Wegfindung in 2 nahezu gleichgroßen 2D-Räumen mit Hindernissen

Mierswa, Daniel
Matrikelnummer: 681696
daniel.b.mierswa@student.hs-rm.de

22. Januar 2014

Inhaltsverzeichnis

1 Einleitung

Die Robotik ist längst ein großes Anwendungsfeld der Informatik. Roboter, als Entitäten der Robotik, werden z.B. stationär in der Industrie oder autonom in Gefahrengebieten genutzt. Dadurch wird ersichtlich, dass die Einsatzbereiche sehr heterogen sind und entsprechend andere Anforderungen an einen Roboter existieren. In Zukunft werden Roboter, auch bedingt durch die wachsende Anzahl an eingebetteten Geräten, die einem im Alltag begegnen, vermehrt in anderen Gebieten erwartet, z.B. im Pflegebereich (Ambient Assited Living) oder Servicebereich, wodurch sie auch Serviceroboter genannt werden. Bei stationären Robotern ist die Umsetzung der Kinematik ein wesentlicher Aspekt und wird in der Regel durch entsprechende Mathematik unterstützt. Für alle Robotertypen ist die Navigation, also insbesondere die Wegfindung bzw. Bahnplanung und die damit verbundene Kollisionsprüfung eine zu lösende Herausforderung und ein Anwendungsgebiet der Informatik. Für die Bahnplanung existieren verschiedene Konzepte, inklusive derer Modelle und Algorithmen. Aufgrund der vielfältigen Möglichkeiten und Lösungsansätze zur Planung und Umsetzung eines mobilen Roboters, entstand im Rahmen der Lehrveranstaltung Robotik im Master-Studiengang Informatik der Hochschule RheinMain, ein Projekt, welches einen konkreten Anwendungsfall umsetzen soll.

2 Aufgabe

Die Aufgabe, die dieses Projekt versucht zu lösen, umfasst die Bahnplanung in zwei nahe zu gleichgroßen 2D-Räumen. Die Räume werden mit festen Blockaden bzw. Gegenständen bestückt und einer Tür bestückt. In diesen Räumen sollen statistisch Wegpunkte verteilt werden, welche dann als Knoten für einen gewichteten Graphen dienen. Anhand dieses Graphen, soll dann mit dem Dijkstra-Algorithmus der kürzeste Pfad von einem Start- zu einem Endpunkt berechnet werden. Falls genügend Bearbeitungszeit zur Verfügung, kann als zusätzliche Alternative der A*-Algorithmus verwendet werden. Nachdem ein Pfad gefunden wurde, soll geprüft werden, ob dieser mit den Wänden des Raumes oder den darin befindlichen Objekten kollidiert. Falls keine Kollision auftritt, wird mittels der Anwendung von Catmull-Rom Splines eine Kurve erstellt, die die entsprechenden Wegpunkte berührt. Erneut wird nach dem Erstellen der Kurve geprüft, ob Kollisionen auftreten. Zur Veranschaulichung der Lösung, soll ein Programm entwickelt werden, welches mindestens die folgenden Funktionalitäten anbietet:

- Darstellung der 2D-Räume und der darin befindlichen Objekte
- Roboter-Entität als kreisförmiges Objekt
- Eingabe des Start- und Zielpunkts
- Verändern der Anzahl der Wegpunkte
- Anzeige der Programmlaufzeit für die Bahnplanung
- Veranschaulichung einer möglichen auftretenden Kollision

Aufgrund der Anforderungen müssen im Programm folgende Aspekte umgesetzt werden:

- Verarbeitung einer Bilddatei eines Raumes
- Erstellen einer Benutzeroberfläche
- Zeichnen einer 2D-Szene inklusive entsprechenden Veränderungen, die durch den Benutzer gesteuert werden
- Messen von Programmlaufzeiten für die Durchführung verschiedener Aktionen

3 Grundlagen

Um die im weiteren Verlauf beschriebenen Lösungsansätze besser zu verstehen, werden in diesem Kapitel die Begriffe, die in der Aufgabenstellung relevant sind, näher erläutert.

3.1 Bahnplanung

Um eine Bahnplanung programmatisch zu realisieren, werden Modelle erstellt, in denen dann Algorithmen einen Pfad finden können. Die Modelle werden entweder statisch hinterlegt oder dynamisch zur Laufzeit berechnet. Je nach Einsatzgebiet des Roboters empfiehlt sich die eine oder andere Methode. Auch vom Einsatzgebiet und der Aufgabenstellung des Roboters ist abhängig, ob ein 2D- oder 3D-Modell erstellt werden muss.

Eine mögliche 2D-Modellierung ist das Umgebungsmodell. In einem Umgebungsmodell wird die Umgebung in ein Raster eingeteilt. Die Zellen in dem Raster können dann entweder frei, besetzt oder, bei einem groben Raster, teilbesetzt sein. Als Hilfestellung dienen oft CAD-Pläne des Raumes (oder Industriehalle, o.ä.). Der Roboter wird dann entsprechend seiner Form in dieses Modell projiziert und die Aufgabe ist es dann, den Roboter in diesem Umgebungsmodell so navigieren zu lassen, dass er keine belegten Zellen berührt, um so eine Kollisionsprüfung zu vermeiden. Im Falle einer Berührung mit einer teilbesetzten Zelle, muss eine Kollisionsprüfung vorgenommen werden.

Eine Erweiterung oder Alternative zum Umgebungsmodell ist das Konfigurationsmodell. Im Konfigurationsmodell wird der Roboter als Punkt (in der Bildverarbeitung z.B. ein Pixel) modelliert. Die Objekte im Raum werden entsprechend des Durchmessers des Roboters mit einem Rand versehen. Die Betrachtung dieses Modells erlaubt eine einfachere Bahnplanung. Der Unterschied der beiden Modelle kann analog in der Bildverarbeitung bei der Erosion und Dilation wiedergefunden werden.

