

Technische Hochschule Nürnberg Georg Simon Ohm

Abschlussarbeit in der Fakultät EFI

Autonome Exploration unbekannter Umgebungen unter Berücksichtigung negativer Hindernisse

Leon Pinzner

Fichtenstraße 13

90530 Wendelstein

Betreuer: Prof. Dr. May (TH Nürnberg)
Prof. Dr. Arndt (TH Nürnberg)

Nürnberg, den 01. Oktober 2023

Hinweis: Diese Erklärung ist in alle Exemplare der Abschlussarbeit fest einzubinden. (Keine Spiralbindung)

Prüfungsrechtliche Erklärung der/des Studierenden

Angaben des bzw. der Studierenden:

Name: Pinzner

Vorname: Leon

Matrikel-Nr.: 3195886

Fakultät: efi

Studiengang: MSY

Semester: 3

Titel der Abschlussarbeit:

Autonome Exploration unbekannter Umgebungen unter Berücksichtigung negativer Hindernisse

Ich versichere, dass ich die Arbeit selbständig verfasst, nicht anderweitig für Prüfungszwecke vorgelegt, alle benutzten Quellen und Hilfsmittel angegeben sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe.

Wendelstein, 30.09.2023



Ort, Datum, Unterschrift Studierende/Studierender

Erklärung der/des Studierenden zur Veröffentlichung der vorstehend bezeichneten Abschlussarbeit

Die Entscheidung über die vollständige oder auszugsweise Veröffentlichung der Abschlussarbeit liegt grundsätzlich erst einmal allein in der Zuständigkeit der/des studentischen Verfasserin/Verfassers. Nach dem Urheberrechtsgesetz (UrhG) erwirbt die Verfasserin/der Verfasser einer Abschlussarbeit mit Anfertigung ihrer/seiner Arbeit das alleinige Urheberrecht und grundsätzlich auch die hieraus resultierenden Nutzungsrechte wie z.B. Erstveröffentlichung (§ 12 UrhG), Verbreitung (§ 17 UrhG), Vervielfältigung (§ 16 UrhG), Online-Nutzung usw., also alle Rechte, die die nicht-kommerzielle oder kommerzielle Verwertung betreffen.

Die Hochschule und deren Beschäftigte werden Abschlussarbeiten oder Teile davon nicht ohne Zustimmung der/des studentischen Verfasserin/Verfassers veröffentlichen, insbesondere nicht öffentlich zugänglich in die Bibliothek der Hochschule einstellen.

Hiermit genehmige ich, wenn und soweit keine entgegenstehenden Vereinbarungen mit Dritten getroffen worden sind,
 genehmige ich nicht,

dass die oben genannte Abschlussarbeit durch die Technische Hochschule Nürnberg Georg Simon Ohm, ggf. nach Ablauf einer mittels eines auf der Abschlussarbeit aufgebrachten Sperrvermerks kenntlich gemachten Sperrfrist

von Jahren (0 - 5 Jahren ab Datum der Abgabe der Arbeit),

der Öffentlichkeit zugänglich gemacht wird. Im Falle der Genehmigung erfolgt diese unwiderruflich; hierzu wird der Abschlussarbeit ein Exemplar im digitalisierten PDF-Format auf einem Datenträger beigefügt. Bestimmungen der jeweils geltenden Studien- und Prüfungsordnung über Art und Umfang der im Rahmen der Arbeit abzugebenden Exemplare und Materialien werden hierdurch nicht berührt.

Wendelstein, 30.09.2023



Ort, Datum, Unterschrift Studierende/Studierender

Datenschutz: Die Antragstellung ist regelmäßig mit der Speicherung und Verarbeitung der von Ihnen mitgeteilten Daten durch die Technische Hochschule Nürnberg Georg Simon Ohm verbunden. Weitere Informationen zum Umgang der Technischen Hochschule Nürnberg mit Ihren personenbezogenen Daten sind unter nachfolgendem Link abrufbar: <https://www.th-nuernberg.de/datenschutz/>

Danksagung

Zuallererst möchte ich mich bei Herrn Prof. Dr. Stefan May bedanken, der es mir ermöglichte, meine Masterarbeit im Labor für mobile Robotik zu schreiben.

Außerdem möchte ich mich bei Herrn Johannes Vollet und dem gesamten Team AutonOHM bedanken, die mir während dieser Arbeit mit ihrer Hilfsbereitschaft stets zur Seite standen und mir meine Fragen jederzeit beantworten konnten.

Zudem gilt mein Dank der Familie Wittman, die mir eine große Hilfe bei dem Bau eines Testparcours gewesen ist.

Schließlich gilt mein Dank meinen Freunden und meiner Familie, die mich bei der Erstellung dieser Arbeit und im gesamten Studium sehr unterstützt haben.

Inhaltsverzeichnis

| | |
|--|-----------|
| Abkürzungsverzeichnis | vi |
| 1 Einleitung | 1 |
| 2 Grundlagen | 3 |
| 2.1 ROS | 3 |
| 2.1.1 ROS-Packages | 4 |
| 2.1.2 Systemaufbau | 4 |
| 2.1.3 ROS-Messages | 5 |
| 2.1.4 Launch- & YAML-Files | 6 |
| 2.1.5 RViz | 6 |
| 2.1.6 tf2 | 6 |
| 2.1.7 ROS Navigation Stack | 7 |
| 2.2 Gazebo | 8 |
| 2.3 Kobuki | 8 |
| 2.4 ODROID-XU4 | 9 |
| 2.5 Schrödi | 10 |
| 2.6 Orbbec Astra Stereo S | 11 |
| 3 Implementierung in einer Simulation | 13 |
| 3.1 Roboter- und Parcoursauswahl | 13 |
| 3.2 Planung und Vorgehensweise | 15 |
| 3.3 Mapping | 16 |
| 3.3.1 costmap_2d | 16 |
| 3.3.2 grid_map | 20 |
| 3.3.3 elevation_mapping | 24 |
| 3.4 Erkennen negativer Hindernisse | 28 |
| 3.5 move_base | 34 |

| | |
|---|-----------|
| 3.6 explore_lite | 45 |
| 4 Testaufbau für Vorversuche | 50 |
| 4.1 Parcours | 50 |
| 4.1.1 Material | 50 |
| 4.1.2 Layout | 54 |
| 4.2 Tests und Ergebnisse | 56 |
| 4.2.1 Test Kobuki | 56 |
| 4.2.2 Test Mapping | 61 |
| 4.2.3 Test autonome Navigation | 65 |
| 4.2.4 Test autonome Exploration | 71 |
| 5 Teilnahme und Tests bei Wettbewerben | 74 |
| 5.1 Vorbereitung | 74 |
| 5.1.1 Hardware | 74 |
| 5.1.2 Software | 75 |
| 5.2 RoboCup German Open | 77 |
| 5.2.1 Vorbereitung und Training | 77 |
| 5.2.2 Wettbewerb | 79 |
| 5.2.3 Resümee | 80 |
| 5.3 RoboCup Weltmeisterschaft | 80 |
| 5.3.1 Vorbereitung und Training | 81 |
| 5.3.2 Wettbewerb | 81 |
| 5.3.3 Resümee | 83 |
| 6 Zusammenfassung und Ausblick | 84 |
| Literaturverzeichnis | 86 |
| Anhang | 90 |

Abkürzungsverzeichnis

| | |
|--------------|-----------------------------|
| ROS | Robot Operating System |
| YAML | YAML Ain Markup Language |
| RViz | ROS Visualization |
| tf | Transform Library |
| tf2 | Transform Library 2 |
| IMU | Inertial Measurement Unit |
| SSH | Secure Shell |
| Lidar | Light Detection and Ranging |

1 Einleitung

Die vorliegende Arbeit entstand im Labor für mobile Robotik der Fakultät Elektrotechnik, Feinwerktechnik und Informationstechnik an der Technischen Hochschule Nürnberg Georg Simon Ohm. Dieses Labor, das in Kooperation mit der Fakultät Maschinenbau besteht, widmet sich der Erforschung und Entwicklung verschiedener Aspekte der mobilen Robotik wie z. B. Lokalisations- und Kartografiealgorithmen, Explorationsstrategien oder der Entwicklung von robusten Multi-Roboterkontrollarchitekturen. [1]

Ein Teil der wissenschaftlichen Mitarbeiter, Professoren und Studenten des Labors bilden das Team AutonOHM. Dieses engagierte Team nimmt sowohl an nationalen als auch internationalen Wettbewerben im Bereich der mobilen Robotik teil. Auf internationaler Ebene ist das Team beim RoboCup aktiv, einem Wettbewerb, der seine Anfänge im Juli 1993 mit der Gründung der Robot J-League, einer japanischen Roboterfußballliga, hatte. Aufgrund des starken internationalen Interesses wurde der RoboCup ins Leben gerufen und fand erstmals 1997 in Nagoya, Japan statt. Seitdem hat sich der RoboCup, der verschiedene Disziplinen umfasst, zu einer bedeutsamen Plattform für Robotik-Enthusiasten entwickelt. Neben der populären Fußballliga gibt es auch die Rescue League, die sich auf die Entwicklung mobiler Rettungsroboter in zerstörten Umgebungen konzentriert. Diese Liga ermöglicht es Forschern, sich mit den komplexen Herausforderungen der mobilen Robotik auseinanderzusetzen und innovative Lösungen zu erarbeiten.

Auf nationaler Ebene nimmt das Team auch am deutschen Ableger des RoboCup teil, den RoboCup German Open, wo es unter anderem in der Rescue Robot League antritt. [2]

Wie in vielen anderen Bereichen der Robotik spielt auch in der mobilen Robotik Autonomie eine immer größere Rolle. Deshalb liegt der Fokus auch beim RoboCup immer mehr auf Autonomie. In zahlreichen Aufgabenstellungen werden zusätzliche Bewertungspunkte für die Umsetzung von Autonomie vergeben, wobei in einigen Fällen die strikte Vorgabe besteht, dass die Roboter ausschließlich autonom navigieren dürfen. Ein Beispiel dafür ist die Aufgabe des RoboCup namens „Avoid Holes“, bei der ein Parcours mit negativen Hindernissen bewältigt werden muss. In Abbildung 1 ist ein Beispiel für einen solchen

Parcours dargestellt. In dieser Herausforderung geht es darum, autonom durch den Parcours zu navigieren und dabei gleichzeitig eine Karte des Parcours zu erstellen.



