Fachbereich 4: Informatik

Arbeitsgruppe Aktives Sehen

Bestimmung von Räumen in Gebäudegrundrissen

Studienarbeit
im Studiengang Computervisualistik

vorgelegt von

Kay Kowalski

Betreuer:	Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik
Erstgutachter:	Prof. Dr.-Ing. Dietrich Paulus, Institut für Computervisualistik, Fachbereich Informatik
Zweitgutachter:	Dipl.-Inf. Johannes Pellenz, Institut für Computervisualistik, Fachbereich Informatik

Koblenz, im März 2007

Erklärung

Ich versichere, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Die Richtlinien der Arbeitsgruppe für Studien- und Diplomarbeiten habe ich gelesen und anerkannt, insbesondere die Regelung des Nutzungsrechts

Mit der Einstellung dieser Arbeit in die Bibliothek bin ich einverstanden.	ja ☐	nein ☐
Der Veröffentlichung dieser Arbeit im Internet stimme ich zu.	ja ☐	nein ☐

Koblenz, den .... Unterschrift

Inhaltsverzeichnis

1 Einleitung **11**

2 Raumfindung in Gebäudegrundrissen **15**

2.1 Definitionen 15

2.1.1 Hindernis 15

2.1.2 Raum 15

2.1.3 Durchgang 16

2.2 Vorbereitung 16

2.2.1 Occupancy Grid 16

2.2.2 Kettencodes 17

2.3 Suchen von Kandidaten 18

2.4 Partnersuche 21

2.5 Auswerten der Durchgangspaare 23

2.5.1 Schließen von Türen 24

2.5.2 Erkennen von Räumen 26

3 Implementierung **29**

3.1 Klassenbeschreibung 32

3.1.1 FloodFill 32
3.1.2 ChainAngle 32
3.1.3 DoorFinder 33
3.1.4 GenChainCode 35
3.1.5 RoomDescriptor 37
3.2 Schwellwerte 38

4 Experimente und Ergebnisse 41
4.1 Versuchsaufbau 41
4.2 Ergebnisse 42
4.2.1 Zwischenschritte 42
4.2.2 Verschiedene Grundrisse 42
4.2.3 Schwellwerte 45
4.2.4 Sonderfälle 46
4.2.5 Laufzeit 46
4.3 Bewertung 47

5 Fazit 55

A UML–Diagramme 57

B Quellcode 61

Verzeichnis der Bilder

2.1 Beispiel eines Kettencodes, die gespeicherten Richtungen sind 0, 0, 0, 0, 2, 2, 2, 2, 2, 4, 4, 4, 4, 4, 6, 6, 6, 6, 6, 0 18

2.2 mögliche Richtungen im Kettencode, links: vier mögliche Richtungen, rechts: acht mögliche Richtungen 19

2.3 links: ein Kettencode mit vier Richtungen, rechts: ein Kettencode mit acht Richtungen, beide besitzen den gleichen Startpunkt oberhalb des Hindernisses 19

2.4 Beispiel eines einfachen intuitiven Durchgangs 20

2.5 das gleiche Beispiel wie in 2.4, jedoch hier ohne Vorsprünge an den Wänden 20

2.6 Vorsprünge werden durch Linkskurven, innen liegende Ecken durch Rechtskurven beschrieben 21

2.7 zwei wandumschreibende Kettencodes 23

2.8 der Umfang des zu schließenden Raumes als Entfernung im Kettencode, die umrandeten Pixel entsprechen Tür- und Gegenkandidat 24

2.9 oben links: der ursprüngliche Kettencode, oben rechts: der Kettencode bis zum Türkandidaten, unten links: der Kettencode mit geschlossener Tür, unten rechts: der neue geschlossene Kettencode 25

2.10 links: der ursprüngliche Kettencode, Mitte: der Kettencode von Türkandidat bis Gegenkandidat, rechts: der neue geschlossene Kettencode 26

2.11 oben links: der ursprüngliche Kettencode, oben Mitte: der Kettencode bis zum Türkandidaten, oben rechts: der Kettencode inklusive geschlossener Tür, unten links: der Kettencode mit integriertem wandumschreibenden Kettencode, unten rechts: der neue geschlossene Kettencode 27

3.1 eine Bildschirmaufnahme der grafischen Oberfläche 30

3.2 links: der Kettencode bis zum Türkandidaten, rechts: einzelne Bereiche der Kontur werden doppelt erfasst 34

3.3 Hindernisse befinden sich immer linksseitig des Kettencodes 35

3.4 links: das Originalbild, rechts: die Karte nach dem ersten Schritt der Kettencodeerstellung 36

3.5 Startrichtung der Kettencodes in Schritt 2 der Erstellung 37

4.1 oben links: das Originalbild, oben rechts: sämtliche gefundenen Türkandidaten, unten links: die raumumschreibenden Kettencodes der erkannten Räume, unten rechts: die Visualisierung der Räume 43

4.2 Eine künstlich erstellte Kopie des Grundriss des dritten Stockwerkes des B–Gebäudes der Universität Koblenz, oben: das Originalbild, unten: die Ausgabe des Programms nach der Raumerkennung 44

4.3 links: das von Robbie erstellte Originalbild, rechts: die Visualisierung der gefundenen Räume 45

4.4 links: eine von Robbie erstellte Karte, rechts: die in dieser Karte gefundenen Räume 45

4.5 Künstlich erstellte Grundrisse des dritten Stockwerkes (Bild 1) bzw. des Erdgeschosses (Bild 2) des B–Gebäudes der Universität Koblenz, oben: die Originalbilder, unten: die aus den Karten erstellten Kettencodes 50

4.6 Ein künstlich erstellter Grundriss des Erdgeschosses des B–Gebäudes der Universität Koblenz, oben: das Originalbild, unten: die in der Karte gefundenen Räume. Zu einzelnen freistehenden Hindernissen werden keine Durchgänge geschlossen. 51

A.1 Klassenhierarchie der implementierten Klassen. Grau hinterlegte Klassen entsprechen bereits existierenden Klassen aus PUMA 58

A.2 Klassenhierarchie der implementierten Klassen. Grau hinterlegte Klassen entsprechen bereits existierenden Klassen aus PUMA 59

A.3 Integration der Klassen in die verschiedenen Module in PUMA 59

Kapitel 1

Einleitung

In der autonomen Robotik stellen das Erstellen und das Auswerten von Karten einen wichtigen Teilbereich dar. Hierbei spielen Karten je nach Aufgabe und Umgebung des Roboters eine unterschiedliche Rolle:

- Empfangsrobotor, deren Aufgabe das Begleiten von Gästen zu einem bestimmten Ort ist[1], besitzen bereits eine Karte einer bekannten Umgebung und steuern in dieser bekannte Ziele an.
- Rettungsroboter, die sich in einer „feindseligen“ Umgebung befinden, wie z. B. auf dem Mars[2], unter Wasser[3] oder in eingestürzten Bürogebäuden[4], müssen eine Karte zur Navigation während der Umgebungserkundung selbst erstellen.

Neben den hier genannten Beispielen sind noch viele weitere Szenarien denkbar, in denen sich der Einsatz der evtl. noch zu erstellenden Karte nach dem Verwendungszweck des jeweiligen Roboters richtet. Dem entsprechend existieren in der Robotik zu verschiedenen Aufgaben der Kartographie unterschiedliche Schwerpunkte in der Forschung. Vor allem

[1] `http://www.uni-koblenz.de/~agas/lehre/ss03/robbie.html`

[2] vgl. die Marsmission „Pathfinder“ [Gol99]

[3] vgl.[GGH+04]

[4] vgl. `http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/Researches/ActiveVision/Robbie-6`

auf dem Gebiet der „simultanen Lokalisation und Kartographie“ (SLAM) [Mon03], welche in unbekannten Umgebungen eingesetzt wird, findet eine syntaktische Auswertung der bisher vorhandenen Kartenteile bereits während der Erstellung statt. Dies geschieht, um die Genauigkeit der Karte zu verbessern (vgl. [SHB04]). Aber auch in anderen Bereichen der Robotik wie der autonomen Wegfindung (siehe [Yam97]) werden Karten analysiert, hier mit dem Ziel, einen hindernisfreien Weg zwischen zwei Punkten in der Karte zu finden.

Ein von diesen Betrachtungen abgetrennter Bereich hingegen ist die semantische Auswertung von Karten. Für das Auslesen semantischer Informationen gibt es bereits verschiedene Ansätze. Die Unterteilung von verschiedenen Bereichen einer Karte, eine relativ geringe Form der Abstraktion, stellt hierbei einen ersten Schritt dar. Dies umfasst sowohl die Bestimmung von Räumen in Gebäudegrundrissen als auch von Plätzen, Gebäuden und Straßen in Stadtkarten oder die Unterscheidung zwischen Bergen und Tälern in topographischen Landkarten. Beipiele für eine solche Unterteilung finden sich in [PGKKP] oder bei [MMSB05].

In der erstgenannten Veröffentlichung werden in Gebäudegrundrissen, welche als „Occupancy Grid“ vorliegen, Räume von anderen durch Rechteckstraversierung abgetrennt. In dieser Repräsentation der Umgebung wird zu jedem Punkt in einer Karte gespeichert, ob dieser durch ein Hindernis belegt ist. In der vorliegenden Karte wird der erste „freie“ Punkt gesucht, ab dem ein Rechteck aufgespannt wird. Dieses Rechteck wird in die jeweilige Richtung durch evtl. vorkommende Hindernisse begrenzt. Hierdurch entsteht eine Abhängigkeit von der Form der Räume, da Gebäudeteile mit einem nicht recktwinkligen Grundriss nicht vollständig erfasst werden, sondern immer jeweils rechtwinklige Teilbereiche dieser Fläche. Ebenfalls können Gebäudeteile mit rechtwinkligem Grundriss, welche jedoch nicht orthogonal zur Ausrichtung der Karte vorliegen, nicht vollständig erfasst werden.

In [MMSB05] werden Räume als Teil eines Gebäudes definiert, welche nur durch Durchgänge von anderen Gebäudeteilen abgegrenzt sind. Diese Durchgänge werden mit Hilfe von Laserreferenzscans bereits während der Erstellung der Karte durch einen Roboter bestimmt. Die Erkennung von Durchgängen findet teilweise überwacht („supervised“) statt, mit dem Ziel, den Roboter ab einer bestimmten Menge an vorliegenden Referenzdaten die

Durchgangsbestimmung selbständig vornehmen zu lassen. Dieses Verfahren wurde bereits erfolgreich getestet, stößt jedoch bei bisher unbekannten Durchgangsformen an seine Grenzen (z. B. Schiebetüren, die, da sie in eine Wand verschoben werden können, auf einer Seite in einem Laserscan als einfache Wand erfasst werden). Außerdem spielt die Reichweite des Laserscanners hierbei eine entscheidende Rolle: Durchgänge, welche auf Grund ihrer Breite nicht vollständig durch einen Laserscan an einem Punkt erfasst werden können (z. B. Tore bei Lagerhallen), werden nicht als solche erkannt.

Aber auch andere Ansätze höherer semantischer Auswertung wurden bereits implementiert: in [GSC^{+}05] werden verschiedene Gebäudeteile über einen Watershed–Algorithmus[5] von einander abgetrennt. Innerhalb dieser Räume werden Objekte (Hindernisse), welche mit Hilfe einer Farbkamera aufgenommen wurden, mit vorliegenden Bildern von Referenzobjekten verglichen. Anhand dieser Objekte findet dann eine autonome Klassifizierung der Räume statt. So werden Räume, in denen sich ein Objekt befindet, welches mit einem Referenzobjekt „Sofa“ vergleichbar ist, als „Wohnzimmer“ klassifiziert und ein Raum ohne ein „Sofa“ jedoch mit einem vergleichbaren Objekt „Kaffeemaschine“ als „Küche“ eingeordnet.

Auch in [MMSB05] wurde noch eine weitere Art der semantischen Auswertung einer Karte vorgestellt: an Hand der Form eines Gebäudeteils wird dieser als „Raum“ oder als „Korridor“ eingestuft. Auch diese Unterscheidung findet z. T. überwacht und mir Hilfe von Referenzdaten statt.

In dieser Arbeit wird eine neue Methode präsentiert, mit der Räume in Gebäudegrundrissen bestimmt werden können. Diese definiert ebenfalls Räume als Gebäudeteile, welche über Durchgänge mit anderen Gebäudeteilen verbunden sind. Allerdings werden keine Referenzdaten zur Findung dieser Durchgänge verwendet, sondern der vorliegende Grundriss wird in verschiedene Repräsentationsformen umgewandelt („Occupancy Grid“ und „Chain Codes“, vgl. Kapitel 2.2). In diesen Repräsentationen der Karte werden zunächst Kandidaten und Gegenkandidaten für Durchgänge gesucht (Kapitel 2.3, 2.4). Aus diesen Paaren von Durchgangskandidaten und Gegenkandidaten wird im nächsten Schritt das „beste“ Paar ausgewählt und der Durchgang an dieser Stelle virtuell geschlossen (Kapitel 2.5). Dieser Algorithmus wird rekursiv durchlaufen, bis keine weiteren Durchgangskan-

[5] siehe hierzu [DVG94]

didaten in der Karte mehr gefunden werden. Diese Form der Abgrenzung von Objekten wurde bereits erfolgreich in [BEP+01] zur Trennung von Objekten auf einem Fließband verwendet.

In Kapitel 3 wird die gewählte Implementationsform inklusive der Einbettung in bestehende Projekte beschrieben sowie auf einzelne Sonderfälle und Besonderheiten während der Umsetzung eingegangen.

Anschließend werden im Kapitel 4 verschiedene Karten und die jeweiligen Ergebnisse der Raumfindung vorgestellt. Zudem wird hier die Laufzeit des Programms unter der Berücksichtigung eines Einsatzes in aktuellen mobilen Robotiksystemen vorgestellt.

Den Abschluß der Arbeit bildet mit Kapitel 5 der Überblick über die erreichte Stufe der semantischen Auswertung sowie deren Unterschied zu den bereits vorgestellten Arbeiten. Außerdem gibt Kapitel 5 einen Ausblick über mögliche weiterführende Forschungsgebiete.

Kapitel 2

Raumfindung in Gebäudegrundrissen

2.1 Definitionen

Wie bereits in Kapitel 1 beschrieben, stellt das Unterscheiden und das Trennen von Gebäudeteilen, insbesondere von Räumen, das Ziel der angestrebten semantischen Auswertung dar. Hierzu sind dem entsprechend folgende Definitionen nötig:

2.1.1 Hindernis

Als Hindernis werden in einer Karte eines Gebäudegrundrisses alle Bereiche erachtet, welche keine freie Fläche beschreiben (vgl. Kapitel 2.2.1).

2.1.2 Raum

Ein Raum ist ein Teil eines Gebäudes, der in einem Gebäudegrundriss als freie Fläche repräsentiert wird, durch Hindernisse begrenzt ist und nur durch einen oder mehrere Durchgänge mit anderen Gebäudeteilen verbunden sein kann. Zusätzlich können innerhalb eines Raumes weitere Hindernisse vorhanden sein. Jeder Gebäudeteil kann hierbei jeweils nur Teil eines Raumes sein. Die Form des Gebäudeteils spielt hierbei keine Rolle.

2.1.3 Durchgang

Ein Durchgang (im Folgenden auch „Tür“ genannt) besteht aus zwei Teilen: einem Türkandidaten und einem dazugehörigen Gegenkandidaten. Alle Tür- und Gegenkandidaten befinden sich in unmittelbarer Nähe zu einem Hindernis auf der Karte. Hierbei muss es sich für ein Paar aus Türkandidat und Gegenkandidat nicht um das selbe Hindernis handeln. Auf der Strecke zwischen beiden Punkten darf jedoch kein Hindernis vorhanden sein. Die direkte Entfernung von Kandidat zu Gegenkandidat muss signifikant geringer (vgl. Kapitel 3.2) sein als der Umfang der gesamten freien Fläche, welche sich z. T. zwischen den beiden Punkten befindet.

