Bachelor-Report

Erfassung einer dreidimensionalen Umgebung mit zweidimensional messenden Laserscannern

Manfred Eppe

25. September 2008

Gutachter und Betreuer:
Prof. Dr. Bernd Krieg-Brückner, Dr. Bernd Gersdorf

Universität Bremen – Fachbereich 3
Mathematik und Informatik

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich diese Arbeit selbstständig durchgeführt und keine außer den angegebenen Hilfsmitteln und Quellen verwendet habe.

Bremen, den 25. September 2008

Inhaltsverzeichnis

1 Einleitung **1**

2 Vorher-Zustand: Navigation auf Basis der zweidimensionalen Wahrnehmung durch horizontal angebrachte Laserscanner **3**

2.1 Einlesen der Scanpunkte in der Klasse `LocalRobot` 3

2.2 Umrechnen der Distanzen in 2D-Weltkoordinaten mit der Klasse `ScanPointsProvider` 3

2.2.1 Berechnung der Punkte in Weltkoodinaten 3

2.2.2 Verschiedene Arten von Hindernispunkten 5

2.3 Mapping der Punkte in Weltkoordinaten auf das `EvidenceGrid` in der Klasse `PolygonLocalMapper` 5

2.3.1 Erzeugung des `EvidenceGrid` 5

3 Erweiterungen und Modifikationen als Grundlage für die 3D-Wahrnehmung **7**

3.1 Modifikation der roboterspezifischen Daten 7

3.2 Modifikationen an den RoSiML Szenario-Dateien 7

3.3 Modifikationen bei der Erzeugung der `LaserScanCollection` in der Simulation 8

3.4 Modifikation in der Modulkonfiguration des Szenarios 8

3.5 Die Klasse `Pose3D` 8

3.5.1 Konkatenation der `Pose3D` („+“-Operator) 9

3.6 Die Klassen `Matrix` und `RotationMatrix` 9

4 Erstellung einer zweidimensionalen Umgebungskarte (`EvidenceGrid`) aus 3D Daten **11**

4.1 Einlesen der Scanpunkte in der Klasse `LocalRobot` 11

4.2 Umrechnen der Distanzen in 3d-Weltkoordinaten mit der Klasse `ScanPointsProvider3D` 11

4.2.1 Berechnung der Punkte in Weltkoodinaten 11

4.2.2 Der Boden als Hindernis . . . 11
4.2.3 Verschiedene Arten von Hindernispunkten . . . 13
4.2.4 Abgrunderkennung . . . 13
4.2.5 Fehlmessungen . . . 13
4.3 Abbildung der Punkte in Weltkoordinaten auf das `EvidenceGrid3D` in der Klasse `PolygonLocalMapper25D` . . . 15
4.3.1 Erzeugung des `EvidenceGrid3D` . . . 15

5 Ergebnis und weiterführende Überlegungen 21
5.1 Funktionalität . . . 21
5.2 Performance . . . 23
5.3 Der bewegliche Scanner . . . 23
5.4 Hintergründige Überlegungen zu unsicherem Wissen . . . 24

A Anhang 25
A.1 Wichtige Klassen und Methoden . . . 25
A.1.1 `Pose2D`, `Pose3D` . . . 25
A.1.2 `ScanPoints::Point`, `ScanPoints3D::Point3D` . . . 25
A.1.3 `ScanPoints`, `ScanPoints3D` . . . 26
A.1.4 `ScanPointsProvider`, `ScanPointsProvider3D` . . . 26
A.1.5 `EvidenceGrid`, `EvidenceGrid3D` . . . 27
A.1.6 `PolygonLocalMapper`, `PolygonLocalMapper25D` . . . 27
A.2 Auszüge aus dem Quelltext . . . 28
A.2.1 Algorithmus zur Beseitigung der Fehlmessungen . . . 28
A.3 Konfigurationsdateien . . . 29
A.3.1 Einstellungen am Roboter: `settings.cfg` . . . 29
A.3.2 RoSiML-Repräsentation des Hokuyo-Laserscanners . . . 29
A.3.3 Modulkonfiguration . . . 29

Abbildungsverzeichnis 31

Literaturverzeichnis 31

1 Einleitung

In dieser Arbeit wird erörtert, wie gut man mit Hilfe von Laserscannern die jeweils nur eine Ebene (2D) vermessen können, Hindernisse und Umgebungen in drei Dimensionen erfassen kann. Dies geschieht unter Einsatz des Rollstuhls des Projektes Rolland@Home der Universität Bremen [ROLLAND]. Das Projekt hat sich unter anderem zum Ziel gesetzt, diesen Rollstuhl weitgehend autonom fahren lassen zu können. Hierfür muss er natürlich Hindernisse erkennen, um ihnen auszuweichen. Zur Zeit geschieht dies mit zwei horizontal ausgerichteten Laserscannern vom Typ Siemens LS4, welche vorne und hinten am Rollstuhl in einer Höhe von 12 cm angebracht sind. Damit wird eine 2D-Repräsentation der Umgebung erzeugt, nach der der Rollstuhl navigiert. Da diese Karte jedoch nur die Ebene in 12 cm Höhe repräsentiert, kann der Rollstuhl Hindernissen die höher oder tiefer sind nicht ausweichen. Von Tischen, Stühlen oder Betten z.B. werden nur die Beine erkannt; der Rollstuhl würde also unter einem Bett, zwischen den Beinen hindurch fahren wollen. (S. Abb. 1.1).

Ziel dieser Arbeit ist es, herauszufinden wie gut man mit einem dritten Laserscanner, der schräg, von oben nach vorne unten, strahlt, Hindernisse erfassen kann, die ausserhalb besagter 12-cm-Ebene liegen. Dieser Laserscanner soll im Idealfall über dem Kopf des Fahrers angebracht werden. Somit ist eine Erfassung von „Abgründen“ wie z.B. Bordsteinkanten, Treppen etc. ebenfalls möglich.

Als Framework für die Simulation wurde SimRobot [SIMROBOT] benutzt, ein von Dr. Thomas Röfer und anderen Mitarbeitern der Universität Bremen entwickeltes Tool zur Robotersimulation, welches hauptsächlich für den RoboCup vom Team der Universität Bremen geschrieben wurde. Als Entwicklungsumgebung zum Programmieren und Debuggen wurde Microsoft Visual Studio 2005 [VISUALSTUDIO] benutzt.

Dieses Dokument besteht nach dieser Einleitung im Wesentlichen aus vier Teilen: Es erläutert zunächst die Navigation nach 2D-Wahrnehmung, beschreibt im Folgenden die Modifikationen die als Grundlage für die 3D-Wahrnehmung dienen, und möchte dann

die Navigation nach 3D-Wahrnehmung erklären. Zum Schluss werden noch auftretende Probleme besprochen und ein Ausblick mit weiterführenden Überlegungen und möglichen Verbesserungen gegeben.
Im Anhang sind interessante Teile des Quelltextes sowie *qualitative* Referenzen zu den benutzten Klassen und deren wichtigsten Methoden gegeben welche für ein vollständiges Verständnis dieser Arbeit konsultiert werden sollten.

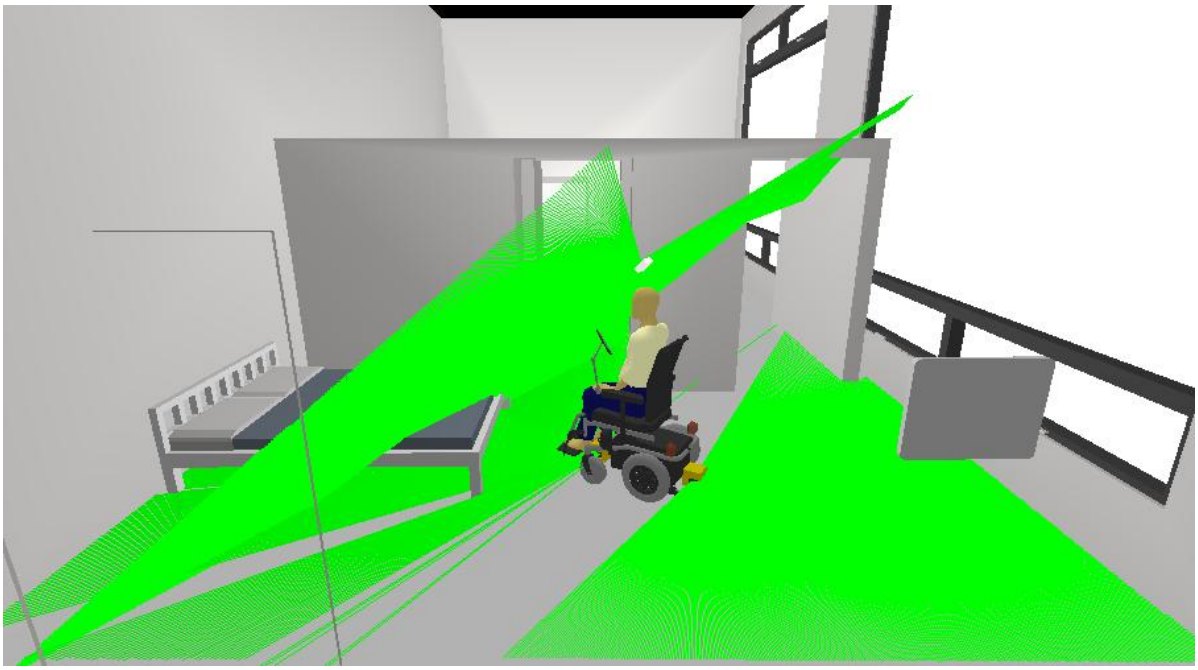

Abbildung 1.1: Rolland in der Simulation

Ohne den dritten Laserscanner über dem Kopf des Fahrers würde der Rollstuhl zwischen den Beinen des Bettes hindurch fahren wollen. Mit dem dritten Laserscanner erkennt Rolland jedoch die Bettoberfläche als Hindernis.

2 Vorher-Zustand: Navigation auf Basis der zweidimensionalen Wahrnehmung durch horizontal angebrachte Laserscanner

Um zu beschreiben wie die 2D-Karte (`EvidenceGrid`) nach der der Rollstuhl navigiert erzeugt wird, werde ich im Folgenden den Weg der aufgenommenen Scanpunkte durch den ursprünglichen Code verfolgen. Dieser Weg ist in der Modulansicht (Abb. 2.1) gut nachvollziehbar.

