



# Entwicklung eines Indoor-Assistenzsystems für Multicopter

## mit Hilfe von Monocularer Tiefenbild Rekonstruktion

### Studienarbeit

Studiengang Angewandte Informatik  
Duale Hochschule Baden-Württemberg Karlsruhe

von

Christoph Meise, Max Lenk

Datum der Abgabe: 15.05.2017  
Bearbeitungszeitraum: 2 Semester  
Matrikelnummern und Kurse: 4050853, 3460046, TINF14B2, TINF14B1  
Betreuer: Markus Strand

Copyright Vermerk:  
All rights reserved. **Copyright.**

© 2017

# Erklärung

(gemäß §5(3) der „Studien- und Prüfungsordnung DHBW Technik“ vom 29. 9. 2015) Ich versichere hiermit, dass ich meine Studienarbeit mit dem Thema:

*Entwicklung eines Indoor-Assistenzsystems für Multicopter*

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Walldorf, der 15.05.2017

---

CHRISTOPH MEISE, MAX LENK

**Restriction notice**

This report contains confidential information of

SAP SE

Dietmar-Hopp-Allee 16

69190 Walldorf, Germany

It may be used for examination purposes as a performance record of the department of Applied Computer Science at the Cooperative State University Karlsruhe. The content has to be treated confidentially.

Duplication and publication of this report - as a whole or in extracts - is not allowed.

**Sperrvermerk**

Der Inhalt dieser Arbeit darf weder als Ganzes noch in Auszügen Personen außerhalb des Prüfungsprozesses und des Evaluationsverfahrens zugänglich gemacht werden, sofern keine anders lautende Genehmigung der Ausbildungsstätte vorliegt.

# **Abstrakt**

Die Zusammenarbeit von Mensch und Machine bekommt eine zunehmende Bedeutung. Mit Hilfe von Assistenzsystemen wird versucht, das Fehlerpotential möglichst gering zu halten.

Im Rahmen dieser Arbeit soll daher ein Assistenzsystem für einen Quadrokopter konzipiert und erarbeitet werden.

# Inhaltsverzeichnis

<b>Erklärung</b>	<b>I</b>
<b>Abbildungsverzeichnis</b>	<b>VII</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	2
1.2 Aufbau . . . . .	3
<b>2 Grundlagen</b>	<b>5</b>
2.1 AR.Drone 2.0 . . . . .	5
2.2 ROS . . . . .	6
2.2.1 Allgemeines . . . . .	6
2.2.2 Design Prinzipien . . . . .	7
2.2.3 ROS Nodes . . . . .	9
2.3 Simulation mit Gazebo . . . . .	10
2.4 Fuzzylogik . . . . .	12
2.4.1 Allgemeines . . . . .	12
2.4.2 Fuzzylite . . . . .	14
2.5 Kinect . . . . .	16
2.5.1 Allgemeines . . . . .	16
2.5.2 Vergleich der verschiedenen Stacks zur Implementation . . . . .	16

2.6	Punktwolken . . . . .	17
2.7	Point Cloud Library . . . . .	18
<b>3</b>	<b>Software Architektur</b>	<b>19</b>
3.1	Anforderungen . . . . .	19
3.2	Bildverarbeitung . . . . .	21
3.2.1	Semi-Direct Monocular Visual Odometry - SVO . . . . .	21
3.2.2	Kamerakalibrierung . . . . .	23
3.2.3	Regularized Monocular Depth Estimation - REMODE . . . . .	27
3.2.4	Performanceprobleme . . . . .	30
3.3	Implementierung . . . . .	32
3.3.1	Grundlegende Herausforderungen . . . . .	32
3.3.2	Architektur . . . . .	32
3.3.3	Ansteuerung der Kinect . . . . .	35
3.3.4	Kinect Controller . . . . .	37
3.3.5	Fuzzy Controller . . . . .	37
3.3.6	Drone Controller . . . . .	38
3.3.7	ROS Nodes . . . . .	39
3.4	Assistenzsystem . . . . .	41
3.4.1	Problemanalyse . . . . .	41
3.4.2	Lösungsansätze . . . . .	42
3.4.2.1	Referenzobjekt . . . . .	42
3.4.2.2	Referenzpunktwolke . . . . .	45
3.4.2.3	Alternative Lösungen . . . . .	46
<b>4</b>	<b>Evaluation</b>	<b>47</b>
4.1	Ergebnis . . . . .	47

4.2 Ausblick . . . . .	48
4.2.1 Andere Simulatoren . . . . .	48
4.2.2 Assistenzsystem . . . . .	49
<b>Literaturverzeichnis</b>	<b>51</b>

# Abbildungsverzeichnis

2.1	Parrot AR.DRONE 2.0 Elite Edition [1] . . . . .	5
2.2	Publish-Subscribe Pattern . . . . .	10
2.3	Programmstruktur mit dem realen Quadrocopter . . . . .	12
2.4	Programmstruktur unter Verwendung von Gazebo . . . . .	13
2.5	Beispiel einer visualisierten Punktwolke [2] . . . . .	17
3.1	Semi-Direct Monocular Visual Odometry im Simulator . . . . .	23
3.2	Prinzip der Lochkamera [3] . . . . .	25
3.3	REMODE 3D-Modell, Flug über Trümmer [4] . . . . .	28
3.4	Übersicht zur Softwarearchitektur . . . . .	35
3.5	Darstellung der Körperteile in RVIZ . . . . .	36
3.6	UML Klassendiagramm des Kinect Controllers . . . . .	38
3.7	Zusammenspiel der verschiedenen ROS-Nodes . . . . .	40
3.8	Approximiertes Tiefenbild durch REMODE . . . . .	41
3.9	Objektreferenz einer Tür . . . . .	42
3.10	Verlauf der zweiten Ableitung im LoG Algorithmus [5] . . . . .	44
3.11	Correspondence Grouping mit dem PCL Algorithmus [6] . . . . .	45
4.1	AirSim von Microsoft in Aktion[7] . . . . .	49

# 1 Einleitung

Autonomes Fahren, Machine Learning und Industrie 4.0. Hinter diesen aktuellen Themen steckt das Ziel, Abläufe und Zusammenhänge kontinuierlich zu automatisieren und für den Menschen zu vereinfachen. Das Thema der Automation kann vor allem in der Interaktion zwischen Mensch und Maschine hilfreich sein. Auf Grund der geringen Kosten und der hohen Anwendungsvielfalt bieten sich vor allem Drohnen für die Forschung zu Automation in der Robotik an.

Mit Modellen ab 30 und bis zu mehreren tausend Euro gibt es Ausführungen für nahezu jeden Anwendungsfall. Meist mit mehreren Sensoren und Kameras ausgestattet, stellen sie nicht nur Spielzeug dar, sondern sind essentiell für reale Anwendungsgebiete.

So werden heute schon Drohnen genutzt, um Katastrophengebiete und Kriegsregionen aus sicheren Standorten aufzuklären, oder um die Feuerwehr bei der Branderkundung und Menschensuche zu unterstützen.

Natürlich können sie auch genutzt werden, um alltägliche Probleme zu lösen, wie die schnelle und direkte Lieferung von Paketen.

Da diese große Zahl an Drohnen nicht mehr manuell gesteuert werden kann, müssen sich diese größtenteils autonom bewegen. Dabei treten eine Vielzahl von komplexen Problemen auf, wie das zurechtfinden in einem unbekannten Raum und die Objekterkennung.

Außerdem ist bei Fluggeräten das Problem, dass die verwendete Ausrüstung ten-

denziell leicht und klein sein muss, damit die Flugeigenschaften nicht eingeschränkt werden bzw. die Drohne nicht zu groß wird.

Im Umfang dieser Arbeit soll eine Vorstufe zum autonomen Fliegen untersucht und implementiert werden: ein Assistenzsystem für den manuellen Flug. Dies ist Vergleichbar mit den Assistenzsystemen in PKWs, bei denen ein Tempomat, Licht- und Regensensoren oder Spurhalteassistenten den Fahrer unterstützen, jedoch das Fahren nicht abnehmen.

Eine solche semi-autonome Assistenzfunktion könnte beispielsweise dabei helfen, die Drohne durch enge Räume und durch Hindernisse zu fliegen, also Flugmanöver, bei denen ein Mensch eine hohe Fehleranfälligkeit aufweist.

## 1.1 Motivation

Das Ziel besteht darin, dass die Drohne aktiv die Umgebung auswertet und dabei Objekte wie Türen, oder Wände erkennt. Der Unterschied zu bereits bestehenden Projekten in diesem Themengebiet besteht darin, dass nur eine einzelne monokulare Kamera verwendet werden soll, anstatt externe Tiefenbildkameras zu montieren.

Dadurch trifft man auf eine Vielzahl von komplexen Problemen, welche im weiteren Verlauf dieser Arbeit dargestellt werden. Auf der anderen Seite könnte man dadurch teure Hardware sparen und somit auch auf andere Projekte anwenden.

Anhand der Tiefenbilder soll die Drohne anschließend in der Lage sein, die Entscheidungen des Nutzers zu unterstützen und abzuändern, um somit beispielsweise Kollisionen zu vermeiden, oder gezielt durch Hindernisse zu fliegen.

Das Problem soll nur durch die integrierte Hardware gelöst werden, also einer einfachen 720p Kamera die nach vorn gerichtet an der Drohne befestigt ist. Weiterhin ist das Ziel der Arbeit eine vollständige Modularisierung der Softwarearchitektur.

Dies ist essentiell, da das Assistenzsystem sowohl bei einer realen Drohne, als auch in einer Simulation funktionieren soll.

## 1.2 Aufbau

Diese Studienarbeit besteht aus 3 Kapiteln. Im ersten Abschnitt der Arbeit sollen die Grundlagen erklärt werden. Zuerst wird dabei die genutzte Drohne und deren Spezifikationen dargestellt.

Anschließend wird das Software Framework Roboter Operating System (*ROS*) eingeführt, welches ein Hauptbestandteil der Projektarchitektur ausmacht.

Im weiteren Verlauf wird dann die Simulationsumgebung beschrieben, in der eine simulierte Drohne geflogen werden kann. Im letzten Teil dieses Kapitels wird die Tiefenbildkamera Kinect beschrieben, welche zur Umsetzung der Gestensteuerung genutzt wird.

Der zweite Abschnitt dieser Arbeit umfasst die Software Architektur. Zuerst werden hierbei die technischen und Anforderungen an das Projekt aufgestellt. Anschließend ist das Kapitel in drei logische abgegrenzte Unterkapitel unterteilt: Bildverarbeitung, Implementierung und das Assistenzsystem.

Im Teil Bildverarbeitung wird beschrieben, wie aus den einfachen Bildern zuerst die Position der Kamera im Raum approximiert wird, um anschließend aus aufeinanderfolgenden Aufnahmen Tiefeninformationen zu gewinnen.

Im Folgenden wird im Abschnitt Implementierung die softwaretechnische Umsetzung und die Architektur beschrieben. Abschließend dient das Unterkapitel Assistenzsystem dazu, eine allgemeine Hinführung zu bieten und mögliche Implementierungstechniken vorzuschlagen.

Der Letzte Teil der Arbeit ist die Evaluation. Diese besteht aus dem erzielten Ergebnis, sowie aus dem Ausblick für weitere Betrachtungen dieser Problematik. Dabei

werden außerdem aufgetretene Probleme und Hindernisse detailliert beschrieben.

## 2 Grundlagen

### 2.1 AR.Drone 2.0

Bei der AR.Drone 2.0 handelt es sich um einen ferngesteuerten Quadrocopter des französischen Herstellers Parrot SA. [8] Die Drohne ist standardmäßig steuerbar mit einer mobilen Applikation für Android und iOS Geräte. Dafür baut sie ein WLAN Netzwerk auf, mit dem sich die Geräte verbinden können. Zur Steuerung stellt die AR.Drone ein Interface zur Verfügung, mit dem sie ferngesteuert werden kann.

Im Umfang der Studienarbeit wird, wie in der folgenden Abbildung zu sehen, die aktuellste Version der AR.Drone 2.0 verwendet.



Abbildung 2.1: Parrot AR.DRONE 2.0 Elite Edition [1]

Diese zeichnet sich unter Anderem durch eine Frontkamera mit einer Auflösung von  $1280 \times 720$  Pixeln und einer Bildrate von 30 fps aus. Weiterhin ist Sie mit einer zum Boden gerichteten QVGA Kamera ausgerüstet, welche 60 Bilder pro Sekunde aufnimmt.

Die Drohne orientiert sich beim Fliegen mit Hilfe einer Vielzahl von Sensoren. Dazu gehören ein dreiachsiges Gyroskop und ein Magnetometer. Weiterhin nutzt sie Beschleunigungs-, Ultraschall- und Luftdrucksensoren.

Der Grund für die Wahl der Drohne ist vor allem der vergleichsweise niedrige Preis von ca. 200€ und der starken Verbreitung in der Forschung. Dadurch gibt es bereits eine Vielzahl von Projekten, die dazu führen, dass die Drohne und das dazugehörige Interface zu einem großen Umfang fehlerfrei funktionieren.

Weiterhin gibt es schon ROS Nodes (siehe 2.2) und konfigurierte Modelle in Simulationsumgebungen, welche die Arbeit an dem Projekt beschleunigen.

