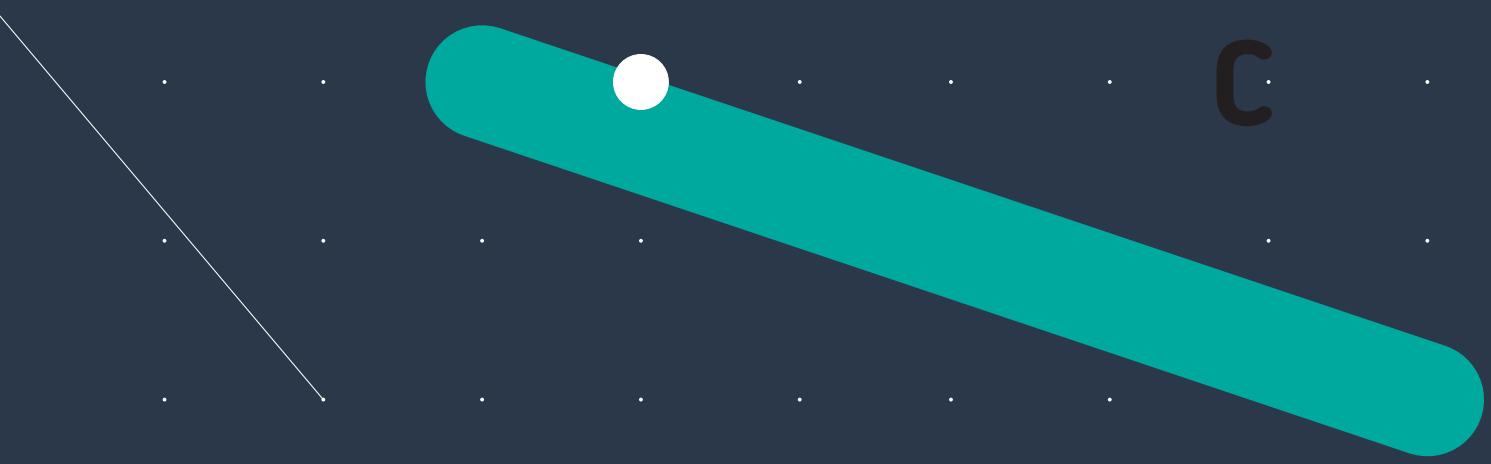
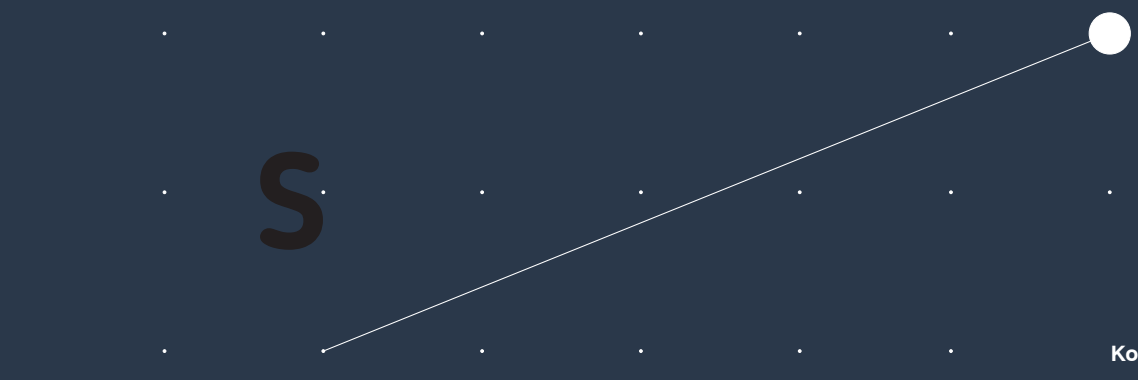
Hochschule Konstanz
Fakultät Informatik
Institut für Optische Systeme

Eingereicht von
Lorenz Bung
Matrikelnummer 295261

lorenz.bung@htwg-kostanz.de



Rekonstruktion von Meshes für industrielles Bin-Picking und Depalettieren



Konstanz, 31.07.2020

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

an der

Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Fakultät Informatik
Studiengang Angewandte Informatik

Thema:	Rekonstruktion von Meshes für industrielles Bin-Picking und Depalettieren
Kandidat:	Lorenz Bung Banater Str. 9 78467 Konstanz
1. Prüfer:	Prof. Dr. Georg Umlauf
2. Prüfer:	Simon Schmeißer
Ausgabedatum:	01.05.2020
Abgabedatum:	31.07.2020

Ehrenwörtliche Erklärung

Hiermit erkläre ich, *Lorenz Bung, geboren am 26.06.1997 in Konstanz*, dass ich

(1) meine Bachelorarbeit mit dem Titel

Rekonstruktion von Meshes für industrielles Bin-Picking und Depalettieren

bei Isys Vision GmbH unter Anleitung von Prof. Dr. Georg Umlauf und Simon Schmeißer selbständig und ohne fremde Hilfe angefertigt und keine anderen als die angeführten Hilfen benutzt habe;

(2) die Übernahme wörtlicher Zitate, von Tabellen, Zeichnungen, Bildern und Programmen aus der Literatur oder anderen Quellen (Internet) sowie die Verwendung der Gedanken anderer Autoren an den entsprechenden Stellen innerhalb der Arbeit gekennzeichnet habe;

(3) dass die eingereichten Abgabe-Exemplare in Papierform und im PDF-Format vollständig übereinstimmen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 31.07.2020

(Unterschrift)

Abstract

Thema:	Rekonstruktion von Meshes für industrielles Bin-Picking und Depalettieren
Kandidat:	Lorenz Bung
Betreuer:	Prof. Dr. Georg Umlauf Institut für Optische Systeme
	Simon Schmeißer Isys Vision GmbH
Abgabedatum:	31.07.2020
Schlagworte:	Meshrekonstruktion, Robotik, Geometrisches Modellieren, ROS, PCL, Punktwolke

Describe the objective and results of this thesis in a few words. Typically one page.

In dieser Arbeit werden verschiedene Ansätze zur Rekonstruktion von Meshes aus 3D-Punktwolken analysiert, miteinander verglichen und bewertet. Die erste Methode ist YAK, eine Variante von Kinect Fusion, bei der ein Truncated Signed Distance Field zum Einsatz kommt. Der zweite Ansatz ist, ein einzelnes Teil mithilfe einer beweglichen Kamera am Roboterarm aus mehreren Posen einzuscannen und so ein Gesamtbild zu erhalten. Im dritten Ansatz wird eine

Extended Abstract

Thema:	Rekonstruktion von Meshes für industrielles Bin-Picking und Depalettieren
Kandidat:	Lorenz Bung
Betreuer:	Prof. Dr. Georg Umlauf Institut für Optische Systeme
	Simon Schmeißer Isys Vision GmbH
Abgabedatum:	31.07.2020
Schlagworte:	Meshrekonstruktion, Robotik, Geometrisches Modellieren, ROS, PCL, Punktwolke

Extended Abstract über 2 Seiten. Beispielhafte Texte aus anderen Teamprojekten oder Abschlussarbeiten können aus dem verlinkten Dokument entnommen werden.

Dieser Text soll als Dokumentation des Teamprojekts für den zukünftigen Jahresbericht des Institut für Optische Systeme dienen. Gerne können auch Bilder eingefügt werden. Ebenso wichtig ist es auch die Referenzen aufzulisten wie z.B. [1].

Literatur

[1] R. B. Rusu und S. Cousins, “3d is here: Point cloud library (pcl)”, in *2011 IEEE international conference on robotics and automation*, IEEE, 2011, S. 1–4.

Inhaltsverzeichnis

Abkürzungsverzeichnis

PCL Point Cloud Library

ROS Robot Operating System

KinFu Kinect Fusion

YAK Yet Another Kinect Fusion

ICP Iterative Closest Point

4PCS Four-Points Congruent Sets

TSDF Truncated Signed Distance Field

OpenGR Open Global Registration

VVM Vertex-Vertex-Mesh

LUT Lookup Table

CUDA Compute Unified Device Architecture

ECE Euclidean Cluster Extracion

RoI Region of Interest

MSE Mean Squared Error

KNN K Nearest Neighbors

1. Einleitung

1.1 Mikado

Isys vision GmbH [1] ist ein Unternehmen aus Freiburg im Breisgau, das sich mit Systemintegration und industrieller Bildverarbeitung beschäftigt. Neben Entwicklungen in der 2D-Bildverarbeitung (zum Beispiel in der Leiterplattenproduktion) spielt auch Machine Vision im 3D-Bereich in Kombination mit Robotik eine große Rolle.

Mikado [2] ist ein Softwarepaket von isys vision, welches zur Robotersteuerung und 3D-Bildverarbeitung in der Industrie eingesetzt wird. Es besteht aus zwei Komponenten: Mikado 3D dient der eigentlichen Bildverarbeitung, während Mikado Adaptive Robot Control (ARC) zusätzlich dazu die Robotersteuerung und Kollisionsplanung beinhaltet. Die Hauptanwendung ist dabei das sogenannte ”Bin-Picking”, also das Greifen von sortenreinen Teilen aus einer unsortierten Kiste. Das Softwarepaket basiert auf der Open-Source-Bibliothek Robot Operating System (ROS) [3]. Zur Erfassung der Bilddaten kommen 3D-Kameras von Ensenso [4] zum Einsatz.

1.2 Motivation

Bei der Bestimmung der 6D-Posen von Objekten wird bei Mikado unter Anderem der ”surface based matching”-Algorithmus von MvTec Halcon [5] verwendet. Dafür wird ein CAD-Modell des Objekts benötigt, welches jedoch in vielen Fällen nicht vorhanden ist.

Die Gründe dafür sind vielfältig. Etwa können die existierenden Modelle im aktuellen Fertigungszustand nicht vorhanden sein. Ein weiterer Grund für fehlende Modelle ist, dass diese aus organisatorischen Gründen schwer zu bekommen sind, beispielsweise

wenn nur eine Weiterverarbeitung eines zugelieferten Bauteils stattfindet.

Neben fehlenden Modellen sind auch häufig falsche Modelle ein Problem. So kann es vorkommen, dass die existierenden Daten fertigungsbedingt nicht zum tatsächlichen Produkt passen und Teile daher nicht genau erkannt werden können.

Die Generierung eines CAD-Modells aus den Daten der 3D-Kamera ist somit eine Möglichkeit, um für das tatsächlich vorhandene Produkt eine korrekte Repräsentation zu finden, welche im weiteren Bildverarbeitungsprozess genutzt werden kann. Weiterhin dient dies auch der Vereinfachung der Endanwendung von Mikado ARC. Besonders bei oft variierenden Produktkonfigurationen wird der Anwendungsprozess vereinfacht, wenn nicht erst ein entsprechendes CAD-Modell organisiert werden muss.

