Diplomarbeit

im Rahmen des Forschungsprojektes Mobot-IIIC

Explorationsstrategien für autonome mobile Roboter in Indoor-Umgebungen

Entwurf und Implementierung der Explorationskomponente von Mobot-III

Jörg Roth

Dezember 1991

Bearbeitet von: Jörg Roth

Auf der Vogelweide 1

6750 Kaiserslautern

Betreuung der Diplomarbeit: Dipl.-Inform. Thomas Edlinger
Universität Kaiserslautern, Fachbereich Informatik

Anschrift: Universität Kaiserslautern
Fachbereich Informatik
AG Prozeßrechentechnik
Prof. Dr. rer. nat. E. von Puttkamer
Postfach 3049

6750 Kaiserslautern

Vorwort

Die vorliegende Diplomarbeit entstand im Rahmen des MOBOT-III-C Forschungsprojekts in der AG von Puttkamer des Fachbereichs Informatik an der Universität Kaiserslautern.

MOBOT-III-C ist eine Weiterentwicklung des autonomen mobilen Roboters MOBOT-III. Das Ziel dieser Weiterentwicklung stellt eine in Serie produzierbare Reinigungsmaschine dar. Die Konzeption des Gesamtsystems geht von einer Anwendung innerhalb von Gebäuden aus. Bei der Entwicklung standen vor allem die Autonomie und Explorationsfähigkeit des Fahrzeugs im Vordergrund.

Das Fahrzeug soll sich ohne zusätzliche Hilfsmittel, wie Induktionsschleifen oder spezielle Navigationspunkte in seinem Einsatzgebiet selbständig fortbewegen können. Dazu soll der AMR mit Hilfe eines Multisensorsystems seine Umwelt erkunden und die Reinigung ohne externe Unterstützung ausführen.

Bedanken möchte ich mich für die zahlreichen Anregungen und weiterführenden Diskussionen bei

Herrn Professor Dr. rer. nat. Ewald von Puttkamer,

Herrn Dipl.-Inform. Thomas Edlinger,

Herrn Dipl.-Inform. Torsten Ramsbott,

Herrn Uwe Roth,

Herrn Volker Böhm.

Kaiserslautern,
im Dezember 1991

Jörg Roth

Inhalt

1. Einleitung und Motivation

Im folgenden soll eine kleine Einführung in das Gebiet autonomer mobiler Roboter gegeben werden, um dann allgemeine grundlegende Aspekte der Exploration zu diskutieren.

1.1 Einleitung

Die Aufgabe eines **autonomen mobilen Roboters** (AMR) ist es, sich selbständig in einer zunächst unbekannten Umgebung zu bewegen und dabei schrittweise seine Umgebung zu erkunden, um darin einfache Aufgaben zu erfüllen.

Der in Kaiserslautern entwickelte Roboter Mobot-III wurde konzipiert, um mit seinen Sensoren die Umwelt zu kartographieren und Transport- oder Reinigungsaufgaben zu erfüllen. Als Einsatzorte kommen hierbei Büroumgebungen oder Lagerhallen in Frage. Als Sensoren werden neben einem sogenannten Laserradar, das einen Rundumblick auf die Umwelt bietet und dabei Entfernungsdaten zurückliefert, Ultraschall und Infrarot eingesetzt.

Um die nötigen Programme vor dem Einsatz in einer realen Umgebung zu testen, wurde eine Simulationsumgebung geschaffen, die ein hinreichend genaues Modell der Sensorik und Mechanik darstellt.

Die vorliegende Arbeit befaßt sich mit der Exploration einer unbekannten Umgebung, sowie eine für Reinigungs- oder Transportaufgaben geeignete Kartographierung.

1.2 Motivation

Exploration im Kontext eines AMR ist die selbständige Erfassung und Kartographierung der Umwelt. Die Exploration wird normalerweise der Aktionsplanung eines Roboters zugeordnet, da sie eine abstrakte Aufgabe darstellt, die selbständig in kleine Teilaufgaben zerlegt werden muß.

Der Begriff "Exploration" selbst ist unbestimmt, solange die Begriffe "Erfassung der Umwelt" und "Kartographierung" nicht hinreichend genau definiert sind. Somit gibt es nicht *die* Exploration der Umwelt, sondern nur verschiedene Möglichkeiten, die alle auf die zu erfüllende Aufgabe zugeschnitten sein müssen. Die Umwelterfassung muß einen bestimmten Grad von Vollständigkeit erfüllen, die von der Problemstellung abhängt. Die entstehende Karte sollte hinreichend genau sein, und alle erwarteten Details enthalten. Umgekehrt sollten alle Objekte aus der Karte auch eine Entsprechung in der realen Welt haben. Die Karte, die produziert wird, sollte so aussehen, daß die zu erfüllende Aufgabe günstig zu bewältigen ist.

Die Exploration besteht abstrakt aus den Teilen **Wegeplanung** und **Kartographierung**. Die Planung ist dann abgeschlossen, wenn die Karte den vorgegebenen Grad der Vollständigkeit besitzt.

Verschiedene Anforderungen an die Karte ziehen verschiedene Arten der Exploration mit sich. Kriterien, wann eine Karte als vollständig betrachtet wird, könnten dabei folgende sein:

- Alle Objekte müssen so genau erfaßt worden sein, daß eine Objekterkennung erfolgreich sein kann
- Alle Objekte müssen mit einer ausreichenden Höhenschichtinformation versehen sein, so daß zum Beispiel die dreidimensionale Bewegung eines Greifarms kollisionsfrei erfolgen kann
- Alle Objekte müssen so erkannt worden sein, daß eine Planung von Reinigungsbahnen erfolgen kann
- Alle Objekte müssen in ihren Ausmaßen so erkannt worden sein, daß ein AMR ohne Kollision hindurch fahren kann.

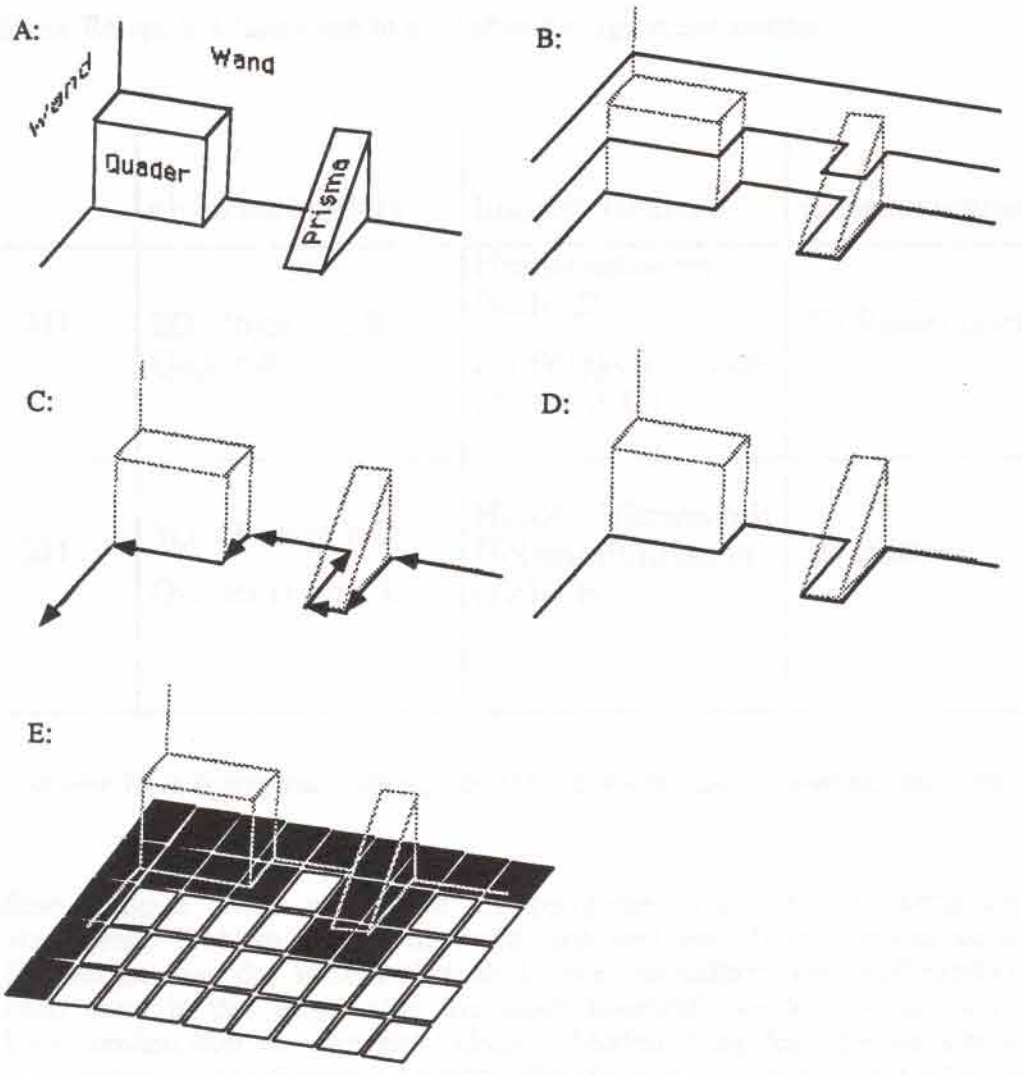


Bild 1.2.1: Möglichkeiten zur Umweltrepräsentation

Beispiele einer Umweltrepräsentation wären die folgenden:

- **A:** Die Karte enthält die vollständige 3D-Information aller Objekte (zum Beispiel als Menge von abstrakten Objekten wie z. B. Quader).
- **B:** Die Karte enthält die 3D-Informationen in Form von auf Höhenschichten projizierte Linien.
- **C:** Die Karte enthält 2D-Informationen in Form von geschlossenen Linienzügen.
- **D:** Die Karte enthält nur die 2D-Information der auf die Ebene projizierten Objekte in Form von einzelnen Linien.
- **E:** Die Karte enthält die 2D-Information in Form von gesetzten oder freien Rasterpunkten.

Diese Beispiele können nun in ein Schema eingeordnet werden:

	objektorientiert	**linienorientiert**	**rasterorientiert**
2D	2D Objekte z.B. Quadrate	Hindernislinien (siehe D) als Polygon verkettet (siehe C)	2D Raster (siehe E)
3D	3D Objekte z.B. Quader (siehe A)	Hindernislinien mit Höheninformation (siehe B)	3D Raster

Tabelle 1.2.2: Zusammenstellung von Möglichkeiten der Umweltrepräsentation

Eine Aufgabe ist es nun zuerst ein geeignetes Vollständigkeitskriterium für das vorliegende Problem zu finden, sowie eine geeignete Kartenstruktur zu definieren. Hiermit ist dann das Wissen über die Umwelt modelliert. Das Vollständigkeitskriterium kann in der Regel aber nur dann überprüft werden, wenn man auch die Unwissenheit über die Umwelt modelliert. Modellierung des Unwissens bedeutet hier, festzulegen, welche Bereiche, Objekte oder Flächen noch nicht von der Sensorik erfaßt wurden und somit noch zur Exploration anstehen.

Eng gekoppelt mit der Darstellung von Wissen und Unwissen ist, eine geeignete Strategie zu finden, die sicherstellt, daß die Karte auch effizient aufgebaut wird. Eine Strategie, die den optimalen Aufbau garantiert, wird auch einen sehr hohen Rechenaufwand verlangen, weswegen ein Einsatz in einer Echtzeitumgebung nicht

praktikabel ist. Es reicht aber in der Regel ein "einigermaßen" effizientes Vorgehen aus, so daß man an geeigneten Heuristiken interessiert ist, die den Ablauf steuern.

Alle genannten Begriffe sind noch einmal in folgendem Bild wiederzufinden.

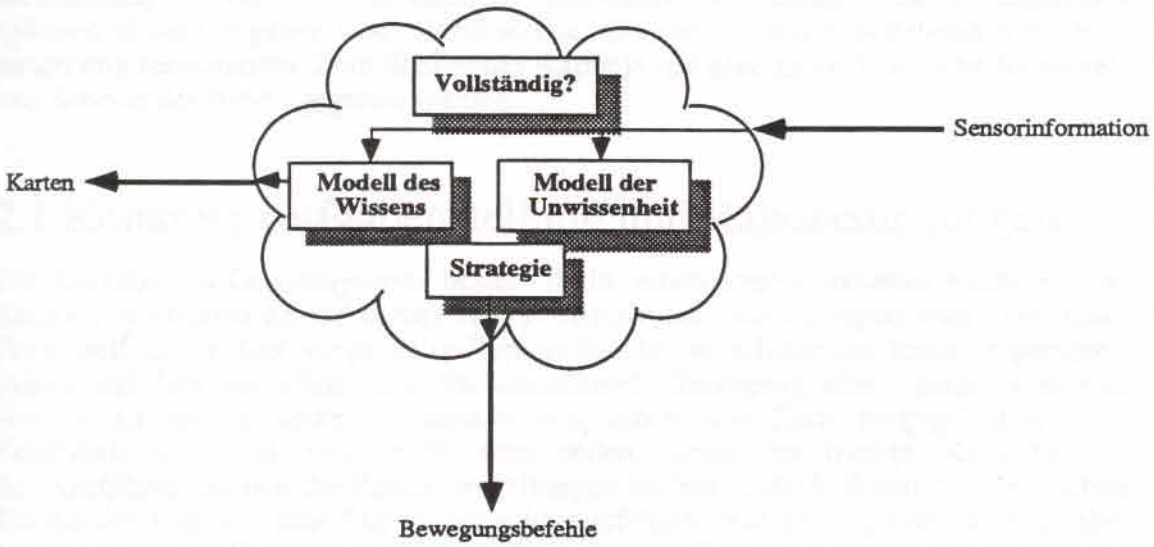


Bild 1.2.3: Allgemeines Vorgehen beim Explorieren

Ein AMR wird so, durch die Strategie gesteuert, geeignete Bewegungsmanöver durchführen und damit seine Umgebung solange erkunden, bis das Vollständigkeitskriterium erfüllt ist. Hierbei baut er sich eine Karte auf, die dann an die weiterverarbeitenden Komponenten übergeben wird.

Es gibt viele Gründe, warum ein autonomer Roboter seine Umgebung selbst erkunden und nicht eine vorgefertigte Karte benutzen sollte. Ein Grund liegt sicher darin, daß eine fertige Karte schnell veraltet und damit unbrauchbar wird. Gerade in einer Büroumgebung oder in einer Lagerhalle ändert sich die Konfiguration ständig. Somit müßte der AMR ständig die Differenz zwischen innerer Karte und realer Umgebung erfassen und darauf entsprechend reagieren. Diese Erfassung wäre aber fast schon eine Exploration, so daß man aus diesem Grund schon völlig auf eine eingegebene Karte verzichten kann.

Ein anderer Grund ist, daß sich die verarbeitenden Algorithmen sehr nahe an die Sensorkonfiguration anlehnen. Somit ist eine Karte, die schon bestimmten Randbedingungen genügt sicher wertvoller als eine Karte, die von außen eingegeben, erst in ein brauchbares Format übersetzt werden muß.

Ein letzter, nicht zu verachtender Grund ist, daß ein AMR, der sich seine Karte selbst aufbaut, optimal gegen Bedienfehler ausgerüstet ist.

2. Spezifikation

Es wird sich nun von der allgemeinen Form gelöst, um sich der hier vorliegenden Problemstellung zu widmen. Von Interesse sind dabei Randbedingungen, die durch den späteren Einsatz gegeben sind, sowie solche, die von der schon bestehenden Systemumgebung herstammen. Zum Schluß des Kapitels soll eine grobe Übersicht der einzelnen Schritte der Arbeit gegeben werden.

2.1 Konkrete Aufgabenstellung und Randbedingungen

Die Aufgabe des Gesamtsystems besteht darin, einen abgeschlossenen Komplex von Räumen zu explorieren und darin einen Transport- oder Reinigungsauftrag zu erfüllen. Prinzipiell ist die hier vorgestellte Exploration für verschiedenen Einsätze geeignet, jedoch soll hier vor allem eine flächendeckende Reinigung aller Räume betrachtet werden. Es soll zu keinem Zeitpunkt von außen eine Karte eingegeben werden. Zusätzlich hat man sich dafür entschieden, keine langfristige Kartenhaltung durchzuführen, da sich die Einsatzumgebungen laufend ändern. Somit muß vor *jedem* Einsatz des Roboters eine Explorationsphase erfolgen. Auf die Exploration muß dabei besonders großen Wert gelegt werden, da die interne Repräsentation nach der Exploration nicht mehr korrigiert wird. Aus diesem Grund darf sich während oder nach der Exploration auch nichts mehr an der Umgebung ändern. Es ist zwar sichergestellt, daß sich keine Kollision mit unvorhergesehenen Hindernissen ereignet, jedoch kann es dann passieren, daß der Auftrag des Roboters nicht mehr ordnungsgemäß abgewickelt werden kann, da eine zu große Differenz zwischen Welt und Weltmodell besteht.

An die Einsatzumgebungen werden keine Bedingungen gestellt, außer daß es sich um abgeschlossene Gebäudeumgebungen handelt. Es wird aber weder eine bestimmte Geometrie der Räume (z.B. nur rechteckige Räume) vorausgesetzt, noch werden Markierungen zur Orientierung (Landmarken) erwartet.

Die Exploration und die Reinigung sollen sich abwechseln, und zwar in jedem *Teilraum* ein Mal, wobei noch zu klären ist, was hier unter "Teilraum" verstanden wird. Somit zerfällt die Exploration logisch in zwei Teile: die **lokale** und die **globale Exploration**. Während der Exploration soll noch keine Reinigung erfolgen, da die Fahrbewegungen hierfür keine Optimierung zulassen. Außerdem wäre der Vorteil für die Reinigung nur gering, da der Sensor eine wesentlich größere Reichweite besitzt als der Reinigungsapparat und somit die Fahrbewegungen nur eine geringe Fläche erreichen würde.

Die Exploration hat als eine höhere Funktion des Gesamtsystems eine abstrakte Sicht auf die Sensordaten. Diese werden nach einer ersten Abstraktion und Korrelation als eine Menge von Linieninformationen geliefert. Dementsprechend verläßlich sind auch diese Eingabeinformationen. Sie werden deshalb erst einmal als korrekt angesehen und nur nach groben Inkonsistenzen in Frage gestellt. Obwohl der Entfernungssensor auch die Höheninformation eines Meßpunkts bereitstellt, wurde darauf verzichtet diesen mitzuverarbeiten. Sowohl für die Reinigungsphase als auch für die Navigation ist eine auf die Ebene projizierte Darstellung der Daten geeigneter.

Entsprechend abstrakt wie die Sensorik stellt sich die Befehlsschnittstelle zur Steuerung des Bewegungsapparats dar. Fahrbefehle werden in Form von Vektoren an einen *Navigator* gegeben, der aufgrund der bis dahin gewonnenen Umgebungsdaten

den günstigsten Weg plant. Letztlich gibt es eine noch tiefere Komponente, den *Piloten*, der kleine Umfahrungen von unvorhergesehenen (z.B. dynamischen) Objekten vornimmt. Gibt der Explorator einen Fahrbefehl, so bekommt er eine Rückmeldung der Form "Ziel erreicht" oder "Ziel kann nicht angefahren werden" zurück. Auch diese Information kann zur Exploration genutzt werden.

Da der Pilot das Umfahren kleiner Objekte autonom vornimmt, wurde entschieden, kleine Objekte nicht in die Karte aufzunehmen (es sei denn, sie können nicht umfahren werden). Das bringt den Vorteil, daß die Reinigungsphase mit einem geringeren Datenaufkommen zu rechnen hat.

Die Exploration hat dafür zu sorgen, daß die erkundete Umgebung in eine für die Reinigungsphase aufbereitete Form gebracht wird. Die gewünschte Form ist eine Menge von geschlossenen Polygonzügen, die den Freiraum vom nicht befahrbaren Raum trennen. Die Orientierung bestimmt dabei das "Innen" und "Außen". Wichtig bei diesem Punkt ist, daß alle Polygone geschlossen sein müssen, damit die Reinigungsplanung darauf arbeiten kann. Dieser Punkt erwies sich als bestimmend für diese Arbeit. Ein erster Vorschlag, bei dem der Explorator eine Linienkarte an eine Zwischenkomponente liefert, die daraus geschlossene Polygonzüge erstellt, wurde verworfen. Das Problem das dabei auftritt, ist, daß viele Linien, die zum Schießen der Polygone nötig sind, vom Sensor nicht eingesehen werden können. Der Explorator ist somit auf "Vermutungen" angewiesen, Polygone sinnvoll zu vervollständigen. Er muß dabei auf Informationen zugreifen, die eine Zwischenkomponente nicht mehr zur Verfügung hat, z. B. welche Bereiche schon befahren wurden. Außerdem kann nur der Explorator "eingehendere Untersuchungen" anordnen, falls ein Polygon noch nicht sinnvoll geschlossen werden kann. Somit obliegt es dem Explorator alleine, diese Karte zu produzieren. Eines der Hauptprobleme dieser Arbeit war es dabei in der Tat zu garantieren, daß immer sinnvoll geschlossene Polygone an die Reinigung übergeben werden.

An das Programm selbst werden folgende Anforderungen gestellt:

- **Zeit:**

  Die Anforderungen an die Rechengeschwindigkeiten sind hier weniger streng zu sehen, als bei Anwendungen direkt am Sensor oder direkt an der Bewegungssteuerung. Ein Grund dafür ist, daß die Eingabedaten schon sehr verdichtet vorliegen und die Verarbeitung somit effektiver gestaltet werden kann. Ein anderer ist, daß die Eingaben nicht unbedingt schritthaltend verarbeitet werden müssen. Da sich die Karten von Aufnahme zu Aufnahme nur relativ wenig ändern, fällt es kaum ins Gewicht, wenn eine Aufnahme übergangen wurde. Zusätzlich kann man sich vorstellen, daß man die Fahrgeschwindigkeit adaptiert, sollte eine Datenüberflutung stattfinden.

- **Speicher:**

  Hier sind keine Probleme zu erwarten, da alle Karten die anfallen (also auch die für Zwischenberechnungen) sehr kompakt dargestellt werden können. Auf dem Zielrechner stehen einige MByte zur Verfügung. Von daher ist ein Speicherüberlauf nicht zu erwarten.

- **Robustheit:**

  Diesem Punkt muß einige Aufmerksamkeit geschenkt werden. Ziel ist es ein

Programm zu entwickeln, daß auf der Basis unvollständiger Informationen grundsätzlich brauchbare Ergebnisse liefert. Ein Algorithmus, der nicht unbedingt die optimale Lösung liefert, aber dafür immer eine fast optimale ist einem Algorithmus vorzuziehen, der häufig optimal arbeitet, dafür aber in sehr seltenen Fällen völlig falsch liegt.

- **Genauigkeit:**

  Auf die Genauigkeit muß keinen übertriebenen Wert gelegt werden. Da die Sensorik nur bis auf mm auflöst, ist dies auch die Untergrenze für Berechnungen. Um jedoch Rundungen zu vermeiden, die bei der Verwendung von Integerzahlen auftreten, wurde sich einheitlich für ein 4-Byte-Real Format entschieden. Da die Zielhardware mit numerischem Coprozessor ausgestattet ist, ist diese Lösung vertretbar.

2.2 Beschreibung der Systemkomponenten

Die Exploration ist in ein System fertiger Komponenten eingebunden, die im Mobot-III Projekt schon entwickelt wurden. Das System trägt dem Namen *Mobot-III-CS* (Mobot-III control structure) und umfaßt zur Zeit folgende Komponenten:

- **PFE** (Primary Feature Extraction):

  Zusammenfassung der Sensorrohdaten zu abstrakten Linieninformationen und statistische Auswertung

- **CCG** (Correlator, Cartographer, Geographer):

  Korrektur des Positionsfehlers, Verknüpfung der verschiedenen Sensorinformationen, langfristige Kartographierung

- **Topographer:**

  Erkennen abstrakter Objekte, Anlegen einer topologischen Karte

- **Navigator:**

  Entgegennahme komplexer Fahrbefehle, selbständige Suche nach Fahrweg unter Berücksichtigung der Befahrbarkeit

- **Pilot:**

  Entgegennahme einfacher Fahrbefehle, bestenfalls Umfahren kleiner Hindernisse

Hierauf werden nun die Komponenten **Exploration** EEX (Environment Explorer) und **Reinigungsphase** CCP (Cleaning Course Planner) aufgesetzt. Obenauf steht noch eine Komponente "Supervisor", die eine Initialisierung vornimmt und den Ablauf steuert.

Das System aller Komponenten von Mobot-III-CS stellt sich somit wie folgt dar:

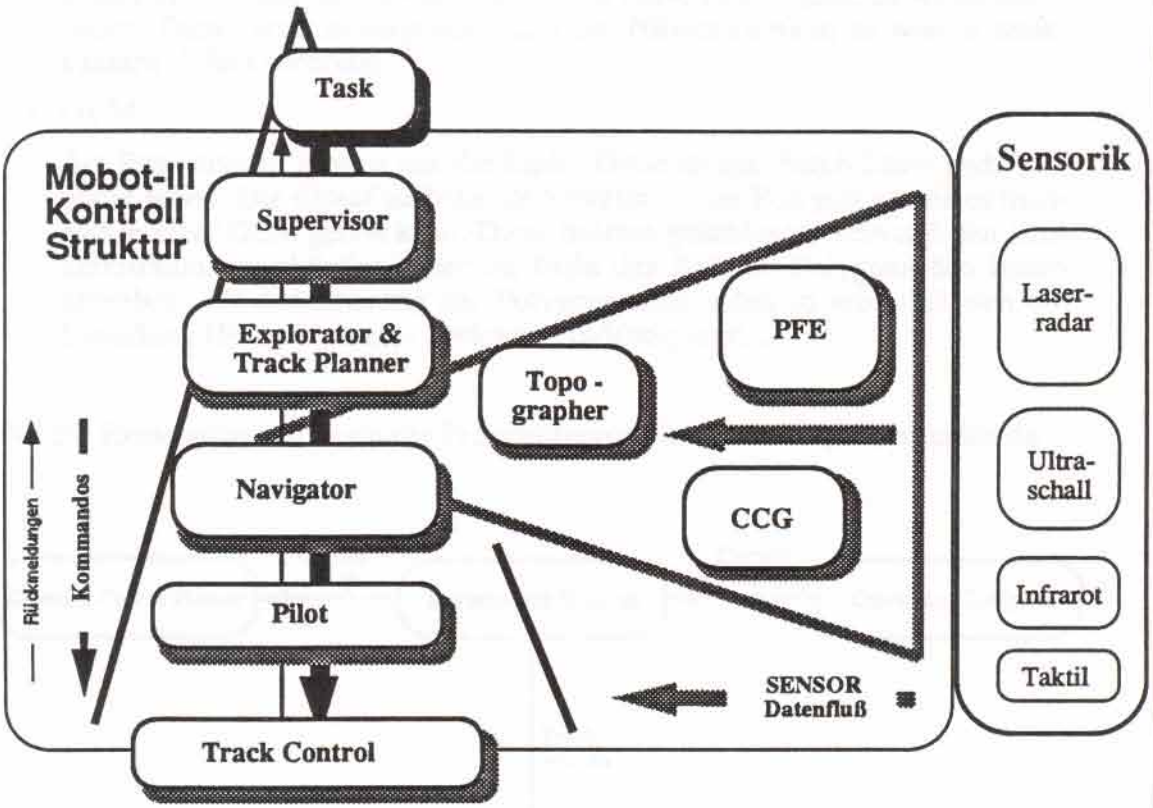


Bild 2.2.1: Systemkomponenten von Mobot-III-CS

Für die Exploration interessieren nur die Komponenten CCG, Navigator und CCP. Von CCG wird die aktuelle, von Meß- und Positionsfehlern bereinigte Karte CSM (Current Sensor Map) eingelesen. An den Navigator werden die Fahrbefehle (Vektors) abgegeben. CCP erhält letztlich die fertige Karte OLM (Outline Map), auf der er Reinigungsbahnen berechnen kann. Die Datenstrukturen beinhalten dabei im einzelnen:

- **CSM:**

  Eine Menge von Objekten, die jeweils durch Anfangs- und Endpunkt gegeben sind und entweder *Linie* oder *Cluster* sind. Cluster sind kleine Punktgruppen, denen man keine Richtung zuordnen kann. Die Objekte sind so abgelegt, daß man erkennen kann von welcher Seite sie aufgenommen sind, also wo sich der Frei- und wo der Hindernisraum befindet.

- **Track Vectors:**

  Gegeben wird ein Vektor (Ort und Richtung) der die aktuelle Position wiedergibt, und ein Vektor, der die gewünschte neue Position darstellt. Zusätzlich wird die maximale Geschwindigkeit während der Fahrt mitgegeben. Da man gerade bei der Exploration auch während der Fahrt die Kontrolle über das System besitzen möchte, besteht die Möglichkeit, daß der Navigator nicht die volle Distanz bis zum Ziel fährt, sondern auch nach einer kürzeren Entfernung die Kontrolle über das Fahrzeug zurückgibt. Der Navigator wird dann ersteinmal die Fahrt stoppen. Die Exploration kann dann entweder den

alten Befehl wiederholen, oder falls es ein besseres Ziel gibt, dieses anfahren lassen. Damit wird sichergestellt, daß der Navigator nicht zu weit in unbekanntes Gebiet vordringt.

- **OLM:**

  Als Basisstruktur gibt es nur die Linie. Diese ist nur durch Start- und Endpunkt geben. Die darauf aufbauende Struktur ist das Polygon wovon es mehrere in einer OLM geben kann. Diese müssen geschlossen sein und den Hindernisraum umschließen, oder im Falle des äußeren Polygons den Raum umgeben. Die Orientierung der Polygone muß dabei so sein, daß sich die Einteilung Hindernisraum - Freiraum eindeutig ergibt.

Für die Exploration stellt sich die Programmumgebung also folgendermaßen dar:

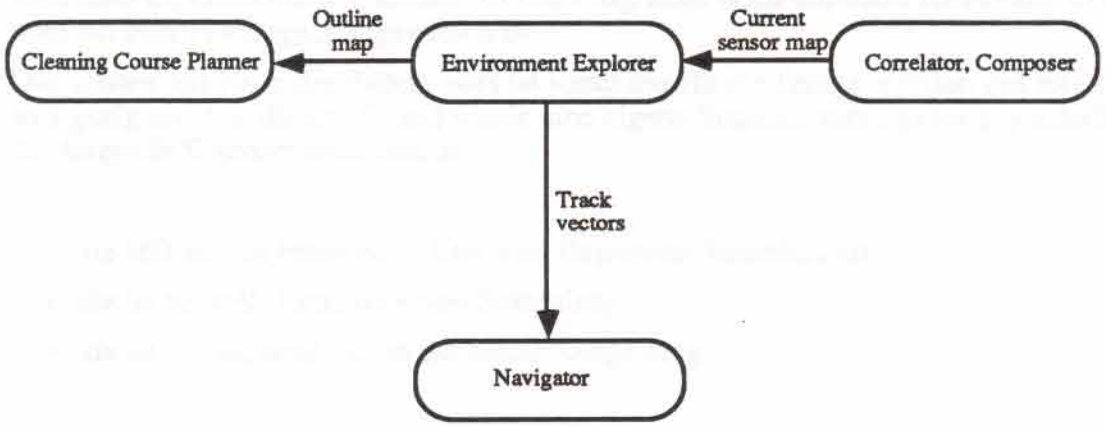


Bild 2.2.2: Programmumgebung des Explorators

2.3 Bereitstellung einer eigenen Simulationsumgebung

Zur Simulation des kompletten Zielsystems wurde eine Komponente *Supervisor* entwickelt, die ein Multiprozessorsystem auf einem einzigen Rechner simuliert. Hierbei werden die Sensoren, die Fahrbewegungen sowie die Einsatzumgebung von der Simulationsumgebung (*3D7*) simuliert und die Daten bzw. Befehle in Form von Dateien übergeben. Die weiterverarbeitenden Komponenten werden dann so nacheinander aufgerufen, daß es sich im Gesamtablauf wie ein Zusammenspiel paralleler Systeme darstellt. Der Supervisor wurde geschaffen, um den zu testenden Komponenten eine weitgehend realistische Umgebung ähnlich der Zielhardware zu vermitteln.

Im Gegensatz zu anderen Komponenten wie z.B. Reinigungsphase oder PFE, kann der Explorator nicht auf einen Satz fertiger Eingabedaten zurückgreifen, und nach der gesamten Berechnung Ausgabedaten liefern. Vielmehr produziert er schon während der Berechnung Ausgabedaten, nämlich Fahrbefehle, die zu neuen Eingabedaten führen, nämlich neue Karten. Da der Explorator die Schleife zwischen Sensordaten und Fahrbewegungen schließt, muß sich die Simulationsumgebung ständig auf neue Gegebenheiten einstellen können.

Die Umgebung des Supervisors für die Entwicklung des Explorators zu verwenden, erwies sich dabei als untragbar. Vor allem zwei Gründe sprachen dagegen:

- Nur fertige Applikationen können eingebunden werden. Die unerläßlichen Hilfsmittel beim Debuggen eines Programms sind so nicht zugreifbar.
- Die Zeiten für einen einzigen Simulationsschritt (von der Sensorsimulation bis zur Eingabekarte für die Exploration, sowie vom Navigator bis zur Fahrsimulation) spielt sich im Rahmen von Minuten ab. Wesentliche Gründe dafür sind die rechenintensive Simulation der Sensorik sowie der Ablauf auf einem einzigen Rechner, der dazu noch eine Fileschnittstelle besitzt.

Da die Exploration eines Raumes aus einigen hundert Einzelschritten besteht, ist die Form der Supervisorsimulation ungeeignet. Gerade in der Anfangsphase der Arbeit muß man experimentieren. Zudem ist das Programm noch teilweise fehlerhaft, so daß man auf einen Debugger angewiesen ist.

Das Testen mit Hilfe des Supervisors ist somit nur für die fertige Version des Explorators geeignet. Aus diesem Grund wurde eine eigene Simulationsumgebung geschaffen, die folgende Eigenschaften besitzt:

- sie läßt sich in Form einer Unit zum Explorator hinzubinden
- sie ist schnell (Bruchteile von Sekunden)
- sie ist hinreichend nah an der realen Umgebung

Die "kleine" Simulationsumgebung, die hierbei entstand, simuliert die Komponenten Laserradar, PFE sowie CCG auf der Sensorseite relativ gut. Hierbei wurde der Umweg über die Einzelpunkte bei der Radaraufnahme umgangen, und direkt auf die Linieninformationen zugegriffen, wie sie im Umgebungseditor erzeugt werden. Diese werden mit dem Sensorradius geschnitten und einem Sichtbarkeitstest unterzogen. Da hier für die Exploration nur eine Höhenschicht verlangt wurde, ist diese Methode ausreichend.

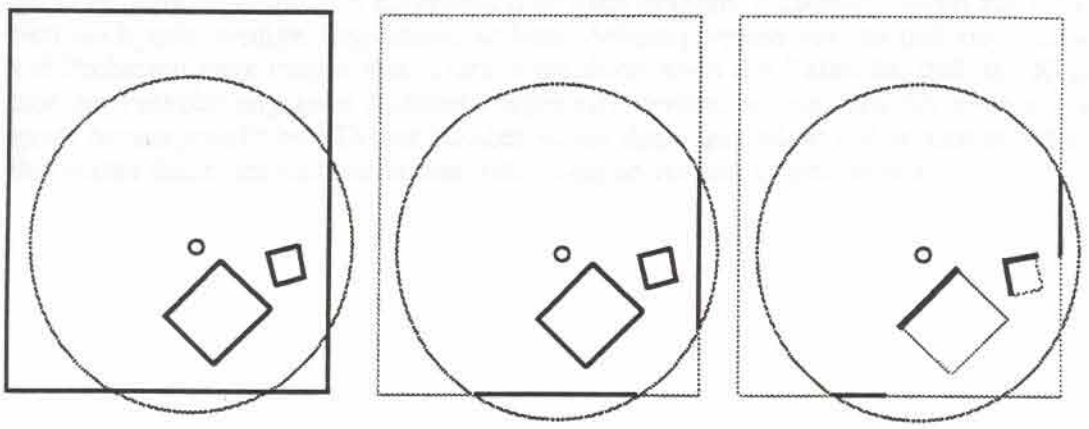


Bild 2.3.1: Simulationsschritte beim Scannen in der eigenen Simulationsumgebung

Die Simulation auf der Bewegungsseite gestaltete sich schwieriger. Die Navigation komplett zu simulieren erwies sich als zu kompliziert, so daß darauf verzichtet wurde. Vielmehr wird erst einmal davon ausgegangen, daß die angegebenen Fahrbefehle ohne Umweg in direkter Linie befahrbar sind. Dies führt zu dem Problem, daß eventuell Aufnahmen aus dem Inneren von Objekten gemacht werden können, die natürlich die Kartenhaltung in inkonsistente Zustände bringt. Um dieses Problem zumindest zu entschärfen, wurde in die Exploration ein Algorithmus aufgenommen, der Umfahrungen mit maximal einem Umfahrungspunkt selbst vornimmt. Wie wir später noch sehen werden, bringt dieser Algorithmus auch aus anderen Gründen Vorteile.

Um die Übertragung ins Zielsystem problemlos zu gestalten, wurde darauf geachtet, daß die Schnittstellen zu dieser Simulationsumgebung genauso gestaltet sind, wie zu den entsprechenden Komponenten.

2.4 Schritte der Arbeit

Die Arbeit umfaßt mehrere Schritte die teilweise mehrmals bearbeitet wurden:

1. Einarbeitung in das Programmsystem "MacApp" und Implementierung einer ersten einfachen Benutzeroberfläche zur Handhabung der Tests
2. Konzeption und Implementierung einer Simulationsumgebung
3. Konzeption und Implementierung einer Kartenfusionierung
4. Bereitstellung einer darauf aufbauenden lokalen Exploration (d.h. in einem Raum)
5. Bereitstellung einer globalen Exploration (d.h. von Raum zu Raum)
6. Nach intensivem Testen in der eigenen Simulationsumgebung übertragen auf die Simulationsumgebung unter dem Supervisor, dort nochmaliges Testen

Insbesondere die Schritte 3 und 4 erwiesen sich als sehr arbeitsintensiv, da die vorgegebenen Randbedingungen eingehalten werden mußten. Außerdem lagen zur Exploration noch sehr wenige Ergebnisse anderer Arbeitsgruppen vor, so daß man teilweise auf Probieren angewiesen war. Daraus resultiert auch die Tatsache, daß zur Konzeption der Feinplanung zwei Ansätze vorgestellt werden, wovon sich der erste als Sackgasse herausgestellt hat. Dieser ist aber schon deswegen nicht uninteressant, da er auf den ersten Blick der natürliche und naheliegende Ansatz zu sein scheint.

3. Konzeption

Als erstes soll eine Einführung in die Grundkonzeption gegeben werden, die jetzt auf die gegebene Aufgabenstellung zugeschnitten ist. Dann werden zwei Lösungsansätze verfolgt, und diese konkretisiert. Breiten Raum wird dabei eine Analyse der Vor- und Nachteile der Verfahren, auch im direkten Vergleich einnehmen. Die auftretenden Probleme werden dann klassifiziert, so daß man Grenzen der einzelnen Verfahren sowie der Machbarkeit überhaupt festlegen kann.

3.1 Konzeptioneller Überblick

Die Exploration stellt die Eingabe für die Reinigungsphase bereit. Im oben erwähnten Sinne, muß das Ziel der Exploration sein, eine für die Reinigungsphase optimale Karte zu liefern. Da die Planung der Reinigungsbahnen auf in sich abgeschlossenen Teilbereichen arbeiten soll, wurde ein polygonaler Ansatz bevorzugt. Die hier beschriebene Exploration läßt sich folgendermaßen einordnen:

- die auszugebende Karte besteht aus geschlossenen Polygonen, deren Umlaufsinn eindeutig befahrbaren Raum und Hindernisraum definiert.
- eine Exploration gilt als erfüllt, wenn die umgebende Raumstruktur sowie alle im Raum befindlichen Hindernisse durch geschlossene Polygone wiedergegeben werden.

Somit könnte eine Karte zu einem Raum folgendermaßen aussehen:

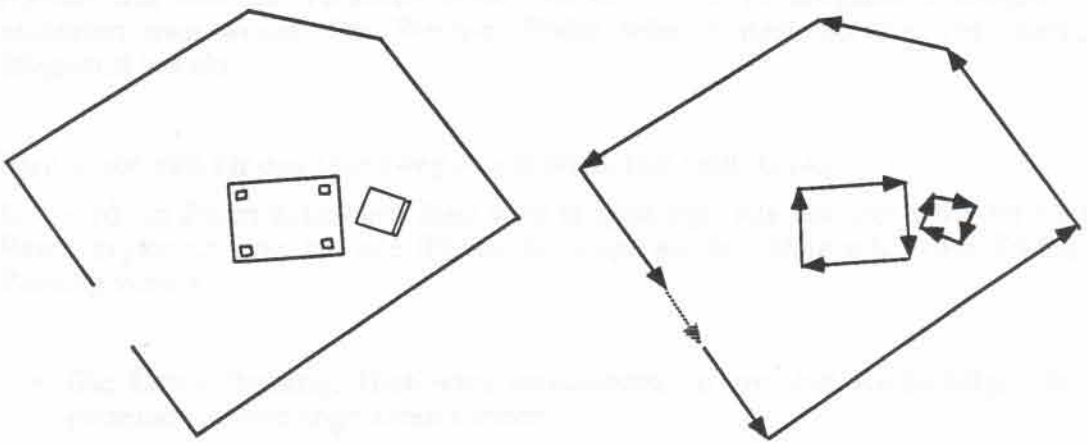


Bild 3.1.1: Raum mit generierter Karte

Der polygonale Ansatz bringt hier zwei Vorteile mit sich: zum einen legt die Sensorik einen solchen Ansatz nahe, da als Ausgabe nach der Abstraktion Linien vorliegen. Zum anderen vereinfacht eine so angefertigte Karte das Berechnen der Reinigungsbah-

nen, da zusammenhängende Linien schon zusammenhängend abgelegt sind und eine Definition von "Innen" und "Außen" vorliegt.

Zu definieren bleiben nun folgende Dinge:

- Wie werden Einzelkarten zu einer Gesamtkarte zusammengesetzt?
- Wie wird Unwissenheit modelliert?
- Welche Strategie wird angewendet, um aus Unwissen möglichst effizient Wissen zu machen?

Ein weiterer Punkt, der bis jetzt noch nicht betrachtet wurde ist die Häufigkeit, mit der fertige Karten erzeugt werden. Die Strategie läßt sich nicht losgelöst davon betrachten. Folgende Extreme sind möglich:

- Man exploriert zuerst die Welt komplett und gibt dann erst die Karte an weiterverarbeitende Komponenten.
- Man exploriert die Welt, gibt aber schon laufend Karten an weiterverarbeitende Komponenten weiter, so daß u.U. noch während der Exploration Aufgaben erledigt werden können.

In der Regel wird es noch einen Mittelweg geben:

- Man exploriert solange, bis ein Teilkriterium erfüllt ist.