2.1 Einlesen der Scanpunkte in der Klasse `LocalRobot`

Diese Klasse wird nur in der Simulation verwendet. In ihrer Methode `update()` werden die Scanpunkte eingelesen indem über jeden Scanner und jede gemessene Distanz iteriert wird. Das die Messung repräsentierende Objekt `LaserScanCollection` ist vom Typ `Collection<LaserScan>`, wobei jedes Objekt `LaserScan` die Daten für jeweils einen Scanner repräsentiert. (Scannertyp und -details, Entfernungen der Scanpunkte, etc.) Die Collection beinhaltet hier also drei Objekte.
Dadurch, dass der `ModuleManager` also `LocalRobot->update()` aufruft, werden im Wesentlichen die Distanzen der Scanpunkte aktualisiert. Dies geschieht ca. 30 mal pro Sekunde.

2.2 Umrechnen der Distanzen in 2D-Weltkoordinaten mit der Klasse `ScanPointsProvider`

2.2.1 Berechnung der Punkte in Weltkoodinaten

Weltkoordinaten meint ein Kartesisches Koordinatensystem dessen Ursprung im Ort des Einschaltens des Rollstuhls liegt. (S. Anhang A.1.2, A.1.3, A.1.4.) Die Werte in den

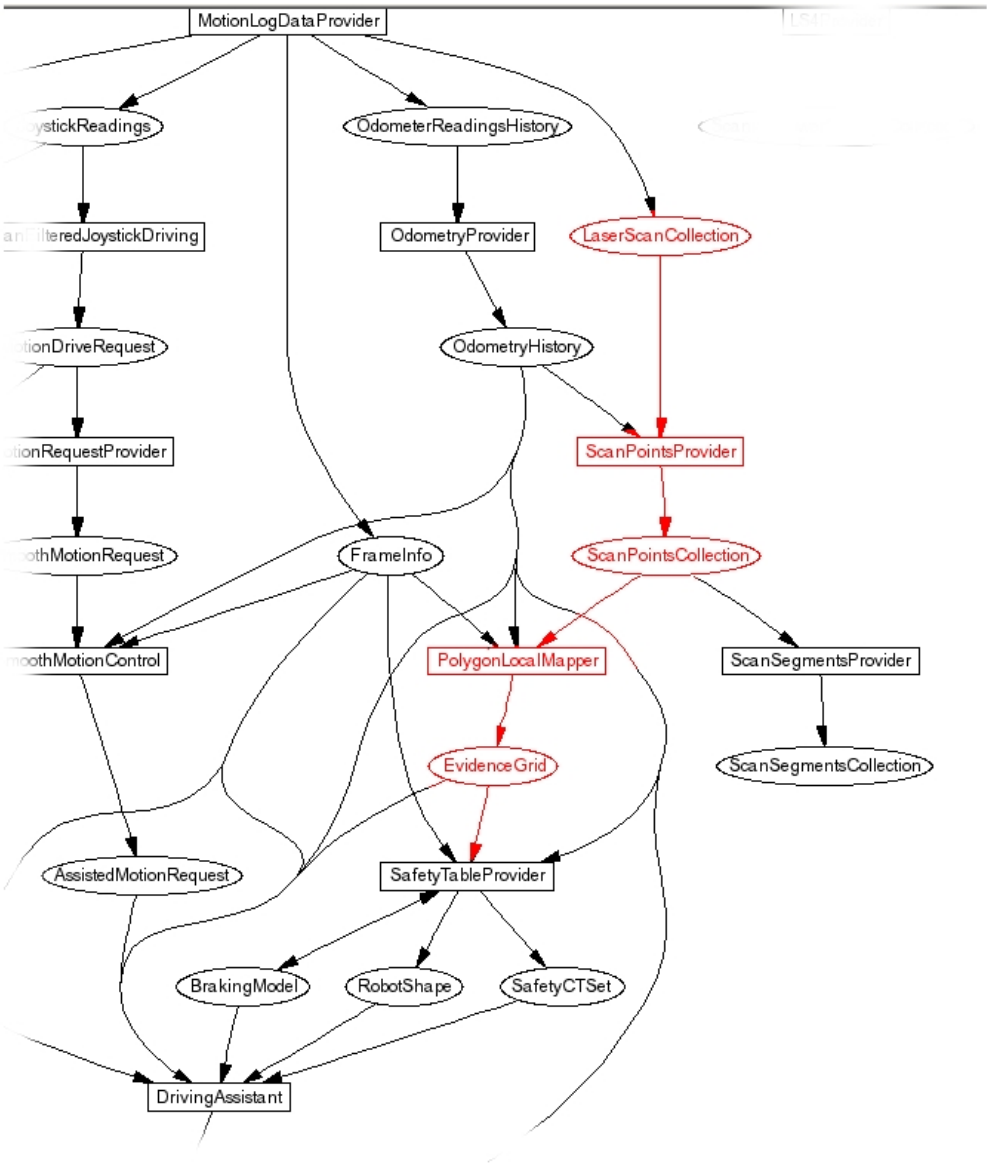

Abbildung 2.1: Modulansicht für die 2D-Navigation

Interessant für uns ist der rot illustrierte Bereich. Die `LaserScanCollection` *ist hier die Eingabe. Diese sind wiederum Grundlage für die Berechnung des* `EvidenceGrid` *welches im* `PolygonLocalMapper` *erstellt wird. Dieses besteht aus 250 x 250 Pixeln und repräsentiert die Umgebung des Rollstuhls. Auf Grundlage dieser Karte navigiert Rolland III. In meiner Arbeit habe ich für die Navigation das Modul* `DrivingAssistant` *benutzt, welches den Rollstuhl intelligent steuert.*

jeweiligen Dimensionen entsprechen Millimetern. Der `ScanPointsProvider` errechnet aus Position und Ausrichtung der Laserscanner sowie den aufgenommenen Distanzen die Position der Punkte in Weltkoordinaten. Die besagten Punkte sind Objekte vom

Typ `ScanPoints::Point` welche in der Datei ScanPoints.h deklariert wurden. Für diese Berechnung konkateniert man zunächst die mit Hilfe der Odometrie errechneten Pose des Rollstuhls mit der des Scanners (im Bezug zum Rollstuhl). Dann wird die Resultierende Pose um den Winkel des einzelnen Laserstrahls der für den Reflektionspunkt verantwortlich ist rotiert. Nun braucht man nur noch die gemessene Distanz als Pose konkatenieren und erhält als Ergebnis eine Pose dessen Translation den Koordinaten des Punktes in Weltkoordinaten entspricht.

2.2.2 Verschiedene Arten von Hindernispunkten

Jeder Punkt in diesen Weltkoordinaten hat ein bestimmtes Flag:

- `infinite`: Wenn die gemessene Distanz größer als die vom Scanner erkennbare Maximaldistanz. Diese Werte werden ignoriert, da diese hohen Distanzen auf eine Fehlmessung hinweisen.
- `isInIgnoreArea`: Diese `IgnoreAreas` wurden hinzugefügt, damit die vom Scanner erfassten Castor-Räder beispielsweise nicht als Hindernis erkannt werden.
- `nextToIgnoreArea`: Manchmal wird der Laserstrahl nur teilweise vom Castorrad reflektiert. Auch dann muss dieser Distanzwert ignoriert werden.
- `normal`: Dieser Wert ist gültig und wird für die weiterführende Navigation verwendet.

2.3 Mapping der Punkte in Weltkoordinaten auf das `EvidenceGrid` in der Klasse `PolygonLocalMapper`

In der Klasse `PolygonLocalMapper` werden die Punkte in Weltkoordinaten auf das `EvidenceGrid` übertragen nachdem der Rollstuhl navigiert. Es besteht aus 250 x 250 Pixeln und repräsentiert einer Karte der Umgebung des Rollstuhls. (S. Anhang A.1.5, A.1.6.)

2.3.1 Erzeugung des `EvidenceGrid`

Das `EvidenceGrid` wird vom `ModuleManager` immer wieder neu, mit jedem Aufruf von `PolygonLocalMapper->update()` an den `PolygonLocalMapper` übergeben. Zunächst wird die aktuelle Position des Rollstuhls in Weltkoordinaten der Odometrie entnommen. Das `center` des `EvidenceGrid` wird daraufhin auf diese Koordinaten gesetzt. (Die

Mitte der Hinterachse des Rollstuhls ist die Mitte des `EvidenceGrid`. Es rotiert jedoch nicht mit dem Rollstuhl mit.)

Daraufhin wird über alle im `ScanPointsProvider` erzeugten Punkte iteriert. Diese werden mit der Methode `EvidenceGrid::worldToGrid()` in „Gridkoordinaten“, also Pixel im `EvidenceGrid` umgerechnet. Dabei werden alle Punkte ignoriert, die nicht das im `ScanPointsProvider` gesetzte Flag `normal` haben. Die so errechneten Punkte sind jetzt vom Typ `PolygonLocalMapper::Point`. (Ähnlich, aber nicht zu verwechseln mit dem Typ `ScanPoints::Point`.)

2.3.1.1 Clipping

Jetzt werden mit der Funktion `PolygonLocalMapper::clipPointP2()` alle Punkte die außerhalb des `EvidenceGrid` liegen soweit gekürzt, dass sie am Rand des `EvidenceGrid` liegen. Man stelle sich einen Vektor vom Laserscanner zu den Koordinaten des Punktes vor, dessen Betrag soweit gekürzt wird, dass er wieder im `EvidenceGrid` liegt. Seine Richtung beleibt dabei enthalten.

2.3.1.2 Ausfüllen der Ebenen zwischen Laserscanner und Hindernis

Wenn ein Scanner einen bestimmten Punkt erfasst, kann davon ausgegangen werden, dass die Ebene zwischen Laserscanner und Punkt kein Hindernis enthält. Daher kann die Hinderniswahrscheinlichkeit der Pixel im `EvidenceGrid`, durch die der Strahl hindurch geht, verringert werden.

2.3.1.3 Einzeichnen der Hindernisse

Alle Punkte, die das Flag `normal` haben, werden nach Bresenham mit der Methode `PolygonLocalMapper::drawObstacleLines()` in das `EvidenceGrid` eingezeichnet. Das `EvidenceGrid` ist nun fertig und kann vom `DrivingAssistant` für die Navigation genutzt werden.