3.2 Dijkstra-Algorithmus

Der Dijkstra-Algorithmus wurde 1956 vom Wissenschaftler Edsger Dijkstra entwickelt. Er dient zur Findung des kürzesten Pfades in einem gewichteten Graphen mit positiven Kantengewichtungen. Der Algorithmus ist *greedy*, was bedeutet, dass er das beste Ergebnis in jedem Schritt betrachtet, um so eine Gesamtlösung zu formen. Die Knoten werden mit einer Distanz versehen, welche

sich aus verschiedenen Schritten errechnet. Initial werden alle Knoten mit einer Maximaldistanz (*Unendlich*) versehen. Der Startknoten bekommt die Distanz *0*. Weiterhin wird initial eine Menge erstellt, in der alle Knoten (ohne den Startknoten) enthalten sind. Nun werden alle noch nicht betrachteten Knoten aus dieser Menge iterativ betrachtet. Dabei wird immer zunächst der Knoten, mit der minimalen Distanz gewählt. Bei der Betrachtung wird die Distanz neu berechnet, indem die Gewichtung der Kante mit einbezogen wird. Dieser Knoten wird dann mit der kleinsten Distanz markiert und im Algorithmus nicht weiter betrachtet. Ist man am Endknoten angelangt, so kann der Algorithmus hier terminieren, wenn nur der kürzeste Weg zwischen einem Start- und Endknoten gesucht wird.

3.3 A*-Algorithmus

Der A*-Algorithmus ist im wesentlichen eine Erweiterung und Verbesserung des Dijkstra-Algorithmus. Er wurde im Stanford Forschungsinstitut 1968 erstmalig beschrieben. Um die Laufzeit zu verbessern, benutzt der A*-Algorithmus eine heuristische Funktion. Genau wie beim Dijkstra-Algorithmus, wird aus der Menge der Knoten, welche noch nicht betrachtet wurden, ein Knoten zur Betrachtung ausgewählt. Dieser Knoten wird so ausgewählt, dass die heuristische Funktion (oft die euklidische Distanz) minimal ist. Anders formuliert ist der Dijkstra-Algorithmus eine Spezialisierung des A*-Algorithmus, wobei die heuristische Funktion stets einen festen Wert (*0*) liefert. Damit der Algorithmus optimal ist, darf die heuristische Funktion keine größere Distanz zwischen zwei Knoten liefern, als die kleinste tatsächliche Distanz zwischen ihnen.

3.4 Catmull-Rom Splines

Catmull-Rom Splines ist eine Klasse von interpolierenden Polygonzügen, die von Edwin Catmull und Raphael Rom 1974 entwickelt und beschrieben wurde. Sie werden in der Modellierung bis zur Animation verwendet. Interpolierende Polygonzüge berühren dessen Stützpunkte, was zur Folge hat, dass bei der Benutzung dieser Splines eine Kontrolle über einige Punkte des Splines erlaubt. Ebenfalls hat der Catmull-Rom Spline lokale Eigenschaften, so dass bei der Änderung an bestimmten Kontrollpunkten nur ein bestimmtes Teilsegment der Kurve beeinträchtigt wird. Catmull-Rom Splines sind weiterhin glatt, was für die Animation eine relevante Eigenschaft ist. In der Klasse der Splines unterscheidet man zwischen verschiedenen Parametrisierungen, so dass die Stützpunkte in verschiedener Weise durchlaufen werden.

Der Spline ist kubisch, was bedeutet, dass 4 Berechnungspunkte für eine Kurve (bzw. ein Kurvensegment) benötigt werden, welche durch 2 Stützpunkte verläuft. Die zusätzlich benötigten 2 Stützpunkte enthalten den vorherigen Punkt und den nächsten Punkt der Kurve. Da es im letzten Kurvensegment keinen nächsten Punkt bzw. im ersten Kurvensegment keinen vorherigen Punkt gibt, müssen diese beiden Segmente separat betrachtet werden. Die Punkte der Verbindungslinie können mit linearer Algebra berechnet werden.

4 Konzept

Im folgenden Kapitel wird das Konzept des Programms näher erläutert. Zunächst wird die Architektur und die Komponenten des Programms erklärt. Für jede Komponente folgt eine zusätzliche Beschreibung, in der feiner erklärt wird, wie die entwickelten Teillösungen die Lösung der Aufgabenstellung unterstützen.

Das Programm besitzt im wesentlichen 3 Teilsysteme: Graphische Bedienoberfläche, Algorithmen und Datenverarbeitung, sowie ein Teilsystem, welches die Szene in der sich der virtuelle Roboter bewegt, visualisiert. Sofern möglich, wurde versucht, die einzelnen Systeme modular in Programmcode zu entwickeln.

4.1 Graphische Bedienoberfläche

Das GUI wird mit Qt entwickelt. Die Oberfläche besteht aus einer 2-teiligen Menüführung (wie üblich unter der Titelzeile) und einem 2-spaltigen Layout als zentrale Komponente. Die Menüführung dient zum einen dazu, das Programm zu beenden und zum anderen dazu, den aktuellen Stand des Programms in eine Datei zu speichern, so dass dieser zu einem späteren Zeitpunkt nachgeladen werden kann. Ein weiterer Menüpunkt wird bereitgestellt, um ein Bild eines Raumes zu laden, worin der Roboter seine Bahnplanung durchführen soll.

Wurde ein Bild eines Raumes erfolgreich geladen, so wird in der rechten Spalte der Oberfläche eine Szene angezeigt, in der sich verschiedene Elemente befinden. In der linken Spalte befinden sich verschiedene Auswahlkästchen, mit denen die Szene beeinflusst werden kann. Um die Elemente in der Szene besser sichtbar zu mache, werden diese unterschiedlich eingefärbt.

4.2 Visualisierung

Für die Visualisierung wurde das OpenGL-Framework benutzt, welches es ermöglicht sehr detaillierte Szenen zu konstruieren. Das Framework war für das Projekt nicht zwingend erforderlich, wurde aber dennoch gewählt, da es bereits in vorherigen Lehrveranstaltungen, die der Projektdurchführende besucht hat, verwendet wurde und somit eine gewisse Erfahrung vorhanden war. Als Alternative hätten ebenfalls Komponenten des GUI (z.B. QGraphicsScene o.ä.) verwendet werden können.

