



**hochschule mannheim**

# **Object Recognition for a Service Robot with Intel RealSense R200 3D Camera**

Oushesh Haradhun

Bachelor Thesis  
for the acquisition of the academic degree Bachelor of Science (B.Sc.)  
Course of Studies: Automation Technology

Department of Electrical Engineering  
University of Applied Sciences Mannheim

31.08.2016

Tutors  
Prof. Thomas Ihme, Hochschule Mannheim  
Dr. Christoph Hübner, Hochschule Mannheim

**Haradhun, Oushesh:**

Object Recognition for a Service Robot with Intel RealSense R200 3D Camera /

Oushesh Haradhun. –

Bachelor Thesis, Mannheim : University of Applied Sciences Mannheim, 2016. ??

pages.

**Haradhun, Oushesh:**

Bildverarbeitung zur Objekterkennung für einen Serviceroboter mit der Intel Re-

alSense R200 3D Kamera / Oushesh Haradhun. –

Bachelor Thesis, Mannheim : Hochschule Mannheim, 2016. ?? Seiten.

## **Erklärung**

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Mannheim, 31.08.2016

Oushesh Haradhun

# Abstract

## ***Bildverarbeitung zur Objekterkennung für einen Serviceroboter mit der Intel RealSense R200 3D Kamera***

Die Bedeutung der 3D Bildverarbeitung in der Robotik wird immer größer und die 3D Bildverarbeitung hat ein breites Anwendungsspektrum; von den Griff-in-die Kiste Projekte bis zum Serviceroboter. An der FH Mannheim im Institut für Robotik gibt es einige Serviceroboter wie der Pioneer 3DX und die Katana. Geplant ist den zukünftigen Einsatz der *Intel RealSense R200 Kamera mit diesen Roboter*.

Im Rahmen dieser Bachelorarbeit wird eine Herangehensweise für ein 3D Objekterkennungssystem mit der Verwendung von der *Intel RealSense R200 Kamera* vorgestellt um täglichen Objekte in übersichtliche Szene erkennen zu können. Die Kamera wird kalibriert und die gelieferte Punktwolke wird danach analysiert. Der Stand der Technik und Methoden wie Merkmale und Descriptors für einen Mustervergleich und Abstimmung zw. dem zu erkennenden Objekt und der Szene werden ebenfalls untersucht. Eine Software Architektur ist letztendlich vorgestellt um diese Herangehensweise zu implementieren und die Einsatzbarkeit von diesem 3D Sensor zu prüfen. Dies entspricht das Hauptziel dieser Arbeit und dabei wird keine Optimierung für einen Einsatz mit einem bestimmten Roboter durchgeführt.

## ***Object Recognition for a Service Robot with Intel RealSense R200 3D Camera***

The importance of 3D image processing in the field of robotics has expanded and 3D image processing has a large applicationspectrum ranging from bin picking projects till service robots.

We provide in this thesis an approach for a 3D recognition system to detect everyday objects in uncluttered environments under the usage of the *Intel RealSense R200 Camera* after having calibrated the camera, tested the quality and accuracy of the provided point clouds and eventually analysed several state-of-the-art methods, like features and descriptors to develop a method of matching models with the scene. A software architecture is also presented to implement this approach and then tested to see the applicability of this 3D Sensor for object recognition tasks. This corresponds to the main objective of

this thesis and in doing so not any sort of optimization for a ready usage with a given robot is conducted.

# Contents

<b>1 Einleitung</b>	<b>1</b>
1.1 Hintergrund & Motivation . . . . .	1
1.2 Arbeitsumfang . . . . .	2
1.3 Arbeitsübersicht . . . . .	2
<b>2 3D Sensoren</b>	<b>3</b>
2.1 Intel RealSense R200 . . . . .	4
2.1.1 Aufbau und Funktionsweise . . . . .	4
2.2 R200 'unboxing' & Erste Szene . . . . .	5
2.2.1 Auswertung der Szene . . . . .	5
<b>3 Analyse der Kalibrierung</b>	<b>7</b>
3.1 RGB Kalibrierung . . . . .	7
3.1.1 Kamera Einstellung . . . . .	7
3.1.2 Wahl des Kalibrierungsmusters . . . . .	8
3.1.3 Kamera Modell . . . . .	8
3.2 StereoKalibrierung . . . . .	9
<b>4 Durchführung der Kalibrierung</b>	<b>10</b>
4.1 RGB Kalibrierung . . . . .	10
4.2 Tiefe (Stereo)Kalibrierung . . . . .	14
4.3 Evaluation der Genauigkeit der Tiefe Abschätzung . . . . .	15
<b>5 Stand der Technik in 3D Bildverarbeitung</b>	<b>19</b>
5.1 Punktewolke . . . . .	19
5.2 Filterung . . . . .	19
5.2.1 Voxel Filter . . . . .	20
5.2.2 Passthrough Filter . . . . .	21
5.2.3 Statistical Outlier Removal Filter . . . . .	22
5.3 Segmentierung . . . . .	23
5.3.1 Euclidean Clustering . . . . .	23
5.4 Features and Descriptors . . . . .	24
5.4.1 Surface Normal Estimation . . . . .	27
5.4.2 Viewpoint Feature Histogram, VPFH . . . . .	27
5.4.3 Signature of Histograms of Orientations, SHOT [6] . . .	28

5.5	Registration . . . . .	30
5.5.1	Iterative Closest Point Algorithm . . . . .	30
5.5.2	Feature-Based Registration . . . . .	31
5.6	Analysis . . . . .	31
<b>6</b>	<b>Recognition Pipeline</b>	<b>32</b>
6.1	Mögliche Anwendung . . . . .	32
6.2	Datentyp . . . . .	33
6.3	Pipeline . . . . .	34
6.3.1	Clustering Subscriber & Publisher . . . . .	35
6.3.2	Matching . . . . .	36
6.3.3	Gesamte Pipeline . . . . .	37
<b>7</b>	<b>Validierung, Ergebnisse &amp; Optimierung</b>	<b>38</b>
7.1	Testing Setup . . . . .	38
7.2	Ergebnisse + Schlussfolgerungen . . . . .	39
7.2.1	Umgebungserkennung . . . . .	39
7.2.2	Objekterkennung . . . . .	40
7.3	Laufzeit . . . . .	42
7.4	Optimierung . . . . .	43
<b>8</b>	<b>Quellecode</b>	<b>44</b>
<b>Bibliography</b>		<b>62</b>

# **Chapter 1**

## **Einleitung**

In diesem Kapitel wird einen kurzen Zusammenfassung über den Hintergrund & die Motivation hinter dieser Arbeit. Der Arbeitsumfang und der Arbeitsübersicht wird ebenfalls erläutert.

### **1.1 Hintergrund & Motivation**

3D Bildverarbeitung hat in diesem Jahrzehnt mehr Wert gewonnen und wird immer bedeutender und wichtiger in der Robotik. 3D Bildverarbeitung ist noch attraktiver wenn wir einen Hinblick auf seine Anwendungsmöglichkeiten geben. Die 3D Bildverarbeitung hat ein sehr breites Anwendungsspektrum im Robotik; vom traditionellen 'Griff in die Kiste' Projekt bis zum Maschinelles Lernen anhand von Neuronalen Netzwerke wie *Convolutional Neuronal Networks*<sup>1</sup> & *Deep Learning*<sup>2</sup>.

Die Intel RealSense Technologie bietet die Möglichkeit die Tradition, die die Kinect Kamera seit 2010 hinterlassen hat zu erhalten. Es geht nämlich um den Einsatz von günstigen gewerblichen 3D Sensoren für wissenschaftliche Arbeiten da die Laser Scanner üblicherweise sehr teuer sind. In der Vergangenheit bewies der Kinect Sensor über sein Potenzial genügend rauschfreien Daten zu liefern um überhaupt in Objekterkennungsprojekten eingesetzt zu werden. Viel Projekte sind noch nicht mit dem Einsatz von 3D Sensoren der Intel RealSense Technologie veröffentlicht sodass die Möglichkeiten noch breiter sind. Die Gelegenheit die auf dem Markt neulich vorgestellte *Intel RealSense R200 Kamera* zu nutzen hat ebenfalls zu der Entscheidung beigetragen, meine Abschlussarbeit in diesem Bereich zu bearbeiten.

Das Ziel dieser Arbeit ist die Untersuchung der Einsatzbarkeit der *Intel RealSense R200 Kamera* für Objekt & Umgebungserkennung. Dabei wird zuerst die Kamera kalibriert und die vorhandene Kalibrierung getestet ob sie plausibel ist. Die Genauigkeit der gelieferten Tiefe Werte in der Punktwolke wird danach analysiert.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Convolutional\\_neural\\_network](https://en.wikipedia.org/wiki/Convolutional_neural_network)

<sup>2</sup><https://developer.nvidia.com/deep-learning-frameworks>

Wenn soweit alles in Ordnung ist werden unterschiedliche Methoden für die Segmentierung, Clustering und Features untersucht und anschließend eine Software Architektur für eine *Recognition Pipeline* vorgestellt, implementiert und getestet. Mögliche Erweiterungen werden im letzten Kapitel am Ende erwähnt.

## 1.2 Arbeitsumfang

Diese Arbeit basiert sich hauptsächlich auf dem *Open Source C++ Framework Point Cloud Library (PCL)*, deren Ursprung die Doktorarbeit von Rusu et al.<sup>3</sup> ist. Diese Arbeit liefert keine Konkurrenz gegen industrielle Bildverarbeitungssoftware wie *MVTech Halcon*<sup>4</sup> und *Cognex VisionPro*<sup>5</sup> sondern dient nur zur Erkennung von täglichen Objekten in übersichtliche Szene; sog. *uncluttered scenes* anhand einem grundsätzlichen Konzept.

Es wird in diesem Projekt *Open Source C++ Bibilotheke PCL-1.7 & ROS Indigo Ingloo* auf *Ubuntu 14.04. LTS* eingebunden. Diese Version ist stabil und wird insofern verwendet.

## 1.3 Arbeitsübersicht

Folgendermaßen sind kurze Überblicke von jedem Kapitel.

- In *Kapitel 2* werden die Taxonomie von 3D Sensoren gegeben und anhand den Komponenten der *Intel®RealSense-TM Camera R200* seine Funktionsweise erklärt.
- In *Kapitel 3 & Kapitel 4* geht es um die Kalibrierung von unserem Sensor und seine Tiefe Genauigkeit.
- *Kapitel 5* präsentiert den Stand der Technik in der 3D Bildverarbeitung wobei über Filterung, Segmentierung, Clustering, Features & Descriptors erläutert wird.
- *Kapitel 6* handelt sich um die Implementierung eines Erkennungssystem; sog. *Recognition Pipeline*.
- In *Kapitel 7* werden die Grenzen des eingesetzten Verfahrens, Ergebnisse & Schlussfolgerungen nach dem Testen und anschließend die Optimierungsmöglichkeiten veranschaulicht.

---

<sup>3</sup><http://pointclouds.org/>

<sup>4</sup><http://www.halcon.com/>

<sup>5</sup><http://www.cognex.com/products/machine-vision/visionpro-vision-software/?id=15291&langtype=1031>

## Chapter 2

# 3D Sensoren

Dieses Kapitel beschäftigt sich mit 3D Sensor und seine Taxonomie. Die Funktionsweise & der Aufbau der verwendeten R200 Kamera wird erläutert und anhand einer Szene die Grenze seiner Technologie erklärt.

Seitdem Markteinführung vom Kinect 3D Sensor im 2010 gibt es zurzeit zahlreiche 3D Sensoren. Die Taxonomie von Tiefe Erfassungsverfahren ist deutlich in Abbildung 2.1 dargestellt. [1]

Besonders in Hinblick auf diese Arbeit ist die Kategorie '*optical*', die weiterhin in aktive und passive Systeme unterteilt ist. Passive Tiefe erfassung verweist auf die in der Szene schon anwesenden Lichtstrahlen, z.B. das Stereo-Vision System. Andererseits verweist die Aktive Tiefe erfassung auf erhaltene Messwerte die nach der Projektion von externen Strahlen wie z.B. Infrarot entstehen. Die neuen 3D Sensoren kombinieren das *Aktiv-* und *Passiv* System um ein besseres Erfassungssystem zu erhalten. Es entsteht dadurch ein *Hybrid* System. Die *Intel RealSense R200 Kamera* in Abbildung 2.2 ist ein Beispiel vom *Hybrid Sensor*.

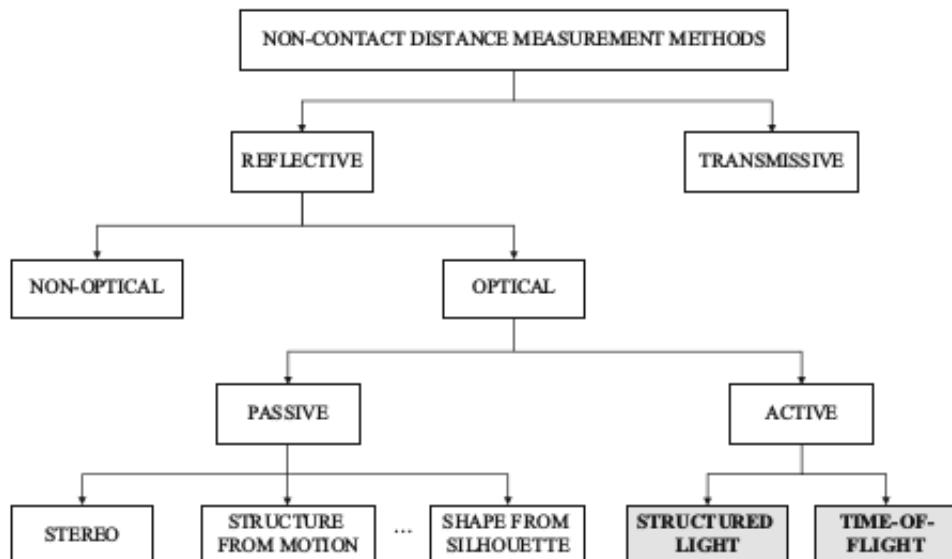


Figure 2.1: Distance Measuring Methods[1]

## 2.1 Intel RealSense R200

Einen Blick auf die Intel RealSense R200 Kamera wird geworfen und gleichzeitig seinen Aufbau und seine Funktionsweise wird erläutert. Letztendlich wird eine Beispielszene analysiert.

### 2.1.1 Aufbau und Funktionsweise



Figure 2.2: Intel Realsense R200[1]

Die RealSense Technologie vom Intel ist ein Exemplar von 3D Sensing Hardware module, in diesem Fall das Intel RealSense R200 Modell. Wie vorher erwähnt ist die R200 Kamera ein Beispiel der Kategorie Hybrid Sensoren, das erfolgreiche passive Infrarot Kamerasystem mit einem emittierten Infrarot Gitter kombiniert. Die R200 Hardware Module hat ein stereoskopisches Tiefe Kamerasystem, das 2 empfindliche IR global-shutter VGA CMOS Sensoren für die 3D Rekonstruktion enthält.

Die IR Sensoren sind von einander um 7 cm; sog. baseline distance entfernt. Ebenfalls gibt es einen 850 nm IR 'Illuminator' der eine Textur projiziert um die 3D Rekonstruktion zu unterstützen wenn Szene zu wenig intrinsische Details haben, z.B. bei großen Ebene. Dabei ist auch eine 2MP RGB Kamera vorhanden. Die R200 hat Echtzeit Fähigkeit Tiefe Daten auszurechnen dank seiner onboard ASIC ('DS4', Imaging Processor s. Abbildung 2.3), der präzise Tiefemap mit sehr gering Strom Verbrauch erzeugen kann. Dafür wird die USB 3.0 Schnittstelle benötigt. Mit einem USB 2.0 Port funktioniert die Kamera nicht da die Stromversorgung und die Datenrate vom USB 2.0 einfach nicht ausreichend sind. Einen USB 3.0 Port ist zwingend erforderlich.

In Abbildung 2.3 sieht man die wichtige Komponenten wie, 2 IR Kameras & der IR Projektor + die Farbkamera. Die Intel RealSense R200 benötigt keine 'Virtuelle Kamera'[1] wie die Kinect da ein Stereokamerasystem vorhanden ist und die 'Baseline' Entfernung, die die Auflösung der Tiefenkamera beeinflusst ist die 70 mm Entfernung zw. beiden IR Kameras.

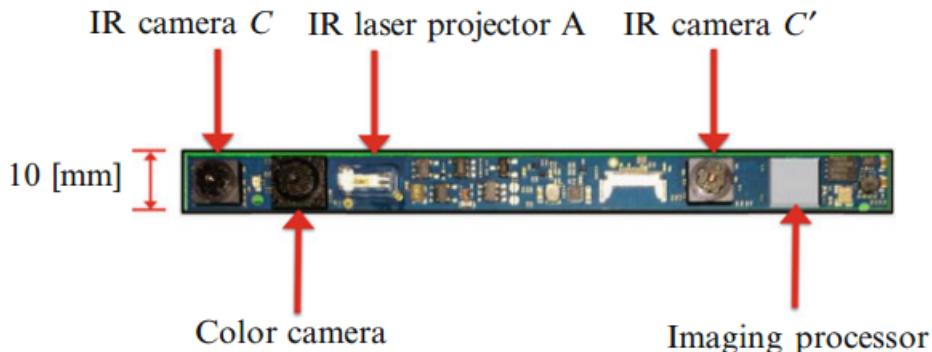


Figure 2.3: R200 Components[1]

## 2.2 R200 'unboxing' & Erste Szene

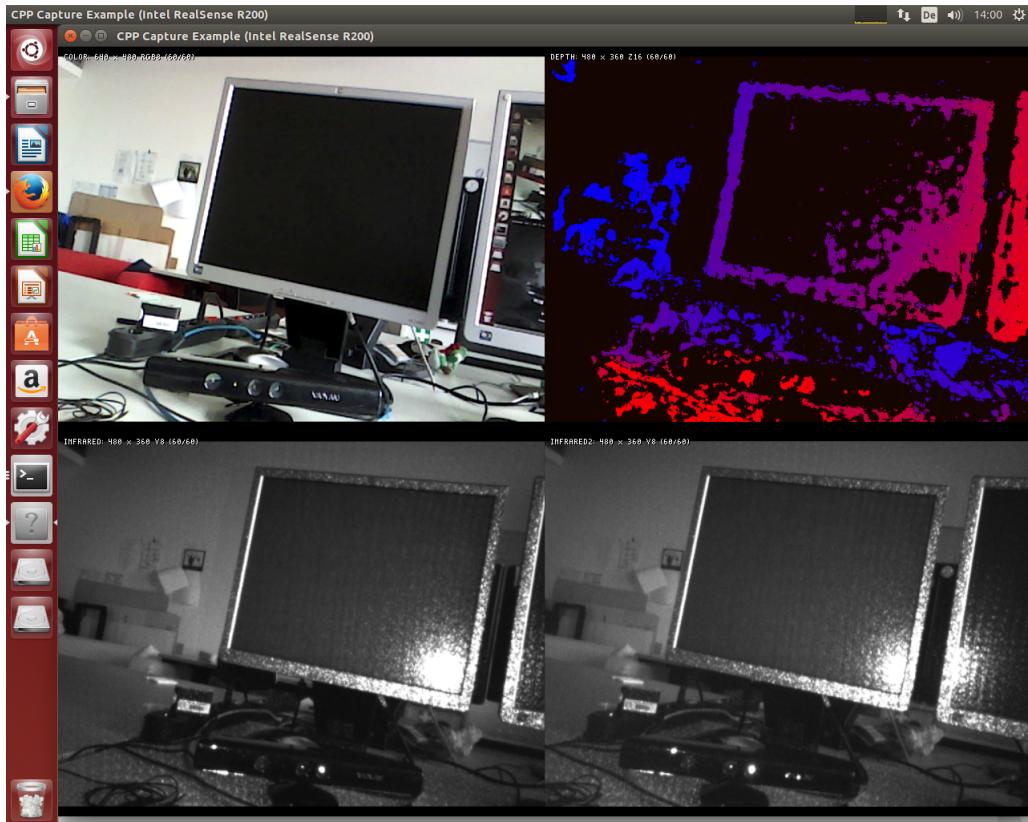


Figure 2.4: Szene

Mit der R200 Kamera kommunizieren zu können ist der Treiber bzw. die Bibliothek vom Intel, '*librealsense*'<sup>a</sup> erforderlich. Nach der Installation der Bibliothek '*librealsense*', die der ganze Triber vom R200 enthält, wird einige '*executables*' erzeugt. Der wichtigster davon ist der */cpp-capture*, der für den ersten Eindrück durchgeführt wird. Bespielsweise entsteht die Szene wie in Abbildung 2.3, die danach erläutert wird.