3 Erweiterungen und Modifikationen als Grundlage für die 3D-Wahrnehmung

3.1 Modifikation der roboterspezifischen Daten

In der Datei `Rolland/Config/Robots/Shareit/settings.cfg` können einige Eigenschaften des Rollstuhls festgelegt werden. Dies beinhaltet auch die Position und Ausrichtung der Laserscanner. Bisher gab es dort einen Abschnitt `[LS4]`, in dem die beiden Siemens-LS4-Scanner beschrieben wurden. Um die Nutzung von Laserscannern zur Umgebungserfassung zu generalisieren habe ich diesen Abschnitt durch einen Abschnitt `[laserScanners]` ersetzt, welcher zusätzlich zu Position, Ausrichtung und Kalibierierungsdaten der Scanner auch noch den Typ enthält. Hierdurch muss nicht mehr für jedes neue Laserscanner Modell eine eigene Sektion erstellt werden, was eine praktische Generalisierung darstellt. Desweiteren ist in diesem Abschnitt die Position des Scanners um die z-Koordinate sowie die Rotation um x- und y-Achse erweitert worden (S. Anhang A.3.1).

3.2 Modifikationen an den RoSiML Szenario-Dateien

SimRobot verwendet die XML-Sprache RoSiML für die Erzeugung der Simulationsumgebung [ROSIML]. Hier habe ich auf Grundlage eines Szenarios für den DrivingAssistant im BAALL ein Szenario für den Einsatz des dritten Laserscanners erzeugt. Ein weiteres Szenario für die Abgrunderkennung habe ich auf Grundlage eines Szenarios für den SafetyAssistant erzeugt. In der szenariospezifischen `modules.cfg` mussten die Module angepasst, also die 2D-Module durch die neuen 3d-Module ersetzt werden, und der neue Laserscanner vom Typ Hokuyo URG04LX brauchte ebenfalls eine RoSiML-Repräsentation. Als Grundlage hierfür diente die schon vorhandene Repräsentation des Siemens LS4. Die Repräsentationen werden in RoSiML als Makros angelegt (S. Anhang A.3.2).

3.3 Modifikationen bei der Erzeugung der `LaserScanCollection` in der Simulation

Die hier gemachten Modifikationen wurden zwar nicht direkt für die dreidimensionale Umgebungserkennung vorgenommen, verbessern jedoch die Struktur und Übersichtlichkeit des Codes.
Bisher war die Position und Ausrichtung der Laserscanner in der `LocalRobot.cpp` hart kodiert. Die Position und Ausrichtung stand zwar im Abschnitt `[LS4]` der `settings.cfg` (s.o.) wurde dort aber nur für die Messung am realen Rollstuhl in der Klasse `LS4` entnommen. In der Simulation hatten die dort angegebenen Werte keinerlei Einfluss.
Nun wird die settings.cfg im Konstruktor der Klasse LocalRobot eingelesen, und die entsprechenden Variablen für Position, Kalibrierung und Typ der Laserscanner werden beim Erzeugen einer Instanz dieser Klasse gesetzt. Dadurch ist es nun auch möglich, die Position der Scanner zur Laufzeit zu ändern, d.h. die Scanner intelligent schwenken zu können.

3.4 Modifikation in der Modulkonfiguration des Szenarios

In der Datei `modules.cfg` wird eingestellt, welche Module der `ModuleManager` laden und ausführen soll. Hier wurden die Einträge der alten 2D-Klassen auskommentiert und die Einträge über die 3D-Klassen wurden hinzugefügt. (S. Anhang A.3.3.)

3.5 Die Klasse `Pose3D`

Analog zur bisher benutzten Klasse `Pose2D` wurde eine Klasse `Pose3D` erstellt. (S. Anhang A.1.1). Diese repräsentiert Position und Ausrichtung in drei Dimensionen. Sie enthält verschiedene Operatoren, die größtenteils auf Matrizenoperationen beruhen. Die Klasse wurde vom RoboCup-Team der Universität Bremen geschrieben und mir freundlicherweise zur Verfügung gestellt. Ich modifizierte sie geringfügig.
Die Position wird in Kartesischen Koordinaten angegeben, die Rotation in Winkel um die x, y und im 3D-Fall die z-Achse.
Die für das Verständnis wichtigste Funktion dieser Klasse ist die Konkatenation („+“ - Operator), welche im Folgenden kurz erklärt wird.

3.5.1 Konkatenation der `Pose3D` („+“-Operator)

Die Konkatenation meint das „anhängen“ einer Pose an eine andere. D.h.: Gegeben sei eine Pose `A` die die Koordinaten `x`,`y` und `z`, sowie die Rotationen `rotX`, `rotY` und `rotZ` hat. An diese soll eine Pose `B` mit `x'`,`y'` und `z'`, sowie `rotX'`, `rotY'` und `rotZ'` angehängt werden. Dann ist der Ursprung des Translationsvektors der Pose `B` genau am Ende des Translationsvektors der Pose `B`. Die Richtung des Translationsvektors der Pose `B` ist nicht absolut zum Koordinatensystem, sondern ergibt sich aus `rotX`, `rotY` und `rotZ`, den Rotationen der Pose `A`. (S. Abb. 3.1)

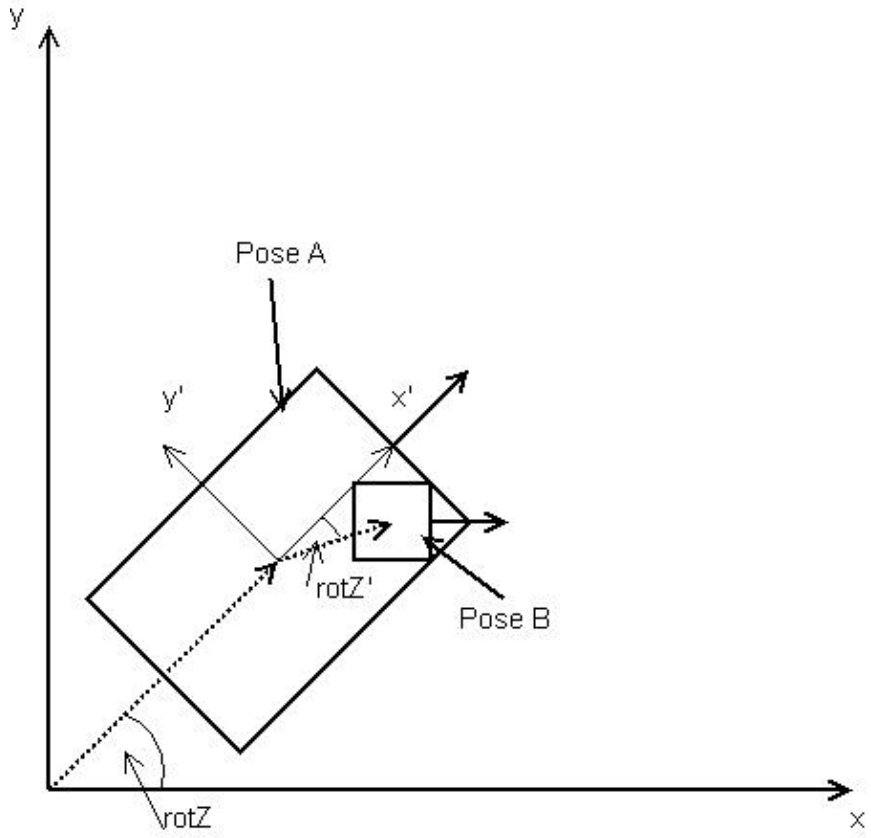

Abbildung 3.1: Konkatenation zweier Posen

3.6 Die Klassen `Matrix` und `RotationMatrix`

Diese Klassen wurden ebenfalls vom RoboCup Team Bremen geschrieben und mir zur Verfügung gestellt. Sie sind beide in der Datei `Rolland/Tools/Math/Matrix.ccp`

bzw. `Matrix.h` zu finden. Sie enthalten Matrizenoperationen, die hauptsächlich für die Konkatenation der `Pose3D` benötigt werden.

4 Erstellung einer zweidimensionalen Umgebungskarte (`EvidenceGrid`) aus 3D Daten

Der Weg zum zweidimensionalen `EvidenceGrid` hat sich nun ein wenig geändert. Im Wesentlichen wurden jedoch die 2D-Klassen und -Repräsentationen durch ihre 3D-Pendants ersetzt, wie man in der Modulansicht gut erkennen kann.(Abb. 4.1.)

4.1 Einlesen der Scanpunkte in der Klasse `LocalRobot`

Die Scanpunkte werden wie im 2D-Fall in der Klasse `LocalRobot` eingelesen. Nun müssen jedoch durch den dritten Laserscanner 683 Punkte mehr verarbeitet werden.

4.2 Umrechnen der Distanzen in 3d-Weltkoordinaten mit der Klasse `ScanPointsProvider3D`

4.2.1 Berechnung der Punkte in Weltkoodinaten

Zur Erzeugung der Punkte in Weltkoordinaten wird analog zum 2D Fall vorgegangen. (S. Anhang A.1.2, A.1.3, A.1.4.)
Es wird die Klasse `ScanPointsProvider3D` anstelle der alten Klasse `ScanPoints-Provider` benutzt. (Modifikation in `modules.cfg`.) Die errechneten Punkte sind nun Objekte vom Typ `ScanPoints3D::Point3D`, enthalten also wieder eine z-Koordniate für die dritte Dimension. Die Berechnung erfolgt wie im 2D-Fall, nur dass diesmal die entsprechenden 3D-Klassen benutzt werden.