Hierbei läßt man die Teilkarte weiterverarbeiten und Teilaufgaben erledigen, dann exploriert man weiter. Der Wechsel findet solange statt, bis die Vollständigkeit insgesamt besteht.

Hier wurde sich für den Mittelweg entschieden. Das heißt konkret:

Erst wird ein Raum exploriert, dann wird er gereinigt. Als nächstes wird der nächste Raum exploriert usw. bis alle Räume gereinigt wurden. Dies setzt zwei Stufen der Planung voraus:

- Die **Grob-Planung**: Hier wird entschieden, in welcher Reihenfolge die einzelnen Räume angefahren werden.
- Die **Fein-Planung**: Hier wird der Weg *in* einem Raum geplant.

Die Feinplanung selbst läßt sich in das Finden von Teilzielen unterteilen.

Eine analoge Einteilung kann man bei der Verarbeitung der Sensorinformation vornehmen. In einem Raum wird eine geometrische Karte angelegt, während man zwischen den Räumen eine topologische Karte verwendet. Diese Einteilung hat den Vorteil, daß man in einem Raum aufgrund von geometrischen Daten navigieren kann (was bei der begrenzten Datenmenge praktikabel ist), während auf dem Niveau der Räume auf einem topologischen Wegenetz die Planung vorgenommen wird. Die Suche beschränkt sich hier auf ein Graphensuchproblem während der Navigator mit rein geometrischen Karten überlastet wäre.

Die globale Übersicht über die notwendigen Komponenten der Exploration gibt das nächste Bild.

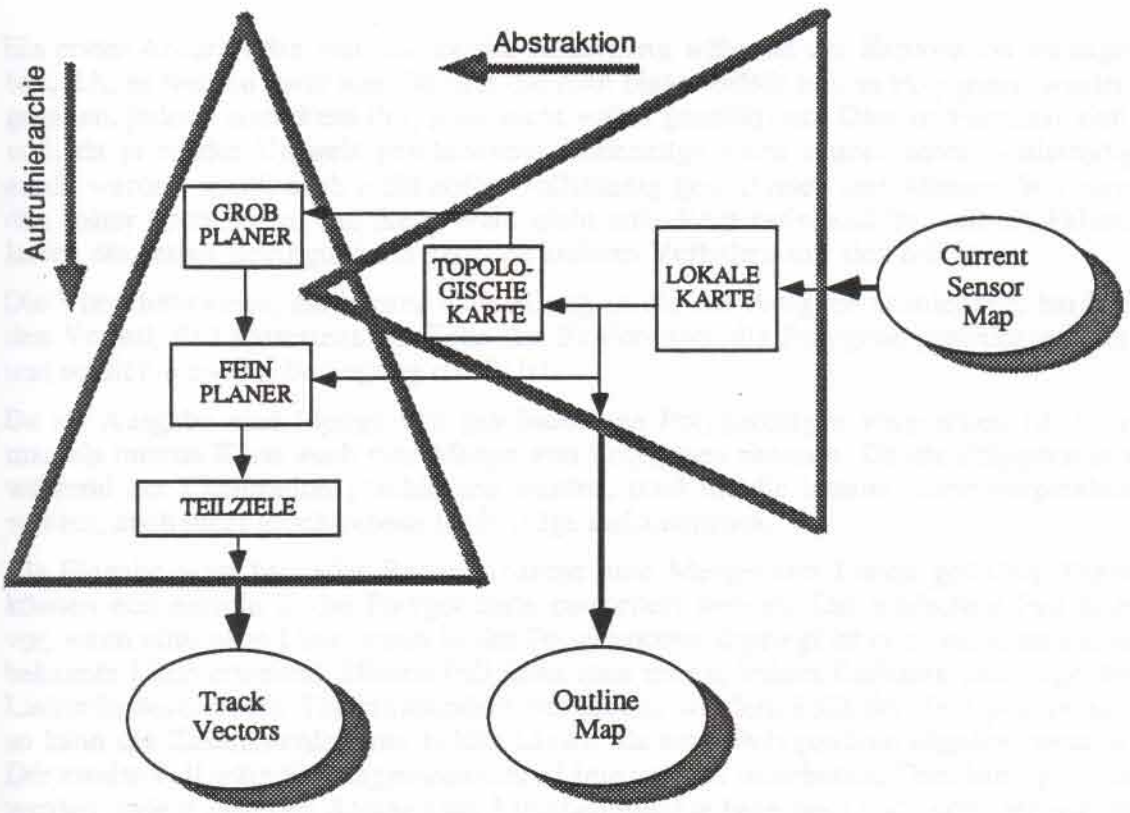


Bild 3.1.2: Übersicht der Komponenten der Exploration

3.2 Die Umweltkartographierung - ein erster Ansatz

Mit der Methode, wie die Umwelt kartographiert wird, legt man das Modell des Wissens und des Unwissens über die Umgebung fest. Da die Strategie der Wegeplanung auf diese Strukturen zurückgreift, wird das Verhalten des Roboters bei der Exploration durch die Kartographierung stark beeinflußt.

Es liegt sehr nahe, daß man die interne Karte, die während der Exploration benutzt wird, an die Kartenstruktur anlehnt, die man für die Ausgabe vorgesehen hat. Hierfür eignet sich eine Struktur, bei der eine der Vorbedingungen, die eine Ausgabekarte

erfüllen muß, aufgibt. Modelle, die durch Weglassen einer oder mehrerer Bedingungen entstehen, werden in der künstlichen Intelligenz **relaxierte Modelle** genannt. Diese haben verschiedene Vorteile. Zum einen lassen sie schon beschränkte Aussagen über die fertige Karte zu, so daß der Ablauf gezielt gesteuert werden kann. Zum anderen kann so exploriert werden, daß am Ende auch die vorerst zurückgestellte Bedingung erfüllt ist, somit kann eine Nachbearbeitung entfallen. Hiermit hat man für Exploration ein einfaches Grundprinzip gewonnen, das im folgenden diskutiert werden soll.

In unserem Fall waren die Bedingungen:

1: Alle Objekte müssen durch eigene Polygone repräsentiert werden.

2: Alle Polygone müssen geschlossen sein.

Ein erster Ansatz wäre nun, die zweite Bedingung während der Exploration aufzugeben, d.h. es werden zwar alle Objekte die man bisher erfaßt hat, in Polygonen wiedergegeben, jedoch sind diese Polygone nicht sofort geschlossen. Dies ist zunächst sinnvoll, da ja in der Umwelt geschlossene Linienzüge nicht immer sofort vollständig erfaßt werden, somit auch nicht sofort vollständig geschlossen sein können. Wir werden später noch sehen, daß diese Wahl nicht unbedingt zwingend ist, und ein Fallenlassen der ersten Bedingung ein gänzlich anderes Verhalten mit sich bringt.

Die Vorgehensweise, die interne Darstellung an die der Ausgabe anzulehnen, hat hier den Vorteil, daß spätestens am Ende der Exploration alle Polygone geschlossen sind und somit die zweite Bedingung erfüllt ist.

Da als Ausgabe eine Menge von geschlossenen Polygonzügen vorgesehen ist, kann man als interne Karte auch eine Menge von Polygonen nehmen. Da die Polygone erst während der Exploration geschlossen werden, muß für die interne Karte vorgesehen werden, auch nicht geschlossene Linienzüge aufzunehmen.

Als Eingabe wird bei jeder Radaraufnahme eine Menge von Linien geliefert. Diese können nun einzeln in die Polygonkarte einsortiert werden. Der einfachste Fall liegt vor, wenn eine neue Linie schon in der Polygonkarte abgelegt ist oder zumindest eine bekannte Linie erweitert. Diesen Fall kann man testen, indem Richtung und Lage der Linien in bestimmten Toleranzbändern verglichen werden. Fällt der Test positiv aus, so kann die Zusammenfassung beider Linien als neue Polygonlinie abgelegt werden. Der zweite Fall wäre ein Angrenzen einer Linie an eine bestehende. Dies kann getestet werden, indem man den Abstand der Linienendpunkte berechnet und auch hier wieder ein Toleranzband zugrundelegt. Fällt hier ein Test positiv aus, so kann ein Polygon um eine Kante erweitert werden. Für solche Polygone wird untersucht, ob sie sich schließen lassen.

Trifft keiner der beiden Fälle zu, so muß ein neues Polygon eingetragen werden, das nur diese Kante enthält. Somit hat der Kartographierungsalgorithmus folgendes Aussehen:

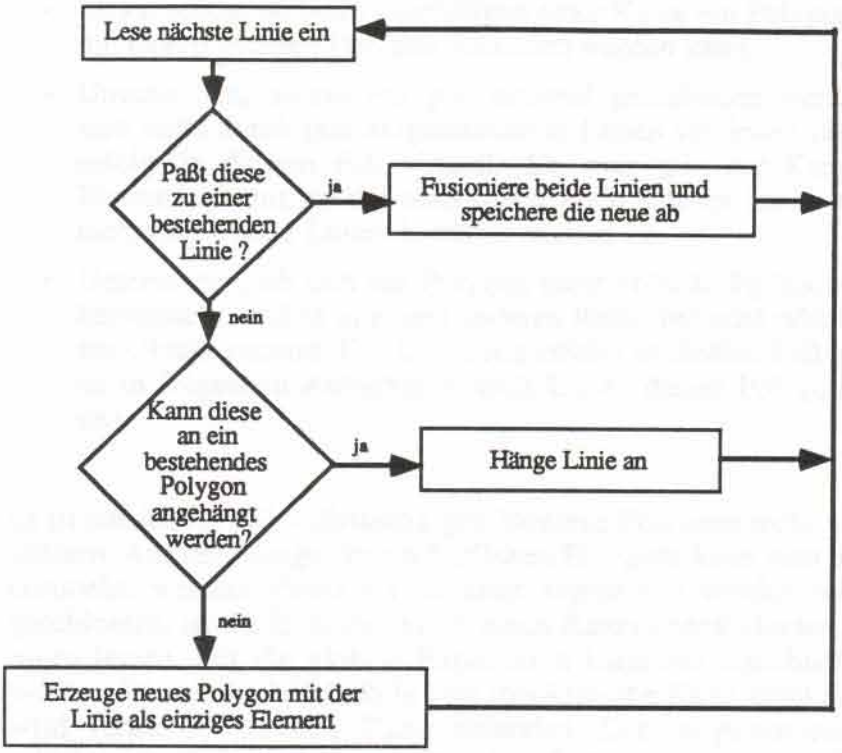


Bild 3.2.1: Vorschlag zur Fusionierung von Linien zu einer Gesamtkarte

Auf den ersten Blick scheint dieses Vorgehen vernünftig. Der zugrundeliegende Algorithmus ist sehr einfach und kann auch sehr effizient implementiert werden. Es ergeben sich jedoch folgende Probleme:

- Eine Linie kann unter Umständen mehreren Polygonen zugeordnet werden. Es müssen somit zusätzliche Kriterien zur Entscheidungsfindung herangezogen werden.
- Es kann vorkommen, daß ein Polygonzug ersteinmal auf verschiedene Teilpolygone verteilt abgelegt wird. Hier muß eine Fusionierung stattfinden.
- Es gibt Polygone, die trotz genauer Analyse nicht vollständig eingesehen werden können. Diese müssen sinnvoll geschlossen werden. Die verwendeten Linien, die das Schließen ermöglichen, sollen **virtuelle Linien** genannt werden. Im Gegensatz dazu sollen real aufgenommene Kante **massive Linien** genannt werden.
- Es gibt Linien, die der Sensor in anderen Räumen (z.B. durch eine Tür) eingesehen hat. Diese Polygonzüge müssen entfernt werden.

Aus diesen Gründen muß der Algorithmus um Funktionen erweitert werden, die wann immer sich die Polygone in einer bestimmten Weise ändern, versuchen die Karte zu verbessern. Die Funktionen werden hier **Dämonen** genannt. Folgende Funktionen müssen dabei mindestens ausgeführt werden:

- Untersuchen, ob beim Hinzufügen einer Kante ein Polygon geschlossen oder mit einem anderen Polygon fusioniert werden kann.
- Untersuchen, ob ein Polygon sinnvoll geschlossen werden kann, wenn es sich nicht durch real aufgenommene Linien schließen läßt. Die Schließung erfolgt in diesem Fall virtuell, d.h. man gibt der Kante eine bestimmte Eigenschaft mit, an der man später noch ablesen kann, daß die Schließung nicht durch reale Linien bestätigt worden ist.
- Untersuchen, ob sich ein Polygon nicht vollständig löschen läßt, wenn sich herausstellt, daß es in einem anderen Raum befindet oder von einem bewegten Objekt stammt. Die Löschung erfolgt in diesem Fall jedoch nur virtuell, da in folgenden Aufnahmen noch Linien dieses Polygons erscheinen können.

Es ist einsichtig, daß vollständig geschlossene Polygone nicht weiter exploriert werden müssen. Aus der Menge der noch offenen Polygone kann man nun nach einer Strategie ermitteln, welcher Punkt als nächstes angelaufen werden soll. Sind alle Polygone geschlossen, ist die Exploration für einen Raum abgeschlossen, und man kann ihn reinigen lassen. Für die globale Exploration kann das umschließende Polygon genutzt werden. Dieses wird deshalb in eine topologische Karte geschrieben. Zusätzlich hierzu wird vermerkt, wo sich Türen befanden. Der so gewonnene topologische Graph erlaubt eine einfachere Navigation als auf der geometrischen Ebene. Die resultierende Gesamtstruktur stellt sich folgendermaßen dar:

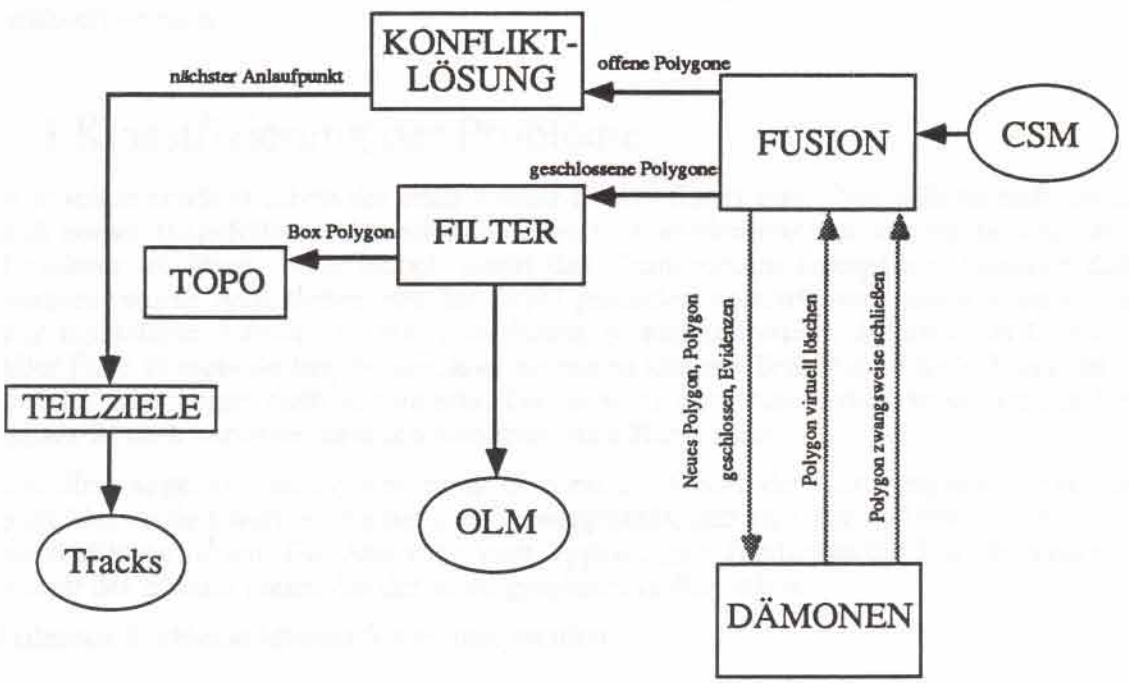


Bild 3.2.2: Versuch der Architektur

Die Topologie sowie die Konfliktlösung werden in der endgültigen Fassung genauer beschrieben. Genauere Betrachtung verdienen im Moment nur die Komponenten

Fusion und *Dämonen*. Die Fusion ist unmittelbar verständlich. Die Linien werden nach den genannten Verfahren in die Polygonkarten eingeschrieben. Probleme liefern die Dämonen, denn es ist nicht sofort klar, wann diese ihre Arbeit starten sollen. So ist es z.B. nicht genau definiert, wann ein offenes Polygon sich endgültig nicht durch real aufgenommene Kanten schließen läßt. Genauso kann man oft nicht während der Fahrt entscheiden, ob ein Polygon gelöscht werden soll. Aber gerade solche Polygone könnten als nächstes zur Untersuchung anstehen. All diese Ereignisse kann man unter dem Begriff **Evidenzen** zusammenfassen. Gesucht sind also zusätzliche Mechanismen, die Dämonen aktivieren. Beispielsweise könnte das wiederholte abscannen einer bestimmten Region eine Evidenz dafür sein, daß sich ein bestimmtes Polygon nicht durch real aufgenommene Linien schließen läßt. Somit ist der Dämon zu aktivieren, der ein Polygon virtuell schließt.

Die Problematik die sich jedoch stellt, ist, daß das Verhalten bei der Exploration von Erkenntnissen abhängt, die sich später als falsch herausstellen könnten. Es handelt sich hier also um nichtmonotone Erkenntnisgewinnung die bestimmte Revisionsmechanismen fordert. Während es aber bei reinen Datenmanipulationen möglich ist, diese zurückzunehmen, ist es unmöglich Aktionen des Roboters nicht geschehen zu machen. Eine Aktion des Roboters, die zu Problemen führen könnte, ist das Fahren in einen anderen Raum. Damit wäre die Randbedingung verletzt, daß der Roboter erst einen Raum zu Ende exploriert, bevor er sich dem nächsten Raum widmet. Zur Entschärfung, können die Dämonen so sicher wie möglich entworfen werden, jedoch bleibt das Kernproblem bestehen, daß aufgrund unvollständiger Daten Aktionen durchgeführt werden, die nicht wieder rückgängig gemacht werden können.

Auch wenn schon allein dieses Problem prinzipiell nicht lösbar ist, so ergaben sich noch zusätzliche Probleme mit dieser Art der Kartographierung, die im folgenden erläutert werden.

3.3 Klassifizierung der Probleme

Wie schon erwähnt führte der erste Ansatz in eine Sackgasse. Dies äußerte sich darin, daß immer ausgefeiltere Algorithmen entwickelt werden mußten, um die bestehenden Probleme zu lösen, ohne jedoch damit das Kernproblem anzugehen. Dadurch daß versucht wurde Ausnahmen von der Regel gesondert abzuarbeiten, handelte man sich nur zusätzliche Ausnahmen ein. Dies führte zu einem System, das zwar im Großteil aller Fälle akzeptable Ergebnisse, aber zu einem kleinen Teil absolut nicht brauchbare lieferte oder sogar nicht terminierte. Damit wird aber eine Grundvoraussetzung für dieses System verletzte, nämlich eine maximale Robustheit.

Die oben angedeuteten Probleme mit dem ersten Ansatz der Kartographierung sollen jetzt klassifiziert werden. Es hat sich herausgestellt, daß sich alle auf drei Grundtypen zurückführen lassen. Die Analyse dieser Typen ergibt fundamentale Schwierigkeiten, womit der Einsatz dieser Art der Kartographierung fraglich ist.

Folgende Probleme können festgestellt werden:

- **Zuordnungsproblem:**

  Objekte wie Linien und Punktgruppen werden eingegeben. Diese müssen nun auf vorhandene Polygone der internen Karte verteilt werden bzw. führen zu neuen Polygoneinträgen. Die Zuordnung muß dabei nicht immer eindeutig erfolgen, sondern führt u.U. zu zwei oder mehr Möglichkeiten. Um

jedoch auf einer neuen Karte aufbauend neue Entscheidungen zu treffen, kann die Wahl darüber, wo ein Objekt eingeordnet wird, nicht beliebig lange aufgeschoben werden. Die Entscheidung, die letztlich gefällt wird, kann dabei zu beliebig fehlerhaften Karten führen. Stellvertretend sollen zwei Beispiele besprochen werden:

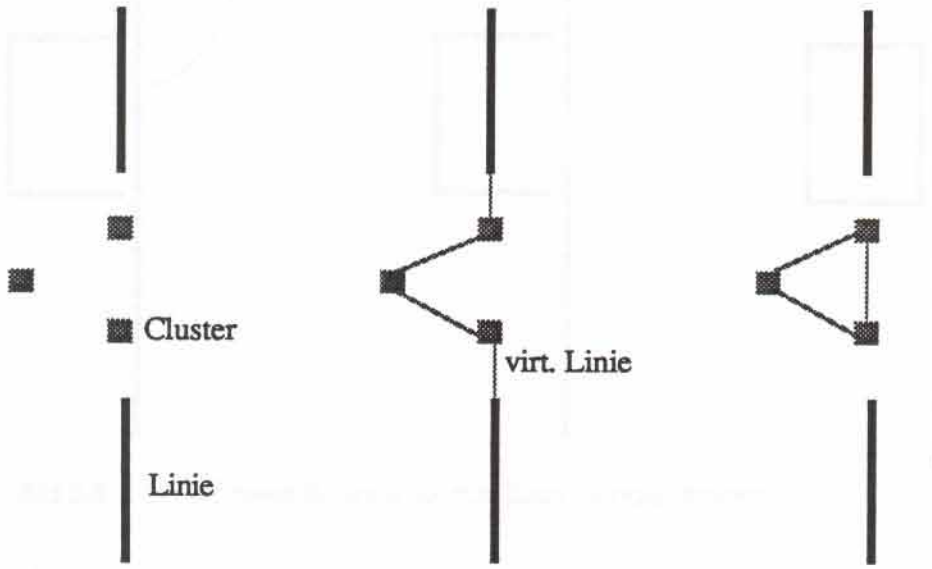


Bild 3.3.1: Ein Beispiel für das Zuordnungsproblem

Links sind die Objekte dargestellt, die dem System eingegeben wurden. Die Reihenfolge, mit der diese Informationen zugänglich gemacht werden, kann dabei beliebig sein. In der Mitte ist eine mögliche resultierende Konfiguration dargestellt. Hierbei wurden die Cluster durch virtuelle Linien sinnvoll verbunden, so daß dieser Bereich als erfaßt angesehen werden kann.

Eine andere Möglichkeit könnte aber auch die rechte sein, hervorgerufen beispielsweise durch eine andere Reihenfolge, mit der die Objekte in das System gegeben wurden. Hier wurden zuerst die Cluster miteinander verbunden. Dies erscheint beispielsweise dann sinnvoll, wenn die Linien zuerst gar nicht wahrgenommen wurden. Durch die Anreicherung der Clustergruppe mit virtuellen Linien, ist diese als exploriert anzusehen, da ein geschlossenes Polygon vorliegt. Werden nun die Linien in Betracht gezogen, so steht man vor dem Problem diese sinnvoll in die Gesamtkarte einzubinden. Hier bleiben nur die Möglichkeiten, entweder eine große virtuelle Linie einzufügen (was man sehr ungern macht, da dann vielleicht auch unzusammenhängende Bereiche plötzlich zusammenhängen können) oder man führt eine Rekonfiguration aller Polygone durch. Hierbei wäre dann aber unklar, wann diese durchzuführen ist. Außerdem handelt es sich dann um eine sehr aufwendige Funktion, da man prinzipiell zwischen allen zusammenhängenden Bereichen virtuelle Linien einfügen kann, was zu einer kombinatorischen Explosion führt.

Ein weiteres Beispiel für dieses Problem zeigt das folgende Bild:

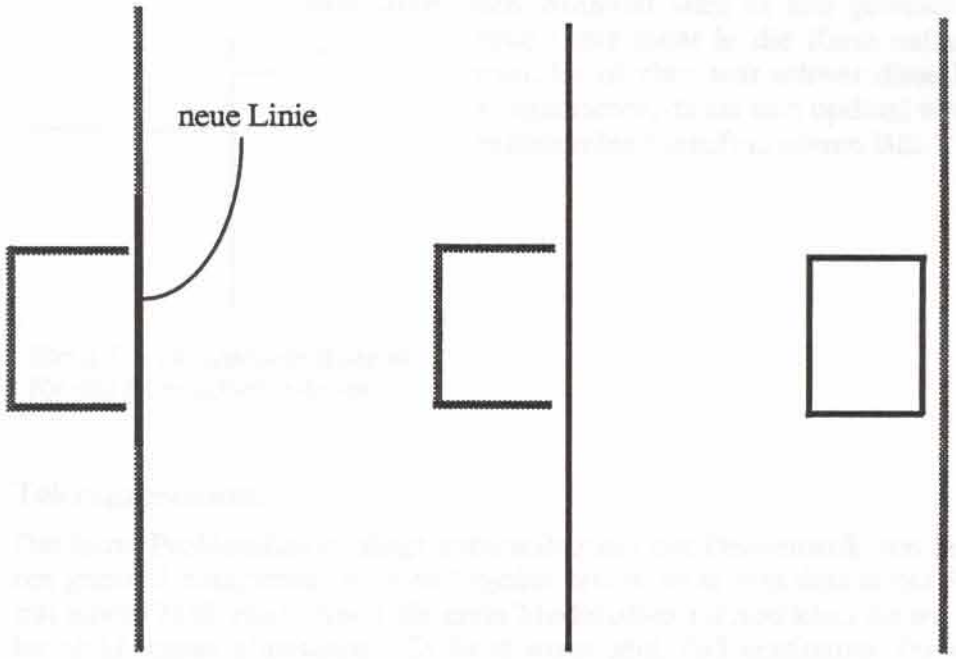


Bild 3.3.2: Ein weiteres Beispiel für das Zuordnungsproblem

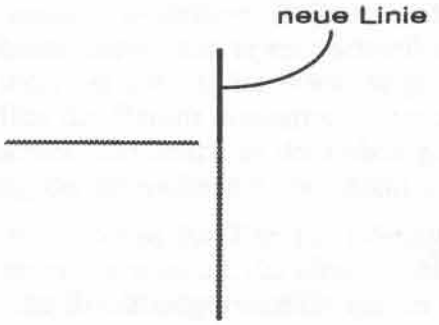
Links ist die interne Karte dargestellt, zu der die schwarze Linie hinzugefügt wird. Der Algorithmus wird daraufhin die Linie dort einsortieren, wo sie unbestritten am besten paßt, nämlich bei der großen Linie. Das Ergebnis ist in der Mitte zu sehen. Eine andere, jedoch wesentlich effektivere Variante wäre es jedoch, diese Linie zu verkürzen, und dem unvollständigen Rechteck zuzuordnen. Dies erfordert aber ein komplizierteres Verfahren beim Einsortieren, da auch Teilabschnitte zur Verfügung stehen und es so zu Linienresten kommt. Läßt man den Algorithmus jedoch unmodifiziert, so wird der unvollständige Polygonzug nie geschlossen, weil eine entsprechende Linie auch immer zur großen Linie paßt.

- **Kompetenzproblem:**

  Hierbei handelt es um ein Problem, das ausschließlich aus der Randbedingung erwächst, alle Teilräume separat zu explorieren. Nimmt der Sensor Objekte außerhalb eines Raumes auf, indem er beispielsweise durch eine geöffnete Tür oder ein offenes Fenster sieht, so entstehen dadurch meistens offene Polygone, also potentielle nächste Teilziele. Werden diese ausgewählt und angesteuert, so verläßt der Roboter den Raum. Dies wäre eine unerlaubte Aktion.

  Eine anderes Problem dieser Klasse zeigt das folgende Bild:

  Die grauen Linien stellen die schon gewonnene Karte dar. Es könnte sich dabei um aneinanderstoßende Wände handeln. In diesem Fall stoßen die Wände jedoch nicht exakt zusammen, so daß ein kleiner Spalt die Sicht auf eine Weiterführung der rechten Wand gibt. Aus einem steilen Winkel nimmt der Sensor jetzt die schwarze Linie auf. Obwohl die beiden Wände benachbart sind, können sie nun nur noch über eine virtuelle Linie verbunden wer-



den. Sinnvoll wäre es hier gewesen, die neue Linie nicht in die Karte aufzunehmen. Es ist aber sehr schwer diese Linie zu ignorieren, da sie sich optimal zu einer bestehenden hinzufusionieren läßt.

Bild 3.3.3: Ein weiteres Beispiel für das Kompetenzproblem

- **Toleranzproblem:**

  Die letzte Problemklasse hängt unmittelbar mit der Problematik von Sensoren generell zusammen. Werden Objekte erfaßt, so erfolgt dies in der Regel mit einem Meßfehler. Auch die erste Merkmalsextraktion kann diesen Fehler nicht immer eliminieren. Es kann sogar sein, daß bestimmte Verfahren bei der Merkmalsextraktionen erst Fehler verursachen. Deshalb müssen Vergleiche z.B. von Linien immer mit Toleranzen durchgeführt werden. Diese sind für die Sensorkonfiguration spezifisch ausgelegt, damit nicht unnötig viel Information verschenkt wird. Es kann aber dennoch sein, daß entweder dadurch Punkte oder Linien miteinander identifiziert werden, die nicht zusammengehören, oder daß zusammenhängende Teile nicht als solche erkannt werden. Das folgende Bild zeigt ein Beispiel für den zuletzt genannten Fall.

  Es werden zwei Linien aufgenommen, die unmittelbar aneinander angrenzen, doch entweder durch die Sensorfehler oder den Extraktionsalgorithmus liegen die Endpunkte der Linien nur mit einer räumlichen Differenz vor. Angenommen, es wird das Toleranzband nun so gelegt, daß die Entfernung der Endpunkte zu groß ist. Die Linien werden nun nicht benachbart abgelegt, sondern auf zwei Polygone verteilt. Sollte sich dieser Umstand nicht durch spätere Aufnahmen bessern, so bekommt man nicht geschlossene Polygone. Eine Lösung wären hier wieder Evidenzen, die ein Zwangsidentifizierung der Endpunkte erlauben. Damit wird das Problem jedoch nur verschoben, da man sich unter Umständen einhandelt, nicht zusammenhängende Teile zu verbinden. Ein anderer Ansatz ist es die Information darüber, ob Merkmale zusammenhängen direkt am Sensor zu detektieren. Dazu muß zum einen der Sensor gegebenenfalls modifiziert werden, zum anderen in den weiterverarbeitenden Komponenten diese Eigenschaft berechnet und weitergegeben werden.

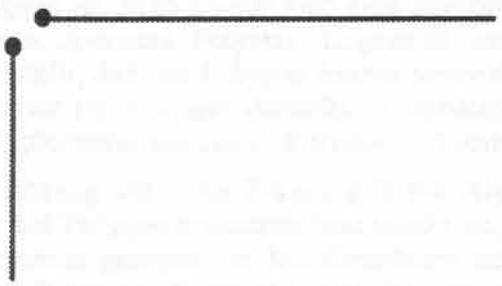


Bild 3.3.4: Ein Beispiel für das Toleranzproblem

Zusätzlich zu den genannten Problemen wurde auch noch nicht geklärt, wie man

Unwissenheit modelliert. Befindet sich beispielsweise der Roboter in einem sehr großen Raum, dann kann er sehr schnell seine Aufgabe erfüllen, alle Polygone zu schließen, indem er einmal der Wand folgt. Es kann nun aber sein, daß der Sensor nicht in die Mitte des Raums hineinreicht, wo sich noch ein Gegenstand befindet; dieser bliebe unbeachtet. Zusätzlich zu dem oben genannten Verfahren benötigt man also eine Darstellung der Bereiche, die noch nicht erfaßt worden sind.

Obwohl teilweise Ansätze zur Lösung der drei Problemklassen bestehen, strebt man doch einen Entwurf an, der diese Probleme von vorneherein umgeht. Eine Möglichkeit bietet der **Bubblealgorithmus**, der im folgenden beschrieben werden soll.

3.4 Ein zweiter Ansatz - der Bubblealgorithmus

Das folgende Kapitel führt den zweiten verfolgten Ansatz vor. Dabei wird von der allgemeinen Form ausgegangen, und diese für die konkrete Problemstellung angepaßt. Die endgültige Form wird dann in drei Blöcke unterteilt, und alle Teile einzeln behandelt.

3.4.1 Allgemeine Einführung in das Verfahren

Im ersten Ansatz wurde auf der Basis eines relaxierten Modells eine der Bedingungen aufgegeben, die an eine fertige Karte gestellt wurden. Die Bedingungen waren hierbei

1: Alle Objekte müssen durch eigene Polygone repräsentiert werden.

2: Alle Polygone müssen geschlossen sein.

Im ersten Ansatz wurde die zweite Bedingung aufgegeben, was jedoch keinesfalls zwingend ist. Man könnte sich eine interne Karte vorstellen, in der alle Objekte durch *ein* geschlossenes Polygon dargestellt werden. Hierbei bleibt zu klären, wie man sicherstellt, daß das Polygon immer sinnvoll geschlossen ist und wie man alle Objekte durch nur ein Polygon darstellt. Außerdem muß definiert werden, wie man am Ende der Exploration aus einem Polygon mehrere kleine gewinnt.

Eine Lösung auf diese Fragen gibt ein Algorithmus, der in seiner allgemeinen Form nicht auf Polygonstrukturen beschränkt ist, und prinzipiell für verschiedene Arten der Exploration geeignet ist. Die Grundidee ist hierbei, daß sich der Roboter in einer virtuellen "Blase" befindet, die zunächst durch die Reichweite des Sensors beschränkt ist. Bewegt sich der Roboter, wird die Blase in Fahrtrichtung deformiert, und zwar so, daß der gescannte Bereich immer vollständig innerhalb der Blase liegt. Wird ein Objekt erfaßt, legt sich die Blase an erfaßte Linien an und verfestigt sich. Die Exploration gilt dann als erfolgreich, wenn die komplette Blase sich verfestigt hat.

Grundüberlegungen sind auf den nächsten Bildern gezeigt. Dargestellt ist ein Roboter mit einem im Verhältnis zum Raum gering reichenden Sensor. Steht er mitten im Raum, so bekommt er noch keine Sensorinformationen. Dieser Sachverhalt ist im ersten Bild dargestellt. Die Blase entspricht einem Kreis mit dem Radius der Scannerreichweite. Fährt der Roboter nun in Richtung einer Wand so deformiert sich die Blase, bis die Wand erreicht wird.

Um möglichst viele virtuelle Linien in verfestigte Linien umzuwandeln ergibt sich auch eine einfache Strategie: es wird immer in Richtung einer virtuellen Linie gefahren. In unserem Fall bedeutet dies z.B. nach links. So wird nach und nach der gesamte Raum erkundet. Genauere Betrachtung verdient dabei noch der Ausgang des Raums. Beim Vorbeifahren reicht der Scanner in den Nachbarraum hinein. Gibt man der Blase jedoch eine bestimmte "Viskosität", so bleibt die virtuelle Linie als Raumbegrenzung erhalten. Diese Viskosität kann gesteuert werden mit dem Abstand zwischen zwei real aufgenommenen Punkten, durch die sich die Blase gerade noch hindurchdrücken kann.

In unserem Fall ist die Exploration abgeschlossen, mit dem Ergebnis, daß für den Raum ein Ausgang existiert. Diese Information wird später für die globale Exploration genutzt, um ein topologisches Wegenetz aufzubauen.

Das nächste Beispiel zeigt das Verhalten, wenn sich ein Gegenstand im Raum befindet. Wird die erste Linie des Objekts detektiert, so wird die Linie derart in die interne Karte fusioniert, daß ein geschlossenes Polygon erhalten bleibt. Ermöglicht wird dies durch eine virtuelle Linie von der Wand zum Objekt. Im folgenden wird die Exploration im Raum fortgesetzt, und so die Wände erkundet. Bei dieser Aktion wird auch das Objekt umfahren bis alle vier Seiten aufgenommen sind. Was nun bleibt sind zwei virtuelle Linien, die sicherstellen, daß alle Linien miteinander verbunden sind. Die Exploration ist somit abgeschlossen. In einem nachgeschalteten Schritt muß nun auch die zurückgestellte Bedingung einer Ausgabekarte erfüllt werden. Dies geschieht trivialerweise indem die Blase genau an den Stellen aufgeschnitten wird, an denen sich Paare kolinearer virtueller Linien befinden. Die Schnittreste werden geschlossen und man erhält einzelne Polygone, hier für die Außenwand und für das Objekt je eines.

Im oben vorgestellten Zusammenhang kann die Bubblexploration somit folgendermaßen dargestellt werden:

- **Vollständigkeitskriterium:**

  Es wird solange exploriert, bis die sich die Blase komplett verfestigt hat, oder aus Paaren kolinearer virtueller Verbindungslinien besteht.

- **Modell des Wissens:**

  Das Wissen ist mit den Linien modelliert, die sich verfestigt haben.

- **Modell des Unwissens:**

  Alles was außerhalb der Blase liegt, gilt als noch nicht erfaßt.

- **Strategie:**

  Die Exploration wird so gesteuert, daß immer in Richtung virtueller Linien gefahren wird. Somit werden sukzessive virtuelle Linien in massive Linien umgewandelt.

Es fällt auf, daß die Modellierung der Unwissenheit implizit erfolgt. Man hat durch das Verfahren sichergestellt, daß alles was innerhalb der Bubble liegt, auch mit dem Scanner erfaßt worden ist. Auch die Strategie erweist sich als sehr einfach zu realisieren. Man muß hierzu nur noch definieren, welche der zur Verfügung stehenden virtuellen Linien man aus der Menge auswählt.

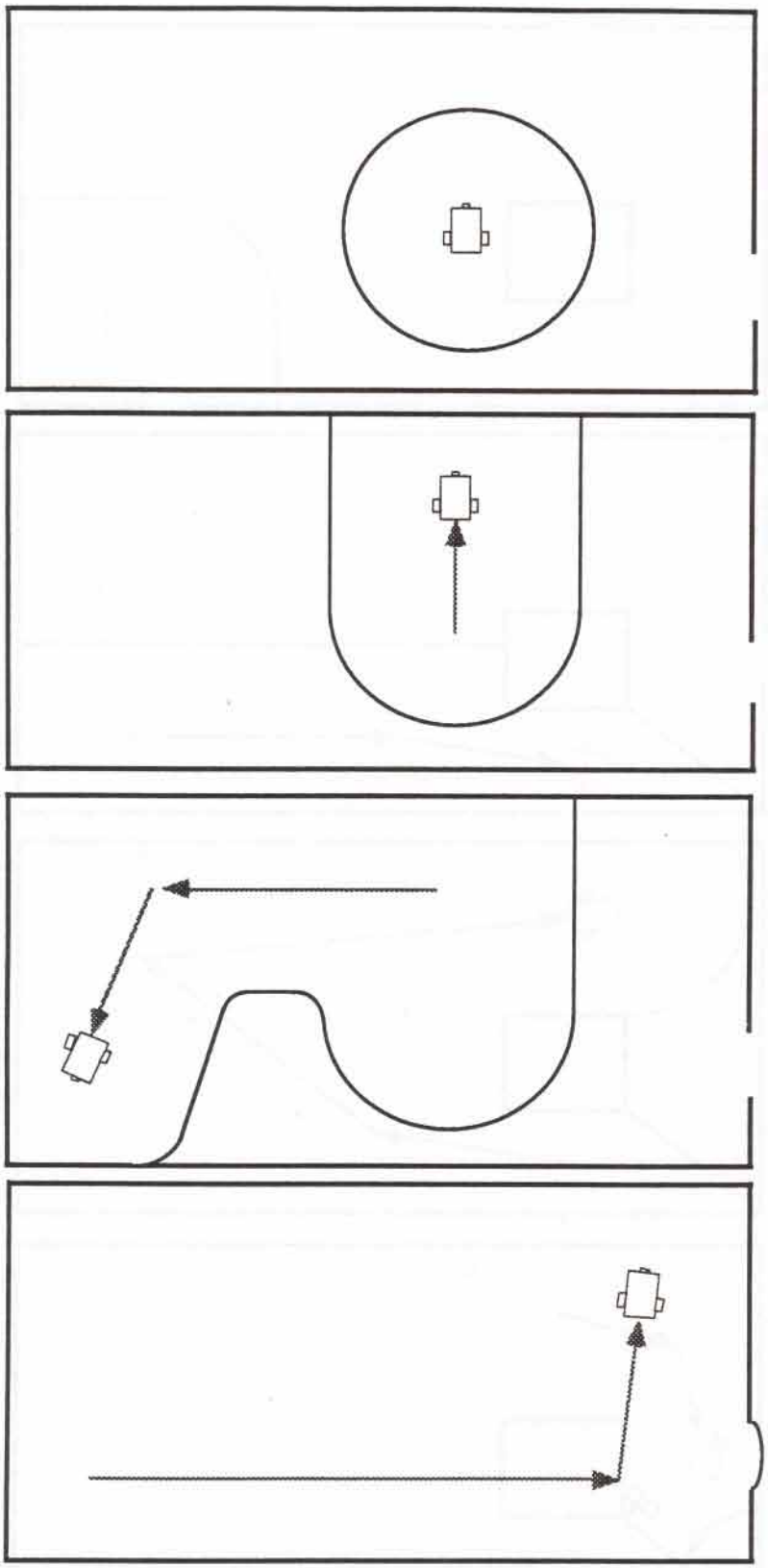


Bild 3.4.1: Grundidee der "Bubbleexploration"

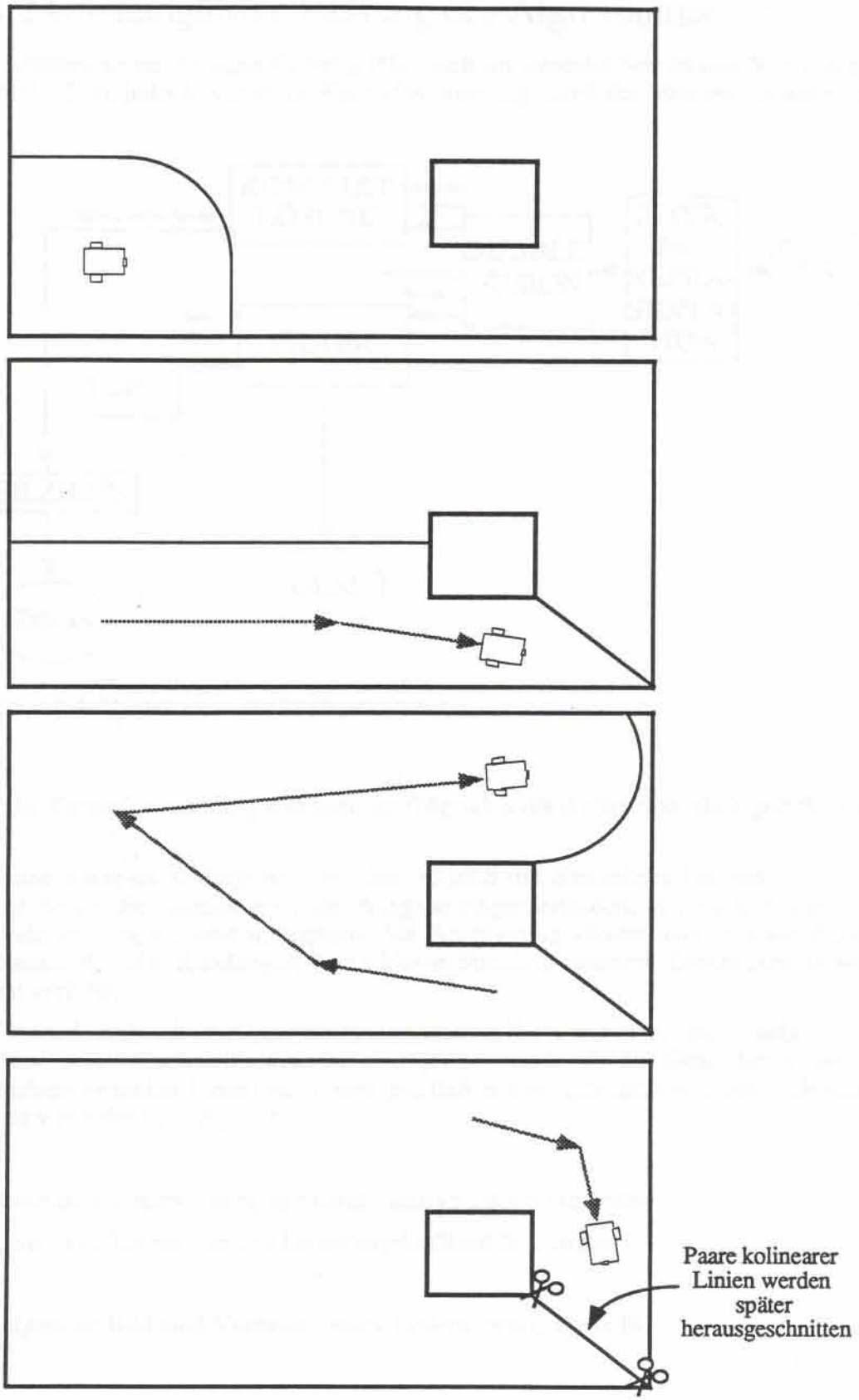


Bild 3.4.2: Exploration mit einem Gegenstand im Raum

3.4.2 Die endgültige Fassung des Algorithmus

Die Architektur der fertigen Fassung lehnt sich im wesentlichen an den Vorschlag aus Kapitel 3.2 an, jedoch wurde die Kartenfusionierung durch den zweiten Ansatz ersetzt.