Abbildung 1: Parcours mit negativen Hindernissen

Diese Arbeit widmet sich der Entwicklung und Erprobung von Softwarelösungen für den autonomen Betrieb von Rettungsrobotern. Besonderes Augenmerk liegt auf der Bewältigung des Parcours mit negativen Hindernissen, bei dem der Roboter eigenständig Hindernisse überwinden und vielfältige Aufgaben bewältigen muss. Im Fokus stehen hierbei die Entwicklung von Softwarelösungen für die autonome Navigation, Kartierung und Exploration. Diese Lösungen sollen nicht nur in Simulationstests, sondern auch in realen Testläufen auf dem Roboter selbst erfolgreich funktionieren.

Das übergeordnete Ziel dieser Arbeit besteht darin, die Fähigkeiten des Roboters im Rahmen des RoboCup-Wettbewerbs weiter zu verbessern und auszubauen, um den hohen Anforderungen des Wettbewerbs gerecht zu werden. Dieser Beitrag soll letztendlich zur Weiterentwicklung und Verbesserung der Rettungsrobotik beitragen.

In den kommenden Kapiteln werden die Entwicklungsschritte, Tests und Ergebnisse der implementierten Lösungen im Kontext eines Parcours mit negativen Hindernissen detailliert vorgestellt. Dabei werden sowohl technische Aspekte als auch praktische Erfahrungen während der Teilnahme an den Wettbewerben ausführlich behandelt. Zusätzlich werden mögliche Perspektiven für zukünftige Entwicklungen und Erweiterungen beleuchtet, um den stetig wachsenden Anforderungen dieses spannenden und anspruchsvollen Forschungsfeldes gerecht zu werden.

2 Grundlagen

In diesem Kapitel wird auf die Grundlagen eingegangen, welche zur Umsetzung der Arbeit benötigt werden. Von Seiten der Software werden das Roboter-Framework ROS und dessen Bestandteile thematisiert. Außerdem wird mit Gazebo eine Möglichkeit zur Simulation des Roboters in einer Testumgebung vorgestellt. Seitens der Hardware werden die Roboter näher beschrieben, die sowohl für Testzwecke als auch für den Einsatz beim RoboCup verwendet werden. Zuletzt wird noch die verwendete 3D-Kamera und die Gründe für ihre Verwendung erläutert.

Die Arbeit wird auf einem Rechner in einer Dual-Boot-Konfiguration mit den Betriebssystemen Ubuntu 20.04 und Ubuntu 18.04 durchgeführt. Ubuntu 18.04 wird verwendet, da diese Distribution auf dem Wettkampfroboter installiert ist. Der Standardsupport dieser Distribution endete allerdings im April 2023, weswegen im Hinblick auf die Zukunft hauptsächlich in Ubuntu 20.04 entwickelt und simuliert wird. [3]

2.1 ROS

ROS (Robot Operating System) ist ein flexibles und weitverbreitetes Open Source Framework für die Entwicklung und Steuerung von Robotern. Es bietet eine umfangreiche Sammlung von Tools, Bibliotheken und Middleware, die es ermöglichen, komplexe Roboteranwendungen zu entwickeln. ROS basiert auf einer verteilten Architektur, die es erlaubt, einzelne Komponenten eines Roboter-Systems unabhängig voneinander zu entwickeln und zu testen. Durch die Verwendung von ROS können Entwickler auf eine große Auswahl an vorgefertigten Funktionen zugreifen, was die Entwicklung von Robotikanwendungen beschleunigt. Zudem bietet ROS eine große Community von Entwicklern, die ihr Wissen und ihre Erfahrungen teilen, womit der Einstieg und die Weiterentwicklung erleichtert wird. [4]

2.1.1 ROS-Packages

Die modulare Struktur von ROS ermöglicht die Organisation von Robotikanwendungen in Form von Paketen. ROS-Packages sind eigenständige Einheiten, die spezifische Funktionen bereitstellen und zur Modulbildung und Wiederverwendung von Code beitragen. Jedes Paket kann mehrere Knoten enthalten, die einzelne Aufgaben oder Funktionen ausführen. Durch die Verwendung von ROS-Paketen können Entwickler ihre Anwendungen effizienter strukturieren, den Code wiederverwenden und die Entwicklungsgeschwindigkeit erhöhen. Darüber hinaus stellt die ROS-Community eine umfangreiche Sammlung von vorgefertigten ROS-Paketen zur Verfügung, die eine Vielzahl von Funktionalitäten abdecken, wie beispielsweise Bewegungssteuerung, Sensor- oder Bildverarbeitung. [5]

2.1.2 Systemaufbau

Das ROS-System besteht aus verschiedenen Komponenten die zusammenarbeiten, um die Funktionen des Roboters zu steuern. Zentrales Element ist der sogenannte ROS-Master, der als Vermittler zwischen den verschiedenen Knoten fungiert. Er verwaltet die Kommunikation und ermöglicht den Austausch von Daten zwischen den Knoten. Die Knoten (Nodes) können verschiedene Funktionen erfüllen, wie z. B. die Steuerung von Sensoren, die Verarbeitung von Informationen oder die Durchführung von Berechnungen. Die Kommunikation zwischen den Knoten erfolgt über verschiedene Mechanismen wie Topics, Services und Actions.

Topics sind benannte Datenströme, über die Knoten Nachrichten (Messages) austauschen können. Ein Knoten kann Nachrichten auf einem bestimmten Topic veröffentlichen (publish), während andere Knoten, die sich für dieses Topic interessieren, die Nachrichten abonnieren (subscribe) können. In Abbildung 2 wird die Kommunikation über ein Topic dargestellt.

Ein Service ermöglicht es einem Knoten, einen spezifischen Dienst anzubieten, auf den andere Knoten zugreifen können. Im Gegensatz zu Topics, die asynchron sind, ermöglichen Services eine synchrone Kommunikation, bei der ein Knoten eine Anfrage sendet und auf eine Antwort wartet.

Darüber hinaus gibt es Actions, die eine weitere Möglichkeit zur Kommunikation zwischen Knoten bieten. Actions sind ähnlich wie Services, jedoch speziell für langwierige Aufgaben konzipiert, bei denen Feedback über den Fortschritt wichtig ist. Sie erlauben es einem Knoten, eine Aufgabe zu starten und dabei den Fortschritt zu überwachen oder wenn

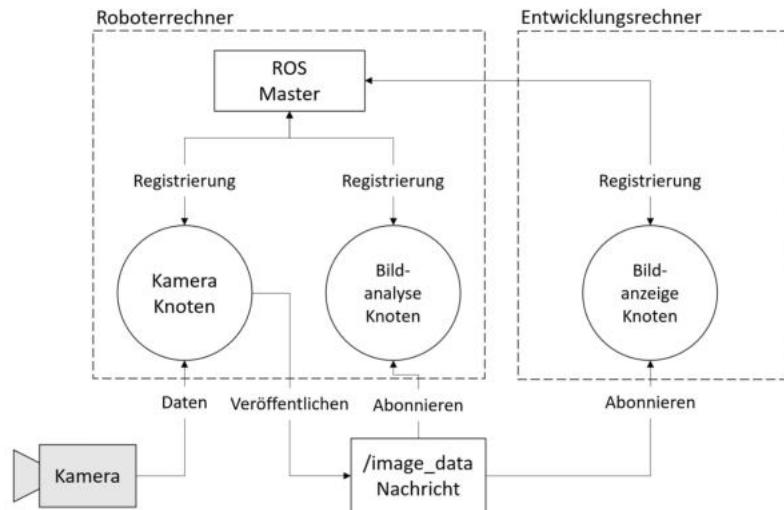


Abbildung 2: Beispiel Systemaufbau ROS [7]

nötig abzubrechen. Dies ist besonders nützlich in Szenarien, in denen Roboter komplexe Aktionen ausführen müssen, wie z. B. das Durchfahren eines Parcours oder das Greifen eines Objekts. [6]

2.1.3 ROS-Messages

ROS-Messages (ROS-Nachrichten) sind die grundlegenden Datenstrukturen, die von ROS zur Kommunikation zwischen verschiedenen Modulen oder Knoten verwendet werden. Sie ermöglichen den Austausch von Informationen über Topics, Services und Actions. ROS-Messages definieren den Datentyp, den eine Nachricht enthält, und legen damit das Format fest, in dem Informationen zwischen den Knoten ausgetauscht werden. Eine ROS-Message besteht aus einer festen Struktur von Feldern, die verschiedene Datentypen wie Zahlen, Zeichenketten, Arrays oder benutzerdefinierte Nachrichten enthalten können. Durch die Verwendung von ROS-Messages können verschiedene Komponenten eines Robotersystems, unabhängig davon, in welcher Programmiersprache sie geschrieben sind, miteinander kommunizieren. Dies erleichtert die Modularität, Wiederverwendbarkeit und Interoperabilität von Robotersoftware. [8]

2.1.4 Launch- & YAML-Files

Launch- und YAML-Files („YAML Ain't Markup Language“-Dateien) werden verwendet, um den Start und die Konfiguration von Robotikanwendungen zu erleichtern. Launch-Files sind XML-basierte Dateien, die es ermöglichen, mehrere ROS-Knoten gleichzeitig zu starten und zu konfigurieren. Durch die Verwendung von Launch-Files können komplexe ROS-Anwendungen mit nur einem Befehl gestartet werden.