2.2 Vorbereitung

Die vorliegende Karte wird in zwei Datenstrukturen umgewandelt: einem zweidimensionalen „Occupancy Grid“ und in einer Menge von „Chain Codes“. Die beiden Strukturen werden im Folgenden näher beschrieben.

2.2.1 Occupancy Grid

Ein Occupancy Grid („OG“) beschreibt zu jedem Punkt in einer Karte, ob dieser belegt („occupied“) ist[1]. Das klassische OG besteht hierbei aus einer einfachen zweidimensionalen Matrix, in der zu jedem Punkt der Karte der binäre Zustand („belegt“ oder „frei“) gespeichert wird. Da verschiedene Gegenstände in der Realität die kartographierten Bereiche nur zum Teil belegen, sind jedoch auch OGs möglich, in der der Grad der Belegung nicht binär, sondern z. B. als Wert zwischen 0 und 255 gespeichert wird.

Zu den beiden genannten Varianten von OGs sind für die Speicherung weitere Modifikationen möglich. In [Ben75] wurde bereits ein Verfahren vorgestellt, welches durch das Strukturieren der Daten zu mehrdimensionalen Bäumen den benötigten Aufwand bei der Datensuche reduzierten. Diese Methode lässt sich auch auf ein OG anwenden: in einem

[1] vgl. [Elf89]

„kd-tree“ wird zu jedem Feld in der Matrix ein mehrdimensionaler Baum gespeichert. Die Wurzel des Baumes stellt hierbei die größte zusammenhängende Fläche dar, welche als „belegt“ oder als „frei“ gespeichert werden kann. Besteht eine solche Fläche sowohl aus belegten als auch aus freien Teilflächen, wird diese in zwei gleich große Fächen unterteilt. Diese Flächen werden erneut auf ihren Status überprüft und als Kinder im binären Baum eingegliedert. Die maximale Tiefe des Baumes entspricht somit der maximalen Detailstufe der Karte.

Eine besondere Form dieser Zusammenfassung gleichartiger Bereiche sind „Quadtrees“ [Sam84]. Hier werden die untersuchten Flächen, in denen eine unterschiedliche Belegung der Teilflächen vorliegt, in vier gleich große Segmente unterteilt. Quadtrees stellen somit eine Spezialform der kd-trees dar. Auch für kd-trees oder für Quadtrees besteht die Möglichkeit, den Grad der Belegung eines Bereiches genauer zu erfassen [KPGU04].

Zur Verarbeitung der Karte wurde in dieser Arbeit ein einfaches binäres OG gewählt, da dieses sowohl für durch Roboter erstellte Karten (vgl. Kapitel 4) als auch für Gebäudegrundrisse anderer Quellen (z. B. architektonische Grundrisse) durch Binarisierung des Eingabebildes[2] erstellt werden kann.

2.2.2 Kettencodes

Kettencodes („Chain Codes“) sind eine effiziente Möglichkeit, Konturen von Objekten zu speichern [Fre74]. Hierbei wird ab einem Startpunkt, welcher sich an einer Kante des Objektes befindet, die Kontur des Objektes „abgefahren“ und die jeweilige Richtung gespeichert. Ein Beispiel findet sich in Bild 2.1. Dabei kann die Abarbeitung der Kontur sowohl auf als auch neben der Kante des Objektes erfolgen. Unterschieden wird zwischen Kettencodes mit acht oder mit vier möglichen Richtungen (vgl. Bild 2.2). Da ein Kettencode mit acht Richtungen die Konturen der zu umschreibenden Objekte genauer repräsentiert, können Pixelfehler oder ähnliche Ungenauigkeiten zu einer Unterteilung eines Objektes in mehrere Teilkonturen führen (vgl. Bild 2.3).

Für die hier vorliegende Arbeit wurde daher ein Kettencode mit vier Richtungen gewählt.

[2] vgl. Kapitel 3

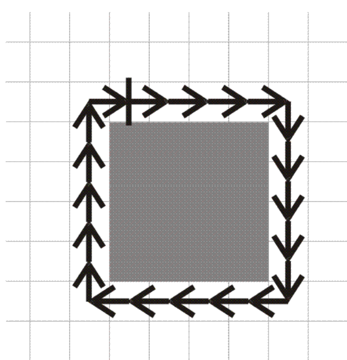


Bild 2.1: Beispiel eines Kettencodes, die gespeicherten Richtungen sind 0, 0, 0, 0, 2, 2, 2, 2, 2, 4, 4, 4, 4, 4, 6, 6, 6, 6, 6, 0

Die Startpunkte und die jeweiligen Richtungen wurden dabei so angepasst, dass sich der Kettencode immer auf der rechten Seite, also neben dem zu umschreibenden Objekt befindet. Durch diese Rahmenbedingungen ergeben sich die beiden folgenden Arten von Kettencodes: *raumumschreibende* und *wandumschreibende*. Während *raumumschreibende* Kettencodes eine freie Fläche umschließen, umranden *wandumschreibende* ausschließlich Hindernisse. Als Indikator zur Unterscheidung dient hierbei die eingegrenzte Fläche. Durch den gewählten Algorithmus zur Flächenberechnung in Kettencodes ergibt sich für die Fläche folgende Eigenschaft: ein Kettencode, dessen berechneter Flächeninhalt positiv ist, umschließt eine freie Fläche; ein Kettencode mit berechnetem negativem Flächeninhalt deutet auf einen wandumschreibenden Kettencode hin. Eine detailierte Beschreibung der Methode zur Flächenberechnung in Kettencodes findet sich in Kapitel 3.1.5.

2.3 Suchen von Kandidaten

In Kapitel 2.1.3 wurde ein Durchgang durch ein Paar von Türkandidat und dazugehörigem Gegenkandidat definiert. Zusätzlich wurde eine signifikant geringe Entfernung zwischen diesen beiden Punkten gefordert. Diese tritt jedoch nur an Vorsprüngen und Ecken in Hindernissen auf. Ein Hindernis, welches keine Vorsprünge, Vertiefungen oder Ecken besitzt, beschreibt somit eine glatte, gleichmäßige Kontur. Dies widerspricht jedoch der intuiti-

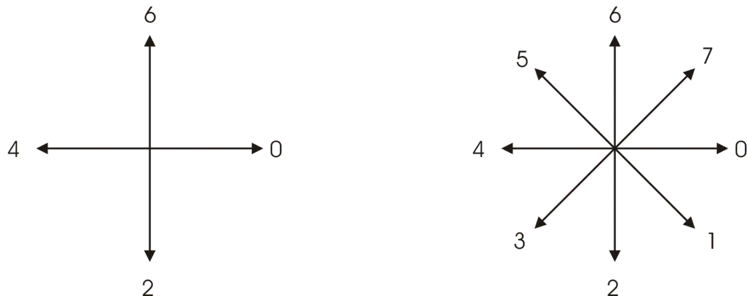


Bild 2.2: mögliche Richtungen im Kettencode, links: vier mögliche Richtungen, rechts: acht mögliche Richtungen

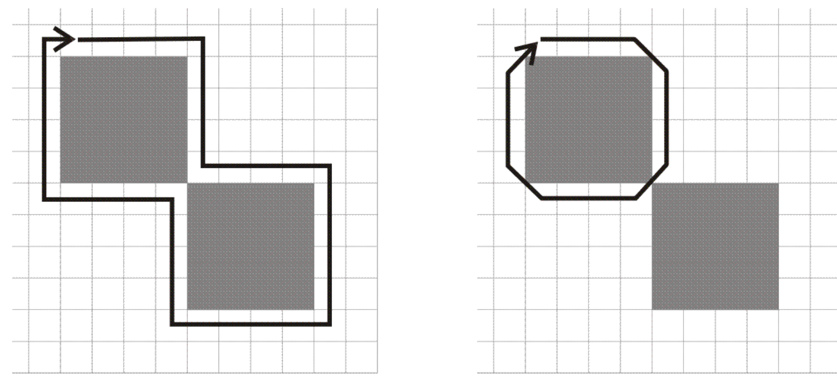


Bild 2.3: links: ein Kettencode mit vier Richtungen, rechts: ein Kettencode mit acht Richtungen, beide besitzen den gleichen Startpunkt oberhalb des Hindernisses

von Vorstellung eines Durchgangs: einer „Verengung" zwischen zwei Hindernissen (oder innerhalb des selben Hindernisses). Der in Bild 2.4 vorhandene der Durchgang, welcher durch die beiden Vorsprünge in dem eingezeichneten Hindernis erkennbar ist, ist in Bild 2.5, in der diese Vorsprünge nicht vorhanden sind, nicht mehr vorhanden. Daher wurde für die Suche nach Kandidaten die Existenz von Vorsprüngen oder Ecken in Hindernissen als Kriterium gewählt.

Da die in Kapitel 2.2.2 beschriebenen Kettencodes die Eigenschaft besitzen, sich rechts neben den von Ihnen umschriebenen Objekten zu befinden, existiert an jedem Vorsprung und an jeder außen liegenden Ecke eines Objektes eine Linkskurve in seiner Kontur (vgl. Bild 2.6). Rechtskurven entstehen hingegen an innen liegenden Ecken eines Objektes (vgl.

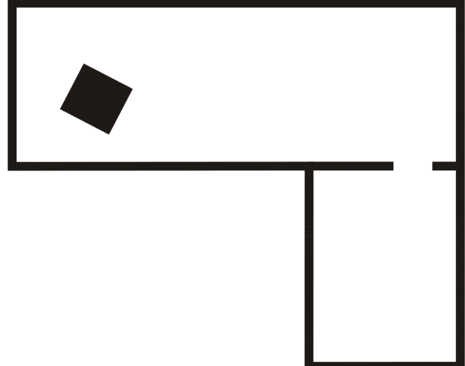


Bild 2.4: Beispiel eines einfachen intuitiven Durchgangs

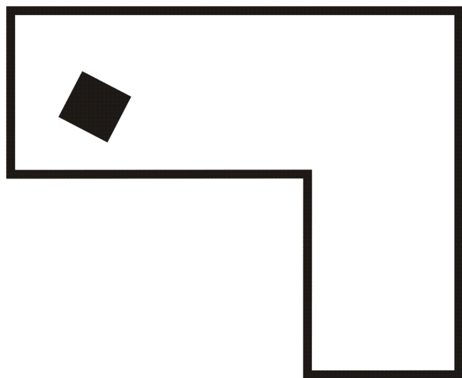


Bild 2.5: das gleiche Beispiel wie in 2.4, jedoch hier ohne Vorsprünge an den Wänden

Bild 2.6), sie sind für die Türkandidatensuche uninteressant.

Zur Suche nach Türkandidaten werden alle erstellten Kettencodes einzeln elementweise untersucht. Um die aktuelle Richtungänderung auszuwerten, werden vom aktuellen Element ausgehend zwei Vektoren aufgespannt: ein Vektor zu dem n Elemente vorher gespeicherten Punkt und ein Vektor zu dem n Elemente nach dem aktuellen Element im Kettencode gespeicherten Punkt. Zwischen diesen beiden Vektoren wird der rechtsseitige Winkel α berechnet. Gilt für diesen Winkel

$$\alpha > \alpha_0$$

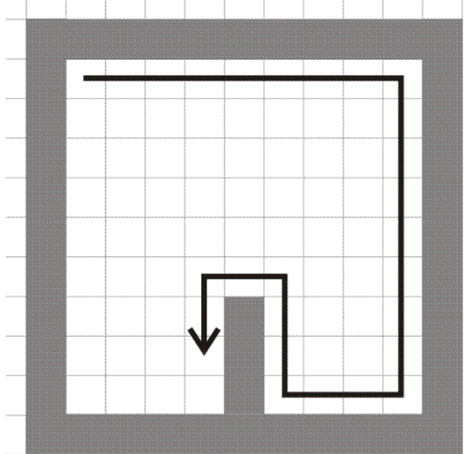


Bild 2.6: Vorsprünge werden durch Linkskurven, innen liegende Ecken durch Rechtskurven beschrieben

so befindet sich an der aktuellen Stelle im Kettencode eine Linkskurve. α_0 beschreibt hierbei den Schwellwert, ab dem eine Richtungsänderung als Linkskurve angesehen wird. Nähere Informationen zu den verwendeten Schwellwerten findet sich in Kapitel 3.2. Befindet sich an der aktuell betrachteten Stelle im Kettencode eine Linkskurve, so ist der aktuelle Punkt, ab welchem die beiden Vektoren zur Winkelberechnung aufgespannt wurden, ein Türkandidat.

2.4 Partnersuche

Zu jedem gefundenen Türkandiat[3] wird in einem nächsten Schritt ein Gegenkandidat gesucht. Diese können sich sowohl in dem gleichen Kettencode befinden, in welchem auch der Türkandidat gefunden wurde, als auch in einem der von diesem Kettencode eingeschlossenen wandumschreibenden Kettencodes (vgl. Bild 2.7). Für Türkandidaten, welche sich in wandumschreibenden Kettencodes befinden, werden somit nur Gegenkandidaten in diesem Kettencode gesucht. Türen zwischen zwei wandumschreibenden Kettencodes

[3] vgl. Kapitel 2.3

oder zwei raumumschreibenden Kettencodes können durch diese Einschränkung nicht geschlossen werden. Ein Beispiel für diese nicht zu schließenden Durchgänge findet sich in Kapitel 4.2.

Obwohl sich ein Gegenkandidat im Gegensatz zu Türkandidaten an jeder Stelle in einem der genannten Kettencodes befinden kann, müssen folgende Bedingungen erfüllt sein:

- es darf sich kein Hindernis zwischen Kandidat und Partner im Occupancy Grid befinden (vgl. Kapitel 2.2.1 und 2.1.3)
- die Entfernung zwischen Kandidat und Gegenkandidat darf ein gewisses Maß d_{max}, die *maximale Türbreite*, nicht überschreiten (vgl. Kapitel 2.1.3)
- das Verhältnis von Umfang U des zu umschließenden Raumes zu direkter Entfernung d der beiden Punkte muss größer sein als der hierfür angegebene Schwellwert r_0:

$$\frac{U}{d} > r_0$$

Als Wert für den zu berechnenden Umfang wurde die Entfernung von Türkandidat zu Gegenkandidat im Kettencode gewählt (siehe Bild 2.8). Da dieser die Kontur des raumumschließenden Hindernisses beschreibt, entspricht diese Entfernung dem Umfang des Raumes. Außerdem ist durch die Nummerierung der Elemente eines Kettencodes die Entfernung zweier Punkte entlang der beschriebenen Kontur ein automatisches Nebenprodukt dieser Repräsentationsform.

Diese einfache Lösung zur Berechnung des Umfangs stößt jedoch auf Probleme, wenn sich Türkandidat und Gegenkandidat nicht im selben Kettencode befinden. Da keine Verbindung zwischen den jeweiligen Kettencodes besteht, kann anhand dieser nicht die Entfernung berechnet werden. Ein passendes Maß für den Umfang U, welches sich im Laufe der Implementierung durch Experimente ergeben hat (vgl. Kapitel 4), stellt folgende Formel dar:

$$U = \frac{1}{4} * (l_0 + l_1)$$

Hierbei entspricht l_i der Gesamtlänge des jeweiligen Kettencodes. Da somit für jeden Gegenkandidaten, welcher sich in einem anderen Kettencode befindet als sein jeweiliger Tür-

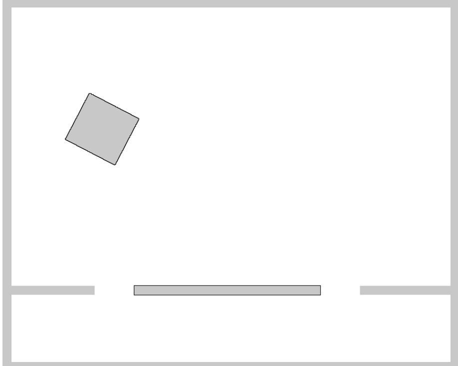


Bild 2.7: zwei wandumschreibende Kettencodes

kandidat, der Umfang U des Raumes gleich ist, kann dieser Wert nur als Anhaltspunkt gesehen werden.

