

# Navigation einer Flugdrohne anhand eines 3D-Umweltmodells abgeleitet aus konsekutiven 2D-Bildaufnahmen und Sensordaten

## STUDIENARBEIT

5. & 6. Semester

des Studiengangs Angewandte Informatik  
an der  
Dualen Hochschule Baden-Württemberg Karlsruhe

von  
Christian Verdion und Daniel Betsche

Bearbeitungszeitraum 06.07.2015 - 6.06.2016  
Matrikelnummer 5698770 7583051  
Kurs TINF13B2  
Ausbildungsfirma Fiducia & GAD IT AG cjt Systemsoftware AG  
Karlsruhe  
Betreuer Prof. Dr. Marcus Strand  
Abgabedatum 6.06.2016

## Copyrightvermerk

Dieses Werk einschließlich seiner Teile ist **urheberrechtlich geschützt**. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Autors unzulässig und strafbar. Das gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen sowie die Einspeicherung und Verarbeitung in elektronischen Systemen.

## Erklärung

Erklärung gemäß § 5 (3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 22. September 2011.

Ich habe die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

---

Ort, Datum

---

Unterschrift

---

Ort, Datum

---

Unterschrift

## Abstract

Die Welt der Robotik bewegt sich massiv in Richtung vollständiger Autonomie. Roboter werden zunehmend selbstständiger und sind dadurch in der Lage Aufgaben zu übernehmen, welche bislang als zu komplex galten. Klassischen Robotern wurden Bewegungsabläufe fest einprogrammiert und eine Abweichung von ihrem Programm war nicht möglich. Moderne Systeme allerdings verlassen sich auf immer ausgereiftere Sensorik und können dynamisch auf Veränderungen reagieren.

Aktuell liegen fliegende Drohnen wie Quadro-, Hexa- und Octocopter immer mehr im Trend. Aufgrund ihrer Geschwindigkeit und ihres relativ geringen Kostenfaktors sind sie beliebt in privatem, professionellen und wissenschaftlichem Einsatz. In privaten Haushalten finden sie meist Verwendung als Spielzeug für Modellflieger. Im professionellen Einsatz werden sie genutzt für Kameraaufnahmen bei Großveranstaltungen und Filmen oder um Transporte durchzuführen. Für den Wissenschaftlichen Bereich eignen Sie sich hervorragend, da sie von Haus aus mit einer Vielzahl von Sensoren ausgestattet sind und weiter aufgerüstet werden können.

Viele dieser Flugdrohnen nutzen bereits eingebaute Bildverarbeitung um einfache Ziele identifizieren zu können. Je nach Konfiguration sind sie in der Lage ihre Ziele mit der eingebauten Sensorik analysieren und vermessen zu können und damit dazu befähigt ihnen zu folgen oder auszuweichen. Die vorliegende Studienarbeit setzt an diesem Punkt an und setzt auf die Erweiterung der Aufnahme und Verarbeitung von Bildern im dreidimensionalen Raum. Dies verspricht neben simultaner Lokalisierung und Kartografie auch Verbesserungen in der Erkennung von Objekten und Hindernissen.

Im Rahmen von zwei Testszenarien soll diese Fähigkeit exemplarisch vorgeführt werden.

# Inhaltsverzeichnis

<b>1 Einführung und Motivation</b>	<b>1</b>
1.1 Aufgabenstellung und Zielsetzung . . . . .	2
1.2 Kontextabgrenzung . . . . .	2
1.3 Schnittstellendefinition . . . . .	3
1.4 Aktuelle Situation und Forschung . . . . .	4
<b>2 Grundlegende Konzepte und Technologien</b>	<b>5</b>
2.1 Die Drohne . . . . .	5
2.1.1 Parrot AR-Drone 2.0 . . . . .	5
2.1.2 AR-Drone SDK von Parrot . . . . .	6
2.1.3 AR Drone Modul für NodeJS . . . . .	7
2.1.4 AR Drone Autonomy . . . . .	8
2.2 Digitale Bildverarbeitung . . . . .	9
2.2.1 Die Kamera . . . . .	9
2.2.2 Verzeichnung . . . . .	10
2.2.3 Intrinsische und extrinsische Parameter . . . . .	11
2.2.4 Stereoskopie . . . . .	12
2.2.5 Organisation der 3D-Daten . . . . .	14
2.2.6 Point Clouds . . . . .	15
2.3 Entwicklungsumgebung . . . . .	17
2.3.1 JavaScript . . . . .	17
2.3.2 Node JS . . . . .	18
2.3.3 C++ . . . . .	18
2.3.4 ROS . . . . .	19
2.3.5 Point Cloud Library . . . . .	25
2.4 Arbeiten mit Simulationen . . . . .	26
2.4.1 Warum Simulieren . . . . .	26
2.4.2 Relevante Parameter . . . . .	27
2.4.3 Rviz . . . . .	27
2.4.4 Gazebo . . . . .	28
<b>3 Architekturübersicht</b>	<b>31</b>
3.1 Reale Drohne in Kombination mit Rubbertrion . . . . .	31
3.2 Virtualisierte Lösung . . . . .	32
3.3 Aufbau der ROS-Pakete . . . . .	33
<b>4 Implementierung der Simulation</b>	<b>35</b>

4.1	TU Darmstadt - Hector Quadrotor . . . . .	35
4.2	TU München - Simulator . . . . .	37
4.3	Modellierung von Welt und Drohne . . . . .	38
4.4	Gazebo Plugins . . . . .	40
4.4.1	Controller . . . . .	41
4.4.2	Sensoren . . . . .	42
<b>5</b>	<b>Aufnahme und Verarbeitung der Kameradaten</b>	<b>44</b>
5.1	Physische Drohne . . . . .	44
5.1.1	Zusammenarbeit mit der Firma Myestro . . . . .	44
5.2	Simulation . . . . .	45
<b>6</b>	<b>Aufstellen und Auswerten eines 3D-Umweltmodells</b>	<b>50</b>
6.1	Vorverarbeitung der Tiefendaten . . . . .	50
6.2	Zusammensetzung der 3D Ausschnitte zu einer Szene . . . . .	51
6.2.1	Iterative Closest Point Algorithmus . . . . .	52
6.3	Objekterkennung im drei Dimensionalen Raum . . . . .	54
<b>7</b>	<b>Autonome Steuerung</b>	<b>57</b>
7.1	Einer Säule ausweichen . . . . .	58
7.2	Eine Tür durchqueren . . . . .	58
<b>8</b>	<b>Ergebnis und Ausblick</b>	<b>61</b>
8.1	Fazit . . . . .	61
8.2	Ausblick . . . . .	61
<b>9</b>	<b>Literatur</b>	<b>63</b>
<b>10</b>	<b>Anhänge</b>	<b>65</b>

## Abkürzungsverzeichnis

- PCL** Point Cloud Library  
**ROS** Robot Operating System  
**IMU** Inertial Measurement Unit  
**SLAM** Simultaneous Localization and Mapping

## Abbildungsverzeichnis

1	ARDrone 2.0 . . . . .	5
2	Das Licht scheint durch Linse und Blende auf den Sensor [24] . . . . .	10
3	Die verschiedenen Verzerrungen ausgelöst durch die Kameralinse [5] . .	10
4	Aufnahme einer Kalibrierungsmaske mit der Frontkamera der AR-Drone	11
5	Zwei Augen, die eine Haus und ein Baum ansehen [2] . . . . .	14
6	Module und Abhängigkeiten der Point Cloud Library [16] . . . . .	25
7	PointCloud von Myestro Büro in Rviz . . . . .	28
8	Virtualisierte Umgebung mit AR Drone . . . . .	29
9	Architekturskizze mit echter Drohne und Rubbertrion . . . . .	32
10	Vollvirtualisierte Architekturskizze . . . . .	33
11	Baumansicht der Pakethierarchie . . . . .	34
12	Beispiel hector_quadrotor Stack mit SLAM . . . . .	36
13	Beispiel Tum Simulator mit echter und virtualisierter Drohne . . . . .	37
14	Ein verrausches Tiefenbild, das durch Vorverarbeitung bereinigt wurde[17]	51
15	Tiefenbilder von einem Gesicht zusammengesetzt zu einem Ganzen [27]	52
16	Zwei PointClouds mit markierten FeaturePoints [18] . . . . .	53
17	Iteration über die nächstgelegenen Punkte [14] . . . . .	53
18	Tiefenbilder von einem Raum aus verschiedenen Perspektiven [19] . .	54
19	Tiefenbild von einem Raum aus verschiedenen Perspektiven zusam- mengesetzt [20] . . . . .	54
20	Die verschiedenen Arten von Kanten [4] . . . . .	55
21	Markierte Kanten in einem Tiefendatenbild [3] . . . . .	56
22	Schematische Darstellung des Ausweichvorgangs bei Kollisionskurs mit einer Säule . . . . .	58
23	Schematische Darstellung des Durchquerens eines Türrahmens . . . .	59

## Listings

1	Einfache Flugsequenz mit AR Drone Modul . . . . .	7
2	Auslesen der Kamera- und Sensordaten . . . . .	8
3	Der exemplarische Aufbau einer PCD Datei . . . . .	15
4	Der exemplarische Aufbau einer PCD Datei . . . . .	16
5	Launchfile zum Starten einer Drohne . . . . .	20
6	Die coordinate.msg definiert eine Koordinatennachricht . . . . .	22
7	Die spawn.srv für den Service <i>spawn</i> . . . . .	23
8	Ein Beispiel für eine 'package.xml' Datei, die ein ROS Projekt beschreibt	23
9	Eine 'CMakeList.txt' Datei zum Bauen eines ROS Projektes . . . . .	24
10	Grundelemente einer Welt . . . . .	38
11	Definition einer einzelnen Säule . . . . .	38
12	Physikalische Konfiguration der Welt . . . . .	39
13	Drohnenkörper . . . . .	39
14	Include der Plugins . . . . .	41
15	Konfiguration des State-Controllers . . . . .	41
16	Einbinden eines GPS Sensors . . . . .	42
17	Gazebo Sensor Konfiguration . . . . .	46
18	Kameraspezifische Konfiguration . . . . .	46
19	Konfiguration des Sensor-Plugins . . . . .	47
20	Sensoren der Drohne . . . . .	65
21	Controller der Drohne . . . . .	67
22	Vollständiges Launchfile . . . . .	68

# 1 Einführung und Motivation

Die Welt der Robotik befindet sich in einer Phase der Veränderung. Roboter werden zunehmend selbstständiger und verrichten immer neue Aufgaben, welche lange Zeit als zu komplex galten. Die Entwicklung zu autonomen Robotern wird getrieben durch immer bessere Sensoren, bessere Software und billigere Hardware. Ein Feld hat sich besonders stark entwickelt, sodass zum Zeitpunkt der Arbeit eine große Diskussion über die rechtlichen Rahmenbedingungen geführt wird. Die Sprache ist von Flugdrohnen und Multicoptern.

Sie finden eine immer größere Verbreitung nicht nur im professionellen Umfeld, wo bei sie hauptsächlich für Bild- und Videoaufnahmen genutzt werden, sondern auch im privaten Bereich. Dort werden sie meist von Hobbypiloten und Bastlern als Spielzeug genutzt, mit denen sogar Wettrennen durchgeführt werden. Eine immer größere Verwendung finden Drohnen auch in der zivilen Wissenschaft, da sie es deutlich einfacher machen schwierige Messvorgänge durchzuführen. Multicopter besitzen allein um flugfähig zu sein eine Vielzahl an Sensoren und können mit zusätzlicher Ausrüstung aufgestockt werden.

Die Autonomie spielt bei Multicoptern eine besondere Rolle. Damit etwa ein Quadrocopter fliegen kann müssen, alle der vier Rotoren in einem empfindlichen Gleichgewicht gehalten werden. Dieser Ausgleich wurde erst möglich, als leistungsfähigere Mikrocontroller, die Daten von Gyroskopen auslesen und verarbeiten konnten, um die Drehmomente der Motoren auszugleichen. Diese Technologie wurde erst um die Jahrtausendwende herum verfügbar und im Jahre 2004 in Serie produziert.

In diesem Kontext ist es von Bedeutung die verschiedenen Ausprägungen von Autonomie zu unterscheiden. Diese erste Stufe erlaubt einen stabilen Flug zu gewährleisten ohne direkte Kontrolle der Motorleistung. Die nächste Stufe wird erreicht, wenn die Flugdrohne mit weiteren Sensoren wie einem GPS, Höhenmesser und Kompass ausgerüstet wird. Dadurch können Flugrouten geplant und autonom verfolgt werden. Den Zenit bildet der vollautonome Flug. Hierbei wird keine vorprogrammierte Route durchflogen oder vom Menschen eingegriffen, sondern alle Entscheidungen werden anhand einer Zielvorgabe von der Drohne selbst getroffen. Anwendungsfälle für den vollautonomen Flug sind hauptsächlich in den Gebieten der Aufklärung und Überwachung zu finden. Darunter etwa die Erkundung von schwer zugänglichen Orten wie Katastrophengebieten, Aufklärung bei Rettungseinsätzen in schwer zugänglichem Gelände oder für einfachere Aufgaben wie dem Kartieren oder der Beobachtung von Wildtieren.

## 1.1 Aufgabenstellung und Zielsetzung

An dieser Stelle des vollautonomen Flugs setzt die Studienarbeit an, mit der Aufgabe die Erkennung von Objekten und Hindernissen zu vereinfachen und anhand ihrer Umgebung selbstständig Flugrouten planen und verfolgen zu können. Die Drohne soll sich dabei allein auf ihre Sensoren stützen und anhand der ausgewerteten Daten navigieren können. Die Objekte und Hindernisse werden in dieser Arbeit dargestellt durch Säulen und Türrahmen. Die Aufgabe ist es Säulen ausweichen und Türrahmen durchqueren zu können.

Dazu muss im ersten Schritt aus den zweidimensionalen Kameradaten ein dreidimensionales Modell erzeugt werden. Dieser Vorgang wird 3D-Rekonstruktion (3D-Reconstruction) genannt. Für diesen Schritt wurde im Rahmen dieser Studienarbeit eine Kooperation mit der Firma Myestro vereinbart. Myestro arbeitet an moderner Software in den Bereichen der Computer-Vision und Machine-Vision. Dazu gehört auch der Bereich Stereo-Vision, unter dem man das Tiefensehen mit zwei oder mehr Kameras versteht. Handelsübliche Hard- und Software stützt sich in diesem Bereich auf die Verwendung von zwei Kameras mit festem Abstand zueinander. Die Firma Myestro verspricht eine Lösung, die nicht auf diese Vorgabe angewiesen ist, sondern mit konsekutiven Bildern einer einzelnen Kamera ein vergleichbares Ergebnis liefern kann.

Die erhaltenen Punktwolken werden anschließend zu einem dreidimensionalen Modell zusammengesetzt und mit weiteren Sensordaten verknüpft um eine gleichzeitige Lokalisierung und Kartenerstellung (Simultaneous Localization and Mapping (SLAM)) zu realisieren. Anhand der dadurch erhaltenen Daten soll eine Planung der möglichen Flugrouten erfolgen und eine Auswahl getroffen werden.

## 1.2 Kontextabgrenzung

Die Studienarbeit beschränkt sich auf die Erfassung aller Sensordaten der Flugdrohne zur Erkennung von Hindernissen direkt in ihrem Pfad. Es wird angenommen, dass die Frontkamera immer in Bewegungsrichtung der Drohne ausgerichtet ist und Hindernisse sich eindeutig vor dem Quadrocopter und innerhalb des Bildbereichs befinden.

Mögliche Störungen der Flugfähigkeit und unerwartete Einflüsse auf das Flugverhalten der Drohne werden aus Gründen der Aufwandsbeschränkung ausgeschlossen. Das bedeutet, ein Flug wird ausschließlich in geschlossenen Räumen durchgeführt. Wetterbedingte Einflüsse, wie etwa unerwartete Windstöße, werden damit vermieden.

Im Rahmen dieser Studienarbeit betrachten wir alle Daten der Kamerasensoren, des Sonars zur Höhenmessung und der internen Inertial Measurement Unit (IMU) zur Messung von Beschleunigung um den Zustand der Drohne bestimmen zu können. Bei der Aufnahme von homogenen Flächen zur Analyse mit bildverarbeitender Software ist es üblich, zusätzlich zur Kamera einen Laser mit Specklemuster zu nutzen. Die verwendete Drohne besitzt keinen solchen Laser, in der im Handel erhältlichen Ausführungen. Es ist möglich einen solchen Laser an die Drohne anzubringen, allerdings sind dafür weitere Modifikationen notwendig. Der Quadrocopter soll jedoch ohne zusätzliche Anpassungen genutzt werden. Zur Vereinfachung wird davon ausgegangen, dass jede aufgenommene Fläche mit einem heterogenen Muster, wie einem Punktemuster (siehe Abbildung 4), versehen ist, um die spätere Erkennung zu erleichtern.

### 1.3 Schnittstellendefinition

Um die Zusammenarbeit mit der Firma Myestro möglichst reibungslos zu gestalten, gilt es eine klare Definition der Schnittstellen für beide Parteien festzulegen. Die in der Software von Myestro verwendeten proprietären Algorithmen werden als Hauptprodukte vertrieben und stellen damit das wirtschaftliche Standbein der Firma dar. Daher wurde die Vereinbarung getroffen, dass die Software als geschlossenes Paket, als Blackbox, genutzt wird und im Rahmen der Kooperation Schnittstellen eingebaut werden, um eine einfache Integration zu gewährleisten.

Die Anforderungen von Myestro an die Eingabedaten sind zum einen die aufgenommenen zweidimensionalen Bilder im PNG-Format sowie die dazugehörige intrinsische Korrekturmatrixt zur Entzerrung der Bildaufnahmen. Die Korrekturmatrixt wird dabei im Voraus und in der Zusammenarbeit mit Myestro berechnet und muss nur einmalig zum Start der Software mitgegeben werden.

Das Ausgabeformat der Punktwolken wurde für die Studienarbeit festgelegt auf das PCD Format[21]. Dieses Dateiformat war bereits in die Software implementiert und ist das Standardformat der am weitesten verbreiteten Open Source Softwarebibliothek zur Verarbeitung von PointClouds, der Point Cloud Library. Das Format selbst bietet verschiedene Einstellungen an, welche Informationen enthalten sein sollen und in welchen Datentypen sie abgelegt werden. Nähere Informationen dazu werden unter Paragraph 2.2.6 erläutert. Alle diese Informationen werden in dem Header einer PCL-Datei festgelegt. Es wurde vereinbart, dass für jeden Punkt seine Koordinaten und die Farbe in 8-bit RGB ausgegeben werden. Alle Daten werden in Binärform abgespeichert und das Datenset als organisierte PointCloud hinterlegt. Eine organisierte

PointCloud ist ähnlich einer Matrix in Reihen und Spalten strukturiert. Diese Form unterstützt später die Berechnung der Abstände zweier aufeinanderfolgender Aufnahmen und vereinfacht damit das Zusammensetzen zu einer Szene.

## 1.4 Aktuelle Situation und Forschung

Die Besonderheit der Arbeit liegt in Anforderung die Rekonstruktion von dreidimensionalen Modellen aus konsekutiven zweidimensionalen Bildaufnahmen mit einer Geschwindigkeit und Genauigkeit durchführen zu können, sodass eine Flugdrohne fähig ist, mit möglichst geringer Verzögerung auf ihre Umgebung reagieren zu können. Die ersten veröffentlichten Algorithmen zur dreidimensionalen Rekonstruktion gibt es seit 1941, zu finden in dem Werk von Josef Krames „Zur Ermittlung eines Objektes aus zwei Perspektiven“ [15]. Zwar ist die „Produktion von 3D-Modellen ein beliebtes Forschungssgebiet für eine lange Zeit gewesen, und wichtige Fortschritte wurden erzielt [...]“[22], allerdings liegt auch heute noch der Fokus auf Genauigkeit, Rechenaufwand und der Schwierigkeit von nicht kalibrierten Bildfolgen.

Da die Rekonstruktion vollständig durch die Software von Myestro übernommen wird und keine Details zur Implementierung bekannt gegeben werden, ist ein Vergleich mit aktuellen Verfahren nicht ausreichend möglich.

Der Fokus der Arbeit selbst liegt auf der Implementierung der Objekterkennung durch Kantenextraktion und einer daraus resultierenden autonomen Steuerung. Zu diesem Thema konnten keine vergleichbaren Arbeiten gefunden werden, welche mit dreidimensionalen Modellen arbeiten. Aktuelle Projekte setzen auf Mmrkerbasiertes Tracking. So auch ein Projekt der TU München, welches sich mit Kamerabasierter Navigation auseinandersetzt und auf GitHub veröffentlicht wurde unter [github.com/tum-vision/tum\\_ardrone](https://github.com/tum-vision/tum_ardrone).

## 2 Grundlegende Konzepte und Technologien

### 2.1 Die Drohne

DANIEL BETSCHE

Das folgende Unterkapitel enthält Informationen über die in der Studienarbeit genutzte Drohne. Es wird sowohl die Hardware als auch Software vom Hersteller abgedeckt sowie die verfügbaren OpenSource-Bibliotheken zur Modifikation und Steuerung.

#### 2.1.1 Parrot AR-Drone 2.0

Die Parrot AR-Drone 2.0 ist das Nachfolgermodell der AR-Drone 1.0 und ist dieser in einigen technischen Punkten wie Flugleistung, Kameraqualität und Steuerungssensibilität überlegen. Eine Gegenüberstellung der beiden Drohnen ist nicht Teil dieser Arbeit. Gründe für die Verwendung dieser Drohne sind zum einen der modulare Aufbau, so dass jedes Einzelteil der Drohne ohne viel Aufwand ausgetauscht werden kann. Zu anderen kann die Drohne sowohl über eine API gesteuert werden, aber auch mit einem eigenen Betriebssystem ausgestattet werden. Im Zuge dieser Arbeit ist der Funktionsumfang der Originalsoftware ausreichend, sodass kein eigenes Betriebssystem geschrieben werden muss.



Abbildung 1: ARDrone 2.0  
Quelle: [parrot.com/de/produkte/ardrone-2/](http://parrot.com/de/produkte/ardrone-2/)

Die Drohne verfügt unter anderem über zwei Kameras. Die eine ist an der Unterseite angebracht, sodass der Untergrund gefilmt werden kann. Die andere ist nach vorne gerichtet. Des Weiteren sind sowohl ein Ultraschall Emitter als auch ein Ultraschall Sensor verbaut, mit deren Hilfe auf die Flughöhe geschlossen werden kann. Die Kommunikation mit der Steuereinheit erfolgt über Wireless LAN. Die Drohne erstellt ein Netzwerk, mit dem sich die Steuereinheit verbindet. Über diesen Übertragungskanal können die Daten der Sensoren übertragen werden und der Drohne Steuerbefehle gesendet werden.