Das Framework bietet mit der Verarbeitung von Texturen und dem Zeichnen von Punkten und Linien, 2 wesentliche Komponenten, die zur Bearbeitung des Projekts benötigt wurden. Entsprechend der Einstellungen aus der graphischen Bedienoberfläche, werden in der Szene Aktionen ausgeführt. Dadurch muss das Modul der Visualisierung eine Schnittstelle anbieten, die von der Bedienoberfläche benutzt werden kann. Die Schnittstelle umfasst das Setzen von Optionen (entsprechend jener aus der Bedienoberfläche), das Abfangen von Mausklicks, das Laden von einer Raum-Bilddatei, sowie die grundlegende Schnittstelle zum Initialisieren der Graphikszene und dem Zeichnen der aktuellen Szene.

4.3 Algorithmen zur Bahnplanung

Die Algorithmen, welche zur Berechnung der notwendigen Punkte gebraucht werden, befinden sich in extra Modulen. Sie werden bei der Visualisierung einmalig verwendet und die Ergebnisse im Programmspeicher hinterlegt. Dies hat

zur Folge, dass der Speicherverbrauch etwas wächst, jedoch die Performance stark positiv beeinflusst, da die Berechnungen sonst bei jedem Neuzeichnen der Szene (die Visualisierungskomponente zeichnet die Szene stets neu) durchgeführt werden müssten. Für die Module existieren 2 große Teilmengen von Algorithmen: Algorithmen zur Planung eines Pfades durch verschiedene Punkte und Algorithmen zur Verarbeitung einer Bilddatei.

5 Umsetzung

Im folgenden Kapitel wird die konkrete Umsetzung basierend auf dem vorher entwickelten Konzept 4 detailliert beschrieben. Dabei wird zentral betrachtet, wie eine Problemstellung versucht wurde zu lösen, insbesondere welche Lösungsansätze es gibt und warum sich für eine bestimmte Lösung entschieden wurde. Weiterhin werden Algorithmen, welche für die Ausführung benötigt wurden und im Grundlagen-Kapitel 3 nicht erklärt wurden, sofern als sinnvoll erachtet, beschrieben.

5.1 Bildverarbeitung

Für die Verarbeitung des Raumes wird nicht jeder Pixel des Bildes benötigt. Vielmehr wird das Bild so analysiert, dass die Rahmen der darin befindlichen Objekte, sowie der Rahmen des Raumes und der darin befindlichen Türen separat gespeichert werden. Damit die Algorithmen zur Analyse und Verarbeitung korrekt funktionieren, müssen die Rahmen und Wände jeweils geschlossen sein und die Tür rechteckig im Bild vorhanden sein. Eine Beispieldatei liegt dem Programmcode bei. Es folgt eine Erklärung der einzelnen Schritte der Bildverarbeitung.

5.1.1 Bildeigenschaften

Damit das Programm ein Bild überhaupt laden kann, muss es gewisse Eigenschaften aufzeigen. Es werden nur Bilder vom Typ *Portable Network Graphics* (kurz: *PNG*) akzeptiert. Diese dürfen nur die beiden Farbmodelle RGB oder RGB mit Alpha-Komponente enthalten. Eine solche Bilddatei wird mittels der Bibliothek *DevIL* in den Programmspeicher geladen.

Um eine Verarbeitung des Bildes im Kontext eines Raumplans zu ermöglichen, müssen die Wände und Rahmen von Hindernissen bzw. Objekten im Raum schwarz sein (RGB-Komponenten jeweils auf Minimum). Eine weiße Farbe (RGB-Komponenten auf Maximum) bedeutet, dass sich dieser Pixel des Bildes außerhalb des Raumes befindet. Die Farbe Grau (RGB-Komponenten auf 200 bei einer Skala von 0-255) bedeutet, dass dieser Pixel Teil einer Tür des Raumes ist. Dies ist im weiteren Verlauf notwendig, damit ohne weitere Algorithmen erkannt wird, wo sich eine Tür im Raum befindet, damit genau in der Mitte einer Tür ein Wegpunkt gesetzt werden kann. Jegliche andere Farbe bedeutet, dass sich dieser Pixel im Raum befindet. Somit ist es z.B. möglich den Boden des Raumes einzufärben.

5.1.2 Rahmenpolygone des Raumes

Die Schnittstelle zum Modul der Bildverarbeitung bietet nun eine Funktion, um das Rahmenpolygon des Raumes, sowie der darin befindlichen Objekte und der Türen zu erhalten. Im 1. Schritt wird eine Abbildung erzeugt, welche für jede Koordinate im Bild speichert, welche Position sie im Raum entspricht (Im Raum, Außerhalb des Raumes, Tür oder Wand). Dann wird für jeden Pixel, der zu einer Tür oder sich im Inneren des Raumes befindet geprüft, ob sich in der nähe eine Wand befindet. Dazu wird geprüft, ob sich innerhalb des Radius des Roboters eine Wand befindet, falls ja, wird dieser Pixel aus der Menge der zu betrachtenden Knoten entfernt. Der letzte Schritt umfasst das Expandieren der noch zu betrachtenden Pixel.

Jeder Pixel, der noch zu einer Tür oder sich im Raum befindet und nicht mit einer Wand kollidiert, falls sich an dieser Raumposition der Roboter befindet, wird nun expandiert. Das Expandieren bedeutet, dass von der betrachtenden Position in eine Richtung gewandert wird und geprüft wird, ob sich dort ein Nachbarpixel befindet der ebenfalls den gleichen Typ repräsentiert (Tür oder Innenseite des Raumes). Falls nicht wird die Richtung geändert, in der die Pixel betrachtet werden. Genau dieser Pixel, an dem die Richtung geändert wird, ist somit ein Eckpunkt eines Polygons. Befindet man sich Ende der betrachteten Pixel, so ist das Polygon geschlossen. Das Resultat dieser Schritte ist eine Liste von Polygon, die man als Begrenzungen (*Constraints*) des Raumes auffassen kann.

