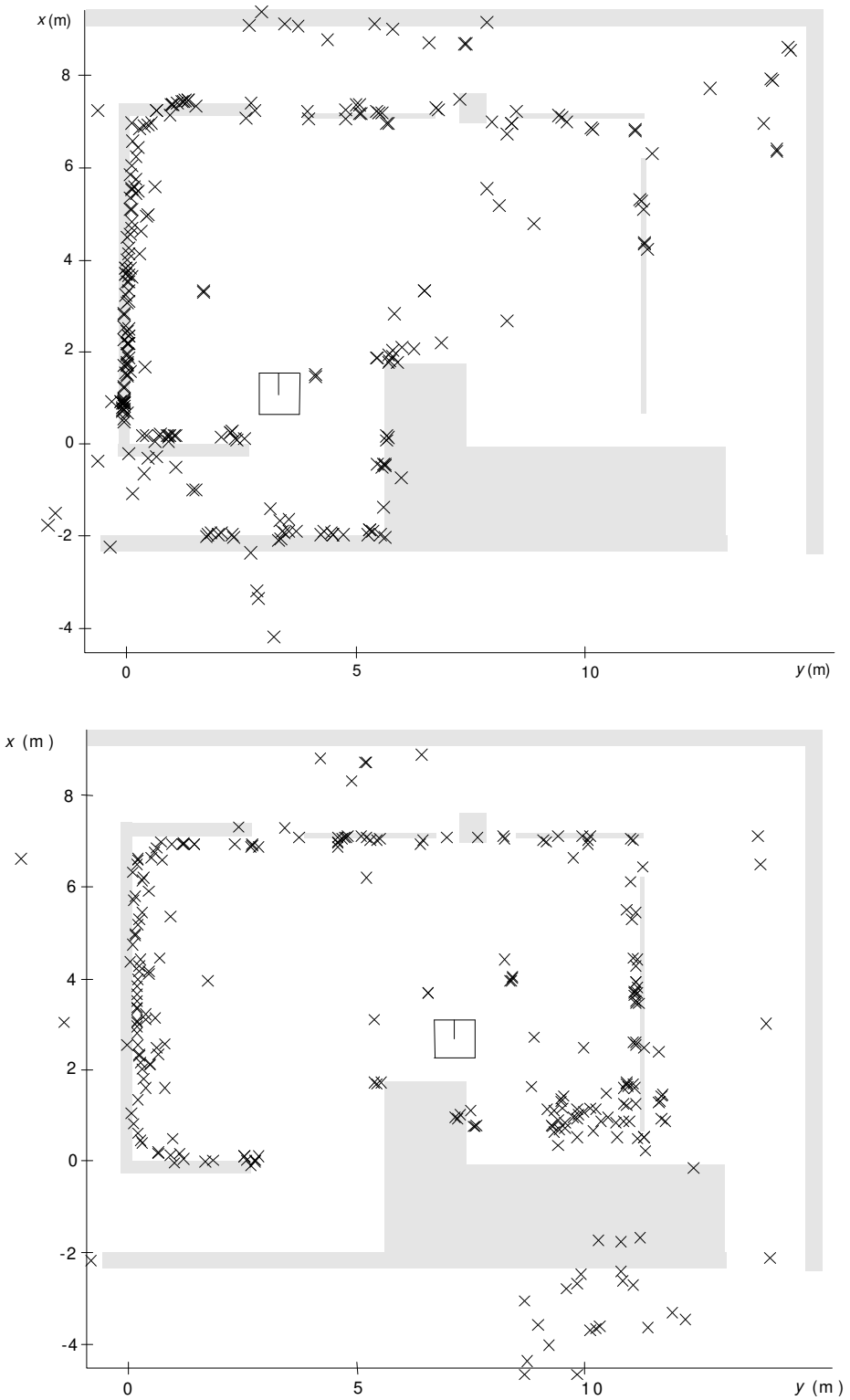


Abbildung 5.3: Vom Stereoalgorithmus von zwei verschiedenen Positionen aus gelieferte Punkte in dem Testraum. Im Gegensatz zu einzelnen Punkten sind die Wände als ganzes relativ gut wiederzuerkennen. Die Störungen im unteren Bild unten rechts werden durch dort stehende Pflanzen verursacht.

5.2 Entwurf

In Abbildung 5.3 ist deutlich zu sehen, daß die Wände im Gegensatz zu einzelnen Punkten durchaus als Geraden in beiden Rundblicken wiederzuerkennen sind.

Für die Aufgaben der Globalnavigation sind genau diese Merkmale besonders wichtig: Wände und Schränke bleiben über die Zeit normalerweise konstant und zeichnen sich insbesondere durch lineare Flächen und rechte Winkel aus; andere Hindernisse können auch von der Lokalnavigation erkannt und umfahren werden.

Navigation außerhalb von Gebäuden wird hier nicht betrachtet. Es liegt also nahe, innerhalb der vom Stereoalgorithmus gelieferten Landmarken nach Punkten zu suchen, die eine Gerade bilden und bei Mehrdeutigkeiten Geraden zu bevorzugen, die rechte Winkel bilden.

Im folgenden wird beschrieben, wie die Geraden aus den Stereodaten extrahiert werden, wie die Datenstrukturen für diese zusätzliche Abstraktionsebene erweitert werden und welche Konsequenzen sich für die verwendeten Algorithmen aus der neuen Repräsentation ergeben. Außerdem wird eine einfache Möglichkeit der globalen Selbstlokalisation auf Grundlage des neuen Umweltmodells aufgezeigt.

5.2.1 Geradendetektor

Es gibt bereits eine Vielzahl von Algorithmen zur Kantenerkennung in Bildern, die sich auch zur Bildung von Geraden aus Punktmengen eignen.

Ein leistungsfähiges und einfaches Verfahren ist die *Hough-Transformation* (siehe z.B. [Ste93]). Bei der Hough-Transformation werden alle möglichen Geraden, die innerhalb einer Diskretisierung durch einen Punkt gelegt werden können, bestimmt, wie in Abbildung 5.4 angedeutet. Für jede dieser Geraden wird in einer Matrix über Winkel und Entfernung der *Hesse-Form* der Geraden ein Zähler erhöht.

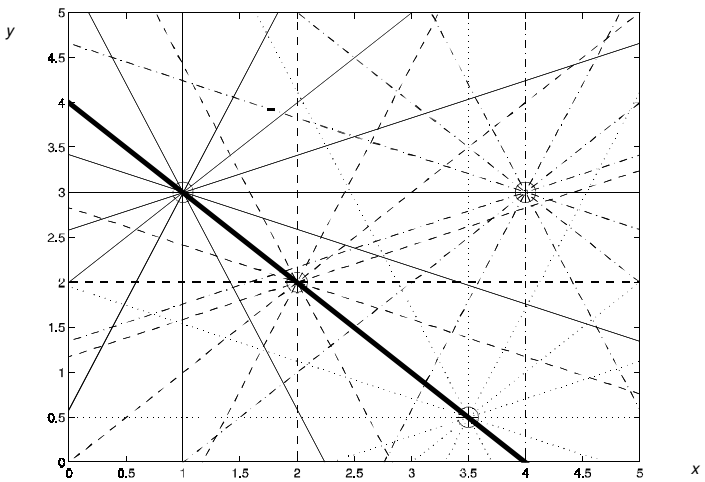


Abbildung 5.4: Bei der Hough-Transformation werden alle möglichen Geraden durch die Punkte gelegt, wie in der Grafik angedeutet.

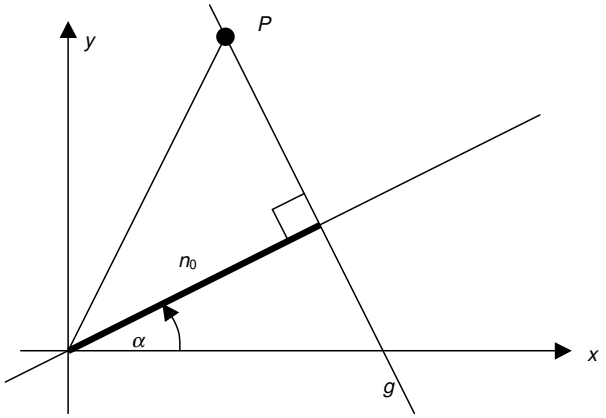


Abbildung 5.5: Hesse-Form einer Geraden (nach [AB74])

Die Hesse-Form beschreibt eine Gerade durch ihren Abstand vom Ursprung n_0 und den Winkel α einer Senkrechten zu der Geraden, wie in Abbildung 5.5 dargestellt. Dabei gilt für die Koordinaten x und y der Punkte der Geraden:

$$x \cos \alpha + y \sin \alpha - n_0 = 0; n_0 \geq 0 \tag{5.1}$$

Angenommen, der Winkel soll in 1°-Schritten diskretisiert und Abstände bis 10 m in 10 cm-Schritten erfaßt werden.

Die Hough-Matrix hat dann 360 × 100 Einträge. Um die Geraden durch einen Punkt P (x, y) in die Hough-Matrix einzutragen, müssen alle Winkel durchlaufen werden. Der entsprechende Abstand der Geraden durch P vom Ursprung ergibt sich durch Auflösung der Hesse-Form nach n_0:

$$n_0 = x \cos \alpha + y \sin \alpha \tag{5.2}$$

Insgesamt ergibt sich damit für die Eintragung eines Punktes:

```
for α := 0 to 359 do
    n0 := x cos α + y sin α
    if n0 > 0 then
        index := round (n0 / 10);
        HoughMatrix [α, index] := HoughMatrix [α, index] + 1;
    fi;
od;
```

Algorithmus 5.1: Beispiel für die Eintragung eines Punktes (x, y) in die Hough-Matrix

Nach Eintragung aller Punkte bestimmen die Maxima der Hough-Matrix die Parameter der Geraden, auf denen die meisten Punkte liegen. In Abbildung 5.6 entsprechen die Schnittpunkte den Maxima der Hough-Matrix.

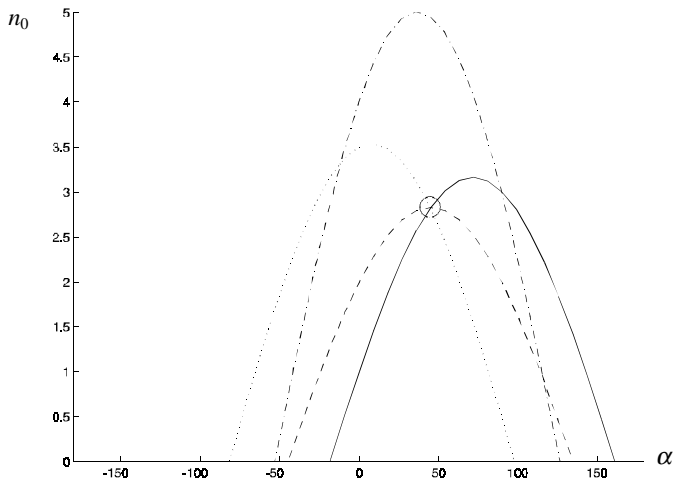


Abbildung 5.6: Hough-Matrix für die Punkte aus Abbildung 5.4. Jede Kurve entspricht einem Punkt aus Abbildung 5.4, das Muster ist dabei identisch zu dem Muster der jeweils angedeuteten Geradenmenge. Schnittpunkte der Kurven entsprechen Geraden, die durch mehrere Punkte gehen. Der Markierte Schnittpunkt entspricht der in Abbildung 5.4 markierten Gerade durch drei von vier Punkten.

Da die Größe der Hough-Matrix konstant ist, ist die Laufzeit nur linear zur Anzahl der Punkte. Dabei muß allerdings berücksichtigt werden, daß auch die konstanten 360 Iterationen pro Punkt aus dem Beispiel ein nicht zu vernachlässigender Faktor sein können.

Auch die Bestimmung der besten Geraden erfordert nur einen Durchlauf durch die gesamte Matrix, ist also auch in konstanter Laufzeit möglich. Konstante Laufzeit bedeutet für das Beispiel allerdings sogar schon 36.000 Iterationen.

Die Hough-Transformation hat gegenüber anderen Verfahren für diese Anwendung zwei wesentliche Vorteile:

1. Im Gegensatz zu anderen Verfahren (wie z.B. in [Ein97] verwendet) verfolgt die Hough-Transformation nicht lokal eine Vorzugsrichtung solange, bis die Daten zu sehr abweichen. Mehrere Geradensegmente werden also nicht als einzelne Strecken behandelt, sondern als eine Gerade erkannt, was zu einer größeren Zuverlässigkeit des Winkels führt.
2. Die Hough-Transformation benutzt nicht – wie einige andere Verfahren – die Suche in der Umgebung eines Punktes. Diese Suche läßt sich in einer Bitmap-Repräsentation der Eingabedaten leicht realisieren, nicht so gut aber in der vorhandenen Listenrepräsentation.

5.2.2 Kartenrepräsentation

Im folgenden wird beschrieben, welche Änderungen an der Umweltrepräsentation die Speicherung von Wänden erfordert und wie die Aufteilung in Zellen in dem neuen Umweltmodell vorgenommen wird.

Wände

Statt der bisherigen Speicherung von Punkten in Zellen müssen nun Geraden gespeichert werden. Grundsätzlich wäre es denkbar, dazu die bestehenden Datenstrukturen beizubehalten und künstliche Knoten an den Schnittpunkten der Geraden zu erzeugen. Allerdings würde ein derartiges Vorgehen einige Probleme mit sich bringen:

- Zwischen der Hesse- und der Punktdarstellung muß ständig konvertiert werden.
- Soll eine Gerade korrigiert werden, muß bei der Positionskorrektur der Punkte berücksichtigt werden, daß diese so verschoben werden, daß die anderen durch diese Punkte gestützten Geraden nicht verändert werden.
- Für Türen müßten die Geraden künstlich unterteilt werden, auch das würde eine Korrektur der Geraden als „Ganzes“ erschweren.

Aufgrund dieser Probleme erscheint es sinnvoll, die Repräsentation derart neu zu gestalten, daß Wände direkt in der Hesse-Form gespeichert werden.

Um ein zusätzliches Kriterium für den Vergleich zweier Geraden zu erhalten, kann bei jeder Wand ein Attributvektor mit den gefundenen Merkmalen gespeichert werden. Der Attributvektor soll folgende Informationen enthalten:

- summierte *Gewichte* der diskretisierten Stellen der Wand zugeordneten Landmarken. Das Gewicht eines Punktes aus dem Stereoverfahren gibt an, wieviele Bildpunkte einer vertikalen Kante auf diesen Punkt zusammenfallen
- einen Eintrag, ob hinter dieser Stelle der Wand andere Merkmale gesehen wurden, ob also diese Stelle passierbar sein kann
- Informationen, ob diese Stelle der Wand angefahren und aus der Nähe betrachtet wurde.

Konvexe Zellen

Das Konzept der konvexen Zellen wird aufgrund der in Kapitel 4 aufgeführten Vorteile beibehalten. Eine Zelle besteht nun statt aus einer Menge von Punkten aus einer Menge von Geraden, wobei die Attribute einer Geraden auch zu mehren Zellen gehören können.

Bei den Zellen wird auch gespeichert, welche Zellen Nachbarzellen sind. Auf die explizite Verknüpfung einer Diskontinuität mit einer Nachbarzelle wird verzichtet, da Informationen über Diskontinuitäten in dem Merkmalsvektor der Wände abgelegt werden und der Aufwand der Verwaltung der Information pro 10 cm Wand zu hoch erscheint.

5.2.3 Rekalibration

Die Rekalibration wird durch die Verwendung von Geraden erheblich vereinfacht:

- Da sehr viele Punkte zu Geraden zusammengefaßt werden, ist die Datenmenge geringer, aber zuverlässiger.
- Auf ein explizites Fehlermodell kann verzichtet werden.
- Fehlzuordnungen sind bei Geraden unwahrscheinlicher als bei Punkten, da die Geraden mehr Informationen beinhalten.

Für die Rekalibration wird zuerst ein Erwartungsbild für die aktuell vermutete Position und Ausrichtung ermittelt. Die Geraden des Erwartungsbildes werden mit den gesehenen Geraden verglichen und die ähnlichsten paarweise zugeordnet.

Mit dieser Zuordnung wird zuerst die Roboterausrichtung und danach die Position rekalibriert.

5.2.4 Globale Selbstlokalisation

Wenn der Roboter seine Position vollständig verliert, weil er beispielsweise ausgeschaltet bewegt wurde, kann kein Erwartungsbild aus dem Wissen der ungefähren Position erzeugt werden.

Trotzdem können aus den gesehenen Kanten Hypothesen für eine neue Position und Orientierung aufgestellt werden: Zwei sich schneidende gesehene Geraden werden mit allen Ecken aller gespeicherten konvexen Zellen mit ähnlichem Winkel zur Deckung gebracht. Für alle diese Hypothesen kann ein Erwartungsbild berechnet und für die beste Übereinstimmung der Rekalibrationsalgorithmus angewendet werden.

Wird keine hinreichende Übereinstimmung gefunden, muß eine neuer Plan aufgebaut werden. Die Pläne können eventuell später – wenn der Roboter eine bekannte Stelle erreicht – fusioniert werden.

Diese globale Selbstlokalisation ist insgesamt erheblich aufwendiger als eine einfache Rekalibration, ermöglicht aber eine Positionsbestimmung ohne Odometrieinformationen. Sie kann immer dann eingesetzt werden, wenn die einfache Rekalibration nicht angewendet werden kann, also bei Verlust zuverlässiger Odometrieinformationen und zur Überprüfung vermuteter Kreisschlüsse.

5.3 Implementierung

Im folgenden wird die Implementierung der erweiterten Repräsentation beschrieben. Dabei werden zuerst die Datenstrukturen und danach die Änderungen im globalen Ablauf, die Geradenextraktion, die Rekalibration und die Aktualisierung der Datenstruktur dargestellt.

5.3.1 Datenstrukturen

Die zur Implementierung genutzten Klassen orientieren sich direkt an der im Entwurf entwikkelten Repräsentation. Eine Referenz zu der aktuellen Zelle und Informationen über den Roboterstatus werden in der Klasse *GlobalNavigation* gespeichert. Die Klasse *Cell* speichert dabei nun statt einer sortierten Liste von Punkten eine sortierte Liste von Wänden. Diese Wände werden durch die neue Klasse *Wall* repräsentiert. Da eine Wand mehreren konvexen Zellen zugeteilt werden kann, werden die gemeinsamen Informationen in einer eigenen Klasse *WallAttrVector* gespeichert. Diese nimmt Einträge für diskretisierte Positionen auf der Wand vom Typ *WallAttr* auf.