4.2.2 Der Boden als Hindernis

Eine wesentliche Neuerung ist bei Betrachtung von drei Dimensionen, dass der Boden nun zunächst auch ein Hindernis darstellt: Die Laserstrahlen werden natürlich auch vom

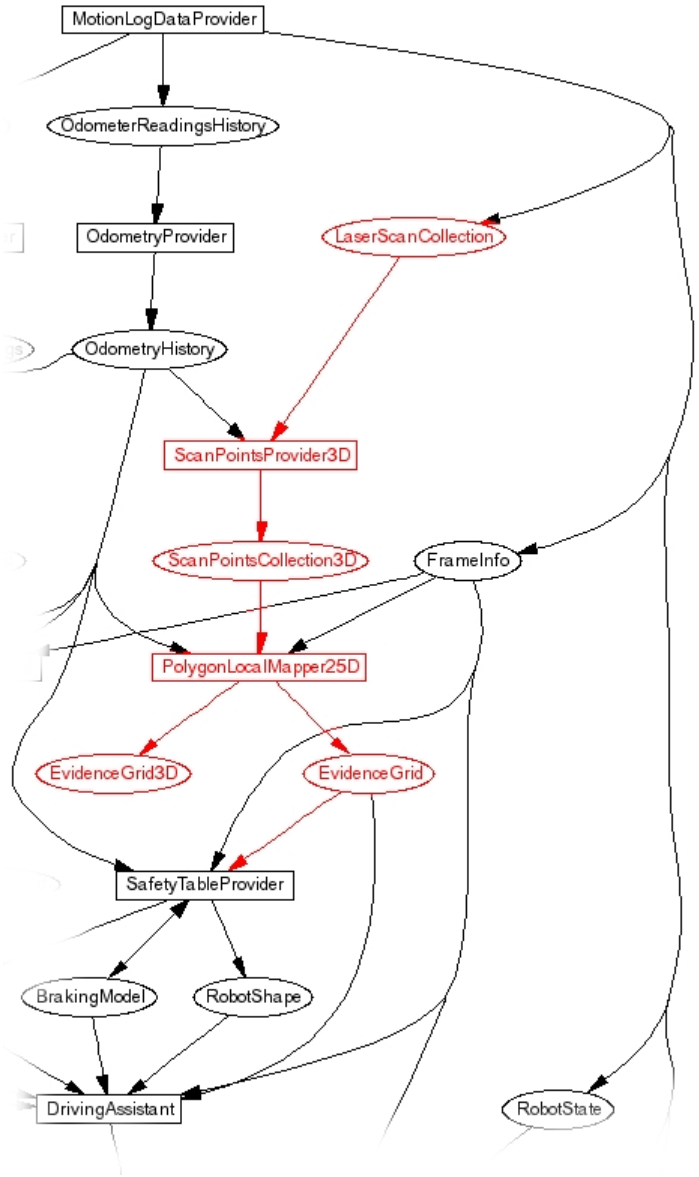

Abbildung 4.1: Modulansicht für die 3D-Navigation

Interessant für uns ist wieder der rot illustrierte Bereich. Die `LaserScanCollection` *ist wie im 2D-Fall die Eingabe. Die Klassen die dann jedoch benutzt werden sind zunächst der* `ScanPointsProvider3D`*, wo wieder die Punkte in Weltkoordinaten berechnet werden. Diese Punkte sind nun jedoch vom Typ* `ScanPointsCollection3D`*, und enthalten eine z-Koordinate. Sie sind Grundlage für die Berechnung des dreidimensionalen* `EvidenceGrid3D`*.*

Boden reflektiert, wodurch normalerweise *jeder* am Boden gescannte Punkt ein Hindernis ist. Darum wurde im `ScanPointsProvider3D` eine `obstacleHeightTolerance` eingeführt. Befindet sich also ein reflektierter Punkt um ± `obstacleHeightTolerance`

mm höher oder tiefer als der Boden ist er ein Hindernis, sonst nicht. Führe man z.B. über unebenen Boden könnte man hiermit auch die „Geländegängigkeit“ des Rollstuhls festlegen.

4.2.3 Verschiedene Arten von Hindernispunkten

Neben den bisher erwähnten Arten von Hindernispunkten (`infinite`, `isInIgnoreArea`, `nextToIgnoreArea`, `normal`) werden für die Navigation in 3D zwei weitere Typen ergänzt: `isOnFloor` sowie `isAbove`. Sie repräsentieren die Information, ob ein Punkt in Weltkoordinaten auf dem Boden (und somit kein Hindernis) oder über dem Rollstuhl (und somit ebenfalls kein Hindernis) ist.

4.2.4 Abgrunderkennung

Eine weitere wichtige Motivation für die Navigation in drei Dimensionen war die Abgrunderkennung. Ist die z-Koordinate eines Punktes kleiner als die eben erwähnte `-obstacleHeightTolerance`, wurde ein Abgrund erkannt. In diesem Fall wird mathematisch gesehen, der Laserstrahl gekürzt bis die z-Koordinate = 0 ist. Der Punkt wird dann mit den sich nach Umrechnung (clipping) ergebenden x- und y-Koordinaten als `normal`, also als Hindernis betrachtet.

4.2.5 Fehlmessungen

Durch die Erweiterung auf 3 Dimensionen tritt ein weiteres Problem auf: Fehlmessungen. Obwohl im 2D-Fall natürlich auch Fehlmessungen auftreten, fallen diese dort nicht so schwer ins Gewicht da wie in 2.3.1.2 beschrieben die Geraden zwischen Scanner und Hindernis immer wieder als „kein Hindernis“ angenommen werden können. Dies geschieht in 2D nur in *einer* Ebene die kontinuierlich neu eingescannt wird. Fehlmessungen, einzelne Punkte, werden also genauso kontinuierlich wieder eliminiert.
Im 3D-Fall gibt es bereits in der Simulation verhältnismäßig viele Fehlmessungen, da die Strahlen der horizontal angebrachten Laserscanner in SimRobot vermutlich als Objekte modelliert werden, und somit die Strahlen des schräg nach unten ausgerichteten Scanners reflektieren. Die so entstehenden einzelnen Punkte, „Ausreißer“, sind, zumindest für den Menschen, offensichtlich erkennbar. (S. Abb. 4.2.)
Daher wurde ein Algorithmus implementiert, der diese Ausreißer als Fehlmessung herausfiltert. Dieser Algorithmus hat nicht den Anspruch perfekt zu sein; für ein besseres Ergebnis würde man vermutlich eine Historie der Messungen mit einbeziehen. Er stellt

jedoch ein seinen Zweck ausreichend gut erfüllendes Werkzeug dar und wird im Folgenden kurz beschrieben:
Es werden immer fünf nebeneinander liegende Distanzwerte betrachtet, in der Mitte der zu analysierende. Davon werden immer jeweils die drei linken, die drei mittleren, sowie die drei rechten Distanzen als Einheit (Tripel) betrachtet. Weicht bei allen drei Tripeln der Distanzwert des mittleren Punktes um mehr als 5 cm vom Durchschnitt des jeweils rechts und links gelegenen Wertes ab, wird der mittlere der fünf Distanzwerte als Fehlmessung interpretiert. (S. Abb. 4.3, Anhang A.2.1.) Dieser Algorithmus könnte wie gesagt noch stark verbessert werden, wenn man die Historie der Messungen in betracht zöge. Dann würde man die Fehlmessungen am besten direkt im `EvidenceGrid` behandeln.

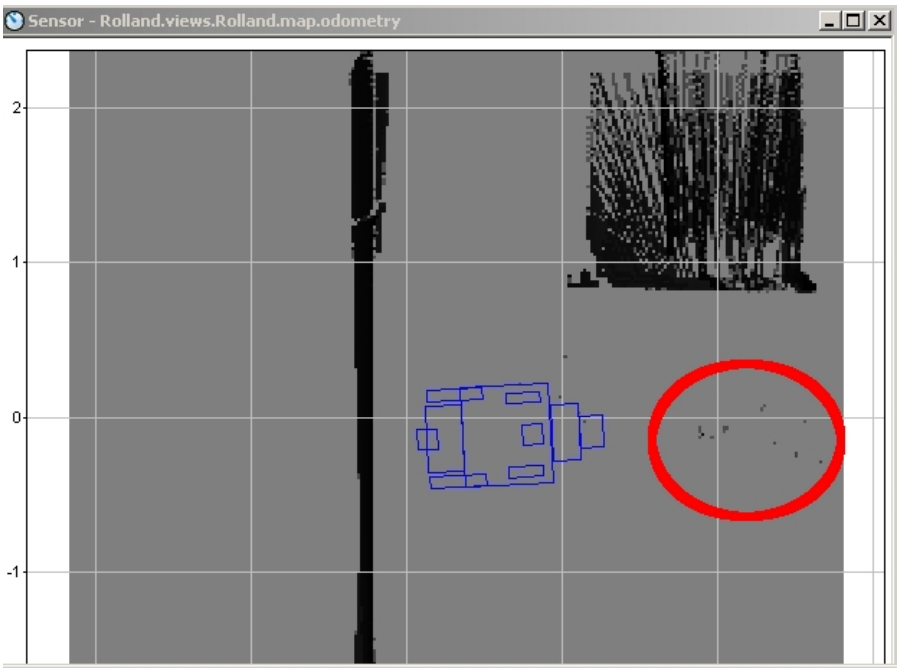

Abbildung 4.2: Fehlmessungen

Die im rot umrandeten Bereich erkennbaren Pixel sind Fehlmessungen, die vermutlich auf die Reflektion der Strahlen des Hokuyo-Scanners an den Strahlen der Siemens-Scanner zurückzuführen sind.

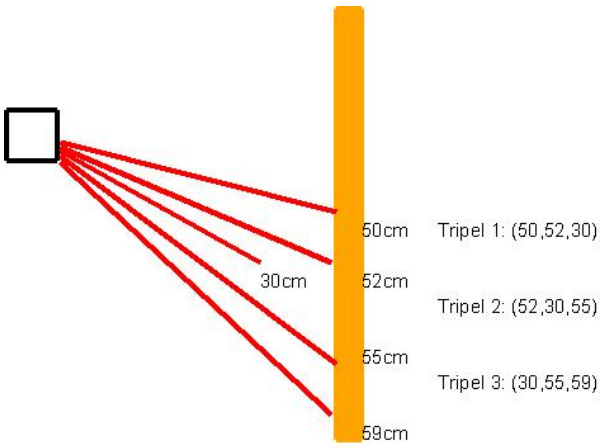

Abbildung 4.3: Entfernen von Fehlmessungen

Mittelwert der äuseren Distanzwerte von allen drei Tripeln unterscheidet sich um mehr als 5 cm vom jeweils mittleren Wert der Tripel. Daher liegt hier eine Fehlmessung vor.

4.3 Abbildung der Punkte in Weltkoordinaten auf das `EvidenceGrid3D` in der Klasse `PolygonLocalMapper25D`

In der Klasse `PolygonLocalMapper25D` werden analgog zum 2D-Fall die Punkte in Weltkoordinaten auf das `EvidenceGrid3D` übertragen. (S. Anhang A.1.5, A.1.6.) Hier waren jedoch einige Finessen und Überlegungen notwendig um das gewünschte Ergebnis zu erzielen.

Der Name dieser Klasse endet nicht mit „3D“, da ja die dritte Dimension eigentlich nur eine Hilfe ist, um das 2D-`EvidenceGrid` korrekt zu erzeugen. Das Mapping geschieht also quasi $2\frac{1}{2}$-dimensional.

4.3.1 Erzeugung des `EvidenceGrid3D`

Das `EvidenceGrid3D` ist ein 3-dimensionales `EvidenceGrid`, erweitert um eine z-Achse. Es ist in den Dateien EvidenceGrid3D.h bzw. EvidenceGrid3D.cpp, welche Weiterentwicklungen des 2-dimensionalen `EvidenceGrid` sind, deklariert und definiert.

Diese dreidimensionale Karte wird im Wesentlichen wie im 2D-Fall erzeugt. (S. Abschnitt 2.3.1.) Es werden dazu die jeweiligen 3D-Klassen und Repräsentationen verwen-

det.