### 2.2.1 Auswertung der Szene

Die Abbildung 2.4 zeigt die Bilder, die die R200 Kamera liefert. Auf der linken obigen Ecke befindet sich das RGB Bild und daneben das Tiefe Bild. Darunter sind die 2 IR Bilden, erzeugt von der IR Kamera nämlich C und C<sup>1</sup>.(s. Abbildung 2.4)

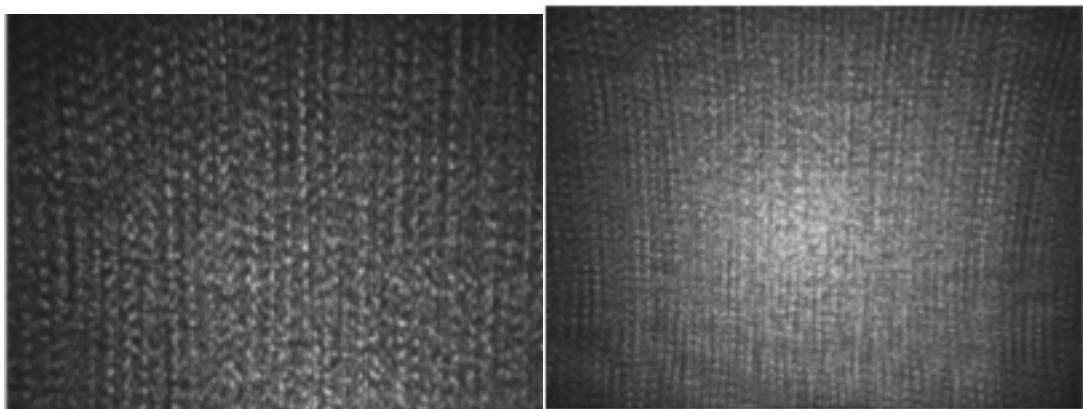
<sup>a</sup><https://github.com/IntelRealSense/librealsense>

Besonders interessant ist dabei das Projektierungsmuster des IR-Bildes. Dieses Muster sieht wie in Abbildung 2.5 aus.

- Zu erkennen ist, dass die R200 viel mehr grauwerte Stoffe fürs Muster projizieren kann im Vergleich zu dem Muster von der Kinect Kamera. Die Kinect Kamera liefert nur ein binäres Muster (sog. Collimated Dots [1]), das entweder schwarz oder weiß ist. Näher gesehen erkennt man auf der Abbildung 2.5 unterschiedliche grauwerte Stoffe, nicht nur weiß und schwarz. Die R200 benutzt deswegen eine 'Range Multiplexing' für die Verarbeitung dieser Stoffe.
- Die R200 kann schwarze Körper wie zB. die Kinect Kamera in der Abbildung 2.4 nicht ausreichend gut rekonstruieren weil die IR ausgesendete Strahlen vom Objekt absorbiert werden.
- In Hinblick auf die Hintere Objekte merkt man, dass sie ebenfalls im Vergleich zu den vorderen Objekten nicht exakt rekonstruiert werden können. Unterschiedliche Pixel haben aufgrund des 'Range Multiplexing' unterschiedliche Beleuchtungsstärke. Infolgedessen dunklere Anteile der Szene sind mit niedriger Beleuchtungsstärke gekennzeichnet. Angesichts der Tatsache dass die Beleuchtungsstärke des emittierten Projektierungsmusters,  $I$  sich mit der quadrat der radialen Entfernung zw. der Kamera und dem Objekt,  $d^2$  abnimmt

$$I \propto \frac{1}{d^2} \quad (2.1)$$

ist es wohl sehr schwierig die Entfernung in Hinweise auf die dunklere Anteile der Szene zu schätzen.



**Figure 2.5:** IR Pattern Zoomed & IR Pattern[1]

## **Chapter 3**

# **Analyse der Kalibrierung**

In diesem Kapitel erhalten wir einen Blick im Wesentlichen auf die wichtigen Aspekte der Kalibrierung. Hierbei wird erstens über die beste Einstellung der R200 Kamera und später über die Wahl des Kalibrierungsmusters diskutiert. Das Kamera-Modell, die Kamera Matrix und die interne sowie die externe parameter der Kamera anhand der Stereokalibrierung werden anschließend erläutert.

### **3.1 RGB Kalibrierung**

Hierbei geht es um die sog. monokulare Kalibrierung um die Interne Parameter zu kriegen. Die folgende Stichpunkte

- Kamera Einstellung
- Wahl des Kalibrierungsmusters &
- Kamera Modell

werden beachtet und jeder einzelner Teil erläutert.

#### **3.1.1 Kamera Einstellung**

Die R200 Kamera verfügt über zahlreiche Kombinationen von RGB Auflösung und Tiefe/IR-Auflösung, die in Wirklichkeit durch die maximale Bandbreite des USB 3.0 Ports begrenzt ist.

Tendenziell je höher die Auflösung desto genauer sollte der Algorithmus die Kreuzpunkte im Schachbrettmuster (s. Abbildung 4.2) finden. Die höchste verfügbare Auflösung von RGB Streaming ist  $1920 \times 1080$ . Mit dieser Auflösung reicht die Bandbreite für die gleichzeitige Tiefe Streaming allerdings nicht. Es muss jedoch einen Kompromiss zw. Auflösung und Bildwiederholfrequenz(fps) gefunden werden.

### 3.1.2 Wahl des Kalibrierungsmusters

Das ROS Kalibrierungswerkzeug unterstützt Kalibrierungsmuster ohne besonderer Merkmal, d.h. das Kalibrierungsmuster verfügt über kein Referenzmerkmal um die Orientierung zu bestimmen. Insofern können die  $x$  &  $y$  Achse entlang der Kreuzpunkte in beliebig Orientierungen zugeordnet werden. Grunddessen sollte das Schachbrett asymmetrisch sein, d.h. eine Seite sollte eine gerade Anzahl an Eckpunkten aufweisen die andere eine ungerade Anzahl. Bei einem asymmetrischen Schachbrett kann die Orientierung des Schachbretts stets exakt bestimmt werden. (s. Abbildung 4.2)

### 3.1.3 Kamera Modell

Mathematisch wird eine *3D* Kamera anhand vom Kamera-Modell gekennzeichnet. Dieses Model, dargestellt in Gleichung 3.1 beschreibt die Projektion von Koordinaten im Camera Coordinate System, CCS,  $sm^1$  zu den entsprechenden Koordinaten im World Coordinate System, WCS,  $M^1$ .

$$sm^1 = KRM^1 \quad (3.1)$$

In Gleichung 3.1 sind auch Matrizen wie  $K$ , *Kamera Matrix* &  $R$ , *Rectification Matrix* vorhanden.

Im allgemeinen lautet die Kamera Matrix,  $K$  
$$\begin{bmatrix} f_x & 0 & C_x \\ 0 & f_y & C_y \\ 0 & 0 & 1 \end{bmatrix}$$
 mit  $f_x$  &  $f_y$  die Brennweite und  $C_x$  &  $C_y$  die optische Schwerpunkte in *Pixel Koordinaten*, CCS. Die Rektifizierungsmatrix ist auch eine  $3x3$  Matrix und kümmert um die Nachbesserung der Verzerrung. Diese Verzerrung ist in Wirklichkeit wegen der Linse und Sensorchip der Kamera vorhanden. Diese Verzerrung ist ebenfalls mathematisch beschrieben und wird erläutert.

Im Normalfall treten 2 Arten von Verzeichnungen [3] in Verbindung miteinander auf. Die radiale Verzeichnung beschreibt eine tonnen- oder kissenförmige Bildverzerrung die durch die Beschaffenheit der Linsen in der Optik entsteht. Des Weiteren beschreibt die tangentiale Verzeichnung den durch Verkipfung zwischen Linsen und Sensorchip entstandenen Fehler. Beide Verzeichnungsarten können mithilfe des Brown-Conrady Models, sog. 'plumb-bob -Model'<sup>a</sup> beschrieben und softwaretechnisch ausgeglichen werden. Dafür bietet ROS schon ein 'Kalibrierungswerkzeug'<sup>b</sup>. Dieses Werkzeug basiert sich auf dem Kalibrierungsalgorithmus vom OpenCV<sup>c</sup>.

$$\text{Entzerr – Koeffizienten } (k1, k2, k3, p1, p2) \quad (3.2)$$

<sup>a</sup>[https://de.wikipedia.org/wiki/Entzerrung\\_\(Fotografie\)](https://de.wikipedia.org/wiki/Entzerrung_(Fotografie))

<sup>b</sup>[http://wiki.ros.org/camera\\_calibration/Tutorials/MonocularCalibration](http://wiki.ros.org/camera_calibration/Tutorials/MonocularCalibration)

<sup>c</sup>[http://docs.opencv.org/2.4/modules/calib3d/doc/camera\\_calibration\\_and\\_3d\\_reconstruction.html](http://docs.opencv.org/2.4/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html)

## 3.2 StereoKalibrierung

Die Tiefe Kalibrierung bezieht sich in unserem Fall auf die Stereokalibrierung da die R200 Kamera ein aktives Stereokamerasystem ist. Das Stereokamerasystem, die *Epipolare Geometrie*<sup>1</sup>, die *Intrinsische*<sup>2</sup> und *Extrinsische*<sup>3</sup> Parameter wurden schon in der Abschlussarbeit von *Hannes Raudies* [4] beschrieben und werden hier nur kurz erwähnt. Durch die Stereokalibrierung sind üblicherweise die *Intrinsische Parameter* von jeder einzelnen Kamera, die die Verzerrungen wie zuvor erwähnt beschreiben und ebenfalls die relative Position einer Kamera zur anderen im 3D Raum als eine '*rigid transformation matrix*', sog. *Extrinsische Parameter* geliefert, die für die 3D Rekonstruktion anhand der *Epipolare Geometrie* erforderlich sind. Hauptsächlich kennzeichnet einen Epipolare Fehler kleiner als 1 den Erfolg der Stereokalibrierung und wird häufig mit dem Pixelmaßstab gemessen. Weiterhin kann anhand der Kalibrierung die *Extrinsische Parameter* nachgemessen werden um zu ebenfalls zu prüfen ob die Kalibrierung erfolgreich war.

---

<sup>1</sup>[https://en.wikipedia.org/wiki/Epipolar\\_geometry](https://en.wikipedia.org/wiki/Epipolar_geometry)

<sup>2</sup>[https://en.wikipedia.org/wiki/Camera\\_resectioning#Intrinsic\\_parameters](https://en.wikipedia.org/wiki/Camera_resectioning#Intrinsic_parameters)

<sup>3</sup>[https://en.wikipedia.org/wiki/Camera\\_resectioning#Extrinsic\\_parameters](https://en.wikipedia.org/wiki/Camera_resectioning#Extrinsic_parameters)

## Chapter 4

# Durchführung der Kalibrierung

In diesem Kapitel erhalten wir einen Blick auf die Durchführung der Kalibrierung. Erstens wird eine RGB Kalibrierung verfolgt von einer Tiefe (Stereo) Kalibrierung durchgeführt. Anschließend wird die Genauigkeit der berechneten Tiefe anhand der Kamera evaluiert. Stichpunktformig entsteht unsere Herangehensweise wie folgend aus:

- RGB Kalibrierung
- Tiefe(Stereo) Kalibrierung
- Evaluation der Genauigkeit der Tiefe Abschätzung

### 4.1 RGB Kalibrierung

Laut der Untersuchung im Kapitel 3 wird es letztendlich als Kompromiss mit  $640 \times 480$  Auflösung das RGB Bild und mit  $480 \times 360$  Auflösung das Tiefe Bild, beide mit 30 fps gestreamt. Diese Einstellung lieferte genug scharfe RGB und IR Bilder mit einer ausreichenden Bildwiederholfrequenz. Die Einstellung sieht wie im Algorithmus 4.1 aus und wird im *launch file; realsense\_r200\_rgbd.launch* der Kamera eingetragen. Diese Einstellung wird im ganzen Lauf von diesem Projekt beibehalten.

---

#### Algorithmus 4.1 Resolution Settings

---

```
<arg name="mode" default="manual" />
<arg name="color_width" default="640" />
<arg name="color_height" default="480" />
<arg name="depth_width" default="480" />
<arg name="depth_height" default="360" />
<arg name="depth_fps" default="30" />
<arg name="color_fps" default="30" />
```

---



**Figure 4.1:** Schachbrettmuster

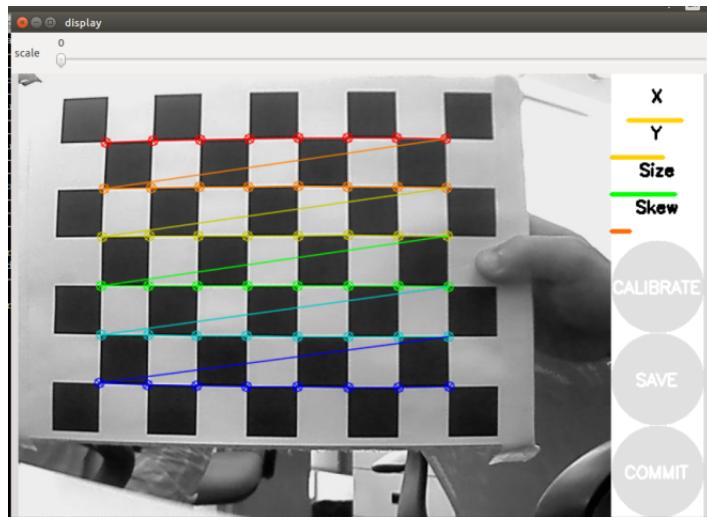
Auf ROS wird zunächst die Kalibrierungsnode wie im Algorithmus 4.2 aufgerufen. Es muss dabei die anzahl an Kreuzpunkte, hier **farbig** markiert (s. Abbildung 4.1) und die geometrische Form des Musters, hier '*Square*' und anschließend die größe der Quadrat (s. Algorithmus 4.2) eingegeben werden. Diese 'node' greift auf dem Topic des RGB Bildes zu; */camera/image\_rgb*, das die Kamera auf ROS veröffentlicht und führt die Kalibrierung durch. In unserem Experiment für die Kalibrierung wurden 100 Bilder zur Bestimmung der radialen ( $k_1, k_2, k_3$ ) und tangentialen ( $p_1, p_2$ ) Entzerr-Koeffizienten aus Formel 3.2 im *Kapitel 3* aufgenommen mit unterschiedlichen Drehungen und Verkippungen. Laut OpenCV ist mindestens 10 Bilder erforderlich. Theoretisch werden ohne Rücksicht auf die Verzerrung nur zwei Aufnahmen des Schachbretts benötigt um die Parameter zu bestimmen. In der Praxis wird eine wesentlich höhere Anzahl an Aufnahmen gemacht um eine robuste Kalibrierung zu erhalten.

---

#### **Algorithmus 4.2** Camera Calibrator

```
$ rosrun camera_calibration cameracalibrator.py --size 8x6 --square 0.108 image:=/camera/image_rgb camera:=/camera
```

---



**Figure 4.2:** ROS Calibration Window

Zusätzlich leuchten die 4 Parameter  $x$ ,  $y$ , *size* & *skew* im ROS Kalibrierungs Fenster grün (s. Abbildung 4.2) wenn genug Bilder vorhanden sind. Danach werden die Interneparameter als .txt Datei (s. Abbildung 4.3) ausgegeben. Diese .txt Datei wird im .yaml Datei konvertiert und falls die Kalibrierung erfolgreich gewesen ist mit einem 'click' auf dem 'commit' Taster (s. Abbildung 4.2) eingenommen werden. Auf der erzeugten Kalibrierungsdatei in der Abbildung 4.3 nach der Durchführung des Kalibrierungsvorgangs sind 4 Matrizen zu erkennen :

- Camera Matrix,  $K$
- Distortion Matrix,  $D$  (s. Gl. 3.2 )
- Rectification Matrix,  $R$
- Projection Matrix,  $P$

```
ost.txt x
# ost version 5.0 parameters

[image]
width
640

height
480

[narrow_stereo]

camera matrix
649.492477 0.000000 315.750022
0.000000 654.568684 219.752664
0.000000 0.000000 1.000000

distortion
0.201861 -0.730304 -0.003824 0.000393 0.000000

rectification
1.000000 0.000000 0.000000
0.000000 1.000000 0.000000
0.000000 0.000000 1.000000
| 

projection
652.569153 0.000000 315.120330 0.000000
0.000000 661.810669 218.277808 0.000000
0.000000 0.000000 1.000000 0.000000
```

**Figure 4.3:** Interneparameter

Aufgrund dass die Werte  $C_x$ , hier mit dem Betrag von 315.75002 &  $C_y$ , hier mit dem Betrag von 219.7526640 im K-Matrix nach der Kalibrierung naherungsweise ihre idealen Werte 320 & 240 entsprechen, ist dieses ein Zeichen dafür, dass die Kalibrierung gut geklappt hat.

## 4.2 Tiefe (Stereo)Kalibrierung

Die IR Bilder werden von der 2 IR Kamera aufgenommen und wie vorher bei der RGB Kalibrierung wird das OpenCV Kalibrierungswerkzeug im ROS aufgerufen. Bei der Aufnahme des IR Kamerabildes muss zusätzlich der Projektor R200 Kamera verdeckt werden. Das Projizierte Punktemuster erschwert die Erkennung des Schachbretts und verfälscht die Positions berechnung der Kreuzungspunkte. Das Infrarotlicht der Umgebung ist leider zu schwach für die IR Kamera. Aus diesem Grund muss das Schachbrett mit Infrarotlicht beleuchtet werden. Hierzu kann z.B. eine starke Halogenlampe genutzt werden.

Es ist gute Praxis, die Kamera auf gemessene Höhe zu erhalten und nur das Schachbrett auf einem flachen Ebene zw. jeder Aufnahme zu gleiten. Die *max. Höhe* muss genau bekannt sein und die *min. Höhe* bei der größten Verkipfung mit einem Neigungsmesser und durch die bekannte Breite des Schachbretts ermittelt werden.

Die erhaltene Parameter wurden analysiert und werden stichpunkt förmig veranschaulicht:

1. Der Epipolare Fehler war  $0.18\text{pixel} < 1\text{pixel}$ , was unbedingt erforderlich ist für die 3D Rekonstruktion. Ist diese Bedingung nicht erfüllt wird die 3D Rekonstruktion zu schwach und unbrauchbar.
2. Die Werte von  $C_x$  &  $C_y$  betragen Beziehungsweise 234.8 & 178.8, was näherungsweise ihre idealen Werte 240 & 180 entsprechen.
3. Laut dem Treiber *librealsense in die Abbildung 4.4* sollte nur eine translatorische Abweichung in der  $x$ -Achse und keine Rotation zw. dem Linken und Rechten IR Bild entstehen, was anhand der Ergebnisse gekennzeichnet sind. Der erhaltene translatorische Vektor,  $\vec{T}[t_x, t_y, t_z]$  mit m als SI Einheit lautet  $[-0.05897, -0.000053558, -0.0001496]$  und die Rotationsmatrix,  $R$  ist eine

$$3 \times 3 \text{ Identitätsmatrix} \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}. t_y \text{ und } t_z \text{ sind gegenüber } t_x \text{ vernachlässigbar.}$$

Die Tatsache dass eine *Identitätsmatrix* heraus kamme deutet auf keine Rotation zw. den beiden IR Bilden.