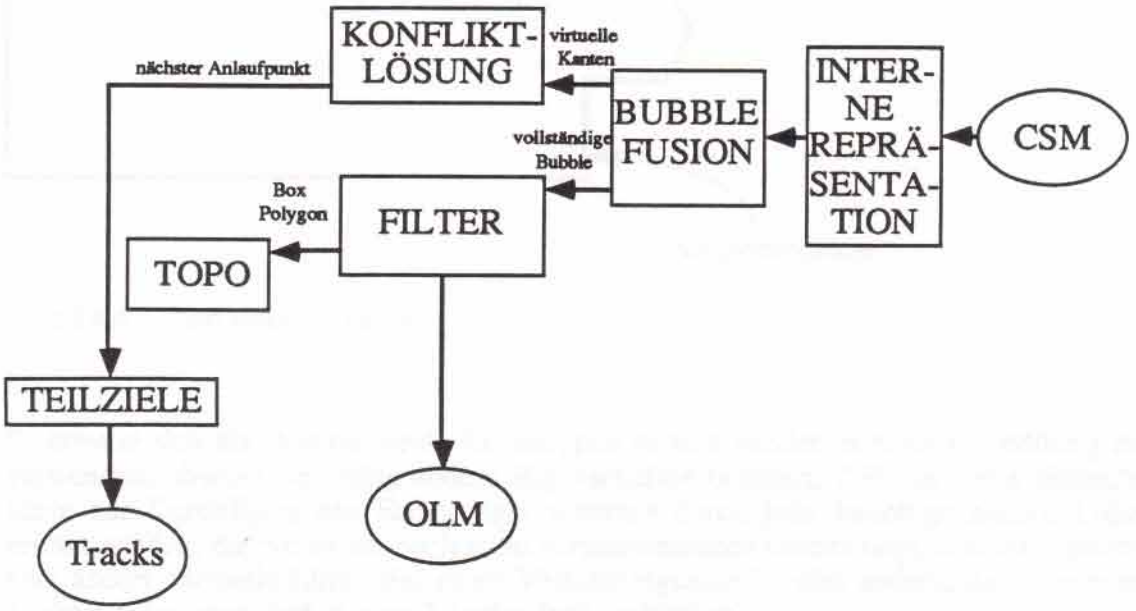


Bild 3.4.3: Architektur der endgültigen Fassung

Auf die Kartenfusionierung soll auch im folgenden die Aufmerksamkeit gerichtet werden:

Soll eine Blase exakt dargestellt werden, so muß mit Kreisbögen hantiert werden. Da jedoch weder die Eingabe noch die Ausgabe Bögen erfordert, erweist sich diese Zwischendarstellung als sehr unbequem. Als Annäherung könnte man sich ein Polygon vorstellen, das alle Rundungen durch kleine Strecken annähert. Dieser Ansatz wurde zuerst verfolgt.

Der Vorteil, daß sich somit die komplette interne Karte mit *einer* einheitlichen Datenstruktur beschreiben läßt, erweist sich jedoch auch als Problem. Beim genauen Betrachten virtueller Linien stellt man fest, daß es zwei qualitativ sehr unterschiedliche Sorten virtueller Linien gibt:

- virtuelle Linien, die benachbarte massive Linien verbinden
- virtuelle Linien, die den Erfahrungshorizont begrenzen

Im folgenden Bild sind Vertreter beider Linientypen dargestellt.

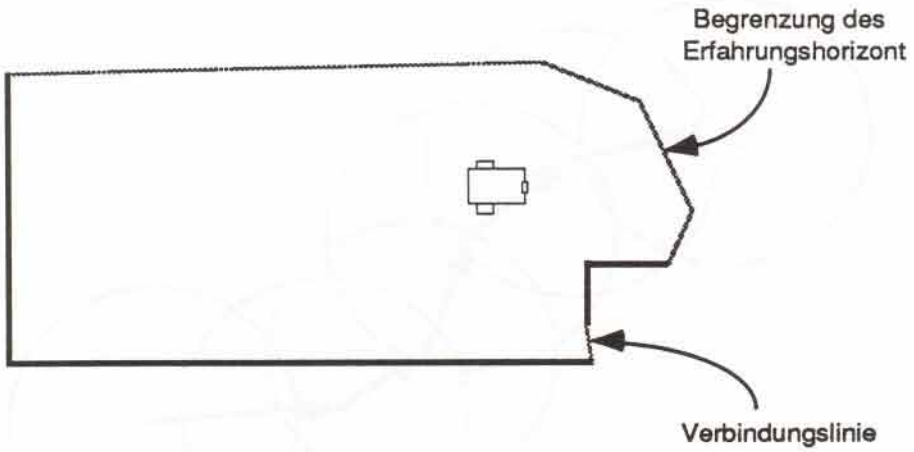


Bild 3.4.4: Typen virtueller Linien

Es erweist sich als störend, beide Linientypen nebeneinander in einer Darstellung zu verwenden, obwohl sie völlig andere Eigenschaften besitzen. Z.B. darf eine virtuelle Linie zur Darstellung des Erfahrungshorizontes durch jede beliebige andere Linie ersetzt werden, die weiter im noch nicht aufgenommenen Gebiet liegt, also auch durch eine andere virtuelle Linie. Bei einer Verbindungslinie ist dies anders, da diese eine Viskosität besitzen darf, die ein "Ausbeulen" verhindert.

Die Lösung liefert eine Datenstruktur, die aus zwei Unterstrukturen besteht. In der einen befinden sich massive Linien sowie virtuelle Verbindungslinien (*Sensorbubble*), in der anderen wird vermerkt, wo sich der Erfahrungshorizont befindet (*Horizonbubble*). Während man die Sensorbubble exakt darstellen kann, muß man bei der Horizonbubble auf eine Näherung zurückgreifen, will man die unbequemen Kreissegmente vermeiden. Es wurde sich hier für eine einfache aber wirkungvolle Technik entschieden.

Theoretisch könnte man *alle* Scannerpositionen während der Fahrt speichern. Für einen Punkt könnte man so ermitteln, ob er in der erfaßten Fläche liegt oder nicht. Man müßte nur feststellen, ob er zu einem der gespeicherten Punkte eine Entfernung hat, die unter dem Scannerradius liegt. Da die Aufnahmen zeitdiskret erfolgen, gibt es nur endlich viele Aufnahmepositionen.

Um jedoch das Datenaufkommen zu reduzieren, speichert man nur einen Bruchteil der Positionen. Wieviele Aufnahmepositionen man speichert, legt die Genauigkeit fest, mit der der erfaßte Bereich wiedergegeben wird. Speichert man alle, so ist der Bereich zwar exakt dargestellt, man handelt sich aber einen erhöhten Suchaufwand ein. Experimente haben gezeigt, daß eine Entfernung von 30% des Radius zwischen zwei Positionen einen guten Kompromiß zwischen Genauigkeit und Datenaufkommen darstellt.

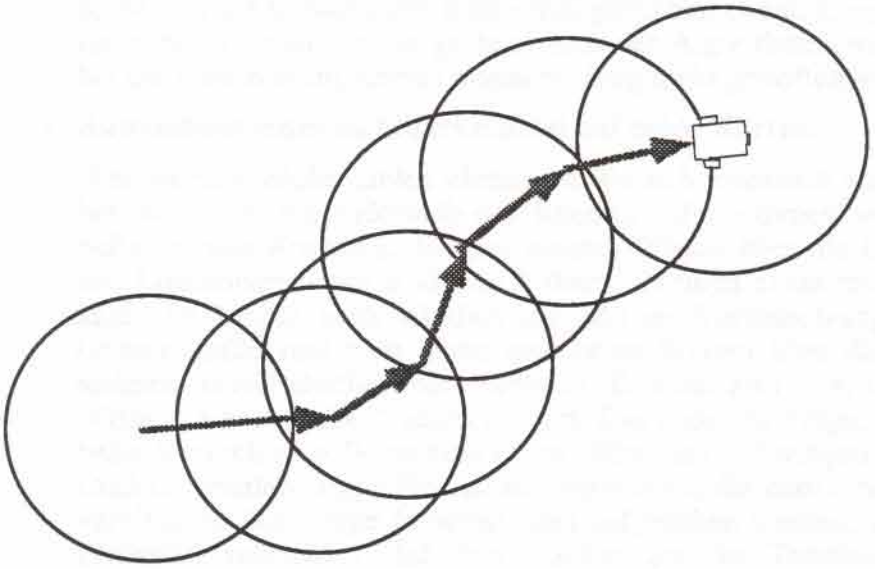


Bild 3.4.5: Darstellung des Erfahrungshorizontes

Wie wir später noch sehen werden, braucht man um ein flächendeckendes Abfahren zu garantieren, nicht die Liste der Positionen jedesmal komplett zu durchsuchen, um noch frei Punkte zu finden. Vielmehr gibt es eine kleine Menge von "verdächtigen" Punkten die zur Untersuchung anstehen. Somit ist der Algorithmus, der auf dieser Struktur aufbaut effektiv. Genaueres hierzu wird im Zusammenhang mit der Befahrungsstrategie diskutiert.

Die Sensorbubble besteht im wesentlichen aus einer Liste von Kanten. Diese Liste ist als Ring angeordnet, so daß in der Bearbeitungsreihenfolge nach der letzten Kante wieder die erste kommt. Um beim Einfügen oder Modifizieren massiver Kanten eine Änderung virtueller Kanten zu vermeiden, wurden diese nur implizit abgespeichert. Eine virtuelle Kante liegt dann vor, wenn benachbarte massive Linien keinen gemeinsamen Endpunkt besitzen. Um eine Linienfusion zu beschleunigen, wird zu jeder Linie das umgebende Rechteck (*bounding rect*) sowie der Winkel der Verbindungslinie vom aktuellen Standpunkt zu den Linienendpunkten abgespeichert. Außerdem muß verzeichnet sein, ob die Linie als Cluster aufgenommen wurde. Cluster sind so kleine Punktgruppen, daß man ihnen keine Richtung zuordnen kann. Darauf muß bei der Fusionierung geachtet werden.

Beide Strukturen werden als doppelt verkettete Listen realisiert, um einen schnellen Zugriff auf die einzelnen Komponenten zu gewährleisten. Außerdem kann das Datenaufkommen schlecht abgeschätzt werden, so daß sich eine Darstellung als Feld nicht eignet.

Als nächstes soll der eigentliche Fusionierungsalgorithmus dargestellt werden. Gegeben ist hierbei die aktuelle Gesamtkarte, die alle bisher gesammelten Informationen enthält, sowie eine neue Karte die der aktuellen Radaraufnahme entspricht. Die Fusionierung erfolgt grob in drei Schritten:

- **Konvertierung der neuen Karte in das Bubbleformat:**

  Diese Konvertierung hat einen einfachen Grund: prinzipiell sind die neue

Karte und die Gesamtkarte ersteinmal gleichberechtigt. Liegen beide Karten im selben Format vor, so gestaltet sich der Algorithmus sehr homogen, da bei der Fusionierung keine Unterscheidung mehr getroffen werden muß.

- **Anwendung eines Sichtbarkeitstest auf beide Karten:**

  Hier werden solche Linien eliminiert, die sich innerhalb von Gegenständen befinden oder die außerhalb des Raumes aufgenommen wurden. Nur hier befinden sich Routinen, die gewonnenes Wissen über die Umwelt vernichten. Löschoperationen sind sehr kritisch, da nicht exakt festgestellt werden kann, ob Linien auch wirklich die nötigen Voraussetzungen mitbringen. Gründe dafür sind zum einen unsicheres Wissen über die Umwelt, zum anderen unvollständige Informationen. Es kann also sein, daß brauchbares Wissen "aus Versehen" gelöscht wird. Das hätte zur Folge, daß das System beim Versuch eine Wissenslücke zu füllen, ins Schwingen geraten könnte. Deshalb wurden einige Heuristiken entwickelt, die zum einen relativ sicher verhindern, daß nötige Informationen aufgegeben werden, zum anderen im Fehlerfall vermeiden, daß durch Schwingen die Terminierung in Frage gestellt wird.

- **Zusammenfassung der resultierenden Karten zu der neuen Gesamtkarte:**

  Durch den Sichtbarkeitstest ist sichergestellt, daß sich die Karten relativ problemlos zu einer neuen zusammenfassen lassen, ohne daß sich Angaben widersprechen. Jede einzelne Kante der neuen Karte wird nun in die Gesamtkarte an eine passende Stelle einsortiert.

Die daraus resultierende Karte enthält nun bis auf die gelöschten Kanten alle Informationen, die sich aus den bis dahin gewonnenen Aufnahmen gewinnen lassen. Die einzelnen Schritte sollen nun genauer analysiert werden.

3.4.2.a Die Formatumwandlung

Geliefert wird von außen eine Menge von Linien. Im Fall eines rotierenden Sensors, werden spätestens nach der ersten Merkmalsextraktion alle Linien in einem bestimmten Umlaufsinn abgelegt. Da aber auch andere Sensorkonfigurationen berücksichtigt werden sollen, werden alle einkommenden Linien nach aufsteigendem Winkel sortiert. Zwischen Linien, die nicht zusammenstoßen liegen dann nachher virtuelle Kanten, so wie im nächsten Bild gezeigt.

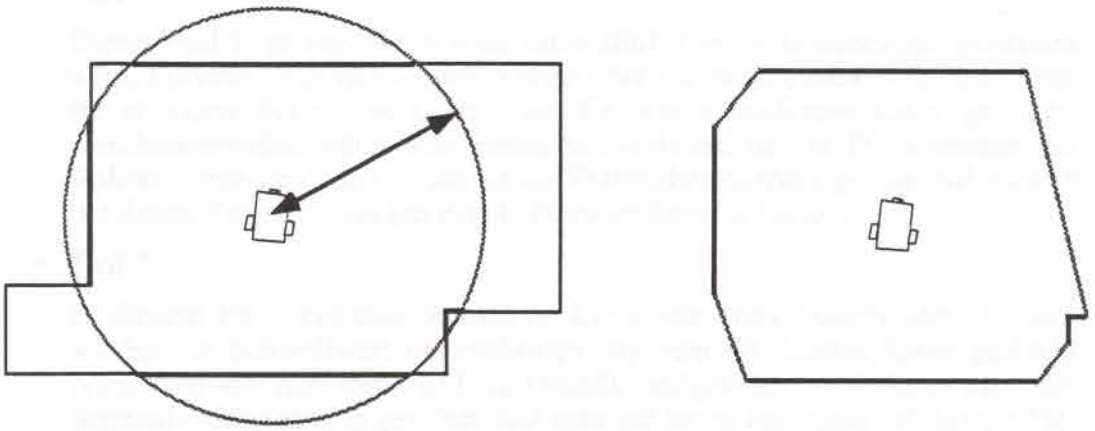


Bild 3.4.6: Umwandlung der gelieferten Karte in eine Sensorbubble

Da man den Aufnahmeort zu jedem Scan kennt, läßt sich auch ermitteln wo bei jeder Linie der Frei- und wo der Hindernisraum liegt. Da die fertige Karte geschlossene Polygonzüge enthalten soll, deren Umlaufsinn auch genau diese Trennung erlaubt, ist es an dieser Stelle sinnvoll, schon alle Linien in dieser bestimmten Orientierung abzulegen. Jede Linie wird bespielsweise so repräsentiert, daß wenn man sich am Startpunkt befindet und in Richtung Endpunkt blickt, der Hindernisraum auf der rechten Seite liegt.

Nach der Sortierung wird auch die zusätzlich benötigte Information hinzugeschrieben wie bounding rect und Winkelinformationen.

3.4.2.b Der Sichtbarkeitstest

Wie erwähnt werden hier zwei Ziele verfolgt:

- Eliminierung der Linien, die innerhalb eines Objekts liegen.
- Eliminierung der Linien, die außerhalb des Raums liegen.

Das Problem, was sich hierbei stellt, ist daß man die innerhalb-außerhalb-Relation schon nachtesten muß *bevor* ein geschlossenes Polygon vorliegt. Zu entscheiden, ob ein Punkt innerhalb eines geschlossenen Polygons liegt wäre recht einfach. In der Literatur wurden diverse Algorithmen erläutert, z.B. in [Meier86]. Für den Fall eines offenen Polygons ist man an geeigneten Heuristiken interessiert, die es ermöglichen, schon vorab relativ genau zu entscheiden, welcher Fall vorliegt. Dazu kann man zum einen das Wissen über die Umwelt ausnutzen zum anderen Wissen über die Größe des Roboters. Zum Beispiel weiß man, daß in der Regel Räume keine Wände besitzen, die stark in das Rauminnere hineinreichen. Weiter weiß man, daß eine Linie, die sehr nah hinter einer aufgenommenen Linie erfaßt wird, wahrscheinlich ignoriert werden kann, da sie innerhalb eines Objektes liegen wird. Folgende Typen von Linienlöschung existieren:

- **Fall 1:**

  Dieser Fall liegt vor, wenn eine Linie **hinter** einer bestehenden gefunden wird. Entscheidend ist nun der Abstand der Linien zueinander. Unterschreitet er einen Schwellwert, so wird die weiter entfernte Linie gelöscht. Geschickterweise wählt man einen Schwellwert, der in Dimensionen der Roboterbreite liegt, denn dann ist die Wahrscheinlichkeit gering, daß es sich bei dieser Konstellation um eine befahrbare Strecke handelt.

- **Fall 2:**

  In diesem Fall liegt eine detektierte Linie **vor** einer bestehenden. Ist hier wieder ein Schwellwert unterschritten, so wird die hintere Linie gelöscht oder, wie im nächsten Bild dargestellt, aufgeteilt. Auch hier wird der Schwellwert wieder so gewählt, daß man auf jeden Fall nicht befahrbare Flächen abschneidet.

- **Fall 3:**

  Dieser Fall tritt ein, wenn sich eine Linie **hinter** einer **virtuellen** Linie befindet. Neben der Entfernung ist vor allem die Länge der virtuellen Linie relevant. Unterschreitet sie eine bestimmte Größe, so kann man die dahinterliegende Linie als außerhalb des Raums annehmen. Der Schwellwert hier wird jedoch zweckmäßigerweise nicht auf die minimale Breite zum Durchfahren gesetzt. Da diese virtuelle Linie auch von einer Tür zu einem anderen Raum stammen könnte, sollte der Schwellwert auf die maximale Türbreite gesetzt werden.

Im folgende Bild sind noch einmal alle Fälle dargestellt.

Die Untersuchung, ob Linien gelöscht oder aufgeteilt werden sollen, wird nun gegenseitig zuerst auf die neue Karte, dann auf die schon bestehende durchgeführt. Das Resultat kann der letzten Stufe zugeführt werden.

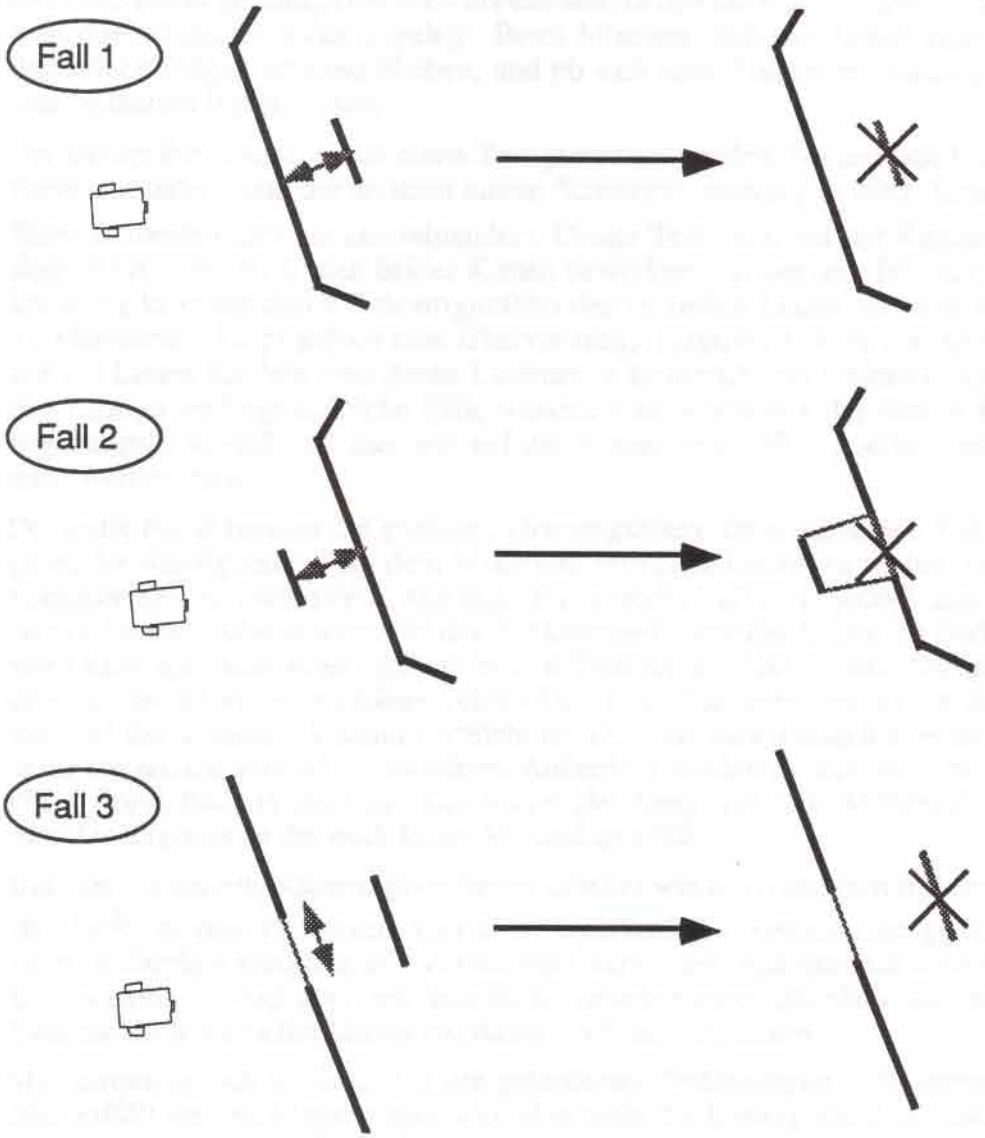


Bild 3.4.7: Fälle für die Löschung erfaßter Linien

3.4.2.c Die Kartenzusammenfassung

Die Aufgabe, die sich nun stellt ist keineswegs so trivial, wie es zuerst scheint. Linien aus zwei Quellen müssen so vermischt werden, daß sich die resultierende Karte in einem konsistenten Zustand befindet. Konsistent heißt hier:

- Nachbarschaftsbeziehungen sollen erhalten bleiben.
- Es sollen keine Überkreuzungen zwischen Linien existieren.
- Die Summe der Längen virtueller Linien soll minimal sein.

Der erste Punkt gestaltet sich noch als einfach. In den einzelnen Karten sind Nachbarlinien direkt hintereinander abgelegt. Beim Mischen muß nur darauf geachtet werden, daß solche Folgen erhalten bleiben, und ob sich neue Folgen mit Beteiligten aus beiden Teilkarten bilden lassen.

Der zweite Punkt kann durch einen Test garantiert werden, bei der jede Linie aus einer Karte mit jeder Linie der anderen einem Schnittest unterzogen wird (Linien aus einer Karte schneiden sich nie untereinander). Dieser Test kann mit der Komplexität $O(n^2)$ über der Anzahl der Linien beider Karten bewerkstelligt werden. Im Falle einer Überkreuzung kann nur durch Rekonfiguration der virtuellen Linien versucht werden diese zu eliminieren. Liegt jedoch eine Überkreuzung massiver Linien vor, so kann es sich nur um Linien handeln, von denen Linienreste innerhalb von Objekten oder außerhalb des Raumes vorliegen. Solche Fälle wurden aber schon von der letzten Komponente abgehandelt, so daß sich hier nur auf die Kreuzungen mit virtuellen Linien konzentriert werden muß.

Der dritte Punkt bereitet die größten Schwierigkeiten. Im allgemeinen Fall ist die Aufgabe, die Konfiguration mit dem Minimum virtueller Linien zu finden, nicht in polynomialer Zeit zu lösen (siehe Anhang). Für unseren Fall ist es jedoch ausreichend, ein relativ "gutes" Subminimum für das Aufkommen virtueller Linien zu finden. Desweiteren kann man ausnutzen, daß die beiden Teilkarten schon in einer Sortierung vorliegen, bei der die virtuellen Linien relativ kurz sind. Die neue Sensorkarte liegt sogar so vor, daß das absolute Minimum erreicht ist. Dies hat man dadurch erreicht, alle Linien in aufsteigenden Winkeln anzuordnen. Außerdem handelt es sich bei dem Problem um einen Spezialfall, in dem die Dreiecksungleichung gilt (als Abstandsfunktion wird zweckmäßigerweise der euklidsche Abstand gewählt).

Das hier vorgestellt Näherungsverfahren arbeitet wie der Schnittest mit der Komplexität $O(n^2)$, so daß für diesen Schritt die quadratische Komplexität gehalten werden kann. Außerdem wird hier von vorneherein verhindert, daß man schneidende virtuelle Linien erhält, so daß man den Schnittest entweder ganz fallenläßt oder nur noch auf Linienschnitte virtueller Linien mit massiven Linien reduziert.

Man nimmt jedoch in Kauf, daß die geforderten Bedingungen nicht immer im vollen Maß erfüllt werden können; man wird also nicht die Lösung mit dem absoluten Minimum erhalten. Das Verfahren basiert darauf, jede Linie einer Karte in die andere Karte "optimal" einzufügen. Ist die Wahl erst einmal getroffen, so wird diese nicht mehr rückgängig gemacht, auch wenn sich bei folgenden Kanten die Bedingungen ändern. Es handelt sich also hierbei also um einen *Greedy-Algorithmus*. Die Optimalität beim Einsortieren richtet sich danach, welches Aufkommen virtueller Linien entstehen würde. Für jede Kante werden alle Einfügepositionen durchgetestet, und beim Minimum dann tatsächlich eingefügt. Hierbei muß man nicht, wie man zuerst annehmen könnte, beide Möglichkeiten austesten, einen Linienendpunkt zu einer Seite der Lücke zuzuordnen. Da die Linien eine eindeutige Orientierung besitzen, die angibt auf welcher Seite der Freiraum liegt, kann die Einfügung nur in einer Weise stattfinden. Betrachtet man nämlich einen geschlossenen Linienzug in der Richtung in der die Linien vorliegen, darf der Freiraum immer nur auf einer bestimmten Seite liegen. Hiermit halbiert sich der Aufwand beim Einfügen.

Der Algorithmus, um eine Kante einzufügen, hat somit folgendes Aussehen:

Eingabe: Linie (P_1,P_2), Karte: Menge von Linien (Q_{i1},Q_{i2}) mit $1 \le i \le n$

Ausgabe: Index i_{Min}, an der die Kante optimal eingefügt wird

Algorithmus "Ermittle Einfügeposition":

Setze $val_{Min}:=\infty$

Für alle i mit $1 \le i \le n$:

Setze $j:=(i \text{ Mod } n)+1$

Berechne $\Delta:=|(Q_{i2},P_1)|+|(P_2,Q_{j1})|-|(Q_{i2},Q_{j1})|$

Wenn $\Delta<val_{Min}$ dann

Setze $val_{Min}:=\Delta_1$, $i_{Min}:=i$

Algorithmus 3.4.8: Einsortieren einer Kante in eine Karte

Der Algorithmus, um zwei Karten zu fusionieren, sieht dann so aus:

Eingabe: Menge von Linien (P_{i1},P_{i2}) mit $1 \le i \le m$, und Menge von Linien (Q_{i1},Q_{i2}) mit $1 \le i \le n$

Ausgabe: Menge von Linien (Q_{i1},Q_{i2}) mit $1 \le i \le n+m$

Algorithmus "Fusioniere Karten":

Für alle i mit $1 \le i \le m$:

Ermittle Einfügeposition von (P_{i1},P_{i2}) in (Q_{j1},Q_{j2}) mit $1 \le j \le n+i-1 \rightarrow i_{Min}$

Füge (P_{i1},P_{i2}) an Stelle i_{Min} in (Q_{j1},Q_{j2}) ein

Algorithmus 3.4.9: Fusionierung zweier Karten

Das folgende Bild veranschaulicht das Vorgehen bei einer einzigen Kante, die in einem zusammenhängenden Bereich von vier Kanten optimal eingefügt werden soll:

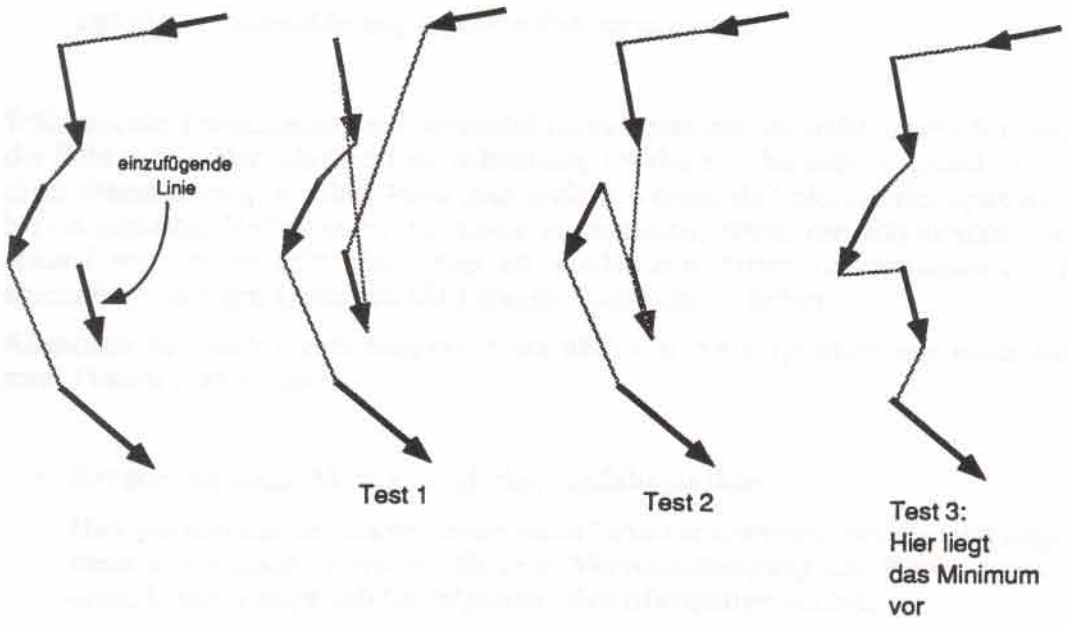


Bild 3.4.10: Suche nach der optimalen Einfügeposition

Hiermit ist die Kartenfusionierung beschrieben. Man hat nun eine Technik zur Verfügung, die es ermöglicht, aus einer Menge von Eingabedaten ein aussagekräftige Karte aufzubauen. Diese enthält alle notwendigen Informationen, um darauf eine Planung durchzuführen. Im folgenden soll beschrieben werden, wie anhand der internen Karte ein Wegplanung durchgeführt wird, die es ermöglicht, die Karte sukzessive zu vervollständigen. Der letzte Punkt der lokalen Exploration ist es dann, aus der vollständigen Karte eine für die Ausgabe zugeschnittene Karte zu konstruieren.

3.5 Weiterführende Komponenten

Nach der Beschreibung der Kartographierung wird nun darauf eingegangen, wie Fahrbefehle auf der Basis der aktuellen Karte generiert werden. Der Generierung findet dabei in zwei Stufen statt, die beide ausführlich beschrieben werden. Die nach einer erfolgreichen Fahrt aufgebaute Karte muß noch einer weiteren Komponente zugeführt werden, bevor sie einer weiteren Applikation zur Verfügung gestellt wird. Dieser Komponente ist ein weiterer Punkt gewidmet.

3.5.1 Die Strategie zur Wegplanung

Auf der Basis der noch unvollständigen Karte werden nun Fahrbefehle erzeugt. Werden diese ausgeführt und der Roboter bewegt sich, so wird die Karte mit neuen Informationen angereichert, da die Umwelt von einer neuen Position wahrgenommen wird.

Die Fahrbefehle sollen dabei so erzeugt werden, daß

- die Informationen möglichst schnell aufgesammelt werden,
- und die Kartographierung in jedem Fall terminiert.

Während die Terminierung ein unverrückbares Kriterium darstellt, ist die Minimierung der Fahrzeit wieder relativiert zu betrachten. Da die Fahrbefehle aufgrund unvollständiger Daten erzeugt werden, kann man nicht erwarten, daß hieraus ein optimales Verhalten entsteht. Vielmehr ist es schon ausreichend, wenn die Exploration "relativ" schnell von statten geht. Auch hier ist wieder eine Strategie vorzuziehen, die statt einem fragwürdigen Optimum ein sicheres Suboptimum liefert.

Allgemein läßt sich die Erzeugung eines nächsten Anfahrpunktes aus einer Karte in zwei Phasen unterteilen:

- **Erzeugung einer Menge möglicher Anfahrpunkte:**

  Hier werden aus der Karte "interessante" Punkte ausgerechnet. Es sind allgemein alle Punkte vertreten, die eine Vervollständigung der Karte versprechen. Diese Menge soll im folgenden **Konfliktmenge** heißen.

- **Konfliktlösung:**

  Aus der Menge der Punkte wird nun ein nächster Punkt ermittelt. Kriterien für die Auswahl sind die zu erwartende Informationsgewinne, sowie der Aufwand beim Anfahren. Nach einem Bewertungsschema werden diese Kriterien gewichtet und derjenige Punkt genommen, der das maximale Gewicht besitzt.

Das Vorgehen wird noch einmal im nächsten Bild erläutert.

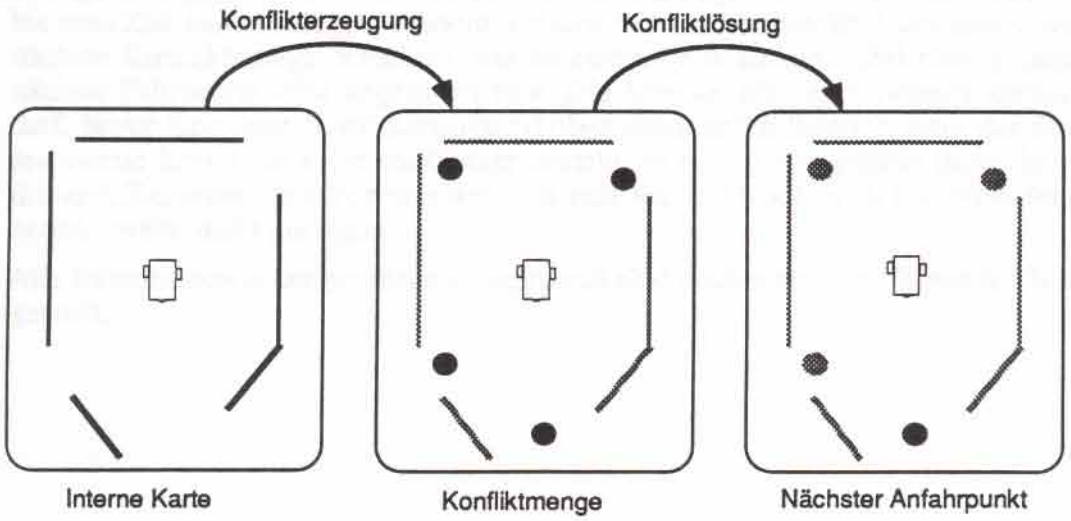


Bild 3.5.1: Erzeugung des nächsten Anfahrpunktes aus einer Karte

Hierbei handelt es sich nun nicht um eine festgelegte Strategie. Vielmehr kann man durch die Methode, die Konfliktmenge zu erzeugen und zu lösen, die unterschiedlichsten Strategien gewinnen. Es handelt sich also hierbei um ein allgemeines Prinzip zum Modellieren einer Befahrungsstrategie für die Exploration.

Die Konfliktmenge ändert sich nun *während* der Fahrt dynamisch mit der Karte. Es verschwinden Konfliktpunkte, sollten sich neue Erkenntnisse ergeben, oder sie wandern in Richtung des neuen Erfahrungshorizontes. Die Frage ist nun, wie oft die Konfliktmenge erhoben werden soll. Die Extreme wären hierbei:

- **Extrem 1:**

  Die Konfliktmenge wird nach jeder Veränderung der internen Karte neu berechnet und eventuell abgegebene Fahrbefehle gegebenenfalls modifiziert.

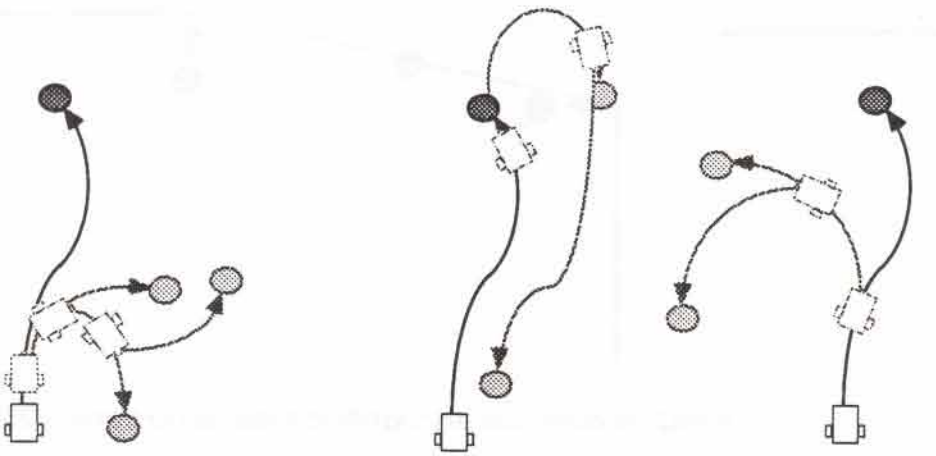
- **Extrem 2:**

  Die Konfliktmenge wird erst wieder neu berechnet, wenn ein Fahrbefehl komplett abgearbeitet wurde, also wenn ein Konfliktpunkt erreicht ist.

Extrem1 hat den Nachteil, daß sich laufend die Fahrbefehle ändern. Der Navigator müßte also nach jeder Karte eine neue Route berechnen. Außerdem ist die Erhebung der Konfliktmenge ein Vorgang, den man aus Zeitgründen nicht schritthaltend mit der Kartographierung machen möchte. Andererseits reagiert das System maximal flexibel auf neue Informationen. Extrem2 birgt die Gefahr, daß neue Informationen erst sehr spät zu neuem Verhalten führen. So kann es sein, daß schon direkt nach Antreten der Fahrt, der Anlaß zu derselben nicht mehr besteht. Hier wäre es dann sinnvoll zu einem anderen Ziel umzuschwenken. Ein anderes Problem entsteht, wenn ein Konfliktpunkt hinter Hindernissen liegt, die noch nicht erfaßt worden sind. Der Navigator berechnet eine Route die diese noch nicht mitberücksichtigt, und erst eine niedrige Kontrollstufe (Pilot) würde einen Zusammenstoß verhindern.

Aus diesen Gründen wurde eine Erhebungshäufigkeit gewählt, die zwischen beiden Extremen liegt. Es wird eine Konfliktmenge berechnet und ein Fahrbefehl an den Navigator abgegeben. Dieser wird aber zusätzlich angewiesen, nicht die volle Distanz bis zum Ziel zurückzulegen, sondern nur eine bestimmte Strecke. Nach dieser wird die nächste Konfliktmenge berechnet, die zu einem ganz anderen Ziel führen kann. Der nächste Fahrbefehl wird abgegeben usw. Die Strecke, die der Navigator zurücklegen darf, bevor eine neue Konfliktmenge erhoben wird, sollte deutlich unter der Scannerreichweite liegen, da sonst die Gefahr besteht, in noch nicht erfaßte Bereiche vorzudringen. Experimente haben ergeben, daß sich Werte zwischen 30 und 50% der Scannerreichweite recht gut eignen.

Alle Extreme sowie der gewählte Kompromiß sind noch einmal im folgenden Bild dargestellt.



Extrem1:
Jede neue Karte führt potentiell zu einem neuen Ziel

Extrem2:
Erst im Ziel wird ein neuer Anlaufpunkt berechnet

Synthese:
Nach einer bestimmten Distanz wird neu berechnet

Bild 3.5.2: Möglichkeiten der Erhebungshäufigkeit einer Konfliktmenge

3.5.1.a Erzeugung der Konfliktmenge

Der hier vorgeführte Vorschlag der Konfliktmengenerzeugung basiert im wesentlichen darauf, dort Konfliktpunkte hinzulegen, wo sich virtuelle Linien befinden.