### **2.1.2 AR-Drone SDK von Parrot**

Die Firma Parrot bezeichnet sich selbst als einen Hersteller fortschrittlicher Technologieprodukte für Endverbraucher im Bereich Smartphones und Tablets. Daneben produziert Parrot auch eine Reihe zivilen Wasser-, Land- und Flugdrohnen, welche dafür entworfen wurden im Hobbybereich und als Spielzeuge genutzt zu werden. Zu diesem Zweck hat Parrot nicht nur selbst Apps für Smartphone und Tablet zur Steuerung ihrer Drohnen entwickelt, sondern stellt auch ein SDK für Entwickler bereit. Nach Angaben des Herstellers, ist das SDK dazu gedacht, um eigenen Apps und Spiele rund um die Drohnen zu erstellen.

Das von Parrot bereitgestellte SDK ist unter dem Entwicklerbereich der Herstellerwebseite [1] zu finden. Es enthält eine allgemeine Anleitung für Entwickler, wie das SDK zu nutzen ist, sowie Bibliotheken zur Ansteuerung und entsprechende Beispiele für alle unterstützten Plattformen. Unterstützt werden Anwendungen auf Linux und Windows Rechnern sowie auf iOS und Android-Geräten. Das SDK liefert lediglich Anleitungen um externe Anwendungen zu schreiben. Es wird ausdrücklich darauf hingewiesen, dass das Schreiben von Embedded Software nicht unterstützt wird, weil der direkte Zugriff auf die Hardware nicht erlaubt ist.

Die Vielfalt an unterstützten Endgeräten ist möglich durch die Ansteuerung der Drohne über TCP/UDP/IP-Stacks welche, von dem Quadrocopter selbst in seinem eigenen WLAN-Netz erstellt werden. Aufgrund der reinen Kommunikation von Drohne und Kontrolleinheit über Netzwerkprotokolle kann die Steuerung rein theoretisch in jeder Programmiersprache auf jedem Gerät mit einer WLAN fähigen Netzwerkkarte realisiert werden. Dabei entsteht lediglich ein Mehraufwand, um die von Parrot entwickelten Protokolle zur Kommunikation mit der Drohne nachzubilden.

### 2.1.3 AR Drone Modul für NodeJS

Das AR Drone Modul ist ein Beispiel für eine solche Eigenimplementierung der AR Drone Bibliotheken. Das Modul wurde hauptsächlich von Felix Geisendorfer [13] geschrieben und ist zur Entwicklung von JavaScript-Anwendungen für die AR Drone 2.0 konzipiert worden. Es stellt die Grundlage dar für alle Anwendungen, welche im Rahmen der OpenSource-Community nodecopter.com [12] entwickelt wurden. Alle Anwendungen und Module der Community wurden in JavaScript geschrieben und benötigen einen NodeJS-Server zur Ausführung.

**NodeJS** ist eine Open Source und Multi-Plattform Laufzeitumgebung für serverseitige Webanwendungen. Viele seiner Module sind in JavaScript geschrieben und werden zur Laufzeit mit Google's JavaScript V8 Engine interpretiert. NodeJS eignet sich aufgrund seiner ereignisgesteuerten Architektur besonders gut für asynchrone I/O Anwendungen. Das Steuern einer Drohne und Auswerten ihrer Sensordaten ist eine solche Anwendung.

Das AR Drone Modul befindet sich noch in der Entwicklung, unterstützt aber fast alle Funktionen, welche auch von den Bibliotheken des AR Drone SDKs angeboten werden. Es wurde besonders Wert gelegt auf die Abstraktion der Befehle auf eine hohe an die menschliche Sprache angelehnte Ebene. Als Beispiel dient folgender Abschnitt Code einer einfachen Flugsequenz.

Listing 1: Einfache Flugsequenz mit AR Drone Modul

```

1 var arDrone = require('ar-drone');
2 var client = arDrone.createClient();
3
4 client.takeoff();
5
6 client
7   .after(5000, function() {
8     this.clockwise(0.5);
9   })
10  .after(3000, function() {
11    this.stop();
12    this.land();
13  });

```

Zuerst wird eine Instanz von arDrone aus dem Modul erzeugt. Damit wird eine Verbindung mit dem Quadrocopter hergestellt und bei Erfolg ein Client-Objekt erzeugt. Auf dem Client lassen sich von nun an Steuerbefehle ausführen. In dem oben stehenden Ausschnitt wird die Drohne gestartet, dreht sich nach 5 Sekunden für 3 Sekunden mit halber Geschwindigkeit mit dem Uhrzeigersinn, beendet dann alle aktuellen Bewegungen und landet.

Neben der Steuerung ist auch das Auslesen von Sensor- und Kameradaten ein elementarer Bestandteil der API und kann wie folgt realisiert werden.

Listing 2: Auslesen der Kamera- und Sensordaten

```

1 // Ausgeben von Sensordaten auf die Standardausgabe
2 client.on('navdata', console.log);
3
4 // Ausgeben von Kamerabildern auf die Standardausgabe
5 var pngStream = client.getPngStream();
6 pngStream.on('data', console.log);
7
8 var videoStream = client.getVideoStream();

```

Wie man aus den obigen Codebeispielen erkennen kann, ist nur eine geringe Einarbeitungszeit in das Modul nötig um einfache Flugsequenzen zu ermöglichen. Das Modul eignet sich vor allem hervorragend in Kombination mit Webanwendungen um die Steuerung zu implementieren und gleichzeitig die empfangenen Daten anschaulich darstellen zu können. Da die Drohne allerdings ein eigenes WLAN-Netz aufbaut, zu welchem sich der Controller verbinden muss, muss auch der genutzte NodeJS-Server lokal betrieben werden.

### 2.1.4 AR Drone Autonomy

AR Drone Autonomy basiert auf dem originalen SDK von Parrot um die AR Drone 1.0 und 2.0 anzusteuern. Alle Funktionen des SDK sind dabei für ROS angepasste Funktionen verpackt. Die Steuerung erfolgt wie alle Funktionen in ROS über den internen Message Passing Service. Alle Nachrichten werden auf einen zentralen Bus gelegt und von den Paketen abgefangen, welche sich für die verschickten Nachrichten interessieren. Dies erlaubt nicht nur eine optimale Modularisierung der Software, sondern auch das gleichzeitige Ansteuern von unterschiedlichen Aus- und Eingabeformaten. Somit kann also eine Simulation aufgesetzt werden, welche das Verhalten der Drohne dar-

stellt und dennoch die echte Drohne gleichzeitig gesteuert wird. Dabei unterscheidet sich die Steuerungssoftware nicht, es kommt lediglich ein Empfänger der Nachrichten hinzu. Im Verlauf dieser Studienarbeit wird hauptsächlich die Simulation verwendet, da hier das Verhalten ohne Risiken getestet und auch in kürzester Zeit angepasst werden kann.

## **2.2 Digitale Bildverarbeitung**

CHRISTIAN VERDION

In diesem Unterkapitel wird zunächst auf die Kameras der Drohne eingegangen. Darauf folgend wird der Vorgang untersucht, aus zweidimensionalen Bildern Tiefendaten zu extrahieren.

### **2.2.1 Die Kamera**

Die AR-Drone verfügt über zwei Kameras. Da sich diese stark voneinander unterscheiden, werden zuerst deren technische Details einander gegenübergestellt werden, um so zu entscheiden, auf welche Kamera sich in der Arbeit gestützt wird.

Die nach unten gerichtete Kamera nimmt 60 Bilder pro Sekunde auf. Die Auflösung dabei beträgt allerdings nur QVGA, was 320x240 Pixeln entspricht. Die Drohne selbst verwendet diese Kamera um die Fluggeschwindigkeit zu bestimmten. Ursprünglich sollte mithilfe dieser Kamera ein dreidimensionales Raster des überflogenen Untergrunds erstellen. Aufgrund der geringen Auflösung hat sich dies allerdings als nicht praktikabel herausgestellt, da die Drohne auch nicht in ihren Hardwarespezifikationen verändert werden sollte.

Die nach vorne gerichtete Kamera nimmt 30 Bilder pro Sekunde auf, hat dafür allerdings eine deutlich bessere Auflösung. Diese beträgt 1280x720. Mit dieser Kamera können ausreichend detaillierte Fotos erstellt werden, um daraus Tiefendaten zu extrahieren. Die niedrigere Bildwiederholfrequenz der Kamera hat keinen Einfluss auf das Ergebnis, da zur Berechnung der Tiefendaten die Bilder in einem höheren Intervall betrachtet werden. Weitere technische Details hierfür werden im Kapitel Stereoskopie behandelt.

## 2.2.2 Verzeichnung

In den meisten gängigen Kameras befinden sich Linsen, um das Licht auf einen Sensor zu fokussieren. Diese Kameralinsen bringen aufgrund ihrer physischen Beschaffenheit zwangsläufig auch eine Verzeichnung des Bildes mit sich. Siehe Abbildung 2.

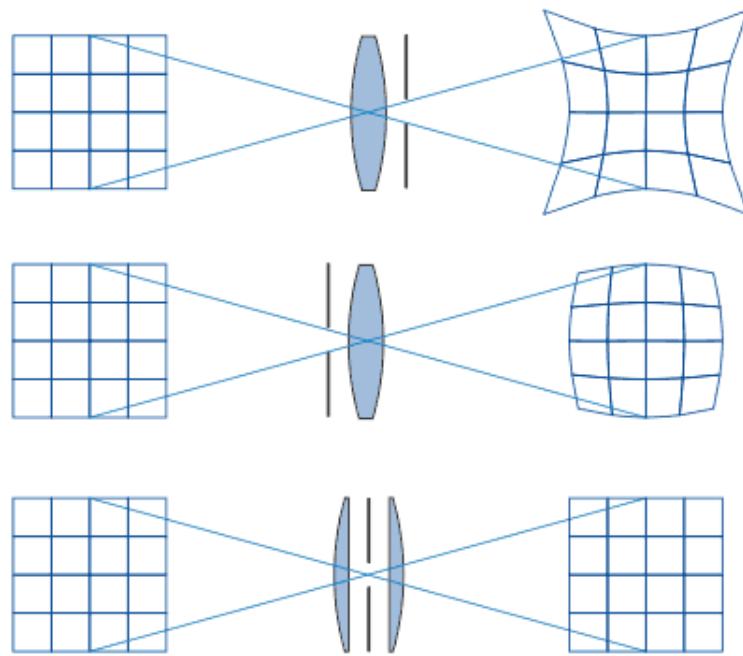


Abbildung 2: Das Licht scheint durch Linse und Blende auf den Sensor [24]

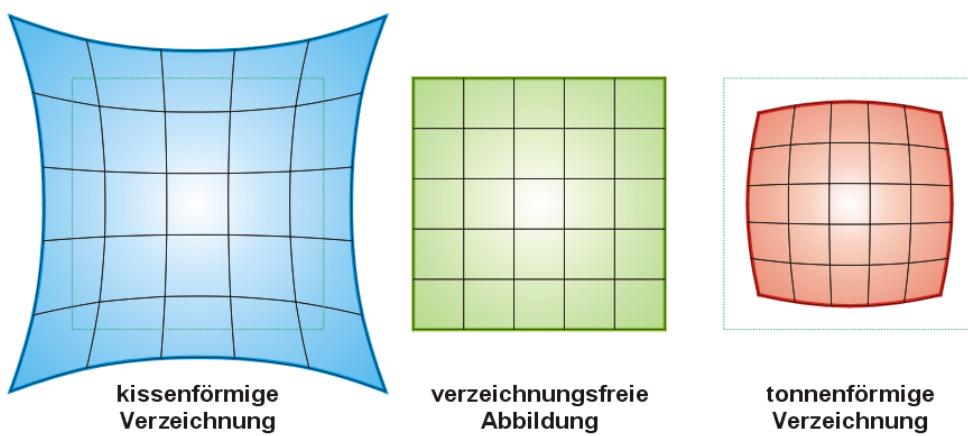


Abbildung 3: Die verschiedenen Verzerrungen ausgelöst durch die Kameralinse [5]



Abbildung 4: Aufnahme einer Kalibrierungsmaske mit der Frontkamera der AR-Drone

Gerade bei einfacheren Objektiven ist das im Bereich von nahen Motiven bis etwa drei Meter sehr ausgeprägt. Je größer der Abstand vom Objektiv zum Motiv ist, desto geringer fällt der Fehler jedoch aus, beziehungsweise relativiert sich dieser Fehler. Wie man in Abbildung 3 sehen kann, gibt es dabei zwei verschiedene Varianten der Verzeichnung. Je nach Position von Linse, Blende sowie Abstand zwischen Linse und Sensor, also Brennweite, tritt eine kissenförmige oder tonnenförmige Verzerrung auf, sodass die Weltkoordinaten nicht so auf den Bildkoordinaten abgebildet werden, wie man es erwartet. In Falle der AR-Drone Kamera tritt eine deutliche tonnenförmige Verzeichnung auf. In Abbildung 4 wurde ein Bild mit der Frontkamera der Drohne von einer Kalibrierungsmaske erstellt. Tritt bei Bilddaten eine solche Verzeichnung auf, so muss der intrinsische Parameter der Kamera bestimmt werden, um das Bild zu entzerrten.

### 2.2.3 Intrinsische und extrinsische Parameter

Um das Verhältnis zwischen 3D-Weltkoordinaten und 3D-Kamerakoordinaten bestimmen zu können, benötigt man die sogenannten extrinsischen Parameter. Dabei handelt es sich um die Position in X, Y und Z-Richtung. Zudem werden die Rotationen in den drei Richtungen betrachtet. Sind alle Parameter bekannt können Formeln aufgestellt werden, um Bildkoordinaten in Weltkoordinaten zu transformieren und Weltkoordina-

ten in Bildkoordinaten mit der Umkehrfunktion zu transformieren. Da im Rahmen dieser Arbeit vom Ursprung des Koordinaten Systems an der Position der Drohne ausgegangen wird, werden die extrinsischen Parameter nicht weiter betrachtet. Um das Bild zu entzerrn, werden die intrinsischen Parameter benötigt. Diese bestimmen die Transformationsfunktion von Kamerakoordinaten zu Pixelkoordinaten. Das heißt, die Übersetzung von einem Punkt des Motivs zu einem Pixel im Bild. Mit der Umkehrfunktion kann auch hier aus den Pixelkoordinaten zu Kamerakoordinaten transformiert werden. Allerdings kann dabei kein Rückschluss auf die Tiefendaten gemacht werden. Diese intrinsischen Parameter werden in der Regel durch sogenannte Kalibrierungsmasken bestimmt. Siehe Abbildung 4. Dabei handelt es sich um ein Bild, welches bekannte Formfaktoren hat. Auf dem Motiv sind in der Regel mehrere Objekte einer Form abgebildet wie zum Beispiel Quadrate oder Kreise. Dabei werden häufig Kreise gewählt, da dabei die Orientierung der Maske irrelevant ist und auch leichte Drehungen des Bildes nicht beachtet werden müssen. Die Abstände der Objekte und deren Größe auf der Kalibrierungsmaske müssen zuvor bekannt sein. Nur so ist es möglich genau berechnen zu können, um welche Vektoren das Bild verzerrt oder verzeichnet wird. Diese Maske wird in verschiedenen Positionen und mit unterschiedlichen Winkeln vor die Kamera gehalten. Aus den daraus resultierenden Bildern können mithilfe von speziellen Programmen die intrinsischen Parameter bestimmt werden, sodass die entstandenen Aufnahmen entzerrt werden können.

#### 2.2.4 Stereoskopie

Nachdem der Übergang der Kamerakoordinaten in Weltkoordinaten gegeben ist, müssen aus den Bildern Tiefendaten generiert werden. Wie vorher angesprochen kann bei der Rücktransformation von Pixelkoordinaten in Kamerakoordinaten zwar die Höhe und Breite der Punkte, oder beziehungsweise die X- und Y-Koordinate berechnet werden, allerdings gibt die Transformation keinen Rückschluss auf die Tiefe oder Z-Koordinate des Punktes. Um die Z-Koordinate bestimmen zu können, muss ein zweites Bild, aus einem anderen Winkel betrachtet werden. Ein Mensch ist in der Lage aus zweidimensionalen Bildern die Tiefendaten abzuschätzen. Das geschieht in der Regel durch unterschiedliche "[...] Größenverhältnisse, Perspektive, Texturen, Luftphänomene, Licht und Schatten." [2] Unser Gehirn füllt die fehlenden Informationen durch Erfahrung und bereits wahrgenommene Phänomene auf, wodurch man das Gefühl einer gewissen Tiefe erhält [2]. Diese monokularen Tiefeninformationen könnte ein Computer mit Hilfe von neuronalen Netzen auch ableiten um, so eine Tiefenkarte eines Bildes zu erstellen. Doch auch das Gehirn und eben diese neuronalen Netze können ausgetrickst werden,

wie eine Vielzahl von Bildern zur optischen Täuschung beweisen. Um zuverlässige Tiefendaten zu erhalten, führt derzeit kein Weg an binokularen Tiefendaten vorbei. Diese erhält man durch den Einsatz von zwei Kameras, welche auf das gleiche Motiv gerichtet sind. Da die beiden Kameras nicht den selben Platz im Raum einnehmen können, entsteht bei beiden Bildern ein perspektivischer Unterschied. Wenn die beiden Bilder ähnlich genug, aber dennoch verschieden sind müssen die gleichen Punkte des Motivs in beiden Bildern identifiziert werden. In Abbildung 5 sind zwei Augen abgebildet, die auf ein Haus und einen Baum schauen. Das Haus ist dabei im Vordergrund und der Baum ist hinter dem Haus. Im unteren Bereich der Abbildung ist die Sicht aus dem jeweiligen Blickwinkel gegeben. Dabei ist zu beachten, dass aus der Sicht des linken Auges der Baum zu einem größeren Teil verdeckt ist als aus der Sicht des rechten Auges. Mit Hilfe des Abstands der beiden Verdeckungen und dem Abstand der Augen oder der Kameras von einander ist es nun möglich das Verhältnis der beiden Objekte zu bestimmen. Zuvor muss allerdings die rechte Kante des Hauses im rechten und im linken Bild gesucht werden. Danach muss die linke Kante des Baumes in beiden Bildern gesucht werden. Oder allgemein werden Punkte in den Bildern gesucht, die in beiden enthalten sind und klar zu identifizieren sind. Es werden also "Matching Feature Points" gesucht und durch die Abstände dieser Punkte in Pixel Rückschluss auf deren relativen Abstand getroffen.

Nachdem die Relativen Abstände der Gegenstände bekannt sind müssen, die relativen Werte in absolute Werte umgerechnet werden. Dabei werden ebenfalls die Daten der intrinsischen Kalibrierung genutzt. Eine Voraussetzung an die Kalibrierungsmaske ist, dass die Größe und der Abstand der Objekte bekannt sein müssen. Durch diese Informationen ist es nun möglich zuverlässig die X-, Y- und Z-Koordinate der einzelnen Punkte im Raum zu bestimmen.

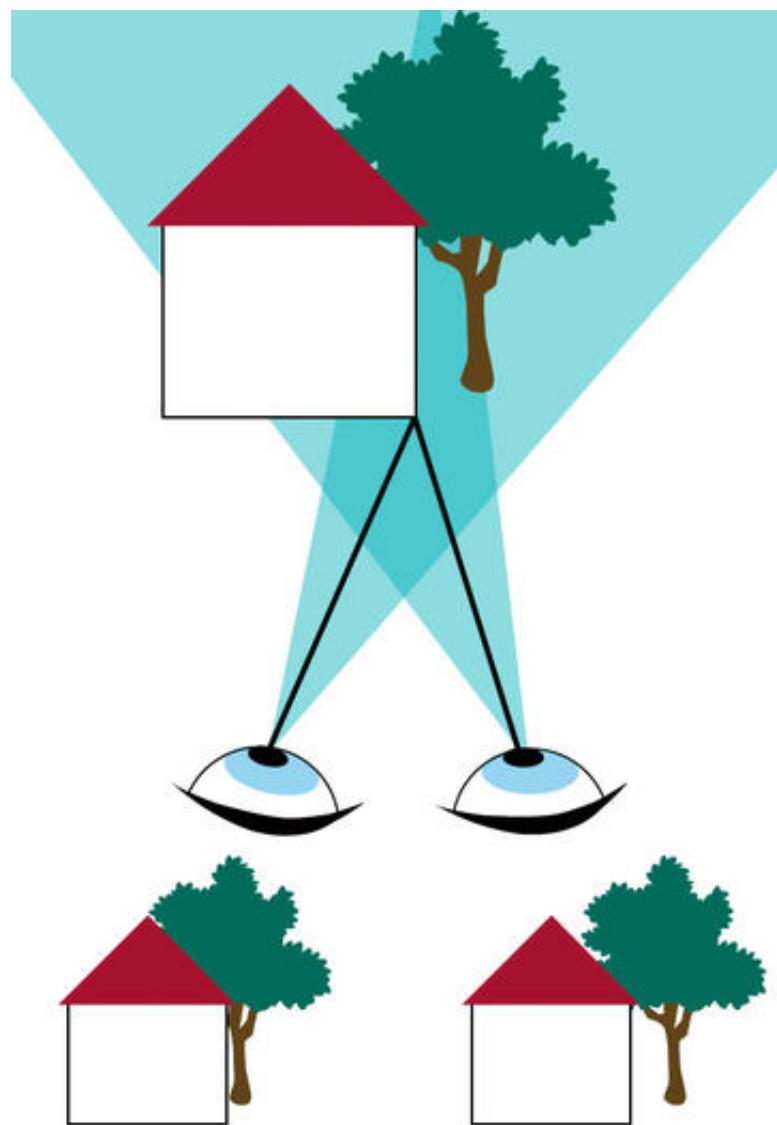


Abbildung 5: Zwei Augen, die eine Haus und ein Baum ansehen [2]

## 2.2.5 Organisation der 3D-Daten

Als Datenformat zur Speicherung und Verarbeitung der Tiefendaten wird das PCD-Format eingesetzt. Dabei handelt es sich um ein Dateiformat, dass von der Point Cloud Library (PCL) entwickelt wurde und im Aufbau stark an andere 3D-Punktwolken Dateiformate erinnert, jedoch um Funktionen und Attribute der PCL ergänzt wurde. Gespeichert werden die Punkte in der Regel mit ihren X,Y,Z Koordinate und können unterdessen optional einen Farbwert erhalten, wie es auch in dem Beispiel gehandhabt wird. Im Anschluss an den Header, der unter anderem die Definitionen für den Dateiaufbau und die Größe beinhaltet, folgt der Body. In diesem Werden Zeilenweise

die Tiefendaten aufgelistet. In der Regel entspricht die Breite und Höhe der Punktwolke der, des aufgenommenen Bildes, sodass jeder Bildpunkt auf einen Punkt der Punktwolke mit XYZ-Koordinate und Farbwert abgebildet wurde. Konnte zu einem Pixel des Bildes kein Rückschluss auf die Position getroffen werden, da sich dieser zum Beispiel außerhalb des Definitionsbereichs befanden, also zu nah oder zu weit weg war um Angaben über dessen Position zu machen, wird dieser mit „nan“, also „Not a Number“ für jedes Attribut angegeben. Um Speicherplatz zu sparen, wird aus diesem Grund oft die Binärversion der PCD-Dateien erstellt.