Die Klasse *Accumulator* speichert Punkte vom Typ *WeightedPoint* vor der Geradenextraktion zwischen; die Template-Klasse *Matrix* realisiert die für die Hough-Transformation erforderliche Matrix.

Klasse GlobalNavigation

Die Klasse *GlobalNavigation* speichert die aktuelle Position und Orientierung des Roboters sowie eine Referenz *pCell* auf die Zelle, in der sich der Roboter gerade befindet.

Klasse Cell

Die Klasse *Cell* speichert – wie schon in der ersten Repräsentation – die Informationen, die zu einer konvexen Zelle gehören. Dies sind die Wände vom Typ *Wall* in der Variablen *walls,* der zuletzt ermittelte Ursprung der Zelle in globalen Koordinaten in der Variablen *origin* und Verbindungen zu Nachbarzellen in der Variablen *neighbours*. Eine Referenz auf die Globalnavigation wird in *pMain* gespeichert. Bei Sicherstellung der Konvexität werden die Wände nach ihren Lotwinkeln sortiert.

Klasse Wall

Die Klasse *Wall* speichert eine Gerade, die eine Wand definiert, in der Hesse-Form relativ zum Ursprung der Zelle. Die der Wand zugeordneten Merkmale werden getrennt in der Klasse *WallAttrVector* gespeichert, da eine Wand zu mehreren Zellen gehören kann, wobei die Attribute übereinstimmen, nicht jedoch die Hesse-Darstellung. Die Klasse *Wall* stellt Methoden zur Translation und Rotation von Geraden in Hesse-Form, zur Schnittpunktberechnung, zum Vergleich zweier Wände und zum einfachen Zugriff auf die Merkmale zur Verfügung.

Klasse WallAttrVector

Die Klasse *WallAttrVector* speichert den Merkmalsvektor einer Wand. Attribute einer Wand sind in der Wandebene gesehene Merkmale und Diskontinuitäten (insbesondere Türen).

Klasse WallAttr

Der Merkmalsvektor einer Wand setzt sich aus Einträgen vom Typ *WallAttr* zusammen.

Klasse WeightedPoint

Die Klasse *WeightedPoint* ist ein Erbe der im letzen Kapitel vorgestellten Klasse *Point* (4.4.1). Neben den Koordinaten wird zusätzlich das im Stereoalgorithmus ermittelte Gewicht der Landmarke gespeichert, das angibt, wieviele Punkte sich im Stereobild senkrecht auf diesen Punkt akkumuliert haben (siehe [Bar98]).

Klasse Accumulator

Die Klasse *Accumulator* speichert vom Stereoalgorithmus gelieferte Punkte bis zur Geradenextraktion zwischen und stellt Methoden zum Anhängen neuer Punkte, zur Hough-Transformation und zum Löschen des Akkumulators zur Verfügung.

Template-Klasse Matrix

Die Template-Klasse *Matrix* stellt Hilfsfunktionen für beliebigdimensionale Matrizen zur Verfügung, die für die Hough-Transformation bei der Wanderkennung benötigt werden. Ferner

wird eine Methode zur Verfügung gestellt, um die in der Matrix enthaltenen Werte für die Visualisierung zu normieren (siehe Anhang).

5.3.2 Änderungen im globalen Ablauf

Beim Eingang einer Bildnachricht vom Modul Depthmap werden die Daten zunächst zwischengespeichert, um – falls mehrere Aufnahmen von einem Standort aus gemacht werden – für die Hough-Transformation möglichst aussagekräftige Daten benutzen zu können.

Erst bei einer entdeckten oder generierten Bewegung wird die Bildanalyse durchgeführt. Die eigentliche Bildanalyse besteht aus drei Schritten:

1. Die Geraden werden mittels Hough-Transformation aus der zwischengespeicherten Punktmenge extrahiert.
2. Ein Erwartungsbild wird generiert und mit den extrahierten Geraden verglichen. Anhand der Übereinstimmung des Erwartungsbildes mit den extrahierten Geraden wird die Roboterposition- und Orientierung kalibriert.
3. Das Umweltmodell wird an die neuen Daten angepaßt.

Diese Schritte werden im folgenden detailliert beschrieben.

5.3.3 Geradenextraktion

Sind alle angeforderten Aufnahmen gesammelt oder wird der Roboter bewegt, so führt die Bildanalyse zunächst die Hough-Transformation für die gesammelten Daten von dem Modul Depthmap durch. In die Hough-Matrix wird dabei für jede mögliche Gerade, die innerhalb einer festgelegten Diskretisierung durch einen von dem Stereoverfahren gelieferten Punkt läuft, das vom Stereoverfahren gelieferte Gewicht des Punktes eingetragen.

Die dem höchsten Wert im Hough-Raum entsprechende Gerade wird als Hauptrichtung gespeichert. Punkte in der Umgebung der Geraden werden aus dem Hough-Raum gelöscht.

Die Hough-Transformation und Löschung der Punkte wird dann solange wiederholt, bis keine signifikanten Geraden mehr zu erkennen sind. Dabei werden nun Geraden, die parallel oder senkrecht zur Hauptrichtung liegen, höher bewertet.

Die Iteration der Hough-Transformation ist nötig, um nicht lokale Maxima suchen zu müssen. Eine Alternative zur Iteration wäre, alle Punkte im Hough-Raum in der Umgebung des globalen Maximums zu löschen und danach das neue globale Maximum zu suchen. Bei diesem Verfahren wurden allerdings schnell Geraden, auf denen viele Punkte von zwei korrekten Geraden lagen, die aber selbst keine „echte“ Wand darstellten, als Wand erkannt.

Die durch die Hough-Transformation gewonnen Geraden werden noch weiter untersucht: Die Position der Geraden wird durch Mittlung aller der Geraden zugeordneten Punkte genauer bestimmt. Enthält die Gerade merkmalsfreie Teilstücke, die länger als eine einstellbare Konstante sind, wird sie vor der weiteren Untersuchung an diesen Stellen unterteilt. Für alle Teilstücke wird eine Gauß-Ellipse bestimmt. Diese Gauß-Ellipse wird aus allen Punkten (x_i, y_i) mit dem Gewicht g_i, die auf einer Geraden liegen, berechnet, wie in Gleichung 5.3 dargestellt. Ist die durch die Gauß-Ellipse bestimmte Varianz der Punkte senkrecht zur Geraden, die dem Radius r_2 der Ellipse entspricht, zu groß oder weicht die Ausrichtung θ der Ellipse zu sehr von der durch die Hough-Transformation ermittelte Richtung ab, so wird das entsprechende Teilstück verwor-

fen. In Abbildung 5.8 ist neben den gefunden Geraden ein Beispiel für ein verworfenes und ein akzeptiertes Teilstück dargestellt.

Schließlich werden Punkte, die in der Umgebung der nicht verworfenen Teilstücke der Geraden liegen, auf die Gerade projiziert und deren Gewicht an der entsprechenden Stelle des Attributvektors der Geraden gespeichert.

Wie schon im Entwurf dargelegt, ist die Laufzeit der Hough-Transformation linear zur Anzahl der Punkte, die in die Hough-Matrix eingetragen werden. Pro Iteration wird mindestens eine Gerade gefunden. Da zur Bestimmung einer Geraden mindestens zwei Punkte notwendig sind, kann die Gesamtlaufzeit mit $O(n^2)$ abgeschätzt werden; n ist dabei die Anzahl der vom Stereoalgorithmus gelieferten Punkte. Die Sortierung der Punkte in der Richtung einer Geraden zur Suche von merkmalsfreien Teilstücken hat keinen Einfluß auf die Gesamtlaufzeit, da für den gesamten Algorithmus summiert maximal n Punkte sortiert werden.

Abbildung 5.8 zeigt die nach der iterierten Hough-Transformation in Realdaten gefundenen Geraden.

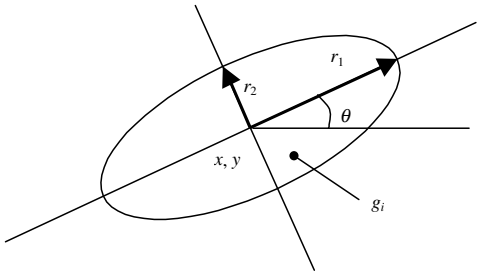


Abbildung 5.7: Die Parameter der Ellipse. Der zweite Radius r_2 entspricht der Varianz der Punkte senkrecht zur gefundenen Geraden, θ muß der durch die Hough-Transformation ermittelten Richtung entsprechen.

$$M_0 = \sum_i g_i \qquad S_{xx} = \frac{M_{xx}}{M_0} - \left(\frac{M_x}{M_0}\right)^2 \qquad x = \frac{M_x}{M_0}$$

$$M_x = \sum_i g_i x_i \qquad S_{yy} = \frac{M_{yy}}{M_0} - \left(\frac{M_y}{M_0}\right)^2 \qquad y = \frac{M_y}{M_0}$$

$$M_y = \sum_i g_i y_i \qquad S_{xy} = \frac{M_{yy}}{M_0} - \frac{M_x \cdot M_y}{{M_0}^2} \qquad r_1 = \sqrt{\frac{K}{d_1 - \sqrt{{d_2}^2 + {S_{xy}}^2}}}$$

$$M_{xx} = \sum_i g_i {x_i}^2 \qquad K = S_{xx} \cdot S_{yy} - {S_{xy}}^2 \qquad r_2 = \sqrt{\frac{K}{d_1 + \sqrt{{d_2}^2 + {S_{xy}}^2}}}$$

$$M_{yy} = \sum_i g_i {y_i}^2 \qquad d_1 = \frac{S_{xx} + S_{yy}}{2} \qquad \theta = \frac{\arctan 2(S_{xy}, d_2)}{2}$$

$$M_{xy} = \sum_i g_i x_i y_i \qquad d_2 = \frac{S_{xx} - S_{yy}}{2}$$

Gleichung 5.3: Bestimmung der beiden Radien r_1 und r_2 und der Orientierung θ der Gauß-Ellipse. Die x_i und y_i sind die Koordinaten der Punkte; die g_i sind die entsprechenden Gewichte.

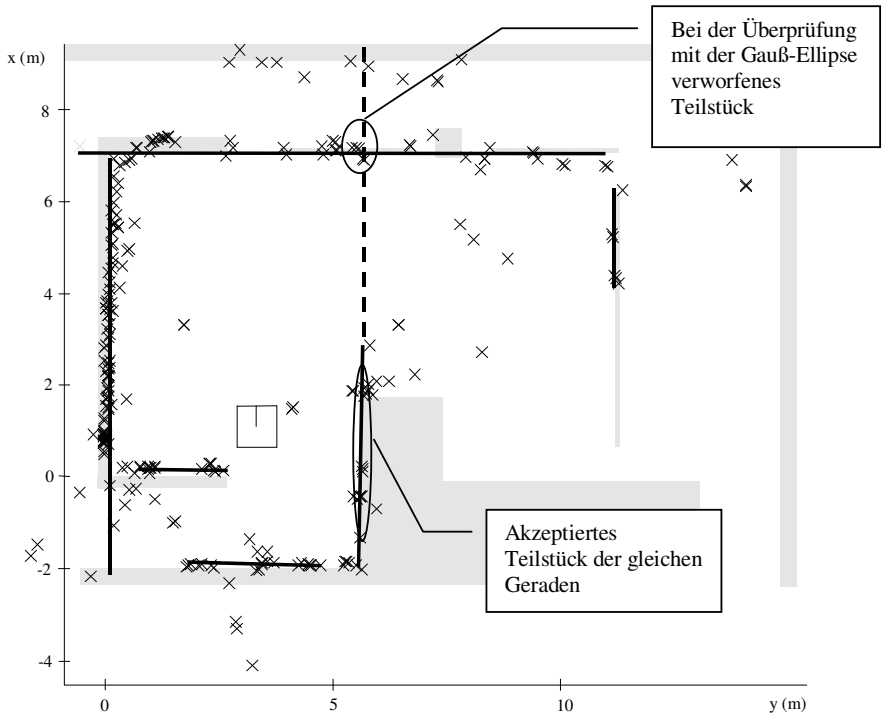


Abbildung 5.8: Nach der iterierten Hough-Transformation gefundene Geraden; ein Plan des Testraumes, in dem die Daten aufgenommen wurden, ist hellgrau hinterlegt.

---

function *Accumulator.hough*: **array of** *Wall*;

// Initialisierungen

hough := *Matrix.create* (*Distances*, *Angles*);
alignAngle := *NAN*;
lines := [];

while *true* **do**

// Teil 1: Hough-Matrix bestimmen und auswerten

hough.fill (0);

for $i = 0$ **to** *accumulator.count* – 1 **do**;
a := *accumulator* [*i*];

if *a.weight* > 0 **then**
for $j := 0$ **to** *Angles* – 1 **do**
$\alpha := 2\pi \cdot j$ / *Angles* – π;
$n_0 := a.x \cdot \cos(\alpha) + a.y \cdot \sin(\alpha)$;
$idx := n_0$ / *MaxDistance* · *Distances*;

if $n_0 \geq 0 \wedge n_0 <$ *MaxDistance* **then**
if *alignAngle* = NAN $\vee$ | ($\pi + \alpha$ – *alignAngle*) modulo (π / 2) | $< 2\pi$ / *Angles* **then**
hough [*idx*, *j*] := *hough* [*idx*, *j*] + *a.weight* · *AlignFactor*;

```
                    else
                        hough [idx, j] := hough [idx, j] + a.weight;
                    fi;
                fi;
            od;         // j
        fi;             // weight > 0
od;                     // i

// beste Gerade bestimmen

mx := hough.getMax (y, x);
bestAngle := x / Angles · 2π – π;
bestDistance := y / Distances · MaxDistance;

// Abbruch, falls gefundene Gerade nicht mehr signifikant

if mx < MinWeight then
    break;
fi;

// Hauptrichtung bestimmen, falls noch nicht geschehen

if alignAngle = NAN then
    alignAngle := bestAngle;
fi;

// Teil 2: genaue Position der Geraden bestimmen

center := Point.create;
weightSum := 0;

for i := 0 to accumulator.count – 1 do

    // Parallele zur gefunden Gerade durch Punkt bestimmen

    a := accumulator [i];

    if |bestDistance – (a.x · cos (bestAngle) + a.y · sin (bestAngle))|
            < 2 · MaxDistance / Distances then
        center := center + a · a.weight;
        weightSum := weightSum + a.weight;
    fi;
od;

// Gerade auf Gerade durch center verschieben

center := center / weightSum;
bestDistance := (center.x · cos (bestAngle) + center.y · sin (bestAngle));

// Stützpunkt center auf Lotpunkt korrigieren

center := Point.create (cos (bestAngle), sin (bestAngle)) · bestDistance;

// Punkte, die auf der nun verschobenen Geraden liegen, sammeln und
// im Puffer durch setzen des Gewichts auf 0 als gelöscht markieren

weightSum := 0;
set := [];

for i := 0 to accumulator.count – 1
    a := accumulator [i];

    if a.weight > 0 then
```

```
            if |bestDistance – (a.x · cos (bestAngle) + a.y · sin (bestAngle))|
                                        < 2 · MaxDistance / Distances then
                dx := center.distance (a);
                if normAngle (a.getAngle – bestAngle) < 0 then
                    dx := –dx;
                fi
            fi
            dy := bestDistance – (a.x · cos (bestAngle) + a.y · sin (bestAngle));
            set.append (WeightedPoint.create (dx, dy, a.weight));
            weightSum := weightSum + a.weight;
            a.weight := 0;
        fi
    od
    // Abbruchbedingung erneut überprüfen
    if weightSum < MinWeight then
        break;
    fi
    // an merkmalsfreien Teilstücken getrennte Geradensegmente einzeln untersuchen;
    // dazu zuerst Punkte auf der Geraden sortieren
    set.sort (lessX);
    wall := Wall.crate (bestAngle, bestDistance);
    gaussSet := [];
    set.append (WeightedPoint.create (∞, 0, 0));
    d0 := –∞;
    anyok := false;
    // Durchlauf durch die sortierte Punktmenge; die gefundenen Geradensegmente
    // werden direkt mit der Gauß-Ellipse genauer untersucht
    for i := 0 to set.count – 1 do
        d1 = set [i].x;
        if d1 – d0 > 2 then
            if gaussSet.count > 2 then
                ge := GaussEllipse.create (gaussSet);
                if (max (ge.r1, ge.r2) / min (ge.r1, ge.r2) > MinRadiusFactor) ∧
                                        (max (ge.r1, ge.r2) > MinRadius) then
                    for j := 0 to gaussSet.count – 1 do
                        wall [gaussSet [j].x].weight := wall [gaussSet [j].x].weight
                                                    + gaussSet [j].weight;
                    od;
                    anyok := true;
                fi;
            fi;
            gaussSet := [];
        fi;
        gaussSet.append (set [i]);
        d0 := d1;
    od;
```

```
        // Gerade nur in das Ergebnis aufnehmen, falls mindestens ein Teilstück die
        // Prüfung mittels Gauß-Ellipse bestanden hat.
        if anyok then
            lines.append (wall);
        fi;
    od;
    return lines.
```

Algorithmus 5.2: Iterierte Hough-Transformation zur Bestimmung von Wänden

5.3.4 Rekalibration

Die Rekalibration läßt sich in drei Hauptschritte gliedern: Zuerst wird ein Erwartungsbild generiert, dann werden den aktuell detektierten Geraden entsprechende Geraden aus dem Erwartungsbild zugeordnet und schließlich wird die Roboterorientierung und -position anhand der Zuordnungen kalibriert. Diese drei Schritte werden im folgenden detailliert dargestellt.

Generierung eines Erwartungsbildes

Wie schon bei der Punkt-Repräsentation wird als erster Schritt der Rekalibration ein Erwartungsbild generiert. Das Erwartungsbild soll die Geraden enthalten, die von der aktuellen vermuteten Position aus sichtbar sind.

Dazu können zunächst alle Geraden der aktuellen Zelle, die zumindest teilweise in dem sichtbaren Winkelbereich liegen, in eine Liste der sichtbaren Geraden eingetragen werden. Dann können alle Geraden auf Diskontinuitäten untersucht werden. Für Zellen jenseits dieser Diskontinuitäten müßte – wie schon bei der ersten Repräsentation – diese Funktion rekursiv mit einem entsprechend verkleinerten Winkelbereich aufgerufen werden. Neu in die Liste aufgenommen werden dürfen nur Geraden, die nicht bereits in der Liste enthalten sind.

Theoretisch sichtbare Merkmale im Attributvektor der Geraden könnten bei diesem Funktionsaufruf altern, um Merkmale, die z.B. durch Fehlzuordnungen in den Attributvektor aufgenommen wurden, aber nicht wirklich in der Welt existieren, mit der Zeit wieder zu entfernen.