1.3 Zielsetzung

Für die Rekonstruktion aus der Punktwolke gibt es drei mögliche Ansätze:

1. Es liegt ein stationäres Objekt vor, welches mit einer am Roboter angebrachten 3D-Kamera erfasst wird. Der Roboter bewegt die Kamera um das Objekt herum, um so Informationen aus mehreren Perspektiven zu erhalten. Diese werden danach miteinander kombiniert; der Hintergrund der erfassten Daten wird anschließend eliminiert.
2. Die Kamera ist stationär angebracht, während das zu erkennende Objekt freihändig demonstriert wird. So kann das Objekt aus mehreren Ansichten aufgenommen werden, welche dann zusammengeführt werden.
3. Es liegen mehrere Objekte der selben Art in verschiedenen Orientierungen vor, beispielsweise in einer Kiste. Nach Segmentierung der Objekte wird aus den unterschiedlichen Teilansichten ein repräsentatives Modell generiert.

Ziel der Arbeit ist es, geeignete Lösungen für die verschiedenen Ansätze zu entwickeln sowie die unterschiedlichen Rekonstruktionsmöglichkeiten zu implementieren.

Weiterhin sollen die Ergebnisse untereinander und mit bereits bestehenden Algorithmen verglichen werden. Dieser Vergleich soll sowohl bezüglich der wichtigsten Eigenschaft der Qualität, als auch auf Basis untergeordneter Faktoren wie Geschwindigkeit der Algorithmen und Nutzungskomfort der Ansätze stattfinden.

2. Grundlagen

Zum Verständnis des Themas der Arbeit ist die Erklärung einiger Grundlagen notwendig. In 2.1 werden zunächst wichtige Datenstrukturen zur Arbeit mit 3D-Daten eingeführt und die verschiedenen Vor- und Nachteile beleuchtet. Weiterhin werden in 2.2 unterschiedliche Methoden und Repräsentationen zur Arbeit mit Polygonnetzen erklärt. Auch hier wird diskutiert, welche positiven und negativen Aspekte die jeweiligen Ansätze auszeichnet.

2.1 Aufnahme und Speicherung von 3D-Bildern

Zur Aufnahme von 3D-Bilddaten gibt es mehrere verschiedene Möglichkeiten. Ein LIDAR-System sendet beispielsweise mehrere Lichtstrahlen in verschiedene Richtungen, die anschließend Informationen über die Entfernung zu einem Objekt in diesem Punkt liefern. Eine Stereokamera liefert im Gegensatz dazu zwei Bilder, die anschließend durch spezielle Software zu einem dreidimensionalen Bild zusammengesetzt werden. Eine weitere Möglichkeit besteht darin, ein bestimmtes Muster auf die Umgebung zu projezieren, dieses dann aus einer anderen Perspektive aufzunehmen und aus der räumlichen Verzerrung des Musters die Tiefe zu errechnen.

Die so gewonnenen Informationen lassen sich durch mehrere verschiedene Datenmodelle repräsentieren.

2.1.1 Tiefenbild

Ein Tiefenbild ist eine einfache Möglichkeit, in einem zweidimensionalen Bild zusätzlich Informationen über die Entfernung der Kamera zu Objekten abzuspeichern. Diese Technik ist unter Anderem aus der Computergrafik bekannt, wo sie beim Z-Buffering Anwendung findet [6, S. 32]. Dabei werden sowohl das aufgenommene Bild, aber zusätzlich auch ein 2D-Array mit der Tiefeninformation des zugehörigen Pixels gespeichert. Das Tiefenbild dient somit als Lookup Table (LUT), um die z-Information zu einem Pixel im 2D-Bild zu erhalten. Ein Beispiel für den Zusammenhang zwischen diesen beiden Komponenten ist in Abbildung 2.1 dargestellt.

So kann das originale 2D-Bild um eine dritte Dimension erweitert werden, wodurch sich beispielsweise dreidimensionale Formen modellieren oder rekonstruieren lassen [7]. Auch in der 3D-Fotografie finden Tiefenbilder Anwendung [8].

Abbildung 2.1: 2D-Farbbild und zugehöriges Tiefenbild. Entnommen aus [9, S. 649]

Tiefenbilder haben im Vergleich zu anderen möglichen Datenmodellen den Vorteil, dass 2D-Bilder sehr einfach um zugehörige Tiefeninformationen erweitert werden können. Jedoch gibt es auch einige Nachteile, die nicht umgangen werden können:

- Die 3D-Daten sind ausschließlich aus der Kameraperspektive vorhanden. Um Informationen aus einer anderen Perspektive zu erhalten, muss erst aufwändig umgerechnet werden.
- Da (in den meisten Fällen) nur ein zweidimensionales Array mit einem Kanal für die Tiefeninformation angelegt wird, können nur die Entfernungen zu den der Kamera nächsten Objekte gespeichert werden. Verdeckte, reflektierende oder durchsichtige Oberflächen können nicht gespeichert werden.
- Das Array ist an die Auflösung der Kamera gebunden. Insbesondere bei großer Entfernung zu Objekten werden diese aufgrund des Bildwinkels sehr schlecht abgetastet, was später zu Aliasing-Effekten führen kann.

- Die Bittiefe der Werte im Array kann - je nach gewünschter Auflösung - zu niedrig sein, bzw. muss erhöht werden. Eine typische Farbtiefe eines Grauwertbilds von 8 Bit repräsentiert beispielsweise nur $2^8 = 256$ Stufen - was schnell zu wenig wird. Insbesondere bei hoher notwendiger Auflösung im nahen Bereich, aber gleichzeitig vorhandenen weit entfernten Objekten kann dies zum Problem werden.

2.1.2 Voxel Grid

Bei einem als Bitmap vorliegenden 2D-Grauwertbild werden die Bilddaten diskretisiert in einem zweidimensionalen Array I gespeichert. Mit einer Bittiefe b, einer Bildhöhe i und einer Bildbreite j ergibt sich damit die in Gleichung 2.1 dargestellte Matrix für I.

$$I = \begin{pmatrix} v_{11} & v_{12} & v_{13} & & v_{1j} \\ v_{21} & v_{22} & v_{23} & \cdots & v_{2j} \\ v_{31} & v_{32} & v_{33} & & v_{3j} \\ & \vdots & & \ddots & \vdots \\ v_{i1} & v_{i2} & v_{i3} & \cdots & v_{ij} \end{pmatrix}, \; v_{ij} \in [0; 2^b), \; b, i, j \in \mathbb{N}^+ \tag{2.1}$$

Dieses Prinzip lässt sich einfach auf den dreidimensionalen Raum erweitern. Die so erhaltene Datenstruktur nennt sich Voxel Grid. Als Voxel bezeichnet man somit eine einzelne Datenzelle im 3D-Array, also das dreidimensionale Äquivalent zum Pixel.

Voxel Grids finden in vielen Bereichen Anwendung, zum Beispiel in der Medizin [10–13] oder Geographie [14]. Auch zum Downsampling von Punktwolken werden sie verwendet [15].

Gegenüber einem Tiefenbild hat ein Voxel Grid den inhärenten Nachteil, dass Speicherplatz für jeden Voxel benötigt wird. Dies führt schnell zu großem Speicherbedarf M, da dieser kubisch zur Auflösung r steigt: $M = r_x * r_y * r_z * b$ bei Bittiefe b. Da der Speicher bereits bei der Initialisierung reserviert werden muss, ist die Änderung der Auflösung oder der Größe des abgedeckten Raumes unmöglich. In diesem Fall müssen ein neues Voxel Grid (mit der neuen Auflösung bzw. Größe) angelegt und sämtliche Daten kopiert werden.

Zur Reduzierung des großen Speicherbedarfs werden deswegen häufig Baumstrukturen (sogenannte Octrees) eingesetzt. Dabei wird der Raum in 8 Subvoxel geteilt, welche jeweils ein Blatt des Baumes darstellen. Ist der Voxel gefüllt, wird er wieder entsprechend in 8 Subvoxel geteilt. Dies wird wiederholt, bis die gewünschte Auflösung

erreicht oder eine bestimmte Tiefe im Baum erreicht ist. Szenen, die große Unterschiede in der Auflösung aufweisen oder viele leere Voxel beinhalten, brauchen so deutlich weniger Speicherplatz.

Ein Vorteil im Vergleich zum Tiefenbild ist jedoch, dass die Abhängigkeit von der Kameraperspektive wegfällt. Dadurch können hier auch Objekte modelliert werden, die im Tiefenbild durch eine Verdeckung nicht sichtbar wären. Desweiteren ist der Raum einheitlich diskretisiert. Dies kann je nach Szene und Kameraposition sowohl ein Vor- als auch ein Nachteil sein: Bei einem Tiefenbild nimmt der Abstand der Messpunkte mit der Tiefe ab. Ist die Entfernung zwischen Objekt und Kamera gering, werden diese mit einer deutlich besseren Auflösung abgetastet als sehr weit entfernte Objekte. Im Voxel Grid werden Objekte jedoch überall gleich abgetastet. Dies führt zu verbesserter Auflösung bei entfernten und zu schlechterer Auflösung bei nahen Objekten.

2.1.3 Punktwolken

Ein weiteres häufig verwendetes Datenmodell ist eine Punktwolke. Als Punktwolke bezeichnet man eine Menge $M \subset \mathbb{R}^3$ von Punkten im (mindestens) dreidimensionalen Raum. Zusätzlich zur räumlichen Information können auch noch weitere Daten pro Punkt gespeichert sein, wie RGB-Werte, Normalen, Genauigkeit oder Objektklasse (falls schon eine Segmentierung vorgenommen wurde). Dadurch gilt Gleichung 2.2 für M.

$$M = \begin{pmatrix} p_x^1 & p_y^1 & p_z^1 & \cdots \\ p_x^2 & p_y^2 & p_z^2 & \cdots \\ p_x^3 & p_y^3 & p_z^3 & \cdots \\ \vdots & \vdots & \vdots & \ddots \end{pmatrix} \tag{2.2}$$

Die Nutzung von Punktwolken bringt im Vergleich zu anderen 3D-Datenmodellen einige Vorteile.