Listing 3: Der exemplarische Aufbau einer PCD Datei

```

1 # .PCD v.7 - Point Cloud Data file format
2 VERSION .7
3 FIELDS x y z rgb
4 SIZE 4 4 4 4
5 TYPE F F F F
6 COUNT 1 1 1 1
7 WIDTH 213
8 HEIGHT 1
9 VIEWPOINT 0 0 0 1 0 0 0
10 POINTS 213
11 DATA ascii
12 0.93773 0.33763 0 4.2108e+06
13 0.90805 0.35641 0 4.2108e+06
14 nan nan nan nan
15 0.97192 0.278 0 4.2108e+06
16 0.944 0.29474 0 4.2108e+06
17 [...]

```

## 2.2.6 Point Clouds

Point Clouds, zu deutsch Punktwolken, sind nichts anderes als eine beliebig große Menge an Punkten im dreidimensionalen Raum. Sie beschreiben zumeist Objekte und Räume und sind das Produkt der Anwendung von Machine Vision. Sie können unterschiedliche Informationen beinhalten, je nachdem von welchem Typ sie sind und welchen Anwendungszweck sie erfüllen. Allen Varianten ist jedoch gemeinsam, dass sie Punkte in 3 Koordinaten X,Y und Z beschreiben. Sie enthalten zumeist auch Zusatzinformationen über die Farbe oder entsprechende Grauwerte. Die Farbinformationen

gestalten sich analog wie bei herkömmlichen zweidimensionalen Bildern. Man unterscheidet erneut die Systeme RGB, YMC und HSV.

**Farbsysteme** Bei den verschiedenen Farbsystemen unterscheidet man zum einen das additive Farbsystem vom subtraktiven Farbsystem. Bei dem additiven Farbsystem geht man von der Farbe schwarz aus und fügt immer mehr Farben hinzu. Die hinzufügbaren Farben sind dabei Rot, Grün und Blau. Durch Mischung dieser drei Farben mit verschiedener Sättigung lassen sich nahezu alle Farben mischen. Eingesetzt wird dieses Verfahren zum Beispiel bei LCD (Liquid Crystal Display). Bei subtraktiven Farbsystemen spricht man in der Regel von YMC. Hier geht man von Weiß als Grundfarbton aus und zieht die entsprechenden Farben ab. Das Farbsystem besteht in diesem Fall aus Gelb, Magenta und Cyan. Dieses Verfahren wird vorzugsweise im Druck eingesetzt, da das Papier weiß ist und es somit möglich ist, Farben auf dem Papier korrekt darzustellen. Während man RGB und YMC in einem 3-Dimensionalen Koordinatensystem als Würfel darstellen kann wird HSV gängig als Kegel dargestellt. Hier bestehen die drei Komponenten aus Hue (Farbton), Saturation (Farbsättigung) und Value (Helligkeit). Der Farbton wird in dieser Darstellung als Winkel um den Kegel angegeben, die Farbsättigung repräsentiert den Abstand zur Mittelachse und die Helligkeit wird über die Höhe des Kegels definiert.

Eine Punktwolke wird meist in einer .pcd Datei hinterlegt. Dabei handelt es sich um ein Dateiformat, dass von der PCL entwickelt wurde und im Aufbau stark an andere 3D-Punktwolken Dateiformate erinnert, jedoch um Funktionen und Attribute der PCL ergänzt wurde.

Der Header enthält Informationen über die Struktur der darauf folgenden Daten, wie die PCL-Version, Anordnung der Datenfelder, Größe, Typ, Anzahl und Breite sowie Höhe des Bildes. Im Body werden Zeilenweise die Tiefendaten aufgelistet. In der Regel entspricht die Breite und Höhe der Punktwolke der des aufgenommenen Bildes, sodass jeder Bildpunkt auf einen Punkt der Punktwolke mit XYZ-Koordinate und Farbwert abgebildet wurde.

Listing 4: Der exemplarische Aufbau einer PCD Datei

```

1 # .PCD v.7 - Point Cloud Data file format
2 VERSION .7
3 FIELDS x y z rgb
4 SIZE 4 4 4 4

```

```

5  TYPE F F F F
6  COUNT 1 1 1 1
7  WIDTH 213
8  HEIGHT 1
9  VIEWPOINT 0 0 0 1 0 0 0
10 POINTS 213
11 DATA ascii
12 0.93773 0.33763 0 4.2108e+06
13 0.90805 0.35641 0 4.2108e+06
14 nan nan nan nan
15 0.97192 0.278 0 4.2108e+06
16 0.944 0.29474 0 4.2108e+06
17 [...]

```

## 2.3 Entwicklungsumgebung

DANIEL BETSCHE

Der folgende Abschnitt beschreibt, welche Software und Werkzeuge genutzt werden, um die Anwendung in dieser Studienarbeit zu entwickeln. Dabei gilt es unterschiedliche Aspekte abzudecken im Bereich Programmiersprachen, Frameworks und IDEs.

### 2.3.1 JavaScript

JavaScript hat trotz der Namensverwandtschaft keine großen Gemeinsamkeiten mit der Programmiersprache Java. Es ist entstanden als eine Skriptsprache für die dynamische Gestaltung von HTML in Webbrowsern. Der ursprüngliche Zweck war es, Funktionen des Servers auf den Client auszulagern, um so an Performance zu gewinnen und Netzwerkverkehr zu reduzieren. JavaScript hat die Fähigkeit Benutzereingaben zu verarbeiten, Inhalte der Webseite zu verändern, nachzuladen oder zu generieren. Seit seiner Erscheinung 1995 hat es sich stark weiterentwickelt und ist inzwischen auch auf Servern und Mikrocontrollern zu finden. Ein Beispiel für den serverseitigen Einsatz ist NodeJS.

### 2.3.2 Node JS



Quelle:  
[nodejs.org](http://nodejs.org)

Wie in 2.1.3 beschrieben ist NodeJS eine Laufzeitumgebung für serverseitige Webanwendungen. Das herausragende Merkmal von NodeJS ist seine ereignisbasierte Architektur, welche sich vor allem für asynchrone I/O basierte Anwendungen eignet.

Node JS wurde ursprünglich im Jahr 2009 von Ryan Dahl entwickelt. Der erste Release unterstützte nur Linux-Umgebungen. Die Architektur der Anwendung hat sich im Vergleich zu seiner Entstehungszeit kaum verändert. Eine Kombination der Google V8 JavaScript Engine zusammen mit einer Event-Loop in Verbindung mit einer low-level I/O API stellt den Grundpfeiler von Node JS dar. Die Ziele bei der Entwicklung waren einen möglichst hohen Durchsatz zu erzielen und das System beliebig stark skalieren zu können.

Heutzutage bildet Node JS ein wichtiges Fundament des gesamten Internets und wird von Tausenden unabhängigen OpenSource-Entwicklern und interessierten Internetkonzernen genutzt und vorangetrieben. Diese Menge an Ressourcen für die Weiterentwicklung ermöglichen durchgehende Performance-Steigerungen, Effizienz, Stabilität und neue Features wie etwa JavaScript ES6.

### 2.3.3 C++

Roboter stellen ganz besondere Anforderungen an ihre Entwickler und werden im industriellen Umfeld häufig mit Hersteller eigenen Programmiersprachen entwickelt. Eine Analyse relevanter Frameworks zur Entwicklung von Robotikanwendungen an der Uni Magdeburg zeigt jedoch, dass „hinsichtlich der Programmiersprachen [...] C++ schon wegen der vielfältigen (Standard-)Bibliotheken an erster Stelle [steht][25, S.38]“. Aus der Analyse wird auch klar, dass die Sprache Python als weitverbreitete Alternative eingesetzt wird.

Roboter wurden ursprünglich mit dem Einsatz von Mikrocontrollern realisiert. Die vorherrschende Programmiersprachen war zu diesem Zeitpunkt Assembler und C. Mit dem Fortschritt in der Technologie wurden auch an die Programmiersprachen neue Anforderungen gestellt. Daraus entwickelte sich die Sprache C++, welche als Erweiterung von C mit neuen Konzepten aus der Softwareentwicklung entworfen wurde. C++ basiert damit auf dem damaligen Stand von C im Jahre 1990. Sie wird offiziell als „allgemein Programmiersprache mit einer Tendenz zur Systemprogrammierung, welche

ein besseres C ist“[26] von ihrem Entwickler Bjarne Stroustrup beschrieben.

Der Vorteil von C++ ist die Kombination von modernen mächtigen Sprachmitteln und einer effizienten maschinennahen Programmierung. Dadurch lassen sich komplexe Programme hinter simplen Interfaces abstrahieren.

### 2.3.4 ROS

CHRISTIAN VERDION UND DANIEL BETSCHE



Quelle:  
[ros.org](http://ros.org)

Das Robot Operating System (ROS) ist ein kollaboratives Open Source Projekt, mit dem Ziel durch gemeinsames Wissen eine Basis zur Steuerung von Robotern und Drohnen mit den bestmöglichen Komponenten zu entwickeln. Roboter sind hochkomplexe Maschinen und benötigen Fachwissen in viele unterschiedliche Disziplinen der Naturwissenschaften. Erst die Summe aller notwendigen Bausteine ermöglicht es, einen funktionierenden Roboter zu entwickeln und zu steuern. Das Fachwissen aus Ingenieurbereichen wie Mathematik, Physik und Informatik allein reicht zwar aus um einen betriebsfähigen Roboter zu erstellen, aber die Performance wird stark limitiert sein. Der Gedanke hinter ROS gilt der Zusammenarbeit von Expertenteams weltweit mit unterschiedlichen Spezialisierungen. 'Zum Beispiel hat ein Labor vielleicht Experten im Abbilden von Umgebungen in Gebäuden und kann ein Weltklasse System zum Erstellen von Karten liefern. Eine andere Gruppe hat vielleicht Experten darin solche Karten zur Navigation zu nutzen und eine weitere Gruppe hat einen Computer Vision Ansatz entdeckt um viele kleine vertraute Objekte sehr exakt erkennen zu können.'[9] ROS dient hier als gemeinsame Plattform, um sich auszutauschen und auf der Arbeit der jeweils anderen aufbauen zu können. Durch diese Zusammenarbeit von unterschiedlichen Expertengruppen wird es möglich, mit deutlich höherer Qualität und Effektivität zu arbeiten. Auch zur Steuerung der Parrot AR Drone 2.0 existiert ein Paket für ROS, es nennt sich AR-Drone Autonomy.

**ROS Core und ROS Nodes** Softwaresysteme die Gebrauch von ROS machen werden für gewöhnlich in kleine Teile aufgeteilt. Durch einen modularen Aufbau vereinfacht man die Wartung und Fehlersuche bei bestehenden Systemen und kann beim Entwickeln neuer Systeme auf andere Softwareteile zugreifen, die bereits programmiert

worden sind. Ebenfalls lassen sich einzelne Module einfach aktualisieren und austauschen, ohne Einfluss auf andere Module zu haben.

Diese Module nennt man im Kontext von ROS 'Nodes'. Jede Node wird im System registriert und kommuniziert mit anderen Nodes über vorher fest definierte Nachrichtentypen. Das Herzstück des ROS Ökosystems bildet ROS Core. Mit dem Start dieses Programms werden drei dieser Nodes gestartet. Eine Master-Node, der Parameterserver und der Logserver *rosout*.

Die Aufgabe des Master Knotens ist es, naming und registration services für alle anderen Nodes im System bereitzustellen. Er verwaltet alle Herausgeber und Abonnenten von Topics und Services und stellt sicher, dass individuelle ROS Nodes sich gegenseitig finden können. Ist eine Verbindung erst einmal hergestellt, so kommunizieren diese Nodes direkt miteinander.

Der Parameterserver dient als geteilte Ressource, um Konfiguration zur Laufzeit auszutauschen und gegebenenfalls anzupassen. Er ist nicht auf hohe Performance ausgelegt und sollte daher nur für statische Konfiguration genutzt werden. Der Logserver dient als zentrale Stelle zur Ausgabe von Logdaten und entspricht einer Standardausgabe für ROS Nodes.

**rosrun und Launchfiles** Eine Node in ROS kann über den Befehl *rosrun* gestartet werden. Die Syntax ist *rosrun <package> <node>*. Dadurch wird eine einzelne Node initialisiert, beim Master angemeldet und beginnt ihre Arbeit. Sollte kein Master Knoten vorher gestartet worden sein, so wird eine Fehlermeldung ausgegeben und die Ausführung angehalten.

Eine Anwendung in ROS besteht eigentlich immer aus einer Vielzahl von verschiedenen Nodes, welche gleichzeitig aktiv sein müssen. Jede beteiligte Node einer Anwendung einzeln über *rosrun* zu starten ist zeitaufwändig und bietet ein hohes Fehlerpotential. Zur Vereinfachung der Ausführung von komplexen Anwendungen werden dafür '.launch' Dateien genutzt, die *Launchfiles*. In ihnen kann nicht nur eine Menge von zu startenden Knoten definiert werden, sondern auch zusätzliche Konfiguration festgelegt werden. In Listing 5 ist ein Auszug aus der Launchfile zu sehen, welche verwendet wird um die Drohne zu starten. Hier ist zu sehen, wie zuerst die Beschreibung der Drohne und ihre Referenzrahmen an den Parameter geschickt werden. Danach folgt die Definition von drei Nodes welche mit weiteren Parametern versehen werden.

---

Listing 5: Launchfile zum Starten einer Drohne

```

1 <param name="robot_description" command="$(find xacro)/xacro --inorder
2   '$(arg model)' base_link_frame:=base_link world_frame:=world" />
3 <param name="base_link_frame" type="string" value="base_link"/>
4 <param name="world_frame" type="string" value="world"/>
5
6 <!-- push robot_description to factory and spawn robot in gazebo -->
7 <node name="spawn_robot" pkg="gazebo_ros" type="spawn_model"
8   args="--param robot_description
9   -urdf
10  -x '$(arg x)'
11  -y '$(arg y)'
12  -z '$(arg z)'
13  -R '$(arg R)'
14  -P '$(arg P)'
15  -Y '$(arg Y)'
16  -model quadrotor"
17    respawn="false" output="screen"/>
18
19 <!-- start robot state publisher -->
20 <node pkg="robot_state_publisher" type="robot_state_publisher" name="
21   robot_state_publisher" output="screen" >
22   <param name="publish_frequency" type="double" value="50.0" />
23   <param name="tf_prefix" type="string" value="" />
24 </node>
25
26 <!-- tranform kinect pointcloud data to correct -->
27 <node pkg="tf" type="static_transform_publisher" name="
28   rotate_kinect_pointcloud" args="0 0 0 -1.58 0 -1.58 /base_link /
29   ray_link 100"/>
```

Eine Launchfile wird mit dem Befehl `roslaunch <package> <launchfile>` gestartet. Die Verarbeitung erfolgt immer sequentiell entsprechend der Reihenfolge, wie sie in der Datei festgelegt wurde. Anders als bei `rosrun` ist es nicht notwendig vorher einen Master Knoten zu starten. Ist ein aktiver Knoten vorhanden so wird dieser genutzt, falls nicht wird ein neuer Masterknoten gestartet.

**ROS Topics und Messages** Bei der Steuerung von Robotern und Drohnen entstehen viele Sensordaten, die als kontinuierliche Ströme versendet werden. Nicht selten kommen dabei m:n-Kardinalitäten zustanden. Das bedeutet, dass eine beliebige Anzahl von Sendern an eine beliebige Anzahl von Empfängern sendet und sowohl jeder Sender an alle Empfänger sendet als auch jeder Empfänger von allen Sendern empfängt. Die Nachrichten, die zwischen den einzelnen Komponenten versendet werden nennt man Messages. Der Aufbau, beziehungsweise die Datenstruktur einer solchen Message wird durch eine ".msg"-Datei definiert. Um zum Beispiel eine Nachricht über 2D-Koordinaten mit X und Y-Achse zu versenden, muss die Message Datei wie folgt geschrieben werden. Dabei ist zu beachten, dass zuerst der Datentyp und danach der Name der Variable angegeben wird.

Listing 6: Die coordinate.msg definiert eine Koordinatennachricht

```
1 int32 x
2 int32 y
```

Aus diesen Meta-Daten wird beim Kompilieren der C++ Code generiert, welcher später die Datenstruktur für die Nachrichten bildet. Jede Nachricht einer Node wird auf einer dafür bestimmten Topic veröffentlicht. Eine Topic kann zum einen von beliebig vielen Nodes abonniert werden, als auch von beliebig vielen Nodes veröffentlicht werden. Informationen über den Status einer Topic können jederzeit über den Befehl *rostopic info <topic\_name>* bezogen werden. Es ist anzumerken, dass auch jede Node beliebig viele Topics herausgeben und abonnieren kann. Die Mindestanforderung allerdings ist, dass jede Node die Topic */rosout* veröffentlicht und als Standardausgabe für Logeinträge nutzt.

Zur Übertragung der Nachrichten sieht ROS eine auf TCP/IP oder auf UDP basierende Übertragung vor. Standardmäßig wird die auf TCP/IP basierende Lösung genutzt, welche TCPROS genannt wird.[23]

**ROS Services** Mit ROS Services wird das Prinzip von Request und Response im ROS Ökosystem implementiert. Ein Knoten kann dadurch einen Service Call an einen anderen Knoten absetzen und erhält, anders als bei einer Message, garantiert eine Antwort zurück. Selbstverständlich muss der angesprochene Knoten aktiv sein, ansonsten wird eine Fehlermeldung ausgegeben. Services erlauben damit einen zuverlässigen Austausch von Informationen, wobei der Ansprechpartner eine konkrete ROS Node ist und keine Topic. Dadurch kann ein Service auch nicht von mehreren Nodes gleichzeitig angeboten werden.

Die Implementierung erfolgt ähnlich wie bei den ROS Messages, wobei die Informationen über das Protokoll in '.srv' Dateien hinterlegt wird. Auch diese werden beim Kompilieren in C++ Code übersetzt und bilden damit den Container für die Service Calls. In Listing 7 ist als Beispiel der Spawn Service des Tutorial-Pakets *Turtlesim*. Die Datei ist unterteilt in zwei Bereiche. Alles über den '---' gehört zum Request, alles darunter zur Response.

Listing 7: Die spawn.srv für den Service *spawn*

```

1 float32 x
2 float32 y
3 float32 theta
4 string name
5 ---
6 string name

```

**Buildprozess mit Catkin** Der Buildprozess von ROS Paketen wurde mit dem Release von Hydro von *rosbuild* auf *catkin* umgestellt. Catkin ist ein auf CMake basiertes Buildsystem welches speziell für ROS entwickelt wurde. Es bietet einen Standard CMake-Workflow an mit zusätzlichen Makros und Python Scripts zur Unterstützung von stark verteilten Anwendungen, wie es bei ROS standardmäßig der Fall ist.[10]

Die wichtigsten Informationen über das Projekt werden in den 'CMakeList.txt' und 'package.xml' Dateien festgehalten.

In der 'package.xml' Datei stehen Organisatorische Details wie zum Beispiel der Name des Projekts, Version oder Autor. Aber auch die Abhängigkeiten zu anderen Paketen sowohl während des Build-Vorgangs als auch zur Laufzeit werden in dieser Datei beschrieben.

Listing 8: Ein Beispiel für eine 'package.xml' Datei, die ein ROS Projekt beschreibt

```

1 <?xml version="1.0"?>
2 <package>
3   <name>beginner_tutorials</name>
4   <version>0.1.0</version>
5   <description>The beginner_tutorials package</description>
6
7   <maintainer email="you@yourdomain.tld">Your Name</maintainer>
8   <license>BSD</license>
9   <url type="website">http://wiki.ros.org/beginner_tutorials</url>

```

```

10 <author email="you@yourdomain.tld">Jane Doe</author>
11
12 <buildtool_depend>catkin</buildtool_depend>
13
14 <build_depend>roscpp</build_depend>
15 <build_depend>rospy</build_depend>
16 <build_depend>std_msgs</build_depend>
17
18 <run_depend>roscpp</run_depend>
19 <run_depend>rospy</run_depend>
20 <run_depend>std_msgs</run_depend>
21
22 </package>

```

Die 'CMakeList.txt' Datei dagegen enthält Informationen über den Buildvorgang an sich. Auch hier ist wieder der Projektname zu finden. Doch ebenso stehen hier welche Dateien zur Generierung von ROS Messages benötigt werden oder welche Dienste erstellt werden sollen, damit diese später ausführbar sind.

Listing 9: Eine 'CMakeList.txt' Datei zum Bauen eines ROS Projektes

```

1 cmake_minimum_required(VERSION 2.8.3)
2 project(beginner_tutorials)
3
4 ## Find catkin and any catkin packages
5 find_package(catkin REQUIRED COMPONENTS roscpp rospy std_msgs genmsg)
6
7 ## Declare ROS messages and services
8 add_message_files(DIRECTORY msg FILES Num.msg)
9 add_service_files(DIRECTORY srv FILES AddTwoInts.srv)
10
11 ## Generate added messages and services
12 generate_messages(DEPENDENCIES std_msgs)
13
14 ## Declare a catkin package
15 catkin_package()

```

Jedes Projekt lädt seine eigenen Abhängigkeiten und führt diese aus. An dieser Stelle erfolgt auch die Unterteilung von Paketen in die einzelnen Nodes. Jede Node besteht

dabei immer am Ende aus einer ausführbaren Binärdatei.