## 2.2 ROS

### 2.2.1 Allgemeines

Das Robotic Operating System, kurz ROS, ist eine Sammlung von Softwareframeworks für die Entwicklung von Software für persönliche Roboter. Es stellt entsprechende Bibliotheken und Werkzeuge zur Verfügung, um Entwicklern die Programmierung zu vereinfachen. Dabei bietet ROS einem Betriebssystem ähnliche Funktionalitäten auf Basis eines homogenen Computercluster. Dazu gehören Hardwareabstraktion, low-level Steuerung, Nachrichtevermittlung zwischen verschiedenen Prozessen und Paketmanagement. Trotz der Notwendigkeit hoher Reaktivität und geringer Latenz bei der Steuerung von Robotern handelt es sich es de facto nicht um ein richtiges Betriebssystem , obwohl es durch den Namen("Operating System") suggeriert wird. Dennoch ist es möglich Echtzeitcode ("realtime code") in ROS zu integrie-

ren [9]. ROS ist eins der am meisten genutzten Frameworks und hat eine stark wachsende Gemeinschaft, was es in Kombination mit dessen Features zu einer enorm wichtigen Technologie macht.[10] [11]

### 2.2.2 Design Prinzipien

Das Robotic Operating verfolgt fünf grundlegende Design Prinzipien:

- Peer-to-Peer
- werkzeugbasiert
- Mehrsprachigkeit
- Unabhängigkeit
- Open Source

**Peer-to-Peer:** Im Normalfall besteht ein Roboter aus mehreren Komponenten und oft aus verschiedenen Recheneinheiten. Oft gibt es einen zentralen leistungsstarken Rechner der unabhängig vom Roboter existiert, welcher für die Koordination und rechenintensive Aufgaben, wie zum Beispiel Bildverarbeitung verantwortlich ist. Da dieser oft wegen Mobilitätsgründen nicht per Kabel mit dem Roboter verbunden ist, geschieht die Kommunikation mittels WLAN oder vergleichbaren drahtlosen Kommunikationsmitteln. Dies kann unter Umständen sich schnell zu einem Flaschenhals entwickeln, da große Datenmengen transportiert werden müssen, wenn die zentrale Einheit für den Datentransport zuständig ist. Daher baut ROS auf ein Peer-to-Peer Konzept auf bei dem zentrale Einheit sich lediglich darum kümmert die kommunizierenden Nodes vor Kommunikationsbeginn miteinander verbindet. Dafür ist lediglich ein Auskunftsmechanismus notwendig, der Prozessen oder ähnlichem ermöglicht korrespondierende Kommunikationspartner zur Laufzeit zu finden und eine Verbindung herzustellen. Somit können mögliche Flaschenhälse entschärft

werden.[10][12]

**Werkzeugbasiert:** Dabei handelt es sich um ein weiteres grundlegendes Prinzip um die Benutzbarkeit zur vereinfachen und gleichzeitig eine höher Modularität zu gewähren. Anstelle eines großen komplexen Werkzeug zum Arbeiten mit ROS, existieren mehrere kleine Werkzeuge die dem Single-Responsibility-Prinzip(SRP)[13] folgen, also nur eine konkrete Aufgabe haben. Dieses Prinzip lässt sich mit einer Mikroservice-Architektur vergleichen, wobei es sich allerdings um Werkzeuge handelt und keine Services.

**Mehrsprachigkeit:** Damit möglichst viele Systeme an ROS angebunden werden können und auch die Vorteile einzelner Programmiersprachen ausgereizt werden können, existiert das Prinzip der Mehrsprachigkeit. Somit wird es dem Nutzer ermöglicht die Sprache mit der programmieren will, frei zu wählen und erhöht den Nutzkomfort. Momentan werden C++, Python und Lisp durch sogenannte ROS Client Libraries. Weiterhin existieren experimentelle Client Libaries für Sprachen GO, Haskell, Java und viele weitere.[14]

**Unabhängigkeit:** Die Kopplung der Software mit der Hardware bei Robotikszenarien ist immer relativ hoch, da die Treiber meistens plattformspezifisch sind. Um allerdings die Wiederverwendbarkeit diverser Algorithmen zu fördern, setzt man darauf die Algorithmen möglichst unabhängig vom jeweiligen Roboter zu implementieren, sodass sie im Optimalfall bei jedem System Anwendung finden können. Entwickler stellen die entsprechenden als Bibliotheken zur Verfügung und die starke Kopplung wird etwas entschärft.[10][12]

**Open Source:** Da ROS unter der BSD-Lizenz<sup>1</sup> steht, kann es sowohl für nicht-kommerzielle Projekte als auch kommerzielle Projekt verwendet werden. Der Source Code ist öffentlich zugänglich. Entwickler können ihre eignen Module beliebig lizenziieren.[10][12]

---

<sup>1</sup>siehe: <http://www.linfo.org/bsdlicense.html>

### 2.2.3 ROS Nodes

ROS baut auf einem einfachen Konzept auf, dem Publish-Subscribe Pattern, bei welchem ein Publisher eine Nachricht mit einem festem Thema verschicken kann und ein beliebiger Subscriber, der sich für das Thema interessiert, ist in der Lage diese zu empfangen, zu verarbeiten und unter Umständen erneut zu versenden. Somit besteht eine ROS Anwendung aus der Regel aus vielen kleinen Teilen, den sogenannten Nodes. Jede Node hat ihre eigenen Aufgabe und kann auf diverse Themen registrieren, sogenannte Topics. Für diese Topics werden von anderen Nodes Nachrichten publiziert, welche empfangen und verarbeitet werden können. Ist die Verarbeitung abgeschlossen, besteht die Möglichkeit die verarbeiteten Daten für andere Nodes zur Verfügung zu stellen, in dem sie mit einer neuen Topic publiziert werden. Somit kann eine Node sowohl Publisher, als auch Subscriber sein. Topics werden in ROS durch einen festen Nachrichtentyp definiert. Dessen exakter Aufbau muss vor Verwendung deklariert werden um einen einheitlichen Nachrichtenaustausch zu gewähren. Wie in der Abbildung unterhalb zu sehen, wird der Nachrichtentransfer durch eine zentrale Einheit geregelt, sodass Daten wirklich nur die Nodes erreichen, die sich auch dafür registriert haben. Dieser zentrale Bestandteil ist in ROS der Roscore, bei welchem sich alle Nodes zur Erstellung registrieren. Sozusagen eine Masternode, welche sich um die Verwaltung der anderen Nodes kümmert. Im Unterschied zum Pattern kümmert sich der Core nur um die anfängliche Vermittlung der Nodes untereinander. Er stellt somit sicher, dass sich die Nodes untereinander finden können. Er fungiert in diesem Falle als ein sogenannter Message Broker[15], was eine Erweiterung des klassischen Eventbus-Muster ist[16]. Weiterhin können ROS Nodes Services anbieten. Im Gegensatz zum regulären Nachrichtenaustausch basiert ein Serviceaufruf auf dem Request Response Prinzip, das heißt wenn ein Service aufgerufen wird, erhält der Aufrufer definitiv eine Antwort. Hierbei wird der Aufruf auch nicht über eine Topic realisiert, sondern die Node wird konkret angesprochen und muss dafür

aktiv sein.[10][12]

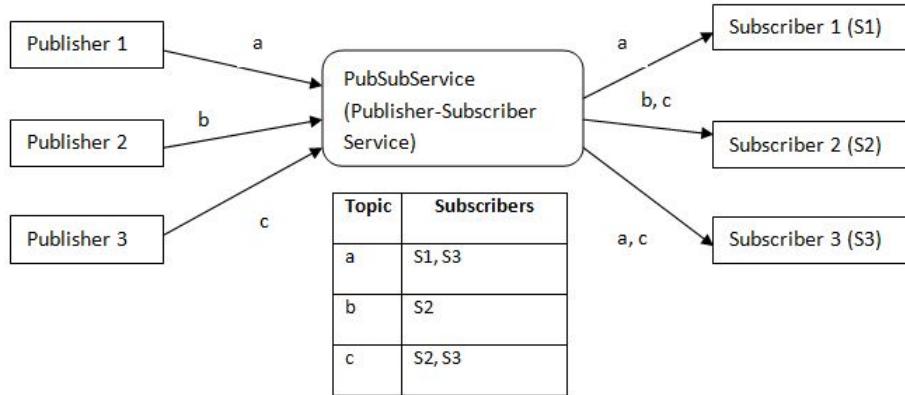


Abbildung 2.2: Publish-Subscribe Pattern

ROS Nodes können in verschiedenen Sprachen implementiert werden, da die Kommunikation über die festen Nachrichtentypen geschieht, welche über den Roscore ausgetauscht werden. Dadurch ist es möglich Nodes in C++, Python und ebenfalls Lisp zu programmieren. Wodurch das ganze Konzept enorm flexibel gestaltet wird. Die fest definierten Nachrichtentypen verhindern, dass es an den Schnittstellen zu Problemen kommt und Nachrichten von allen Nodes einheitlich empfangen und versendet werden.

Durch diese Modularität ist einfacher Komponenten und Funktionalitäten sowohl zu verwalten, als auch zu warten. Ebenfalls sind durch einheitliche Schnittstellen der Austausch von Nodes einfach und ermöglicht ein flexibles Ökosystem.

## 2.3 Simulation mit Gazebo

Da es besonders beim Flug von Quadrocoptern schnell zu Schäden kommen kann und das sowohl in langen Ausfallzeiten resultieren kann, als auch zu erhöhten Materialkosten führt ist es sinnvoll Testflüge in simulierte Umgebungen auszulagern. Speziell beim Test von autonomem Verhalten ist der Test der Features in realer Um-

gebung somit mit hohem Risiko verbunden. Um dies zu vermeiden ist die Simulation von Quadrocoptern und verschiedener Umgebungen unabdingbar. Ebenfalls erleichtert eine simulierte Version der Drohne die Entwicklung, da sie nicht immer physikalisch vorhanden sein muss. Dabei ist es allerdings notwendig, insbesondere bei Bilddaten, dass die reale Situation möglichst realitätsnah abgebildet wird, so dass das Verhalten im realen Umfeld entsprechend ähnlich ist. Insbesondere bei der Verarbeitung von Kameradaten ist es allerdings abzuwagen ob Ergebnisse in der Simulation mit Resultaten in realen Umgebungen vergleichbar sind, da die simulierten Bilddaten in der Regel eher steril sind.

Durch die Integration von ROS kommt die Simulationsumgebung Gazebo als Bestandteil mit. Anfänglich handelte es sich um ein ROS Paket, mittlerweile ist es allerdings ein eigenständiges Ubuntupaket und benötigt de facto kein ROS zur Laufzeit. Da die realistische Simulation einer Drohne sehr umfangreich ist wird das ROS Paket `tum simulator` verwendet. Es enthält eine Implementation der AR Drone 2.0 für den Gazebo Simulator. Es wurde von Hongrong Huang und Jürgen Sturm aus der Computer Vision Group von der Technischen Universität München entwickelt.[17] Die Drohne wird komplett abstrahiert, wodurch die Simulation sich ohne Veränderung mit der realen Drohne austauschen lässt. Es ist auch möglich sowohl den Quadrocopter real zu steuern, als auch gleichzeitig in der Simulation. Das ermöglicht einen direkten Vergleich zwischen simulierten und realen Ergebnissen, vorausgesetzt, dass die simulierte Umgebung der echten annähernd entspricht. Die Steuerung der Drohne, wird bei dem Package nativ über einen Playstation 3 Controller realisiert. Dessen Eingaben werden von der ROS-Node "Joy Node" verarbeitet und anschließend an die Node `ädrone joystick` weitergeleitet. Dort werden die Daten entsprechend übersetzt um entweder von Gazebo und/oder der realen Drohne verwendet werden zu können. Wenn sie an den realen Quadrocopter gesendet werden sollen, müssen sie noch entsprechend vom `ädrone driver` übersetzt werden, da lediglich Bitfolgen per

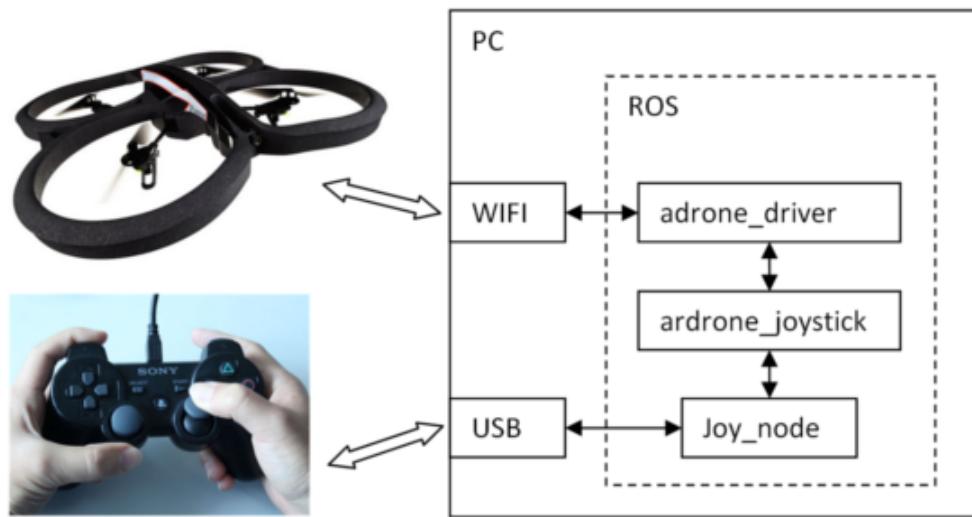


Abbildung 2.3: Programmstruktur mit dem realen Quadrocopter [18]

WLAN versendet werden.