Zu jedem Türkandidaten wird ein Gegenkandidat gespeichert, welcher die o. a. Bedingungen erfüllt. Sollten hierfür mehrere Gegenkandidaten in Frage kommen, so wird jener gespeichert, welcher das „beste“ Verhältnis $\frac{U}{d}$ besitzt, also für welchen $\frac{U}{d}$ den höchsten Wert annimmt.

2.5 Auswerten der Durchgangspaare

Durch die Bedingungen, welche für die Suche nach Gegenkandidaten gelten[4], existieren evtl. Türkandidaten, zu denen kein Gegenkandidat gefunden werden kann. Da diese Kandidaten somit keinen Durchgang beschreiben können, werden diese nun aus der Liste der Türkandidaten entfernt. Hierdurch werden im Folgenden nur noch Paare von Kandidaten und zugehörigem Gegenkandidaten betrachtet, welche der Definition eines Durchgangs[5] entsprechen.

Allerdings hat das Löschen von Türkandidaten zur Folge, dass in dem aktuell betrachteten Kettencode evtl. keine Kandidaten mehr vorhanden sind und somit auch keine möglichen

[4] vgl. Kapitel 2.4
[5] vgl. Kapitel 2.1.3

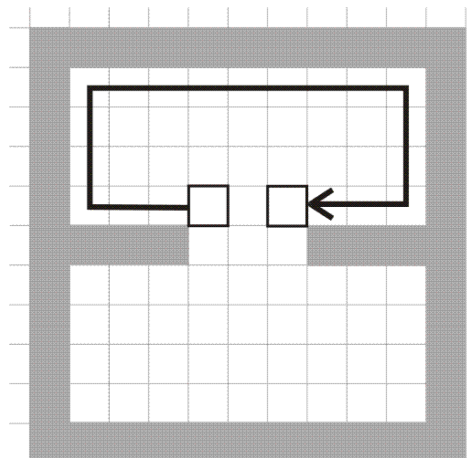


Bild 2.8: der Umfang des zu schließenden Raumes als Entfernung im Kettencode, die umrandeten Pixel entsprechen Tür- und Gegenkandidat

Durchgänge in diesem Gebäudeteil existieren. Daher wird an dieser Stelle für den weiteren Verlauf unterschieden, ob Paare aus Türkandidat und Gegenkandidat vorhanden sind, oder ob alle Türkandidaten gelöscht wurden.

2.5.1 Schließen von Türen

Existieren nach dem Löschen einzelner Türkandidaten weitere Paare aus Kandidaten und Gegenkandidat, so werden diese nach jenem Paar durchsucht, welches für das Schließen eines Durchganges am Besten geeignet ist. Diese Suche entspricht dem Vergleich einzelner Gegenkandidaten in Kapitel 2.4: es wird jenes Paar gewählt, welches den höchsten Wert für das Verhältnis $\frac{U}{d}$ von Umfang U zu direkter Entfernung d der beiden Punkte hat.

Dieses Paar wird verwendet, um den Kettencode an dieser Stelle zu schließen. Hierzu werden zwei neue Kettencodes erstellt: zum Einen wird der ursrpüngliche Kettencode an der Stelle des Türkandidaten unterbrochen, eine per Bresenham–Algorithmus [Bre63] erzeugte Linie von Türkandidat zu Gegenkandidat dem Kettencode hinzugefügt und ab der Stelle des Gegenkandidaten die ursprüngliche Kontur des Kettencodes wieder aufgenommen (siehe Bild 2.9). Zum Anderen wird ein Kettencode erstellt, der an der Stelle des

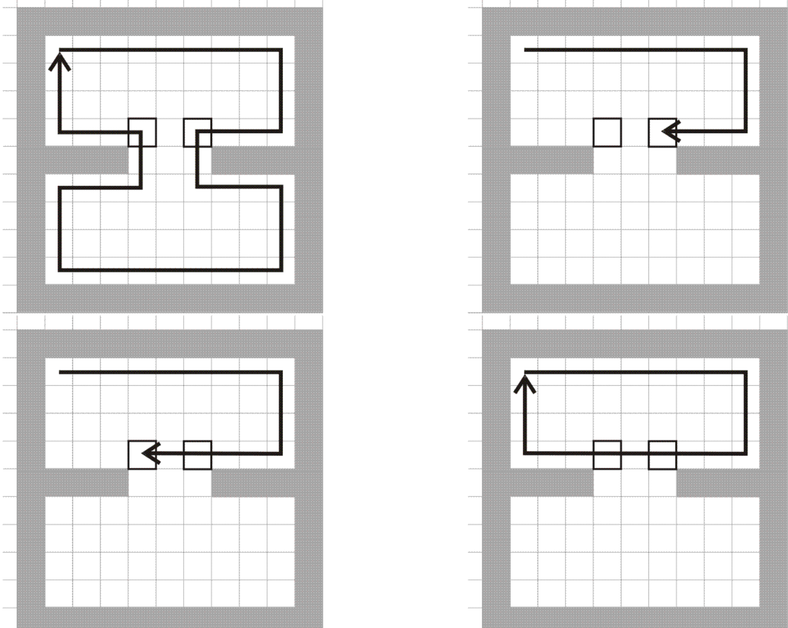


Bild 2.9: oben links: der ursprüngliche Kettencode, oben rechts: der Kettencode bis zum Türkandidaten, unten links: der Kettencode mit geschlossener Tür, unten rechts: der neue geschlossene Kettencode

Türkandidaten der Kontur des ursprünglichen Kettencodes bis zur Stelle des Gegenkandidaten folgt, und von dort die o. g. neu erzeugte Linie in entgegengesetzter Richtung bis zum Türkandidaten hinzufügt (siehe Bild 2.10).

Befinden sich Türkandidat und Gegenkandidat in verschiedenen Kettencodes, so findet das Erstellen eines neuen Kettencodes nach einem ähnlichen Muster statt: der zu erstellende Kettencode beginnt wie oben beschrieben an der gleichen Stelle wie der ursprüngliche Kettencode (als ursprünglicher Kettencode dient hierbei jener, in welchem der Türkandidat enthalten ist) und ist zu diesem bis zur Stelle des Türkandidaten identisch. Ab diesem Punkt wird ebenfalls wieder eine nach dem Bresenham–Algorithmus [Bre63] erstellte Linie dem Kettencode hinzugefügt. Da sich der Gegenkandidat jedoch in einem anderen wandumschreibenden Kettencode befindet, wird dieser nun ab der Stelle des Gegenkandidaten bis zu seinem Endpunkt (welcher durch die Geschlossenheit aller Kettencodes auch

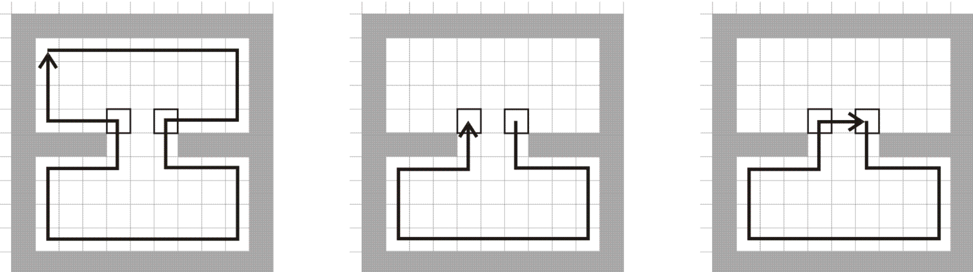


Bild 2.10: links: der ursprüngliche Kettencode, Mitte: der Kettencode von Türkandidat bis Gegenkandidat, rechts: der neue geschlossene Kettencode

dessen Startpunkt ist, siehe Kapitel 2.2.2) und ab diesem bis zur Stelle des Gegenkandidaten dem neu erstellten Kettencode hinzugefügt. Als nächstes wird die o. g. Linie von Gegenkandidat zu Türkandidat dem Kettencode angefügt und dieser mit der Kontur des urspünglichen Kettencodes von Türkandidat zu dessen Endpunkt vervollständigt und geschlossen. Somit beschreibt der neu erstellte Kettencode eine Kontur, welche der Integration des wandumschreibenden Kettencodes in die Kontur des ursprünglichen Kettencodes an der Stelle des Türkandidaten entspricht. Eine Verdeutlichung hiervon findet sich in Bild 2.11.

Diese nach einer der genannten Methoden neu erstellten Kettencodes haben wiederum sämtliche Eigenschaften, welche in Kapitel 2.2.2 beschrieben wurden. Somit können sie verwendet werden, um diese rekursiv nach Durchgängen zu untersuchen.

2.5.2 Erkennen von Räumen

Sind nach dem Löschen von Türkandidaten, welches in Kapitel 2.5 beschrieben wurde, keine Türkandidaten und Gegenkandidaten mehr vorhanden, handelt es sich bei dem vorliegenden Kettencode um einen Gebäudeteil, in welchem keine Durchgänge existieren. Somit muss an dieser Stelle geprüft werden, ob es sich bei der betrachteten Kontur um einen wandumschreibenden oder einen raumumschreibenden Kettencode handelt. Hierzu wird die eingeschlossene Fläche nach einem Verfahren berechnet, welches für eingeschlossen freie Flächen deren Flächeninhalt ausgibt, für umschriebene Hindernisse ohne

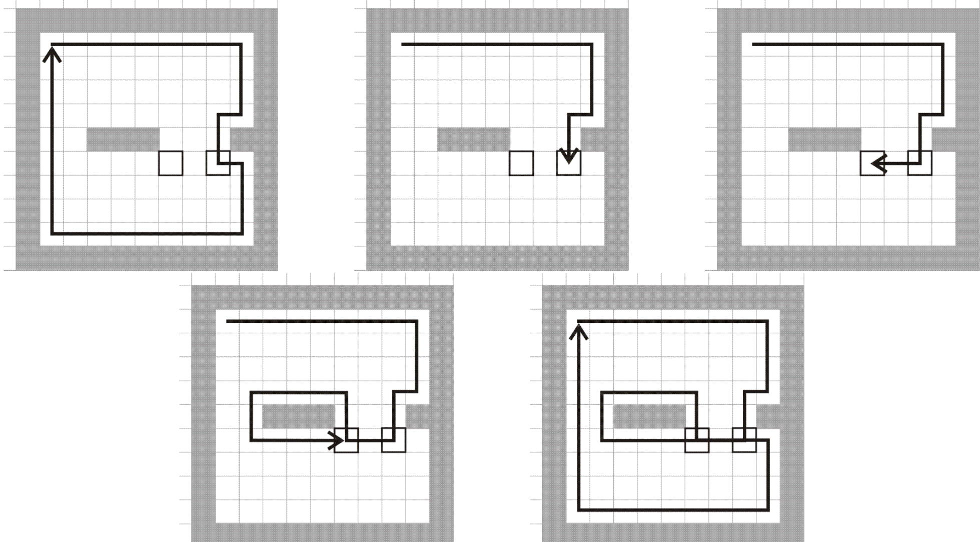


Bild 2.11: oben links: der ursprüngliche Kettencode, oben Mitte: der Kettencode bis zum Türkandidaten, oben rechts: der Kettencode inklusive geschlossener Tür, unten links: der Kettencode mit integriertem wandumschreibenden Kettencode, unten rechts: der neue geschlossene Kettencode

eingeschlossene freie Fläche jedoch einen negativen Wert, welcher der Fläche des Hindernisses entspricht[6].

Handelt es sich bei dem aktuell betrachteten Kettencode um einen wandumschreibenden, so wird dieser zur Liste der wandumschreibenden Kettencodes hinzugefügt, damit diese auf der Suche nach Gegenkandidaten[7] eingeschlossen werden können. Beschreibt die aktuelle Kontur eine freie Fläche ohne Durchgänge, so wird diese entsprechend der in Kapitel 2.1.2 angegebenen Definition als Raum erkannt. Zu diesem Raum werden schließlich eine Reihe von Informationen berechnet: die bereits erwähnte eingeschlossene Fläche, die „bounding box“, welche dem kleinsten Rechteck entspricht, welches den gefundenen Raum vollständig einschließt, sowie der Schwerpunkt [8] des Raumes. Der Schwerpunkt einer Fläche beschreibt hierbei einen Punkt, durch den eine beliebige Gerade die Eigenschaft

[6]siehe hierzu `http://pages.cpsc.ucalgary.ca/~parker/chain.c`

[7]vgl. Kapitel 2.4

[8]engl. „center of Gravity“

besitzt, dass sich auf beiden Seiten dieser Gerade jeweils die Hälfte des Flächeninhalts dieser Fläche befindet.

Kapitel 3

Implementierung

In diesem Kapitel soll die praktische Umsetzung des in Kapitel 2 vorgestellten Algorithmus durch eine Implementierung näher beleuchtet werden. Dazu wurde die „Programmier-Umgebung für die Muster-Analyse“ (PUMA)[1] verwendet, in welches die Implementierung integriert wurde. Als Programmiersprache wurde C++ gewählt[2], um eine Kompatibilität mit PUMA, vor allem in Bezug auf die Wartung, zu erreichen.

Eines der Hauptziele der Umsetzung war eine modulare Struktur, so dass einzelne Elemente des Algorithmus an anderer Stelle wiederverwendet werden können. Dies wurde über eine Klassenhierarchie realisiert, in welcher jeder abgeschlossene Teilalgorithmus in einer eigenen Funktionsklasse implementiert wurde (siehe Kapitel 3.1). Somit besteht das eigentliche ausführbare Programm (welche sich in PUMA im Modul „ANIMAL“ befinden) lediglich aus einer Reihe von Funktionsaufrufen aus diesen Klassen. Dieses Programm wurde sowohl als Kommandozeilenprogramm realisiert, als auch mit einer grafischen Oberfläche versehen. Bild 3.1 zeigt eine Bildschirmaufnahme dieser Oberfläche[3].

Das Programm liest Grundrisse aus pixelbasierten Farb- oder Grauwertbildern verschiedenster Formate (.png, .gif, .jpg) aus. Diese werden ggfs. über bereits in PUMA vorhan-

[1] `http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/puma`

[2] [Str00]

[3] Die grafische Oberfläche wurde von Oliver Lammersdorf für das Projektpraktikum „Robbie 7“ der Arbeitsgruppe Aktives Sehen (AGAS) entwickelt und für diese Arbeit angepasst.

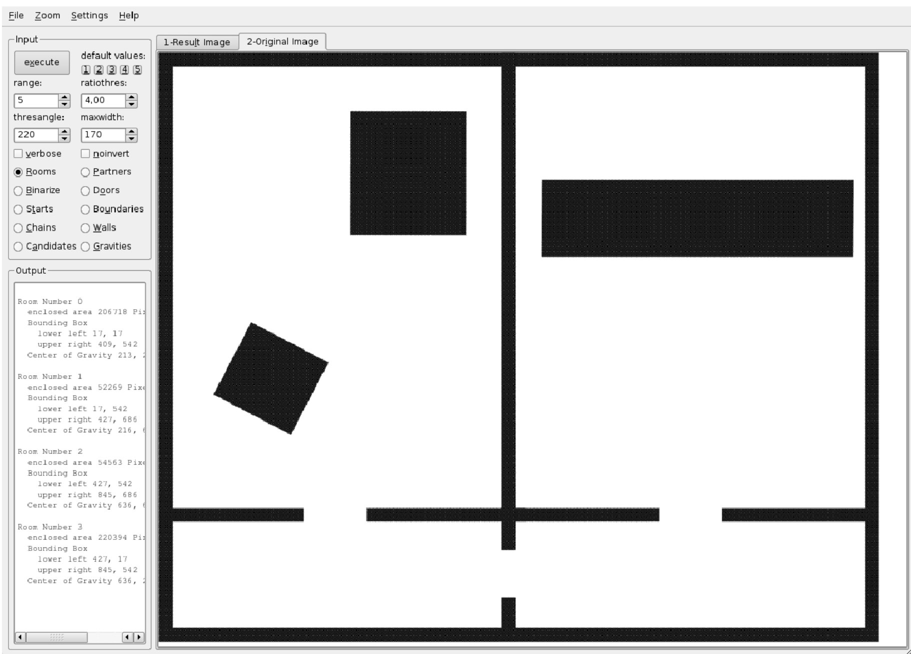


Bild 3.1: eine Bildschirmaufnahme der grafischen Oberfläche

dene Mechanismen in ein Graustufenbild umgewandelt. Zudem werden diese in einem weiteren Schritt binarisiert, da dies dem Umwandeln in ein binäres Occupancy Grid entspricht (vgl. hierzu Kapitel 2.2.1). Da die Erstellung der Kettencodes wie in Kapitel 2.2.2 beschrieben nicht auf, sondern neben den zu beschreibenden Hindernissen geschieht, wurde das Originalbild außerdem invertiert. Die Klasse zum Erstellen der Kettencodes[4] arbeitet somit auf den Hindernissen, um eine höhere Flexibilität der Klassenfunktionalität zu gewährleisten.

