

Masterthesis im Fach Human Centered Computing zur
Erlangung des Grades Master of Science (M.Sc.)

Entwicklung und Simulation eines Long-Term visual SLAM Systems

Robin Connor Schramm
Matrikel-Nr.: 761392

Erstprüfer: Prof. Dr.-Ing. Cristóbal Curio

Zweitprüfer: Prof. Dr.-Ing. Benjamin Himpel

Abgabedatum: 20.9.2021

Hochschule Reutlingen, Fakultät Informatik



Hochschule Reutlingen
Reutlingen University

Abstract:

Simultaneous localization and mapping (SLAM) is a key technology needed for autonomous mobile robot navigation. The increased usage of cameras and focus of deployment of robots in non-stationary environments means, that visual SLAM (vSLAM) and long-term SLAM (LT-SLAM) play an increasingly important role in the SLAM community. There are solutions to the vSLAM problem in non-stationary environments, but they can not be considered robust and broadly applicable. Furthermore, it is especially difficult to test robot applications in non-stationary environments since those types of environments usually have either dynamic objects in them or change in appearance over time, which makes experiments in those environments non-repeatable. In this paper, a approach to long-term visual SLAM systems using 3D-segmentation, 3D features and a kd-tree search will be presented. In addition, a new Simulation for non-stationary environments based on Unity and ROS for evaluation of LT-SLAM systems is introduced. It has been shown, that it is possible to apply supervoxel segmentation and viewpoint feature histograms in combination to detect objects over multiple frames in pointclouds generated from rgb-d cameras.

Zusammenfassung:

Simultaneous localization and mapping (SLAM) ist eine der Schlüsseltechnologien, um die Navigation autonomer mobiler Roboter zu ermöglichen. Der vermehrte Nutzen von Kameras und der Einsatz von Robotern in nicht-stationären Umgebungen hat zur Folge, dass visual SLAM (vSLAM) und Long-Term SLAM (LT-SLAM) eine immer wichtigere Rolle in der SLAM Community spielen. Es existieren bereits Lösungen für das vSLAM Problem in nicht-stationären Umgebungen, diese gelten jedoch noch nicht als robust und können bisher nur in bestimmten Umgebungen verwendet werden. Zusätzlich ist es herausfordernd, Roboterapplikationen in nicht-stationären Umgebungen zu evaluieren, da diese Art von Umgebungen dynamische Objekte beinhalten oder sich über die Zeit verändern. Dadurch sind Experimente in nicht-stationären Umgebungen nicht wiederholbar. In dieser Arbeit wird ein Ansatz vorgestellt, der anhand 3D-Segmentierung, 3D Features in Kombination mit einem kd-tree die Basis für ein long-term visual SLAM System dienen kann. Außerdem wurde eine neue Robotersimulation auf Basis von ROS und Unity zur Entwicklung und Evaluierung von LT-SLAM Systemen vorgestellt. Es wurde gezeigt, dass die Kombination aus Supervoxel Segmentierung und Viewpoint Feature Histograms verwendet werden können, um Objekte über mehrere Frames in Punktwolken aus RGB-D Kameras zu erkennen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Codeausschnittverzeichnis

Abkürzungsverzeichnis

1 Einleitung	1
1.1 Motivation und Ziele	2
1.2 Methodologie	2
2 Grundlagen	3
2.1 Mobile Robotik	3
2.1.1 Sensorik	4
2.1.2 Technische Spezifikationen	7
2.2 Simultaneous localization and mapping	9
2.2.1 Visual simultaneous localization and mapping	10
2.2.2 Nicht-stationäre Umgebungen	11
2.2.3 Long-Term simultaneous localization and mapping	12
2.3 Robot Operating System	14
2.3.1 Roslaunch	15
2.3.2 ROS-Bags	15
2.4 Robotersimulation	16
2.4.1 Gazebo	18
2.4.2 Unity	19
2.4.3 Koordinatensysteme	20
2.5 Segmentierung von Punktwolken	21
2.6 3D Feature Matching	23
3 State of the Art	24
3.1 Simultaneous localization and mapping	24
3.1.1 Datensätze	24
3.1.2 Bestehende Systeme	26
3.2 Verfahren zur Segmentierung von 3D-Punktwolken	31
3.2.1 Methoden	31

4 Methodik	34
4.1 Simulation nicht-stationärer Umgebungen für mobile Roboter	34
4.1.1 Auswahl der Simulationsumgebung	34
4.1.2 Nutzwertanalyse	37
4.1.3 Aufbau der Unity-Simulation	45
4.1.4 Evaluierung der Simulation	54
4.1.5 Limitationen	57
4.2 Erkennung semi-statischer Objekte	58
4.2.1 Aufbau des Projekts	58
4.2.2 Auswahl des Segmentierungsalgorithmus	58
4.2.3 Parameterauswahl	59
4.2.4 Parameter Evaluation	61
4.2.5 Detektieren und Extrahieren von 3D-Features	65
4.3 Diskussion der Ergebnisse	68
5 Fazit	70
5.1 Zusammenfassung	70
5.2 Ausblick	71
Literatur	72
Anhang	78

Abbildungsverzeichnis

2.1	Darstellung der Daten eines 2D und 3D Laserscanners. Oben: 3D-Punktwolke eines Velodyne 3D-Laserscanners aus dem KITTI Datensatz. Links Unten: Darstellung einer Szene aus dem Gazebo Simulator. Rechts Unten: 2D-Punkte eines 2D-Laserscanners innerhalb der Simulierten Umgebung aus dem linken Bild.	6
2.2	Darstellung der Daten einer Intel RealSense D435 Tiefenkamera. Links oben sind die Daten der Tiefenkamera zu sehen, links unten die Daten der Farbkamera. Auf der rechten Seite sind die Farb- und Tiefendaten kombiniert in Form einer farbigen Punktwolke zu sehen.	8
2.3	Arbeitsablauf bei der Entwicklung von Roboterapplikationen. Informationen von Unity's Unite Now ¹	17
2.4	Darstellung der Daten von einer Farbkamera (links oben), einer Tiefenkamera (links unten) sowie einer farbigen Punktwolke (rechts). Die Umgebung wurde in Unity simuliert und in Rviz visualisiert.	21
2.5	Zuteilung und Richtung der Achsen in den 3D-Koordinatensystem in Unity und Robot Operating System (ROS), angepasst aus der ROS# Dokumentation ²	22
3.1	Ein Ausschnitt aus dem TUM RGB-D SLAM Datensatz aus der Sequenz <i>rgbd_dataset_freiburg2_large_with_loop</i> . Links: Bild der Farbkamera. Rechts: Bild der Tiefenkamera.	25
3.2	Ausschnitte aus den OpenLORIS-Scene Datasets. Links oben: Bild der Farbkamera. Links unten: Bild der Tiefenkamera. Rechts: Punkte des 2D-Laserscanners.	26
3.3	Ausschnitt aus der Object Segmentation Database.	27
4.1	Ausschnitt aus der AirSim Unity Simulation für PKWs.	39
4.2	Ausschnitt aus dem Nvidia Isaac Sim aus der Szene <i>small_Warehouse</i>	40
4.3	Ausschnitt aus der ROS# Demo Szene in Unity. Bei dem simulierten Roboter handelt es sich um den Turtlebot3.	42
4.4	Ausschnitt aus einer Szene mit dem Unity Robotics Hub. Bei der Umgebung handelt es sich um eine abgeänderte Version des <i>Big_Warehouse</i> von ISAAC. Bei dem Roboter handelt es sich um die Basis des Care-O-bot 4.	43

4.5	Einstellungen für die Verbindung von Unity und ROS.	45
4.6	Fenster zur Generierung von Klassen von ROS Messages in Unity. .	46
4.7	Objekte, die in der Simulation zu finden sind. Von oben Links: Regale mit Kisten, eine Säule mit Feuerlöscher und Mülltonne, ein Gabelstapler und Paletten.	47
4.8	Die in der Simulation verwendete Care-O-bot 4 Base mit zwei Kameras.	48
4.9	Struktur der Unity Simulation und deren Kommunikation zu ROS. .	49
4.10	Semi-statische Objekte in der Unity Simulation. Links: Volle Regale, Mitte: Regale mit zufälligen fehlenden Kisten, Rechts: Leere Regale.	50
4.11	UML Diagramm der Kameras.	51
4.12	Demo-Szene mit einem Tiefenshader.	53
4.13	Der Tiefenshader in der Simulationsumgebung.	53
4.14	Aufbau der Unity-Szene.	56
4.15	Ausschnitt aus dem Unity Profiler bei laufender Simulation.	56
4.16	Die für die Parameterauswahl verwendete Referenz-Szene. Links ist die Sicht der Farbkamera in der Simulation, rechts die zugehörige Punktwolke.	61
4.17	Punktwolke mit Ground Truth Labeln.	62
4.18	Das VFH von einem segmentierten Objekt aus einer Punktwolke. .	66
4.19	Die für die Objekterkennung verwendete segmentierte Punktwolke.	67
4.20	Die sechs Segmente mit den niedrigsten Distanzen zur Ground Truth.	69
A1	Ausschnitt des TF-Tree des care-o-bot 4 in der Unity Simulation. .	83

Tabellenverzeichnis

4.1	Paarweiser Vergleich der Kriterien, die eine Simulation erfüllen sollte.	37
4.2	Die für die Simulation durchgeführte Nutzwertanalyse.	37
4.3	Parameter, deren Datentyp und der getestete Wertebereich für die Supervoxel Segmentierung.	61
4.4	Parameter, und deren ermittelter bester Wert.	64
4.5	Werte der Boolean-Parameter, die die besten Segmentierungen erreichten.	64
A1	K-d tree Distanzen durch VFH Matching.	84

Codeausschnittverzeichnis

1	ROS Launch Datei um den TCP Server für die Unity Simulation zu starten.	78
2	Lesen der Bilddaten von Unity aus der Grafikkarte und das Senden an ROS.	79
3	Tiefenshader für die Unity Simulation.	80
4	Beispiel einer CameraInfo Message der simulierten Farbkamera in ROS.	81
5	Funktionen, die für die Ermittlung und den Vergleich der VFH verwendet wurden.	82

Abkürzungsverzeichnis

2D Zweidimensionalität, zweidimensional

3D Dreidimensionalität, dreidimensional

BA Bundle Adjustment

BOWW Bag-of-visual-words

BOW Bag-of-words

BRIEF Binary Robust Independent Elementary Features

DGPS Differential Global Positioning System

FPS Frames pro Sekunde

GiB Gibibyte, ein GiB entspricht 2^{30} Bytes.

GLONASS Global'naya Navigatsionnaya Sputnikovaya Sistema, Globales Satellitennavigationssystem der Russischen Föderation

GNSS Global Navigation Satellite System, Globales Navigationssatellitensystem

GPS Globales Positionsbestimmungssystem

Hz Hertz, Einheit zur Angabe der Anzahl der Vorgänge pro Sekunde

IFR International Federation of Robotics

IMU Inertiale Messeinheit

kd-tree k-dimensional tree, ein Baum mit k Dimensionen.

KI Künstliche Intelligenz

LIDAR Light detection and ranging

LT-SLAM Long-Term simultaneous localisation and mapping

NARF Normal Aligned Radial Feature

ORB Oriented FAST and Rotated BRIEF

PCL Point Cloud Library

RANSAC RANdom SAmple Consensus

RGB-D RGB-Farbbild mit zugehörigen Tiefendaten

RIFT Rotation-Invariant Feature Transform

ROS Robot Operating System

RTK Real-time kinematic positioning

SDF Simulation Description Format

SIFT Scale Invariant Feature Transform

SLAM Simultaneous localisation and mapping

STM Short-Term Memory Modul

SURF Speeded up robust Features (SURF)

UI User Interface

URDF Universal Robot Description Format

UX User Experience

VFH Viewpoint Feature Histogram

vSLAM Visual simultaneous localisation and mapping

XML Extensible Markup Language

1 Einleitung

Mobile Roboter erlangten durch ihr breites Anwendungsfeld große Popularität in den letzten Jahren. So sind nach dem World Robotics Report 2020¹ der International Federation of Robotics (IFR) die Aktien von industriellen Robotern auf dem höchsten Punkt in der Geschichte. Deutschland ist nach Stand 2020 mit 221.500 industriellen Robotern das Land mit dem größten Roboterbestand in Europa und dem fünft größtem Bestand weltweit. Dabei ist der Logistiksektor der vorherrschende Bereich, in dem mobile Roboter eingesetzt werden. Damit sich mobile Roboter effizient und ohne menschlichen Eingriff fortbewegen können, wurden verschiedene Technologien zur Roboternavigation entwickelt. So ist das simultaneous localisation and mapping (SLAM) Verfahren eine der Schlüsseltechnologien, um autonome mobile Roboter zu ermöglichen [SK16]. Die meisten SLAM-Systeme sind dabei jedoch auf statische Umgebungen mit beschränkter Größe limitiert. Long-Term SLAM (LT-SLAM) Verfahren gehen das Problem an, mobile Roboter über längere Zeit in sich verändernden Umgebungen autonom einzusetzen. Dies ist jedoch ein herausforderndes Unterfangen, weswegen zwar bereits einige vielversprechende LT-SLAM Systeme existieren, diese jedoch noch nicht robust in einer Vielzahl von Umgebungen einsetzbar sind. Eine weitere SLAM-Variante namens visual SLAM (vSLAM), bei der Kameras als primärer Sensor dienen, wird derzeit in vielen Gebieten genutzt und gewinnt weiter an Beliebtheit. Durch eine höhere Datenvielfalt und geringere Kosten im Vergleich mit herkömmlichen SLAM-Systemen mit Laserscannern, wird vSLAM in Zukunft vermutlich eine essentielle Rolle für mobile Roboter spielen [SK16]. Des Weiteren spielen Simulationen zur Entwicklung von Roboterapplikationen durch die steigende Komplexität der Software und den Umgebungen, in denen Roboter eingesetzt werden sollen, eine große Rolle. Gerade zur Evaluierung von LT-SLAM-Systemen innerhalb von nicht-stationären Umgebungen bieten Simulationen großes Potential, welches bisher noch nicht voll ausgeschöpft ist.

Jede Website, die in dieser Arbeit referenziert wird, wurde zuletzt am 19.09.2021 aufgerufen.

¹<https://ifr.org/ifr-press-releases/news/record-2.7-million-robots-work-in-factories-around-the-globe>

1.1 Motivation und Ziele

Am Fraunhofer-Institut für Produktionstechnik und Automatisierung IPA in der Gruppe für Servicerobotik für Industrie und Gewerbe werden Verfahren zur autonomen Navigation entwickelt. Dabei werden unter anderem Systeme für den Einsatz in nicht-stationären Umgebungen und für LT-SLAM betrachtet. In dieser Arbeit wird eine neuartige Herangehensweise für das LT-SLAM Problem vorgeschlagen. Dabei werden verschiedene Verfahren zur Segmentierung von 3D Punkt wolken untersucht und prototypisch umgesetzt. Die durch die Segmentierung entstehenden Regionen sollen durch 3D-Feature Deskriptoren und Suchalgorithmen über längere Zeit verfolgt werden können. Da bisher keine Simulationen existieren, die darauf ausgelegt sind, LT-SLAM Systeme zu testen, wird im Zuge dieser Arbeit eine Simulationsumgebung für nicht-stationäre Umgebungen und LT-SLAM Systeme erstellt und evaluiert.

1.2 Methodologie

Zunächst werden in Kapitel 2 die für die Arbeit relevanten Begriffe, Methoden und Grundlagen beschrieben. Dabei wird der Fokus auf das Feld der mobilen Robotik und die dort eingesetzte Software zur Navigation und Kartierung gesetzt. Des Weiteren werden hier die Grundlagen der Robotersimulation und Methoden der Bildverarbeitung wie die Segmentierung erläutert.

Anschließend wird in Kapitel 3 der State of the Art für SLAM-Systeme und für die Segmentierung von 3D-Punkt wolken vorgestellt. Dabei wird sowohl auf die existierenden Verfahren, als auch auf die Datensätze zur Evaluierung dieser Verfahren eingegangen.

In Kapitel 4 wird die Methodik des praktischen Teils der Arbeit vorgestellt. So wird zunächst die Auswahl und die Entwicklung der in den Zielen beschriebenen Simulation erklärt. Daraufhin wird der Fokus auf die Erkennung der semi-statischen Objekte durch Segmentierung und 3D-Features gelegt. Zum Schluss werden die entwickelten Verfahren in diesem Kapitel ausgewertet und die entstandenen Ergebnisse diskutiert.

Im Fazit in Kapitel 5 wird schließlich eine Zusammenfassung über die Arbeit sowie ein Ausblick für weitere mögliche Vorgehen gegeben.

2 Grundlagen

Im folgenden werden die für dieses Dokument wichtigsten Grundlagen erläutert. Die Robotik ist ein breites und interdisziplinäres Feld, daher wird der Fokus auf die Grundlagen der mobilen Roboternavigation sowie der Bildverarbeitung gelegt.

2.1 Mobile Robotik

Hertzberg et al. definieren in ihrem Buch [HLN12] mobile Roboter als programmierbare Maschinen, die auf Basis von Sensordaten agieren. Mobile Roboter bewegen sich innerhalb gewisser Grenzen in Umgebungen, die entweder dynamisch, nicht vollständig erfassbar oder zur Zeit der Programmierung unbekannt sind.

Die mobile Robotik ist eines der am schnellsten wachsenden Forschungsfelder [RVLA19]. Mobile Roboter werden häufig sowohl in der Wissenschaft, als auch in der Industrie eingesetzt. So bildeten sich die Felder der Dienstleistungs- und Servicerobotik, in denen Anwendungsmöglichkeiten von Industrierobotern aus der Automatisierungstechnik in neue Anwendungsgebiete übertragen wurden [HLN12]. Serviceroboter können sowohl im privaten, wissenschaftlichen, als auch im industriellen Umfeld eingesetzt werden und ersetzen hier oft Menschen in ihren Aufgaben und Tätigkeiten. Rubio et al. beschreiben in [RVLA19] ein breites Spektrum an Anwendungsfeldern für mobile Roboter. Dazu gehören beispielsweise industrielle Automatisierung, Konstruktion, Unterhaltung, Museumsführungen, persönlicher Service¹, Transport, Medizinische Versorgung oder Einsätze in extremen Umgebungen wie z.B. dem Mars² oder in der Tiefsee³.

Damit Roboter in verschiedenen Umgebungen autonom eingesetzt werden können, wird in der Regel eine Karte der Umgebung erstellt, in der sich der Roboter

¹<https://www.care-o-bot.de/en/care-o-bot-4.html>

²<https://mars.nasa.gov/mars2020/>

³<https://www.jpl.nasa.gov/news/robotic-navigation-tech-will-explore-the-deep-ocean>

gleichzeitig lokalisiert. Dieser Prozess wird simultaneous localisation and mapping (SLAM) genannt und wird in Kapitel 2.2 näher beschrieben. Um Mobile Roboter und SLAM Systeme realisieren zu können, werden verschiedene Arten von Sensoren eingesetzt.

2.1.1 Sensorik

Die Auswahl und Funktionsweise von Sensoren haben einen entscheidenden Einfluss darauf, in welchem Umfeld und mit welcher Effektivität ein mobiler Roboter eingesetzt werden kann.

Im folgenden wird auf Basis von [HLN12] eine Auswahl an Sensorkategorien, die in der mobilen Robotik häufig genutzt werden, vorgestellt und deren grundlegende Funktionsweise beschrieben.

- i. **Drehwinkelsteller:** Zur Messung von Drehwinkeln, beispielsweise an Rädern eines Roboters um dessen Geschwindigkeit zu bestimmen, werden Impulsgeber verwendet. An der Drehachse ist hierbei eine sich mitdrehende Scheibe befestigt. Diese Scheibe ist so aufgebaut, dass sie bei gewissen Winkeln des Rads einen entsprechenden Impuls abgibt, wodurch die Stellung des Rads ermittelt werden kann. Drehwinkelsteller werden bei Robotern oft zur Schätzung von Position und Orientierung verwendet, was Odometrie genannt wird. Die Positionsbestimmung anhand der Radodometrie allein ist jedoch sehr ungenau und fehleranfällig, weswegen häufig weitere Sensoren ergänzend eingesetzt werden.
- ii. **Beschleunigungssensoren:** Zur Schätzung der Beschleunigung eines Systems werden in der Regel inertiale Messeinheiten (IMUs) eingesetzt. Diese bestimmen ihre Geschwindigkeit und Orientierung anhand von Massenträgheit über die Zeit.
- iii. **Global Navigation Satellite Systems (GNSS):** GNSS sind eine weit verbreitete Methode zur Lokalisierung verschiedenster Objekte, darunter mobile Roboter [H WLW08]. Das in Europa bekannteste GNSS ist das globale Positionsbestimmungssystem (GPS). GPS lokalisiert einen am Roboter befestigten GPS-Empfänger, dessen Position anhand mehrerer Satelliten gemessen wird. Durch die großen Entfernung, die die Signale zurücklegen müssen, unterliegt die Positionsmeßung durch GPS Messfehlern aus mehreren Quellen, womit größere Abweichungen der Messergebnisse entstehen. Eine ebenfalls häufig verwendete Alternative zu GPS bietet das in Russland entwickelte Global'naya Navigatsionnaya Sputnikovaya Sistema (GLONASS). Um die Messfehler zu verringern und eine erheblich höhere Genauigkeit zu erzielen, kann eine zusätzliche Basisstation, deren Position exakt bekannt

ist, als Referenz verwendet werden. Dieses Verfahren zur Korrektur wird Differential Global Positioning System (DGPS) genannt. Eine besondere Art des DGPS ist das Real-time kinematic positioning (RTK) Verfahren, bei dem die Korrektur nicht anhand des Inhalts der Signale stattfindet, sondern anhand der Frequenzen der Trägersignale, wodurch Genauigkeiten im Zentimeterbereich möglich sind [TY09].

- iv. **Laserscanner:** Laserscanner setzen sich trotz ihres hohen Gewichts, Energiebedarfs und Preises in der mobilen Robotik als primärer Sensor zur Be trachtung der Umgebung durch, da sie genaue Entfernungsmessungen über große Distanzen ermöglichen. Laserscanner senden Laserstrahlen aus und messen über Lichtlaufzeitmessung oder Phasendifferenzmessung die Entfernung bis zum nächsten Hindernis. Durch diese Technik und mit dem Einsatz von Drehspiegeln wird so zweidimensionale (2D) oder dreidimensionale (3D) Abdeckung durch Laser ermöglicht. Die Methode, mit der eine Entfernung mit Hilfe eines Lasers bestimmt wird, nennt man light detection and ranging (LIDAR)⁴. Die Verwendung von LIDAR hat in den letzten beiden Jahrzehnten einen Aufschwung erlebt und wird heutzutage oft zur Kartierung, Lokalisierung und zur Objekterkennung eingesetzt [SK16] (Seite 714). Abbildung 2.1 zeigt die Darstellung durch Punkt wolken von 2D- und 3D-Laser-scannern gemessenen Daten.
- v. **Kameras:** Kameras wurden durch den steigenden Fortschritt der digitalen Bildverarbeitung und die sinkenden Kosten in der Herstellung eine der preiswertesten und meist genutzten Sensorarten für Roboter [SK16]. Kameras sind verglichen mit Laserscannern zudem leicht in Gewicht und haben einen niedrigeren Stromverbrauch [FPRARM15]. Kameras bieten zusätzlich den Vorteil, dass sie eine hohe Bandbreite an Daten erzeugen können. Kameradaten beinhalten die notwendigen Informationen, um daraus Entfernungen, sowie das Erscheinungsbild, Farben und Texturen der Umgebung zu bestimmen [FPRARM15]. All diese Informationen können von Bildverarbeitungssystemen verwendet werden, um beispielsweise Objekte in der Umgebung oder den aktuellen Standort des Roboters zu erkennen. Dabei handelt es sich jedoch um komplexe Probleme, da Bilddaten zwar reich an Informationen sind, dafür aber Computer diese nur schwer interpretieren können.
- vi. **Tiefenkameras:** Tiefeninformationen sind in der Robotik oft für die Lokalisierung notwendig und können neben Laserscannern auch von Kamerasy stemen aufgenommen werden. Sie werden in der Regel in Form von Punkt wolken oder Tiefenbildern dargestellt und verarbeitet. Die beiden gängigsten Methoden zur Akquise von Tiefendaten durch Kameras sind Stereoka-

⁴<https://oceanservice.noaa.gov/facts/lidar.html>

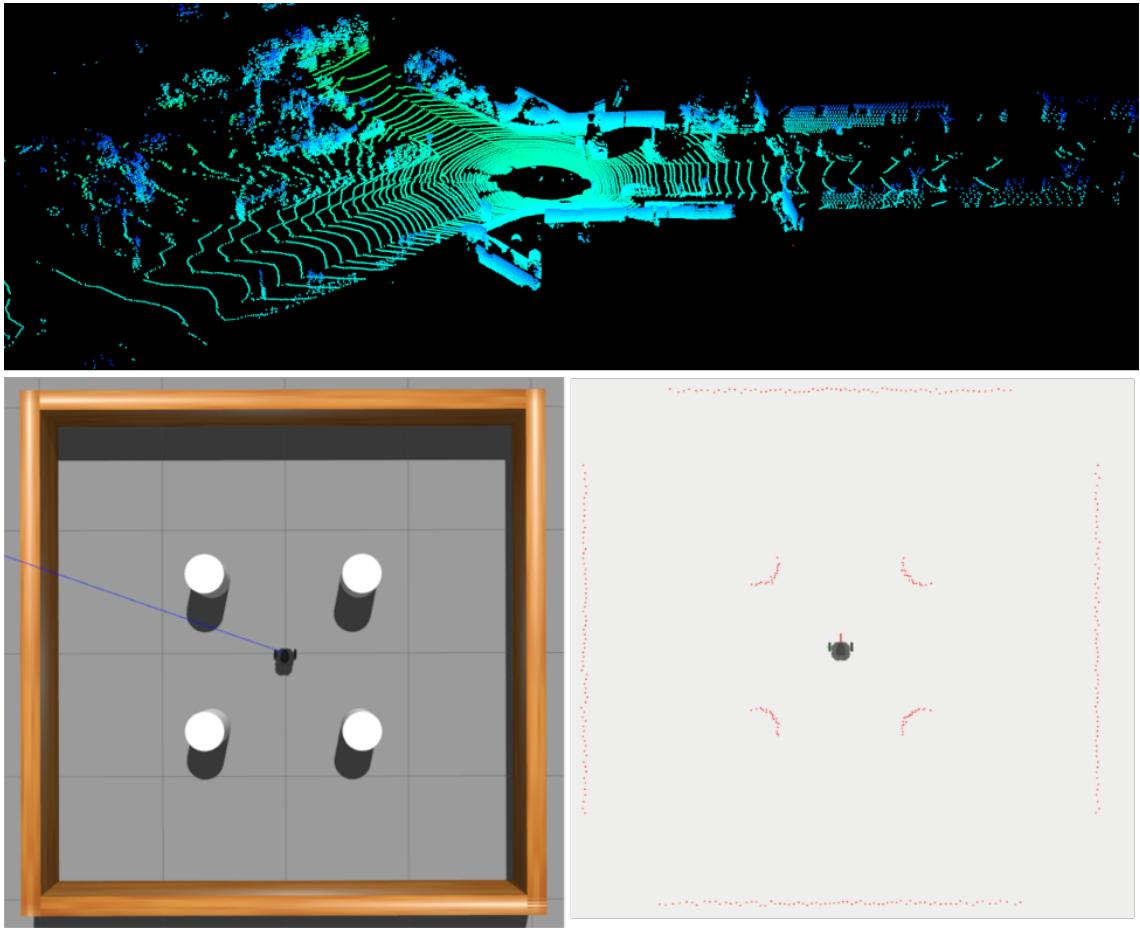


Abbildung 2.1: Darstellung der Daten eines 2D und 3D Laserscanners. Oben: 3D-Punktwolke eines Velodyne 3D-Laserscanners aus dem KITTI Datensatz. Links Unten: Darstellung einer Szene aus dem Gazebo Simulator. Rechts Unten: 2D-Punkte eines 2D-Laserscanners innerhalb der Simulierten Umgebung aus dem linken Bild.

meras und 3D-Kameras. Bei Stereokameras nehmen zwei Kameras gleichzeitig Bilder auf. Diese Bilder werden anschließend fusioniert, wodurch aus der Differenz von den jeweils zusammengehörigen Bildpunkten die Tiefe eines Punkts berechnet werden kann. Bei Stereokameras wird auch von passiven Sensoren gesprochen, da sie nur das Licht der Umgebung aufnehmen und interpretieren. 3D-Kameras, auch RGB-D Kameras genannt [MM21], verwenden Laserdioden, welche die Szene mit moduliertem Licht beleuchten. Das reflektierte Laserlicht wird detektiert und anhand der Laufzeit der Laser werden Abstandswerte ermittelt. Diese Art von Kameras werden auch als aktive Kameras bezeichnet, da sie die Umgebung selbst beleuchten. Tiefenkameras haben gegenüber 3D-Laserscannern den Vorteil, mit einer höheren Datenrate Tiefendaten zu generieren. So hat die Micro-

soft Azure Kinect⁵ eine Datenübertragungsrate von bis zu 30 Hertz (Hz) [TDCH21]. Im Vergleich dazu haben Laserscanner in der Regel eine Datenübertragungsrate von 1 Hz. Reine Tiefenkameras können auch mit Farbkameras kombiniert werden, um RGB-D Daten zu generieren. Dabei wird jedem Punkt ein Farbwert zugewiesen, woraus wiederum farbige 3D Punktwolken erstellt werden können. Außerdem sind Tiefenkameras deutlich kostengünstiger als Laserscanner. So ist die Xbox One Kinect ab 129,90€⁶ erhältlich, die Intel Realsense D435 ab 189.00\$⁷. Die vom Care-o-bot 3⁸ eingesetzten Laserscanner kosten dagegen beispielsweise 1.184,05€⁹ für den Hokuyo URG-04LX-UG01 und 2.280,00€¹⁰ für den Sick Sicherheitslaserscanner S30B-2011DA. Die Nachteile von Tiefenkameras wie der Kinect liegen dabei laut [HLN12] im geringeren Öffnungswinkel, deutlich kleinerem Messbereich und größerem Rauschen. In Abbildung 2.2 sind die Messungen einer Intel Realsense D435 Tiefenkamera in Form eines Farbbilds, eines Tiefenbilds und einer farbigen Punktwolke abgebildet.