### 2.3.5 Point Cloud Library

CHRISTIAN VERDION



Quelle:  
pointclouds.org

Die PCL ist eine Sammlung von Werkzeugen und state-of-the-art Algorithmen zur Bearbeitung von Punktwolken im mehrdimensionalen Raum. Von Willow Garage 2010 gegründet, einem Hersteller für Hard- und Open Source Software für Personalroboter, wurde es schnell von der Community um die Robotik angenommen und wird mittlerweile von einem großen Konsortium finanziert. Seit 2012 wird es als Projekt bei der Open Perception Foundation verwaltet, welche aus Willow Garage hervorging. Verwendet wird die PCL mittlerweile von großen Firmen wie Intel, NVidia und auch diversen Hochschulen. Lizenziert ist das Framework über die BSD-Lizenz und ist somit sowohl Kommerziell als auch zu Forschungszwecken frei zur Verwendung freigegeben. Eine stetige Erweiterung des Funktionsumfangs ist vor allem durch die Open-Source Community gegeben. [11][16] Das Framework ist wie das ROS in C++ geschrieben, was in der Robotikgemeinschaft viel Anklang findet, da diese sehr gut miteinander zu kombinieren sind und sich für diverse Plattformen wie MacOS, Windows und Linux verwenden lassen. Bei Design und Entwicklung wurde ebenfalls an die oft dürftigen Systemressourcen der Roboter und eingebetteten Systemen gedacht, weshalb das Framework ebenfalls sehr modular aufgebaut ist und in kleine Module aufgeteilt wurde.

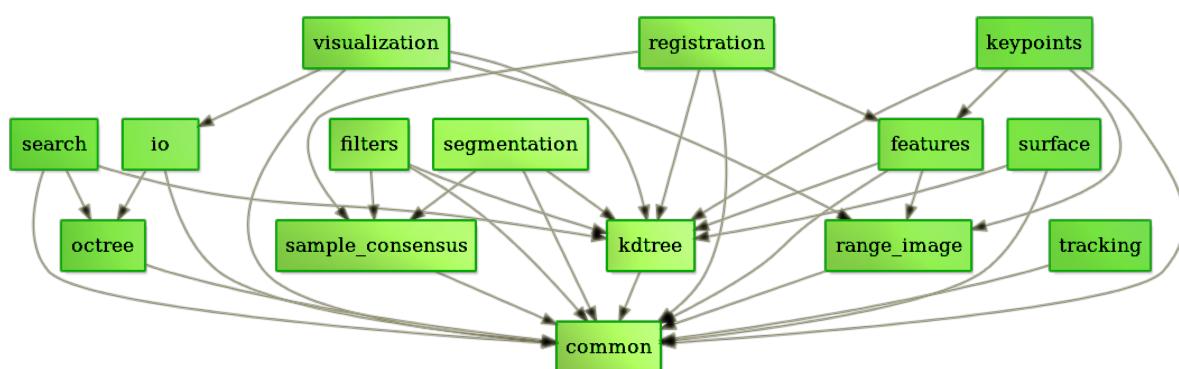


Abbildung 6: Module und Abhängigkeiten der Point Cloud Library [16]

In dieser Arbeit kamen die Pakete „registration“ für den Iterative Closest Point Algo-

rithmus und „features“ zur Kantenerkennung zum Einsatz. Zu Testzwecken wurde das „visualization“ Paket zur Darstellung von PointCloud Daten verwendet.

## 2.4 Arbeiten mit Simulationen

DANIEL BETSCHE

Simulationen dienen als dritte Säule in der Forschung neben der Theorie und dem Experiment. Die Simulation ist ein Gedankenexperiment und wurde vor der Erfindung des Computers auch als solches im Kopf oder auf dem Papier durchgeführt. Selbstverständlich hat dies die Nutzbarkeit deutlich eingeschränkt. Mit modernen Computersystemen lassen sich sehr komplexe Simulationen erstellen und der Detailgrad kann im laufenden Betrieb dynamisch abgeändert werden. Zum Beispiel das an- und ausschalten von Gravitation mit nur einem Klick. Simulationen dienen hauptsächlich zur Entwicklung und Analyse von Systemen. Das Vorgehen ist dabei meist, dass ein Modell erstellt wird von dem zu überprüfenden Sachverhalt mit frei veränderbaren Parametern. Die Simulation ist so wichtig geworden, dass ein ganzer Forschungszweig, das High Performance Computing, sich damit beschäftigt möglichst effizient und schnell Simulationen für ungelöste Probleme zu entwickeln und durchzuführen.

### 2.4.1 Warum Simulieren

Simulationen werden immer häufiger genutzt, weil sie schnell, effizient, billig und ungefährlich sind. In manchen Fällen sind Experimente schlichtweg zu teuer, zu gefährlich oder einfach nicht möglich. Das ist unter anderem der Fall, wenn die Systeme dynamischer Natur sind und eine korrekte Messung der gesuchten Parameter nicht durchführbar ist. Simulationen haben den Vorteil, dass ohne Verschleiß und zusätzliche Kosten im Labor schnell Ergebnisse erzielt werden können. Gerade in der Robotik ist dies unerlässlich um schnell Algorithmen zu testen, Roboter zu designen und modellieren sowie wiederholt Szenarien durchzuspielen. In unserem Fall dient es der Entwicklungszeit der Drohnensteuerung aufgrund der eingelesenen Sensor- und Bilddaten. Beim Einsatz der echten Drohne besteht immer die Gefahr die Drohne dauerhaft zu beschädigen durch fehlerhafte Steuerbefehle oder unvorhergesehene äußere Einflüsse wie etwa starke Windstöße. In einer Simulation lassen sich alle relevanten Parameter auf kleinster Detailebene kontrollieren.

## 2.4.2 Relevante Parameter

Um eine korrekte Simulation durchführen zu können, ist es notwendig die Parameter zu identifizieren welche einen Einfluss auf das Ergebnis der Simulation haben. Für viele Parameter können im Kontext der Studienarbeit die Standardwerte übernommen werden. Es gibt keinen Grund, warum die Gravitation oder das Magnetfeld angepasst werden sollten. Von Bedeutung sind vor allem die Werte für die Flugdrohne. Darunter fällt das Gewicht, die Abmessungen sowohl des visuellen Körpers als auch des Kollisionskörpers und die Trägheitsvektoren.

Die gleichen Vorgaben gelten selbstverständlich für alle Körper die in der Simulation miteinander interagieren, auch allen Sensoren. Zur Vereinfachung des Modells wird allerdings angenommen das der Quadrocopter ein homogener Körper ist und damit die Massen und Trägheitsvektoren der Sensoren in seinen Parametern mit einfließen. Die Sensoren selbst stellen also neutrale Objekte dar, ohne Masse, Trägheit oder Kollision.

## 2.4.3 Rviz

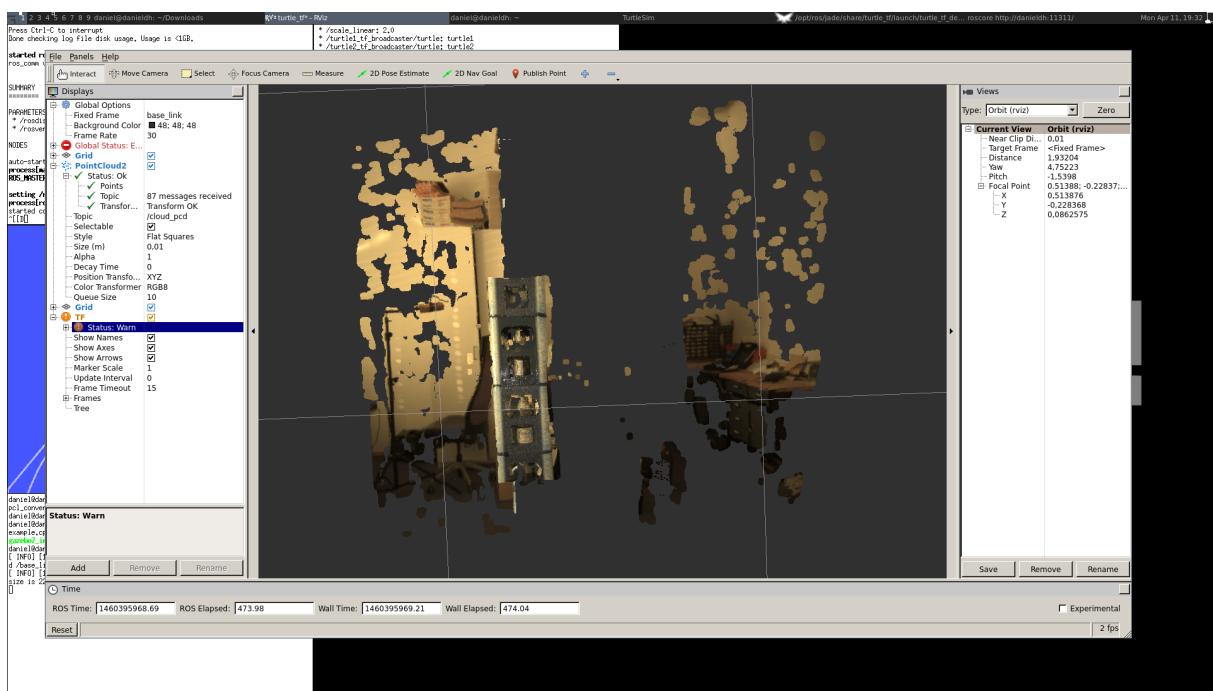
ROS bringt viele unterschiedliche Werkzeuge mit, welche zur Entwicklung, Steuerung und Analyse von Robotern hilfreich sind. Eines der wichtigsten dieser Werkzeuge ist Rviz. Wie es dem Namen bereits zu entnehmen ist, dient Rviz zur Visualisierung von Daten. Das Haupteinsatzgebiet ist die Darstellung von Sensorausgaben, damit ein Entwickler genau nachverfolgen kann welche Informationen sein Roboter gerade verarbeitet. Rviz ist in das Ökosystem von ROS eingegliedert und kann Sensordaten direkt über die veröffentlichten Topics mitlesen. Für alle relevanten Datentypen gibt es eine Darstellungsform, die wichtigsten davon sind:

- Kameradaten
- Tiefendaten
- Gitternetze
- Laserscans
- Karten
- Pfade
- PointClouds
- Geometrische Objekte

- Roboter Modelle

Im Allgemeinen lassen sich also alle Sensordaten eines Roboters so visualisieren. Bevor dies möglich war, mussten Robotiker anhand von reinen Zahlenwerten überprüfen können, warum der Roboter in einer Situation so handelte, wie er es tat. Durch die Visualisierung der vom Roboter gesehenen und erfahrenen Welt lässt sich dies deutlich einfacher nachvollziehen. Rviz baut ein dreidimensionales Modell auf und ordnet anhand der TF Daten alle Objekte in den gewünschten Referenzrahmen (auch Frame genannt) ein. Die Steuerung erfolgt allein durch Bewegungen mit der Maus. Es ist möglich die Anzeige auf allen Achsen zu drehen oder zu zoomen.

Abbildung 7: PointCloud von Myestro Büro in Rviz



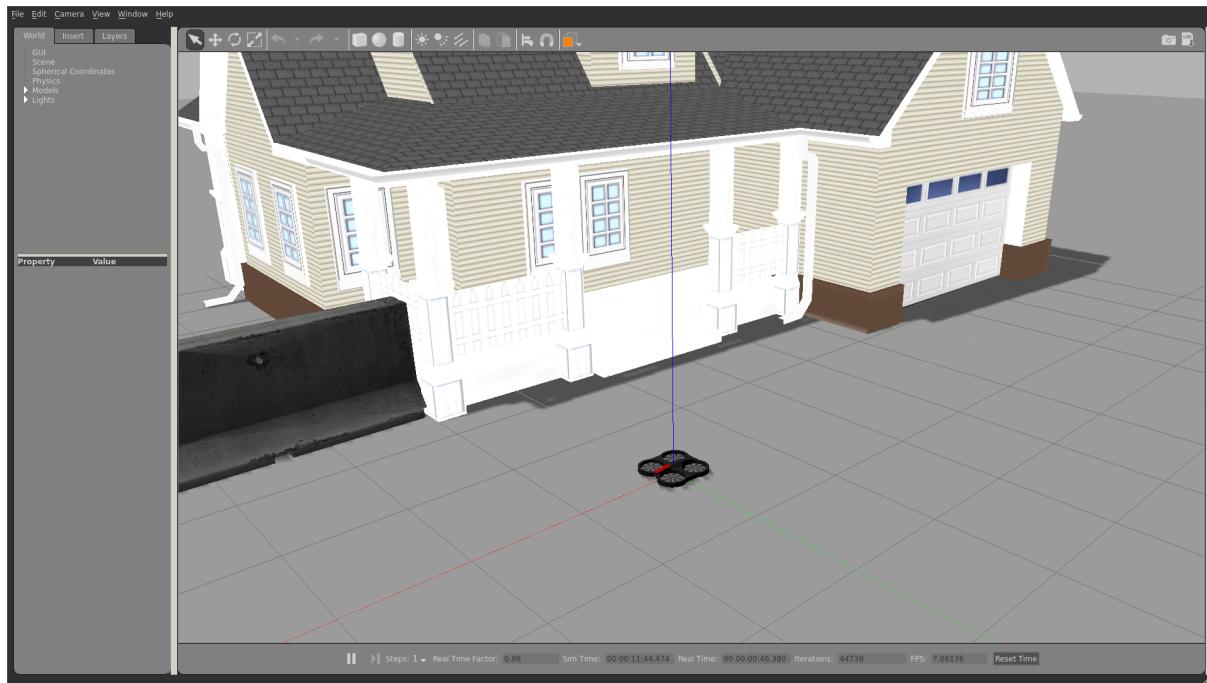
#### 2.4.4 Gazebo

Gazebo hat begonnen als eine Simulationsumgebung innerhalb des ROS-Ökosystems. „Mit dem Release von Gazebo 1.9 und ROS Hydro, besitzt Gazebo keine weiteren Abhängigkeiten von ROS und ist nun verfügbar als eigenständiges Ubuntu-Paket[6]“.

Gazebo ist ein Simulator speziell für den Bereich der Robotik entwickelt. Er dient als Werkzeug um „schnell Algorithmen testen, Roboter designen und Regressionstest durchführen zu können mit realistischen Szenarien[7]“. Der besondere Vorteil ist die auch nach der Abspaltung immer noch sehr gute Integration in das ROS Framework

und native Unterstützung von Topics und Services.

Abbildung 8: Virtualisierte Umgebung mit AR Drone



**SDF** „SDF ist ein XML Format zur Beschreibung von Simulation, Visualisierung und Steuerung von Robotern. Es wurde ursprünglich als ein Teil des Gazebo Simulators entworfen mit dem Blick auf den Einsatz in wissenschaftlichen Robotikanwendungen. Über die Jahre hinweg hat es sich entwickelt zu einem zuverlässigen, robusten und erweiterbarem Format zur Beschreibung jeglicher Aspekte von Robotern, statischen und dynamischen Objekten, Lichtquellen, Terrains und physikalischen Eigenschaften.[8]“

SDF bildet im aktuellen ROS System die neue Version von URDF und XACRO. Die alten Formate werden vor der Ausführung in Gazebo von Präprozessoren zu SDF konvertiert. Aus Gründen der Rückwärtskompatibilität werden alle Formate weiterhin unterstützt, jedoch nach und nach durch reines SDF ersetzt. Die vollständige Spezifikation findet sich unter <http://sdformat.org/spec>.

**World, Model und Sensor** Die wichtigsten Elemente in SDF für die Studienarbeit sind World, Model und Sensor. In einem World Element werden ganze Welten definiert, sie bilden das Grundgerüst der Simulation. Als Grundlage für die meisten Simulationen wird eine leere Welt genutzt, welche dann mit weiteren Modellen bestückt wird.

Die Modelle bilden die Objekte innerhalb des Simulators. Ein Model kann statisch sein, wie etwa eine Säule, oder dynamisch, z.B. ein Fahrzeug. Jedes Model kann weiter mit Sensoren bestückt werden. Davon gibt es eine Auswahl von allen gängigen Sensortypen, welche in der Robotik Verwendung finden. Von Kameras über Beschleunigungssensoren, Kompassen, Sonare und GPS bis hin zu Laserscannern und Radiowellenempfängern.

**Gazebo Plugins** Bis zu diesem Schritt sind alle Elemente der Simulation deklarativ in XML formuliert. Gazebo Plugins ermöglichen eine programmatische Manipulation der Welt und ihrer Modelle. Dazu werden C++ Bibliotheken geschrieben, welche direkt an die API von Gazebo angebunden sind um ihre Parent-Elemente steuern zu können.

Es gibt drei unterschiedliche Arten von Plugins. Model-Plugins, World-Plugins und System-Plugins. Vor allem von Interesse sind die Model-Plugins, denn sie erlauben es den Robotern die Fähigkeit zu geben sich zu bewegen. Die Plugins können direkt an das ROS Framework angebunden werden und über eine Node kommunizieren.

Aufgrund des Einsatzes von C++ Bibliotheken können Plugins beliebig komplexe Funktionen ausführen und ermöglichen eine sehr realistische Nachbildung von Modellen angelehnt an ihre physikalischen Vorbilder.

### 3 Architekturübersicht

DANIEL BETSCHE

Die Anwendung welche in dieser Studienarbeit entwickelt wurde, basiert auf einem stark modulbasierten Ansatz. Der Grund dafür ist zum einen die Nutzung des ROS Frameworks welches selbst auf unabhängigen Modulen aufgebaut ist, die über ein Message-Passing-System miteinander kommunizieren. Zum anderen ist es notwendig für die vorliegende Studienarbeit zur Verbesserung der Entwicklungszeit mit einer Abstraktion der physischen Komponenten in einem kontrollierten Umfeld arbeiten zu können, sprich der Simulation.

Zur Eingliederung beider Fälle, echter und virtualisierter Drohne, muss also eine Architektur entworfen werden, welche beide Varianten unterstützt und möglichst einfach zwischen ihnen wechselt lässt.

#### 3.1 Reale Drohne in Kombination mit Rubbertrion

Die anfängliche Projektplanung hatte ausschließlich eine Architektur vorgesehen, bei der eine AR Drone 2.0 genutzt wird, welche dem System Kamera- und Sensordaten liefert. Die Kameradaten werden, wie in Figur 9 zu sehen ist, von der Drohne in einen ROS-Node übertragen welcher diese Daten als Stream einzelner Bilder ausgibt. Die Software von Myestro erhält diese Bilder als Eingabe und gibt mit Hilfe des Rubbertrion Algorithmus zu jeweils zwei Bildern eine passende Punktwolke aus. Diese Punktwolken bilden den Ansatz für die Analyse der Umgebung. Die Besonderheit liegt darin, dass anstelle eines hochqualitativen Laserscanners, wie er in vielen Robotern zum Einsatz kommt, eine einfache Kamera durchschnittlicher Qualität ausreicht.

Wie in der Architekturskizze zu erkennen ist, werden die einzelnen Punktwolken zu einer Gesamtansicht zusammengefasst. Diese stellt ein dreidimensionales Modell der Umgebung der Drohne dar. Dieses Modell wird mit den Sensordaten der Drohne kombiniert um eine vollständig Ansicht des aktuellen Zustands zu erhalten. Dieser Zustand ist grundlegend um Entscheidungen über die nächsten Aktionen treffen zu können. Die Drohne soll zwar autonom fliegen können, jedoch sollte der Benutzer die Fähigkeit besitzen jederzeit eingreifen zu können. Dazu dient die Anbindung eines Joysticks zur Drohnensteuerung. Mit diesem kann zwischen manuellem und autonomen Modus gewechselt werden. Die Flugbefehle werden in einer sehr hoch abstrahierten Form an

das Paket AR\_Drone Autonomy weitergeleitet. Dieses ist wie in 2.1.4 beschrieben auf dem offiziellen SDK von Parrot basiert und überträgt die Befehle auf Protokoll-Ebene zur Drohne.

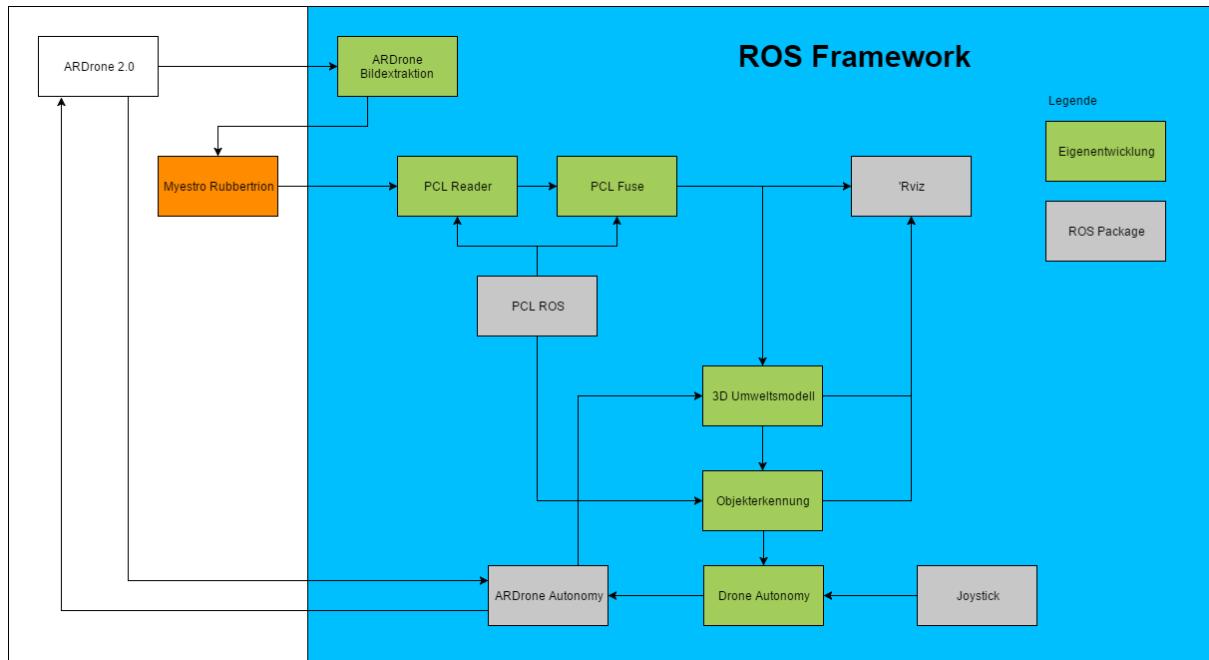


Abbildung 9: Architekturskizze mit echter Drohne und Rubbertrion

## 3.2 Virtualisierte Lösung

Die virtualisierte Drohne bietet gegenüber der echten einige Vorteile. Zum einen ist sie unabhängig von mechanischen Problemen und kann nicht durch unvorhergesehenes Verhalten beschädigt werden. Zum anderen ist diese Realisierung unabhängig von der sich noch in Entwicklung befindenden Software von Myestro. Möglich gemacht wird diese Lösung durch den Einsatz von 2.4.4Gazebo. Hierbei ist die Grundidee den Schritt der Auswertung der Bilddaten durch einen simulierten Laserscanner zu ersetzen um direkt Punktwolken zu erhalten. Damit kann unabhängig von Rubbertrion gearbeitet werden.

