”Demoapplikation Serviceroboter fürs Kinderzimmer – Integration von Bildverarbeitung, Handlungsplanung und Navigation“

Diplomarbeit an der Fachhochschule Brandenburg
Fachbereich Informatik und Medien

vorgelegt von Oliver Sachse

1. Gutachter: Prof. Dr. J. Heinsohn
2. Gutachter: Dipl.-Inform. I. Boersch

Brandenburg, den 15. Januar 2001

Aufgabenstellung

***"Demoapplikation Serviceroboter fürs Kinderzimmer –
Integration von Bildverarbeitung, Handlungsplanung und Navigation"***

Zielstellung des Themas ist die Entwicklung einer vollständigen Demoapplikation mit einem realen Roboter für folgendes Szenario: Auf einem Teppichuntergrund liegt verstreut farbiges Spielzeug, das durch einen Roboter eingesammelt und in einer Sammelstelle abgelegt werden soll. Die Objekte sollen mittels optischer Objekterkennung lokalisiert und gezielt aufgenommen werden. Für die Lösung der Aufgabe ist eine intelligente Verknüpfung von Bildverarbeitung, Handlungsplanung und Navigationsleistungen notwendig. Besonderes Augenmerk ist auf die Robustheit und die „Gutmütigkeit" des Systems zu legen.

Inhaltsverzeichnis

I Einführung 9

1 Serviceroboter 9

2 Die Aufgabe 11

II Laborumgebung 13

3 Der Pioneer 2 CE 13
3.1 Odometrie 14
3.2 Sonar 15
3.3 Greifer 16
3.4 Kamera 17

4 Saphira 18
4.1 Überblick 18
4.2 Architektur 19
4.3 Behaviors 21
4.4 Activities 21
4.5 Der Simulator 21

5 Bildverarbeitungssoftware 22
5.1 Der Bildverarbeitungsserver 22
5.2 Intel® Image Processing Library 23
5.3 Das Bildverarbeitungsprogramm „AdOculos“ 24

6 MS Visual C++ 5.0 25

III Theoretische Grundlagen 26

7 Einleitung 26

8 Einführung in die digitale Bildverarbeitung **29**
8.1 Was ist digitale Bildverarbeitung? . . . 29
8.2 Digitalisierte Grauwertbilder . . . 30
8.3 Mehrkanalige Bilder . . . 31
8.4 Einfache Grauwerttransformationen . . . 32
8.5 Operationen im Ortsbereich . . . 33
8.6 Kantendetektion . . . 35
8.6.1 Der Begriff der *Kante* . . . 35
8.6.2 Kantenmodelle . . . 36
8.6.3 Anforderungen und Probleme der Kantendetektion . . . 37
8.6.4 Einfache Verfahren zur Kantenextraktion . . . 38
8.6.5 Kantenbeschreibung mit Gradienten . . . 38
8.7 Morphologische Operationen . . . 39
8.8 Segmentierung . . . 40
8.9 Weiterverarbeitung segmentierter Bilder . . . 43
8.10 Die Hough-Transformation als Verfahren zur Linienerkennung . . . 45
8.10.1 Tracking . . . 48

9 Robotersteuerung **50**
9.1 Einleitung . . . 50
9.2 Steuerungsarchitekturen . . . 50
9.2.1 Überblick . . . 50
9.2.2 Der klassische Ansatz . . . 51
9.2.3 Der reaktive Ansatz . . . 57
9.2.4 Hybride Architekturen . . . 60
9.3 Navigation mobiler Roboter . . . 60
9.3.1 Der Begriff Navigation . . . 60
9.3.2 Probleme bei der Navigation . . . 61

IV Konzeption **64**

10 Einleitung **64**

11 Konzeption der Bildverarbeitungskomponente 64
11.1 Bildverarbeitung für mobile Robotersysteme 64
11.2 Einleitung 65
11.3 Teppicherkennung 66
11.3.1 Vorraussetzungen 66
11.3.2 Einleitung 67
11.3.3 Extraktion der Geradenpixel 68
11.3.4 Überführung in Geradengleichungen 70
11.3.5 Implementierungsmöglichkeiten 71
11.4 Spielzeugerkennung 73
11.4.1 Vorraussetzungen 73
11.4.2 Ansätze und Implementierungsmöglichkeiten 74
11.4.3 Gültigkeit von erkannten Objekten 76
11.5 Erkennung der Ablage 76
11.6 Zusammenfassung 79

12 Konzeption der Robotersteuerungskomponente 81
12.1 Einleitung 81
12.2 Rahmenbedingungen 81
12.3 Steuerungsarchitektur 84
12.4 Sensoren 85
12.5 Das Suchverhalten 86
12.6 Die Phasen der Steuerung 89
12.6.1 Objekt suchen 90
12.6.2 Objekt aufnehmen 91
12.6.3 Objekt ablegen 92
12.7 Bildverarbeitungsdaten 94

V Implementierung 95

13 Implementierung der Bildverarbeitung 95
13.1 Einleitung 95
13.2 Der Bildverarbeitungsserver 95

13.2.1 Übersicht . . . 95
13.2.2 Architektur . . . 96
13.2.3 Oberflächen-Einstellungen . . . 98
13.3 Die BV-Bibliothek für Saphira . . . 99
13.4 Änderungen am Bildverarbeitungsserver . . . 100
13.5 Implementierung der notwendigen Funktionen . . . 102
13.6 Teppicherkennung . . . 102
13.6.1 Filterung nach der Farbe ROT . . . 103
13.6.2 Einfacher Kantenfilter . . . 104
13.6.3 Hough-Transformation . . . 104
13.6.4 Hough-Akkumulator-Analyse . . . 105
13.7 Spielzeugerkennung . . . 106
13.7.1 Farbkantenfilter . . . 106
13.7.2 Konturen verbinden . . . 107
13.7.3 Objekte extrahieren . . . 107
13.7.4 Gültigkeit der Objekte feststellen . . . 108
13.8 Erkennung der Ablage . . . 108
13.9 Zeichenroutinen . . . 110
13.10 Der gemeinsame Speicher . . . 110
13.11 Die Saphira-Bibliothek . . . 111
13.12 Erweiterungen . . . 111
13.12.1 Verwendung der IPL . . . 111
13.12.2 Linien löschen . . . 112
13.13 Ein Beispiel . . . 113

14 Implementierung der Robotersteuerungskomponente **117**
14.1 Einleitung . . . 117
14.2 Objekt suchen . . . 117
14.3 Objekt aufnehmen . . . 122
14.4 Objekt ablegen . . . 125
14.4.1 Odometriegesteuerte Navigation zur Ablage . . . 126
14.4.2 Bildgesteuerte Navigation zur Ablage . . . 128
14.4.3 Das eigentliche „Objekt ablegen“ . . . 130

14.5 Das Haupt-Activity 131
14.6 Probleme bei der Implementierung 132

VI Testszenarien **135**

15 Ergebnisse **135**

16 Testreihen **137**

17 Ein Beispiel-Szenario **140**

VII Zusammenfassung und Ausblick **145**

18 Zusammenfassung **145**

19 Rahmenbedingungen und Einschränkungen **147**

20 Ausblick **149**

Abbildungsverzeichnis

1 Der Pioneer 2 CE - „Alfa“ 13
2 Das Sonar 15
3 Der Greifer 16
4 Die Saphira-Oberfläche 19
5 Nutzung der Bildverarbeitung mit Saphira 23
6 IPL-Beispiel-Anwendung 23
7 „AdOculos“ im Einsatz 25
8 Datenfluß 28
9 Kantenmodelle. Links oben: ideale Stufenkante; Rechts oben: ideale Rampenkante; Links unten: ideale Dachkante; Links unten: ideale Treppenkante 36
10 Eine reale Kante ist meist eine Kombination idealer Kantenprofile, überlagert mit Rauschen 37
11 Bild nach einer Segmentierung 44
12 Spektrum der Steuerungsarchitekturen 51
13 Die Klötzchenwelt 54
14 Ein Beispiel zur Subsumptions-Architektur 59
15 Der Einsatzort 67
16 Das Sichtfeld des Roboters 67
17 Das Bild von Abbildung 16 auf Seite 67 nach Ausführung einer Differenzoperation der Farbkanäle $ROT - GR\ddot{U}N$ und anschließender Binarisierung 68
18 Das Bild von Abbildung 17 auf Seite 68 nach einer Kantenextraktion mit dem Laplace-Operator und anschließender Binarisierung 70
19 Die markierte Flasche 77
20 Schematische Darstellung des Einsatzortes 83
21 Die Suchgebiete 87
22 Der Bildverarbeitungsserver 96
23 Das Ursprungsbild 113
24 Linienerkennung (links: Ausgabebild Rot-Filter; rechts: Ausgabebild Kantenfilter) 114

25 Farbkantenfilter auf das Ursprungsbild angewendet 115
26 Objekterkennung (links: nach der closing-Operation; rechts: nach dem Linien löschen 115
27 Das Ergebnisbild 116
28 Die Suchgebiete 118
29 Die Ausgangsposition 137
30 Die Ausgangsposition 140
31 links: Zufahrt auf das erste Objekt; rechts: Ausrichten an der Startlinie 141
32 links: Ausrichten an der Linie vor der Kiste; rechts: Ablegen des Objekts 141
33 links: Die Klebstoffflasche wurde erkannt; rechts: Start der Suche in Sektor 3 142
34 links: Ankunft vor der Kiste nach einer odometriebasierten Fahrt; rechts: Fahrt zur Kiste über das bildverarbeitungsbasierte Verfahren . 143
35 links: Die Suche ist abgeschlossen, Fahrt zurück zur Kiste ; rechts: Fertig! 144

Teil I

Einführung

1 Serviceroboter

Thema dieser Arbeit ist die Erstellung einer Demoapplikation, die eine Einsatzmöglichkeit von Servicerobotern im Haushalt zeigen soll. Bevor etwas näher auf die Aufgabenstellung eingegangen werden soll, wird zunächst ein Überblick über das Gebiet der Roboter und speziell der Serviceroboter gegeben.

Wenn man den Begriff *Roboter* hört, dann denkt man oft an Konstruktionen, die dem Menschen nachgebildet sind bzw. genau so aussehen wie er, und die ihm von der Intelligenz her sogar noch überlegen sind. Neben den normalen Sinnen des Menschen haben diese Maschinenmenschen noch weitere Spezialfähigkeiten, die natürlich jeweils bis zur Perfektion beherrscht werden.
Dieses Bild wurde maßgeblich von den verschiedenen Darstellungen in Science Fiction Filmen und Büchern beeinflußt.
In der Realität ist man jedoch noch weit entfernt von solchen „denkenden“ Maschinen. Es wird aber versucht, Technologien zu entwickeln, mit denen irgendwann ein solches System realisiert werden kann. Die meisten Roboter haben zur Zeit auch keinen menschenähnlichen Aufbau, sind aber optimal für ihre jeweiligen Aufgaben konstruiert worden. Hierbei lässt sich schon erkennen, dass heutige Roboter meist nur für bestimmte vorher festgelegte Einsatzgebiete entwickelt werden. Ein Universal-Roboter, der für viele verschiedene Aufgaben geeignet ist, existiert dagegen noch nicht.

Auf dem Gebiet der Robotik unterscheidet man die folgenden drei wesentlichen Entwicklungsstufen:

- Industrieroboter,
- Serviceroboter und den
- „Personal Robot“

Diese unterscheiden sich im Grad der Autonomie der Aufgabenausführung und in Bezug auf die Anforderungen an die Umwelt.
Industrieroboter wurden entwickelt, um sich ständig wiederholende Arbeitsabläufe mit einer hohen Präzision ausführen zu können. Beispiele hierfür sind z.B. in der Automobilindustrie zu finden. Diese Roboter ersetzen bei einer solchen Arbeit den Menschen und arbeiten effizienter als er, da sie z.B. keine Ermüdungserscheinungen aufweisen und somit keine Pausen und keinen Schlaf benötigen. Industrieroboter sind aber sehr stark an ihren festen Standort, ihre vorgegebene Umwelt und ihre definierte Aufgabenstellung gebunden. Ändert sich ein Aspekt, ist auch ihre Funktion beeinträchtigt.
Eine zweite Stufe ist das Gebiet der Serviceroboter, der auch Gegenstand dieser Arbeit sein soll. Serviceroboter stellen eine Weiterentwicklung heutiger Industrieroboter und autonomer Transportsysteme dar und bilden die Grundlage für eine künftige Generation der sogenannten „Personal Robots“. Letztere sind in Bezug auf Umwelt und Aufgabenstellungen am flexibelsten und sollen später sogar zum normalen Leben gehören, wie heutzutage der PC.

Grundsätzlich werden als Serviceroboter solche Roboter bezeichnet, die den Menschen bei einer Vielzahl von Tätigkeiten in seiner natürlichen Umgebung unterstützen und vielfältige Dienstleistungen erbringen [Bis98]. Sie werden somit nicht (wie Industrieroboter) direkt bei der industriellen Erzeugung von Sachgütern eingesetzt.
Da sie den Menschen unterstützen sollen, existiert auch keine vollständige Trennung der Arbeitsbereiche von Mensch und Maschine mehr. Vielmehr ist bei solchen Systemen sogar eine enge Interaktion zwischen Benutzer und Roboter gefordert. Hierbei muss dann großen Wert auf ein funktionierendes Sicherheitskonzept gelegt werden, um eine Gefährdung oder gar Schädigung des Menschen zu verhindern.
Serviceroboter stellen in der Regel autonome mobile Systeme dar. Dies bedeutet, dass sie sich in ihrer Umwelt unabhängig von einem menschlichen Überwacher bewegen und ihre Aufgaben erledigen können.
Serviceroboter sind schon für viele Anwendungsgebiete entwickelt worden. Der ganz große Durchbruch ist ihnen bis jetzt allerdings noch nicht gelungen. Dies liegt zum einen darin begründet, dass alle bisherigen Entwicklungen nur teure Speziallösungen

sind, die sich nicht selbständig an verschiedene Aufgaben und Einsatzumgebungen anpassen können. Hierfür sind zunächst noch hochqualifizierte Fachkräfte notwendig, die die Einrichtung und explizite Programmierung übernehmen.
Bis man also eine Roboter-Haushaltshilfe für wenig Geld im Laden kaufen kann, die man zuhause nur noch einschalten braucht und die dann gleich richtig funktioniert, wird somit noch einige Zeit vergehen.
Trotzalledem werden Serviceroboter schon in vielen Bereichen, wie z.B. Baugewerbe, Gastronomie, Medizin oder Weltraumtechnik, erfolgreich genutzt. Als Beispiele seien erwähnt: automatische Rasenmäher, autonome Minensuch- und entschärfroboter, mobile Reinigungsroboter oder autonome Systeme zur Kanalinspektion. Eine ausführliche Liste von Servicerobotern im Einsatz ist in der Serviceroboter-Datenbank am Fraunhofer Institut [SRD98] zu finden.

2 Die Aufgabe

In der vorliegenden Arbeit soll ein Anwendungsbereich für Serviceroboter im Haushalt dargestellt werden. Als Szenario wurde hierfür das Aufräumen des Kinderzimmers gewählt. Dies ist ein durchaus realistisches Anwendungsgebiet, da die Aufräumarbeit in der Regel bei allen Beteiligten (Kind und Eltern) gleichermaßen unbeliebt ist.
Wie in der Aufgabenstellung bereits beschrieben wurde, ist das Szenario hierzu Folgendes:
Auf einem Teppichuntergrund liegt verstreut farbiges Spielzeug, das durch einen Roboter eingesammelt und in einer Sammelstelle abgelegt werden soll. Die Objekte sollen mittels optischer Objekterkennung lokalisiert und gezielt aufgenommen werden.
Als Entwicklungsplattform für die vorliegende Arbeit kommt der Pioneer 2-Roboter der Firma ActivMedia zum Einsatz, dessen Aufbau im nächsten Teil beschrieben wird.
Ziel der Arbeit ist eine komplette Demoapplikation, die den Einsatz von Servicerobotern demonstriert, und die auch an verschiedenen Orten einsetzbar sein soll. Es sind hierbei jedoch bestimmte Rahmenbedingungen zu berücksichtigen, die im weiteren Verlauf der Arbeit erläutert werden.

Auf Grund der technischen Möglichkeiten mit dem zur Verfügung stehenden Robotersystem wird in der Arbeit nur mit einer verkleinerten und vereinfachten „Version“ eines Kinderzimmers gearbeitet. Dieses besteht aus einem definierten Einsatzgebiet von ca 3-4 m^2 und darin herumliegenden kleinen LEGO-DUPLO-Bausteinen. Da hierbei auch nicht mit „echten“ Wänden gearbeitet wird, soll für die Erkennnung der Begrenzung des Gebietes wie auch zur Wahrnehmung des Ablageplatzes eine optische Lösung verwendet werden. Im realen Szenario „Kinderzimmer“ könnte anstatt einer optischen Erkennung der Begrenzung auch mit einer Wanderkennung auf Basis von Ultraschallsensoren gearbeitet werden, da davon ausgegangen werden kann, dass der Roboter den gesamten Bereich des Kinderzimmers, der ja durch Wände begrenzt ist, aufräumen soll. Für diese Aufgabe, die eine portable Lösung zum Ziel hat, ist der optische Weg jedoch der Bessere.

In den folgenden Teilen soll als Erstes die zur Verfügung stehende Hard- und Software vorgestellt, als Zweites die theoretischen Grundlagen der berücksichtigten Themen und Techniken erläutert und anschließend meine Überlegungen und die letztendliche Realisierung vorgestellt werden.

Den Anfang macht also die Laborumgebung.

Teil II

Laborumgebung

3 Der Pioneer 2 CE

Die Entwicklungsplattform für diese Arbeit stellt der Pioneer 2 CE - Roboter der Firma ActivMedia sowie die dafür verfügbare Steuerungssoftware dar. Im Folgenden soll daher dieses System etwas näher beschrieben werden.
Zunächst werde ich dazu auf die Hardware-Komponenten des Roboters eingehen und im zweiten Teil dann die eingesetzte Steuerungssoftware vorstellen.

Der Pioneer 2 CE gehört zur Familie der Pioneer 2 - Roboter und wird von der Firma ActivMedia entwickelt und vertrieben. Im Vergleich zu anderen Systemen stellt er einen relativ preiswerten Einstieg in die Roboterwelt dar.
Das Labor für künstliche Intelligenz der Fachhochschule Brandenburg ist seit Sommer 1999 mit zwei dieser Roboter ausgestattet, welchen die Namen „Alfa“ und „Romeo“ gegeben wurden. Für diese Arbeit kam der Roboter „Alfa“ zum Einsatz, da er eine etwas andere Kamera-Konstruktion besitzt. Abbildung 1 zeigt eine Aufnahme von „Alfa“.

Abbildung 1: Der Pioneer 2 CE - „Alfa“

Der Pioneer 2 CE besitzt zwei vordere Antriebsräder und ein hinteres Stützrad. Durch diese Konstruktion ist es ihm möglich, auf der Stelle zu drehen. Des Weiteren ist er mit einem an der Vorderseite gelegenen Sonarring ausgestattet.
Zusätzlich besitzen die an der Fachhochschule Brandenburg eingesetzten Roboter noch folgende Komponenten:

- einen Greifer,
- einen Kompaß sowie
- eine auf einer motorgesteuerten Schwenk-Neige-Einheit befindliche Farb-Kamera.

Herz des Roboters ist ein Board mit einem mit 20 MHz getakteten Siemens 88C166-basierten Mikrocontroller.
Gesteuert wird der Roboter üblicherweise über einen externen Rechner, auf dem dann die Steuerungssoftware läuft. Mit diesem Rechner ist er über ein serielles Kabel oder Funkmodem verbunden. Es entsteht dann eine Client/Server-Architektur, wobei der Client durch die externe Steuerungssoftware und der Server durch das auf dem Roboter laufende Betriebssystem (P2OS) dargestellt wird. Für die Roboter „Alfa“ und „Romeo“ kommen als Steuerungsrechner Notebooks der Firma Compaq zum Einsatz.
In den nächsten Abschnitten werden die wichtigsten Komponenten noch einmal näher beschrieben.

3.1 Odometrie

Zur Positionsbestimmung und -aktualisierung besitzt der Pioneer 2 CE Odometriesensoren auf Basis von Shaftencodern. Diese sind an den Antriebsrädern angebracht. Gemessen werden mit ihnen die von den Rädern zurückgelegten Entfernungen. Werden die Sensordaten beider Räder kombiniert, so können auch Richtungsinformationen erhalten werden.
Leider liefern diese Sensoren nur auf kurze Entfernungen einigermaßen verlässliche Werte. Da sie die von den Rädern zurückgelegten Entfernungen messen, werden sie sehr stark von den bei den Bewegungen entstehenden Ungenauigkeiten, z.B. durch

Radschlupf, beeinflußt. Besonders störend wirken sich hierbei schnelle, ruckartige Bewegungen aus. Je weiter sich der Roboter bewegt, um so mehr summieren sich dann auch die Fehler auf und die Positionseinschätzungen werden somit immer ungenauer.

3.2 Sonar

Für die Objekterkennung und Abstandsbestimmung (z.B. zu Wänden) ist der Pioneer 2 CE mit einem aus 8 Ultraschallsensoren bestehenden Ring ausgestattet. Dieser Sonarring ist an der Vorderseite in einer Höhe von 18.5 cm angebracht. Hierbei sind 6 der Sensoren in 20 Grad-Intervallen nach vorne und zwei weitere nach den Seiten gerichtet. Auf Grund dieser Anordnung kann der Roboter dann Hindernisse in einem 180 Grad-Bereich erfassen. In Abbildung 2 sind die sich dabei ergebenden Positionen der einzelnen Sensoren dargestellt.

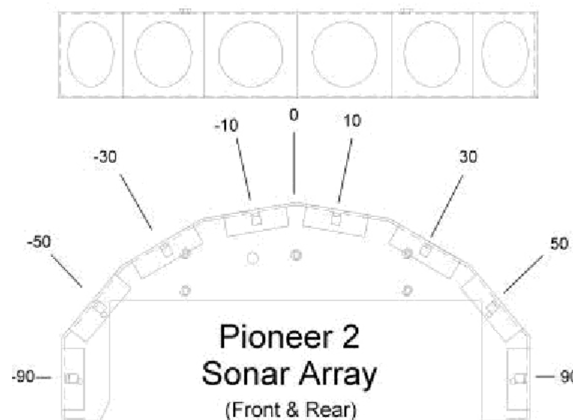

Abbildung 2: Das Sonar

Die einzelnen Sonarsensoren feuern nacheinander (in einer vorher definierten Reihenfolge) Ultraschallsignale („Pings“) ab und warten dann auf eine „Rückmeldung“. Trifft das abgesandte Signal dabei auf ein Hindernis, dann wird es reflektiert und wieder zum Sensor zurückgeschickt. Dort wird es wieder registriert und auf Grund der verstrichenen Zeit kann dann auf die Entfernung zwischen Roboter und Hindernis geschlossen werden. Die Genauigkeit der Messungen hängt dabei zu einem großen Teil von der Oberfläche der Hindernisse ab.

Die Feuerrate, mit der vom Sonar Ultraschallimpulse ausgesandt werden, beträgt 25 Hz und der Abstand in dem Objekte registriert werden können, liegt zwischen 10 und 500 cm.

3.3 Greifer

Zusätzlich zu den Standardkomponenten sind die beiden an der Fachhochschule Brandenburg eingesetzten Roboter mit einem Greifer ausgestattet. Dieser ist an der Vorderseite unter dem Sonar angebracht (Abbildung 3) und ermöglicht das Aufnehmen von Objekten durch den Roboter.

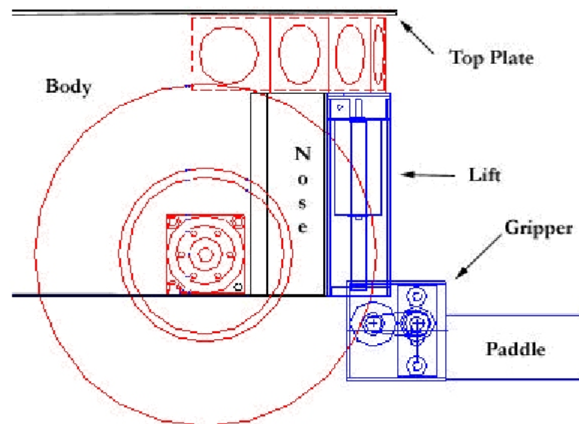

Abbildung 3: Der Greifer

Neben den motorbetriebenen Greiferfingern (Paddles) besteht er dabei noch aus einem Liftmechanismus. Hierdurch kann der Roboter Objekte mit einem Maximal-Gewicht von 2 kg um bis zu 9 cm anheben.
Besonders hervorzuheben sind auch die in den Greifer-Fingern eingebauten Lichtschranken, welche sich jeweils am vorderen und hinteren Ende befinden. Durch die Abfrage ihrer Werte kann festgestellt werden, ob sich ein Objekt im Greifer befindet. Leider sind diese Sensoren, zumindest bei den in der FH Brandenburg eingesetzten Pioneer-Robotern, bei komplett geöffneten Greifern sehr unzuverlässig. Abhilfe schafft nur ein teilweises Schließen der Greifer, bis zu einem Abstand von ca. 10-15 cm.

3.4 Kamera

Eine weitere und besonders für die vorliegende Arbeit wichtige Komponente ist die Kamera.
Zum Einsatz für den Pioneer 2 kommt eine mit einem Zoomobjektiv ausgestattete Farbkamera (EVI-D31) der Firma Sony mit einer maximalen Auflösung von 752 x 585 Bildpunkten. Diese Kamera ist auf einer motorgesteuerten Schwenk-Neige-Einheit (Pan-Tilt-Unit) angebracht. Der Steuerungssoftware ist es dann möglich, die Bewegungen der Kamera darüber zu regeln. Des Weiteren besitzt die Kamera ein eingebautes *Auto Tracking*, welches die Funktionalität bietet, automatisch ein durch seine Farben vorher definiertes Objekt zu verfolgen. Die Bewegungen werden hierbei von der Kamera-Hardware selbst gesteuert, d.h. unabhängig von der Roboter-Steuerungssoftware.
Die von der Kamera gelieferten Bilder können über eine Funkverbindung an den Steuerungsrechner übermittelt werden.
Als optionale Komponente ist für die Pioneer 2-Roboter ein Board der Firma Newton Labs erhältlich, das eine einfache Bildverarbeitungsfunktionalität zur Verfügung stellt. Dieses Board kann auf verschiedene Farben trainiert werden und entsprechend-farbige Objekte erkennen und verfolgen. Die Ergebnisse (Koordinaten des Objektes im Bild) stellt es dann der Steuerungssoftware zur Verfügung. Ein Anwendungsbeispiel hierzu zeigt die Arbeit von Matthias Jung an der FH Hamburg [Jun00].
Komplexere Bildverarbeitungsmethoden, wie sie in der vorliegenden Arbeit notwendig sind, können mit dieser Hardware allerdings nicht realisiert werden.
Die Roboter „Alfa“ und „Romeo“ sind nicht mit einem solchen Board ausgestattet. Anstatt der Hardware- kommt daher eine Software-Lösung zum Einsatz. Hierfür wurde an der Fachhochschule eine entsprechende Bildverarbeitungssoftware entwickelt, zu der ein kleiner Überblick im Anschluß an den folgenden Abschnitt gegeben wird.

4 Saphira

4.1 Überblick

Maßgeblich an der Entwicklung der Systemarchitektur der Pioneer-Roboter war und ist Kurt Konolige [Kon96] beteiligt. Neben der Roboterplattform hat er dabei auch ein Steuerungskonzept auf Basis einer Client/Server-Architektur entworfen. Ein Ergebnis dessen ist die Saphira-Entwicklungsumgebung, welche Funktionen zur Erstellung von Client-Anwendungen beinhaltet. Den Server stellt der Roboter selbst bzw. das dort laufende Betriebssystem dar. Er hat dann die Funktion einer low-level-Steuerung und kümmert sich z.B. um die direkte Motoransteuerung oder das Abfeuern und wieder Aufsammeln von Sonarimpulsen. Hierbei führt er die vom Client gewünschten Funktionen aus und stellt diesem wiederum Statusinformationen zur Verfügung. Der Client selbst braucht somit nicht die genauen Details des Roboterservers kennen, was bedeutet, dass dort Steuerungsroutinen auf einer höheren Ebene entwickelt werden können. Dadurch sind die Client-Programme theoretisch unabhängig von der eigentlichen Roboterhardware. Genutzt wird diese Unabhängigkeit beim im Saphira-Paket enthaltenen Pioneer-Simulator.
Saphira ist für verschiedene Betriebssysteme erhältlich und wird standardmäßig mit der in Abbildung 4 auf der nächsten Seite zu sehenden Oberfläche ausgeliefert. In der Oberfläche werden die Positionen des Roboters und der erkannten bzw. definierten Objekte (Artefakte) sowie die Sonarmessungen grafisch dargestellt. Zusätzlich enthält sie einen Kommandozeileninterpreter, über den direkt Steuerungbefehle an den Roboter gesendet werden können.

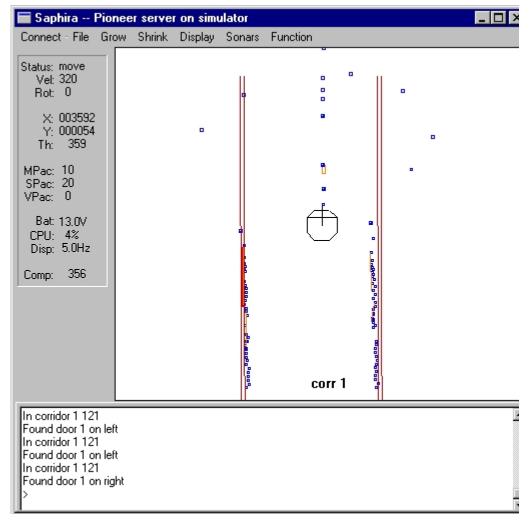

Abbildung 4: Die Saphira-Oberfläche

4.2 Architektur

Das Saphira-System kann man sich als aus zwei übereinander liegenden Architekturen zusammengesetzt vorstellen. Die untere Ebene ist dabei die *Saphira-System-Architektur* und die obere die *Roboter-Kontroll-Architektur* (näheres dazu im Saphira Manual [Act99c]).

In der Saphira-System-Architektur werden dabei die grundlegenden Funktionen zur Roboteransteuerung zur Verfügung gestellt. Hier befindet sich zunächst ein synchrones Betriebssystem, das die registrierten saphira-eigenen und benutzerdefinierten Routinen in einem 100 ms-Zyklus verwaltet. Die hiervon verwalteten Routinen haben dabei die Struktur von endlichen Automaten (*finite-state machines*). Neben diesen synchronisierten Routinen ist es auch möglich vom Saphira-Zyklus asynchron laufende Funktionen zu starten, um beispielsweise komplexere Planungsaufgaben erledigen zu können.
Des Weiteren wird von dieser Ebene die gesamte Paket-Kommunikation mit dem Roboter-Server gesteuert.
Als drittes befindet sich hier auch der sog. *State reflector*, der den momentanen Status des Roboters (z.B. Sensor-, Bewegungs- und Positionsinformationen) auf

dem Client zur Verfügung stellt und sich um das Aktualisieren der Werte kümmert. Hier können dann auch *motion setpoints* definiert werden, die zur Steuerung des Roboters notwendig sind und zu ihm übertragen werden. Es kann z.B. ein Wert für die Geschwindigkeit festgelegt werden, wobei von der Steuerung dann versucht wird, diesen Wert zu erreichen (beispielsweise durch Beschleunigen).

Über dem *State reflector* setzt nun die Roboter-Kontroll-Architektur auf. Dort befinden sich Routinen zur Verarbeitung von Sensordaten sowie Routinen zur Steuerung des Roboters. Erstere können noch unterteilt werden in Funktionen zur Interpretation von Sensordaten, Funktionen zur Aktualisierung der Roboterposition in der Karte und Funktionen zur grafischen Darstellung der Ergebnisse in Saphira. Saphira arbeitet hierbei mit zwei verschiedenen Koordinatensystemen, dem *Local Perceptual Space* und dem *Global Map Space*. Ersteres ist eine vom Roboter aus egozentrische (lokale) Karte mit geringen Ausmaßen, während Zweiteres eine globale Karte der Welt darstellt. Vom Roboter mit den Sensoren erfaßte und von den Interpretierungsroutinen klassifizierte Objekte können dann in diese Karte als Artefakte eingetragen werden. Für diese Artefakte gibt es bestimmte vordefinierte Typen, z.B. Punkte, Korridore, Türen. Des Weiteren ist es möglich, in Saphira sog. *Map*-Dateien zu laden, die bereits eine Menge von Artefakten enthalten und eine Karte einer bestimmten Umgebung, beispielsweise eines Raumes, darstellen. Der Roboter kann diese Karte dann als Navigationshilfe benutzen und versuchen die von ihm erkannten Objekte mit den in der Karte enthaltenen abzugleichen und seine Position im Raum festzustellen.
Neben den sensorabhängigen Routinen befinden sich in der Kontroll-Architektur noch die von Saphira benötigten Routinen, die zur Verwaltung von Steuerungsprogrammen notwendig sind.
Grundsätzlich gibt es in Saphira zwei Arten von Programmen, die erstellt werden können, *Behaviors* und *Activities*. Behaviors werden dabei von einem Fuzzy-Controller gesteuert, während Activities von der *Colbert Executive* verwaltet werden. Es können in Saphira beliebig viele Activities und Behaviors gestartet werden, die dann parallel abgearbeitet werden. Beide Arten unterliegen dann aber dem 100 ms Saphira-Zyklus, d.h. alle 100 ms wird das entsprechende Programm aufgerufen und kann seine Abarbeitung für eine bestimmte Zeit fortsetzen.

4.3 Behaviors

Behaviors sind einfache kleine Steuerungsprogramme, die mit Fuzzy-Regeln implementiert werden und jeweils nur eine bestimmte Aufgabe erfüllen. Einige Behaviors werden mit Saphira schon mitgeliefert, z.B. zur Kollisionsvermeidung. Sie können aber auch unter Zuhilfenahme des im Paket ebenfalls enthaltenen Behavior-Compilers (`bgram`) selbst erstellt werden.
Eine Besonderheit ist, dass Behaviors mit einer Priorität versehen werden können, welche bei der Auswertung der Fuzzy-Regeln vom Fuzzy-Controller berücksichtigt wird. Behaviors können allerdings keine anderen Programme aufrufen.

4.4 Activities

Als Zweites können in Saphira noch Activities definiert werden. Diese sind dann auch für komplexere Steuerungs- und Planungsaufgaben geeignet.
Erstellt werden Activities in einer in Saphira definierten Programmiersprache mit dem Namen *COLBERT* [Kon99]. COLBERT ist an die Sprache C angelehnt, bietet aber nur einen Teil der dort enthaltenen Sprachkonstrukte. Die grundlegende Struktur der Activities basiert auf endlichen Automaten (*finite state machines*). Im Gegensatz zu Behaviors können Activities andere Activities oder Behaviors aufrufen, besitzen allerdings keine Prioritäten.
Zusätzlich ist es noch möglich, von COLBERT aus externe Bibliotheksfunktionen aufzurufen, die für MS WINDOWS beispielsweise in *Dynamic Link Libraries* (DLLs) zur Verfügung gestellt werden. Dieser Vorgang ist geeignet, um aufwendige Berechnungen auszulagern (da Activities nicht zunächst kompiliert sondern von der Colbert Executive interpretiert werden) oder die in COLBERT enthaltenen Einschränkungen zu umgehen. Zum Beispiel sind die zur Ansteuerung von Greifer und Kamera zur Verfügung gestellten Funktionen in solchen Bibliotheken untergebracht.

4.5 Der Simulator

Wie schon erwähnt wurde, enthält das Saphira-Paket noch einen Simulator, der das Verhalten des Pioneer-Roboters nachbildet. Er besitzt dabei realistische Fehlermodelle für die Sonar- und Odometrie-Sensoren.

Auf Grund der Client/Server-Architektur ist es nun möglich, Programme zu schreiben und zunächst mit dem Simulator zu testen. Als Server fungiert dann nicht der echte Roboter, sondern der Simulator. Auf Grund der einheitlichen Kommunikations-Schnittstellen brauchen die Programme später für den richtigen Roboter aber nicht mehr verändert werden.

5 Bildverarbeitungssoftware

5.1 Der Bildverarbeitungsserver

Da in den in der Fachhochschule Brandenburg eingesetzten Robotern keine Bildverarbeitungshardware vorhanden ist, wurde eine entsprechende Software entwickelt. Der Ablauf dabei ist Folgender:
Die Bildsignale der Kamera werden zunächst über eine Funkverbindung an eine im Steuerungsrechner installierte Framegrabber-Karte übertragen, wo sie digitalisiert werden. Die Bildverarbeitungssoftware läuft auf diesem Rechner dann als eigenständiger Prozeß, also zunächst unabhängig von Saphira, und verarbeitet diese Bilder.
Die Ergebnisse müssen aber in irgendeiner Form noch dem Saphira-System zur Verfügung gestellt werden. Bei dem hier besprochenen Ansatz wurde der Weg über einen gemeinsamen Speicherbereich gewählt.
Dies bedeutet nun Folgendes:
Die Bildverarbeitungssoftware (Bildverarbeitungsserver) und eine Bibliothek (DLL) enthalten jeweils Funktionen zum Zugriff auf einen gemeinsamen Speicherbereich, wobei der Bildverarbeitungsserver seine Ergebnisse dort ablegt. Die Bibliothek kann nun in Saphira geladen werden und enthält wiederum Funktionen, die den gemeinsamen Speicherbereich von dort aus zugänglich machen. Somit können die Bildverarbeitungsergebnisse (z.B. Schwerpunkte von Objekten) dann für die Robotersteuerungsprogramme genutzt werden.
Abbildung 5 auf der nächsten Seite stellt diese Zusammenhänge noch einmal grafisch dar.
Weitere Details zum Bildverarbeitungsserver werden im Abschnitt „Implementierung der Bildverarbeitung“ dargestellt.

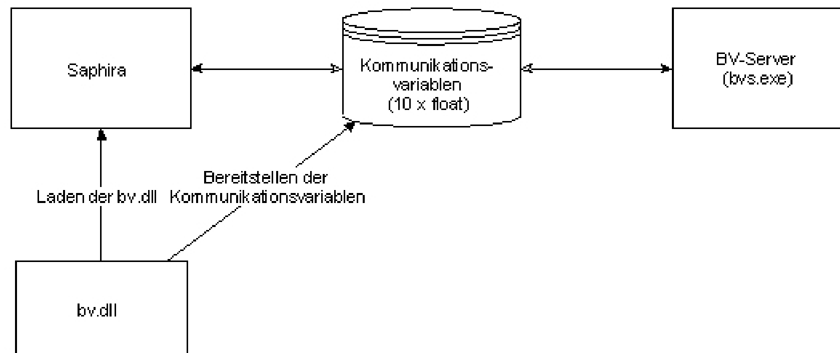

Abbildung 5: Nutzung der Bildverarbeitung mit Saphira

5.2 Intel® Image Processing Library

Die Image Processing Library v2.2 ist eine Bibliothek, die einfache Bildverarbeitungsfunktionen, wie z.B. Kantenfilter enthält. Sie wird von der Firma Intel® entwickelt und ist speziell für Intel®-Prozessoren und die MMX™-Technologie optimiert.
Neben C-Header-Dateien, die man für eigene Anwendungen nutzen kann, ist ihr auch eine Beispiel-Anwendung (`IPLib Image Editor`) beigefügt, mit der man schnell die vorhandenen Funktionen auf eigene Bilder anwenden kann. Ein Beispiel dazu ist in Abbildung 6 zu sehen.

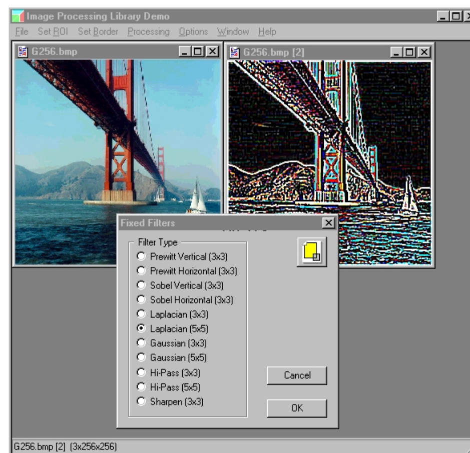

Abbildung 6: IPL-Beispiel-Anwendung

5.3 Das Bildverarbeitungsprogramm „AdOculos“

Wenn man für seine eigene Anwendung auf der Suche nach geeigneten Bildverarbeitungsmethoden ist, steht man zunächst vor einem Problem. Für jede Art von Bildverarbeitungsfunktionen gibt es meistens verschiedene Algorithmen, die je nach Anwendungsfall mehr oder weniger gut geeignet sind. Ein Beispiel dafür ist die Menge an vorhandenen Kantenfiltern (siehe dazu auch 8.6 auf Seite 35). Da gibt es zum Beispiel einen einfachen Differenzoperator, den LaPlace-Operator oder den Sobeloperator. Wenn man in seiner eigenen Anwendung nun aber einen Kantenoperator benötigt, stellt sich die Frage, welcher denn der geeignetste ist. Die Antwort darauf steht nicht unbedingt in irgendwelchen Bildverarbeitungsbüchern, sondern muss oft erst durch Tests ermittelt werden. Jetzt kann man also versuchen, sich geeignete Verfahren auszuwählen, diese in die eigene Anwendung zu implementieren, und das Ergebnis auszuwerten. Diese Vorgehensweise ist natürlich sehr aufwendig und auch nicht in jedem Fall erforderlich. Als Alternative können vorhandene Bildverarbeitungsprogramme genutzt werden, mit denen es möglich ist, schnell und einfach selbst komplexe Algorithmenketten zu entwerfen und zu testen. Als Basis dafür bieten diese Tools vorgefertigte Algorithmenbibliotheken, aus denen man sich geeignete Methoden auswählen, kombinieren und auf Testbilder anwenden kann. Beispiele für solche Programme sind „AdOculos“ von der DBS GmbH, das in einer Studentenedition dem Buch [Bäs98] beiliegt, oder „Image++“ der Image Integration GmbH, das als Sharewareversion unter [Ima01] zu haben ist. Für meine ersten Testversuche, habe ich das Programm „AdOculos“ benutzt, da es einerseits im KI-Labor der FH Brandenburg installiert ist, und andererseits das dazugehörige Buch [Bäs98], das die Möglichkeiten und Algorithmen des Programmes gut beschreibt, in der Hochschulbibliothek verfügbar ist. In Abbildung 7 auf der nächsten Seite ist das Programm beim Ausführen der Houghtransformation, einer Methode zur Geradenextraktion, zu beobachten. Im untersten Fenster ist hierbei die Algorithmenkette zu sehen, während die oberen Fenster die Bilder der einzelnen Bearbeitungsschritte zeigen. Wie zu erkennen ist, ist dieser Algorithmus zu umfangreich, als dass man ihn „nur mal so zum Testen“ implementieren kann.

