Ein verhaltensorientierter Ansatz zum flächendeckenden Explorieren und Navigieren eines AMR

Dirk Müller

Juli 1998

Betreuung: Michael Kasper

Universität Kaiserslautern
Fachbereich Informatik
AG Professor von Puttkamer

Erwin-Schrödinger-Straße
Postfach 3049

D-67653 Kaiserslautern

Inhaltsverzeichnis

Kapitel 1

Einleitung

Servicerobotern, nach [Schraft 98] als "Leitprodukte der Zukunft" bezeichnet, wird in den kommenden Jahren ein großes Wachstumspotential vorausgesagt.

Der Einsatz autonomer mobiler Roboter wird in diesem Dienstleistungs- und Servicesektor eine Vielzahl von Zukunftsanwendungen erschließen, beispielsweise in Form von Transportaufgaben oder bei der Inspektion schwer zugänglicher oder gefährlicher Bereiche [Kasper 94].

Einen relativ großen Anteil an diesen, auf den Dienstleistungs- und Servicesektor ausgerichteten Robotern, werden dabei verschiedenste Reinigungsmaschinen einnehmen. Neben spezialisierten Robotern zur Fassaden- oder Fahrzeugreinigung kommt hier der Bodenreinigung eine Schlüsselrolle zu.

Während bei Transportaufgaben das Finden eines möglichst kurzen, kollisionsfreien Pfades im Vordergrund steht, muß bei der Bodenreinigung die gesamte Umgebung flächendeckend befahren werden.

In der Vergangenheit hat sich die Forschung auf diesem Gebiet fast ausschließlich auf das Erarbeiten von Lösungsansätzen zur Reinigung großer Verkehrsflächen wie Hallen, Flughafenterminals oder Bahnsteigen konzentriert. Im Laufe der Zeit hat sich dadurch die Entwicklung von geführten Maschinen – z.B. mit Hilfe von Induktionsschleifen – bis hin zu autonom arbeitenden Robotern fortgesetzt.

Seit einigen Jahren befinden sich (teil-)autonome Bodenreinigungsroboter für diese Bereiche im kommerziellen Einsatz [HAKO 94], [Kärcher 96], [KENT 98]. Erst in jüngster Zeit wurden auch Protoypen vorgestellt, die eine Reinigung in relativ engen, mit starkem Publikumsverkehr frequentierten Umgebungen, erlauben [Lawitzky et al. 98].

Dagegen existieren für den Heimbereich bisher nur wenige Prototypen, wie etwa die Roboter der Firmen Electrolux [Electrolux 97] oder Minolta.

Dies mag vor allem an den besonderen Anforderungen liegen: *Autonomous Home Cleaning Robots* müssen leicht und klein genug sein, um zwischen Stuhl- und Tischbeinen reinigen zu können. Sie benötigen darüberhinaus eine robuste und fehlertolerante Steuerung, welche auch mit dynamischen Objekten umgehen kann.

Um die Reinigungsaufgabe autonom durchführen zu können, wird eine flächendeckende Bahnplanung benötigt. Diese soll sicherstellen, daß jede von dem Roboter erreichbare Stelle von diesem mindestens einmal überfahren wird. Der Kurs soll dabei so gewählt werden, daß nicht zu viele Flächen mehrfach bearbeitet werden.

Die Steuerung soll also zielgerichtet und effizient arbeiten.

Beim Befahren dieser Bahn sollte der Roboter zudem erkennen, wann eine komplette Flächendeckung erreicht ist, damit er die Reinigung beenden und sich gegebenenfalls neuen Aufgaben widmen kann.

Für eine flächendeckende Bearbeitungsaufgabe benötigt der Roboter ein Umweltmodell und eine Möglichkeit seine eigene Position zu bestimmen.

Die kommerziellen, schon im Einsatz befindlichen Roboter erhalten in der Regel das Umweltmodell in Form von Karten, welche von außen vor Beginn der Fahrt vorgegeben werden. Aufgrund der Einsparung an Arbeitsaufwand bei den großen Flächen, die diese Maschinen regelmäßig bearbeiten, und der Tatsache, daß die Einsatzumgebung sich selten ändert, ist der Aufwand, der zur Erstellung und Eingabe der Karten notwendig ist, durchaus gerechtfertigt.

Dagegen erscheint für die Reinigung im Heimbereich der Aufwand einer a priori Kartenerstellung, sei es durch ein vom Benutzer vorgegebens CAD-Modell oder eine getrennte Explorationsfahrt vor der eigentlichen Reinigung, nicht gerechtfertigt. Dies begründet sich aus den, sich im Heimbereich häufig ändernden Einsatzumgebungen. Zum einen wird der gleiche Roboter an verschiedenen Orten eingesetzt, zum anderen werden Hindernisse wie Stühle, Tische oder ähnliches nicht immer an der gleichen Stelle zu finden sein.

Ein Verfahren, welches eine a priori Kartenerstellung voraussetzt, mindert die praktische Einsatzbarkeit eines Roboters drastisch.

Aus diesem Grund ist es für die Akzeptanz eines Bodenreinigungsroboters im Heimbereich wesentlich, daß der Roboter *parallel* zur Reinigungsaufgabe seine Umgebung exploriert und kartiert. Dabei ist keine oder nur eine sehr geringe Umweltveränderung zur Unterstützung der Positionsbestimmung zulässig.

Zudem ist die Akzeptanz abhängig von den Kosten eines solchen Systems. Deshalb sollte ein Verfahren zum flächendeckenden Fahren für den Einsatz im Heimbereich robust genug sein, um auch mit einfacher Hardware (preiswerte Sensoren, Mikro-Controller, etc.) zu funktionieren.

Der letzte Punkt stellt aus heutiger Sicht, neben der begrenzten Batterietechnologie, die größte Einschränkung dar. Die technologischen Anforderungen lassen sich mit entsprechender Ausstattung elegant lösen. Durch die finanziellen Restriktionen, die viel stärker als im kommerziellen Bereich durchschlagen, und den daraus resultierenden Beschränkungen in der Sensorik und der Rechenleistung, scheiden jedoch viele klassische Lösungsansätze von vornherein aus.

In der Literatur finden sich mehrere Verfahren, welche ein flächendeckendes Fahren realisieren. Als Beispiele, die auf einer vorher erstellten Karte aufbauen, sollen exemplarisch die Verfahren von Christian Hofner [Hofner 97] und Alexander Zelinski [Zelinski 94] erwähnt werden.

Bei dem Verfahren von Hofner erfolgt die Bahnplanung mit Hilfe von Bewegungsmakros, welche sich aus den Bahnelementen *Geradenstück* und *Kreisbogen* zusammensetzen. Sie geschieht durch Parametrisierung und Verkettung dieser Bewegungsmakros derart, daß eine Flächendeckung erreicht wird. Das Verfahren wird auf die Geometrie und Kinematik des Roboters angepaßt.

Zelinski stellt einen Ansatz vor, welcher zu Beginn ein Umweltmodell in Form einer Rasterkarte erhält. Das Verfahren breitet zunächst, von einem Zielpunkt ausgehend, eine Wellenfront aus, welche die einzelnen Rasterzellen des Freiraums mittels einer Kostenfunktion belegt. Diese berücksichtigt dabei die minimale Distanz

zum Ziel, sowie einen Sicherheitsabstand zu Hindernissen. Zur Kursplanung wird von der aktuellen Roboterposition ausgehend unter den noch nicht erfaßten Nachbarzellen immer diejenige ausgesucht, welche den höchsten Wert besitzt. Die Wegesuche endet, wenn der Zielpunkt erfaßt ist.

Ansätze zur flächendeckenden Bahnplanung ohne a priori Wissen der Umwelt, finden sich in der Literatur wenige. Als Beispiele seien [Burhanpurkar 94] und [Furuhashi et al. 92] genannt, wobei allerdings zum Teil die Algorithmen nicht veröffentlicht sind oder kein Nachweis der praktischen Einsetzbarkeit bekannt ist.

Die vorliegende Arbeit konzentriert sich auf den Entwurf der Steuerungskomponente eines autonomen Bodenreinigungsroboters, wobei insbesondere die für den Einsatz im Heimbereich genannten Anforderungen, wie die zur Bearbeitung parallele Erstellung eines Umweltmodells, Berücksichtigung finden.

Kapitel 2

Aufbau des simulierten AMR

Im folgenden wird beschrieben, welche Anforderungen die einzelnen Komponenten für den geplanten Einsatz erfüllen müssen und welche Auswirkungen diese für den simulierten AMR implizieren.

2.1 Geometrie und Einsatzumgebung

Als Reinigungsroboter wird der AMR sich in einer Indoor-Umgebung befinden, welche zunächst keine Stufen enthält. Treppenabgänge könnten später durch einen speziellen Sensor erkannt und ebenso wie Treppenaufgänge als Hindernisse betrachtet werden. Der Roboter bewegt sich somit in einer zweidimensionalen Ebene und muß keine Höhenunterschiede berücksichtigen. Die Simulation sieht Einsatzgebiete bis zu einer Größe von 20 x 20 Metern vor [1)].

Der Roboter soll sich innerhalb seines Einsatzgebietes möglichst frei bewegen können. Beim Reinigen in einem Wohnraum muß der Roboter vielen Hindernissen ausweichen und muß enge Durchfahrten passieren. Aus diesem Grund sollte er möglichst klein und kompakt sein. Die beste Manövrierbarkeit ergibt sich bei einem kreisförmigen Grundschnitt. Simuliert wird daher ein runder Roboter mit einem Durchmesser von 40 cm [1)].

2.2 Antriebssystem

Das Antriebssystem soll so konzipiert sein, daß der Roboter keine komplizierten Rangier- und Wendemanöver durchführen muß. Deshalb kommt eine Anordnung von zwei starren angetriebenen Rädern und einem freilaufenden Schlepprad, wie sie aus Abbildung 2.1 ersichtlich ist, zum Einsatz. Durch die runde Geometrie kann der Roboter auf der Stelle wenden ohne dabei an Hindernisse zu stoßen.

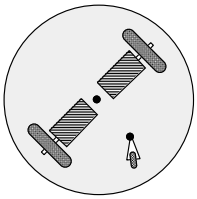


Abb. 2.1: Radanordnung

1) Diese Simulationsparameter lassen sich in einer Konfigurationsdatei ändern.

Bei einer solchen *differential drive* Anordnung ergibt sich die Kurvenfahrt aus den unterschiedlichen Geschwindigkeiten der einzelnen Räder. Zur Ansteuerung soll eine (v,ω)-Schnittstelle verwendet werden, die es ermöglicht, die Bewegung des Roboters durch Angabe der Lineargeschwindigkeit (v) und der Winkelgeschwindigkeit (ω) zu steuern. Die Lineargeschwindigkeit bezieht sich auf die Translation und die Winkelgeschwindigkeit auf die Rotation des Roboters pro Zeitintervall in einem definierten Punkt (dem kinematischen Zentrum). Der Zusammenhang zwischen (v, ω) und den einzelnen Radgeschwindigkeiten ergibt sich aus Gleichung (2.1). In der Simulation erfolgt die Angabe der Lineargeschwindigkeit in [cm/s] und die der Winkelgeschwindigkeit in [Grad/s].

$$v = \frac{v_{links} + v_{rechts}}{2} \qquad \omega = \frac{v_{rechts} - v_{links}}{d} \tag{2.1}$$

2.3 Sensorik

Sensoren liefern dem Roboter Informationen über interne Zustände (interne Sensoren) sowie über seine Umwelt (externe Sensoren). Da der Roboter kein Vorwissen z.B. in Form einer Karte über seine Einsatzumgebung erhält, sind sie die einzigen Informationsquellen zur Exploration seiner Umwelt.

Der Roboter benötigt im wesentlichen drei Arten von Sensoren:

- zur Erkennung von Hindernissen und Bestimmung deren Lage
- zur Erkennung von Kollissionen
- zur Bestimmung seiner Position

2.3.1 Hinderniserkennung

Sensoren zur Erkennung und Lagebestimmung von Hindernissen sind notwendig, damit der Roboter während der Fahrt eine Karte aufbauen und rechtzeitig auf Hindernisse reagieren kann.

In realen Systemen kommen hierzu berührungslose entfernungsmessende Sensoren zum Einsatz. Diese Sensoren basieren auf dem Prinzip von ausgesendeten elektromagnetischen Wellen oder Schallwellen, welche am Hindernis reflektiert werden und mittels eines Empfängers am Roboter wieder aufgenommen und vermessen werden. Das Schätzen der Entfernung geschieht meist indirekt über die Bestimmung von Hilfsgrößen wobei verschiedene Verfahren üblich sind:

- Triangulation
- Laufzeitmessung
- Phasenverschiebungsmessung

Bei der Entfernungsbestimmung mittels *Triangulation* errechnet sich die Entfernung aus dem Einfallswinkel der reflektierten Welle. Bei der *Laufzeitmessung* er-

gibt sich die Enfernung aus der Zeit, die zwischen Senden und Empfang der reflektierten Welle vergeht. Auch die Messung der *Phasenverschiebung* läßt sich zur Entfernungsbestimmung nutzen, in dem auf die ausgesandte Welle ein periodisches Signal aufmoduliert wird, so daß sich die Entfernung aus der Phasenverschiebung von dem ausgesandten und dem empfangenen Signal errechnen läßt.

Üblicherweise werden die Daten eines enfernungsmessenden Sensors zusammengefaßt und z.B. als Sectorscan der Kontrolleinheit zur Verfügung gestellt. Die einzelnen Werte eines Sectorscans geben die minimale Entfernung zu einem Hindernis innerhalb des jeweiligen Kreissektors an. (Abb. 2.2)

In der Simulationsumgebung findet ebenfalls ein Sectorscan Verwendung, wobei allerdings von der tatsächlichen technischen Realisierung abstrahiert wird und aus Performancegründen nur ein enger Strahl innerhalb des entsprechenden Sektors verfolgt und die Entfernung zum ersten Hindernis, welches der Strahl trifft, zurückgeliefert wird. (Abb. 2.3)

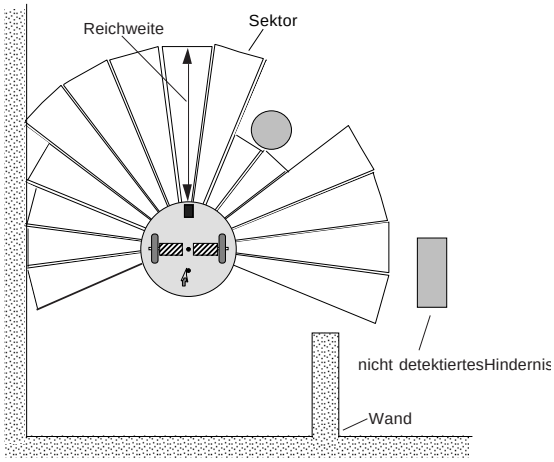


Abb. 2.2: Sector Scan

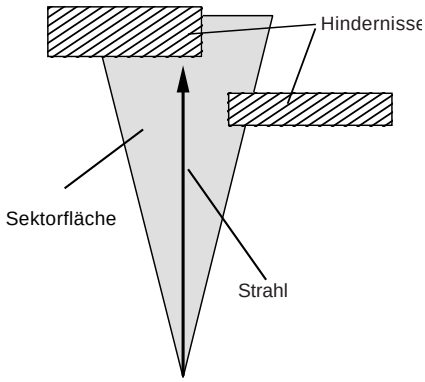


Abb.2.3: in Simulation

In der Simulation kann die maximale Reichweite des Sensors als Parameter eingestellt werden. Die benutzte maximale Entfernung beträgt 2m. Der simulierte Sensor erreicht dabei eine Entfernungsauflösung von 1cm.

2.3.2 Kollisionserkennung

Je nach ausgewähltem Verfahren der Hinderniserkennung gibt es immer wieder Hindernisse, die aufgrund ihres Materials oder ihrer Oberflächenbeschaffenheit (z.B. Glas, Styropor) für oben genannte Sensoren schlecht oder gar nicht erkennbar sind. Deshalb ist es unerläßlich, daß der Roboter erkennt, wenn er gegen Hindernisse stößt. Hierzu kommen taktile Sensoren (z. B. Bumper) zum Einsatz. Dies können am Umfang des Roboters verteilte Schalter sein, die bei Kontakt mit einem Hindernis auslösen, so daß der Roboter ein Signal erhält, falls er gegen ein Hindernis stößt.

2.3.3 Positionsbestimmung

Das flächendeckende Fahren erfordert zwangsläufig die Erstellung einer globalen Karte der Umwelt. Während zum Eintragen der abgefahrenen Flächen, es aufgrund der Kreisform des Roboters ausreicht die Position zu kennen, erfordert das Eintragen von Hindernissen in die Karte zusätzlich das Wissen um die Orientierung des Roboters. Die Sensordaten des Sectorscans bestimmen die Lage der Hindernisse nur relativ zu Position und Orientierung des Roboters.

Die Simulation geht von dem Vorhandensein eines wie auch immer gearteten Moduls zur Positionsbestimmung aus. Einige Ansätze zur tatsächlichen Realisierung sollen im folgenden genannt werden.

Eine Positionsbestimmung alleine durch Odometrie ist praktisch nicht möglich, da der Roboter relativ lange unterwegs ist und sich zu viele Meßfehler, die durch Schlupf oder andere Ungenauigkeiten entstehen können, sich akkumulieren und dies schnell zu großen Abweichungen führt.

Das Problem der Akkumulation der Fehler läßt sich vermeiden, wenn der Roboter seine Position anhand markanter Objekte in seiner Umgebung (natürliche oder künstliche Landmarken) kontrollieren kann.

Eine Möglichkeit zur Kontrolle der Position besteht in Verfahren, welche die Roboterposition in Bezug auf die Startposition anhand der anfallenden Sensordaten bestimmen. Die Arbeiten von [Weiß 94], [Lu und Milos 97] oder [Thrun et al. 98] zeigen Möglichkeiten auf, wie z.B durch Korrelation von Sensordaten die Roboterposition ausreichend genau bestimmt werden kann.

Das Aufstellen von Barken als künstliche Landmarken, die über spezielle Sensoren erkannt und deren Position bestimmt wird, bieten eine weitere Möglichkeit. Es erfordert allerdings Veränderungen der Umwelt.

Eine weitere Möglichkeit der Positionsbestimmung von außen ist über verschiedene Arten der Peilung denkbar.

Die Orientierung läßt sich entweder durch einen Kreiselkompaß bestimmen oder aus den Positionsdaten über mehrere Bewegungsschritte hinweg ermitteln.

2.4 Kontrollstruktur

Unter der Kontrollstruktur (control structure) versteht man das generelle Organisationsmuster der Datenverarbeitung in einem AMR [Puttkamer].

Der klassische Ansatz einer Kontrollstruktur wird als funktionale oder horizontale Kontrollstruktur bezeichnet. Bei diesem Ansatz werden die einzelnen Aufgaben in Teilaufgaben über mehrere Hierarchieebenen zerlegt, wobei die Sensordaten in den Ebenen von unten nach oben hin mehr verdichtet werden und das Abstraktionsniveau der Planung zunimmt.

In dieser Arbeit soll allerdings eine erweiterte verhaltensorientierte Kontrollstruktur zum Einsatz kommen. Die verhaltensorientierte Kontrollstruktur geht auf Arbeiten von Rodney A. Brooks aus dem Jahre 1986 zurück [Brooks 86]. Bei diesem Ansatz wird das Gesamtverhalten des Systems in einzelne Verhaltensschichten zerlegt. Jede dieser Verhaltensschichten verfolgt ein eigenes Ziel und hat prinzipiell Zugriff auf alle Sensordaten. Die einzelnen Verhaltensschichten konkurrieren untereinan-

der, so daß zu jeder Zeit von allen Verhaltensweisen ein Vorschlag zur Steuerung der Aktuatoren vorliegen kann.

Welche Verhaltensweise sich in einer bestimmten Situation tatsächlich auf die Aktuatoren auswirkt entscheidet eine Instanz, die als Verhaltensfusion (behavior fusion) oder Arbitrierung (arbiter) bezeichnet wird. Brooks schlägt die *subsumption architecture* vor, dabei sind Verhalten höherer Schichten in der Lage Verhalten niedriger Schichten zu unterdrücken. Einzelne Verhaltensweisen werden als endliche Automaten implementiert, und globale Daten, wie zum Beispiel ein Umweltmodell, sind nicht zugelassen.

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

Bei der erweiterten verhaltensorientierten Kontrollstruktur wird nicht der strikte Ansatz von Brooks verfolgt. Anstelle der Verhaltensfusion durch *subsumption*, steht hier eine allgemeine Arbitrierungseinheit (siehe Abb. 2.4). Auch werden globale Daten und insbesondere ein Umweltmodell zugelassen.

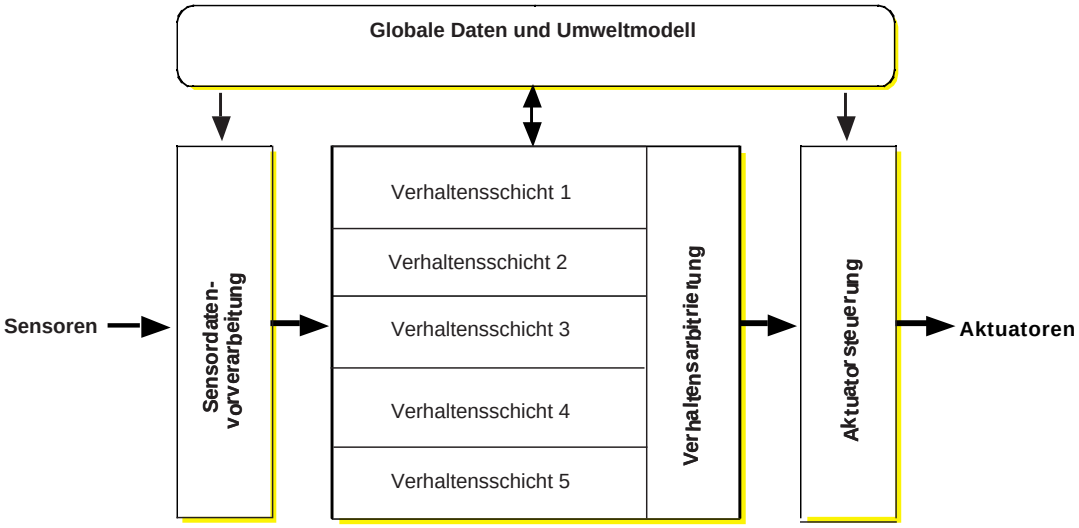


Abb. 2.4: erweiterte verhaltensorientierte Kontrollstruktur

Vorteile dieser Kontrollstruktur gegenüber dem klassischen funktionalen Ansatz liegen in der oben angesprochene Robustheit und dem geringeren Ressourcenbedarf. Weiterhin lassen sich die einzelnen Verhaltensweisen praktisch unabhängig voneinander implementieren und austesten.

Kapitel 3

Das implementierte Verfahren

Die Aufgabenstellung sieht die Entwicklung eines Verfahren vor, welches ohne a priori Kenntnis der Umwelt, eine flächendeckende Fahrweise ermöglicht. Während der Fahrt soll der Roboter eine Karte erstellen, in der er vermerkt, welche Bereiche bereits abgefahren sind.

Da diese Karte in der Regel noch unvollständig ist, soll zur Navigation nur lokal auf die Karte zurückgegriffen werden. Dies bedeutet, daß auf der Suche nach freien, noch nicht befahren Stellen nicht die ganze Karte durchsucht wird. Um dennoch zu gewährleisten, daß jeder mögliche Weg befahren und somit die ganze Fläche abgedeckt wird, empfiehlt sich ein Backtrackingalgorithmus.

Backtracking bedeutet, daß sich das Verfahren an jeder Stelle, an der sich der Weg aufgabelt, für einen Weg entscheidet und die Alternativen auf einem Stack abspeichert. Endet ein Weg, wird der letzte Eintrag vom Stack geholt und zu diesem zurückgekehrt. Wenn kein Eintrag mehr auf dem Stack vorhanden ist, sind alle möglichen Wege abgearbeitet und das Verfahren endet.

Die Idee des vorgestellten Verfahrens ist es nun, den Roboter die Fläche in einzelnen Bahnen abfahren zu lassen und noch abzufahrende Bereiche auf dem Stack festzuhalten.

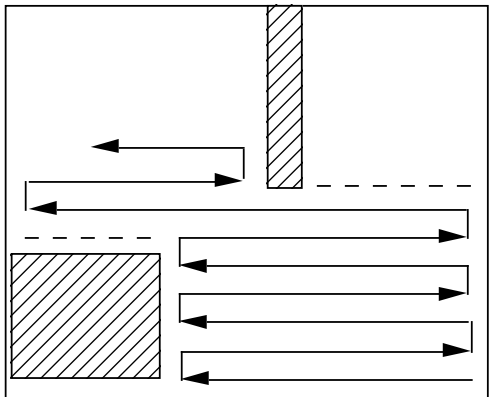


Abb. 3.1: Befahren der Fläche in einzelnen Bahnen

Während der Roboter sich auf einer solchen Bahn bewegt, untersucht er die be-

nachbarten Parallelbahnen auf ihre Befahrbarkeit hin. Jeder befahrbare aber noch nicht befahrene Streckenabschnitt der Parallelbahn wird auf dem Stack vermerkt.