Die Ausgabe erfolgt ebenfalls in einem wählbaren Format. Hierbei können verschiedene Formen der Ausgabe gewählt werden: während bei einem Aufruf des Programms ohne Optionen der Grundriss in verschiedene Räume unterteilt wird, welche in verschiedenen Graustufen angezeigt werden (vgl. Kapitel 4), können über verschiedene Optionen auch Zwischenschritte des Vorgangs im Ausgabebild visualisiert werden. Eine Liste dieser Optionen findet sich in Tabelle 3.1. Zusätzlich können mit der Option `showinformation`

[4] siehe Kapitel 3.1.4

Option	Visualisierung
`showbinarize`	Zeigt das binarisierte, invertierte Originalbild an
`showstarts`	Zeigt die Startpunkte aller Kettencodes in einem angedeuteten Originalbild an
`showchains`	Zeigt alle raumumschreibenden Kettencodes in einem angedeuteten Originalbild an
`showwalls`	Zeigt alle wandumschreibenden Kettencodes in einem angedeuteten Originalbild an
`showcandidates`	Zeigt alle Türkandidaten in einem angedeuteten Originalbild an
`showpartners`	Zeigt alle Türkandidaten mit jeweiligem Gegenkandidaten in einem angedeuteten Originalbild an
`showdoors`	Zeigt alle Türkandidaten und Gegenkandidaten in einem angedeuteten Originalbild an, welche zum Schließen von Kettencodes verwendet wurden
`showboundaries`	Zeigt die raumumschreibenden Kettencodes aller gefundenen Räume in einem angedeuteten Originalbild an
`showgravities`	Zeigt zu showboundaries noch sämtliche Schwerpunkte aller Räume an

Tabelle 3.1: Optionen des ausführbaren Programms, um verschiedene Zwischenschritte zu visualisieren

die in Kapitel 2.5.2 beschriebenen Zusatzinformationen zu den gefundenen Räumen ausgegeben werden.

Aus PUMA wurden zwei bereits implementierte Klassen verwendet: die Klasse `chain` und die Klasse `binarize`. Die erstgenannte ist hierbei eine Klasse zur Verwaltung einzelner Kettencodes. Sie liefert Möglichkeiten zum Speichern von Punkten und Richtungen in Kettencodes sowie die Möglichkeit des Löschens einzelner Elemente oder dem Anfügen bereits existenter Kettencodes.

Die Klasse `binarize` hingegen ist eine Funktionsklasse, welche Funktionen zum binari-

sieren von Eingabebildern zur Verfügung stellt. Hierzu stehen verschiedene Algorithmen sowie veschiedene Schwellwerte zur Verfügung, um das Ergebnis anzupassen[5].

3.1 Klassenbeschreibung

Wie bereits zu Beginn dieses Kapitels geschrieben wurde die Klassenhierarchie modular strukturiert. Einige Diagramme zur Modellierung[6] der verwendeten Klassen findet sich in Anhang A. Im Folgenden sollen einzelne Klassen und deren spezifische Eigenschaften der Implementierung näher beschrieben werden.

3.1.1 FloodFill

Die Klasse `FloodFill` realisiert einen Algorithmus zum farblichen Füllen gleichartiger Flächen [, Floodfill]. Allerdings ist die einfach rekursive Variante dieses Algorithmus sehr speicher- und laufzeitintensiv, da für jedes bearbeitete Pixel eine weitere Rekursionsstufe verwendet wird. Um dies zu umgehen wurde daher eine modifizierte Form verwendet [, Linefilling], welche zuerst die x–Achse des aktuellen Blocks füllt und erst in einem weiteren Schritt die an diese Linie angrenzenden Blöcke ober- bzw. unterhalb untersucht. Somit wurde die Rekursionstiefe von n^2 auf $c\,n$ verringert, was wiederum in einer Verbesserung der Laufzeit resultiert[7].

3.1.2 ChainAngle

Wie in Kapitel 2.3 beschrieben wird zur Kandidatensuche der Winkel an der aktuellen Stelle im Kettencode berechnet. Hierzu wird die Formel

$$\cos\alpha = \frac{a_1 b_1 + a_2 b_2}{\sqrt{a_1^2 + a_2^2} * \sqrt{b_1^2 + b_2^2}}$$

[5] Näheres findet sich in der Dokumentation zu PUMA unter `http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/puma/Dokumentation`

[6] die verwendete Modellierungssprache ist die Unified Modelling Language (UML) [FS97]

[7] vgl. Kapitel 4

verwendet. Hierbei stehen a_i bzw. b_i für die Werte innerhalb des jeweiligen Vektors $\boldsymbol{a}$ bzw. $\boldsymbol{b}$. Da diese Formel jedoch durch Benutzung des cosinus nur einen Winkel zwischen 0 und 180 Grad ermitteln kann, wurden die Winkel zwischen den beiden Vektoren über einen Umweg berechnet: zunächst wird der Winkel zwischen dem betrachteten Vektor und einem Standardvektor $\boldsymbol{t_0}$ berechnet. Dieser normierte Standardvektor besitzt lediglich einen Wert für eine Verschiebung in x–Richtung. Hierdurch kann nun anhand der y–Richtung des betrachteten Vectors unterschieden werden, ob sich dieser rechts- oder linksseitig von $\boldsymbol{t_0}$ befindet. Diese Berechnung wird mit beiden Vectoren a_i und b_i durchgeführt. Die Differenz der beiden Winkel ergibt nun den gesuchten Winkel zwischen beiden Vektoren in einem Bereich von 0 bis 180 Grad.

Durch die Berechnung der Winkel an jedem Element des Kettencodes kann es vorkommen, dass an einer Ecke in der Kontur des Hindernisses mehrere Türkandidaten gefunden werden, da die o. a. Bedingung für eine Linkskurve für mehrere Punkte dieser Ecke zutrifft. Um eine solche Vervielfachung an Türkandidaten zu verhindern, wurde innerhalb eines solchen Blocks an Türkandidaten nur jener gespeichert, welcher durch die beiden Vektoren den höchsten Winkel aufspannte. Ein Block beginnt hierbei mit dem ersten Element im Kettencode, welches an einer Linkskurve liegt, und endet mit dem ersten folgenden Element, an dem keine klare Linkskurve mehr zu erkennen ist.

3.1.3 DoorFinder

In der Klasse `DoorFinder` wurde sowohl die Suche nach Türkandidaten und Gegenkandidaten[8] als auch die Auswertung der Räume[9] umgesetzt. Außerdem befinden sich in dieser Klasse die Funktionen zur Visualisierung der in Tabelle 3.1 benannten Zwischenschritte.

In Kapitel 2.5.1 wurde das Schließen eines Kettencodes an einem Durchgang beschrieben. Hierzu wurde der aktuell betrachtete Kettencode an der Stelle des Türkandidaten unterbrochen und durch eine Linie zwischen Türkandidat und Gegenkandidat fortgesetzt. Damit wurde jedoch angenommen, dass der Türkandidat im Kettencode vor dem Gegenkandi-

[8] vgl. Kapitel 2.3 und 2.4
[9] siehe Kapitel 2.5

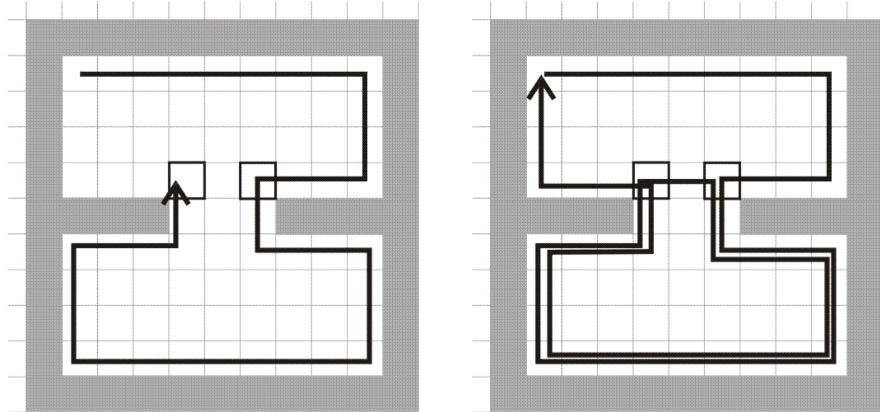


Bild 3.2: links: der Kettencode bis zum Türkandidaten, rechts: einzelne Bereiche der Kontur werden doppelt erfasst

daten gespeichert ist. Würde in einem umgekehrten Fall der beschriebene Algorithmus ohne Modifikation angewandt, so würde dies zu einer Mehrfacherfassung einer Teilkontur führen (siehe Bild 3.2). Um dies zu verhindern, wird in der Implementation die Position von Tür- und Gegenkandidat im Kettencode verglichen. Befindet sich der Gegenkandidat vor seinem zugehörigen Türkandidaten im Kettencode, so werden die beiden Variablen vertauscht, so dass die ursprünglich gewünschte Reihenfolge wieder hergestellt ist und die genannten Nebeneffekte vernachlässigt werden können.

Als Räume werden nach Kapitel 2.5.2 alle freien Flächen erfasst, welche keine Durchgänge mehr besitzen und deren berechneter Flächeninhalt positiv ist. Dies hat jedoch zur Folge, dass kleinere Bereiche wie Pixelfehler oder Fehler während der Kartenerstellung ebenfalls als Räume erkannt werden. Daher wurde als Schwellwert für Räume eine Mindestfläche von 20 Pixeln vorrausgesetzt. Die Bereiche mit einer Fläche A, welche die Eigenschaft $0 < A < 20$ Pixeln haben, werden in den verschiedenen Ausgabedateien[10] sowie Tabelle 3.1 als Hindernis dargestellt.

[10] vgl. Kapitel 3.1

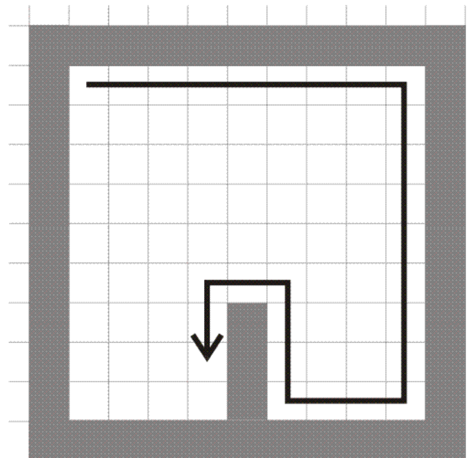


Bild 3.3: Hindernisse befinden sich immer linksseitig des Kettencodes

3.1.4 GenChainCode

Wie bereits in Kapitel 2.2.2 angedeutet, wurden alle Kettencodes in dieser Arbeit so erstellt, dass die Hindernisse, deren Kontur der jeweilige Kettencode beschreibt, sich auf der linken Seite des Kettencodes befinden (siehe Bild 3.3). Dazu sind zwei Schritte notwendig: zunächst wird ab der linken oberen Ecke des Occupancy Grids der erste Punkt gesucht, an dem sich kein Hindernis befindet. Ab diesem wird in positiver x–Richtung (vgl. Bild 3.3) entlang des Hindernisses der Kettencode erstellt, bis dieser „geschlossen“ ist, also Start- und Endpunkt des Kettencodes übereinstimmen. Sowohl das soeben erfasste Hindernis als auch die freie Fläche, die dieses Hindernis umschließt, werden nun mit einem Grauwert „geflutet“[11]. Der Grauwert muss sich hierbei sowohl von dem der Hindernisse als auch von dem der hindernisfreien Flächen unterscheiden. Dieser Vorgang der Kettencodeerstellung wird mit der Suche nach dem nächsten freien Punkt in der Karte fortgesetzt, bis keine freien Bereiche mehr in der Karte existieren.

Nachdem die Karte vollständig nach dem o. g. Schema durchsucht worden ist, ist es jedoch möglich, dass einzelne Hindernisse, welche keine freien Fläche umschließen, nicht erfasst wurden (siehe ABBILDUNG C). Die Erstellung der Kettencodes für diese Hindernisse

[11] vgl. Kapitel 3.1.1

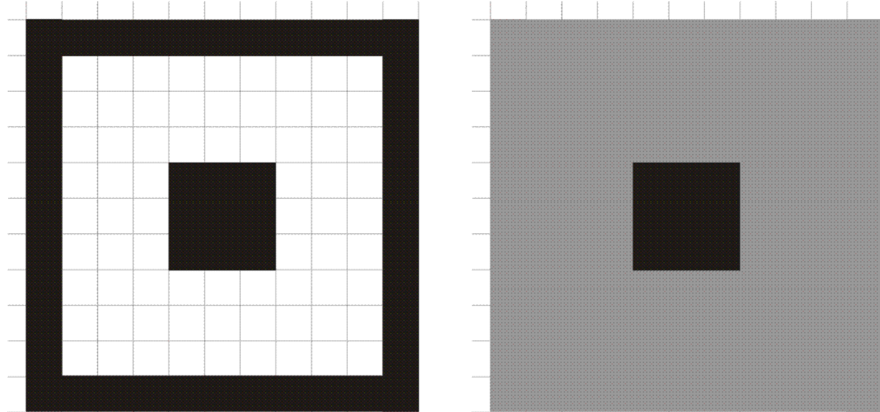


Bild 3.4: links: das Originalbild, rechts: die Karte nach dem ersten Schritt der Kettencodeerstellung

erfolgt in einem zweiten Schritt: das Occupancy Grid, in welchem keine freien Flächen mehr existieren, wird ab der oberen linken Ecke der erste Punkt eines Hindernisses gesucht. Da im ersten Schritt bereits wie oben beschrieben alle Hindernisse, welche bereits erfasst wurden, mit einem abweichenden Grauwert „geflutet“ wurden, befindet sich der gefundene Punkt eines Hindernisses auf einem Objekt, welches bisher nicht erfasst wurde. Für dieses Hindernis wird nun ebenfalls ein Kettencode erstellt, welcher sich neben dem Hindernis befindet. Die Startrichtung ist hierbei in positiver y–Richtung (siehe ABBILDUNG D), damit sich das Hindernis wie in Kapitel 2.2.2 gefordert links neben dem Kettencode befindet.

Durch diese beiden Schritte werden alle Konturen von Hindernissen in der Karte von allen Seiten erfasst. Um zudem zu gewährleisten, dass zusammenhängende Objekte auch in einem Kettencode als solche repräsentiert werden, wird als Abbruchbedingung während der Erstellung sowohl der aktuelle Punkt als auch die aktuelle Richtung mit den Werten des Startpunktes des Kettencodes verglichen. Da nur bei Übereinstimmung beider Werte mit den Startwerten die Kontur des Objektes als Kettencode gespeichert wird, werden evtl. vorkommende ein Pixel starke Nischen in Objekte zwar erfasst, führen jedoch nicht zu mehreren Teilkonturen. Zusätzlich wurde durch diese Methode garantiert, dass alle Kettencodes „geschlossen“ sind, dass also Start- und Endpunkt eines Kettencodes identisch

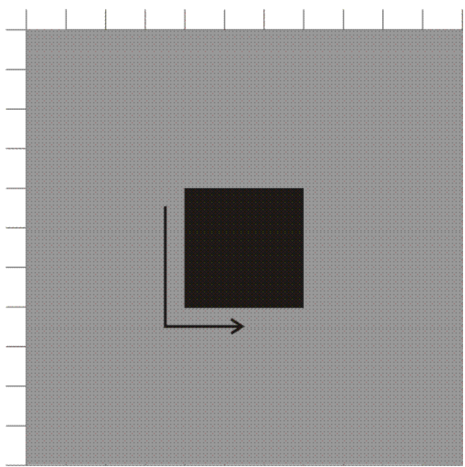


Bild 3.5: Startrichtung der Kettencodes in Schritt 2 der Erstellung

sind.

