Universität des Saarlandes
Fachbereich 6.2 Informatik
Lehrstuhl für künstliche Intelligenz

Fortgeschrittenen-Praktikum

Wegsuche in polygonbasierten Umgebungsmodellen

Sebastian Waßmuth
(sebastian.w@ssmuth.net)

30. März 2006

Betreuer: Dipl.-Inform. Christoph Stahl

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Zielsetzung . . . 1
1.2 Besonderheiten der Fußgängernavigation . . . 1
1.3 Angepasste Umgebungsmodellierung . . . 2

2 Algorithmen für Polygone und Graphen **3**
2.1 Polygonbasierte Algorithmen . . . 3
2.2 Graphbasierte Algorithmen . . . 3
2.3 A* – graphbasierte informierte Suche . . . 3

3 Lösungsansätze zur Generierung von Wegenetzgraphen **4**
3.1 Rasterbasierter Ansatz . . . 4
3.1.1 Wegpunkte . . . 4
3.1.2 Wegkanten . . . 4
3.2 Optimierter Ansatz . . . 5
3.2.1 Wegpunkte . . . 5
3.2.2 Kanten . . . 6
3.3 Zusammenfassung . . . 6

4 Der Wegsuche Algorithmus im Detail **7**
4.1 Erzeugung der Wegpunkte . . . 7
4.1.1 Wegpunkte an “kurzen” begehbaren Polygonkanten . . . 7
4.1.2 Wegpunkte an “langen” begehbaren Polygonkanten . . . 9
4.1.3 Wegpunkte an überstumpfen Ecken . . . 9
4.2 Erzeugung der Wegkanten . . . 9
4.3 Suche eines Weges – A* . . . 11
4.3.1 Die Funktionsweise von A* . . . 12
4.4 Verschönerung des gefundenen Weges . . . 13
4.4.1 Verkürzen des gefundenen Weges . . . 14
4.4.2 Projektion des verkürzten Weges auf das Polygonmodell . 14
4.5 Mögliche Erweiterungen und Optimierungen . . . 14

5 Implementation, Entwicklungsumgebung **15**
5.1 Programmiersprache . . . 15
5.2 Entwicklungsumgebung . . . 15
5.3 Anbindung an Yamamoto . . . 15

6 Dokumentation der PathFinder-Klasse **18**
6.1 Konstruktor 18
6.2 Wegsuche 18
6.3 Verschönerung des gefundenen Weges 19
6.4 Ausgabe des Weges 19

7 Laufzeit und Speicherplatzverbrauch **20**
7.1 Konstruktor 20
7.1.1 Berechnen der Polygon-Nachbarschafts-Beziehungen . . . 21
7.1.2 Berechnen der Wegpunkte 21
7.2 Wegsuche 22
7.2.1 Zuordnen der Start- und Zielpunkte zu einem Polygon . . 22
7.2.2 Initialisierung der Variablen 23
7.2.3 Wegsuche 23
7.3 Verschönerung des gefundenen Weges 25
7.3.1 Verkürzen des gefundenen Weges 25
7.3.2 Projektion des verkürzten Weges auf das Polygonmodell . 26
7.4 Ausgabe des Weges 26

Abbildung 1: Besonderheiten der Fußgängernavigation bei dem Überqueren großer Plätze

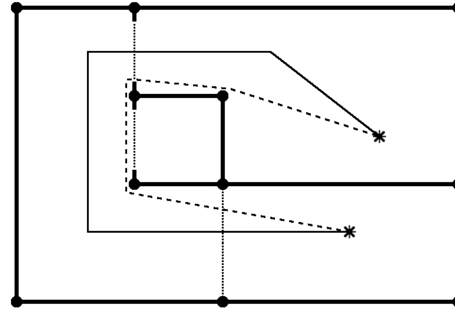

Abbildung 2: Fußgänger benutzen in Gebäuden nicht den mathematisch kürzesten Weg

1 Einleitung

1.1 Zielsetzung

Diese Arbeit beschäftigt sich mit der Wegsuche in Fußgängernavigationssystemen für Innen- und Außenumgebungen. Ziel der Arbeit ist es, kürzeste Wege zwischen zwei beliebigen Punkten eines Umgebungsmodells zu berechnen. Dabei sollen die Besonderheiten der Fußgängernavigation möglichst gut berücksichtigt werden.

1.2 Besonderheiten der Fußgängernavigation

Im Vergleich zu Fahrzeugen sind Fußgänger meist freier in ihrer Bewegung. Sie sind nicht an ein festgelegtes Streckennetz gebunden und können sich innerhalb begehbarer Flächen (wie z.B. Fußgängerzonen) bewegen. Beim Überqueren von großen Plätzen nehmen Fußgänger gerne den direkten Weg und kürzen ab. Ein herkömmliches Navigationssystem auf Basis fester Wegsegmente würde einen unnatürlichen und sehr eckigen Weg berechnen. Dieses Verhalten ist in Abbildung 1 dargestellt. Ein Fußgänger wählt den durchgehend eingezeichneten Weg. Das vereinfachtes Streckennetz ist gestrichelt dargestellt. Siehe dazu auch [1].

Innerhalb von Gebäuden allerdings tritt eher der umgekehrte Fall ein. In langen Gängen führt ein natürlich aussehender Weg nicht direkt an der Wand entlang, sondern mehr durch die Mitte des Gangs. Es wird also nicht unbedingt der mathematisch kürzeste Weg benutzt. Ein Beispiel für dieses Verhalten ist in Abbildung 2 dargestellt.

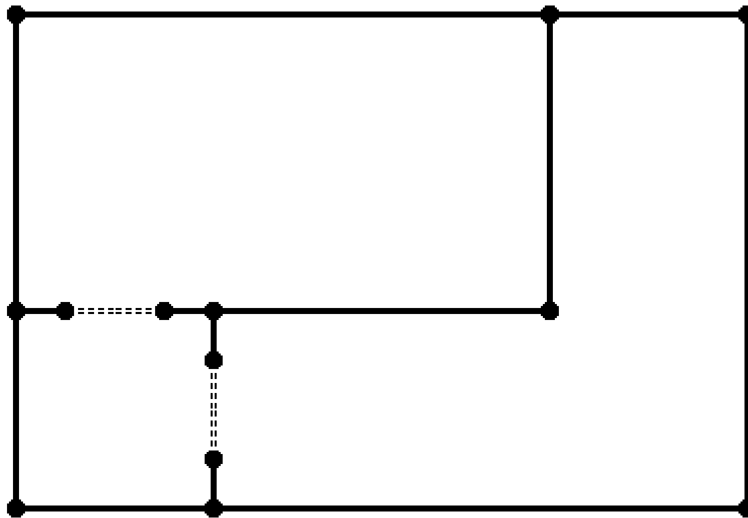

Abbildung 3: Beispiel für ein polygonbasiertes Umgebungsmodell

1.3 Angepasste Umgebungsmodellierung

Die Umgebung wird bei herkömmlichen Navigationssystemen als Graph aus Knoten (Kreuzungen) und Kanten (Straßen) dargestellt. Für ein Fußgängernavigationssystem ist ein solches vereinfachtes Streckennetz als Umgebungsmodell nicht mehr ausreichend. Um jeden beliebigen Weg innerhalb von begehbaren Flächen zu ermöglichen, muss das Modell alle begehbaren Flächen beschreiben und darf diese nicht zu Strecken vereinfachen.

Mit dem grafischen Editor Yamamoto [1, 2] können solche Umgebungsmodelle erstellt werden. Das Modell besteht dort aus einem Netz von benachbarten Polygonen. Die Polygone sind durch gemeinsame Kanten voneinander abgegrenzt. Diese Kanten besitzen Attribute wie z.B. “begehbar für Fußgänger”, “begehbar für Rollstuhlfahrer” oder “nicht begehbar”. Ein Beispiel ist in Abbildung 3 dargestellt. Begehbare Kanten sind gestrichelt eingezeichnet.

2 Algorithmen für Polygone und Graphen

Um nicht einen komplett neuen Algorithmus entwickeln zu müssen, wurden zuerst bereits existierende Algorithmen im Hinblick auf die Problemstellung verglichen. Wichtig waren dabei das Ergebnis (kurzer und schöner Weg), sowie Laufzeit und Speicherplatzverbrauch. Die grundsätzliche Frage war: Ist ein polygon- oder graphbasierter Algorithmus besser geeignet?

2.1 Polygonbasierte Algorithmen

Existierende Algorithmen für die Wegsuche in Polygonen [3, 4, 5] schienen die erste Wahl zu sein, da die Modelle nicht als Graph vorlagen, sondern als ein Netz von Polygonen. Jedoch waren alle diese Algorithmen nur für einzelne Polygone geeignet. Zusätzlich hatten diese Algorithmen Probleme mit Löchern bzw. mit Rundgängen im Polygon. Wenn Löcher erlaubt waren, dann nur in geringer Zahl, um Laufzeit oder Speicherplatzverbrauch annehmbar klein zu halten.