Um die Konfiguration komplexer Knoten weiter zu vereinfachen, können Launch-Files YAML-Files aufrufen. YAML-Files sind Konfigurationsdateien, die dazu dienen, Parameter in einem übersichtlichen und leicht lesbaren Format zu speichern. [9, 10]

2.1.5 RViz

RViz (ROS Visualization) ist ein leistungsstarkes Visualisierungstool, welches für ROS verwendet wird. Es bietet die Möglichkeit, Daten aus ROS-Anwendungen in 3D-Visualisierungen darzustellen und zu analysieren. Mit RViz können komplexe Robotermodelle, Sensordaten, Karten und vieles mehr visualisiert werden. Es ermöglicht eine interaktive und immersive Darstellung der Umgebung, in der ein Roboter agiert. RViz unterstützt verschiedene Arten von Visualisierungen wie Punktwolken, Meshes, Pfade und Koordinatensysteme. Es ermöglicht auch das Hinzufügen interaktiver Steuerelemente zur Überwachung und Steuerung von Robotern in Echtzeit. RViz ist ein äußerst nützliches Werkzeug für die Entwicklung, Fehlersuche und Validierung von Robotikanwendungen. In Abbildung 3 ist ein Beispiel für die Visualisierung mehrerer Daten zu sehen. Im linken Bereich der Abbildung werden RGB-Bilddaten einer Tiefenkamera visualisiert. Diese Tiefenkamera liefert zusätzlich auch eine 3D-Punktwolke, welche auf der rechten Seite der Abbildung zu sehen ist. [11]

2.1.6 tf2

tf2 (Transform Library 2) ist eine Kernkomponente von ROS und ermöglicht die Verwaltung von Transformationen zwischen verschiedenen Koordinatensystemen. Es bietet eine flexible und effiziente Möglichkeit, Koordinatentransformationen zwischen verschiedenen Teilen eines Roboters oder zwischen Robotern und ihrer Umgebung zu verwalten. tf2 verwaltet die Transformationen in einem Baumstrukturenmodell, wodurch die Beziehungen zwischen den verschiedenen Koordinatensystemen hierarchisch organisiert werden.

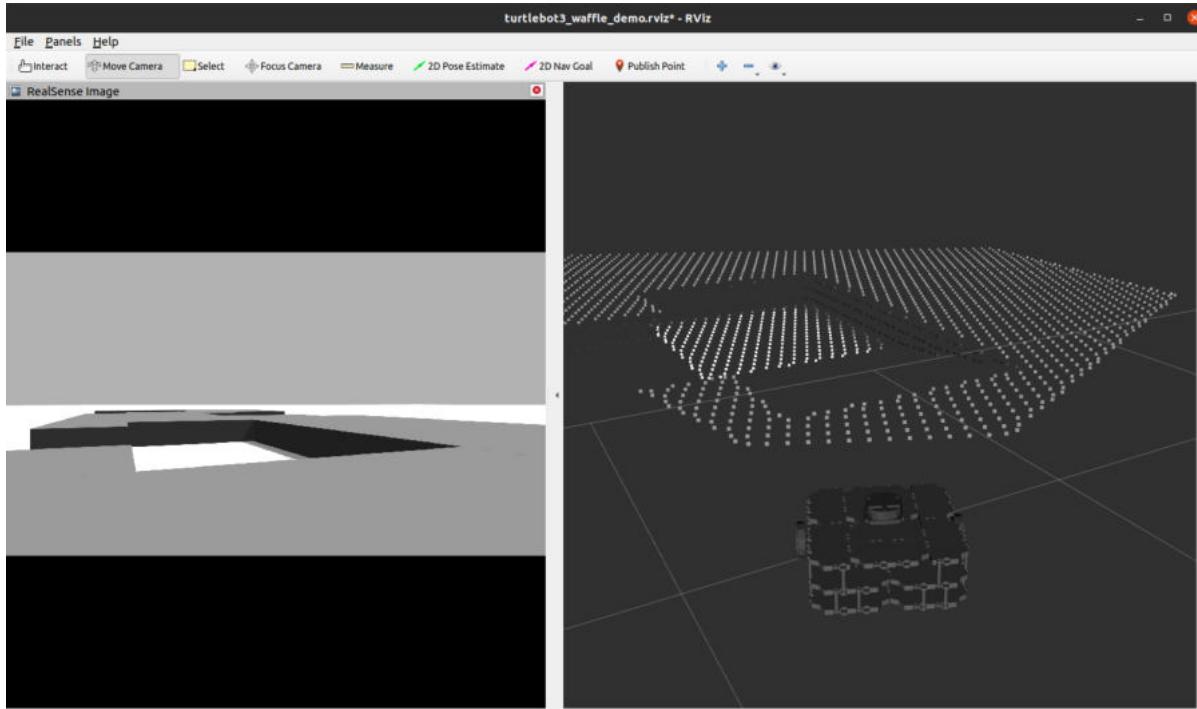


Abbildung 3: Beispiel für die Visualisierung verschiedener Daten in RViz

Durch die Nutzung von tf2 können ROS-Knoten die Positionen und Orientierungen von Objekten in einem gemeinsamen Bezugssystem berechnen und miteinander kommunizieren. Dies ermöglicht beispielsweise die präzise Steuerung von Robotern, die Fusion von Sensorinformationen aus verschiedenen Quellen oder die Navigation in einer dreidimensionalen Umgebung. [12]

In dieser Arbeit wird dennoch für alles, was mit diesem Paket zu tun hat, die Bezeichnung „tf“ verwendet. Dies geschieht aufgrund der Tatsache, dass Funktionen, Knoten oder Topics oft die Zahl „2“ in ihren Bezeichnungen auslassen, da sie aus einer alten Version der Bibliothek stammen.

2.1.7 ROS Navigation Stack

Der ROS Navigation Stack ist eine Sammlung von ROS-Paketen und Algorithmen, die zur Navigation von mobilen Robotern in komplexen Umgebungen entwickelt wurden. Der Navigation Stack ermöglicht es einem Roboter, autonom Hindernisse zu erkennen, Karten zu erstellen, Routen zu planen und sich sicher durch die Umgebung zu bewegen. Der Stack umfasst verschiedene Komponenten, wie z. B. den Map Server zur Bereitstellung von Karteninformationen, den Costmap Generator zur Erstellung einer Umgebungskarte und

den Path Planner zur Berechnung optimaler Pfade. Durch die Integration dieser Komponenten ermöglicht der ROS Navigation Stack eine zuverlässige und präzise autonome Navigation von Robotern. [13, 14]

2.2 Gazebo

Gazebo ist eine leistungsstarke Open-Source-Simulationsumgebung, die speziell für die Robotik entwickelt wurde. Diese Software ermöglicht die realistische Modellierung und Simulation von Robotern, Sensoren, Aktuatoren und ihrer physischen Umgebung. Gazebo bietet eine breite Palette von Funktionen, darunter 3D-Visualisierung, dynamische Physiksimulation und eine nahtlose Integration in ROS. Darüber hinaus stellt Gazebo eine umfangreiche Sammlung von vorgefertigten Modellen, Szenarien und Simulationswerkzeugen zur Verfügung. Diese Ressourcen erleichtern die Entwicklung und Validierung von Robotiksystemen erheblich. Aus diesen Gründen ist Gazebo eine hervorragende Wahl, um Robotikanwendungen vor der Durchführung von realen Tests in einer simulierten Umgebung zu evaluieren und zu optimieren.

Ein Beispiel für eine solche Simulation ist in Abbildung 9 zu sehen. Auf diese wird aber in einem späteren Kapitel noch genauer eingegangen. [15]

Für alle Simulationen, welche im Zusammenhang mit dieser Arbeit durchgeführt werden, wird Gazebo verwendet.

2.3 Kobuki

Der Kobuki-Roboter, zu sehen in Abbildung 4, ist ein mobiles Plattformrobotersystem, das speziell für den Einsatz in der Robotik entwickelt wurde. Er zeichnet sich durch seine kompakte Größe, Wendigkeit und Zuverlässigkeit aus. Der Kobuki-Roboter basiert auf einer differenziellen Antriebsplattform mit zwei Rädern und einem passiven Stabilisator an der Vorder- und Rückseite. Er verfügt über verschiedene Sensoren wie Taster oder ein IMU (Inertial Measurement Unit), die es ihm ermöglichen, seine Umgebung wahrzunehmen und sich darin zu bewegen. Der Kobuki-Roboter ist mit ROS kompatibel, was die Integration und Steuerung in ROS-basierten Robotikanwendungen erleichtert. Er eignet sich für eine Vielzahl von Anwendungen, wie z. B. mobile Robotikforschung, Bildverarbeitung, Navigation und Kollisionsvermeidung. [16]



Abbildung 4: Roboter Kobuki [16]

2.4 ODROID-XU4

Der ODROID-XU4, zu sehen in Abbildung 5, ist ein leistungsstarker Einplatinen-Computer, der von der Firma Hardkernel entwickelt wurde. Dieser Einplatinen-Computer bietet eine bemerkenswerte Rechenleistung in einem kompakten Formfaktor. Der Computer wird von einem Octa-Core-Prozessor angetrieben und verfügt zusätzlich über 2GB RAM. Außerdem besitzt der ODROID-XU4 eine breite Palette von Schnittstellen, darunter USB 3.0, USB 2.0, Gigabit-Ethernet, HDMI, Audio-Ausgang und mehr. Auch besteht die Möglichkeit, auf diesem Einplatinen-Computer das Betriebssystem Linux sowie andere Betriebssysteme zu installieren.

Aufgrund seiner leistungsstarken Hardware und seiner umfangreichen Schnittstellen kann der Einplatinen-Computer für verschiedene Anwendungen wie z. B. Heimautomatisierung, Medioplayer, IoT-Projekte oder Robotik eingesetzt werden. [17]

In dieser Arbeit dient der ODROID als zentrale Recheneinheit für den Kobuki-Roboter. Als Betriebssystem kommt Ubuntu MATE 20.04 zum Einsatz. Dies ermöglicht die Bereitstellung derselben ROS-Version wie auf dem Entwicklungsrechner, was eine reibungslose Integration zwischen den Systemen gewährleistet.

