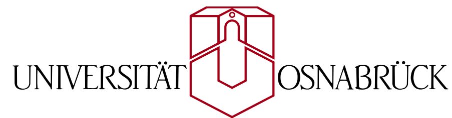


Institut für Informatik
AG Wissensbasierte Systeme

Automatische Generierung dreidimensionaler Polygonkarten für mobile Roboter

Dissertation

zur Erlangung des Doktorgrades (Dr. rer. nat.)
des Fachbereichs Mathematik/Informatik
der Universität Osnabrück

vorgelegt von

Thomas Wiemann

April 2013

Erstgutachter:	Prof. Dr. Joachim Hertzberg
Zweitgutachter:	Prof. Dr. Andreas Nüchter
Drittgutachter:	Prof. Michael Beetz, PhD

Zusammenfassung

Die 3D-Kartierung von Umgebungen spielt in der Robotik eine zunehmend wichtige Rolle. Mit Hilfe von 3D-Sensoren lassen sich dreidimensionale Umgebungen präzise erfassen. Allerdings vermessen selbst hochaufgelöste Scanner Oberflächen nur stichprobenartig. Zudem verbrauchen die gewonnenen Punktwolken viel Speicher. Eine Möglichkeit, die Diskretisierung aufzulösen und die Darstellung zu optimieren, ist, eine polygonale Umgebungsdarstellung aus den Punktdaten zu erzeugen. In dieser Arbeit wird ein Verfahren vorgestellt, mit dem sich komprimierte Polygonkarten automatisch erstellen lassen. Die Oberflächenrekonstruktion basiert auf einem modifizierten Marching-Cubes-Algorithmus. Die mit diesem Verfahren erzeugten Polygonnetze werden durch Optimierungsschritte in eine kompakte Darstellung überführt, die sich für Anwendungen in der Robotik einsetzen lässt, wie anhand von verschiedenen Einsatzbeispielen demonstriert wird.

Abstract

Three dimensional environment mapping plays an increasingly important role in mobile robotics. 3D sensors can be used to create very precise point clouds of the mapped scenes. But even with high point density, these devices only deliver a sampling of the present surfaces, not a continuous representation. Besides that, dealing with point clouds requires a lot of memory. One approach to get a continuous environment model is to calculate a polygonal mesh based on the sampled data. This thesis presents a method to automatically produce compressed polygonal meshes of arbitrary environments using a modified Marching Cubes algorithm. The initially created models are optimized using several filters to create an optimal mesh that can be used in robotic applications. The practical usability of the created maps is shown on several examples.

Danksagung

Besonders bedanken möchte ich mich bei Prof. Dr. Joachim Hertzberg für die intensive Betreuung während der Bearbeitung des Themas und für die angenehmen Arbeitsathmossphäre, die er in seiner Arbeitsgruppe pflegt. Bedanken möchte ich mich auch bei Prof. Dr. Andreas Nüchter für den kreativen Input und bei Prof. Dr. Michael Beetz, der sich bereit erklärt hat, mich als Drittgutachter zu betreuen.

Ein weiterer großer Dank gilt Kai Lingemann, der mir immer wieder geduldig geholfen hat, wenn ich beim Setzen mit LaTeX mal wieder etwas nicht richtig hinbekommen habe. Auch meinen anderen Kollegen – Sven Albrecht, Martin Günther, Jochen Sprickerhof und Sebastian Stock – sei in dieser Stelle für die vielen Hilfestellungen und anregenden Gespräche bei offenen Problemen gedankt.

Ein weiteres großes Dankeschön geht an Marion Wiemann und Klaus-Jürgen Gran für die viele Arbeit, die sie in das Korrekturlesen der Arbeit gesteckt haben.

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Wissenschaftlicher Beitrag . . . 2
1.2 Zielsetzungen . . . 2
1.3 Aufbau der Arbeit . . . 3

2 Oberflächenrekonstruktion aus 3D-Punktwolken **5**
2.1 Sensoren zur Erstellung dreidimensionaler Punktwolken . . . 6
2.1.1 3D-Laserscanner . . . 6
2.1.2 3D-Kameras . . . 8
2.1.3 Weitere Sensoren . . . 11
2.1.4 Fazit 3D-Sensoren . . . 11
2.2 Registrierung von 3D-Punktwolken . . . 11
2.2.1 Registrierung mit Markern . . . 11
2.2.2 Markerlose Registrierung . . . 12
2.3 Erzeugung von Polygonnetzen mit Marching Cubes . . . 15
2.3.1 Datenstrukturen für Polygonnetze . . . 15
2.3.2 Marching Cubes . . . 21
2.3.3 Distanzfunktionen für Marching Cubes . . . 23
2.3.4 Erweiterte Marching-Cubes-Varianten . . . 31
2.4 Weitere Rekonstruktionsverfahren . . . 35
2.4.1 Delaunay-Triangulation . . . 35
2.4.2 Smart Growing Cells . . . 37
2.4.3 Ball-Pivoting . . . 37
2.4.4 Direkte Triangulation auf Tiefenbildern . . . 38
2.4.5 Rekonstruktion durch Model-Fitting . . . 38
2.5 Meshoptimierung . . . 39
2.6 Normalenschätzung für 3D-Punktwolken . . . 41

3 Generierung optimierter Polygonkarten **45**
3.1 Anforderungen an das Rekonstruktionsverfahren . . . 45
3.1.1 Zielsetzungen . . . 46
3.1.2 Zu implementierende Funktionalitäten . . . 48
3.2 Software-Architektur . . . 50

3.2.1 IO-Bibliothek . . . 51
3.2.2 Rekonstruktionsbibliothek . . . 54
3.2.3 Meshoptimierung . . . 57
3.2.4 Zusammenfassung . . . 58
3.3 Normalenschätzung . . . 58
3.3.1 Least-Squares-basierte Normalenschätzung . . . 60
3.3.2 RANSAC-basierte Normalenschätzung . . . 60
3.3.3 Adaptive Anpassung der k-Nachbarschaft . . . 63
3.4 Rekonstruktion mit Marching Cubes . . . 65
3.4.1 Gitterstruktur für die Marching-Cubes-Rekonstruktion . . . 65
3.4.2 Auswertung der Distanzfunktion . . . 68
3.4.3 Standard Marching Cubes . . . 70
3.4.4 Marching Tetrahedrons . . . 71
3.4.5 Extended Marching Cubes . . . 72
3.4.6 Planar Marching Cubes . . . 74
3.5 Meshoptimierung und Segmentierung . . . 76
3.5.1 Aufbau einer Halbkantendarstellung . . . 76
3.5.2 Filterung von Artefakten . . . 79
3.5.3 Schließen von Löchern . . . 81
3.5.4 Extraktion und Optimierung zusammenhängender planarer Regionen . . . 85
3.6 Automatische Erzeugung von Texturen . . . 102
3.6.1 Texture-Mapping . . . 102
3.6.2 Automatische Generierung von Texturen aus Laserdaten . . . 104
3.6.3 Ausblick . . . 106
3.7 Parallelisierung mit OpenMP . . . 106

4 Evaluation **109**
4.1 Kartengenauigkeit . . . 109
4.1.1 Rekonstruktionen aus hochaufgelösten Scans . . . 110
4.1.2 Umgebungsrekonstruktion mit einem mobilen Roboter . . . 115
4.1.3 Fazit Geometrische Genauigkeit . . . 117
4.2 Texturemapping . . . 118
4.2.1 Evaluation eingefärbter Punktwolken . . . 119
4.2.2 Qualität der Texturen . . . 124
4.2.3 Weitere Anwendungen für Texturen . . . 126
4.2.4 Fazit automatische Texturerstellung . . . 127
4.3 Laufzeitanalyse und Speicherverbrauch . . . 127
4.3.1 Laufzeitverhalten der einzelnen Komponenten . . . 127
4.3.2 Evaluation Nächste-Nachbar-Suche . . . 132
4.3.3 Speicherbedarf . . . 135
4.3.4 Datenkompression . . . 136
4.3.5 Fazit Laufzeit, Speicherverbrauch, Datenkompression . . . 140
4.4 Vergleich mit anderen Verfahren . . . 141
4.4.1 Alternative Meshing-Verfahren . . . 141

4.4.2 Meshoptimierung . . . 148
4.5 Zusammenfassung . . . 151

5 Einsatzbeispiele für 3D-Polygonkarten 153
5.1 Umgebungsrekonstruktionen in Simulatoren . . . 153
5.2 6D-Pose-Tracking mittels einer PMD-Kamera . . . 156
5.2.1 Problemstellung . . . 156
5.2.2 Das Posetracking-Verfahren . . . 157
5.2.3 Evaluation . . . 162
5.2.4 Zusammenfassung . . . 165
5.3 Semantische Interpretation . . . 166
5.3.1 Motivation . . . 166
5.3.2 Modellbasierte Objekterkennung . . . 167
5.3.3 Erzeugung von Objekthypothesen aus der Ontologie . . . 169
5.3.4 Hypothesenverifikation . . . 171
5.3.5 Experimente . . . 172
5.3.6 Objekterkennung in Kinect-Punktwolken . . . 178
5.3.7 Zusammenfassung . . . 179

6 Zusammenfassung und Ausblick 181

A Parameter der Rekonstruktionssoftware 185

B SICK-Laserscans und Kinect-Frames 187

Abbildungsverzeichnis

2.1 Hochaufgelöster Laserscan der Fassade des Osnabrücker Schlosses 7
2.2 Die Roboter Kurt3D und Kurt360 8
2.3 Scans von rotierenden SICK LMS 200 Laserscannern 9
2.4 Beispieldaten einer Kinect-Kamera 9
2.5 Punktwolken einer PMD-amera 10
2.6 Marker zur Registrierung von Laserscans 12
2.7 Automatische Zuordnung von Bildmerkmalen 14
2.8 Sequenzielle Definition von Vertices 16
2.9 Beispiel für einen indizierten Vertexbuffer 17
2.10 Kantenliste 17
2.11 Kanten, Vertices und Faces in der Winged-Edge-Datenstruktur 18
2.12 Verzeigerung der Kanten und Faces in der Half-Edge-Datenstruktur 19
2.13 Beispiel für eine Marching-Squares-Approximation 22
2.14 Grundmuster beim Marching-Squares-Algorithmus 22
2.15 Interpolation der Kantenschnittpunkte 23
2.16 Grundmuster des Marching-Cubes-Algorithmus 24
2.17 Prinzip der Rekonstruktion ohne Isoflächen-Interpolation 25
2.18 Beispielrekonstruktion ohne Interpolation 25
2.19 Diskretisierungsfehler bei nicht ausgerichteten Ebenen 26
2.20 Prinzip der Vertexinterpolation 26
2.21 Konstruktion der Distanzfunktion nach Hoppe 27
2.22 Beispiel einer Rekonstruktion mit Kinect-Fusion. 28
2.23 Veranschaulichung der MLS-Rekonstruktion 29
2.24 Veranschaulichung einer MLS-Projektion 30
2.25 Mehrdeutige Musterkonfiguration beim Marching-Cubes-Algorithmus 32
2.26 Zerlegung von kubischen Gitterzellen in 5 oder 6 Tetraeder 33
2.27 Die 7 möglichen Approximationsmuster beim Marching-Tetrahedrons-Verfahren . 34
2.28 Approximation von scharfen Kanten und Features 35
2.29 Interpolation der Vertices mit Kobbelts Distanzfunktion und Knickpunktfindung 35
2.30 Approximation von scharfen Kanten mit Extended Marching Cubes 36
2.31 Beispiel für ein α-Shape 36
2.32 Oberflächenrekonstruktion mit Growing Cell Struktures 37
2.33 Meshoptimierung durch Edge-Collapsing 39

2.34 Verschiedene Verfahren zur Berechnung von Flächennormalen 42
2.35 Ausrichtung von Normalen zu einem Referenzpunkt 42

3.1 Architektur der Rekonstruktionssoftware 50
3.2 UML-Klassendiagramm der IO-Bliobliothek 52
3.3 Aufbau der Rekonstruktionsbibliothek 54
3.4 UML-Diagramm zur Normalenschätzung und Distanzfunktion 55
3.5 UML-Klassendiagramm zur Polygonalisierung 57
3.6 Einfluss der Normalenqualität auf das Rekonstruktionsergebnis 59
3.7 Beispiel einer ungleichmäßigen Punkteverteilung in einem Laserscan 62
3.8 Beispiel für die Fluktuation von Normalen aufgrund zu kleiner k-Nachbarschaft . 63
3.9 Bounding-Box-Kriterium zur Erkennung ungünstiger Punktkonfigurationen . . . 64
3.10 Adaptive Normalenschätzung und Einfluss auf die Rekonstruktion 64
3.11 Suche nach bereits vorhandenen Vertices 66
3.12 Visualisierung der Gitterstruktur 67
3.13 Einfluss des Parameters k_d auf die Rekonstruktion 69
3.14 Vertexkorrespondenzen bei der Tetraederzerlegung 71
3.15 Vergleich Marching Cubes und Marching Tetraedrons 72
3.16 Approximationsmuster für Extended Marching Cubes 73
3.17 Approximation scharfer Konturen durch Edge-Flipping 74
3.18 Prinzip von Planar Marching Cubes 75
3.19 Rekonstruktion mit Planar Marching Cubes 75
3.20 Mesherstellung und Optimierung in LVR 76
3.21 Erzeugung eines Halbkantennetzes 78
3.22 Automatische Entfernung frei schwebender Artefakte 79
3.23 Kleine Regionen als Artefakte in Meshes 81
3.24 Schließen von Löchern durch Edge-Collapsing 82
3.25 Löcherschließen über Kanten 83
3.26 Problematische Topologien beim Edge-Collapsing 84
3.27 Ergebnisse des Region-Growing basierten Clusterings 87
3.28 Iterativer Ausgleich für ebene Cluster 88
3.29 Ergebniss einer Schnittkantenoptimierung 90
3.30 Polygonoptimierung durch iteratives Entfernen von Vertices 91
3.31 Polygonoptimierung nach Reumann-Witkam 92
3.32 Polygonoptimierung nach Lang 92
3.33 Polygonoptimierung nach Douglas-Peucker 93
3.34 Polygonklassen 94
3.35 Ear-Cutting 94
3.36 Einbinden von Löchern in Polygonzüge 95
3.37 Triangulation nach Seidel 96
3.38 Triangulation monotoner Polygone 97
3.39 Winding Numbers 98
3.40 Typen von Dreiecks-Listen in OpenGL 98
3.41 Entstehung neuer Vertices wärend der Triangulation 99

3.42 Neutriangulation der Konturen mit OpenGL 100
3.43 Klassifizierung der Ebenen nach Normalenrichtung 101
3.44 Randwiederholungsfunktionen beim Texture-Mapping 102
3.45 Perspektivenverzerrungen beim Texture-Mapping 103
3.46 Berechnung optimaler Texturen für planare Polygone 104
3.47 Beispiel einer Rekonstruktion mit Textur 105
3.48 Automatisch erzeugte Texturen 106

4.1 Ausschnitte aus Referenzdatensätzen 110
4.2 Vergleich von Punktwolke und Rekonstruktion in CloudCompare 112
4.3 Vermessung einer Büroumgebnung mit Kinect und Rotationseinheit 115
4.4 Rekonstruktionen aus Kinect- und Rotationseinheitsdaten 116
4.5 Datensätze zur Evaluation der Texturierung 118
4.6 Texturierte Rekonstruktion des Bürodatenssatzes 119
4.7 Detailausschnitte aus der Bürorekonstruktion 120
4.8 Beispiele für Texturen im Büroscan 121
4.9 Texturierung des Schloss-Datensatzes 123
4.10 Texturierung des Kinect-Datensatzes 124
4.11 Einfluss der Pixelgröße auf die Texturqualität 124
4.12 Vergleich der Renderingqualität bei Halbierung der Auflösung 125
4.13 Weitere Anwendungen für Texturen bei der Rekonstruktion 126
4.14 Laufzeit der einzelnen Schritte der Rekonstruktion 129
4.15 Laufzeit der Verfahren zur Meshoptimierung 130
4.16 Laufzeiten der implementierten Marching-Cubes-Varianten 131
4.17 Rekonstruktion eines SICK-Laserscans bei verschiedenen Auflösungen 131
4.18 Laufzeitverhalten verschiedener knn-Suchbibliotheken bei variation der Punktezahl 132
4.19 Parallelisierbarkeit der knn-Suche 133
4.20 Laufzeitanalyse verschiedener knn-Bibliotheken bei Variation der Nachbarzahl . . 134
4.21 Speicherbedarf der Rekonstruktion 135
4.22 Datenkompression bei Variation der Voxelgröße 136
4.23 Datenkompression bei Variation des Normalenkriteriums 138
4.24 Datenkompression und Fehler bei der Konturoptimierung 139
4.25 GCS-Meshing und LVR-Optimierung an einem SICK-Laserscan 142
4.26 GCS-Meshing und LVR-Optimierung am Beispiel des Kirchenscans 143
4.27 Poisson-Rekonstruktion mit Meshlab 144
4.28 Ball-Pivot-Rekontruktion mit Meshlab 145
4.29 Kinect-Fusion und LVR-Optimierung 146
4.30 Rekonstruktion mit Power Crust und Alpha-Shapes 147
4.31 Abstandsfehler und Laufzeit bei Variation der Kompression beim Edge-Collapsing 149
4.32 Vergleich der Triangulation nach Edge-Collapsing und LVR-Optimierung 150

5.1 Rendering von Umgebungsmodellen in Gazebo. 154
5.2 Simulation eines 2D-Laserscans in Gazebo 155
5.3 Ablaufdiagramm zum 6D-Posetracking 157

5.4 Filterung von PMD-Daten 158
5.5 Lochkameramodell der PMD-Kamera 159
5.6 Kalibrierung der PMD Kamerapose 160
5.7 Matching von simulierten und gemessenen PMD-Daten 160
5.8 PMD-Tracking in der Ebene 162
5.9 Evaluation der PMD-Höhenschätzung 164
5.10 Aufbau und Ergebnisse des Rampenexperiments 165
5.11 Simulation einer Kinect-Punktwolke in LVR-Rekonstruktionen 166
5.12 Integration verschiedener Objekterkennungsverfahren 169
5.13 Ein Auschnitt aus der Ontologie zur Möbelmodellierung 170
5.14 3D-Punktwolke und geclusterte Rekonstruktion einer Büroumgebung 172
5.15 Ergebnisse des Matchings zweier Tischmodelle mittels ICP 174
5.16 Segmentierung einer Tischplatte nach dem Hinzufügen von Objekten 176
5.17 Registrierte Punktwolke der Büroumgebung mit erkannten Möbeln 177
5.18 Modellersetzung in Kinect-Daten 180

Tabellenverzeichnis

2.1 Übersicht über die Komplexitäten topologischer Abfragen in Polygonnetzen . . . 20

3.1 Übersicht über die Buffer-Klassen 53
3.2 Kodierung gemeinsamer Eckpositionen im Rekonstruktionsgitter 67
3.3 Ausschnitt aus der Marching-Cubes-Triangulationstabelle 70

4.1 Abstandsfehler zwischen Eingangsdaten und Rekonstruktion 113
4.2 Rekonstruktionen von Kinect- und Rotationseinheitsdaten 117
4.3 Kenngrößen initialer und optimierter Rekonstruktionen 118
4.4 Kenngrößen einer Textur des Schlossscans 126
4.5 Kennzahlen der Datensätze zur Laufzeitanalyse 128
4.6 Relative Beschleunigung der Normalenschätzung 134
4.7 Ergebnisse der Evaluation von GCS-Meshing 141
4.8 Vergleich Meshoptimierung in LVR mit Edge-Collapsing 148

5.1 Rotation des Tischmodells und berechnete Orientierung mittels PCA. 173
5.2 Rekonstruierte Flächen beim Tischexperiment 175
5.3 Erkennungsraten der Referenzmodelle in den Kinect-Frames 179

Kapitel 1

Einleitung

In den vergangenen Jahren hat in der mobilen Robotik die Verwendung von Sensoren, die in der Lage sind, die Umgebung eines Roboters dreidimensional zu erfassen, stark zugenommen. Neben 3D-Laserscannern werden im Indoor-Bereich zunehmend auch 3D-Kameras eingesetzt. Diese Entwicklung wurde insbesondere durch die Einführung der Microsoft Kinect-Kamera beschleunigt. Solche 3D-Sensoren liefern eine diskrete Abtastung der Umgebung, eine so genannte Punktwolke. Die Auflösungen der gelieferten Punktwolken variieren je nach Sensor stark: Von einigen hunderttausend Punkten pro 3D-Scan bei der Verwendung von nickenden oder rotierenden 2D-Laserscannern bis zu mehreren Zig-Millionen Punkten bei hochaufgelösten terrestrischen Scannern. 3D-Kameras liefern in der Regel vergleichsweise gering aufgelöste Punktwolken (Kinect: 640×480 Tiefenpunkte), diese dafür aber mit einer hohen Datenrate, typischerweise 25-30 Bilder pro Sekunde. Durch konsistente Integration mehrerer Punktwolken, die aus geeigneten unterschiedlichen Posen aufgenommen wurden, kann man auch größere Szenarien komplett erfassen.

Mobile Roboter benötigen für alle ihre Aktionen eine an ihre Hardware angepasste Umgebungsrepräsentation, eine Karte. Punktwolken sind jedoch als Karten für mobile Roboter keine gute Umgebungsrepräsentation. Neben der zu verarbeitenden Datenmenge ist vor allem die Diskretisierung problematisch. Egal, wie hoch die Auflösung des Scanners ist oder wie viele Scans man aus unterschiedlichen Posen aufnimmt, man erhält prinzipbedingt keine kontinuierliche Oberflächendarstellung. Zudem ist es schwierig, in einer solchen Darstellung zusätzliche Informationen zu hinterlegen. Man kann Attribute nur mit einzelnen Raumpunkten oder Gruppen von Punkten assoziieren.

Weiterhin werden viele Probleme der mobilen Robotik, wie z.B. Lokalisierung, überwiegend auf Grundlage von zweidimensionalen Umgebungsrepräsentationen wie Raster- oder 2D-Polygonkarten gelöst. Diese lassen sich zwar durch Projektion in geeignete Ebenen aus den Punktwolkendaten erstellen, jedoch muss für jeden Sensor, der sich nicht mit anderen Sensoren eine Blickebene teilt, eine eigene Karte erstellt werden. Wünschenswert wäre es, eine konsistente und gleichzeitig kompakte Repräsentation der Umgebung zu haben, die mit beliebigen Sensorkonfigurationen verwendet werden kann.

Ein Ansatz, diese Probleme zu lösen, ist die Generierung dreidimensionaler Polygonnetze aus Punktwolken. Eine solche Darstellung bietet viele Vorteile gegenüber zweidimensionalen Karten. So wird durch die Polygondarstellung die Diskretisierung aufgelöst. Zudem lassen sich Polygonnetze in effizienten Baumstrukturen, z.B. BSD-Bäumen, ablegen, die es ermöglichen, mittels Raytracing effizient Sensormodelle für verschiedene Sensoren (Laserscanner, 3D-Kameras) zu generieren. Auf diese Art und Weise kann man eine einmal generierte Karte leicht für verschiedene Sensorkonfigurationen wiederverwenden. Neben der Geometrie lassen sich in Polygonnetzen effizient weitere Informationen hinterlegen. So können durch die Definition von Texturen für die Polygone leicht Farbinformationen über die aufgenommenen Oberflächen abgelegt werden. Solche Informationen lassen sich z.B. nutzen, um mit Hilfe von Kameras visuelle Features wiederzuerkennen.

1.1 Wissenschaftlicher Beitrag

Die Oberflächenrekonstruktion von 3D Punktwolken ist ein aktives Forschungsfeld im Bereich der Computergrafik. Dort wurden bereits viele Verfahren entwickelt, die allerdings in der Regel für spezifische Arten von Modellen, oftmals Scans von in sich geschlossenen Objekten wie Skulpturen oder Maschinenteilen optimiert wurden. Für die Anwendung in der Robotik fehlte bis dato eine Software, mit der sich automatisiert Polygonmodelle von größeren und unstruktierten Umgebungen erstellen lassen. Der wissenschaftliche Beitrag dieser Arbeit besteht darin, dass geeignete Methoden aus dem Bereich der Computergrafik in den hier vorliegenden Anwendungskontext transferiert und geeignet erweitert wurden, um so ein neuartiges Verfahren zum automatischen Erstellen von robotisch nutzbaren Polygonkarten zu erhalten. Dazu wurde zunächst evaluiert, welche Verfahren der Computergrafik sich auf den vorliegenden Anwendungsfall übertragen lassen. Basierend auf den gewonnenen Erkenntnissen wurde eine frei verfügbare Software entwickelt, die darauf ausgelegt ist, mit modernen 3D-Sensoren wie Laserscannern und Kinect-Kameras zusammenzuarbeiten. Dazu war es notwendig, die speziellen Charakteristika wie Auflösung, Punktdichte und Messrauschen dieser Sensoren zu berücksichtigen und entsprechend optimierte Rekonstruktionsverfahren zu entwickeln. Es wurde anhand von Anwendungsbeispielen gezeigt, dass sich die mit dem neuen Verfahren automatisch generierten Karten in der Robotik für verschiedene Anwendungen benutzen lassen.

1.2 Zielsetzungen

Dreidimensionale Polygonkarten werden in der mobilen Robotik bis jetzt wenig genutzt, da die manuelle Erstellung solcher Karten sehr zeitaufwändig ist. Vorhandene Software zur Flächenrekonstruktion stammt überwiegend aus dem Bereich der Computergrafik und wurde unter anderen Gesichtspunkten, z.B. Dreiecksqualität der erzeugten Netze, entwickelt. Vor allem gibt es dort keine zeitlichen Restriktionen. Für robotiknahe Anwendungen ist es essentiell wichtig, dass die benötigten Umgebungskarten schnell und möglichst auf mobiler Hardware (Laptops) direkt am

Einsatzort erstellt werden können. Dazu soll eine Software entwickelt werden, die die folgenden Randbedingungen erfüllt:

Genauigkeit. Umgebungskarten sind die Grundlage für alle Aktionen eines Roboters. Dementsprechend muss die Datengrundlage, auf der seine Aktionen geplant werden, möglichst exakt der Realität entsprechen.

Performanz. Die Polygonkarten sollen möglichst zeitnah aus den Punktwolken erstellt werden. Die Laufzeit der implementierten Algorithmen spielt dementsprechend eine entscheidende Rolle. Da nahezu alle modernen Prozessoren über mehrere Prozessorkerne verfügen, soll ein besonderes Augenmerk auf die Parellelisierbarkeit der verwendeten Verfahren gelegt werden.

Datenkompression. Die erstellten Daten sollen unter Beibehaltung der Umgebungsgeometrie möglichst wenige Polygone enthalten.

Speichereffizienz. Die Software soll auf handelsüblichen Laptops lauffähig sein. Da diese in der Regel über weniger Arbeitsspeicher als Desktop-Rechner verfügen, müssen die verwendeten Algorithmen möglichst speichereffizient sein.

Die erstellte Software soll gemäß den oben genannten Kriterien mit Referenzdaten und wenn möglich mit bereits existierenden Softwarepaketen verglichen werden. Darüber hinaus sollen die praktischen Vorteile dreidimensionaler Polygonkarten beispielhaft an einigen Demonstratoren gezeigt werden.

1.3 Aufbau der Arbeit

Die vorliegende Arbeit gliedert sich wie folgt:

Kapitel 1 Dieses Kapitel liefert eine Einführung in das Thema der Arbeit und legt den wissenschaftlichen Beitrag dar.

Kapitel 2 Das zweite Kapitel schildert den aktuellen Stand der Forschung bezüglich Oberflächenrekonstruktion aus 3D-Punktwolken. Dazu werden auch kurz aktuelle 3D-Sensoren und Verfahren zur Registrierung von Punktwolken vorgestellt.

Kapitel 3 Ausgehend von den in Kapitel 2 vorgestellten Methoden werden im dritten Kapitel dieser Arbeit die Anforderungen an die zu erstellende Software konkretisiert und die Verfahren präsentiert, die zur Erzeugung von Polygonkarten implementiert wurden. Besonderes Augenmerk liegt auf der laufzeit- und speichereffizienten Implementierung der Algorithmen. Parallelisierung spielt dabei eine wichtige Rolle.

Kapitel 4 Der Schilderung der Implementierung schließt sich eine Evaluation der in die Software integrierten Verfahren bezüglich der in Kapitel 3 genannten Bewertungskritieren an. Falls vorhanden, werden sie mit anderer verfügbarer Software verglichen.

Kapitel 5 Das fünfte Kapitel demonstriert die Nutzbarkeit der erzeugten Polygonkarten anhand von drei Anwendungsbeispielen: Nutzung der Rekonstruktionen als Umgebungsrepräsentationen in Simulatoren, 6D-Posetracking mit einer 3D-Kamera und semantische Szeneninterpretation.

Kapitel 6 Das letzte Kapitel fasst die Ergebnisse der Arbeit zu einem Fazit zusammen und zeigt noch offene Forschungsfragen auf.

Kapitel 2

Oberflächenrekonstruktion aus 3D-Punktwolken

Das Problem, aus einer Menge gesampleter Oberflächenpunkte eine mathematische Beschreibung der originalen Oberfläche zu generieren, kann auf vielerlei Art und Weise betrachtet werden. Eine Betrachtungsweise ist z.B. Model Fitting: Man legt als Flächenbeschreibung ein parametrierbares mathematisches Flächenmodell, z.B. eine mehrdimensionale Polynomfunktion oder B-Splines, zugrunde, und versucht die Parameter so einzustellen, dass die Fehlersumme der quadratischen Abstände der gemessenen Punkte zum Flächenmodell minimal wird (sog. Least-Squares-Fits [135]). Solche Parametersätze lassen sich mit Hilfe der linearen Algebra (z.B. durch SVD-Zerlegungen [69]) oder randomisierte Verfahren (z.B. RANSAC-Varianten [54]) algorithmisch gut bestimmen [62]. Solange die gewählten Modelle gut zu den gesampleten Daten passen, werden mit diesem Ansatz akkurate Ergebnisse erzielt.

Allgemeine Oberflächen lassen sich in der Regel aber nicht durch ein einfaches einzelnes Modell beschreiben. Durch räumliche Gewichtungen und Überlagerung von lokalen Modellen ("Moving Least Squares")kann man sehr gute Approximationen erreichen [6, 74, 106, 149], jedoch sind solche Modelle äußerst komplex, benötigen zur Erstellung viel Rechenzeit und lassen sich schlecht visualisieren. Als Alternative zu diesen modellbasierten Verfahren haben sich in der Computergrafik und algorithmischen Geometrie daher polygonale Netze als eine Standardrepräsentation durchgesetzt. In einem polygonalen Netz werden Oberflächen lokal durch ebene Polygone einer festen Anzahl an Knoten, Vertices genannt, repräsentiert. In der Praxis werden dazu Dreiecke ("Dreiecksnetze") oder Vierecke ("Quads") verwendet. Zwar sind die Polygone in den Netzen lokal eben, mit einer ausreichend großen Anzahl kleiner Netzelemente lassen sich aber auch gekrümmte Flächen sehr gut approximieren. Zudem ist die Darstellung äußerst kompakt, da viele Flächen sich Knoten teilen und diese daher nicht doppelt abgelegt werden müssen. Es entsteht eine hohe Datenkompression, vor allem, wenn die modellierten Umgebungen viele ebene Anteile haben. In Kombination mit Texturen und ausgefeilten Beleuchtungsalgorithmen sind Polygonnetze derzeit die Standarddatenstrukturen für 3D-Modelle in der Computergrafik.

Polygonnetze wurden schon sehr früh zur Rekonstruktion von Oberflächen aus 3D-Punktwolkendaten verwendet. Bekanntestes Beispiel ist wohl das "Digital Michelangelo"-Projekt der Stanford University [107]. Im Rahmen dieses Projektes wurden Skulpturen des Bildhauers Michelangelo im Museum in Florenz mit Hilfe von Streifenlichtprojektoren hochauflösend dreidimensional vermessen. Auf Basis dieser Daten wurden dann hoch aufgelöste Polygonnetze der gescannten Skulpturen generiert [19], deren Auflösung hoch genug war, die Meißelposition des Bildhauers an den Plastiken wiederzugeben [17].

Neben der hohen Datenkompression sind Polygonnetze auch für algorithmische Zwecke interessant. So lassen sich durch geeignete Baumstrukturen effizient Schnittpunktberechnungen zwischen Strahlen und Ebenen realisieren [178, 187, 207]. Gerade diese Eigenschaft macht sie als dreidimensionale Kartenrepräsentation für Anwendungen in der Robotik interessant. Durch Strahlenverfolgung ("Raytracing") lassen sich auf diese Weise durch ein geeignetes Sensormodell aus den Karten Sensordatenhypothesen für 2D- und 3D-Laserscanner an beliebigen Posen generieren. Darüber hinaus verwenden die bekannten Simulatoren in der Robotik (Gazebo, UsarSIM) polygonale Modelle für die Physiksimulation.

In den folgenden Abschnitten soll der Stand der Forschung zum Thema "Automatische Erstellung Polygonaler Netze aus 3D-Punktwolken" dargestellt werden. Dazu werden zunächst gebräuchliche Sensoren zur Erfassung von 3D-Oberflächen vorgestellt. Danach wird kurz auf das Problem der Registrierung einzelner Punktwolken zu einem konsistenten Gesamtmodell eingegangen. Dem folgt eine Übersicht über gängige Datenstrukturen zur Repräsentation polygonaler Netze. Kapitel 2.3 präsentiert Algorithmen zur automatischen Erstellung von polygonalen Netzen aus 3D-Punktwolken. Kapitel 2.6 behandelt ein weiteres Thema, das in diesem Kontext wichtig ist, und zwar die Interpolation von Normalen an die Datenpunkte.

2.1 Sensoren zur Erstellung dreidimensionaler Punktwolken

Im Folgenden sollen kurz die derzeit in der mobilen Robotik gängigen Sensoren zur Gewinnung von 3D-Punktwolken und deren Charakteristika vorgestellt werden. Dabei wird im Wesentlichen grob zwischen Laserscannern und 3D-Kameras unterschieden.

2.1.1 3D-Laserscanner

Laserscanner sind aktive Sensoren, die Umgebungsgeometrie mit Hilfe eines Laserstrahls erfassen. Das Messprinzip beruht darauf, in einer bestimmten Richtung kohärentes Licht einer bestimmten Wellenlänge auszusenden. Trifft dieses Signal auf eine Oberfläche, wird es reflektiert. Aus der Analyse dieses Signals kann der Abstand des Objektes bestimmt werden. Durch die starke Fokussierung des Laserstrahls kann dabei eine hohe Ortsauflösung auch bei Messungen über große Distanzen erzielt werden. 2D-Laserscanner besitzen einen Drehspiegel, der den Laserstrahl in die zu messende Richtung ablenkt. Durch Rotation um eine weitere Achse kann ein dreidimensionales Bild der Umgebung aufgenommen werden.

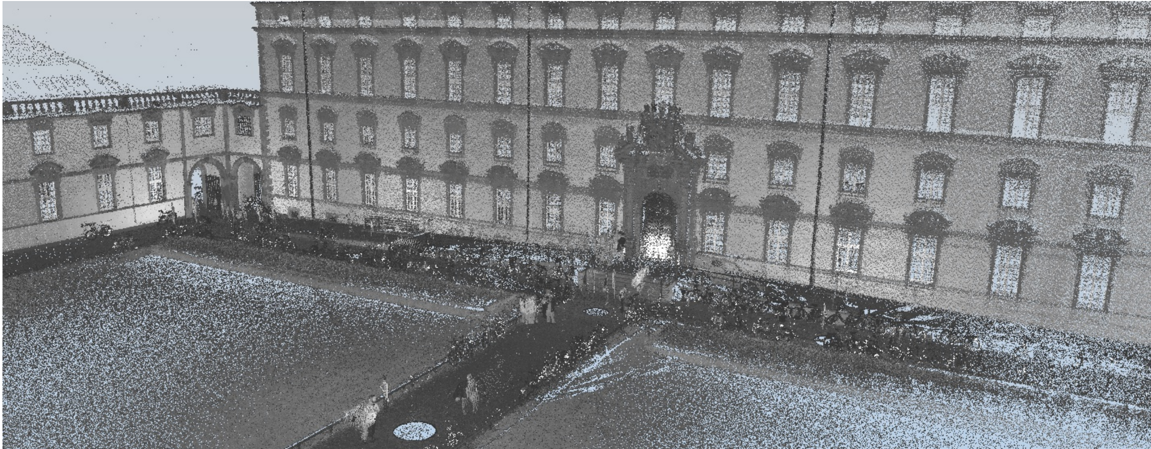


Abbildung 2.1: Hochaufgelöster Laserscan der Fassade des Osnabrücker Schlosses, aufgenommen mit einem Leica HDS 6000 Laserscanner. Der Originaldatensatz besteht aus insgesamt ca. 250 Mio. Punkten, aufgenommen von 12 Standorten. Für dieses Bild wurde der Originaldatensatz auf 15 Mio. Punkte reduziert, um noch gerendert werden zu können.

Laserscanner arbeiten entweder gepulst (PW-Systeme) oder mit kontinuierlichen Signalen (CW-Systeme). Bei der gepulsten Methode werden kurze Laserblitze ausgesandt und die Laufzeit Δt des Lichts gemessen. Aus der Lichtgeschwindigkeit c kann dann direkt die Entfernung des Objektes Δs gemessen werden. Bei der anderen Variante werden kontinuierliche Lichtsignale ausgesandt. Die Entfernung wird aus der Phasenverschiebung zwischen ausgesendetem und empfangenem Licht bestimmt (nach [195]).

In den vergangenen Jahren hat sich die Technologie im Bereich der Laserscanner, insbesondere im Profimarkt, enorm weiter entwickelt. Moderne Geräte aus dem Bereich des terrestrischen Laserscannings besitzen eine hohe Winkelauflösung und sind in der Lage, mit einem einzelnen Scan mehrere hundert Millionen Punkte aufzunehmen. Die Scandauer liegt dabei im Bereich von einigen Minuten. Die Preise solcher Geräte bewegen sich momentan im Bereich von 80.000 bis 100.000 Euro. Abbildung 2.1 zeigt einen hochauflösenden Laserscan des Osnabrücker Schlosses, der mit einem Leica HDS 6000 Laserscanner aufgenommen wurde.

Solche Geräte kommen aufgrund der hohen Kosten und weiteren Faktoren wie langer Scandauer, hohem Gewicht und Stromverbrauch im Bereich der mobilen Robotik selten zum Einsatz. Dort werden zur 3D-Umgebungserfassung in der Regel 2D Laserscanner mit entsprechenden Mechaniken gedreht oder genickt. Abbildung 2.2 zeigt zwei 3D-Laserscanner, die in der Arbeitsgruppe Wissensbasierte Systeme in Osnabrück auf Kurt-Robotern eingesetzt werden. Die rotierende Variante dieses Scanners kann auch während der Fahrt Umgebungsscans aufnehmen, beim anderen Aufbau muss der Roboter zum Aufnehmen eines Scans stehen bleiben. Die dort zum Einsatz kommenden Scanner der Firma SICK sind mit einer Winkelauflösung von 1° und einem Öffnungswinkel von 180° bei einer Messvarianz von 2 cm in der Entfernung nur in der Lage, ein vergleichsweise grob aufgelöstes und verrauschtes Umgebungsbild aufzunehmen (vgl. Abbildung 2.3). Die Herausforderung besteht darin, Rekonstruktionsverfahren zu entwickeln, die auch fähig sind, aus solchen Daten brauchbare Umgebungskarten zu erzeugen.

Abbildung 2.2: Die Roboter Kurt3D mit nickendem SICK LMS 200 Laserscanner (links) und Kurt360 mit Rotationseinheit (rechts), die als Testplattform für das Rekonstruktionsverfahren dienen sollen. Die maximale Auflösung des verwendeten 2D-Laserscanners beträgt 721 Punkte pro Scan bei einem Öffnungswinkel von 180°. Der Winkelbereich des nickenden Scanners beträgt maximal 120° mit einer Winkelauflösung von bis zu 0.5°. Die Rotationseinheit kann kontinuierlich Daten aufnehmen, der minimale Drehwinkel zwischen zwei 2D-Scanns beträgt 0.035°.

2.1.2 3D-Kameras

Microsoft Kinect

Neben Laserscannern spielen in der mobilen Robotik 3D-Kameras eine zunehmend größere Rolle. Dieser Trend wurde insbesondere durch die Einführung der Microsoft Kinect Kamera und der schnellen Entwicklung von Open-Source-Bibliotheken zur Ansteuerung der Kinect-Hardware [78, 89] und deren Integration in Robotik-Middleware wie ROS [155] oder Player [36] verstärkt. Diese Kamera liefert farbige Tiefenbilder (RGB-D Bilder) mit einer Auslösung von 640 × 480 Pixeln bei 30 Bildern pro Sekunde. RGB-D bedeutet dabei, dass zu jedem Bildpunkt ein Tiefenwert für die Entfernung der gemessenen Oberfläche gegeben wird.

Die Kinect-Kamera basiert auf einem aktiven Projektionsverfahren. Ein Projektor projiziert im nicht sichtbaren Infrarotbereich Zufallsmuster aus Refrenzpunkten helleren Punkten in der Mitte des Sichtbereichs in die Szene. Eine parallel montierte Infrarotkamera mit dem Basisabstand b zum Projektor erfasst das durch die Objekte im Raum verzerrte Referenzmuster. Um die Entfernungen bestimmen zu können, ist für einen festgelegten Tiefenwert ein Referenzmuster in der Kamera gespeichert. Zur Bestimmung der realen Tiefe d eines Objektpunktes wird in einem rechteckigen Bereich um einen Pixel des empfangenen Infrarotbildes die beste Übereinstimmung mit dem Referenzmuster gesucht. Der Versatz zwischen detektiertem Muster und Referenzmuster (in Pixeln) wird als Deviation Δd bezeichnet. Für eine kalibrierte Kamera mit Brennweite f lässt sich die Tiefe eines jedes Pixels mit Hilfe der folgenden Gleichung bestimmen [59]:

$$d = \frac{b * f}{\Delta d}$$

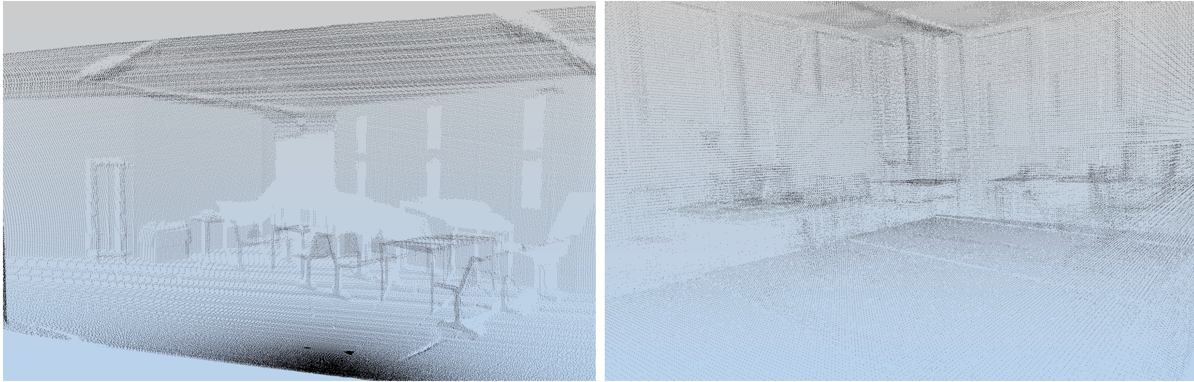


Abbildung 2.3: Zwei Scans, die mit geschwenkten SICK LMS 200 Laserscannern aufgenommen wurden. Der linke wurde mit dem nickenden Laserscanner des Kurt3D-Roboters aufgenommen, der rechte mit der Rotationseinheit des Kurt360 Roboters. Die Scans des nickenden Sensors weisen eine zu den Rändern hin abnehmende Punktdichte aus, während die Punkte des Rotationseinheitsscans gleichmäßiger verteilt sind, aber eine insgesamt geringere Dichte haben.

Abbildung 2.4: Daten der Kinect-Kamera. Links: 2D-Bild der beobachteten Szene, Mitte: Tiefenbild in Falschfarben, rechts: 3D-Ansicht der erzeugten Punktwolke. Man erkennt deutlich das Rauschen und die Diskretisierung in den Messdaten.

Dieses Verfahren funktioniert nur in Umgebungen, in denen das ausgesendete Muster nicht durch andere Lichtquellen (z.B. Sonnenlicht) überstrahlt wird. Daher werden diese Kameras in der Robotik vorwiegend in Indoor-Anwendungen eingesetzt. Problematisch ist zudem die Erfassung von transparenten oder spiegelnden Oberflächen, da das Referenzmuster auf solchen Flächen nicht erfasst werden kann. Zudem weisen die Entfernungswerte ein gewisses Grundrauschen auf und die Werkskalibrierung der Kamera ist nicht immer optimal, so dass eigentlich gerade Flächen wie Wände in den Daten eine gewisse Krümmung aufweisen können. Durch eine Nachkalibrierung, z.B. mit einem Schachbrettmuster nach Zhang [212], können solche Effekte teilweise kompensiert werden. Eine systematische Analyse zum Sensormodell findet sich in [95]. Abbildung 2.4 zeigt beispielhaft eine farbige Punktwolke, die mit einer Kinect-Kamera aufgenommen wurde.

Neben diesen Nachteilen bieten die Daten der Kinect-Kamera aber auch diverse Vorteile: So werden Farbinformationen für die Punktwolken durch die integrierte Farbkamera gleich mitgeliefert. Zudem sorgt die hohe Bildwiederholungsrate dafür, dass sich Trackingverfahren und Registrierungsverfahren gut implementieren lassen, da der Versatz zwischen zwei Bildern gering ist und

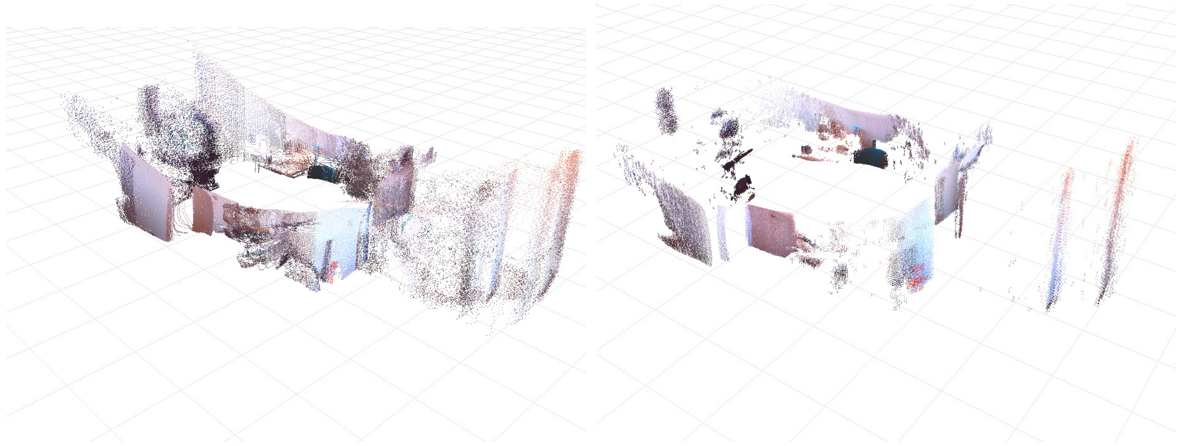


Abbildung 2.5: Umgebungsmodelle, die mit einer gegen eine Webcam kalibrierten PMD-Kamera aufgenommen wurden. Die gezeigten Punktwolken bestehen aus mehreren 3D-Bildern, die mittels ICP registriert wurden. Im linken Bild erkennt man deutlich die für diesen Sensor typischen "Geisterpunkte" an Tiefensprüngen. Im rechten Bild wurde diese mit dem von Huhle [85] vorgestellten Verfahren gefiltert. Die Filterung ist in der Lage, solche Punkte zu entfernen, allerdings werden im Bereich der Wände auch Punkte entfernt, die starkes lokales Rauschen aufweisen, aber keine Kantenartefakte sind.

Korrespondenzen zwischen den Bildern so leichter gefunden werden können [79, 182]. Auf diese Art und Weise lassen sich mit Kinect-Kameras auch größere Szenen dreidimensional erfassen.

Time-of-Flight Kameras

Eine andere Art von 3D-Kameras stellen sogenannte Time-of-Flight (ToF) Kameras dar. Diese Sensoren arbeiten nicht mit einem Referenzmuster, sondern senden moduliertes Licht aus. Das von der Szene reflektierte Signal wird dann über eine Linse auf einen Chip mit für die entsprechende Wellenlänge empfindlichen Fotozellen geleitet ("Photonic Mixer Device", "PMD-Sensor"). Aus der Differenz zwischen ausgesendetem und empfangenen Signal kann die Entfernung des gemessenen Objektes für jeden Pixel auf dem Empfängerchip berechnet werden. Bekannte Modelle sind PMD-Kameras der Firma PMD Technologies und Swissranger.

ToF-Kameras haben in der Regel eine relativ geringe Auflösung. Selbst die aktuellen Modelle verfügen über nicht mehr als 20.000 Pixel. Der PMD Tech CamCube 3.0 hat eine Auflösung von 200×200 Pixeln [152], die SwissRanger SR4000 erreicht 146×144 Pixel [121]. Zudem weisen solche Kameras an Tiefensprüngen "Geisterpixel" zwischen dem vorderen und hinteren Objekt auf. Diese lassen sich zwar relativ gut aus den Daten filtern [118], jedoch wird dadurch die Zahl der Messpunkte weiter verringert. Abbildung 2.5 zeigt Punktwolken, die mit einer PMD-Kamera (Modell `O3D100`) aufgenommen wurden, vor und nach einer Filterung. Es existieren Arbeiten, in denen komplette Räume mit derartigen Kameras vermessen wurden [117]. Aufgrund der geringen Auflösung und des hohen Messrauschens spielen diese Sensoren bei der großflächigen Erfassung von Geometrien kaum eine Rolle und werden in der Robotik hauptsächlich zur Hindernisvermeidung eingesetzt.

2.1.3 Weitere Sensoren

Neben Laserscannern, RGB-D und 3D-Kameras gibt es es noch weitere Verfahren, um Umgebungsgeometrien dreidimensional aufzunehmen. Dazu gehören unter anderem Streifenlichtprojektoren und Stereokameras. Streifenlichtprojektoren erreichen hohe Genauigkeiten in kontrollierten Umgebungen. Stereokameras erzielen nur im Nahbereich brauchbare Ergebnisse. Zur großflächigen dreidimensionalen Vermessung sind sie daher nicht geeignet und werden hier nur der Vollständigkeit halber erwähnt.

2.1.4 Fazit 3D-Sensoren

Es gibt inzwischen eine ganze Fülle von 3D-Sensoren, mit denen sich dreidimensionale Punktwolken generieren lassen. Die Qualität der aufgenommenen Punktwolken variiert stark unter den verschiedenen Sensortypen. Terrestrische Scanner erreichen hohe Punktdichten bei hoher Genauigkeit, benötigen aber viel Zeit zum Scannen. Nickende oder rotierende 2D-Laserscanner benötigen weniger Zeit, erreichen aber nicht annähernd Punktdichte und Genauigkeit terrestrischer Laserscanner. 3D-Kameras liefern Umgebungsbilder mit einer potentiell hohen Bildfrequenz, dafür sind die gelieferten Punktwolken vergleichsweise dünn und weisen ein hohes Messrauschen, insbesondere bei Tiefensprüngen, auf.

Da die vorgestellten Sensoren in der Praxis sehr verbreitet sind, müssen diese unterschiedlichen Eigenschaften auch bei der Entwicklung eines Rekonstruktionsverfahrens berücksichtigt werden. Die implementierten Algorithmen müssen unempfindlich gegen Messrauschen sein und auch bei geringer Punktdichte akzeptable Ergebnisse erreichen. Ebenso müssen sie effizient sein, um auch hochaufgelöste Laserscans verarbeiten zu können.

2.2 Registrierung von 3D-Punktwolken

Großflächige und komplexe Umgebungen lassen sich durch einzelne Laserscans nicht erfassen. Dazu sind in den meisten Fällen mehrere Scans an unterschiedlichen Standorten erforderlich. Da jeder 3D-Sensor in seinem eigenen Koordinatensystem misst, müssen die einzelnen Teildatensätze konsistent aus dem lokalen in ein globales Koordinatensystem registriert werden. Ein exaktes Einmessen der einzelnen Scanpositionen zur Registrierung ist in der Praxis zu aufwändig. Im Folgenden werden die Standardverfahren zur Registrierung von 3D-Laserscans ohne vorherige Einmessung der Scanposition kurz vorgestellt.

2.2.1 Registrierung mit Markern

Registrierung von Laserscans mit Hilfe von speziellen Markern, auch Targets genannt, wird von den Software-Paketen der bekannten Hersteller terrestrischer Laserscanner unterstützt. Dabei werden vom Vermesser an ausgewählten Positionen Objekte mit bekannter Geometrie oder speziellen Reflektionseigenschaften in der Szene platziert. Die verwendenten Referenzobjekte sind

Abbildung 2.6: Verschiedene Marker zur Registrerung von 3D-Laserscans. Das linke Bild zeigt eine Kugel als Referenzgeometrie. Die mittleren Bilder zeigen verschiedene Klebemarker mit Referenzmustern [20, 105]. Das rechte Bild zeigt die gemessene Reflektivität der Markeroberfläche. Man kann gut erkennen, dass sich das Zentrum der Markierung deutlich vom Hintergrund absetzt [105].

herstellerspezifisch. Verbreitet sind sphärische oder zylinderförmige Referenzobjekte sowie Marker mit einer speziellen gut reflektierenden Oberfläche oder mit speziellen Referenzmustern. Eine kleine Auswahl solcher Targets zeigt Abbildung 2.6. Aufgrund ihrer speziellen Eigenschaften lassen sich die Targets in den aufgenommenen Daten leicht wiederfinden. Anschließend werden Korrespondenzen zueinander ermittelt, d.h. welche Markierungen waren in sich überlappenden Bereichen verschiedener Scans sichtbar. Die Transformation der einzelnen Scans in das globale Koordinatensystem wird dann durch Triangulation bestimmt. Die Extraktion der Markerpositionen und die Ermittlung der Korrespondenzen geschieht je nach verwendeter Software und System automatisch oder durch Benutzerinteraktion.

Mit Hilfe von Markern lassen sich Laserscans äußerst präzise registrieren [94], allerdings ist der Zeitaufwand dafür sehr groß, da sehr viel Benutzerinteraktion erforderlich ist. Zum einen müssen die Marker in der Szene an möglichst sinnvoll gewählten Positionen platziert werden. Zum anderen müssen die Korrespondenzen oft semi-manuell definiert werden. Ein weiterer Nachteil ist, dass die Marker auch in den aufgenommenen Daten auftauchen und somit die Wiedergabe verfälschen. Darüber hinaus ist das Anbringen vor Markern nicht immer möglich, z.B. um Beschädigungen an den zu vermessenden Objekten zu verhindern oder wenn die Umgebungen von Menschen nicht zu erreichen sind, z.B. in Rescue-Szenarien in der mobilen Robotik. Daher wurden vornehmlich in diesem Umfeld in den vergangenen Jahren Verfahren zur markerlosen Registrierung von 3D-Laserscans entwickelt.

2.2.2 Markerlose Registrierung

Die markerlose Registrierung von Laserscans geschieht durch automatisches Auffinden von Korrespondenzen zwischen den sich überlappenden Anteilen der einzelnen Laserscans. Dies kann durch visuelle oder geometrische Features realisiert werden. Im Gegensatz zur markerbasierten Registrierung werden dieses Features allerdings nicht künstlich in die Szene eingebracht, sondern müssen algorithmisch aus den aufgenommenen Rohdaten extrahiert werden. Wird die Pose des ersten Scans nicht in einem globalen Koordinatensystem eingemessen, werden die einzelnen Teilscans in der Regel relativ zum ersten aufgenommenen Scan registriert, d.h. die Scanposition der ersten Aufnahme legt den Ursprung des Koordinatensystems fest.

ICP

Das am weitesten verbreitete Verfahren zur markerlosen Registrierung mehrerer Laserscans ist der "Iterative Closest Points"-Algorithmus [21]. Dieses Verfahren kommt gänzlich ohne die Berechnung von Features in den einzelnen Scans aus. Der ICP-Algorithmus registriert zwei sich partiell überlappende Laserscans, indem iterativ zwischen dem ersten Scan (Modellscan M, $M \subset \mathbb{R}^3$) und einem weiteren Scan (Datenscan D, $D \subset \mathbb{R}^3$) die Punkte berechnet werden, die sich am nächsten sind. Anschließend werden Rotation $\mathbf{R}$ und Translation $\mathbf{t}$ berechnet, so dass die folgende Fehlersumme minimiert wird:

$$E(\mathbf{R}, \mathbf{t}) = \sum_{i=1}^{N_m} \sum_{j=1}^{N_d} w_{ij} \|m_i - (Rd_j + t)\|^2$$

Dabei sind $m \subset M$ und $d \subset D$ die Datenpunkte der jeweiligen Scans. Der Gewichtsfaktor w_{ij} ist 1 falls m und d nächste Nachbarpunkte sind, ansonsten ist er 0. Es wird also diejenige Rotation und Translation berechnet, durch die die gefundenen nächsten Nachbarn optimal aufeinander abgebildet werden. Die gefundene Transformation wird dann auf den Datenscan angewendet. Dadurch können neue Korrespondenzen entstehen, so dass die Korrespondenzsuche iterativ fortgeführt wird, bis keine weitere Verbesserung erreicht wird. Das Verfahren konvergiert immer, findet aber nicht zwangsläufig die global beste Lösung. ICP wird häufig in der mobilen Robotik angewendet. Die Poseschätzungen der Odometrie des Roboters werden dabei zur initialen Registrierung verwendet und dann mittels dieses Verfahrens optimiert.

Eine geschlossene Lösung des ICP-Minimierungsproblems ist effizient möglich [47]. Die Berechnung der nächsten Nachbarn zwischen den Teilscans ist allerdings aufwändig, insbesondere wenn große Datenmengen verarbeitet werden. Die naive Brute-Force-Suche hat quadratische Laufzeit. Durch den Einsatz von *kd*-Bäumen [61] kann die Korrespondenzsuche auf logarithmische Laufzeit beschleunigt werden. Speziell auf die Problemstellung - genau ein nächster Nachbar wird gesucht - angepasste und approximative Verfahren verringern die Laufzeit [2, 73, 142]. Weitere Leistungssteigerungen können durch Parallelisierung der Suche erreicht werden. Dazu sind angepasste Implementierungen des Suchverfahrens notwendig [37, 114, 127, 140]. Kleinere Datensätze können darüber hinaus auf GPUs berechnet werden [154]. Inzwischen existieren diverse freie OpenSource Implementierungen solcher Suchverfahren, die sich bezüglich ihres Laufzeitverhaltens beträchtlich unterscheiden. Ein Vergleich wird in Kapitel 4.3.2 durchgeführt. In der Praxis werden insbesondere bei sehr großen Datenmengen mit geeigneten Verfahren ausgedünnte Scans in Kombination mit *kd*-Suchverfahren verwendet.

Da ICP nicht notwendigerweise in einem globalen Optimum konvergiert, können sich Fehler durch nicht korrekte Registrierung akkumulieren. Diese können durch Verfahren zum Erkennen von Schleifen in der Folge der Scans ausgebessert werden. Die Idee dabei ist, dass die berechneten Posen von zwei Scans, die von der selben Position aufgenommen wurden, auch noch nach einer Serie von mehreren Scans identisch sein müssen. Besteht ein Versatz in der berechneten Pose, gibt dieser den globalen Fehler wieder, der bei der Registrierung der zwischenzeitlich verarbeiteten Scans akkumuliert wurde. Wird eine solche Schleife erkannt, kann der Fehler durch verschiedene

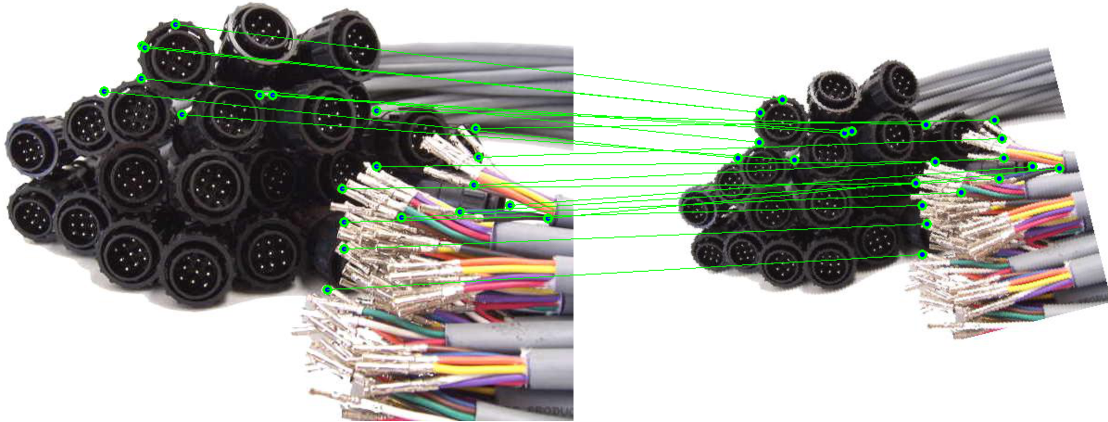


Abbildung 2.7: Automatische Zuordnung von Bildmerkmalen: Eine Szene wird aus zwei unterschiedlichen Blickwinkeln betrachtet. In den generierten Bildern werden dann Bildmerkmale gesucht und automatisch gematcht, d.h. zu jedem extrahierten Merkmal des einen Bildes wird das am besten korrespondierende Merkmal im anderen Bild gesucht (Bild entnommen aus [194]).

Verfahren auf die vorherigen Scanposen zurückgerechnet werden. Ein graphenbasiertes Verfahren für zweidimensionale Umgebungen wurde 1997 von Lu und Milos vorgestellt [112]. [24] erweitert dieses Verfahren auf sechs Freiheitsgrade. Eine weitere Heuristik zur näherungsweisen Verteilung des Fehlers wird in [90] vorgestellt.

Merkmals–basierte Verfahren

Featurebasierte Registrierungsverfahren nutzen anstelle von künstlich in die Szene eingebrachten Markierungen Methoden der Bildverarbeitung, um Merkmale aus den aufgenommenen Daten zu gewinnen. Diese Features werden dann dreidimensional in der Szene lokalisiert. Im Falle von 3D-Laserscans kann dies durch Kalibrierung des Laserscanners gegen eine Kamera oder durch das Berechnen auf einem so genannten Tiefenbild passieren. Zur Erstellung eines Tiefenbildes wird eine virtuelle Bildebene durch einen Scan gelegt. In diese Bildebene wird ein Pixelraster einer gewählten Auflösung gelegt und für jeden dieser Pixel der Abstand zum nächsten Datenpunkt, der als Grauwert für diesen Pixel kodiert wird, berechnet. Um ein besseres Kontrastverhältnis zu erreichen, kann die Grauwertberechnung auch mit Hilfe einer nichtlinearen Gewichtsfunktion geschehen [183]. Im Spezialfall eines RGB-D-Bildes, kann die Position eines Merkmals direkt durch die Pixelposition im Bild, den dazugehörigen Tiefenwert und die Kamerakalibrierung bestimmt werden. In der Regel kommen dabei SIFT- oder SURF-Merkmale und ähnliche zum Einsatz [3, 15, 111]. Die Registrierung erfolgt wiederum durch Auffinden der Korrespondenzen zwischen Bildmerkmalen der einzelnen Scans (vgl. Abbildung 2.7). Diese Zuordnungen werden nicht immer fehlerfrei gefunden, der Fehler ist aber klein, sobald ein Bild viele Merkmale enthält. Für kleine Auflösungen (z.B. auf Kinect-Daten) arbeiten solche Verfahren sehr schnell. In [182] z.B. wird ein Verfahren zum Matchen von Kinect-Frames gezeigt.

2.3 Erzeugung von Polygonnetzen mit Marching Cubes

Wie eingangs erwähnt, sind polygonale Netze in der Computergrafik die Standarddatenstrukur zur Modellierung dreidimensionaler Umgebungen. Auch die Hardware aller Grafikkarten ist direkt für die Verarbeitung solcher Datenstrukturen optimiert. Zwar lassen sich mit programmierbaren Shadereinheiten auch viele Effekte erzeugen, die nicht auf polygonalen Darstellungen basieren. So können z.B. Flüssigkeiten, Feuer und Rauch durch Partikeleffekte [53, 56] dargestellt werden, in Computerspielen und Simulationen werden Gras und einfache Pflanzen durch texturbasiertes Billboarding realisiert [125]). Flächige Modelle werden in der Regel weiterhin durch Polygone repräsentiert.

Da Grafikhardware schon früh auf die Darstellung polygonaler Netze spezialisiert war, wurde der Frage, wie man polygonale Modelle automatisiert aus verschiedenen Arten von Eingangsdaten generieren kann, schon früh nachgegangen. Das bekannteste Verfahren ist der Marching-Cubes-Algorithmus. Er wurde bereits 1987 von Lorensen und Cline vorgestellt [110] und diente ursprünglich der Visualisierung von Computertomographie-Daten. Die diesem Algorithmus zugrundeliegende Idee lässt sich aber leicht auch auf andere Daten übertragen. Im Folgenden werden die Grundlagen dieses und anderer Verfahren dargestellt.

Dazu werden zunächst die grundlegenden Datenstrukturen zur Repräsentation dreidimensionaler Polygonnetze vorgestellt. Der Marching-Cubes-Algorithmus wird in Kapitel 2.3.2 beschrieben. Ein erstes Verfahren, um diesen Algorithmus auch zur Flächenrekonstruktion aus unstrukturierten Punktwolken zu erzeugen, wurde 1992 von Hoppe veröffentlicht [83]. Es wird zusammen mit den seitdem entwickelten Erweiterungen in Kapitel 2.3.3 vorgestellt. Kapitel 2.3.4 stellt Erweiterungen des ursprünglichen Marching-Cubes-Verfahrens vor.

2.3.1 Datenstrukturen für Polygonnetze

Die grundlegenden Datenstrukturen sind in meiner Masterarbeit dargestellt [195]. Um die Darstellung in der vorliegenden Arbeit zu vervollständigen, wird sie hier in überarbeiteter Fassung wiederholt. Ein Polygonnetz ist eine endliche Menge von miteinander verbundenen planaren Polygonen, die die Oberfläche eines Objektes approximieren. Die Polygone im Netz werden als Faces oder Facetten bezeichnet. Polygone werden durch Angabe von Vertices (Eckpunkten) und Kanten (Verbindung von zwei Vertices) definiert. Üblicherweise werden die Vertices eines Polygons gegen den Uhrzeigersinn durchnummeriert. Die Verbindung zwischen zwei aufeinander folgenden Vertices bildet eine Kante. Ein Polygonnetz heißt geschlossen, wenn jede Kante im Netz zu genau zwei Polygonen gehört. Polygonnetze werden in den meisten Fällen aus Dreiecken oder Vierecken aufgebaut, da diese mit relativ geringem Aufwand gerendert werden können. Je nach Anwendung können sie aber auch aus komplexeren Faces bestehen. Zur Verwaltung von Polygonnetzen gibt es verschiedene Datenstrukturen. Die gebräuchlichsten Repräsentationen werden im Folgenden kurz erläutert.

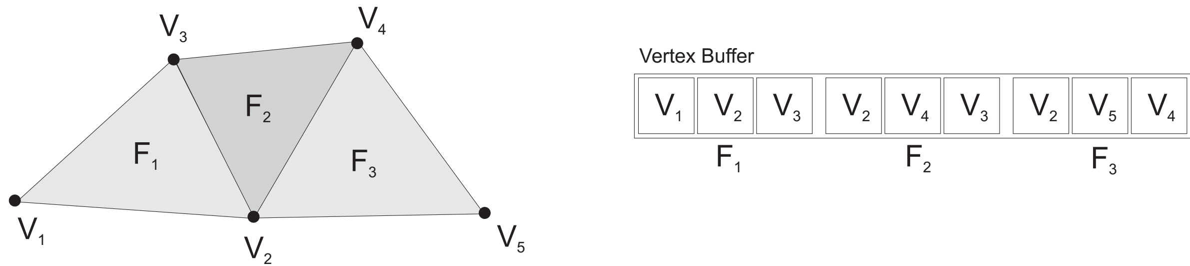


Abbildung 2.8: Ein Dreiecksnetz mit der dazugehörigen einfachen Datenstruktur bei sequenzieller Definition der Vertices. Alle Vertices, die zu mehreren Dreiecken gehören, sind mehrfach abgespeichert.

Einfache Datenstrukturen

Ein naiver Ansatz zur Definition eines Polygonnetzes ist, sequenziell alle Vertices eines jeden Polygons anzugeben. Vorteil dieser Datenstruktur ist, dass bei der Generierung von Oberflächen die erzeugten Vertices unabhängig von den bisher vorhandenen Polygonen direkt abgespeichert werden können. Ein Beispielnetz mit der dazugehörigen Datenstruktur zeigt Abbildung 2.8.

Der Nachteil dieser Methode ist, dass Vertices redundant im Netz gespeichert werden. Da in geschlossenen Netzen jeder Vertex zu mehreren Flächen gehört, würde es eigentlich ausreichen, ihn nur einmal zu speichern und auf ihn zu verweisen. Genau diese Idee wird beim indizierten Vertexbuffer verfolgt.

Indizierter Vertexbuffer

Ein indizierter Vertexbuffer besteht aus zwei getrennten Listen. In der ersten Liste sind die Positionsdaten der Vertices abgespeichert (Vertex-Buffer). Die zweite Liste definiert die Polygone und besteht aus Zeigern auf die Positionen der Vertices im Vertexbuffer (Index-Buffer, siehe Abbildung 2.9). Es wird also die Geometrie (Vertexpositionen) von der Topologie (Beziehungen zwischen den Vertices) des Netzes getrennt. Die Vertexdefinitionen können je nach Anwendung neben den reinen Positionsdaten auch noch weitere Informationen wie Normalenvektoren oder Farbinformationen enthalten. Da im Indexbuffer nur die Zeiger auf die Vertices abgelegt werden müssen, kann viel Speicher eingespart werden, indem auf bereits im Vertexbuffer vorhandene Vertices verwiesen wird.

Kantenliste

In einer Kantenliste werden die Faces im Polygonnetz durch Zeiger auf ihre Kanten definiert. Kanten bestehen in dieser Repräsentation aus Zeigern auf die beiden beteiligten Vertices und die beiden angrenzenden Faces. Es werden also drei Listen benötigt: ein Vertex-Buffer, der die Vertexdefinitionen enthält, ein Edge-Buffer, in dem die Kanten des Netzes abgelegt sind, und

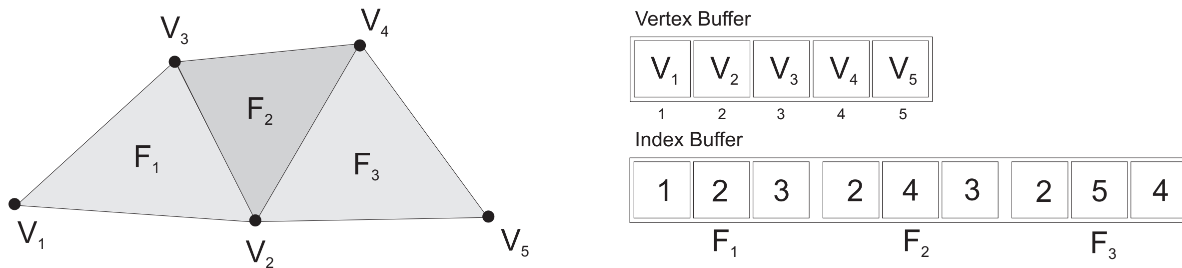


Abbildung 2.9: Beispiel einen indizierten Vertexbuffer. Das Modell wird dabei durch Index- und Vertexbuffer repräsentiert. Jeder Vertex muss nur einmal gespeichert werden.

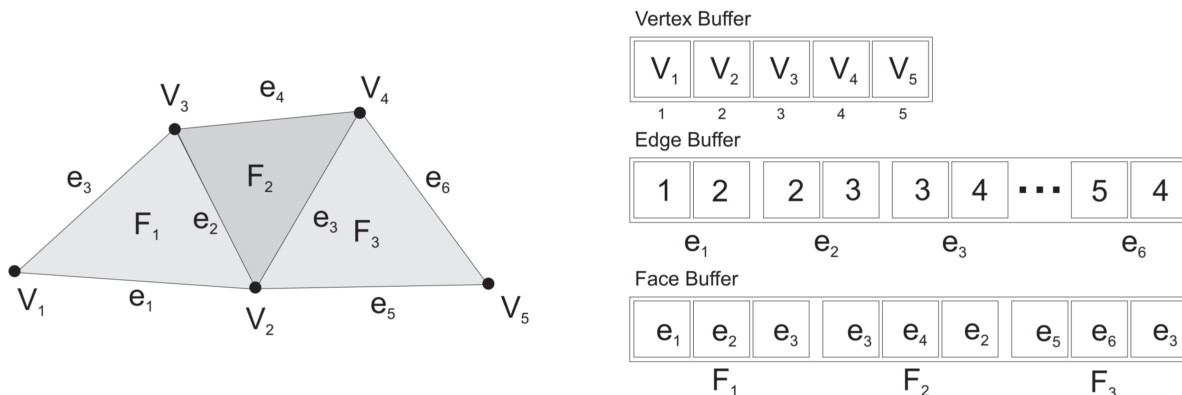


Abbildung 2.10: Ein Dreiecksnetz mit den Datenstrukturen für eine Kantenliste. Im Vertex-Buffer werden die Vertexdefinitionen gespeichert. Im Edge-Buffer sind die Kanten abgespeichert. Die Polygone im Netz werden durch die Zeiger auf die begrenzenden Kanten im Face-Buffer definiert.

ein Face-Buffer, der die Faces durch Zeiger auf die Begrenzungskanten definiert (siehe Abbildung 2.10). Eine Kantenliste ermöglicht es, in linearer Zeit (in der Anzahl der Kanten eines Faces) alle Nachbarn zu ermitteln, indem die Kanten einer Fläche traversiert und die Zeiger auf den anderen Nachbarn der Kante ausgewertet werden. Zudem können Randkanten in einem Netz schnell gefunden werden, da diese nur einen Zeiger auf ein zugehöriges Face haben. Komplexe Abfragen (z.B. alle Faces, die sich einen Vertex teilen) sind aber, wie in der Eckenliste, nur sehr ineffizient möglich.

Die Winged-Edge-Datenstruktur

Alle bisher vorgestellten Datenstrukturen sind nicht verzeigert, d.h. es werden keine Informationen über die Beziehungen der Flächen, Kanten und Vertices zueinander abgespeichert. Die Winged-Edge-Datenstruktur ist die älteste Datenstruktur, mit der Nachbarschaftsbeziehungen in polygonalen Netzen abgespeichert werden können. Sie wurde bereits 1975 von Baumgart beschrieben [14].

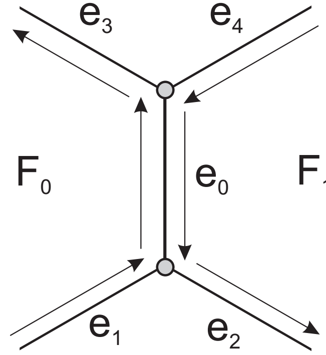


Abbildung 2.11: Kanten, Vertices und Faces in der Winged-Edge-Datenstruktur

Die Grundidee dieser Datenstruktur ist, dass neben Zeigern auf die Vertices und Nachbarn einer Kante auch Zeiger auf ihre Vorgänger und Nachfolger abgelegt werden. Werden die Vertices aller Polygone im Netz gegen den Uhrzeigersinn definiert, wird bei der Traversierung aller Faces im Netz jede Kante zweimal durchlaufen, allerdings in entgegengesetzten Richtungen. Aus diesem Grund müssen insgesamt vier Zeiger pro Kante abgelegt werden, nämlich die Vorgänger und Nachfolger, die beim Durchlaufen der Kanten eines Faces gegen den Uhrzeigersinn auf die aktuelle Kante folgen. Die Faces im Netz benötigen nur noch einen Zeiger auf eine beliebige Begrenzungskante. Von dieser aus können sie durch Aufruf der entsprechenden Nachfolgekante gegen den Uhrzeigersinn traversiert werden.

Abbildung 2.11 soll diese Tatsachen verdeutlichen. Die Kante e_0 wird bei der Traversierung der Fläche F_0 von unten nach oben durchlaufen. Die Vorgängerkante ist e_1. Bei der Traversierung von F_1 wird e_0 hingegen von oben nach unten durchlaufen. Vorgänger und Nachfolger sind nun e_4 bzw. e_2. Es müssen also Zeiger auf diese vier Kanten in e_0 gespeichert werden. Sie bilden die *„Flügel"* dieser Kante. Daher stammt der Name Winged-Edge. Zusätzlich erhält jeder Vertex Zeiger auf eine von ihm ausgehende Kante. So können insgesamt neun verschiedene Abfragen realisiert werden: Welche Kante, Ecke oder welches Face gehört zu jeder Kante, Ecke oder jedem Face [202]?

Die Half-Edge-Datenstruktur

Die Half-Edge-Datenstruktur speichert dieselben Informationen wie die Winged Edge Repräsentation. Allerdings können einige Abfragen etwas effizienter erfolgen. In dieser Darstellung wird die Tatsache, dass Kanten im Polygonnetz von benachbarten Faces in verschiedenen Richtungen durchlaufen werden, berücksichtigt, indem jede Kante der Länge nach in zwei gerichtete Halbkanten aufgespaltet wird (siehe Abbildung 2.12). Jede dieser Halbkanten enthält einen Zeiger auf das Face, zu dem sie bei der Traversion gegen den Uhrzeigersinn gehört. Damit die Nachbarschaftsbeziehungen zwischen den Faces nicht verloren gehen, besitzt jede Halbkante einen Zeiger auf die jeweilige Partnerkante. Zusätzlich enthalten die Vertices im Polygonnetz eine Liste mit Zeigern auf alle Halbkanten, die aus ihnen herauslaufen.

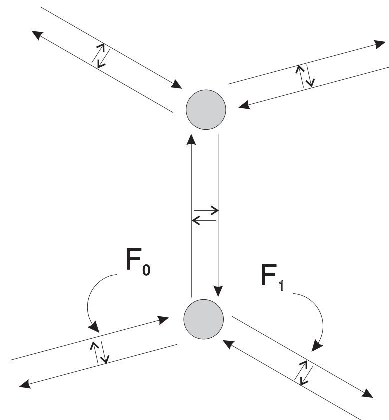


Abbildung 2.12: Verzeigerung der Kanten und Faces in der Half-Edge-Datenstruktur

Vergleich der Datenstrukturen

Einfache Datenstrukturen haben den Vorteil, dass sie leicht zu generieren sind und wenig Speicher verbrauchen. Dafür liegen aber nur wenige Informationen über die Topologie vor. Zur reinen Repräsentation eines Polygonnetzes sind indizierte Buffer optimal, da keine Vertices redundant gespeichert werden. Sollen aber Operationen wie z.B. das Entfernen eines Faces im Netz durchgeführt werden, benötigt man verzeigerte Datenstrukturen wie Winged-Edge oder Half-Edge, um die beteiligten Objekte bestimmen zu können. Tabelle 2.1 zeigt die Komplexitäten verschiedener topologischer Abfragen in den unterschiedlichen Datenstrukturen.

Man kann erkennen, dass alle Abfragen, die die Kanten betreffen, in der Eckenliste nicht verfügbar sind, da dort keine Kanten definiert sind. Abfragen nach Nachbarschaftsbeziehungen haben in dieser Datenstruktur eine Laufzeit, die linear von der Anzahl der Faces abhängt, weil alle Faces einmal überprüft werden müssen, um Nachbarn zu finden.

Die anderen vorgestellten Datenstrukturen erlauben alle Abfragen, die Laufzeiten variieren aber. Kantenliste und Winged-Edge haben dabei dieselben Laufzeiten, die Winged-Edge benötigt aber weniger Speicher, denn für die Definition der Faces muss nur ein Zeiger auf eine Kante abgelegt werden. Die Half-Edge-Datenstruktur ist aufgrund der Tatsache, dass die Vertices Listen mit Zeigern auf alle von ihnen ausgehenden Kanten haben, von der Laufzeit her am effizientesten. Allerdings ist der Speicherdedarf höher, da jedes Objekt in der Datenstruktur mehrere Zeiger enthält.

Austauschdatenformate

Zur Speicherung von Polygonnetzen existieren eine Reihe verschiedener sowohl offener als auch proprietärer Austauschdatenformate. Die wichtigsten werden an dieser Stelle kurz vorgestellt.

Tabelle 2.1: Übersicht über die Komplexitäten topologischer Abfragen in verschiedenen Repräsentationen von Polygonnetzen. Dabei bezeichnet k die Anzahl aller Kanten im Netz, k_f die Anzahl der Kanten eines Faces, k_ν die Anzahl der Kanten, die einem Vertex entspringen, und f die Anzahl der Faces im Netz (entnommen aus [202]).

Abfrage	Ind. Vertexbuffer	Kantenliste	Winged-Edge	Half-Edge
Kante → Ecken	nicht möglich	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante → Faces	nicht möglich	$\mathcal{O}(1)$	$\mathcal{O}(1)$	$\mathcal{O}(1)$
Kante → angrenzende Kanten	nicht möglich	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(k_\nu)$
Eckpunkt → Kanten	$\mathcal{O}(f \cdot k_f)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$O(k_\nu)$
Eckpunkt → Faces	$\mathcal{O}(f \cdot k_f)$	$\mathcal{O}(k)$	$\mathcal{O}(k)$	$\mathcal{O}(k_\nu)$
Face → Kanten	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$
Face → Eckpunkte	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$
Face → angrenzende Faces	$\mathcal{O}(f \cdot k_f^2)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$	$\mathcal{O}(k_f)$

Stanford PLY Das Stanford PLY-Dateiformat [180] wurde im Rahmen des Digital-Michelangelo-Projektes entwickelt. In diesem Format lassen Polygonnetze aus beliebigen Primitiven ablegen. Die Definition der Oberflächen erfolgt durch indizierte Vertexbuffer. Den in der Datei hinterlegten Elementen können benutzerdefinierte Attribute wie Normalen für Dreiecke und Vertices oder Farbinformationen mitgegeben werden. Die Daten lassen sich dabei als ASCII-Werte oder binär speichern. Werden in einer PLY-Datei vorhandene Elemente von einer Software nicht unterstützt, können diese dank des sequenziellen Dateiaufbaus beim Parsen ignoriert werden. Aufgrund dieser Flexibilität wird das PLY-Format von sehr vielen Programmen unterstützt. Der große Nachteil dieses Formates ist, dass sich keine Mesh-Segmentierungen speichern lassen, so dass die Zuweisung von Materialattributen und Texturinformationen schwierig zu realisieren ist.

Wavefront OBJ Im Wavefront OBJ-Format [205] lassen sich Polygonnetze in Form von indizierten Vertexbuffern ablegen. Standardmäßig können den Vertices Normalendefinitionen und Texturkoordinaten mitgegeben werden. Im Gegensatz zu PLY-Dateien lassen sich durch Definition von so genannten Objekten Gruppen von Oberflächen definieren. Diese können mit Materialinformationen wie Farbe, Reflektivität und Texturen assoziiert werden. Die Definition der Materialien geschieht in separaten .mtl-Dateien. Auf diese Art und Weise lassen sich für vorhandene Geometrien die Materialeigenschaften leicht austauschen. Der Nachteil dieses Formates ist, dass sich alle Daten nur im ASCII-Format ablegen lassen, so dass das Einlesen und Schreiben bei großen Modellen unter Umständen sehr lange dauert und die Dateien sehr viel mehr Speicherplatz benötigten als Binärformate.

STL-Dateien Das STL-Format [203] ist ein Standard Austauschformat aus dem CAD-Bereich. In STL-Dateien können lediglich Dreiecksflächen repräsentiert werden. Die Flächen werden dabei durch jeweils drei Vertices definiert. Gemeinsame Vertices der Flächen werden nicht berücksichtigt, d.h. es werden sehr viele redundante Elemente abgelegt. Zusätzlich

werden die Flächennormalen der Dreiecke abgelegt. In STL-Dateien lässt sich also nur die reine Geometrie abspeichern. Weitere Attribute sind nicht möglich. Es gibt allerdings die Möglichkeit, die Daten binär abzuspeichern. STL-Dateien werden oft als der "kleinste gemeinsame Nenner" beim Austausch von Geometrien angesehen und werden daher von nahezu allen CAD-Software-Paketen unterstützt.

Collada Collada [35] ist ein XML-basiertes Austauschformat für 3D-Modelle. Neben Unterstützung für verschiedene Oberflächendarstellungen sowie Textur- und Materialinformationen lassen ich in diesem Format auch Informationen über Kamerapositionen und Animationen hinterlegen. Das Austauschformat wird von vielen Open-Source-Projekten, u.a. Blender unterstützt. Auch die Simulationsumgebung Gazebo verwendet dieses Format zur Umgebungsrepräsentation.

Autodesk 3DS 3DS ist ein proprietäres binäres Austauschformat der Firma Autodesk. 3D-Modelle werden im 3DS-Format hierarchisch, d.h. in Form von Objektgruppen, hinterlegt. Neben Textur-, Material- und Geometrieinformationen können ebenfalls Kamera- und Animationseinstellungen gespeichert werden. Obwohl es ein proprietäres Format ist, existieren einige Open-Source-Bibliotheken zum einlesen von 3DS-Dateien, z.B. [108].

2.3.2 Marching Cubes

Der Marching-Cubes-Algorithmus wurde 1987 von Wiliam E. Lorensen und Harvey E. Cline [110] vorgestellt. Es handelt sich dabei um ein Verfahren, das ein Dreiecksnetz aus einer impliziten Flächenbeschreibung generiert (*Isosurface-Approximation*). Eine implizite Flächenbeschreibung hat die Form $f(\mathbf{p}) = c$, wobei c eine Konstante und $\mathbf{p} = (x, y, z)^T$ ein Punkt in $\mathbb{R}^3$ ist. Anhand der Konstante c kann bestimmt werden, wo sich ein Punkt relativ zur Fläche befindet. Es gilt: $f(\mathbf{p}) < c$, falls sich der Punkt unterhalb der Fläche befindet, $f(\mathbf{p}) = c$, falls sich der Punkt auf der Fläche befindet, und $f(\mathbf{p}) > c$, falls sich der Punkt oberhalb der Fläche befindet.

Marching Cubes wurde ursprünglich zur Visualisierung der Ergebnisse von Computertomografien verwendet, da die CT-Daten für unterschiedliche Gewebeschichten andere Werte aufweisen. So lassen sich z.B. die verschiedenen Gewebeschichten im menschlichen Körper durch Veränderung des Schwellwertes c darstellen. Durch geeignete Interpolationsverfahren lässt sich dieses Verfahren aber auch auf Laserscanner-Daten anwenden.

Marching-Squares

Zur Veranschaulichung des Grundprinzips von Marching Cubes, wird das Verfahren zunächst nur in zwei Dimensionen beschrieben. In dieser Vereinfachung werden zusammenhängende Linienzüge zur Approximation einer Isofläche erzeugt. Im zweidimensionalen Fall liegen alle Datenpunkte in einer Ebene. Diese wird in ein regelmäßiges Gitter aus Quadraten unterteilt (daher der Name *Marching-Squares*). Jedes Quadrat hat vier Ecken, für die bestimmt wird, ob sie sich innerhalb oder außerhalb der zu approximierenden Fläche befinden. Für jede Belegungskombination lässt sich nun der Verlauf der Fläche innerhalb der Zellen durch Geraden approximieren. Liegt zum

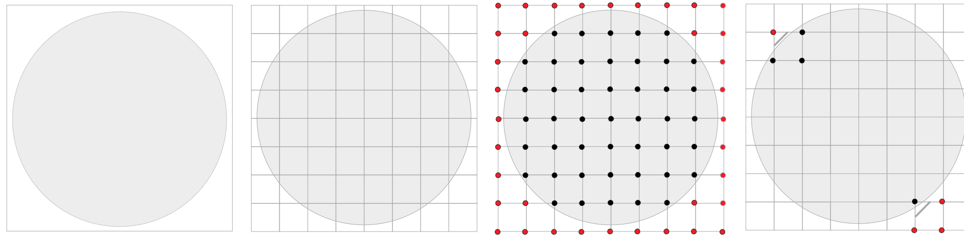


Abbildung 2.13: Beispiel für Linienapproximation eines Kreises mit Hilfe des Marching-Squares-Algorithmus. Die Ebene wird in quadratische Zellen unterteilt, und für jeden Eckpunkt wird bestimmt, ob er sich außerhalb der Fläche befindet (rot = außen). Ganz links sind zwei komplementäre Zellbelegungen mit der entstehenden Approximationslinie zu sehen.

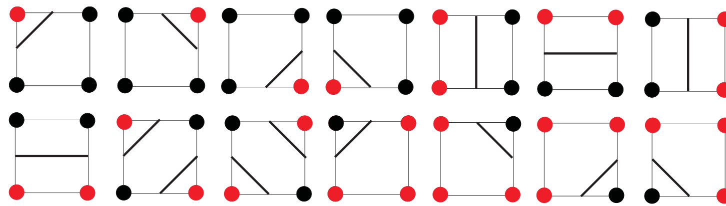


Abbildung 2.14: Mögliche Linienzüge beim Marching-Squares-Algorithmus. Man kann erkennen, dass komplementäre Belegungen die gleichen Linien erzeugen.

Beispiel nur ein Eckpunkt außerhalb der Fläche, wird eine Gerade durch die Mittelpunkte der an diese Ecke des Quadrates angrenzenden Kanten gezogen (s. Abbildung 2.13). Insgesamt ergeben sich im Zweidimensionalen $2^4 = 16$ mögliche Kombinationen von Belegungen. Da komplementäre Belegungen gleiche Approximationen ergeben, können insgesamt nur 7 Linienzüge entstehen. Diese sind in Abbildung 2.14 dargestellt. Befinden sich alle Ecken innerhalb oder außerhalb der Fläche, werden keine Linien erzeugt.

Erweiterung auf drei Dimensionen: Marching Cubes

Im dreidimensionen Fall wird der von den Messdaten eingenommene Raum in würfelförmige Zellen, auch *Voxel* genannt, eingeteilt. Wie im zweidimensionalen Fall wird dann ermittelt, welche der acht Ecken eines Würfels sich außerhalb der Isofläche befinden. Insgesamt können dabei 256 verschiedene Belegungskombinationen vorkommen. Allerdings reichen 16 Grundmuster zur Flächenapproximation aus (s. Abbildung 2.16), da komplementäre Belegungskombinationen die gleichen Approximationsmuster liefern und es zusätzlich auch noch Rotationssymmetrien gibt.

Die Approximationsmuster, die den Verlauf der Fläche innerhalb einer Zelle approximieren, bestehen aus bis zu vier Dreiecken, die Punkte auf den Würfelkanten als Vertices haben. Welche Punkte bei welchem Belegungsmuster verwendet werden, steht in einer zuvor berechneten Tabelle. Jede Zeile der Tabelle repräsentiert ein Muster und besteht aus mehreren Indizes, die auf Punkte auf den zwölf Würfelkanten verweisen. Jeweils drei aufeinanderfolgende Indizes definieren dabei ein Dreieck. Auf diese Art und Weise können Approximationen in konstanter Zeit

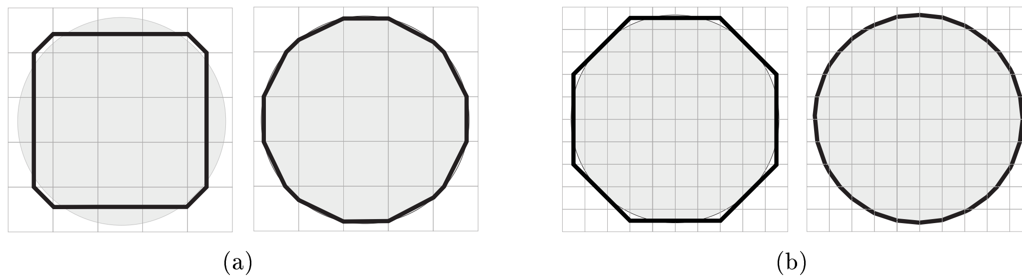


Abbildung 2.15: Verbesserung der Approximation im zweidimensionalen Fall durch Berechnung der Kantenschnittpunkte für zwei verschiedene Gitterkonstanten. Links sind jeweils die Muster zu sehen, die bei Verwendung des Kantenmittelpunktes zur Approximation benutzt werden. Selbst bei relativ grober Gitterauflösung kann so noch ein gutes Ergebnis erreicht werden s. Bild (a) rechts.

bestimmt werden. Die einzige Operation, die zur Bestimmung des zu einer Belegungskombination gehörenden Musters nötig ist, ist die richtige Zeile in der Tabelle aufzufinden. Dazu wird jeder Konfiguration ein Index zugewiesen, der diese Zeile repräsentiert. Dieser Index besteht aus einem 8-Bit Integer-Wert, in dem jedem Bit eine Ecke im Volumenelement zugeordnet ist. Befindet sich eine Ecke außerhalb der Fläche, wird das entsprechende Bit auf 1 gesetzt, ansonsten auf 0.

Im Standardfall werden die Mittelpunkte der Würfelkanten zur Definition der Vertices genommen. Dies hat den Nachteil, dass durch die vorgegebene Diskretisierung der möglichen Vertexpositionen eine Art "Treppeneffekt" entsteht. Eine Möglichkeit, dies zu verhindern, ist es, den Schnittpunkt der Fläche mit der Kante durch lineare Interpolation der impliziten Funktionswerte an den Würfelecken zu ermitteln. Dadurch entstehen weichere Übergänge zwischen den Approximationsmustern und das erzeugte Modell wird deutlich besser. Zwei Beispiele im zweidimensionalen Fall zeigt Abbildung 2.15.

2.3.3 Distanzfunktionen für Marching Cubes

Um das Marching-Cubes-Verfahren zur Rekonstruktion von 3D-Punktwolken anwenden zu können, muss aus den Daten eine Repräsentation gewonnen werden, die die aufgenommenen Oberflächen als Iso-Fläche beschreibt.