- Die zugrundeliegende Datenstruktur ist extrem trivial; es handelt sich um eine einfache Liste. Dies ermöglicht sehr schnelle Operationen, wie zum Beispiel das Hinzufügen eines Punktes oder die Erweiterung um zusätzliche Informationen. Es lässt sich sehr einfach über die Punkte iterieren.
- Der Speicherbedarf einer Punktwolke wächst dynamisch, und zwar linear mit der Zahl der enthaltenen Punkte. Beim Voxel Grid muss dagegen bereits am Anfang

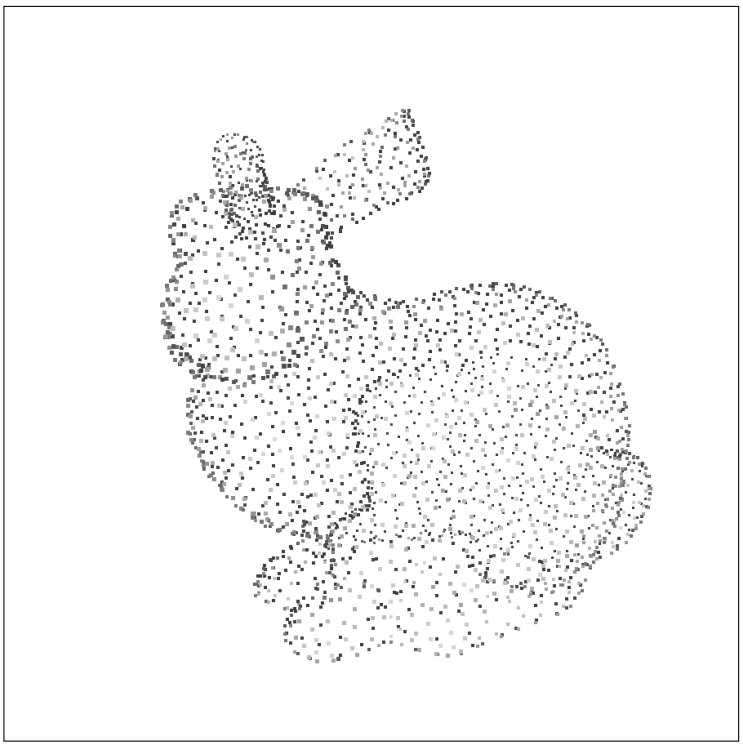


Abbildung 2.2: Punktwolke des Stanford Bunny [16]

Speicher für jede Zelle reserviert werden, egal ob ein Punkt enthalten ist oder nicht.

- Man ist nicht, wie beim Tiefenbild oder Voxelgrid, auf eine feste Anzahl an Punkten limitiert. Beim Tiefenbild können maximal $n * m$ Punkte (ein Wert pro Pixel) mit einer Tiefeninformation versehen werden, beim Voxel Grid maximal $x * y * z$ Punkte (ein Wert pro Voxel). Eine Kapazitätserweiterung, wie beispielsweise das Hinzufügen neuer Pixelspalten oder -zeilen im Tiefenbild, ist bei der Punktwolke nicht notwendig.
- Die Auflösung ist nur durch die Gleitkommagenauigkeit der Maschine limitiert. Beim Voxel Grid ist sie im Gegensatz dazu durch die Größe der Voxel limitiert. Beim Tiefenbild ist die Auflösung abhängig von der Bittiefe und der z-Tiefe im Bild - bei weiter entfernten Objekten ist die Auflösung auch niedriger.

Es gibt jedoch auch einige Nachteile gegenüber den anderen Repräsentationen:

- Die Rekonstruktion eines Meshs ist nicht eindeutig. Beim Voxel Grid lässt sich, beispielsweise mithilfe des Marching-Cubes-Algorithmus [17], ein eindeutiges Mesh

rekonstruieren. Bei einer Punktwolke ist im Gegensatz dazu nicht festgelegt, welche Punkte miteinander verbunden werden sollen.

- Viele bekannte Techniken aus der 2D-Bildverarbeitung (wie zum Beispiel Segmentierung, Filterung usw.) lassen sich auf Tiefenbilder direkt übertragen. Dies ist bei Punktwolken nicht möglich.

2.2 Meshrepräsentationen

Die bisher angesprochenen Datenstrukturen beinhalten ausschließlich Informationen über Punkte im dreidimensionalen Raum (Geometrie), nicht jedoch aber über die Relation, in der diese zueinander stehen (Topologie). Das Polygonmesh erweitert diese Daten durch Kanteninformationen: Punkte (sogenannte Vertices) sind durch Kanten (Edges) miteinander verbunden. Die so entstehenden Flächen der Polygone werden als Faces bezeichnet. Da jedes Polygon in Dreiecke zerlegt werden kann, werden meist nur diese gespeichert.

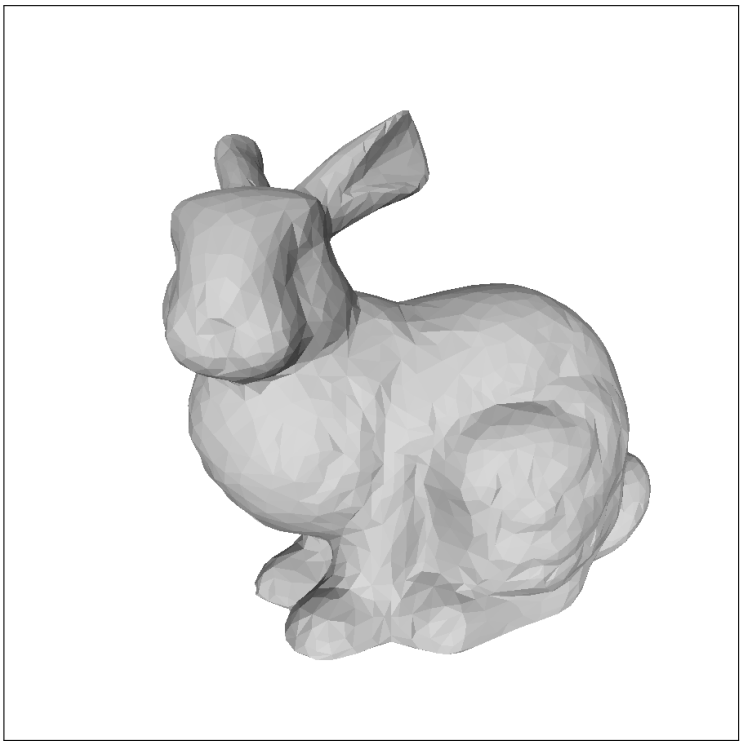


Abbildung 2.3: Mesh des Stanford Bunny [16]

Vertices, Edges und Faces können in verschiedenen Repräsentationen gespeichert werden. Diese haben unterschiedliche Eigenschaften und Vor- und Nachteile, die im folgenden genauer beleuchtet werden. Zum besseren Verständnis wird das in Abbildung 2.4 dargestellte Polygonnetz jeweils in den unterschiedlichen Strukturen codiert.

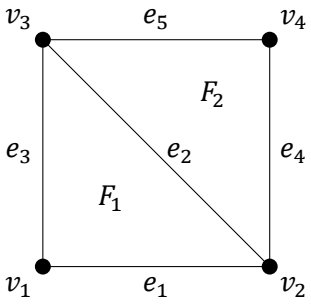


Abbildung 2.4: Beispielhaftes Polygonnetz, bestehend aus Knoten, Kanten und Facetten

2.2.1 Vertex-Vertex-Mesh

Beim Vertex-Vertex-Mesh (VVM) werden nur Vertices gespeichert, Kanten und Flächen ergeben sich implizit. Zu den in Abbildung 2.4 dargestellten Dreiecken wird also die Liste in Tabelle 2.1 gespeichert.

Vertex	verbundene Vertices
v_1	v_2, v_3
v_2	v_1, v_3, v_4
v_3	v_1, v_2, v_4
v_4	v_2, v_3

Tabelle 2.1: Vertextabelle beim VVM

Der Speicherbedarf dieser simplen Darstellung ist sehr gering. Die Einträge in der Tabelle können als Indizes der Vertexliste gespeichert werden. Weiterhin sind Vertex-basierte Operationen sehr schnell. Zum Hinzufügen eines Knotens muss beispielsweise nur ein neuer Eintrag in der Liste angelegt werden, sowie der neue Knoten zu den verbundenen Vertices hinzugefügt werden.

Um jedoch Informationen zu bestimmten Kanten oder Flächen zu bekommen, muss über die gesamte Liste iteriert werden. Dies ist sehr langsam und schränkt den praktischen Nutzen von VVMs stark ein.

Ein Vergleich und eine Abwägung der Vor- und Nachteile mit anderen Datenstrukturen findet sich in [18, Kap. 11]. Neben dem VVM kann man in beliebigen Kombinationen auch Listen für Kanten oder Faces hinzufügen. Dies erhöht zwar den Speicherbedarf und die Komplexität der Datenstruktur, bestimmte Zugriffe werden dadurch jedoch stark beschleunigt.

2.2.2 Winged Edge

In der Winged-Edge-Datenstruktur [19] werden sowohl Vertices als auch Edges und Faces explizit abgespeichert. Diese Repräsentation ermöglicht es, die Geometrie des Meshes im Vergleich zu anderen Strukturen leichter zu verändern. Der Nachteil ist jedoch, dass der Speicheraufwand sehr hoch und die Datenstruktur im Vergleich zu den anderen Darstellungen sehr komplex ist.

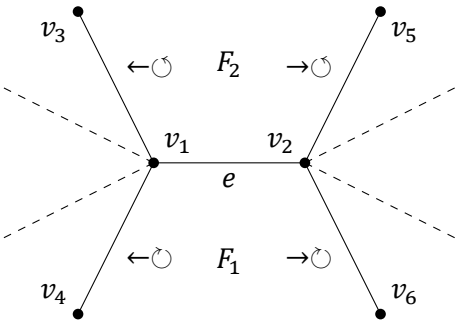


Abbildung 2.5: Vor- und Nachfolgerkanten der Winged-Edge-Datenstruktur nach [19, S. 591]

Bei der Winged-Edge-Darstellung werden in der Kantentabelle jeweils die vorhergehenden und nachfolgenden Kanten im und gegen den Uhrzeigersinn gespeichert. Dies ermöglicht die schnelle Bestimmung von angrenzenden Kanten, Vertices oder Faces, bedeutet aber gleichzeitig auch viel Verwaltungsaufwand der Indizes. In Abbildung 2.5 findet sich eine Übersicht über die entsprechenden Kanten.

Die Winged-Edge-Datenstruktur zum Beispiel in Abbildung 2.4 ist in Tabelle 2.2 zu sehen.

Edges						
Edge	Vertices	Faces	←↺	←↻	→↺	→↻
e_1	v_1, v_2	F_1	e_3	e_3	e_2	e_4
e_2	v_2, v_3	F_1, F_2	e_1	e_4	e_3	e_5
e_3	v_3, v_1	F_1	e_2	e_5	e_1	e_1
e_4	v_2, v_4	F_2	e_2	e_1	e_5	e_5
e_5	v_4, v_3	F_2	e_4	e_4	e_2	e_3

Vertices	
Vertex	Edges
v_1	e_1, e_3
v_2	e_1, e_2, e_4
v_3	e_2, e_3, e_5
v_4	e_4, e_5

Faces	
Face	Edges
F_1	e_1, e_2, e_3
F_2	e_4, e_5, e_2

Tabelle 2.2: Vertex-, Edge- und Face-Tabellen bei der Winged-Edge-Darstellung

2.2.3 Half Edge

Im Gegensatz zu bisherigen Modellen ist die Idee bei der Half-Edge-Datenstruktur, die Edges in jeweils zwei Halbkanten aufzuteilen. Eine Halbkante hat somit einen Vor- und Nachfolger, sowie einen gegenüberliegenden Nachbarn.