2.2 Graphbasierte Algorithmen

Algorithmen für die Wegsuche in Graphen [6] haben gegenüber der ersten Gruppe deutliche Vorteile in der Laufzeit. Nachteil hier ist die Inkompatibilität zu den vorhandenen Modellen. Um diese Gruppe von Algorithmen verwenden zu können, müssen zuerst aus den polygonbasierten Modellen Wegenetz-Graphen erzeugt werden.

2.3 A* – graphbasierte informierte Suche

Die Wahl des Algorithmus fiel schließlich auf A* [7]. Dies ist ein graphbasierter Algorithmus, der zielgerichtet sucht und immer den kürzesten Weg findet. Eine kurze Erläuterung findet sich in Kapitel 4.3.1.

Das Ziel des Projektes war es nun, auf möglichst geschickte Weise Wegenetz-Graphen (Wegpunkte und Wegkanten) aus den vorhandenen polygonbasierten Umgebungsmodellen zu erzeugen.

Um einen "schönen" Weg finden zu können, muss der Wegenetz-Graph möglichst detailiert sein. Auf der anderen Seite sollen aber auch möglichst wenige Wegpunkte und Wegkanten erzeugt werden, um die abschließende Suche effizient durchführen zu können.

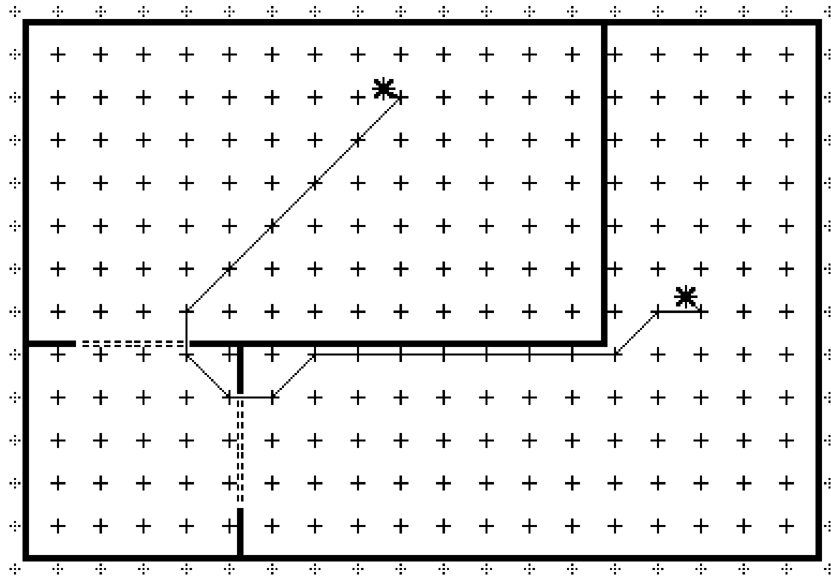

Abbildung 4: Erzeugung der Wegpunkte durch ein Rasterverfahren

3 Lösungsansätze zur Generierung von Wegenetzgraphen

3.1 Rasterbasierter Ansatz

Der erste und einfachste Ansatz zur Erzeugung der Wegpunkte war ein Rasterverfahren: Ähnlich einem Schachbrett werden in regelmäßigen Abständen Wegpunkte erzeugt. Ein Beispiel ist in Abbildung 4 dargestellt. Dabei sind Rasterpunkte innerhalb von Polygonen (= Wegpunkte) schwarz gezeichnet. Rasterpunkte außerhalb des Modells sind gestrichelt gezeichnet.

3.1.1 Wegpunkte

Durch ein Raster werden die Wegpunkte an fest vorgegebenen Stellen erzeugt. Die Rasterbreite kann dabei beliebig gewählt werden. Wegpunkte werden jedoch nur innerhalb von Polygonen erzeugt. Nur modellierte Flächen sind begehbar.

3.1.2 Wegkanten

Um die Anzahl der Wegkanten möglichst gering zu halten, werden nur zwei direkt benachbarte Wegpunkte verbunden, sofern dabei keine Wandkante geschnitten wird (einfacher Test im zweidimensionalen Raum, ob die gewünschte Wegkante eine Wandkante schneidet).

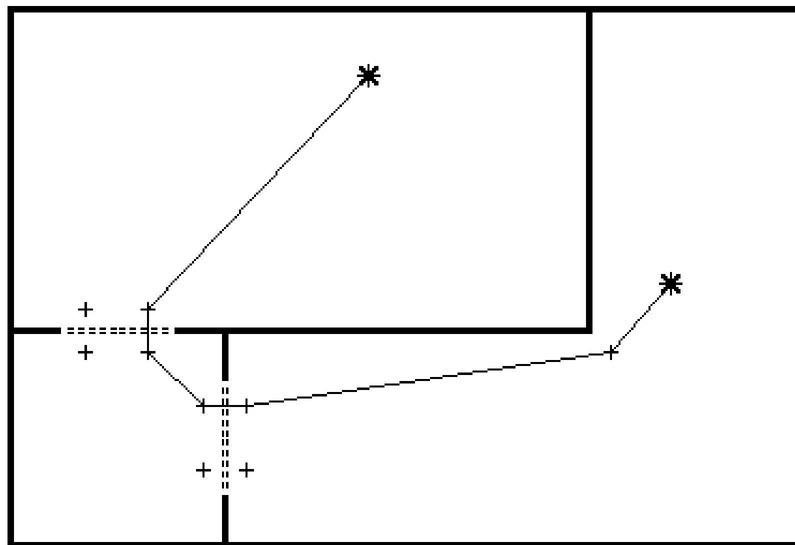

Abbildung 5: Optimierte Erzeugung der Wegpunkte nur an relevanten Stellen (1)

3.2 Optimierter Ansatz

Der nächste Schritt war, Wegpunkte nur an den Stellen zu erzeugen, wo sie auch benötigt werden: an den Stellen, wo die Laufrichtung geändert wird. Ein Beispiel ist in Abbildung 5 dargestellt. Dort wurden Wegpunkte an begehbaren Kanten, sowie an Ecken im Raum erstellt, die einen Winkel von mehr als 180° einschließen.

3.2.1 Wegpunkte

Wegpunkte müssen nur erzeugt werden, wenn die Bewegungsrichtung geändert werden soll:

- beim Übergang von einem Polygon ins nächste (bei der Überquerung von begehbaren Polygonkanten)
- bei Bewegungen innerhalb eines Polygons, das nicht konvex ist (beim Umlaufen einer Polygonecke, die einen Winkel von mehr als 180° einschließt)

Um nicht mit der Wand zu kollidieren werden als Wegpunkte die leicht ins Innere verschobenen Begrenzungspunkte der Polygone verwendet.

Bei begehbaren Kanten wird an beiden Enden und in beiden angrenzenden Polygonen jeweils ein Wegpunkt erzeugt. Pro begehbarer Kante werden also vier Wegpunkte erzeugt.

3.2.2 Kanten

Zwischen zwei Wegpunkten wird eine Wegkante erzeugt, wenn zwischen beiden Punkten eine Sichtlinie besteht (einfacher Test im zweidimensionalen Raum, ob die gewünschte Wegkante eine Wandkante schneidet).

3.3 Zusammenfassung

Der Rasterbasierte Ansatz

hat deutliche Nachteile. Bei großem Rasterabstand ist der gefundene Weg sehr "zackig" und entspricht keinesfalls einem "normalen" Weg, den ein Mensch gehen würde. Andererseits werden bei einem kleinem Rasterabstand sehr viele Wegpunkte und Wegkanten erzeugt, was die Suche ineffizient macht.

Obwohl vor allem bei der Laufzeit noch erhebliches Verbesserungspotential besteht, z.B. Space-Partition-Tree der Wandkanten aufbauen, um nicht immer sämtliche Polygonkanten auf Schnitt mit dem gewünschten Weg testen zu müssen, blieb es bei einer kleinen Fallstudie. Auf Grund der schlechten Qualität des gefundenen Weges ist dieser Ansatz leider ungeeignet.

Der Optimierte Ansatz

ist in allen Bereichen überlegen. Bezüglich der Weglänge wird ein optimaler Weg gefunden. Die Qualität des Weges ist vergleichsweise gut. Außerdem werden deutlich weniger Wegpunkte erzeugt, was vor allem der Laufzeit zu Gute kommt.

Genau genommen ist die Länge des Weges im mathematischen Sinn nicht optimal. Unter Berücksichtung der in Kapitel 1 kurz erläuterten Besonderheiten bei der Fußgängernavigation wird der Weg länger als nötig. Diese "Umwege" sind aber vernachlässigbar klein, sodass der Weg trotzdem als optimal bezeichnet werden kann.

Auch bei diesem Ansatz besteht noch erhebliches Verbesserungspotential. Sowohl bei der Qualität des gefundenen Weges (bei Türen genau die Mitte benutzen), als auch bei der Laufzeit (nicht immer alle Wandkanten auf Schnitt prüfen, sondern nur die, die sich ungefähr in der Nähe der zu testenden Sichtlinie befinden).