Erreicht der Roboter eine Wand, und auf einer der beiden Seiten befindet sich noch nicht befahrener Freiraum, so führt er eine U-Wende durch und fährt in der entgegengesetzten Richtung zurück.

Ist es nicht möglich eine U-Wende durchzuführen, da keine freie Fläche in der Nähe ist, fährt er zurück zur letzten freien Fläche, deren Position auf dem Stack vermerkt wurde. Hat er diese Position erreicht, richtet er sich an der abgefahrenen Bahn aus und beginnt von neuem. Das Verfahren endet, wenn der Stack leer und somit die komplette Fläche abgefahren ist. Abbildung 3.1 verdeutlicht das Vorgehen, wobei die gestrichelte Linie, die auf dem Stack befindlichen Bahnen beschreiben.

Das Verfahren wird durch folgenden Algorithmus beschrieben :

1. fahre geradeaus, bis ein Hindernis im Weg ist und beobachte dabei die benachbarten Bahnen

2. wenn Weiterfahrt in verschiedene Richtungen möglich, dann
   - entscheide dich für eine Richtung
   - speichere Alternativen auf Stack

3. wenn keine Weiterfahrt mehr möglich, dann

   falls Stack leer ist, dann

   gehe zu 4.

   sonst
   - hole nächsten Eintrag vom Stack
   - fahre dorthin und beginne wieder mit 1.

4. ENDE - die komplette Fläche ist abgefahren

Dieses Schema zeigt die drei wichtigsten Verhaltensschichten, die für einen verhaltensorientierten Ansatz zum Lösen der Aufgabe notwendig sind:

- Geradeausfahren (*Drive*)

  der Roboter fährt geradeaus bzw. parallel zur letzten Spur

- Drehen an der Wand (*TurnAtWall*)

  wenn der Roboter an ein Hindernis herankommt, führt er eine U-Wende durch, damit er versetzt zur letzten Spur steht

– fahren zum nächsten Ziel (*DriveToNextArea*)

der Roboter plant einen Weg zum nächsten Ziel, fährt dorthin und richtet sich parallel zur letzten Spur aus

Das Verhalten *Drive* setzt sich wiederum aus drei einzelnen Verhaltensweisen zusammen. Dies sind:

– geradeaus fahren (*DriveStraight*)

– parallel zur eigenen Spur fahren (*FollowTrack*)

– an einer Wand entlang fahren (*FollowWall*)

Die Verhaltensweisen konkurrieren untereinander und bilden zusammen das Verhalten *Drive.*

Außer diesen, die Fahrt steuernden Verhalten werden noch folgende, planerische Komponenten benötigt:

– MapBuilder

– Observer

Die Aufgabe des *MapBuilders* ist die Erstellung der benötigten Karten. Dies ist zum einen eine Rasterkarte in hoher Auflösung, die dazu dient die abgefahrenen Flächen festzuhalten. Weiterhin werden die von den Sensoren erkannten Hindernisse in diese Karte eingetragen. Für die Karte ist eine hohe Auflösung notwendig, weil der Roboter mit ihrer Hilfe die Parallelfahrt zu den einzelnen Bahnen regeln muß. Bei zu grober Auflösung entstehen Lücken und dies müßte durch eine größere Überlappung der einzelnen Bahnen ausgeglichen werden.

Bei der zweiten Karte, für die der *MapBuilder* verantwortlich ist, handelt es sich um eine weitere Rasterkarte, die von dem Verhalten *DriveToNextArea*zur Navigation verwendet wird. Diese Karte stellt die befahrbare Fläche in gröberer Auflösung dar.

Die Aufgabe des *Observers* ist die Beobachtung der zur Fahrbahn benachbarten Parallelbahnen und die Verwaltung des Stacks.

Der *Observer* muß dabei entscheiden ob, sich neben der Fahrbahn noch Strecken befinden, die abzufahren sind. Die Anfangs- und Endpunkte dieser Strecken müssen auf dem Stack gesichert werden, damit sie später wieder angefahren werden können.

Die oben genannten Verhaltensweisen sind prinzipiell ausreichend um ein flächendeckendes Fahren zu realsieren. Da allerdings die Möglichkeit besteht, daß während der Bearbeitung einer Teilfläche an eine schon bearbeitete Teilfläche herangefahren wird, besteht die Gefahr, daß einzelne Abschnitte mehrfach bearbeitet werden.

Zum einen bricht das Verhalten *Drive* nur ab, wenn der Roboter sich in der Nähe eines Hindernisses befindet. Deshalb ist ein weiteres Verhalten *TurnAtTrack* not-

wendig, welches *Drive* unterdrückt falls der Roboter sich auf bearbeitetem Gebiet befindet.

Desweiteren können Teilflächen, die auf dem Stack liegen in der Zwischenzeit bereits abgearbeitet sein. Diese Ziele würden dann später unnötigerweise angefahren. Um dies zu verhindern muß nach jeder Bewegung die Gültigkeit aller Stackeinträge überprüft werden. Ungültige, schon erledigte Ziele müssen vom Stack gelöscht werden. Hierzu wird ein planerisches Verhalten *CheckStack* eingeführt, welches diese Aufgabe löst.

Bei dem besprochenen Verfahren werden alle Bahnen parallel zueinander gelegt. Dies kann unter bestimmten Umständen ungünstig sein. Deshalb wird ein weiteres Verhalten *FindBestDirection* definiert, welches für eine gegebene Bahn eine günstigere Richtung bestimmen kann.

Aus all diesen Verhaltensweisen setzt sich die gesamte Kontrollstruktur zusammen:

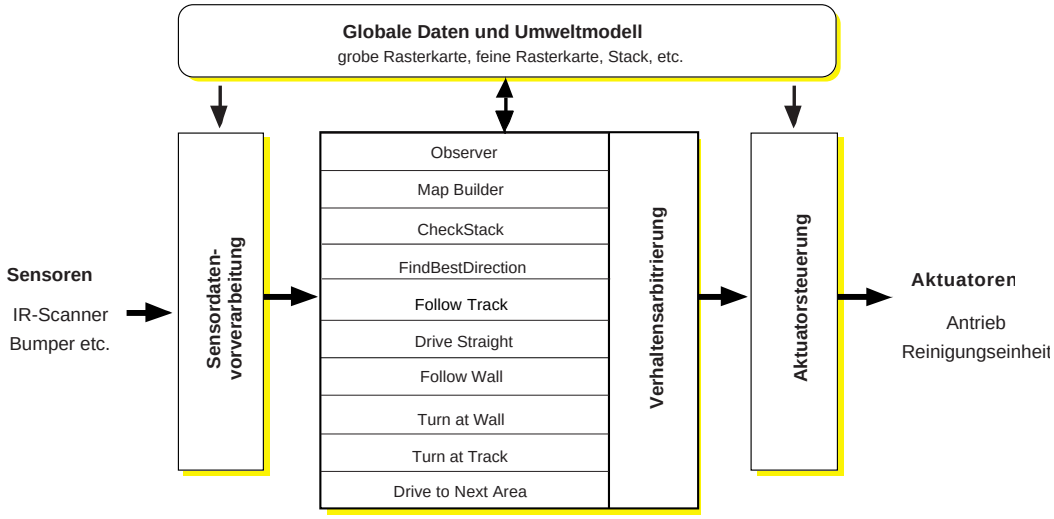


Abb. 3.2: Kontrollstruktur flächendeckendes Fahren

In den nun folgenden Abschnitten werden der Aufbau und die Funktion der einzelnen Verhaltensschichten sowie die grundlegenden globalen Daten näher erläutert.

3.1 Globale Datenstrukturen

Um ihre Funktion zu erfüllen benötigen die einzelnen Verhaltensweisen Zugriff auf verschiedene globale Daten (z.B. verschiedene Karten) sowie auf Statusinformationen untergeordneter Verhaltensschichten. Da die gleichen Daten von verschiedenen Verhaltensschichten benutzt werden, soll der Aufbau dieser Datenstrukturen hier kurz erläutert werden. Bedingungen, die einzelne Verhaltensschichten an die globalen Daten stellen werden in den entsprechenden Abschnitten später vertieft.

Zur Erstellung eines Umweltmodells dienen die beiden Rasterkarten

– *GridMap*
– *Navigational GridMap*

Gridmap

Die *GridMap* wird zum exakten Eintragen der abgefahrenen Fläche und der Lage von Hindernissen benutzt. Anhand dieser Karte wird die Fahrt auf Neuland, also noch nicht befahrenem Gebiet, geplant. Sie dient auch als Grundlage für das paßgenaue Abfahren benachbarter Bahnen. Damit keine Lücken zwischen den einzelnen Bahnen entstehen muß diese Karte eine hohe Auflösung besitzen. In der Simulation wurde hierzu eine Auflösung von 1 cm in jeder Richtung gewählt.

Jedes Feld der Rasterkarte kann einen von folgenden drei Werten annehmen:

1. Das Feld wurde nicht befahren
2. Das Feld wurde befahren
3. Das Feld liegt im Bereich eines Hindernisses und ist somit nicht befahrbar

Um diese drei Werte zu kodieren wären zwei Bit notwendig. Da es in der Simulation allerdings von Interesse ist, wie oft eine Stelle überfahren wurde, wird ein Byte für jedes Feld der Karte reserviert. Der Wert des Feldes gibt die Anzahl an, wie oft das Feld überfahren wurde. Ein Wert von 255 bleibt allerdings zum Eintragen von Hindernissen reserviert. Durch die Verwendung eines Bytes pro Rasterfeld erhöht sich in der Simulation außerdem die Zugriffsgeschwindigkeit auf die Karte. Dies ist hier besonders wichtig, da zum Zeichnen der Karte auf jedes einzelne Rasterfeld zugegriffen wird.

Die Karte selbst wurde als dynamisches Array in der Größe des abzufahrenden Gebietes angelegt. Bei einer maximalen Größe von 20m x 20m in einer Auflösung von 1 cm ergibt sich somit ein Speicherbedarf von etwa 4000 Kilobyte. Bei der Verwendung von nur zwei Bit würde entsprechend nur ein Viertel dieses Wertes benötigt.

Navigational GridMap

Die zweite Rasterkarte dient der Navigation innerhalb des schon befahrenen Gebietes und wird im folgenden auch als *Navigational GridMap* bezeichnet. Anhand dieser Karte wird die Punkt-zu-Punkt Fahrt des Verhaltens *DriveToNextArea* geplant.

DriveToNextArea verwendet zur Navigation ein Distance Transform Verfahren, das in Abschnitt 3.3 näher vorgestellt wird. Für dieses Verfahren wird eine Karte benötigt, welche alle vorherigen Roboterpositionen als zusammenhängende Fläche darstellt. In die Karte werden im Verlaufe des Verfahrens die Abstände jedes einzelnen Rasterfeldes zu dem Zielpunkt eingetragen. Zu diesem Zweck werden 2 Byte reserviert.

Um den Aufwand, den das DistanceTransform Verfahren benötigt in Grenzen zu halten, sollte die Auflösung der *Navigational GridMap* so grob wie möglich gehalten werden. Die obere Grenze wird allerdings durch den Radius des Roboters bestimmt. Eine genaue Abschätzung hierzu wird in Abschnitt 3.5 in Zusammenhang mit dem *MapBuilder* beschrieben.

In der Simulation wurde bei einem Radius des Roboters von 20 cm eine Auflösung der *Navigational GridMap* von 14 cm x 14 cm benutzt. Dies ergibt für den Speicherbedarf des dynamischen Arrays bei einer Feldgröße von zwei Byte und einer maximalen Szenariogröße von 20m x 20m lediglich etwa 40 Kilobyte.

PointStack

Als dritte wichtige globale Datenstruktur ist der Stack (*Pointstack*) zu nennen, auf den der Observer die Anfangs- und Endpunkte der noch abzufahrenden Strecken legt. Weiterhin enthält jeder Stackeintrag die Beschreibung der Geraden durch die beiden Punkte in Hessescher Normalenform. Die Verhaltensweisen *DriveToNextArea* und *TurnAtWall* können mittels der Funktion *GetNextLine* auf den Stack zugreifen.

Wenn ein Zielpunkt aufgrund temporärer Hindernisse nicht anzufahren ist, kann der entsprechende Stackeintrag mit Hilfe der Funktion *PutOnBottom* auf den Boden des Stacks zurückgelegt werden. Um mitzuzählen, wie oft dies geschehen ist, enthält jeder Stackeintrag einen Zähler.

Nextline

Die aktuelle Strecke, die vom Stack geholt wurde und welcher der Roboter folgen soll, wird in dem Record *Nextline* gespeichert. Nextline wird durch den Aufruf der Funktion *GetNextLine* belegt.

Die Richtung der Strecke wird beim Aufruf von *GetNextLine* derart bestimmt, daß derjenige Punkt als Startpunkt gewählt wird, der der aktuellen Roboterposition in Luftlinie am nächsten liegt. Dies führt zu einer optimalen Ausrichtung der Strecke, wenn sich ein Punkt in unmittelbarer Nähe des Roboters befindet. Bei entfernteren Punkten hat die Distanz in Luftlinie nichts mehr mit der tatsächlichen Entfernung, bei der auch Hindernisse berücksichtigt werden müssen, gemein. Für eine genaue Bestimmung der Entfernung müßte hier eine aufwendigere Bahnplanung zu den einzelnen Punkten, wie sie etwa *DriveToNextArea* benutzt, Verwendung finden. Da sich allerdings in den meisten Fällen die nächste abzufahrende Bahn in direkter Nachbarschaft zur aktuellen Bahn befindet, ist das angewandte Verfahren ausreichend.

Mit der Funktion *GetDistToNextline* kann der aktuelle Abstand des Roboters zur Geraden durch Anfangs- und Endpunkt dieser Strecke abgefragt werden.

GetSideToLine bestimmt, auf welcher Seite vom Startpunkt aus gesehen sich der Roboter zu *Nextline* befindet.

Statusinformationen

Da die zugrundeliegende Simulation eCat keine Threads für die einzelnen Verhaltensschichten zuläßt, ist eine echte Parallelität zwischen diesen nicht möglich. So werden zu jedem Simulationsschritt die einzelnen Verhalten nacheinander als Prozeduren aufgerufen. Damit die Zustände über die einzelnen Prozeduraufrufe hinweg erhalten bleiben, werden die Statusinformationen in einzelnen Records gespeichert.

Anstelle des Versendens von Nachrichten an andere Verhaltensschichten werden in den entsprechenden Statusrecords Flags gesetzt. Auf diese Flags können die anderen Verhaltensschichten bei Bedarf zugreifen.

3.2 Drive

Das Verhalten *Drive* soll eine möglichst geradlinige Fahrt an der eigenen Spur entlang bis zum nächsten Hindernis realisieren. Die Hauptrichtung wird dabei durch die in *NextLine* definierte Strecke bestimmt.

Generell kann Drive von dem Verhalten *TurnAtWall* unterdrückt werden, wenn sich der Roboter einem Hindernis nähert. Bei Hindernissen, die nur teilweise in die Bahn hineinragen erweist es sich allerdings als Nachteil, wenn der Roboter sofort umkehrt. Der Kontur des Hindernisses würde nicht gefolgt und somit die Fläche nur ungenügend abgefahren. Es ist eleganter, wenn der Roboter in Richtung der schon abgefahrenen Fläche ausweicht und das Hindernis so umfährt (*FollowWall*).

Um dies zu erreichen wird die Fahrt in zwei Phasen unterteilt. Während der ersten Phase, in der sich der Roboter zwischen Anfangs und Endpunkt der Strecke befindet, kann *Drive* nicht vom Verhalten *TurnAtWall* unterdrückt werden. Erst wenn der Endpunkt der Strecke passiert ist, wird ein Flag *passed* in den Statusinformationen von *Drive* gesetzt und *TurnAtWall* kann eine Drehung durchführen.

Solange der Roboter sich in der ersten Phase befindet, existiert eine Spur, welcher er folgen kann. Dies ergibt sich aus der Arbeitsweise des Observers, der nur die direkte Umgebung während der Fahrt untersucht und somit alle Strecken auf dem Stack in direkter Nachbarschaft zu schon befahrenen Bahnen stehen.

Wenn der Roboter über den Endpunkt der Strecke hinausfährt, existiert unter Umständen keine benachbarte Bahn, deren Spur er folgen könnte. Dann muß auf ein neues Verhalten *DriveStraight* umgeschaltet werden, daß ihn nur noch der Hauptrichtung folgen läßt.

Abbildung 3.3 zeigt eine solche Situation. Die gestrichelte Linie repräsentiert dabei die durch *Nextline* definierte Strecke.

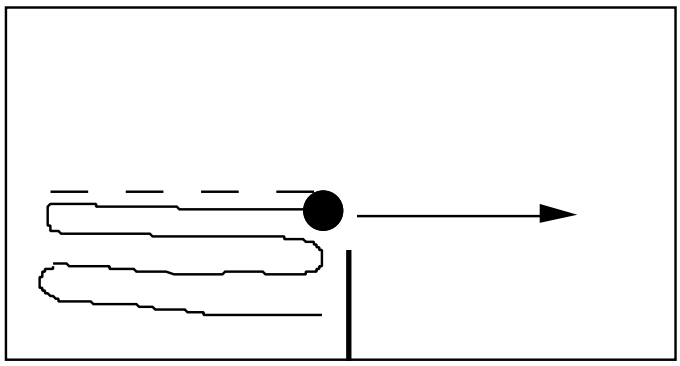


Abb. 3.3

Es ergeben sich somit für beide Phasen folgende Aufgaben, die von Drive gelöst werden müssen:

Phase 1 : (Endpunkt der Strecke wurde noch nicht passiert)
- FollowTrack
- FollowWall, wenn ein Hindernis in der Nähe ist

Phase 2: (Endpunkt der Strecke wurde schon passiert)
- DriveStraight

Das Verhalten *Drive* bricht ab, wenn in Phase 2 ein Hindernis auftaucht, oder der Roboter sich um mehr als den doppelten Radius von der Geraden entfernt hat.

Die drei untergeordneten Verhaltensschichten, die diese Aufgaben erledigen, werden im folgenden näher erläutert.

3.2.1 DriveStraight

Dies ist die einfachste der drei Verhaltensschichten: Der Roboter bewegt sich entlang einer Geraden. Dazu wird in großer Entfernung ein Zielpunkt auf der Geraden benutzt, auf den sich der Roboter während der Fahrt ausrichtet. Dieses Verhalten greift, sofern *Drive* aktiv ist und der Endpunkt von *Nextline* passiert wurde.

3.2.2 FollowTrack

Diese Verhaltensschicht hat zwei Ziele zu verfolgen. Zum einen die Einhaltung der Hauptrichtung und zum anderen das Folgen der Spur. Das Einhalten der Hauptrichtung ist wichtig, weil der Roboter sich über die Strecke hinweg auf einer möglichst geraden Bahn bewegen soll. Zu viele Kurven beeinträchtigen die korrekte Arbeitsweise des Observers und Verhindern gleichzeitig ein zügiges Vorankommen.

Die Steuerung zum Entlangfahren an der eigenen Spur wird mit Hilfe eines P-Reglers realisiert. Hierzu wird der Abstand zur letzten Spur bestimmt und anhand dessen wird die ω-Komponente der Bewegungsschnittstelle gesetzt.

Um den Abstand zur Spur zu bestimmen wird die *GridMap* benutzt. Vor dem Roboter wird hierzu eine zur Fahrtrichtung senkrecht verlaufende Linie verfolgt um die Kante zwischen befahrenem und nicht befahrenem Gebiet zu finden.

Der Sollwert ist erreicht, wenn diese Kante sich im Abstand r, dem Radius des Roboters, befindet. Entsprechend der Abweichung vom Sollwert r wird der ω-Wert geregelt, um eine Korrektur nach rechts oder links zu erreichen (Abb. 3.4).

Durch diese Art der Regelung entsteht eine Fahrt in leichten Schlangenlinien. Damit sich dies nicht über mehrere Bahnen hinweg verstärkt, wird die Regelung nach einer Seite hin begrenzt. Das heißt der Roboter darf die Gerade, welche die Haupt-

fahrtrichtung bestimmt, nur zu der Seite hin verlassen, in der sich schon befahrenes Gebiet befindet. Wird die Gerade auf die andere Seite hin überschritten, dann werden alle Regelungen durch die sich der Roboter noch weiter von der Geraden entfernt ignoriert.

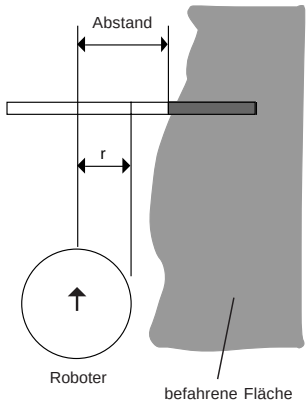


Abb. 3.4: Regelung bei FollowTrack

3.2.3 FollowWall

Die Verhaltensschicht *FollowWall* bleibt so lange inaktiv, wie kein Hindernis in der Nähe vor dem Roboter erkannt wird. Wird ein Hindernis erkannt, schaltet *FollowWall* durch und übernimmt die alleinige Kontrolle innerhalb von *Drive*.

Das Ausweichen an einem Hindernis erfolgt in drei Schritten (vgl. Abb. 3.5):

1. Drehen um einen festen Winkel
2. Folgen der Wand
3. Ausrichten in Richtung von *Nextline*

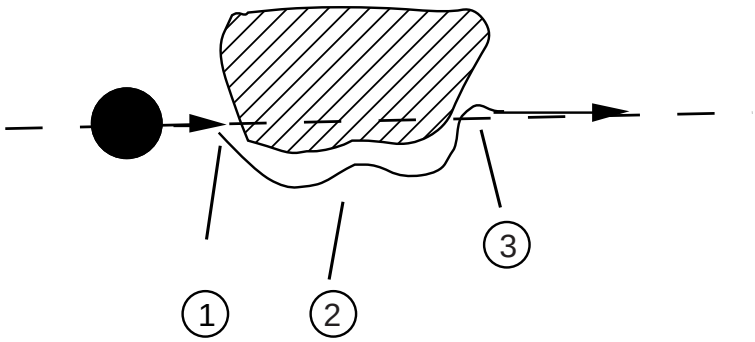


Abb. 3.5: FollowWall

Da *FollowWall* in Richtung der Seite, auf der sich die schon befahrenen Fläche be-

findet dem Hindernis folgen soll, muß sich das Hindernis auf der gegenüberliegenden Seite befinden. Dies wird mit einer Drehung um 60° in die entsprechende Richtung erreicht.

Im zweiten Schritt folgt der Roboter der Kontur des Hindernisses, wobei darauf geachtet wird, daß der Mindestabstand zur Wand eingehalten wird. Die Einhaltung des Mindestabstandes erfolgt mit Hilfe eines weiteren P-Reglers.

Der zweite Schritt wird solange ausgeführt, bis der Roboter *Nextline* kreuzt. Ist dies der Fall, dann ist das Hindernis komplett umfahren und *FollowTrack* darf wieder die Bewegung bestimmen.

Bevor die Kontrolle an *FollowTrack* zurückgegeben werden kann, muß der Roboter noch in Richtung der Geraden ausgerichtet werden. Dies geschieht im dritten Schritt.

Falls das Hindernis nicht komplett umfahren werden kann und der Roboter zu weit von der Geraden abweicht, wird die entsprechende Abbruchbedingung von *Drive* erreicht und somit auch *FollowWall* abgebrochen.

Da beim Ausweichen an einem Hindernis die Hauptrichtung verlassen wird, muß der Observer deaktiviert werden. Dadurch werden die letzten beobachteten Streckenabschnitte auf dem Stack gesichert. Während der Fahrt am Hindernis vorbei kann nicht auf Flächen gestoßen werden, die auf dem Stack gesichert werden müssen. Deshalb stellt es kein Problem dar, wenn der Observer erst nach Beendigung von *FollowWall* wieder aktiviert wird.

3.3 DriveToNextArea

Wenn ein Teilbereich komplett abgefahren ist und es keine direkte Möglichkeit zur Weiterfahrt mehr gibt, muß ein neues Ziel, das auf dem Stack liegt, angefahren werden. Diese Aufgabe übernimmt *DriveToNextArea*.