Der Vorteil dieser Art der Speicherung ist, dass sowohl Kantenvorgänger und -nachfolger schnell bestimmt werden können, aber insbesondere auch aneinander angrenzende Faces. Das Hinzufügen, Entfernen oder Ändern von Daten ist jedoch leichter als bei der Winged-Edge-Darstellung, da nicht zu jeder Kante 4 Nachbarkanten und alle Faces gespeichert werden müssen.

Die Vertex- Edge- und Face-Tabellen zu Abbildung 2.4 sind in Tabelle 2.3 aufgelistet. Statt der vollständigen Außen- bzw. Innenzyklen können bei den Faces auch nur einzelne Halbkanten gespeichert werden. Die entsprechenden Vorgänger- und Nachfolgerhalbkanten repräsentieren den kompletten Zyklus implizit.

Half-Edges					
Edge	← Edge	→ Edge	Twin	Origin	Face
e_{1a}	e_{3a}	e_{2a}	e_{1b}	v_1	F_1
e_{1b}	e_{4b}	e_{3b}	e_{1a}	v_2	-
e_{2a}	e_{1a}	e_{3a}	e_{2b}	v_2	F_1
e_{2b}	e_{5a}	e_{4a}	e_{2a}	v_3	F_2
e_{3a}	e_{2a}	e_{1a}	e_{3b}	v_3	F_1
e_{3b}	e_{1b}	e_{5b}	e_{3a}	v_1	-
e_{4a}	e_{2b}	e_{5a}	e_{4b}	v_2	F_2
e_{4b}	e_{5b}	e_{1b}	e_{4a}	v_4	-
e_{5a}	e_{4a}	e_{2b}	e_{5b}	v_4	F_2
e_{5b}	e_{3b}	e_{4b}	e_{5a}	v_3	-

Vertices	
Vertex	ausgehend
v_1	e_{1a}, e_{3b}
v_2	e_{1b}, e_{2a}, e_{4a}
v_3	e_{2b}, e_{3a}, e_{5b}
v_4	e_{4b}, e_{5a}

Faces		
Face	Außenzyklus	Innenzyklus
F_1	e_{1b}, e_{3b}, e_{2b}	e_{1a}, e_{2a}, e_{3a}
F_2	e_{4b}, e_{2a}, e_{5b}	e_{2b}, e_{4a}, e_{5a}

Tabelle 2.3: Vertex-, Edge- und Face-Tabellen bei der Half-Edge-Darstellung nach [20]

2.3 Point Cloud Library

Bei der Point Cloud Library (PCL) [21] handelt es sich um eine Bibliothek, welche die Arbeit mit 3D-Punktwolken immens vereinfacht. Sie bietet Möglichkeiten zur Filterung, Segmentierung, Oberflächenrekonstruktion und Visualisierung von Punktwolken, sowie weitere Module.

Neben den zahlreichen unterstützten Anwendungsgebieten und implementierten Algorithmen bietet die PCL den weiteren entscheidenden Vorteil, dass sie direkt zu ROS kompatibel ist. Die Kompatibilität wird durch ROS-Nodelets im Paket `perception_pcl` hergestellt, sodass Punktwolken, Meshes oder andere Datenstrukturen direkt per Message versendet werden können.

3. Vorhandene Arbeit

In diesem Kapitel werden die Kernaufgaben dieser Arbeit behandelt: Registrierung von mehreren Punktwolken in einem Koordinatensystem (3.1), Segmentierung in einzelne Cluster (3.2) und Triangulation der Wolke zu einem Polygonnetz (3.3).

Neben Erläuterung der Problematik wird ein Blick auf die aktuelle Forschung zum jeweiligen Thema geworfen. Außerdem werden mögliche Lösungsansätze auf ihren praktischen Nutzen untersucht und die Vor- und Nachteile diskutiert.

3.1 Registrierung von Punktwolken

Beim Erfassen von Objekten in der Realität, beispielsweise mithilfe eines Laserscanners, werden häufig Punktwolken aus verschiedenen Perspektiven aufgenommen. Der Grund dafür ist, dass aus einer Scanposition nicht immer die gesamte Oberfläche sichtbar ist, etwa aufgrund von Verschattungen durch das Objekt selbst. Weiterhin können so Reflektionen und spiegelnde Oberflächen wie auch Hintergrundrauschen gemittelt werden, sodass weitgehend robuste Daten erhalten werden.

In vielen Fällen ist die zugehörige Aufnahmeposition nicht bekannt, unzureichend genau oder es liegt kein kalibriertes System vor. Aus diesem Grund müssen die erfassten Punktwolken erst in ein globales Koordinatensystem gebracht werden. Diesen Prozess nennt man Registrierung.

3.1.1 Lokale Registrierung

Bei der lokalen Registrierung von zwei Punktwolken müssen diese bereits grob aneinander ausgerichtet sein. Der Registrierungsalgorithmus verfeinert diese Ausrichtung dann weiter und findet ein lokales Minimum. Einer der bekanntesten Algorithmen zur lokalen Registrierung ist Iterative Closest Point (ICP) und seine vielen verschiedenen Varianten.

3.1.1.1 Iterative Closest Point

ICP [22] ist der wohl bekannteste Algorithmus zur lokalen Registrierung von Punktwolken. Im Laufe der Zeit wurden zahlreiche Varianten und Optimierungen dafür entwickelt, die beispielsweise auch die Normalen in den Punkten in Betracht ziehen oder besonders resistent gegen statistische Ausreißer sind [23–25]. Unter anderem existiert auch eine vorhandene Implementierung in der PCL [26].

Algorithmus 1 beschreibt, wie Punktwolke A an Punktwolke B registriert werden kann. Der Ablauf lässt sich folgendermaßen grob erklären:

1. A wird iterativ durch Anwendung von Translationen und Rotationen in eine neue Position gebracht, zum Beispiel mithilfe von Quaternionen [27].
2. Zu jedem Punkt $p \in A$ wird der Punkt $q \in B$ gesucht, der den geringsten Abstand $e = \|p - q\|$ von p hat.
3. Berechne den Fehler $\varepsilon = \sum e^2$, Standard ist der Mean Squared Error (MSE).
4. Wiederholung, bis eine der Abbruchbedingungen eintritt.

Die Abbruchbedingungen sind dabei je nach Implementierung unterschiedlich. Meist terminiert der Algorithmus jedoch, wenn

- $\varepsilon < \alpha$ mit Grenzwert $\alpha > 0$ für den Restfehler
- $\Delta e > \delta$, mit $\delta > 0$ als Grenzwert für die Änderung des Fehlers
- eine festgelegte Zahl von Iterationen abgelaufen ist.

Selbstverständlich lassen sich beliebig viele zusätzliche Kriterien definieren, beispielsweise eine festgelegte maximale Laufzeit.

Algorithm 1 Iterative Closest Point

function ICP($cloud1, cloud2$)
  $PointMap \leftarrow \emptyset$
  for all $p \in cloud1$ **do** // Find corresponding points
    $q \leftarrow$ findNearestNeighbor($cloud2, p$)
    if dist(p, q) $< maxDist$ **then**
      $PointMap$.add(p, q)
    end if
  end for
  while termination Criteria not met **do** // Now, align the clouds
    $t \leftarrow$ estimateTransformation()
    $cloud1$.transform(t)
    $\varepsilon \leftarrow$ calculateError($PointMap$)
  end while
end function
function calculateError($PointMap$)
  $\varepsilon \leftarrow 0$
  for all $p, q \in PointMap$ **do**
    $e \leftarrow \|p - q\| = \sqrt{(p_x - q_x)^2 + (p_y - q_y)^2 + (p_z - q_z)^2}$
    $\varepsilon \leftarrow \varepsilon + e^2$
  end for
  return ε
end function

3.1.1.2 Kinect Fusion

Durch die Veröffentlichung der Microsoft Kinect im Jahr 2010 wurde aufgrund ihres niedrigen Preises erstmals der breiten Masse der Zugang zu Tiefenkameras ermöglicht [28, 1:55]. Die Kinect ist eine 3D-Kamera, welche ursprünglich für die Nutzung im Entertainment- und Gamingbereich entwickelt worden ist. Bestehend aus einem Lichtemitter, Infrarotsensor, einer 2D-RGB-Kamera und weiteren Sensoren, liefert sie 3D-Daten bei einer Bildwiederholfrequenz von 30 Hz. Neben dem ursprünglichen Anwendungsgebiet findet sie heute auch in vielen anderen, auch wissenschaftlichen Bereichen Anwendung, beispielsweise bei der Aufzeichnung geomorphologischer Daten [29].

Kinect Fusion (KinFu) ist das Ergebnis einer Forschungsarbeit von Microsoft Research [30, 31] und bietet eine Möglichkeit, mithilfe der Kinect 3D-Rekonstruktionen in Echtzeit durchzuführen. KinFu kombiniert dabei die lokale Registrierung durch den ICP-Algorithmus mit einem Truncated Signed Distance Field (TSDF). Ein TSDF ist im Grunde genommen nichts anderes als ein Voxel Grid. Während der Wert eines Voxel im

ursprünglichen Voxel Grid Informationen über die Daten im betreffenden Voxel enthält, wird hier die Entfernung zur nächsten Oberfläche gespeichert [32]. Somit stehen innerhalb des Objekts negative Werte in den Voxeln und außerhalb positive Werte. Die Oberfläche wird damit implizit durch den Nulldurchgang der Distanzen in den Voxeln definiert.

Dies ermöglicht bei einer relativ geringen Auflösung dennoch eine recht genaue Rekonstruktion der vorhandenen Oberflächen. Um dies zu erreichen, werden die Distanzen in den Voxeln interpoliert und Fehler durch die geringe Auflösung oder Sensorrauschen minimiert.

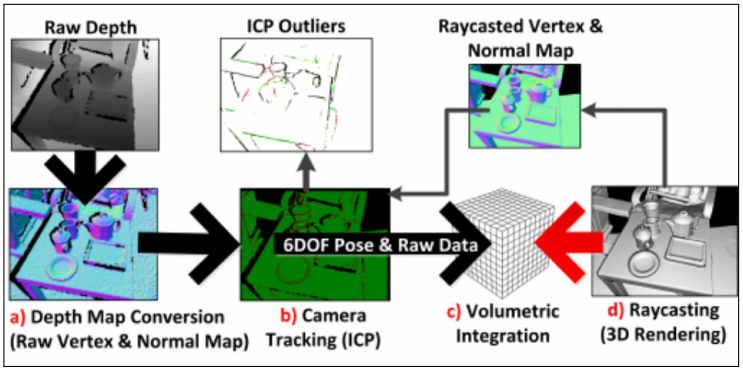


Abbildung 3.1: Integration eines Tiefenbilds in KinFu. Entnommen aus [30]

Durch eine schnelle GPU-Implementierung auf Basis von Compute Unified Device Architecture (CUDA) erreicht KinFu dabei eine Integration neuer Tiefenbilder bei 30 Hz, der Bildwiederholfrequenz der Kinect. Somit ist die zeitliche Differenz zwischen zwei Aufnahmen sehr niedrig. Es wird davon ausgegangen, dass die Kamera handgeführt (bzw. mit geringer Geschwindigkeit bewegt) wird, daher hält sich auch die räumliche Distanz zwischen einzelnen Frames in Grenzen. Aufgrund dieser Gegebenheiten reicht bei KinFu eine lokale Registrierung aus, es kann also direkt ICP verwendet werden.