3.1.5 RoomDescriptor

Die Klasse `RoomDescriptor` stellt eine Methode zur Verfügung, die zu einem gefundenen Raum die in Kapitel 2.5.2 beschriebenen Zusatzinformationen berechnet. Zur Berechnung der Fläche, die von einem Kettencode umschlossen wird, wird hierbei ein Algorithmus verwendet[12], welcher die Eigenschaft der Kettencodes, sich immer rechts neben den umschriebenen Hindernissen zu befinden, verwendet. Hierdurch werden alle freien Flächen im Uhrzeigersinn umschlossen. Somit haben alle Kettencodeelemente, die in negativer x–Richtung verlaufen, an diesem Punkt einen höheren y–Wert als Kettencodeelemente, die in positiver x–Richtung verlaufen. Werden nun die y–Werte aller Kettencodeelemente mit negativer x–Richtung addiert und die y–Werte aller Kettencodeelemente mit positiver x–Richtung subtrahiert, so erhält man den Flächeninhalt der von diesem Kettencode eingeschlossenen Fläche.

Da wandumschreibende Kettencodes keine freie Fläche umschreiben, sondern lediglich einzelne Hindernisse, werden diese gegen den Uhrzeigersinn umschlossen. Wendet man

[12] vgl. `http://pages.cpsc.ucalgary.ca/~parker/chain.c`

hier die o. g. Flächenberechnung an, so ergibt sich ein negativer Flächeninhalt, da die y–Werte der Kettencodeelemente in negativer x–Richtung geringer sind als die y–Werte der Kettencodeelemente in positiver x–Richtung. Somit kann die Berechnung zur Unterscheidung von raum- bzw. wandumschreibenden Kettencodes verwendet werden.

Zur Berechnung der „Bounding Box“[13] werden die jeweils höchsten und niedrigsten x– und y–Werte eines Kettencodes ermittelt. Diese werden dann in einer üblichen Form als unterer linker sowie als oberer rechter Punkt der Bounding Box gespeichert.

Der Schwerpunkt [14] einer Fläche, welche durch einen Kettencode umschlossen wird, wird nach einem Algorithmus berechnet, welcher dem der Flächenberechnung ähnelt. Hier werden jedoch nicht die reinen y–Werte der einzelnen Kettencodeelemente addiert bzw. subtrahiert, sondern die Gauss'sche Reihe

$$\frac{n^2 + n}{2}$$

über dem jeweiligen y–Wert. Teilt man diese Summe schließlich durch die Anzahl der umschlossenen Punkte, also durch den Flächeninhalt des Raumes, so erhält man den durchschnittlichen y–Wert der Fläche. Führt man diese Berechnung ebenfalls für alle x–Werte durch, so erhält man den Punkt mit dem durchschnittlichen x–Wert und dem durchschnittlichen y-Wert der Fläche. Dieser Punkt entspricht dem Schwerpunkt der Fläche.

3.2 Schwellwerte

Im bisherigen Verlauf der Arbeit wurden verschiedene Schwellwerte vorgestellt, die während der Durchgangssuche verwendet wurden. Vor allem bei der Suche nach Gegenkandidaten[15] wurden Schwellwerte eingesetzt, um die in Kapitel 2.1 geforderten Bedingungen zu erfüllen. Um ein hohes Maß an Flexibilität zu gewährleisten, können einige dieser Schwellwerte beim Programmaufruf über Optionen verändert werde. So kann z. B. über den Wert der maximalen Türbreite verhindert werden, dass Durchgänge in langen Korri-

[13] vgl. Kapitel 2.5.2
[14] engl. „center of gravity“
[15] vgl. Kapitel 2.4

doren erkannt werden (siehe Kapitel 4). Diese anpassbaren Schwellwerte sollen im Folgenden näher betrachtet werden.

In Kapitel 2.1.3 wurde als Bedingung für einen Durchgang genannt, dass die Entfernung von Türkandidat zu seinem Gegenkandidat „signifikant geringer“ sein muss. Diese Abhängigkeit wurde durch die Variable `ratiothres` implementiert. Um einen Gegenkandidaten als Partner zu einem Türkandidaten zu speichern, muss unter anderem gelten:

$$\frac{U}{d} > r_0$$

r_0 beschreibt hierbei den zu überschreitenden Schwellwert, d die Entfernung der Punkte von Tür- und Gegenkandidat und U den Umfang des kleineren der beiden Räume, welcher durch ein Verbinden der beiden Punkte entstehen würde. Für eine detailierte Beschreibung der Berechnung des Umfangs siehe Kapitel 2.4. Der voreingestellte Wert der Variablen liegt bei $6,0$.

Bei der Suche nach Türkandidaten im Kettencode[16] wurde der Schwellwert α_0 eingeführt, um Linkskurven als solche zu erkennen. Dieser entspricht der implementierten Variablen `thresangle`. Um bei der Erkennung von Linkskurven evtl. vorkommende Pixelfehler oder geringe zu vernachlässigende Vorsprünge nicht zu berücksichtigen, wurde für `thresangle` ein Standardwert gewählt, welcher mit 220 Grad deutlich über den vorher genannten 180 Grad liegt.

Zur Berechnung der Winkel werden, wie in Kapitel 2.3 beschrieben, zwei Vektoren aufgespannt, jeweils von dem aktuellen Punkt der Verarbeitung zu dem Kettencodeelement, welches n Elemente vor bzw. nach dem aktuellen gespeichert ist. Der theoretische Wert n wurde mit der Variablen `range` umgesetzt. Wird dieser Schwellwert nicht über eine Option beim Programmaufruf überschrieben, so entspricht er dem Wert 5.

Eine der Vorraussetzungen während der Gegenkandidatensuche in Kapitel 2.4 war die maximale Breite einer Tür. Die Variable `maxdoorwidth` entspricht dieser maximalen Entfernung d_0 von Türkandidat und Gegenkandidat. Der Standardwert für die Entfernung beträgt 40 Pixel.

[16]vgl. Kapitel 2.3

Kapitel 4

Experimente und Ergebnisse

Im folgenden Abschnitt werden Versuche und Beispiele vorgestellt, welche mit dem in Kapitel 2 vorgestellten Algorithmus nach Räumen durchsucht wurden.

4.1 Versuchsaufbau

Die Versuche wurden unter dem UNIX–Betriebssystem Linux mit der Kernelversion 2.6.15 durchgeführt. Die Linuxdistribution war hierbei „ubuntu Dapper Drake“[1] Version 6.06. Zudem wurde die „Programmier–Umgebung für die Muster–Analyse“ (PUMA)[2] installiert, da die vorgestellte Implementation in diese Umgebung integriert wurde. Die verwendete Hardware war eine CPU[3] der Marke AMD[4] Athlon 3500+ bei einer Taktrate von 2200 MHz sowie 1 GB Arbeitsspeicher. Sämtliche Versuche wurden sowohl als Kommandozeilenaufruf als auch mit der hierfür zur Verfügung stehenden grafischen Oberfläche getestet. Als Grundrisse dienten hierbei sowohl eigens künstlich erzeugte Karten als auch solche, die durch den Roboter „Robbie“ der Arbeitsgruppe Aktives Sehen (AGAS)[5] durch die

[1] http://www.ubuntu.com
[2] http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/puma
[3] Central Processing Unit
[4] Advanced Micro Devices, http://www.amd.com
[5] http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus

Raum Nr.	Flächeninhalt	untere linke Ecke	rechte obere Ecke	Schwerpunkt
0	206718 Pixel	(17, 542)	(409, 17)	(213, 279)
1	52269 Pixel	(17, 686)	(427, 542)	(216, 621)
2	54563 Pixel	(427, 686)	(845, 542)	(636, 621)
3	220394 Pixel	(427, 542)	(845, 17)	(636, 279)

Tabelle 4.1: Zusatzinformationen zu den in der Karte a.gif gefundenen Räumen, die Punkte der unteren linken sowie der oberen rechten Ecke benennen die x– bzw. y–Koordinaten der Ecken der Bounding Box

Zusammenführung von Laserscans erstellt wurden.

4.2 Ergebnisse

4.2.1 Zwischenschritte

In Kapitel 3 wurden u. a. in Tabelle 3.1 Optionen erwähnt, über welche verschiedene Zwischenschritte der Verarbeitung in dem erstellten Ausgabebild angezeigt werden können. In Bild 4.1 finden sich neben dem Originalbild und der Visualisierung der gefundenen Räume auch eine Darstellung aller Türkandidaten, die während der Kandidatensuche gefunden werden. Außerdem zeigt das Bild alle raumumschreibenden Kettencodes der gefundenen Räume sowie deren Schwerpunkte in einem der Teilbilder an. Die zu diesen vier gefundenen Räumen berechneten Zusatzinformationen, welche in Kapitel 2.5.2 vorgestellt wurden, werden in Tabelle 4.1 je Raum aufgelistet.

4.2.2 Verschiedene Grundrisse

Nachdem in Kapitel 4.2.1 einzelne Zwischenschritte eines Grundrisses untersucht wurden, werden hier die Ergebnisse der Raumsuche verschiedener Karten vorgestellt. Die in Tabelle 4.2 gezeigten Karten wurden künstlich erstellt und dienen der Verdeutlichung un-

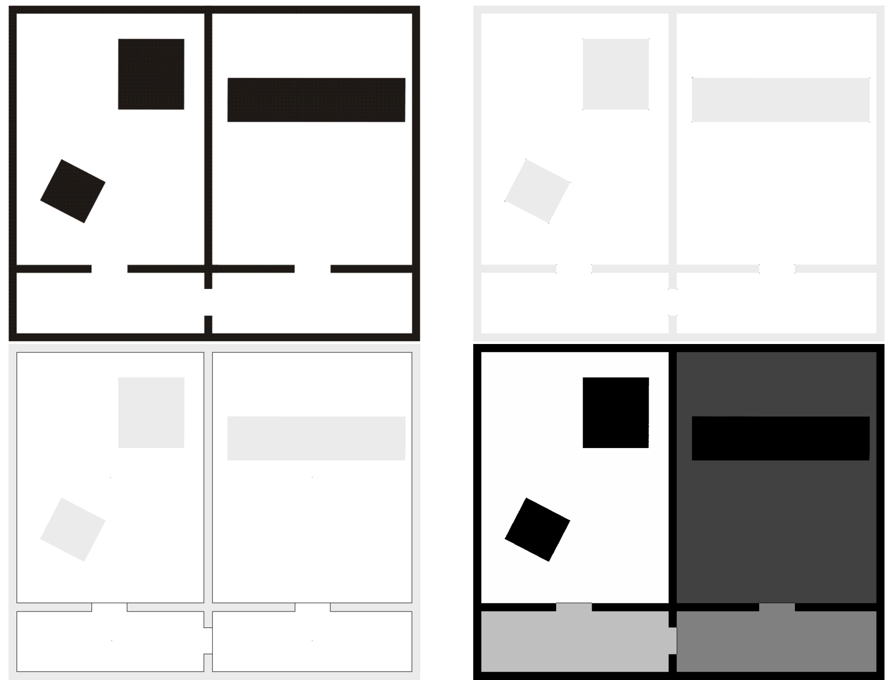


Bild 4.1: oben links: das Originalbild, oben rechts: sämtliche gefundenen Türkandidaten, unten links: die raumumschreibenden Kettencodes der erkannten Räume, unten rechts: die Visualisierung der Räume

terschiedlicher Kartentypen. Karte `a.gif` zeigt einen Gebäudeteil mit drei Durchgängen und einzelnen in dem Gebäudeteil vorhandenen Hindernissen. In Beispiel `b.gif` wird ein Raum mit eingeschlossenem wandumschreibenden Kettencode gezeigt, welcher zum Schließen von Durchgängen verwendet wurde. Diese beiden Karten wurden aus [PGKKP] übernommen, da sie dort bereits zur Veranschaulichung der gefundenen Räume verwendet wurden. Der dritte Grundriss in Tabelle 4.2 schließlich zeigt eine Reihe von Räumen mit unterschiedlicher Form und Größe.

Die in Tabelle 4.2 angezeigten Werte r_0 und d_0 entsprechen hierbei dem Schwellwert für das Verhältnis von Umfang zu direkter Entfernung (vgl. Kapitel 2.4) bzw. für die maximale Türentfernung `maxdoorwidth` (vgl. Kapitel 3.2).

In Bild 4.2 wird ebenfalls eine künstlich erstellte Karte gezeigt. Diese ist jedoch eine

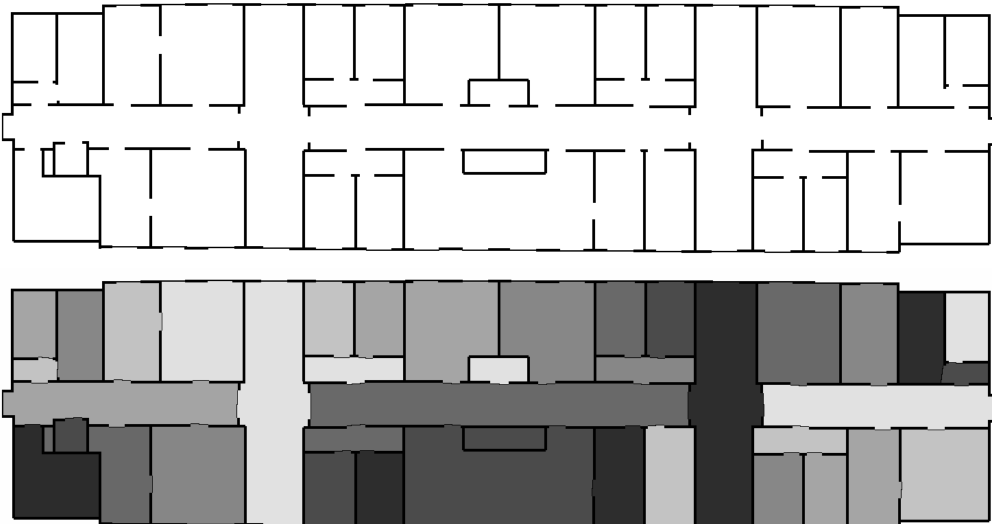


Bild 4.2: Eine künstlich erstellte Kopie des Grundriss des dritten Stockwerkes des B–Gebäudes der Universität Koblenz, oben: das Originalbild, unten: die Ausgabe des Programms nach der Raumerkennung

Nachbildung des Grundrisses des dritten Stockwerkes des B–Gebäudes der Universität Koblenz. Der hier dargestellte Grundriss unterscheidet sich von den bisher beschriebenen vor allem in der Anzahl der gefundenen Räume (siehe Tabelle 4.5) sowie durch das Vorkommen langer Flure[6].