## 2.4 Fuzzylogik

### 2.4.1 Allgemeines

Fuzzylogik, englisch fuzzy für unscharf oder verschwommen, ist eine Theorie die 1965 von Lotti Zadeh entwickelt wurde. Sie beschäftigt sich mit dem Modellierung von unscharfen Wissen und wird hauptsächlich für die Kontrolltheorie, sowie künstliche Intelligenz verwendet. Im Gegensatz zur klassischen Logik, die als Ergebnis nur wahr oder falsch zulässt, kann man mit Fuzzylogik auch die Ausprägung einer Zugehörigkeit feststellen. Diese sogenannte *Fuzziness* ermöglicht Zuordnungen wie "stark" oder "ziemlich". Wesentliche Elemente sind linguistische Variablen, welche, was der Name impliziert, Wörter und Ausdrücke anstellen von Zahlen verwenden um diese unscharfen Mengen zu repräsentieren. Diesen wird eine Funktion zugeordnet um die jeweilige Ausprägung der Variable feststellen zu können.[20]

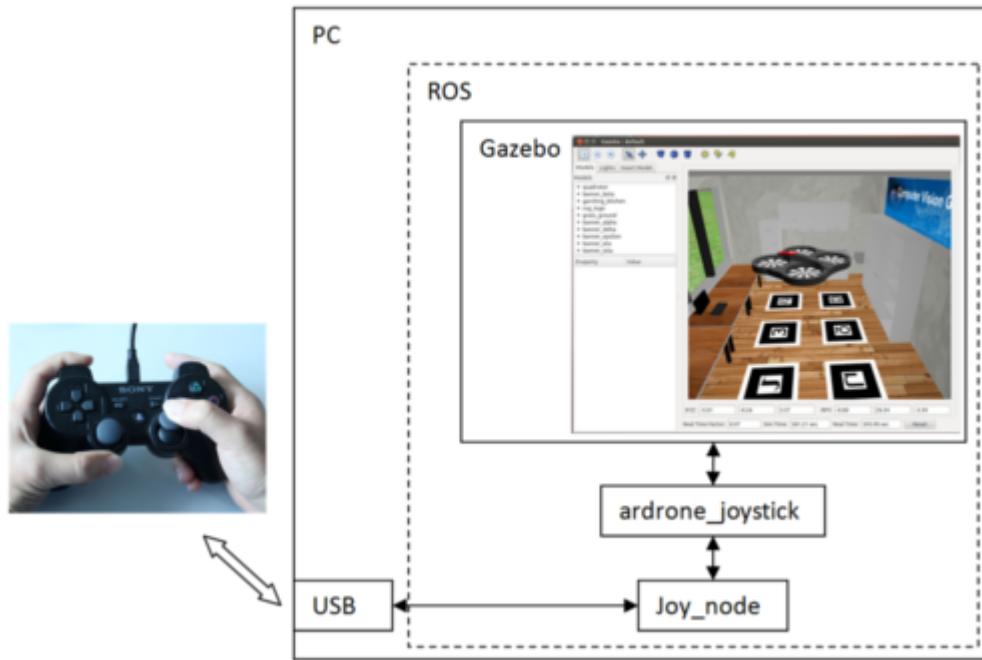


Abbildung 2.4: Programmstruktur unter Verwendung von Gazebo [19]

Im folgenden werden fachspezifische Terme aus der Fuzzylogik verwendet, dafür werden nachfolgende Grundbegriffe kurz erklärt:

- Unscharfe Menge (fuzzy set): Mathematisches Konzept zur Darstellung vager Angaben, welches teilweise Mengenzugehörigkeit erlaubt.[21][22]
- Unscharfe Logik (fuzzy logic): Isomorphes Konzept zur Unscharfen Menge, welches sich um den Umgang mit graduell ausgeprägten Wahrheitswerten kümmert.[21][22]
- Unscharfe Regel (fuzzy rule): Simple Regel, deren Prädikate unscharfe Mengen enthält.[21][22]
- Unscharfes Schließen (fuzzy reasoning): Formalisierung und Auswertung der unscharfen Regelbasen durch einen mathematischen Rahmen.[21][22]
- Unscharfe Regelung (fuzzy control): Einbindung unscharfer regelbasierter Sy-

stem in technischen Anwendungen, wird auch als Fuzzy Interferenz bezeichnet [21][22].

Das verwendete Fuzzymodell besteht insgesamt aus acht linguistischen Variablen, vier Ein- und Ausgabevariablen, von welchen jede fünf Zugehörigkeitsfunktionen besitzt. Weiterhin gibt es einen Regelblock um aus den Eingaben auf die entsprechenden Ausgaben zu schließen.

#### 2.4.2 Fuzzylite

Für die Implementation der Fuzzylogik wird die C++ Bibliothek FuzzyLite von Juan Rada-Vilela verwendet. Es handelt sich dabei um eine gratis Opensource Libary und unterstützt alle gängigen Funktionen der Fuzzylogik.[23]

Listing 2.1: Beispielcode zur Erzeugung eines neuen Fuzzymodells

```
void FuzzyController::init() {
    engine = new Engine;
    engine->setName("input");
    InputVariable* backward = new InputVariable;
    backward->setName("backward");
    backward->setEnabled(true);
    backward->setRange(-1.000, 1.000);
    backward->addTerm(
        new Trapezoid("strongForward", -1.000, -1.000, -0.800, -0.400));
    backward->addTerm(
        new Trapezoid("mediumForward", -0.900, -0.600, -0.400, 0.000));
    backward->addTerm(
        new Trapezoid("mediumBackward", 0.000, 0.400, 0.600, 0.900));
    backward->addTerm(
        new Trapezoid("strongBackward", 0.400, 0.800, 1.000, 1.000));
    engine->addInputVariable(backward);
}
```

Mit dem Code erzeugt man zunächst eine neue Engine und anschließend eine Eingabevariable. Dieser wird ein Wertebereich zugewiesen und Zugehörigkeitsfunktionen erstellt. Schlussendlich wird die Variable bei der Engine registriert.

Listing 2.2: Code zur Erzeugung von einem Regelsystem

```
RuleBlock* ruleBlock = new RuleBlock;
ruleBlock->setEnabled(true);
ruleBlock->setConjunction(new Minimum);
ruleBlock->setDisjunction(new Maximum);
ruleBlock->setImplication(new Minimum);
ruleBlock->setActivation(new General);
ruleBlock->addRule(Rule::parse(
    "if_backward_is_strongForward_then_backwardSpeed_is_strongForward",
    engine));
engine->addRuleBlock(ruleBlock);
```

Mit diesem Code erzeugt man ein neues Regelsystem und fügt eine beispielhafte Regel hinzu. Ein komplettes System besteht aus wesentlich mehr Regeln, da es normalerweise komplex ist.

Listing 2.3: Code zur Auswertung des Regelsystems

```
backward->setValue(back);
sideward->setValue(side);
up->setValue(upValue);
rotation->setValue(rotateRight);
engine->process();
```

Schließlich kann man den Eingabebenennen konkrete Werte zuweisen und das System mithilfe von unscharfen Schließen auswerten.

## 2.5 Kinect

### 2.5.1 Allgemeines

Um Gesten des Benutzers zur erkennen und entsprechend auszuwerten wird der visuelle Sensor Microsoft Kinect verwendet. Normalerweise wird er zur Steuerung der Videospiel Konsole Xbox360/Xbox One verwendet. Das System ist in der Lage Tiefenbilder zu erstellen, besitzt sowohl eine 1080p Farbkamera, als auch einen Infrarotsensor und mithilfe mehrere Mikrofone Sprache und Bewegung im Raum zu erkennen. Anhand der verschiedenen Kameraeingaben ist es möglich den Körper von bis zu 2 Nutzern zu erkennen und auch entsprechend zu verfolgen. Für Windows stellt Microsoft ein entsprechendes Software Development Kit (SDK) zur Verfügung um die Benutzung der Funktionalität zu vereinfachen.[24] [25] [26]

### 2.5.2 Vergleich der verschiedenen Stacks zur Implementation

Für ROS existieren zwei Stacks für die Verwendung des Kinect Sensors:

- Freenect Stack[27]
- OpenNI Stack[28]

Beide Implementationen liefern einen Treiber um den Sensor anzusprechen und die Kameradaten auszulesen. Allerdings gibt es auch einige Unterschieden zwischen den zwei Stacks. Der Freenect-Treiber ermöglicht es neben den Kameradaten ebenfalls die LED, einen Beschleunigungssensor, sowie die Audiodaten der Kinect zu bedienen.[27] Dies ist mit dem Treiber von OpenNI allerdings nicht möglich. Dafür kommt auf dessen Stack noch ein Tracker mit dem es möglich ist Benutzer zu erkennen und deren Bewegungen zu tracken. Ebenfalls ermöglicht OpenNI die Segmentation der Bilddaten, Handerkennung und Gestensteuerung. Die extrahierten Merkmalspunkte können weiter verwendet werden. Beide Implementierungen sind Open

Source und stehen unter der Apache2.0 Lizenz[29][27]. Durch das Skeletontracking biete OpenNI allerdings einiges an Vorteilen für die Implementation der Steuerung, daher fällt die Wahl auf diesen Stack, da für die Usecase wesentlich besser passt.

## 2.6 Punktwolken

Die Tiefenbilder die REMODE erstellt sind in Form von sogenannten Punktwolken, bzw. Point Clouds. Diese Punktwolken sind eine Menge von Punkten in einem Vektorraum, welche jeweils durch ihre Raumkoordinaten in einem dreidimensionalen kartesischen Koordinatensystem beschrieben sind. Somit ist jedes Element im Datensatz durch die Attribute X, Y und Z gekennzeichnet. [30][2]

Sobald ein Tiefenbild approximiert wurde, werden diese Informationen über das ROS Topic */remode/depth/pointcloud* geteilt.



Abbildung 2.5: Beispiel einer visualisierten Punktwolke [2]

Die Darstellung zeigt beispielhaft die Visualisierung einer Punktwolke. Dabei entscheidet die Helligkeit der Punkte über ihre relative Tiefe in Bezug zur Kamera. Der

Vektorraum  $V \in \mathbb{R}^3$  mit den entsprechenden Datenpunkten wird das ROS Topic in einem Intervall von wenigen Sekunden übertragen.

## 2.7 Point Cloud Library

Die Punktwolken, wie in Abschnitt 2.6 beschrieben, bilden die Grundlage, um essenzielle Auswertungen und Analysen für das Assistenzsystem auszuführen. Da hierfür teilweise komplexe Algorithmen und mathematische Berechnungen notwendig sind, würde die eigene Implementierung den Rahmen dieser Arbeit sprengen.

Das OpenSource Projekt Point Cloud Library *PCL* stellt für diesen Anwendungsfall in dessen Programmzbibliothek zahlreiche Algorithmen bereit, die einfach implementiert werden können. Diese helfen bei Problemen der 2D und 3D Bildverarbeitung, sowie der Punktwolkenverarbeitung. Dabei werden Anwendungen in der Bildverarbeitung bereitgestellt, wie die Filterung, Segmentierung und Visualisierung von Punktwolken.

Zusätzlich verfügt die PCL eine Schnittstelle zu ROS, wodurch sie sich bestens in die Projektumgebung integrieren lässt.

# 3 Software Architektur

## 3.1 Anforderungen

Im Folgenden sollen die Anforderungen an die Studienarbeit festgelegt werden. Diese gliedern sich in viel Hauptbestandteile auf.

Der erste Teil besteht darin, ein bestehendes Vorgängerprojekt in eine andere Umgebung zu portieren. Das Projekt ermöglicht die Steuerung einer AR.Drone 2.0 mit Hilfe von Gesten. Die Gesten sind vorgeschriebene Positionen der Arme. So soll der Nutzer beispielsweise mit einer Bewegung von beiden ausgestreckten Armen, nach oben und unten, die Drohne starten, steigen, senken und landen lassen können.

Dabei wurde die Erkennung der Positionen mit einer Kinect Stereo Kamera von Microsoft umgesetzt. Die Daten der Kamera werden im Programmcode verarbeitet. Dieser wurde in der Sprache C# geschrieben und funktioniert auf Grund einer Vielzahl von externen Libraries, wie die Anbindung der Kinect, nur unter Windows.

Als Grundlage für die weiteren Ziele dieser Arbeit soll dieses Projekt für UNIX basierte Betriebssysteme umgeschrieben werden. Grundlage für die Softwarearchitektur bildet das ROS Framework, welches eine Modularisierung der Projektbestandteile in ROS Nodes vorgibt. Ziel soll es sein, die Funktionalität des Bestandsprojektes vollständig nachzubilden.

Die zweite Anforderungsteil besteht darin, dass jegliche Funktionalität auch in einer Simulation verfügbar sein soll. Somit kann man für Präsentationen, in denen der Flug einer Drohne nicht möglich ist, den Quadrocopter in einer frei gestaltbaren, si-

mulierten Umgebung fliegen lassen.

Es soll in diesem Zusammenhang möglich sein, problemlos zwischen einer realen und einer simulierten Drohne wechseln zu können. Ein weiterer Bestandteil dieser Arbeit umfasst die Approximation von Tiefeninformationen. Dafür soll ausschließlich eine unveränderte AR.Drone verwendet werden, welche nur mit einer monokularen<sup>1</sup> Frontkamera ausgestattet ist.

Um aus den Bildern dieser Kamera Tiefenbilder zu gewinnen, soll ein externes Projekt in die Projektlandschaft integriert werden. Hierbei soll es wiederum möglich sein, dass der Videostream sowohl von der realen, als auch von der simulierten Drohne gesendet werden kann. Es soll dabei ermittelt werden, ob die Nutzung der externen Arbeiten für den Anwendungszweck praktikabel und sinnvoll ist.

Der vierte Bestandteil dieser Arbeit besteht im Erarbeiten, Testen und Bewerten von möglichen Ansätzen und Limitationen der Implementierung eines Assistenzsystems. Das Ziel ist es herauszufinden, wie mit den Tiefeninformationen die manuelle Steuerung der Drohne durch eine Person, mit Hilfe von Assistenzfunktionen, unterstützen werden kann.

Ein Anwendungsbeispiel ist hierbei das sichere Fliegen durch ein Hindernis wie eine offene Tür, oder das verhindern von Kollisionen mit Objekten in der Umgebung.

---

<sup>1</sup>Monokular ist die Bezeichnung für Kameras mit einer einzelnen Linse

## 3.2 Bildverarbeitung

In diesem Abschnitt wird beschrieben, wie die Position der Kamera im Raum bestimmt werden kann, um anschließend aus aufeinanderfolgenden Aufnahmen Tiefeinformationen zu gewinnen. Die beschriebenen Implementierungen beziehen sich dabei auf die Arbeit von Christian Forster, Matia Pizzoli und Davide Scaramuzza, welche ihre Abhandlungen zum Thema zusammen mit dem Programmcode frei zugänglich gemacht haben.

### 3.2.1 Semi-Direct Monocular Visual Odometry - SVO

Eine Grundanforderung an das Projekt ist die Nutzung einer nicht modifizierten AR.Drone. Dadurch entsteht die Problematik, dass keine Tiefenbildkamera genutzt werden kann, um in Echtzeit Tiefebilder zu erhalten. Die Drohne ist lediglich mit einer monokularen Frontkamera ausgestattet.

Um Tiefeinformationen aus den Bildern einer solchen Kamera zu gewinnen, wird eine Szene aus verschiedenen Perspektiven aufgenommen. Anschließend gibt es unterschiedliche Ansätze um aus den aufeinanderfolgenden Bildern Kamerapositionen und Umgebungsstrukturen zu ermitteln.