Schließlich deuten die obige Hinweise auf eine erfolgreiche Stereokalibrierung.

```
#### R200
1. Left and right infrared images are rectified
* The two infrared streams have identical intrinsics
* The two infrared streams have no distortion
* There is no rotation between left and right infrared images (identity matrix)
* There is translation on only one axis between left and right infrared images ('translation[1]' and 'translation[2]' are zero)
* Therefore, the 'y' component of pixel coordinates can be used interchangeably between these two streams
```

Figure 4.4: Relative Position[2]



Figure 4.5: measurement\_test

### 4.3 Evaluation der Genauigkeit der Tiefe Abschätzung

Die Evaluation der Genauigkeit der Tiefe ist am bedeutsamsten in der 3D Bildverarbeitung weswegen die 3D Rekonstruktion & 3D Position eines Körpers davon abhängig ist.

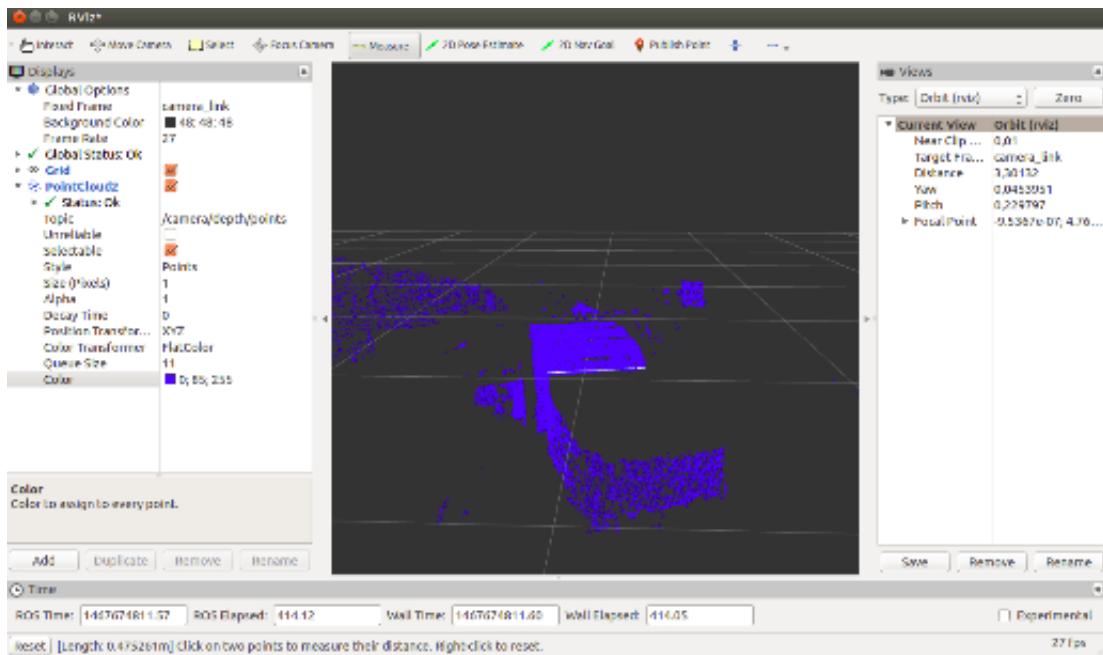
Deswegen wird einen kleinen Test durchgeführt wobei die Genauigkeit des 3D Sensors evaluiert wird. Die Abbildung 4.5 zeigt einen Schreibtisch. Die Größe vom Schubkasten, hier mit **rotem Pfeil** dargestellt beträgt  $47,5\text{cm}$  nach Bemessung mit einem Lineal. Aus der Punktwolke lautet die ermessene Größe  $0,475261\text{m}$ , die zu einem Fehler vom  $0,000261\text{m}$  führt. Die ganze Vorgehensweise vom Test ist wie folgend erläutert:

1. Die Kamera wird mit 'roslaunch'<sup>a</sup> gestartet & veröffentlicht die Punktwolke als Topic */camera/depth\_registered/points*
2. Das Visualisierungsplugin auf *ROS*, *rviz*<sup>b</sup> wird gestartet & greift aufs Topic zu.
3. Mit dem 'Tool' '**Measure**' wird die durch rotes Pfeil gezeichnete Entfernung gemessen. In Abbildung 4.6 sind die Punkt Wolke **dunkles blau** eingefärbt und die gemessene Entfernung ist im rviz Visualisierungsfenster mit einer weißen gerade veranschaulicht.

---

<sup>a</sup><http://wiki.ros.org/roslaunch>

<sup>b</sup><http://wiki.ros.org/rviz>



**Figure 4.6:** measurement\_tool

Nach diesem kleinen Test muss nichtsdestotrotz die Tiefe Genauigkeit auf verschiedene  $z$ -Entfernung evaluiert werden. Die R200 hat einen Arbeitsbereich zw. 0,5m und 4m. Die Kamera hat eine Tiefe Auflösung von 16 bit, d.h. es gibt in dem Arbeitsbereich von R200 max. nur  $2^{16}$  mögliche dargestellte Tiefe Werte. [2] Die Genauigkeit beträgt infogedessen  $\frac{4-0,5}{2^{16}}$ , unter 1mm. Stell dir vor, du hast  $2^{16}$  parallele Ebene vor der R200 Kamera in die  $z$ -Richtung. Jeder Tiefe-Punkt darf nur auf diese Ebene existieren. Infolgedessen werden alle andere Punkte die in zwischen liegen einfach auf dem nächsten Ebene approximiert. Es entsteht dadurch einen Quatisierungsfehler von  $\frac{4-0,5}{2^{16}} m$ , der aber unter 1mm ist, d.h. unbemerkbar gegenüber der Größe von täglichen Objekte die wir matschen wollen.

Desweiteren wird eine Kiste mit einer gemessene Breite,  $l_{real}$  von 0,227m benutzt. Das Objekt wird senkrecht auf die gleiche Höhe zur R200 Kamera ab 0,6m mit einem Intervall von 0,2m gelegt und weiterhin bis auf 2,0m bewegt. Auf jede Position wird die Breite des Objekts,  $l_{measured}$  mit dem 'Measure Tool' vom [rviz](#) wie vorher dreimal geschätzt und einen Mittelwert,  $\overline{l}_{measured}$  gebildet. Die Ergebnisse sind in Tabelle 4.7a dargestellt.

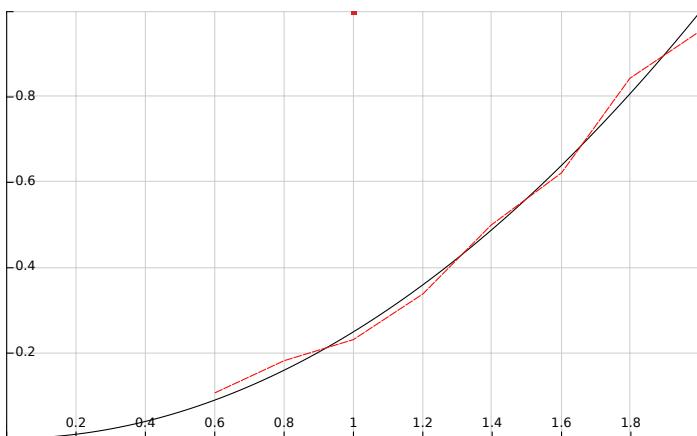
Zum Verstehen sind die folgende Abkürzungen erklärt:

- Radiale Entfernung zw. Objekt & Kamera,  $d/m$
- Absolute Differenz,  $|\Delta_l|$

Die Abbildung 4.7b zeigt danach anhand der Werte aus der Tabelle den Verlauf vom 'Normalised Error' gegen  $d/m$ . Dieser Verlauf darstellt eine quadratische Funktion. Die **rote** Kennlinie darstellt die Gerade, die jede Messwerte mit der nächsten verbindet und die **schwarze** die approximierte geglättete Kurve. Auf der  $x$ -Achse ist  $d/m$  und auf der  $y$ -Achse den 'Normalised Error'.

$d/m$	$\bar{l}_{measured}/m$	$ \Delta_l /m$	Normalised Error
0, 6	0, 22791	$5, 120 \times 10^{-4}$	0, 10
0, 8	0, 227962	$9, 216 \times 10^{-4}$	0, 18
1, 0	0, 227205	$1, 229 \times 10^{-3}$	0, 24
1, 2	0, 226874	$1, 792 \times 10^{-3}$	0, 35
1, 4	0, 227061	$2, 560 \times 10^{-3}$	0, 50
1, 6	0, 228119	$3, 226 \times 10^{-3}$	0, 63
1, 8	0, 231250	$4, 250 \times 10^{-3}$	0, 83
2, 0	0, 232120	$5, 120 \times 10^{-3}$	0, 98

(a) Measurement Values



(b) Measurement Value Graph

### Experiment von Intel®

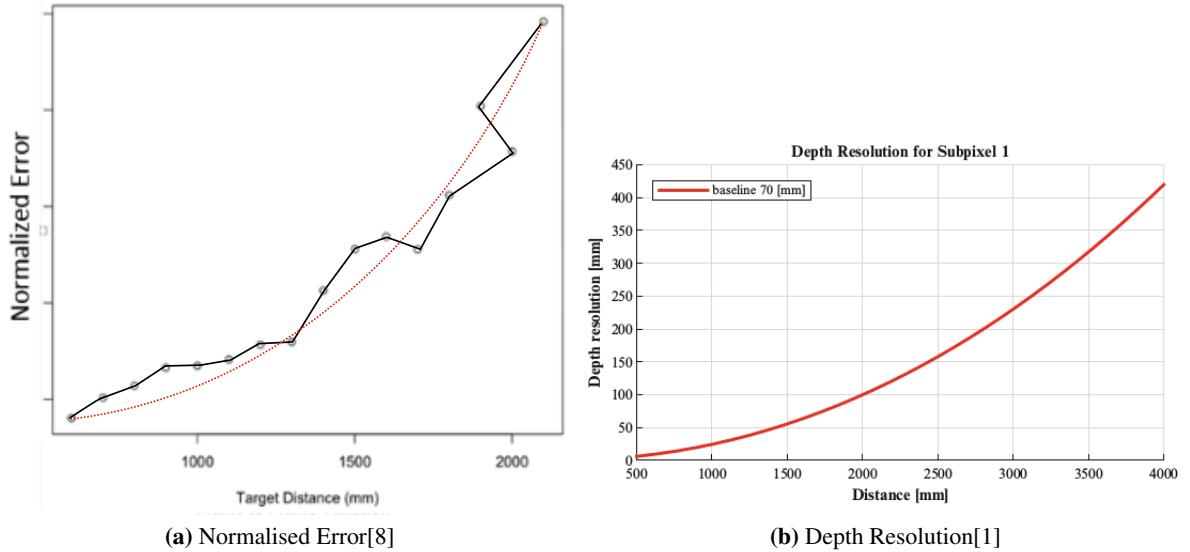


Figure 4.7: Experiment Setup & Analysis[8]

Eine genauere Evaluation der Tiefe Abschätzung auf einem industriellen Standard ist eher schwierig. Um aus diesen Verlauf aussagekräftige Schlussfolgerungen ziehen zu können wird unserer Test mit dem von Intel® [8] durchgeföhrten Experiment verglichen und die Ergebnisse (s. Abbildung 4.8) werden *analysiert*.

Die Experimentannordnung von Intel® ist in Abbildung 4.7 dargestellt. Im Rahmen dieses Tests wird die Beleuchtung kontrolliert. Die Bewegung der Kamera ist sehr präzise mit einem Schrittmotor und Encoder durchgeführt werden sodass nur die Entfernung in  $z$ -Achse sich ändern sollte ohne Veränderung in der  $x$ -&  $y$ -Achse.

Anschließend ist auch ein ToF Geräte z.B. hoch kalibriertes laser Messgeräte (s. *rote* Geräte in Abbildung 4.7) vorhanden.



**Figure 4.8:** Norminalised Error[8]

Auf der linken Seite befindet sich die Abbildung, in der die 'Normalised Error' gegen die Radiale Entfernung dargestellt ist. Diesen Verlauf (*rote Kurve*) wird gegen die zweite Abbildung; '*Depth Resolution*' gegen die '*Distance*' analysiert und eine Erklärung für den Verlauf in Abbildung 4.8a vorgegeben. Die '*Depth Resolution*' und die '*Distance*'; Entfernung zw. Objekt und Kamera sind wie folgend miteinander bezogen:

$$\Delta_z = \left( \frac{z^2}{b \cdot f} \right) \Delta_d \quad (4.1)$$

$\Delta_z$ , *Depth Resolution*,

$z$ , Tiefe des gemessenen Objekts; '*Distance*'

$b$ , Baseline Distance von 70 mm zw. den IR Kamera.

$f$ , Brennweite

$\Delta_d$ , Disparity Resolution; in diesem Fall 1

**Die** Norminalized Error nimmt mit steigender Entfernung zu.

Dieses ist zu erwarten weil die Depth Resolution einen ähnlichen Verlauf ausprägt. (s. Abbildung 4.8b). Je Weiter Weg befindet sich das Objekt von der Kamera desto größer ist die Projektionsebene. Die 'Depth Resolution' weist den min. gemessenen Tief-Unterschied auf und nimmt mit quadrierten Enternung,  $z$  zu. d.h. Je weiter Weg wir die Tiefe bestimmen desto größer ist die gekennzeichntete Entfernung 2 benachbarten Pixeln. Es wird dabei bemerkt dass die Kanten vom Objekt in Punktwolke häufiger schwanken. Diese Schwankung zeichnet den Zufallsfehler aus, der zunimmt und bezieht sich mit  $z$  folgedermaßen:

$$\sigma_z = \left( \frac{m}{f \cdot b} \right) z^2 \sigma_{d^1} \quad (4.2)$$

## Chapter 5

# Stand der Technik in 3D Bildverarbeitung

In diesem Kapitel geht es um den Stand-der-Technik in der 3D Bildverarbeitung um aus der Punktwolke '*Features*' extrahieren und anschließend Objekte erkennen zu können.

### 5.1 Punktwolke

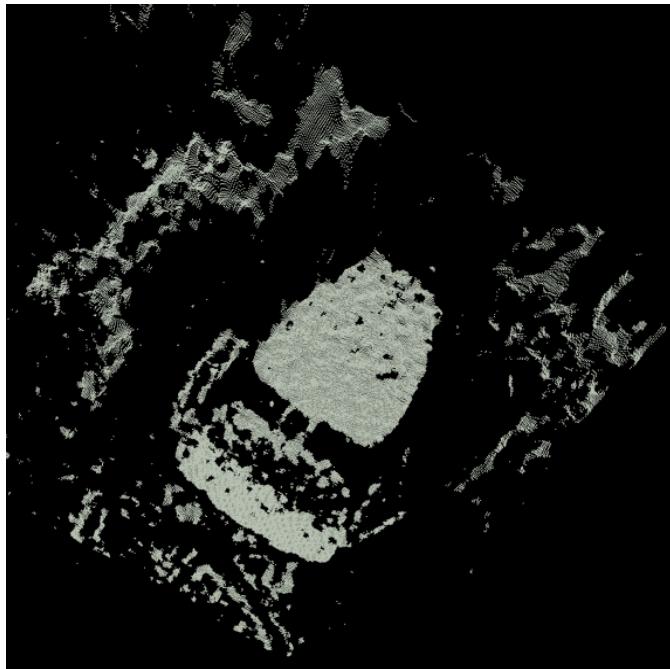
Die Abbildung 5.1 zeigt die Punktwolke der Szene, in der, einen Stuhl im Raum dargestellt ist. Die Punktwolke sind anhand einem 3D Koordinatensystem beschrieben. Jeder Punkt enthält Informationen im Vektorform [  $x, y, z, \vec{n}$  ];  $x, y$  &  $z$ - Koordinaten und einen normalen Vektor,  $\vec{n}$ . Diese sind die sog. *Attributes*. Prinzipiell sind die Attribute wie zuvor erwähnt die wichtigste, was fast alle Datentypen liefern. In einigen Datentypen wie `pcl::PointXYZRGB` werden zusätzlichlicherweise auch die Farbe codiert und für jeder Punkt mitgeliefert sodass mehr Information für die Bearbeitung zur Verfügung sind.

### 5.2 Filterung

Filterung ist vor allem der erste Schritt in Datenverarbeitung im Allgemeinen und ebenfalls anwesend in der 3D Bildverarbeitung.

Das Ziel dabei ist:

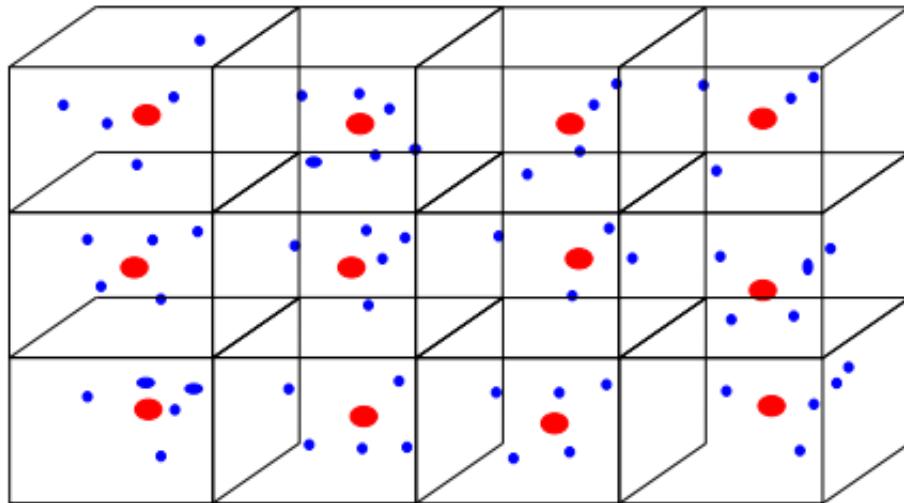
1. die Anzahl der Punkte zu reduzieren währenddessen die 'Keypoints' für die spätere Verarbeitung drin zu lassen.
2. die undefinierte Punkte, Punkte mit sog. NaN Werte raus zu filtern. Bei der R200 kommen eine Menge von NaN Punkte vor da der IR Projektor in zufällige Richtung IR Licht ausstrahlen und anhand der 'Range Multiplexing' die Grauwerte Stufje des projizierten Musters auszuwerten. Wie zuvor im Kapitel 2 erwähnt, häufig in Szene wo größere Objekte die Hintere Objekte verhindern erreichen sehr schwache bis kaum Strahlen wegen Streuungen und Reflexionen den IR Empfänger. An dieser Stelle erhalten die Punktwolke NaN Werte, die die spätere Berechnungen wie z.B. '*Normal Estimation*' erschweren kann.
3. die Rechenzeit zu reduzieren.