In Abbildung 3.1 ist der Ablauf der Integration eines neuen Tiefenbilds ins TSDF bei KinFu dargestellt. Das Tiefenbild wird zunächst in eine Punktwolke konvertiert. Diese wird anschließend durch ICP registriert, um dann unter Bildung eines Mittelwerts in das TSDF integriert zu werden. Zur Darstellung der Szene wird dieses anschließend mithilfe eines Raycasters gerendert.

Es gibt zahlreiche Optimierungen und veränderte Versionen von KinFu. Unter anderem gibt es in der PCL eine freie Implementierung [33]. Eine optimierte Version ist beispielsweise Chisel [34], wo eine effizientere Voxelstrategie verwendet wird. Weiterhin ist Chisel eine reine CPU-Implementierung, was die Nutzung auf Mobilgeräten ohne Grafikkarte ermöglicht.

Die in dieser Arbeit verwendete Implementierung ist Yet Another Kinect Fusion (YAK). YAK ist eine auf ROS angepasste Version von KinFu, um Trajectory Waypoints für die Robotersteuerung zu errechnen [35]. Die Verwendung von KinFu in industriellen Robotikanlagen ist in vielerlei Hinsicht sinnvoll:

- Objekte können aus mehreren Perspektiven gescannt werden, was eine bessere Darstellung der Szene liefert.
- Rauschen durch den Bildsensor wird aufgrund der Mittelwertbildung minimiert. Das Ergebnis ist ein glatteres und realistischeres Modell.
- Durch die GPU-Implementierung können Tiefenbilder in Echtzeit integriert werden, was insbesondere bei Robotikanwendungen ein großer Vorteil ist.
- Ein häufig in der Bildverarbeitung auftretendes Problem sind reflektierende bzw. spiegelnde Oberflächen. Die betroffenen Regionen können oft nur falsch, verzerrt oder gar nicht modelliert werden. Da hier eine Aufnahme aus mehreren Perspektiven möglich ist, können derartige Fehler weitestgehend vermieden werden. In Abbildung 3.2 ist dies an einem Beispiel besonders gut sichtbar.

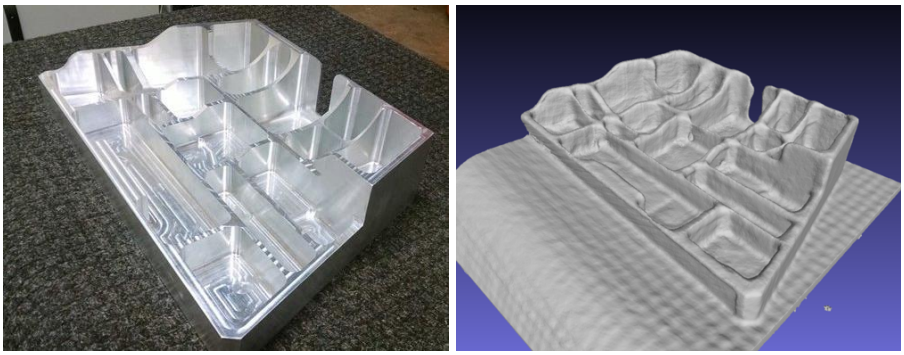


Abbildung 3.2: Szene und Rekonstruktion eines reflektierenden Objekts durch YAK. Entnommen aus [35]

Der Vorteil der Echtzeitintegration ist jedoch gleichzeitig auch ein großes Problem. In industriellen Robotikanlagen sind nur selten Grafikkarten verbaut. Eine Hochleistungs-GPU-Einheit mit CUDA-Unterstützung ist dabei aber unerlässlich, da sonst nur ein Bruchteil der Geschwindigkeit erreicht werden kann.

3.1.2 Globale Registrierung

Bei der globalen Registrierung müssen sich die beiden Punktwolken nicht nahe der endgültigen Ausrichtung befinden - Translation und Rotation können beliebig sein. Ein Nachteil ist jedoch, dass die globale Registrierung oft nur eine Grobregistrierung bietet, also keine optimalen Ergebnisse liefert. Aus diesem Grund bietet es sich meist an, anschließend noch eine lokale Registrierung zur Verbesserung der Ergebnisse durchzuführen.

Zur globalen Registrierung von Punktwolken existieren verschiedene Ansätze [36–38]. Die in dieser Arbeit verwendete Methode ist Super4PCS [39], eine optimierte Version von Four-Points Congruent Sets (4PCS) [40]. Dieser Ansatz wird verwendet, da die vorhandene Implementierung Open Global Registration (OpenGR) [41] über einen Wrapper bereits zur PCL kompatibel ist.

3.2 Segmentierung

Ziel der Segmentierung ist es, zusammengehörende Bildregionen zu identifizieren und durch Zuweisung verschiedener Klassen voneinander zu trennen. Dies ist, insbesondere in der 2D-Bildverarbeitung, ein altes Problem, für das es viele verschiedene Lösungsansätze gibt. Auch im dreidimensionalen Raum ist eine Segmentierung häufig notwendig. In vielen Fällen gibt es dabei nicht eine beste Methode - die Objektart, -größe, -form und viele weitere Faktoren beeinflussen die Wahl eines geeigneten Ansatzes.

Die Forschung ist in diesem Bereich weiterhin sehr aktuell: Beispielsweise zeigen Ückermann, Haschke und Ritter einen Versuch, in Echtzeit und ohne vorher bekannte Objekte zu segmentieren [42]. Eine Übersicht bieten Nguyen und Le, die verschiedene Varianten vergleichen und die Vor- und Nachteile diskutieren [43]. In den letzten Jahren findet auch verstärkt Forschung zur Segmentierung mithilfe von Neuronalen Netzen statt [44].

Für die Segmentierung können viele Informationen über die vorhandene Szene relevant sein. Neben der reinen geometrischen Information können zum Beispiel auch Textur (also RGB-Werte), Normalenorientierung oder Topologie hilfreich sein, um zusammenhängende Regionen voneinander unterscheiden zu können.

Die Segmentierung muss dabei keineswegs immer auf 2D-Bildern oder 3D-Punktwolken

stattfinden. In [45] wird beispielsweise eine Übersicht über Segmentierungsverfahren auf 3D-Meshes vorgestellt. Weiterhin müssen auch nicht zwingend unterschiedliche Objekte getrennt werden. Das Ziel kann es auch sein, nur eine Trennung verschiedener Partitionen innerhalb eines Objekts zu erzielen, zum Beispiel um eine Hand am menschlichen Körper zu identifizieren [46].

3.2.1 Segmentierung im 3D

Im dreidimensionalen Raum gibt es viele verschiedene Eigenschaften (Features), die zur Segmentierung von Punktwolken herangezogen werden können. Es spielt selbstverständlich die Distanz zwischen den Punkten eine Rolle, aber auch andere Faktoren wie Normalenorientierung, Farbe oder Krümmung bieten in vielen Fällen nützliche Zusatzinformationen.

Im Folgenden werden die zwei getesteten Methoden Euclidean Cluster Extracion (ECE) und Region Growing zur Segmentierung von dreidimensionalen Punktwolken erklärt. Eine gute Übersicht über die beiden Verfahren bietet auch Rusu in [47, S. 88–93].

3.2.1.1 Euclidean Cluster Extracion

Diese Methode der Segmentierung im dreidimensionalen Raum ist sehr einfach. Sie basiert auf der Annahme, dass Punkte so lange zu einem Cluster gehören, so lange ihr Abstand unter einem Grenzwert liegt.

In Algorithmus 2 wird beschrieben, wie die Cluster $\mathcal{C} \subseteq P$ für die Punktwolke P nach [47, S. 89–90] errechnet werden. Der Threshold δ gibt dabei den minimalen Abstand zwischen zwei Clustern an.

Nach der Berechnung der verschiedenen Cluster werden diese noch gefiltert. Dabei werden nur Cluster mit $\theta_{min} < |\mathcal{C}| < \theta_{max}$ erhalten, alle anderen werden verworfen. Die untere Grenze θ_{min} dient dabei der Robustheit gegen Rauschen und statistische Ausreißer, während die obere Grenze θ_{max} verhindert, dass untersegmentierte Regionen zurückgegeben werden.

Algorithm 2 Euclidean Cluster Extracion

$\mathcal{C} \leftarrow \emptyset$ // list of clusters
$\mathcal{Q} \leftarrow \emptyset$ // current cluster queue
for all $p \in P$ **do**
  $\mathcal{Q} \leftarrow p$
  for all unprocessed points $q \in \mathcal{Q}$ **do**
    mark q as processed
    $\mathcal{N} \leftarrow$ nearestKSearch(q, δ) // δ-region around q
    $\mathcal{Q} \leftarrow$ all unprocessed points in $\mathcal{N}$
  end for
  $\mathcal{C} \leftarrow \mathcal{Q}$ // add processed queue to clusters
  $\mathcal{Q} \leftarrow \emptyset$ // reset the queue
end for
return $\mathcal{C}$

3.2.1.2 Region Growing

Ein großer Nachteil von ECE ist, dass nur die räumliche Distanz betrachtet wird. Bei nicht optimal getrennten Regionen wird die Punktwolke daher untersegmentiert, und aufgrund der Filterung am Ende werden viele Cluster wieder verworfen. Dies ist auch bei den meisten Anwendungen in der Realität der Fall. Selbst bei gesetzter Region of Interest (RoI) und somit bereits gefiltertem Hintergrund liegen Objekte häufig nahe beieinander.

Der Region-Growing-Algorithmus umgeht dieses Problem, indem statt der Distanzmetrik die Krümmung im entsprechenden Punkt betrachtet wird. Unter der Krümmung versteht man die Änderung der Normalen in einem Punkt. Eine gute Approximation liefert die in Gleichung 3.1 dargestellte Metrik. Weiterhin spielt auch die Glattheit der Oberfläche eine Rolle (siehe dazu Definition 2).

Definition 1 *Sei $p \in P, P \subset \mathbb{R}^3$. Dann sind die $k \leftarrow$ K Nearest Neighbors (KNN)*

$$Q_k(p) = \bigcup_{n=1}^{k} q_n(p), Q_k \subseteq P \setminus \{p\} \text{ mit } \|p-q_1\| \leq \|p-q_2\| \leq ... \leq \|p-q_k\| \leq ... \leq \|p-q_{|P|}\|.$$

Sei E_k nun eine Ebene, welche die Punkte $Q_k(p)$ nach least squares [48], RANSAC [49] oder anders approximiert. Dann ist

$$\kappa = r^2 = \|p - E_k\|^2, \tag{3.1}$$

also der MSE zwischen E_k und p, eine Metrik zur Schätzung der Krümmung in p (nach [50, Abs. 2.1.2]).