Feature basierte Ansätze sind der aktuelle Standard zur Berechnung der Kameraposition. Diese versuchen die wichtigsten Merkmale eines Bildes, die Features, zu extrahieren. Mit Hilfe von Feature Deskriptor Algorithmen werden Vektoren mit Informationen zu invarianten Bildbereichen berechnet. Diese Vektoren verhalten sich wie ein einzigartiger Fingerabdruck, der die Merkmale repräsentiert.

Aufeinanderfolgende Bilder werden dann mit Hilfe dieser Deskriptoren abgeglichen und sowohl Kamerabewegungen, als auch Strukturen werden rekonstruiert. Zur Optimierung sind abschließend die ermittelten Kamerapositionen anzulegen. Dies

geschieht mit Hilfe von Algorithmen zur Minimierung des Reprojektionsfehlers.<sup>2</sup> Ein weiterer Ansatz ist die direkte Methode. Hierbei werden die Features nicht über Deskriptor Algorithmen bestimmt, sondern das Problem wird über die Intensitäten der Pixel gelöst. Bei einem Graustufenbild entspricht diese Intensität der Helligkeit von Bildbereichen.

Somit kann bei der Rekonstruktion im Gegensatz zum Feature basierten Ansatz auch die Richtung der Gradienten von Intensitäten genutzt werden. Dadurch funktioniert diese Methode auch bei Bildern mit sehr wenig Textur, Bewegungsunschärfe und fehlerhaftem Kamerafokus.

Das für diese Arbeit relevante Vorgehen kombiniert die Vorteile der beschriebenen Methoden. Die semi-direkte Odometrie<sup>3</sup> verwendet einen Algorithmus der ebenfalls auf Zusammenhängen von Features basiert. Diese werden jedoch implizit aus einer direkten Bewegungsabschätzung bezogen, anstatt explizit durch Algorithmen mit Feature Deskriptoren berechnet zu werden.

Dadurch müssen Features nur extrahiert werden, wenn diese noch nicht auf einem der vorherigen Bildern vorhanden waren.

Insgesamt ist dieser Ansatz sehr schnell, da wenig Berechnungen pro Bild stattfinden und auf Grund der Verwendung von Intensitätsgradienten äußerst genau und robust.

Diese Eigenschaften sind für die Anforderungen der Studienarbeit essentiell, da die Drohne sich sehr schnell bewegen kann und somit in möglichst kurzer Zeit neue Bilder auswerten muss. Das beschriebene Verfahren minimiert damit die Auswirkungen der typischen Probleme von Drohnen. Diese sind einerseits die niedrige Texturierung der Umgebung, welche hauptsächlich in Innenräumen auftritt und andererseits kameraspezifische Probleme wie Bewegungsunschärfe und der Verlust des Kamera-

---

<sup>2</sup>Reprojektionsfehler sind geometrische Fehler die im Zusammenhang zwischen abgebildeten und berechneten Bildpunkten entstehen. [31]

<sup>3</sup>Odometrie bezeichnet die Verwendung Bewegungssensordaten zur Bestimmung der Positionsänderung über die Zeit.[32]

fokus.

Im Folgenden zeigt die Abbildung wie die Nutzung von SVO in der Praxis aussieht.

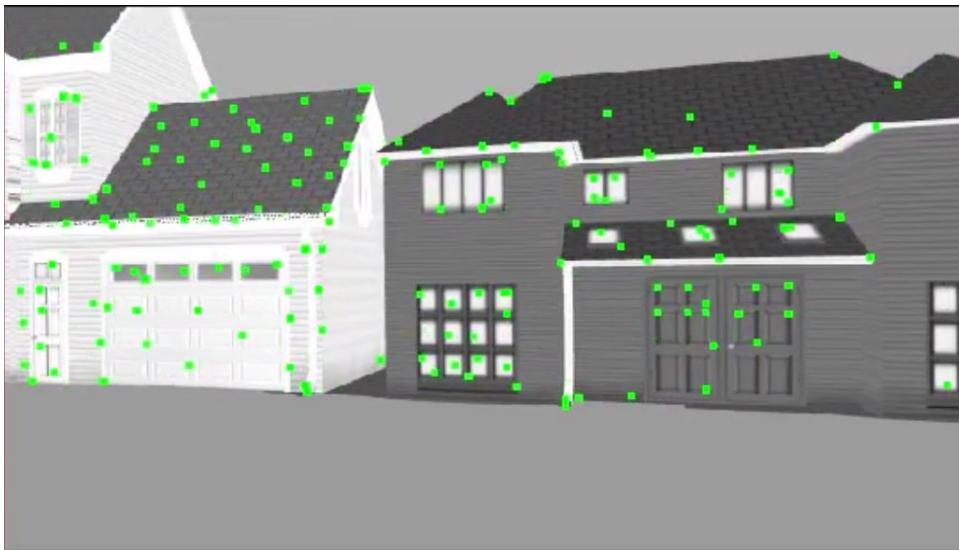


Abbildung 3.1: Semi-Direct Monocular Visual Odometry im Simulator

Hierbei stammt das Kamerabild von einer Drohne im Simulator Gazebo in einer frei verfügbaren Testwelt mit einigen Gebäuden. Die grünen Punkte sind die Features, die SVO anhand der beschriebenen Strategie ermittelt hat.

Die Anzahl der aktuell gefundenen Features kann dabei stark variieren. Diese Schwankung entsteht hauptsächlich durch die unterschiedliche Texturierung und Anzahl der Kanten von Objekten in der Umgebung.

Weiterhin kann auch die Kamera selbst ein Grund für eine geringe Anzahl gefundener Features sein. Gründe und Lösungen dafür werden im Folgenden beschrieben.

### 3.2.2 Kamerakalibrierung

Ein Grundproblem der Bildverarbeitung ist die Verzerrung des Bildes. Da die Bestimmung der Kameraposition möglichst genau sein soll, muss die Kamera vorher

kalibriert werden. Dies bedeutet, dass Ungenauigkeiten der Linse erkannt und softwareseitig ausgeglichen werden. Dafür werden die intrinsischen Parameter der Kamera bestimmt

### Kalibrierungsmodelle

Hierbei unterstützt SVO drei Kamera Modelle: ATAN, Ocam und Lochkamera. [33] Das ATAN Modell basiert auf dem *Field of View* (FOV) Verzerrungsmodell SStraight lines have to be straight [...] von Devernay und Faugeras.

Der Vorteil dieser Kalibrierungsmethode ist die äußerst schnelle Berechnung der Projektion des Bildes. Das Modell vernachlässigt jedoch tangentiale Verzeichnung, welche auftritt, wenn optische und mechanische Bestandteile des Objektives, sowie der CCD-Sensor<sup>4</sup> nicht perfekt zueinander ausgerichtet sind. Weiterhin sollte die Kamera mit einem globalem Shutter ausgestattet sein, um die Extraktion von Bildmerkmalen bei Bewegungen zu gewährleisten. Kameras mit Global-Shutter-CMOS<sup>5</sup> Sensoren und CCD-Sensoren nehmen Bild nicht zeilen- und spaltenweise, sondern vollständig auf und sind daher für das Verfahren geeignet.

Die Drohne besitzt eine veraltete und günstige Kamera mit einem CMOS Sensor, wodurch sowohl tangentiale Verzerrung, als auch der Rolling-Shutter-Effekt auftreten können.

Daher ist das ATAN Modell zwar eine der besten Kalibrierungsmethoden für teure Hochleistungskameras, jedoch ist es für die Betrachtungen dieser Arbeit nicht optimal.

Der zweite Ansatz zur Kalibrierung ist das Ocam Modell von Davide Scaramuzza.

---

<sup>4</sup>CCD steht für *charge-coupled device*, was übersetzt ladungsgekoppeltes Bauteil bedeutet. Dieses lichtempfindliche elektronische Bauteil wird zur Bildaufnahme verwendet.

<sup>5</sup>CMOS steht für *Complementary metal-oxide-semiconductor* und ist ein spezieller Halbleiter der zur Bildaufnahme verwendet wird.

Diese Methode sollte für Kameras mit sehr weitem Sichtfeld, oder omnidirektionalen Kameras genutzt werden. Damit ist es für die Drohne nicht geeignet.

Die dritte unterstützte Kalibrierungsmethode ist das Modell der *Lochkamera*, bzw. auf Englisch *Pinhole Model*. Der Name und das zugrunde liegende Prinzip basieren, wie der Name es sagt, auf der Lochkamera. Die Abbildung zeigt die grundsätzliche Funktionsweise.

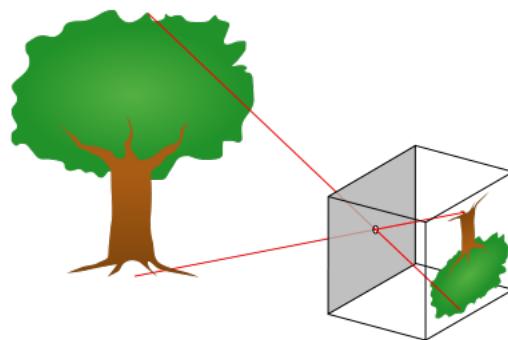


Abbildung 3.2: Prinzip der Lochkamera [3]

Die Darstellung zeigt, dass bei der Lochkamera Licht durch eine kleine Öffnung in einen kleinen Hohlkörper fällt. Dadurch entsteht auf der Rückseite ein auf dem Kopf stehendes Bild.

Bei dem Pinhole Model handelt es sich um den aktuellen Standard in OpenCV<sup>6</sup> und ROS. Hierbei wird die Verzerrung mit Hilfe von fünf intrinsischen Parametern beschrieben, welche im Rahmen der Kalibrierung bestimmt werden müssen.

$$\text{Distortion}_{\text{coefficients}} = (k_1 \ k_2 \ p_1 \ p_2 \ k_3)$$

OpenCV betrachtet dabei radiale und tangentiale Faktoren. Die Formel für radiale Verzeichnung ist die Folgende:

---

<sup>6</sup>“OpenCV is the leading open source library for computer vision, image processing and machine learning, and now features GPU acceleration for real-time operation.”

$$\begin{aligned}x_{\text{corrected}} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\y_{\text{corrected}} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6)\end{aligned}$$

Hierbei wird aus einem alten Bildpunkt  $(x, y)$  des Eingabebildes die korrigierte Position  $x_{\text{corrected}}, y_{\text{corrected}}$  bestimmt.

Die Berechnung der tangentialen Verzerrung erfolgt durch die Formel:

$$\begin{aligned}x_{\text{corrected}} &= x + [2p_1 xy + p_2(r^2 + 2x^2)] \\y_{\text{corrected}} &= y + [p_1(r^2 + 2y^2) + 2p_2 xy]\end{aligned}$$

Abschließend werden die Einheiten angepasst:

$$\begin{bmatrix}x \\ y \\ w\end{bmatrix} = \begin{bmatrix}f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1\end{bmatrix} \begin{bmatrix}X \\ Y \\ Z\end{bmatrix}$$

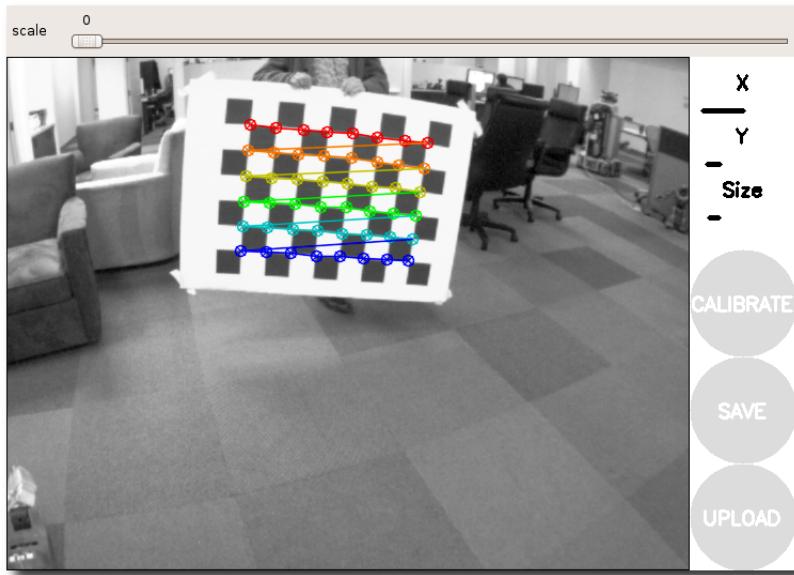
Dabei entspricht  $f_x$  und  $f_y$  der Brennweite der Linse und  $c_x$ , sowie  $c_y$  beschreiben die optische Bildmitte in Pixelkoordinaten.

Dieser Ansatz ist der einfachste und funktioniert grundsätzlich mit jeder Kamera.

### Umsetzung der Kalibrierung

Um die intrinsischen Parameter der Kamera zu bestimmen, wird ein bekanntes Bild oder Muster aufgenommen. Dazu wird ein Vergleich zwischen den theoretischen und tatsächlichen Abmessungen angestellt. Hierzu wird meist ein einfaches Schachbrettmuster genutzt, welches im möglichst vielen verschiedenen Perspektiven aufgenommen wird. Bei der realen Drohne wird dazu das Muster ausgedruckt, bei der Simulation muss hingegen ein solches Objekt in die Welt eingefügt werden. Da die Kamera in der Simulation ohnehin keine intrinsischen Fehler aufweisen sollte, kann auf eine Kalibrierung verzichtet werden.

Die Kalibrierung wurde mit dem frei verfügbaren ROS Node *camera\_calibration* umgesetzt. Das Ergebnis ist abhängig von der Anzahl der Perspektiven und der



Qualität der Aufnahmen. Aufgegeben wird dann die List der Parameter, die für das Lochkamera Modell notwendig sind.

### 3.2.3 Regularized Monocular Depth Estimation - REMODE

Im vorherigen Abschnitt wurde beschrieben, wie anhand von aufeinanderfolgenden Bildern die Position der Kamera im Raum bestimmt werden kann. Dieses Problem wird seit mehr als 20 Jahren untersucht und wird als Structure From Motion *SFM* in der Bildverarbeitung und Simultaneous Localization and Mapping *SLAM* in der Robotik bezeichnet.

Um den Nutzer aktiv bei der Steuerung der Drohne zu unterstützen werden jedoch Tiefeninformationen benötigt. Dazu müssen Tiefenbilder und Tiefenkarten (footnote) aus den Bildern der monokularen Kamera bestimmt werden.