Zur Zeit ist ein prototypischer Algorithmus implementiert, der einfach alle Wände der aktuellen Zelle und der Nachbarzellen zurückliefert.

Zuordnung

In Algorithmus 5.3 wird jede Gerade des Erwartungsbildes (*expectation*) mit allen aktuell gesehenen Geraden (*visibleWalls*) verglichen, und die Geraden mit der geringsten Abweichung bezüglich Winkel, Entfernung und Merkmale werden einander zugeordnet.

Eine Fehlzuordnung könnte in den nachfolgenden Stufen des Algorithmus erkannt und durch eine globale Rekalibration aufgelöst werden.

```
    procedure GlobalNavigation.match (visible, expectation: array of Wall);
    sum := 0;
```

```
for i := 0 to visible.count – 1 do
    bestMatch := 0;
    bestIndex := –1;
    for j := 0 to expectation.count – 1 do
        // compare vergleicht die Hesse-Formen zweier Geraden einschließlich der
        // Merkmalverteilung auf der Wand. Das Ergebnis ist normiert zwischen
        // 0 (keine Übereinstimmung) und 1 (vollkommene Übereinstimmung)
        match := visible [i].compare (expectation [j]);
        if match > bestMatch then
            bestMatch := match;
            bestIndex := j;
        fi;
    od;
    if bestMatch > MinMatch then
        sum := sum + bestMatch;
        visible [i].matched := expectation [bestIndex];
    else
        visible [i].matched := NIL;
    fi
return sum / visible.count.
```

Algorithmus 5.3: Zuordnung von gesehenen Geraden und Erwartungsbild

In dem Algorithmus wird (noch) nicht sichergestellt, daß keine Gerade aus dem Erwartungsbild mehrfach zugeordnet wird.

Die Laufzeit für den Vergleich der Merkmale der Geraden ist linear zur Länge der Geraden; mit dem Rahmen ergibt sich insgesamt das Produkt aus der maximalen Länge der Geraden l, der Anzahl der Geraden im Erwartungsbild e und der Anzahl der aktuell gesehenen Geraden v, also $O(l \cdot e \cdot v)$. Dabei ist zu berücksichtigen, daß die Anzahl der gesehenen Geraden in einem Raum relativ klein ist, entsprechend auch die Anzahl der Geraden im Erwartungsbild.

Winkel- und Positionsrekalibration

Als Winkelabweichung Δ_α wird die gemittelte Winkelabweichung zwischen den Winkeln α_i der n gesehenen Geraden, denen eine Gerade aus dem Erwartungsbild zugeordnet werden konnte und den entsprechenden Winkeln β_i der Geraden aus dem Erwartungsbild ermittelt:

$$\Delta_\alpha = \frac{\sum_{i=1}^{n} \alpha_i - \beta_i}{n} \tag{5.4}$$

Der berechnete Winkelfehler wird dann zur aktuell gespeicherten Roboterausrichtung addiert.

Die Positionsabweichung wird genauso ermittelt: Für jedes zugeordnete Geradenpaar wird die Positionsabweichung d_i bestimmt; mit den gemittelten Werten wird die Position des Roboters rekalibriert. Bei der Positionskorrektur wird zusätzlich berücksichtigt, daß eine Bewegung parallel zu einer Wand mit Abstandsmessungen bezüglich dieser Wand nicht korrigiert werden kann. Eine Gerade geht also nur in die Korrektur der x- bzw. y-Komponente der Position ein,

sofern sie nicht parallel oder annähernd parallel zu der entsprechenden Achse liegt. Dazu wird sichergestellt, daß der Faktor, mit dem die Korrektur zur entsprechenden Richtung beiträgt, größer als 0,3 ist; ansonsten wird die Positionskorrektur in der entsprechenden Richtung nicht gewertet.

$$\Delta_x = \frac{\sum_{i=1}^{n} \delta(|\cos(\alpha_i)| > 0{,}3) \cdot \sin(\alpha_i) \cdot d_i}{\sum_{i=1}^{n} \delta(|\cos(\alpha_i)| > 0{,}3)} \tag{5.5}$$

$$\Delta_y = \frac{\sum_{i=1}^{n} \delta(|\sin(\alpha_i)| > 0{,}3) \cdot \sin(\alpha_i) \cdot d_i}{\sum_{i=1}^{n} \delta(|\sin(\alpha_i)| > 0{,}3)} \tag{5.6}$$

Bei der Winkel- und Positionsrekalibration kann zusätzlich die Varianz ermittelt und bei Überschreitung eines Toleranzwertes eine globale Selbstrelokation versucht werden.

Die Laufzeiten für sowohl die Winkelrekalibration als auch die Positionsrekalibration sind jeweils linear zur Anzahl der gesehenen Geraden.

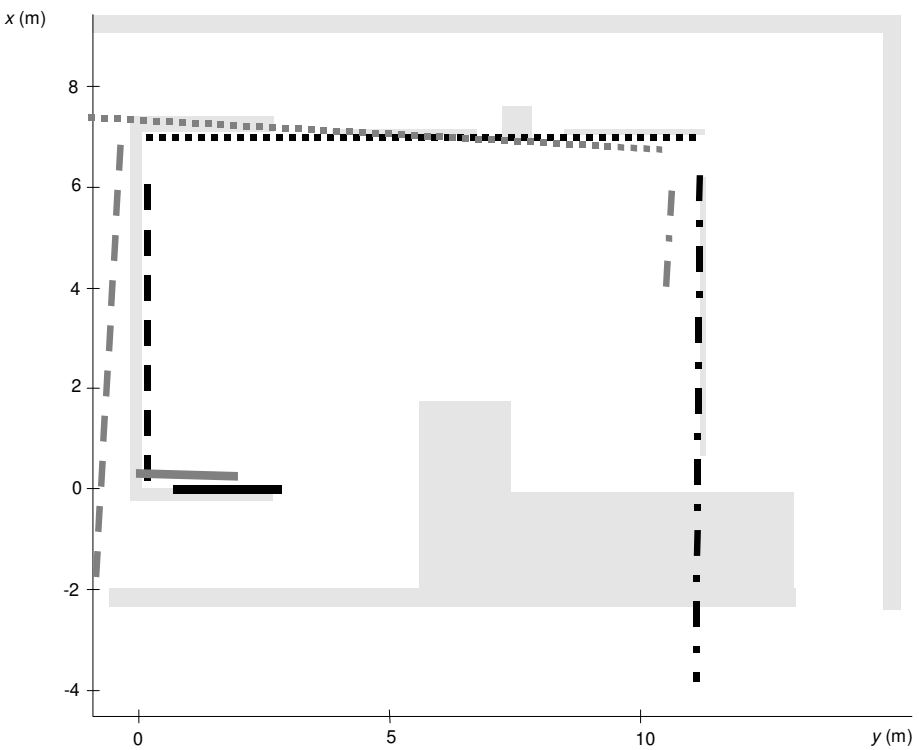


Abbildung 5.9: Überlagerung von detektierten Geraden und Erwartungsbild (aus Realdaten). Die grauen Linien sind das Erwartungsbild, die schwarzen Linien wurden aktuell gesehen. Neben der relativ großen horizontalen Positionsabweichung ist auch eine leichte Winkelabweichung zu erkennen. Einander zugeordnete Linien sind mit dem gleichen Muster versehen

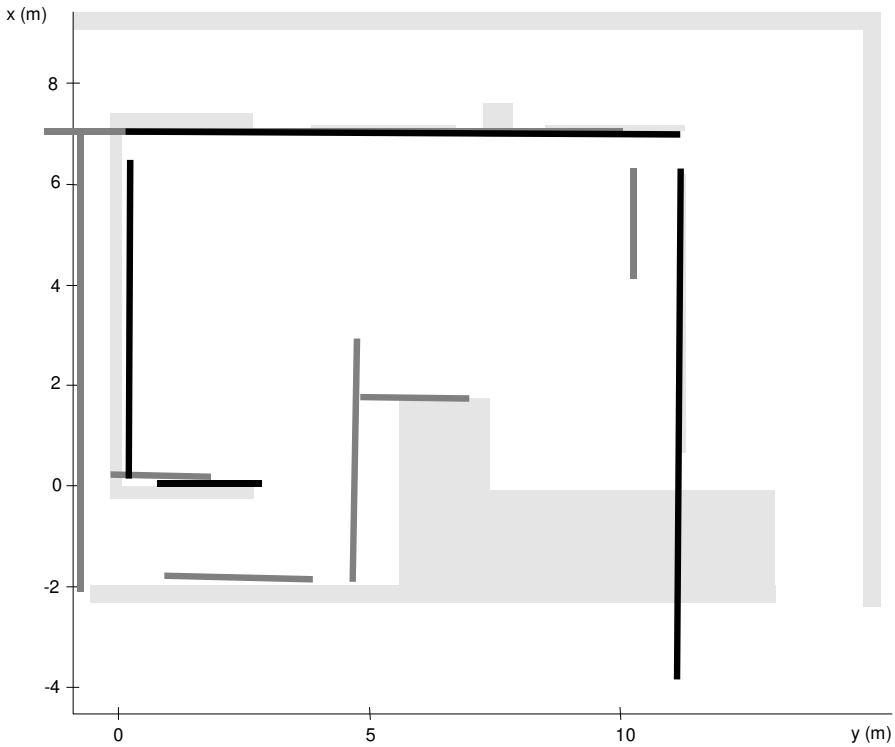


Abbildung 5.10: Überlagerung der Geraden aus Abbildung 5.9 nach der Winkelrekalibration

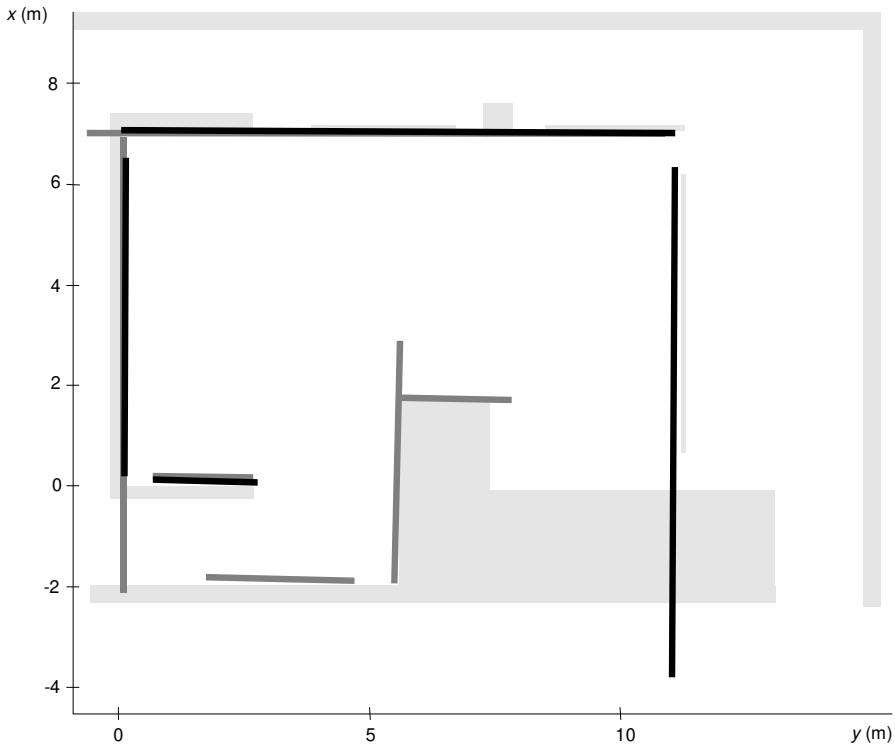


Abbildung 5.11: Überlagerung der Geraden nach Winkel- und Positionsrekalibration

5.3.5 Aktualisierung der Datenstruktur

Die Aktualisierung der Datenstruktur wird in fünf Schritten durchgeführt:

1. Einfügen aller neuen Geraden in die aktuelle Zelle
2. Positionskorrektur wiedererkannter Geraden, also der Geraden, denen eine Gerade aus dem Erwartungsbild zugeordnet werden konnte
3. Detektion von Diskontinuitäten anhand der neuen Daten
4. Verschmelzung der Merkmale wiedererkannter Geraden
5. Verschiebung von Geraden, die nun in dieser Zelle komplett verdeckt sind, in eine andere Zelle

Diese Schritte werden im folgenden detaillierter betrachtet.

Einfügen neuer Geraden

Neue Geraden, also Geraden, die keiner Geraden des Erwartungsbildes zugeordnet werden konnten, werden zunächst einfach an die Datenstruktur der Zelle angehängt. Eine Zuordnung zur richtigen Zelle findet erst im letzten Schritt der Aktualisierung der Datenstruktur statt.

Positionskorrektur wiedererkannter Geraden

Für Geraden, die einer Geraden aus dem Erwartungsbild zugeordnet wurden, können Winkel und Distanz aus dem alten und neu gemessenen Wert gemittelt werden.

Detektion von Diskontinuitäten

Für alle gesehenen Geraden wird für jedes gesehene Merkmal auf dieser Geraden überprüft, ob die Gerade vom Roboter zu diesem Merkmal eine andere Gerade schneidet, wie in Abbildung 5.12 dargestellt. In diesem Fall muß an dem Schnittpunkt eine Diskontinuität vorliegen, die in den Attributvektor der geschnittenen Geraden eingetragen wird.

Die entsprechende Methode (Algorithmus 5.4) muß für jedes Merkmal auf jeder neuen Wand die Verdeckung durch alle Geraden der Zelle überprüfen, die Laufzeit beträgt also insgesamt $O(l \cdot v \cdot n)$, wobei l die maximale Wandlänge, v die Anzahl der neuen Geraden und n die Anzahl der Geraden der Zelle ist. Die Anzahl der Wände bleibt dabei relativ klein; eine rechtwinklige konvexe Zelle hat beispielsweise genau vier Wände.

```
procedure Cell.checkForDoors (pos: Point, wall: Wall);
for i := 0 to walls.count – 1 do
    if (¬walls [i].isSame (wall)) then
        // zuerst alte Wand an Standpunkt verschieben
        translated = walls [i].translate (pos – origin);
        // alle Punkte der neuen Wand auf Verdeckung durch Wand i testen
```

```
            for j := wall.getMin to wall.getMax step WallStep do
                if wall [j].weight > 0 then
                    thePoint := wall.getPoint (j);
                    distance := translated.getDistance (thePoint);
                    translated [translated.distance · tan (thePoint.angle
                                - translated.angle)].transparent := true;
                fi;
            od;
        fi;
    od.
```

Algorithmus 5.4: Bestimmung von Diskontinuitäten einer Geraden durch dahinter sichtbare Merkmale.

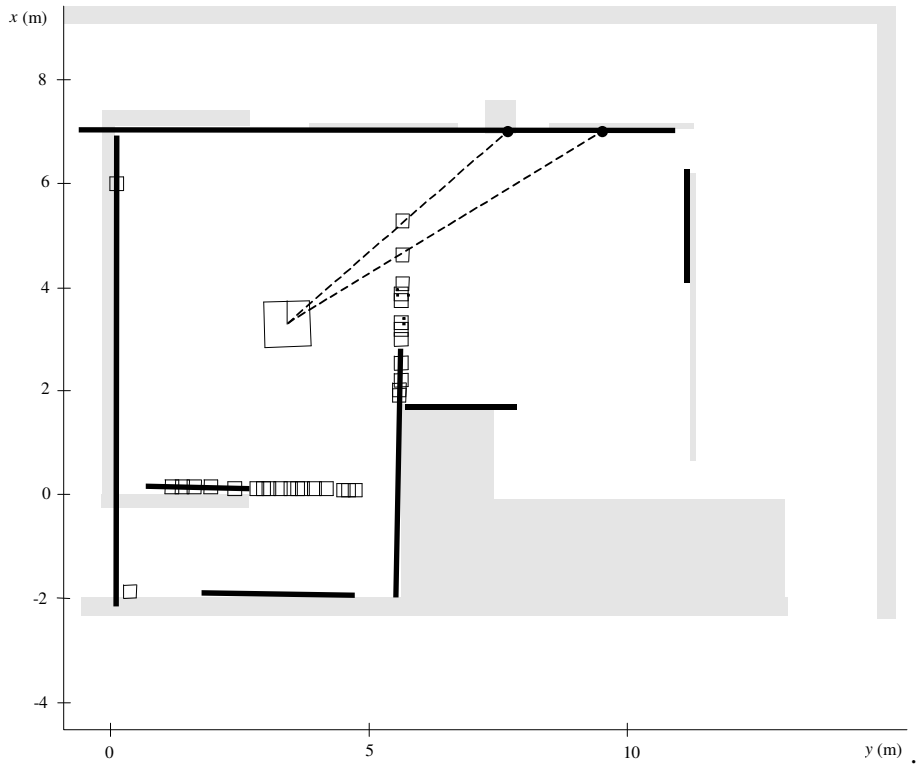


Abbildung 5.12: Bestimmung von Diskontinuitäten einer Geraden durch dahinter sichtbare Merkmale, im Bild durch Kästchen markiert.

Merkmalverschmelzung

Die im Attributvektor gesehener Geraden gespeicherten Merkmale werden zum Attributvektor der zugeordneten Geraden addiert.

Zellenbereinigung

Da neue Geraden ungeprüft in die Zelle eingefügt wurden, muß nun die Konvexität der Zelle wieder sichergestellt werden. Dazu wird für jede Gerade überprüft, ob sie einen Schnittpunkt mit einer anderen Geraden besitzt, der nicht durch eine dritte Gerade verdeckt wird. Sind alle

Schnittpunkte der Geraden verdeckt, so gehört die Gerade nicht zur aktuellen Zelle (wie in Abbildung 5.13 dargestellt) und wird einer anderen Zelle zugeordnet.

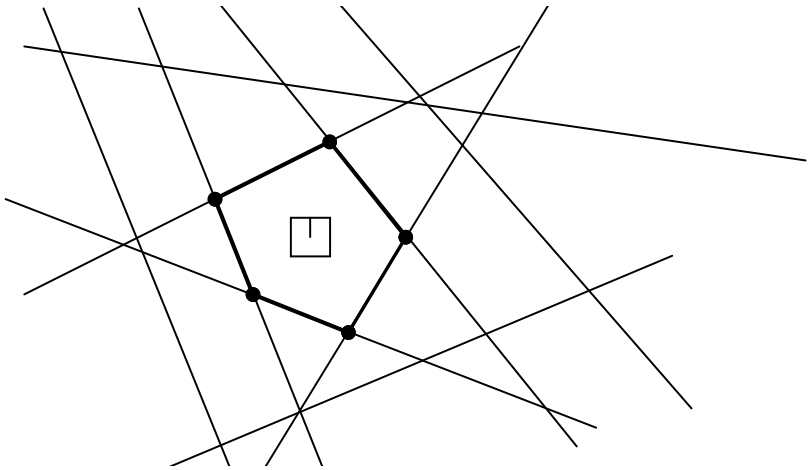


Abbildung 5.13: Zur konvexen Zelle gehören nur Geraden, die einen nicht verdeckten Schnittpunkt besitzen.