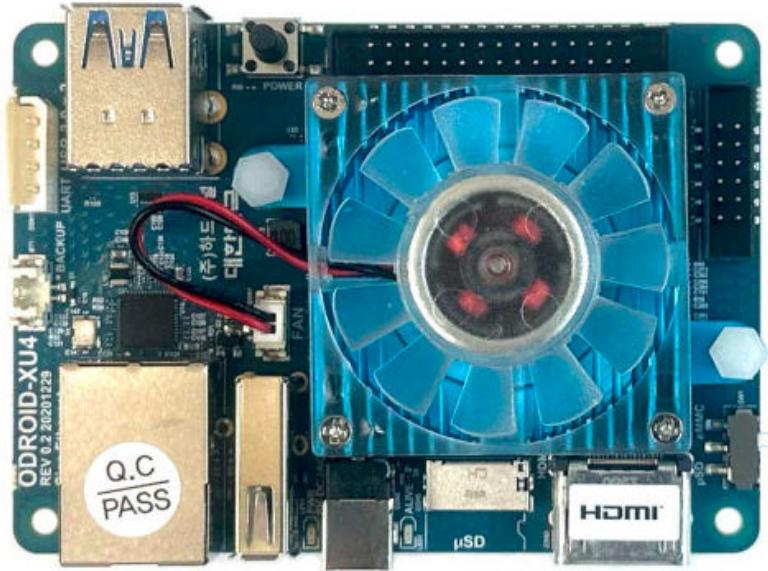


Abbildung 5: Einplatinen-Computer ODROID-XU4 [17]

2.5 Schrödi

Schrödi ist ein Rettungsroboter, der für den Einsatz bei Rettungseinsätzen und in Katastrophenszenarien entwickelt wurde. Seine Softwarearchitektur basiert auf ROS, was ihm eine flexible und leistungsstarke Grundlage für seine Funktionalitäten verleiht. Das Chassis von Schrödi wurde in enger Zusammenarbeit mit dem RoboCup Rescue Team der Fachhochschule Kärnten Villach entwickelt. Dank seines Kettenantriebs und vier Flippern besitzt Schrödi die Fähigkeit, auch durch anspruchsvolle Hindernisse wie Treppen und Stufenfelder zu navigieren. Zusätzlich verfügt der Roboter über einen manipulativen Roboterarm, der problemlos Inspektions- und Manipulationsaufgaben wie das Öffnen von Türen bewältigen kann. [18]

Schrödi ist mit verschiedenen Sensoren ausgestattet, darunter Farbkameras, ein Laser-scanner und eine Wärmebildkamera. Diese Sensoren ermöglichen es ihm, Hindernissen auszuweichen, Brandherde zu erkennen und Personen in dichtem Rauch zu lokalisieren. Die Steuerung von Schrödi erfolgt über einen PlayStation-Controller. Dies vereinfacht die Bedienung und gestaltet sie vertraut. Abbildung 6 zeigt den Roboter im Einsatz beim Öffnen einer Tür. [18]



Abbildung 6: Roboter Schrödi bei den RoboCup Rescue German Open 2023

2.6 Orbbec Astra Stereo S

Die Orbbec Astra Stereo S Kamera, zu sehen in Abbildung 7, ist eine leistungsfähige 3D-Stereokamera, die für eine Vielzahl von Anwendungen in der Robotik, Computer Vision und Augmented Reality entwickelt wurde. Die Kamera nutzt ein Stereo-Bildgebungssystem mit integrierten Farbkameras und Infrarotprojektoren. Durch die Kombination von Stereovision und Infrarottechnologie ermöglicht die Orbbec Stereo S Kamera eine genaue Erfassung der räumlichen Tiefe und eine zuverlässige Erkennung von Objekten und Umgebungsmerkmalen. [19]

Die Kamera bietet eine hohe räumliche Auflösung und Genauigkeit, wodurch sie gut für Anwendungen wie die Gestenerkennung, 3D-Modellierung sowie Hindernisvermeidung in Robotiksystemen geeignet ist. Die hohe Bildrate der Orbbec Stereo S ermöglicht eine schnelle Verarbeitung von Tiefeninformationen und bietet so eine zuverlässige Erfassung von Bewegungen und Szenen. Zusätzlich wird die Kamera durch eine robuste und benutzerfreundliche Softwareplattform unterstützt, die eine einfache Integration und Konfiguration in verschiedenen Anwendungen ermöglicht. Dank der offenen Schnittstellen und der Kompatibilität mit gängigen Entwicklungsumgebungen wie ROS und OpenCV ist die Kamera für Entwickler und Forscher leicht zugänglich und flexibel einsetzbar. [19]

Insgesamt bietet die Orbbec Stereo S Kamera fortschrittliche 3D-Bildgebungsfunctionen in einem kompakten und zuverlässigen Design. Ihre Präzision und Vielseitigkeit machen sie zu einem wertvollen Werkzeug für innovative Anwendungen in der Robotik.

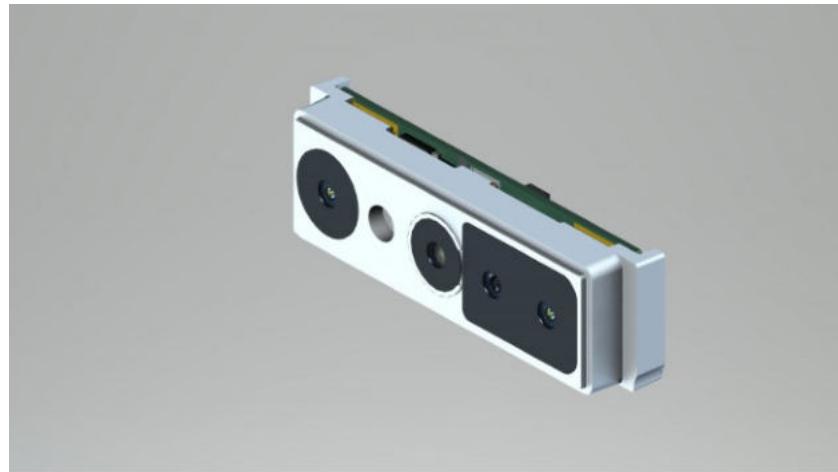


Abbildung 7: 3D-Kamera Orbbec Astra Stereo S [20]

3 Implementierung in einer Simulation

Bereits der Aufbau eines Testparcours mit negativen Hindernissen bringt einen enormen Aufwand mit sich. Auch besteht die Gefahr, dass der Roboter während der Testläufe vom Parcours abkommt und dies zu Beschädigungen des Roboters führen könnte. Aufgrund dieser Aspekte wird die Implementierung zur Lösung des Problems zunächst in einer Simulation stattfinden.

3.1 Roboter- und Parcoursauswahl

Wie bereits im Kapitel der Grundlagen erwähnt wurde, wird für die Simulation Gazebo verwendet. Für eine effiziente Bearbeitung der Probleme wird sich nicht zu intensiv mit der Modellierung in Gazebo beschäftigt. Stattdessen wird nach einem bereits vorhandenen Roboter-Simulationsmodell gesucht. Die hierfür wichtigsten Attribute sind ein Differentialantrieb, das Vorhandensein einer 3D-Kamera und zuletzt die Baugröße. Der Differentialantrieb ist wichtig, da auch die beiden Roboter Schrödi und Kobuki diese Antriebsart verwenden. Ohne die 3D-Kamera kann der Roboter nichts „sehen“ und somit auch keine Hindernisse detektieren. Hier ist es allerdings nicht von Bedeutung, das richtige Kameramodell zu verwenden, da simulierte 3D-Kameras ohne Rauschen alle ein sehr ähnliches Bild erzeugen. Zuletzt sollte das Robotermode in etwa die Größe des Kobuki aufweisen, welcher später für reale Tests verwendet wird.

Zwar besitzt der Kobuki bereits ein fertiges Gazebo-Modell, allerdings ohne 3D-Kamera. Eine weitere Option stellt das Modell des Turtlebot2 dar. Dieser besitzt eine 3D-Kamera und verwendet als Basis den Kobuki. Allerdings ist dieses Simulationsmodell nur für ältere ROS-Versionen verfügbar. Somit fiel die Wahl auf den Turtlebot3 vom Typ „Waffle“, welcher in Abbildung 8 zu sehen ist. Dieser Roboter erfüllt die Anforderungen an einen

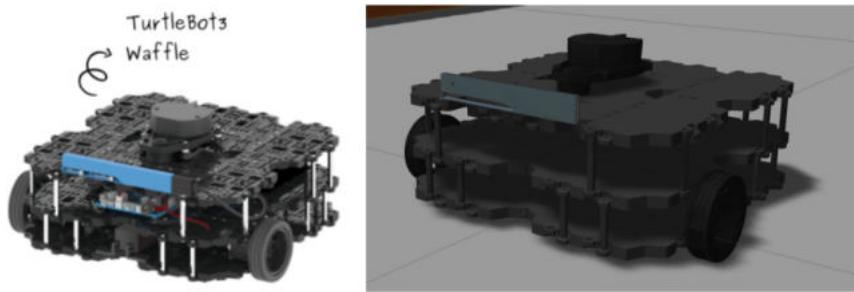


Abbildung 8: TurtleBot3 real (links) und simuliert (rechts) [24]

Differentialantrieb und verfügt über Simulationsmodelle, die eine 3D-Kamera beinhalten. Auch die Baugröße entspricht in etwa der Größenordnung des Kobuki. [21]

Neben dem Robotermodell wird zusätzlich noch ein Parcours benötigt, welcher negative Hindernisse aufweist. Hierfür wird sich an einem Parcours aus dem Rulebook des RoboCup Rescue 2019 orientiert, zu sehen in Abbildung 9. Die genauen Abmessungen des Parcours sind hier zwar nicht gegeben, sie lassen sich allerdings schätzungsweise aus der Größe der verwendeten Steine, 6 inch Betonblöcke (440 x 220 x 150 mm), errechnen. Auch der simulierte Parcours ist in der Abbildung zu sehen. [22, 23]