Um das neue Ziel zu bestimmen wird zunächst *GetNextLine* aufgerufen. Diese Funktion nimmt das letzte Ziel vom Stack und bestimmt die abzufahrende Linie *Nextline.* Da dieses Ziel im allgemeinen nicht direkt angefahren werden kann, erfolgt anschließend eine Bahnplanung. Dabei soll ein Navigationsverfahren angewendet werden, welches auf das während der bisherigen Fahrt erlangte Wissen (in Form von Karten) zurückgreift und daraus den kürzesten Weg bestimmt.

Da die verwendeten Karten in der Regel noch unvollständig sind, soll die Planung des Weges nur auf schon befahrenem Gebiet erfolgen. Selbst wenn der Zielpunkt im sichtbaren Bereich des Roboters liegt, soll kein bisher unbearbeitetes Gebiet überfahren werden. Man nimmt dabei zwar Umwege in Kauf, stellt dafür allerdings sicher, daß noch abzuarbeitende Bereiche nicht unnötigerweise in kleinere Abschnitte unterteilt werden.

Allgemein gilt:

Solange die Umgebung nicht durch dynamische Hindernisse verändert wird, existiert mindestens ein Weg auf befahrenem Gebiet von der aktuellen Roboterposition zum Zielpunkt, da der Roboter auf dem Weg zu seiner jetzigen Position den

Zielpunkt schon passiert hat.

Für die Planung eines Weges von Punkt zu Punkt anhand von Karten existieren verschiedene Verfahren. Da hier ein Umweltmodell in Form einer Rasterkarte vorliegt empfiehlt sich ein *Distance Transform* Verfahren, welches auf Rasterkarten basiert. Es handelt sich hierbei um eine vereinfachte Potentialfeldmethode.

Potentialfeldmethoden basieren auf dem Prinzip, daß der Roboter auf seinem Weg von einem Potentialfeld um das Ziel angezogen und von umgekehrten Potentialfeldern um die Hindernisse abgestoßen wird.

Ein Problem beim Einsatz von Potentialfeldern ist, daß durch die Wechselwirkung von anziehenden und abstoßenden Potentialfeldern der Roboter in ein lokales Minimum geraten kann und so sein Ziel nicht erreicht.

Bei dem verwendeten Verfahren wird nur das anziehende Potential um das Ziel herum betrachtet; abstoßende Potentiale von Hindernissen werden nicht berücksichtigt. Dadurch entstehen zwar keine lokalen Minima, aber es besteht prinzipiell die Gefahr, daß der Roboter auf seinem Weg zu nahe an ein Hindernis heranfährt und mit diesem kollidiert, falls kein gesondertes Verhalten zur Kollisionsvermeidung zum Einsatz kommt.

Als Potentialfeld wird bei dem *Distance Transform* Verfahren der jeweilige kürzeste Abstand zum Zielpunkt verwendet. Der Roboter bewegt sich innerhalb des Rasters auf dem Weg des steilsten Abstiegs (*steepest descent*), d. h. er bewegt sich immer zu der Rasterzelle, die dem Ziel am nächsten liegt und findet somit den kürzesten Weg zum Ziel.

Zum Eintragen der Entfernungen in die Rasterfelder wird dabei in umgekehrter Richtung vom Ziel- zum Startpunkt hin vorgegangen. Jedes Rasterfeld erhält den Wert desjenigen benachbarten Feldes mit dem niedrigsten Wert zuzüglich dem Abstand zu diesem Feld. Dabei breitet sich eine Wellenfront um das Ziel über alle freien Rasterfelder aus. Diese Wellenfront fließt um sämtliche Hindernisse herum bis der Startpunkt erreicht oder der gesamte Freiraum abgedeckt ist.

Der nachfolgende Algorithmus veranschaulicht diese Vorgehensweise.

Algorithmus **SetDistancesInGrid**

Als Startkonfiguration erhält der Algorithmus eine Rasterkarte **R** mit den Werten,

explored : für freie und schon bearbeitete Zellen
unexplored : für Hindernisse oder nicht bearbeitetes Gebiet

R[p] sei der Wert der Rasterkarte an Punkt p

Weiterhin werden zwei zunächst leere Listen **L1** und **L2** von Rasterpunkten benötigt.

Sei **Z** der Zielpunkt, **S** der Startpunkt

Setze alle freien Zellen auf Max

R[Z] := 0;

Abstand := 1;

```
Für jeden Nachbarpunkt p von Z mit R[p] <> unexplored
    füge p in Liste L1 ein
    R[p] := Abstand
Solange (L1 nicht leer) und (R[S] = Max)
    Abstand := Abstand +1
    Für jeden Punkt p aus L1
        Für jeden Nachbarpunkt q von p mit (R[q]<>unexplored)
        und (R[q]>Abstand)
            füge q in Liste L2 ein
            R[q] := Abstand;
    Leere L1
    Vertausche L1 und L2
```

Durch diesen Algorithmus wird die Rasterkarte so ausgefüllt, daß an jedem Punkt die Länge des kürzesten Weges zum Zielpunkt eingetragen ist. Dabei definiert sich die Länge aus der Zahl der Rasterfelder, die bei diesem Weg überschritten werden.

Abbildung 3.6 zeigt ein Beispiel einer solchen Karte, wobei die grau hinterlegten Felder einen Weg zum Ziel nach steepest descent anzeigen.

			4	3	2	2	2	2	2
			4	3	2	1	1	1	2
7	**S**	5	4	3		1	**Z**	1	2
7	6	5	4	4		1	1	1	2
7			5	5				2	2
8			6	6	5	4	3	3	3

Abb. 3.6: Beispiel von Distance Transform

Das Beispiel zeigt gleichzeitig ein Problem des Algorithmus auf. Innerhalb der erzeugten Rasterkarte führen verschiedene Wege derselben Länge zum Ziel, obwohl diese eine unterschiedliche reale Länge besitzen. Dies beruht darauf, daß im gezeigten Algorithmus alle Nachbarn eines Feldes mit dem gleichen Abstand versehen wurden. Dadurch erhält ein Weg über die Diagonale den gleichen Abstand, wie zum Beispiel der Weg in der horizontalen. Bei der Suche des kürzesten Weges nach

steepest descent muß deshalb, wenn mehrere Nachbarfelder den gleichen Abstand zum Ziel aufweisen, dasjenige ausgewählt werden, welches nicht über eine Diagonale verläuft.

In der Abbildung läßt sich erkennen (beim Übergang von 3 nach 2), daß der Roboter zu dicht an das Hindernis heranfährt. In diesem Fall könnte es zur Kollision kommen. Dadurch, daß in dem von *DriveToNextArea* verwendeten Verfahren die Planung nur auf dem Gebiet erfolgt, welches der Roboter schon befahren hat, kann er durch *Distance Transform* nur dann zu nahe an ein Hindernis heranfahren, wenn dies zuvor schon geschehen ist. Um diesem Fall vorzubeugen reicht ein lokales Ausweichmanöver aus.

Um die Navigation mittels *Distance Transform* durchzuführen, benötigt das Verhalten *DriveToNext* die *Navigational GridMap,* in welcher der Observer das überfahrene Gebiet markiert hat. Da der Startpunkt von *Nextline* in noch nicht befahrenem Gebiet liegt und somit eine Planung zu diesem Rasterpunkt nicht möglich ist, wird zunächst als Zielpunkt der benachbarte schon befahrene Rasterpunkt gewählt.

Für die anschließende Fahrt zum eigentlichen Zielpunkt ist keine Planung mehr erforderlich, da dieser in direkter, erreichbarer Nähe liegt.

Das Verhalten *DriveToNextArea* gliedert sich somit in folgende Schritte

1. Neues Ziel von Stack holen mittels *GetNextLine*
2. Entfernungen zum Zwischenziel in *Navigational GridMap* eintragen
3. Mittels *steepest descent* dem ermittelten Weg folgen
4. Geradeausfahrt vom Zwischenziel zum Startpunkt von *NextLine*
5. In Richtung der neuen Bahn ausrichten

Zunächst setzt diese Art der Navigation voraus, daß sich die Umwelt nicht verändert. Wird dem Roboter durch ein plötzlich auftauchendes Hindernis der Weg blockiert, wird er sein Ziel nicht mehr erreichen.

Um einer sich veränderten Umwelt mit dynamischen oder temporären Hindernissen gerecht zu werden wurde das Verhalten folgendermaßen erweitert.

Während der Fahrt zum Ziel werden die Sensordaten mit der *Navigational GridMap* verglichen. Wird dabei ein Hindernis erkannt, daß sich an einer schon befahrenen Stelle befindet, so werden die entsprechenden Bereiche in der Rasterkarte als temporäre Hindernisse markiert. Daraufhin erfolgt eine Neuplanung des Weges (Schritt 2).

Durch das Vorkommen von temporären Hindernissen ist es möglich, daß kein Weg mehr zum Ziel existiert. In diesem Fall wird das Ziel ganz unten auf den Stack zurück gelegt und versucht, das nächste Ziel vom Stack anzufahren.

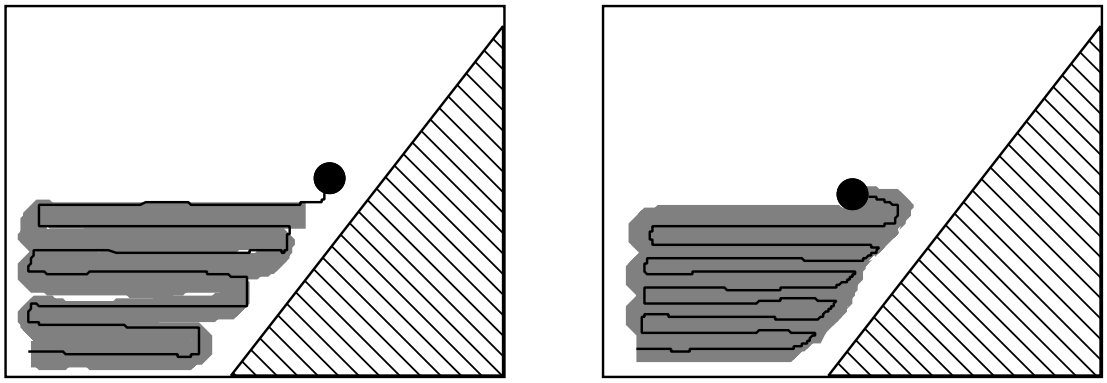
Da das erste Ziel sich nun unten auf dem Stack befindet, wird zu einem späteren Zeitpunkt wieder versucht dieses anzufahren. Es besteht schließlich die Möglichkeit, daß der Weg dann wieder frei ist. Damit das Verfahren nicht endlos läuft wird die Zahl, wie oft versucht wird dasselbe Ziel anzufahren, begrenzt. Wird die maximale Zahl von fünf Versuchen überschritten, dann wird das Ziel vom Stack gelöscht.

Hat der Roboter sein Ziel erreicht, werden alle temporären Hindernisse aus der *Navigational GridMap* gelöscht und wieder als schon befahren markiert.

3.4 TurnAtWall

Bei Erreichen einer Wand soll der Roboter eine Drehung durchführen und zum Startpunkt der benachbarten Bahn fahren. Im Gegensatz zu dem allgemeineren Verhalten *DriveToNextArea*, welches eine Fahrt zu einem beliebigen Zielpunkt plant, soll beim Verhalten *TurnAtWall* beim Anfahren der nächsten Bahn der Kontur des Hindernisses gefolgt werden. Dies ist nur dann möglich, wenn sich die nächste Bahn in direkter Nachbarschaft befindet. Eine aufwendige Planung zur Navigation, wie sie *DriveToNextArea* vorsieht, ist hier nicht notwendig.

Die Bedeutung des Abfahrens der Kontur wird in folgender Abbildung veranschaulicht. Abbildung 3.7a zeigt dabei eine starre U-Wende, wie sie in etwa beim ausschließlichen Einsatz von *DriveToNextArea* entsteht; Abbildung 3.7b zeigt den Einsatz von *TurnAtWall* .



a) mit DriveToNextArea b) mit TurnAtWall

Abb. 3.7: Drehen an einer Wand

Man erkennt, daß bei dem ersten Verfahren ein "Treppeneffekt" entsteht. Die Dreiecksflächen in der Nähe des Hindernisses bleiben entweder unbefahren oder müßten in einem späteren Schritt abgefahren werden. Durch Einsatz von *TurnAtWall* geschieht dies in einem Durchgang und das flächendeckende Fahren wird effizienter.

TurnAtWall übernimmt die Kontrolle genau dann, wenn sich ein Hindernis im Weg befindet und eine noch nicht befahrene Nachbarbahn existiert. Das Verhalten untergliedert sich in folgende Schritte.

1. Drehen in Richtung des Startpunktes der nächsten Bahn
2. Folgen der Kontur des Hindernisses, bis neue Bahn erreicht ist
3. Drehen in Richtung der abzufahrenden Bahn

Der erste Schritt ist erforderlich, damit der Roboter grob in die korrekte Richtung ausgerichtet ist und somit im zweiten Schritt nur noch Korrekturen zum Einhalten des Mindestabstandes zur Wand nötig sind.

Im zweiten Schritt kommt ein P-Regler zum Einsatz, der den korrekten Abstand zur Wand regelt. Dabei entsteht allerdings folgendes Problem: Wenn die Wand endet, bevor der Roboter die neue Bahn erreicht hat, besteht die Gefahr, daß der Roboter der Wand auf der anderen Seite weiter folgt und sich somit immer mehr von der neuen Bahn entfernt.

Um dies zu verhindern darf der Regler auf abrupte Richtungsänderungen der Wand nicht reagieren. Überschreitet der Unterschied zwischen gemessenem Abstand und Sollabstand einen Maximalwert, so fährt der Roboter nur noch geradeaus weiter. Starke Richtungsänderungen werden nur durchgeführt, sofern sich der Roboter dabei von der Wand entfernt, aber nie zur Wand hin.

Erreicht der Roboter die neue Bahn, muß er sich noch in die Richtung dieser Bahn drehen, damit die Anfangsbedingungen für das nachfolgende Verhalten *Drive* gegeben sind.

Das Verhalten *TurnAtWall* darf nur zum Einsatz kommen, wenn direkt rechts oder links von der aktuellen Bahn weitergefahren werden kann. Um dies festzustellen wird auf den *Observer*, der genau dies überwacht, zurückgegriffen. Solange der *Observer* sich im Zustand *FindEnd* befindet, muß eine benachbarte Bahn bearbeitet werden.

3.5 MapBuilder

Die Aufgabe des MapBuilders ist die Erstellung der *GridMap* und *Navigational GridMap* anhand der von den Sensoren gelieferten Daten.

Zur Erstellung der *GridMap* muß zunächst die abgefahrene Fläche eingetragen werden. Hierzu wird in definierten Zeitabständen die aktuelle Position des Roboters bestimmt. Sind diese Intervalle klein genug gewählt, so daß der Roboter sich nur minimal innerhalb dieser Zeit bewegt hat, so genügt das Eintragen der Grundfläche des Roboters an der aktuellen Position in die Karte. Können die Intervalle nicht klein genug gewählt werden, da zum Beispiel die genaue Position nicht schnell genug ermittelt werden kann, müssen die Roboterpositionen zwischen den einzelnen Schritten interpoliert werden, damit keine Lücken in der Karte entstehen.

In der Simulation wurde das Zeitintervall zur Aktualisierung der Karte so gewählt, daß es einem Simulationsschritt entspricht. Damit wird die Karte 10 mal pro Sekunde aktualisiert. Dies erweist sich bei einer maximalen Bahngeschwindigkeit von 1 m/s als ausreichend.

Im Gegensatz zu einer späteren Realisierung des Verfahrens soll in der Simulation die Karte gleichzeitig Aufschluß darüber geben, wie oft eine Stelle überfahren wurde. Während es später genügt jede besuchte Stelle als befahren zu markieren, muß in der Simulation mit jedem neuen Besuch ein Zähler erhöht werden.

Da sich der Roboter in einem Schritt nur minimal von der letzten Position wegbewegt, ist der Schnitt aus der im letzten Schritt abgedeckten und der neuen Fläche nicht leer. Damit der Zähler nur an den neu überdeckten Flächen erhöht wird, darf im nächsten Schritt nicht die komplette Grundfläche des Roboters in der Karte aktualisiert werden. Die zu aktualisierende Fläche ergibt sich aus der Fläche an der neuen Position ohne die an der letzten Position abgedeckte Fläche. Siehe Abbildung 3.8.

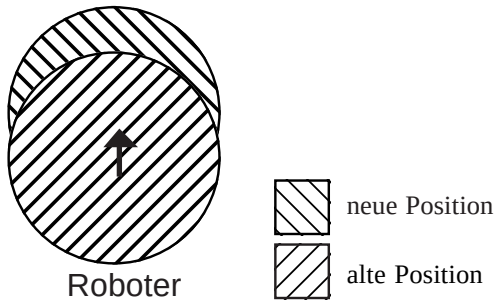


Abb. 3.8

Die Zweite Aufgabe bei der Erstellung der *GridMap* ist das Eintragen von Hindernissen in diese Karte. Als Grundlage hierfür dienen die Informationen des Sectorscans, der eventuell vorhandene Hindernisse in ihrer relativen Lage zur Position und Orientierung des Roboters beschreibt. Um anhand dieser Daten die Karte zu aktualisieren, müssen zunächst aus den lokalen Polarkoordinaten der Hindernisse die globalen Koordinaten bestimmt werden. Danach werden die entsprechenden

Bereiche in der Karte als Hindernisse markiert.

Aufgrund der Tatsache, daß aus Performancegründen der Sectorscan bei der Simulation nur einzelne Strahlen verfolgt (siehe Abschnitt 2.3.1 Hinderniserkennung), werden nur einzelne Punkte der Hindernisse zurückgeliefert. Um dennoch in der Karte einen geschlossenen Bereich für die einzelnen Hindernisse zu erhalten, wird folgende Methode verwendet.

Für je zwei erkannte Hindernispunkte eines Sectorscans wird untersucht, wie weit diese Punkte voneinander entfernt liegen. Liegen zwei Punkte näher als der Durchmesser des Roboters zuzüglich des einzuhaltenden Sicherheitsabstandes beisammen, so wird in die Karte eine Linie eingetragen, welche beide Punkte verbindet. Dies ist sinnvoll, weil aufgrund des geringen Abstandes eine Durchfahrt zwischen den beiden Punkten nicht möglich ist.

Ein von den anderen Hindernispunkten isolierter Punkt wird auf diese Weise in der Karte nicht berücksichtigt. Da in großer Entfernung vom Roboter die im Sectorscan verfolgten Strahlen weit voneinander abweichen, werden Hindernisse in großer Entfernung somit noch nicht in die Karte eingetragen. Dies ist allerdings nicht von Nachteil, da die Genauigkeit der Sensoren mit zunehmender Entfernung in der Regel abnimmt.

Ein Nachteil des obigen Verfahrens besteht darin, daß jedes von den Sensoren erkannte Hindernis fest in die Karte eingetragen wird. Dabei wird davon ausgegangen, daß jedes erkannte Hindernis gültig ist. Von dieser Annahme kann bei einem realen System nicht ausgegangen werden, da Fehlmessungen die Regel sind.

Auch in einer Umgebung mit dynamischen Hindernissen ist diese Methode nicht einsetzbar, da Freiflächen, die durch das Wegbewegen eines dynamischen Hindernisses entstehen in der Karte nicht mehr freigegeben werden.

Beide Probleme lassen sich durch ein probabilistisches Verfahren zur Umweltmodellierung beheben, wie es z.B. von Alberto Elfes vorgeschlagen und von anderen erweitert und ergänzt wurde ([Elfes 89],[Jörg 94],[Thrun et al. 98]).

Bei diesen Verfahren handelt es sich um ein statistisches Modell, welches die Korrektheit der ermittelden Meßdaten durch Überlagerung mehrerer Messungen überprüft. Jedem Feld der Rasterkarte *(occupancy grid*) wird ein Wert zugeordnet, der die Wahrscheinlichkeit angibt, daß das betreffende Feld belegt ist. Der Wert 0 entspricht dabei einem freien Feld, 1 einem belegten Feld und 0.5 deutet auf einen unbekannten Zustand. Zunächst sind alle Felder der Rasterkarte als unbekannt markiert. Mit jeder Messung werden alle Rasterfelder, die im Bereich des gemessenen Sektors liegen aktualisiert. Dabei wird die Wahrscheinlichkeit, die sich aus der aktuellen Meßung ergibt, verrechnet mit der Belegung der *occupancy grid*, welche die Wahrscheinlichkeit unter Berücksichtigung aller vorherigen Messungen beinhaltet.

Da falsch belegte Rasterfelder auf diese Art und Weise korrigiert werden können, stellt das Verfahren von Elfes eine Verbesserung dar, die den Einsatz in einem realen System möglich macht. Die Implementierung dieses Verfahrens soll allerdings als Erweiterung für später vorbehalten bleiben, da sie mit zusätzlichem Aufwand an Speicherplatz und Rechenzeit verbunden ist.

Zunächst ist mit den gegebenen Einschränkungen das in der Simulation implementierte Verfahren ausreichend um die prinzipielle Funktionsweise und Einsetzbarkeit der vorgestellten Methode zum flächendeckenden Fahren zu untersuchen.

Neben der Erstellung der *GridMap* ist der *MapBuilder* noch mit der Aufgabe der Erstellung der *Navigational GridMap* betraut. Sie soll die abgefahrene Fläche mit geringerer Auflösung als in der *GridMap* darstellen.

Damit die Erstellung der *Navigational GridMap* unabhängig von der Implementierung der zweiten Rasterkarte bleibt, wird sie während der Fahrt aus den einzelnen Roboterpositionen generiert.

Hierzu wird zunächst jede Roboterposition auf das Raster der *Navigational GridMap* abgebildet und als befahren markiert. Dabei muß die Auflösung der Karte so gewählt werden, daß bei der Abbildung einer Roboterposition in das Raster, diese nicht auf die falsche Seite eines Hindernisses abgebildet wird. Abbildung 3.9 zeigt ein Beispiel mit einem zu groben Raster. Der Pfeil demonstriert dabei die Abbildung der Roboterposition auf die Mitte des Rasters.

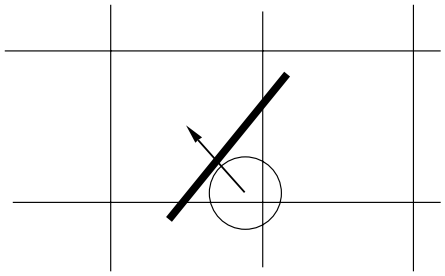


Abb. 3.9

Damit eine korrekte Abbildung gewährleistet ist, ergibt sich somit für die Größe des Rasters mit der Seitenlänge *d* in Abhängigkeit zum Radius des Roboters folgende Beziehung:

$$d < \sqrt{2}\ Radius \tag{3.1}$$

Durch das Markieren der oben genannten Felder wird allerdings nur ein Teilbereich der abgefahren Fläche in die Karte eingetragen. Der Weg zum Ziel anhand dieser Karte entspräche den einzelnen Bahnen, auf denen der Roboter zu seiner jetzigen Position gefahren ist, da – wie Abbildung 3.10 zeigt – eine Verbindung zwischen den Bahnen nur an den Umkehrpunkten besteht.

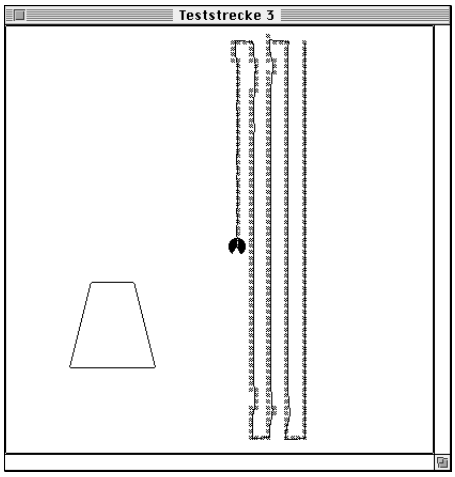


Abb. 3.10: NavigationalGridMap (falsch)

Um die komplette befahrene Fläche zu markieren, müssen die zur Roboterposition benachbarten Felder mit markiert werden. Dabei werden allerdings nur die Felder berücksichtigt, in deren Bereich sich kein Hindernis befindet.

Mit Hilfe dieses Verfahrens entsteht – wie in Abb. 3.11 zu erkennen – eine geschlossene Fläche von Rasterfeldern, die den befahrenen Bereich markiert.