2.1.2 Technische Spezifikationen

Zu jedem Sensor gibt es in der Regel eine technische Spezifikation, welche diverse Metriken definiert. Anhand dieser Metriken kann für jeden Einsatzzweck eines Roboters die passende Sensorik ausgewählt werden. Im folgenden werden einige Metriken aus [HLN12] aufgezählt und erläutert.

- i. **Messbereich:** Innerhalb dieses Bereichs besteht der Anspruch an den Sensor, dass die Messabweichungen innerhalb festgelegter Grenzen bleiben. So hat der SICK Laserscanner des Typs LMS1xx beispielsweise einen Messbereich von 0.5m-50m.¹¹ Der Sensor kann zwar außerhalb dieses Messbereiches funktionieren und Daten aufnehmen, die Messergebnisse sind in diesem Fall aber nicht zuverlässig.
- ii. **Auflösung:** Die Auflösung eines Sensors gibt den kleinsten messbaren Unterschied zwischen zwei Messwerten an. Dabei kann es sich je nach Sensor-

⁵<https://www.microsoft.com/en-us/d/azure-kinect-dk/8pp5vxmd9nhq>

⁶https://www.ideal.de/preisvergleich/OffersOfProduct/4404512_xbox-one-kinect-microsoft.html

⁷<https://store.intelrealsense.com/buy-intel-realsense-depth-camera-d435.html>

⁸<https://www.care-o-bot.de/en/care-o-bot-3/hardware/technical-data.html>

⁹<https://www.robotshop.com/de/de/hokuyo-urg-04lx-ug01-scan-laser-entfernungsmesser-eu.html>

¹⁰<https://de.wiautomation.com/sick/general-automation/sensors/1026822>

¹¹<https://www.sick.com/de/en/detection-and-ranging-solutions/2d-lidar-sensors/lms1xx/c/g91901>

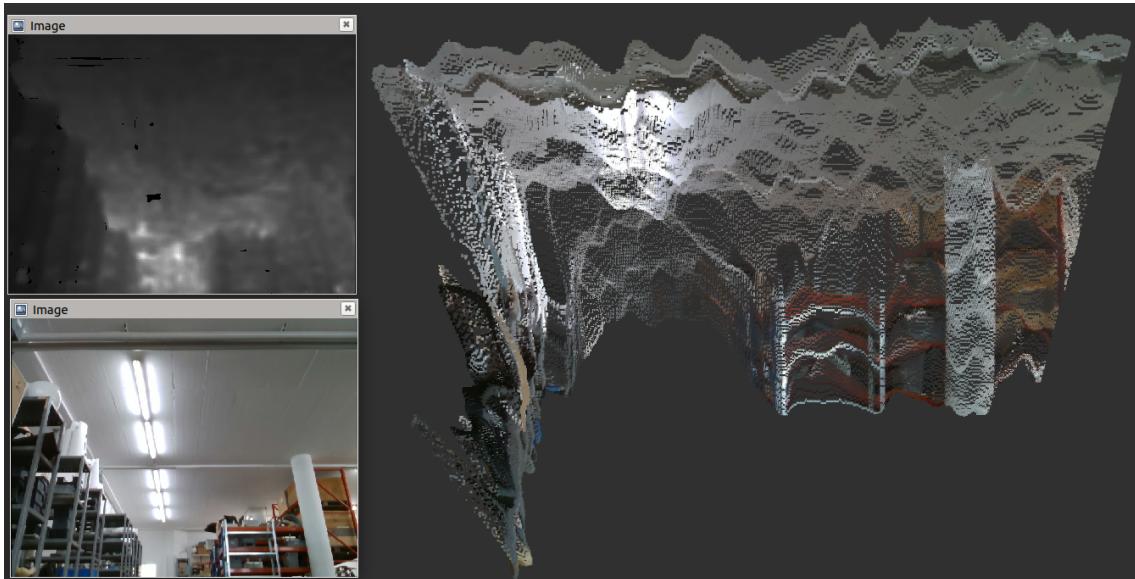


Abbildung 2.2: Darstellung der Daten einer Intel RealSense D435 Tiefenkamera. Links oben sind die Daten der Tiefenkamera zu sehen, links unten die Daten der Farbkamera. Auf der rechten Seite sind die Farb- und Tiefendaten kombiniert in Form einer farbigen Punktwolke zu sehen.

typ um beliebige physikalische Größen wie Spannung oder Entfernungen handeln.

- iii. **Messfrequenz:** Die Messfrequenz gibt die Anzahl der Messungen innerhalb eines bestimmten Zeitraums an. Die Frequenz wird üblicherweise in Hz angegeben, was die Anzahl der Vorgänge pro Sekunde beschreibt.
- iv. **Empfindlichkeit:** Die Empfindlichkeit eines Sensors beschreibt den Einfluss des Gemessenen auf den daraus entstehenden Messwert. Eine hohe Empfindlichkeit ist in der Regel erwünscht, jedoch sind empfindlichere Sensoren auch anfälliger für Störungen.
- v. **Messfehler:** Der Messfehler beschreibt die Differenz des gemessenen Wertes und des tatsächlichen Wertes. Dabei wird zwischen zwei Arten der Messfehler unterschieden. **Systematische Messfehler** beinhalten Abweichungen, die nach deterministischen Prinzipien funktionieren. Ein Beispiel dafür ist die Lufttemperatur, die die Messung von einem Ultraschallsensor beeinflusst. Diese Fehler können durch Einsatz von Software oder durch Hinzunahme weiterer Sensoren zu einem gewissen Grad kompensiert werden. Dennoch führt die Existenz von systematischen Messfehlern dazu, dass Messungen grundsätzlich nicht zu 100% die Realität widerspiegeln können. **Zufällige Messfehler** beschreiben die Unsicherheit einer Messung durch

stochastische Prozesse wie *Rauschen*. Dabei werden die Messwerte selbst bei Wiederholung der Messung unter den selben Bedingungen abweichen oder streuen. Sensordaten sind also grundsätzlich fehlerbehaftet und müssen entsprechend gehandhabt werden.

- vi. **Genauigkeit:** Die Genauigkeit beschreibt den relativen Messfehler, also den prozentualen Wert einer Abweichung bezüglich des realen Werts.
- vii. **Weitere Daten:** Weitere relevante Eigenschaften eines Sensors zur Konzeption eines Robotersystems beinhalten Größe, Gewicht, Spannungsbedarf, Energiebedarf sowie die Kosten des Sensors.

2.2 Simultaneous localization and mapping

Simultaneous localisation and mapping (SLAM) ist eine der wichtigsten grundlegenden Technologien für autonome mobile Roboter [SK16]. SLAM wird verwendet, um das Problem von autonomer Roboternavigation in unbekannten Umgebungen zu lösen. Ein SLAM-System ermöglicht es einem Roboter eine Karte einer Umgebung zu erstellen und gleichzeitig diesen Roboter in Relation zu der erstellten Karte zu lokalisieren. Für die Kartierung können verschiedene Sensoren eingesetzt werden. Die für die Kartierung aktuell am häufigsten eingesetzten Sensoren sind Kameras und Laserscanner.

Im Springer Handbook of Robotics [SK16] wird SLAM formal als ein probabilistisches Problem definiert. Die Pose eines Roboters zu einer Zeit t ist als x_t definiert. Bei Umgebungen mit einem flachen Untergrund besteht x_t aus einer 2D Koordinate und einem Rotationswert θ . Zusammen werden die 2D Koordinate und θ in einem 3D Vektor gespeichert. Ein SLAM System muss anfangs einen Initialwert x_0 setzen, der als Referenzpunkt für den Rest des Positionsschätzungsalgorithmus dient. Jeder Punkt $\{x_1, x_2, \dots, x_T\}$ nach der Initialisierung bis zum letzten Zeitpunkt T wird in Relation zu x_0 berechnet. Um die Relation zwischen Punkten x_n und x_{n+1} zu erfahren werden Odometriedaten, wie in Kapitel 2.1.1 beschrieben, verwendet.

Ein SLAM System besteht üblicherweise aus einem Front-End und einem Back-End [FTS15]. Das Front-End repräsentiert meist die Schnittstelle zu den Sensoren des Roboters und ist für die direkte Verarbeitung der Sensordaten wie z.B. für das Feature Matching verantwortlich. Im Front-End wird auch die sogenannte Datenassoziation betrieben, wobei unter anderem relevante Merkmale, auch Features genannt, der Umgebung aus den Sensordaten extrahiert und verschiedenen Landmarken zugeordnet werden. Das Back-End ist für die letztendliche Kartierung und Lokalisierung anhand der Daten aus dem Front-End zuständig. Diese

Aufteilung ist jedoch nur eine allgemeine, vereinfachte Beschreibung, SLAM Systeme können sich je nach Entwicklung und Anwendung stark sowohl in Struktur als auch in Funktionsweise unterscheiden.

SLAM Systeme kümmern sich auch um die Probleme, die bei dead-reckoning Verfahren auftreten. Hierbei wird in der Navigation die aktuelle Position anhand der vorigen Positionen bestimmt, wie es beispielsweise bei der in Kapitel 2.1.1 beschriebenen Radodometrie der Fall ist. Da Sensoren inhärent eine gewisse Fehlerquote aufweisen und jede neue Position des Roboters auf den vorigen Positionen basiert, wird der durch den Odometriesensor gemessene Pfad über längere Zeit immer vom realen Pfad des Roboters abweichen, was auch *Drift* genannt wird. Diese Abweichung akkumuliert sich immer weiter, wodurch ein gerader Pfad für den Roboter als Kurve erscheinen kann. Somit ist die Odometrie anhand von Drehwinkelmessern allein nicht ausreichend zur Lokalisierung eines mobilen Roboters und es müssen weitere Sensoren oder Verfahren verwendet werden, um den Drift zu kompensieren. Eine Methode ist dabei die sogenannte visuelle Odometrie, welche in der Regel bei Robotersystemen mit Kameras eingesetzt wird. Ein Beispiel, bei dem die visuelle Odometrie eine wichtige Rolle spielte, waren die Mars Rover Spirit und Opportunity, bei denen die visuelle Odometrie zusätzlich zur Radodometrie eingesetzt wurde [MCM07]. Hierbei wird die Pose des Roboters aktualisiert, indem die Features der Umgebung zwischen RGB-D Bildpaaren in Pixel- und Weltkoordinaten verglichen werden. Die Differenz dieser Koordinaten wird berechnet, wodurch eine Schätzung zur zurückgelegten Strecke gegeben werden kann. Falls jedoch zu wenige Features gefunden wurden oder der Roboter die Schätzung für zu unwahrscheinlich hält, wird als Back-up die Radodometrie verwendet.

Eine weitere wichtige Funktion von SLAM Systemen sind sogenannte Loop Closures [CCC⁺16]. Dabei handelt es sich um die Technik, die von dem Roboter erstellte Karte über Zeit durch Aktualisierung zu korrigieren. Wenn der Roboter an einen Punkt in der Umgebung gelangt, an dem er schon einmal war, kann die Karte der Umgebung und die Position des Roboters entsprechend aktualisiert werden. Somit wird auch durch den Einsatz von Loop Closures der durch Sensorfehler entstehende Drift eliminiert, ähnlich wie bei visueller Odometrie. Loop Closures helfen dem Roboter außerdem dabei, die reale Topologie der Umgebung zu erkennen, wodurch der Roboter leichter den kürzesten Weg zu seinem Ziel erfahren kann.

2.2.1 Visual simultaneous localization and mapping

Wenn für ein SLAM System der primäre Sensor zur Datenaufnahme eine Kamera ist, wird auch von visual simultaneous localisation and mapping (vSLAM)

gesprochen [YFB⁺20], [FPRARM15]. Wie in Kapitel 2.1.1 beschrieben wurde, sind Kameras aufgrund ihrer Kosten, ihres Gewichts und ihres Stromverbrauchs wirtschaftlich gesehen Laserscannern in einigen Anwendungsfällen überlegen. Hinzu kommt die höhere Datenvielfalt, die eine Kamera gegenüber Laserscannern liefern kann. Dabei kommen oft Systeme mit visueller Odometrie zusammen mit verschiedenen Methoden zur Datenassoziation zum Einsatz, um wie bei regulären SLAM Systemen eine Karte der Umgebung zu erstellen und den Roboter in dieser zu lokalisieren. In Kapitel 3.1.2 werden State of the Art vSLAM Systeme vorgestellt und näher beschrieben.

Ein Verfahren, welches bei vSLAM oft im Kern des Systems eingesetzt wird, ist Bundle Adjustment (BA) [ABL17]. BA ist ein Optimierungsproblem, wobei eine 3D Rekonstruktion der Umgebung anhand verschiedener Eingaben wie der Pose, Kalibrierung und Verzerrung einer Kamera aus mehreren Perspektiven erstellt wird. Dabei wird auch versucht, durch die Sensorik auftretende Fehler wie sie in Kapitel 2.1.1 beschrieben wurden, zu minimieren.

2.2.2 Nicht-stationäre Umgebungen

Für die Konzeption eines SLAM Systems ist es wichtig, die Art der Umgebung, in der das System eingesetzt werden soll, zu kennen. Dabei können Arten von Umgebungen in Hinsicht auf viele Aspekte definiert sein. Die häufigste Art, eine Umgebung für SLAM-Systeme zu klassifizieren, ist die Dynamik der Umgebung. So unterscheiden Hentschel et al. in [HW11] zwischen drei Arten von Objekten und Umgebungen:

- i. **Statisch:** Eine statische Umgebung verändert sich nicht in ihrer Struktur und beinhaltet keine Objekte, die sich aktiv bewegen oder mit der Zeit ihren Ort oder ihre Form ändern. Rein statische Umgebungen können in der Realität nicht existieren, können jedoch in Simulationen dargestellt werden. Daher wird auch von statischen Umgebungen gesprochen, wenn sich die Umgebung größtenteils nicht verändert und sich bewegende Objekte zwar vorkommen, jedoch eher eine Ausnahme darstellen und von einem SLAM System als Ausreißer betrachtet werden können.
- ii. **Semi-statisch:** Semi-statische Umgebungen verändern sich und beinhalten bewegliche Objekte. Diese Veränderungen finden jedoch in der Regel außerhalb der Sensorreichweite des betrachtenden statt und werden daher oft erst im späteren Verlauf eines Durchgangs erkannt.

iii. **Dynamisch:** Dynamische Veränderungen finden innerhalb der Sensorreichweite des Betrachters statt. Meist werden damit aktiv bewegende Objekte und Menschen gemeint, denen ein Roboter beispielsweise Ausweichen muss.

In der Realität bestehen die meisten Umgebungen aus allen drei dieser Kategorien. Wichtig ist dabei, welche Kategorie der Umgebung vorherrschend ist und über welchen Zeitraum die Umgebung betrachtet wird. Am Beispiel des Parkplatzes wird deutlich, dass die Umgebung sowohl als statisch, semi-statisch und dynamisch betrachtet werden kann. Über Nacht ist ein Großteil eines Parkplatzes statisch, da sich hier nur der Parkplatzuntergrund, nebenstehende Objekte wie Bäume oder Mülleimer, und über Nacht parkende Autos befinden. Wenn der Parkplatz jedoch am Tag über mehrere Stunden betrachtet wird, ist dieser oft primär semi-statisch, da sich in dieser Zeit die meisten Autos auf den Parkplatz oder von dem Parkplatz weg bewegen und somit die Umgebung ständig verändert wird. Über einen kurzen Zeitraum betrachtet kann der Parkplatz auch dynamisch wirken, wenn gerade mehrere Menschen zu ihren Autos laufen und auch Autos aktiv herumfahren oder am Ein- und Ausparken sind.

Umgebungen, die zwar statische Elemente beinhalten, deren Fokus jedoch primär auf semi-statischen und dynamischen Objekten liegt, werden im folgenden als nicht-stationäre Umgebungen bezeichnet [Sch21].

Morris et al. [MDC⁺14] haben beispielsweise ein SLAM System entwickelt, welches speziell auf nicht-stationäre Umgebungen angepasst ist. Ihre Testumgebung bestand aus einem Büro-Szenario, wobei Büroteiler und Stühle zwischen den Durchläufen bewegt wurden, um den Pfad des Roboters zu blockieren. Während den Durchläufen bewegten sich außerdem zwei Personen aktiv um den Roboter herum.

2.2.3 Long-Term simultaneous localization and mapping

Long-Term SLAM (LT-SLAM) Verfahren adressieren das Problem, die Karte der Umgebung entsprechend zu aktualisieren, wenn der Roboter Änderungen in der Umgebung erkennt. Dabei wird wie bei regulären SLAM Systemen der Roboter gleichzeitig innerhalb dieser Karte lokalisiert. Der Fokus liegt dabei auf dem längeren Einsatz ohne äußere Eingriffe des Roboters in einer ihm unbekannten Umgebung. Die Zeitspanne kann dabei zwischen mehreren Tagen oder Wochen, bis hin zu mehreren Jahren liegen. Da über längere Zeit viele Umgebungen, in denen mobile Roboter eingesetzt werden, nicht-stationär erscheinen, müssen LT-SLAM Systeme auch dafür gemacht werden, in nicht-stationären Umgebungen über längere Zeit robust zu agieren [Dör20].

Konolige und Bowman beschreiben in [KB09] einige Probleme, mit denen ein Langzeit-Kartierungssystem umgehen muss. So muss das System die Karte kontinuierlich aktualisieren, auch während der Lokalisierung. Es reicht also nicht aus, die Umgebung zu Beginn ein mal zu kartieren, um danach im Lokalisierungsmodus zu arbeiten, wie es bei einigen SLAM-Systemen für statische Umgebungen der Ablauf ist. Dazu müssen neue Loop Closures auch während der Laufzeit gesucht werden und bestehende Loop Closures müssen kontinuierlich optimiert werden. Der Roboter sollte sich auch jederzeit erneut lokalisieren können, wenn ein Fehler auftritt oder die Lokalisierung verloren geht. So können beispielsweise die Sensoren des Roboters blockiert werden, der Roboter kann von äußeren Kräften bewegt werden oder die Umgebung verändert sich während der Roboter sich beim Ladevorgang befindet.

Shi et al. fassten in [SLZ⁺20] die größten Herausforderungen an die Software eines LT-SLAM Systems zusammen, welche im folgenden vorgestellt werden:

- i. **Veränderte Objekte:** Wenn der Roboter eine bereits erkundete Umgebung erneut betritt, können sich Objekte bewegt oder verändert haben. Dies entspricht den in Kapitel 2.2.2 beschriebenen semi-statischen Objekten.
- ii. **Dynamische Objekte:** Wie in Kapitel 2.2.2 beschrieben, können sich dynamische Objekte innerhalb der betrachteten Szene aktiv bewegen oder sich anderweitig verändern.
- iii. **Veränderte Beleuchtung:** Die Beleuchtung kann sich verändert haben. Die Veränderung reicht dabei von kleinen Fluktuationen in Helligkeit oder Position der Beleuchtung, bis zu drastischen Veränderungen wie von Tageslicht zu Dunkelheit.
- iv. **Veränderter Blickwinkel:** Der Roboter kann Objekte oder komplett Umgebungen aus verschiedenen Richtungen betrachten.
- v. **Degradierende Sensoren:** Durch verschiedene äußerliche Einflüsse wie mechanischer Belastung, Temperaturveränderungen, Schmutz oder Nässe können unvorhersehbare Sensorfehler entstehen. Diese Fehler zeigen sich in der Regel durch verstärktes Rauschen oder durch Fehlkalibrierung.

Die ersten drei dieser Punkte fassen die in Kapitel 2.2.2 beschriebenen Herausforderungen zusammen, was zeigt, dass LT-SLAM primär in nicht-stationären Umgebungen eingesetzt wird. Zusätzlich werden hier in den Punkten vier und fünf die Herausforderungen beschrieben, die durch den längeren Einsatz eines Roboters zustande kommen, was nicht zwingend mit der Art der Umgebung zusammenhängt.

2.3 Robot Operating System

Das Robot Operating System (ROS) ist eine Open Source Sammlung von Softwarebibliotheken und Werkzeugen zur Entwicklung von Anwendungen für Roboter¹². Da Entwicklung und Programmierung von Roboterverhalten ein breit gefächertes, interdisziplinäres Feld ist, ist es wichtig, dass leicht zu verstehende Schnittstellen und Abstraktionen zwischen verschiedenen Komponenten einer Roboter Hard-, und Software existieren. Dafür bietet ROS verschiedene Hilfen an¹³, darunter Gerätetreiber, Visualisierungsprogramme wie Rviz¹⁴, Paketmanagement, sowie ein einheitliches Nachrichtenformat.

Ein wichtiges Konzept von ROS ist die Aufteilung von Programmen in sogenannte *Packages*, welche aus einzelnen *Nodes* bestehen, sowie das Veröffentlichen und Abonnieren von *Messages* über *Topics*. Im folgenden wird eine kurze Übersicht über diese Konzepte gegeben¹⁵:

- i. **Package:** Packages in ROS helfen dabei, Software zu organisieren. Packages können ausführbaren Code in Form von Nodes enthalten, können jedoch auch beispielsweise aus Konfigurationsdateien, Softwarebibliotheken oder Datensätzen bestehen. Packages machen Software leicht wiederverwendbar und organisiert. Wenn eine bereits bestehende Softwarekomponente notwendig ist, kann diese meist leicht in Form eines Packages installiert und verwendet werden.
- ii. **Node:** Ein Node ist ein ausführbarer Rechenprozess. Ein Roboter Kontrollsystem besteht in der Regel aus vielen Nodes, wobei jeder Node eine spezifische Aufgabe wie das Verarbeiten von Bilddaten oder das Aufbauen einer Karte erfüllt. Nodes kommunizieren untereinander über Streams, welche über Topics ablaufen.
- iii. **Topic:** Eine Topic ist ein Bus, also ein System zur Datenübertragung mit einem gemeinsamen Übertragungsweg, über den Nodes Messages austauschen. Topics verfügen über einen Namen, mit dem sie eindeutig identifiziert werden können. Nodes können mit Hilfe von Topics Nachrichten bestimmter Typen anonym veröffentlichen oder abonnieren. Dabei können mehrere Nodes über die gleiche Topic Daten anfragen oder verschicken. Woher die Nachricht kommt oder an wen die Nachricht gesendet wird, ist für die jeweiligen Nodes nicht relevant. Der Nachrichtenaustausch über Topics ist unidirektional.

¹²<https://www.ros.org/about-ros/>

¹³<http://wiki.ros.org/Documentation>

¹⁴<http://wiki.ros.org/rviz>

¹⁵<http://wiki.ros.org/ROS/Tutorials/UnderstandingNodes>

- iv. **Message:** Messages in ROS sind einfache Datenstrukturen, welche aus typisierten Feldern bestehen. Die unterstützten Datentypen sind dabei primitive Typen wie Integer, Floats, etc. sowie Arrays von primitiven Typen. Ähnlich wie Structs in C können Strukturen und Arrays in Messages beliebig verschachtelt werden.
- v. **Services:** Services erweitern die Funktionen von Topics und Messages um ein Request / Response System, wobei ein Node einen Service unter einem Namen anbietet und ein Client einen Request an den Node sendet und darauf eine Response erhält.

2.3.1 Roslaunch

Zusätzlich bietet ROS das Werkzeug roslaunch¹⁶ an, welches über ein spezielles XML-Format¹⁷ erlaubt, mehrere Nodes mit einem einzelnen Befehl zu starten. Dazu werden eine oder mehrere .launch Dateien mit entsprechenden Parametern, Nodes und Maschinen an roslaunch übergeben, wodurch dann alle definierten Programme auf den zugehörigen Maschinen ausgeführt werden.

2.3.2 ROS-Bags

ROS bietet über sogenannte *Bags*¹⁸ ein Dateiformat, in dem Daten aus ROS Messages gespeichert werden können. Bags sind in ROS der primäre Mechanismus zur Datenerfassung. Bags werden in der Regel mit dem Werkzeug rosbag¹⁹ erstellt, das ähnlich wie Nodes eine Topic subscribed und die empfangenen Daten der Message serialisiert in einer .bag Datei speichert. Diese .bag Datei kann in ROS wieder abgespielt werden, um die gespeicherten Daten wieder über Topics anzubieten. So macht es für Nodes in ROS keinen Unterschied, ob ein realer Roboter Daten sendet, oder nur die früheren Daten eines Roboters abgespielt und als Daten gesendet werden. In der Forschung werden Bags oft verwendet, um Datensätze aufzunehmen, diese zu visualisieren, die Daten zu labeln und zur späteren Verwendung zu speichern. Bags helfen dabei, Experimente für Robotersoftware wiederholbar zu machen, da immer exakt die selbe Sequenz an Daten abgespielt wird. So können verschiedene Verfahren hinsichtlich der Performance und der Ergebnisse an einem Datensatz verglichen und evaluiert werden. So sind im TUM RGB-D Datensatz aus [SEE⁺¹²] die Daten jeweils entweder als

¹⁶<http://wiki.ros.org/roslaunch>

¹⁷<http://wiki.ros.org/roslaunch/XML>

¹⁸<http://wiki.ros.org/Bags>

¹⁹<http://wiki.ros.org/rosbag>

.tgz Archiv mit Einzelbildern oder als komplette .bag Datei verfügbar. Wenn eine in ROS geschriebene Robotersoftware auf einem Datensatz getestet werden soll, ist es sehr hilfreich, wenn die Daten bereits im Bag Format vorhanden sind. Wenn wiederum die einzelnen Bilder eines Bags gebraucht werden, können diese mit verschiedenen Tools extrahiert werden.

2.4 Robotersimulation

Simulationen spielen beim Entwickeln und Testen von Robotern eine große Rolle. So beschreibt Dörr in [Dör20], dass Referenzdaten, oder sogenannte Ground Truth Daten, unabdingbar für die Evaluation von SLAM Systemen sind. Die Ground Truth beschreibt die reale Pose des Roboters als auch den wirklichen Aufbau der Umgebung. Diese Referenzdaten können entsprechend mit den Schätzungen des SLAM-Systems verglichen werden, wodurch das SLAM-System in Genauigkeit bewertet werden kann. Jedoch sind Ground Truth Daten in der realen Welt meist nicht, oder zumindest in nicht ausreichender Präzision, verfügbar, da diese Daten ebenfalls mit zusätzlichen Sensoren aufgenommen werden müssen. So bietet der bekannte TUM RGB-D Datensatz aus [SEE⁺12] viele Sequenzen mit kompletten Ground Truth Daten. Jedoch sind bei längeren Sequenzen aufgrund von Limitationen des eingesetzten Motion-Capture Systems keine Ground Truth Daten für die gesamte Strecke verfügbar. So sind bei der Sequenz *freiburg2_large_no_loop*²⁰, die für SLAM-Systeme konzipiert wurde, nur am Anfang und am Ende der Sequenz Ground Truth Daten hinterlegt. Daher sind virtuelle Simulationen für Roboter ein beliebter erster Schritt, um ein System zur Lokalisierung und Kartierung zu testen. Simulationen können zwar die reale Welt nur bis zu einem gewissen Punkt realistisch nachbilden, bieten dafür aber große Flexibilität bei den simulierten Umgebungen, detaillierte Ground Truth Daten und auch je nach Bedarf bis zu 100% Reproduzierbarkeit eines Testszenarios. All diese Punkte sind besonders wichtig im wissenschaftlichen Umfeld. Experimente mit echten Robotern, vor allem in nicht-stationären Umgebungen, sind streng genommen nicht reproduzierbar [SNS11]. Um quantifizierbare Ergebnisse über das Verhalten eines Roboters zu erhalten, muss eine ausreichend große Zahl von Durchläufen eines Experiments vorhanden sein, was in der Regel sehr zeitaufwändig ist. Castillo et al. beschreiben in [CPATT10] die Simulation von Robotersystemen außerdem als ein essenzielles Werkzeug für die Lehre, Wissenschaft und Entwicklung. Um die Grundlagen und Physik von Robotern in der Lehre effektiv zu vermitteln, sollten diese entsprechend visualisiert sein, oder es sollten reale Experimente mit Robotern stattfinden. Da Roboter oft teuer sind, bieten Simulationen eine kostengünstige Alternative, um mit Robotern zu experimentieren und diese zu Testen.

²⁰<https://vision.in.tum.de/data/datasets/rgbd-dataset/download>

Abbildung 2.3 zeigt einen beispielhaften Arbeitsablauf zur Entwicklung von Roboterapplikationen. Sobald der Code für einen bestimmten Anwendungszweck geschrieben wurde, kann dieser in einer Simulation schnell getestet werden. Anschließend werden die aus den Tests entstehenden Ergebnisse evaluiert. Wenn die Anwendung alle Tests besteht, muss sie noch auf einem realen Roboter getestet werden, da eine Simulation die Realität und vor allem das Verhalten eines realen Roboters nicht perfekt widerspiegelt und somit nicht allein ausreicht, um ein Robotersystem zu testen.