**Figure 5.1:** PointCloud

### 5.2.1 Voxel Filter

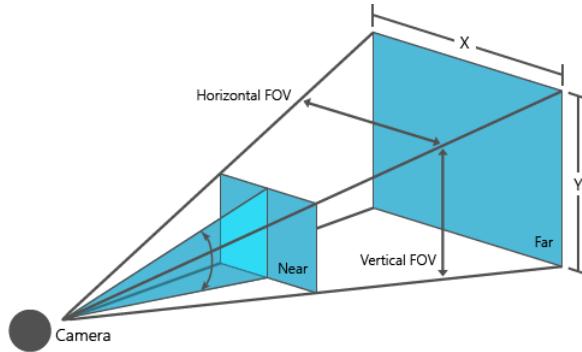
Der bekannteste Filter in der *PCL Bibliothek* ist der Voxel Filter [2]. Beim Anwenden wird den Raum im kleinen Quader unterteilt, die sog. *Voxel* deren Größe durch die eingegebene Parameter '*Leaf Size*' bestimmen werden. In jedem Voxel wird danach nur der Schwerpunkt, '*Centroid*' der gesamten Punkte, die tatsächlichen Punkte ersetzen. Der Vorteil dieses Filters ist die gleichzeitige Filterung von NaN Punkten. Der Voxel Filter prüft nämlich jeder Punkt und nimmt danach alle Punkte, die keine NaN Werte erhalten und daraus werden die Schwerpunkte berechnet. Letztendlich entsteht eine Punktwolke nur mit gültigen Punkten.



**Figure 5.2:** Voxel Grid[10]

### 5.2.2 Passthrough Filter

Der normale Arbeitsbereich der R200 liegt zw.  $0,5m$  &  $4m$ . Allerdings an beiden Grenzen entstehen in die Punktwolke viel Rauschen. Die R200 verwendet einen Disparitätsraum und hat einen maximalen Suchebereich von  $63\text{ pixels}$  in Horizontal Richtung und nichtsdestotrotz beim Verlauf von einem Software-Stereomatching auf dem Stereobild kann keine Disparität größer als die gesamte Breite des Stereobildes erhalten werden.[2] Daraus erfolgt, dass die minimale radiale Tiefe-Entfernung bei einer nominalen Tiefe-Auflösung von  $640 \times 480$  ca.  $72\text{cm}$  ( $\frac{640 \times 70\text{mm}}{63}$ ) ist. Andererfalls bei  $480 \times 360$  verkürzt sich diese Entfernung auf ca.  $50\text{cm}$ . Wie zuvor erwähnt wird im Lauf von diesem Projekt mit  $480 \times 360$  IR Bild gestreamt. In Bezug auf die beiden Kriterien wird daher den '**Passthrough Filter**' angewendet um das Intervall in  $z$ -Richtung ab  $0,7m$  bis auf  $3,5m$  zu reduzieren.



**Figure 5.3:** Field of View[9]

### 5.2.3 Statistical Outlier Removal Filter

Punktwolke generiert von 3D Sensoren haben keine homogene Punktdichte. In einer Szene werden die vordere Objekte mit einer dichten Punktwolke dargestellt als die hintere Objekte. Es liegt daran, dass je weiter Weg ein Objekt von der Kamera sich befindet desto größer wird die Projektierungsebene auf diese Entfernung sein. (s. Abbildung 5.3) Die Kamera streamt allerdings mit konstanter Auflösung sodass die gleiche Anzahl von Pixeln auf diese größere Ebene eine niedrigere Punktdichte verursacht. Daher werden die ferne Objekte mit viel geringer Punkt dargestellt vergleichsweise zu nahe Objekte. Infolgedessen erscheinen die ferne Objekte die sehr ungenau rekonstruiert werden als Rauschen. Diese müssen entfernt werden.

Es wird deswegen eine statistische Analyse über die Umgebung von jedem Punkt durchgeführt und die vorgegebene Bedingung nicht erfüllten Punkte werden rausgefiltert. Die '*Outlier Removal Filter*' unterscheiden sich voneinander je nach vorgegebene Bedingung. Ein Beispiel von '*Outlier Removal Filter*' ist der '*Sparse Outlier Removal Filter*'.

Der Sparse Outlier Removal Filter basiert sich auf die Berechnung der Gaußsche Verteilung mit dem mittelwert,  $\mu$  und der Varianz,  $\sigma^2$  von einem Punkt zu seinen n-Benachbarten Punkten. Das Prinzip von Sparse Outlier Removal Filter ist durch die folgende algorithmische Schritte erklärt:

1. Für jeder Punkt  $P_i$ , wird der mittelwert der Entfernung  $\bar{d}_i$  von alle Nachbarn gerechnet.
2. Es wird angenommen dass  $\bar{d}$  durch die Gaussche Verteilung  $f(x|\mu, \sigma^2)$  modelliert ist.
  - Wenn die Entfernung  $|\bar{d}_i - \mu| > \sigma$  ist, wird  $P_i$  raus gefiltert.
  - Andererfalls wird der Punkt erhalten.

## 5.3 Segmentierung

### 5.3.1 Euclidean Clustering

Nach der Filterung sind die wertvolle Punktewolke geblieben. Allerdings sind die Punkte unorganisiert. Das 'Clustering' dient demnächst zur Klassifizierung des Modells einer Punktewolke  $P$  in kleinere Anteile damit die gesamte Bearbeitungszeit drastisch reduziert werden kann. Jedes 'Cluster' enthält somit ein potenzielles Objekt, das für die Erkennung wichtig ist.

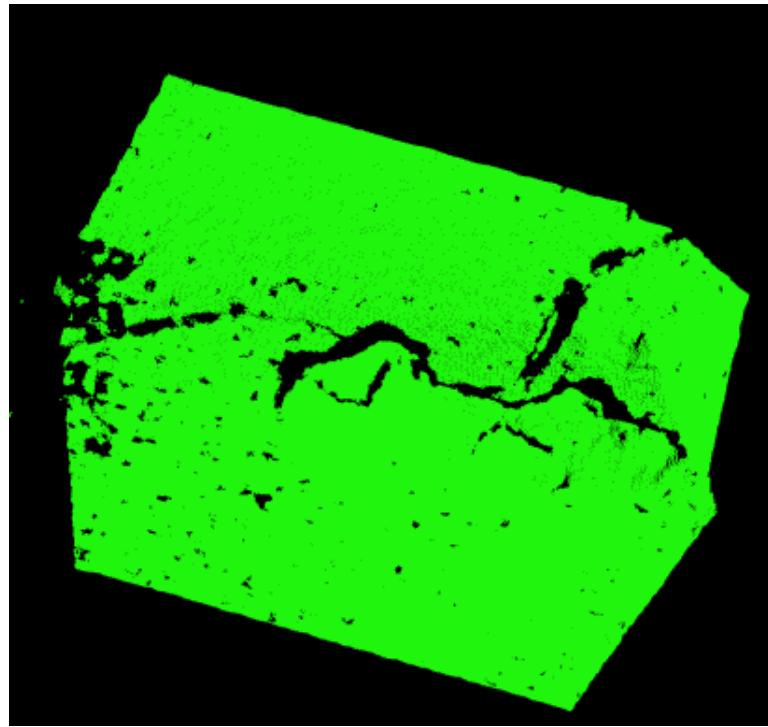
Das Prinzip vom *Euclidean Clustering* wird wie folgend erklärt:

1. Erstelle eine Kd-Tree<sup>a</sup> Darstellung für die Punktewolke  $P$ .
2. Stell leere Liste von Clusters  $C$ , und eine Reihe von Punkten  $Q$  die geprüft werden müssen.
3. Für jeder Punkt  $p_i \in P$ , wird folgendermaßen durchgeführt:
  - a) Füge  $p_i$  zur derzeitigen Reihe  $Q$
  - b) Für jeden Punkt  $p_i \in Q$ , wird folgendermaßen durchgeführt:
    - i. Suche für die Gruppe von Punkten  $P_i^k$  die Benachbarten Punkte von  $p_i$  innerhalb eines Kugels mit radius  $r < d_{th}$ ;
    - ii. Für jeden Nachbar  $p_i^k \in P_i^k$ , prüfe ob der Punkt schon bearbeitet wurde, wenn nein füge ihn in  $Q$  ein;
  - c) Wenn alle Punkte in  $Q$  bearbeitet wurden, füge  $Q$  zu den Liste von Clusters  $C$  ein, dann setze  $Q$  zur leeren Gruppe zurück.
4. Der Algorithmus wird beendet wenn alle Punkte  $p_i \in P$  schon bearbeitet wurden und sind ein Teil von der Liste von Clusters  $C$ .

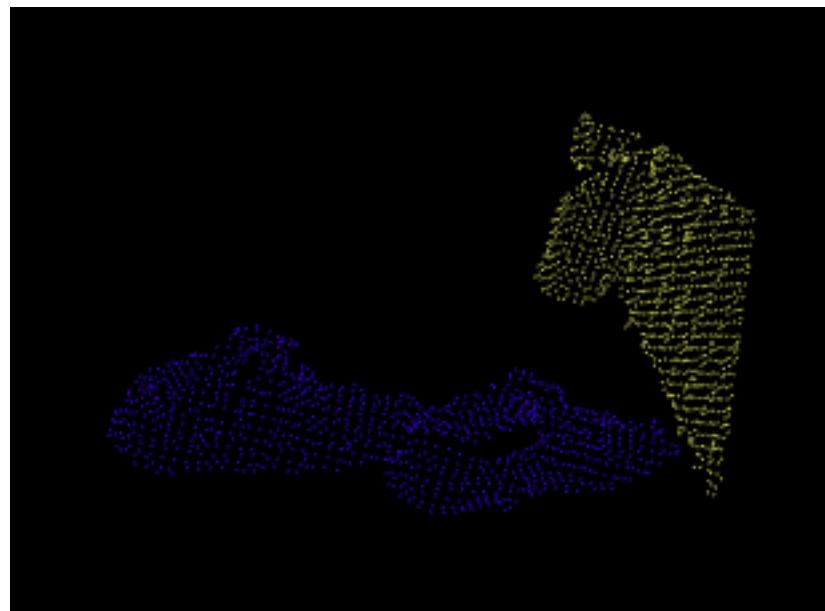
Die Abbildung 5.4 zeigt ein Beispielszene mit einem Schuhepaar auf den Fußböden, die von der R200 Kamera erhalten wurde. Hierbei sind die Punkte noch nicht klassifiziert. Das *Euclidean Clustering* wird angewendet und die erhaltene *Clusters* sind in Abbildung 5.5 dargestellt. Da sind 2 Clusters angezeigt; ein mit *Blau* gezeichnet, das andere mit *leicht Gelb*. Das blaue Cluster ist das Schuhepaar, das zu erkennende Objekt und das andere die Fassade der neben stehende Schublade. Insofern sind jetzt die Ursprungliche Punktewolke in 2 Clusters unterteilt. Die Punkte sind gut angeordnet und erleichtet die spätere weiterverarbeitung wie '*Feature Extraction*'.

---

<sup>a</sup>[http://pointclouds.org/documentation/tutorials/kdtree\\_search.php](http://pointclouds.org/documentation/tutorials/kdtree_search.php)



**Figure 5.4:** Scene Euclidean Clustering



**Figure 5.5:** Scene Clusters

## 5.4 Features and Descriptors

In der 3D Welt Darstellung werden nur das Kartesische Koordinatensystem verwendet.

In einer Szene enthält jeder Punkt authentische Informationen in einem Vektor Form  $[x, y, z]$ , die die 3D Kamera liefern kann. Ein Objekt beschrieben nur anhand von 3D Koordinaten in einer Szene erhält seine Unverwechselbarkeit lediglich in dieser Szene. Allerdings fördert die 3D Bildverarbeitung dass ein Objekt unter gewisse rauschen in unterschiedliche Szene unabhängig von Orientation erkannt werden muss. Deswegen sind allein die Attribute  $x$ ,  $y$  &  $z$  nicht ausreichend. Dafür sind zusätzliche Informationen erforderlich. Diese extra Attribute werden dank '**Features**' zur Verfügung gestellt. Der wichtigste '**Feature**' ist die erste differentielle Ordnung der 3D Vektoren; die Normalen-Vektoren. Zahlreiche 3D Sensoren stellen extra Informationen für jeden Punkt zur Verfügung wie z.B. Intensität, Farbe usw... Allerdings sind diese Informationen nicht auskömmlich. Grunddessen ist ein neues Konzept erforderlich. Der Konzept von einzelnen Punkten als eine Einheit verschwindet und ein neuer Konzept von '*Local Descriptor*' findet statt. Die Descriptors sind zuständig für die Kodierung der Verkrümmung eines Objektes in der Umgebung von *Keypoints* im 3D Raum. Die Art und Weise dieser Kodierung unterscheiden die Descriptors voneinander. Descriptors werden dank deren unterschiedlichen Eigenschaften klassifiziert. Die taxonomische Einordnung von 3D Descriptors ist anhand von Eigenschaften wie Kategorie, Unique LRF(Local Reference Frame) und Farbe durchgeführt.

## Taxonomie

1. Unique LRF(Local Reference Frame)
  - a) Unique LRF wird berechnet; Local Descriptor
  - b) Unique LRF wird nicht berechnet; Global Descriptor
2. Kategorie
  - a) Signature
  - b) Histogram

In Prinzip sind *Local Descriptors* bevorzugt gegenüber *Global Descriptors*. Die Verwendung von Local Descriptors erlaubt die Verwendung einer Datenbank mit vollen Objekten anstatt von partiellen Momentaufnahme; *partial snapshots*, die für die 6 Dof Pose Abschätzung im 3D Raum zusätzliche Vorgehensweise wie *Registration*<sup>a</sup> erfordert wobei das Objekt von unterschiedlichen Blickwinkeln aufgenommen wird um später die Punktwolke einzelner Momentaufnahme in einem kompletten 3D Modell zusammenzuheften. Resampling wird hierbei erforderlich da die sich schneidenden Anteile der Punktwolke eine höhere Punktdichte aufweisen. Die Normalen-Vektoren müssen ebenfalls wieder orientiert werden so dass sie nach außen aufzeigen. Diese zusätzliche Verarbeitungen sind dagegen mit dem Einsatz vom vollen Modell erspart und ist das Matching in Prinzip vom niedrigen Aufwand. Die ausgeprägte Eigenschaft in der Taxonomie von 3D Descriptors ist nichtsdestotrotz die Kategorie, vorher erwähnt.

---

<sup>a</sup>[http://pointclouds.org/documentation/tutorials/registration\\_api.php](http://pointclouds.org/documentation/tutorials/registration_api.php)

Daher wird zunächst einmal eine anschauliche Erläuterung durchgeführt um die Kategorie genauer zu verstehen.

## 1. Signature

Signature beschreibt die 3D geometrische Struktur in der Umgebung eines *Keypoints*. Hierbei wird ein lokales Koordinatenbezugssystem, sog. *Local RF* definiert in dem die Geometrische Messungen auf jedem lokalen Punkt berechnet wird. Dagegen modifizieren kleine Fehler in der Definition vom lokalen RF oder kleine Schwankungen im Kodierten Merkmal den resultierenden 'Descriptor'.

- a) Vorteil: Sehr deskriptive
- b) Nachteil: Nicht robust gegen Störungen

## 2. Histogram

Andererseits beschreibt das Histogramm die Struktur in der Umgebung eines *Keypoints* durch die Häufigkeit (*count*, s. Abbildung 5.6) von topologischen Einheiten wie Vertices, Mesh Triangle Areas usw. in einem quantisierten Domain.

- a) Vorteil: Robust dank Kompression der Information in 'bins'
- b) Nachteil: Nicht so deskriptive wie Signature

Der Unterschied im Konzept zw. Signature und Histogram ist in Abbildung 5.6 illustriert.

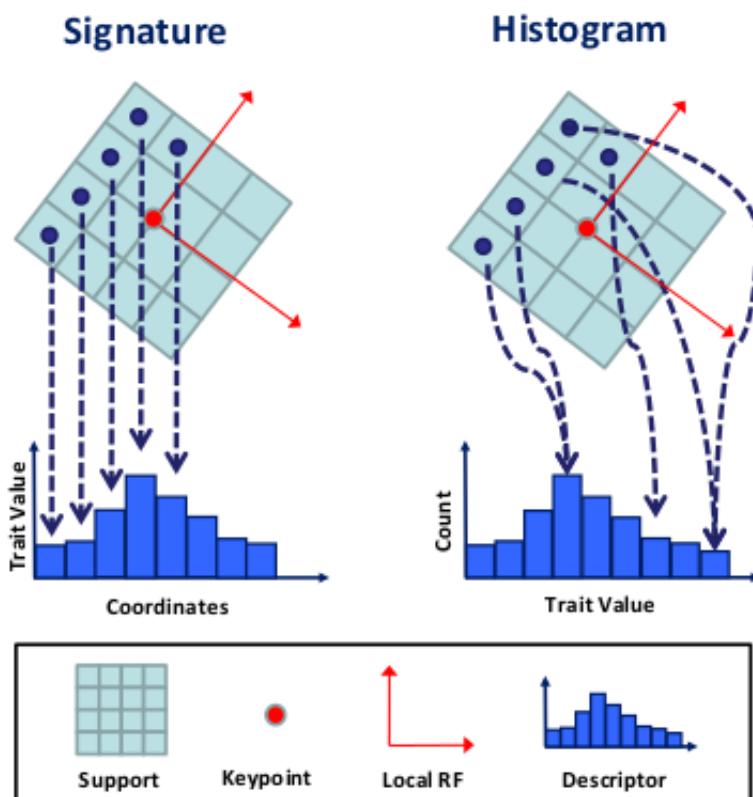


Figure 5.6: Signature v/s Histogramm[8]

### 5.4.1 Surface Normal Estimation

Die Berechnung von Normalen-Vektoren für jeder Punkt ist die Grundlage von 'Features'. Ein Mangel an Kenntnis über die Normalen-Vektoren macht die Berechnung von fortgeschrittenen *Features*, die später erläutert werden unmöglich. Weitere Informationen über die Mathematische Berechnung dieses Verfahren ist in der Bachelorarbeit vom Aaron Klein vorhanden und werden hierfür nur erwähnt.[9] In Abbildung 5.7 sind die Normalen-Vektoren von der Szene in Abbildung 5.4 veranschaulicht. Das Schuhpaar ist deutlich zu erkennen. Wichtig ist hierbei dass die Vektoren in Bezug auf dem selben Koordinatensystem orientiert sind und nach außen zeigen.

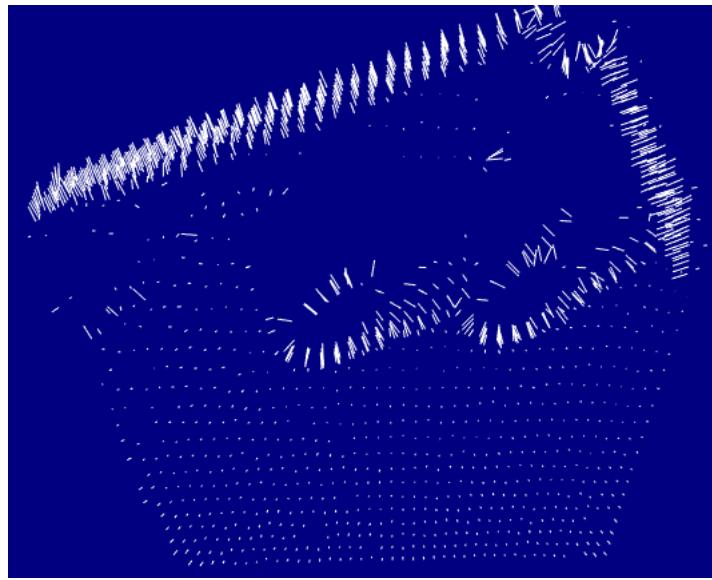


Figure 5.7: Normal Estimation

### 5.4.2 Viewpoint Feature Histogram, VPFH

Das 'Viewpoint Feature Histogram' wurde früher schon in der BA von Neri Marschik und Aaron Klein erwähnt. Hierbei wird eine kurze Analyse über seine Funktionsweise gegeben. Das *VPFH* basiert sich auf dem *FPFH*<sup>a</sup>. Das *VPFH* ist eine Erweiterung vom *FPFH* wobei die Information über den Blickwinkel zusätzlich kombiniert wird. Demnächst enthält das *VPFH* 2 Komponenten nämlich einen *Blickwinkel Richtungskomponent* und der *FPFH* Komponent. Für die Berechnung vom Blickwinkel Richtungskomponent wird der Schwerpunkt des Objektes ermittelt. Danach, der Vektor zw. dem Blickwinkel Richtung Komponent und dem Schwerpunkt des Objektes wird berechnet. Letztendlich werden für alle Punkte in dem Cluster vom Objekt der Winkel zw. diesem Vektor und deren Normalen-Vektoren berechnet und in einem Histogramm gruppiert; sog. *binning*. Der zweite Komponent ist wie der *FPFH* berechnet.