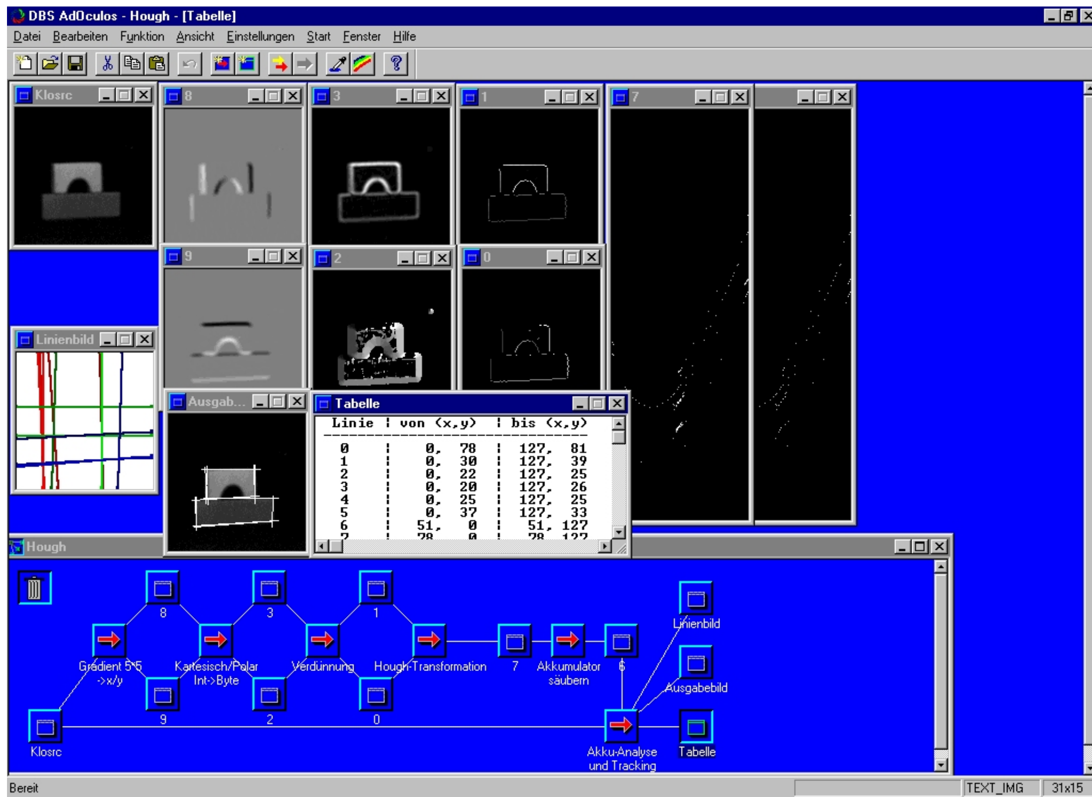

Abbildung 7: „AdOculos“ im Einsatz

6 MS Visual C++ 5.0

Visual C++ ist eine Entwicklungsumgebung von Microsoft, die zum Erstellen von MS Windows-Anwendungen geeignet ist. Dazu stellt sie z.B. die Microsoft Foundation Class Library (MFC) bereit, die die entsprechenden benötigten Elemente (Fenster, Schaltflächen...) in C++ Klassen verpackt.
Zum Einsatz für die Programmierung des Pioneer 2 kommt diese Umgebung bei der Erstellung von in Saphira ladbaren Bibliotheken, die z.B. rechenaufwendige Funktionen enthalten. Diese müssen dann allerdings in Standard C programmiert werden.
Des Weiteren wurde mit dieser Umgebung und unter Verwendung der MFC auch der Bildverarbeitungsserver erstellt.

Teil III

Theoretische Grundlagen

7 Einleitung

Wie schon aus der Aufgabenstellung hervorgeht, liegen die besonderen Schwerpunkte bei der Lösung der Aufgabe in den Bereichen Bildverarbeitung, Handlungsplanung und Navigation.
Der Roboter muss auf einem Teppich verstreutes Spielzeug erkennen, es aufnehmen, zu einer Ablage bringen und dort ablegen. Dabei sollte er sich selbst immer auf dem Teppich befinden und möglichst nicht über Spielzeug fahren. Des Weiteren soll er nach getaner Arbeit wieder auf seine Ausgangsposition zurückkehren. Daraus ergeben sich für den Roboter die folgenden Fragestellungen:

- Wie erkenne ich überhaupt den Teppich, das Spielzeug und die Ablage?
- Wann liegt ein Objekt auf dem Teppich?
- Wie finde ich ein Objekt? bzw. Welches ist das nächste Objekt, das ich aufnehmen soll?
- Wie nehme ich ein Objekt auf?
- Wie liefere ich es ab?
- Wie bleibe ich innerhalb der Teppichbegrenzungen?
- Wie kann ich garantieren, dass ich nicht über Spielzeug fahre?
- Wann ist meine Aufgabe beendet?

Die Grundprobleme dabei sind also (wie auch generell bei der Roboterprogrammierung) die Wahrnehmung der Umgebung und die entsprechende Reaktion darauf. Von den verschiedenen Sensoren werden Daten gesammelt, diese dann vom Robotersteuerungsprogramm verarbeitet und entsprechende Reaktionen festgelegt und ausgeführt.

Die zu erkennende Umgebung stellt in meinem Fall der Teppich, das sich darauf befindliche Spielzeug sowie die Ablage dar. Wie bereits erwähnt wurde, kommt hierfür die am Roboter angebrachte Kamera zum Einsatz, aus deren Bildern mit geeigneten Verarbeitungsmethoden die relevanten Informationen herausgefiltert werden müssen.
Es ist daher sinnvoll, die Arbeit in die zwei Schwerpunkte Bildverarbeitung und Robotersteuerung aufzuteilen. Für die oben angesprochenen Fragestellungen bedeutet dies, dass sich der Teil Bildverarbeitung mit den ersten beiden und die Robotersteuerung mit den weiteren Punkten befaßt. Weiterer Grund für diese Zweiteilung ist natürlich auch die, im letzten Abschnitt vorgestellte, Architektur des Systems. Um die Bildverarbeitung realisieren zu können, ist es zwangsläufig nötig, eine von Saphira eigenständige Anwendung zu schreiben. Als Grundlage hierfür habe ich mich für den bereits vorhandenen Bildverarbeitungsserver entschieden.
Die Steuerung des Pioneers wird mittels Activities und Behaviors unter Saphira realisiert. Zur Kommunikation und Synchronisation der beiden Teile wird, wie gesagt, ein gemeinsamer Speicherbereich genutzt. In Abbildung 8 auf der nächsten Seite wird der sich daraus ergebene Datenfluß noch einmal dargestellt, der wie folgt beschrieben werden kann:
Die auf dem Roboter angebrachte Kamera liefert die Bilder und überträgt sie via Funk an den entsprechenden Empfänger, der an die Framegrabberkarte des Steuerungs-Notebooks angeschlossen ist. Dort wird dann das Digitalisieren der Bilder veranlasst. Die erhaltenen Digitalbilder werden dann vom Bildverarbeitungsserver, der eine eigenständige Applikation ist, je nach Aufgabenstellung verarbeitet. Um die dabei extrahierten Daten dann den eigentlichen Robotersteuerungsprogrammen zugänglich zu machen, wurde eine Lösung mit einem gemeinsamen Speicherbereich implementiert. Über eine DLL wird dieser Speicher für Saphira verfügbar. Nachdem die Daten also in diesem Bereich abgelegt wurden, können sie von den Steuerungsroutinen (Activities,Behaviors,C++ - DLLs) weiterverarbeitet werden, um eine entsprechende Reaktion des Roboters festzulegen. Die dazu nötigen Steuerungsdaten, werden dann über ein serielles Kabel oder ein Funkmodem an den Roboter gesendet und dort in die Tat umgesetzt.

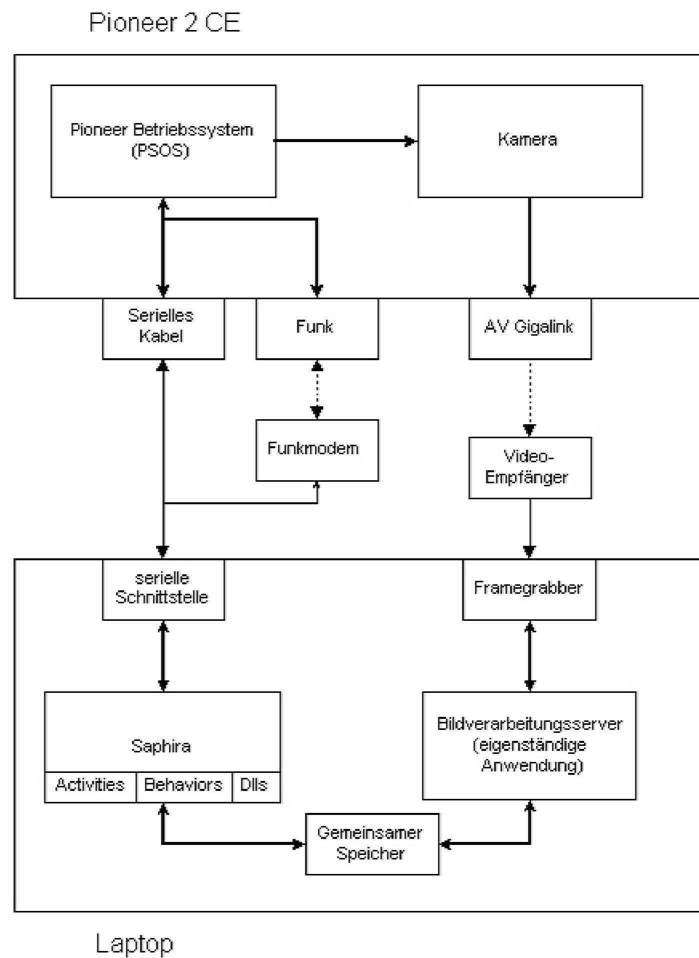

Abbildung 8: Datenfluß

Wie oben angesprochen, wird diese Arbeit in die Themen *Bildverarbeitung* und *Robotersteuerung* unterteilt. Für beide Gebiete werden nacheinander zunächst ein theoretischer Überblick, danach meine konzeptionellen Überlegungen und schließlich die genaue Form der Implementierung dargelegt.

Den Anfang macht ein theoretischer Überblick über *digitale Bildverarbeitung*.

8 Einführung in die digitale Bildverarbeitung

8.1 Was ist digitale Bildverarbeitung?

In den vergangenen Teilen wurde schon oft der Begriff *Bildverarbeitung* erwähnt und auch seine Wichtigkeit für die Lösung der gestellten Aufgabe betont. In diesem Abschnitt möchte ich daher etwas genauer auf die *digitale Bildverarbeitung* eingehen und entsprechende Verfahren, die für diese Arbeit von Bedeutung sind, vorstellen. Die Grundlage dafür bilden die Bücher [Hab91], [Hab95], [Bäs98] und [Ste93].
Doch zunächst allgemein zum Begriff *digitale Bildverarbeitung*. Wenn man nun in der Literatur nach einer genauen Definition für den Begriff *digitale Bildverarbeitung* sucht, wird man kaum etwas finden. Meist wird er dort nur über die Anwendungsmöglichkeiten erklärt. Im einfachsten Fall lässt sich jedoch dazu sagen, dass sich die digitale Bildverarbeitung mit der Verarbeitung von visuellen Informationen auf Digitalrechnern beschäftigt. Als die 5 sich daraus ergebenden wesentlichen Anwendungsbereiche werden in [Bäs98] genannt:

- *Generierung von Bildern* in Bereichen wie Desktop Publishing
- *Bildübertragung*, also der Transport von Bildern über Datenleitungen und damit verbunden Verfahren zur Kompression der enormen Bilddatenmengen
- *Bildbearbeitung*, also die Entfernung von Störungen, die Veränderung von Bildern zur Unterstützung der Bildanalyse durch Menschen sowie Veränderungen aus ästhetischer Sicht. (Solche Verfahren können auch als Vorstufe der Bild- und Szenenanalyse genutzt werden, wobei man von Bildvorverarbeitung spricht.)
- *Bildanalyse*, hier geht es um das Extrahieren von Bildinformationen im Sinne der Messtechnik, also z.B. die Erkennung von Schriftzeichen
- *Szenenanalyse*, im Gegensatz zum eher industriellen Einsatz der *Bildanalyse*, erstreckt sich dieses Gebiet in die Bereiche der Künstlichen Intelligenz, wo es z.B. um das „elektronische Auge“ autonomer Roboter geht

Die vorliegende Arbeit fällt also in den letzten Anwendungsbereich, wobei zur Vorverarbeitung natürlich auch hier Verfahren des Punktes *Bildbearbeitung* eingesetzt

werden können.

8.2 Digitalisierte Grauwertbilder

Die Grundlage für die digitale Bildverarbeitung ist das digitale Bild. Das normale Bild an sich stellt ja ein analoges Signal dar. Um es nun mit einem Rechner verarbeiten zu können, muss es zunächst in eine rechnerkompatible Form überführt werden. Dieser Vorgang wird als *Digitalisierung* bezeichnet und das entsprechende Ergebnis ist ein *digitales Bild*. Die Digitalisierung besteht aus den zwei Schritten Rasterung (oder Abtastung) und Quantisierung.

Bei der Rasterung wird das zu digitaliserende Bild mit einem rechteckigen oder quadratischen Raster überlagert und in Rasterflächenstücke unterteilt. Die Größe des Rasters hat dabei einen entscheidenden Einfluß auf die Qualität des digitalisierten Bildes und somit auch auf die weitere Bildverarbeitung. Ist das Raster zu grob, können wichtige Bildinhalte verloren gehen, ist es dagegen zu fein, fallen große Datenmengen an, was sich wiederum in erhötem Speicher- und Rechenaufwand niederschlägt.

Als Quantisierung bezeichnet man den Vorgang, dass jedem Rasterstück ein Farbwert aus einer vorher definierten Menge zuordnet wird. Für ein Grauwertbild ist dies z.B. die Menge $G = \{0, 1, \ldots, 255\}$, wobei 0 als Schwarz und 255 als Weiß interpretiert wird, für ein Binärbild dagegen entweder 0 oder 1. Zur besseren Visualisierung wird bei Binärbildern anstatt 0 und 1 oft 0 und 255 verwendet. Die Quantisierung hat in der Regel keinen so großen Einfluß auf die Bildqualität wie die Rasterung. Je nach Anforderung an das Bildverarbeitungssystem heißt es aber genau abzuwägen, welche Eigenschaften das digitalisierte Bild haben soll, damit die gestellten Aufgaben optimal gelöst werden können.

Ein (einkanaliges) digitales Grauwertbild lässt sich also wie folgt beschreiben:

$G = \{0, 1, \ldots, 255\}$	Grauwertmenge
$\mathbf{S} = (s(x, y))$	Bildmatrix des Grauwertbildes
$x = 0, 1, \ldots, L - 1$	Bildzeilen
$y = 0, 1, \ldots, R - 1$	Bildspalten
(x, y)	Ortskoordinaten des Bildpunktes
$s(x, y) \in G$	Grauwert des Bildpunktes

Ein digitales Grauwertbild stellt somit eine rechteckige Matrix $\mathbf{S} = (s(x,y))$ mit Bildzeilen und Bildspalten dar. Der Zeilenindex ist x und der Spaltenindex y. Der Bildpunkt an der Stelle (Zeile,Spalte)=(x, y) besitzt den Grauwert $s(x,y)$, welcher aus der Menge $G = \{0, 1, \ldots, 255\}$ stammt.
Ein digitales Bild kann auch als Funktion $s(x,y)$ zweier diskreter (Orts-)Variablen x und y aufgefaßt werden. Es gilt dann wiederum:

$x = 0, 1, \ldots, L-1$
$y = 0, 1, \ldots, R-1$ und
$s(x,y) \in G$

wobei G die Menge aller möglichen Grauwerte ist.

8.3 Mehrkanalige Bilder

Bis jetzt wurden nur einkanalige Bilder betrachtet, also z.B. Grau- und Binärbilder. Oft hat man es jedoch mit Farbbildern bzw. mehrkanaligen Bildern zu tun. Für diese muss das bisherige Modell erweitert werden. Ein mehrkanaliges Bild ist dann definiert durch $\mathbf{S} = (s(x,y,n))$, wobei x und y wieder die Zeilen- und Spaltenzähler darstellen. Hinzugekommen ist jetzt der Kanalzähler n, der bei einem N-kanaligen Bild von 0 bis $N-1$ läuft. Ein N-kanaliges Bild kann man sich als dreidimensionale Bildmatrix vorstellen, deren Kanäle als Schichten hintereinanderliegen. Ein Punkt dieses Bildes ist dann ein N-dimensionaler Vektor

$$\vec{s}(x,y) = (g_0, g_1, \ldots, g_{N-1})^T,$$

wobei die Komponenten g_n aus der Grauwertmenge G sind. Jeder Kanal entspricht somit einem eigenen Grauwertbild.
Für ein Farbbild ist es üblich, je einen Rot-, Grün- und Blaukanal zu definieren, woraus sich $N = 3$ ergibt. Die Bedeutung der einzelnen Kanäle kann dabei willkürlich gewählt werden. Für Windows-Bitmaps, wie sie auch in der vorliegenden Arbeit verwendet werden, sind die Kanäle jedoch schon folgendermaßen definiert: $(s(x,y,0))$ ist der Blauanteil, $(s(x,y,1))$ ist der Grünanteil und $(s(x,y,2))$ der Rotanteil.

8.4 Einfache Grauwerttransformationen

Die einfachste Form von Bildverarbeitungsfunktionen stellen die Grauwertmanipulationen dar. Hauptsächlich werden diese Methoden zur Bildverbesserung, z.B. durch Kontrastverstärkung, eingesetzt, aber auch eine einfache Form der Segmentierung, die Binarisierung, ist durch sie möglich. Weitere Anwendungsmöglichkeiten sind das Aufhellen oder Abdunkeln von Bildern. Das Grundprinzip dieser Methode ist, dass jedem im Eingabebild vorkommenden Grauwert ein neuer Grauwert zugeordnet wird.
Mathematisch sieht dies wie folgt aus:
Vorraussetzung ist ein einkanaliges Grauwertbild $\mathbf{S} = (s(x, y))$ mit der Grauwertmenge $G = \{0, 1, \ldots, 255\}$ Eine Grauwertmanipulation ist dann eine Abbildung $f : G \rightarrow G$ wobei f die Funktion der Grauwerttransformation ist. Für f ist im Prinzip jede Funktion geeignet, die über G definiert und beschränkt ist:

$$min\{f\} > -\infty; \quad max\{f\} < +\infty$$

Ist dies der Fall, dann kann f zu f_n normiert werden, so dass $f : G \rightarrow G$ erfüllt ist:

$$f_n(x) = \frac{f(x) - min\{f\}}{max\{f\} - min\{f\}} \cdot c$$

Dabei ist c ein geeignet zu wählender Skalierungsfaktor (z.B. 255). Wird f_n auf das (Eingabe-)Grauwertbild $\mathbf{S}_e$ angewendet, so berechnet sich das Ergebnisbild $\mathbf{S}_a$ wie folgt:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a :$$
$$s_a(x, y) = f_n(s_e(x, y)), 0 \leq x \leq L - 1, 0 \leq y \leq R - 1.$$

In der Praxis wird man die Grauwerttransformation $\mathbf{S}_e \rightarrow \mathbf{S}_a$ nicht wie eben beschrieben implementieren. Auf Grund der diskretisierten Grauwertmenge kann man stattdessen eine *look-up*-Tabelle zu verwenden. Eine look-up-Tabelle besteht aus 256 Speicherplätzen, und enthält zu der Grauwertmenge $G = \{0, 1, \ldots, 255\}$ die entsprechenden Werte von $f_n(g), g \in G$. Da die look-up-Tabelle nur einmal vor Beginn der Transformation gefüllt werden muss, wird wertvolle Rechenzeit eingespart.

Binarisierung Wie schon erwähnt wurde, ist das Binarisieren eine Form der Grauwertmanipulation. Um eine Binarisierung durchführen zu können, muss zunächst ein Schwellwert c definiert werden. Alle Grauwerte des Eingabebildes, die kleiner als der Schwellwert sind, werden im Ausgabebild auf 0 gesetzt und alle anderen Werte auf 1. Die Funktion $f_n(g)$ sieht dann folgendermaßen aus:

$$f_n(g) = \begin{cases} 0 & \text{falls } g = s(x,y) \leq c, \\ 1 & \text{sonst.} \end{cases}$$

Werden dabei anstatt der Grauwerte 0 und 1 beliebige Grauwerte g_1 und g_2 eingesetzt, dann wird von einem *Zweipegelbild* gesprochen. Auf Grund der Tatsache, dass die Grauwerte 0 und 1 auf einem Bildschirm nicht zu unterscheiden sind, werden zur Visualisierung von Binärbildern häufig Zweipegelbilder mit den Werten $g_1 = 0$ und $g_2 = 255$ eingesetzt.

8.5 Operationen im Ortsbereich

Die nächste Form von Bildverarbeitungsmethoden sind die sog. *lokalen Operatoren.* Diese werden häufig zur Beseitigung von Störungen (Glättung) oder aber zur Verstärkung von Grauwertdifferenzen (Kantenfilter) eingesetzt. Grundlage für diese Filteroperationen ist ein Operatorfenster bzw. eine Faltungsmaske. Zur Bestimmung des Ausgabewertes, des jeweils bearbeiteten Eingabepixels, betrachtet man dann nicht mehr nur den Grauwert des aktuellen Pixels, sondern auch die Grauwerte in seiner Umgebung. Die Anzahl und entsprechende Wichtung der Nachbarschaftspixel ergibt sich dabei durch die Faltungsmaske. Nacheinander wird also um das jeweils aktuelle Pixel ein Operatorfenster geöffnet, das die in ihm enthaltenen Pixelwerte gewichtet zu einem Ergebnisgrauwert verknüpft. In der Praxis werden dabei meist quadratische Masken mit den Größen 3x3 oder 5x5 eingesetzt.

Vorraussetzung sei also ein Grauwertbild $\mathbf{S}_e = (s_e(x,y))$ und eine Faltungsmaske $\mathbf{H} = (h(u,v))$. Die Bildpunkte $s_a(x,y)$ des Ausgabebildes $\mathbf{S}_a$ ergeben sich durch eine additiv, gewichtete Verknüpfung des Bildpunktes $s_e(x,y)$ mit seinen benachbarten Bildpunkten. Dies enstpricht einer *Faltung* des Bildes $\mathbf{S}_e$ mit der

Maske **H** und kann folgendermaßen beschrieben werden:

$$\mathbf{S}_e \rightarrow \mathbf{S}_a : \quad s_a(x,y) = \frac{1}{m^2} \sum_{u=0}^{m-1} \sum_{v=0}^{m-1} s_e(x+k-u, y+k-v) \cdot h(u,v)$$

wobei m die Größe der Maske **H** angibt und für k gilt: $k = \frac{m-1}{2}$. Für die oben erwähnten 3x3 oder 5x5-Masken gilt dann also $m = 3$ bzw. $m = 5$.
Je nach den in **H** enthaltenen Koeffizienten unterscheidet man *Summenoperatoren* (nur positive Koeffizienten) und *Differenzoperatoren* (positive und negative Koeffizienten). Bei Differenzoperatoren kann auf den Normierungsfaktor $\frac{1}{m^2}$, der bei Summenoperatoren dafür sorgen soll, dass der Mittelwert des Bildes erhalten bleibt, verzichtet werden. Soll das Ergebnis jedoch wieder in die Grauwertmenge $G = \{0, \ldots, 255\}$ abgebildet werden, dann müssen die Werte noch geeignet skaliert werden. Differenzoperatoren werden hauptsächlich zur Kantenerkennung eingesetzt. Wenn die Summe der Koeffizienten von **H** bei Differenzoperatoren den Wert 0 hat, dann gilt, dass der Operator für homogenen Bildbereiche den Wert 0 liefert und sonst eine Maßzahl für die „Stärke“ des Übergangs.

Ein Problem bei dem oben genannten Algorithmus stellt die Verarbeitung von Randpixeln dar. Da um das jeweils aktuelle Pixel ein Operatorfenster geöffnet wird, steht man bei Randpixeln vor dem Problem, dass dieses Fenster über den Bildrand hinausragt. Als Lösung dafür gibt es verschiedene Ansätze. Der erste ist, dass man die Randpixel einfach nicht bearbeitet. Für das Ausgabebild muss man sich dann aber überlegen, ob die Randpixel dort ebenfalls weggelassen oder ob sie mit einem bestimmten Wert gefüllt werden sollen. Lässt man sie einfach weg, dann ist das Ausgabebild um genau den Rand kleiner als das Eingabebild. Dies ist in der Praxis meist unerwünscht und hätte auch den Effekt, dass bei mehreren Bildverarbeitungsschritten hintereinander das Ergebnisbild immer kleiner werden würde. Normalerweise füllt man den Randbereich des Ausgabebildes daher mit konstanten Werten (z.B. 0, 127 oder 255). Weiterhin kann man natürlich auch die Originaldaten verwenden, was aber bei weiteren Operationen Störungen verursachen kann. Wenn man die Randpixel dagegen mitbearbeiten will, so kann man das Problem auch lösen, indem man sich das Eingabebild gespiegelt oder periodisch vorstellt und die Bilddaten vom oberen bzw. unteren und rechten bzw. linken Bildrand verwendet.

Nachfolgend sind einige Beispiele für Faltungsmasken aufgeführt:

$$\mathbf{H} = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad \text{3x3-Mittelwert-Operator}$$

$$\mathbf{H} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad \text{3x3-Laplace-Operator}$$

8.6 Kantendetektion

Eine der zentralen Bildverarbeitungsmethoden ist die Kantendetektion. Auf Grund der Wichtigkeit dieser Methode für meine Arbeit wird an dieser Stelle ein Überblick darüber geben, was eine Kante in einem Bild überhaupt darstellt und welche Verfahren es gibt, diese herauszufiltern.

8.6.1 Der Begriff der *Kante*

Prinzipiell meint man bei Kanten Grauwertkanten, also solche, die man aus einkanaligen Grauwertbildern extrahieren kann. Von einer Farbkante spricht man, wenn in mindestens einem Farbkanal eine Grauwertkante auftritt [Hab91]. Man kann also für Farbbilder dieselben Algorithmen nutzen, indem man sie auf die einzelnen Farbkanäle anwendet und die Ergebnisse geeignet kombiniert.
Doch was ist nun eine Kante?
In einem Bild besitzen die einzelnen Pixel unterschiedlich intensive Grau- bzw. Farbwerte. Eine Kante ist dann eine Diskontinuität im Verlauf dieser Intensitätswerte [Ste93]. Daraus ergibt sich, dass Kantenpixel hohe lokale Grauwertdifferenzen aufweisen. Letzteres gilt allerdings auch für Bildstörungen wie Rauschen, die aber keine Kanten darstellen (siehe dazu auch „Anforderungen und Probleme der Kantendetektion“).

8.6.2 Kantenmodelle

In [Ste93] werden die folgenden verschiedenen Modelle für Kanten definiert (vergleiche Abbildung 9):

- ideale Stufenkante
- ideale Rampenkante
- ideale Dachkante
- ideale Treppenkante
- reale Kantenform (Abbildung 10 auf der nächsten Seite)

Die reale Kantenform ist dabei meist eine mit Störungen, wie Rauschen, überlagerte ideale Kante. Oft sind auch mehrere ideale Kantenformen in der realen Kante überlagert wiederzufinden. Diese Tatsache erschwert natürlich die Kantenerkennung.

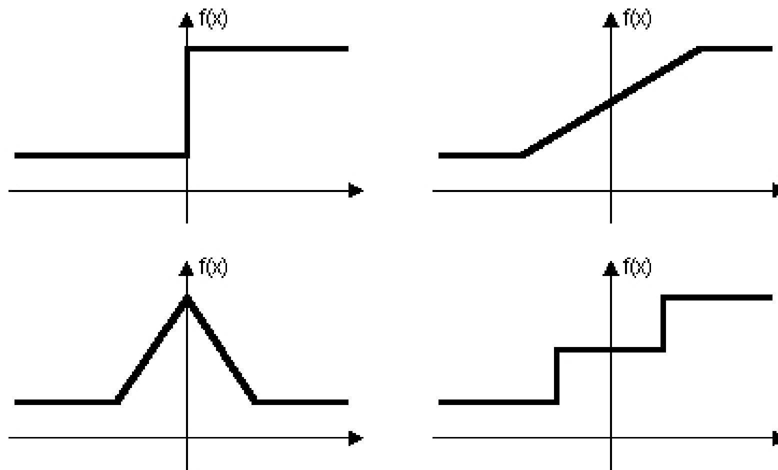

Abbildung 9: Kantenmodelle. Links oben: ideale Stufenkante; Rechts oben: ideale Rampenkante; Links unten: ideale Dachkante; Links unten: ideale Treppenkante

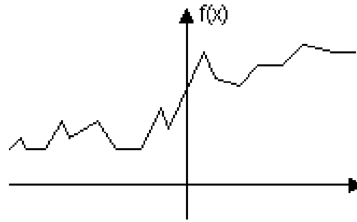

Abbildung 10: Eine reale Kante ist meist eine Kombination idealer Kantenprofile, überlagert mit Rauschen

8.6.3 Anforderungen und Probleme der Kantendetektion

Die Methoden der Kantendetektion sollten natürlich eine möglichst geringe Fehlerrate besitzen, doch was heißt das?
Tatsächliche Kantenpunkte sollten auch als solche erkannt werden, während Punkte, die nicht zu einer Kante gehören, auch nicht als solche markiert werden dürfen. Des Weiteren sollte eine Kante möglichst an ihrer wirklichen Position (Lokalisation) und auch nur einmal erkannt werden. Zusätzlich spielen noch subjektive Punkte und numerische Kriterien, wie Berechenbarkeit, eine wichtige Rolle. Diese ganzen Anforderungen sind natürlich nicht leicht zu erfüllen, da wie schon festgestellt wurde, die normalerweise im Bild anzutreffende Kante durch Rauschen gestört ist. Das Problem bei dem Rauschen ist, dass es sich dabei, ebenso wie bei der zu detektierenden Kante um hochfrequente Bildanteile handelt, die durch einen Kantenfilter zusätzlich noch verstärkt werden. Um nun die Kanten möglichst korrekt zu ermitteln, ist es daher ratsam, vor dem eigentlichen Kantenfilter geeignete Verfahren zur Bildverbesserung auf das Bild anzuwenden, um das Rauschen zumindest teilsweise zu unterdrücken. Hauptsächlich kommt hierfür ein Glättungsoperator in Frage. Das Problem hierbei ist jedoch, dass durch seine Filtereigenschaften die Kanten verschmieren können und damit die Güte der Lokalisation verschlechtert wird. Die Algorithmenkette ist dann:

f(x,y) ⟶ [Glättung] ⟶ [Kantenverstärkung] ⟶ [Binarisierung] ⟶ Kantenbild

Zunächst wird also ein Glättungsfilter angewendet, der das Rauschen entfernt, danach die eigentliche Kantendetektion durchgeführt und zum Schluß das Bild mit einem geeigneten Schwellwert binarisiert.

8.6.4 Einfache Verfahren zur Kantenextraktion

Wie schon erwähnt wurde, zeichnen sich Grauwertkanten dadurch aus, dass sie in einer lokalen Umgebung hohe Grauwertdifferenzen aufweisen. Es liegt also nahe, einen lokalen Differenzoperator zur Kantenerkennung einzusetzen. Die Grundlagen dazu wurden im Abschnitt „Operationen im Ortsbereich“ erläutert. Das einfachste Verfahren einer Kantenextraktion ist die Verwendung eines lokalen Filters, der direkt ein kantenextrahiertes Bild liefert. Ein Beispiel für solch einen Filter ist der *Laplace-Operator*. Dieser kann mit den folgenden Masken nachgebildet werden:

$$\mathbf{H} = \begin{pmatrix} 0 & -1 & 0 \\ -1 & 4 & -1 \\ 0 & -1 & 0 \end{pmatrix} \quad \mathbf{H} = \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \quad \mathbf{H} = \begin{pmatrix} 1 & -2 & 1 \\ -2 & 4 & -2 \\ 1 & -2 & 1 \end{pmatrix}$$

Der Laplace-Operator ist dabei nicht auf eine Größe von 3x3 festgelegt, sondern kann auch auf 5x5 oder noch größere Filtermasken skaliert werden. Ein Nachteil dieses Operators ist seine Anfälligkeit gegenüber Bildstörungen, daher sollte gerade vor seiner Anwendung eine Rauschfilterung durchgeführt werden.

8.6.5 Kantenbeschreibung mit Gradienten

Eine zweite Möglichkeit der Kantendetektion ist die Bestimmung des sog. *Gradienten*. Dabei wird eine partielle Differentiation in Zeilen- und Spaltenrichtung nachgebildet. Das Ergebnis sind Betrag („Stärke“) und Richtung der Grauwertänderung. Um diese Operation durchführen zu können, benötigt man zunächst zwei verschiedene Filterkerne $\mathbf{H}_x$ und $\mathbf{H}_y$, die im einfachsten Fall wie folgt aussehen:

$$\mathbf{H}_x = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 0 \\ 0 & 1 & 0 \end{pmatrix} \quad \mathbf{H}_y = \begin{pmatrix} 0 & 0 & 0 \\ 0 & -1 & 1 \\ 0 & 0 & 0 \end{pmatrix}$$

Durch Anwendung dieser beiden Filter auf das Eingabebild entstehen die zwei Ausgabebilder $\mathbf{S}_x = (s_x(x, y))$ und $\mathbf{S}_y = (s_y(x, y))$. Der Betrag des Gradienten berechnet sich dann durch

$$l = \sqrt{s_x(x, y)^2 + s_y(x, y)^2}$$

und die Richtung des Gradienten durch

$$\tan \varphi = \frac{s_x(x,y)}{s_y(x,y)}$$

In der Regel ist der genaue Wert des Gradienten nicht von Interesse, sondern nur seine relative Größe. Um nun Rechenaufwand einzusparen, wird anstatt der oben genannten Betragsformel häufig Folgendes verwendet:

$$l = \mid s_x(x,y) \mid + \mid s_y(x,y) \mid$$

Da auch die beiden gezeigten einfachen Filterkerne sehr störungsanfällig sind, werden sie in der Praxis seltener eingesetzt. Besser geeignet ist z.B. der *Sobeloperator*, welcher zusätzlich noch eine Glättung quer zur Differenzierungsrichtung enthält und zur Differenzbildung jeweils nur die übernächste Zeile bzw. Spalte miteinbezieht. Der 3x3-Sobeloperator besteht aus den folgenden Masken:

$$\mathbf{H}_x = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \quad \mathbf{H}_y = \begin{pmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{pmatrix}$$

8.7 Morphologische Operationen

Die morphologische Bildverarbeitung leitet sich aus dem Begriff Morphologie ab, welcher „Lehre von der Form“ bedeutet, und stellt eine Menge von nichtlinearen lokalen Operatoren dar. Die Grundidee dabei ist, diese Operatoren gemäß den Formen der zu bearbeitenden Bildbereiche zu gestalten.
Im Mittelpunkt der morphologischen Operationen steht daher eine lokale Operatormaske, die hierbei als *Strukturelement* bezeichnet wird und relativ frei definiert werden kann. In diesem Strukturelement sind eine bestimmte Menge an Punkten markiert und definieren die für einen Bildpunkt für die morphologische Operation zu berücksichtigenden Nachbarn.
Im Wesentlichen unterscheidet man nun zwei morphologische Operatoren, die *Erosion* und die *Dilatation*.
Bei der Anwendung dieser Operatoren auf Binärbilder gilt Folgendes:

- Das Strukturelement (Operator-Fenster) wird zunächst nacheinander an alle möglichen Positionen im Bild gelegt.
- Bei der Erosion wird der aktuelle Bildpunkt auf den Wert 1 gesetzt, wenn *sämtliche* Bildpunkte eines Bereiches bzw. Objektes des Ursprungsbildes vom Strukturelement überdeckt werden (also den Wert 1 haben)
- Bei der Dilatation wird der aktuelle Bildpunkt auf den Wert 1 gesetzt, wenn mindestens ein Bildpunkt eines Bereiches im Ursprungsbild vom Strukturelement überdeckt wird

Die Erosion trägt daher Pixel von Bereichs- bzw. Objekträndern ab, während die Dilatation Bildpunkte hinzufügt. Die Art des Abtrages bzw. der Hinzufügung wird dabei vom Strukturelement bestimmt. Zu beachten ist noch, dass die Erosion nicht die Umkehroperation der Dilatation darstellt.

Besonders wichtig bei der Bildverarbeitung mit morphologischen Operatoren ist noch eine Kombination beider. Man spricht dabei von *Opening*- und *Closing*-Operationen. Diese sind folgendermaßen charakterisiert:
Ein Opening ist eine Erosion gefolgt von einer Dilatation und dient im Wesentlichen dem Abtragen „ausgefranster“ Bereichsränder und dem Eliminieren kleiner Bereiche.
Ein Closing ist umgekehrt eine Dilatation gefolgt von einer Erosion und schließt die Lücken zwischen „Fransen“. Sie kann daher zum Schließen von Objektkonturen nach Anwendung eines Kantenfilters dienen.

8.8 Segmentierung

Zwei wesentliche Anwendungsbereiche der digitalen Bildverarbeitung sind die Bild- und die Szenenanalyse. Mit den bisher vorgestellten Verfahren ist es zunächst möglich, beispielsweise Störungen zu entfernen oder Grauwertdifferenzen hervorzuheben. Um nun aber die Bildinformationen richtig interpretieren zu können, bedarf es weiterer Methoden. Eine davon ist die *Segmentierung*. Der Begriff *Segmentierung* hat in der Bildverarbeitung die Bedeutung, Bildpunkte, die die gleichen oder ähnliche Eigenschaften besitzen, zu größeren Einheiten zusammenzufassen.

Segmentierung mit Schwellwertbildung

Die einfachste Form der Segmentierung stellt die Binarisierung dar. Sie wird oft verwendet, wenn man auf einem Bild ein Objekt sucht, dass sich gut vom Hintergrund absetzt, also z.B. sehr hell ist auf einem dunklen Untergrund. Mit einem geeigneten Schwellwert kann man eine Binarisierung durchführen, die das Objekt sauber vom Hintergrund trennt. Das Ergebnis ist dann ein Binär- oder Zweipegelbild, in dem der Hintergrund den Wert 0 und die Objektpixel den Wert 1 bzw. 255 besitzen. Der optimale Schwellwert in diesem Fall kann durch eine *Histogrammanalyse* ermittelt werden. Ein *Histogramm* ist ein einfaches Diagramm, das die Grauwertverteilung im Bild darstellt. Zu jedem möglichen Grauwert wird die Anzahl der im Bild auftretenden zugehörigen Bildpunkte eingetragen. Hat das Histogramm zwei deutlich voneinander getrennte Maxima, also wenn z.B. wie oben das Objekt hell und der Hintergrund eher dunkel ist, so wird es als *bimodal* bezeichnet. Ist dies der Fall, dann kann als Schwellwert der Wert des Tales zwischen den Grauwertmaxima verwendet werden. Dasselbe Prinzip kann natürlich auch genutzt werden, wenn mehrere unterschiedlich farbige Objekte in dem Bild vorhanden sind. Das Bild wird dann zwar nicht binarisiert, aber jedem Bildpunkt ein Wert gegeben, um die Zugehörigkeit zu einer bestimmten Farbgruppe zu signalisieren. Zum Beispiel kann für die Grauwerte $\{0, \ldots, 50\}$ der Wert 0, für die Grauwerte $\{51, \ldots, 150\}$ der Wert 1 und für den Rest der Wert 2 vergeben werden, was dann eine Aufteilung Hintergrund (0), Objekt1 (1), Objekt2 (2) darstellen könnte.

Segmentierung über Templates

Wenn die Anzahl der im Bild vorkommenden, unterschiedlichen Objekte eng begrenzt und die jeweilige Form bekannt ist, dann kann eine Segmentierung direkt über entsprechende Schablonen (Templates) erfolgen. Die Schablonen werden dazu in den möglich vorkommenden Orientierungen an alle Positionen auf das Bild gelegt und dann die darunterliegenden Bildpunkte analysiert. Wenn diese Pixel dann in Form und Grauwert zu einem vorher bestimmten Prozentsatz der Schablone übereinstimmen, werden sie dem entsprechenden Objekt zugeordnet. Dieses Verfahren wird häufig in der industriellen Umgebung angewendet, da dort die Beleuchtung fast ideal ist, und das zu segmentierende Bild daher nur noch wenige Graustufen enthält.

Kantenbasierte Segmentierung

Eine weitere Möglichkeit der Segmentierung stellt die kantenbasierte Segmentierung dar. Da die im Bild detektierten Kanten oft den Konturen der enthaltenen Objekte entsprechen, können die Kantenerkennungsmethoden zur Segmentierung eingesetzt werden. Im Gegensatz zu den vorher vorgestellten Verfahren hat man dann natürlich nicht mehr die Objekte als extrahierte Flächen, sondern nur noch ihre Umrandungen. Ein Problem dabei ist, dass diese Umrandungen nach Durchführung einer Kantendetektion selten vollständig geschlossen sind. Um dies zu beheben, müssen Verfahren angewendet werden, die die Lücken schließen wie z.B. ein Maximumoperator.