Neben den bisherigen Karten, welche alle künstlich gezeichnet wurden, wurde der implementierte Algorithmus auch zur Suche nach Räumen in Karten verwendet, welche durch den Roboter der Arbeitsgruppe Aktives Sehen (AGAS[7]) „Robbie“[8] erstellt wurden. Beispiele hierfür finden sich in den Bildern 4.3 und 4.4. Allerdings führen einige der fehlerhaften Fragmente in der Karte (z. B. in Bild 4.4), welche während der Kartenerstellung entstanden sind, zu einem Schließen von Durchgängen zu diesen Fragmenten. Außerdem

[6] siehe hierzu Kapitel 4.2.3

[7] `http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus`

[8] `http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/Researches/ActiveVision/Robbie-6`

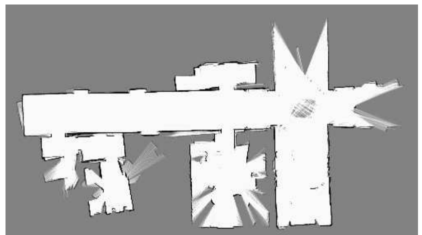


Bild 4.3: links: das von Robbie erstellte Originalbild, rechts: die Visualisierung der gefundenen Räume

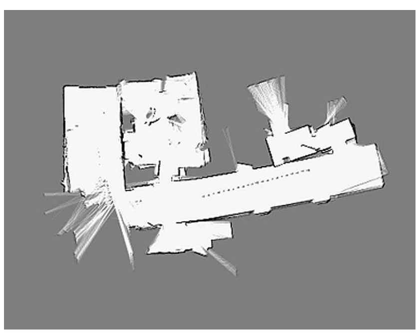


Bild 4.4: links: eine von Robbie erstellte Karte, rechts: die in dieser Karte gefundenen Räume

werden nicht vollständig erfasste Bereiche der Karte, welche nur kurz durch den Laserscanner abgetastet wurden und somit einzelnen strahlenförmigen Korridoren entsprechen, als Räume erkannt und z. T. auch in mehrere Räume unterteilt (vgl. Bild 4.3). Dieses Verhalten kann durch ein Anpassen der in Kapitel 3.2 genannten Schwellwerte nicht behoben werden, da diese Bereiche der Definition eines Raumes[9] entsprechen.

4.2.3 Schwellwerte

Die Anpassung der in Kapitel 3.2 beschriebenen Schwellwerte kann zu unterschiedlichen Ergebnissen der Raumerkennung führen. Einige Beispiele hierfür finden sich in Tabelle

[9] vgl. Kapitel 2.1.2

4.3. Während ein höherer Wert für das Verhältnis r_0[10] zu einer geringeren Anzahl an gefundenen Durchgängen führt, erhöht sich diese bei einer Verringerung des Wertes. Da die maximale Türbreite d_0 ein restriktiver Wert ist, entspricht ein hoher Wert einer geringeren Einschränkung und damit dem Speichern einer hohen Anzahl an Durchgängen, während ein geringer Wert für die maximale Türbreite in einer Verringerung der Anzahl der Durchgänge resultiert.

Die Anzahl der Räume, welche in den Beispielen in Tabelle 4.3 gefunden wurden, findet sich in Tabelle 4.6.

4.2.4 Sonderfälle

Wie bereits in Kapitel 2.4 beschrieben, können für verschiedene Türkandidaten die entsprechenden Gegenkandidaten nur in dem gleichen Kettencode gesucht werden. Die hieraus resultierende Einschränkung bei der Suche nach Durchgängen zeigt sich in dem Beispiel der Karte des Erdgeschosses des B–Gebäudes der Universität Koblenz `erdgeschoss.gif`: diese besteht im Gegensatz zur Karte `AGAS.gif` des dritten Stockwerkes des B–Gebäudes größtenteils aus wandumschreibenden[11], nicht aus raumumschreibenden[12] Kettencodes (siehe Bild 4.5). Dieser Unterschied entsteht durch die Durchgänge an der linken bzw. rechten Seite der Karten. Zwischen den verschiedenen wandumschreibenden Kettencodes werden keine Durchgänge geschlossen, was in einzelnen freistehenden Hindernissen in der Karte resultiert (siehe Bild 4.6). Durch ein Anpassen der Schwellwerte ändert sich das Ergebnis analog zu Kapitel 4.2.3, ein Schließen von Durchgängen zwischen den wandumschreibenden Kettencodes kann jedoch dadurch nicht erzielt werden (siehe Tabelle 4.4).

4.2.5 Laufzeit

Im Folgenden werden die Laufzeiten des Programms für die bisher gezeigten Beispiele vorgestellt. In Tabelle 4.5 sind die Arbeitszeiten des implementierten Algorithmus für alle

[10] vgl. Kapitel 2.4
[11] siehe hierzu Kapitel 2.2.2
[12] siehe hierzu Kapitel 2.2.2

in diesem Kapitel dargestellten Karten aufgelistet. Tabelle 4.6 zeigt die Laufzeiten für die Karte `AGAS.gif` mit den verschiedenen Schwellwerten, welche in Tabelle 4.2.3 gezeigt wurden.

Weder die Größe der Karte noch die eingestellten Schwellwerte haben einen signifikanten Einfluss auf die Laufzeit. Lediglich eine Kombination aus einer hohen Anzahl an Kettencodes und einer großen Menge an gefundenen Räumen führt zu einer erkennbar längeren Rechenzeit[13]. Diese Auswertung ist jedoch nur bedingt aussagekräftig, da mehrere Ausführungen mit derselben Karte unterschiedliche Laufzeiten mit einer Differenz bis zu einer Sekunde ergaben.

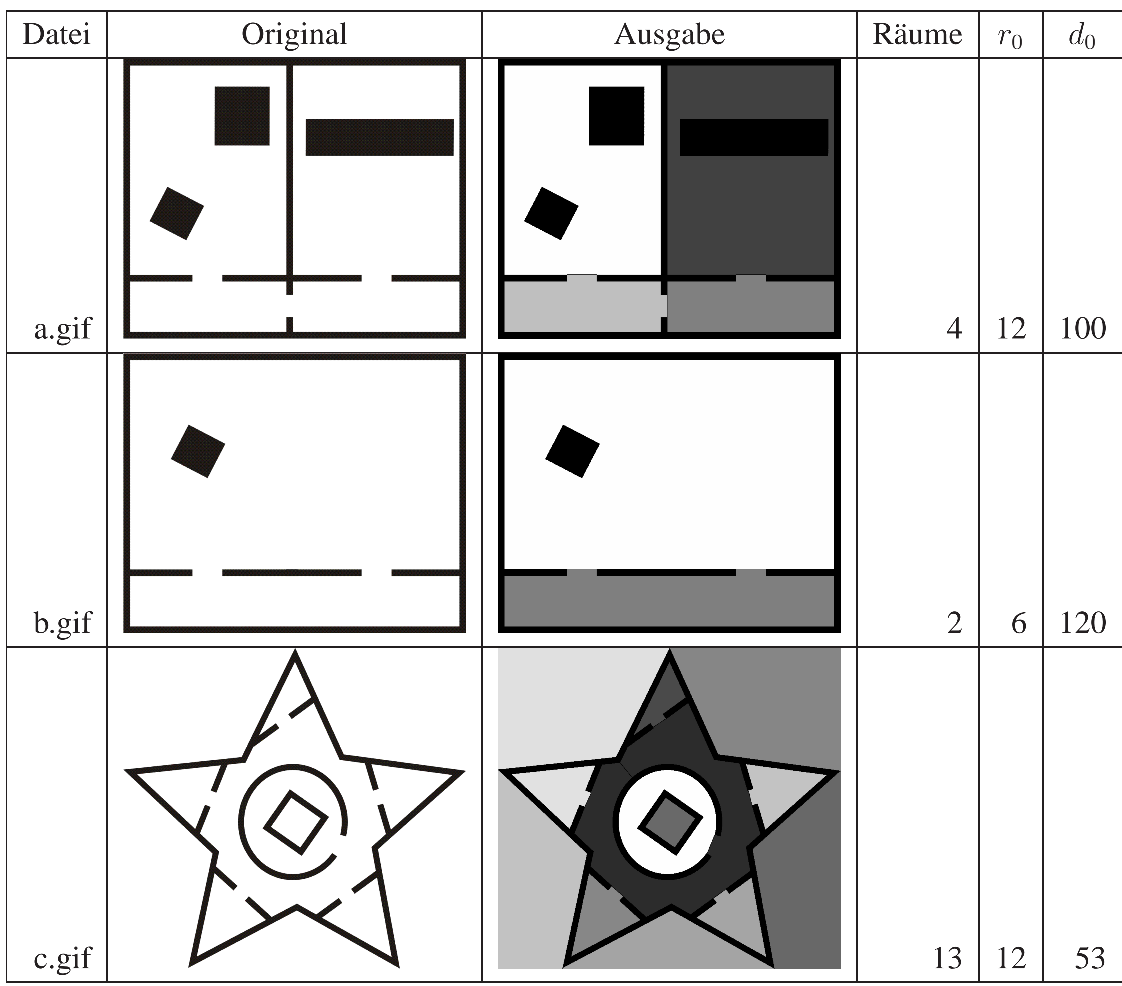
4.3 Bewertung

Die in den vergangenen Kapitel erzielten Ergebnisse entsprechen den intiutiv zu erwartenden Räumen der jeweiligen Grundrisse. Die in Kapitel 4.2.5 vorgestellten Laufzeiten sind relativ konstant und für das geforderte Ergebnis akzeptabel.

Allerdings ist für das Erreichen der gewünschten Ergebnisse je Grundriss eine Anpassung der Schwellwerte notwendig. Schlecht gewählte Schwellwerte führen zu unerwünschten Nebeneffekten (vgl. Tabelle 4.3). Zudem können einzelne Fehler im Grundriss, wie sie in automatisch erstellten Karten vorkommen, das Ergebnis verfälschen, wie an den Karten `robbie1.gif` und `robbie2.gif` gezeigt wurde.

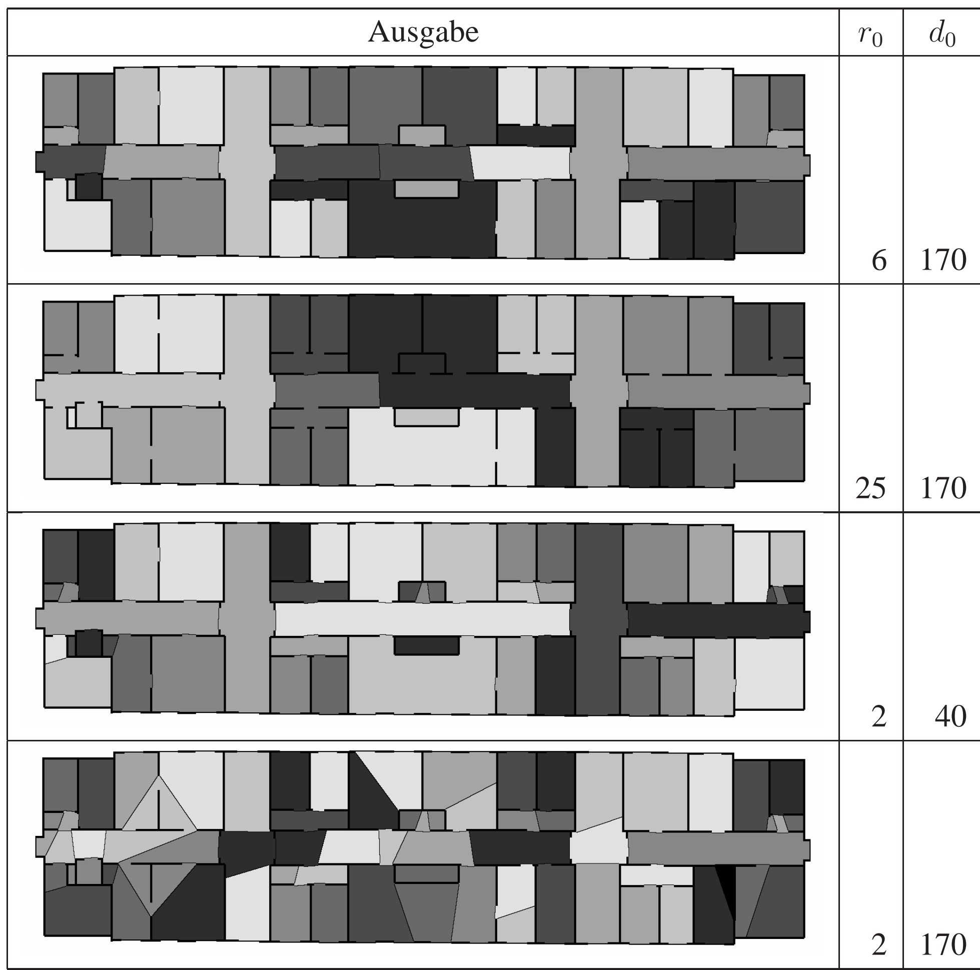
Zusätzlich können bestimmte Durchgänge, wie den Kapiteln 2.4 und 4.2.4 beschrieben, nicht geschlossen werden. Da es sich hierbei um einen durch den gewählten Algorithmus nicht zu lösenden Spezialfall handelt, kann dies auch nicht durch ein Anpassen der Schwellwerte umgangen werden, wie in Tabelle 4.4 deutlich wird. Somit eignet sich der hier vorgestellte Ansatz nicht für alle Grundrisse zur Raumsuche. Während für Karten mit zusammenhängenden Hindernissen akzeptable Ergebnisse erzielt werden können, führen Karten, welche analog zu dem in Bild 4.6 gezeigten Beispiel vorwiegend aus fragmentierten Hindernissen bestehen, zu unerwünschten Ergebnissen.

[13] vgl. Tabelle 4.5 Bild 5



Datei	Original	Ausgabe	Räume	r_0	d_0
a.gif			4	12	100
b.gif			2	6	120
c.gif			13	12	53

Tabelle 4.2: verschiedene getestete Grundrisse



Ausgabe	r_0	d_0
	6	170
	25	170
	2	40
	2	170

Tabelle 4.3: Auswirkung unterschiedlicher Schwellwerte auf das Ergebnis

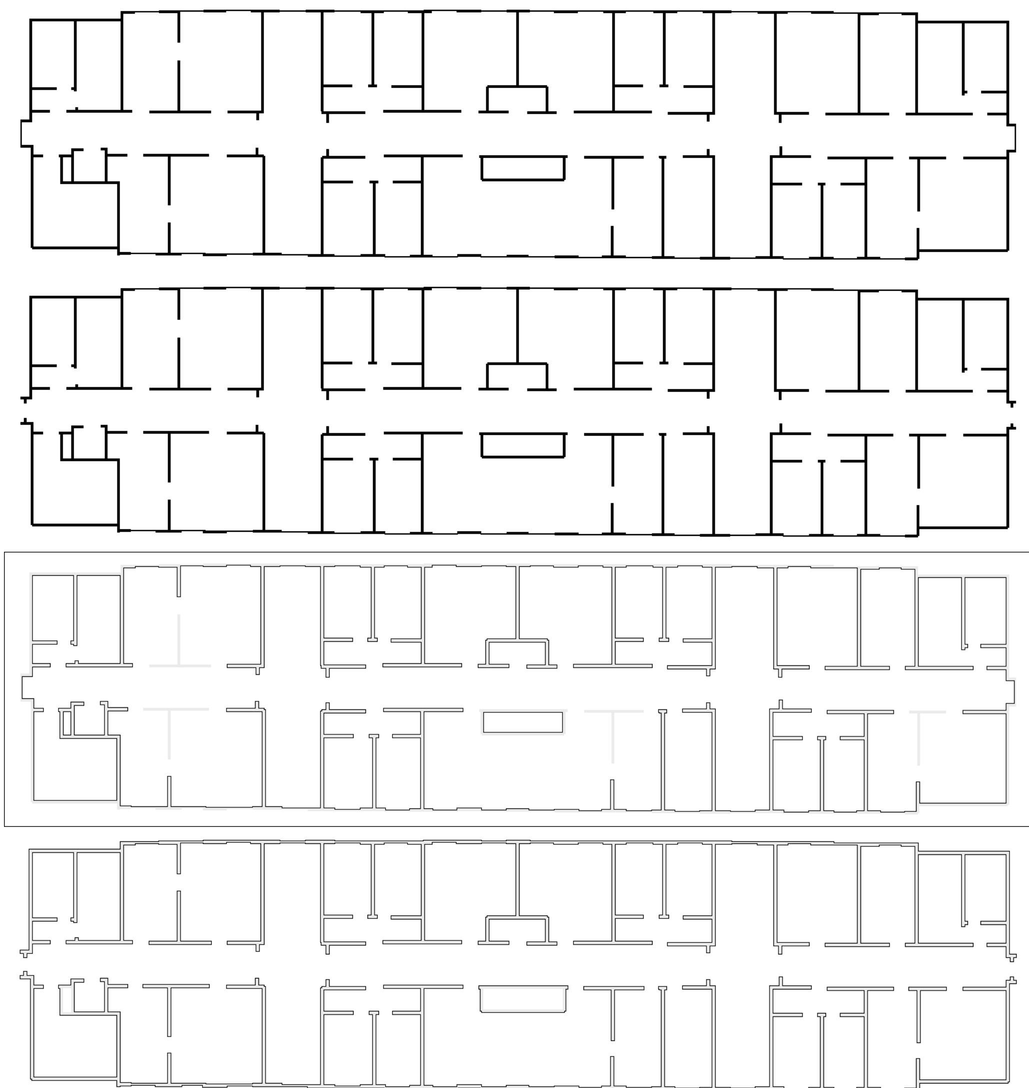


Bild 4.5: Künstlich erstellte Grundrisse des dritten Stockwerkes (Bild 1) bzw. des Erdgeschosses (Bild 2) des B–Gebäudes der Universität Koblenz, oben: die Originalbilder, unten: die aus den Karten erstellten Kettencodes

Bild 4.6: Ein künstlich erstellter Grundriss des Erdgeschosses des B–Gebäudes der Universität Koblenz, oben: das Originalbild, unten: die in der Karte gefundenen Räume. Zu einzelnen freistehenden Hindernissen werden keine Durchgänge geschlossen.