Rekonstruktion ohne Interpolation

Mit Hilfe eines einfachen Ansatzes kann man auch ohne Interpolation Rekonstruktionen aus 3D-Punktdaten erzeugen [195, 198]. Dazu werden diejenigen Punkte im Gitter auf den Wert "1" gesetzt, in deren Umfeld sich Datenpunkte befinden. Alle anderen Punkte erhalten den Wert "0". Auf jede Zelle werden dann die Marching-Cubes-Muster angewendet, wobei die Mittelpunkte der Gitterzellen zwischen Kanten mit unterschiedlichen Werten als Vertices der Dreiecke verwendet werden. Das Verfahren entspricht der Extraktion der Iso-Fläche mit dem Wert 0.5 auf der diskreten Distanzfunktion

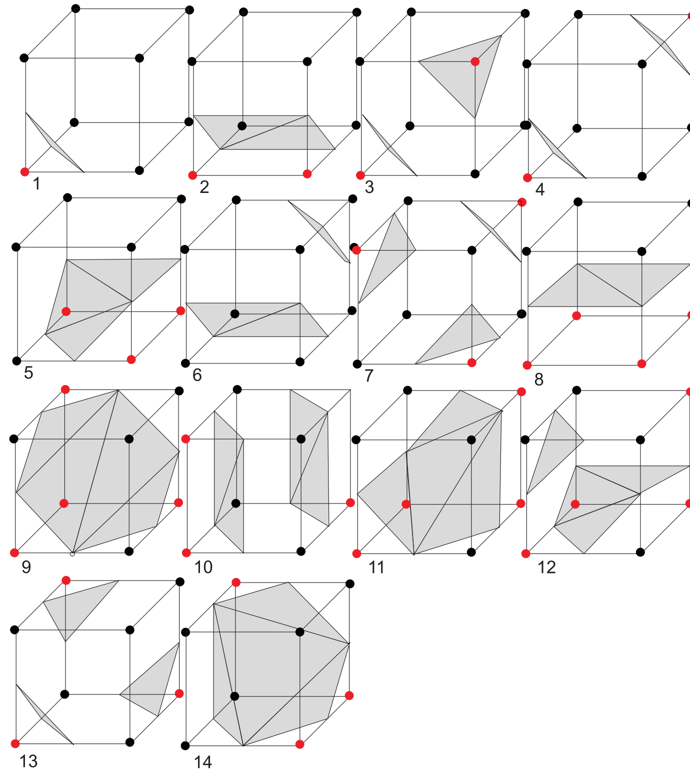


Abbildung 2.16: Grundmuster des Marching-Cubes-Algorithmus

$$f(x, y, z) = \begin{cases} 0, \text{falls Punkt in Umgebung} \\ 1, \text{sonst} \end{cases}$$

wobei x, y, z die Koordinaten eines Punktes im Rekonstruktionsgitter sind. Abbildung 2.17 veranschaulicht das Prinzip.

Die mit diesem Verfahren erzeugten Meshes weisen einen "Klötzcheneffekt" auf, der durch die Diskretisierung der Vertexpositionen auf den Kantenmittelpunkten der Zellen entsteht (siehe Abbildung 2.17. Dieser Effekt entsteht nicht nur bei Krümmungen, sondern kann auch bei ebenen Flächen auftreten, wenn diese nicht parallel zu den Gitterachsen ausgerichtet sind (siehe Abbildung 2.19). Die abrupten Sprünge in der Rekonstruktion entstehen bei Ebenen immer dann, wenn bei der Diskretisierung der Messdaten eine "Zeile" oder "Spalte" im Gitter verlassen wird, analog zum Treppeneffekt bei der Darstellung von diagonalen Linien in 2D-Pixelgrafiken. Die Ausprägung dieser Artefakte hängt von der gewählten Gitterauflösung ab. Eine feinere Auflösung sorgt dafür, dass die Sprünge häufiger auftreten, aber nicht so ausgeprägt sind. Eine gröbere Gitterauflösung wird bei gleicher Ausrichtung weniger Sprünge erzeugen, aber die Rekonstruktion wird noch mehr an Details verlieren.

Neben der Diskretisierung weisen die ohne Interpolation gewonnenen Rekonstruktionen einen

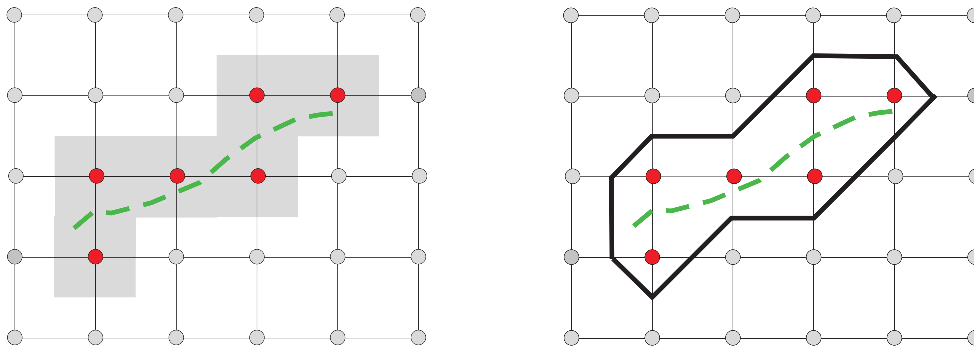


Abbildung 2.17: Prinzip der Rekonstruktion ohne Isoflächen-Interpolation. Diejenigen Eckpunkte im Gitter, in deren Umgebung (grau) sich Messpunkte befinden, werden markiert. Zellen, in deren Umgebung sich keine Datenpunkte befinden, bleiben unmarkiert. Auf die so entstehende Belegung werden die Marching-Cubes-Muster angewendet.

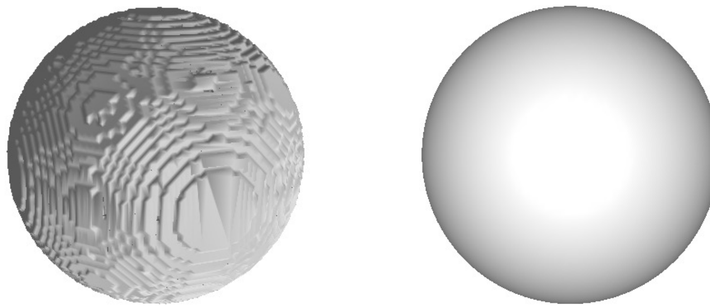


Abbildung 2.18: "Klötzcheneffekt" bei Marching-Cubes-Rekonstruktionen ohne Interpolation am Beispiel der Rekonstruktion einer gesampelten Kugeloberfläche. Durch die fehlende Interpolation ergibt sich eine Diskretisierung der Vertexpositionen in der Rekonstruktion, die vor allem bei gekrümmten Oberflächen auffällt (links). Durch Interpolation der Vertexpositionen lässt sich ein wesentlich besseres Approximationsergebnis bei gleicher Gitterauflösung erzielen.

weiteren Effekt auf: Um die Messdaten wird immer eine geschlossene Hülle erzeugt. Das liegt daran, dass durch das Schieben der Datenpunkte auf die Gitterzellen immer 2 Übergänge zwischen den Isowerten "0" und "1" entstehen. Es ist also mit diesem einfachen Verfahren nicht möglich, einzelne Oberflächen zu rekonstruieren. So wird z.B. die Oberfläche einer gescannten Wand immer zwei Konturen erzeugen: eine Fläche vor den Messdaten und eine dahinter, wie ebenfalls in in Abbildung 2.19 zu sehen ist. Aus dem selben Grund führen Ausreißer in den Messdaten zu diamantförmigen Artefakten, da alle 8 den Punkt umgebenden Zellen jeweils ein Dreieck erzeugen (gemäß Muster 1 der Marching-Cubes-Tabelle).

In gut zu den Gitterachsen ausgerichteten Datensätzen mit vielen ebenen Anteilen lassen sich aber auch mit diesem Verfahren visuell ansprechende Rekonstruktionen erzielen. Die Diskretisierung begrenzt allerdings die im Mittel zu erreichende Vertexgenauigkeit auf $\frac{a}{2}$, wenn a die gewählte Gitterauflösung ist. Problematisch ist zudem die Bildung der Doppelkonturen, da diese die Anzahl der Dreiecke im Mesh verdoppelt. Zusätzlich kann dies bei der Nutzung des Meshes als Karte für einen Roboter, z.B. zur Lokalisierung, problematisch sein kann.

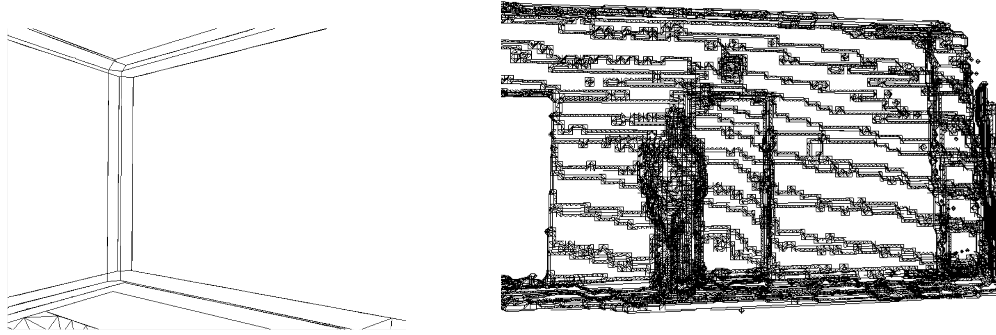


Abbildung 2.19: Diskretisierungsfehler bei nicht ausgerichteten Ebenen und Doppenhüllen. Im Linken Bild waren die Wände der Umgebung an den Achsen des Sensorkoordinatensystems ausgerichtet. Dadurch entstanden innerhalb der Ebenen keine Sprünge. Man kann auch gut den Doppelhülleneffekt erkennen. Im rechten Bild war der Scanner leicht nach unten geneigt. Die hintere Wand steht dadurch schräg in der Szene, wodurch die Stufen hervorgerufen werden.

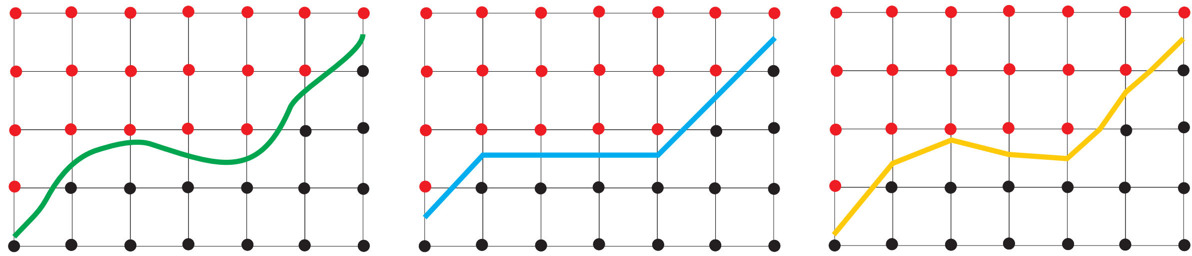


Abbildung 2.20: Prinzip der Vertexinterpolation bei Approximation des grünen Isoflächenverlaufs. Die blaue Linie im mittleren Bild gibt den Verlauf der Rekonstruktion ohne Interpolation wieder. Die gelbe Linine im rechten Bild zeigt die Rekonstruktion mit Interpolation.

Interpolierte Distanzfunktionen

Das gerade beschriebene Problem der Diskretisierung kann gelöst werden, indem die Vertices der Marching-Cubes-Muster auf den Zellenkanten interpoliert werden, d.h. es wird der Punkt auf der Kante approximiert, der die Isofläche schneidet (vgl. Abbildung 2.20). Durch die Interpolation kann der tätsächliche Lininenverlauf im dort gezeigten Beispiel bei gleicher Gitterauflösung deutlich besser approximiert werden. Die Herausforderung bei der Rekonstruktion besteht darin, eine mathematische Isoflächenbeschreibung der durch die Scanpunkte repräsentierten Oberfläche zu finden. Die Schnittpositionen der Vertices lassen sich besonders leicht interpolieren, wenn die Datenpunkte $\boldsymbol{p_i}$ mit einer linearen Funktion $d(\boldsymbol{p_i})$ approximiert werden, die an diesen Stellen den Wert 0 annimmt:

$$d(\boldsymbol{p_i}) = 0$$

Bei einer solchen Funktion haben Punkte, die auf der einen Seite der Fläche liegen, einen positiven Isowert, alle anderen einen negativen (*Signed Distance Function*). Durch eine derartige Repräsentation wird auch das Problem der Doppelkonturen gelöst, da es immer nur genau einen

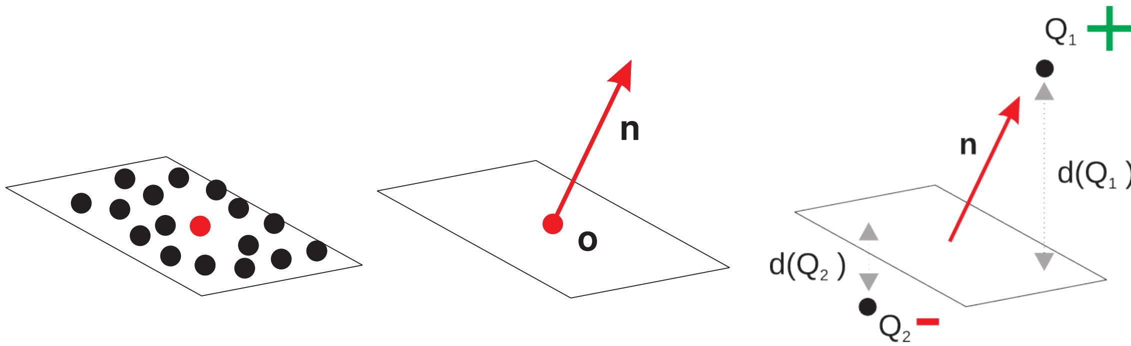


Abbildung 2.21: Konstruktion der Distanzfunktion nach Hoppe. Für jeden Datenpunkt wird eine Tangentialebene mittels eines Least-Squares-Fits an k Datenpunkte berechnet (links). Jede Ebene ist durch ihren Schwerpunkt $\boldsymbol{o}$ und eine Flächennormale $\boldsymbol{n}$ definiert (Mitte). Die Distanz ergibt sich aus der senkrechten Projektion eines Abfragepunktes $d(\boldsymbol{q_i})$ in die nächstgelegene Tangentialebene. Das Vorzeichen ergibt sich aus der relativen Orientierung von $d(\boldsymbol{q_i})$ zu dieser Ebene, die anhand der Flächennormale bestimmt wird (rechts).

Vorzeichenwechsel an der Position der repräsentierten Fläche gibt. Dementsprechend erzeugen auch nur Zellen Flächen, in denen unterschiedliche Vorzeichen vorkommen. In solchen Zellen können die Schnittpunkte der betroffenen Kanten mit der approximierten Fläche leicht durch lineare Interpolation berechnet werden. Für den Interpolationspunkt $\boldsymbol{i}$ gilt:

$$\boldsymbol{i} = \boldsymbol{c_2} - \lambda \cdot (\boldsymbol{c_2} - \boldsymbol{c_1})$$

für zwei Gitterpositionen $\boldsymbol{c_1}$ und $\boldsymbol{c_2}$ mit unterschiedlichen Vorzeichen. Da $d(\boldsymbol{x})$ linear ist und an den gesuchten Stellen den Wert 0 annimmt, kann man λ wie folgt bestimmen:

$$\begin{aligned} 0 &= f(\boldsymbol{c_2}) - \lambda \cdot (f(\boldsymbol{c_1}) - f(\boldsymbol{c_2})) \\ \Leftrightarrow f(\boldsymbol{c_2}) &= \lambda \cdot (f(\boldsymbol{c_1}) - f(\boldsymbol{c_2})) \\ \Leftrightarrow \lambda &= \frac{f(\boldsymbol{c_2})}{f(\boldsymbol{c_1}) - f(\boldsymbol{c_2})} \end{aligned}$$

Hoppes Distanzfunktion

Funktionen dieser Art können auf verschiedene Art und Weise konstruiert werden. Ein bekannter Ansatz wurde bereits 1992 von Hoppe [83] vorgestellt. Die Idee dabei ist, jeden Datenpunkt mit einer lokalen Tangentialebene $T(\boldsymbol{p_i})$ zu assoziieren. Die Tangentialebene wird dabei durch einen Least-Sqaures-Fit an k Nachbarpunkte berechnet ("k-Nachbarschaft"). Jede dieser Tangentialebenen wird durch den Schwerpunkt $\boldsymbol{o}$ der k-Nachbarschaft und die Normale $\boldsymbol{n_i}$ der gefitteten Ebene repräsentiert:

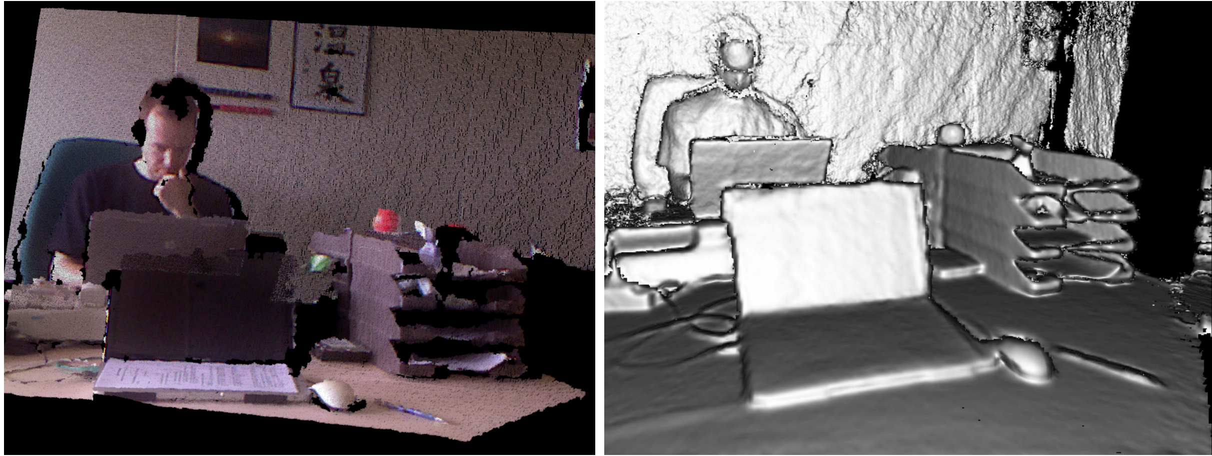


Abbildung 2.22: Beispiel einer Rekonstruktion mit Kinect-Fusion.

$$T(\boldsymbol{p_i}) = \boldsymbol{o_i} \cdot \boldsymbol{n_i}$$

Die Distanzfunktion $d_T(\boldsymbol{p})$ für jeden Punkt $\boldsymbol{p}$ definiert Hoppe wie folgt:

$$d_T(\boldsymbol{p}) = s(\boldsymbol{p}) \cdot d(\boldsymbol{p}, T)$$

In dieser Gleichung ist $d(\boldsymbol{p}, T)$ der senkrechte Abstand des Punktes $\boldsymbol{p}$ zur nächstgelegenen Tangentialebene. Der Faktor $s(\boldsymbol{p}) \in \{1-, +1\}$ legt das Vorzeichen fest. Es wird relativ zur Flächennormale bestimmt, d.h. wenn die Normale in Richtung des Punktes zeigt, wird die Distanz positiv, ansonsten negativ. Daher ist eine konsistente Ausrichtung der berechneten Normalen entscheidend. Abbildung 2.21 veranschaulicht die Konstruktion von Hoppes Distanzfunktion.

Diese Art der Repräsentation hat mehrere Vorteile. In dichten Datensätzen mit wenig Messrauschen kann die Anzahl der k-Nachbarn zum Fitten der lokalen Tangentialebene klein gewählt werden ($k >= 3$), so dass die Berechnung sehr effizient möglich ist. In verrauschten Datensätzen kann durch Wahl einer größeren k-Nachbarschaft das Messrauschen gut ausgeglichen werden, da sich dieses durch den Least-Squares-Fit herausmittelt. Da die so konstruierte Funktion nicht zwingend stetig ist, lassen sich auch scharfe Kanten und Sprünge in den gemessenen Oberflächen relativ gut repräsentieren, allerdings werden diese je nach Größe der k-Nachbarschaft durch den Least-Squares-Fit mehr oder weniger stark abgerundet.

Eine modifizierte Version so einer Distanzfunktion wird auch von Kinect Fusion [88] verwendet. Dabei wird insbesondere die gleichmäßige Anordung der Messpunkte als Tiefenbild ausgenutzt, um Punktnormalen und somit die lokalen Approximationsebenen zu bestimmen [136]. Das Verfahren ist durch eine massiv parallelisierte GPU-Implementierung in der Lage, aus Kinect-Punktwolken qualitativ hochwertige Rekonstruktionen in Echtzeit zu erstellen. Da Grafikkarten in der Regel vergleichweise wenig Arbeitsspeicher zur Verfügung stellen, ist das Verfahren derzeit nur in der Lage, Rekonstruktionen begrenzter Größe zu generieren. Laut Original-Papier [88]

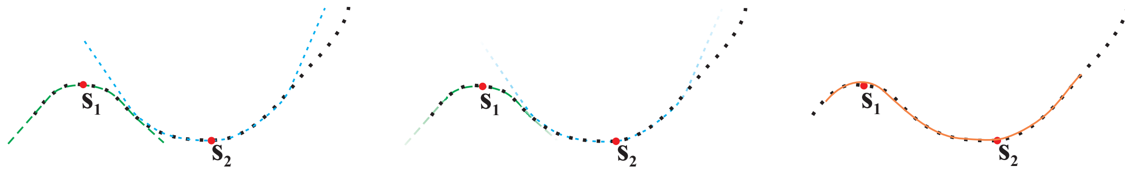


Abbildung 2.23: Veranschaulichung der MLS-Rekonstruktion am Beispiel von zwei Stützpunkten S_1 und S_2. Zunächst werden für die Stützpunkte lokale Approximationen berechnet (links: grün für S_1 und blau für S_2). Durch die Gewichtung tragen diese Funktionen nur lokal zur globalen Approximation bei (Mitte). Die globale MLS-Fläche wird durch Überlagerung der gewichteten lokalen Approximationen gebildet (rechts: gelb).

benötigt ein Gitter mit 512^3 Zellen bereits 512 MB Arbeitsspeicher. Das Gitter für die Rekonstruktion ist in der PCL-Implemetierung auf 3 x 3 x 3 Meter begrenzt. In dieser Konfiguration werden schon 6 GB Grafikspeicher benötigt, die auf den derzeit gängigen Grafikkarten in der Regel nicht vorhanden sind. Ein Beispiel für eine Kinect-Fusion-Rekonstruktion zeigt Abbildung 2.22. Typisch für diese Art der Rekonstrutktion ist, dass in den Meshes scharfe Kanten schlecht repräsentiert werden, wie im linken Bild der Abbildung deutlich zu erkennen ist.

Approximation durch Polynomfunktionen

Hoppes Ansatz benutzt lokale Ebenen, um die Oberflächen zu approximieren. Insbesondere bei gekrümmten Flächen bietet sich die Flächenapproximation durch Polynome höheren Grades an, da Hoppes Verfahren keine stetige Oberflächenbeschreibung generiert, was zu Unebenheiten in den Rekonstruktionen führen kann [6]. Solche Verfahren basieren häufig auf dem *Moving Least Squares*-Verfahren [102]. Das Prinzip wird im Folgenden kurz erklärt. Als Grundlage der Schilderung dient der Aufsatz von Nealen [135], in dem auch die dazugehörigen mathematischen Details und Formulierungen zu finden sind.

MLS berechnet für jeden Punkt lokal eine gewichtete Least-Squares-Approximation für ein Polynom n-ten Grades über alle Messpunkte. Der Referenzpunkt wird also über den Datensatz bewegt. Diese Funktionen werden mit einer symmetrischen Gewichtsfunktion, die mit wachsendem euklidischen Abstand zum Referenzpunkt gegen Null konvergiert, multipliziert. In der Praxis werden zur Gewichtung oft Gaußfunktionen oder die Wendland-Funktion [193] genutzt. Aus der so berechneten Menge lokaler Approximationen wird dann diejenige Überlagerung einer Teilmenge dieser Funktionen bestimmt, die den globalen Fehler minimiert. Abbilung 2.23 veranschaulicht das Verfahren. Durch die Gewichtung wird sicher gestellt, dass die einzelnen Funktionen lokal sehr genau sind, global auf weit entfernte Punkte aber wenig Einfluss haben. Da die Gewichtsfunktionen sehr schnell konvergieren, muss in der Praxis zur Berechnung der lokalen Polynome nicht der gesamte Datensatz betrachtet werden, sondern nur Punkte, die nahe am Referenzpunkt sind, da weiter entfernte Stützpunkte kaum noch Einfluss auf das Ergebnis haben. Die Größe des zu betrachtenden Gebietes hängt dabei von der Parametrierung der Gewichtsfunktion ab.

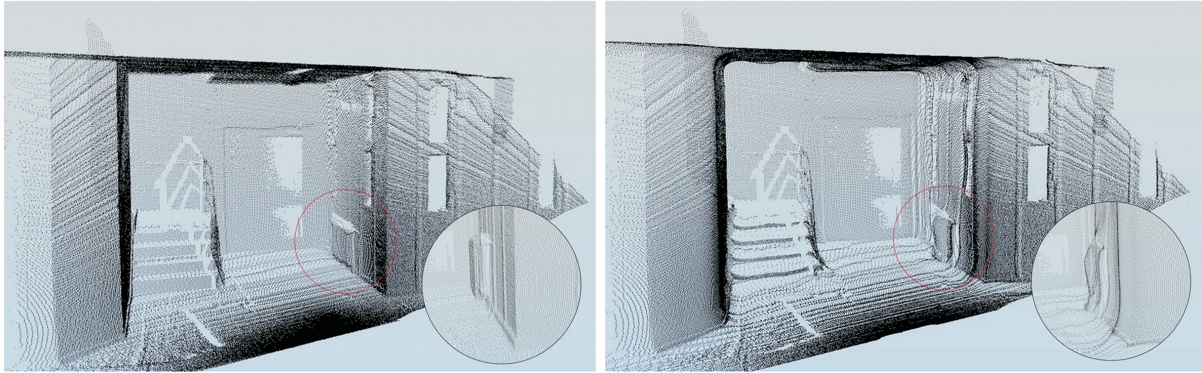


Abbildung 2.24: Veranschaulichung einer MLS-Projektion. Die Datenpunkte des Originalscans (linkes Bild) wurden auf die berechnete MLS-Oberfläche projiziert (rechtes Bild). Rauschen und Ausreißer werden vermindert, aber scharfe Kanten gehen verloren. Zur Veranschaulichug wurden die Paramter so gewählt, dass der Effekt stark zum Vorschein kommt. Bei kleineren Suchradien sind die Abrundungen weniger stark ausgeprägt.

Levin [106] hat gezeigt, dass MLS-Repräsentationen genau dann stetig sind, wenn die Gewichtsfunktion stetig ist [135], was in der Regel der Fall ist. Die resultierenden Flächen sind also sehr weich und können keine Unstetigkeiten repräsentieren, weshalb der Einsatz solcher Verfahren für die Rekonstruktion von Gebäudegeometrien nur bedingt sinnvoll ist. Darüber hinaus ist das Berechnen der MLS-Oberfläche selbst bei Beschränkung der genutzten Stützpunkte sehr rechenaufwändig.

Zur Rekonstruktion mit Marching Cubes können solche Darstellungen benutzt werden, indem die Polynomfunktionen in ein zum Gitter ausgerichtetes Bezugssystem transformiert werden. Analog zu Hoppes Funktion kann dann der orientierte Abstand der Gitterpunkte zu der berechneten Fläche ausgerichtet werden. In Meshlab werden zwei spezielle MLS-Varianten (APSS [74] und RIMLS [149]) zur Marching-Cubes-basierten Flächenrekonstruktion verwendet.

MLS-Flächen werden nicht nur zur meshbasierten Rekonstruktion eingesetzt, sondern auch zur Filterung von Messdaten. So können verrauschte Datensätze durch Projektion der Messpunkte auf die MLS-Fläche ausgeglichen werden. Ebenso lassen sich die Flächen zum Up- und Downsamplen verwenden, um fehlende Daten zu ergänzen (z.B. Schattierungen in 3D-Laserscans) oder die Daten gleichmäßig zu reduzieren. In PCL [166] ist eine Funktion zur Projektion auf MLS-Oberflächen implementiert. Abbildung 2.24 zeigt die resultierenden Ergebnisse für verrauschte Kinect-Scans bei unterschiedlicher Parametrisierung des MLS-Algorithmus. Man kann darin deutlich erkennen, dass die scharfen Kanten zwischen den Wänden abgerundet sind und alle Flächen einen sehr weichen Übergang ineinander haben.

Poisson-Rekonstruktion

Ein weiterer Ansatz, Messpunkte durch eine Isofläche zu approximieren, wurde von Kazhdan [92] vorgestellt. Dabei wird angenommen, dass es eine so genannte *Indikatorfunktion* χ gibt, die

innerhalb eines Modells den Wert 1 und außerhalb den Wert 0 hat. Die mit Normalen versehenen Datenpunkte werden als Vektorfeld $\vec{V}$ interpretiert, das die Ableitung $\nabla\chi$ von χ stichprobenartig repräsentiert. Das Problem, die Indikatorfunktion zu berechnen, kann nun so formuliert werden: Es wird eine skalare Funktion gesucht, deren Ableitung $\vec{V}$ am besten approximiert. Es muss also $min_{\chi}\|\nabla\chi - \vec{V}\|$ gefunden werden. Dies lässt sich als Poisson-Problem formulieren:

$$\Delta\chi \equiv \nabla \cdot \nabla\chi = \nabla \cdot \vec{V}$$

Dieser Ansatz bietet diverse Vorteile: Anstelle von lokalen Approximationen wird eine globale Lösung für das Problem gefunden. Das Verfahren ist dadurch auch robust gegenüber Rauschen. Zur Bestimmung von χ wird eine Überlagerung aus trivarianten B-Splines berechnet, die aufgrund der Randbedingung, dass χ außerhalb der Fläche 0 sein soll, mit Abstand zur Isofläche rasch gegen 0 konvergiert. Die Rekonstruktion lässt sich sehr schnell berechnen, allerdings wächst das zu lösende Gleichungssystem mit der Anzahl der Punkte, wodurch viel Arbeitsspeicher benötigt wird. Für große Punktmengen ist die Poisson-Rekonstruktion also nicht geeignet. Zur Lösung von Poisson-Problemen gibt es auch GPU-basierte Implementierungen [70].

Durch die Verwendung von B-Splines ist die Rekonstruktion sehr weich. Scharfe Kanten können nicht repräsentiert werden. Durch die Formulierung des Problems der Rekonstruktion über die Indikatorfunktion, die lediglich Bereiche innerhalb und außerhalb des Modells definiert, liefert das Verfahren lediglich auf approximativ konvexen Modellen brauchbare Ergebnisse (siehe 4.4.1). Zur Polygonalisierung der berechneten Isofläche wird ein Octree in Verbindung mit einem modifizierten Marching-Cubes-Algorithmus verwendet [91, 92, 206].

2.3.4 Erweiterte Marching-Cubes-Varianten

Bei Anwendung des originalen Marching-Cubes-Algorithmus kann es durch Mehrdeutigkeiten bei der Zuordnung von Flächenmustern auf Gitterkonfigurationen zu Löchern in den Rekonstruktionen kommen. Darüber hinaus ist der Algorithmus nicht in der Lage, scharfe Kanten innerhalb einer Gitterzelle wiederzugeben. Um diese Probleme zu lösen, lassen sich in der Literatur viele Ansätze finden. Eine umfassende Übersicht über Marching-Cubes-Varianten und deren Eigenschaften gibt Newman in [137] aufgeführt. In diesem Abschnitt sollen die wichtigsten kurz basierend auf [137] dargestellt werden.

Alternative Triangulationstabellen

Die in Abschnitt 2.3.2 gezeigten Flächenmuster erzeugen in der Regel geschlossene Flächen ohne Löcher. Bei einigen Kombinationen von Flächenmustern kann es allerdings zu Mehrdeutigkeiten kommen [45], d.h., es werden je nach Konfiguration lokal unterschiedliche Muster benötigt, um geschlossene Flächen zu rekonstruieren. Ein Beispiel einer mehrdeutigen Kombination wird in Abbildung 2.25 gezeigt (entnommen aus [195]). Die Anwendung von Muster 4 und Muster 6 der Originaltabelle führt dazu, dass ein Loch im Mesh entsteht (obere Zeile). Durch Einführung einer für diesen Sonderfall bestimmten alternativen Triangulation kann dieses geschlossen werden. Solche mehrdeutigen Konfiguration können bei 7 der ursprünglichen Approximationsmuster

auftreten: (3, 4, 6, 7, 10, 12 und 13) [34, 66, 134]. Van Gelder hat experimentell bestimmt, dass solche Muster typischerweise zwischen 3% und 5,6% aller Zellen vorkommen [66].

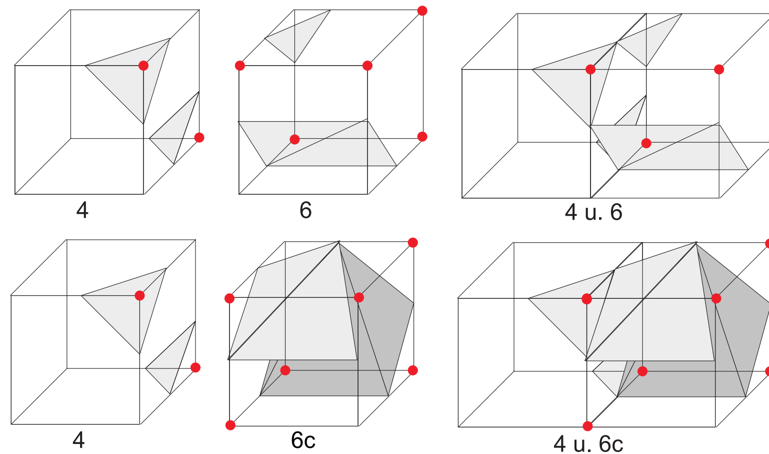


Abbildung 2.25: Beispiel für eine mehrdeutige Musterkonfiguration beim Marching-Cubes-Algorithmus. Durch Kombination der Grundmuster 4 und 6 entsteht ein Loch im Polygonnetz (oben). Durch eine alternative Triangulation (6c) kann dieses geschlossen werden [195].

Alternative Triangulationsmuster werden in vielen Implementationen durch spezielle Tabellenstrukturen aufgelöst: Sobald in einer Zelle ein mehrdeutiges Muster auftritt, werden die Muster der Nachbarzellen mit betrachtet und dann die für die lokale Konfiguration passende Triangulation bestimmt [12]. Nagae und andere [131, 138, 139] haben gezeigt, dass man auch mit einer einzigen Triangluationstabelle löcherfreie Flächen erstellen kann, indem man auf die Ausnutzung der Komplementärsymmetrien verzichtet und für jeden dieser Fälle individuelle Triangulationsmuster einführt.

Marching Tetrahedrons

Außer durch alternative Triangulationstabellen lassen sich Mehrdeutigkeiten durch Unterteilung der kubischen Gitterzellen in Tetraeder auflösen. Dieses Verfahren ist unter dem Namen "Marching Tetrahedrons" bekannt [151, 173]. Die Zerlegung der Gitterzellen kann in 5 oder 6 Tetraeder erfolgen (siehe Abbildung 2.26), üblicherweise wird aber die Zerlegung mit 6 Tetraedern gewählt, da diese dann immer gemeinsame Kanten mit den Nachbarzellen haben, was die Integration neuer Dreiecke in das Polygonnetz erleichtert. Da die Tetraeder aus 4 Vertices bestehen, kann es 16 mögliche Belegungen der Ecken geben. Durch Ausnutzung der Komplementärsymmetrie reduziert sich die Zahl der tatsächlich zu betrachtenden Muster auf 7 (siehe Abbildung 2.27).

Der große Vorteil dieses Verfahrens ist, dass die Triangulationstabellen aufgrund der wenigen auftretenden Muster leicht erstellt werden können. Nachteilig ist, dass mehr Dreiecke generiert werden als bei Marching Cubes, da prinzipiell jeder Tetraeder bis zu zwei Dreiecke erzeugen kann. Je nach gewählter Zerlegung können also bis zu 10 Dreiecke pro Zelle entstehen. Im Mittel werden in etwa doppelt so viele Dreiecke erzeugt wie mit einem kubischen Gitter (nach [195]).

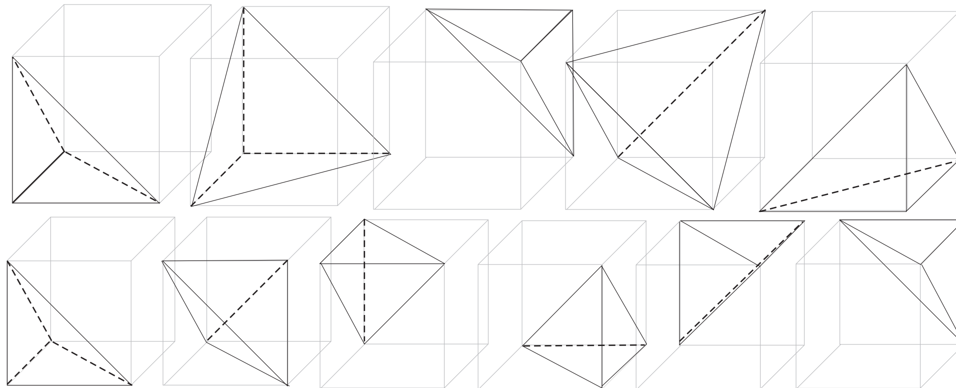


Abbildung 2.26: Zerlegung von kubischen Gitterzellen in 5 oder 6 Tetraeder

Extended Marching Cubes

Betrachtet man die Approximationsmuster des Marching-Cubes-Algorithmus, wird man sehr schnell erkennen, dass sich mit dem Verfahren scharfe Kanten nicht genau wiedergeben lassen, denn es wird immer nur auf den Kanten der Würfel zwischen den Abständen zur zu approximierenden Isofläche linear interpoliert. Verläufe der Fläche, die innerhalb der Voxel stattfinden, können daher nicht erfasst werden, weil die Auflösung durch die Diskretisierung beschränkt ist und die Vertexpositionen nur entlang der Hauptachsen des Gitters verschoben werden können. Es existieren also keine Muster, die Knicke der Isofläche mit Subvoxelgenaugkeit bestimmen können. Abbildung 2.28 verdeutlicht die Problematik. Zur Behandlung solcher Fälle wurde von Kobbelt ein Verfahren namens "Extended Marching Cubes" vorgeschlagen [98]. Es führt zwei Erweiterungen zum Originalalgorithmus ein: verbesserte Schätzung der Vertexpositionen auf den Gitterkanten durch Auswertung eines dreidimensionalen Abstandwertes zur Isofläche anstatt der direkten Interpolation des skalaren Isowertes und ein Verfahren zur Approximation von Knicken der Isofläche innerhalb einer Gitterzelle.

Die erste Erweiterung geht auf das Problem ein, dass bei ungünstigen Verläufen der Isofläche durch lineare Interpolation des euklidischen Abstandes schlechte Schätzungen für die Vertexpositionen erzielt werden können. Betrachtet man lediglich die geringste senkrechte Entfernung zur Fläche, kann dies unter Umständen dazu führen, dass die interpolierte Vertexposition stark vom tatsächlichen Schnitt zwischen Kante und Isofläche abweicht. Das linke Bild in Abbildung 2.29 zeigt ein Beispiel für so eine Situation. Die Auswertungspunkte im Gitter haben in etwa den betragsmäßig gleichen euklidischen Abstand zur Fläche, was dazu führt, dass ein Schnittpunkt in der Mitte der Kante berechnet wird (grauer Punkt), obwohl der rote Punkt den eigentlichen Schnittpunkt darstellt. Zur Lösung dieses Problems schlägt Kobbelt vor, nicht den direkten Abstand zu interpolieren, sondern für jeden Punkt im Gitter drei Abstandswerte entlang der Gitterachsen zu bestimmen (Abbildung 2.29 rechts). Zur Interpolation der Vertices werden jetzt die jeweiligen gerichteten Abstände entlang der entsprechenden Achse verwendet. Dazu ist keine Anpassung des Originalalgorithmus notwendig. Kobbelt schlägt vor, die Abstände direkt zur Punktwolke mit einem Raytracing-Verfahren zu ermitteln, indem jeweils der Abstand zum nächs-

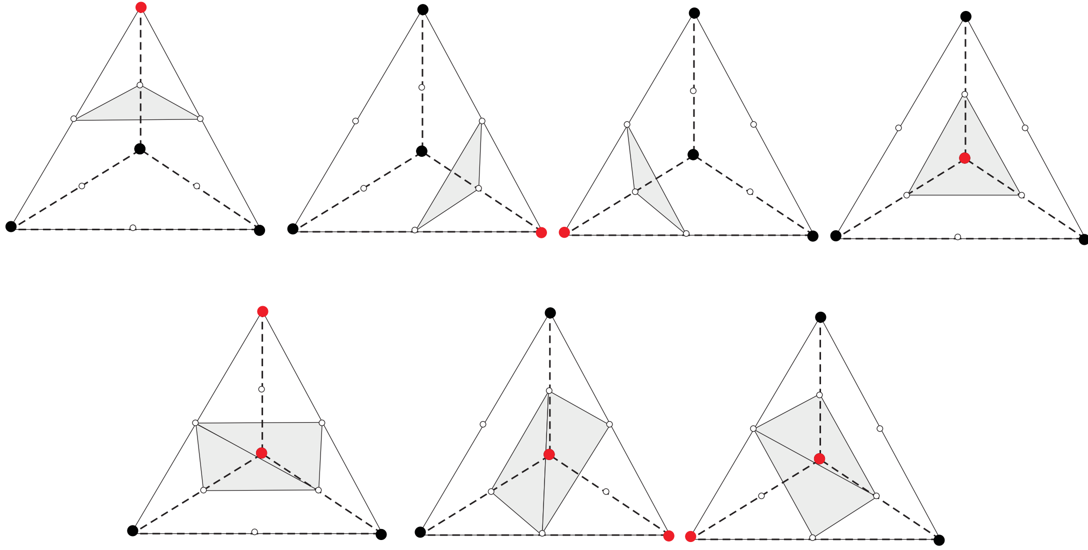


Abbildung 2.27: Die 7 möglichen Approximationsmuster beim Marching-Tetrahedrons-Verfahren. Pro Tetraederzelle wird je nach Zellkonfiguration ein Dreieck erzeugt.

ten Messpunkt ausgewertet wird, wie in [176] beschrieben.

Die zweite Erweiterung soll die Approximation von scharfen Kanten verbessern. Kobbelts Ansatz dabei ist, Fälle, in denen die Isofläche einen Knick innerhalb der Zelle hat, zu identifizieren und eine alternative Triangulation zu verwenden. Dazu wird die Position des Knickes durch Verschneiden der Tangentialebenen auf beiden Seiten des Knickes bestimmt (siehe Abbildung 2.29 rechts). Zum Auffinden solcher Knickpunkte verwendet Kobbelt zwei einfache Heuristiken, die die Normalen der Punkte innerhalb einer Gitterzelle betrachten. Das erste Kriterium bestimmt, ob innerhalb einer Zelle ein scharfes Merkmal vorhanden ist. Dazu wird der maximale Öffnungswinkel θ zwischen den Normalen $\boldsymbol{n}_i$ betrachtet:

$$\theta = \max_{i.j}(\boldsymbol{n}_i^T \boldsymbol{n}_j)$$

Ist dieser größer als ein gegebener Schwellenwert θ_{sharp}, wird angenommen, dass ein Sonderfall vorliegt. Sobald ein Sonderfall detektiert wurde, wird bestimmt, ob es sich um eine Knickkante oder um einen Eckpunkt handelt. Dazu wird die Normale $\boldsymbol{n^*}$ der Ebene, die durch die Normalen mit dem größten Öffungswinkel zueinander aufgespannt wird, berechnet. Danach wird der maximale Abstand ϕ der Normalen zu dieser Ebene bestimmt:

$$\phi = \max_i |\boldsymbol{n}_i^T \boldsymbol{n}^*|$$

Ist ϕ kleiner als ϕ_{corner} liegt eine Kante innerhalb der Zelle vor, ansonsten eine Ecke. Die alternative Triangulation erfolgt, indem zunächst die Dreiecke zwischen dem Knickpunkt und den Vertices mit Vorzeichenwechsel bestimmt werden. Um die Knickkanten zu rekonstruieren, werden anschließend die Kanten zwischen zwei Dreiecken mit berechneten Knickpunkten getauscht (siehe Abbildung 2.30).

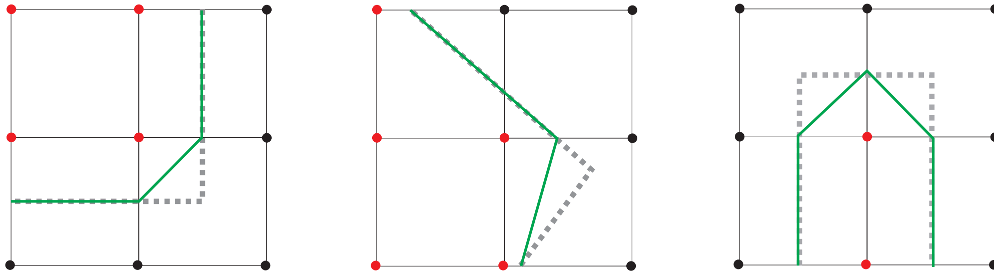


Abbildung 2.28: Probleme des Marching-Cubes-Verfahrens bei der Approximation von scharfen Kanten und Flächenverläufen innerhalb der Gitterzellen an einem zweidimensionalen Beispiel. Der Verlauf der zu approximierenden Isofläche ist grau dargestellt, die Rekonstruktion grün. Alle Flächenverläufe innerhalb der Zellen können mit den Standardmustern nicht genau wieder gegeben werden.

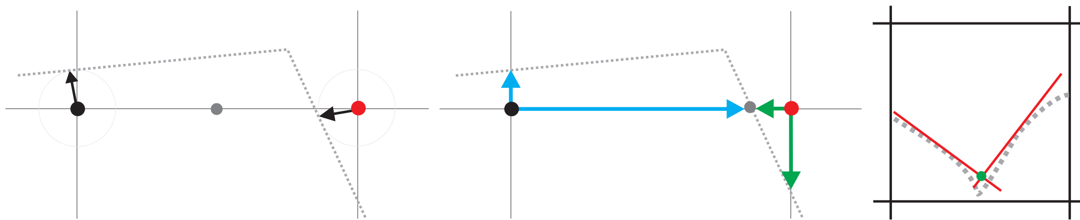


Abbildung 2.29: Interpolation der Vertices mit Kobbelts Distanzfunktion und Knickpunktfindung (nach [98]). Verwendet man lediglich die euklidische Distanz der Gitterecken, kann es zu verfälschten Interpolationspunkten kommen, wenn beide Punkte in etwa den gleichen Abstand zu Fläche haben. Kobbelt schlägt vor, für jeden Punkt den Abstand entlang der passenden Gitterhauptachse zu verwenden (Mitte). Dadurch lässt sich der exakte Schnittpunkt bestimmen. Um die Position eines Knickpunktes innerhalb einer Zelle zu approximieren, werden die Tangentialebenen der Flächen um den Knick verschnitten (rechts).

2.4 Weitere Rekonstruktionsverfahren

Neben dem Marching-Cubes-Ansatz gibt es noch weitere Ansätze, um aus 3D-Punktdaten Polygonnetze zu erzeugen. Diese werden im folgenden Abschnitt kurz vorgestellt.

2.4.1 Delaunay-Triangulation

Ein weiteres Verfahren, um eine Oberfläche aus Punktdaten zu erzeugen, ist, die Delaunay-Triangulation der Punktmenge zu berechnen [23]. Im Fall einer zweidimensionalen Punktmenge ist die Delaunay-Triangulation eine Zerlegung der Messdaten in planare Dreiecke, so dass sich innerhalb des Umkreises eines Dreiecks keine weiteren Messdaten mehr befinden. Auf 3D-Punktdaten ist die Delaunay-Triangulation eine Zerlegung der Punktwolke in Tetraeder. Die Dreiecksflächen der Tetraeder, die keinen benachbarten Tetraeder haben, bilden die Rekonstruktion der durch die Punktwolke repräsentierten Oberfläche. Da Oberflächenrekonstruktionen mit Delaunay-Verfahren die Hülle einer Punktwolke triangulieren, ist dieses Verfahren nur für geschlossene Modelle sinnvoll. Zur Bestimmung der Triangulation gibt es effiziente Implementie-

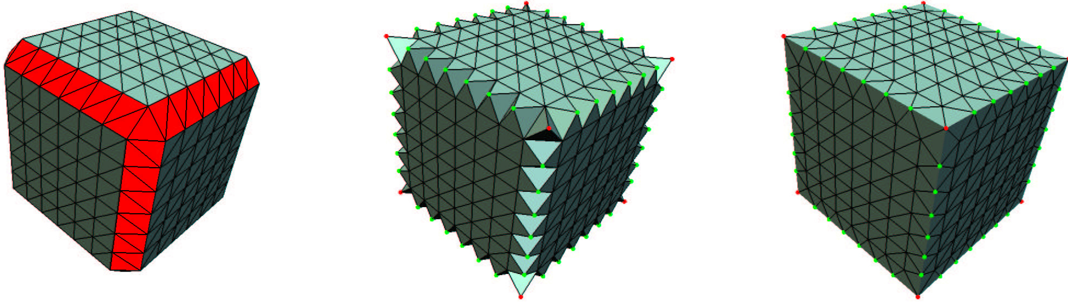


Abbildung 2.30: Approximation von scharfen Kanten bei "Extended Marching Cubes". Zunächst werden die Zellen mit Knickpunkten bestimmt (links). Anschließend werden alternative Approximationsmuster mit den Knickpunkten als Mittelpunkt berechnet (Mitte). Um den Verlauf der Schnittkanten zu rekonstruieren, wird ein Kantentauschen zwischen benachbarten Dreiecken mit Knickpunkten durchgeführt (rechts, entnommen aus [98].

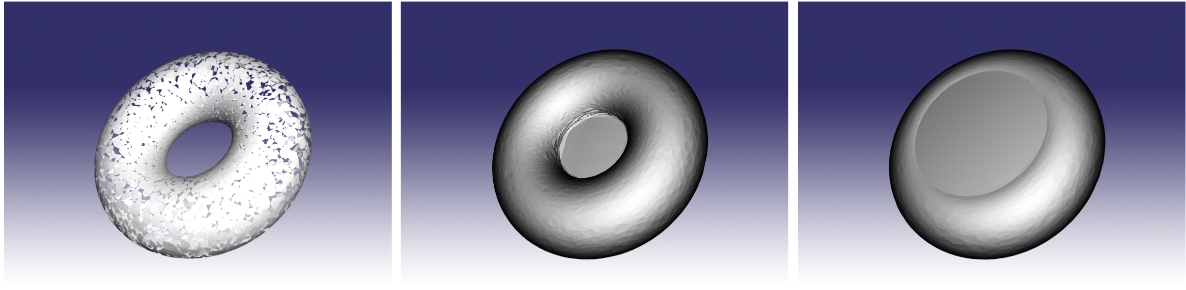


Abbildung 2.31: Beispiel für ein Alpha-Shape mit verschiedenen α-Werten. Zur Rekonstruktion wurde die CGAL-Implementierung [30] verwendet. Wird α zu klein (0.01) gewählt, entstehen Löcher in der Oberfläche (links). In der Mitte wurde ein automatisch von der Software ermittelter Schwellwert verwendet ($\alpha = 0.25$). Das rechte Bild zeigt die Rekonstruktion mit einem sehr hohen α-Wert (200).

rungen [43], z.B. in CGAL [30]. Eine bekanntes Verfahren, das robust gegen Ausreißer ist, ist der Crust-Algorithmus [7]. Eine Erweiterung davon ist unter dem Namen Power-Crust bekannt [8], von der es eine freie Implementierung gibt. Diese wird in Abschnitt 4.4.1 getestet.

Eine Erweiterung dieses Ansatzes bieten die so genannten α-Shapes [76]. Das α-Shape wird bestimmt, indem aus dem Volumen der konvexen Hülle der Messpunkte die Volumenanteile entfernt werden, in die eine Kugel mit dem Radius α passt, ohne dass sie Messpunkte beinhaltet oder berührt. Für $\alpha = 0$ bleiben demnach nur die Messwerte selbst übrig. Für $\alpha = \infty$ bleibt die konvexe Hülle erhalten, da kein Raum entfernt werden kann. Bei der Implementierung dieses Verfahrens werden aus praktischen Gründen alle Elemente der Delaunay-Triangulation entfernt, deren Umkreis einen Radius kleiner als α hat. Das Problem bei diesem Ansatz ist es, für einen gegebenen Datensatz den passenden Schwellenwert α zu finden, so dass die Daten gut durch die Triangulation approximiert werden, wie in Abbildung 2.31 zu sehen ist.

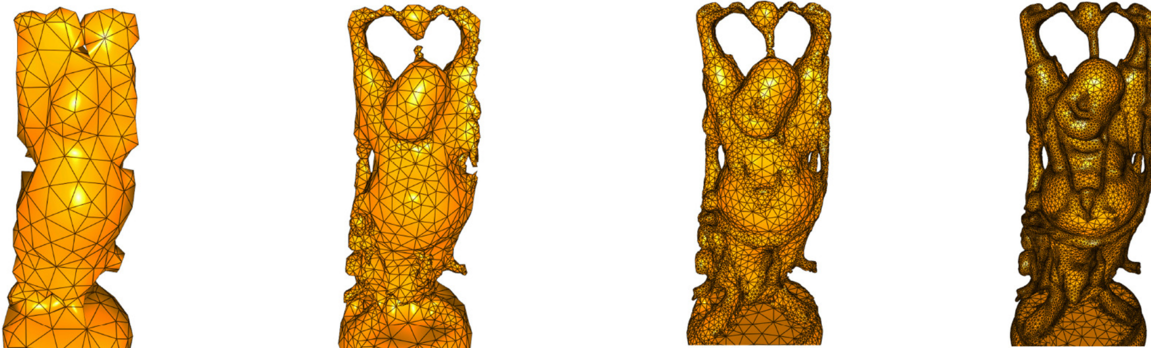


Abbildung 2.32: Oberflächenrekonstruktion mit "Growing Cell Struktures" am Beispiel des "Happy Buddha"-Datensatzes aus dem Stanford 3D Scanning Repository (544.000 Punkte). Das Verfahren beginnt mit einer sehr groben Approximation der Punktwolke und fügt sukzessize Vertices ein, bis die gewünschte Auflösung erreicht ist (von links nach rechts: 400, 2500, 10.000 und 50.000 Vertices). Die Laufzeit für die rechte Approximation betrug 9:15 Minuten [9].

2.4.2 Smart Growing Cells

Ein relativ neuer Ansatz zur Rekonstruktion von Dreiecksnetzen ist "Smart Growing Cells" [9]. Der Ansatz basiert auf der Idee, eine initial sehr einfache Hülle durch Einfügen und Verschieben von neuen Vertices sukzessive an die gegebenen Daten anzupassen (siehe Beispiel in Abbildung 2.32). Diese Anpassung erfolgt durch ein unüberwachtes Lernverfahren, das ein neuronales Netz generiert. Dafür stehen verschiedene Operatoren zur Verfügung, die es dem Verfahren erlauben, neue Vertices an geeigneten Stellen einzufügen und auch zuvor zusammenhängende Gebiete zu trennen, um eine bessere Anpassung an die Daten zu erreichen. Auf diese Art und Weise können mit dem Verfahren auch Szenen rekonstruiert werden, die aus vielen nicht zusammenhängenden Objekten bestehen.

Im Vergleich zu Marching-Cubes werden zur Approximierung keine Standardmuster verwendet. Die Vertexpositionen sind daher frei positionierbar und nicht an ein starres Raumgitter gebunden. Das dem Verfahren zugrunde liegende Lernverfahren sorgt dafür, dass neue Vertices nur dann eingefügt werden, wenn sie zu einer besseren Approximation führen. Ansonsten wird versucht, durch Verschieben bereits vorhandener Vertices eine bessere Approximation zu erreichen. Dieses Verhalten sorgt dafür, dass die Dreiecksdichte in den Rekonstruktionen lokal sehr unterschiedlich ist, dafür werden im Vergleich zum Marching-Cubes-Ansatz wesentlich weniger redundante Dreiecke generiert. Das wesentliche Manko dieses Algorithmus ist seine ausgesprochen lange Laufzeit (siehe auch Kapitel 4.4.1).

2.4.3 Ball-Pivoting

Der Ball-Pivoting-Algorithmus wurde 1998 von Bernardini vorgestellt [18]. Das Prinzip des Verfahrens ist sehr simpel: Zunächst werden drei zufällig ausgewählte Punkte einer Punktwolke zu einem Dreieck verbunden. Ausgehend von diesem Dreieck wird eine Kugel, deren Radius so groß

gewählt ist, dass sie nicht zwischen den Messpunkten durchfallen kann, von einer Kante des Dreiecks her über die Oberfläche gerollt. Befindet sich ein weiterer Punkt in der Nähe, kommt die Kugel in einer "Mulde" zum Liegen. In diesem Fall wird ein Dreieck zum neuen Auflagepunkt eingefügt. Wird kein weiterer Auflagepunkt gefunden, wird das Verfahren rekursiv von einer noch nicht benutzten Dreieckskante neu gestartet.

Mit diesem Verfahren lassen sich beliebige Oberflächen approximieren. Offensichtlich funktioniert Ball-Pivoting am besten auf Daten mit gleichmäßiger Punktverteilung. Für nicht gleichmäßig gesampelte Oberflächen besteht das Problem darin, einen geeigneten Kugelradius zu bestimmen. Bei zu kleinem Radius werden keine zusammenhängenden Flächen erzeugt, da die Kugel immer zwischen den Punkten "durchfällt". Bei zu groß gewähltem Radius können Details verloren gehen, da viele Punkte nicht von der Kugel berührt werden. Dieses Verfahren ist eher für gekrümmte Objekte geeignet, da scharfe Kanten nicht gut rekonstruiert werden können.

2.4.4 Direkte Triangulation auf Tiefenbildern

Für den Sonderfall, dass die Eingabedaten als Tiefenbild auf einer strukturierten Pixelmatrix vorliegen, können Oberflächen direkt trianguliert werden, indem benachbarte Pixel zu Dreiecken verbunden werden, sofern die Tiefenwerte sich nicht um mehr als einen vordefinierten Schwellwert unterscheiden. Dieses Verfahren kommt zum Beispiel in der "Zipper"-Software zum Einsatz, die im Rahmen des "Digital Michelangelo"-Projektes entwickelt wurde [107, 189]. Neben der Randbedingung, dass die Daten geordnet vorliegen müssen, liegt eine weitere Beschränkung des Verfahrens in der Tatsache, dass ein Mesh immer nur für ein einzelnes Bild erzeugt werden kann. Sobald mehrere Bilder registriert werden, geht die Ordnung auf den Daten verloren. Um Rekonstruktionen aus mehreren Bildern zu erzeugen, werden in der "Zipper"-Software die Meshes, die aus einzelnen Bildern gewonnen wurden, durch einen ICP-Ansatz miteinander registriert, indem Korrespondenzen zwischen den Vertices in sich überlappenden Teilmeshes gesucht werden. Sobald zwei Meshes auf diese Art und Weise registriert sind, werden die Dreiecke der sich überlappenden Regionen durch spezielle Clipping-Operationen konsistent in das Gesamtmesh integriert.

2.4.5 Rekonstruktion durch Model-Fitting

Eine Alternative zum Meshing stellt die Oberflächenrekonstruktion durch modellbasierte Verfahren dar. Dabei wird in den vorliegenden 3D-Daten nach vordefinierten, eventuell parametrierbaren Objekten gesucht. Die Punkte werden dann durch die vordefinierten Modelle ersetzt. Dieses Verfahren bietet sich an, wenn bereits a-priori Informationen über die in der Punktwolke vorhandenen Objekte vorhanden ist. Modellbasierte Rekonstruktion wird häufig im Bereich des Reverse-Engineering zur Erstellung von CAD-Modellen eingesetzt.

Eine Variante dieses Verfahrens ist, ebene Flächen in den Daten zu finden. Ebenen lassen sich in Punktwolkendaten sehr gut durch Hough-Transformationen erkennen [26]. Das Problem ist, dass sich mit diesem Verfahren keine Informationen über die Kontur einer ebenen Fläche gewinnen

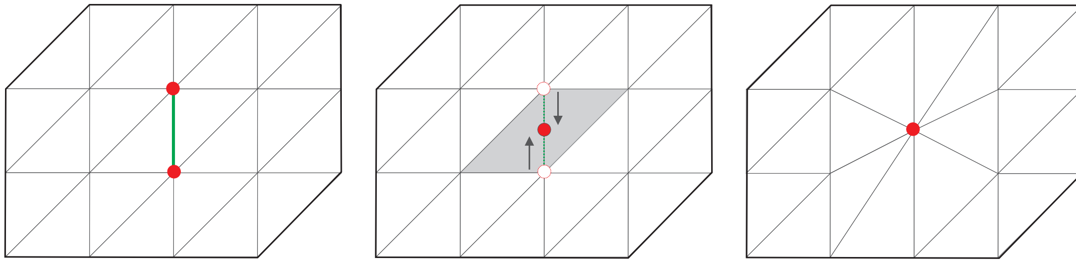


Abbildung 2.33: Meshoptimierung durch Edge-Collapsing. Wenn die grüne Kante mit den rot markierten Vertices zusammengezogen wird, werden die beiden grauen Driecke gelöscht. Das rechte Bild zeigt die Triangulation nachdem die Kante entfernt wurde.

lassen. Durch Berechnung der Schnittkanten von Ebenen lassen sich die Konturen von Objekten, die nahezu vollständig aus flächigen Anteilen bestehen, gut rekonstruieren [11, 161]. Solche Ansätze kommen oft bei der Rekonstruktion von Gebäudemodellen zum Einsatz. Modifizierte Hough-Transformationen lassen sich auch zum Suchen von Kugeln und anderen parametrierbaren Flächen verwenden, allerdings steigt mit der Anzahl der Modellparameter der benötigte Rechenaufwand [86].

Andere Verfahren rekonstruieren Oberflächen durch Einpassen von einfachen parametrierbaren Modellen wie Quadern, Zylindern oder Kugeln, die dann an die vorliegenden Daten gefittet werden. In [64] wird beispielweise ein probabilistisches Verfahren zur Rekonstruktion komplexer Objekte aus solchen Grundformen vorgestellt. Zur Berechnung der Modellparameter und -positionen werden in der Regel Least-Squares-basierte Fehlerminimierungsansätze verwendet. In [75] werden synthetische Punktwolken, die aus gesampleten CAD-Vorlagen erzeugt wurden, mittels ICP in die Punktwolken eingepasst, um Instanzen dieser Modelle zu finden.

Der große Vorteil von modellbasierten Verfahren ist, dass sie relativ unempfindlich gegen Rauschen und Löcher in den Messdaten sind, solange die Modelle mit den Daten gut übereinstimmen. So wird in [169] ein modellbasierter Ansatz vorgestellt, mit dem sich Objekte, die nur teilweise erfasst wurden, rekonstruieren lassen. Ein weiterer Vorteil ist, dass normalerweise eine gute Flächenrepräsentation durch die Modelle bereits von vornherein gegeben ist und nicht aus den Daten rekonstruiert werden muss. Jedoch sind modellbasierte Ansätze auf eine bestimmte Anwendungsdomäne beschränkt, in der nur eine kleine Anzahl an möglichen Objekten vorkommt.

2.5 Meshoptimierung

Die mit dem Marching-Cubes-Verfahren erstellten Meshes bestehen typischerweise aus wesentlich mehr Dreiecken als zur Wiedergabe der zugrunde liegenden Topologie notwendig wären. Zur Reduzierung der Dreieckszahl existieren verschiedene Ansätze. Viele basieren darauf, suksessive redundante Dreiecke aus den Meshes zu entfernen. Um keine Löcher in zuvor geschlossene Flächen zu reißen, werden die Dreiecke nicht direkt entfernt, sondern es werden Kanten durch Edge-Collapse Operationen entfernt. Dabei werden die zwei Vertices einer Kante gelöscht und in der

Verzeigerung durch deren Mittelpunkt ersetzt (siehe Abbildung 2.33). Pro Edge-Collapse werden dementsprechend zwei Dreiecke entfernt.

Um zu bestimmen, welche Dreiecke entfernt werden können, gibt es verschiedene Metriken und Heuristiken. Die einfachste Variante ist, zuerst die kürzesten Kanten zu löschen, da der Fehler der durch die Neupositionierung des neuen Vertex bei der Edge-Collapse-Operation klein ist. Diese Heuristik betrachtet aber nicht den Fehler, den das Entfernen der Kante an der ursprünglichen Geometrie verursacht. Eine Metrik zur Bestimmung dieses Fehlers Δ wurde von Garland und Heckbert definiert [65].

$$\Delta(\boldsymbol{v}) = \sum_{\boldsymbol{p} \in \text{planes}(\boldsymbol{v})} (\boldsymbol{p}^T \boldsymbol{v})^2$$

Wobei $\boldsymbol{p} = [abcd]^T$ die Ebene ist, die durch die Gleichung $ax + by + cz + d = 0$ mit $a^2 + b^2 + c^2 = 1$ definiert ist, und $\text{planes}(\boldsymbol{v})$ sind die Ebenen, in denen die angrenzenden Dreiecke inital liegen. Der Fehler ist immer auf die ursprüngliche Geometrie bezogen, d.h. initial sind alle Fehler 0, da der Vertex in jeder gegebenen angrenzenden Ebene liegt. Durch Propagieren der Fehler nach einer Edge-Collapse Operation wird dieser Wert steigen, je mehr der Vertex sich von der Ursprungskonfiguration entfernt. Mit Hilfe dieser Metrik ist es darüber hinaus möglich, eine optimale Position für den Ersatzvertex bei Edge-Collapse zu bestimmen, anstatt den Mittelpunkt der Kante zu verwenden. Es werden nun nach und nach die Kanten entfernt, deren Kollabieren einen minimalen Fehler erzeugt. Dabei kann entweder der Fehler einzelner Vertices oder der aufsummierte Fehler eines Dreiecks als Kriterium verwendet werden.

Eine weitere Metrik wurde von Melax definiert [120]. Die Kosten $\text{cost}(\boldsymbol{u}, \boldsymbol{v})$, die Kante zwischen den Vertices $\boldsymbol{u}$ und $\boldsymbol{v}$ zu entfernen, wird mit folgender Formel berechnet:

$$\text{cost}(\boldsymbol{u}, \boldsymbol{v}) = \|\boldsymbol{a}, \boldsymbol{b}\| \max_{a \in T_u} \left\{ \min_{b \in T_{uv}} \{1 - \boldsymbol{n}_a \cdot \boldsymbol{n}_b\} \right\}$$

In dieser Gleichung sind T_{uv} die Dreiecke, die sowohl $\boldsymbol{u}$ als auch $\boldsymbol{v}$ enthalten und T_u diegenigen, die nur $\boldsymbol{u}$ enthalten. Der Term $\min\{1 - \boldsymbol{n}_a \cdot \boldsymbol{n}_b\}$ bestimmt die beiden an die Kante angrenzenden Dreiecke, deren Normalen $\boldsymbol{n}_a$ und $\boldsymbol{n}_b$ den größten Wikel zu einander haben. Dieser Term wird insbesondere dann groß, wenn einer der Vertices auf einer Kante liegt. Dies macht Sinn, da ein Edge-Collapse auf einer Kante die Kontur des Meshes verändern kann. Die Gewichtung mit der Länge $\|\boldsymbol{a}, \boldsymbol{b}\|$ sorgt dafür, dass bevorzugt kurze Kanten gelöscht werden. Der Vorteil dieser Fehlerfunktion ist, dass sie sehr schnell zu berechnen ist. Sie kann daher auch bei Computerspielen zur Online-Reduktion von Meshes verwendet werden, um z.B. weiter entfernte Objekte weniger detailgetreu zu rendern.

Ein weiterer Ansatz besteht darin, neben den redundanten Elementen auch die Vertexpositionen nachträglich zu optimieren. Hoppe berechnet dazu eine globale „Energiefunktion", die durch Entfernen von Elementen und Verschieben von Vertices minimiert wird [83]. Diese Funktion besteht aus drei Termen:

$$E = E_{Dist} + E_{Rep} + E_{Spring}.$$

E_{Dist} ist der quadratische Abstand aller Vertices von den ursprünglichen Messpunkten. Der Term E_{Rep} ist proportional zur Anzahl der Vertices im Netz und bestraft Repräsentationen mit vielen

Vertices. Der Term E_{Spring} entspricht einer Art „Federenergie” mit einer vom Benutzer festgelegten Federkonstanten, die garantieren soll, dass sich das Netz tendenziell „zusammenzieht”. So wird verhindert, dass Vertices weit nach außen verlagert werden, und sichergestellt, dass ein Minimum existiert.

2.6 Normalenschätzung für 3D-Punktwolken

Ein wichtiger Schritt bei der Rekonstruktion von Oberflächen aus Punktdaten ist die Berechnung von Normalenvektoren für die gesampleten Oberflächenpunkte, da sie die lokale Orientierung der repräsentierten Flächen definieren. Die Qualität der geschätzten Normalen hat dabei einen entschiedenen Einfluss auf die Qualität der Rekonstruktion. Bei Marching-Cubes-basierten Rekonstruktionsverfahren werden die Normalen z.B. benötigt, um den Abstand der Gitterpunkte zur Oberfläche zu bestimmen. Stehen die Normalen nicht genau senkrecht auf der zugrunde liegenden Fläche, wird der berechnete Abstand fehlerhaft sein, was wiederum eine ungenaue Interpolation der Vertices der Approximationsmuster nach sich zieht. Darüber hinaus ist es entscheidend, die berechneten Normalen konsistent zu orientieren. Es muss also nicht nur die qualitative Ausrichtung stimmen, sondern es müssen lokal alle Normalen von der Fläche weg zweigen.

Viele Verfahren basieren darauf, Ausgleichsebenen $\mathcal{P}$ der Form

$$\mathcal{P} := n_x x + n_x y + n_z z + d = 0$$

an k nächste Nachbarn des betrachteten Punktes zu fitten und deren Normalenvektor zu übernehmen. Dieses Problem ist analog zur Bestimmung des Koefizientenvektors $\boldsymbol{c}, \boldsymbol{c} \neq \boldsymbol{0}$ im linearen Gleichungssystem

$$\begin{bmatrix} \boldsymbol{b}(\boldsymbol{x_i})^T \\ \vdots \\ \boldsymbol{b}(\boldsymbol{x_k})^T \end{bmatrix} \boldsymbol{c} = \boldsymbol{0}$$

mit $\boldsymbol{b} = [x_i y_i z_i 1]^T$ und $\boldsymbol{c} = [abcd]^T$ für die k nächsten Nachbarn $\boldsymbol{x_i}$ [135]. Um sicher zu stellen, dass bei der Lösung dieses Gleichungssystems nicht die triviale Lösung $\boldsymbol{c} = \boldsymbol{0}$ gefunden wird, muss man die zusätzliche Randbedingung $a^2 + b^2 + c^2 = 1$ einführen. Alternativ kann die Lösung in einem geeigneten lokalen Koordinatensystem mit einer Ebenenrepräsentation in Form $f(x, y) = z$ berechnet werden [192]. Die Gleichungssysteme sind in den meisten Fällen überbestimmt, da mehr Punkte in der Nachbarschaft liegen als freie Parameter vorhanden sind. In der Praxis werden zur Lösung solcher Probleme numerische Implementationen von SVD-Zerlegungen verwendet [83]. Darüber hinaus existieren Arbeiten, die das Approximationsproblem mit Hilfe von Hauptkomponentenanalysen lösen [16] oder RANSAC-basierte Verfahren [162] verwenden, um die beste Ausgleichsebene zu bestimmen. Um das Herausmitteln von Kanten mit solchen Verfahren abzumildern, besteht die Möglichkeit, die Punkte der k-Nachbarschaft nach ihrem Abstand zu gewichten [122].

Neben linearen Approximationen gibt es auch Ansätze mit quadratischen Polynomen [148]. Das Problem dabei ist, dass die Anzahl der zu bestimmenden Parameter steigt, was einerseits die

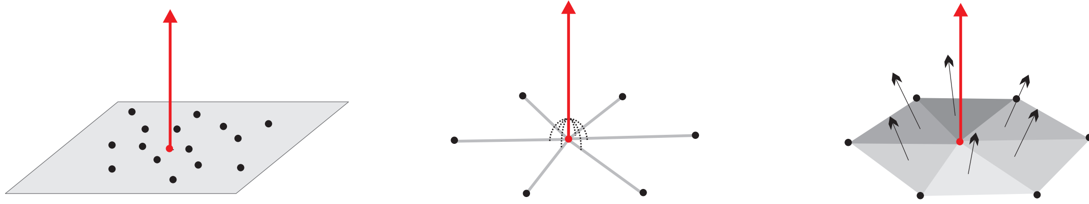


Abbildung 2.34: Verschiedene Verfahren zur Berechnung von Flächennormalen (nach [97]). Links: Normalenschätzung durch Interpolation einer lokalen Ausgleichsebene. Mitte: Bestimmung der Normalen durch Maximierung des Winkels zu den Kanten zu den nächsten Nachbarn. Rechts: Mittelung der Normalen der Delaunay-Triangulation der lokalen Nachbarn.

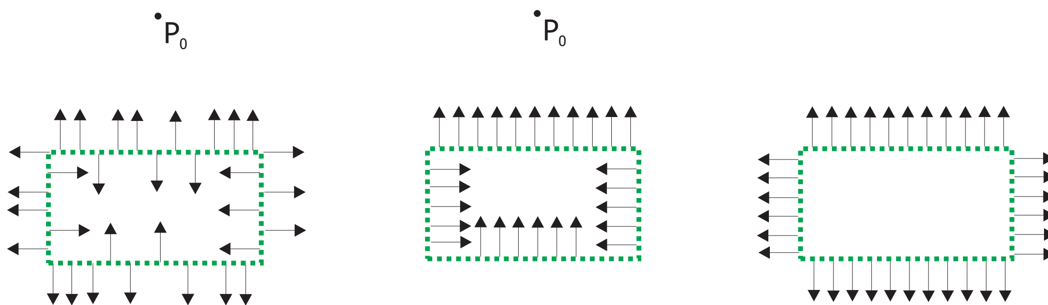


Abbildung 2.35: Ausrichtung von inkonsitenten Normalen (links) zu einem Referenzpunkt (P_0). Nach Ausrichtung kann es zur in der Mitte gezeigten Konfiguration kommen. Für eine korrekte Rekonstruktion wird eine konsiste Ausrichtung wie rechts dargestellt benötigt.

Rechenzeit erhöht, andererseits dazu führen kann, dass das Gleichungssystem bei zu wenigen verwendeten Nachbarn unterbestimmt ist. Andere Verfahren basieren darauf, die Winkel zwischen den Tangentialvektoren zu den nächsten Nachbarn oder den Dreiecken der Delaunay-Triangulation [72] der k-Nachbarschaft zu maximieren [97] (siehe Abbildung 2.34). Für die Daten von 3D-Kameras gibt es Verfahren, die zur Berechnung der Normalen die regelmäßige Anordnung der Daten in der Bildmatrix der Kamera ausnutzen [81].

Die gerade beschriebenen Ansätze erzeugen in der Regel keine konsistent ausgerichteten Normalen. Ihre Ausrichtung ändert sich mit der Reihenfolge, in der die Punkte betrachtet werden. Bei bekannter Aufnahmeposition kann das Problem gelöst werden, indem die berechneten Normalen in Richtung der Sensorposition gedreht werden. Diese einfache Heuristik funktioniert für einzelne Datensätze gut, da die Sensoren normalerweise immer den ersten Auftreffpunkt mit einer Oberfläche bestimmen. Die Objekte werden also alle aus der gleichen Richtung betrachtet, so dass die Normalenorientierung vorgegeben ist. Problematisch wird es, wenn "echte" 3D-Daten vorliegen, also auch Rückseiten von Objekten erfasst wurden. In diesem Fall müssen die Normalen global konsistent orientiert werden (siehe Abbildung 2.35). Das linke Bild zeigt die inkosistent orientierten Normalen. Wurden Objekte von mehreren Seiten erfasst, und die berechneten Nor-

malen anschließend in Richtung des Bezugspunktes P_o ausgerichtet, können Konfiguration wie im mittleren Bild enstehen. Dies kann unter Umständen zu Löchern und falschen Dreiecken in der Rekonstruktion führen. Für eine korrekte Rekonstruktion wird eine konsistene Ausrichtung wie im rechten Bild benötigt.

In überwiegend konkaven Szenen (z.B. einzelnen Räumen) kann man das Problem approximativ durch Ausrichten der Normalen zum Mittelpunkt der Szene lösen. Für eine global konsistente Lösung existieren graphenbasierte Verfahren wie das von Hoppe [83], die allerdings sehr rechenintensiv sind. Ein weiterer Weg, das Problem in Szenen, die aus mehren Scans bestehen, zu lösen, ist die Normalen für jeden Teilscan einzeln zu berechnen und nach der Registrierung zu interpolieren. Auf diese Art und Weise lassen sich approximativ konsistente Orientierungen bestimmen, ohne auf Hoppe's Verfahren [83] zurückgreifen zu müssen.

Kapitel 3

Generierung optimierter Polygonkarten

In diesem Kapitel werden die im Rahmen dieser Arbeit implementierten Verfahren zur Generierung polygonaler Karten sowie die Struktur der entwickelten Software beschrieben. Im ersten Abschnitt wird die Auswahl der implementierten Algorithmen erörtert. Der zweite Abschnitt widmet sich der Beschreibung der Struktur der Software und der Definition von Schnittstellen für die Interaktion der einzelnen Komponenten. Die darauf folgenden Sektionen beschreiben die konkreten Implementierungen der Verfahren zur Schätzung von Punktnormalen, Polygonalisierung mittels Marching Cubes, Meshoptimierung und Erzeugung von Texturen für die generierten Meshes. Der letzte Abschnitt vergleicht die Vor- und Nachteile der einzelnen Verfahren.

3.1 Anforderungen an das Rekonstruktionsverfahren

Die im Rahmen dieser Arbeit erzeugten Polygonkarten sollen in erster Linie für fahrende mobile Roboter erzeugt werden. Als Testplattform sollen hauptsächlich Kurt-Roboter [185] mit verschiedenen 3D-Sensoren dienen. Die Kurt-Roboter bestehen aus einem 6-rädrigen Chassis mit Differenzialantrieb, das mit verschiedenen Aufbauten versehen werden kann. Momentan gibt es zwei Versionen mit 3D-Laserscannern: Kurt3D und Kurt360. Kurt3D verfügt über einen nickenden Sick LMS200 Laserscanner, auf Kurt360 wird der SICK-Scanner mittels einer Rotationseinheit gedreht. Zwei Beispielscans dieser Systeme wurden bereits in Abbildung 2.3 gezeigt. Bilder der Plattformen und die technischen Daten der Laserscanner zeigt Abbildung 2.2. Neben den 3D-Laserscannern stehen auch Kinect-Kameras zur Verfügung. Die Punktwolken, die mit diesen Systemen aufgenommen werden, sollen die Datengrundlage zur Evaluation des Rekonstruktionsverfahrens bilden. Um die Skalierung auf höher Aufgelöste Daten zu testen, sollen aber auch hochaufgelöste terrestrische Laserscans betrachtet werden.

Die Punktwolken, die mit rotierenden Laserscannern und Kinect-Kameras aufgenommen werden, haben eine geringere Punktdichte und wesentlich höheres Messrauschen als dedizierte terrestrische Laserscanner. Da sich die Technik im Bereich des 3D-Laserscanning immer weiter entwickelt und auch hochauflösende Scanner immer preiswerter werden, ist zu erwarten, dass diese Sensoren

zukünftig auch im Bereich der Robotik öfters zum Einsatz kommen werden. Es gibt bereits erste Roboter wie Irma3D [48], auf denen hochauflösende Scanner zum Einsatz kommen. Außerhalb der mobilen Robotik werden derartige Scanner u.a. auf Autos eingesetzt, um 3D-Punktwolken ganzer Stadtteile aufzunehmen, wie es z.B. die Firmen Google [71] und Velodyne [109] praktizieren.

Bei sehr hohen Auflösungen tritt das Problem der Verwaltung der großen Datenmengen in den Vordergrund. Zum Beispiel kann ein einzelner Scan eines Leica HDS 6000 Laserscanners in der zweit höchsten Auflösungsstufe bereits 80 Millionen Punkte enthalten [104]. In 8 GB Hauptspeicher lassen sich bei einfacher Genauigkeit (keine Remissions- oder Farbinformationen, 12 Byte pro Punkt) so lediglich 8 Scans halten. Deshalb werden die Daten zur Weiterverarbeitung in der Regel ausgedünnt oder in speziellen komprimierten Datenstrukturen abgelegt, um ein Caching zu realisieren, so dass nur die Punkte in den Hauptspeicher geladen werden, die gerade benötigt werden (z.B. Pointools Vortex Engine [153]). Der Einsatz hochauflösender Scanner ist auch für die Erstellung von Umgebungskarten für die mobile Robotik interessant. Aufgrund der großen Reichweite (bis zu 700m) lassen sich bereits mit wenigen Scanpositionen nahezu vollständige 3D-Punktwolkenmodelle auch weitläufiger Umgebungen oder kompletter Gebäude erstellen. Daher soll das Rekonstruktionsverfahren auch an hoch aufgelösten Laserscans getestet werden. Interessant ist dabei vor allem, welche Datenkompression der Eingangsdaten sich mit einer polygonalen Darstellung erzielen lässt und wie genau die erstellten Approximationen die mit dem Laserscanner aufgenommene Geometrie wiedergeben. Aus diesen Anwendungsszenarien lassen sich Anforderungen an die zu entwickelnde Software ableiten. Diese werden im folgenden Abschnitt konkretisiert.

3.1.1 Zielsetzungen

Die Umgebungsrekonstruktionen sollen durch Dreiecksnetze repräsentiert werden. Im Gegensatz zu anderen polygonalen Netzstrukturen, z.B. Quads, sind die Flächenelemente minimal, d.h. ein Dreieck ist das kleinste zu repräsentierende Flächenelement. Das bietet den Vorteil, dass sich sowohl ebene als auch gekrümmte Flächen in der selben Datenstruktur durch Anpassung der Größe der Dreiecke sehr genau darstellen lassen. Eine polygonale Flächendarstellung hat gegenüber parametrisierten Darstellungen den Vorteil, dass sich auch die Konturen der Flächen explizit repräsentieren lassen. Die Information über die Grenzen einer Fläche spielt in der mobilen Robotik eine wichtige Rolle. Zum Beispiel ist das Wissen, wo ein befahrbarer Bereich endet, für einen mobilen Roboter eine wichtige Information zur Planung eines sicheren Pfades von seiner derzeitigen Position zu einem Zielpunkt. Durch eine rein parametrisierte Darstellung, lässt sich zwar kodieren, dass es eine Ebene in Bodennähe gibt, die durch eine Menge von Datenpunkten gestützt wird, aber nicht, wie die Kontur der ebenen Fläche beschaffen ist, insbesondere wo sie endet. Ein weiterer Vorteil von Dreiecksnetzen ist, dass sich durch verzeigerte Datenstrukturen leicht Nachbarschaftsbeziehungen kodieren lassen, was bei der Entwicklung von effizienten Algorithmen, z.B. zum Auffüllen von Scanschatten oder zur Meshoptimierung, von Vorteil ist. Weiterhin lassen sich die einzelnen Polygone in den Karten leicht mit zusätzlichen Attributen wie Farbinformationen und Texturen versehen.

Texturen bieten die Möglichkeit, Flächen mit Bitmap-Mustern zu versehen. In diesen Mustern können verschiedene Arten von Informationen hinterlegt werden. Die bekannteste Verwendung ist die Nutzung von Farbbildern, die auf die erzeugten Polygone projiziert werden. Neben der Möglichkeit, die Umgebungen fotorealistisch zu rendern, kann die Generierung von Texturen auch für den Einsatz in der Robotik interessant sein. Auf mobilen Robotern werden häufig auch Kameras eingesetzt. Aus den Kameradaten extrahierte Bildmerkmale werden dann z.B. zur Lokalisierung eingesetzt (*Visual SLAM*) [41]. Durch die Verknüpfung von Bilddaten mit der Umgebungsgeometrie lassen sich den extrahierten Features direkt absolute 3D-Koordinaten zuweisen. Sind die Polygone in der Karte semantisch klassifiziert, lassen sich weitere Beziehungen zwischen den Features und der Umgebungsgeometrie herleiten, z.B. "Feature XY befindet sich an an einer Wand". Soweit vorhanden, sollen Texturen daher wenn möglich aus vorhandenen Farb- oder Remissionswerten der 3D-Punktwolken erzeugt und auf die Polygonkarten übertragen werden.

Texturierte Dreiecksnetze werden von vielen Datenaustauschformaten unterstützt, so dass die erstellten Karten leicht in verschiedenen Softwarepaketen eingesetzt werden können. Durch die Unterstützung geeigneter Austauschformate wie Wavefront OBJ oder PLY können die erzeugten Karten direkt weiter benutzt werden. Zum Beispiel unterstützt die Robotersimulationsumgebung Gazebo die Formate STL und Collada. Werden die erzeugten Karten in einem dieser Formate exportiert, können sie als Umgebungsrepräsentation in dieser Simulationsumgebung verwendet werden [99, 129]. Eine zeitaufwändige manuelle Erstellung entfällt. Gazebo bietet zudem Interfaces zu den Robotersteuerungs-Middlewares ROS und Player, so dass über diese Schnittstelle die Karten direkt in Verbindung mit bestehender Roboterkontrollsoftware genutzt werden können. Darüber hinaus existiert ein Treiber für Player, der im Wavefront OBJ-Format vorliegende 3D Polygonkarten zur Lokalisierung und Hindernisvermeidung verwendet [208].

Sollen die automatisch generierten Karten für derartige Anwendungen verwendet werden, ist es von besonderer Bedeutung, dass die aufgenommene Umgenung korrekt wiedergegeben wird. Bedingt durch Sensorrauschen und Registrierungsfehler ist davon auszugehen, dass die Eingangsdaten nicht hunderprozentig genau sind. Die Rekonstruktion soll daher robust gegen Messrauschen und Ausreißer in den Eingangsdaten sein. Moderne Lokalisierungsverfahren erlauben eine lokalisierung des Roboters auf wenige Zentimeter genau [57]. Dies liegt im Bereich des Sensorrauschens von in der Robotik üblicherweise verwendeten 3D Sensoren. Für den Einsatz zur Lokaliserung und Pfadplanung sollten die Karten eine Genaugkeit in diesem Bereich aufweise. Der maximale Fehler einer Karte soll daher weniger als 5 cm sein, um eine sichere Navigation zu ermöglichen. Für andere Anwendungen, z.B. Greifen von Objekten, wird eine höhere Genauigkeit benötigt, in der Regel unterhalb von 1 cm.

Radgetriebene Roboter kommen in der Regel innerhalb von Gebäuden oder auf be- und umbauten Außenflächen zum Einsatz. In dieser Art von Anwendungsszenario besteht die Umgebung zu großen Anteilen aus ebenen oder nahezu ebenen Flächen, die durch die Architektur der Bebauung definiert werden. Zudem weisen in Innenbereichen viele Objekte, vor allem Möbel, ebenfalls ebene Oberflächen auf. Diese Beobachtungen legen nahe, die Planarität der Umgebungen auszunutzen, um eine möglichst kompakte Darstellung zu realisieren. In Bereichen, in denen viele ebene Anteile vorhanden sind, soll daher eine möglichst große Datenkompression erreicht werden.

In Bereichen, in denen wenig ebene Anteile vorhanden sind, muss die Anzahl der Polygone in den Rekonstruktionen höher sein, um auch gekrümmte Oberflächen in ausreichender Auflösung wiedergeben zu können. In bebauten Umgebungen treffen ebene Flächen oftmals in rechten Winkeln aufeinander. Auch diese sollen in den Rekonstruktionen präzise wiedergegeben werden, um zu gewährleisten, dass diese markanten Umgebungsmerkmale in den Roboterkarten repräsentiert werden. Solche geometrischen Merkmale werden z.B. zur Lokalisierung benötigt.

Die Karten sollen in möglichst kurzer Laufzeit auch auf mobiler Hardware erstellt werden können. Das automatische Registrieren von 3D-Daten ist heute bereits in Echtzeit möglich, daher sollen auch die auf diesen Daten basierenden Karten ohne größere Verzögerung zur Verfügung stehen. Für Daten von 3D-Kameras mit Bilderholraten von 30 Bildern pro Sekunde wird diese Anforderung nicht ohne Weiteres zu erfüllen sein. Die Rekonstruktion zwischen zwei Laserscans, die jeweils einige Sekunden benötigen, scheint aber aufgrund der Erfahrungen aus Vorarbeiten realistisch zu sein [195]. Eine Möglichkeit, die Verfahren zu beschleunigen, besteht darin, die Algorithmen zu parallelisieren. Mehrkernprozessoren sind auch auf Laptops bereits Standard. Aus diesem Grund soll bei der Implementierung der Verfahren besonderer Wert auf die Parallelisierung gelegt werden, um die vorhandenen Ressourcen möglichst gut auszunutzen. Auf eine gezielte Optimierung für Grafikprozessoren soll in dieser Arbeit verzichtet werden. GPU-basierte Implementierungen können die benötigte Zeit für einzelne Schritte, z.B. Normalenschätzung, zwar deutlich verringern, jedoch sind dafür leistungsfähige Grafikkarten mit sehr viel Grafikspeicher erforderlich, um auch größere Datensätze verarbeiten zu können. Zudem ist man bei der Implementierung durch die vorgegebene Hardwarestruktur stark eingeschränkt. Wesentliches Ziel dieser Arbeit soll es sein, die für die Lösung der Problemstellung geeigneten Verfahren zu identifizieren, zu implementieren und zu testen. Als Testsystem soll daher ein PC mit Mehrkernprozessor und 8 bis 16 GB Arbeitsspeicher dienen. Diese Anforderungen werden auch von handelsüblichen Laptops erfüllt.

Die Kartenerstellung soll in drei Schritten erfolgen [160, 197, 199]: Erzeugung der Eingabedaten, Erstellen einer initialen Meshrepräsentation und Meshoptimierung. Dabei wird davon ausgegangen, dass die Eingabedaten registriert vorliegen, d.h., die polygonale Rekonstruktion erfolgt global auf Basis dieser Daten.

3.1.2 Zu implementierende Funktionalitäten

Die gestellten Anforderungen an die zu generierenden Karten legen nahe, ein Marching-Cubes-basiertes Verfahren mit einer geeigneten Distanzfunktion für die Rekonstruktion zu verwenden. Dieser Ansatz bietet diverse Vorzüge gegenüber anderen Verfahren. Zum einen können mit Marching Cubes beliebige Oberflächen rekonstruiert werden. Im Gegensatz zu modellbasierten Ansätzen ist man nicht auf durch die verwendeten modellierten Klassen von Objekten beschränkt. Zum anderen ist die Marching-Cubes-basierte Rekonstruktion aufgrund des Einsatzes der vorberechneten Flächenmuster sehr schnell und ist darüber hinaus gut parallelisierbar, da die Flächenmuster unabhängig voneinander in den Voxeln des Rekonstruktionsgitters erzeugt werden. Durch die Verwendung von Hashing-Verfahren oder Octrees ist der Aufbau des Rekonstruktionsgitters sehr speichereffizient möglich, indem nur die Raumzellen erstellt werden, die auch Datenpunkte

enthalten [66, 93, 195]. Die Berechnung der Delaunay-Triangulation hingegen ist sehr rechenaufwändig und für nicht geschlossene Geometrien nicht sinnvoll. Aufgrund der in der Literatur zu findenden Laufzeitangaben (z.B. [8]) ist davon auszugehen, dass diese Verfahren darüber hinaus eine für den angestrebten Einsatzbereich zu lange Laufzeit haben werden, insbesondere bei großen Datensätzen. Das selbe gilt für Growing-Cell-Structures [9]. Da Alpha-Shapes eine Variante der Delauney-Triangulation sind und zudem je nach Umgebung einen anderen Schwellwert zur Flächenextraktion benötigen, entspricht auch dieses Verfahren nicht den gestellten Ansprüchen an Geschwindigkeit und Robustheit. Des Weiteren hinaus sind diese Verfahren aufgrund ihrer Komplexität im Gegensatz zu Marching Cubes nicht einfach zu parallelisieren.

Unabhängig von der gewählten Rekonstruktionsmethode wird es notwendig sein, die erzeugten Meshes zu optimieren, da alle vorgestellten Verfahren auf ebenen Flächen zu viele Dreiecke produzieren. Die Randbedingung der Planariät soll durch Region-Growing basierte Optimierung ähnlich zu [195] realisiert werden. Es ist damit zu rechnen, dass die Rekonstruktionen aufgrund von Messrauschen und Abschattungen Artefakte wie Löcher, freischwebende Dreiecke, Unebenheiten usw. enthalten werden. Unebenheiten auf ebenen Flächen lassen sich durch Verschieben der generierten Vertices in die gemeinsame Ebene ausgleichen. Nach der Berechnung der Ausgleichsebenen sollen scharfe Kanten durch Verschneidung dieser Ebenen optimiert werden. Löcher bestehen aus Konturen innerhalb der Ebenen und lassen sich durch Edge-Callapse-Operationen schließen. Einzelne Dreiecke oder kleine Flächengruppen lassen sich durch Region-Growing basiertes Clustering erkennen [195]. Zur Implementierung dieser Operationen ist es notwendig, die Dreiecksnetze in einer verzeigerten Datenstruktur abzulegen. Dazu bietet sich eine Halbkantendarstellung an, da diese gemäß Tabelle 2.1 das günstigste Laufzeitverhalten bezüglich lokaler Suchen in Dreiecksnetzen hat. Polygonnetze werden in Austauschdatenformaten wie PLY oder OBJ nicht verzeigert, sondern in Form von indizierten Buffern abgelegt. Daher wird es erforderlich sein, zwischen verzeigerten Datenstrukturen und Buffer-Darstellungen zu konvertieren.