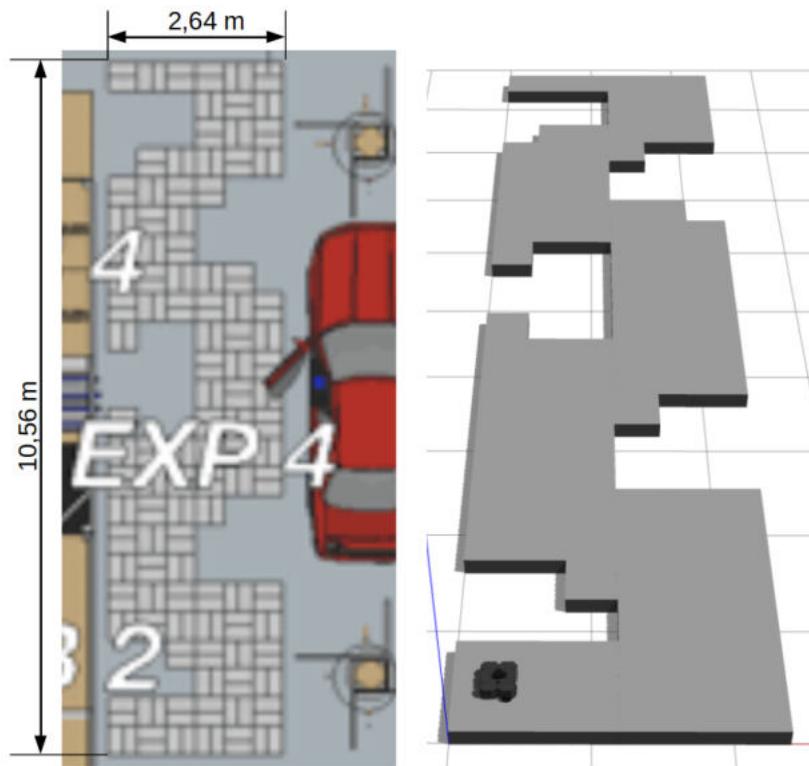


Abbildung 9: Parcours negative Hindernisse Layout (links) und Simulation (rechts) [23]

3.2 Planung und Vorgehensweise

Im Rahmen des angestrebten Ziels dieser Arbeit, die autonome Exploration eines Parcours mit negativen Hindernissen, ergeben sich drei wesentliche Hauptaufgaben:

1. Erstellung einer Umgebungskarte mithilfe einer 3D-Kamera.
2. Autonome Navigation basierend auf dieser Karte.
3. Autonome Vervollständigung der Karte durch Anwendung eines Explorationsalgorithmus.

Bei der Planung der Herangehensweise zur Problemlösung erweist es sich als vorteilhaft, die Anordnung der Probleme aus einer alternativen Perspektive zu betrachten. Innerhalb der ROS-Entwicklung ist es gängige Praxis, auf bereits bestehende Pakete zurückzugreifen. Dabei kann es vorkommen, dass ein bestimmtes Paket von anderen Paketen abhängig ist und ohne diese Abhängigkeiten nicht korrekt funktioniert.

Angenommen, man verfolgt den herkömmlichen sequenziellen Ansatz, so wird zuerst die autonome Navigation mit verschiedenen ROS-Paketen realisiert. Anschließend wird nach einem geeigneten Explorationspaket gesucht. Jedoch besteht hier die Möglichkeit, dass man ein von der Funktion her geeignetes Explorationspaket findet, welches allerdings von bisher ungenutzten Paketen abhängig ist. In einer derartigen Situation müsste die Entscheidung getroffen werden, ob auf die Nutzung des Explorationspaketes verzichtet wird oder ob die Navigation erneut unter Einbeziehung der erforderlichen Pakete implementiert werden sollte.

Aufgrund dieser Überlegungen wird für diese Arbeit der Ansatz umgekehrt, indem die Planung mit der letzten Aufgabe der Suche nach einem geeigneten Explorationspaket beginnt.

Bei der Suche nach Paketen für die autonome Exploration fallen vor allem zwei Optionen auf: `frontier_exploration` und `explore_lite`. Die Entscheidung für das Paket `explore_lite` ergibt sich aus verschiedenen Gründen. Zum einen zeichnet sich `explore_lite` durch eine besonders ressourcenschonende Arbeitsweise aus, da es keine eigene Karte erstellt. Zum anderen ist zu beachten, dass `frontier_exploration` ausschließlich für ältere Versionen von ROS entwickelt wurde, was möglicherweise zu Inkompatibilitätsproblemen mit neueren Paketen führen kann. Daher fällt die Wahl aus Gründen der Effizienz und Kompatibilität auf `explore_lite`. [25, 26]

Im nächsten Schritt der Planung erfolgt eine genauere Untersuchung des Pakets `explore_lite` und seiner Abhängigkeiten. Um `explore_lite` erfolgreich einzusetzen, sind sowohl ein Knoten des Pakets `move_base` als auch eine Karte notwendig, die als ROS-Nachricht des Typs `nav_msgs/OccupancyGrid` vorliegen muss. Das Paket `move_base` ist ein etablierter Bestandteil des ROS Navigation Stacks und somit optimal für die anstehende Aufgabe der autonomen Navigation geeignet. Besonders vorteilhaft ist dabei, dass `move_base` die Fähigkeit besitzt, den Nachrichtentyp `nav_msgs/OccupancyGrid`, welcher für die Exploration erforderlich ist, zu interpretieren. So kann `move_base` entweder eine solche Karte einlesen und verwenden oder selbstständig eine Karte dieses Typs generieren. Alternativ kann die Karte auch von einem anderen Mapping-Paket erstellt werden, sofern dieses den genannten Nachrichtentyp unterstützt. [26, 27]

Das Vorgehen bei der Abarbeitung der Aufgaben lässt sich daher in folgender Reihenfolge festlegen: Zuerst wird versucht, eine Karte der Umgebung bzw. des Parcours als Nachrichtentyp `nav_msgs/OccupancyGrid` aufzunehmen. Anschließend wird mithilfe des Pakets `move_base` eine autonome Navigation umgesetzt. Schließlich erfolgt unter Verwendung des Pakets `explore_lite` die autonome Exploration des Parcours.

3.3 Mapping

Wie im vorherigen Abschnitt erläutert, ist die erste Aufgabe die Erstellung einer Umgebungskarte mithilfe einer 3D-Kamera. Wie üblich gibt es auch für diese Aufgabe vorgefertigte ROS-Pakete. Daher gilt es nun im folgenden Schritt, verschiedene Mapping-Pakete zu untersuchen, zu bewerten und gegebenenfalls zu testen. Pakete, die die Karte in Form einer 3D-Punktwolke speichern, werden von vornherein nicht betrachtet, da sie zu viele Ressourcen benötigen.

3.3.1 costmap_2d

Das Paket `costmap_2d` ist Bestandteil des ROS Navigation Stack und wird von `move_base` verwendet. `costmap_2d` nutzt Daten von Sensoren wie Laserscannern und Tiefenkameras, um die Umgebung des Roboters abzubilden. Dabei werden Informationen über die Belegung von Zellen in einer 2D-Gitterstruktur gespeichert. Jede Zelle hat einen bestimmten Kostenscore, der anzeigt, wie teuer oder sicher es ist, sich in diesem Bereich zu bewegen.

Hindernisse oder unwegsames Gelände erhalten hohe Kostenscores, während freie und sichere Bereiche niedrige Kostenscores haben. [28]

Ein Vorteil des `costmap_2d`-Pakets besteht darin, dass es dynamisch ist, dies bedeutet, dass es sich an Änderungen in der Umgebung anpassen kann. Wenn sich Hindernisse bewegen oder neue Hindernisse auftauchen, wird die Karte entsprechend aktualisiert, um dem Roboter aktuelle Informationen zu liefern.

Zwar ist eine der Anforderungen zur Funktionalität dieses Pakets ein horizontal montierter Laserscanner. Dennoch wird dieses Paket aufgrund seiner integralen Rolle im Navigation Stack für einen Test herangezogen. [28]

Vorbereitung

Um die Funktionen von `costmap_2d` zu nutzen, ist es erforderlich, den `move_base`-Knoten zu starten. Dieser Knoten wird mithilfe eines Launch-Files gestartet, welches die erforderlichen Konfigurationsinformationen in YAML-Files enthält. Da `move_base` später in einem eigenen Kapitel ausführlich behandelt wird, werden an dieser Stelle nur die Einstellungen erwähnt, die für die Erstellung der Karte relevant sind.

Die für diesen Test relevantesten Einstellungen sind das Hinzufügen eines Sensors und dessen Parameter. In Abbildung 10 ist ein Ausschnitt der Konfigurationsdatei zu sehen.

Sowohl diese Datei als auch der gesamte restliche Code dieser Arbeit sind unter folgender Adresse zu finden:

https://github.com/PinznerLe/autonomous_exploration_negative_obstacles_ros.git

```
observation_sources: scan          # create observation source named "scan"
scan: {sensor_frame: camera_depth_frame,
       data_type: PointCloud2,
       topic: /camera/depth/points,
       marking: true,           # allow to mark obstacles
       clearing: true,          # allow to clear out freespace
       max_obstacle_height: 0.1, # max height of a sensor reading considered valid
       min_obstacle_height: 0.00} # min height of a sensor reading considered valid
```

Abbildung 10: Ausschnitt der Konfigurationsdatei für `costmap_2d`

Systemaufbau

Die vereinfachte Darstellung des verwendeten Systemaufbaus ist in Abbildung 11 dargestellt. Diese Darstellung folgt einer bestimmten Konvention. Kreise repräsentieren Knoten im System, während Topics in Rechtecken dargestellt werden. In der ersten Zeile jedes

Rechtecks ist der Name des Topics angegeben, darunter der Nachrichtentyp. Ein Pfeil, der von einem Knoten zu einem Topic verläuft, zeigt an, dass dieser Knoten Nachrichten auf dieses Topic veröffentlicht. Umgekehrt abonniert ein Knoten ein Topic, wenn der Pfeil vom Topic zum Knoten verläuft. Die einzige Ausnahme ist der Pfeil, der von der Kamera zum Turtlebot-Knoten verläuft. In diesem Fall bedeutet der Pfeil lediglich, dass die Kamera Daten an den Knoten überträgt, ohne dass der Knoten das Topic aktiv abonniert.