Diese Liste ist hilfreich um zu prüfen, ob Wegpunkte innerhalb des Raumes sind oder um sie dort statistisch zu verteilen. Weiterhin werden die Kanten der Raumpolygone benötigt, um zu prüfen, ob der berechnete Pfad mit einem Objekt im Raum oder einer Wand kollidiert.

5.2 Bedienoberfläche

Das GUI wurde mit dem Framework Qt in der Vorabversion 5.4 entwickelt. Zunächst wurde mit der stabilen Version 4.8 programmiert, was sich später jedoch als nicht weiter durchführbar herausstellte, da für das Visualisieren des Roboters eine Komponente (*QOpenGLWidget*) benötigt wurde, welche erst in der Vorarbversion verfügbar war.

Wurde ein Bild über die Menüoption *Project -> Load Room* geladen, welches die geforderten Bildeigenschaften 5.1.1 erfüllt, so wird die linke Spalte etwas schmaler und auf der rechten Seite erscheint das Bild des Raumes. Für das geladene Bild wird zufällig ein Start- und Zielpunkt für den Roboter gewählt, welcher sich im Raum befindet. Die generierten Start- und Zielpunkte werden entsprechend grün und rot in der Szene als runder Kreis gemalt.

Auf der linken Seite befinden sich Bedienelemente für die Szene:

- Optionen zur Modifizierung von Wegpunkten
- Auswahlkästchen zur Anzeige von Überprüfungshilfen
- Knöpfe zum Animieren des Roboters und Anzeigen von Statistiken zur Programmlaufzeit bei verschiedenen Berechnungne
- Aufklappmenü zur Wahl des Algorithmus zum Berechnen des Pfades

- Ein Textkästchen für Statusnachrichten
- Ein Textkästchen für Hilfestellungen

Derzeit steht an Algorithmen zur Pfadberechnung der Dijkstra- und A*-Algorithmus (siehe Grundlagenkapitel 3) zur Auswahl. Das Textkästchen für Hilfestellungen gibt eine Kurzhilfe, wenn die Maus über eines der Elemente der linken Spalte schwebt. Das Textkästchen für Statusnachrichten wird zur Programmlaufzeit Text enthalten, welcher insbesondere Fehler beschreibt. Die Statusnachrichten bleiben bis zur nächsten Statusnachricht enthalten, Hilfestellungen verschwinden beim Verlassen der entsprechenden Schwebefläche mit der Maus.

5.2.1 Modifizieren von Wegpunkten

Es ist möglich Wegpunkte interaktiv im Raum zu verteilen. Dazu muss entsprechend in der linken Spalte eine entsprechende Wegpunkt-Option ausgewählt werden und im Raum mit der linken Maustaste geklickt werden. Das Programm fängt den Mausklick ab und überprüft, welches Auswahlkästchen gerade gesetzt ist und führt die entsprechende Operation durch. Sollten sich Wegpunkte überlappen bzw. wird versucht Wegpunkte zu löschen, die nicht vorhanden sind, wird eine Fehlermeldung als Statusnachricht ausgegeben. Startpunkt und Endpunkt können ebenfalls über diese Methode gesetzt werden. Gleich zu Beginn, also nach dem Laden des Bildes, werden Wegpunkte in der Mitte von Türen erzeugt. Die aktuelle Anzahl von Wegpunkten befindet sich rechts neben der Kennzeichnung *Amount of nodes*.

Dieses Kästchen hat eine weitere Eigenschaft. Wird dort eine Zahl eingegeben, so werden statistisch Wegpunkte im Raum verteilt. Dieses Vorgehen löscht alle vorher gesetzten Wegpunkte. Wegpunkte in der Mitte der Türen bleiben erhalten, somit kann die Zahl die in dem Feld erscheint höher sein, als jene die eingegeben wurde. Sollten die Wegpunkte der Türen nicht gewünscht sein, müssen sie interaktiv gelöscht werden.

5.2.2 Einstellungen und Überprüfungen der Szene

Unter den Modifizierungen befinden sich Optionen, die die Ausgabe der Szene beeinflussen. Die erste Option *Show triangulation* zeigt eine Triangulierung der Wegpunkte, inklusive des Start- und Endpunkts. Diese wird in einer hellblauen Farbe in der Szene angezeigt. Die Option *Show room triangulation* sorgt dafür, dass die komplette Triangulierung der Rahmenpolygone, die zuvor ermittelt wurden, in einer etwas helleren blauen Farbe in der Szene erscheint. Je nach Komplexität des Raumes kann dies das Zeichnen von vielen Linien nach sich ziehen, wodurch diese Option ausschließlich zu Testzwecken genutzt werden sollte. Möchte man Wegpunkte oder den generierten Pfad angezeigt bekommen, so setzt man einen Haken in die entsprechende Optionen *Show waypoints* oder *Show path*. Die Wegpunkte werden als gelbe Punkte in der gleichen Größe wie Start- und Zielpunkt gezeichnet. Eine letzte Option *Show neighbours* dient dazu, sich die Nachbarn von Wegpunkten anzeigen zu lassen und zusätzlich darstellen zu lassen, ob eine direkte Verbindung zum Nachbarn mit einem Raumobjekt oder einer Wand kollidiert. Um diese Funktion nutzen zu können, müssen die Auswahlkästchen in den Modifizierungen (oben) alle leer sein. Ein Mausklick

auf einen bereits hinzugefügten Wegpunkt hat nun zur Folge, dass Linien zu den direkten Nachbarn gezeichnet werden. Grüne Linien bedeuten, dass diese Nachbarn nicht mit Objekten kollidieren. Rote Linien verlaufen durch Objekte oder Wände. Unter den Kästchen befinden sich 2 Knöpfe, die zum einem den Roboter mit einem lila Kreis vom Start- bis zum Endpunkt fahren lässt, falls ein Pfad existiert (*Animate*) und eine Statistik ausgibt, worin die Programmlaufzeiten für verschiedene Berechnungen hinterlegt sind (*Statistics*). Unter den Knöpfen befindet sich ein Aufklappmenü, in dem festgelegt werden kann, welcher Algorithmus genutzt wird, um den Graphen für den Pfad zu planen.