Im Rahmen dieser Arbeit soll evaluiert werden, welche Marching-Cubes-Varianten unter den gegebenen Anforderungen die besten Ergebnisse liefern. Dazu sollen neben dem standard Marching Cubes Algorithmus auch Marching Tetrahedrons implementiert werden. Zwar generiert diese Variante mehr Dreiecke, dafür sind die erzeugten Flächen aber auf jeden Fall lokal konsistent [151, 173]. Darüber hinaus lässt sich der Algorithmus leicht implementieren, da insgesamt nur 16 Flächenmuster betrachtet werden müssen, was die Erstellung der Triangulationstabellen im Gegensatz zu kubischen Gittern erleichtert. Da die zu rekonstruierende Umgebung viele scharfe Kanten aufweist, soll auch Extended Marching Cubes nach Kobbelt [98] zum Vergleich herangezogen werden. Um Laufzeit und Speicherverbrauch der Verfahren objektiv vergleichen zu können, sollen diese in einem gemeinsamen Framework, "Las Vegas Surface Reconstruction Toolkit" (LVR) [196], implementiert werden, dessen Architektur im folgenden Abschnitt vorgestellt wird.

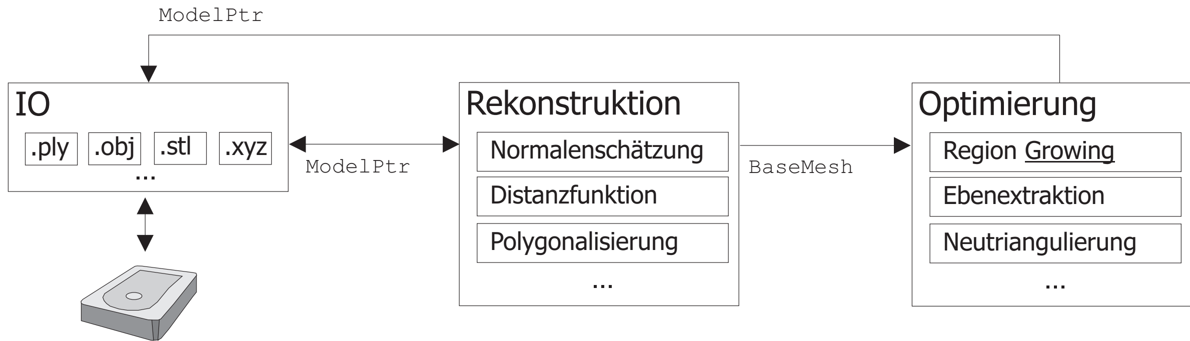


Abbildung 3.1: Architektur des Las Vegas Surface Reconstruction Toolkits. Die IO-Bibliothek bildet die Schnittstelle zum Lesen und Schreiben von Daten. Das Rekonstruktionsmodul beinhaltet die Algorithmen, die zur Mesherzeugung benötigt werden. Die Verfahren zur Meshoptimierung sind in der Optimierungsbibliothek implementiert. Datenaustausch zwischen den Modulen erfolgt über die `ModelPtr` und `HalfEdgeMesh`-Interfaces.

3.2 Software-Architektur

Das Las Vegas Surface Reconstruction Toolkit besteht im Wesentlichen aus drei Komponenten: IO-Bibliothek, Rekonstruktionsbibliothek und Meshoptimierungsbibliothek (s. Abbildung 3.1). Der Austausch von Daten zwischen den einzelnen Komponenten erfolgt über fest definierte Interfaces. Die IO-Bibliothek ist für das Laden und Speichern von Punktwolken- und Meshdaten zuständig. In der Rekonstruktionsbibliothek sind die zur Oberflächenrekonstruktion benötigten Algorithmen implementiert. Die Optimierungsbibliothek kapselt die Funktionalitäten zur Meshoptimierung. Schnittstelle zur IO-Bibliothek ist das `ModelPtr`-Interface. Ein `ModelPtr` kann sowohl Mesh- als auch Punktwolkendaten enthalten. Das Speichern und Laden der `ModelPtr` in verschiedenen Datenformaten erfolgt über entsprechend spezialisierte Konverterklassen. Aus den in einem `ModelPtr` enthaltenen Punktwolkendaten werden im Rekonstruktionsmodul Dreiecksmeshes in einer Halbkantendarstellung generiert. Diese `HalfEdgeMesh`-Klasse implementiert verschiedene Meshoptimierungsalgorithmen, die auf das erzeugte Mesh angewendet werden können. In der Verarbeitungspipeline ist es jederzeit möglich, die Zwischenergebnisse durch Konvertieren der Daten in eine `ModelPtr`-Struktur mit Hilfe der IO-Bibliothek zwischenzuspeichern.

Die Modellierung der Interfaces orientiert sich dabei vornehmlich an der Funktionalität der Module. Dabei wurde darauf geachtet, die benötigten Datenstrukturen und Operationen so weit wie möglich zu abstrahieren. Deshalb werden, wann immer es sinnvoll ist, C++-Templates verwendet, um von speziellen Datentypen unabhängig zu sein. So können die Vertices in den Mesh-Klassen durch verschiedene Vektorrepräsentationen ersetzt werden. Vorausgesetzt wird lediglich die Implementierung des `[]`-Operators zum Koordinatenzugriff. Diese Art von Abstraktion hat den Vorteil, dass sich vorhandene Software leicht durch Wrapper integrieren lässt. Zum Beispiel müssen die zur Rekonstruktion verwendeten Daten nicht zwangsläufig aus Dateien gelesen werden. Denkbar ist auch, die Daten online aus den Sensoren auszulesen, sei es durch eigene Treiber oder Middlewares wie ROS oder Player. Dazu ist es lediglich notwendig, die empfangenen Nachrichten in eine `ModelPtr`-Struktur umzusetzen. In der anderen Richtung lassen sich

aus den ModelPtr-Daten leicht Nachrichten für diese Frameworks generieren. Der Aufbau der einzelnen Module zusammen mit den relevanten Schnittstellendefintionen wird in den folgenden Abschnitten vorgestellt.

3.2.1 IO-Bibliothek

In der IO-Bibliothek sind Klassen zum Laden und Speichern von Meshes und Punktwolken implementiert. Jede Klasse, die einen Import oder Export zur Verfügung stellt, muss das BaseIO-Interface implementieren. Dieses besteht aus den beiden Methoden read und save (siehe Abbildung 3.2). Alle Daten auf IO-Ebene werden über ModelPtr-Objekte ausgetauscht. Dabei handelt es sich um Instanzen der Klasse Model, die in verwalteten Pointern abgelegt werden. Die Verwendung von verwalteten Pointern hat den Vorteil, dass die geladenen Daten automatisch wieder freigegeben werden, sobald sie nicht mehr benötigt werden. Die Ressourcenverwaltung wird dabei in der kapselnden Pointer-Klasse realisiert.

Die Model-Klasse wird verwendet, um Meshdatenstrukturen mit Punktwolkendaten zu assoziieren. Auf diese Art und Weise lassen sich in Datenformaten, die dies unterstützen, Oberflächenmodelle zusammen mit den Punktwolkendaten ablegen, aus denen sie rekonstruiert wurden. Dies ist z.B. im PLY-Format möglich. Die Mesh-Daten werden in MeshBuffer-Objekten abgelegt, analog dazu gibt es die Klasse PointBuffer für Punktwolkendaten. Alle benötigten Informationen werden in eindimensionalen Bufferstrukturen abgelegt. Diese Struktur wird beim Speichern von vielen Austauschdatenformaten direkt unterstützt und kann z.B. auch zum Rendern mit OpenGL (z.B. für Vertex- und Index-Pointer) direkt verwendet werden [39].

Die Bufferstrukturen werden in der Regel nur temporär benötigt: Entweder werden sie zum Speichern aus einer anderen Datenstruktur (z.B. einem verzeigerten Mesh) erzeugt oder werden nach dem Laden in eine verzeigerte Struktur umgewandelt, z.B. wenn Punktdaten in einen kd-Baum eingefügt werden. Es ist daher wichtig, diese temporären Strukturen freizugeben, wenn sie nicht mehr benötigt werden. Dies wird durch den Einsatz von Smart-Pointern gewährleistet. In der Software wird die shared_pointer-Implementierung der Boost-Bibliothek verwendet [156]. Diese ist zum C++11-Standard kompatibel, kann aber auch von diversen Kompilern verwendet werden, die diesen Standard noch nicht vollständig umsetzen.

Die Buffer stellen die unterste Schnittstelle zwischen Rekonstruktionsverfahren und den Ein- und Ausgabegeräten dar. Daher müssen alle Komponenten, die Punkt- und Meshdaten verwenden, eine entsprechende Schnittstelle für Buffer-Objekte zur Verfügung stellen. Problematisch ist, dass einige Bibliotheken, z.B. STANN [113], zur Nächsten-Nachbar-Suche zweidimensionalen Zugriff mittels [][]-Operator auf die Punktdaten benötigen. Dabei indiziert der erste Zugriff den Punkt im Buffer, der zweite Zugriff liefert die Raumkoordinate. Eine einfache Lösung des Problems besteht darin, die Daten in die benötigte Struktur umzukopieren. Das Umkopieren der Daten von der eindimensionalen in die zweidimensionale Struktur verdoppelt aber den benötigten Arbeitsspeicher. In Bibliotheken, die Templates unterstützen, kann dies umgangen werden, indem ein entsprechender Zugriffsoperator für den Koordinatenzugriff implementiert wird. Dazu gibt es in den Buffer-Klassen Zugriffsmethoden, die die eindimensionalen Arrays in ein Array einer Struktur des Types Coord umcasten, die über einen entsprechenden Zugriffsoperator verfügt,

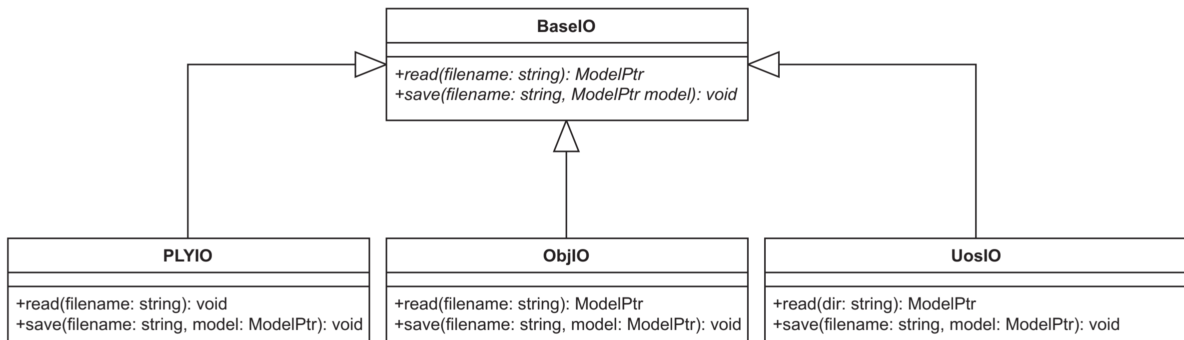


Abbildung 3.2: UML-Klassendiagramm der IO-Bliobliothek. Alle Klassen, die das `BaseIO`-Interface implementieren, können genutzt werden, um Daten zu speichern und zu lesen. Die zu speichernden Objekte (Punktwolken und Meshes) werden über `ModelPtr`-Instanzen definiert.

so dass die Buffer in Template-Funktionen direkt über `[][]`-Zugriffe ohne Datenverdopplung verwendet werden können:

```
template<typename CoordT>
struct coord
{
    CoordT x;
    CoordT y;
    CoordT z;
    CoordT& operator[]( const size_t i ) {
        switch ( i ) {
            case 0: return x; break;
            case 1: return y; break;
            case 2: return z; break;
            default: return z;
        }
    }
};
```

Diese Template-Struktur bietet Platz für drei Variablen `x`, `y` und `z`. Generell muss man bei der Verwendung von C-Strukturen auf korrektes "Alignment" der Daten achten, d.h. der Compiler fügt unter Umständen so genannte Padding-Bytes ein, wenn dies für die Hardware-Architektur von Vorteil ist [200]. Instanziert man diese Struktur mit Standarddatentypen, ist ein dichtes Alignment immer sicher gestellt, da alle Member vom selben Typ sind. Für einfache Datentypen sind die Strukturen also immer dicht gepackt, so dass die folgenden beiden Deklarationen das selbe Muster im Speicher erzeugen und daher gefahrlos durch ein Casting konvertiert werden können:

```
float buffer1[256 * 3];
coord<float> buffer2[256];
```

Tabelle 3.1: Übersicht der Elemente, die in den Buffer-Klassen abgelegt werden können. Für jedes Element stehen Zugriffsmethoden für den eindimensionalen und zweidimensionalen Koordinatenzugriff zur Verfügung. Die letzte Spalte gibt an, wie viele Koordinaten ein Element des entsprechenden Typs besitzt.

Datenstruktur	Element	Name	Datentyp	# Koordinaten
MeshBuffer	Vertices	vertexArray	float	3
	Dreiecksdefintionen	indexArray	unsigned int	3
	Vertexnormalen	vertexNormalArray	float	3
	Vertexfarben	vertexColorArray	float	3
	Flächenfarben	faceColors	unsigned char	3
	Texturindizes	textureIndices	unsigned int	1
	Texturkoordinaten	textureCoords	float	2
PointBuffer	Punkte	pointArray	float	3
	Normalen	pointNormalArray	float	3
	Farben	pointColorArray	unsigned char	3
	Remissionswerte	pointIntensities	unsigned char	1

Dementsprechend wurden für alle Buffer Zugriffsmethoden für eindimensionale und zweidimensionale Array-Typen implementiert, damit die geladenen Daten in möglichst vielen Anwendungsfällen ohne Speicheroverhead verwendet werden können. Um Speicherlecks zu verhindern, werden auch die konvertierten Arrays über Smartpointer-Instanzen zurück gegeben. Folgendes Beispiel zeigt die Umwandlung eines eindimensionalen Vertexbuffers in eine verwaltete Struktur, die zweidimensionalen Koordinatenzugriff unterstützt. Die Template-Instanzierungen für Standarddatentypen wurden zur besseren Lesbarkeit durch `typedef`-Aliase ersetzt:

```
typedef boost::shared_ptr<coord<float> > coord3fArr;

coord3fArr PointBuffer::getIndexedVertexArray( size_t &n )
{
    n = m_numVertices;
    coord3fArr p = *((coord3fArr*) &m_points);
    return p;

}
```

Neben Geometrieinformationen können in den Buffer-Klassen noch weitere Attribute wie Farben für Punkte und Flächen, Normalen und Texturinformationen abgelegt werden. Tabelle 3.1 gibt eine Übersicht über die unterstützten Attribute und die verwendeten Datentypen. Die Wahl der Datentypen entspricht dabei gängigen Konventionen.

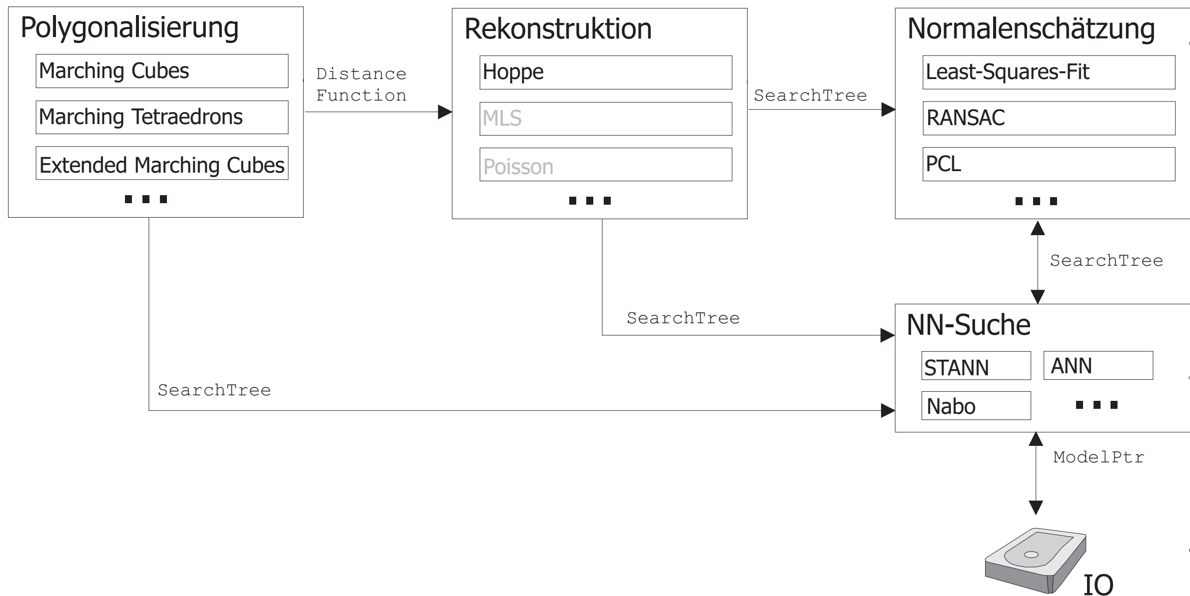


Abbildung 3.3: Aufbau der Rekonstruktionsbibliothek. Nächste-Nachbar-Suchanfragen werden über das `SearchTree`-Interface, z.B. zur Normalenschätzung ausgetauscht. Die geschätzten Normalen werden von den Rekonstruktionsklassen verwendet, die das `DistanceFunction`-Interface zur Verfügung stellen, das von den verschiedenen Polygonalisierungsverfahren zur Isoflächenapproximation verwendet wird.

3.2.2 Rekonstruktionsbibliothek

Der allgemeine Aufbau der Rekonstruktionsbibliothek ist in Abbildung 3.3 dargestellt. Die Verarbeitungspipeline besteht in der Regel aus den Schritten “Normalenschätzung”, “Flächenrekonstruktion” und “Polygonalisierung”. Die Ausgangsdaten werden von der IO-Bibliothek mittels `ModelPtr` zur Verfügung gestellt. Aus diesen Daten wird dann eine Suchbaumstruktur zur Nächsten-Nachbar-Suche erstellt. Dazu können verschiedene Bibliotheken zum Einsatz kommen. Es muss lediglich ein Wrapper für das `SearchTree`-Interface implementiert werden. Über diese Schnittstelle können Nächste-Nachbar-Suchanfragen von den einzelnen Komponenten an die verwendete Suchbibliothek gestellt werden. `SearchTree` stellt verschiedene Suchanfragen, z.B. k nächste Nachbarn oder Radiussuche, zur Verfügung.

Enthalten die Eingabedaten keine Normaleninformationen, werden diese aus den Eingabedaten erzeugt. Dazu können verschiedene Verfahren zum Einsatz kommen. Aus den Punkt- und Normalendaten wird die Isofläche bestimmt. Eine Isofläche wird von Klassen repräsentiert, die das `DistanceFunction`-Interface zur Verfügung stellen. Über dieses Interface kann zu einem gegebenen Raumpunkt der gerichtete Abstand zur approximierten Oberfläche bestimmt werden. Diese Information wird letztlich zur Erzeugung eines Polygonnetzes verwendet. Als Isoflächenrepräsentation wird derzeit lediglich Hoppes Distanzfunktion verwendet. Eine Integration anderer Verfahren ist in dieser Architektur aber prinzipiell auch möglich. Es muss nur sichergestellt werden, dass das `DistanceFunction`-Interface korrekt implementiert wird. Der letzte Schritt besteht in der Polygonalisierung der berechneten Isofläche. Die Ausgabe des Meshes erfolgt über

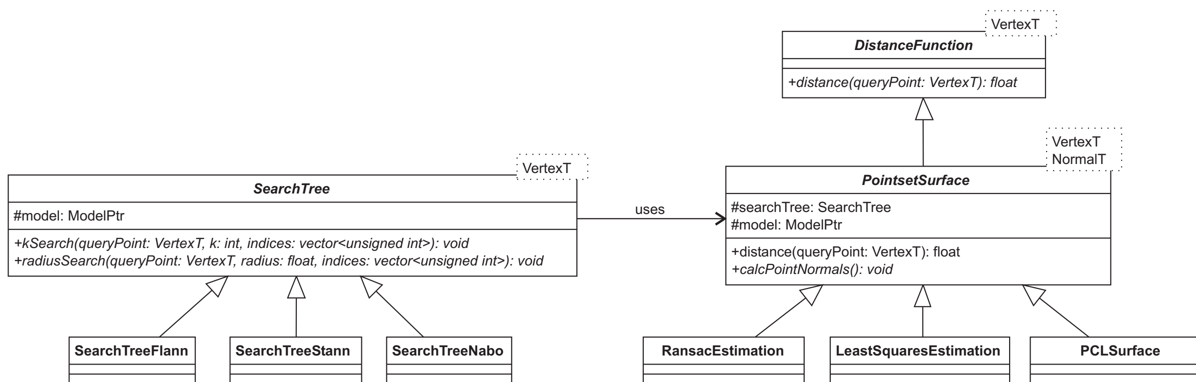


Abbildung 3.4: UML-Diagramm zur Normalenschätzung und Distanzfunktion. Die Klasse `PointsetSurface` implementiert das `DistanceFunction`-Interface. Die verschiedenen Verfahren zur Normalenschätzung werden in Unterklassen von `PointsetSurface` gekapselt. Diese nutzen eine Implementierung der `SearchTree`-Schnittstelle zur Nächste-Nachbar-Suche.

das `BaseMesh`-Interface. In dieser Schnittstelle sind die erforderlichen Operationen zum Aufbau eines Dreiecksnetzes definiert.

Normalenschätzung und Distanzfunktion

Die Klassenhierarchie und Interaktion der Komponenten zur Normalenschätzung und Isoflächenbestimmung sind in Abbildung 3.4 dargestellt. Die Isoflächenrepräsentation nach Hoppe wird in der Klasse `PointsetSurface` realisiert. Diese implementiert das `DistanceFunction`-Interface und nutzt zur Berechnung des Distanzwertes Flächennormalen der Punktwolke. Verschiedene Verfahren zur Normalenschätzung sind in Spezialisierungen von `PointsetSurface` realisiert. Die dazu benötigten Punktdaten sind im `ModelPtr` der gemeinsamen Oberklasse hinterlegt. Durch Aufruf einer Implementierung der Methode `calcNormals()` einer Unterklasse werden die Normalen berechnet und im entsprechenden Buffer des `ModelPtr`s gespeichert. Zur Berechnung der Normalen wird die Suchbaumstruktur der `PointsetSurface`-Klasse genutzt.

Nächste-Nachbar-Suchalgorithmen sind in Unterklassen von `SearchTree` integriert. Diese müssen die abstrakten Methoden `kSearch` und `radiusSearch` mit den folgenden Signaturen implementieren:

```
void kSearch(VertexT queryPoint, int k, vector<unsigned int> neighbors);
void radiusSearch(VertexT queryPoint, float radius, vector<unsigned int> nb);
```

Die Variable `queryPoint` ist jeweils der Punkt, zu dem die nächsten Nachbarn gesucht werden sollen. `VertexT` ist ein Template-Parameter der Klasse und bestimmt den Typ der verwendeten Klasse zur Speicherung von Punkten. Dieser Typ muss Koordinatenzugriff mittels `[]`-Operator unterstützen. Die Parameter `k` und `radius` geben die Anzahl der zu findenden Nachbarn bzw. den Suchradius an. Das Ergebnis der Suchanfragen besteht aus den Indizes der gefundenen Punkte im abgelegten Punktbuffer. Diese werden im übergebenen Vektor gespeichert. Die Rückgabe

der Indizes hat den Vorteil, dass darüber auch auf die anderen im Buffer hinterlegten Attribute der einzelnen Punkte, z.B. Flächennormalen, zugegriffen werden kann. Die einmal erstellte Suchbaumstruktur kann also mehrfach verwendet werden, nämlich zur Berechnung der Normalen aus der k-Nachbarschaft eines Punktes, die dann im Normalenbuffer des Modells gespeichert werden, und zur Bestimmung der Distanzfunktion, indem bei der Berechnung über die Indizes auf die zuvor berechneten Normalen zurückgegriffen wird.

Bei der Entwicklung dieser Klassenstruktur wurde darauf geachtet, möglichst viel Funktionalität in den Oberklassen zu implementieren. So ist die Berechnung der Distanzfunktion samt Normalenschätung in der Klasse `PointsetSurface` zusammengefasst. Dies hat den Vorteil, dass sich andere weitere Verfahren zur Normalenschätzung einfach in Unterklassen integrieren lassen, indem diese nur die Methode `calcNormals()` implementieren. Ähnliches gilt für die Klasse `SearchTree`. Sie bietet neben den beiden oben beschriebenen Funktionen eine ganze Reihe weitere Funktionen mit anderen Signaturen, mit denen Nächste-Nachbar-Suchanfragen gestellt werden können. Diese wurden aus Gründen der Übersichtlichkeit nicht in das UML-Diagramm in Abbildung 3.4 aufgenommen. Durch das Bereitstellen verschiedener Signaturen ist das Suchinterface sehr flexibel und gleichzeitig leicht implementierbar. Für die Integration externer Bibliotheken müssen lediglich `kSearch` und `radiusSearch` in den Unterklassen programmiert werden, die restlichen Funktionalitäten stellt `SearchTree` zur Verfügung.

Polygonalisierung

Abbildung 3.5 zeigt die Struktur des Moduls zur Erzeugung von Polygonnetzen aus einer Isofläche. Alle in der Software verwendeten Rekonstruktionsverfahren implementieren das `SurfaceReconstruction`-Interface. Durch Aufruf der `getMesh`-Methode wird ein `BaseMesh` aus den vorhandenen Daten erzeugt. Die `BaseMesh`-Klasse stellt dabei eine Abstraktion eines allgemeinen Dreiecksnetzes dar, ohne die zur Flächenrepräsentation benutzte Datenstruktur festzulegen. Auf diese Art und Weise lassen sich verschiedene verzeigerte und unverzeigerte Datenstrukturen für unterschiedliche Zwecke erzeugen, z.B. unverzeigerte Strukturen, die direkt gespeichert werden können, oder komplexe Strukturen, die für den jeweiligen Verwendungszweck entwickelt wurden. In der Software sind momentan die Klassen `StaticMesh` für unverzeigerte Meshes, `HalfEdgeMesh` als verzeigerte Datenstruktur zur optimierten Abfrage von Nachbarschaftsbeziehungen und `CollapseMesh` zur Meshoptimierung nach Melax und Garland / Heckbert basierend auf Jeff Somers Quellcode [179] integriert. Die `BaseMesh`-Klassen müssen Funktionen zum Hinzufügen von Vertices und Normalen (`addVertex(VertexT v)`) bzw. `addNormal(NormalT n)` und Dreiecken (`addTriangle(int a, int b, int c)`) unterstützen. Dreiecke werden dabei durch Referenzen auf bereits erzeugte Vertices definiert. Die jeweiligen Implementierungen müssen sicher stellen, dass die neu erzeugten Objekte korrekt in die jeweils verwendete interne Meshrepräsentation integriert werden. Die Methode `finalize()` ist in den Spezialisierungen dafür zuständig, die intern verwendeten Datenstrukturen in eine `MeshBuffer`-Repräsentation zu überführen, so dass die Meshes danach über die IO-Schnittstelle exportiert werden können.

Die Marching-Cubes-Rekonstruktion ist in der Klasse `VoxelGridReconstruction` realisiert. Diese Klasse baut ein Gitter aus `VoxelGridBox`-Instanzen auf, die jeweils einen Voxel für die loka-

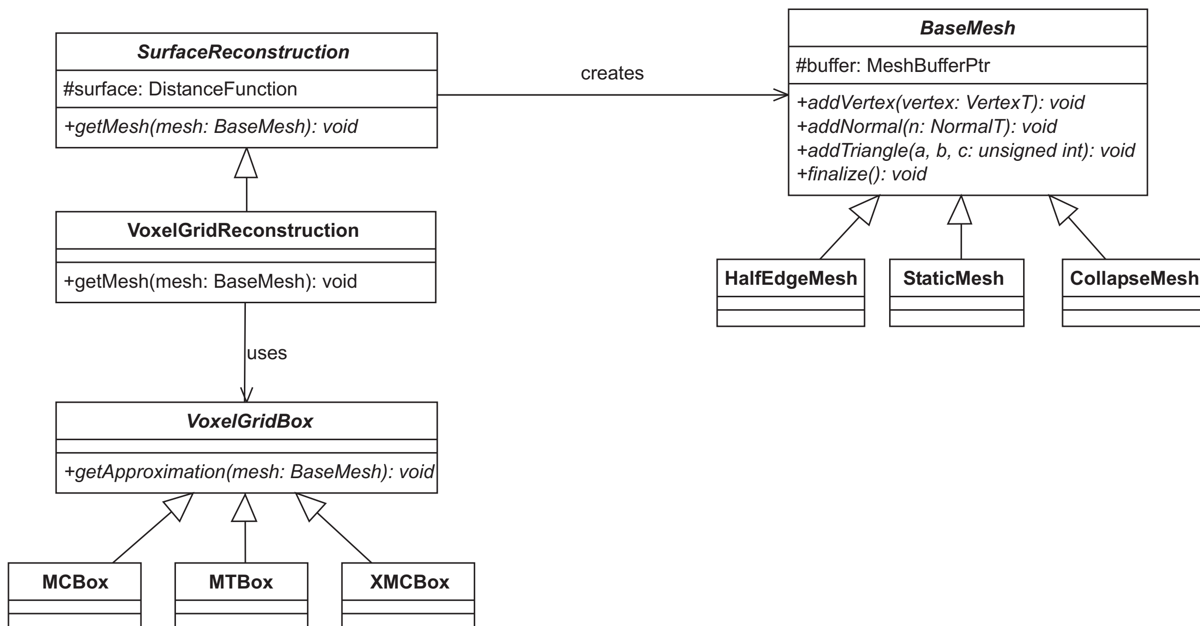


Abbildung 3.5: UML-Klassendiagramm zur Polygonalisierung. Die verschiedenen Marching-Cubes-Algorithmen sind als Unterklassen von `VoxelgridBox` realisiert. Diese werden von einem `VoxelGridReconstruction`-Objekt generiert, das das Raumgitter zur Rekonstruktion aufbaut. Zum Aufbau der Meshes wird eine `BaseMesh`-Instanz verwendet. Die verwendeten Datenstrukturen sind in den entsprechenden Unterklassen implementiert.

le Approximation repräsentieren. Die zur lokalen Approximation verwendete Marching-Cubes-Variante wird in Unterklassen von `VoxelGridBox` implementiert. Die Dreieckserzeugung wird durch die Implementierung der `getMesh`-Methode realisiert, die als Parameter eine Referenz auf die verwendete Mesh-Datenstruktur übergeben bekommt. Bei der Rekonstruktion werden die im `BaseMesh` definierten Methoden zum Integrieren der lokalen Flächenmuster in das Gesamtmesh aufgerufen (`addVertex`, `addNormal` und `addTriangle`).

3.2.3 Meshoptimierung

Die Meshoptimierung findet direkt auf den erzeugten Datenstrukturen statt, d.h. die einzelnen Funktionen werden in Methoden der entsprechenden Meshklassen implementiert, die von `BaseMesh` abgeleitet werden. Die direkte Integration in die einzelnen Klassen hat den Vorteil, dass die implementierten Methoden direkten Zugriff auf die interne Meshstruktur haben. Zudem ist die Definition eines gemeinsamen Interfaces für Meshoptimierungsalgorithmen schwer möglich, da unterschiedliche Datenrepräsentationen und Funktionalitäten vorausgesetzt werden. Zwar werden von vielen Meshoptimierungsverfahren elementare Operationen wie Löschen von Dreiecken, Entfernen von Kanten durch Zusammenziehen auf einen einzelnen Vertex (“EdgeCollapse”) und Kantentausch (“EdgeFlip”) verwendet, es ist aber schwierig, dafür ein gemeinsames Interface zu definieren, da die Identifizierung der betroffenen Elemente von ihrer Repräsentation

abhängig ist. So können Kanten in einer Kantenliste z.B. über einen Integer-Index angesprochen werden. In einer Halbkanten-Datenstruktur gibt es eine derartige Liste nicht. Hier werden die Kanten direkt durch Zeiger auf die entsprechenden Objekte adressiert. Ein ähnliches Problem ergibt sich bei der Definition einer Signatur für einen generischen Meshoptimierungsaufruf, da die Kritierien zur Optimierung von diversen Parametern, z.B. der verwendeten Fehlermetrik, abhängen. Ist eine Repräsentation für eine Klasse von Optimierungsverfahren gut geeignet, können die Algorithmen entweder als neue Methoden implementiert werden, oder die entsprechenden Funktionalitäten werden in Unterklassen überschrieben.

In der vorliegenden Software sind die neu entwickelten Methoden zur Meshoptimierung, die die Planarität der Umgebungsstruktur ausnutzen, in der Klasse `HalfEdgeMesh` implementiert. Meshoptimierung durch Edge-Collapsing sind in der Klasse `CollapseMesh` realisiert.

3.2.4 Zusammenfassung

Die Struktur der Software wurde so entwickelt, dass sich die zur Lösung verschiedener Teilprobleme verwendeten Algorithmen und Datenstrukturen leicht austauschen lassen. Dies ist insbesondere für die Evaluation der verschiedenen Verfahren von Vorteil. Die integrierten Verfahren werden in den folgenden Abschnitten beschrieben.

3.3 Normalenschätzung

Der erste Schritt zur Rekonstruktion eines Dreiecksnetzes ist die Approximation von Normalen für die geladenen Datenpunkte. Diese Informationen werden bei der Auswertung der Distanzfunktion benötigt, um den gerichteten Abstand der Anfragepunkte von der approximierten Oberfläche zu bestimmen.

Dabei ist es von zentraler Bedeutung, dass die geschätzten Normalen präzise die Krümmung der zugrunde liegenden Oberfläche widerspiegeln und zudem konsistent orientiert sind. Fehlerhaft orientierte Normalen verfälschen die berechneten Abstandswerte und sorgen so dafür, dass die Schnittpunkte der Voxel mit der definierten Isofläche bei der Marching-Cubes-Rekonstruktion nicht korrekt interpoliert werden. Bei Hoppes Distanzfunktion ist der Fehler, der bei der Berechnung des Abstandes gemacht wird, dabei proportional vom Kosinus zwischen Anfragepunkt $\boldsymbol{p}$ und der nächsten Tangentialebene T_i mit Schwerpunkt $\boldsymbol{o_i}$ und Normalen $\boldsymbol{n_i}$:

$$d_T = (\boldsymbol{p} - \boldsymbol{o}_i) \cdot \boldsymbol{n}_i$$

Somit ist es wichtig, die Orientierungsfehler zu begrenzen, da je nach Lage der Fläche relativ zum Abfragepunkt schon kleine Fehler in der Orientierung der Normalen den berechneten Abstandsfehler stark verfälschen können. Fehler bei der Normalenschätzung fallen in den Rekonstruktionen insbesondere bei ebenen Flächen auf. Dort sorgen Fehlschätzungen dafür, dass diese Flächen nicht glatt sind, sondern wellenförmige Strukturen erzeugen, die auf die Fluktuation der geschätzten Normalen zurückzuführen sind. Eine konsistente Orientierung wird benötigt,

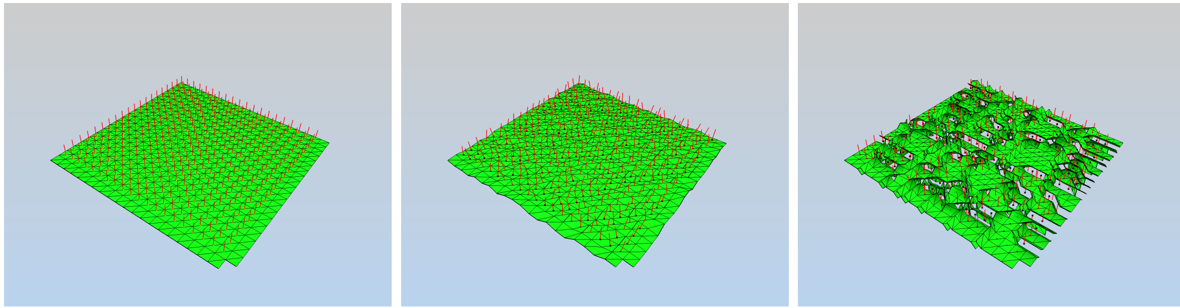


Abbildung 3.6: Einfluss der Normalenqualität auf das Rekonstruktionsergebnis am Beispiel einer ebenen Fläche. Im linken Bild wurden die korrekten Normalen zur Rekonstruktion verwendet. Im mittleren Bild fluktuieren die Normalen um 5°. Im rechten wurden die Normalen zusätzlich zufällig umorientiert.

um geschlossene Flächenmuster zu rekonstruieren. Wechseln die Normalen inkonsistent ihre Orientierung, entstehen Löcher in geschlossenen Flächen. Diese Effekte werden in Abbildung 3.6 verdeutlicht.

Im linken Bild wurde die ebene Fläche mit korrekt ausgerichteten Normalen rekonstruiert. Die berechneten Dreiecke liegen alle in einer gemeinsamen Ebene. Im mittleren Bild weicht die Ausrichtung der Normalen um 5° von der korrekten Achse ab. Durch diese Abweichungen werden die Vertices der Dreiecke in der Rekonstruktion aus der Ebene herausgezogen, die Fläche ist nicht mehr glatt. Im rechten Bild wurden die Normalen zusätzlich zufällig umorientiert. Dies führt wie erwartet zu Löchern im Mesh.

Für die Genauigkeit der Rekonstruktion ist die Qualität der berechneten Punktnormalen also von großer Bedeutung. Bei der Schätzung von Normalen bereiten das Messrauschen und die schwankende Punktdichte bei einigen Sensorarten Probleme. Least-Squares-Verfahren sind gegen normalverteiltes Rauschen relativ unempfindlich, da es sich bei der Approximation herausmittelt. Problematisch sind randomisierte Ausreißer, sie können das Ergebnis unter Umständen stark verfälschen. RANSAC-basierte Ansätze sind robuster gegenüber stochastischem Rauschen, finden aber nicht garantiert die optimale Approximation. Eine weitere Problematik ist die Bestimmung der optimalen Anzahl von k Nachbarpunkten, die zur Approximation der Normalen benötigt wird. Eine hohe Anzahl an Nachbarpunkten verringert den Einfluss des systematischen Rauschens, mittelt aber auch scharfe Kanten stärker heraus und erhöht die Laufzeit. Ziel ist es dementsprechend, ein möglichst kleines k zu bestimmen, so dass das Verhältnis zwischen Laufzeit und Normalenqualität optimiert wird. In Datensätzen mit einer annähernd gleichverteilten Punktdichte kann ein festes k gewählt werden. In Messdaten mit wechselnder Punktdichte muss dieser Wert je nach lokaler Dichte adaptiert werden.

Um die Qualität der geschätzten Normalen weiter zu verbessern, kann in einem zweiten Schritt eine Mittelwertbildung über eine bestimmte Anzahl benachbarter Normalen erfolgen. Dabei wird jede Normale $\boldsymbol{n}$ durch den Mittelwert $\boldsymbol{n}_i$ dieser Nachbarn $\boldsymbol{n}_k$ ersetzt:

$$\boldsymbol{n}_i = \frac{1}{k} \sum_{j=1}^{k} \boldsymbol{n}_j$$

Die Anzahl der zur Mittelwertbildung verwendeten Normalen muss nicht identisch mit der im ersten Schritt verwendeten Anzahl sein. Im Folgenden wird daher zwischen den Parametern k_n für die Größe der initialen k-Nachbarschaft und k_i für die Anzahl der zur Mittelwertbildung verwendeten Normalen unterschieden.

Zum Vergleich wurden in der Software ein Least-Squares-basierter und ein RANSAC-basierter Algorithmus zur Schätzung von Normalen integriert. Zusätzlich wurde ein Verfahren entwickelt, das die Größe der k_n-Nachbarschaft in dünnen Datensätzen, speziell von nickenden Laserscannern, anpasst. Die Orientierung der Normalen kann entweder zum Ursprung oder zum Schwerpunkt der Szene geschehen.

3.3.1 Least-Squares-basierte Normalenschätzung

Das implementierte Least-Squares-Verfahren zur Schätzung von Normalen basiert auf einer Hauptkomponentenanalyse. Dieses Verfahren hat den Vorteil, dass die Normalen in einem lokalen Bezugsystem berechnet werden und umgeht die in Kapitel 2.6 genannten numerischen Probleme, die beim Fitten einer allgemeinen Ebenengleichung auftreten. Dabei werden die Punkte der k_n-Nachbarschaft in einer neuen dreidimensionalen Orthogonalbasis dargestellt, so dass der quadratische Abstand der Punkte zur durch diese Basis definierten Ebene minimiert wird. Die Basis wird durch die Eigenvektoren $\boldsymbol{e_1}, \boldsymbol{e_2}, \boldsymbol{e_3}$ zu den Eigenwerten $\lambda_1 \leq \lambda_2 \leq \lambda_3$ der Kovarianzmatrix C der Punkte im Schwerpunkssystem mit Schwerpunkt $\boldsymbol{c}_k$ definiert [16]:

$$C = \frac{1}{k} \sum_{i=1}^{k} (\boldsymbol{x}_i - \boldsymbol{c}_k) \times (\boldsymbol{x}_i - \boldsymbol{c}_k)^T$$

Die Eigenvektoren $\boldsymbol{e_1}$ und $\boldsymbol{e_2}$ zu den Eigenwerten λ_1 und λ_2 spannen die gefittete Ebene auf, der Eigenvektor $\boldsymbol{e_2}$ zu λ_3 bildet die Normale [16]. Wie bei anderen Verfahren auch, ist die Orientierung der Normalen nicht eindeutig bestimmt, d.h. auch bei diesem Verfahren müssen die Normalen nach der Berechnung konsistent orientiert werden.

Solche Eigenwertprobleme lassen sich numerisch mit gängigen Bibliotheken zur linearen Algebra lösen. Da die Normalenschätzung parallelisiert erfolgen soll, ist es wichtig, dass die verwendete Bibliothek Thread-sicher ist. Für die vorliegende Implementierung wird die C++-Bibliothek "Eigen" verwendet. Diese Open-Source-Bibliothek ist Thread-sicher und mit den in der Software verwendeten Vertex-Klassen kompatibel. Zudem unterstützt sie Optimierungsfunktionen aktueller Prozessoren und bietet entsprechend hohe Performanz [62].

3.3.2 RANSAC-basierte Normalenschätzung

Die zweite Variante zur Normalenschäzung benutzt einen modifizierten RANSAC-Algorithmus ("Random Sample Consensus") [54]. Der RANSAC-Algorithmus wird in vielen Varianten benutzt [163, 170], um parametrisierte Modelle in Daten mit Ausreißern zu fitten. Die Idee des RANSAC-Verfahrens ist, ein parametrierbares Modell mit n Parametern gegen eine Menge an Sampledaten zu testen. Die Originalvariante arbeitet folgendermaßen: Aus der Menge S der

Samplepunkte werden n Punkte zufällig ausgesucht, aus denen die Modelparameter mittels Regressionsrechnung bestimmt werden. Anschließend werden alle Punkte in S gegen das aktuelle Modell getestet. Die Punkte, deren Abstand zum Modell kleiner als ein vordefinierter Fehlerwert sind, werden in der so genannten "Konsensmenge" C gespeichert. Diese Schritte werden mehrmals wiederholt. Nach einer festgelegten Anzahl von Iterationen werden im letzten Schritt die Modelparameter aus C bestimmt. Durch das Eleminieren der Punkte aus C, die nicht gut zu einem gewählten Modell passen, werden potentielle Ausreißer in den Daten gefiltert. Da zur Bestimmung des Modells nur eine minimale Anzahl an Punkten verwendet wird, arbeitet das Verfahren sehr schnell, wenn die Zeit zur Bestimmung der Parameter wesentlich größer ist, als die Zeit, die für den Test der Samplemenge gegen das Modell benötigt wird.

Zur Normalenschätzung kommt ein modifizierter RANSAC-Algorithmus zum Einsatz. Dabei werden zufällig drei disjunkte Punkte $\boldsymbol{a}$, $\boldsymbol{b}$ und $\boldsymbol{c}$ aus der k_n-Nachbarschaft ausgewählt. Aus diesen drei Punkten werden zwei Vektoren $\boldsymbol{d}_1$ und $\boldsymbol{d}_2$ mit dem gemeinsamen Verbindungspunkt $\boldsymbol{b}$ gebildet:

$$\boldsymbol{d}_1 = \boldsymbol{b} - \boldsymbol{a}$$
$$\boldsymbol{d}_2 = \boldsymbol{b} - \boldsymbol{c}$$

Dabei wird angenommen, dass sich die Vektoren in einer gemeinsamen Ebene befinden. Die Normale $\boldsymbol{n}$ dieser Ebene ist dann das Kreuzprodukt der beiden Vektoren:

$$\boldsymbol{n} = \boldsymbol{d}_1 \times \boldsymbol{d}_2$$

Anschließend wird der quadratische Abstand ϵ aller Punkte der k_n-Nachbarschaft zur durch $\boldsymbol{n}$ und den Aufpunkt $\boldsymbol{b}$ definierten Ebene bestimmt. Anschließend werden die Schritte iteriert. Dabei wird jeweils die Normale, die den kleinsten gemeinsamen Abstand zu den Modellpunkten hat, gespeichert. Die Iteration wird fortgesetzt, bis eine vorher festgelegte Maximalzahl i_{max} an Iterationen erreicht wurde oder zwischen i_n Iterationen keine bessere Normale mehr gefunden wurde (siehe Algorithmus 3.1).

Durch dieses Verfahren wird bei kleinen k_n-Nachbarschaften relativ schnell eine gute Approximation erreicht, da es nur wenige mögliche Kombinationen zur Ziehung der drei Punkte gibt. Aus demselben Grund kann auch auf die Bildung einer Konsensmenge verzichtet werden, da viele der möglichen Hypothesen durchgetestet werden und die Wahrscheinlichkeit daher hoch ist, mit wenigen Iterationen das beste Modell zu bestimmen. Trotzdem bleibt die Robustheit gegenüber Ausreißern erhalten. Befinden sich nämlich signifikante Ausreißer in den Daten, wird ein Modell, das mit einem oder mehreren Ausreißern gebildet wurde, einen vergleichsweise großen quadratischen Fehler erzeugen, so dass die daraus berechnete Normale verworfen wird. Zu guter Letzt lässt sich der Algorithmus leicht parametrieren, da lediglich die maximale Anzahl an Iterationen und nicht verbessernden Iterationen vorgegeben werden muss. Dieser lässt sich an die Größe der k_n-Nachbarschaft koppeln. Im originalen RANSAC-Verfahren wäre der Schwellwert zu bestimmen, unterhalb dessen ein Punkt in die Konsensmenge aufgenommen wird. Dieser hängt allerdings von der Skalierung der Eingangsdaten ab.

Algorithmus 3.1 RANSAC-basierte Normalenberechnung.

```
bestPlane ← some initial value
bestDistance ← max float
nonImprovingIterations ← 0
iterations ← 0
niMax ← maximum number of non-improving iterations
iMax ← maximum number of iterations
while nonImprovingIterations < niMax and iterations < iMax do
    modelPoints ← getDisjuctRandomPoints(3)
    plane ← calcPlaneRepresentation(modelPoints)
    distance ← 0
    for i = 0 to numNeighborPoints do
        distance ← distance + distanceToPlane(plane, neighborPoints[i])
    end for
    if distance < bestDistance then
        bestDistance ← distance
        nonImprovingIterations ← 0
        bestPlane ← plane
    else
        nonImprovingIterations ← nonImprovingIterations + 1
    end if
    iterations ← iterations + 1
end while
return getNormal(bestPlane)
```

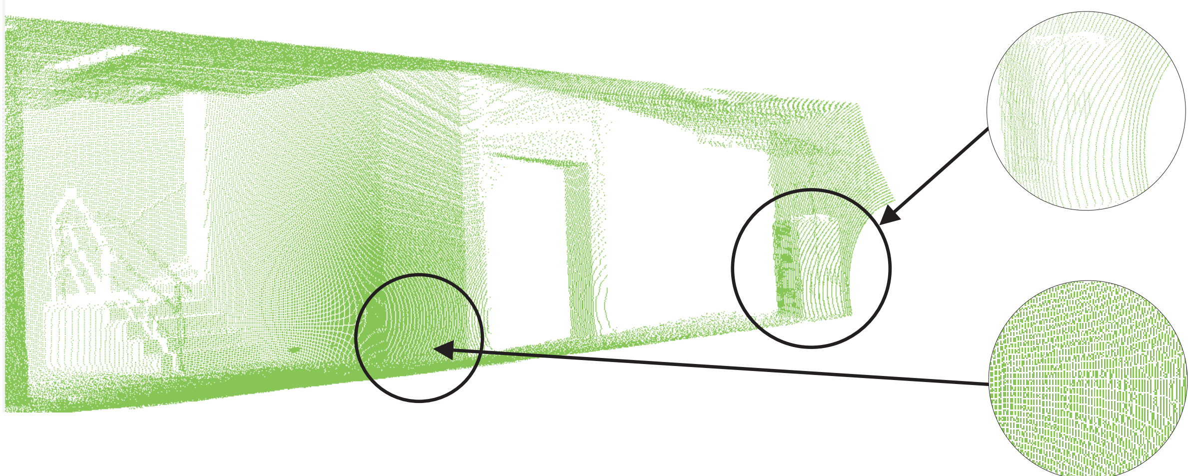


Abbildung 3.7: Beispiel einer ungleichmäßigen Punkteverteilung in einem Laserscan. Die Punktdichte ist in der Nähe des Scanners deutlich höher als an den Rändern. Dort bilden die Punkte Linienartefakte, die durch die Diskretisierung der Winkel bei der Ablenkung des Laserstrahls entstehen.

Abbildung 3.8: Beispiel für die Fluktuation von Normalen aufgrund zu kleiner k-Nachbarschaft mit $k_n = 5$. Man kann deutlich erkennen, dass mit abnehmender Punktdichte die Normalen um die einzelnen Scanlinien schwanken und teilweise ihre Richtung stark ändern (vor allem im rechten Ausschnitt).

3.3.3 Adaptive Anpassung der k-Nachbarschaft

Laserscanner und andere 3D-Sensoren erzeugen häufig Punktwolken mit schwankender Punktdichte. Bei rotierenden und nickenden Laserscannern ist die Punktdichte in der Nähe des Scannerzentrums dichter als an den Rändern, da die Winkelunterschiede zwischen zwei Messungen mit steigender Entfernung aufgrund des Strahlensatzes größere Punktabstände erzeugen. Abbildung 3.7 zeigt einen Laserscan, der mit dem Kurt3D-Laserscanner aufgenommen wurde. In den gezeigten Ausschnitten kann man diesen Effekt gut erkennen. Bei der Aufnahme des Scans befand sich der Roboter im linken Bereich der Szene. Dementsprechend ist die Punktdichte in diesem Bereich höher. Zum Rand des Messbereiches im rechten Teil des Bildes nimmt die Punktdichte so weit ab, dass sich gebogene "Linienartefakte" bilden. Diese entstehen durch die Diskretierung der Winkelstellungen des Laserscanners.

Diese Linienartefakte sind bei der Schätzung von Normalen problematisch. Für eine möglichst kurze Laufzeit des Verfahrens muss die k_n-Nachbarschaft zur Normalenapproximation möglichst klein gewählt werden. Dies kann zur Folge haben, dass die k_n-Nachbarschaft in dünnen Bereichen nicht mehr gleichmäßig in einer Ebene verteilt ist, wie dies in dichten Gebieten der Fall ist, sondern ein Liniensegment bildet. Bei solchen Punktanordnungen ist die Approximation von Ausgleichsebenen nicht mehr konsistent möglich, da das Modell schlecht zu den Daten passt. Das Ergebnis der Ausgleichsrechnung ist allein vom Rauschen der Punkte auf den Liniensegmenten bestimmt, also nicht mehr vorhersehbar. Die Normalen beginnen zur fluktuieren. Dieser Effekt ist in Abbildung 3.8 deutlich zu erkennen.

Zur Verminderung dieses Effektes muss sichergestellt sein, dass die zur Berechnung der Ausgleichsebene verwendeten Punkte von mindestens zwei Linien stammen. Durch diese Randbe-

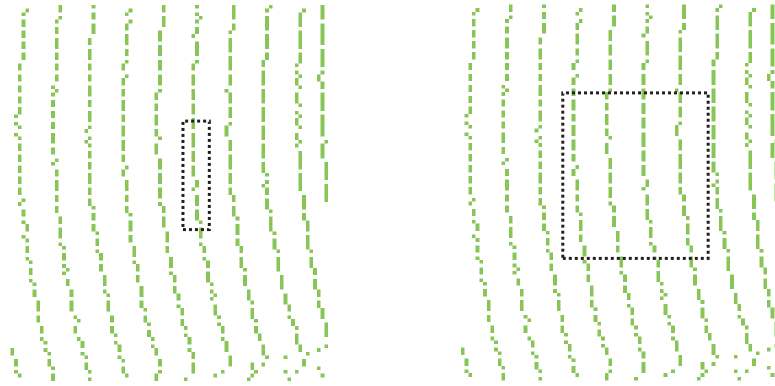


Abbildung 3.9: Bounding-Box-Kriterium zur Erkennung ungünstiger Punktkonfigurationen. Sind die Punkte auf einer Linie angeordnet, ist die Bounding-Box tendenziell länglich (links). Bei gleich verteilten Punkten ist die Ausdehnung gleichförmig (rechts)

Abbildung 3.10: Einfluss der adaptiven Normalenschätzung auf die der Qualität der Normalen und die Rekonstruktion. Das linke Bild zeigt das Ergebnis der Normalenschätzung mit Anpassung der k_n-Nachbarschaft. Das mittlere Bild zeigt eine Rekonstruktion basierend auf Normalen ohne Adaption, das rechte Bild zeigt die Rekonstruktion mit optimierten Normalen.

dingung ist sichergestellt, dass die lokale Ausgleichsebene in der Rotation um das Linienartefakt fixiert wird. Um dies zu erreichen, muss dementsprechend der k_n-Wert ausreichend groß gewählt werden, so dass die Nächste-Nachbar-Suche Punkte von zwei Linien liefert. Das kann sichergestellt werden, indem nach jeder Suchanfrage der minimale achsenparallele Quader berechnet wird, der die gefundenen Punkte einschließt ("Bounding-Box"). Diese Bounding-Box wird im lokalen Bezugssystem, das durch die geschätzte Normale und die assoziierte Ebene aufgespannt wird, berechnet, indem alle gefundenen Punkte in dieses System projiziert werden. Ein solches Vorgehen ist notwendig, da nicht sichergestellt ist, dass die Linien achsenparallel zum globalen System sind. Die Basisvektoren dieses lokalen Bezugssystems werden dabei entweder vom Least-Squares-Fit oder vom RANSAC-basierten Modell geliefert.

Um zu erkennen, ob die gefundenen Punkte auf einer Linie liegen, wird die Form der Bounding-Box untersucht. Liegt eine Linienkonfiguration vor, wird die Länge der Bounding-Box in einer Dimension deutlich größer sein als in den anderen beiden. Bei einer gleichmäßigen Verteilung

werden die zwei maximalen Ausdehnungen in etwa identisch sein (siehe Abbildung 3.9). Da die Punkte tendenziell in einer Ebene liegen, wird die kleinste Ausdehnung jeweils vernachlässigt. Seien d_1 und d_2 die betragsmäßig größten Dimensionen der Bounding-Box, dann wird angenommen, dass eine ungünstige Verteilung der Punkte vorliegt, wenn $d_1 > \epsilon \cdot d_2$ oder $d_2 > \epsilon \cdot d_1$ für einen Toleranzwert ϵ gilt.

Trifft eine dieser Bedingungen zu, wird die Schätzung mit einer vergrößerten k_n-Nachbarschaft wiederholt. Um die Anzahl der erforderlichen Suchanfragen zu reduzieren, ist es ratsam, die Anzahl der nächsten Nachbarn deutlich zu vergrößern. In der vorliegenden Implementierung wird die Anzahl bei jeder Iteration verdoppelt, prinzipiell kann man k_n aber mit einer beliebigen Heuristik ansteigen lassen. Die Verdoppelungsstrategie hat sich in der Praxis bewährt. Den qualitativen Einfluss auf die Qualität der Normalen und deren Einfluss auf die Rekonstruktion zeigt Abbildung 3.10.

3.4 Rekonstruktion mit Marching Cubes

In diesem Abschnitt wird die Implementierung der verschiedenen Marching-Cubes-Verfahren und der dazu benötigten Datenstrukturen beschrieben.

3.4.1 Gitterstruktur für die Marching-Cubes-Rekonstruktion

Für die Marching-Cubes-Rekonstruktion wird eine Gitterstruktur einer vorgegebenen Auflösung benötigt. Um Speicherplatz zu sparen, sollen nur Zellen erstellt werden, die auch Datenpunkte enthalten. In der Literatur werden dazu in der Regel modifizierte Octrees verwendet [137, 206], in denen die Blätter die Rekonstruktionszellen bilden. Octrees können parallelisiert aufgebaut werden. Dies ist allerdings mit einem erheblichen Aufwand bei der Verwaltung und Synchronisation der parallel arbeitenden Threads verbunden [13]. In einem Gitter können sich bei der Rekonstruktion benachbarte Zellen Vertices teilen. Damit keine Vertices doppelt berechnet und abgespeichert werden, sollen einmal berechnete Vertices mehrfach referenziert werden, um Speicher zu sparen. Dazu ist es bei der Rekonstruktion notwendig, in Nachbarzellen nach bereits erzeugten Vertices zu suchen (siehe Abbildung 3.11). Gerade das Auffinden von benachbarten Zellen ist in Octree-Strukturen nur durch eine zusätzliche Verzeigerung oder durch Suchen in der Baumstruktur möglich.

Gitterrepräsentation

Zur Vermeidung dieser Probleme wird in der Software ein Hashing-basiertes Verfahren zur Verwaltung der Gitterzellen angewendet. Jede Gitterzelle wird dabei eindeutig durch drei Indizes i, j und k identifiziert, die die Positionen im dreidimensionalen Raumgitter angeben, analog zu einem dreidimensionalen Array. Die Gitterzellen werden in einer Hashmap verwaltet. Mit Hilfe

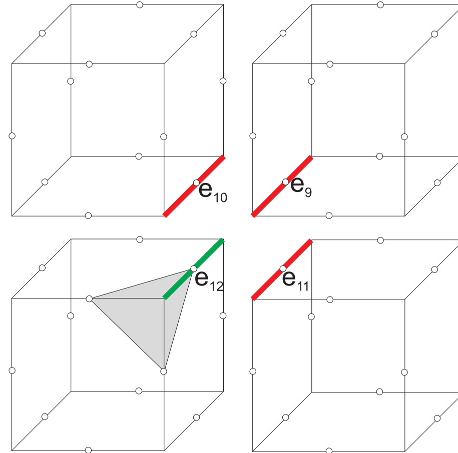


Abbildung 3.11: Suche nach bereits im Modell vorhandenen Vertices. Zur Generierung der grauen Fläche wird der Kantenvertex e_{12} benötigt. Um zu überprüfen, ob er bereits erzeugt wurde, muss kontrolliert werden, ob einer der Vertices e_{11}, e_9 oder e_{10} in den benachbarten Zellen bereits erzeugt wurde (die Nummerierung entspricht der von Lorensen und Cline [110]).

dieser Indices wird für jede erzeugte Zelle ein eindeutiger Hashwert H bestimmt:

$$H(i, j, k) = i \cdot dim_x \cdot dim_y + j \cdot dim_y + k$$

Die Konstanten dim_x und dim_y sind dabei die in x- bzw. y-Richtung vorkommenden Indizes. Diese werden anhand der achsenparellenen Bounding-Box und der der gewählten Voxelgröße d bestimmt:

$$dim_x = \lceil \frac{x_{\max} - x_{\min}}{d} \rceil$$
$$dim_y = \lceil \frac{y_{\max} - y_{\min}}{d} \rceil$$

Dabei sind $x_{\min}$, $x_{\max}$ usw. die maximalen und minimalen Koordinatenwerte der Bounding-Box in der entsprechenden Raumrichtung. Für jeden Messwert werden beim Einlesen die Indizes i, j und k bestimmt. Damit nur positive Indizes auftreten, wird das Koordinatensystem entsprechend verschoben:

$$i = \lfloor \frac{x - x_{\min}}{d} \rfloor$$

Analog werden die Indices j und k anhand der y- und z-Koordinaten berechnet. Anhand der Indices wird überprüft, ob sich in der Hashmap bereits eine Zelle für das Indextripel befindet. Ist dies nicht der Fall, wird eine neue Zelle erzeugt und in die Hashmap eingefügt. Die Zellen im Gitter sind vom Typ `VoxelGridBox`, in deren Unterklassen die verschiedenen Marching-Cubes-Varianten implementiert sind. Der Vorteil der Indizierung ist, dass sich benachbarte Zellen im Gitter leicht finden lassen, indem die Indices in der Raumrichtung, in der gesucht wird, um 1 erhöht oder erniedrigt werden. Eine zusätzliche Verzeigerung ist nicht notwendig.

In STL-Hashmaps können die Elemente in konstanter Laufzeit ermittelt werden. Die Zugriffe erfolgen daher in $O(1)$. Für die Verwaltung der Hashmap entsteht ein gewisser Speicheroverhead,

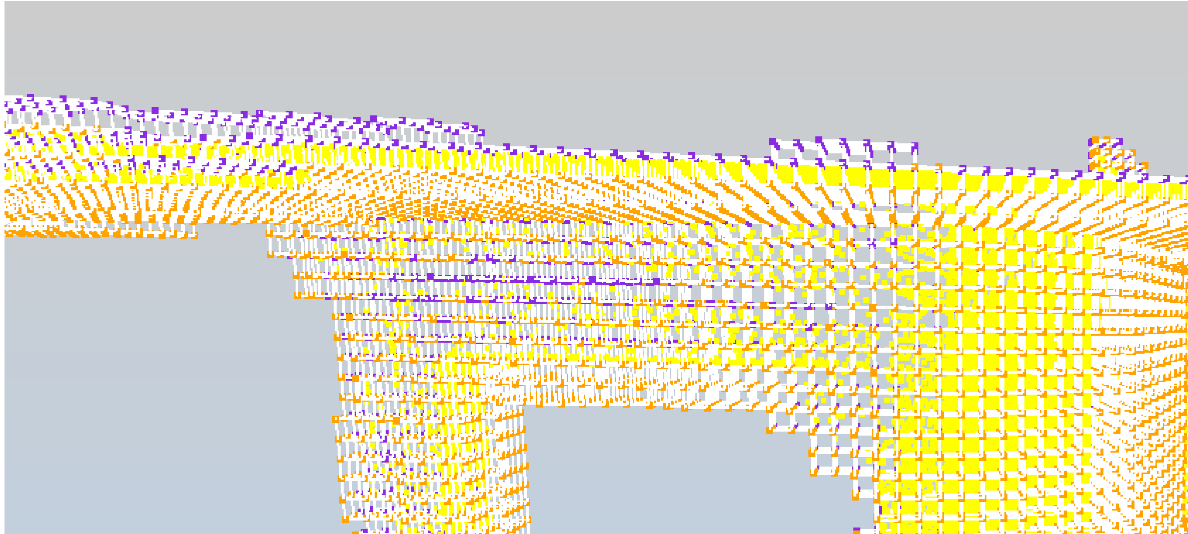


Abbildung 3.12: Visualisierung des erzeugten Gitters für einen Kurt3D-Laserscan (gelbe Punkte). Die Farbe der Eckpunkte gibt die relative Orientierung zur gescannten Oberfläche an.

der von der Implementierung abhängt. Dieser ist aber gegenüber einer vollen Verzeigerung aller Zellen mit ihren 26 Nachbarn zu vernachlässigen. Ein Beispiel für ein Gitter, das mit diesem Verfahren erstellt wurde, ist in Abbildung 3.12 dargestellt. Wie dort zu sehen ist, werden nur Gitterzellen erzeugt, die auch Messpunkte enthalten.

Suche nach Nachbarzellen und doppelten Vertices

Wie in Abbildung 3.11 gezeigt, teilen sich die Zellen im Gitter Vertices. Für eine möglichst speicherschonende Darstellung dürfen keine Vertices mehrfach gespeichert werden. Daher werden für die Eck- und Kantenpunkte indizierte Buffer verwendet, d.h. in den Zellen werden nur Verweise auf die Vertexdefinitionen abgelegt. Jede Zelle hat dafür je einen Kanten- und einen

Tabelle 3.2: Auszug aus der Tabelle zur Kodierung gemeinsamer Eckpositionen. Jede Zeile repräsentiert eine Ecke einer Zelle. Die Offsets Δ_i, Δ_j und Δ_k bestimmen eine Nachbarzelle im Gitter, v_{n1}, v_{n2} und v_{n3} sind die Positionen im Buffer der entsprechenden Zelle, wo der Vertex-Index zu finden ist, wenn er bereits erzeugt wurde.

v_i	Δ_i	Δ_j	Δ_k	v_{n1}	Δ_i	Δ_j	Δ_k	v_{n2}	Δ_i	Δ_j	Δ_k	v_{n3}
-1	0	0	1	4	0	-1	-1	6	0	0	-1	2
1	1	0	0	3	0	0	-1	5	1	1	1	3
2	0	0	-1	6	0	-1	1	4	0	1	0	0
⋮												
11	1	0	0	10	0	1	0	9	1	1	0	8

Ecken-Buffer. Die Suche nach gemeinsamen Vertices in Nachbarzellen geschieht mit Hilfe vorberechneter Lookup-Tabellen. In den Tabellen werden in jeder Zeile die relativen Positionen der Nachbarzellen und die Nummer des gemeinsamen Vertex' abgelegt. Die Nummer der Zeile entspricht dabei dem Vertex, für den Indizes in den Nachbarzellen gesucht werden sollen. Die Einträge in den Zellen sind Quadrupel, deren ersten drei Werte die relative Position der zu suchenden Nachbarzelle im Gitter angibt. Der vierte Eintrag legt fest, welche Nummer der gemeinsame Punkt in diesem Voxel hat. Ein Beispiel für so eine Codierung ist in Tabelle 3.2 gezeigt.

Das erste Quadrupel $\{-1, 0, 0, 1\}$ bedeutet, dass sich eine potentielle Definition für Vertex $\boldsymbol{e}_0$ der betrachteten Zelle in der Nachbarzelle, die durch die Indizes $i - 1$, $j + 0$ und $k + 0$ bestimmt ist, im Buffer-Eintrag für den Vertex $\boldsymbol{e}_1$ befindet. Solche Tabellen gibt es sowohl für Eckpunkte als auch für Kantenpunkte. Eckpunkte teilen sich bis zu 8 Zellen, Kantenpunkte maximal 3. Dementsprechend hat die Tabelle für Kantenpunkte 12 Zeilen und 12 Spalten, die zur Suche von Eckpunkten 8 Zeilen und 24 Spalten. Die Einträge in den Tabellen hängen von der Nummerierung der Elemente in den Zellen ab. Wenn nicht anders angeben, wird immer die in Abbildung 3.11 vorgestellte Nummerierung verwendet.

Die Voxelrepräsentationen im Gitter enthalten jeweils zwei Index-Arrays: eines für die Eckpunkte und eines für die Kantenpunkte, die auf die Positionsdefinitionen in einem globalen Vertexarray verweisen. Ist ein Vertex bereits im globalen Buffer vorhanden, wird sein Index in den gemeinsamen Zellen an den entsprechenden Positionen eingetragen. Noch nicht erzeugte Vertices werden durch einen speziellen Wert, z.B. -1, markiert. Wird in allen Nachbarzellen kein gültiger Eintrag gefunden, wird ein neuer Vertex generiert und im globalen Array abgelegt. Anschließend werden die beteiligten Zellen aktualisiert. Zu Beginn werden alle Index-Array Einträge mit "noch nicht erzeugt" belegt.

Durch diese Kodierung lassen sich alle bereits vorhandenen Vertexdefintionen in konstanter Zeit ermitteln. Alternativ könnten die Vertices auch in STL-Sets abgelegt werden, um Doppelungen zu verhindern. Diese Art der Repräsentation wird von einigen Software-Paketen, z.B. [179], intern verwendet. Das Auffinden bereits vorhandener Elemente erzeugt in solchen Implementierungen wiederum eine logarithmische Zugriffszeit, da STL-Sets intern Suchbäume zum Verwalten der gespeicherten Elemente verwenden [128]. Des Weiteren würde sich ein zusätzlicher Verwaltungsaufwand für die Indizes ergeben, da diese in einer Set-Implementierung nicht direkt zur Verfügung stehen.

3.4.2 Auswertung der Distanzfunktion

Nachdem die Gitter-Vertices erzeugt wurden, wird für jeden dieser Punkte die Distanzfunktion ausgewertet. Dies kann, wenn es die verwendete Hardware unterstützt, auch parallel erfolgen. Die Berechnung erfolgt in einer für das verwendete Verfahren spezialisierten `DistanceFunction`-Klasse. Die Ergebnisse werden in einem Buffer mit Objekten vom Typ `QueryPoint` abgelegt. Dabei handelt es sich um eine Struktur, in der jeder erzeugte Punkt im Gitter mit einem Distanzwert assoziiert wird:

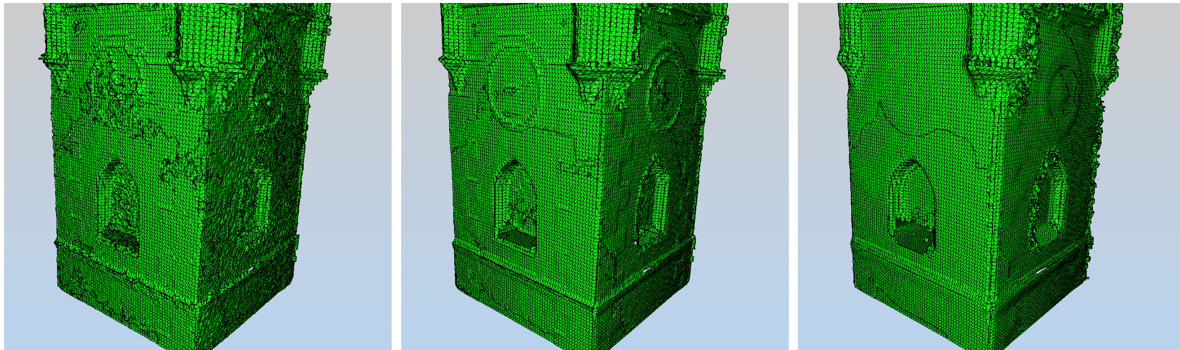


Abbildung 3.13: Beispiel einer Rekonstruktion aus einem hoch aufgelösten Laserscan mit $k_s = 1$, $k_d = 10$ und $k_d = 100$ (von links nach rechts). Zur Rekonstruktion wurden in den drei Beispielen dieselben zuvor geschätzten Normalen verwendet. Man kann deutlich erkennen, dass die ebenen Anteile mit steigenden k_d deutlich rauscharmer rekonstruiert, die Details aber verwaschen werden.

```
template<typename VertexT, typename DistanceT>
class QueryPoint
{
  VertexT position;
  DistanceT distance;
};
```

Dieser Buffer wird dann von den `VoxelGrid`-Unterklassen zur Berechnung der Dreiecksapproximation verwendet. Prinzipiell sind in dieser Struktur auch andere Metriken denkbar, da die Punktrepräsentation des Gitters und der Typ der Abstandsmetrik durch die Template-Parameter unabhängig von einander sind. Momentan wird zur Abstandsberechnung Hoppes Distanzfunktion verwendet.

Um das Verfahren unempfindlicher gegen Rauschen zu machen, wird für die Rekonstruktion nicht nur der Abstand zur nächsten Tangentialebene bestimmt, sondern es werden die k_d nächsten Tangentialebenen gesucht und der Mittelwert der Abtände dieser Ebenen zum Anfragepunkt bestimmt. Für den Spezialfall $k_d = 1$ entspricht das Hoppes Ansatz. Die Tangentialebenen nach Hoppe sind hier durch die Punkte mit ihren dazugehörigen Normalen definiert. Der zur Normalenschätzung verwendete Parameter k_n entspricht dementsprechend der k-Nachbarschaft gemäß Hoppes Definition. Durch die Mittelung wird eine weitere Filterung des Messrauschens bewirkt. Der Parameter k_d sollte insbesondere in rauscharmen Daten nicht zu hoch gewählt werden, da durch die Ausgleichsrechnung bei gleicher Auflösung Details herausgemittelt werden.

Abbildung 3.13 zeigt diesen Effekt an einer Rekonstruktion basierend auf einem hoch aufgelösten Laserscan von einem Riegl-Scanner. Für die drei dort gezeigten Beispiele wurden dieselben zuvor berechneten Normalen verwendet, allerdings wurde der k_d-Wert variiert. Das linke Bild zeigt das Ergebnis mit $k_d = 1$. Hier ist insbesondere im linken Teil ein deutliches Rauschen zu erkennen. Im mittleren Bild wurde k_d auf 10 gesetzt. Die Ebenen sind deutlich glatter, es sind aber dennoch viele Details, wie die abgesetzten Steine an den Kanten und die Zeiger der Uhr, zu erkennen. Im rechten Bild wurde k_d auf 100 erhöht. Die zuvor sichtbaren Details sind stark verwaschen,

Tabelle 3.3: Ausschnitt aus der Marching-Cubes-Triangulationstabelle. Die Zahlen referenzieren die Indices der Kantenschnittpunkte. Der Eintrag -1 signalisiert, dass keine weiteren Dreiecke folgen.

#	Dreieck 1			Dreieck 2			Dreieck 3			Dreieck 4			
0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
1	8	3	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
2	9	0	1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
⋮													
124	10	7	6	11	9	0	11	0	2	-1	-1	-1	-1
125	7	6	10	3	2	8	8	2	11	8	11	9	-1
⋮													
253	9	1	0	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
254	8	0	3	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
255	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

und an der hinteren Kante rechts im Bild sind einzelne Artefakte zu erkennen. Diese kommen daher, dass für so große k_d-Werte der Mittelwert des Abstands sich für zwei Gitterzellen kaum noch unterscheidet und somit falsche Konfigurationen in den Marching-Cubes-Zellen entstehen. Die Abbildung verdeutlicht, dass sich mit sinnvoll gewählten k_d-Werten eine offenkundige Verbesserung der Rekonstruktionsqualität gegenüber Hoppes Ansatz erreichen lässt. In der Praxis reichen dazu bereits in den meisten Datensätzen kleine k_d-Werte aus, so dass die Laufzeit des Verfahrens kaum beeinträchtigt wird. Für große k_d-Werte steigt die Laufzeit aufgrund das höheren Suchaufwands, gleichzeitig verliert die Rekonstruktion Details. Eine genaue Analyse des Laufzeitverhaltens findet sich in Abschnitt 4.3.2.

Nach der Auswertung der Distanzfunktion für alle Gitterpunkte findet die eigentliche Erstellung des Dreieckesnetzes statt, indem über alle Zellen iteriert wird und die jeweilige lokale Approximation erzeugt wird. Dazu können verschiedene Marching-Cubes-Varianten verwendet werden, die im Folgenden beschrieben werden.

3.4.3 Standard Marching Cubes

Die Rekonstruktion mittels Standard-Marching-Cubes erfolgt mit Hilfe der von Paul Bourke zur Verfügung gestellten Triangluationstabelle [28]. In dieser sind die Approximationsmuster für alle 256 möglichen Konfigurationen der Zellenecken codiert. Einen Ausschnitt zeigt Tabelle 3.3. Jede Zeile der Tabelle definiert ein Approximationsmuster für einen bestimmten Index. Die positiven Einträge referenzieren die Schnittpunkte auf den Kanten der Zelle gemäß der verwendeten Nummerierung. Jeweils drei davon definieren ein Dreieck. Der Eintrag -1 zeigt an, dass keine weiteren Dreiecksdefinitionen folgen. An der Indizierung der Einträge kann man die Komplemantärsymmetrie erkennen. Die oberen und unteren drei Zeilen referenzieren dieselben Vertices, allerdings

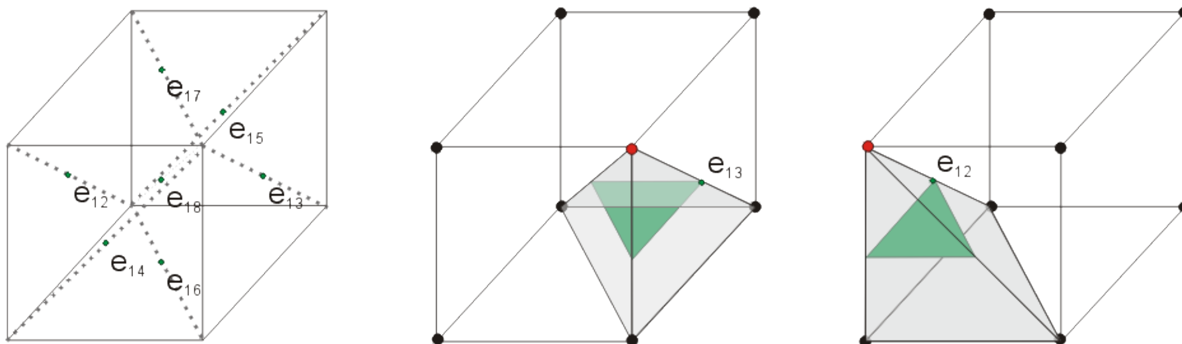


Abbildung 3.14: Vertexkorrespondenzen bei der Tetraederzerlegung. Die Nummerierung der zusätzlichen Vertices ist im linken Bild dargestellt. Die rechten beiden Würfel teilen sich den Vertex auf einer seitlichen Flächendiagonalen. Vertex $\boldsymbol{e}_{13}$ im mitteleren Würfel entspricht $\boldsymbol{e}_{12}$ im rechten Würfel..

mit vertauschter Reihenfolge, damit alle Dreiecksdefinitionen konsistent gegen den Uhrzeigersinn erfolgen.

Der erste Schritt bei der Rekonstruktion besteht darin, für jede erzeugte Zelle die passende Zeile zu finden. Der Zeilenindex wird aus einem 8-stelligen Bitmuster bestimmt, in dem jedes Bit eine Ecke der Zelle repräsentiert. Die Bits der Zellen, deren relativer Abstand zur Isofläche < 0 ist, werden gesetzt. Die Integer-Interpretation des entstandenen Bitmusters ist die gesuchte Zeile in der Tabelle. Im zweiten Schritt werden die Dreiecksdefinitionen zum Mesh hinzugefügt. Wurde ein benötigter Vertex bereits erzeugt, wird der im Indexbuffer der Kantenvertices vorhandene Eintrag verwendet. Falls nicht, wird ein neuer Vertex auf der entsprechenden Kante interpoliert. Der Index dieses neu erstellten Punktes wird anschließend an den entsprechenden Buffer-Stellen in den Nachbarzellen eingetragen.

3.4.4 Marching Tetrahedrons

Bei dieser Art der Rekonstruktion werden die Zellen des Rekonstruktionsgitters in 6 tetraederförmige Unterzellen zerlegt. Als Eckpunkte der Tetraeder dienen Ecken der würfelförmigen Zellen, d.h. es werden keine zusätzlichen Gitterpunkte verwendet. Es können also auch bei dieser Raumzerlegung die zuvor berechneten Abfragepunkte mit den dazugehörigen Distanzwerten verwendet werden. Die Dreiecksapproximation wird für jeden dieser Tetraeder, basierend auf einer vorberechneten Tabelle, analog zum Standard-Marching-Cubes-Verfahren bestimmt. Wie in Kapitel 2.3.4 erläutert, besteht diese aber nur aus 16 Einträgen und kann daher mit relativ wenig Aufwand berechnet werden. Nach Abbildung 3.14 können bei dieser Raumunterteilung auch Vertices auf den Flächendiagonalen sowie im Inneren des Würfels entstehen. Insgesamt gibt es nun 19 mögliche Vertexpositionen pro Zelle. Die Vertices auf den Flächen können von benachbarten Zellen im Gitter geteilt werden. Dementsprechend müssen die Nachbarschaftstabellen für diese Unterteilung erweitert werden, um auch diese Vertices zu berücksichtigen. Die Herausforderung bei der Implementierung besteht im Wesentlichen darin, die Lookup-Tabellen konsistent zu definieren. Die Struktur des Rekonstruktionsalgorithmus unterschiedet sich allerdings nicht

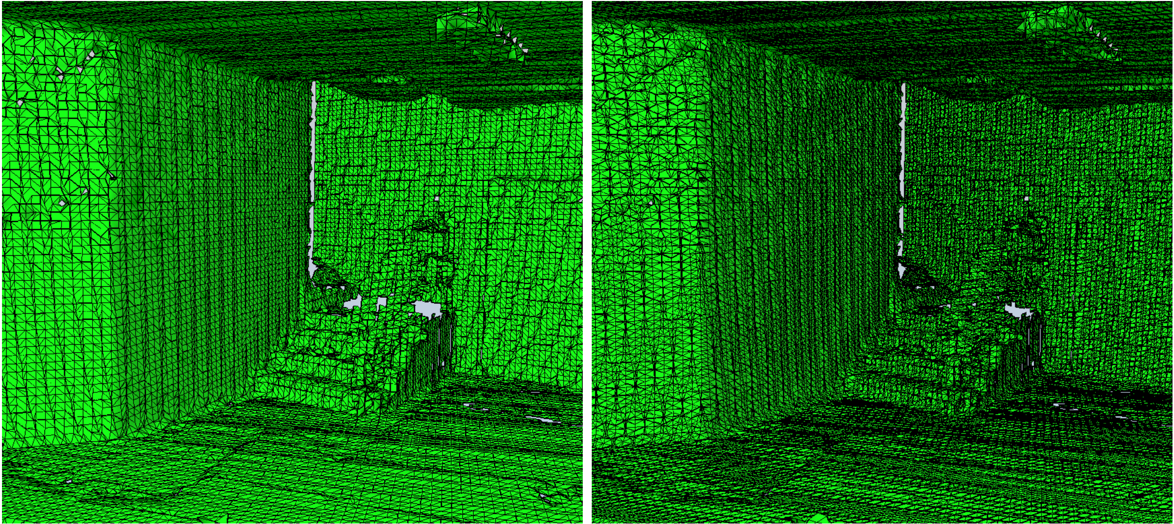


Abbildung 3.15: Vergleich zwischen Marching Cubes (links) und Marching Tetraedrons (rechts) bei gleicher Voxelgröße.

wesentlich vom Standardverfahren: Für jeden Tetraeder wird das Bitmuster der Approximation bestimmt und die dazu in der Lookup-Tabelle definierten Dreiecke werden in die übergebene Mesh-Struktur integriert.

Ein Vergleich zwischen Marching Cubes und Marching Tetraedrons ist in Abbildung 3.15 gezeigt. Bei der Rekonstruktion wurde jeweils eine Voxelgröße von 6 cm verwendet. Bei genauer Betrachtung kann man erkennen, dass bei der Marching-Tetraedrons-Rekonstruktion an einigen Stellen keine Löcher im Mesh auftauchen, dafür ist die Anzahl der Dreiecke aber beinahe viermal so hoch (81.251 vs. 318.921 Dreiecke). Dies liegt offensichtlich daran, dass der Datensatz aus vielen Ebenen besteht, in denen bei Marching Tetraedrons pro Zelle bis zu vier Dreiecke als Approximation erzeugt werden, während es bei MC lediglich zwei sind.

3.4.5 Extended Marching Cubes

Die von Kobbelt beschriebene Distanzfunktion bestimmt den Abstand der Zellenecken separat entlang der drei Basisvektoren des Gitters. Zur Interpolation der Approximationsvertices wird jeweils der Abstandswert entlang der Zellenkante verwendet. Der jeweilige Abstand der Zellenecken wird duch ein Raytracing-Verfahren nach [168] bestimmt. Dazu werden die Punkte mit einer Scheibe eines festgelegten Radius assoziiert. Der Radius wird dabei etwas größer als die Punktdichte gewählt, die Ausrichtung der Scheibe ist durch die zu dem Punkt gehörende Normale bestimmt. Der Abstand entlang einer Raumrichtung ist der erste Schnittpunkt eines ausgesandten Strahls vom Abfragepunkt entlang der entsprechenden Kante. Um Zugriff auf die Normaleninformationen zu bekommen, erhält jede Zelle einen Zeiger auf die `PointsetSurface`-Instanz, die die entsprechenden Suchbaumstrukturen zur Verfügung stellt.

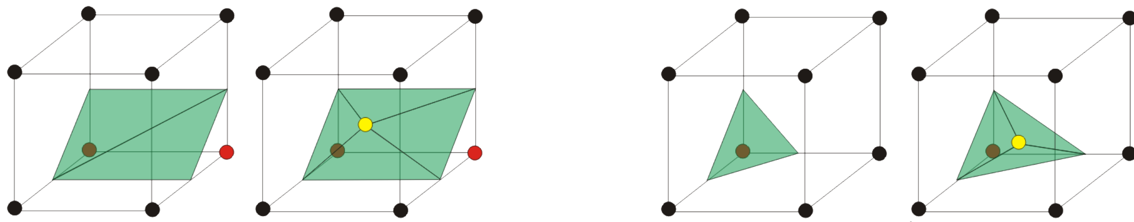


Abbildung 3.16: Approximationsmuster für Extended Marching Cubes für Kanten (links) und Ecken (rechts) im Vergleich zu den standard Marching Cubes Mustern. In beiden Fällen wird zunächst durch Verschneidung der lokalen Ebenen ein Vertex für Approximation eines scharfen Features berechnet (gelb). Dieser wird dann in der alternativen Triangulation verwendet.

Mit dem eben beschriebenen Raytracing-Ansatz kann es vorkommen, dass es keinen Schnittpunkt eines Strahles mit einer Scheibe gibt, z.B. wenn in der entsprechenden Richtung keine Datenpunkte vorhanden sind. In einem solchen Fall wird der Vertex auf der entsprechenden Kante mit Hoppes Distanzfunktion interpoliert. In der Regel sind diese Sonderfälle unproblementisch, da Vertices nur an Kanten berechnet werden, an denen ein Schnitt mit der Isofläche stattfindet, d.h., es sollten Daten innerhalb der Zelle vorhanden sein, die einen Vorzeichenwechsel an der Kante stützen, so dass der Raytracer einen Schnittpunkt findet.

Die eigentliche Rekonstruktion mit Extended Marching Cubes erfolgt in zwei Schritten. Zunächst wird überprüft, ob eine Zelle einen Knick in der Isofläche enthält, indem die beiden von Kobbelt definierten Heuristiken ausgewertet werden. Dafür ist es notwendig, die Normalen aller Punkte innerhalb der Zelle zu betrachten. Da es in den Suchbaum-Implementierungen keine Funktionen für eine volumenbasierte Suchanfrage gibt, wird als Approximation eine k-Suche mit einer relativ großen Punktzahl durchgeführt. Alle Rückgabepunkte, die außerhalb der Zelle liegen, werden bei der Auswertung der Heuristiken ignoriert. Enthält eine Zelle eine scharfe Kante oder Ecke, wird eine alternative Triangulationstabelle verwendet, in der die in diesem Fall zu verwendenden Flächenmuster kodiert sind. Diese Flächenmuster enthalten immer einen zusätzlichen Vertex, der nicht auf den Zellenkanten liegt. Weil diese Vertices Knicke innerhalb der Zelle approximieren, kann es allerdings nicht passieren, dass so ein Vertex mit benachbarten Zellen geteilt wird. Eine Suche nach bereits erzeugten Instanzen in Nachbarzellen wie z.B. bei Marching Tetraedrons ist daher nicht erforderlich.

Ecken können immer dann auftreten, wenn eine Zellecke alleine steht, d.h., alle mit ihr verbundenen Kanten haben ein anderes Vorzeichen. Dies ist u.a. bei den Marching-Cubes-Grundmustern 1 und 6 der Fall. In einem solchen Fall werden die zu den Vertices des ursprünglichen Musters nächsten Tangentialebenen verschnitten, um den Eckpunkt zu approximieren. Prinzipiell ist dabei nicht garantiert, dass es einen gemeinsamen Schnittpunkt dieser Ebenen gibt. Allerdings wird die Heuristik zur Eckendetektion nur erfüllt, wenn die lokalen Ebenen sehr spitzwinklig aufeinandertreffen, so dass die Wahrscheinlichkeit hoch ist, dass ein guter Approximationspunkt gefunden wird. Schneiden sich die Ebenen nicht, wird die Standard-Marching-Cubes-Approximation verwendet. Alle Zellen, in denen Kobbelts Verfahren angewendet wurde, werden im Gitter markiert, da nach der Erzeugung dieser initialen Muster noch eine Nachberechnung erfolgt, in der die endgültige Geometrie festgelegt wird.

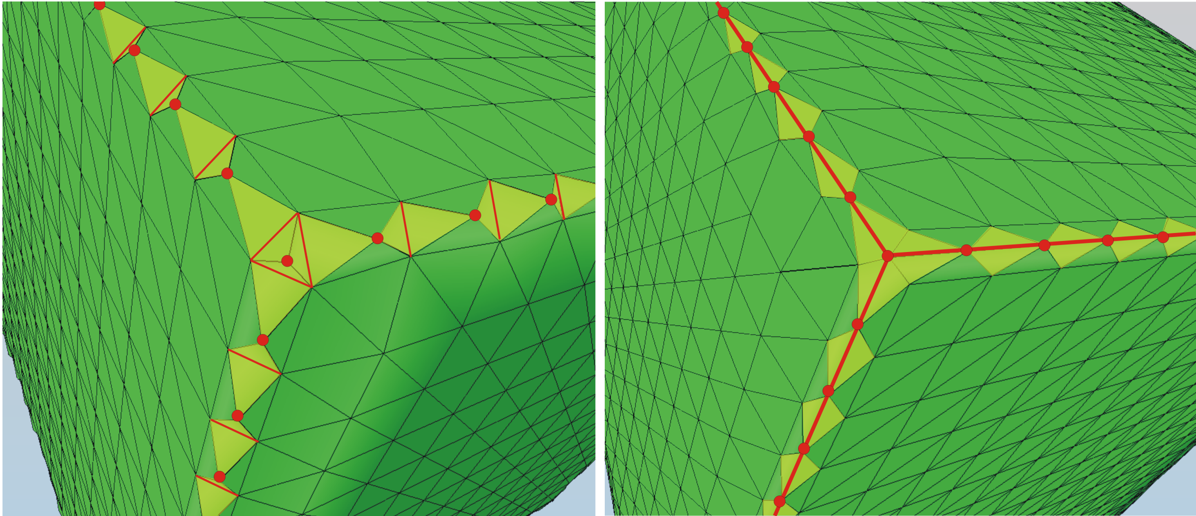


Abbildung 3.17: Approximation scharfer Konturen durch Edge-Flipping. Die zunächst erzeugten Dreiecksmuster sind im linken Bild gezeigt. Die roten Vertices stellen die geschätzten Knick- bzw. Eckpunkte dar. Duch Edge-Flipping der roten Kanten entsteht eine Approximation der scharfen Kanten (rechtes Bild).

Kanten treten immer dann auf, wenn ebene Grundmuster erzeugt werden, d.h., entlang einer Kante sind die Vorzeichen gleich, die gegenüberliegenden Vertices haben aber beide das andere Vorzeichen (z.B. Grundmuster 2 und 10). In diesen Fällen werden die Tangentialebenen zu den Kantenvertices, an deren Ende sich unterschiedliche Vorzeichen befinden, verschnitten. Der neu zu erzeugende Vertex ist in diesem Fall die Projektion des Zellmittelpunktes auf die Schnittkante. Die Struktur der Approximationsmuster mit den so berechneten Vertices ist in Abbildung 3.16 gezeigt.

Nachdem die initiale Geometrie erzeugt wurde, werden die Kanten und Knickpunte durch Edge-Flipping rekonstruiert. Dabei werden jeweils die Kanten zwischen den Verbindungsdreiecken benachbarter Zellen getauscht. Auf diese Art und Weise werden die neu erzeugten Vertices, die durch das Verschneiden der aufeinandertreffenden Ebenen erzeugt wurden, verbunden, so dass die scharfen Konturen akkurat wiedergegeben werden (s. Abbildung 3.17).

3.4.6 Planar Marching Cubes

Bei standard Marching Cubes werden die Vertices der Approximationsmuster bei der Interpolation auf den Voxelkanten verschoben. Dies kann insbesondere bei ebenen Flächen dazu führen, dass das Ende der Fläche nicht korrekt interpoliert wird. Ein Beispiel für so eine Situation zeigt das linke Bild in Abbildung 3.18. In der gezeigten Konfiguration werden die Vertices in der Rekonstruktion nur auf die “Höhe” der Fläche interpoliert. Die Begrenzung der Fläche wird nicht korrekt wiedergegeben. Durch diese Einschränkung entsteht eine Diskretisierung in der Rekonstruktion, da jeweils zwei Koordinaten eines interpolierten Vertexes durch die Gitterstruktur vorgegeben sind. Von dieser Problematik sind alle Randkanten in einem Mesh betroffen.

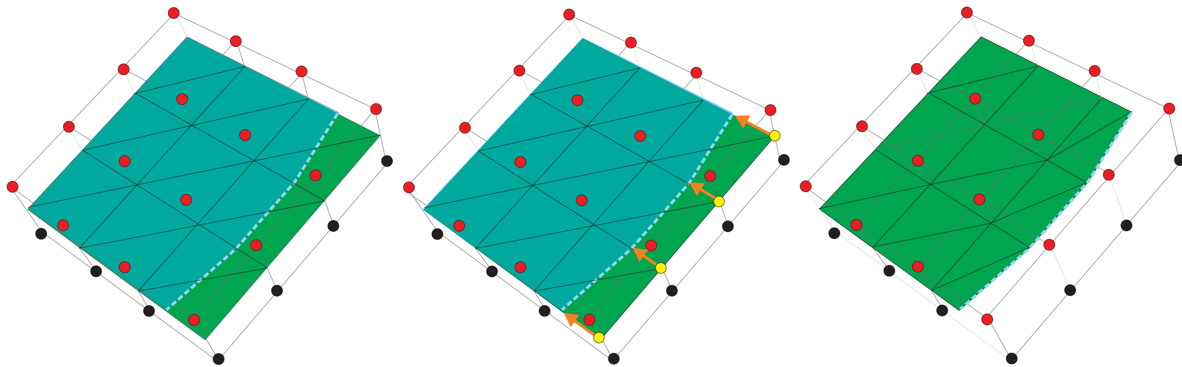


Abbildung 3.18: Prinzip von Planar Marching Cubes. Die gescannte Fläche ist blau dargestellt. Bei standard Marching Cubes wird die Begrenzung nicht approximiert. Durch Verschiebung der gelben Vertices auf die Kontur (Mitte) kann der genaue Flächenverlauf rekonstruiert werden (rechts).