In dieser Version des Projektes wurde keine der geplanten Verbesserungen implementiert, da zu diesem Zeitpunkt das Kartenmaterial auf die dritte Dimension für Gebäude mit mehreren Stockwerken erweitert wurde. Das hatte grundlegende Änderungen im Projekt zur Folge.

4 Der Wegsuche Algorithmus im Detail

Die aktuelle Version von PathFinder basiert auf dem in Kapitel 3.2 beschriebenen Ansatz. Hauptsächlich die Erzeugung der Wegkanten wurde verändert, um Modelle mit mehreren Ebenen zu unterstützen. Aber auch die Erzeugung der Wegpunkte wurde optimiert, um den gefundenen Weg noch schöner zu machen.

Zwei Beispiele für das Endergebnis bei einer Wegsuche sind in den Abbildungen 6 und 7 dargestellt. Beide Bilder wurden mit Hilfe von Yamamoto [2] erstellt. Zur Integration von PathFinder in Yamamoto siehe Kapitel 5.3.

4.1 Erzeugung der Wegpunkte

Die erzeugten Wegpunkte haben folgende Eigenschaften:

- Sie befinden sich immer innerhalb von Polygonen.
- Sie sind eindeutig einem Polygon zugeordnet.
- Sie haben einen gewissen Abstand zu den Begrenzungslinien der Polygone. Dieser Abstand ist für jedes Modell individuell konfigurierbar.
- Sie entstehen aus den Begrenzungspunkten oder aus den Mittelpunkten der Begrenzungslinien der Polygone.

Da die Wegpunkte für jedes Polygon separat erzeugt werden, funktioniert der alte Ansatz auch bei Modellen mit mehreren Stockwerken. Verbessert wurde die Erzeugung der Wegpunkte an begehbaren Polygonkanten. Diese werden nun in zwei Gruppen eingeteilt: “kurze” und “lange” Kanten. Durch Angabe eines Schwellenwertes bei der Wegsuche kann bestimmt werden, welche Kanten noch als “kurz” betrachtet werden. Unverändert geblieben ist die Erzeugung der Wegpunkte an überstumpfen Ecken im Polygon.

Die drei verschiedenen Arten der Wegpunkt-Erzeugung werden im Folgenden näher erläutert. Ein Beispiel ist in Abbildung 8 dargestellt. Der Abstand der Wegpunkte zu den Wandkanten ist schraffiert.

4.1.1 Wegpunkte an “kurzen” begehbaren Polygonkanten

Bei schmalen Durchgängen (z.B. Türen in Gebäuden) benutzt ein Fußgänger vorzugsweise die Mitte zum Durchqueren. Damit auch PathFinder dieses Verhalten zeigt, war es notwendig, die Erzeugung der Wegpunkte an solchen Kanten zu modifizieren. Anstatt an beiden Enden der Kante jeweils zwei Wegpunkte zu erzeugen, werden nur in der Mitte der Kante zwei Wegpunkte erzeugt. Jedem der angrenzenden Polygone wird dabei ein Wegpunkt zugeordnet.

Bei einer “kurzen” Kante werden insgesamt zwei Wegpunkte erstellt.

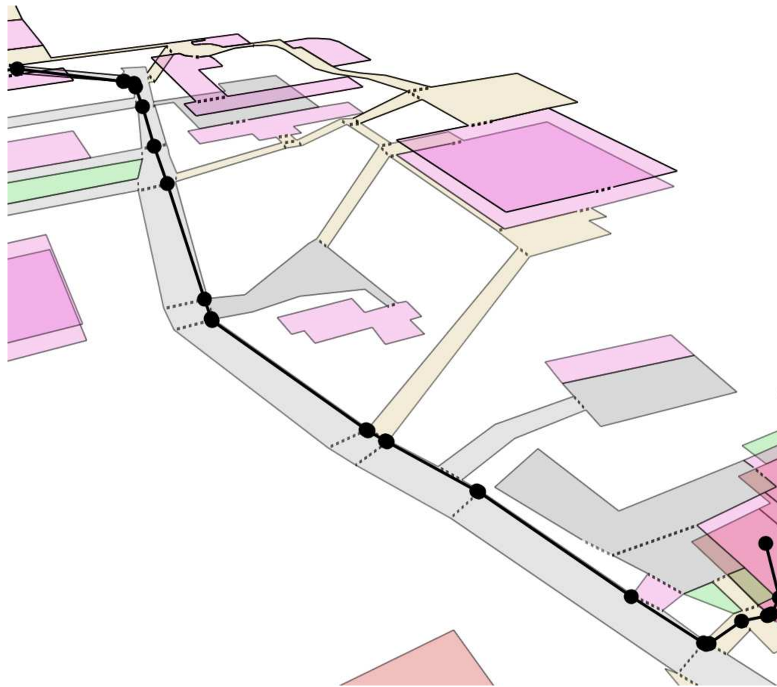

Abbildung 6: Wegsuche auf dem Campus der Universität Saarbrücken

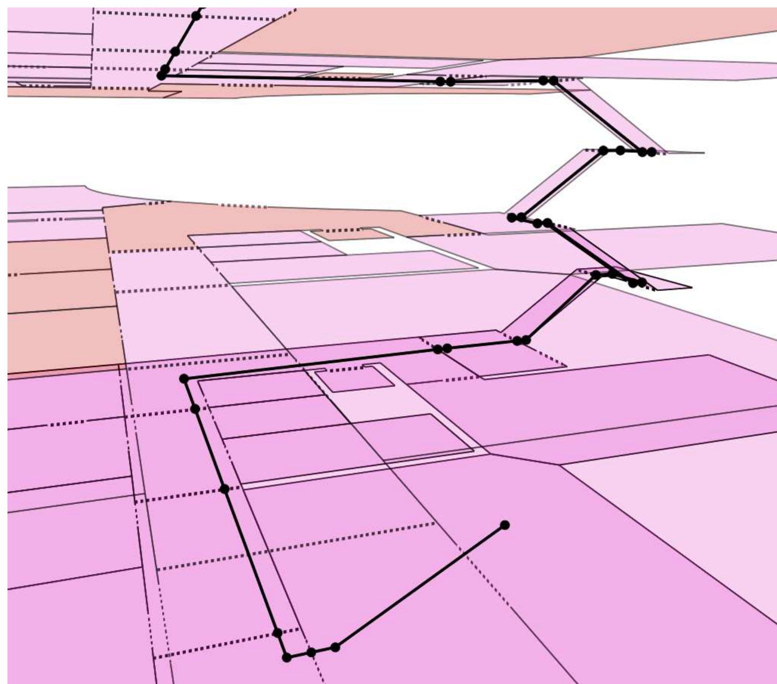

Abbildung 7: Wegsuche über drei Stockwerke im DFKI Saarbrücken

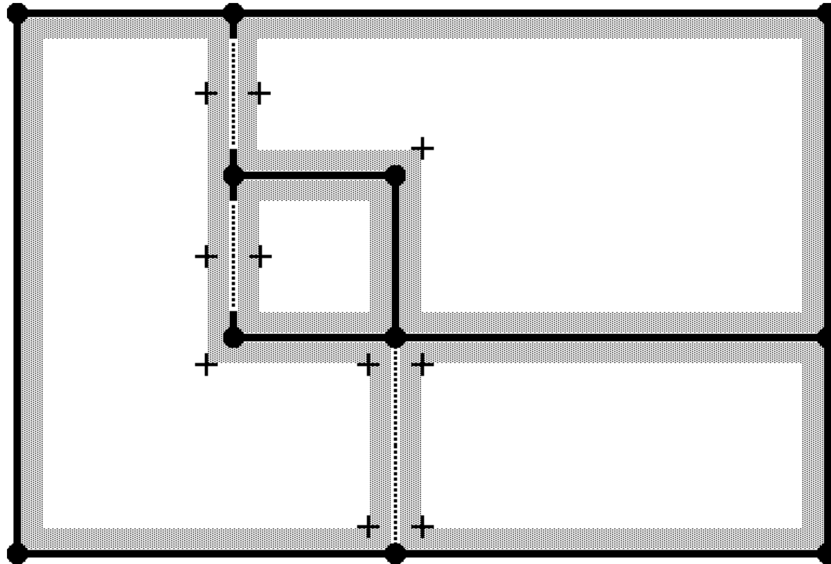

Abbildung 8: Optimierte Erzeugung der Wegpunkte nur an relevanten Stellen (2)

4.1.2 Wegpunkte an "langen" begehbaren Polygonkanten

Breite Durchgänge werden von Fußgängern an beliebigen Stellen durchquert. Es reicht aus, jeweils an den Enden der Kante zwei Wegpunkte zu erzeugen. Diese ermöglichen das Umlaufen der Begrenzungen des Durchganges. Diese Abgrenzungen haben unter Umständen keine Entsprechung in der realen Welt. Es kann sich dabei um modellinterne Trennlinien handeln, die große Bereiche in kleine Polygone aufteilen.