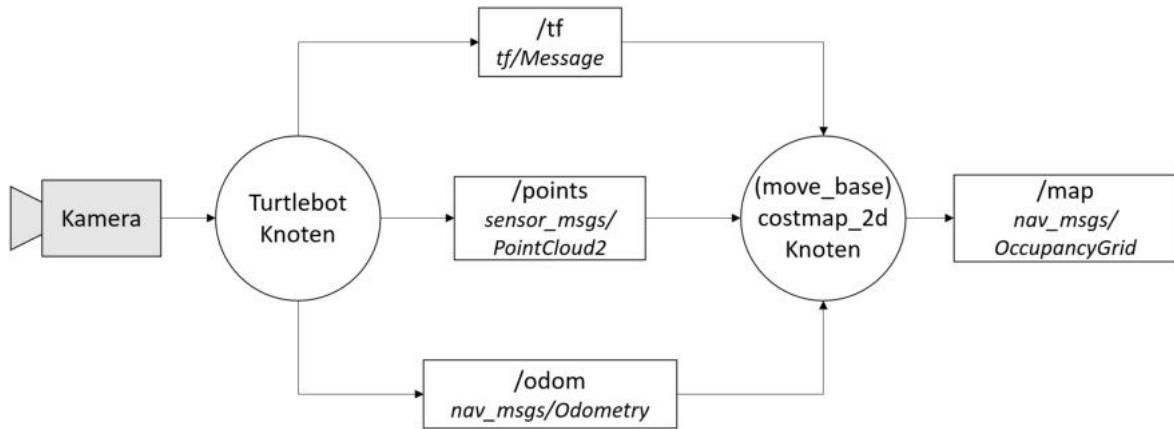


Abbildung 11: Vereinfachter Systemaufbau `costmap_2d`

Mithilfe der 3D-Kamera wird eine Punktwolke über das Topic `/points` von dem Turtlebot-Knoten bereitgestellt. Damit `costmap_2d` jedoch eine Karte aus den Daten der Punktwolke generieren kann, benötigt `costmap_2d` zusätzlich die Topics `/tf` und `/odom`. Das `/tf`-Topic enthält Informationen über die Transformationen der Koordinatensysteme, während das `/odom`-Topic zur Lokalisierung des Roboters verwendet wird. Abschließend wird die erstellte Karte über das Topic `/map` veröffentlicht.

Test

Wird der Knoten nun über das Launch-File gestartet, wird entsprechend der Punktwolke des Sensors eine Costmap erstellt, zu sehen in Abbildung 12. Rechts in der Abbildung ist die aktuelle Situation in der Simulationsumgebung zu sehen, dies dient einem besseren Überblick. Im linken Teil wird die Aufnahme verschiedener Daten in der Visualisierungs-umgebung RViz dargestellt. Zum einen sind hier die einzelnen, in schwarz gefärbten Punkte der von der 3D-Kamera erstellten Punktwolke zu sehen. Des Weiteren wird die von `costmap_2d` aus den Punkten generierte Karte angezeigt. Die rosafarbenen Felder repräsentieren dabei erkannte Hindernisse wie den tieferliegenden Boden, während die

blauen Felder automatisch erzeugt werden und als Puffer zu den erkannten Hindernissen dienen. Felder, welche nicht als Hindernis erkannt werden, bleiben frei.

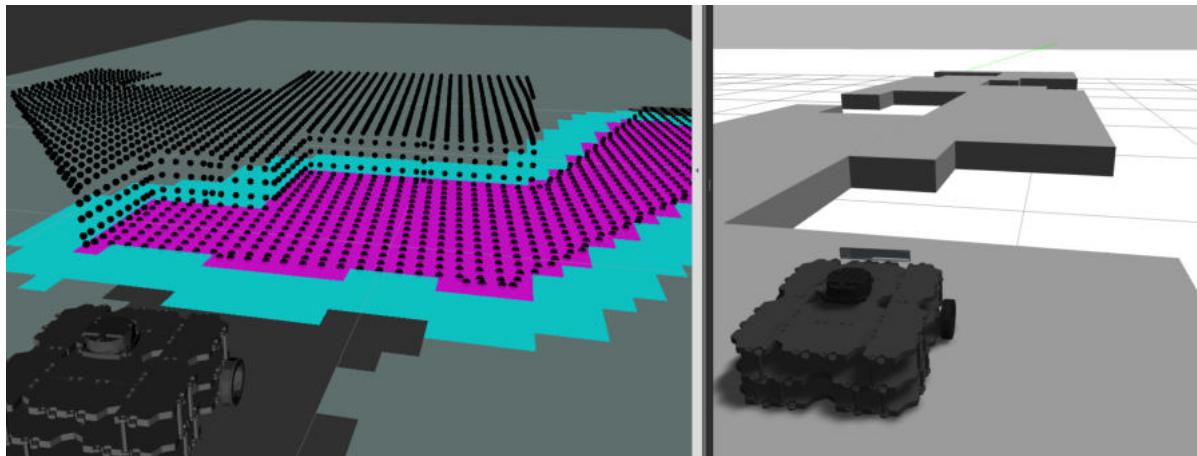


Abbildung 12: Funktion von `costmap_2d` in RViz (links) und Überblick über die Umgebung in Gazebo (rechts)

Beim weiteren Testen dieses Pakets stellt sich jedoch ein Problem beim Rotieren des Roboters dar. Dieses Problem ist in Abbildung 13 dargestellt. Auch hier wird im oberen Screenshot rechts ein Abschnitt der Simulationsumgebung Gazebo gezeigt, der lediglich ein besseres Verständnis der Versuchsumgebung vermitteln soll.

Bei Betrachtung der linken Seite sieht man, dass der Boden im Sichtfeld des Roboters gut erkannt wird. Das nun auftretende Problem ist im unteren Bild zu sehen. Wenn der Roboter rotiert, in diesem Fall nach rechts, kommt es dazu, dass am Rand der Punktwolke die Felder, die gerade eben noch als Boden erkannt wurden, nun als frei markiert werden. Dies hat zur Folge, dass sie nicht mehr als Hindernisse erkannt werden.

Eine mögliche Lösung für dieses Problem besteht darin, dem 3D-Sensor lediglich das Setzen von Hindernissen zu erlauben und das Freigeben zu verbieten. Allerdings resultiert aus dieser Lösung der Verlust des Vorteils einer dynamischen Hinderniserkennung. Dieser Verlust würde sich insbesondere bei einer realen 3D-Kamera bemerkbar machen, da hier zur Punktwolke noch ein gewisses Rauschen hinzukommt. Ohne die Möglichkeit, Felder wieder freizugeben, könnten einzelne Störer oder Rauschen die gesamte Karte unbrauchbar machen.

Für diese Arbeit erscheint es jedoch nicht sinnvoll, weiter in die Analyse dieses Problems einzusteigen. Wie bereits erwähnt, erfordert dieses Paket normalerweise die Verwendung eines Laserscanners. Darüber hinaus sind sowohl das `costmap_2d`-Paket als auch das dazugehörige `move_base`-Paket in ihrem Umfang und ihrer Komplexität sehr umfangreich. Daher wird stattdessen nach einer alternativen Lösung zur Kartenerstellung gesucht.

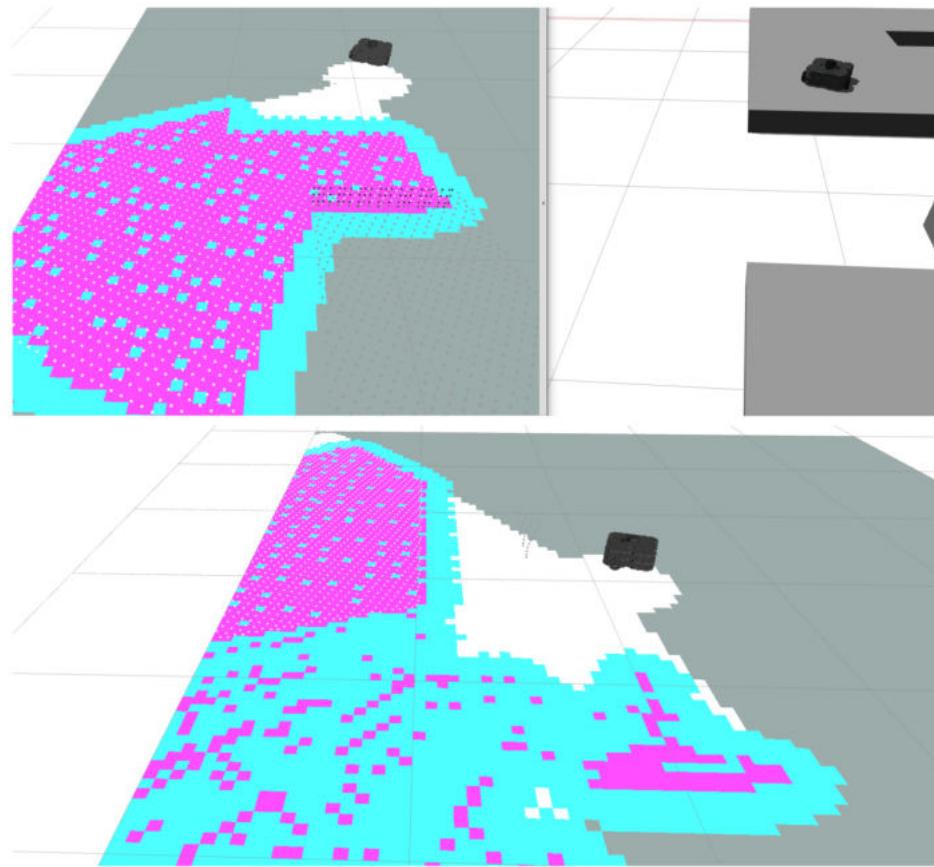


Abbildung 13: Fehlerhaftes Verhalten von `costmap_2d` bei Rotation des Roboters

3.3.2 `grid_map`

Eine Alternative zu `costmap_2d` ist das Paket `grid_map`. Anders als `costmap_2d`, welches speziell auf autonome Navigation und Pfadplanung ausgelegt ist, ist `grid_map` allgemeiner ausgelegt und bietet eine breitere Palette von Funktionen für die Verarbeitung von 2D-Rasterdaten.