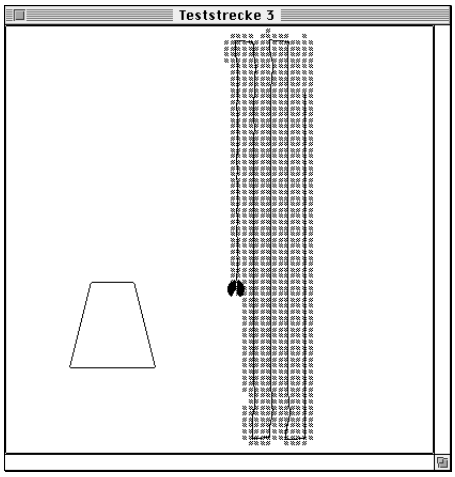


Abb. 3.11: Navigational GridMap (korrekt)

3.6 Observer

Der Observer ist für die Beobachtung der Umgebung zuständig, um gegebenfalls benachbarte, noch abzufahrende Bahnen auf dem Stack zu sichern. Abbildung 3.12 zeigt ein Beispiel einer Fahrt in der Nähe eines Hindernisses, wobei die gestrichelten Linien die Strecken zeigen, die dabei vom Observer auf den Stack gelegt werden.

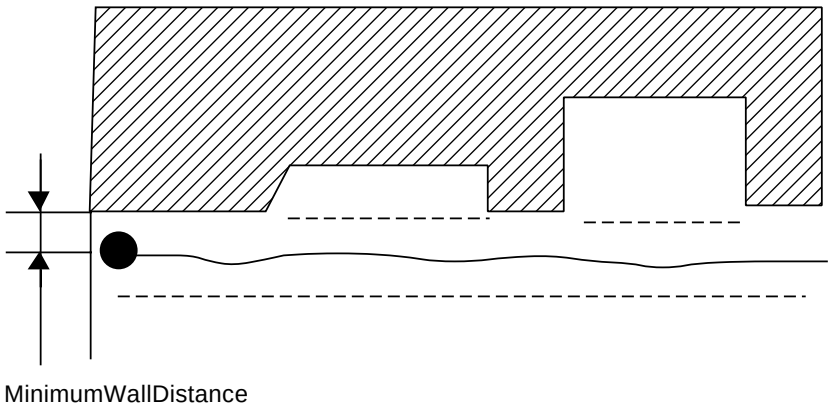


Abb. 3.12: Arbeitsweise des Observers

Um festzustellen, ob die Nachbarbahn noch zu bearbeiten ist, wird die Rasterkarte *GridMap* benutzt. Dazu wird der Bereich neben dem Roboter bis zur Enfernung *MinimumWallDistance*, dem einzuhaltenden minimalen Abstand vom Robotermittelpunkt zu Hindernissen, untersucht.

Der Bereich gilt genau dann als zu befahren, wenn

– kein Hindernis innerhalb dieses Bereiches ist, und

– sich unbefahrene Stellen darin befinden.

Befindet sich ein Hindernis innerhalb des untersuchten Bereiches, dann handelt es sich um den freien Bereich, der zur Einhaltung des Mindestabstandes erforderlich ist. Deshalb kann in diesem Fall die Nachbarbahn nicht mehr befahren werden.

Mit Hilfe dieser Beobachtungen werden während der Fahrt die gesamten Nachbarbahnen erforscht. Immer wenn ein befahrbarer Bereich erkannt wird, wird die Startposition festgehalten. Danach wird so lange gewartet, bis die entsprechende Nachbarbahn nicht mehr befahrbar ist. Ist dies der Fall, wird die Endposition festgehalten und die Strecke auf den Stack gelegt.

Da die noch abzufahrende Nachbarbahn als Gerade durch den festgehaltenen Anfangs- und Endpunkt definiert wird, arbeitet das Verfahren nur solange korrekt, wie sich der Roboter annähernd auf einer Geraden bewegt. Deshalb muß bei einer Drehung die Beobachtung abgebrochen werden. Dies geschieht durch das Löschen des Flags *ObserverActive*.

Das Verhalten läßt sich durch einen endlichen Automaten, mit den Zuständen *FindStart* und *FindEnd* wie in Abbildung 3.13 gezeigt, beschreiben.

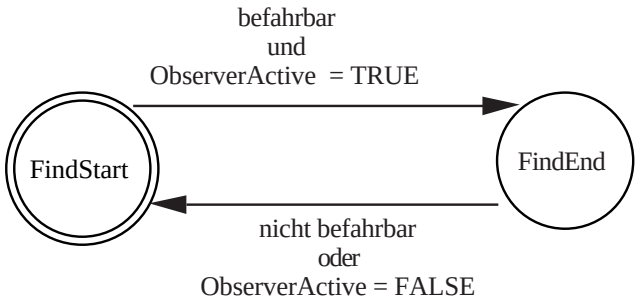


Abb. 3.13: endlicher Automat zum Observer

Der Automat befindet sich zu Beginn im Zustand *FindStart* bis eine befahrbare Fläche erkannt wird. Beim ansschließenden Übergang nach *FindEnd* wird der Startpunkt der Strecke festgehalten.

Im Zustand FindEnd bleibt der Automat solange, bis zum ersten Mal die benachbarte Strecke nicht mehr zu befahren ist oder *ObserverActive* auf FALSE gesetzt wird. Beim Übergang zurück nach *FindStart* wird der Endpunkt der Strecke gesichert und die komplette Strecke auf den Stack gelegt.

Eine Strecke wird nur dann auf dem Stack gesichert, wenn sie länger ist als der Durchmesser des Roboters zuzüglich dem einzuhaltenden Mindestabstand. Ist die Strecke kürzer, so ist der Roboter nicht in der Lage sie anzufahren, und sie wird deshalb vernachläßigt.

Der *Observer* besteht letztendlich aus zweien dieser endlichen Automaten, je einem pro Seite.

3.7 CheckStack

Eine Fläche, für deren Bearbeitung ein Eintrag auf dem Stack angelegt wurde, kann in der Zwischenzeit schon bearbeitet worden sein. Es besteht, wie Abbildung 3.14 zeigt die Möglichkeit, daß der Roboter diese Fläche abgearbeitet hat, indem er sich ihr von einer anderen Seite genähert hat. In diesem Fall muß der Stackeintrag gelöscht werden.

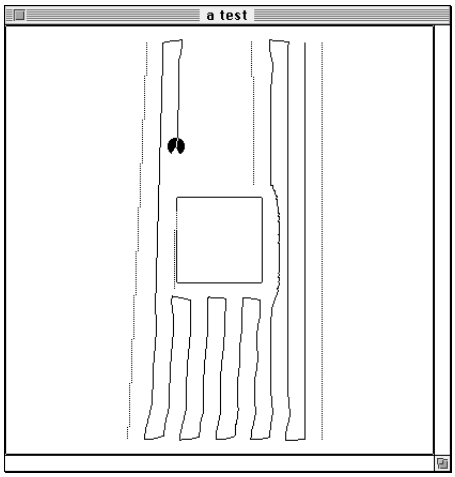


Abb. 3.14: CheckStack

Die Schwierigkeit besteht nun darin, festzustellen, wann eine Strecke gelöscht werden kann. Prinzipiell könnte eine Strecke, die erreicht wird, immer komplett vom Stack gelöscht werden. Auch wenn sie gekreuzt wird und so nur ein kleiner Bereich der Strecke abgedeckt wird, könnte sie gelöscht werden, da entlang der neuen Spur die Umgebung wieder beobachtet und zum späteren Abfahren vorgemerkt wird.

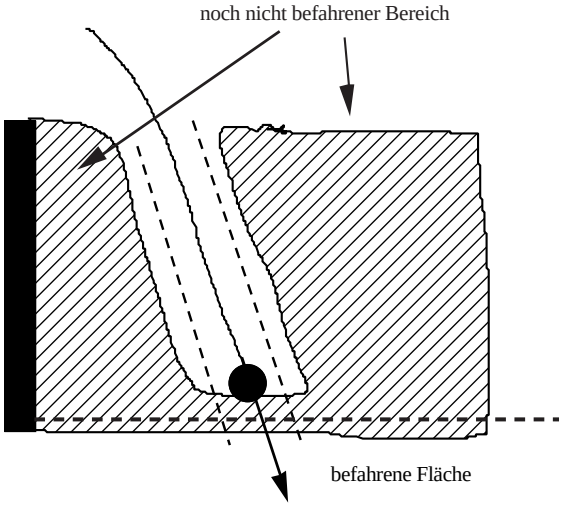


Abb. 3.15: Löschen eines Stackeintrags

Die Skizze in Abbildung 3.15 verdeutlicht diesen Zusammenhang. Die noch zu bearbeitende Fläche (schraffiert dargestellt) wurde bei einer früheren Vorbeifahrt erkannt und die entsprechende Strecke (waagrecht gestrichelte Linie) auf dem Stack vorgemerkt. Bei der Roboterfahrt wird nun die Fläche unterteilt und dann die Strecke gekreuzt. Die Skizze verdeutlicht, daß dabei parallel zur Bahn zwei neue Strecken auf den Stack gelegt werden (dünne gestrichelte Linien). Die Abarbeitung dieser Strecken führt dazu, daß die komplette Fläche abgedeckt wird, selbst wenn der erste Stackeintrag gelöscht wird.

Da der Observer Strecken nur ab einer bestimmten Länge auf den Stack legt, kann es bei schmalen, noch zu bearbeitenden Gebieten dazu kommen, daß das Löschen der Strecke diese Gebiete unbearbeitet läßt.

Sicherer ist es daher, die überfahrene Strecke nicht vom Stack zu löschen, sondern sie, wie in Abbildung 3.16 ersichtlich, an der Schnittstelle so aufzutrennen, daß die schneidende Bahn darin nicht mehr enthalten ist. Solange die entstehenden Teilstücke die Mindestgröße überschreiten, bleiben sie auf dem Stack gespeichert.

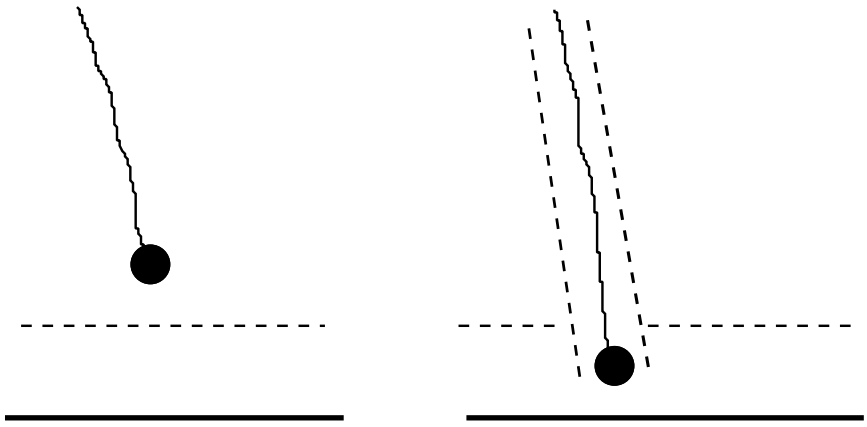


Abb. 3.16: Auftrennen der Strecke

Diese Vorgehensweise führt allerdings zu relativ viel Bewegung auf dem Stack, da wie die Skizze zeigt, die einzelnen Strecken immer wieder aufeinandertreffen und somit im Verlauf der weiteren Fahrt häufig neu aufgetrennt werden. Dadurch kann ein unübersichtliches und umständliches Gesamtverhalten entstehen.

Es wurden beide Verfahren implementiert und ausgetestet. Das erste Verfahren führt nur zu minimalen Flächen, die nicht bearbeitet werden, so daß es prinzipiell ausreichend ist. Soll dieses Risiko ausgeschlossen werden, so muß das zweite Verfahren gewählt und dafür die geringere Effizienz in Kauf genommen werden.

Das Verhalten *CheckStack* vergleicht in jedem Simulationsschritt die Roboterposition mit jeder der Strecken auf dem Stack. Befindet sich der Roboter in der direkten Nähe einer dieser Strecken, so wird der Stack auf die beschriebene Art und Weise verändert.

3.8 TurnAtTrack

Wie schon in Kapitel 3.7 angedeutet und aus Abbildung 3.17 ersichtlich, kann es vorkommen daß der Roboter auf seine eigene Spur, also auf eine schon befahrene Fläche trifft. Die bisher besprochenen Verhalten ignorieren diese Tatsache, was zwar nicht zu einem Fehlverhalten aber dennoch zu unnötigen Wegen führt.

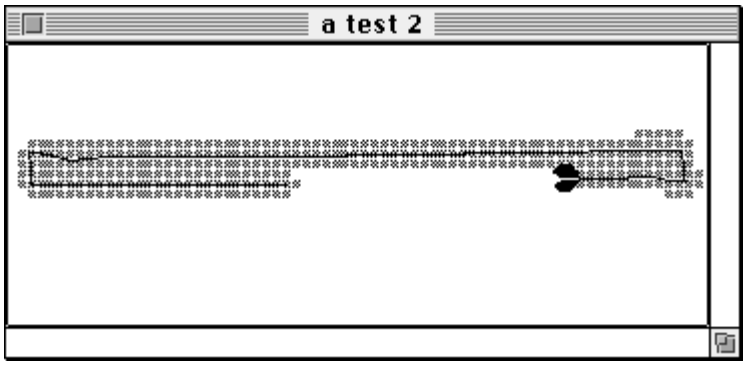


Abb. 3.17 : Auftreffen auf schon bearbeitetes Gebiet

Damit der Roboter auch dann umkehrt, wenn er sich auf schon bearbeitetes Gebiet bewegt wird das Verhalten *TurnAtTrack* eingeführt.

Das Verhalten beobachtet die Fläche vor dem Roboter: Befindet sich über dessen volle Breite befahrene Fläche, so löst *TurnAtTrack* ein Signal aus, durch welches *Drive* unterdrückt und *DriveToNextArea* ausgelöst wird.

Da *TurnAtTrack* nur dazu dient, das Verhalten *Drive* zu unterdrücken, ist es nur dann aktiv wenn *Drive* die Kontrolle des Roboters übernimmt. Weiterhin kann *Drive* nur unterdrückt werden, wenn das Flag *passed* in den Statusinformationen von *Drive* gesetzt ist, was bedeutet, daß *Drive* den Endpunkt der abzufahrenden Strecke schon erreicht hat.

Ein Abbrechen der Fahrt von *Drive* durch *TurnAtTrack* zu einem früheren Zeitpunkt ist auch deswegen nicht möglich, da nicht unterschieden werden kann, ob der gesamte Rest der Strecke schon bearbeitet ist oder es sich nur um eine kurze Unterbrechung der freien Bahn handelt. Denn der Roboter kann durch die Regelung in *Drive* für kurze Zeit komplett auf die Nachbarbahn gesteuert werden.

3.9 FindBestDirection

Der Observer beobachtet seine Umgebung nur während der Fahrt und es werden nur Strecken ab einer bestimmten Länge auf den Stack gelegt.

Dadurch entsteht bei engen Durchfahrten, wie in Abbildung 3.18, das Problem, daß die Strecke zu kurz ist, um die benachbarte Strecke zu beobachten. In Richtung der Strecke kann sich der Roboter nur minimal bewegen, da er gerade in die Durchfahrt paßt. Da die Länge der beobachteten Strecke der Verschiebung des Roboters entspricht, wird die Mindestlänge nicht erreicht und die Nachbarstrecke erscheint nicht mehr auf dem Stack.

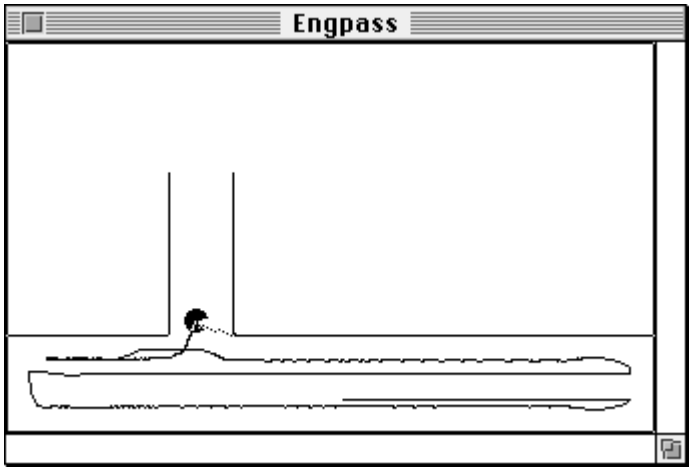


Abb. 3.18: Beispiel einer engen Durchfahrt

Es ist für enge Passagen allerdings auch nicht sinnvoll die abzufahrenden Bahnen, wie in der Abbildung, quer zur Durchfahrt zu legen. Besser wäre es wenn der Roboter an dieser Stelle der Wand folgt und somit eine längere Bahn zurücklegen kann.

Die bisher vorgestellten Verhaltensweisen erlauben dies nicht, da alle Bahnen möglichst parallel zur Ausgangsbahn gelegt werden. Die Hauptrichtung, die der Roboter zu Beginn der Fahrt erhält, wird im wesentlichen über die gesamte Fahrt beibehalten.

Damit auch enge Passagen korrekt bearbeitet werden, wird das Verfahren durch das Verhalten *FindBestDirection* erweitert. Mit dessen Hilfe wird unter gegebenen Umständen, d.h. wenn *Nextline* die notwendige Mindestlänge unterschreitet, nach einer besseren Richtung zur Weiterfahrt gesucht und *NextLine* entsprechend gesetzt.

Hierbei stellt sich die Frage, welche die beste Richtung zur Weiterfahrt ist und wie diese bestimmt werden kann. Unter der besten Richtung wird allgemein die Richtung verstanden, bei deren Einhaltung der Roboter vorraussichtlich die wenigsten Drehungen durchführen muß und die längsten Bahnen fahren kann.

Dieses Problem läßt sich nicht optimal lösen, da der Roboter nur die nähere, lokale Umgebung erfassen kann und eine Richtung, die in der Nähe optimal erscheint, kann durch die Beschaffenheit der Hindernisse in größerer Entfernung dennoch ungünstig sein.

Aus diesem Grund wurde eine einfache Heuristik implementiert, welche die Richtung derart bestimmt, daß die erste abzufahrende Bahn möglichst lang ist. Dazu wird diejenige Richtung gewählt, in der die freie Strecke am längsten ist. Dabei bezieht sich 'frei' sowohl darauf, daß kein Hindernis im Weg steht, als auch darauf, daß sich im 'freien' Bereich kein schon bearbeitetes Gebiet befindet.

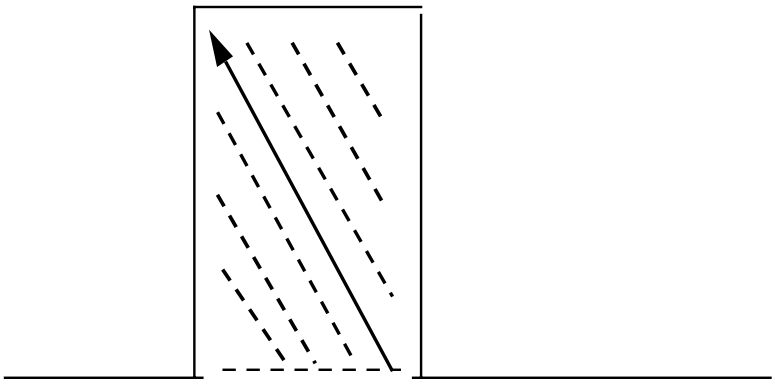


Abb. 3.19: Auswahl der besten Richtung

Abbildung 3.19 zeigt die Auswahl der Richtung an einem Beispiel. Dabei wird das Problem dieser Art der Richtungsbestimmung deutlich. Die größte Entfernung in der ersten Bahn läßt sich in der Diagonalen zurücklegen. Dies führt jedoch für die weiteren Bahnen zu einem schlechten Ergebnis, da der Roboter schräg auf das Hindernis trifft und somit die anschließenden Bahnen immer kürzer werden.

Ein besserer Ansatz zur Bestimmung der optimalen Richtung wäre die Bestimmung der Hauptrichtung der Wände über ein Winkelhistogramm. Aufeinanderfolgende Entfernungsdaten werden so ausgewertet, daß die Ausrichtung der Hindernisse an den einzelnen Stellen bestimmt wird. Werden all diese Einzelrichtungen in einem Histogramm zusammengestellt, so läßt sich daraus die Hauptrichtung bestimmen (vgl. [Weiß et al. 94], [Knieriemen 91]). Diese Hauptrichtung ermöglicht die Auswahl einer Bahn, die möglichst parallel zu den umgebenden Hindernissen verläuft. Dadurch werden auch die nächsten dazu parallel liegenden Bahnen günstig liegen.

Das aus der Literatur bekannte Winkelhistogrammverfahren berücksichtigt ausschließlich die Richtung von Wänden und Hindernissen. Es wird nicht berücksichtigt, ob die Hauptrichtung über schon bearbeitetes Gebiet führt und somit die Bahnen unnötigerweise lang werden.

Nicht immer sind Hindernisse ausschlaggebend dafür, daß die abzufahrende Strecke *NextLine* die Mindestlänge unterschreitet, sondern dies kann auch aufgrund der Grenzen von schon überfahrenem Gebiet geschehen. In diesem Fall erweist sich der zweite Ansatz als weniger geeignet.

Den besten Erfolg verspricht eine Kombination der beiden Ansätze, was allerdings späteren Versionen vorbehalten bleiben soll.

Nach der Bestimmung der besten Richtung muß das Verhalten *FindBestDirection* noch die neue Richtung in der globalen Variablen *Nextline* setzen, damit das Verhalten *Drive* darauf zurückgreifen kann. Da in der Regel die neue Bahn sich nicht mehr parallel zu schon befahrenem Gebiet befindet, muß *Drive* dann mittels der untergeordneten Schicht *DriveStraight* die Fahrt bestimmen.

Kapitel 4

Simulationsumgebung

Als Grundlage für die Simulation des flächendeckenden Fahrens dient die Simulationsumgebung *eCat*, welche im Rahmen einer Projektarbeit entwickelt wurde [Müller 97].

Ziel war die Untersuchung von lernenden Algorithmen in einem Szenario mit mehreren Robotern. Das Programm eCat simuliert dabei die Bewegung der einzelnen Roboter und stellt diese in einem Grafikfenster, welches das gesamte Szenario enthält, dar. Sie ist aufgrund der einfachen Möglichkeit, verschiedene Steuerungsalgorithmen zu implementieren und in unterschiedlichen Szenarien auszutesten geeignet, um das vorgestellte Konzept zum flächendeckenden Fahren auszuwerten.

Da das Programm ursprünglich für einen anderen Zweck konzipiert wurde und sich daraus die Funktion und Benennung der einzelnen Komponenten erklärt, soll im folgenden Abschnitt zunächst das Programm selbst erläutert werden. Im Anschluß daran wird darauf eingegangen, welche notwendigen Änderungen für das Austesten des flächendeckenden Fahrens durchgeführt wurden.

4.1 Beschreibung des Programms eCat

Ziel des Programm ist es, eine Simulationsumgebung zu schaffen, in der sich mehrere Roboter gleichzeitig bewegen, damit verschiedene Strategien zur Steuerung der Roboter ausgetestet werden können.

Die zugrundeliegende Idee, ist dabei die einer Katze, die Mäuse fängt. Während der Verfolgung einer Maus verbraucht die Katze Energie, wenn sie die Maus frißt erhält sie Energie dazu.

Aus dieser Grundidee ergeben sich die Bezeichnungen der einzelnen Roboter. Katze (Cat) bezieht sich dabei auf den Roboter, auf den die Simulation das Hauptaugenmerk richtet. Für diesen Roboter stellt die Simulation einen Sectorscan zur Hinderniserkennung, eine Kollisionserkennung, und Sensoren zur Erkennung der anderen Roboter zur Verfügung. Während jedes Szenario nur einen Roboter dieses Typs enthalten darf, kann es darüber hinaus noch weitere Roboter enthalten, die als Mäuse (Mouse) bezeichnet werden. Für diese Roboter wird nur eine Sensorik zur Kollisionserkennung simuliert.

4.1.1 Anlegen einer Szenariodatei für eCat

Die Beschreibung der Umgebung, in der sich der Roboter bewegen soll geschieht mit Hilfe einer Szenariodatei. Dies ist eine ASCII-Datei, die in einem beliebigen Editor erstellt wird und folgendem Format genügt:

Dimensionen der Umgebung:

d
<*Breite*> <*Höhe*>

Hierbei werden die Breite und die Höhe des rechteckigen Feldes, in dem sich die Roboter bewegen, angegeben.

Position der Katze:

c
<*xxx*> <*yyy*><*aaa*><*rrr*>

mit 'xxx' x-Koordinate, 'yyy' y-Koordinate. 'aaa' Winkel der Blickrichtung zur x-Achse und 'rrr' Radius der Katze. Die Winkelangabe erfolgt im mathematisch positiven Sinn in Grad.

Anzahl der Mäuse:

m
<*Anzahl der Mäuse*>

Da die Simulation zum flächendeckenden Fahren außer dem der Katze keine weiteren Roboter zuläßt, ist die Anzahl der Mäuse auf 0 zu setzen.