In Abbildung 10 ist die angepasste Architektur zu sehen. Dabei werden die Sensordaten von Gazebo generiert indem die virtuelle Drohne mit einem Blocklaserscanner überlagert wird. Es kann auch die Kamera der Drohne angesteuert werden um tatsächliche Bilddaten zu erhalten, jedoch beschränkt sich diese Arbeit darauf einen Stream von Punktwolken des Laserscanners zu verarbeiten.

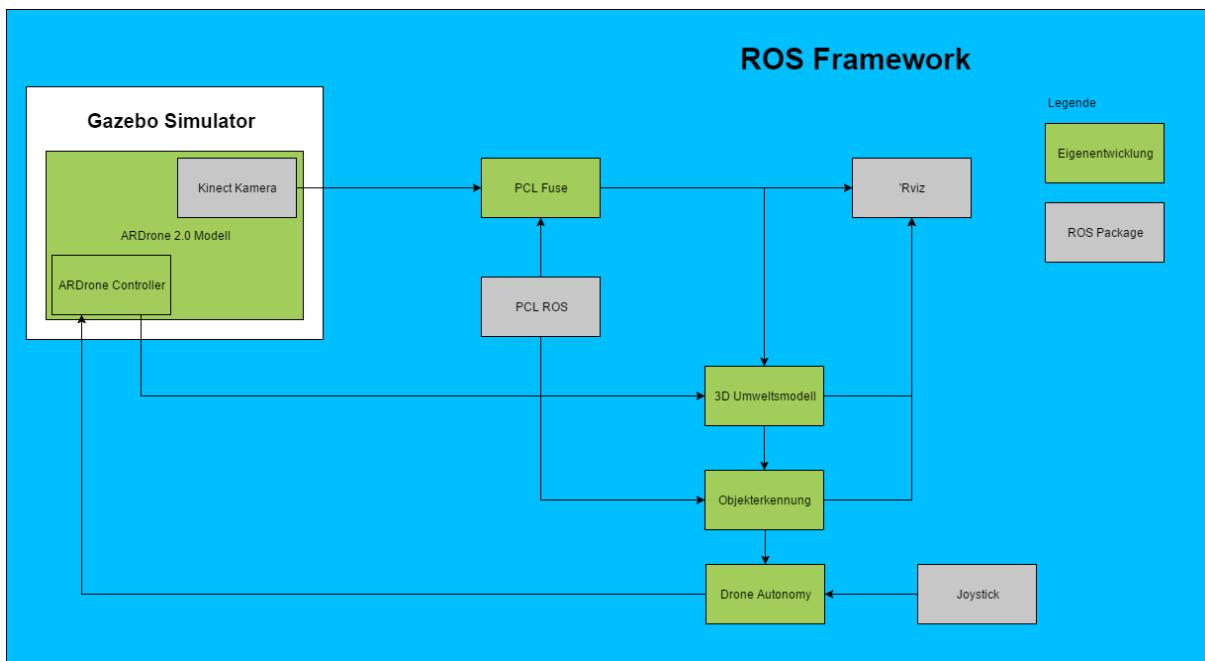


Abbildung 10: Vollvirtualisierte Architekturskizze

### 3.3 Aufbau der ROS-Pakete

Die ROS-Pakete sind hierarchisch aufgebaut um Übersichtlichkeit zu gewähren im Bezug auf die Verantwortlichkeiten nach dem Prinzip *Separation of Concerns*. Dies fördert vor allem die geringe Kopplung zwischen den Paketen sowie einen hohen inneren Zusammenhalt innerhalb der Pakete. Eine Aufteilung ist der Baumansicht in 11 zu entnehmen. Die Wurzel bildet dabei das Paket *Studienarbeit*, welches als einziges später vom Anwender angesprochen wird. Wie in Kapitel 2.3.4 beschrieben können Anwendungen innerhalb des ROS Frameworks auf unterschiedliche Arten gestartet werden. Aufgrund der hohen Anzahl beteiligter Nodes ist es sinnvoll mit Launchfiles zu arbeiten um eine korrekte Ausführung gewährleisten zu können.

Das Paket *Studienarbeit* beinhaltet drei dieser Launchfiles entsprechend der Aufgabenstellung der Studienarbeit. Alle drei Launchfiles starten die Simulation mit allen notwendigen Parametern, Plugins, Bibliotheken und Nodes. Der Unterschied liegt in der Bestückung der simulierten Welt. Es wird unterschieden zwischen einer leeren Standardwelt, einer Welt mit Säulen welchen es auszuweichen gilt und einer Welt mit Wänden und darin eingelassenen Türrahmen welche es zu durchqueren gilt.

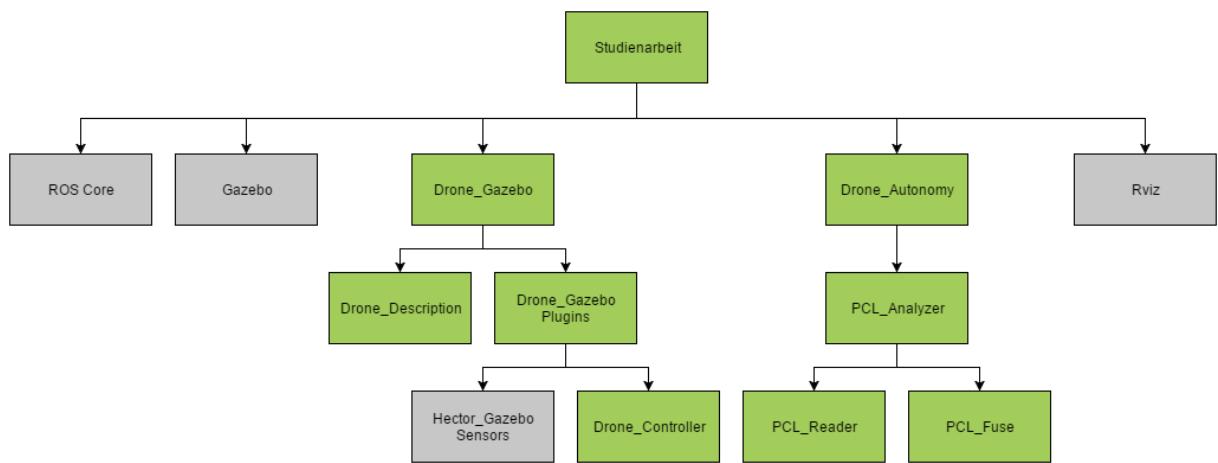


Abbildung 11: Baumansicht der Pakethierarchie

## 4 Implementierung der Simulation

DANIEL BETSCHE

Dieses Kapitel behandelt die Implementierungsdetails der Simulation. Wie bei den meisten Softwareprojekten, gibt es auch hier andere Projekte und Teams die an ähnlichen Problemstellungen interessiert sind. Bei der Suche nach Projekten, welche auch eine Simulation der ARDrone innerhalb von Gazebo inklusive Anschluss an das ROS Ökosystem umsetzen, sind zwei Projekte besonders hervorgestochen. Diese Projekte sind an anderen Universitäten durchgeführt worden und bestehen beide aus mehreren Bachelor- und Masterarbeiten. Aus beiden konnten viele Grundbausteine gewonnen werden auf denen die umgesetzte Lösung basiert, sowie auch wertvolle Erkenntnisse über die Eigenschaften der Simulationsumgebung selbst.

### 4.1 TU Darmstadt - Hector Quadrotor

Das Projekt der TU Darmstadt wird über Github unter der URL <https://github.com/tu-darmstadt-ros-pkg> veröffentlicht. Hier zu finden sind alle ROS Packages welche in wissenschaftlichen Arbeiten an der TU Darmstadt entwickelt wurden. Es sind knapp unter hundert Repositories vorzufinden mit unterschiedlichsten Einsatzgebieten und Anwendungsfällen. Das für die Simulation einer Flugdrohne interessante Repository nennt sich `hector_quadrotor`. Es enthält 11 Pakete mit den Daten für die Modellierung von Welten, Modellen, Controllern, Sensoren, Plugins und Eingabegeräten. Die hier modellierte Flugdrohne ist jedoch ein anderes Modell mit entsprechend anderen Flugeigenschaften. Die von Projekt Hector umgesetzten Flugeigenschaften zielen auf eine Hardwaresimulation ab. Das bedeutet, dass jeder einzelne Motor eines Quad-, Hexa- oder Octocopters ansteuerbar ist und damit der Realität sehr viel näher kommt als eine als homogener Körper betrachtete Drohne. Aufgrund dieses hohen Abstraktionslevels ist für den Einsatz der Controller im Rahmen der Studienarbeit ein gemeinsames Interface notwendig, welches das Abstraktionslevel auf eine höhere Ebene verlagert. Dieses Interface zu implementieren stellt allerdings eine komplexe und Zeitaufwändige Aufgabe dar. Daher wurde zuerst nach weiteren Projekten gesucht welche eventuell auch die selbe Hardware zur Verfügung hatten.

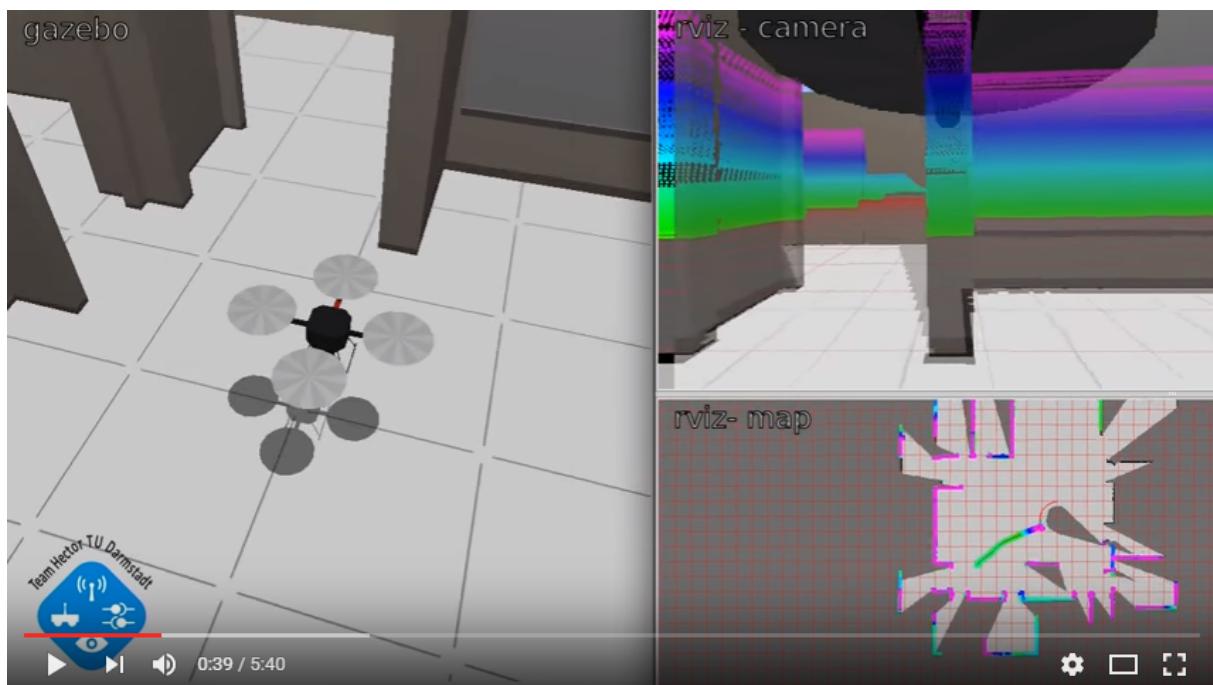


Abbildung 12: Beispiel hector\_quadrotor Stack mit SLAM  
Quelle: [youtube.com/watch?v=IJbJbcZVY28](https://www.youtube.com/watch?v=IJbJbcZVY28)

In Abbildung 12 zu sehen ist eine Simulation mit den Paketen von Hector und dem Stack mit SLAM.

**SLAM** kurz für Simultaneous Localization and Mapping bezeichnet ein Problem der Robotik, bei welchem ein mobiler Robotor gleichzeitig eine Karte seines umliegenden Terrains aufzeichnet und daraus seine relative Position bestimmen muss. Das Gebiet ist vorher unbekannt und der Robotor ist ausschließlich auf seine eigene Sensorik angewiesen. Dieser Anwendungsfall ist notwendig, da viele Einsatzgebiete für Roboter nicht vorher vermessen werden können. Szenarien in denen eine solche mobile Einheit eingesetzt werden kann variieren sehr stark in ihren Anforderungen. Diese reicht von einfachen Kostenfragen, wobei eine Vermessung schlichtweg zu teuer ist, über die Erkundung von Katastrophengebieten um das Einsatzrisiko für Menschen zu reduzieren, bis hin zu der Erkundung von anderen Planeten.

Als Nutzbare Komponenten für die Studienarbeit kann aus diesem Projekt die Modellierung der Welten und der allgemeinen Konfiguration der Simulation entnommen werden. Es gilt jedoch weiterhin das Modell des Quadrocopters zu ersetzen und die Flugeingeschaften korrekt nachzumodellieren um möglichst nah an die ARDrone 2.0 heranzukommen.

## 4.2 TU München - Simulator

Das ARDrone Projekt der TU München, auch TUM Simulator genannt, baut auch auf den Paketen der Universität Darmstadt auf um die Flugeigenschaften einer Drohne nachzubilden. Allerdings wurde das Drohnenmodell durch eine Nachbildung der ARDrone 2.0 ersetzt sowie auch die Controller welche als Plugins für die Steuerung eingesetzt werden umgeschrieben. Diese Controller wurden mit dem Blick auf das Paket ARDrone\_Autonomy erstellt und bieten die gleichen Schnittstellen für Steuerbefehle und zum Auslesen von Sensordaten an. Dadurch ist es möglich eine simulierte und eine echte Drohne mit den gleichen Befehlen zu steuern. Ein Beispiel dafür ist in Abbildung 13 zu sehen. Auf der linken Bildhälfte befindet sich die physische Drohne inklusive der Ausgabe ihrer Kamera, auf der rechten Seite die virtuelle Drohne. In dem Video selbst wird gezeigt, wie sich beide Drohnen mit nur einem Joystick gesteuert werden und synchron zueinander bewegen.



Abbildung 13: Beispiel Tum Simulator mit echter und virtualisierter Drohne  
Quelle: [youtube.com/watch?v=s\\_SBLeXRrhE](https://www.youtube.com/watch?v=s_SBLeXRrhE)

Aufgrund der gleichen Schnittstellen können auch alle Pakete, welche mit ARDrone Autonomy verwendet werden ohne weitere Anpassungen genutzt werden. Darunter unter anderem das Paket zur Steuerung per Joystick.

Als Nutzbares Material für die Studienarbeit kann aus diesem Projekt also die Implementierung der Controller übernommen werden, sowie das Modell für die Drohne und

die Steuerung. Das Problem allerdings ist, das alle diese Funktionen für das ROS Groovy Release geschrieben wurden und teilweise auf die Nachfolgerversion ROS Hydro portiert wurden. Wie bereits in 2.3.4 beschrieben ist das aktuelle Release ROS Jade und damit bereits der übernächste Release. Mit anderen Worten, der hier vorgenfundene Code des letzten Projektes, dass sich ausführlich mit der ARDrone 2.0 beschäftigt hat ist im Jahr 2012 geschrieben worden und erhielt geringfügige Aktualisierungen in 2014. Besonders gravierend ist der Fakt, dass über die letzten zwei veröffentlichten Versionen der Buildvorgang von *Rosbuild* auf *Catkin* umgestellt wurde.

### 4.3 Modellierung von Welt und Drohne

Aus den vorhandenen Vorlagen konnten viele Informationen gewonnen werden, allerdings sind viele dieser Informationen auf veralteten Versionen basiert. Aus diesem Grund mussten erst alle Grundlagen auf den bereitgestellten Tutorials für Gazebo und das ROS Ökosystem erarbeitet werden um einen Rahmen für den eigentlichen Inhalt zu bieten. Die Modellierung von Welten erfolgt in Gazebo mit dem XML-Format SDF (siehe 2.4.4). In den Listings 10 bis 12 ist die Beschreibung der Welt zu sehen, welche für den Use Case *Säulen ausweichen* genutzt wird.

Listing 10: Grundelemente einer Welt

```

1 <world name="pillars">
2   <include>
3     <uri>model://sun</uri>
4   </include>
5   <include>
6     <uri>model://ground_plane</uri>
7   </include>
```

Dieser Abschnitt des SDFs enthält die Definition der Welt und fügt zwei Modelle ein. Diese sind zum einen eine Sonne, welche zwar nicht zwingend notwendig ist, allerdings eine einfache Lichtquelle liefert welche für Kameraaufnahmen relevant ist. Das andere eingefügte Modell ist eine Ebene Fläche, welche als Boden dient. Ohne diese Fläche würden durch die Gravitation alle Objekte ins leere fallen. Sie dient uns außerdem für alle weitergehenden Betrachtungen als Ebene in XY-Richtung mit z=0.

Listing 11: Definition einer einzelnen Säule

```

1 <include>
2   <uri>model://pillar</uri>
```

```

3   <pose>1.0 1.0 1.0 0 0 0</pose>
4   </include>
```

Danach folgen einige Modelle, welche die Säulen darstellen denen es auszuweichen gilt. Die hier eingebundenen Modelle sind selbst wieder SDF Dokumente, die weitere Details enthalten. Durch das attribut `<pose>` wird für die eingefügte Säule die Position innerhalb der Welt festgelegt. Die Dokumente sind alle so aufgebaut, damit eine Wiederverwendung einfacher ist und Codeduplizierungen minimal gehalten werden. Neben dem Einfügen von Modellen in die Welt können auch Elementare Einstellungen geändert werden. Wie in Listing 12 zu sehen werden hier die physikalischen Grundlagen und Genauigkeit der simulierten Welt festgelegt. Für die meisten Anwendungen genügen hier die Standardwerte.

Listing 12: Physikalische Konfiguration der Welt

```

1 <physics name='default_physics' default='0' type='ode'>
2   <max_step_size>0.001</max_step_size>
3   <real_time_factor>1</real_time_factor>
4   <real_time_update_rate>1000</real_time_update_rate>
5   <gravity>0 0 -9.8</gravity>
6   <magnetic_field>5.5645e-06 2.28758e-05 -4.23884e-05</magnetic_field>
7 </physics>
```

Die Modellierung der Drohne ist im Gegensatz zu den Welten deutlich aufwändiger, da hier nicht auf allgemeine Standards zurückgegriffen werden kann. Es gilt für den Drohnenkörper und alle daran befestigten Erweiterungen wie etwa die Sensoren die entsprechenden Parameter anzugeben. Darunter fallen wie bereits in 2.4.2 besprochen alle Links und Joints. Zu jedem Link wird die Masse, der Trägheitsvektor, sowie eine Beschreibung der Sichtbaren Elemente und eine Beschreibung der Kollision gefordert. Für viele Objekte in Gazebo ist es am einfachsten mesh-Dateien zu nutzen, welche mit 3D-Modellierungsanwendungen wie dem Open-Source Tool Blender erzeugt werden können. Es ist allerdings auch möglich Objekte komplett in SDF zu beschreiben, der Aufwand dafür ist jedoch unverhältnismäßig hoch.

Listing 13: Drohnenkörper

```

1 <xacro:macro name="drone_base_macro">
2   <link name="base" />
3
4   <joint name="base_joint" type="fixed">
5     <parent link="base" />
```

```

6   <child link="base_link" />
7 </joint>
8
9 <link name="base_link">
10 <inertial>
11   <mass value="1.477" />
12   <origin xyz="0 0 0" />
13   <inertia ixx="0.01152" ixy="0.0" ixz="0.0" iyy="0.01152"
14     iyz="0.0" izz="0.0218" />
15 </inertial>
16
17 <visual>
18   <origin xyz="0 0 0" rpy="0 0 0" />
19   <geometry>
20     <mesh filename="package://studienarbeit/models/quadrotor_4.dae" />
21   </geometry>
22 </visual>
23
24 <collision>
25   <origin xyz="0 0 0" rpy="0 0 0" />
26   <geometry>
27     <mesh filename="package://studienarbeit/models/quadrotor_4.stl" />
28   </geometry>
29 </collision>

```

In Listing 13 ist zu sehen, wie der Drohnenkörper definiert wird. Das Modell selbst wurde aus dem Projekt der TU Darmstadt übernommen und ist ein mit Blender hergestelltes mesh. Die Unterscheidung von sichtbarem Modell und seiner Kollision ist in der notwendigen Rechenzeit begründet. Meist wird ein Kollisionsmodell gegenüber den sichtbaren Elementen vereinfacht um die Simulationszeit zu reduzieren. Es ergibt sich, dass je komplexer das Kollisionsmodell ist, desto aufwändiger wird die Berechnung ausfallen.

## 4.4 Gazebo Plugins

Die Steuerung und Sensoren der Drohne werden über Gazebo Plugins realisiert, welche über ein Include in der drone\_body SDF hinzugefügt werden. Auch hier ist wieder

der Grund, dass Code und Konfiguration wiederverwendet werden können und die Kopplung zwischen Modulen möglichst gering gehalten wird.

Listing 14: Include der Plugins

```
1 <xacro:include filename="$(find studienarbeit)/urdf/plugins/
drone_plugins.urdf.xacro" />
```

#### 4.4.1 Controller

Die Controller dienen zur Manipulation des Modells innerhalb von Gazebo. Der eigentliche Code des Plugins befindet sich in den Dateien *quadrotor\_simple\_controller.cpp* und *quadrotor\_state\_controller.cpp*. Diese wurden aus dem Projekt der TU München übernommen und aktualisiert, sodass sie mit dem aktuellen Release Jade kompatibel sind. Das Produkt nach dem Kompilieren sind zwei Bibliotheken, welche innerhalb der Plugin-Tags referenziert werden.