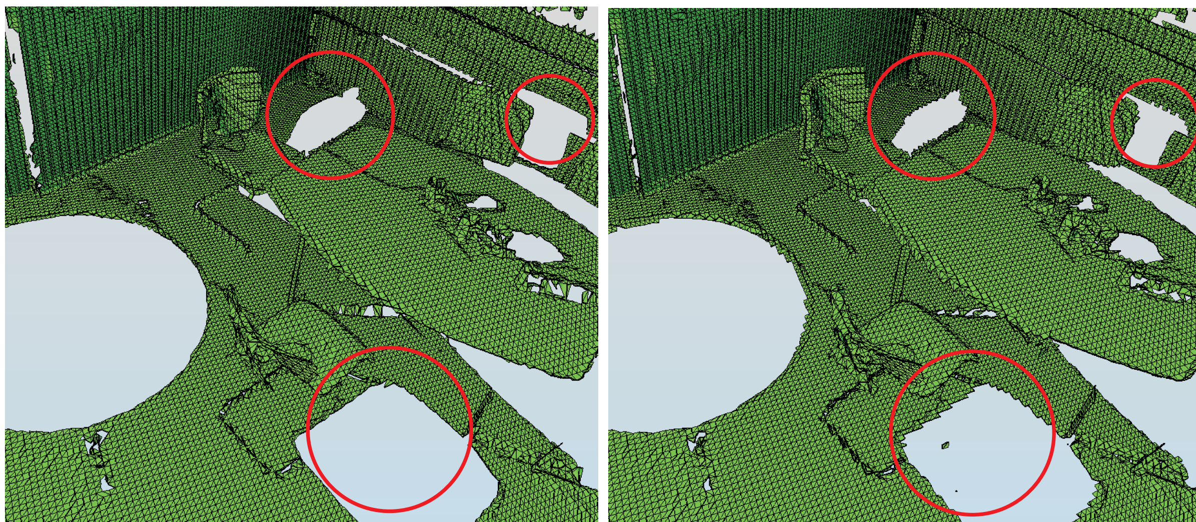


Abbildung 3.19: Rekonstruktion mit Planar Marching Cubes.

Um eine bessere Approximation zu ermöglichen, werden daher nach der initialen Mesherzeugung mit standard Marching Cubes die Vertices aller Randkanten im Mesh optimiert. Dazu werden die nächsten Datenpunkte innerhalb der betroffenen Zelle in die Ebene des Randdreiecks projiziert. Anschließend werden die Vertices der Randkante auf den nächsten Projektionspunkt gezogen. Auf diese Art und Weise bleibt die interpolierte Höhe des Isoflächenschnittes erhalten, die Vertices werden aber an die Kontur angepasst (siehe Abbildung 3.18). Dieses Verfahren nennen wir "Planar Marching Cubes". Die gescannte Fläche ist dirt in blau dargestellt. Bei standard Marching Cubes wird die Begrenzung nicht approximiert. Durch Verschiebung der gelb markierten Vertices auf die Kontur (Mitte) kann der genaue Flächenverlauf rekonstruiert werden (rechts). Die Verbesserungen, die in realen Laserscans durch dieses einfache Verfahren erreicht werden konnten, sind in Abbildung 3.19 dargestellt. Der Effekt der Optimierung ist besonders an den Konturen der Scanschatten und der Tischplatte zu erkennen (rot markiert).

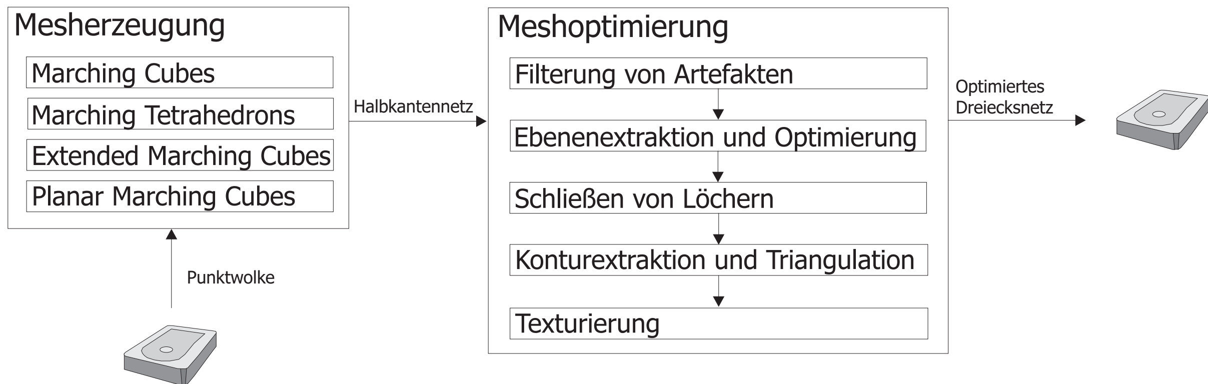


Abbildung 3.20: Mesherstellung und Optimierung in LVR. Zur Meshoptimierung werden Netze in einer Halbkantendarstellung benötigt. Diese Repräsentation wird dann in mehreren Schritten Optimiert.

3.5 Meshoptimierung und Segmentierung

In diesem Abschnitt werden die implementierten Verfahren zur Optimierung von Dreiecksnetzen beschrieben. Die Meshoptimierung erfolgt als Nachbearbeitungsschritt im Anschluss an die initiale Rekonstruktion mit einer der zuvor vorgestellten Marching-Cubes-Varianten. Die Optimierung erfolgt dabei in mehreren Schritten (siehe Abbildung 3.20): Filterung von Artefakten, Segmentierung von ebenen Anteilen und Auffüllen von Löchern, Optimierung der Schnittkanten zwischen Ebenen sowie Neutriangulation der Konturen der Ebenen zur Reduzierung der Dreieckszahl. Nach der Optimierung können Texturen aus Farbinformationen der Punktwolken erstellt werden. Alle Optimierungsschritte setzen eine Halbkantendarstellung voraus. Daher wird zunächst beschrieben, wie die bei der Rekonstruktion generierten Dreiecke konsistent in die Struktur eingebunden werden, gefolgt von einer Beschreibung der einzelnen Verfahren.

3.5.1 Aufbau einer Halbkantendarstellung

Dreiecksmeshes in einer Halbkantendarstellung werden durch die Klasse `HalfEdgeMesh` modelliert. Diese implementiert das `BaseMesh`-Interface und kann daher von allen Rekonstruktionsverfahren verwendet werden. Wie in Abschnitt 2.3.1 beschrieben, besteht diese verzeigerte Datenstruktur aus Knoten, Halbkanten und Dreiecken.

Die Knoten im Netz werden durch die Klasse `HalfEdgeVertex` repräsentiert. In dieser Klasse sind neben der Position des Knotens auch weitere Attribute wie Normale und Farbe hinterlegt. Als Erweiterung zur klassischen Halbkantendarstellung werden darüber hinaus auch Listen mit Zeigern auf alle Kanten, die in diesen Vertex hinein- oder hinauslaufen, abgelegt. Diese Informationen werden benötigt, um Konturen von ebenen Flächen verfolgen zu können.

Kanten werden durch die Klasse `HalfEdge` repräsentiert. Diese besteht aus zwei Vertices und Zeigern auf die nächste Kante bei der Traversion eines Faces gegen den Uhrzeigersinn. Des Weiteren werden Zeiger auf das Face und die Partnerkante gespeichert. Die Dreiecke im Mesh (Klas-

Algorithmus 3.2 Erzeugung einer HalfEdge Darstellung. Der Methode werden die Indizes der drei Vertices des vom Marching-Cubes-Algorithmus erzeugten Dreiecks und ein Zeiger auf eine Liste mit den Faces im Dreiecksnetz übergeben.

```
edges ← empty list of half edges
faces ← empty half edge face
for i = 0 to 3 do
    currentVertex ← vertexIndices[i]
    nextVertex ← vertexIndices[(i + 1) mod 3]
    if edge to current vertex exists then
        edges[i] ← pair-edge of edge to vertex
        edges[i].face ← currentFace
    else
        create new edge with corrensponding pair and link them
        update in and out lists of edge vertices
        edges[i] ← newly created half edge
    end if
end for
for i = 0 to 3 do
    edges[i].next ← edges[(i + 1) mod 3]
end for
face ← edges[0]
add face to mesh
```

se `HalfEdgeFace`) enthalten lediglich einen Zeiger auf eine ihrer im Uhrzeigersinn umlaufenden Halbkanten. Spezialisierungen dieser Klasse können weitere Attribute wie Materialinformationen, semantische Klassifikation usw. enthalten.

Der Aufbau des Polygonnetzes geschieht durch Aufrufe der Interface-Methoden `addTriangle` und `addVertex` in den Rekonstruktionsalgorithmen. Die übergebenen Informationen über die erzeugten Dreiecke müssen dabei konsistent in die Halbkantendarstellung integriert werden. Das Hinzufügen eines neuen Knotens verursacht keine Probleme, da doppelt vorhandene Vertices bereits bei der Rekonstruktion in Nachbarzellen erzeugt werden. Daher kann bei jedem Aufruf von `addVertex` eine entsprechende Instanz zur Knotenliste des Meshes hinzugefügt werden.

Die Integration neuer Dreiecke in die Datenstruktur erfordert mehr Aufwand, da diese mit existierenden Knoten und Kanten verlinkt werden müssen. Im ersten Schritt werden die das neue Face umrundenden Halbkanten aus den gegebenen Vertexbuffer-Verweisen erzeugt. Da es passieren kann, dass ein Vertex bereits von einer anderen Kante verwendet wird, die mit der neuen Kante verlinkt werden muss, wird zunächst überprüft, ob es bereits eine Kante gibt, die den aktuellen Vertex zeigt. Dazu wird die Liste der eingehenden Kanten durchsucht. Ist eine solche Kante vorhanden, wird sie in die Liste der umgebenden Kanten aufgenommen, und der Face-Zeiger dieser Kante wird auf das aktuelle Dreieck gesetzt. Ansonsten wird ein neues Halbkantenpaar erzeugt und die Ein- und Ausgangslisten der beiden beteiligten Vertices werden aktualisiert. Falls nötig, werden auch die Verweise auf die Partnerkanten aktualisiert. Nachdem alle umgebenden

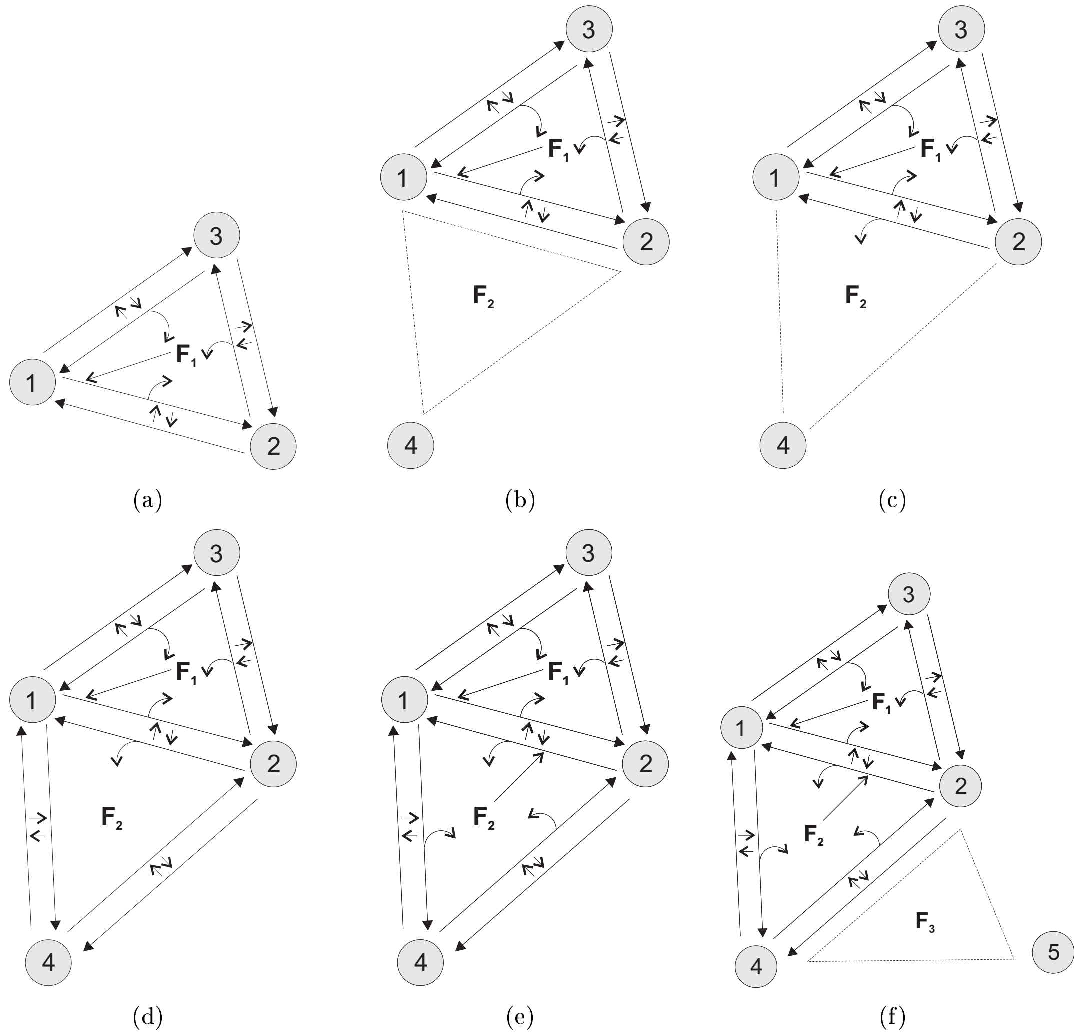


Abbildung 3.21: Erzeugung eines Halbkantennetzes. Zunächst werden das erste Face erzeugt und alle Objekte korrekt verzeigert (a). Ein neu erzeugtes Face ist noch nicht eingebunden (b). Im nächsten Schritt werden alle Halbkanten gesucht, die zuvor Randkanten waren und auf einen der Vertices des neu entstandenen Faces zeigen; die Zeiger dieser Kanten werden auf das neue Dreieck gesetzt. Es ist nun in das Netz eingebunden (c). Werden keine weiteren Kanten gefunden, die das Face bereits vorher begrenzt haben, werden die fehlenden Kantenpaare neu generiert (d) und anschließend verzeigert (e). Sobald ein weiteres Face entsteht, startet der Prozess von vorn.

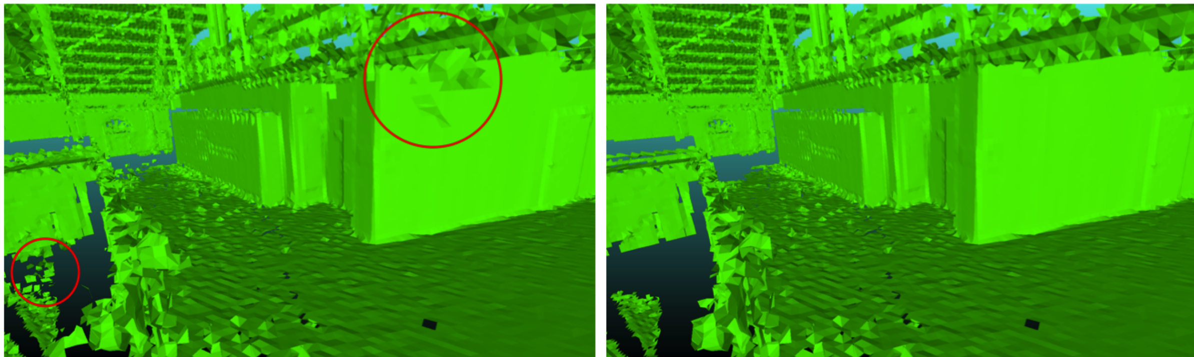


Abbildung 3.22: Automatische Entfernung frei schwebender Artefakte.

Kanten definiert und verzeigert wurden, erhält das neue `HalfEdgeFace` einen Verweis auf eine der Kanten. Der komplette Prozess ist in Abbildung 3.21 und Algorithmus 3.2 dargestellt.

Auf diese Art und Weise lässt sich effizient ein korrekt verzeigertes Netz aufbauen. Fast alle dazu benötigten Abfragen können in konstanter Zeit erfolgen. Lediglich das Durchsuchen der Listen mit den eingehenden Kanten erzeugt eine lineare Laufzeit in der Anzahl der auf den aktuellen Knoten zeigenden Kanten. Da in der Regel aber nur wenige Kanten auf einen Vertex zeigen, verlangsamt die Suche den Prozess nicht wesentlich.

3.5.2 Filterung von Artefakten

Ausreißer und Fehlmessungen bei der Aufnahme von 3D-Daten sowie Messrauscher und Interpolationsfehler bei der Rekonstruktion führen zu unerwünschten Artefakten in den Rekonstruktionen. Diese sollen möglichst automatisiert aus den Meshes entfernt werden. Dazu wurden zwei Verfahren implementiert. Mit Hilfe des ersten Algorithmus werden Artefakte duch Ausreißer entfernt, das zweite Verfahren bereitet die Entfernung von Artefakten und Messrauschen für spätere Optimierungsschritte vor.

Entfernung freischwebender Artefakte

Artefakte durch Messausreißer stellen sich üblicherweise als kleine Cluster von frei schwebenden Dreiecken dar, die nicht mit dem Rest des Meshes verbunden sind. Ein Beispiel für diese Art von Artfakten zeigt das linke Bild in Abbildung 3.22. Solche Artefakte können z.B. dann entstehen, wenn Staubpartikel in der Luft vom Laserstrahl getroffen werden. Weitere mögliche Quellen von Fehlmessungen sind Reflexionen an spiegelnden Oberflächen oder Kanten.

Zur automatischen Entfernung solcher Artefakte wurde ein Region-Growing basierter Algorithmus implementiert (*Remove Dangling Artifacts*, RDA). Im ersten Schritt des Algorithmus werden mittels Region-Growing Cluster von zusammenhängenden Dreiecken gebildet, indem die Rekursion so lange fortgesetzt wird, bis eine Randkante gefunden wird (siehe auch Kapitel 3.5.4). So

entsteht eine Menge von nicht miteinander verbundenen Regionen im Mesh. Alle Regionen, die nicht größer als ein zuvor festgesetzter Schwellwert sind, werden entfernt. Um von der Skalierung der Daten unabhängig zu sein, wird für diesen Parameter die Anzahl der Dreiecke pro Cluster und nicht die Fläche vorgegeben. Die maximale Clustergröße muss mit Bedacht gewählt werden, da bei einem zu groß gewählten Wert auch tatsächlich in der Szene vorhandene Flächen entfernt werden können. Artefakte durch Ausreißer oder Reflexionen bestehen in der Regel allerdings nur aus sehr wenigen Dreiecken, so dass in den meisten Fällen ein klein gewählter Schwellwert ausreicht, um viele unerwünschte Dreiecke aus der Rekonstruktion zu entfernen.

Ein Beispiel für die Anwendung des RDA-Algorithmus zeigt Abbildung 3.22. Im ersten Beispiel wurden Cluster bis zu einer Größe von 5 Dreiecken gelöscht. Dadurch konnten die freischwebenden Dreiecke im unteren linken Kreis entfernt werden. Bei einer Erhöhung des Schwellwertes auf 20 wurden auch die im oberen rechten Kreis vorhandenen Cluster entfernt. Das mit diesem Schwellenwert aufbereite Mesh ist im rechten Bild gezeigt.

Entfernung kleiner Regionen

Ziel des zweiten Filterverfahrens ist es, sehr kleine Regionen im Mesh zu löschen. Dabei werden auch sehr kleine Cluster, die mit dem Mesh verbunden sind, entfernt. Dadurch können zwar zunächst zusätzliche Löcher entststehen, diese werden aber in einem späteren Optimierungsschritt (siehe Abschnitt 3.5.3) automatisch geschlossen. Kleine Cluster entstehen bevorzugt an Kantenübergängen (durch Interpolationsfehler) und durch Messrauschen auf ebenen Flächen. Die Tatsache, dass man vor der eigentlichen Meshoptimierung in Kauf nimmt, zusätzliche Löcher in die erzeugten Repräsentationen zu reißen, mag auf den ersten Blick verwundern. Dass ein solches Vorgehen sinnvoll ist, hängt mit der Art und Weise zusammen, wie in der Software Löcher geschlossen werden, wie im Folgenden gezeigt wird.

Abbildung 3.23 zeigt Beispiele für Artefakte, die als kleine Regionen automatisiert entfernt werden. Die blau eingekreisten Artefakte entstehen hauptsächlich durch Interpolationsfehler an Kanten und verhindern eine scharfe Repräsentation. Die Stärke dieses Effektes hängt von verschiedenen Faktoren wie der zur Rekonstruktion verwendeten Gittergröße (große Werte verstärken den Effekt) und der Größe der k-Nachbarschaft beim Bestimmen der Distanzfunktion ab. Auch hier sorgen größere Werte tendenziell für eine stärkere Ausprägung des Effektes. Entfernt man diese Übergänge zwischen den Ebenen, entstehen Löcher, die über die Kante gehen. Diese Löcher werden beim automatischen Löcherschließen auch über Eck geschlossen, so dass bessere Kantenapproximationen generiert werden.

Die rot eingekreisten Dreiecke entstehen durch starkes Rauschen und lokale Ausreißer in den Messdaten. Rauschen in den Messdaten erzeugt in den Rekonstruktionen wellenförmige Artefakte, die in der Regel gut durch einen Ebenenausgleich entfernt werden können (siehe Abschnitt 3.5.4). Wird das Rauschen zu hoch oder sind signifikante Ausreißer in den Eingangsdaten vorhanden, entstehen auf ebenen Flächen pyramidenförmige Artefakte. Die Seiten dieser Artefakte weisen einen relativ spitzen Winkel zueinander auf, so dass sie bei einem ebenenbasierten Region-Growing nicht als Cluster erkannt werden und daher Regionen aus einzelnen Dreiecken

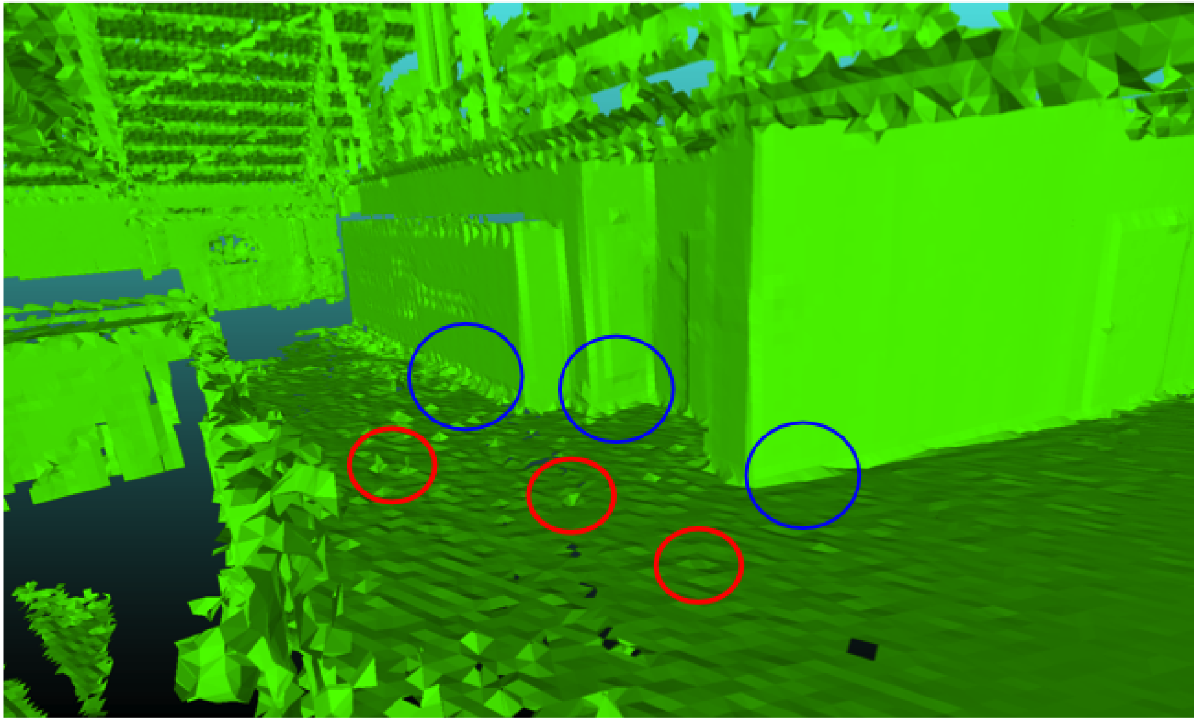
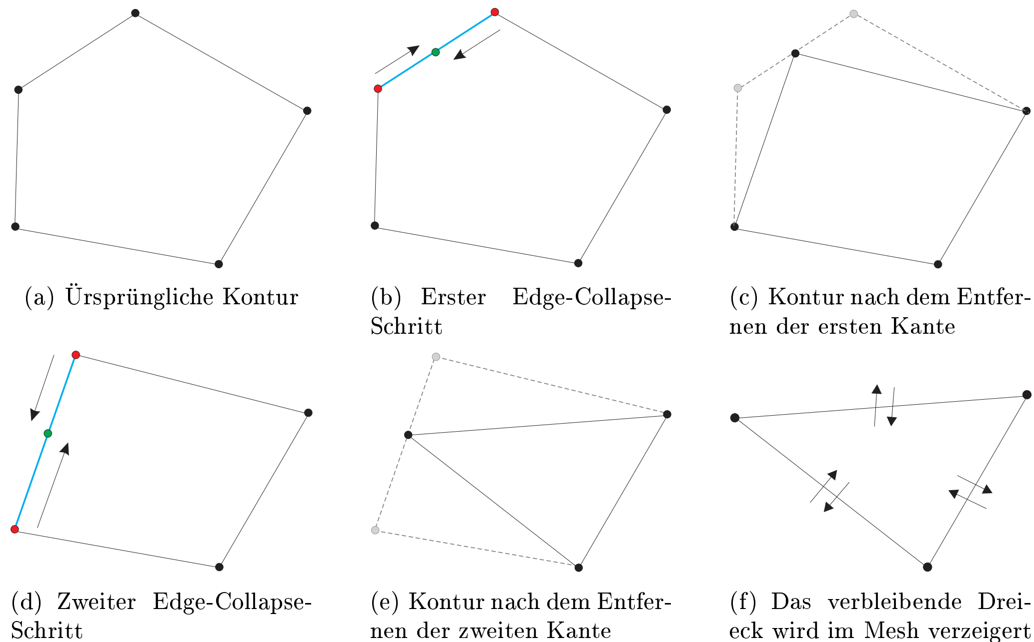


Abbildung 3.23: Kleine Regionen als Artefakte in Meshes. Die blau eingekreisten Dreiecke zeigen Artefakte durch Interpolationsfehler, die häufig an Kanten auftreten. Die roten Kreise markieren Artefakte durch lokale Ausreißer und Messrauschen.

bilden. Durch das Entfernen solcher sehr kleinen Regionen werden die Anteile in der Rekonstruktion gefiltert, die aus der umgebenden Ebene herausragen. Die so entstehenden Löcher liegen dementsprechend in dieser Ebene. Das Schließen dieser Konturen wird also anschließend eine durchgehende Repräsentation der eigentlichen Ebene erzeugen.

3.5.3 Schließen von Löchern

In Halbkantendarstellungen lassen sich Löcher in Dreiecksnetzen sehr leicht durch Edge-Collapse-Operationen schließen. Bei einem Edge-Collapse werden die zwei Vertices einer Kante auf einen neuen Vertex entlang der Kante zusammengezogen. Die Position des neuen Vertex kann entweder aus gegebenen Informationen interpoliert werden, z.B. so, dass der Abstand zu Isofläche oder der Fehler am Mesh [65, 120] minimiert wird. Liegen solche Informationen nicht vor, wird als Position die Mitte der Kante gewählt. Durch iteratives Anwenden dieser Operation auf den Kanten einer Lochkontur kann das Loch geschlossen werden. Die Kontur zieht sich dabei immer mehr zusammen, bis nur noch ein einzelnes dreieckiges Loch übrig bleibt. Die Vertices dieser Fläche werden dann im letzten Schritt in das Mesh eingefügt, so dass dieses zuvor “negative” Dreieck die letzte Lücke schließt (siehe Abbildung 3.24).



(a) Ürsprüngliche Kontur

(b) Erster Edge-Collapse-Schritt

(c) Kontur nach dem Entfernen der ersten Kante

(d) Zweiter Edge-Collapse-Schritt

(e) Kontur nach dem Entfernen der zweiten Kante

(f) Das verbleibende Dreieck wird im Mesh verzeigert

Abbildung 3.24: Beispiel für das Schließen von Löchern mittels Edge-Collapsing. Durch Zusammenziehen der beiden blau markierten Kanten der Kontur schrumpft diese auf ein dreieckförmiges Loch zusammen. Dieses wird dann als positive Fläche verzeigert in das Mesh integriert.

Das Hauptproblem ist des Erkennen von Löchern. Löcher in einer Fläche stellen sich in Halbkantenmeshes als Konturen dar, deren Außenkanten keine verzeigerten Dreiecke haben. Diese Art von Konturen kann durch Suchen nach entsprechenden Randkanten im Mesh aufgefunden werden. Wird im Mesh eine Randkante gefunden, wird durch Kantenverfolgung die Kontur aufgebaut. So eine Kontur kann in komplexen Szenen die komplette Umrandung eines zusammenhängenden dreidimensionalen Bereiches umfassen. Als Heuristik zur Identifizierung eines Loches wird die Anzahl der Kanten der Kontur betrachtet. Ist die kürzer als eine vorgegebene Maximallänge, wird davon ausgegangen, dass diese Kontur ein Loch im vorliegenden Mesh repräsentiert. Die so extrahierten Lochkonturen müssen nicht zwingend in einer Ebene liegen, so dass Löcher über angrenzende Flächen geschlossen werden können. Dementsprechend werden so auch die Löcher, die durch das Löschen der kleinen Regionen über Eck erzeugt wurden, wieder geschlossen. Der Vorteil dabei ist, dass durch die Edge-Collapse-Operationen die neu erzeugten Vertices tendenziell näher an den Kanten der an diese Löcher angrenzenden Regionen erzeugt werden, wodurch viele der schrägen Übergangsflächen eleminiert werden (siehe Abbildung 3.25).

Dieses Verfahren bereitet bei konvexen Konturen keine Probleme. Bei nicht-konvexen Konturen können Konfigurationen vorkommen, bei denen Kanten nicht entfernt werden dürfen, um eine konsistente Mesh-Struktur zu gewährleisten. So dürfen sich die nach dem Edge-Collapse entstehenden Flächen nicht mit bereits im Netz vorhandenen Dreiecken überlappen. Diese Probleme treten bei speziellen lokalen Topografien auf, die jeweils eine Sonderbehandlung erforderlich machen.

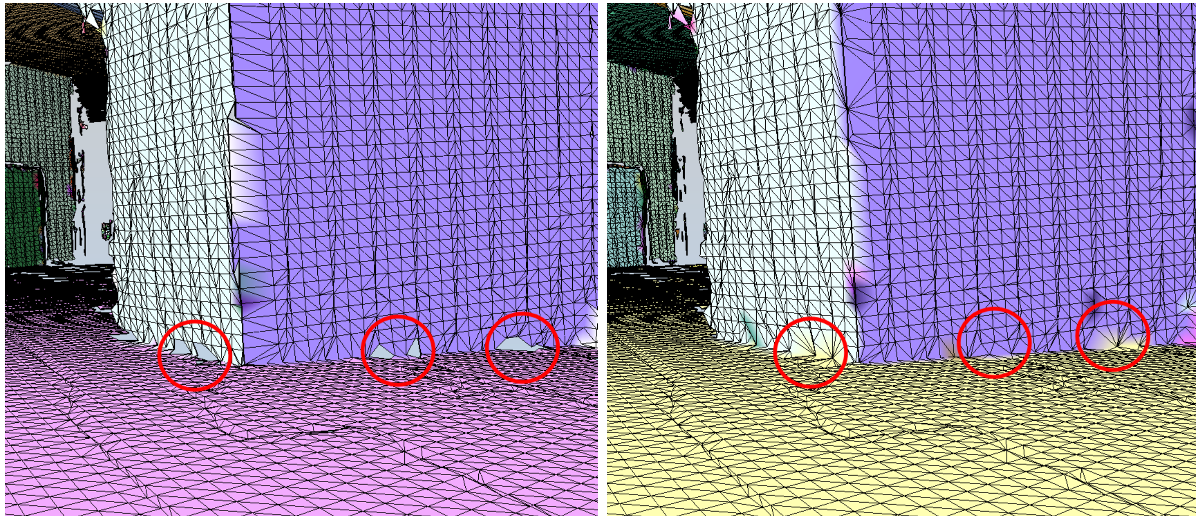


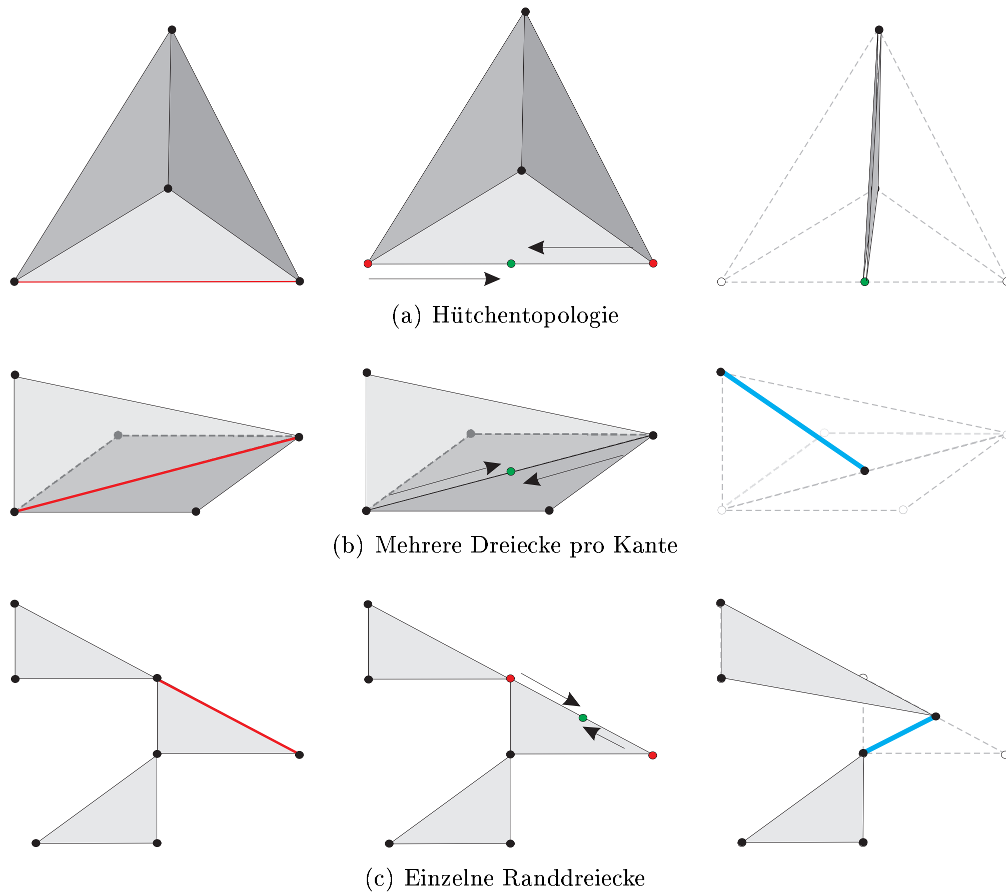
Abbildung 3.25: Beispiel für das Schließen von Löchern über Kanten. Das linke Bild zeigt das Mesh nach dem Entfernen der Übergangsdreiecke. Durch das Auffüllen der Löcher entstehen neue Dreiecke mit Vertices in der Nähe der Übergangskante.

Die behandelten kritischen Konfigurationen zeigt Abbildung 3.26. Die erste Konfiguration besteht aus einer hütchenformigen Anordnung von Dreiecken. Wird z.B. die rote Kante zusammengezogen, wird das mittlere Dreieck gelöscht und die benachbarten Dreiecke teilen sich alle Vertices. Bei dieser lokalen Konfiguration muss also darauf geachtet werden, dass das neu entstehende Dreieck nur einmal in das Mesh eingefügt wird.

Der zweite Sonderfall entsteht, wenn sich Vertices mehrere Kanten teilen. Dies kann aufgrund der Marching-Cubes-Approximationsmuster vorkommen, wenn in benachbarten Zellen entsprechende Dreiecksmuster generiert und in das Mesh eingefügt werden. Zöge man in dieser Konfiguration die rote Kante zusammengezogen, würden die unteren beiden Dreiecke entfernt und das obere Dreieck würde degenerieren, d.h. nur noch aus zwei Vertices bestehen. Solche Konfigurationen lassen sich auflösen, indem die Kontur am oberen, überlappenden Dreieck aufgespalten wird (das Dreieck speziell verzeigert doppelt eingefügt) und die so entstehenden Teilkonturen jeweils separat geschlossen werden.

Der dritte Sonderfall tritt bei alleinstehenden Dreiecken auf, die nur über gemeinsame Vertices, nicht aber über gemeinsame Kanten, mit dem Mesh verbunden sind, auf. Würde man in so einer Konfiguration die markierte Kante kollabieren, würde das Dreieck wiederum entarten. In solchen Fällen bilden die beteiligten Dreiecke minimale Löcher und können direkt als neue Flächen ins Mesh eingefügt werden.

Mit Hilfe dieser Ansätze lässt sich ein Großteil der Löcher bis zur vorgegebenen Konturgröße zusammenziehen, wie in Abbildung 3.25 zu sehen ist. Die vorher durch das Löschen der kleinen Regionen enstandenen Löcher im Mesh werden zuverlässig verschlossen. Für die gezeigte Rekonstruktion wurde eine maximale Konturlänge von 20 Kanten für die Lochumrandungen gewählt. Dieser Schwellwert für die maximale Lochgröße sollte nicht zu groß gewählt werden, da sonst



(a) Hütchentopologie

(b) Mehrere Dreiecke pro Kante

(c) Einzelne Randdreiecke

Abbildung 3.26: Problematische Topologien beim Edge-Collapsing. Bei der "Hütchentopologie" würden durch Zusammenziehen der roten Kante zwei Dreiecke aufeinander liegen. Teilen sich mehrere Dreiecke eine Kante, entsteht ein degeneriertes Face. Bei alleinstehenden Randdreiecken führt das Zusammenziehen bestimmter Kanten ebenfalls zu degenerierten Dreiecken.

unter Umständen Löcher in den Meshes entfernt werden, die eigentlich nicht geschlossen werden sollten (z.B. Fenster).

Eine mögliche Erweiterung des Verfahrens wäre es, über weitergehende Überlegungen die zu schließenden Löcher zu identifizieren. Dies könnte neben geometrischen Überlegungen auch durch Erkennung von möglichen Abschattungen bei bekannter Scannerpose [147] passieren. Eine weiterer Ansatz ist die Auswertung semantischer Informationen, z.B. "Eine Wand geht hinter einem Möbel weiter". Solche Überlegungen erfordern allerdings eine Menge an Informationen, die aus den Rekonstruktionen gewonnen werden müssen. Der wesentliche Beitrag dieser Arbeit soll in der automatischen Erzeugung von geeigneten Karten für solche weitergehenden Analysen liegen, daher werden diese Überlegungen an dieser Stelle nicht weiter verfolgt.

3.5.4 Extraktion und Optimierung zusammenhängender planarer Regionen

Typische Einsatzumgebungen von mobilen Robotern enthalten viele planare Anteile: das Innere von Gebäuden besteht größtenteils aus ebenen Flächen (Fußboden, Decken, Wände, Tischoberflächen usw.), aber auch viele von Menschen bebaute Flächen außerhalb von Gebäuden erfüllen diese Annahme. Diese Randbedingung wird im ersten Optimierungsschritt ausgenutzt, um die Anzahl der Dreiecke in den initial erzeugten Dreiecksnetzen erheblich zu reduzieren. Dazu werden zunächst Dreiecke, die in einer gemeinsamen Ebene liegen, mit einem Region-Growing-Algorithmus zusammengefasst. Die Dreiecke dieser Regionen werden anschließend in die gemeinsame Ausgleichsebene gezogen, um die Flächen zu glätten. Die Konturen der so erzeugten Cluster werden anschließend in eine optimierte Polygondarstellung überführt und neu trianguliert, so dass alle Flächen im Mesh wieder in einer Dreiecksrepräsentation vorliegen.

Region-Growing

Für den verwendeten Region-Growing-Algorithmus wird vorausgesetzt, dass die gegebenen Meshes in einer Halbkantendarstellung vorliegen. Der Algorithmus startet mit einem zufällig gewählten Dreieck und überprüft, ob angrenzende Dreiecke in derselben Ebene liegen. Ist das der Fall, wird von diesen Flächen aus rekursiv eine neue Suche nach Dreiecken begonnen, die ebenfalls in der Ausgangsebene liegen. Um zu verhindern, dass Dreiecke doppelt geprüft werden, wird jedes besuchte Dreieck markiert. Die Rekursion wird so lange fortgesetzt, bis ein Dreieck entdeckt wurde, das nicht mehr in der ursprünglichen Ebene liegt, oder eine Randkante im Mesh erreicht wurde. Solche Kanten bilden die Kontur einer ebenen Region im Netz und werden in einer Liste von Konturkanten abgelegt, aus denen später die Polygondarstellung der Umrandung gewonnen wird. Die Rekursion bricht ebenfalls ab, sobald ein bereits besuchtes Dreieck gefunden wurde. Die Kanten zu bereits besuchten Dreiecken bilden allerdings keine Begrenzung der Fläche und werden dementsprechend nicht in die Randkantenliste aufgenommen. Der Pseudocode für das Region-Growing-Verfahren ist in Algorithmus 3.3 gezeigt.

Der rekursive Ansatz dieses Verfahrens kann bei sehr großen zusammenhängenden Flächen mit vielen Dreiecken dazu führen, dass eine hohe Rekursionstiefe aufgebaut wird, bevor eine Begrenzungskante gefunden wird. Dies kann unter Umständen zu einem Stack-Overflow und somit zu einem Absturz des Programms führen. Aus diesem Grund wird die Rekursionstiefe bei der Implementierung begrenzt. Sobald die maximale Tiefe erreicht ist, wird das letzte besuchte Dreieck in eine Liste noch zu behandelnder Dreiecke aufgenommen und die Rekursion abgebrochen. Anschließend wird eine neue Suche gestartet. Sobald alle Rekursionen beendet sind, wird die Liste der noch offenen Dreiecke abgearbeitet, d.h. von diesen Dreiecken werden wiederum neue Suchen gestartet. Das qualitative Verhalten des Algorithmus wird dadurch nicht verändert, durch die Begrenzung der Rekursionstiefe werden Fehler durch Stack-Overflows aber ausgeschlossen.

Um effizient festzustellen, ob sich zwei Dreiecke in einer Ebene befinden, werden ihre Normalen betrachtet. Ist der Winkel zwischen zwei betrachteten Normalen $\boldsymbol{n_1}$ und $\boldsymbol{n_2}$ kleiner als ein Schwellwert, wird angenommen, dass die dazugehörigen Dreiecke in einer Ebene liegen. Bekanntlich gibt das Skalarprodukt zwischen zwei Normalen den Kosinus des eingeschlossenen Winkels

Algorithmus 3.3 Der Region-Growing-Algortihmus zum Zusammenfassen ebenen Region in Half-Edge-Meshes.

```
function SIMPLIFY
    for all faces do
        current face ← visited
        FUSE(current normal, current face, currentList)
        borderLists ← currentList
        CREATEPOLYGON(border list)
        currentList ← empty
    end for
end function

function FUSE(start normal, current face, list of borders)
    current face ← visited
    for all neighbors of current face do
        angle ← start normal · neighbor normal
        if angle < ϵ and neighbor not visited then
            FUSE(start normal, neighbour, listOfBorders)
        else
            list of borders ← border edge to neighbor
        end if
    end for
end function
```

wieder, so dass direkt über diesen Wert ϵ parametriert wird. Es werden also alle Flächen zusammengefasst, die die Bedingung $\boldsymbol{n_1} \cdot \boldsymbol{n_2} < \epsilon$ erfüllen.

Die Größe von ϵ beeinflusst dabei maßgeblich die Robustheit des Verfahrens gegen Rauschen. In verrauschten Daten fluktuieren die Normalen der erzeugten Dreiecke auch in Ebenen, so dass in diesen Datensätzen tendenziell ein höherer Wert gewählt werden sollte. Wird ϵ allerdings zu niedrig gewählt, kann es passieren, dass auch gekrümmte Flächen zusammengefasst werden. Ein Beispiel für diesen Effekt ist in Abbildung 3.27 dargestellt. Die Rekonstruktion in der oberen Reihe zeigt einen Bürostuhl und einen Rollcontainer in einem Vorlesungssaal, berechnet aus einem hoch aufgelösten Leica-Laserscan. Die extrahierten Cluster sind dabei in unterschiedlichen Farben dargestellt. Im linken Bild ($\epsilon = 0.85$) werden die leicht gekrümmten Flächen des Sitzes und der Rückenlehne des Stuhls noch zu einem Cluster zusammengefasst. Im rechten Bild wurde ϵ auf 0.99 erhöht. In diesem Fall entstehen mehrere kleine Cluster in den zuvor zusammengefassten Bereichen, die ebenen Bereiche werden allerdings auch bei diesem Wert noch sehr gut segmentiert.

Die untere Zeile der Abbildung zeigt Rekonstruktionen, die mit einer Kinect-Kamera aufgenommen wurden. Beim Clustering wurden dieselben Parameter wie zuvor verwendet. Aufgrund des hohen Rauschens in den Eingangsdaten werden mit einem Schwellenwert von $\epsilon = 0.99$ bis auf die Bodenfläche kaum zusammenhängende Cluster gefunden (linkes Bild). Im rechten Bild wurde ϵ auf 0.85 abgesengt. Mit diesem Wert lässt sich auch in den verrauschten Daten eine gute Segmentierung erreichen.

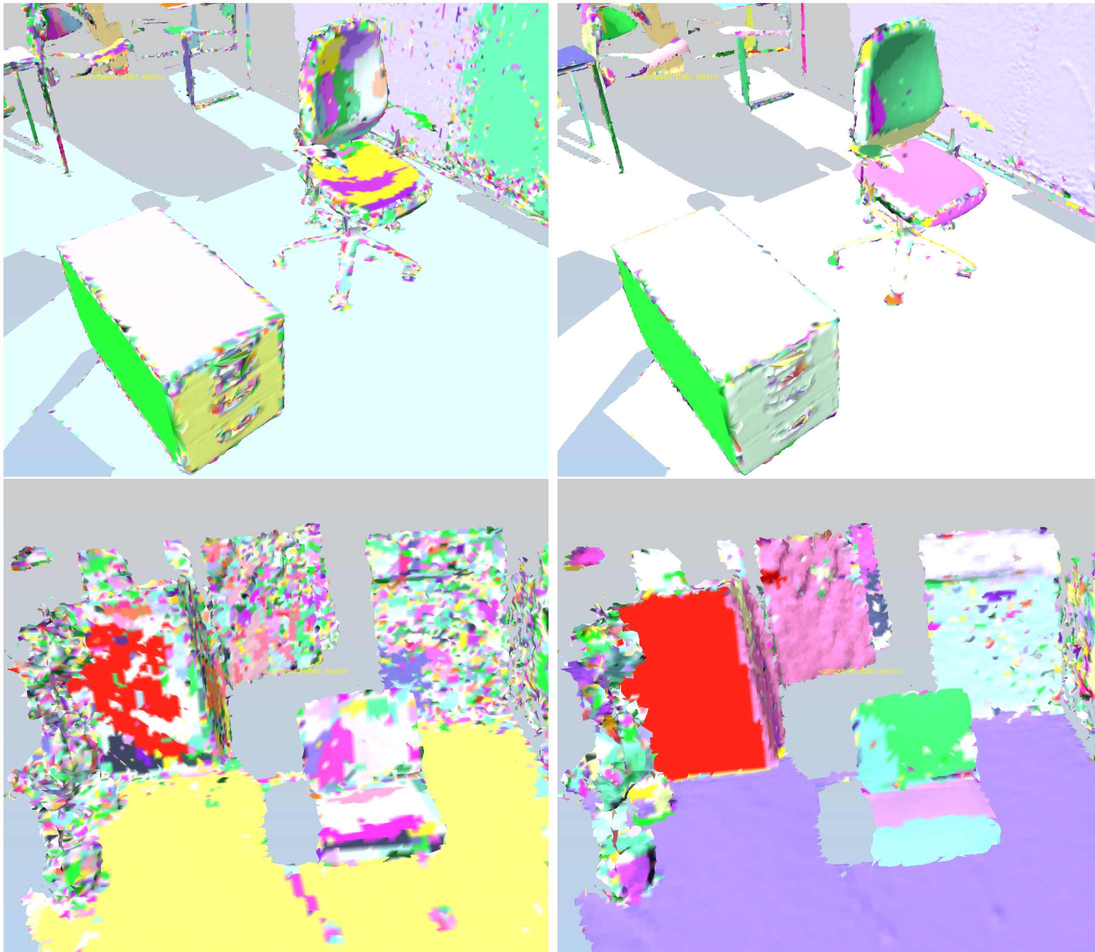


Abbildung 3.27: Ergebnisse des Region-Growing basierten Clusterings am Beispiel verschiedener Eingangsdaten. Die obere Reihe zeigt die ermittelte Clusterung in einem Leica-Laserscan (15 Mio. Punkte) mit verschiedenen Schwellwerten (links: $\epsilon = 0.99$, rechts $\epsilon = 0.85$). Die untere Reihe zeigt das Clustering mit den selben Schwellenwerten auf einer Kinect-Punktwolke (250.000 Punkte).

Diese beiden Beispiele zeigen, dass die Wahl von ϵ stark von der Qualität der Eingabedaten abhängt. In verrauschten Daten ist für eine gute Segmentierung ein niedrigerer Wert erforderlich. Wird ϵ allerdings zu niedrig gewählt, besteht die Gefahr, dass auch nicht-planare Flächen zusammengefasst werden. In der Praxis hat sich für die meisten Eingabedaten (terrestrische Scanner, rotierende SICK-Scanner, Kinect) ein Wert von $\epsilon = 0.85$ als guter Ausgangswert erwiesen.

Flächenoptimierung

Der gerade beschriebene Region-Growing-Algorithmus erzeugt eine Menge von Dreiecksclustern, von deren Dreiecken angenommen wird, dass sie sich in einer gemeinsamen Ebene befinden. Der nächste Schritt der Optimierung besteht darin, die Vertices dieser Dreiecke in die gemeinsame

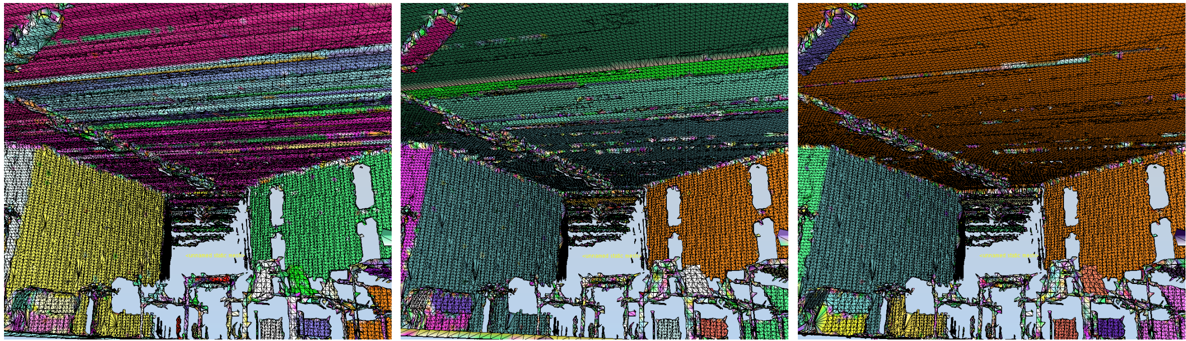


Abbildung 3.28: Iterativer Ebenenausgleich der vom Region-Growing-Algorithmus gefundenen Dreieckscluster. Das linke Bild zeigt die initial ermittelte Gruppierung. Im mittleren Bild wurden die Vertices der Dreiecke in die Ausgleichsebenen projiziert. Das rechte Bild zeigt das Ergebnis nach der Konvergenz des Verfahrens.

Ausgleichsebene zu projizieren. Auf diese Art und Weise werden die Unebenheiten, die sich aus Messrauschen und Interpolationsfehlern ergeben, ausgeglichen. Ziel dieses Vorgehens ist es, die gefundenen Flächen durch ideale Ebenen zu ersetzen. Dabei wird in Kauf genommen, dass unter Umständen geometrische Details, die in den Ausgangsdaten vorhanden waren, herausgemittelt werden. Dazu zählen z.B. Fugen zwischen Bodenfliesen oder die welligen Oberflächen von Natursteinen. In hoch aufgelösten Scans können diese erfasst werden, und auch die Rekonstruktionen sind bei hoch gewählter Auflösung in der Lage, derartige Feinheiten in der Geometrie wiederzugeben.

In der Robotik ist eine solch hochauflösende Darstellung nicht erforderlich und auch mit Blick auf die benötigte Rechen- und Speicherkapazität, auch nicht sinnvoll. So ist beispielsweise für einen mobilen Roboter im Wesentlichen die Befahrbarkeit der Bodenebene interessant. Für Anwendungen, in denen eine möglichst realitätsnahe Wiedergabe erforderlich ist, existieren Verfahren, mit denen sich solche Feinstrukturen realistisch auf einer vereinfachten Geometrie darstellen lassen, z.B. fotorealistische Texturen in Kombination mit Displacement Mapping, Normal Mapping oder anderen Verfahren aus der Computergrafik. Details dazu können in den gängigen Lehrbüchern gefunden werden [50, 174].

Die Ausgleichsebene eines Clusters wird mit Hilfe des RANSAC-Verfahrens bestimmt, das auch zur Schätzung der Punktnormalen verwendet wird. Nachdem die Parameter der Ebene bestimmt wurden, werden alle Vertices in diese Ebene projiziert, wodurch der erwünschte Glättungseffekt eintritt. Da sich durch die Projektion der Dreiecksvertices die Normalen der entsprechenden Faces verändern, kann es passieren, dass nach dem Ausgleich zwei benachbarte Flächen nun das Normalenkriterium des Region-Growing-Algorithmus erfüllen und dementsprechend zusammengefasst werden können. Um diese Fälle zu berücksichtigen, wird nach der Projektion ein weiteres Region-Growing angestoßen und die Ausgleichsrechnung auf den eventuell neu entstandenen Clustern erneut durchgeführt. Diese Schritte werden so lange wiederholt, bis das Verfahren konvergiert, d.h. keine neuen planaren Regionen mehr gefunden werden.

Algorithmus 3.4 Pseudocode zur Ebenenoptimierung.

```
function OptimizePlanes
    while new clusters found do
        for all planar clusters do
            currentPlane ← calcRANSACPlane(clusterVertices)
            for all faces in cluster do
                project(vertices, currentPlane)
            end for
        end for
    end while
end function
```

Die Auswirkungen der einzelnen Schritte sind in Abbildung 3.28 am Beispiel eines Kurt3D-Laserscans dargestellt. Das linke Bild zeigt die Ergebnisse des Clusterings. Besonders im Bereich der Decke enstehen durch die wellenförmigen Artefakte des verwendeten Laserscanners mehrere Cluster. Das mittlere Bild zeigt die Projektion der entsprechenden Dreiecksvertices in die berechneten Ausgleichsebenen. Man kann deutlich erkennen, dass die zuvor vorhandenen Wellen ausgeglichen werden und alle Dreiecke eines Clusters nun eine ebene Fläche bilden. Das rechte Bild zeigt das Ergebnis der iterativen Anwendung des Verfahrens bis zur Konvergenz. Die zuvor getrennten Cluster an der Decke wachsen zusammen.

Optimierung von Schnittkanten

Nach der Detektion der ebenen Flächen werden die Schnittkanten zwischen den Ebenen optimiert. Ziel dieses Schrittes ist, scharfe Übergänge zwischen den Flächen zu repräsentieren. Durch die Interpolation der Isofläche und der anschließenden Rekonstruktion mit Marching Cubes werden scharfe Kanten nicht realitätsgetreu abgebildet, sondern abgerundet dargestellt. Dieser Effekt kann zwar durch die Verwendung des Extended-Marching-Cubes-Ansatzes abgemildert werden, es hat sich aber gezeigt, dass gerade in verrauschten Datensätzen die Heuristiken zur Kantenerkennung oft nicht ausreichend sind und nicht alle Knickkanten erkannt werden.

Mit Hilfe der RANSAC-Optimierung wurden im vorherigen Schritt bereits die Parameter zur mathematischen Beschreibung der Ebenen in der Rekonstruktion bestimmt. Die Berechnung der Schnittgeraden erfolgt mit der in [52] beschriebenen Methode: Schneiden sich zwei Ebenen E_1 und E_2 mit den Ebenengleichungen $\boldsymbol{n_1} \cdot \boldsymbol{x} = d_1$ und $\boldsymbol{n_2} \cdot \boldsymbol{x} = d_2$ in einer Geraden $G = \boldsymbol{P} + t\boldsymbol{d}$, muss die Richtung $\boldsymbol{d}$ dieser Geraden senkrecht zu den Normalen beider Ebenen sein, da sie in beiden Ebenen liegt. Es folgt also: $\boldsymbol{d} = \boldsymbol{n_1} \times \boldsymbol{n_2}$. Zur Bestimmung des Aufpunktes $\boldsymbol{P}$ der Ebene wird die Tatsache ausgenutzt, dass $\boldsymbol{P}$ sich in einer Ebene senkrecht zu $\boldsymbol{d}$ und in E_1 und E_2 befindet. Eine solche Ebene wird durch die Normalen $\boldsymbol{n_1}$ und $\boldsymbol{n_1}$ der beteiligten Ebenen aufgespannt, es gilt also: $\boldsymbol{P} = k_1\boldsymbol{n_1} + k_2\boldsymbol{n_2}$. Die zwei Parameter k_1 und k_2 lassen sich durch das Lösen des folgenden Gleichungssystems bestimmen:

$$\boldsymbol{n_1} \cdot (k_1\boldsymbol{n_1} + k_2\boldsymbol{n_2}) = d_1$$
$$\boldsymbol{n_2} \cdot (k_1\boldsymbol{n_1} + k_2\boldsymbol{n_2}) = d_2$$

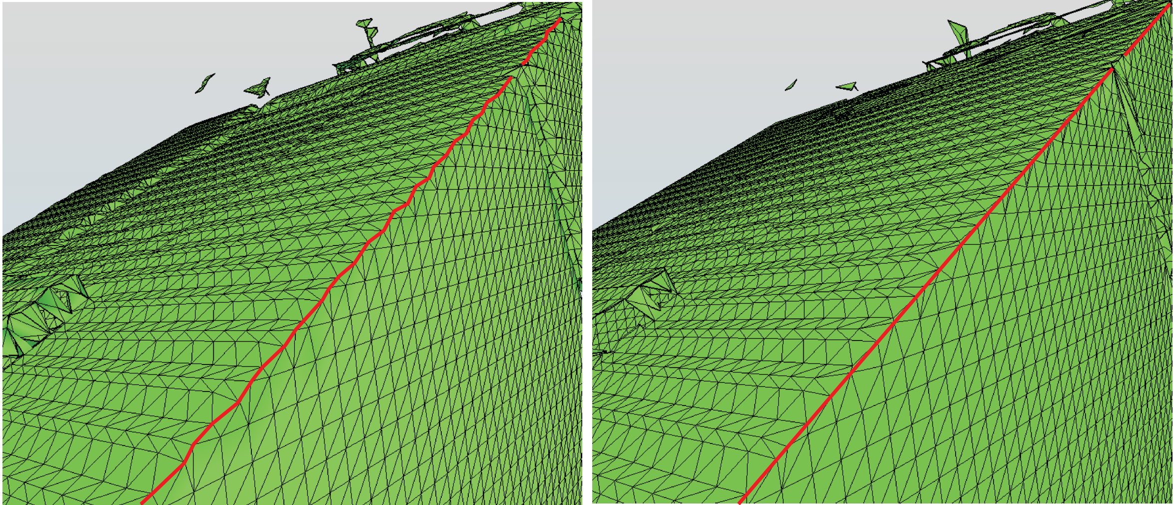


Abbildung 3.29: Ergebnis einer Schnittkantenoptimierung

Dieses Gleichungssystem lässt sich wie folgt lösen:

$$k_1 = \big(d_1(\boldsymbol{n_2} \cdot \boldsymbol{n_2}) - d_2(\boldsymbol{n_1} \cdot \boldsymbol{n_2})\big)/t$$
$$k_2 = \big(d_2(\boldsymbol{n_1} \cdot \boldsymbol{n_1}) - d_1(\boldsymbol{n_2} \cdot \boldsymbol{n_2})\big)/t$$

mit $t = (\boldsymbol{n_1} \cdot \boldsymbol{n_1})(\boldsymbol{n_2} \cdot \boldsymbol{n_2}) - (\boldsymbol{n_1} \cdot \boldsymbol{n_2})^2$. Mit Hilfe dieser Gleichungen lässt sich aus den zuvor geschätzten Ebenenparametern also leicht die exakte Schnittgerade bestimmen. Befindet sich nun die Randkante eines planaren Clusters genau zwischen zwei angerenzenden Regionen, werden ihre Vertices auf die Ausgleichsgerade projiziert. Der Projektionspunkt $\boldsymbol{V'}$ des Vertex $\boldsymbol{V}$ auf die Schnittgerade G berechnet sich dann nach Definition des Skalarprodukts wie folgt:

$$\boldsymbol{V'} = \boldsymbol{P} + \big((\boldsymbol{V} - \boldsymbol{P}) \cdot \boldsymbol{d}\big)/|\boldsymbol{d}|^2$$

Die Auswirkungen der Kanteninterpolation auf die Rekonstruktion demonstriert Abbildung 3.29. Die Bilder zeigen einen Ausschnitt aus der Rekonstruktion eines Büroscans. Im linken Bild ist der Verlauf der rekonstruierten Kante zwischen einer Wand und der Deckes des Raumes vor der Optimierung markiert. Es ist deutlich zu erkennen, dass die Vertices keine gerade Linie bilden. Durch die Interpolation und Rekonstruktion mit den Marching-Cubes-Mustern gibt es regelmäßige "Einknicke" im Linienverlauf. Nach der Berechnung der Schnittkanten und Vertexoptimierung ergibt sich eine nahezu perfekte Gerade.

Konturextraktion

Zur Datenreduzierung werden aus Konturen der optimierten planaren Regionen Polygondarstellungen gewonnen. Die Liste der Randkanten, die mit Hilfe des Region-Growing-Ansatzes erstellt wird, ist allerdings unsortiert. Daher werden die Kanten zunächst sortiert, indem von einer zufällig gewählten Startkante aus die Kontur verfolgt wird, zu der sie gehört. Dies ist durch die

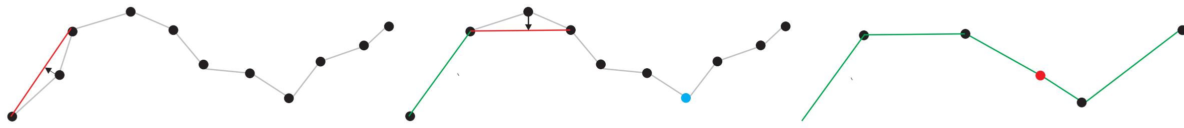


Abbildung 3.30: Polygonoptimierung durch iteratives Entfernen von Vertices in Dreiersegmenten (rote Linien). Vertices, die nicht weiter als eine gegebene Maximaldistanz vom betrachteten Segment entfernt sind, werden gelöscht. Im gezeigten Beispiel wäre der blaue Vertex im mittleren Bild außerhalb der Toleranz und bliebe erhalten. Da die Dreiertupel nacheinander geprüft werden, kann es sein, das noch redundante Vertices übrig bleiben (rote Punkte im rechten Bild).

erweiterte Verzeigerung der intern verwendeten Halbkantendarstellung leicht möglich, da jede Kante Zeiger auf ihre Nachfolgekanten abspeichert. Ausgehend von der Startkante wird die Kontur so lange weiter verfolgt, bis sie geschlossen ist. Sind nach der Konturverfolgung noch Kanten in der Liste, hat die entsprechende Region mehrere Konturen. Dies kommt immer dann vor, wenn es innerhalb der Kontur Löcher gibt.

Mit Hilfe der Konturverfolgung lassen sich zwar polygonale Darstellungen erreichen, diese sind aber nicht optimal, da gerade Kanten durch die Diskretisierung der ursprünglichen Dreiecke unnötig oft unterbrochen sind. Daher müssen die gewonnenen Konturen nachbearbeitet werden. In der Software wird dazu die "Polyline Simplification"-Bibliothek (`psimpl`) von Elmar de Koning [46] verwendet. Diese C++-Bibliothek benutzt Templates zur Modellierung der Vertices. Einzige Randbedingung ist, dass die verwendeten Datenstrukturen zur Repräsentation der Polygonvertices Koordinatenzugriffe mittels `[]`-Operator unterstützen. Sie ist daher kompatibel mit den intern in der Software verwendeten Datenstrukturen und kann direkt genutzt werden.

In der `psimpl`-Bibliothek sind verschiedene Algorithmen zur Optimierung von Polygondarstellungen implementiert. Diese werden im Folgenden kurz vorgestellt. Neben relativ naiven Algorithmen, die anhand von Abstandskriterien zu Segmenten aus jeweils drei Punkten redundante Vertices aus den Polygonzügen entfernen (senkrechter Abstand, Linienabstand nach Reumann-Witkam[158], Opheim- und Lang-Verfahren [103, 145]), wird auch der bekannte Douglas-Peucker-Algorithmus [44] implementiert. Die nachfolgende Beschreibung der verschiedenen Verfahren basiert auf der Dokumentation von `psimpl` [100]. Eine detaillierte Diskussion der einzelnen Algorithmen ist in [172] zu finden.

Beim Verfahren des kleinsten Abstandes werden iterativ drei aufeinanderfolgende Vertices betrachtet. Der erste und letzte Vertex des Tripels bilden eine Referenzgerade. Ist der senkrechte Abstand des mittleren Vertexes kleiner als ein gegebener Fehlerwert, wird der Vertex entfernt (siehe Abbildung 3.30). Ausgehend vom letzten Vertex der Dreiergruppe wird das Verfahren über die gesamte Kontur wiederholt. Da immer nur Dreiergruppen betrachtet werden, kann es passieren, dass noch Vertices mit einem sehr geringen Fehlerwert bei einem Durchgang übrig bleiben (siehe rote Punkte im rechten Bild in Abbildung 3.30). Solche Vertices können durch wiederholtes Anwenden des Verfahrens auf der Kontur entfernt werden. Insgesamt können mit diesem Verfahren maximal 50% der Vertices eines Polygonzuges in einem Durchgang gelöscht werden.

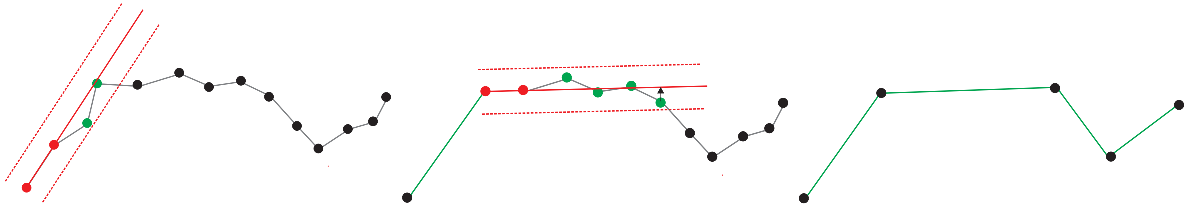


Abbildung 3.31: Polygonoptimierung nach Reumann-Witkam. Jeweils zwei Vertices definieren eine Referenzlinie, um die es einen "Fehlerkorridor" (rot gestrichelt) gibt. Alle Vertices innerhalb dieses Bereiches werden entfernt, der letzte auf die Line projiziert und das Verfahren wird ausgehend von dieser Position wiederholt (Mitte). Das rechte Bild zeigt das komplette mit Hilfe dieses Verfahrens optimierte Polygon.

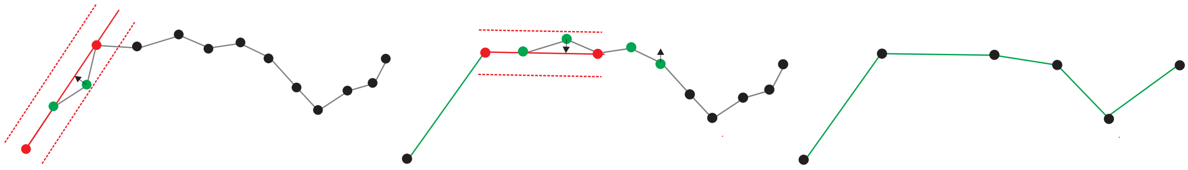


Abbildung 3.32: Polygonoptimierung nach Lang mit einer Suchlängenbegrenzung von $n = 4$ Vertices. Durch diese Begrenzung ist die Anzahl der Vertices, die pro Durchlauf entfernt werden kann, beschränkt. Es können unter Umständen Vertices mit einem kleinen Abstandsfehler zu einer Polygonlinie in der Kontur verbleiben (siehe roter Vertex im rechten Bild).

Beim Reumann-Witkam-Verfahren bilden zunächst die ersten beiden Vertices eine Refenrenzlinie. Parallel zu dieser Linie wird ein Toleranzkorridor mit einer gegebenen Breite definiert. Alle Vertices, die innerhalb dieses Toleranzbereiches liegen, werden als Teil der Referenzlinie angesehen und entfernt. Sobald ein Vertex gefunden wurde, der außerhalb der Toleranz liegt, wird dieser auf die Referenzlinie projiziert und zwischen diesem neuen Vertex und dem nächsten ein neuer Toleranzkorridor eröffnet (siehe Abbildung 3.31). Das Opheim-Verfahren basiert auf einem ähnlichen Ansatz, verwendet aber neben dem maximalen Abstand einen zweiten Parameter, der einen Radius um den ersten Referenzvertex festlegt. Zur Definition des Toleranzkorridors wird der entfernteste Punkt in der durch den Radius definierten Kugel verwendet. Befindet sich in diesem Gebiet kein Punkt, wird wie bei Reumann-Witkam der nächste Vertex der Kontur zur Definition der Referenzlinie verwendet.

Beim Lang-Verfahren wird zunächst eine feste Segmentlänge n vorgegeben. Der erste und letzte Vertex des Segmentes bilden die Referenzlinie. Ist ein Vertex dieses Segmentes weiter als ein Fehlerwert entfernt, wird es um den letzten Punkt gekürzt. Es wird so lange weiter gekürzt, bis alle Punkte unterhalb der Toleranz liegen oder nur noch zwei Punkte übrig geblieben sind. Nachdem ein Intervall gefunden wurde, das keine Punkte außerhalb des Tolenranzbereiches enthält, wird ausgehend vom letzten Punkt des Segmentes eine neue Optimierung gestartet. Problematisch bei diesem Ansatz ist die Tatsache, dass der Suchbereich zum Zusammenfassen von Vertices durch einen festen Wert begrenzt ist. Es können immer nur maximal so viele Vertices zusammengefasst werden, wie durch die maximale Segmentlänge vorgegeben ist. Es kann also passieren, dass im Ergebnis immer noch unnötige Unterteilungen vorhanden sind (siehe Abbildung 3.32). Die maximal erreichbare Kompression beträgt also $1/n$.

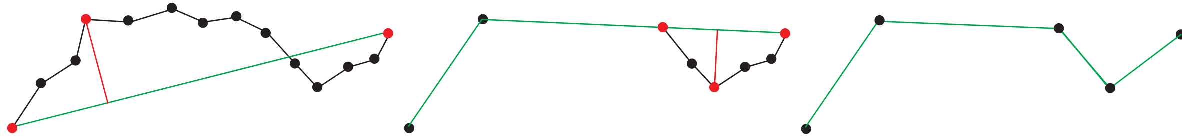


Abbildung 3.33: Rekursive Polygonoptimierung nach Douglas-Peucker. Der jeweils am weitesten von der Referenzlinie entfernte Punkt bildet einen neuen Vertex in der optimierten Kontur. Alle Vertices, die weniger als einen Maximalabstand von der Refrenzlinie entfernt sind, werden gelöscht. Anschließend steigt die Rekursion auf den neuen Teilstrecken ab, bis alle Vertices innerhalb der Fehlertoleranz sind.

Der Douglas-Peucker-Algorithmus berechnet die optimierte Kontur rekursiv. Es werden zunächst der erste und letzte Punkt der Kontur verbunden. Dann wird der am weitesten von dieser Linie entfernte Punkt bestimmt, der außerhalb einer vorher festgelegten Fehlertoleranz ist. Dieser Punkt teilt die Linie in zwei Segmente. Für jedes neu entstandene Segment wird der Prozess wiederholt, bis keine Vertices außerhalb des Toleranzwertes mehr gefunden werden (siehe Abbildung 3.33). Die Laufzeit des Verfahrens beträgt im Worst-Case $O(n \cdot m)$, wobei m die Länge des optimierten Linienzuges ist. Die mittlere Laufzeit beträgt $O(n \cdot log\ n)$.

Im Praxiseinsatz haben alle Algorithmen sehr kurze Laufzeiten (siehe Abschnitt 4.3). Das Region Growing inklusive Konturpolygonerstellung benötigt nur einen kleinen Teil der Gesamtlaufzeit. Das Douglas-Peucker-Verfahren ist insofern für die Optimierung das interessanteste der vorgestellten Verfahren, da es den maximalen Vertexfehler als einzigen Eingabeparameter hat und trotzdem alle redundanten Vertices innerhalb des Toleranzbereiches in nur einem Durchgang entfernt. Der für die Optimierung akzeptable Komprimierungsfehler lässt sich auf diese Weise sehr leicht vorgeben, so dass dieses Verfahren, wenn in den gezeigten Ergebnissen nicht anders angegeben, standardmäßig verwendet wird.

Neutriangulierung

Die extrahierten Konturpolygone sind normalerweise nicht konvex. Für die praktische Anwendung (Speicherung in Austauschformaten, Rendering mit OpenGL oder DirectX) müssen diese in eine Dreiecksdarstellung überführt werden. D.h., die komplexen Konturen müssen neu trianguliert werden. Die Neutriangulierung der extrahierten planaren Region geschieht, wie schon in [195] beschrieben, mit Hilfe des OpenGL-Tesselators. Der Vollständigkeit halber werden die entsprechenden Passagen hier noch einmal wiederholt. Zunächst werden die Standardverfahren zur Tesselierung kurz dargestellt, anschließend wird skizziert, wie sich die in OpenGL vorhandenen Tesselierungsroutinen verwenden lasssen, um die Konturen zu triangulieren.

Verfahren zur Triangulation von Polygonen

Polygonzüge und Vertices in Polygonen können unterschiedliche Eigenschaften haben, anhand derer sie klassifiziert werden. Einfache Polygone sind Polygone, die keine Löcher oder sich schneidende Kanten haben. Ein einfaches Polygon wird als konvex bezeichnet, wenn es nur konvexe Vertices besitzt. Ein Vertex v_i eines Polygons wird als konvex bezeichnet, wenn bei einer Traversierung des Polygons entgegen dem Uhrzeigersinn beim Übergang von v_{i-1} nach v_{i+1} an der

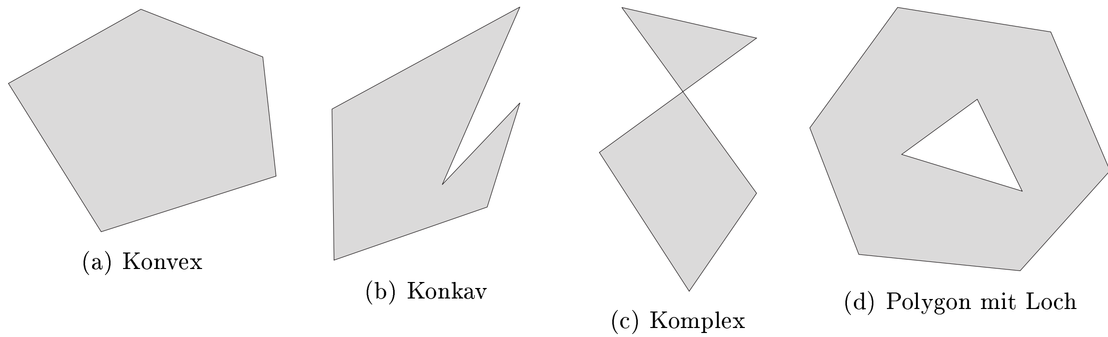


Abbildung 3.34: Verschiedene Klassen von Polygonen

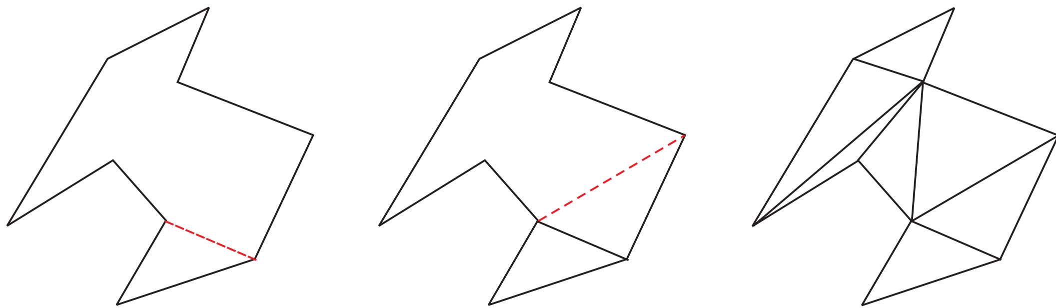


Abbildung 3.35: Ear-Cutting. Das Polygon wird durch wiederholtes Abtrennen von Ohren trianguliert.

Stelle v_i ein Knick nach links entsteht. Entsteht ein Knick in die andere Richtung, wird der Vertex als konkav bezeichnet. Einfache Polygone, die mindestens einen konkaven Vertex haben, nennt man konkav, die anderen konvex (siehe Abbildung 3.34).

Komplexe Polygone sind Polygone, die eine oder mehrere sich schneidende Kanten oder Löcher haben. Ein Loch ist ein Polygon, das vollständig in einem anderen liegt. Löcher werden manchmal auch als Inseln bezeichnet. Eine Diagonale im Polygon ist eine Verbindungslinie zweier Vertices v_i und v_j, die vollständig im Polygon liegt und keine Kanten schneidet. Ein monotones Polygon ist ein Polygon, dessen Kontur aus zwei monotonen Ketten besteht. Eine Folge von Vertices heißt x- bzw. y-monoton, wenn die entsprechenden Koordinaten der Punkte in der Kette aufsteigend sind.

Die Triangulation von konvexen Polygonen ohne Löcher ist trivial: Man wählt einen beliebigen Vertex aus und zieht Diagonalen zu allen anderen Punkten. Die Komplexität dieses Verfahrens beträgt $\mathcal{O}(n)$ für ein Polygon mit n Ecken. Die Frage, ob ein Polygon konvex ist, kann mit linearem Zeitaufwand entschieden werden. Dazu wird für jeden Vertex v_i überprüft, ob er konvex ist, d.h., ob er sich links von der durch v_{i-1} und v_{i-2} definierten Geraden befindet.

Einfache Polygone lassen sich immer triangulieren, wie Meisters 1975 gezeigt hat [63]. Er bewies, dass jedes Polygon mit mehr als drei Vertices mindestens zwei sich nicht überlappende Ohren besitzt (*Two Ears Theorem*). Ein Vertex wird als Ohr eines Polygons bezeichnet, falls die Verbindungsstrecke der Nachbarn v_{i+1} und v_{i-1} eines Polygonvertex' v_i komplett im Polygon

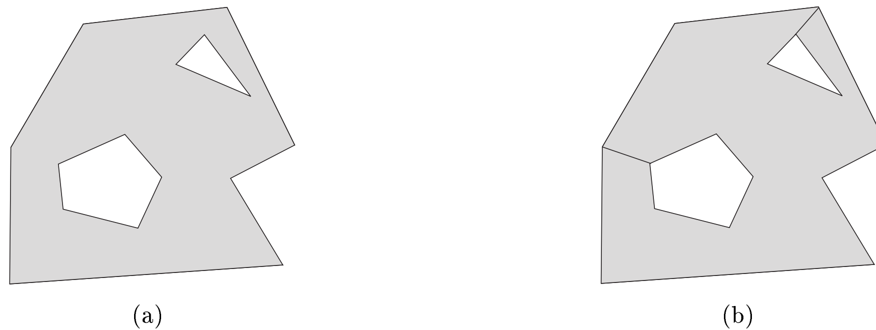


Abbildung 3.36: Einbinden von Löchern in Polygonzüge. In Bild (a) ist ein einfaches Polygon mit Löchern gezeigt. Durch Verbinden der Vertices, die den kürzesten Abstand zur umschließenden Kontur haben, werden die Konturen der Löcher in den Polygonzug mit eingebunden und ein einfaches Polygon entsteht (b).

liegt. Das aus diesen drei Vertices gebildetete Dreieck wird ebenfalls als Ohr bezeichnet. Jedes einfache Polygon lässt sich triangulieren, indem sukzessive alle Ohren abgetrennt werden (siehe Abbildung 3.35). Dabei entstehen immer mindestens $n-2$ Dreiecke. Dieses Verfahren wird *Ear Cutting* genannt. Das Hauptproblem bei dieser Vorgehensweise ist das Auffinden der Ohren.

Ein einfacher Algorithmus zum Auffinden von Ohren mit quadratischer Laufzeit in der Anzahl der Vertices wurde von Kong vorgestellt [210]. Die Laufzeit des Algorithmus beträgt $\mathcal{O}(kn)$, wobei k die Anzahl der konkaven Vertices im Polygon ist. Im Worst-Case beträgt die Laufzeit $\mathcal{O}(n^2)$, da insgesamt $\mathcal{O}(n)$ konkave Vertices auftreten können. Es gibt allerdings auch Algorithmen, die dieses Problem in linearer Laufzeit lösen können. Ein bekanntes stammt von El Gindy [77]. Dieser zerteilt ein Polygon rekursiv in „Gute Unterpolygone” (*Good Sub Polygons, GSP*), bis ein Ohr gefunden wurde. Ein GSP ist ein Unterpolygon, das mit dem Rest des ursprünglichen Polygons nur eine gemeinsame Kante hat (die sog. *Cutting Edge*). Da jedes Unterpolygon höchstens $n-2$ Vertices haben kann, ergibt sich eine lineare Laufzeit.

Ear-Cutting Algorithmen haben den Vorteil, dass sie leicht verständlich und einfach zu implementieren sind. Allerdings haben sie hohe Worst-Case-Laufzeiten. Da jedes Polygon in $n-2$ Dreiecke unterteilt werden kann, muss der Auffinde-Algorithmus beim Ear-Cutting $(n-2)$-mal aufgerufen werden. Es entstehen demnach Worst-Case Laufzeiten von $\mathcal{O}(n^3)$ bei Verwendung des quadratischen Auffindeverfahrens, bzw. $\mathcal{O}(n^2)$ mit der linearen Variante. Die Laufzeiten werden allerdings in beiden Fällen von der Anzahl der konkaven Vertices im Polygon bestimmt. Da in der Praxis mehr konvexe als konkave Vertices auftreten, liegen die tätsachlichen Laufzeiten in der Regel unter dem Worst-Case.

Problematisch ist die Einbeziehung von Löchern in die Triangulierung. Befinden sich Löcher in den Polygonen, besteht eine Lösung darin, die Konturen zunächst zu verbinden, indem die kürzeste Verbindung zwischen der äußeren Kontur und dem Loch als neue Kante eingefügt wird [123]. Diese Strecke wird doppelt eingefügt, so dass sie beim Traversieren des Polygons einmal als „Hinweg” und einmal als „Rückweg” vorhanden ist. Auf diese Art und Weise entsteht wieder eine einzelne geschlossene Kontur (Abbildung 3.36).

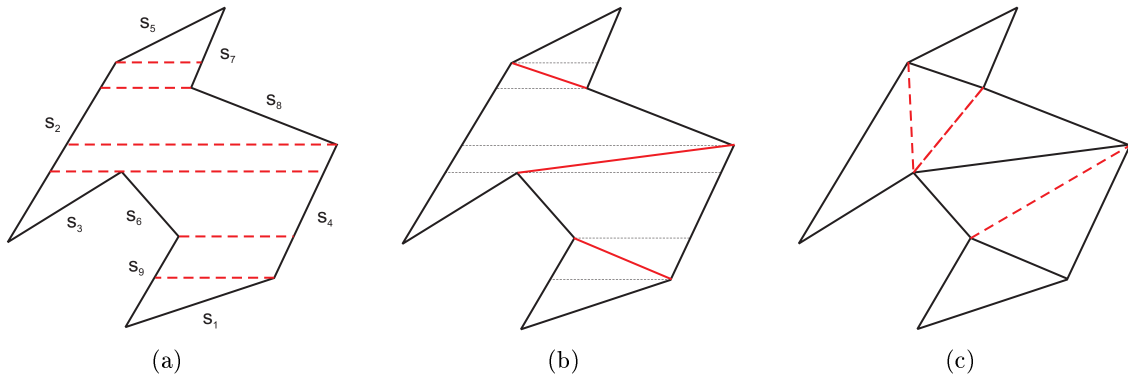


Abbildung 3.37: Triangulation nach Seidel. Zunächst werden die Kanten des Polygons zufällig durchnummeriert (s_1 bis s_9). Danach wird durch Ziehen von horizontalen Linien eine Zerlegung in Trapezoide hergestellt (a). Anschließend werden duch Einfügen von Diagonalen monotone Polygone erzeugt (b), die im nächsten Schritt trianguliert werden (c).

Ein ebenfalls relativ einfach zu implementierender Triangulationsalgorithmus mit einer Worst-Case-Laufzeit von $\mathcal{O}(n \cdot log(n))$ wurde 1995 von Narkhede und Manocha [133] vorgestellt und basiert auf dem mit gleicher Laufzeit arbeitenden Algorithmus von Seidel [171]. Narkhedes und Manochas Algorithmus besteht aus drei Schritten: Zunächst werden die Polygone in Trapezoide zerlegt. Anschließend werden die Trapezoide in monotone Polygone zerlegt, die danach durch Entfernen konvexer Vertices trianguliert werden.

Um das Polygon in Trapezoide zu zerlegen, werden die Kanten des Polygons zufällig durchnummeriert. Ausgehend von den Endpunkten der Segmente wird eine horizontale Line nach links und rechts gezogen, bis sie auf ein weiteres Segment trifft. Die Strecken zwischen Endpunkt und Schnittpunkt, die innerhalb des Polygons liegen, bilden die horizontalen Begrenzungslinien der Trapezoide (s. Abbildung 3.37(a)). Diese Zerlegung geschieht in $\mathcal{O}(n \cdot log(n))$ Zeit. Die monotonen Polygone werden aus der Trapezzerlegung gewonnen, indem, wann immer möglich, Diagonalen zwischen zwei Vertices der ursprünglichen Kanten gezogen werden, die sich auf verschiedenen Seiten des Trapezoiden befinden (s. Abbildung 3.37b). Die dafür benötigte Zeit beträgt $\mathcal{O}(n)$.

Monotone Polygone können mit einem Scanline-Algorithmus in $\mathcal{O}(n)$ trianguliert werden. Dieser benutzt einen Stack, um die noch zu verarbeitenden Vertices zu verwalten. Die Funktionsweise soll hier für y-monotone Polygone gezeigt werden (nach [146]). Für x-monotone Polygone muss die Formulierung angepasst werden. Zunächst wird der Stack mit den oberen beiden Vertices initialisiert. Dann werden die Vertices in absteigender Reihenfolge durchlaufen. Liegen der oberste Vertex des Stacks und der aktuelle Vertex auf derselben Seite des Polygons, werden so oft wie möglich Diagonalen eingefügt und die entsprechenden Vertices vom Stack entfernt. Anschließend wird der aktuelle Vertex abgelegt. Liegt der oberste Vertex auf der gegenüberliegenden Seite des aktuellen Vertexes, werden Diagonalen zu allen Vertices auf dem Stack gezogen und die Elemente gelöscht. Das oberste Element wird allerdings gespeichert und nach dem Ziehen der Diagonalen zusammen mit dem aktuellen Vertex neu abgelegt. Diese Schritte werden fortgesetzt, bis der letzte Vertex erreicht ist. Abbildung 3.38 verdeutlicht den Ablauf. Jeder Vertex, der nicht

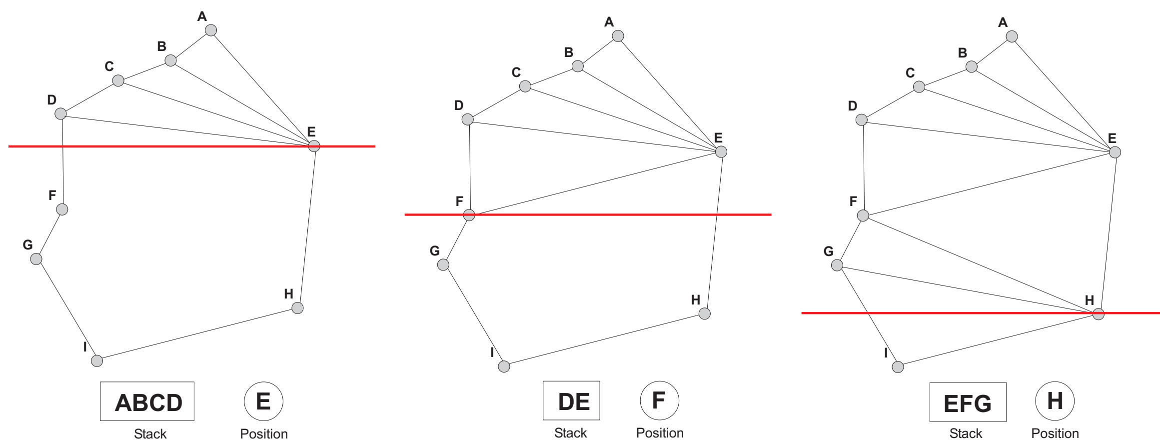


Abbildung 3.38: Triangulation monotoner Polygone. Hier sind die Situationen gezeigt, in denen der zuvor aufgebaute Stack abgearbeitet wird.

mehr verwendet wird, wird sofort aus dem Stack gelöscht. Daraus ergibt sich eine Laufzeit der Komplexitätsklasse $\mathcal{O}(n)$.

Die meiste Zeit nimmt das Erstellen der Trapezdekomposition in Anspruch, so dass sich eine fast lineare Laufzeit von $\mathcal{O}(n \cdot log(n))$ ergibt. Da dieser Algorithmus schnell arbeitet und leicht zu implementieren ist, findet er in der Praxis häufig Verwendung. Den bisher einzigen deterministischen Algorithmus zur Triangulation in linearer Zeit hat Chazelle 1990 vorgestellt [32]. Er ist allerdings dermaßen kompliziert, dass eine Implementation praktisch unmöglich ist. Ein einfacherer probabilistischer Algorithmus, der eine erwartete Laufzeit von $\mathcal{O}(n)$ hat, wurde 2001 von Amato et. al [130] veröffentlicht. Frei verfügbare Implementationen von solchen Algorithmen gibt es meines Wissens bisher nicht, so dass auf die genauen Funktionsweisen hier nicht weiter eingegangen werden soll.

Tesselierung mit OpenGL

Ein effektives, frei verfügbares Werkzeug zum Triangulieren von Polygonen bietet der Tesselator der OpenGL Utility Library. Diese Bibliothek benötigt keinen Renderkontext, so dass sie unabhängig vom verwendeten Grafiktreiber ist. Sie lässt sich daher im Gegensatz zu den anderen OpenGL-Modulen auch auf Systemen ohne grafische Benutzeroberfläche verwenden und zusammen mit anderen APIs, z.B. DirectX, verwenden. Des Weiteren ist der OpenGL-Tesselator auch in der Lage, komplexe Polygone zu triangulieren. Sich schneidende Kanten oder Löcher bereiten keine Probleme.

Damit die Triangulation die gewünschten Ergebnisse liefert, muss bei der Initialisierung des Tesselators festgelegt werden, wie Löcher in Polygonen behandelt werden sollen. Dies geschieht durch Festlegung einer so genannten *Windungsregel* (engl. *Winding Rule*), die Gebiete innerhalb eines Polygons anhand von Windungszahlen (*Winding Numbers*) klassifiziert.