Segmentierung durch Gebietswachstum

Vorraussetzung für dieses Verfahren ist, dass man schon vorher den Grau- bzw. Farbwert der gesuchten Fläche kennt. Im Bild wird dann ein erster Repräsentant dieser Klasse gesucht und als Ausgangspunkt für das Flächenwachstum verwendet. Danach wird seine Umgebung betrachtet und alle Bildpunkte, die ebenfalls (bis auf eine Toleranz) in die Klasse fallen, der Region hinzugefügt. Um kleinere Störungen zu kompensieren, kann man anstatt einzelner Pixelwerte auch den Mittelwert einer ganzen Umgebung betrachten.

Farbbasierte Segmentierung mit Differenzbildern

Eine Möglichkeit der Segmentierung stellt, wie oben schon erwähnt, die Segmentierung über Schwellwertbildung dar, also eine Aufteilung über Farbwerte. Hierbei wird jedoch üblicherweise von Grauwertbildern ausgegangen. Bei Farbbildern hat man das Problem, dass hier mehrere Kanäle vorhanden sind, die es zu berücksichtigen gilt. Oft wird in Farbbildern aber nur nach bestimmten Farben (z.B. ROT) gesucht. Um diese nun herausfiltern zu können, kann man im Wesentlichen zwei Ansätze verfolgen. Der erste ist, dass man die genaue Farbkombination (in der Regel die RGB-Werte) der gesuchten Farbe kennt und davon ausgeht, dass diese auch in den zu verarbeitenden Bildern konstant bleibt. Nacheinander können nun die Werte der Bildpunkte mit der Vorgabe verglichen werden, und unter Verwendung einer Toleranzgrenze bestimmt werden, ob sie der gesuchten Farbe zugehörig sind oder nicht. Das Ergebnis wäre dann wieder ein Binär- bzw. Zweipegelbild, oder wenn

gleichzeitig nach mehreren Farben gesucht wurde, ein Bild, das die Zugehörigkeit der Pixel zu den vorgegebenen Farben (z.B. über Indices) enthält. Nachteil bei diesem Verfahren ist die Anfälligkeit gegenüber wechselnden Lichtverhältnissen, da bei unterschiedlicher Beleuchtung auch die Objekte andere Farbmerkmale besitzen. Für die korrekte Arbeitsweise ist daher eine konstante Beleuchtung zwingend erforderlich. Ein zweiter Ansatz, der auch schon im Bildverarbeitungsserver eingesetzt wird, ist die Verwendung von Differenzbildern. Dieses Verfahren kann angewendet werden, wenn man nach reinen Farben bzw. solchen, die sich in ihren einzelnen Farbkanälen durch hervorstechende Merkmale festlegen lassen, sucht. Beispiele hierfür wären die Farben ROT oder $GRÜN$. Hierbei wird dann nicht mehr nach der genauen Farbkombination, sondern nach ebendiesen Merkmalen gesucht. Wenn in einem Bild jetzt die Farbe rot gesucht wird, dann wird also nicht nach beispielsweise $RGB = \{255, 0, 0\}$, sondern nach Pixeln, deren Rotanteil deutlich größer ist, als der Blau- und Grünanteil. In der Praxis werden dafür häufig Differenzbilder verwendet. Differenzbilder entstehen, indem ein Eingabebild $\mathbf{S}_{e2}$ von einem anderen Bild $\mathbf{S}_{e1}$ abgezogen wird. Die Bildpunkte des Ausgabebildes $\mathbf{S}_a$ berechnen sich dann durch:

$$s_a(x, y) = s_{e1}(x, y) - s_{e2}(x, y)$$

Für die Farbsuche kann man Differenzen der Grauwertbilder der verschiedenen Farbkanäle verwenden. Ist man z.B. auf der Suche nach roten Objekten, dann reicht häufig das Differenzbild $ROT - GRÜN$ aus. Da durch die Differenz auch negative Werte auftreten können, sollte das Ergebnisbild wieder in den normalen Grauwertraum transformiert werden. Dies kann z.B. dadurch erreicht werden, dass man negative Werte auf 0 setzt. Anschließend empfiehlt sich eine Binarisierung mit einem geeigneten Schwellwert. In unserem Beispiel der Rotsuche würde er dann die Stärke des Rots angeben. Vorteil des Verfahrens ist, dass es auch bei wechselnden Lichtverhältnissen noch gut funktioniert, solange die Relationen in den Farbkanälen mit den vorher definierten Merkmalen übereinstimmen.

8.9 Weiterverarbeitung segmentierter Bilder

Nach Durchführung einer Segmentierung wurden die Bildpunkte erstmal „nur" klassifiziert, also einer bestimmten Merkmalsgruppe zugeordnet. Wenn man auf dem

Bild nur ein zu erkennendes Objekt erwartet, dann wird man nach der Segmentierung schon das gewünschte Endergebnis haben, also die einzelnen Bildpunkte entweder dem Objekt oder dem Hintergrund zugeordnet haben. Die Objektpixel können dann schon direkt weiterverarbeitet werden, es kann z.B. der Schwerpunkt des Objekts (der Objektpixel) bestimmt werden. Oft wird man jedoch nicht nur ein zu erkennendes Objekt auf dem Bild haben, sondern mehrere. Wenn diese dann alle unterschiedliche Merkmale, also z.B. unterschiedliche Farben, besitzen sollen, stellt dies auch noch kein größeres Problem dar, da die entsprechenden Bildpunkte ebenfalls gleich weiterverarbeitet werden können. Problematischer wird es, wenn mehrere gleichartige Objekte (Segmente) im Bild auftreten, da jetzt nach **zusammenhängenden** Regionen gesucht werden muss. Abbildung 11 zeigt ein Beispiel dazu. Wie zu sehen ist, gibt es zwei unterschiedliche Objekte, die zwar denselben

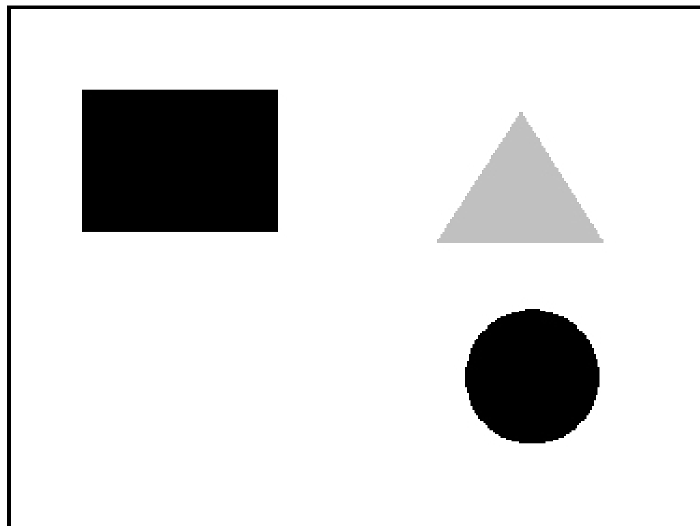

Abbildung 11: Bild nach einer Segmentierung

Farbwert besitzen, aber ansonsten absolut nichts miteinander zu tun haben. Es muss also ein Algorithmus gesucht werden, der dies erkennt und die vorsegmentierten Bilddaten in einzelne zusammenhängende Objekte aufteilt. Die einfachste Möglichkeit hierzu besteht darin, Bildpunkt für Bildpunkt durchzugehen, sich die Merkmalsklasse des aktuellen Pixels zu nehmen und mit seinen Nachbarn zu vergleichen, um dadurch die Zusammengehörigkeit zu einem bestimmten Objekt oder einer bestimmten Region festzustellen. Dabei hat man die Wahl, entweder eine 4er-

oder eine 8er-Nachbarschaft zu betrachten. Der Begriff 4er-Nachbarschaft bedeutet, dass man zu einem Bildpunkt jeweils seine horizontalen und vertikalen Nachbarpixel betrachtet. Bei der 8er-Nachbarschaft sind dies zusätzlich noch die in diagonaler Richtung angrenzenden Bildpunkte.
Nachdem die klassifizierten Bildpunkte in einzelne Objekte aufgeteilt wurden, werden sie in irgendeiner Form weiterverarbeitet. Da eine Weiterverarbeitung der einzelnen Objekte in Form von „Punktewolken“ recht ungünstig ist, wird man nach anderen objektbeschreibenden Parametern suchen. Einige mögliche Parameter sind der Flächenschwerpunkt, das kleinste objektumschreibende Rechteck oder eine Liste der Randpunkte. Will man aus dem Bild geometrische Objekte (z.B. Linien) extrahieren, so wird man dagegen die entsprechenden geometrischen Merkmale (z.B. Geradengleichung) nutzen.

8.10 Die Hough-Transformation als Verfahren zur Linienerkennung

Ein möglicher Algorithmus der Geradenextraktion ist die *Hough-Transformation*. Diese Methode ist sehr robust und kann auch verrauschte bzw. unterbrochene Linien noch korrekt detektieren kann.
Die Idee zur Hough-Transformation legte Paul V.C. Hough 1962 in einer Patentschrift nieder. Das Verfahren wurde aber erst später dazu eingesetzt, Geraden in digitalisierten Bildern zu erkennen. Grundlage für die Durchführung einer Hough-Transformation ist ein kantenextrahiertes Binärbild, welches die möglichen Kantenpunkte enthält. Ausgangspunkt für die Geradenextraktion ist nun, dass eine Gerade in der Ebene durch die Angabe von zwei Parametern festgelegt ist. Die bekannteste Form, um eine Gerade zu beschreiben, ist die *kartesische Normalform*, welche definiert ist durch:

$$y = m \cdot x + n$$

Dabei ist m der Anstieg der Geraden und n der Schnittpunkt der Geraden mit der y-Achse.
Um die Hough-Transformation durchführen zu können, benötigt man zunächst einen zweidimensionalen, diskreten *Parameterraum*, also eine zweidimensionale Matrix mit

endlich vielen Zeilen und Spalten. Die Zeilen und Spalten entsprechen dann den Parametern der Gerade, also z.B. Anstieg und Schnittpunkt mit der y-Achse. Daraus folgt, dass eine Gerade in der Ebene somit einem Punkt im Parameterraum entspricht. Des Weiteren ergibt sich aus dem diskretisierten Parameterraum, dass auch nur eine endliche Menge an Geraden repräsentiert werden kann. Dies hat zur Folge, dass man bei Geradengleichungen in kartesischer Normalform vor einem Problem steht. Da hier Geraden einen unendlich großen Anstieg haben können, können diese nicht mehr im Parameterraum dargestellt werden. Aus diesem Grund wird für die Hough-Transformation eine andere Form der Geradengleichung gewählt, die *Hessesche Normalform*. Diese ist definiert durch:

$$r = x \cdot cos\theta + y \cdot sin\theta$$

Dabei ist θ der Winkel zwischen der x-Achse und dem Lot vom Koordinatenursprung auf die Gerade und r die Länge des Lotes. Die Zeilen und Spalten des Parameterraumes repräsentieren nun also r und θ, man spricht daher auch von einem (r, θ)-Raum. Der Winkel θ liegt dabei im Intervall $0 \leq \theta < 2\pi$ und der maximale Abstand der Geraden vom Koordinatenursprung ergibt sich aus den Maßen des betrachteten Bildes. Da nur Punkte betrachtet werden, die innerhalb des Bildes liegen, ergibt sich, dass der maximal mögliche Abstand eines Punktes vom Ursprung des Bildes gegeben ist durch:

$$r_{max} = \sqrt{Bildbreite^2 + Bildh\ddot{o}he^2}$$

Jedes Feld des Parameterraumes, stellt nun einen Zähler dar, der zu Beginn der Transformation mit 0 vorbelegt ist. Dieser Raum wird daher auch als *Akkumulator* bezeichnet. Die eigentliche Hough-Transformation läuft nun folgendermaßen ab:
Für jeden Bildpunkt des Eingabebildes, der als möglicher Linienpunkt markiert ist, werden alle Geraden berechnet, die durch ihn laufen können (*Geradenbüschel*) und die entsprechenden Zähler im Akkumulator um 1 erhöht. Die, für ein Geradenbüschel zu erhöhenden, Akkumulatorzellen liegen dabei auf einer Kurve mit einem typischen Verlauf, welche auch als *sinoidale* Kurve bezeichnet wird. Als Verfahren zur Berechnung der Geradenbüschel wird üblicherweise folgender Algorithmus verwendet:

Die Geradengleichung war ja definiert durch

$$r = x \cdot cos\theta + y \cdot sin\theta$$

Für einen gegebenen Bildpunkt, hat man die Werte für x und y bereits. Um nun das Geradenbüschel zu erhalten, geht man jetzt einfache sämtliche Winkel (je nach Aufteilung des Parameterraumes) nacheinander durch, setzt die Werte von x, y und θ in die obige Gleichung ein und erhält den Wert für r. Das Ergebnis nutzt man, um die entsprechende Akkumulatorzelle (r, θ) zu inkrementieren. Die dabei notwendigen Berechnungen von Cosinus und Sinus werden in der Praxis über look-up-Tabellen erledigt, um Rechenzeit zu sparen.
Das Ergebnis der Hough-Transformation ist nun Folgendes:
Zu jedem Kantenpunkt wurden die Zähler der jeweils möglichen Geraden um 1 erhöht. Wenn nun mehrere Bildpunkte auf einer Geraden liegen, dann ist der Zähler dieser Gerade bei jedem Punkt erhöht worden und entspricht nun der Anzahl der auf dieser Geraden liegenden Punkte. Es lässt sich also sagen, dass jede Akkumulatorzelle, deren Wert größer als 1 ist, eine Gerade im Bild repräsentiert. Üblicherweise sucht man allerdings Geraden, die durch mehr als zwei Bildpunkte verlaufen. Es empfiehlt sich dann, mit einem Schwellwert zu arbeiten, der die minimale Anzahl an auf einer Geraden liegenden Bildpunkte angibt.

Wie man leicht erkennen kann, werden bei dem Algorithmus auch unterbrochene Linien korrekt erkannt. Des Weiteren werden natürlich auch mehrere im Bild existierende Linien registriert, da es durchaus möglich ist, dass im Parameterraum in mehreren Zellen größere Werte zu finden sind. Die Genauigkeit der erkannten Geraden richtet sich bei dem Verfahren nach der Aufteilung des Parameterraumes. Die Parameter können z.B. in 1er, 2er oder 5er-Schritten eingeteilt werden, wobei kleinere Werte, speziell bei den Winkeln, einen erhöhten Rechenaufwand zur Folge haben (siehe Berechnung der Geradenbüschel). Auf Grund der Diskretisierung des Raumes und der möglichen Bildstörungen hat man es in der Praxis bei einer erkannten Geraden auch nicht unbedingt mit nur **einem** Maximalwert in **einer** Akkumulatorzelle zu tun, sondern mit mehreren ähnlich großen Werten, die verstreut in dicht beieinanderliegenden Zellen zu finden sind. Zur Auswertung

empfiehlt sich dann eine Mittelwertbildung solcher Maxima-Häufungen. Für die Aufteilung des Parameterraumes lässt sich noch sagen, dass ein gröberes Raster nicht unbedingt schlecht sein muss. Da davon ausgegangen werden muss, dass eine Gerade mehrere benachbarte Zellen füllt, kann ein durch eine gröbere Einteilung enstehendes Zusammenfallen von Akkumulatorzellen durchaus erwünscht sein.

Der vorgestellte Algorithmus ist auf Grund der Geradenbüschelberechnungen recht aufwendig. Zur Optimierung besteht jedoch die Möglichkeit, anstelle der einfachen Kantendetektion mit Differenzoperatoren eine Kantendetektion, die als Ergebnis einen Gradienten liefert, zu nutzen. Dieser Ansatz wird auch in „AdOculos“ verfolgt und wird daher im dazugehörigen Buch [Bäs98] vorgestellt. Das Grundprinzip dabei ist Folgendes:
Wie wir wissen (siehe Abschnitt 8.6.5 auf Seite 38), beschreibt die Richtung des Gradienten die Richtung der stärksten Grauwertänderung. Der Gradientenvektor steht somit senkrecht zur Kante und legt dadurch die Richtung der Kante (Gerade) fest. Daraus folgt, dass ein Berechnen des Geradenbüschels in diesem Punkt unnötig ist, da die Richtung der einzig möglichen Gerade bereits feststeht. In der Hough-Transformation kann dann der, bei der Gradientenoperation erhaltene, Winkel für die Berechnung des Abstandes der Gerade zum Ursprung herangezogen werden. Es muss also für jeden Kantenpunkt nur noch eine Funktionsgleichung gelöst werden, gegenüber der je nach Aufteilung des Parameterraumes mehr oder weniger großen Menge an Gleichungen, die bei der oben aufgeführten Variante nötig sind.

8.10.1 Tracking

Eine Einschränkung der Hough-Transformation ist, dass durch sie zwar die im Bild vorhandenen Linien bzw. Objektkonturen erkannt werden, nicht jedoch ihre Start- und Endpunkte. Sind diese allerdings wichtig, dann muss noch ein Verfahren gesucht werden, dass sie erkennt. In [Bäs98] wird dazu ein sogenanntes *Tracking* benutzt. Beim Tracking wird ein *Glider* eingesetzt, der auf den errechneten Geraden über das Ursprungsbild läuft und nach Bereichen mit signifikanten Grauwertdifferenzen sucht. Hierzu vergleicht er die Pixel rechts und links entlang der Geraden. Des Wei-

teren hat der Glider eine vorher festzulegende Länge, die die Toleranz gegenüber Unterbrechungen bestimmt. Wenn der Glider in seinem Bereich nun entsprechende Grauwertdifferenzen registriert, dann befindet er sich mit hoher Wahrscheinlichkeit auf einer Objektkontur. Die Ein- und Austrittspunkte solcher Bereiche bestimmen dann die Anfangs- und Endpunkte von Segmenten (Geraden). Das Tracking hat allerdings den entscheidenden Nachteil, dass es sehr rechenintensiv ist und nur bei relativ wenigen Bildobjekten eingesetzt werden sollte. Des Weiteren entsteht noch ein Problem auf Grund der Ungenauigkeiten bei der Geradenextraktion:
Wenn der Winkel der extrahierten Gerade nur ein Grad von der wirklich im Bild vorhandenen abweicht, dann hat dies den Effekt, dass nach einigen hundert Bildpunkten der Abstand zwischen realer und extrahierter Gerade schon so groß ist, dass das Tracking diesen nicht mehr kompensieren kann, und somit den Endpunkt der Geraden nicht richtig berechnet.

9 Robotersteuerung

9.1 Einleitung

Nachdem im vergangenen Abschnitt näher auf die Grundlagen der digitalen Bildverarbeitung eingegangen wurde, soll auf den folgenden Seiten nun ein Überblick über zwei Themen der Robotersteuerung, die bei der Erstellung dieser Arbeit eine Rolle spielen werden, gegeben werden. Zum einen ist dies das Thema der Steuerungsarchitekturen und zum anderen das der Navigation mobiler Roboter.

9.2 Steuerungsarchitekturen

9.2.1 Überblick

Die unterschiedlichen existierenden Steuerungsarchitekturen, stellen zunächst verschiedene Konzepte dar, die die Organisation und Interaktion der zur Robotersteuerung eingesetzten Softwarekomponenten beschreiben. Neben der reinen Struktur geht es dabei gerade im Bereich der KI zusätzlich darum, dass mit einer solchen Architektur auch ein *intelligentes* System dargestellt werden kann.
In der Vergangenheit wurden daher schon mehrere solcher Konzepte vorgestellt, die sich mehr oder weniger durchgesetzt haben. Eine ultimative und für alle Anwendungsgebiete geeignete Lösung existiert allerdings noch nicht. Je nach Einsatzort des Roboters sind also bestimmte Ansätze besser geeignet als andere. Es gilt daher, zunächst die richtige Architektur auszuwählen oder sogar ein eigenes Konzept zu entwerfen.
Trotz der Menge an unterschiedlichen Architekturen kann man aber im Groben zwei verschiedene Richtungen unterscheiden:
den klassischen (deliberativen) und den reaktiven Ansatz.

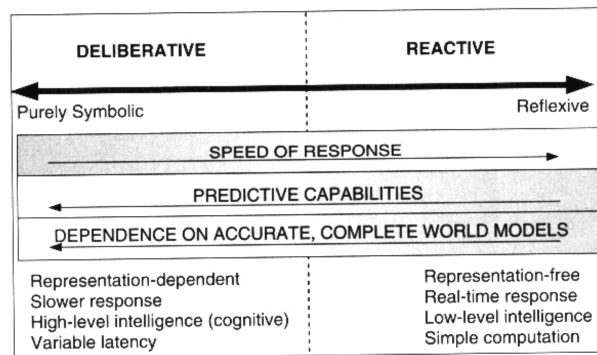

Abbildung 12: Spektrum der Steuerungsarchitekturen

In Abbildung 12, die dem Buch *„Behavior-Based Robotics“* [Ark99] entnommen wurde, sind die wichtigsten Eigenschaften dieser Steuerungsarchitekturen dargestellt. Wie zu sehen ist, gibt es zwar die beiden genannten Extrema, allerdings müssen die verschiedenen Konzepte nicht unbedingt **entweder** der einen **oder** der anderen Gruppe angehören, sondern können durchaus Eigenschaften beider beinhalten.
Im Folgenden werden nun die wichtigsten Merkmale der zwei wesentlichen Ansätze vorgestellt.

9.2.2 Der klassische Ansatz

Der klassische oder auch deliberative Ansatz basiert auf den klassischen Mitteln der Wissensverarbeitung. Grundlage dafür bildet daher ein Weltmodell, also eine symbolische Darstellung der Umgebung. Auf Grund dieses Weltmodells kann der Roboter dann seine Aktionen planen und koordinieren. Der grobe Steuerungsablauf wird dann also durch eine Sequenz folgender Aktionen dargestellt:

1. Sensordaten sammeln und aufbereiten
2. Weltmodell bilden
3. Planen
4. Aufgaben ausführen
5. Aktoren ansteuern

Die Struktur in diesem Ansatz ist auch oft charakterisiert durch eine Menge von hierarchisch (in Schichten) angeordneten Modulen, die jeweils eine bestimmte Funktionalität besitzen. In den dabei entstehenden Schichten kommunizieren die Funktionsblöcke dann in einer zuvor festgelegten Art und Weise. Dabei steuern die Komponenten in höher angesiedelten Schichten die Module, die in der Hierarchie tiefer liegen. Weiteres Merkmale bei dieser Anordnung ist, dass sich die Planungstiefe von unten nach oben vergrößert. In den tieferen Schichten wird dann beispielsweise zeitlich kürzer und örtlich in einem geringeren Umfeld geplant.
Damit ein solcher Ansatz richtig funktionieren kann, wird ein nahezu vollständiges Weltbild benötigt. Des Weiteren sollte es natürlich konsistent, zuverlässig und auch sicher sein. Da der gesamte Ansatz auf diesem Modell aufbaut, führen Fehler in ihm unweigerlich auch zu Fehlern in der Steuerung und damit zu unerwünschten Ergebnissen. Sehr gut geeignet ist dieses Konzept daher für Umgebungen, die überschaubar sind und keinen großen Änderungen unterworfen sind, wie z.B. in Produktionsstätten. Ein Vorteil, den dieser Ansatz dann bietet, ist, dass das Planungssystem einen optimalen Plan zur Erfüllung der Aufgabe erstellen kann und somit unnötige Aktionen wegfallen. Soll aber eine Steuerung für mobile Roboter, die in der realen Welt zum Einsatz kommen sollen, entwickelt werden, dann genügt es nicht mehr, nur von einem zu Beginn vorhandenen Weltmodell auszugehen. Vielmehr muss dieses dann während der Fahrt mit den Daten, die durch Sensoren wahrgenommen werden, abgestimmt und modifiziert werden. Ein Ansatz dazu ist das *reaktive Planen*, das aber nicht mit einer rein reaktiven Architektur verwechselt werden darf.

Pläne und Planungssysteme Wie gesagt wurde, basiert der gesamte klassische Ansatz auf Plänen. Nachfolgend wird daher etwas näher auf die Grundlagen von Plänen und Planungssystemen eingegangen.

Damit ein Planungssystem einen Plan erstellen kann, benötigt es zunächst die folgenden Grundvorraussetzungen:

- einen Anfangszustand,
- einen Zielzustand, und
- eine Liste aller mögliche Aktionen.

Aufgabe des Systemes ist dann, eine Abfolge von Aktionen zu finden, die, vom Ausgangszustand ausgeführt, einen Zustand erreichen, in dem das Ziel gilt. Das Ergebnis dabei ist dann ein *linearer Plan*. Pläne stellen somit eine Sequenz von Aktionen dar. Dieser Vorgang der Planerstellung wird auch als klassisches Planen bezeichnet. Neben der Einschränkung, dass nur ein Ziel erreicht werden kann, wird zusätzlich noch davon ausgegangen, dass die Ergebnisse der vorangegangenen Aktionen sicher sind. In einer dynamischen Umwelt, muss dies jedoch nicht immer der Fall sein. Es existieren daher Ansätze (z.B. reaktives Planen), die die Zwischenergebnisse des Plans überwachen und ihn gegebenenfalls abändern. Für die Robotersteuerung umfaßt die Planerstellung (Handlungsplanung) daher oft die automatische Generierung, Fehlerbehebung und Optimierung von Plänen.

Um die Arbeit von Planungssystemen zu veranschaulichen, greift man in der KI häufig auf die sogenannte *Klötzchenwelt* zurück. Die Klötzchenwelt kann folgendermaßen definiert werden:

1. Auf einem Tisch stehen Würfel neben- und übereinander; es ist genug Platz um alle Blöcke auf den Tisch zu stellen

2. Es gibt eine Greifhand, die genau einen Würfel zu einer Zeit aufheben kann, falls kein anderer über diesem steht

3. Ein Würfel steht entweder auf dem Tisch oder auf genau einem anderen Würfel oder wird von der Hand gehalten

4. Mit der Greifhand können folgende Operationen ausgeführt werden: einen Würfel von einem Würfel auf einen anderen bewegen (Move), einen Würfel von einem anderen nehmen und auf dem Tisch ablegen (Unstack), einen Würfel vom Tisch aufnehmen und auf einen anderen Würfel ablegen (Stack)

In der Literatur gibt es gerade bei den möglichen Aktionen unterschiedliche Ansätze. Hier wurden die Operationen aus [Hei99] verwendet. Abbildung 13 auf der nächsten Seite zeigt zwei mögliche Situationen in der Klötzchenwelt.

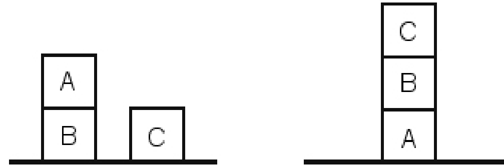

Abbildung 13: Die Klötzchenwelt

Die Grundvorraussetzungen für Planungssysteme sind die verschiedenen Situationen bzw. Zustände in der gegebenen Welt, die zunächst beschrieben werden müssen. Der erste Ansatz zur Situationsmodellierung in der Forschung zur Künstlichen Intelligenz ist der *Situationskalkül*. Dieser wurde von McCarthy und Hayes auf Basis der Prädikatenlogik entwickelt und stellt den Ausgangspunkt vieler heutiger Planungssysteme dar. Beim Situationskalkül gibt es spezielle eingeschränkte Variablen- und Konstantensymbole:

- Individuenvariablen $x, y, \ldots$ und -konstanten $A, B, \ldots$ und
- Situationsvariablen $s_1, s_2, \ldots$ und -konstanten $S_1, S_2, \ldots$.

Eine Situation ist dann ein Schnappschuß der repräsentierten Welt zu einem bestimmten Zeitpunkt. Zur Beschreibung der Eigenschaften und Relationen von Situationen wird die Prädikatenlogik erster Stufe mit den oben aufgeführten Symbolen verwendet. Für Konfigurationsangaben in der Klötzchenwelt benutzt man die folgenden Prädikate:

- On(x, y): Klötzchen x liegt direkt auf Klötzchen y.
- Clear(x): Auf Klötzchen x liegt kein anderes Klötzchen.
- Table(x): Klötzchen x liegt auf dem Tisch.

Eine einfache Möglichkeit zur Beschreibung einer Situation ist dann eine Angabe der Eigenschaften, die in ihr gelten. Die Beziehung zwischen Eigenschaften und

Situationen in der Klötzchenwelt wird durch eine Relation *holds* hergestellt. Formal ist diese definiert durch:

$$\text{holds}(d, s)$$

Dabei ist d der Situationsdeskriptor, welcher eine tatsächlich geltende Eigenschaft in der Situation s darstellt.
Für den linken Zustand in Abbildung 13 auf der vorherigen Seite gilt z.B. Folgendes:

$$\begin{array}{ll} \text{holds}(\text{Table}(B), S_1) & \text{holds}(\text{Clear}(A), S_1) \\ \text{holds}(\text{Table}(C), S_1) & \text{holds}(\text{Clear}(C), S_1) \\ \text{holds}(\text{On}(A, B), S_1) & \end{array}$$

Bei einer Beschreibung von Situationen nach diesem Schema muss allerdings darauf geachtet werden, dass diese jeweils zu einem konsistenten Modell führen sollte. Die folgende Situationsbeschreibung ist z.B. inkonsistent, da ein Klötzchen nicht gleichzeitig auf dem Tisch und auf einem anderen Klötzchen stehen kann:

$$\text{holds}(\text{Table}(A), S_1) \wedge \text{holds}(\text{On}(A, B), S_1)$$

Um solche Fälle zu vermeiden, muss daher noch eine Menge von Beschränkungen bzw. *allgemeinen Gesetzen* (engl. *state constraints*) angegeben werden, die vom System automatisch überprüft werden sollten.

Damit ein Planungssystem einen Plan generieren kann, benötigt es noch eine Liste aller möglichen Aktionen. In der Klötzchenwelt können z.B. die folgenden drei Operationen definiert werden:

- Move(x, y, z): Bewege x von y auf z
- Unstack(x, y): Nimm x von y und lege ihn auf den Tisch
- Stack(x, y): Hebe x vom Tisch auf und lege ihn auf y

Eine neue Situation s' ensteht dann aus einer gegebenen Situation s dadurch, dass in s eine Aktion a ausgeführt wird. Dieser Sachverhalt wird durch die Operation *apply* dargestellt:

$$s' = \text{apply}(a, s)$$

Da für die Ausführbarkeit von Aktionen in der Regel bestimmte Vorbedingungen erfüllt sein müssen, müssen diese noch zusätzlich angegeben werden.

Mit dem genannten Wissen ausgestattet, kann ein Planungssystem versuchen, eine entsprechende Aktionsfolge, die vom Ausgangs- zum Zielzustand führt, zu erarbeiten. Hierbei sind aber noch die folgenden Probleme zu beachten:

- Frameproblem:
  Welche Eigenschaften eine Situation ändern sich nach Ausführung einer Aktion *nicht*.

- Qualifikationsproblem:
  Typischerweise sind bei der Spezifikation von Operatoren nicht alle denkbaren Vorbedingungen berücksichtigt. Eine Move-Aktion in der Klötzchenwelt kann z.B. fehlschlagen, da ein Klötzchen zu schwer oder auf dem Tisch festgeklebt ist. Dies allerdings explizit in den Vorbedingungen der Aktion aufzunehmen ist zu unökonomisch.

- Ramifikationsproblem:
  Es ist nicht nur schwer zu spezifizieren, welche Aspekte einer Situation sich nach Ausführung einer Aktion *nicht* ändern, sondern auch schwer zu umreißen, was sich als Folge der Aktion alles ändert. Wird z.B. Klötzchen A von Klötzchen B auf C bewegt, dann muss zusätzlich zur neuen Position von A noch angegeben werden, dass sich A nicht mehr an der initialen Position befindet sowie dass C nicht mehr und B wieder frei ist.

Ein bekanntes Planungssystem, das auf dem Situationskalkül basiert, ist der STRIPS-Planer. Eine Aktion a ist dort durch drei Bestandteile definiert:

- **Pre**(a): eine Liste der Vorbedingungen für die Aktion a

- **Del**(a): eine Liste der Fakten, die nach der Ausführung von a nicht mehr gültig sind, und daher aus der Situationsbeschreibung zu entfernen sind

- **Add**(a): eine Liste der Fakten, die nach Ausführung von a zusätzlich gültig sind und daher der Situationsbeschreibung hinzugefügt werden müssen

Das Frame- und das Ramifikationsproblem wird dabei durch die Add- und Delete-Listen gelöst, da hiermit alle neuen sowie die nicht mehr geltenden Fakten explizit aufgeführt werden.
Einige weitere Planungssysteme sind unter [Koe00] aufgeführt.

9.2.3 Der reaktive Ansatz

Die zweite wichtige Steuerungsarchitektur basiert auf einem reaktiven Ansatz. Im Gegensatz zum klassischen Ansatz wird hierbei nicht mehr mit einem symbolischen Weltmodell gearbeitet, sondern mit der realen Welt selbst, also mit den Informationen, wie sie über die Sensoren wahrgenommen werden. Bei der Entwicklung dieser Architektur hat man sich am beobachteten Verhalten einfacher Lebewesen, wie z.B. Insekten, die vermutlich auch kein Weltmodell haben, orientiert und versucht, dieses nachzubilden.
Entstanden ist daraus eine Architektur, in der von einer Menge verschiedener, voneinander unabhängiger, parallel laufender Verhalten ausgegangen wird. Jedes dieser Verhalten, ein sog. *Behavior*, überwacht dabei bestimmte Sensordaten und kann direkt die entsprechenden Aktoren ansteuern. Das Ergebnis ist dann eine Verarbeitung nach folgendem Schema:

$$\text{Reiz} \longrightarrow \text{Behavior} \longrightarrow \text{Antwort}$$

Es fehlen somit die zeitintensiven und fehleranfälligen Schritte der Modellbildung und Planung. Ziel dieser Architektur ist es, eine Menge von kompakten Behaviors zu erzeugen, die jeweils nur eine kleine Aufgabe erfüllen und daher sehr schnell auf die Umwelt reagieren können. Sie kommt daher besonders für Robotersteuerungen in dynamischen Umgebungen zum Einsatz. Für einen Roboter wären mögliche Behaviors, z.B. *Kollisionen vermeiden* oder *eine konstante Geschwindigkeit halten*.
Da jedes Behavior von den anderen unabhängig agiert, ist es sehr leicht, ein solches System zu erweitern. Des Weiteren wird auch davon ausgegangen, dass um ein komplexes Roboterverhalten zu erzeugen, nicht unbedingt auch eine komplexe Steuerung notwendig sind. Vielmehr sollte es möglich sein, einfach zum bestehenden System zusätzliche kompakte Behaviors hinzuzufügen, die dann das komplexere Verhalten bewirken. Doch gerade hier liegt der Knackpunkt. Es wurden bis jetzt

zwar erfolgreich verhaltensbasierte Steuerungen entwickelt, die der Kompetenz von Insekten nahekommen, allerdings ist zur Zeit unklar, ob dieser Ansatz so skaliert werden kann, dass ein der menschlichen Intelligenz nahekommendes Verhalten erreicht wird. Er ist daher nur für einfache Anwendungen in dynamischen Umgebungen besonders gut geeignet, da mit zunehmender Komplexität die Gefahr besteht, dass das System nicht mehr überschaubar und darstellbar wird.

Neben der Erweiterbarkeit hat die Modularität in diesem Konzept aber noch einen weiteren Vorteil. Während beim klassischen Ansatz der Ausfall einer Softwarekomponente oder eines Sensors gleich das gesamte System lahmlegen würde, betrifft dies hier nur bestimmte Module. Da alle Module voneinander unabhängig sein sollen, können die nicht betroffenen ihre Arbeit ungestört fortsetzen. Damit besteht die Möglichkeit, dass das System noch sinnvoll weiterarbeiten kann.

Oft hört man auch, dass ein System, das mit einer reaktiven Steuerungskomponente ausgestattet ist, als *intelligent* erscheint. Wie Brooks in [Bro91] schrieb:

> *Intelligence is in the eye of the observer.*
> *Intelligence is determined by the dynamics of*
> *interaction with the world.*

ergibt sich diese Intelligenz aber nur aus der Interaktion des Roboters mit seiner Umgebung und ist auch vom Betrachter abhängig.
Ein weitere Frage dazu ist, ob ein System mit der Kompetenz von Insekten überhaupt schon als intelligent bezeichnet werden kann.

Die Subsumptions-Architektur Eine spezielle Art der reaktiven Steuerung stellte Rodney A. Brooks mit seiner *Subsumptions-Architektur* vor (siehe [Bro85]). Seine Idee dabei ist, dass ein komplexes Verhalten in verschiedene funktionale Blöcke zerlegt wird, welche dann in horizontalen Schichten angeordnet werden. Hierbei stellen die unteren Schichten einfache und grundsätzlich notwendige und die oberen Schichten komplexere Verhalten dar, wobei es im Prinzip jeder Schicht erlaubt ist, die Sensoren und Aktoren anzusteuern. Die höheren Ebenen können dabei die Ergebnisse der unteren Schichten nutzen. Es ist auch möglich, dass sie die unteren Schich-

ten kontrollieren, indem sie ihre Outputs hemmen oder ersetzen. Jede Schicht stellt aber ein eigenes Modul dar. Die unterschiedlichen Module arbeiten bei diesem Ansatz asynchron, können aber untereinander kommunizieren. Zu beachten ist hierbei aber, dass diese Module dann nicht mehr vollständig unabhängig voneinander sind. In Abbildung 14 ist ein Beispiel einer Steuerung nach der Subsumptions-Architektur zu sehen.

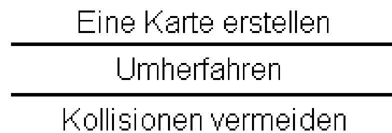

Abbildung 14: Ein Beispiel zur Subsumptions-Architektur

Soll nun eine Steuerung nach diesem Ansatz entwickelt werden, dann sollte nach Brooks folgendermaßen vorgegangen werden:
Zunächst werden die Module der untersten Schicht entwickelt und getestet. Ist dies erfolgreich, wird die nächste Schicht entwickelt, usw. Man geht also die Schichten von unten nach oben nacheinander durch. Nach Fertigstellung einer Schicht sollte es dabei nicht mehr notwendig sein, dort noch Änderungen durchführen zu müssen. Um also ein zusätzliches bzw. komplexeres Verhalten zu erreichen, reicht es somit im Prinzip aus, einfach noch weitere Schichten über die bestehenden aufzusetzen, ohne diese jedoch verändern zu müssen. Wie eingangs bei den reaktiven Steuerungen schon gesagt wurde, ist aber noch nicht klar, ob damit auch komplexe Verhalten dargestellt werden können. Jedes zusätzliche Verhalten sollte in seinem Aufbau aber trotzdem wieder so einfach wie möglich gehalten werden. Des Weiteren sollte darauf geachtet werden, dass das System robust gegenüber verrauschten oder ausgefallenen Sensoren sein sollte.
Auf Grund der parallelen Abarbeitung und der Tatsache, dass alle Module die Aktoren ansteuern können, ergibt sich das Problem, dass zu einem Zeitpunkt die unterschiedlichen Verhalten verschiedenene Aktionen ausführen möchten (z.B. ein Verhalten möchte den Roboter nach rechts lenken, während ein anderes ihn nach links steuern will). Um dieses Problem zu lösen, werden den einzelnen Schichten unterschiedliche Prioritäten zugeordnet.

9.2.4 Hybride Architekturen

In der Übersicht zu den Steuerungsarchitekturen wurde bereits erwähnt, dass es noch keine für alle Gebiete geeignete Architektur gibt. Der reaktive Ansatz hat zwar in dynamischen Umgebungen seine Vorteile, jedoch sind komplexere Steuerungen besser mit dem klassischen Ansatz realisierbar. Auf Grund der dabei notwendigen Modellbildung und Planung sind diese Steuerungen jedoch nicht so schnell und flexibel. Es scheint also, dass der Weg zu einer wirklichen künstlichen Intelligenz weder im einen noch im anderen Ansatz zu finden ist, sondern in einer Verknüpfung beider, einer sog. *hybriden* Architektur.
Hierbei tut man sich allerdings schwer, eine richtige Verbindung zwischen der symbolischen Darstellung bei der klassischen Architektur und der „realen“ Darstellung im reaktiven Ansatz zu finden. Dies stellt daher zur Zeit einen Forschungsschwerpunkt dar. Ein erster Schritt dazu ist beispielsweise das reaktive Planen.
Einige hybride Architekturen sind z.B. hierarchisch aufgebaut, wobei in einer höheren Schicht ein Planungssystem arbeitet, während in tieferen Schichten reaktive Komponenten zum Einsatz kommen.
Es sollte jedoch noch erwähnt werden, dass auch solch eine Architektur keine Garantie liefert, dass damit auch intelligente Systeme entwickelt werden können. Denn was nützt die beste Architektur, wenn das System zehnmal hintereinander denselben Fehler macht und immer wieder gegen die Wand fährt? Es muss daher so konzipiert werden, dass es auch aus seinen Fehlern *lernt*, was wiederum bedeutet, dass zusätzlich noch entsprechende Lernmodule integriert sein müssen.

9.3 Navigation mobiler Roboter

9.3.1 Der Begriff Navigation

Im Folgenden wird nun ein kurzer Überblick über das Thema *Navigation autonomer mobiler Roboter* gegeben und die mit dieser Aufgabe verbundenen Probleme vorgestellt. Zunächst soll dazu der Begriff näher beschrieben werden.

Eine Möglichkeit der Definition des Begriffes Navigation lautet:

Unter Navigation versteht man die Problemstellung,

- ein bewegliches Objekt
- ausgehend von einer momentanen Position
- auf Basis teilweise unvollständiger Information
- unter Berücksichtigung vorgegebener Randbedingungen
- zu einem vorgegebenen Ziel zu bringen.