Da virtuelle Linien grundsätzlich von massiven Linien umgeben sind, besteht eine große Wahrscheinlichkeit, daß sich beim Fahren in diese Richtung die massiven Linien vervollständigen. Es bleibt nur noch zu klären, wie man einer virtuellen Linie Konfliktpunkte zuordnet. Ein erster Ansatz wäre genau den Mittelpunkt einer virtuellen Linie zu nehmen. Handelt es sich jedoch um eine sehr große Linie (z.B. in der Größenordnung der Scannerreichweite), so führt die Halbierung dazu, daß man sich von den angrenzenden massiven Linien entfernt. Somit würde sich unter Umständen das Wissen über massive Kanten nicht vermehren. Der Mittelpunkt einer virtuellen Linie ist jedoch nicht gänzlich uninteressant, da sich zumindest der Erfahrungshorizont erweitert.

Eine bessere Wahl der Konfliktpunkte ist, sie an die beiden Enden einer virtuellen Linie zu legen. Da sich jedoch diese Position nicht exakt vom Navigator anfahren läßt, da sie sich in der Nähe einer massive Linie befindet, wird der Punkt etwas in den Freiraum projiziert. Man erhält somit zu jeder virtuellen Linie drei Konfliktpunkte. Bei kurzen Linien fallen diese Punkte immer näher zusammen, so daß man unter einer bestimmten Länge nur noch einen Punkt erhält. Dort kann es dann auch passieren, daß der Mittelpunkt einer virtuellen Linie dann nicht mehr vom Navigator erreicht werden kann. Aus diesem Grund wird bei kleinen virtuellen Linien der Mittelpunkt in den Freiraum projiziert.

Im Falle von drei unterschiedlichen Punkten wird man den äußeren ein höheres Gewicht mitgeben, als dem mittleren.

Bild 3.5.3: Konstruktion der Konfliktpunkte aus virtuellen Linien

Der Vorteil dieser Methode (in Verbindung mit einer geeigneten Konfliktlösung) besteht darin, daß eine Wandverfolgungsstrategie entsteht, ohne daß man diese explizit programmieren muß. Durch die Konfliktpunkte an der Wand wird der Roboter immer näher an eine Wand fahren. Von dann an wird er solange auf den Konfliktpunkt zusteuern, der immer unmittelbar neben der Wand entsteht, bis man auf schon aufgenommene Wände stößt. Im einfachsten Fall bedeutet dies, daß man einmal an der Außenwand entlang gefahren ist. Später wird dieser Vorgang genauer in einem Testlauf gezeigt werden.

Eine weitere Möglichkeit Konfliktpunkte zu setzen liegt dann vor, wenn die Karten, die man erhält, vollständig leer sind. Das kann passieren, wenn sich der Roboter inmitten eines sehr großen Raumes befindet, so daß der Scanner keine Wände aufnimmt. In diesem Fall konstruiert man einen einzigen Konfliktpunkt in einer bestimmten Richtung (z.B. in der Richtung in der der Roboter gerade steht). Dies führt zu dem erwünschten Verhalten, daß der Roboter einfach solange in eine Richtung fährt, bis er zum erstenmal eine Linie wahrnimmt.

Eine letzte Art von Konfliktpunkten entsteht aus der Horizonbubble. Wie oben erwähnt kann man nicht erwarten, daß wenn alle Polygone geschlossen sind, auch automatisch der Raum vollständig erfaßt wurde, da es noch Punkte geben kann, die nicht vom Sensor aufgenommen worden sind. Aufgrund der Horizonbubble kann man nun genau solche Punkte zu der Menge der Konfliktpunkte hinzunehmen. Da es sich jedoch bei der Menge, der noch nicht aufgenommenen Punkte um eine Fläche handelt, muß man sich auf endlich viele Repräsentanten beschränken. Man wählt hier solche Punkte für die gilt, daß wenn diese alle angefahren wurden und keine neuen Punkte hinzugekommen sind, man die gesamte Fläche erfaßt hat. Das folgende Bild beschreibt das Konstruktionsprinzip:

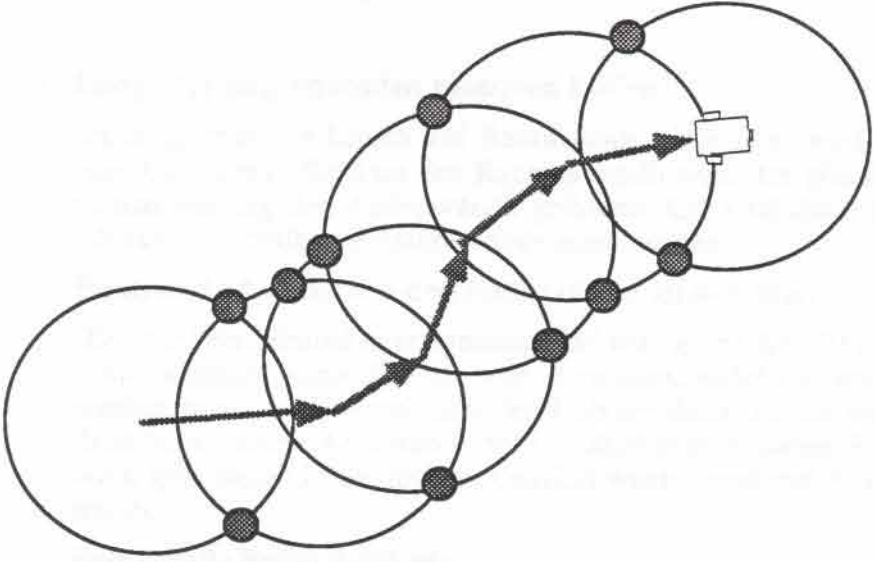


Bild 3.5.4: Konstruktion der Konfliktpunkte aus der Horizonbubble

Man schneidet jeweils zwei Kreise, die man um Punkte zieht, die direkt hintereinander in der Horizonbubble abgelegt sind. Die zwei Schnittpunkte liefern jeweils neue potentielle Konfliktpunkte. Es müssen nur solche Punkte aussortiert werden, die schon innerhalb anderer Kreise liegen. Dadurch daß man den Radius der Kreiskonstruktionen variiert, kann man verschiedene Grade von Überlappung erreichen.

Man erhält durch die beschriebenen Verfahren eine Menge Konfliktpunkte, die nun der nächsten Stufe zugeführt werden.

3.5.1.b Konfliktlösung

Jedem Punkt der Konfliktmenge wird nun ein Gewicht zugeordnet, das seine "Qualität" für die Exploration wiederspiegelt. Hierbei spielen verschiedene Teilkriterien eine rolle, die jeweils ein individuelles Gewicht besitzen. Das Gesamtgewicht der Kriterien x_i, mit den einzelnen Gewichten g_i wird also durch das Skalarprodukt

$$G=(x_1 \dots x_i \dots x_n) \begin{pmatrix} g_1 \\ \dots \\ g_i \\ \dots \\ g_n \end{pmatrix}$$

berechnet.

Hier wurde sich für die folgenden Teilkriterien entschieden:

- **Länge der angrenzenden massiven Linien:**

  Da lange massive Linien auf Raumwände hindeuten, wird so erreicht, daß zuerst die grobe Struktur des Raumes erfaßt wird. Im günstigsten Fall wird einmal entlang der Außenwände gefahren. Oft sind dann auch schon alle Objekte innerhalb des Raumes abgetastet worden.

- **Positionsänderung, um den Punkt direkt zu befahren:**

  Hier wird erst einmal angenommen, daß man einen Konfliktpunkt in direkter Linie befahren kann. Hierzu wird berechnet, welche Distanz zurückgelegt werden muß und wieviel sich der Roboter dazu drehen muß. Auch wenn dann beim echten Abfahren unter Umständen eine andere Route genommen wird, gibt diese Größe doch tendenziell wieder, welcher Aufwand zu erwarten ist.

- **Prinzipielle Befahrbarkeit:**

  Durch die Konstruktion von Konfliktpunkten zum flächendeckenden Abfahren können Anfahrpunkte entstehen, die außerhalb der Raumes liegen. Auch solche von virtuellen Linien können unter Umständen nicht erreicht werden, da sie nur durch zu schmale Gänge mit dem Freiraum verbunden sind. Zwar würde spätestens durch eine Fehlermeldung des Navigators dieser Umstand klar werden, aber es wäre besser, offensichtliche Fälle direkt auszuschließen. Da der Navigator wesentlich mehr Wege untersuchen muß um eine Befahrbarkeit auszuschließen, als im Durchschnitt nötig, um eine Route zu planen, sollte dies nur in Einzelfällen passieren. Es wurde sich deswegen dafür entschieden, einen einfachen Navigationsalgorithmus zu entwerfen, der maximal einen Umfahrungspunkt zum Ziel berechnet. Dieser eigene Navigator ist, wie wir später noch sehen werden, auch aus anderen Gründen sinnvoll, so daß dieser Aufwand gerechtfertigt ist.

Ein weiterer Aspekt ist die Gefahr, daß das System ins Schwingen gerät. Wie in den vorigen Kapiteln beschrieben wurde, kann nicht vollständig ausgeschlossen werden, daß gültige Linien aus der Karte entfernt werden. Die zurückbleibenden Lücken produzieren neue Konfliktpunkte d.h. diese Linien liegen wieder zur wiederholten Erkundung an. In seltenen Fällen können sich Paare von Linien herausbilden, die sich bei der Erkundung gegenseitig Löschen. Solche Fälle führen zum Schwingen des Gesamtsystems, wodurch keine Terminierung erfolgt.

Um das Problem zumindest zu entschärfen, wurden solche Konfliktpunkte aus der Konfliktmenge aussortiert, die eine bestimmte Entfernung zu schon befahrenen Punkten unterschreiten. Sinnvollerweise zieht man zu diesem Test wieder die Horizonbubble heran. Es kann zwar dadurch nicht garantiert werden, daß die Karte vollständig geschlossen vorliegt, aber das System beendet die Exploration. Somit besteht zumindest die Chance, daß solche offenen Polygone am Ende sinnvoll geschlossen werden.

Die Konfliktmengenerzeugung und Konfliktlösung steuern nun solange die Exploration, bis keine weiteren Punkte mehr zur Befahrung vorliegen. Die dann vorliegende Karte wird nun einer abschließenden Komponente zugeführt.

3.5.2 Die abschließende Kartenerzeugung

Der nun folgende Schritt verwandelt die vervollständigte interne Karte in eine Polygonkarte (Outline Map). Dazu muß die Blase so aufgetrennt werden, daß nachher eine Menge geschlossener Polygonzüge vorliegt. Außerdem wird hier noch teilweise versucht, die Unzulänglichkeiten während der Exploration (siehe Kapitel 3.4.2.b und 3.4.2.c) auszugleichen. Dazu werden auf die extrahierten Polygone diverse Filter angewendet. Die einzelnen Stufen werden im folgenden beschrieben:

- **Stufe 1: Aufteilen der Bubble auf Polygone**

  Diese Stufe arbeitet folgendermaßen: jede massive Kante der Bubble wird nacheinander in die Polygonkarte eingeschrieben. Läßt sie sich direkt an ein bestehendes Polygon anfügen, so wird dies getan. Ansonsten wird ein neues Polygon erzeugt. Die Struktur der Bubble garantiert nun, daß diese Verteilung in dem Fall, wo eine Linie direkt an ein Polygon angrenzt, dieses auch eindeutig gefunden wird. Es kann nur passieren, daß eine Kante, die durch eine virtuelle Linie vom zugehörigen Polygon getrennt vorliegt, zu einem neuen eigenen Polygon gemacht wird. Aus diesem Zweck wurde die zweite Stufe entworfen.

- **Stufe 2: Fusionierung zusammengehöriger Polygone**

  Hier wird nun wesentlich aufwendiger getestet, ob nicht getrennte Polygone ein einziges geschlossenes Polygon bilden können. Dazu wird für jedes Polygon versucht es mit einem anderen zu zu einem Ring zusammenzuschließen.

- **Stufe 3: "Bubble"-sorting**

  Hier wird vor allem versucht, die Probleme bei der Exploration auszugleichen, die dadurch entstehen, daß man eine vorgegebene Menge massiver Linien in einer bestimmten Reihenfolge ablegen muß. Wie im Anhang beschrieben wird, muß hier auf ein Näherungsverfahren zurückgegriffen werden. Gerade aber bei dicht aneineinanderliegenden Clustergruppen, wie sie z.B. durch Gitter entstehen können, liefert der hier angewendete Algorithmus relativ schlechte Ergebnisse. Aus diesem Grund wird abschließend noch einmal versucht, die Reihenfolge der massiven Linien zu ändern. Hierzu wird für je zwei benachbarte Linien eines Polygons das Aufkommen virtueller Linien im Falle eines Tauschs ermittelt. Kommen in diesem Fall weniger lange virtuelle Linien zustande, so wird der Tausch tatsächlich vollzogen. Die resultierende Karte muß zwar immer noch nicht optimal sein, aber sie hat wesentlich bessere Eigenschaften, als vor dieser Stufe.

- **Stufe 4: Einfügen virtueller Linien**

  Virtuelle Linien existierten bisher nur implizit. Sie waren dadurch angegeben, daß zwei massive Linien nicht gemeinsame Linienendpunkte besaßen. Die Reinigungsplanung verlangt jedoch eine Karte mit vollständig geschlossenen Polygonzügen. Sie unterscheidet dabei auch nicht, ob es sich bei Linien um virtuelle oder massive handelt. Aus diesem Grund werden die Polygone nun mit virtuellen Linien angereichert. Für die topologische Kartographierung ist es von Bedeutung, ob es sich bei einer virtuellen Linie um eine Tür handelt. Dieser Fall wird anhand der Länge der Linie und der Lage der benachbarten massiven Linien entschieden.

- **Stufe 5: Löschen kolinearer Linie**

  Diese Stufe ist nur deshalb eingeführt worden, weil der Reinigungsplanungsalgorithmus keine benachbarten kolinearen Linien einlesen darf. Dies kann jedoch dann auftreten, wenn sich massive und virtuelle Linien abwechseln, da diese bei der Kartographierung nicht fusioniert wurden. Um die Karte dennoch der Reinigungsplanung vorzulegen, werden hier solche Paare von Linien "leicht verbogen", d.h. die Endpunkte werden solange um einige mm versetzt, bis sie nicht mehr in einer Linie liegen.

- **Stufe 6: Löschen kleiner Polygone**

  Um das Datenaufkommen zu reduzieren und den Rechenaufwand der Reinigungsplanung zu vermindern, wurde sich dafür entschieden, Objekte, die eine Mindestfläche unterschreiten, aus der Polygonkarte zu entfernen. Dieser Test wird hier durchgeführt. Während der Reinigung werden dann solche Objekte vom Piloten umfahren.

Die so entstandene Polygonkarte kann nun der Reinigungsplanung zugeführt werden. Außerdem wird dasjenige Polygon, das die Außenwand des Raums repräsentiert, für die topologische Kartographierung abgelegt. Anhand dieser Struktur kann eine globale Exploration durchgeführt werden, da hier Informationen über das Ausmaß und Lage der Räume vorliegt, sowie Daten über vorhandene Türen zu anderen Räumen.

3.6 Die Grobplanung

Die Grobplanung spiegelt das Vorgehen auf lokaler Ebene wieder, jedoch auf einer wesentlich abstrakteren Sicht. Auch hier wird der Ablauf mit Hilfe einer Konfliktmenge und einer Konfliktlösungsstrategie gesteuert, nur handelt es sich um wesentlich einfacherer Mechanismen, da die Vorarbeit schon während der lokalen Exploration geleistet wurde.

Die Menge der interessanten Punkte besteht hier aus offenen Türen, die bisher nur von *einer* Seite eingesehen wurden. Fährt man durch sie hindurch, so wird ein Raum betreten, der noch nicht erkundet wurde. Es wird eine lokale Exploration eingeleitet, in deren Verlauf der Raum erfaßt wird. Hiernach kann die betreffende Tür als potentieller Anlaufpunkt gestrichen werden, da sie von zwei Räumen aus gesehen wurde. Durch die Exploration des Raumes sind aber möglicherweise neue Türen entdeckt worden. Somit setzt sich dieser Vorgang fort, bis keine Tür mehr existiert, die nur von einer Seite gesehen wurde.

Bei der Auswahl des nächsten Ziels werden weniger Kriterien herangeholt, als bei der lokalen Exploration. Da grundsätzlich alle Räume exploriert werden müssen, ist es wenig hilfreich, den zu erwartetenden Informationsgewinn eines Teilziels abzuschätzen. Von alleinigem Interesse dürfte nur der Weg dorthin sein. Somit wurde hier das nächste Teilziel nur nach der Entfernung vom aktuellen Standort aus ermittelt.

Ist ein Teilziel ersteinmal gefunden, muß ein Weg dorthin geplant werden. Dieser wird auf einer topologischen Karte durchgeführt, die sukzessive aus den Ergebnissen der lokalen Exploration aufgebaut wurde. Die Ergebnisse sind so verdichtet, daß nur noch Informationen über das Ausmaß der Räume, sowie Lage der Türen übrigblieb. Diese Informationen kann man nun zu einem bipartiten Graph zusammenfügen, d.h. zu einem Graph, der sich in zwei Klassen einteilen läßt, von denen jedes Element nur mit Elementen der jeweils anderen Klasse verbunden ist. Die Klassen sind hier Türen und

Räume. Gleiche Türen, die aus verschiedenen Räumen gesehen wurden, liegen hierbei nur als *ein* Knoten vor.

Die Suche nach dem kürzesten Weg reduziert sich auf eine einfache Graphensuche. Dabei wird zuerst davon ausgegangen, daß die Strecken innerhalb von Räumen in direkter Linie befahrbar sind. Später wird der Navigator zwar hier noch den konkreten Weg planen müssen, jedoch macht man bei dieser Vereinfachung keinen großen Fehler.

Da es sich bei Gebäudestrukturen in der Regel nur um einige wenige (max. 20) Räume handelt, wurde bei der Graphensuche keine Optimierung vorgenommen.

Ist der Weg zum nächsten Ziel geplant, wird er abgefahren. Hierbei wird jeweils nur der Pfad bis zur nächsten Tür an den Navigator weitergegeben, d.h. die Wegplanung des Navigator beschränkt sich auf das Innere eines einzigen Raumes. Dieser wäre auch bei der Gesamtplanung überfordert, da er nur auf der geometrischen Ebene navigiert. Die Raum-zu-Raum-Planung ist hingegen auf topologischer Ebene wesentlich einfacher.

Bevor noch näher auf die Wechselwirkung zwischen globaler und lokaler Exploration eingegangen wird, sollen diese gegenübergestellt werden. Man wird feststellen, daß es starke Parallelen bei beiden Abstraktionsstufen gibt.

	lokal	**global**
Kartenobjekte	massive Linien	Räume
Konfliktpunkte	virtuelle Linien	Türen
Konfliktlösung	Entfernung, Informationsgewinn	Entfernung
Navigation	geometrisch mit Navigator	topologische Graphensuche, im Raum geometrisch

Tabelle 3.6.1: Gegenüberstellung von lokaler und globaler Exploration

3.7 Rückkopplungen

In der bisherigen Ausführung sind globale und lokale Exploration stark hierarchisch angelegt, d.h. die Ergebnisse der Erkundungen im Raum werden der topologischen Kartographie zugeführt, es findet jedoch kein Informationsaustausch in umgekehrter Richtung statt.

Diese Einteilung ist sehr bequem, da eine Komponente einer niedrigeren Abstraktions-

stufe (hier lokale Exploration) nur Entscheidungen auf der Basis noch untergeordneter Stufen treffen muß. Deshalb findet man eine stark hierarchische Gliederungen auch in allen Bereichen der Sensordatenverarbeitung.

In unserem speziellen Fall gibt es jedoch Probleme, die nicht optimal gelöst werden können, wenn man nur Zugriff nach unten hat. Die lokale Exploration soll schon auf der Basis von Linien, Konstruktionen wie Räume und Türen erkennen, was sehr schwer und fehleranfällig ist. Aus diesem Grund wurde das stark hierarchische Prinzip durchbrochen und eine Rückkopplung von der globalen zur lokalen Exploration vorgesehen. Bei der Exploration eines Raumes liegen also Informationen über andere Räume vor. Die Architektur des Gesamtsystems verändert sich damit folgendermaßen:

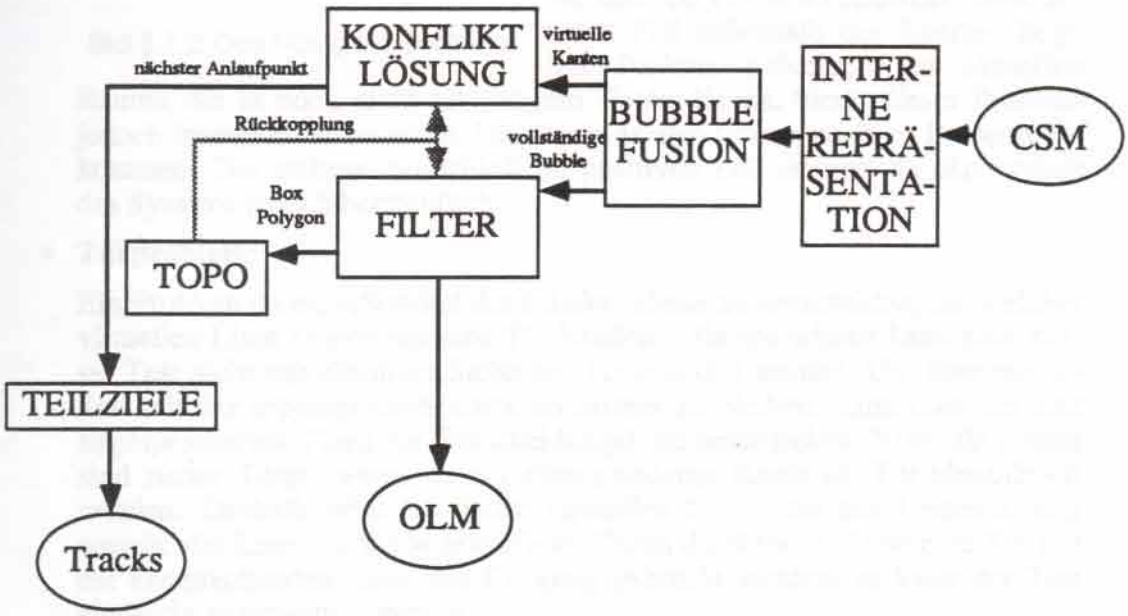


Bild 3.7.1: Endgültige Architektur mit Rückkopplungen

Zur letzten Fassung sind nun zwei Pfade hinzugekommen, die Informationen aus der topologischen Karte zur Konfliktlösung sowie zum abschließenden Filter bewegen. Damit sollen vor allem zwei Probleme bewältigt werden:

- **Navigatorproblem:**

  Der Navigator ist eine Komponente, die rein auf der Geometrie Wege zu Zielpunkten plant. Konstruktionen wie Räume und Türen sind für diesen unbekannt. Ein Weg zu einem Ziel ist somit eine gültige Lösung, auch wenn bei der Fahrt der aktuelle Raum verlassen wird. Somit könnte folgendes Problem entstehen:

  Es wurde ein Konfliktpunkt konstruiert, der außerhalb des Raumes liegt. Dies kann durch Punkte entstehen, die von der Horizonbubble herstammen. Angenommen, dieser Punkt wird nun ausgewählt. Der Navigator bekommt nun einen Fahrbefehl. Doch anstatt die Fahrt dorthin zu verweigern, konstruiert er einen Pfad, der aus dem Raum hinausführt. Es ist nun relativ schwierig für die Exploration herauszufinden, ob das Fahren in eine andere Rich-

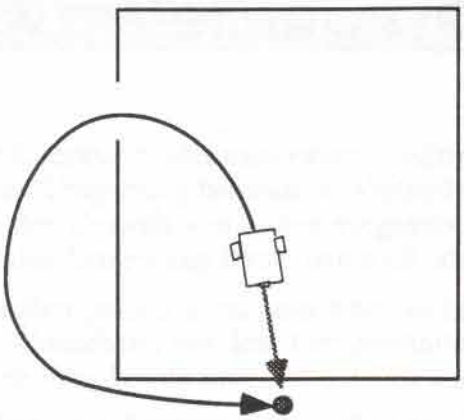


Bild 3.7.2: Das Navigatorproblem

tung davon herstammt, den Raum zu verlassen oder nur eine "normale" Umfahrung ist. Eine Möglichkeit, solche Probleme zu umgehen, ist, zuerst den internen Navigator aufzurufen. Eine andere Möglichkeit ist es, zu testen, ob ein Konfliktpunkt nicht schon in einem anderen Raum liegt. In der topologischen Karte liegt eine Menge *geschlossener* Polygone vor, nämlich die, der schon erkundeten Räume. Anhand dieser kann ein einfacher Test durchgeführt werden, ob ein Konfliktpunkt nicht auf jeden Fall außerhalb des Raumes liegt. Bei Punkten außerhalb des aktuellen Raums, die in noch nicht erkundetem Gebiet liegen, bleibt dieses Problem jedoch immer noch bestehen. Hier kann es also immer noch zu Fehlerfällen kommen. Der sichere Ausschluß im positiven Fall steigert die Robustheit des Systems jedoch beträchtlich.

- **Türproblem:**

  Ein Problem ist es, schon auf der lokalen Ebene zu entscheiden, bei welcher virtuellen Linie es sich um eine Tür handelt. Wie wir wissen kann auch dieser Test nicht mit absoluter Sicherheit entschieden werden. Um aber zumindest mit der topologischen Karte konsistent zu bleiben, kann man die dort abgespeicherten Türen zur Entscheidungshilfe heranziehen. Virtuelle Linien sind *sicher* Türen, wenn sie aus einem anderen Raum als Tür identifiziert wurden. Deshalb wird bei jeder virtuellen Linie, die zur Untersuchung ansteht, die Liste der schon erkundeten Türen durchsucht. Kann eine Tür mit der entsprechenden Linie zur Deckung gebracht werden, so kann der Test eindeutig entschieden werden.

Die beiden Rückkopplungsverbindungen erhöhen hierbei die Wahrscheinlichkeit, daß der Roboter den aktuellen Raum nicht verläßt, und zwar auf der Basis schon gefundener Räume und Türen. Die Beschreibung der lokalen und globalen Exploration ist hiermit komplett.

4. Vorstellung weiterer Verfahren

In der Literatur findet man kaum Ansätze, die das Problem der Erkundung einer unbekannten Umgebung behandeln. Vielmehr wird meistens davon ausgegangen, daß eine Karte der Umwelt von außen eingegeben wurde, und die Sensorik nur die Differenz zur realen Umgebung feststellen muß oder Positionsfehler erkennen soll.

Hier sollen jedoch zumindest zwei weitere Ansätze vorgestellt werden. Der erste hat dabei Ähnlichkeit mit dem hier gewählten Vorschlag, während der zweite einen völlig anderen Weg beschreitet.

Am Ende des Kapitels sollen alle vorgestellten Ansätze miteinander verglichen werden, um so Vor- und Nachteile aufzudecken.

4.1 Das Verfahren von Iijima, Asaka und Yuta

Dieses Verfahren wurde in [Iijima89] vorgestellt. Auf der Basis eines Entfernungssensors soll eine 2D-Karte eines Raums erstellt werden. Diese Karte soll eine Menge geschlossener Polygonzüge enthalten, die alle Objekte umschließen sowie die Außenwand des Raums repräsentiert. Die Ausgabekarte ist somit mit einer Outline Map vergleichbar. An die zu explorierende Umwelt werden folgende Bedingungen gestellt:

- Die Umgebung ist vom Roboter vollständig befahrbar.
- Die Umgebung besteht aus genau einem Raum. Dieser muß geschlossen sein.
- Die Oberflächen der Objekte im Raum sind entweder vertikal oder horizontal.
- Es muß möglich sein, die Grenze zwischen Boden und Wand oder Boden und Objekt mit einer Kamera erkennen zu können.

Das vorgestellte System befaßt sich nicht allein mit der Exploration, sondern übernimmt auch Aufgaben der Sensorvorverarbeitung, sowie der Positionskorrektur. Von der Sensorik wird eine Karte zur Verfügung gestellt, die noch unkorreliert ist (*Range Map*).

Linien dieser Karte werden von Positionsfehlern bereinigt und in eine Polygonkarte (*Vector Map*) eingetragen. Die Art der Kartenfusionierung ist hierbei mit dem ersten Ansatz aus Kapitel 3.2 vergleichbar. Die Darstellung des Unwissens über die Umwelt erfolgt rasterorientiert. Eine Karte (*Cell Map*), bestehend aus kleinen Quadraten gibt Auskunft darüber, welche Bereiche schon abgetastet wurden, und welche noch angefahren werden müssen.

Der nächste Anlaufpunkt ist einer aus der folgenden Menge:

- Endpunkt der Linie, die gerade verfolgt wird
- Endpunkt einer Linie, die die gerade verfolgte Linie verdeckt

- Der nächste Endpunkt eines noch offenen Polygons
- Der nächste Rasterpunkt, der noch nicht als exploriert markiert wurde

Die Priorität dieser Möglichkeiten ist von oben nach unten abfallend. Somit entsteht eine Wandverfolgungsstrategie. Die Exploration gilt als abgeschlossen, wenn alle Polygone geschlossen und alle Rasterpunkt exploriert sind.

Ein neuer Anfahrpunkt wird erst dann berechnet, wenn der aktuelle erreicht worden ist. Somit können die in Kapitel 3.5.1 beschriebenen Probleme eintreten. Der nächste Anfahrpunkt, muß immer direkt befahrbar sein. Diese Bedingung entsteht dadurch, daß kein Navigationsalgorithmus zur Verfügung steht. Die Probleme, die dadurch auftreten, daß ein potentieller Anfahrpunkt nicht erreichbar ist, werden somit ausgeklammert.

4.2 Das Verfahren von Lumelsky, Mukhopadhyay und Sun

Das folgende Verfahren wurde in [Lumelsky89] vorgestellt. Es arbeitet auch mit einem Entfernungssensor und erzeugt eine 2D-Repräsentation der Umwelt. Diese darf nur aus einem einzigen geschlossenen Raum bestehen, der zudem noch rechteckig sein muß.

Die Grundidee besteht darin, den Raum in parallelen Bahnen abzufahren. Liegen diese Bahnen dicht genug, so wird jedes Objekt zumindest erst einmal von einer Seite eingesehen. Zur genaueren Untersuchung, kann die Hauptbahn zur einmaligen Umrundung des Objekts verlassen werden. Danach wird fortgefahren, den Raum in Streifen abzufahren. Folgende Möglichkeiten bestehen bei Objekten im Raum:

- Ein Objekt kann allein, indem die Hauptbahn abgefahren wird, vollständig erfaßt worden sein. Solche Objekte erfordern keine besondere Behandlung.
- Ein Objekt kann die Hauptbahn schneiden. Das Objekt muß einmal umrundet werden, um es zu erfassen, und noch einmal halb, um wieder auf die Hauptbahn zu kommen.
- Ein Objekt liegt zwar nicht auf der Hauptbahn, kann aber von dort aus nicht vollständig eingesehen werden. Solche Objekte werden von der Hauptbahn aus in direktem Weg angefahren, einmal umrundet und wieder verlassen.

Die Grundidee, sowie die drei Objekttypen sind im nächsten Bild zu sehen.

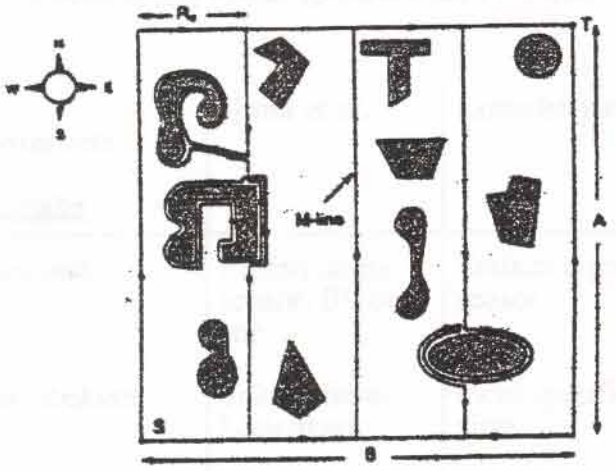


Bild 4.2.1: Grundidee des Verfahrens (aus (Lumelsky89))

Diese Methode der Exploration bringt jedoch einige ungelöste Fragen mit sich:

- Wie kann eine flächendeckende Bahn berechnet werden, *bevor* eine Exploration durchgeführt wurde?
- Was passiert, wenn ein Hindernis nicht so umfahren werden kann, daß wieder die Hauptbahn erreicht wird?
- Was passiert, wenn es überhaupt nicht umfahren werden kann, weil es z.B. an einer Wand steht?
- Wie kann ein Hindernis so umfahren werden, daß es vollständig eingesehen wird?

Fragen und Probleme, die sich mit dem flächendeckenden Abfahren von Räumen, sowie dem Umfahren von Hindernissen in einem bestimmten Abstand befassen, sind ausführlich in [Ramsbott91] abgehandelt worden.

4.3 Gegenüberstellung der vorgestellten Verfahren

Folgende Verfahren sollen gegenübergestellt werden:

- Das Verfahren von Iijima, Asaka und Yuta
- Das Verfahren von Lumelsky, Mukhopadhyay und Sun
- Der erste eigene Ansatz
- Die endgültige Version, basierend auf dem Bubble-Verfahren

Obwohl der erste Ansatz noch nicht vollständig konzipiert war, bevor er verworfen wurde (es fehlte die Darstellung des Unwissens, sowie die globale Strategie), bietet er

dennoch genug Anhaltspunkte zum Vergleich.

Kriterien:	Ijima et al.	Lumelsky et al.	1. eigener Ansatz	endgültige Version
Eingabe				
Sensoren	Entfernungs-sensor, Bildsen-sor	Entfernungs-sensor	Entfernungs-sensor	Entfernungs-sensor
Eingabekarte	unkorrelierte Linienkarte	nicht spezifi-ziert	korrelierte Linienkarte	korrelierte Linienkarte
Bedingungen	ein Raum, alle Objekte vertikal oder horizontal, alles befahrbar	ein rechteckiger Raum, alles befahrbar	keine Bedin-gungen	keine Bedin-gungen
Dynamische Objekte	nicht erlaubt	nicht erlaubt	nur soweit von CCG entdeckt	nur soweit von CCG entdeckt
Ausgabe				
Ausgebekarte	Polygonkarte	nicht spezifi-ziert	Polygonkarte	Polygonkarte
Dimensionen	2	2	2	2
Algorithmus				
Umfang	Explorator, Korrelator	nicht spezifi-ziert	nur Explorator	nur Explorator
Hierarchie	nicht notwendig	nicht notwendig	noch nicht spe-zifiziert	lokale und glo-bale Explora-tion
Navigation	nicht vorhanden	implizit	noch nicht spe-zifiziert	lokal:NAV, global: Gra-phensuche
interne Karten:				
Hindernisrepräs.	Linien	Linien	Linien	Linien
Freiraumrepräs.	Raster	Bahnen	noch nicht spe-zifiziert	Aufnahmepositi-onen
Erhebung Kon-fliktmenge	Erst nach Anlaufen eines Punktes	keine Konflikt-lösung vorhan-den	nach vorgege-bener Distanz	nach vorgege-bener Distanz

Tabelle 4.3.1: Vergleich der behandelten Verfahren

5. Benutzersicht auf das System

Im folgenden soll beschrieben werden, wie ein Benutzer die Explorationskomponente handhaben kann. Das Programm wurde unter MacApp (siehe [Apple89]) implementiert, einer objektorientierten Bibliothek von Methoden zur Steuerung der Benutzeroberfläche. Diese unterstützt die Entwicklung soweit, daß dabei Programme gemäß den Macintosh-Richtlinien [Apple87] entstehen. Somit dürfte die Programmbedienung zumindest für den Macintosh erfahrenen Benutzer kein Problem sein.

5.1 Beschreibung der Menüpunkte

Das Programm umfaßt, abgesehen vom Apfelmenü, fünf Menüpunkte.

Bild 5.1.1: Die Menüzeile der Explorationskomponente

Zwei davon sind auch in andereren Programmen vertreten, nämlich das File- und das Edit-Menü. Alle Menüs werden im folgenden beschrieben:

Bild 5.1.2: Das File-Menü

Alle Einträge des File-Menüs kann man auch in anderen Programmen wiederfinden. Der wichtigste dürfte hier der Punkt **'Open'** zum Öffnen einer Datei sein. Hiermit kann man eine Karte einlesen, auf der der Explorationsalgorithmus arbeiten soll. Diese

Karte kann im Umgebungseditor erzeugt, und als *Environment Edge Map* gesichert werden.

Dieser Punkt öffnet neben der Datei das Haupt-Dokumentfenster. Von dort aus kann das Programm gesteuert und Aktionen eingeleitet werden. Die genaue Beschreibung dieses Fensters erfolgt später.

Da die Programmumgebung es ermöglicht, mehrere Dokumente zu verwalten, ist es möglich, mehrmals 'Open' auszuführen und mit mehreren verschiedene Karten zu arbeiten. Der eigentliche Algorithmus besitzt aber seine Variablen nur ein einziges Mal, so daß diese beim Umschalten zwischen verschiedenen Dokumenten gelöscht werden, um Inkonsistenzen zu vermeiden. Somit ist es zwar möglich mehrere Karten gleichzeitig einzuladen, aber es kann immer nur auf einer gearbeitet werden.

Mit **'Close'** wird das jeweils vorderste Fenster geschlossen. Dieselbe Wirkung hat das Klicken auf die jeweilige Go-Away-Box. Wird das Haupt-Dokumentfenster geschlossen, so werden zusätzlich alle temporären Variablen freigegeben.

Fast zu jeder Zeit kann das jeweils vorderste Fenster auf einen Drucker ausgegeben werden. In diesem Fall kann das mit **'Print'**-Kommando geschehen.

Mit **'Quit'** wird das Programm verlassen.

Es existiert keine Möglichkeit, ein Karte zu speichern, da diese in dem Explorationsprogramm nicht verändert wird. Die üblichen Menüs 'Save' sowie 'Save As' entfallen somit. Alle in Frage kommenden Dateien können über das Haupt-Dokumentfenster gespeichert werden.

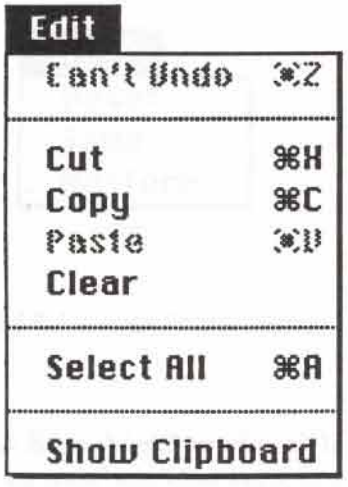


Bild 5.1.3: Das Edit-Menü

Bei diesem Menü handelt es sich um das Standard-Edit-Menü, so wie es in den meisten Programmen zu finden ist. Hier sind vor allem Befehle vertreten, die das Clipboard betreffen. Alle Dialogeinträge können dort abgelegt werden. Außerdem besteht die Möglichkeit, den Inhalt des Dokumentfensters ins Clipboard zu legen. Dort kann es z.B. in Graphikprogramme zur Weiterverarbeitung portiert werden.

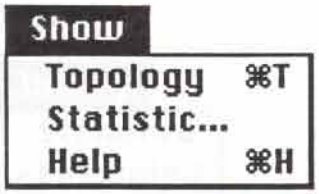


Bild 5.1.4: Das Show-Menü

Im Show-Menü hat man die Möglichkeit zusätzliche Fenster zu dem Haupt-Dokumentfenster zu öffnen. Mit **'Topology'** kann das topologische Raumnetz angezeigt werden, soweit schon eines vorliegt. Ist es erst einmal angezeigt, so wird es während eines Laufs sukzessive erweitert.

Mit **'Statistic'** kann man sich markante Daten über einen Explorationslauf anzeigen lassen, soweit diese schon erhoben wurden. Genaueres dazu wird in einem späteren Kapitel dargelegt.

'Help' öffnet ein Fenster zur Online-Hilfe. Auf diesem Fenster werden Hilfstexte zu dem Teil angezeigt, auf den der Mauscursor gerade zeigt. Der Fensterinhalt ändert sich dabei laufend in Abhängigkeit zur Mausposition, ohne daß eine zusätzliche Eingabe vom Benutzer erwartet wird. Möchte man die Online-Hilfe verlassen, so muß nur das entsprechende Fenster geschlossen werden.

Bild 5.1.5: Das State-Menü

Das State-Menü erwies sich als unentbehrliche Hilfe zur Programmentwicklung. Hier ist es möglich, den aktuellen Programmzustand mit **'Save'** auf ein Speichermedium zu retten, und zu jeder Zeit exakt mit **'Restore'** zu restaurieren. Der Grund dafür besteht darin, daß man versucht, charakteristische Problemsituationen zu erzeugen, wie sie in Testläufen entstehen. Den entsprechenden Zustand kann man dann retten, und in der Pascal-Entwicklungsumgebung restaurieren, wo man Debugging-Möglichkeiten besitzt. Diese Situationen direkt in der Entwicklungsumgebung zu erzeugen wäre sehr zeitaufwendig, da ein Lauf bis zu zehn mal langsamer wird, als in der fertigen Applikation.

Mit **'Reset'** versetzt man den Algorithmus in einem definierten Anfangszustand, in dem nichts über die Umwelt bekannt ist. Man kann diesen Punkt anwählen, wenn man einen schon explorierten Raum noch einmal erkunden lassen möchte. Der Algorithmus weigert sich einen Raum zweimal zu explorieren, deswegen ist ein Rücksetzen not-

wendig.

Bild 5.1.6: Das Dialog-Menü

Dieses Menü erzeugt keine expliziten Reaktionen. Hier kann der Benutzer definieren, welche Art Dialog er im Haupt-Dokumentdialog angezeigt bekommen möchte. Dies muß geschehen *bevor* eine Karte geöffnet wird. Nachträgliche Änderungen haben erst eine Wirkung auf spätere Dokumente. Beim Programmstart ist dieses Menü auf **'Standard'** eingestellt. Für die meisten Testläufe sind die dann einstellbaren Parameter ausreichend. Für Verfeinerungen ganz spezieller Parameter kann **'Extended'** eingestellt werden. Die zusätzlichen Einstellmöglichkeiten erfordern jedoch einige Erfahrungen und die Wirkungen sind oft nicht direkt einzusehen, so daß hier in der Regel keine Änderungen erfolgen sollten. Eine eingehendere Beschreibung der Parameter findet im nächsten Kapitel statt.

Damit sind alle Menüpunkte beschrieben, und es soll sich jetzt auf das Haupt-Dokumentfenster konzentriert werden, da hier die meisten Einstellmöglichkeiten bestehen.