Für diesen Schritt gibt es unterschiedliche Ansätze. Der State of the Art ist die Berechnung mit Hilfe des Bayes-Schätzers. Dabei handelt es sich um eine Schätzfunktion in der mathematischen Statistik, welche eventuell vorhandenes Vorwissen bei

der Schätzung eines Parameters berücksichtigt. Dabei wird in der bayesschen Statistik das initiale Vorwissen mit Hilfe der A-priori-Verteilung modelliert, die bedingte Wahrscheinlichkeit des Parameters unter Betrachtung dieses Vorwissens mit der A-posteriori-Verteilung.

Im Umfang dieser Arbeit wird die Forschung und Implementierung des Projekts Regularized Monocular Depth Estimation *REMODE* genutzt. In dieser Ausarbeitung von Matia Pizzoli, Christian Forster und Davide Scaramuzza wird die bayessche Schätzung mit neusten Entwicklungen in der Konvexoptimierung verbunden. Hierbei stützen sie ihre Forschungen auf die Ergebnisse von G. Vogiatzis und C. Hernandez und ihrer Abhandlung mit dem Titel "Video-based, real-time multi-view stereo" von 2011.

*REMODE* erfüllt den Zweck, mit Hilfe der gewonnenen Informationen ein dreidimensionales Modell des Raumes zu erstellen. Die Abbildung zeigt ein Beispiel dazu, welches von den Entwicklern verbreitet wurde.

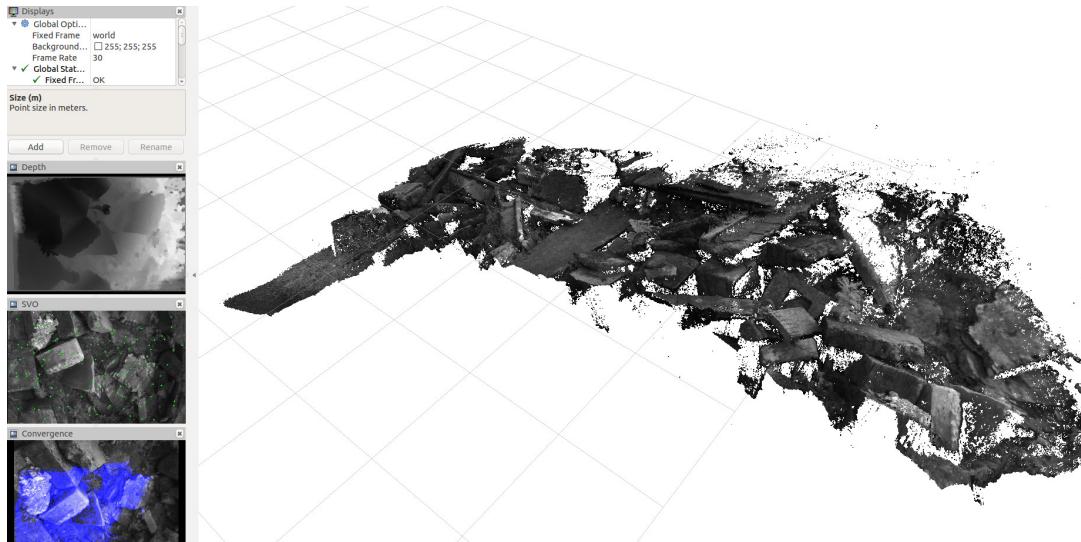
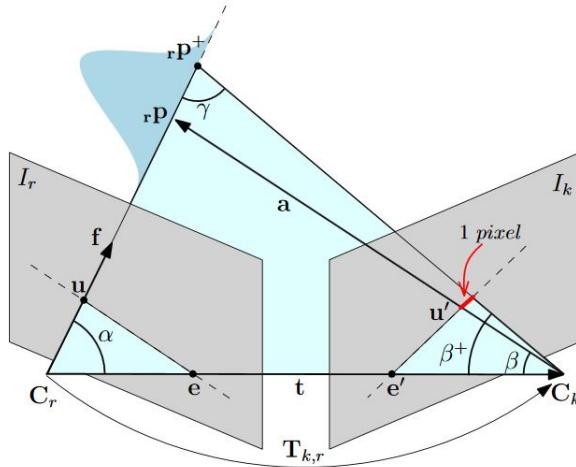


Abbildung 3.3: REMODE 3D-Modell, Flug über Trümmer [4]

Die Darstellung zeigt ein 3D Modell eines Trümmerhaufens, welches mit Hilfe der aufeinanderfolgenden Einzelbildern generiert wurde. Die Visualisierung erfolgte mit dem Standard 3D Visualisierungstool für ROS, genannt *RVIZ*.

REMODE ermöglicht eine Berechnung der Tiefeninformationen in Echtzeit und auf Pixelbasis. Weiterhin ist die aktuelle Genauigkeit und Fehlerrate im Vergleich zur Realität zu jeder Zeit sichtbar.

Im Folgenden wird der grundsätzliche Ansatz der Implementierung skizziert. Die genauen Details übersteigen dabei auf Grund der hohen Komplexität den Umfang dieser Arbeit.



In der Übersicht sieht man die beiden Kamerapositionen  $I_r$  und  $I_k$  mit den zugehörigen Kamerazentren  $C_r$  und  $C_k$ . Die Positionen im Raum dieser Kameras wurden im vorherigen Schritt mit Hilfe von SVO ermittelt.  $T_{k,r}$  zeigt dabei die starre Transformation der Kamerabilder.

Der Punkt  $rP$  ist die aktuelle Schätzung der Position eines Punktes auf den epipolaren Flächen. Die Varianz der Abweichung von einem Pixel auf der epipolaren Linie durch estrich und ustrich wird berechnet mit der Gleichung  $tkquadr$ . Mit Hilfe der oben angesprochenen mathematischen und statistischen Auswertungen kann nun

die Tiefe eines Punktes  $r_P$  approximiert werden.

Das Zusammenspiel von SVO und REMODE ist für handelsüblichen Laptops mit CPU und GPU ausgelegt. Dabei läuft SVO komplett auf dem Prozessor und REMODE verwendet das Framework NVIDIA CUDA, (footnote) was auf den Grafikchip des Rechners zugreift.

Die Implementation setzt auf eine durchschnittliche Bewegung der Kamera von 0.0038 Meter pro Sekunde und einer mittleren Tiefe von einem Meter bei einer Berechnungszeit von 3.3 ms pro Bild.

### 3.2.4 Performanceprobleme

Trotz der Nutzung neuster Methoden und Techniken zur Berechnung der Kamera positionen und Tiefenbilder, gibt es Probleme hinsichtlich der Performance. Dabei ist das Hauptproblem die Differenz im verfolgten Ziel zwischen dieser Arbeit und der Implementierung von SVO und REMODE.

Der Hauptanwendungszweck ist ein langsamer, stetiger Flug einer Drohne über ein Gebiet. Dabei zeigt die aufnehmende Kamera nach unten und hat sowohl eine sehr hohe Qualität, als auch ein großes Sichtfeld von mehr als 110 Grad. Die Bewegungen der Drohne sind nur nach seitlich, nach vorne und nach hinten, nicht jedoch um die eigene Achse. Somit wird sicher gestellt, dass immer genügend Referenzfeatures vorhanden sind, damit zu jedem Zeitpunkt die Position der Kamera bekannt ist.

Im Gegensatz dazu sind die Anforderungen der Arbeit stark abweichend. So ist sowohl die Qualität, als auch die Verarbeitung der Kamera minderwertig. Auch das Blickfeld ist mit 90 Grad deutlich zu klein, wodurch weniger Features auf einem Bild Platz finden.

Dadurch, dass die Kamera nach vorne und nicht nach unten gerichtet ist, treten weitere Komplikationen auf. Gleiche Bewegungen verursachen somit größere Änderun-

gen am Bild, wodurch mehr Berechnungen notwendig werden und somit die Fehleranfälligkeit steigt. Vor allem Drehbewegungen um die eigene Achse sorgen für erhebliche Änderungen und führen meist zum Abbruch von SVO.

Auch die Kalibrierung der Kamera der realen Drohne war auf Grund der schlechten Qualität schwierig. So konnte der Richtwert des Reprojektionsfehlers von rund 0,1 Pixel nicht erreicht werden, sondern lag eher bei 0,3 Pixel.

Bei Tests in Innenräumen mit der AR.Drone konnte SVO nicht mehr als 50 Features finden. Um jedoch die Position der Kamera zu bestimmten, sind mehr als 100 Features notwendig. Diese Beobachtungen aus den Testläufen decken sich mit den Hinweisen zur Performance in der Projektdokumentation[34].

### 3.3 Implementierung

#### 3.3.1 Grundlegende Herausforderungen

Es bestand eine nicht modularisierte, schlecht dokumentierte C# Dokumentation für eine Windowsumgebung. Diese galt es teilweise für ROS zu übernehmen. Für Windows existiert ein Kinect Software Development Kit, welches die Programmierung erleichtert. Ein weiteres generelles Problem war die Versionsinkompatibilitäten verschiedener ROS-Nodes untereinander, der ROS-Version oder mit der Betriebssystemversion. Hierbei ist es notwendig das Zusammenspiel verschiedenster Versionen zu testen und zwischen Alternativen abzuwählen.

#### 3.3.2 Architektur

Die Architektur der Anwendung wird maßgeblich durch die Verwendung des Robotic Operating System geprägt, da es eine rahmenartige Struktur bildet. Durch dieses framework-artige Verhalten bestimmt es vorrangig die Programmstruktur, welche hauptsächlich aus verschiedenen Nodes besteht. Grob betrachtet, kann man sie in 2 grundlegende Bereiche trennen:

- Assistenzsystem
- Steuerung des Quadrocopters

Die Bereiche sind alleine nutzlos, zwar können sie ihren Zweck gut erfüllen, allerdings werden die Ergebnisse nicht verwertet. Dadurch ist die Kommunikation der Bestandteile für ein optimal funktionierende Anwendung essentiell.

Ein wesentlicher Anforderung an die Architektur ist, das Simulator und Drohne ohne großen Aufwand ausgetauscht werden können. Hier hilft wiederum die ROS Umgebung optimal das umzusetzen, da durch die lose gekoppelten Nodes, die lediglich mit Hilfe eines Message Brokers kommunizieren, Komponenten einfach ausge-

tauscht werden können. Ob mit oder ohne Simulator gestartet wird entscheidet die Wahl der entsprechenden Launch File, wo je nach dem entweder der Treiber für den realen Quadrocopter gestartet wird oder lediglich die Simulationsumgebung. Ebenfalls gut austauschbar ist die Art der **Steuerung des Quadrocopters**, zwar liegt die Kontrolle via Kinect im Fokus, dennoch ist es möglich ein zusätzlichen Tastaturkontroller zu starten und diesen sogar parallel zu betreiben, da die Ansteuerung über ein und dieselbe Topic geschieht. Dies bietet auch Freiraum für zusätzliche Erweiterungen um zur Steuerung mit Kinect noch einen Spotter einzubringen, der im Notfall eingreifen kann.

Die Steuerung durch Gesten wird im Kinect Controller realisiert, wo die Bilddaten des optischen Sensors eingelesen werden. Der Nutzer wird erkannt, dessen Bewegungen getrackt und anhand dessen mit Hilfe von Fuzzy Logik ein Steuerbefehl erzeugt. Auf die konkrete Implementierung wird in den nachfolgenden Abschritten näher eingegangen.

Die Hauptaufgabe des **Assistenzsystem** ist die Verarbeitung der Bildeingaben und einer eventuellen Korrektur der Steuerbefehle zum Vorteil des Benutzers. Die Bilddaten des Quadrocopters, sowie auch die Bilder des Simulators werden über dieselbe Topic versandt, wodurch es wiederum keinen Unterschied für die Bildverarbeitung macht, ob es simuliert wird oder es sich um ein reales Geschehen handelt. Diese Kameraaufnahmen werden Frame für Frame an die SVO-Node gesendet, wo anhand der Bildverschiebung, welche durch einen leichten Drift des Quadrocopters bedingt wird, Features berechnet und anschließend getrackt werden. Diese Featuredaten können von der REMODE-Node nachfolgend verarbeitet werden und mit deren Hilfe wird versucht ein Tiefenbild beziehungsweise eine Punktwolke zu erstellen. Die Punktwolke kann anschließend analysiert werden, wobei hauptsächlich versucht wird Hindernisse und Öffnungen, zum Beispiel Türen oder Fenster, zu erkennen und je nach aktuellem Befehlsstatus eventuell einzugreifen und dem Nutzer

zu assistieren. Für bestimmte Konstellation können vorgefertigte Steuerungsabläufe durchgeführt werden um den Quadrocopter zum Beispiel durch ein Fenster zu führen oder um zu verhindern, dass er gegen eine Wand oder Hindernis fliegt.

Ebenso ist theoretisch das Assistenzsystem komplett austauschbar oder erweiterbar in dem man einfach noch ein zusätzliches hinzufügt. Das ist möglich, da die Befehle in der Steuereinheit vom Kinect Controller priorisiert werden, wenn dies intelligent geschieht lässt sich das System um eine Vielzahl von Assistenten erweitern. Die Abbildung 3.3.2 stellt eine grob vereinfachte Visualisierung der Architektur dar und dient dazu einen groben Überblick zu schaffen wie die Komponenten miteinander in Relation stehen. Im Folgenden werden die konkret implementierten Bestandteile genauer betrachtet.