Um diese Aufgaben zu lösen, wird die Navigation in zwei Phasen aufgeteilt. Die erste Phase ist die Wegplanung und die zweite Phase die eigentliche Fahrt. Es wird also zuerst der zu befahrende Weg bestimmt, und anschließend dieser Plan in die Tat umgesetzt. Der entworfene Plan wird dabei aber nicht unbedingt als fest angesehen, sondern kann während der Fahrt in Abhängigkeit der Umwelt noch verändert werden, um z.B. Kollisionen zu vermeiden. Da die Wegplanung während der Fahrt auf Grund der Sensorinformationen immer wieder überprüft werden muss, ergibt sich hierfür die Forderung nach Echtzeitfähigkeit. Beim Auftauchen eines Hindernisses muss das System beispielsweise schnell reagieren können, um ihm noch rechtzeitig ausweichen zu können.
Um die Qualität des Ergebnisses der Navigationsaufgabe zu bewerten, existieren die folgenden Optimalitätskriterien:

- Kollisionsfreier Weg
- Minimale Gesamtlänge
- Minimale Fahrzeit
- Minimale Gesamtrotation des autonomen mobilen Roboters
- Minimale Rechenzeit zur Wegbestimmung
- Maximale Sicherheit

9.3.2 Probleme bei der Navigation

Das Hauptproblem in der Navigation ist die Frage nach der augenblicklichen Position des autonomen mobilen Roboters. Üblicherweise kennt der Roboter zu Beginn seiner Fahrt seine Position. Während der Fahrt wird diese Position dann mit Hilfe

der eingebauten Sensoren ständig aktualisiert, jedoch ist dieser Vorgang nicht fehlerlos.
Im Pioneer-Roboter wird die Positionsaktualisierung z.B. mit einer Odometrie auf Basis von Shaftencodern durchgeführt. Die entsprechenden Sensoren sind an jedem Rad angebaut und messen dort den zurückgelegten Weg. Durch Kombination der Ergebnisse mehrerer Räder, z.B. an einer Achse, kann dann zusätzlich zum Weg auch noch die jeweilige Richtungsänderung bestimmt werden. Der Nachteil einer solchen Lösung ist, dass sie nur für kurze Wegstrecken und unter der Vorraussetzung, dass relativ wenige Drehbewegungen stattgefunden haben, eine einigermaßen genaue Positionsbestimmung ermöglicht. Durch Einflüsse wie Radschlupf und Driften, z.B. auf rutschigem Untergrund, schleichen sich im Laufe der Zeit immer größer werdende Fehler ein, die der Roboter nur unter Zuhilfenahme anderer Sensoren wieder berichtigen kann. Selbst wenn die Sensoren ab einem bestimmten Zeitpunkt einwandfrei arbeiten, gilt, dass sobald ein Fehler im System vorhanden ist, dieser nicht mehr berichtigt werden kann, sondern im Gegenteil sogar noch zunimmt. (In [Bor96] werden die zunehmenden Ungenauigkeiten durch, im Laufe der Zeit, größer werdende Fehlerellipsen dargestellt. Des Weiteren sind dort auch zusätzliche Informationen zu Odometriefehlern zu finden.)
Weitere mögliche Sensoren zur Selbstlokalisation sind Ultraschallsensoren, Laserscanner, Videokameras oder der Kreiselkompass. Außerdem kann auch noch mit natürlichen oder künstlichen Landmarken, z.B. Transpondern, oder aber einem satellitengestützten Positionssystem (GPS) gearbeitet werden.
Die optimale Lösung für einen mobilen Roboter stellt die Benutzung mehrerer Sensoren und die anschließende Kombination der Ergebnisse dar. Hierzu kann z.B. der *Kalman-Filter* verwendet werden. Dabei wird berücksichtigt, dass kein Sensor hundertprozentig genaue Ergebnisse liefert, sondern immer mit einer bestimmten Fehlertoleranz arbeitet. Durch Verknüpfung meherer Sensordaten erhält man dann ein Ergebnis, das genauer ist als die Einzelergebnisse.
Wie an den oben vorgestellten verschiedenen Sensortypen zu erkennen ist, ist es für einen Roboter wichtig, dass er in seiner Welt Orientierungsmöglichkeiten hat, durch die er seine Position berichtigen kann. Als Beispiel wurden schon Landmarken genannt. Oft orientieren sich Roboter mit Ultraschallsensoren oder Laserscannern an Wänden oder anderen größeren Gegenständen.

Ein anderes Problem bei der Navigation ist auch die Anforderung, dass ein autonomer mobiler Roboter in seiner Umgebung kollisionsfrei navigieren soll. Hierfür ist es notwendig, dass er schnell auf die Umwelt reagieren kann und seine Wegplanung ständig überprüft. Wenn also ein Hindernis im Weg steht, dann sollte der Roboter dieses umfahren. Er muss danach allerdings eine Strategie entwickeln, die ihn dann wieder auf den korrekten Weg in Richtung Zielpunkt führt. Dabei kann natürlich auch der Fall auftreten, dass der Weg komplett verstellt ist, und somit auch keine Möglichkeit mehr existiert, um das Ziel zu erreichen. Das System sollte auch solche Situationen erkennen und gegebenenfalls den Roboter stoppen.

Teil IV

Konzeption

10 Einleitung

Im vergangenen Abschnitt wurden die theoretischen Grundlagen von Bildverarbeitung und Steuerung etwas genauer erläutert. In diesem Teil der Arbeit wird nun die Konzeption der, zur Lösung der Aufgabe, benötigten Komponenten beschrieben. Die groben Anforderungen hierzu wurden eingangs bereits erwähnt. Auf Grund der Systemarchitektur gliedert sich der Teil „Konzeption“ wieder in die beiden Bereiche *Bildverarbeitung* und *Robotersteuerung*, wobei zunächst näher auf die Bildverarbeitung eingegangen wird.

11 Konzeption der Bildverarbeitungskomponente

11.1 Bildverarbeitung für mobile Robotersysteme

Ein wesentliches Merkmal von mobilen Robotern ist ja, dass sie sich in ihrer Umgebung bewegen. Für mit Kamera ausgestattete Roboter, wie unseren Pioneer 2, führt dies dazu, dass in der Bildverarbeitung ständig neue Bilder mit veränderten Objektpositionen zu betrachten sind. Die Folge davon kann sein, dass die Bildverarabeitungskomponente gerade ein Bild analysiert und die Daten an die Steuerungskomponente übergeben hat, inzwischen sich der Roboter aber schon soweit bewegt hat, dass die erarbeiteten Daten nicht mehr aktuell sind und eine in der neuen Situation falsche Reaktion bewirken. Die Bildverarbeitung steht also vor dem Problem, dass sie zum einen richtig funktionieren soll, also alle wichtigen Objekte erkennt, und zum anderen, dass sie auch in einer ausreichenden Geschwindigkeit arbeitet, sodass das Robotersteuerungsprogramm mit den von der Bildverarbeitungskomponente gelieferten Daten auch noch etwas anfangen kann. Denn was nützt es, wenn die Bildverarbeitung ein Hindernis in einem Meter Entfernung meldet, der Roboter

aber bereits dagegengefahren ist?
Es heißt also oftmals, Kompromisse zu schließen. Die Fahrgeschwindigkeit des Roboters sowie die Bewegungen der Kamera müssen zunächst auf die Bildverarbeitungsdauer abgestimmt werden. Um nun aber gerade bei komplexeren Bildanalysen den Roboter nicht im Schneckentempo navigieren zu lassen, muss die Bildverarbeitung noch auf Geschwindigkeit hin optimiert werden. Dies wird aber meist auf Kosten der Genauigkeit geschehen. Einfache Möglichkeiten dazu sind z.B. die Verwendung von Grauwert- anstatt von Farbbildern, eine Verringerung der Auflösung oder das Betrachten eines bestimmten Bildteils, in dem wichtige Objekte vermutet werden, anstelle des gesamten Bildes. Wenn man mit einer gewissen Ungenauigkeit leben kann, und diese auch bei den Steuerungsroutinen berücksichtigt, so kann man mit Sicherheit noch etwas Geschwindigkeit herausholen. Oft ist in der Robotersteuerung, die ja sowieso mit Ungenauigkeiten (z.B. bei der Navigation) arbeiten muss, auch nicht hundertprozentige Genauigkeit vonnöten. Wichtig ist vielmehr, dass das System schnell auf die sich ändernde Umgebung reagieren kann. Es reicht dann beispielsweise eine Aussage der Form: „Ungefähr 30cm vor mir befindet sich das gesuchte Objekt“. Ob dies in Wirklichkeit 25cm oder 30.1cm sind, ist dagegen irrelevant. Wenn in einem Robotersystem eine entsprechende Hardware vorhanden ist, die eine schnelle Bildverarbeitung auch mit komplexen Algorithmen zulässt, dann sollte man diese natürlich auch voll ausnutzen. In unserem Fall des Pioneer-Roboters gilt dies jedoch nur bedingt, sodass hier also Kompromisse geschlossen werden müssen.

11.2 Einleitung

Im Folgenden werden nun die zur Lösung meiner Aufgabe notwendigen Bildverarbeitungsschritte vorgestellt. Wie im Teil 3 „Einleitung“ schon gesagt wurde, sollen mit Hilfe der Kamera das Spielzeug, die Ablagekiste und der Teppichuntergrund erkannt werden. Dieser Abschnitt, wird daher in die drei Gebiete Teppicherkennung, Spielzeugerkennung und Ablageerkennung unterteilt. Als Grundlage für die Bildverarbeitung dienen dabei die mit der Roboterkamera aufgenommenen (RGB)Farbbilder. Die Verarbeitung von Grauwertbildern wäre zwar schneller gewesen, da dort im Gegensatz zu den 3 Farbkanälen bei RGB-Bildern nur ein Kanal berücksichtigt werden muss, jedoch hat man bei Farbbildern eine größere Anzahl

an Möglichkeiten, z.B. wenn bestimmte Farben oder Objekte mit bestimmten Farbkombinationen extrahiert werden sollen.

Zunächst also zur Teppicherkennung.

11.3 Teppicherkennung

11.3.1 Vorraussetzungen

Bevor man nun mögliche Verfahren zur Teppicherkennung finden kann, muss zunächst definiert werden, was ein Teppich ist bzw. welche Merkmale er hat.
Grundsätzlich gilt, dass der Teppichuntergrund möglichst einfarbig sein sollte, damit sich das darauf befindliche Spielzeug deutlich von ihm abheben kann. Da das Spielzeug bunte Farben hat, wäre als Untergrundfarbe somit etwas Dunkles am geeignetsten. Weiterhin gibt es zwei Möglichkeiten den Teppich zu definieren. Die erste ist, dass es wirklich einen (einfarbigen) Teppich gibt. Dieser müsste sich einerseits farblich deutlich von der angrenzenden Umgebung absetzen und andererseits matt sein, sodass er keine störenden Lichtreflexionen erzeugen kann. Die Teppicherkennung kann dann über die Farbe realisiert werden. Zweite Möglichkeit ist, dass man den Rand des Teppichs besonders markiert, wobei aber weiterhin die Einschränkung bezüglich seiner Farbe gilt. Vorteil hierbei ist jedoch, dass die Applikation toleranter gegenüber Lichtreflexionen ist.
Ich habe mich bei meiner Arbeit für die zweite Möglichkeit entschieden. Den Teppich an sich stellt der Laborfußboden dar, und die Begrenzung wird mittels farbigem Klebeband realisiert. Neben der zu erwartenden Robustheit, hat dies noch den Vorteil, dass man die fertige Demoapplikation schnell an verschiedenen Orten aufbauen kann, vorrausgesetzt der Untergrund hat die oben beschriebenen Eigenschaften. Die Farbe der Teppichbegrenzung sollte gut zu erkennen sein, und sich auch vom Fußboden abheben. Auf dem blaugrauen Laborfußboden ist ein „knalliges“ Rot daher ideal. Des Weiteren wird definiert, dass die gesamte Teppichbegrenzung ein Rechteck bildet, und die einzelnen Kanten somit gerade Linien darstellen. In Abbildung 15 auf der nächsten Seite ist der sich daraus ergebene Einsatzort des Roboters im Labor dargestellt.

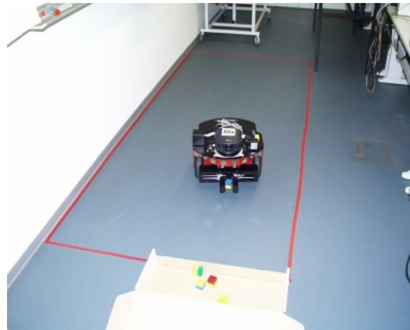

Abbildung 15: Der Einsatzort

11.3.2 Einleitung

Die Teppichbegrenzung bildet also ein rotes Rechteck. Es stellt sich nun die Frage, mit welchen Methoden der Bildverarbeitung man diese Begrenzung am Besten in einem vom Roboter gelieferten Bild erkennen kann. Abbildung 16 zeigt dazu zunächst

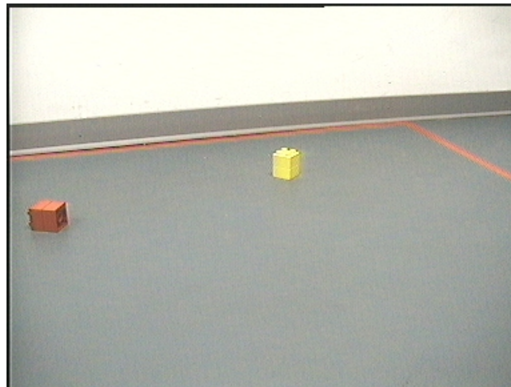

Abbildung 16: Das Sichtfeld des Roboters

eine Aufnahme mit der Roboterkamera, auf der die zu erkennende Teppichbegrenzung sichtbar ist.
Ziel der Bildverarbeitung soll sein, alle Bildpunkte, die auf der Teppichbegrenzung liegen, zu extrahieren und in irgendeiner Weise abzuspeichern, sodass sie später dem Robotersteuerungsprogramm zur weiteren Bearbeitung, z.B. für ein mögliches Verhalten *Fahre nicht über die Teppichgrenzen*, zugeführt werden können. Neben der Speicherung als Punkteliste, die allerdings viel Platz benötigt, kommt auf Grund der Eigenschaft, dass es sich um Pixel, die auf einer bzw. mehreren Geraden liegen,

noch die Speicherung in Form einer Geradengleichung in Frage. Da Geraden keinen definierten Start- und Endpunkt besitzen, kann es durchaus sinnvoll sein, diese noch zusätzlich abzulegen, vorrausgesetzt, sie können korrekt erkannt werden.
Die Aufgaben der Teppicherkennung in dieser Reihenfolge sind also erstens **Extraktion der Geradenpixel** und zweitens **Überführung in Geradengleichungen**.

11.3.3 Extraktion der Geradenpixel

Die erste Möglichkeit hierzu, die sich aus den vorgestellten Methoden der Bildverarbeitung ergibt, ist die Kantenextraktion (siehe Abschnitt 8.6 auf Seite 35), da die roten Teppichlinien eine deutliche Diskontinuität im Verlauf der Farbwerte des Bildes sind. Das Problem hierbei ist zunächst, dass wir mit Farbbildern arbeiten. Die Folge wäre, dass die ausgewählte Kantenextraktionsmethode auf die einzelnen Farbkanäle angewendet werden muss und die Ergebnisse zum Schluß noch in irgendeiner Art zu kombinieren sind. Dies wäre also recht aufwendig und das Ergebnis mit Sicherheit nicht zufriedenstellend, da ja alle möglichen Kanten sowie das Rauschen übrigbleiben würden. Allerdings wurde oben eine Einschränkung bezüglich der Farbe der Teppichbegrenzung gegeben. Diese soll ja die Farbe Rot haben. Daraus ergibt sich, dass das Bild zunächst nach dieser Farbe gefiltert werden kann, was man recht einfach mit Hilfe einer Differenzoperation (siehe Teil „Farbbasierte Segmentierung mit Differenzbildern“ in Abschnitt 8.8 auf Seite 42) realisieren kann. Nach der Differenzbildung empfiehlt sich, wie bei der Segmentierung schon angesprochen wurde, eine Binarisierung mit einem geeigneten Schwellwert, um nur noch die „richtig“ roten Bildpunkte übrigzubehalten. In Abbildung 17 auf der vorherigen Seite ist das Bild von Abbildung 16 auf Seite 67 nach einer Differenzoperation der Kanäle $ROT - GRÜN$ und anschließender Binarisierung zu sehen. Wie man erkennen kann, ist das Ergebnisbild sehr gut gelungen und erleichtert die Weiterverarbeitung enorm. Ein Problem (welches sich auch auf der Abbildung zeigt) dabei ist jedoch, dass nach der Rotfilterung nicht nur die Teppichgrenzen übrigbleiben, sondern natürlich auch sämtliche anderen roten Objekte. Diese Tatsache muss bei der weiteren Bearbeitung berücksichtigt werden. Des Weiteren kann es von Nachteil sein, dass die extrahierten Objekte in ihrer kompletten Fläche übriggeblieben sind. Speziell für die umfangreichen Berechnungen, die zum Finden der Geradengleichun-

Abbildung 17: Das Bild von Abbildung 16 auf der vorherigen Seite nach Ausführung einer Differenzoperation der Farbkanäle $ROT - GR\ddot{U}N$ und anschließender Binarisierung

gen nötig sind, wäre es besser, wenn man die Objekte in „verdünnter“ Form vorliegen hätte, also anstelle der mehrere Pixel dicken Teppichkanten nur eine ein Pixel breite Linie. Der Vorgang der Verdünnung von Objekten wird in der Bildverarbeitung auch als *Skelettierung* bezeichnet. Das Ergebnis einer solchen Operation ist eine ein Pixel breite Linie, die ungefähr in der Mitte des Objekts liegt und die die ursprüngliche Form dessen wiederspiegeln soll. Zur Skelettierung werden häufig *morphologische Operationen* eingesetzt. Ein wesentlicher Nachteil der Methode ist, dass dafür meistens mehrere Iterationsschritte hintereinander ausgeführt werden müssen bis sich das Skelett stabilisiert hat. Das hat zur Folge, dass natürlich auch mehr Rechenzeit benötigt wird. Als Alternative kann man sich nochmal den anfangs besprochenen kantenorientierten Ansatz vor Augen führen. Der Ausgangspunkt ist ja nun nicht mehr ein Farbbild, sondern ein Binärbild, in dem die unwichtigen Objekte bereits entfernt wurden. Eine Teppichkante stellt in diesem Bild in den meisten Fällen (je nach Entfernung, Rauschen) eine mehrere Pixel breite Linie dar. Wird jetzt auf dieses Bild ein Kantenoperator angewendet, dann bleiben von dieser Linie nur noch die seitlichen Begrenzungen übrig. Das Ergebnis einer solchen Operation ist in Abbildung 18 auf der nächsten Seite zu sehen. Dieses Bild wurde mit „AdOculos“ erzeugt, wobei zur Kantenextraktion der Laplace-Operator eingesetzt wurde. Da das zu bearbeitende Bild ein Binärbild ist und Störungen kaum mehr vorhanden sind, kann zur Kantenextraktion auch ein einfacher Differenzoperator, der jeweils die Differenz

des aktuellen Bildpunktes mit z.B. seinem linken und oberen Nachbarn berechnet, eingesetzt werden. Außerdem enthalten die entsprechenden Operatormasken keine Wichtungen der Bildpunkte, womit auch keine rechenintensiven Multiplikationen mehr nötig sind. Das Ergebnis ist nun also ein Bild, in dem von den ursprünglich

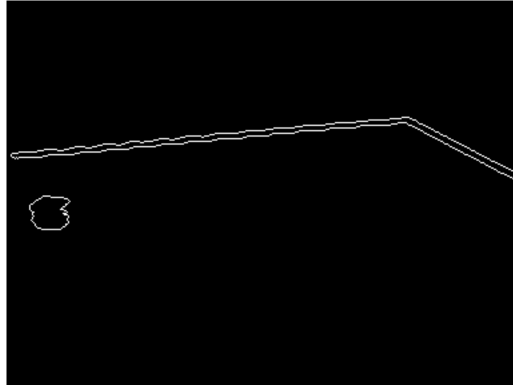

Abbildung 18: Das Bild von Abbildung 17 auf Seite 68 nach einer Kantenextraktion mit dem Laplace-Operator und anschließender Binarisierung

mehrere Bildpunkte breiten Teppichkanten nur noch ein Pixel breite Linien übriggeblieben sind, wobei aber zu beachten ist, dass dies in der Regel zwei parallele Linien zu einer Teppichkante sind.

11.3.4 Überführung in Geradengleichungen

Als Nächstes gilt es nun, die extrahierten Bildpunkte in eine Geradenform zu bringen. Hierbei muss darauf geachtet werden, dass zum einen nicht alle gefundenen roten Bildpunkte auch auf Teppichkanten liegen müssen und zum anderen, dass in einem Bild durchaus mehrere Linien gefunden werden können. Das zur Geradenextraktion eingesetzte Verfahren sollte also diese Rahmenbedingungen berücksichtigen. In dieser Arbeit soll daher die Hough-Transformation verwendet werden, deren Grundlagen bereits beschrieben wurden. Wie dort bereits gesagt wurde, ist dieses Verfahren sehr gut geeignet, um auch Linien zu erkennen, die unterbrochen oder verrauscht sind.
Das Ergebnis der Transformation ist ein gefüllter Hough-Akkumulator, der anschließend ausgewertet werden muss. Hierfür wurde bereits ein Verfahren, das auf einer Mittelwertsbildung basiert, empfohlen. Zusätzlich sollte bei der Auswertung ein

Schwellwert benutzt werden, der eine minimale Anzahl von Geraden-Bildpunkten angibt. Dies ist gerade bei der gestellten Aufgabe wichtig, da in den Bildern nicht nur rote Teppichbegrenzungen, sondern möglicherweise auch andere rote Objekte, wie z.B. Spielzeug, sichtbar sein können. Deren Begrenzungen könnten möglicherweise auch als Linien interpretiert werden, was natürlich zu Fehlern führen würde.
Nachdem die Geradengleichungen ermittelt wurden, kann noch ein Tracking durchgeführt werden, um die genauen Start- und Endpunkte der Linien zu finden. Es wurde jedoch schon festgestellt, dass dieses Verfahren sehr rechenintensiv und fehleranfällig ist. In dieser Arbeit soll daher darauf verzichtet werden.

11.3.5 Implementierungsmöglichkeiten

In den vergangenen Abschnitten wurden die grundlegenden Schritte zur Linienerkennung beschrieben. Eine Möglichkeit für eine hierfür konkret zu realisierende Algorithmenkette ist also Folgende:

1. Vorfilterung nach der Farbe ROT
2. Kantendetektion mit einfachem Differenzoperator
3. Hough-Transformation mit Berechnung der Geradenbüschel
4. Analyse des Parameterraumes mit Mittelwertbildung

Anstelle einer Skelettierung wurde hier ein einfacher Kantenfilter verwendet, da die Skelettierung möglicherweise mehr Rechenzeit verbraucht, als zum Schluß durch sie gewonnen wird. Es muss dann allerdings berücksichtigt werden, dass die Geradenerkennung zwei dicht beieinanderliegende parallele Geraden liefern kann.
Wegen den, im letzten Abschnitt, genannten Gründen wird auch auf ein Tracking verzichtet. Es ist auch noch nicht klar, ob die Robotersteuerung unbedingt auf die Anfangs- und Endpunkte der Geraden angewiesen ist.

Ein anderer möglicher Ansatz für die Algorithmenkette ist die Verwendung eines gradientenbasierten Kantenoperators und anschließender optimierter Hough-Transformation (siehe Abschnitt 8.10 auf Seite 45). Für die Kantendetektion könnte dann z.B. der Sobel-Operator eingesetzt werden. Das Problem dabei ist jedoch,

dass ein solcher Operator bei Binärbildern schlecht funktioniert und somit das rotgefilterte Bild nicht mehr eingesetzt werden kann. Des Weiteren ist es, wenn man genaue Ergebnisse haben will, erforderlich einen größeren Gradientenoperator (z.B. 5x5) zu benutzen, was wiederum zu erhöhtem Rechenaufwand führt.

Beide Ansätze haben also ihre Vor- und Nachteile. Es scheint aber, dass die erste Lösung die bessere ist, zumal die notwendigen Berechnungen noch verringert werden können, indem der Parameterraum entsprechend gröber gerastert wird. Daher soll dieser Ansatz implementiert werden.

Als Grundlage für die Implementierung sämtlicher Bildverarbeitungsmethoden soll der vorhandene Bildverarbeitungsserver genutzt werden, welcher über einen gemeinsamen Speicherbereich mit dem Robotersteuerungsprogramm verbunden ist. Alle Bildverarbeitungsfunktionen sind in diesem Programm in einem Thread zusammengefaßt. Das heißt, dass auch die von mir gewählte Algorithmenkette an dieser Stelle implementiert werden soll. Einzelheiten zum Bildverarbeitungsserver und der Implementierung meiner Funktionalitäten folgen dann später in dem entsprechenden Abschnitt. Hier soll zunächst nocheinmal aufgeführt werden, welche Methoden benötigt werden:

1. Eine Methode, die eine Filterung nach der Farbe ROT ermöglicht
   Eingabe: RGB-Farbbild → Ausgabe: Binärbild
2. Eine Kantendetektion mit einfachem Differenzoperator
   Eingabe: Binärbild → Ausgabe: Binärbild
3. Eine Methode zur Hough-Transformation mit Berechnung der Geradenbüschel
   Eingabe: RGB-Farbbild → Ergebnis: gefüllter Hough-Akkumulator
4. Eine Methode, die den Hough-Akkumulator über eine Mittelwertbildung analysiert

Dabei gibt es folgende Punkte zu beachten:

- Die jeweiligen Methoden sollen parametrisierbar sein, d.h. insbesondere die Schwellwerte für die Binarisierungen sollen variabel sein, sodass sie auch bei laufendem Programm verändert werden können.

- Da in dieser Lösung keine Skelettierung durchgeführt wird, muss damit gerechnet werden, dass zwei dicht beieinanderliegende Geraden extrahiert werden können. Die Akkumulatoranalyse muss dies berücksichtigen.
- Bei der Akkumulatoranalyse muss beachtet werden, dass die Punkte an den Rändern des Akkumulators Nachbarn sind, da die Winkel zyklisch sind.

11.4 Spielzeugerkennung

11.4.1 Vorraussetzungen

Zunächst heißt es wieder, die Eigenschaften und Merkmale der gesuchten Objekte zu bestimmen. In der Aufgabenstellung wurden die Objekte nur als „farbiges Spielzeug“ beschrieben. Es wurden also keine Einschränkungen bezüglich Farbe und Form gegeben. Trotzdem müssen einige Eigenschaften festgelegt werden.
Die erste ist, dass das Spielzeug **nicht** die Farbe des Untergrundes haben darf, es muss sich also deutlich sichtbar davon abheben. Des Weiteren muss der Roboter in der Lage sein, das Spielzeug aufzunehmen. Es darf somit nicht größer als der Greiferzwischenraum sein, allerdings auch nicht zu klein. Ideal wäre ein Würfel mit einer Kantenlänge von ca. 6 cm. Des Weiteren sollten die Objekte so auf dem Boden verteilt sein, dass es für den Roboter möglich ist, unterschiedliche Objekte auch als getrennt anzusehen, sie sollten also nicht zu dicht beeinanderliegen. Eine weitere Einschränkung ergibt sich aus der Teppichranderkennung. Da dort nach roten Objekten gefiltert wird und anschließend mit sämtlichen extrahierten Punkten eine Hough-Transformation durchgeführt wird, kann es zu Fehlinterpretationen kommen, wenn zu viele rote Objekte auf dem Boden liegen. Bilden diese dann aus der Sicht des Roboters auch noch eine Linie, dann kann eine Teppichkante erkannt werden, wo eigentlich gar keine sein sollte.
Als Spielzeug werden daher zunächst handelsübliche LEGO-DUPLO-Bausteine verwendet. Diese besitzen grelle, bunte Farben und sind in ihrer Größe variabel, d.h. es können mehrere davon zusammengesetzt werden. Letzteres ist auch notwendig, da ein einzelner Baustein etwas zu klein ist.

11.4.2 Ansätze und Implementierungsmöglichkeiten

Für die Spielzeugerkennung gibt es im Wesentlichen zwei Ansätze, einen farb- und einen kantenorientierten.

Der farborientierte Ansatz Das Grundprinzip dieser Methode ist eine Suche nach Objekten, deren Farbe nicht der des Hintergrundes entspricht. Vor Beginn der Erkennung müsste also festgelegt werden, welche Farbe der Teppich hat. Diese könnte dann, ggf. mit einer Toleranz, aus dem Bild herausgefiltert werden. Die verbleibenden Bildpunkte würden dann Objektbildpunkte darstellen und könnten zu einzelnen Objekten zusammengefaßt werden. Ein Problem dabei stellt jedoch der Einfluß von wechselnden Lichtverhältnissen dar, da dies bedeutet, dass die ursprünglich als Hintergrund definierte Farbe immer wieder angepaßt werden müsste. Des Weiteren sind auch Reflexionen auf dem Untergrund nicht ausgeschlossen. Für die Konzeption habe ich zunächst einige Probeaufnahmen im KI-Labor gemacht. Auf ihnen ist deutlich der Einfluß von Lichtreflexionen zu sehen. Diese führen dazu, dass gerade in größerer Entfernung weite helle Flächen erkennbar sind, die absolut nicht mehr der Teppichfarbe entsprechen. Mit dem oben aufgeführten Algorithmus würden diese Flächen nicht mehr als Hintergund sondern als Vordergrund erkannt werden, was natürlich zu Fehlern führt. Eine Implementierung dieses Algorithmus ist also nicht empfehlenswert.
Für die farbbasierte Objekterkennung kann man sich auch noch einen anderen Ansatz vorstellen. Das Spielzeug ist ja definiert, als „bunte“ Objekte, wobei als bunt jetzt knallige Farben wie rot, grün oder gelb angenommen werden. Wenn weiterhin davon ausgegangen wird, dass der Hintergrund unbunt, also eher grau ist, dann kann die Spielzeugerkennung folgendermaßen realisiert werden:
In einem RGB-Farbbild ist ein Grau-Ton dadurch definiert, dass in allen drei Farbkanälen etwa ähnliche Werte auftreten. Wenn das Eingabebild nun so gefiltert wird, dass alle Bildpunkte, deren RGB-Werte einen Grau-Ton darstellen, als Hintergrund definiert werden, dann bleibt nur noch buntes Spielzeug (und natürlich der Teppichrand) übrig. Die Filterung könnte dabei so ablaufen, dass man mit Differenzen der drei Farbkanalwerte des Bildpunktes arbeitet und mit einem Schwellwert die

„Farbigkeit" bestimmt. Es können z.B. die Differenzen der Kanäle $ROT - GRÜN$ und $ROT - BLAU$ gebildet und dann die Beträge der Ergebnisse mit einem Schwellwert verglichen werden. (Alternativ könnte man auch, wenn die Bilder im HSI-Farbmodell gespeichert sind, den dort vorhandenen Wert für die Farbsättigung zugrundelegen und darauf dann ein Schwellwertverfahren anwenden. In der vorliegenden Arbeit soll jedoch mit RGB-Bildern gearbeitet werden.) Ein Nachteil dieses Ansatzes ist jedoch, dass die Farben schwarz und weiß ebenfalls herausfallen würden, wobei gerade weißes Spielzeug durchaus üblich ist. Des Weiteren wird hier davon ausgegangen, dass die Hintergundfarbe immer einen Grau-Ton darstellt, was jedoch die Einsatzfähigkeit der fertigen Demoapplikation an verschiedenen Orten einschränkt.

Der kantenorientierte Ansatz Als Alternative kommt im Prinzip nur eine farbunabhängige, kantenbasierte Lösung der Spielzeugerkennung in Frage. Die einzigen Rahmenbedingungen dafür sind, dass der Untergrund einen homogenen Farbverlauf haben sollte und sich die Objekte deutlich davon abheben müssen. Eine Kantenerkennung kann dann so realisiert werden, dass ein Kantenoperator auf jeden Farbkanal angewendet und die Ergebnisse dann verknüpft (z.B.Addition) werden.
Allerdings ist auch dieses Verfahren nicht ohne Probleme. Bei der Kantendetektion gibt es immer die Schwierigkeit, dass durch Bildstörungen Kanten an Stellen erkannt werden können, wo in Wirklichkeit gar keine existieren. Des Weiteren wird bei dem Verfahren nicht zwangsläufig der komplette Umriß eines Objektes registriert, sodass dort Lücken auftreten können. Es muss daher nach der Kantenextraktion noch ein Verfahren zur Konturverknüpfung angewendet werden, dass diese Lücken wieder schließt. Hierfür können z.B. morphologische Bildoperatoren eingesetzt werden, aber auch ein einfacher Maximumoperator ist möglich. Es stellt sich dabei trotzdem die Frage, ob nach Durchführung eines solchen Bearbeitungsschrittes sämtliche Objektkonturen geschlossen sind, oder ob er vielleicht (speziell bei der morphologischen Closing-Operation) noch ein zweites oder drittes Mal angewendet werden sollte.
An dieser Stelle können dazu keine weiteren Einzelheiten gegeben werden, da sich diese erst später bei der Implementierung und den zugehörigen Tests herausstellen werden. Dort muss zunächst eine Kantendetektion und dann eine Methode zur Verknüpfung der extrahierten Konturen realisiert werden. Für die letztendliche Objektextraktion ist jedoch noch eine Methode nötig, die diese Konturen zu Objekten

zusammenfaßt. Als Möglichkeit dafür bietet sich die bereits im Bildverarbeitungsserver enthaltene Objekterkennung, die mit einer 4er-Nachbarschaft arbeitet, an. Vorraussetzung für das Funktionieren dieser Methode ist aber, dass die Objektkonturen vorher auch wirklich geschlossen wurden.

11.4.3 Gültigkeit von erkannten Objekten

Ein wesentlicher Punkt bei der Spielzeugsuche wurde bisher außer acht gelassen, und zwar die Gültigkeit der gefundenen Objekte.
Nachdem eine Objekterkennung erfolgt ist, stellt sich die Frage, ob das gefundene Spielzeug auch innerhalb der Teppichbegrenzungen liegt. Dieser Sachverhalt kann aber unter Zuhilfenahme der ermittelten Geradengleichungen festgestellt werden:
Wenn der Roboter sich auf dem Teppich befindet, dann ist ein Objekt ebenfalls auf dem Teppich, wenn es sich aus Sicht des Roboters vor allen sichtbaren Linien befindet. Mathematisch bedeutet dies: ein Objekt ist gültig, wenn sein Schwerpunkt unterhalb aller Geraden liegt. Dies kann sehr leicht unter Verwendung der Geradengleichungen überprüft werden.
Es muss dann von der Steuerung aber garantiert werden, dass die Kamera nie soweit über die Begrenzungen hinwegschaut, dass diese im Bild nicht mehr sichtbar sind. Es könnten sonst Objekte erkannt werden, die außerhalb des Einsatzbereiches liegen.

Als Alternative zum beschriebenen Vorgehen kann man sich auch vorstellen, die im Bild vorhandenen Geraden genauer zu untersuchen. Oben wurde nur mit den Geradengleichungen an sich gearbeitet. Zusätzlich könnte man auch noch versuchen, Start- und Endpunkte dieser Geraden im Bild zu finden. Damit kann dann eindeutiger bestimmt werden, wo sich eine Teppichecke befindet. Leider ist hierfür wiederum das sehr komplexe und fehleranfällige Linienverfolgen (Tracking (siehe Abschnitt 8.10.1 auf Seite 48)) notwendig. Ich habe mich daher dagegen entschieden.

11.5 Erkennung der Ablage

Die letzte Aufgabe der Bildverarbeitungskomponente soll das Erkennen der Ablage sein. Bezüglich der Form der Ablage gibt es eigentlich keine besonderen Ein-

schränkungen. Sie sollte natürlich so groß sein, dass auch etwas Spielzeug hineinpaßt und der Rand so niedrig, dass der Roboter ein gegriffenes Objekt in der Kiste ablegen kann. Ferner sollte sie so breit sein, dass eventuelle Ungenauigkeiten bei der Robotersteuerung nicht gleich dazu führen, dass sie weggeschoben wird. Des Weiteren muss sie direkt an den Teppich angrenzen. Für die Bildverarbeitung sind jedoch die optischen Besonderheiten wichtig. Die Ablage sollte daher eine spezielle Markierung haben, damit sie gut wahrgenommen und auch bei mehreren sichtbaren Objekten eindeutig bestimmt werden kann. Als Möglichkeiten ergeben sich entweder eine bestimmte Farbkombination oder ein Muster (z.B. drei Punkte).
Für Demonstrationszwecke bei der Computer-Messe CeBIT 2000 wurde an der Fachhochschule Brandenburg eine Methode zur Erkennung einer besonders markierten Flasche entwickelt, welche im Bildverarbeitungsserver des Pioneer 2 implementiert ist. Die Flasche hat dabei ein spezielles Farbmuster (siehe Abbildung 19). Dieses

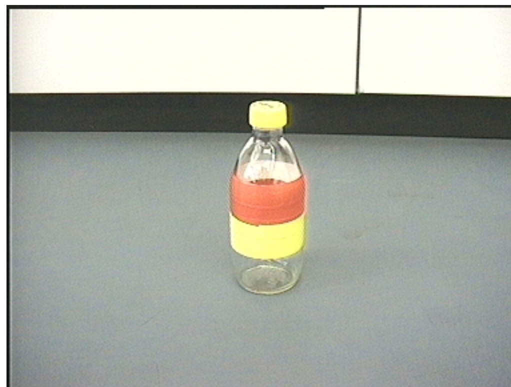

Abbildung 19: Die markierte Flasche

setzt sich aus einem gelben Deckel und einer rot-gelben Farbkombination am Bauch der Flasche zusammen. Der Ablauf der implementierten Bildverarbeitungsroutine ist Folgender:
Zuerst werden alle Bildpunkte klassifiziert in rote, gelbe und unwichtige Pixel. Dann wird ein Verfahren angewendet, dass die einzelnen Bildpunkte in Regionen zusammenhängender gleichartiger Pixel zusammenfaßt. Das Ergebnis sind dann größere Regionen mit einem bestimmten Farbmerkmal, also z.B. gelb. Anschließend wird in diesen Regionen die gewünschte Farbkombination gesucht. Die theoretische Grundlage für diese Suche stellen Bildanalysen mit hierarchischen Modellen (siehe dazu

[Fuc00]) dar.
Die gesuchte Farbkombination setzt sich zusammen aus:

- Einer gelben Region
- Einer roten Region, die um einen bestimmten Faktor größer ist als die gelbe Region und sich mit einem bestimmten Abstand zentriert unter ihr befindet, und
- Einer gelben Region, die direkt zentriert unter der roten Region liegt und ungefähr dieselbe Größe hat.

Zum Schluß wird dann aus der Verknüpfung dieser drei Regionen der Schwerpunkt berechnet und an die Robotersteuerung übergeben. Da es eher unwahrscheinlich ist, dass in einem Bild eine solche besondere Farbkombination mehrfach auftritt, arbeitet diese Methode recht zuverlässig. Es liegt daher nahe, dieses Verfahren mit einigen Modifikationen auch für die Erkennung der Ablage einzusetzen.
Zunächst muss also ein spezielles Farbmuster festgelegt werden.
Auf Grund der Tatsache, dass die Teppichbegrenzung bereits die Farbe rot besitzt, ist es nicht empfehlenswert, diese auch für die Farbkombination der Ablage einzusetzen. Eine mögliche Kombination sind drei farbige, gleichgroße, direkt übereinander liegende Streifen mit der Farbfolge grün-gelb-grün. Die Frage ist nun, wie diese an der Ablage angebracht werden können, damit der auf dem Teppich befindliche Roboter sie erkennen kann. Hierfür gibt es im Prinzip mehrere Möglichkeiten.
Die erste ist, dass die komplette Seite der Ablage, die dem Teppich zugewandt ist, entsprechend markiert wird, also durch drei lange Farbstreifen. Zweite Möglichkeit ist, nur den Ablagemittelpunkt an der Teppichseite zu markieren. Und die dritte ist das Verwenden mehrerer gleichartiger Farbmarkierungen, die sich z.B. an der rechten und linken Seite befinden. Als Zielpunkt für den Roboter, auf den er dann mit einem aufgenommenen Objekt zufahren kann, könnte dann immer der Schwerpunkt der jeweiligen Regionen dienen. Da eine Verwendung mehrerer Markierungen, aus denen dann ein Schwerpunkt berechnet werden soll, bei der Erkennung eine Unsicherheit darstellt, soll dieser Ansatz hier nicht weiter verfolgt werden. Als Alternative bleibt dann die Markierung an einer Stelle bzw. an der kompletten Fläche, wobei ich mich

für die erstere Möglichkeit, also das Kennzeichnen der Ablagemitte, entschieden habe.
Wie schon erwähnt wurde, ist eine Funktionalität, die ein Farbmuster erkennen kann, bereits vorhanden. Diese muss dann für die neue Problemstellung nur noch angepaßt werden. Daher möchte ich an dieser Stelle auf den Teil „Implementierung“ verweisen, in dem die genaue Realisierung erläutert wird.

11.6 Zusammenfassung

An dieser Stelle soll eine kleine Zusammenfassung gegeben werden, die alle sich bei der Konzeption der Bilverarbeitungskomponente ergebenden Randbedingungen aufzeigt.
Zunächst jedoch noch etwas zur Implementierung der Funktionen.
Sämtliche erarbeiteten Bildverarbeitungsmethoden sollen in den bereits vorhandenen Bildverarbeitungsserver integriert werden. Die Ergebnisse der einzelnen Funktionen sind dabei entweder einzelne Objekte (Spielzeug, Ablage) oder aber Gleichungen von Geraden (Teppichkanten). Diese Ergebnisse müssen der Robotersteuerung zur Verfügung gestellt werden, wozu bereits der Ansatz über einen gemeinsamen Speicherbereich realisiert ist. Es stellt sich dann nur noch die Frage, in welcher Form diese Daten, speziell bei Spielzeug und Ablage, abgespeichert werden sollen. Im Allgemeinen ist zur Kennzeichnung eines Objektes sein Schwerpunkt ausreichend, jedoch können auch weitergehende Informationen (Minimum, Maximum, Ausdehnung) von Bedeutung sein. Die genaue Form wird dann bei der Implementierung vorgestellt, nachdem dann auch die steuerungsseitigen Anforderungen genannt wurden.