5.3 Szene

Die Szene dient zur Veranschaulichung und zum Testen der entwickelten Lösung. In der Szene wird das Bild des Raumes als Hintergrund (Textur) hinterlegt, worauf dann verschiedene Zeichnungen durchgeführt werden. Dargestellt wird immer mindestens ein Start- und Endpunkt. Zusätzliche Einblendungen können über die Bedienoberfläche 5.2 vorgenommen werden.

Die Szene wird durch die Verwendung von *QOpenGLWidget* in die Qt-Oberfläche eingebettet. Diese Komponente bietet die Möglichkeit durch Ableitung der Klasse und wenigen Zeilen Code einen OpenGL-Kontext, der zur Verwendung des Frameworks benötigt wird, zu nutzen. Zusätzlich zum Code zur Initialisierung und Größenänderung der Szene, wird ein Timer eingesetzt, der dafür sorgt, dass sobald das Ereignissystem von Qt bereit ist, die Szene neu zeichnet.

Im Programmcode der Visualisierung werden bei der Initialisierung *Vertex Arrays* gebaut, die benötigt werden, um schnell einfache primitive Objekte (hier: Kreise und Kreuze) zu zeichnen. Direkt nach dem Initialisieren stehen die OpenGL-Befehle breit, somit wird dann vom geladenen Bild eine Textur erzeugt und eingebettet. Funktional wäre das Programm auch korrekt, wenn die Textur nicht angezeigt werden würde, allerdings hätte dies zur Folge, dass man nicht nachsehen kann, wo sich der Roboter oder andere Dinge im Raum befinden. Je nach Verfügbarkeit von Rechenzeit wird Qt dann versuchen die Szene zu zeichnen, also den OpenGL-Kontext erstellen und die Aufgabe an den Programmcode der Visualisierung delegieren. Dort werden dann die Primitiven, die zuvor initialisiert worden sind, je nach Einstellungen der Bedienoberfläche, gezeichnet. Wird der Roboter animiert, so wird versucht eine Bewegung in Schritten von 60 Hertz durchzuführen. Dazu wird ein Timer gestartet, der alle 17 Millisekunden den nächsten Punkt des Pfades in der Animationsfarbe darstellt. Da die Anzahl der Pfadpunkte zwischen 2 Wegpunkten immer identisch ist, scheint es, als würde sich der Roboter mal langsamer und mal schneller bewegen, je nach Entfernung der Wegpunkte zueinander. Eine Animation kann nicht abgebrochen werden.

5.4 Wegfindung

Die einzelnen Schritte der Wegfindung werden im Programm systematisch getrennt, sie umfassen die Triangulierung der Wegpunkte, das Herausfinden von direkt benachbarten Wegpunkten, das Herauslöschen von kollidierenden Kanten, das Erstellen der Pfadkurve und das Markieren von kollidieren Punkten der Pfadkurve. Eine wesentliche Säule zur Lösung dieser Teilprobleme ist die

Geometrie und die darin befindlichen Algorithmen. In diesem Abschnitt werden die einzelnen Teilprobleme genauer beschrieben.

5.4.1 Triangulierung des Raumes

Wie bereits im Kapitel Bildverarbeitung (spez. Rahmenpolygone des Raumes 5.1.2) erklärt wurde, werden die Eckpunkte des Rauminneren (inkl. der Eckpunkte der darin befindlichen Objekte) zu entsprechenden Kanten verbunden, so dass sie die Grenzen des Raumes abbilden können. Durch einfache mathematische Formeln kann dann anhand der Dreiecke überprüft werden, ob sich Punkte innerhalb oder außerhalb des Raumes befinden. Als Hilfestellung dient die *Constrained Delaunay Triangulation* (kurz: *CDT*). Sie wurde in der CGAL-Bibliothek [devc] implementiert und noch einmal im Handbuch näher beschrieben [deva]. Als Basis dient die *Delaunay Triangulation* (siehe Triangulierung der Wegpunkte 5.4.2). Nachdem die Kantenpunkte der Rahmen mit Delaunay trianguliert wurden, werden alle Dreiecke die einen Punkte auf den gewollten Kanten der Raumpolygone (*Constraints*) haben entfernt, sofern die Kanten nicht schon Teil des Dreieckes ist. Das entstandene leere Polygon zwischen den Punkten wird dann erneut einfach trianguliert (nicht mit Delaunay). Aus diesem Grund ist die *Constrained Delaunay Triangulation* unter Umständen keine vollständige *Delaunay Triangulation* mehr.