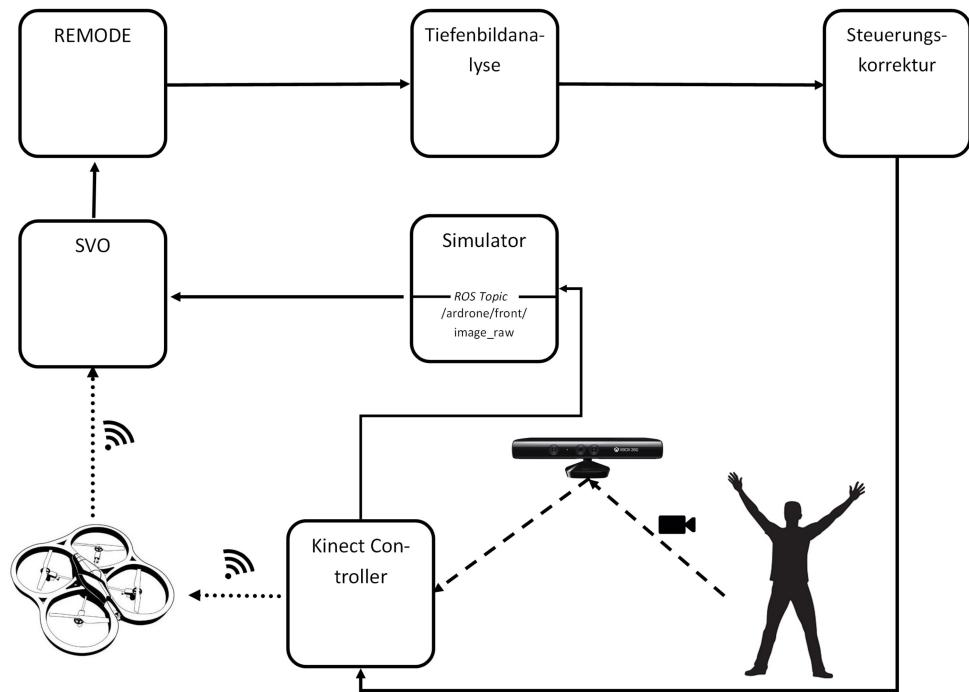


Abbildung 3.4: Übersicht zur Softwarearchitektur

### 3.3.3 Ansteuerung der Kinect

Der visuelle Sensor Microsoft Kinect dient als Nutzerschnittstelle zur Bedienung, indem der Nutzer Gesten benutzt um die Drohne zu steuern. Um die Sensoreingaben richtig verwenden zu können sind 3 Schritte notwendig:

- Korrekte Ansteuerung der Sensorhardware
- Körpererkennung und Tracking
- Verarbeitung der separaten Merkmale

Hierfür ist der erste Schritt die Hardware des Sensors richtig anzusteuern. Da die Microsoft Software nicht für Linux Systeme kompatibel ist, kommt einen Open Source Treiber zum Einsatz. Hierbei handelt es sich um OpenNI[28], welches auch eine Middleware bereit stellt um Körperbewegung und Gesten zur erkennen, sowie zu tracken. Die ROS-Node “openni\_launch”[35] realisiert die Ansteuerung dess Kinect Sensors. Ebenfalls wird der zweite Schritt: Körpererkennung und Tracking durch OpenNI gelöst, da die Middelware auch wie oben erwähnt dies kann. Das geschieht mithilfe der ROS-Node “openni\_tracker”[36]. Für die den dritten Punkt: Verarbeitung der seperaten Merkmale ist die selbst implementierte ROS-Node “kinect\_controller” zuständig. Dort werden die durch OpenNI extrahierten Merkmale, wie zum Beispiel linke Hand, rechte Hand, Kopf, usw. als Grundlage eingegeben und Ziel ist es die entsprechenden Befehle anhand der Eingaben für den Quadrocopter zu erstellen.

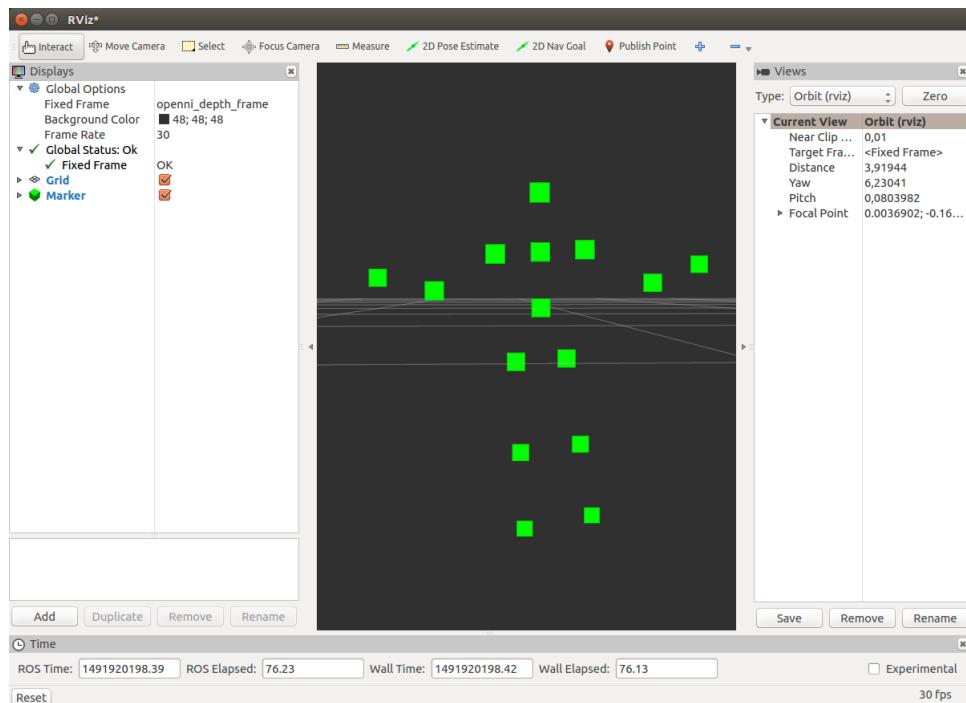


Abbildung 3.5: Darstellung der Körperteile in RVIZ

### 3.3.4 Kinect Controller

Der Kinect Controller ist eine eigenständige ROS-Node welche sich vorrangig um die Vorverarbeitung der Daten von der Kinect, in dem Falle die Ausgabe der Node des OpenNi Trackers, kümmert. Der Tracker veröffentlicht dauerhaft die aktuellen Transformationen, sowie Translationen der Koordinatensysteme der einzelnen Körperfunkte. Dies geschieht über die Topic “/tf”, wo die Nachrichten mit dem Typ “TFMessage.msg”[37] verschickt werden. Die Klasse registriert sich bei der Initialisierung auf diesen Nachrichtenkanal und empfängt mithilfe der Methode “messageCallback” die Nachrichten. Allerdings wird pro erkanntem Merkmal jeweils eine Nachricht versandt, weswegen die berechneten Punkte mithilfe einer weiteren C++ Klasse “SkeletonPoints” aggregiert werden. Diese Klasse ist allein dafür ausgelegt um die einzelnen Punkte zu verwalten und somit für andere Komponenten einfach zugänglich zu machen. Die Zuordnung der Punkte geschieht mithilfe des Attributes “child\_frame\_id” der jeweiligen Nachricht in welchem der Name des erkannten Körperteils angegeben ist. Nachdem die Daten aggregiert wurden, kann mithilfe des Fuzzy Controllers berechnet werden, wie der Quadrocopter gesteuert werden soll. Dieser Steuerbefehl wird anschließend als Nachricht in der Topic “/drone\_command” publiziert. Dafür ist ein eigener Nachrichtentyp notwendig, welcher die Werte für die Geschwindigkeit für Pitch, Roll, Yaw, sowie für nach oben/unten.

### 3.3.5 Fuzzy Controller

Der Fuzzy Controller ist in der C++ Klasse “kinect\_controller” integriert und somit kein eigenständige ROS-Node. Mithilfe der C++ Library FuzzyLite[23] wird das erstellte Fuzzy Modell implementiert. Das Modell orientiert sich stark an dem Fuzzy Modell aus der Studienarbeit “Gestensteuerung eines Flugroboters im AR-Kontext - I believe I can fly”[38], da der Sachverhalt ähnlich ist und nur lediglich kleine Anpassungen notwendig sind um es in der aktuellen Umgebung verwenden zu können.

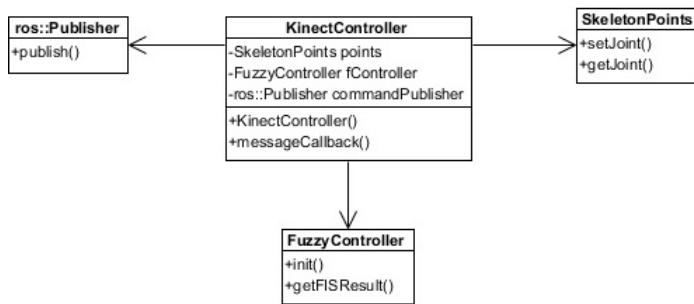


Abbildung 3.6: UML Klassendiagramm des Kinect Controllers

Es erhält die vier Eingabewerte aus den aggregierten Merkmalen und berechnet die korrespondierenden Geschwindigkeiten der vier möglichen Bewegungsrichtungen. Diese werden anschließend an die Node “drone\_controller” übergeben.

### 3.3.6 Drone Controller

Die ROS-Node “drone\_controller” ist der zentrale Punkt wenn es darum geht den Quadrocopter anzusprechen. Die Notwendigkeit die Steuerung nochmals zu wrappen besteht aus dem Grund, dass mit der regulären Steuerung zwar Kommandos von verschiedenen Stellen gesendet werden können, aber ein Chaos entsteht, da die Befehle in keiner Weise reguliert werden. Der Drone Controller übernimmt diese Aufgabe indem er die Steuerung auf eine Ebene höher abstrahiert. Er kann Steuerbefehle über die Topic “/drone\_command” von mehreren Nodes gleichzeitig empfangen und diese Nachrichten anhand eines Indexes der Priorität unterschiedlich priorisieren. Dadurch ist es zum Beispiel für das Assistenzsystem möglich die Steuerbefehle des Nutzers zu korrigieren. Weiterhin können Befehle wie Landen, Starten oder der Notfallmodus ungehindert ausgeführt werden und ungewolltes Verhalten wird vermieden. Ebenfalls ist es möglich, dass zum Beispiel ein Spotter in den Flug eingreifen kann, wenn es zu Komplikationen kommt. Dadurch wird die Sicher-

heit von Fluggerät und auch Pilot enorm verbessert. Theoretisch ist ebenso möglich durch geschicktes Priorisieren der Befehle mehr als ein Assistenzsystem gleichzeitig zu verwenden. Jedoch wird es mit zunehmender Anzahl schwieriger die richtigen Befehle auszuwählen und man sollte beachten, dass die Systeme sich nicht gegenseitig versuchen auszuhebeln.

### 3.3.7 ROS Nodes

Für den Gesamten Prozess von Nutzereingabe durch den Kinectsensor bis zur Steuerung der Drone werden eine Vielzahl von Komponenten benötigt, da einige Arbeitsschritte notwendig sind um die Daten zu verarbeiten. Zunächst wird die Microsoft Kinect durch die ROS-Node “openni\_launch” angesteuert und die Bilder der verschiedenen Kameras eingelesen. Anschließend werden diese Bilder durch die Node “openni\_tracker” analysiert und es wird, sofern ein Nutzer erkannt wurde ein Motion-Tracking erstellt. Dadurch werden bestimmte Features, in diesem Fall konkrete Körperteile erkannt. Diese werden als Nachricht unter der Topic “/tf” publiziert. Der kinect\_controller ist auf die Topic registriert und empfängt die gesendeten Nachrichten. Anhand der Merkmalspunkte werden vier Werte berechnet, die als Eingabe an die Fuzzy Logik übergeben werden. Anhand des Fuzzy Controllers ergeben sich vier Ausgangswerte, welche die Geschwindigkeiten für den Quadrocopter in den entsprechenden Richtungen, Neigung nach vorne/hinten, Neigung links/rechts, Rotation, sowie steigen/sinken angibt. Die Befehle werden an die Node “drone\_controller” übergeben wo sie nochmal verarbeitet werden. Dort ist der Ort wo auch das Assistenzsystem eingreifen kann, indem es Korrektureingaben an den Controller sendet und die Steuerbefehle des Nutzer anpasst oder gar verwirft. Schließlich wird der Befehl an den Treiber in der Node “ardrone\_autonomy” übergeben, auf Hardwareebene übersetzt und an den Quadrocopter. Für den Fall das die Drohne nur simuliert wird, sendet der wird der Befehl des “drone\_controller” direkt

an die Simulationsumgebung und somit an den virtuellen Treiber gesendet. In der untenstehenden Abbildung wird die Kooperation der unterschiedlichen Nodes visualisiert.

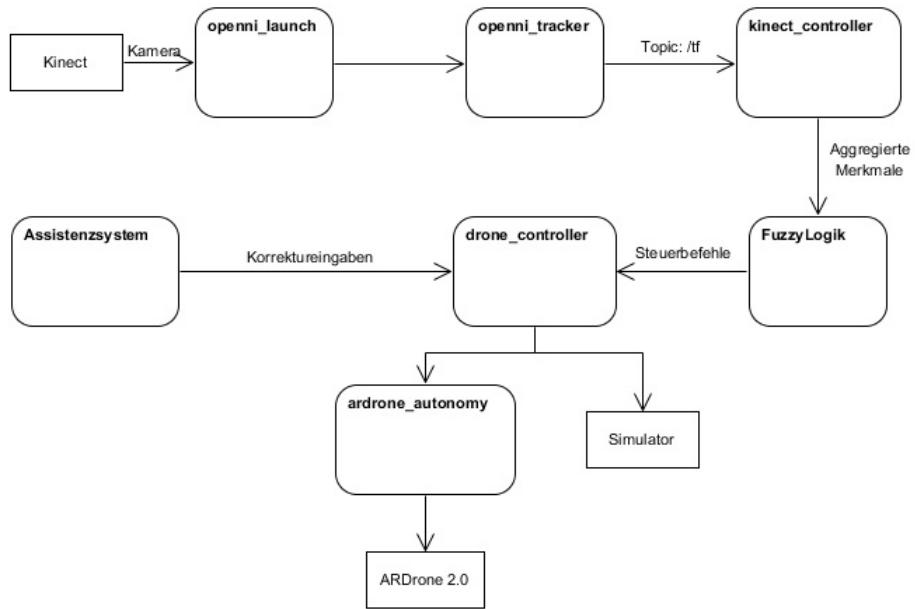


Abbildung 3.7: Zusammenspiel der verschiedenen ROS-Nodes

## 3.4 Assistenzsystem

Im Kapitel 3.2 wurde der Prozess beschrieben, wie aus den monokularen Aufnahmen der Frontkamera Tiefenbilder ermittelt werden können. Aufbauend darauf soll nun erarbeitet werden, wie mit Hilfe dieser Informationen die Implementation eines grundlegenden Assistenzsystems aussehen könnte.