Im Folgenden sind die Eigenschaften der einzelnen Objekte nocheinmal zusammengestellt:

Eigenschaften des Untergrundes:

- da von buntem Spielzeug ausgegangen wird, sollte er einen eher dunklen Farbton haben
- homogener Farbverlauf

- sollte keine allzu großen Lichtreflexionen ermöglichen

Eigenschaften der Teppichränder:

- sind in roter Farbe
- bilden ein Rechteck

Eigenschaften des Spielzeugs:

- muss sich farblich vom Untergrund abheben
- muss für den Roboter greifbar sein (nicht zu klein, nicht zu groß)
- verschiedene Objekte sollten deutlich erkennbar auseinanderliegen
- es sollten nicht zu viele rote Objekte vorhanden sein, um die Teppicherkennung nicht zu beeinflussen

Eigenschaften der Ablage:

- muss so groß sein, dass auch einige Objekte abgelegt werden können
- kleinere Ungenauigkeiten bei der Robotersteuerung müssen bei der Breite berücksichtigt
- der Rand zum Teppich hin muss so niedrig sein, dass der Roboter ein Objekt darüberheben kann
- muss direkt an den Teppich grenzen
- ist an der Teppichseite in der Mitte mit einem grün-gelb-grünen Farbmuster markiert

12 Konzeption der Robotersteuerungskomponente

12.1 Einleitung

In diesem Teil der Arbeit möchte ich nun meine konzeptionellen Überlegungen für eine Realisierung der Robotersteuerung beschreiben.
Hierbei werden dann auch die konkreten, von der Bildverarbeitungskomponente zu liefernden, Informationen bestimmt.

12.2 Rahmenbedingungen

Bevor näher auf die Struktur der Steuerung eingegangen wird, müssen noch einige Rahmenbedingungen geklärt werden. Aus Sicht der Bildverarbeitung wurden im letzten Abschnitt bereits Einschränkungen genannt.
Hier soll jetzt genauer die Größe des Teppiches, der Standort und die Form der Kiste sowie der Standort des Roboters festgelegt werden.
Zunächst zur Größe des Teppiches.

Wie bei der Bildverarbeitung schon erwähnt, wird das Spielzeug in erster Linie durch relativ kleine Bausteine dargestellt. Dies wirft aber das Problem auf, dass gerade, wenn etwas mehr Spielzeug vorhanden ist, der Aufräumvorgang für eine Demo-Applikation zu lange dauert. Um die Menge an Objekten etwas einzuschränken, sollte der Teppich daher nicht zu groß gewählt werden. Zum Einsatz für diese Arbeit kommt somit ein Teppich mit einer Größe von ca. 1.3x3 m. Die Breite wurde hierbei indirekt vom Untergrund des zukünftigen Einsatzortes vorgegeben, da dort im Fußboden schon Linien vorhanden waren. Diese habe ich als Begrenzung genutzt, um eine spätere Beeinflussung der Steuerung zu vermeiden. Bei der Größenfestlegung ging es daher hauptsächlich um das Finden einer geeigneten Länge, die ich auf den genannten Wert festgesetzt habe.

Zweiter Punkt ist die Form der Ablage. Ihre Eigenschaften wurden bereits im Teil Bildverarbeitung grob definiert. Ich habe mich entschlossen, eine Computer-

Tastatur-Verpackung (58x23x10 cm) als Ablage zu verwenden, bei der die, dem Teppich zugewandte, Seite von der Höhe her etwas verkleinert wird (auf 6 cm). Der Vorteil dieser Kiste ist ihre Breite, wodurch auch kleinere Navigationsungenauigkeiten toleriert werden. Die Mitte der Ablage wird mit dem definierten Farbmuster markiert.

Als Letztes müssen noch die Standorte von Roboter und Ablage definiert werden. Diese könnten z.B. variabel sein. Das würde dann bedeuten, dass sich der Roboter anfangs irgendwo auf und die Kiste irgendwo neben dem Teppich (aber direkt angrenzend) befindet. Hierbei entsteht allerdings ein Problem, das deutlich wird, wenn man sich einen möglichen Ablauf des Aufsammelvorganges vorstellt: Nach seiner Aktivierung würde der Roboter ein greifbares Objekt suchen. Hat er eines gefunden, dann würde er darauf zufahren und es aufnehmen. Anschließend müsste er nach der Ablage Ausschau halten und, wenn er sie entdeckt hat, zu ihr hinfahren. Das angesprochene Problem ist nun, dass der Roboter, um zur Ablage zu kommen, auf dem Teppich entlangfahren muss. Dort können sich allerdings noch weitere Objekte befinden, denen er, um eine Kollision zu vermeiden, immer ausweichen müsste. Ist der Teppich dabei relativ „voll", würde dies zu einer nahezu unlösbaren Navigationsaufgabe führen.
Ich habe mich daher gegen eine Lösung mit variablen Standorten entschieden, und stattdessen diese festgesetzt. Um nun den eben beschriebenen Fall auszuschließen, sind sie dicht beieinander angeordnet. Die Idee dabei ist, dass der Roboter mit seiner Suche direkt vor der Kiste anfängt und sich anschließend, indem er immer das nächstgelegene Objekt aufnimmt, nach und nach bis zum Ende durcharbeitet. Auf eine Ausweichsteuerung könnte somit verzichtet werden. Liegen direkt vor der Kiste mehrere Objekte, dann ist es aber trotzdem noch möglich, dass der Roboter beim Aufnehmen bzw. Ablegen andere Objekte berührt, oder über sie hinwegfährt. Es sollte somit darauf geachtet werden, dass dieser Platz relativ frei ist, um ein Funktionieren der Steuerung zu gewährleisten.
In Abbildung 20 auf der nächsten Seite, auf der eine schematische Darstellung des Einsatzortes zu sehen ist, sind die Positionen von Roboter und Ablage an der linken unteren Ecke eingezeichnet.

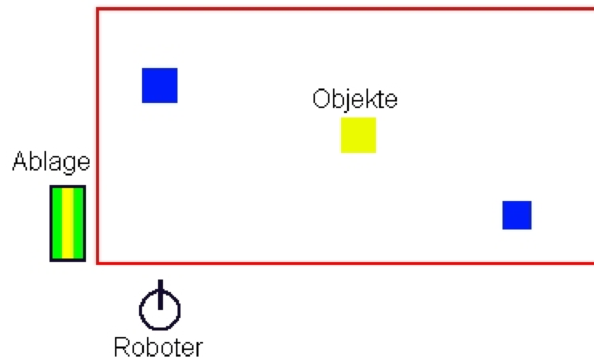

Abbildung 20: Schematische Darstellung des Einsatzortes

Die Ablage ist dabei durch die in der Konzeption der Bildverarbeitungskomponente bestimmte Farbkombination und der Roboter durch das aus dem Pioneer-Simulator bekannte Symbol dargestellt. Wie zu erkennen ist, befindet sich der Roboter zu Beginn der Aufgabe außerhalb des Teppiches. Dies liegt darin begründet, dass er auf dem Teppich zunächst einen zu großen Raum beanspruchen würde. Des Weiteren würde ein in der Realität eingesetzter „Aufräumroboter“ mit hoher Wahrscheinlichkeit auch nicht ständig im Kinderzimmer herumstehen, sondern hätte vielmehr eine abgeschiedenen Platz außerhalb bei einer Ladestation. Ein solcher Fall soll hiermit dargestellt werden.
Zusammengefaßt lauten somit die genannten Randbedingungen:

- Größe des Teppiches ca 1.3x3 m
- Größe der Ablage (Computer-Tastatur-Verpackung) 58x23x10 cm
  (an der Teppichseite h=6 cm)
- Roboter und Ablage befinden sich vor der linken unteren Teppichseite, wobei die untere Begrenzung eine lange und die linke Begrenzung eine kurze Teppichseite darstellt (vgl. Abbildung 20)
- Vor der Ablage sollten sich kaum Objekte befinden

12.3 Steuerungsarchitektur

Im Teil „Theoretische Grundlagen der Robotersteuerung“ wurden bereits die wesentlichen Steuerungsarchitekturen vorgestellt. Es stellt sich zunächst die Frage, welche dieser Architekturen zur Lösung der Aufgabenstellung am geeignetsten ist.
Wenn man sich dazu die Aufgabe aus Sicht der Steuerung einmal genauer ansieht, lässt sie sich in die folgenden Schritte unterteilen:

1. Objekt suchen
2. Zum Objekt fahren
3. Objekt aufnehmen
4. Zur Ablage fahren
5. Objekt ablegen und wieder zurück zu 1.

Zusätzliche Randbedingungen sind, möglichst nicht über andere Steine oder die Teppichbegrenzungen zu fahren. Außerdem ist zu beachten, dass der Roboter, nachdem er den gesamten Teppichbereich abgesucht und kein Objekt mehr gefunden hat, wieder zur Startposition zurückkehren soll. Er muss also „wissen“, wann seine Arbeit beendet ist. Man kommt daher um ein systematisches Absuchen des Teppiches nicht herum. Mit einem rein reaktiven Ansatz ist so etwas allerdings schwer möglich, da dort ohne ein genaues Weltwissen gearbeitet wird. Des Weiteren besteht eine rein reaktive Steuerung nur aus einer Menge unabhängiger, parallel laufender Module, die jeweils eine kleine bestimmte Funktion haben und direkt auf bestimmte Sensordaten reagieren können (derartige einfache Verhalten können im Saphira-System mit den *Behaviors* dargestellt werden). Für die gegebene Aufgabenstellung sind solche parallelen Aktionen zunächst aber nicht gewollt, da schon im Voraus klar ist, welche Funktion wann benötigt wird. Des Weiteren stellt der Aktionsablauf in diesem Fall auch eine klar definierte Sequenz (siehe oben) dar. Man kann sich somit eine Steuerung aus funktionell verschiedenen Modulen vorstellen, die von einem zentralen Modul aus koordiniert werden, das dann diesen sequentiellen Ablauf steuert.
Wie schon im Teil über die Laborumgebung erwähnt wurde, bietet das Saphira-System neben den *Behaviors* noch die Möglichkeit, sog. *Activities* einzusetzen.

Diese sind für komplexere Aufgaben gedacht und auf Grund ihrer Struktur auch für einfache Planungen geeignet. Da in unserer Steuerung von einer Aktionssequenz ausgegangen wird, müssen natürlich verschiedene Fälle, z.B. wenn bestimmte Aktionen fehlschlagen, abgefangen werden. Somit ist eine gewisse Planung notwendig, wozu die *Activities* eine einfache Möglichkeit bieten.
Die Architektur für meine Steuerung soll also zunächst auf einem deliberativen Ansatz mit *Colbert-Activities* basieren. Dies schließt aber nicht aus, dass diese *Activities* auch teils reaktive Komponenten besitzen bzw. aufrufen können. Es würde dabei dann eine hybride Architektur entstehen, wobei der Schwerpunkt aber trotzdem auf dem planerischen Anteil liegt.
Wichtig ist noch, dass eine direkte Planungskomponente in der gewählten Architektur nicht vorgesehen ist. In den Activities soll vielmehr der Plan schon vordefiniert und nur noch abgearbeitet werden.
Die Orientierung in die eine (deliberative) Architektur-Richtung muss auch nicht gleichzeitig bedeuten, dass man die Prinzipien der anderen unbeachtet lässt. Als Beispiele aus dem reaktiven Bereich seien das Einfach- und Robusthalten genannt, wobei gerade die Robustheit des Systemes eine Aufgabe in dieser Arbeit darstellt.

Nachdem die Grundlagen der Steuerung bestimmt wurden, kann jetzt eine feinere Gliederung überlegt werden. Wie eingangs schon dargestellt wurde, ist die Aufgabenstellung in einzelne Teilbereiche unterteilbar. Es ist daher empfehlenswert, jede dieser Teilfunktionen in einem eigenen Activity zu realisieren. Diese einzelnen Teile müssen dann nur noch von einem zentralen Steuerungs-Activity aus koordiniert werden. Neben der besseren Übersichtlichkeit und auch Wiederverwendbarkeit hat diese Aufteilung noch den Vorteil, dass einzelne Activities für sich genommen aktiviert und vor allem getestet werden können. Da die Activities später in erster Linie sequentiell gestartet werden sollen, kann somit schon auf das künftige Verhalten des Roboters geschlossen werden.

12.4 Sensoren

Als in Frage kommende Sensoren für die Aufgaben existieren nur die Kamera, die Odometrie sowie die im Greifer eingebauten Lichtschranken. Das Sonar kann leider

nicht zum Einsatz kommen, da die gesuchten Objekte zu klein sind und der Teppichrand auch nicht als „Wand“ vorhanden ist.
Der auch in der Aufgabenstellung definierte Hauptsensor ist also die Kamera. Es gilt daher als Erstes, das Sichtfeld des Roboters zu untersuchen. Zum Einsatz bei dieser Diplomarbeit kam der Roboter „Alfa“. Dieser hat gegenüber der normalen Konstruktion eine schräg stehende Kamera, die es ermöglicht, die nähere Umgebung direkt vor dem Roboter besser einzusehen. Dabei ist aber zu beachten, dass sich die Kamera in vertikaler Richtung auf einer schrägen Ebene dreht, was wiederum zu leicht verzerrten Bildern führt. Grundsätzlich hat die Konstruktion aber den Vorteil, dass der Roboter gerade bei den eingesetzten kleinen Objekten nicht allzulange „blind“ ist, wenn er zum Greifen auf sie zufährt. Wie Tests gezeigt haben, kann der Roboter in einem Abstand von ca. 20 cm trotzdem nichts sehen. Die Größe des Roboters noch mit eingerechnet, entsteht dann vom „Mittelpunkt“ des Roboters ein Kreis mit einem Radius von 40 cm, der für ihn von seiner Position aus nicht einsehbar ist. Dies muss bei den künftigen Verhalten berücksichtigt werden. Außerdem ist zu beachten, dass die aufgenommenen Bilder beim zur Zeit eingesetzten Bildverarbeitungsserver (näheres dazu im Abschnitt über die Implementierung der Bildverarbeitung) eine ziemlich schlechte Qualität besitzen und die Bildverarbeitung auf weite Entfernungen schnell ungenau wird.

12.5 Das Suchverhalten

Als Nächstes soll jetzt überlegt werden, wie letztendlich das Verhalten des Roboters aussehen soll. Die Grundlagen dafür bilden zunächst die in Abbildung 20 auf Seite 83 zu sehende schematische Darstellung des Einsatzortes sowie die im vorletzten Abschnitt genannten Rahmenbedingungen.
Die Grundidee des Zielverhaltens wurde dort bereits vorweggenommen. Diese war, dass der Roboter mit seiner Suche direkt vor der Ablage beginnt, und sich dann langsam vorarbeitet. Die allgemeinen Schritte dazu lauten vereinfacht: Objekt suchen, Objekt greifen und Objekt ablegen. Der Schwerpunkt in diesem Teil liegt jetzt darin, einen exakten Ablauf für ein Suchverfahren zu finden, mit dem der gesamte Teppich abgesucht werden kann und wo der Roboter am Ende auch „weiß“, dass seine Aufgabe beendet ist.

Grob kann die Suche folgendermaßen unterteilt werden:
Erster Schritt ist, den Bereich vor der Ablage abzusuchen und gefundene Objekte bei ihr abzuliefern. Wenn dieser Vorgang abgeschlossen ist, kann auf Grund der geringen Breite des Teppiches anschließend auch gleich noch der von der Roboterstartposition aus obere Teil abgesucht werden. Der weitere Aufsammelvorgang braucht dann nur noch in Längsrichtung laufen.
Abbildung 21 zeigt diese drei Bereiche nummeriert nach ihrer Ausführungsreihenfolge.

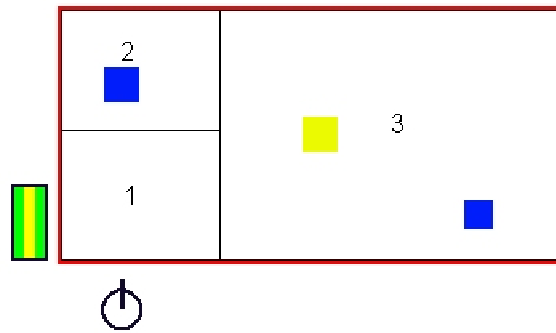

Abbildung 21: Die Suchgebiete

Die Details einmal außer acht gelassen, kann dies wie folgt realisiert werden:
Der Roboter befindet sich zu Beginn außerhalb des Teppiches. Von dort aus kann er das erste Gebiet überblicken und absuchen. Werden dabei Objekte gefunden, dann fährt der Roboter auf sie zu, greift sie, fährt zur Kiste und legt sie ab. Da der Roboter in einem Radius von 40 cm nichts sehen kann, ist es empfehlenswert, dass er, nachdem ein Objekt im ersten Sektor gefunden und abgelegt wurde, wieder auf die Startposition fährt. Von dort aus kann er dann die Suche in diesem Sektor fortsetzen.
Ist dieser Vorgang beendet, fährt er zu einem Punkt vor der Kiste und sucht dann den Bereich von Sektor 2 ab. Wenn dies dann ebenfalls erledigt ist, kann er sich in Längsrichtung drehen und beginnen Sektor 3 zu untersuchen.
Hierfür gibt es verschiedene Möglichkeiten.
Zunächst könnte man sich vorstellen, dass der Roboter von einem Punkt vor der

Ablage den gesamten Teppich überschauen kann. Wie allerdings schon erwähnt wurde, ist dies mit der momentan vorhandenen Bildverarbeitung auf Grund der schlechten Bildqualität, gerade in größeren Entfernungen, nicht möglich.
Es muss daher eine schrittweise Suche zum Einsatz kommen, für die ebenfalls wieder verschiedene Alternativen exisitieren.
Eine Möglichkeit wäre, dass der Roboter nacheinander verschiedene in Längsrichtung angeordnete Punkte, die jeweils in einem bestimmten Abstand auseinanderliegen, anfährt und von dort aus den jeweils vor ihm liegenden Bereich absucht. Wurde dabei von einem Punkt aus ein Objekt gesichtet, dann wird dieses aufgenommen und in der Kiste abgelegt. Anschließend könnte der Roboter dann auf seine letzte Position zurückfahren und die Suche fortsetzen.
Auf Grund der Odometrie-Ungenauigkeiten kann die Navigation zu den einzelnen Punkten aber nicht exakt erledigt werden. Abhilfe würde hier nur die Verwendung einer Karte in Saphira und die ständige Neukalibrierung der internen Position schaffen. Eine solche Kalibrierung ist allerdings nur unter Zuhilfenahme der Bildverarbeitungsdaten möglich. Der Roboter könnte z.B. die Abstände von seiner Position zu den Teppichlinien berechnen und auf Grund dessen dann seine Position in der Karte neu bestimmen. Diese Aufgabe kann allerdings auch nicht genau erledigt werden. Zum einen kann nicht davon ausgegangen werden, dass sämtliche Linien von der momentanen Roboterposition aus sichtbar sind und zum anderen werden für die Berechnung verschiedene Sensorinformationen benötigt, bei denen jeweils Ungenauigkeiten eingeplant werden müssen, wie z.B. die Lage der Teppichlinie im Bild. Zusätzlich entstehen durch die bei der Ausrichtung notwendigen Drehbewegungen des Roboters neue Positionsungenauigkeiten.
Weiterer Nachteil einer solchen, sich ständig neu kalibrierenden, Lösung ist auch, dass das sich daraus ergebende Roboterverhalten für den Betrachter etwas „unnatürlich“ aussieht.

Ein zweiter Ansatz ist daher, eine beim Fahren laufende Objektsuche einzusetzen. Auch hierfür gibt es mehrere Möglichkeiten.
Man könnte z.B. davon ausgehen, dass die Kamera in ihrer horizontalen Richtung fest eingestellt ist und immer in Fahrtrichtung des Roboters blickt. Da dann allerdings nicht die gesamte Teppichbreite überblickt werden kann, würde dies

bedeuten, dass sich der Roboter in einer Kurvenform über den Teppich bewegen müsste. Dafür in Frage käme beispielsweise eine sinusartige Kurve. Ein solcher Ansatz ist dann aber auch für etwas größere Teppich geeignet.
Eine weitere Möglichkeit, die auch in dieser Arbeit eingesetzt werden soll, ist, dass die Kamera beim Fahren nach rechts und links geschwenkt wird. Hierzu sollte sich der Roboter dann auf einer parallel zum Teppichrand verlaufenden Fahrspur bewegen. Für den gegebenen Teppich kann diese Bahn in der Mitte des Teppiches angeordnet werden. Bei der gewählten Teppichbreite reicht es dann, wenn sich die Kamera jeweils um 60 Grad nach links bzw. rechts dreht. Um keinen Bereich auszulassen, muss diese Drehbewegung aber in Schritten von ca. 20 Grad erfolgen. Zu beachten ist hierbei, dass, wenn die Kamera gerade die linke Seite absucht, natürlich keine Objekte auf der rechten Seite erkannt werden können. Dies bedeutet, dass die Fahrgeschwindigkeit an die Schwenkbewegungen angepaßt werden muss. Ein genauer Wert dafür lässt sich nur durch Test ermitteln.
Für große Teppiche könnte die Fahrspur ähnlich wie im ersten Ansatz anstelle einer Gerade auch eine Kurve darstellen. Für die gewählte, relativ schmale Teppichgröße ist dies allerdings nicht notwendig.
Das eben beschriebene Suchverhalten kann sich dann beenden, sobald die der Kiste gegenüberliegende Teppichbegrenzung erkannt und der Bereich direkt davor abgesucht wurde.

12.6 Die Phasen der Steuerung

Im Abschnitt „Steuerungsarchitektur“ wurde bereits festgelegt, dass das fertige System aus einzelnen Colbert-Activities bestehen soll, die jeweils einen bestimmten Aufgabenkomplex bearbeiten. Nachdem im letzten Punkt auch das grobe Zielverhalten bestimmt wurde, gilt es nun, die einzelnen benötigten Activities und ihre Aufgaben festzulegen.
Wie schon erwähnt wurde, wird zunächst ein zentrales Steuerungs-Activity benötigt, das sämtliche Aktionen koordiniert und eine Suche in der im letzten Teil bestimmten Reihenfolge durchführt. Des Weiteren sind spezielle Verhalten für die bereits genannten Unterpunkte notwendig:

- Objekt suchen

- Objekt aufnehmen
- Objekt ablegen

Im Folgenden sollen diese einzelnen Schritte genauer untersucht werden.

12.6.1 Objekt suchen

Wie bei der Konzeption der Bildverarbeitung festgestellt wurde, kann die Gültigkeit und Auswahl der zu greifenden Objekte direkt vom Bildverarbeitungsserver erledigt werden. In unserem Such-Activity brauchen wir uns von dieser Komponente somit nur noch das nächstgelegene Objekt liefern lassen. Dies bedeutet, dass sich die Suche im Wesentlichen auf die, bei ihr notwendigen, Kamera- und Roboterbewegungen beschränkt.
Weil die Bildverarbeitung in größeren Entfernungen ungenau wird und um immer genau das nächstgelegene Objekt zu erhalten, ist es dabei empfehlenswert, die Kamera sehr weit nach unten geschwenkt zu halten.
Da die gesamte Objekterkennung im Prinzip nur von der Bildverarbeitung erledigt wird, muss zusätzlich noch Folgendes beachtet werden:
Die Kamera darf nicht so positioniert werden, dass sie zuweit über den Teppichrand hinausblickt und die Begrenzungslinien im Bild nicht mehr sichtbar sind. Dies hätte zur Folge, dass Objekte außerhalb des Teppiches als gültig klassifiziert werden und der Roboter dann versuchen würde, diese aufzunehmen. Die Suche sollte also auf die Begrenzungslinien achten und sich gegebenenfalls beenden.

Ein Such-Activity müsste also die folgenden Funktionen beinhalten:

- Ein Bild durchsuchen und bei einem gefundenen Objekt entsprechende Schritte veranlassen (sich beenden, das Greif-Activity aufrufen oder eine globale Variable beschreiben)
  Je nach Fall müssen hierbei auch die erkannten Linien unterschiedlich berücksichtigt werden.
  Das Bilddurchsuchen kann durch Aufruf einer Funktion, die das nächstgelegene Objekt liefert, realisiert werden.

- Die Kamerabewegungen steuern
  Je nachdem, in welchem Sektor gesucht wird, sind unterschiedliche Kamerabewegungen denkbar.
  Vom Startpunkt aus ist es z.B. sinnvoll einmal von links nach rechts zu suchen, um somit als erstes die direkt vor der Ablage befindlichen Objekte zu erkennen, während bei der Suche in Sektor 3 ein ständiges nach rechts und links schwenken notwendig ist.

- Die Roboterbewegungen steuern
  Dies gilt in erster Linie für die Vorwärtsbewegung bei der Suche in Sektor 3.

- Die Begrenzungslinien überwachen.
  Da das Suchverhalten die Roboterbewegungen steuert, kann es auch die Linien überwachen. Eine explizites „Fahre nicht über den Teppichrand"-Verhalten, ist somit nicht notwendig.

12.6.2 Objekt aufnehmen

Wenn jetzt ein Objekt im Bild als gültig erkannt wurde, heißt es, auf dieses zuzufahren und es aufzunehmen. Als Grundlage hierfür verwende ich das von meiner Gruppe für die Abschlußaufgabe der Lehrveranstaltung „Applikationen intelligenter Systeme" im WS 99/00 entwickelte Verhalten.
Dieses ist folgendermaßen charakterisiert:
Vorraussetzungen sind, dass der Objektschwerpunkt annähernd in Bildmitte liegt, der Greifer geöffnet ist und die Kamera in die gleiche Richtung wie der Roboter ausgerichtet ist, d.h. der Pan-Winkel der Kamera hat den Wert 0. Die vertikale Ausrichtung der Kamera soll dann auch nicht weiter verändert werden. Der Bildverarbeitungsserver liefert jeweils die Schwerpunktkoordinaten des gesuchten Objektes.
Der Ablauf ist nun Folgender:

- Der Roboter bewegt sich mit geringer Geschwindigkeit auf das Objekt zu

- Wenn das Objekt um einen Schwellwert nach rechts oder links vom Bildmittelpunkt abweicht, dreht sich der Roboter in die entsprechende Richtung

- Wenn das Objekt um einen Schwellwert nach oben oder unten vom Bildmittelpunkt abweicht, schwenkt die Kamera in die entsprechende Richtung, d.h. der Tilt-Winkel der Kamera wird angepaßt

- Nachdem das Objekt nach unten aus dem Bild verschwunden ist, bewegt sich der Roboter noch solange in die Richtung, bis die innere Lichtschranke des Greifers ein Objekt meldet (Dies kann so erledigt werden, da das Objekt nun ca. 10 cm vor dem Greifer liegt und bei gleichbleibender Fahrtrichtung somit irgendwann im Greifer landen sollte)

Anschließend wird der Greifer geschlossen und das Objekt angehoben.
Da dieses Verhalten gut funktioniert, sind hierfür keine größeren Änderungen notwendig. Es muss nur noch so angepaßt werden, dass zunächst die Vorbedingungen erfüllt sind, also das Objekt annähernd im Bild zentriert ist und die Kamera und der Roboter in dieselbe Richtung ausgerichtet sind. Des Weiteren muss immer der Schwerpunkt des nächstgelegenen Objektes vom Bildverarbeitungsserver geholt werden. Es ist hierbei zu beachten, dass, nachdem ein Objekt nach unten aus dem Bild herausgelaufen ist, sich Roboter und Kamera nicht nach einem anderen möglicherweise vorhandenen Objekt ausrichten, sondern das Erstere zunächst aufgenommen wird.

12.6.3 Objekt ablegen

Nachdem ein Objekt angehoben wurde, muss der Roboter damit zur Kiste fahren und es dort ablegen.
Dazu wurde bereits definiert, dass die Markierung der Ablage von der Bildverarbeitung erkannt werden soll. In der Steuerung könnte der Ablauf dann so aussehen, dass zunächst die Kiste gesucht wird und der Roboter anschließend (ähnlich wie bei den Objekten) darauf zufährt. Bei der optischen Ablageerkennung existieren jedoch zwei Probleme. Das Erste ist, dass der Roboter bekanntermaßen in einem bestimmen Umkreis nichts sehen kann. Eine zweites Problem ergibt sich durch die Beschaffenheit der Kiste.
Bei der urprünglichen Farbmustererkennung wurde von einer Flasche, die ja von allen Seiten gleich aussieht, ausgegangen. Die jetzt eingesetzte markierte Kiste jedoch sieht je nach Betrachtungswinkel anders aus. Wenn die Kamera in einem

spitzen Winkel auf sie schaut, dann ist von der angebrachten Markierung nicht mehr viel zu erkennen, wodurch auch keine exakte Identifizierung mehr möglich ist. Da der eingesetzte Teppich jedoch nicht sehr breit ist und somit die möglichen Blickwinkel des Roboters zur Ablage keinen großen Spielraum besitzen, fällt dieses Problem für größere Entfernungen jedoch weg. Trotzdem existieren aber Einschränkungen in einem ca. 1 m breiten Streifen vor der Ablage.
Direkt vor ihr (Suchgebiet 1) entstehen sie dadurch, dass der Roboter einfach nichts sieht und oberhalb von ihr (Suchgebiet 2) sind sie winkelbedingt (vgl. Abbildung 21 auf Seite 87).
Wurden in diesem Bereich also Objekte gefunden und sollen sie anschließend zur Kiste gebracht werden, dann kann hierfür nicht auf die Ablageerkennung der Bildverarbeitung zurückgegriffen werden. Die einzige Alternative ist somit die Odometrie.
Zunächst muss also ein Punkt-Artefakt, dass sich direkt vor der Ablage befindet, generiert werden. Dies kann beispielsweise von der Startposition aus erledigt werden. Soll nun ein Objekt abgelegt werden, dann kann der Roboter einfach wieder zu diesem Punkt zurückkehren.
Bei beiden Möglichkeiten, also beim bildverarbeitungs- und beim odometriegesteuerten Zur-Ablage-Fahren müssen aber die hierbei möglicherweise auftretenden Navigationsungenauigkeiten berücksichtigt werden. Diese können aber kompensiert werden, indem ein Ausrichten vor der Kiste erfolgt. Als Orientierung dienen dabei die beiden Teppichlinien.

Abschließend lässt sich also sagen, dass für das Objekt-Ablegen zwei verschiedene Verfahren benötigt werden. Für die nähere Umgebung der Kiste ist dies eine odometrie- und für größere Entfernungen eine bildverarbeitungsbasierte Lösung. Diese Aufteilung hat den Vorteil, dass sie der Sensorverlässlichkeit entspricht. Im Nahbereich ist die Odometrie genauer, während die Bildverarbeitung keine Daten liefern kann. Bei größeren Entfernungen wird die Odometrie jedoch sehr ungenau, während das Muster der Kiste in der Kamera noch richtig erkennbar ist.

12.7 Bildverarbeitungsdaten

Als Letztes sollen noch einmal die von der Bildverarbeitung im gemeinsamen Speicherbereich abzulegenden Daten zusammengefaßt werden:

- Die Schwerpunktkoordinaten des nächstgelegenen Objektes
- Die Schwerpunktkoordinaten der Markierung der Ablage
- Die erkannten Linien als Geradengleichungen

Teil V

Implementierung

13 Implementierung der Bildverarbeitung

13.1 Einleitung

Dieser Teil der Arbeit befaßt sich mit der Implementierung der Bildverarbeitungskomponente. Wie schon oft erwähnt wurde, dient als Grundlage für die Bildverarbeitung der bereits vorhandene Bildverarbeitungsserver sowie das Konzept über einen gemeinsamen Speicher zwischen ihm und einer in Saphira zu ladenen Bibliothek. Zunächst soll daher diese Applikation näher vorgestellt und anschließend meine Änderungen und Erweiterungen beschrieben werden. Ich möchte hierbei allerdings nicht allzu sehr ins Detail gehen, sondern nur die grundlegenden Ideen und Konzepte vorstellen.

13.2 Der Bildverarbeitungsserver

13.2.1 Übersicht

Als Bildquelle für den Bildverarbeitungsserver dient zunächst die im Steuerungsrechner (Laptop) eingebaute PCMCIA-Framegrabber-Karte FG30 der Firma HaSoTec. Diese wiederum erhält die Bilder direkt vom Roboter, mit dem sie über eine Funkverbindung verbunden ist.
Der Framegrabber-Karte liegt ein Softwareentwicklungs-Paket bei, in dem auch zwei OCX-Steuerelemente enthalten ist. Diese können in eigene Anwendungen integriert werden und kümmern sich dann um die Steuerung des Framegrabbers und die Anzeige der Bilder.
Für die Steuerung wird das FG30-OCX und für die Bildanzeige das DIB-OCX eingesetzt. Letzteres stellt ein Bitmap-Bild dar und erledigt zusätzlich noch die Darstellung des Bildes.
Diese beiden Steuerelemente wurden auch im Bildverarbeitungsserver verwendet.
Ferner kam für die Entwicklung dieser Anwendung die Entwicklungsumgebung

MS Visual C++ 5.0 und die darin verfügbare MFC-Bibliothek zum Einsatz.
Abbildung 22 zeigt die grafische Benutzer-Oberfläche des Bildverarbeitungsservers. Neben dem Teil für die Darstellung des Bildes (über das DIB-OCX) sind in ihr auch verschiedene Schaltflächen zur Steuerung der Bildverarbeitung zu sehen. Bevor aber weitere Details dazu erläutert werden, soll zunächst die Architektur des Servers beschrieben werden.

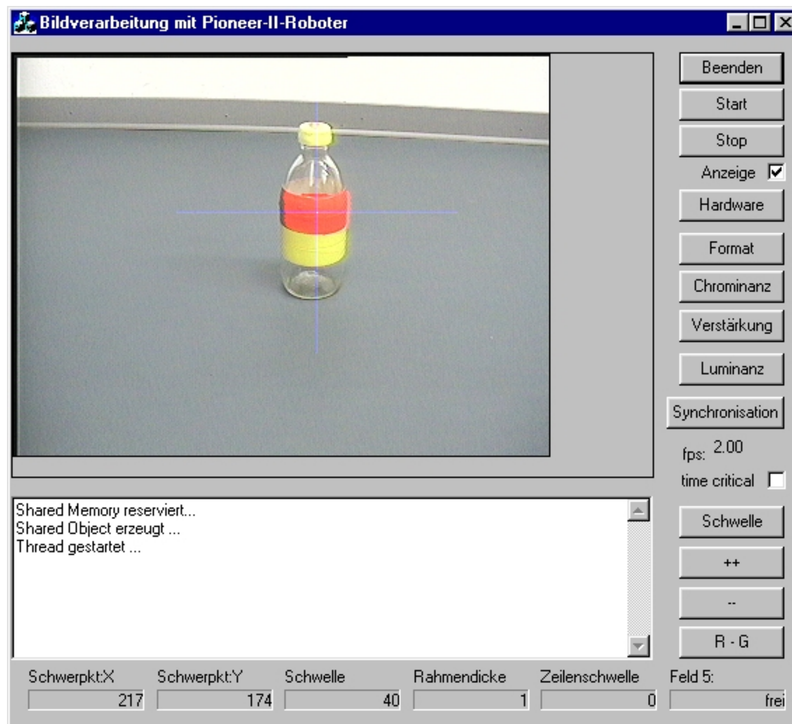

Abbildung 22: Der Bildverarbeitungsserver

13.2.2 Architektur

Grundsätzlich gibt es im Server zwei verschiedene Threads, die parallel laufen. Zum einen ist dies der Haupt-Thread der Anwendung und zum anderen ein Arbeits-Thread für die eigentlichen Bildverarbeitungsroutinen. Diese Aufteilung war notwendig, um bei laufenden Bildverarbeitungsmethoden die Oberfläche noch bedienbar zu halten.

Der Arbeits-Thread kann dabei über in der Oberfläche enthaltene Schaltflächen gestartet und gestoppt werden und aktualisiert nach jeder Verarbeitung eines Bildes die genutzten Kommunikationsvariablen, die im gemeinsamen Speicher untergebracht sind.

Der gemeinsame Speicher Für die Kommunikation zwischen Saphira und der Bildverarbeitung wurde das Konzept über einen gemeinsamen Speicherbereich realisiert. Zu beachten ist, dass der gemeinsame Speicher nur die Verbindung zwischen Bildverarbeitungsserver und einer in Saphira zu ladenden Bibliothek schafft und nicht der Kommunikation zwischen Haupt- und Arbeits-Thread dient.
Die Implementierung des gemeinsamen Speichers ist in den Dateien `shared_mem.h` und `shared_mem.c` untergebracht. Für die Kommunikation werden dort eine Menge von Variablen angelegt, auf die dann mit den entsprechenden Funktionen zugegriffen werden kann. Des Weiteren sind dort Funktionen zum Initialisieren und Freigeben des Speichers vorhanden.

Der Haupt-Anwendungs-Thread Zum Einsatz für die Entwicklung des Bildverarbeitungsservers kam das Programm MS Visual C++ 5.0. Mit dem darin enthaltenen MFC-Anwendungsassistenten wurde zunächst ein Anwendungsgerüst für eine Dialogfeldbasierte Anwendung erstellt.
Hierbei entstanden eine Anwendungsklasse (`CBVSApp`) eine Dialog-Klasse (`CBVSDlg`) sowie eine Resourcen-Datei. Letztere legt das Aussehen der grafischen Benutzeroberfläche und die darin enthaltenen Steuerelemente fest und kann über einen mitglieferten Editor leicht verändert werden.
Dieses Gerüst wurde nun erweitert, um die gewünschte Funktionalität bereitzustellen.
Als Erstes wurden dazu die in der Übersicht erwähnten OCX-Steuerelemente eingefügt. Hierbei entstanden auch zwei Wrapper-Klassen, die die Funktionalität dieser Elemente von C++-Funktionen aus steuerbar machen. Für das FG30-OCX wurde die Klasse `CFG30` und für das DIB-OCX die Klasse `CDib` erstellt, wobei `CFG30`-Objekte den Framegrabber und `CDib`-Objekte die von ihm gelieferten Bilder repräsentieren. Des Weiteren wurden noch diverse Schaltflächen zur Steuerung des Framegrabbers und der Bildverarbeitung sowie ein größeres Ausgabefeld der Ober-

fläche hinzugefügt. Um diese Komponenten der Oberfläche für den Arbeits-Thread zugänglich zu machen, werden beim Initialisieren des Dialoges Zeiger auf sie erstellt und in globalen Variablen abgelegt.

Der Arbeits-Thread Für die eigentlichen Bildverarbeitungsfunktionen kommt ein separat laufender Thread zum Einsatz. Dieser ist in der Datei `thread.cpp` definiert und dort in einer eigenen Funktion (`WorkerThread`) untergebracht. Wie schon erwähnt, wird der Arbeits-Thread von der Oberfläche aus gestartet und gestoppt. Die eigentlichen Verarbeitungsschritte sind im Thread in einer Schleife untergebracht. Der Ablauf dort ist Folgender:

- Bild holen (über `CFG30`)
- Bild verarbeiten
- Kommunikationsvariablen aktualisieren
- Bild anzeigen (über `CFG30`)

Soll der Thread gestoppt werden, dann wird von der Oberfläche aus ein globales Flag gesetzt, das vom Arbeits-Thread vor jedem Schleifendurchlauf abgeprüft wird.

13.2.3 Oberflächen-Einstellungen

Die grafische Benutzer-Oberfläche soll natürlich nicht nur zur Anzeige der verarbeiteten Bilder sondern auch zur Steuerung des Framegrabbers und der Bildverarbeitung dienen. Hierzu wurden dort entsprechende Schaltflächen eingefügt, welche in Abbildung 22 auf Seite 96 an der rechten Seite zu sehen sind. Die obersten drei sind dabei zum Verlassen der Anwendung sowie zum Starten und Beenden des Arbeits-Threads bestimmt. Mit den sechs darunterliegenden Knöpfen kann der Framegrabber gesteuert werden (z.B. Bildformat verändern), wobei jeweils die entsprechenden Methoden des `CFG30`-Objektes aufgerufen werden.
Des Weiteren sind in der Oberfläche auf der rechten und auf der unteren Seite noch weitere Elemente zu sehen. Diese stellen nun bildverarbeitungsspezifische Funktionalitäten dar.

Eine Grundidee ist ja, dass der Bildverarbeitungsserver unabhängig von den direkten Bildverarbeitungsmethoden sein soll, um ihn für mehrere solcher Aufgaben einsetzen zu können. Die gesamte Definition der eigentlichen Bildverarbeitung soll dabei in den Thread-Dateien untergebracht werden. Trotzdem sollte die Verarbeitung aber von außen beeinflußbar sein, um beispielsweise Schwellwerte zur Laufzeit ändern zu können. Hierfür gibt es nun die spezifischen Knöpfe und Ausgabefelder. Die Beschriftung dieser Elemente kann dabei vom Thread aus verändert werden, indem Funktionen aufgerufen werden, die auf die, vom Dialog bei der Initialisierung abgelegten, globalen Zeiger-Variablen zugreifen.
Die Knöpfe auf der rechten Seite dienen nun dazu, von der Oberfläche aus bestimmte vom Arbeits-Thread benutzte Werte zu verändern. Wird einer dieser Knöpfe gedrückt, dann wird die Funktion `void action(int button)` (`button` = Index des aktivierten Knopfes) aufgerufen, die in der Datei `thread.cpp` definiert ist. Dort kann dann entschieden werden, ob und wie darauf reagiert werden soll.
Eine dafür in Frage kommende Funktionalität, die für eine Beispiel-Anwendung bereits integriert ist, ist Folgende:
Die Bildverarbeitung hat zwei veränderbare Variablen; einen Schwellwert für eine Farbextraktion und einen Wert für die Breite des äußeren Bildrandes, für den die Bildverarbeitung die Pixel nicht berücksichtigen soll. Diese sollen nun über die Oberfläche zur Laufzeit verändert werden können.
Über den obersten frei definierbaren Knopf wird daher festgelegt, welcher dieser Werte gerade verändert wird und über die beiden darunterliegenden wird der gerade ausgewählte Wert inkrementiert oder dekrementiert. Die eigentliche Funktionalität ist dabei über die `action`-Funktion in der Thread-Datei definiert. Dort sind dann auch Werte für Schwelle, Rahmendicke und gerade ausgewählten Modus vorhanden, die nach der Aktivierung der entsprechenden Schaltfläche angepaßt werden. Zusätzlich werden dort auch die Beschriftungen der Knöpfe und der Inhalte der unteren Felder (zur Darstellung der gerade aktuellen Werte) aktualisiert.