Den Schnittpunkten zweier Geraden in Hesse-Form erhält man durch Gleichung 5.7, dabei ist n_{01} der Abstand der ersten Geraden vom Ursprung, n_{02} der Abstand der zweiten Geraden vom Ursprung, α_1 der Lotwinkel der ersten Geraden und α_2 der Lotwinkel der zweiten Geraden; x und y sind die Koordinaten des Schnittpunktes.

$$x = -\frac{n_{02} \cdot \sin(\alpha_1) - n_{01} \cdot \sin(\alpha_2)}{\cos(\alpha_1) \cdot \sin(\alpha_2) - \cos(\alpha_2) \cdot \sin(\alpha_1)}$$

$$y = -\frac{n_{01} \cdot \cos(\alpha_2) - n_{02} \cdot \cos(\alpha_1)}{\cos(\alpha_1) \cdot \sin(\alpha_2) - \cos(\alpha_2) \cdot \sin(\alpha_1)}$$

Gleichung 5.7: Bestimmung des Schnittpunktes zweier Geraden in Hesse-Form

Die Laufzeit für Algorithmus 5.5 beträgt pro Zelle offensichtlich $O(n^3)$, wobei n die normalerweise sehr kleine Anzahl der Wände der Zelle ist.

Eine Verbesserungsmöglichkeit für Algorithmus 5.5, der alle Geraden einzeln überprüft, wäre, einen nicht verdeckten Schnittpunkt als Zeugen für beide beteiligten Geraden zu nutzen.

```
procedure Cell.cleanUp
// Alle Wände walls der Zelle überprüfen
i := 0;
while i < walls.count do
    anyIntersection := false;
    allHidden := true;
    for j := 0 to walls.count – 1 do
        if (i ≠ j) ∧ (walls [i].intersection (walls [j]), p)) then
            hidden := false;
            anyIntersection := true;
```

```
                for k := 0 to walls.count – 1 do
                    if (k ≠ i) ∧ (k ≠ j) ∧ (walls [k].getDistance (p) > 0) then
                        hidden := true;
                        break;
                    fi;
                od;
                if ¬hidden then
                    allHidden := false;
                    break;
                fi;
            fi;
        od;
        // Wand verlagern, falls erforderlich
        if anyIntersection ∧ allHidden then
            pNeighbour := find (origin, walls [i]));
            if pNeighbour = NIL then
                pNeighbour := Cell.create (pMain);
                pNeighbour.neighbours.append (this);
                neighbours.append (pNeighbour);
                pNeighbour.origin := origin + walls [i].getPoint (0) · 0,8;
            fi
            pNeighbour.check (walls [i].translate (pNeighbour.origin – origin));
            walls.delete (i);
        else
            i := i + 1;
        fi;
    od;
    // Sortierung (für Globale Selbstlokalisation)
    walls.sort;
    sorted := true.
```

Algorithmus 5.5: Entfernung von Wänden, die nicht zur aktuellen Zelle gehören

5.3.6 Globale Selbstlokalisation

Bei der globalen Selbstlokalisation kann kein Wissen über die ungefähre Position zur Positionsbestimmung des Roboters benutzt werden. Wie im Entwurf beschrieben, werden die aktuell gesehenen Wände mit Erwartungsbildern aus jedem Raum verglichen: Zwei sich schneidende Geraden aus einem aktuellen Rundblick werden jeweils mit allen Ecken der konvexen Zellen mit ähnlichen Winkeln zur Deckung gebracht. Für alle diese Positionshypothesen wird ein Erwartungsbild generiert. Für die beste Übereinstimmung kann dann der lokale Rekalibrationsalgorithmus angewendet werden.

Die Methode *globalRecalibration* der Klasse *GlobalNavigation* (Algorithmus 5.6) leistet den ersten Teil der globalen Selbstlokalisation. Sie bekommt im Parameter *visibleWalls* die aus dem Rundblick generierten Geraden übergeben, für die alle n^2 Schnittpunkte berechnet werden. Für jeden Schnittpunkt wird dann für jede Zelle die Methode *compareCorners* der Klasse *Cell* aufgerufen.

In *compareCorners* (Algorithmus 5.7) wird die übergebene Ecke mit allen Ecken der konvexen Zelle verglichen. Bei einer annähernden Übereinstimmung des Winkels wird ein Erwartungsbild unter der Positionshypothese generiert, die sich ergibt, wenn man die beiden Ecken zur Dekkung bringt.

Da die Wände der konvexen Zelle im Gegensatz zu den aktuell gesehen Geraden nach Winkeln sortiert sind, müssen nur für jeweils zwei aufeinanderfolgende Geraden die Schnittpunkte berechnet werden, in diesem Fall ist der Aufwand also nur linear. Bei den gesehen Geraden kann diese Sortierung nicht durchgeführt werden, da sie noch nicht in konvexe Zellen zerlegt worden sind.

Die Laufzeit der globalen Selbstlokalisation ist um den Faktor $v^2 \cdot c \cdot n_c$ höher als eine lokale Rekalibration, wobei v die Anzahl der aktuell gesehenen Geraden, c die Anzahl der Zellen und n_c die maximale Anzahl von Wänden einer Zelle ist. Dabei ist wiederum zu berücksichtigen, daß die beiden Anzahlen der Wände sehr klein bleiben.

Eine Verbesserung, die aber keinen Einfluß auf die Größenordnung der Laufzeit hat, wäre, den Vergleich von gesehenen Geraden und Erwartungsbild abzubrechen, sobald feststeht, daß der bisher beste Wert nicht erreicht werden kann.

```
procedure GlobalNavigation.globalRecalibration (visibleWalls: Array of Wall);
bestFit := 0;
bestHeading:= 0;
bestPosition :=  Point.create (0,0);
// Alle Schnittpunkte bestimmen
for i := 0 to visibleWalls.count – 1 do
    for j := 0 to visibleWalls.count – 1 do
        if i ≠ j then
            wall1 := visibleWalls [i];
            wall2 := visibleWalls [j];
            if wall1.intersection (wall2) then
                ip := wall1.intersectionPoint (wall2);
                angle := normAngle (wall2.angle – wall1.angle);
                if angle < 0 then
                    swap (wall1, wall2);
                    angle := –angle;
                fi;
                // Mit aktuellem Schnittpunkt alle Zellen durchlaufen
                for m := 0 to cells.count – 1 do
                    cells [m].compareCorners (visibleWalls,
                                              wall1.distance, wall2.distance, angle,
                                              bestFit, bestPos, bestAngle)
                fi;
            fi;
        fi;
    od;
od;
```

```
if bestFit > MinFitGlobalRecalibration then
    calibratedHeading := bestHeading;
    calibratedPosition := bestPosition;
fi.
```

Algorithmus 5.6: Der Rahmen der globalen Selbstlokalisation

```
procedure Cell.compareCorners (visibleWalls: array of Wall; ip: Point;
                               distance1, distance2, angle: Real;
                               var bestFit, bestPosition, bestHeading: Real);
if walls.count < 2 then
    return;
fi;
// Alle Wände der Zelle durchlaufen und Winkel prüfen
for i := 0 to walls.count – 1 do
    wall2 := walls [i];
    if i = 0 then
        wall1:= walls [walls.count – 1];
    else
        wall1:= walls [i – 1];
    fi;
    if wall1.intersection (wall2) ∧
            |normAngle (wall2.angle – wall1.angle) – angle| < CornerTolerance then
        // Winkel OK, nun Schnittpunkt und dadurch implizierte Position bestimmen
        ip2 := walls1.intersectionPoint (wall2)
        wall1.distance := wall1.distance – distance1;
        wall2.distance := wall2.distance – distance2;
        estPosition := wall1.intersectionPoint (wall2);
        // Erwartungsbild von der angenommenen Position bestimmen
        visibleFromEstPos := getVisibleWalls (estPosition,
                normAngle ((ip2 – estPosition).getAngle – ip.getAngle), –π, π);
        // Übereinstimmung von gesehenen Geraden und Erwartungsbild bestimmen
        fitness := pMain.match (visibleWalls, visibleFromEstPos);
        // Positionshypothese in den Referenzparametern speichern, falls bisher beste
        if fitness > bestFit then
            bestFit := fitness;
            bestPosition := estPos;
            bestAngle := normAngle ((ip2 – estPos).getAngle – ip.getAngle;
        fi;
    fi;
od;

return.
```

Algorithmus 5.7: Vergleich einer Ecke aus dem aktuellen Rundblick mit allen Ecken einer konvexen Zelle.

Kapitel 6

Integration der Repräsentation

Die entwickelte Umweltrepräsentation muß noch in ein Verhalten des Roboters eingebettet werden, das Sensordaten anfordert, in die Repräsentation aufnimmt und die Umwelt durch Bewegung aktiv erkundet.

In diesem Kapitel wird die Integration der entwickelten Umweltrepräsentation in das Navigationsmodul und die Interaktion des Navigationsmoduls mit den anderen Systemmodulen des Roboters zur Erkundung der Umwelt dargestellt.

6.1 Anforderungen

Die Globalnavigation soll ein Verhalten des Roboters erzeugen, das zu einer vollständigen Kartierung seiner Umgebung führt. Dabei soll berücksichtigt werden, daß bei Fertigstellung des Verhaltensmoduls (siehe 2.4) der Globalnavigation jederzeit die Kontrolle über den Roboter entzogen werden kann.

6.2 Entwurf

Die Erkundung der Umwelt wird in drei mögliche Verhaltensweisen zerlegt:

- Generierung und Vervollständigung eines Rundblickes,
- Gezielte Untersuchung einer Wand auf Merkmale durch Anfahrt und Bildanforderung und
- Untersuchung eines neuen Raumes durch Generierung einer Durchfahrt.

Die drei Verhaltensweisen des Roboters resultieren in verschiedene Aufträge oder Ziele für andere Module. Für den Rundblick und bei Erreichen einer Wand muß eine Bildanforderung generiert werden, für die gezielte Untersuchung einer Wand innerhalb der aktuellen Zelle muß eine lokale Bewegung generiert werden und zur Erkundung einer anderen Zelle muß schließlich eine globale Bewegung generiert werden.

Um die geforderte Unterbrechbarkeit zu erhalten, soll die Zielgenerierung hauptsächlich von den Informationen aus Nachrichteneingängen, also dem aktuellen „Wissen“ des Roboters, abhängig sein. Von dem zuletzt generierten Ziel dagegen soll das nächste Ziel möglichst unabhängig sein. So kann ein generiertes, aber nicht ausgeführtes Ziel gegebenenfalls neu generiert werden, sofern das in der geänderten Situation noch sinnvoll ist. Wenn der Roboter beispielsweise

durch ein anderes Modul bewegt wurde, soll die neue Position bei der Generierung eines neuen Zieles berücksichtigt werden, was auch dazu führen kann, das ein anderes Ziel ausgewählt wird, das nun näher an der Roboterposition liegt.

Grundsätzlich sendet die Globalnavigation eine initiale Verhaltensanforderung. Danach wartet sie auf Nachrichten als Ergebnis der Verhaltensanforderung oder einer Aktion eines andren Moduls. Nach Verarbeitung der Nachricht wird ein neues Verhalten generiert.

Ein Überblick über die Struktur der Globalnavigation wird in Abbildung 6.1 gegeben; die Einbindung in die Gesamtarchitektur des Roboters ist in Abbildung 2.1 dargestellt.

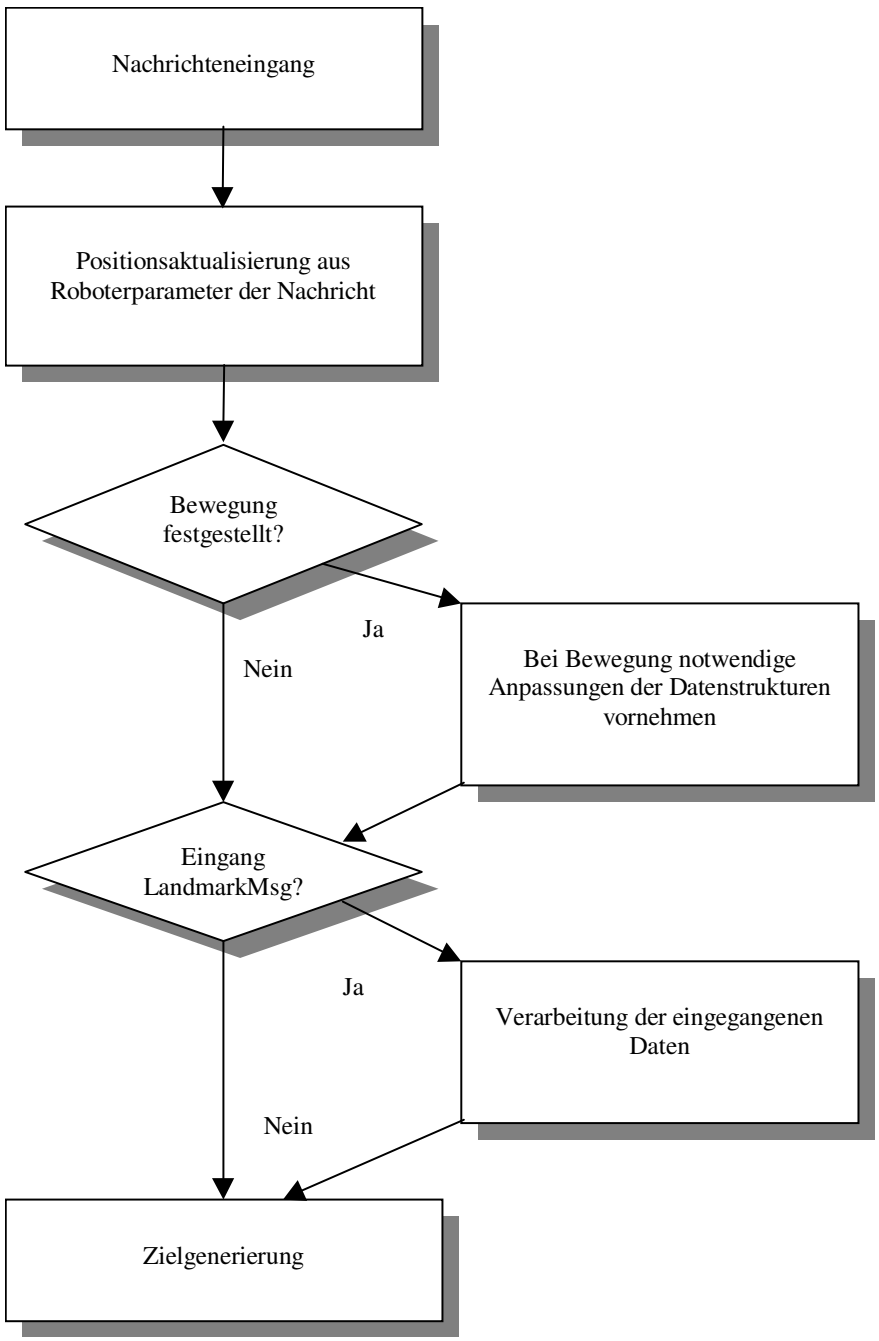


Abbildung 6.1: Überblick über die internen Abläufe der Globalnavigation

6.3 Implementierung

Im folgenden werden zunächst die verwendeten Datenstrukturen vorgestellt. Anschließend wird die Implementierung der im Entwurf ausgearbeiteten Generierung von Zielen und der Verarbeitung eingegangener Nachrichten beschrieben.

6.3.1 Datenstrukturen

Der Modulrahmen besteht aus der Klasse *GlobalNavigation*, die ein Erbe der Klasse *DepthMapClient* ist, die die nicht objektorientierten Bildanforderungen an das Modul Picserver kapselt. Neben dem Modulrahmen werden noch die Nachrichtenklassen *MoveReqMsg* und *MoveAckMsg* für die Kommunikation mit der Lokalnavigation und *LandmarkMsg* für Nachrichten von dem Modul Depthmap definiert.

Klasse DepthMapClient

Die abstrakte Klasse *DepthmapClient* kapselt die Kommunikation der Globalnavigation mit dem Picserver, um die Codierung der Bildanforderung in das von dem Picserver erwartete Format aus dem Hauptprogramm herauszuhalten. In der Simulation wird für Bildanforderungen ein Nachrichtenobjekt vom Typ *LookAtMsg* verwendet, das zu dem vom Picserver erwarteten Format nicht kompatibel ist. Durch diese Zwischenklasse kann die Klasse *GlobalNavigation* ohne Änderungen auf dem Roboter und in der Simulation verwendet werden.

Klasse GlobalNavigation

Die Klasse *GlobalNavigation* ist das Hauptmodul der Globalnavigation und steuert den groben Ablauf des Programmes, insbesondere den Empfang von Nachrichten anderer Module. Außerdem speichert das Objekt den aktuellen für die Globalnavigation bekannten Zustand des Roboters.

GlobalNavigation ist Erbe von *DepthmapClient* und überschreibt die von dem abstrakten PLANET-Prozeßmodul geerbte Methode *run*.

Nachrichtenklassen

Die Nachrichtenklassen kapseln die Generierung und Analyse von PLANET-Nachrichten. Ein abstrakter Vorfahre enthält Methoden, um sich selbst in PLANET-Kanäle zu schreiben, aus PLANET-Nachrichten zu dekodieren oder sich binär oder lesbar in einen Stream zu schreiben oder sich aus einem Stream zu dekodieren. Die Erben müssen zur Nutzung dieser Möglichkeiten nur eine allgemeine Methode zur Codierung überschreiben, die durch Benutzung von Referenzübergaben sowohl die Codierung als auch die Decodierung leisten kann.

Die Klasse *MoveReqMsg* kann Zielpunkte aufnehmen und sich an die Lokalnavigation übermitteln. Positionszielbestätigungen erhält die Globalnavigation von der Lokalnavigation als Objekt vom Typ *MoveAckMsg*. Die Klasse *LandmarkMsg* dient der Speicherung und Übermittlung von Punkten aus dem Stereoalgorithmus.