Die Windungszahl legt für jeden Punkt im Polygon fest, wie oft er umrundet wird, wenn man einer Kontur des Polygons folgt. Durchläuft man die Kontur gegen den Uhrzeigersinn, ist die

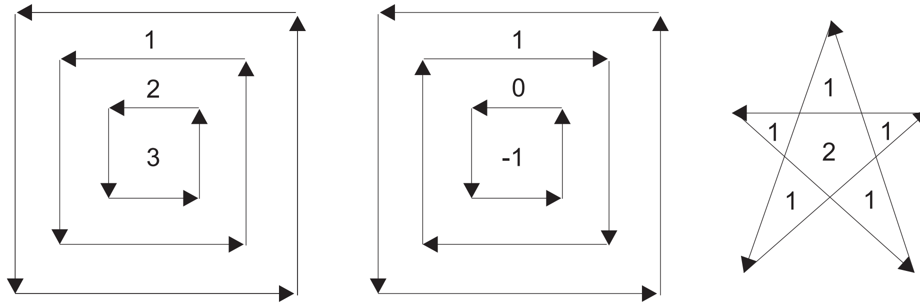
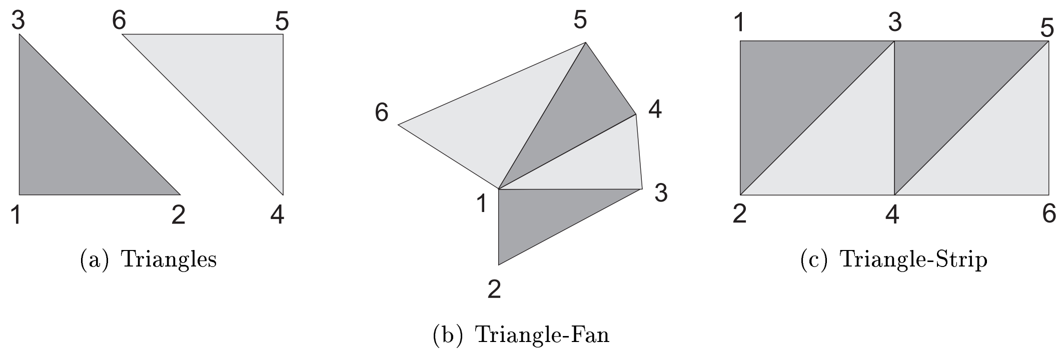


Abbildung 3.39: Windungszahlen für verschiedene Polygone (nach [39]).



(a) Triangles

(b) Triangle-Fan

(c) Triangle-Strip

Abbildung 3.40: Typen von Dreiecks-Listen in OpenGL

Windungszahl positiv, ansonsten negativ. Besitzt ein Polygon mehrere Konturen, werden in innen liegenden Gebieten die Windungszahlen aller umgebenden Konturen addiert. Beispiele für die Zuweisung von Windungszahlen zeigt Abbildung 3.39. Mit Hilfe dieser Zahlen kann anhand einer Windungsregel entschieden werden, welche Gebiete innerhalb eines Polygonzuges durch Dreiecke approximiert werden sollen.

In den Löchern der Konturpolygone des Polygonnetzes können keine weiteren Konturen mehr liegen, da bei der Flächenextraktion nur zusammenhängende Gebiete vereinigt wurden. Es können also nur die Windungszahlen eins oder zwei auftreten, wobei die durch eins gekennzeichneten Gebiete diejenigen sind, die trianguliert werden müssen. Regionen mit der Windungszahl zwei liegen innerhalb von Löchern und müssen nicht beachtet werden. Diese Regel ist in der LVR-Implementierung entsprechend vorgegeben. Nach der vollständigen Definition eines Polygons wird die Triangulation gestartet. Im Verlauf der Verarbeitung werden vom Tesselator verschiedene Callback-Methoden aufgerufen. Mit Hilfe dieser Methoden können u.a die erzeugten Dreiecke gespeichert und auftretende Fehler erkannt werden.

Der Tesselator liefert die Dreiecke im Verlauf der Triangulation in verschiedenen Datenstrukturen, wie sie auch beim Rendern mit OpenGL verwendet werden. Möglich sind einzelne Dreiecke (Triangles), Triangle-Fans und Triangle-Strips. Bei einer Triangle-List bilden jeweils drei aufeinander folgende Vertices ein Dreieck. Ein Triangle-Fan ist eine zusammenhängende Fläche, in der alle Dreiecke einen gemeinsamen Vertex haben. Sobald mehr als zwei Vertices vorhanden sind,

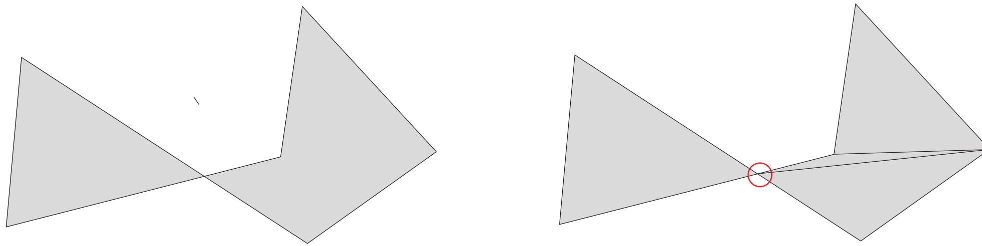


Abbildung 3.41: Triangulierung eines konplexen Polygons mit sich schneidenden Kanten. An der rot markierten Stelle muss während der Triangulation ein neuer Vertex eingefügt werden.

bilden die Vertices 1, i und $i-1$ ein Dreieck, wobei i die Nummer des aktuellen Vertexes ist. Ein Triangle-Strip ist ebenfalls eine zusammenhängende Fläche. Dort haben aufeinanderfolgende Dreiecke eine gemeinsame Kante. Bei gerader Vertexnummer und $i > 3$ bilden die Vertices i, $i-1$ und $i-2$ ein Dreieck. Andernfalls wird das Dreieck durch i, $i-2$ und $i-1$ definiert. Durch diese Art der Nummerierung wird gewährleistet, dass die Dreiecke immer entgegen dem Uhrzeigersinn definiert werden. Bei Triangle-Fan und Triangle-Strip wird mit jedem neuen Vertex auch ein neues Dreieck definert. Abbildung 3.40 zeigt die verschiedenen vom Tesselator erzeugten Datenstrukturen.

Während der Tesselierung werden bevorzugt Triangle-Fans [175] generiert. Das liegt vermutlich daran, dass der Tesselator versucht, konvexe Unterpolygone zu erzeugen, die dann durch Ziehen von Diagonalen trianguliert werden. Dabei entstehen automatisch Triangle-Fans. Zudem können Triangle-Fans extrem schnell gerendert werden. Gelegentlich kommt es vor, dass der Tesselator einen neuen Vertex generiert. Das geschieht z.B. dann, wenn sich zwei Kanten eines Polygons schneiden (s. Abbildung 3.41). Die Koordinaten des neuen Vertexes werden vom Tesselator berechnet.

Problematisch ist die Integration dieser neu erzeugten Vertices in die globale Datenstruktur zur Meshrepräsentation. Damit keine Vertices doppelt abgelegt werden, werden alle vom Tesselator generierten Vertexdefinitionen in einer Map-Datenstruktur abgelegt, mit den Vertexkoordinaten als Schlüssel zum dazugehörigen Indexwert. Mit Hilfe einer Suchanfrage an die Map kann in logarithmischer Zeit festgestellt werden, ob der Vertex schon existiert. Ist dies der Fall, wird der hinterlegte Indexwert für Referenzierung benutzt. Ansonsten wird der Vertex mit einem neuen Schlüssel angefügt. Das Einfügen der Dreiecke und Vertices in die hinterlegten Datenstrukturen erfolgt wie bei der Erzeugung des ursprünglichen Meshes durch eine Implementierung des `MeshBuffer`-Interfaces.

Abbildung 3.42 zeigt des Ergebnis einer Neutriangulation der ebenen Anteile eines Meshes nach Optimierung und Polygonextraktion. Man kann deutlich erkennen, dass die Konturen der Ebenen entlang der geraden Kanten zu einer kompakten polygonalen Darstellung zusammengefasst wurden. Durch die Neutriangulierung der Polygone wurde die Anzahl der Dreiecke im Mesh deutlich reduziert (von 371.460 auf 98.648), ohne dass die ursprügliche rekonstruierte Geometrie geändert wurde.

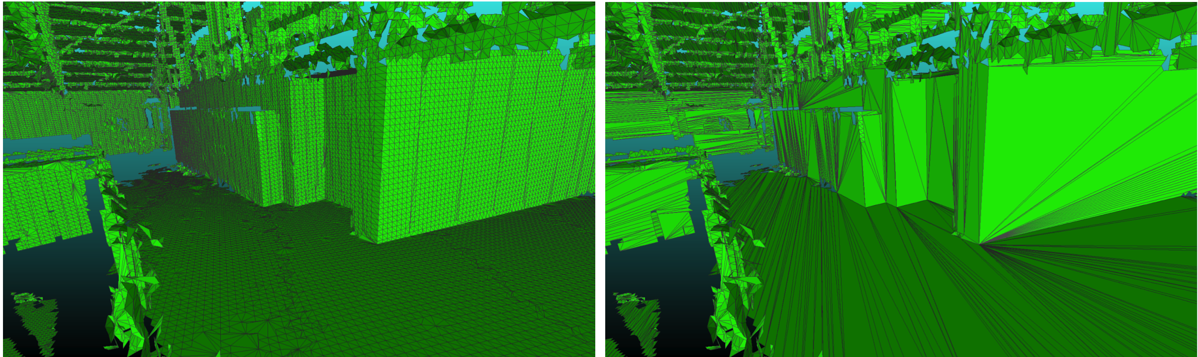


Abbildung 3.42: Neutriangulation der ebenen Konturen eines Meshes mit Hilfe des OpenGL Tesselators nach erfolgter Konturextraktion. Das linke Bild zeigt das Mesh nach der Ebenenoptimierung und dem Schließen der Löcher. Das rechte Bild zeigt dieselbe Szene nach der Extraktion der Konturpolygone und Neutriagulation. Die Anzahl der Dreiecke wurde von 371.460 auf 98.648 reduziert.

Clusterlabeling

Geht man davon aus, dass der zur Aufnahme der rekonstruierten Szene verwendete Sensor waagerecht ausgerichtet war, lassen sich die extrahierten Cluster durch Analyse der Normalenrichtungen in verschiedene Klassen einteilen. Die Identifizierung von waagerechten Flächen ist trivial, deren Normalen zeigen immer nach oben oder unten, so dass sie durch Vergleich des Winkels zwischen Flächennormale und einer Referenznormalen mit der Richtung $(0, 1, 0)$ erkannt werden können. Genau senkrechte Normalen haben unabhängig von ihrer Orientierung bei der Projektion in die Bodenebene eine Länge von 1. Mit Hilfe dieser beiden Kriterien und mit passend gewählten Toleranzwerten lassen sich diese Klassen von Ebenen sehr leicht unterscheiden. Bezieht man die Schwerpunkte der Ebenen mit ein, insbesondere ihre Höhe über der niedrigsten Fläche, lassen sich auch Fußböden und Deckenflächen extrahieren. Die niedrigste zusammenhängende Fläche definiert den Fußboden, die höchste die Decke. Nüchter et al. [144] nutzen weitere semantische Relationen zwischen den Ebenen, um z.B. auch Türen zu erkennen. Ein weiterführender Ansatz zur Erkennung von Möbeln in Laserscans wird in Kapitel 5.3 noch ausführlich dargestellt.

Zwei Beispiele für diese Art von Labeling zeigt Abbildung 3.43. Die oberen Bilder zeigen die Rekonstruktion einer Rescue-Arena mit sehr eng gewählten Schwellwerten für Klassifizierung von ebenen Flächen. Erweitert man die Toleranzwerte und verzichtet auf den Vertexoptimierungsschritt bei Clustering, lässt sich auch in unebenen Szenen eine gute Segmentierung erreichen, wie in den unteren Bildern zu sehen ist. Diese Beispiele wurden aus einem registrierten Scan der Kvarntorp-Mine in Schweden [116] generiert. Interessant für mobile Roboter sind in beiden Fällen die rot eingefärbten Flächen, da diese potenziell befahrbare Gebiete markieren.

Fazit Ebenensegmentierung

Mit Hilfe des Region-Growing-Ansatzes lassen sich auf einfache Art und Weise zusammenhängende ebene Regionen in den Meshes segmentieren. Im Vergleich zum Clustern der Dreiecke ist

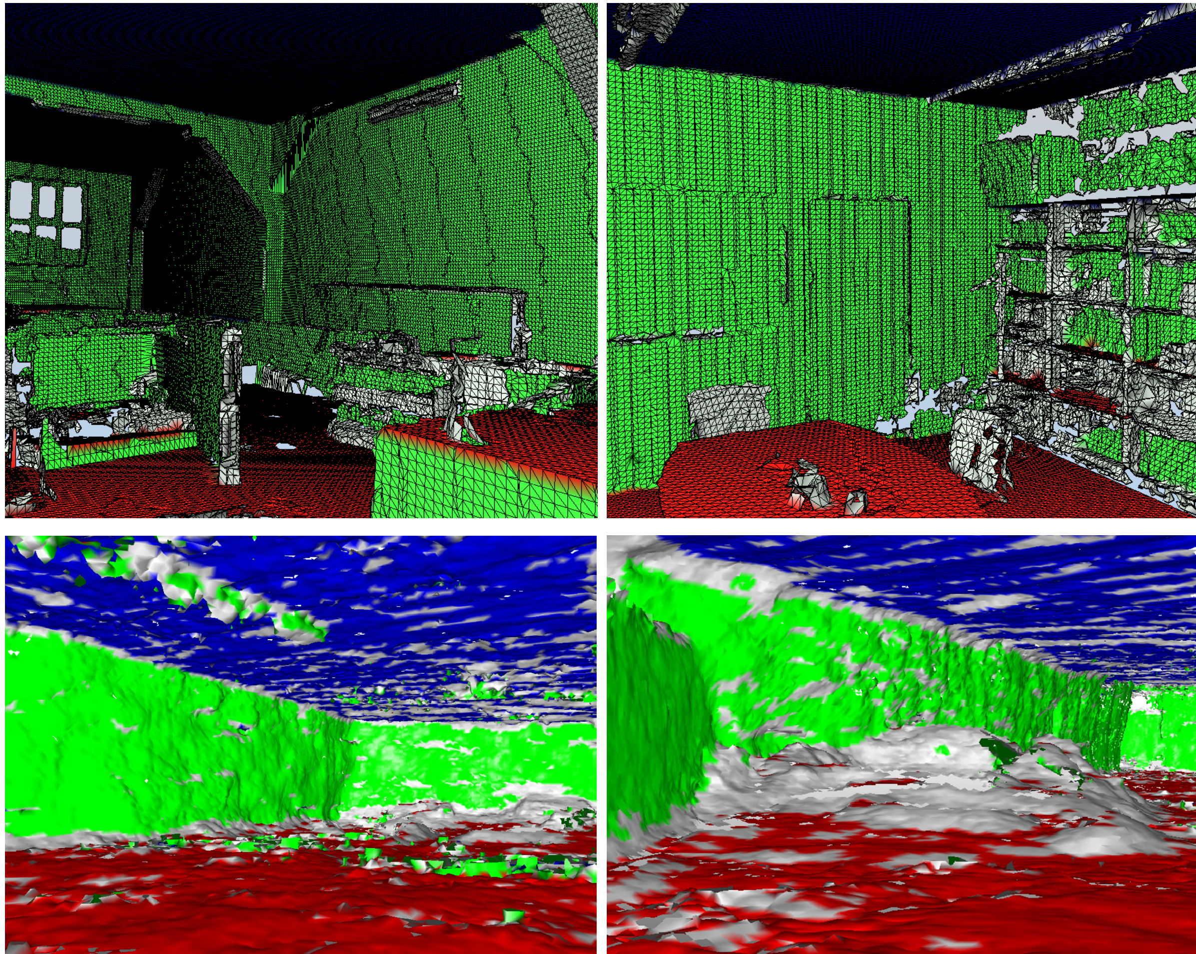
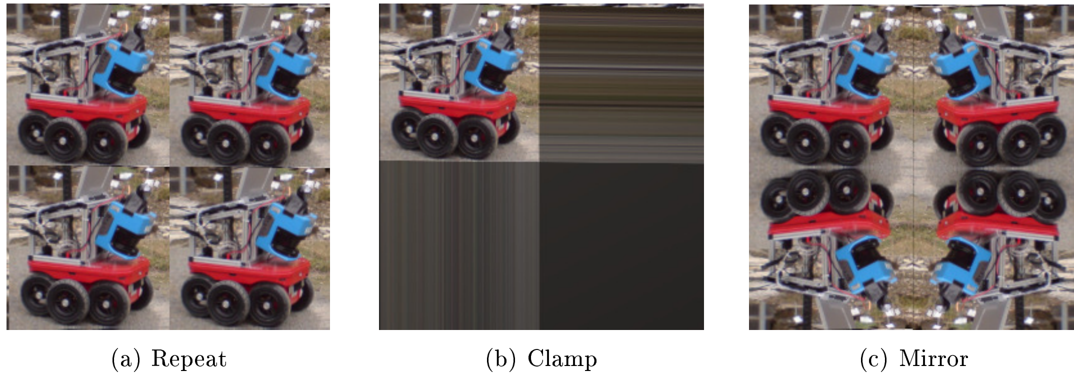


Abbildung 3.43: Klassifizierung der im Mesh gefundenen Ebenen nach Normalenrichtung (Fußboden rot, Decke blau, Wand grün, unklassifiziert grau). Die oberen Bilder zeigen zwei Gebäudescans, die unteren Bilder zwei Scans aus einer Mine [116]. Dort wurden die Ebenen mit einem größeren Schwellenwert des Normalenkriteriums beim Region Growing extrahiert, um der Unebenheit der Szene gerecht zu werden.

die algorithmische Aufbereitung der Konturen zu einer optimalen Darstellung aufwändig. Ähnliches gilt für die Neutriangulation der Polygone. Die zur Lösung dieser Probleme verwendeten Verfahren stammen aus einem lange erforschten Bereich der Computergrafik. Die Entwicklung eigener Verfahren hätte den Rahmen dieser Arbeit mit Sicherheit gesprengt. Mit Hilfe frei verfügbarer Bibliotheken ist es aber möglich, die benötigten Funktionalitäten effizient in die Verarbeitungspipeline der Software zu integrieren, so dass die Zielsetzung, eine Datenkompression durch Ausnutzung der Planarität in den Rekonstruktionen zu erreichen, realisiert werden konnte.



(a) Repeat (b) Clamp (c) Mirror

Abbildung 3.44: Randwiederholungsfunktionen beim Texture-Mapping.

3.6 Automatische Erzeugung von Texturen

Beim Texture-Mapping wird die Fläche von Polygonen mit Bilddaten assoziiert, so genannten Texturen. Beim Rendern der Szene werden diese Bilder gemäß der Position der virtuellen Kamera perspektivisch verzerrt auf die Polygone projiziert. Mit Hilfe von Texturen lassen sich Details auf die Meshes übertragen, die in der reinen 3D-Geometrie nicht zu erkennen sind. Ein bekanntes Beispiel für den Einsatz von Texturen ist die Verwendung eines relativ kleinen randlosen Musters, z.B. einer Ziegelsteinreihe, das wiederholt nebeneinander gerendert wird, so dass der Eindruck einer größeren Mauer entsteht. Der Vorteil dieses Verfahrens liegt auf der Hand: mit Hilfe eines relativ kleinen Bitmap-Musters, überlichweise 32×32 oder 64×64 Pixel, lässt sich ein sehr realitätsnaher Eindruck dieser Szene erzielen, ohne dass jeder Stein in der Mauer individuell modelliert werden muss.

Als Texturen müssen nicht zwangläufig sich wiederholende Muster eingesetzt werden. Es können Bilder auch großflächig auf Polygone übertragen werden. Dieser Ansatz ist für robotiknahe Anwendungen sehr interessant, da sich so Bilddaten mit geometrischen Informationen verknüpfen lassen. Viele 3D-Sensoren liefern neben den Tiefeninformationen weitere Informationen über die Beschaffenheit der erfassten Oberflächen. Dies können Farb- oder Remissionswerte, aber auch andere Attribute, wie z.B. Thermokameradaten sein [25]. Zur Aufwertung der Polygonkarten sollen diese, falls vorhanden, in die Rekonstruktionen als Texturen integriert werden. Im Folgenden sollen kurz die technischen Grundlagen zum Texture-Mapping dargestellt werden (nach [195]), bevor anschließend das Verfahren zur Texturgenerierung aus den Laser-Farbdaten vorgestellt wird.

3.6.1 Texture-Mapping

Beim Texture-Mapping erhält jeder Vertex eines Polygons zwei weitere Koordinaten (u, v), die seine relative Position in der verwendeten Textur definieren. Diese Texturkoordinaten sind überlicherweise normiert, d.h. die obere rechte Ecke eines Bildes hat die Texturkoordinaten $(0.0, 0.0)$, die rechte untere $(1.0, 0.0)$. Es ist möglich, den Vertices auch Texturkoordinaten außerhalb des

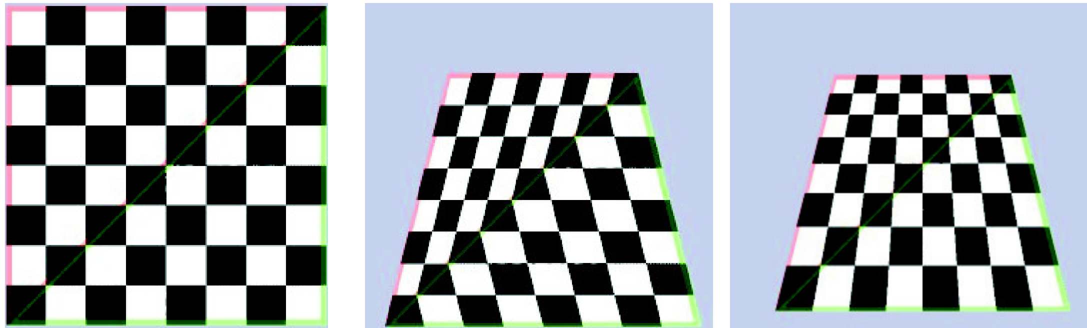


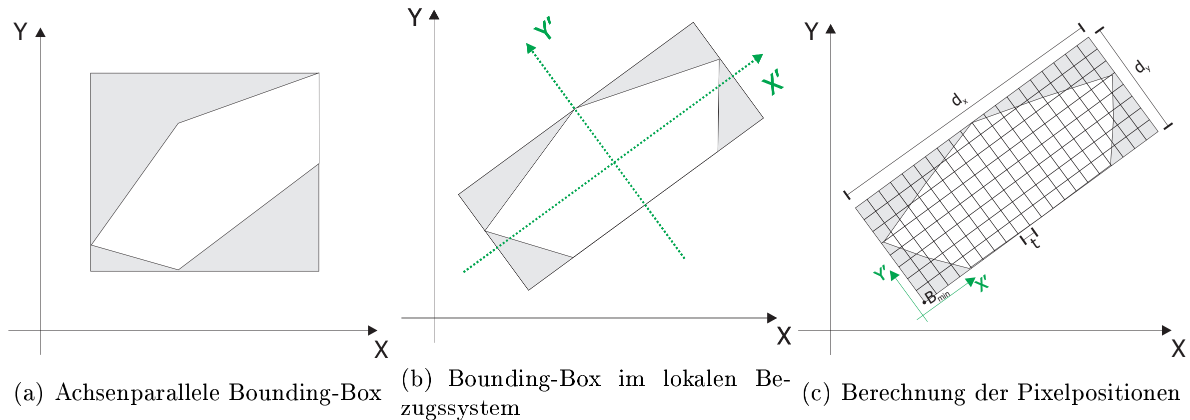
Abbildung 3.45: Perspektivenverzerrungen beim Texture-Mapping. Links die verwendete Textur, die auf zwei Dreiecke gemappt wird. Das Bild in der Mitte zeigt das Ergebnis der Interpolation ohne Perspektivenkorrektur. Rechts sieht man das Ergebnis unter Berücksichtigung der z-Komponente. (entnommen aus [204])

Intervalls $[0, 1]$ zuzuweisen. Wie diese Koordinaten behandelt werden, wird über die sogenannte Randwiederholungsfunktion festgelegt. Bekannte Verfahren sind die Wiederholung der Textur oder Spiegelung. Die im Einzelnen bereit gestellten Funktionalitäten hängen von der verwendeten Render-Engine ab. Gängige Funktionen sind in Abbildung 3.44 dargestellt. Beim Rendern der Polygone werden im einfachsten Fall zunächst die Koordinaten linear entlang der Begrenzungskanten interpoliert. Zwischen den so gewonnenen Werten wird zeilen- oder spaltenweise entlang des Bildschirms interpoliert und die Farbe des korrespondierenden Pixels in der Textur übernommen. Bei diesem Verfahren können aber perspektivische Verzerrungen auftreten, da die Interpolation erst durchgeführt wird, nachdem die Polygone in den zweidimensionalen Texturraum projiziert wurden (s. Abbildung 3.45). Daher muss auch die z-Komponente der Polygonvertices berücksichtigt werden, um eine perspektivisch korrekte Wiedergabe zu erreichen [174]:

$$\begin{aligned} u &= (u/z) \\ v &= (v/z) \end{aligned}$$

Beim Rendern der Polygone in einem Mesh werden die Farbwerte aus den Texturen gemäß den definierten Texturkoordinaten an den Vertices übernommen. Da bei der Interpolation der Texturkoordinaten auch Pixelpositionen in den Texturen bestimmt werden können, die zwischen zwei Pixeln in der Textur liegen, müssen die zu verwendenden Farbwerte mit geeigneten Verfahren interpoliert werden. Die einfachste Variante ist, die Farbe des nächst gelegenen Pixels zu übernehmen. Da Farbwerte eines Pixels beim Rendern der Polygone mehrfach übernommen werden können, wirken die Texturen auf den Objekten sehr „pixelig". Um diesen Effekt zu vermindern, werden die Farbwerte daher in der Regel mit einem geeigneten Filter interpoliert. Eine weit verbreite Methode ist die bilineare Filterung.

Neben Farbinformationen können mit Hilfe von Texturen auch andere Informationen auf Polygone übertragen werden. So werden beim Bump-Mapping beim Rendern vorher festgelegte Normalenwerte zugewiesen, um Beleuchtungseffekte zu erzielen. Beim Displacement-Mapping werden mit Hilfe von in Texturen gespeicherten Informationen die Vertexpositionen beeinflusst.



(a) Achsenparallele Bounding-Box (b) Bounding-Box im lokalen Bezugssystem (c) Berechnung der Pixelpositionen

Abbildung 3.46: Berechnung optimaler Texturen für planare Polygone. Bild (a) zeigt die achsenparallele Bounding-Box, in (b) ist sie an den Hauptachsen des Polygons ausgerichtet. Die jeweils zur Texturierung genutzte Fläche ist weiß dargestellt. Die zur Berechnung der Pixelspositionen benötigten Größen sind in (c) abgebildet.

3.6.2 Automatische Generierung von Texturen aus Laserdaten

Texturen für die Polygone im Mesh werden erzeugt, indem die Bounding-Box einer ebenen Region mit einer vorgegebenen Rastergröße ("Texelsize") diskretisiert wird. Jede Rasterzelle entspricht einem Pixel in der zu berechnenden Textur und bekommt einen eindeutigen Farbwert zugewiesen, der aus den zugrundeliegenden Punktdaten bestimmt wird, indem die Farbwerte der k nächsten Nachbarn zum Mittelpunkt des Pixels gemittelt werden. Bei diesem Ansatz ist es erforderlich, die Ausrichtung der Bounding-Box optimal zu wählen, um die Größe der Textur möglichst klein zu halten. Achsenparallele Bounding-Boxen sind zwar leicht zu bestimmen, minimieren die benötigte Fläche oftmals aber nicht.

Dieser Effekt ist in den ersten beiden Bildern der Abbildung 3.46 zu sehen. Das linke Bild zeigt ein Polygon und die dazugehörige achsenparallele Bounding-Box. Im mittleren Bild ist die Bounding-Box an den Hauptachsen des Polygons ausgerichtet. In beiden Bildern ist die zur Texturierung nicht nutzbare Fläche grau eingezeichnet. Man kann deutlich erkennen, dass im mittleren Bild die benötigte Fläche wesentlich kleiner ist (nur ca. 70% der achsenparallelen Variante) und zugleich der effektiv für die Polygontexur nutzbare Bereich deutlich größer ist.

Aus diesem Grund wird die Bounding-Box in einem lokalen Koordinatensystem, das von den normierten Hauptachsen $\boldsymbol{X'}$ und $\boldsymbol{Y'}$ des Polygons aufgespannt wird, berechnet. Die Hauptachsen werden mittels Hauptkomponentenanalyse bestimmt. Zur Berechnung der Bounding-Box werden alle Polygonvertices in das durch $\boldsymbol{X'}$ und $\boldsymbol{Y'}$ definierte Koordinatensystem projiziert und die minimal und maximal auftretenden Punkte $\boldsymbol{B}_{\text{min}}$ und $\boldsymbol{B}_{\text{max}}$ bestimmt. Ausgehend von $\boldsymbol{B}_{\text{min}}$ werden nun entlang der Hauptachsen die Pixelpositionen $\boldsymbol{P}_{i,j}$ der Textur berechnet:

$$\boldsymbol{P}_{i.j} = \boldsymbol{B}_{\text{min}} + \begin{pmatrix} i \cdot \boldsymbol{X'} \cdot \text{step}_x \\ j \cdot \boldsymbol{Y'} \cdot \text{step}_y \end{pmatrix}$$

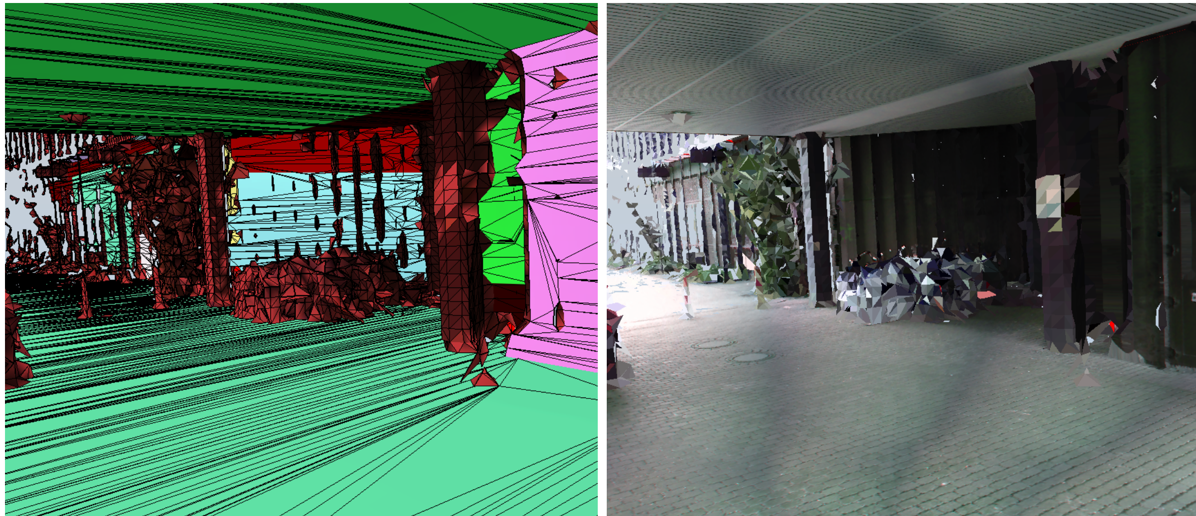


Abbildung 3.47: Beispiel einer Rekonstruktion mit Textur aus einer farbigen Punktwolke. Die Texturen für Fußboden, Decke und Wände verleihen der Szene einen fotorealistischen Eindruck. Kleine Flächen ohne Textur wurden einfarbig gerendert. Auch dies führt zu einem natürlicheren Eindruck bei der Wiedergabe, z.B. bei den rot-weißen Pfosten.

Die Schrittweiten step_x und step_y berechnen sich aus Höhe d_y und Breite d_x der Bounding-Box sowie der eingestellten Texelgröße t (Benennung siehe Abbildung 3.46). Die Indices laufen dabei von $i = 1..\lceil dx/t \rceil$ und $j = 1..\lceil dy/t \rceil$:

$$\text{step}_x = d_x/t$$
$$\text{step}_y = d_y/t$$

Für kleine Flächen, die nur aus sehr wenigen Dreiecken bestehen, werden keine eigenen Texturen berechnet, da deren Fläche sehr klein gegenüber zusammengefassten Regionen ist. Die Texturen beständen dementsprechend nur aus einigen wenigen Pixeln und somit wäre keine wesentliche Steigerung des Detailgrades zu erwarten. Aus diesem Grund werden solche Flächen einheitlich mit einer aus den nächsten Nachbarpunkten gemittelten Durchschnittsfarbe eingefärbt. Die Auflösung der berechneten Texturen hängt bei dem vorgestellten Verfahren von der eingestellten Texelgröße ab. Je kleiner dieser Wert ist, um so mehr Pixel werden generiert. Der potenziell darstellbare Detailgrad steigt also. Allerdings wächst auch die Anzahl der erzeugten Pixel quadratisch, so dass in Szenen mit großflächigen Polygonen die Texturen sehr schnell sehr groß werden und daher viel Speicher benötigt wird. Zudem hängt der erreichbare Detailgrad auch stark von der Qualität der Eingabedaten ab, so dass der Parameter so angepasst werden muss, dass ein akzeptabler Kompromiss zwischen Texturgröße und Auflösung gefunden wird.

Ein Beispiel für eine Rekonstruktion mit Texturen zeigt Abbildung 3.47. Die Szene wurde mit einem Faro-Laserscanner aufgenommen und zeigt den Vorplatz (Untergeschoss) des AVZ-Gebäudes der Universität Osnabrück. Durch den Einsatz der Texturen sind die Pflasterung des Platzes und die gemaserten Oberflächen der Gebäude sowie die Vertäfelung der Überdachung deutlich zu erkennen. Einige der generierten Texturen sind in Abbildung 3.48 gezeigt. In den Bereichen, die

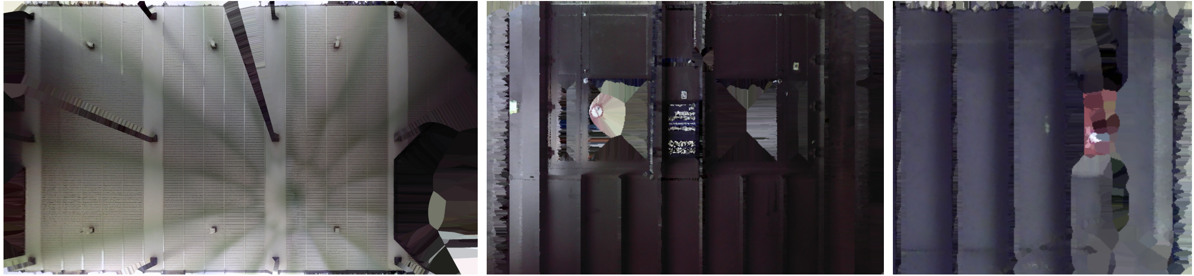


Abbildung 3.48: Beispiele für automatisch erzeugte Texturen der Rekonstruktion aus Abbildung 3.48

beim Texture-Mapping verwendet werden, erscheinen die Bilder fotorealistisch. Die teilweise linienförmigen Artefakte in den nicht genutzten Bereichen sind auf die Farbübertragung mittels Nächster-Nachbar-Suche zurückzuführen, da für jeden Pixel innerhalb der Bounding-Box eines Polygons Farbwerte berechnet werden und sich die Farben dementsprechend außerhalb wiederholen, da auch mit steigendem Abstand zur Polygonkontur mit hoher Wahrscheinlichkeit dieselben Punkte als nächste Nachbarn gefunden werden. Insgesamt wirken die Bilder sehr realistisch, allerdings leicht verschwommen. Die Auflösung kann durch eine kleinere Texelgröße weiter erhöht werden, jedoch hängt die maximal erreichbare Qualität von der Dichte der Eingangsdaten ab.

3.6.3 Ausblick

Derzeit wird für jedes planare Polygon genau eine individuelle Textur erzeugt. Sehr oft bestehen Texturen größerer Flächen aus sich wiederholenden Mustern. Sinnvoll wäre es, solche Muster, z.B. durch Frequenzanalyse, automatisch zu erkennen und daraus kleinere Kacheltexturen zu erzeugen, die von mehreren Polygonen mit ähnlicher Oberflächenstruktur wieder verwendet werden könnten. Die so berechneten Texturen könnten in einer Referenztdatenbank abgespeichert und bei Bedarf abgerufen werden. Auf diese Art und Weise ließe sich der Speicherbedarf bei gleichzeitig guter Darstellungsqualität weiter verringern. Ein solcher Ansatz wird derzeit im Rahmen einer Masterarbeit untersucht [159].

3.7 Parallelisierung mit OpenMP

Viele Schritte der Rekonstruktion laufen nach dem SIMD-Schema ab [55], d.h., es wird bei der Berechnung immer wieder die gleiche Operation auf verschiedenen, unabhängigen Daten ausgeführt. Dies ist z.B. bei der Schätzung der Normalen für die Datenpunkte der Fall. Die berechnete Orientierung der Normalen hängt dabei nur von den Datenpunkten ihrer lokalen k-Nachbarschaft ab. Die Ergebnisse der Schätzung beeinflussen sich gegenseitig nicht. Ein anderes Beispiel ist die Berechnung der Distanzwerte für die Anfragepunkte im Rekonstruktionsgitter. Hier wird für jeden Punkt der Abstand zur Isofläche bestimmt. Ebenso erfolgt die Erzeugung der Dreiecke in der Marching-Cubes-Rekonstruktion lokal in den einzelnen Zellen, es muss lediglich

das Einfügen neuer Vertices synchronisiert werden. Derartige Prozesse können auf modernen Prozessoren mit mehreren Kernen sehr gut parallelisiert werden.

Die gerade skizzierten Algorithmen zeichnen sich dadurch aus, dass sehr viele relativ einfache Instruktionen immer wieder auf den Daten, in der Regel innerhalb einer Schleife, ausgeführt werden. Solche Anwendungsfälle lassen sich mit Hilfe von OpenMP sehr leicht parallelisieren. OpenMP [1] ist eine standardisierte offene Spezifikation für Multi-Prozessing und definiert eine High-Level-API zur Parallelisierung von SIMD-Prozessen für verschiedene Betriebssysteme und Compiler. Der große Vorteil dieser hohen Abstraktion ist, dass sich der Programmierer nicht mit der Umsetzung der Parallelisierung auf die vorliegende Hardware bzw. das verwendete Betriebssystem kümmern muss. Wird OpenMP von einem Compiler unstützt, wird der entsprechend optimierte Maschinencode zur parallelen Abarbeitung direkt erzeugt. Auch werden verschiedene Probleme, z.B. die Datensynchronisierung, direkt in den API-Funktionen behandelt, d.h. der Programmierer kann sich darauf verlassen, dass die entsprechenden Funktionalitäten in den entsprechenden Konstrukten korrekt umgesetzt sind.

Wichtig bei der Verwendung von OpenMP ist, dass die in den parallelisierten Programmteilen benutzten Funktionsaufrufe Thread-sicher sind. So müssen z.B. die verwendeten Bibliotheken zur Nächsten-Nachbar-Suche mehrere parallel zu bearbeitende Suchanfragen zulassen, d.h., die internen Datenstrukturen müssen so angelegt sein, dass durch parallelen Aufruf der implementierten Funktionalitäten keine Datenkonflikte ("Race Conditions") auftreten. Diese Art von Suchanfragen werden z.B. von FLANN und STANN unterstüzt. Sind diese Randbedingungen erfüllt, lassen sich in OpenMP insbesondere Schleifen sehr leicht durch spezifische Compileranweisungen parallelisieren, wie folgendes Beispiel zeigt:

```
#pragma omp parallel for
for( int i = 0; i < (int)m_queryPoints.size(); i++){
    float distance;
    this->m_surface->distance(m_queryPoints[i].m_position, distance);
    m_queryPoints[i].m_distance = distance;
}
```

Das Codebeispiel zeigt die Parallelisierung der Schleife zur Berechnung der Entfernungen der `QueryPoints` zur gegebenen Iso-Fläche (`this->m_surface`). Durch die Direktive `#pragma omp parallel for` wird der Compiler angewiesen, die Schleife zu parallelisieren. Konkret bedeutet dies, dass für jeden Schleifenindex `i` ein eigener Thread angelegt wird. Typischerweise werden die Threads in Gruppen ausgeführt, um den Overhead durch Kontextwechsel zu minimieren und eine bessere Lokalität der Daten zu erreichen [33]. Dazu werden so viele Threads in einer Gruppe erzeugt, wie vom System gleichzeitig abgearbeitet werden können. Diese bekommen dann je nach Bedarf die Schleifenindizes zugewiesen. OpenMP untersützt auch gemeinsamen Speicher für die angelegten Threads ("Shared Memory"). Dies ist zum Beispiel für so genannte Reduktionen, z.B. das parallelisierte Berechnen einer Summe, interessant. Solche Variablen werden durch das Schlüsselwort `shared` deklariert. Die Vermeidung von Schreib-Schreib- und Schreib-Lesekonflikten wird dabei von der OpenMP-Implementierung übernommen. Weitere Konstrukte zur Synchronisierung lassen sich in den entsprechenden Lehrbüchern oder Online-Tutorials finden [1, 31, 201].

Das hier gezeigte Beispiel soll demonstrieren, dass sich durch den Einsatz von OpenMP verschiedene Teile des Codes leicht parallelisieren lassen. Wichtig dabei ist, dass die Algorithmen auf das SIMD-Schema abgebildet werden. Aus diesem Grund erfolgt zum Beispiel die Berechnung der Abstandswerte zur Isofläche erst nach der Erzeugung des Raumgitters, da erst dann feststeht, welche Zellen erzeugt werden mussten, und alle Abhängigkeiten untereinander aufgelöst sind. Eine ausführliche Evaluation der Laufzeiten der vorliegenden Implementierung erfolgt in Kapitel 4.3.

Kapitel 4

Evaluation

In diesem Kapitel wird die Software in Bezug auf die in Abschnitt 3.1 gestellten Anforderungen evaluiert. Im ersten Teil wird untersucht, wie gut die Rekonstruktionen die vermessenen Umgebungen wiedergeben. Dabei wird zunächst die geometrische Genauigkeit geprüft. Anschließend wird die Qualität von automatisch aus Laserscans generierten Texturen betrachtet. Eine wesentliche Anforderung an das Verfahren ist es, die benötigten Karten zeitnah mit möglichst wenig Speicherbedarf aus den Eingangsdaten zu erzeugen. Dieser Aspekt wird in Abschnitt 4.3 evaluiert. Neben dem Laufzeitverhalten der einzelnen Rekonstruktionsstufen wird auch der Einfluss der verschiedenen verfügbaren Bibliotheken zur Nächste-Nachbar-Suche auf die Gesamtlaufzeit untersucht. In Abschnitt 4.3.4 wird die Wirkung verschiedener Parameter auf die Größe der erzeugten Meshes diskutiert. Der letzte Abschnitt dieses Kapitels vergleicht die vorliegende Software mit anderen Programmen zur Oberflächenrekonstruktion.

4.1 Kartengenauigkeit

Sollen die erzeugten Karten auf einem mobilen Roboter oder in einer Simulationsumgebung als Umgebungsrepräsentation verwendet werden, ist es von besonderer Bedeutung, dass die Karten die erfasste Szenerie korrekt wiedergeben. Fehler in der Karte können auf einem Roboter dazu führen, dass potenzielle Hindernisse nicht als solche erkannt werden, und es kann im schlimmsten Fall zu einer Kollision mit einem nicht in der Karte verzeichneten Objekt kommen. Auch Objekte, die zwar in der Karte modelliert sind, aber inzwischen aus der Szene entfernt wurden, können Auswirkungen auf die Performanz eines robotischen Systems, z.B. bei der Pfadplanung, haben. In der Regel ist es nicht erwünscht, dass Trajektorien um nicht in der Szene vorhandene Objekte geplant werden. Ähnliche Überlegungen gelten für die Verwendung in Simulationsumgebungen. Hier bilden die verwendeten Karten die Grundlage für alle simulierten Daten. Fehler in der Rekonstruktion verursachen falsche Sensorwerte und erschweren die Arbeit mit dem Simulator, da keine verlässlichen Sensorwerte generiert werden. Im Folgenden wird daher die geometrische Genauigkeit der Rekonstruktionen untersucht.

Abbildung 4.1: Ausschnitte aus den verwendeten Referenzdatensätzen: Büroumgebung, Straßenszene und Kirchenkuppeln.

Die Evaluation der geometrischen Genauigkeit wird durch verschiedene Experimente untersucht. Zunächst wird die mit den zur Verfügung stehenden Mitteln maximal erreichbare Qualität bestimmt, indem drei hoch aufgelöste Datensätze aus verschiedenen Umgebungen betrachtet werden. Dazu werden die Parameter bei der Rekonstruktion und Optimierung so gewählt, dass ein subjektiv optimales Ergebnis erreicht wird. Die auf diese Art und Weise erzeugten Meshes werden anschließend mit Hilfe des Open-Source-Programms `CloudCompare` [67, 68] gegen die Ausgangsdaten verglichen. Mit Hilfe dieses Programms lassen sich Punktwolken und Meshes vergleichen. Dabei wird der Abstand der Punkte im Referenzdatensatz zum Mesh bzw. der anderen Punktwolke bestimmt. Diese Abstände werden in Falschfarben dargestellt. Neben der farbigen Darstellung der Unterschiede gibt es auch einige einfache statistische Analysefunktionen wie Fehlerhistogramme und Berechnung von Mittelwert und Standardabweichung der Punktabstände zur Referenz. Mit Hilfe dieser Analyse lässt sich bestimmen, inwiefern die optimierten Meshes die tatsächlich gemessenen Daten wiedergeben. Weil gerade hochauflösende Laserscans wenig Rauschen aufweisen, lassen sich mit solchen Daten Problemfälle und systematische Fehler in den Rekonstruktionen leicht darstellen.

Da hochauflösende Laserscanner in der mobilen Robotik selten zum Einsatz kommen, sollen im zweiten Teil der Evaluation Rekonstruktionen aus Kinect-Daten und Laserscans von rotierenden SICK-Scannern betrachtet werden. Dadurch soll abgeschätzt werden, welche Kartengüte sich in der Praxis mit mobilen Robotern erreichen lässt.

4.1.1 Rekonstruktionen aus hochaufgelösten Scans

Die bei der Auswertung verwendeten Datensätze sollen beispielhaft verschiedene Typen von Umgebungen repräsentieren. Der erste Datensatz ("Büro") wurde mit einem Leica HDS 6000 Laserscanner in einem eingerichteten Büro aufgenommen und besteht aus 20 Millionen Datenpunkten. Dieser Scanner erreicht dabei laut Hersteller eine Distanzgenauigkeit von unter 2 mm. Die Erfahrung in der Praxis hat gezeigt, dass sich in Innenräumen auch kleinere Strukturen, wie z.B. Poster an der Wand, noch auflösen lassen. Dieser Datensatz soll eine typische Indoor-Umgebung repräsentieren und enthält viele ebene Anteile. Das eingescannte Büro war nicht ausgeräumt, d.h. es sind auch etliche kleinere, nicht-planare Objekte enthalten.

Der zweite betrachtete Datensatz ("Kirche") wurde zur archäologischen Dokumentation des Inneren einer griechischen Kirche aufgenommen. Dieser Datensatz enthält neben ebenen Flächen auch sehr viele gekrümmte Oberflächen an den Kuppeln und an verschiedenen Säulen in der Szene. Er soll besonders den Einfluss der implementierten Optimierungsverfahren auf gekrümmte Flächen verdeutlichen. Er wurde mit einem Leica HDS 3000 Scanner aufgenommen, die Punktgenauigkeit beträgt 6 mm. Weitere Details zu diesem Datensatz finden sich in [22].

Der dritte Datensatz ("Straße") wurde mit einem Riegel Laserscanner aufgenommen (Distanzgenauigkeit ca. 5 mm) und zeigt eine Straßenszene. Er enthält neben einigen Gebäuden und Straßenzügen auch Bäume und diverse Objekte wie Pfosten und Briefkästen am Straßenrand. Diese Aufnahme soll stellvertretend für eine typische Outdoorszenerie stehen. Ausschnitte aus den drei betrachteten Datensätzen zeigt Abbildung 4.1. Die evaluierten Rekonstruktionen sind in Abbildung 4.2 dargestellt. Die linke Spalte zeigt die optimierten Rekonstruktionen, die rechte Spalte die ermittelten Falschfarbenbilder in einem Farbgradienten von blau (kleiner Fehler) bis rot (großer Abstand zur Rekonstruktion). Man kann in den Bildern deutlich erkennen, dass die größten Abweichungen an gekrümmten und unregelmäßigen Strukturen auftreten.

Um eine Vergleichbarkeit der Daten zu gewährleisten, wurden alle Punktwolken so skaliert, dass sämtliche Abstände in mm-Einheiten vorlagen. Für eine optimale Rekonstruktion wurden die Voxelgrößen so eingestellt, dass die maximal mögliche Auflösung erreicht wurde, ohne dass Löcher in zusammenhängenden dicht abgetasteten Flächen entstanden. Für den Kirchendatensatz wurde ein Wert von 30 mm eingestellt, die Straßenszene wurde mit 50 mm rekonstruiert. und der Bürodatensatz konnte mit einer Voxelgröße von 15 mm rekonstruiert werden. Die für die Normalenschätzung benötigen Parameter k_n und k_i wurden mit $k_n = 100$ und $k_i = 50$ relativ hoch gewählt, da die vorliegenden Datensätze sehr dicht waren und so eine sehr hohe Qualität der geschätzten Normalen erreicht werden konnte. Diese hohen Werte gingen zwar zu Lasten der Laufzeit, welche in diesen Experimenten aber vernachlässigt wurde, da hier der Fokus auf Rekonstruktionsqualität lag. Brauchbare Rekonstruktionen lassen sich aber auch mit niedrigeren Werten erzielen. So liefern $k_n = 10$ und $k_i = 10$ bereits gute Ergebnisse. Die Parameter zur Meshoptimierung wurden den einzelnen Datensätzen angepasst. Bei der Kirche wurde ein relativ strenger Schwellwert von 0.99 für das Region Growing gewählt, damit bei der Projektion in die Ausgleichsebenen die Flächen in den Kuppeln nicht flach gedrückt wurden. Bei den anderen Datensätzen konnte der Standardwert von 0.85 verwendet werden. Ein weiterer wichtiger Faktor ist der Schwellwert für die Zusammenfassung von Polygonsegmenten bei der Erzeugung der Konturpolygone der ebenen Flächen. Die Optimierung wurde so eingestellt, dass signifikante Kanten, wie z.B. senkrechte Wandverläufe, optimal zusammengefasst wurden, ohne Ecken abzuschneiden. Bis auf den Straßendatensatz, bei dem der Standardwert von 0.1 auf 0.5 hoch gesetzt werden musste, konnte die ursprüngliche Einstellung beibehalten werden. Auf die Anpassung der weiteren Optimierungsparameter soll hier nicht weiter eingegangen werden, da sich durch die Variation der Standardwerte nur noch minimale Verbesserungen erzielen ließen. Eine Liste aller Parameter zusammen mit einer Erklärung ihrer Bedeutung und den voreingestellten Standardwerten findet sich in Anhang A.

Mit der so gefundenen Parametrierung wurden dann die implementierten Marching-Cubes-Varianten Standard Marching Cubes (MC), Marching Tetrahedrons (MT), Planar Marching Cubes

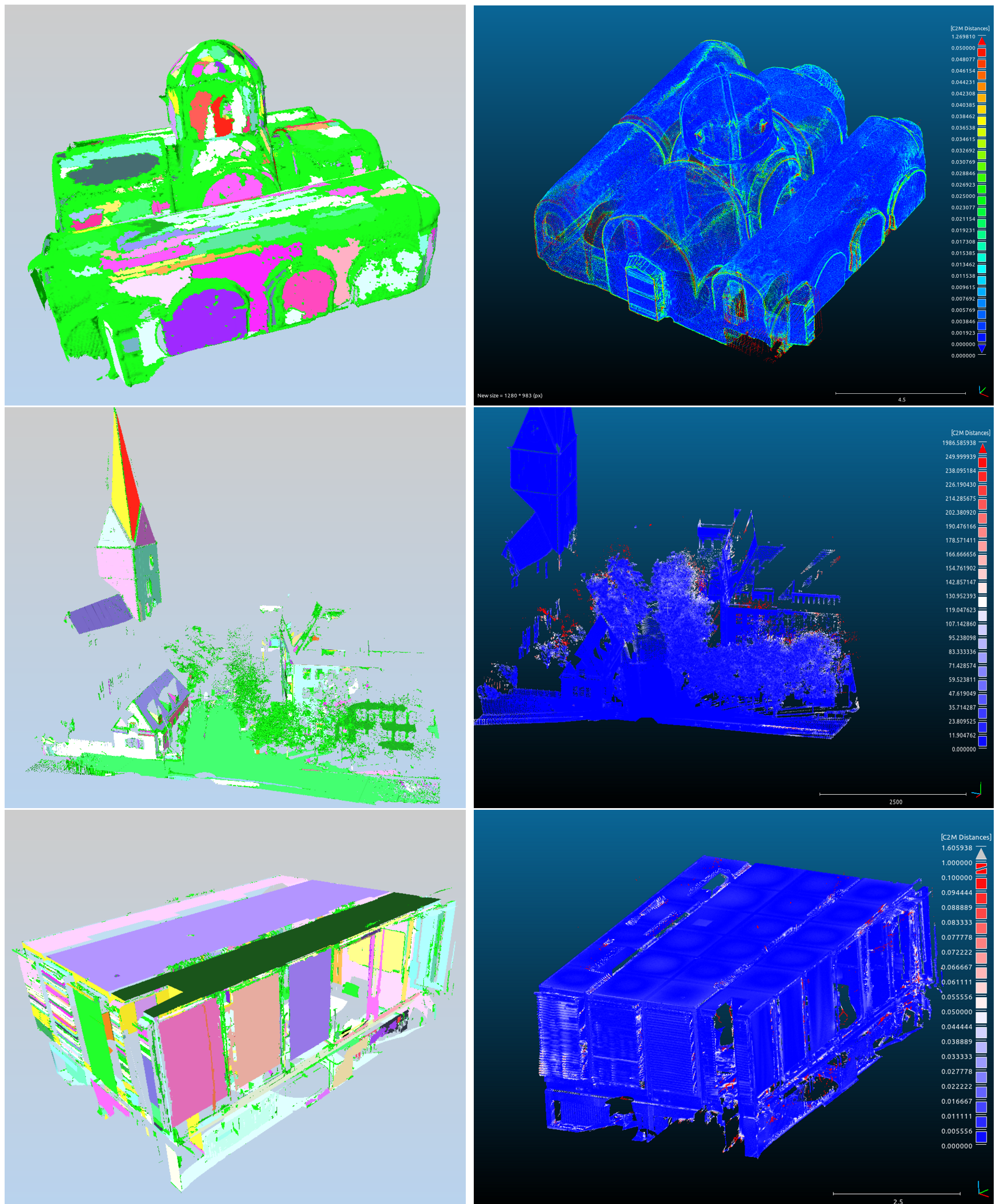


Abbildung 4.2: Vergleich der Punktwolken mit der optimierten Rekonstruktion in `CloudCompare`. Die linke Spalte zeigt die geclusterten optimierten Meshes, die rechte Spalte die nach Punktabstand (von blau nach rot) eingefärbte Ausgangspunktwolke.

Tabelle 4.1: Ergebnisse des Vergleichs der Ausgangspunktdaten mit den automatisch erstellten Rekonstruktionen. Dargestellt sind die Anzahl der Dreiecke, der mittlere Punktabstand von Punktwolke zu Mesh und die dazugehörige Standardabweichung für verschiedene Verfahren vor und nach der Optimierung für drei hochaufgelöste Laserscans (Angaben der Abstände in mm).

		Dreiecke		Mittelwert		Abweichung		Max. Abstand	
		Rek.	Opt.	Rek.	Opt.	Rek.	Opt.	Rek.	Opt.
Büro	MC	885 377	276 039	1.51	0.45	1.93	2.44	1678.96	1678.96
	MT	3 208 144	908 980	1.04	0.41	1.71	2.32	1678.96	1678.96
	PMC	885 377	224 172	1.48	0.58	1.93	2.61	1678.96	1678.96
	XMC	887 509	279 131	1.59	0.50	1.92	2.49	1678.96	1678.96
Kirche	MC	1 066 811	572 995	1.50	2.56	4.27	4.55	1280.21	1280.25
	MT	3 865 614	1 956 094	1.00	2.49	3.46	4.13	1050.08	1088.21
	PMC	1 066 811	548 957	1.40	2.77	4.22	4.59	1280.21	1280.25
	XMC	1 066 821	549 283	1.59	3.22	4.27	4.71	1280.21	1280.25
Straße	MC	2 461 545	952 979	2.22	19.31	4.22	5.06	1974.43	875.11
	MT	12 625 858	6 421 956	0.55	0.64	2.55	2.45	1940.12	726.76
	PMC	2 462 545	1 075 960	2.15	12.39	4.20	4.75	1974.43	875.11
	XMC	2 507 277	1 040 821	2.22	11.29	4.21	4.79	1974.43	875.11

(PMC) und Extended Marching Cubes (XMC) getestet und die Rekonstruktionen vor und nach der Optimierung mit der Punktwolke verglichen. Beispielhafte Ergebnisse der Rekonstruktionen sowie der mit `CloudCompare` erstellten Falschfarbenbilder sind in Abbildung 4.2 dargestellt. Tabelle 4.1 zeigt die ermittelten statistischen Kennzahlen des Vergleichs der Punktwolke mit den Rekonstruktionen: mittlerer Punktabstand, Standardabweichung des Abstandes und maximal gemessener Abstand.

Auffällig ist zunächst, dass die Rekonstruktion mit Marching Tetrahedrons in allen drei Datensätzen den niedrigsten quadratischen Abstand liefert. Dies ist nicht weiter verwunderlich, da durch die Unterteilung der Voxel im Rekonstruktionsgitter eine höhere effektive Auflösung als bei den Marching-Cubes-Verfahren auftritt. Dafür ist die Anzahl der erzeugten Dreiecke jeweils deutlich höher. Bei den anderen Verfahren schneidet PMC in allen Fällen am besten ab. Dies ist dadurch zu erklären, dass durch die Optimierung der Randkonturen die rekonstruierten Flächen die Punkte deutlich besser approximieren als die standard Marching-Cubes-Muster.

Weiterhin fällt auf, dass die mittels Extended Marching Cubes berechneten Approximationen nicht ersichtlich besser sind als die MC-Rekonstruktionen. Eigentlich wäre zu erwarten gewesen, dass zumindest im Bürodatensatz ein besseres Ergebnis erzielt worden wäre, da dort viele scharfe Kanten vorhanden sind. Bei genauerer Betrachtung hat sich herausgestellt, dass aufgrund des im Laserscan vorhandenen Kantenrauschens die XMC-Optimierungsheuristiken nur in we-

nigen Fällen eine Kante detektieren. Dieses Ergebnis passt zu der bereits gemachten Aussage, dass das Verfahren auf synthetischen Datensätzen gut funktioniert, bei der Rekonstruktion aus Punktdaten allerdings selten eine Verbesserung bringt.

Nach der Meshoptimierung war der mittlere Fehler beim Bürodatensatz bei allen angewendeten Verfahren geringer als im initialen Mesh. In den anderen Datensätzen ist der Fehlerwert zum Teil deutlich angestiegen. Dieser Effekt spiegelt die Tatsache wider, dass die implementierten Optimierungsverfahren auf planare Umgebungen ausgelegt sind. Daher ist es nicht verwunderlich, dass sich die Qualität der Rekonstruktion durch die Optimierung in Fällen, in denen diese Randbedingung verletzt ist, verschlechtert. Im Kirchendatensatz fällt der Anstieg des mittleren Fehlers noch vergleichsweise moderat aus, da auch dort sehr viele ebene Flächen existieren. Der dennoch vorhandene Anstieg ist darauf zurück zu führen, dass trotz des strengen Schwellwertes bei der Meshoptimierung Flächen in den Kuppeln geclustert und deren Vertices dementsprechend in die Ausgleichsebene gezogen wurden. Dadurch wurden Teile der in der Punktwolke repräsentierten Rundungen platt gedrückt und der ermittelte Punktabstand stieg. Am deutlichsten steigt der Fehler im Straßendatensatz. Auch hier tritt der Effekt auf, dass Rundungen zusammengefasst werden. Allerdings werden hier vor allem an der Straße und den Dachflächen besonders viele Vertices in die ermittelten Ausgleichsebenen gezogen und einige der in der initialen Rekonstruktion vorhandenen Details, z.B. Fugen zwischen den Steinen des Kopfsteinpflasters der Straße und die individuellen Erhebungen der Dachpfannen, gehen verloren, wodurch der mittlere Fehler merklich ansteigt. Durch die gerade beschriebenen Effekte ist auch zu erklären, dass in allen Fällen die ermittelte Standardabweichung nach der Optimierung ansteigt.

Die gemessenen maximalen Entfernungswerte haben hingegen wenig Aussagekraft. Sie sind bei allen Verfahren vor und nach der Optimierung sehr ähnlich, weil die Filterung von Artefakten im Mesh dafür sorgt, dass zu den die Messpunkten, aus denen die Artefakte ursprünglich entstanden sind, keine in der Nähe liegenden Korrespondenzdreiecke mehr gefunden werden. Diese Werte liefern daher für den vorliegenden Anwendungsfall keine verwertbaren Informationen und werden hier lediglich der Vollständigkeit halber wiedergegeben.

Die Evaluation am Beispiel der hoch aufgelösten Laserscans hat gezeigt, dass sich mit den implementierten Verfahren in planaren Umgebungen Rekonstruktionen erzielen lassen, die die in Kapitel 3.1 gestellten Anforderungen voll erfüllen. Der mittlere Abstand der Punkte der Ausgangsdaten betrug in der Bürorekonstruktion weniger als einen Millimeter. Bei den anderen Datensätzen waren die ermittelten Werte höher, der Fehler blieb aber im unteren Zentimeterbereich. Besonders in Scans, in denen keine Planarität gegeben ist, steigt der Fehler nach der Reduzierung der Dreiecksflächen an. Dies ist besonders am Straßendatensatz zu erkennen. An dieser Stelle darauf hingewiesen, dass die hier erfolgte Evaluation durch Vergleich der Punktwolken mit den Rekonstruktionen viele Aspekte nicht berücksicht: So werden Dreiecke im Mesh, die durch die Marching-Cubes-Muster entstehen und durch keine nahen Messpunkte gestützt werden, nicht einbezogen. Ähnliches gilt für aufgefüllte Löcher durch Scanschatten. Trotzdem lässt sich zusammenfassend feststellen, dass sich mit den implementierten Verfahren in ebenen Umgebungen sehr gute Rekonstruktionen erzeugen lassen, wenn die Qualität der Ausgangsdaten entsprechend hoch ist.

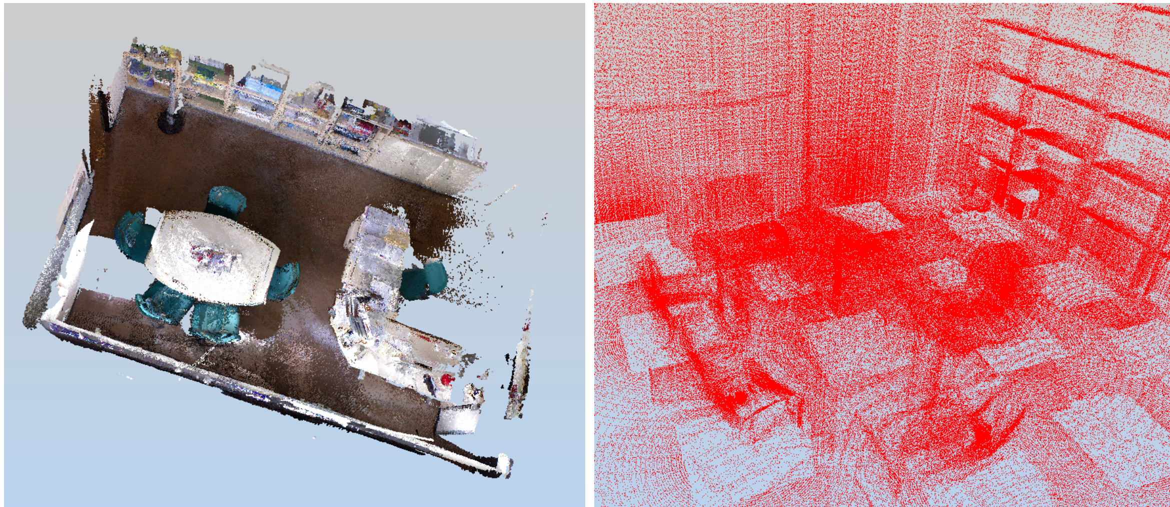


Abbildung 4.3: Vermessung der Büroumgebung mit Kinect (links) und Rotationseinheit (rechts). Die Registrierung erfolgte vollautomatisch, basierend auf der Odometrieschätzung der Roboterpose.

In der Praxis stehen derart hochauflösende Daten allerdings nur selten zur Verfügung. Um die Güte der Rekonstruktion aus realistischen Roboterdaten zu evaluieren, wurde die Büroumgebung noch einmal mit einem teleoperierten Kurt-Roboter, ausgestattet mit Kinect-Kamera und rotierendem Laserscanner, vermessen.

4.1.2 Umgebungsrekonstruktion mit einem mobilen Roboter

Mit Hilfe des Kurt-Roboters wurden in der Büroumgebung zwei weitere Datensätze aufgenommen. Der erste Datensatz besteht aus insgesamt 380 Kinect-Frames, der zweite aus 30 Laserscans, die mit der Rotationseinheit des Roboters aufgenommen wurden. Die einzelnen Datensätze wurden vollautomatisch mit `slam6d` registriert [141]. Als Ausgangsposeschätzung dienten dazu die von der Odometrie geschätzten Roboterposen zum Zeitpunkt der Aufnahme. Die daraus gewonnenen Punktwolken sind in Abbildung 4.3 dargestellt. Die Kinect-Punktwolke ist im linken Bild gezeigt. Man kann an der linken Stirnwand deutlich erkennen, dass es am Ende der Fahrt einen deutlichen Registrierungsfehler gibt, der dazu führt, dass Wand und Teile der Tische doppelt vorhanden sind. Die Registrierung der Scans der Rotationseinheit hat besser funktioniert (siehe rechtes Bild), aber auch hier gibt es kleine Fehler in der Registrierung (zu sehen z.B. an den Stuhllehnen).

Die Ergebnisse der Rekonstruktionen zeigt Abbildung 4.4. Die ermittelten Abweichungen zwischen Punktwolke und Mesh sind in Tabelle 4.2 zusammengefasst. Die Parameter wurden wieder so gewählt, dass eine möglichst gute Approximation erreicht wird. Wie erwartet, weist die Kinect-Rekonstruktion aufgrund des stärkeren Rauschens der Eingangsdaten höhere Abweichungen im Vergleich zum Rotationseinheitsscan auf. Das stärkere Rauschen in den Eingangsdaten sorgt hier allerdings dafür, dass die mittleren Punktabstände nach der Ebenenoptimierung zunehmen. Da die Vertices des Meshes in die Ausgleichsebene gezogen werden, schwanken die stärker verrausch-

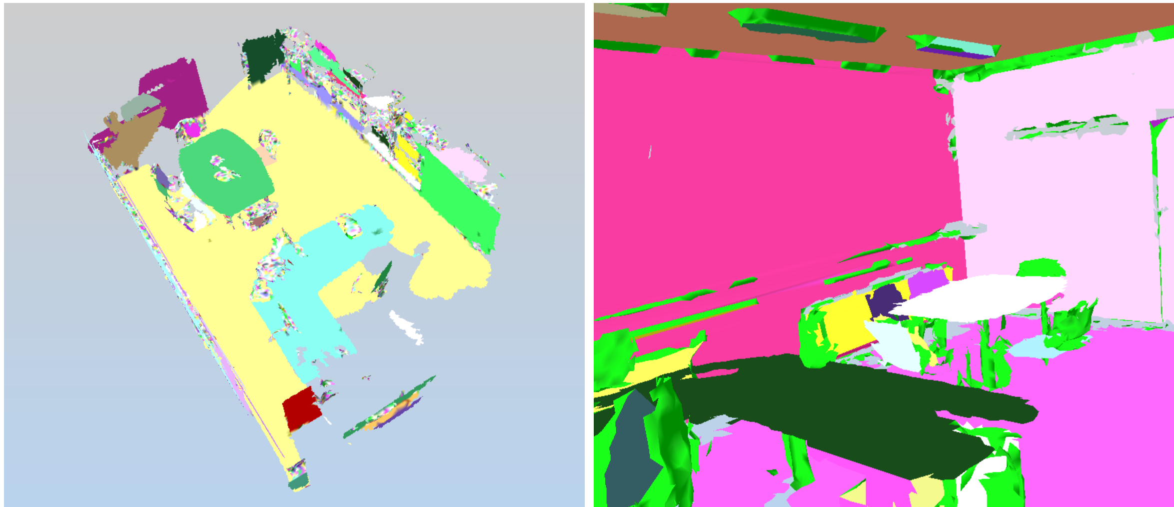


Abbildung 4.4: Rekonstruktionen aus Kinect- und Rotationseinheitsdaten.

ten Messpunkte um die errechnete Ausgleichsebene. Da sich mit Hilfe dieses Evaluationsverfahrens offensichtlich keine verlässlichen Aussagen über die geometrische Genauigkeit machen lassen, wurden in den Rekonstruktionen einige markante Größen wie Länge und Breite des Raumes, Höhe der Tische usw. in Meshlab ausgemessen. Die dazugehörigen realen Daten wurden mit Hilfe eines Laser-Distometers im Büro ermittelt. Zusätzlich wurden die festgelegten Kennzahlen auch in den Eingabedaten nachgemessen. Mit dem in Meshlab vorhandenen Messtool lassen sich nur Distanzen zwischen Vertices und Punkten bestimmen. Daher wurden die evaluierten Maße so gewählt, dass sie sich an Knicken in der Geometrie befinden, damit die Wahrscheinlichkeit hoch ist, dort einen geeigneten Vertex zu finden. Die Ergebnisse der Evaluation sind in Tabelle 4.3 zusammengefasst.

Wie zu erwarten, liefern die Rekonstruktionen aus den Leica-Daten die besten Werte beim Vergleich mit Ground Truth. In den Rekonstruktionen aus den anderen Datensätzen zeigen sich stellenweise Abweichungen im cm-Bereich. Diese lassen sich erklären, wenn man die Eingabedaten genauer betrachtet. So ist die Registrierung der Daten in dem Laserscan und insbesondere in den Kinect-Punktwolke nicht sehr genau. Deshalb erscheint die Tischplatte durch versetzte Mehrfachmessungen in diesen Datensätzen größer als sie eigentlich ist, so dass eine größere Diagonale ermittelt wird. Der gleiche Effekt zeigt sich auch an der Türbreite. Diese wird in den verrauschten Datensätzen unterschätzt. Die Tischhöhe, die dadurch, dass der Roboter in einer Ebene fuhr, nicht von den Registrierungsungenauigkeiten betroffen war, wird in allen Rekonstruktionen sehr gut wiedergegeben. Die Werte schwanken lediglich um wenige Millimeter. Bei den großen Aufmaßen wie den Wandlängen zeigen sich auch beim Leica-Scanner Abweichungen im Zentimeterbereich. Dies kann darauf zurückgeführt werden, dass die Wände im betrachteten Büro nicht genau parallel sind. Wenn die in den Rekonstruktionen ausgewählten Vertices zur Abstandsbestimmung nicht genau mit den im Büro vermessenen Positionen übereinstimmen, können dadurch unterschiedliche Werte entstehen.

Tabelle 4.2: Vergleich der Rekonstruktionen aus den Kinect- und Rotationseinheitsdaten der Büroumgebung mit den Eingangsdaten. Angegeben sind Anzahl Dreieke vor und nach der Meshoptimierung sowie ermittelte Punktabstände in mm).

		Dreiecke		Mittelwert		Abweichung		Max. Abstand	
		Rek.	Opt.	Rek.	Opt.	Rek.	Opt.	Rek.	Opt.
Kinect	MC	315 208	74 446	2.53	11.90	3.49	4.90	526.30	526.30
	MT	1 106 150	165 156	2.28	11.51	3.40	5.41	263.67	182.47
	PMC	315 208	82 567	2.52	11.99	3.49	4.95	526.30	526.30
	XMC	318 208	97 151	2.53	13.83	3.49	5.13	526.30	526.30
Laserscan	MC	259 940	82 879	10.94	7.26	5.30	4.95	1184.30	1184.30
	MT	1 013 981	314 892	10.14	6.50	5.00	4.57	855.33	855.33
	PMC	259 940	87 438	10.31	7.03	5.25	4.92	1184.30	1184.30
	XMC	260 348	81 733	10.78	7.26	5.30	4.94	1184.30	1184.30

4.1.3 Fazit Geometrische Genauigkeit

Die hier durchgeführten Messungen haben gezeigt, dass die mit LVR erstellten Rekonstruktionen die Eingabedaten sehr gut approximieren. Die mittleren Abstände zwischen Punktwolke und Rekonstruktion lagen sowohl bei aufgelösten Laserscans im Millimeter-Bereich. Bei stärker rauschenden Eingangsdaten war der Fehler in der Größenordnung von einem Zentimeter. In Umgebungen mit vielen ebenen Anteilen konnte mit der Software eine deutliche Reduktion der Anzahl der Dreiecke in den Meshes erreicht werden, ohne einen signifikanten Anstieg des mittleren Abstandsfehlers zu verursachen. In der Rekonstruktion aus dem hoch aufgelösten Datensatz konnte der Fehler sogar verringert werden. Sowohl in den verrauschten aus auch in den hochauflösenden Daten wurde die Tischhöhe äußerst präzise rekonstruiert. Andere Kennzahlen wie Länge von Wänden usw. weichen aufgrund von Registrierungsfehlern und systematischen Messfehlern um einige Zentimeter von den realen Daten ab. Dieser Fehler ist aber bereits in den Eingangsdaten vorhanden und wird nicht durch Rekonstruktion erzeugt. Insgesamt hat sich gezeigt, dass sich auch aus den vergleichsweise verrauschten und dünnen Eingangsdaten von Kinect und rotierendem Laserscanner akkurate Rekonstruktionen erzeugen lassen, die für robotische Anwendungen zu gebrauchen sind und den in Abschnitt 3.1 gestellten Anforderungen entsprechen. Von den gestesten Marching-Cubes-Varianten lieferte PMC bei gegebener Auflösung die genauesten Ergebnisse. Durch Marching Tetrahedrons lässt sich der Fehler nur leicht verringern, die Anzahl der generierten Dreiecke ist dabei aber deutlich höher, so das PMC für den praktischen Einsatz das Verfahren der Wahl ist.

Tabelle 4.3: Vergleich von Kenngrößen in den initialen und optimierten Rekonstruktionen im Vergleich mit aus den Punktwolken bzw. per Hand nachgemessenen Werten (Angaben in mm).

		Türbreite	Tischhöhe	Tischdiagonale	Stirnwand	Seitenwand
Initial	Kinect	889	705	1847	4069	n/v
	Laserscan	880	711	1828	4623	6999
	Leica	930	700	1738	4690	7084
Optimiert	Kinect	920	702	1844	4077	n/v
	Laserscan	906	701	1857	4640	6959
	Leica	930	708	1740	4690	7084
Rohdaten	Kinect	903	798	1856	4061	6987
	Laserscan	887	709	1835	4620	6987
	Leica	935	700	1739	4698	7080
Handmessung		935	700	1741	4680	7100

4.2 Texturemapping

Für die Qualität des Texture Mappings sind zwei Aspekte von zentraler Bedeutung: die richtige Position der Texturen, d.h. die Informationen, die aus den vorhandenen Farbinformationen gewonnen wurden, müssen an den korrekten Positionen in der Polygonkarte platziert sein, sowie die Auflösung der Texturen.

Die Beurteilung der Positionierung der Texturen erfolgt durch visuelle Inspektion. Dabei werden insbesondere die Ränder von texturierten Flächen betrachtet, da hier vorhandene Versätze in der Positionierung stark auffallen. Stimmt die Texturierung an den Rändern eines Polygons

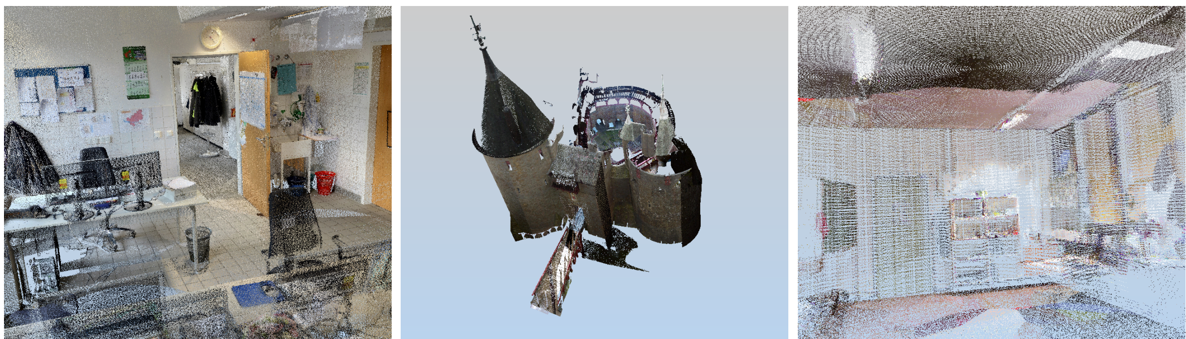


Abbildung 4.5: Datensätze zur Evaluation der Texturierung: Büro (links) und Schloss (Mitte) wurden mit terrestrischen Scannern aufgenommen. Der rechte Scan stammt von der Kurt-Rotationseinheit und wurde mit Kinect-Daten eingefärbt.

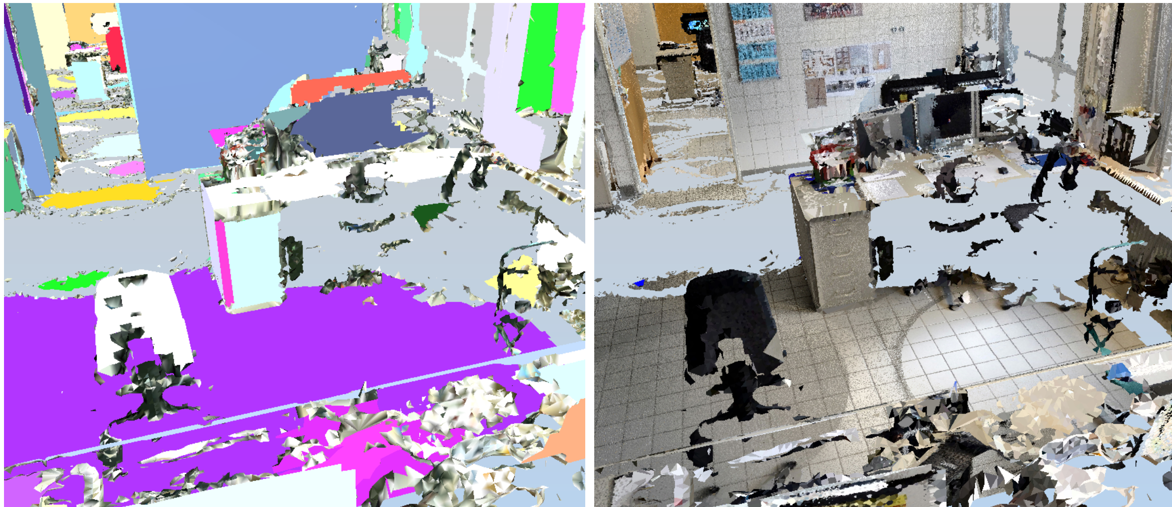


Abbildung 4.6: Texturierte Rekonstruktion des Bürodatensatzes.

in beiden Richtungen, kann davon ausgegangen werden, dass auch innerhalb der Fläche keine Verzerrungen vorliegen und die vorhandenen Farbinformationen korrekt abgebildet werden. Für eine stichprobenartige geometrische Evaluation der Texturen könnten prinzipiell auch einzelne markante Features in der Karte und der Punktwolke vermessen werden. Leider erlauben die zur Verfügung stehenden 3D-Tools nur die Anzeige von Vertexpositionen. Da die relevanten Punkte der Texturen in der Regel innerhalb der Polygone liegen, scheidet diese Möglichkeit der Evaluation leider aus.

Zur Beurteilung des Verfahrens werden drei Datensätze betrachtet: zwei eingefärbte hochaufgelöste Laserscans sowie eine mit Farbwerten einer Kinect-Kamera eingefärbte Punktwolke der Kurt-Rotationseinheit. Bei den hochaufgelösten Scans handelt es sich um den Scan eines Büros und einer mittelalterlichen Burg ("Schlosscan"). Der hochaufgelöste Büroscan soll hier stellvertretend für planare Umgebungen stehen. Der Schlossscan besteht aus vielen Rundungen und weist nur relativ wenige ebene Flächen auf. An diesem Beispiel soll dargestellt werden, welche Ergebnisse sich in nicht-planaren Umgebungen erzielen lassen. Anhand des dritten Datensatzes soll die in verrauschten Daten erreichbare Texturqualität abgeschätzt werden. Ausschnitte aus den verwendeten Eingangsdaten sind in Abbildung 4.5 gezeigt. Die Ergebnisse der Rekonstruktionen werden in den folgenden Abschnitten besprochen.

4.2.1 Evaluation eingefärbter Punktwolken

Der erste Testdatensatz wurde von der Hamburger Polizei mit einem Zoller+Froehlich-Scanner erstellt. Er besteht aus 10 Einzelscans mit je ca. 40 Millionen Punkten. Für die hier gezeigten Experimente wurden die Eingangsdaten mittels Modulofilter auf insgesamt 80 Millionen Punkte reduziert, so dass noch ausreichend Arbeitsspeicher für die Rekonstruktion zur Verfügung stand. Die Farbwerte in diesem Datensatz wurden aus hochaufgelösten HDR-Aufnahmen mit

Abbildung 4.7: Detailausschnitte aus der Bürorekonstruktion mit exemplarisch markierten Texturrändern an scharfen Kanten.

langer Belichtungszeit ohne externe Beleuchtung erzeugt. Dadurch wird eine sehr kontrastreiche Farbwiedergabe und gleichmäßige Ausleuchtung der Szene erreicht.

Einen Ausschnitt aus der texturierten Rekonstruktion dieses Datensatzes zeigt Abbildung 4.6. Wie im linken Bild zu sehen, werden die hintere Wand und der Fußboden als zusammenhängende Flächen erkannt. Die für diese Flächen berechneten Texturen sind in Abbildung 4.8 gezeigt. Darüber hinaus werden dort noch beispielhaft weitere Texturen für andere Objekte in der Szene dargestellt. Bei der Rekonstruktion wurde die Auflösung der Texturen möglichst hoch gewählt. Man kann deutlich erkennen, dass die Rekonstruktion durch die Texturierung erheblich realistischer wirkt. Im Gegensatz zur Eingangspunktwolke kann man im Viewer der Las-Vegas-Software in der texturierten Rekonstruktion flüssig durch die Szene navigieren.

Der vorliegende Datensatz zeichnet sich durch eine sehr schwankende Punktdichte aus. In einigen runden Bereichen um die Scannerposition war - entgegen der Erwartung - die Punktdichte besonders hoch. Dadurch wird in diesen Bereichen eine sehr scharfe, fast fotorealistische Wiedergabe der Umgebung erreicht. Diese Bereiche lassen sich auf der Bodentextur deutlich ausmachen. Texturen, die für Bereiche mit weniger Messpunkten berechnet wurden, wirken leicht unscharf

Abbildung 4.8: Beispiele für berechnete Texturen im Büroscan für die hintere Wand (oben, Originalgröße 16384 × 8192 Pixel), Rollkästen und Monitore (linke Spalte, 2048 × 2048 bzw. 2048 × 512 Pixel im Original) und Fußboden (rechte Spalte, 4096 × 4096).

und verrauscht, was der Farbzuordnung durch die Nächste-Nachbar-Suche bei der Berechnung der Texturpixel zuzuschreiben ist. Der Effekt, dass die Farben der Ränder von nicht genutzten Bereichen in diese hineinwachsen, ist am Beispiel der Tür an der hinteren Wand der Szene sehr gut zu erkennen. Für Wand und Fußboden mussten in diesem Datensatz sehr große Texturen (16384×8192, bzw. 4096×4096 Pixel) erzeugt werden.

Weitere Detailausschnitte aus der Rekonstruktion sind in Abbildung 4.7 gezeigt. Um die Positionierung der Texturen in der Rekonstruktion zu beurteilen, wurden Blickwinkel auf scharfe Kanten gewählt, an denen Versätze deutlich auffallen. Durch Vergleich mit aus derselben Perspektive aufgenommenen Renderings der nicht-texturierten Rekonstruktion wurden einige Kanten in die texturierten Bilder übertragen. Diese sind als rot-getrichelte Linien in den Bildern zu sehen. Man kann deutlich erkennen, dass die Ränder der Texturen sehr gut mit der Geometrie der Rekonstruktion korrespondieren. Diese Bilder verdeutlichen zudem, dass sich durch Verwendung von Texturen sehr viele Details, die durch die geometrische Rekonstruktion nicht erfasst werden, in der Rekonstruktion darstellen lassen. Neben Verzierungen wie Poster oder Bilder an den Wänden kann es sich dabei auch um potenziell für Roboter relevante Informationen wie die Position von Lichtschaltern oder Steckdosen handeln (siehe oberes linkes Bild in Abbildung 4.7). Durch die Kombination von eingefärbten Dreiecken mit Texturen können in der visuellen Darstellung sogar einige Fehler, die durch die Meshoptimierung entstanden sind, ausgeglichen werden. So wirkt die Rückenlehne des Schreibtischstuhls, deren Rundung beim Region-Growing entfernt wurde, in der texturierten Rekonstruktion wesentlich plastischer als im einfarbigen Mesh.

Wie oben erwähnt sind die für diesen Datensatz erzeugten Texturen sehr groß. Da die Größe von Texturen je nach verwendeter Grafikkarte begrenzt ist, kann es bei so großen Flächen zu Problemen beim Rendern kommen. Dies kann durch Beschränkung der Größe der Texturen oder durch Wahl einer geringeren Auflösung vermieden werden. Beispiele, welchen Einfluss die gewählte Auflösung auf die Qualität der Texturen hat, werden in Abschnitt 4.2.2 besprochen.

Der zweite Datensatz wird auf der Webseite der Firma *TerraDat Geomatics* [186] zur Verfügung gestellt. Er wurde mit einem Topcon GLS-1000-Laserscanner mit kalibrierter Kamera in Castle Coch in der Nähe von Cardiff (UK) aufgenommen. Abbildung 4.9 zeigt die texturierte Rekonstruktion dieses Datensatzes mit Detailansichten. Im oberen linken Teilauschnitt sieht man eine deutliche Schattenlinie entlang der Turmmauer. Diese ist darauf zurückzuführen, dass die Rundung des Turmes durch die Ebenenoptimierung segmentiert wurde und deshalb durch Beleuchtungseffekte beim Rendern eine deutlich zu sehende Abschattung entsteht. Für die einzelnen Segmente wurden jeweils eigene Texturen angelegt. Betrachtet man den Verlauf der Ziegelsteine an der Mauer, sieht man allerdings deutlich, dass die einzelnen Texturen sehr gut ineinander übergehen. Abgesehen von der Beleuchtung ist die Begrenzung der einzelnen Polygone kaum zu erkennen.

Im unteren linken Bildbereich sieht man, dass sich ein Gebiet um das Burgtor farblich deutlich vom Rest der Rekonstruktion hervorhebt. Diese unpassende Einfärbung ist allerdings bereits in den Eingangsdaten vorhanden und ist wahrscheinlich auf eine ungünstige Beleuchtung des entsprechenden Bildausschnittes bei der Aufnahme der zum Einfärben benutzten Bilddaten zurückzuführen.

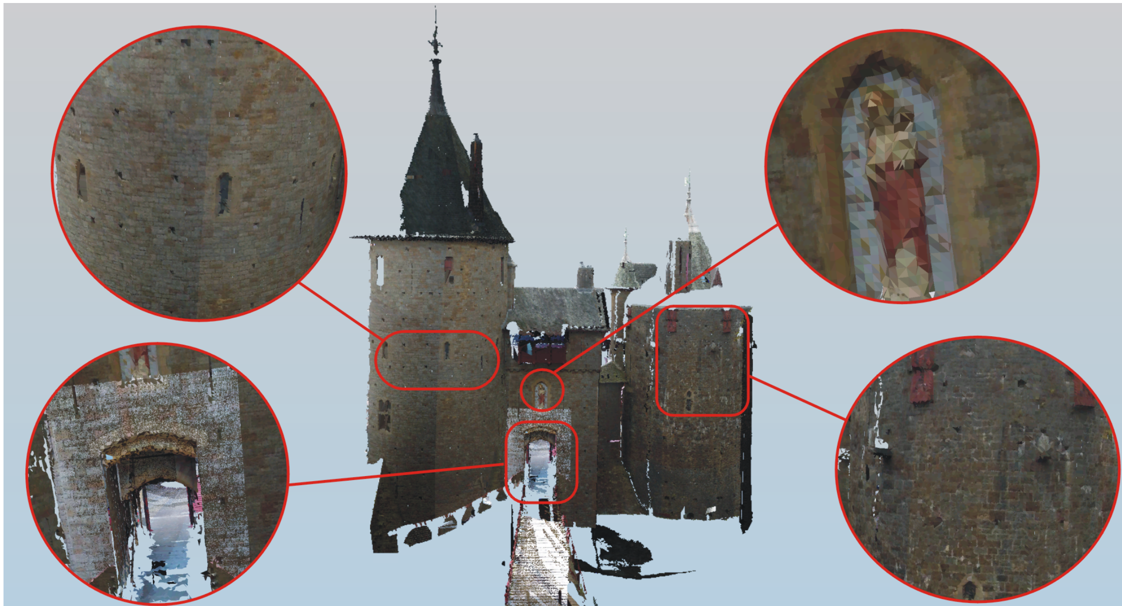


Abbildung 4.9: Texturierung des Schloss-Datensatzes

Die Bilder auf der rechten Seite verdeutlichen noch einmal die guten Ergebnisse beim Aufeinandertreffen von texturierten Regionen und einfarbig eingefärbten Dreiecken, insbesondere bei der Figur über dem Eingangstor (oberer Ausschnitt). Hier werden die einzelnen Dreiecke, die die Oberfläche der Statue repräsentieren, gleichmäßig gemäß der Punktwolke eingefärbt. Da hier zur Oberflächenmodellierung sehr viele kleine Dreiecke erzeugt wurden, ergibt sich so eine im Vergleich zum einfarbigen Modell deutlich realistischere Wiedergabe. Im oberen Abschnitt der Nische ist der fließende Übergang zwischen texturierten und einfarbigen Flächen zu erkennen. Der untere Ausschnitt zeigt diesen Effekt beim Übergang zwischen Mauern und Fensterläden.

Im dritten Datensatz wurden Farbinformationen von Kinect-Daten auf die Punktwolke eines rotierenden SICK-Laserscanners gemappt. Dazu wurden jeweils Kinect- und Rotationseinheitsdaten registriert. Da die Laserscannerdaten eine höhere geometrische Genauigkeit als die Kinect-Daten haben, wurden die Farbwerte aus der Kinect-Punktwolke mittels Nächste-Nachbar-Suche auf die Laserpunktwolke übertragen. Die so entstandene Einfärbung weist selbstverständlich nicht die hohe Qualität eines kalibrierten terrestrischen Scanners auf, dennoch lässt sich auch mit dieser Methode eine halbwegs realistische Farbwiedergabe erreichen.

Die Rekonstruktion aus einem mit diesen Verfahren aufgenommenen Datensatzes zeigt das linke Bild in Abbildung 4.10. Das rechte Bild zeigt das erzeugte texturierte Mesh. Aufgrund der geringen Punktdichte und des fehlerbehafteten Mappings der Eingangsdaten wirken die Texturen sehr verwaschen, dennoch stimmt die Einfärbung grob mit der Realität überein. An der Wand lassen sich sogar verschwommen Poster und Kalender erkennen. Weite Bereiche der hinteren Wand des Raumes und der Decke sind von einem dunklen Schatten bedeckt. Dieser Effekt kommt dadurch zustande, dass die Kinectbilder von unterschiedlichen Positionen aufgenommen wurden, an denen sich die Beleuchtung durch einfallendes Licht signifikant unterschied.

Abbildung 4.10: Texturierung des Kinect-Datensatzes

Abbildung 4.11: Einfluss der Pixelgröße auf die Qualität der generierten Texturen. Eingestellte Werte von links nach rechts: 20 mm, 5 mm und 1 mm

4.2.2 Qualität der Texturen

Die Qualität der Texturen, die aus den Punktwolken erzeugt werden, hängt wesentlich von der eingestellten Auflösung, also der Größe eines Pixels innerhalb der zu texturierenden Region ab. Je kleiner die eingestellte Pixelgröße ist, um so mehr werden dementsprechend erzeugt. Zu kleine Pixelgrößen sind allerdings aus mehreren Gründen nicht sinnvoll. Auf der einen Seite steigt zwar die Qualität der Umgebungswiedergabe an, allerdings wird auch mehr Speicher benötigt. Zudem ist es nicht sinvoll, die Pixelgröße viel kleiner als die maximale Punktdichte der Punktwolke einzustellen, da durch die Nächste-Nachbar-Suche entsprechend viele Pixel zwischen den Punkten interpoliert werden, was in etwa denselben Effekt bewirkt wie die Skalierung geringer aufgelöster Texturen. Eine Verbesserung der Qualität wird unterhalb dieser Grenze nicht mehr erreicht. Dieser Effekt ist in Abbildung 4.11 zu sehen. Die Bilder zeigen Detailausschnitte, die aus Texturen

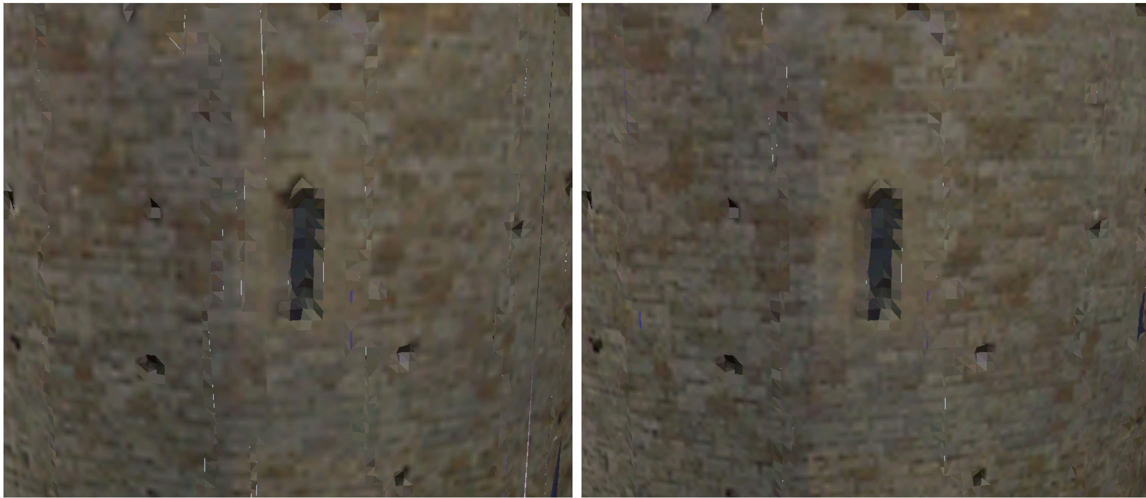


Abbildung 4.12: Vergleich der Renderingqualität bei Halbierung der Auflösung (10 mm links und 5 mm rechts)

des Schloss-Datensatzes entnommen wurden. Um die Qualität der unterschiedlich aufgelösten Texturen vergleichen zu können, wurden sie auf die gleiche Größe skaliert. Dies entspricht in etwa, abgesehen von eventuell aktivierten Texturfiltern, der Art und Weise, wie die Texturen auch beim echten Rendering auf ein Dreieck gleicher Größe projiziert würden.