5.2 Beschreibung des Dokumentfensters

Das Haupt-Dokumentfenster stellt sich nach dem 'Open' Befehl wie folgt dar:

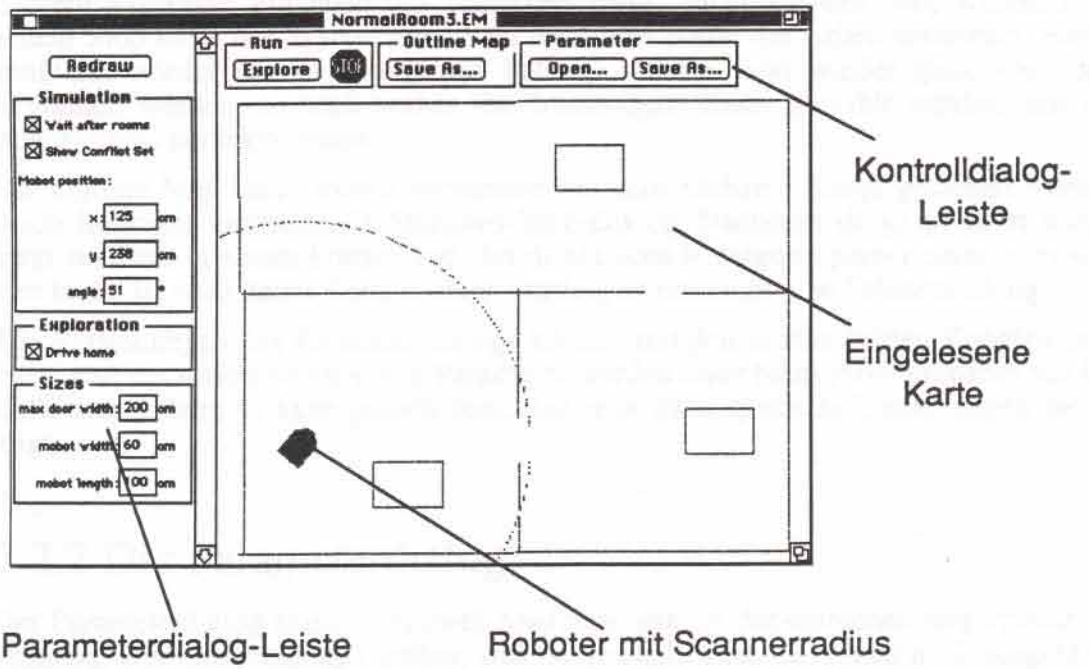


Bild 5.2.1: Das Haupt-Dokumentfenster

Das Fenster ist dreigeteilt. Oben ist das Kontrolldialogfenster dargestellt. Hier kann der Algorithmus gestartet und Dateioperationen vorgenommen werden. Links befindet sich der Parameterdialog. Alle Einstellungen, die den Algorithmus betreffen können hier vorgenommen werden.

Den größten Raum nimmt die zu explorierende Karte ein. Dort befindet sich auch der Roboter, der mit einem Kreis versehen ist, um die Scannerreichweite anzudeuten. Die aktuelle Roboterposition, sowie die Richtung kann einfach mit der Maus eingestellt werden. Hierbei wird die Maustaste an einer entsprechenden Position gedrückt, und durch Ziehen bei gehaltener Taste die Richtung bestimmt. Wird hierbei noch gleichzeitig die Optionstaste gehalten, so 'schnappt' die Position in einem 50 cm Raster und der Winkel im 45° Abstand.

5.2.1 Der Kontrolldialog

Der Kontrolldialog stellt sich wie folgt dar:

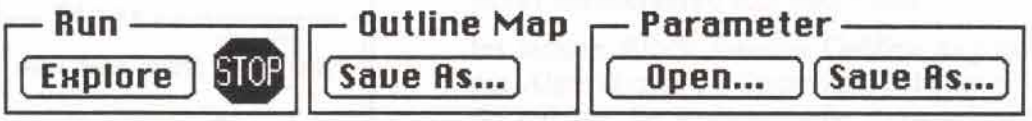


Bild 5.2.2: Der Kontrolldialog

Der Explore-Knopf startet den eigentlichen Algorithmus. Von dann an reagiert das System auf keine Eingaben des Benutzers mehr, außer auf den Stop-Knopf. Nach einem Stop kann das System wieder an derselben Stelle die Arbeit fortsetzen. Hierzu muß nur wieder der Explore-Knopf betätigt werden. Soll wieder ganz von vorne begonnen werden, so muß vorher das Menü State-Reset gewählt werden, um den Algorithmus zurückzusetzen.

Die Outline Map kann, soweit vorhanden mit dem nächsten Knopf gesichert werden. Nach Betätigen erscheint ein Standard-Save-Dialog. Nachdem sie gespeichert wurde, liegt sie dann in einem Format vor, das direkt vom Reinigungsplaner eingelesen werden kann. Ist noch keine Karte erzeugt worden, so erscheint eine Fehlermeldung.

Die Einstellungen des Parameterdialogs können mit den letzten beiden Knöpfen gesichert und restauriert werden. Die Parameter werden zwar beim Programmstart mit Initialwerten belegt, es kann jedoch sein, daß man ganz spezielle Einstellungen bevorzugt.

5.2.2 Der Parameterdialog

Der Parameterdialog existiert in zwei Ausführungen, in der einfachen und erweiterten Fassung. Die Entscheidung darüber, welche gewählt wird, kann man im Dialog-Menü festlegen. Die einfache Version präsentiert sich wie folgt:

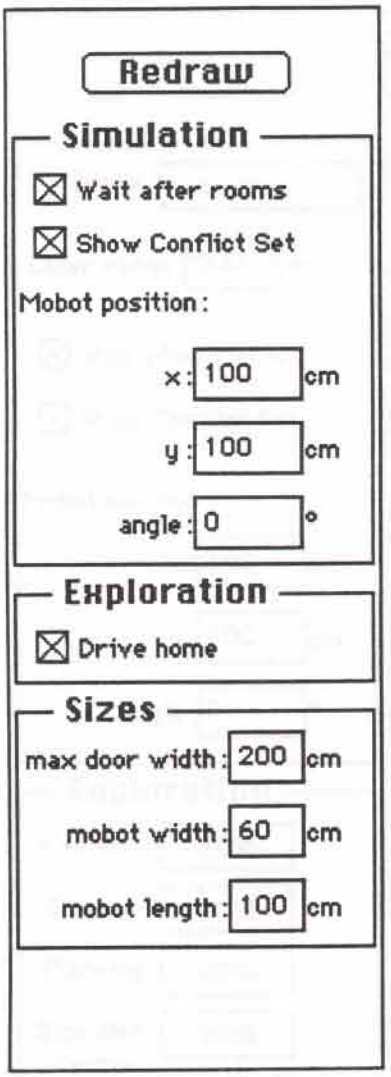


Bild 5.2.3: Der einfache Parameterdialog

Mit 'Redraw' wird die Karte neu gezeichnet. Dies ist dann notwendig, wenn man Robotereinstellungen wie z.B. Position über diesen Dialog geändert hat. Dieselbe Wirkung hätte die Returntaste.

Im Bereich, der mit 'Simulation' überschrieben ist, kann man Einstellungen vornehmen, die nicht direkt den Algorithmus betreffen. So kann man mit 'Wait after room' festlegen, ob der Explorationslauf nach jedem Raum eine Pause machen soll. In diesem Fall wird der Lauf mit dem Explore-Knopf des Kontrolldialogs fortgesetzt. Mit der zweiten Box kann man festlegen, ob die Konfliktmenge während der Darstellung laufend gezeigt werden soll. Die letzten drei Fenster dienen der Einstellung von Roboterposition und Winkel, wenn man diese nicht mit der Maus in der Karte vornehmen möchte.

Mit der Box 'Drive Home' wird festgelegt, ob der Roboter nach getaner Arbeit wieder zum Startpunkt zurückfahren soll oder nicht.

Im letzten Block werden Größen angegeben, die der Algorithmus zu Steuerung unbedingt benötigt. So werden die Robotermaße dazu gebraucht festzustellen, ob ein Bereich befahren werden kann oder nicht. Die maximale Türgröße wird erhoben um festzustellen, ob eine Raumöffnung eine Tür ist, oder nur zu einem Bereich führt, der noch zum selben Raum gehört.

Der gesamte Dialog ist scrollbar, d.h. sollte der zur Verfügung stehende Raum des Fensters nicht ausreichen, erscheint rechts neben dem Dialog ein Balken, mit dem man den sichtbaren Bereich festlegen kann.

Der erweiterte Parameterdialog ist im nächsten Bild dargestellt. Hier stehen nun soviele Einstellmöglichkeiten zur Verfügung, daß der Dialog hier aus Platzgründen zweispaltig dargestellt wurde.

Im folgenden werden nur die Änderungen bezüglich des einfachen Dialoges dargestellt.

Im Block 'Simulation' sind zwei Neuerungen hinzugekommen. Zum einen kann man die mittlere Fahrgeschwindigkeit einstellen. Es stehen die Stufen slow (30 cm/s), medium (60 cm/s) und fast (1 m/s) zur Verfügung. Zum anderen kann man die Reichweite des Laserscanners verändern. Diese steht standardmäßig auf 5 m. Man kann hier beispielsweise kleinere Werte eingeben, um die Reaktion bei anderen Sensorkonfigurationen zu testen.

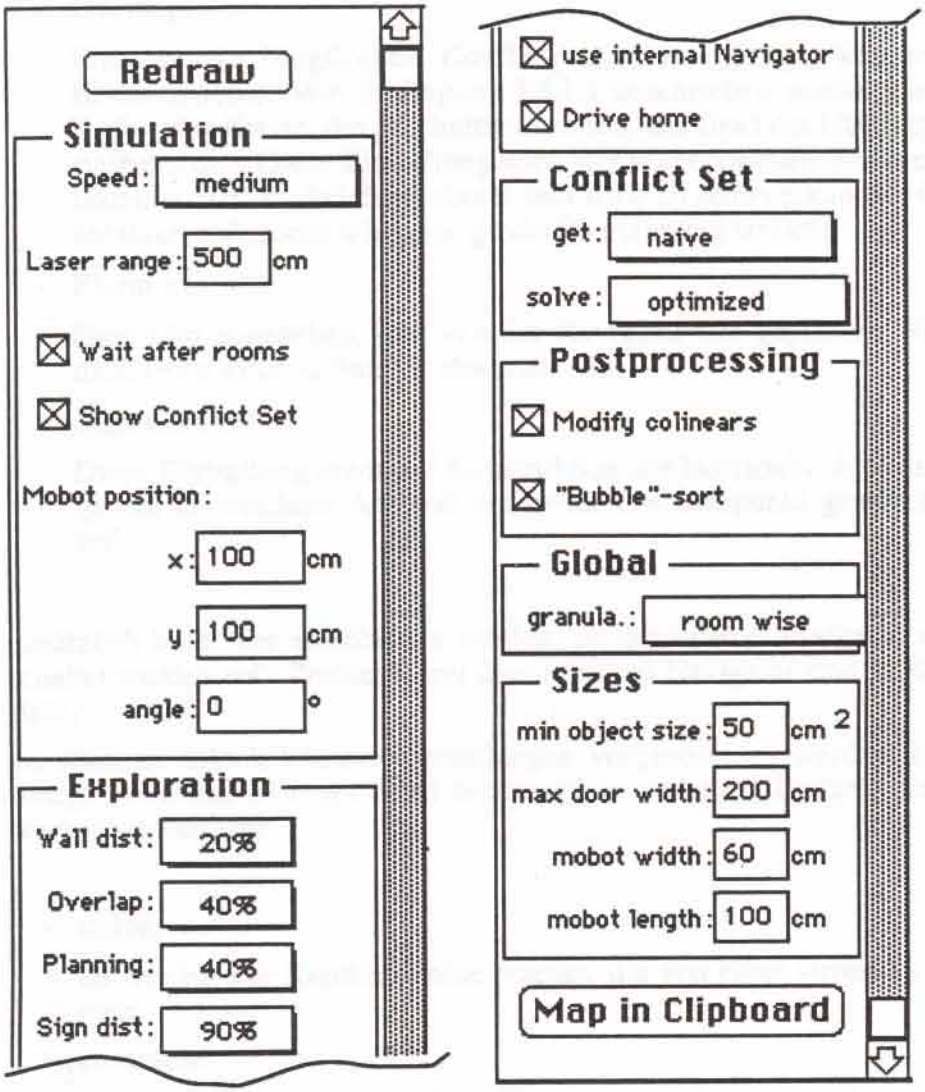


Bild 5.2.4: Das erweiterte Parameterdialog

Im Block 'Exploration sind nun mehrere Einstellungen hinzugekommen. Die ersten vier werden alle in % angegeben, womit das Verhältnis zur Scannerreichweite gemeint ist. Da diese direkt im Zusammenhang mit der Reichweite des Sensor stehen, können so verschiedene Reichweiten getestet werden, ohne diese Größen ändern zu müssen.

Im einzelnen sind folgende Einstellungen möglich:

- **Wall distance:**

  Bei der Erzeugung der Konfliktpunkte werden Punkte in der Nähe von massiven Linien in den Freiraum hinein projiziert (siehe Kapitel 3.5.1.a). Hier wird angeben wie weit die Projektion erfolgen soll. Da dies der Abstand ist, der im Grenzfall (Wandverfolgung) zur Wand eingehalten wird, ist der Name 'Wandabstand' sinnvoll.

- **Overlap:**

  Eine weitere Möglichkeit Konfliktpunkte zu erzeugen kommt durch die Horizonbubble. Wie in Kapitel 3.5.1.a beschrieben wurde, kann mit dem Radius der Kreise, die geschnitten werden, der Grad der Überlappung eingestellt werden. Diese Einstellung wird hier vorgenommen. Ein niedriger Wert bedeutet hierbei, daß der Roboter sehr nahe an schon erkundete Gebiete heranfahren soll, somit wird eine große Überlappung erreicht.

- **Planning:**

  Hier wird angegeben, wie weit der Navigator den geplanten Weg abfahren darf, bevor er neue Befehle abwarten soll.

- **Sign distance:**

  Diese Einstellung dient der Konstruktion der Horizonbubble. Es wird angegeben, in welchem Abstand ein neuer Positionspunkt gespeichert werden soll.

Zusätzlich kann hier entschieden werden, ob der interne Navigator vollständig abgeschaltet werden soll. Probleme mit dem externen Navigator sind in Kapitel 3.7 dargestellt.

Im nächsten Block können Einstellungen vorgenommen werden, die die Konfliktmenge betreffen. Mit 'get' wird festgelegt, in welchem Umfang die Konfliktmenge erzeugt werden soll:

- **naive:**

  Es werden nur Konfliktpunkte erzeugt, die von einer virtuellen Linie stammen:

- **complete:**

  Es werden zusätzlich Konfliktpunkte aus der Horizonbubble erzeugt, so daß ein flächenmäßges Abfahren garantiert werden kann.

Mit 'solve' wird festgelegt, ob die Lösung nur auf die lokalen Gewichte der Konfliktpunkte ausgelegt wird, oder ob hierzu noch die Kriterien Entfernung zum Roboter, sowie Drehwinkel hinzugenommen werden sollen.

Zur abschließenden Kartenerzeugung stehen mehrere Filter zur Verfügung (siehe Kapitel 3.5.2). Einige davon können mit 'Postprocessing' abgeschaltet werden. Der eine Filter ist die Routine, die kolineare Linien verhindert, der andere die 'Bubble'-Sortierung. Diese Filter sind nur zu Testzwecken nach außen zugänglich gemacht worden. Es ist unwahrscheinlich, daß man sie wirklich abschalten muß.

Genauso verhält es sich beim nächsten Punkt. Standardmäßig steht die Granularität auf 'room wise', d.h. es soll ein Raum nach dem anderen exploriert, und dabei eine topologische Karte aufgebaut werden. Die Möglichkeit 'total' wurde nur zu Testzwecken eingebaut. Dort wird nicht versucht, Türen zu erkennen, sondern alles wird in ein einziges Raummodell eingeordnet. Damit sollte untersucht werden, wie sich eine totale Exploration auswirkt (siehe Zusammenfassung).

Bei 'sizes' ist nur ein Maß hinzugekommen. Die minimale Objektgröße ist die Fläche

unter der ein Polygon aussortiert wird. Der Wert "0" bedeutet dabei, daß alle Objekte genommen werden.

Als letztes besteht die Möglichkeit, die Karte in das Clipboard zu schreiben, um sie anderen Programmen zugänglich zu machen. Mit dem Knopf 'Map in Clipboard' kann dies geschehen. Man kann den Erfolg mit dem Menü Edit-Show Clipboard kontrollieren.

5.3 Weitere Ausgaben

Hier sollen nun solche Fenster beschrieben werden, die nicht im Haupt-Dokumentfenster liegen.

5.3.1 Das Topologie-Fenster

Mit dem Menü Show-Topology kann man das Fenster mit der bis dahin erarbeiteten topologischen Struktur öffnen. Dieses könnte sich dann wie im nächsten Bild präsentieren:

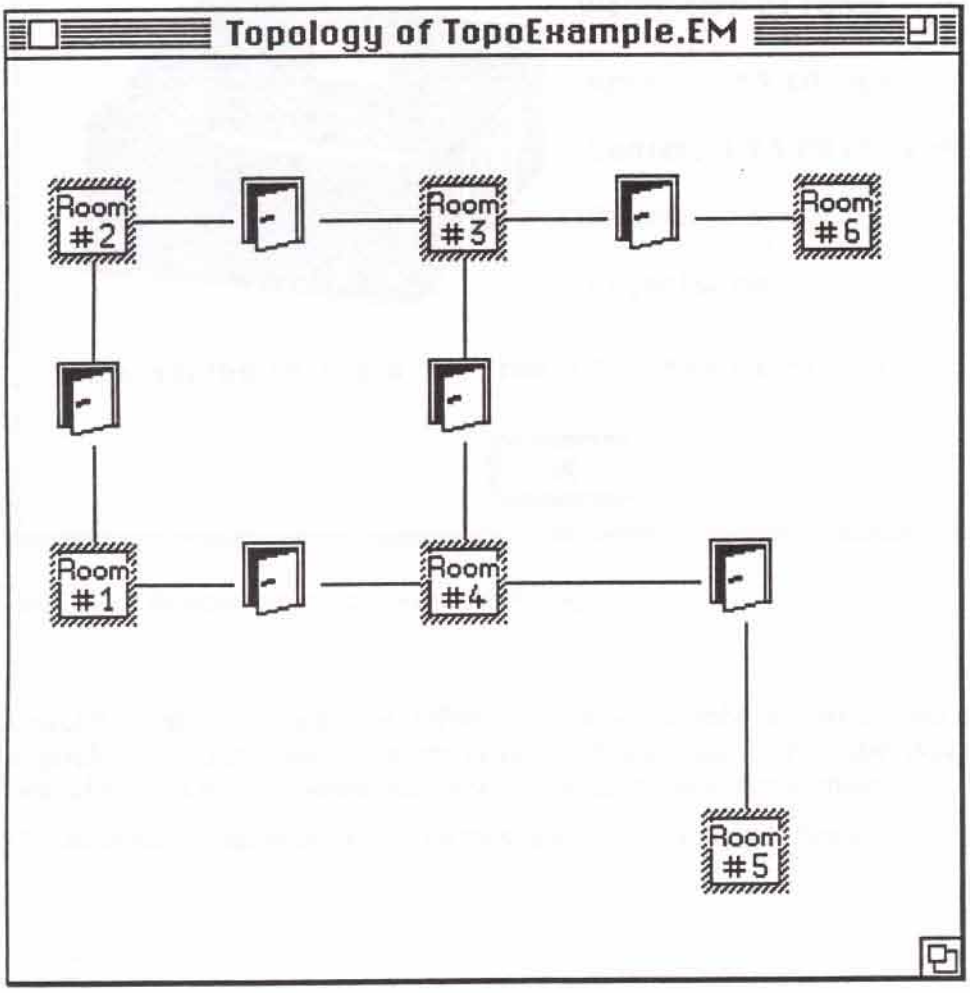


Bild 5.3.1: Beispiel eines Topologie-Fensters

Die Räume sind mit Icons dargestellt, die die Aufschrift 'Room#1' bis 'Room#9' tragen. Sollten mehr als neun Räume vertreten sein, werden alle Folgeräume mit 'Room#...' angezeigt. Alle Türen werden symbolisch durch geöffnete Türen dargestellt. Der Ort der Symbole richtet sich nach dem Ort im Weltmodell. Alle Türsymbole werden an der Position gezeigt, wo der Mittelpunkt der entsprechenden Linien liegt. Alle Räume werden an dem Ort dargestellt, wo der Mittelpunkt des entsprechenden Polygons liegt.

Somit stellt sich die Topologie zunächst etwas unsortiert dar. Man kann jedoch alle Symbole mit dem Mauszeiger an andere Positionen ziehen. Dazu muß man ein Symbol anwählen und mit gehaltener Maustaste das entsprechende Rechteck, was nun an der Stelle des Symbols erscheint, an die gewünschte Stelle schieben. Hält man zusätzlich noch die Optionstaste, so schnappt der Endpunkt in einem Raster.

Mit einem Doppelklick auf ein Symbol kann man sich Informationen über das entsprechende Objekt anzeigen lassen. Das nächste Bild zeigt einen Informationsdialog eines Raums:

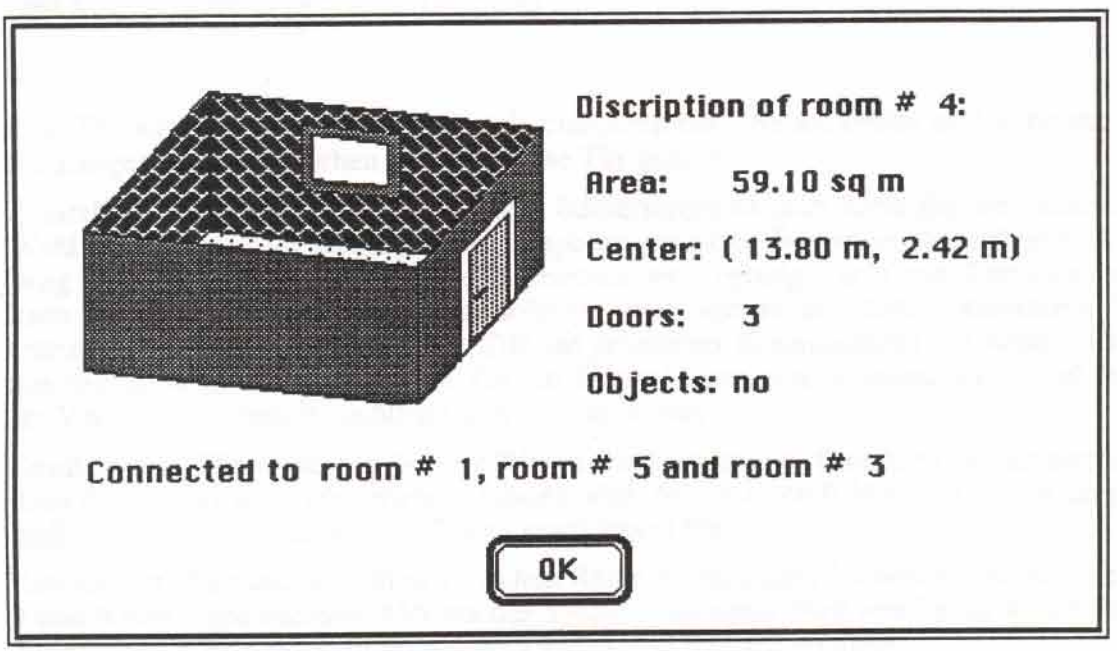


Bild 5.3.2: Beispiel einer Raumbeschreibung

Gezeigt werden geometrische Informationen wie Fläche und Mittelpunkt, sowie topologische Informationen. Letztere sind die Anzahl der Türen, die Anzahl der Raum befindlichen Objekte, sowie mit welchen Räumen er verbunden ist.

Genauso kann man sich Informationen über Türen anzeigen lassen.

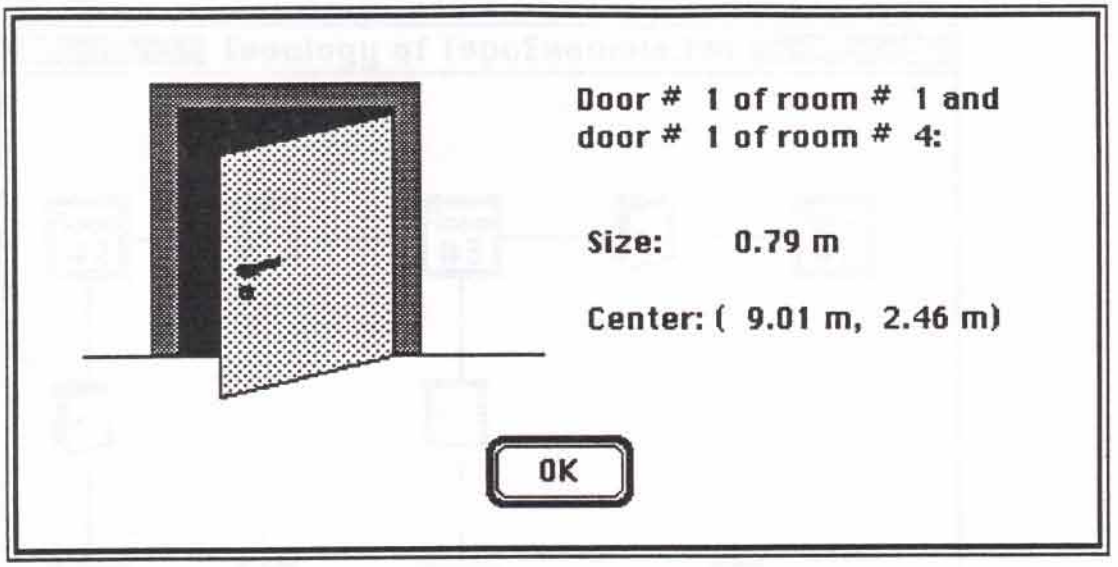


Bild 5.3.3: Beispiel einer Türbeschreibung

Eine Tür wird durch weniger Merkmale charakterisiert. Neben Größe und Mittelpunkt wird angegeben, zu welchen Räumen diese Tür gehört.

Zusätzlich zu den bisher beschriebenen Editiermöglichkeiten kann der Benutzer die Wahl eines Weges im topologischen Graph im gewissen Schranken beeinflussen. Ein Weg wird auf der Basis der bis dahin gewonnenen Topologie ermittelt. Der Benutzer kann nun verbieten, daß in einem solchen Weg bestimmte Türen vorkommen. Er erreicht dies durch Anklicken einer Tür mit gehaltener Kommandotaste. Dadurch wird aus dem Symbol einer geöffneten Tür ein Symbol einer geschlossenen Tür. Wird dieser Vorgang wiederholt, so öffnet sich die Tür wieder.

Geschlossene Türen werden bei der Wegesuche umgangen. Dies kann im Extremfall dazu führen, daß kein Weg mehr gefunden wird. Im nächsten Bild ist ein Graph dargestellt, in dem der Benutzer zwei Türen geschlossen hat.

Hier hat der Benutzer die Türen zwischen Raum 3 und Raum 4 sowie zwischen Raum 4 und Raum 5 geschlossen. Möchte das System nun einen Weg von Raum 4 zu Raum 3 planen, so würde es einen Weg über Raum 1 und Raum 2 erhalten.

Alle bisher beschriebenen Fenster sind druckbar, d.h. sie können über das Menü File-Print auf einen Drucker ausgegeben werden.

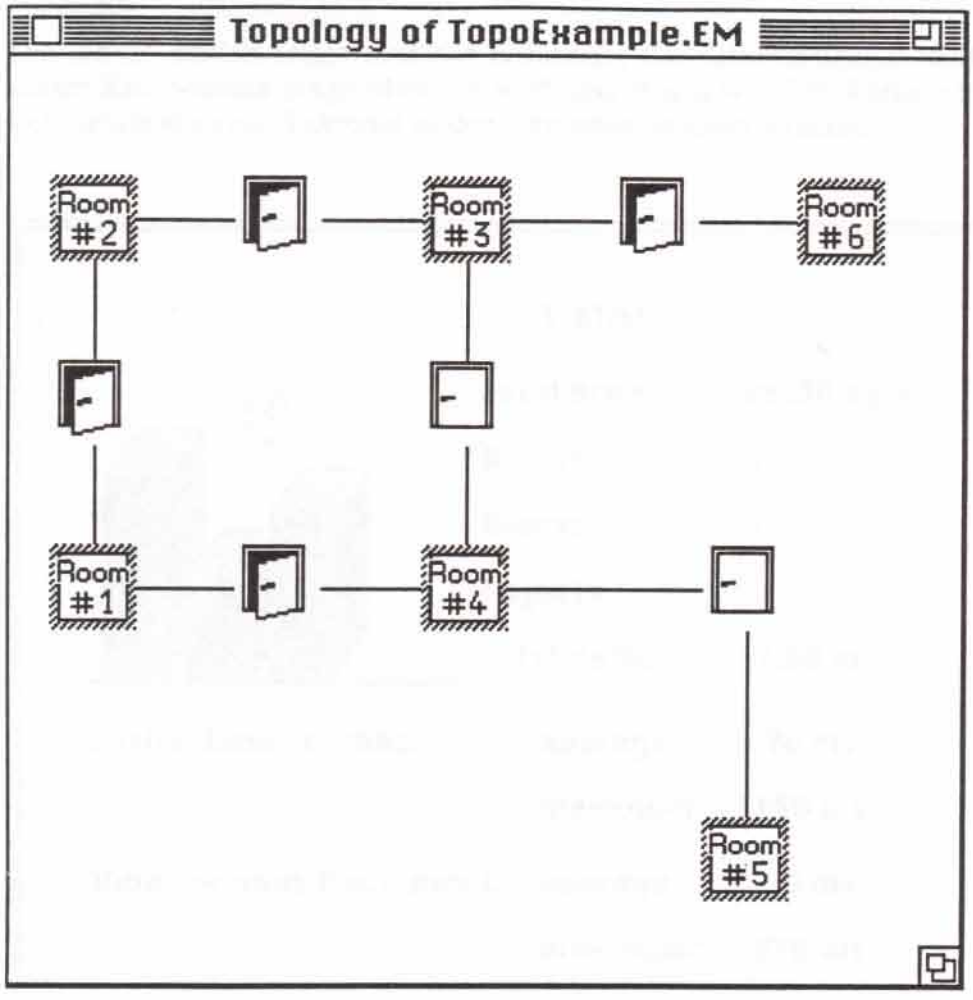


Bild 5.3.4: Ein Graph mit geschlossenen Türen

5.3.2 Das Statistik-Fenster

Dieses Fenster (eigentlich ein Dialog) wird über das Menü Show-Statistic geöffnet. Obwohl es sich hierbei um einen Modaldialog handelt, also normalerweise kein Menü anwählbar ist, kann der Inhalt über das Print-Menü ausgedruckt werden.

Im nächsten Bild ist das Statistikfenster gezeigt.

Es wird der Flächeninhalt des gesamten explorierten Gebietes ausgerechnet und angegeben. Als weitere Informationen werden Anzahl der Türen, Anzahl der Räume sowie Gesamtzahl der Objekte in Räumen ausgegeben.

Die gesamte Pfadlänge bezieht sich auf die gefahren Strecke in der programmeigenen Simulationsumgebung. Da in der Realität unter Umständen erheblich mehr Weg zurückgelegt werden muß, um bestimmte Randbedingungen zu erfüllen, ist dieser Wert nur ein sehr grober Anhalt.

Die nächsten Daten sind die mittleren und die maximalen Ausführungszeiten der Kartenfusionierung und der Wegplanung. Diese Daten helfen den Zeitbedarf grob abzu-

schätzen.

Die letzten beiden Daten sind zur Ermittlung des Kommunikationsaufkommens mit dieser Komponente vorgesehen. Es wird angegeben, wieviele Karten eingelesen wurden, sowie wieviele Vektoren an den Navigator geliefert wurden.

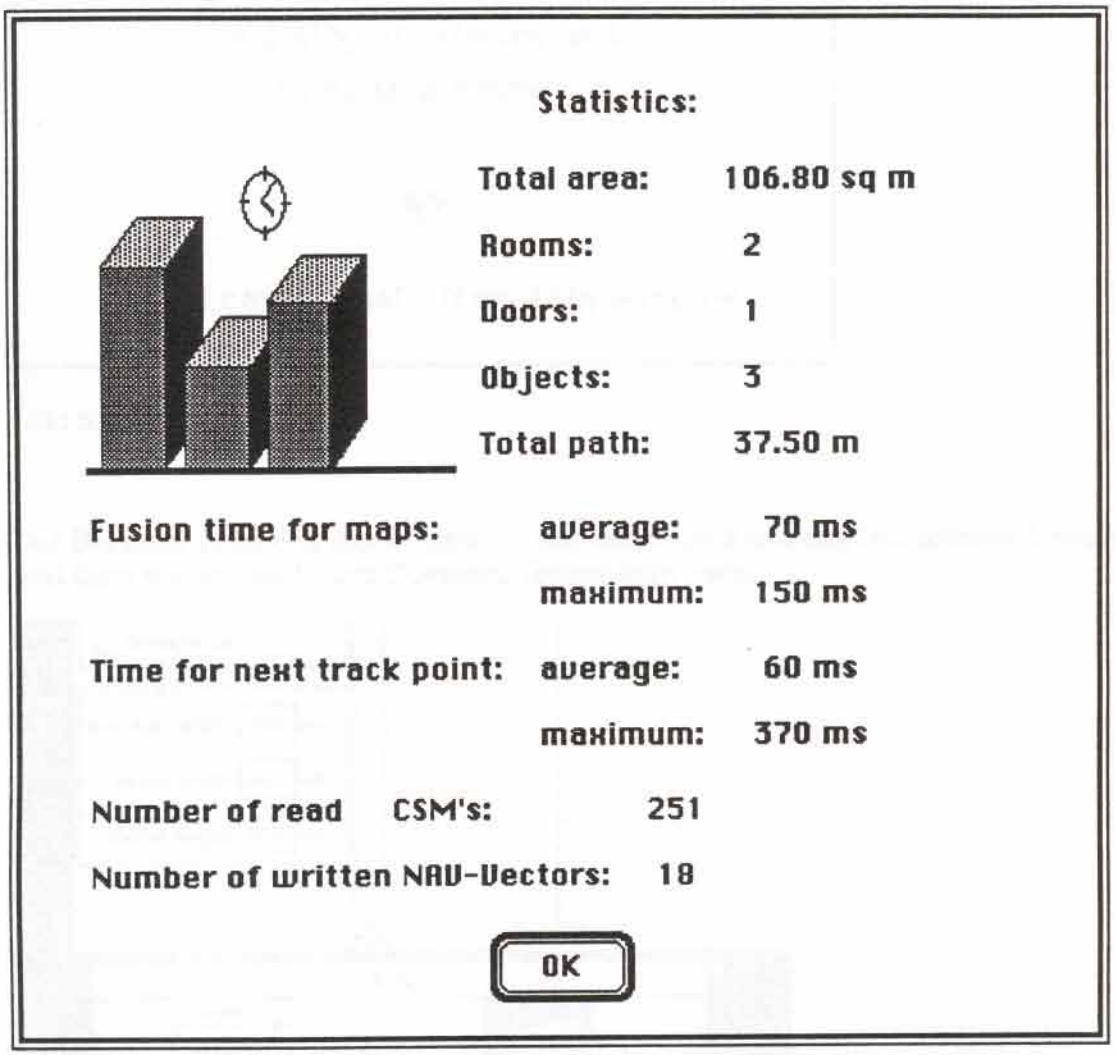


Bild 5.3.5: Ein Beispiel für das Statistik-Fenster

5.3.3 Das Hilfe-Fenster

Mit Menü Show-Help kann der Benutzer die Online-Hilfe aktivieren. Es erscheint folgendes Fenster:

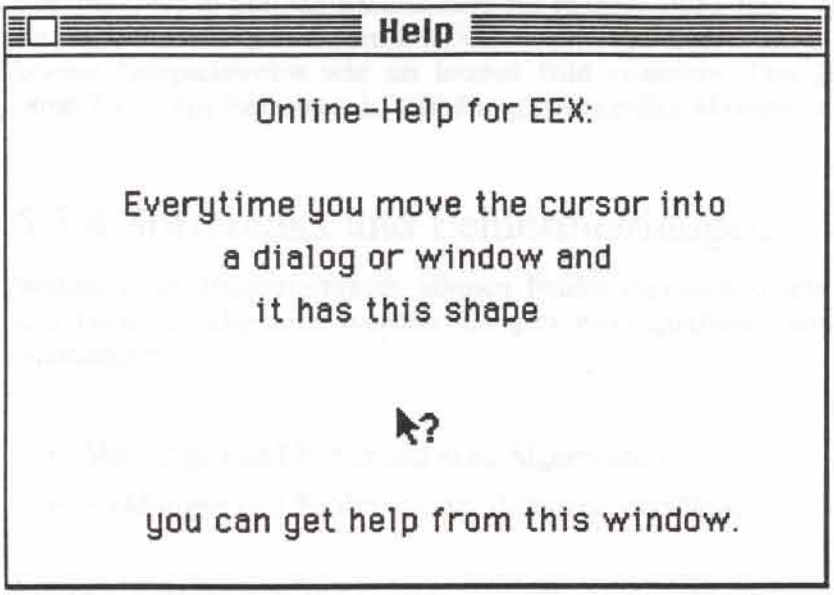


Bild 5.3.6: Das Hilfe-Fenster

Der Benutzer kann nun das Fenster an eine beliebige Stelle des Bildschirms bewegen, und dann wieder das Haupt-Dokumentfenster aktivieren.

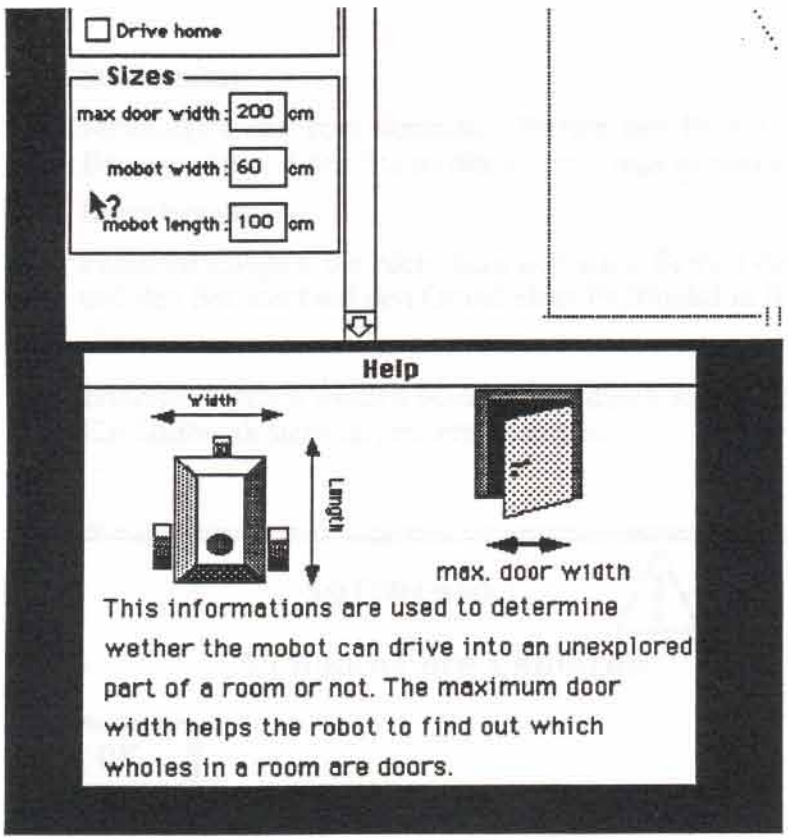


Bild 5.3.7: Beispiel einer Online-Hilfe

Jedesmal, wenn nun am Mauscursor ein Fragezeichen steht, kann man dem Hilfe-Fenster eine Erklärung von dem Bereich, in dem der Cursor gerade steht, entnehmen. Dies könnte beispielsweise wie im letzten Bild aussehen. Das Hilfe-Fenster ändert sich ohne Zutun des Benutzers in Abhängigkeit von der Mausposition.

5.3.4 Softbreaks und Fehlermeldungen

Während des Programmlaufs können Fehler eintreten sowie sonstige Meldungen an den Benutzer abgesetzt werden. Es gibt zwei qualitativ unterschiedliche Arten von Meldungen:

- Meldungen und Fehler aus dem Algorithmus
- Meldungen und Fehler aus der Benutzeroberfläche

Die Unterscheidung wird deshalb getroffen, da auf dem Zielsystem nur noch der Algorithmus vorliegt und keine Benutzeroberfläche mehr. Aus diesem Grund wurde speziell für die Algorithmen eine Fehler- und Reportschnittstelle entworfen, sogenannte Errorbreaks und Softbreaks. Diese werden aus dem Algorithmus heraus über spezielle Funktionen aufgerufen, die im Falle des simulierten Systems Dialogboxen öffnen, im Falle des Zielsystems Meldungen an eine Schnittstelle nach außen absetzen. Die zwei Arten von Meldungen bedeuten dabei folgendes:

- **Softbreaks:**

  Meldungen, die zum normalen Betrieb des Programms gehören, und den Benutzer nur auf den Status des Algorithmus hinweisen sollen.

- **Errorbreaks:**

  Fehlermeldungen, die nicht zum normalen Betrieb des Programms gehören, und den Benutzer auf den Grund einer Fehlfunktion führen sollen.

Im simulierten System werden beide Arten, durch verschiedene Dialogboxen repräsentiert. Ein Softbreak sieht folgendermaßen aus:

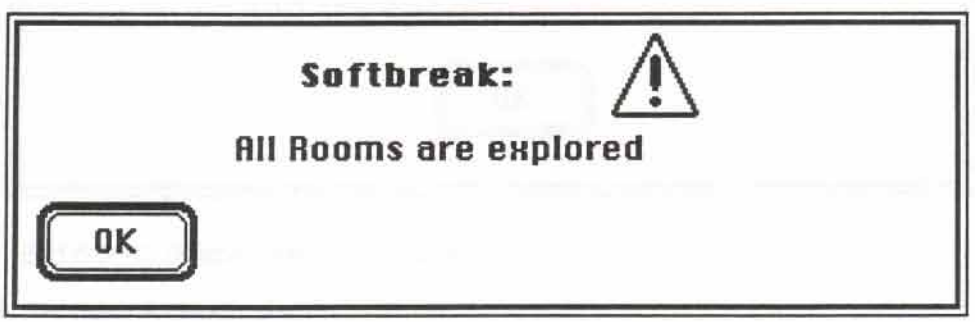


Bild 5.3.8: Beispiel einer Softbreak-Meldung

Eine Errorbreak-Dialogbox hat ein ähnliches Aussehen:

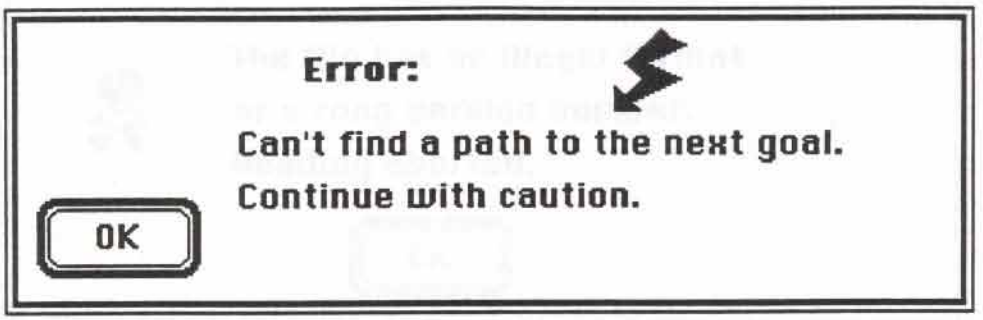


Bild 5.3.9: Beispiel einer Errorbreak-Meldung

Eine ähnliche Unterscheidung findet man auch bei den Meldungen aus der Benutzeroberfläche:

- **Warnungen:**

  Meldungen, die Benutzer auf einen Mißstand hinweisen sollen, der leicht durch entsprechende Maßnahmen behoben werden kann.