In Listing 15 ist zu sehen, wie der State-Controller eingebunden und konfiguriert wird. Er kontrolliert anhand der Daten aus der Simulationsumgebung und der Sensoren die Position des Quadcopters. Zudem kann er diese Position manipulieren über Nachrichten auf der Topic *cmd\_vel*. Der Simple-Controller wird analog eingebunden, besitzt jedoch einen größeren und ergänzenden Funktionsumfang. Beide Controller zusammen sorgen für eine originalgetreue Nachbildung der Flugeigenschaften der echten ARDrone 2.0.

Listing 15: Konfiguration des State-Controllers

```
1 <plugin name="quadrotor_state_controller" filename="
libhector_gazebo_quadrotor_state_controller.so">
2   <alwaysOn>true</alwaysOn>
3   <updateRate>0.0</updateRate>
4   <bodyName>base_link</bodyName>
5   <stateTopic>ground_truth/state</stateTopic>
6   <imuTopic>ardrone imu</imuTopic>
7   <sonarTopic>sonar_height</sonarTopic>
8   <topicName>cmd_vel</topicName>
9 </plugin>
```

#### 4.4.2 Sensoren

Neben den Controllern werden auf diese Art weitere Sensoren eingebunden. In Listing 16 ist als Beispiel dafür der GPS Sensor aufgezeigt. Im Unterschied zu den Controllern, welche selbst geschrieben und kompiliert werden müssen, gibt es hierfür bereits vorgefertigte Bibliotheken. Dieses Plugin stammt wie nahezu alle verwendeten Sensoren aus dem Projekt Hector der TU Darmstadt. Diese stehen als Pakete im Ubuntu Repository von ROS zum Download bereit.

Listing 16: Einbinden eines GPS Sensors

```

1 <plugin name="drone_gps_sim" filename="libhector_gazebo_ros_gps.so">
2   <alwaysOn>true</alwaysOn>
3   <updateRate>4.0</updateRate>
4   <bodyName>base_link</bodyName>
5   <topicName>fix</topicName>
6   <velocityTopicName>fix_velocity</velocityTopicName>
7   <drift>5.0 5.0 5.0</drift>
8   <gaussianNoise>0.1 0.1 0.1</gaussianNoise>
9   <velocityDrift>0 0 0</velocityDrift>
10  <velocityGaussianNoise>0.1 0.1 0.1</velocityGaussianNoise>
11  </plugin>
```

Insgesamt ist der Quadrocopter mit folgenden Sensoren ausgestattet:

- Trägheitssensor
- Barometer
- Magnetometer
- GPS
- Front- und Boden-Kamera
- Sonar
- Kinect-Kamera

Zusätzlich hinzugefügt im Vergleich zum Projekt der TU München wurde die Kinect-Kamera. Diese dient als Quelle für Punktwolken und Tiefenmessdaten. Im Gegensatz zu einer echten Kinect wurden die Dimensionen des Modells auf einen Würfel mit einer Kantenlänge von einem Zentimeter reduziert, welcher parallel zur Frontkamera an der

Vorderseite des Quadrocopters angebracht wurde. Die Kinect ermöglicht es den Verarbeitungsschritt zu überspringen, in dem mit der Software der Firma Myestro Punktwolken aus Kameradaten erzeugt werden. Aus diesem Grund ist es dementsprechend sinnvoll in der Modellierung auf ein ausgedehntes physisches Modell zu verzichten. Die Konfiguration der Kameraaufnahmen wird ausführlich im folgenden Abschnitt diskutiert.

## 5 Aufnahme und Verarbeitung der Kameradaten

Dieses Kapitel behandelt alle notwendigen Schritte zur Aufnahme und Vorverarbeitung der Kameradaten bevor sie in ein 3D-Umweltsmodell überführt werden können. Der Hauptaugenmerk liegt dabei auf den Differenzen der zwei unterschiedlichen Datenquellen, der echten Drohne und der Simulation. Die Problemstellungen ähneln sich zwar, besitzen allerdings sehr verschiedene Ausprägungen in den Bereichen der Bildqualität und der Anzahl an Vorverarbeitungsstufen.

### 5.1 Physische Drohne

CHRISTIAN VERDION

Der Versuchsaufbau sieht vor, dass die Drohne durch einen präparierten Raum fliegt, in dem sich Säulen befinden. Die Drohne filmt mit ihrer Kamera die Außenwelt, woraufhin mithilfe der Software von Myestro jeweils ein Tiefenbild aus 3 aufeinander folgenden Bildern entsteht. Dabei gilt zu beachten, dass sich die Drohne dabei bewegen muss, da nur durch die verschiedenen Perspektiven aus verschiedenen Positionen Tiefendaten erstellt werden können. Hierfür sollte ein bestimmtes Bewegungsmuster implementiert werden, wodurch die Drohne in allen 3 Dimensionen die Position verändert. Dieses Bewegungsmuster ähnelt einem Korkenzieher, da der Quadcopter, während er nach vorn fliegt nach oben und unten, sowie rechts und links fliegt. Daraus ergeben sich viele Bilder, die in die gleiche Richtung zeigen, aber aus leicht veränderten Winkeln aufgenommen wurden. Die aufgenommenen Bilder werden anschließend an der Computer übertragen, auf dem die Steuerung und 3D-Modellierung läuft. Die Bilder werden nun vorverarbeitet. Dabei werden die intrinsischen Parameter, die zuvor bestimmt wurden ausgelesen, um die Bilder daraufhin zu entzerren. Als nächsten Schritt können die Bilder an die Software von Myestro übertragen werden, woraufhin die Tiefendaten oder PointClouds ausgegeben werden sollen.

#### 5.1.1 Zusammenarbeit mit der Firma Myestro

In dieser Studienarbeit wurde darauf gesetzt von der Firma Myestro eine Software zu bekommen, die den Rubbertrion Algorithmus implementiert. Dieser Algorithmus ist eine firmeninterne Entwicklung, die es ermöglichen soll aus drei Bildern von einem Motiv aus verschiedenen Perspektiven die Tiefendaten zu extrahieren. Das Besondere daran

ist, dass für die gewöhnliche Extraktion von Tiefendaten zwei Bilder verwendet werden, aus denen schließlich ein stereoskopisches Bild generiert wird, woraus wiederum die Tiefendaten abgeleitet werden können. Hierbei ist zu beachten, dass zum generieren der Bilder ein fester, und bekannter Abstand zwischen den beiden Linsen herrschen muss um die Daten zweifelsfrei zu berechnen. Ein Problem, dass sich dabei ergibt ist, dass man dadurch 2 Kameras benötigt. Die Drone verfügt allerdings nur über eine Kamera, die auf das selbe Motiv gerichtet ist.

Anfangs wurde über eine Lösung diskutiert, die es vorsah zwei Dronen nebeneinander Fliegen zu lassen. Das Problem dabei bestand aber darin, dass der Abstand der beiden Dronen untereinander bekannt sein muss. Denn selbst mit dem optional verfügbaren Global Positioning System Sensor wäre eine Positionsbestimmung auf 20m zu ungenau gewesen.

Die Alternative bestand nun daraus mit einer Drohne drei Bilder von einem Motiv zu erstellen. Der Rubbertrion Algorithmus sieht es wie bereits beschrieben vor, aus 3 Bildern Tiefendaten zu erstellen und das ohne den Abstand der Bilder zu wissen. Genaue Implementierungsdetails hat Myestro unter Verschluss behalten.

Leider war die Implementierung firmenintern lediglich als Studentenprojekt angesetzt was zur Folge hatte, dass die Software zum einen lange auf sich hat warten lassen und zum anderen nicht effizient implementiert wurde. Das hat zur Folge, dass zur Ausführung der Software bestimmte Hardwarekomponenten vorgesehen sind. Darunter ist ein bestimmter Prozessor von Intel und ein bestimmtes Grafikkartenmodell von AMD. Unter Verwendung des korrekten Intel Prozessors sei es zwar eventuell möglich den eingebauten Grafikchip zu verwenden, jedoch benötigt selbst die leistungsstarke AMD Grafikkarte mehr als vier Sekunden zur Berechnung aus einem Bildertriplets. Als diese Daten bekannt wurden musste eine alternative Lösungsstrategie gesucht werden und dafür bot sich eine vollständig virtualisierte Lösung in einer Simulation an.

## 5.2 Simulation

DANIEL BETSCHE

Die Simulierte Drohne besitzt wie in Kapitel 4.4.2 beschrieben zusätzlich zu den nachmodellierten Sensoren eine Kinect Kamera direkt über der Frontkamera. Für die Simulation ist es hier wichtig die korrekten Werte für alle relevanten Parameter einzustellen. Folgende sind hier zu beachten:

- Bildwiederholrate
- Horizontales Sichtfeld
- Bildauflösung
- Farbformat
- Minimaler und Maximaler Bildausschnitt
- Rauschen
- Linsenverzerrung
- Linsentyp
- Brennweite
- Ausgabeformat

Die Konfiguration der Parameter erfolgt durch Beschreibung in dem für Gazebo typischen XML-Format SDF. Das Hauptelement ist vom Typ `<sensor>` und enthält in diesem Fall drei Kindelemente. Dies sind im Kontext der Studienarbeit immer die Bildwiederholrate unter `<update_rate>` die Konfiguration der Kamera unter `<camera>` sowie die Einstellungen für das angewandte Plugin unter `<plugin>`.

Listing 17: Gazebo Sensor Konfiguration

```
1 <sensor type="depth" name="${name}_camera_sensor">
2   <update_rate>20</update_rate>
```

Die Bildwiederholrate setzen wir fest mit 20Hz. Dieser Wert wurde gewählt um einen Kompromiss zwischen der Genauigkeit der Steuerung und der verfügbaren Hardware herzustellen. Mit einer höheren Bildwiederholrate kann die Steuerung feiner justiert werden und besser auf schnelle Bewegungen reagieren. Bei einer Flugdrohne ist dies nicht ungewöhnlich und ein elementarer Faktor der Handhabung. Auf der anderen Seite wiederum ist für eine sehr hohe Bildwiederholrate auch dementsprechend Leistungsstarke Hardware notwendig um die Verarbeitung durchführen zu können. Da die verfügbaren Mittel eingeschränkt sind auf handelsübliche Verbraucherhardware wurde die oben genannte Rate gesetzt.

Listing 18: Kameraspezifische Konfiguration

```
1 <camera>
2   <horizontal_fov>${60*pi/180}</horizontal_fov>
3   <image>
```

```

4   <width>640</width>
5   <height>480</height>
6   <format>R8G8B8</format>
7   </image>
8   <clip>
9     <near>0.05</near>
10    <far>10</far>
11  </clip>
12  <noise>
13    <type>gaussian</type>
14    <mean>0</mean>
15    <stddev>1</stddev>
16  </noise>
17 </camera>

```

Unter dem Element `<camera>` wird das horizontale Sichtfeld, die Bildauflösung, das Farbformat sowie das Rauschen festgelegt. Die Konfiguration dieser Parameter erfolgt wie in Listing 18 zu sehen. Das Sichtfeld wird auf einen Öffnungswinkel von 60 Grad festgelegt. Die Bildauflösung wird recht klein gewählt zu 640x480 aufgrund der Tatsache das die Flugfähigkeit der Drohne auf eine Ebene von ungefähr einem Meter über dem Boden beschränkt wird. Für die beiden zu erreichenden Use-Cases ist eine höhere Auflösung nicht notwendig. Sollten die Ergebnisse dieser Studienarbeit in folgenden Projekten genutzt werden kann dieser Wert jedoch sehr einfach korrigiert werden. Farben werden im 8bit RGB Format aufgenommen. Dies ist der Standardwert und es bestand kein Grund ihn abzuändern. Das simulierte Rauschen wird in Form einer Standardnormalverteilung realisiert, also einer Gaußverteilung mit dem Erwartungswert  $\mu = 0$  und einer Varianz  $\sigma^2 = 1$ .

An dieser Stelle innerhalb des SDF Dokuments besteht auch die Möglichkeit die Linsenverzerrung und die Linsenart festzulegen. Es fällt auf, dass diese Elemente nicht in der Konfiguration vorkommen. In diesem Fall werden Standardwerte angenommen. Ein Anwendungsszenario für Werte, die vom Standard abweichend ist unter anderem das Testen der Linsenkorrektur von Myestro's Rubbertrion. Sollte eine folgende Arbeit diese Software benutzen können hier Messwerte der echten Linse zu Testzwecken übernommen werden.

Listing 19: Konfiguration des Sensor-Plugins

```

1 <plugin name="${name}_camera_controller" filename="
  libgazebo_ros_openni_kinect.so">

```

```

2  <cameraName>/${sim_name}/${name}</cameraName>
3  <alwaysOn>true</alwaysOn>
4  <updateRate>20</updateRate>
5  <imageTopicName>/${sim_name}/${name}/rgb/image_raw</imageTopicName>
6  <cameraInfoTopicName>/${sim_name}/${name}/rgb/camera_info
7  </cameraInfoTopicName>
8  <depthImageTopicName>/${sim_name}/${name}/depth/image_raw
9  </depthImageTopicName>
10 <depthImageCameraInfoTopicName>/${sim_name}/${name}/depth/camera_info
11 </depthImageCameraInfoTopicName>
12 <pointCloudTopicName>/${sim_name}/${name}/depth/points
13 </pointCloudTopicName>
14 <frameName>${name}_link</frameName>
15 <pointCloudCutoff>0.4</pointCloudCutoff>
16 <pointCloudCutoffMax>50</pointCloudCutoffMax>
17 <hackBaseline>0.07</hackBaseline>
18 <distortionK1>0.0</distortionK1>
19 <distortionK2>0.0</distortionK2>
20 <distortionK3>0.0</distortionK3>
21 <distortionT1>0.0</distortionT1>
22 <distortionT2>0.0</distortionT2>
23 <CxPrime>0.0</CxPrime>
24 <Cx>0.0</Cx>
25 <Cy>0.0</Cy>
26 <focalLength>0.0</focalLength>
27 </plugin>

```

Das Plugin-Element wie man es in Listing 19 sieht, bindet eine Bibliothek ein, welche die Funktionen des zu Simulierenden Plugins in C++ Code umgesetzt beinhaltet. Darin wird das Verhalten des Sensors beschrieben und die notwendigen Berechnungen durchgeführt, um aus den Simulationsdaten die gewünschten Sensordaten zu generieren. Ein solches Plugin greift dazu auf die API der Simulationsumgebung zu um alle notwendigen Informationen zu erhalten. Aus diesen werden die erwarteten Datentypen generiert und an den gewünschten Schnittstellen zur Verfügung gestellt. Alle dynamischen Parameter werden innerhalb des SDF Dokuments festgelegt und an das Plugin weitergeleitet.

Der Sensor wird hier so eingestellt, dass die Aktualisierungsrate mit dem im Camera-

Element festgelegten Wert übereinstimmt und keine Verzerrung auftritt. Die simulierte Kinect Kamera bietet an dieser Stelle drei unterschiedliche Ausgaben mit den entsprechenden Datentypen. Diese werden auf dazu passende ROS-Topics ausgegeben. Parallel dazu erhält jede Ausgabe auch eine Info-Topic welche Informationen beinhaltet, wie etwa die Verzerrungsfaktoren und die intrinsische Kameramatrix für die Rohdaten. Die Variablen in dem SDF Dokument werden bei der Kompilierung so ersetzt, dass die resultierenden Topics wie folgt ausfallen:

Format	ROS Topic
RGB Bild	/ardrone/ray/rgb/image_raw
Tiefenbild	/ardrone/ray/depth/image_raw
PointCloud	/ardrone/ray/depth/points

Tabelle 1: Zuordnung der Ausgabeformate zu ROS Topics

Damit kann die simulierte Kinect Kamera zum einen die Frontkamera ersetzen, der Grund für die Wahl einer Kinect allerdings ist die Ausgabe der PointClouds. Diese entsprechen der eigentlichen Ergebnisse der Rubbertrion Software und dienen als Eingabe für die Objekterkennung.

## 6 Aufstellen und Auswerten eines 3D-Umweltmodells

CHRISTIAN VERDION

### 6.1 Vorverarbeitung der Tiefendaten

Um den Versuchsaufbau so realistisch wie möglich durchzuführen wurde beschlossen bei den Einstellungen der Kinect zufällig Ausreißer hinzuzufügen. Diese können bei realen Bildern eine Vielzahl von Ursachen haben. Denn durch Komponenten wie Verstärker, CCD Sensor, Software wie eine Bildkompression oder andere Einflüsse wie Nebel werden die Bilddaten zusätzlich verfälscht. Wenn die Bilder nun nicht mehr der Realität entsprechen können auch gute 3D-Rekonstruktionsverfahren die Tiefendaten nicht mehr einwandfrei berechnen. Aus diesem Grund entsteht aus natürlichen Bildern immer sogenannter „Noise“. Also Lärm, durch den es schwieriger wird, die tatsächlichen Daten zu verstehen. Je höher der Noisefloor ist, also der Grundpegel, auf dem die Daten verrauscht sind, desto schwieriger wird es auf die tatsächlichen Daten zurückzuschließen zu können. Um diesen Lärm dennoch zu entfernen ist es möglich eine statistische Analyse der Nachbarschaftsbeziehung der einzelnen Punkte durchzuführen. Dabei wird für jeden Punkt der Mittelwert des Abstands zu all seinen Nachbarpunkten berechnet. Mithilfe der Gaußverteilung, dessen Mittelwert und der Standardabweichung wird ein Schwellwert berechnet. Liegt der Mittelwert des Punktes über diesem Schwellwert, so ist es sehr unwahrscheinlich, dass dieser Punkt ein Tatsächlich aufgenommener Wert ist, sondern lediglich eine Fehlinterpretation oder ein Rechenfehler ist. Die resultierende Punktwolke kann dabei helfen, dass keine Fehler bei der Zusammensetzung der einzelnen PointCloud Daten auftreten.

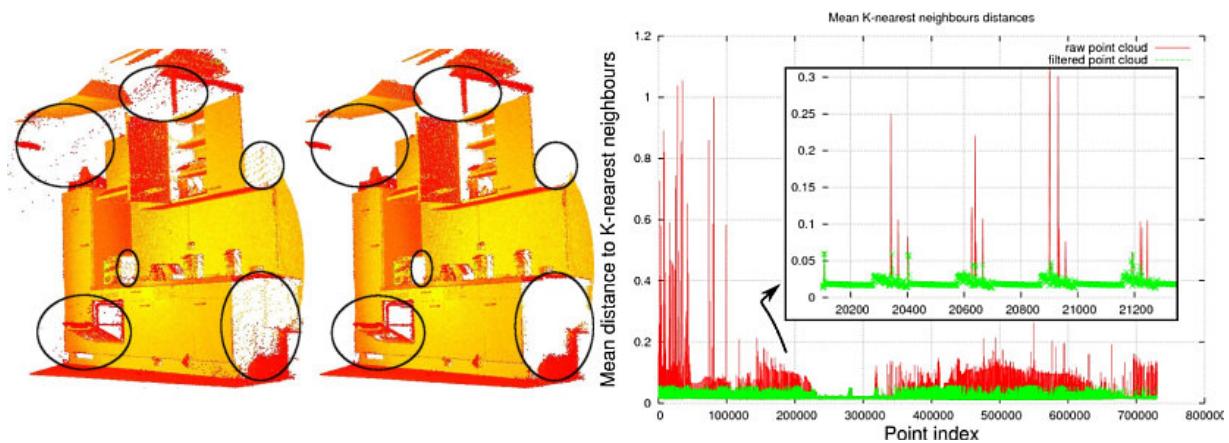


Abbildung 14: Ein verrausches Tiefenbild, das durch Vorverarbeitung bereinigt wurde[17]

## 6.2 Zusammensetzung der 3D Ausschnitte zu einer Szene

Wie zuvor beschrieben, werden alle PointCloud Daten der Kinect über das Topic „/ardrone/ray/depth/points“ übertragen. Um diese zu empfangen genügt es einen Abonenten oder „Subscriber“ zu registrieren. Dieser ruft eine CallBack Methode auf, sobald eine Message in diesem Topic versendet wird. Der Abonnent ist auch mit einem Puffer ausgerüstet, mit dessen Hilfe er die letzten Daten vorhalten kann, sodass die Daten in der eintreffenden Reihenfolge verarbeitet werden können. Sollte der Puffer allerdings voll sein, so gehen die weiter folgenden Nachrichten verloren. Dieser Puffer kann für jeden Abonnenten individuell eingestellt werden, falls sich deren Verarbeitungsgeschwindigkeit voneinander unterscheidet. Zusätzlich zu dem Abonnenten muss auch ein „Advertiser“ registriert werden, der die zusammengesetzten 3D Szenen, die ebenfalls eine PointClouds ist, veröffentlicht. Auch für diese Methode ist ein Puffer vorgesehen, falls die Daten schneller versendet werden sollen, als das Netzwerk gestattet. Dabei sollte man bedenken, ob der Empfänger besser alte zusammenhängende Daten empfangen soll, oder es wichtig ist, dass er immer die neuste Nachricht haben soll. In diesem Fall ist es wichtig, dass die neuste Szene versendet wird, da diese in der Regel die selben oder sogar mehr Daten erhalten als die Nachricht zuvor. Nachdem die PointCloud Daten der Drohne empfangen werden können, müssen diese Verarbeitet werden, um sie anschließend mit dem Advertiser an den nächsten Verarbeitungsschritt zu senden. Da der Abonnent eine CallBack Methode übergeben bekommen hat kann die Hauptbefehlsschleife ruhen bis eine Nachricht eintrifft. Sobald das der Fall ist wird die Methode aufgerufen und die übertragene PointCloud wird in die Szene mit aufgenommen.

### 6.2.1 Iterative Closest Point Algorithmus

Der Iterative Closest Point Algorithmus wird dazu verwendet PointCloud Daten zu rekonstruieren. Er wird häufig eingesetzt, um mehrere Tiefendaten zusammenzusetzen, die zum Beispiel von einem Objekt aus verschiedenen Perspektiven gemacht wurden, wie man in Abbildung 15 erkennen kann.