4.3.1.1 Besonderheiten beim Clipping

Der Grund warum im 3D-Fall die Punkte über dem Rollstuhl das Flag `isAbove` bekommen, also im `PolygonLocalMapper25D` ignoriert werden, wird beim Clipping ersichtlich: Der Hokuyo URG04LX hat einen Strahlwinkel von insgesamt 240°. D.h. nach vorn unten geneigt angebracht strahlt er auch nach hinten oben, über den Rollstuhl. Wird jetzt ein nach hinten gerichteter Strahl (jenseits der nach vorn gerichteten 180°) mit `PolygonLocalMapper25D::clipPointP2()` soweit gekürzt, dass der daraus resultierende Punkt seitlich über dem Rollstuhl ist, würde er als Hindernis schräg über dem Kopf des Fahrers dargestellt. Bei der Projektion auf das zweidimensionale `EvidenceGrid` befände sich das Hindernis also später neben dem Fahrer. Da ein Punkt mit dem Flag `isAbove` jedoch gar nicht vom `PolygonLocalMapper25D` beachtet wird, passiert dies nicht.

4.3.1.2 Ausfüllen der Ebenen zwischen Laserscanner und Hindernis

Wenn ein Scanner einen bestimmten Punkt erfasst, kann wie im 2D-Fall davon ausgegangen werden, dass die Ebene zwischen Laserscanner und Punkt kein Hindernis enthält. Diese Ebene hat bei einem geneigt montierten Scanner nun jedoch dieselbe Neigung.
Es kann also die Hinderniswahrscheinlichkeit der Pixel im `EvidenceGrid3D`, die von dieser Ebene berührt werden, verringert werden. Die Wahrscheinlichkeit direkt null zu setzen wäre nicht empfehlenswert, da die Scanner durch abgefälschte Lichtreflektionen und andere Lichtquellen oft für nur eine Messung Hindernisse sehen, die in Wirklichkeit nicht da sind.

Die Linien werden mit der Methode `PolygonLocalMapper25D::blurShortenedLine()` gezeichnet, die jetzt den Bresenham-Algorithmus zum Zeichnen von Linien über Pixel verwendet. Die Linien werden ja eigentlich in das `EvidenceGrid3D` herein „gewischt“, da die Werte (Wahrscheinlichkeiten) nur verringert und nicht absolut gesetzt werden. Daher der Name „blur“`shortenedLine()`. Der Code der Funktion stammt aus dem Internet [BRESENHAM] und darf laut Inhaber der Webseite für das Projekt genutzt werden.

Es gibt hier ein Problem, denn wenn man zwei Linien über verschiedene Pixel je von einem Punkt zu einem anderen zeichnet, so kann es passieren, dass derselbe Pixel von beiden Linien ausgefüllt wird. Dies geschieht umso öfter, desto schlechter die Auflösung

ist. Da wir in z-Richtung eine sehr geringe Auflösung im Bereich von 10-20 Pixeln über das gesamte `EvidenceGrid3D` haben, geschieht dies gerade bei Linien die in z-Richtung schräg gezeichnet werden sehr oft. Und genau dies ist der Fall bei dem dritten Laserscanner, der schräg nach unten strahlt. (S. Abb. 4.4.)

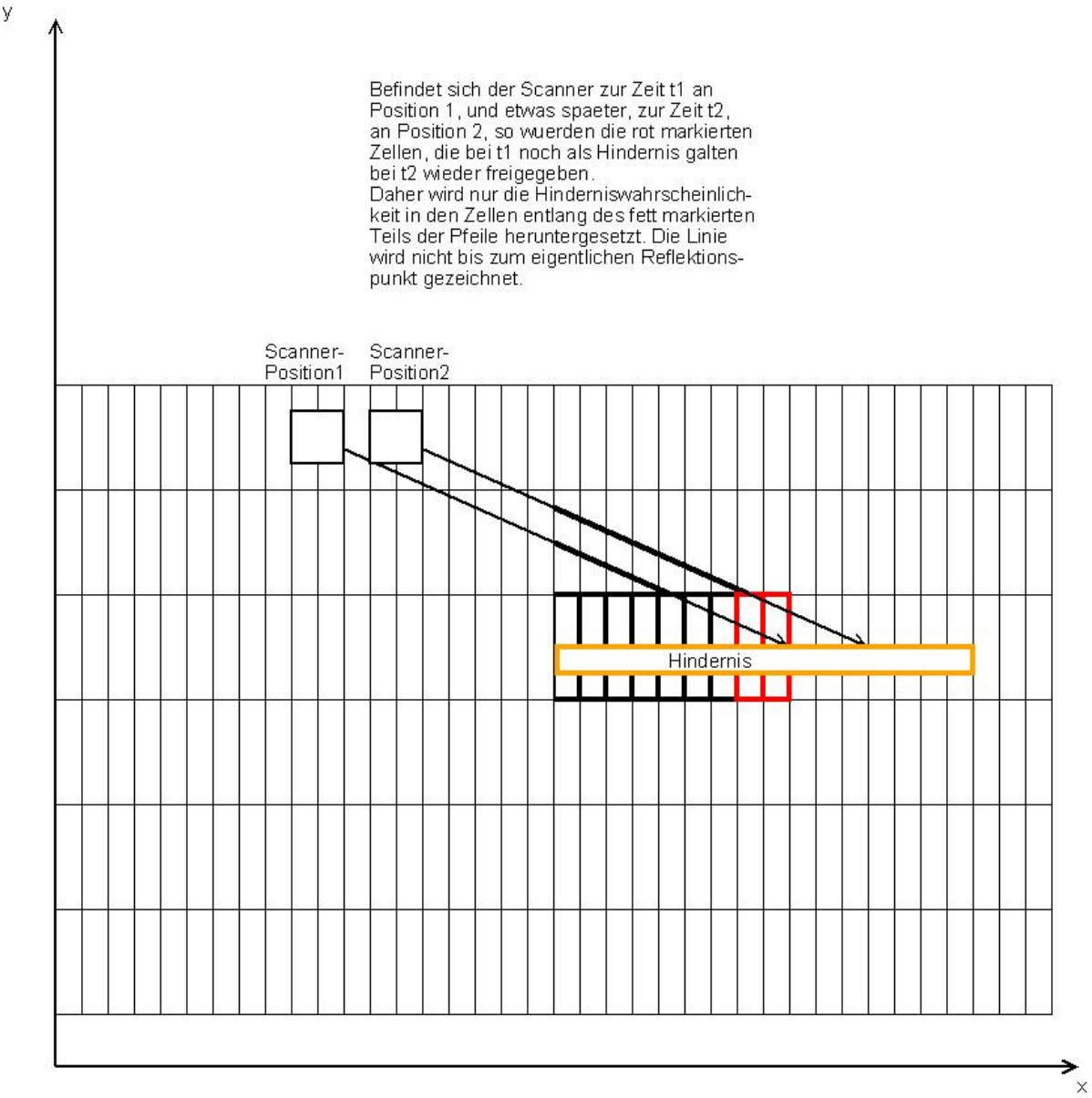

Abbildung 4.4: Entfernen von Hindernissen entlang einer verkürzten Linie

Die Linien werden nicht ganz bis zum Ende gezeichnet.

In der Praxis passiert folgendes: Angenommen der Rollstuhl fährt auf einen Tisch zu und der schräg angebrachte Laserscanner erfasst die Tischplatte. Der Rollstuhl bewegt sich um einige Millimeter nach vorn, nur soweit, dass der Strahl der von der Tischplatte reflektiert wird nun noch durch den Pixel geht, der vorher als Hindernis (Tischplatte) identifiziert wurde. Dann würde der Hinderniswahrscheinlichkeitswert dieses Pixel durch das Zeichnen der Linie fälschlicherweise dekrementiert, obwohl ja vorher ein Hindernis erkannt wurde. Daher zeichnet `blurShortenedLine()` nicht ganz bis zum Endpunkt, sondern nur bis zu dem Punkt, an dem die Gerade noch einen Pixel in irgendeiner Richtung vom Endpunkt entfernt ist. Sei die Position des Laserscanners beispielsweise (x,y,z)=(0,0,5) und die des zuerst erkannten Punktes auf der Tischplatte (15,15,2). Nachdem der Rollstuhl ein Stück vorwärts gefahren ist, erkennt er den Punkt (16,15,2) als Hindernis. Der Bresenham-Algorithmus würde dann wahrscheinlich die Linie durch den Punkt (15,15,12) zeichnen und somit das zuerst erkannte Hindernis löschen. Da er aber einen Pixel voher aufhört, würde die Line wahrscheinlich nur bis ungefähr (10,10,3) gezeichnet werden und das Hindernis bliebe erhalten.

Doch leider ist damit das Problem immer noch nicht vollständig gelöst. Wenn die Gerade vom Scanner zum Boden führt kann es sein, dass von ihr Pixel geschnitten werden, die ein Hindernis in einer Höhe zwischen Scanner und Boden repräsentieren sollen. Diese Pixel sind dann aber wahrscheinlich nicht Pixel die gleichzeitig den Boden repräsentieren, sondern befinden sich eine Ebene höher. Diese werden in so einem Fall leider, in der Annahme, dass auf Strecke des Laserstrahls zwischen Scanner und ein Pixel vor dem Boden kein Hindernis existiert, fälschlicherweise als „kein Hindernis“ interpretiert. (S. Abb. 4.5.)

Eine Möglichkeit beide Probleme zu lösen wäre, das Verhältnis von Auflösung der z-Achse zur Auflösung der x- bzw. y-Achse kleiner als oder gleich dem Tangens des Neigungswinkels des Laserscanners zu machen. Bei einem Winkel von 45° nach unten hieße das, dass die Auflösung der z-Achse mindestens so hoch wie die der x-bzw. y-Achse sein müsste, was jedoch einen viel zu hohen Rechenaufwand erforderte.