- **Fehler:**

  Meldungen, die den Benutzer auf Fehler hinweisen sollen, die nicht durch Maßnahmen im Programm behoben werden können.

Eine Warnung hat folgendes Aussehen:

Bild 5.3.10: Beispiel einer Warnung

Eine Fehlermeldung sieht so aus:

Bild 5.3.11: Beispiel einer Fehlermeldung

Im folgenden sollen alle Meldungen aus dem Algorithmus und aus der Benutzeroberfläche aufgelistet werden. Falls ein Fehler vorliegt, der behoben werden kann, wird eine Empfehlung gegeben, wie man in einem solchen Fall reagieren sollte.

Warnungen:

The robot has an illegal position:

Beim Bestimmen der Position des Roboters wurde diese zu nahe an eine Linie gelegt. In diesem Fall muß einfach eine neue Position bestimmt werden, die einen ausreichenden Sicherheitsabstand bietet.

This room is already explored:

Die Position des Roboters wurde in schon exploriertes Gebiet gesetzt. Entweder wird die Position in einen neuen Raum gesetzt, oder man wählt das Menü State-Reset, um einem neuen Lauf durchzuführen.

No outline map to save:

Man hat den entsprechende Knopf gedrückt, um eine Karte abzuspeichern, ohne daß eine vorliegt. In diesem Fall muß solange exploriert werden, bis ein Raum komplett erfaßt wurde.

Fehler:

The file has an illegal format:

Es wurde versucht eine Datei einzulesen, die entweder durch einen Plattenfehler verfälscht wurde oder eine falsche Versionsnummer besitzt. Da sich die Dateien mit fortschreitender Programmversion auch ändern, kann so verhindert werden, daß eine unpassende Struktur in den Speicher geladen wird, und so das Verhalten des Programms nicht mehr zu kontrollieren ist.

Clipboard error:

Der Benutzer hat 'Map in Clipboard' gedrückt, und es ist ein Fehler aufgetreten. In diesem Fall liegt das Problem im System begründet, und kann nicht durch den Benutzer behoben werden.

Softbreaks:

All rooms are explored:

Der Algorithmus hat die globale Exploration beendet und ist somit fertig. Dies entspricht dem normalen Ende eines kompletten Programmlaufs.

Shutdown command received:

Der Benutzer hat den Stop-Knopf betätigt. Im realen System entspricht dies dem Shutdown-Kommando des Supervisors. Die Meldung deutet somit auf ein vorzeitiges Ende des Programms hin.

Errorbreaks:

Memory overflow:

Diese Meldung dürfte bei einem durchschnittlich ausgebauten Rechner sehr selten auftreten. Bei jeder Speicherallokierung wird vorher ein Test durchgeführt, ob noch ausreichend viel Speicherplatz vorliegt. Da der Algorithmus selbst nur den Platz für eigene Variable anlegt, ist ein Speicherüberlauf sehr unwahrscheinlich.

Dispose error:

Es wurde versucht ein Speicherbereich freizugeben, der vorher nicht allokiert wurde. Diese Fehlermeldung wurde nur für die Testphase implementiert, und dürfte deshalb im normalen Betrieb nicht auftreten.

Can't find path to the next goal:

Die globale Exploration schlägt einen neuen Anlaufpunkt vor, doch dieser kann nicht erreicht werden. Ein Grund könnte sein, daß der Benutzer im topologischen Graph zu viele Türen geschlossen hat.

Can't drive path to next room:

Der Navigator weigert sich eine Strecke im globalen Pfad in einem Raum von Tür zu Tür abzufahren. Da die globale Exploration in einem solchen Fall keine Ausweichpunkte zur Verfügung hat, tritt somit ein vorzeitiges Ende ein. In der simulierten Fassung kann dieser Fehler nicht auftreten, da in der globalen Planung alle Wege in direkter Linien befahren werden, ohne daß auf Hindernisse geachtet wird. Im Zielsystem könnte dieser Fehler auftreten, wenn ein Raum nach einer lokalen Exploration verändert wurde.

Too many doors detected:

Die Liste der Türen zu einem Raum ist eine der wenigen Strukuren, die als Feld abgelegt wurde. Aus diesem Grund ist es möglich, wenn auch sehr

unwahrscheinlich, daß dieses Feld überläuft. In einem solchen Fall wird die Exploration weitergeführt, als würden die zusätzlichen Türen nicht existieren.

Unknown command:

Auf dem Zielsystem bekommt der Algorithmus vom Supervisor Kommandos, die er zu interpretieren hat. Wird ein Kommando eingelesen, das diese Komponente nicht versteht, so wird dieser Fehler ausgegeben. Im simulierten System tritt dieser Fehler nicht auf.

6. Implementierung

In diesem Kapitel sollen die Grundlagen des implementierten Systems, auch im Hinblick auf die Programmumgebung dargelegt werden. Hierbei werden auch, in Ergänzung zu dem Programmlisting, die Module beschrieben.

6.1 Randbedingungen

6.1.1 Verschiedene Ablaufmodi

Die Komponente "Explorator" kann in drei verschiedenen Konfigurationen gestartet werden:

- **Stand-alone-Betrieb:**

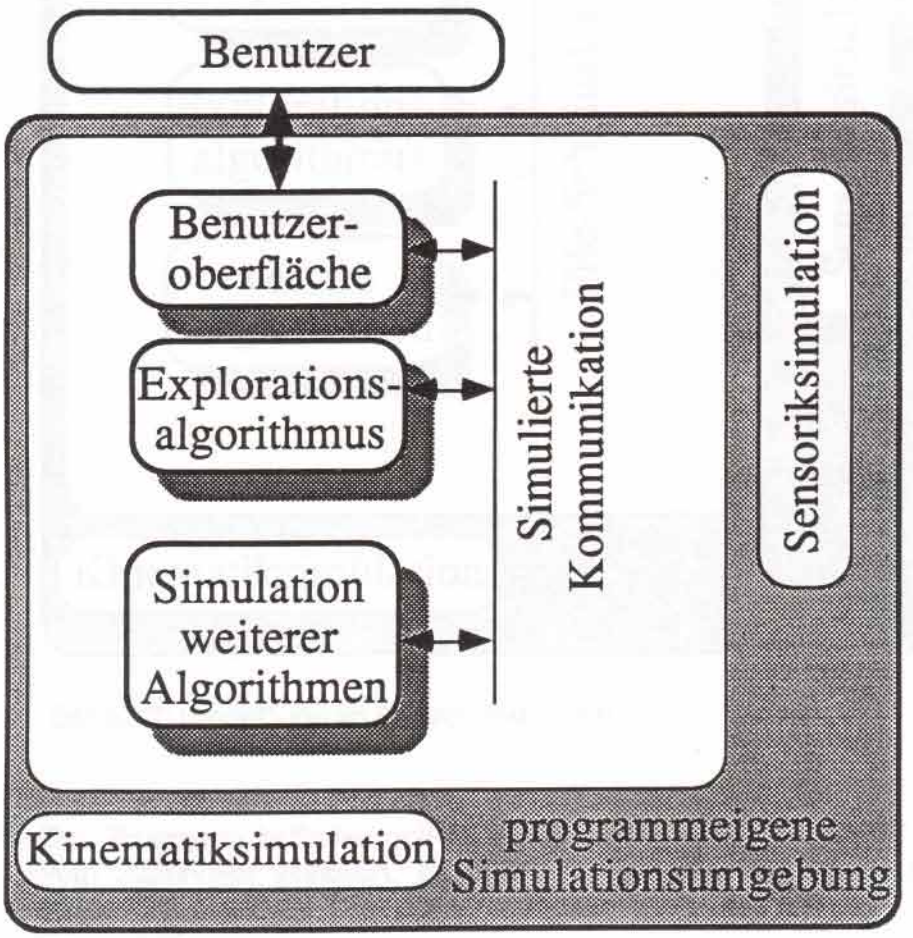


Bild 6.1.1: Umgebung im Stand-alone-Betrieb

Das Programm läuft mit eigener Simulationsumgebung und wird vom Benutzer kontrolliert. So können vor allem neue Algorithmen schnell getestet werden.

Man kann erkennen, daß hier nicht nur die Sensorik und die Kinematik simuliert werden, sondern auch die Komponenten, die den Explorator umgeben. Die steuernden Kommandos kommen über eine Oberfläche vom Benutzer. Diese werden über eine Kommandoschnittstelle dem Algorithmus zugänglich gemacht. Hierbei wird das Kommunikationsmedium selbst auch simuliert.

- **Simulationumgebung:**

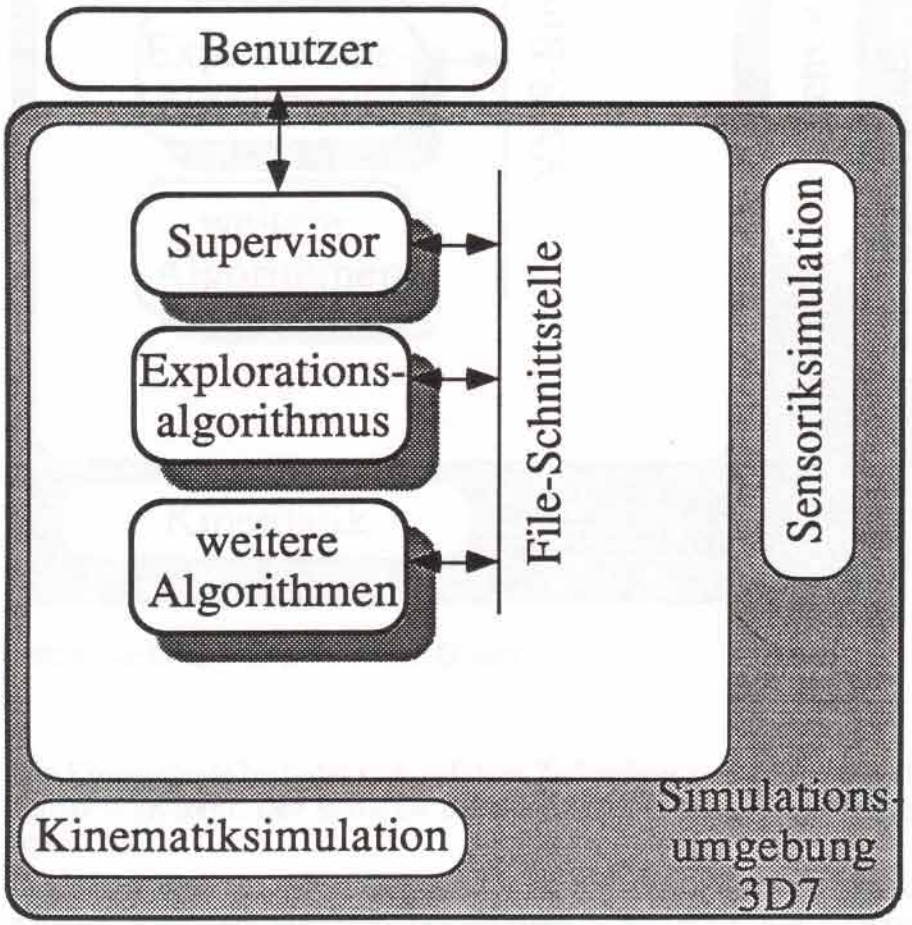


Bild 6.1.2: Umgebung im Simulationsbetrieb

Das Programm befindet sich in der Simulationsumgebung 3D7 und wird vom Supervisor gesteuert. Es findet eine rudimentäre Benutzersteuerung statt.

Die umgebenden Komponenten, werden nun nicht mehr simuliert, sondern werden vom Supervisor aktiviert. Lediglich die Sensorik und die Kinematik müssen noch simuliert werden. Die Kommunikation findet über eine File-Schnittstelle statt.

- **Einsatzumgebung:**

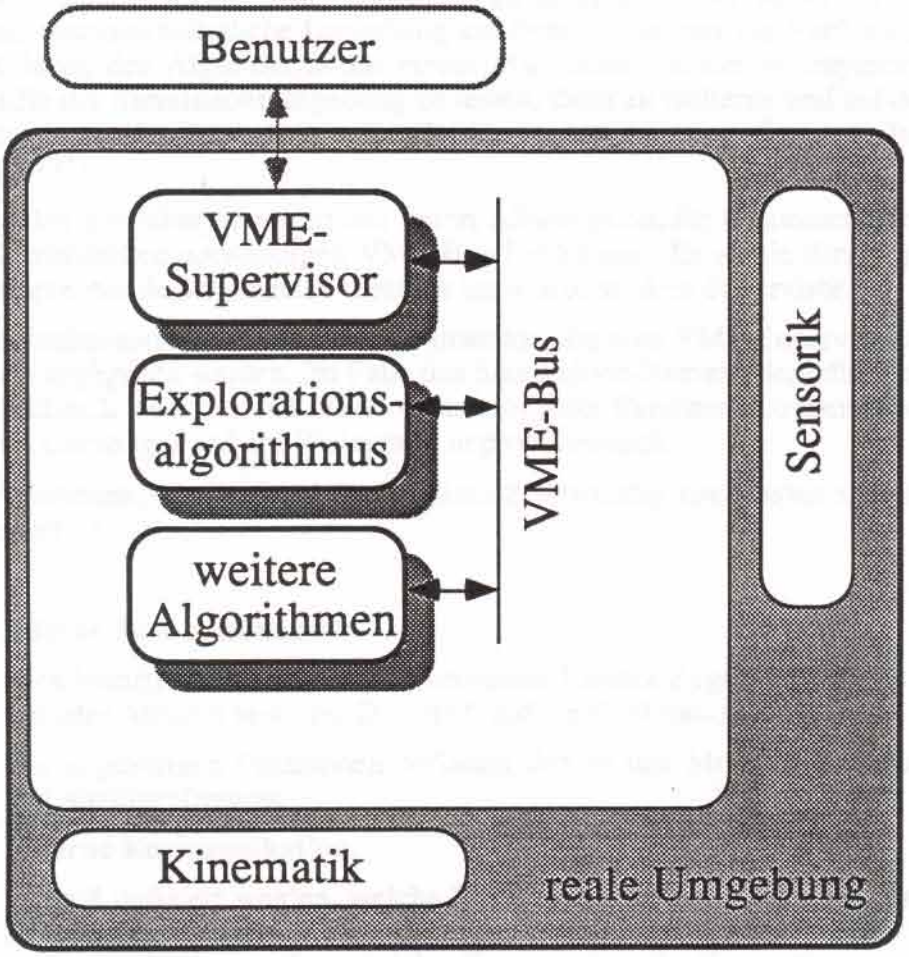


Bild 6.1.3: Umgebung im realen Einsatz

Die Komponente befindet sich auf dem Zielrechner und wird mit realen Sensordaten versorgt. Der Benutzer hat nur noch minimale Eingreifmöglichkeiten.

Hier liegen alle Komponenten, sowie die Umgebung real vor. Die Kommunikation wird über ein VME-Bus-System abgewickelt. Ein Supervisor hat auch hier wieder die Kontrolle, nur übernimmt dieser hier andere Aufgaben als im simulierten System.

Wie man sieht, unterscheiden sich die Ablaufmodi durch den Grad der Benutzerkontrolle sowie durch die Sensor- und Aktionsumgebung. Während der Algorithmus fester Bestandteil aller drei Konfigurationen ist, müssen die entsprechenden Kommunikations- und Simulationsmodule ausgetauscht werden. Eine feste Schnittstelle zwischen Algorithmus und Kommunikationsmedium garantiert, daß ein Austausch umproblematisch von statten geht.

6.1.2 Das Albatross-System

Bei dem Albatross-System handelt es sich um einen Betriebssystemkern, der eine weitgehend Macintosh-ähnliche Umgebung auf dem Zielrechner zur Verfügung stellt. Ziel ist es dabei, den Algorithmus auf einem Macintosh-Rechner zu implementieren und mit Hilfe der Simulationsumgebung zu testen, dann zu isolieren und auf den Zielrechner zu portieren. Eine genaue Beschreibung des Albatross-Systems findet man in [Wetzler91].

Neben den Betriebssystemroutinen liefert Albatross die für Kommunikation mit anderen Komponenten notwendigen VME-Bus-Funktionen. Es wurde darauf geachtet, daß die entsprechende Schnittstelle dieselbe ist, wie unter dem Supervisor.

Die Ablaufsteuerung erfolgt über Kommandos, die vom VME-Supervisor an die Algorithmen abgegeben werden. Im Falle des Stand-alone-Betriebs liegt dieselbe Kommandoschnittstelle vor, nur daß die Kommandos vom Benutzer kommen. Somit ist auch hier ein Übertragen auf das Zielsystem unproblematisch.

Die Funktionen, die für den späteren Einsatz notwendig sind, lassen sich in vier Gruppen einteilen:

- **Interne Kommunikation:**

  Dies betrifft Komponenten, die im realen Einsatz Zugriff auf eigene Sensoren oder Aktoren besitzen. Dies trifft auf die Exploration nicht zu.

  Die zugehörigen Funktionen befinden sich in den Modulen *AuxiliaryPorts* und *AuxiliaryDrivers*.

- **Externe Kommunikation:**

  Es muß definiert werden, welche Datenstrukturen eingelesen oder geschrieben werden. Hierzu zählt auch die Kommunikation mit dem Supervisor. Es muß spezifiziert werden, welche Kommandos die Komponente ausführen kann.

  Die zugehörigen Funktionen befinden sich in den Modulen *ExternalPorts*, *ExternalTypes*, *GlobalCommands*, *RealTimePorts*, *MessagePorts*, *MissionPorts* und *TimeManager*.

- **Status- und Reportschnittstellen:**

  Es muß deklariert werden, welche Meldungen gegebenenfalls über eine Schnittstelle zu einem Benutzer gelangen sollen. Außerdem wird diese Möglichkeit auch benutzt, im simulierten System Tätigkeiten einzuleiten, die auf dem realen System nicht mehr ausgeführt können. Z.B. können Zeitmessungen durchgeführt werden, um Statistiken zu erstellen.

  Die zugehörigen Funktionen befinden sich in den Modulen *Monitor*, *BreakRoutines* und *Scheduler*.

- **Der Algorithmus selbst:**

  Dieser muß sowohl für dem simulierten, als auch für den realen Einsatz ausgelegt sein. Allgemein hat der Algorithmus folgendes aussehen:

```
BEGIN
    Initialisiere Algorithmus und Schnittstellen;
    WHILE nicht zuende DO
    BEGIN
        Lese nächstes Kommando ein;
        CASE Kommando OF
            Kommando_1:Führe Kommando_1 aus;
            ...
            Kommando_n:Führe Kommando_n aus;
        END
    END;
    Deinitialisiere Algorithmus und Schnittstellen
END;
```

Die zugehörigen Funktionen befinden sich in den Modulen *MainFrame*, *CommandLoop*, *Algorithm*, *Utilities*, *DataStructures*.

Eine Beschreibung der für die Exploration relevanten Module wird in einem späteren Kapitel vorgenommen.

6.1.3 MacApp

Bei MapApp handelt es sich um eine Bibliothek von Methoden, die zur Modellierung einer Benutzeroberfläche dienen [Apple89]. Die Bibliothek ist objektorientiert, so daß es möglich ist, Teile auszutauschen, ohne die restliche Funktionalität zu beeinflussen.

Obwohl die Vorteile einer objektorientierten Programmierung unbestritten sind, wurde der eigentliche Algorithmus "klassisch" ausgeführt. Dies hat vor allem zwei Gründe:

- Objekte werden in Pascal über sogenannte *Handles* abgelegt. Diese Struktur ermöglicht es, den Speicher neu zu organisieren (z.B. durch eine Garbage-Collection), ohne daß das Programm davon beeinflußt wird. Auf dem Albatross-System wurde diese Möglichkeit nicht verwirklicht, da ein solches Ereignis die Echtzeitfähigkeit in Frage stellt. Somit können Objekte auf dem Zielrechner nicht benutzt werden.
- Durch *Message-Passing* bei der Abarbeitung objektorientierter Programme entsteht ein unnötiger Overhead. Dieser Aufwand ist auf dem realen System nicht vertretbar.

Für die Benutzeroberfläche ist die objektorientierte Darstellung jedoch geeignet, da ein erhöhter Rechenaufwand nicht ins Gewicht fällt.

Die Objektbibliothek behandelt alle Belange der Benutzeroberfläche, diese sind:

- **Menüs:**

  Zur Behandlung des File-Menüs, sowie des Edit-Menüs werden Default-Methoden bereitgestellt, so daß bei der Programmierung hier kein Aufwand entsteht. Bei eigenen Menüpunkten wird eine Methode aktiviert, die sich nach dem aktuell offenen Dokument richtet.

- **Fenster, Dialoge:**

  Diese werden unter MacApp unter dem Begriff *Views* zusammengefaßt. Mit einem komfortablen Editor (ViewEdit) könne Views erzeugt und geändert werden. Im Programm müssen dann nur noch nicht standardmäßig auftretende Aktionen definiert werden. Solche wie z.B. Editieren einer Zahl, Vergrößern eines Fensters oder Drucken eines Fensterinhalts werden von MacApp autonom durchgeführt, ohne daß dies im Programm explizit deklariert werden muß.

- **Events:**

  MacApp stellt eine Haupteventloop zur Verfügung, die in der Regel nicht überschrieben werden muß. Nur Ereignisse, die das Programm behandelt werden herausgeführt, und aktivieren entsprechende Methoden.

Das Programmieren unter MacApp beschränkt sich somit auf das Erzeugen von zusätzlichen Objektklassen, und das Überschreiben einiger Methoden.

Im nächsten Bild ist der Hierarchiebaum der Objektklassen dargestellt. Dabei sind die eigenen Klassen hervorgehoben dargestellt. Diese sollen im folgenden beschrieben werden:

- **TEEXApplication:**

  Diese Klasse stellt vor allem Methoden zur Verfügung, die die Applikationsspezifischen Menüpunkte abhandeln sollen. Außerdem wird dort festgelegt, wie ein Dokument geöffnet werden soll.

- **TEEXDocument:**

  Hier befindet sich die Beschreibung eines Programmdokuments. Darin enthalten sind Referenzen auf die verwendeten View, sowie die File-Operationen, die dazu dienen, ein Dokument von einem Plattenmedium zu laden.

- **TEdgeMap:**

  Die Environment Edge Map aus dem Mapeditor wird in einer Unterstruktur des Dokuments abgelegt. Diese befindet sich in dieser Klasse.

- **TParameterDocument, TStateDocument, TOutlineDocument:**

  Da jede dieser Dokumente eigene File-Operationen benötigt, sind die zugehörigen Datenstrukturen samt Zugriffsroutinen in eigenen Klassen abgelegt worden.

- **TMapWindowView, TTopoWindowView, THelpWindowView:**

  Hier befinden sich solche Viewdefinitionen, die zu Fenstern gehören. Im

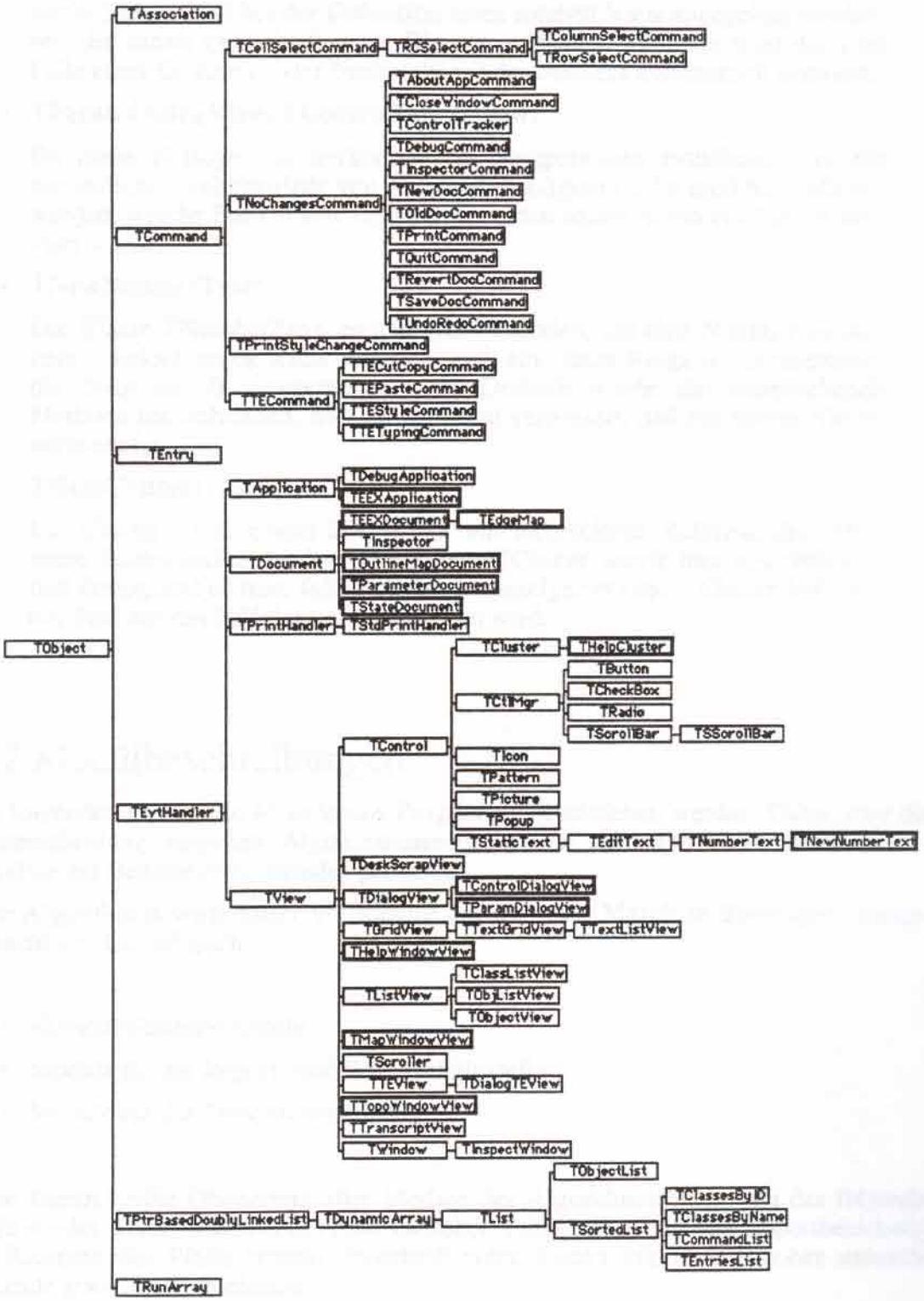


Bild 6.1.4: Objektklassen der Explorationskomponente unter MacApp

wesentlichen muß bei der Definition eines solchen View angegeben werden, wie der Inhalt gezeichnet wird. Die entsprechende Methode wird dann im Falle eines Eröffnens oder Neuzeichnens des Fensters automatisch aktiviert.

- **TParamDialogView, TControlDialogView:**

  Da diese Dialoge nur herkömmliche Komponenten enthalten, wird die wesentliche Funktionalität von MacApp bereitgestellt. Es muß nur definiert werden, welche Funktionen ausgeführt werden sollen, wenn ein Button aktiviert wurde.

- **TNewNumberText:**

  Die Klasse TNumberText, enthält alle Methoden, um eine Nummer zu editieren. Jedoch ist es leider möglich auch eine leere Eingabe vorzunehmen, die dann zu "0" ausgewertet wird. Deshalb wurde die entsprechende Methode überschrieben, und dahingehend verbessert, daß ein solche Aktion verboten ist.

- **THelpCluster:**

  Ein Cluster ist in einem Dialog nur ein rechteckiger Rahmen, der selbst keine Funktionalität besitzt. Die Klasse TCluster wurde hier neu definiert, und ermöglicht es nun, falls sich der Mauszeiger in einem Cluster befindet, ein Text auf das Hilfefenster ausgegeben wird.

6.2 Modulbeschreibungen

Im folgenden sollen alle Module des Programms beschrieben werden. Dabei wird die Unterscheidung zwischen Algorithmusmodulen und sonstigen Modulen (also z.B. Module der Benutzerschnittstelle) getroffen.

Der Algorithmus wird später vollständig auf die reale Maschine übertragen. Ausgetauscht werden lediglich

- Kommunikationsmodule
- Module für die Report- und Fehlerschnittstelle
- Module für die Speicherverwaltung

Eine hierarchische Gliederung aller Module des Algorithmusblock gibt das folgende Bild wieder. Dabei bedeutet ein Pfeil zwischen Teilblöcken, daß eine Importbeziehung in Richtung des Pfeils besteht. Innerhalb eines Blocks importieren höher stehende Module jeweils tieferstehende.

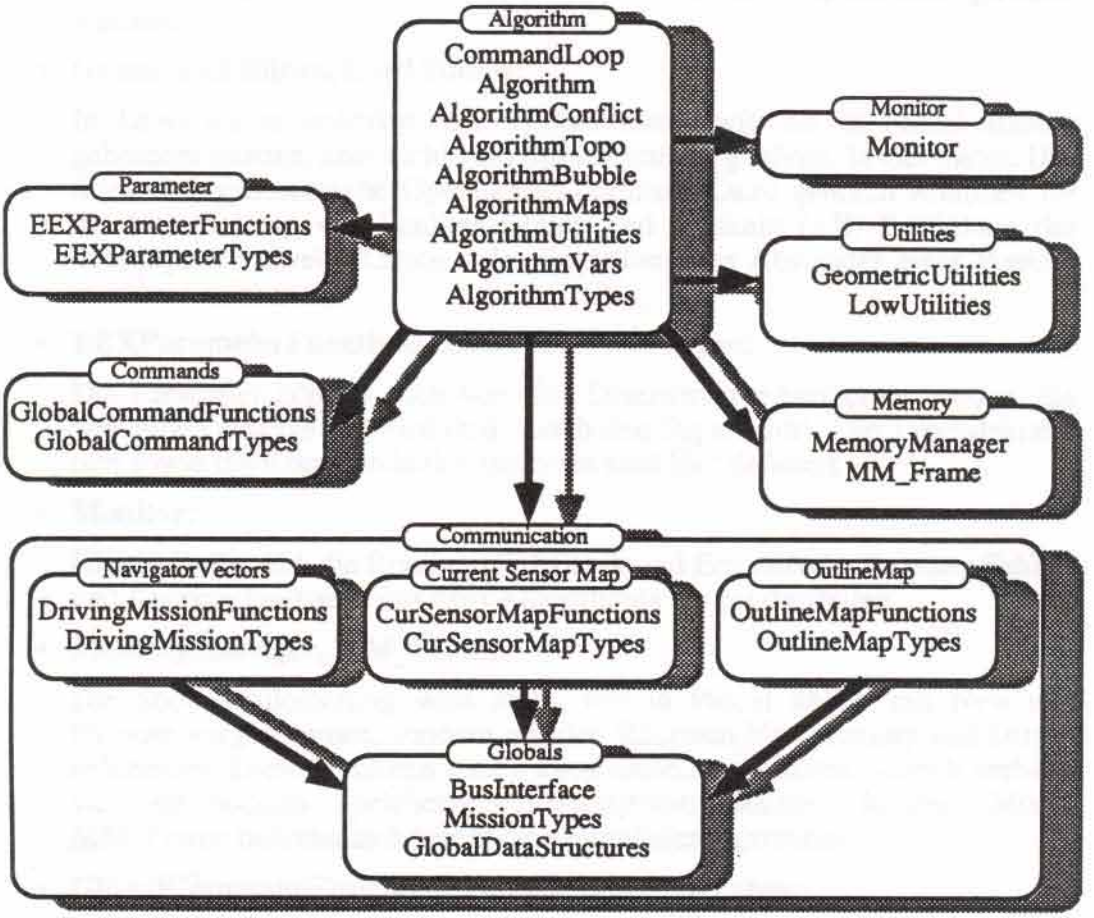


Bild 6.2.1: Module des Algorithmusblocks und deren Importbeziehungen

Die Aufgaben sind folgendermaßen auf die Module verteilt:

- **CommandLoop, Algorithm:**

  Neben der Initialisierung und Deinitialisierung des Algorithmus steht hier eine Prozedur zur Verfügung, die einen einzigen Explorationsschritt durchführt. Dieser umfaßt das Einlesen einer neuen Karte und gegebenenfalls Abgabe eines neuen Fahrbefehls.

- **AlgorithmConflict, AlgorithmTopo, AlgorithmBubble:**

  Alle höheren Funktionen des Algorithmus befinden sich in diesen drei Modulen. Dazu gehören lokale und globale Kartographierung sowie Konfliktlösungsstrategien.

- **AlgorithmMaps, AlgorithmUtilities:**

  Für jede interne Listen- oder Kartenstruktur wurden Installations-, Deinstallations- und Zugriffsroutinen implementiert. Diese befinden sich in diesen beiden Modulen.

- **AlgorithmVars, AlgorithmTypes:**

  Alle für den Algorithmus relevanten Typen werden in AlgorithmTypes defi-

niert. In AlgorithmVars befinden sich die für den Algorithmus globalen Variablen.

- **GeometricUtilities, LowUtilities:**

  In LowUtilities befinden sich Hilfsroutinen, wie sie in Pascal ständig gebraucht werden, aber nicht zum Sprachumfang gehören. In Geometric Utilities sind geometrische Operationen definiert. Dazu gehören Routinen für die Verarbeitung von Punkten Linien und Winkeln (z.B. Ermittlung des Schnittpunkts zweier Linien, oder Ermittlung des Abstandes eines Punktes zu einer Linie).

- **EEXParameterFunctions, EEXParameterTypes:**

  Die Parameter können auch wie eine Datenstruktur betrachtet werden, die von außen eingegeben wird (z.B. durch den Supervisor). Die Typendeklaration sowie die Kommunikationsroutinen sind hier definiert.

- **Monitor:**

  Hier befinden sich die Routinen SoftBreak und ErrorBreak, die einen Fehler- und Reportschnittstelle aus dem Algorithmus heraus darstellen.

- **MemoryManager, MM_Frame:**

  Die Speicherallokierung wird nicht, wie in Pascal üblich mit New und Dispose vorgenommen, sondern mit den Routinen NewMemory und DisposeMemory. Diese Routinen haben zwar dieselbe Semantik, jedoch verbirgt sich ein anderes Speicherverwaltungssystem dahinter. In dem Modul MM_Frame befindet sich hierzu eine Initialisierungsroutine.

- **GlobalCommandFunctions, GlobalCommandTypes:**

  Die Kommandos zur Steuerung gelangen vom Supervisor über das Kommunikationsmedium zum Algorithmus. In diesem Modulen werden alle möglichen Kommandos sowie eine Routine zum Einlesen eines Kommandos definiert.

- **DrivingMissionFunctions, DrivingMissionTypes, CurSensorMapFunctions, CurSensorMapTypes, OutlineMapFunctions, OutlineMapTypes:**

  Alle Deklarationen aller Strukturen, die mit andern Applikationen ausgetauscht werden sowie die notwendigen Zugriffsroutinen befinden sich in diesen Modulen.

- **BusInterface, MissionTypes, GlobalDataStructures:**

  Hier befinden sich allgemeine Deklarationen z.B. grundlegende Einheitentypen, sowie Routinen der Busverwaltung.

Die Importbeziehungen der Module im Stand-alone-Betrieb sind im nächsten Bild dargestellt.

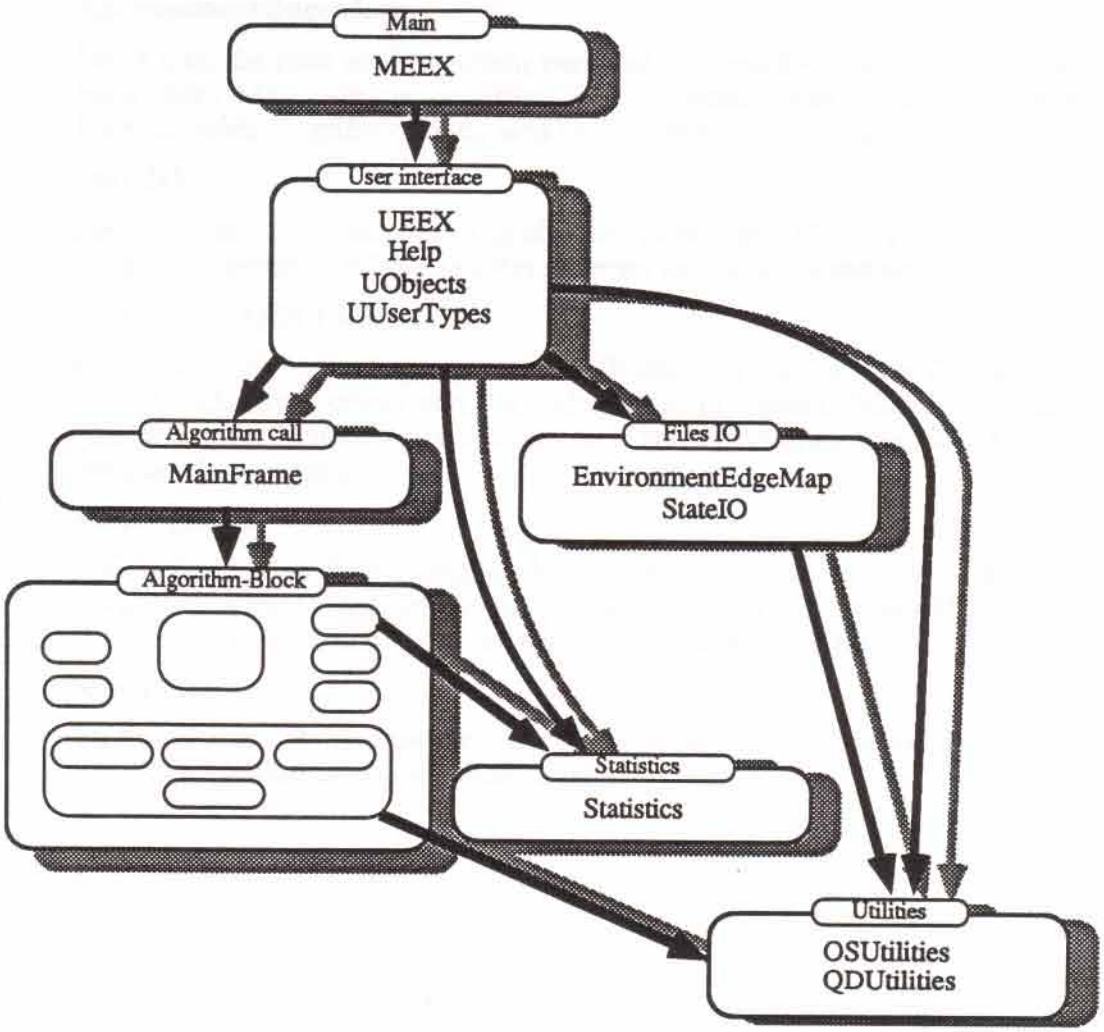


Bild 6.2.2: Module des gesamten Programms und deren Importbeziehungen

- **MEEX:**

  Das Hauptprogramm: hier wird der Speicher initialisiert und die erste Methode des Programms aufgerufen. Von dort an verläuft das Programm objektorientiert.

- **UEEX:**

  Alle für die Benutzeroberfläche relevanten Objektklassen sind hier implementiert.

- **Help:**

  Hier befinden sich die Routinen zur Online-Hilfe.

- **UUserTypes, UObjects:**

  Alle Typen, die zur Steuerung der Benutzeroberfläche notwendig sind, werden in diesen beiden Modulen deklariert. Während in UUserTypes die grundlegenden Definitionen vorgenommen werden, liegen in UObjects die Objektklassen vor.

- **EnvironmentEdgeMap:**

  Die Karte, die man als Umgebungsstruktur einlesen kann, ist eine Environment Edge Map, wie sie im Editor erzeugt werden kann. Das zugehörige Format, sowie Zugriffsroutinen sind in diesem Modul definiert.

- **StateIO:**

  Der komplette Zustand des Algorithmus kann hier auf ein Plattenmedium gesichert werden, um ihn zu einem späteren Zeitpunkt zu restaurieren.

- **OSUtilities, QDUtilities:**

  Funktionen, die den Umgang mit dem Betriebssystem und den Grafikroutinen (Quickdraw) erleichtern, befinden sich in diesem beiden Modulen. Hierzu zählen Routinen der Zeitmessung und Fileoperationen, sowie komfortable Grafikroutinen.

- **MainFrame:**

  Hier befindet sich die Hauptschleife, die den Algorithmus bis zum erfolgreichen Ende immer wieder aufruft. Zusätzlich wird hier überprüft, ob der Benutzer den Algorithmus schon vorzeitig abbrechen möchte.

- **Statistics:**

  Im Stand-alone-Betrieb werden hier statistische Informationen erhoben, wie z.B. Kartenaufkommen oder Pfadlänge.

7. Testläufe und Auswertungen

In diesem Kapitel soll anhand von Testläufen die Güte des Verfahrens plausibel gemacht werden. Die Spannweite der Beispiele reicht dabei von trivialen Umgebungen bishin zu Problemsituationen. Exemplarisch soll die Einsatzfähigkeit der erzeugten Karte demonstriert werden, indem sie der Reinigungsplanungskomponente vorgelegt wird. Die Bilder entstammen alle dem implementierten System. Die Zeitmessungen wurden auf einem Macintosh II fx vorgenommen.

7.1 Einfache Beispiele

Die folgende Testumgebung besteht aus einem Raum, in dem sich ein Gegenstand befindet. Der Roboter startet in einer Ecke des Raumes.