Bei einer "langen" Kante werden insgesamt vier Wegpunkte erstellt.

4.1.3 Wegpunkte an überstumpfen Ecken

Damit Wege um überstumpfe Polygonecken herumführen können, müssen auch an diesen Stellen Wegpunkte erzeugt werden.

Bei überstumpfen Ecken im Polygon wird ein Wegpunkt erstellt.

4.2 Erzeugung der Wegkanten

In Abbildung 9 sind alle Wegkanten eines Modells eingezeichnet. Neben den modellabhängigen Wegpunkten sind bereits Start- und Zielpunkt einer Wegsuche integriert. Die Bedingungen für die Erzeugung von Wegkanten wird nachfolgend näher erläutert.

Wegen der Erweiterung der Umgebungsmodelle auf drei Dimensionen musste

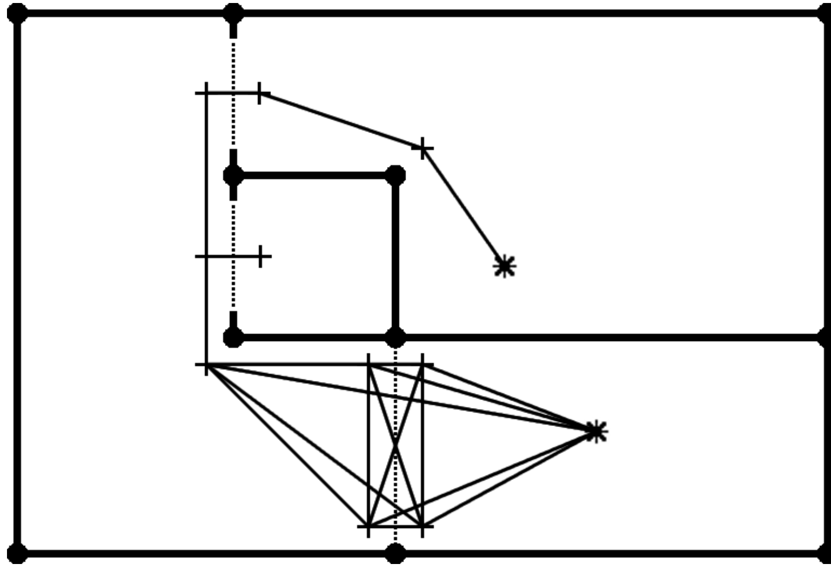

Abbildung 9: Erzeugung der Wegkanten

die Erzeugung der Wegkanten grundlegend verändern werden. Da Polygonkanten keine Höhe haben, werden bei windschiefen Strecken keine Überschneidungen mehr festgestellt. Potentielle Wegkanten (= Verbindungslinien zweier Wegpunkte) müssen aber auf Schnitt mit nicht begehbaren Polygonkanten (z.B. Wände, Abhänge oder Flüsse) überprüft werden. Um dieses Problem zu lösen, werden die zu testenden Strecken jeweils auf die x-y-Ebene (Vernachlässigung der z-Koordinate) projeziert. Diese Vorgehensweise erzeugt jedoch ein neues Problem: Wegkanten im dritten Stockwerk überschneiden sich nun fälschlicherweise mit Polygonkanten aus dem ersten Stockwerk.

Um das zu verhindern, ist die Erzeugung einer Wegkante an eine zusätzliche Bedingung geknüpft: Damit zwei Wegpunkte durch eine Kante verbunden sind, müssen die beiden Punkte entweder im selben Polygon liegen, oder aus benachbarten Polygonen stammen. Dabei gelten zwei Polygone als benachbart, wenn sie eine gemeinsame Kante besitzen, die begehbar ist (z.B. Tür oder Durchgang).

Die veränderte Erzeugung der Wegkanten hat einen positiven und einen negativen Nebeneffekt: Zum einen werden bei entsprechender Modellierung[1] nur noch linear viele Wegkanten bezüglich der Anzahl der Polygone erzeugt. Zum anderen ist es aber nun nicht mehr möglich, ein Polygon zu durchqueren, ohne einen darin enthaltenen Wegpunkt zu benutzen. Das lässt den Weg vor allem in

[1]Die Anzahl der erzeugten Wegkanten wächst quadratisch bezüglich der Anzahl der Punkte pro Polygon und auch bezüglich der Anzahl der Nachbarn pro Polygon. Sind diese Werte sehr klein (typisch erreichbare Werte sind jeweils 10 und 5) gegenüber der Gesamtanzahl der Polygone, können deren Quadrate als Konstanten angesehen werden.

```
bool findPath(startPoint, destinationPoint)
{
  openList.add(startPoint);
  while (openList.isNotEmpty())
  {
    currentPoint = openList.removeBestPoint();
    if (currentPoint == destinationPoint)
      break;
    openList.addSuccessors(currentPoint);
    switchToClosedList(currentPoint);
  }
  return (currentPoint == destinationPoint);
}
```

Abbildung 10: A* – die Hauptschleife (Pseudo Code)

langen Fluren "zackig" aussehen und macht eine abschließende Verschönerung notwendig.

Damit Wege nicht direkt an Wänden vorbeiführen, müssen Wegkanten einen gewissen Abstand zu nicht begehbaren Polygonkanten einhalten. Dieser Mindestabstand gilt auch für Wegpunkte und wurde bereits in Kapitel 4.1 angesprochen.

4.3 Suche eines Weges – A*

Die Wegpunkte werden alle im Voraus berechnet. Die Wegkanten werden erst zur Suchzeit dynamisch berechnet. Dadurch wird die Suche beschleunigt, da nur benötigte Kanten berechnet werden. Die Erzeugung von Wegkanten ist im Vergleich zur Erzeugung von Wegpunkten sehr langsam, da die Verbindung zweier Wegpunkte auf Schnitt mit mehreren Polygonkanten (alle nicht begehbaren Kanten von ein bzw. zwei Polygonen) getestet werden muss.

Der Wegenetzgraph, in dem A* einen Weg suchen kann, besteht aus den Wegpunkten und Wegkanten. Die Funktionsweise von A* ist in Abbildung 10 dargestellt und wird im Folgenden noch kurz erläutert. Detailierte Erklärungen finden sich z.B. in [6].

Das Ergebnis der Wegsuche besteht aus einer Liste von Wegpunkten. Erster Punkt ist der Startpunkt der Suche, letzter Punkt der Zielpunkt. Start- und Zielpunkt einer Wegsuche sind wie auch die modellabhängigen Wegpunkte in der Knotenmenge des Wegenetzgraphen enthalten.

4.3.1 Die Funktionsweise von A*

Der A*-Algorithmus [7] ist eine zielgerichtete Suche. Dabei wird eine Heuristik verwendet, um Punkte zu finden, die am wahrscheinlichsten zum Ziel führen. Es wird immer der kürzeste Weg gefunden.

Wegpunkte

Die Wegpunkte lassen sich in drei Kategorien einteilen:

- noch nicht besucht (enthält zu Beginn der Suche alle Punkte, außer dem Startpunkt)
- besucht, ein Weg zu diesem Wegpunkt (eventuell nicht der beste) wurde gefunden ("open list", enthält zu Beginn der Suche den Startpunkt)
- abschließend besucht, der kürzeste Weg zu diesem Wegpunkt wurde gefunden ("closed list")

Wegsuche

Solange die "open list" Punkte enthält, wird daraus in einer Schleife ein "bester Punkt" ausgewählt. Falls der Zielpunkt gefunden wurde, ist die Suche beendet. Andernfalls werden der "open list" neue Punkte hinzugefügt: alle noch nicht besuchten Punkte, die vom aktuellen Punkt aus durch Benutzen einer Wegkante erreicht werden können. Der aktuelle Punkt wird der "closed list" hinzugefügt und die Schleife beginnt wieder von vorne.

Was ist der "beste Punkt"?

Jedem Punkt werden zwei Weglängen zugeordnet:

- bisherige Länge des Weges vom Startpunkt bis zu diesem Punkt
- geschätzte Restlänge des Weges bis zum Ziel (wird immer unterschätzt, Verwendung des euklidischen Abstandes als Heuristik)

Der Punkt mit der geringsten Summe aus beiden Werten ist der "beste Punkt". Wahrscheinlich führt der kürzeste Weg über diesen Punkt. Diese Funktionalität ist in der Funktion `removeBestPoint()` gekapselt.