### 3.4.1 Problemanalyse

Wie zuvor beschrieben, ist ein einfacher Anwendungsfall das Fliegen durch Hindernisse wie offene Türen. Dabei stößt man in der Bildverarbeitung auf eine Reihe von Herausforderungen. Wie in 3.2.4 beschrieben, sind SVO und REMODE nicht optimal für den Anwendungsfall dieser Arbeit. Dabei treten Probleme vor allem bei der Analyse des Tiefenbildes auf. Dabei ist sowohl die niedrige Qualität der Tiefenbilder problematisch, als auch die hohe Zeit, welche zwischen den Bewegungen der Drohne und den Berechnungen der Tiefenpunktfolge vergehen kann.

Die Folgende Darstellung zeigt ein Beispiel für ein von REMODE berechnetes Bild.

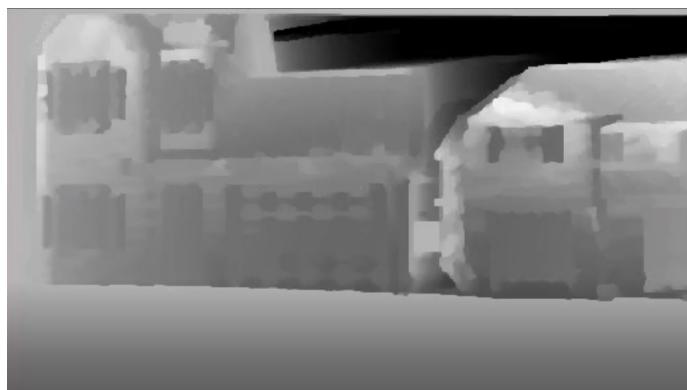


Abbildung 3.8: Approximiertes Tiefenbild durch REMODE

Dabei wird ersichtlich, dass der grundlegende Kontext für einen menschlichen Beobachter verständlich ist - in diesem Fall sind dies zwei Häuser mit einer schmalen

Lücke als Zwischenraum. Dabei sind genaue Texturen stark reduziert und teilweise verloren gegangen. Weiterhin sind Objektränder zum Teil von einem leichten hellen Schimmer umgeben, welcher in der Bildanalyse zu Problemen führen kann.

### 3.4.2 Lösungsansätze

Um an diesem Punkt ein Objekt wie eine Tür, bzw. den Zwischenraum zu erkennen, gibt es verschiedene Ansätze, um die Punktwolke zu analysieren.

#### 3.4.2.1 Referenzobjekt

Eine Möglichkeit ist die gezielte Suche nach definierten Abmessungen, Abständen und Formen im Bild. Dabei wird das Zielobjekt in die einzelnen Teilstücke aufgeteilt und die relative Beziehung dieser Teilstücke beschrieben. Am Beispiel einer Tür entspricht dies dem Rahmen, welcher aus zwei, zueinander parallelen, Geraden besteht, die rechtwinklig auf dem Untergrund aufliegen. Die beiden Geraden werden am Ende durch eine rechtwinklige Gerade miteinander verbunden.

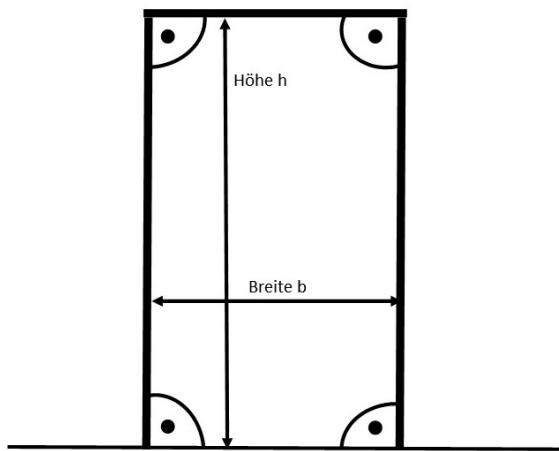


Abbildung 3.9: Objektreferenz einer Tür

Die Abbildung visualisiert die beschriebenen Anforderungen.

Wie in der vorherigen Darstellung 3.4.1 erkennbar, gibt es keine klaren Kanten und Abgrenzungen. Dies erschwert einen Vergleich mit dem Referenzobjekt zusätzlich. Abhilfe für dieses Problem schaffen in der Bildverarbeitung Filter. Filtern ist eine Technik um Bilddaten zu modifizieren, oder sie in einer Art zu verbessern. Hauptsächlich wird diese Methode dafür genutzt, um hohe Frequenzen in einem Bild zu verringern, also um das Bild zu glätten, oder um niedrige Frequenzen zu verstärken, also um Kanten hervorzuheben.

Filter basieren auf einer Berechnung in der Nachbarschaft<sup>7</sup> von Pixeln, wobei der Wert eines gegebenen Pixels im Ausgabebild bestimmt wird, in dem Algorithmen auf Pixel in der direkten Nachbarschaft dieses Datenpunktes angewandt werden. [39]

Da die Tiefenbilder verwaschen und unscharf sind, müssen niedrige Frequenzen verstärkt werden. Ein Standard, um Kanten in einem Bild hervorzuheben ist der Marr-Hildreth-Operator oder auch Laplacian of Gaussian *LoG* genannt. [5] Dieser Algorithmus führt auf den Matrizen der Punktwolken des Eingangsbildes Faltungsoperationen<sup>8</sup> durch, um ein Ausgabebild mit verdeutlichten Kanten zu erzeugen.

Dabei sucht der LoG nicht nach Kanten, sondern nach Gebieten mit rapiden Änderungen von Pixelintensitäten. Die zweite Ableitung erzeugt eine Kurve, bei der die beiden Seiten einer Kante durch einen positiven und einen negativen Wert gekennzeichnet sind. Die eigentliche Kante liegt hierbei an dem Punkt, wo der Graph Null durchquert. Die Abbildung 3.4.2.1 verdeutlicht dieses Verhalten. Die Point Cloud Library bietet dafür mit der Methode

`pcl::Edge<PointInT, PointOutT>::detectEdgeLoG`

eine Implementierung für diesen Algorithmus an. [40]

---

<sup>7</sup>Die Nachbarschaft um einen Pixel ist eine Menge an Pixeln, die sich durch ihre relative Distanz zu diesem auszeichnen. [39]

<sup>8</sup>todo; was ist faltungsoperation



Abbildung 3.10: Verlauf der zweiten Ableitung im LoG Algorithmus [5]

Nach dieser erste Schritt abgeschlossen ist, liegt ein Tiefenbild mit hervorgehobenen Kanten vor. Dies ist jedoch nicht ausreichend, um nun zuverlässig mit Hilfe eines Vergleichs mit dem Referenzobjekts z.B. Türen im Bild zu finden.

Einerseits ist das Referenzobjekt grundsätzlich ein einfaches Rechteck. Diese gibt es potentiell sehr oft im Bild, wie beispielsweise Tische, Fenster oder Bilder an Wänden. Eine geöffnete Tür zeichnet sich dabei dadurch aus, dass der Inhalt des Rahmens weiter weg ist, also in der Punktfolge tiefer ist. Im Falle von sehr genauen Tiefenbildinformationen kann man an dieser Stelle mit einer hohen Wahrscheinlichkeit durch die Veränderung der Tiefe gute Ergebnisse bei der Suche erzielen.

Wie bereits beschrieben ist das Tiefenbild jedoch sehr ungenau, da es nur durch einzelne Bilder approximiert wird. Somit ist die Verwendung eines Referenzobjektes und die Anwendung von Filtern zur Erkennung keine valide, praktisch einsetzbare Lösung.

### 3.4.2.2 Referenzpunktwolke

Der zweite Ansatz basiert auf der Verwendung einer Referenzpunktwolke. Es wird somit kein relatives Objekt definiert, sondern man vergleicht die Punktewolke des gesamten Tiefenbildes, mit einer Punktewolke für eine Tür.

Damit könnte ein besseres Ergebnis erzielt werden, da auch die Änderung der Tiefe zwischen Türrahmen und dem Bereich innerhalb des Rahmens betrachtet wird. Die Point Cloud Library bietet dafür eine Implementation an, welche als *Correspondence Grouping*, also die Gruppierung nach Übereinstimmungen bezeichnet wird.

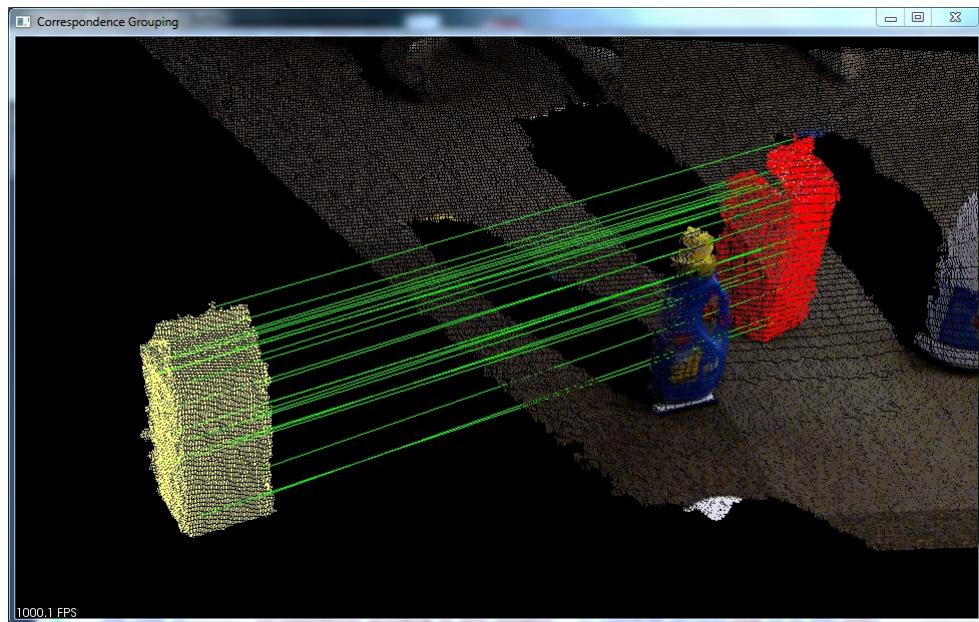


Abbildung 3.11: Correspondence Grouping mit dem PCL Algorithmus [6]

Wie im Bild erkennbar wird mit Hilfe des Algorithmus versucht, Übereinstimmungen zwischen den Features der beiden Punktewolken zu finden.

Schwierig ist hierbei die Auswahl und Generierung der Referenzpunktewolke. Es ist nicht möglich eine Lösung zu finden, die sowohl in der Simulation, als auch mit der realen Drohne funktioniert. Die Referenz ist immer nur dann sinnvoll, wenn sie

nahezu exakt die gleichen Tiefeninformationen aufweist, wie das zu vergleichende Bild.

Somit funktioniert die Auswertung auch nicht mehr, wenn eine leicht abgeänderte Tür verwendet wird, obwohl sich sowohl das Referenzobjekt, als auch das tatsächliche Objekt in der Simulation bzw. in der Realität befinden. Weiterhin lassen sich die Tiefeninformationen eines bestimmten Objekts nur sehr schwer aus dem Gesamtbild extrahieren. Um dies zu erreichen müsste bereits bekannt sein, wo sich das Objekt im Bild befindet, also ein typisches Henne-Ei Problem.

### 3.4.2.3 Alternative Lösungen

Weitere Überlegungen zur Objekterkennung sind spezielle Kennzeichnungen, die künstlich hinzugefügt werden. Denkbar sind beispielsweise farbliche Markierungen, wie ein roter Türrahmen, welcher sich farblich von der Umgebung abgrenzt und somit leicht zu finden ist. Dieser Ansatz ist grundsätzlich möglich und einfach umzusetzen, jedoch hat das Tiefenbild keine Farbinformationen.

Es ist denkbar für dieses Zweck das Ausgangsbild zu verwenden, welches noch alle Farbinformationen besitzt, jedoch zweidimensional ist. Dies würde die Anforderungen an die Arbeit verletzen, da die Analyse nicht mehr auf den Tiefeninformationen basiert.

Auch bei der Betrachtung anderer Objekte wie Wände kommt es zu schwerwiegenden Problemen. So kann der Abstand zwischen der Drohne und einer Wand in der Umgebung nicht genau genug bestimmt werden, um eine Kollision zu vermeiden, ohne unnötig in der Steuerung einzutreten.

## 4 Evaluation

### 4.1 Ergebnis

Insgesamt kam es im Verlauf der Arbeit zu einer Vielzahl von unerwarteten Schwierigkeiten und Verzögerungen. Die Anfangsphase war hauptsächlich von technischen Hardwareproblemen geprägt. Dabei war zuerst die Drohne und anschließend die Grafikkarte des Laptops defekt.

Auch softwareseitig kam es zu Verzögerungen. Die externen Projekte SVO und REMODE setzen eine sehr spezielle Umgebungskonfiguration voraus. So musste durch einen iterativen Prozess die richtige Kombination aus Betriebssystemversion, ROS Distribution und Grafikkartentreiber herausgefunden werden. Hinzu kommen eine Vielzahl von nötigen Abhängigkeiten, wie beispielsweise NVIDIA CUDA, deren Versionen wiederum untereinander kompatibel sein mussten.

Auch die Kalibrierung der Kamera und die Konfiguration der Parameter stellte eine Herausforderung dar. Der Grund dafür sind Abweichungen in der Projektumgebung. SVO und REMODE sind dafür entwickelt worden, um die Bilder einer hochauflösenden Weitbildkamera auszuwerten, welche nach unten gerichtet ist. In dieser Arbeit wurde jedoch nur eine 720p Kamera mit 92 Grad Blickfeld verwendet, welche nach vorn gerichtet ist. Weiterhin muss die Kamera gelten unterschiedliche Bedingungen mit der simulierten und der realen Drohne.

Letztendlich konnte eine passende Konfiguration der Parameter für die Simulation gefunden werden, bei der richtigen Drohne war jedoch jeglicher Versuch zur Gewin-

nung von Tiefenbildern gescheitert.