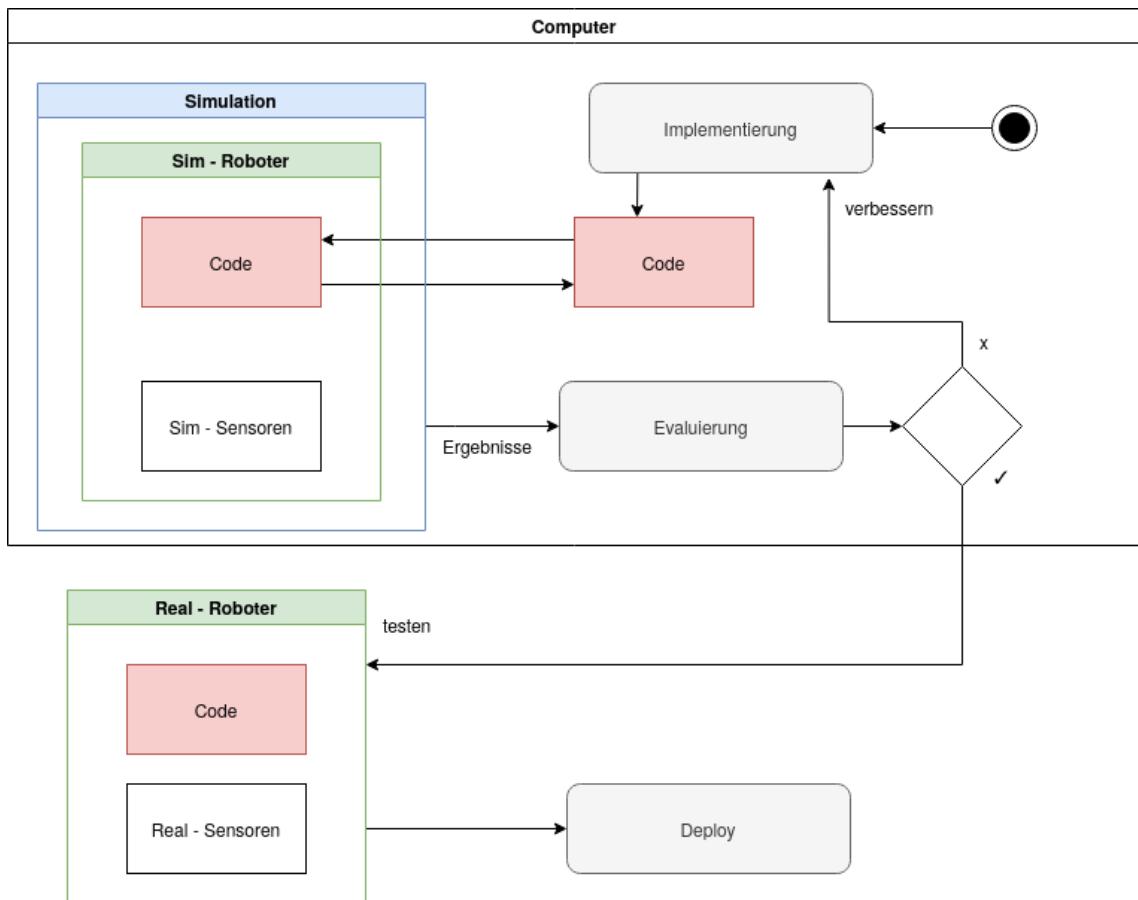


Abbildung 2.3: Arbeitsablauf bei der Entwicklung von Roboterapplikationen. Informationen von Unity's Unite Now²¹.

Damit der Einsatz einer Simulation sinnvoll ist, muss diese eine Reihe von Anforderungen erfüllen. Im folgenden werden einige Anforderungen an eine Robotersimulation von Hertzberg et al. [HLN12] zusammengefasst:

- i. Die Umgebung muss die Realität ausreichend simulieren. Dazu gehören sowohl die Qualität von Modellen und Texturen, als auch der realistische Auf-

²¹<https://resources.unity.com/unitenow/onlinesessions/simulating-robots-with-ros-and-unity> (Video ab 2:23)

bau der Umgebung und der Objekte die sich darin befinden. So müssen im besten Fall auch semi-statische Objekte und dynamische Objekte simuliert werden, wenn die reale Umgebung diese auch beinhaltet würde.

- ii. Der Roboter muss in Aktorik und Sensorik hinreichend gut simuliert sein. So sollte jeder Sensor des realen Roboters auch in der Simulation vorhanden sein und Messergebnisse des selben Typs liefern. Darüber hinaus sollten auch Ungenauigkeiten von Sensoren wie Messfehler und Rauschen in der Simulation abgebildet werden.
- iii. Der Roboter darf nur über die simulierten Aktoren und Sensoren mit der Simulationsumgebung interagieren. In der Simulation sind alle Zustände präzise bekannt, wie die Geschwindigkeit des Roboters oder die Position eines Objekts. Der Roboter darf auf diese Informationen keinen Zugriff haben, sondern die Umgebung nur über seine Sensoren beobachten. Lediglich zur Evaluation der Robotersoftware sind die Informationen aus der Simulation als Ground Truth Daten zu verwenden.
- iv. Die Software und die Schnittstelle müssen für die Simulation die selbe sein, wie für den realen Roboter. So muss am Beispiel von ROS der Roboter in der Simulation über entsprechende Messages aus ROS gesteuert werden und seine Sensordaten über Topics publishen. Für die Nodes in ROS macht es also keinen Unterschied ob die Daten von einem echten Roboter, aus einem Video, oder aus einer Simulation stammen.

Da Simulationen bereits seit den Anfängen der Robotik wichtig waren [CPATT10], existiert bereits eine Vielzahl an Programmen, die auf Robotersimulation spezialisiert sind. In den folgenden Kapiteln wird eine Auswahl an Anwendungen zur Robotersimulation vorgestellt.

2.4.1 Gazebo

Der Gazebo Simulator²² ist ein kostenloser Open Source Robotersimulator für 2D und 3D Welten. Gazebo wurde 2004 in [KH04] vorgestellt, in einer Zeit, in der die meisten Simulatoren auf 2D Welten beschränkt waren. Die Entwickler legten dabei Wert auf einen hohen Grad an Realismus, so verfügen die simulierten Objekte über physische Eigenschaften wie Masse und Reibung [NPRC17]. Gazebo ist in ROS integriert und somit weit verbreitet.

Gazebo ist vor allem auf statische Umgebungen mit wenigen dynamischen Objekten, meist die zu testenden Roboter, spezialisiert. Es bestehen Möglichkeiten,

²²<http://gazebosim.org/>

semi-statische und dynamische Objekte zu integrieren, jedoch nur bis zu einem gewissen Grad und mit hohem Aufwand. So gibt es auf GitHub das Projekt *dynamic_logistics_warehouse*²³, welches ein Warenlager mit verschiedenen Objekten wie Kisten und Regalen, sowie neun bewegliche, animierte Personen simuliert. Die Animationen sind dabei jedoch rudimentär und die Personen können sich nur auf einer jeweils vorgegebenen Linie bewegen. Ein weiteres Problem von Gazebo ist das System zur Einbindung von neuen Umgebungen und Objekten. Alle Objekte müssen als Universal Robot Description Format (URDF) oder Simulation Description Format (SDF)²⁴ Dateien vorliegen, welche primär zur Beschreibung von Robotern konzipiert sind. Zwar sind URDF und SDF praktische Formate und sind gut dazu geeignet, komplett Roboter in wenigen Dateien abzubilden, jedoch müssen auch primitive Objekte wie einfache Boxen oder komplexere Umgebungsinhalte wie Lichter in URDF oder SDF vorliegen. Dabei liegt der Fokus auf XML als Format, wodurch die Dateien leicht zu speichern und zu teilen sind. Das Lesen und Bearbeiten der Dateien durch Menschen ist dadurch jedoch erschwert, da die Dateien oft mehrere hundert Zeilen lang sind. Dabei werden oft weitere Dateien und Plugins eingebunden, was die Daten noch komplexer macht. Gerade für Anfänger der Thematik wird so die Integration und Entwicklung erschwert und verlangsamt.

Insgesamt eignet sich der Gazebo Simulator vor allem für statische Umgebungen, sowie für Roboter, die primär mit Laserscannern ausgestattet sind. Zur Simulation von nicht-stationären Umgebungen für Roboter mit Kamerasystemen ist Gazebo jedoch weniger geeignet.

2.4.2 Unity

Unity ist eine Plattform zur Entwicklung von interaktivem Echtzeit-Content in 2D und 3D mit C# als primärer API für Scripts. Unity wurde als Game Engine entwickelt, wird jedoch mittlerweile auch für eine Vielzahl von Anwendungen außerhalb der Videospielindustrie verwendet. So werden auf der Webseite Unity Solutions²⁵ einige Projekte außerhalb der Videospieleindustrie vorgestellt, darunter auch Simulationen für Roboteranwendungen²⁶. So existieren mehrere Ansätze von namhaften Firmen, die Unity zur Robotersimulation verwenden. Darunter Unity selbst mit dem Unity Robotics Hub²⁷, Siemens mit ROS#²⁸, Microsoft mit

²³https://github.com/belal-ibrahim/dynamic_logistics_warehouse

²⁴<http://sdformat.org/>

²⁵<https://unity.com/solutions>

²⁶<https://unity.com/solutions/automotive-transportation-manufacturing/robotics>

²⁷<https://github.com/Unity-Technologies/Unity-Robotics-Hub>

²⁸<https://github.com/siemens/ros-sharp>

AirSim²⁹ und Nvidia mit Isaac³⁰.

Game Engines wie Unity bieten eine attraktive Alternative zu den speziell für die Robotersimulation entwickelten Simulatoren. So hat Konrad in [Kon19] Unity mit Gazebo in vielen Aspekten verglichen. Gazebo hat in einigen Fällen realistischere Ergebnisse bezüglich der Physiksimulation geliefert und ist somit besser geeignet, um Hardwarekonfigurationen von Robotern zu validieren. Ansonsten ist Unity vergleichbar mit Gazebo bezüglich der Ergebnisse aus Benchmarks. Gerade als Simulator für mobile Roboter und SLAM Systeme ist Unity nach Aussage der Studie geeignet.

In Abbildung 2.4 sind die Messungen einer simulierten Tiefenkamera in Form eines Farbbilds, eines Tiefenbilds und einer aus den Farb- und Tiefenbildern generierte Punktwolke abgebildet.

2.4.3 Koordinatensysteme

Roboter bestehen in der Regel aus vielen verschiedenen Komponenten wie Akto- ren, Sensoren und Verbindungsteilen. Dabei muss die Robotersoftware Informa- tionen über die gegenseitige räumliche Lage der Teile besitzen, um z.B. Odome- triedaten mit anderen Sensordaten wie die von Laserscannern zu verbinden, um eine einheitliche Karte zu erstellen. ROS verwendet dazu das tf³¹ Package, wel- ches dem Nutzer helfen soll, mehrere Koordinatensysteme über Zeit zu verwal- ten. Als Visualisierungswerkzeug kommt dabei der sogenannte TF-Tree zum Ein- satz, der sowohl statische, als auch dynamische Verknüpfungen abbilden kann. In Abbildung A1 ist ein TF-Tree beispielhaft dargestellt. Es existieren jedoch ver- schiedene Arten von Koordinatensystemen. So gibt es bei 3D Koordinatensyste- men grundlegend links- und rechtshändige Systeme. So ist bei linkshändigen Koordinatensystemen die Drehung in die positive Richtung ausgehend von der nach oben zeigenden Achse im Uhrzeigersinn, was auch linksdrehend genannt wird. In einem rechtshändigen Koordinatensystem ist diese Drehung entspre- chend gegen den Uhrzeigersinn, auch rechtsdrehend genannt. Ebenfalls wichtig ist die Zuteilung der Achsen zu den Richtungen. So zeigt bei ROS die Z-Achse beispielsweise nach oben, bei Unity hingegen zeigt die Y-Achse nach oben. So ist es also wichtig wenn ROS mit externen Anwendungen wie einer Simulation verknüpft wird, die Koordinatensysteme entsprechend zu transformieren. Abbil- dung 2.5 zeigt die Zuteilung der Achsen zu den Richtungen von Unity und ROS, sowie eine Tabelle, die eine Konvertierung der Achsen zwischen Unity und ROS und deren Rotationsrichtungen enthält.

²⁹<https://microsoft.github.io/AirSim/Unity/>

³⁰<https://docs.nvidia.com/isaac/isaac/doc/simulation/unity3d.html>

³¹<http://wiki.ros.org/tf>

³²https://github.com/siemens/ros-sharp/wiki/Dev_ROSUnityCoordinateSystemConversion

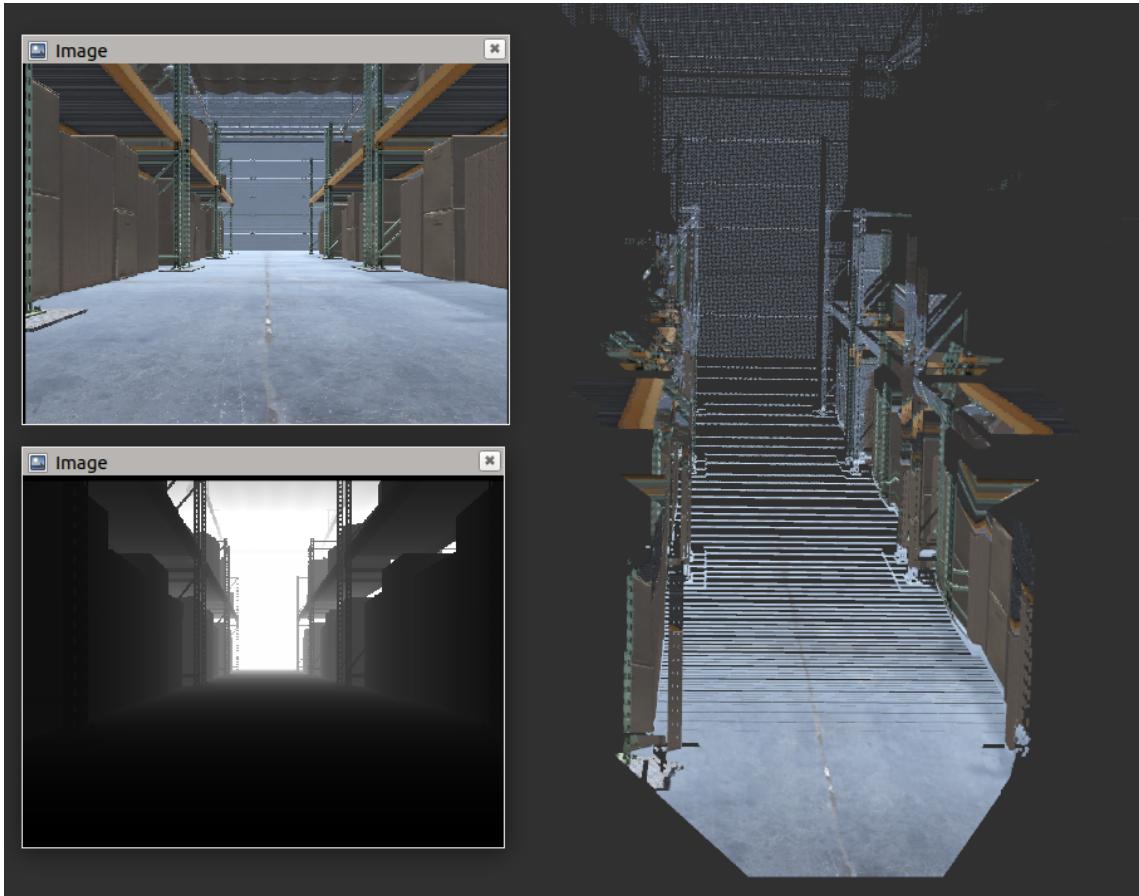


Abbildung 2.4: Darstellung der Daten von einer Farbkamera (links oben), einer Tiefenkamera (links unten) sowie einer farbigen Punktwolke (rechts). Die Umgebung wurde in Unity simuliert und in Rviz visualisiert.

2.5 Segmentierung von Punktwolken

Im Feld der Bildverarbeitung ist das Segmentieren von 2D Bildern ein klassisches und viel erforschtes Problem [NL13]. Seitdem Methoden zur Aufnahme von 3D Daten weit verfügbar sind, wurde in der Forschung und Industrie mehr Aufmerksamkeit auf das Verarbeiten von Punktwolken gelegt. So wurde beispielsweise 2011 die Point Cloud Library (PCL) erstellt, in der Tutorials und State of the Art Algorithmen zur Verwendung mit Punktwolken beschrieben werden. Um Szenen besser zu verstehen und relevante Objekte und Regionen zu erkennen, müssen Punktwolken entsprechend segmentiert werden. Die Segmente können dann verwendet werden, um Objekte zu erkennen, zu lokalisieren und zu klassifizieren. Shamir [Sha06] beschreibt in seinem Survey einige Methoden zur Segmentierung von Punktwolken, darunter convex decomposition, watershed analysis,

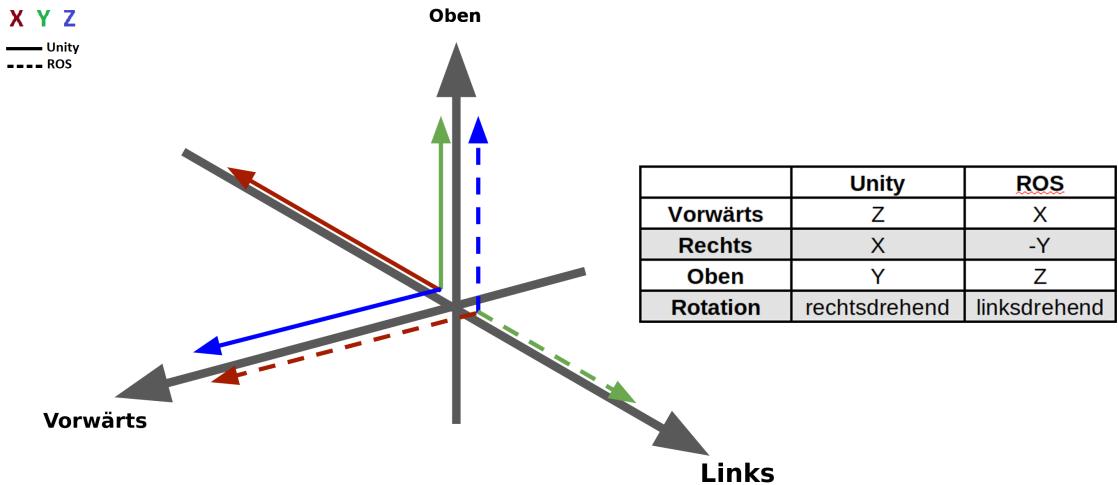


Abbildung 2.5: Zuteilung und Richtung der Achsen in den 3D-Koordinatensystem in Unity und ROS, angepasst aus der ROS# Dokumentation³²

hierarchical clustering, region growing und spectral clustering, welche in Kapitel 3.2 näher beschrieben werden. Die meisten dieser Algorithmen werden dazu verwendet, um Regionen zu erkennen. Anguelov [ATC⁺05] schlug 2005 vor, dass Segmentierungsalgorithmen für 3D Punktwolken drei Eigenschaften aufweisen sollten:

- i. Der Algorithmus sollte ausnutzen, dass verschiedene Teile einer Umgebung verschiedene Features besitzen. So besteht der Boden oft aus einer Fläche und Bäume unterscheiden sich stark von Autos in ihrem Aussehen in der Punktwolke.
- ii. Punkte in Regionen mit wenigen anderen Punkten oder wenigen Informationen sollten zusammen mit anderen nahen Punkten in eine Region zusammengefasst werden.
- iii. Der Algorithmus sollte auf den eingesetzten Sensor angepasst werden. So unterscheiden sich z.B. Laserscanner, Tiefenkameras und synthetische Punktwolken stark in ihrem Aussehen.

Das Segmentieren von Punktwolken ist keine triviale Aufgabe [NL13]. So sind die Daten von 3D Punktwolken oft unorganisiert, spärlich und beinhalten Rauschen. Außerdem ist aufgrund physischer Eigenschaften der Sensoren die Dichte der aufgenommenen Punkte oft nicht gleichmäßig verteilt. Zudem ist oft der Vordergrund stark mit dem Hintergrund verbunden, wie es beispielsweise in Abbildung 2.2 in der Punktwolke zu sehen ist.

2.6 3D Feature Matching

Image matching, also das Erkennen von Gemeinsamkeiten zwischen verschiedenen Bildern, ist eine der fundamentalen Aspekte vieler Probleme im Feld der Bildverarbeitung [Low04]. Diese Probleme beinhalten beispielsweise Objekterkennung, 3D-Rekonstruktion der Umgebung aus Bildern oder Motion Tracking. Hierfür werden in der Regel lokale invariante Features verwendet wie Scale Invariant Feature Transform (SIFT) [Low04] oder Oriented FAST and Rotated BRIEF (ORB) [RRKB11] verwendet. Lokale invariante Features werden bereits erfolgreich in vielen Bereichen der Bildverarbeitung wie der Objekterkennung oder Kategorisierung von Objekten eingesetzt. Dabei besteht das Verfahren zur Merkmalsbestimmung aus zwei Teilen. Zuerst kommt der Schritt der Detektierung, bei dem interessante und hervorstechende Regionen in dem Bild gesucht werden, meist nach einem bestimmten Schema wie z.B. der Suche nach Ecken, Kanten oder Helligkeitsunterschieden. Das Ergebnis aus diesem Schritt besteht aus mehreren Punkten, welche *Keypoints* genannt werden. Jeder Keypoint besteht zu diesem Zeitpunkt in der Regel aus einer Position. Im zweiten Schritt wird für jeden Bildausschnitt an dem sich ein Keypoint befindet, ein sogenannter *Deskriptor* erstellt. Ein Deskriptor besteht aus einem Algorithmus, der in einem Bild hervorstechende Informationen in einen numerischen *Fingerabdruck* speichert. In der Regel werden lokale Deskriptoren verwendet, bei denen die Regionen um die Keypoints kodiert werden. Die von Deskriptoren kodierten Informationen sind dabei im besten Fall auch bei Transformationen des Bilds unverändert, damit das Merkmal auch bei einem veränderten Bild oder veränderten Blickwinkel wieder gefunden werden kann. [YN10] Um Objekte aus Bildern wieder zu erkennen, müssen die gesammelten Features aus verschiedenen Bildern in einer Art Datenbank gespeichert werden. Wenn nun ein erkanntes Objekt aus dieser Datenbank in einem neuen, bisher unbekannten Bild gefunden werden soll, muss jeder einzelne Keypoint aus der Datenbank mit jedem Keypoint des neuen Bildes miteinander verglichen werden. Dies ist je nach Menge der Keypoints rechnerisch fordernd, weshalb oft alternative Arten zur Speicherung und zum Matching verwendet werden. Beispiele hierfür sind das bag-of-visual-words (BOVW) Modell oder ein k-dimensional tree (kd-tree).

Ähnliche Methoden können ebenfalls auf Punktwolken statt auf Bildern angewendet werden. So existieren bereits einige 3D-Feature Deskriptoren³³ wie das Normal Aligned Radial Feature (NARF) [SK10], der Rotation-Invariant Feature Transform (RIFT) [LSP05] und das Viewpoint Feature Histogram (VFH) [RBTH10].

³³<https://github.com/PointCloudLibrary/pcl/wiki/Overview-and-Comparison-of-Features>

3 State of the Art

3.1 Simultaneous localization and mapping

Im Folgenden wird der State of the Art von visual simultaneous localisation and mapping (vSLAM) anhand existierender Methoden und Datensätzen beschrieben.

3.1.1 Datensätze

Um den State of the Art für SLAM Systeme festzustellen, ist eine frei zugängliche und standardisierte Methode um die Systeme zu testen notwendig. Eine häufig eingesetzte Form um dies zu erreichen, sind öffentliche Datensätze. Im folgenden wird eine Auswahl an für vSLAM relevanten Datensätzen vorgestellt und näher beschrieben.

TUM RGB-D SLAM Dataset

Das TUM RGB-D SLAM Dataset wurde von Sturm et al. [SEE⁺12] in 2012 vorgestellt. Der Datensatz ist unter der Creative Commons Attribution Lizenz auf der Webseite der Technischen Universität München¹ verfügbar. Der Datensatz beinhaltet primär RGB-D Daten von einer Microsoft Kinect in voller 640 x 480 Pixel Auflösung, sowie Farb- und Tiefenbilder, die mit einer Frequenz von 30 Hertz (Hz) aufgenommen wurden. Zusätzlich wurden Ground Truth Daten von einem Motion Capture System bestehend aus acht Hochgeschwindigkeits-Trackingkameras mit 100 Hz aufgenommen. Der Datensatz umfasst 39 Sequenzen aus verschiedenen Szenen in einem Büro und in einer Industriehalle. Die Kinect-Kamera wurde dabei teilweise per Hand und teilweise von einem Pioneer 3 Roboter geführt. Somit ist der Datensatz zur Evaluierung mobiler Roboter mit RGB-D Sensoren geeignet. In Abbildung 3.1 ist ein Ausschnitt aus der *rgbd_dataset_freiburg2_large_with_loop* Sequenz des TUM Datensatzes abgebildet.

¹<http://vision.in.tum.de/data/datasets/rgbd-dataset>



Abbildung 3.1: Ein Ausschnitt aus dem TUM RGB-D SLAM Datensatz aus der Sequenz *rgbd_dataset_freiburg2_large_with_loop*. Links: Bild der Farbkamera. Rechts: Bild der Tiefenkamera.

OpenLORIS-Scene datasets

Die OpenLORIS-Scene datasets wurden 2020 von Shi et al. in [SLZ⁺20] vorgestellt und sind im Internet unter der CC BY-ND 4.0 Lizenz verfügbar². Der Datensatz wurde speziell zur Evaluation von Long-Term SLAM (LT-SLAM) Systemen entworfen. Entsprechend beinhaltet der Datensatz Aufnahmen von realen Szenen innerhalb verschiedener Gebäude. Jeder Ort wurde dabei auch zu verschiedenen Zeitpunkten besucht, um die Veränderungen der Umgebung über die Zeit und außerhalb der Sensorreichweite abzubilden. Die OpenLORIS-Scene datasets heben sich von anderen Datensätzen in einigen Punkten ab. Die Daten wurden beispielsweise in realen Umgebungen mit Menschen aufgenommen, darunter ein Café, ein Büro und ein Supermarkt. Jede Szene wurde mehrmals durchlaufen, wodurch semi-statische Objekte und veränderte Beleuchtung mit aufgenommen werden können. Zusätzlich wurden mehrere Sensoren wie RGB-D, Stereo-Kameras mit Fischaugeobjektiven, inertiale Messeinheit (IMU), Radodometrie und light detection and ranging (LIDAR) eingesetzt, um eine Vielzahl an Algorithmen testen zu können. In Abbildung 3.2 ist ein Ausschnitt der *corridor1-1.bag* Datei aus dem OpenLORIS Datensatz abgebildet. Zu sehen sind die Daten der RGB-D Kamera und des 2D-Laserscanners.

²<https://shimo.im/docs/HhJj6XHYhdRQ6jjk/read>

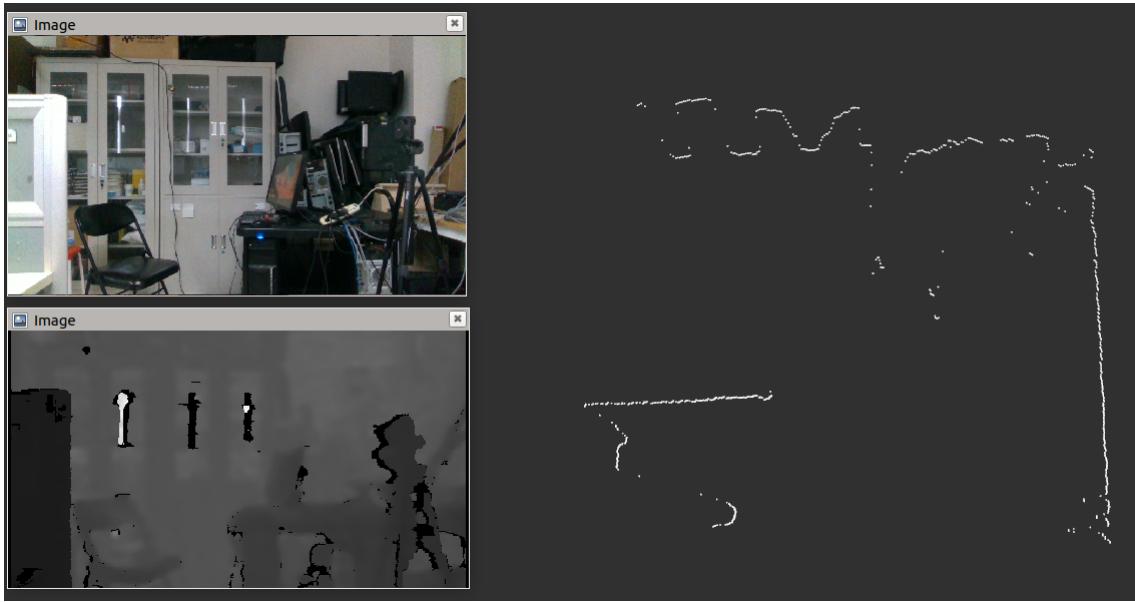


Abbildung 3.2: Ausschnitte aus den OpenLORIS-Scene Datasets. Links oben: Bild der Farbkamera. Links unten: Bild der Tiefenkamera. Rechts: Punkte des 2D-Laserscanners.

OSD

Die Object Segmentation Database (OSD) wurde von Richsfeld [Ric13] im Jahr 2013 vorgestellt und ist auf der Webseite der TU Wien verfügbar³. Der Datensatz besteht aus 111 Szenen, in denen verschiedene Objekte auf einem Tisch unterschiedlich angeordnet sind. Primärer Zweck des Datensatzes ist das Testen von Segmentierungsalgorithmen, speziell das Segmentieren unbekannter Objekte in RGB-D Bildern. Der Datensatz wurde manuell in Segmente unterteilt und mit Labels versehen und beinhaltet somit eine zuverlässige Ground Truth. Die Daten sind in Form von Bildern, sowie in Form von Punktfolgen im pcd Format in binär kodierter Form verfügbar. Die pcd Dateien enthalten dabei die Label für die Ground Truth. Abbildung 3.3 zeigt das Bild test_56 aus der OSD.

3.1.2 Bestehende Systeme

Merzlyakov und Macenski geben in [MM21] einen ausführlichen Vergleich zwischen modernen und universell einsetzbaren vSLAM-Systemen. Der Fokus wird dabei auf die drei Systeme ORB-SLAM3 [CER⁺21], OpenVSLAM [SSS19] und RTAB-Map [LM19] gelegt, welche den State of the Art für frei verfügbare, generische

³<https://www.acin.tuwien.ac.at/en/vision-for-robotics/software-tools/osd>



Abbildung 3.3: Ausschnitt aus der Object Segmentation Database.

vSLAM Systeme darstellen. Im folgenden werden die Funktionsweisen sowie die Stärken und Schwächen dieser drei Systeme vorgestellt.