Aufgrund dass dieses *Descriptor* die Verwendung vom Blickwinkel enthält, der für ein Cluster fest ist, ist *VPFH* ein *Global Descriptor* und benötigt demnächst falls es in eine Recognition Pipeline benutzt wird alle zusätzliche Verarbeitungen wie vorher beim *Global Descriptor* erwähnt wurden (s. *Taxonomie*).

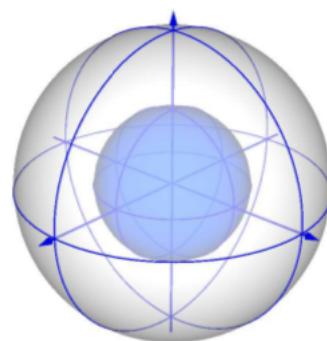
<sup>a</sup>[http://pointclouds.org/documentation/tutorials/fpfh\\_estimation.php](http://pointclouds.org/documentation/tutorials/fpfh_estimation.php)

### 5.4.3 Signature of Histograms of Orientations, SHOT [6]

Signature of Histograms of Orientations (SHOT) Descriptor kodiert eine Signatur von Histogrammen für die Darstellung topologischer Merkmale. Sie ist unveränderlich zu Rotation & Translation und robust gegenüber Rauschen und Unordnung. Der Deskriptor für einen bestimmten '*keypoint*' wird durch die Berechnung von lokalen Histogramme gebildet, die geometrische Informationen von '*keypoints*' innerhalb einer sphärischen Stützstruktur enthalten. Dadurch kodieren wir die Information über die Struktur ähnlich wie eine '*Signatur*'.

Die Herangehensweise wird durch die folgende Stichpunkte erläutert.

1. Eine Gruppe von lokalen Histogramme werden berechnet.
2. Für jedes lokal Histogramm werden Punkte anhand von cosinus des Winkels  $\theta_q$  zw. dem normalen Vektor an dem Punkt,  $n_q$ , und die lokale  $z$ -Achse an dem *Feature Point (keypoint)*,  $z_q$  gesammelt. Der Grund für die Berechnung von cosinus ist zweifach.
  - a)  $\cos \theta_q = z_k \cdot n_q$  kann einfach und schnell berechnet werden.
  - b) Die Gleiche räumliche 'Binning' auf  $\cos \theta_q$  ist äquivalent mit einer räumliche variierende 'Binning' auf  $\theta_q$ . Einerseits wird eine grobe Gruppierung für Richtungen Nähe zur Normalreferenz Richtung durchgeführt andererseits eine feinere für die Orthogonal Richtungen. Somit, entstehen kleine Unterschiede in Orthogonal Richtungen zum normalen Vektor. Diese sind am aufschlussreichsten und werden in unterschiedliche Histogramme erhoben.
3. Es wird für die Signature Struktur eine isotropische Kugelförmige Gitternetze (s. Abbildung 5.8 ) verwendet, die Trennwände entlang der radialen-, Azimuth- und Elevationsachsen umfassen. Hierbei wird eine grobe Unterteilung der Achsen bevorzugt aufgrund der beschriebenden lokalen Histogram, die schon viele Informationen liefern. Sonst wird zu groß Rechnerleistung erforderlich.



**Figure 5.8:** Signature Structure[6]

Als *pseudocode* ist SHOT wie im Algorithmus 5.1 implementiert.

---

**Algorithmus 5.1** SHOT Descriptor

---

$F = \{\text{feature points}\}$   
 $S_p = \{\text{points in the sphere of radius } R \text{ around } p\}$   
RGBData = a boolean indicating RGB-D data  
compute normal  $n_i$  for every point  $i$   
if RGBData then  
    convert  $RGB_i$  to  $Lab_i$  for every point  $i$   
end if  
for  $p \in F$  do  
    //Computation of the local Reference Frame  
     $M = 0$   
    for  $q \in S_p$  do  
         $d = \|p - q\|_2$   
         $M = M + (R - d)(p - q)(p - q)^T$   
    end for  
     $M = VDV^{-1}$ ,  $V = [x^+ \ y^+ \ z^+]$  //Compute Eigen Vector Decomposition (EVD)  
    //Disambiguate axes  
     $x = x^+$  if  $|S_x^+| \leq |S_x^-|$  else  $x^-$   
     $z = z^+$  if  $|S_z^+| \leq |S_z^-|$  else  $z^-$   
     $y = z \times x$   
    //Computation of the signature of histograms  
    for  $q \in S_p$  do  
        compute local coordinates  $(q_x, q_y, q_z)$  of  $q$  wrt  $[x \ y \ z]$   
        quantize  $(q_x, q_y, q_z)$  wrt to the spatial grid  
         $\theta \leftarrow n_q \cdot z$   
        quantize  $\theta$  wrt to the shape histogram bins  
        if RGBData then  
             $\gamma \leftarrow \|Lab_q - Lab_p\|_1$   
            quantize  $\gamma$  wrt to the texture histogram bins  
        end if  
        quadrilinear interpolation to accumulate  $q$   
    end for  
    normalize the descriptor to Euclidean norm 1  
end for

---

## 5.5 Registration

Die Aufgabe unterschiedliche Sichten von 3D Punktewolke Daten in einem kompletten Model auszurichten heißt '*Registration*' [11], Registrierung. Das Ziel hierbei ist die relative Lagen und Positionen des erzielten Ansichten in einem globalen Koordinatensystem zu ermitteln so dass die überkreuzenden Flächen sich miteinander überlappen.

### 5.5.1 Iterative Closest Point Algorithm

Iterative Closest Point (ICP) ist sehr bekannt für die Registrierung von 3D Daten und wird sehr häufig verwendet aufgrund seiner Einfachheit in der Implementierung für Echt-zeit Aufgaben.[10] Die ICP wird unerfolgreich sein wenn die Differenz zw. 2 Punktewolken groß ist. Daher eignet sich ICP besonders gut in Kombination mit Korrespondenz Suche Verfahren wie z.B. SHOT, vorher beschrieben wobei erstens eine grobe Übereinstimmung zw. dem Modell und der Szene gerechnet wird und danach eine Verfeinerung mit ICP durchgeführt wird. Die Herangehensweise mit ICP ist in der Art:

1. Korrespondenzen zw. Modell und Szene suchen (s. Abbildung 5.9)
2. Korrespondenzen mit schlechter Güte rausfiltern
3. Euklidische Transformation; '*Rigid Transformation*' mithilfe von den guten Korrespondenzen abschätzen
4. Iteration bis Konvergenzkriterium wie zB.  
*kleinste Quadrierte Summe (LMS)  $\approx 0$  steht.*

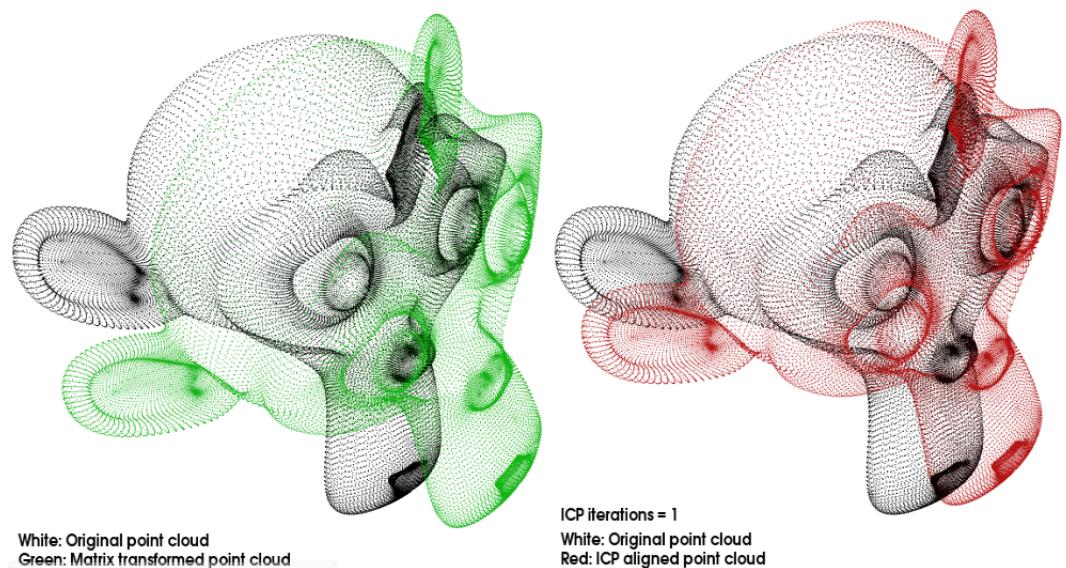
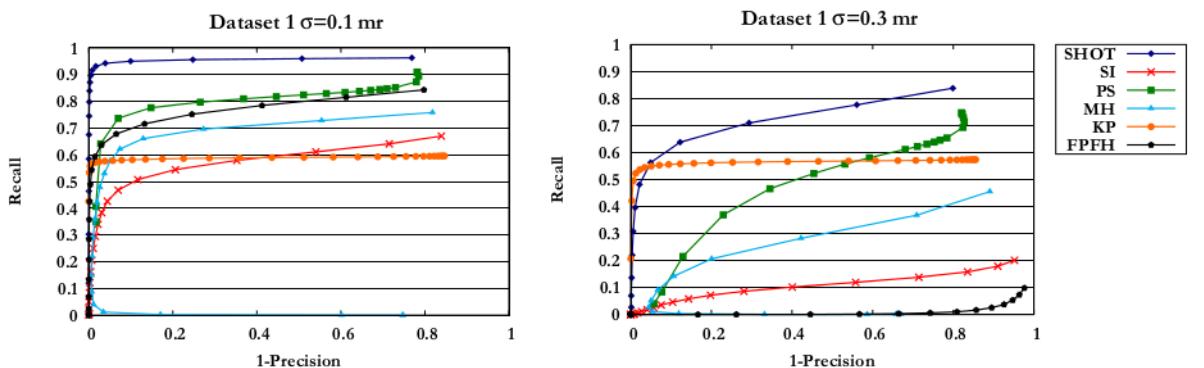


Figure 5.9: ICP[11]

### 5.5.2 Feature-Based Registration

Hierbei werden zuerst Keypoints in die 2 Punktewolke gesucht, die gegenübergemätscht werden. Dann wird es herausgefunden ob die 2 Punktewolke gemeinsam Keypoints aufweisen. Fall mindestens 3 Korrespondenzen gefunden werden, wird eine Euklidische Transformation, sog. *rigid transformation* berechnet. Für genauere Ergebnisse müssen mehrere Korrespondenzen gefunden werden. Aufgrund dass in diese Phase ebenfalls Keypoints gesucht und Features berechnet werden ist eigentlich das Feature-Based Registration in Prinzip doppelt aufwändig, was für die Implementierung ziemlich ungünstig sein kann.

## 5.6 Analysis



**Figure 5.10:** Precision & Recall Test[6]

Anhand von der Studie im Artikel Computer Vision and Image Understanding[6] wurden unterschiedlichen 3D Descriptors miteinander verglichen und die Abbildung 5.10 zeigt die Ergebnisse dieser Studie. In dieser Studie wurden 6 unterschiedliche Modelle aus dem *Stanford 3D Scanning Repository*[12] in unverdeckte Szene mit erstmal Gauß'sches weißes Rauschen mit *Standardabweichung*  $\sigma = 0.1$  & letztendlich mit  $\sigma = 0.3$  gesucht. Es ist hierbei deutlich erkennbar dass *SHOT 3D Descriptor*, deren Kennline hier *blaues* eingefärbt alle anderen Descriptor übertrifft. Bei  $\sigma = 0.1$  der Recall-Wert vom SHOT sich gegenüber Genauigkeitswert konstant verhält weswegen SHOT sehr robust erscheint. Bei  $\sigma = 0.3$  ist dennoch *Keypoint Matching*, *KP 3D Descriptor* der robuster von allem. Diese Robustheit beim *KP* setzt sich wegen der hohen Smoothing der Punktewolke durch. Dagegen verliert sich *KP* seine Deskriptivstärke gegenüber *SHOT* beim niedrigen Rauschpegel. Anschließend, ist *FPFH* am schlechtesten bei  $\sigma = 0.3$  und nur genügend bei  $\sigma = 0.1$ . Aufgrund dass *VPFH* vom *FPFH* hergeleitet ist wird seine Leistung auch ähnlich sein wie die Leistung vom *FPFH*. Grunddessen wird *SHOT* als 3D Descriptor in Kombination mit *ICP* für die Implementierung im Kapitel 6 verwendet.

## Chapter 6

# Recognition Pipeline

In diesem Kapitel werden über die mögliche Anwendung, Datentypen im ROS & im PCL verfolgt von einer vorgeschlagener Vorgehensweise nach der Analyse vom Features und Descriptors in Kapitel 5 erläutert.

### 6.1 Mögliche Anwendung

Den Einsatz von der *Intel RealSense R200 Camera in 3D Bildverarbeitung in der mobil Service Robotik kann in 2 Arten sein:*

#### 1. Umgebungserkennung

Umgebung erkennen anhand von Objekten wie z.B. *Tisch, Couch und Rollstuhl* die in einem Raum anwesend sind.

#### 2. Objekteerkennung

Hier geht es um die sog. '*Tisch Szene*' wo kleine Objekte wie z.B. Milk Box, Juice Box, Bowl, Mug usw... zu erkennen sind.

In diesem Sinne werden beides probiert und die Ergebnisse der Erkennung sowie die Evaluation im [Kapitel 6](#) erläutert.

Der nächste Schritt ist die Akquisition von rauschfreies Modell von dem zu erkennenden Objekt. Die folgende Möglichkeiten bieten sich an:

- Die erste ist der Einsatz vom CAD Model als .vtk Datei die ebenfalls im *Point Cloud Library* unterstützt sind.
- Die zweite ist die Benutzung von Punktwolke die von sehr guten Szene erhalten sind. Daraus werden das zu erkennende Objekt segmentiert. Die Rauschen anhand von Smoothing <sup>a</sup> eliminiert und die resultierende Punktwolke hoch gesampelt, sog. *upsampling* <sup>b</sup> und als Model für die Erkennung gespeichert.

---

<sup>a</sup><http://pointclouds.org/documentation/tutorials/resampling.php>

<sup>b</sup>[http://robotica.unileon.es/index.php/PCL/OpenNI\\_tutorial\\_2:\\_Cloud\\_processing\\_\(basic\)#Upsampling](http://robotica.unileon.es/index.php/PCL/OpenNI_tutorial_2:_Cloud_processing_(basic)#Upsampling)

## 6.2 Datentype

Die R200 Kamera veröffentlicht eine menge von Topics worauf wir für die weitere Verarbeitung zugreifen müssen. Das allerwichtigste ist das Topic `/camera/depth_registered/points`. Laut dem heutigen Stand der Technik unterstützt PCL unterschiedliche Datentypen für die Darstellung der Punktwolke wie `pcl::PointCloud<T>` und `sensor_msgs::PointCloud2`. Allerdings nur die 2te Datentyp wird auch im ROS unterstützt. Dafür muss man die Header-Datei `#include <sensor_msgs/Point Cloud2.h>` (s. Abbildung 6.1) einbinden. Deshalb wird diese Datentyp am häufigsten verwendet. Für die Darstellung von Normalen-Vektoren, 'Clusters' (s. Abbildung 5.5) und 'Correspondences' (s. Abbildung 7.2) wird die `.pcd` Format verwendet da diese 3 sehr gut im *PCL Visualizer* dargestellt werden können.

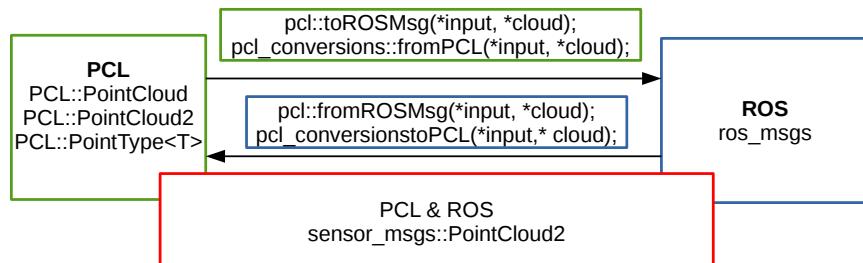


Figure 6.1: Message Types

## 6.3 Pipeline

Aufgrund dass ROS als Software Framework verwendet wird, erfolgt die Programmierung gemäß *ROS Dateisystem & ROS Computation Graph*. Nachrichten werden über einen *Publisher-Subscriber Mechanismus* verteilt. Ein Knoten veröffentlicht Nachrichten zu einem Thema (publish to a topic) identifiziert über einen Namen. Mehr dazu ist im *ROS wiki* zu finden. (<http://wiki.ros.org/de/ROS/Concepts> [13]) Mit dem *Publisher-Subscriber Mechanismus* wird die '*Recognition Pipeline*' gekoppelt und erfolgt wie folgend in 2 Hauptschritte:

### 1. Clustering Subscriber & Publisher

Hierbei greift das Programm auf das von dem R200 veröffentlichten Topic, */camera/depth\_registered/points* und veröffentlicht die Clusters auf das Topic */clusters* und werden ebenfalls für weitere Verarbeitung als *record2.pcd* gespeichert.

### 2. Matching

*record2.pcd* wird aufgerufen und wird gegen das Modell vom gesuchten Objekt übereingestimmt. Zusätzlich werden die Anzahl an Korrespondenzen zw. der Szene und dem Modell gezeigt.

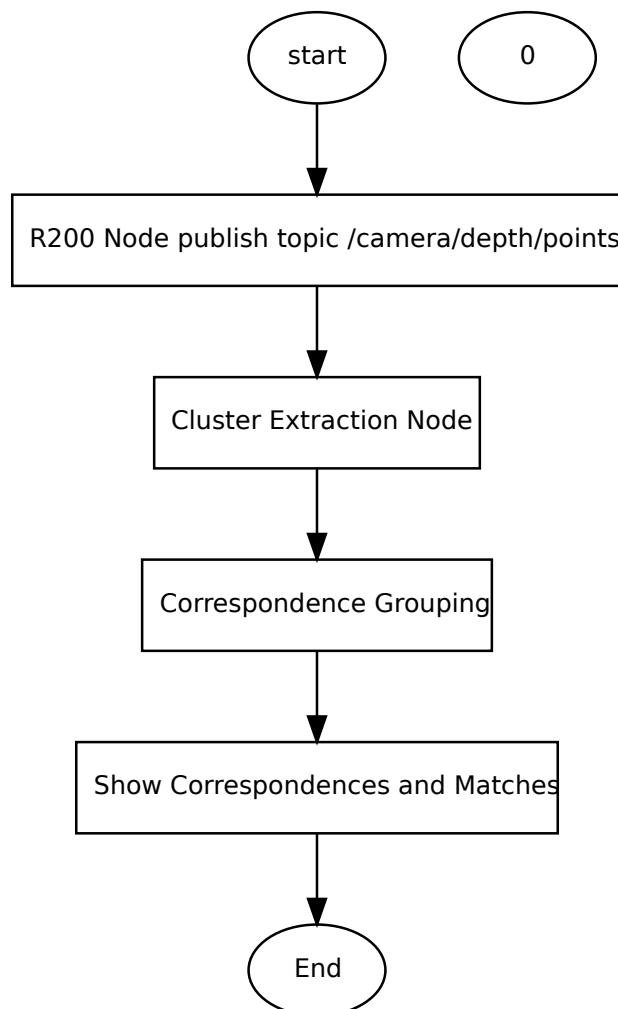


Figure 6.2: Overall Pipeline

### 6.3.1 Clustering Subscriber & Publisher

Der 'Euclidean Clustering' Algorithmus im Kapitel 5 ist der 2te. Knoten, sog. *node* von unserer *Pipeline*, ist als *Subscriber & Publisher* geschrieben und seine Schritte wie folgend veranschaulicht. Das Clustering erfolgt in Echtzeit. Im Anhang ist die Quellecode dazu zu finden.