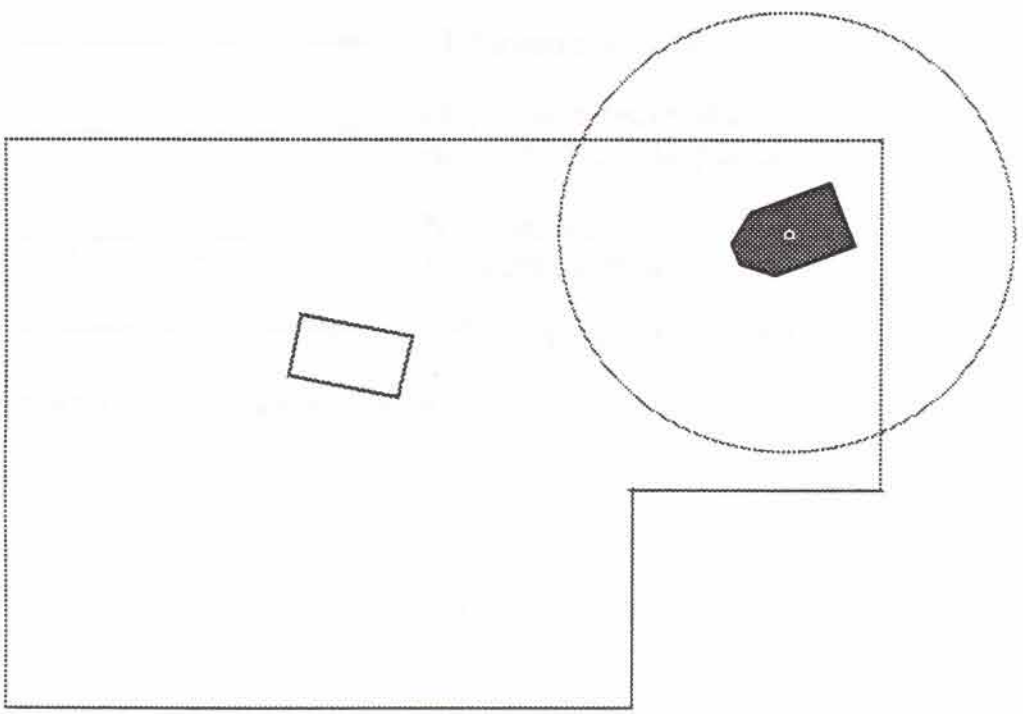


Bild 7.1.1: Umgebung des Testlaufs

Der Roboter hat noch keinen Scan eingelesen, d.h. er hat noch kein Wissen über die Umwelt. Durch den Kreis um den Roboter soll der Scannerradius angedeutet werden. Im folgenden wird der Roboter seinen ersten Scan einlesen und seine Karte aufbauen. Darauf basierend wird die Konfliktmenge erzeugt. Deren Lösung führt dann zu dem Punkt, der als nächstes angesteuert wird. Nach einer gewissen gefahrenen Strecke wird die Konfliktmenge neu erhoben und die Strecke neu berechnet. Die folgenden Bilder entstammen alle einem Testlauf des implementierten Systems. Sie wurden nur durch Pfeile angereichert, die die gefahrenen Strecken andeuten. Die Bilder haben einen

gewissen zeitlichen Abstand, so daß zwischenzeitlich die Konfliktmenge mehrmals erzeugt wurde. Aus Gründen der Übersichtlichkeit sind nur solche Konfliktpunkte erhoben worden, die zur Vervollständigung der Sensorbubble notwendig sind.

Eine Legende zu dem Lauf ist im nächsten Bild gezeigt.

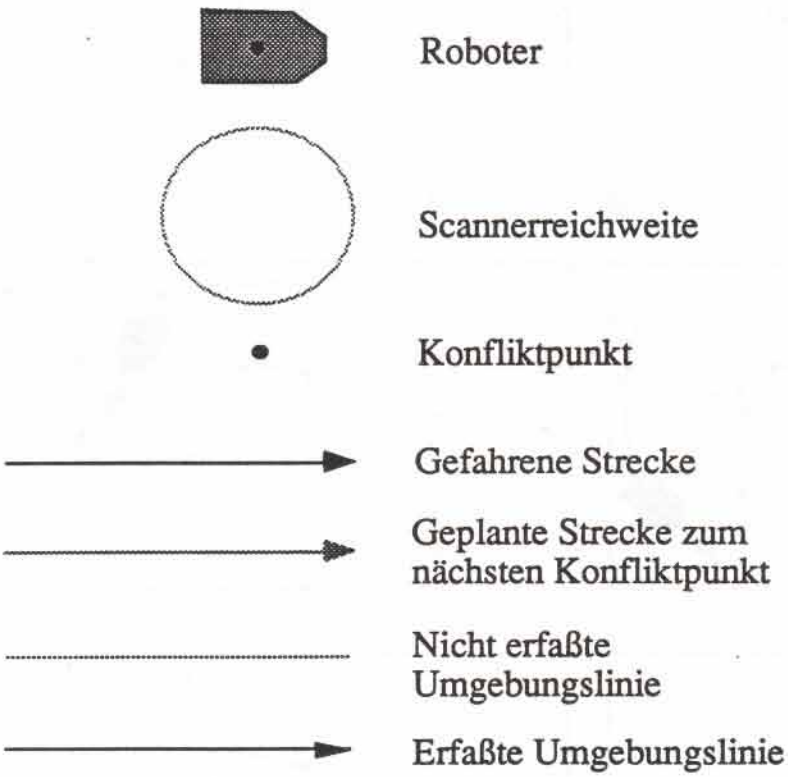


Bild 7.1.2: Legende zum Testlauf

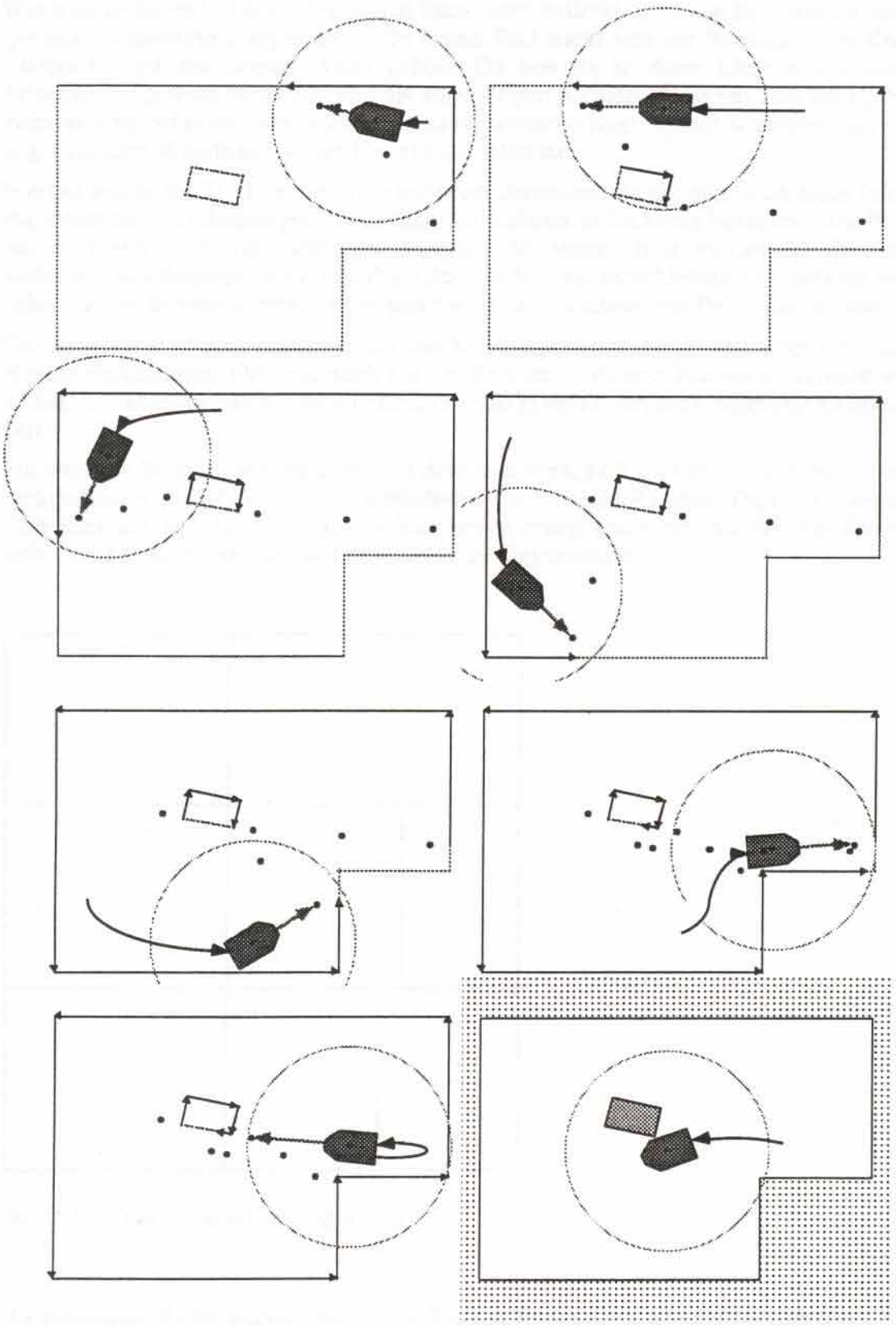


Bild 7.1.3: Ein Testlauf

Wie man in diesem Testlauf feststellen kann, wird indirekt durch die Befahrungsstrategie eine Wandverfolgung erreicht. Im ersten Bild sucht sich der Roboter einen Konfliktpunkt, der zur oberen Wand gehört. Da von da an diese Linie den größten Informationsgewinn verspricht und die zugehörigen nächsten Ziele mit dem geringsten Aufwand zu befahren sind, wird diese Linie weiterverfolgt. Später schließen sich die angrenzenden Wände an, bis der Umlauf komplett ist.

Hierbei wurde der Tisch in der Mitte teilweise abgetastet. Es gilt nun, auch diese Informationen zu vervollständigen. Dazu fährt der Roboter in Richtung Mitte und schafft es so, auch den Tisch vollständig zu erkunden. Im letzten Bild wurden die Bereiche außerhalb des Raumes sowie innerhalb des Tisches mit verschiedenen Grautönen versehen, um zu demonstrieren, daß es sich hierbei um geschlossene Polygone handelt.

Gut zu sehen sind auch immer Tripel von Konfliktpunkten, die jeweils einer virtuellen Kanten entstammen. Offensichtlich hat der Roboter in diesem Fall seine Aufgabe gut gelöst. Er hat sogar einen relativ optimalen Pfad gewählt, um die Umgebung zu erkunden.

Ein weiteres Beispiel soll die Fähigkeit demonstrieren, sich auch in vielen Räumen zurechtzufinden. Gegeben ist eine Gebäudestruktur mit acht Räumen. Da es bei diesem Test jetzt nur auf die topologische Kartographierung ankommt, wurden die Räume selbst sehr einfach gehalten und besitzen keine Gegenstände.

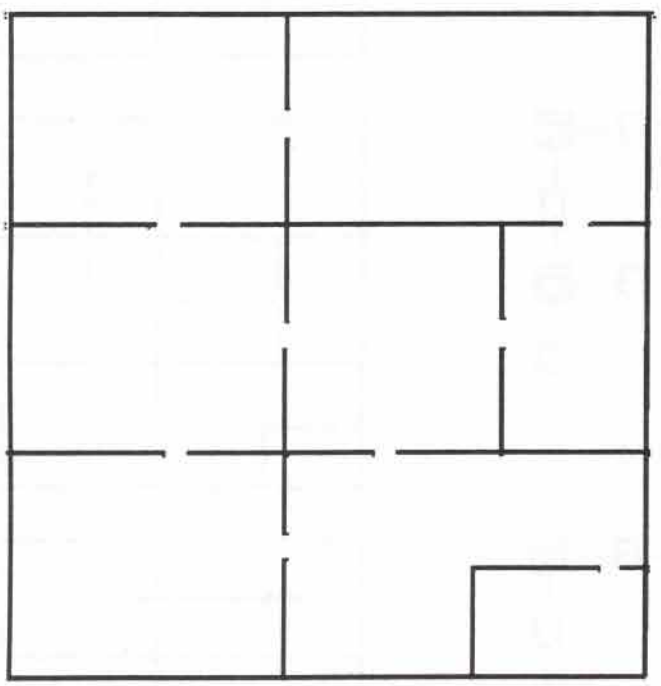


Bild 7.1.4: Eine weitere Testumgebung

Die folgenden Bilder wurden nach jeder lokalen Exploration von der bis dahin gespeicherten topologischen Karte gemacht. Sie entstammen auch dem implementierten System. Der Roboter startet im mittleren Raum.

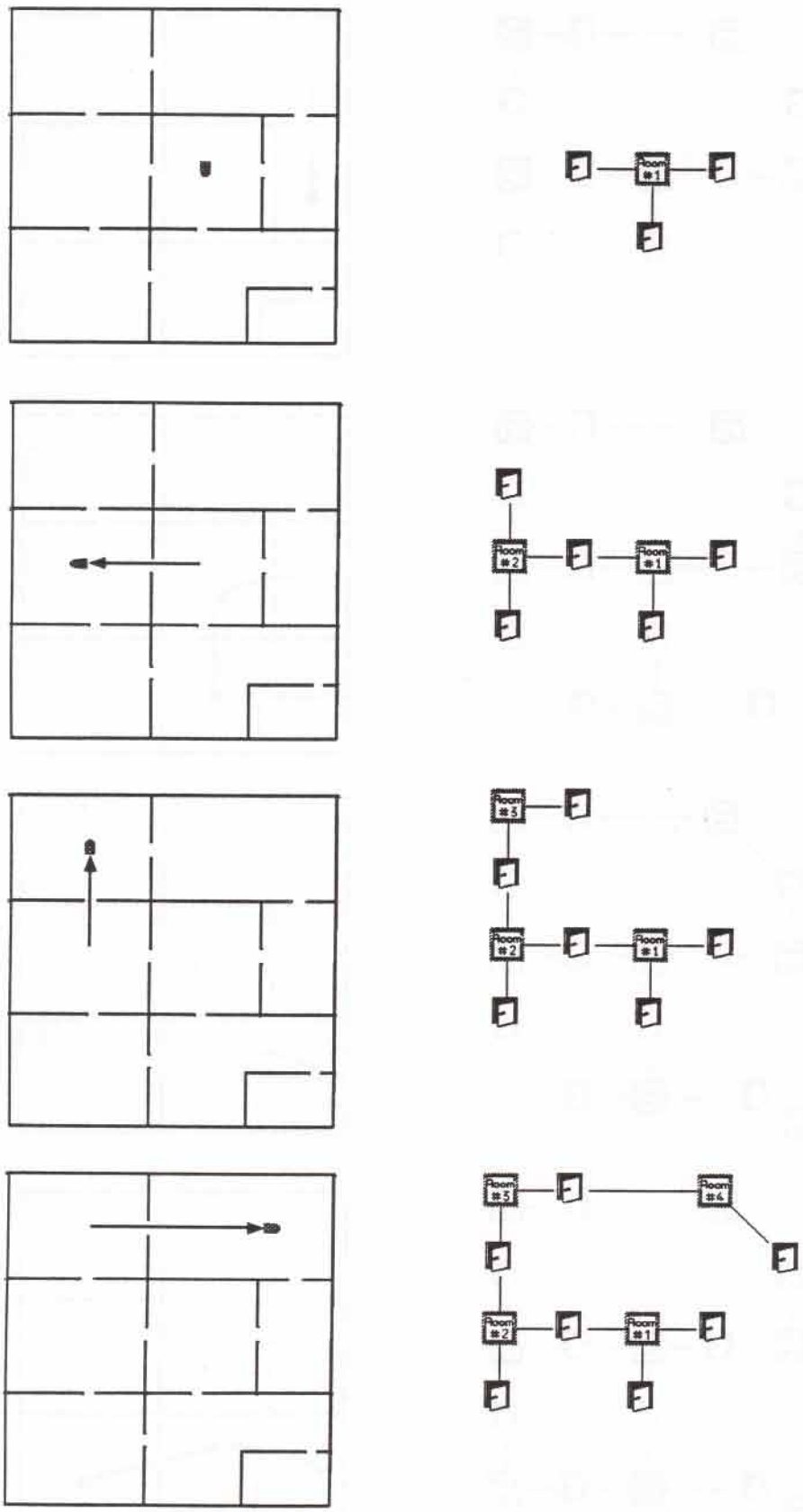


Bild 7.1.5: Schritte 1 bis 4 des Testlaufs

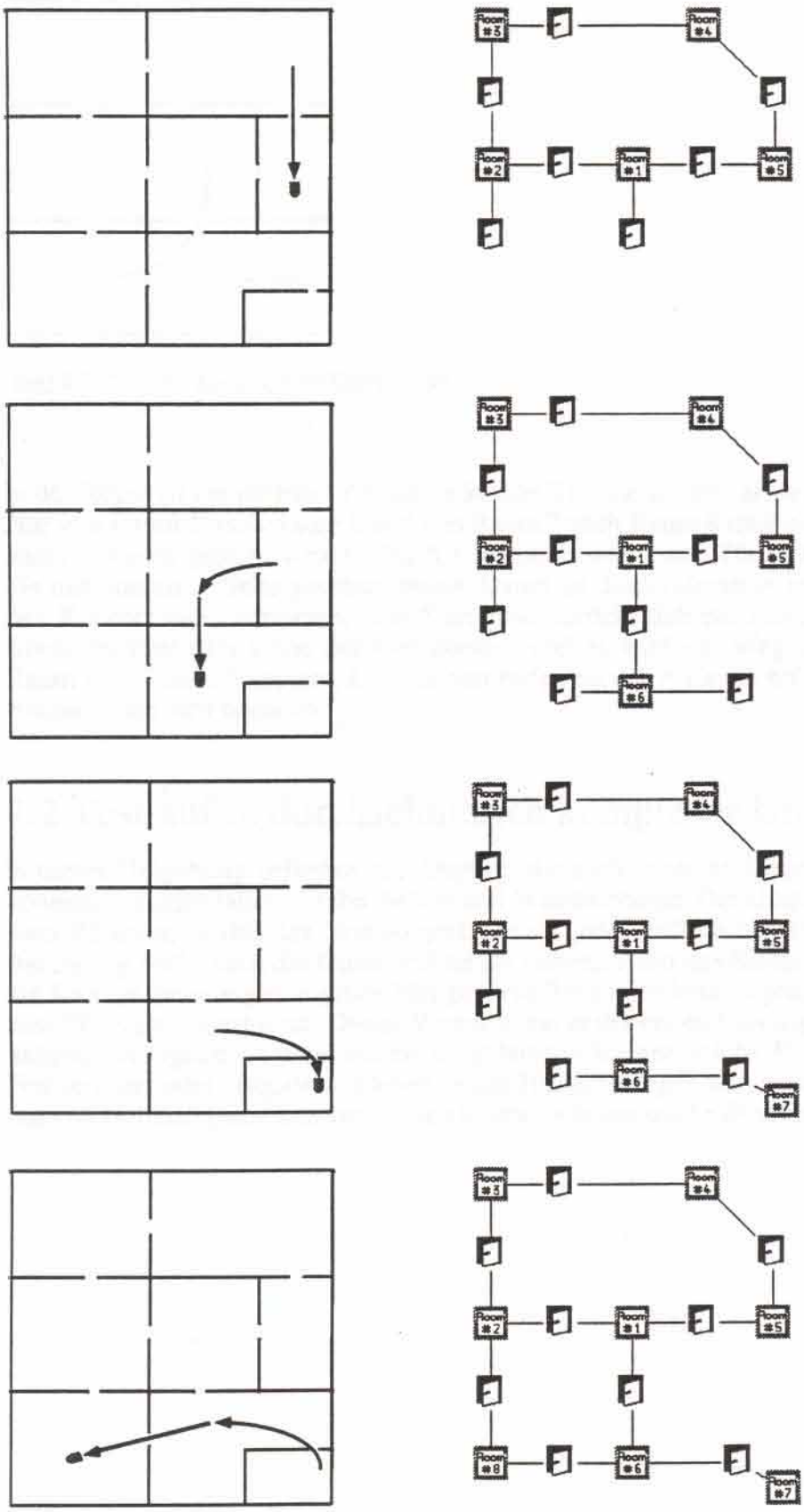


Bild 7.1.6: Schritte 5 bis 8 des Testlaufs

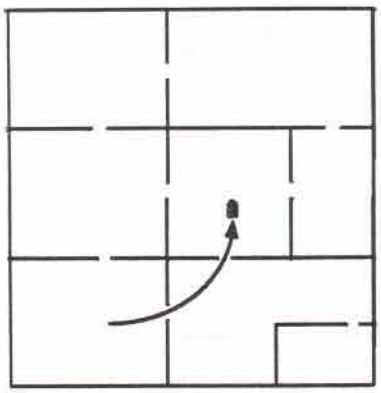


Bild 7.1.7: Fahrt zurück zum Startpunkt

In der Regel ist ein nächster Anlaufpunkt eine Tür, die zu dem aktuellen Raum gehört. Nur von Raum 5 nach Raum 6 und von Raum 7 nach Raum 8 muß ein Weg über einen anderen Raum geplant werden. Nach 8 Schritten steht keine Tür mehr zur Verfügung, die nur von einer Seite gesehen wurde. Damit ist die Exploration beendet. Man kann den Roboter nun instruieren, zum Startpunkt zurückzufahren. Auf der topologischen Ebene bereitet dies keine Schwierigkeiten, und es wird ein Weg von Raum 8 über Raum 6 zu Raum 1 geplant. Dort ist nun endgültig die Aufgabe erfüllt und der Algorithmus kann sich beenden.

7.2 Testlauf in durchschnittlich komplexer Umgebung

In dieser Umgebung befinden sich Objekte, die auch in realen Umgebungen auftreten können; darunter fallen Tische, Stühle und Wandschränke. Die Umgebung besteht aus zwei Räumen, so daß das Grundobjekt Tür auf jeden Fall auch erkannt werden muß. Beachtung sollte auch die Gitterstruktur am unteren Rand des Bildes finden. Hier muß die Kartographierung eine große Menge virtueller Linien hinzufügen, um ein geschlossenes Polygon zu erhalten. Dieser Vorgang war in der ersten Fassung des Algorithmus nahezu unmöglich. In einer realen Umgebungen können solche Punktgruppen durch Fenstergitter oder Regale entstehen. Auch Hinderniskegel, wie man sie zum Absperren von Gebäudebereichen verwenden könnte, würden solche Strukturen verursachen.

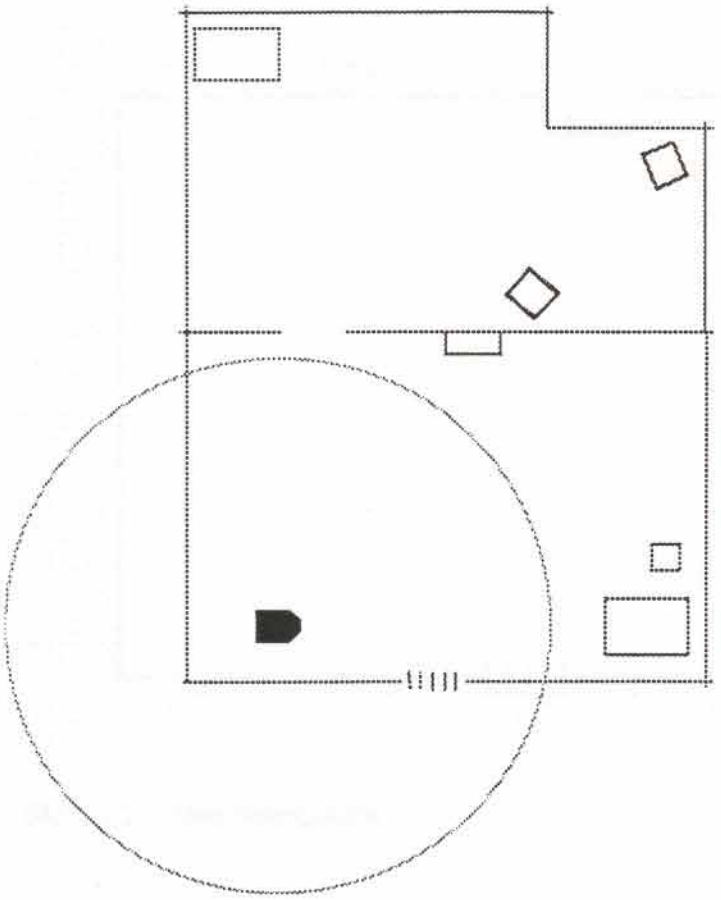


Bild 7.2.1: Startbedingungen des Testlaufs

Der Roboter startet in einer Ecke des unteren Raumes. Im folgenden sollen jeweils die Ergebnisse nach einem lokalen Explorationslauf dargestellt werden. Fast immer stellt sich der Pfad einer solchen Exploration als ein Umlauf um den Raum dar, an den die genauere Erkundung einzelner Objekte angefügt wird. Das nächste Bild zeigt das Ergebnis der ersten Raumerkundung.

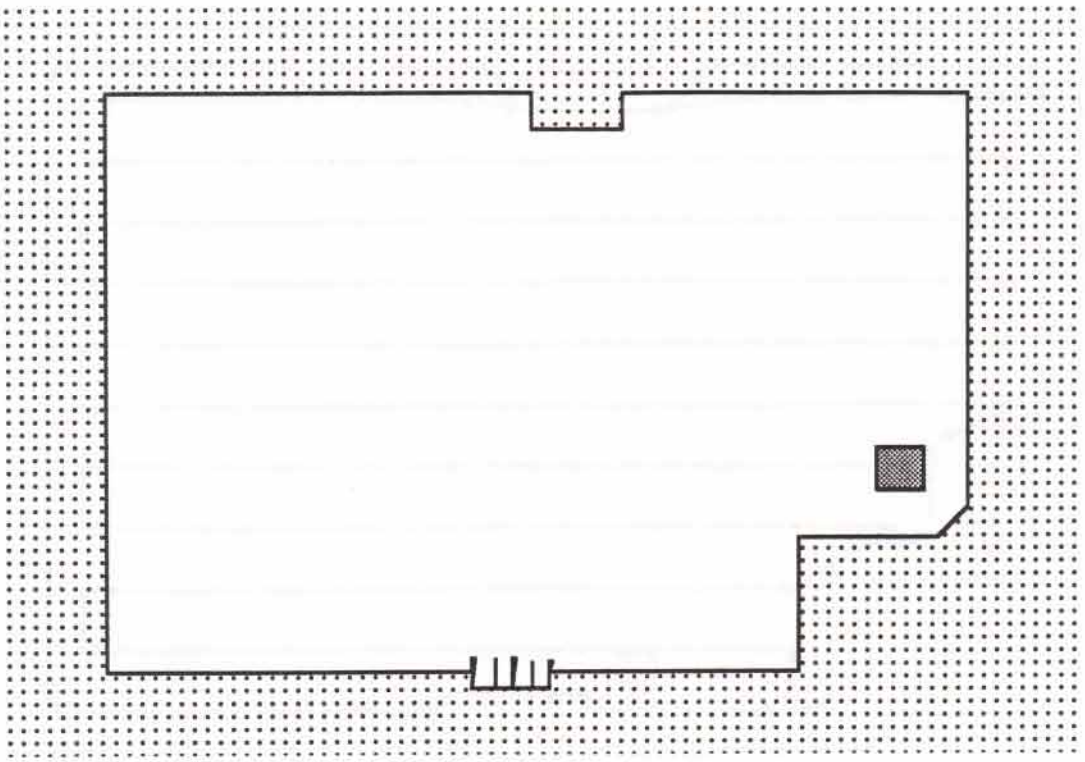


Bild 7.2.2: Erstes Teilergebnis

Man kann erkennen, daß die Struktur des unteren Raumes vollständig erkannt wurde; die Tür zum anderen Raum wurde virtuell geschlossen. Die Gitterstruktur bleibt zwar in ihrer Ausprägung bestehen, jedoch liegen die Zwischenräume geschlossen vor.

Im Rauminneren wurde nur *ein* separates Objekt wahrgenommen. Der Wandschrank wurde in das Wandpolygon integriert. Für das System besteht hier auch kein unterschied zu Wandlinien. Weit mehr Beachtung dürfte allerdings die Tatsache finden, daß auch der Tisch unten rechts in die Wandstruktur eingebunden wurde, da dieser ja nicht direkt an der Wand steht. Der Roboter konnte jedoch bei seiner Erkundung nicht alle vier Begrenzungslinien des Tisches wahrnehmen, da dieser so nah an der Wand steht, daß der Winkel zu steil wäre. Der Abstand ist jedoch auch so klein, daß der Roboter nicht in den Freiraum zwischen Tisch und Wand hineinfahren kann. Somit handelt es sich hierbei eigentlich um einen Hindernisraum. Die Reinigungsplanung würde also keine anderen Bahnen planen, wenn der Tisch durch ein eigenes Polygon repräsentiert würde. Man kann also die Regel aufstellen, daß jeder Freiraum der nicht befahrbar ist, auch virtuell geschlossen werden darf (aber nicht muß). Da diese Regel implizit durch die Viskosität der Bubble realisiert wird, braucht sie nicht speziell programmiert zu werden.

Die fertige Karte kann nun dem Reinigungsplaner [Ramsbott91] übergeben werden. Hier sollen für einige Beispiele die berechneten Reinigungsbahnen gezeigt werden.

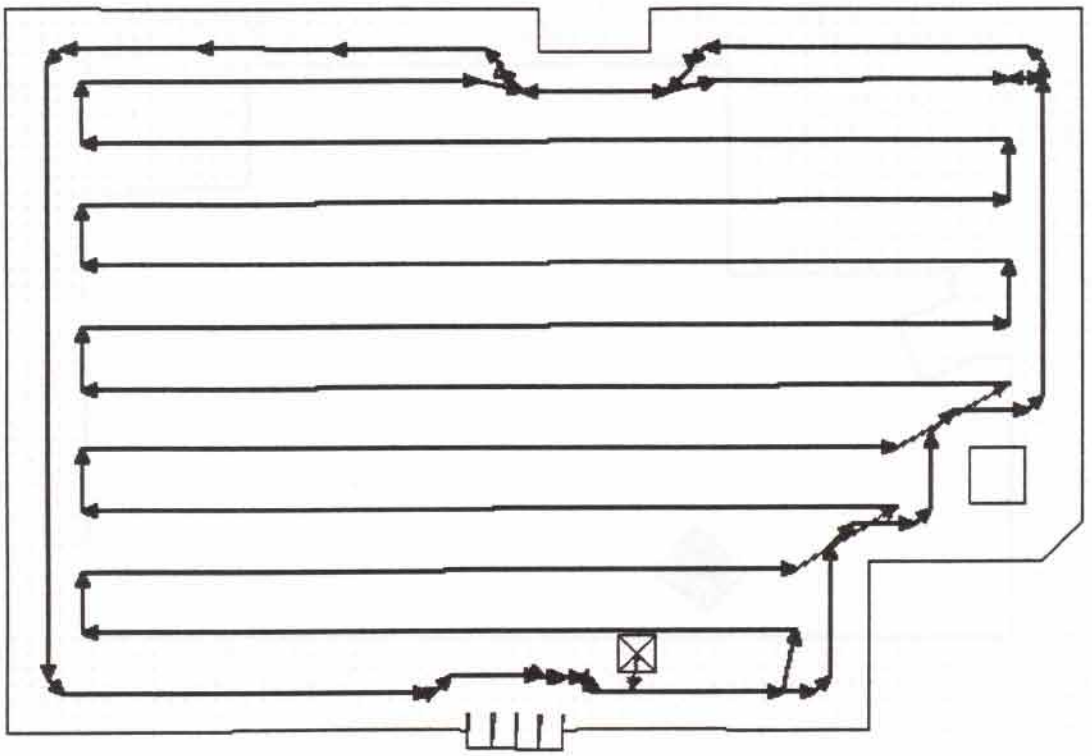


Bild 7.2.3: Reinigungsbahnen für den ersten Raum

Der Reinigungsplaner versucht zunächst eine grobe Raumorientierung zu finden. Dann legt er parallele Bahnen so, daß der Raum flächenmäßig abgefahren werden kann. Am Ende fährt er noch einmal außen an der Wand entlang.

Nach der Reinigungsphase kommt wieder ein neuer Explorationslauf. Bei der Exploration für den ersten Raum wurde eine Tür erkannt. Diese wird nun angefahren, um den Raum dahinter zu erkunden. Das Ergebnis ist im nächsten Bild zu sehen.

Auch hier wurden wieder zwei Objekte in die Wandstruktur integriert. Nur eines wurde mit einem eigenen Polygon repräsentiert.

Diese Karte kann nun auch dem Reinigungsplaner übergeben werden. Dieser ermittelt wieder die Hauptrichtung und fährt die Bahnen parallel ab. Gut ist auch hier wieder zu erkennen, daß beide Tische bei der Reinigungsplanung gleich behandelt wurden, obwohl nur einer durch ein eigenes Polygon repräsentiert wurde.

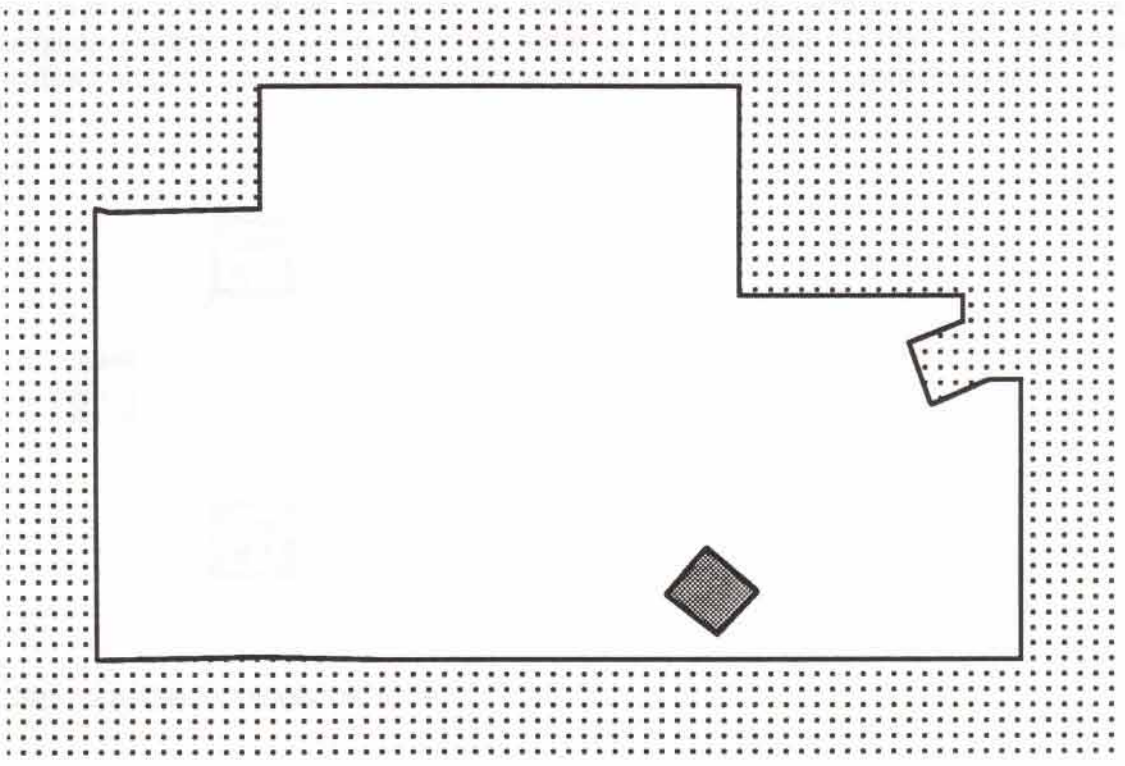


Bild 7.2.4: Zweites Teilergebnis

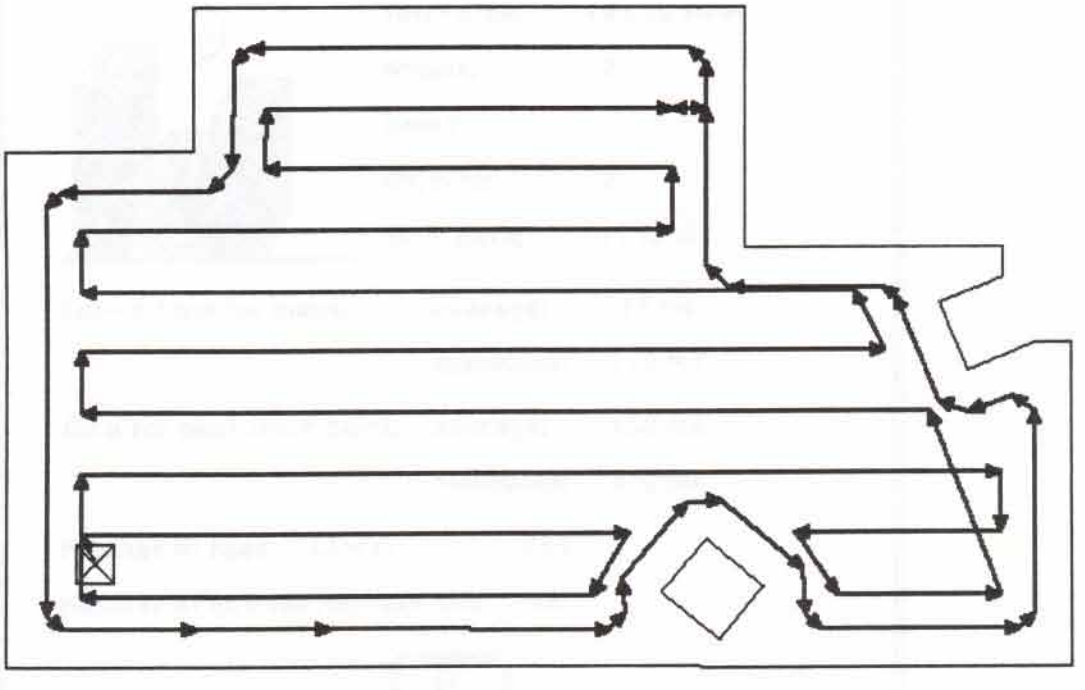


Bild 7.2.5: Reinigungsbahnen für den zweiten Raum

Im folgenden Bild ist die topologische Karte dargestellt, die während der Exploration aufgebaut wurde. Man sieht, daß keine weiteren Türen entdeckt wurden, und somit der Auftrag abgeschlossen ist. Der Roboter kann nun zum Startpunkt zurückfahren.

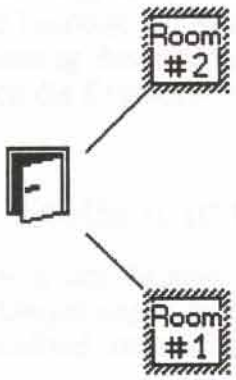


Bild 7.2.6: Die topologische Karte

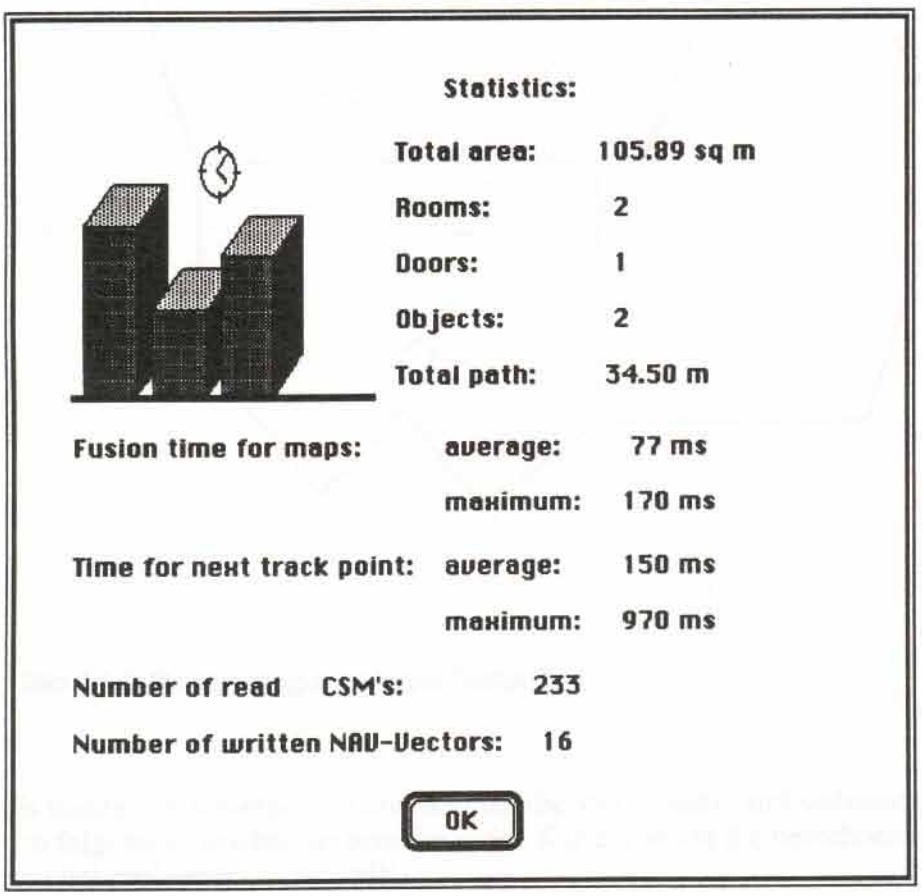


Bild 7.2.7: Die statistische Auswertung

Das statistische Auswertung gibt Aufschluß über die geleistete Arbeit während des gesamten Explorationsvorgangs. Bemerkenswert ist hierbei, daß die Fusionierungszeit im Durchschnitt sowie im Maximum unter der Zeit liegt, die der Sensor für eine Radaraufnahme benötigt (250 ms). Somit kann dieser Vorgang schritthaltend erfolgen, ohne daß Aufnahmen verloren gehen.

Anders liegt der Fall bei der Wegplanung. Im Mittel wird zwar auch nicht viel Zeit dafür benötigt, im Maximum tritt jedoch eine größere Zeit auf, was auf eine häufige Benutzung des internen Navigators zurückzuführen ist. Alles in allem verhält sich jedoch die Exploration in diesem Beispiel echtzeitfähig.

7.3 Testlauf in nicht rechtwinklig begrenzten Räumen

Das folgende Beispiel soll demonstrieren, daß der Einsatz des Systems nicht auf übliche Umgebungen begrenzt ist, sondern auch in Räumen möglich ist, die nicht dem Normalbild entsprechen. Folgende Umgebung ist gegeben.

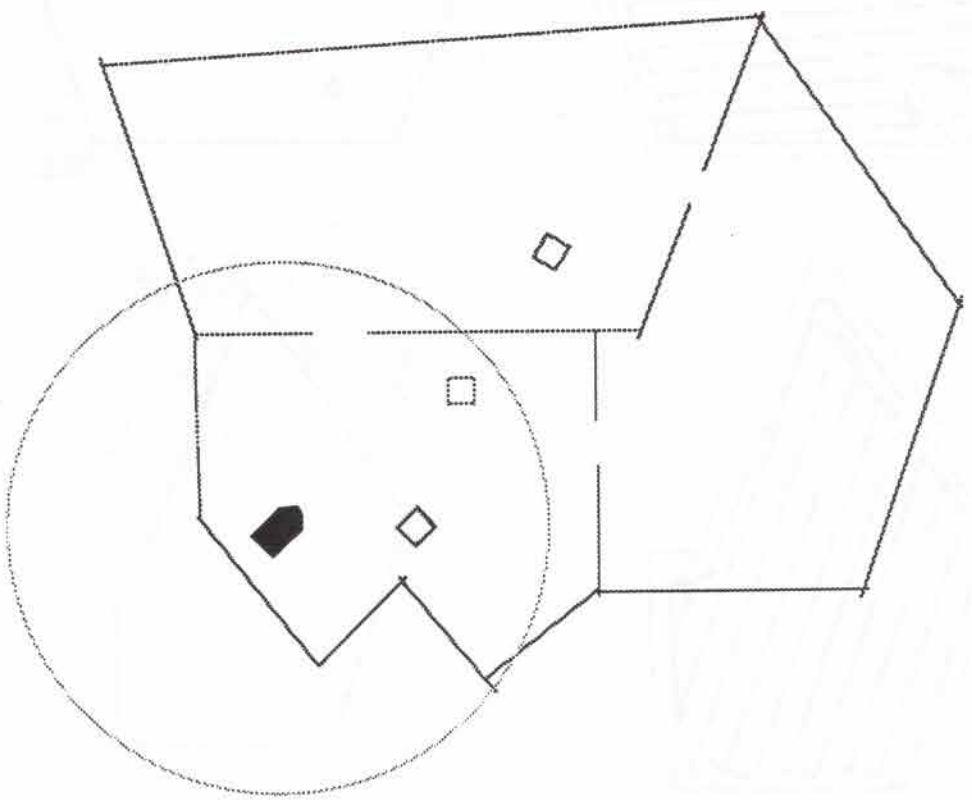


Bild 7.3.1: Startbedingungen des Testlaufs

Es handelt sich hierbei um drei Räume, die vier-, sechs- und siebeneckig begrenzt sind. Im folgenden werden die resultierenden Karten, sowie die berechneten Reinigungsbahren nebeneinander dargestellt.