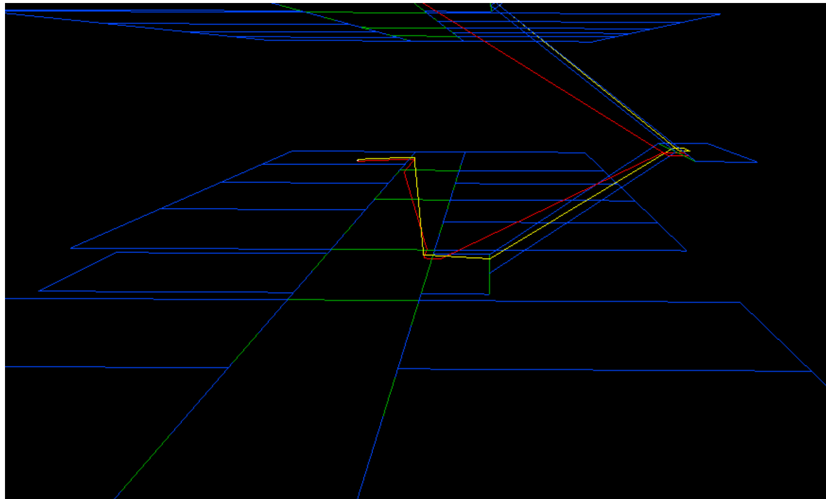

Abbildung 11: Verschönerung des gefundenen Weges

Abschließende Betrachtung eines Punktes

Zuerst werden noch nicht besuchte Punkte in die "open list" eingefügt und bereits besuchte Punkte aktualisiert. Es kann vorkommen, dass zu einem bereits besuchten Punkt (schon in der "open list" enthalten) ein kürzerer Weg gefunden wird. Dann muss die Länge des Weges bis zu diesem Punkt aktualisiert werden. Zum Schluss wird der Punkt in die "closed list" eingefügt und damit als abschließend besucht markiert. Diese Funktionalität ist in den Funktionen `addSuccessors()` und `switchToClosedList(Point)` gekapselt.

"open list"

Die "open list" ist eine Prioritätswarteschlange, bei der die Operationen `insert` (einfügen eines Elementes) und `removeMinimum` (entfernen des "besten Punktes") sehr häufig ausgeführt werden. Die verwendete Datenstruktur (binärer Heap) benötigt für diese Operationen jeweils logarithmische Laufzeit.

4.4 Verschönerung des gefundenen Weges

Durch die spezielle Erzeugung des Wegenetzgraphen wird nicht direkt der gewünschte Weg gefunden. Nach der eigentlichen Wegsuche wird der anfänglich gefundene Weg daher in zwei Schritten verschönert. Beide Schritte sind in Abbildung 11 dargestellt. Der anfänglich gefundene Weg ist rot eingezeichnet, der verschönerte Weg gelb. Dieses Bild stammt aus der selbsterstellten Entwicklungsumgebung unter Linux.

4.4.1 Verkürzen des gefundenen Weges

In langen Fluren, die durch mehrere Polygone modelliert sind, wird in jedem durchlaufenen Polygon mindestens ein Wegpunkt benutzt. In diesem Schritt werden diese überflüssigen Wegpunkte entfernt, sofern der verkürzte Weg weiterhin gültig ist. Gültige Wege dürfen keine Wandkanten schneiden und müssen einen gewissen Abstand zu ihnen einhalten.

4.4.2 Projektion des verkürzten Weges auf das Polygonmodell

Vor allem an Treppen verläuft der Weg oft durch die Luft und folgt nicht dem gewünschten Höhenverlauf. In diesem Schritt werden an Polygon-Übergängen zusätzliche Wegpunkte erstellt, um den Weg auf die Ebene der durchquerten Polygone zu projezieren.

4.5 Mögliche Erweiterungen und Optimierungen

Folgende Punkte können noch weiter verbessert werden. Hier eine Auswahl:

- Den derzeit verwendeten binären Heap durch eine effizientere Datenstruktur ersetzen. Ein Fibonacci Heap hat für die benötigten Operationen z.B. amortisiert konstante Laufzeit.
- Wahlweise auch alle Wegkanten im Voraus berechnen und den Wegenetzgraphen abspeichern. Dies benötigt mehr Speicherplatz, beschleunigt aber die eigentliche Wegsuche.
- A* modifizieren, um besser auf leicht geänderte Bedingungen reagieren zu können. Geht der Benutzer einen anderen Weg als den vorgeschlagenen, muss die derzeitige Implementierung den Weg komplett neu berechnen.
- Der gefundene Weg ist als Streckenzug immer noch ein wenig “kantig”. Durch Anzeige als Spline können die Ecken abgerundet werden und entsprechen mehr dem menschlichen Fortbewegen.

5 Implementation, Entwicklungsumgebung

5.1 Programmiersprache

PathFinder ist komplett in C++ geschrieben. An externen Bibliotheken wird lediglich die `STL` (Standard Template Library) benutzt. Das systemunabhängige Design ermöglicht den Betrieb von PathFinder auch unter "kleinen" Betriebssystemen, wie zum Beispiel Windows CE auf Handheld PCs. PathFinder ist in der aktuellen Version sowohl unter Linux als auch unter Windows lauffähig.

5.2 Entwicklungsumgebung

PathFinder wurde unter Linux (Debian) entwickelt. Als Compiler wurde g++ in der Version 3.3.6 verwendet. Die verwendeten Umgebungsmodelle wurden mit dem Modellierungstool Yamamoto erstellt und liegen im YML-Format vor. Dieses Format basiert auf XML und nennt sich davon abgeleitet YML. Yamamoto ist in C# geschrieben und nur unter Windows lauffähig. Insbesondere für Linux gab es noch keine anderen Programme, die dieses Dateiformat unterstützen.

Um die Umgebungsmodelle und insbesondere auch die Ergebnisse der Wegsuche unter Linux betrachten zu können, wurde eine eigene Entwicklungsumgebung erstellt. Sie ist in der Lage die im YML-Format gespeicherten Modelle zu lesen. Die grafische Ausgabe ist OpenGL basiert und kann verschiedene Datensätze anzeigen. Zur Auswahl stehen:

- das Umgebungsmodell (Punkte, begehbare Kanten und nicht begehbare Kanten können jeweils getrennt angezeigt werden)
- der gefundene Weg (der anfänglich gefundene Weg, sowie die verschönerten Wege können getrennt angezeigt werden, siehe Kapitel 4.4)
- die Wegpunkte (nach einer Wegsuche werden die Punkte je nach Besuchsstatus farblich gekennzeichnet: nicht besucht, besucht oder abschließend besucht)
- die Wegkanten

Ein Beispiel für die grafische Ausgabe ist in Abbildung 11 zu sehen.

5.3 Anbindung an Yamamoto

Die Portierung von PathFinder auf Windows war recht unkompliziert. Nur wenige Zeilen Code mussten geändert werden, um ein lauffähiges Windows-Programm

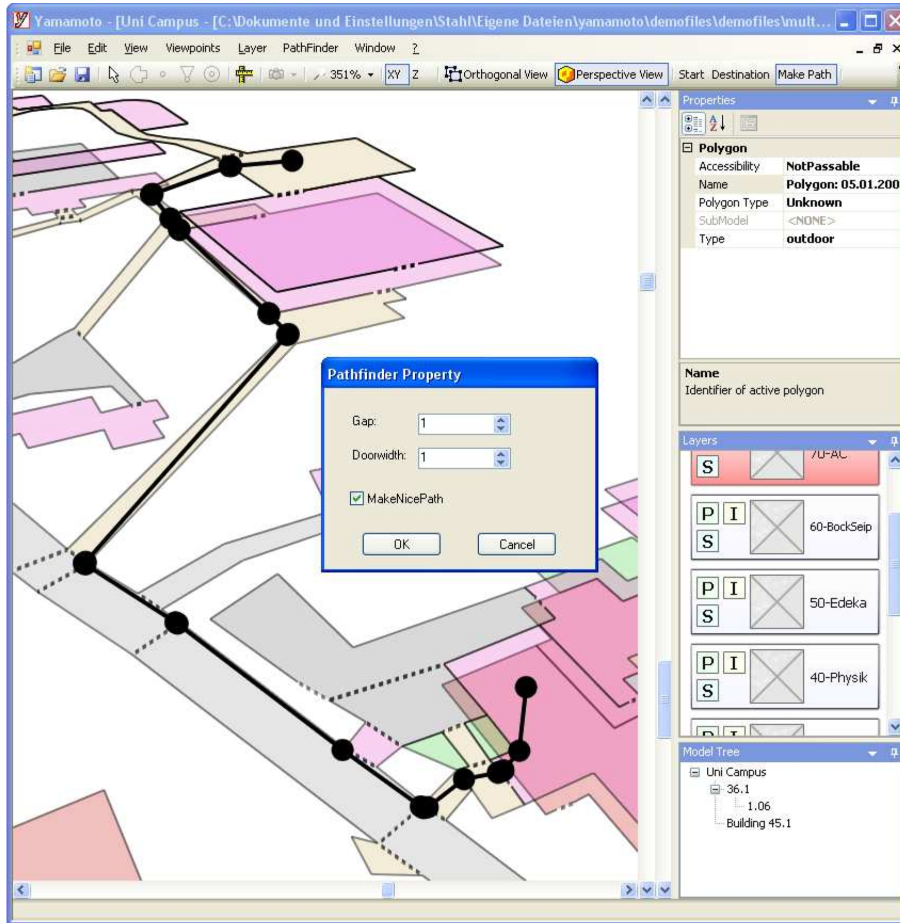

Abbildung 12: Anbindung von PathFinder an Yamamoto

zu erstellen. Recht aufwendig dagegen war die Anbindung an Yamamoto. Zur nahtlosen Integration wurde ein C# Interface für PathFinder benötigt. Als recht schwierig gestaltete sich dabei die Realisierung der Schnittstelle zwischen unmanaged Code (C++) und managed Code (C#).