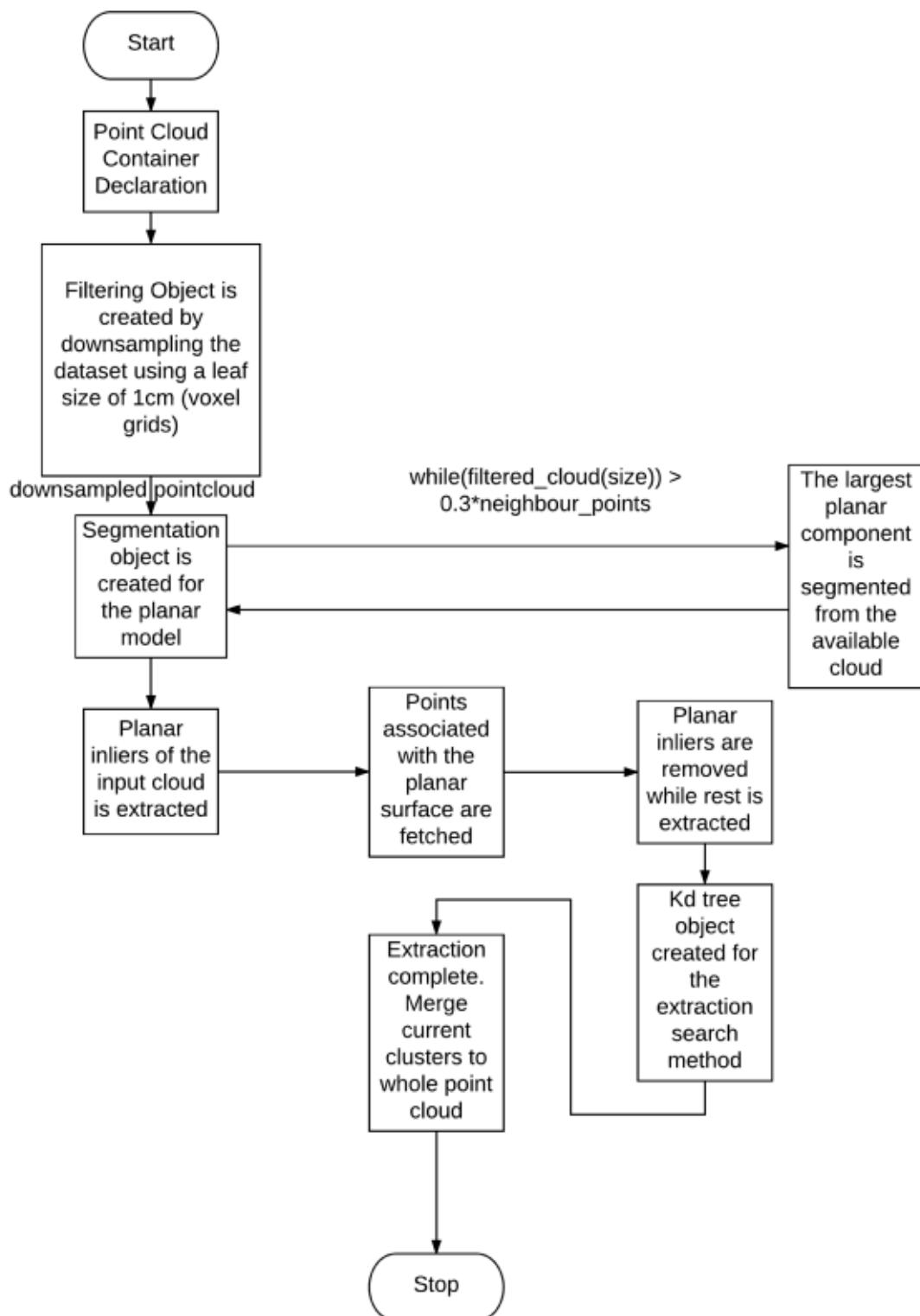
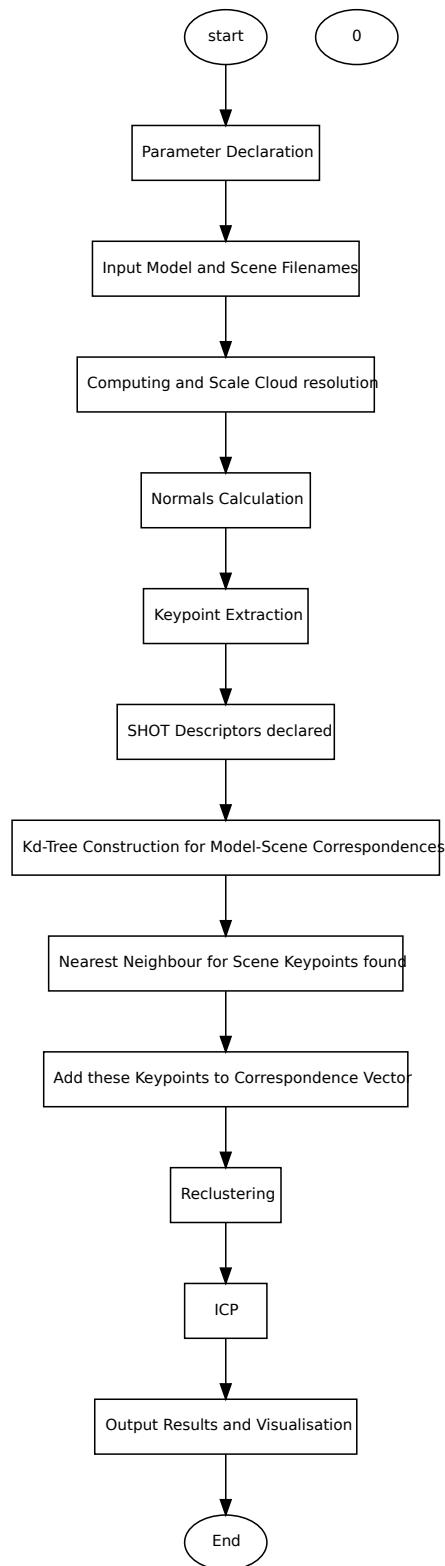


Figure 6.3: Clustering Pipeline

### 6.3.2 Matching



**Figure 6.4:** Matching

Die vom *Clustering Knoten* veröffentlichten *Clusters* müssen fürs Matching vom *Matching Knoten* zugegriffen werden. Dieser Knoten wird wie in Abbildung 6.4 erklärt.

### 6.3.3 Gesamte Pipeline

Die gesamte Pipeline enthält jeden Schritt der *3D* Bildverarbeitung; von der Filterung bis zur Registrierung, sog. *Registration*.

1. **Filterung** Die Punktewolke wird erstmal gefiltert. Hierbei wird die rohe Punktewolke durch den *Passthrough Filter* eingekreist. Die gelieferte Punktewolke von der *R200* Kamera ist inhomogene und nicht dicht, sondern enthält viel 'Löchern'. Deswegen sind die streue Punkte dank des *Statistical Outlier Removal Filters* zum Teil eliminiert. Danach wird die Punktewolke mithilfe vom *Voxel Grid Filter* runter gesampelt.
2. **Segmentierung** Diese Punktewolke wird anhand vom *Euclidean Clustering* segmentiert und Clusters werden erhalten.
3. **Feature + Descriptor** Die Normalen-Vektoren an jedem Punkt im *Cluster* werden berechnet und diese Vektoren werden benutzt um unsere 'Descriptor' *SHOT* (s. Abbildung 5.6) zu definieren und es eventuell an jedem Punkt kalkuliert.
4. **Correspondence Search** Die Korrespondenzen zw. dem Model und der Szene, sog. '*Model-Scene Correspondences*' werden anhand dem Kd-Tree[9] gesucht.
5. **Correspondence Filtration** Die guten Korrespondenzen werden erhalten und die schlechteren raus geschmissen.
6. **Registration** Es wird danach eine Euklidische Transformation, sog. '*rigid transformation*' der besten Gruppe von Korrespondenzen mit *ICP* durchgeführt.

## Chapter 7

# Validierung, Ergebnisse & Optimierung

In diesem Kapitel geht es zunächst einmal um das Testen und Beurteilung von der Erkennung und daraus welche Schwäche vorhanden sind und worauf sie sich beziehen. Letztendlich beenden wir diesen Abschnitt mit Weiterentwicklung und Optimierung.

### 7.1 Testing Setup

Die Bedingungen, die während dem Testen anwesend sind und die Versuchsanordnung beeinflussen die Ergebnisse. Insofern ist es erforderlich dass die Ergebnisse wissenschaftlich reproduzierbar sind.

Fürs Testen wurde folgende Schritte durchgeführt.

1. Die R200 Kamera mit roslaunch,
2. Das *ROS Visualisierungsplugin rviz*,
3. Das Subscriber cluster\_extraction,
4. & letztendlich das Programm correspondence\_grouping gestartet.

Components	Name
CPU	Intel®Core™i7 – 3610QM CPU@2.30GHz×8
Memory	7,7GiB
Graphics Adapter	Intel®Ivybridge Mobile Nvidia Optimus 650M
OS	ubuntu 14.04 LTS 64 bit
Compiler	gcc 4.8.4
Libraries	ROS Indigo, Point Cloud Library 1.7

**Table 7.1:** Hardware Setup

Die originale *Intel RealSense R200 Kamera* ist ohne Firmware Upgrade verwendet. In Tabelle 7.1 wird das Hardware Setup dargestellt. Der Rechner verfügt sich über eine Graphik Karte, *Intel®Ivybridge Mobile Nvidia Optimus 650M*, was aber nicht mit *PCL* zur Nutzung gebracht werden konnte. Die OMP Klassen vom *PCL* wurden benutzt um die Berechnungszeit zu verringern. Diese Klassen verwenden die '*multi-core/multi-threaded paradigms*' vom *OpenMP*.

Getestet wurde im Raum mit '*ambient*' Umgebungslicht. Die *Intel RealSense R200 Kamera* wurde ab einer radialen Entfernung von ~ 70cm platziert und während der Aufnahme und Erkennung stabil erhalten. Im Rahmen des Tests waren immer das zu erkennende Objekt deutlich sichtbar und nicht verdeckt.

## 7.2 Ergebnisse + Schlussfolgerungen

Es wurde ebenfalls im Freien am sonnigen Tag probiert und schnell bemerkt dass der IR Projektor keine Wirkung gegenüber dem Umgebungslicht der Sonne hat. Die Rekonstruktion der Umgebung und des Bodens war schlecht. Fürs RGB Bild kann dennoch bis zu einer gewissen Grenze die dynamische Parameter wie z.B. *gain*, *exposure* & *fps* des Streamings für ein besseres Ergebnis optimiert werden. Aus diesem Grund geschah die Erkennungsversuche nur im Raum.

Die Ergebnisse werden in die Kategorie Umgebungserkennung & Objekterkennung unterteilt.

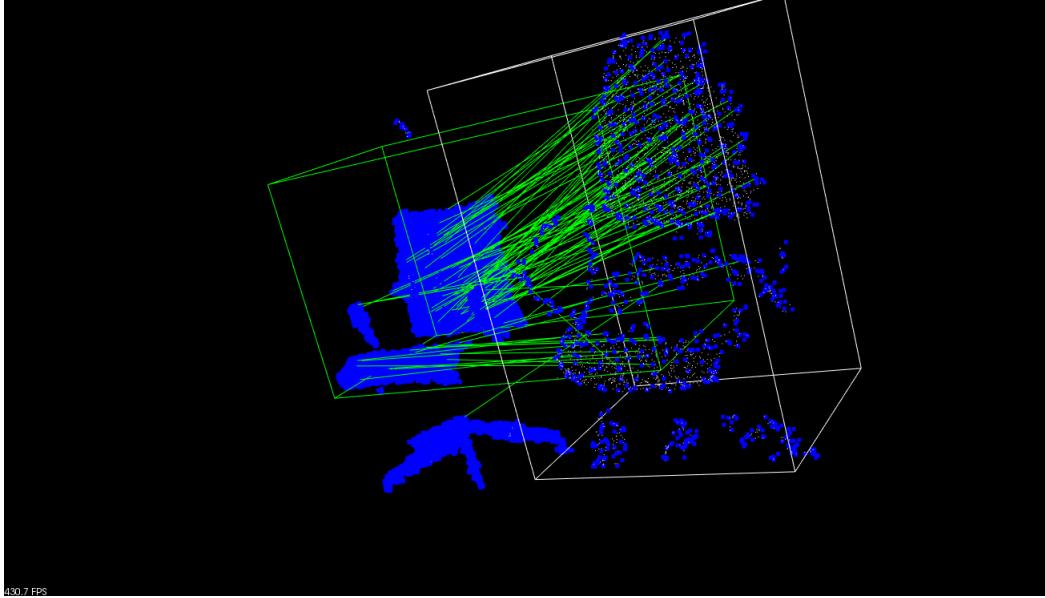
### 7.2.1 Umgebungserkennung

In diesem Fall hat die Erkennung sich zuverlässig gezeigt da die Objekte wie zB. Rollstuhl, Tisch und Sofa einfach mehr Punkte erhalten.



**Figure 7.1:** Chair Scene

In die obigen Abbildung ist festzustellen dass große Schatten in der Szene vorhanden sind. Dieses zeichnet einen Nachteil des Aktiv Stereosystems. In Wirklichkeit wie im Kapitel 5 erwähnt erhalten die ferne Objekte die Teilweise aufgrund eines vorderen großen Objekts kein Projektionsmuster mit genug Beleuchtungsstärke, weswegen die 'uniqueness' dieses Teils verschwindet. Infolgedessen haben diese 3D Koordinaten keine Tiefe Punkte und dadurch entstehen 'schwarze Löchern' in der Szene.



**Figure 7.2:** Correspondences

Das Objekt ist jedoch erkannt aufgrund der großen Anzahl an Keypoints und deren guten Verteilung im Objekt. Deswegen ist hier keine Segmentierung wie z.B. *Euclidean Clustering* erforderlich.

Die Anzahl an Keypoints in der Szene und Model muss soweit wie möglich in der gleichen Ordnung sein. Sonst ergeben viele 'false positives'. Das runtersampeln muss aus diesem Grund sehr gut eingestellt werden. In unserem Test waren der Sucheradius fürs Runtersampeln der Szene 3mal so groß wie der Radius fürs Model. Einerseits verursacht stark Runtersampeln schwache Erkennungen aufgrund der verlorenen wichtigen Merkmale, andererseits zu wenig Runtersampeln ergibt 'false positives.'

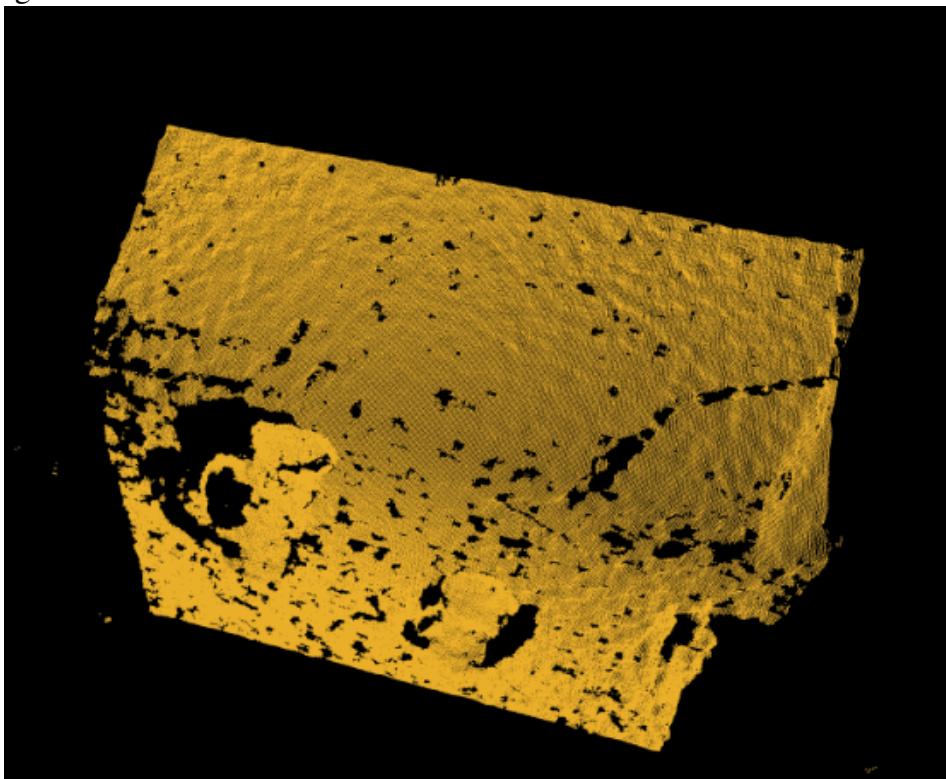
### 7.2.2 Objekterkennung

In dieser Kategorie wurde Objekte wie z.B. *Milkbox*, *Juicebox*, *Bowl*, *Mug & Glass* getestet. Alle Objekte waren unverdeckt und auf einem Tisch gelegt. Hierbei ist Segmentierung erforderlich. Es wird *Euclidean Clustering* verwendet. Die Punktwolke, die die *Intel®Real-SenseTM Camera R200* liefert, enthält im meisten Fällen große Lücke und verrauschte Clusters (s. Abbildung 7.3a), was teilweise anhand des 'Statistical Outlier Removal Filters' eliminiert werden kann. Nichtsdestotrotz stellen diese große Lücke ein Problem für den Algorithmus von Euclidean Clustering wenn die Lücke größer als die Toleranz-Distanz, sog. 'Cluster Tolerance Distance' (s. Algorithmus 8.2) sind. Infolgedessen muss die *Intel RealSense R200 Kamera* für die Tisch Szene, bzw. für die Objekt Erkennung zu einer geschlossenen Ecke ausgerichtet werden.



(a) Mug Scene Open

Beispielsweise sind die Objekte auf dem Boden gelegt und folgendermaßen ist eine 'gute Szene' ohne Lücke zu sehen.

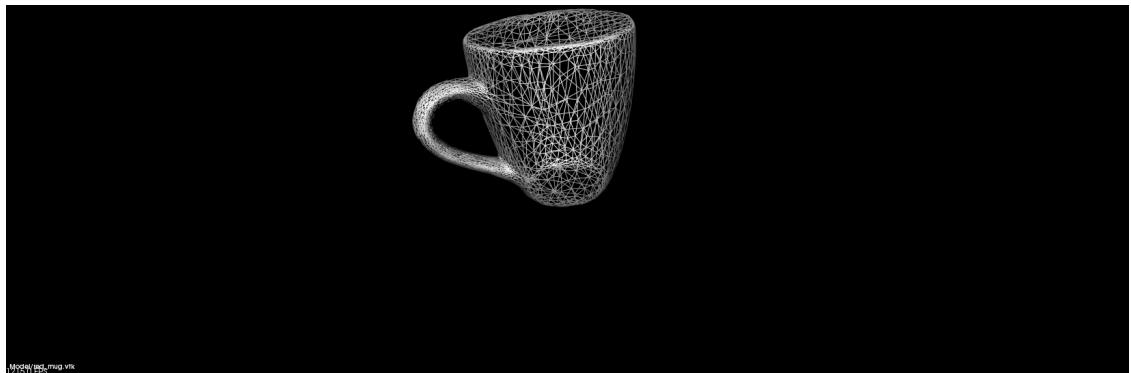


**Figure 7.3:** Mug Scene

Danach wird diese Szene in Clusters zerlegt. Es entstand 3 Clusters die in Abbildung 7.4 veranschaulicht sind. Jedes Cluster wird mit dem Model abgestimmt und das erste Cluster (s. Abbildung 7.4) stimmt mit dem Model (s. Abbildung 7.5).