ORB-SLAM3

Oriented FAST and Rotated BRIEF (ORB)-SLAM3 [CER⁺21] ist der aktuellste Ableger des seit 2015 laufenden ORB-SLAM Projekts und ist öffentlich auf GitHub verfügbar⁴. ORB-SLAM ist ein vSLAM System, welches eine breite Auswahl an Sensoren wie monokulare Kameras, Stereokameras und RGB-D Kameras unterstützt. ORB-SLAM verfügt über drei primäre parallele Threads [MAT17], jeder dieser Threads kümmert sich um eine von drei Arten der Datenassoziation [CER⁺21]:

- i. **Kurzfristige Datenassoziation:** Während der kurzfristigen Datenassoziation werden die Elemente, die in den letzten Sekunden erfasst wurden, verarbeitet. Die meisten Systeme zur reinen visuellen Odometrie verwenden

⁴https://github.com/UZ-SLAMLab/ORB_SLAM3

alleinig die kurzfristige Datenassoziation, was zu dauerhaften Abweichungen der Schätzungen führen kann. Sobald hier Merkmale der Umgebung aus der Sicht des Roboters verschwinden, werden diese direkt wieder vergessen.

- ii. **Mittelfristige Datenassoziation:** Bei der mittelfristigen Datenassoziation werden der Kamera nahe Elemente mit der Karte verglichen und im Bundle Adjustment (BA) eingesetzt. Somit kann der Drift in kartierten Umgebungen eliminiert werden. Kurz- und mittelfristige Datenassoziation kommt bei ORB-SLAM aus den Deskriptoren von ORB.
- iii. **Langfristige Datenassoziation:** Durch Place-Recognition Verfahren können Objekte in bereits erkundeten Umgebungen wieder erkannt werden, egal ob sich Drift akkumuliert hat, die Karten zusammengeführt werden müssen oder der Roboter seine Position nicht mehr kennt. Durch die langfristige Datenassoziation kann der Drift zurückgesetzt werden und die Karte durch Graphoptimierung und BA aktualisiert werden. Dies ist der Schlüssel zu genauen SLAM Systemen in großen Umgebungen. Für Loop Closures und Relokalisierung wird die bag-of-words Bibliothek DBoW⁵ [GLT12] verwendet.

Campos et al. beschreiben in ihrer Arbeit [CER⁺21], dass ORB-SLAM das bisher einzige vSLAM-System ist, welches alle drei dieser Typen der Datenassoziation integriert, was der Schlüsselfaktor für die hohe Genauigkeit von ORB-SLAM sei.

Die namensgebenden ORB-Features aus [RRKB11] werden bei ORB-SLAM sowohl zur Lokalisierung, Kartierung als auch zur Place Recognition verwendet. ORB-Features wurden ausgewählt, da sie robust gegenüber Rotation und Skalierung sind und sich auch bei sich automatisch ändernden Kameraparametern wie der Lichtempfindlichkeit (Gain) und des Lichtwerts (Exposure) nicht verändern. Außerdem können ORB-Features schnell extrahiert und verglichen werden, was sie nützlich für Echtzeitsysteme wie Roboter macht. Zudem erzielen sie gute Ergebnisse in Place-Recognition Algorithmen, die den bag-of-words (BOW)-Ansatz verwenden.

ORB-SLAM wurde auf verschiedenen Datensätzen getestet. So wurde ORB-SLAM2 [MAT17] auf dem KITTI Datensatz, dem TUM RGB-D SLAM Datensatz und dem EuRoC Datensatz evaluiert. ORB-SLAM2 hat in einigen Fällen die höchste Genauigkeit im Vergleich mit anderen State of the Art vSLAM-Systemen erzielt und war zum Zeitpunkt der Veröffentlichung der Arbeit in 2017 die beste Lösung für das Stereo SLAM Problem von KITTI. ORB-SLAM3 wurde auf dem EuRoC Datensatz

⁵<https://github.com/dorian3d/DBoW2>

und auf dem KITTI Datensatz getestet, wo ORB-SLAM3 hinsichtlich der Genauigkeit die anderen Systeme übertraf. Die größte Schwäche von ORB-SLAM sind Umgebungen mit wenig Texturen, da hier oft wenige Features existieren [CER⁺21].

RTAB-Map

RTAB-MAP [LM19] ist eine Open-Source Bibliothek für vSLAM und auf LIDAR basierendem SLAM. RTAB-MAP ist frei auf GitHub⁶ und als Robot Operating System (ROS)-Package⁷ verfügbar. RTAB-MAP wurde 2009 ursprünglich als System zur visuellen Erkennung von Loop Closures mit Fokus auf langfristige Operation in großen Umgebungen konzipiert und ist seit 2013 als Open-Source Bibliothek verfügbar. Über Zeit wurde die Bibliothek erweitert und ist nun ein vollwertiges SLAM-System. Da RTAB-Map sowohl mit Kameras als auch mit LIDAR als primäre Sensoren funktioniert, die Odometriedaten aus einer beliebigen externen Quelle kommen können und RTAB-MAP in ROS integriert ist, kann RTAB-Map leicht auf einer Vielzahl von Robotersystemen eingesetzt werden.

Die Karte ist dabei graphbasiert und besteht somit aus Knoten und Verbindungen. Nachdem die Sensoren synchronisiert wurden, erstellt das sogenannte Short-Term Memory Modul (STM) einen Knoten mit den Informationen über die Pose des Roboters aus der Odometrie und verschiedene, aus den Sensordaten extrahierte Informationen wie visual Words für die Loop Closure Erkennung und ein local occupancy grid für die globale Karte. Die Verbindungen zwischen den Knoten beinhalten jeweils eine Transformation zwischen zwei Knoten. Verbindungen sind in drei Arten der Beziehung unterteilt: Nachbar, Loop Closure und Nähe. Nachbarn werden während der kurzfristigen Lokalisierung erkannt und beschreiben direkt aufeinander folgende Positionsänderungen. Anderweitig nahe Punkte und Loop Closures werden durch einen BOW Ansatz erkannt und lösen bei der Erkennung den Graphoptimierer aus. Bei dem Optimierer propagiert das System durch den gesamten aufgebauten Graph und versucht mit den neu gewonnenen Informationen Fehler zu beseitigen und so den Drift zu eliminieren. RTAB-Map gibt die gewonnenen Informationen aus dem Graph in verschiedenen Formen weiter, darunter als OctoMap, Punktwolke und als 2D Occupancy Grid.

Wichtig für die langfristige Nutzung des Systems ist der Ansatz des Speichermanagements [LM13], welcher 2013 in RTAB-Map eingebaut wurde. Ohne das Speichermanagement würde bei längeren Operationen oder großen Umgebungen die Echtzeitfähigkeit des Systems gefährdet werden, da viele der oben beschriebenen Operationen aufwändig sind. Daher ist der Speicher von RTAB-Map in

⁶<https://github.com/introlab/rtabmap>

⁷<http://wiki.ros.org/rtabmap>

einen Arbeitsspeicher (Working Memory) und einen langfristigen Speicher (Long-Term Memory) unterteilt. Der Arbeitsspeicher kann hierbei nur eine gewisse Anzahl an Knoten beinhalten, sobald ein Grenzwert an Knoten erreicht wird, werden überschüssige Knoten anhand einer Gewichtung in den langfristigen Speicher übertragen.

Loop Closures werden bei RTAB-Map durch den bag-of-visual-words (BOVW) Ansatz gefunden. Wenn ein neuer Knoten erstellt wird, werden im STM visuelle Features aus den RGB-Bildern extrahiert und in ein visuelles Vokabular aufgenommen. Dabei können jegliche Typen von Features verwendet werden, die von OpenCV unterstützt werden. Dazu gehören unter anderem Scale Invariant Feature Transform (SIFT), Speeded up robust Features (SURF), ORB und Binary Robust Independent Elementary Features (BRIEF). Die Features werden anschließend mit den Nodes im Arbeitsspeicher verglichen, um Loop Closures zu erkennen. Dabei werden die Nodes aus dem STM nicht betrachtet, da diese sehr nah aneinander liegen und viele der selben Features teilen, wodurch oft nicht existierende Loop Closures erkannt werden würden. Für den Vergleich wird ein Bayes Filter wie er in [SZ03] beschrieben wird, eingesetzt. Sobald eine Loop Closure erkannt wurde, wird eine neue Verbindung zwischen den beiden passenden Knoten erstellt. RTAB-Map wurde auf vier Datensätzen mit Ground Truth Daten Evaluier [LM19]: KITTI, TUM RGB-D, EuRoC und PR2 MIT State Center [FJKL13]. Die Ergebnisse bei der Pfadplanung waren dabei vergleichbar mit anderen State of the Art Systemen.

OpenVSLAM

OpenVSLAM [SSS19] ist ein von Sumikura et al. in 2019 vorgestelltes vSLAM-Framework und ist ebenfalls auf GitHub⁸ und als ROS-Package⁹ verfügbar. OpenVSLAM wurde so konzipiert, dass es leicht bedienbar und erweiterbar ist. Es beinhaltet einige Funktionen, die es erleichtern, OpenVSLAM in Bibliotheken Dritter einzubinden.

OpenVSLAM besteht aus einer modularen Struktur und ist grob in drei Teile aufgeteilt. Zum einen gibt es das Trackingmodul, welches einem System zur visuellen Odometrie gleicht. Dabei wird für jedes aufgenommene Frame die Pose der Kamera anhand von Keypoint-Matching und Posenoptimierung geschätzt. In diesem Modul wird auch bestimmt, ob ein neues Keyframe erstellt wird, welches

⁸<https://github.com/OpenVSLAM-Community/openvslam>

⁹https://github.com/OpenVSLAM-Community/openvslam_ros

dann an die weiteren Module weitergegeben wird. Im Kartierungsmodul wird anhand der im Trackingmodul erstellen Keyframes die Karte erstellt und durch lokales BA optimiert. Im globalen Optimierungsmodul findet die Erkennung von Loop Closures, die Optimierung des Graphs und globales BA statt.

OpenVSLAM wurde mit dem EuRoC Mav und dem KITTI Odometrie Datensatz getestet und dabei mit ORB-SLAM2 in Punkten Genauigkeit und Zeit der Lokalisierung verglichen. Im EuRoC MAV Datensatz lieferte OpenVSLAM vergleichbare Ergebnisse in der Genauigkeit wie ORB-SLAM2, schnitt in den dunklen Szenen jedoch besser ab. Die Evaluierung zeigte außerdem, dass OpenVSLAM schnellere Performance bei der Lokalisierung als ORB-SLAM2 erzielt, was laut den Autoren an der besser optimierten Version der ORB-Feature-Extraktion, sowie an dem effizienten Verhindern des Vergrößerns der lokalen Karte liegt. Beim KITTI Datensatz wurden ähnliche Ergebnisse wie beim EuRoC Datensatz erzielt.

3.2 Verfahren zur Segmentierung von 3D-Punktwolken

Nguyen und Le [NL13] beschreiben die Segmentierung von 3D-Punktwolken als einen Prozess, um Punktwolken in mehrere homogene Regionen zu unterteilen, in denen die Punkte innerhalb einer Region die selben Eigenschaften aufweisen. Dieser Prozess ist hilfreich im Feld der Robotik, speziell in den Anwendungen von intelligenten Fahrzeugen sowie der autonomen Kartierung und Navigation, wie sie in SLAM-Systemen zum Einsatz kommt. Diese Art der Segmentierung ist jedoch herausfordernd, da Daten in Form von Punktwolken eine hohe Redundanz aufweisen, die Dichte der aufgenommenen Punkte oft ungleichmäßig verteilt ist und die Punktwolke keine spezielle Struktur aufweist. Im Gegensatz zur Segmentierung von 2D-Bildern wurden Verfahren zur Segmentierung von 3D-Punktwolken bisher weniger erforscht [VTS17].

3.2.1 Methoden

Nguyen und Le [NL13] klassifizieren die Methoden zur Segmentierung von Punktwolken in fünf Kategorien, die im folgenden vorgestellt werden.

Kantenbasierte Methoden

Bei kantenbasierten Methoden (Edge based methods) wird versucht, Objekte und Regionen anhand der Eigenschaften ihrer Umrisse, also deren Kanten, zu erkennen. Oft werden dafür die Normalenvektoren der Punkte betrachtet und anhand der Richtungsänderung der Normalenvektoren Kanten erkannt. Kantenbasierte Methoden ermöglichen eine schnelle Segmentierung, sind jedoch sehr anfällig für Störungen wie Rauschen und ungleichmäßiger dichte der Punkte, worunter die Genauigkeit der Segmentierung leidet. Beide dieser Probleme kommen zudem häufig in Punktwolken vor.

Regionsbasierte Methoden

Regionsbasierte Methoden (Region based methods) verwenden die Informationen umliegender Punkte, um die Punkte mit ähnlichen Eigenschaften in isolierte Regionen zu Gruppieren. Gleichzeitig werden dabei Unterschiede in den Eigenschaften verschiedener Regionen gesucht, um diese voneinander abgrenzen zu können. Regionsbasierte Methoden sind robuster gegenüber Rauschen als kantenbasierte Methoden, haben dabei jedoch Probleme entweder zu viel oder zu wenig zu Segmentieren und die Regionsgrenzen genau zu bestimmen. Regionsbasierte Methoden werden in die Kategorien *seeded* (ausgesät) und *unseeded* (ungesät) unterteilt.

Seeded-Region Methode: Hier wird der Segmentierungsprozess gestartet, indem eine Anzahl an Punkten gewählt, oder *ausgesät*, wird. In diesem ersten Schritt müssen die Punkte anhand verschiedener Kriterien wie der Krümmung, dem Normalenvektor oder der Distanz zu einer bestimmten Fläche ausgewählt werden. Anhand der gewählten Punkte wächst jede Region, indem benachbarte Punkte hinzugefügt werden, wenn sie bestimmte Kriterien erfüllen. Diese Methode hat die Nachteile, dass sie empfindlich gegenüber Rauschen und zeitaufwändig ist.
Unseeded-Region Methode: Hierbei werden zuerst alle Punkte in eine gemeinsame Region gruppiert. Anschließend wird diese Region so lange in weitere Regionen unterteilt, bis eine festgelegte Schwelle erreicht wurde. Die Schwierigkeit liegt dabei in der Entscheidung, wo und wann eine Region unterteilt werden soll. Zusätzlich werden hier viele Informationen im Voraus benötigt, so wie die Art der zu segmentierenden Objekte oder die gewünschte Anzahl der Regionen.

Attributbasierte Methoden

Attributbasierte Methoden sind robuste Herangehensweisen zur Segmentierung von Punktwolken anhand der Attribute von Clustern der Punktfolke. Die Verfah-

ren laufen dabei in zwei Schritten ab. Im ersten Schritt werden die Attribute wie z.B. die Texturen der Oberfläche berechnet. Im zweiten Schritt wird die Punktwolke anhand der Attribute in Cluster aufgeteilt. Die Methode zur Bestimmung der Cluster beinhaltet dabei Flexibilität in der Verwendung der räumlichen Beziehungen zwischen den Punkten und den Attributen. Attributbasierte Methoden stellen ein robustes Verfahren mit genauen Ergebnissen zur Gruppierung von Punkten in homogene Regionen dar. Die Methoden verlassen sich dabei jedoch stark auf die Definition, was als benachbarter Punkt gilt, sowie auf die dichte der Punktwolke. Außerdem sind diese Verfahren zeitaufwändig wenn mehrdimensionale Attribute bei einer großen Anzahl an Punkten verwendet werden.

Modellbasierte Methoden

Modellbasierte Methoden nutzen primitive Formen wie Kugeln, Kegel, Zylinder und Flächen um Punkte zu gruppieren. Dabei werden die Punkte, die die selbe mathematische Repräsentation aufweisen, in ein Segment gruppiert. Ein bekannter Algorithmus, der diese Methode anwendet, ist Random Sample Consensus (RANSAC) [FB81]. RANSAC gilt als State of the Art um mathematische Merkmale wie Linien, Kreise, etc. in Daten zu erkennen. Modellbasierte Methoden sind durch ihren reinen mathematischen Hintergrund schnell und robust, sind jedoch ungenau, wenn sie mit verschiedenen Quellen von Punktwolken arbeiten sollen.

Graphenbasierte Methoden

Graphenbasierte Methoden interpretieren Punktwolken in Form eines Graphen. Ein einfaches Modell hierfür wäre, dass jeder Punkt der Punktwolke einen Knoten in einem Graphen repräsentiert. Jeder benachbarte Punkt wird dabei über Verbindungen verknüpft. Sie erlangten durch ihre Genauigkeit und Effizienz Beliebtheit im Einsatz bei Roboterapplikationen. Graphbasierte Ansätze haben zusätzlich den Vorteil, dass sie auch Daten, die Rauschen beinhalten oder ungleichmäßig verteilt sind, mit genaueren Ergebnissen Segmentieren können. Bei dieser Art von Daten können die Systeme jedoch nicht in Echtzeit operieren oder benötigen weitere Daten von anderen Sensoren.

4 Methodik

4.1 Simulation nicht-stationärer Umgebungen für mobile Roboter

Im Zuge dieser Arbeit wurde eine Roboter Simulationsumgebung mit der Unity Engine, aufbauend auf dem Unity Robotics Hub¹ erstellt, um vSLAM und LT-SLAM Systeme für semi-statische Umgebungen zu entwickeln und zu testen. In den folgenden Kapiteln wird die Auswahl, die Entwicklung und die Evaluierung dieser Simulation beschrieben.

4.1.1 Auswahl der Simulationsumgebung

Im Zuge der Recherche dieser Arbeit wurden mehrere Methoden zur Evaluierung der Ergebnisse der Segmentierung und Objekterkennung analysiert. Die ursprüngliche Auswahl an Evaluierungsmethoden bestand aus dem Gazebo Simulator, ROS-Bags und aus realen Robotern. Durch den Fokus der Arbeit auf LT-SLAM entpuppten sich diese drei Methoden jedoch als suboptimal.

Der Gazebo Simulator eignet sich, wie in Kapitel 2.4.1 beschrieben, vor allem für statische Umgebungen und Roboter, die primär mit Laserscannern ausgestattet sind. Vor allem das für die Simulation von LT-SLAM notwendige Einbinden von semi-statischen und dynamischen Objekten wird von Gazebo nur bedingt unterstützt.

Bei Bags besteht die Problematik in der Verfügbarkeit. Es existieren wenige Datensätze für mobile Roboter in nicht-stationären Umgebungen, insbesondere nicht mit RGB-D Daten in Verbindung mit Ground Truth. Das Erstellen von eigenen Bags mit ausreichender Ground Truth ist ohne spezielle Hardware ebenfalls nicht ohne weiteres möglich.

Das Testen von LT-SLAM Verfahren während dem Entwicklungsprozess auf realen Robotern ist ebenfalls nicht durchführbar. So muss jede neue Version der Software auf den Roboter geladen werden und dieser durch die Umgebung bewegt werden. Dynamische und semi-statische Objekte müssten dabei manuell

¹<https://github.com/Unity-Technologies/Unity-Robotics-Hub>

bewegt werden und jeder Durchlauf wäre zeitaufwändig, würde keine Ground Truth beinhalten und wäre streng genommen nicht mit der exakt selben Umgebung wiederholbar.

Aufgrund dieser Probleme wurden Alternativen zu diesen drei für statischen SLAM gängige Methoden gesucht. Vielversprechend wirkten dabei Robotersimulationen, die auf der in Kapitel 2.4.2 beschriebenen Unity-Engine aufbauten. Um die passende Simulationsumgebung zu identifizieren, wurden im voraus acht grundlegende Kriterien an eine Simulationsumgebung aufgestellt, die in einem paarweisen Vergleich relativ zueinander bewertet wurden. Die daraus resultierenden Gewichtungen wurden darauf in einer Nutzwertanalyse zur Bewertung verschiedener Simulationen verwendet. Die Nutzwertanalyse in Kombination mit einem paarweisen Vergleich ist nach dem Organisationshandbuch des Bundesverwaltungsamts² eine der häufigst genutzten qualitativen Bewertungsmethoden und wurde daher ausgewählt. Die Kriterien wurden dabei im Gespräch mit Experten beim Fraunhofer IPA sowie anhand der Literatur [HLN12], [RSF13] identifiziert. Die Kriterien beziehen sich dabei mehr auf die technischen Eigenschaften und den Funktionsumfang der Simulation. Im folgenden werden die Ausgewählten Kriterien vorgestellt und deren Relevanz begründet:

- i. **Quality:** Wie in Kapitel 2.4 beschrieben, sollte die Realität sowie die Sensorik und Aktorik ausreichend simuliert sein. Außerdem darf der Roboter nur über die simulierten Akteuren und Sensoren mit der Umgebung interagieren, wobei die Schnittstelle für die Datenübertragung die selbe sein muss, wie für einen realen Roboter. Zur Qualität gehören auch die visuelle Qualität des Systems, was beispielsweise für simulierte Kameras relevant ist.
- ii. **Overhead:** Die Simulation soll eine gute Balance zwischen einem hohen Funktionsumfang und einer gewissen Kompaktheit wahren. So sollten im besten Fall die grundlegenden Funktionen vorliegen, wobei weitere Funktionen durch weitere Module im Nachhinein installiert werden können.
- iii. **Performance:** Die Simulation soll performant sein. Der interaktive Teil soll im besten Fall auf herkömmlichen Computern eine Bildrate von mindestens 30 Frames pro Sekunde (FPS) aufweisen. Die Übertragungsrate der Informationen, die der simulierte Roboter sendet und empfängt, sollte ebenfalls ausreichend sein. Die Übertragungsrate von Sensoren hängt in der Regel vom Sensortyp ab. So überträgt die Microsoft Kinect Daten mit einer Frequenz von 30 Hz [HLN12], Transformationen in ROS werden üblicherweise mit 100 Hz gesendet.

²https://www.orghandbuch.de/OHB/DE/Organisationshandbuch/6_MethodenTechniken/65_Wirtschaftlichkeitsuntersuchung/652_Qualitative/qualitative-node.html

- iv. **Import & Export:** Die Simulation sollte den Import von Robotermodellen unterstützen. Dies ist vor allem für die korrekte Anordnung der Aktoren und Sensoren des Roboters wichtig. Übliche Formate zur Beschreibung sind dabei das Universal Robot Description Format (URDF) und das Simulation Description Format (SDF). Auch die Funktion zum Importieren und Exportieren von Umgebungen und anderen Objekten sind hierbei relevant.
- v. **Sensors:** Die Simulation sollte die gängigen Sensortypen aus Kapitel 2.1.1 unterstützen. Dazu gehören unter anderem Kameras, Tiefenkameras sowie 2D- und 3D Laserscanner. Die Sensoren sollten realistische Daten liefern können, dazu kann beispielsweise Rauschen simuliert werden. Hierzu gehört auch das veröffentlichen der Transforms der Roboterkomponenten.
- vi. **Little Complexity:** Das System sollte nicht zu komplex sein. Dazu gehört auch die Verständlichkeit und Erweiterbarkeit des Codes, sowie vorhandene Dokumentation und Anleitungen.
- vii. **Usability:** Die Simulation sollte leicht zu bedienen sein. Die betrachteten Hauptpunkte sind hier das User Interface (UI) und die User Experience (UX). Funktionen sollten beispielsweise leicht zu finden sein und Menüs sollten sich gut bedienen lassen. Auch die Steuerung der Kamera und des Roboters in der Simulationsansicht sollte gängigen Konventionen folgen, damit kein neues Steuerungsschema vom Benutzer gelernt werden muss.

In Tabelle 4.1 ist der paarweise Vergleich der oben genannten Simulationskriterien abgebildet. Dabei wird jede Kombination an Kriterien miteinander verglichen und einen Wert zwischen Null und Zwei zugewiesen. Der Wert zeigt ausgehend von der linken Spalte an, welcher der beiden Kriterien als wichtiger erachtet wird. So wird beispielsweise beim ersten Wert der ersten Zeile der Qualität gegenüber dem Overhead zwei Punkte zugewiesen, da die Qualität im relativen Vergleich als wichtiger erachtet wird. In jeder Zeile ergibt sich somit eine Summe aus den Werten, welche in Relation mit der gesamten Summe gesetzt wird, woraus eine Gewichtung errechnet werden kann. Diese Gewichtung kommt anschließend in der Nutzwertanalyse zum Einsatz.

Für die Nutzwertanalyse wurden im voraus vier bestehende Frameworks, die entsprechend für LT-SLAM Systeme angepasst werden können, identifiziert und getestet. Diese bestehen aus der Microsoft Aerial Informatics and Robotics Platform, dem NVIDIA Isaac SDK, ROS# und dem Unity Robotics Hub. Zusätzlich wurden diese vier Systeme mit dem Gazebo Simulator verglichen.

Rating 0-2	Quality	Overhead	Performance	Import & Export	Sensors	Little Complexity	Usability	Summe	Gewichtung
Quality	x	2	1	2	1	2	1	9	21%
Overhead	0	x	0	0	0	1	0	1	2%
Performance	1	2	x	1	1	2	1	8	19%
Import & Export	0	2	1	x	0	1	1	5	12%
Sensors	1	2	1	2	x	2	1	9	21%
Little Complexity	0	1	0	1	0	x	1	3	7%
Usability	1	2	1	1	1	1	x	7	17%
								42	100%

Tabelle 4.1: Paarweiser Vergleich der Kriterien, die eine Simulation erfüllen sollte.

4.1.2 Nutzwertanalyse

Die Nutzwertanalyse bewertet die oben genannten Systeme anhand eines Punktesystems mit Werten zwischen Null und Zehn in Kombination mit der errechneten Gewichtung aus dem paarweisen Vergleich. Für jedes Kriterium wird eine Bewertung gegeben, welche mit der entsprechenden Gewichtung multipliziert wird. Die Punkte werden anschließend für jedes System aufsummiert, womit sich die Endgültige Bewertung des jeweiligen Systems ergibt. In Tabelle 4.2 ist die Nutzwertanalyse, wie sie durchgeführt wurde, abgebildet. Die höchste Punktzahl erzielte hier der Unity Robotics Hub mit einer Wertung von 8,74, gefolgt von ISAAC mit einer Wertung von 7,12.

Nr.	Kriterium	Gewichtung	AirSim		ISAAC		ROS#		RoboticsHub		Gazebo	
			Bewertung	Punkte	Bewertung	Punkte	Bewertung	Punkte	Bewertung	Punkte	Bewertung	Punkte
1	Quality	0,21	5	1,07	10	2,14	8	1,71	8	1,71	5	1,07
2	Overhead	0,02	7	0,17	3	0,07	0	0,00	10	0,24	8	0,19
3	Performance	0,19	10	1,90	5	0,95	0	0,00	10	1,90	8	1,52
4	Import & Export	0,12	0	0,00	10	1,19	10	1,19	10	1,19	7	0,83
5	Sensors	0,21	2	0,43	9	1,93	8	1,71	8	1,71	10	2,14
6	Little Complexity	0,07	9	0,64	0	0,00	7	0,50	9	0,64	3	0,21
7	Usability	0,17	2	0,33	5	0,83	8	1,33	8	1,33	0	0,00
Summe			4,55		7,12		6,45		8,74		5,98	

Tabelle 4.2: Die für die Simulation durchgeführte Nutzwertanalyse.

Im folgenden werden die vier Systeme vorgestellt und anhand der Kriterien beschrieben.

Microsoft Aerial Informatics and Robotics Platform

Die Microsoft Aerial Informatics and Robotics Platform³ (AirSim) ist eine Open-Source Roboter Simulationsplattform und komplett auf GitHub verfügbar⁴. AirSims primärer Nutzen ist die Simulation autonomer Drohnen und Fahrzeugen in Outdoor-Szenarien in der Unreal Engine. Zusätzlich existiert ein prototypischer Ansatz einer Simulation aufbauend auf Unity. AirSim beinhaltet zusätzliche Python-Skripte, die als APIs zur Datenerfassung und zur externen Kontrolle der Simulation dienen. Die visuelle Qualität von AirSim ist dabei abhängig von den entsprechenden Umgebungen der Simulation in den Game-Engines und kann somit realistische Grafik, wie sie die Unreal- und Unity Engine anbieten, erreichen. Auch die Sensorik und die Aktorik der Fahrzeuge sind mit hoher Qualität simuliert, so können die Kameras und Räder in der Autosimulation einzeln angesteuert werden. Durch den Fokus auf Kraftfahrzeuge und Drohnen sind jedoch nicht alle in der Robotik gängigen Sensoren und Aktoren in Airsim verfügbar. Die Standardkonfiguration von AirSim besteht entweder aus einem PKW oder einer Drohne, jeweils mit einer Tiefenkamera und Farbkamera ausgestattet, sowie einer Ansicht, die die Umgebung in gewisse Segmente unterteilt. In Abbildung 4.1 ist ein Ausschnitt aus der AirSim Unity Simulation für PKW in der inkludierten Standardumgebung zu sehen. Im unteren Teil des Bilds sind die drei Ansichten der Kameras wie sie in der Simulation dargestellt sind, zu sehen. Bei der Simulation für PKWs in Unity besteht die Standardumgebung aus einer großen Fläche mit wenigen Hindernissen und einer einfachen Mauer, die die Umgebung einzäunt. Die Performance innerhalb dieser Umgebung war durch die wenigen Merkmale sehr gut, die Simulation der Sensoren verlangsamte den Echtzeitmodus dabei nicht. Zur Simulation von mobilen Robotern ist AirSim jedoch weniger geeignet, da keine Import und Export-Funktion von Robotermustern verfügbar ist und wenige Typen von Sensoren und Aktoren unterstützt werden. Zusätzlich war der getestete Unity Prototyp nicht sehr Usable. Die Menüs waren sehr minimal und der Hauptteil der Steuerung lief über Python Skripte.

³<https://www.microsoft.com/en-us/research/project/aerial-informatics-robotics-platform/>

⁴<https://github.com/microsoft/AirSim>



Abbildung 4.1: Ausschnitt aus der AirSim Unity Simulation für PKWs.