Die Ausarbeitungen zu dem Assistenzsystem haben zu dem Ergebnis geführt, dass im Rahmen dieser Arbeit eine Implementierung unter den gegebenen Anforderungen nicht möglich ist. Hauptsächlich schuld ist dafür die mangelnde Performance von SVO und REMODE in der Projektumgebung, wodurch Tiefeninformationen nur langsam, ungenau und instabil gewonnen werden können. Da die Entwicklung eines Assistenzsystems auf Objekterkennung basiert, müssen aus den verschwommenen Tiefenbildern möglichst genau Objekte wie z.B. Türen erkannt werden. Die im Rahmen dieser Arbeit betrachteten Herangehensweisen für diese Problematik haben sich alle als nicht valide herausgestellt. Im Speziellen gibt es keine verwendbaren Algorithmen in der Point Cloud Library, wodurch eine Realisierung mit enorm hohem Eigenaufwand verbunden wäre. Auf Grund der vielen Verzögerungen und dem begrenzten Bearbeitungszeitraum der Studienarbeit, war dies nicht mehr möglich. Die Anforderung, die Gestensteuerung der Drohne von C# unter Windows auf C++ unter Ubuntu zu migrieren konnte jedoch erfolgreich umgesetzt werden.

## 4.2 Ausblick

### 4.2.1 Andere Simulatoren

Die Simulationsumgebung Gazebo ist nicht die einzige verfügbare zur Simulation von Quadrocoptern. Jedoch bringt sie durch die einfache Anbindung von ROS einiges an Vorteilen mit sich. Damit gehen allerdings auch Nachteile einher. So sind die Kameraeingaben nicht sehr realistisch und Szenarien die in der Simulation funktionieren müssen in der Realität nicht funktionieren. Ebenso ist das Flugverhalten in manchen Situationen nicht realitätsgerecht und kann zu verfälschten Ergebnissen führen. Um diesem Vorzubeugen ist es ratsam die Resultate mit anderen Simulationen vergleichen. Eine aktuelle Simulationsumgebung zur Simulation von Quadrocoptern ist

der AirSim von Microsoft.[7] Ursprünglich entwickelt um Trainingsdaten zum maschinellem Lernen sammeln, kann er ebenfalls auch für herkömmliche Simulation verwendet werden. Aktuell ist allerdings nur für Windows Betriebssysteme verfügbar. [41] Er bietet eine fotorealistische Umgebung und ein akkurate Flugverhalten, dadurch ist besonders für Demos und Showcases besser geeignet.

Es existieren weiterhin andere Simulatoren, allerdings sind die meisten spielerisch veranlagt und bieten keine programmatische Schnittstelle, weshalb sie für den Zweck des Anwendungsfalles nicht sinnvoll verwendet werden können.

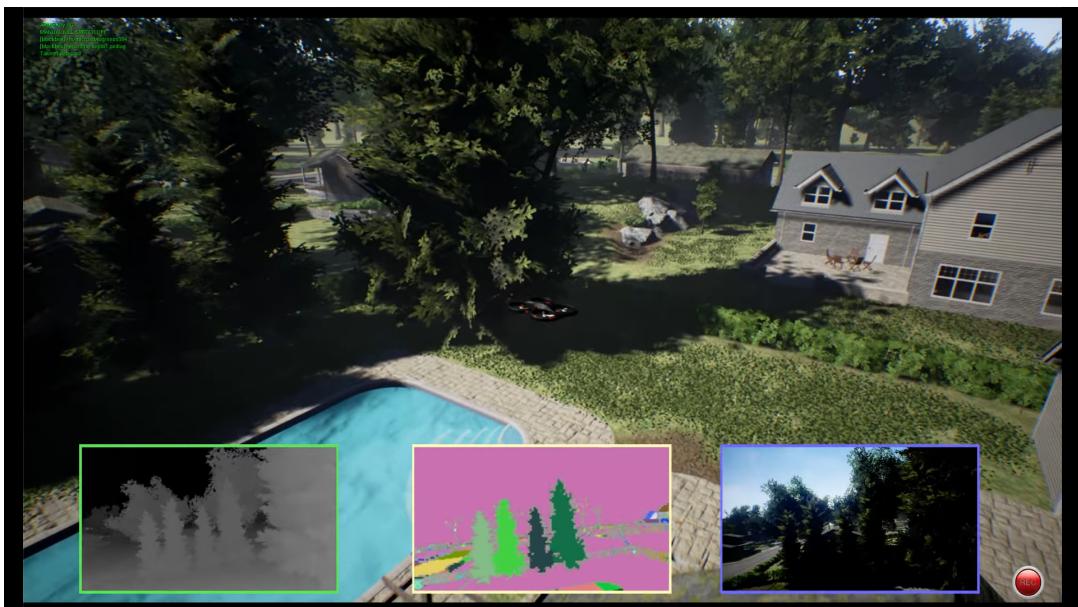


Abbildung 4.1: AirSim von Mircosoft in Aktion[7]

#### 4.2.2 Assistenzsystem

Wie bereits beschrieben, konnte im Rahmen dieser Arbeit keine Lösung für ein Assistenzsystem gefunden werden. Die Bearbeitung hat jedoch diverse Erkenntnisse hervorgebracht, welche für weitere Arbeiten relevant sein können.

Eines der Hauptprobleme besteht in der Verwendung von den externen Projekten SVO und REMODE. Diese sind für eine bessere Kamera ausgerichtet, welche vor allem nach unten gerichtet ist. Es hat sich herausgestellt, dass die Software nicht funktioniert, sobald die Drohne sich um die eigene Achse dreht, oder nach vorne bzw. hinten fliegt. Um dieses Problem zu beheben, könnte entweder der Code angepasst werden, oder eine eigene Implementierung vorgenommen werden.

Weiterhin könnte es sinnvoll sein sich genauer mit den Algorithmen der Point Cloud Library zu beschäftigen. Hierbei ist hauptsächlich die Objekterkennung in den Tiefenbildern problematisch.

Insgesamt ist jedoch die Grundvoraussetzung für ein funktionierendes Assistenzsystem, dass die darauf basierenden Tiefeninformationen sowohl aktuell, als auch möglichst exakt sind. Andernfalls ist das damit verbundene Fehlerpotential zu hoch, wodurch die Steuerung der Drohne negativ beeinflusst wird.

## Literaturverzeichnis

- [1] "Parrot AR.Drone." [https://www.parrot.com/fr/sites/default/files/styles/product\\_teaser\\_hightlight/public/parrot\\_ar\\_drone\\_gps\\_edition.png?itok=0shlzcXW](https://www.parrot.com/fr/sites/default/files/styles/product_teaser_hightlight/public/parrot_ar_drone_gps_edition.png?itok=0shlzcXW), 2017. visited: 01.22.2017.
- [2] "PointCloud visualisiert." [https://student.myvectorworks.net/public\\_newsletters/2016/07/6/StudenPortal-eNL\\_Detail\\_content2.html?utm\\_campaign=newsletteredu&utm\\_source=image&utm\\_medium=portal&utm\\_content=2](https://student.myvectorworks.net/public_newsletters/2016/07/6/StudenPortal-eNL_Detail_content2.html?utm_campaign=newsletteredu&utm_source=image&utm_medium=portal&utm_content=2), July 2016. visited: 03.05.2017.
- [3] "Lochkamera Prinzip." [https://en.wikipedia.org/wiki/Pinhole\\_camera\\_model#/media/File:Pinhole-camera.svg](https://en.wikipedia.org/wiki/Pinhole_camera_model#/media/File:Pinhole-camera.svg), 2017. visited: 03.22.2017.
- [4] "Lochkamera Prinzip." [https://github.com/uzh-rpg/rpg\\_open\\_remode/wiki/Run-Using-SVO](https://github.com/uzh-rpg/rpg_open_remode/wiki/Run-Using-SVO), Dec. 2015. visited: 02.15.2017.
- [5] "Marr-Hildreth-Operator." <https://de.wikipedia.org/wiki/Marr-Hildreth-Operator>, Mar. 2017. visited: 03.07.2017.
- [6] "PCL Correspondance Grouping." [http://pointclouds.org/documentation/tutorials/correspondence\\_grouping.php#correspondence-grouping](http://pointclouds.org/documentation/tutorials/correspondence_grouping.php#correspondence-grouping), 2017. visited: 03.05.2017.
- [7] Shital Shah and Debadeeptha Dey and Chris Lovett and Ashish Kapoor,

- "Microsoft AirSim." <https://github.com/Microsoft/AirSim>, 2017. visited: 03.16.2017.
- [8] "Parrot AR.Drone." [https://de.wikipedia.org/wiki/Parrot\\_AR.Drone](https://de.wikipedia.org/wiki/Parrot_AR.Drone), 2016. visited: 01.22.2017.
- [9] "Realtime code in ROS." <http://www.willowgarage.com/blog/2009/06/10/orocos-rtt-and-ros-integrated>, 2016. visited: 01.22.2017.
- [10] L. Joseph, *ROS Robotics Projects*. Packt Publishing, March 2017.
- [11] "ROS Einführung." <http://wiki.ros.org/ROS/Introduction>, 2017. visited: 04.11.2017.
- [12] "ROS Konzepte." <http://wiki.ros.org/ROS/Concepts>, 2017. visited: 05.01.2017.
- [13] "Single Responsibility Principle." <http://clean-code-developer.de/die-grade/orangener-grad/>, 2017. visited: 05.01.2017.
- [14] "Single Responsibility Principle." <http://wiki.ros.org/Client%20Libraries>, 2017. visited: 05.01.2017.
- [15] "Message Broker." [http://www.enterpriseintegrationpatterns.com/ramblings/03\\_hubandspoke.html](http://www.enterpriseintegrationpatterns.com/ramblings/03_hubandspoke.html), 2017. visited: 04.11.2017.
- [16] "Eventbus." <http://timnew.me/blog/2014/12/06/typical-eventbus-design-patterns/>, 2017. visited: 04.11.2017.
- [17] "TUM Simulator." [http://wiki.ros.org/tum\\_simulator](http://wiki.ros.org/tum_simulator), note = visited: 03.22.2017, 2017.

- [18] "Programmstruktur mit realer Drohne." [http://wiki.ros.org/tum\\_simulator?action=AttachFile&do=get&target=real\\_structure.png](http://wiki.ros.org/tum_simulator?action=AttachFile&do=get&target=real_structure.png), 2017. visited: 03.22.2017.
- [19] "Programmstruktur mit Simulation." [http://wiki.ros.org/tum\\_simulator?action=AttachFile&do=get&target=sim\\_structure.png](http://wiki.ros.org/tum_simulator?action=AttachFile&do=get&target=sim_structure.png), 2017. visited: 03.22.2017.
- [20] "Fuzzylogik." <http://www.spektrum.de/magazin/prinzipien-der-fuzzy-logic/820699>, 2017. visited: 04.11.2017.
- [21] L. Zadeh, *Fuzzy Sets and Systems*. 1965.
- [22] D. S. Keller, "Wissensbasierte systeme." Vorlesung, 2017.
- [23] J. Rada-Vilela, "fuzzylite: a fuzzy logic control library," 2017.
- [24] "Microsoft Kinect Golem." <https://www.golem.de/1008/77265.html>, note = visited: 05.04.2017, 2017.
- [25] "Microsoft Kinect ." <https://developer.microsoft.com/de-de/windows/kinect>, note = visited: 05.04.2017, 2017.
- [26] "Microsoft Kinect SDK." <https://developer.microsoft.com/de-de/windows/kinect/develop>, note = visited: 05.04.2017, 2017.
- [27] "Freenect." <https://github.com/OpenKinect/libfreenect>, 2017. visited: 04.08.2017.
- [28] "OpenNI." <https://github.com/OpenNI/OpenNI>, 2017. visited: 04.08.2017.
- [29] "Licensing of OpenNI."
- [30] "Definition PointCloud." <http://whatis.techtarget.com/definition/point-cloud>, Oct. 2016. visited: 03.05.2017.

- [31] "Reprojektionsfehler." [https://en.wikipedia.org/wiki/Reprojection\\_error](https://en.wikipedia.org/wiki/Reprojection_error), 2017. visited: 04.02.2017.
- [32] "Odometrie." <https://en.wikipedia.org/wiki/Odometry>, 2017. visited: 04.02.2017.
- [33] Robotics and Perception Group, "Camera Calibration." [https://github.com/uzh-rpg/rpg\\_svo/wiki/Camera-Calibration](https://github.com/uzh-rpg/rpg_svo/wiki/Camera-Calibration), 2014. visited: 02.19.2017.
- [34] "Perfomance Dokumentation SVO." [https://github.com/uzh-rpg/rpg\\_svo/wiki/Obtaining-Best-Performance](https://github.com/uzh-rpg/rpg_svo/wiki/Obtaining-Best-Performance), June 2014. visited: 02.18.2017.
- [35] "OpenNI Launch." [http://wiki.ros.org/openni\\_launch](http://wiki.ros.org/openni_launch), 2017. visited: 04.08.2017.
- [36] "OpenNI Tracker." [http://wiki.ros.org/openni\\_tracker](http://wiki.ros.org/openni_tracker), 2017. visited: 04.08.2017.
- [37] "Nachrichten Typ TFMessage." [http://docs.ros.org/jade/api/tf2\\_msgs/html/msg/TFMessage.html](http://docs.ros.org/jade/api/tf2_msgs/html/msg/TFMessage.html), 2017. visited: 04.11.2017.
- [38] N. B. D. V. M. von Bergen, *Gestensteuerung eines Flugroboters im AR-Kontext - I believe I can fly.* 2015.
- [39] "Filter in der Bildverarbeitung." <https://de.mathworks.com/help/images/what-is-image-filtering-in-the-spatial-domain.html>, 2017. visited: 03.06.2017.
- [40] "Dokumentation PCL." [http://docs.pointclouds.org/trunk/classpcl\\_1\\_1\\_edge.html](http://docs.pointclouds.org/trunk/classpcl_1_1_edge.html), Mar. 2017. visited: 03.29.2017.
- [41] S. Shah, D. Dey, C. Lovett, and A. Kapoor, "Aerial Informatics and Robotics platform," Tech. Rep. MSR-TR-2017-9, Microsoft Research, 2017.