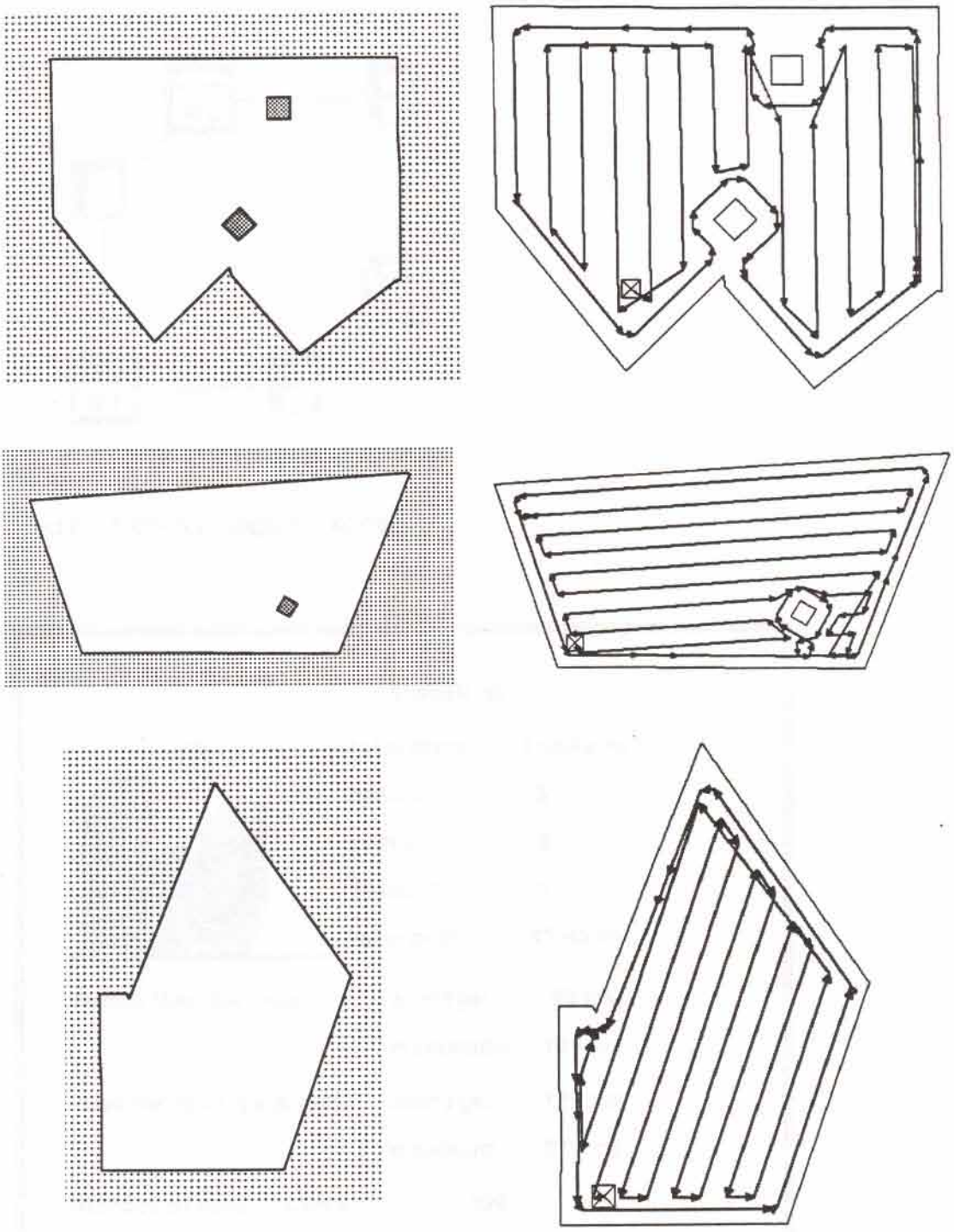


Bild 7.3.2: Die Ergebisse und Reinigungsbahnen für drei Räume

Man erkennt, daß auch hier der Auftrag erfüllt wurde. Der Reinigungsplaner kann zwar nicht mehr die Reinigungsbahnen so legen, daß sie zu zwei Wänden parallel liegen, aber durch zusätzliche Bahnen wird eine Flächendeckung erreicht.

Auch hier soll der topologische Graph und die statistische Auswertung gezeigt werden.

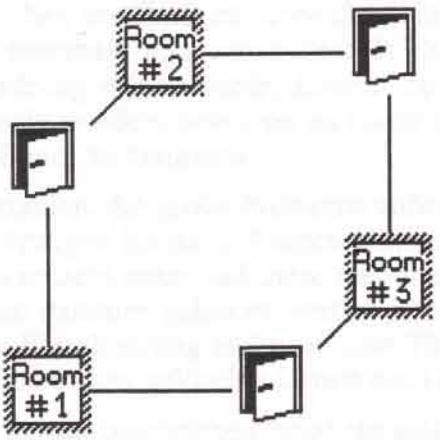


Bild 7.3.3: Die topologische Karte

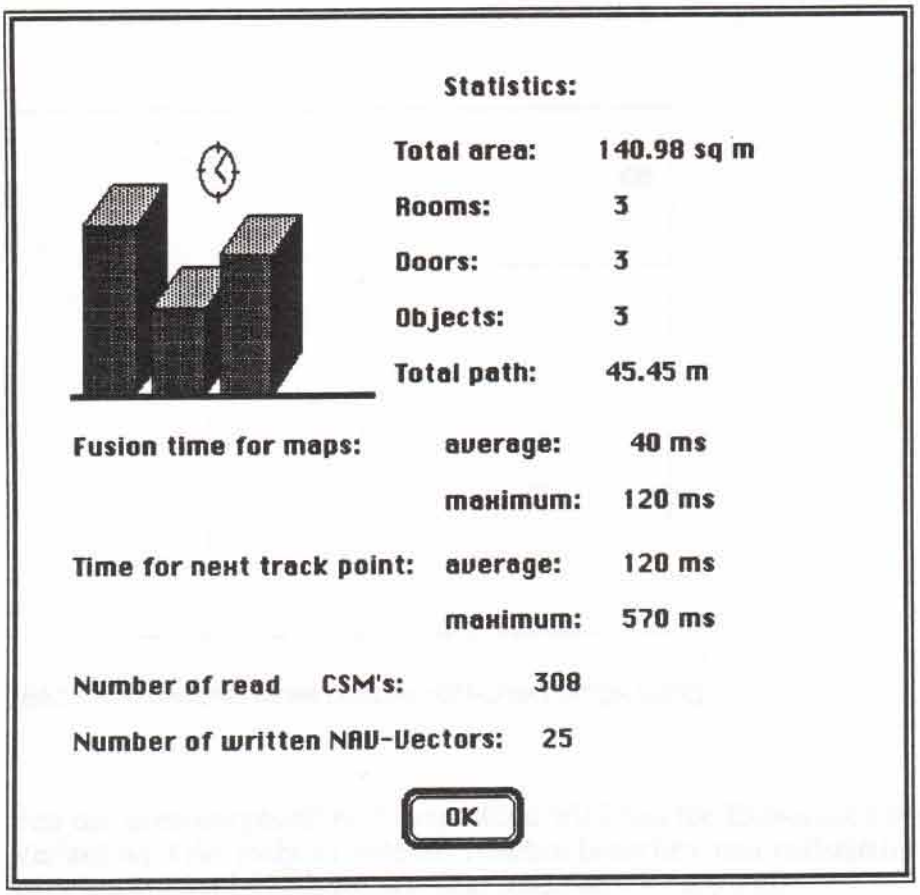


Bild 7.3.4: Die statistische Auswertung

7.4 Eine Umgebung, die Probleme aufwirft

Wie oben erwähnt, ist eines der Hauptprobleme bei der Exploration zu entscheiden, was innerhalb und was außerhalb des aktuellen Raums liegt, noch bevor der Raum vollständig erfaßt wurde. Linien die wahrscheinlich außerhalb liegen, müssen dabei gelöscht werden, bevor sie zu einem neuen Ziel führen, und den Roboter veranlassen, den Raum zu verlassen.

Situationen, die große Probleme aufwerfen sind deshalb solche Räume, die große Einschnürungen besitzen. Liegen Objekte hinter einer solchen Einschnürung, dann liegt der Verdacht nahe, daß diese nicht zum aktuellen Raum gehören. Dies führt dazu, daß Linien dahinter gelöscht werden können. Stellt sich nachher heraus, daß es sich bei einer Einschnürung nicht um eine Tür handelt, könnte der Roboter noch einmal dorthin fahren, um gelöschte Linien noch einmal zu erfassen.

Wie jedoch beschrieben, wird ein gelöschter Bereich nicht noch einmal angefahren, da das System so ins Schwingen geraten könnte. Entsprechende Linien bleiben gelöscht, und sind somit in keiner Karte verzeichnet.

Die folgende Karte zeigt ein Beispiel für einen Raum mit starken Einschnürungen.

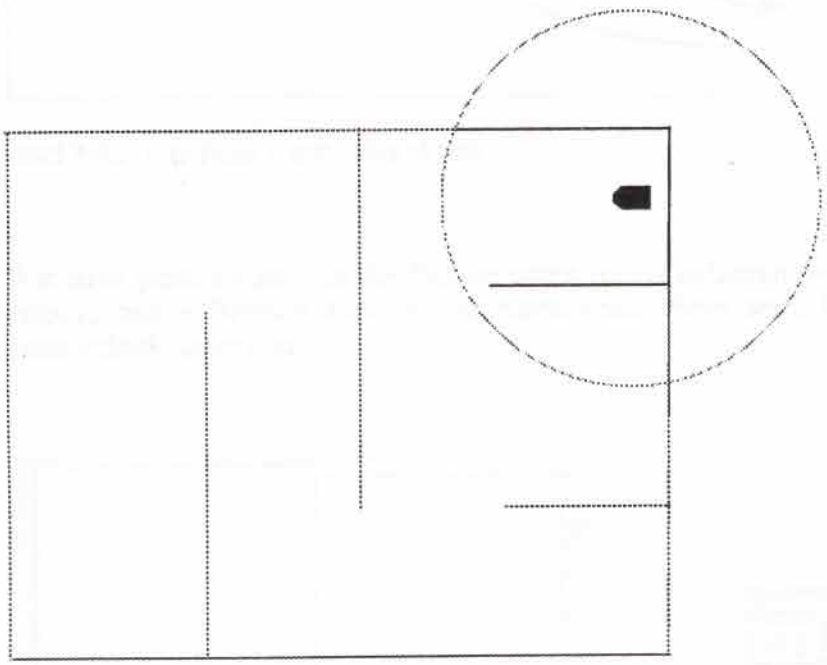


Bild 7.4.1: Beispiel einer problematischen Umgebung

Von der oben dargestellten Startposition wird nun die Exploration begonnen. In deren Verlauf wird der Roboter jede der Nischen besuchen, und vollständig erfassen. Der resultierende Pfad ist im nächsten Bild schematisch dargestellt.

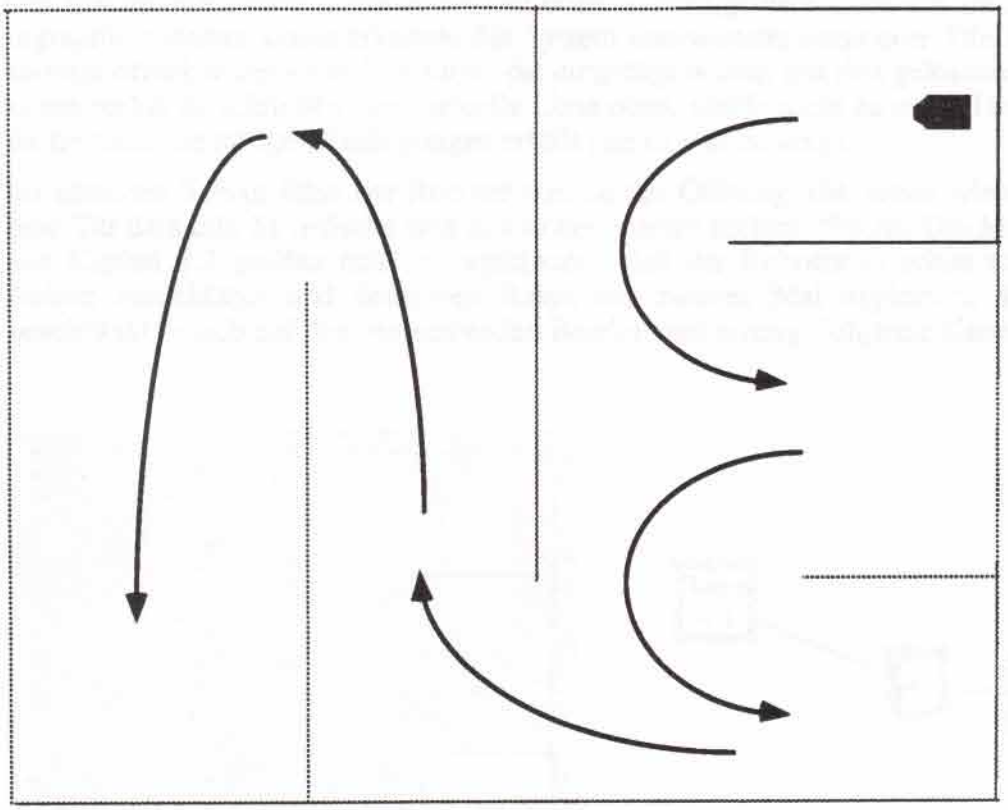


Bild 7.4.2: Pfad des Explorationslaufs

Wie man sieht, ist auch in die Nische unten rechts gefahren worden. Damit müßte der entsprechende Bereich auch in der Karte verzeichnet sein. Die resultierende Karte sieht jedoch anders aus.

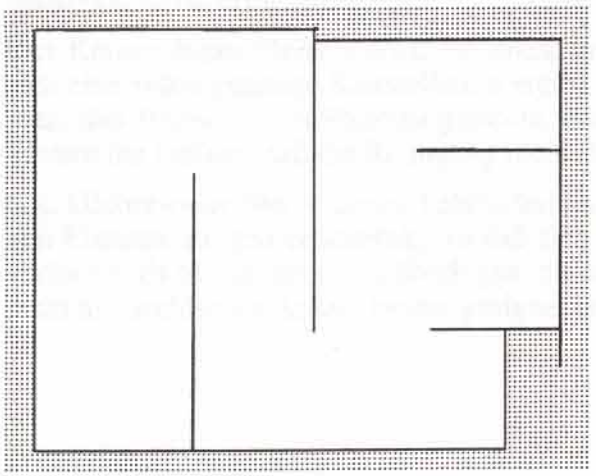


Bild 7.4.3: Erstes Teilergebnis

Die Nische unten rechts, sowie ein Teil oben wurden gelöscht, obwohl sie schon kartographiert waren. Dafür erkannte das System unerwarteterweise eine Tür. Diese entstammt offenbar der virtuellen Linie, die eingefügt wurde, um den gelöschten Bereich unten rechts zu schließen. Die virtuelle Linie oben, wurde nicht zu einer Tür gemacht, da sie nicht die nötigen Bedingungen erfüllt (sie ist viel zu lang).

Im nächsten Schritt fährt der Roboter nun zu der Öffnung, die seiner Meinung nach eine Tür darstellt. Er befindet sich nun in der unteren rechten Nische. Die Maßnahmen aus Kapitel 3.7 greifen nun und verhindern, daß der Roboter in schon exploriertes Gebiet zurückfährt und denselben Raum ein zweites Mal exploriert. Stattdessen beschränkt er sich auf den verbleibenden Bereich und erzeugt folgende Karte:

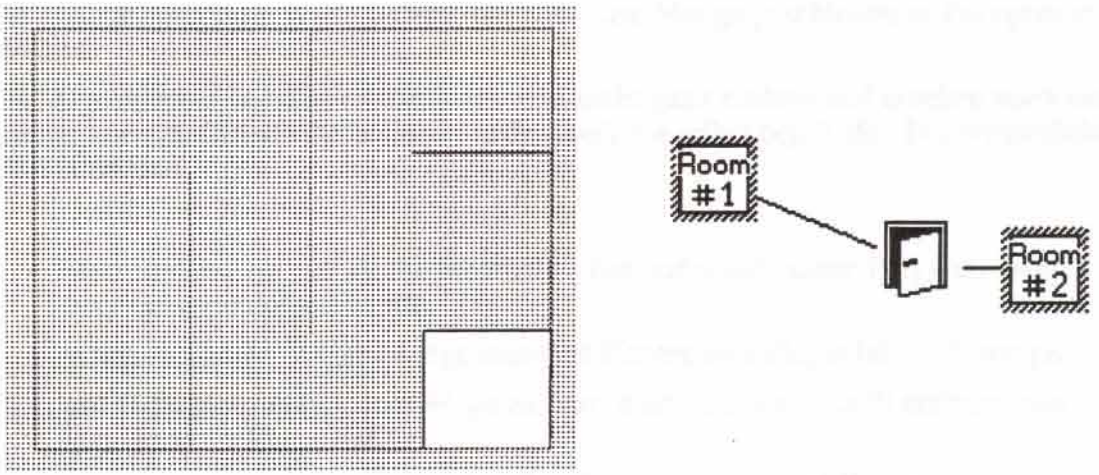


Bild 7.4.4: Zweites Teilergebnis

Der Auftrag gilt jetzt als erfüllt, da keine offenen Türen mehr vorhanden sind. In diesem Fall wurde somit trotz der Fehlinterpretation der Umgebung, eine flächendeckende Karte ermittelt.

Das Kernproblem bleibt jedoch bestehen, und man kann nicht immer erwarten, daß sich eine solch günstige Konstellation ergibt. Im schlechtesten Fall kann es dazu kommen, daß Bereiche von Räumen gelöscht, und nicht wieder angefahren werden. Somit besteht die Gefahr, daß die Reinigung nicht flächendeckend erfolgt.

Glücklicherweise bleibt dieses Fehlverhalten auf solche ungünstigen Räume mit starken Einschnürungen beschränkt, so daß sich der Einsatz dieses Verfahrens nicht ganz verbietet. Es bleibt jedoch zu überlegen, ob nicht für beliebige Einsatzumgebungen ein nicht hierarchischer Ansatz besser geeignet ist.

8. Zusammenfassung und Ausblick

In dieser Arbeit wurden Algorithmen und Techniken vorgestellt, die es ermöglichen, einen vollständig unbekannten Komplex von Räumen zu erkunden, ohne daß zu irgendeiner Zeit von außen Hilfe zur Verfügung gestellt wird. Es wurde dabei an die Struktur und den Aufbau der Räume keine Bedingungen gestellt. Als einzige Eingabe dient ein Entfernungssensor. Die Exploration sollte hierarchisch erfolgen, d.h. es gibt eine lokale Exploration, die sich mit Objekten und Begrenzungen innerhalb von Räumen befaßt, und eine globale Exploration, die den Zusammenhang der Räume untereinander erkundet. Als Ausgabe sollte das System eine, für z.B. eine flächendeckende Reinigung optimierte Karte liefern, die aus einer Menge geschlossener Polygonzügen besteht.

Die Algorithmen, die dies ermöglichen sind nicht ganz einfach und arbeiten auch nicht immer korrekt. Gründe dafür liegen in der Aufgabe selbst begründet. Die wesentlichen Probleme sind

- festzustellen, ob ein Punkt innerhalb oder außerhalb eines Polygons liegt, *bevor* es geschlossen wurde,
- virtuelle Kanten in eine Menge massiver Kanten einzufügen (siehe Anhang),
- mit Hilfe eines Navigators Wege zu berechnen, der nur Zugriff auf eine rein geometrische Ebene hat,
- nur anhand von Linien, abstrakte Objekte wie Türen und Räume zu erkennen.

Für die ersten beiden Probleme wurden Näherungsalgorithmen angegeben. Zu den letzten Problemen wurde versucht, eine Menge von Heuristiken anzugeben, die in den meisten Fällen gute Ergebnisse liefern. Um die Lösung noch effektiver zu gewinnen, wurde das streng hierarchische Prinzip aufgegeben, und eine Rückkopplung eingeführt, die es ermöglicht, abstrakte Informationen der geometrischen Ebene zugänglich zu machen. Jedoch blieb die inhaltliche Trennung der beiden Ebenen bestehen.

Diese Trennung, nämlich im Raum eine lokale Exploration durchzuführen, und im Gesamtkomplex eine globale Erkundung vorzunehmen, ist nicht unbedingt zwingend. Der Vorteil liegt sicher darin über große Distanzen die Graphensuche als Mittel der Navigation benutzen zu können. Würde die Trennung entfallen, so müßte der Roboter auf einmal den gesamten Komplex explorieren. Konstruktionen wie Räume und Türen, wären dann gar nicht existent. Somit würden auch viele Probleme wegfallen. So wäre es dann beispielsweise unkritisch, wenn der Navigator einen Weg plant, der aus dem aktuellen Raum hinausführt.

Es hängt sicher von der Einsatzumgebung ab, welcher der Möglichkeiten die bessere ist. Besteht z.B. die Umgebung aus einer Unzahl von Räumen (und damit Linien), so ist der hierarchischen Konzeption den Vorzug zu geben. Handelt es sich jedoch um ein relativ begrenztes Gebiet von einigen Räumen, die in sich sehr kompliziert aufgebaut sind, so ist eine Lösung mit nur einer Explorationsebene vorzuziehen, da erstens der Navigator ohne weiteres auch mehrere Räume befahren kann, und zweitens, die Probleme mit der Raumstruktur derart überwiegen würden, daß unter Umständen keine

brauchbare Lösung entsteht.

Einige weiterführende Probleme wurden bisher ausgeklammert.

- **Dynamische Objekte:**

  Die Sensorvorverarbeitung versucht zwar dynamische Objekte auszufiltern, jedoch gelingt dies nur zum Teil. Da während der Exploration eine abstraktere Sicht auf die Daten vorliegt, könnte man hier noch einmal versuchen, dynamische Objekte zu erkennen, bevor sie eingetragen werden. Da dynamische Objekte später in der Karte Spuren hinterlassen würden, die wie feststehende Gegenstände aussehen, sind bewegte Objekte bis jetzt noch nicht erlaubt.

- **Adaptives Verhalten:**

  Sollte die Kartenfusionierung einmal nicht mit der Sensoreingabe schritthalten können, so gehen Karten verloren. Man könnte sich vorstellen, daß man in einem solchen Fall die Geschwindigkeit des Roboters senkt, damit sich das Umweltbild nicht zu schnell ändert. Experimente haben jedoch gezeigt, daß die Fusionierung selbst im schlechtesten Fall wesentliche schneller als die Kartenproduktion ist. Für diese Art des Algorithmus ist somit ein adaptives Verhalten nicht sinnvoll. Für andere Explorationsverfahren könnte dieser Ansatz jedoch interessant sein.

- **Mehrere Höhenschichten:**

  Die Kartographierung ist in unserem Fall rein zweidimensional. Der Sensor erhebt jedoch dreidimensionale Daten, so daß hier viel Information verschenkt wird. Man könnte diese Information z.B. ausnutzen, Linien ein bestimmtes Gewicht zuzuordnen, wenn sie in mehreren Höhenschichten erfaßt wurden. Nur Linien über einem bestimmten Gewicht könnten dann in die endgültige Karte als Wand eingefügt werden, da Wände in der Regel in mehreren Höhenschichten aufgenommen werden. Auch andere Anwendungen sind denkbar. Man müßte dann aber auch die Fusionierungsalgorithmen abändern. Das könnte soweit führen, daß man sogar die Bubblestruktur um eine Dimension erweitert. Allerdings wären die Algorithmen für eine dreidimensionale Bubble wesentlich komplizierter. Jedoch sind viele Möglichkeiten denkbar.

Ein weiterer Ansatz würde zumindest die Problematik der Kartenfusionierung vereinfachen. Man könnte die vom Sensor erfaßten Linien einem *nachbarschafterhaltenden neuronalen Netz* zufügen. Hierbei könnte das Netz so konstruiert sein, daß zusammenhängende Gebiete auch in der internen Karte zusammenhängen, eigenständige Objekte aber auch durch eigene Bereiche repräsentiert werden. Damit würde eine abschließende Analyse darüber, welche Linien, zu bestimmten Objekten gehören, wegfallen.

Eine Beschreibung sogenannter topologieerhaltender neuronaler Netze, deren Grundidee von Kohonen stammt, findet man in [Brause91] sowie in [Ritter90]. Dort wird auch demonstriert, wie man das Problem des Handelsreisenden mit neuronalen Netzen in einer sehr guten Annäherung lösen kann. Wie im Anhang beschrieben wird, ist dieses Problem mit dem hier zu lösenden eng verwandt, so daß ein entsprechender Ansatz zumindest vielversprechend ist.

A.1 Optimales Einfügen virtueller Kanten

Dieses Problem stellt gewissermaßen ein Kernproblem dar. Eine Menge real aufgenommener Linien muß früher oder später mit virtuellen Linien angereichert werden, um geschlossene Polygonzüge zu erhalten (siehe Kapitel 3.4.2.c). Hierbei ist nicht von vornherein klar, wie die real aufgenommenen Linien verknüpft werden sollen, damit ein Minimum virtueller Linien entsteht. Auch in der die Reinigungsplanung existiert ein analoges Problem. Dort müssen zusammenhängende Bereiche (Teilräume) in einer Reihenfolge angefahren werden, so daß ein Minimum von zusätzlichem Fahrweg entsteht [Ramsbott91].

Im folgenden soll zuerst die Komplexität dieser Aufgabe im allgemeinen Fall untersucht werden. Dann wird das oben vorgestellte Näherungsverfahren betrachtet, und versucht, Schranken anzugeben, in denen sich die Güte des Verfahrens bewegt.

A.1.1 Komplexität im allgemeinen Fall

Das Problem, das gelöst werden soll, lautet folgendermaßen:

Gegeben ein azyklischer Graph ohne Verzweigungen, der aus n Teilgraphen besteht sowie eine Abstandsfunktion d, die einem Paar von Punkten des Graphs deren Abstand zuordnet.

Gesucht wird eine weitere Menge von Kanten, so daß ein Rundweg existiert, der alle Punkte verbindet.

Da von den gegebenen Teilgraphen nur die jeweiligen Start- und Endpunkte interessieren, können diese, ohne das Problem zu verändern, durch jeweils eine Kante ersetzt werden. Das Problem stellt sich somit formal folgendermaßen dar:

Gegeben: Graph $G=(V,E)$ und $d: V \times V \rightarrow \mathbb{N}$ totale Gewichtsfunktion mit

- $\forall_{e \in E}: e=(u,v) \Rightarrow u \neq v$ (d.h. keine elementaren Zykel)
- $\forall_{e,f \in E}: e=(u,v), f=(x,y), e \neq f \Rightarrow \{u,v\} \cap \{x,y\} = \varnothing$ (d.h. keine zwei Kanten sind miteinander verbunden)

Gesucht Menge zusätzlicher Kanten E2 mit

- $\forall_{e \in E2}: e=(u,v) \Rightarrow u,v \in V$ (Jede neue Kante verbindet alte Punkte)

- $\forall_{v\in V}\, \exists^{1}_{e\in E2,\, u\in V}$: $e=(u,v) \vee e=(v,u)$ (Jeder Punkt wird auch von genau einer neuen Kante erreicht)
- $\forall_{u,v\in V}$: $u \leftrightarrow v$ mit $u \leftrightarrow v$ gdw $((u,v)\in E \cup E2) \vee (\exists_{w\in V}: u \leftrightarrow w, w \leftrightarrow v)$ (d.h. im neuen Graph ist jeder Punkt mit jedem verbunden)
- $(\Sigma_{e\in E2, e=(u,v)}\, d(u,v))$ ist minimal.

Im folgenden soll gezeigt werden, daß das Problem im allgemeinen Fall (d.h. ohne zusätzliche Einschränkungen des Graphs oder der Abstandsfunktion) nicht polynomial ist. Der Beweis soll folgendermaßen geführt werden (gemäß [Horwitz78]):

- Konstruiere eine Transformation, die jedem Problem des Handlungsreisenden (TSP) ein virtuelle-Kanten-Problem (VEP) zuordnet, so daß gilt: hat man VEP gelöst, so liegt auch eine Lösung für das entsprechende TSP vor
- Weise nach, daß die Transformation in polynomialer Zeit erfolgen kann

Vom TSP weiß man, daß es nicht in polynomialer Zeit gelöst werden kann (vorausgesetzt $P \neq NP$). Das entsprechende Entscheidungsproblem ist sogar NP-vollständig. Wäre das VEP polynomial, dann hätte man ein einfaches Verfahren, auch TSP in polynomialer Zeit zu berechnen. Man würde einfach folgendes durchführen:

- Konstruiere nach Vorschrift aus einem TSP ein VEP (in polynomialer Zeit)
- Löse das VEP (in polynomialer Zeit)
- Verwende die Lösung als Lösung für das TSP

Hat man eine Transformation gefunden, folgt dann daraus, das das VEP nicht polynomial sein kann, da wir vom TSP wissen, daß kein Algorithmus es in polynomialer Zeit löst. Die gesuchte Transformation ist hier sehr einfach:

Sei TSP ein Problem des Handlungsreisenden mit n Städten P_i ,$1 \leq i \leq n$, und der Abstandsfunktion $d(P_i,P_j)$. Konstruiere daraus ein VEP:

- ordne jedem P_i zwei Punkte u_{i1},u_{i2} zu, $1 \leq i \leq n$
- ordne jedem P_i eine Kante e_i zu, $1 \leq i \leq n$, mit $e_i=(u_{i1},u_{i2})$
- ordne der Abstandsfunktion d eine Funktion d' zu mit

  $d'(u_{ij},u_{kl})=d(P_i,P_k)$, $1 \leq i, k \leq n$, $1 \leq j, l \leq 2$

Da $d(P_i,P_i)=0$ für $1 \leq i \leq n$, gilt $d'(u_{i1},u_{i2})=0$. Somit entarten die Kanten des VEP zu Punkten. Sollen hierzu die virtuelle Kanten minimal sein, so erhält man direkt eine minimale Rundreise für das TSP.

Diese Transformation kann sicher in polynomialer Zeit erfolgen, sie ist sogar in der Zeit O(n) berechenbar. Somit ist der Nachweis erbracht, daß das Problem virtuelle Kanten optimal einzufügen, im allgemeinen Fall nicht in polynomialer Zeit gelöst werden kann.

A.1.2 Güte des Näherungsverfahrens

Das in Kapitel 3.4.2.c beschriebene Näherungsverfahren soll nun auf seine Güte hin untersucht werden. Hierbei wird als Vereinfachung angenommen, daß die massiven Linien verschwindend klein sind, so daß ein optimaler Rundweg in einer Punktmenge gesucht werden kann.

A.1.2.1 Güte im allgemeinen Fall

Es soll nun untersucht werden, wie schlecht das Verfahren maximal werden kann, wenn keine zusätzlichen Randbedingungen an die Abstandsfunktion gegeben sind. Hierzu sollen erst einmal folgende Vorüberlegungen angestellt werden:

Drei Punkte lassen sich nur in einer Weise verbinden, so daß hier garantiert der optimale Fall vorliegt. Kommt ein vierter Punkt hinzu, so wird das Näherungsverfahren alle Kombinationen durchtesten, die möglich sind. Somit wird auch bei vier Punkten das Optimum gefunden. Der erste Fall, in dem das Verfahren einen Fehler machen kann, liegt also frühestens bei fünf Punkten vor. Hierzu soll das folgende Rundreiseproblem betrachtet werden:

	A	B	C	D	E
A	-	1	2	1	1
B	1	-	1	2	x
C	2	1	-	1	1
D	1	2	1	-	x
E	1	x	1	x	-

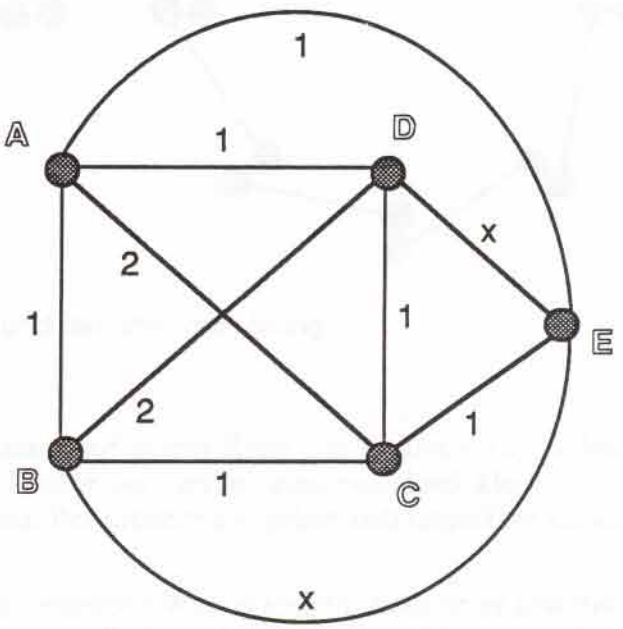


Bild A.1.1: Beispiel eines TSP mit schlechtem Ergebnis des Näherungsverfahrens

Hierbei sei x ein unbestimmter, aber sehr großer Wert. Es läßt sich leicht feststellen, daß der optimale Rundweg A-E-C-D-B-A ist mit den Kosten 6. Angenommen der Näherungsalgorithmus bekommt die Punkte nun in der Reihenfolge A, B, C, D, E zugeteilt. Für die ersten vier Schritte ermittelt er dann die Reihenfolge A-B-C-D-A. Wird nun E hinzugefügt, so steht nur noch ein Minimum von 4+x zur Verfügung (z.B. A-E-B-C-D-A). Somit ist das Verhältnis gefundene Kosten zu minimale Kosten (4+x)/6. Da man x beliebig groß wählen kann, ist also dieser Näherungsalgorithmus im allgemeinen Fall beliebig schlecht.

A.1.2.2 Güte im Fall der Dreiecksungleichung

Diese sehr schlechte Abschneiden muß relativiert werden. In der Umgebung, in der der Algorithmus eingesetzt wird, gilt die Dreiecksungleichung. Das schlechte Ergebnis im vorigen Fall kam nur deshalb zustande, da diese Ungleichung hochgradig nicht galt (z.B hatte der Weg von B-E die Kosten x, und von B-A-E nur die Kosten 2<<x). Für den Fall der Dreiecksungleichung ist das Verhalten wesentlich besser. Es soll aber gezeigt werden, daß selbst dann ein doppelt so langer Weg wie das Optimum berechnet werden kann.

Hierzu sei folgendes Rundreiseproblem gegeben:

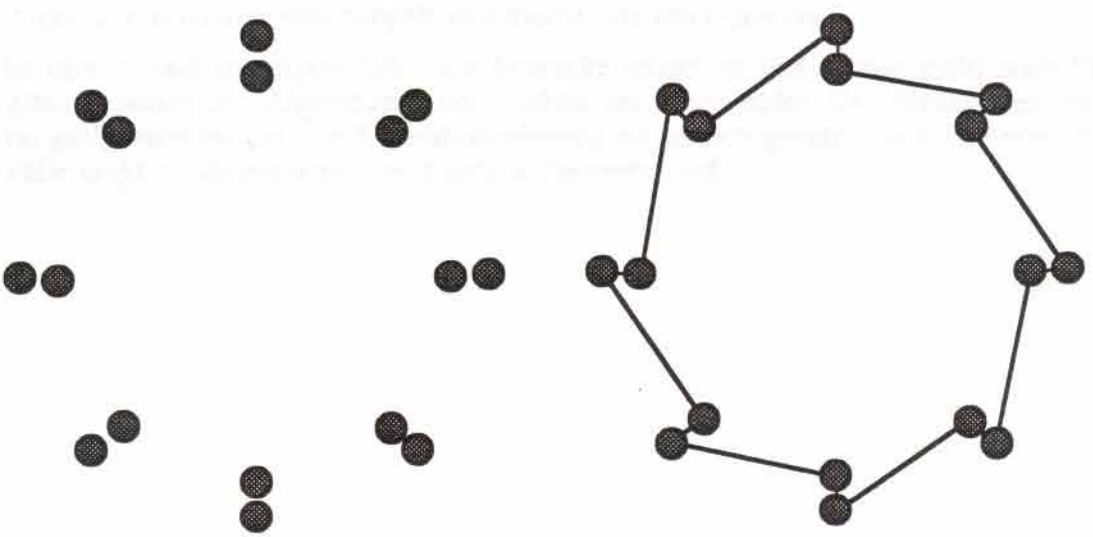


Bild A.1.2: Ein Rundreiseproblem und die optimale Lösung

2n Punkte sind paarweise äquidistant auf einem Kreis mit Radius r angeordnet. Die Abstände der eng benachbarten Punkte sei hierbei verschwindend klein. Rechts ist eine optimale Rundreise angegeben. Bei großem n ergeben sich ungefähre Kosten von $2\pi r$.

Der Näherungsalgorithmus würde denselben Weg erzeugen, bekäme er erst die äußeren Punkte, dann die inneren. Wird die Reihenfolge jedoch modifiziert, so kann folgendes passieren:

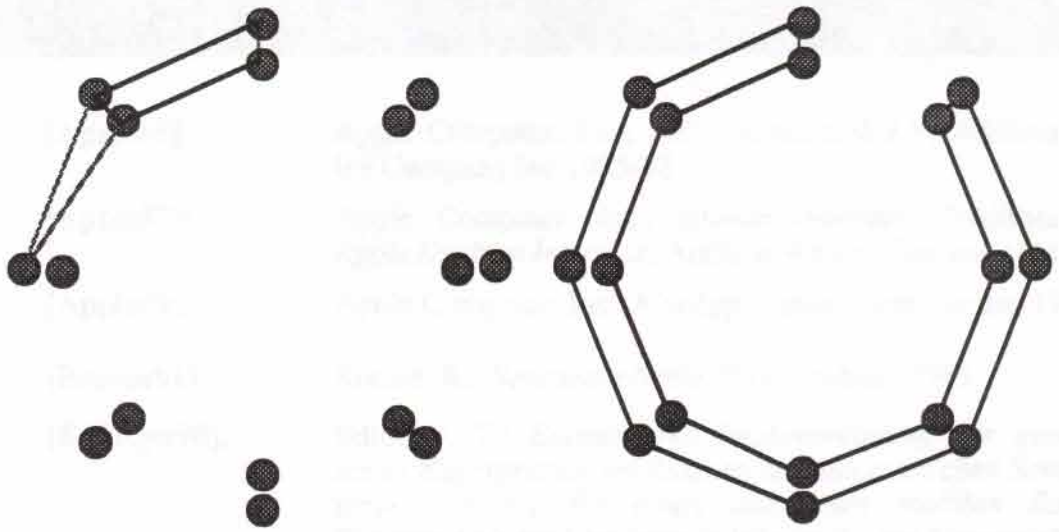


Bild A.1.3: Die Lösung des Näherungsalgorithmus

Hier wurden jeweils einzelne Paare hintereinander dem Algorithmus zugefügt. In diesem Fall konstruiert der Algorithmus eine Rundreise von $\approx 4\pi r$ (für große n). Der Algorithmus ist hier also doppelt so schlecht, wie das Optimum.

Es bliebe noch zu zeigen, daß diese Schranke scharf ist, daß es also nicht noch Fälle gibt, in denen der Algorithmus noch schlechter abschneidet. Der Einsatz des Näherungsalgorithmus bei der Kartenfusionierung hat jedoch gezeigt, daß entweder diese Fälle nicht existieren oder bisher nicht aufgetreten sind.

Literatur

[Apple85]: Apple Computer, Inc.: *Inside Macintosh I-V*, Addison Wesley Company Inc 1985-88

[Apple87]: Apple Computer, Inc.: *Human Interface Guidlines: The Apple Desktop Interface*, Addison Wesley Company Inc 1987

[Apple89]: Apple Computer, Inc.: *MacApp*, Apple Computer Inc 1989

[Brause91]: Brause, R.: *Neuronale Netze*, B.G. Teubner, 1991

[Edlinger90]: Edlinger, T.: *Entwurf und Implementierung der geometrischen Kartographie im Rahmen der automatischen Sensordatenverarbeitung für einen autonomen mobilen Roboter*, Diplomarbeit im Fachbereich Informatik der Universität Kaiserslautern, 1990

[Horowitz78]: Horowitz, E.; Sahni, S.: *Fundamentals of Computer Algorithms*, Pitman Published LTD, 1978

[Iijima89]: Iijima, J.; Asaka, S.; Yuta, S.: *Searching unknown environment by a mobile robot using range sensor - an algorithm and experiment*, IEEE/RSJ International Workshop on intelligent Robots and Systems, Sept. 89, Seite 46 bis 53

[Jerkov87]: Jerkov, G.; Knieriemen, T.: *Autonome mobile Roboter - Einführung und Überblick*, Interner Bericht 171/87 im Fachbereich Informatik der Universität Kaiserslautern, 1987

[Knieriemen91]: Knieriemen, T.: *Autonome mobile Roboter*, B.I. Wissenschaftsverlag, 1991

[Lumelsky89]: Lumelsky, V.; Mukhopadhyay, S.; Sun, K.: *Sensor-based terrain acquisition: a "Seed spreader" Strategie*, IEEE/RSJ International Workshop on intelligent Robots and Systems, Sept. 89, Seite 62 bis 67

[Meier86]: Meier, A.: *Methoden der grafischen und geometrischen Datenverarbeitung*, B.G. Teubner, 1986

[Ramsbott91]: Ramsbott, T.: *Geometrische Freiraumzerlegung und flächendeckende Bahnplanung für einen autonomen mobilen Roboter*, Diplomarbeit im Fachbereich Informatik der Universität Kaiserslautern, 1991

[Richter89]: Richter, M.: *Prinzipien der künstlichen Intelligenz*, B.G. Teubner, 1989

[Ritter90]: Ritter, H.; Martinez, T.; Schulten, K.: *Neuronale Netze*, Addison Wesley, 1990

[Strauch91]: Strauch, T.: *Implementierung eines Navigationsalgorithmus in einer dynamischen Testumgebung*, Diplomarbeit im Fachbereich Informatik der Universität Kaiserslautern, 1991

[Wetzler91]: Wetzler, C.: *Entwicklung und Implementierung des multitaskingfähigen Echtzeit-Netzwerk-Betriebssystemkernes Albatross*, Diplomarbeit im Fachbereich Informatik der Universität Kaiserslautern, 1991

[Wilson90]: Wilson, D.; Rosenstein, L.; Shafer, D.: *Programming with MacApp*, Addison Wesley, 1990

Schlagwort-Index