Die eingestellten Pixelgrößen betrugen von links nach rechts 20 mm, 5 mm und 1 mm. An den Bildern kann man den gerade geschilderten Effekt erkennen. Die mittlere Textur zeigt deutlich mehr Details als die linke. Das rechte Bild unterscheidet sich, obwohl die Auflösung fünf mal so hoch ist, optisch kaum von der mittleren Darstellung. Da die Punktdichte nach Angaben des Herstellers im vorliegenden Datensatz im Bereich zwischen zwei und vier Millimetern liegt [186], ist es nicht weiter verwunderlich, dass die optische Qualität unterhalb der in der Mitte gezeigten 5 mm-Auflösung nicht mehr nennenswert ansteigt. Bei der Rekonstruktion sollte die Pixelgröße daher entsprechend der vorhandenen Punktdichte eingestellt werden, wenn eine optimale Texturqualität erreicht werden soll.

Der Einfluss der Auflösung auf den benötigten Speicher ist in Tabelle 4.4 für die oben gezeigte Textur dargestellt. Die Anzahl der effektiv genutzten Pixel pro Textur modulo Rundungseffekte steigt quadratisch. Die zur Erzeugung benötigte Zeit steigt mit der Auflösung überproportional an. Der Sprung im benötigten Speicherplatz zwischen 2.5 mm und 1.0 mm lässt sich dadurch erklären, dass hier lediglich die effektiv genutzte Texturgröße angegeben wurde. Da Dimensionen der Textur jeweils auf die nächste Zweierpotenz erweitert werden, kann die intern tatsächlich erzeugte Bitmap deutlich größer sein als der effektiv genutzte Bereich. Die hier gezeigten Beispiele sollen demonstrieren, dass große Auflösungen unterhalb einer gewissen Grenze keine Verbesserung der Qualität der Rekonstruktion hervorrufen, aber überproportional mehr Ressourcen benötigen. Bedenkt man die Tatsache, dass derzeit gebräuchliche Grafikkarten Grafikspeicherim Bereich von 2 GB haben und zudem die maximal zulässige Texturgröße beschränkt ist, sollte bei der Wahl der Pixelgröße ein Kompromiss zwischen Auflösung und Bildgröße gewählt werden.

Tabelle 4.4: Kenngrößen einer Textur des Schlossscans bei Variation der Texturauflösung. Die jeweils eingestellten Pixelgrößen sind in mm angegeben. Breite und Höhe geben den effektiv genutzten Bereich der Textur in Pixeln wieder. Weiterhin sind die zur Erstellung aller Texturen benötigte Zeit und die Größe der erzeugten Textur im Speicher angegeben.

Pixelgröße	Breite	Höhe	Zeit	Bildgröße
1	8525	5112	1449 s	384 MB
2.5	3423	2045	248 s	24 MB
5	1710	1034	71 s	12 MB
10	851	518	26 s	768 kb
20	428	260	13 s	384 kb

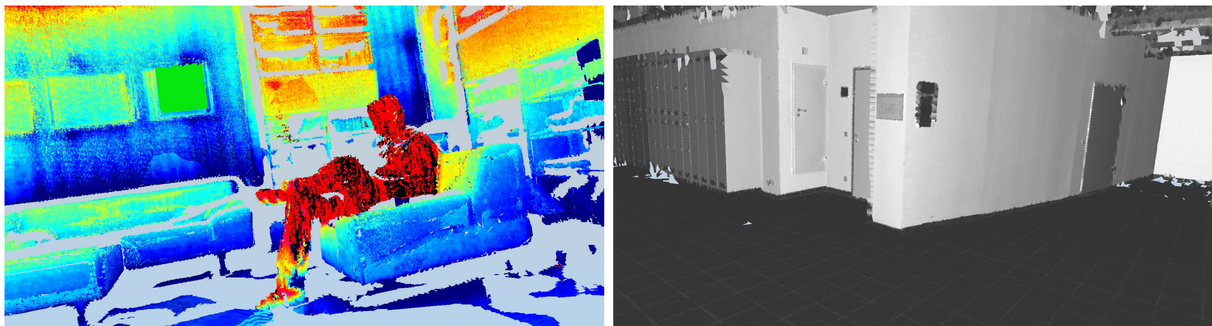


Abbildung 4.13: Beispiele für weitere Anwendungen für Texturen bei der Rekonstruktion: Visualisierung von Thermodaten (links) und Remissionswerten eines Laserscanners (rechts).

Sollen die Rekonstruktionen lediglich der Visualisierung dienen, reichen in der Praxis aufgrund der beim Rendern von Texturen angewandten Filter bereits relativ gering aufgelöste Texturen aus, um eine realistische Wiedergabe der Umgebung zu erreichen, wie in Abbildung 4.12 gezeigt wird. Das links abgebildete Modell wirkt in der Darstellung etwas verwaschener als das rechte, die wesentlichen Details sind aber erkennbar, obwohl durch die Halbierung der Auflösung wesentlich weniger Ressourcen benötigt werden.

4.2.3 Weitere Anwendungen für Texturen

Neben Farbinformationen aus Punktwolken lassen sich mit Hilfe von Texturen auch weitere mit den Punkten assoziierte Informationen zur Visualisierung aufbereiten. Zwei Beispiele dafür sind in Abbildung 4.13 gezeigt. Im linken Bild wurden die Punkte eines Riegl-Laserscanners gegen die Temperaturwerte einer Thermokamera gemappt [27]. Die von der Kamera für jeden Punkt gelieferten Rohdaten wurden in einem passend gewählten Falschfarbengradienten umgerechnet. Die so berechneten Farben wurden dann auf entsprechende Texturen bzw. die Dreiecke in der Rekonstruktion übertragen. Man kann in diesem Bild sehr intuitiv die warmen Bereiche (Mensch auf dem Sessel, obere Wandbereiche) von den kalten Regionen im Mesh unterscheiden.

Ähnlich wie die Thermodaten wurden im rechten Bild die Remissionswerte des Laserscanners als Grauwerte auf die Rekonstruktion übertragen. Da verschiedene Materialien unterschiedliche Reflektionseigenschaften haben, lassen sich mit diesen Daten in den Texturen Details in den Rekonstruktionen wiedergeben, die nicht durch die Geometrie repräsentiert werden. Beispiele sind die Schilder an der Wand, die Kachelung des Fußbodens und Zugangstür hinter den Schließfächern. Prinzipiell lassen sich mit den hier vorgestellten Verfahren beliebige Punkt-Attribute in Texturen übersetzen, sofern ein geeignetes Mapping auf RGB-Werte gefunden wird.

4.2.4 Fazit automatische Texturerstellung

Mit dem hier vorgestellten Verfahren zur Texturierung und Einfärbung von Dreiecksnetzen lassen sich je nach Qualität der Eingangsdaten sehr realistische Umgebungsrepräsentationen erstellen. Ob die dabei erreichte Qualität ausreicht, um die Texturinformationen für Lokalisierung á la Visual Slam [40, 58] einzusetzen, ist fraglich. Das Verfahren ist für die Robotik allerdings immer dann interessant, wenn ein Roboter auf Basis der von ihm aufgenommenen Daten teleoperiert wird. Die hier erzeugten 3D-Rekonstruktionen vermitteln einem Operator einen deutlich intuitiveren Eindruck der Umgebung als z.B. reine Kamera- oder Laserscannerdaten. Die Auflösung der Texturen kann je nach Anwendungsfall variiert werden, so dass ein passender Kompromiss zwischen Datenreduktion und Auflösung erreicht wird. Die Evaluation hat allerdings gezeigt, dass die maximal erreichbare Auflösung durch die Punktdichte der Eingangsdaten begrenzt ist. Eine Verbesserung der Auflösung ließe sich durch die Verwendung von hochaufgelösten Kamerabildern erreichen. Dazu müsste bei der Rekonstruktion allerdings die jeweilige Kalibrierung zwischen verwendeter Kamera und Scanner bekannt sein.

Der größte Nachteil dieses Verfahrens ist, dass jeweils pro Polygon eine Textur erzeugt wird. In großräumigen planaren Umgebungen kommen so auch mit groben Auflösungen sehr große Texturen zu Stande, die die Speicherkapazitäten handelsüblicher Grafikkarten sprengen. Ein derzeit verfolgter Ansatz, dieses Problem zu lösen, ist, Umgebungen mit sich wiederholenden Texturmustern zu approximieren. So ließen sich die Mauersteine des Schloss-Datensatzes auch durch ein kleines, sich wiederholendes Endlosmuster darstellen, ohne dass sich die visuelle Wahrnehmung bei der Betrachtung des Datensatzes stark ändert. Solche Pattern-Texturen sind Standard im Bereich der Computerspiele und erlauben dort texturiertes Rendering von sehr großen Umgebungen. Die Herausforderung besteht darin, solche regulären Muster aus den Eingangsdaten zu erzeugen oder bereits vorhandene vorberechnete Muster passend auf die Rekonstruktionen zu übertragen. Erste vielversprechende Ergebnisse zeigt [159].

4.3 Laufzeitanalyse und Speicherverbrauch

4.3.1 Laufzeitverhalten der einzelnen Komponenten

Das Laufzeitverhalten der einzelnen Schritte der Rekonstruktion wurde an einer Auswahl an Datensätzen von verschiedenen Sensoren untersucht. Da die Texturierung sehr lange braucht und

Tabelle 4.5: Kennzahlen der zur Laufzeitanalyse verwendeten Datensätze. Die letzten beiden Spalten geben die Anzahl der Dreiecke vor und nach der Meshoptimierung an.

Datensatz	Punkte	Auflösung	Zellen	Dimensionen [m]	Dreiecke	Dreiecke Opt
Scan 1	125 514	3 cm	39 251	2.89 × 1.68 × 2.32	58 032	25 365
Scan 2	281 227	5 cm	185 295	19.75 × 3.94 × 12.64	106 336	50 784
Scan 3	289 252	5 cm	120 140	23.38 × 4.07 × 8.06	123 905	20 626
Scan 4	292 301	5 cm	62 221	6.19 × 3.75 × 7.37	67 514	20 967
Kinect 1	234 980	3 cm	42 593	4.07 × 0.93 × 4.52	65 125	18 648
Kinect 2	255 942	3 cm	16 032	2.36 × 1.29 × 0.98	21 261	12 690
Kinect 3	240 752	3 cm	24 203	2.49 × 1.31 × 2.23	33 577	23 124
Büro	2 234 980	5 cm	444 881	11.09 × 12.17 × 7.35	415 039	199 504
Kirche	10 255 942	4 cm	429 467	7.13 × 13.77 × 2.94	130 090	66 223
Schloss	15 240 752	20 cm	633 520	196.49 × 154.99 × 43.79	649 188	257 227

optinal ist, werden hier nur die Zeiten Rekonstruktion der Geometrie betrachtet. Dazu werden Laserscans von nickenden SICK-Laserscannern, einzelne Kinect-Punktwolken und hochaufgelöste Laserscans (Kirchen-, Büro- und Schlossdatensatz) als Eingangsdaten verwendet. Die Parameter zur Rekonstruktion wurden dabei wieder so eingestellt, dass eine möglichst hochwertige Rekonstruktion erzielt wurde. Die Kennzahlen dieser Rekonstruktionen sind in Tabelle 4.5 zusammengefasst. Die Datensätze aus den terrestrischen Laserscans erfassen ein deutlich größeres Volumen als die SICK- und Kinect-Scans. Sie wurden absichtlich ausgewählt, um zu demonstrieren, dass sich auch größere Szenen mit der Software performant rekonstruieren lassen. Die Laufzeiten der einzelnen Schritte der Rekonstruktion zeigt Abbildung 4.14.

Die benötigtem Zeiten der Rekonstruktion für die 3D-Laserscans aus den nickenden SICK-Laserscannern bewegten sich im Bereich von 0.8 bis 1.5 Sekunden pro Datensatz. Die Rekonstruktion aus Kinect-Daten benötigte bei den gewählten Auflösungen zwischen 1.3 und 1.5 Sekunden pro Frame. Die hochaufgelösten Datensätze konnten in unter einer Minute rekonstruiert werden. Dazu kommt jeweils die benötigte Zeit zur Meshoptimierung, so dass sich Gesamtlaufzeiten von 1.3 bis 5 Sekunden für Laserscans, 1.3 bis 2 Sekunden für Kinect-Frames und 62 bis 72 Sekunden für hochauflösende Laserscans ergaben. Bis auf den zweiten Scan benötigte die Normalenschätzung die meiste Zeit. In diesem Fall war die benötigte Zeit zur Meshoptimierung beinahe doppelt so lang wie die Zeit, die zur Mesherzeugung inklusive Normalenschätzung benötigt wurde. In den anderen Fällen benötigten die Meshoptimierungsschritte in etwa so viel Laufzeit wie die eigentliche Rekonstruktion.

Auffällig ist, dass bei den kleineren Datensätzen die Auswertung der Distanzfunktion relativ wenig Zeit in Anspruch nimmt, da die aufgenommenen Umgebungen eher klein waren und daher nur relativ wenige Voxel erzeugt werden mussten. Dementsprechend korrelieren auch die Zeiten für Gittererstellung und Marching Cubes: Wenn viele Zellen generiert werden, müssen potentiell auch viele Dreiecke berechnet werden. Bei den hochaufgelösten Datensätzen ist die benötigte

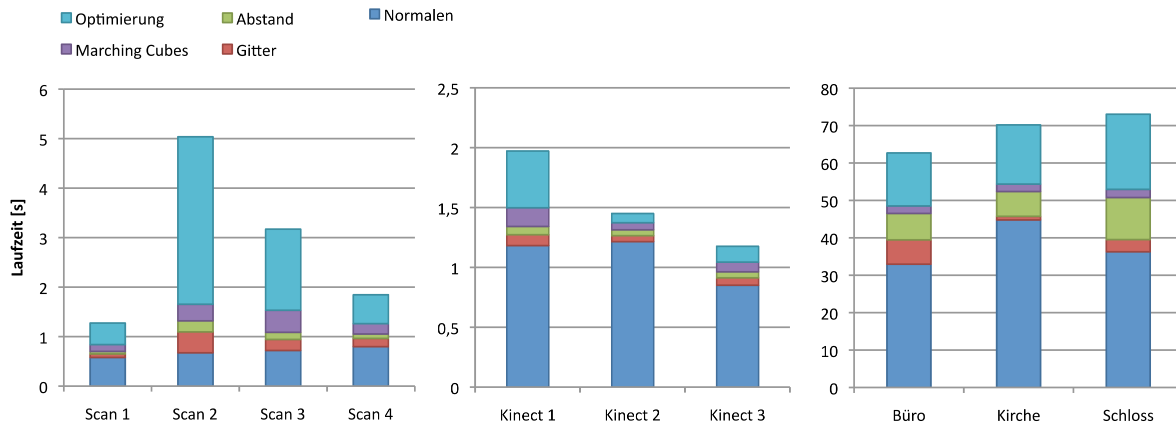


Abbildung 4.14: Laufzeit der einzelnen Schritte bei der Rekonstruktion. Gezeigt sind die Laufzeiten für Normalenberechnung (dunkelblau), Berechnung des Rekonstruktionsgitters (rot), Auswertung der Distanzfunktion (grün), Rekonstruktion mit Planar Marching Cubes (violett) und Meshoptimierung (hellblau).

Zeit zur Auswertung der Distanzfuktion vergleichsweise lang. Dies liegt daran, dass in großen Datensätzen auch die Laufzeit für die Nächste-Nachbar-Suche ansteigt und diese daher im Verhältnis zu den anderen Schritten der Rekonstruktion, die ohne solche Anfragen auskommen, verhältnismäßig mehr Zeit benötigt. Insgesamt brauchen Auswertung der Distanzfuktion und Normalenberechnung in allen Fällen die meiste Rechenzeit.

Die Laufzeit der Meshoptmierungsschritte variiert zwischen den einzelnen Datensätzen. Die gemessenen Werte sind in Abbildung 4.15 dargestellt. In allen Fällen benötigen das Löschen der kleinen Regionen und das Schließen von Löchern die meiste Zeit. Wie viel Zeit für diese Schritte benötigt wird, variiert von Fall zu Fall. So werden bei Scan 1 und Scan 4 für diese Schritte lediglich 250 bzw. 700 Millisekunden benötigt, während sie bei Scan 2 und Scan 3 mehre Sekunden in Anspruch nehmen. Um diesen Effekt zu verstehen, muss man die Struktur der untersuchten Umgebungen bedenken. Scan 2 und Scan 3 deckten eine vergleichsweise große Szene ab. Dementsprechend gab es durch das Ausdünnen der Punktwolken an den Rändern in diesen Bereichen viele Löcher, die aufgefüllt werden mussten. Zudem steigt das Messrauschen mit der Entfernung, so dass die Normalenschätzung unpräziser und inkonsistenter wird, was dazu führt, dass in solchen Gebieten das Region Growing oftmals vorzeitig terminiert, was zu einer steigenden Anzahl von kleinen Regionen führt, die entfernt werden, wodurch wiederum die Anzahl der zu schließenden Löcher noch weiter ansteigt.

Das Region-Growing mit Projektion der Vertices in die Ausgleichsebenen der gefundenen Cluster benötigt in allen Fällen vergleichsweise wenig Zeit und ist in den Gruppen der verschiedenen Eingangsdaten relativ konstant. Die Laufzeit der Schnittkantenoptimierung hängt wiederum von der Struktur der Umgebung ab. Werden viele sich schneidende Ebenen gefunden, müssen entsprechend viele Vernschneidungen berechnet werden. Aus diesem Grund ist die Laufzeit dieses Schrittes im Kirchendatensatz besonders hoch. Dieser besteht aus vielen ebenen Flächen, die sich häufig schneiden, z.B. zwischen den Sockeln der Säulen und dem Fußboden und an den

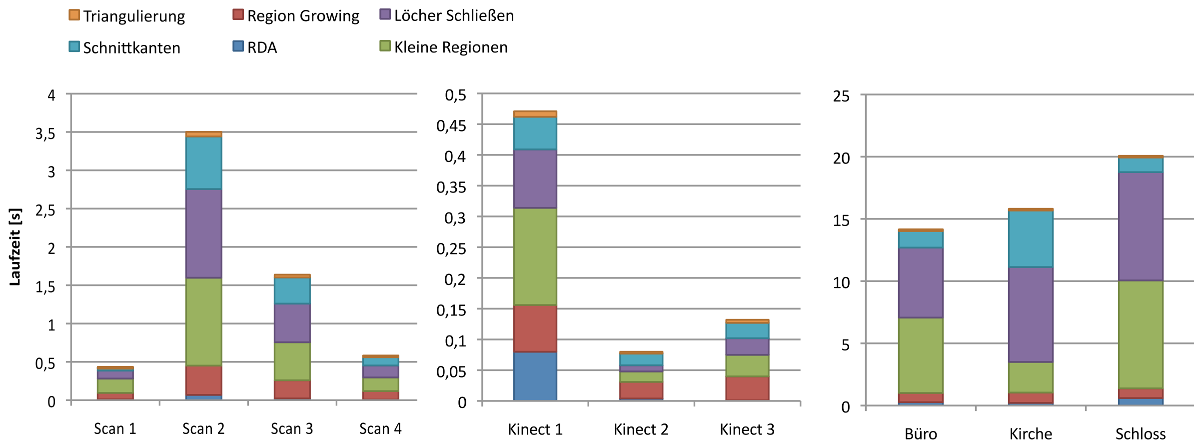


Abbildung 4.15: Laufzeit der Verfahren zur Meshoptimierung: Entfernen von frei schwebenden Artefakten (RDA, dunkelblau), Löschen kleiner Regionen (grün), Schließen von Löchern (violett), Region Growing mit Optimierung der Vertices (rot), Optimierung der Schnittkanten von Ebenen (hellblau) und Neutriangulierung.

Durchgängen zwischen den verschiedenen Räumen. Büro- und Schlossdatensatz enthalten zwar auch viele Ebenen, hier gibt es allerdings viele Ebenen, die sich gerade nicht schneiden, z.B. die Tischplatten und Regaleinlegeböden. Da annähernd parallele Flächen bei der Schnittoptimierung heraus gefiltert werden, werden diese bei der Berechnung der Schnittkanten auch nicht berücksichtigt, wodurch die Anzahl der möglichen Schnitte sinkt.

Verglichen mit der Rekonstruktion sind die Laufzeiten der Meshoptimierungsschritte relativ hoch, weil sich diese Verfahren schlecht parallelisieren lassen. Das Region-Growing könnte prinzipiell ausgehend von mehreren Seed-Punkten parallelisiert werden, allerdings müsste durch eine entsprechende Verwaltung der gerade laufenden Prozesse sichergestellt werden, dass Rekursionen, die ineinander laufen, passend fusioniert werden. Ähnliche Probleme ergeben sich beim Schließen von Löchern. Hier müsste durch entsprechende Locking-Konstrukte sichergestellt werden, dass nicht gerade zwei Prozesse gleichzeitig auf eine Kante im Netz zugreifen. Insgesamt ergeben sich bei solchen Prozessen in der Praxis sehr viele Detailprobleme, z.B. durch die Berücksichtigung von topologischen Sonderfällen, so dass auf eine parallelisierte Implementierung bisher verzichtet wurde.

In den oben gezeigten Versuchen wurde ausschließlich die Rekonstruktion mit Planar Marching Cubes bei unterschiedlichen Gitterauflösungen betrachtet, das dieses Verfahren die qualitativ besten Ergebnisse liefert. Das Laufzeitverhalten der anderen implementierten Marching-Cubes-Varianten bei Veränderung der Voxelgröße ist in Abbildung 4.16 für einen SICK-Laserscan und einen Kinect-Frame dargestellt. Wie zu erwarten, ist Standard Marching Cubes die schnellste Variante, da keine Nachoptimierungen wie bei Planar Marching Cubes oder Extended Marching Cubes durchgeführt werden müssen, wobei Extended Marching Cubes immer das langsamste Verfahren ist. Planar Marching Cubes ist nur unwesentlich langsamer als Standard Marching Cubes. Marching Tetrahedrons schneidet in den Versuchen immer am zweitschlechtesten ab, was

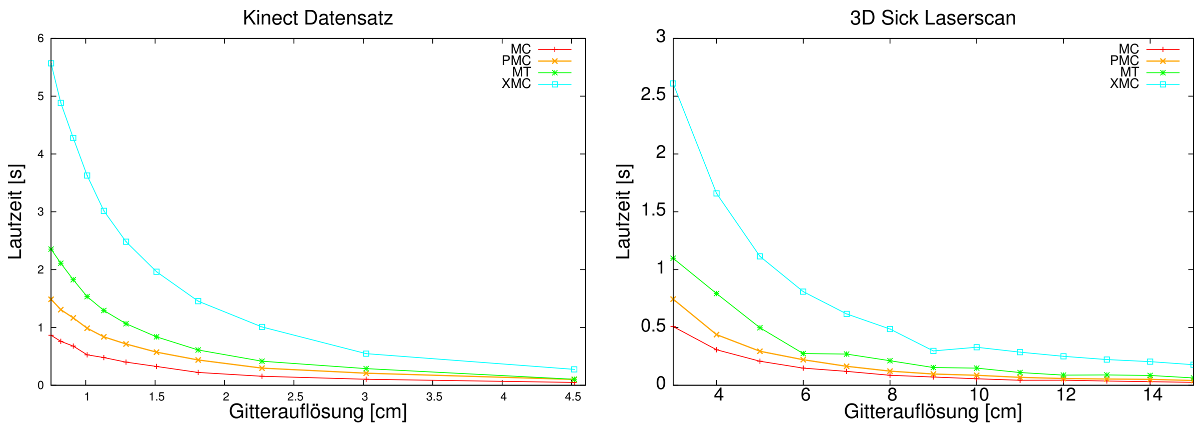


Abbildung 4.16: Laufzeiten der implementierten Marching-Cubes-Varianten

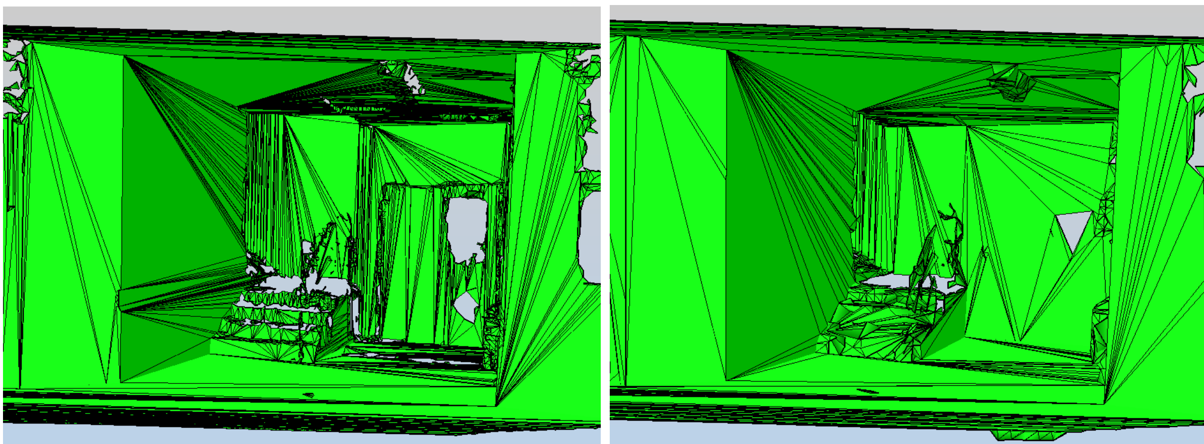


Abbildung 4.17: Rekonstruktion eines SICK-Laserscans mit 5 cm bzw. 10 cm Gitterauflösung.

darauf zurückzuführen ist, dass wesentlich mehr Dreiecke berechnet werden müssen als bei den anderen Verfahren. Die Grafik verdeutlicht darüber hinaus, dass die benötigte Zeit mit höherer Auflösung drastisch sinkt.

Die Zeiten in Abbildung 4.14 wurden mit der maximalen noch sinnvollen Auflösung gemessen. Sie stellen also praktisch ein Worst-Case-Scenario dar. Für praktische Anwendungen können je nach Anwendungsfall auch gröbere Auflösungen verwendet werden, wodurch sich die Laufzeit noch weiter verringern lässt. Für Rekonstruktionen aus SICK-Laserscans lassen sich auch mit einer Gitterauflösung von 10 cm Karten erstellen, die den in Kapitel 3.1 gestellten Anforderungen entsprechen. Der Unterschied in der Rekonstruktionsqualität für 5 bzw. 10 cm Voxelgröße bei der Rekonstruktion mit Planar Marching Cubes ist in Abbildung 4.17 gezeigt. Die allgemeine Geometrie unterscheidet sich auf den ersten Blick kaum. Lediglich relativ feine Konturen wie die der Fenster im Hintergrund werden stark vereinfacht wiedergegeben. Mittelwert und Standardabweichung des Punktabstandes zwischen Punktwolke und optimiertem Mesh betrugen 0.41 cm und 1.86 cm bei einer Voxelgröße von 5 cm , bzw. 0.73 cm und 2.03 cm für 10 cm Gitterauflösung.

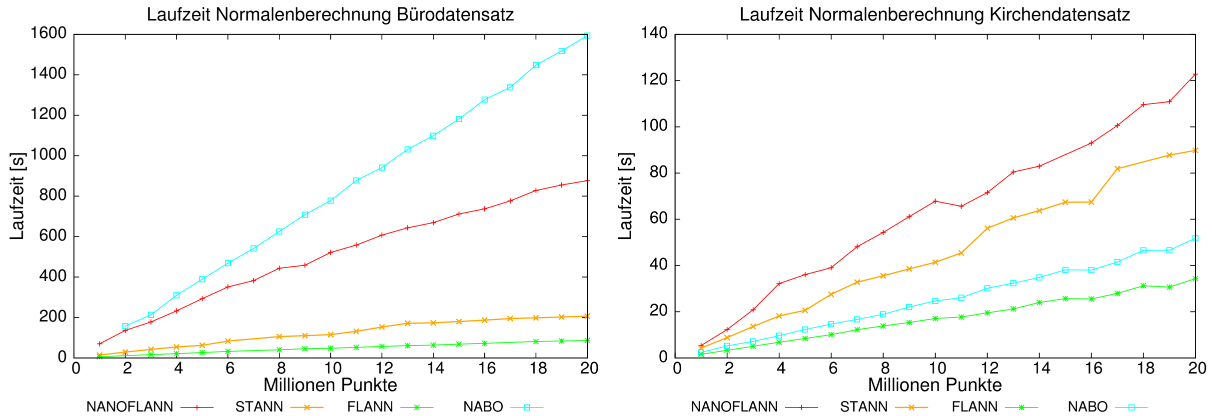


Abbildung 4.18: Laufzeitverhalten verschiedener knn-Suchbibliotheken bei Variation der Punktezahl am Beispiel von zwei hochaufgelösten Laserscans.

4.3.2 Evaluation Nächste-Nachbar-Suche

Wie im vorherigen Abschnitt gezeigt wurde, benötigen die Normalenschätzung und die Auswertung der Distanzfunktion die meiste Laufzeit bei der Rekonstruktion. Diese Schritte erfordern jeweils Nächste-Nachbar-Suchen. Daher muss die dazu verwendete Bibliothek in der Lage sein, diese Aufgabe performant zu lösen. In der Software können dazu die Bibliotheken STANN [37, 113], FLANN [126, 127] NANOFlann [132] und Nabo [115] verwendet werden. Andere Nächste-Nachbar-Implementierungen wie ANN [10, 124] wurden nicht evaluiert, da sie entweder nicht threadsicher sind oder für Spezialanwendungen wie ICP [140, 141] optimiert wurden und daher lediglich einen nächsten Nachbarn suchen können. Sie erfüllen die hier benötigen Anforderungen somit nicht.

Eine ausführliche Evaluation einiger der hier verwendeten Bibliotheken findet sich in [49]. Dort wurde das Laufzeitverhalten der im 3DTK implementierten Nächste-Nachbar-Suche sowie der in STANN, FLANN, NANOFLANN und NABO vorhandenen Funktionen im Kontext von ICP-basiertem Scanmatching untersucht. Da hier jedoch ein anderer Anwendungskontext vorliegt und die verwendeten Bibliotheken seit den dort durchgeführten Experimenten weiter entwickelt wurden, soll das Laufzeitverhalten der verschiedenen Implementierungen an verschiedenen Datensätzen am Beispiel der Normalenschätzung exemplarisch untersucht werden.

Abbildung 4.18 zeigt die Laufzeiten der verschiedenen Verfahren bei Variation der Punktzahl. Dazu wurden zwei strukturell unterschiedliche Datenätze (Büroumgebung und Straßendatensatz aus Abbildung 4.2) mittels eines Zufallsfilters auf verschiedene Punktzahlen ausgedünnt. Auf diesen Daten wurde dann eine parallelisierte Normalenschätzung mit jeweils 100 Nachbarpunkten ausgeführt. Um die Ergebnisse vergleichbar zu machen, wurde die Abstandssortierung der gefundenen Nachbarpunkte nach Entfernung falls vorhanden abgeschaltet. Es ist gut zu erkennen, dass sich die Laufzeiten der einzelnen Suchverfahren merklich unterscheiden. In beiden Experimenten lieferte FLANN bei allen Punktdichten mit Abstand die schnellsten Ergebnisse. Die Rangfolge der anderen Bibliotheken ist jedoch unterschiedlich. So ist NANOFLANN in der

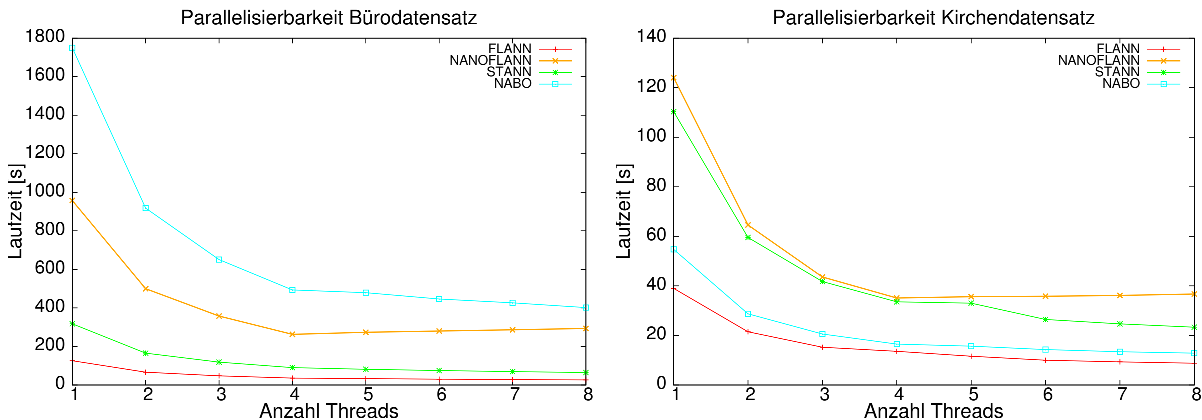


Abbildung 4.19: Parallelisierbarkeit der knn-Suche.

vorliegenden Version immer langsamer als STANN, das Laufzeitverhalten von NABO variiert allerdings in den beiden Datensätzen. In der Büroumgebung ist diese Suche mit Abstand die langsamste, im unstrukturierten Straßendatensatz ist NABO allerdings nur wenig langsamer als FLANN und liefert die zweitbesten Ergebnisse. Die Beobachtung, dass die Laufzeit von NABO bei verschiedenen Datensätzen stark variiert, wurde auch schon bei der Entwicklung der Software beobachtet. Eine Variation der Bucketgröße hatte keinen nennenswerten Einfluss auf die Laufzeit der Normalenschätzung. Ein Erklärungsversuch für dieses Verhalten ist, dass die internen Baumstukturen bei dieser Implementierung in ungünstigen Geometrien entarten und sich die Suchzeit so drastisch erhöht. Theoretisch ist insgesamt ein logarithmisches Laufzeiverhalten zu erwarten. Diese Tendenz ist bei den schnelleren Bibliotheken im linken Bild in Abbildung 4.18 bei höheren Punktzahlen qualitativ zu erkennen. Auf eine detaillierte Auswertung des asymptotischen Verhaltens soll hier verzichtet werden, da für diese Arbeit vorrangig das qualitative Verhalten der Bibliotheken evaluiert werden soll, um zu entscheiden, welches Suchverfahren in der Praxis die schnellsten Ergebnisse liefert.

Einige Ergebnisse aus [49] konnten anhand der betrachteten Beispiele nachvollzogen werden, nämlich dass NABO unter bestimmten Umständen sehr kurze Laufzeiten hat und STANN langsamer als FLANN ist. Die Tatsache, dass FLANN immer langsamer als NABO ist, konnte hier nicht nachvollzogen werden. In den hier durchgeführten Experimenten und bei der Arbeit mit anderen Datensätzen hat sich gezeigt, dass FLANN in der Regel die kürzeste Laufzeit hat. Allerdings gab es nach der Publizierung von [49] ein Update der Software, dass eine starke Verbesserung der Laufzeit zur Folge hatte. Die in [49] verwendete Version wurde daher hier nicht evaluiert. Insgesamt hat sich gezeigt, dass FLANN in der Praxis faktisch immer die kürzeste Laufzeit aufwies.

Für moderne Mehrkernprozessoren ist die Parallelisierbarkeit der Suche ein wichtiger Aspekt, um die von der Hardware zur Verfügung gestellte Rechenleistung optimal auszunutzen. Dazu muss die Suchbibliothek parallele Suchanfragen auf der erzeugten Baumstruktur erlauben. Die intern verwendeten Synchronisationsstrukturen können dabei potenziell zu Laufzeitverschlechterungen führen, zumal nicht jede Suchanfrage die gleiche Laufzeit haben muss. Aus diesem Grund

Tabelle 4.6: Relative Beschleunigung der Normalenschätzung bei 8 Threads gegenüber der nicht-parallelisierten Implementierung.

Datensatz	FLANN	STANN	nano FLANN	NABO
Kirche	4.50	4.77	3.35	4.26
Büro	4.78	4.67	3.25	4.44

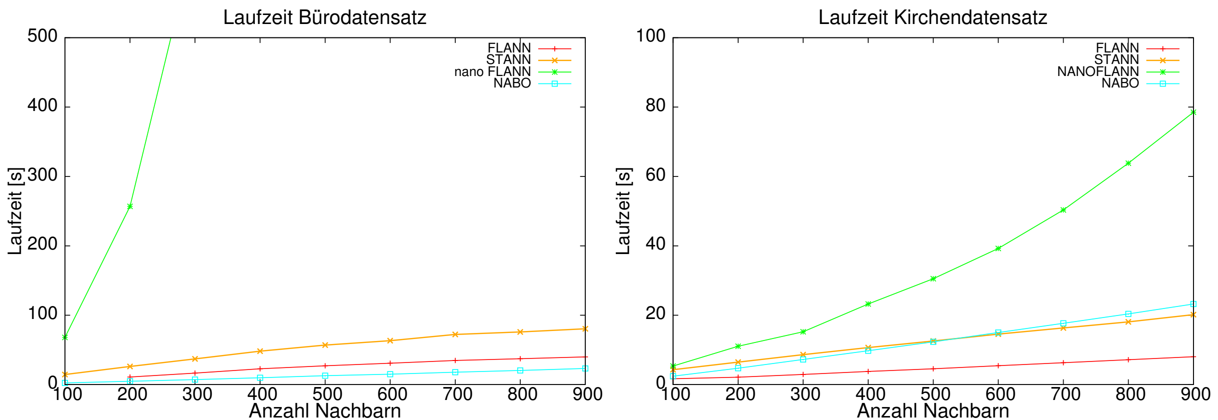


Abbildung 4.20: Laufzeitanalyse verschiedener knn-Bibliotheken bei Variation der Zahl der zu findenden Nachbarn.

wurde auch der Geschwindigkeitsgewinn bei der Nutzung mehrerer Threads untersucht. Das vorhandene Testsystem verfügte über einen Intel Core i7-Prozesser mit 4 Kernen und Hardware-Unterstützung für die parallele Ausführung von zwei Threads pro Kern ("Hyperthreading"), also insgesamt 8 Threads.

Im Experiment wurde die Anzahl der von OpenMP zur parallelen Abarbeitung der Normalenschätung verwendeten Threads variiert. Wie in Abbildung 4.19 zu sehen ist, skaliert die Rechenzeit bis 4 Threads wie erwartet. Die Laufzeit halbiert sich in etwa mit jedem hinzugefügten Thread. Der Zeitgewinn flacht danach deutlich ab, d.h. es wird keine wesentliche Verbesserung der Laufzeit mehr erzielt. Die maximal erreichten relativen Geschwindigkeitszuwächse zeigt Tabelle 4.6. Je nach Datensatz unterscheiden sich die Ergebnisse leicht. Tendenziell erreichen bis auf NANOFLANN alle Suchbibliotheken auf dem System mehr als die 4-fache Beschleunigung bezogen auf einen einzelnen Thread.

Wie in den Abschnitten zur Normalenberechnung und Distanzfunktion gezeigt, kann in verrauschten Datensätzen die Erhöhung der Anzahl der betrachteten Nachbarn für qualitativ bessere Ergebnisse sorgen. Wird die Anzahl der zu findenden Nachbarn bei einer Suchanfrage erhöht, steigt auch die Laufzeit. Das Verhalten der Bibliotheken bei Variation der Nachbarzahl ist in Abbildung 4.20 gezeigt. Die Laufzeit der Suche steigt dabei bei FLANN, STANN und NABO annähernd linear mit der Anzahl der zu suchenden Nachbarn. Bei NANOFLANN steigt die Suchzeit sehr viel schneller an. Die hier gezeigten Laufzeiten stellen allerdings so etwas wie

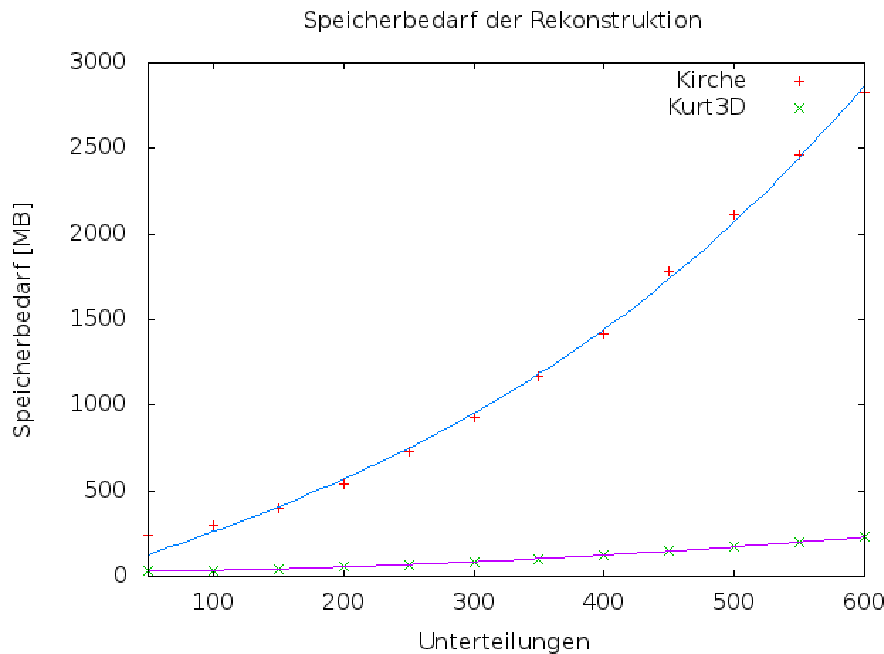


Abbildung 4.21: Speicherbedarf der Rekonstruktion für einen terrestrischen Scan und einen SICK Lasercan, der mit Kurt3D aufgenommen wurde. Die eingezeichnete Regressionskurven sind Polynome dritten Grades.

Worst-Case-Szenarios dar, da selten mehr als 50 Nachbarn benötigt werden, um ausreichend gute Rekonstruktionsergebnisse zu erzielen. Insgesamt lässt sich feststellen, dass auch in diesem Experiment FLANN die höchste Performanz zeigt.

4.3.3 Speicherbedarf

Wie viel Speicher für die Rekonstruktion benötigt wird, hängt im Wesentlichen von zwei Faktoren ab: der Größe des aus der Punktwolke generierten Suchbaums zur Nächste-Nachbar-Suche und der Zellengröße des zur Rekonstruktion verwendeten Gitters. Die Größe des Suchbaums hängt dabei nicht von Parametern der Rekonstruktion ab, sondern von der Anzahl der geladenen Punkte. Dasselbe gilt für den Speicher, der für die zu berechnenden Punktnormalen benötigt wird. Die Größe des Rekonstruktionsgitters hängt dagegen von der eingestellten Auflösung ab.

Andere Datenstrukturen im Rekonstruktions- und Optimierungsprozess benötigen vergleichsweise wenig Speicher. Nachdem die Halbkantendarstellung der rekonstruierten Oberfläche erstellt wurde, wird für die folgenden Optimierungsschritte kaum noch Arbeitsspeicher benötigt. Beim Region-Growing müssen lediglich die Konturkanten der Flächen gespeichert werden. Das Schließen der Löcher und die Konturoptimierung benötigen ebenfalls kaum Speicher. Bei der Erzeugung von Texturen muss lediglich ausreichend Speicher für die gerade behandelte Textur vorhanden sein. Daher soll hier vor allem der Einfluss der Gittergröße auf den Speicherbedarf diskutiert

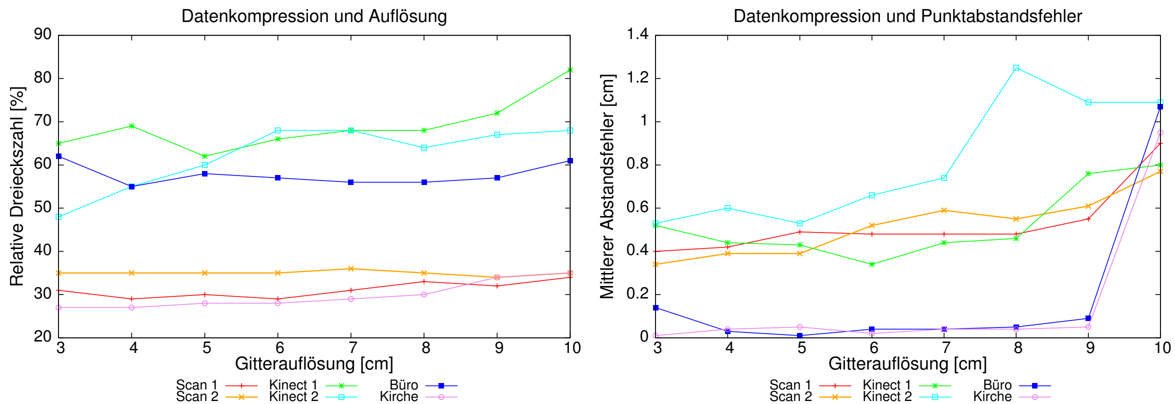


Abbildung 4.22: Datenkompression bei Variation der Voxelgröße für die verwendeten Testdatensätze (links) und mittelerer Punktabstandsfehler (rechts).

werden.

Die Anzahl der erstellten Gitterzellen wächst dabei kubisch mit der Auflösung. Die verwendete Hashing-Datenstruktur verhindert dabei zwar, dass Zellen für leere Anteile der Daten erzeugt werden; solange jede Zelle bei Verringerung der Auflösung aber noch Daten enthält, wächst die Anzahl der erzeugten Zellen kubisch. Erst wenn leere Zellen erzeugt werden müssten, wird sich der Vorteil der Hash-Struktur zeigen. Der annähernd kubische Verlauf des Speicherbedarfs für drei Beispiel-Datensätze (griechische Kirche, Leica Scan des Büros und ein Kurt3D-Scan) ist in Abbildung 4.21 gezeigt. Die Messdaten wurden durch ein Polynom dritten Grades approximiert, um den kubischen Anstieg zu verdeutlichen.

Der Speicherbedarf variiert hier je nach Größe der rekonstruierten Umgebung. In der Praxis werden mit Voxelgrößen im Bereich von 3 cm bis 5 cm brauchbare Ergebnisse erziehlt. Die Rekonstruktion aus dem Kurt3D-Laserscan benötigte bei einer Gitterauflösung von 5 cm maximal 157.7 MB. Bedingt durch die Größe der rekonstruierten Umgebung wurde bei der Kirche und dem Büroscan wesentlich mehr Speicher benötigt (980 MB). Die Daten lassen sich aber noch problemlos auf Standard-Laptops mit 4 GB Arbeitsspeicher handhaben. Die hier vorgestellten Werte zeigen lediglich den maximal benötigten Speicher. Nach der Rekonstruktion können die angelegten Gitterstrukturen und die Punktdaten, wenn sie nicht für andere Zwecke weiter benötigt werden, aus dem Speicher entfernt werden, so dass die Ressourcen wieder anderen Prozessen zur Verfügung stehen.

4.3.4 Datenkompression

In diesem Abschnitt wird der Einfluss verschiedener Parameter auf die Reduzierung der Dreieckszahl nach der Meshoptimierung evaluiert. Ein wesentlicher Parameter bei der Generierung der initialen Dreiecksnetze ist die gewählte Auflösung des Rekonstruktionsgitters. Durch eine Erhöhung der Auflösung werden entsprechend mehr redundante Dreiecke in ebenen Flächen erzeugt, allerdings können so auch mehr Details repräsentiert werden. Daher wird zunächst untersucht,

welchen Einfluss die gewählte Gitterauflösung auf die Reduzierung der Dreieckszahl hat. Dazu wurden die bereits im vorherigen Abschnitt vorgestellten Datensätze verwendet. Im Experiment wurde die Auflösung in einem Bereich von 3 cm bis 10 cm variiert, da in diesem Intervall in allen Datensätzen bei kleiner Voxelgröße noch zusammenhängende Flächen produziert wurden und auch bei der geringsten Auflösung die Umgebungsgeometrie nach optischem Eindruck noch gut wiedergegeben wurde. Um den durch die Kompression verursachten Fehler zu messen, wurde bei allen Durchläufen mittels `CloudCompare` der mittlere Punktabstandsfehler bestimmt.

Abbildung 4.22 zeigt die Ergebnisse der Experimente. Die linke Grafik zeigt die relative Dreieckszahl bezogen auf die Anzahl der Dreiecke im Eingangsmesh, d.h. kleinere Zahlen bedeuten eine größere Reduktion. Zu beachten ist, dass hier relative Werte angegeben wurden. Absolut gesehen steigt die Anzahl der generierten Dreiecke in den initialen Rekonstruktionen und in den optimierten Meshes mit steigender Auflösung. Das rechte Bild zeigt die gemessenen Punktabstandsfehler. Man kann erkennen, dass die relative Reduzierung sich bei den meisten Datensätzen kaum ändert. Erst bei Auflösungen jenseits von 9 cm ist ein Abfall der erreichten Kompression zu erkennen. Bei den Laserscans sind die Messwerte über den betrachteten Bereich sehr konstant. Eine ähnliche Tendenz zeigt sich beim Punktabstandsfehler. Auffällig ist, dass die ermittelten Werte in den Kinect-Frames stärker schwanken. Dies ist durch eine stärkere Varianz in den Eingangsdaten zu erklären, die folglich Schwankungen im ermittelten Mittelwert verursacht. Die Laserscans weisen weniger Rauschen auf, entsprechend konstant sind die ermittelten Werte.

Die Konstanz der ermittelten Werte lässt sich dadurch erklären, dass die implementierten Optimierungsverfahren auf Planarität spezialisiert sind. Die erreichbare Kompression hängt also im Wesentlichen von der Geometrie der rekonstruierten Umgebung ab. Wie in Abbildung 4.17 gezeigt, bleibt auch bei gröberen Auflösungen die vorherrschende Geometrie erhalten. Bei gröberen Rekonstruktionen können feine Details in den Szenen nicht mehr repräsentiert werden, dementsprechend steigt der Punktabstandsfehler. Der leichte Abfall der Kompressionsrate kommt zustande, weil bei gröberen Rekonstruktionen auch größere Dreiecke erzeugt werden. Diese sorgen dafür, dass der Abbruchschwellwert beim Region-Growing insbesondere im Bereich von Kanten schneller erreicht wird, da diese großen Flächen abgeschrägt werden, so dass relativ gesehen weniger Dreiecke zusammengefasst werden können.

Ein weiterer wichtiger Faktor bei der Optimierung ist das Abbruchkriterium beim planaren Region-Growing. Je größer dieser Wert gewählt wird, um so mehr Flächen werden unter Inkaufnahme eines größeren Fehlers zusammengefasst. Wie sich dieser Faktor auf die Optimierung auswirkt, zeigt Abbildung 4.23. Im gezeigten Experiment wurden für verschiedene Abbruchschwellwerte Kompression und Abstandsfehler bestimmt.

Wie erwartet, steigt die erreichte Kompression mit steigender Toleranz, da mehr Flächen zusammengefasst werden können. Mit der Kompression steigt auch der Fehler. Beim Bürodatensatz und im SICK-Laserscan steigen Fehler und Kompression nur relativ langsam und beide Kurven flachen zu größeren Werten hin ab, weil diese Datensätze in strukturierten planaren Umgebungen aufgenommen wurden. In solchen Szenen bricht das Region-Growing in der Regel an sehr scharfen Kanten ab. Daher werden auch bei Erhöhung des Toleranzwertes nicht wesentlich mehr Flächen zusammengefasst. Dieser Effekt spiegelt sich auch im geringen Ansteigen des Abstandsfehlers wieder. Ein anderes Bild zeigt sich beim Datensatz der Kuppelkirche. In dieser Szene

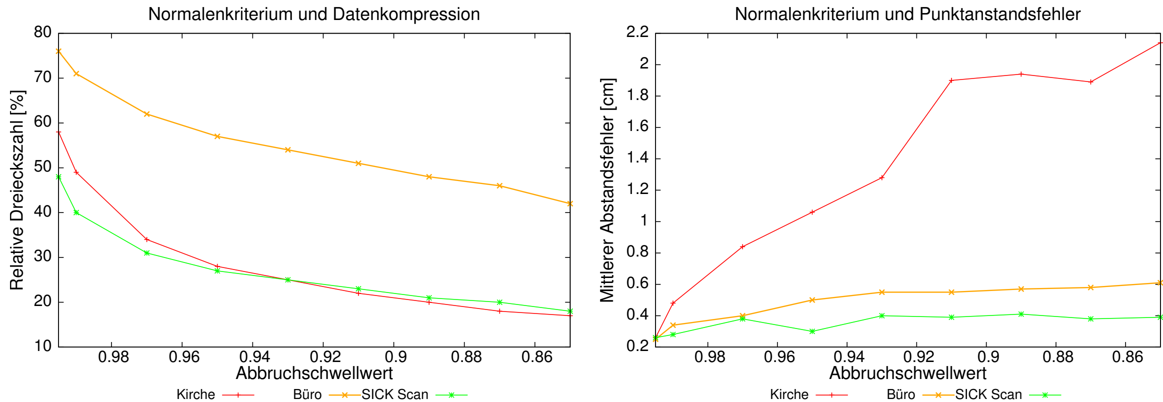


Abbildung 4.23: Datenkompression bei Variation des Normalenkriteriums (links) und mittlerer Punktabstandsfehler (rechts).

gibt es im Bereich der Kuppeln viele nicht-planare Anteile. Durch Erhöhung der Fehlertoleranz beim Region-Growing werden bereits bei sehr geringen Winkelschwellwerten viele Bereiche an den Kuppel "plattgedrückt", wodurch der Fehler sehr schnell ansteigt. Bei den ebenen Datensätzen liegt der maximal gemessene mittlere Abstandsfehler bei 0.6 cm. Beim Kirchendatensatz steigt er bis auf über 2 cm an. Bei kleineren Schwellenwerten steigt in den ebenen Datensätzen die Kompression relativ stark an, der mittlere Fehler wächst aber nur gering. Bei zu kleinen Toleranzen können bereits kleine Winkel zwischen zwei Dreiecken zu einem Abbruch des Region Growing führen. Durch das Messrauschen der Daten werden aber auch ebene Flächen nicht vollkommen glatt rekonstruiert. Solche Flächen weise eine gewisse Maserung auf, die oftmals vom Rauschen herrührt. Daher wird eine gewisse Toleranz benötigt, damit in diesen Regionen die Dreiecke zusammengefasst werden können.

Insgesamt hat sich gezeigt, dass ebene Datensätze relativ fehlertolerant gegenüber zu hoch gewählten Schwellwerten sind. In Datensätzen, in denen die Planaritätsbedingung nicht erfüllt wird, kann allerdings bereits ein kleiner Schwellenwert relativ große Fehler erzeugen. Dies ist bei der Wahl des Parameters die Umgebungsgeometrie zu berücksichtigen. Da die Software primär für den Einsatz in planaren Umgebungen konzipiert wurde, ist als Standardwert für das Region-Growing ein recht hoher Toleranzwert von 0.85 eingestellt.

Ein weiterer Faktor, der Kompression und Fehler beeinflusst, ist der Toleranzwert bei der Optimierung der Konturpolygone vor der Neutriangulierung. Je mehr Kanten zusammengefasst werden, um so weniger Vertices müssen bei der Triangulierung berücksichtigt werden, wodurch die Zahl der erzeugten Dreiecke signifikant sinkt. Wie in Abschnitt 3.5.4 dargestellt, sind in der verwendeten Optimierungsbibliothek `psimpl` verschiedene Verfahren implementiert. Bei der Evaluation hat sich gezeigt, dass sich die verschiedenen Algorithmen vorliegenden Anwendungsfall kaum unterscheiden. Allen in `psimpl` verwendeten Algorithmen wird als Parameter der maximal erlaubte Abstand eines Knotens von der Refenrezkontur vorgegeben. Standardmäßig wird in LVR der Douglas-Peucker-Algorithmus [44] verwendet. Der Einfluss des Toleranzabstandes auf die Datenkompression zeigt Abbildung 4.24.

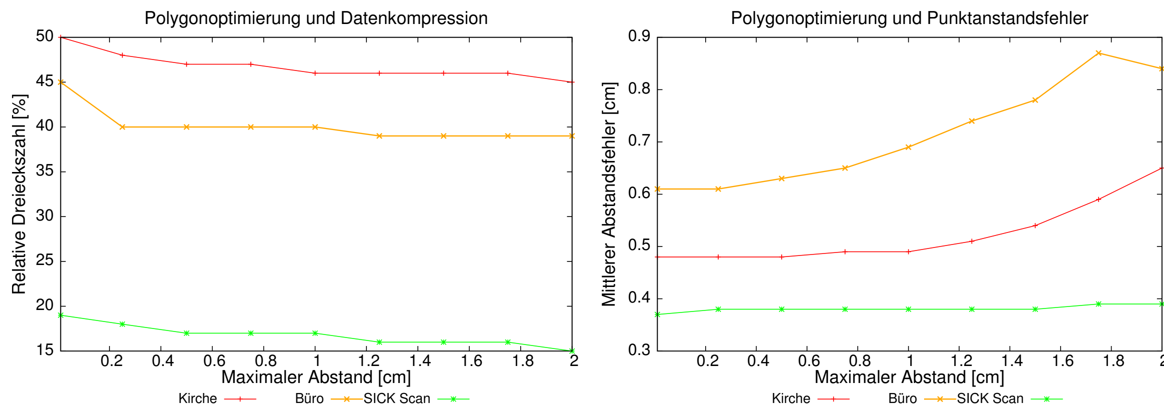


Abbildung 4.24: Datenkompression und Punktabstandsfehler bei Konturoptimierung mit dem Douglas-Peucker-Algorithmus [44]. Links ist die relative Dreieckszahl des optimierten Meshes bei Variation des erlaubten Punktabstandsfehlers bei der Konturverfolgung gemessen. Das rechte Bild zeigt den Verlauf des mittleren Punktabstandsfehlers.

Die Grafiken zeigen, dass eine Variation des Toleranzwertes kaum Einfluss auf die Datenkompression hat. Auch der Abstandsfehler zwischen Rekonstruktion und Punktwolke bleibt über weite Bereiche annähernd konstant. Erst bei höheren Abstandstoleranzen steigt der Fehler signifikant an. Dieses Verhalten ist dadurch zu erklären, dass durch eine Erhöhung der Toleranz bei der Polygonoptimierung nur relativ wenige Vertices eingespart werden können, da der vorgegebene Wert ein Maximalwert ist und in ebenen Bereichen mit scharfen Kanten in den Konturen sehr viele Vertices entlang einer Kante liegen, so dass der Maximalabstand selten erreicht wird. Der Fehlerwert beeinflusst eher, wie gut Knicke in den Konturen approximiert werden. Bei nicht-ebenen Datensätzen weisen die Konturen tendenziell Wölbungen auf, wodurch das Entfernen von Vertices, die solche Konturen repräsentieren, einen relativ hohen Fehler in der Approximation erzeugen kann. Daher steigen die Fehlerkurven beim Kirchendatensatz relativ schnell an. Auch im Bürodatensatz ist bei großen Toleranzen ein Anstieg zu erkennen. Dieser ist allerdings nicht so ausgeprägt wie beim Kirchendatensatz. Dies ist dadurch zu erklären, dass in diesem Datensatz viele Objekte in der Szene vorhanden sind deren Konturen bei hohen Toleranzen nicht mehr gut approximiert werden. Entsprechend dieser Beobachtungen bleiben Kompression und Fehler im SICK-Scan, der in einem leeren Flur aufgenommen wurde, annähernd konstant.

Alle hier angeführten Experimente haben gezeigt, dass die erreichbare Kompression im Wesentlichen von der Umgebungsgeometrie abhängt. Auflösung und Schwellwert bei der Konturoptimierung haben nur wenig Einfluss auf Kompression und Genauigkeit. In ebenen Datensätzen ist das implementierte Verfahren robust gegen Variation des Normalenkriteriums beim Region-Growing. In nicht-planaren Umgebungen kann ein zu hoch gewählter Wert jedoch große Fehler erzeugen. Für derartige Szenerien werden andere Verfahren zur Meshoptimierung benötigt.

4.3.5 Fazit Laufzeit, Speicherverbrauch, Datenkompression

Die Evaluation der implementierten Rekonstruktions- und Optimierungsverfahren hat gezeigt, dass sich mit der vorliegenden Software Rekonstruktionen aus Laserscans in Echtzeit erzeugen lassen. Die Laufzeit der Marching-Cubes Rekonstruktion mittels Standard Marching Cubes und PMC lag bei SICK-Laserscans beim Voxelgrößen zwischen 8 und 10 cm im Bereich einiger hundert Millisekunden pro Scan. Mit Optimierung wurden Laufzeiten von einigen wenigen Sekunden erreicht. Verglichen mit der Zeit, die benötigt wird, um einen Scan aufzunehmen (ca. 4 Sekunden), kann man also von einer vollständigen Prozessierung der Daten zwischen zwei Scans ausgehen. Ähnliche Laufzeiten ergeben sich für Kinect-Frames. Allerdings wird hier aufgrund der hohen Framerate des Sensors (30 Hz) keine Echtzeitfähigkeit erreicht. Dies ist in der Regel aber auch nicht erforderlich, da man in der Praxis normalerweise nicht jeden Frame direkt polygonalisieren will. Vielmehr macht es Sinn, mehrere Frames zu registrieren und aufgrund des starken Rauschens des Sensors vorzufiltern und die gefilterten und registrierten Punktwolken erst nach einiger Zeit zu polygonalisieren.

Der Speicherbedarf der Rekonstruktionsverfahren hängt von der Größe der Umgebung und der Voxelgröße im Rekonstruktionsgitters ab. Für einzelne Laserscans und Kinect-Frames wurden je nach Auflösung 100 MB bis 500 MB benötigt. Die Software kann also problemlos auf handelsüblichen Laptops verwendet werden.

Bei den Untersuchungen hat sich gezeigt, dass die Funktionalitäten, die auf Nächster-Nachbar-Suche basieren, mit Abstand die meiste Laufzeit benötigen. Dementsprechend wichtig ist die Wahl der passenden Bibliothek zur Lösung dieses Problems. In Versuchen hat sich gezeigt, dass FLANN und NABO die geringsten Laufzeiten haben und auch bei der Parallelisierung der Algorithmen gut mit der Anzahl der Prozessorkerne skalieren. Für zukünftige Anwendungen sind daher Implementierungen der benötigten Suchanfragen auf GPUs oder Verteilung der Prozesse auf mehrere Systeme via MPI eine Möglichkeit, die Leistung weiter zu steigern. Für Kinect-Daten gibt es mit Kinect-Fusion bereits eine freie GPU-Implementierung eines echtzeitfähigen Rekonstruktionsverfahrens, dieses stellt aber besondere Ansprüche an die Hardware und kann nur einen sehr kleinen Ausschnitt der Umgebung verarbeiten. Die bei den SICK-Scans und Kinect-Frames gemachten Beobachtungen lassen sich auch auf terrestrische Scans übertragen.

Durch die implementierten Meshoptimierungsverfahren lässt sich die Anzahl der Dreiecke in den Rekonstruktionen bei geeigneter Parametrierung je nach Umgebungsgeometrie auf 20 bis 30 Prozent der ursprünglichen Zahl verringern. Für planare Umgebungen hat sich dabei gezeigt, dass über einen weiten Parameterbereich beim Region-Growing und der Polygonoptimierung sehr gute Kompressionsraten bei gleichzeitig geringem Fehler erreichen lassen. In nicht-ebenen Umgebungen muss der Normalenschwellwert niedrig gewählt werden, damit durch das Region Growing die runden Anteile nicht zusammengefasst werden. Insgesamt hat die Auswertung der verschiedenen Datensätze gezeigt, dass die in Kapitel 3.1 gestellten Anforderungen bezüglich Laufzeit, Datenkompression und Speicherbedarf erfüllt wurden.

Tabelle 4.7: Ergebnisse der Evaluation von GCS-Meshing: Laufzeiten des Verfahrens auf verschiedenen Datensätzen, Anzahl der Dreiecke im Ausgangsmesh und nach der Optimierung sowie die dazu ermittelten Fehlerwerte.

Datensatz	Laufzeit	Dreiecke (Rek.)	Dreiecke (Opt.)	Fehler (Rek.)	Fehler (Opt.)
Scan 1	76.76 s	123 367	4 775	0.35 cm	0.75 cm
Scan 2	39.39 s	67 300	1 644	0.34 cm	0.96 cm
Kinect 1	55.60 s	64 271	15 317	0.24 cm	0.90 cm
Kinect 2	14.15 s	21 014	3 679	0.31 cm	0.74 cm
Kirche	274.27 s	413 704	124 248	0.05 cm	0.30 cm
Büro	85.85 s	130 462	14 966	0.11 cm	0.49 cm

4.4 Vergleich mit anderen Verfahren

Im letzten der Teil der Evaluation werden die in LVR implementierten Algorithmen mit anderen Rekonstruktions- und Meshtoptimierungsverfahren verglichen. Die mit den alternativen Rekonstruktionsverfahren erstellten Meshes können durch Speicherung in einem von LVR unterstüzten Austauschdatenformat eingelesen und anschließend mit den Meshtoptimierungsverfahren bearbeitet werden. Anders herum können die mit LVR erzeugten initialen Meshes anschließend mit externer Meshoptimierungssoftware weiterverarbeitet werden.

4.4.1 Alternative Meshing-Verfahren

Growing Cell Structures

Die Implementierung des Growing-Cell-Struktures-Verfahrens (GCS) [9] wurde von Hendrik Annuth zur Verfügung gestellt. Mit Hilfe der Software lassen sich Punktwolken in Meshes überführen, die als PLY-Dateien abgelegt werden. Als Abbruchkriterium für die Rekonstruktion wird die Anzahl der Dreiecke in der Rekonstruktion vorgegeben, d.h. der Algorithmus verfeinert das Mesh immer weiter, bis die vorgegebene Dreieckszahl erreicht wurde. Auf diese Art und Weise können sehr präzise Rekonstruktionen erzielt werden. Allerdings haben die mit GCS erstellten Meshes ebenfalls die Eigenschaft, dass ebene Flächen unnötig oft unterteilt werden. Die mit GCS gewonnenen Rekonstruktionen können aus Ausgangsdaten für die Meshoptimierung mit LVR verwendet werden. Das Verfahren wurde an einigen der in Tabelle 4.5 vorgestellten Datensätze getestet, so dass wieder SICK-Scans, Kinect-Frames und Daten von terrestrischen Laserscannern berücksichtigt wurden. Um dieses Verfahren mit den in LVR implementierten Marching-Cubes-Verfahren vergleichen zu können, wurde als Abbruchkriterium für die Rekonstruktion die in Tabelle 4.5 von LVR benötigte Dreieckszahl bei der initialen Rekonstruktion vorgegeben. Die so erstellten Meshes wurden anschließend mit den Algorithmen zur planaren Meshoptimierung aufbereitet. Die Ergebnisse der Experimente sind in Tabelle 4.7 dargestellt.

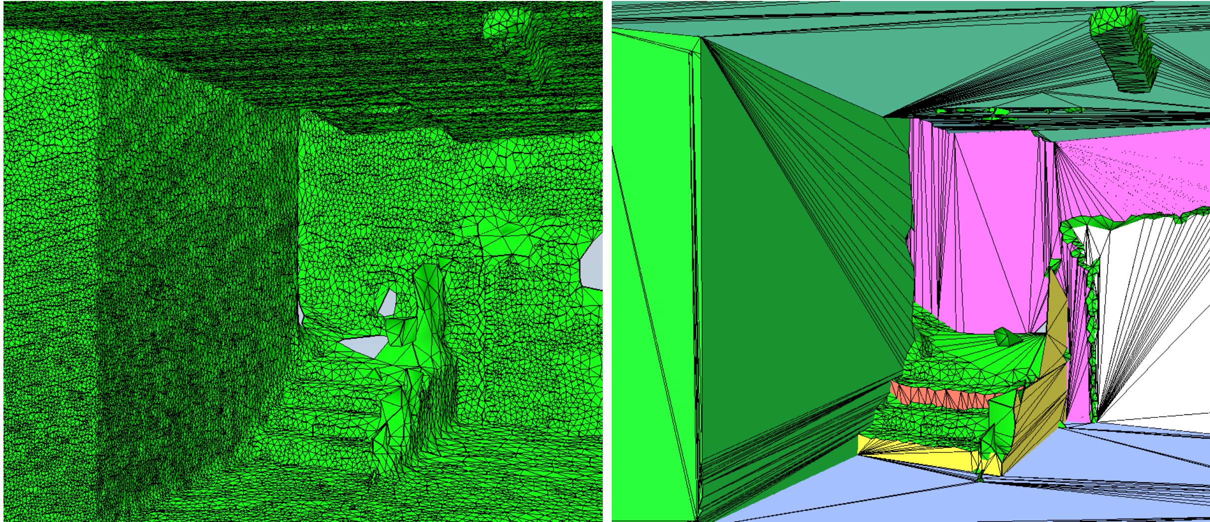


Abbildung 4.25: GCS-Meshing und LVR-Optimierung am Beispiel eines SICK-Laserscans einer Indoor-Szene. Durch den hohen Anteil von ebenen Flächen und der initial guten Schnittkantendarstellung kann eine besonders hohe Datenkompression erreicht werden..

Wie in der Tabelle zu erkennen ist, sind die Laufzeiten des Verfahrens um ein Vielfaches höher als die der Marching-Cubes-basierten Rekonstruktion. Die erreichten Fehlerwerte bei den Kinect-Frames und SICK-Laserscans sind allerdings wesentlich kleiner. So lag der mittlere Punktabstandsfehler bei den SICK-Laserscans im Bereich von gut einem Zentimeter, mit GCS wurden Werte um 0.3 cm erreicht. Wie in den Experimenten zuvor stieg auch hier der ermittelte Fehlerwert nach der Rekonstruktion leicht an. Besonders auffällig sind die hohen Kompressionsraten in den ebenen Laserscans. Sie liegen deutlich höher als bei der Marching-Cubes-basierten Rekonstruktion. Das ist dadurch zu erklären, dass GCS im Bereich von Knickkanten mehr Dreiecke erzeugt und diese so vor der Optimierung schon besser approximiert, wodurch kompaktere Konturpolygone entstehen, die besonders gut retrianguliert werden können. Dieser Effekt ist im linken Bild von Abbildung 4.25 zu sehen.

In dieser Darstellung zeigt sich ein weiterer Effekt, der bei der Rekonstruktion mit GCS auftreten kann. So kann es vorkommen, dass Hüllen über relativ große Löcher oder Regionen mit wenigen Messpunkten gelegt werden. Dieser Effekt kann besonders gut im Bereich des Treppenaufgangs und des Geländers beobachtet werden. Dies liegt daran, dass der Algorithmus versucht, mit der zur Verfügung stehenden Zahl an Dreiecken immer die beste Approximation der Messpunkte zu erzielen. Er beginnt mit kleiner Dreieckszahl und fügt sukzessive neue Dreiecke ein, bis das Maximum erreicht wurde. So kann es manchmal den Fehler verkleinern, solche Regionen zusammenzufassen. Mit einer höheren Zieldreieckszahl werden solche Artefakte oftmals aber wieder aufgebrochen, allerdings schlägt sich das im Speicherverbrauch und der immer weiter steigenden Verarbeitungszeit nieder.

Beim Kirchendatensatz (siehe Abbildung 4.26) zeigt sich, dass der Fehler des initialen Meshes besonders klein ist. Das liegt daran, dass dem Verfahren mit mehr als 400.000 vorgegebenen

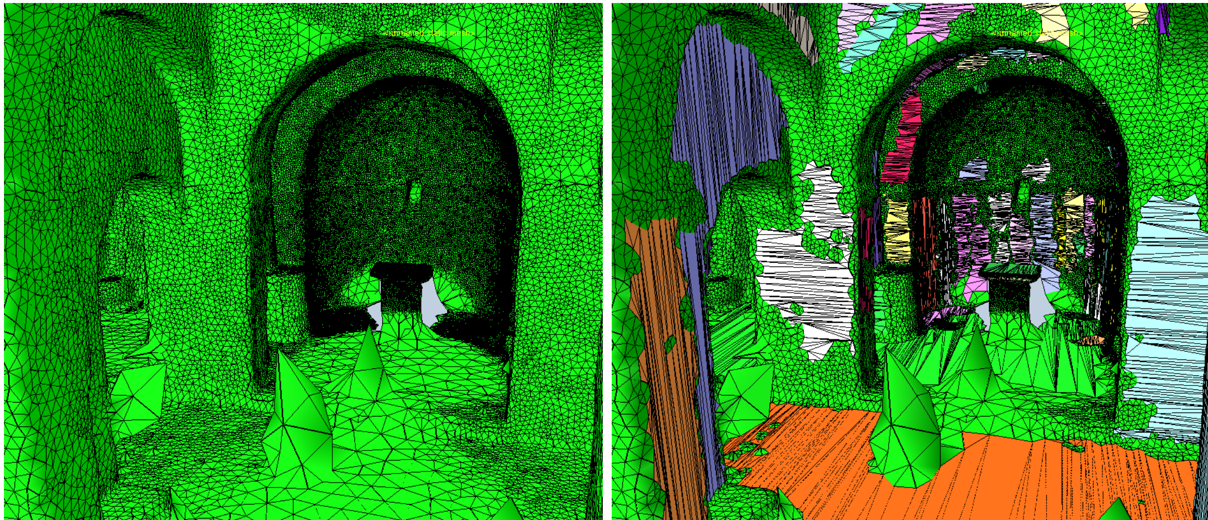


Abbildung 4.26: GCS-Meshing und LVR-Optimierung am Beispiel des Kirchendatensatzes. Im linken Bild sind die durch Messausreißer entstandenen "Hüllenartefakte" besonders deutlich zu erkennen.

Dreiecken viel Möglichkeit gegeben wurde, die Rundungen und Details in den Eingangsdaten zu approximieren. Dies zeigt sich insbesondere an den Durchgängen und im Bereich des Altars im Hintergrund. Diese werden sehr detailreich wiedergegeben. Entsprechend gering ist der Fehler. Allerdings zeigen sich in der Mitte des Bildes auch wieder Artefakte durch Hüllen, die von Messausreißern verursacht werden. Wie auch bei der Marching-Cubes-basierten Rekonstruktion ist durch den geringen Anteil ebener Flächen die erreichte Datenkompression vergleichsweise gering. Lediglich im Bereich des Fußbodens und der Seitenwand konnten Flächen zusammengefasst werden. Durch die sehr feinteilige Approximation der Rundungen bestehen die Konturpolygone aus relativ vielen Segmenten, so dass auch die Neutriangulierung dieser Regionen relativ viele Dreiecke aufweist.

Insgesamt hat die Evaluation gezeigt, dass GCS auch eine geeignete Methode ist, um aus den verschiedenen vorliegenden Eingangsdaten akkurate Rekonstruktionen zu erzeugen. Besonders in ebenen Datensätzen lässt sich durch die initial gute Kantendarstellung eine hohe Datenkompression in Verbindung mit den LVR-Optimierungen erreichen. Problematisch sind die vergleichsweise lange Laufzeit, die mit der Anzahl der zu erzeugenden Dreiecke weiter ansteigt, sowie insbesondere die Hüllenartefakte, die bevorzugt durch Messrauschen hervorgerufen werden. Durch Ausreißer in den Eingangsdaten kann es passieren, dass einzelne Bereiche der Szene in der Rekonstruktion blockiert erscheinen. Dies ist insbesondere beim Einsatz der Meshes als Umgebungsrepräsentation für Roboter problematisch, da durch die Artefakte eigentlich freie Bereiche als nicht-befahrbar dargestellt werden oder - was noch gefährlicher ist - große Löcher als befahrbar repräsentiert werden. Zwar werden auch bei der LVR-Optimierung Löcher geschlossen, dort wird die maximale Lochgröße aber begrenzt. Das ist hier nicht der Fall.

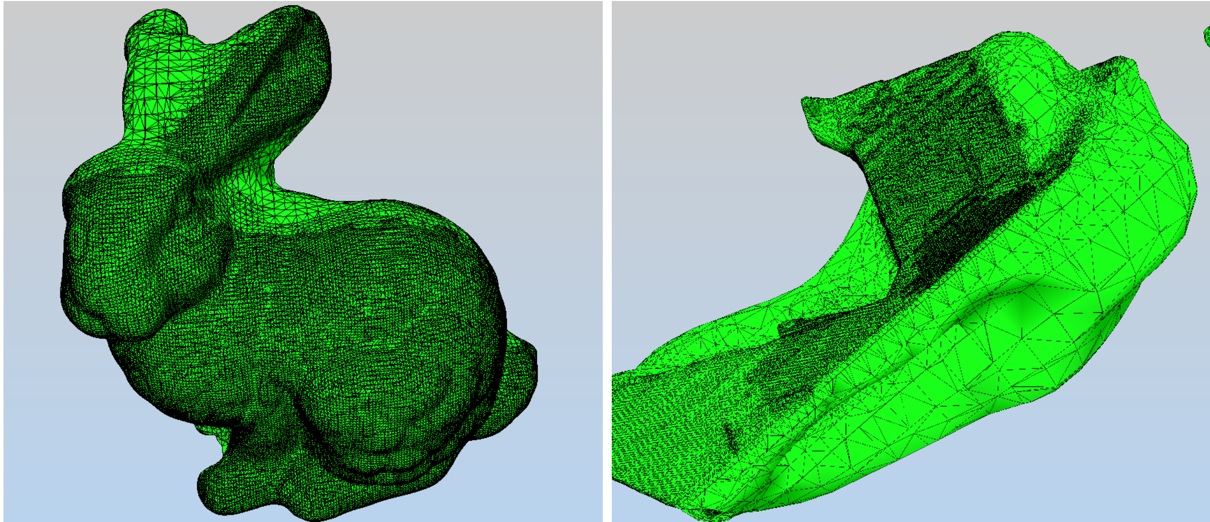


Abbildung 4.27: Poisson-Rekonstruktion mit Meshlab. Links die Rekonstruktion des bekannten Stanford-Bunnys. Das rechte Bild zeigt eine Aufsicht auf die Rekonstruktion des Flurdatensatzes. Man erkennt deutlich die Tendenz des Verfahrens, konvexe Konturen zu erzeugen.

Oberflächenrekonstruktion mit Meshlab

Die Open-Source-Software Meshlab beeinhaltet laut Dokumentation verschiedene Verfahren zum Vermaschen von Punktwolkendaten, u.a. Poisson-Rekonstruktion [92], zwei Marching-Cubes-Varianten mit MLS-basierter Distanzfunktion (RIMLS und AMLS) [74, 149], sowie Ball-Pivoting [18]. Meshlab beeinhaltet zwar sehr viele Funktionen zum Bearbeiten von Polygonnetzen, ist aber nicht auf die Verarbeitung von großen Punktmengen spezialisiert. Eine Möglichkeit, Meshlab unabhängig von der Benutzeroberfläche zu verwenden, ist spezielle Skripte zu generieren und diese vom so genannten Meshlab-Server prozessieren zu lassen. Leider hat sich gezeigt, dass sich die Handhabung der Software bisweilen schwierig gestaltet, da sie sehr instabil ist und häufig abstürzt. Zudem ist es schwierig aufgrund der fehlenden Dokumentation die richtigen Parameter für die einzelnen Verfahren zu finden.

Bei der Verwendung der Marching-Cubes-Varianten konnte trotz intensiven Probierens kein Parametersatz gefunden werden, der eine brauchbare Rekonstruktion aus den Eingangsdaten erzeugt. Im besten erreichten Ergebnis wurden die Eingangsdaten nicht vollständig approximiert und es waren viele Artefakte vorhanden. Da die Ergebnisse nicht vorzeigbar sind, wird auf eine Präsentation an dieser Stelle verzichtet. Die benötigte Zeit für die Rekonstruktion lag bei über 9 Minuten. Insgesamt lassen sich mit diesen Implementierungen keine geeigneten Rekonstruktionen mit vertretbarem Aufwand erzeugen.

Das zweite betrachtete Verfahren ist Poisson-Rekonstruktion. Sie funktioniert es besonders gut mit konvexen Objekten. Daher wurde neben dem Testdatensatz (SICK-Flurscan 2) auch noch die Vorderseite des Stanford-Bunnys verarbeitet. Die Ergebnisse der Rekonstruktion sind in Abbildung 4.27 gezeigt. In beiden Datensätzen zeigt sich das Problem der geschlossenen Hülle.

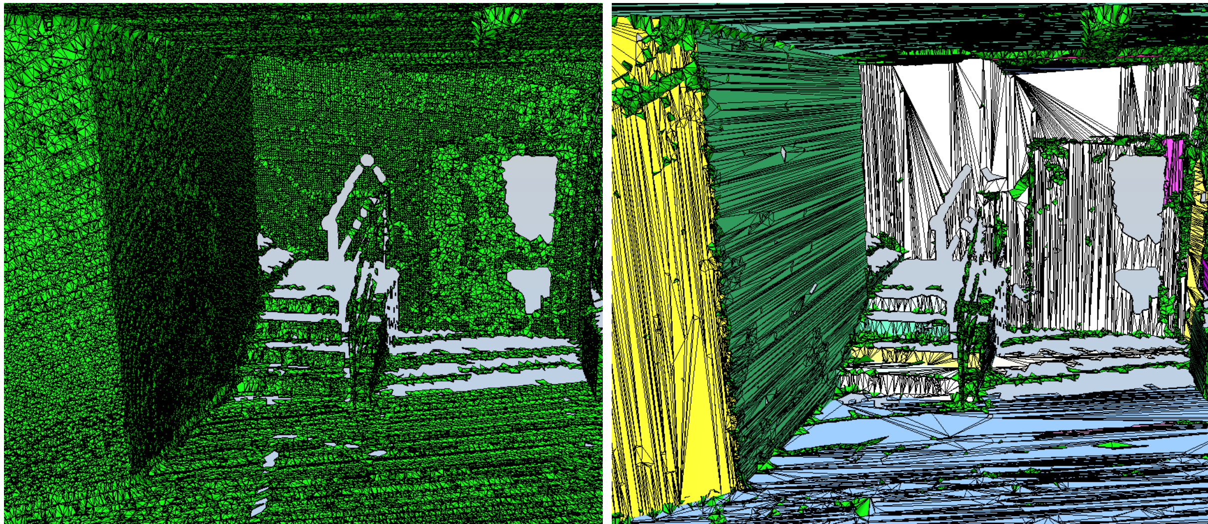


Abbildung 4.28: Ball Pivot Rekonstruktion mit Meshlab (links) und das Ergebnis der LVR-Optimierung (rechts).

So wird die vordere Seite des Kaninchenmodells gut approximiert, zwischen den Ohren werden aber auch Flächen erzeugt. Ein ähnliches Bild zeigt sich bei Rekonstruktion des Flurscans. In Teilen werden die Messpunkte auch hier gut approximiert, es werden aber über weitere Bereiche zusätzliche Flächen erzeugt, die in den Eingangsdaten nicht vorhanden sind. Zudem ist der Speicherbedarf sehr hoch. Für die Rekonstruktion aus den gut 290.000 Messpunkten wurden ca. 4 GB Arbeitsspeicher benötigt. Außer in Meshlab ist die Poisson Rekonstruktion auch in CGAL und PCL implementiert. Hier zeigte sich bezüglich Meshqualität und Speicherverbrauch ein ähnliches Verhalten.

Die besten Ergebnisse wurden mit dem Ball-Pivoting Verfahren in Meshlab erreicht. Die Laufzeit auf dem vorliegenden Datensatz betrug allerdings mehr als 11 Minuten. Das Ergebnis der Rekonstruktion ist im linken Bild von Abbildung 4.28 gezeigt. Die grobe Geometrie wird korrekt wiedergegeben, allerdings ist das Verfahren sehr empfindlich gegen Messrauschen, was sich an den nicht-glatten Wänden bemerkbar macht. Zudem weist das Mesh extrem viele kleine Löcher auf. Mit Hilfe der LVR-Meshoptimierung konnten diese teilweise geschlossen werden, wie im rechten Bild von Abbildung 4.28 zu sehen ist. Allerdings bleibt die Qualität der Rekonstruktion niedrig. Aufgrund der Anfälligkeit gegen Messrauschen und der extrem langen Laufzeit ist das Verfahren für den vorliegenden Anwendungszweck nicht geeignet.

Kinect Fusion

Kinect-Fusion [88] ist ein Verfahren, das GPU-gestützt Meshes fast in Echtzeit aus den Tiefenbildern von Kinect-Kameras erzeugt. Eine freie Implementierung findet sich in PCL. Bewegungen der Kamera werden automatisch getrackt, und die neuen Daten werden sofort in die Rekonstruktion integriert. Bei langsamer Bewegung funktioniert die Integration neuer Daten sehr gut. Bei

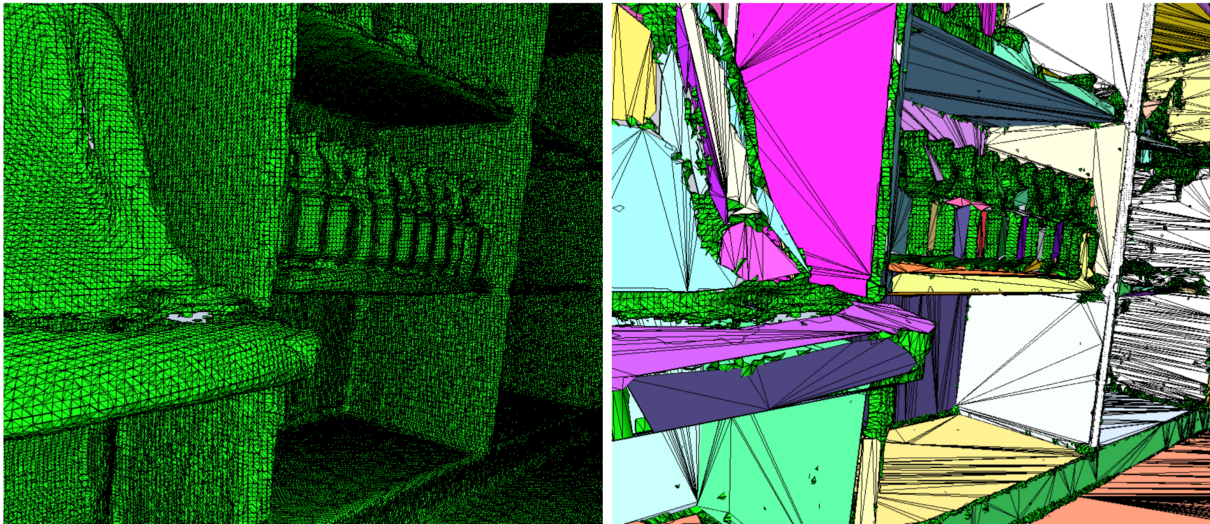


Abbildung 4.29: Kinect Fusion und LVR-Optimierung. Das linke Bild zeigt eine mit Kinect-Fusion erzeugte Umgebungsrekonstruktion. Das rechte Bild zeigt das mit LVR optimierte Mesh.

ruckartigen Bewegungen kann es zu Registrierungsfehlern kommen. Kinect-Fusion verwendet intern eine Variante von Hoppes Distanzfunktion in Kombination mit Marching Cubes. Die erzeugten Meshes können direkt aus der Software im PLY-Format exportiert werden. Rekonstruktionen mit Kinect Fusion weisen die übliche Struktur von Marching-Cubes-Rekonstruktionen mit entsprechend vielen redundanten Dreiecken auf. Das Rauschen der Kinect-Daten wird durch starkes Filtern der Daten und Mittelwertbildung ausgeglichen. Dadurch werden nahezu alle Kanten in der Rekonstruktion abgerundet. Die Größe des maximal rekonstruierbaren Bereichs wird dabei durch die Größe des Arbeitsspeichers der Grafikkarte begrenzt.

Abbildung 4.29 zeigt das Ergebnis einer Rekonstruktion mit Kinect-Fusion. Wie zu erkennen ist, weist das Mesh die typische Marching-Cubes-Struktur auf. Um die Qualität des Meshes zu verbessern, wurde es mit der LVR-Meshoptimierung automatisch aufbereitet. Das rechte Bild zeigt das Mesh nach der Optimierung mit LVR. Durch die Kantenoptimierung konnten im Bereich der Regalböden die zuvor vorhandenen Abrundungen teilweise ausgeglichen und durch scharfe Linienzüge ersetzt werden. Bei diesem Experiment hat sich allerdings gezeigt, dass die von Kinect-Fusion exportierten Meshes nicht optimal repräsentiert sind. Sie enthalten redundante und nicht-referenzierte Vertices sowie degenerierte Dreiecke, die die gleichen Vertices mehrfach referenzieren, also keine Fläche mehr darstellen. Damit die Daten von LVR verarbeitet werden konnten, mussten dieses Artefakte zuvor aus dem Mesh entfernt wurden. Dazu wurden die Reperaturfilter von Meshlab verwendet.

Bei der Aufbereitung mit Meshlab hat sich gezeigt, dass die Vertices der Dreiecke im Mesh dupliziert im Vertexbuffer abgelegt wurden, d.h. gemeinsam genutzte Vertices wurden mehrfach abgelegt. Durch das Entfernen der redundanten Vertices konnte deren Zahl von gut 4 Mio. auf ca. 590.000 reduziert werden. Wesentlich problematischer für die Nachbearbeitung waren die degenerierten Dreiecke. Durch die Mehrfachindizierung gab es Probleme bei der Konturoptimierung

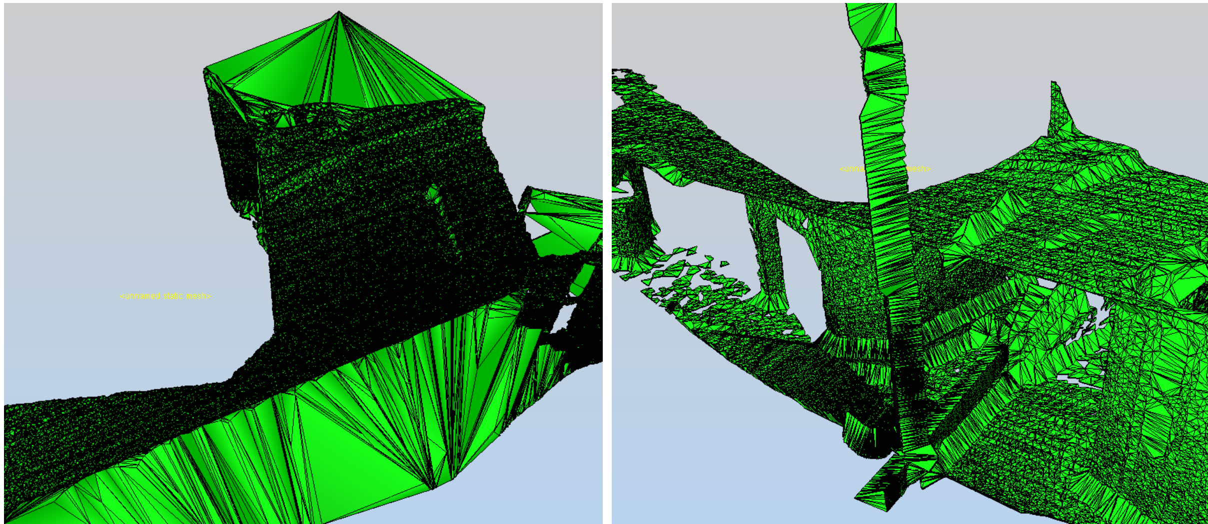


Abbildung 4.30: Rekonstruktion mit Power Crust (links) und Alpha Shapes (rechts).

und dem Schließen der Löcher. Es konnte z.B. passieren, dass eine "Kante" aus zwei gleichen Vertices bestand. Dieser Fall wurde in der ursprünglichen Implementierung nicht berücksichtigt, so dass es zu einem Programmabsturz kam. Insgesamt waren im gezeigten Datensatz ca. 4.200 von den gut 1.1 Mio. ursprünglich vorhandenen Dreiecken betroffen. Nach der Filterung mit Meshlab konnten die Netze dann mit LVR optimiert werden. Dadurch konnte die Zahl der Dreiecke im reparierten Mesh von ursprünglich 3.918.194 auf 300.063 verringert werden, was einer Kompression auf 7.7% der ursprünglichen Größe entspricht.

Prinzipiell lässt sich Kinect-Fusion sehr gut mit LVR kombinieren und ist auch für den Einsatz zur Umgebungsrekonstruktion in der Robotik interessant. In der derzeitigen Implementierung ist die Struktur der erzeugten Meshes allerdings suboptimial, sie müssen dementsprechend vorgefiltert werden, bevor sie mit LVR verwendet werden können. Dies kann manuell mit Meshlab geschehen. Prinzipiell ließe sich der Prozess aber auch automatisieren. Nach der Aufbereitung sind die vorhandenen Meshoptimierungsverfahren in der Lage, die Unebenheiten und Abrundungen in den Eingangsmeshes zumindest teilweise auszugleichen. Problematisch sind allerdings die hohen Anforderungen an die Hardware: Es wird zumindest eine NVIDIA Grafikkarte mit Fermi-Architektur benötigt. Weiterer Nachteil ist der begrenzte Bereich, der mit diesem Verfahren erfasst werden kann.