Eines der Hauptmerkmale von `grid_map` ist seine Flexibilität bei der Verwaltung von Rasterkarten. Es ermöglicht die Erstellung und Verwaltung von 2D-Karten in einem festgelegten Rasterformat, wobei in jedem Rasterfeld beliebige Informationen wie Höhenwerte, Traversabilität, Hindernisse oder andere relevante Messdaten gespeichert werden können. Zusätzlich bietet `grid_map` die Möglichkeit, dass eine Karte mehrere sogenannte „Layer“, also Ebenen, enthalten kann. Diese Funktion ermöglicht es den einzelnen Rasterfeldern, mehrere Informationen in unterschiedlichen Ebenen zu speichern.

Außerdem unterstützt `grid_map` verschiedene Datentypen und ermöglicht die Integrati-

on von Sensordaten aus verschiedenen Quellen. Dies vereinfacht die Implementierung und Anpassung an spezifische Anforderungen erheblich, was die Arbeit mit komplexen Robotikanwendungen erleichtert.

Darüber hinaus bietet das Paket fortschrittliche Funktionen zur Datenverarbeitung und Analyse, darunter Algorithmen zur Interpolation, Filterung, Fusion und Transformation von Daten in der Rasterkarte. In diesem Zusammenhang sei erwähnt, dass das `grid_map`-Paket einen Knoten enthält, welcher es ermöglicht, die erzeugte Karte in den Nachrichtentyp `nav_msgs/OccupancyGrid` zu wandeln. Zunächst wird allerdings mit dem Paket eigenem Datentyp `GridMap` gearbeitet. [29]

Aufgrund der genannten Vorteile wird auch das Paket `grid_map` für einen Test herangezogen.

Vorbereitung

Im Gegensatz zu `costmap_2d` existiert kein vorgefertigter Knoten, welcher die Daten der Punktwolke direkt zu einer Karte verarbeiten kann. Daher ist es notwendig, einen eigenen Knoten zu entwickeln. In Anbetracht des Rahmens dieser Arbeit ist es nicht sinnvoll, den Code zeilenweise zu erläutern. Stattdessen wird nachfolgend lediglich die Funktionsweise des Algorithmus zur Kartenerstellung erörtert. Für diesen Zweck wurde der folgende Pseudocode erstellt.

Algorithm 1 Algorithmus zum Erstellen einer GridMap-Karte aus einer PointCloud

| | |
|-----------------------------|--------------------------------|
| Input: pointCloudMsg | // pointcloud of the 3D-Camera |
| gridMap["elevation"] | // grid map with one layer |
| Output: gridMapMsg | // grid map to be published |

```

1: pointCloud ← tf_transform(pointCloudMsg)
2: pointCloudXYZ ← convert(pointCloud)
3: for all point ∈ pointCloudXYZ do
4:     Position pointXY(point.x, point.y)
5:     if gridMap.isInside("elevation", pointXY) then
6:         gridMap.atPosition("elevation", pointXY) ← point.z
7:     end if
8: end for
9: gridMapMsg ← convert(gridMap)

```

Der Algorithmus zur Kartenerstellung verwendet zwei Eingangsvariablen, die Punktwolke der 3D-Kamera, repräsentiert als Punktwolken-Nachrichtentyp, und die aktuelle Karte im Format einer `GridMap`. Als Ausgangsvariable liefert der Algorithmus eine aktualisierte

Karte im GridMap-Nachrichtentyp.

Der Ablauf des Algorithmus beginnt mit der Transformation der Punktwolke und ihres Koordinatensystems in das Koordinatensystem der Karte. Anschließend erfolgt eine Konvertierung der Punktwolke in einen anderen Datentyp, um einen einfacheren Zugriff auf die einzelnen Punkte und deren Positionsdaten zu ermöglichen. Danach wird die gesamte Punktwolke durchlaufen, wobei für jeden Punkt eine Fallunterscheidung erfolgt. Wenn sich der Punkt innerhalb der Karte befindet, wird der Höhenwert aus der z-Koordinate des Punktes in die entsprechende Zelle übertragen. Nachdem die gesamte Punktwolke verarbeitet wurde, erfolgt zum Abschluss die Konvertierung der Karte in einen passenden Datentyp, um sie anschließend veröffentlichen zu können.

Systemaufbau

Der vereinfachte Systemaufbau, der in Abbildung 14 dargestellt ist, ähnelt dem von `costmap_2d`. Der Unterschied besteht darin, dass anstelle des `move_base`-Knotens ein `grid_map`-Knoten das Erstellen der Karte übernimmt, welcher außerdem einen anderen Datentyp ausgibt. Ein zweiter Unterschied ist es, dass dem `grid_map`-Knoten die Daten des `tf`-Topics genügen und er keine zusätzlichen Odometrie-Daten benötigt.

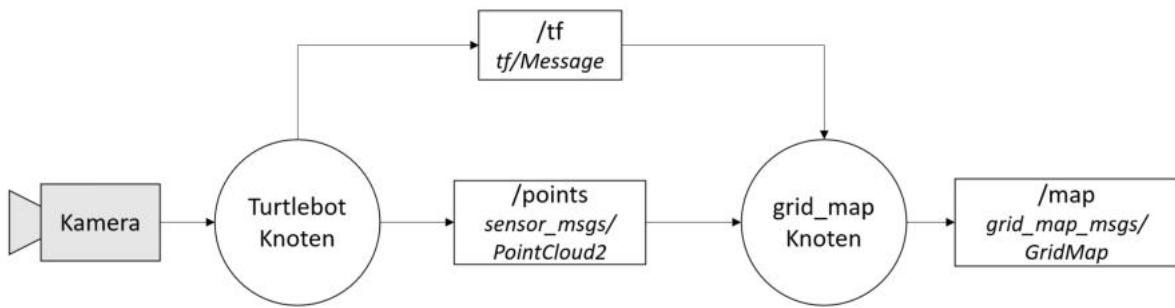


Abbildung 14: Vereinfachter Systemaufbau `grid_map`

Test

In den folgenden Abbildungen sind mehrere Screenshots des Tests zu sehen. Wie in Abbildung 15 zu erkennen ist, wird beim Starten des Knotens eine GridMap aus den Daten der Punktwolke generiert. Die unterschiedlichen Farben der Karte stellen hierbei unterschiedliche Höhen dar. Aber auch hier stößt man auf ein Problem, welches die Rotation des Roboters betrifft. Dieses Problem wird anhand von Abbildung 16 und Abbildung 17 dargestellt.

Die rechte Seite von Abbildung 16 dient zur Einordnung der Situation. Auf der linken Seite des Bilds ist die aufgenommene Karte als GridMap zu sehen. Diese wurde erstellt, indem der Roboter sich einmal mit minimaler Geschwindigkeit 360 Grad um die eigene z-Achse gedreht hat. Das Problem wird nun offensichtlich, wenn man Abbildung 17 betrachtet. Die Kanten der erstellten Karte verlaufen nicht mehr parallel mit der x- und y-Achse des Koordinatenursprungs. Diese Verzerrung ist auf die erhöhte Drehgeschwindigkeit bei der Aufnahme der Karte zurückzuführen. Das Problem liegt also darin, dass die Karte bei zu hoher Winkelgeschwindigkeit in Drehrichtung verzerrt wird.

Somit ist auch dieses Paket zur Erstellung einer präzisen Karte ungeeignet. Allerdings gibt es von den Entwicklern dieses Pakets ein weiteres Paket, das speziell für das Mappen mit mobilen Robotern entwickelt wurde. Dieses wird im nächsten Kapitel erläutert.