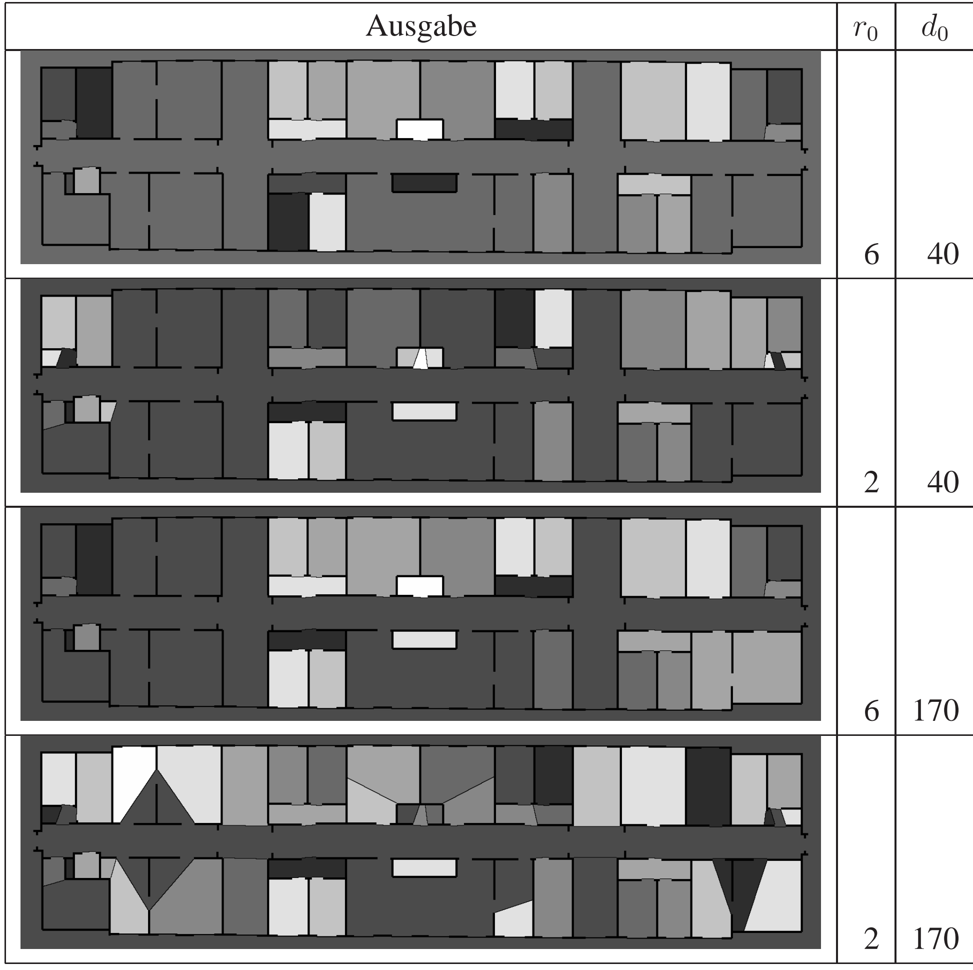
Ausgabe	r_0	d_0
	6	40
	2	40
	6	170
	2	170

Tabelle 4.4: Ein Schließen von Durchgängen zu den einzelnen freistehenden Hindernissen ist auch durch eine Anpassung der Schwellwerte nicht möglich

Datei	Größe des Bildes	Anzahl der Räume	Anzahl der Kettencodes	Rechenzeit
a.gif	863x704	4	4	2,437s
b.gif	863x704	2	3	3,183s
c.gif	947x885	13	5	3,237s
AGAS.gif	1500x400	41	4	3,466s
robbie1.gif	500x290	32	223	4,430s
robbie2.gif	800x620	70	302	14,285s

Tabelle 4.5: Laufzeiten der einzelnen Grundrisse

r_0	d_0	Räume	Laufzeit
6	40	41	3,466s
6	170	46	4,514s
25	170	17	4,854s
2	40	50	3,392s
2	170	69	6,587s

Tabelle 4.6: Laufzeiten bei verschiedenen Schwellwerten für die Karte AGAS.gif

Kapitel 5

Fazit

Die in Kapitel 1 beschriebenen Ziele dieser Arbeit, Gebäudeteile von anderen abzugrenzen, wurden erfolgreich umgesetzt. In den in Kapitel 4 vorgestellten Gebäudegrundrissen wurden Räume erkannt, die den intuitiv erwarteten entsprechen. Zudem spielte bei der Erkennung dieser Räume weder die Form oder die Größe eine Rolle, da für die Durchgangserkennung Verhältnisse gewählt wurden und keine Vergleichswerte zu Hilfe genommen werden mussten. Auch wurden Karten ausgewertet, welche von verschiedenen teils künstlichen und teils automatischen Quellen stammen. Die Laufzeit, welche hierfür benötigt wird, liegt in einem akzeptablen Rahmen. Zwar können die Karten nicht in Echtzeit ausgewertet werden, durch eine Rechenzeit im einstelligen Sekundenbereich für die vorgestellten Karten ist eine Einarbeitung in das Projekt „Robbie“ der Arbeitsgruppe Aktives Sehen (AGAS) jedoch möglich[1].

Allerdings herrscht für alle Ergebnisse eine Abhängigkeit von der Einstellung der Schwellwerte: werden diese schlecht gewählt, treten unerwünschte Effekte auf[2]. Die Anpassung dieser Schwellwerte erfolgt manuell, ein rein automatisches Auswerten der Karten findet bisher nur bedingt statt. Durch die bislang nur geringe Vorverarbeitung, welche mit dem Eingabebild geschieht, führen außerdem fehlerhafte Kartenfragmente und Kartographie-

[1] `http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/Researches/ActiveVision/Robbie-6`

[2] siehe Kapitel 4.3

fehler zu unerwarteten Durchgängen und Räumen[3].

Zudem spielt für das erzielte Ergebnis die Anordnung der Hindernisse eine Rolle (vgl. Kapitel 4.2.4 und 4.3). Somit eignet sich der vorgestellte Algorithmus nicht für alle Karten und Grundrisse zur Raumsuche.

Es existieren folglich mehrere Möglichkeiten der Erweiterung dieser Arbeit: durch eine Verbesserung der Eingabedaten (z. B durch Medianfilter[4] o. Ä.) könnten die erreichten Ergebnisse verbessert werden. Zudem wäre eine automatische Berechnung der Schwellwerte wünschenswert, da das Programm im Projekt „Robbie“ in einem autonomen Roboter zum Einsatz kommen soll. Um für die beschriebenen Spezialfälle ebenfalls die gewünschten Ergebnisse zu erzielen, wäre eine Kombination mit anderen Ansätzen und Methoden möglich[5].

Die hier vorgestellte Form der semantischen Auswertung von Karten stellt mit dem Unterteilen von Gebäudeteilen eine relativ geringe Stufe der Abstraktion dar. Diese kann jedoch verwendet werden, um darauf aufbauend weitere semantische Ebenen[6] zu erfassen.

---

[3] vgl. Kapitel 4

[4] siehe hierzu [Ara96]

[5] Beispiele hierfür wären sowohl die in Kapitel 1 vorgestellten Arbeiten als auch weitere Algorithmen zur Unterteilung von Gebäudegrundrissen und Kartenteilen, welche zum Zeitpunkt der Fertigstellung dieser Arbeit Gegenstand aktueller Forschung der Arbeitsgruppe Aktives Sehen (AGAS) an der Universität Koblenz–Landau sind `http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus`

[6] ein Beispiel hierfür wäre die in Kapitel 1 vorgestellten Möglichkeit der Raumtyperkennung über die Form des Raumes oder die darin enthaltenen Gegenstände

Anhang A

UML–Diagramme

Die “Unified Modelling Language“ (UML) [FS97] stellt eine gute Möglichkeit zur Visualisierung von Klassen dar. Die Bilder A.1 und A.2 zeigen die Hierarchie der implementierten Klassen sowie die Einbindung existierender Klassen in einem UML Klassendiagramm. Da die „Programmierumgebung zur Musteranalyse“ (PUMA) [1] aus verschiedenen Modulen besteht[2], wird in Bild A.3 die Integration der Klassen in die verschiedenen Module aus PUMA verdeutlicht.

[1]http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/puma

[2]siehe hierzu http://www.uni-koblenz.de/FB4/Institutes/ICV/AGPaulus/puma/Dokumentation

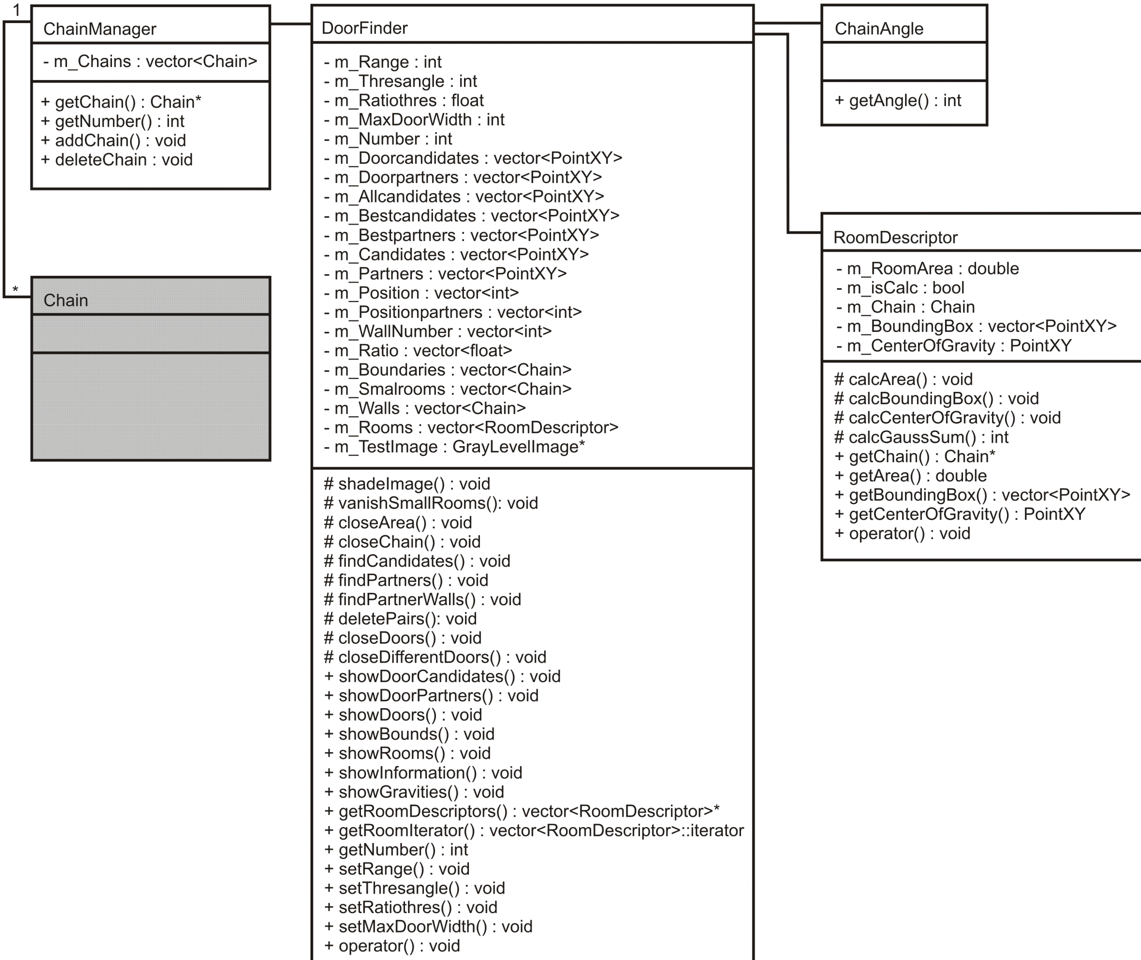


Bild A.1: Klassenhierarchie der implementierten Klassen. Grau hinterlegte Klassen entsprechen bereits existierenden Klassen aus PUMA

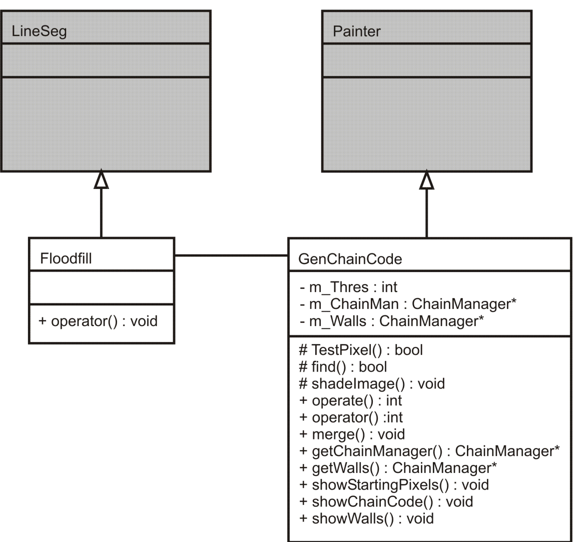


Bild A.2: Klassenhierarchie der implementierten Klassen. Grau hinterlegte Klassen entsprechen bereits existierenden Klassen aus PUMA