Definition 2 *Sei f eine Funktion. Dann gibt die Glattheit $\mathcal{G}(f)$ an, wie oft f differenzierbar ist:*

$$\mathcal{G}(f) = \begin{cases} 0 & f \text{ nicht differenzierbar} \\ n & f^{(n)} \text{ definiert} \\ \infty & f \text{ unendlich oft differenzierbar} \end{cases} \tag{3.2}$$

Ansonsten verfährt sich der Region-Growing-Algorithmus sehr ähnlich zu ECE. Einzig die Fehlermetrik wird geändert; statt der euklidischen Norm $\|\cdot\|_2$ werden Normalenwinkel und Krümmung betrachtet. Algorithmus 3 beschreibt den Ablauf bei maximaler Krümmung ϕ, Normalendifferenz α und unter Betrachtung der k KNN.

Algorithm 3 Region Growing

```
C ← ∅          // list of clusters
Q ← ∅          // current cluster queue
sort P by ascending curvature
for all p ∈ P do
    Q ← p
    for all unprocessed points q ∈ Q do
        mark q as processed
        N ← getKNeighbors(q, k)          // k nearest neighbors of q
        for all unprocessed candidates c ∈ N do
            if curvature(c) < φ ∧ cos⁻¹(n_p, n_c) < α then
                Q ← c
            end if
        end for
    end for
    C ← Q          // add processed queue to clusters
    Q ← ∅          // reset the queue
end for
return C
```

3.2.2 Watershed-Transformation

Die bisherigen Ansätze sind Möglichkeiten zur Segmentierung in 3D-Punktwolken. Während dies häufig deutliche Vorteile hat, können auch viele Nachteile dabei das Ergebnis beeinflussen. Beispielsweise können Objekte sehr nah aneinander liegen und ECE damit erschweren. Kanten zwischen Objekten können im 2D-Grauwertbild jedoch deutlich besser sichtbar sein. Die Segmentierung auf zweidimensionalen RGB- oder Grauwer-

ten kann daher in manchen Fällen bessere Ergebnisse als eine Segmentierung auf Basis der Geometrie liefern.

Die Watershed-Transformation ist eine der bekanntesten Methoden, um eine solche Segmentierung vorzunehmen. Der Gedanke dahinter stammt aus der Geographie und beschreibt das Verhalten von ansteigendem Wasser in unebenem Terrain [51, S. 1–2]. Dabei bilden sich Wasserbecken, welche bei steigendem Pegel an bestimmten Linien zusammenlaufen. Diese Linien beschreiben im Fall des Clusterings die Grenzen zwischen unterschiedlichen Bildregionen.

Der Ablauf der Watershed-Transformation ist in Algorithmus 4 beschrieben. Dabei werden zunächst die garantierten Hintergrundpixel einheitlich auf 0 gesetzt und ein Preprocessing in Form eines Laplace-Filters vorgenommen. Weiterhin basiert die Binarisierung des Eingabebilds auf dem Thresholding nach Otsu [52], welches die Varianz σ zwischen den Grauwertbins minimiert. Hier kann auch eine beliebige andere Methode verwendet werden, jedoch liefert dieser Ansatz bei einer binären Verteilung im Histogramm optimale Ergebnisse.

Algorithm 4 Watershed-Transformation [53]

$\mathcal{I} \leftarrow input$
Set guaranteed background pixels to 0
$L \leftarrow \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix}$
filter($\mathcal{I}, L$) // sharpening using laplace filter
binarize($\mathcal{I}$, "Otsu") // binarize (e.g. Otsu's thresholding)
distanceTransform($\mathcal{I}$) // distance to next background pixel
$M \leftarrow$ local minima
$W \leftarrow$ local maxima
applyMask($input, W$) // Add region borders to original image
for all $marker \in M$ **do**
    floodFill($input, marker$)
end for

3.3 Triangulierung

Die Konvertierung einer Datenstruktur zu einem aus Polygonen bestehenden Mesh nennt sich Triangulierung. Gerade bei der Triangulierung von Punktwolken gibt es jedoch nicht immer eine einzelne korrekte Lösung dieses Problems. Es ist beispielweise

nicht immer sichergestellt, dass jeder Punkt auch tatsächlich die zu rekonstruierende Oberfläche darstellt - Rauschen, Reflektionen oder sonstige Fehler können auftreten. Weiterhin stellt sich die Frage, aus welchen der Punkten in der Wolke sich ein bestimmtes Polygon zusammensetzt.

Diese und andere Probleme führen dazu, dass es viele verschiedene Ansätze zur Triangulierung gibt.

3.3.1 Marching Cubes

Bei Marching Cubes [17] handelt es sich um einen Algorithmus, der die Triangulation einer Punktwolke auf ein Voxel Grid zurückführt. Der Ablauf ist in Algorithmus 5 beschrieben.

Die eingegebene Punktwolke wird also zunächst diskretisiert, was bei geringer Kantenlänge der Voxel zu Datenverlust führt. Anschließend wird jeder Voxel getrennt betrachtet und anhand der Eckkonfiguration die zu verbindenden Kanten mithilfe einer LUT ermittelt. Dieses Vorgehen wird nun für jeden Voxel wiederholt, was zu einer vollständigen Rekonstruktion der Oberfläche führt.

Einer der entscheidenden Nachteile bei diesem Algorithmus ist, dass die Qualität des Meshs direkt abhängig von der gewählten Kantenlänge der Voxel ist. Dies macht viele Vorteile der Punktwolke gegenüber einem Voxel Grid zunichte. Bei zu geringer Kantenlänge nimmt die Zahl der möglichen rekonstruierten Polygone ab, was zu einer geringeren Qualität führt. Wählt man die Auflösung jedoch zu hoch, treten Oversampling-Effekte auf und es entstehen Lücken im Mesh. Insbesondere bei großen Differenzen der Distanzen zwischen Punkten in der Punktwolke führt dies zu Problemen.

Weiterhin weisen die Meshes in der Standardimplementierung zackenartige Artefakte auf. Dies passiert dann, wenn die Vertices zu den entsprechenden Kanten mittig auf diesen gewählt werden. Durch eine Interpolation zwischen den Werten der beiden Eckpunkte des Voxels lässt sich dieser Effekt minimieren.

Algorithm 5 Marching Cubes [17, Abs. 4]

```
function marchingCubes
    vertexList ← ∅          // The list of output vertices (Faces implicit)
    V ← Voxels
    for all v ∈ V do
        index ← calculateIndex(v)
        edgeList ← edgeTable[index]
        for all edge ∈ edgeList do
            vertex ← interpolate(edge[0], edge[1])          // interpolate corners
            vertexList.add(vertex)
        end for
    end for
end function
function calculateIndex(voxel)
    voxelIndex ← 0
    for cornerIndex ∈ [0..8] do
        if voxel[cornerIndex] < isolevel then          // corner inside isosurface
            voxelIndex | = 1 << cornerIndex          // Set the i-th bit of the index
        end if
    end for
    return voxelIndex
end function
```

3.3.2 Poisson

Die 2006 von Kazhdan, Bolitho und Hoppe präsentierte Poisson-Rekonstruktion [54] basiert auf der impliziten Darstellung der Oberfläche als Funktion. Die mathematische Darstellung dieser Funktion lässt sich Definition 3 entnehmen.

Definition 3 *Sei M die zu rekonstruierende Oberfläche. Dann ist*

$$\chi_M(p) = \begin{cases} 0 & p \text{ außerhalb von } M \\ 1 & p \text{ innerhalb von } M \end{cases}, \forall q \in M : \chi_M(q) = 0.5 \tag{3.3}$$

die implizite Funktion, die die Oberfläche im Punkt $p \in M$ darstellt.

Der Gradient $\nabla\chi_M$ dieser Funktion nimmt also im optimalen Fall überall den Wert 0 an, außer in der Oberfläche selbst. Dieser Gradient entspricht exakt den nach innen gerichteten Normalen in Oberflächenpunkten [54, Abs. 3]. Somit gilt für die Indikatorfunktion:

$$\begin{aligned} \nabla\chi_M &= \vec{V} \\ \Leftrightarrow \quad \nabla^2\chi_M &= \nabla\vec{V} \\ \Leftrightarrow \quad \Delta\chi_M &= \nabla\vec{V} \end{aligned}$$

Dies ist eine Poisson-Gleichung [55, Kap. 2], deren Lösung sich beispielsweise mit dem Gauß-Seidel-Verfahren iterativ berechnen bzw. annäheren lässt. Da eine genaue Lösung der Poissongleichung nur in der Nähe der Oberfläche erforderlich ist, kann nun ein Octree aufgestellt werden.

Mit der Poisson-Rekonstruktion generierte Polygonmeshes weisen die Eigenschaft auf, immer wasserdicht zu sein. Dies ist in der Darstellung der Oberfläche als mehrdimensionale, kontinuierliche Funktion begründet.

Die Auflösung des Meshes lässt sich durch die maximale Baumtiefe einstellen. Mit der Tiefe und damit der Qualität steigen jedoch auch Laufzeit und Speicherbedarf drastisch an. Weiterhin kann konfiguriert werden, wie viele Punkte einem Blatt im Baum zugewiesen werden. Hohe Werte führen zu einer verringerten Auflösung, aber auch zu einem gleichzeitigen Glättungseffekt. Außerdem verringern sich aufgrund der niedrigeren Zahl an Blättern der benötigte Speicher und die Laufzeit.

3.3.3 Weitere Ansätze

Neben den bereits genannten Rekonstruktionsalgorithmen Marching Cubes und Poisson gibt es noch weitere unterschiedliche Ideen, um ein Polygonnetz zu triangulieren.

3.3.3.1 Greedy-Triangulierung

Bei der Greedy-Triangulierung der Wolke $\mathcal{P}$ werden zunächst alle Punkte $p, q \in \mathcal{P}$ miteinander verknüpft. So entstehen Kantenkandidaten, welche nun nach ihrer Länge sortiert werden. Diese werden nun nacheinander akzeptiert oder verworfen, wenn ein Schnittpunkt mit einer bereits akzeptierten Kante auftritt [56, S. 235–237].

Diese Art der Triangulierung funktioniert nur im zweidimensionalen Raum, da der Schnittpunkt zwischen zwei Geraden in höheren Dimensionen nicht als Kriterium ausreicht.