13.3 Die BV-Bibliothek für Saphira

Die Ergebnisse der Bildverarbeitung werden in einem gemeinsamen Speicher zwischen dem Bildverarbeitungsserver und einer in Saphira zu ladenden Bibliothek

abgelegt. Im Folgenden soll nun diese Bibliothek kurz beschrieben werden.
Für Saphira unter Windows können Bibliotheken als DLLs (Dynamic Link Libraries) genutzt werden. Dabei müssen in der Bibliothek die Funktionen `void sfLoadInit(void)` und `void sfLoadExit(void)` vorhanden sein, welche beim Laden (`sfLoadInit`) und Entladen (`sfLoadExit`) aufgerufen werden. Beim vorhandenen Bildverarbeitungsserver sind diese Funktionen in der Datei `bv_com.c` enthalten, welche dann zu einer extra DLL kompiliert wird.
Um jetzt die Kommunikationsvariablen für Saphira bekannt zu machen, wird in der `sfLoadInit`-Funktion zunächst der gemeinsame Speicher initialisiert. Anschließend werden die Variablen nacheinander mit der Saphira-Funktion `sfAddEvalVar` bekannt gemacht.
Beim Entladen der DLL wird dann der Zugriff auf den gemeinsamen Speicher wieder freigegeben.

13.4 Änderungen am Bildverarbeitungsserver

Ein Problem beim Bildverarbeitungsserver unter Verwendung der OCX-Steuerelemente ist die geringe Geschwindigkeit (Framerate). Sie liegt auch ohne Bildverarbeitungsroutinen für Bilder mit einer Auflösung von 384x288 und 24 Bit Farbtiefe bei kaum 3 Hz. Auch eine Verringerung der Auflösung bringt dabei keine größeren Geschwindigkeitsvorteile.
Abhilfe für dieses Problem schaffte Bogdan Kwolek, der im Sommer 2000 die Bildverarbeitung verbesserte (Näheres dazu unter [BIH00]). Anstatt den Framegrabber über das OCX anzusteuern, hat er dabei Methoden entwickelt, die dies direkt über Assembler-Befehle erledigen. Das Ergebnis dessen ist eine schnellere Bildverarbeitung mit ca. 8 Hz. Leider hat aber auch diese Lösung einen kleinen Nachteil. Zum einen sind die Bilder von der Größe her etwas kleiner (256x256) und zum anderen sind es auch keine echten 24-Bit-Bilder mehr. Sie werden zwar als 24-Bit (8 Bit pro Kanal) abgespeichert, vom Framegrabber kommen sie allerdings nur mit 15 Bit (5 Bit pro Kanal). Speziell Letzteres bewirkt, dass die Bilder von der Qualität her deutlich schlechter als die Ursprungsbilder sind.
Auf Grund der höheren Verarbeitungsgeschwindigkeit habe ich mich aber trotzdem dazu entschieden, diese assemblerbasierten Framegrabber-Funktionen zu verwenden.

Im ursprünglichen Bildverarbeitungsserver sind sämtliche Bildverarbeitungsfunktionen in der Thread-Funktion enthalten. Dies ist möglich, da dort nur auf einem Bildpuffer gearbeitet wird und auch die Verarbeitungsschritte nicht so aufwändig sind. Da für meine Aufgabe jedoch viele verschiedene Bearbeitungsschritte notwendig sind und dabei auch mehrere Bildpuffer benötigt werden, würde die Beibehaltung dieses Konzeptes wahrscheinlich zu einer sehr unübersichtlichen Implementierung führen. Wie bei der Konzeption schon bestimmt wurde, werden die einzelnen Schritte daher als eine Menge von Funktionen implementiert, die jeweils ein Eingabe- und ein Ausgabebild als Parameter übernehmen können.
Zusätzlich habe ich mich dazu entschlossen, die Bildverarbeitung objektorientiert zu implementieren, d.h. also eine Bildverarbeitungs-Klasse und eine Bild-Klasse zu entwickeln.
Um dies zu realisieren, habe ich daher die drei Klassen `CImage`, `CFG30Image` und `CImageProcessing` erstellt.
`CImage` stellt dabei ein Bild dar und beinhaltet neben den eigentlichen Bilddaten noch Informationen zu den Bildabmessungen und zur Farbtiefe.
`CFG30Image` ist von `CImage` abgeleitet und stellt zusätzlich noch die von B.Kwolek entwickelten schnelleren Funktionen zum Bildgrabben bereit.
In `CImageProcessing` sind nun die eigentlichen Bildverarbeitungsfunktionen enthalten, welche in der Regel `CImage`-Objekte als Parameter übernehmen.

Der Ablauf für den Thread soll nun Folgender sein:
Nachdem der Thread gestartet wurde, erstellt er zunächst die benötigten Objekte:

- ein Bildverarbeitungsobjekt (`CImageProcessing`),
- ein `CFG30Image`-Objekt, um die Bilder zu grabben und
- je nach Notwendigkeit, eine Menge von `CImage`-Objekten, für die einzelnen Zwischenschritte

In der Schleife des Threads, ist der Ablauf vom Prinzip her gleichgeblieben:

- Bild grabben (über `CFG30Image`)

- Bild verarbeiten (über `CImageProcessing`)
- Bild anzeigen (die Ergebnisse der Verarbeitungsschritte (Objekte, Linien) können in das Ausgangsbild eingezeichnet und dieses dann dargestellt werden)

13.5 Implementierung der notwendigen Funktionen

Nachdem der Grundaufbau der Bildverarbeitung vorgestellt wurde, geht es im Folgenden nun um die konkrete Realisierung der eigentlichen Bildverarbeitungsmethoden, welche schon bei der Konzeption dieser Komponente vorgestellt wurden.
Sämtliche Funktionen sollen in der Klasse `CImageProcessing` untergebracht werden. Als Parameter übernehmen sie je nach Notwendigkeit ein oder zwei Bild-Objekte (in der Regel eine Eingabe- und ein Ausgabebild). Das Ausgabebild muss aber vorher schon angelegt und ein entsprechender Speicher für die Bilddaten reserviert worden sein. Es wird also nicht in der Funktion erstellt.
Zusätzlich existiert noch die Anforderung auch zur Laufzeit bestimmte Werte (z.B. Schwellwerte, Randbreite) festlegen zu können. Dies bedeutet, dass diese Werte noch zusätzliche Parameter darstellen. Die Veränderung und Aktualisierung der Werte soll dabei über die im Thread enthaltene `Action`-Funktion erledigt werden. Eine Festlegung der Randbreite ist hierbei notwendig, da die gelieferten Bilder an ihren äußeren Rändern oft Bildstörungen enthalten. Damit diese die eigentlichen Verarbeitungsroutinen nicht weiter beeinflussen, werden sie einfach herausgefiltert, d.h. die entsprechenden Pixel werden nicht weiter bearbeitet.

Den folgenden Teil möchte ich nun wieder in die drei Teile *Teppicherkennung*, *Spielzeugerkennung* und *Erkennung der Ablage* unterteilen.

13.6 Teppicherkennung

Zunächst sollen nocheinmal die für die Teppich(linien)erkennung benötigten Funktionen aufgeführt werden:

1. Eine Methode, die eine Filterung nach der Farbe *ROT* ermöglicht
   Eingabe: RGB-Farbbild → Ausgabe: Binärbild

2. Eine Kantendetektion mit einfachem Differenzoperator
   Eingabe: Binärbild → Ausgabe: Binärbild

3. Eine Methode zur Hough-Transformation mit Berechnung der Geradenbüschel
   Eingabe: RGB-Farbbild → Ergebnis: gefüllter Hough-Akkumulator

4. Eine Methode, die den Hough-Akkumulator über eine Mittelwertbildung analysiert

Wie man sieht, werden für die ersten beiden Funktionen Binärbilder benötigt. Auf Grund der einfacheren Bearbeitung habe ich mich dazu entschlossen, keine direkten Binärbilder sondern einkanalige 8-Bit-Grauwertbilder zu verwenden. Anstatt 0 und 1 werden dafür dann die Grauwerte 0 und 255 verwendet. Solche Bilder können dann auch besser dargestellt werden.
Um die Farbtiefe festzulegen, besitzt die Klasse `CImage` einen entsprechenden Konstruktor, dem man Bildbreite, -höhe und Anzahl der Kanäle als Parameter übergeben kann.

13.6.1 Filterung nach der Farbe ROT

Die Filterung des Bildes nach der Farbe ROT ist in der Methode

```
int CImageProcessing::RedFilter( int iBorderWidth, int iThreshold,
CImage *pImageIn, CImage *pImageOut )
```

untergebracht, wobei `iBorderWidth` die Breite des äußeren Bildrandes und `iThreshold` einen Schwellwert darstellt. `pImageIn` ist dabei ein Zeiger auf das 24-Bit-RGB-Farbbild und `pImageOut` ein Zeiger auf das 8-Bit-Grauwertbild, in dem als ROT interpretierte Bildpunkte den Wert 255 und allen anderen den Wert 0 besitzen.
In dieser Funktion werden nacheinander die Bildpunkte des Eingabebildes durchgegangen. Dabei werden dann für jeden Punkt Differenzen der Kanäle $ROT - GRÜN$ sowie $ROT - BLAU$ gebildet. Liegen beide Werte über dem übergebenen Schwellwert, dann wird der entsprechende Bildpunkt des Ausgabebildes auf den Wert 255 gesetzt und sonst auf den Wert 0.

13.6.2 Einfacher Kantenfilter

Der einfache Kantenfilter wird von der Methode

```
int CImageProcessing::EdgeDetection( CImage *pImageIn, CImage
*pImageOut )
```

bereitgestellt.
Als Eingabebild wird hier das Ergebnis des Rotfilters verwendet. Ausgabe ist wiederum ein 8-Bit-Grauwertbild, in dem Kantenpunkte den Wert 255 und alle anderen den Wert 0 besitzen.
Wie schon erwähnt, ist hier nur ein einfacher Kantenoperator notwendig, da mit einem Binärbild gearbeitet wird und eine Kante somit am Übergang zwischen 0 und 1 bzw. 0 und 255 auftritt. Es wird somit keine komplizierte Operatormaske benötigt.
Der Ablauf in der Funktion ist nun Folgender:
Jeder Bildpunkt wird mit seinem linken und seinem oberen Nachbarn verglichen. Treten dabei Unterschiede auf, ist also die Differenz zwischen dem betrachteten Bildpunkt und einem dieser Nachbarn ungleich Null, dann wird er als Kantenpunkt mit dem Wert 255 im Ausgabebild markiert und ansonsten auf 0 gesetzt.

13.6.3 Hough-Transformation

Mit der Hough-Transformation wird nun versucht, die extrahierten Kanten-Pixel zu Geraden zusammenzufassen. Sie ist in der Methode

```
int CImageProcessing::HoughTransformation( int iBorderWidth,
CImage *pImage )
```

untergebracht.
Der Algorithmus der Hough-Transformation ist im Prinzip recht einfach und wurde bereits im Theorie-Teil vorgestellt.
Zunächst wird dafür der sog. *Hough-Akkumulator* benötigt, der die Parameter aller möglichen Geraden darstellt. Für meine Implementierung ist er ein zweidimensionales Feld, das eine Membervariable der Bildverarbeitungsklasse (`CImageProcessing`) ist. Im Konstruktor dieser Klasse wird dafür Speicher reserviert, womit ein ständiges Neuanlegen bei jedem Funktionsaufruf entfällt. Seine Größe ist dabei von der

Bildgröße und der Genauigkeit abhängig.
Neben dem Akkumulator werden für die Berechnungen der Geradenbüschel noch verschiedene Sinus- und Cosinuswerte benötigt. Diese sind als look-up-Tabellen ebenfalls Membervariablen der Bildverarbeitungsklasse und werden beim Konstruieren gefüllt.
Der Ablauf ist wie folgt:
Als Erstes wird der Hough-Akkumulator initialisiert, d.h. alle Werte auf 0 gesetzt. Anschließend werden alle Bildpunkte des Eingabebildes nacheinander durchgegangen. Trifft man dabei auf einen Kantenpunkt (Wert != 0), dann wird das durch ihn laufende Geradenbüschel bestimmt (über die bekannte Formel $r = x \cdot cos\theta + y \cdot sin\theta$, in der für θ alle möglichen Werte eingesetzt werden) und die entsprechenden Akkumulatorzellen inkrementiert. Um später keine Geraden zu erhalten, die durch die eingestellte Randbreite beim Rotfilter entstanden sind, wurde diese der Funktion als Parameter übergeben und wird zur Bestimmung der zu bearbeitenden Bildpunkte eingesetzt.
Das Ergebnis der Transformation ist ein gefüllter Akkumulator, der jetzt ausgewertet werden kann.

13.6.4 Hough-Akkumulator-Analyse

Die Auswertung ist der wichtigste Schritt bei der Linienerkennung und wird in der Methode:

```
void CImageProcessing::AnalyzeHoughAccumulator( int iInit )
```

realisiert (der Parameter wird später erklärt).
Da die Linien zur weiteren Verarbeitung zunächst in der Bildverarbeitungsklasse, aber auch im gemeinsamen Speicher abgelegt werden sollen, muss dafür eine geeignete Form gewählt werden. Ich habe mich für eine Struktur mit der Bezeichnung `LINE` entschieden, da diese im Gegensatz zu Klassenobjekten mit `sfAddEvalStruct` auch in Saphira bekanntgemacht werden kann (alle Strukturen sind in `defs.h` definiert). Die Haupt-Elemente dieser Struktur sind die Parameter der jeweiligen Geradengleichung (Winkel θ und Abstand r). Zusätzlich gibt es noch einen Wert für die Anzahl der auf der Geraden liegenden Pixel (aus dem Hough-Akkumulator).
Der Ablauf bei der Akkumulatoranalyse ist wie folgt:

Auf Grund der Berechnung der Geradenbüschel gibt es im Akkumulator sehr viele Zellen, die einen Wert größer 0 besitzen. Um nun nur die wirklichen Geraden zu erhalten, wird über einen Schwellwert zunächst bestimmt, welche Zellen berücksichtigt werden. Der Schwellwert bedeutet dabei die Anzahl der mindestens auf einer Geraden liegenden Bildpunkte. Durch dieses Verfahren wird die Anzahl der zu berücksichtigenden Akkumulatorzellen schon sehr eingeschränkt. Die übriggebliebenen Zellen werden nun als `LINE`-Strukturen in einem dynamischen Feld gespeichert. Aus diesen Linien werden dann über weitere Toleranz(Schwell)werte ähnliche Geraden gesucht und zu Gruppen zusammengefaßt. Anschließend wird in den Gruppen jeweils der Mittelwert der Geradenparameter unter Berücksichtigung einer Wichtung nach der Anzahl der auf den Geraden liegenden Bildpunkte berechnet.
Das Ergebnis sind dann eine Menge von Geraden, die im gemeinsamen Speicher sowie für die Objekterkennung noch in der Bildverarbeitungsklasse gespeichert werden.

Die Teppichlinienerkennung ist hiermit zunächst abgeschlossen.

13.7 Spielzeugerkennung

Nächster Punkt ist die Realisierung einer Spielzeugerkennung auf Basis eines Farbkantenfilters. Die notwendigen Verarbeitungsschritte sind:

1. Farbkantenfilter anwenden
2. Konturen verbinden
3. Objekte extrahieren
4. Gültigkeit der Objekte feststellen

13.7.1 Farbkantenfilter

Der Farbkantenfilter ist in der Funktion

```
int CImageProcessing::ColorEdgeDetection( int iBorderWidth, int
iThreshold, CImage *pImageIn, CImage *pImageOut )
```

enthalten und funktioniert ähnlich wie der bei der Linienerkennung eingesetzte, nur das jetzt nach einer Kante in einem der drei Farbkanäle gesucht wird. Eingabe-Bild ist hierbei das originale vom Framegrabber gelieferte Bild und Ausgabe ein Grauwert-Bild, in dem Kantenpixel den Wert 255 und sonstige den Wert 0 besitzen.

13.7.2 Konturen verbinden

Um nun die Konturen zu verbinden, wurden bei der Konzeption die möglichen Verfahren eines Maximum-Operators sowie die Verwendung von morphologischen Operatoren vorgeschlagen. Letztere sind jedoch üblicherweise sehr rechenintensiv. Ich hatte mich daher zunächst entschlossen, einen Maximum-Operator einzusetzen, welcher in einer entsprechenden Funktion

```
int CImageProcessing::Maximum( CImage *pImageIn, CImage
*pImageOut )
```

enthalten ist und mit einer 3x3-Operatormaske arbeitet.

13.7.3 Objekte extrahieren

Nachdem die Objektbildpunkte jetzt soweit extrahiert sind, kann versucht werden, diese zu Objekten zusammenzufassen. Dies wird von der Methode

```
int CImageProcessing::ExtractObjects( CImage *pImage )
```

erledigt. Als Eingabe dient das Ausgabebild der Konturpunktverknüpfung. Die erkannten Objekte werden dabei in einem Feld, das Member der Bildverarbeitungsklasse ist, zur späteren Weiterverwendung gespeichert.
Wie schon erwähnt wurde, existierte eine entsprechende Funktionalität im Bildverarbeitungsserver bereits, sodass ich diese auch für meine Aufgabe verwendet habe. Sie arbeitet mit einer 4er-Nachbarschaft, d.h. sie verknüpft Bildpunkte zu Objekten, wenn diese in horizontaler oder vertikaler Richtung nebeneinander liegen. Für die Speicherung von Objekten wurde hierbei die Struktur `REGINFO` verwendet, die die Eigenschaften der Objekte (Schwerpunkt, Punkte des umschreibenden Rechtecks...) enthält.

13.7.4 Gültigkeit der Objekte feststellen

Die Gültigkeit von Objekten soll unter Zuhilfenahme der extrahierten Geradengleichungen festgestellt werden.
Befindet sich der Roboter auf dem Teppich, dann sind alle Objekte gültig, deren Schwerpunkt unterhalb aller sichtbaren Linien liegt.
Anders sieht es aus, wenn der Roboter noch nicht auf dem Teppich steht. Da dies aber nur am Start auftritt, kann dieser Fall folgendermaßen behandelt werden:
Wenn im Bild die Startlinie gefunden wird, dann sind die Objekte gültig, deren Schwerpunkt über ihr und unter allen anderen erkannten Linien liegt.
Um anzuzeigen, dass sich der Roboter bei der Start-Suche befindet, existiert im gemeinsamen Speicher ein Flag, das von der Steuerung gesetzt wird und das der Hough-Akkumulator-Analyse als Parameter übergeben wird. Dort wird dann bei gesetztem Flag neben der eigentlichen Akkumulator-Analyse auch nach einer Startlinie gesucht. Zur Bestimmung, ob nun die Objekte über- oder unterhalb einer Geraden als gültig interpretiert werden, ist in der `LINE`-Struktur ein entsprechendes Flag vorhanden, welches bei der Hough-Akkumulator-Analyse gesetzt wird.

Die Methode

```
void CImageProcessing::ExtractValidObjects()
```

stellt nun die oben beschriebene Funktionalität bereit und greift dabei auf die im vorherigen Bearbeitungsschritt extrahierte Objektliste sowie die extrahierten Geradengleichungen zurück.
Zusätzlich zur Lage der Objekte wird hier noch die Größe der Objekte überprüft, welche eine bestimmten Wert nicht unterschreiten sollte.

Das Ergebnis sind dann eine Menge von auf dem Teppich liegenden Objekten, welche im gemeinsamen Speicher abgelegt werden.

13.8 Erkennung der Ablage

Letzte Aufgabe der Bildverarbeitung ist die Erkennung der Markierung an der Ablage.

Ich hatte mich ja dazu entschlossen, die bereits vorhandene Funktionalität, wie sie für die Erkennung einer rot-gelb-rot markierten Flasche entwickelt wurde, zu verwenden. Nur, dass die Farben jetzt grün-gelb-grün sind.
Diese ist in der Funktion

```
int CImageProcessing::LookForTheBox( int iBorderWidth, int
iBlackThreshold, int iYellowThreshold, CImage *pImageIn, CImage
*pImageOut )
```

untergebracht.
Eingabe ist wieder das Originalbild und Ausgabe ein 24-Bit-RGB-Farbbild, in dem die Bildpunkte, die einer der Markierungsfarben zugeordnet sind, für eine bessere Visualisierung besonders dargestellt werden. Neben der Rahmendicke werden zusätzlich noch Schwellwerte für die Farbextraktion als Parameter übergeben.
Der Ablauf ist Folgender:
Die Bildpunkte des Eingabebildes werden zunächst klassifiziert in grüne, gelbe und unwichtige Bildpunkte. Anschließend werden gleichfarbige Bildpunkte mit den Methoden, wie sie bei der Spielzeugerkennung eingesetzt wurden, zu Regionen zusammengefaßt.
In diesen extrahierten Regionen wird nun nach der Markierung gesucht. Dabei wird als Erstes eine grüne Region gewählt. Dazu wird eine gelbe Region gesucht, die ungefähr dieselben Ausmaße wie die grüne besitzt, aber zentriert direkt unter ihr liegt und anschließend wieder eine grüne Region, die wiederum zentriert direkt unter der gelben liegt.
Wurde eine entsprechende Kombination gefunden, dann werden alle Regionen zu einer einzigen zusammengefaßt und abgespeichert. Hierbei wird auch wieder auf eine Mindestgröße geachtet. Sollten mehrere solcher Kombinationen entdeckt werden, dann wird nur die mit den größten Ausmaßen weiterverwendet.

Beim Testen dieser Funktionalität gab es aber ein Problem. Der Laborfußboden, der eigentlich eher eine blau-graue Farbe hat, wird zum Teil als grün interpretiert. Dies liegt hauptsächlich an den Grabfunktionen, die ja nicht echte 24 Bit sondern nur 15 Bit-Bilder liefern. Auch eine entsprechende Veränderung des Schwellwertes brachte keinen richtigen Erfolg. Abhilfe schafft hier nur eine

Veränderung der Farbkombination. Ich verwende daher eine schwarz-gelb-schwarze Kombination.
Da die Laborumgebung in Richtung der Kiste nicht sehr viele schwarze Objekt hat und durch die gute Beleuchtung auch kaum störende Schatten vorhanden sind, kann hier diese Kombination gewählt werden. Die Funktionalität der Farbmustererkennung kann aber sehr leicht wieder auf andere Farben umgestellt werden, wenn die Applikation einmal an einem Standort mit beispielsweise schwarzem Untergrund zum Einsatz kommen soll. Für den vorgesehenen Einsatzort ist sie im Moment aber ausreichend.

13.9 Zeichenroutinen

Zusätzlich zu den eigentlichen Verarbeitungsfunktionen habe ich noch einige Routinen zum Einzeichnen der Objekte in ein Bild entwickelt:

```
int CImageProcessing::DrawValidObjects( CImage *pImage )
int CImageProcessing::DrawLines( CImage *pImage )
int CImageProcessing::DrawBox( CImage *pImage )
```

13.10 Der gemeinsame Speicher

Die Ergebnisse der Verarbeitungsschritte wurden im gemeinsamen Speicher abgelegt. Diese Objekte sind:

- ein Feld von `REGINFO`-Strukturen für die gefundenen Spielzeugobjekte
- ein Feld von `LINE`-Strukturen für alle gefundenen Linien
- eine `REGINFO`-Struktur, die die Markierung der Ablage darstellt

Anstatt bei den Spielzeug-Objekten nur die Schwerpunktkoordinaten abzulegen, werden hier die entsprechenden Strukturen gespeichert. Dies führt dann zu mehr Übersichtlichkeit und hat auch noch den Vorteil, dass zusätzliche Informationen bei der Steuerung berücksichtigt werden können.
Zum Zugriff auf die abgelegten Objekte sind in den entsprechenden Dateien Funktionen enthalten, die die Objekte mit Mutex-Variablen vor gleichzeitigem Zugriff

schützen, um z.B. zu verhindern, dass Daten gelesen werden, obwohl sie noch nicht zu Ende geschrieben wurden. Diese Funktionen werden beim Setzen durch die Bildverarbeitung sowie beim Holen der Werte von der Saphira-Bibliothek verwendet.

13.11 Die Saphira-Bibliothek

Die Funktionen, die für Saphira zur Verfügung stehen sollen, werden in der Datei `bvdll.c` definiert. Die Bezeichnung der Bibliothek ist im zugehörigen Projekt definiert und lautet `pioneerbv.dll`.
In der Datei sind dann auch die Funktionen `sfLoadInit` und `sfLoadExit` enthalten, wobei Erstere mit `sfAddEvalFn` und `sfAddEvalStruct` das eigentliche „Bekanntmachen“ der Funktionen und Strukturen erledigt. Eine Funktion, die für Saphira zur Verfügung stehen soll, ist z.B. das nächstgelegene Objekt zurückzuliefern. Dies wird dadurch erledigt, dass in der Objektliste nach dem Objekt, das am nächsten zur Bildunterkante liegt, gesucht wird.
Weitere Einzelheiten zu den Funktionen der Bibliothek und deren Benutzung folgen im nächsten Teil, welcher die Implementierung der Steuerung zum Thema hat.

13.12 Erweiterungen

13.12.1 Verwendung der IPL

Im Abschnitt über die Laborumgebung wurde bereits kurz die Intel® Image Processing Library vorgestellt. Leider bin ich erst auf sie gestoßen, als die Bildverarbeitung schon ziemlich fertig war. Dies ist aber nicht weiter tragisch, da die meisten Funktionalitäten, so wie ich sie brauchte, dort auch nicht enthalten sind, z.B. ein auf einem Bild anwendbarer Differenzoperator zweier Kanäle, ohne dabei Bilddaten kopieren zu müssen.
Eine Funktion ist dort aber vorhanden, die für meine Aufgabe sehr nützlich ist, und zwar die morphologische Closing-Operation. Wie schon erwähnt, kann sie sehr gut zur Verbindung lückenhafter Konturen eingesetzt werden. Allerdings ist sie auch, speziell wenn man mehrere Iterationsschritte verwenden will, sehr rechenaufwendig. Ich hatte mich daher anfangs gegen sie entschieden und stattdessen einen Maximumoperator verwendet. Jetzt aber besteht die Möglichkeit, mit der für die

Intel®-Architektur optimierten Funktion `iplClose` aus dieser Bibliothek, diese doch noch verwenden zu können. Allerdings arbeitet die Bibliothek wieder mit ihrer eigenen Bilddefinition, welche in der Struktur `IplImage` definiert ist. Es gibt jedoch die Bibliotheks-Funktion `iplTranslateDIB`. Dieser kann man einen Zeiger auf ein `BITMAP` (wie es z.B. vom Framegrabber geliefert wird) übergeben und sie erzeugt dann ein `IplImage`, das einen Zeiger auf dieselben Bilddaten besitzt. Ein Kopieren dieser Daten ist somit nicht notwendig.
Ich habe die Closing-Operation nach dem Farbkantenfilter der Objekterkennung anstelle des Maximumoperators eingefügt. Die Funktion soll dabei 5-Iterationsschritte ausführen. Dies funktioniert recht schnell und die Objektkonturen sind anschließend fast komplett geschlossen.

13.12.2 Linien löschen

Ein Problem bei der Objekterkennung tritt an den Teppichlinien auf. Wie gesagt, wird bei der Objekterkennung mit einem Farbkantenfilter gearbeitet. Dies bedeutet, dass natürlich die Linien an sich auch als Objekte erkannt werden, wobei dies auf Grund der Prüfung mit den Geradengleichungen noch nicht das größte Problem ist. Problematischer ist hingegen, wenn ein Objekt sich vom Roboter aus direkt vor einer Teppichkante befindet. Auf Grund der Konturverknüpfung wird es dann mit der Linie zu einem Gesamt-Objekt verbunden und fällt dann bei den Prüfungen weg. Man kann dieses Problem allerdings lösen, indem man die Linien vor der Objektextraktion aus dem Bild löscht. Dies habe ich in der Funktion

```
int CImageProcessing::DeleteLines( CImage *pImage, int iWidth )
```

realisiert.
Dort werden für jede Gerade ihre Punkte im Bild berechnet und von jedem dieser Punkte aus `iWidth` Bildpunkte nach oben und nach unten aus dem Bild entfernt (Der Einfachheit halber nach oben und unten anstelle von der Geraden aus im korrekten Winkel nach rechts und links). Um noch korrekt erkannt zu werden, sollten Objekte aber trotzdem noch einen Abstand von min. 10 cm zu den Rändern besitzen.

13.13 Ein Beispiel

Nachdem jetzt alle Bildverarbeitungsmethoden vorgestellt worden sind, soll ihre Funktionsweise an einem konkreten Beispiel einmal verdeutlicht werden. Ausgangspunkt hierfür ist das in Abbildung 23 zu sehende Bild, das mit der Roboterkamera aufgenommen wurde.

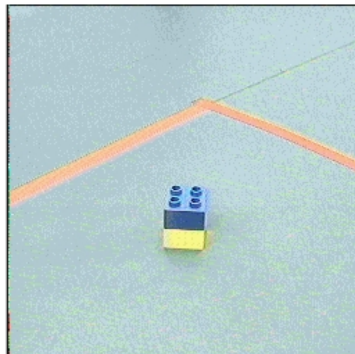

Abbildung 23: Das Ursprungsbild

Als Erstes werden die Methoden der Linienerkennung vorgestellt. Die Algorithmenkette war dort:

- Filterung nach der Farbe *ROT*
- Kantenerkennung
- Hough-Transformation
- Hough-Akkumulatoranalyse

Da in meiner Implementierung der Hough-Akkumulator nicht visualisiert werden kann, können nur Beispielbilder für die ersten beiden Funktionen gezeigt werden. Auf das Ursprungsbild wird zunächst eine Filterung nach der Farbe *ROT* und auf das dabei entstehende Ergebnisbild der Kantenfilter angewendet. Die Ausgabebilder beider Operationen sind in Abbildung 24 auf der nächsten Seite zu sehen.

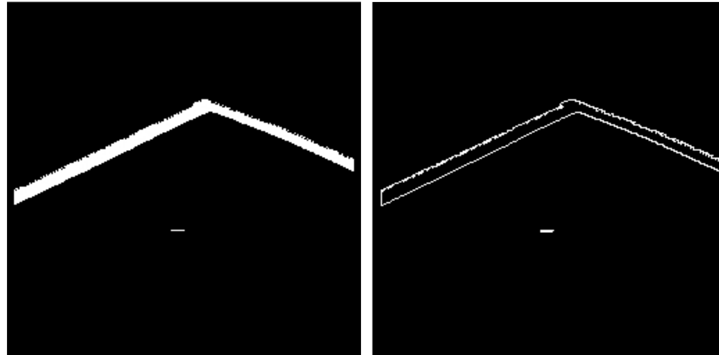

Abbildung 24: Linienerkennung (links: Ausgabebild Rot-Filter; rechts: Ausgabebild Kantenfilter)

Mit den im kantenextrahierten Bild erhaltenen Kantenpixeln wird eine Hough-Transformation durchgeführt und anschließend der Akkumulator analysiert.

Nächster Punkt ist die Spielzeugerkennung:
Der Ablauf hier ist:

- Farbkantenfilter
- Konturen verbinden
- Linien löschen
- Objekte extrahieren
- Gültigkeit feststellen

Die Basis stellt wieder das Ursprungsbild dar, auf das jetzt jedoch zunächst ein Farbkantenfilter angewendet wird. Dessen Ausgabe ist in Abbildung 25 auf der nächsten Seite sichtbar.

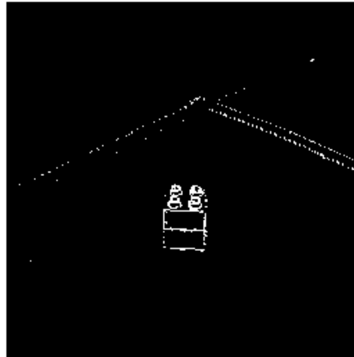

Abbildung 25: Farbkantenfilter auf das Ursprungsbild angewendet

Auf dieses Bild wird nun die morphologische closing-Operation mit 5 Iterationsschritten angewendet und anschließend noch die Linien unter Zuhilfenahme der Geradengleichungen aus dem Bild gelöscht. Die Ergebnisse sind in Abbildung 26 zu sehen.

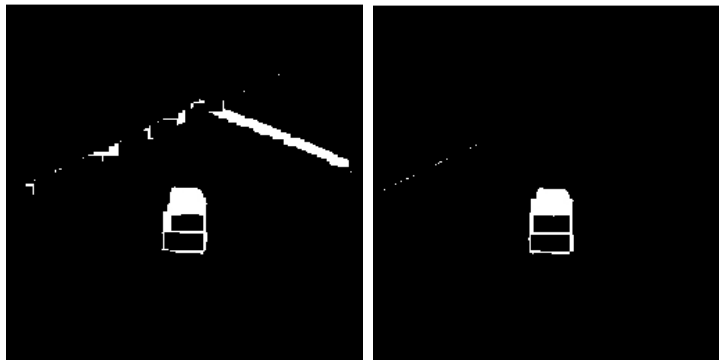

Abbildung 26: Objekterkennung (links: nach der closing-Operation; rechts: nach dem Linien löschen

Die eigentliche Extraktion der Objekte über eine Nachbarschaftsbeziehung sowie das Feststellen der Gültigkeit kann wiederum nicht visualisiert werden.
Abbildung 27 auf der nächsten Seite zeigt aber das Ergebnisbild, in das die extrahierten Linien (grün) und die als gültig erkannten Objekte (rot umrandet) eingezeichnet wurden.

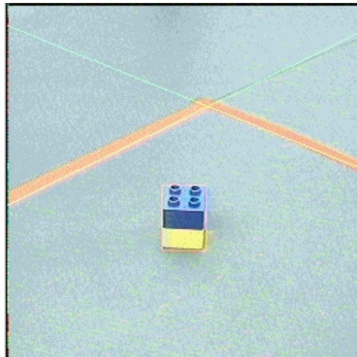

Abbildung 27: Das Ergebnisbild

14 Implementierung der Robotersteuerungskomponente

14.1 Einleitung

Im letzten Teil der Arbeit soll jetzt die konkrete Realisierung der Steuerungskomponente, die auf Basis von *Colbert-Activities* erfolgen soll, beschrieben werden.
Der Aufbau der Steuerung, das Zielverhalten sowie die sich dabei ergebenden Activities und deren Aufgaben, wurden im Abschnitt „Konzeption der Robotersteuerungskomponente“ besprochen.
In Anlehnung an die funktionalen Schritte möchte ich auch diesen Teil zunächst wieder in die drei Gebiete: *Objekt suchen*, *Objekt aufnehmen* und *Objekt ablegen* unterteilen. Im Anschluß daran wird auf das Haupt-Activity, das sämtliche Schritte koordinieren soll, eingegangen.

14.2 Objekt suchen

Den Anfang macht die Objektsuche. Hierzu wurde bereits festgestellt, dass es dabei im Wesentlichen um die Steuerung des Roboters und der Kamera geht, da der Rest von der Bildverarbeitung erledigt wird.
Es ist hierbei aber zu beachten, dass für die drei Suchgebiete auch unterschiedliche Bewegungen von Kamera und Roboter notwendig sind. Zunächst soll deshalb die Charakteristik der einzelnen Suchmethoden näher beschrieben werden.

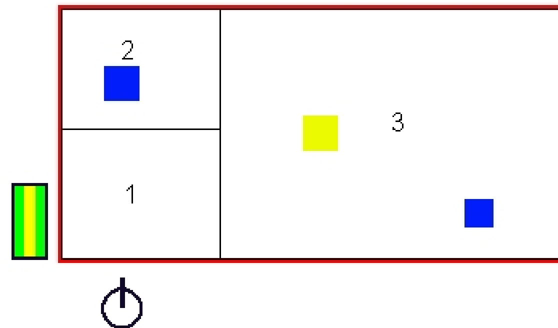

Abbildung 28: Die Suchgebiete

Suche in Sektor 1 Vorraussetzung für die erste Suche ist, dass sich der Roboter an seiner Ausgangsposition befindet. Er sollte dabei senkrecht zur Teppichlinie stehen und für die Kamera, wenn sie ganz nach unten geschwenkt ist, diese Linie im unteren Teil des Sichtfeldes erkennbar sein. Um jetzt den ersten Sektor von dieser Position aus komplett abzusuchen, wird wie folgt vorgegangen:
Die Kamera schwenkt als Erstes soweit nach links, bis die Teppichecke gefunden wird. Der Winkel der Kamera wird abgespeichert. Anschließend wird die Kamera von dieser Position aus in Teilschritten bis zu einem festgelegten Winkel nach rechts geschwenkt (nach jedem Schritt wird dabei festgestellt, ob sich im Kamerabild ein gültiges Objekt befindet). Danach fährt sie wieder auf den gespeicherten Winkel nach links zurück, bewegt sich um einen festgelegten Wert nach oben und startet erneut einen Rechtsschwenk. Wurde bei einem Schwenk ein gültiges Objekt gefunden, dann wird die Suche beendet. Der Roboter bewegt sich während der Suche nicht.

Suche in Sektor 2 und 3 Vorraussetzung für beides ist, dass sich der Roboter an der jeweiligen Ausgangsposition befindet und in Suchrichtung blickt. Startpunkt für die Suche in Sektor 2 ist ein Punkt vor der Kiste und für die Suche in Sektor 3 ein Punkt auf selber Höhe, der aber ungefähr in Teppichmitte liegt. Der Pan-Winkel der Kamera ist dabei 0.
Zur Suche schwenkt die Kamera zunächst wieder in Teilschritten bis zu einem festgelegten Wert nach links und anschließend nach rechts. Von den Ausgangsposi-

tionen wird dieses Verhalten als Erstes ohne Roboterbewegung gestartet.
Ist die Suche hierbei erfolglos, dann werden die Kamerabewegungen nach der eben beschriebenen Art fortgesetzt und zusätzlich der Roboter vorwärts bewegt. Die Suche beendet sich dann, wenn ein Objekt gefunden oder die jeweilige Begrenzungslinie entdeckt wurde. Sollte kein Objekt gefunden worden sein, dann kann vom letzten Punkt aus nocheinmal eine Suche ohne weitere Roboterbewegung erfolgen.

Für die eigentlichen Objekt-Suchphasen in allen Sektoren habe ich das Activity

```
act search( int iInitialSearch, int iMove, int iMinPan, int
iMaxPan, int iPanStep )
```

entworfen, welches die Roboter- und Kamerabewegungen steuert und nach jedem Kamerabewegungsschritt überprüft, ob ein Objekt im Bild vorhanden ist. Diesem können die gewünschten Eigenschaften der Bewegungen als Parameter übergeben werden.
Es beendet sich, sobald ein gültiges Objekt gefunden wurde, oder, wenn sich der Roboter beim Suchen bewegt hatte, eine Begrenzungslinie entdeckt wurde. Das aufrufende Activity muss nach Beendigung daher nur noch den Status der Suche überprüfen und kann dann z.B. das Greif-Activity starten. Um den Status des Activities zu ermitteln, kann die Saphira-Funktion `sfGetTaskState` benutzt werden. Diese funktioniert jedoch nicht sehr zuverlässig, sodass anstelle dessen ein globales Flag gesetzt wird, wenn ein Objekt gefunden wurde.
Die Parameter des Activities haben die folgenden Bedeutungen:

- `iInitialSearch`
  1 - Kamerabewegung vom momentanen Kameraschwenkwinkel (Pan) nur nach rechts
  0 - Kamera von Schwenkwinkel 0 abwechselnd nach links und rechts bewegen

- `iMove`
  1 - Vorwärtsbewegung des Roboters und kontinuierliches Rechts/Links-Schwenken der Kamera
  0 - Roboter bewegt sich nicht und es wird nur ein Schwenk ausgeführt

- `iMinPan,iMaxPan` - Minimaler/Maximaler Schwenkwinkel der Kamera
- `iPanStep` - Schrittweite für Kamerabewegungen

Neben den Kamera/Roboterbewegungen findet im Suchactivity auch die eigentliche Suche nach Objekten im Bild statt. Bis jetzt wurde dazu nur gesagt, dass die gefundenen Objekte von der Bildverarbeitung im gemeinsamen Speicher abgelegt und über eine selbstentwickelte Bibliothek (`pioneerbv.dll`) für Saphira bekannt gemacht werden. Ich habe mich aber dafür entschieden, die gefundenen Objekte nicht direkt für Saphira verfügbar zu machen, sondern anstelle dessen in der Bibliothek Funktionen zur Auswertung dieser Daten zur Verfügung zu stellen. Eine davon ist die Funktion

```
BOOL GetClosestObject(REGINFO *region, BOOL bCheckTestedRegions),
```

welche aus sämtlichen gefundenen gültigen Objekten das heraussucht, das den kleinsten Abstand zur Bildunterkante besitzt. Dieser Funktion übergibt man einen Zeiger auf eine `REGINFO`-Struktur, die mit den Daten des gefundenen Objektes, das am nächsten zur Bildunterkante liegt, gefüllt wird. Als Rückgabewert liefert sie ein Flag, das anzeigt, ob eine Objekt gefunden wurde.
Im Suchactivity kann diese Funktion nun einfach aufgerufen werden und über den Rückgabewert überprüft werden, ob sich im Sichtfeld ein gültiges Objekt befindet. Allerdings existiert hier noch ein Problem. Die Gültigkeitsüberprüfung von Objekten wird nur von der Bilderverarbeitung unter Berücksichtigung aller sichtbarer Linien erledigt. Es kann hierbei aber vorkommen, dass speziell in den äußeren Bildregionen Objekte gefunden werden, die eigentlich keine sind bzw. die nicht auf dem Teppich liegen. Als Beispiel seien hier Teile von Begrenzungslinien erwähnt, die zwar so kurz sind, dass sie noch nicht als Linie erkannt werden, allerdings so groß sind, dass sie als Objekte interpretiert werden.
Die Lösung dazu ist, dass der Roboter, nachdem im Bild ein Objekt erkannt wurde „etwas genauer hinschaut“. Dies habe ich im Activity

```
act tryCenterObject( int x, int y )
```

realisiert, dem der Schwerpunkt des zu prüfenden Objektes übergeben wird.
In diesem Activity wird die Kamera so auf die Schwerpunktkoordinaten ausgerichtet, dass sich das Objekt anschließend ungefähr in der Mitte des Bildes befinden

sollte. Dadurch kann gewährleistet werden, dass die Umgebung des vermeintlichen Objektes und vor allem die sich dort möglicherweise befindenden Begrenzungslinien korrekt erkannt werden können. Das Ausrichten der Kamera erfolgt, indem der Abstand des Schwerpunktes zum Bildmittelpunkt (jeweils in X- und Y-Richtung) bestimmt und dieser dann mit einem geeigneten Faktor multipliziert als Steuerungswinkel für die Kamera verwendet wird. Dies ist zwar kein absolut exaktes Verfahren, da die Entfernung zum Objekt sowie die Bewegung der Kamera auf der schrägen Konstruktion des Pioneer „Alfa“ nicht berücksichtigt werden, aber für die Bedürfnisse vollkommen ausreichend.
Der Roboter darf sich bei der Überprüfung eines Objektes natürlich nicht bewegen. Nachdem die Kamera also so ausgerichtet wurde, dass das Objekt ungefähr in der Bildmitte liegen sollte, wird die Bibliotheksfunktion

```
BOOL GetClosestObject2Point( int x, int y, int iMaxDistance,
REGINFO *region )
```

aufgerufen, die das nächstgelegene Objekt zum Punkt $P(x, y)$, welches nicht weiter als `iMaxDistance`-Pixel davon entfernt liegen darf, liefert.
Durch den Aufruf der Funktion mit den Koordinaten des Bildmittelpunktes kann festgestellt werden, ob sich ein Objekt in der Nähe dessen befindet (`iMaxDistance` gibt hierbei die Toleranz an). Ist dies nicht so, dann war das vorher entdeckte Objekt mit hoher Wahrscheinlichkeit nicht gültig, und die Kamera wird im Activity `tryCenterObject` wieder auf die ursprünglichen Einstellungen zurückgesetzt. Wurde jedoch ein Objekt in der Nähe des Mittelpunktes gefunden, dann wird ein Flag gesetzt, das im Suchactivity überprüft wird. Zusätzlich wird vom Activity noch der Neige(Tilt)-Winkel der Kamera überwacht. Wurde beim Versuch des Zentrierens der kleinstmögliche Tilt-Winkel erreicht, d.h. also wenn die Kamera nicht tiefer schwenken kann, dann wird überprüft, ob sich ein Objekt in der unteren Bildhälfte befindet. Ist dies der Fall, dann wird dieses Flag ebenfalls gesetzt.