4.3.1.3 Einzeichnen der Hindernisse

Alle Punkte, die das Flag `normal` haben, werden mit der Methode `PolygonLocal-Mapper25D::drawObstacleLines()` in das `EvidenceGrid3D` eingezeichnet. Hier wird ebenfalls ein `blurLine()` verwendet. die Punkte werden also wieder nur „gewischt“ und

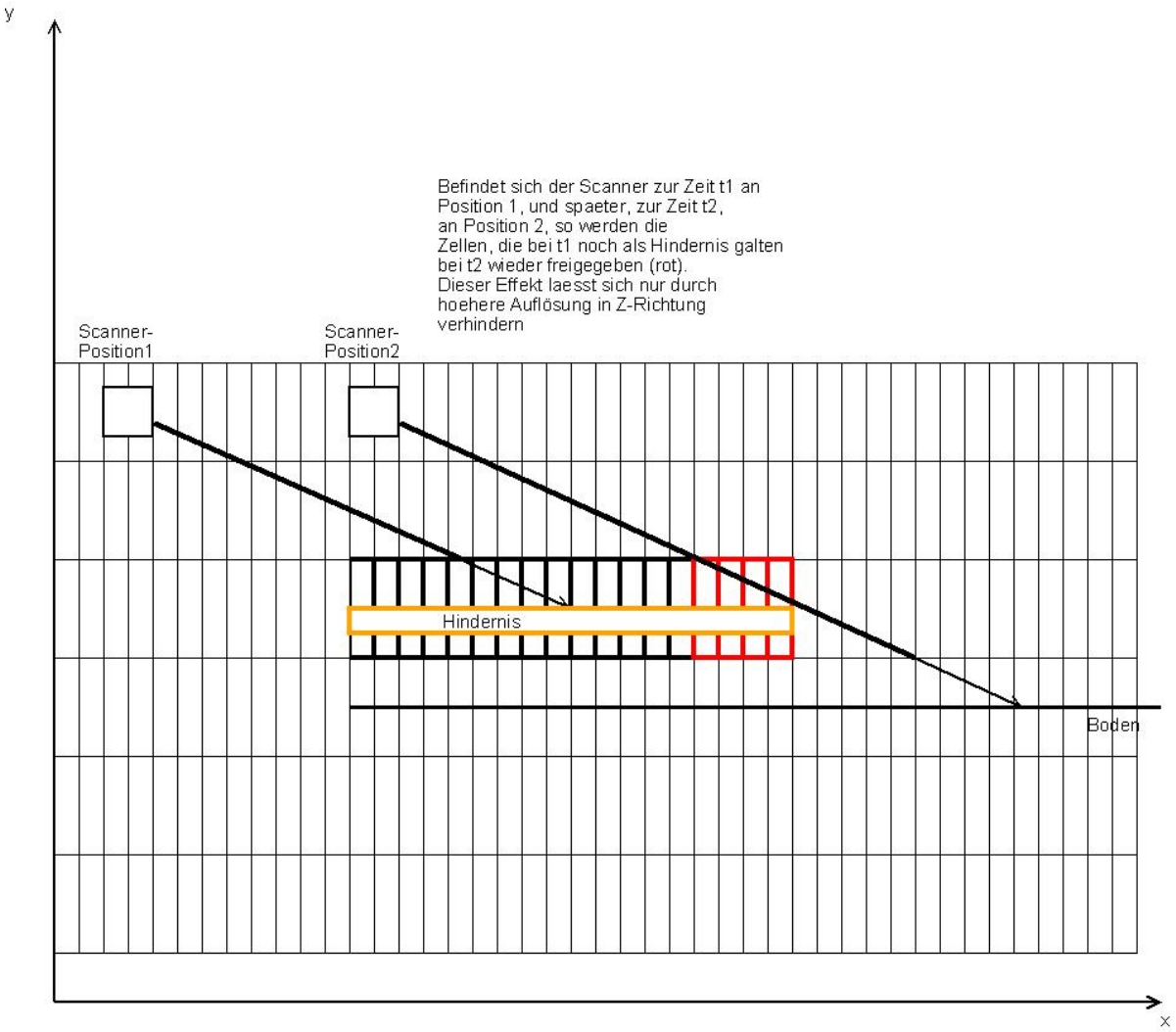

Abbildung 4.5: Entfernen von Hindernissen entlang der „Sichtline“ des Scanners: Hindernis wird fälschlicherweise entfernt.

Die letzten Pixel des Hindernisses werden wieder als „kein Hindernis“ interpretiert.

nicht absolut eingetragen.

4.3.1.4 Umwandlung des `EvidenceGrid3D` in ein 2D-`EvidenceGrid`

Ist das `EvidenceGrid3D` komplett errechnet, wird es in ein 2D-`EvidenceGrid` übertragen. Dabei wird einfach der Wert der höchsten Hinderniswahrscheinlichkeit über alle z zu einem festen 2-Tupel aus x und y in den entsprechenden Pixel des 2D-`EvidenceGrid` übertragen. Somit ist das Bild welches auf dem Odometry-view in der Simulation zu

sehen ist zwischen Hindernis und Scanner nicht mehr weiß sondern grau: „unbekannt“. Weiß würde es, wenn alle Pixel zu einem festen x-y-Paar im `EvienceGrid3D` die Hinderniswahrscheinlichkeit null hätten. Dies ist natürlich auch sinnvoller, denn nur weil man in einer bestimmten Höhe kein Hindernis sieht, heißt das ja nicht, dass darunter oder darüber kein Hindernis existiert.

5 Ergebnis und weiterführende Überlegungen

5.1 Funktionalität

Die im beschriebenen Szenario getestete Software verhält sich sehr gut, was die Hinderniserkennung angeht. Die Arbeit kann also grundsätzlich als voller Erfolg gewertet werden. Ein kritisches Objekt war beispielsweise ein Bett, dessen Liegefläche nicht erkannt wurde, sondern nur die Beine. Mit dem dritten Scanner wurde das gesamte Bett als Hindernis in das `EvidenceGrid` eingetragen. (S. Abb. 5.1). Ausserdem gab es im Modell des BAALL eine Art Tisch, in Form einer Platte die ohne zusätzliche Tischbeine einfach im rechten Winkel an der Wand angebracht war. (S. Abb. 5.2.) Auch dieser wurde nun nicht mehr gerammt.

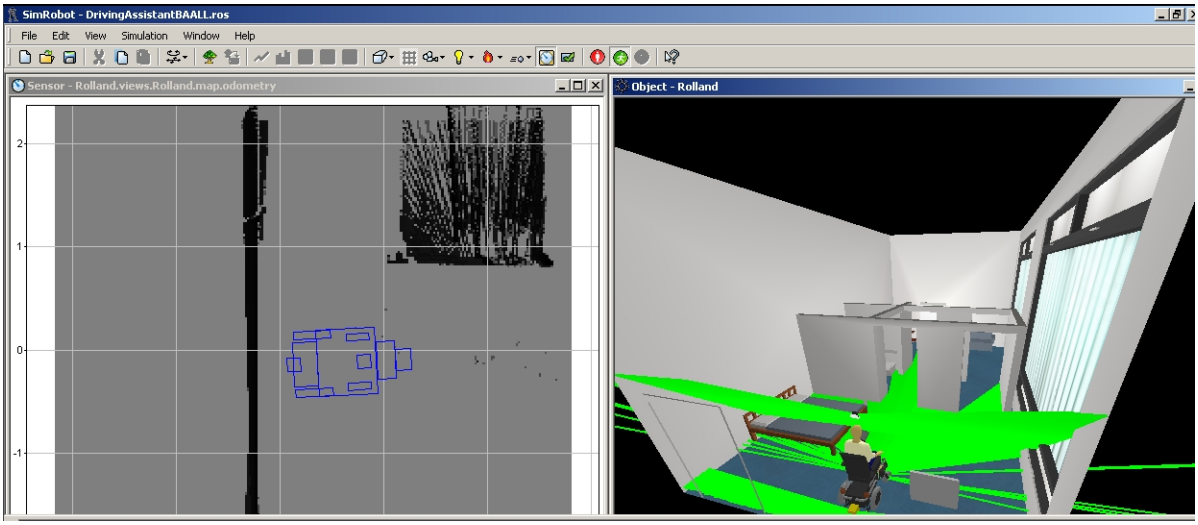

Abbildung 5.1: Erkennung von Hindernissen oberhalb der 12cm-Ebene: Bett

Desweiteren verhindert die Abgrunderkennung nun, dass der Rollstuhl Treppen o.Ä. hinunter fährt. (S. Abb. 5.3.) Ein Nachteil der Abgrunderkennung ist jedoch, dass auch

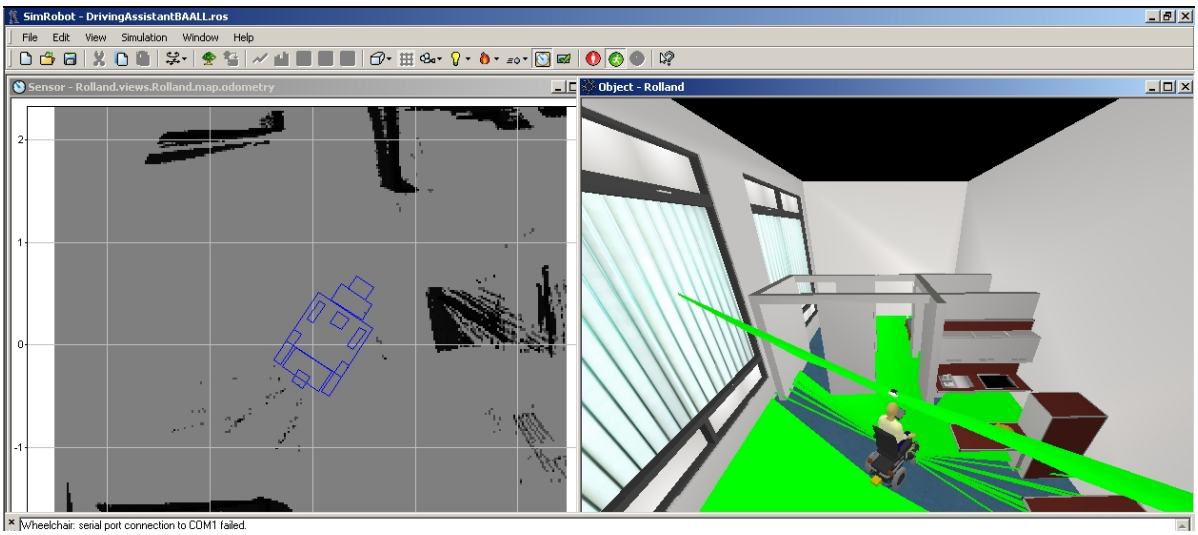

Abbildung 5.2: Erkennung von Hindernissen oberhalb der 12cm-Ebene: Tisch

Rampen als Abgründe gewertet werden, wo deren Höhe sich um den Wert `ostacleHeightTolerance` von der Höhe des Rollstuhls unterscheidet. (S. Abb. 5.4.) Hier müsste man mit Gradienten anstatt mit absoluten Höhen als Hindernisindikator arbeiten, wofür die Architektur jedoch ganz und gar nicht ausgelegt ist.