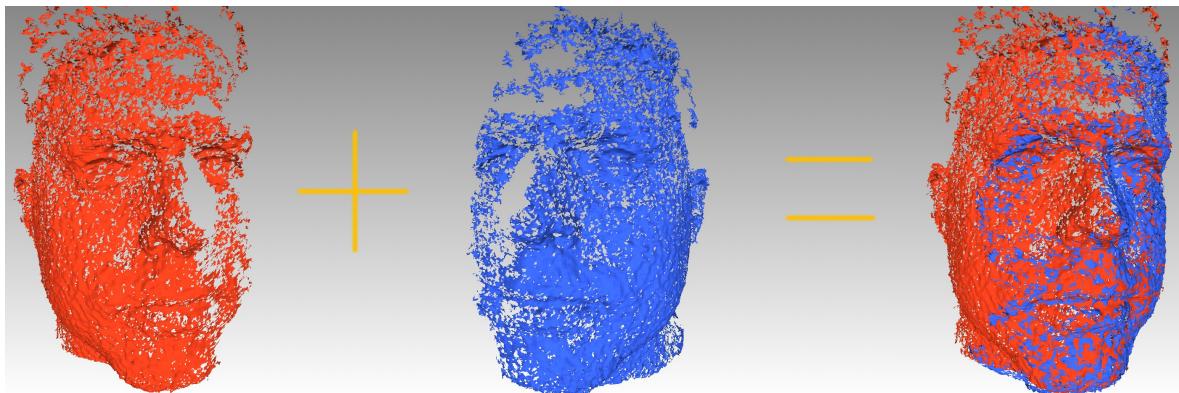


Abbildung 15: Tiefenbilder von einem Gesicht zusammengesetzt zu einem Ganzen [27]

Der Algorithmus ist ebenfalls einsetzbar um einzelne Segmente eines Raumes zu einem Gesamtbild beziehungsweise einer Gesamtszene zusammenzufügen wie es für diese Studienarbeit nötig war. Hierzu betrachtet man zunächst die einzelnen PointClouds individuell. In Abbildung 18 sind 6 verschiedene PointClouds zu sehen. Um diese zusammenzusetzen werden immer 2 Datensätze miteinander verbunden. Dabei wird zu jedem Punkt des einen Datensatzes der nächstgelegene Punkt des anderen Datensatzes gesucht. Dabei wird, wie in Abbildung 17 gezeigt, versucht die Punktwolken in jeder Iteration näher zusammenzuführen, bis deren Abstand unter einem bestimmten Epsilon liegt. Für dichtbesetzte Punktwolken kann dies sehr zeitaufwendig werden. Aus diesem Grund werden bei großen Datensätzen sogenannte „FeaturePoints“ berechnet. Also Merkmale, die sehr markant für die Umgebung sind. Wenn genügend „FeaturePoints“ für beide PointClouds gefunden wurden wie in Abbildung ??, so wird für diese Merkmale der iterative Prozess durchgeführt.

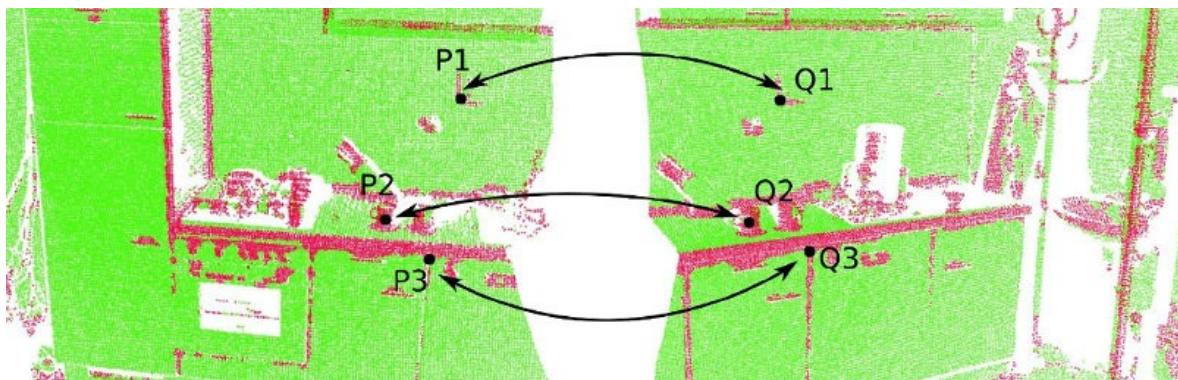
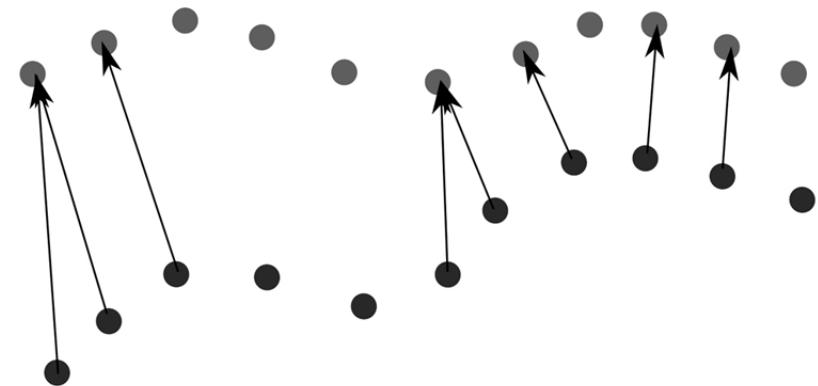
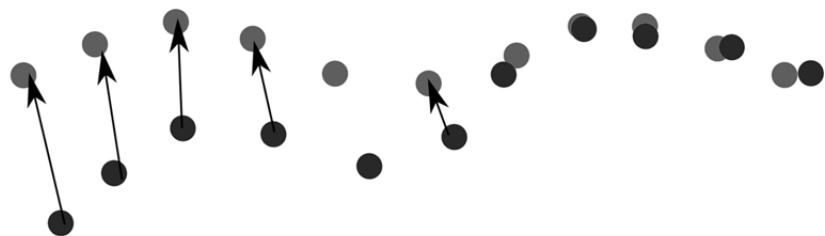


Abbildung 16: Zwei PointClouds mit markierten FeaturePoints [18]

### Itération 1



### Itération 2



### Itération 3



Abbildung 17: Iteration über die nächstgelegenen Punkte [14]

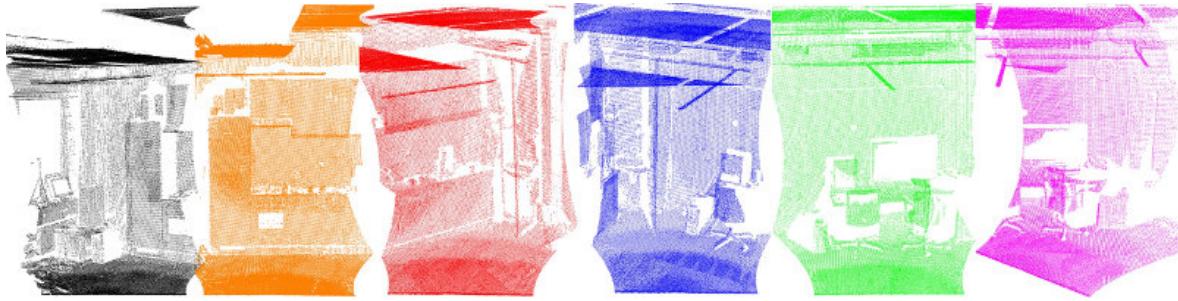


Abbildung 18: Tiefenbilder von einem Raum aus verschiedenen Perspektiven [19]

Am Ende des Prozesses muss evaluiert werden, ob die Punkte oder FeaturePoints nah genug sind und es somit zu einem Match kam oder ob die Iterationen ohne das gewünschte Kriterium zu erfüllen beendet wurde. Wenn es zu einem Match kam, so werden die berechneten Transformation und Rotation auf den hinzuzufügenden Datensatz angewendet. Das Ergebnis ist schließlich eine fertige Punktfolke, die im Grunde genommen, der in Abbildung 19 Ähnelt.

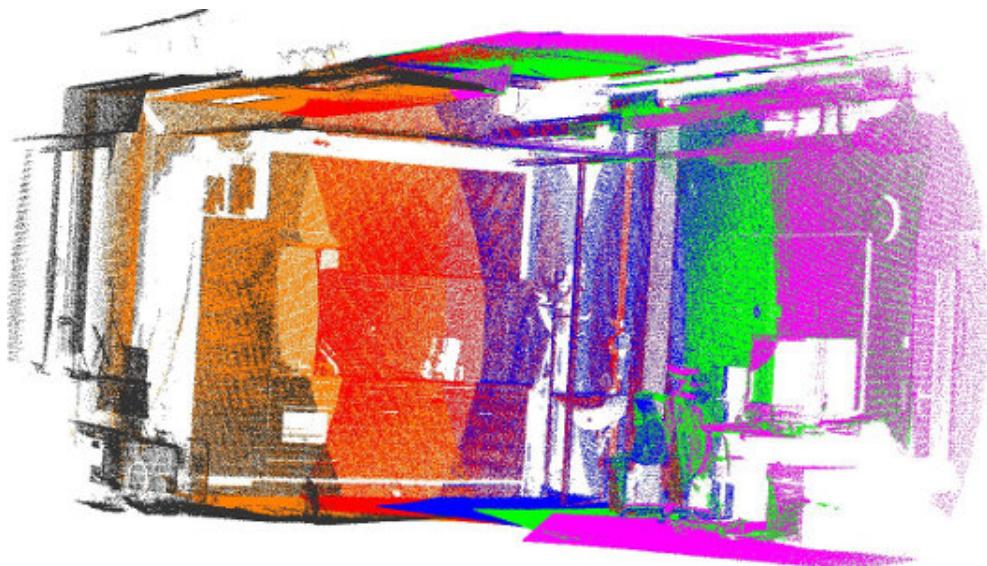


Abbildung 19: Tiefenbild von einem Raum aus verschiedenen Perspektiven zusammengesetzt [20]

### 6.3 Objekterkennung im drei Dimensionalen Raum

Als finaler Schritt bei der Verarbeitung der Punktfolken muss erkannt werden, wo sich eine Tür und wo sich eine Säule befindet. Dabei werden zunächst die Kanten erkannt. Daraus kann bestimmt werden wo sich die Objekte befinden. Daraufhin muss das Objekt korrekt erkannt werden. Zur Vereinfachung der Aufgabe wurde definiert, dass bei

einer Tür die beiden Türrahmen rechts und links ein Meter voneinander entfernt sind. Die dicke einer Säule hingegen ist 50cm. Zunächst müssen also die verschiedenen Kannten im 3-Dimensionalen Bild erkannt werden. Dabei existieren verschiedene Arten von Kanten wie in Abbildung 20 zu sehen ist. Die oberste Kante ist erkennbar aufgrund der Änderung der Oberflächennormalen. Bei der Oberflächennormalen handelt es sich um den Normalenvektor, also der Senkrechten, die von der Fläche ausgeht, die betrachtet wird. Ändert sich dieser Vektor abrupt, so kann man von einer Kante zwischen zwei Flächen ausgehen. Ändert er eher sich stetig, so ist es sehr wahrscheinlich, dass es sich lediglich um eine gekrümmte Fläche handelt. Bei der nächsten Kante handelt es sich um einen Sprung der Tiefenkoordinate. Diese Art von Kanten sind in Punktwolken einfacher zu erkennen als in einfachen Bildern. Als nächstes erfolgt eine Kante, aufgrund der Änderung der Farbe. Diese Methode wird oft angewendet, wenn es keinen Unterschied in den Tiefendaten gibt. Also zum Beispiel bei geschriebenen Text oder Objekten deren Tiefe aufgrund von Messtoleranzen nicht wahrgenommen werden. Zuletzt folgt eine Änderung in der Beleuchtungsfunktion. Damit werden in der Regel Schatten erkannt, es kann dadurch allerdings auch zu Fehlinterpretationen von Kanten führen, falls das Objekt teilweise im Schatten eines anderen Objektes liegt.

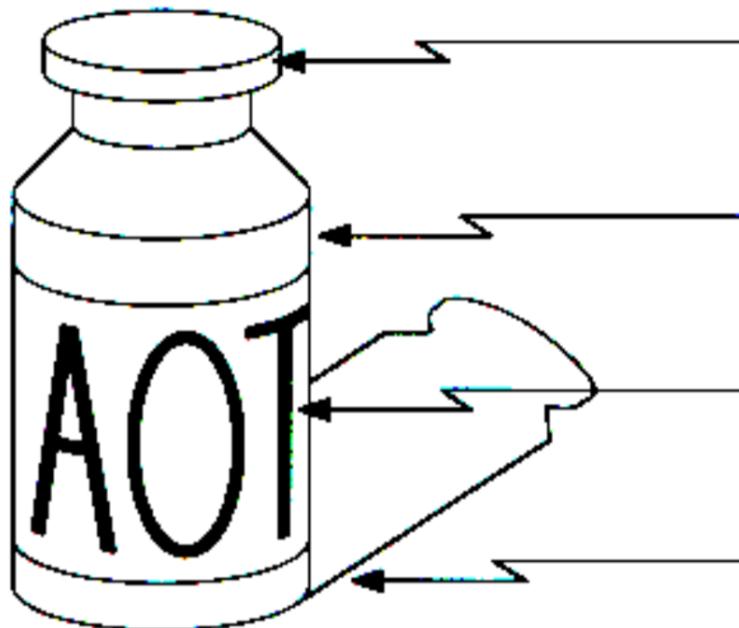


Abbildung 20: Die verschiedenen Arten von Kanten [4]

Um Kanten in diesem Versuchsaufbau zu erkennen genügt es sich mit den nicht stetigen Änderungen der Tiefendaten zu beschäftigen, da diese sehr aussagekräftig sind. Die Unterschiede verschiedener Oberflächennormalen sowie die des Schattens und

des Farbwertes sind nicht nötig. Glücklicherweise existieren bereits zahlreiche Implementierungen zur Kantenerkennung, sowie Implementierungen, die sich die Open Point Cloud Library zunutze machen. Die Höhe des Türrahmens ist vorerst auch irrelevant, da der Sensor zur Messung der Flughöhe lediglich dafür genutzt wird die Drohne auf einem Meter Höhe zu halten. Die eingesetzte Implementierung ist in der Lage die geforderten Kanten des Tiefendatensprungs zu erkennen, in Abbildung 21 grün dargestellt. Zusätzlich können damit die Kanten der Schatten und die der Krümmungen der Oberflächen, also der Änderungen der Oberflächennormalen erkannt werden. Diese werden in der Abbildung in rot und gelb dargestellt, werden aber zur weiteren Verarbeitung nicht beachtet.

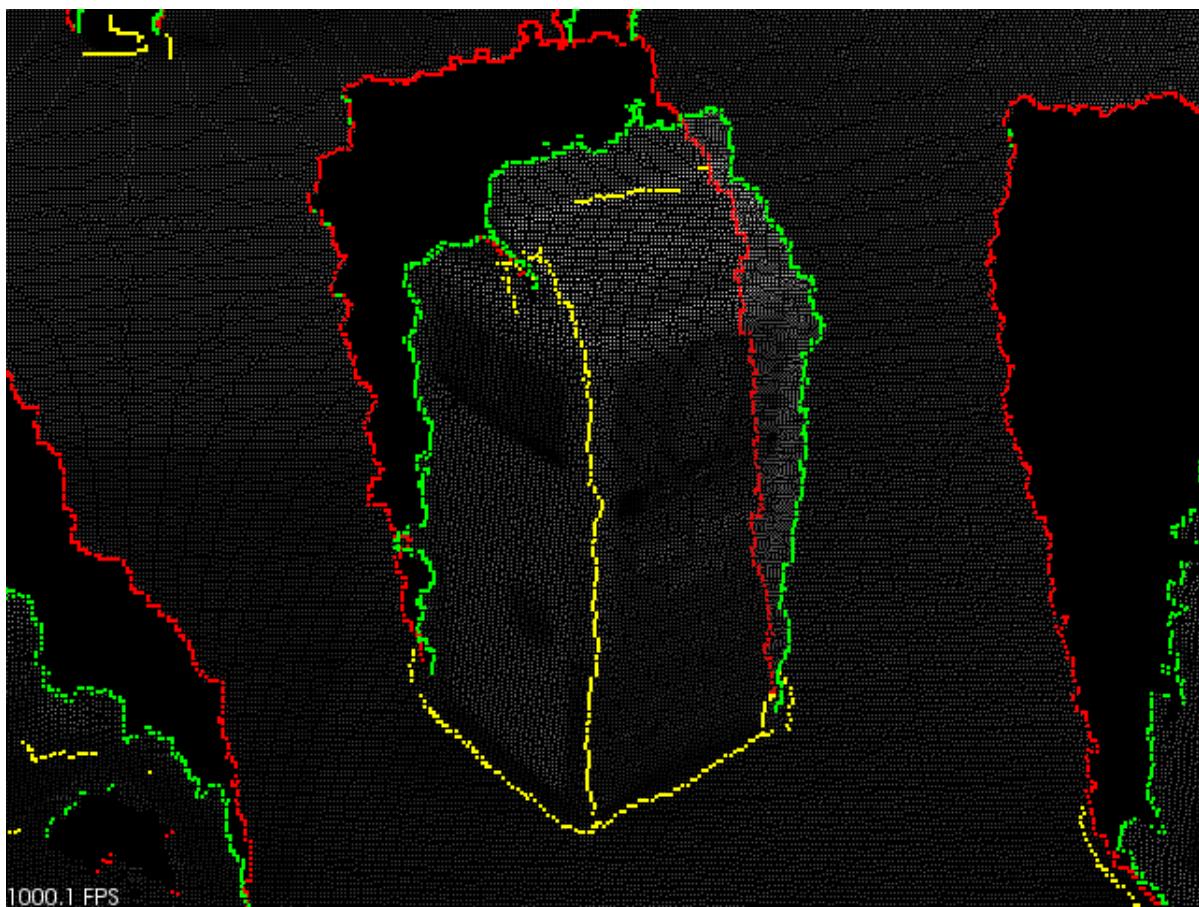


Abbildung 21: Markierte Kanten in einem Tiefendatenbild [3]

An die X- und Z-Achse deren Koordinaten können nun Gitterzellen gesetzt werden. Diese sollen bei der autonomen Steuerung genutzt werden um festzustellen, ob es sich bei dem vorliegenden Objekt um eine Tür oder eine Säule handelt. Abschließend müssen diese Gitterzellen zu einem Datenpaket zusammengefügt werden um diese an das Modul zu autonomen Steuerung zu senden.

## 7 Autonome Steuerung

DANIEL BETSCHE

Das Ergebnis der Analyse des 3D-Umweltmodells liefert eine Menge von Gitterzellen. Diese Zellen repräsentieren Kanten innerhalb der von der Drohne wahrgenommenen Welt. Es gilt nun aus den Positionen der Kanten berechnen zu können, wo eine Säule, eine Wand oder eine Tür sich relativ zur aktuellen Position der Flugdrohne befindet. Die Arbeit soll an dieser Stelle lediglich aufzeigen können, dass die Erkennung von dreidimensionalen Objekten anhand von Tiefendaten möglich ist. Eine uneingeschränkte Objekterkennung und Klassifizierung im dreidimensionalen Raum würde den Rahmen der Arbeit sprengen. Daher wurde die Vereinbarung getroffen, die Ausmaße von Säulen und Türen eindeutig festzulegen um die Detektion zu erleichtern.

- Säulen haben einen Durchmesser von 50 cm
- Türen haben eine Breite von einem Meter
- Keine Tür bzw. Säule hat einen Abstand von weniger als zwei Metern zur nächsten Tür/Säule
- Die Ecke einer Wand ist mindestens zwei Meter vom nächsten Türrahmen entfernt

Der Algorithmus ist darauf ausgelegt beide Anwendungsszenarien getrennt zu behandeln. Es ist zwar auch möglich ein gemeinsames Szenario mit gemischten Hindernissen zu erstellen, allerdings ist dafür weiterer Entwicklungsaufwand notwendig. Wie bereits erwähnt soll die Steuerung nur exemplarisch aufzeigen wie eine Navigation anhand von dreidimensionalen Daten durchgeführt werden kann.

## 7.1 Einer Säule ausweichen

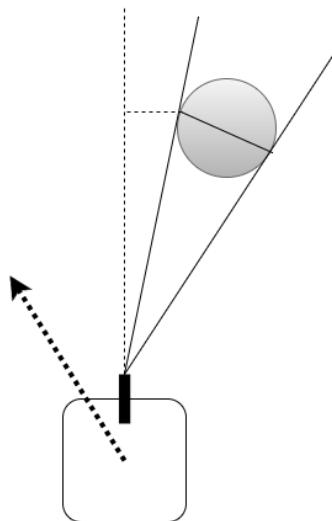


Abbildung 22: Schematische Darstellung des Ausweichvorgangs bei Kollisionskurs mit einer Säule

Der Algorithmus arbeitet im Fall der Säulen wie folgt: Alle Kanten und ihre Positionen sind bekannt. Von Interesse sind nur Kanten die im Blickfeld der Drohne und damit in ihrem Weg liegen. Von jeder dieser Kanten wird der Abstand zu allen anderen bekannten Kanten ermittelt. Haben zwei Kanten einen Abstand von 50cm zueinander so wurde eine Säule erkannt. Dem Abstand wird ein Fehler von 20% zugestanden um eventuelle Fehler in der Kantendetektion sowie weitere Abweichungen aufgrund des Kamerawinkels auszugleichen. Im optimalen Fall hat eine runde Säule allerdings aus beliebigem Blickwinkel immer den gleichen Durchmesser. Wurde eine Säule erkannt muss ihre Relative Position zur Drohne bestimmt und eine Entscheidung über eventuelle Kursabweichungen getroffen werden. Die Vorgehensweise hierbei ist auch denkbar einfach, sind beide Kanten links oder rechts zur Mitte der Drohne und mindestens um die Drohnenbreite verschoben, so ist keine Aktion erforderlich. Ist der Abstand allerdings geringer, muss der Weg angepasst werden. Als Ziel wird immer ein Wegpunkt links oder rechts von der Säule angesteuert. Ein Beispiel dafür ist in Abbildung 22 zu sehen. Die Säule befindet sich Rechts von dem Quadrocopter und auf Kollisionskurs, also wird eine Ausgleichsbewegung nach links veranlasst.