6.3.2 Hauptschleife

Beim Start des Roboters erzeugt die Globalnavigation eine initiale Bildanforderung. Danach reagiert sie in einer Endlosschleife auf eingehende Nachrichten und ordnet sie der entsprechenden Verarbeitung zu:

Wird eine Nachricht empfangen, so wird zuerst die interne Position der Globalnavigation auf den Stand der internen Position der Plattform gebracht, die in jeder Nachricht, die die Globalnavigation empfangen kann, enthalten ist.

Falls eine Nachricht vom Modul Depthmap empfangen wurde, wird diese zwischengespeichert oder in der Bildanalyse verarbeitet.

Nach Verarbeitung der Nachrichten wird die Zielgenerierung aufgerufen.

Die Hauptschleife ist in der Methode *run* der Klasse *GlobalNavigation* implementiert (Algorithmus 6.1).

```
procedure GlobalNavigation.run;

 while true do
     generateTarget;

     receive (buffer, size);
     case getMsgId (buffer) of
     Msg_moveAck:
          ack := MoveAckMsg.create (buffer, size);
          updatePosition (Point (ack.x, ack.y), rad (ack.phi));
          end;
     Msg_landmarks:
          msg := LandmarkMsg.create (buffer, size);
          updatePosition (msg.robotStatus);
          analyze (LandmarkMsg.landmarks);
          end;
     fo;
od.
```

Algorithmus 6.1: Hauptschleife der Globalnavigation.

6.3.3 Verhalten und Zielgenerierung

Die Zielgenerierung untersucht zunächst, ob von der aktuellen Position aus eine Bildanforderung nötig ist. Falls nicht, werden zunächst zwischengespeicherte Bilder analysiert und danach versucht, eine lokale Bewegung zu generieren. Falls auch keine lokale Bewegung erforderlich ist, wird eine globale Bewegung generiert.

```
function GlobalNavigation.generateTarget: Boolean;

if generateLookAt then
    return true;
else
    prepareMovement;
    if generateLocalMove then
        return true;
    else
        return generateGlobalMove;
    fi;
fi.
```

Algorithmus 6.2: Zielgenerierung

Bildanforderung

Zuerst wird überprüft, ob von der aktuellen Zelle bereits ein kompletter Rundblick erstellt wurde. Falls nicht, wird der Rundblick durch eine Bildanforderung gestartet oder mit der nächsten Bildanforderung fortgesetzt.

An den Picserver werden dazu Bildanforderungen geschickt; die resultierenden Aufnahmen werden durch setzen der Weiterleitungsadresse in der Anforderung von dem Picserver automatisch an das Modul Depthmap durchgereicht: Das Navigationssystem arbeitet nicht direkt auf den Kamerabildern, sondern erhält von der Stereobildverarbeitung eine Liste von Winkel- und Tiefenangaben für Punkte, für die eine Tiefenmessung möglich war.

Ist der Rundblick für die Zelle komplett, kann aus der Karte der aktuellen Zelle ermittelt werden, ob im Nahbereich des Roboters eine Aufnahme für die Überprüfung einer fraglichen Wand notwendig ist.

Lokale Bewegung

Falls am aktuellen Standpunkt keine weiteren Aufnahmen erforderlich sind, werden zunächst eventuell gepufferte Bilddaten abgearbeitet und die Konvexität des Graphen sichergestellt.

Dann wird die aktuelle Zelle auf Wandstücke untersucht, die mindestens für die Breite des Roboters keine Merkmale aufweisen und noch nicht näher untersucht wurden. Wird mindestens ein solches Wandstück gefunden, so wird an die Lokalnavigation eine Bewegungsaufforderung in die Nähe der nächsten fraglichen Wand geschickt.

Globale Bewegung und Pfadplanung

Ist die konvexe Zelle, in der sich der Roboter aufhält, komplett erforscht, wird im Graph mit einer Breitensuche eine noch nicht vollständig untersuchte Zelle gesucht.

Für diese Zelle wird dann zunächst ein Fahrbefehl zur entsprechenden Verbindung generiert, ist der Roboter bereits bei der Verbindung, wird ein Fahrbefehl zur Durchfahrt generiert, solange bis der Roboter die Zelle erreicht hat.

Hat der Roboter die Zelle erreicht, so wird als neues Ziel eine Bildanforderung oder lokale Bewegung generiert.

6.3.4 Nachrichtenverarbeitung

Nachrichten können entweder vom Modul Depthmap ermittelte Punktkoordinaten als Antwort auf eine Bildanfrage enthalten oder nach einer Bewegung eine Positionsmeldung von der Lokalnavigation sein.

Die Verarbeitung der mit den Nachrichten übermittelten Daten ist bei der Repräsentation in Kapitel 5 beschrieben.

Kapitel 7

Visualisierung

Bei der Entwicklung der Globalnavigation stellte sich bald das Problem, daß der aufgebaute Graph auf seine Korrektheit geprüft werden sollte. Eine textuelle Ausgabe der Daten erwies sich als ineffektiv, eine graphische Visualisierung sollte eine einfachere Möglichkeit schaffen, die Operationen der Globalnavigation zu kontrollieren.

Bei einer Visualisierung für einen autonomen Roboter ergeben sich einige besondere Probleme. So möchte man bei der Entwicklung möglichst die Verarbeitungsschritte direkt mitverfolgen. Aber auch bei der autonomen Operation soll im Fehlerfall eine Analyse der Gründe, die zum Versagen geführt haben, möglich sein, ohne daß eine ständige Verbindung zum Roboter bestehen muß. Ein zusätzliches Problem bei dem Institutsroboter Arnold ist, daß keine „direkte“ graphische Ausgabemöglichkeit besteht.

Für die Visualisierung wurde ein *JAVA*[1]-Programm entwickelt, das einfache Zeichenbefehle, die über eine *Socket*-Verbindung [Ste90] empfangen oder aus einer Datei ausgelesen werden, ausführen kann. Das Visualisierungsprogramm ermöglicht mittlerweile, Befehle an den Roboter zu senden und wird auch von anderen Modulen als der Globalnavigation genutzt.

Im folgenden werden die Anforderungen an die Visualisierung und die Implementierung beschrieben. Die Dokumentation der C++-Schnittstelle findet sich im Anhang.

7.1 Anforderungen

Die Anforderungen an die Visualisierung bestanden zunächst nur darin, Strecken, Texte und Ellipsen aus einer Protokolldatei, die von der Globalnavigation erstellt wurde, darstellen zu können. Die Anforderungen wuchsen mit dem Stand der Implementierung: Die Protokolldatei erwies sich schnell als umständlich und sollte durch eine direkte Verbindung ersetzt werden. Später sollte eine Möglichkeit, innerhalb des Aufbaus der Karte „zurückblättern“ und Ausschnitte vergrößern zu können folgen; schließlich sollte über die Visualisierung auch der Roboter direkt gesteuert werden können.

Mit zunehmenden Anforderungen wurde neben dem Visualisierungsserver selbst eine Schnittstelle zur einfachen Übergabe der Zeichenbefehle erforderlich.

---

[1] Siehe http://java.sun.com

7.2 Entwurf

Wesentliche Frage beim Entwurf war, ob eine Visualisierung speziell für die Datenstrukturen der Globalnavigation oder eine allgemeinere Visualisierungsmöglichkeit entwickelt werden sollte.

Da eine spezielle Lösung auf Grundlage der Datenstrukturen der Globalnavigation nicht weniger aufwendig schien als ein allgemeines Programm, wurde die allgemeinere Lösung implementiert. Eine allgemeine Lösung besitzt auch den Vorteil, daß bei einer Änderung der Datenstruktur nur das Navigationsprogramm angepaßt werden muß und keine Änderungen an der Visualisierung notwendig sind. Ferner konnte so das Problem umgangen werden, die verzeigerte Datenstruktur zu serialisieren.

7.3 Implementierung

Die Implementierung der Visualisierung ist in einen Client- und einen Serverteil aufgeteilt. Der Server übernimmt die Fensterverwaltung und kümmert sich um die eigentliche Darstellung, während der Client-Teil die Verbindung zu einem Fenster des Servers in eine Klasse und die Zeichenbefehle in Methoden dieser Klasse kapselt.

7.3.1 Visualisierungsserver

Die Visualisierung wurde zunächst in *IDL*[1] programmiert. Mit den steigenden Anforderungen erwies sich IDL jedoch schnell wegen der fehlenden Objektorientierung als wenig geeignet: Um ein „Zurückblättern“ zu ermöglichen, mußten die Zeichenbefehle zwischengespeichert werden. Eine Neuinterpretation sollte aus Geschwindigkeitsgründen vermieden werden. In einer objektorientierten Programmiersprache können aus den Zeichenbefehlen einfach Objekte kreiert werden, die sich bei Bedarf selbst in einem Grafikkontext darstellen. Die Programmiersprache JAVA bietet moderne Sprachkonzepte und ermöglicht, die Programme auf allen relevanten Plattformen ohne neucompilierung auszuführen. Eine Entwicklungsumgebung für JAVA ist für sehr viele Betriebssysteme kostenlos erhältlich. Im folgenden werden der Ablaufrahmen einer Visualisierung und die Interaktionsmöglichkeiten des Anwenders dargestellt.

Ablaufrahmen

Beim Start der Visualisierung wird ein Socket-Serverport geöffnet. Der Visualisierungsserver wartet dann auf eingehende Verbindungen. Für jede Verbindung wird ein Fenster geöffnet. Über die Verbindung eingehende Zeichenbefehle werden gespeichert und in dem entsprechenden Fenster umgesetzt. Neben den Zeichenbefehlen gibt es einen Befehl, um eine neue Seite zu erzeugen. Wird eine neue Seite angefordert, so wird der Bildschirminhalt vor der Umsetzung neu eintreffender Zeichenbefehle gelöscht, nicht jedoch die abgespeicherten Zeichenbefehle. Bei Bedarf kann der Nutzer mit der Visualisierung auf alte Seiten zurückblättern, um zum Beispiel die Vorbedingungen eines fehlerhaften Aufbaus eines Graphen zu überprüfen.

---

[1] IDL: Interactive Data Language

Interaktionsmöglichkeiten

Neben der schon vorgestellten Möglichkeit, in einem Fenster Seiten vor- oder zurückzublättern, können beliebige Bereiche der Grafik vergrößert werden, indem die Maustaste in der oberen linken Ecke gedrückt wird und mit gehaltener Maustaste dann der zu vergrößernde Bereich markiert wird. Wird die Maustaste losgelassen, so wird in dem aktuellen Fenster nur der markierte Bereich – entsprechend vergrößert – dargestellt. Bei Bedarf können aus diesem Bereich wiederum Ausschnitte vergrößert werden; mit der Taste „Zoom out“ gelangt man zur jeweils vorhergehenden Vergrößerungsstufe zurück.

Der aktuelle Bildschirminhalt kann mit der Taste „Print“ ausgedruckt werden.

Ein Visualisierungsclient kann zusätzlich noch eigene Tasten definieren, die Anwendungsspezifische Funktionen haben. So kann beispielsweise zum Test der Globalnavigation der Roboter an einen zuvor durch einen Mausklick definierten Punkt geschickt werden, um dort einen neuen Rundblick zu erstellen.

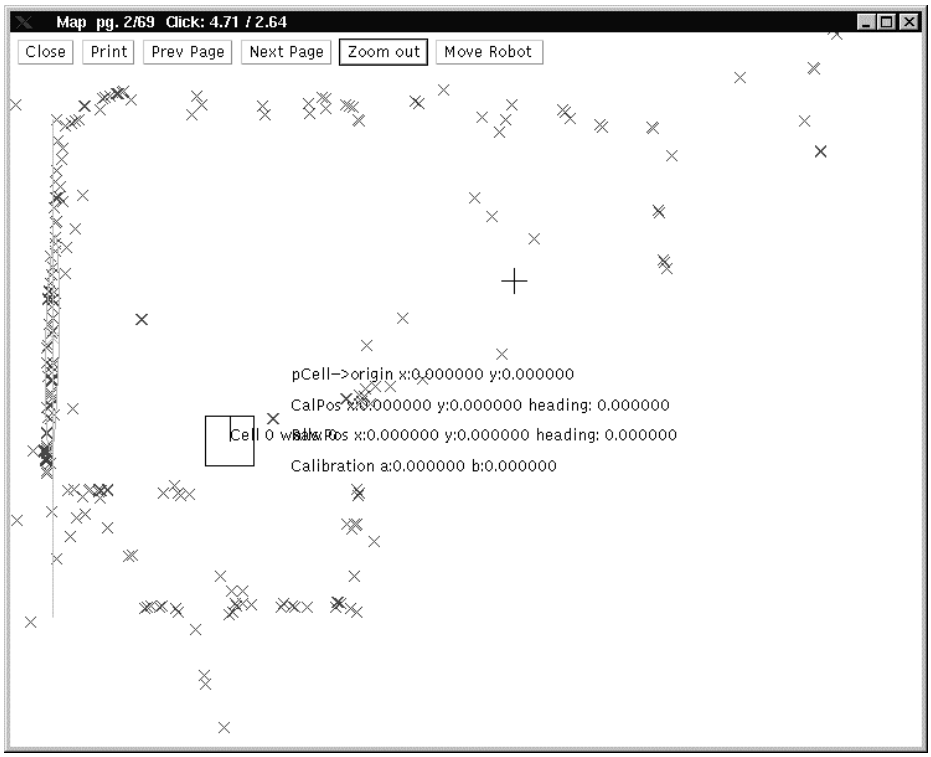


Abbildung 7.1: Ein Fenster der Visualisierung

7.3.2 Visualisierungsclient

Programme, die den Visualisierungsserver für graphische Ausgaben nutzen, müssen nicht die Zeichenbefehle selbst über eine Socketverbindung verschicken, sondern können dazu die Zeichenbefehle der Klasse *Canvas* nutzen, die dem Aufbau der Verbindung zum Server und die Codierung der Zeichenbefehle kapselt. Die Klasse *Canvas* nutzt zur Parameterübergabe einige Hilfsklassen für Konzepte wie Punkte und Linien.

Die Klasse *Point* wurde bereits bei der ersten Repräsentation vorgestellt. Sie wird bei der Definition von Markerpositionen, von Rahmen für die Darstellung von Bitmaps und für Linienstützpunkte benutzt.

Zur Darstellung von Linien kann auch die Klasse *Line* benutzt werden, die ebenfalls bei der ersten Repräsentation vorgestellt wurde.

Bitmaps werden als „*unsigned char*“ typisierte Template-Klasse *Matrix* übergeben. So kann beispielsweise die in der erweiterten Repräsentation benutze Hough-Matrix einfach ausgegeben werden.

Eine detaillierte Beschreibung der Klassen der Visualisierung findet sich im Anhang.

Kapitel 8

Ergebnisse

In diesem Kapitel werden die Ergebnisse der beiden implementierten Umweltrepräsentationen vorgestellt.

Während die Resultate der ersten Umweltrepräsentation hauptsächlich auf Simulationen beruhen, wurde die erweiterte Umweltrepräsentation während der gesamten Entwicklung mit Realdaten getestet.

Um die positiven Ergebnisse der erweiterten Repräsentation zu bekräftigen, ist es notwendig, die baulichen und technischen Voraussetzungen für eine Bewegung des Roboters über den Testraum hinaus zu schaffen.

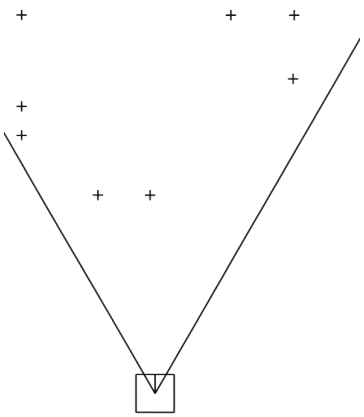
8.1 Erste Umweltrepräsentation

Die Fehlertoleranz der ersten Umweltrepräsentation wurde für eine konvexe Zelle bereits von Daniel Tepas untersucht [Tep97].

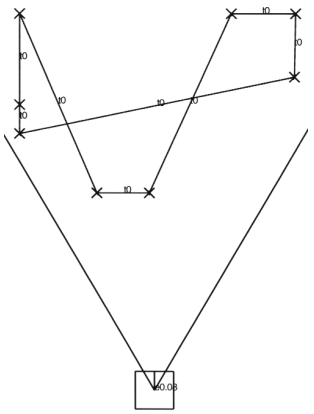
Im Rahmen dieser Arbeit wurde der Aufbau der Karte mit Daten ohne künstliche Fehler über eine Zelle hinaus simuliert. Der einwandfreie Kartenaufbau für eine Testumgebung und das dafür notwendige Verhalten des Roboters konnte bis zur Verfügbarkeit von Realdaten sichergestellt werden, wie in Abbildung 8.1 und Abbildung 8.2 dargestellt.

Bei Verfügbarkeit der Realdaten wurde das vorhandene Modell mit diesen Daten getestet. Dabei schien eine Zuordnung einzelner Punkte, auf der das Umweltmodell beruht, hoffnungslos (siehe Kapitel 5).

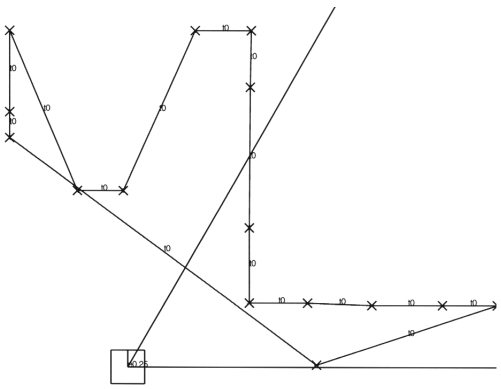
Der geplante nächste Schritt, eine weitergehende Simulation mit ähnlich den Realdaten verrauschten Daten, wurde daher zugunsten der Einführung einer zusätzlichen Abstraktionsebene verworfen.



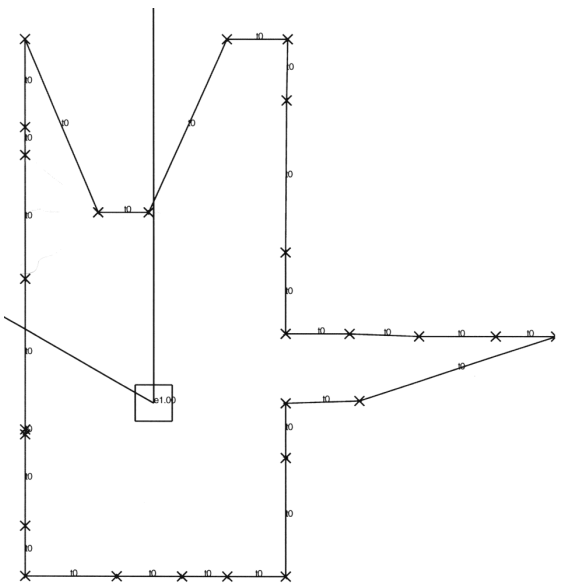
Vom Roboter werden simulierte Punkte gesehen.