Anzahl der Wände:

w
<*Anzahl der Wände*> <*Anzahl einblendbarer Hindernisse*>

Diese Szenariodefinition wurde erweitert um die Möglichkeit einblendbarer Hindernisse zu schaffen. Unter einem einblendbaren Hindernis wird eine Gruppe von Wänden verstanden, die während des Simulationslaufs aktiviert bzw. deaktiviert werden kann. Die Anzahl einblendbarer Hindernisse bezieht sich dabei auf die Zahl der Gruppen und nicht auf die Zahl der Wände.

Danach folgt für jede Wand eine Zeile, die Anfangs und Endpunkt definiert:

<*ax*><*ay*><*bx*><*by*> <*HindernisNr*>

mit 'ax', 'ay' als Koordinaten des Anfangspunktes und 'bx', 'by' des Endpunktes. Auch diese Definition wurde erweitert, dabei gibt *HindernisNr* nun die Nummer des einblendbaren Hindernisses an. Wird die Hindernisnummer weggelassen oder besitzt sie den Wert 0, so handelt es sich um eine normale Wand, die immer aktiviert ist. Die vier Wände, die das Szenario nach außen hin abgrenzen, werden vom Programm automatisch eingefügt. Sie werden deshalb in der Szenariodatei nicht definiert.

Alle Einträge müssen in oben angeführter Reihenfolge erfolgen. Es können allerdings Kommentarzeilen, beginnend mit einem '#' eingefügt werden.

Eine exemplarische Szenariodatei ist im Anhang zu finden.

4.1.2 Einstellbare Parameter

Die Simulation selbst bietet drei Gruppen von Parametern.

– Simulationsparameter

– Katzenparameter

– Mausparameter

Jede dieser Parametergruppen wird in einer eigenen Dialogbox, die über den entsprechenden Menüpunkt im Menü 'Parameter' aufgerufen wird, abgefragt. Jede Parametergruppe kann für sich getrennt durch Anklicken des entsprechenden Buttons in der Dialogbox gespeichert und geladen werden. Zusätzlich dazu können alle Gruppen zusammen unter den Menüpunkten 'Alle Sichern' und 'Alle Öffnen' gespeichert bzw. geöffnet werden.

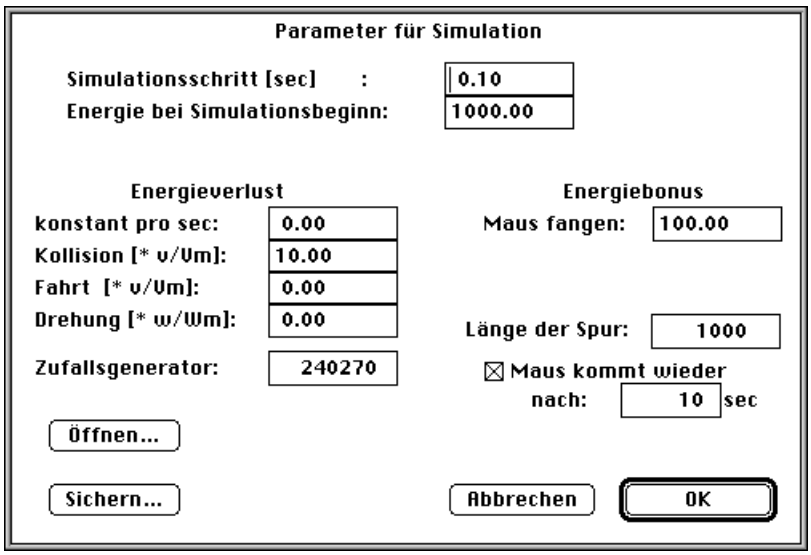


Abb. 4.1: Simulationsparameter

Von den in Abbildung 4.1 gezeigten Simulationsparametern sind für das flächendeckende Fahren nur die Dauer eines Simulationsschrittes und die Länge der Spur relevant.

Die Dauer eines Simulationsschritts gibt an, wieviel Zeit zwischen zwei Aufrufen des Steuerungsalgorithmus vergeht. Während dieser Zeit bewegt sich der Roboter mit gleichbleibender Geschwindigkeit und enthält keine weiteren Sensorinformationen.

Die Länge der Spur ist ein Parameter der nachträglich für das flächendeckende Fahrens eingeführt wurde. Er bestimmt die Länge der im Grafikfenster angezeigten Spur, die der Roboter hinter sich herzieht. Mit Hilfe dieser Spur läßt sich der zurückgelegte Weg verfolgen. Da diese Anzeige zur Kontrolle des flächendeckenden Fahren von entscheidender Bedeutung ist, wurde der Parameter eingeführt. Somit ist die Länge variabel und kann den gegebenen Umständen angepaßt werden.

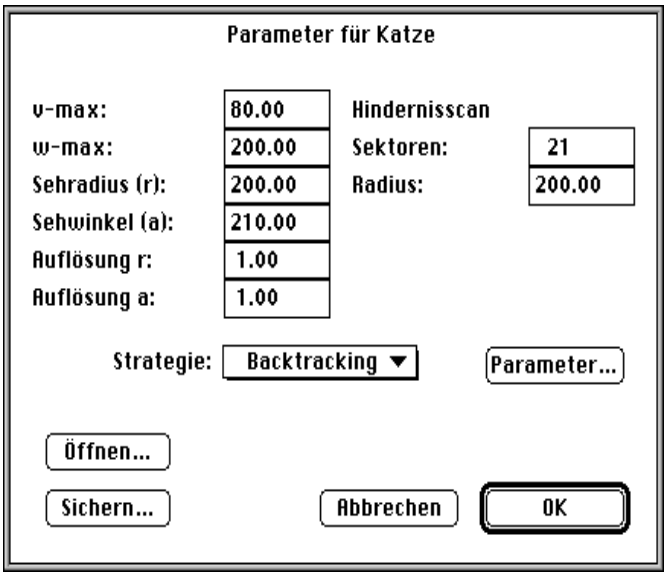


Abb. 4.2: Parameter für Katze

Abbildung 4.2 zeigt den Dialog zum Einstellen der Parameter der Katze, dem Roboter, der das flächendeckende Fahren durchführt. Damit werden die Parameter für Antrieb, Sensorik und das verwendete Steuerungsprogramm festgelegt.

v_max, w_max:

Diese Werte legen die maximale Bahn- (v_max) und Winkelgeschwindigkeit (w_max) des Roboters fest.

Sehradius , Auflösung r und *Auflösung a* bestimmen die Parameter für die Sensorik zum Erkennen anderer Roboter. Sie besitzen keinen Einfluß beim flächendeckenden Fahren.

Der *Sehwinkel* bestimmt den Öffnungswinkel des Blickfeldes für die Erkennung von Hindernissen. Das Flächendeckende Fahren setzt einen Sehwinkel von 210° voraus.

Sektoren gibt die Anzahl der Sektoren an, in die das Blickfeld unterteilt wird. Der vorliegende Algorithmus benutzt eine Unterteilung in 21 Sektoren von 10° Breite.

Mit der *Reichweite* wird angegeben, bis zu welcher Entfernung Hindernisse erkannt werden können.

Mit dem Popupmenü zur Einstellung der Strategie wird der Steuerungsalgorithmus ausgewählt. Beim flächendeckenden Fahren existiert nur eine Einstellung. Werden unterschiedliche Algorithmen eingebunden, erfolgt die Auswahl über dieses Menü.

Mit Hilfe des Buttons *Parameter* wird der Dialog zum Einstellen der Algorithmenparameter des ausgewählten Verfahrens aufgerufen. Diese werden im folgenden beschrieben.

Abb. 4.3: Algorithmenparameter flächendeckendes Fahren

Die Parameter, welche das flächendeckende Fahren beeinflussen, werden über die in Abbildung 4.3 gezeigte Dialogbox eingestellt. Die gezeigten Werte sind Standardwerte, die in den Tests für einen Roboter mit einem Durchmesser von 20 cm vernünftige Ergebnisse erzielen.

Auflösung GridMap:

Hiermit wird festgelegt, welche Auflösung die *GridMap* in x- und y-Richtung besitzt.

Mindestabstand zur Wand:

Der einzuhaltende minimale Abstand zwischen Robotermittelpunkt und einem Hindernis wird hier festgelegt.

Geschwindigkeit FollowWall und Geschwindigkeit FollowTrack:

Bei der Regelung zur Fahrt an einem Hindernis entlang (*FollowTrack*) bzw. an der eigenen Spur entlang darf nicht mit der vollen Geschwindigkeit gefahren werden. Während die ω-Komponente von dem jeweiligen Regler vorgegeben wird, bleibt die Bahngeschwindigkeit konstant. Der entsprechende Wert, der von Robotergröße und einzuhaltendem Mindestabstand abhängt, kann hiermit eingestellt werden.

TurnAtWall, CheckStack, FindBestDirection:

Mit Hilfe dieser Checkboxen lassen sich die einzelnen Verhaltensweisen aktivieren bzw. deaktivieren. Für ein optimales Gesamtverhalten sollten alle drei Verhaltensweisen aktiviert sein. Es besteht aber die Möglichkeit sie einzeln auszuschalten und zu beobachten, welche Auswirkungen dies auf das Gesamtverhalten hat.

4.1.3 Einbindung der Kontrollstruktur in die Simulationsumgebung

Die Simulationsumgebung ist in der Programmiersprache Pascal erstellt. Abbildung 4.4 zeigt ihren allgemeinen Aufbau. Zum Implementieren der Kontrollstukturen der einzelnen Roboter dienen die Units *Cat* und *Mouse.* Diese Units enthalten Schnittstellen, mit deren Hilfe die Steuerungsprozeduren eingebunden werden.

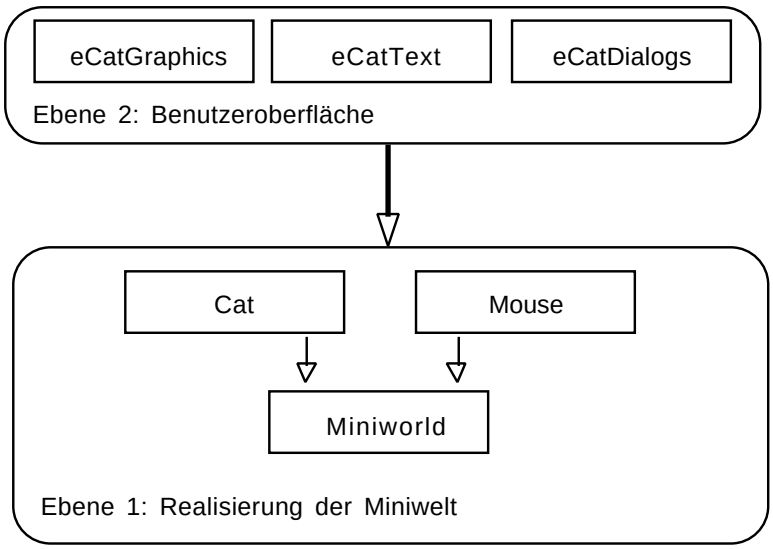


Abb. 4.4: Modulhierarchie von eCat

Die Simulationsumgebung verfolgt dabei eine Polling-Strategie. Dabei wird in jedem Simulationsschritt die Steuerungsprozedur aufgerufen. Diese muß dann mit Hilfe von bereitgestellten Funktionen die neuen Geschwindigkeitsparameter der (v,ω)-Schnittstelle setzen. Danach wird die Bewegung simuliert und die neuen Sensordaten berechnet.

Die Implementierung der Verhaltensweisen zum flächendeckenden Fahren erfolgt innerhalb der Unit *Cat.* Diese stellt hierzu folgende Funktionen, die als Schnittstelle zur Simulationsumgebung dienen, zur Verfügung:

– NextCatMove
– InitCat
– SetCatAlgorithmnParameters

In der Funktion *NextCatMove* wird der Aufruf für die eigentliche Kontrollstruktur eingetragen. Die Funktion *InitCat* enthält den Aufruf für die Initialisierung der Kontrollstruktur. Die Behandlung der Dialogbox zum Einstellen der Algorithmenparameter wird in der Funktion *SetCatAlgorithmnParameters* definiert.

Für detailiertere Informationen über den Aufbau der Simulationsumgebung sei auf [Müller 97] verwiesen.

4.2 Notwendige Erweiterungen

Über die angesprochenen Änderungen der Szenariodefinition und der Parameter hinaus sind weitere Veränderungen der Simulationsumgebung notwendig. Diese Veränderungen dienen der Veranschaulichung und Kontrolle des Ansatzes zum flächendeckenden Fahren. Dazu zählen:

– Darstellung der Rasterkarte in eigenem Fenster

– Darstellung der *Navigational GridMap*

– Darstellung des Stacks

Rasterkarte

Die Rasterkarte wird in einem eigenen Fenster dargestellt. Bei der Standarddarstellung entspricht jedes Pixel einem Punkt der Rasterkarte. Da das Szenario bis zu 2000 x 2000 Punkte zuläßt, stellt das Fenster nur einen Ausschnitt der Rasterkarte dar. Dieser Ausschnitt wird im Szenariofenster mit einem Rechteck markiert, so daß beim Scrollen des Rasterkartenfensters zu erkennen ist, welcher Bereich gerade angezeigt wird.

Um auch die gesamte Rasterkarte in dem Fenster anzeigen zu können, kann sie mit Hilfe des Menüpunktes 'Rasterkarte zoomen' im Menü 'Fenster' verkleinert werden. Dabei werden je vier Rasterpunkte zu einem Pixel zusammengefaßt. Damit keine wichtigen Informationen wie das Vorhandensein von Wänden verloren gehen, geschieht dies derart, daß das Pixel den höchsten Wert der vier Rasterpunkte annimmt. Da bei verkleinertem Raster viermal so viele Punkte berücksichtigt werden, wie bei der normalen Darstellung, verlangsamt sich allerdings der Aufbau des Fensters merklich. Deshalb sollte die gezoomte Karte nicht während des Simulationlaufs angezeigt werden.

Wird sie dennoch angezeigt, so wird sie alle zehn Simulationsschritte aktualisiert, so daß die Bewegung des Roboters auch anhand der Rasterkarte nachvollzogen werden kann. Wird die Simulation durch die Darstellung der Rasterkarte zu langsam, so kann sie im Menü 'Fenster' deaktiviert werden.

In der Rasterkarte werden noch nicht befahrene Flächen weiß und erkannte Hindernisse schwarz dargestellt. Je öfter eine Stelle überfahren wurde desto dunkler wird der entsprechende Bereich dargestellt.

Das Zeichnen der Rasterkarte im Fenster wird realisiert mit Hilfe einer Offscreen Pixmap der Farbtiefe 8 bit. Dies ermöglicht das Setzen von Pixeln mittels Direct-Bit, wodurch die Byte-Werte des Arrays direkt in den Offscreen-Bildspeicher kopiert werden. Dadurch wird ein, im Gegensatz zur Verwendung von SetPixel, sehr schnelles Neuzeichnen des Fensters ermöglicht.

Die Farbkodierung erfolgt durch die Definition einer Palette, die für jeden Wert die verwendete Farbe festlegt.

Navigational GridMap

Eine schnelle Möglichkeit zur Anzeige, welche Bereiche des Szenarios schon überfahren wurden, besteht mit der *Navigational GridMap*.

Solange der Simulationslauf angehalten wird, kann diese Karte im Szenariofenster eingeblendet werden. Dies geschieht durch Aufruf des Menüpunktes 'Navigational-Grid' im Menü 'Stack'. Dabei wird für jedes Feld der Karte, das als schon befahren gilt, an der entsprechenden Stelle im Szenariofenster ein graues Quadrat eingezeichnet. Wird der Simulationslauf fortgesetzt, so wird die eingeblendete *Navigational GridMap* wieder gelöscht.

PointStack

Zur Analyse des Verhaltens ist es notwendig, daß man während des Simulationslaufs einen Überblick über die auf dem Stack vermerkten Strecken erhält.

Hierzu gibt es mehrere Möglichkeiten der Anzeige, die im Menü 'Stack' zusammengefaßt sind. Die Anwahl dieser Menüpunkte ist nur möglich, wenn die Simulation angehalten wurde.

Die Strecken der einzelnen Stackeinträge werden dabei als gestrichelte Linien in das Szenariofenster eingetragen. Mit der Fortsetzung der Simulation werden diese Linien wieder gelöscht.

Durch Anwahl des Menüpunktes 'Stack (complete)' läßt sich der gesamte Stack im Szenariofenster anzeigen.

Mit 'Stack (Top)' wird nur der oberste Eintrag im Stack angezeigt. Damit läßt sich erkennen, welches Ziel der Roboter mit dem nächsten Einsatz von *DriveToNext-Area* ansteuert.

Mit Hilfe von 'NextLine' läßt sich die Strecke einblenden, welcher der Roboter gerade folgt.

Es gilt zu beachten, daß das Zeichnen der Stackeinträge mittels XOR erfolgt, was bedeutet, daß durch das erneute Aufrufen eines Menüpuktes die gezeichneten Linien wieder aus dem Fenster gelöscht werden. Erfolgt ein Aufruf zum Zeichnen des kompletten Stacks direkt nach dem Zeichen des obersten Stackeintrags, so werden alle Linien neu gezeichnet, und somit die schon gezeichnete Strecke des obersten Stackeintrags wieder gelöscht.

Zusätzlich zur Möglichkeit, die Stackeintrage über eine Auswahl der Menüpunkte anzuzeigen, wird während der Fahrt jede Strecke, die auf den Stack gelegt wird im Szenariofenster eingeblendet. Auf diese Weise läßt sich die Arbeitsweise des Observers verfolgen.

Falls dies nach längerer Fahrt zu einer unübersichtlichen Darstellung führt, kann durch ein kurzes Anhalten der Simulation ein Neuzeichnen des Fensters veranlaßt werden. Das Neuzeichnen erfolgt ohne die Darstellung der Stackeinträge.

Kapitel 5

Experimentelle Ergebnisse

In den vorangehenden Kapiteln wurde das allgemeine Konzept des Verfahrens sowie die daraus abgeleiteten Verhaltensweisen erklärt. Im folgenden wird nun das Zusammenspiel der genannten Verhaltensweisen an Beispielen gezeigt und untersucht, welche Flächendeckung dabei erreicht wird.

5.1 Beispiel einer einfachen Umgebung

Da bisher auf die ausführliche Darstellung von Beispielfahrten verzichtet wurde, soll nun anhand einer Fahrt durch eine einfache Umgebung das Zusammenspiel der Verhaltensweisen verdeutlicht werden.

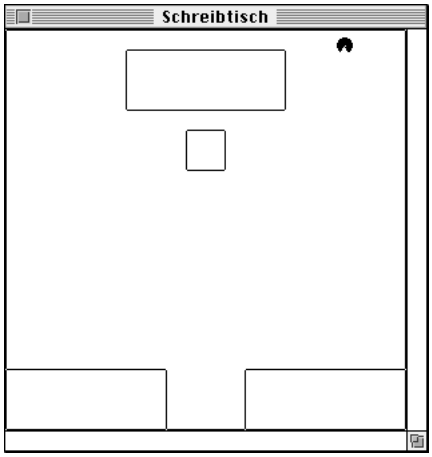


Abb. 5.1: einfaches Beispiel

Abbildung 5.1 zeigt die Beispielumgebung zu Beginn der Simulation. Die Umgebung enthält wenige Hindernisse mit relativ viel Raum dazwischen. Der Roboter ist so ausgerichtet, daß er zu Beginn eine lange gerade Strecke zurücklegen kann.

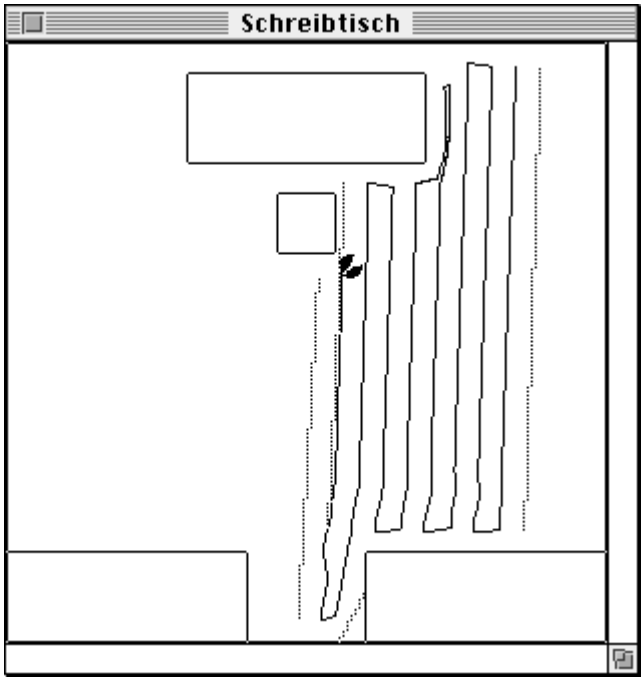


Abb. 5.2

Vom Ausgangspunkt hat sich der Roboter zunächst auf einer Geraden bis zum nächsten Hindernis bewegt. Man erkennt, wie durch *TurnAtWall* eine Drehung durchgeführt wurde. Abbildung 5.2 zeigt, wie sich der Roboter in Schlangenlinien zwischen den begrenzenden Hindernissen bewegt. Während der Roboter der Geraden folgt, ist *FollowTrack* dafür verantwortlich, daß sich die Bahnen korrekt aneinanderfügen.

In der gezeigten Situation versperrt ein Hindernis teilweise die Bahn. Es kann an dieser Stelle noch keine Umkehr erfolgen, da die Bahn (wie die gepunktete Linie oberhalb des Roboters anzeigt) noch nicht komplett bearbeitet ist. Aus diesen Gründen wird *FollowWall* aktiv und leitet ein Ausweichmanöver um das Hindernis ein.

Hat der Roboter anschließend die Wand erreicht, muß er das nächste Ziel suchen und den Weg dorthin planen. Dieses Ziel wird die gepunktete Linie links neben der Bahn sein, da diese zuletzt auf den Stack gelegt wurde.

Den Zustand der Rasterkarte (*GridMap*) in dieser Situation kann man in Abbildung 5.3 erkennen. Die Karte zeigt das bisher bekannte Gebiet, wobei die Wände schwarz dargestellt sind und das überfahrene Gebiet grau schattiert ist. Vergleicht man die Lage der Stackeinträge in Abbildung 5.2 mit der Rasterkarte, so wird deutlich, daß sie die Grenzen des bekannten Gebietes markieren. Dies sind die Stellen, in die sich das bekannte Gebiet ausbreitet.

Die Planung der einzelnen Bahnen orientiert sich hauptsächlich an dem schon befahrenen Gebiet. Im Gegensatz zu Verfahren, die ein a priori Wissen über die Umwelt besitzen, wird keine Bahn geplant, die optimal zwischen den Hindernissen liegt.

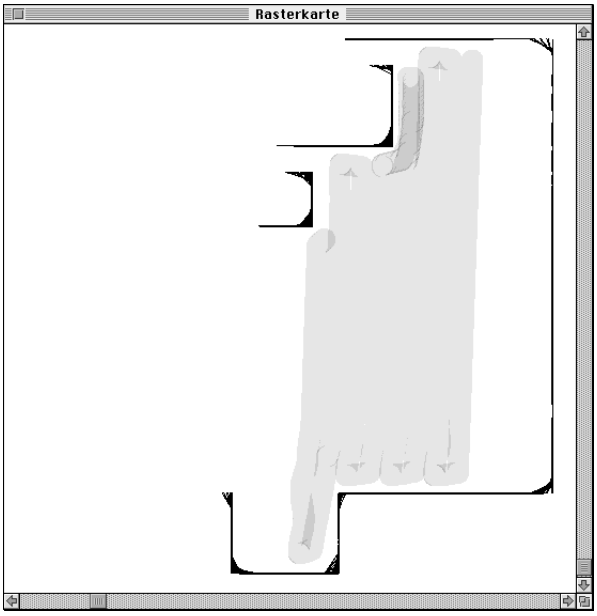


Abb. 5.3: Rasterkarte

Wird an dieser Stelle die Fahrt fortgesetzt, so wird der Rest des Szenarios genau auf die gleiche Art und Weise fortgesetzt, bis der linke Rand erreicht ist. Von dort aus erfolgt eine Bahnplanung in die Nische am unteren Szenariorand, um ein kleines Stück, das noch unbearbeitet geblieben ist, zu bearbeiten. Danach wird die noch freie Fläche auf der rechten Seite angefahren.

Die folgenden Abbildungen zeigen den zurückgelegten Weg und die Rasterkarte am Ende der Bearbeitung.