NVIDIA Isaac SDK

Das NVIDIA Isaac SDK⁵ (Isaac) ist ein Toolkit zur Roboterentwicklung, welches vor allem für kommerzielle, auf Künstliche Intelligenz (KI) gestützte Roboter entwickelt wurde. Die Robotersimulation, genannt Isaac Sim, ist nur ein kleiner Teil des kompletten Isaac Stacks. Das Projekt ist bereits sehr reif und beinhaltet eine große Bandbreite an Funktionen und Anwendungsmöglichkeiten. Die Qualität der Simulation erfüllt alle Kriterien. So wird die Sensorik und die Aktorik der Roboter hinreichend simuliert. ISAAC unterstützt ebenfalls die Anbindung an ROS über ROS Bridge⁶. Die visuelle Qualität der Standardumgebungen ist sehr hoch. Die Umgebungen bestehen aus verschiedenen großen Warenhäusern, inklusive einiger Assets wie Kisten und Regalen. Objekte wie Lampen, die oft in einer Szene vorkommen, haben verschiedene Versionen mit unterschiedlicher Anzahl von Polygonen und unterschiedlicher Auflösung von Texturen, wodurch Performance der Simulation optimiert wird. ISAAC setzt zusätzlich auf verschiedene Assets aus dem Unity Asset Store wie Adobe Substance 3D⁷ für hochauflösende, realistische Texturen. Außerdem beinhalten die Szenen Post-Processing, womit die Umgebungen realistischer für visuelle Systeme wirken. In Abbildung 4.2 ist ein Ausschnitt aus dem Nvidia Isaac Sim aus der Szene *small_warehouse* abgebildet.

Jedoch ist auch der Overhead von Isaac entsprechend groß. Um Isaac Sim zu

⁵<https://developer.nvidia.com/isaac-sdk>

⁶http://wiki.ros.org/rosbridge_suite

⁷<https://substance3d.adobe.com/plugins/substance-in-unity/>

verwenden, um eigene Applikationen zu testen, muss die komplette Isaac Suite installiert werden. Ein weiteres Problem war die Performance. Die Workstation, auf der die Tests durchgeführt wurden, ist mit einem AMD Ryzen Threadripper 2990WX Prozessor, 64GB Arbeitsspeicher und einer Nvidia GeForce RTX 2080 Grafikkarte ausgestattet. Die Testszene *small_Warehouse* wies dabei durchschnittlich 10 FPS auf. Der Grund dafür ist der Fokus von ISAAC auf die Entwicklung von KI-Anwendungen und der Optimierung der Systeme auf spezielle Chips wie dem NVIDIA Jetson. So ist nach den Anforderungen für Isaac Sim 2021.1 beispielsweise die NVIDIA RTX2080 die Mindestanforderung, jedoch nicht empfohlen, was die niedrige Bildrate erklären könnte. ISAAC unterstützt dafür alle gängigen Sensorarten mit einem leichten Fokus auf RGB-D Kameras und beinhaltet Werkzeuge für das einfache Importieren und Exportieren von Robotmodellen. Die ISAAC Suite ist jedoch sehr komplex. Vor allem wenn eigene Funktionalitäten eingebunden werden sollen, ist die Größe und Komplexität des Systems ein großes Hindernis. Zusätzlich ist in den vorgegebenen Fällen zur Verwendung des Systems die Usability sehr hoch und die Demos sind leicht zu bedienen. Jedoch ist das Erstellen von eigenen Szenarien und Use Cases in ISAAC komplex und schwer verständlich, da neben Unity auch einige andere Programme verwendet werden müssen. Insgesamt ist ISAAC ein mächtiges Werkzeug zur Entwicklung von Roboterapplikationen, vor allem, wenn die notwendige Hardware und Einarbeitungszeit vorhanden ist. Im Zuge des Anwendungsfalls in dieser Arbeit ist ISAAC mit 7,12 Punkten in der Nutzwertanalyse auf dem zweiten Platz.

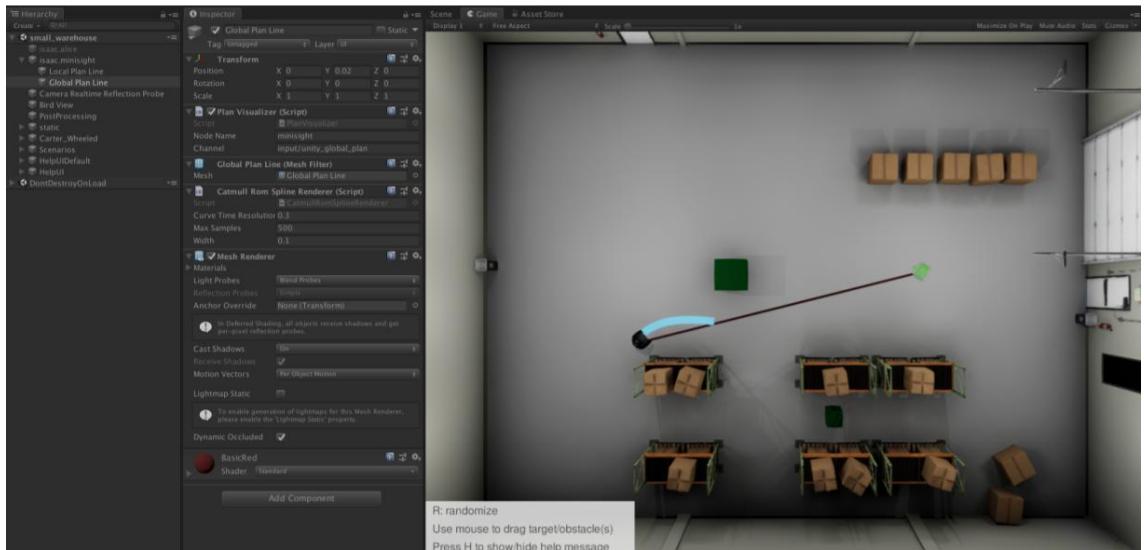


Abbildung 4.2: Ausschnitt aus dem Nvidia Isaac Sim aus der Szene *small_Warehouse*.

ROS#

ROS#⁸ ist eine von Siemens entwickelte Sammlung von Open Source Software Bibliotheken und Werkzeugen um ROS mit .NET Anwendungen zu verknüpfen. Der Fokus liegt dabei auf der Anwendung mit Unity als Simulation. ROS# beinhaltet eine einfache Demo Szene mit einem Turtlebot3 als Roboter, die in Abbildung 4.3 dargestellt ist. Die Szene enthält alle notwendigen Scripte, um den Roboter mit der Maus in Unity zu steuern und die Daten seiner Farbkamera an ROS zu senden. Die Qualität der Simulation ist ähnlich wie bei AirSim und ISAAC sehr hoch und ist rein durch die Funktionen der Unity Engine limitiert. Das Senden und Empfangen von Daten über die ROS Bridge ist jedoch nicht für alle Datentypen vorhanden und muss teils selbst implementiert werden. ROS# enthält nur die notwendigsten Funktionen und ist so leicht zu installieren und in ein existierendes Projekt einzubinden. Die größte Schwäche von ROS# ist die Performance. Jede Message, die an ROS über ROS Bridge gesendet wird, muss zuvor in Unity in das JSON-Format konvertiert werden. Dazu werden die Daten von ROS# jeweils in mehrere Strings umgewandelt, bevor sie als JSON Datei serialisiert werden. Jedoch ist die häufige Konkatenation von Strings in Unity sehr schlecht für die Performance und für den Garbage Collector. Die Unity Szene wird bei einer Auflösung von 640x480 Pixeln nur mit durchschnittlich 15 FPS gerendert, auch wenn sehr wenige Objekte in Sichtweite sind. Zusätzlich ist die Serialisierung der Messages zu langsam, wodurch einige Daten verloren gehen, oder in falscher Reihenfolge in ROS ankommen. Da ROS Messages im besten Fall mehrmals pro Sekunde published werden, kann Unity außerdem nicht schnell genug die alten Daten im Arbeitsspeicher löschen. So steigt der Speicherverbrauch von Unity im Arbeitsspeicher von Anfangs 2GiB bis hin zu 20GiB, woraufhin die Simulation unerwartet abbricht und aufgrund fehlendem Speicher automatisch geschlossen wird. In der Unity Dokumentation wird bei der Beschreibung von automatischem Speichermanagement⁹ ebenfalls vor der übermäßigen Verwendung von Strings gewarnt. Der Import und Export von Robotmodellen funktioniert hingegen sehr gut. ROS# beinhaltet die Funktion, URDF Dateien in Unity einzubinden, worauf automatisch die Hierarchie des Roboters in Unity-Objekte umgewandelt wird. Lediglich die Verbindung der Komponenten mit der ROS Bridge muss von Hand erledigt werden. ROS# beinhaltet wenige vorgefertigte Sensoren, bietet aber durch ROS Bridge die Unterstützung für alle Arten von ROS Messages. Durch die niedrige Komplexität und die leicht zu verändernde Scripte können recht leicht eigene Sensoren eingebunden werden. Die Usability von ROS# ist ebenfalls hoch. Die Parameter der Scripte sind in der Regel selbsterklärend und es existieren einige Tutorials auf der ROS# GitHub Seite.

⁸<https://github.com/siemens/ros-sharp>

⁹<https://docs.unity3d.com/Manual/UnderstandingAutomaticMemoryManagement.html>

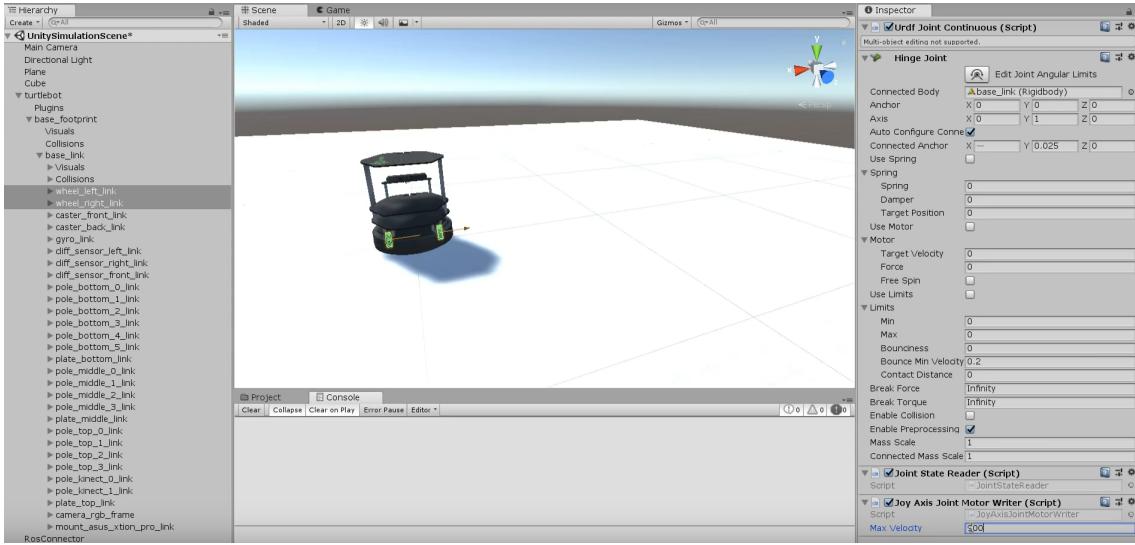


Abbildung 4.3: Ausschnitt aus der ROS# Demo Szene in Unity. Bei dem simulierten Roboter handelt es sich um den Turtlebot3.

Unity Robotics Hub

Der Unity Robotics Hub¹⁰ ist eine von Unity Technologies entwickelte Simulationsumgebung für die Unity Engine. Die von Unity gelisteten Gründe, ihre Engine als Roboter Entwicklungsplattform zu verwenden, sind das schnelle Erstellen von Prototypen durch den Unity Asset Store, eine genaue Physiksimulation durch PhysX sowie das Erstellen komplexer und realistischer Umgebungen¹¹. Der Unity Robotics Hub baut teilweise auf ROS# auf. So wurde der URDF Importer und der TCP Connector von der ROS# Repository geforked und erweitert¹². Dabei wurde die Art der Serialisierung der messages geändert, damit nun nicht mehr das problematische JSON Format verwendet wird. Stattdessen werden die Daten hier in Unity direkt in das von ROS verwendete Format zur Serialisierung umgewandelt. Somit werden die Probleme von JSON in Unity umgangen sowie die Performance durch das eliminieren eines Schritts zur Umwandlung der Daten verbessert. Außerdem basiert die Kommunikation mit ROS auf TCP, wodurch es unmöglich ist, dass die Nachrichten in falscher Reihenfolge auftreten können. Die Qualität der Simulation ist ähnlich wie bei ROS# von Unity abhängig und entsprechend hoch. Roboter interagieren hier ebenfalls nur durch die erlaubten Schnittstellen in Form von Sensoren und Aktoren mit der Umgebung. In Abbildung 4.4 ist ein Ausschnitt einer Szene des Unity Robotics Hubs dargestellt. Die Umgebung ist dabei eine abgeänderte Version des *Big_Warehouse* von ISAAC, bei dem Robo-

¹⁰<https://github.com/Unity-Technologies/Unity-Robotics-Hub>

¹¹<https://unity.com/solutions/automotive-transportation-manufacturing/robotics>

¹²<https://github.com/Unity-Technologies/Unity-Robotics-Hub/blob/main/faq.md#how-does-your-unity-integration-compare-to-ros>

ter handelt es sich um eine Basis des Care-O-bot 4. Der Unity Robotics Hub ist ähnlich wie ROS# strukturiert, wodurch hier ebenfalls ein geringer Overhead existiert. Die Performance ist dafür im Gegensatz zu ROS# sehr hoch. So erreicht die Simulation bei einer Auflösung von 640x480 Pixeln eine Bildrate von 120 FPS und empfängt und überträgt Daten an ROS mit einer durchschnittlichen Frequenz von 44Hz. Der Verbrauch an Arbeitsspeicher beträgt dabei konstant 1GiB. Das Importieren von Robotermodellen funktioniert wie bei ROS# sehr gut, da der Großteil der Funktion von ROS# übernommen wurde. Die Sensoren sind ebenfalls ähnlich wie ROS# nicht komplett implementiert. Hier müssen anhand der vorhandenen Typen der Messages selbst die Sensoren implementiert werden, was auch durch die niedrige Komplexität und den verständlichen Code wenige Probleme bereitet. Die Verknüpfung zwischen Unity und ROS ist durch das automatische Generieren der notwendigen Objekte in Unity vereinfacht. Insgesamt schnitt der Unity Robotics Hub mit 8,74 Punkten in der Nutzwertanalyse am besten ab und wurde daher für den in dieser Arbeit betrachteten Use-Case ausgewählt.

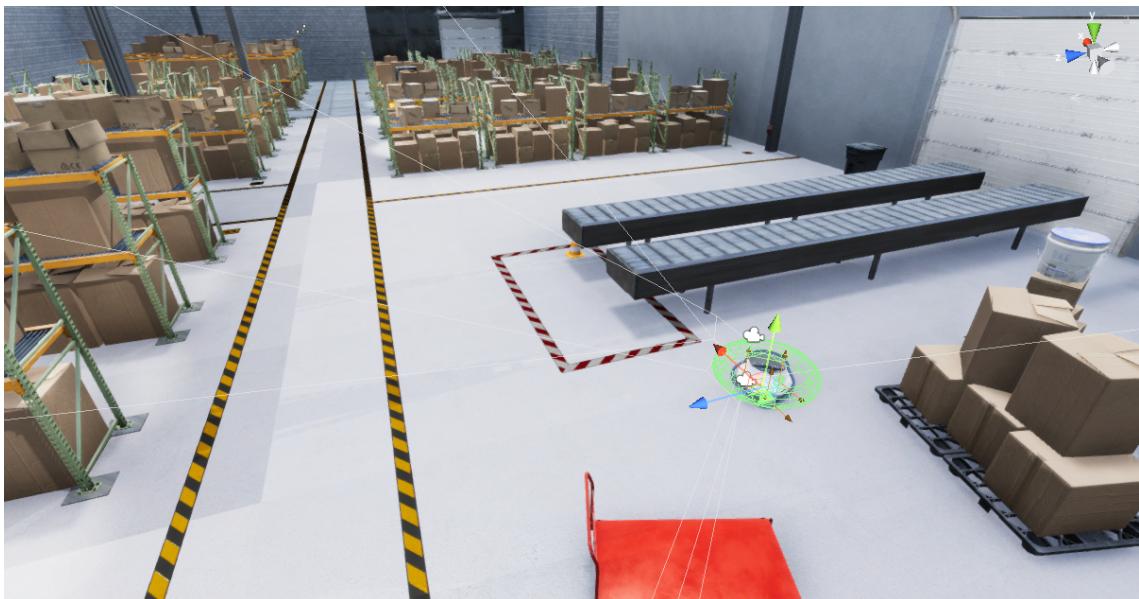


Abbildung 4.4: Ausschnitt aus einer Szene mit dem Unity Robotics Hub. Bei der Umgebung handelt es sich um eine abgeänderte Version des *Big_Warehouse* von ISAAC. Bei dem Roboter handelt es sich um die Basis des Care-O-bot 4.

Gazebo

Als Repräsentant für herkömmliche Robotersimulationen wurde der in Kapitel 2.4.1 beschriebene Gazebo Simulator mit den auf Unity aufbauenden Simulationen verglichen. Die größten Nachteile von Gazebo liegen hier bei der visuellen Qualität der Simulation, der Komplexität und der Usability. Aufbauend auf der Game Engine Unity bieten die anderen Simulatoren im Gegensatz zu Gazebo das einfache Einbinden von hoch qualitativen Objekten und Texturen sowie die Möglichkeit, Post-Processing Effekte zu verwenden und eigene Shader einzubinden. Zusätzlich beinhaltet Gazebo zwar viele Funktionalitäten, viele davon sind jedoch wenig selbsterklärend oder schwer und kompliziert zu verwenden. Hinzu kommt die Schwierigkeit, dynamische und semi-statische Objekte in Gazebo einzubinden. Insgesamt ist Gazebo ein mächtiger Simulator, wird für den Use Case von nicht-stationären Umgebungen und in der Usability jedoch von den meisten der Unity-basierten Simulatoren übertroffen.

4.1.3 Aufbau der Unity-Simulation

Die im Zuge dieser Arbeit erstelle Simulation basiert, wie im vorigen Kapitel beschrieben, auf der Unity Engine und dem Unity Robotics Hub. Im folgenden wird der Aufbau der Simulation und deren Komponenten beschrieben.

Grundlegende Einstellungen

Um den Unity Robotics Hub in Unity einzubinden, muss lediglich das GitHub Repository des Unity Robotics Hub¹³ über den Unity Package Manager eingebunden werden. Nach der Installation verfügt Unity's Menüleiste über den Punkt *Robotics* mit zwei Unterpunkten. Über den ersten Unterpunkt *ROS Settings* werden die Parameter für die Verbindung mit ROS gesetzt. In Abbildung 4.5 ist das Fenster *ROS Settings* dargestellt. Hier kann eingestellt werden, ob sich Unity direkt beim Start mit ROS verbinden soll, über welche IP Adresse und welchen Port der ROS-Server verfügt und wie sich das Programm bei einer unerwarteten Trennung der Verbindung verhalten soll. Auf der Seite von ROS muss vor Start der Simulation der Server zur TCP-Verbindung gestartet werden. Die Launch-Datei dafür ist in Codeausschnitt 1 dargestellt. Wichtig ist dabei, dass die IP-Adresse und der Port in ROS und Unity übereinstimmen. Außerdem muss hier der Pfad für das in Unity verwendete Robotermodell im URDF-Format gesetzt werden, damit die Darstellung des Roboters und dessen Transforms in Unity und in ROS übereinstimmen.

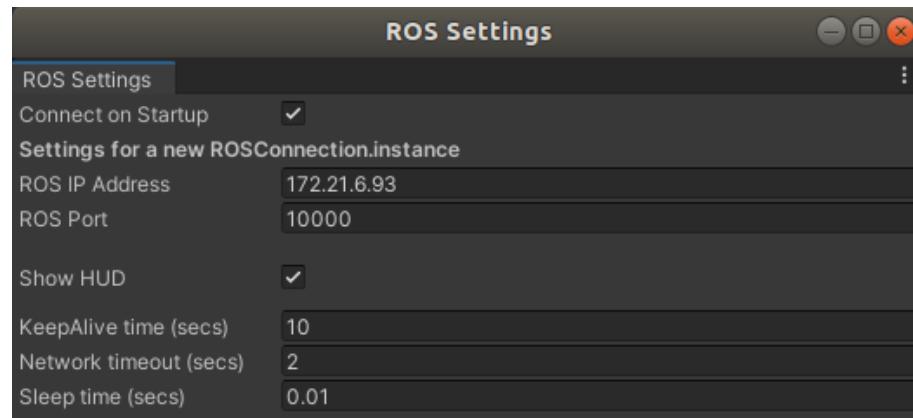


Abbildung 4.5: Einstellungen für die Verbindung von Unity und ROS.

Der Zweite Unterpunkt mit dem Titel *Generate ROS Messages...* wird verwendet, um Unity-Scripte beliebiger Message-Arten aus ROS zu generieren. Dazu muss,

¹³<https://github.com/Unity-Technologies/Unity-Robotics-Hub>

wie in Abbildung 4.6 dargestellt, der Systempfad zu einer ROS-Installation angegeben werden. Hier können anschließend die benötigten Messages aus einer Auswahl von allen in ROS installierten Packages ausgewählt werden. So sind für dieses Projekt beispielsweise unter anderem die Messages CameraInfo.msg und CompressedImage.msg notwendig. Durch das Klicken auf einen *Build msg* Button, werden die entsprechenden Scripte generiert. Dabei wird jeder Typ von Message in Form einer C# Klasse mit den zugehörigen Variablen und Funktionen erstellt.

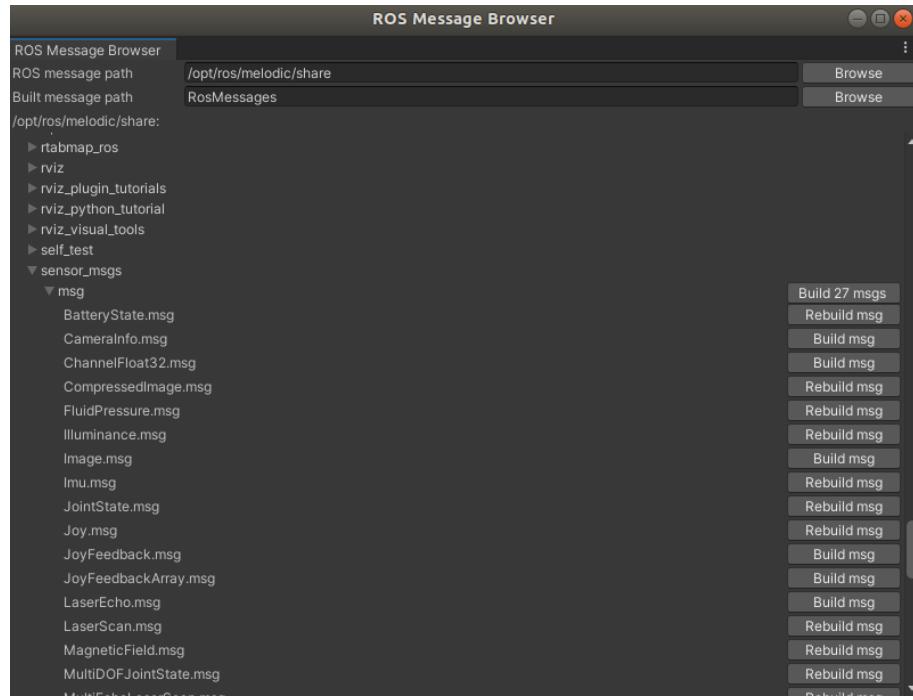


Abbildung 4.6: Fenster zur Generierung von Klassen von ROS Messages in Unity.

Umgebung

Zu Testzwecken wurde die Umgebung *big_warehouse* aus dem Unity Simulator von ISAAC verwendet. Diese Umgebung ist vor allem zur Evaluierung von Robotersystemen für den Indoor Nutzen geeignet und beinhaltet einige nützliche Arten von Objekten und Features. In Abbildung 4.7 sind die für LT-SLAM wichtigsten Objekte dargestellt. Oben links ist ein Gang zwischen Regalen voll mit Pappkartons zu sehen. Die Umgebung besteht primär aus diesen Reihen, die Kartons sind dabei teilweise unterschiedlich angeordnet. Oben rechts ist eine Säule mit einem angebrachten Feuerlöscher und einer Mülltonne daneben abgebildet. Auf dem Boden sind zusätzlich gelb-schwarz gestreifte Linien aufgezeichnet, die verschiedene Areale in der Halle kennzeichnen. Säulen kommen häufig in Hallen vor

und bieten oft gute Möglichkeiten für einen Roboter, sich zu orientieren. Unten links im Bild ist ein Gabelstapler zu sehen. Dieser bietet der Simulation die Möglichkeit eines einfachen nicht-stationären Objekts. Ein Gabelstapler kann sich bewegen, kann jedoch auch über längere Zeit stationär sein. Zusätzlich muss ein Gabelstapler im Gegensatz zu beispielsweise Personen wenig animiert sein, um ein realistisches Objekt für die Simulation zu bieten. Unten rechts sind Paletten abgebildet. Diese können ebenfalls als nicht-stationäre Objekte dienen, sind jedoch auch praktisch wenn ein Roboter beispielsweise Objekte auf Paletten abstellen oder von ihnen aufheben soll.



Abbildung 4.7: Objekte, die in der Simulation zu finden sind. Von oben Links: Regale mit Kisten, eine Säule mit Feuerlöscher und Mülltonne, ein Gabelstapler und Paletten.

Roboter

Bei dem Roboter, der zu Testzwecken in der Simulation eingesetzt wurde, handelt es sich um die Basis des Care-O-bot 4 [KFS⁺15], welche in Abbildung 4.8

dargestellt ist. Die Basis ist zusätzlich mit zwei Intel RealSense D435 Tiefenkameras ausgestattet. Eine der Kameras ist unten an der Basis angebracht und ist leicht nach oben geneigt. Die zweite Kamera ist über der Basis angebracht und ist leicht nach unten geneigt. Die Care-O-bot Basis ist mit drei drehbaren Rädern ausgestattet und verfügt so über eine hohe Manövriergängigkeit. Die Positionen der Kameras sind in Abbildung 4.8 auf der linken Seite zu sehen, die Anordnung der Räder auf der rechten Seite. In Abbildung A1 ist ein Ausschnitt des TF-Tree des Care-O-bot 4 in der Simulation dargestellt. Die Hierarchie startet dabei bei dem odom Transform, gefolgt von base_footprint und base_link. An base_link sind die Sensoren des Care-O-bot angebracht, dazu gehören beispielsweise die Laserscanner und Tiefenkameras. Des Weiteren befindet sich unter base_link der base_chassis_link, welcher als Überknoten für den Ladepunkt und die Räder des Care-O-bots dient.



Abbildung 4.8: Die in der Simulation verwendete Care-O-bot 4 Base mit zwei Kameras.

Aufbau der Simulation

In Abbildung 4.9 ist der grobe Aufbau der Struktur der Simulation in Kombination mit ROS zu sehen. Hier wird deutlich gemacht, dass die Simulation und ROS über einen einzelnen Kanal kommunizieren. So besteht eine TCP-Verbindung zwischen dem ROSConnector auf Unity's Seite und dem server_endpoint auf Seite von ROS. Die einzigen Informationen, die für diese Verbindung notwendig sind, sind die gemeinsam genutzte IP-Adresse und deren Port. Auf Seite des

server_endpoint wird zusätzlich eine Liste geführt, unter welchen Topics Daten von ROS und an ROS gesendet werden. Zu den Daten, die von ROS an Unity gesendet werden, gehören beispielsweise das Robotermodell, dargestellt als robot_model.urdf, sowie die Richtung, in die der Roboter fahren soll, dargestellt als cmd_vel. Die Daten, die wiederum empfangen werden, sendet der Server and den robot_state_publisher, der die Daten über Topics in Form von Messages an das Kernsystem von ROS sendet, hier roscore genannt. In ROS werden intern durch weitere Packages die Daten verarbeitet. So können beispielsweise Bilder oder die Odometrie in RViz grafisch dargestellt werden, oder anderweitig verarbeitet werden, wie z.B. in einer Segmentierung der 3D Daten.

Auf Seite von Unity werden die aus ROS erhaltenen Daten an Objekte in der Scene weitergeleitet. So befindet sich in der Scene beispielsweise der simulierte Roboter, der über Scripte zur Datenverarbeitung verfügt. Beispielsweise werden über das Script ROSCamera, welches in Abbildung 4.11 näher beschrieben wird, Bilddaten aufgenommen. Das Objekt cmd_vel erhält die Daten aus dem gleichnamigen package aus ROS und bewegt den Roboter in der Simulation entsprechend. Über das Script tf_publisher werden die Transformationsdaten in der Simulation in ROS-Messages umgewandelt und weitergesendet. Alle Scripte sind wiederum mit dem ROSConnector verbunden und senden durch ihn ihre Informationen zurück an ROS über die TCP-Verbindung.

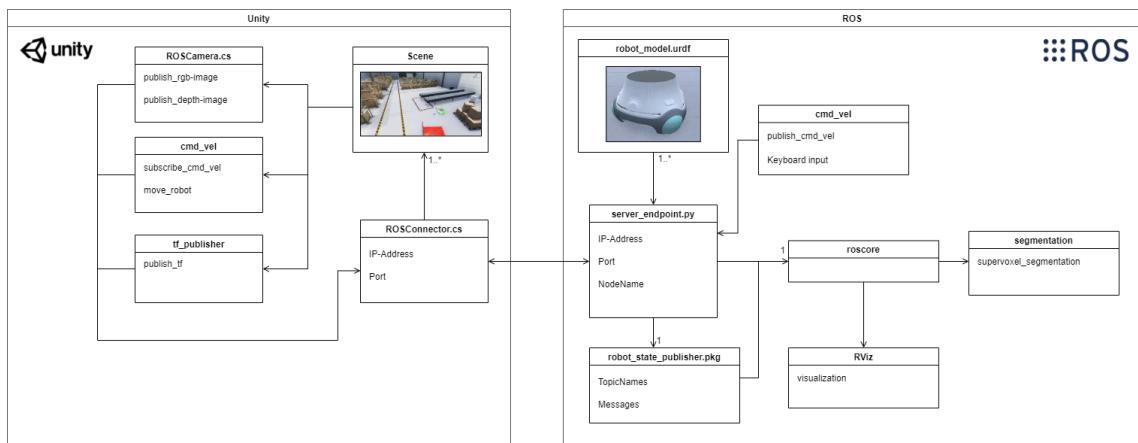


Abbildung 4.9: Struktur der Unity Simulation und deren Kommunikation zu ROS.