Die Punkte werden in die aktuelle Zelle eingetragen. Wände ergeben sich implizit durch Verbindung der Punkte im Uhrzeigersinn.

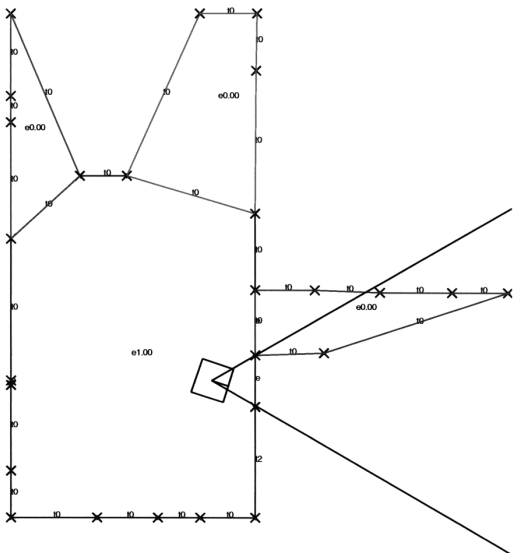


Der Kamerakopf wird gedreht, um den Rundblick zu vervollständigen.

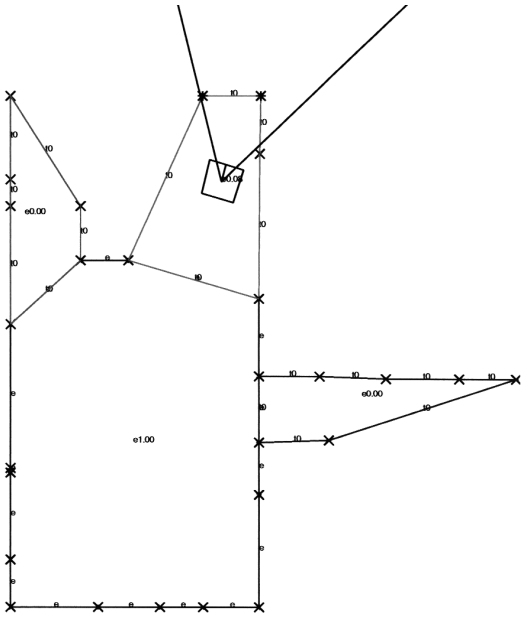


Alle Aufnahmen des ersten Rundblicks sind in der Datenstruktur eingetragen.

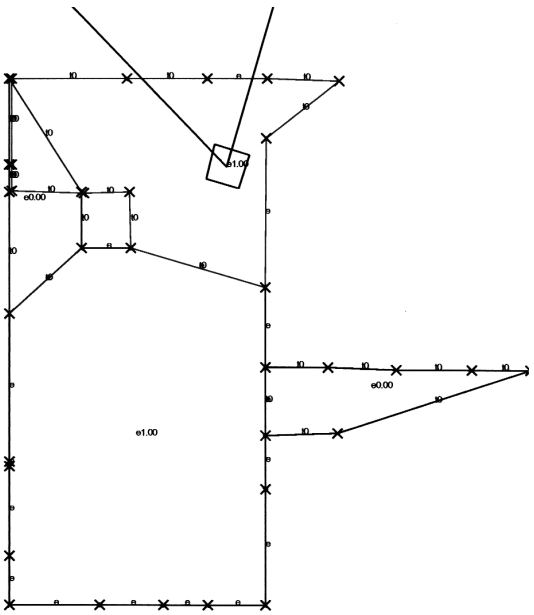
Abbildung 8.1: Erster Rundblick in einer Simulationsumgebung



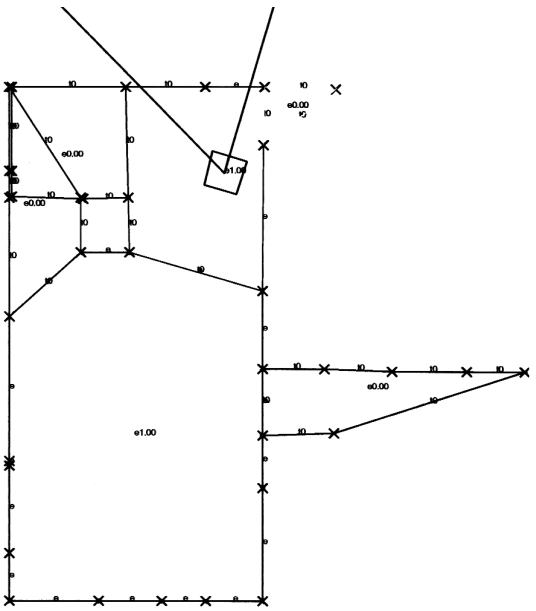
Nach Vervollständigung des Rundblicks wird die Konvexität hergestellt und die ursprüngliche Zelle in konvexe Teilzellen zerlegt. Der Roboter untersucht gerade die Verbindungsstrecke zweier Merkmale auf eine Diskontinuität.



Nach vollständiger Überprüfung der ersten Zelle hat sich der Roboter in die nächste Zelle bewegt.



Graph nach einem vollständigen Rundblick in der zweiten besuchten Zelle.



Die zweite Zelle ist in konvexe Teilzellen zerlegt.

Abbildung 8.2: Zerlegung der Simulationsdaten in konvexe Zellen und Erkundigung der weiteren Umgebung

8.2 Erweiterte Umweltrepräsentation

Die erweiterte Umweltrepräsentation wurde von Anfang an mit realen Daten getestet.

Die Globalnavigation konnte dabei für einen Testraum von verschieden Standpunkten aus ein korrektes Umweltmodell für den sichtbaren Bereich erstellen; Abbildung 8.5 und folgende zeigen die aus den Rundblicken von verschiedenen Standpunkten aus erstellten Karten des Testraumes.

Die Laufzeiten für die Hough-Transformation lagen bei der Auswertung kompletter Rundblicke ohne Belastung durch andere Prozesse – abhängig von den Bildern – zwischen 0,7 und 1,6 Sekunden (Tabelle 8.1). Im Vergleich zu der Zeit, die zur Aufnahme der Bilder und für den Stereoalgorithmus benötigt wird, ist diese Zeit vernachlässigbar. Bei der verwendeten Implementierung werden dabei noch zur Vereinfachung direkt die vom Stereoverfahren gelieferten Fließkommazahlen für die Gewichte der Punkte benutzt. Bei der Verwendung von Ganzzahlen ist eine weitere Senkung der Laufzeit zu erwarten.

Für alle getesteten Standpunktkombinationen war dabei eine korrekte globale Selbstlokalisation (5.2.4) auch bei vollständig falscher Odometrieinformation möglich.

Da der Bewegungsbereich des Roboters zur Zeit noch auf den Testraum beschränkt ist und für die neue Repräsentation keine Simulation zur Generierung geeigneter Daten entwickelt wurde, konnte das Verhalten zum Kartenaufbau mit der neuen Repräsentation nicht hinreichend getestet werden. Da das Verhalten jedoch in einer Simulation mit der ersten Umweltrepräsentation erfolgreich getestet wurde, sind keine besonderen Probleme zu erwarten.

Analysierte Punkte	Gefundene Geraden	Laufzeit / s
119	3	0,68
224	5	1,96*
235	5	0,99
274	5	1,39
285	7	1,56
310	7	1,38

Tabelle 8.1: Für verschiedene Standpunkte gemessene Laufzeiten. Die zweite Messung (*) wurde unter Belastung durch andere Prozesse ermittelt.

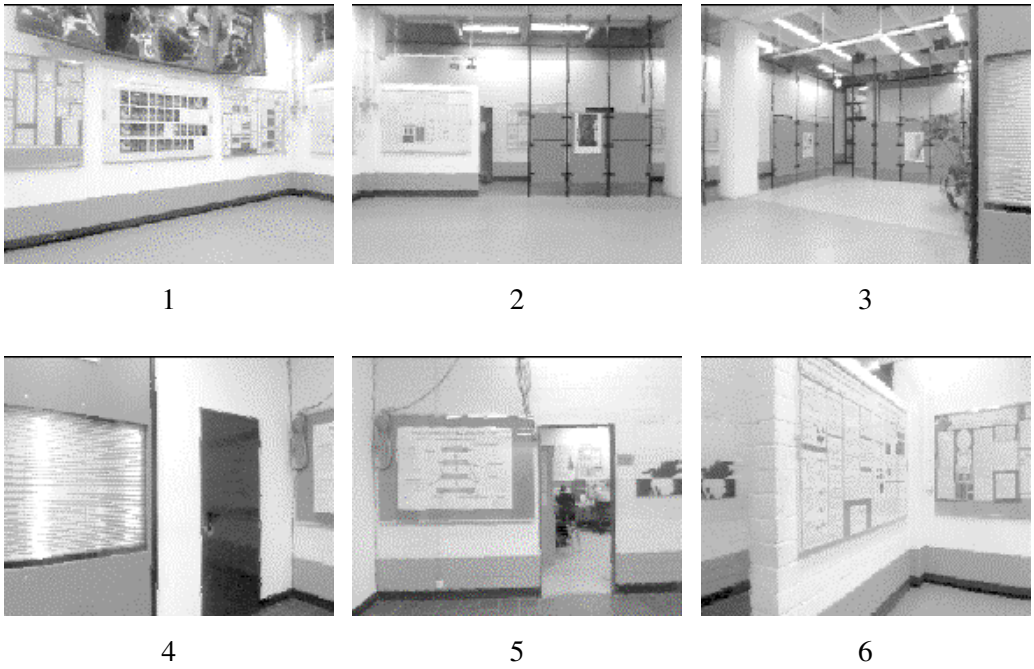


Abbildung 8.3: Mit einer Peripheriekamera des Roboters in 60°-Schritten aufgenommener Rundblick im Testraumes. Im dritten Bild sind rechts einige Pflanzen zu sehen, die beim Stereoverfahren ein starkes Rauschen erzeugen.

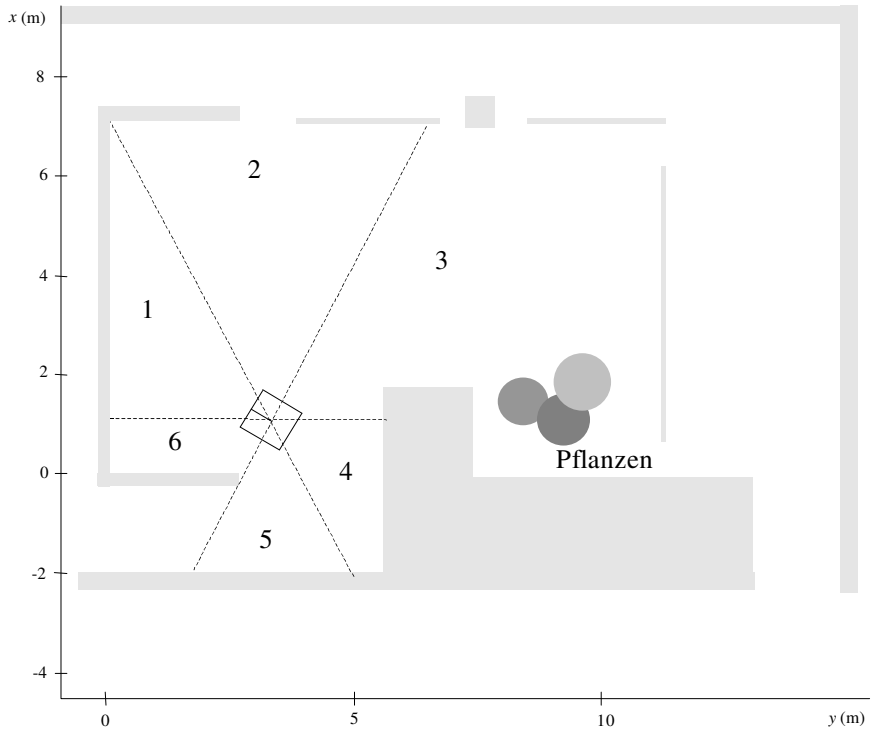


Abbildung 8.4: Grundriss des Testraumes. Die Ziffern entsprechen der Numerierung der Bilder aus Abbildung 8.3.

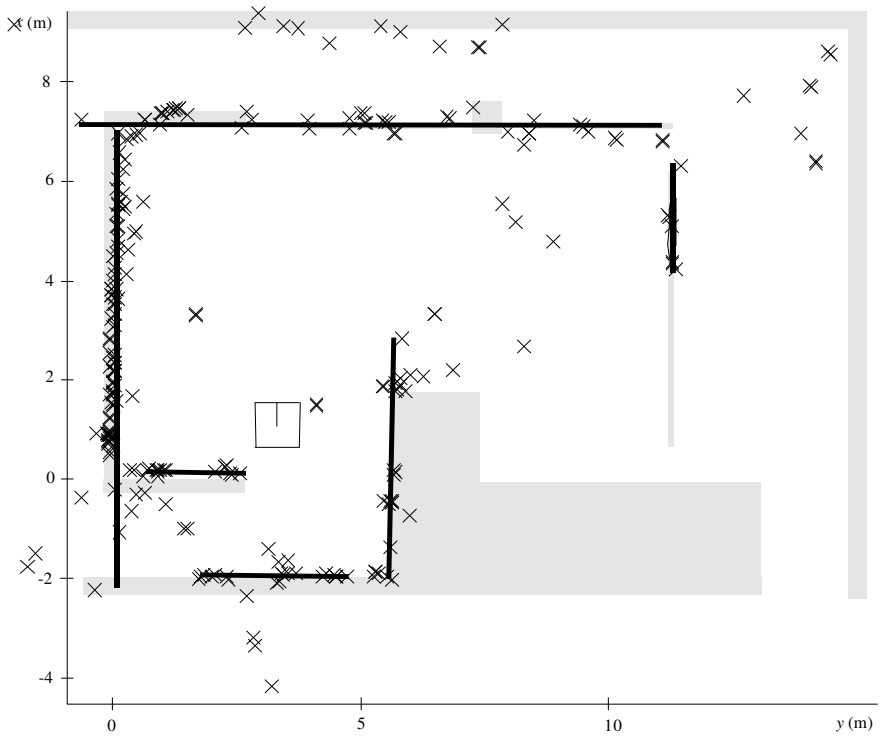


Abbildung 8.5: In dem Testraum erstellter Rundblick und gefundene Geraden. Die dargestellte Länge der Geraden entspricht dem Bereich, in dem Punkte gefunden wurden. Prinzipiell sind die Längen nicht beschränkt. Für eine konvexe Zelle werden die Geraden jeweils bis zu den Schnittpunkten betrachtet.

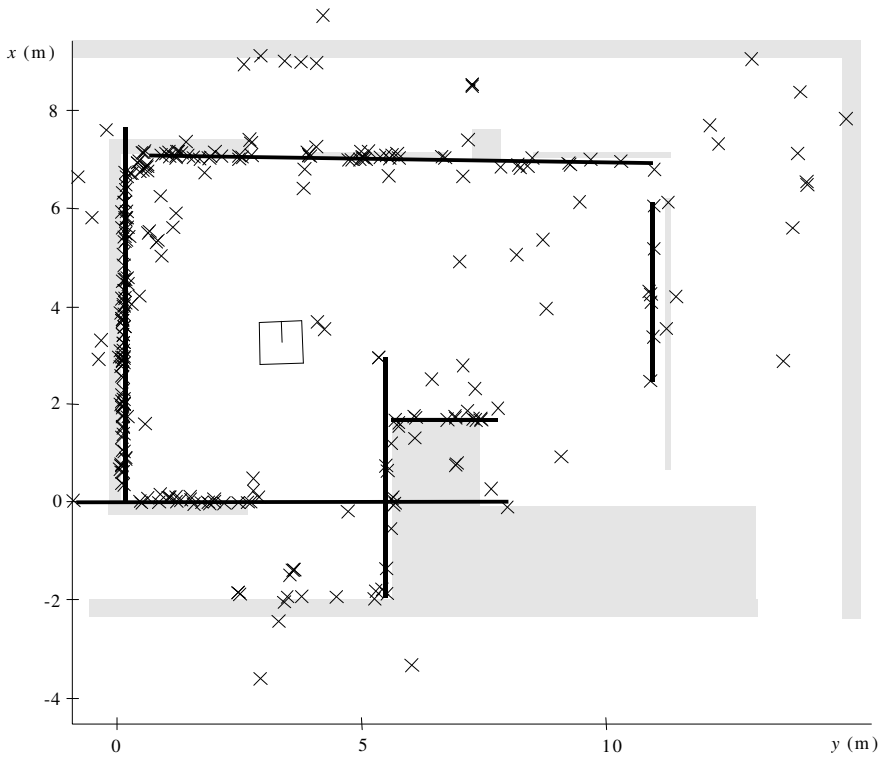


Abbildung 8.6: Von einer zweiten Position werden zum größten Teil die gleichen Geraden gefunden.

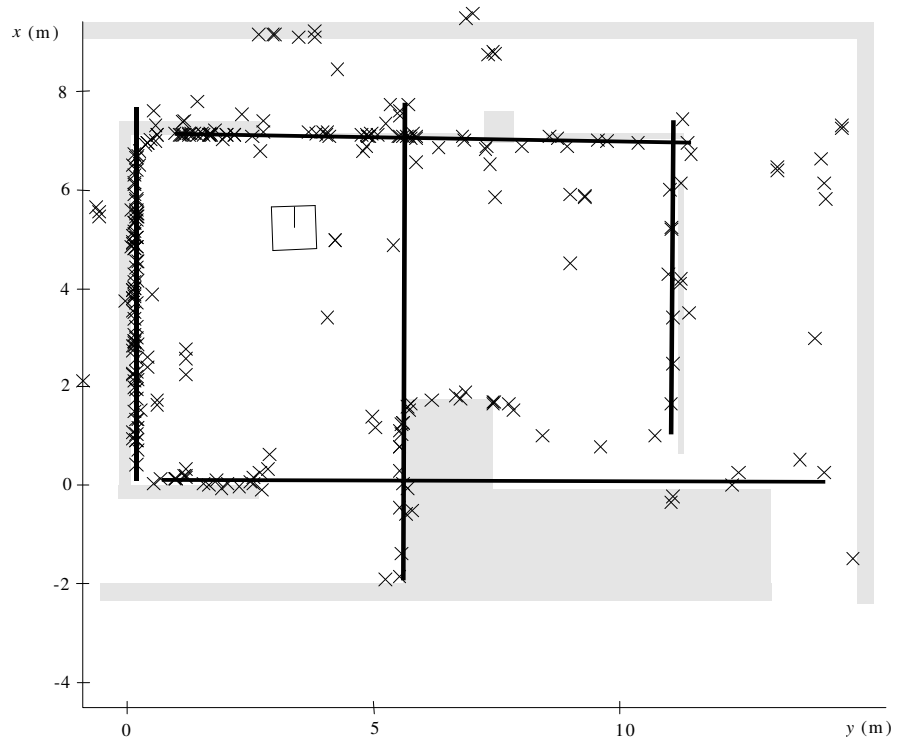


Abbildung 8.7: Weiterer Testläuf auf der linken Seite des Testraumes. Der Speicheraufand für die Geraden beträgt zur Zeit pro diskretisiertem Teilstück – also je 10 cm – Speicherplatz für eine Fließkommazahl, die das summiertes Gewicht speichert und für eine boolesche Variable, die die Passierbarkeit repräsentiert. Werden dafür pro Wandposition 12 byte berechnet, ergibt sich für das Bild ein Speicherverbrauch von ca. 6,3 kilobyte.