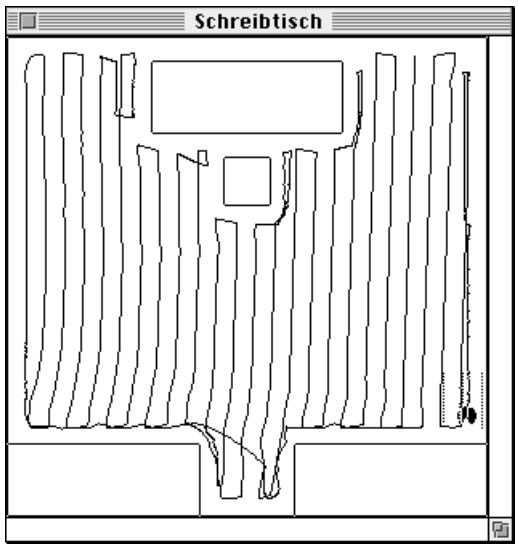


Abb. 5.4: Ende der Bearbeitung

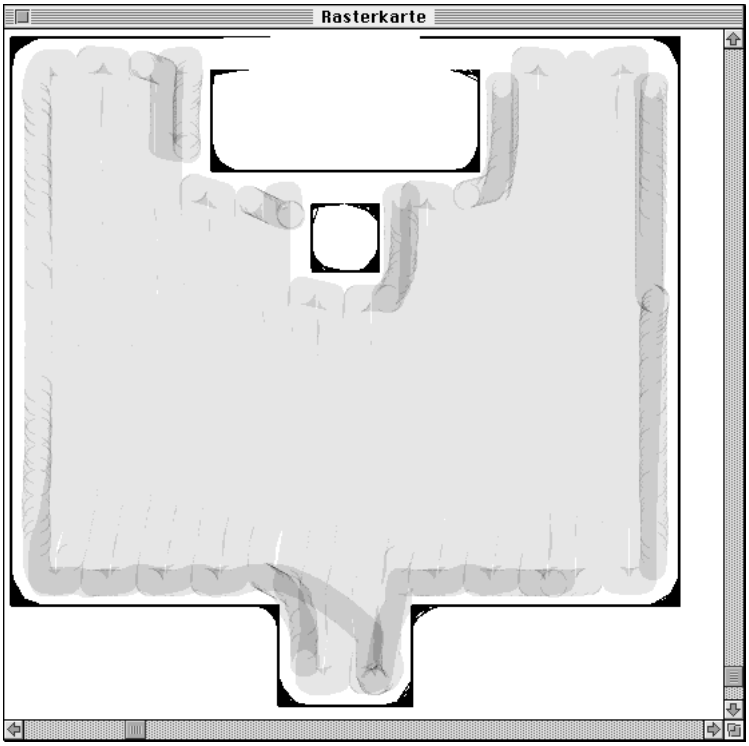


Abb. 5.5: Rasterkarte nach Beendigung der Fahrt

Bewertung

Bei der Beurteilung der Güte des Verfahrens sind zwei Kriterien zu beachten:

– Wieviel der Fläche ist nicht bearbeitet ?

– Wie effizient arbeitet das Verfahren ?

Die erste Frage bewertet, inwiefern das Verfahren seine Aufgabe überhaupt erfüllt, was sich durch einen Blick auf die Rasterkarte leicht klären läßt. Für das gezeigte Beispiel erkennt man, daß nur wenige kleine Lücken entstanden sind. Die Flächendeckung ist gut. Die verbleibende Restfläche ist im wesentlichen auf den einzuhaltenden Mindestabstand sowie auf die für den Roboter unerreichbaren Flächenstücke zurückzuführen.

Für die Beurteilung der Effizienz muß zunächst festgelegt werden, was diese bestimmt. Generell zeigt sich eine gute Effizienz durch möglichst schnelle Bearbeitung, d.h. einen möglichst kurzen Gesamtweg und wenige Drehungen auf der Stelle.

Unter der Berücksichtigung, daß das Verfahren die ganze Fläche abdeckt, bedeutet ein möglichst kurzer Weg möglichst wenige Überschneidungen. Damit läßt sich die Effizienz auch aus der Rasterkarte ablesen. Je höher der Anteil der nur einmal befahrenen Fläche an der gesamten überfahrenen Fläche ist, desto besser arbeitet das

Verfahren. Für das obige Beispiel wurden anhand der Rasterkarte folgende Werte für die überfahrene Fläche ermittelt.

Überfahrene Fläche insgesamt : 694246 cm^2

davon:	
1 x befahren	85,8 %
2 x befahren	12,4 %
3 x befahren	1,6 %
4 x und mehr	0,2 %

In etwa 14 % der überfahrenen Fläche wurden öfters als einmal angefahren. Dies ist zum einen auf das Zurückfahren vom linken zum rechten Rand zurückzuführen, zum anderen wurde am Rand ein sehr schmaler Streifen bearbeitet, was zwangsläufig zu einer größeren Überlappung der Bahnen führt.

Der optische Eindruck der Fahrt bestätigt, daß das Ergebnis, welches dieses Szenario liefert, einen Anhaltspunkt für eine gute Effizienz bietet. Der Roboter bewegt sich flüssig über die Fläche. Er gerät dabei in wenige Sackgassen, aus denen er zur weiteren Bearbeitung zurückfahren muß. Auch die angesprochene Überlappung der Bahn in der Nähe von Hindernissen hält sich in Grenzen.

Die guten Werte des Verfahrens bei diesem Szenario erklären sich dadurch, daß sich wenig Hindernisse im Raum befinden und alle Wände rechtwinklig zueinander angeordnet sind. Es bleiben Freiflächen, die genügend Raum für lange Bahnen bieten. Außerdem liegt die Startposition recht günstig. Der Roboter startet nahezu parallel zur Wand. Die Hauptrichtung, die der Roboter zu Beginn einschlägt, kann über die ganze Zeit beibehalten werden.

5.2 Beispiel einer schwierigeren Umgebung

Das folgende Beispiel zeigt eine Umgebung, welche mehrere Probleme aufzeigt, die es bei der Implementierung des Verfahrens zu berücksichtigen galt. Es wird sich zeigen, daß der vom Roboter zurückgelegte Weg nicht so übersichtlich bleibt, wie in der in Abschnitt 5.1 gezeigten.

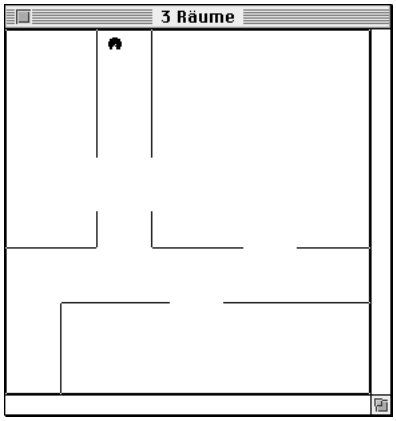


Abb. 5.6

Dieses Szenario wird durch mehrere Wände in einzelne Räume und schmale Flure unterteilt. Die Flure und Durchgänge zu den einzelnen Räumen sind schmal, so daß der Roboter zu Richtungswechseln gezwungen sein wird.

Die Einfahrt in den ersten Raum führt schon zu einer Abweichung von der ursprünglichen Richtung. Abbildung 5.8 zeigt den Zeitpunkt, zu dem Roboter an dem Durchgang vorbeifährt. Durch die schräge Fahrt an dieser Stelle wird die Linie für die nächste Bahn entsprechend schräg eingetragen.

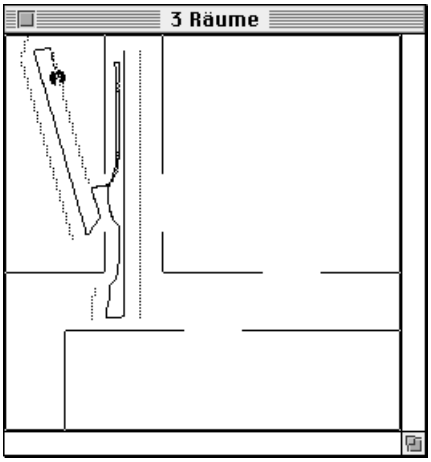


Abb. 5.7

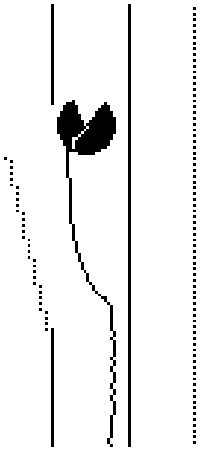
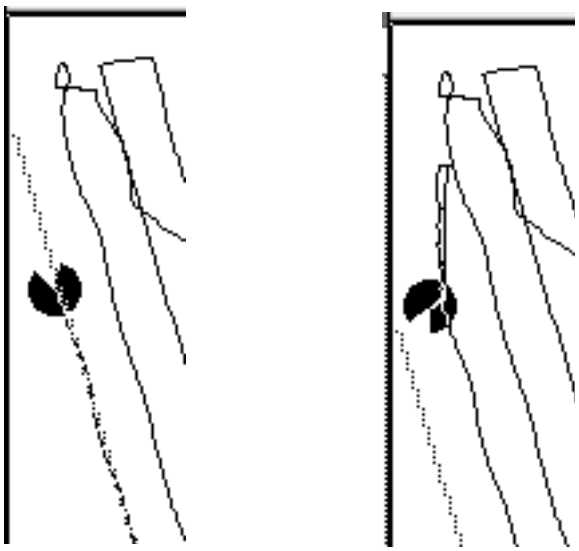


Abb.5.8: Ausschnitt

Die nachfolgenden Bahnen durchqueren den Raum weiterhin diagonal. Beim Auftreffen im spitzen Winkel auf die Wand derart, daß der nächste Freiraum in Richtung der Wand liegt, ist folgende Bahn zu beobachten.



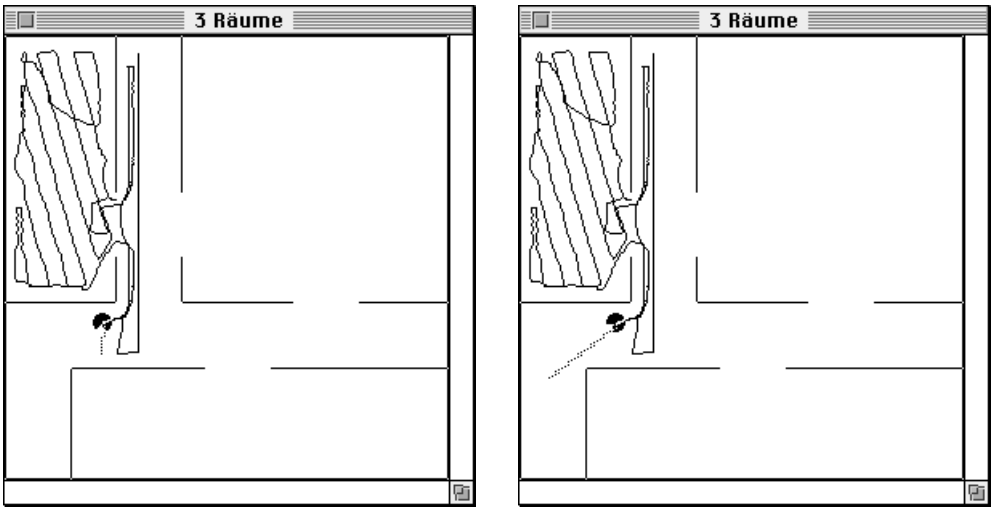
a) b)

Abb. 5.9: Auftreffen auf Wand in spitzem Winkel

Abbildung 5.9a zeigt die Situation zu dem Zeitpunkt, an dem der Roboter sich der Wand soweit genähert hat, daß ein Ausweichmanöver eingeleitet wird. Es ist zu erkennen, daß die Strecke noch nicht komplett abgefahren ist. Daraus folgt, daß *TurnAtWall* nicht durchschalten kann; statt dessen wird mit Hilfe von *FollowWall* der Wand gefolgt. Erst wenn der Roboter auf der Höhe der vorhergehenden Bahn ist, erkennt er, daß die Bahn beendet ist. Daraufhin kehrt er mit Hilfe von *DriveToNext*, wie in Abbildung 5.9b zu erkennen, zurück und fährt die nächste Bahn an.

Auf den ersten Blick erscheint die dadurch entstehende Schleife unnötig. Es erscheint sinnvoller, wenn in dieser Situation mit Hilfe von *TurnAtWall* sofort umgedreht würde. Aber das Folgen der Wand ist erforderlich, da sonst in der Nähe der Wand ein dreieckiges Stück unbearbeitet bleibt. Nur auf diese Art und Weise wird der Kontur der Wand bestmöglich gefolgt.

In der, in Abbildung 5.10a gezeigten Situation unterschreitet die eingeplante Strecke die Mindestlänge. Es wird mit Hilfe von *FindBestDirection* eine neue Richtung gesucht, in welcher der Roboter eine längere Bahn abfahren kann.



a) b)

Abb. 5.10: Suche nach einer längeren Bahn

In Abbildung 5.10b wird die von *FindBestDirection* ermittelte neue Bahn angezeigt. Sie verläuft diagonal, da so die längste Bahn zurückgelegt wird.

Die folgenden Abbildungen zeigen schrittweise die restliche Fahrt des Roboters.

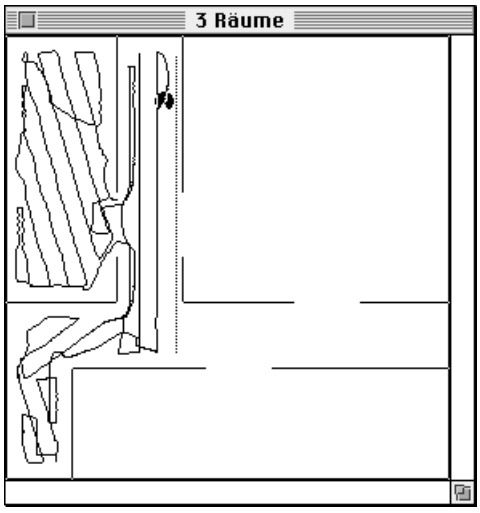


Abb. 5.11

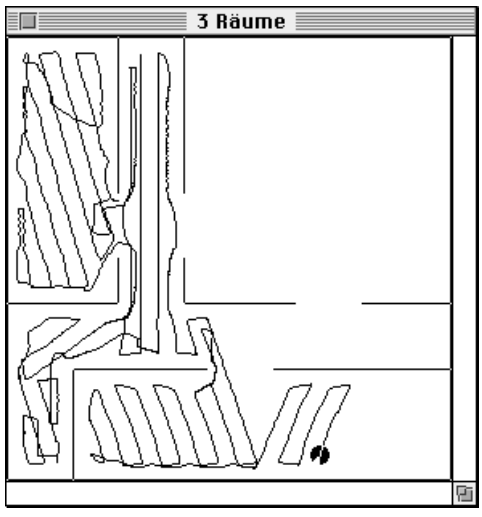


Abb. 5.12

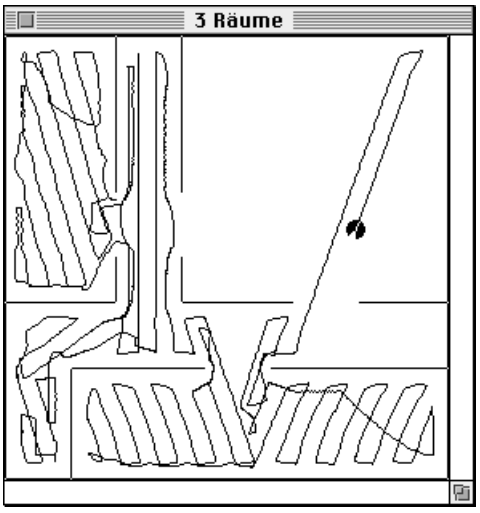


Abb. 5.13

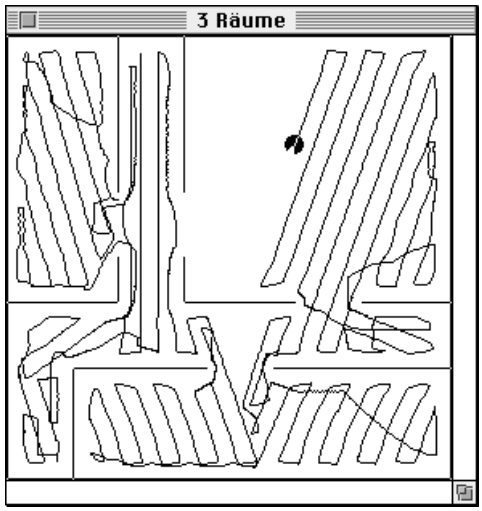


Abb. 5.14

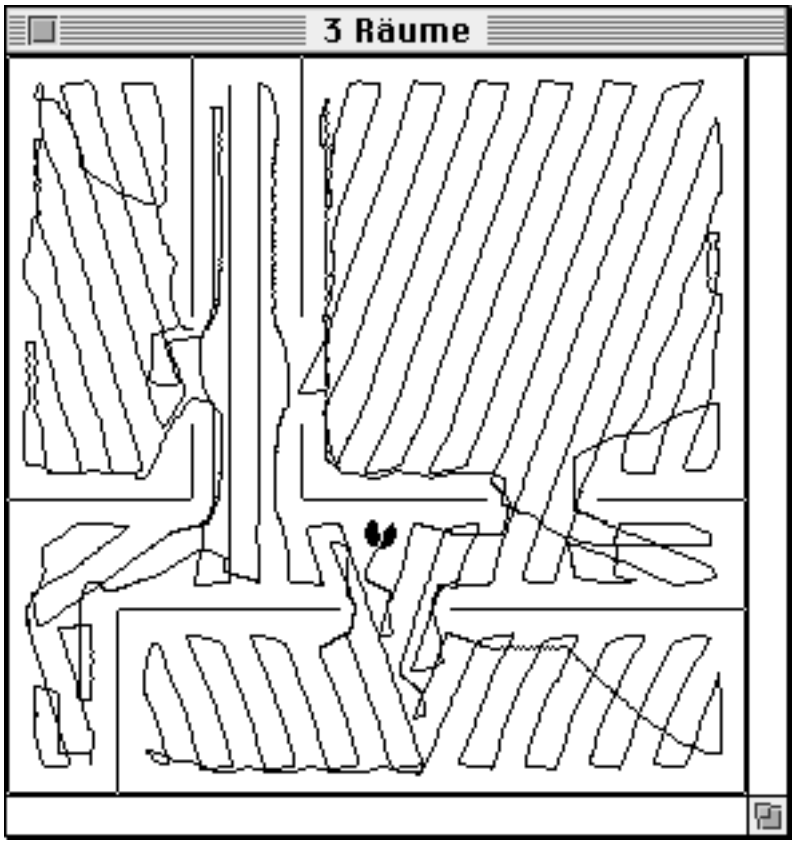


Abb. 5.15: Ende der Fahrt

Die Abbildungen verdeutlichen, daß es bei diesem Szenario zu sehr vielen Richtungsänderungen kommt. Bedingt durch die Änderung der Hauptfahrtrichtung stößt der Roboter häufig auf die eigene Spur, so daß den Verhaltensweisen *TurnAtTrack* und *CheckStack* hier große Bedeutung zu kommt. Ohne diese würde die befahrene Strecke deutlich länger.

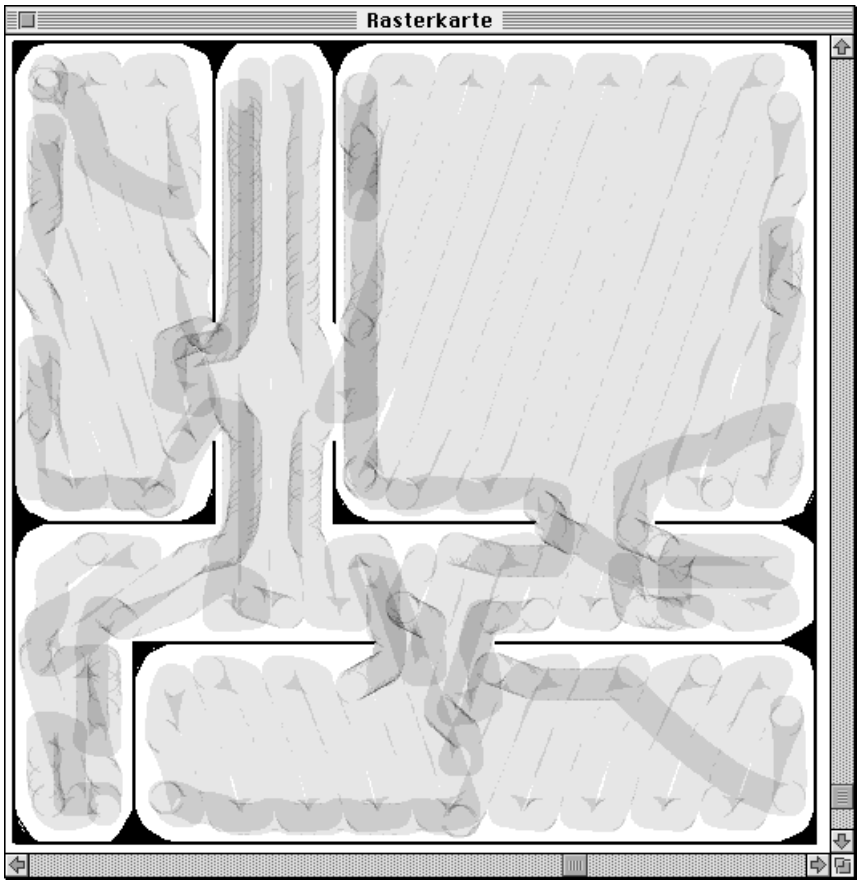


Abb. 5.16: Rasterkarte am Ende der Fahrt

Überfahrene Fläche insgesamt : 835133 cm^2

davon:	1 x befahren	71,3 %
	2 x befahren	22,4 %
	3 x befahren	5,6 %
	4 x und mehr	0,5 %

Die Rasterkarte weist deutlich Bereiche auf, an den es zu mehrmaligem Befahren der gleichen Stelle kommt. Es zeigt sich, daß im Bereich der linken unteren Ecke der Anteil an mehrfach befahrenen Flächen besonders hoch ist. Je schmaler die Durchfahrt ist, desto mehr Richtungsänderungen sind notwendig und damit verbunden werden größere Flächenanteile mehrmals befahren.

Vergleicht man die obigen Zahlen mit denen des Szenarios in Abschnitt 5.1, so sieht man, daß der Anteil an mehrfach befahren Flächen hier doppelt so hoch ist. Dennoch werden auch hier nahezu drei Viertel der gesamten befahrenen Fläche nur einmal überfahren. Dieses Ergebnis kann durchaus als zufriedenstellend betrachtet werden.

5.3 Behandlung dynamischer Hindernisse

Bei der Untersuchung des Verfahrens wurde bisher von einer sich nicht verändernden, statischen Umwelt ausgegangen. In einer realen Einsatzumgebung kann das Vorkommen dynamischer Objekte allerdings nicht ausgeschlossen werden. Zum Beispiel können Menschen die Bahn des Roboters kreuzen, oder Türen werden geöffnet oder geschlossen. Der Roboter sollte in der Lage sein, sich auch in einer solchen Umgebung zu bewegen.

Dieser Abschnitt beschäftigt sich mit den Auswirkungen auf das Verfahren bei einem Einsatz in einer Umgebung mit dynamischen Hindernissen. Es wird gezeigt, in wiefern das Verfahren in der Lage ist, auf diese zu reagieren und welche Erweiterungen noch notwendig sind.

Umgebungen mit dynamischen Hindernissen führen zu folgenden Problemen beim flächendeckenden Fahren:

- Bewegliche Objekte erfordern eine gesonderte Form der Kollisionsvermeidung, da das Objekt selbst sich auf Kollisionskurs befinden kann.
- Zu bearbeitende Flächen werden nicht erkannt, wenn der Zugang gerade durch ein dynamisches Hindernis blockiert ist.
- Dynamische Hindernisse dürfen nicht statisch in eine erstellte Karte der Umwelt eingeplant werden.
- Eine geplante Bahn kann durch ein dynamisches Hindernis blockiert sein, so daß ein neuer Weg zum Ziel gesucht werden muß.

Das erste Problem wird vermieden, wenn keine 'aggressiven' Hindernisse zugelassen werden, so daß einfaches Stehenbleiben im Nofall ausreicht, um eine Kollision zu vermeiden. Ansonsten wird versucht, das dynamische Hindernis auf die gleiche Art und Weise zu umfahren wie die statischen.

Das zweite Problem läßt sich ohne a priori erstellte Karte nicht lösen, da der Roboter dynamische Hindernisse von statischen nicht unterscheiden kann. Er kann weder entscheiden, ob ein Durchgang in Zukunft frei sein wird, noch wann dies geschehen wird.

Fährt der Roboter zum Beispiel an einer geschlossenen Tür vorbei, erkennt er an dieser Stelle eine Wand. Wird diese Tür erst später geöffnet und der Raum somit befahrbar, dann wird dieser nur bearbeitet, wenn der Roboter zufällig die Tür einweiteres Mal passiert und dabei feststellt, daß sich an dieser Stelle noch unbearbeitetes Gebiet befindet.