Yamamoto benutzt Generics, die erst ab Visual Studio 2005 verfügbar sind. Zum Zeitpunkt des Projektes existierte von Visual Studio 2005 aber nur eine Beta Version. Diese unterstützte noch keine gemischten Projekte aus C++ und C#.

Als Übergangslösung wurde eine mit Visual Studio 2003 erstellte Bibliothek (`pathfinder.dll`) um eine “Wrapper-Klasse” erweitert. Diese Klasse “übersetzt” die C++ Variablen und Methoden von PathFinder zu C# kompatiblen Variablen und Methoden. Das in Yamamoto integrierte C# Interface von PathFinder benutzt die Funktionalität dieser Bibliothek, um eine C# Klasse “PathFinder” zu realisieren.

In Abbildung 12 ist das Ergebnis zu sehen. In der rechten Ecke der Toolbar von Yamamoto befinden sich drei Buttons: 'Start', 'Destination' und 'Make Path'. Damit können Start- und Zielpunkt der Wegsuche festgelegt werden, sowie eine Wegsuche gestartet werden. Zu Beginn einer Suche erscheint ein Fenster (auf der Abbildung in der Mitte zu finden) um die Parameter der Wegsuche nach Belieben einstellen zu können.

Der Parameter 'Gap' legt dabei den Abstand fest, den der Weg zu den Wandkanten mindestens haben muss. Der Parameter 'Doorwidth' gibt den Schwellenwert an, bis zu dem begehbare Polygonkanten als "kurz" betrachtet werden. Längere begehbare Polygonkanten werden als "lang" betrachtet. Siehe dazu auch Kapitel 4.1. 'MakeNicePath' gibt an, ob der anfänglich gefundene Weg direkt ausgegeben werden soll, oder mit den in Kapitel 4.4 beschriebenen Methoden verschönert werden soll.

Abhängigkeiten von pathfinder.dll

Die mit Visual Studio 2003 erstellte Bibliothek mit COM-Interface ist von anderen Systembibliotheken abhängig. Diese müssen ebenfalls auf dem Zielsystem installiert werden, sofern sie nicht schon vorhanden sind. Im Einzelnen sind dies:

- `advapi32.dll`, Advanced Windows 32 Base API
- `kernel32.dll`, Windows NT BASE API Client DLL
- `mscoree.dll`, Microsoft .NET Runtime Execution Engine
- `msvcr71d.dll`, Microsoft C Runtime Library
- `ntdll.dll`, NT Layer DLL
- `rpcrt4.dll`, Remote Procedure Call Runtime

6 Dokumentation der PathFinder-Klasse

In diesem Kapitel wird nur das C# Interface beschrieben. Das umfangreichere C++ Interface ist, wie auch das C# Interface, ausreichend im Quellcode dokumentiert.

Neben grundlegenden Datentypen wie `bool`, `int` und `float` werden folgende Datentypen aus dem Namensraum `Yamamoto` verwendet:

`Model` repräsentiert ein Modell mit Punkten, Kanten und Polygonen

`Point` repräsentiert einen Punkt im dreidimensionalen Raum. Die wichtigsten Eigenschaften sind die `X`, `Y` und `Z` Koordinaten. Andere Eigenschaften von `Point` werden innerhalb von PathFinder nicht benutzt.

Eine genaue Beschreibung dieser Datentypen findet sich in [2].

6.1 Konstruktor

```
PathFinder(Model model, float gap, float doorWidth);
```

Der Kontruktor initialisiert die PathFinder-Klasse mit allen nötigen Daten und berechnet die Wegpunkte. Für eine Wegsuche muss dann nur noch `FindPath` aufgerufen werden.

`model` das Polygon-Modell der Karte in der ein Weg gesucht werden soll. Dort sind alle relevanten Punkte, Kanten und Polygone enthalten.

`gap` der Abstand zwischen Weg und Wänden, der mindestens eingehalten wird. Wird dieser Abstand zu groß gewählt, können kurze begehbare Polygonkanten unpassierbar werden.

`doorWidth` die Breite, bis zu der eine begehbare Polygonkante als Tür betrachtet wird. Siehe Kapitel 4.1 für Details zur Erzeugung der Wegpunkte.

6.2 Wegsuche

```
bool FindPath(Point start, Point destination);
```

Sucht einen Weg vom Startpunkt zum Zielpunkt.

`start` Beginn der Wegsuche

`destination` Ziel der Wegsuche

Der Rückgabewert gibt an, ob ein Weg gefunden wurde.

6.3 Verschönerung des gefundenen Weges

```
void MakeNicePath();
```

Verschönert den gefundenen Weg. Falls kein Weg gefunden wurde, erzeugt diese Funktion eine Fehlermeldung. Der Aufruf dieser Funktion ist optional.

6.4 Ausgabe des Weges

```
List<Point> GetPath();
```

Diese Funktion gibt den gefundenen Weg als eine Liste von Punkten zurück. Falls kein Weg gefunden wurde, erzeugt diese Funktion eine Fehlermeldung.

Da `MakeNicePath` den gefundenen Weg in der Regel verändert, unterscheiden sich die Rückgabewerte von `GetPath` vor und nach dem Aufruf von `MakeNicePath`.

7 Laufzeit und Speicherplatzverbrauch

Im Folgenden wird auf einige Werte Bezug genommen, die durch die Eingabe festgelegt sind:

points Anzahl der Punkte im Modell

edges Anzahl der Polygonkanten im Modell

polygons Anzahl der Polygone im Modell

ppp durchschnittliche Anzahl Punkte pro Polygon (= durchschnittliche Anzahl Kanten pro Polygon, $\neq \frac{points}{polygons}$ da Punkte in mehreren Polygonen enthalten sein können)

waypoints Anzahl der erzeugten Wegpunkte, kann je nach Modell stark variieren (bei entsprechender Modellierung ist jedoch $waypoints \in \mathrm{O}(points)$)

wppp durchschnittliche Anzahl Wegpunkte pro Polygon (= $\frac{waypoints}{polygons}$ da Wegpunkte immer eindeutig einem Polygon zugeordnet sind)

npp durchschnittliche Anzahl Nachbarn, die ein Polygon hat

pathlength Anzahl der benutzten Wegpunkte auf dem gefundenen Weg (makeNicePath ändert diese Anzahl unter Umständen)

Die Werte ppp, $wppp$ und npp ändern sich in der Regel nicht mit der Größe des Eingabe-Modells, sondern hängen nur von der Art der Modellierung ab. Bei entsprechender Modellierung übersteigen sie eine gewisse Schranke nicht. Daher können diese Werte als konstant angesehen werden.

7.1 Konstruktor

Yamamoto stellt zur Zeit keine Möglichkeit zur Verfügung, über alle Kanten des Modells zu iterieren. Da PathFinder diese Funktionalität benötigt, muss das Modell zunächst in PathFinder-Format konvertiert werden. Dieser Schritt wird hier nicht näher betrachtet.

Um zwischen zwei beliebigen Punkten einen Weg suchen zu können, muss der Konstruktor zuerst die Nachbarschafts Beziehungen zwischen den Polygonen und die Wegpunkte berechnen.

Wie nachfolgend beschrieben, benötigt der Konstruktur lineare Laufzeit und linearen Speicherplatz bezüglich $polygons$, $edges$ und $points$.

$$\begin{aligned} \text{Laufzeit:} \quad & \mathrm{O}(2 * ppp * polygons + edges) \\ \text{Speicherplatz:} \quad & \mathrm{O}(npp * polygons + 2 * edges + waypoints) \end{aligned}$$

7.1.1 Berechnen der Polygon-Nachbarschafts-Beziehungen

Die derzeitige Implementierung ist einfach gehalten und benötigt quadratische Laufzeit bezüglich $polygons$.

Bei jedem Polygon wird jedes andere Polygon auf Nachbarschaft getestet (Laufzeit $\mathrm{O}(polygons^2)$). Bei einem Test auf Nachbarschaft wird jede Kante des ersten Polygons mit jeder Kante des zweiten Polygons verglichen (Laufzeit $\mathrm{O}(ppp^2)$). Außer dem Speicherplatz für Nachbarschafts-Beziehungen (Speicherplatz $\mathrm{O}(polygons * npp)$) wird kein zusätzlicher Speicherplatz verbraucht.

$$\begin{aligned} \text{Laufzeit:} \quad & \mathrm{O}(polygons^2 * ppp^2) \\ \text{Speicherplatz:} \quad & \mathrm{O}(polygons * npp) \end{aligned}$$

Es würde aber auch schneller gehen. Durch leichte Erhöhung des Speicherplatzverbrauchs lässt sich eine lineare Laufzeit bezüglich $polygons$ erreichen.