Simulation nicht-stationärer Umgebungen

Da Unity's Fokus als Game Engine auf interaktivem Echtzeitcontent liegt, ist es einfach, nicht-stationäre Objekte zu simulieren. So können dynamische Objekte, wie der in Abbildung 4.7 dargestellte Gabelstapler, durch ein einfaches Script bewegt werden. Dort wird die Geschwindigkeit des Objekts gesetzt. Durch Collider,

die als Trigger dienen, kann das dynamische Objekt beispielsweise an bestimmten Punkten dazu gezwungen werden, eine Kurve zu fahren. Semi-statische Objekte sind ebenfalls mit Leichtigkeit einzubinden. So können einzelne, semi-statische Objekte wie Kisten mit einem Tag versehen werden. Ein Script kann durch diese Tags auf die Objekte zugreifen, und sie in zufälligen oder vorgegebenen Mustern verschwinden, wieder erscheinen oder an eine andere Stelle bewegen lassen. So kann simuliert werden, dass eine Kiste außerhalb der Sensorreichweite des Roboters bewegt wird. Wenn vorgegebene Muster verwendet werden, ist die Simulation so für wissenschaftliche Experimente zu 100% wiederholbar. In Abbildung 4.10 ist ein Beispiel für semi-statische Objekte dargestellt. Links sind die vollen Regale mit allen Kisten darin zu sehen. Im Mittleren Bild wurden einige der Kisten durch ein Script entfernt. Im rechten Bild sieht man die leeren Regale ohne jegliche Kisten in ihnen.

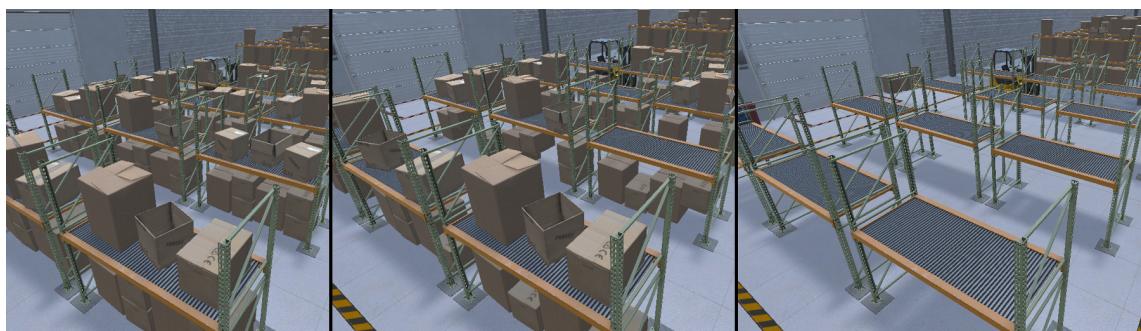


Abbildung 4.10: Semi-statische Objekte in der Unity Simulation. Links: Volle Regale, Mitte: Regale mit zufälligen fehlenden Kisten, Rechts: Leere Regale.

Simulation der Sensoren

Da der Fokus der Arbeit auf vSLAM Verfahren liegt, wurden in der Simulation hauptsächlich Sensoren implementiert, die bei vSLAM-Systemen zum Einsatz kommen. Die wichtigsten Sensoren sind hierfür eine RGB-D Kamera, Odometrie, sowie die Transforms des Roboters.

Tiefenkameras bestehen, wie in Kapitel 2.1.1 beschrieben, entweder aus einer Farbkamera und einem separaten Tiefensensor (RGB-D Kamera) oder aus zwei Farbkameras (Stereokameras).

Da Farbkameras und Tiefenkameras über einige Überschneidungen in ihren notwendigen Parametern verfügen, wurde für die Simulation eine Superklasse namens *RosCamera* erstellt. Die Attribute, Methoden und Beziehungen dieser Klasse sind in Abbildung 4.11 im UML Format dargestellt. Jedes Element in der Simulation, welches in der Lage ist, Messages an ROS zu senden, verfügt über ein

ROSConnection Objekt. All diese Objekte verfügen zusätzlich über ein Timer Objekt und einen messageFrequency Wert. Über diese beiden Attribute kann die Frequenz gesteuert werden, mit der die Messages an ROS gesendet werden. Zudem sind eine FrameId und ein Topic-Name für die Messages notwendig. Neben der Awake Funktion für die Initialisierung der Attribute verfügt RosCamera über die Funktion GetRawTextureData, in der eine Textur von Unity in einen Byte-Array umgewandelt wird, der in ROS Messages verwendet werden kann.

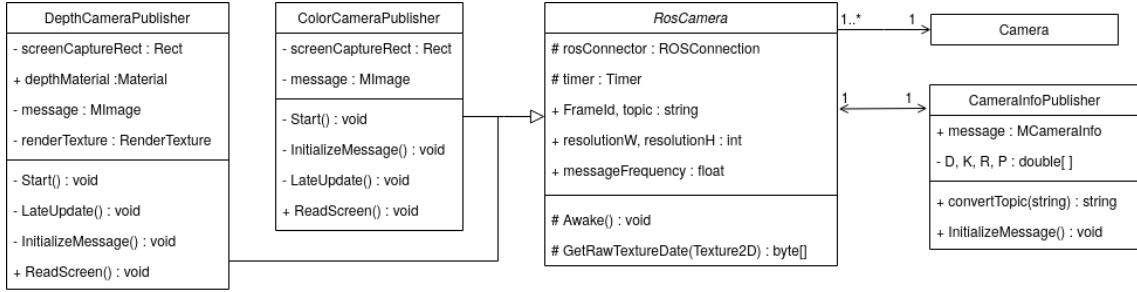


Abbildung 4.11: UML Diagramm der Kameras.

Die Klassen DepthCameraPublisher und ColorCameraPublisher sind Unterklassen von RosCamera. Sie verfügen zusätzlich beide über die Attribute screenCaptureRect und message. ScreenCaptureRect beschreibt den Bildausschnitt, der aus der Grafikkarte gelesen werden soll. Die message ist die Repräsentation einer ROS Message in Unity. Ausschnitte der Methoden der Klasse ColorCamera-Publisher befinden sich im Anhang in Codeausschnitt 2. Die Funktion LateUpdate befindet sich in allen Scripten, die Daten über Messages an ROS senden. Hier wird nach jedem fertigen Frame die verstrichene Zeit geprüft. Wenn mehr Zeit vergangen ist als die Frequenz, die für die jeweilige Message angegeben wurde, wird das aktuelle Bild eingelesen und an ROS versendet. Bei der Kamera ist dies beispielsweise 30 mal in einer Sekunde, wie es bei der Intel RealSense D435 der Fall ist. Das Bild wird mit Hilfe von Unity's Graphics.Blit Funktion in eine temporäre RenderTexture geladen. Durch die in Kapitel 2.4.3 beschriebenen Unterschiede in den Koordinatensystemen zwischen ROS und Unity, muss das Bild zusätzlich vertikal gespiegelt werden. Die resultierende RenderTexture wird für eine kurze Zeit als aktive RenderTexture gesetzt, um anschließend mit der Texture2D.ReadPixels Funktion die Pixelwerte aus der aktiven RenderTexture zu lesen. Diese Pixelwerte werden zum Schluss in eine ROS Message umgewandelt und über den ros-Connector an den ROS Server gesendet.

Um an Tiefeninformationen in der Unity-Simulation zu gelangen, gibt es verschiedene Möglichkeiten. Zum einen könnte durch Raytracing ähnlich wie bei einem realen Laserscanner Strahlen ausgesendet werden, welche Distanzwerte berechnen könnten. Die andere Möglichkeit ist es, auf die Tiefeninformationen, über die Unity ohnehin bereits verfügt, zuzugreifen. Dies passiert in Shadern als

Postprocessing Effekt, wodurch das 2D Bild am Ende eines Renderzyklus beeinflusst wird. Dieses Tiefenbild kann anschließend ähnlich wie ein Farbbild nach der oben beschriebenen Methode aus der Grafikkarte gelesen und an ROS gesendet werden. In Codeausschnitt 3 ist ein Ausschnitt des in der Simulation verwendeten Tiefenshaders abgebildet. Der Shader basiert teilweise auf einem Tutorial von der Seite Ronja's tutorials¹⁴ und teilweise auf den Shadern aus ISAAC und AirSim. Im Fragment Shader kann durch die Funktion `Linear01Depth(depth)` direkt auf die Tiefeninformation jedes Pixels zwischen der Kamera und der Far Clipping Plane (die weit entfernteste Ebene, die noch gerendert wird) zugegriffen werden. Um die Tiefe in Z Richtung, in Unity Koordinaten nach vorne, zu bekommen, müssen die Tiefendaten noch mit den `_ProjectionParams.z` multipliziert werden. Dies führt jedoch abhängig von der Far Clipping Plane zu extremen Werten, wodurch alle Tiefendaten entweder als sehr nah oder sehr fern interpretiert werden. Um realistischere Ergebnisse zu erhalten, muss das Ergebnis dieser Rechnung durch einen weiteren Wert dividiert werden. In dem in Kapitel 4.1.3 beschriebenen Szenario eines großen Warenlagers erwies sich die Konstante 30 als passender Wert. Alternativ kann diese Konstante auch dynamisch anhand der Near- und Far Clipping Plane berechnet werden, was sich jedoch auf die Performance auswirken könnte. In Abbildung 4.12 ist eine Demo-Szene zu Testzwecken für den Tiefenshader abgebildet. Die links abgebildete Szene zeigt eine große, leicht nach oben geneigte Ebene. Darüber schweben versetzt vier Würfel mit unterschiedlicher Entfernung zur Kamera. Rechts ist die Sicht der Kamera mit dem angewendeten Tiefenshader zu sehen. Dabei erscheinen nahe Objekte schwarz, ferne Objekte wirken grau. Der weiße Streifen am Horizont ist die Far Clipping Plane, da sich keine Objekte vor dieser Ebene befinden. Auch wenn sich Objekte hinter dieser eigentlich unsichtbaren Ebene befinden würden, würden sie nicht gerendert werden und in dem Shader durch die weiße Überdeckung nicht sichtbar sein.

In Abbildung 4.13 wurde der Tiefenshader auf eine der Kameras in der Simulation angewendet. Links ist hier die Sicht der Farbkamera zu sehen, rechts ist die Sicht der Tiefenkamera zu sehen. Der Code der Tiefenkamera hierfür ist nahezu identisch zu der `ReadScreen` Funktion der Farbkamera aus Codeausschnitt 2. Einzig bei dem Aufruf `Graphics.Blit` muss hier zusätzlich ein Material angegeben werden, welches den Tiefenshader beinhaltet. Des Weiteren muss das Encoding für die ROS Message geändert werden. So ist das Encoding hier statt `rgb8` für 8-Bit Farbwerte das nötige Encoding "32FC1", was für 32-Bit Tiefenwerte als Float steht, da jeder Pixel nicht aus Farbwerten besteht, sondern aus Tiefenwerten, die sich zwischen Schwarz und Weiß über Grautöne darstellen lassen.

Des Weiteren verfügt jede `RosCamera` über eine Referenz auf eine Unity Camera, sowie über einen `CameralInfoPublisher`. Ein `CameralInfo` Objekt in ROS beschreibt

¹⁴<https://www.ronja-tutorials.com/post/017-postprocessing-depth/>

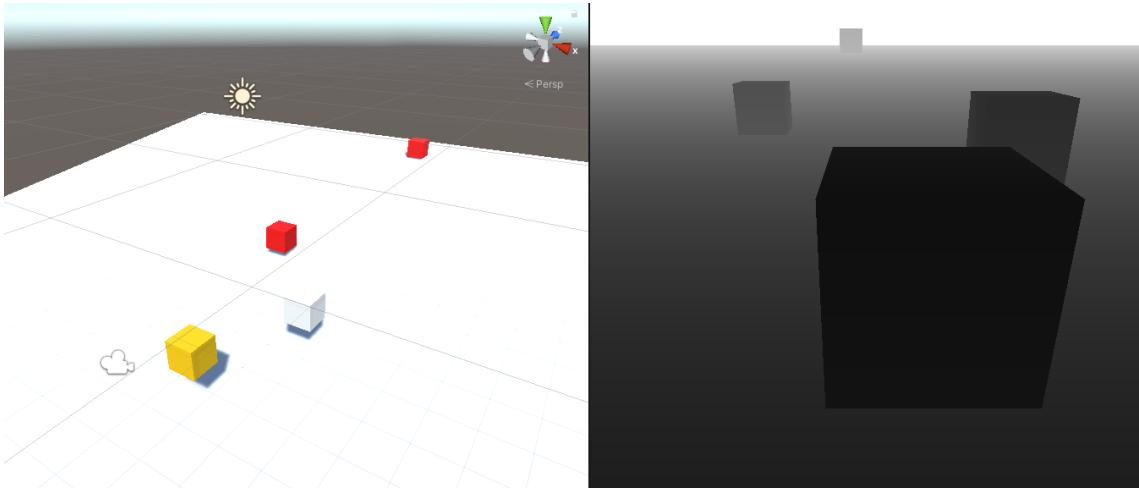


Abbildung 4.12: Demo-Szene mit einem Tiefenshader.

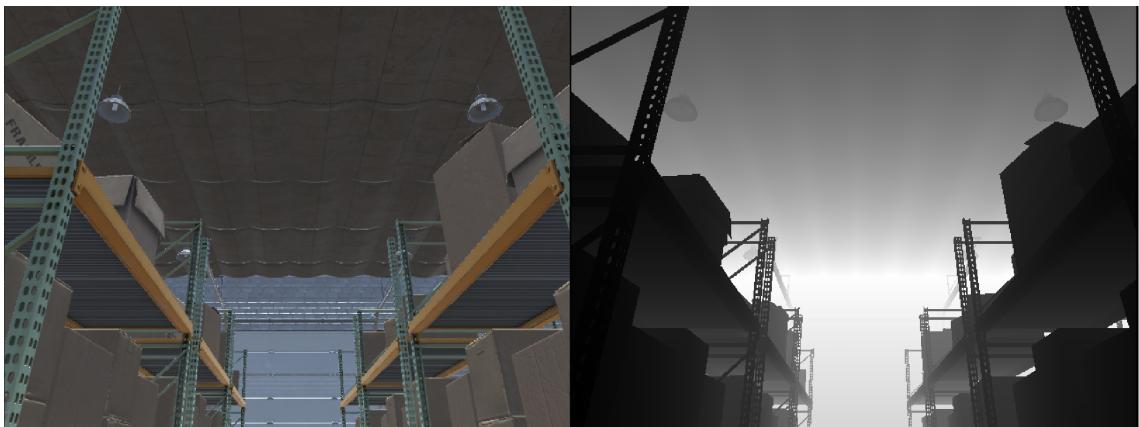


Abbildung 4.13: Der Tiefenshader in der Simulationsumgebung.

die Parameter, die eine reale Kamera aufweist. Somit kann selbst in einer Simulation ein bestimmter Typ von Kamera simuliert werden. Die dafür relevanten Parameter bestehen alle aus mehreren double Werten, die in Arrays gespeichert werden und haben die Namen D, K, R und P. In der ROS Dokumentation¹⁵ sind die Werte ausführlich beschrieben. Der Wert D bestimmt, welches Modell der optischen Verzerrung auf die Kameralinse zutrifft. In K wird in einer 3x3 Matrix beschrieben, wie 3D Punkte im Koordinatensystem der Kamera in 2D Pixel-Koordinaten umgewandelt werden. In R wird bei Stereokameras in einer 3x3 Matrix eine Rotation beschrieben, durch die beide Kamerakoordinaten parallel zueinander liegen. Im Wert P wird für Stereokameras in einer 3x4 Matrix beschrieben, wo sich das Zentrum der zweiten Kamera im Koordinatensystem der ersten Kamera befindet. Bei einer monokularen Kamera besteht der P-Wert meist aus einer Einheitsma-

¹⁵http://docs.ros.org/en/noetic/api/sensor_msgs/html/msg/CameraInfo.html

trix. Die Parameter für die CameraInfo für den Test-Aufbau wurden anhand der Parameter einer realen Intel RealSense D435 Kamera gewählt, welche in Codeausschnitt 4 zu sehen sind.

4.1.4 Evaluierung der Simulation

Die im Zuge dieser Arbeit erstellte Simulation wurde anhand der Anforderungen an eine Robotersimulation von Hertzberg et al. [HLN12] aus Kapitel 2.4 und den in Kapitel 4.1.1 vorgestellten Kriterien evaluiert. Im folgenden werden die relevanten Punkte erneut zusammengefasst aufgezählt und diskutiert, ob die erstellte Simulation die Anforderungen und Kriterien erfüllt.

Die Anforderungen von Hertzberg et al. [HLN12] umfassen folgende vier Punkte:

i. **Die Umgebung muss die Realität ausreichend simulieren.**

Im Vergleich mit bereits bestehenden Robotersimulationen und anhand der Qualität der aus der Simulation entstehenden Kamerabildern lässt sich ableiten, dass die Simulation die Realität ausreichend simuliert. Die in der Testszene verwendeten Meshes sind von hoher Qualität, so bestehen die Boxen je nach Komplexität aus ca. 40 Polygonen bis hin zu ca. 5700 Polygonen. Die Texturen sind ebenfalls hochauflösend. So verfügen die bei den Boxen verwendeten Texturen über eine Auflösung von 4096x4096 (4K) Pixeln und verwenden Normal Maps und Height Maps für realistisch wirkende Oberflächen. Die für den Boden und für die Wände verwendeten Texturen von Adobe Substance 3D verfügen ebenfalls über 4K Auflösung und über Normal Maps und Height Maps. Zusätzlich können semi-statische und dynamische Objekte simuliert werden, was die simulierte Umgebung zur LT-SLAM Anwendung ebenfalls realistischer werden lässt.

ii. **Der Roboter muss in Aktorik und Sensorik hinreichend gut simuliert sein.**

Jeder Sensor, der in dem verwendeten URDF Modell des Roboters vorhanden ist, wurde in der Simulation abgebildet. So unterstützt die Simulation aktuell Farb- und Tiefenkameras, Laserscanner und Odometriesensoren. Zusätzlich ist es möglich, in der Simulation statische und dynamische Transformationen zwischen Frames des Roboters an die TF-Topic zu senden. Durch die direkte Umwandlung der aufgenommenen Daten der simulierten Sensoren in das Format von ROS Messages werden jederzeit die Messergebnisse des korrekten Typs generiert. Ungenauigkeiten der Sensoren wie Messfehler und Rauschen werden jedoch aktuell nicht simuliert.

Es ist jedoch möglich, diese Funktionalität im Nachhinein zu implementieren. Hierfür könnten die modular aufgebauten Klassen der Sensoren abgeleitet werden und das Rauschen in Klassen wie *NoisyImageCamera* eingebaut werden. Hinsichtlich der Aktorik werden die Räder des Roboters über die eingehende cmd_vel aus ROS angesteuert. Dabei wird jedem Rad die entsprechende Geschwindigkeit zur Rotation zugewiesen, um die gegebene Bewegung umzusetzen. Die verwendete Basis des Care-O-bot verfügt über keine weiteren Aktoren.

iii. Der Roboter darf nur über die simulierten Aktoren und Sensoren mit der Simulationsumgebung interagieren.

Der Roboter interagiert nur über die simulierten Aktoren und Sensoren mit der Umgebung. Die bei der eingesetzten Care-O-bot 4 Basis einzigen Aktoren sind drei Räder. Diese bewegen durch Rotation den Roboter und sind auch in Geschwindigkeit und Manövriergängigkeit limitiert. Die Sensoren haben ebenfalls nur Zugriff auf die von den jeweiligen Objekten generierten Daten. So greifen die Kamera-Sensoren nur auf die von den Unity-Kameras aufgenommenen Bilddaten zu. Die Odometrie wird anhand der Bewegung des Roboters gemessen, die von den simulierten Rädern abhängig ist.

iv. Die Software und die Schnittstelle müssen für die Simulation die selbe sein wie für den realen Roboter.

Wie bereits beschrieben, werden die Daten in Unity in ROS-Messages umgewandelt und durch den *ROSConnector* über die entsprechenden Nodes an ROS gesendet. Für ROS sieht es also so aus, als ob die Daten von einem echten Roboter stammen.

Zusätzlich wurde die Simulation anhand der in Kapitel 4.1.1 vorgestellten Kriterien evaluiert. Die Punkte zu *Quality* und *Sensors* wurden bereits diskutiert und werden daher im folgenden nicht aufgezählt:

- i. **Overhead:** Die Simulation wurde von Beginn an mit Modularität und niedrigem Overhead konzeptioniert. So beinhaltet die Unity-Szene wie in Abbildung 4.14 dargestellt nur drei Komponenten: Die Umgebung, in diesem Fall das Gebäude *building*, der simulierte Roboter *cob4-18* und der optionale *TF_Publisher*. Der Roboter ist dabei mit den nötigen Scripten für die Sensoren und Aktoren ausgestattet. Die benötigten Definitionen der ROS Messages können selektiv durch die Funktion zum generieren der Messages installiert werden.
- ii. **Performance:** Die Simulation läuft bei einer Full HD Auflösung von 1920x1080 Pixeln durchgehend über 60 FPS. Die durchschnittliche Bildrate beträgt dabei, je nach Kontext in der Umgebung, zwischen 100 FPS und 120 FPS. Die Werte stammen dabei zum einen aus der Berechnung der Bildrate durch

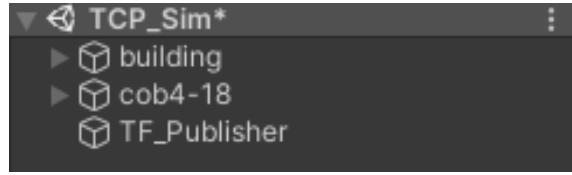


Abbildung 4.14: Aufbau der Unity-Szene.

ein Script, bei dem die Bildrate anhand Unity's *deltaTime* berechnet wird. Die Ergebnisse dieser Berechnung schwanken dabei zwischen 110 FPS und 120 FPS. Zum anderen wurde Unity's Analysetool mit dem Namen *Profiler* verwendet. In Abbildung 4.15 ist ein Ausschnitt des Profilers für die Berechnungen der CPU dargestellt. Die Auslastung schwankt dabei durchgehend zwischen 60 FPS und 200 FPS, was nach dem *Stats* Fenster in der Echtzeitansicht bei laufender Simulation zu durchschnittlich 110 FPS führt.

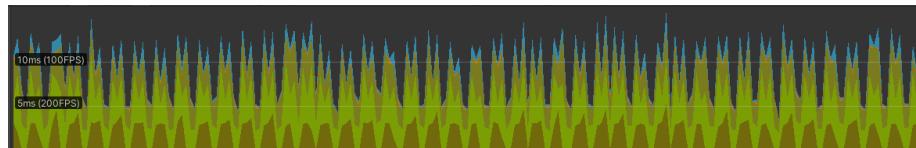


Abbildung 4.15: Ausschnitt aus dem Unity Profiler bei laufender Simulation.

Die Übertragungsrate der ROS Messages übertrifft ebenfalls die Anforderungen. So beträgt die Übertragungsrate der Daten der Farbkamera über 30 Sekunden gemessen durchschnittlich 43.916 Hz bei einer Standardabweichung von 0.00482 Sekunden. Die Daten der Topic TF erreichen eine Frequenz von 100.119 Hz bei einer Standardabweichung von 0.01653 Sekunden. Gemessen wurden diese Werte mit der Funktion *rostopic hz* von ROS.

- iii. **Import & Export:** Der Import von URDF Dateien ist über den URDF Importer des Unity Robotics Hub möglich. Hierfür muss lediglich die URDF Datei im Dateisystem des Betriebssystems in den Assets Ordner des Unity Projekts gelegt werden oder per Drag&Drop in das Unity Projekt importiert werden. Anschließend muss per Rechtsklick auf die Datei im Unity Editor das Menü zum importieren geöffnet werden.
Das Importieren von Objekten und Umgebungen ist eine Grundfunktion von Unity und ebenfalls über den Assets Ordner, per Drag&Drop oder per Import-Dialog in Unity möglich. Dabei wird der Import von Dateiformaten aus jeder gängigen 3D-Modellierungssoftware unterstützt¹⁶.

- iv. **Little Complexity:** Die Komplexität des Systems hält sich in Grenzen. So folgt die Struktur des Projekts der gängige Struktur eines Unity Projekts mit

¹⁶<https://docs.unity3d.com/Manual/ImportingModelFiles.html>

Erweiterungen für die Nutzung zusammen mit ROS. So besteht die überliegende Ordnerstruktur aus den Ordnern Materials, Prefabs, RosMessages, Scenes, Scripts, URDF und Warehouse. Der Code wurde nach den Paradigmen von Clean Code erstellt und enthält demnach Variablen und Funktionen mit aussagekräftigen Namen. Komplexe Funktionen und Klassen wurden aufgeteilt oder nach Bedarf in mehrere Klassen gespalten. Das System ist einfach um weitere Funktionen wie neue Sensoren und Aktoren erweiterbar.

- v. **Usability:** Zur Bedienung des Systems sind wenige Schritte erforderlich. So muss nach Festlegung der IP und des Ports des TCP Connectors auf ROS Seite der Server gestartet werden. Nach Import des Roboters in Unity und festlegen der Parameter kann das System gestartet werden. Der Roboter lässt sich über die gängige Topic cmd_vel nach beliebiger Methode steuern. Wichtige Parameter wie die Frameld der Roboterkomponenten oder die Namen der Topics lassen sich über das Interface des Unity Inspectors festlegen. Neue Dateien lassen sich wie oben beschrieben beispielsweise leicht per Drag&Drop importieren. Durch die direkte Anbindung der Simulation an ROS kann der gewohnte Workflow von ROS genutzt werden, da die Simulation über eine einzige Schnittstelle mit ROS kommuniziert.

4.1.5 Limitationen

Der im Zuge dieser Arbeit entwickelte Simulator erfüllt die Anfangs definierten Anforderungen sehr gut, dennoch weist er noch einige Limitationen auf. So ist die bisherige Auswahl von Sensoren auf Farbkameras, Tiefenkameras und 3D-Laserscanner beschränkt. Außerdem werden Messfehler von realen Sensoren wie Rauschen nicht simuliert, wodurch sich die generierten Daten teils von realen Sensordaten unterscheiden. Vorhandene Aktoren bestehen bisher nur aus den Rädern zur Fortbewegung eines mobilen Roboters. Des weiteren werden zum aktuellen Stand Multi-Roboter Simulationen nicht unterstützt.

4.2 Erkennung semi-statischer Objekte

Um LT-SLAM Systeme zu ermöglichen ist es von Vorteil, Objekte in die in Kapitel 2.2.2 beschriebenen Kategorien statisch, semi-statisch und dynamisch einzzuordnen. Statische Objekte sollten dabei immer fester Bestandteil der globalen Karte sein, da sie sich gar nicht oder nur sehr wenig bewegen oder verändern und daher als Referenz und Landmarken geeignet sind. Dynamische Objekte hingegen werden in der Regel komplett gefiltert und in keiner Karte aufgenommen, da sie sich aktiv bewegen oder verändern und daher nicht als Referenz geeignet sind. Semi-statische Objekte, die häufig in LT-SLAM Szenarien und nicht-stationären Umgebungen vorkommen, können nicht einfach wie dynamische Objekte gefiltert werden oder wie statische Objekte in die Karte aufgenommen werden. So wirken sie so lange wie statische Objekte, bis sie sich bewegen. Aus diesem Grund wird im folgenden ein Ansatz beschrieben, wie semi-statische Objekte mit Hilfe von Tiefensensoren, insbesondere RGB-D Kameras erkannt werden können. Hierfür wird zunächst ein Algorithmus zur Segmentierung von Punktwolken ausgewählt. Anhand dieser Segmente werden mit Hilfe von 3D Features Objekte erkannt. Um die Algorithmen zu testen, werden zwei Punktwolken, die aus der oben beschriebenen Simulation stammen, verwendet.

4.2.1 Aufbau des Projekts

Aufgrund der breiten Auswahl an vorgefertigten Funktionen und der Integration mit der oben beschriebenen Unity Simulation wurde ROS als primäre Bibliothek zur Entwicklung der Objekterkennung verwendet. Bei dem verwendeten Build-System handelt es sich um catkin, welches mit Hilfe von CMake aus C++ Quellcode ausführbare Dateien erstellt. Für die Manipulation und Bearbeitung von Punktwolken wurden die Funktionen der Point Cloud Library (PCL)¹⁷ verwendet.

4.2.2 Auswahl des Segmentierungsalgoritmus

Als Basis für die Auswahl der Segmentierungsalgorithmen wurden die Implementierungen von Lundell und Verdoja verwendet. Auf GitHub¹⁸ stellen sie unter der MIT Lizenz vier Segmentierungsalgorithmen zur Verwendung mit ROS zur Verfügung. Dabei handelt es sich um Algorithmen für die Euclidean Cluster Extraction,

¹⁷<https://pointclouds.org/>

¹⁸https://github.com/aalto-intelligent-robotics/point_cloud_segmentation

Region Growing Segmentation, Plane model Segmentation und Supervoxel Segmentation. Besonders vielversprechend ist dabei die Implementierung zur Supervoxel Segmentation¹⁹, da Verdoja et al. einen Artikel [VTS17] über ihre Implementierung veröffentlichten. Dabei liegt der Fokus des Algorithmus auf der Anwendung mit 3D Punktwolken, die aus RGB-D Kameras stammen und ist nach den Autoren schnell genug für den Einsatz in Echtzeitsystemen. Getestet wurde der Algorithmus im Kontext eines Roboter Greifarms, für den primär kleine Objekte auf einem Tisch segmentiert wurden. Der dafür verwendete Datensatz ist die Object Segmentation Database (OSD) aufgrund der Ground Truth für Segmentierungen, die der Datensatz enthält. Für den Einsatz auf mobilen Robotern in nicht-stationären Umgebungen wurde der Algorithmus jedoch nicht getestet. Die Supervoxel Segmentierung von Verdoja et al. [VTS17] baut auf der Basis von Papon et al. [PASW13] auf und erweitert deren Ansatz durch das Zusammenführen von Farbinformationen und geometrischen Informationen.