## 7.2 Eine Tür durchqueren

Ist dies der Fall muss die ARDrone zuerst wieder eine Tür identifizieren. Dazu werden erneut alle relevanten Kanten analysiert und der Abstand zueinander ausgerechnet.

Wurden zwei Kanten gefunden, welche einen Abstand von einem Meter aufweisen können werden diese als Türrahmen identifiziert. Auch hier wird ein Fehler von bis zu 20% zugestanden. Anderst als bei der Säule gilt es hier nun einen sauberen Flug durch den Türrahmen zu realisieren. Dazu wird ein Wegpunkt vor die beiden Kanten gelegt, sodass vor der Durchquerung eine Kante links und eine rechts von der Mittellinie des Quadrocopters liegt. Wurde dieser Wegpunkt erreicht wird die Drohne vorsichtig nach vorne beschleunigt. Dabei gilt es stetig beide Kanten in gleichem Abstand zum Mittelpunkt zu halten. Ein Beispiel dafür ist in Abbildung 23 zu sehen. Die Tür befindet sich links von der Drohne, zwei Wegpunkte werden geplant und angeflogen.

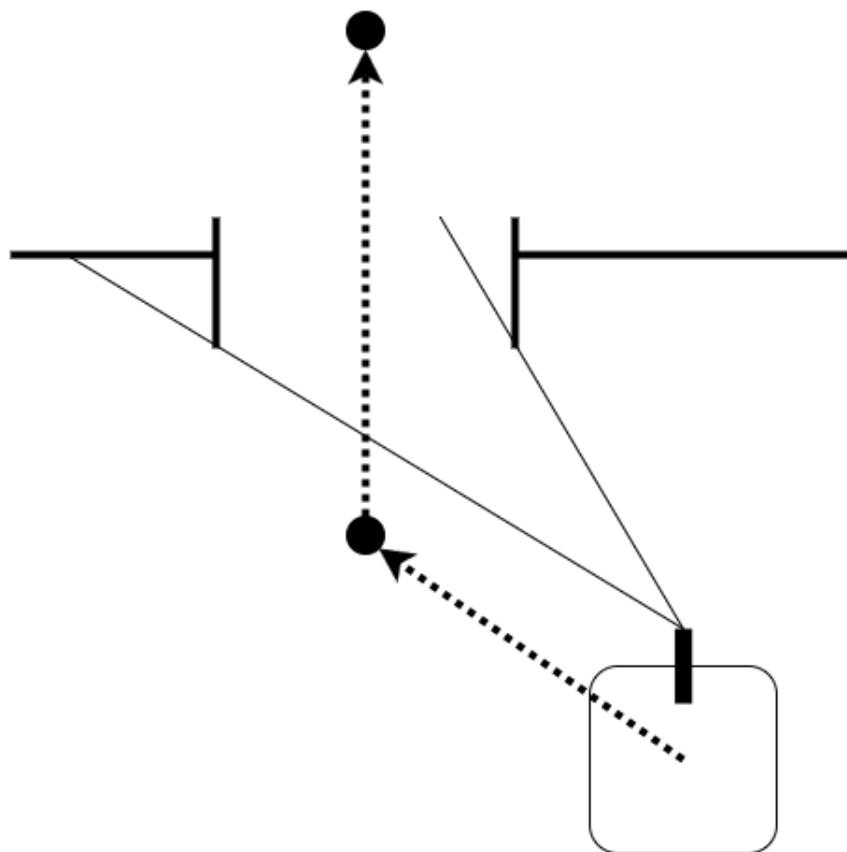


Abbildung 23: Schematische Darstellung des Durchquerens eines Türrahmens

Auf Basis dieser Szenarien lassen sich weitaus komplexere Steuerungen entwickeln, allerdings konnten die grundlegenden Funktionen der autonomen Steuerung umgesetzt werden. Diese basieren hauptsächlich auf der Annäherung an gesetzte Wegpunkte. Die Wegpunkte werden intern in dem Paket *drone\_autonomy* verarbeitet. Die Differenz der aktuellen Position und des nächsten zu erreichenden Wegpunktes wird in einen zweidimensionalen Vektor umgerechnet und als Geschwindigkeit an das *cmd\_vel* Topic geschickt. Die Länge des Vektors entscheidet dabei über die Geschwindigkeit der Bewegung. Je kleiner der Abstand, desto langsamer wird angeglichen. Das Resul-

tat dieser Steuerung ist dadurch ein schwebender Zustand um den Wegpunkt herum. Diese Funktion wurde so implementiert, da die Simulierte Drohne auch mit einem Störfaktor in ihrer Flugeigenschaft versehen wurde. Wird kein Befehl gesendet driftet der Quadrocopter unkontrolliert durch die Simulation. Daher wird bei Start der Anwendung der erste und aktuelle Wegpunkt an der Position (0,0) gewählt. Die autonome Steuerung lässt sich per ROS Service Call an und ausschalten.

## 8 Ergebnis und Ausblick

### 8.1 Fazit

Abschließend lässt sich sagen, dass es sich das Thema definitiv als schwieriger herausgestellt hat, als zuvor angenommen wurde. Nicht ohne Grund ist das SLAM Problem (Simultaneous Localization and Mapping) weiterhin ein Teil vieler Masterthesen und Dissertationen. Dabei handelt es sich letztendlich um ein klassisches Henne-Ei-Problem, da sowohl Position im Raum als auch der Raum selbst unbekannt sind und somit beides gleichzeitig erstellt werden und voneinander abgeleitet werden muss. Daraus ergibt sich auch, dass es eine Vielzahl von Ansätzen gibt, aber eine optimale Lösung noch nicht existiert.

Zu Beginn der Arbeit stand die Drohne sehr im Vordergrund, geriet aber im Laufe des Projektes immer mehr in den Hintergrund, bis letztendlich alles über eine Simulation durchgespielt wurde. Das liegt mit daran, dass es einfacher schneller uns sicherer ist, die Simulation zu starteten und die Drohne nicht durch falsche Manöver zu beschädigen. Außerdem gelang es so eine viel größere Anzahl an Testfällen zu testen, Situationen nach zu spielen und gegebenenfalls mit geänderten Parametern oder in langsamerer Geschwindigkeit zu wiederholen.

Letztendlich kann man sagen, dass der Einsatz der Simulation eine deutliche Bereicherung für die Durchführung dieser Studienarbeit war. Mit nur einer Drohne ist es schwierig mit zwei Personen an einem Projekt zu arbeiten, da nur eine Person in der Lage ist den gerade entstandenen Code zu testen. Ebenso verringert es das Risiko von Materialverschleiß und Schäden durch Kollisionen.

Wäre früher bekannt gewesen, dass die Software von Myestro nicht einsetzbar ist, so hätte man sich auch früher darauf eingestellt eine Simulation zu verwenden. Ebenfalls hätte man daraufhin weitere Features einbauen können, die in der Kürze der Zeit nicht mehr zu implementieren waren.

### 8.2 Ausblick

Die Autonomie von Maschinen hat in der Robotik einen starken Wandel ausgelöst. Es werden immer mehr Sensoren verwendet und darauf basierend noch mächtigere Algorithmen implementiert. Diese Studienarbeit sollte das ursprüngliche Ziel verfolgen mit möglichst einfachen Mitteln in handelsüblicher Hardware und geschickter Kombination

der Sensordaten zu dieser Entwicklung beizutragen. Leider konnten nicht alle Ziele erreicht werden, sodass ein Einsatz der echten Drohne erst möglich wird durch weitere Fortschritte.

Die in der Arbeit aufgezeigten Lücken und Ansätze versprechen ein hohes Potential zur Weiterentwicklung. Im Bereich der Bildaufnahme gibt es zwar funktionierenden Algorithmen, allerdings ist notwendig diese von eigens dafür konzipierten Spezialmaschinen auf handelsübliche Rechner zu portieren. Besonders wünschenswert sind Verbesserungen in der Performance. Eine wünschenswerte Entwicklung hierfür ist unter anderem ein hybrider Ansatz. Die Umwandlung zu 3D Tiefendaten wird weiterhin ein aufwendiger Prozess bleiben, was einer schnellen Reaktionszeit entgegenwirkt. Im hybriden Ansatz allerdings, werden Analysen auf 2D und 3D Bilddaten ausgeführt. Für schnelle Reaktionen kann auf das einfachere Format gesetzt werden, für komplexere Aufgaben werden die Tiefendaten betrachtet.

Neben der Verbesserung der Aufnahme von Bildern sind auch die Algorithmen zur Auswertung und zur Steuerung zu verbessern. Hierbei wurden lediglich simple Formen mit stark eingeschränkten Parametern betrachtet. Diese müssen für einen tatsächlichen autonomen Flug durch reale Umgebungen weiter verbessert und ausgebaut werden. Aufgrund deutscher Normen ist es nicht weit hergeholt, dass jede Tür eine gewisse Breite aufweist, allerdings besteht ein Gebäude aus noch deutlich mehr Strukturen. Ein möglicher Ansatz ist es, hieran anzuknüpfen und weitere typische Formen zu erkennen wie Treppen, Tische und Stühle.

Unabhängig von der Arbeit kann man sagen, dass durch die Simulation mit Gazebo von nun an die Entwicklung von Szenarien und Algorithmen deutlich einfacher und schneller durchgeführt werden kann. Diese können im Anschluss ohne weitere Anpassungen mit echter Hardware getestet werden sobald genug Konfidenz in die Anwendungen vorhanden ist.

## 9 Literatur

- [1] AR Drone SDK. [http://developer.parrot.com/docs/ardrone/ARDrone\\_SDK\\_2\\_0\\_1.zip](http://developer.parrot.com/docs/ardrone/ARDrone_SDK_2_0_1.zip). Abgerufen am 3. März 2015.
- [2] Camgaroo Academy. Stereogrundlagen. <http://www.camgaroo.com/camgaroo-academy/artikel/details/stereoskopie-grundlagen-teil-1-historie/>, 2010. Abgerufen am 23. Februar 2016.
- [3] Changhyun Choi. Edge detection. 2012.
- [4] Marc Pollefeyts Cornelia Fermueller. Edge detection.
- [5] Fantagu. Verzeichnung3. <https://commons.wikimedia.org/wiki/File%3AVerzeichnung3.png>, 2009. Abgerufen am 13. Februar 2016.
- [6] Open Source Robotics Foundation. [http://gazebosim.org/tutorials?tut=ros\\_overview&cat=connect\\_ros](http://gazebosim.org/tutorials?tut=ros_overview&cat=connect_ros). Abgerufen am 5. Juni 2016.
- [7] Open Source Robotics Foundation. <http://gazebosim.org/>. Abgerufen am 5. Juni 2016.
- [8] Open Source Robotics Foundation. <http://sdformat.org/>. Abgerufen am 5. Juni 2016.
- [9] Open Source Robotics Foundation. Ros. <http://www.ros.org/about-ros/>. Abgerufen am 9. März 2015.
- [10] ROS Foundation. [http://wiki.ros.org/catkin/conceptual\\_overview](http://wiki.ros.org/catkin/conceptual_overview). Abgerufen am 4. Juni 2016.
- [11] Willow Garage. <https://www.willowgarage.com/blog/2012/06/18/open-perception>. Abgerufen am 3. Mai 2016.
- [12] Felix Geisendorfer. Nodecopter. <http://www.nodecopter.com/>, 2012. Abgerufen am 3. März 2015.
- [13] Felix Geisendorfer. Node ar drone. <https://github.com/felixge/node-ar-drone>, 2013. Abgerufen am 3. März 2015.
- [14] Pierre-Yves Gilliuron. <http://flashinformatique.epfl.ch/spip.php?article2581>. Abgerufen am 28. März 2016.
- [15] Josef Krames. Zur ermittlung eines objektes aus zwei perspektiven. (ein Beitrag zur theorie der gefährlichen Örter.). *Monatshefte für Mathematik und Physik*, 49(1):327–354, 1941.

- [16] Open Point Cloud Library. <http://pointclouds.org/about/>. Abgerufen am 3. Mai 2016.
- [17] Open Point Cloud Library. [http://pointclouds.org/documentation/tutorials/statistical\\_outlier.php#statistical-outlier-removal](http://pointclouds.org/documentation/tutorials/statistical_outlier.php#statistical-outlier-removal). Abgerufen am 4. August 2015.
- [18] Open Point Cloud Library. <http://robotica.unileon.es/mediawiki/images/a/ab/Correspondence.jpg>. Abgerufen am 28. März 2016.
- [19] Open Point Cloud Library. [http://pointclouds.org/documentation/tutorials/\\_images/scans.jpg](http://pointclouds.org/documentation/tutorials/_images/scans.jpg). Abgerufen am 28. März 2016.
- [20] Open Point Cloud Library. [http://pointclouds.org/documentation/tutorials/\\_images/s1-6.jpg](http://pointclouds.org/documentation/tutorials/_images/s1-6.jpg). Abgerufen am 28. März 2016.
- [21] Open Point Cloud Library. [http://pointclouds.org/documentation/tutorials/pcd\\_file\\_format.php](http://pointclouds.org/documentation/tutorials/pcd_file_format.php). Abgerufen am 3. Juni 2016.
- [22] Theo Moons, Luc Van Gool, and Maarten Vergauwen. *3D reconstruction from multiple images, Part 1: Principles*. Now Publishers Inc, 2009.
- [23] Clearpath Robotics. <http://www.clearpathrobotics.com/guides/ros/Intro%20to%20the%20Robot%20Operating%20System.html>. Abgerufen am 3. Mai 2016.
- [24] Rudolfo4. Verzeichnung blende wikimedia. [http://olypedia.de/static/images/d/d2/Verzeichnung\\_Blende\\_Wikimedia.png](http://olypedia.de/static/images/d/d2/Verzeichnung_Blende_Wikimedia.png), 2009. Abgerufen am 13. Februar 2016.
- [25] Felix Penzlin Christoph Walter Nico Hochgeschwender Sebastian Zug, Thomas Poltrack. Analyse und vergleich von frameworks für die implementierung von robotikanwendungen. 2013.
- [26] Bjarne Stroustrup. <http://www.stroustrup.com/C++.html>. Abgerufen am 4. Juni 2016.
- [27] Institute of Systems University of Combra and Robotics. [http://dynface4d.isr.uc.pt/images/database/MergePoints1Snap2\\_a.jpg](http://dynface4d.isr.uc.pt/images/database/MergePoints1Snap2_a.jpg). Abgerufen am 28. März 2016.

## 10 Anhänge

Listing 20: Sensoren der Drohne

```

1 <?xml version="1.0"?>
2
3 <robot name="drone"
4   xmlns:sensor="http://playerstage.sourceforge.net/gazebo/xmlschema/#sensor"
5   xmlns:controller="http://playerstage.sourceforge.net/gazebo/xmlschema/#controller"
6   xmlns:interface="http://playerstage.sourceforge.net/gazebo/xmlschema/#interface"
7   xmlns:xacro="http://ros.org/wiki/xacro">
8
9 <xacro:macro name="drone_sensors">
10 <gazebo>
11   <plugin name="drone_imu_sim" filename="libhector_gazebo_ros_imu.so">
12     <alwaysOn>true</alwaysOn>
13     <updateRate>100.0</updateRate>
14     <bodyName>base_link</bodyName>
15     <frameId>ardrone_base_link</frameId>
16     <topicName>/ardrone/imu</topicName>
17     <rpyOffsets>0 0 0</rpyOffsets> <!-- deprecated -->
18     <gaussianNoise>0</gaussianNoise> <!-- deprecated -->
19     <accelDrift>0.5 0.5 0.5</accelDrift>
20     <accelGaussianNoise>0.35 0.35 0.3</accelGaussianNoise>
21     <rateDrift>0.1 0.1 0.1</rateDrift>
22     <rateGaussianNoise>0.05 0.05 0.015</rateGaussianNoise>
23     <headingDrift>0.0</headingDrift>
24     <headingGaussianNoise>0.00</headingGaussianNoise>
25   </plugin>
26
27   <plugin name="drone_baro_sim" filename="libhector_gazebo_ros_baro.so">
28     <alwaysOn>true</alwaysOn>
29     <updateRate>10.0</updateRate>
30     <bodyName>base_link</bodyName>

```

```
31      <topicName>pressure_height</topicName>
32      <altimeterTopicName>altimeter</altimeterTopicName>
33      <offset>0</offset>
34      <drift>0.1</drift>
35      <gaussianNoise>0.5</gaussianNoise>
36  </plugin>
37
38  <plugin name="drone_magnetic_sim" filename="
39      libhector_gazebo_ros_magnetic.so">
40      <alwaysOn>true</alwaysOn>
41      <updateRate>10.0</updateRate>
42      <bodyName>base_link</bodyName>
43      <topicName>magnetic</topicName>
44      <offset>0 0 0</offset>
45      <drift>0.0 0.0 0.0</drift>
46      <gaussianNoise>1.3e-2 1.3e-2 1.3e-2</gaussianNoise>
47  </plugin>
48
49  <plugin name="drone_gps_sim" filename="libhector_gazebo_ros_gps.so">
50      <alwaysOn>true</alwaysOn>
51      <updateRate>4.0</updateRate>
52      <bodyName>base_link</bodyName>
53      <topicName>fix</topicName>
54      <velocityTopicName>fix_velocity</velocityTopicName>
55      <drift>5.0 5.0 5.0</drift>
56      <gaussianNoise>0.1 0.1 0.1</gaussianNoise>
57      <velocityDrift>0 0 0</velocityDrift>
58      <velocityGaussianNoise>0.1 0.1 0.1</velocityGaussianNoise>
59  </plugin>
60
61  <plugin name="drone_groundtruth_sim" filename="libgazebo_ros_p3d.so"
62      >
63      <alwaysOn>true</alwaysOn>
64      <updateRate>100.0</updateRate>
65      <bodyName>base_link</bodyName>
66      <topicName>ground_truth/state</topicName>
67      <gaussianNoise>0.0</gaussianNoise>
68      <frameName>map</frameName>
```

```

67      </plugin>
68
69      </gazebo>
70  </xacro:macro>
71 </robot>

```

Listing 21: Controller der Drohne

```

1  <?xml version="1.0"?>
2
3 <robot name="drone"
4   xmlns:sensor="http://playerstage.sourceforge.net/gazebo/xmlschema/#"
5     sensor"
6   xmlns:controller="http://playerstage.sourceforge.net/gazebo/xmlschema/#"
7     controller"
8   xmlns:interface="http://playerstage.sourceforge.net/gazebo/xmlschema/#"
9     interface"
10  xmlns:xacro="http://ros.org/wiki/xacro">
11
12  <!-- Drohnen Control Plugin, hier sollte die Steuerung der ARdrone
13    nachgebildet werden -->
14  <xacro:macro name="drone_controller">
15    <gazebo>
16      <plugin name="quadrotor_simple_controller" filename="
17        libhector_gazebo_quadrotor_simple_controller.so">
18        <alwaysOn>true</alwaysOn>
19        <updateRate>0.0</updateRate>
20        <bodyName>base_link</bodyName>
21        <stateTopic>ground_truth/state</stateTopic>
22        <imuTopic>ardrone imu</imuTopic>
23        <topicName>cmd_vel</topicName>
24        <rollpitchProportionalGain>10.0</rollpitchProportionalGain>
25        <rollpitchDifferentialGain>5.0</rollpitchDifferentialGain>
26        <rollpitchLimit>0.5</rollpitchLimit>
27        <yawProportionalGain>2.0</yawProportionalGain>
28        <yawDifferentialGain>1.0</yawDifferentialGain>
29        <yawLimit>1.5</yawLimit>
30        <velocityXYProportionalGain>5.0</velocityXYProportionalGain>
31        <velocityXYDifferentialGain>1.0</velocityXYDifferentialGain>

```

```

27      <velocityXYLimit>2</velocityXYLimit>
28      <velocityZProportionalGain>5.0</velocityZProportionalGain>
29      <velocityZDifferentialGain>1.0</velocityZDifferentialGain>
30      <velocityZLimit>0.5</velocityZLimit>
31      <maxForce>30</maxForce>
32      <motionSmallNoise>0.05</motionSmallNoise>
33      <motionDriftNoise>0.03</motionDriftNoise>
34      <motionDriftNoiseTime>5.0</motionDriftNoiseTime>
35    </plugin>
36
37    <plugin name="quadrotor_state_controller" filename="
38        libhector_gazebo_quadrotor_state_controller.so">
39      <alwaysOn>true</alwaysOn>
40      <updateRate>0.0</updateRate>
41      <bodyName>base_link</bodyName>
42      <stateTopic>ground_truth/state</stateTopic>
43      <imuTopic>ardrone/imu</imuTopic>
44      <sonarTopic>sonar_height</sonarTopic>
45      <topicName>cmd_vel</topicName>
46    </plugin>
47  </gazebo>
48 </xacro:macro>
49 </robot>

```

Listing 22: Vollständiges Launchfile

```

1 <?xml version="1.0"?>
2 <launch>
3   <arg name="model" default="$(find studienarbeit)/urdf/drone.urdf.xacro"
4     />
5   <arg name="x" default="0"/>
6   <arg name="y" default="0"/>
7   <arg name="z" default="0"/>
8   <arg name="R" default="0"/>
9   <arg name="P" default="0"/>
10  <arg name="Y" default="0"/>
11  <!-- send the robot XML to param server -->
12  <param name="robot_description" command="$(find xacro)/xacro --inorder
13    '$(arg model)' base_link_frame:=base_link world_frame:=world" />

```

```
12 <param name="base_link_frame" type="string" value="base_link"/>
13 <param name="world_frame" type="string" value="world"/>
14
15 <!-- push robot_description to factory and spawn robot in gazebo -->
16 <node name="spawn_robot" pkg="gazebo_ros" type="spawn_model"
17 args="--param robot_description
18 -urdf
19 -x '$(arg x)'
20 -y '$(arg y)'
21 -z '$(arg z)'
22 -R '$(arg R)'
23 -P '$(arg P)'
24 -Y '$(arg Y)'
25 -model quadrotor"
26 respawn="false" output="screen"/>
27
28 <!-- start robot state publisher -->
29 <node pkg="robot_state_publisher" type="robot_state_publisher" name="
30   robot_state_publisher" output="screen" >
31   <param name="publish_frequency" type="double" value="50.0" />
32   <param name="tf_prefix" type="string" value="" />
33 </node>
34
35 <!-- tranform kinect pointcloud data to correct -->
36 <node pkg="tf" type="static_transform_publisher" name="
37   rotate_kinect_pointcloud" args="0 0 0 -1.58 0 -1.58 /base_link /
     ray_link 100"/>
38
39 </launch>
```