Algorithm 6 Greedy-Triangulierung

$\mathcal{C} \leftarrow \emptyset$
for all $p, q \in \mathcal{P},\ p \neq q$ **do** // add edge candidates to $\mathcal{C}$
  $\mathcal{C} \leftarrow \overline{pq}$
end for
sortByLength($\mathcal{C}$)
$\mathcal{T} \leftarrow \emptyset$ // the accepted edges
for all $e \in \mathcal{C}$ **do**
  for all $f \in \mathcal{T}$ **do**
    if $e \cap f$ **then** // e intersects f, remove from candidates
      $\mathcal{C} \leftarrow \mathcal{C} \setminus \{e\}$
    end if
  end for
  $\mathcal{T} \leftarrow \mathcal{T} \cup \{e\}$
end for
return $\mathcal{T}$

Im $\mathbb{R}^3$ werden daher die Punkte über ihre Normale auf eine Ebene projeziert. Zur Anwendung auf 3D-Punktwolken ist daher eine vorherige Normalenschätzung notwendig. Anschließend der in Algorithmus 6 dargestellte Ablauf regulär durchgeführt werden.

Ein enormer Vorteil der Greedy-Triangulierung ist ihre Geschwindigkeit. So zeigen Berg, Cheong, Kreveld und Overmars, dass ein Polygon mit n Vertices in $\mathcal{O}(n \log n)$ rekonstruiert werden kann [20, S. 56–58]. Dies ist beispielsweise für Echtzeitanwendungen relevant, wie die in [57] vorgestellte Methode demonstriert.

3.3.3.2 Delaunay-Triangulierung

Dreiecke mit mindestens einem kleinen Innenwinkel sind häufig problematisch in der Computergrafik. Beim Renderingprozess können aufgrund von Präzisions- und Rundungsfehlern Artefakte und Aliasingeffekte auftreten.

Die Delaunay-Triangulierung behebt dieses Problem weitestgehend, da sie die minimalen Innenwinkel der generierten Dreiecke maximiert [20, S. 199]. Eine weitere Eigenschaft ist, dass kein Umkreis eines der Dreiecke einen Knoten beinhaltet, der nicht eine Ecke des Dreiecks ist [20, S. 196]. Der Ablauf der Delaunay-Triangulierung ist in Algorithmus 7 zu sehen.

Algorithm 7 Delaunay-Triangulierung [20, S. 200–201]

$\mathcal{P} \leftarrow$ list of n vertices $p_1, \ldots, p_n$
$\mathcal{T} \leftarrow$ a triangle p_0qr with $p_0 \in \mathcal{P}$ so that $\forall p \in \mathcal{P} : p$ inside p_0qr
$\mathcal{P} \leftarrow \mathcal{P} \setminus \{p_0\}$
while $\mathcal{P} \neq \emptyset$ **do**
    $p_r \leftarrow$ pop($\mathcal{P}$)
    find triangle $t \in \mathcal{T}$ so that p_r is inside t
    $\mathcal{T} \leftarrow \mathcal{T} \setminus \{t\}$
    if p_r is on edge $t_1t_2 \in t$ **then**
        find vertices f_1 and f_2 of the two triangles seperated by t_1t_2
        create triangles $t_1p_rf_1$, $t_1f_2p_r$, $t_2f_1p_r$ and $t_2p_rf_2$
        check the untouched edges using checkFlipped(p_r, edge)
        add the triangles to $\mathcal{T}$
    else
        p_r is inside $t = t_1t_2t_3$
        split t into three triangles with p_r as vertex
        check the old edges using checkFlipped(p_r, edge)
        add them to $\mathcal{T}$
    end if
    remove all triangles $t \in \mathcal{T}$ which have q or r as vertex
end while
return $\mathcal{T}$
function checkFlipped(new vertex p, edge t)
    if the smallest angle is bigger after a flip **then**
        get two triangles $a, b \in \mathcal{T}$ separated by t
        connect vertices $p_1 \in a$ and $p_2 \in b$ not connected by t
        remove t
        check old edges not ending at p using checkFlipped(p, edge)
    end if
end function

3.3.3.3 Ball Pivoting

Die Ball-Pivoting-Rekonstruktion liefert eine sehr anschauliche und intuitive Möglichkeit, eine Punktwolke zu triangulieren. Dabei wird ein imaginärer Ball mit Radius ρ auf ein Seed-Dreieck gelegt und nacheinander über die Kanten gerollt [58]. Trifft er dabei auf einen weiteren Punkt, wird aus diesem und der bereits existierenden Kante ein neues Dreieck gebildet. Falls er auf keinen Punkt trifft, handelt es sich um eine Außenkante, die nicht weiter betrachtet wird. Der genaue Ablauf ist in Algorithmus 8 dargestellt.

Algorithm 8 Ball Pivoting [58, Abs. 4]

$\rho \leftarrow$ the ball radius
$\mathcal{F} \leftarrow$ seed triangle
while true **do**
  while $e_{i,j} \leftarrow$ getActiveEdge($\mathcal{F}$) **do**
    if $\sigma_k \leftarrow$ ballPivot($e_{i,j}$) $\wedge$ (notUsed(σ_k) $\vee$ onFront(σ_k)) **then**
      do stuff
    end if
  end while
end while

Ein Nachteil des Ball-Pivoting-Algorithmus ist, dass bei zu groß gewählten Radii einige Details verloren gehen können. Dies tritt insbesondere dann auf, wenn die rekonstruierte Oberfläche stark konkav ist und die Abstände zwischen den Punkten deutlich kleiner als der gewählte Radius sind.

Um diesem Fall vorzubeugen kann man mehrere Triangulierungen mit unterschiedlichen Werten für ρ kombinieren. Aufgrund der mehrfachen Ausführung führt dies jedoch zu einer deutlich erhöhten Laufzeit.

4. Implementierung

4.1 Aufbau

4.2 Interaktion und Parametrisierung

Die Interaktion zwischen Mikado und dem Rekonstruktionstool findet über ROS statt. Das Tool ist dabei ein `ros::Subscriber` auf das von Mikado veröffentlichte Topic `/point_cloud`. Bei Aufnahme einer neuen Punktwolke wird somit das Durchlaufen der Pipeline automatisch angestoßen.

Weiterhin lassen sich über `rosparam set` und `rosparam get` Parameter setzen oder auslesen. Diese werden dann im Programm geparst und - falls nicht vorhanden - durch Standardwerte ersetzt. Dies bietet mehrere Vorteile gegenüber anderen Optionen:

- Parameter können optional über die Konsole, auf Programmebene oder auch gar nicht gesetzt werden.
- Das Programm muss nicht für jede Konfiguration neu kompiliert werden und kann bei Laufzeit parametrisiert werden.
- Die Interaktion mit anderen Programmen ist durch die Verwendung der ROS-Schnittstelle erheblich vereinfacht.

Auch die Triangulation wird durch einen externen Trigger angestoßen. Dieser wird als `rosservice` zur Verfügung gestellt, welcher mit dem Output-Dateipfad aufgerufen wird. Auch hier ist durch die Verwendung der ROS-basierten Schnittstelle eine hohe Flexibilität gegeben.

4.3 Pipelineablauf

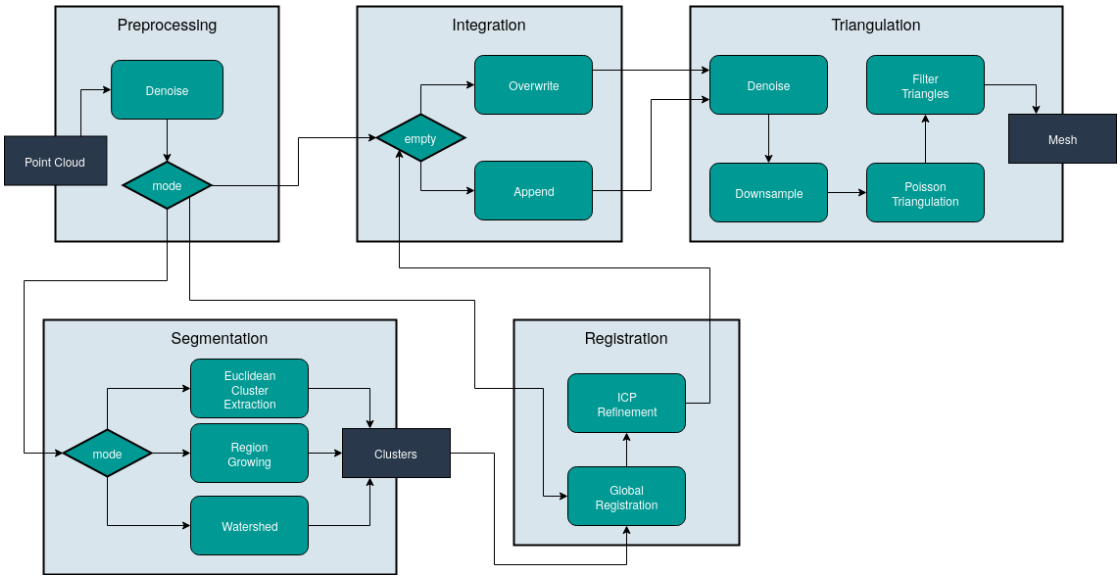


Abbildung 4.1: Ablauf der Pipeline

5. Auswertung

6. Fazit

[59]

Literatur

[1] *Isys Vision GmbH*. Adresse: https://www.isys-vision.de/ (besucht am 30. 03. 2020).

[2] *Mikado - 3D Bin Picking System for all applications*. Adresse: https://www.mikado-robotics.com (besucht am 30. 03. 2020).

[3] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler und A. Y. Ng, "ROS: an open-source Robot Operating System", in *ICRA workshop on open source software*, Kobe, Japan, Bd. 3, 2009, S. 5.

[4] *Ensenso - Stereo 3D Cameras*. Adresse: https://www.ensenso.com (besucht am 30. 03. 2020).

[5] B. H. Drost und M. Ulrich, *Recognition and pose determination of 3D objects in 3D scenes*, US Patent 8,830,229, Sep. 2014.

[6] E. Catmull, "A subdivision algorithm for computer display of curved surfaces", UTAH UNIV SALT LAKE CITY SCHOOL OF COMPUTING, Techn. Ber., 1974.

[7] A. Arsalan Soltani, H. Huang, J. Wu, T. D. Kulkarni und J. B. Tenenbaum, "Synthesizing 3d shapes via modeling multi-view depth maps and silhouettes with deep generative networks", in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2017, S. 1511–1519.

[8] A. Redert, R.-P. Berretty, C. Varekamp, O. Willemsen, J. Swillens und H. Driessen, "Philips 3D solutions: From content creation to visualization", in *Third International Symposium on 3D Data Processing, Visualization, and Transmission (3DPVT'06)*, IEEE, 2006, S. 429–431.

[9] K. Muller, P. Merkle und T. Wiegand, "3-D video representation using depth maps", *Proceedings of the IEEE*, Jg. 99, Nr. 4, S. 643–656, 2010.

[10] F. van der Lijn, T. Den Heijer, M. M. Breteler und W. J. Niessen, "Hippocampus segmentation in MR images using atlas registration, voxel classification, and graph cuts", *Neuroimage*, Jg. 43, Nr. 4, S. 708–720, 2008.