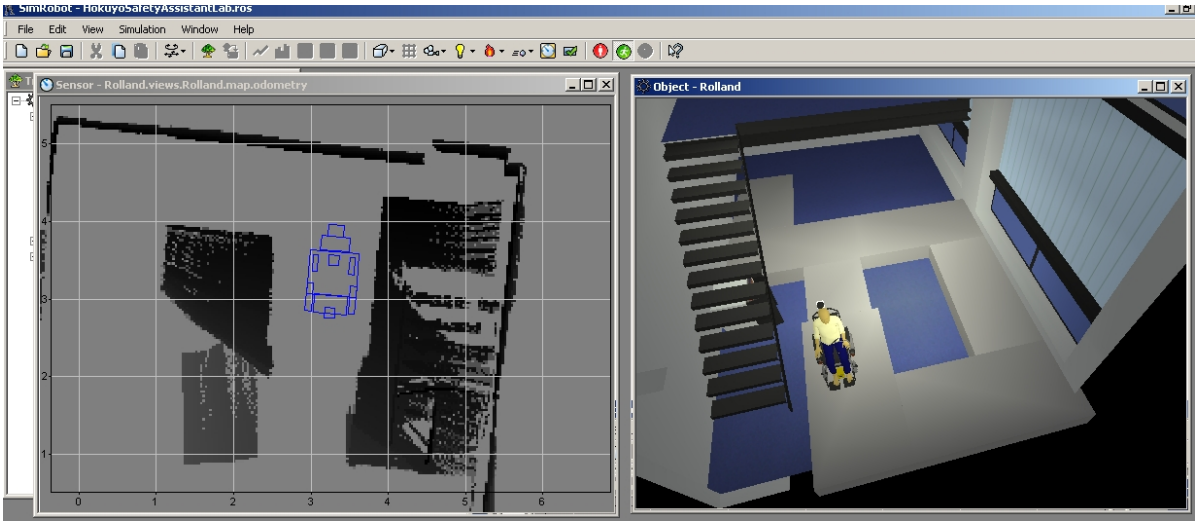

Abbildung 5.3: Abgrunderkennung

Der Rollstuhl navigiert sicher über den Parcour.

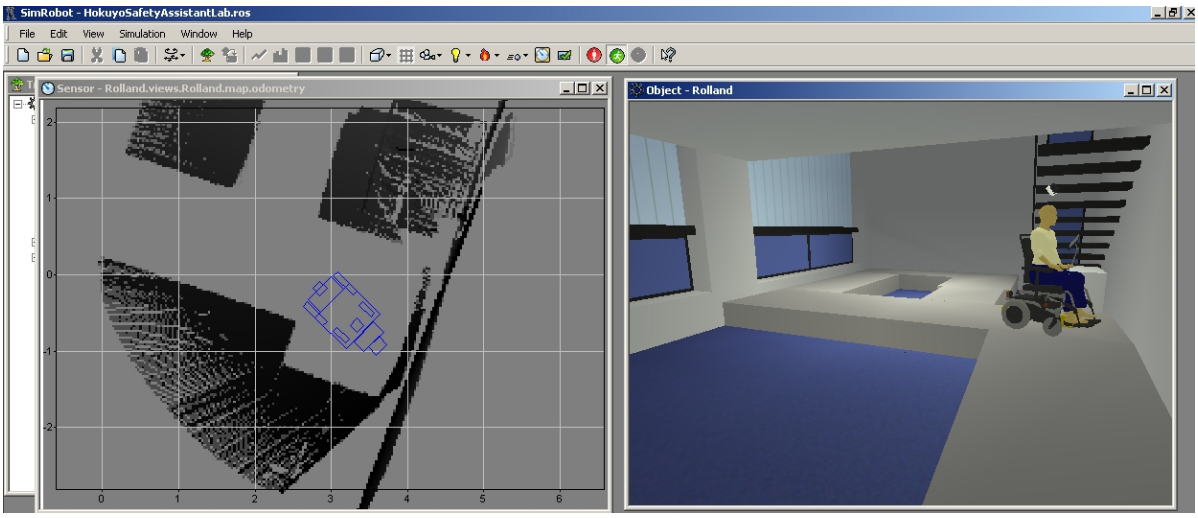

Abbildung 5.4: Interpretation von Rampen als Hindernis

Leider fährt Rolland keine Rampen mehr herunter oder herauf: Sie werden wegen des Höhenunterschieds als Hindernis erkannt.

5.2 Performance

Natürlich steigert sich durch die Erweiterung um eine *Dimension* die benötigte Rechenleistung um ein Vielfaches. Dies kann jedoch dadurch reduziert werden, dass die Auflösung in z-Richtung sehr gering gehalten wird. Im Versuch gab es nur 20 Ebenen auf der z-Achse, welche vollkommen ausreichten. Desweiteren kommen die 683 ausgelesenen Distanzwerte des Hokuyo-Scanners hinzu, welche die Performance zusätzlich stark beeinträchtigen. Obwohl alle anderen Erweiterungen kaum Leistungseinbußen bringen, wird die Anzahl der Frames pro Sekunde (fps) im Vergleich zur 2-dimensionalen Auswertung auf ca. ein Drittel reduziert. Die leistungsoptimierte Version läuft auf einer Maschine mit 2GHz-Prozessor und 1 GB RAM bei ca. 20 fps, was in jedem Fall noch ausreichend ist.
Ein viel versprechender Ansatz zur Steigerung der Performance besteht darin, die Grafikkarte für jegliche Koordinatenberechnungen heranzuziehen.

5.3 Der bewegliche Scanner

Wird trotz des Algorithmus zur Beseitigung von Fehlmessungen im 3-dimensionalen `EvidenceGrid` irgendwo ein Pixel als Hindernis identifiziert, so müsste der Rollstuhl zumindest mit fest arretiertem Laserscanner erst wieder zurück fahren, um diesen Punkt

neu als „kein Hindernis“ erkennen zu können. In der Simulation passiert dies sehr häufig, da die Laserstrahlen dort scheinbar als „feste Gegenstände“ modelliert werden. Trifft ein Strahl des Hokuyo-Scanners also auf einen Strahl des horizontal ausgerichteten Siemens LS4, so wird dieser als Hindernis gewertet. Praktisch heißt das für die Simulation, dass hin und wieder einzelne Pixel in einer sonst freien Fläche als Hindernis auftauchen. Und auch im Realen kann es natürlich passieren, dass etwa eine Fliege die vor dem Rollstuhl her fliegt als Hindernis erkannt wird.

Abhilfe würde hier geschaffen, wenn der Scanner mechanisch beweglich wäre. Etwa mit einer beweglichen Halterung und einem Servomotor, der von der Software gesteuert werden kann. Wenn der Scanner dann mehr oder weniger intelligent auf und abschwenken würde, könnten solche „schein“-Hindernisse sehr leicht entlarvt werden. Auch bewegliche Hindernisse, etwa die Füße eines vorauslaufenden Menschen, könnten schnell wieder entfernt werden, wenn sie sich denn tatsächlich nicht mehr bei den gemessenen Koordinaten befinden.

5.4 Hintergründige Überlegungen zu unsicherem Wissen

Selbst dem Menschen mit seiner hervorragenden Sensorik kann man ein Bein stellen und er wird stolpern weil seine Augen es soweit unten nicht erfassen können. Er bewegt sich, obwohl ihm seine Umgebung nicht vollkommen bekannt ist. Vor diesem Hintergrund kann es also keine Maschine geben, die sich 100%ig sicher ist, dass sie durch ihre Interaktionen mit der Umwelt keinen Schaden anrichtet. Man muss also eine Grenze definieren, ab der man Aktionen zulässt. Hier wurde diese im `EvidenceGrid` auf den Wert 128 gesetzt, also genau dem Mittelwert. Diese Grenze wird im kommerziellen Bereich mit zunehmender „Roboterisierung“ der Gesellschaft wohl von Versicherungen, TÜVs und anderen Sicherheitsüberwachungseinrichtungen bestimmt werden.

A Anhang

A.1 Wichtige Klassen und Methoden

A.1.1 `Pose2D`, `Pose3D`

Beschreibung

Repräsentiert Position und Rotation eines Objektes, enthält Funktionen für geometrische Operationen.

Attribute

- `rotation`: Rotation des Objektes als Matrix
- `translation`: Translation des Objektes als 3D-Vektor

Wichtige Methoden und Operatoren

- +-Operator: Konkatenation einer `Pose2D` bzw. `Pose3D` mit einer anderen.

A.1.2 `ScanPoints::Point`, `ScanPoints3D::Point3D`

Beschreibung

Stellt die mit den Laserscannern erfassten Punkte in Weltkoordinaten dar. Ist Kindklasse von `Vector3` bzw. `Vector2`, erbt also also x, y und ggf. z-Koordinate.

Attribute

- `type`: Typ des erfassten Punktes.
  - `infinite`: Wenn gemessene Distanz größer der vom Scanner erkennbaren Maximaldistanz. Diese Werte werden ignoriert, da diese hohen Distanzen auf eine Fehlmessung hinweisen.

- `isInIgnoreArea`: Diese `IgnoreAreas` wurden hinzugefügt, damit die vom Scanner erfassten Castor-Räder beispielsweise nicht als Hindernis erkannt werden.
- `nextToIgnoreArea`: Manchmal wird der Laserstrahl nur teilweise vom Castorrad reflektiert. Auch dann muss dieser Distanzwert ignoriert werden.
- `normal`: Dieser Wert ist gültig und wird für die weiterführende Navigation verwendet.
- `isAbove`: (Nur `Point3D`.) Der Punkt befindet sich oberhalb des Rollstuhls und braucht nicht beachtet zu werden.
- `isOnFloor`: (Nur `Point3D`.) Der Punkt befindet sich auf dem Boden und wird nicht beachtet, da der Boden kein Hindernis darstellt.

A.1.3 `ScanPoints`, `ScanPoints3D`

Beschreibung

Stellt die mit den Laserscannern erfassten Punkte in Weltkoordinaten dar.

Attribute

- `points3D`: Alle mit den Laserscannern erfassten Punkte. Diese sind jeweils vom Typ `ScanPoints::Point` bzw. `ScanPoints3D::Point3D`.

A.1.4 `ScanPointsProvider`, `ScanPointsProvider3D`

Beschreibung

Erstellt aus den gemessenen Distanzen der Laserscanner die Koordinaten der Punkte in Weltkoordinaten.

Wichtige Methoden und Operatoren

- `update()`: Wird aufgerufen wenn neue Messung der Laserscanner vorliegt. Neue Koordinaten werden berechnet.