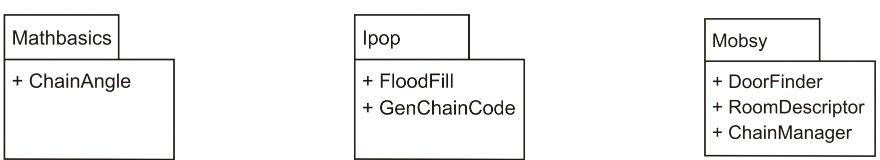


Bild A.3: Integration der Klassen in die verschiedenen Module in PUMA

Anhang B

Quellcode

Um den Rahmen der hier vorliegenden Arbeit nicht zu sprengen, wird im Folgenden lediglich der Quelltext einer Klasse exemplarisch dargestellt. Hierzu wurde die Klasse `FloodFill` gewählt. Durch die verwendete Programmiersprache C++ (vgl. Kapitel 3) besteht diese aus zwei Dateien: der Headerdatei `FloodFill.h` und der Sourcedatei `FloodFill.cpp`.

```
/******************************************************************************
* FloodFill.h
*
* (C) 2006 AG Aktives Sehen <agas@uni-koblenz.de>
* Universitaet Koblenz-Landau
*
* Information on Code Review state:
* Â§Author: KK; DevelTest: 2006/05/23; Reviewer: Initials; Review: Date; State: NOKÂ§
*
* Additional information:
* $Id: sa-0b.tex,v 1.1 2007/03/30 14:36:33 kaykowa Exp $
******************************************************************************/

#ifndef FloodFillOp_H
#define FloodFillOp_H

#include <stdio.h>
#include <vector>

#include "hippos/GrayLevelImage.h"
#include "hippos/PointXY.h"
```

```
#include "ipop/Painter.h"

#define THIS FloodFill
#define BASE Painter
#define BLACK 0
#define WHITE 255

/**
* @file ipop/FloodFill.h
*
* @class FloodFill
*
* @author Kay Kowalski
*
* @brief Fills an area inside an image with a desired Color.
*
* With the member functions of this class it is possible to fill an area
* inside an image with a desired Color. The area is determined by the starting
* pixel and a boundary Color. All Pixels which are either not in the boundary
* Color or are already in the desired fillcolor will be set to fillcolor.
* If no boundary Color is set, the algorithm uses the fillcolor as boundary color.
*/

class THIS : BASE
{
  public:

  /**
   * The standard constructor
   */
  THIS::THIS();

  /**
   * A constructor calling operator() (GrayLevelImage* image, byte fillColor, int x, int y)
   * after the creation of the object
   * @param image  a pointer to the image to fill a segment in
   * @param fillColor the color to fill the segment with
   * @param x the x value of the point inside the segment to be filled
   *  to start the floodfill from
   * @param y the y value of the point inside the segment to be filled
   * to start the floodfill from
   * @param boundColor the boundary color to border to segment to be filled with, if not set,
   * the default value is fillColor
   */
  THIS::THIS(GrayLevelImage* image, byte fillColor, int x, int y, int boundColor = -1);

  /**
   * A constructor calling operator() (GrayLevelImage* image, byte fillColor, PointXY* p)
```

```
 * after the creation of the object
 * @param image  a pointer to the image to fill a segment in
 * @param fillColor the color to fill the segment with
 * @param p a pointer to the point inside the segment to be filled to start the floodfill from
 * @param boundColor the boundary color to border to segment to be filled with, if not set,
 * the default value is fillColor
 */
THIS::THIS(GrayLevelImage* image, byte fillColor, const PointXY* p, int boundColor = -1);

/**
 * The destructor
 */
virtual THIS::~THIS();



/**
 * This method fills a segment of the image to which param image points to with fillColor.
 * The segment is specified by the point (x, y). The segment is bordered by pixels in
 * fillColor or in boundColor. If no boundColor is given, the segment is bordered
 * by fillColor.
 * @param image  a pointer to the image to fill a segment in
 * @param fillColor the color to fill the segment with
 * @param x the x value of the point inside the segment to be filled to start the floodfill from
 * @param y the y value of the point inside the segment to be filled to start the floodfill from
 * @param boundColor the boundary color to border to segment to be filled with, if not set,
 * the default value is fillColor
 */
void THIS::operator() (GrayLevelImage* image, byte fillColor, int x, int y, int boundColor=-1);

/**
 * This method fills a segment of the image to which param image points to with fillColor.
 * The segment is specified by the point param p points to. The segment is bordered by
 * pixels in fillColor or in boundColor. If no boundColor is given, the segment is bordered
 * by fillColor.
 * @param image  a pointer to the image to fill a segment in
 * @param fillColor the color to fill the segment with
 * @param p a pointer to the point inside the segment to be filled to start the floodfill from
 * @param boundColor the boundary color to border to segment to be filled with, if not set,
 * the default value is fillColor
 */
void THIS::operator() (GrayLevelImage* image,byte fillColor,const PointXY* p,int boundColor=-1);

/**
 * Sets the verbose status flag
 * @param verb the flag which sets the verbose status
 */
inline void THIS::setVerbose(bool verb);
```

```
  protected:

  private:

  /**
   * The verbose status flag
   * Standard value is FALSE
   * If set to TRUE, verbose messages will be printed to cout
   * to view the actual point of operation
   */
  bool THIS::m_Verbose;

};

// inline def
void THIS::setVerbose ( bool verb ) { THIS::m_Verbose = verb; }

#undef THIS
#undef BASE

#endif


/*******************************************************************************
* FloodFill.cpp
*
* (C) 2006 AG Aktives Sehen <agas@uni-koblenz.de>
* Universitaet Koblenz-Landau
*
* Information on Code Review state:
* Â§Author: KK; DevelTest: 2006/05/23; Reviewer: Initials; Review: Date; State: NOKÂ§
*
* Additional information:
* $Id: sa-0b.tex,v 1.1 2007/03/30 14:36:33 kaykowa Exp $
*******************************************************************************/

#include "ipop/FloodFill.h"

#define THIS FloodFill
#define BASE Painter


THIS::THIS() : BASE()
{
  THIS::m_Verbose = FALSE;
}
```

```
THIS::THIS(GrayLevelImage* image, byte fillColor, int x, int y, int boundColor) : BASE()
{
  THIS::m_Verbose = FALSE;
  THIS::operator() (image, fillColor, x, y, boundColor);
}


THIS::THIS(GrayLevelImage* image, byte fillColor, const PointXY* p, int boundColor) : BASE()
{
  THIS::m_Verbose = FALSE;
  THIS::operator() (image, fillColor, p, boundColor);
}


THIS::~THIS()
{
}




void THIS::operator() (GrayLevelImage* image, byte fillColor, const PointXY* p, int boundColor)
{
  THIS::operator() (image, fillColor, p->x(), p->y()); // call function with staring pixel p
}


void THIS::operator() (GrayLevelImage* image, byte fillColor, int x, int y, int boundColor)
{
  int count = 0;
  // indicator for pixel above actual one, 1 for fillColor, initial 1 if just 1 pixel obtained
  bool leftup = 1;
  // indicator for pixel underneath actual one, 1 for fillColor, initial 1 if just 1 pixel obtained
  bool leftdown = 1;
  bool rightup;  // indicator for pixel above actual one, 1 for fillColor
  bool rightdown;  // indicator for pixel underneath actual one, 1 for fillColor
  bool leftcorner = 0;  // indicator for left corner, 0 for not reached yet
  bool rightcorner = 0;  // indicator for right corner, 0 for not reached yet
  vector<int> up;  // array for pixels to continue in the line above the actual one, x value
  vector<int> down;  // array for pixels to continue in the line underneath actual one, y value
  vector<int>::iterator iterate;  // iterator to go through vectors
  int leftx = x;  // starting at actual pixel
  int rightx = x;  // starting at actual pixel
  // if BoundColor hasn't been set, set it to fillColor so it behaves like a "normal" floodfill
  if (boundColor < 0)
  {
    boundColor = fillColor;
  }
```

```
/* filling line in left direction */
while (!leftcorner)  // repeat until left corner is reached
{
  /* probing actual pixel */
  // if actual color is diffrent than fillColor
  if (image->getValue(leftx, y) != fillColor && image->getValue(leftx, y) != boundColor)
  {
    image->setValue(leftx, y, fillColor);  // set it to FillColor
  }

  /* probing pixel above actual one */
  // if pixel above diffrent than fillColor
  if (y > 0
    &&   image->getValue(leftx, y-1) != fillColor
    &&   image->getValue(leftx, y-1) != boundColor)
  {
    if (leftup == 1)
    {
      up.push_back(leftx);
    }
    leftup = 0;  // set state to "diffrent"
  }
  else
  {
    leftup = 1;  // set indicator for pixel above
  }

  /* probing pixel underneath actual one */
  // if pixel underneath diffrent than fillColor
  if (y < (image->getysize()-1)
    &&   image->getValue(leftx, y+1) != fillColor
    &&   image->getValue(leftx, y+1) != boundColor)
  {
    if (leftdown == 1)
    {
      down.push_back(leftx);
    }
    leftdown = 0;  // set state to "diffrent"
  }
  else leftdown = 1;  // set indicator for pixel above

  /* move actual pixel one left if possible */
  // if pixel exists and is not in fillColor
  if (leftx > 0
    &&   image->getValue(leftx-1, y) != fillColor
    &&   image->getValue(leftx-1, y) != boundColor)
  {
```

```
    leftx--;  // decrease x value of actual pixel
  }
  else
  {
    leftcorner = 1;  // can't move any further left -> leave loop
  }
}

/* filling line in right direction */

/* initial readout of pixels above and underneath */
if (y > 0
  &&   image->getValue(rightx, y-1) != fillColor
  &&   image->getValue(rightx, y-1) != boundColor)
{
  rightup = 0;  // initial readout of pixel above
}
else
{
  rightup = 1;
}

if (y < (image->getysize()-1)
  &&   image->getValue(rightx, y+1) != fillColor
  &&   image->getValue(rightx, y+1) != boundColor)
{
  rightdown = 0;  // initial readout of pixel underneath
}
else
{
  rightdown = 1;
}

while (!rightcorner)  // repeat until right corner is reached
{
  /* move actual pixel one right if possible */
  // if pixel exists and is not in fillColor
  if (rightx < (image->getxsize()-1)
    &&   image->getValue(rightx+1, y) != fillColor
    &&   image->getValue(rightx+1, y) != boundColor)
  {
    rightx++;  // decrease x value of actual pixel

    /* probing actual pixel */
    image->setValue(rightx, y, fillColor);  // set it to FillColor

    /* probing pixel above actual one */
    // if pixel above diffrent than fillColor
```

```
    if (y > 0
      &&   image->getValue(rightx, y-1) != fillColor
      &&   image->getValue(rightx, y-1) != boundColor)
    {
      if (rightup == 1)
      {
        up.push_back(rightx);
      }
      rightup = 0;  // set state to "diffrent"
    }
    else
    {
      rightup = 1;  // set indicator for pixel above
    }

    /* probing pixel underneath actual one */
    // if pixel underneath diffrent than fillColor
    if (y < (image->getysize()-1)
      &&   image->getValue(rightx, y+1) != fillColor
      &&   image->getValue(rightx, y+1) != boundColor)
    {
      if (rightdown == 1)
      {
        down.push_back(rightx);
      }
      rightdown = 0;  // set state to "diffrent"
    }
    else
    {
      rightdown = 1;  // set indicator for pixel above
    }
  }
  else
  {
    rightcorner = 1;  // can't move any further right, leave loop
  }
}


/* Call of floodFill recursively, for each pixel in arrays up and down */

/* for pixels above actual line */
iterate = up.begin();
for(int i=0; i<up.size(); i++)
{
  THIS::operator() (image, fillColor, up[i], y-1, boundColor);
  iterate++;
}
```

```
  /* for pixels underneath actual line */
  iterate = down.begin();
  for(int i=0; i<down.size(); i++)
  {
    THIS::operator() (image, fillColor, down[i], y+1, boundColor);
    iterate++;
  }

}

#undef THIS
#undef BASE
```

Literaturverzeichnis

[Ara96] ARAKAWA, Kaoru: Median filter based on fuzzy rules and its application to image restoration. In: *Fuzzy Sets and Systems* 77 (1996), 1, Nr. 1, S. 3–13

[Ben75] BENTLEY, Jon L.: Multidimensional binary search trees used for associative searching. In: *Communications of the ACM* 18 (1975), 9, Nr. 9, S. 509–517

[BEP+01] BALTHASAR, Dirk ; ERDMANN, Thomas ; PELLENZ, Johannes ; REHRMANN, Volker ; ZEPPEN, Jörg ; PRIESE, Lutz: Real-Time Detection of Arbitrary Objects in Alternating Industrial Environments. In: *In Proceedings Scandinavian Conference on Image Analysis* 1 (2001), 6, 321-328. `http://www.uni-koblenz.de/~lb/publications/Balthasar2001RDO.pdf`

[Bre63] BRESENHAM, Jack: An incremental algorithm for digital plotting. In: *ACM National Conference*, 1963

[DVG94] DOBRIN, Bogdan J. ; VIERO, Timo J. ; GABBOUJ, Moncef: Fast watershed algorithms: analysis and extensions. In: DOUGHERTY, Edward R. (Hrsg.) ; ASTOLA, Jaako (Hrsg.) ; LONGBOTHAM, Harold G. (Hrsg.) ; The International Society for Optical Engineering (Veranst.): *Nonlinear Image Processing V* The International Society for Optical Engineering, 1994 (Proceedings of SPIE 2180), S. 209–220

[Elf89] ELFES, A.: Using occupancy grids for mobile robot perception and navigation. In: *Computer* 22 (1989), 6, Nr. 6, S. 46–57

[Fre74] FREEMAN, Herbert: Computer Processing of Line-Drawing Image. In: *ACM Computer Surveys,* (1974), Nr. 6, S. 57–97

[FS97] FOWLER, Martin ; SCOTT, Kendall: *UML Distilled - Applying the Standard Object Modeling Language*. Addison-Wesley, www.addison-wesley.de, 1997

[GGH+04] GEORGIADES, Christina ; GERMAN, Andrew ; HOGUE, Andrew ; LIU, hui ; PRAHACS, Chris ; RIPSMAN, Arlene ; SIM, Robert ; TORRES, Luz-Abril ; ZHANG, Pifu ; BUEHLER, Martin ; DUDEK, Gregory ; JENKIN, Michael ; MILIOS, Evangelos: AQUA: an aquatic walking robot. In: *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems,* (2004)

[Gol99] GOLOMBEK, M. P.: The Mars Pathfinder Mission and Science Results. In: *Workshop on Mars 2001: Integrated Science in Preparation for Sample Return and Human Exploration* (1999)

[GSC+05] GALINDO, C. ; SAFFIOTTI, A. ; CORADESCHI, S. ; BUSCHKA, P. ; FERNANDEZ-MADRIGAL, J.A. ; GONZALEZ, J.: Multi-Hierarchical Semantic Maps for Mobile Robotics. In: *Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS-05)*, 2005

[KPGU04] KRAETZSCHMAR, Gerhard ; PAG
'ES GASSULL, Guillem ; UHL, Klaus: Probabilistic Quadtrees for Variable-Resolution Mapping of Large Environments. In: RIBEIRO, M. I. (Hrsg.) ; SANTOS VICTOR, J. (Hrsg.) ; Instituto Superior Técnico (Veranst.): *Proceedings of the 5th IFAC/EURON Symposium on Intelligent Autonomous Vehicles*. Lisbon, Portugal : Elsevier Science, 7 2004

[MMSB05] MARTINEZ MOZOS, Oscar ; STACHNISS, Cyrill ; BURGARD, Wolfram: Supervised Learning of Places from Range Data using AdaBoost. In: *Proc. of the IEEE Int. Conf. on Robotics & Automation (ICRA)*, 2005

[Mon03] MONTEMERLO, Michael: *FastSLAM: A Factored Solution to the Simultaneous Localization and Mapping Problem with Unknown Data Association*. Pittsburgh, PA, Robotics Institute, Carnegie Mellon University,

http://www.ri.cmu.edu/home.html, Diss., 7 2003. `http://www.ri.cmu.edu/pubs/pub_4434.html`

[PGKKP] PAG
'ES GASSULL, Gassull ; K. KRAETZSCHMAR, Gerhard ; PALM, Günther: Finding Rooms on Probabilistic Quadtrees University of Ulm, Department of Neuroinformatics

[Sam84] SAMET, H.: The quadtree and related hierarchical data structures. In: *ACM Computer Surveys* 16 (1984), S. 187–260

[SHB04] STACHNISS, Cyrill ; HÄHNEL, Dirk ; BURGARD, Wolfram: Exploration with active Loop-Closing for FastSLAM. In: *Proceedings of the 2004 IEEE/RSJ International Conference on Intelligent Robots and Systems* (2004)

[Str00] STROUSTRUP, Bjarne: *The C++ Programming Language*. Addison-Wesley, www.addison-wesley.de, 2000

[Yam97] YAMAUCHI, B.: A frontier-based approach for autonomous exploration. In: *1997 IEEE International Symposium on Computational Intelligence in Robotics and Automation* (1997), S. 146 ff.