Allerdings sind auch damit noch nicht alle Probleme gelöst. Es kann nämlich durchaus vorkommen, dass im Bild mehrere Objekte vorhanden sind, die überprüft werden müssen. Sucht man sich dabei nur das nächstgelegene aus und versucht, es mit `tryCenterObject` zu überprüfen, kann der Fall auftreten, dass dieses als

„ungültig“ bestimmt und anschließend die Suche mit der nächsten Kamerastellung fortgesetzt wird, ohne die anderen Objekte im Bild zu beachten.
Um auch dieses zu lösen, wird in der Bibliothek (`pioneerbv.dll`) eine Liste mit den zurückgelieferten Objektpositionen mitgeführt, die nach jedem Aufruf der Funktion `GetClosestObject` aktualisiert wird. Die Funktion übernimmt dazu noch den Parameter `bCheckTestedRegions`. Hat dieser den Wert `false`, dann wird die Liste zurückgesetzt und nur das nächstgelegene Objekt geliefert. Anderenfalls wird das Objekt zurückgegeben, das am nächsten zur Unterkante liegt und einen Mindestabstand zu allen bereits gelieferten Objekten besitzt (um zu verhindern, dass dasselbe Objekt nochmal geliefert wird). Sollte kein entsprechendes Objekt gefunden werden, dann gibt die Funktion `false` zurück.

Um das Suchactivity `search` übersichtlicher zu gestalten, habe ich für das Durchsuchen eines Bildes das Activity

```
act searchInImage( int iMove )
```

entworfen, das die eben beschriebenen Funktionalitäten enthält. Der Parameter `iMove` gibt hierbei an, ob sich der Roboter beim Suchen bewegt. Dies ist wichtig, damit der Roboter vor dem „genauer hinschauen“ angehalten wird.
Es werden dabei nacheinander alle möglichen Objekte im Bild mit `tryCenterObject` untersucht, bis entweder ein gültiges Objekt gefunden wurde, oder alle bereits überprüft wurden. Das aufrufende Activity `search` testet dann nur noch ein Flag ab und braucht ansonsten nur noch die Kameraschwenks und das Vorwärtsbewegen des Roboters steuern. Das Ergebnis der Überprüfung mit `searchInImage` ist, dass entweder kein gültiges Objekt gefunden wurde, und sich die Kamera wieder in ihrer Ausgangslage befindet, oder, dass ein Objekt gefunden wurde und dieses sich ungefähr im Mittelpunkt des Bildes befindet.

14.3 Objekt aufnehmen

Nachdem ein Objekt von der Suche als gültig bestimmt wurde, soll es vom Roboter aufgenommen werden. Hierzu dient das Activity

```
act takeObject().
```

Ausgangsposition hierfür ist das vom Such-Activity im Kamerabild zentrierte Objekt.
Als Erstes wird vom Activity die momentane Roboterposition gespeichert, um, falls ein Fehler auftreten sollte, wieder dorthin zurückkehren und die Suche normal fortsetzen zu können.
Die Kamera hat das Objekt immernoch ungefähr in der Bildmitte und hat eine bestimmten Schwenk(Pan)-Winkel zum Roboter. Um als Nächstes den Roboter in Richtung des Objektes auszurichten, wird er um diesen Winkel gedreht und anschließend die Kamera wieder auf den Pan-Winkel 0 bewegt. Das Objekt befindet sich danach wieder ungefähr in Bildmitte und der Roboter ist darauf ausgerichtet. Anschließend wird, soweit dies noch nicht geschehen ist, der Greifer geöffnet und nach unten bewegt.
Jetzt kann der Roboter auf das Objekt zufahren. Hierzu habe ich das bei der Konzeption beschriebene Verfahren mit kleinen Änderungen übernommen. Grundsätzlich ist dieses dadurch charakterisiert, dass der Roboter auf das Objekt zufährt und dabei versucht, es im Bildmittelpunkt zu halten. Bewegt es sich aus dem Bild nach unten heraus, wird noch solange weiter vorwärts gefahren, bis es im Greifer von den dort eingebauten Lichtschranken-Sensoren wahrgenommen wird.
Der genaue Ablauf darin ist wie folgt:
Zunächst werden die Koordinaten des nächstgelegenen Objektes über eine Bibliotheksfunktion geholt. Wird eine Objekt gefunden, dann wird der Abstand zum Bildmittelpunkt in X- und Y-Richtung bestimmt. Diese Werte dienen zur Steuerung. Um das Objekt in vertikaler Richtung zentriert zu halten, wird der Neigungs(Tilt)-Winkel der Kamera angepaßt. Um es horizontal zu zentrieren, führt der Roboter eine Rechts- bzw. Linksbewegung aus. Zusätzlich bewegt er sich mit Hilfe des Befehles `speed` vorwärts.
Zur Steuerung der Roboterbewegung nach rechts und links können dabei zwei verschiedene Befehle verwendet werden. Der Erste ist das Kommando `turn`, das den Roboter um einen absoluten Winkel dreht. Der Zweite ist der Befehl `rotate`, der eine kontinuierliche Drehbewegung des Roboters mit einer bestimmten Drehgeschwindigkeit veranlasst. Der Vorteil von `rotate` ist, dass die Bewegungen des Roboters weicher aussehen. Allerdings gab es bei Tests mit diesem Kommando das Problem, dass es sich nach Aktivierung nicht wieder stoppen, sondern den

Roboter immer weiter drehen lässt. Dieser Effekt scheint dabei vor allem in Verbindung mit dem `speed`-Kommando zu existieren, d.h. also bei gleichzeitiger Vor- und Seitwärtsbewegung. Abhilfe schaffte hierzu aber ein Eintrag in der Pioneer-Newsgroup auf der ActivMedia-Homepage, der empfahl, nach Beenden von `rotate` noch einen `turn(0)`-Befehl einzufügen.
In meiner jetzigen Implementierung wird also, nachdem dieses Problem gelöst wurde, der `rotate`-Befehle zur Steuerung verwendet.

Die Vorwärtsbewegung wird gestoppt, sobald kein Objekt mehr sichtbar ist und die letzte Position an der Bildunterkante war, oder wenn die Kamera bereits ganz nach unten geschwenkt ist und das Objekt sich an der Bildunterkante befindet. Das Activity schlägt fehl, wenn auf mehreren Bildern hintereinander kein Objekt gefunden wurde und nicht die Erste der eben genannten Bedingungen zutrifft.

Wenn kein Fehler aufgetreten ist, dann sollte sich das Objekt jetzt mit einem geringen Abstand direkt vor dem Roboter befinden. Wenn sich der Roboter jetzt also in seiner momentanen Ausrichtung weiter vorwärts bewegt, dann müssten die Lichtschranken des Greifers irgendwann ein Objekt melden. Wie allerdings Tests dieser Sensoren aufgezeigt haben, liefern sie bei komplett geöffnetem Greifer keine zuverlässigen Daten. Um die Zuverlässigkeit der Meßergebnisse zu verbessern, können die Greifer jedoch etwas geschlossen werden. Da die Objekte (LEGO-Duplo-Bausteine) recht klein sind und auch die Entfernung zu ihnen nicht allzu groß sein sollte, kann ein solches Vorgehen verwendet werden, da somit die bei der Bewegung oder beim vorherigen auf-das-Objekt-zufahren auftretenden Ungenauigkeiten nicht allzu sehr stören und die Objekte trotzdem noch zwischen den Greiferhänden landen sollten.

Im Activity `takeObject` werden also zunächst die Greifer etwas geschlossen und anschließend fährt der Roboter in seine momentane Richtung weiter. Er hält an, wenn sich das Objekt im Greifer befindet oder er eine bestimmte Wegstrecke (im Fehlerfall) zurückgelegt hat. Für Letzteres kommt das von mir erstellte Activity

```
act watchDist( point *pStartPos, float fMaxDist )
```

zum Einsatz, das auch an anderen Stellen noch verwendet wird.
Ihm wird ein Startpunkt und ein Wert für die Distanz übergeben. Wenn der Abstand des Roboters vom Startpunkt zu seiner augenblicklichen Position, welchen man mit dem Saphira-Befehl `sfPointDist` feststellen kann, größer ist, als der übergebene Wert, dann wird ein globales Flag gesetzt.
Für das Zufahren auf das Objekt wird es folgendermaßen genutzt:
Vor Start der Bewegung wird `watchDist` mit der momentanen Position und einer maximalen Distanz von 30cm nicht-blockierend gestartet. Der Roboter soll sich dann solange bewegen, bis die Lichtschranken die gewünschten Werte liefern (z.B. Objekt in äußerer Lichtschranke). Zusätzlich zu diesen Werten wird auch noch das Flag, das `watchDist` beschreibt, abgeprüft und ggf. die Ausführung von `takeObject` beendet.

Im Normalfall sollten die Lichtschranken aber rechtzeitig ein Objekt erkennen. Wenn dies der Fall ist, dann wird der Roboter gestoppt, der Greifer geschlossen und nach oben bewegt. Danach fährt der Roboter wieder ein kleines Stück (20cm) zurück. Dies ist nützlich, um bei mehreren nebeneinanderliegenden Objekten zu verhindern, dass der Roboter bei anschließenden Drehbewegungen andere Objekte berührt bzw. umherschiebt.

Das Ergebnis des Activities ist entweder, dass ein Objekt gefunden und angehoben wurde, oder, dass sich der Roboter ungefähr wieder an der Ausgangsposition befindet. Zweiteres wird dadurch realisiert, dass der Roboter im Fehlerfall zum anfangs gespeicherten Punkt zurückfährt und die Kamera wieder an ihre ursprüngliche Ausrichtung bewegt.
Der Status des Activities (Objekt gegriffen j/n) wird in einem globalen Flag gespeichert.

14.4 Objekt ablegen

Der nächste Punkt ist das Ablegen des Objektes in der Kiste. Im Wesentlichen kann dieser Vorgang in die zwei Phasen *„Zum Punkt vor der Kiste fahren“* und *„Objekt in der Kiste ablegen“* unterteilt werden. Für die erste Phase wurde bereits

festgelegt, dass dazu zwei verschiedene Activities benötigt werden. Je nach Position des Roboters auf dem Teppich sind dies einmal eine odometrie- und einmal eine bildverarbeitungsgesteuerte Lösung. Ich habe daher drei Activities entworfen, die die einzelnen Schritte ausführen sollen und nachfolgend beschrieben werden.

14.4.1 Odometriegesteuerte Navigation zur Ablage

Die odometriebasierte Lösung ist im Activity

```
act goToTheBoxA()
```

realisiert.
Vorraussetzung für dieses Verhalten ist, dass eine Position vor der Ablage als Saphira-Punkt-Artefakt existiert. Das Anlegen dieses Artefaktes erfolgt zu Beginn des Gesamt-Verhaltens von der Startposition aus. Der Roboter steht dort zur Startlinie ausgerichtet. Da die Kiste in derselben Ecke untergebracht ist, kann ein Punkt erstellt werden, der sich in einem bestimmten Abstand von der Position aus auf dem Teppich befindet.
Nachdem ein Objekt also aufgenommen wurde, braucht der Roboter dann nur noch zu dieser Position zu fahren.
Zur Navigation zu einem Punkt habe ich ein eigenes Activtiy

```
act goToPos( point *p, int iTurn )
```

anstelle des saphira-eigenen `sfGoToPos` verwendet. Dieses Activity dreht den Roboter zunächst in Richtung des Zielpunktes. Anschließend wird darin das Saphira-Behavior `sfFollow` gestartet, das einer Linie vom Startpunkt zum Zielpunkt folgt, bis es an letzerem angekommen ist. Über den Parameter `iTurn` kann noch angegeben werden, ob sich der Roboter nach der Ankunft am Ziel noch auf den Winkel des Zielpunktes drehen soll.
Nach Start von `goToTheBoxA` fährt der Roboter zunächst also mit `goToPos` zum gespeicherten Punkt vor der Ablage. Auch wenn dieses Activity nur für kleinere Entfernungen gedacht ist, treten trotzdem Navigationsungenauigkeiten auf. Nachdem der Roboter also an dem Punkt vor der Ablage angekommen ist, muss er sich noch ausrichten, damit er das Objekt auch an der richtigen Stelle ablegt. Als Orientierung können hier die beiden Teppichbegrenzungslinien in der unteren Ecke

dienen. Um eine geeignete Ausrichtung vorzunehmen, habe ich verschiedene Tests durchgeführt. Dabei hat sich herausgestellt, dass wenn der Roboter bei ganz nach unten geschwenkter Kamera auf jede dieser beiden Linien soweit zufährt, dass sie gerade aus dem Blickfeld verschwunden ist, er eine ideale Position in der unteren Ecke besitzt.
Um dieses Ausrichten in der Ecke zu realisieren, habe ich die beiden Acitivities

```
act moveToLine( int iSpeedBW, int iSpeedFW ) und
act centerRobotToLine( int iMaxDiff )
```

geschrieben.
`centerRobotToLine` übernimmt dabei ein Ausrichten zu einer Linie. In diesem Activity wird über die Bibliotheksfunktion

```
BOOL GetHLine( int iMaxAngleDiff, LINE *line )
```

nach einer ungefähr (je nach Parameter `iMaxAngleDiff`) horizontal im Bild liegenden Linie gesucht. Der Roboter dreht sich dann solange, bis diese Linie horizontal im Bild liegt. Der Parameter `iMaxDiff` gibt hierbei die Toleranz für `centerRobotToLine` an.
Das Activity `moveToLine` bewegt den Roboter nach Aktivierung zunächst solange rückwärts, bis eine Linie gefunden wurde. Dann startet es `centerRobotToLine`, um den Roboter zur Linie auszurichten und anschließend bewegt es den Roboter solange vorwärts, bis die Linie nach unten aus dem Bild verschwunden ist. Wird `moveToLine` für jede der beiden Linien aktiviert, dann befindet sich der Roboter an der gewünschten Position.

Die Navigation zum Startpunkt im Activity `goToTheBoxA` wird in der folgenden Reihenfolge abgearbeitet:
Als Erstes wird das Activity `goToPos` mit dem gespeicherten Punkt gestartet. Wenn der Roboter dort angekommen ist, dreht er sich in Richtung Startlinie und richtet sich mit `moveToLine` zu ihr aus. Anschließend dreht er sich um -90 Grad und steht ungefähr senkrecht zur Linie vor der Ablage. Den Rest erledigt dann das Objekt-ablegen-Activity.

14.4.2 Bildgesteuerte Navigation zur Ablage

Um auch aus größeren Entfernungen noch richtig zum Punkt vor der Ablage navigieren zu können, wird noch eine bildverarbeitungsbasierte Lösung benötigt. Den einfachsten Ansatz, den man sich dazu denken kann, ist ein Verfahren ähnlich dem beim Objekt-greifen eingesetzten. Dies würde dann bedeuten, dass der Roboter auf einer Linie vom Startpunkt des Verhaltens, also dem Ort, wo das Objekt aufgenommen wurde, bis zur Markierung der Ablage fahren würde. Ein solches Vorgehen hat aber den Effekt, dass der Roboter zwar direkt auf die Kiste zufährt, nach Abschluß des Verhaltens aber schräg davor steht. Er muss sich dann noch drehen, was in Fällen, wo er sehr nah an die Kiste herangefahren ist, dazu führen kann, dass die Kiste selbst bei diesen Bewegungen verschoben wird.
Für meine Implementierung kommt daher eine andere Methode zum Einsatz, die auf einem einfachen *P-Regler* basiert. Ziel dieser Methode soll sein, zum Punkt vor der Kiste zu navigieren und am Ende senkrecht zu ihr zu stehen. Die Ausrichtung sollte dabei schon während der Fahrt erfolgen, d.h. der Roboter fährt von seinem Startpunkt aus auf einer gekrümmten Bahn auf die Ablage zu. Zur Regelung werden die Linie, die direkt unter der Markierung der Ablage zu erkennen ist sowie der Schwenkwinkel der Kamera berücksichtigt.
Vorraussetzung ist zunächst ein Verhalten, das versucht, die Markierung der Kiste immer in der Mitte des Bildes zu halten, indem die Kamera in die entsprechenden Richtungen bewegt wird. In den Bildern sollte neben der Markierung dann auch die darunterliegende Linie erkennbar sein.
Die eigentliche Idee der Regelung ist nun Folgende:
Die Linie im Bild besitzt einen bestimmten Winkel. Aus der Differenz dieses Winkels zu 90 Grad (horizontale Linie) kann dann mit einem entsprechenden Faktor `p1` ein *Soll-Wert* ermittelt werden, der den gewünschten Schwenkwinkel der Kamera darstellt (zu jedem Winkel der Linie soll also die Kamera eine bestimmten Winkel zum Roboter haben). Um jetzt diesen Winkel der Kamera zum Roboter zu erreichen, muss der Roboter noch in die jeweilige Richtung gedreht werden. Hierbei kommt anstelle eines `turn`-Befehles wieder das `rotate`-Kommando zum Einsatz. Die Größe der Differenz zwischen Soll- und Ist-Wert des Kamerawinkels gibt dabei, mit einem weiteren Faktor `p2` multipliziert, die Geschwindigkeit der Drehbewegung

an.
Steht der Roboter also schräg zur Kiste, dann ist auch die Differenz des Winkels der Linie im Bild zu 90 Grad relativ hoch, was bedeutet, dass auch der Soll-Wert des Schwenkwinkels der Kamera entsprechend groß ist. Der Roboter dreht sich dann, während die Kamera die Markierung der Ablage weiter in Bildmitte hält, mit einer hohen Geschwindigkeit von der Kiste weg.
Ist die Linie vor der Ablage im Bild dagegen nahezu horizontal, dann ist der Soll-Wert des Schwenk-Winkels annähernd 0, was bedeutet, dass Kamera und Roboter in dieselbe Richtung „blicken".

Das Ergebnis der Regelung ist dann, dass wenn der Roboter zusätzlich noch eine Vorwärtsbewegung ausführt, er in einer bogenförmigen Bahn auf die Kiste zufährt und zum Schluß gerade zu ihr steht. Die Faktoren `p1` und `p2` bestimmen dabei das Aussehen der Bahn, also ob es eher eine linienförmige oder eine „bauchige" Bahn wird.

Für die Implementierung dieser Regelung existieren die beiden Activities:

```
act centerTheBox() und
act goToTheBoxR( float fP1, float fP2, int iMaxVel )
```

Ersteres hat dabei die Funktion, die Markierung der Ablage ständig in der Bildmitte zu halten. Dazu wird dort über die entsprechenden Befehle die Kamera bewegt.
Das zweite Activity übernimmt die eigentlichen Regelungsfunktionen. Die Parameter `fP2` und `fP2` sind dabei die oben angesprochenen Faktoren, die die Bewegung beeinflussen. Mit `iMaxVel` kann zusätzlich noch die maximale Geschwindigkeit, mit der sich der Roboter auf die Kiste zubewegen soll, festgelegt werden. Die genaueste Regelungsfunktion wird dabei erreicht, wenn sich der Roboter nur sehr langsam der Kiste nähert. Um die Fahrt allerdings nicht allzu lange andauern zu lassen, kann dieser Wert etwas höher gewählt werden. Das daraus folgende Ergebnis ist dann aber nicht immer, dass der Roboter auch wirklich exakt senkrecht zur Kiste ankommt. Häufig steht er dann zwar mittig, also direkt vor der Markierung, aber trotzdem noch etwas schräg zu ihr. Dies kann aber dadurch gelöst werden, indem nach Abschluß des Verhaltens noch ein Ausrichten an der Linie erfolgt.
Der Ablauf im Activity sieht folgendermaßen aus:
Als Erstes wird der Roboter in Richtung Kiste gedreht, wobei der vor der Kiste gespeicherte Punkt als Orientierung verwendet wird. Anschließend wird die Kamera in eine Position gebracht, bei der die Kiste im Bild sichtbar sein sollte. Ist dies der Fall, dann setzt die eigentliche Regelung ein. Anderenfalls bewegt sich der Roboter in die Richtung des Startpunktes, bis die Markierung zu sehen oder aber eine Maximal-Distanz erreicht ist.
Wenn die Markierung sichtbar ist, dann wird das Activity `centerTheBox` und die Regelungsfunktion gestartet. Beendet wird das Verhalten, wenn die Markierung nicht mehr sichtbar ist und die Kamera einen bestimmten Schwenkwinkel erreicht hat. Abschließend erfolgt ein Ausrichten an der Linie.

14.4.3 Das eigentliche „Objekt ablegen“

Nachdem der Roboter durch Aktivierung eines der beiden vorgestellten Activities zum Punkt vor der Ablage gefahren ist und sich in die Richtung der Kiste gedreht hat, kann das gegriffene Objekt abgelegt werden. Hierzu dient das Activity

```
act dropObject().
```

Um eine geeignete und für jeden Ablagevorgang ungefähr gleiche Position zu erreichen, wird darin zunächst das Activity `moveToLine` gestartet, das die Linie vor der Kiste sucht, den Roboter zu ihr ausrichtet und anschließend solange auf sie zufährt, bis sie aus dem Bild verschwindet. Danach bewegt sich der Roboter durch den `move`-Befehl einen bestimmten Weg nach vorne. Der Greifer mit dem Objekt sollte sich jetzt direkt über der Kiste befinden und wird geöffnet. Anschließend bewegt sich der Roboter wieder dieselbe Wegstrecke zurück und speichert diese Position wieder ab, um später wieder eine Navigationshilfe zu besitzen.

14.5 Das Haupt-Activity

In den letzten Abschnitten wurden die einzelnen Phasen der Steuerung sowie die sich daraus ergebenden Activities beschrieben. Damit das System seine Aufgabe erfüllen kann, müssen diese Activities noch koordiniert werden. Je nach Suchgebiet sind hierbei unterschiedliche Kombinationen und auch unterschiedliche Parameter der Activities notwendig. Für die drei Suchgebiete existieren daher die Activities

```
act lookAtStartPos()
act lookInSector2() und
act lookInSector3( int iWidth ).
```

Vorraussetzung ist jeweils, dass sich der Roboter an seinem Startpunkt für die entsprechende Suche befindet und korrekt ausgerichtet ist. Die Steuerung in diesen Activities sorgt dann dafür, dass der jeweilige Sektor komplett abgesammelt wird. Die Activities beenden sich erst, wenn dies der Fall ist. Die eigentliche Koordination dieser Verhalten wird von einem Haupt-Activity

```
act mainActivity()
```

erledigt, welches auch als einziges vom Anwender zum Start des Aufsammel-Vorganges aktiviert werden muss.
Der grobe Ablauf darin ist wie folgt:
Der Roboter steht ungefähr auf seiner Startposition. Um ihn richtig auszurichten, wird zunächst das Activity

```
act goToStartPos()
```

gestartet. Dieses sucht die Startlinie, richtet den Roboter zu ihr aus und bewegt ihn bis zu einem bestimmten Abstand zu ihr. Nachdem der Roboter ausgerichtet ist, wird im `mainActivity` ein Punkt erstellt, der in einem bestimmten Abstand vor dem Roboter liegt. Dieser Punkt ist die Position vor der Ablage, die später zur Navigation benutzt wird. Anschließend wird mit `lookAtStartPos` die Suche in Sektor 1 gestartet.
Ist dieser Vorgang beendet, fährt der Roboter auf den Punkt vor der Kiste und beginnt mit `lookInSector2` mit der Suche im zweiten Gebiet. Zum Schluß dieses Verhaltens befindet er sich vor der, der Startlinie gegenüberliegenden, Teppichbegrenzung. Unter Zuhilfenahme des zurückgelegten Weges und des Abstandes zwischen dem Roboter und dieser Linie, kann die ungefähre Breite des Teppiches bestimmt werden. Dieser Wert wird nun dazu benutzt, dass der Roboter zur Mitte des Teppiches fährt, um die richtige Position zur Suche im dritten Suchgebiet einzunehmen. Diese wird dann mit `lookInSector3` gestartet. Wenn auch diese beendet ist, fährt er zurück zur Startposition und die Aufgabe ist erfüllt.

14.6 Probleme bei der Implementierung

Bei der Implementierung der Steuerung unter Saphira (in der Version 6.2c) gab es einige Probleme, die teilweise bereits angesprochen wurden. An dieser Stelle sollen sie noch einmal zusammengefaßt werden. Zu beachten ist, dass sie bei der speziellen, für die Arbeit eingesetzten Hard- und Software, beobachtet wurden. Dies bedeutet, dass sie nicht zwangsläufig auf allen ähnlichen Systemen in dieser Form auftreten müssen.

- Status von Activities
  Normalerweise sollte der Status von Activities über den Befehl `sfGetTaskState` abgefragt werden können, um z.B. festzustellen, ob es erfolgreich war oder fehlgeschlagen ist. Leider ist dieses Kommando nicht sehr zuverlässig, sodass anstelle dessen der Weg über globale Statusvariablen gewählt wurde.

- Greifer-Lichtschranken
  Wie an der entsprechenden Stelle schon angesprochen wurde, liefern die Lichtschranken des Pioneer 2 „Alfa“ bei komplett geöffnetem Greifer über den Befehl `sfP2GripBeam` keine korrekten Werte. Abhilfe schafft hier ein teilweises Schließen der Greifer.

- Das `rotate`-Kommando
  Ein netter Effekt trat in Verbindung mit dem `rotate`-Kommando auf, das den Roboter mit einer bestimmten Geschwindigkeit drehen lassen sollte. Übergibt man diesem Befehl einen Wert, dann dreht sich der Roboter auch so, wie er es eigentlich tun sollte. Problematisch wird es erst, wenn man versuchen will, die Drehbewegung über ein `rotate(0)` oder `halt` wieder zu stoppen. Dies gelingt häufig nicht. Hierbei ist aufgefallen, dass dieses Problem besonders im Zusammenhang mit einer gleichzeitigen Vorwärtsbewegung über `speed` auftritt. Eine Lösung dafür habe ich in der Pioneer-Newsgroup auf der ActivMedia-Homepage gefunden. Nach Aufruf des Stopp-Befehles, muss noch ein `turn(0)` verwendet werden, um die Dreh-Bewegung zu beenden.

- Saphira-Positionierung
  Ein sehr großes Problem gab es in Verbindung mit den globalen Koordinaten des Roboters, das besonders bei Ausführung des `move`-Befehls deutlich wurde. Unter bestimmten Bedingungen werden die Koordinaten des Roboters eine gewisse Zeit lang nicht richtig aktualisiert. Der Effekt ist dann, dass sich der Roboter bei einem Befehl wie `move(200)` anstatt 20 cm ca 1 m weit bewegt, aber auf Grund seiner Koordinaten trotzdem der Meinung ist, nur 20 cm zurückgelegt zu haben. Für eine Aufgabe wie die in dieser Arbeit, die sich zu einem großen Teil auf die Positionierung verlassen muss, ist ein solcher Fehler gravierend.
  Um die Ursache des Problems zu finden, wurden verschiedene Versuche durchgeführt, welche allerdings erfolglos blieben. Eine Schwierigkeit bestand dabei vor allen darin, dass der Fehler nur sporadisch und nicht immer an den gleichen Stellen auftrat.
  Da das Problem somit weiterhin besteht, wurden in der Steuerung zusätzliche Sicherungen eingebaut, die dieses kompensieren können. Ein Beispiel hierzu ist

ein selbstentwickeltes `move`-Kommando, welches neben den Roboterkoordinaten noch die seit seinem Start vergangene Zeit überwacht. Es bewegt den Roboter mittels `speed` vorwärts und beendet sich, wenn der Roboter die gewünschte Distanz zurückgelegt hat (Überprüfung anhand seiner Koordinaten) oder aber eine vorher ermittelte Zeit überschritten wurde. Damit kann, selbst wenn die Koordinaten sehr ungenau werden, noch eine annähernd richtige Bewegung erzielt werden.

Teil VI

Testszenarien

15 Ergebnisse

Um die Funktion der entwickelten Lösung zu überprüfen, habe ich verschiedene Tests durchgeführt. Der Ablauf war dabei, dass zunächst auf dem Teppich Objekte verteilt wurden, anschließend der Roboter auf seine Startposition gebracht wurde und das Haupt-Activity mit `start mainActivity` gestartet wurde.
Ziel sollte sein, dass sich zum Schluß sämtliche Objekte in der Ablage befinden und der Roboter wieder seinen Standort an der Ausgangsposition eingenommen hat und die Erfüllung der Aufgabe meldet. Ein (menschliches) Eingreifen während der Arbeit des Roboters sollte hierbei nicht notwendig sein.
Nachfolgend möchte ich einige Ergebnisse dieser Tests vorstellen sowie auf ein ausgewähltes Szenario etwas genauer eingehen.

Grundsätzlich ist festzustellen, dass in den meisten Fällen der Aufsammelvorgang ohne Probleme vonstatten ging.
Anfangs gab es noch Schwierigkeiten mit dem im letzten Abschnitt angesprochenen Positionierungsfehler (siehe 14.6 auf Seite 132). Nachdem diese jedoch ausgeräumt wurden, traten kaum noch außergewöhnliche Ereignisse auf.
Es treten zwar gelegentlich noch kleinere Störungen auf, z.B. bei Bildstörungen, diese werden aber in der Regel kompensiert.

Bei den Experimenten hat sich aber gezeigt, dass es sehr wichtig ist, dass die Rahmenbedingungen (siehe Abschnitt 19 auf Seite 147) eingehalten werden. Hierbei ist besonderes Augenmerk auf die Beschaffenheit der Objekte zu legen. Für die Bildverarbeitung ist dabei zunächst die Farbigkeit von Bedeutung.
Es wurde definiert, dass sich das Spielzeug deutlich vom Untergrund abheben soll. Dies sollte dann jedoch auch in den aufgenommenen Kamerabildern gelten. In der Praxis können Objekte, die sich für den Betrachter vom Untergrund abheben, in der Kameraaufnahme als mit dem Teppich vereint zu erkennen sein. Sie werden von

den Bildverarbeitungsmethoden somit nicht mehr registriert. Dieser Effekt kann auf dem genutzen Untergrund besonders mit hellen (weißen), grünen oder blauen Objekten auftreten, wobei anzumerken ist, dass solche Fälle eher selten sind.
Wichtiger als die Farbe sind meist die Ausmaße und das Material der Objekte. Sie sollten daher so beschaffen sein, dass sie vom Roboter registriert und aufgenommen werden können. Probleme können bei zu kleinen oder durchsichtigen Objekten auftreten, da diese von den Greiferlichtschranken nicht mehr wahrgenommen werden können. Der Aufnehmvorgang für ein solches Objekt würde nach einer bestimmten Zeit zwar abgebrochen und die Suche normal fortgesetzt werden, jedoch ist es möglich, dass dasselbe Objekt bei der folgenden Suche wieder erkannt wird, und der Roboter erneut versucht, es aufzunehmen. Dies könnte zu einer Endlosschleife führen.
Ähnliches gilt auch für zu große Objekte.
Zusätzlich sollte auf die Objekthöhe geachtet werden.
In der Implementierung wird der Greifer, nachdem er geschlossen wurde, auf seine maximal mögliche Höhe bewegt, damit er anschließend über den Rand der Kiste bewegt werden kann. Sind die Objekte dabei allerdings zu hoch (z.B. Becher mit h=13 cm), kann der Fall auftreten, dass sich diese beim Anheben unter dem Sonarring verklemmen und der Greifer nicht mehr ganz nach oben bewegt werden kann. Dies führt dazu, dass die Kiste beim Ablegen verschoben wird und die Markierung später nicht mehr sichtbar ist.
Problematisch sind auch Objekte, die sich beim Zugreifen verformen können und sich in Richtung Boden ausdehnen, wie z.B. Papierknäul. Es tritt beim Ablegen dann ein ähnlicher Effekt, wie bei zu hohen Objekten auf.
Zusätzlich sollten auch Objekte vermieden werden, die im unteren Bereich (wo sie vom Roboter gegriffen werden sollen) breiter werden und eine glatte Oberfläche besitzen, z.B. kegelförmige Objekte. Dort kann der Fall auftreten, dass sie während des Anhebens oder der Fahrt nach unten aus dem Greifer herausfallen.
Neben den Objekteigenschaften sollte auch darauf geachtet werden, dass sich der Roboter anfangs auf der richtigen Position befindet. Besonders wichtig ist hierbei, dass er nicht zu weit von der Ablage entfernt steht.

16 Testreihen

Als Beispiel für durchgeführte Testreihen dient das folgende Szenario:
Auf dem Untergrund werden 10 Objekte verteilt, die vom Roboter aufgenommen werden sollen. Abbildung 29 zeigt die Ausgangsposition.
Der Versuch wird einige Male wiederholt, wobei dieselben Objekte an unterschiedlichen Orten verwendet werden.

Abbildung 29: Die Ausgangsposition

In der folgenden Tabelle sind die Ergebnisse einer ersten Testreihe zusammengefaßt:

Versuch	Erfolg (j/n)	Dauer (min)	Bemerkung
1	j	26	
2	j	26	Die weiße Kugel wurde nicht erkannt.
3	j	25	
4	j	27	Die Flasche Kleber konnte 1x nicht korrekt gegriffen werden, sodass der Roboter 1x umsonst zur Ablage fuhr. Beim zweiten Versuch hat er es aber geschafft.
5	j	25	
6	j	27	Der angesprochene Positionierungsfehler ist aufgetreten, konnte beim Ausrichten aber kompensiert werden.
7	j	25	
8	j	26	

(Das Problem mit der weißen Kugel in Versuch 2 war, dass sie bei der Suche

von der Startposition aus, auf Grund ihrer Farbe und Position nicht erkannt wurde. Beim Zufahren auf das nächste zu greifende Objekt, landete sie aber im Greifer und wurde anstelle diesen gegriffen.)

Das Ergebnis dieser Testreihe ist somit, dass in allen Versuchen die Aufgabe, auch mit teilweisen Schwierigkeiten, erfüllt wurde, d.h. am Ende alle Objekte in der Kiste lagen und der Roboter wieder auf der Ausgangsposition stand.

Nachdem noch kleinere Änderungen in der Steuerung durchgeführt wurde, welche jedoch nicht die eigentliche Logik betrafen, wurde eine weitere Testreihe mit gleichen Vorraussetzungen durchgeführt, welche auch zwei Ausnahmefälle aufzeigt (die aber nicht versionsbedingt sind).
Die Ergebnisse:

Versuch	Erfolg (j/n)	Dauer (min)	Bemerkung
1	j	24	
2	n	18	Fehler bei der Kamerasteuerung 6 Objekte korrekt abgelegt
3	j	28	
4	j	26	Positionierung, komp.
5	j	27	
6	j	26	Positionierung, komp.
7	j	27	
8	n	27	Hinterste Begrenzungslinie nicht gefunden; Suche nicht beendet. Alle Objekte korrekt abgelegt.

In Versuch 2 trat ein Fehler bei der Kamerasteuerung auf:
Beim Zurückfahren zur Ablage, nachdem ein Objekt aufgenommen wurde, wird die Kamera etwas nach oben bewegt, um nach der Markierung der Kiste zu suchen. Der Steuerungsbefehl wird dabei normalerweise sofort ausgeführt. In Versuch 2 jedoch sind zwischen Kommando und Ausführung 15-20 s vergangen. Der Roboter war nach dieser Zeit schon odometriegesteuert zur Ablage gefahren und gerade beim Ausrichten an den Linien. Da an dieser Stelle nicht mit einer sich ungewollt nach oben bewegenden Kamera gerechnet wird, ist die Steuerung davon ausgegangen,

dass die Linie verloren wurde und hat die Applikation beendet.
In Versuch 8 wurde bei der letzten Suche (nachdem alle Objekte bereits abgelegt wurden) die hinterste Begrenzungslinie nicht erkannt, sodass der Roboter immer weiter fuhr und die Applikation (per Hand) gestoppt werden musste.

Das Ergebnis der zweiten Testreihe ist somit, dass die Aufgabe nur in 75% der Fälle erfolgreich ausgeführt wurde. Es ist hierbei aber anzumerken, dass die zwei Fehler wirkliche Ausnahmefälle darstellen.

Neben den vorgestellten Testreihen wurden im Laufe der Arbeit auch andere Szenarien getestet. Für die Laufzeit der Applikation lässt sich sagen, dass pro Objekt ungefähr 2-2.5 min gerechnet werden müssen. In einem getesteten Fall z.B. 46 Minuten für 21 Objekte.

17 Ein Beispiel-Szenario

Wie eingangs erwähnt wurde, soll abschließend ein konkreter Fall etwas genauer beschrieben werden. Im Folgenden werden dazu auch einige Aufnahmen aus diesem Szenario abgebildet. Das Video hierzu, das den gesamten Vorgang zeigt, ist auf der, dieser Arbeit beigelegten, CD zu finden.
Zunächst die Ausgangssituation:
Auf dem Teppich liegen verteilt 6 Objekte und der Roboter befindet sich auf seiner Ausgangsposition (Abbildung 30).

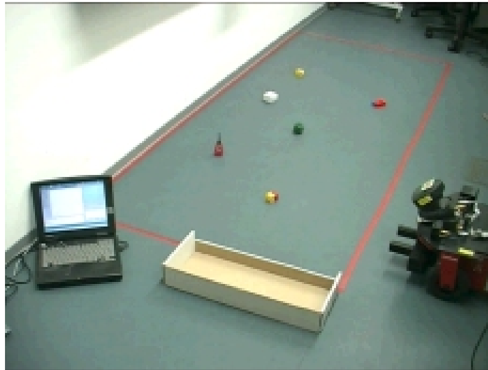

Abbildung 30: Die Ausgangsposition

Nach dem Start beginnt die Suche im ersten Sektor, d.h. vor der Kiste. Dabei wird zunächst die, vom Roboter aus, linke untere Teppichecke gesucht und anschließend mit der Kamera ein Rechtsschwenk durchgeführt.
Das sich in diesem Sektor befindliche Objekt wird hierbei erkannt und der Greifvorgang gestartet (Abbildung 31 auf der nächsten Seite). Nachdem das Objekt aufgenommen wurde, bewegt sich der Roboter auf den, vom Startpunkt aus generierten, Punkt vor der Ablage und beginnt mit dem Ausrichten an den Linien (Abbildung 31 auf der nächsten Seite und Abbildung 32 auf der nächsten Seite).

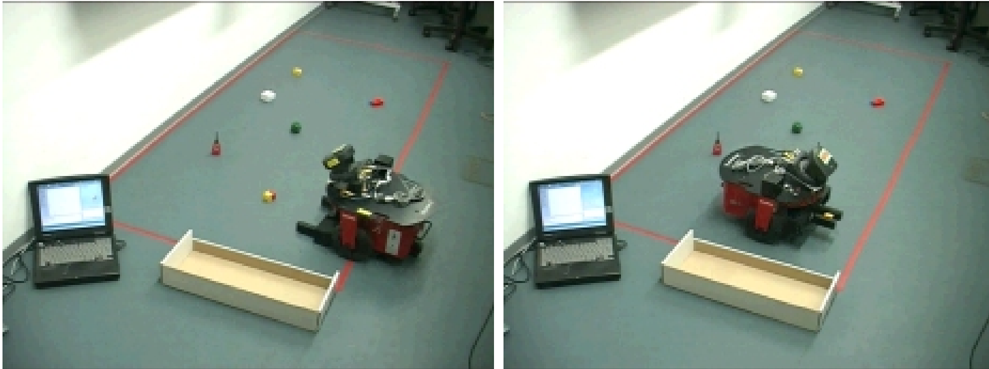

Abbildung 31: links: Zufahrt auf das erste Objekt; rechts: Ausrichten an der Startlinie

Im Anschluß an das Ausrichten wird das Objekt in der Kiste abgelegt (Abbildung 32) und der Roboter kehrt auf seine Startposition zurück. Von dort aus wird nochmals der erste Sektor abgesucht, da sich dort möglicherweise noch weitere Objekte befinden, die von einem Standort vor der Kiste aus nicht erkannt werden können.