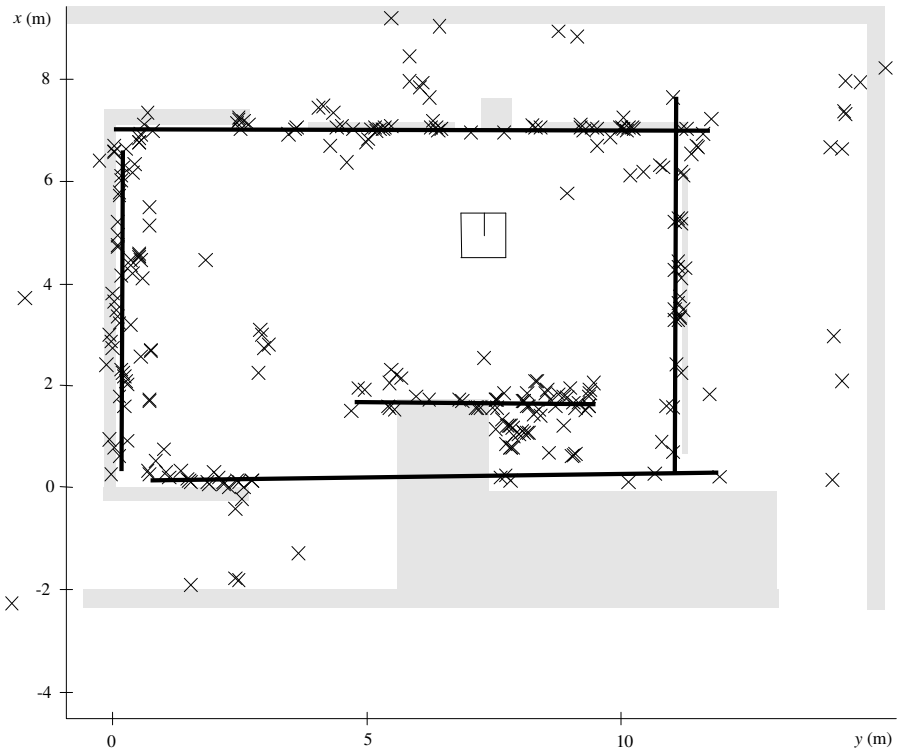


Abbildung 8.8: Auf der rechten Seite des Testraumes ist zu erkennen, daß die Pflanzen starke Störungen verursachen.

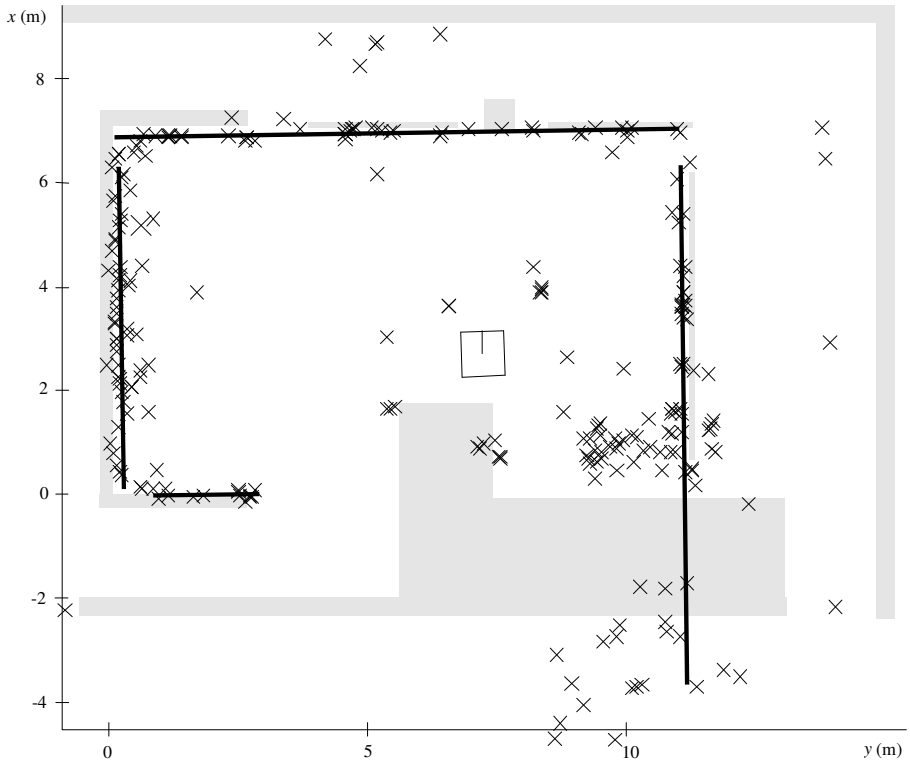


Abbildung 8.9: Weiterer Testlauf auf der rechten Seite des Raumes. Unten rechts sind wieder deutlich Störungen durch Pflanzen zu sehen. Die Pflanzen erzeugen viele Punkte, aber keine eindeutige Geraden. Obwohl die Entfernung zur linken Wand und damit die Fehlerstreuung der Punkte erheblich zugenommen hat, wird die Wand noch gut erkannt.

Kapitel 9

Diskussion

In diesem Kapitel wird das entwickelte Navigationsverfahren mit bestehenden Arbeiten verglichen. Außerdem wird auf den Bezug zur Biologie eingegangen.

9.1 Vergleich mit anderen Arbeiten

Die hier entwickelte Umweltrepräsentation greift auf verschiedene Konzepte bestehender Navigationsverfahren zurück. So werden innerhalb einer konvexen Zelle relativ genaue geometrische Informationen gespeichert – wie auch in anderen Verfahren mit Streckenrepräsentationen (siehe 3.2.1). Für die Wände an sich existiert eine eindimensionale Struktur, die mit den zweidimensionalen Bitmaps bestehender Verfahren in einigen Punkten vergleichbar ist (siehe 3.1). Schließlich werden Verbindungsinformationen zwischen den konvexen Zellen gespeichert, ähnlich zu den Topologie-Repräsentationen (siehe 3.2.2).

Auch die Trennung zwischen Global- und Lokalnavigation wurde beispielsweise schon in [BBC+95] realisiert.

Einige besondere Vorteile der hier entwickelten Repräsentation resultieren direkt aus der Kombination: Die einfache und genaue Rekalibration ist hauptsächlich durch die Geometrieinformationen möglich, während die Pfadplanung von der Aufteilung in konvexe Zellen und den Topologieinformationen profitiert.

Ein weiterer Vorteil ist, daß fehlende Sensorinformationen nicht unbedingt als freier Fahrraum interpretiert werden. So werden bei erkannten Wänden Merkmalsfreie Flächen erst dann als Diskontinuität gespeichert, wenn ein Beweis in Form von dahinter sichtbaren Merkmalen gefunden wird.

9.2 Bezug zur Biologie

> *Tagaktive Primaten sind „Augentiere". Mindestens 60% ihrer Großhirnrinde sind retinotop organisierte „elementare" visuelle Felder, visuelle Assoziationsregionen oder visuell-okulomotorische Integrationsregionen. ([GG95], S. 299)*

Biologisches Vorbild für einen „Maschinenmenschen" ist naheliegenderweise der Mensch selbst. Jedoch besteht allein der visuelle Kortex des Menschen aus einer Anzahl von Neuronen im Bereich von 10^{10} [HW79]. Heutige künstliche neuronale Netzen liegen mit bis zu 10^5 Neuronen um einige Größenordnungen darunter, wobei nicht ganz sicher ist, daß aktuell eingesetzte Simulationen alle funktionskritischen Aspekte natürlicher Neuronen abdecken [Ham97].

Unter der Annahme, daß die Biologie durch die Evolution für ihre Rahmenbedingungen nahezu optimale Strukturen hervorbringt, also insbesondere Strukturen, die bei annähernd gleicher Leistung nicht durch signifikant einfachere Strukturen zu ersetzen sind, erscheint es fragwürdig, ein Verfahren mit einem relativ hohem Abstraktionsgrad wie die Globalnavigation direkt mit einer Simulation einzelner Neuronen realisieren zu wollen.

Allerdings besteht die Möglichkeit, daß in der Biologie mangels anderer Möglichkeiten Verfahren mit neuronalen Netzen realisiert werden, die anders auch auf heutigen Rechnern effizient implementierbar sind, oder daß relevante Teile des zusammengefaßten Verhaltens einer Menge von Neuronen sich effizienter simulieren lassen als eine Betrachtung auf Zellebene.

Das in dieser Arbeit entwickelte Verfahren zeichnet sich zunächst sicherlich nicht durch eine besondere Nähe zur Biologie aus. Der entscheidende Schritt zur Funktionsfähigkeit des Verfahrens war die Einführung der Geraden als zusätzliche Abstraktionsebene zwischen den Punkten, die das Stereoverfahren liefert, und der eigentlichen Karte. Diese durchgeführte Vorverarbeitung der Daten, die Geometrieinformationen über Wände liefert, kann jedoch wiederum einen interessanten Ansatzpunkt für die Implementierung eines biologienäheren Verfahrens liefern: Beruht in der Biologie die Wiedererkennung von Plätzen wahrscheinlich auf der Wiedererkennung von Ansichten [Pou93], heißt das noch nicht, daß diese Ansichten sehr direkt repräsentiert sein müssen. So ist eine der Wiedererkennung von Ansichten vorgeschaltete Geometrieerkennung denkbar, um beispielsweise die im sichtbaren Bereich vorhandenen Wandtexturen möglichst invariant zu extrahieren. Insbesondere muß eine Geometrieerkennung in der Biologie auf jeden Fall für andere Zwecke geleistet werden [Gus96].

Es bleibt zu hoffen, daß die Erkenntnisse aus der direkten biologischen Forschung und aus Nachbildungsversuchen zu einer Konvergenzbewegung führen, die ein immer besseres Verständnis der Abläufe im Gehirn ermöglichen.

Kapitel 10

Zusammenfassung und Ausblick

In diesem Abschnitt wird das entwickelte Roboternavigationssystem kurz zusammengefaßt und es werden einige interessante Erweiterungsmöglichkeiten aufgezeigt.

10.1 Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde ein in einer Vorarbeit entwickeltes und teilweise simuliertes Umweltmodell auf einen realen Roboter übertragen. Da die simulierte Repräsentation sich in Versuchen als für Realdaten ungeeignet erwies, wurde ein erweitertes Umweltmodell entwickelt. Durch Einführung einer neuen Abstraktionsebene konnte mit dieser Repräsentation unter Beibehaltung der Vorteile des ersten Umweltmodells ein Testraum reproduzierbar von beliebigen Punkten aus korrekt kartiert werden. Zusätzlich erlaubt das neue Verfahren eine globale Selbstlokalisation. Eine im Rahmen dieser Arbeit für das Navigationsverfahren entwickelte Visualisierung wird mittlerweile auch bei der Entwicklung der Lokalnavigation und der Armkontrolle eingesetzt.

10.2 Ausblick

Einige Erweiterungsmöglichkeiten, die aus der in Teilen prototypischen Implementierung resultieren, werden bereits in Kapitel 5 aufgezeigt. Besonders interessante Möglichkeiten, auf die im folgenden näher eingegangen wird, bieten die Wanddarstellung und die Benutzung weiterer Eingangsdaten.

Daneben sind noch andere Erweiterungen vorstellbar, wie die Implementierung einer Selbst-Rekalibration für die Vergenz der Kameras, die sich in den Versuchen sehr leicht verstellte oder eine Erweiterung der Datenstruktur, um mehrere Ebenen eines Gebäudes darstellen zu können.

Für die Visualisierung wäre eine nähere Anlehnung an einen bestehende Grafikstandard, beispielsweise GKS[1] und CGM[2] zur Datenübertragung denkbar, insbesondere wenn die Visualisierung zur Beschleunigung auch Daten im Binärformat übertragen können soll.

---

[1] Graphical Kernel System [ISO85] (zitiert nach [Fel92])

[2] Computergraphics Metafile [ISO87] (zitiert nach [Bor91])

10.2.1 Wanddarstellung

Bisher wird in der Datenstruktur nur vermerkt, ob an einer diskretisierten Position auf einer Wand ein Merkmal gesehen wurde oder nicht.

Zusätzlich kann die gesamte Wand jedoch mit aus den aufgenommenen Bildern extrahierten Texturdaten belegt werden, was die Möglichkeit schafft, die Daten des Raumes dreidimensional zu visualisieren. Außerdem wäre vorstellbar, die in anderen Verfahren eingesetzte Landmarkenerkennung für diese Daten einzusetzen, wobei durch die bestehende Erkennung von Geometrieinformationen direkt vergleichbare Bilder ohne Verschiebungs- oder Skalierungsprobleme zu erhalten wären.

Abbildung 10.1 zeigt ein Originalbild und an den vom Stereoverfahren gelieferten Punkten extrahierte lokale Texturdaten, die auf zwischen den Merkmalen liegende Flächen interpoliert wurden. Zu erkennen ist, daß die lokalen Texturen nahe beieinander liegender senkrechter Kanten in dem Bild aus der ersten Repräsentation zu ähnlich sind, um die Identifikation einzelner Kanten signifikant zu verbessern. Für den Vergleich größerer Flächen könnten die Texturdaten jedoch durchaus dazu beitragen, Unsicherheiten bei der globalen Selbstlokalisation zu beseitigen – beispielsweise in einem Raum mit 180° drehsymmetrischem Grundriß.

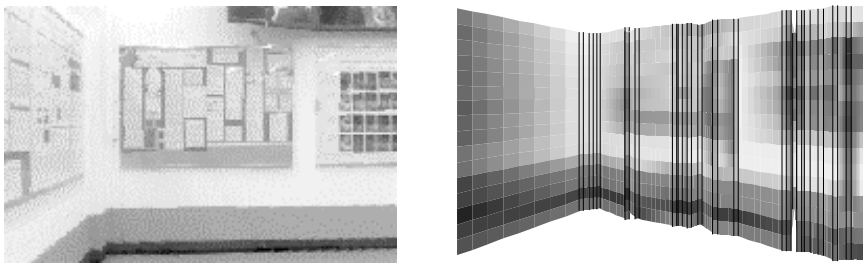


Abbildung 10.1: Mit der ersten Repräsentation wurden schon Versuche zur Texturextraktion gemacht, allerdings wurden nur Texturdaten in der unmittelbaren Umgebung der extrahierten vertikalen Kanten gespeichert und die Flächen interpoliert. Das linke Bild ist eine Aufnahme einer Kamera des Roboters, das rechte Bild ist aus der entsprechenden Stelle der Karte (um einige Grad gedreht) der ersten Repräsentation erstellt.

10.2.2 Inverse Perspektive

Für den Kartenaufbau können neben den Daten des Stereomoduls auch Daten aus der *inversen Perspektive*, die zwischenzeitlich für die Lokalnavigation implementiert worden ist, einbezogen werden. Die invers perspektivische Abbildung wird zur Detektion von Objekten, die aus der Bodenebene herausragen, verwendet. Dazu wird aus dem linken Kamerabild für die Annahme, daß alle Punkte in der Bodenebene liegen, ein „theoretisches“ rechtes Kamerabild berechnet. Differenzen zwischen aufgenommenem und berechnetem rechten Kamerabild lassen auf Punkte schließen, die nicht in der Bodenebene liegen, also Hindernisse darstellen.

Neben der höheren Robustheit bei der Verwendung von Daten aus zwei verschiedenen Verfahren ist die inverse Perspektive unter dem Aspekt, daß sich die Entwicklung des Roboters an biologischen Vorbildern orientieren soll, interessant:

Da auch einäugige Menschen sich in der Umwelt relativ problemlos zurechtfinden [Gus96], kann Stereosehen nicht die einzige Informationsquelle für Tiefeninformationen für das Gehirn sein. Zwar nutzt auch die inverse Perspektive Stereoinformationen, jedoch ist die Art der Aus-

wertung sicherlich schon etwas näher an einer geometrischen Bildauswertung (monokularer Bilder) orientiert als die Zuordnung von Punkten im Stereoverfahren.

Anhang A

Positionsunsicherheit in der ersten Repräsentation

Im folgenden wird die genaue Berechnung der Gauß-verteilten Ellipse zur Speicherung der Positionsunsicherheit der Landmarken aus der ersten Umweltrepräsentation dargestellt.

Die konstruktive Form einer Gauß-verteilten Zufallsfunktion mit elliptischer Grundfläche lautet:

$$G(x, y) = M \cdot e^{-(A(x-\mu_x)^2+B(y-\mu_y)^2+2C(x-\mu_x)(y-\mu_y))} \tag{A.1}$$

Dabei legen die Koordinaten μ_x und μ_y das Zentrum der Verteilung fest, A, B, C bestimmen die Ausdehnung und M die Wahrscheinlichkeit am Ellipsenmittelpunkt.

Für μ_x und μ_y werden die vom Stereosystem gelieferten Koordinaten eingesetzt, die restlichen Parameter werden aus dem Distanzfehler Δ_d, dem Winkelfehler σ_α und dem Winkel θ, in dem die Landmarke zu sehen ist, berechnet:

$$\gamma_1 = \frac{1}{2}\left(\frac{1}{\left(\frac{1}{2}\Delta_d\right)^2} + \frac{1}{{\sigma_\alpha}^2}\right) \quad A = \frac{\gamma_1 + \gamma_2\cos(2\theta)}{2}$$

$$\gamma_2 = \frac{1}{2}\left(\frac{1}{\left(\frac{1}{2}\Delta_d\right)^2} - \frac{1}{{\sigma_\alpha}^2}\right) \quad B = \frac{\gamma_1 - \gamma_2\sin(2\theta)}{2} \tag{A.2}$$

$$M = \frac{1}{2\pi\sqrt{\frac{1}{{\gamma_1}^2 - {\gamma_2}^2}}} \qquad C = \frac{\gamma_2\cos(2\theta)}{2}$$

Anhang B

Belegung der Konstanten

In der folgenden Tabelle B.1 sind die Konstantenbelegungen aufgeführt, die in der Globalnavigation verwendet werden.