Eine Flächendeckung kann nur garantiert werden für die Fläche, die während der gesamten Fahrt des Roboters zugänglich ist.

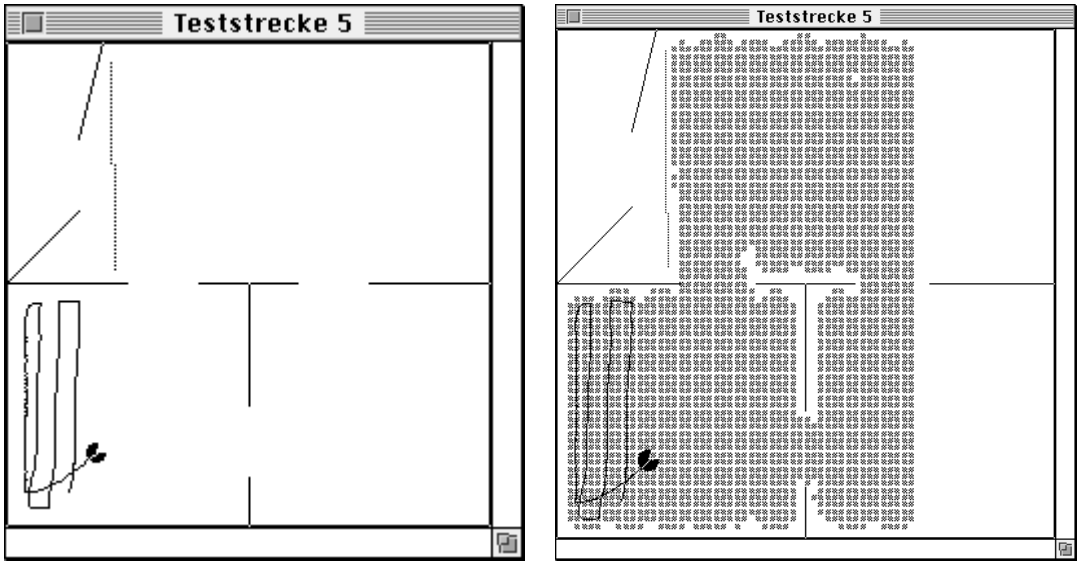
Die aktuelle Implementierung des *MapBuilders* plant jedes erkannte Hindernis sogleich in die Karte ein. Eine Möglichkeit, dieses aus der Karte zu entfernen gibt es nicht. Da ein dynamisches Hindernis nicht als solches erkannt wird, wird es entsprechend fest in die Karte eingeplant. Wenn dieses sich weiterbewegt wird es folg-

lich zusätzlich an der neuen Position eingetragen. Die Karte wird somit schnell unbrauchbar und das gesamte Verfahren scheitert. Kommt aber später bei der Implementierung des *MapBuilders* ein probalistisches Verfahren zur Umweltmodellierung (siehe Kapitel 3.5) zum Einsatz, so können die Freiflächen, die beim Wegbewegen des Hindernisses entstehen, auch wieder als solche in der Karte vermerkt werden.

Das letzte Problem, welches die Navigation auf schon bekanntem Gebiet betrifft, ist in der vorliegenden Implementierung des Verfahrens schon berücksichtigt. Während der Roboter auf dem Weg zu einem neuen Ziel ist, wird die *Navigational GridMap* zur Planung benutzt. Trifft der Roboter auf diesem Weg auf ein dynamisches Hindernis, so kann er dieses als solches Erkennen, da es auf bekanntem Gebiet liegt. Er vermerkt es als temporäres Hindernis in der *Navigational GridMap* und plant einen neuen Weg. Es wird später aus der Karte wieder gelöscht, so daß er dann wieder in der Lage ist, eine Fahrt durch das momentan belegte Gebiet zu planen. (siehe auch Kapitel 3.3 *DriveToNextArea*)

Die Simulation bietet noch keine Möglichkeit, bewegliche Objekte zu verwenden. Es können aber temporäre Hindernisse ein- und ausgeblendet werden. So läßt sich das oben genannte Verhalten austesten. Dabei ist allerdings darauf zu achten, daß aufgrund der angesprochenen Einschränkung durch den *MapBuilder* die Hindernisse nur in der Nähe des Roboters eingeblendet werden dürfen, wenn dieser sich auf dem Weg zu einem neuen Ziel (Verhalten *DriveToNextArea*) befindet.

Das folgende Beispiel verdeutlicht das Verhalten bei eingeblendeten Hindernissen.



a) b)

Abb. 5.17: Ausgangssituation

In der in Abbildung 5.17a gezeigten Situation muß der Roboter einen Weg zu der Linie in der linken oberen Ecke des Szenarios planen. Abbildung 5.17b zeigt dabei den schon befahren Bereich, innerhalb dessen die Planung erfolgt (als graue Fläche dargestellt). Zunächst plant der Roboter den direkten Weg durch die obere Tür und beginnt sich in diese Richtung zu bewegen.

Diese Tür wird nun mit einem einblendbaren Hindernis verschlossen. Während der Roboter sich auf das Hindernis zubewegt, erkennt er dieses als temporär und vermerkt die entsprechenden Bereiche in der *Navigational GridMap*. In Abbildung 5.18 sind diese als graue Kästchen im Szenariofenster zu sehen.

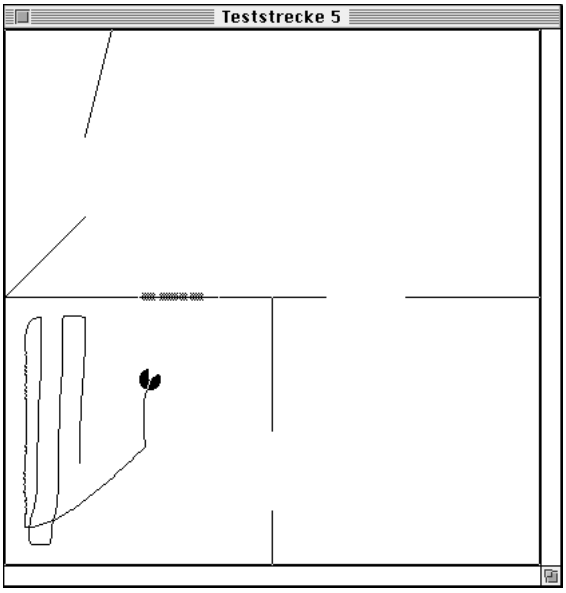
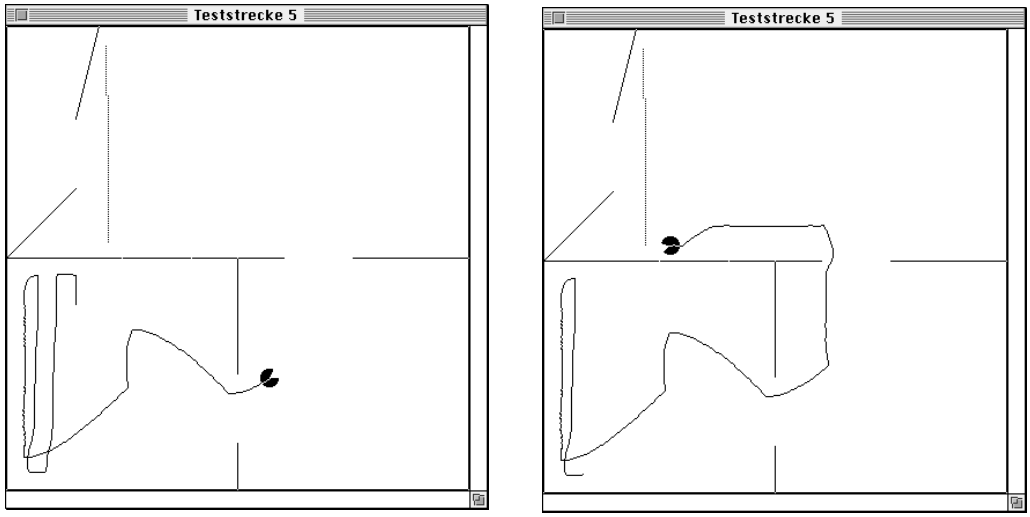


Abb. 5.18: Eintragen des temporären Hindernisses

Durch das eingetrage temporäre Hindernis wird die geplante Fahrt ungültig und es wird eine neue Fahrt durch den Nebenraum geplant.



a) b)

Abb. 5.19: neugeplanter Weg zum Ziel

Wird der in Abbildung 5.19 gezeigte Kurs durch das Einblenden eines weiteren Hindernisses blockiert, so muß wieder eine Neuplanung erfolgen. Im Beispiel existiert jetzt allerdings kein Weg mehr zu dem Ziel. Es wird daher mit der Bearbeitung des nächsten Stackeintrags, der anfahrbar ist, begonnen. (Abbildung 5.20)

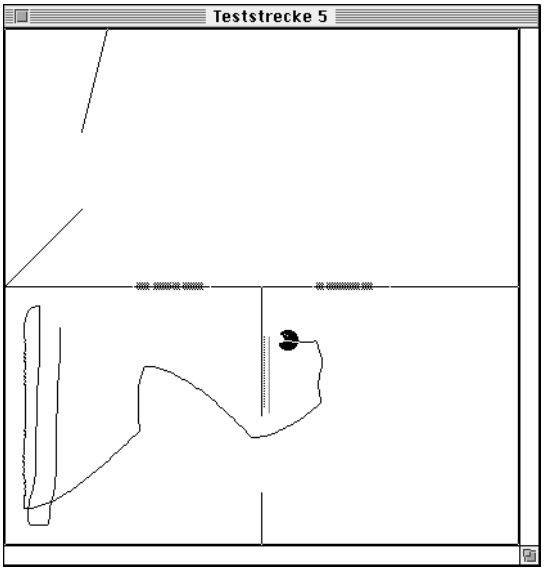


Abb. 5.20: Wechsel zu nächstem Stackeintrag

Da der nicht angefahrene Stackeintrag nicht gelöscht, sondern lediglich auf dem Stack nach unten verschoben wurde, wird nach Bearbeitung der übrigen Stackeinträge erneut versucht, die Strecke anzufahren. Ist zu diesem Zeitpunkt eine der versperrten Durchfahrten wieder frei, so wird der entsprechende Weg eingeplant.

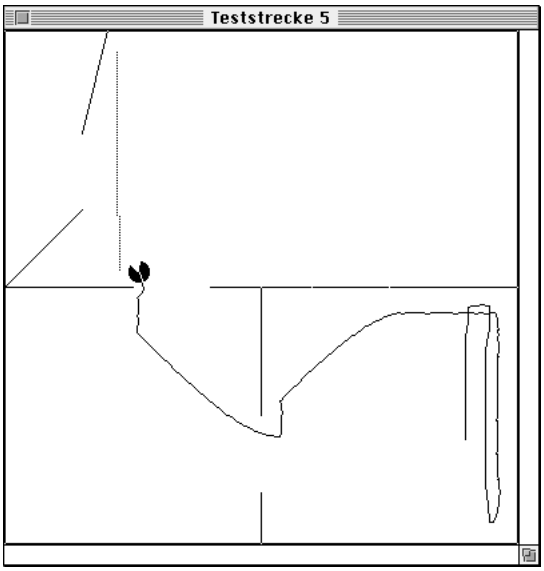


Abb. 5.21: Fahrt durch wieder geöffneten Durchgang

Kapitel 6

Zusammenfassung

Im Rahmen dieser Diplomarbeit wurde ein Verfahren entwickelt, welches die effiziente Steuerung eines AMR zum flächendeckenden Fahren ermöglicht, wobei parallel zur Bearbeitung ein rasterbasiertes Umweltmodell erstellt wird. Ein a priori vorhandenes Umweltmodell wird nicht benötigt.

Zur Vereinfachung wird von einer runden Geometrie des Roboters sowie von der Existenz eines Localizer-Moduls ausgegangen, welches gegebenfalls mit Hilfe spezieller Sensorik die Positionsbestimmung des Roboters erlaubt.

Bei dem vorgestellten Ansatz wird die Fläche in einzelnen, zueinander parallelen Bahnen mäanderförmig befahren. Die Flächendeckung wird durch ein Backtrackingverfahren sichergestellt. Dabei werden während der Fahrt auf jeder Bahn alle Strecken, die potentiell noch zu bearbeiten sind auf einem Stack festgehalten und somit zur späteren Bearbeitung vorgemerkt.

Für das Verfahren wurde innerhalb einer Simulation eine verhaltensorientierte Kontrollstruktur implementiert. Diese eignet sich aufgrund ihres modularen Aufbaus, den kurzen Reaktionszeiten und der hohen Fehlertoleranz auch für eine spätere, tatsächliche Realisierung. Sie untergliedert sich in aktuatorische Verhaltensschichten, welche direkt die Steuerung des Roboters beeinflussen und planerische Verhalten, welche die globalen Datenstrukturen, wie z.B. Karten, verändern.

Die aktuatorischen Verhaltensweisen entsprechen dabei den einzelnen, während der Fahrt benötigten Bewegungsfolgen. So existieren Verhaltenschichten für die bahnparallele Fahrt (*DriveStraight*, *FollowTrack*), das Drehen an einem Hindernis (*TurnAtWall*) und für die Navigation zu einem entfernten Punkt (*DriveToNextArea*). Die Verhaltensschichten *FollowWall* und *TurnAtWall* ermöglichen dabei ein konturparalleles Ausweichen an Hindernissen. Die Navigation mittels *DriveToNextArea* erfolgt nur auf schon bearbeitetem Gebiet. Dabei kommt zur Punkt-zu-Punkt Planung ein Distance Transform Verfahren zum Einsatz, welches zu den rasterbasierten Potentialfeldmethoden zu zählen ist.

Die planerischen Verhaltensschichten umfassen einen *MapBuilder* zur Erstellung der benötigten Rasterkarten, einen *Observer* zur Beobachtung der Nachbarbahnen und dem Aufbau des Stacks, sowie eine Verhaltensschicht *FindBestDirection*, die es bei sehr kurzen Bahnen ermöglicht, von der bahnparallelen Fahrt abzuweichen und eine neue Richtung zu planen in der ein effizienteres Arbeiten möglich ist.

Darüberhinaus werden zwei weitere Verhaltensschichten implementiert, welche dem Umstand Rechnung tragen, daß im Verlaufe einer Fahrt, die zuvor auf dem Stack vorgemerkten Bereiche verändert werden. Dies sind die Verhaltensschichten,

CheckStack zur Aktualisierung der Stackeinträge und *TurnAtWall*, welche die Geradeausfahrt beendet, falls die Bahn in schon bearbeitetes Gebiet führt.

Bei der Implementierung des *MapBuilders* wurde ein einfaches Verfahren gewählt, welches jedes erkannte Hindernis fest in eine Rasterkarte einträgt. Für die Zukunft empfiehlt sich hier ein probalistischer Ansatz welcher neben der Korrektur von Fehlmessungen auch den korrekten Umgang mit dynamischen Hindernissen erlaubt.

Das Verfahren gestattet bereits jetzt in begrenztem Maße eine Umwelt mit dynamischen Hindernissen. In der vorliegenden Implementierung erfolgt eine Reaktion auf diese allerdings nur während der Navigationsphase auf schon bearbeitetem Gebiet. Dies beruht auf der o.g. Implementierung des *MapBuilders*. Treten während der Fahrt in bekanntem Gebiet unerwartete Hindernisse auf, so ermöglicht die Navigation ein Umplanen des Kurses.

Die Implementierung der Kontrollstruktur basiert auf der schon vorhandenen Simulation eCat, welche das Austesten verschiedener Steuerungsalgorithmen in unterschiedlichen, über eine Datei definierbaren Umgebungen vorsieht. Diese wurde dahingehend erweitert, daß sie für das Austesten des flächendeckenden Fahrens geeignet ist. Zunächst wurde die Szenariostruktur ergänzt, um auch Umgebungen mit einblendbaren temporären Hindernissen zuzulassen. Weiterhin wurden Möglichkeiten geschaffen, welche die Beobachtung und die Bewertung des Verfahrens gestatten. Es können die erzeugten Rasterkarten und der Stack angezeigt werden. Die Rasterkarte des Umweltmodells wird so dargestellt, daß anhand der Einfärbung erkennbar wird, wie oft jede Stelle des Gebietes überfahren wurde. Am Ende der Simulation erfolgt die Angabe der befahrenen Fläche, und welcher Anteil davon mehrmals befahren wurde.

Das implemetierte Verfahren wurde mit einer Reihe von unterschiedlichen Szenarien ausgetestet.

Literaturverzeichnis

[Brooks 86] Brooks, R. A.: *A Robust Layered Control System for a Mobile Robot*; IEEE Journal of Robotics & Automation, 2(1), 1986

[Burhanpurkar 94] Burhanpurkar, V.P.: *Real World Apllication of a Low-Cost High-Performance Sensor System for Autonomous Mobile Robots*; Proceedings of the International Conference on Intelligent Robots and Systems 1994 (IROS); München; 1994

[Edlinger 97] Edlinger, T.: *Hierarchische Steuerung für einen mobilen Roboter zur autonomen Erkundung seiner Einsatzumgebung*; Fortschr.-Ber. VDI Reihe 8 Nr. 638; VDI Verlag; Düsseldorf;1997

[Electrolux 97] Pressemitteilung der Firma Electrolux zur Vorstellung eines autonomen Bodenreinigungsroboters; *http://www.electrolux.se/corporate/pressnotes/274a4b2.htm*; siehe auch *http://www.electrolux.se/robot/*; Dezember 1997;

[Elfes 89] Elfes, A.: *Using Occupancy Grids for Mobile Robot Perception and Navigation*, IEEE Computer 6/89

[Furuhashi et al. 92] Furuhashi, H. et al.: *Development of an Autonomous Cleaning Robot with an External power Cable*; Proceedings of the International Conference on Motion and Vibration Control, Yokohama, Japan, 1992

[Hako 94] Hako - Werke GmbH & Co., *Vollautomatischer Reinigungsroboter Hako-Robomatic 80 für die Reinigung großflächiger Hartböden;* Bad Oldesloe; 1994

[Hofner 97] Hofner, C.: *Automatische Kursplanung und Fahrzeugführung für mobile Roboter bei flächendeckenden Bearbeitungsaufgaben*; Dissertation am Lehrstuhl für Steuerungs- und Regeltechnik, TU München, 1997

[Jones & Flynn 93] Jones, J. L; Flynn, A. M.: *Mobile Robots: inspiration and implementation*; A K Peters; Wellesley; 1993

[Jörg 94] Jörg, K.-W.: *Echtzeitfähige Multisensorintegration für autonome mobile Roboter*; BI-Wissenschafts-Verlag;Mannheim; 1994

[Kärcher 96] KÄRCHER GmbH: *Informationsblätter zum Reinigungsroboter BR700*; Winnenden, 1996

[Kasper 94] Kasper, M.: *Ein einfaches, direktes und effektives Benutzerinterface für einen komplexen, ferngesteuerten Roboter*; Diplomarbeit im Fachbereich Informatik; 1994

[Kasper 97] Kasper, M.: *Unterlagen zum Robotikproktikum*; Fachbereich Informatik; 1997

[Kent 98] Im Internet: *http://www.kentco.com/robokent/*

[Knieriemen 91] Knieriemen, T.: *Autonome mobile Roboter: Sensorinterpretation und Weltmodellierung zur Navigation in unbekannter Umgebung*; BI-Wissenschaftsverlag; Mannheim; 1991

[Latombe 91] Latombe, J.-C.: *Robot Motion Planning*; Kluwer Academic Publishers; Boston; 1991

[Lawitzki et al. 98] Endres, H.; Feiten, W.; Lawitzki, G.: *Field Test of a Navigation System: Autonomous Cleaning in Supermarkets*; Proceedings of the 1998 IEEE International Conference on Robotics & automation (ICRA); Leuven, Belgium, May 1998

[Lu und Milos 97] Lu, F.; Milos, E.: *Robot Pose Estimation in unknowm Environments by matching 2D Range Scans*; Journal of Intelligent and Robotic Systems. vol. 18 pp. 249-275, 1997

[Müller 97] Müller, D.: *eCat – Eine Simulation für Artificial Life Untersuchungen;*Projektarbeit im Fachbereich Informatik; Februar 1997

[Puttkamer] von Puttkamer, E.: *Skript zur Vorlesung Autonome mobile Roboter*;

[Schraft 98] Schraft,R. D.; Schmierer, G.: *Serviceroboter: Produkte, Szenarien, Visionen*; Springer; 1998

[Thrun et al. 98] Thrun, S.; Fox, D. ; Burgard, W.: *Probalistic Mapping Of An Environment By A Mobile Robot*; Proceedings of the 1998 Conference on Robotics & Automation (ICRA); Leuven, Belgium; May 1998

[Weiß et al. 94] Weiß, G.; Wetzler, C.; von Puttkamer, E.: *Track of Position and Orientation of Moving Indoor Systems by Correlation of Range-Finder Scans*; IROS '94; München, Germany; 1994

[Zelinski 94] Zelinski, A.: *Using Path Transforms to Guide the Search for Findpath in 2D*; International Journal of Robotics Research; MIT Press; 1994; Vol. 13, Nr. 4, S. 315 ff

Anhang A

Beispiel einer Szenariodatei

Beispiel einer Szenariodatei

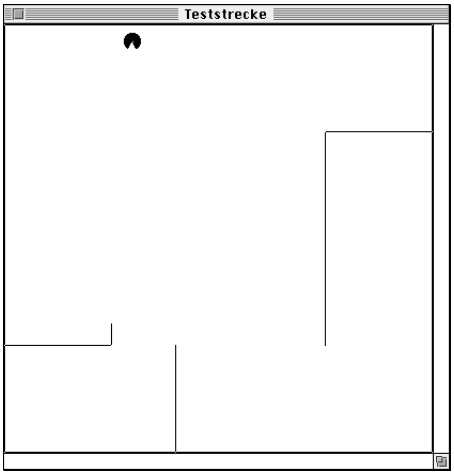
```
# ================================
# Beipielszenario aus Kapitel 5.2
# ================================

# Dimensionen des Feldes
    d   1000 1000  #(10m x 10m)
# Position des Roboters
#       x    y    phi  Radius
    c   700  960  -90  20
    m   0 #(keine weiteren Roboter)

# Anzahl der Wände und Zahl der dynamischen Objekte
    w   9    2

# Anfangs- und Endpunkte der Wände
#   ax  ay   bx   by   Nr. des dyn. Hindernisses
    200 1000 150  800
    0   500  150  650
    0   500  250  500
    400 500  600  500
    750 500  1000 500
    500 500  500  250
    500 0    500  100
    600 500  750  500  1     # Wand gehört zu Hindernis 1
    250 500  400  500  2     # Wand gehört zu Hindernis 2
```

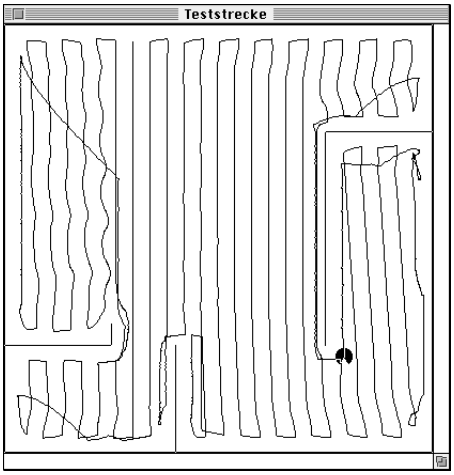
Eine Darstellung dieses Szenarios ist in den Abbildungen 5.17 - 5.21 in Kapitel 5.2 zu sehen.