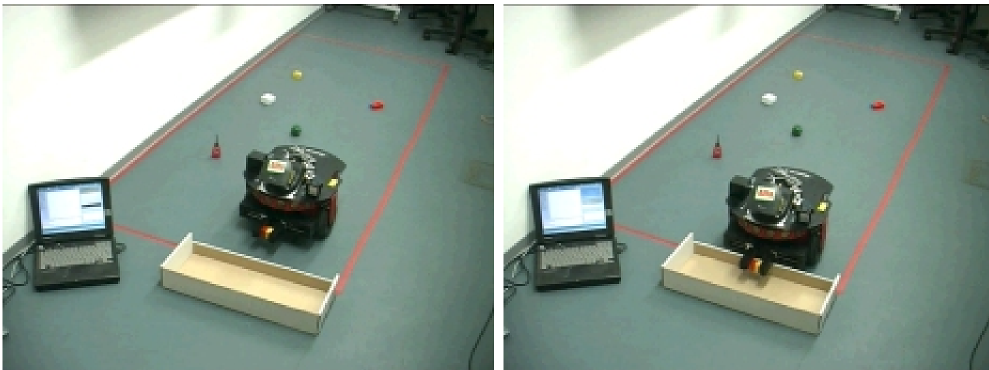

Abbildung 32: links: Ausrichten an der Linie vor der Kiste; rechts: Ablegen des Objekts

Im gegebenen Fall sind jedoch keine weiteren Objekte vorhanden, sodass die Suche in Sektor 1 abgeschlossen wird.
Der Roboter bewegt sich daher auf den Punkt vor der Kiste und startet die Suche im zweiten Sektor. Um dies zu erledigen, wird zunächst von dieser Position aus ein Kameraschwenk ohne Roboterbewegung ausgeführt. Hierbei wird im Beispiel auch die Klebstoffflasche erkannt (Abbildung 33 auf der nächsten Seite).

Nachdem dieses Objekt aufgenommen wurde, fährt der Roboter über die Odometriesensoren zum gespeicherten Punkt vor der Kiste zurück, richtet sich an den Eck-Linien aus und legt das Objekt ab. (Ein bildverarbeitungsbasiertes Fahren zur Ablage kann im zweiten Sektor nicht erfolgen, da die Markierung der Kiste von dort aus nicht korrekt erkannt werden kann.). Anschließend wird die Suche in Sektor 2 erneut gestartet. Nachdem dabei von der Position vor der Ablage aus keine weiteren Objekte erkannt wurden, beginnt der Roboter, sich in Richtung der, der Startlinie gegenüberliegenden, Linie zu bewegen, wobei er über die Kamera weiterhin Objekte sucht. Die Fahrt dauert solange an, bis die Linie gefunden wurde. Ist dieser Vorgang beendet, kommt von der letzten Position aus noch einmal eine im Stand laufende Suche zum Einsatz. Im Beispiel werden im zweiten Suchgebiet keine weiteren Objekte gefunden. Der Roboter bewegt sich daher in Richtung Teppichmitte, dreht sich in Längsrichtung und beginnt die Suche in Sektor 3 (Abbildung 33).

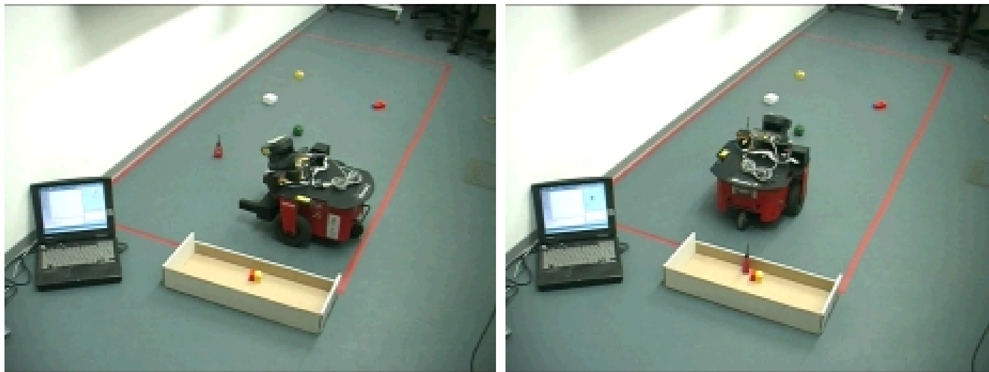

Abbildung 33: links: Die Klebstoffflasche wurde erkannt; rechts: Start der Suche in Sektor 3

Der Ablauf der Suche ist derselbe, wie das im zweiten Sektor eingesetzte Verfahren, d.h. also zunächst ohne Bewegung, dann mit Vorwärtsbewegung, bis die Linie erkannt wird (diesmal die der Ablage gegenüberliegende) und abschließend eine im Stand laufende Suche. Der Unterschied liegt im Wesenlichen in dem, bei der Fahrt zurück zur Kiste, eingesetzten Verfahren.
Der Roboter prüft, nachdem ein Objekt aufgenommen wurde, welches der möglichen Verfahren dafür eingesetzt wird. Hierzu dreht er sich zunächst in Richtung des

gespeicherten Punktes vor der Ablage.
Wenn die Distanz zum Startpunkt einen bestimmten Wert unterschreitet und die Kiste nicht im Bild sichtbar ist, dann kommt die odometriebasierte Lösung zum Einsatz und anderenfalls das bildverarbeitungsgesteuerte Verfahren. Im Beispiel-Szenario wird das erste (grüne) Objekt per Odometrie zur Ablage befördert, während bei den weiteren Objekten die bildverarbeitungsbasierte Lösung eingesetzt wird (Abbildung 34).

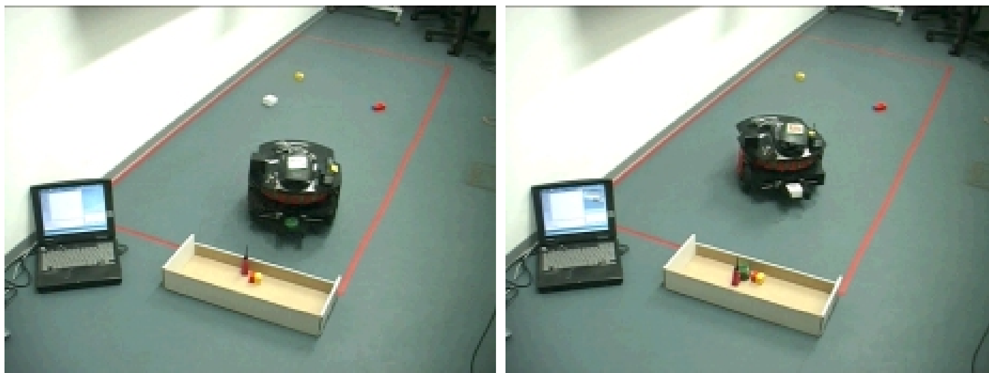

Abbildung 34: links: Ankunft vor der Kiste nach einer odometriebasierten Fahrt; rechts: Fahrt zur Kiste über das bildverarbeitungsbasierte Verfahren

Nachdem jeweils ein Objekt erkannt und aufgenommen wurde, startet die Suche im dritten Sektor erneut. Wird dabei kein Objekt mehr gefunden, dann fährt der Roboter zurück in Richtung Kiste und bewegt sich abschließend auf seine Startposition (Abbildung 35 auf der nächsten Seite). Zum Schluß wird in Saphira noch eine Ausgabe erzeugt und das Verhalten beendet.

Im aufgeführten Szenario trat nur eine Besonderheit auf. Auf Grund von Funkstörungen wurden fehlerhafte Bilder verarbeitet, sodass in Sektor 3 einmal Objekte erkannt wurden, die nicht vorhanden waren. In der Regel sind solche Störungen jedoch nur von sehr kurzer Dauer und schon beim „genauer hinschauen" nicht mehr existent. Hier dauerten sie allerdings etwas länger an, was bewirkte, dass sich der Roboter zum Objektaufnehmen schon in Richtung der vermeintlichen Objekte drehte. (Auf dem Video ist dies sehr schön zu sehen.) Nachdem er sich gedreht hatte, ließen die Störungen aber wieder nach, sodass die Objekte wieder

aus dem Bild verschwanden. Der Roboter konnte sich daher wieder zurückdrehen und die Suche normal fortsetzen.

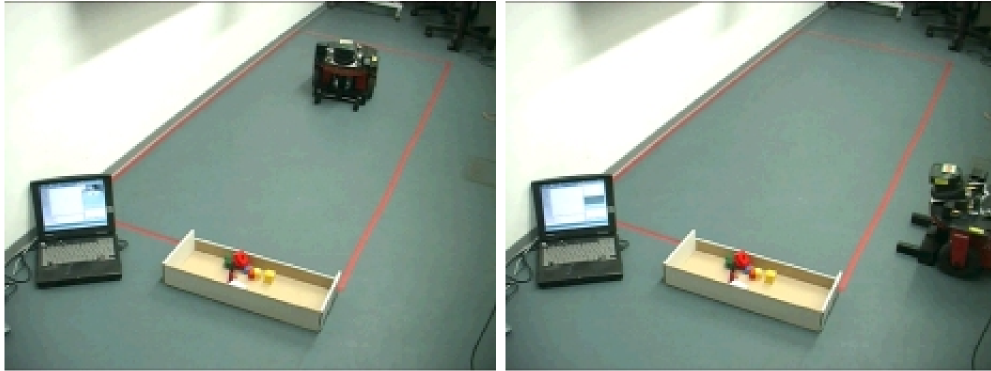

Abbildung 35: links: Die Suche ist abgeschlossen, Fahrt zurück zur Kiste ; rechts: Fertig!

Teil VII

Zusammenfassung und Ausblick

Nachdem in der Arbeit eine Realisierung für die Lösung der Aufgabenstellung und auch Testszenarien vorgestellt wurden, sollen abschließend die Ergebnisse zusammengefaßt und ein kleiner Ausblick auf weitere denkbare Funktionalitäten gegeben werden.

Die erste Frage hierzu ist: *Was habe ich erreicht?*

18 Zusammenfassung

Die Aufgabe war, eine vollständige Demoapplikation für den Einsatz eines Serviceroboters im Haushalt, speziell im Kinderzimmer, auf Basis des Pioneer 2-Roboters zu entwickeln. Hauptziel war dabei nicht, ein konkretes, im realen Kinderzimmer einsetzbares, System zu entwerfen, sondern mit einer vereinfachten Umgebung ein Beispiel aufzuzeigen, wie etwas Derartiges aussehen könnte.
Ich denke, dass mir dieses im Großen und Ganzen recht gut gelungen ist.
Es wurde hierbei gezeigt, welche Bildverarbeitungsmethoden für die geforderten Aufgaben geeignet sind, und wie diese in das System integriert werden können. In diesem Zusammenhang war besonders die Kommunikationsschnittstelle zwischen der Bildverarbeitungssoftware und der eigentlichen Steuerungssoftware (Saphira) von Bedeutung. Des Weiteren wurde eine Steuerungsarchitektur auf Basis von COLBERT-Activities entwickelt, welche zwar auf einem vorher definierten Plan basiert, aber trotzdem reaktive Komponenten besitzt.

Zur Aufgabe an sich lässt sich noch sagen, dass diese doch etwas umfangreicher ausgefallen war, als ich zunächst vermutet hatte. Dies liegt vor allem in den komplexen zu berücksichtigenden Aspekten, wie Bildverarbeitung und Steuerung, begründet. Gerade die Steuerungsroutinen stellten eine besondere Herausforderung dar. Normalerweise ist man gewohnt, dass die Befehle in selbst geschriebenen Programmen korrekt abgearbeitet werden. Für die Steuerung eines Roboters

allerdings lässt sich dies nicht behaupten. Bei jeder Aktion, die der Roboter ausführen soll, muss, gerade bei der gewählten Architektur, überlegt werden, was dabei alles schiefgehen kann, und wie man am Besten darauf reagiert, wobei selbst die einfachsten Kommandos, wie `move` die merkwürdigsten Effekte nach sich ziehen können.
Ich könnte auch noch einige Monate oder gar Jahre an dem Thema weiterarbeiten, was aber den Rahmen einer Diplomarbeit sprengen würde.
Dies bedeutet aber nicht, dass ich auf halber Strecke aufgehört habe, sondern vielmehr, dass die Entwicklung eines Robotersystemes, das ein Höchstmaß an Flexibilität besitzt und für alle Eventualitäten gerüstet ist, eben einen enormen Zeitaufwand bedingt. Trotzdem ist ein System entstanden, das die geforderte Aufgabe erfüllt. Hierbei sind jedoch einige Rahmenbedingungen bzw. Einschränkungen zu berücksichtigen, die im Laufe der Arbeit bereits vorgestellt wurden und im nächsten Abschnitt noch einmal zusammengefaßt werden.

Im letzten Satz der Aufgabenstellung wurden zwei unscharfe Begriffe genannt, die es, neben dem technischen Aspekt, bei der Umsetzung zu beachten galt. Ziel war, nicht nur ein System zu entwickeln, das Spielzeug findet, aufsammelt und wieder abliefert, sondern das auch noch robust und „gutmütig“ arbeitet. Das heißt also, dass die fertige Lösung tolerant gegenüber möglichen Störungen sein sollte. Da in den verschiedenen Activities in erster Linie ein Plan abgearbeitet wird, müssen dort auch verschiedene Aktionen, die fehlschlagen können, berücksichtigt werden.
Weil es für die genannten Begriffe aber keine hundertprozentigen Definitionen gibt, und die Interpretationen teilweise subjektiv sind, kann ich jetzt z.B. nicht per Definition sagen, dass das System robust ist, oder nicht. Ich habe aber einiges dazu getan, dass dieses in meinen Augen so ist.
Nachfolgend sind hierzu Beispiele aufgeführt:

- Wenn Aktionen ausgeführt werden, die auf Werte eines bestimmten Sensors warten, dann werden, wenn möglich, zusätzlich noch andere Sensorinformationen berücksichtigt.
  Beispiel beim Objektgreifen, nachdem das Objekt nach unten aus dem Bild gelaufen ist: *Fahre solange, bis die Greiferlichtschranken ein Objekt erkennen, aber höchstens 30 cm.* Zusätzlich zu den Greiferdaten werden hier also noch

Odometrieinformationen überprüft.

- Das „genauer Hinschauen“ bei der Objektsuche hat die Aufgabe, die nähere Umgebung des Objektes zu untersuchen und somit zu verhindern, dass der Roboter auf etwas zufährt, das in Wirklichkeit kein Objekt darstellt.
- Beim Zufahren auf eine Linie oder ein Objekt, können diese kurzzeitig aus dem Bild verschwinden (z.B. bei Kamerafehlern), ohne dass das entsprechende Verhalten gleich fehlschlägt. Der Roboter hält dann kurz an.
- Bei der Suche im letzten Sektor schaut der Roboter gelegentlich, ob die hinterste Begrenzungslinie (gegenüber der Ablage) sichtbar ist, und richtet sich ggf. an ihr aus. Damit werden Ungenauigkeiten, die bei den Fahr- und Drehbewegungen auftreten können, ausgeglichen.
- Durch das Ausrichten an den Linien vor der Kiste werden dort Navigationsungenauigkeiten kompensiert.

Abschließend kann ich also sagen, dass ein „rundes“ System entstanden ist, das die geforderten Aufgaben erfüllt und auch recht zuverlässig arbeitet.

19 Rahmenbedingungen und Einschränkungen

Wie angesprochen, sollen jetzt noch einmal die sich während der Arbeit ergebenden Einschränkungen und Rahmenbedingungen der verschiedenen Objekte zusammengefaßt werden.

Teppich:

- der eigentliche Untergrund sollte eine Farbe besitzen, von der sich Spielzeug und Teppichränder deutlich abheben
- homogener Farbverlauf
- sollte keine allzu großen Lichtreflexionen ermöglichen

- Die Teppichränder sind für den vorgestellten Einsatzort in roter Farbe und bilden ein Rechteck von ca. 1.3x3 m

Spielzeug:

- hebt sich farblich vom Untergrund ab
- muss für den Roboter greifbar sein, (nicht zu klein wg. Lichtschranken, nicht zu groß wg. Greifer); hauptsächlich kommen hier LEGO-Duplo-Bausteine mit einer Größe von ca. 3x3x4 cm zum Einsatz
- verschiedene Objekte sollten deutlich erkennbar auseinanderliegen
- es sollten nicht zu viele rote Objekte vorhanden sein, um die Teppicherkennung nicht zu beeinflussen
- sollten von den Teppichrändern einen Abstand von ca. 10 cm einhalten
- direkt vor der Ablage (Suchgebiet 1) sollten sich kaum Objekte befinden

Ablage:

- Größe:
  - sollte so groß sein, dass auch einige Objekte abgelegt werden können
  - sollte etwas breiter sein, damit kleinere Ungenauigkeiten bei der Robotersteuerung nicht ins Gewicht fallen
  - der Rand zum Teppich hin muss so niedrig sein, dass der Roboter ein Objekt darüberheben kann

    (ich habe daher mit einer Tastaturverpackung mit den Maßen 58x23x10 cm gearbeitet)
- muss direkt an den Teppich grenzen
- ist an der Teppichseite in der Mitte mit einem grün-gelb-grünen Farbmuster markiert

20 Ausblick

Eine Frage, die sich abschließend stellt, ist, wie man die entwickelte Lösung noch verbessern kann. Wie schon erwähnt, könnte man sich noch eine lange Zeit mit der Arbeit beschäftigen und würde immer wieder etwas Neues entdecken, das noch realisiert werden könnte. Ich möchte an daher an dieser Stelle noch einige Anregungen geben, die mir bei der Implementierung aufgefallen sind.

Bildverarbeitung
Ein erster verbesserungswürdiger Punkt ist die Bildverarbeitung. Diese basiert momentan auf 15 Bit-Bildern und läuft mit bzw. ohne Verarbeitungsroutinen bei ca. 4 bzw. 8 Hz. Gerade die Bildqualität lässt dabei etwas zu wünschen übrig. Als Ansatz zur Verbesserung kann der neue USB-Videobus der Firma Belkin, der bereits erfolgreich getestet wurde, verwendet werden. Dieser liefert bei hohen Frameraten sehr gute Bilder. Durch seinen Einsatz sind auch komplett andere Suchverfahren denkbar.
Ich hatte immer mit dem Problem zu kämpfen, dass die Bildqualität mit zunehmender Entfernung, also höherstehender Kamera, deutlich schlechter wurde, und habe die Kamera daher so weit wie möglich nach unten geneigt. Mit den besseren Bildern über den Videobus, können aber Verfahren entwickelt werden, die vorrausschauender sind und somit auch die Geschwindigkeit der Suche erhöhen.
Um die Bilder über den Videobus zu verarbeiten, wurde an der FH Brandenburg bereits ein neuer Bildverarbeitungsserver entwickelt, welcher Video For Windows (VfW)-Funktionen benutzt. Die Grundlage hierfür bilden Kamera- und Bild-Klassen aus der Open Source Computer Vision Library (OpenCV) [Int01b] der Firma Intel. Die OpenCV nutzt die Verarbeitungsfunktionen der IPL, die bereits in Abschnitt 6 auf Seite 23 vorgestellt wurde, und erweitert sie um einige spezielle Verfahren.
Wie schon erwähnt, besitzt die IPL ihre eigene Bildstruktur (`IplImage`), welche in der OpenCV zusätzlich in einer eigenen Bildklasse (`CImage`) enthalten ist. Im neuen Bildverarbeitungsserver werden ebenfalls diese Objekte verwendet, um einerseits die dort schon enthaltenen VfW-Funktionalitäten und andererseits die Verarbeitungsmethoden leichter nutzen zu können.
Damit zukünftig auch meine Verarbeitungsroutinen im neuen Server eingesetzt

werden können, habe ich diese bereits auf die Objekte dieser Bibliotheken umgestellt.

Neben den Grabfunktionen können natürlich auch die eigentlichen Bildverarbeitungsmethoden noch verändert werden. Die Ergebnisse sind zwar momentan sehr gut, aber man könnte sich z.B. für die Objekterkennung im Randbereich noch andere Verfahren als das einfache Linien-aus-dem-Bild-entfernen vorstellen. Zusätzlich sind auch Objektidentifizierungsverfahren möglich, die vielleicht eine Klassifizierung des Objektes, z.B. nach Typ vornehmen.

Um den Bildverarbeitungsserver unabhängiger von den eigentlichen Verarbeitungsroutinen zu machen, gibt es bereits Überlegungen, diese in ladbaren Modulen (z.B. über DLLs) unterzubringen, von denen die jeweils benötigte ausgewählt werden kann. Dies ist besonders zu empfehlen, da momentan für jeden Einsatzzweck noch eine separate Bildverarbeitungsanwendung erstellt wird. Zwar basieren diese alle auf dem vorhandenen Server, allerdings werden in jeder Anwendung oft nützliche Änderungen vorgenommen, die im nächsten Projekt dann verlorengehen.

Greiferkamera

In der vorgestellten Lösung bewegt sich der Roboter, nachdem ein zu greifendes Objekt nach unten aus dem Bild verschwunden ist, noch einige Zeit im Blindflug vorwärts und wartet während dessen darauf, dass die Lichtschranken irgendwann ein Objekt melden. Wenn dies der Fall ist, wird der Greifer geschlossen und das Objekt angehoben und abgeliefert. Es kann hierbei allerdings nicht hundertprozentig festgestellt werden, ob sich letztendlich wirklich ein Objekt im Greifer befindet. Eine Abfrage der Lichtschrankenwerte bringt nichts, da sich das Objekt teilweise auch zwischen den beiden Lichtschranken befinden kann. Abhilfe könnte aber eine Kamera schaffen, die den Greiferzwischenraum überwacht.

Steuerung

Ein Punkt, an dem noch im Bereich der Steuerung gearbeitet werden kann, ist die Gesamtgeschwindigkeit des Systemes. Die Fahrgeschwindigkeit des Roboters ist dabei hauptsächlich von der Geschwindigkeit der Bildverarbeitung und der Kamerabewegungen abhängig. Ich habe sie bei den gewählten Suchmethoden auf

höchstmögliche Zuverlässigkeit abgestimmt. Besonders im Zusammenhang mit einer verbesserten und schnelleren Bildverarbeitung kann die Geschwindigkeit aber noch erhöht werden. Es sollte aber beachtet werden, dass die Kamera bei Bewegungen neben der reinen Bewegungszeit teilweise noch etwas mehr Zeit benötigt, um sich scharfzustellen.

Weiterhin ist es möglich, während der Suche eine Karte des Teppiches (in Saphira) zu erstellen, und darin die gefundenen (und möglicherweise identifizierten) Objekte zu vermerken. Diese könnte dann als Orientierung und zur Optimierung der Suche dienen. Zusätzlich können hier beispielsweise auch nicht greifbare Objekte (im Zusammenhang mit dem Einsatz einer Greiferkamera) eingetragen werden, um zu verhindern, dass der Roboter immer wieder versucht, solche aufzunehmen.
Momentan wird auch die gesamte Objektidentifizierung ausschließlich von der Bildverarbeitung erledigt. Das heißt, dass wenn die Kamera, aus welchen Gründen auch immer, so über den Teppichrand hinausblickt, dass die Begrenzungslinien nicht mehr korrekt erkannt werden können, Objekte außerhalb des Suchgebietes als gültig erkannt werden würden. Der Roboter würde dann versuchen, diese aufzunehmen und somit den Einsatzbereich verlassen. Über eine Karte könnten die Objektpositionen dann, ggf. nach einer Entfernungsberechnung, überprüft werden.

Variable Teppichgröße

Für die entwickelte Lösung wurde die Größe des Einsatzortes fest vorgegeben und die Suchverfahren und Kamerabewegungen darauf abgestimmt. Man könnte sich aber auch eine Lösung vorstellen, die von der Teppichgröße unabhängig ist, diese selbständig herausfindet und entsprechende Suchverfahren auswählt.
Momentan ist die Größe im Steuerungsprogramm zwar nicht fest verdrahtet, jedoch kann, speziell bei deutlich breiteren Teppichen, nicht mehr garantiert werden, dass alle Bereiche korrekt abgesucht werden. Besser ist also, ein explizit von der Größe unabhängiges System zu entwickeln.

Anhang A

Verwendete Strukturen

```
/**
 * Struktur, die Informationen einer Gerade der Form
 *
 * r = x*cos th + y*sin th (Hessesche Normalform) speichert.
 *
 * th - der Winkel zwischen x-Achse und dem Lot
 *      vom Koordinatenursprung auf die Gerade
 * r  - Laenge des Lotes
 */
typedef struct tagLINE{
    int iAngle;     // th
    int iDistance;  // r
    int iCount;     // Anzahl Punkte
    int iX1;        // X-Wert des linkesten Geraden-Punktes im Bild
    int iY1;        // Y-Wert des linkesten Geraden-Punktes im Bild
    int iX2;        // X-Wert des rechtesten Geraden-Punktes im Bild
    int iY2;        // Y-Wert des rechtesten Geraden-Punktes im Bild
    BOOL bVBelow;   // Flag, das anzeigt, ob gueltige Objekte unter(TRUE)
                    // oder ueber der Linie sind
} LINE;
```

```
/**
 * Struktur, die Informationen zu im Bild erkannten Regionen haelt,
 * wie z.B. Schwerpunkt, Anzahl Pixel
 *
 */
typedef struct tagREGINFO{
    long int liCount; // Anzahl Punkte
    int iClass;       // Merkmalsklasse, wenn benoetigt
    int iXMin;        // Min x-Wert
    int iYMin;        // Min y-Wert
    int iXMax;        // Max x-Wert
    int iYMax;        // Max y-Wert
    long int liSumX;  // Summe der x,y - Koordinaten aller Punkte ...
    long int liSumY;  // ...zur Schwerpunktberechnung
    int iWidth;       // Breite
    int iHeight;      // Hoehe
    int iXSP;         // Schwerpunkt-...
    int iYSP;         // ...-koordinaten
} REGINFO;
```

```
/**
 * Struktur, die die im SharedMemory zu speichernden Objekte haelt
 *
 */
typedef struct tagCOMM{
    LINE lines[MAX_LINE_COUNT];         // Die gefundenen Linien...
    int iLineCount;                     // ...und deren Anzahl
    REGINFO regions[MAX_REGION_COUNT];  // Die gefundenen Objekte...
    int iRegCount;                      // ...und deren Anzahl
    REGINFO regionTheBox;               // Die Kiste
    LINE boxLine;                       // Die Linie vor der Kiste
    BOOL bBoxFound;                     // Ob eine Kiste gefunden wurde
    BOOL bBoxLineFound;                 // Ob die Linie vor der Kiste
                                        // gefunden wurde

    // Die folgenden Flags dienen der Synchronisation
    // und werden beim Fuellen der SM-Daten durch die
    // Bildverarbeitung gesetzt und beim Auslesen zurueckgesetzt

    BOOL bNewBoxData;       // Ob neue Daten zur Kiste vorhanden sind
    BOOL bNewLineData;      // Ob neue Geradendaten vorhanden sind
    BOOL bNewRegionData;    // Ob neue Objektdaten vorhanden sind

    // Flag, das anzeigt, ob die Applikation in der
    // Startphase (Suche vor der Kiste) ist
    BOOL bInit;
} COMM;
```

Anhang B

Bildverarbeitungsmethoden der Klasse CImageProcessing

```
// Funktionen zur Linienerkennung

// Rotfilter
int RedFilter( int iBorderWidth, int iThreshold,
               IplImage *pImageIn, IplImage *pImageOut );
// Kantenfilter
int EdgeDetection( IplImage* pImageIn, IplImage *pImageOut );
// Hough-Transformation
int HoughTransformation( int iBorderWidth, IplImage *pImageIn );
// Hough-Akku-Analyse
void AnalyzeHoughAccumulator( int iInit );

// Funktionen zur Objekterkennung

// Kantenfilter
int ColorEdgeDetection( int iBorderWidth, int iThreshold,
                        IplImage *pImageIn, IplImage *pImageOut );
// Objekte extrahieren
int ExtractObjects8( IplImage *pImageIn );
// Gueltige Objekte extrahieren -> unter Zuhilfenahme der Linien
void ExtractValidObjects();


// Linien loeschen
int DeleteLines( IplImage *pImage, int iWidth );

// Kiste erkennen
```

```
int LookForTheBox( int iBorderWidth, int iBlackThreshold,
                   int iYellowThreshold, IplImage *pImageIn,
                   IplImage *pImageOut );


// Zeichenfunktionen

// Objekte einzeichnen
int DrawValidObjects( IplImage *pImage );
// Linien einzeichnen
int DrawLines( IplImage *pImage );
// Kiste einzeichnen
int DrawBox( IplImage *pImage );
```

Anhang C

Funktionen im gemeinsamen Speicher

```
/*
 * Rundet den uebergebenen double-Wert in einen int-Wert
 */
int Round( double d );

/*
 * Funktion zum Initialisieren des SharedMemory.
 */
void SMComm_Init( void );

/*
 * Funktion zum Entfernen des SharedMemory.
 */
void SMComm_Close();

/*
 * Funktion, die die im SharedMemory gespeicherten
 * REGINFO-Objekte zurueckliefert.
 */
int SMComm_GetRegions( REGINFO* regions, int iMaxRegionCount );

/*
 * Fuellt die im SharedMemory gespeicherten REGINFO-Objekte.
 */
void SMComm_SetRegions( REGINFO* regions, int iRegionCount );

/*
 * Funktion, die die im SharedMemory gespeicherte
```

```
 * REGINFO-Struktur der Kiste zurueckliefert.
 */
BOOL SMComm_GetTheBox( REGINFO* region );

/*
 * Funktion, die die im SharedMemory gespeicherte Gerade
 * vor der Kiste zurueckliefert.
 */
BOOL SMComm_GetBoxLine( LINE* line );

/*
 * Fuellt die im SharedMemory gespeichert REGINFO-Struktur
 * der Kiste
 */
void SMComm_SetTheBox( REGINFO* region, LINE* boxLine );

/*
 * Funktion, die die im SharedMemory gespeicherten Geraden
 * zurueckliefert.
 */
int SMComm_GetLines( LINE* lines, int iMaxLineCount );

/*
 * Setzt die im SharedMemory gespeicherten Geraden.
 */
void SMComm_SetLines( LINE* lines, int iLineCount );

/*
 * Gibt zurueck, ob neue Objektdaten vorhanden sind.
 */
BOOL SMComm_NewRegionData();

/*
```

```
 * Gibt zurueck, ob neue Geradendaten vorhanden sind.
 */
BOOL SMComm_NewLineData();

/*
 * Gibt zurueck, ob neue Daten zur Kiste vorhanden sind.
 */
BOOL SMComm_NewBoxData();

/*
 * Gibt zurueck, ob das Init-Flag gesetzt ist
 * (wird von der Bildverarbeitung aufgerufen).
 */
BOOL SMComm_GetInit();

/*
 * Setzt das Init-Flag (wird von
 * Colbert-Activities aufgerufen).
 */
void SMComm_SetInit( BOOL bInit );
```

Anhang D

Bibliotheksfunktionen (pioneerbv.dll)

```
/*
 * SharedMemory-Funktionen, die fuer Saphira bekanntgemacht
 * werden.
 */
BOOL NewRegionData();
BOOL NewLineData();
BOOL NewBoxData();
int GetTheBox( REGINFO *region );
SetInit( BOOL bInit );


/*
 * Liefert die Linie vor der Ablage.
 */
BOOL GetBoxLine( LINE* line );

/*
 * Gibt zurueck, ob eine Linie gefunden wurde, die einen
 * bestimmten Bildrand schneidet.
 */
BOOL LineFound( int iBorderIndex, int iMinValue,
                int iMaxValue, int iHeight, int iWidth );

/*
 * Gibt zurueck, ob eine Ecke gefunden wurde.
 */
BOOL CornerFound();
```

```
/*
 * Gibt das Minimum zweier int-Werte zurueck.
 */
int min( int, int );

/*
 * Gibt das Maximum zweier int-Werte zurueck.
 */
int max( int, int );

/*
 * Gibt den absoluten Wert eines int-Wertes zurueck.
 */
int abs( int );

/*
 * Liefert das zur Bildunterkante naechstgelegene Objekt
 */
BOOL GetClosestObject( REGINFO *region, BOOL bCheckTestedRegions );

/*
 * Liefert das zum Punkt (x,y) naechstgelegene Objekt
 */
int GetClosestObject2Point( int x, int y, int iMaxDistance, REGINFO *region ) defin

/*
 * Liefert die unterste horizontale Linie.
 */
BOOL GetHLine( int iMaxAngleDiff, LINE *line );

/*
 * Berechnet den Abstand vom Roboter zu einer Linie, die
 * in Kameramitte verlaeuft, anhand von Neigewinkel und
```

```
 * Kamerahoehe.
 */
int GetLineDistance( float fEviTilt, float fCameraHeight );

/*
 * Liefert die Zeit in ms seit Betriebssystem
 * (benutzt die Win-Funktion timeGetTime(VOID))
 */
int time();
```

Anhang E

Ausgewählte COLBERT-Activities

Datei Main.act

```
/*
 * Das eigentliche Haupt-Activity, welches als Einzigstes
 * vom Anwender mit >start mainActivity< gestartet wird
 */
act mainActivity;
```

Datei GoToPos.act

```
/*
 * Bewegt den Roboter zum uebergebenen Punkt.
 */
act goToPos( point *p, int iTurn );
```

Datei Basic.act

```
/*
 * Activity, das den Roboter senkrecht zur Linie ausrichtet.
 */
act centerRobotToLine( int iMaxDiff );

/*
 * Activity, das den Roboter solange in Richtung einer Linie fahren
 * laesst, bis diese aus dem Bild verschwunden ist.
 */
act moveToLine( int iSpeedBW, int iSpeedFW, int iSearchBoxLine );

/*
 * Activity, das die Entfernung des Roboters zum uebergebenen Punkt
```

```
 * ueberwacht und wenn diese den Maximalwert erreicht, ein globales
 * Flag setzt.
 */
act watchDist( point *pStartPos, float fMaxDist );

/*
 * Activity, das den Roboter eine bestimmte Entfernung fahren laesst
 * (Ersatz fuer das Saphira-Move)
 */
act myMove( float fDist, int iSpeed );
```

Datei Search.act

```
/*
 * Versucht, das Objekt im Bild zu zentrieren
 */
act tryCenterObject( int x, int y );

/*
 * Suchactivity, das je nach Parameter die gewuenschten Bewegungen
 * ausfuehrt und bei einem gefundenen Objekt oder nach kompletter
 * Suche abbricht.
 */
act search( int iInitialSearch, int iMove, int iMinPan,
            int iMaxPan, int iPanStep );

/*
 * Activity, das das aktuelle Bild durchsucht.
 * Alle gefundenen Objekte werden mit tryCenterObject ueberprueft
 * und ggf. die entsprechenden globalen Flags gesetzt
 */
act searchInImage( int iMove );
```

```
/*
 * Suche im zweiten Sektor starten.
 */
act lookInSector2();

/*
 * Suche im dritten Sektor starten.
 */
act lookInSector3( int iWidth );
```

Datei Start.act

```
/*
 * Suche von der Startposition aus.
 */
act lookAtStartPos();

/*
 * Richtet den Roboter an der Startposition aus.
 */
act goToStartPos();
```

Datei TakeObject.act

```
/*
 * Activity, das den Roboter auf das Objekt zufahren laesst
 * und es dann aufnimmt.
 */
act takeObject();
```

Datei GoToTheBox.act

```
/**
 * Der Roboter faehrt zur Kiste und legt das Objekt ab.
```

```
 * Ist r gesetzt, wird das jeweils geeignete Verfahren
 * (odometrie/bildverarbeitungs-basiert) ausgewaehlt
 */
act goToTheBoxAndDropObject( int r );

/*
 * Der Roboter faehrt zum gespeicherten Punkt vor der
 * Kiste und richtet sich nach den Ecklinien aus.
 */
act goToTheBoxA( int iSearch );

/*
 * Richtet den Roboter zur Linie vor der Kiste aus
 * und legt das Objekt ab.
 */
act dropObject();

/*
 * Versucht den Schwerpunkt der Kistenmarkierung in der
 * Mitte des Bildes zu halten, indem die Kamera bewegt
 * wird (fuer reglerbasiertes Fahren).
 */
act centerTheBox();

/*
 * Reglergesteuertes zur-Kiste-fahren
 */
act goToTheBoxR( float fP1, float fP2, int iMaxVel );

/*
 * Activity, das den Roboter die Kiste suchen laesst.
 */
act lookForTheBox();
```

Literatur

[Aac00] *Vorträge zum Seminar „Handlungsplanung“*. 2000. WS 97/98, RWTH-Aachen, http://www-i5.informatik.rwth-aachen.de/LuFG/Lehre/WS97/PLAN/PLAN-WS97.html.

[Act99a] ActivMedia. *Pioneer 2 - Gripper Manual*. 1999. http://robots.activmedia.com,pdf.

[Act99b] ActivMedia. *Pioneer 2 - Operations Manual*. 1999. http://robots.activmedia.com, pdf.

[Act99c] ActivMedia. *Saphira Manual*. 1999. http://robots.activmedia.com, Saphira: http://www.ai.sri.com/~konolige/saphira/index.html, v61f pdf, v62c pdf.

[Ark99] Ronald C. Arkin. *Behavior-Based Robotics*. The MIT Press, 1999.

[Bäs98] Henning Bässmann. *Bildverarbeitung AdOculos*. Springer-Verlag, 1998.

[Bec00] Clemens Beckstein. *Planen – Grundlagen*. 2000. Skript zur Vorlesung *Einführung in die Künstliche Intelligenz*, FSU Jena, http://www.minet.uni-jena.de/www/fakultaet/beckstein/eki-ws9900.html.

[BIH00] B.Kwolek, I.Boersch, und H.Loose. *Verfolgung eines dynamischen Ziels mit dem autonomen mobilen Roboter PIONEER II*. 2000. 4.Brandenburger Workshop Mechatronik 14./15. September 2000, FH Brandenburg.

[Bis98] Rainer Bischoff. Serviceroboter der Zukunft. *Spektrum der Wissenschaft*, 1998. Dossier 4/1998 Roboter erobern den Alltag pdf.

[Bor96] J. Borenstein. *Where am I? Sensors and Methods for Mobile Robot Positioning*. 1996. ftp://ftp.eecs.umich.edu/people/johannb pdf.

[Brä95] Thomas Bräunl. *Parallele Bildverarbeitung*. Addison-Wesley, 1995.

[Bro85] Rodney A. Brooks. *A Robust Layered Control System For A Mobile Robot*. 1985.

[Bro91] Rodney A. Brooks. *Intelligence Without Reason*. 1991. pdf.

[Fuc00] Siegfried Fuchs. *Interpretation von Bildern und Szenen mit komplexen Modellen*. 2000. Computervision-Bildverstehen WS 2000/2001, TU Dresden, http://pikas.inf.tu-dresden.de/~weser/LEHRE/MUBI/cvbv00_inh.html.

[GG99] Jörg Grawe und Oliver Groht. *Entwicklung und Realisierung eines Garbage-Collecting Serviceroboters für partiell bekannte Räume*. 1999. Diplomarbeit an der FH Hamburg - http://fbi010.informatik.fh-hamburg.de/~pioneer/Projekte/projekte.html.

[Gut96] Jens-Steffen Gutmann. *Vergleich von Algorithmen zur Selbstlokalisierung eines mobilen Roboters*. 1996. Diplomarbeit an der Universität Ulm http://www.informatik.uni-freiburg.de/~gutmann/.

[Hab91] Peter Haberäcker. *Digitale Bildverarbeitung*. Hanser-Verlag, 1991.

[Hab95] Peter Haberäcker. *Praxis der Digitalen Bildverarbeitung und Mustererkennung*. Hanser-Verlag, 1995.

[Hei99] Jochen Heinsohn. *Wissensverarbeitung*. Spektrum Akad. Verlag, 1999.

[IDS00] *Skript zur Vorlesung Robotik I*. 2000. Interaktive Diagnose- und Servicesysteme (IDS) - http://gate.fzi.de/divisions/ipt/vorlesungen/vorlesungen_d.htm pdf.

[BHJ00] I.Boersch, J.Heinsohn, K.-H.Jänicke, H.Loose, F.Mündemann, und H.Kanthack. *Vorlesung „Applikationen Intelligenter Systeme„*. 2000. Lehrveranstaltungsbegleitendes Material, FH Brandenburg, SS 00.

[Ima01] Image Integration. *Image++ 2.1*. 2001. http://www.image-integration.com.

[Int01a] Intel. *Intel Image Processing Library*. 2001. http://developer.intel.com/software/products/perflib/ipl/.

[Int01b] Intel. *Open Source Computer Vision Library*. 2001. http://developer.intel.com/software/opensource/cvfl/index.htm.

[Jäh97] Bernd Jähne. *Digitale Bildverarbeitung*. Springer-Verlag, 1997.

[Jun00] Matthias Jung. *Agentenbasiertes Einsammeln verschiedenfarbiger Objekte*. 2000. Studienarbeit an der FH Hamburg - http://fbi010.informatik.fh-hamburg.de/~pioneer/Projekte/projekte.html.

[Koe00] Jana Koehler. *Handlungsplanung - Ein erster Einstieg*. 2000. UNI Freiburg, http://www.informatik.uni-freiburg.de/~koehler/was-ist-planen.html.

[Kon96] Kurt Konolige. *The Saphira Architecture for Autonomous Mobile Robots*. 1996. http://www.ai.sri.com/~konolige/saphira/index.html pdf.

[Kon99] Kurt Konolige. *COLBERT: A Language for Reactive Control in Sapphira*. 1999. http://www.ai.sri.com/~konolige/saphira/index.html pdf.

[Mül98] Michael Müller. *Navigation von autonomen mobilen Robotern*. 1998. Literatur zum Hauptseminar *Mobile Roboter - Schwerpunkt Navigation*, TU München, http://www.informatik.tu-muenchen.de/~muellemi/amr/index.html.

[Pac98] René Pachernegg. *3-Layer Architektur*. 1998. Seminarvortrag zu *Lernalgorithmen in der Robotik*, TU Graz, http://www.cis.tugraz.at/igi/STIB/WS98/Pachernegg/3LayerArchitektur.html.

[SRD98] *IEEE and Fraunhofer IPA Database on Service Robots*. 1998. http://www.ipa.fhg.de/300/320/srdatabase/index.html.

[Ste93] Rainer Steinbrecher. *Bildverarbeitung in der Praxis*. Oldenbourg, 1993.

[Ste00] *Grundlagen der Steuerung mobiler Roboter*. 2000. Praktikum, TU Ilmenau, http://www.systemtechnik.tu-ilmenau.de/~fg_sa/praktika/wbs2/kapitel2.html.

[Vra00] André Vratislavsky. *Reaktive Verhaltenssteuerung*. 2000. Referat für das Seminar Robotik WS 99/00, FU Berlin - http://www.inf.fu-berlin.de/~vratisla/Robotik/ReaktivesVerhalten.htm.

Erklärung

Ich versichere, die vorliegende Arbeit allein und nur unter Zuhilfenahme der im Literaturverzeichnis genannten Quellen angefertigt zu haben.

Oliver Sachse
Brandenburg, den 15. Januar 2001