4.2.3 Parameterauswahl

Der Algorithmus zur Supervoxel Segmentierung benötigt zur optimalen Operation eine Auswahl an verschiedenen Parametern, die festgelegt werden müssen. In Tabelle 4.3 sind die Parameter mit ihren Datentypen dargestellt. Die Parameter adapt_lambda und equalization stammen dabei von Verdoja et al. [VTS17], die restlichen Parameter sind für die Basis des Algorithmus von Papon et al. [PASW13] notwendig. Im folgenden werden die Parameter kurz beschrieben:

- i. **voxel_resolution:** Die Auflösung der Voxel beschreibt den Radius während der Suche nach benachbarten Voxeln in einem kd-tree. Dabei wird in einem Radius von $\sqrt{3} * voxel_resolution$ nach Voxeln gesucht, die eine Fläche, eine Kante oder einen Vertex gemeinsam haben.
- ii. **seed_resolution:** Bei der Supervoxel Segmentierung handelt es sich nach der Definition aus Kapitel 3.2 um eine *Seeded-Region* Methode. Hierfür wird der Raum zuerst in ein Grid aus Voxeln unterteilt. Jeder Voxel aus diesem Grid hat dabei die Größe, die in seed_resolution definiert wurde. Wichtig ist hier, dass seed_resolution einen bedeutend größeren Wert als voxel_resolution aufweist.
- iii. **color_importance:** Die color_importance fließt als Konstante in der Berechnung für die normalisierte Distanz zwischen Supervoxeln ein. Dabei handelt es sich speziell um die euklidische Distanz der Supervoxel im CIELAB-Farbraum. Der CIELAB-Farbraum beschreibt im Gegensatz zu beispielsweise RGB alle vom Menschen wahrnehmbaren Farben.

¹⁹<https://github.com/fverdoja/Fast-3D-Pointcloud-Segmentation>

- iv. **spatial_importance:** Die spatial_importance fließt ebenfalls als Konstante in der Berechnung für die normalisierte Distanz zwischen Supervoxeln ein. Hiermit wird die räumliche Distanz zwischen Supervoxeln beschrieben.
- v. **normal_importance:** Die normal_importance fließt ebenfalls als Konstante in der Berechnung für die normalisierte Distanz zwischen Supervoxeln ein. Hierbei wird die geometrische Ähnlichkeit zwischen Supervoxeln anhand der Normalen der Voxel beschrieben.
- vi. **threshold:** Ohne ein Abbruchkriterium würde der Algorithmus die Regionen des Graphen so lange zusammenführen, bis nur noch zwei Regionen übrig sind. Der threshold stellt dieses Abbruchkriterium dar. Wenn die kleinste Distanz zwischen zwei Regionen in dem Graph größer ist, als der threshold, wird der Algorithmus gestoppt.
- vii. **rgb_color_space:** Wenn dieser Wert *TRUE* ist, wird statt dem oben beschriebenen CIELAB-Farbraum der RGB-Farbraum verwendet.
- viii. **convexity_criterion:** Wenn dieser Wert *TRUE* ist, wird die geometrische Distanz in der Berechnung für die normalisierte Distanz mit einbezogen.
- ix. **Gewichtung der Distanzen:** Um die normalisierte Distanz zwischen Supervoxeln zu berechnen, müssen die Farbdistanz und die geometrische Distanz zusammengefügt werden. Um einen Vergleich zwischen diesen Metriken zu ermöglichen, müssen die Werte zuerst anhand eines Gewichts transformiert werden. Hierfür wurden von Verdoja et al. in [VTS17] drei Herangehensweisen vorgeschlagen.
 Bei **Manual Lambda** (*adapt_lambda* = FALSE) werden alle Werte der Distanzen anhand eines Werts $\lambda \in [0, 1]$ transformiert.
 Bei **Adaptive Lambda** (*adapt_lambda* = TRUE) wird der Wert λ anhand der Verteilung der beiden Distanzen berechnet.
 Bei **Equalization** (*equalization* = TRUE) werden die Werte der Distanzen in eine Stetige Gleichverteilung in dem Bereich $[0, 1]$ umgewandelt.

Um die Parameter zu bestimmen, wurde durch Testen verschiedener Werte eine Vorauswahl getroffen, in der sinnvolle Ergebnisse erzielt wurden. Anschließend wurden die in Tabelle 4.3 beschriebenen Wertebereiche und Schrittgrößen definiert. Der Wertebereich gibt dabei ungefähr den Bereich an, in dem von der Segmentierung sinnvolle Ergebnisse produziert werden, wobei teilweise der niedrigste und höchste Wert bereits das Endergebnis signifikant verschlechtern. Bei den vier untersten Werten in der Tabelle handelt es sich um Variablen vom Typ Boolean. Daher wird jede mögliche Kombination dieser vier Werte für jede Kombination der anderen Werte verwendet. Wie oben beschrieben wird durch die Kombination der Parameter *adapt_lambda* und *equalization* eine der Herangehensweisen aus Manual Lambda, Adaptive Lambda oder Equalization gewählt.

Parameterauswahl			
Parameter	Typ	Wertbereich	Schrittgröße
voxel_resolution	Float	0.01 - 0.03	0.005
seed_resolution	Float	0.25 - 0.75	0.05
color_importance	Float	0.1 - 0.3	0.02
spatial_importance	Float	0.2 - 0.6	0.04
normal_importance	Float	0.8 - 1.3	0.1
threshold	Float	0.1 - 0.3	0.1
rgb_color_space	Boolean	FALSE / TRUE	/
convexity_criterion	Boolean	FALSE / TRUE	/
adapt_lambda	Boolean	FALSE / TRUE	/
equalization	Boolean	FALSE / TRUE	/

Tabelle 4.3: Parameter, deren Datentyp und der getestete Wertebereich für die Supervoxel Segmentierung.

4.2.4 Parameter Evaluation

Um die optimalen Werte für die oben beschriebenen Parameter zu bestimmen, wurde für jede Kombination der Werte mit den in Tabelle 4.3 beschriebenen Wertebereich und Schrittgröße eine segmentierte Punktfolke erstellt und mit der zugehörigen Ground Truth verglichen.

Die Punktfolke, die für diesen Vorgang verwendet wurde, stammt aus der in Kapitel 4.1 beschriebenen Unity Simulation. In Abbildung 4.16 ist die für die Parameterauswahl verwendete Szene abgebildet. Links ist die Sicht der Farbkamera in der Simulation zu sehen, rechts die zugehörige farbige Punktfolke.



Abbildung 4.16: Die für die Parameterauswahl verwendete Referenz-Szene. Links ist die Sicht der Farbkamera in der Simulation, rechts die zugehörige Punktfolke.

Diese Punktfolke wurde mit jeder Kombination der oben beschriebenen Parametern segmentiert. Jede segmentierte Punktfolke wurde zur weiteren Verarbeitung im pcd-Format in ASCII-Form gespeichert. Dabei enthält jede Reihe der Datei die x,y und z Koordinaten jedes Punkts sowie ein zugehöriges Label. Die Label bestehen dabei aus natürlichen Zahlen, angefangen bei der Null.

Um die Qualität der Segmentierung objektiv zu bewerten, wurde jede segmentierte Punktfolke mit einer Ground Truth verglichen. Hierfür wurde die selbe Punktfolke, die als Eingabe für die Segmentierung diente, per Hand mit Labeln versehen. Hierfür wurde die Webseite Supervisely²⁰ verwendet. In Abbildung 4.16 ist die Punktfolke mit den Ground Truth Labeln zu sehen. Die Farben wurden hier zufällig gewählt und wurden nur zum Zweck der Visualisierung verwendet. Die Label der Ground Truth folgen ebenfalls dem Schema von natürlichen Zahlen, angefangen bei der Null.

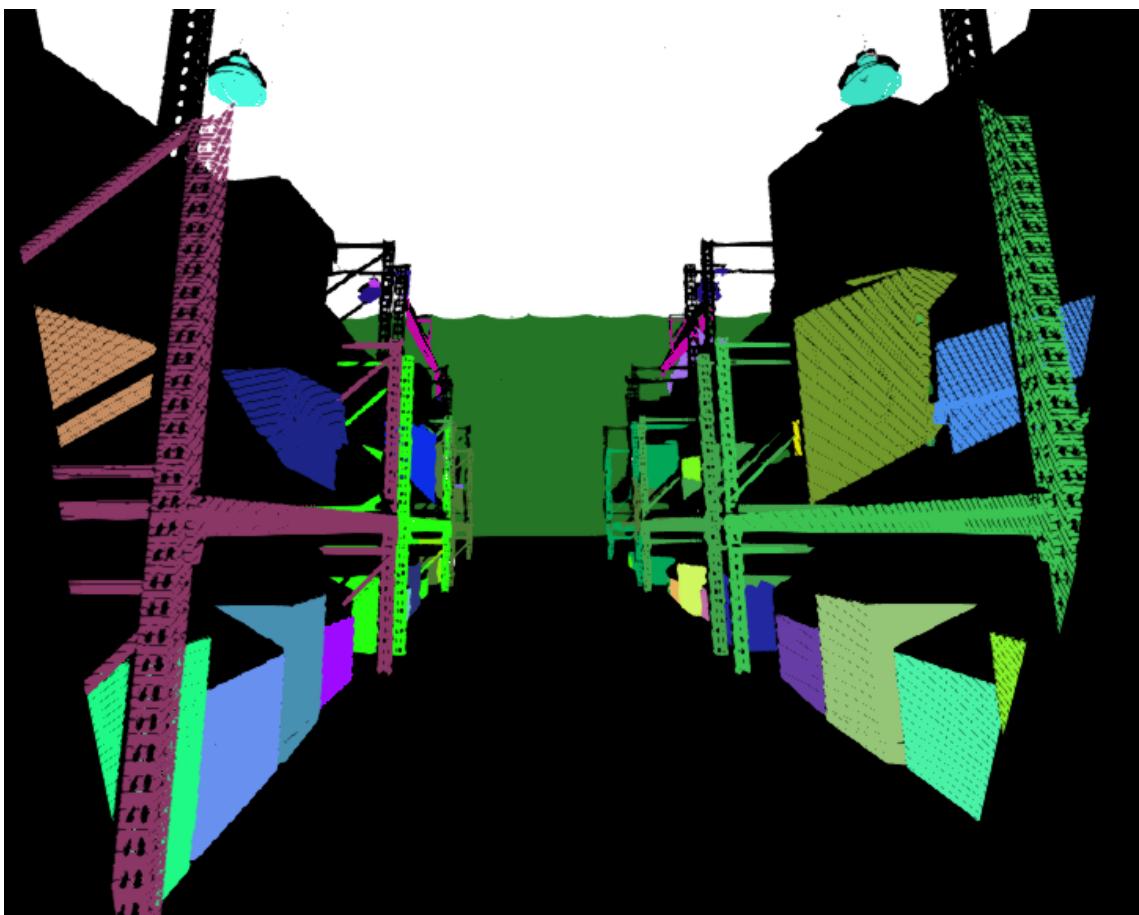


Abbildung 4.17: Punktfolke mit Ground Truth Labeln.

²⁰<https://supervisely.com/lidar-3d-cloud/>

Um die Segmentierung, die am besten mit der Ground Truth übereinstimmt, zu ermitteln, wurde eine abgewandelte Version der Methode von Hoheim et al. [HEH11] und Verdoja et al. [VTS17] angewendet.

Eine der segmentierten Punktwolken P_K ist als Sammlung von Regionen r_k definiert, der Wert N beschreibt die Anzahl der Segmente in der Punktwolke:

$$P_K = \{r_k\}^N$$

Jedes Segment r_K besteht wiederum aus n Punkten x_i :

$$r_K = \{x_i\}^n$$

Die von Hand annotierten Regionen der Ground Truth G sind zusätzlich definiert als g_i :

$$G = \{g_i\}^N$$

Da die Ground Truth Regionen g_i und die Label der Segmente r_k nicht sortiert sind, müssen zuerst die potentiell passenden Regionspaare (g_i, r_k) für jede Kombination von G und P_K gefunden werden. Hierfür wurde für jede Ground Truth Region g_i die Hausdorff Distanz $\delta_h i, r$ zu jeder Region r_k einer segmentierten Punktwolke P_K berechnet. Die Region r_k , die die kleinste Distanz zu einer Ground Truth Region g_i aufweist, bildet mit dieser ein Paar:

$$(g_i, r_k)_i = \min_i(\delta_h i, r)$$

Somit ergibt sich für jede Ground Truth Region g_i eine zugehörige Region r_k . Um die Punktwolke mit der besten Segmentierung zu ermitteln, wird nun die Überlappung O_K aller Regionspaare $(g_i, r_k)_i$ für jede Kombination zwischen G und P_K ermittelt. Die Punktwolke mit der höchsten Überlappung mit der Ground Truth ist somit die beste. Die Überlappung O_K wird anhand der Schnittmenge zwischen jeder Region der Ground Truth der zugehörigen Region $(g_i, r_k)_i$ bestimmt:

$$O_K = \sum_{i=0}^N g_i \cap r_k \mid g_i, r_k \in (g_i, r_k)_i$$

Aus diesen Berechnung hat sich Ergeben, dass die Punktwolken mit den in Tabelle 4.4 dargestellten Werten die beste Segmentierung für die in Abbildung 4.17 dargestellte Ground Truth ergibt. Auf das Ergebnis hatten die Parameter `rgb_color_space`, `convexity_criterion`, `adapt_lambda` und `equalization` wenig Einfluss. So erreichten drei Punktwolken, die sich in diesen vier Punkten unterschieden, den selben Grad an Überlappung mit der Ground Truth. Die Kombinationen der Werte sind in 4.5 abgebildet.

Parameterauswahl - Ergebnis		
Parameter	Typ	Bester Wert
voxel_resolution	Float	0.015
seed_resolution	Float	0.75
color_importance	Float	0.2
spatial_importance	Float	0.4
normal_importance	Float	1.0
threshold	Float	0.2

Tabelle 4.4: Parameter, und deren ermittelter bester Wert.

Parameter	rgb_color_space	convexity_criterion	adapt_lambda	equalization
Variation 1	0	0	1	0
Variation 2	0	1	1	0
Variation 3	1	0	0	1

Tabelle 4.5: Werte der Boolean-Parameter, die die besten Segmentierungen erreichten.

Die Segmentierung lief mit den ermittelten Parametern durchschnittlich mit 0.402 Hz. Eine Segmentierung mit dieser Geschwindigkeit kann möglicherweise im Schritt der globalen Kartierung eines SLAM Systems verwendet werden, da hier die Echtzeitfähigkeit im Hintergrund liegt. Für den Einsatz für eine lokale Karte ist der Ansatz jedoch weniger geeignet, hier wäre eine Segmentierungsgeschwindigkeit von über 1 Hz wünschenswert. Bei niedrigeren Geschwindigkeiten können beispielsweise bei einer Drehung des Roboters bereits nicht alle Objekte erkannt werden, da diese nur für den Bruchteil einer Sekunde in der Punktfolge zu finden sind. Der Code zu Segmentierung könnte möglicherweise noch optimiert werden, indem zusätzlich zur CPU auch die GPU des Systems zur Berechnung verwendet wird, da GPUs für repetitive Berechnungen dieser Art konzipiert sind. Eine weitere Möglichkeit ist das Filtern der Eingangs-Punktfolge. So verfügt ROS über einen VoxelGrid Filter²¹, durch den die Dichte und Anzahl der Punkte verringert werden kann. Zusätzlich beinhaltet der Filter die Funktion des Cutoff, wodurch Punkte außerhalb einer gewissen Distanz zum Roboter aus der Folge entfernt werden, wodurch noch mehr Performance in der Segmentierung erreicht werden kann. So konnte mit einem Cutoff von 5 in jeder Richtung und einer leaf_size von 5 eine Geschwindigkeit von 0.933 Hz bei der Segmentierung erreicht werden. Jedoch wurden bei diesen Einstellungen einige Objekte nicht mehr als Segment erkannt, was sich negativ auf die weiteren Ergebnisse auswirkte.

²¹http://wiki.ros.org/pcl_ros/Tutorials/VoxelGrid%20filtering

4.2.5 Detektieren und Extrahieren von 3D-Features

Um Objekte für LT-SLAM in einer 3D-Punktwolke zu erkennen und diese beispielsweise in einer Karte aufzunehmen, sind Methoden zur Detektierung und Extraktion von Features unabdingbar. Im Gegensatz zur Erkennung von Features im 2D Raum wie in Bildern ist die Erkennung von Features im 3D Raum ein recht unerforschtes Gebiet. Dennoch existieren bereits einige Methoden, um 3D Features in Punktwolken zu erkennen und zu vergleichen. So bietet die PCL eine Liste²², in der verschiedene Arten von 3D Features verglichen werden.

Im folgenden wird eine Auswahl der Feature Deskriptoren, die für diese Arbeit in Frage kommen, vorgestellt:

- i. Normal Aligned Radial Feature (NARF): NARF bezeichnet sowohl eine Methode, um Points of Interest in 3D Tiefenbildern zu detektieren, als auch die Berechnung und Repräsentation dieser Points of Interest in Form von 3D Features [SK10]. Der NARF Deskriptor besteht aus einer lokalen Koordinate mit sechs Freiheitsgraden. Um die Features zu detektieren, wird nach Kanten in den Tiefenbildern gesucht, welche durch harte Übergänge in den Pixelwerten erkannt werden können. Anhand des Verlaufs der Werte in verschiedene Richtungen, wird eine Orientierung des Punkts ermittelt, welche unabhängig von dem Blickwinkel auf den Punkt ist.
- ii. Rotation-Invariant Feature Transform (RIFT): RIFT Features basieren auf den für 2D Bilder häufig genutzten SIFT Features [LSP05]. Bei RIFT wird eine Methode verwendet, um Merkmale anhand von Texturdaten zu erstellen. Jeder Punkt wird dabei durch ein Histogramm repräsentiert, welches wiederum mit anderen Histogrammen verglichen werden kann, um Objekte über mehrere Punktwolken hinweg zu erkennen.
- iii. Viewpoint Feature Histogram (VFH): Das VFH ist ein Deskriptor für Daten aus 3D Punktwolken, in dem Geometrie- und Standpunktdata gespeichert werden. Hierbei besteht der Deskriptor, ähnlich wie bei RIFT, aus einem Histogramm [RBTH10]. Als Eingabe dient direkt eine 3D Punktwolke, für die ein einzelnes Histogramm erstellt wird. Um also Objekte zu erkennen, müssen die Punktwolken zuerst in Segmente aufgeteilt werden.

Aufgrund dem Fokus auf mobile Roboter und RGB-D Daten in Kombination mit Echtzeitfähigkeit durch schnelle Berechnung wurde das VFH für den Einsatz in dieser Arbeit gewählt. Nach den Autoren kann durch eine VFH Signatur ein Objekt und dessen Pose erkannt werden. Die Posenerkennung ist dabei genau genug für die Manipulation des Objekts durch Roboter und ist schnell genug, um

²²<https://github.com/PointCloudLibrary/pcl/wiki/Overview-and-Comparison-of-Features>

in Echtzeitsystemen eingesetzt zu werden. Zusätzlich wurde der VFH Deskriptor speziell für den Einsatz mit Tiefenkameras erstellt und kann dadurch auch zuverlässig mit Daten umgehen, die viel Rauschen beinhalten und Lücken in den Tiefeninformationen beinhalten. Der Deskriptor liefert ein einzelnes Histogramm für alle Punkte in der Eingabe-Punktwolke. Daher ist für die Objekterkennung in größeren Szenen eine a priori Segmentierung notwendig. So kann wie oben beschrieben eine Punktwolke in mehrere Objekte durch Segmentierung unterteilt werden, um schließlich jedem Segment ein Histogramm zuzuweisen. Um in später aufgenommenen Punktwolken die Objekte wieder zu erkennen, können die Histogramme der Objekte verglichen werden. Abbildung 4.18 zeigt das Histogramm eines Ausschnitts der Ground Truth Punktwolke aus Abbildung 4.17. Bei dem Ausschnitt handelt es sich um die blaue Kiste unten links im Bild.

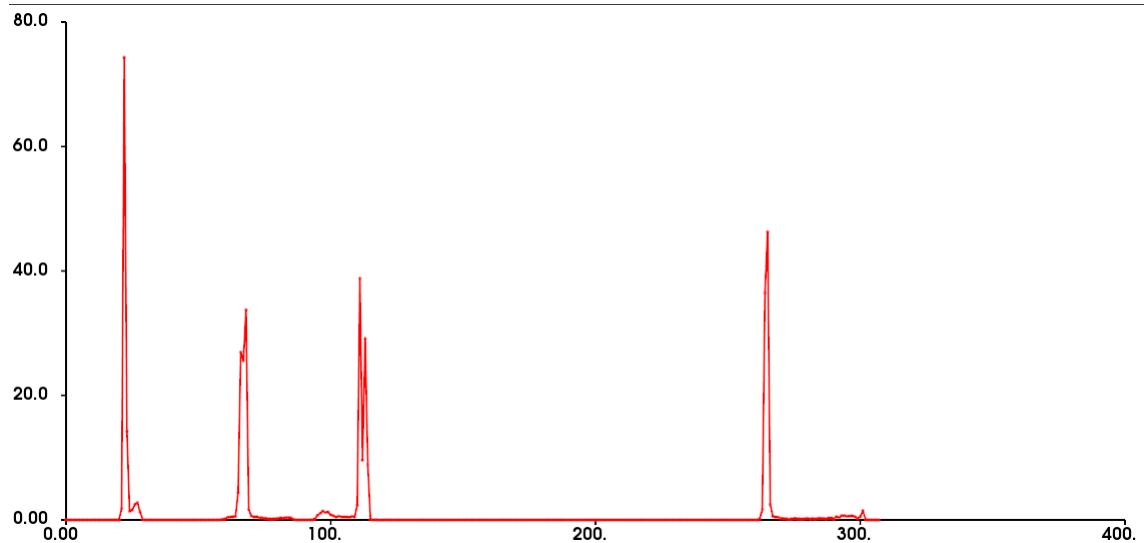


Abbildung 4.18: Das VFH von einem segmentierten Objekt aus einer Punktwolke.

Um die Verwendung von VFH zur Objekterkennung zu ermöglichen, wurde eine Punktwolke anhand des oben beschriebenen Algorithmus zur Segmentierung verwendet. Die segmentierte Punktwolke ist in Abbildung 4.19 dargestellt. Die Punktwolke stammt aus der erstellten Unity Simulation und stellt die Szene der in Abbildung 4.17 gezeigten Ground Truth dar, der Roboter wurde jedoch nach vorne bewegt.

Um die VFH zu vergleichen wurden die Funktionen der PCL verwendet, darunter auch Ausschnitte aus der PCL Dokumentation für VFH Deskriptoren²³. In Codeausschnitt 5 sind die verwendeten Funktionen in C++ für das Erstellen und Vergleichen der VFHs zu sehen. Die Funktionen sind in diesem Ausschnitt ungefähr nach der Reihenfolge der notwendigen Schritte sortiert. Die Befehle zur

²³https://pcl.readthedocs.io/projects/tutorials/en/latest/vfh_recognition.html

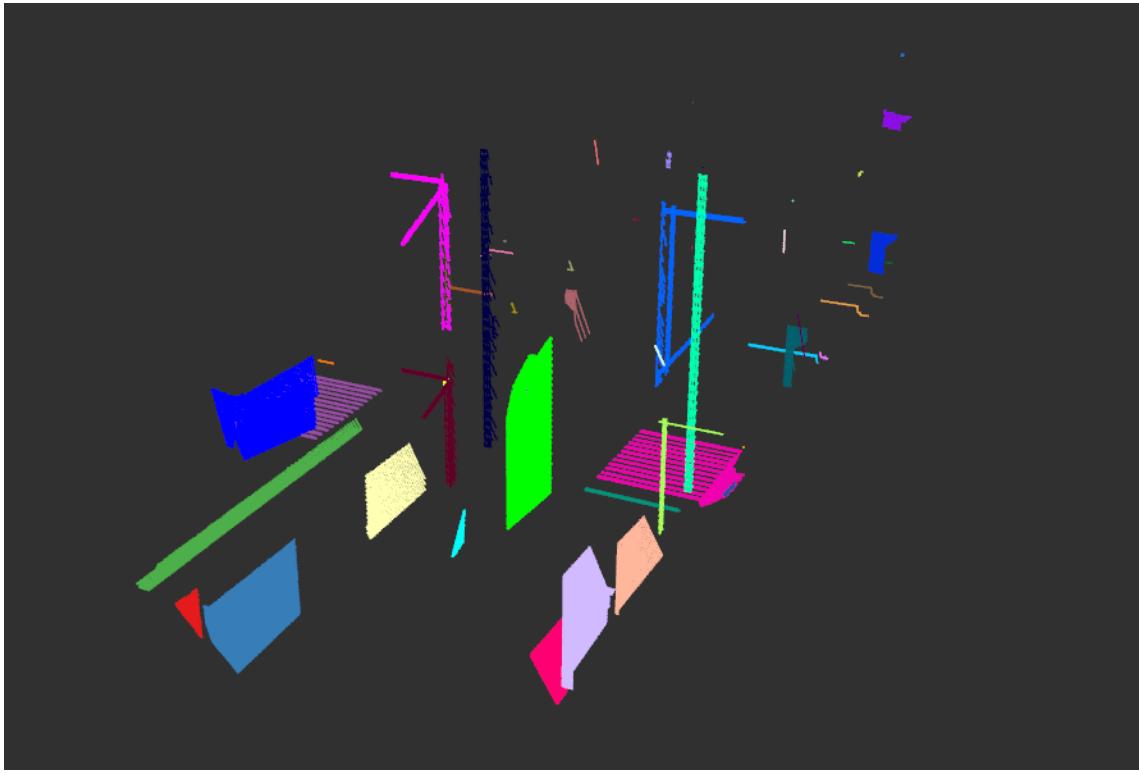


Abbildung 4.19: Die für die Objekterkennung verwendete segmentierte Punktwolke.

Zuordnung von Namespaces wie `pcl::PointCloud<...>` wurden hier zur besseren Lesbarkeit entfernt. In der ersten Zeile wird zusätzlich der Typ `vfh_model` definiert, der aus einem Paar aus einem Integer und einem Float-Vektor besteht. Der Integer Wert stellt dabei eine Kennzahl dar, mit der das Histogramm später identifiziert werden kann. Der Float-Vektor bildet das Histogramm, welches durch 308 Kommazahlen repräsentiert werden kann. Die PCL verfügt zwar über den Typ `VFHSignature308`, die später verwendeten Funktionen benötigen die Histogramme jedoch in Form von primitiven Typen.

Der erste Schritt ist das Importieren der Punktwolken aus den gespeicherten Dateien im .pcl Format über die Funktionen `read_pointcloud` und `read_labeled_pointcloud` eingelesen und in `PointCloud` Objekte der PCL umgewandelt. Anschließend werden aus den Punktwolken die Segmente anhand der Label gelesen und pro Punktwolke in einem Vektor gespeichert. Diese Vektoren beinhalten schließlich jeweils alle Segmente einer Punktwolke, ebenfalls in Form eines `PointCloud` Objekts. In diesem Beispiel beinhaltete der Vektor der Ground Truth Wolke 52 Elemente, der Vektor der Testwolke 34 Elemente.

Der zweite Schritt besteht daraus, die Histogramme aus den Segmenten der Punktwolken zu berechnen. Die PCL verfügt bereits über die Klasse `VFHEstima-`

tion. In der `compute_vfhs` Funktion wird anhand dieser Klasse eines einzelnen Segments im Format einer Punktfolge mit Punkten vom Typ `VFHSignature308` berechnet. In der Funktion `get_vfh_float_model_vector` wird das Histogramm in einen Float-Vektor umgewandelt. Hierfür werden lediglich alle Punkte der eben erstellten Punktfolge in einen leeren Vektor der Größe 308 hinzugefügt.

Im dritten Schritt werden in der Funktion `vfh_compare` Segmente der Ground Truth Wolke und der Testwolke, hier *query* genannt, paarweise miteinander verglichen. Um die Segmente später zur Visualisierung und zur Evaluation der Ergebnisse identifizieren zu können, werden die Vektoren der Segmente zusammen mit einer Zahl in dem vorher definierten `vfh_model` Typ gespeichert. Anschließend wird der Vektor der Ground Truth Segmente in eine FLANN Matrix umgewandelt. Schließlich wird der kd-tree in der Funktion `nearestKSearch` aufgebaut. Diese Funktion wurde direkt aus der PCL Dokumentation *Cluster Recognition and 6DOF Pose Estimation using VFH descriptors* aus dem Abschnitt *Testing* (Zeile 63 bis 82) übernommen. Als Messung für die Distanz zwischen den Histogrammen wurde die Chi-square Distanz [chi08] verwendet. Der Parameter K wurde mit dem Wert 6 initialisiert, als Threshold zuerst eine Distanz von 100 gewählt.

Abbildung 4.20 zeigt eine Visualisierung der sechs Segmente der Testwolke zu entsprechenden Segmenten der Ground Truth Wolke. Die jeweilige Distanz aus dem kd-tree ist neben jeden Segment entsprechend auf die nächste Ganzzahl gerundet abgebildet. Die sechs besten Werte verfügen über Distanzen zwischen 30 und 100. Das Segment mit der Distanz 98 besteht jedoch nur aus wenigen Punkten und wird durch die Koordinatenstrahlen verdeckt. Die anderen fünf sichtbaren Segmente wurden korrekt erkannt und stimmen mit der Ground Truth überein. Drei weitere Elemente, die sich nicht auf der Abbildung befinden, stimmen ebenfalls mit der Ground Truth überein. Die entsprechenden Distanzwerte dieser drei Segmente betragen 101, 107 und 119. Alle weiteren Segmente weisen minimale Distanzen zur Ground Truth zwischen 187 und 388 auf. Die gesamten Ergebnisse befinden sich im Anhang unter A1.

Der Algorithmus ist Echtzeit-geeignet, ein kompletter Durchlauf inklusive Import der Punktfolgen, Aufbau des kd-trees und Visualisierung benötigte zwischen einer und zwei Sekunden, je nach Komplexität der Punktfolgen und Anzahl der Segmente.