Name	Wert	Bedeutung
	6:3:1	Gewichtung von Winkelübereinstimmung zu Entfernungsübereinstimmung zu Übereinstimmung des Attributvektors bei dem Vergleich zweier Wände.
AlignFactor	3	Faktor für die Gewichtung von Geraden, die bei der Hough-Transformation parallel oder orthogonal zur Hauptrichtung liegen.
Angles	360	Anzahl der Winkel-Diskretisierungsschritte bei der Hough-Transformation. 360 Schritte ergeben eine Winkelauflösung von einem Grad.
CornerTolerance	10°	Toleranz für die Übereinstimmung zweier Ecken bei der globalen Selbstlokalisation.
Distances	50	Anzahl der diskreten Bereiche, in die *MaxDistance* bei der Hough-Transformation aufgeteilt wird. Zusammen mit *MaxDistance* = 10 m ergibt sich eine Auflösung von 20 cm.
MaxDistance	10	Maximale Distanz (in m) zu einer Geraden vom Roboterstandpunkt bei der Hough-Transformation
MinFitGlobalRecalibration	10°	Minimale Winkelübereinstimmung bei der globalen Rekalibration.
MinMatch	0,9	Minimale Übereinstimmung zweier Wände, damit sie bei dem Vergleich von aktuell gesehenen Wänden und Erwartungsbild einander zugeordnet werden (Der Übereinstimmungswert für identische Wände beträgt 1,0).
MinRadius	0,5	Mindestmaß für den größeren der beiden Radien der Gauß-Ellipse, damit nach der Hough-Transformation ein Geradensegment akzeptiert wird.

Name	Wert	Bedeutung
MinRadiusFactor	4	Mindestmaß für den Faktor zwischen kleinem und großem Radius der Gauß-Ellipse nach der Hough-Transformation, damit ein Geradensegment akzeptiert wird.
MinWeight	10	Minimales summiertes Gewicht einer Geraden, damit sie bei der Hough-Transformation akzeptiert wird.
WallStep	0,1	Diskretisierung des Attributvektors einer Wand: Je 0,1 m existiert ein Eintrag.

Tabelle B.1: Konstantenbelegungen

Anhang C

Dokumentation der Visualisierungsschnittstelle

C.1 Klasse Canvas

Die Klasse *Canvas* kapselt die Übermittlung der Zeichenbefehle an den Visualisierungsserver und stellt dem Nutzer übersichtliche Methoden zur Verfügung.

Farben werden als 32-Bit-Integer übergeben, dessen niedrigstes Byte die Intensität für Blau angibt, das zweitniedrigste Byte den Grünwert und das nächste Byte die Rot-Intensität. Das höchste Byte ist für Transparenzinformationen reserviert.

Konstruktor der Klasse Canvas

```
Canvas (char* title, Point pmin, Point pmax);
```

Versucht eine Socketverbindung zum Java-Server aufzubauen. Falls dies nicht gelingt, wird im aktuellen Verzeichnis eine Ausgabedatei mit dem Dateinamen „canvas?.plt“, wobei das Fragezeichen für die Fensternummer steht.

Die Parameter bestimmen den Fenstertitel und den maximalen Bereich der Koordinaten.

Methoden der Klasse Canvas

addButton

```
void addButton (int aId, char *title);
```

Fügt eine Taste mit der Beschriftung *title* in das Visualisierungsfenster ein. Der Identifikationscode *aId* wird bei *getEvent* zurückgeliefert, wenn die Taste gedrückt wurde.

clear

```
void clear ();
```

Löscht den aktuellen Anzeigepuffer.

newPage

```
void newPage ();
```

Erzeugt eine neue leere Seite.

ellipse

```
void ellipse (Point p, double angle,
              double r1, double r2);
```

Zeichnet eine Ellipse mit Mittelpunkt *p*, Winkel der Hauptachse *angle* und den Radien r_1 und r_2.

line

```
void line (Line l);
```

Zeichnet die übergebene Linie.

line

```
void line (Point p0, Point p1);
```

Zeichnet eine Linie von p_0 nach p_1.

lineTo

```
void lineTo (Point p);
```

Zeichnet eine Linie von der aktuellen Stiftposition nach *p*. Die neue Stiftposition wird *p*.

bitmap

```
void bitmap (Point p0, Point p1,
             Matrix<unsigned char>& img);
```

Zeichnet die übergebene Bitmap in das durch p_1 und p_2 definierte Rechteck. Die Werte der Matrix werden als Grauwert von Schwarz (0) bis Weiß (255) ausgegeben (siehe Methode *getImage* der Klasse *Matrix*).

bitmap3D

```
void bitmap3D (Point p1, Point p2, Point p3, Point p4,
               int w, int h, int* data);
```

Zeichnet die in *data* übergebene RGB-Bitmap der Höhe *h* und Breite *w* in das durch $p_1 - p_4$ definierte Viereck.

eventAvailable

```
bool eventAvailable ();
```

Überprüft, ob ein Ereignis vom Visualisierungsserver vorliegt. Das Ereignis muß mit *getEvent* ausgelesen werden.

getEvent

```
CanvasEvent getEvent ();
```

Liest ein Ereignis aus; der Aufruf wartet, bis ein Ereignis verfügbar ist. Mit *eventAvailable* kann abgefragt werden, ob ein Ereignis vorliegt.

paint

```
void paint ();
```

Erzwingt die Ausgabe des aktuellen Grafikpuffers, ohne den Bildschirm zu löschen.

repaint

```
void repaint ();
```

Löscht den Bildschirm und gibt den aktuellen Grafikpuffer aus.

setLineColor

```
void setLineColor (int i);
```

Setzt die aktuelle Linienfarbe.

setMark

```
void setMark (Point p);
```

Zeichnet eine Markierung an die übergebene Position.

setMarkColor

```
void setMarkColor (int i);
```

Setzt die Farbe für die Markierungen.

setMarkStyle

```
void setMarkStyle (int i);
```

Setzt den Typ der Markierungen. Mögliche Übergabewerte sind: *MarkerCross*, *MarkerAsterix*, *MarkerPoint*, *MarkerDiamond*, *MarkerTriangle*, *MarkerSquare* und *MarkerX*.

textOut

```
void textOut (Point p, char* txt);
```

Zeichnet die Textzeile *txt* an die Position *p*.

C.2 Klasse CanvasEvent

Die Klasse *CanvasEvent* speichert Informationen über Ereignisse vom Visualisierungsserver.

Variablen der Klasse CanvasEvent

type

```
char type;
```

Typ des Events. Mögliche Werte sind *CanvasMouseClick* und *CanvasButton*.

id

```
int id;
```

Identifikation des auslösenden Elementes.

cursorX

```
double cursorX;
```

x-Position des letzten Mausklicks.

cursorY

```
double cursorY;
```

y-Position des letzten Mausklicks.

C.3 Klasse Point

Die Klasse *Point* dient dem einfachen Umgang mit Punkten im zweidimensionalen Raum

Variablen der Klasse Point

x

```
double x;
```

Speichert die *x*-Koordinate des Punktes.

y

```
double y;
```

Speichert die *y*-Koordinate des Punktes.

Konstruktoren der Klasse Point

```
Point (void);
```

Initialisiert *x* und *y* mit 0.

```
Point (double ax, double ay);
```

Initialisiert *x* mit *ax* und *y* mit *ay*.

Methoden der Klasse Point

angle

```
double angle (Point p2);
```

Berechnet den Winkel zum Punkt p_2.

distance

```
double distance (Point p2);
```

Berechnet die Distanz zum Punkt p_2.

+

```
Point operator+ (Point p2);
```

Erzeugt ein Neues Objekt vom Typ *Point* mit den Koordinaten $x + p_2.x$, $y + p_2.y$.

*

```
Point operator* (double faktor);
```

Erzeugt ein neues Objekt vom Typ *Point* mit den Koordinaten $x \cdot faktor$, $y \cdot faktor$.

/

```
Point operator/ (double divisor);
```

Erzeugt ein neues Objekt vom Typ *Point* mit den Koordinaten *x* / *divisor*, *y* / *divisor*.

C.4 Klasse Line

Die Klasse *Line* stellt verschiedene Operationen für Strecken im zweidimensionalen Raum zur Verfügung.

Variablen der Klasse Line

a `Point a;`

b `Point b;`

Die beiden Punkte, die die Strecke definieren.

Konstruktoren der Klasse Line

```
Line (double x0, double y0, double x1, double y1);
```

Initialisiert die Variable *a* mit *Point* (x_0, y_0) und *b* mit *Point* (x_1, y_1).

```
Line (Point ap0, Point ap1);
```

Initialisiert *a* mit ap_0 und *b* mit ap_1.

Methoden der Klasse Line

intersection

```
bool intersection (Line& l2);
```

Liefert *true*, falls die Strecke sich mit der übergebenen Strecke l_2 schneidet, sonst *false*.

intersection

```
bool intersection (Line& l2,
                   Point& result);
```

Liefert *true*, falls die Strecke sich mit der übergebenen Strecke l_2 schneidet und liefert in *result* den Schnittpunkt zurück. Andernfalls ist der Rückgabewert *false* und der Referenzparameter bleibt unverändert.

len

```
double len ();
```

Ermittelt die Länge der Strecke.

sameSide

```
bool sameSide (Point ap0, Point ap1);
```

Liefert *true*, falls die Punkte ap_0 und ap_1 auf der gleichen Seite der Strecke liegen, ansonsten *false*.

side

```
int side (Point p);
```

liefert zurück, auf welcher Seite der gerichteten Strecke von *a* nach *b* der übergebene Punkt liegt:

0: Der Punkt liegt auf der Geraden durch *a* und *b*

1: Der Punkt liegt rechts von der gerichteten Strecke von *a* nach *b*

–1: Der Punkt liegt links von der gerichteten Strecke von *a* nach *b*

+

```
Line operator+ (Point c);
```

Liefert eine um den durch den Parameter *c* definierten Vektor verschobene Strecke zurück.

C.5 Template-Klasse Matrix

Variablen der Klasse Matrix<T>

dimension

```
int dimension;
```

Dimension der Matrix.

size

```
int* size;
```

Speichert die Anzahlen der Elemente für jede Dimension.

data

```
T* data;
```

Speichert die Elemente der Matrix

Konstruktoren der Klasse Matrix<T>

```
Matrix (int aDimension, ...);
```

Erzeugt eine Matrix der Dimension *aDimension*. Die Größen der einzelnen Dimensionen müssen zusätzlich übergeben werden.

Methoden der Klasse Matrix<T>

at

```
T& at (int i0, ...);
```

Liefert eine Elementreferenz für das i_0-te Element einer eindimensionalen Matrix, bei *n*-dimensionalen Matrizen müssen *n* Indizes angegeben werden.

Startindex ist 0.

clear

```
void clear ();
```

Initialisiert die Matrix mit 0. Für die Elemente der Matrix muß der entsprechend typisierte Zuweisungsoperator überladen sein.

fill

```
void fill (T t);
```

Füllt alle Elemente der Matrix mit *t*.

getMax

```
T getMax ();
```

Liefert das Maximum aller Elemente der Matrix zurück.

getSize

```
int getSize ();
```

Liefert die Gesamtanzahl aller Elemente der Matrix zurück.

getImage

```
Matrix<unsigned char> getImage ();
```

Falls die Matrix zweidimensional ist, wird eine neue Matrix der gleichen Dimensionalität, allerdings mit Elementtyp *unsigned char*, zurückgeliefert. Die Elemente der neuen Matrix werden dabei auf den Bereich von 0 bis 255 normiert.

Literatur

[Alb+97] Alberts R. et al. (Projektgruppe): *Entwicklung der Netzwerk- und Steuersoftware eines autonomen mobilen Roboters.* Technischer Report, Fachbereich Informatik, Universität Dortmund, 1997.

[AB74] Athen H., Bruhn J. (Hrsg.): *Rechnen und Mathematik.* München, Gütersloh, Wien: Verlagsgruppe Bertelsmann GmbH / Bertelsmann Ratgeberverlag, 1974.

[Bal97] Ballard D. H.: *An Introduction to Natural Computation*. MIT Press, 1997

[Bar98] Barkanowitz M.: *Sensordatenverarbeitung für die Navigation eines autonomen Roboters.* Unveröffentlichte Diplomarbeit, Fachbereich Informatik, Universität Dortmund, 1998 (in Druck).

[BBC+95] Buhmann J., Burgard W., Cremers A.B., Fox D., Hofmann T., Schneider F., Strikos J., Thrun S.: The mobile robot Rhino, *AI Magazine 16*, 1, 1995.

[BBD+97] Bergener T., Bruckhoff C., Dahm P., Janßen H., Joublin F., Menzner R.: Arnold: An antropomorphic autonomous robot for human environments. *Selbstorganisation von adaptivem Verhalten (SOAVE'97)*, 1997.

[Bla96] *Webster's New Encyclopedic Dictionary*. New York: Black Dog & Leventhal Publisher's Inc., 1996.

[Bor91] Born, G.: *Referenzhandbuch Dateiformate.* Bonn, München (u.a.): Addison-Wesley, 1991.

[Bra84] Braitenberg, V.: *Vehicles: Experiments in Synthetic Psychology.* The MIT Press, 1984.

[Bro11] *Brockhaus' Kleines Konversations-Lexikon.* Leipzig: F. A. Brockhaus, 1911.

[Bro83] *Brockhaus Kompaktwissen von A bis Z.* Wiesbaden: F. A. Brockhaus, 1983.

[BS96] Birbaumer N., Schmidt R. F. (Hrsg.): Das visuelle System. In *Biologische Psychologie (3. Aufl.)*. Berlin, Heidelberg, New York: Springer-Verlag, 1996, S. 372-410.

[EW95] Edlinger T., Weiß G.: Exploration, Navigation and Self-Localization in an Autonomous Mobile Robot. *Autonome mobile Systeme '95*, Karlsruhe, 1995.

[Ein97] Einsele T.: Real-time self-localization in unknown indoor environments using a panorama laser range finder. *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS'97)*, Grenoble, France, 1997, S. 697-703.

[Eng94] Engelson S.: *Passive map learning and visual place recognition*. Ph.D. thesis, Dept. Computer Science, Yale Uiversity, New Haven, CT, 1994.

[Fel92] Fellner, W.-D.: *Computergrafik*. Mannheim, Leipzig, Wien, Zürich: BI-Wissenschaftsverlag, 1992.

[GG95] Grüsser O.-I., Grüsser-Cornehls U.: Gesichtssinn und Okulomotorik. In R.F. Schmidt, G. Thews (Hrsg.): *Physiologie des Menschen*. Berlin, Heidelberg, New-York: Springer-Verlag, 1995.

[Gus96] Guski, R.: *Wahrnehmen: ein Lehrbuch*. Stuttgart, Berlin, Köln: Kohlhammer, 1996.

[Güt92] Güting, R. H.: *Datenstrukturen und Algorithmen*. Stuttgart: Teubner, 1992.

[Ham97] Hamilton P.: Neues Sehen. *c't 10*, 1997, S. 138.

[HS96] Hanebeck U. D., Schmidt G.: Set Theortic Localization of Fast Moile Robots using an Angle Measurement Technique. *Proceedings of the 1996 IEEE International Conference on Robotics and Automation*, Minneapolis, MN, April 1996, S. 1387-1394.

[HW79] Hubel D. H., Wiesel T.N.: Brain Mechanisms of Vision. *Scientific American 241*, 1979, S. 130-144.

[ISO85] ISO: *Information Processing Systems – Computer Graphics – Graphical Kernel System (GKS) – Functional Description*, IS 7942, 1985.

[ISO87] ISO: *Computer Graphics Metafile for the Storage and Transfer of Picture Description Information*. ISO 8632, Parts 1-4, 1987.

[Jäh97] Jähne B.: *Digitale Bildverarbeitung*. Berlin: Springer-Verlag, 1997.

[ME85] Moravec H. P., Elfes A.: High resolution maps from wide angle sonar. *Proceedings of the IEEE International Conference on Robotics and Automation*, St.Lois, MO, 1985, S. 116-121.

[Mor88] Moravec H. P.: Sensor Fusion in certainlty grids for mobile robots. *AI Magazine*, Summer 1988, S. 61-74.

[KK92] Kosaka A., Kak A. C.: Fast vision-guided mobile robot navigation using model based reasoning and prediction of uncertainties. *Image understanding 56*, 3, 1992, S. 271-329.

[SC94] Schiele B., Crowley J. L.: A Comparison of Position Estimation Techniques Using Occupancy Grids. *Roboters and Autonomous Systems 12*, 1994, S. 153-171.

[Ste90] Stevens W. R.: *Unix Network Programming*. Prentice Hall, 1990

[Ste93] Steinbrecher R.: *Bildverarbeitung in der Praxis*. München, Wien: Oldenbourg Verlag, 1993.

[Tep97] Tepas D.: *Abstraktion eines Umweltmodells für die Pfadplanung autonomer Roboter*. Unveröffentlichte Diplomarbeit, Fachbereich Elektrotechnik, Ruhr-Universität Bochum, 1997.

[VVX96] Vandorpe J., Van Brussel H., Xu H.: Exact Dynamic Map Building for a Mobile Robot using Geometrical Primitives Produced by a 2D Range Finder. *Proceedings of the 1996 IEE international Conference on Robotics and Automation*, Minneapolis, MN, April 1996.

[XVD+95] Xu H., Van Brussel H., De Schutter J., Vandorpe J.: Sensor Fusion and Positioning of the Mobile Robot LiAS. *Proceedings of the International Conference on Intelligent Autonomous Systems 4*, 1995, S. 246-253.

[Yam96] Yamauchi B.: Mobile Robot Localization in Dynamic Environments Using Dead Reckoning and Evidence Grids. *Proceedings of the 1996 IEEE International Conference of Robots and Automation*, Minneapolis, MN, 1996, S. 1401-1405.

[Zim95] Zimmer U. R.: Minimal Qualitative Topologic World Models for Mobile Robots. *Artifical Neural Networks and Expert Systems ´95*, Dunedin, New Zealand, 1995.

[ZP94] Zimmer U. R., von Puttkamer E.: Realtime-learning on an Autonomous Mobile Robot with Neural Networks. *Conference of Euromicro'94*, Realtime-Workshop, Vaesteraas, Sweden, June 1994.