Im ersten Schritt werden alle Kanten von allen Polygonen betrachtet (manche Kanten daher mehrfach, Laufzeit $\mathrm{O}(polygons * ppp)$) und für jede begehbare Kante gespeichert, zu welchen Polygonen sie gehört (Speicherplatz $\mathrm{O}(2 * edges)$, da eine Kante maximal zwei Polygone verbindet).

Im zweiten Schritt werden alle Kanten betrachtet (Laufzeit $\mathrm{O}(edges)$) und mit den vorher erhaltenen Informationen festgestellt, welche Polygone benachbart sind. Dabei wird der Speicherplatz für die Nachbarschafts Beziehungen (Speicherplatz $\mathrm{O}(polygons * npp)$) verbraucht.

$$\begin{aligned} \text{Laufzeit:} \quad & \mathrm{O}(polygons * ppp + edges) \\ \text{Speicherplatz:} \quad & \mathrm{O}(polygons * npp + 2 * edges) \end{aligned}$$

Zur Berechnung von Laufzeit und Speicherplatz für den Konstruktor ist letztere Methode zu Grunde gelegt.

7.1.2 Berechnen der Wegpunkte

Es werden alle Punkte von allen Polygonen betrachtet (manche Punkte daher mehrfach). Erfüllt ein Punkt die Bedingungen für eine Wegpunkt-Erzeugung, wird im betrachteten Polygon ein Wegpunkt erstellt.

Das Berechnen der Wegpunkte benötigt lineare Laufzeit und linearen Speicherplatzverbrauch bezüglich $polygons$ und $waypoints$.

$$\begin{aligned} \text{Laufzeit:} \quad & \mathrm{O}(polygons * ppp) \\ \text{Speicherplatz:} \quad & \mathrm{O}(waypoints) \end{aligned}$$

7.2 Wegsuche

Die Wegsuche gliedert sich in drei Abschnitte, die im Nachfolgenden noch genauer beschrieben werden.

- *Zuordnen der Start- und Zielpunkte zu einem Polygon* benötigt lineare Laufzeit bezüglich der Anzahl der Polygone, erwartungsgemäß ist $polygons < waypoints$
- *Initialisierung der Variablen* benötigt lineare Laufzeit bezüglich der Anzahl der Wegpunkte
- *Ausführen von A** benötigt linear logarithmische Laufzeit bezüglich der Anzahl der Wegpunkte

Bei der asymptotischen Betrachtung der Laufzeit können die beiden ersten Abschnitte vernachlässigt werden, sodass sich für die Wegsuche eine linear logarithmische Laufzeit und ein linearer Speicherplatzverbrauch bezüglich $waypoints$ ergeben.

$$\begin{aligned} \text{Laufzeit:} \quad & \mathrm{O}((npp + 1) * 3 * ppp^2 * waypoints * \log waypoints) \\ \text{Speicherplatz:} \quad & \mathrm{O}(2 * waypoints) \end{aligned}$$

Die nachfolgend verwendeten Begriffe “open list” und “closed list” beziehen sich auf Mengen von Wegpunkten. Der A* Algorithmus verwendet diese Listen, um Wegpunkte in verschiedene Gruppen einzuteilen. Die Funktionsweise des A* Algorithmus ist z.B. in [6] erklärt. Bei der “open list” ist es wichtig, die Operationen `extractMininum()` und `addElement()` effizient durchführen zu können. Daher wurde hierfür als Datenstruktur ein binärer Heap verwendet. Dies ist noch nicht optimal, für zukünftige Versionen von PathFinder besteht hier von Optimierungspotential.

7.2.1 Zuordnen der Start- und Zielpunkte zu einem Polygon

Bedingt durch die verschiedenen Speicherformate des Modells ist es nicht möglich, die Start- und Zielpunkte bereits in Yamamoto einem Polygon zuzuordnen. PathFinder benötigt für Polygone, Kanten und Punkte jeweils eine eindeutige ID, Yamamoto indiziert dagegen nur die Punkte, nicht aber die Polygone oder Kanten.

Um einen Punkt einem Polygon zuzuordnen, werden alle Polygone betrachtet (Laufzeit $\mathrm{O}(polygons)$), und jeweils getestet, ob der Punkt innerhalb dieses Polygons liegt (Laufzeit $\mathrm{O}(ppp)$). Zu diesem Zweck werden Punkt und Polygon auf die x-y-Ebene projeziert (Vernachlässigung der z-Koordinate).

Bei allen Polygonen, die den Punkt enthalten wird anschließend die Abweichung in den z-Koordinaten verglichen. Das Polygon mit der geringsten Abweichung wird ausgewählt.

Ein Punkt ist in maximal $layer$ Polygonen enthalten, wenn $layer$ die Anzahl der Ebenen in einem Modell bezeichnet. In der Regel gilt $layer \ll polygons$. Daher wird im Folgenden $layer$ als so klein angenommen, dass es als konstant betrachtet werden kann. Diese Annahme gilt nicht bei Modellen, die Wolkenkratzer enthalten (dort kann $layer \sim polygons$ gelten).

Dieser Abschnitt benötigt lineare Laufzeit bezüglich $polygons$. Der Speicherplatzverbrauch ist konstant, gespeichert werden lediglich die IDs der beiden Polygone, die Start- und Zielpunkt enthalten (können auch identisch sein).

$$\begin{aligned} \text{Laufzeit:} \quad & \mathrm{O}(polygons * ppp) \\ \text{Speicherplatz:} \quad & \mathrm{O}(1) \end{aligned}$$

Für den nicht betrachteten Fall $layer \sim polygons$ wird der Speicherplatzverbrauch linear bezüglich $polygons$. Die lineare Laufzeit bleibt dabei erhalten.

7.2.2 Initialisierung der Variablen

Für jeden erzeugten Wegpunkt werden Variablen angelegt, die Daten zur folgenden Wegsuche speichern:

Status des Punktes *noch nicht besucht* oder *besucht und auf der Liste der noch zu besuchenden Punkte ("open list")* oder *abschließend besucht ("closed list")*

Pfadlänge die Länge des bisher besten Pfades vom Startpunkt zum aktuellen Punkt

Vorgängerpunkt die ID des Punktes, von dem aus der aktuelle Punkt erreicht wurde

Dieser Abschnitt benötigt lineare Laufzeit und linearen Speicherplatz bezüglich $waypoints$.

$$\begin{aligned} \text{Laufzeit:} \quad & \mathrm{O}(waypoints) \\ \text{Speicherplatz:} \quad & \mathrm{O}(waypoints) \end{aligned}$$

7.2.3 Wegsuche

Die beiden nachfolgenden Schritte werden für jeden Punkt ausgeführt, der abschließend betrachtet wird. Also für jeden vorhandenen Wegpunkt höchstens einmal.

Bestimmung eines Punktes, der abschließend betrachtet werden soll

Sei $g(p)$ die bisherige Pfadlänge vom Startpunkt zu einem Punkt p und sei $h(p)$ die Länge des kürzest möglichen Pfades von p zum Zielpunkt (Luftlinie). Dann wird der Punkt auf der "open list" ausgewählt, der den kleinsten Wert $g(p) + h(p)$ hat.

Bei einem binären Heap das Minimum zu bestimmen benötigt konstante Laufzeit. Das Entfernen benötigt jedoch logarithmische Laufzeit. Da höchstens jeder Wegpunkt einmal im Heap enthalten ist, ergibt sich eine logarithmische Laufzeit bezüglich $waypoints$.

$$\text{Laufzeit:} \quad \mathrm{O}(\log waypoints)$$

Abschließende Betrachtung eines Wegpunktes

In diesem Schritt werden alle Punkte bestimmt, die vom aktuellen Punkt aus direkt erreicht werden können.

Zuerst müssen dazu vom betrachteten Wegpunkt die ausgehenden Wegkanten berechnet werden[2]. Der Test, ob zwischen zwei Wegpunkten p und q aus den Polygonen A und B (können auch identisch sein) eine Kante e existiert, müssen alle Wände (nicht begehbare Polygonkanten) der beiden Polygone A und B auf Schnitt mit e getestet werden (Laufzeit $\mathrm{O}(2 * ppp)$).

Dieser Test muss für den betrachteten Punkt mit allen anderen Punkten aus dem betrachteten Polygon, sowie mit allen Punkten aus den Nachbarpolygonen durchgeführt werden (Anzahl der Punkte $(npp + 1) * ppp$).

Bei jeder gefundenen Kante muss der neu erreichte Wegpunkt in die "open list" eingetragen werden.

- Falls der Wegpunkt bisher noch nicht besucht wurde, muss er neu eingefügt werden. Dies benötigt bei einem binären Heap eine Laufzeit von $\mathrm{O}(\log waypoints)$ (es sind maximal $waypoints$ Elemente im Heap enthalten).