A.1.5 `EvidenceGrid`, `EvidenceGrid3D`

Beschreibung

Repräsentiert die Umgebung des Rollstuhls. Es besteht aus Pixeln und hat eine relativ geringe Auflösung (250 x 250 x 20). Der Inhalt dieser Pixel ist ein Wert zwischen 0 und 255, welcher die Hinderniswahrscheinlichkeit an dem entsprechenden Punkt wiederspiegelt. 0 heißt „kein Hindernis“, 128 „unbekannt“, 255 „Hindernis“. Alle Werte dazwischen sind möglich. Der Rollstuhl betrachtet eine Zelle als passierbar wenn ihr Wert ≤ 128 beträgt.

Attribute

- `center`: Zentrum der Karte in Weltkoordinaten. Wird als `Vector3` bzw. `Vector2` angegeben.
- `timeStamp`: Zeitpunkt der Erstellung.
- `height`: Repräsentierte Höhe in mm.
- `width`: Repräsentierte Breite in mm.
- `depth`: Repräsentierte Tiefe in mm.
- `cellDepth`: Repräsentierte Tiefe pro Zelle in mm.
- `cellSize`: Repräsentierte Breite und Höhe pro Zelle in mm.

Wichtige Methoden und Operatoren

- `worldToGrid`: Liefert Zelle, die einen bestimmten Punkt in Weltkoordinaten repräsentiert.
- `gridToWorld`: Liefert Weltkoordinaten des Mittelpunktes einer angegebenen Zelle.

A.1.6 `PolygonLocalMapper`, `PolygonLocalMapper25D`

Beschreibung

Übertragt Punkte in Weltkoordinaten auf `EvidenceGrid` bzw. `EvidenceGrid3D`.

Attribute

- `polyPoints3D`: Koordinaten der als Hindernis erkannten Pixel im `EvidenceGrid` bzw. `EvidenceGrid3D`.

Wichtige Methoden und Operatoren

- `update()`: Ist eine neue `ScanPointsCollection` bzw. `ScanPointsCollection3D` erstellt worden, wird diese Methode aufgerufen, um eine neues `EvidenceGrid` bzw. `EvidenceGrid3D` zu erstellen.
- `clipPointP2()`: Projiziert einen ausserhalb des `EvidenceGrid` bzw. `EvidenceGrid3D` gelegenen Punkt an dessen Rand.
- `fillScanBoundary()`: Markiert die Ebene zwischen Scanner und Hindernis als passierbar, da angenommen werden kann, dass sich dort kein Hindernis befindet.
- `drawObstacleLine()`: Zeichnet Hindernisse gemäs der `polyPoints3D`.
- `blurShortenedLine()`: Wischt nach Bresenham eine Gerade durch eine dreidimensionale Karte.

A.2 Auszüge aus dem Quelltext

A.2.1 Algorithmus zur Beseitigung der Fehlmessungen

```
//Even distances from LaserScan. If there is one Distance A, one Distance B and one Distance C
//which come from the scanner in this order, distance B should not be too different from A and C,
//but be more or less the average of the two other distances. If not, its a mismeasurement
//and will be treated as infinite and as such ignored.
const int ddt = 50; //distance difference threshold of 5 cm
int distAvg1;
int distAvg2;
int distAvg3;
bool mismeasurement = true;
//TODO: This could be more intelligent and taking more pixels into account.
//      It also would fit better as an own function.

//If distance value hase one value before, and one after
if (i >= 2 && i < laserScan.distances.size()-2) {

  distAvg1 = (laserScan.distances[i-2]+laserScan.distances[i]) / 2;
  distAvg2 = (laserScan.distances[i-1]+laserScan.distances[i+1]) / 2;
  distAvg3 = (laserScan.distances[i]+laserScan.distances[i+2]) / 2;
  if (    ( laserScan.distances[i-1] > (distAvg1 - ddt)
   && laserScan.distances[i-1] < (distAvg1 + ddt))
      ||  ( laserScan.distances[i] > (distAvg2 - ddt)
       && laserScan.distances[i] < (distAvg2 + ddt))
      ||  ( laserScan.distances[i+1] > (distAvg3 - ddt)
       && laserScan.distances[i+1] < (distAvg3 + ddt))
      ) {
    mismeasurement = false;
  }
}
```

A.3 Konfigurationsdateien

A.3.1 Einstellungen am Roboter: `settings.cfg`

Die `laserScanners`-Sektion der `settings.cfg`:

```
[laserScanners]
# type name low_baudrate high_baudrate x_offset y_offset z_offset
# x_rotation y_rotation z_rotation calibration_offset calibration_factor
# type=(siemensLS4|hokuyoURG04LX)
# name=<internally used name for scanner>
# low_baudrate ="none" to use high_baudrate
# typical baud rates: LS4: 59100, 109700, 384000, 76800 URG04LX: 115200
# W A R N I N G, support for high baud rate not supported with URG04LX!
# origin of coordinates: middle between main wheels
# degrees: 0-359
# distances: in millimeter
# Sharit-wheelchair, measured 2007-07-03, B. Gersdorf:
# Coordinates for the simulation:
siemensLS4     front   none   768000  573  0  142     0   0  0  -35 0.992
siemensLS4     back    none   768000 -188  0   142    0  0   180  -15 0.995
hokuyoURG04LX   top   115200  115200 315   0   1662  0   45  0  0   0
# Real coordinates:
#siemensLS4 front none   768000  493  0  142     0   0  0  -35 0.992
#siemensLS4 back  none   768000 -188  0   142    0  0   180  -15 0.995
```

A.3.2 RoSiML-Repräsentation des Hokuyo-Laserscanners

Repräsentation des Hokuyo-Laserscanners in RoSiML:

```
<Macro name="URG04LX">
  <Compound name="URG04LX">
    <Elements>
      <DistanceSensor name="left">
        <Rotation z="60.18"/>
        <Resolution x="342" y="1"></Resolution>
        <OpeningAngles x="120" y="0.0001"></OpeningAngles>
        <Range near="0.02" far="4"></Range>
        <SphericalProjection/>
      </DistanceSensor>
      <DistanceSensor name="right">
        <Rotation z="-59.82"/>
        <Resolution x="341" y="1"></Resolution>
        <OpeningAngles x="120" y="0.0001"></OpeningAngles>
        <Range near="0.02" far="4"></Range>
        <SphericalProjection/>
      </DistanceSensor>
      <Box name="bottom" length="0.10" width="0.10" height="0.04">
        <Translation z="-0.035"/>
        <Appearance ref="rolland-scanner-hokuyo"/>
      </Box>
  <Cylinder name="window" radius="0.050" height="0.07">
    <Rotation y="-9.0"/>
  </Cylinder>
    </Elements>
  </Compound>
</Macro>
```

Zu beachten ist hier, dass der Laserscanner in der Simulation eigentlich aus zwei Scannern besteht. Einer deckt den linken Bereich, einer den rechten Bereich ab.

A.3.3 Modulkonfiguration

Modulkonfiguration für die 3D-Betrachtung. Im 2D-Fall benutzte Module wurden mit # auskommentiert.

```
[Shared]

[Modules]
ChampV2Readings ChampV2Provider
#DLAExportResult DLAExporter
JoystickReadings ChampV2Provider
FrameInfo ChampV2Provider
RobotState ChampV2Provider
OdometerReadingsHistory ChampV2Provider
OdometerState ChampV2Provider
LaserScanCollection LS4Provider
ScannerStateCollection LS4Provider
ScannerPowerRequestCollection LS4Provider
OdometryHistory OdometryProvider
#ScanPointsCollection ScanPointsProvider
ScanPointsCollection3D ScanPointsProvider3D
#ScanSegmentsCollection ScanSegmentsProvider
#EvidenceGrid PolygonLocalMapper
EvidenceGrid PolygonLocalMapper25D
EvidenceGrid3D PolygonLocalMapper25D
MotionDriveRequest MedianFilteredJoystickDriving
SmoothMotionRequest MotionRequestProvider
AssistedMotionRequest SmoothMotionControl
#SafeMotionRequest DrivingAssistant
BrakingModel SafetyTableProvider
RobotShape SafetyTableProvider
SafetyCTSet SafetyTableProvider
ControllerMotionRequest DrivingAssistant # SafetyOnlySafetyLayer
FailSafeMotionRequest VThetaController
FailSafeDriveRequest FailSafeDriveRequestProvider
SignalsRequest HardwareSafety
DriveRequest HardwareSafety
```

Abbildungsverzeichnis

1.1 Rolland in der Simulation 2

2.1 Modulansicht für die 2D-Navigation 4

3.1 Konkatenation zweier Posen 9

4.1 Modulansicht für die 3D-Navigation 12
4.2 Fehlmessungen 14
4.3 Entfernen von Fehlmessungen 15
4.4 Entfernen von Hindernissen entlang einer verkürzten Linie 17
4.5 Entfernen von Hindernissen entlang der „Sichtline“ des Scanners: Hindernis wird fälschlicherweise entfernt. 19

5.1 Erkennung von Hindernissen oberhalb der 12cm-Ebene: Bett 21
5.2 Erkennung von Hindernissen oberhalb der 12cm-Ebene: Tisch 22
5.3 Abgrunderkennung 22
5.4 Interpretation von Rampen als Hindernis 23

Literaturverzeichnis

[BRESENHAM] Bresenham Algorithmus,
http://www.cit.gu.edu.au/ anthony/info/graphics/bresenham.procs
Zugriff am 17.7.2008

[ROSIML] RoSiML-Dokumentation,
http://www.informatik.uni-bremen.de/spprobocup/RoSiML/RoSi.html
Zugriff am 18.9.2008

[SIMROBOT] T. Laue, K. Spiess, T. Röfer (2006). SimRobot - A General Physical Robot Simulator and Its Application in RoboCup. In A. Bredenfeld, A. Jacoff, I. Noda, Y. Takahashi (Hrsg.), RoboCup 2005: Robot Soccer World Cup IX, Nr. 4020, S. 173-183, Lecture Notes in Artificial Intelligence. Springer.

[VISUALSTUDIO] C. Skibo, M. Young, B. Johnson (2006). Arbeiten mit Microsoft Visual Studio 2005, Microsoft Press Deutschland

[ROLLAND] A. Lankenau, T. Röfer, B. Krieg-Brückner (2003). Self-Localization in Large-Scale Environments for the Bremen Autonomous Wheelchair. In C. Freksa, W. Brauer, C. Habel, K. F. Wender (Hrsg.), Spatial Cognition III, Nr. 2685, S. 34-61, Lecture Notes in Artificial Intelligence. Springer.