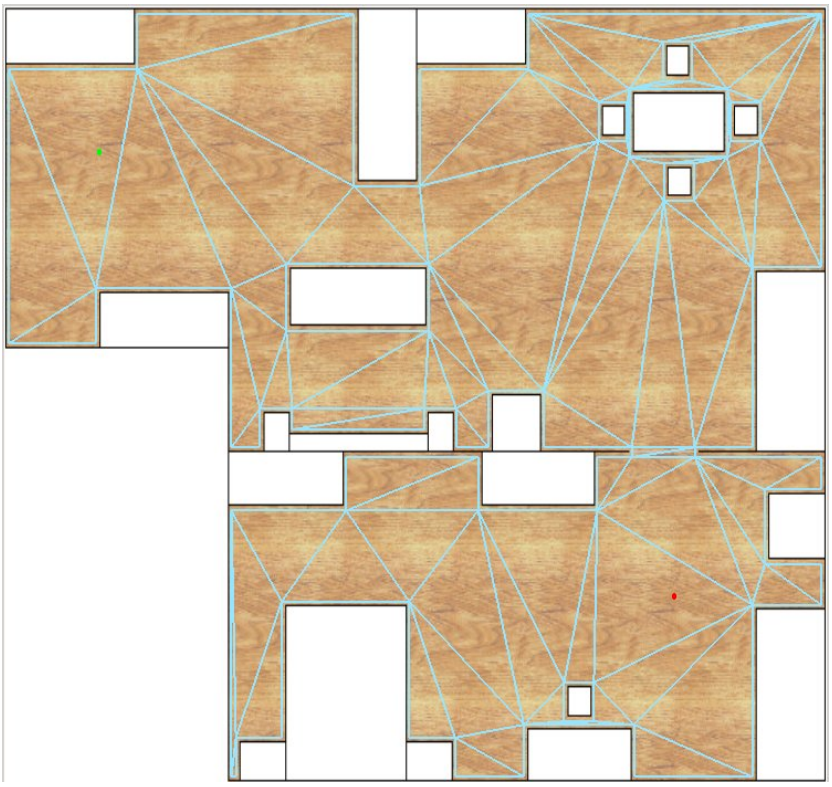

Abbildung 1: Die Triangulierung eines Raumes mit der *Constrained Delaunay Triangulation*

5.4.2 Triangulierung der Wegpunkte

Die Wegpunkte, inkl. Start- und Endpunkte werden mit der *Delaunay Triangulation* trianguliert. Eine *Delaunay Triangulation* (kurz: *DT*) versucht es die Winkel der Dreiecke möglichst groß zu machen und somit kleine Dreiecke zu vermeiden. Dadurch wirkt die Triangulierung "natürlich". Die Implementierung die im Programm benutzt wurde, stammt von der CGAL-Bibliothek [devc] (siehe [devb]). Als Alternative hat die Implementierung auch *Regular triangulations* zur Verfügung gestellt. Es wurde auf die *Delaunay Triangulation* zurückgegriffen, da sich dazu mehr Dokumentation auf Anhieb finden ließ.

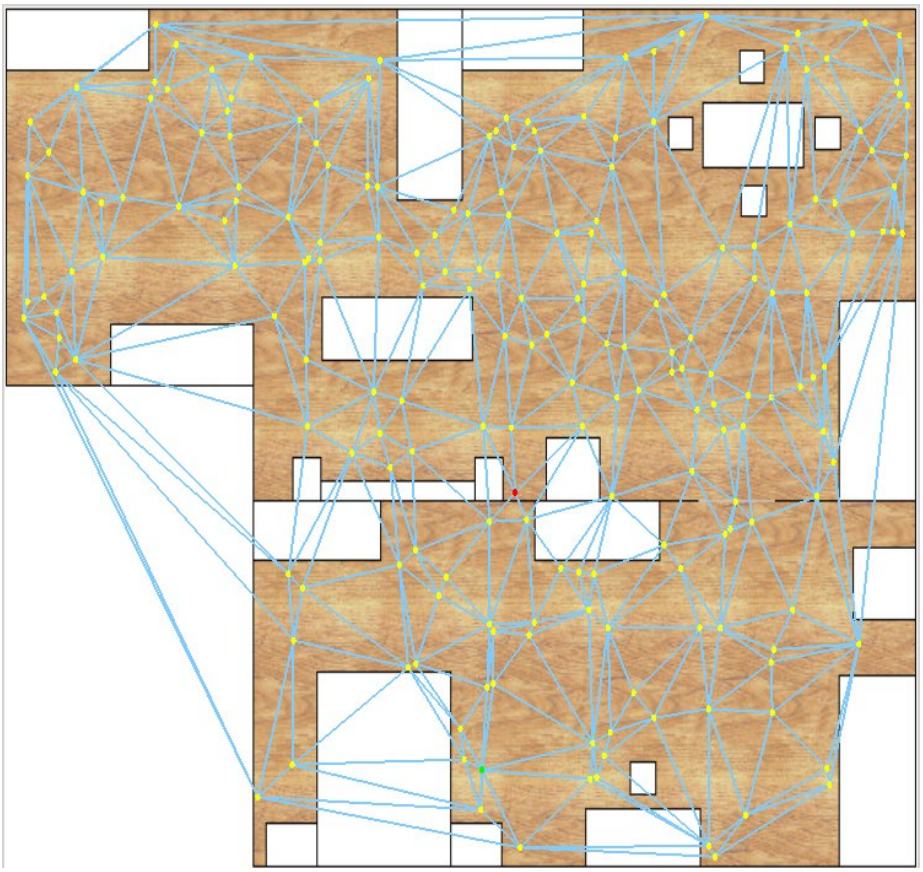

Abbildung 2: Die Triangulierung von Wegpunkten mit der *Delaunay Triangulation*

5.4.3 Kollisionsprüfung im Raum

Werden Wegpunkte oder der Start- bzw. Endpunkt über die Benuteroberfläche gesetzt, so prüft der Programmcode ob dieser innerhalb des Raumes liegen. Dazu dient die Triangulierung der Rahmenpolygone des Raumes (siehe 5.1.2). Das Problem kann darauf zurückgeführt werden, zu prüfen, ob ein Punkt in einem Polygon vorhanden ist. Wie in [Col05] beschrieben, kann dieses Problem weiter eingegrenzt werden, indem man prüft, ob ein Punkt in den Dreiecken

der Polygon-Triangulierung ist. Dazu dienen verschiedene Algorithmen, so z.B. das Ablaufen der Dreieckskanten und Prüfung, ob ein Punkt links oder rechts von der Kante liegt. Die CGAL-Bibliothek, welche die Triangulierungen unterstützt, bietet in der Schnittstelle für die Verwendung der *Constrained Delaunay Triangulation* eine Methode, welche prüft, ob ein Punkt in der Triangulierung vorhanden ist oder nicht. Diese Methode verhindert nun, dass Punkte außerhalb des Raumes gesetzt werden können.