**Figure 7.4:** Mug Cluster



**Figure 7.5:** Mug Mesh

### 7.3 Laufzeit

Es wird neben dem Erkennungsversuch die Laufzeit ebenfalls gemessen. Die Tabelle 7.2 zeigt die Ergebnisse von diesem Test. Je größer die Punktanzahl desto größer ist die Ladezeit außer beim *Milkbox*. Dieses Objekt ist einfach ein Quader deren Punkte geometrisch und kompakt angeordnet sind. Aus diesem Grund ist die Ladezeit am kleinsten gegenüber den anderen. Im allgemeinen werden das Model und die Szene als *.pcd* Datei von der Festplatte geladen und dieses kostet Bearbeitungszeit. Daher wird diese Studie für die Optimierung weiter benutzt.

Objekt	Punktzahl	Ladezeit/ms
<i>RollingChair</i>	6798	56
<i>Table</i>	7527	74
<i>Milkbox</i>	13704	21
<i>Bowl</i>	459	34
<i>Mug</i>	144	31

**Table 7.2:** Loading Time

## 7.4 Optimierung

Für die Erweiterbarkeit ist einen Blick auf die Optimierungsmöglichkeiten erforderlich. Es muss dabei verdeutlicht werden dass die implementierte Pipeline nicht in Konkurrenz mit anderen derzeitigen industriellen Algorithums und Software läuft sondern dient nur zu beweisen dass die *Intel RealSense R200 Kamera* ebenfalls für die Erkennung verwendet werden kann. Jedoch sind hierfür noch einige Teile von unserem Herangehensweise zu optimieren, die wie folgend gelistet werden.

- Verbesserung der Erkennung
  1. Das Rauschen der Punktwolke, was von der *Aktiv Stereobildverarbeitung* und vom *Range Multiplexing* sich vererbt ist noch zu eliminieren. Dafür kann anhand von weiteren Studien ein Rausch-Model gestellt werden und eine Vorgehensweise zur quasi vollständigen Eliminierung entwickelt werden.
  2. Anstatt von einer expliziten Suche der Korrespondenzen des Modells in der Szene kann ein 'Einteachen Verfahren' mit *neuronalen Netzwerken* wie z.B. CNN und Deep Learning implementiert werden. Dann bietet die Erkennung mehr Flexibilität.
- Auf Echtzeit Anwendung zu Optimieren

Die Ladezeit von dem Modell und der Szene können dank der Nutzung von SSDs reduziert werden. Weiterhin können während des '*Clusterings*' anhand der Parallelisierung die anderen Kerne der CPU fürs Laden vom Modell benutzt werden um die Echtzeit anzunähern.

# **Chapter 8**

## **Quellecode**

---

## Algorithmus 8.1 Voxel Filter

---

```
#include <iostream>
#include <ros/ros.h>

//PCL specific includes

#include <sensor_msgs/PointCloud2.h>
#include <pcl_conversions/pcl_conversions.h>
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/approximate_voxel_grid.h>

ros::Publisher pub;

//Modified Callback Function to eliminate noise in z-axis

void cloud_cb (const sensor_msgs::PointCloud2ConstPtr& cloud_msg)
{
    // Container for original & filtered data
    pcl::PCLPointCloud2* cloud = new pcl::PCLPointCloud2;
    pcl::PCLPointCloud2ConstPtr cloudPtr (cloud);
    pcl::PCLPointCloud2 cloud_sampled;
    // Convert to PCL data type pcl_conversions::toPCL(*cloud_msg, *cloud);
    // Perform the actual filtering
    pcl::VoxelGrid<pcl::PCLPointCloud2> sor;
    sor.setInputCloud (cloudPtr);
    sor.setLeafSize (0.01, 0.01, 0.01);
    sor.filter (cloud_sampled);
    //Convert to ROS data type
    sensor_msgs::PointCloud2 output_sampled;
    pcl_conversions::fromPCL (cloud_sampled, output_sampled);
    //Publish the data
    pub.publish (output_sampled);
}

int main (int argc, char** argv)
{
    //Initialize ROS
    ros::init (argc, argv, "my_pcl_tutorial");
    ros::NodeHandle nh;
    // Create a ROS subscriber for the input point cloud
    ros::Subscriber sub = nh.subscribe ("input_sampled", 1, cloud_cb);
    //Create a ROS publisher for the output model coefficients
    //model coefficients is of type pcl_msgs
    pub = nh.advertise<sensor_msgs::PointCloud2> ("output_sampled", 1);
    // Spin
    ros::spin ();
}
```

---

---

## Algorithmus 8.2 Euclidean Clustering 1 of 3

---

```
#include <pcl/ModelCoefficients.h>
#include <pcl/point_types.h>
#include <pcl/io/pcd_io.h>
#include <pcl/filters/extract_indices.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/features/normal_3d.h>
#include <pcl/kdtree/kdtree.h>
#include <pcl/sample_consensus/method_types.h>
#include <pcl/sample_consensus/model_types.h>
#include <pcl/segmentation/sac_segmentation.h>
#include <pcl/segmentation/extract_clusters.h>

int main (int argc, char** argv)
{
    // Read in the cloud data
    pcl::PCDReader reader;
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud (new pcl::PointCloud<pcl::PointXYZ>),
    cloud_f (new pcl::PointCloud<pcl::PointXYZ>);
    reader.read ("table_scene_lms400.pcd", *cloud);

    std::cout << "PointCloud before filtering has: " << cloud->points.size ()
    << "data points." << std::endl; /*

    //Create the filtering object: downsample the dataset using a leaf size of 1cm
    pcl::VoxelGrid<pcl::PointXYZ> vg;
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_filtered (new pcl::PointCloud<pcl::PointXYZ>);
    vg.setInputCloud (cloud);
    vg.setLeafSize (0.01f, 0.01f, 0.01f);
    vg.filter (*cloud_filtered);

    std::cout << "PointCloud after filtering has: " << cloud_filtered->points.size ()
    <<
    "data points." << std::endl; /*

    //Create the segmentation object for the planar model and set all the parameters
    pcl::SACSegmentation<pcl::PointXYZ> seg;
    pcl::PointIndices::Ptr inliers (new pcl::PointIndices);
    pcl::ModelCoefficients::Ptr coefficients (new pcl::ModelCoefficients);
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_plane (new pcl::PointCloud<pcl::PointXYZ> ());
    pcl::PCDWriter writer;
    seg.setOptimizeCoefficients (true);
    seg.setModelType (pcl::SACMODEL_PLANE);
    seg.setMethodType (pcl::SAC_RANSAC);
    seg.setMaxIterations (100);
    seg.setDistanceThreshold (0.02);
```

---

---

### Algorithmus 8.3 Euclidean Clustering 2 of 3

---

```
int i=0, nr_points = (int) cloud_filtered->points.size ();
while (cloud_filtered->points.size () > 0.3 * nr_points)
{
    //Segment the largest planar component from the remaining cloud
    seg.setInputCloud (cloud_filtered);
    seg.segment (*inliers, *coefficients);
    if (inliers->indices.size () == 0)
    {
        std::cout << "Could not estimate a planar model for the given dataset." << std
            ::endl;
        break;
    }
    //Extract the planar inliers from the input cloud
    pcl::ExtractIndices<pcl::PointXYZ> extract;
    extract.setInputCloud (cloud_filtered);
    extract.setIndices (inliers);
    extract.setNegative (false);

    //Get the points associated with the planar surface
    extract.filter (*cloud_plane);
    std::cout << "PointCloud representing the planar component: " << cloud_plane
        ->points.size
    ()
    << " data points." << std::endl;

    //Remove the planar inliers, extract the rest
    extract.setNegative (true);
    extract.filter (*cloud_f);
    *cloud_filtered = *cloud_f;
}

//Creating the KdTree object for the search method of the extraction
pcl::search::KdTree<pcl::PointXYZ>::Ptr tree (new pcl::search::KdTree<pcl::
    PointXYZ>);
tree->setInputCloud (cloud_filtered);

std::vector<pcl::PointIndices> cluster_indices;
pcl::EuclideanClusterExtraction<pcl::PointXYZ> ec;
ec.setClusterTolerance (0.02); //2cm
ec.setMinClusterSize (100);
ec.setMaxClusterSize (25000);
ec.setSearchMethod (tree);
ec.setInputCloud (cloud_filtered);
ec.extract (cluster_indices);

int j = 0;
for (std::vector<pcl::PointIndices>::const_iterator it = cluster_indices.begin
    (); it !=
    cluster_indices.end (); ++it)
{
    pcl::PointCloud<pcl::PointXYZ>::Ptr cloud_cluster (new pcl::PointCloud<pcl
        ::PointXYZ>);
    for (std::vector<int>::const_iterator pit = it->indices.begin (); pit != it
        ->indices.end (); ++pit)
```

---

**Algorithmus 8.4** Euclidean Clustering 3 of 3

---

```
cloud_cluster->points.push_back (cloud_filtered->points[*pit]);
cloud_cluster->width = cloud_cluster->points.size ();
cloud_cluster->height = 1;
cloud_cluster->is_dense = true;

std::cout << "PointCloud representing the Cluster: " << cloud_cluster->
    points.size
()<< " data points." << std::endl;
std::stringstream ss;
ss << "cloud_cluster_" << j << ".pcd";
writer.write<pcl::PointXYZ> (ss.str (), *cloud_cluster, false);/*
j++;
}
return (0);
}
```

---

---

**Algorithmus 8.5** Correspondence Grouping 1 of 13

---

```
#include <pcl/io/pcd_io.h>
#include <pcl/point_cloud.h>
#include <pcl/correspondence.h>
#include <pcl/features/normal_3d_omp.h>
#include <pcl/features/shot_omp.h>
#include <pcl/features/board.h>
//#include <pcl/filters/uniform_sampling.h>
#include <pcl/keypoints/uniform_sampling.h>
#include <pcl/filters/voxel_grid.h>
#include <pcl/filters/approximate_voxel_grid.h>
#include <pcl/recognition/cg/hoough_3d.h>
#include <pcl/recognition/cg/geometric_consistency.h>
#include <pcl/visualization/pcl_visualizer.h>
#include <pcl/kdtree/kdtree_flann.h>
#include <pcl/kdtree/impl/kdtree_flann.hpp>
#include <pcl/common/transforms.h>
#include <pcl/console(parse.h>
#include <pcl/filters/passthrough.h>
#include <pcl/filters/statistical_outlier_removal.h>
#include <pcl/registration/icp.h>

//typedef pcl::PointXYZRGB PointType;
typedef pcl::PointXYZ PointType;
typedef pcl::Normal NormalType;
typedef pcl::ReferenceFrame RFType;
typedef pcl::SHOT352 DescriptorType;

std::string model_filename_;
std::string scene_filename_;

//Algorithm params
bool show_keypoints_ (true);
bool show_correspondences_ (true);
bool use_cloud_resolution_ (false);
bool use_hough_ (true);
//float model_ss_ (0.01f);
float model_ss_ (0.01f);
float scene_ss_ (0.03f);
float rf_rad_ (0.12f);
float descr_rad_ (0.04f);
float cg_size_ (0.24f);
float cg_thresh_ (5.0f);

int icp_max_iter_ (1000);

float icp_corr_distance_ (0.005f);
```

---

---

## Algorithmus 8.6 Correspondence Grouping 2 of 13

---

```
void showHelp (char *filename)
{
    std::cout << std::endl;
    std::cout << "
*****"
    <<
    std::endl;
    std::cout << "*"
        <<
    std::endl;
    std::cout << "*
Correspondence Grouping Tutorial - Usage Guide"
        <<
    std::endl;
    std::cout << "*"
        <<
    std::endl;
    std::cout << "*"
        <<
    std::endl;
    std::cout << "
*****"
    <<
    std::endl << std::endl;
    std::cout << "Usage: " << filename << " model_filename.pcd scene_filename.pcd ["
        Options]" <<
    std::endl << std::endl;
    std::cout << "Options:" << std::endl;
    std::cout << "    -h:                                Show this help." << std::endl;
    std::cout << "    -k:                                Show used keypoints." << std::endl;
    std::cout << "    -c:                                Show used correspondences." << std::endl;
    std::cout << "    -r:                                Compute the model cloud resolution and
        multiply" <<
    std::endl;
    std::cout << "                                each radius given by that value." << std::
        endl;
    std::cout << "    --algorithm (Hough|GC): Clustering algorithm used (default Hough
        )." <<
    std::endl;
    std::cout << "    --model_ss val:      Model uniform sampling radius (default
        0.01)" <<
    std::endl;
    std::cout << "    --scene_ss val:      Scene uniform sampling radius (default
        0.03)" <<
    std::endl;
    std::cout << "    --rf_rad val:       Reference frame radius (default 0.015)" <<
    std::endl;
    std::cout << "    --descr_rad val:   Descriptor radius (default 0.02)" << std::
        endl;
    std::cout << "    --cg_size val:     Cluster size (default 0.01)" << std::endl;
    std::cout << "    --cg_thresh val:  Clustering threshold (default 5)" << std::
        endl<<
    std::endl;
}
```

---

---

**Algorithmus 8.7 Correspondence Grouping** 3 of 13

---

```
void parseCommandLine (int argc, char *argv[])
{
    //Show help
    if (pcl::console::find_switch (argc, argv, "-h"))
    {
        showHelp (argv[0]);
        exit (0);
    }
    //Model & scene filenames
    std::vector<int> filenames;
    filenames = pcl::console::parse_file_extension_argument (argc, argv, ".pcd");
    if (filenames.size () != 2)
    {
        std::cout << "Filenames missing.\n";
        showHelp (argv[0]);
        exit (-1);
    }
    model_filename_ = argv[filenames[0]];
    scene_filename_ = argv[filenames[1]];

    //Program behavior
    if (pcl::console::find_switch (argc, argv, "-k"))
    {
        show_keypoints_ = true;
    }
    if (pcl::console::find_switch (argc, argv, "-c"))
    {
        show_correspondences_ = true;
    }
    if (pcl::console::find_switch (argc, argv, "-r"))
    {
        use_cloud_resolution_ = true;
    }

    std::string used_algorithm;
    if (pcl::console::parse_argument (argc, argv, "--algorithm", used_algorithm) != -1)
    {
        if (used_algorithm.compare ("Hough") == 0)
        {
            use_hough_ = true;
        }
        else if (used_algorithm.compare ("GC") == 0)
        {
            use_hough_ = false;
        }
        else
        {
            std::cout << "Wrong algorithm name.\n";
            showHelp (argv[0]);
            exit (-1);
        }
    }
}
```

---

---

**Algorithmus 8.8 Correspondence Grouping 4 of 13**

---

```
//General parameters
pcl::console::parse_argument (argc, argv, "--model_ss", model_ss_);
pcl::console::parse_argument (argc, argv, "--scene_ss", scene_ss_);
pcl::console::parse_argument (argc, argv, "--rf_rad", rf_rad_);
pcl::console::parse_argument (argc, argv, "--descr_rad", descr_rad_);
pcl::console::parse_argument (argc, argv, "--cg_size", cg_size_);
pcl::console::parse_argument (argc, argv, "--cg_thresh", cg_thresh_);
}

double computeCloudResolution (const pcl::PointCloud<PointType>::ConstPtr &cloud)
{
    double res = 0.0;
    int n_points = 0;
    int nres;
    std::vector<int> indices (2);
    std::vector<float> sqr_distances (2);
    pcl::search::KdTree<PointType> tree;
    tree.setInputCloud (cloud);

    for (size_t i = 0; i < cloud->size (); ++i)
    {
        if (! pcl_isfinite ((*cloud)[i].x))
        {
            continue;
        }

        //Considering the second neighbor since the first is the point itself.
        nres = tree.nearestKSearch (i, 2, indices, sqr_distances);
        if (nres == 2)
        {
            res += sqrt (sqr_distances[1]);
            ++n_points;
        }
    }
    if (n_points != 0)
    {
        res /= n_points;
    }
    return res;
}

int main (int argc, char *argv[])
{
    parseCommandLine (argc, argv);
    pcl::PointCloud<PointType>::Ptr scene_original (new pcl::PointCloud<PointType> ());
    pcl::PointCloud<PointType>::Ptr scene_filtered_x (new pcl::PointCloud<PointType> ());
    pcl::PointCloud<PointType>::Ptr scene_filtered_y (new pcl::PointCloud<PointType> ());
    pcl::PointCloud<PointType>::Ptr scene_filtered (new pcl::PointCloud<PointType> ());
    ;
```

---

---

## Algorithmus 8.9 Correspondence Grouping 5 of 13

---

```
pcl::PointCloud<PointType>::Ptr scene_voxel (new pcl::PointCloud<PointType> ());
pcl::PointCloud<PointType>::Ptr model (new pcl::PointCloud<PointType> ());
pcl::PointCloud<PointType>::Ptr model_voxel (new pcl::PointCloud<PointType> ());
pcl::PointCloud<PointType>::Ptr model_keypoints (new pcl::PointCloud<PointType> ());
pcl::PointCloud<PointType>::Ptr scene (new pcl::PointCloud<PointType> ());
pcl::PointCloud<PointType>::Ptr scene_keypoints (new pcl::PointCloud<PointType> ());
pcl::PointCloud<NormalType>::Ptr model_normals (new pcl::PointCloud<NormalType> ());
pcl::PointCloud<NormalType>::Ptr scene_normals (new pcl::PointCloud<NormalType> ());
pcl::PointCloud<DescriptorType>::Ptr model_descriptors (new pcl::PointCloud<DescriptorType> ());
pcl::PointCloud<DescriptorType>::Ptr scene_descriptors (new pcl::PointCloud<DescriptorType> ());

//  
// Load clouds  
//  
if (pcl::io::loadPCDFile (model_filename_, *model_voxel) < 0)
//if (pcl::io::loadPLYFile (model_filename_, *model_voxel) < 0)
{
    std::cout << "Error loading model cloud." << std::endl;
    showHelp (argv[0]);
    return (-1);
}  
if (pcl::io::loadPCDFile (scene_filename_, *scene_original) < 0)
//if (pcl::loadPolygonFileVTK (scene_filename_, *scene_original) < 0)
{
    std::cout << "Error loading scene cloud." << std::endl;
    showHelp (argv[0]);
    return (-1);
}

//Z pass through Filter for Scene
pcl::PassThrough <PointType> pass;
pass.setInputCloud(scene_original);
pass.setFilterFieldName("z");
pass.setFilterLimits(0.5, 1.5);
pass.filter(*scene_filtered);

//Statistical Outlier Removal
//Create the filtering object
pcl::StatisticalOutlierRemoval<PointType> statistical_outlier_removal;
statistical_outlier_removal.setInputCloud (scene_filtered);
statistical_outlier_removal.setMeanK (50);
statistical_outlier_removal.setStddevMulThresh (1.0);
statistical_outlier_removal.filter (*scene_voxel);
```