- Falls der Wegpunkt schon besucht wurde, und der neue Weg kürzer als der bisher gefundene ist, muss der Wegpunkt aktualisiert werden. Dazu muss die Position des Wegpunktes im Heap gefunden werden. Die derzeitige Implementierung enthält keine Optimierungen für diesen Schritt und benötigt lineare Laufzeit ($\mathrm{O}(waypoints)$) bezüglich der Größe des Heaps. Eine eventuelle Repositionierung des Wegpunktes im Heap benötigt eine Laufzeit von $\mathrm{O}(\log waypoints)$.

[2] Die derzeitige Implementierung berechnet nur alle Punkte des Wegenetz-Graphen im Voraus, nicht aber die Kanten, siehe Kapitel 4.3.

Bedingt durch die Erzeugung der Wegpunkte und Wegkanten kommt dieser Fall praktisch nie vor. Daher wird die lineare Laufzeit in diesem Schritt vernachlässigt.

Insgesamt ergibt sich für diesen Schritt eine logarithmische Laufzeit bezüglich der Anzahl der Wegpunkte.

$$\text{Laufzeit:} \quad \mathrm{O}((npp + 1) * 3 * ppp^2 * \log waypoints)$$

Zusammenfassung

Die Suche eines Weges zwischen zwei Punkten benötigt linear logarithmische Laufzeit bezüglich $waypoints$. Der Speicherplatz wird für den binären Heap benötigt, der die "open list" darstellt. Dadurch kann der Punkt für die abschließende Betrachtung schneller ausgewählt werden. Da jeder Wegpunkt höchstens einmal im Heap enthalten ist, ergibt sich ein linearer Speicherplatz bezüglich $waypoints$.

$$\begin{aligned} \text{Laufzeit:} &\quad \mathrm{O}((npp + 1) * 3 * ppp^2 * waypoints^2) \\ \text{Speicherplatz:} &\quad \mathrm{O}(waypoints) \end{aligned}$$

7.3 Verschönerung des gefundenen Weges

Der Weg wird in zwei Phasen verschönert. Insgesamt ergibt sich eine quadratische Laufzeit und ein linearer Speicherplatzverbrauch bezüglich $pathlength$ (Länge des ursprünglich gefundenen Weges).

$$\begin{aligned} \text{Laufzeit:} &\quad \mathrm{O}(ppp * pathlength^2) \\ \text{Speicherplatz:} &\quad \mathrm{O}(pathlength) \end{aligned}$$

7.3.1 Verkürzen des gefundenen Weges

Zum Verkürzen des Weges sind drei Schleifen ineinander geschachtelt, die alle über die Länge des Weges (oder Teilstücke davon) iterieren.

- Die äußerste Schleife iteriert über alle Punkte auf dem gefundenen Weg (der nach und nach verkürzt wird).
- Ab dem aktuell betrachteten Punkt der äußersten Schleife iteriert die mittlere Schleife solange über alle nachfolgenden Punkte, bis der Weg an dieser Stelle nicht mehr verkürzt werden kann.

- Der Test, ob ein oder mehrere Punkte eingespart werden können, wird in der innersten Schleife getestet: Dabei muss über alle auf dem aktuellen Teilstück eingesparten Punkte iteriert werden und dabei jeweils getestet werden ob die verkürzte Strecke das Polygon, in dem der aktuelle Punkt enthalten ist, korrekt durchläuft. Ein Polygon wird korrekt durchlaufen, sofern gewünschte Weg keine Wand schneidet und einen gewissen Abstand zu den Wänden einhält. Dieser Test benötigt eine konstante Laufzeit von $\mathrm{O}(ppp)$, da alle Begrenzungslinien des betrachteten Polygons einmal auf Schnitt getestet werden müssen.

Auf den ersten Blick ergibt sich eine kubische Laufzeit von $\mathrm{O}(ppp * pathlength^3)$. Genauer betrachtet iterieren aber die beiden äußeren Schleifen zusammen nur einmal über die gesamte Länge des gefundenen Weges: Falls die mittlere Schleife Punkte einsparen kann (nur dann iteriert diese Schleife über ein Teilstück des Weges, das länger ist als zwei), werden die eingesparten Punkte beim weiteren Ausführen der äußersten Schleife übersprungen (die mittlere Schleife verkürzt den Weg).

Damit ergibt sich eine quadratische Laufzeit bezüglich $pathlength$. Bei der Verkürzung des Weges wird kein Speicherplatz verbraucht.

$$\text{Laufzeit:} \quad \mathrm{O}(ppp * pathlength^2)$$

7.3.2 Projektion des verkürzten Weges auf das Polygonmodell

Es wird angenommen, dass alle Polygone eben sind. Um den Weg auf das Polygonmodell zu projezieren, müssen also innerhalb eines Polygons keine zusätzlichen Wegpunkte eingefügt werden, höchstens an Übergängen von einem Polygon zum nächsten. Im Folgenden wird mit $pathlength$ immer noch die Länge des Weges *vor* dem Verkürzen bezeichnet.

Es wird an jedem Übergang von einem Polygon zum nächsten maximal ein zusätzlicher Wegpunkt erzeugt. Da maximal $pathlength$ solcher Übergänge existieren (bedingt durch die Konstruktion der Wegkanten enthält jedes Polygon, dass vom Weg durchlaufen wird, mindestens einen benutzten Wegpunkt), ergeben sich die angegebenen Werte für Laufzeit und Speicherplatz.

$$\begin{aligned} \text{Laufzeit:} \quad & \mathrm{O}(pathlength) \\ \text{Speicherplatz:} \quad & \mathrm{O}(pathlength) \end{aligned}$$

7.4 Ausgabe des Weges

In einer Schleife wird über alle im Weg enthaltenen Punkte iteriert. Da jeder Wegpunkt einen Zeiger auf seinen Vorgänger besitzt, wird der Weg “rückwärts”

vom Zielpunkt bis zum Startpunkt durchlaufen. Damit die Ausgabe trotzdem beim Startpunkt beginnt, wird bei der Wegsuche “rückwärts” vom Zielpunkt zum Startpunkt gesucht. Der vom Benutzer angegebe Startpunkt ist bei der Suche also der Zielpunkt, und umgekehrt.

Die Ausgabe des Weges erfordert lineare Laufzeit und linearen Speicherplatzbedarf bezüglich $pathlength$.

Laufzeit:	$\mathrm{O}(pathlength)$
Speicherplatz:	$\mathrm{O}(pathlength)$

Abbildungsverzeichnis

1 Besonderheiten der Fußgängernavigation bei dem Überqueren großer Plätze 1
2 Fußgänger benutzen in Gebäuden nicht den mathematisch kürzesten Weg 1
3 Beispiel für ein polygonbasiertes Umgebungsmodell 2
4 Erzeugung der Wegpunkte durch ein Rasterverfahren 4
5 Optimierte Erzeugung der Wegpunkte nur an relevanten Stellen (1) 5
6 Wegsuche auf dem Campus der Universität Saarbrücken 8
7 Wegsuche über drei Stockwerke im DFKI Saarbrücken 8
8 Optimierte Erzeugung der Wegpunkte nur an relevanten Stellen (2) 9
9 Erzeugung der Wegkanten 10
10 A* – die Hauptschleife (Pseudo Code) 11
11 Verschönerung des gefundenen Weges 13
12 Anbindung von PathFinder an Yamamoto 16

Literatur

[1] Christoph Stahl, Jens Haupert, *Taking Location Modelling to new Levels: A Map Modelling Toolkit for Intelligent Environments*, 2nd International Workshop on Location- and Context-Awareness. In: M. Hazas, J. Krumm, T. Strang (Eds.): LoCA 2006, LNCS 3987, pages 74–85, Springer-Verlag Berlin Heidelberg, 2006

[2] Jens Haupert, *Entwicklung eines interaktiven Grafikeditors zur hierarchischen geo-referenzierten Umgebungsmodellierung*, Bachelorarbeit, Saarbrücken, 2005

[3] Sanjiv Kapoor, S.N. Maheshwari, Joseph S. B. Mitchell, *An Efficient Algorithm for Euclidean Shortest Paths Among Polygonal Obstacles in the Plane*

[4] Siu-Wing Cheng, Otfried Cheong, Hazel Everett, Renee van Oostrum, *Hierachical Decompositions and Circular Ray Shooting in Simple Polygons*

[5] Alon Efrat, Stephen G. Kobourov, Anna Lubiw, *Computing Homotopic Shortest Paths Efficiently*, 2002

[6] Stuart Russell, Peter Norvig, *Artificial Intelligence, A Modern Approach, Second Edition*, Informed Search and Exploration, pages 94–110, 2003

[7] Hart P., Nilsson N., Raphael B., *A Formal Basis for the Heuristic Determination of Minimum Cost Paths*. In EEE Trans. on Systems Science and Cybernetics, vol. 4, no. 2, pages 100–107, 1968