Weitere Rekonstruktionsverfahren

Des Weiteren wurde die frei verfügbare Implementierung von Amentas Power-Crust-Algorithmus getestet [8]. Dies ist eine Variante der Delaunay-Triangulation, die auch mit verrauschten Eingabedaten zurechtkommen soll. Hier hat sich gezeigt, dass das Verfahren mehrere Minuten benötigt, um die Eingabedaten zu verarbeiten. Die Ergebnisse für die vorliegenden Daten entsprachen in etwa denen der Poisson-Rekonstruktion. Mit dem beigefügten kleinen und konvexen Daten-

Tabelle 4.8: Vergleich Meshoptimierung in LVR mit Edge-Collapsing mit Quadric-Error-Metrik nach Garland und Heckbert. Verglichen werden Punktabstandsfehler und Laufzeit bei vorgegebener Kompressionsrate.

		Fehler		Laufzeit		
Datensatz	Kompression	LVR	Quadric	LVR	LVR (Opt)	Quadric
Kirche	0.47	0.48 cm	0.10 cm	15.80 s	0.90 s	2.72 s
SICK-Scan	0.17	0.37 cm	0.37 cm	1.6 s	0.27 s	0.76 s
Büro	0.40	0.61 cm	0.17 cm	14.17 s	0.79 s	2.50 s

satz ließen sich die Ergebnisse aus den entsprechenden Papern nachvollziehen, für nicht konvexe Umgebungen ist das Verfahren anscheinend nicht geeignet. Weiterhin wurde die α-Shape-Implementierung in CGAL getestet. Nach einigen Versuchen konnte ein für den Testdatensatz passender α-Wert bestimmt werden. Diese Rekonstruktion weist innerhalb der äußeren Geometrie diverse Artefakte auf. Solche Artefakte und die Tatsache, dass die Rekonstruktion sehr von der immer wieder neu zu bestimmenden α-Einstellung abhängig ist, machen dieses Verfahren für den gegebenen Kontext nicht robust genug. Die mit Power-Crust und Alpha-Shape produzierten Rekonstruktionen sind in Abbildung 4.30 gezeigt

4.4.2 Meshoptimierung

Im Folgenden wird die in LVR implementierte Meshoptimierung gegen Meshoptimierung mittels Edge-Collapsing evaluiert. Dazu stehen zwei freie Implementierungen zur Verfügung. Die erste findet sich in Jeff Somers "Mesh Simplification"-Software, die andere ist in Meshlab integriert. In Meshlab wird Garland und Heckberts Fehlermetrik zur Entfernung von Kanten verwendet, Somers Software implementiert zusätzlich Melax' Abstandsmetrik. Hier soll im Folgenden Garland- und Heckberts Verfahren mit der Quadric-Fehlermetrik betrachtet werden, da aus den Experimenten in [195] ist bekannt, das Melax-Funktion bei gleicher Meshqualität in etwa die doppelte Laufzeit benötigt.

Das Verhalten des Edge-Collapsing-Algorithmus wird durch verschiedene Parameter gesteuert: Die Anzahl der zu entfernenden Kanten sowie Strafwerte für das Entfernen von Vertices an Kanten und Knicken, um eine zu starke Veränderung der Topologie zu verhindern. Bei den folgenden Experimenten wurde jeweils die zu erreichende Kompressionsrate vorgegeben. Für alle anderen Parameter wurden die Standardwerte benutzt. Um das Verhalten des Edge-Collapsing im Vergleich zur LVR-Optimierung beurteilen zu können, wurden zunächst die Kompressionsraten, die mit LVR erreicht wurden, vorgegeben und Laufzeit und Punktabstandsfehler gemessen.

Die Ergebnisse für verschiedene Datensätze sind in Tabelle 4.8 dargestellt. Da in LVR die Meshoptimierung auch das Füllen von Löchern und Optimieren von Schnittkanten beinhaltet, wurden in der Tabelle die komplette Laufzeit und benötigte Zeit für das Region-Growing mit Neutriangulierung getrennt angeführt. Man kann deutlich erkennen, dass die Edge-Collapsing-Verfahren

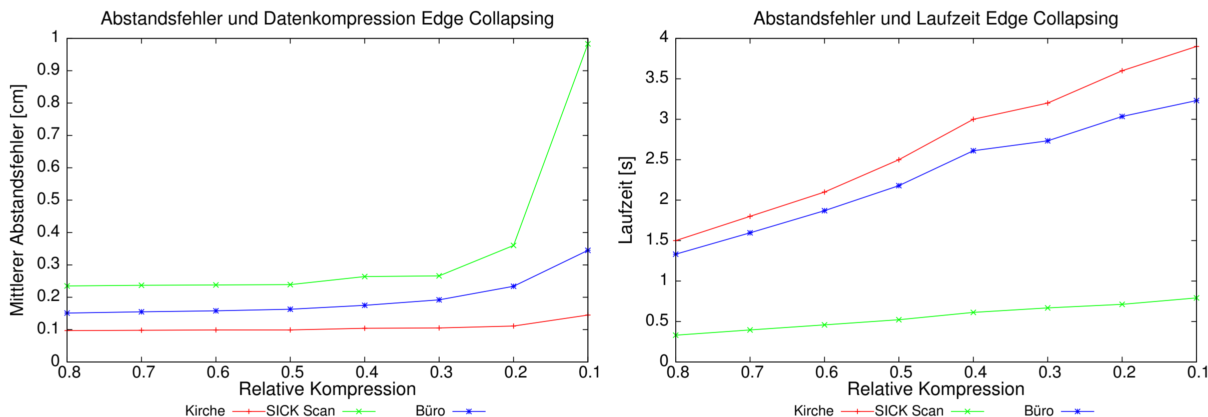


Abbildung 4.31: Abstandsfehler und Laufzeit bei Variation der Kompression beim Edge-Collapsing.

einen deutlich geringeren Abstandsfehler als die LVR-Optimierung verursacht. Dies fällt besonders beim Kirchendatensatz auf. Hier sinkt der Fehler von 0.48 cm auf 0.10 cm. Etwas geringer ist der Unterschied beim Bürodatensatz. Hier beträgt der Fehler des Edge-Collapsings 0.17 cm, LVR verursacht hier einen Fehler von 0.61 cm. Beim SICK-Laserscan verändert sich der Fehlerwert nicht. Dieses Verhalten lässt sich durch die Geometrie der verschiedenen Datensätze erklären. Besonders der Kirchendatensatz profitiert von der Fehlermetrik des Edge-Collapsing-Verfahrens. Sie verhindert, dass in gekrümmten Bereichen Vertices entfernt werden, die einen hohen Fehler im Mesh verursachen. Im leeren Flur gibt es kaum gekrümmte Flächen, so dass der Fehler von LVR von vornherein gering ist. Daher unterschiedet sich der Fehler bei gleicher Kompression nicht.

Betrachtet man die gemessenen Laufzeiten, fällt auf, dass LVR insgesamt sehr viel langsamer ist als das Edge-Collapsing. Dabei ist aber zu berücksichtigen, dass die für die Kompression verantwortlichen Komponenten (Region Growing, Konturextraktion und Neutriangulierung) nur einen Bruchteil der Gesamtlaufzeit des Verfahrens ausmachen. Durch das Edge-Collapse-Verfahren werden keine Löcher im Mesh geschlossen und auch keine Schnittkanten optimiert. Diese Schritte müssen bei der Verwendung des Edge-Collapsing-Algorithmus nach der Reduzierung durchgeführt werden. Wie in Kapitel 4.3.1 gezeigt wurde, benötigen diese Verfahren aber die meiste Laufzeit bei der Optimierung. Daher wurden in Tabelle 4.8 zusätzlich die nur für die zur Komprimierung relevanten Schritte benötigte Zeit angeführt (Spalte *LVR (Opt)*). Betrachtet man diese Zahlen, ist zu erkennen, dass die Region Growing-basierte Meshoptimierung deutlich schneller läuft als das Edge-Collapsing.

Der durch Edge-Collapsing verursachte Fehler in vielen Fällen geringer als der der LVR-Optimierung. Daher ist es von Interesse, wie sich Fehler und Laufzeit entwickeln, wenn man die vorgegebene Datenkompression weiter erhöht. Die Ergebnisse dieses Experimentes sind in Abbildung 4.32 dargestellt. Man kann deutlich erkennen, dass die Fehlerwerte über einen weiten Bereich nahezu konstant bleiben. Dies kommt dadurch zustande, dass die Fehlermetrik zur Berechnung der nächsten zu entfernenden Kanten dafür sorgt, dass nur die Kanten gelöscht werden, die einen geringen Fahler verursachen. D.h., es werden zunächst bevorzugt redundante Vertices

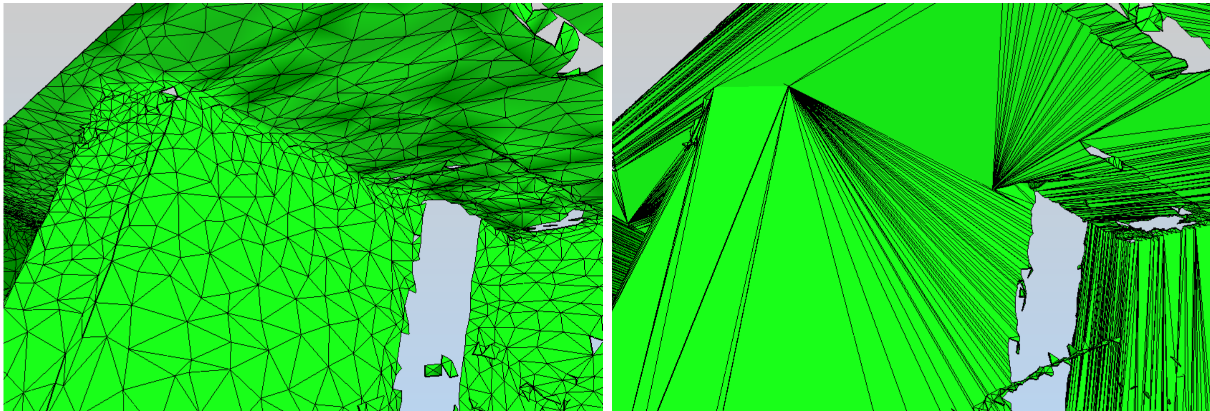


Abbildung 4.32: Vergleich der Triangulation nach Edge-Collapsing und LVR-Optimierung.

entfernt. Erst bei sehr hohen Kompressionsraten werden Vertices entfernt, die kritisch für die Repräsentation der Geometrie sind. Dementsprechend steigt der Fehler bei extrem hohen Kompressionsraten rasant an. Mit der Anzahl der zu entfernenden Kanten steigt auch die Laufzeit des Verfahrens. Bei einer Kompression auf 30% der ursprünglich vorhandenen Dreiecke (bis dahin bleibt der Fehler bei den betrachteten Datensätzen konstant) werden bei den hochaufgelösten Datensätzen schon zwischen 2.5 und 3 Sekunden benötigt.

Abbildung 4.32 zeigt die Unterschiede in der Topologie der optimierten Dreiecksnetze bei gleicher Kompression. Das linke Bild zeigt das Mesh nach dem Edge-Collapsing, das rechte Bild zeigt die von LVR generierte Triangulation. Das Mesh nach dem Edge-Collapsing weist eine sehr viel homogenere Verteilung der Dreiecke auf, während das LVR-Mesh überwiegend aus Triangle-Fans besteht. Im Bereich der Kanten fällt auf, dass diese durch die Linienoptimierung der LVR-Rekonstruktion deutlich schärfer wiedergegeben werden. Das linke Bild zeigt weiterhin die für die Marching-Cubes-Rekonstruktionen charakteristischen weichen Übergänge zwischen den Ebenen.

Wie die Ergebnisse gezeigt haben, haben beide Verfahren Vor- und Nachteile. Durch Edge-Collapsing lassen sich auch in nicht-planaren Umgebungen sehr hohe Kompressionsraten bei geringen Fehlerwerten erreichen. Im Vergleich zur LVR-Optimierung ist die benötigte Laufzeit allerdings sehr hoch. Die mit Edge-Collapsing reduzierten Meshes weisen nach der Optimierung weiterhin eine nicht-optimale Darstellung von scharfen Kanten auf. Hier punktet im Vergleich die LVR-Optimierung. Prinzipiell ist zu überlegen, ob sich die Verfahren nicht kombinieren lassen, indem neben dem Region Growing-basierten Verfahren eine Optimierung der als nicht-eben erkannten Flächen im Netz mittels Edge-Collapsing realisiert wird.

Fazit Alternativen zu LVR

Die Evaluation von Alternativen zu LVR hat gezeigt, dass es für den vorliegenden Anwendungsfall durchaus Alternativen zur Marching-Cubes-basierten Erzeugung von Polygonnetzen gibt. Insbesondere mit Growing-Cell-Structures lassen sich qualitativ hochwertige Meshes erzeugen. Im Kombination mit den implementierten Meshoptimierungsverfahren lassen sich mit diesem

Verfahren sehr kompakte Darstellungen erreichen, wie das Beispiel des SICK-Laserscans gezeigt hat. Das Problem ist, dass dieses Verfahren eine sehr hohe Laufzeit hat, und es momentan keine frei Verfügbare Implementierung gibt.

Die aus der Literatur bekannte Poisson-Rekonstruktion eignet sich hingegen nicht für den betrachteten Anwendungsfall. Mit diesem Verfahren lassen sich nur konvexe Objekte mit guter Qualität rekonstruieren. Auch die in Meshlab vorhandene Implementierung von Ball-Pivoting überzeugt nicht. Delaunay-basierte Verfahren wie Power Crust sind in der Regel gut für konvexe Objekte geeignet. Mit Hilfe von Alpha-Shapes lassen sich geeignete Rekonstruktionen erzeugen. jedoch lässt sich die Findung eines geeigneten α-Wertes nicht ohne Weiteres automatisieren.

Die implementierten Marching-Cubes-Varianten sind frei verfügbar und können zur Rekonstruktion beliebiger Umgebungen eingesetzt werden. Sie sind nicht auf spezielle Geometrien spezialisiert und haben eine kurze Laufzeit. Neben Growing-Cell-Struktures ist Marching Cubes also am besten für den vorliegenden Anwendungsfall geeignet. Meshoptimierung durch Edge-Collapsing bietet eine Alternative zur in LVR verwendeten Datenreduktion durch Region Growing. Allerdings müssen die optimierten Meshes weiter nachbearbeitet werden, um scharfe Kanten repräsentieren zu können. Für zukünftige Weiterentwicklungen ist eine Kombination der Quadric-Fehlermetrik mit der Region Growing-basierten Meshreduzierung denkbar.

4.5 Zusammenfassung

Die Evaluation hat gezeigt, dass sich mit dem Las Vegas Reconstruction Toolkit Karten für mobile Roboter erstellen lassen, die den in Kapitel 3.1 gestellten Anforderungen entsprechen. Bei der Bewertung der Verfahren wurden Datensätze von verschiedenen Sensoren (Kinect-Frames, SICK-Laserscans, hochauflösende Laserscanner) betrachtet, die sowohl die verschiedenen Charakteristika der in der Robotik verwendeten Sensoren (Rauschen, Punktdichte) berücksichtigen, als auch diverse Umgebungsgeometrien abdecken (Indoor-Scans mit vielen ebenen Anteilen, Straßenscan mit unstrukturierten Anteilen und Kuppelkirche mit vielen Rundungen). Die Untersuchung der erreichbaren Genauigkeit hat gezeigt, dass sich mit SICK-Scannern und Kinect-Frames Genauigkeiten unter 1 cm erreichen lassen. Bei Verwendung von terrestrischen Scannern ist die Genauigkeit entsprechend höher. Die Analyse des Laufzeitverhaltens der implementierten Verfahren hat gezeigt, dass das wesentliche Nadelöhr die Nächste-Nachbar-Suche ist. Dazu gibt es inzwischen effiziente Implementierungen, so dass bei Laserscans eine Verarbeitung der Daten in Echtzeit möglich ist. Weiterhin wurde gezeigt, dass sich durch Nutzung der Farbinformationen texturierte Meshes erzeugen lassen, die die Umgebung realistisch wiedergeben. Der Vergleich mit anderen Verfahren hat ergeben, dass die verwendeten Marching-Cubes-Varianten universell einsetzbar sind und einen guten Kompromiss zwischen Präzision der Rekonstruktion und der erforderlichen Laufzeit bieten.

Kapitel 5

Einsatzbeispiele für 3D-Polygonkarten

In diesem Kapitel werden exemplarisch drei Anwendungsbeispiele für dreidimensionale Polygonkarten in der Robotik vorgestellt. Zunächst wird demonstriert, wie sich die automatisch erzeugten Karten als Umgebungsrepräsentation in der Roboter-Simulationsumgebung "Gazebo"[99] nutzen lassen. Im zweiten Abschnitt wird ein Verfahren zum Verfolgen einer Robotertrajektorie in sechs Freiheitsgraden unter Zuhilfenahme einer PMD-Kamera vorgestellt. Drittes Beispiel ist die automatische Erkennung von Möbeln in den Rekonstruktionen, basierend auf semantischer Interpretation der Geometrien der extrahierten Ebenen.

5.1 Umgebungsrekonstruktionen in Simulatoren

Simulationsumgebungen spielen in der Robotik bei der Entwicklung von Algorithmen eine wichtige Rolle. Sie ermöglichen es, neue Verfahren zunächst auf synthetisch generierten Daten zu erproben, bevor sie auf der vorhandenen Hardware getestet werden. Die simulierten Daten werden in einer Simulationsumgebung, basierend auf physikalischen Modellen und Beschreibungen der verwendeten Hardware und der Umgebung, berechnet. Die Umgebungen und Roboter werden häufig durch Polygonnetze beschrieben. In der Regel erfolgt die Modellierung aufwändig per Hand. Für die Robotermodelle ist der dazu benötigte Aufwand nicht so kritisch, da diese in verschiedenen Simulatoren wiederverwendet werden können. Soll ein Roboter aber in einer neuen Szene getestet werden, muss diese zunächst neu modelliert werden. Da die manuelle Konstruktion realitätsnaher 3D-Modelle sehr zeitaufwändig ist, werden oftmals zum Testen nur vereinfachte Umgebungsmodelle verwendet. Durch die Verwendung von Laserscannern in Kombination mit LVR lassen innherhalb sehr kurzer Zeit realitätsnahe polygonale Umgebungsrepräsentationen erzeugen. Hier soll exemplarisch gezeigt werden, dass sich die mit LVR erzeugten Polygonnetze zur Szenenmodellierung im Simulator Gazebo nutzen lassen. Gazebo besitzt eine Anbindung an ROS, so dass sich simulierte und reale Hardware direkt austauschen lassen.

Die Beschreibung der zu simulierenden Umgebung erfolgt in Gazebo durch XML-Scripte, die die physikalischen Eigenschaften, wie z.B. Masse, Reibung usw., der vorhandenen Objekte be-

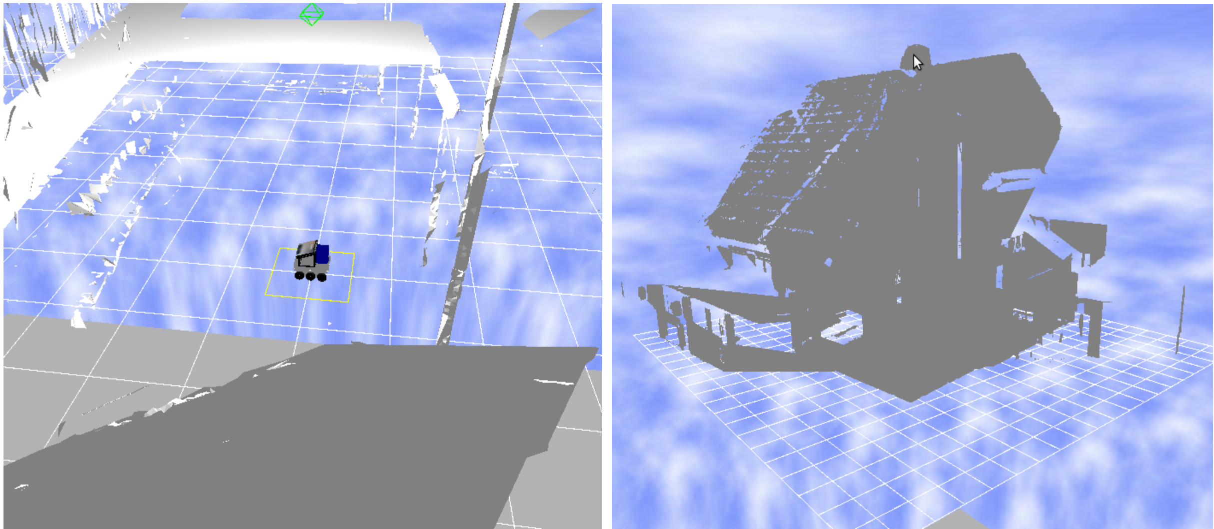


Abbildung 5.1: Rendering von mittels LVR generierten Polygonnetzen in Gazebo. Werden die Polygone einfach eingefügt, werden beim Rendern alle Flächen, deren Normale vom Betrachter wegzeigt, ausgeblendet (links). Werden die Flächen doppelt mit invertierten Normalen eingefügt, wird die Umgebung korrekt dargestellt.

schreiben. Die Umgebungsgeometrie inklusive Texturen kann in Form von Polygonnetzen im Collada-Format geladen werden. In diesem Experiment wurde ein aus einem Leica-Scan erzeugtes Polygonnetz eines Hörsaalgebäudes der Universität Osnabrück (“Reithalle”) dieses Format exportiert und anschließend zusammen mit dem Modell eines Kurt-Roboters in Gazebo geladen. Dabei hat sich gezeigt, dass das Visualiserungstool von Gazebo Probleme beim Rendern hat, wenn die vorhandenen Flächen nicht explizit mit Vorder- und Rückseite modelliert wurden. Ist dies nicht der Fall, werden Flächen, die vom Betrachtungspunkt wegzeigen, ausgeblendet (“Back Face Culling”). Standard Modellierungstools wie Blender oder CAD-Programme erzeugen Modelle oft durch so genannte “Konstruktive Festkörpergeometrie”. Dabei werden komplexe Objekte durch boolsche Operationen von einfachen Primitiven wie Quadern oder Kugeln erzeugt. Auf diese Art und Weise entstehen Objekte, die immer eine klar definierte Außengemetrie haben. Es kann also nicht passieren, dass man in Objekte “hinein” blickt. Daher haben z.B. Wände in so erstellten Modellen immer eine bestimmte Breite. In den von LVR erzeugten Modellen ist dies nicht zwangsläufig der Fall. Wird eine Wand nur von einer Seite erfasst, besteht sie aus einer einzelnen unendlich dünnen Fläche. Dies verursacht hier bei Visualsierung in Gazebo Darstellungsprobleme. Das Problem kann zu Lasten der Darstellungsleistung gelöst werden, indem die erzeugten Polygonflächen doppelt mit invertierten Normalen eingefügt werden (s. Abbildung 5.1). In den Tests hat sich gezeigt, dass Probleme bei der Darstellung in Gazebos Visualierungstool keinen Einfluss auf die Simulation von Laserscannerdaten haben.

Neben dem Dreiecksnetz der Reithalle wurde in die Szene die Simulation eines Kurt-Roboters eingefügt. Diese besteht aus der Simulation des Roboterchassis mit Differentialantrieb sowie eines Sick LMS200 2D Laserscanners. Die Ansteuerung des Roboters erfolgte dabei über eine eigene ROS-Node, die Visualisierung der generierten Laserscans erfolgte in RVIZ. Beim Navigieren des

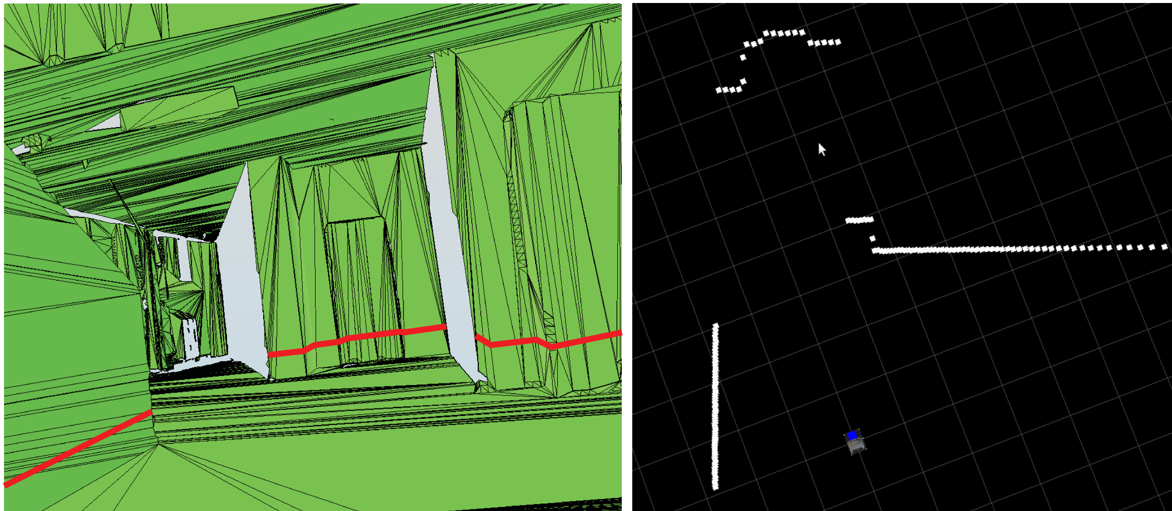


Abbildung 5.2: Simulation eines 2D-Laserscans in Gazebo. Das linke Bild zeigt die verwendete Polygonkarte der Umgebung mit eingezeichneter Scanebene. Rechts der simulierte Scan.

simulierten Roboters hat sich gezeigt, dass die Kollisionserkennung zwischen Umgebung und Robotermodell korrekt funktionierte. So war es nicht möglich, durch Wände zu fahren. Auch die Simulation der Schwerkraft funktionierte wie erwartet. Wurde der Roboter zu hoch über den Boden eingefügt, wurde sein Fall (mit fatalen Folgen für das Modell) von der Bodenebene abgebremst. Abbildung 5.2 zeigt einen Ausschnitt aus der Reithallen-Rekonstruktion zusammen mit einem in RVIZ simulierten 2D-Laserscan. Zum besseren Verständnis ist in der Rekonstruktion die ungefäre Scanebene eingezeichnet. Man kann erkennen, dass der simulierte 2D-Laserscan die Umgebung sinnvoll wiedergibt.

Die Beispiele hier haben gezeigt, dass sich die mit LVR erzeugten Polygonnetze zur Umgebungsrepräsentation in Simulationsumgebungen einsetzen lassen. In Gazebo erfolgt die Erzeugung der Laserscans in der verwendeten Umgebung nahezu in Echtzeit. Die Steuerung des Robotermodells funktionierte problemlos. Prinzipiell ließen sich auch die erzeugten Texturen mit in das Modell einbeziehen. Leider hat sich gezeigt, dass die Konvertierung vom in LVR zur Repräsentation von Texturierten Meshes verwendeten OBJ-Format ins Collada-Format zumindest in Blender und Meshlab nicht zuverlässing funktionierte. Die Zuordnung der Texturen zu den Flächen ging in beiden Fällen verloren. Derzeit gibt es einen experimentellen Collada-Export in LVR, der derzeit allerdings keine Texturen unterstützt. Das Hauptproblem besteht derzeit momentan, dass die Meshes nicht effizient visualisiert werden. Eventuell besteht in Zukunft die Möglichkeit dieses Problem durch ein geeignetes Plugin für Gazebo zu lösen.

Neben Gazebo gibt es noch weitere Simulationsumgebungen. Eine bekannte ist USARSim [29]. Auch für diesen Simulator gibt es ein Modell des Kurt-Roboters [4]. USARSim nutzt die Unreal-Spieleengine in Verbindung mit dem “Unreal Developer Kit” [38] zur Simulation von Robotern. Dies ist für nicht kommerzielle Zwecke frei verfügbar. Diese Engine nutzt ein prorietäres Dateiformat, dass nur von den mitgelieferten Modellierungstools vollständig unterstüzt wird. Damit ist es aber leider nicht gelungen, von LVR erzeugte Meshes zu importieren.

5.2 6D-Pose-Tracking mittels einer PMD-Kamera

Zweites Anwendungsbeispiel für die Nutzung polygonaler 3D-Karten in der mobilen Robotik ist ein Verfahren zum 6D-Pose-Tracking eines mobilen Roboters mit Hilfe einer PMD-Kamera. Das hier vorgestellte Verfahren wurde im Rahmen einer Bachelorarbeit [208] entwickelt und auf einer Fachtagung präsentiert [209]. Die Darstellung der Verfahren und Ergebnisse basiert auf diesen Veröffentlichungen.

5.2.1 Problemstellung

Roboter lokalisieren sich innerhalb von Gebäuden in der Regel mit Hilfe von zweidimensionalen Rasterkarten [80]. Die Pose des Roboters wird dabei in drei Dimensionen (x- und y-Koordinaten und Orientierung) vorgehalten. Stand der Technik sind dabei probabilistische Methoden zur Aufrechterhaltung mehrerer Posehypothesen [57, 188]. Diese Art der Lokaliserung basiert allerdings auf einer zweidimensionalen Umgebungsdarstellung. Für viele Anwendungen werden aber sowohl 3D-Sensordaten als auch 3D-Umgebungsdarstellungen benötigt, um z.B. Hindernissen sicher ausweichen zu können [82, 167]. Daher wurden in den letzten Jahren Verfahren zur Lokalisierung in 6D Posen mittels 3D Daten entwickelt.

Eine verbreitete Art, das 6D-SLAM-Problem zu lösen, ist das Registrieren von 3D Laserscans mittels ICP-Verfahren in Kombination mit Methoden zum Schließen von Schleifen in den Trajektorien [24]. Die einzelnen Laserscans werden dabei iterativ gegeneinander registriert. Der so ermittelte Poseversatz zwischen zwei Scans wird dann verwendet, um die Poseschätzung des Roboters zu aktualisieren. Eventuell auftretende Registrierungsfehler akkumulieren bei dieser Art von inkrementellem Matching. Durch Erkennung von Schleifen in den Trajektorien, d.h. Posen, an denen der Roboter bereits gewesen ist, kann dieser Fehler korrigiert werden, indem der gemessene Posenversatz an einem Schleifenpunkt auf vorherige Posen zurückgerechnet wird. Das Problem bei diesem Ansatz ist, dass das Aufnehmen von 3D-Laserscans vergleichsweise viel Zeit benötigt und Eigenbewegungen des Roboters und anderer Objekte in den Szenen die Messungen verfälschen können. Zudem sind die anfallenden Datenmengen sehr hoch. Mit der Einführung von 3D-Kameras wurden auch 6D-LAM-Verfahren entwickelt, die für diese Art von Daten optimiert sind "RGB-D SLAM" [51, 182]. Diese Verfahren basieren darauf, 3D-Merkmale in den Tiefenbildern der Kameras zu tracken und somit die Poseschätzung aufrecht zu erhalten. Die Karte besteht dabei aus den getrackten Merkmalen bzw. den registrierten Punktwolken.

Die von merkmalsbasierten Verfahren verwendeten Karten sind sehr kompakt, da nur die Positionen und Deskriptoren der zum Verfolgen der Pose verwendeten Merkmale gespeichert werden müssen. Allerdings sind solche Karten sehr speziell und nicht allgemein verwendbar. Punktwolken als Kartenrepräsentation enthalten zwar viel geometrische Information, sind aber sehr speicherintensiv und zur Generierung von Posehypothesen á la Monte Carlo Lokalisierung nicht geeignet. Zudem ist die Darstellung auch bei extrem hoher Auflösung immer noch diskret. Mit Hilfe von Polygonkarten lassen sich Umgebungen sehr kompakt darstellen und das Problem der Diskretisierung entfällt. Mit Hilfe von Raytracing-Verfahren unter Zuhilfenahme eines geeigneten Sensormodells lassen sich in polygonalen Umgebungsdarstellungen effizient synthetische Punktwolken

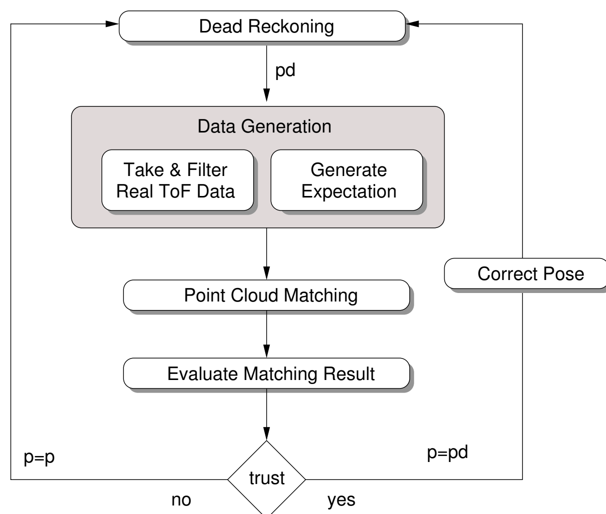


Abbildung 5.3: Ablaufdiagramm zum 6D-Posetracking

sei es für Laserscanner (in 2D oder 3D) oder 3D-Kameras erstellen, die dann als Posehypothesen in einem partikelfilterbasierten Lokaliserungsverfahren verwendet werden können. Polygonkarten können also für verschiedene Arten von Sensoren als Kartenrepräsentation verwendet werden.

In [183] wurden verschiedene 2D-Sensoren effizient in ein Gesamtsystem integriert, das zur Lokalisierung eine 3D Polygonkarte als Umgebungsrepräsentation verwendete. Die Posehypothesen wurden dabei mittels Raytracing generiert. Die Lokalisierung erfolgte dabei aber wiederum in der Ebene. Im Folgenden soll gezeigt werden, dass sich dieser Ansatz auch auf 6D-Posen und 3D-Sensoren erweitern lässt. Da die Definition eines Fehlermaßes zwischen zwei 3D-Punktwolken schwierig ist, soll zur Demonstration der Funktionalität zunächst ein inkrementelles 6D-Posetracking mittels ICP implementiert werden. Das Posetracking wurde mit einer PMD O3D100 Kamera durchgeführt. Prinzipiell lässt sich das Verfahren aber für beliebige 3D-Kameras implementieren. Erste Tests mit Kinect-Kameras wurden bereits durchgeführt.

5.2.2 Das Posetracking-Verfahren

Das Verfolgen der Pose geschieht nach dem in Abbildung 5.3 gezeigten Schema. Zwischen zwei Frames der Kamera wird die Pose des Sensors mittels IMU oder Odometrie nachgehalten, um eine initiale Schätzung der Sensorposition $\hat{\boldsymbol{P}}_t \in \mathbb{R}^6$ zum Zeitpunkt t zu erhalten. Sobald der Sensor neue Daten liefert, werden die eingehenden Daten gefiltert, und es wird basierend auf der letzten Poseschätzung und dem Sensormodell eine simulierte Punktwolke mittels Raytracing aus der Polygonkarte generiert. Sie entspricht den erwarteten Sensorwerten in $\hat{\boldsymbol{P}}_t \in \mathbb{R}^6$. Diese synthetische Punktwolke wird gegen die real gemessenen Sensordaten mittels ICP gematcht. Die Qualität des Matches wird mittels einiger einfacher Heuristiken ausgewertet. Wenn das Ergebnis als vertrauenswürdig eingeschätzt wird, wird die geschätzte Pose $\hat{\boldsymbol{P}}_t$ gemäß des mittels ICP gefundenen Poseversatzes Δ_{ICP} korrigiert ($\boldsymbol{P}_{C,t}$).

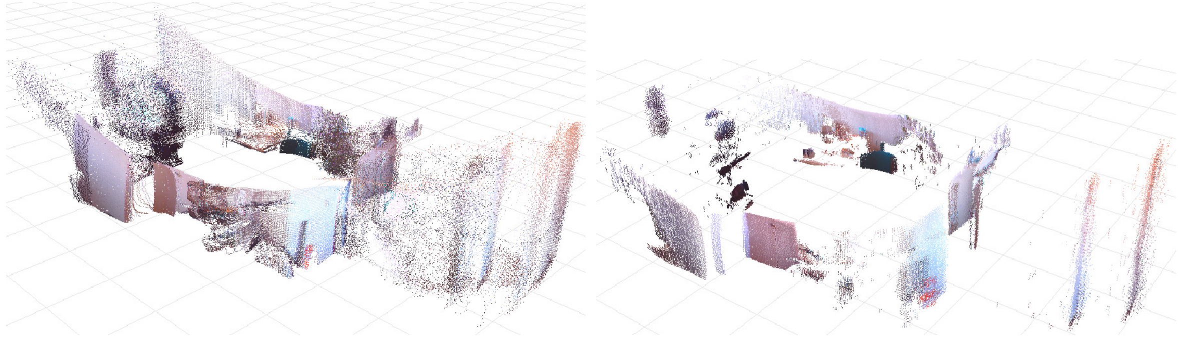


Abbildung 5.4: Filterung von PMD-Daten.

Filtern der Kameradaten

Da aufgrund der relativ geringen Auflösung der Kamera bereits wenige Fehlmessungen in den Eingangsdaten das Ergebnis des ICP-Matchings verfälschen können, werden die Daten zunächst mit dem von Huhle [85] vorgestellten Verfahren gefiltert. Fehlmessungen treten in den Daten vor allem an Tiefensprüngen auf, da in diesen Fällen das von den zwei beteiligten Oberflächen reflektierte Signal gemittelt wird. Auch Einstrahlungen durch Fremdlicht sind problematisch und können zu Ausreißern führen. Das verwendete Verfahren evaluiert für jeden Pixel den Abstand zu seinen Nachbarn. Dabei erhalten Pixel, die auf einer homogenen Oberfläche liegen, eine bessere Bewertung. Liegt die Bewertung unterhalb eines bestimmten Schwellenwertes, wird der Pixel als Outlier betrachtet und aus der Punktwolke entfernt. Dabei wird ein gaußverteilter Fehler in den Tiefenmessungen angenommen. Die Bewertung s_i für einen Pixel wird mit folgender Formel bestimmt, wobei σ_d die empirisch bestimmte Varianz des Tiefenrauschens ist und N_i die Nachbarn des betrachteten Pixels sind:

$$s_i = \sum_{j \in N_i} s_j e^{-\frac{1}{\sigma_d}(d_i - d_j)^2}$$

Abbildung 5.4 zeigt das qualitative Verhalten des Filteralgorithmus. Das linke Bild zeigt die ungefilterte Punktwolke. Man kann deutlich die “Geisterpunkte” an Tiefensprüngen in der Szene erkennen. Durch die Filterung kann ein Großteil dieser Artefakte entfernt werden.

Sensormodell der PMD-Kamera und Raytracing

Zur Modellierung der PMD-Kamera wird ein Lochkameramodell angenommen (siehe Abbildung 5.5. Für jeden Pixel in der Bildebene wird ein Strahl durch den Mittelpunkt $\boldsymbol{o}$ der Kamera berechnet. Der erste Schnittpunkt dieses Strahles mit der Polygonkarte ergibt den virtuellen Datenpunkt. Da die internen Parameter der Kamera (Brennweite f und physikalische Größe der Pixel) nicht bekannt sind, wird das Kameramodell durch die maximalen und minimalen horizontalen (α, β) und vertikalen (ϕ, γ) Öffnungswinkel parametriert. Diese Werte können leicht empirisch bestimmt werden. Auf diese Weise lassen sich auch Fehler in der Zentrierung des Chips

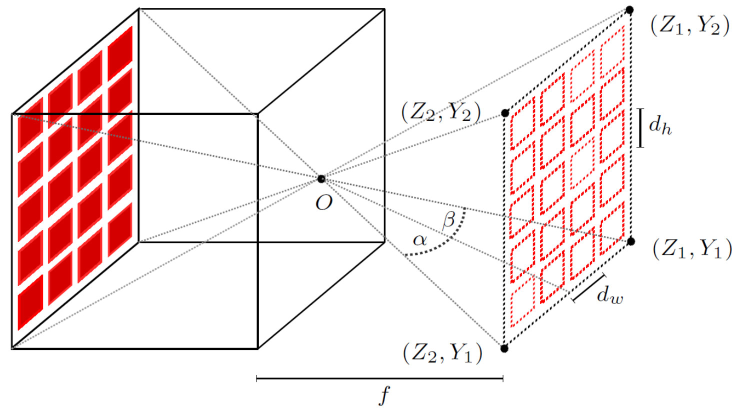


Abbildung 5.5: Lochkameramodell der PMD-Kamera mit virtueller Bildebene und Parametern.

ausgleichen. Mit Hilfe dieser Winkel lassen sich die Positionen der oberen rechten (Y_1, Y_2) und unteren linken Ecke (Z_1, Z_2) der gespiegelten Bildebene der Kamera bestimmen:

$$Z_1 = \begin{pmatrix} f \\ -\tan\alpha \cdot f \end{pmatrix}, \qquad Z_2 = \begin{pmatrix} f \\ \tan\beta \cdot f \end{pmatrix},$$
$$Y_1 = \begin{pmatrix} f \\ -\tan\phi \cdot f \end{pmatrix}, \qquad Y_2 = \begin{pmatrix} f \\ \tan\gamma \cdot f \end{pmatrix}.$$

Aufgrund des Strahlensatzes hat die Brennweite f keinen Einfluss auf die weitere Berechnung, da über die Öffnungswinkel der Kamera parametriert wird und die tatsächliche Position der Bildebene daher keine Rolle spielt. Zur Vereinfachung wird daher $f = 1$ angenommen. Für Höhe h und Breite w der Bildebene ergibt sich aus den obigen Gleichungen:

$$w = d(\overline{Z_1 Z_2}) = \sqrt{1 + (-\tan\alpha - \tan\beta)^2},$$
$$h = d(\overline{Y_1 Y_2}) = \sqrt{1 + (-\tan\phi - \tan\gamma)^2}.$$

Zur Bestimmung der Ausgangspunkte der Strahlen des Raytracings wird diese Bildebene gemäß der physikalischen Pixelauflösung der Kamera diskretisiert:

$$d_x = \frac{w}{r_h} \qquad d_y = \frac{h}{r_v}.$$

Ausgehend von den so berechneten Pixelpositionen in der Bildebene werden jetzt die Schnittpunkte der Strahlen mit dem Polygonmodell bestimmt. Zur effizienten Schnittpunktberechnung zwischen Strahlen und Modell wird CGAL [30] verwendet. Die Dreiecke der Polygonkarte werden dazu in eine AABB-Suchstruktur [191, 211] eingefügt. Bei der Berechnung der Strahlen muss neben der Pose $\boldsymbol{P}_{C,t}$ des Roboters auch die Orientierung der Kamera relativ zu dessen Zentrum berücksichtigt werden ($\boldsymbol{T}_C$). Der translative Anteil von $\boldsymbol{T}_C$ kann durch Ausmessen zumindest grob geschätzt werden. Zur genauen Bestimmung wird die Kamera gegen eine flache Fläche,

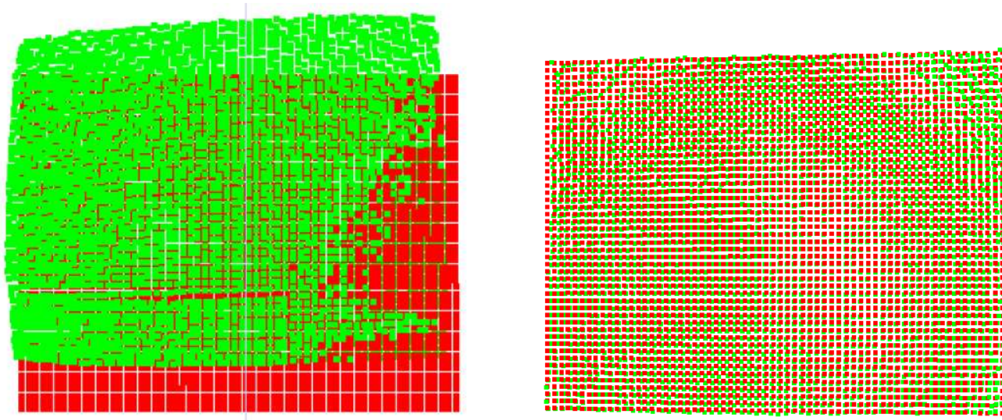


Abbildung 5.6: Kalibrierung der Ausrichtung der PMD Kamera. Ohne eine genaue Bestimmung der Ausrichtung der Kamera besteht ein Offset zwischen simulierten (rot) und real gemessenen Daten (grün).

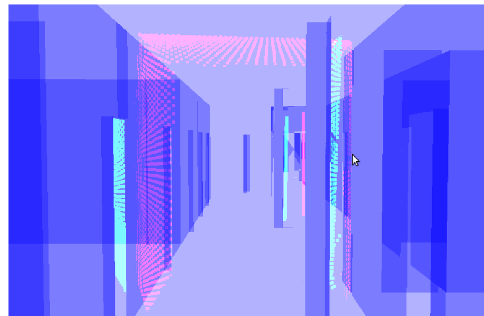


Abbildung 5.7: Matching von simulierten und gemessenen PMD-Daten. Die simulierten Daten sind rot dargestellt. Die real gemessenen grün. Der Offset zwischen diesen beiden Punktwolken wird duch ICP bestimmt.

z.B. einer Wand gehalten. Ohne Kalibrierung wird in diesem Fall ein Offset zwischen den realen Daten und der simulierten Punktwolke zu beobachten sein. Bestimmt man die Transformation zwischen diesen beiden Punktwolken, z.B. mittels ICP, erhält man eine sehr gute Schätzung für $\boldsymbol{T}_C$, wie in Abbildung 5.6 zu sehen ist.

Im linken Bild ist der Offset zwischen der real mit der Kamera gemessenen Punktwolke (grün) und der simulierten Punktwolke (rot) deutlich zu erkennen. Im rechten Bild wurde die simulierte Punktwolke gemäß der mittels ICP-Matching bestimmten Orientierung transformiert. Die beiden Punktwolken sind nun annähernd deckungsgleich.

Beurteilung des Matchings

Während des Posetrackings wird der Poseversatz zwischen der an der geschätzten Stelle simulierten Punktwolke und der real gemessenen Punktwolke mittels ICP bestimmt, wie in Abbildung 5.7 dargestellt. ICP konvergiert immer in einem lokalen Minimum, es ist aber nicht garantiert, dass

jedes Mal der optimale Match zwischen der realen und gemessenen Punktwolke bestimmt wird. Zudem kann es in Umgebungen, in denen es kaum geometrische Merkmale gibt, vorkommen, dass die beiden Punktwolken perfekt aufeinander abgebildet werden, der reale Versatz aber nicht bestimmt werden kann, da die Umgebung überall gleich aussieht. Ein Beispiel für eine solche Situation liegt vor, wenn die Kamera z.B immer parallel zu einer Wand bewegt wird. In diesem Fall werden bei guter initialer Poseschätzung beide Punktwolken direkt aufeinander liegen, es kann also keine Aussage über die zurück gelegte Strecke erfolgen.

Das Matchen mit ICP kann aus verschiedenen Gründen fehlschlagen. Ist zum Beispiel die Karte nicht aktuell oder fehlerhaft, wird die erwartete Punktwolke sehr stark von der gemessenen abweichen, so dass mit großer Wahrscheinlichkeit kein genauer Versatz zwischen den Punktwolken berechnet werden kann. Weitere Probleme können auftreten, wenn transparente oder spiegelnde Objekte in der Umgebung vorhanden sind. Aufgrund des Messprinzips der Kamera werden in diesen Fällen keine zuverlässigen Daten vorliegen. In extremen Fällen kann es auch vorkommen, dass der verwendete Rauschfilter einen Großteil der Messpunkte entfernt, so dass in der aufgenommenen Punktwolke nicht mehr ausreichend Messpunkte vorhanden sind, um die Umgebungsgeometrie ausreichend genau für ein positives Matching wiederzugeben.

Um fehlerhaftes Matching zu erkennen, wurden im Verfahren einige Heuristiken implementiert. Wird eines der folgenden Kriterien verletzt, wird der durch das Matching ermittelte Poseversatz verworfen und das Verfahren mit der ursprünglichen Poseschätzung der Odometrie vorgesetzt. Das Verwerfen einzelner Updates kann in der Regel gut durch die hohe Bildwiederholungsrate der Kamera kompensiert werden, da der Versatz zwischen zwei Frames klein ist.

Zur Bewertung des Matchings werden folgende Kriterien herangezogen:

- Anzahl der Iterationen. Hat der ICP-Algorithmus keine Iterationen durchgeführt, wurden keine Punkt-Punkt-Korrespondenzen gefunden. Es konnte also kein Versatz bestimmt werden.

- Anzahl der Punkte in den Kameradaten. Wurden aufgrund der Rauschfilterung zu viele Punkte entfernt, wird das Ergenis verworfen. Empirisch hat sich bewährt, Punktwolken zu verwerfen, aus denen mehr als die Hälfte der Punkte entfernt wurden.

- Anzahl der gefundenen Punkt-Punkt-Korrespondenzen. Ist die Anzahl der gefundenen Punktpaare deutlich kleiner als die Anzahl der Punkte in der Kamerapunktwolke, wird dem Ergebnis nicht vertraut. Eine solche Situation kann entstehen, wenn in der realen Punktwolke ein Hindernis vorhanden ist, das nicht in der Karte repräsentiert ist.

- Der aufsummierte quadratische Abstand zwischen den Punkt-Punkt-Korrespondenzen ist zu hoch.

- Plausibilität der korrigierten Pose. Liegt die korrigierte Pose innerhalb einer Fläche der Karte, wird sie verworfen, da die Objekte in der Karte als starr und unbeweglich angenommen werden und somit die Pose nicht innerhalb eines Körpers liegen kann.

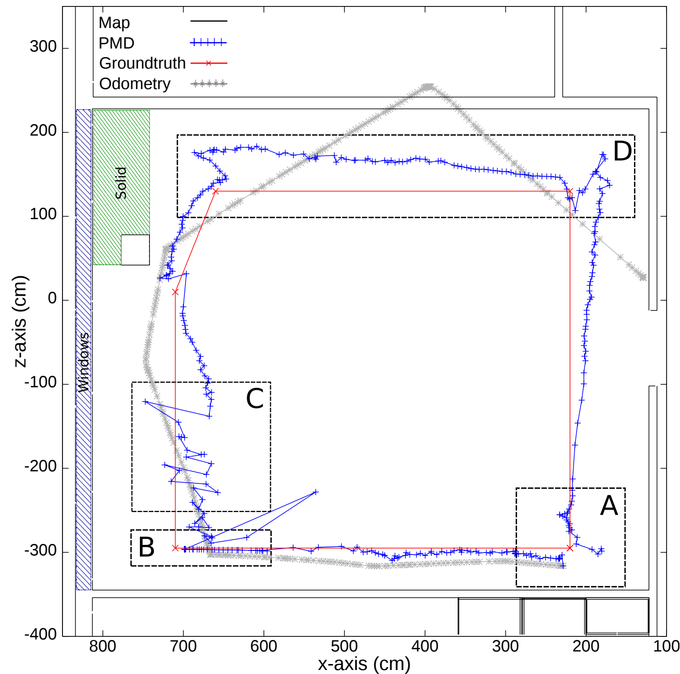


Abbildung 5.8: PMD-Tracking in der Ebene. In einem leeren Seminarraum wurde der rot markierte Pfad abgefahren. Die blauen bzw. grauen Linien zeigen die mittels PMD-Tracking bzw. reiner Odometrie ermittelten Trajektorien.

5.2.3 Evaluation

Zur Evaluation des Posetracking-Verfahrens wurde eine Reihe von Experimenten mit einer an einem Kurt-Roboter befestigten PMD-Kamera durchgeführt. Der Poseversatz zwischen zwei Kameraframes wurde durch Odometrie geschätzt. Im Folgenden werden zwei Experimente vorgestellt. Das erste Experiment wurde in einem leeren Seminarraum durchgeführt. Der Roboter bewegte sich nur in der Ebene. Um zu demonstrieren, dass sich mit dem Verfahren prinzipiell auch 6D-Posen tracken lassen, wurde im zweiten Experiment eine Rampe befahren, d.h. es wurden auch Bewegungen aus der Bodenebene ausgeführt. Die PMD-Kamera war in beiden Experimenten in einem Winkel von ca. 30° am Roboter montiert.

Fahrt im Seminarraum

Das erste Experiment wurde in einem Seminarraum durchgeführt. Als Karte wurde eine bereits vorhandene Polygonkarte der Umgebung verwendet. Diese wies einige Abweichungen von der realen Geometrie auf. Zum einen enthielt sie keine Möbelmodelle (daher wurden alle beweglichen Möbel entfernt), zum anderen waren eine Ecke hinter einer Säule sowie die fest installierte Wandtafel nicht modelliert. Die letzten beiden Abweichungen wurden absichtlich nicht korrigiert, um den Einfluss der Kartengenauigkeit auf das Verfahren zu untersuchen. Weiterhin ist zu er-

wähnen, dass eine Seite des Raumes fast ausschließlich aus Fenstern besteht. Zur Evalution des Tracking-Verfahrens wurde eine Schleife im Raum gefahren. Um Ground-Truth zu erhalten, wurde die zu fahrende Trajektorie mit Klebeband vorgegeben und ausgemessen. Die Gesamtlänge betrug ca. 19m.

Die Ergebnisse des ersten Experimentes sind in Abbildung 5.8 gezeigt. Der Ground-Truth-Pfad ist in rot, die gemessene Odometrie in grau dargestellt. Die blaue Trajektorie wurde mittels PMD-Tracking ermittelt. Man kann deutlich erkennen, dass das PMD-Tracking, bezogen auf die Ground-Truth, ein wesentlich besseres Ergebnis liefert. Auch die Schleife wird annähernd geschlossen. Start und Ende unterscheiden sich um ca. 50cm. An einigen Stellen weicht die Messung aber deutlich von der erwarteten Trajektorie ab.

Im Bereich "B" enthält die gemessene Trajektorie sehr wenige Ausreißer. Dies ist darauf zurückzuführen, dass die Kamera, sobald sie sich diesem Bereich nähert, direkt in die Fenster blickt, und daher die Messungenaugkeiten zunehmen und immer mehr Punkte herausgefiltert werden. Im markierten Bereich waren dann letztendlich nicht mehr ausreichend Punkte für ein Matching vorhanden, so dass die ICP-Ergebnisse verworfen wurden und die Pose ausschließlich durch die Odometrie aktualisiert wurde. Im mit "C" markierten Bereich blickte die Kamera an die gegenüberliegende Wand, so dass ausreichend Punktkorrespondenzen für das ICP-Verfahren gefunden wurden. Allerdings gab es viel Streulicht durch die Fenster und die Messung der entfernten Wand war an der Grenze des erfassbaren Bereiches der Kamera, so dass die Daten sehr stark rauschten. Des Weiteren wirkte sich der Modellierungsfehler in der Karte auf die geschätzte Trajektorie aus. Am Ende des markierten Bereichs gab es einen konstanten Offset zur Ground-Truth. Ein ähnlicher Effekt zeigt sich in "D". In diesem Abschnitt fuhr der Roboter parallel zur Wandtafel. Da diese nicht in der Karte repräsentiert war, wurde die Position konstant um einen festen Wert falsch eingeschätzt. Während der Drehung am Ende von Abschnitt "D" konnte die Raumecke allerdings positiv gematcht werden, so dass die Trajektorie sich wieder der Ground-Truth annäherte. Die Schleife wurde nahezu korrekt wiedergegeben (siehe Abschnitt "A"). Dieses Experiment wurde mehrmals wiederholt, die Ergebnisse waren reproduzierbar (siehe [208]).

Die während des Trackings ermittelten Höhenwerte sind in Abbildung 5.9 dargestellt. Während der Fahrt schwankte die gemessene Höhe. Dies war erwartet, da die Decke des Raumes in den Kameradaten aufgrund des geringen Öffnungswinkels der Kamera nicht zu sehen war. Zudem fuhr der Roboter die meiste Zeit parallel zu einer Wand. Der Offset der Daten in der Ebene konnte daher gut durch ICP ausgeglichen werden. Da in der Höhe aber kaum Merkmale vorhanden waren, konnten die Punktwolken dabei in dieser Richtung gegeneinander verschoben werden, ohne dass ein großer Fehler entstand, so dass die Höhenschätzung leicht variierte. Die Standardabweichung der Messwerte in der Höhe betrug dabei 12 cm. Die in der Abbildung gezeigten Werte beinhalten einen konstanten Offset von 1.3 m, da die Bodenebene in der Karte um 1 m nach unten verschoben war und der Offset der Kamera zum Roboter nicht berücksichtigt wurde.

Rampenexperiment

Das zweite Experiment bestand darin, den Roboter über eine Rampe fahren zu lassen, so dass sich die Blickrichtung aus der Ebene heraus bewegte. Dadurch sollte gezeigt werden, dass sich

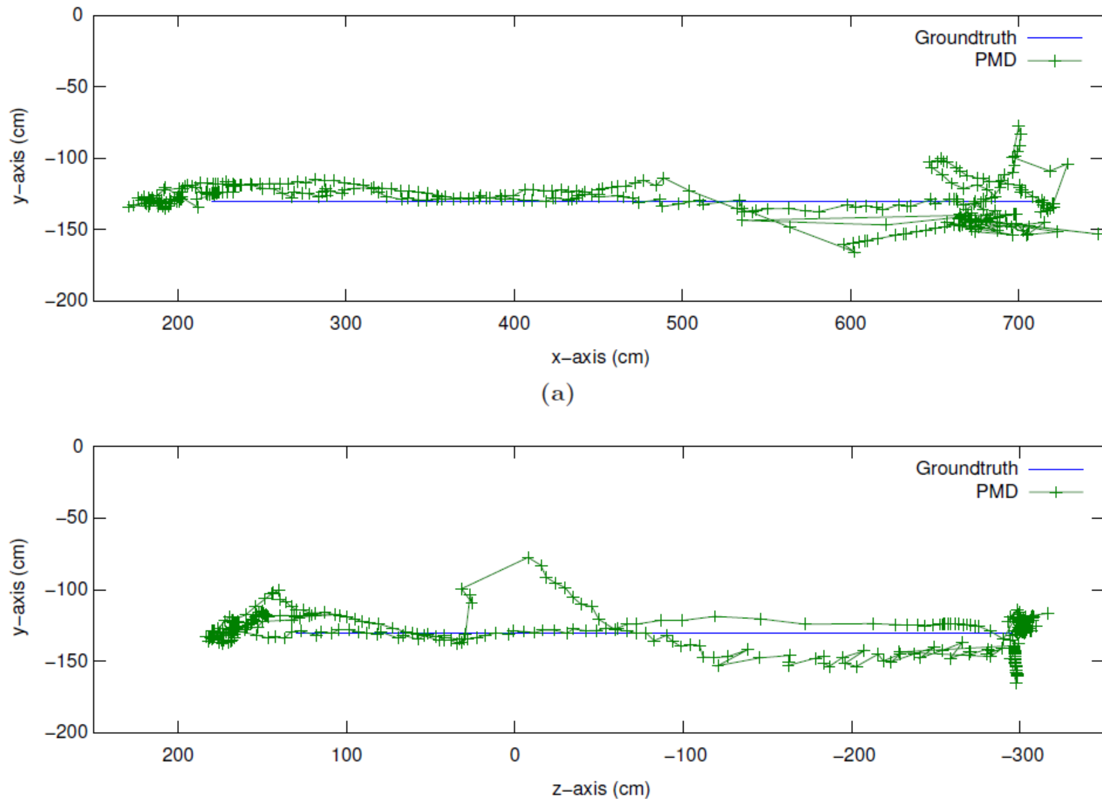


Abbildung 5.9: Evaluation der PMD-Höhenschätzung. Die Werte sind um 1,3m verschoben, da die Bodenebene in der Karte bei -1m lag und der Offset der Kamera zum Roboter in den Messwerten nicht korrigiert wurde.

das Verfahren auch auf mehr Freiheitsgrade erweitern lässt. Den Aufbau des Experimentes und die Ergebnisse der Messungen zeigt Abbildung 5.10. Im ersten Bereich ("A") wurde der Offset der Kamera nicht erkannt, die Höhenschätzung ist konstant zu niedrig. Sobald der Roboter die Rampe befährt ("B"), entspricht die Trajektorie annähernd der Ground-Truth, da Punkte an der Decke gesehen werden und die erwarteten und gemessenen Daten dementsprechend gut aufeinander abgebildet werden können. Am Ende der Fahrt ohne Rampe ("C") werden hauptsächlich Punkte an der Wand gemessen. Es tritt ein ähnlicher Effekt wie im vorherigen Experiment auf: Die Höhenwerte schwanken, da keine Merkmale zu erkennen sind und erwartete und gemessene Daten daher in verschiedenen Höhen gut matchen.

Bei diesen Experimenten erfolgte das Tracking der Pose der Kamera in Echtzeit, d.h. die Auswertung der Daten erfolgte zwischen zwei Kameraframes. Um eine bessere Datenqualität zu erreichen, wurde eine relativ hohe Integrationszeit an der Kamera eingestellt. Im Mittel wurden drei Frames pro Sekunde verarbeitet. Die meiste Laufzeit benötigte die Simulation der Referenzpunktwolke mittels Raytracing. Diese ist durch die Verwendung der AABB-Suchbaumstruktur logarithmisch in der Anzahl der Polygone der verwendeten Karten. Daher ist es für die Laufzeit des Verfahrens von entscheidender Bedeutung, eine bezüglich der Polygonzahl optimierte Karte

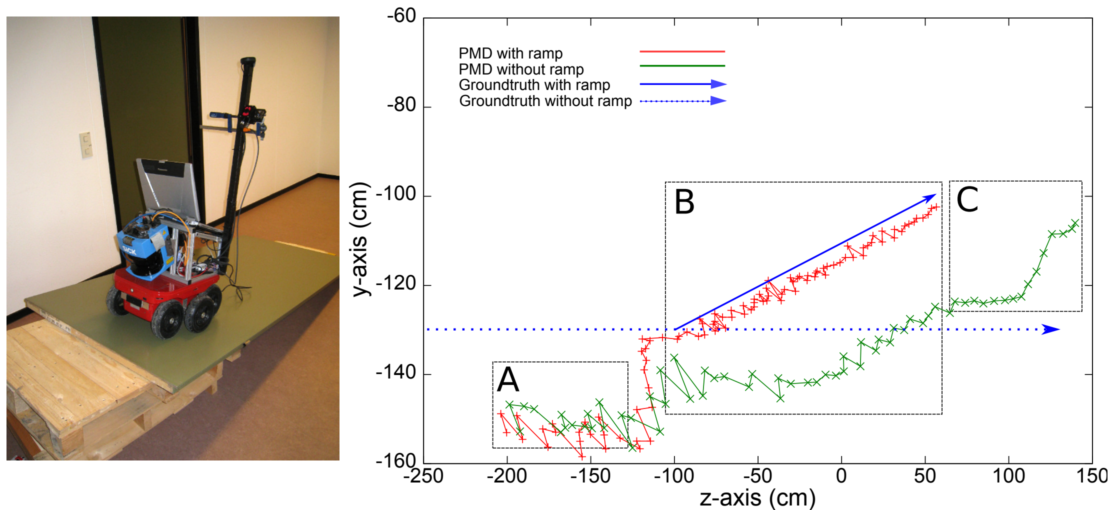


Abbildung 5.10: Aufbau und Ergebnisse des Rampenexperiments. Die gezeigten Trajektorien zeigen den gemessenen Höhenverlauf beim Befahren der Rampe. Zu Beginn der Fahrt (Bereich *A*), wurde die Höhenkoordinate initial falsch geschätzt. Sobald die Raumdecke in des Sichtfeld der Kamera kommt, kann diese Fehlschätzung ausgeglichen werden (Bereich *B*).

zu verwenden. Die für das ICP-Matching benötigte Zeit war - verglichen mit der zum Raytracing benötigten Zeit - vernachlässigbar.

5.2.4 Zusammenfassung

Die vorgestellten Experimente haben gezeigt, dass es mit dem vorgestellten Verfahren prinzipiell möglich ist, 6D-Posetracking einer 3D-Kamera zu realisieren. Im Gegensatz zur standard Monte-Carlo-Lokalisierung wurde dazu keine zweidimensionale Rasterkarte, sondern eine dreidimensionale Polygonkarte verwendet. Zudem erfolgte die Poseschätzung zunächst monomodal, d.h. es wurde nur eine Posehypothese aufrecht erhalten. Die Integration eines Partikelfilters in das bestehende Verfahren ist prinzipiell möglich, allerdings wird dazu ein geeignetes Vergleichsmaß benötigt, um zu bestimmen, welche synthetische Modellpunktwolke an einer möglichen Pose am besten zu den Messdaten passt. Darüber hinaus muss in zukünftigen Arbeiten evaluiert werden, wie man einen Partikelfilter in 6D geeignet umsetzt.

Die wesentliche Neuerung des Verfahrens ist, eine 3D-Polygonkarte in Kombination mit einer 3D-Kamera zur Lokalisierung zu verwenden. Mit Hilfe dieser kompakten Umgebungsrepräsentation konnten die erforderlichen Berechnungen zur Generierung der hypothetischen Sensordaten an der geschätzten Pose effizient durchgeführt werden. Die Experimente haben ergeben, dass das Verfahren robust gegen Ungenauigkeiten in der Karte ist. Die Evaluation hat gezeigt, dass das Hauptproblem das sehr enge Sichtfeld und die begrenzte Reichweite der verwendeten PMD-Kamera ist. Durch diese Einschränkungen war es oft nicht möglich, die zum besseren Matchen benötigten geometrischen Merkmale in der Umgebung zu erfassen. Das Problem des eingeschränkten Sichtfeldes ließe sich durch die Verwendung einer Kinect-Kamera lösen. Diese

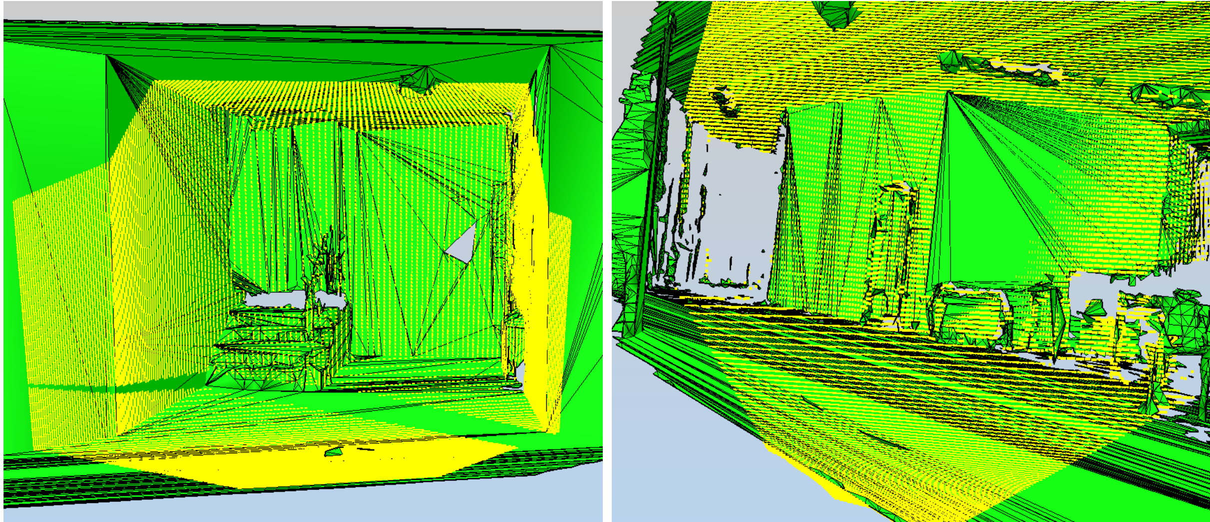


Abbildung 5.11: Simulation einer Kinect-Punktwolke in LVR-Rekonstruktionen.

hat mit $57° \times 43°$ ein wesentlich größeres Sichtfeld. Mit Hilfe eines ähnlichen Lochkameramodells wie für die PMD-Kamera lassen sich auch für diesen Sensor synthetische Punktwolken generieren (siehe Abbildung 5.11), eine ausführliche Evaluation mit diesem Sensor steht allerdings noch aus. Es ist aber zu erwarten, dass das Problem des stark begrenzten Blickwinkels mit diesem Sensor nur abgeschwächt wird, und die prinzipiellen Probleme in Merkmalsarmen Umgebungen auch bei Verwendung einer Kinect anstelle einer PMD Kamera erhalten bleiben.

5.3 Semantische Interpretation

Für viele Anwendungen in der Robotik sind neben der reinen Umgebungsgeometrie auch zunehmend semantische Informationen über die Umgebung von Interesse. Im Folgenden wird als drittes Einsatzbeispiel für Polygonkarten ein Verfahren zur Identifikation von bekannten Möbelinstanzen in 3D-Laserscans präsentiert. Die Darstellung wurde aus den beiden bisher veröffentlichten Publikationen zu diesem Verfahren übernommen([5] und [75]).

5.3.1 Motivation

In semantischen Karten werden in einer Umgebung wahrgenommene Objekte mit einer Bedeutung versehen, indem sie als Instanzen bestimmter Klassen identifiziert werden, wodurch sie mit vorhandenem Hintergrundwissen verknüpft werden, gemäß der Definition aus [80]:

> Eine semantische Karte ist eine Roboterkarte, die zusätzlich zu metrischen und geometrischen Informationen Zuordnungen von kartierten Strukturen und Objekten zu bekannten Konzepten und Relationen enthält. Weiterhin umfasst die semantische

Karte statisches Wissen über diese Konzepte und Relationen, das in einer Wissensbasis abgelegt ist. Inferenzen sind möglich, welche die raumbezogene Information und das statische Wissen über Objekte in der semantischen Karte kombinieren

Diese Verknüpfung von wahrgenommenen Objekten mit vorhandenem Hintergrungwissen ermöglicht es einem Roboter, über seine Umgebung zu schlussfolgern: beispielweise könnte er aus der Tatsache, dass er in seinen Sensordaten eine Spülmaschine und einen Herd sieht, schließen, dass er sich in einer Küche befindet. Des Weiteren kann sich aus der Identifikation der Objekte auch deren Funktionalität erschließen: ein Konferenztisch hat eine andere Funktion als ein Küchentisch.

Die Herausforderung besteht darin, die im Hintergrundwissen modellierten Objektinstanzen in den Sensordaten des Roboters zu erkennen. Der hier vorgestellte Ansatz basiert darauf, aus 3D-Punktwolken zunächst ein Polygonmodell zu erstellen und darin, wie in Kapitel 3.5.4 beschrieben, die vorhandenen Ebenen zu bestimmen. Durch Analyse ihrer geometrischen Eigenschaften (Schwerpunkt, Ausrichtung, Flächeninhalt) sowie der Lage zueinander (zwei Ebenen sind parallel oder senkrecht) sollen Instanzen von bekannten Möbeln in den Daten gefunden werden. Die Beschreibung der Relationen zwischen den Ebenen für verschiedene Referenzmodelle wird in einer OWL-DL-Ontologie modelliert. Basierend auf diesem Hintergrundwissen werden mögliche Positionen für Möbelkandidaten geschätzt. In einem zweiten Schritt werden die Punkte in den Eingangsdaten durch CAD-Modelle der entsprechenden Objekte ersetzt. Dadurch wird sowohl die Qualität der polygonalen Repräsentation erhöht (die vormodellierten CAD-Modelle sind in der Regel genauer als die mittels Marching Cubes erzeugten Rekonstruktionen) als auch eine Interpretation für gemessene 3D-Punkte erzeugt, da bekannt ist, welche Modelle sich wo in der Szene befinden.

5.3.2 Modellbasierte Objekterkennung

Zur modellbasierten Objekterkennung existieren eine Reihe von relevanten Vorarbeiten mit unterschiedlichen Ansätzen. Ein Framework zum Erkennen von Kücheneinrichtung ohne Verwendung von CAD-Modellen wurde von Rusu et al. vorgestellt [164, 165]. Dieser Ansatz benutzt parametrierte kubische Modelle, um die relevanten Objekte zu extrahieren. Lai und Fox [101] benutzen gesamplete CAD-Modelle aus dem Google 3D Warehouse [87], um ein Objekterkennungssystem zu trainieren, das in der Lage ist, die gelernten Modelle in 3D-Laserscans von urbanen Umgebungen zu detektieren. Mian et al. benutzen ein Hashing-Verfahren, um Instanzen von CAD-Modellen in 3D-Punktwolken zu matchen. In [96] wird ein Verfahren vorgestellt, in dem das Matchen von CAD-Modellen benutzt wird, um das Greifen von Objekten in Haushaltsumgebungen zu ermöglichen. Allerdings erfolgt das Matchen der Modelle in Bilddaten und nicht in der 3D Umgebungsrepräsentation. Ein weiteres System zur Objektmanipulation, das Wissensverarbeitung und semantische Interpretation zur Objektmanipulation implementiert, ist in [150] beschrieben.

Neben Verfahren, die eine geometrische Repräsentation, sei es parametriert oder in Form von CAD-Modellen, benutzen, gibt es eine Reihe von merkmalsbasierten Verfahren zur Objekterkennung in 3D-Punktwolken [163, 181, 190]. Ähnliche Ansätze basieren darauf, Objekte in Tie-

fenbildern mit Hilfe von Verfahren aus der Bildverarbeitung zu erkennen. Stiene et al. nutzen eine Eigen-CSS-Repräsentation von Konturen, um Objekte in Tiefenbildern zu erkennen [183]. Nüchter et al. [143] nutzen Haar-Merkmale in Kombination mit AdaBoost [60] zur Objekterkennung in Tiefenbildern. Ein weiteres Verfahren, dass Verschiedene 3D-Descriptoren verwendet, um Modelle aus unterschiedlichen Blickwinkeln zu lernen und zu tracken, wird in [184] vorgestellt.

Das hier vorgestellte Objekterkennungsverfahren verwendet CAD-Modelle von Büromöbeln, die entweder direkt vom Hersteller geliefert wurden oder im Google 3D Warehouse frei zur Verfügung stehen. Die Einschränkung der zu erkennenden Objekte auf Möbel hat diverse Vorteile. Zum einen stehen gibt es von vielen Möbeln bereits CAD-Modelle oder es können ausreichend ähnliche im 3D Warehouse gefunden werden, da die virtuelle Modellierung von Einrichtungen häufig von Architekturbüros zur Visualisierung von Entwürfen verwendet wird. Aus diesem Bereich existieren dementsprechend besonders viele Modelle. Zum anderen bestehen die Oberflächen von Möbeln häufig aus mehr oder minder ebenen Flächen, z.B. Tischplatten, Regalböden, Sitzflächen von Stühlen, so dass diese mit Hilfe der Meshing-Software besonders leicht segmentiert werden können. Ein weiterer wichtiger Aspekt ist, dass Möbel starre Objekte sind, d.h. die Ebenen eines Objektes stehen immer in einer festen Relation zueinander.

Gerade der letzte Punkt ermöglicht es, verschiedene Arten von Möbeln zu identifizieren. So besteht ein Stuhl im Wesentlichen aus einer Sitzfläche und einer Lehne mehr oder weniger senkrecht dazu. Ein Tisch besteht hauptsächlich aus einer waagerechten Fläche in einer bestimmten Höhe. Regale bestehen aus mehreren parallelen Ebenen gleicher Größe. Ähnliche Beschreibungen lassen sich für weitere Arten von Möbeln bestimmen. Der Ansatz zur Möbelerkennung besteht darin, diese charakteristischen Relationen zu identifizieren und entsprechend zu modellieren. Die zur Identifikation relevanten Informationen über Größe, Lage und Ausrichtung der Ebenen in einer Szene werden direkt vom Clustering-Verfahren bei der 3D-Rekonstruktion geliefert.

Ein Teil unserer Methode erfordert, relationales räumliches Schließen zu verwenden. Üblicherweise wird das mit constraint-basierten Kalkülen realisiert [157]. Die Aufgabe hier unterscheidet sich aber vom üblichen Fall insofern, als dass die räumlichen Relationen zwischen den Ebenen als Teil der Definition eines zusammengesetzten Objektes vorgegeben sind. Daher benutzen wir hier Schließen in der Ontologie (Subsumption) dafür. Die Möbelklassen werden in einer OWL-DL-Ontologie modelliert, die für jedes Objekt die relativen Lagen der Ebenen zueinander definiert. Ein weiterer Vorteil dieser Herangehensweise ist, dass mittels SWRL-Regeln [84] numerische Wertebereiche für die modellierten Objekte repräsentiert werden können.

Unser Verfahren kann prinzipiell parallel zusammen mit weiteren Objekterkennungsverfahren, die auf anderen Sensordaten arbeiten, in ein gemeinsames System eingebunden werden. Eine denkbare generische Architektur zeigt Abbildung 5.12. Die verschiedenen Verfahren werten die von ihnen benötigten Sensordaten, z.B. Kameradaten für merkmalsbasierte Bildverarbeitungsverfahren, aus und liefern Objekthypothesen, die in eine gemeinsame Wissensbasis eingetragen werden.

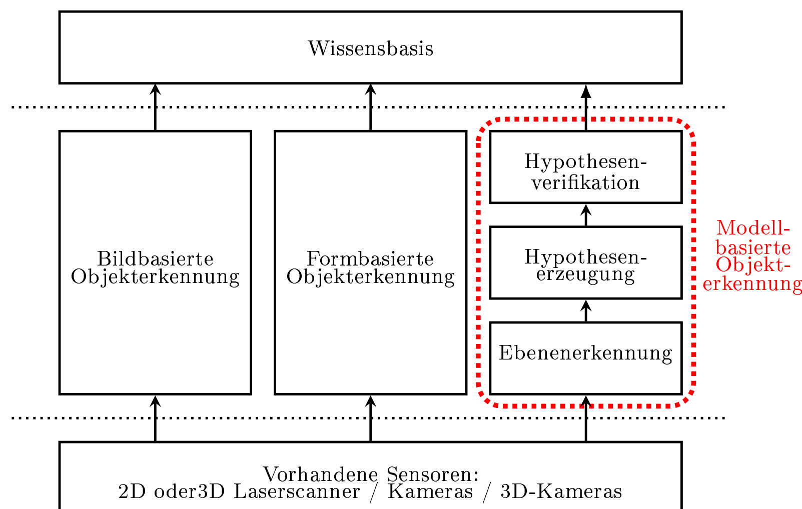


Abbildung 5.12: Schema der Architektur eines Systems zur Kombination verschiedener Oberjekterkennungsverfahren.

5.3.3 Erzeugung von Objekthypothesen aus der Ontologie

Die semantischen Modelle der Möbel werden in einer OWL-DL-Ontologie in Kombination mit SWRL-Regeln modelliert. Diese Wissensrepräsentation wird verwendet, um Hypothesen über die Position und Ausrichtung von vermutlich in der Rekonstruktion vorhandenen Möbeln zu generieren. OWL ("Web Ontology Language") ist ein vom W3C vorgeschlagener Standard für einen Formalismus zur Wissensrepräsentation im Semantic Web. OWL-DL [42] besteht aus drei verschiedenen Untersprachen (OWL-Full, OWL-DL und OWL-Lite). OW-DL entspricht einer Beschreibungslogik, einer Untermenge der Prädikatenlogik erster Stufe, die viele Ausdrucksmöglichkeiten zur Verfügung stellt, gleichzeitig aber Entscheidbarkeit in polynomialer Zeit garantiert. Eine Erweiterung von OWL-DL ist SWRL, "The Semantic Web Rule Language" [84], die unter anderem so genannte "built ins" für arithmetische Berechnungen und Vergleiche zur Verfügung stellt.

Die Modellierung in OWL-DL wurde aus verschiedenen Gründen gewählt: OWL-Ontologien können leicht wiederverwendet werden und mit anderen Quellen von Domänenwissen aus dem Semantic Web verknüpft werden. Darüber hinaus skalieren sie gut und es gibt schnelle Reasoning-Verfahren. In unserer Implementierung nutzen wir den Open-Source-Reasoner Pellet [177], der OWL-DL-Ontologien und SWRL-Regeln vollständig unterstützt.

Einen Ausschnitt aus der verwendeten Wissensrepräsentation zeigt Abbildung 5.13. Die Basisklassen der Repräsentation sind *Furniture* für alle erkannten Möbel und *Plane* für die gefundenen Ebenen, aus den die Objekte zusammegesetzt sind. Jede vom Ebenextraktionsalgorithmus gefunde Ebene wird als eigene Instanz in die Ontologie eingefügt. Zusätzlich werden geeignete Kenngrößen (Höhe der Ebene über dem Boden, Bounding-Box, Schwerpunkt und Flächeninhalt) abgelegt. Außerdem werden für jedes Objekt zwei OWL-Properties abgelegt. Die Eigenschaft *isAbove* wird eingefügt, wenn der in die Bodenebene projizierte Abstand der Schwerpunkte zwei-

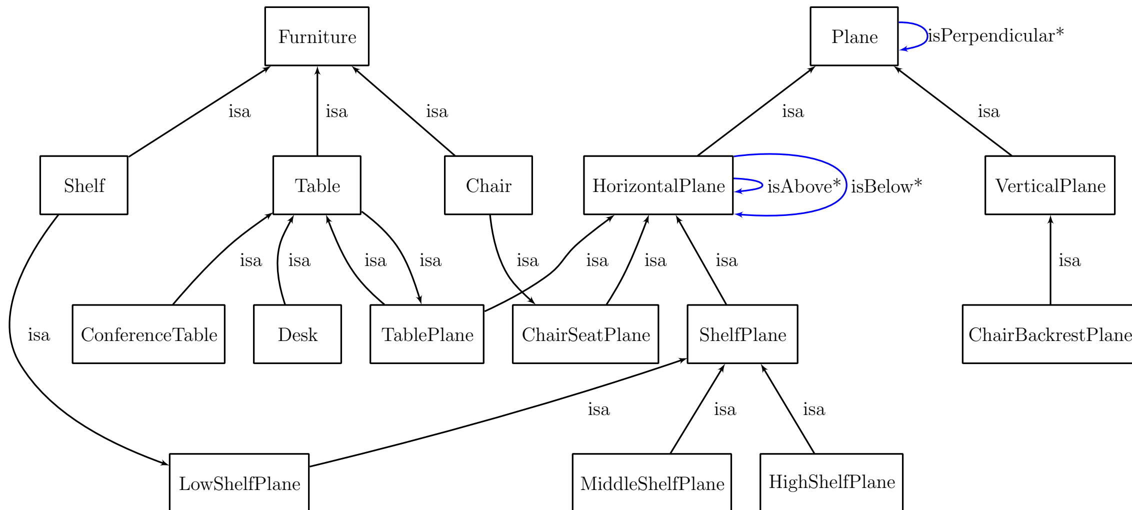


Abbildung 5.13: Ein Ausschnitt aus der Ontologie zur Möbelmodellierung. Im rechten Zweig sind die Relationen zwischen den Ebenen modelliert. Im linken Zweig werden die Relationen der Möbel modelliert.

er Ebenen unterhalb eines bestimmten Wertes ist. Auf diese Art und Weise können ähnlich große übereinander liegende Ebenen (z.B. in Regalen) modelliert werden. Ähnlich wird die Eigenschaft *isPerpendicaular* für senkrecht zu einander stehende Ebenen modelliert. Diese Eigenschaft wird generiert, wenn die Schwerpunkte einer horizontalen und vertikalen Ebene nah genug bei einander sind.

Um bestimmte Instanzen zu finden, werden SWRL-Regeln benutzt. So haben z.B. die meisten Tische eine Höhe von 70 cm. Daher sind alle horizontalen Ebenen, deren Flächeninhalt innerhalb bestimmter Grenzen liegt, Kanditaten für Tische. Hier werden alle Ebenen, die einen Flächeninhalt von mehr als einen Quadratmeter haben und sich auf einer Höhe zwischen 65 cm und 85 cm betrachtet:

```
Table(?p) ← HorizontalPlane(?p) ∧ hasSize(?p, ?s)
   ∧  swrlb:greaterThan(?s, 1.0)  ∧ hasPosY(?p, ?h)
   ∧  swrlb:greaterThan(?h, 0.65)  ∧ swrlb:lessThan(?h, 0.85)
```

Analog lassen sich Stühle modellieren: Ein Stuhl besteht aus einer Sitzfläche (ca. 40 cm über dem Boden) und einer Rückenlehne senkrecht dazu. Die Zuweisung der Ebenen ist dabei nicht eindeutig, so kann eine Ebene, die in einer bestimmten Höhe gefunden wird, mehrere Label bekommen. Ein Regal besteht im Wesentlichen aus einer Reihe von parallelen horizontalen Ebenen in bestimmten Abständen. Die untere Ebene eines Regals kann dementsprechend durch folgende Regel definiert werden:

```
LowShelfPlane(?p) ← HorizontalPlane(?p)
   ∧ hasSize(?p, ?s) ∧ swrlb:greaterThan(?s, 0.01)
   ∧ swrlb:lessThan(?s, 0.5) ∧ hasPosY(?p, ?h)
   ∧ swrlb:greaterThan(?h, 0.08)
   ∧ swrlb:lessThan(?h, 0.18)
```

Ein Regel aus drei Brettern kann dann wie folgt repräsentiert werden:

```
Shelf ≡ LowShelfPlane and
   (isBelow some (MiddleShelfPlane and
      (isBelow some HighShelfPlane)))
```

Um Rekonstruktionsfehler und Abweichungen zum CAD-Modell zu berücksichtigen, werden in den Regeln keine festen Werte hinterlegt, sondern Intervalle, in denen sich die ermittelten Kennzahlen bewegen dürfen. Abweichungen vom CAD-Modell können unter anderem dann auftreten, wenn die Möbel z.B. durch Ändern der Höhe von Einlegeböden vom Benutzer modifiziert wurden. Momentan werden die Möbelmodelle in der Ontologie von Hand modelliert. Es ist aber denkbar, die Relationen in Zukunft automatisiert aus CAD-Modellen zu gewinnen.

5.3.4 Hypothesenverifikation

Die Verifikation der generierten Objekthypothesen erfolgt durch Vergleich der CAD-Modelle mit der aufgenommenen Punktwolke. Dazu werden die Oberflächen der CAD-Modelle zu synthetischen Punktwolken gesampled, die mittels ICP gegen die Eingangsdaten an der vermuteten Pose gematcht werden. Das Samplen der CAD-Modelle erfolgt, indem Punkte mit einer vorgegebenen Dichte innerhalb der Dreiecksdefinitionen der CAD-Meshes berechnet werden [5]. Der mittels ICP ermittelte quadratische Fehler ergibt ein grobes Maß für die Passgenauigkeit zwischen CAD-Modell und Punktwolke. Ist der Fehler nach Konvergenz zu groß, wird die Modellhypothese verworfen. Ein Vorteil dieses Verfahrens ist, dass leicht ungenaue Posehypothesen für die Modelle durch ICP korrigiert werden können. Konvergiert der Algorithmus unterhalb des Toleranzschwellwertes, wird die durch ICP korrigierte Pose für das entsprechende Modell übernommen.

Diese Betrachtung wird nur einige der falschen Hypothesen verwerfen. Es lassen sich viele Fälle konstruieren, in denen trotz falscher Hypothese der quadratische Punkt-zu-Punkt-Fehler zwischen Modell und Punktwolke sehr klein wird, obwohl das entsprechende Modell nicht in der Szene vorhanden ist. So wird die Rückseite eines Regals immer gut gegen eine leere Wand matchen, so dass, je nachdem wie viele Punkte auf der Rückseite liegen, der relative Fehler dennoch klein ist. Daher wurde ein weiterer Ansatz implementiert. Anstatt die gesamplete Punktwolke gegen die Daten zu matchen, wird untersucht, wie gut die Punktwolke die Form des Modells in der finalen Pose wiedergibt. Dazu wird das Modell in ein Voxelmodell diskretisiert. Für jeden Voxel des Modells wird bestimmt, wie viele Punkte der Punktwolke in ihm liegen. Ist diese Anzahl größer als ein bestimmter Schwellwert, wird angenommen, dass dieser Teil des Modells von der Punktwolke wiedergegeben wird. Anschließend wird das Verhältnis von durch die Punktwolke gestützten Voxeln zu nicht unterstützten Voxeln bestimmt. Werden zu wenige Voxel gestützt, wird die Hypothese verworfen. Im Regalbeispiel würde dieses Maß nur die Voxel der Rückseite werten. Voxel, die von den Seitenteilen und Einlegeböden erzeugt wurden, würden nicht gestützt, so dass die Hypothese mit geeignetem Schwellwert verworfen würde. Diese Heuristik liefert nur ein grobes Maß zur Passgenaugkeit. Das Auffinden einer besseren Methode ist Teil der aktuellen Forschung.

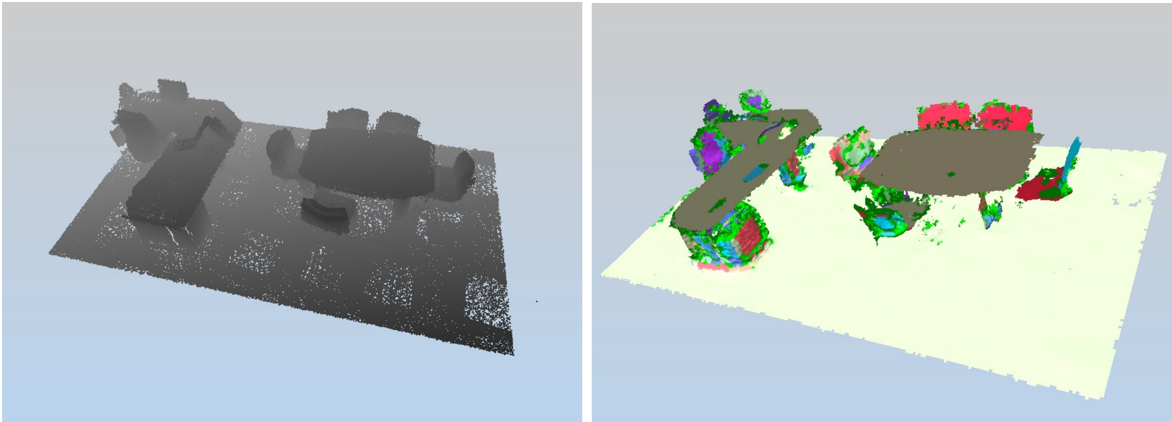


Abbildung 5.14: 3D-Punktwolke der Rekonstruktion (rechts)Büroumgebnung, aufgenommen mit einem rotierenden SICK-Laserscanner (links) und die geclusterte Rekonstruktion (rechts).

5.3.5 Experimente

Für die hier vorgestellten Tests wurden CAD-Modelle des Möbellieferanten der Universität Osnabrück verwendet. Es wird ein Beispiel gezeigt, in dem mit Hilfe unserer Methode zwei unterschiedliche Tische vollautomatisch in 3D-Laserscans gefunden wurden. Zunächst wird die Erzeugung der Objekthypothesen vorgestellt. Anschließend wird demonstriert, dass diese Schätzungen durch ICP verbessert und verifiziert wurden. Im dritten Abschnitt wird diskutiert, in wieweit die Ebenenextraktion robust gegen Abschattungen ist. Zuletzt wird die Erkennung auf einer reihe von Kinect-Frames getestet.

Hypothesengenerierung

Zur Evaluation unserer Objekterkennung wurde ein Büro mit einem rotierenden SICK LMS 200 Laserscanner vermessen. Der aufgenommene Datensatz besteht aus insgesamt 32 Scans, die mittels `slam6d` [141] zu einer Gesamtpunktwolke registriert wurden. Die dazu benötigten Poseschätzungen wurden mittels Maßband geschätzt. Aus dieser Punktwolke wurde ein Dreiecksnetz generiert, in dem anschließend die ebenen Anteile geclustert wurden (vgl. Abbildung 5.14). Die gefundenen Ebenen sind in einem Falschfarbengradienten gerendert, die nicht ebenen Anteile des Meshes in grün. Zur besseren Darstellbarkeit wurde die Szene manuell auf den gezeigten Bereich begrenzt.

Man kann deutlich erkennen, dass in der Rekonstruktion die ebenen Anteile sehr gut segmentiert wurden. Fußboden (weiß), Tischplatten (grau) sowie die Sitzflächen und Rückenlehnen der Stühle (rot, blau, hellgrün) wurden als zusammenhängende ebene Cluster erkannt. Damit die leicht gewölbten Rückenlehnen der Stühle noch segmentiert werden können, müsste der Schwellwert für das Normalenkritierium relativ weit auf 0.8 (entspricht ca. 37°) heruntergesetzt werden. Die benötigten Kennzahlen dieser Flächen wurden in eine Datei geschrieben, die anschließend vom Reasoner verarbeitet wurde. Dabei wurden die beiden Tischplatten als mögliche Kandidaten für

Tabelle 5.1: Rotation des Tischmodells und berechnete Orientierung mittels PCA.

Ground Truth	12.0°	25.0°	55.0°	90.0°	125.0°	160.0°
Besprechungstisch	8.2°	22.1°	51.4°	86.0°	121.0°	157.0°
Schreibtisch	4.0°	28.1°	46.7°	82.0°	118.0°	153.0°

in der Szene vorhandene Möbel erkannt. Da von der Software nur die achsenparallele Boundingbox ermittelt wird, haben wir die Orientierung der Tischplatten mittels PCA bestimmt, um zusammen mit dem Schwerpunkt der ermittelten Daten eine Posehypothese für den Verifikationsschritt zu bestimmen.

Zuvor wurden die zugrunde gelegten CAD-Modelle so transformiert, dass deren Referenzpunkt im Schwerpunkt der Tischebenen liegt, damit deren Bezugssystem zu der vom Clusterring erzeugten Darstellung passt. Das Problem, die Orientierung mittel PCA bestimmen zu müssen, tritt bei Modellen auf, die in der Ontologie nur aus einer horizontalen Referenzebene bestehen. Bei Möbeln, die auch senkrechte Anteile haben, lässt sich deren Pose durch den Schwerpunkt einer senkrechten Ebene und der dazugehörigen Flächennormale repräsentieren. Ein Beispiel dafür sind Stühle. Ihre Orientierung ist durch die Normale der Fläche der Rückenlehne vorgegeben. Alle Modelle, die sich so beschreiben lassen, wurden ebenfalls entsprechend transformiert.