---

---

**Algorithmus 8.10 Correspondence Grouping 6 of 13**

---

```
//  
//Set up resolution invariance  
//  
if (use_cloud_resolution_){  
    float resolution = static_cast<float>(computeCloudResolution(model));  
    if (resolution != 0.0f){  
        {  
            model_ss_ *= resolution;  
            scene_ss_ *= resolution;  
            rf_rad_ *= resolution;  
            descr_rad_ *= resolution;  
            cg_size_ *= resolution;  
        }  
  
        std::cout << "Model resolution: " << resolution << std::endl;  
        std::cout << "Model sampling size: " << model_ss_ << std::endl;  
        std::cout << "Scene sampling size: " << scene_ss_ << std::endl;  
        std::cout << "LRF support radius: " << rf_rad_ << std::endl;  
        std::cout << "SHOT descriptor radius: " << descr_rad_ << std::endl;  
        std::cout << "Clustering bin size: " << cg_size_ << std::endl << std::endl;  
    }  
  
//  
// Downsample Clouds to Extract keypoints  
//  
// Perform the actual filtering  
pcl::VoxelGrid<PointType> sor;  
sor.setInputCloud (model_voxel);  
//sor.setLeafSize(model_ss_,model_ss_,model_ss_);  
sor.setLeafSize (0.001f, 0.001f, 0.001f);  
sor.filter (*model);  
  
pcl::UniformSampling<PointType> uniform_sampling;  
uniform_sampling.setInputCloud (model);  
uniform_sampling.setRadiusSearch (model_ss_);  
//uniform_sampling.filter (*model_keypoints);  
pcl::PointCloud<int> keypointIndices1;  
uniform_sampling.compute(keypointIndices1);  
pcl::copyPointCloud(*model, keypointIndices1.points, *model_keypoints);  
std::cout << "Model total points: " << model->size () << "; Selected Keypoints: "  
     <<  
model_keypoints->size () << std::endl;  
  
sor.setInputCloud (scene_voxel);  
//sor.setLeafSize(scene_ss_,scene_ss_,scene_ss_);  
sor.setLeafSize (0.003f, 0.003f, 0.003f);  
sor.filter (*scene);  
  
uniform_sampling.setInputCloud (scene);  
uniform_sampling.setRadiusSearch (scene_ss_);  
//uniform_sampling.filter (*scene_keypoints);  
pcl::PointCloud<int> keypointIndices2;  
uniform_sampling.compute(keypointIndices2);
```

---

### Algorithmus 8.11 Correspondence Grouping 7 of 13

---

```
pcl::copyPointCloud(*scene, keypointIndices2.points, *scene_keypoints);
std::cout << "Scene total points: " << scene->size () << "; Selected Keypoints: "
<<
scene_keypoints->size () << std::endl;

// 
// Compute Normals
//
pcl::NormalEstimationOMP<PointType, NormalType> norm_est;
//norm_est.setKSearch (10);
norm_est.setInputCloud (model);
// Create an empty kd-tree representation, and pass it to the normal estimation
// object.
// Its content will be filled inside the object, based on the given input dataset
// (as no other
// search surface is given).
pcl::search::KdTree<PointType>::Ptr tree (new pcl::search::KdTree<PointType> ());
norm_est.setSearchMethod (tree);
// Use all neighbors in a sphere of radius 6cm
norm_est.setRadiusSearch(0.06);
norm_est.compute (*model_normals);

norm_est.setInputCloud (scene);
norm_est.compute (*scene_normals);

// 
// Compute Descriptor for keypoints
// pcl::SHOTEstimationOMP<PointType, NormalType, DescriptorType> descr_est;
descr_est.setRadiusSearch (descr_rad_);
descr_est.setInputCloud (model_keypoints);
descr_est.setInputNormals (model_normals);
descr_est.setSearchSurface (model);
descr_est.setSearchMethod(tree);
descr_est.compute (*model_descriptors);

descr_est.setInputCloud (scene_keypoints);
descr_est.setInputNormals (scene_normals);
descr_est.setSearchSurface (scene);
descr_est.setSearchMethod(tree);
descr_est.compute (*scene_descriptors);

// 
// Find Model-Scene Correspondences with KdTree
// pcl::CorrespondencesPtr model_scene_corrs (new pcl::Correspondences ());

pcl::KdTreeFLANN<DescriptorType> match_search;
match_search.setInputCloud (model_descriptors);
// For each scene keypoint descriptor, find nearest neighbor into the model
// keypoints
// descriptor cloud and add it to the correspondences vector.
```

---

---

**Algorithmus 8.12 Correspondence Grouping 8 of 13**

---

```
for (size_t i = 0; i < scene_descriptors->size (); ++i)
{
    std::vector<int> neigh_indices (1);
    std::vector<float> neigh_sqr_dists (1);
    if (!pcl_isfinite (scene_descriptors->at (i).descriptor[0])) //skipping NaNs
    {
        continue;
    }
    int found_neighs = match_search.nearestKSearch (scene_descriptors->at (i), 1,
    neigh_indices, neigh_sqr_dists);
    if(found_neighs == 1 && neigh_sqr_dists[0] < 0.5f) // add match only if the
    squared
        descriptor distance is less than 0.25 (SHOT descriptor distances are between 0
        and 1 by
        design)
    {
        pcl::Correspondence corr (neigh_indices[0], static_cast<int> (i),
        neigh_sqr_dists[0]);
        model_scene_corrs->push_back (corr);
    }
}
std::cout << "Correspondences found: " << model_scene_corrs->size () << std::endl
;

//  

// Actual Clustering  

//  

std::vector<Eigen::Matrix4f, Eigen::aligned_allocator<Eigen::Matrix4f> >
rototranslations;  

std::vector<pcl::Correspondences> clustered_corrs;  

// Using Hough3D  

if (use_hough_)
{
    //  

    // Compute (Keypoints) Reference Frames only for Hough
    //  

    std::cout << "Computing Reference Frames..." << std::endl;
    pcl::PointCloud<RFType>::Ptr model_rf (new pcl::PointCloud<RFType> ());
    pcl::PointCloud<RFType>::Ptr scene_rf (new pcl::PointCloud<RFType> ());

    // BOARD RF estimation object
    pcl::BOARDLocalReferenceFrameEstimation<PointType, NormalType, RFType> rf_est;
    // Search radius (maximum distance of the points used to estimate the X and Y
    axes
    // of the BOARD Reference Frame for a given point).
    rf_est.setRadiusSearch (rf_rad_);
    // Check if support is complete, or has missing regions because it is too close
    to mesh
    //borders.
    rf_est.setFindHoles (false);
    rf_est.setInputCloud (model_keypoints);
    rf_est.setInputNormals (model_normals);
    rf_est.setSearchSurface (model);
```

---

---

**Algorithmus 8.13 Correspondence Grouping 9 of 13**

---

```
rf_est.compute (*model_rf);
rf_est.setInputCloud (scene_keypoints);
rf_est.setInputNormals (scene_normals);
rf_est.setSearchSurface (scene);
rf_est.compute (*scene_rf);

// Clustering
std::cout << "Clustering..." << std::endl;
pcl::Hough3DGrouping<PointType, PointType, RFType, RFType> clusterer;
//Size of each bin in the Hough space.
clusterer.setHoughBinSize (cg_size_);
//clusterer.setHoughBinSize (3);
//Minimum cluster size. Default is 3(as at least 3 correspondences are
needed to compute 6
//DOF pose).
clusterer.setHoughThreshold (cg_thresh_);
//clusterer.setHoughThreshold (3);
//If true, the vote casting procedure will interpolate the score between
neighbouring
//bins in the Hough Space.
clusterer.setUseInterpolation (true);
//If true, the vote casting procedure will use the correspondence's weighted
distance to
compute the Hough voting score.
clusterer.setUseDistanceWeight (false);

clusterer.setInputCloud (model_keypoints);
clusterer.setInputRf (model_rf);
clusterer.setSceneCloud (scene_keypoints);
clusterer.setSceneRf (scene_rf);
clusterer.setModelSceneCorrespondences (model_scene_corrs);
clusterer.cluster (clustered_corrs);
clusterer.recognize (rototranslations, clustered_corrs);
}

else // Using GeometricConsistency
{
    std::cout << "Using GeometricConsistency..." << std::endl;
    pcl::GeometricConsistencyGrouping<PointType, PointType> gc_clusterer;
    gc_clusterer.setGCSIZE (cg_size_);
    gc_clusterer.setGCThreshold (cg_thresh_);

    gc_clusterer.setInputCloud (model_keypoints);
    gc_clusterer.setSceneCloud (scene_keypoints);
    gc_clusterer.setModelSceneCorrespondences (model_scene_corrs);
    //gc_clusterer.cluster (clustered_corrs);
    gc_clusterer.recognize (rototranslations, clustered_corrs);
}

/** 07.08.16.
 */
* Stop if no instances
*/
```

---

---

**Algorithmus 8.14 Correspondence Grouping 10 of 13**

---

```
if (rototranslations.size () <= 0)
{
    cout << "*** No instances found! ***" << endl;
    return (0);
}
else
{
    cout << "Recognized Instances: " << rototranslations.size () << endl << endl;
}

/***
 * Generates clouds for each instances found
 */
std::vector<pcl::PointCloud<PointType>::ConstPtr> instances;

for (size_t i = 0; i < rototranslations.size (); ++i)
{
    pcl::PointCloud<PointType>::Ptr rotated_model (new pcl::PointCloud<PointType> ());
    pcl::transformPointCloud (*model, *rotated_model, rototranslations[i]);
    instances.push_back (rotated_model);
}

// ICP

std::vector<pcl::PointCloud<PointType>::ConstPtr> registered_instances;
if (true)
{
    cout << "---- ICP -----" << endl;

    for (size_t i = 0; i < rototranslations.size (); ++i)
    {
        pcl::IterativeClosestPoint<PointType, PointType> icp;
        icp.setMaximumIterations (icp_max_iter_);
        icp.setMaxCorrespondenceDistance (icp_corr_distance_);
        icp.setInputTarget (scene);
        icp.setInputSource (instances[i]);
        pcl::PointCloud<PointType>::Ptr registered (new pcl::PointCloud<PointType> ());
        icp.align (*registered);
        registered_instances.push_back (registered);
        cout << "Instance " << i << " ";
        if (icp.hasConverged ())
        {
            cout << "Aligned!" << endl;
        }
        else
        {
            cout << "Not Aligned!" << endl;
        }
    }
    cout << "-----" << endl << endl;
}
```

---

---

## Algorithmus 8.15 Correspondence Grouping 11 of 13

---

```
//  
//Output results  
//  
std::cout << "Model instances found: " << rototranslations.size () << std::endl;  
for (size_t i = 0; i < rototranslations.size (); ++i)  
{  
    std::cout << "\n Instance " << i + 1 << ":" << std::endl;  
    std::cout << "Correspondences belonging to this instance:" << clustered_corrs[i].  
        size ()<<  
    std::endl;  
  
    // Print the rotation matrix and translation vector  
    Eigen::Matrix3f rotation = rototranslations[i].block<3,3>(0, 0);  
    Eigen::Vector3f translation = rototranslations[i].block<3,1>(0, 3);  
  
    printf ("\n");  
    printf (" | %6.3f %6.3f %6.3f | \n", rotation (0,0), rotation (0,1),  
           rotation  
(0,2));  
    printf (" R = | %6.3f %6.3f %6.3f | \n", rotation (1,0), rotation (1,1),  
           rotation  
(1,2));  
    printf (" | %6.3f %6.3f %6.3f | \n", rotation (2,0), rotation (2,1),  
           rotation  
(2,2));  
    printf ("\n");    printf (" t = < %0.3f, %0.3f, %0.3f >\n", translation (0)  
        ,  
        translation (1), translation (2));  
}  
  
//  
//Visualization  
//  
pcl::visualization::PCLVisualizer viewer ("Correspondence Grouping");  
viewer.addPointCloud (scene, "scene_cloud");  
  
pcl::PointCloud<PointType>::Ptr off_scene_model (new pcl::PointCloud<PointType>  
());  
pcl::PointCloud<PointType>::Ptr off_scene_model_keypoints (new pcl::PointCloud<  
PointType> ());  
  
if (show_correspondences_ || show_keypoints_)  
{  
    //We are translating the model so that it doesn't end in the middle of the  
    //scene  
    //representation  
    pcl::transformPointCloud (*model, *off_scene_model, Eigen::Vector3f (-1,0,0),  
                           Eigen::Quaternionf (1, 0, 0, 0));  
    pcl::transformPointCloud (*model_keypoints, *off_scene_model_keypoints,  
                           Eigen::Vector3f(-1,0,0), Eigen::Quaternionf (1, 0, 0, 0));
```

---

---

**Algorithmus 8.16 Correspondence Grouping 12 of 13**

---

```
pcl::visualization::PointCloudColorHandlerCustom<PointType>
    off_scene_model_color_handler
(off_scene_model, 255, 255, 128);
viewer.addPointCloud (off_scene_model, off_scene_model_color_handler, "
    off_scene_model");
}

if (show_keypoints_)
{
    pcl::visualization::PointCloudColorHandlerCustom<PointType>
        scene_keypoints_color_handler
(scene_keypoints, 0, 0, 255);
viewer.addPointCloud (scene_keypoints, scene_keypoints_color_handler, "
    scene_keypoints"); viewer.setPointCloudRenderingProperties (pcl::
        visualization::PCL_VISUALIZER_POINT_SIZE, 5,
"scene_keypoints");

pcl::visualization::PointCloudColorHandlerCustom<PointType>
off_scene_model_keypoints_color_handler (off_scene_model_keypoints, 0, 0, 255);
viewer.addPointCloud (off_scene_model_keypoints,
    off_scene_model_keypoints_color_handler,
"off_scene_model_keypoints");
viewer.setPointCloudRenderingProperties (pcl::visualization::
    PCL_VISUALIZER_POINT_SIZE, 5,
"off_scene_model_keypoints");
}

for (size_t i = 0; i < rototranslations.size (); ++i)
{
    pcl::PointCloud<PointType>::Ptr rotated_model (new pcl::PointCloud<PointType>
        ());
    pcl::transformPointCloud (*model, *rotated_model, rototranslations[i]);

    std::stringstream ss_cloud;
    ss_cloud << "instance" << i;

    pcl::visualization::PointCloudColorHandlerCustom<PointType>
        rotated_model_color_handler
(rotated_model, 255, 0, 0);
viewer.addPointCloud (rotated_model, rotated_model_color_handler, ss_cloud.
    str ());

if (show_correspondences_)
{
    for (size_t j = 0; j < clustered_corrs[i].size (); ++j)
    {
        std::stringstream ss_line;
        ss_line << "correspondence_line" << i << "_" << j;
        PointType& model_point = off_scene_model_keypoints->at (clustered_corrs[i]
[j].index_query);
        PointType& scene_point = scene_keypoints->at (clustered_corrs[i][j].
            index_match);
```

---

---

**Algorithmus 8.17 Correspondence Grouping 13 of 13**

---

```
//We are drawing a line for each pair of clustered correspondences found between
//the
//model and the scene
    viewer.addLine<PointType, PointType> (model_point, scene_point, 0, 255, 0,
        ss_line.str
    ());
}
}

while (!viewer.wasStopped ())
{
    viewer.spinOnce ();
}

return (0);
}
```

---

# Bibliography

- [1] M. G. Zanuttigh Pietro, et al., *Time-of-Flight and Structured Light Depth Cameras*. Springer International Publishing, 1. ed., 2016.
- [2] D. Diakopoulos, “Experimental releases from the Intel® RealSense™ lab,” *Github, IntelRealSense, Projection.md*, vol. 1, July 2016.
- [3] H. Raudies, “Optisches Erkennungssystem für Spielfiguren der Eurobot,” *Bachelorarbeit, Hochschule Mannheim, Fakultät Informatik*, pp. 33 – 36, Mar. 2011.
- [4] H. Raudies, “Optisches Erkennungssystem für Spielfiguren der Eurobot,” *Bachelorarbeit, Hochschule Mannheim, Fakultät Informatik*, vol. 1, pp. 36 – 52, Mar. 2011.
- [5] A. Aitor, et al., “Point Cloud Library: Three-Dimensional Object Recognition and 6 DOF Pose Estimation,” *IEEE Robotics and Automation Magazine*, vol. 19, pp. 80 – 91, Sept. 2012.
- [6] L. D. S. Samuele Salti, Tombari Federico, “SHOT: Unique signatures of histograms for surface and texture description,” *Computer Vision and Image Processing*, vol. 125, pp. 251 – 264, May 2014.
- [7] D.-S. L. V. M. Aldoma A., Tombari F., “A global hypothesis verification method for 3D object recognition,” *Computer Vision ECCV 2012*, vol. 125, pp. 511–524, Oct. 2014.
- [8] M. Probst Tom, “Getting started with the Depth Data provided by Intel® RealSense™ Technology,” *Software.Intel.articles*, Dec. 2015.
- [9] Microsoft, “Basic 3D graphics, Field of View,” *Microsoft Developer Network, Field of View*, July 2016.
- [10] A. Klein, “Feature based 3D Object Recognition and Localization in Point Clouds,” *Bachelorarbeit, Hochschule Mannheim, Fakultät Informatik*, pp. 14 – 17, July 2012.

# List of Figures

2.1	Distance Measuring Methods[1] . . . . .	3
2.2	Intel Realsense R200[1] . . . . .	4
2.3	R200 Components[1] . . . . .	4
2.4	Szene . . . . .	5
2.5	IR Pattern Zoomed & IR Pattern[1] . . . . .	6
4.1	Schachbrettmuster . . . . .	11
4.2	ROS Calibration Window . . . . .	12
4.3	Interneparameter . . . . .	13
4.4	Relative Position[2] . . . . .	14
4.5	measurement_test . . . . .	15
4.6	measurement_tool . . . . .	16
4.7	Experiment Setup & Analysis[8] . . . . .	17
4.8	Norminalised Error[8] . . . . .	18
5.1	PointCloud . . . . .	20
5.2	Voxel Grid[10] . . . . .	21
5.3	Field of View[9] . . . . .	22
5.4	Scene Euclidean Clustering . . . . .	24
5.5	Scene Clusters . . . . .	24
5.6	Signature v/s Histogramm[8] . . . . .	26
5.7	Normal Estimation . . . . .	27
5.8	Signature Structure[6] . . . . .	28
5.9	ICP[11] . . . . .	30
5.10	Precision & Recall Test[6] . . . . .	31
6.1	Message Types . . . . .	33
6.2	Overall Pipeline . . . . .	34
6.3	Clustering Pipeline . . . . .	35
6.4	Matching . . . . .	36
7.1	Chair Scene . . . . .	39
7.2	Correspondences . . . . .	40
7.3	Mug Scene . . . . .	41
7.4	Mug Cluster . . . . .	42

7.5 Mug Mesh . . . . .	42
------------------------	----

# Algorithmenverzeichnis

4.1	Resolution Settings . . . . .	10
4.2	Camera Calibrator . . . . .	11
5.1	SHOT Descriptor . . . . .	29
8.1	Voxel Filter . . . . .	45
8.2	Euclidean Clustering 1 of 3 . . . . .	46
8.3	Euclidean Clustering 2 of 3 . . . . .	47
8.4	Euclidean Clustering 3 of 3 . . . . .	48
8.5	Correspondence Grouping 1 of 13 . . . . .	49
8.6	Correspondence Grouping 2 of 13 . . . . .	50
8.7	Correspondence Grouping 3 of 13 . . . . .	51
8.8	Correspondence Grouping 4 of 13 . . . . .	52
8.9	Correspondence Grouping 5 of 13 . . . . .	53
8.10	Correspondence Grouping 6 of 13 . . . . .	54
8.11	Correspondence Grouping 7 of 13 . . . . .	55
8.12	Correspondence Grouping 8 of 13 . . . . .	56
8.13	Correspondence Grouping 9 of 13 . . . . .	57
8.14	Correspondence Grouping 10 of 13 . . . . .	58
8.15	Correspondence Grouping 11 of 13 . . . . .	59
8.16	Correspondence Grouping 12 of 13 . . . . .	60
8.17	Correspondence Grouping 13 of 13 . . . . .	61