Anhang B

Simulationsläufe

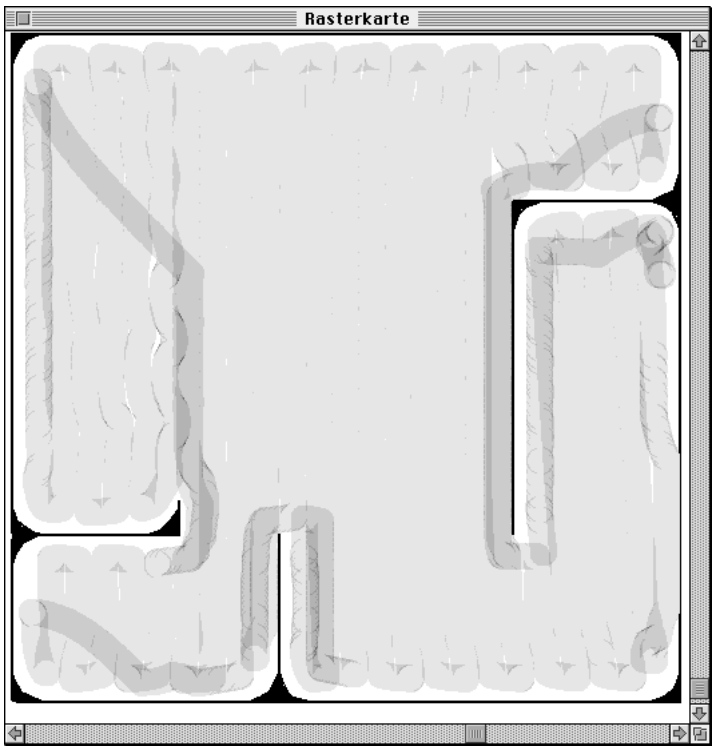
Beispiel T1



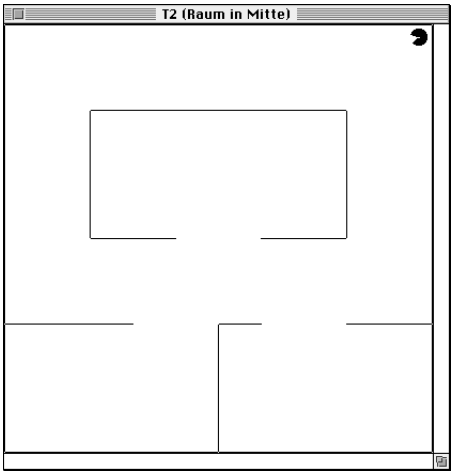


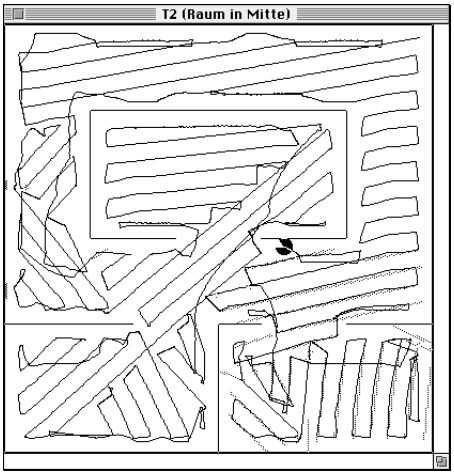
Überfahrene Fläche insgesamt : 883948 cm^2

davon:	1 x befahren	84,0 %
	2 x befahren	13,7 %
	3 x befahren	2,2 %
	4 x und mehr	0,1 %



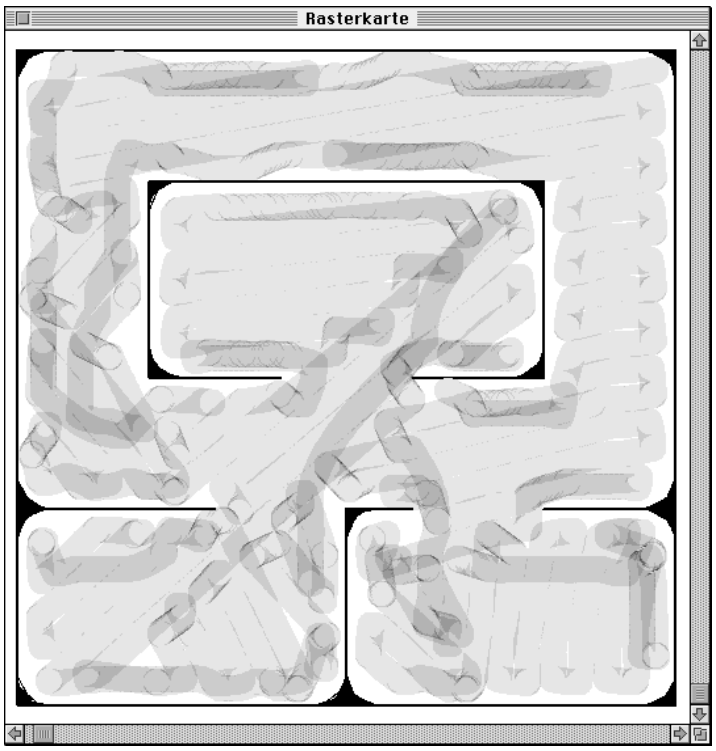
Beispiel T2a



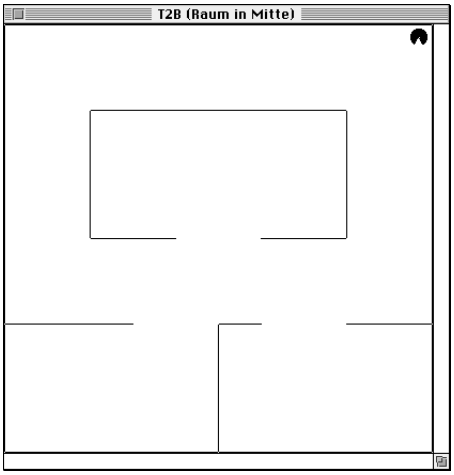


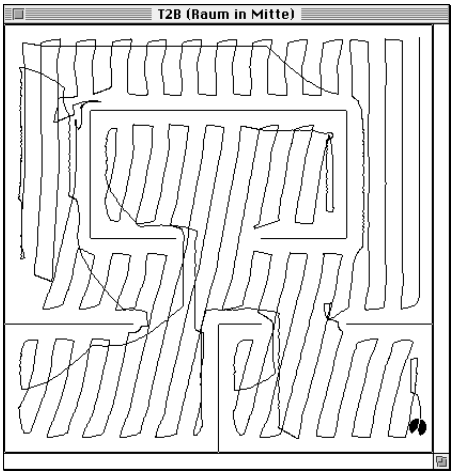
Überfahrene Fläche insgesamt : 830893 cm^2

davon:	1 x befahren	65,9 %
	2 x befahren	27,2 %
	3 x befahren	6,2 %
	4 x und mehr	0,7 %



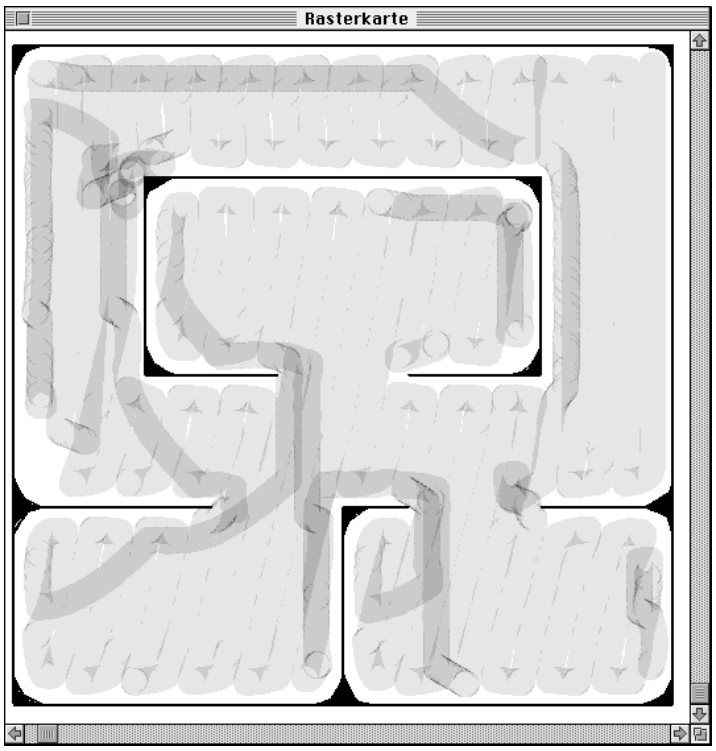
Beispiel T2b



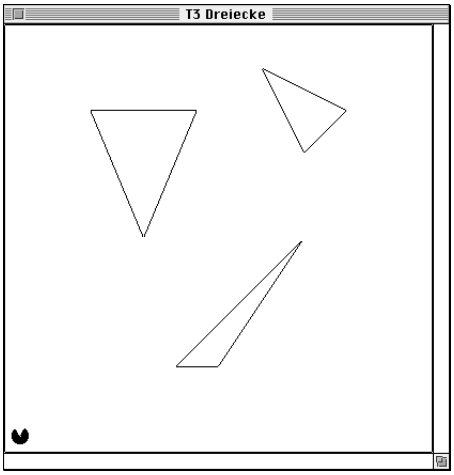


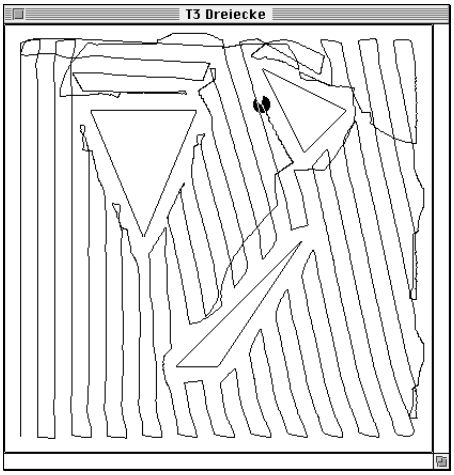
Überfahrene Fläche insgesamt : 830350 cm^2

davon:	1 x befahren	74,7 %
	2 x befahren	21,6 %
	3 x befahren	3,4 %
	4 x und mehr	0,3 %



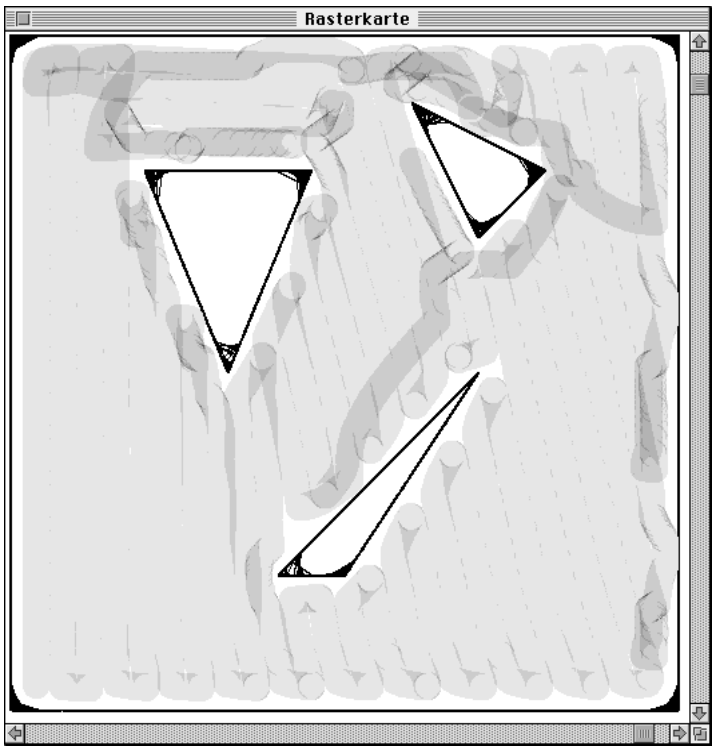
Beispiel T3



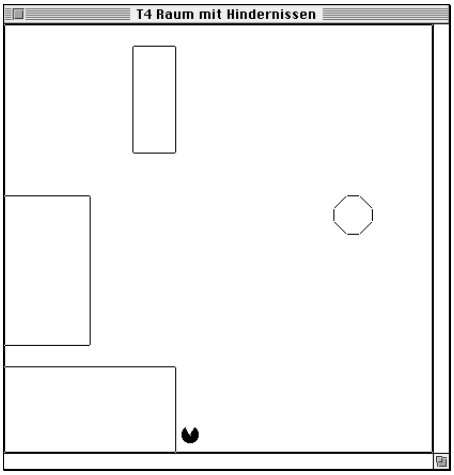


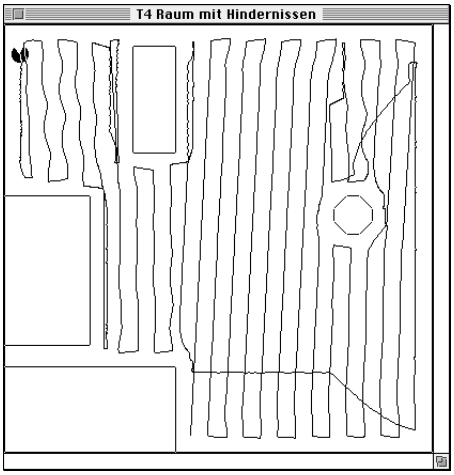
Überfahrene Fläche insgesamt : 817930 cm^2

davon:	1 x befahren	81,9 %
	2 x befahren	15,8 %
	3 x befahren	2,1 %
	4 x und mehr	0,2 %



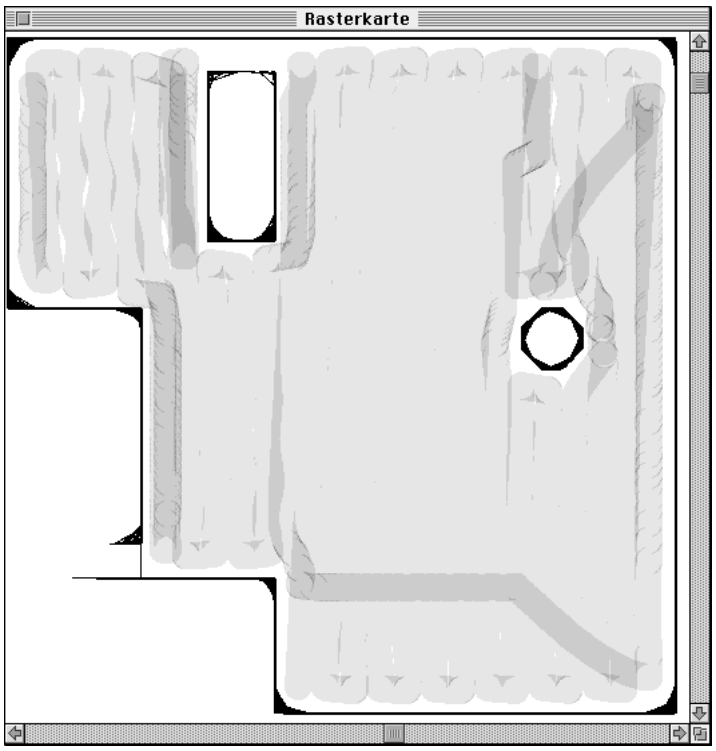
Beispiel T4



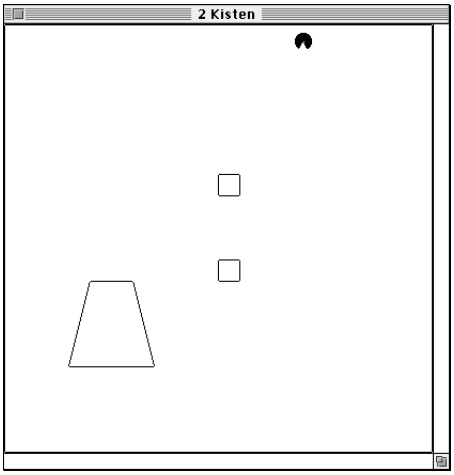


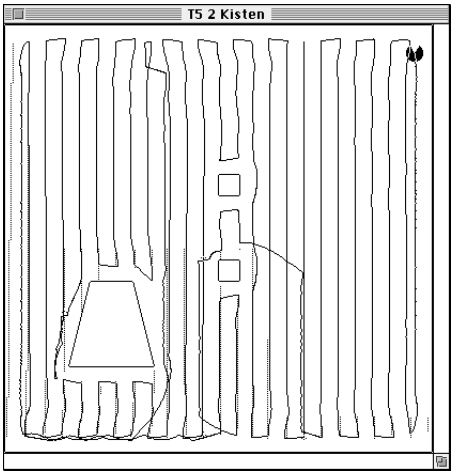
Überfahrene Fläche insgesamt : 711184 cm^2

davon:	1 x befahren	82,2 %
	2 x befahren	15,7 %
	3 x befahren	2,0 %
	4 x und mehr	0,1 %



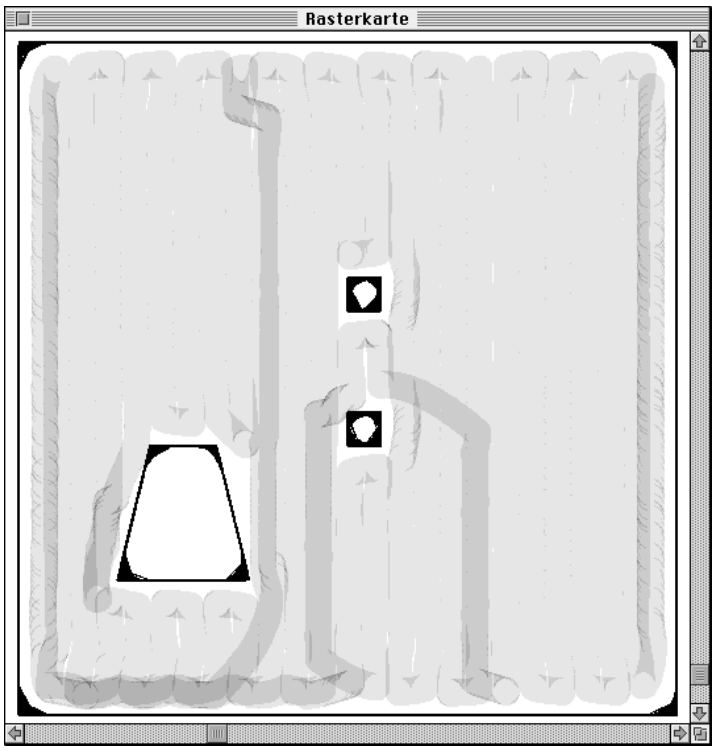
Beispiel T5



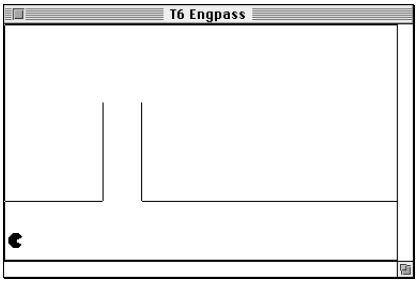


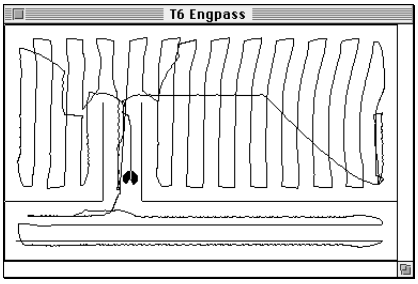
Überfahrene Fläche insgesamt : 864814 cm^2

davon:	1 x befahren	83,2 %
	2 x befahren	15,0 %
	3 x befahren	1,7 %
	4 x und mehr	0,1 %



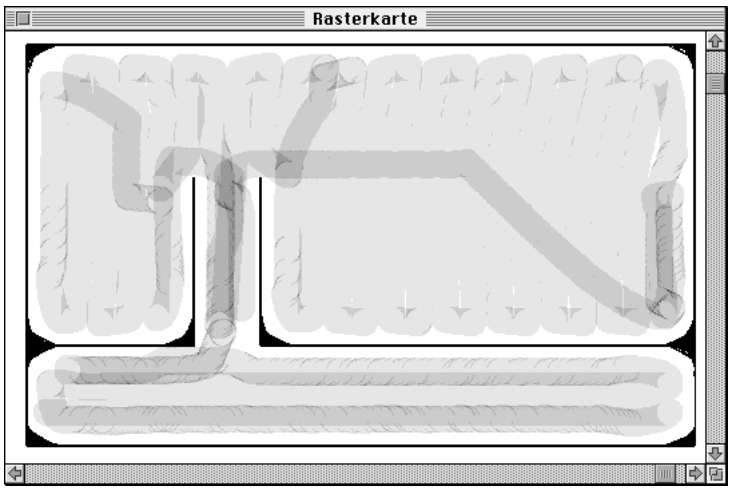
Beispiel T6



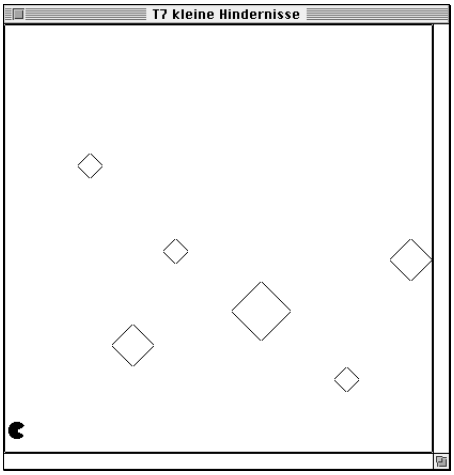


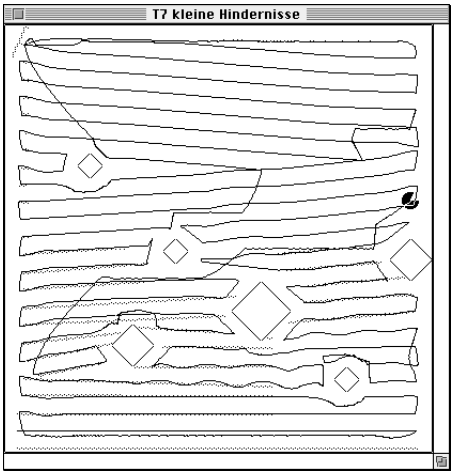
Überfahrene Fläche insgesamt : 484799 cm^2

davon:	1 x befahren	73,4 %
	2 x befahren	22,3 %
	3 x befahren	3,5 %
	4 x und mehr	0,8 %



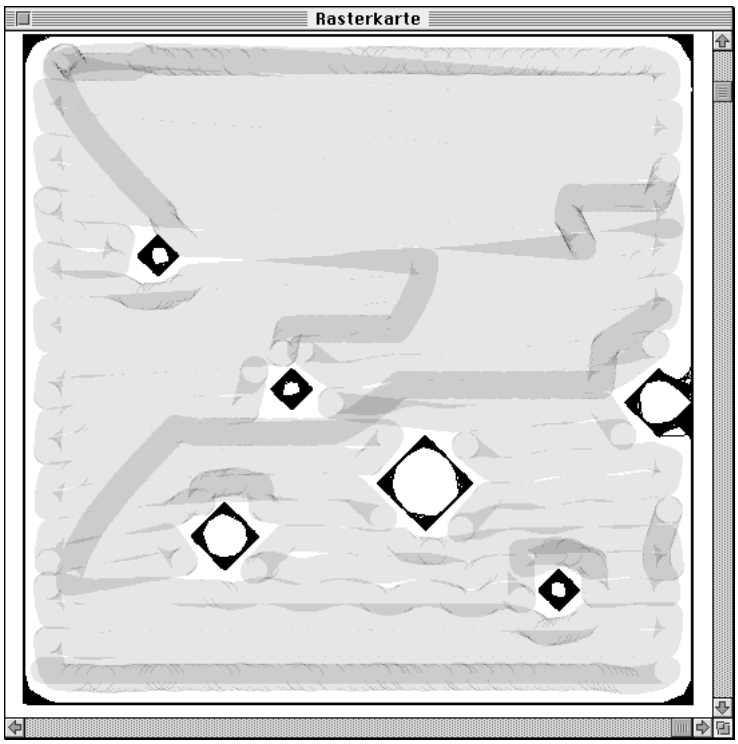
Beispiel T7



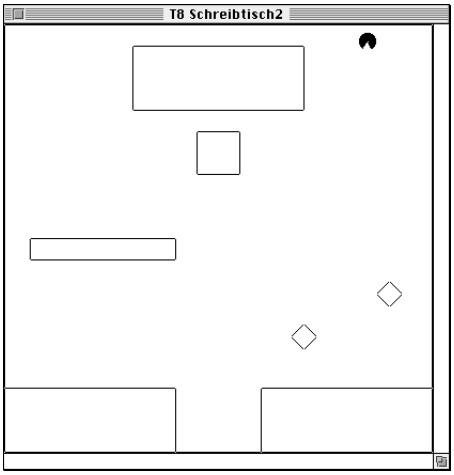


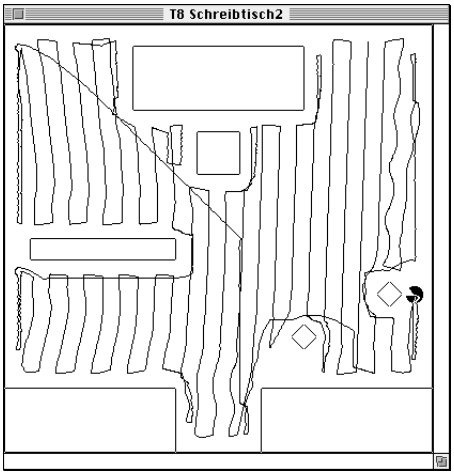
Überfahrene Fläche insgesamt : 864879 cm^2

davon:	1 x befahren	78,6 %
	2 x befahren	20,0 %
	3 x befahren	1,3 %
	4 x und mehr	0,1 %



Beispiel T8





Überfahrene Fläche insgesamt : 650292 cm^2

davon:	1 x befahren	71,1 %
	2 x befahren	23,2 %
	3 x befahren	4,7 %
	4 x und mehr	0,4 %