5.4.4 Generierung des Pfades

Der Pfad wird nun mit Hilfe eines gewichteten Graphen erstellt. Der gewichtete Graph entsteht zum einen durch die Triangulierung der Wegpunkte (siehe 5.4.2) und eine zusätzliche Gewichtung der entstandenen Dreiecksseiten. Zunächst werden alle Kanten zu direkten Nachbarn, also Punkte die mit genau einer Kante vom betrachteten Knoten verbunden sind, in eine Menge eingefügt. Für jede dieser Kanten wird dann geprüft, ob sie eine Kante der Rahmenpolygone (siehe 5.1.2) schneidet. Dazu dient ein Algorithmus, der auf Vektor- und seine Kreuzprodukte basiert (siehe [Ree09]).

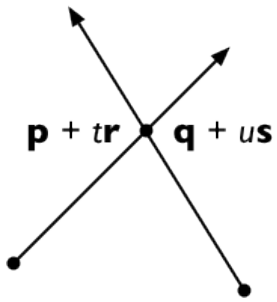

Abbildung 3: Zwei sich schneidende Strecken in Vektordarstellung

Wie in Abbildung 3 gezeigt wird, existiert ein Schnittpunkt wenn ein t und u gefunden wird, so dass $p + t \cdot r = q + u \cdot s$. Bildet man auf beiden Seiten der Gleichung das Kreuzprodukt mit dem Vektor s, erhält man mit linearer Algebra nun Gleichungen für u und t.

$$t = \frac{(q - p) \times s}{r \times s}$$
$$u = \frac{(p - q) \times r}{r \times s}$$

Durch das Auftreten des Nenners $r \times s$ ist es notwendig zu prüfen, dass dieser nie 0 ist. Ist $r \times s = 0$, so gilt:

- Die Strecken sind kollinear, falls $(q - p) \times r = 0$
- Die Strecken sind parallel, sonst

Ist $r \times s \neq 0$, so gilt:

- Die Strecken treffen sich bei $p + t \cdot r = q + u \cdot s$, wenn $0 \leq t \leq 1$ und $0 \leq u \leq 1$ sind

- Die Strecken sind weder parallel, noch treffen sie sich, sonst

Die Schnittpunktprüfung findet nun für jede Kante der Rahmenpolygone statt. Gibt es 2 oder mehrere Schnittpunkte einer Kante mit einem Rahmenpolygon, so wird für jeden Schnittpunkt eine neue Kante zum nächsten Schnittpunkt erzeugt. Beispiel: $Edge_0 = \overline{P0P1}$, $Edge_2 = \overline{P1P2}$, $Edge_3 = \overline{P2P0}$ Für jede dieser Kanten wird dann der Mittelpunkt berechnet und mit Hilfe der Kollisionsprüfung im Raum (siehe 5.4.3) wird festgestellt, ob der Mittelpunkt im Raum liegt. Damit wird verhindert, dass *false positives* auftreten, also Kanten die z.B. auf 2 Rahmen von Objekten liegen, jedoch sonst vollständig im Raum sind. Gibt es nur einen Schnittpunkt, so wird geprüft, ob dieser auf einem Rahmen liegt (Wand oder Objekt), ist dies der Fall, ist die Kante vollständig im Raum, da Rahmen von Objekten grundsätzlich als Ïnnerhalb des Raumesängesehen werden.

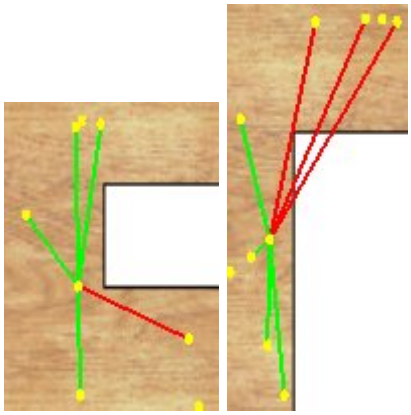

Abbildung 4: Kollisionsprüfung von Kanten zu direkten Nachbarn (rote Kanten kollidieren)

Als nächster Schritt wird eine Abbildung von jedem Wegpunkt zu einer Menge von erreichbaren Nachbarknoten angelegt. Diese Kanten werden dann mit dem euklidischen Abstand gewichtet und man erhält so als Folge einen gerichteten Graphen. Dieser Graph dient als Grundlage zur Berechnung des Pfades basierend auf Wegpunkte mit Dijkstra und A* (beschrieben in 3.2 und 3.3). Für den Programmcode wurden eigene Implementierungen von Dijkstra und A* entwickelt. Dabei teilen sich die beiden Algorithmen den Basiscode, da Dijkstra nur ein Spezialfall des A*-Algorithmus ist [lei09], in der die heuristische Funktion jede Eingabe auf 0 abbildet. Für die Implementierung diente der Pseudocode im Wikipedia-Artikel [Com].

Nachdem die Reihenfolge der Wegpunkte durch den Algorithmus berechnet wurde, wird der Catmull-Rom Spline zwischen den einzelnen Punkten gebildet (siehe 3.4), um eine Kurve zur schönen Darstellung zu formen. Da das 1. und letzte Kurvenstück keinen vorherigen bzw. nächsten Stützpunkt besitzt, wird dort einfach der 1. bzw. letzte Punkt dupliziert.

Falls Kanten zwischen Wegpunkten nicht mit Objekten oder Wänden kollidieren, kann es trotzdem vorkommen, dass die generierte Kurve, also der eigentliche Bewegungslauf des Roboters, mit Gegenständen oder Wanden kollidiert. Deshalb wird geprüft, ob jeder Teilpunkt eines Kurvensegments zwischen