4.3 Diskussion der Ergebnisse

Im Zuge dieser Arbeit wurde ein neuer Ansatz für eine Simulation für Long-Term visual SLAM Systeme aufbauend auf dem Unity Robotics Hub erstellt. Die Simulation bietet eine performante und leicht zu bedienende Alternative zu bereits

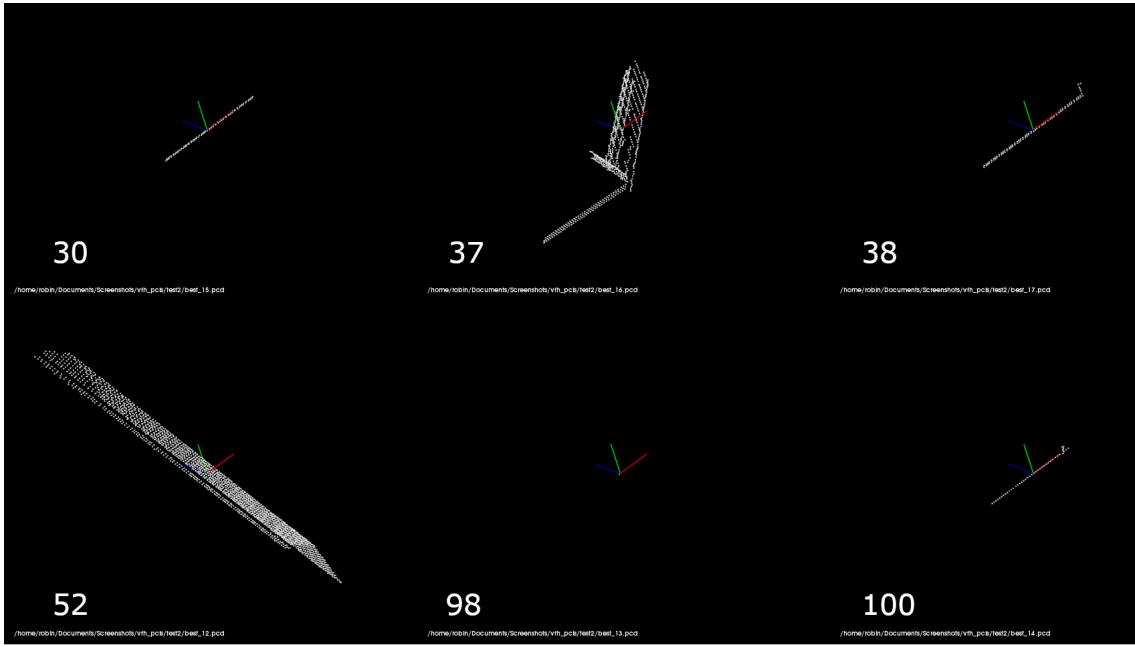


Abbildung 4.20: Die sechs Segmente mit den niedrigsten Distanzen zur Ground Truth.

bestehenden Robotersimulationen, besonders für die Anwendung von mobilen Roboter in indoor Umgebungen mit Fokus auf nicht stationären Umgebungen. Sie erfüllt die in Kapitel 2.4 und Kapitel 4.1.1 erläuterten Anforderungen an eine Robotersimulation. Dennoch bietet die Simulation noch Limitationen, wie in Kapitel 4.1.5 diskutiert wurde. Trotz der Limitationen ist die Simulation bereits im Fraunhofer IPA im Einsatz und wird dort voraussichtlich verbessert und weiterentwickelt.

Des weiteren wurde die Kombination aus einer Supervoxel Segmentierung für 3D Punktwolken in Kombination mit VFH Deskriptoren und einem kd-tree zur Objekterkennung untersucht. Die Ergebnisse zeigen, dass einige der Segmente mit Hilfe der VFH Deskriptoren in einer zweiten Punktwolke wieder erkannt wurden. Die Performance der Segmentierung zusammen mit der nearest-neighbour Suche würde zwar vermutlich auf mobilen Robotern einsetzbar sein, ist voraussichtlich jedoch nicht Echtzeitfähig. Daher wäre es eine Anwendungsmöglichkeit, die besprochenen Algorithmen im globalen Planer eines SLAM Systems einzusetzen. So könnten einige wichtige Objekte, die von dem System erkannt werden, über mehrere Datenpunkte hinweg gespeichert werden.

5 Fazit

5.1 Zusammenfassung

In dieser Arbeit wurde eine neue Simulation zum Entwickeln und Testen von Verfahren, die auf mobilen Roboter in nicht-stationären Umgebungen eingesetzt werden sollen, vorgestellt. Die Simulation verwendet die Unity Engine und baut auf dem Unity Robotics Hub auf. Der Fokus wurde dabei auf Performanz und einfache Bedienbarkeit gesetzt. Es besteht eine feste Integration mit ROS, wobei die Simulation über einen einzelnen Punkt über TCP mit ROS kommuniziert und dabei einen realen Roboter nachahmt. Im Zuge der Arbeit wurden einige Kriterien an eine Robotersimulation aufgestellt, welche schließlich auch größtenteils erfüllt wurden. So werden bisher Farb- und Tiefenkameras und 3D-Laserscanner unterstützt, jedoch keine weiteren Sensoren. Die Simulation kommt derzeit beim Fraunhofer IPA zum Einsatz und wird dort voraussichtlich in Zukunft weiterentwickelt.

Zudem wurden in der Arbeit Algorithmen zur Segmentierung und zum Feature Matching in 3D Punktwolken zum Einsatz in mobilen Robotern mit RGB-D Kameras untersucht. Der Ausgewählte Algorithmus zur Supervoxel Segmentierung lieferte gute Ergebnisse, ist mit einer Frequenz der Ergebnisse pro Punktwolke von 0.402 Hz jedoch nur bedingt echtzeitfähig. Das Feature Matching durch VFH und kd-trees lieferte zufriedenstellende Ergebnisse, kann jedoch nicht allein oder unmodifiziert zur Objekterkennung in LT-SLAM Systemen eingesetzt werden. So stimmten die gefundenen Objekte unter einem Distanz-Threshold von 100 mit der Ground Truth überein, jedoch wurde für den Großteil der Segmente kein passender Kandidat gefunden. Insgesamt bietet die Kombination der Supervoxel Segmentierung und VFH Features Potential im Feld der Bildverarbeitung in drei Dimensionen, müssen jedoch für den Einsatz in Robotern noch weiterentwickelt und angepasst werden.

5.2 Ausblick

Wie bereits angesprochen wird die im Zuge der Arbeit erstellte Unity Simulation voraussichtlich weiterentwickelt. Die Möglichkeiten hierfür sind breit gefächert. So könnten weitere Sensortypen wie 2D Laserscanner, GPS oder Inkrementalgeber zur Odometriemessung implementiert werden oder aber bestehende simulierte Sensoren verbessert werden. So könnten die generierten Daten durch Simulation von Messfehlern wie rauschen realistischere Ergebnisse erzielen. Auch das Hinzufügen von neuen Aktoren wie Greifarmen oder weiteren Methoden zur Fortbewegung wie Beine oder Rotoren wäre sinnvoll. Außerdem könnten weitere Arten von Umgebungen und Szenarien erstellt werden, um beispielsweise Agrar- oder Unterwasserroboter zu simulieren. Des Weiteren werden Anwendungen, die auf mehrere Roboter in Zusammenarbeit ausgelegt sind, beliebter und könnten von der Simulation unterstützt werden.

Hinsichtlich der Objekterkennung könnte die Performanz der Algorithmen verbessert werden. So könnte beispielsweise die Nutzung einer GPU für die repetitiven Berechnungen zum Vergleich der Punkte die Geschwindigkeit erheblich erhöhen. Außerdem wären weitere Kombinationen aus Art der Segmentierung und den verwendeten Feature Deskriptoren möglich. So könnten statt der Supervoxel Segmentierung beispielsweise eine regionsbasierte Segmentierung verwendet werden und statt den VFH könnten auch NARF Features gute Ergebnisse erzielen. Auch Ansätze, die auf Verfahren des maschinellen Lernens basieren, wirken sehr vielversprechend und könnten mit den in dieser Arbeit besprochenen Ansätzen gute Ergebnisse liefern.

Literaturverzeichnis

- [ABL17] Hatem Alismail, Brett Browning, and Simon Lucey. Photometric bundle adjustment for vision-based slam. In Shang-Hong Lai, Vincent Lepetit, Ko Nishino, and Yoichi Sato, editors, *Computer Vision – ACCV 2016*, pages 324–341, Cham, 2017. Springer International Publishing.
- [ATC⁺05] Dragomir Anguelov, B Taskar, Vassil Chatalbashev, Daphne Koller, Dinkar Gupta, Jeremy Heitz, and Andrew Ng. Discriminative learning of markov random fields for segmentation of 3d scan data. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 169–176. IEEE, 2005.
- [CCC⁺16] Cesar Cadena, Luca Carlone, Henry Carrillo, Yasir Latif, Davide Scaramuzza, Jose Neira, Ian Reid, and John J Leonard. Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age. *IEEE Transactions on Robotics*, 32(6):1309–1332, 2016.
- [CER⁺21] Carlos Campos, Richard Elvira, Juan J. Gómez Rodríguez, José M. M. Montiel, and Juan D. Tardós. Orb-slam3: An accurate open-source library for visual, visual-inertial, and multimap slam. *IEEE Transactions on Robotics*, pages 1–17, 2021.
- [chi08] *Chi-Square Distance*, pages 68–70. Springer New York, New York, NY, 2008.
- [CPATT10] Patricio Castillo-Pizarro, Tomas V. Arredondo, and Miguel Torres-Torriti. Introductory survey to open-source mobile robot simulation software. In *2010 Latin American Robotics Symposium and Intelligent Robotics Meeting*, pages 150–155, 2010.
- [Dör20] Stefan Dörr. *Cloud-based cooperative long-term SLAM for mobile robots in industrial applications*. Stuttgart: Fraunhofer Verlag, 2020.

- [FB81] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.
- [FJKL13] Maurice Fallon, Hordur Johannsson, Michael Kaess, and John J Leonard. The mit stata center dataset. *The International Journal of Robotics Research*, 32(14):1695–1699, 2013.
- [FPRARM15] Jorge Fuentes-Pacheco, José Ruiz-Ascencio, and Juan Manuel Rendón-Mancha. Visual simultaneous localization and mapping: a survey. *Artificial Intelligence Review*, 43(1):55–81, 2015.
- [FTS15] J Fosel, K Tuyls, and J Sturm. 2D-SDF-SLAM: A signed distance function based SLAM frontend for laser scanners. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1949–1955, 2015.
- [GLT12] Dorian Gálvez-López and J. D. Tardós. Bags of binary words for fast place recognition in image sequences. *IEEE Transactions on Robotics*, 28(5):1188–1197, October 2012.
- [HEH11] Derek Hoiem, Alexei A Efros, and Martial Hebert. Recovering Occlusion Boundaries from an Image. *International Journal of Computer Vision*, 91(3):328–346, 2011.
- [HLN12] Joachim Hertzberg, Kai Lingemann, and Andreas Nüchter. *Mobile Roboter : Eine Einführung aus Sicht der Informatik / von Joachim Hertzberg, Kai Lingemann, Andreas Nüchter*. eXamen.press. Springer Berlin Heidelberg, 2012.
- [HW11] M Hentschel and B Wagner. An Adaptive Memory Model for Long-Term Navigation of Autonomous Mobile Robots. *Journal of Robotics*, 2011:506245, 2011.
- [HWLW08] Bernhard Hofmann-Wellenhof, Herbert Lichtenegger, and Elmar Wasle. *GNSS - Global Navigation Satellite Systems : GPS, GLONASS, Galileo and more / Bernhard Hofmann-Wellenhof; Herbert Lichtenegger; Elmar Wasle*. Springer, 2008.
- [KB09] Kurt Konolige and James Bowman. Towards lifelong visual maps. *2009 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2009*, pages 1156–1163, 2009.

- [KFS⁺15] Ralf Kittmann, Tim Fröhlich, Johannes Schäfer, Ulrich Reiser, Florian Weißhardt, and Andreas Haug. Let me introduce myself: I am care-o-bot 4, a gentleman robot. In *Mensch und Computer 2015–Tagungsband*, pages 223–232. De Gruyter, 2015.
- [KH04] N. Koenig and A. Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*, volume 3, pages 2149–2154 vol.3, 2004.
- [Kon19] Anna Konrad. Simulation of mobile robots with unity and ros : A case-study and a comparison with gazebo. Master’s thesis, University West, Division of Production Systems, 2019.
- [LM13] Mathieu Labbe and Francois Michaud. Appearance-based loop closure detection for online large-scale and long-term operation. *IEEE Transactions on Robotics*, 29(3):734–745, 2013.
- [LM19] Mathieu Labb  and Fran ois Michaud. RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2):416–446, 2019.
- [Low04] David G Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004.
- [LSP05] Svetlana Lazebnik, Cordelia Schmid, and Jean Ponce. A sparse texture representation using local affine regions. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 27(8):1265–1278, 2005.
- [MAT17] R Mur-Artal and J D Tard s. ORB-SLAM2: An Open-Source SLAM System for Monocular, Stereo, and RGB-D Cameras. *IEEE Transactions on Robotics*, 33(5):1255–1262, 2017.
- [MCM07] Mark Maimone, Yang Cheng, and Larry Matthies. Two years of visual odometry on the Mars Exploration Rovers. *Journal of Field Robotics*, 24(3):169–186, 2007.
- [MDC⁺14] Timothy Morris, Feras Dayoub, Peter Corke, Gordon Wyeth, and Ben Upcroft. Multiple map hypotheses for planning and navigating in non-stationary environments. In *2014 IEEE International Conference on Robotics and Automation (ICRA)*, pages 2765–2770, 2014.

- [MM21] Alexey Merzlyakov and Steve Macenski. A Comparison of Modern General-Purpose Visual SLAM Approaches. *arXiv e-prints*, page arXiv:2107.07589, July 2021.
- [NL13] Anh Nguyen and Bac Le. 3D point cloud segmentation: A survey. *IEEE Conference on Robotics, Automation and Mechatronics, RAM - Proceedings*, pages 225–230, 2013.
- [NPRC17] Farzan M. Noori, David Portugal, Rui P. Rocha, and Micael S. Couceiro. On 3d simulators for multi-robot systems in ros: Morse or gazebo? In *2017 IEEE International Symposium on Safety, Security and Rescue Robotics (SSRR)*, pages 19–24, 2017.
- [PASW13] Jeremie Papon, Alexey Abramov, Markus Schoeler, and Florentin Wörgötter. Voxel cloud connectivity segmentation - supervoxels for point clouds. In *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 2027–2034, 2013.
- [RBTH10] Radu Bogdan Rusu, Gary Bradski, Romain Thibaux, and John Hsu. Fast 3d recognition and pose using the viewpoint feature histogram. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 2155–2162, 2010.
- [Ric13] Andreas Richsfeld. Robust Object Detection for Robotics using Perceptual Organization in 2D and 3D. (October):101, 2013.
- [RRKB11] Ethan Rublee, Vincent Rabaud, Kurt Konolige, and Gary Bradski. ORB: An efficient alternative to SIFT or SURF. *Proceedings of the IEEE International Conference on Computer Vision*, pages 2564–2571, 2011.
- [RSF13] Eric Rohmer, Surya P. N. Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 1321–1326, 2013.
- [RVLA19] Francisco Rubio, Francisco Valero, and Carlos Llopis-Albert. A review of mobile robots: Concepts, methods, theoretical framework, and applications. *International Journal of Advanced Robotic Systems*, 16(2):1–22, 2019.
- [Sch21] Robin Connor Schramm. Visual SLAM in non-stationary environments: State-of-the-Art. In *Perceive (IT) : Informatics Inside, Frühling 2021 : Tagungsband*, pages 106–116, Reutlingen, 2021. Reutlingen University.

- [SEE⁺12] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.
- [Sha06] Ariel Shamir. Segmentation and shape extraction of 3d boundary meshes. In *Eurographics (State of the Art Reports)*, pages 137–149, 2006.
- [SK10] Bastian Steder and Kurt Konolige. NARF : 3D Range Image Features for Object Recognition. *October*, 215:11, 2010.
- [SK16] Bruno Siciliano and Oussama Khatib. *Springer handbook of robotics*. Springer, Berlin, Heidelberg, 2016.
- [SLZ⁺20] X Shi, D Li, P Zhao, Q Tian, Y Tian, Q Long, C Zhu, J Song, F Qiao, L Song, Y Guo, Z Wang, Y Zhang, B Qin, W Yang, F Wang, R H M Chan, and Q She. Are We Ready for Service Robots? The OpenLORIS-Scene Datasets for Lifelong SLAM. In *2020 IEEE International Conference on Robotics and Automation (ICRA)*, pages 3139–3145, 2020.
- [SNS11] Roland Siegwart, Illah Reza Nourbakhsh, and Davide Scaramuzza. *Introduction to autonomous mobile robots*. MIT press, 2011.
- [SSS19] Shinya Sumikura, Mikiya Shibuya, and Ken Sakurada. OpenVSLAM: A versatile visual SLAM framework. *MM 2019 - Proceedings of the 27th ACM International Conference on Multimedia*, pages 2292–2295, 2019.
- [SZ03] Josef Sivic and Andrew Zisserman. Video google: A text retrieval approach to object matching in videos. In *Computer Vision, IEEE International Conference on*, volume 3, pages 1470–1470. IEEE Computer Society, 2003.
- [TDCH21] Michal Tölgessy, Martin Dekan, Ľuboš Chovanec, and Peter Hubinský. Evaluation of the azure kinect and its comparison to kinect v1 and kinect v2. *Sensors (Switzerland)*, 21(2):1–25, 2021.
- [TY09] Tomoji Takasu and Akio Yasuda. Development of the low-cost RTK-GPS receiver with an open source program package RTKLIB. *International Symposium on GPS/GNSS*, pages 4–6, 2009.

- [VTS17] Francesco Verdoja, Diego Thomas, and Akihiro Sugimoto. Fast 3D point cloud segmentation using supervoxels with geometry and color for 3D scene understanding. *Proceedings - IEEE International Conference on Multimedia and Expo*, pages 1285–1290, 2017.
- [YFB⁺20] Shiqiang Yang, Guohao Fan, Lele Bai, Rui Li, and Dexin Li. MGC-VSLAM: A meshing-based and geometric constraint VSLAM for dynamic indoor environments. *IEEE Access*, 8:81007–81021, 2020.
- [YN10] Yi Yang and Shawn Newsam. Bag-of-visual-words and spatial extensions for land-use classification. *GIS: Proceedings of the ACM International Symposium on Advances in Geographic Information Systems*, pages 270–279, 2010.

Anhang

```
1 <launch>
2   <arg name="urdf_file" default="$(find xacro)/xacro.py
3     '$(find cob_hw_config)
4       /robots/cob4-18/urdf/cob4-18.urdf.xacro'"/>
5   <param name="robot_description" command="$(arg urdf_file)" />
6
7   <rosparam file="$(find non_stationary_slam)/config/params.yaml"
8     command="load"/>
9
10  <arg name="ROS_IP_ARG" value="172.21.6.93"/>
11  <env name="ROS_IP" value="172.21.6.93"/>
12  <env name="ROS_HOSTNAME" value="$(arg ROS_IP_ARG)"/>
13
14  <param name="ROS_IP" type="str" value="$(arg ROS_IP_ARG)" />
15  <param name="ROS_TCP_PORT" type="int" value="10000" />
16  <param name="TCP_NODE_NAME" type="str" value="TCPServer" />
17
18  <node name="robot_state_publisher" pkg="robot_state_publisher"
19    type="robot_state_publisher" >
20    <param name="use_tf_static" value="false"/>
21  </node>
22
23  <group ns="ros_unity">
24    <node pkg="non_stationary_slam" name="server_endpoint"
25      type="server_endpoint.py"/>
26  </group>
27 </launch>
```

Codeausschnitt 1: ROS Launch Datei um den TCP Server für die Unity Simulation zu starten.

```
1  public class ColorCameraPublisher : RosCamera
2  {
3      void LateUpdate()
4      {
5          timeElapsed += Time.deltaTime;
6
7          if (timeElapsed > publishMessageFrequency)
8          {
9              ReadScreen();
10             timeElapsed = 0;
11         }
12     }
13
14     // Flips Render Texture and requests the image from GPU
15     public void ReadScreen()
16     {
17         // Flip Image vertically for ROS coordinates
18         Graphics.Blit(renderTexture, renderTextureCopy,
19                         new Vector2(1, -1), new Vector2(0, 1));
20         RenderTexture.active = renderTextureCopy;
21
22         // read screen from GPU
23         texture2D.ReadPixels(screenCaptureRect, 0, 0);
24         message.data = GetRawTextureData(texture2D);
25
26         // send message
27         rosConnector.Send(topic, message);
28         rosConnector.Send(cameraInfo.topic, cameraInfo.message);
29     }
30 }
```

Codeausschnitt 2: Lesen der Bilddaten von Unity aus der Grafikkarte und das Senden an ROS.

```

1 Shader "Postprocessing_depth"
2 {
3     Pass{
4         CGPROGRAM
5             #include "UnityCG.cginc"
6             #pragma vertex vert
7             #pragma fragment frag
8
9             sampler2D _CameraDepthTexture;
10
11         // Vertex shader
12         v2f vert(appdata v)
13         {
14             v2f o;
15             // convert the vertex positions from
16             // object space to clip space so they can be rendered
17             o.position = UnityObjectToClipPos(v.vertex);
18             o.uv = v.uv;
19             return o;
20         }
21
22         // fragment shader
23         fixed4 frag(v2f i) : SV_TARGET
24         {
25             // get depth from depth texture
26             float depth = tex2D(_CameraDepthTexture, i.uv).r;
27             // linear depth between camera and far clipping plane
28             depth = Linear01Depth(depth);
29             // depth as distance from camera in units, divided by 30
30             depth = (depth * _ProjectionParams.z) / 30;
31
32             return depth;
33         }
34     ENDCG
35 }
36 }
```

Codeausschnitt 3: Tiefenshader für die Unity Simulation.

```
1 header:
2     seq: 8
3     stamp:
4         secs: 1626963697
5         nsecs: 792423963
6     frame_id: "camera_bottom_link"
7     height: 480
8     width: 640
9     distortion_model: "plumb_bob"
10    D: [0.0, 0.0, 0.0, 0.0, 0.0]
11    K: [615.256103515625, 0.0, 322.1002502441406, 0.0, 615.34765625,
12        249.78524780273438, 0.0, 0.0, 1.0]
13    R: [1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0]
14    P: [615.256103515625, 0.0, 322.1002502441406, 0.0, 0.0,
15        615.34765625, 249.78524780273438, 0.0, 0.0, 0.0, 1.0, 0.0]
16    binning_x: 0
17    binning_y: 0
18    roi:
19        x_offset: 0
20        y_offset: 0
21        height: 0
22        width: 0
23        do_rectify: False
```

Codeausschnitt 4: Beispiel einer CameraInfo Message der simulierten Farbkamera in ROS.

```

1  typedef std::pair<int, std::vector<float> > vfh_model;
2
3
4  PointCloud<PointXYZ>::Ptr read_pointcloud(string path);
5
6  PointCloud<PointXYZL>::Ptr read_labeled_pointcloud(string path);
7  void save_pointcloud_file(
8      string path, PointCloud<PointXYZL>::Ptr cloud);
9
10 vector<PointCloud<PointXYZL>::Ptr> get_segmented_regions_pcl(
11     PointCloud<PointXYZL>::Ptr cloud);
12
13 PointCloud<VFHSsignature308>::Ptr compute_vfhs(
14     PointCloud<PointXYZL>::Ptr input_cloud);
15
16 vector<vfh_model> get_vfh_float_model_vector(
17     vector<PointCloud<PointXYZL>::Ptr> segmented_regions_vector);
18
19 void vfh_compare(
20     vector<PointCloud<PointXYZL>::Ptr> ground_truth,
21     vector<PointCloud<PointXYZL>::Ptr> query);
22
23 inline void nearestKSearch (
24     flann::Index<flann::ChiSquareDistance<float> &index,
25     const vfh_model &model, int k, flann::Matrix<int> &indices,
26     flann::Matrix<float> &distances)
27
28 void visualizeVfhs(PointCloud<VFHSsignature308>::Ptr vfhs);

```

Codeausschnitt 5: Funktionen, die für die Ermittlung und den Vergleich der VFH verwendet wurden.

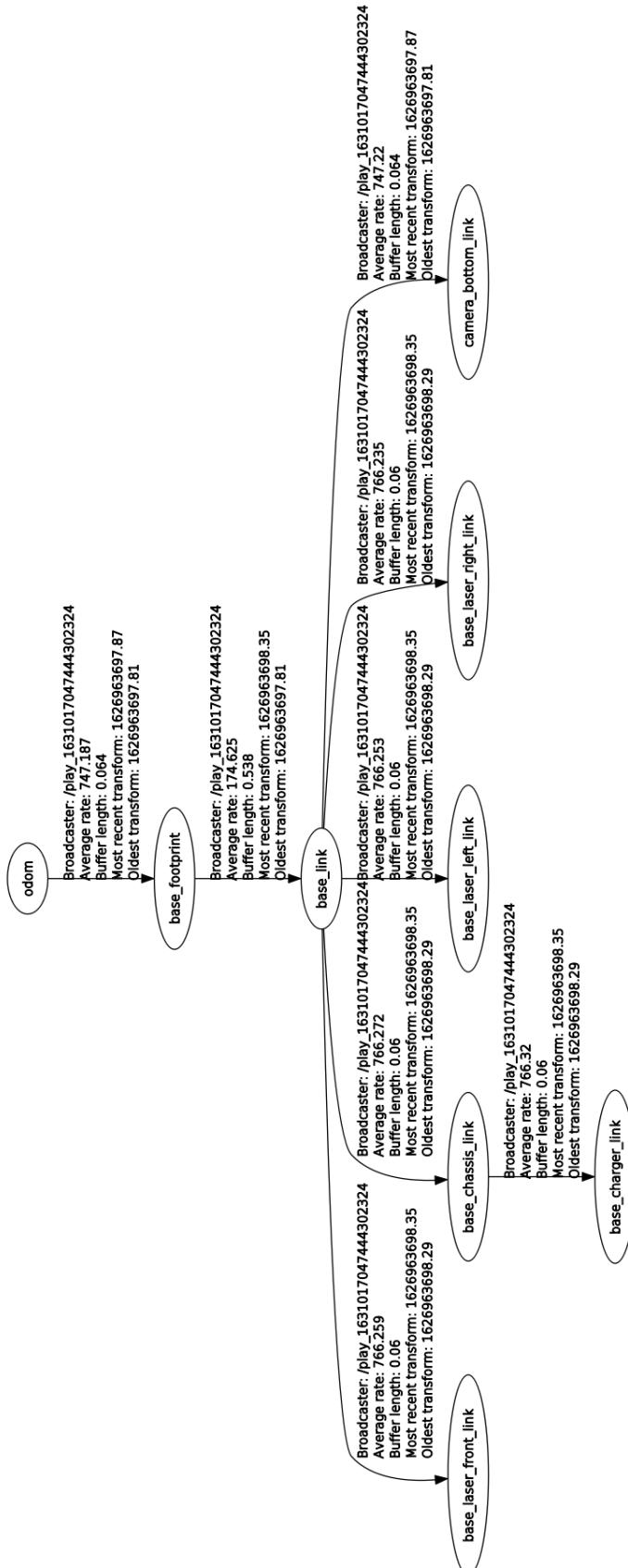


Abbildung A1: Ausschnitt des TF-Tree des care-o-bot 4 in der Unity Simulation.

Index Ground Truth	Index query	Minimale Distanz
0	5	361
1	17	107
2	19	191
3	11	273
4	44	205
5	27	101
6	23	100
7	1	226
8	41	290
9	17	119
10	12	277
11	42	198
12	0	37
13	8	38
14	5	243
15	6	187
16	17	30
17	28	220
18	1	221
19	15	98
20	11	269
21	0	385
22	5	273
23	12	211
24	28	254
25	39	244
26	45	238
27	52	195
28	24	388
29	43	204
30	4	200
31	52	220
32	29	228
33	23	52

Tabelle A1: K-d tree Distanzen durch VFH Matching.

Erklärung zur Abgabe einer Bachelor- / Master-Thesis

Ich versichere ehrenwörtlich, dass ich

- die abgegebene Thesis selbständig verfasst habe,
- alle benutzten Quellen und Hilfsmittel - dazu zählen auch sinngemäß übernommene Inhalte, leicht veränderte Inhalte sowie übersetzte Inhalte - in Quellenverzeichnissen, Fußnoten oder direkt bei Zitaten angegeben habe,
- alle wörtlichen und sinngemäßen Zitate von Textstücken, Tabellen, Grafiken, Fotos, Quellcode usw. aus fremden Quellen als solche gekennzeichnet und mit seitengenauen Quellenverweisen versehen habe,
- die von mir eingereichten Dokumente und Artefakte noch nicht in dieser oder ähnlicher Form einer anderen Kommission zur Prüfung vorgelegt wurden,
- alle nicht als Zitat gekennzeichneten Inhalte selbst erstellt habe und dass ich
- den „Leitfaden für gute wissenschaftliche Praxis im Studiengang MKI“¹ kenne und achte.

Mir ist bekannt, dass unmarkierte und unbelegte Zitate und Paraphrasen Plagiate sind und nicht als handwerkliche Fehler, sondern als eine Form vorsätzlicher Täuschung der Prüfer gelten, da fremde Gedanken als eigene Gedanken vorgetäuscht werden mit dem Ziel der Erschleichung einer besseren Leistungsbewertung.

Mir ist bekannt, dass Plagiarismus die Standards guter wissenschaftlicher Praxis, die Regeln des Studiengangs Medien- und Kommunikationsinformatik, die Studien- und Prüfungsordnung der Hochschule Reutlingen (§ 10 Täuschung und Ordnungsverstoß) und das Landeshochschulgesetz von Baden-Württemberg (§ 3 Wissenschaftliche Redlichkeit Abs. 5, § 62 Exmatrikulation Abs. 3) missachtet und seine studienrechtlichen Folgen vom Nichtbestehen bis zur Exmatrikulation reichen.

Mir ist auch bekannt, dass Plagiate sogar das Urheberrechtsgesetz (§ 51 Zitate, § 63 Quellenangabe, § 106 Unerlaubte Verwertung urheberrechtlich geschützter Werke) verletzen und zivil- und strafrechtliche Folgen nach sich ziehen können.

Nachname: Schramm

Vorname: Robin Connor

Matrikelnummer: 761392

Datum: 20.09.2021

Unterschrift: R. Schramm

¹ <https://bscwserv.reutlingen-university.de/bscw/bscw.cgi/d2871027/GWP.pdf>