[11] S. Klein, M. Staring, K. Murphy, M. A. Viergever und J. P. Pluim, “Elastix: a toolbox for intensity-based medical image registration”, *IEEE transactions on medical imaging*, Jg. 29, Nr. 1, S. 196–205, 2009.

[12] M. Mohanty, P. Atrey und W. T. Ooi, “Secure cloud-based medical data visualization”, in *Proceedings of the 20th ACM international conference on Multimedia*, 2012, S. 1105–1108.

[13] A. Roche, G. Malandain, N. Ayache und S. Prima, “Towards a better comprehension of similarity measures used in medical image registration”, in *International Conference on Medical Image Computing and Computer-Assisted Intervention*, Springer, 1999, S. 555–566.

[14] S. Chmielewski und P. Tompalski, “Estimating outdoor advertising media visibility with voxel-based approach”, *Applied Geography*, Jg. 87, S. 1–13, 2017.

[15] *Downsampling a PointCloud using a VoxelGrid filter*. Adresse: https://pcl.readthedocs.io/projects/tutorials/en/latest/voxel_grid.html (besucht am 08. 07. 2020).

[16] Stanford University Computer Graphics Laboratory, *Stanford Bunny*. Adresse: http://graphics.stanford.edu/data/3Dscanrep/ (besucht am 30. 04. 2020).

[17] W. E. Lorensen und H. E. Cline, “Marching cubes: A high resolution 3D surface construction algorithm”, *ACM siggraph computer graphics*, Jg. 21, Nr. 4, S. 163–169, 1987.

[18] C. Smith, *On vertex-vertex systems and their use in geometric and biological modelling*. University of Calgary, 2006.

[19] B. G. Baumgart, “A polyhedron representation for computer vision”, in *Proceedings of the May 19-22, 1975, national computer conference and exposition*, 1975, S. 589–596.

[20] M. de Berg, O. Cheong, M. van Kreveld und M. Overmars, *Computational Geometry: Algorithms and Applications*. Springer-Verlag, Berlin / Heidelberg / New York, 2008, S. 31–32, isbn: 978-3-540-77974-2. doi: 10.1007/978-3-540-77974-2.

[21] R. B. Rusu und S. Cousins, “3d is here: Point cloud library (pcl)”, in *2011 IEEE international conference on robotics and automation*, IEEE, 2011, S. 1–4.

[22] P. J. Besl und N. D. McKay, “Method for registration of 3-D shapes”, in *Sensor fusion IV: control paradigms and data structures*, International Society for Optics und Photonics, Bd. 1611, 1992, S. 586–606.

[23] S. Rusinkiewicz und M. Levoy, “Efficient variants of the ICP algorithm”, in *Proceedings Third International Conference on 3-D Digital Imaging and Modeling*, IEEE, 2001, S. 145–152.

[24] S. Bouaziz, A. Tagliasacchi und M. Pauly, “Sparse iterative closest point”, in *Computer graphics forum*, Wiley Online Library, Bd. 32, 2013, S. 113–123.

[25] D. Münch, B. Combès und S. Prima, “A modified ICP algorithm for normal-guided surface registration”, in *Medical Imaging 2010: Image Processing*, International Society for Optics und Photonics, Bd. 7623, 2010, 76231A.

[26] D. Holz, A. Ichim, F. Tombari, R. Rusu und S. Behnke, “Registration with the Point Cloud Library - A Modular Framework for Aligning in 3-D”, *IEEE Robotics & Automation Magazine*, Jg. 22, S. 110–124, Dez. 2015. doi: 10.1109/MRA.2015.2432331.

[27] B. K. Horn, “Closed-form solution of absolute orientation using unit quaternions”, *Josa a*, Jg. 4, Nr. 4, S. 629–642, 1987.

[28] D. Kim, *28c3: KinectFusion*. Adresse: https://www.youtube.com/watch?v=RvrCAw1IFG0 (besucht am 09. 04. 2020).

[29] K. D. Mankoff und T. A. Russo, “The Kinect: a low-cost, high-resolution, short-range 3D camera”, *Earth Surface Processes and Landforms*, Jg. 38, Nr. 9, S. 926–936, 2013.

[30] S. Izadi, D. Kim, O. Hilliges, D. Molyneaux, R. Newcombe, P. Kohli, J. Shotton, S. Hodges, D. Freeman, A. Davison u. a., “KinectFusion: real-time 3D reconstruction and interaction using a moving depth camera”, in *Proceedings of the 24th annual ACM symposium on User interface software and technology*, 2011, S. 559–568.

[31] M. Research, *KinectFusion: Real-time 3D Reconstruction and Interaction Using a Moving Depth Camera*. Adresse: https://www.youtube.com/watch?v=KOUSSlKUJ-A (besucht am 09. 04. 2020).

[32] B. Curless und M. Levoy, “A volumetric method for building complex models from range images”, in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, 1996, S. 303–312.

[33] M. Pirovano, “Kinfu–an open source implementation of Kinect Fusion+ case study: implementing a 3D scanner with PCL”, *Project Assignment*, 2012.

[34] M. Klingensmith, I. Dryanovski, S. Srinivasa und J. Xiao, “Chisel: Real Time Large Scale 3D Reconstruction Onboard a Mobile Device using Spatially Hashed Signed Distance Fields”, in *Robotics: science and systems*, Bd. 4, 2015, S. 1.

[35] J. Schornak, *YAK: 3D Reconstruction in ROS2*, Aug. 2019. Adresse: https://rosindustrial.org/news/2019/8/7/yak-3d-reconstruction-in-ros2 (besucht am 27. 05. 2020).

[36] K. N. Chaudhury, Y. Khoo und A. Singer, “Global registration of multiple point clouds using semidefinite programming”, *SIAM Journal on Optimization*, Jg. 25, Nr. 1, S. 468–501, 2015.

[37] Q.-Y. Zhou, J. Park und V. Koltun, “Fast global registration”, in *European Conference on Computer Vision*, Springer, 2016, S. 766–782.

[38] R. B. Rusu, N. Blodow und M. Beetz, “Fast point feature histograms (FPFH) for 3D registration”, in *2009 IEEE international conference on robotics and automation*, IEEE, 2009, S. 3212–3217.

[39] N. Mellado, D. Aiger und N. J. Mitra, “Super 4PCS Fast Global Pointcloud Registration via Smart Indexing”, *Computer Graphics Forum*, Jg. 33, Nr. 5, S. 205–215, 2014, issn: 1467-8659. doi: 10.1111/cgf.12446. Adresse: http://dx.doi.org/10.1111/cgf.12446.

[40] D. Aiger, N. J. Mitra und D. Cohen-Or, “4-points Congruent Sets for Robust Surface Registration”, *ACM Transactions on Graphics*, Jg. 27, Nr. 3, #85, 1–10, 2008.

[41] N. Mellado, *OpenGR: a 3D Global Registration Library*, 2018. Adresse: https://github.com/STORM-IRIT/OpenGR (besucht am 06. 07. 2020).

[42] A. Ückermann, R. Haschke und H. Ritter, “Real-time 3D segmentation of cluttered scenes for robot grasping”, in *2012 12th IEEE-RAS International Conference on Humanoid Robots (Humanoids 2012)*, IEEE, 2012, S. 198–203.

[43] A. Nguyen und B. Le, “3D point cloud segmentation: A survey”, in *2013 6th IEEE conference on robotics, automation and mechatronics (RAM)*, IEEE, 2013, S. 225–230.

[44] G. Te, W. Hu, A. Zheng und Z. Guo, “Rgcnn: Regularized graph cnn for point cloud segmentation”, in *Proceedings of the 26th ACM international conference on Multimedia*, 2018, S. 746–754.

[45] A. Shamir, “A survey on mesh segmentation techniques”, in *Computer graphics forum*, Wiley Online Library, Bd. 27, 2008, S. 1539–1556.

[46] L. Shapira, A. Shamir und D. Cohen-Or, “Consistent mesh partitioning and skeletonisation using the shape diameter function”, *The Visual Computer*, Jg. 24, Nr. 4, S. 249, 2008.

[47] R. B. Rusu, “Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments”, Diss., Computer Science department, Technische Universitaet Muenchen, Germany, 2009.

[48] V Schomaker, J Waser, R. t. Marsh und G Bergman, “To fit a plane or a line to a set of points by least squares”, *Acta crystallographica*, Jg. 12, Nr. 8, S. 600–604, 1959.

[49] M. A. Fischler und R. C. Bolles, “Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography”, *Communications of the ACM*, Jg. 24, Nr. 6, S. 381–395, 1981.

[50] T. Rabbani, F. Van Den Heuvel und G. Vosselmann, “Segmentation of point clouds using smoothness constraint”, *International archives of photogrammetry, remote sensing and spatial information sciences*, Jg. 36, Nr. 5, S. 248–253, 2006.

[51] J. B. Roerdink und A. Meijster, “The watershed transform: Definitions, algorithms and parallelization strategies”, *Fundamenta informaticae*, Jg. 41, Nr. 1, 2, S. 187–228, 2000.

[52] N. Otsu, “A Threshold Selection Method from Gray-Level Histograms”, *IEEE Transactions on Systems, Man, and Cybernetics*, Jg. 9, Nr. 1, S. 62–66, 1979.

[53] OpenCV, *Image Segmentation with Distance Transform and Watershed Algorithm*, 24. Juni 2020. Adresse: https://docs.opencv.org/master/d2/dbd/tutorial_distance_transform.html (besucht am 10. 07. 2020).

[54] M. Kazhdan, M. Bolitho und H. Hoppe, "Poisson surface reconstruction", in *Proceedings of the fourth Eurographics symposium on Geometry processing*, Bd. 7, 2006.

[55] A. C. Ponce, *Elliptic PDEs, Measures and Capacities*. European Mathematical Society, 2016, isbn: 978-3-03719-140-8.

[56] F. P. Preparata und M. I. Shamos, *Computational geometry: An Introduction*. Berlin, Springer-Verlag, 1985, isbn: 978-0-387-96131-6.

[57] Z. C. Marton, R. B. Rusu und M. Beetz, "On Fast Surface Reconstruction Methods for Large and Noisy Datasets", in *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, Kobe, Japan, Mai 2009.

[58] F. Bernardini, J. Mittleman, H. Rushmeier, C. Silva und G. Taubin, "The ball-pivoting algorithm for surface reconstruction", *IEEE transactions on visualization and computer graphics*, Jg. 5, Nr. 4, S. 349–359, 1999.

[59] M. Kazhdan, M. Chuang, S. Rusinkiewicz und H. Hoppe, "Poisson Surface Reconstruction with Envelope Constraints", *Computer Graphics Forum*, 2020, issn: 1467-8659. doi: 10.1111/cgf.14077.

[60] C. Barnes, E. Shechtman, A. Finkelstein und D. B. Goldman, "PatchMatch: A randomized correspondence algorithm for structural image editing", in *ACM Transactions on Graphics (ToG)*, ACM, Bd. 28, 2009, S. 24.