2 Stützpunkten (die Anzahl der Teilpunkte hängt von der festgelegten Granularität im Quellcode ab) im Raum liegt.

6 Durchführung des Anwendungsfalls

Bei der Abgabe des Dokuments an den Dozenten, wurde eine CD mitgegeben, auf der das Programm ausführbar für die Betriebssysteme Windows 7 und Ubuntu 14.04.1 vorliegt. Nach dem Start präsentiert sich die Benutzeroberfläche. Durch Klicken auf *Project -> Load Room* kann eine Bilddatei geladen werden. Auf der CD liegt eine Datei namens *Room.png* bei, die geladen werden kann. Nun können statistisch Wegpunkte verteilt werden, indem man im Feld *Amount of nodes* eine Zahl eingibt und bestätigt. Alternativ können die Wegpunkte hinzugefügt werden, indem man *Add waypoint* markiert und mit der Maus auf die gewünschte Stelle klickt. Die gleiche Vorgehensweise kann angewandt werden, um Wegpunkte zu löschen oder Start- und Endpunkt zu setzen. Zu jedem Zeitpunkt kann man *Show triangulation* anklicken, um sich die Triangulierung der Wegpunkte anzusehen. Klickt man auf *Show path*, so wird ein Pfad angezeigt, falls dieser verfügbar ist. Möchte man z.B. prüfen, ob die benachbarten Knoten von einem Punkt erreicht werden können, entfernt man die Häkchen bei den Wegpunkt-Modifizierungen und setzt ein Häkchen bei *Show neighbours*. Dann klickt man mit der linken Maus auf den zu betrachtenden Wegpunkt. Möchte man sehen wie sich der Roboter bewegt, so klickt man auf *Animate*. Um einen anderen Wegplanungs-Algorithmus zu wählen, steht ein Dropdown-Menü unter dem Knopf zur Verfügung. Eine Gesamtstatistik kann man sich mit einem Klick auf *Statistics* anzeigen lassen.

7 Probleme und Ausblick

Aufgrund der vielen mathematischen Operationen besteht viel Spielraum für Performance-Verbesserung in allen Algorithmen. Insbesondere die Prüfung, ob sich ein Punkt im Raum befindet, ob eine Kante ein Objekt schneidet und ob der Pfad ein Objekt schneidet bedarf mehr mathematische Kenntnisse, um diese zu optimieren. Durch Optimierung fallen möglicherweise verwendete Konzepte und Algorithmen weg. Eine Verbesserung hinsichtlich der Szene kann erreicht werden, indem eine höhere Auflösung für die Textur bzw. des Bildes gewählt wird. Dann könnten die Kreise des Roboters und der Wegpunkte einen größeren Durchmesser bekommen und wären so besser zu erkennen. Der Durchmesser des Roboters ist derzeit nicht dynamisch anpassbar sondern muss im Quelltext bearbeitet werden, was zur Folge hat, dass das Programm neu kompiliert werden muss. Es wurde mit einem Speicher-Debugging-Werkzeug (*Valgrind*) geprüft und durch Verbesserungen im Quelltext verhindert, dass Probleme mit der Behandlung von dynamischen Speicher vorliegen. Bei einer Weiterentwicklung sollte jedoch unbedingt Fehler geprüft werden, wenn unerwartete Benutzereingaben getätigt werden oder unerwartete Daten anfallen (z.B. im Bild). Aufgrund der Aufgabenstellung wurden die Wegpunkte statistisch verteilt, jedoch ist einer zukünftigen Version denkbar, dass andere Verfahren eingesetzt werden, um Wegpunkte zu verteilen (siehe Umsetzungskapitel spez. Wegfindung 5.4).

8 Persönliches Fazit

Bei der Bearbeitung dieses Projekts habe ich erkannt, dass es viele mathematische Teilprobleme zu lösen gilt, um eine Wegfindung durchzuführen. Dabei habe ich vor allem festgestellt, dass Lösungen nie allgemeingültig sind, sondern sehr stark vom Anwendungsfall abhängig sind. So z.B. sind die Algorithmen in diesem Projekt nicht auf geringen Speicherverbrauch bzw. Quelltext optimiert, so dass sie evtl. nicht in eingebetteten System Anwendungen finden können. Ein mathematisches Verständnis in Teilen der Geometrie ist durchaus notwendig, um Algorithmen zu verstehen und entsprechend anzuwenden, eine allgemeine Schnittstelle, um alle Berechnungen durchzuführen gibt es nicht. Ebenfalls war ich in der Lage meine Kenntnisse bezüglich der GUI-Programmierung, insbesondere dem Einbetten von OpenGL-Szenen in solchen.

Bezüglich der Lehrveranstaltung kann ich festhalten, dass ich eine Bahnplanung in einem 2D-Raum durchführen kann und verstanden habe, welche Lösungsansätze es gibt und warum sich manche etabliert haben. Ebenfalls habe ich das Prinzip der Splines verinnerlicht und verstehe, dass eine korrekte kollisionsfreie Bahnplanung immer noch zur Folge haben kann, dass ein Roboter bei entsprechendem Bewegungsradius kollidieren kann.

9 Quellen

9.1 Literaturverzeichnis

[Arm06] Jim Armstrong. Catmull-rom splines, 2006.

[Col05] Ryan Coleman. Point inclusion and processing simple polygons into monotone regions, 2005.

[Com] Community. Wikipedia: A* search algorithm.

[CY] John Keyser Cem Yuksel, Scott Schaefer. On the parameterization of catmull-rom curves.

[deva] CGAL developers. Cgal 4.5 - 2d triangulation: Constrained delaunay triangulations.

[devb] CGAL developers. Cgal 4.5 - 2d triangulation: Delaunay triangulations.

[devc] CGAL developers. Computational geometry algorithms library.

[Joy02] Kenneth I. Joy. On-line geometric modelling notes: Catmull-rom splines, 2002.

[lei09] leiz. Stackoverflow: How does dijkstra's algorithm and a-star compare?, 2009.

[Pet] Samuel Peterson. Computing constrained delaunay triangulations.

[Ree09] Gareth Rees. Stackoverflow: Intersection of two line segments, 2009.

[Twi03] Christopher Twigg. Catmull-rom splines, 2003.

9.2 Verwendete Software und Frameworks

- Qt (`http://qt-project.org`)
- OpenGL, GLU (`http://www.opengl.org`)
- DevIL, ILU (`http://openil.sourceforge.net`)
- CGAL (`http://cgal.org`)
- GLEW (`http://glew.sourceforge.net`)
- Boost (`http://www.boost.org`)
- GMP (`http://gmplib.org`)
- MinGW (`http://www.mingw.org`)
- GCC (`http://gcc.gnu.org`)