Um die Genauigkeit der PCA zu evaluieren, haben wir ein Referenzmodell des Konferenztisches um verschiedene Winkel gedreht und diese mit den von der PCA ermittelten Werten verglichen. Die Ergebnisse sind in Tabelle 5.1 dargestellt. Obwohl die geschätzten Posen um einige Grad von der Ground Truth abweichen, konnten diese im Verifikationsschritt mittels ICP korrigiert werden, wie im nächsten Abschnitt gezeigt wird. Die Rechenzeit der PCA betrug im Experiment nur wenige hundert Millisekunden und ist somit vernachlässigbar.

Hypothesenverifikation und Modellersetzung

Im Verifikationsschritt werden die gesampleten CAD-Modelle, ausgehend von der zuvor aus der polygonalen Darstellung ermittelten Posehypothese, gegen die Punktwolke gematcht. Die Ergebnisse dieses Prozesses sind in Abbildung 5.15 gezeigt. Die Bilder auf der linken Seite zeigen deutlich, dass die initial geschätzte Pose durch ICP signifikant verbessert werden konnte. Für den Konferenztisch erhalten wir einen fast perfekten Fit. Die Einpassung des Schreibtisches gelingt nicht ganz so gut. Hier gibt es auf der rechten Seite eine Abweichung von ca. 2 cm. Diese lässt sich durch Registrierungsfehler in der Punktwolke und Unterschiede zwischen CAD-Modell und dem realen Objekt erklären. So sind im realen Objekt Spalten zwischen den Tischplatten vorhanden, die im CAD-Modell fehlen.

In beiden Fällen war der Fehler der Registrierung allerdings sehr gering, so dass die Objekthypothesen an diesen Stellen verifiziert wurden. Durch den ICP-Schritt konnten, wie erwartet, die zuvor geschätzten Initialposen deutlich verbessert werden. Durch die Verankerung der CAD-Modelle in den Ausgangsdaten wird semantisches Wissen über die aufgenommene Punktwolke

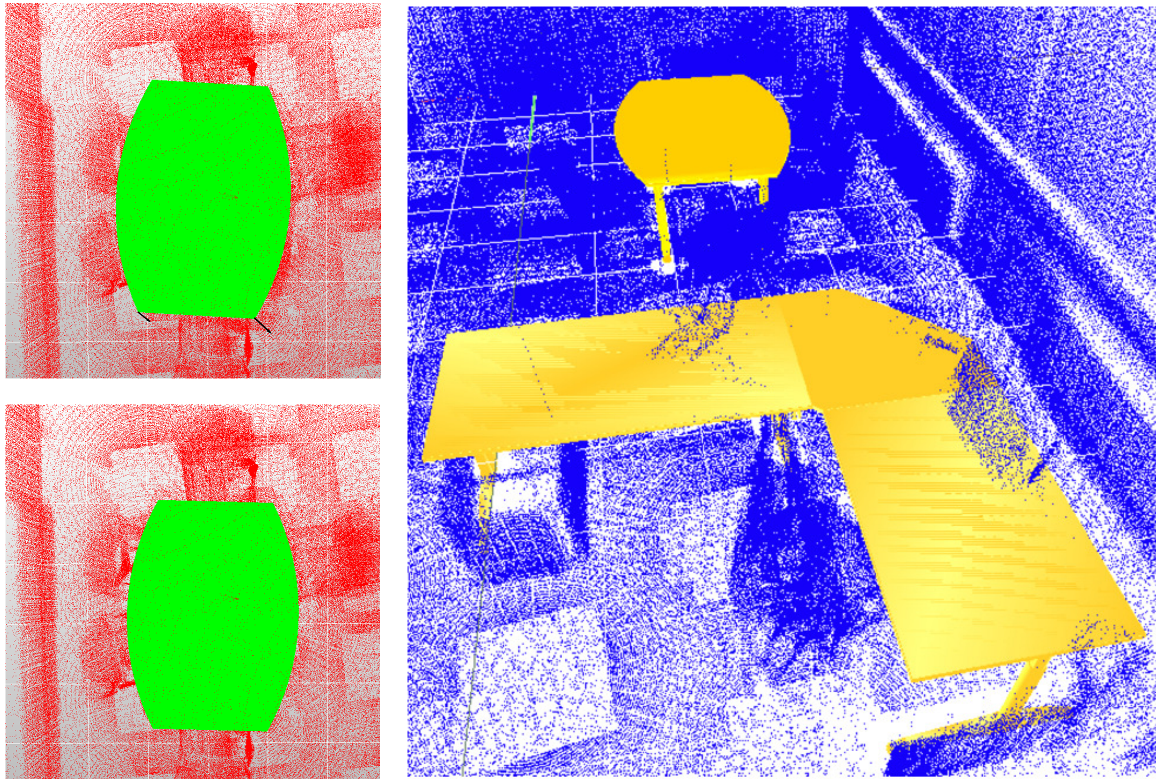


Abbildung 5.15: Ergebnisse des Matchings der zwei Tischmodelle mittels ICP. Die linke Spalte zeigt die Posen des Konferenztisches vor und nach dem Matchen. Die schwarzen Pfeile markieren den Unterschied zwischen initial geschätzter und gematchter Pose. Das Bild auf der rechten Seite zeigt die verwendeten CAD-Modelle an der finalen Pose im Mesh gerendert.

gewonnen, nämlich dass sich in der Szene ein Schreibtisch und ein Konferenztisch befinden. Darüber hinaus können die aufgenommenen Sensordaten aufgebessert werden, indem die Punkte auf den Oberflächen durch die gesampleten Punkte aus dem CAD-Modell ersetzt werden. In der polygonalen Rekonstruktion brauchen die Punkte, die CAD-Modelle repräsentieren, nicht mehr mittels Marching Cubes vermesht zu werden, da die Flächenbeschreibung dieser Objekte direkt aus dem CAD-Modell entnommen werden kann.

Robustheit der Rekonstruktion gegen Abschattungen

Ein wichtiger Aspekt für die praktische Anwendung des Verfahrens ist, dass die zur Segmentierung benötigten Oberflächen im Alltag nicht leer sind, sondern verschiedene Objekte auf den Oberflächen stehen. Durch die so entstehenden Abschattungen entstehen Löcher in den Rekonstruktionen, die die detektierte Fläche vermindern. Da der Flächeninhalt eine für die Identifizierung wichtige Größe ist, wurde für diese Anwendung ein alternativer Ansatz zum Schließen von Löchern in den extrahierten Ebenen implementiert. Anstatt kleine Konturen durch Edge-Collapses zu schließen, wurde lediglich die äußere Kontur eines ebenen Clusters trianguliert.

Tabelle 5.2: Rekonstruierte Flächen beim Tischexperiment nach dem Hinzufügen von mehreren Objekten. Die jeweils erste Zeile zu einem Verfahren zeigt die absolut ermittelte Tischfläche in m^2. Die zweite Zeile zeigt den relativen Anteil zur wirklichen Fläche von $1.6\,m^2$.

Anzahl Objekte	0	2	3	4	5	6	7	12
Region Growing	1.50	1.47	1.47	1.40	1.35	1.26	1.17	0.95
	93%	92%	92%	87%	84%	79%	73%	59%
Konturtriangulation	1.50	1.50	1.49	1.52	1.52	1.52	1.50	1.20
	93%	93%	93%	95%	95%	95%	93%	75%

Dadurch werden automatisch alle Löcher innerhalb der Fläche überdeckt.

Für den hier gezeigten Anwendungsfall macht diese Vorgehensweise Sinn, da Möbelflächen normalerweise keine Löcher aufweisen und man daher davon ausgehen kann, dass alle Konturen innerhalb der äußeren Begrenzung durch Abschattungen von Objekten hervorgerufen wurden. Im Allgemeinen gilt diese Annahme nicht, z.B. sollten Fenster innerhalb von Wänden nicht geschlossen werden. Dies kann im Edge-Collapse-basierten Verfahren durch Wahl eines geeigneten Schwellwertes verhindert werden. Durch die hier gestellte Forderung, dass die Möbeloberflächen löcherfrei sind, ist es aber nicht erforderlich, ein Edge-Collapsing durchzuführen, da die Löcher direkt durch Neutriangulierung geschlossen werden können.

Durch das Region-Growing sollten in den Meshes zusammenhängende Ebenen auch dann geclustert werden, wenn viele Löcher existieren, solange es verbindende Dreiecke in der Ebene dazwischen gibt. Wird die äußere Kontur nicht durch einen Schatten unterbrochen, sollte es stabil möglich sein, mit diesem Ansatz die originale Fläche zu rekonstruieren. Selbstverständlich funktioniert dieser Ansatz in der beschriebenen Weise nur bei freistehenden Objekten (z.B. Tischen). Bei vollgestellten Regalen wird es nur schwer möglich sein, die komplette Kontur eines Einlegebodens zu erkennen, ohne dass die Kontur unterbrochen wird. Es ist geplant, diese Fälle in zukünftigen Arbeiten durch Ausnutzung semantischer Informationen zu behandeln, z.B. dass eine Wand hinter einem Regal weitergeht, ohne dass sie in den Sensordaten vollständig zu sehen ist. So könnte man durch kleine parallele Flächenfragmente vor einer Wand schließen, dass dies die Vorderseiten von Einlegeböden sind.

Die Robustheit des Ansatzes wurde für einen freistehenden Tisch evaluiert, indem nach und nach neue Objekte auf die Tischplatte gestellt wurden. Die Szene wurde dabei aus einer festen Pose mit einer Kinect-Kamera aufgenommen und anschließend rekonstruiert. Nach jedem Schritt wurde die Tischplatte segmentiert und die rekonstruierte Fläche bestimmt. Insgesamt wurden 8 Zwischenschritte aufgenommen. Einige Schritte vom leeren Tisch bis zu dem Punkt, an dem die Tischplatte nicht mehr sauber segmentiert werden konnte, zeigt Abbildung 5.16. Die in den Experimenten ermittelten Flächenwerte sind in Tabelle 5.2 dargestellt. Die tatsächliche Fläche des Tisches betrug dabei $1.6m^2$. In allen Experimenten wurde dieser Wert nicht erreicht. Die maximal rekonstruierte Fläche betrug 95% der Ground Truth. Dies lässt sich dadurch erklären, dass der Tischrand in den Kinect-Daten nicht sehr genau zu erkennen war und diese Kontur in der Rekonstruktion daher einige kleinere "Buchten" aufweist, die den ermittelten Flächenin-

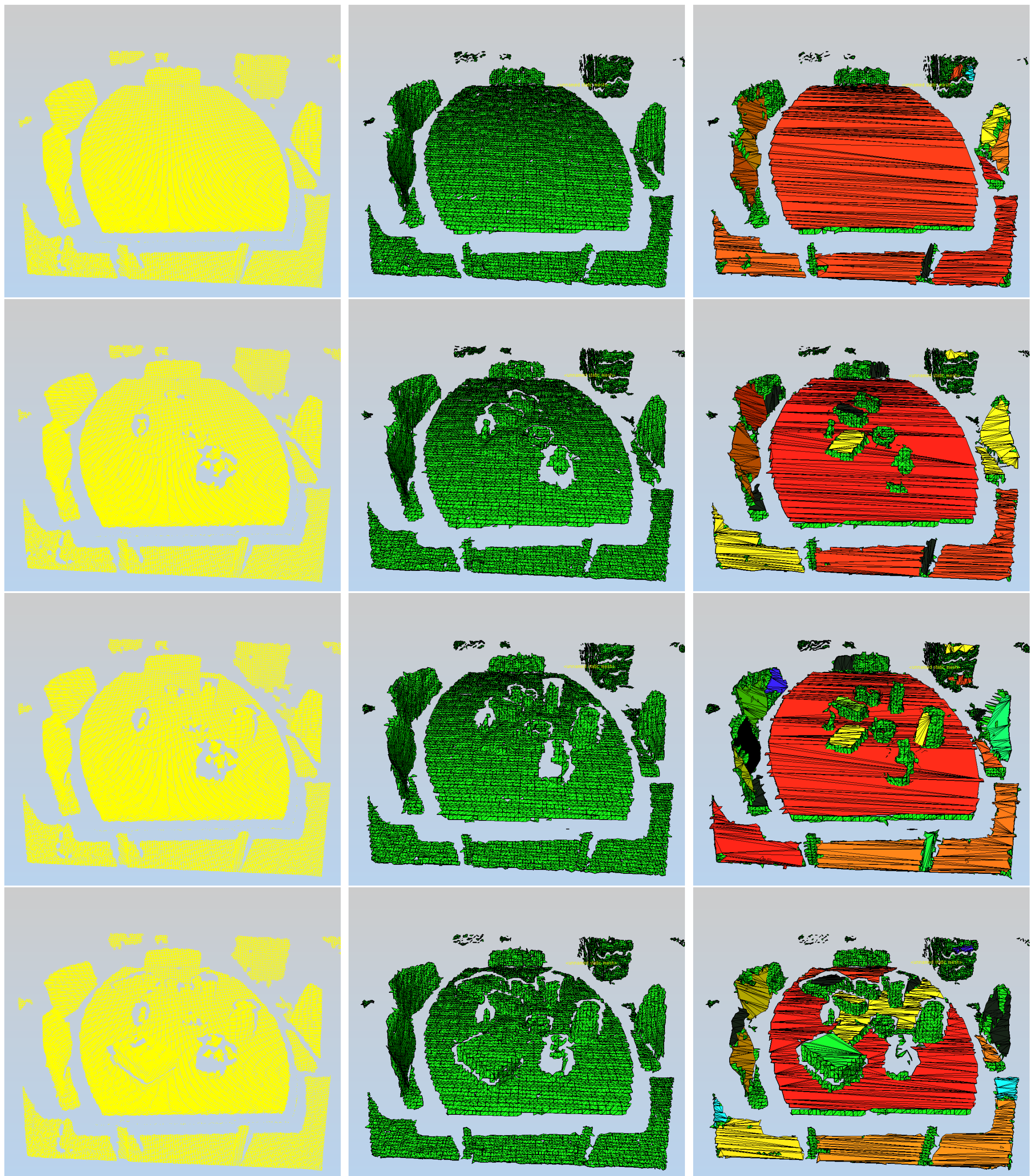


Abbildung 5.16: Segmentierung einer Tischplatte nach dem Hinzufügen von Objekten. Die erste Spalte zeigt die aufgenommenen Punktwolken. Die zweite Spalte zeigt die daraus erzeugten Meshes. Die dritte Spalte zeigt die Segmentierungsergebnisse. Dreiecke, die nicht als nicht-eben klassifiziert wurden, sind in grün gerendert. In jeder Zeile wurde die Anzahl der Objekte auf der Platte erhöht.

Abbildung 5.17: Registrierte Punktwolke der Büroumgebung mit erkannten Möbeln. Der Bildausschnitt unten rechts zeigt die Positionen der erkannten Modelle ohne Punktdaten.

halt verringern. Die ermittelten Werte zeigen deutlich den erwarteten Effekt. Die durch allein durch Region Growing ermittelte Fläche sinkt aufgrund der Verschattungen kontinuierlich, während die durch Retriangulation der Kontur ermittelte Fläche annähernd stabil bleibt. Im letzten Experiment bricht allerdings auch dieser Wert deutlich ein, da ein Objektschatten die Kontur schneidet und dieser somit unterbrochen wird, so dass der ermittelte Flächenwert deutlich zu klein geschätzt wird.

Insgesamt zeigt dieser Versuch, dass sich für freistehende Objekte die Oberfläche mit dem hier vorgestellten Ansatz relativ robust rekonstruieren lässt, so lange die Kontur nicht durch einen Schatten unterbrochen wird. In vielen Anwendungen lässt sich dies durch eine hohe Blickposition der Kamera erreichen. Schatten, die die Kontur schneiden, werden dann vor allem durch Objekte am Rand der Fläche erzeugt. Wirklich problematisch wird es bei "Froschperspektiven" der Kamera. Hier wird dieses Kriterium sehr schnell verletzt.

5.3.6 Objekterkennung in Kinect-Punktwolken

Die in Abschnitt 5.3.5 gezeigten Beispiele wurden in registrierten Laserscans berechnet. Solche Eingangsdaten haben den Vorteil, dass es relativ wenige Abschattungen gibt und die vorhandenen Objekte mehr oder weniger vollständig erfasst wurden. Die Registrierung der Punktwolken erfordert eine gewisse Rechenleistung. Für die Objekterkennung selbst ist keine Registrierung notwendig. Daher soll die Möbelerkennung hier an einem realistischen Online-Szenario getestet werden. Dazu wurde ein Kurt-Roboter mit einer Kinect-Kamera ferngesteuert durch ein Büro gefahren. Die Fahrt dauerte insgesamt 194 s. Zur Reduktion der Datenmenge wurde die Bildwiederholrate der Kinect auf durchschnittlich 2.2 Hz abgesenkt. Insgesamt wurden bei der Fahrt 431 Tiefenbilder aufgenommen. In der Umgebung befanden sich 13 erkennbare Objekte aus 5 Klassen (1 Schreibtisch, 1 Besprechungstisch, 1 Bürostuhl, 5 Einfache Stühle und 5 Regale). Die aufgenommenen Frames der Kinect-Kamera wurden mit Hilfe von `slam6d` registriert.

Zur Evaluation wurde für jeden Frame manuell bestimmt, welche der 13 Objekte zu sehen waren. Die Ground-Truth-Position der Objekte wurden durch manuelles einfügen in die registrierte Szene bestimmt. Zusammen mit der durch Registrierung gewonnenen Trajektorie kann somit die relative Ground-Truth-Position eines Objektes in jedem Frame berechnet werden. Die Registrierung wird hier also lediglich Berechnung der Ground-Truth verwendet. Die Objekterkennung erfolgt auf den einzelnen Frames ohne Poseinformationen. Eine Detektion wird als erfolgreich bewertet, sobald die ermittelte Distanz zur Referenzposition kleiner als ein vorgegebener Schwellwert war. Diese wurde ja nach Größe des zu bestimmenden Objektes angepasst (15 cm für Stühle, 25 cm für Regale und 45 cm für Stühle). Wurden in einem Frame mehrere Objekte entdeckt, wurde nur das mit dem geringsten Abstand gewertet. Die anderen wurde als False-Positives gezählt.

Die Ergebnisses dieses Experimentes sind in Tabelle 5.3 zusammengefasst. Neben den Erkennungsraten sind dort auch die initialen Rotations- und Positionsfehler sowie die korrigierten Werte nach dem CAD-Matching mittels ICP angegeben. Die Orientierungsschätzungen für Tische und Regale wurden mittels PCA ermittelt, die für Stühle basieren auf der Ausrichtung der Rückenlehne. Zusätzlich wurden die Werte auch für die registrierte Punktwolke aller 431 Tiefenbilder bestimmt. Ein Beispiel für detektierte Objekte in der registrierten Gesamtpunktwolke zeigt Abbildung 5.17. Beispiele für erkannte Möbel in den einzelnen Frames zeigt Abbildung 5.18.

Wie erwartet sind die Detektionraten in der kompletten Punktwolke tendenziell höher als in den einzelnen Frames. Dennoch sind die Raten auf einzelnen Frames in Anbetracht der relativ eingeschränkten Sicht in den Eingangsdaten akzeptabel. Erstaunlicherweise hat das ICP-Matching die initiale Poseschätzung auf den Frame-Daten nicht verbessert. Dies liegt wahrscheinlich daran, dass vollständige CAD-Modelle nur gegen unvollständige Repräsentationen der vorhandenen Objekte gematcht wurden. Dieses Problem könnte gelöst werden, indem das eingeschränkte Blickfeld der Kamera und vorhandene Verschattungen beim Sampling der CAD-Modelle - ähnlich wie in [119] beschrieben - berücksichtigt würden.

Tabelle 5.3: Erkennungsraten der Referenzmodelle in den Kinect-Frames. Angeben sind die Detektionen sowie der Posefehler erkannten Instanzen vor und nach der Korrektur mit ICP.

	Positiv		Negativ	Initialer Fehler		Korr. Fehler		Rate	Recall	F_1-Score
	True	False	False	Transl.	Rot.	Transl.	Rot.			
Objekterkennung in einzelnen Frames:										
Regal	82	53	237	16.2 cm	5.1°	60.1 cm	27.2°	60.7 %	25.7 %	36.1 %
Bürostuhl	8	19	6	5.8 cm	61.0°	6.0 cm	10.8°	29.6 %	57.1 %	39.0 %
Stuhl	100	190	114	9.0 cm	51.9°	11.0 cm	56.8°	34.5 %	46.7 %	39.7 %
Schreibtisch	10	11	29	31.6 cm	125.6°	53.7 cm	118.8°	47.6 %	25.6 %	33.3 %
Bespr. Tisch	81	0	0	8.5 cm	8.8°	9.4 cm	6.2°	100.0 %	100.0 %	100.0 %
Gesamt	281	273	386	11.7 cm	28.7°	26.2 cm	34.5°	50.7 %	42.1 %	46.0 %
Objekterkennung in registrierter Punktwolke:										
Regal	1	0	4	24.7 cm	1.0°	131.6 cm	4.1°	100.0 %	20.0 %	33.3 %
Bürostuhl	1	0	0	5.2 cm	68.1°	4.7 cm	14.2°	100.0 %	100.0 %	100.0 %
Stuhl	3	2	2	10.9 cm	55.9°	10.1 cm	49.1°	60.0 %	60.0 %	60.0 %
Schreibtisc	1	0	0	41.8 cm	21.2°	12.8 cm	6.7°	100.0 %	100.0 %	100.0 %
Bespr. Tisch	1	0	0	2.5 cm	4.3°	6.3 cm	0.6°	100.0 %	100.0 %	100.0 %
Gesamt	7	2	6	15.3 cm	37.5°	26.5 cm	24.7°	77.8 %	53.8 %	63.6 %

5.3.7 Zusammenfassung

In diesem Abschnitt wurde ein modellbasiertes Verfahren zur Erkennung von Möbeln in Punktwolken vorgestellt. Die für unseren Ansatz benötigten Informationen über ebene Regionen in den Eingangsdaten wurden aus der polygonalen 3D-Rekonstruktion der Szene gewonnen. Im Gegensatz zu punktbasierten Clustering-Verfahren lässt sich insbesondere der Flächeninhalt der extrahierten Ebenen direkt durch Aufsummierung der einzelnen Dreiecksflächen gewinnen. Andere Verfahren, z.B. Bestimmung der konvexen Hülle, liefern nur dafür approximative Werte. Das präsentierte Experiment hat gezeigt, dass der gewählte Ansatz bei freistehenden Objekten relativ robust gegen Verschattungen ist. Die Auswertung hat ergeben, dass die Möbelerkennung bei vollständiger Umgebungsrepräsentation gut funktioniert. Auch eine Erkennung bei eingeschränkter Sicht ist möglich.

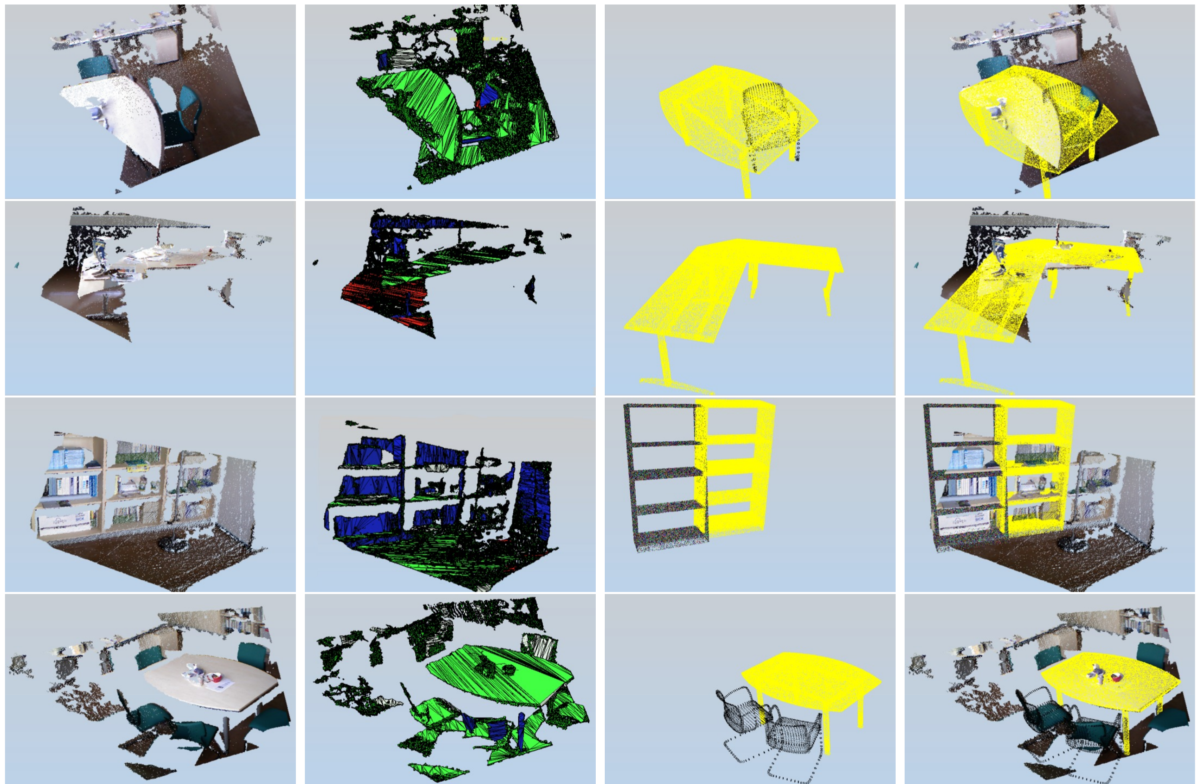


Abbildung 5.18: Modellersetzung in Kinect-Daten. Die Spalten zeigen von links nach rechts: Den verarbeiteten Kinect-Frame, das rekonstrukierte Dreiecksnetz, die gesampleten CAD-Modelle nach der ICP-Posekorrektur und die Punktwolkendaten mit den gefundenen CAD-Modellen. In den ersten beiden Zeilen wurden die Objekte nicht korrekt positioniert, da Teile der Oberflächen verdeckt waren. In den unteren beiden Beispielen wurden die vorhandenen Objekte komplett erfasst und das ICP-Matching konnte die Objekte korrekt platzieren.

Kapitel 6

Zusammenfassung und Ausblick

Im Rahmen dieser Arbeit wurde eine Software zur automatischen Generierung von Polygonkarten aus 3D Punktwolken für den Einsatz in der mobilen Robotik entwickelt. Das vorgegebene Einsatzgebiet setzt voraus, dass die erzeugten Karten verschiedene Anforderungen erfüllen. Ein besonders wichtiger Aspekt ist die Kartengenauigkeit. Da eine Roboterkarte die Grundlage für die Aktionen eines Roboters darstellt, muss die Geometrie der erfassten Umgebung in der Polygonkarte korrekt wiedergegeben werden. In der Regel sind die Ressourcen auf einem mobilen System beschränkt. Um die Karten performant auswerten zu können, sollte die Darstellung möglichst kompakt sein. Modernde 3D-Sensoren sind in der Lage, neben der Geometrie auch weitere Informationen, wie Reflektivität oder Farbe der vermessenen Oberflächen, zu erfassen. Falls vorhanden, sollten diese Informationen in Form von Texturen auf die Polygone übertragen werden. Im Praxiseinsatz müssen die Karten schnell verfügbar sein. Ein weiteres wichtiges Kriterium war daher die Zeit, die benötigt wird, um die Karten zu erstellen. Die in der Robotik verwendeten 3D Sensoren unterscheiden sich signifikant hinsichtlich Messrauschen, Punktdichte und Abstandsgenauigkeit. Mit aktuellen SLAM-Verfahren lassen sich weiträumige Umgebungen vermessen. Daher war es wichtig, dass das entwickelte Verfahren sowohl mit großen Datenmengen zurechtkommt, als auch qualitativ hochwertige Rekonstruktionen aus verrauschten und dünnen Eingangsdaten generiert.

Zur Identifizierung einer geeigneten Technik für die Bewältigung dieser Herausforderungen wurden verschiedene Verfahren zur Oberflächenrekonstruktion aus der Computergrafik betrachtet. Dabei hat sich herausgestellt, dass viele der dort verbreiteten Methoden wie z.B. Delaunay-Triangulation, darauf optimiert sind, geschlossene Objekte zu rekonstruieren. Im vorliegenden Anwendungskontext sind die erfassten Umgebungen in der Regel nicht geschlossen. Daher können solche Methoden zur Lösung des gegebenen Problems nicht angewendet werden. Eine Klasse von Verfahren, die gegebenen Randbedingungen erfüllt, sind Marching-Cubes-Algorithmen. Sie lassen sich speichereffizient implementieren, haben eine sehr kurze Laufzeit und können beliebige Geometrien approximieren. Daher wurden im Rahmen dieser Arbeit verschiedene Marching-Cubes-Varianten implementiert und evaluiert.

Um die gegebenen Anforderungen bezüglich Robustheit gegen Messrauschen und geringer Dichte der Eingangsdaten zu erfüllen, mussten die bekannten Verfahren erweitert und modifiziert werden. So wurde eine adaptive Normalenberechnung entwickelt, die speziell für die Verwendung mit nickenden oder rotierenden 2D Laserscannern optimiert wurde. Damit konnte die Qualität der Rekonstruktionen in dünnen Eingangsdaten deutlich verbessert werden. Zusätzlich wurde das Standard-Marching-Cubes-Verfahren derart erweitert, dass jetzt auch die Konturen von Ebenen korrekt approximiert werden. Weiterhin wurden Filter zum automatischen Entfernen von Artefakten und Löchern, die von Fehlmessungen und Sensorrauschen verursacht werden, entwickelt. Zur Reduzierung der Anzahl der Dreiecke in den Rekonstruktionen wird die Tatsache ausgenutzt, dass die im Bereich der mobilen Robotik vorkommenden Umgebungen überwiegend aus ebenen Flächen bestehen. Die Komprimierung wird erreicht, indem mittels Region Growing alle Ebenen in der Umgebung erkannt werden und deren Kontur in eine optimale Polygondarstellung überführt wird, die anschließend neu trianguliert wird. Die so optimierten Polygone können anschließend durch Übertragen von Zusatzinformationen, wie z.B. Farben, aus den Eingangsdaten texturiert werden. Durch die Verschneidung der detektierten Ebenen können nun auch scharfe Kanten korrekt wiedergegeben werden. Dies war mit den ursprünglichen Marching-Cubes-Varianten nicht direkt möglich.

Die Evaluation der implementierten Verfahren erfolgte auf verschiedenen Datensätzen mit sehr unterschiedlichen Eigenschaften. Ziel war es, einen möglichst breiten Bereich von potenziellen Einsatzgebieten für die Software zu betrachten. Dabei wurden neben hochaufgelösten Laserscans von großräumigen, teilweise nicht-planaren und relativ unstrukturierten Umgebungen, auch Kinect-Punktwolken und SICK-Laserscans von Büros und Fluren, untersucht. Verglichen wurden jeweils die Rekonstruktionsergebnisse von Standard Marching Cubes, Marching Tetrahedrons, Kobbelts Extended Marching Cubes und der hier entwickelten erweiterten Marching Cubes Implementierung für ebene Umgebungen (Planar Marching Cubes). Es hat sich gezeigt, dass Planar Marching Cubes von den betrachteten Varianten bei gleicher Auflösung die besten Approximationsergebnisse liefert. Lediglich Marching Tetrahedrons erbrachte durchgehend bessere Ergebnisse, was aber auf die effektiv höhere Auflösung zurückzuführen ist. Der Fehler lag jeweils im Bereich der Sensorgenauigkeit, d.h. die ursprüngliche Umgebungsrepräsentation wurde durch die Rekonstruktion nicht wesentlich verschlechtert. Auch das Problem der Texturierung konnte erfolgreich gelöst werden. Die erzeugten Texturen werden in den Modellen korrekt zugeordnet und erhöhen die Qualität der Darstellung der Umgebungen beträchtlich. Die Texturierung verbraucht aber je nach Auflösung mit Abstand die meiste Zeit.

Die Laufzeit der geometrischen Rekonstruktion hängt von der eingestellten Auflösung, der Anzahl verwendeter Nachbarn bei der Normalenschätzung und Größe der Eingangsdaten ab. Von wesentlicher Bedeutung ist die Wahl der Bibliothek zur Nächste-Nachbar-Suche. Hier hat sich FLANN als beste Wahl erwiesen. Sie hatte in allen Tests die beste Laufzeit und skaliert sehr gut mit der Anzahl der Prozessorkerne, wodurch sich viele wichtige Schritte bei der Rekonstruktion parallelisieren lassen. Für die Verarbeitung von SICK- und Kinect-Daten lag die benötigte Zeit zur Rekonstruktion je nach Struktur der Umgebung zwischen 0.8 s und 1.5 s. Laserscannerdaten können mit der Software also in Echtzeit rekonstruiert werden.

Im letzten Abschnitt der Arbeit wurde gezeigt, dass sich die erzeugten Karten in der mobilen Robotik einsetzen lassen. Sie wurden erfolgreich in der Simulationsumgebung Gazebo getestet. Mittels eines Raytracing-Ansatzes ist es gelungen, die Pose einer PMD-Kamera unter Zuhilfenahme einer 3D-Polygonkarte in 6 Freiheitsgraden zu tracken. Dies zeigt, dass sich die Mehrinformation über die Umgebung, die in solchen Karten vorhanden ist, leicht ausnutzen lässt, um vorhandene Algorithmen zur Lokalisierung zu erweitern. Als letztes Einsatzbeispiel wurde die Objekterkennung von Möbeln anhand der extrahierten Ebenen gezeigt. Durch die Polygonalisierung konnten die dazu benötigten Informationen, wie Flächeninhalt, Normale und Schwerpunkt einer Ebene, sehr leicht aus den Punktdaten abgeleitet werden. Die 3D-Umgebungsrekonstruktion kann also nicht nur zur Lokalisierung verwendet werden, sondern erleichtert auch das Gewinnen von Informationen zur semantischen Umgebungsinterpretation.

Insgesamt hat sich gezeigt, dass die im Rahmen dieser Arbeit entwickelten Verfahren geeignet sind, die gegebene Problemstellung zu lösen. Mit Hilfe der entwickelten Software lassen sich kompakte texturierte 3D-Polygonkarten aus Laserscans in Echtzeit berechnen. Sie wurden, wie die Anwendungsbeispiele gezeigt haben, bereits erfolgreich im Kontext der mobilen Robotik eingesetzt. Die zukünftige Forschung wird sich vor allem mit den folgenden offenen Problemen befassen:

Verwaltung großer Datenmengen Eine wesentliche Beschränkung des Verfahrens ist, dass zur Rekonstruktion die komplette Punktwolke im Arbeitsspeicher gehalten werden muss. Dies ist bei hochaufgelösten terrestrischen Scans aufgrund der hohen Punktdichte für viele Datensätze problematisch. Sie müssen momentan auf Kosten der Genauigkeit reduziert werden. Derzeit wird daran gearbeitet, passende Out-of-Core Lösungen zu entwickeln, mit denen sich große Datenmengen verarbeiten lassen können, indem nur die zur lokalen Approximation benötigten Daten im Arbeitsspeicher gehalten und nicht benötigte Punkte auf die Festplatte ausgelagert werden.

Konsistente Integration mehrerer Meshes Derzeit lässt sich mit Hilfe der vorliegenden Implementierung immer nur eine einzelne Punktwolke rekonstruieren. Die Eingangsdaten müssen also bereits registriert vorliegen und die Umgebung komplett erfassen. Eine inkrementelle Integration von einzelnen Meshes in eine konsistente globale Karte ist derzeit nicht möglich.

Integration semantischer Informationen Interessant ist auch die Integration von semantischen Informationen in den Rekonstruktionsprozess. Erkannte Instanzen von Objekten ließen sich z.B. durch vormodellierte Meshes ersetzen. Die entsprechenden Punkte müssten also bei der Rekonstruktion nicht berücksichtigt werden. Dadurch würde sich sowohl die Laufzeit weiter verkürzen als auch die Genauigkeit der Karten erhöhen.

Optimierung der Texturen Ein weiteres offenes Forschungsfeld ist die Erzeugung optimierter Texturen für die Modelle. Momentan wird für jedes Polygon eine eigene Textur abgelegt. Das Berechnen der Pixeldaten benötigt zudem sehr viel Zeit. Ziel ist es daher, Texturen wann immer möglich wiederzuverwenden und durch kleine Pattern-Texturen zu ersetzen. Dazu gibt es bereits erste vielversprechende Ansätze [159].

Anhang A

Parameter der Rekonstruktionssoftware

An dieser Stelle werden die wichtigsten Parameter des Programms `reconstruct` des Las Vegas Reconstruction Toolkits wiedergegeben, das für die Rekonstruktion und Optimierung der Meshes verwendet wird. Neben den hier aufgeführten Schaltern existieren noch weitere Einstellmöglichkeiten. Da diese in der Regel nicht verändert werden müssen, sind sie hier nicht aufgeführt. Eine Beschreibung aller Parameter kann durch Verwendung des `--help`-Schalters der Software aufgerufen werden.

Parameter	Erklärung	Vorgabe
`-v`	Voxelgröße des Rekonstrutionsgitters in absoluten Einheiten.	5
`-i`	Anzahl der Unterteilungen entlang der lägsten Seite der Bounding Box des geladenen Datensatz. Kann alternativ zu -v benutzt werden, um die Voxelgröße bei unbekannter Skalierung der Eingangsdaten abzuschätzen.	200
`--kn`	Anzahl der Nachbarpunkte, die zur Normalenberechnung verwendet werden.	10
`--ki`	Anzahl der Nachbarn, die bei der Interpolation der initial berechneten Normalen verwendet werden.	10
`--kd`	Anzahl der Distanzwerte, die bei der Berechnung des Abstands einer Voxelecke zur Isofläche gemittelt werden	10
`--d`	Definiert die zu verwendende Marching-Cubes-Variante. Mögliche Werte sind “MC”, “MT”, “PMC” und “XMC” für Standard Marching Cubes, Marching Tetrahedrons, Planar Marching Cubes und Extended Marching Cubes.	PMC
`--pcm`	Legt die zu verwendende Bibliothek zur Nächste-Nachbar-Suche fest. Mögliche Werte sind “STANN”, “FLANN”, “NABO”, “NANOFLANN”.	FLANN

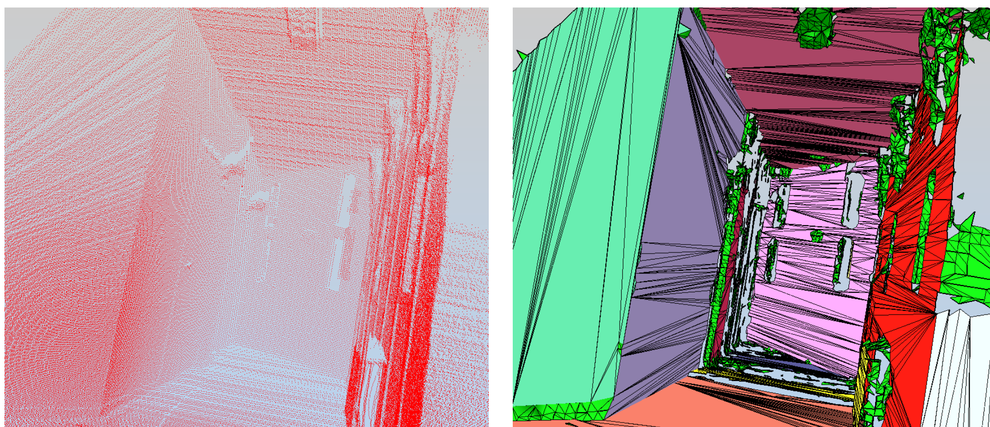
Parameter	Erklärung	Vorgabe
`--ransac`	Wenn dieser Schalter gesetzt wird, werden die Normalen mit Hilfe des vorgestellten RANSAC-Verfahrens berechnet	
`-o`	Aktiviert die Optimierung der ebenen Regionen	
`--rda`	Aktiviert das Löschen von freischwebenden Artefakten. Ein Wert von 0 schaltet den Filter aus.	0
`--srt`	Aktiviert das Löschen von kleinen Regionen im Mesh. Ein Wert von 0 schaltet den Filter aus.	0
`--pnt`	Abbruchkriterium beim Region Growing. Angegeben wird der Kosinus des Winkels. Der Standardwert von 0.85 entspricht dementsprechend einem Maximalwinkel von 30.1°	0.85
`--lft`	Maximaler Anstand eines Vertex beim Zusammenfassen der Konturpolygone der extrahierten Ebenen.	0.01
`-t`	Aktiviert die Neutriangulation der Konturpolygone. Impliziert -o.	
`--gt`	Aktiviert die Erzeugung von Texturen. Impliziert -o und -t.	
`--ts`	Legt die Auslösung der Texturen, d.h. die Größe eines Pixels der Textur auf einem Polygon.	1
`--class`	Legt einen Klassifikator zur farbigen Klassifizierung der ebenen Fest. Mögliche Werte sind “Default”, “IndoorNormals”, “ColorGradient”. Beim Wert “Default” werden alle Dreiecke in einem Grünton eingefärbt. “IndoorNormals” labelt die Flächen gemäß der Orientierung der Normalen. Kann genutzt werden um Wände, Fußböden und Deckenflächen automatisch zu erkennen.	Default
`-e`	Speichert die erzeugten Normalen zusammen mit den ursprünglichen Datenpunkten in der Datei “`pointnormals.ply`”.	
`-g`	Erzeugt eine Datei “`fastgrid.grid`”, die eine Darstellung des Rekonstruktionsgitters enthält. Diese kann mit Hilfe des mitgelieferten Visualisierungsprogramms `qviewer` betrachtet werden.	
`--threads`	Legt die Anzahl der verwendeten Threads bei der Rekonstruktion fest.	8

Anhang B

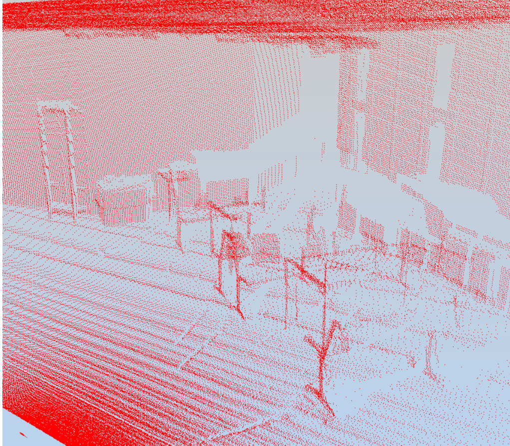
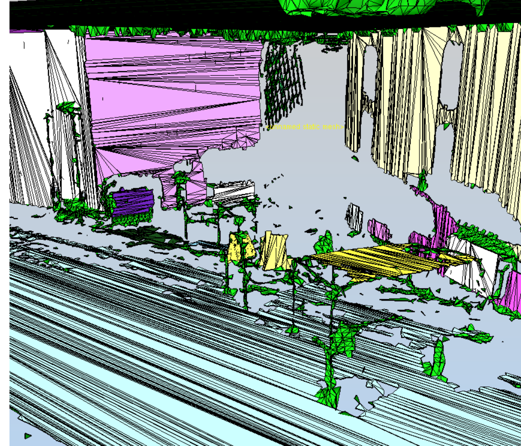
SICK-Laserscans und Kinect-Frames

Die folgenden Bilder zeigen Ausschnitte aus den in Kapitel 4 zur Evaluation verwendeten SICK-Laserscans und Kinect-Frames sowie die daraus berechneten Rekonstruktionen.

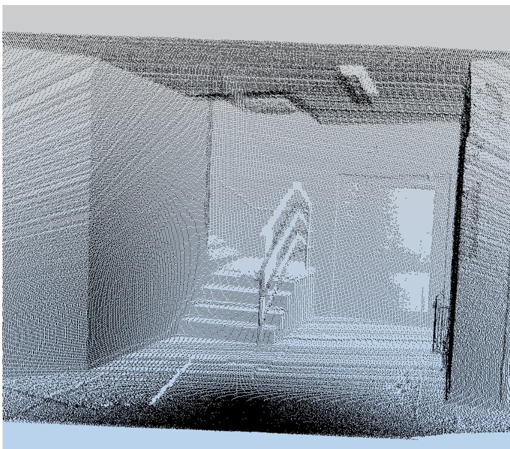
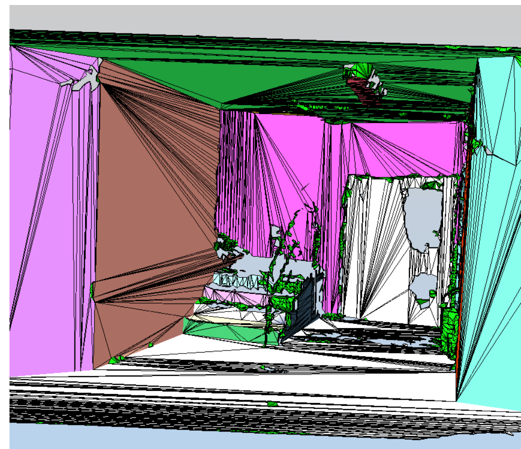
Scan 1



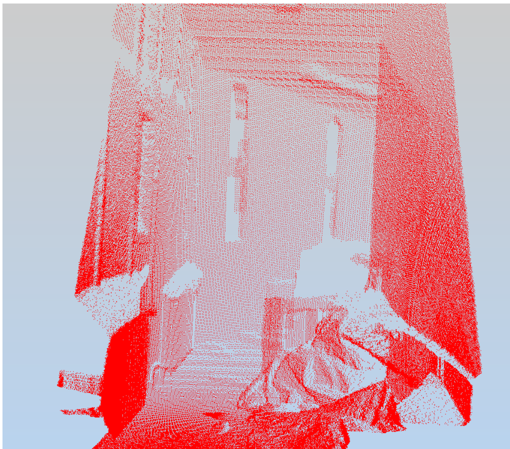
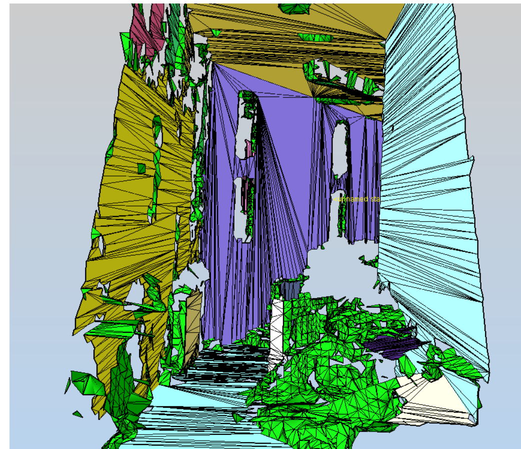
Scan 2

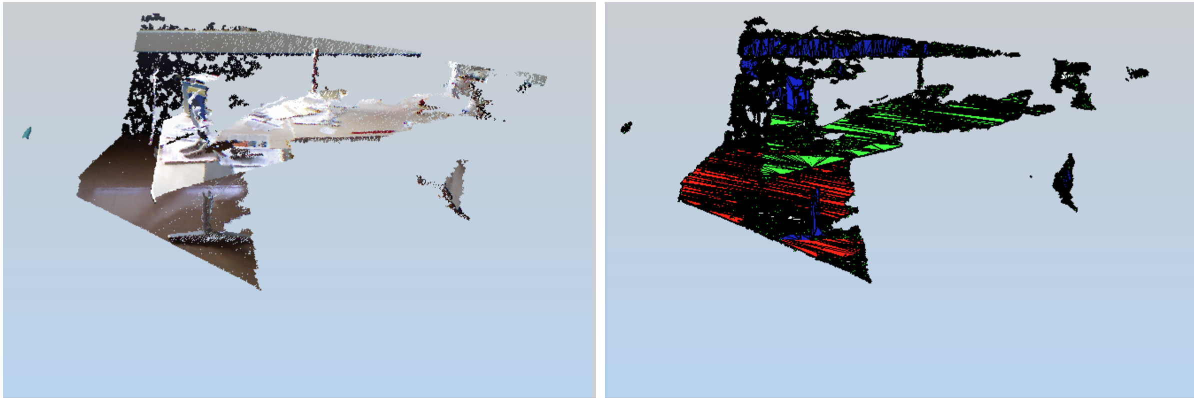
Scan 3

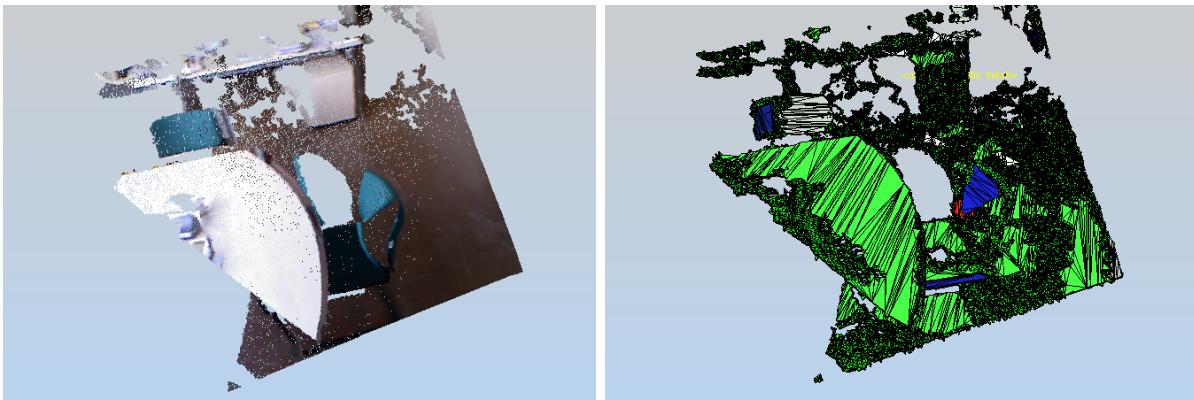
Scan 4

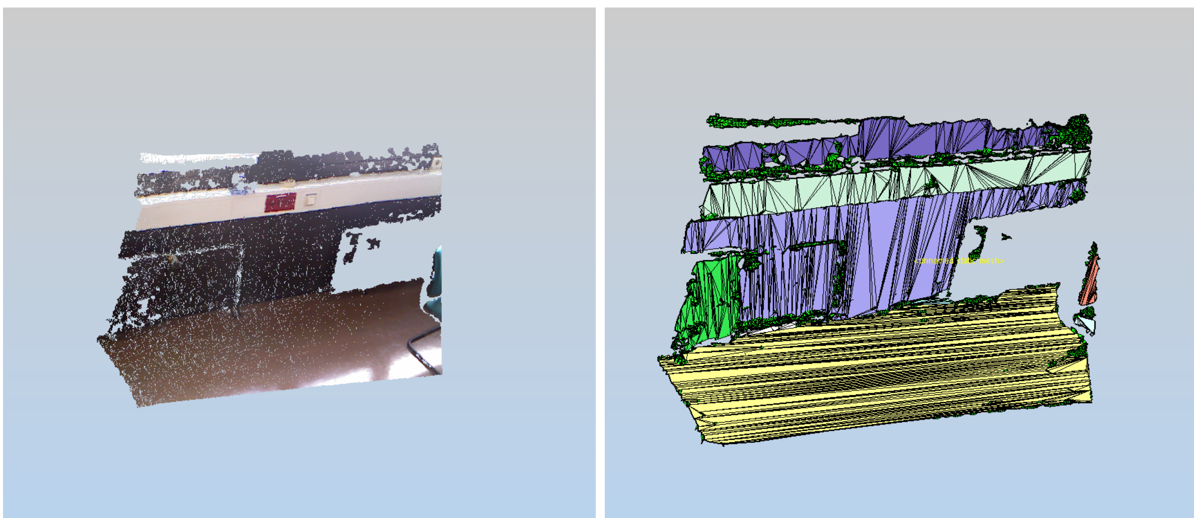
Kinect 1



Kinect 2



Kinect 3



Literaturverzeichnis

[1] *The OpenMP API Specification for Parallel Programming.* 2012. – http://www.openmp.org

[2] A., Sunil; Mount, D. M.: Approximate Range Searching. In: *Proceedings of the 11th Annual ACM Symposium on Compututational Geometry*, 1995

[3] Agrawal, M.; Konolige, K.; Blas, M.: CenSurE: Center Surround Extremas for Realtime Feature Detection and Matching. In: Forsyth, David (Hrsg.); Torr, Philip (Hrsg.); Zisserman, Andrew (Hrsg.): *Computer Vision – ECCV 2008* Bd. 5305. Springer, 2008, S. 102–115

[4] Albrecht, S.; Hertzberg, J.; Lingemann, K.; Nüchter, A.; Sprickerhof, J.; Stiene, S.: Device Level Simulation of Kurt3d Rescue Robots. In: *Third International Workshop on Synthetic Simulation and Robotics to Mitigate Earthquake Disaster (SRMED 2006)*, 2006

[5] Albrecht, S.; Wiemann, T.; Günther, M.; Hertzberg, J.: Matching CAD Object Models in Semantic Mapping. In: *Proceedings of the ICRA 2011 Workshop on Semantic Perception, Mapping and Exploration.* Shanghai, China, May 2011

[6] Alexa, M.; Behr, J.; Cohen-Or, D.; Fleishman, S.; Levin, D.; T. Silva, C.: Computing and Rendering Point Set Surfaces. In: *IEEE Transactions on Visualization and Computer Graphics* 9 (2003), Januar, Nr. 1, S. 3–15

[7] Amenta, N.; Bern, M.; Kamvysselis, M.: A New Voronoi-based Surface Reconstruction Algorithm. In: *Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '98).* New York, NY, USA : ACM, 1998

[8] Amenta, N.; Choi, S.; Kolluri, R. K.: The Power Crust. In: *Proceedings of the 6th ACM Symposium on Solid Modeling and Applications (SMA '01).* New York, NY, USA : ACM, 2001

[9] Annuth, H.; Bohn, C.-A.: Surface Reconstruction with Smart Growing Cells. In: Plemenos, Dimitri (Hrsg.); Miaoulis, Georgios (Hrsg.): *Intelligent Computer Graphics 2010* Bd. 321. Springer Berlin Heidelberg, 2010, S. 47–66

[10] Arya, S.; Mount, D. M.; Netanyahu, N. S.; Silverman, R.; Wu, A. Y.: An Optimal Algorithm for Approximate Nearest Neighbor Searching Fixed Dimensions. In: *Journal of the ACM* 45 (1998), November, Nr. 6, S. 891–923

[11] Baillard, C.; Zisserman, A.: A Plane-Sweep Strategy For The 3D Reconstruction Of Buildings From Multiple Images. In: *ISPRS Journal of Photogrammetry and Remote Sensing*, 2000

[12] Bartsch, M.; Weiland, T.; Witting, M.: Generation of 3D Isosurfaces by Means of the Marching Cubes Algorithm. In: *IEEE Transactions on Magnetics* 32 (1996), may, Nr. 3, S. 1469 –1472

[13] Bartz, D.: Optimizing Memory Synchronization for the Parallel Construction of Recursive Tree Hierachies. In: *Proceedings of the Eurographics Workshop on Parallel Graphics and Visualization (Eurograph 2000)*, 2000

[14] Baumgart, B. G.: Winged Edge Polyhedron Representation. Stanford, CA, USA : Stanford University, 1972. – Forschungsbericht

[15] Bay, H.; Ess, A.; Tuytelaars, T.; Gool, L. van: Speeded-Up Robust Features (SURF). In: *Computer Vision and Image Understanding* 110 (2008), Nr. 3, S. 346 – 359

[16] Berkmann, J.; Caelli, T.: Computation of Surface Geometry and Segmentation Using Covariance Techniques. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 16 (1994), November, Nr. 11, S. 1114–1116

[17] Bernardini, F.; Martin, I.M.; Rushmeier, H.: High-quality Texture Reconstruction From Multiple Scans. In: *IEEE Transactions on Visualization and Computer Graphics* 7 (2001), oct-dec, Nr. 4, S. 318 –332

[18] Bernardini, F.; Mittleman, J.; Rushmeier, H.; Silva, C.; Taubin, G.: The Ball-Pivoting Algorithm for Surface Reconstruction. In: *IEEE Transactions on Visualization and Computer Graphics* 5 (1999), Oktober, Nr. 4, S. 349–359

[19] Bernardini, F.; Rushmeier, H.; Martin, I.M.; Mittleman, J.; Taubin, G.: Building a Digital Model of Michelangelo's Florentine Pieta. In: *Computer Graphics and Applications, IEEE* 22 (2002), jan/feb, Nr. 1, S. 59 –67

[20] Berntsen: *Reflective Targets*. 2012. – `http://www.berntsen.com/Go-Shopping/Surveying/Smart-Targets-Datums-Reflectors`

[21] Besl, P.J.; McKay, H.D.: A Method for Registration of 3-D Shapes. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14 (1992), feb, Nr. 2, S. 239 –256

[22] Böhm, J.; Pateraki, M.: From Point Samples to Surfaces - On Meshing and Alternatives. In: *Proceedings of the ISPRS Commission V Symposium Image Engineering and Vision Metrology*, Universität Stuttgart, 2005

[23] Boissonnat, J.-D.: Geometric Structures for Three-dimensional Shape Representation. In: *ACM Transactions on Graphics* 3 (1984), Oktober, Nr. 4, S. 266–286

[24] Borrmann, D.; Elseberg, J.; Lingemann, K.; Nüchter, A.; Hertzberg, J.: Globally Consistent 3D Mapping with Scan Matching. In: *Journal of Robotics and Autonomous Systems (JRAS)* 65 (2008), Nr. 2, S. 130–142

[25] Borrmann, D.; Elseberg, J.; Nüchter, A.: Thermal 3D Mapping of Building Facades. In: *Proceedings of the 8th Conference on Intelligent Autonomous Systems (IAS '12)*, 2012

[26] Borrmann, D.; J., Elseberg; Lingemann, K.; Nüchter, A.: The 3D Hough Transform for Plane Detection in Point Clouds: A Review and a new Accumulator Design. In: *Journal 3D Research* 2 (2011), März, Nr. 2, S. 32:1–32:13

[27] Borrmann, D.; Nüchter, A.; Dakulovic, M.; Maurovic, I.; Petrovic, I.; Osmankovic, D.; Velagic, J.: Thermal 3D Mapping of Indoor Environments for Saving Energy. In: *Proceedings of the 10th International IFAC Symposium on Robot Control (SYROCO '12)* Bd. 10, 2012

[28] Bourke, P.: *Polygonising a Scalar Field*. 1994. – `http://paulbourke.net/geometry/polygonise/`

[29] Carpin, S.; Lewis, M.; W., Jijun; Balakirsky, S.; Scrapper, C.: USARSim: A Robot Simulator for Research and Education. In: *IEEE International Conference on Robotics and Automation (ICRA 2007)*, 2007

[30] Cgal*, Computational Geometry Algorithms Library*. 2012. – `http://www.cgal.org`

[31] Chapman, B.; Jost, G.; Pas, R. van d.: *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007

[32] Chazelle, B.: Triangulating a Simple Polygon in Linear Time. In: *Foundations of Computer Science* 1 (1990), S. 220–230

[33] Chen, Y.; Li, E.; Li, J.; Zhang, Y.: Accelerating Video Feature Extractions in CBVIR on Multi-Core Systems. In: *Intel Technology Journal* 11 (2007), November, Nr. 04

[34] Chernyaev, E. V.: Marching Cubes 33: Construction of Topologically Correct Isosurfaces / CERN. 1995. – Forschungsbericht

[35] COLLADA.org: *COLLADA – Digital Asset and FX Exchange Schema*. 2012. – `https://collada.org/`

[36] Collett, T. H.; MacDonald, B. A.; Gerkey, B. P.: Player 2.0: Toward a Practical Robot Programming Framework. In: *Proceedings of the Australasian Conference on Robotics and Automation (ACRA)*. Sydney, Australia, 2005

[37] Connor, M.; Kumar, P.: Fast Construction of k-Nearest Neighbor Graphs for Point Clouds. In: *IEEE Transactions on Visualization and Computer Graphics* 16 (2010), July-Aug, Nr. 4, S. 599 –608

[38] Cordone, R.: *Unreal Development Kit Game Programming with UnrealScript*. Packt Publishing Ltd, 2011

[39] D. Shreiner, M. Woo and J. Neider: *OpenGL Programming Guide. The Official Guide to Learning OpenGL, Version 3.0 and 3.1*. 5th. Addison-Wesley Longman, Amsterdam, 2009

[40] Davison, A.J.: Real-time Simultaneous Localisation and Mapping With a Single Camera. In: *Computer Vision, 2003. Proceedings. Ninth IEEE International Conference on*, 2003

[41] Davison, A.J.; Reid, I.D.; Molton, N.D.; Stasse, O.: MonoSLAM: Real-Time Single Camera SLAM. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 29 (2007), june, Nr. 6, S. 1052 –1067

[42] Dean, M.; Schreiber, G.; Bechhofer, S.; Harmelen, F. van; Hendler, J.; Horrocks, I.; McGuinness, D. L.; Patel-Schneider, P. F.; Stein, L. A.: OWL Web Ontology Language Reference / W3C. 2004. – W3C Recommendation

[43] Devillers, O.: The Delaunay Hierarchy. In: *International Journal of Foundations of Computer Science* 13 (2002), S. 163–180

[44] Douglas, D. H.; Peucker, T. K.: Algorithms for the Reduction of the Number of Points Required to Represent a Digitized Line or its Caricature. In: *The Canadian Cartographer* (1973), Nr. 10, S. 112–122

[45] Dürst, M. J.: Additional Reference to Marching Cubes. In: *Computer Graphics* 23 (1988), S. 72–73

[46] E. de Koning: *Polyline Simplification Library* (psimpl). 2010. – `http://psimpl.sourceforge.net/`

[47] Eggert, D. W.; Lorusso, A.; Fisher, R. B.: Estimating 3-D Rigid Body Transformations: A Comparison of Four Major Algorithms. In: *Machine Vision Applications* 9 (1997), März, Nr. 5-6, S. 272–290

[48] Elseberg, J.; Borrmann, D.; Nüchter, A.: Full Wave Analysis in 3D Laser Scans for Vegetation Detection in Urban Environments. In: *2011 XXIII International Symposium on Information, Communication and Automation Technologies (ICAT 2011)*, 2011

[49] Elseberg, J.; Magnenat, S.; Siegwart, R.; Nüchter, A.: Comparison of Nearest-Neighbor-Search Strategies and Implementations for Efficient Shape Registration. In: *Journal of Software Engineering for Robotics (JOSER)* 3 (2012), Nr. 1, S. 2–12

[50] Engel, W.: *Programming Vertex and Pixel Shaders*. Rockland, MA, USA : Charles River Media, Inc., 2004

[51] Engelhard, N.; Endres, F.; Hess, J.; Sturm, J.; Burgard, W.: Real-time 3D Visual SLAM with a Hand-held camera. In: *Proceedings of the RGB-D Workshop on 3D Perception in Robotics at the European Robotics Forum*. Vasteras, Sweden, April 2011

[52] Ericson, C.: *Real-Time Collision Detection*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2004

[53] Fedkiw, R.; Stam, J.; Jensen, H. W.: Visual Simulation of Smoke. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. New York, NY, USA : ACM, 2001

[54] Fischler, M. A.; Bolles, R. C.: Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. In: *Communications of the ACM* 24 (1981), Nr. 6, S. 381–395

[55] Flynn, M. J.: Some Computer Organizations and Their Effectiveness. In: *Computers, IEEE Transactions on* C-21 (1972), sept., Nr. 9, S. 948 –960

[56] Foster, N.; Fedkiw, R.: Practical Animation of Liquids. In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. New York, NY, USA : ACM, 2001

[57] Fox, D.; Burgard, W.; Dellaert, F.; Thrun, S.: Monte Carlo Localization: Efficient Position Estimation for Mobile Robots. In: *Proceedings of the National Conference on Artificial Intelligence (AAAI)*, 1999

[58] Fraundorfer, F.; Scaramuzza, D.: Visual Odometry : Part II: Matching, Robustness, Optimization, and Applications. In: *IEEE Robotics Automation Magazine* 19 (2012), Nr. 2, S. 78 –90

[59] Freedman, B.; Shpunt, A.; Meir, M.; Yoel, A.: *Depth Mapping Using Projected Patterns*. Patent Application, 2008. – WO 2008/120217 A2

[60] Freund, Y.; Schapire, R. E.: A Decision-Theoretic Generalization of On-Line Learning and an Application to Boosting. In: *Journal of Computer and System Sciences* 55 (1997), Nr. 1, S. 119 – 139

[61] Friedman, J. H.; Bentley, J. L.; Finkel, R. A.: An Algorithm for Finding Best Matches in Logarithmic Expected Time. In: *ACM Transactions on Mathematical Software* 3 (1977), September, Nr. 3, S. 209–226

[62] G. Guennebaud and B. Jacob et al.: *Eigen v3*. `http://eigen.tuxfamily.org`, 2010

[63] G. H. Meister: Polygons Have Ears. In: *American Mathematical Monthly* 85 (1975), S. 648–651

[64] Gal, R.; Shamir, A.; Hassner, T.; Pauly, M.; Cohen-Or, D.: Surface Reconstruction using Local Shape Priors. In: *Proceedings of the 5th Eurographics Symposium on Geometry Processing (SGP '07)*, 2007

[65] Garland, M.; Heckbert, P.: Surface Simplification Using Quadric Error Metrics. In: *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '97)*, ACM Press/Addison-Wesley Publishing Co., 1997

[66] Gelder, A. van; Wilhelms, J.: Topological Considerations in Isosurface Generation. In: *ACM Transactions on Graphics* 13 (1994), Oktober, Nr. 4, S. 337–375

[67] Girardeau-Montaut, D.: *CloudCompare*. `http://www.danielgm.net/cc`, 2012

[68] Girardeau-Montaut, D.; Rouxa, M.; Marc, R.; Thibault, G.: Change Detection On Point Cloud Data Acquired With a Ground Laser Scanner. In: *ISPRS Workshop Laser Scanning*, 2005

[69] Golub, G.H.; Reinsch, C.: Singular value decomposition and least squares solutions. In: *Numerische Mathematik* 14 (1970), Nr. 5, S. 403–420

[70] Goodnight, N.; Woolley, C.; Lewin, G.; Luebke, D.; Humphreys, G.: A Multigrid Solver for Boundary Value Problems Using Programmable Graphics Hardware. In: *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS '03)*, Eurographics Association, 2003

[71] Google Inc.: *Streetview*. 2012. – `http://www.google.com/streetview`

[72] Gopi, M.; Krishnan, S.; Silva, C.T.: Surface Reconstruction based on Lower Dimensional Localized Delaunay Triangulation. In: *Computer Graphics Forum* 19 (2000), Nr. 3, S. 467–478

[73] Greenspan, M.; Yurick, M.: Approximate k-d Tree Search for Efficient ICP. In: *Proceedings of the 4th International Conference on 3D Digital Imaging and Modeling (3DIM 2003)*, 2003

[74] Guennebaud, G.; Gross, M.: Algebraic Point Set Surfaces. In: *ACM SIGGRAPH 2007 papers*, 2007

[75] Günther, M.; Wiemann, T.; Albrecht, S.; Hertzberg, J.: Model-based Object Recognition from 3D Laser Data. In: *KI2011: Advances in Artificial Intelligence. 34th Annual German Conference on AI*. Berlin, Heidelberg : Springer-Verlag, 2011

[76] H. Edelsbrunner and E.P. Mücke: Three-Dimensional Alpha Shapes. In: *ACM Transactions on Graphics* 13 (1994), Jan., S. 43–72

[77] H. ElGindy, H. Everett and G.T. Toussaint: Slicing an Ear using Prune-and-Search. In: *Pattern Recognition Letters* (1993), S. 719–722

[78] H. Martin et al.: *libfreenect*. `http://www.freenect.com`, 2010

[79] Henry, P.; Krainin, M.; Herbst, E.; Ren, X.; Fox, D.: RGBD Mapping: Using Depth Cameras for Dense 3D Modeling of Indoor Environments. In: *RGB-D: Advanced Reasoning with Depth Cameras Workshop in conjunction with RSS*, 2010

[80] Hertzberg, J.; Lingemann, K.; Nüchter, A.: *Mobile Roboter*. Springer, 2012

[81] Holz, D.; Holzer, S.; Rusu, R. B.; Behnke, S.: Real-Time Plane Segmentation Using RGB-D Cameras. In: *Proceedings of the 15th RoboCup International Symposium*. Istanbul, Turkey, July 2011

[82] Holz, D.; Lörken, C.; Surmann, H.: Continuous 3D Sensing for Navigation and SLAM in Cluttered and Dynamic Environments. In: *11th International Conference on Information Fusion*, 2008

[83] Hoppe, H.; DeRose, T.; Duchamp, T.; McDonald, J.; Stuetzle, W.: Surface Reconstruction From Unorganized Points. In: *Computer Graphics* 26 (1992), Nr. 2

[84] Horrocks, I.; Patel-Schneider, P. F.; Boley, H.; Tabet, S.; Grosof, B.; Dean, M.: SWRL: A Semantic Web Rule Language Combining OWL and RuleML / World Wide Web Consortium. 2004. – W3C Member Submission

[85] Huhle, B.; Jenke, P.; Strasser, W.: On the Fly Scene Acquisition with a Handy Multisensor System. In: *International Journal of Intelligent Systems Technologies and Applications* 5 (2008), Nr. 3/4, S. 255–263

[86] Illingworth, J.; Kittler, J.: A Survey of the Hough Transform. In: *Computer Vision, Graphics, and Image Processing* 44 (1988), August, Nr. 1, S. 87–116

[87] Inc., Google: *Google 3D Warehouse*. 2012. – `http://sketchup.google.com/3dwarehouse/`

[88] Izadi, S.; Newcombe, R. A.; Kim, D.; Hilliges, O.; Molyneaux, D.; Hodges, S.; Kohli, P.; Shotton, J.; Davison, A. J.; Fitzgibbon, A.: KinectFusion: Real-time Dynamic 3D Surface Reconstruction and Interaction. In: *ACM SIGGRAPH 2011 Talks (SIGGRAPH '11)*. New York, NY, USA : ACM, 2011

[89] J. Blake et al.: *OpenKinect*. `http://openkinect.org`, 2011

[90] J. Sprickerhof, A. N. K. Lingemann L. K. Lingemann; Hertzberg, J.: A Heuristic Loop Closing Technique for Large-Scale 6D SLAM. In: *Automatika* 52 (2011), Nr. 3, S. 199–222

[91] Kazhdan, M.: Reconstruction of Solid Models From Oriented Point Sets. In: *Proceedings of the 3rd Eurographics Symposium on Geometry Processing (SGP '05)*, Eurographics Association, 2005

[92] Kazhdan, M.; Bolitho, M.; Hoppe, H.: Poisson Surface Reconstruction. In: *Proceedings of the 4th Eurographics Symposium on Geometry Processing (SGP '06)*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2006

[93] Kazhdan, Michael; Klein, Allison; Dalal, Ketan; Hoppe, Hugues: Unconstrained Isosurface Extraction on Arbitrary Octrees. In: *Proceedings of the 5th Eurographics symposium on Geometry processing (SGP '07)*. Aire-la-Ville, Switzerland, Switzerland : Eurographics Association, 2007

[94] Kersten, T. P.; Mechelke, K.; Lindstaedt, M.; Sternberg, H.: Methods for Geometric Accuracy Investigations of Terrestrial Laser Scanning Systems. In: *PFG Photogrammetrie, Fernerkundung, Geoinformation* 2009 (2009), 10, Nr. 4, S. 301–315

[95] Khoshelham, K.: Accuracy Analysis of Kinect Depth Data. In: *ISPRS Workshop on Laser Scanning*, 2011

[96] Klank, U.; Pangercic, D.; Rusu, R. B.; Beetz, M.: Real-time CAD Model Matching for Mobile Manipulation and Grasping. In: *9th IEEE-RAS International Conference on Humanoid Robots*. Paris, France, December 7-10 2009

[97] Klasing, K.; Althoff, D.; Wollherr, D.; Buss, M.: Comparison of Surface Normal Estimation Methods for Range Sensing Applications. In: *IEEE International Conference on Robotics and Automation (ICRA '09)*, 2009

[98] Kobbelt, L. P.; Botsch, M.; Schwanecke, U.; Seidel, H-P.: Feature Sensitive Surface Extraction from Volume Data. In: *Proceedings of the 28th annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '01)*. New York, NY, USA : ACM, 2001

[99] Koenig, N.; Howard, A.: Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In: *In IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS '04)*, 2004

[100] Koning, E. de: *Polyline Simplification*. http://www.codeproject.com/Articles/114797/Polyline-Simplification, 2011

[101] Lai, K.; Fox, D.: Object Recognition in 3D Point Clouds Using Web Data and Domain Adaptation. In: *International Journal of Robotics Research* 29 (2010), July, Nr. 8, S. 1019–1037

[102] Lancaster, P.; Salkauskas, K.: Surfaces Generated by Moving Least Squares Methods. In: *Mathematics of computation* 37 (1981), Nr. 155, S. 141–158

[103] Lang, T.: Rules for Robot Draughtsmen. In: *Geographical Magazine* (1969), Nr. 42, S. 50–51

[104] Leica Geosystems: *HDS 7000 Datenblatt*. http://hds.leica-geosystems.com/en/index.htm, 2012

[105] Leica Geosystems: *Targets for the Leica ScanStation C10 and HDS6200*. http://www.leica-geosystems.com/en/Targets_19143.htm, 2012

[106] Levin, D.: The Approximation Power of Moving Least-Squares. In: *Mathematics of Computation* 67 (1998), Nr. 224, S. 1517–1531

[107] Levoy, M.: The Digital Michelangelo Project. In: *Proceedings of the Second International Conference on 3-D Digital Imaging and Modeling*, 1999

[108] *lib3ds*. 2012. – http://code.google.com/p/lib3ds/

[109] Lidar, Velodine: *Webpage*. 2012. – http://www.velodynelidar.com

[110] Lorensen, W. E.; Cline, H. E.: Marching Cubes: A High Resolution 3D Surface Construction Algorithm. In: *ACM SIGGRAPH*, 1987

[111] Lowe, D.G.: Object Recognition from Local Scale-Invariant Features. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision* Bd. 2, 1999

[112] Lu, F.; Milios, E.: Robot Pose Estimation in Unknown Envorinments by Matching 2D Range Scans. In: *Journal of Intelligent and Robotic Systems* 18 (1997), März, Nr. 3, S. 249–275

[113] M. Connor, P. K.: *The Simple, Thread-safe Approximate Nearest Neighbor (STANN) C++ Library*. 2010. – `https://sites.google.com/a/compgeom.com/stann/`

[114] Magnenat, S.: *libnabo*. 2011. – Software Dokumentation. `http://mloss.org/software/view/291/`

[115] Magnenat, S.: *libnabo*. 2012. – `https://github.com/ethz-asl/libnabo`

[116] Magnusson, M.; Nüchter, A.; Lörken, C.; Lilienthal, A. J.; Hertzberg, J.: 3D Mapping the Kvarntorp Mine: A Field Experiment for Evaluation of 3D Scan Matching Algorithms. In: *Proceedings of the Workshop on 3D-Mapping at the IEEE International Conference on Intelligent Robots and Systems (IROS '08)*. Nice, France, September 2008

[117] May, S.: 3D Time-of-flight Ranging for Robotic Perception in Dynamic Environments. In: *Fortschritt-Berichte VDI, Reihe 10: Informatik/Kommunikationstechnik, 798* (2009)

[118] May, S.; Droeschel, D.; Holz, D.; Fuchs, S.; Malis, E.; Nüchter, A.; Hertzberg, J.: Three-dimensional Mapping with Time-of-flight Cameras. In: *Journal of Field Robotics* 26 (2009), Nr. 11–12, S. 934–965

[119] May, S.; Koch, R.; Scherlipp, R.; Nüchter, A.: Robust Registration of Narrow-Field-of-View Range Images. In: *Proceedings of the 10th International IFAC Symposium on Robot Control (SYROCO '12)*, 2012

[120] Melax, S.: A Simple, Fast, and Effective Polygon Reduction Algorithm. In: *Game Developer Magazine* (1998), November, S. 47–49

[121] MESA Imaging: *SwissRanger SR 4000 Datenblatt*. `http://www.mesa-imaging.ch`, 2012

[122] Mitra, Niloy J.; Nguyen, An: Estimating Surface Normals in Noisy Point Cloud Data. In: *Proceedings of the 19th Annual Symposium on Computational Geometry (SCG '03)*. New York, NY, USA : ACM, 2003

[123] Moller, T.; Haines, E.: *Real-Time Rendering*. Second. AK Peters, Ltd., 2002

[124] Mount, D. M.; Arya, S.: *ANN: A Library for Approximate Nearest Neighbor Searching*. 2010. – `http://www.cs.umd.edu/~mount/ANN`

[125] MUHAR, A.: Three-dimensional Modelling and Visualisation of Vegetation for Landscape Simulation. In: *Landscape and Urban Planning* 54 (2001), Nr. 1–4, S. 5 – 17

[126] MUJA, M.: *Fast Library for Approximate Nearest Neighbor*. 2012. – https://github.com/mariusmuja/flann

[127] MUJA, M.; LOWE, D. G.: Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In: *International Conference on Computer Vision Theory and Application VISSAPP'09)*, INSTICC Press, 2009

[128] MUSSER, David R.; DERGE, Gilmer J.; SAINI, Atul: *STL Tutorial and Reference Guide, Second Edition: C++ Programming with the Standard Template Library*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2001

[129] N. KOENIG ET AL.: *Gazebo*. http://gazebosim.org, 2008. – Project Wiki

[130] N. M. AMATO, M. T. GOODRICH AND E. RAMOS: Linear-Time Triangulation of a Simple Polygon Made Easier Via Randomization. In: *Discrete and Computational Geometry* 26 (2001), S. 245–265

[131] NAGAE, Takanori; AGUI, Takeshi; NAGAHASHI, Hiroshi: Surface Construction and Contour Generation from Volume Data. In: LOEW, Murray H. (Hrsg.): *Medical Imaging 1993: Image Processing* Bd. 1898, SPIE, 1993

[132] *C++ header-only fork of the FLANN library for Approximate Nearest Neighbors*. 2012. – http://code.google.com/p/nanoflann/

[133] NARKHEDE, A.; MANOCHA, D.: Fast Polygon Triangulation Based on Seidel's Algorithm. In: *Graphic Gems V* (1995)

[134] NATARAJAN, B. K.: On Generating Topologically Consistent Isosurfaces from Uniform Samples. In: *Visual Computation* 11 (1994), Januar, Nr. 1, S. 52–62

[135] NEALEN, A.: An As-Short-As-Possible Introduction to the Least Squares, Weighted Least Squares and Moving Least Squares Methods for Scattered Data Approximation and Interpolation. 2004. – Forschungsbericht. – http://www.nealen.com/projects/mls/asapmls.pdf

[136] NEWCOMBE, R. A.; IZADI, S.; HILLIGES, O.; MOLYNEAUX, D.; KIM, D.; DAVISON, A. J.; KOHLI, P.; SHOTTON, J.; HODGES, S.; FITZGIBBON, A. W.: KinectFusion: Real-time Dense Surface Mapping and Tracking. In: *International Symposium on Mixed and Augmented Reality (ISMAR 2011)*, 2011

[137] NEWMAN, T; YI, H: A Survey of the Marching Cubes Algorithm. In: *Computers & Graphics* 30 (2006), Nr. 5, S. 854–879

[138] Nielson, G.M.: On Marching Cubes. In: *IEEE Transactions on Visualization and Computer Graphics* 9 (2003), Nr. 3, S. 283–297

[139] Nielson, G.M.; Huang, A.; Sylvester, S.: Approximating Normals for Marching Cubes Applied to Locally Supported Isosurfaces. In: *IEEE Conference on Visualization 2002 (IEEE VIS '02)*, 2002

[140] Nüchter, A.: Parallelization of Scan Matching for Robotic 3D Mapping. In: *Proceedings of the 3rd European Conference on Mobile Robots (ECMR '07)*. Würzburg, Germany, August 2007

[141] Nüchter, A.; Lingemann, K.; Borrmann, D.; Elseberg, J.: *3DTK - The 3D Toolkit*. 2012. – `http://slam6d.sourceforge.net`

[142] Nüchter, A.; Lingemann, K.; Hertzberg, J.: 6D SLAM with Cached k-d tree Search. In: *Proceedings of the 13th IASTED International Conference on Robotics and Applications*. Würzburg, Germany, August 2007

[143] Nüchter, A.; Surmann, H.; Hertzberg, J.: Automatic Classification of Objects in 3D Laser Range Scans. In: *Proceedings of the 8th Conference on Intelligent Autonomous Systems (IAS 2004)*. Amsterdam, The Netherlands, March 2004

[144] Nüchter, A.; Surmann, H.; Lingemann, K.; Hertzberg, J.: Semantic Scene Analysis of Scanned 3D Indoor Environments. (2003), S. 215–222

[145] Opheim, H.: Fast Data Reduction of a Digitized Curve. In: *Geo-Processing* (1982), Nr. 2, S. 33–40

[146] O'Rourke, J.: *Computational Geometry in C*. Cambridge University Press, 1998

[147] Otte, F.; Zimmermann, J.; Sprickerhof, J.; Lingemann, K.; Wiemann, T.; Hertzberg, J.: Automatisiertes Auffinden und Auffüllen von Scanschatten in 3D-Laserscans. In: *Photogrammetrie, Laserscanning, Optische 3D-Messtechnik. Beiträge der Oldenburger 3D-Tage 2011* (2011), S. 46 – 53

[148] OuYang, D.; Feng, H.: On the Normal Vector Estimation for Point Cloud Data From Smooth Surfaces. In: *Computer Aided Design* 37 (2005), September, Nr. 10, S. 1071–1079

[149] Öztireli, A. C.; Guennebaud, G.; Gross, M.: Feature Preserving Point Set Surfaces Based on Non-Linear Kernel Regression. In: *Computer Graphics Forum* 28 (2009), Nr. 2

[150] Pangercic, D.; Tavcar, R.; Tenorth, M.; Beetz, M.: Visual Scene Detection and Interpretation Using Encyclopedic Knowledge and Formal Description Logic. In: *International Conference on Advanced Robotics (ICAR 2009)*. Munich, Germany, 2009

[151] Payne, B. A.; Toga, A. W.: Medical Imaging: Surface Mapping Brain Function on 3D Models. In: *IEEE Compututer Graphics and Applications* 10 (1990), September, Nr. 5, S. 33–41

[152] PMD Technologies: *CamCube 3.0 Datenblatt.* `http://www.pmdtec.com`, 2012

[153] Pointools: *Pointools Vortex Engine.* `http://www.pointools.com`, 2012

[154] Qiu, D.; May, S.; Nüchter, A.: GPU-Accelerated Nearest Neighbor Search for 3D Registration. In: *Proceedings of the 7th International Conference on Computer Vision Systems: Computer Vision Systems (ICVS '09).* Berlin, Heidelberg : Springer-Verlag, 2009

[155] Quigley, M.; Conley, K.; Gerkey, B. P.; Faust, J.; Foote, T.; Leibs, J.; Wheeler, R.; Ng, A. Y.: ROS: An Open-source Robot Operating System. In: *ICRA Workshop on Open Source Software*, 2009

[156] R. Demming, D. D.: *Introduction to the Boost C++ Libraries; Volume I - Foundations.* Datasim Education BV, 2001

[157] Renz, J.; Nebel, B.: Qualitative Spatial Reasoning Using Constraint Calculi. In: Aiello, Marco (Hrsg.); Pratt-Hartmann, Ian (Hrsg.); Benthem, Johan van (Hrsg.): *Handbook of Spatial Logics.* Springer, 2007, S. 161–215

[158] Reumann, K.; Witkam, A.P.: Optimizing Curve Segmentation in Computer Graphics. In: *International Computer Symposium* (1974), S. 467–472

[159] Rinnewitz, K. O.: *Automatische Zuordnung und Generierung von Texturen in 3D Rekonstruktionen*, Universität Osnabrück, Masterarbeit, 2012

[160] Rinnewitz, K. O.; Schalk, S. K.; Wiemann, T.; Lingemann, K.; Hertzberg, J.: Das Las Vegas Reconstruction Toolkit. In: *Photogrammetrie, Laserscanning, Optische 3D-Messtechnik. Beiträge der Oldenburger 3D-Tage 2012* (2012)

[161] Rottensteiner, F.: Automatic Generation of High-quality Building Models from Lidar Data. In: *IEEE Computer Graphics and Applications* 23 (2003), Nov–dec., Nr. 6, S. 42 – 50

[162] Rusu, R. B.: *Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments*, Computer Science department, Technische Universitaet Muenchen, Germany, Diss., October 2009

[163] Rusu, R. B.; Blodow, N.; Beetz, M.: Fast Point Feature Histograms (FPFH) for 3D Registration. In: *Proceedings of the 2009 IEEE international conference on Robotics and Automation (ICRA '09).* Piscataway, NJ, USA : IEEE Press, 2009

[164] Rusu, R. B.; Marton, Z. C.; Blodow, N.; Dolha, M. E.; Beetz, M.: Towards 3D Point Cloud Based Object Maps for Household Environments. In: *Robotics and Autonomous Systems, Special Issue on Semantic Knowledge in Robotics* 56 (2008), Nr. 11, S. 927–941

[165] Rusu, R. B.; Marton, Z. C.; Blodow, N.; Holzbach, A.; Beetz, M.: Model-Based and Learned Semantic Object Labeling in 3D Point Cloud Maps of Kitchen Environments. In: *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS 09)*. St. Louis, USA : IEEE, 2009

[166] Rusu, R.B.; Cousins, S.: 3D is here: Point Cloud Library (PCL). In: *IEEE International Conference on Robotics and Automation (ICRA '11)*, 2011

[167] Schafer, H.; Hach, A.; Proetzsch, M.; Berns, K.: 3D Obstacle Detection and Avoidance in Vegetated Off-road Terrain. In: *IEEE International Conference on Robotics and Automation (ICRA '08)*, 2008

[168] Schaufler, G.; Jensen, H.: Ray Tracing Point Sampled Geometry. In: *Proceedings of the Eurographics Workshop on Rendering Techniques 2000*. London, UK, UK : Springer-Verlag, 2000

[169] Schnabel, R.; Degener, P.; Klein, R.: Completion and Reconstruction with Primitive Shapes. In: *Computer Graphics Forum* 28 (2009), Nr. 2, S. 503–512

[170] Schnabel, R.; Wahl, R.; Klein, R.: Efficient RANSAC for Point-Cloud Shape Detection. In: *Computer Graphics Forum* 26 (2007), Nr. 2, S. 214–226

[171] Seidel, R.: A Simple and Fast Incremental Randomized Algorithm for Computing Trapedozional Decompositions and for Triangulating Polygons. In: *Computational Geometry: Theory and Applications* 1 (1991), Nr. 1, S. 51–64

[172] Shi, W.; Cheung, C.: Performance Evaluation of Line Simplification Algorithms for Vector Generalization. In: *The Cartographic Journal* 43 (2006), Nr. 1, S. 27–44

[173] Shirley, A. P.and T. P.and Tuchman: A Polygonal Approximation to Direct Scalar Volume Rendering. In: *SIGGRAPH Computer Graphics* 24 (1990), November, Nr. 5, S. 63–70

[174] Shirley, P.; Marschner, S.: *Fundamentals of Computer Graphics, 3rd Edition*. Natick, MA, USA : A. K. Peters, Ltd., 2009

[175] Shreiner, D.: *OpenGL Reference Manual. The Official Reference Document to OpenGL, Version 1.4*. 4th. Addison-Wesley Longman, Amsterdam, 2004

[176] Siddiqi, K.; Kimia, B. B.; Shu, C.: Geometric Shock-capturing Eno Schemes for Subpixel Interpolation, Computation and Curve Evolution. In: *Graphical Models and Image Processing* 59 (1997), November, Nr. 5, S. 278–301

[177] Sirin, E.; Parsia, B.; Grau, B. C.; Kalyanpur, A.; Katz, Y.: Pellet: A practical OWL-DL reasoner. In: *Journal of Web Semantics* 5 (2007), Nr. 2, S. 51–53

[178] Smits, B.: Efficiency Issues for Ray Tracing. In: *Journal of Graphic Tools* (1998)

[179] Somer, J.: *Mesh Simplification Viewer.* – http://www.jsomers.com/vipm_demo/meshsimp.html

[180] Stanford University Computer Graphics Laboratory: *The Stanford 3D Scanning Repository.* 2012. – http://graphics.stanford.edu/data/3Dscanrep/

[181] Steder, B.; Rusu, R. B.; Konolige, K.; W., Burgard: Point Feature Extraction on 3D Range Scans Taking Into Account Object Boundaries. In: *IEEE International Conference on Robotics and Automation (ICRA '05).* Shanghai, China, May 2011

[182] Steinbruecker, F.; Sturm, J.; Cremers, D.: Real-Time Visual Odometry from Dense RGB-D Images. In: *Workshop on Live Dense Reconstruction with Moving Cameras at the International Conf. on Computer Vision (ICCV)*, 2011

[183] Stiene, S.; Lingemann, K.; Nuchter, A.; Hertzberg, J.: Contour-Based Object Detection in Range Images. In: *Third International Symposium on 3D Data Processing, Visualization, and Transmission*, 2006

[184] Stückler, Jörg; Behnke, S.: Model Learning and Real-Time Tracking Using Multi-Resolution Surfel Maps. In: *AAAI*, 2012

[185] Surmann, H.; Nuechter, A.; Hertzberg, J.: An Autonomous Mobile Robot with a 3D Laser Range Finder for 3D Exploration and Digitalization of Indoor Environments. In: *Robotics and Autonomous Systems* 45 (2003), Nr. 3 - 4, S. 181 – 198

[186] TerraDat Geomatics (UK): *Castle Coch Datendsatz.* 2012. – http://www.getlaserscanning.com/

[187] Thibault, W. C.; Naylor, B. F.: Set Operations on Polyhedra Using Binary Space Partitioning Trees. In: *Proceedings of the 14th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '87)*, 1987

[188] Thrun, S.; Burgard, W.; Fox, D.: *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents).* The MIT Press, 2005

[189] Turk, G.; Levoy, M.: Zippered Polygon Meshes from Range Images. In: *Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH '94)*. New York, NY, USA : ACM, 1994

[190] Unnikrishnan, R.: *Statistical Approaches to Multi-Scale Point Cloud Processing*. Pittsburgh, PA, Robotics Institute, Carnegie Mellon University, Diss., May 2008

[191] van den Bergen, G.: Efficient Collision Detection of Complex Deformable Models Using AABB Trees. In: *Journal of Graphics Tools* 2 (1998), Januar, Nr. 4, S. 1–13

[192] Vanco, M: *A Direct Approach for the Segmentation of Unorganized Points and Recognition of Simple Algebraic Surfaces*, University of Technology, Chemnitz, Diss., 2009

[193] Wendland, H.: Piecewise Polynomial, Positive Definite and Compactly Supported Radial Functions of Minimal Degree. In: *Advances in Computational Mathematics* 4 (1995), S. 389–396. – ISSN 1019–7168

[194] Wenger, M.: *Exploring CenSurE Features*, Universität Osnabrück, Bachelorarbeit, 2010

[195] Wiemann, T.: *Automatische Rekonstruktion Planarer 3D-Umgebungen*, Universität Osnabrück, Masterarbeit, 2007

[196] Wiemann, T.: *The Las Vegas Surface Reconstruction Toolkit*. 2012. – http://www.las-vegas.uni-osnabrueck.de

[197] Wiemann, T.; Lingemann, K.; Nüchter, A.; Hertzberg, J.: A Toolkit for Automatic Generation of Polygonal Maps – Las Vegas Reconstruction. In: *Proceedings of the 7th German Conference on Robotics (ROBOTIK 2012)*. München : VDE Verlag, 2012

[198] Wiemann, T.; Nüchter, A.; Lingemann, K.; Stiene, S.; Hertzberg, J.: Surface Reconstruction for 3D Robotic Mapping. In: *Proceedings of the Workshop on 3D-Mapping at the IEEE International Conference on Intelligent Robots and Systems (IROS '08)*, 2008

[199] Wiemann, T.; Nüchter, A.; Lingemann, K.; Stiene, S.; Hertzberg, J.: Automatic Construction of Polygonal Maps From Point Cloud Data. In: *IEEE International Workshop on Safety Security and Rescue Robotics (SSRR 2010)*, 2010

[200] Wikipedia: *Data Structure Alignment*. 2012. – http://en.wikipedia.org/wiki/Data_structure_alignment

[201] Wikipedia: *OpenMP*. 2012. – http://en.wikipedia.org/wiki/OpenMP

[202] Wikipedia: *Polygonnetz*. 2012. – http://de.wikipedia.org/wiki/Polygonnetz

[203] Wikipedia: *STL (file format).* 2012. – STL(fileformat)

[204] Wikipedia: *Texture Mapping.* 2012. – http://en.wikipedia.org/wiki/Texture_mapping

[205] Wikipedia: *Wavefront .obj file.* 2012. – http://en.wikipedia.org/wiki/Wavefront_.obj_file

[206] Wilhelms, J.; Van Gelder, A.: Octrees for Faster Isosurface Generation. In: *ACM Transactions on Graphics* 11 (1992), Juli, Nr. 3, S. 201–227

[207] Williams, A.; Barrus, S.; Keith, R.; Shirley, M. P.: An Efficient and Robust Bay-Box Intersection Algorithm. In: *Journal of Graphics Tools* 10 (2003), S. 54

[208] Wülfing, J.: *Localization of a Mobile Robot in 3D With Six Degrees of Freedom Using a 3D Camera*, Universität Osnabrück, Bachelorarbeit, 2009

[209] Wülfing, J.; Hertzberg, J.; Lingemann, K.; Nüchter, A.; Stiene, S.; Wiemann., T.: Towards Real Time Robot 6D Localization in a Polygonal Indoor Map Based on 3D ToF Camera Data. In: *Proceedings of the 7th IFAC Symposium on Intelligent Autonomous Vehicles (IAV 2010)*, 2010

[210] X. Kong, H. Everett and G.T. Toussaint: The Graham Scan Triangulates Simple Polygons. In: *Pattern Recognition Letters* (1990), S. 713–716

[211] Xing, Yi-Si; Liu, X.P.; Xu, Shao-Ping: Efficient Collision Detection Based on AABB Trees and Sort Algorithm. In: *8th IEEE International Conference on Control and Automation (ICCA 2010)*, 2010

[212] Zhang, Z.: A Flexible New Technique for Camera Calibration. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 22 (2000), Nr. 11, S. 1330 – 1334

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie Zitate kenntlich gemacht habe.

Osnabrück, April 2013