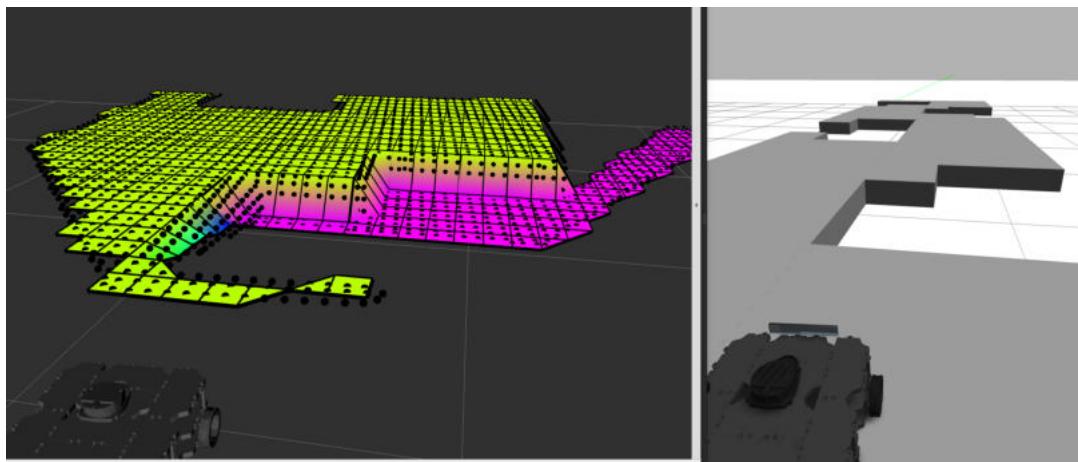


Abbildung 15: Test des entwickelten `grid_map`-Knotens

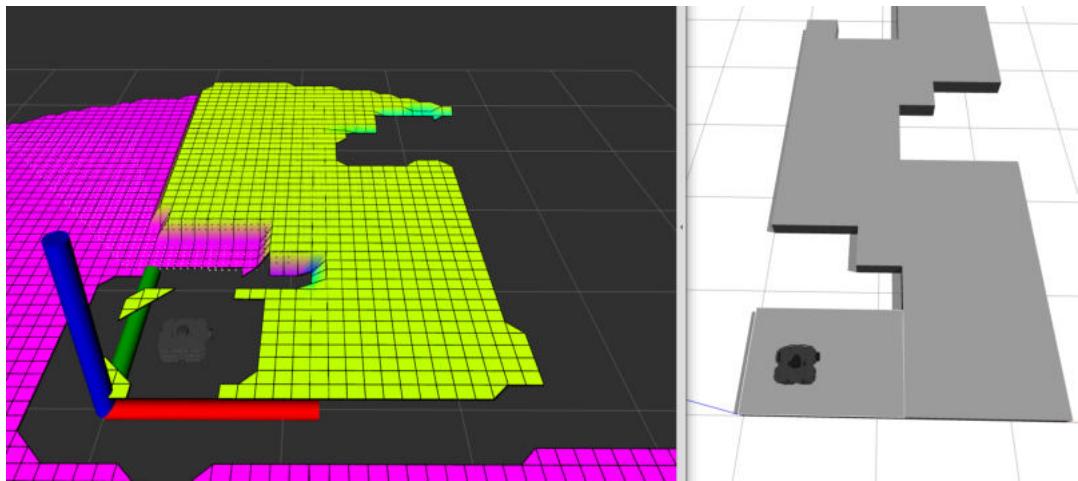


Abbildung 16: Karte bei langsamer Winkelgeschwindigkeit

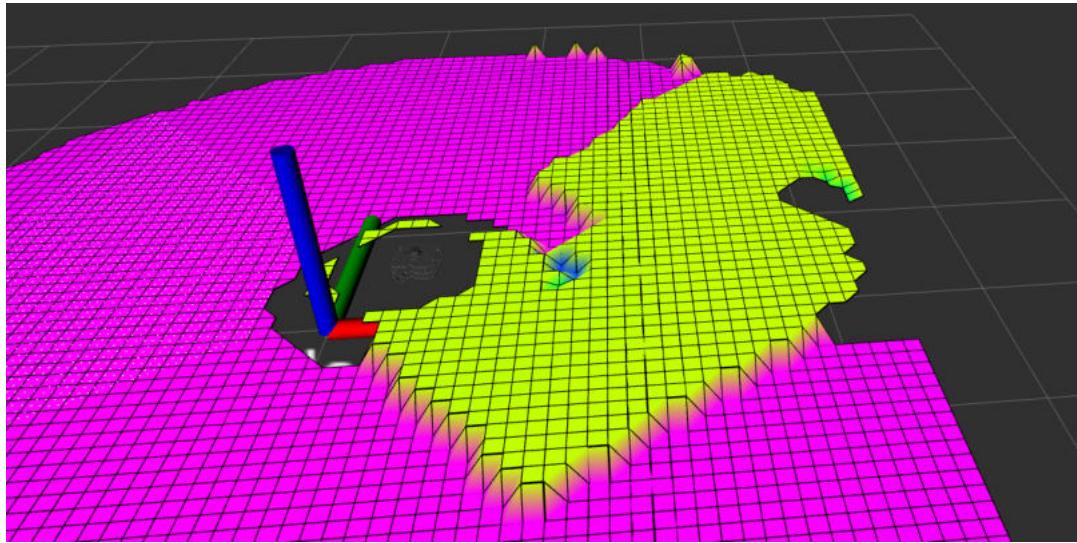


Abbildung 17: Verzerrte Karte bei hoher Winkelgeschwindigkeit

3.3.3 `elevation_mapping`

Das `elevation_mapping`-Paket dient als Erweiterung des ROS-Pakets `grid_map`, welches speziell darauf ausgelegt ist, präzise Höhenkarten zu generieren und zu verarbeiten. Durch die Integration von `grid_map` als Basis profitiert `elevation_mapping` von dessen leistungsstarken Funktionen zur Verarbeitung von Rasterkarten. Dabei konzentriert sich `elevation_mapping` jedoch speziell auf die 3D-Rekonstruktion von Geländedaten aus Punkt wolken, um detaillierte Höhenkarten zu generieren. Diese Kombination ermöglicht eine effiziente und genaue Wahrnehmung der Umgebung in der dritten Dimension für autonome Roboter und mobile Plattformen.

Ein weiterer wichtiger Aspekt von `elevation_mapping` ist seine Fähigkeit zur kontinuierlichen Fusion von mehreren Punkt wolken über die Zeit. Dadurch kann die Höhenkarte während der Fortbewegung des Roboters aktualisiert und verbessert werden, wodurch eine konsistente und genaue Darstellung der Umgebung gewährleistet wird. Dies dürfte auch das Problem lösen, welches beim Test des `grid_map`-Pakets aufgetreten ist. [30]

Auch dieses Paket wird auf Grund der genannten Vorteile getestet und anschließend bewertet.

Vorbereitung

Ähnlich wie der `move_base`-Knoten bei `costmap_2d`, hat auch dieses Paket einen fertigen Knoten, welcher sich über ein Launch-File konfigurieren und starten lässt. Für

den in der Simulation verwendeten Roboter Turtlebot3, gibt es im Paket sogar ein bereits vorgefertigtes Launch-File. So müssen an der Beispielkonfiguration nur wenige Änderungen vorgenommen werden. Bevor die Parameter angepasst werden, wird die im Launch-File vorhandene Gazebo-Simulation durch die Simulation der negativen Hindernisse ausgetauscht. Anschließend werden die Parameter eines YAML-Files angepasst. Die hierbei wichtigsten Parameter sind in Abbildung 18 zu sehen. Zum einen wird die Punktfolge der 3D-Kamera als Eingangsquelle festgelegt. Zusätzlich wird der Parameter `track_point_frame_id` auskommentiert. Dieser Parameter dient dazu, einen Punkt bzw. ein Koordinatensystem zu wählen, mit welchem sich die Karte mitbewegt. Durch die vorgenommene Änderung bewegt sich die Karte nun nicht mehr mit dem Roboter mit, sondern bleibt an einem Ort fixiert. Des weiteren müssen die Parameter für die Größe und Position der Karte an den Testparcours angepasst werden.

```

input_sources:
  front: # A name to identify the input source
    type: pointcloud # Supported types: pointcloud
    topic: /camera/depth/points
    queue_size: 1
    publish_on_update: true # Whether to publish the elevation map after a callback from this source.
    sensor_processor:
      type: perfect
#track_point_frame_id:           base_footprint

length_in_x:          5.0
length_in_y:          12.0
position_x:           1.0
position_y:           5.0
resolution:          0.1

```

Abbildung 18: Ausschnitt der Konfigurationsdatei für `elevation_mapping`

Systemaufbau

Der Systemaufbau, zu sehen in Abbildung 19, ähnelt den vorherigen. In diesem Fall wird die Karte allerdings von einem anderen Knoten, dem `elevation_mapping`-Knoten, erstellt. Der Name des Karten-Topics unterscheidet sich ebenfalls von dem in der vorherigen Konfiguration. Zusätzlich kommt ein weiterer Knoten namens `tf_to_pose_publisher` hinzu. Dieser Knoten ist Teil des `elevation_mapping`-Pakets und wird ebenfalls über das Launch-File gestartet. Die Hauptaufgabe dieses Knotens besteht darin, die `/odom`-Nachricht in eine zusätzliche `/base_footprint_pose`-Nachricht umzuwandeln. Diese spezielle Nachricht ist vom Typ `geometry_msgs/PoseWithCovarianceStamped` und wird für die Kartenerstellung benötigt.

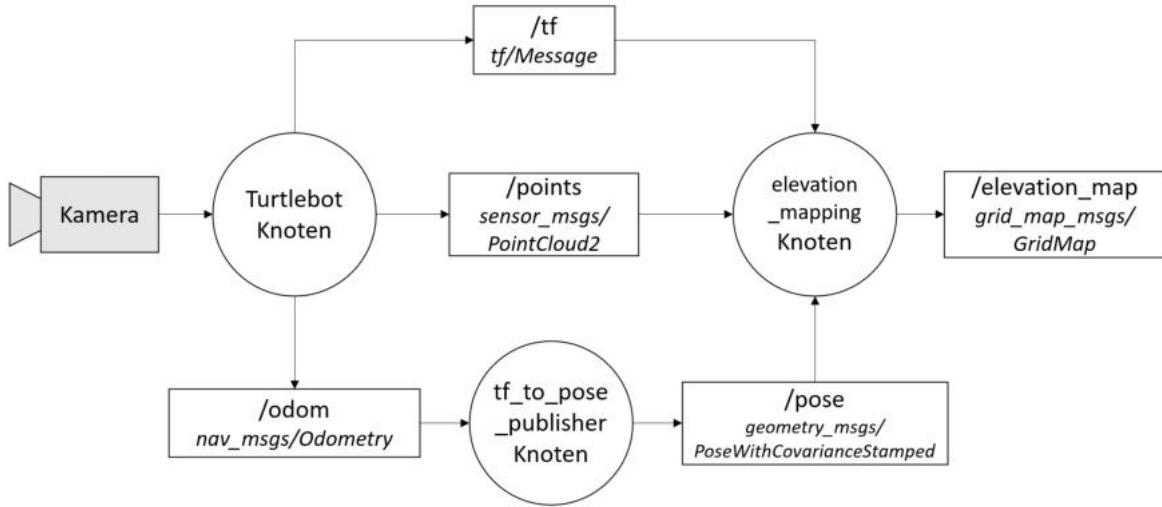


Abbildung 19: Vereinfachter Systemaufbau `elevation_mapping`

Test

Wie erwartet generiert der Knoten anhand der Punktwolken- und Odometrie-Daten eine Karte. Im Vergleich zu `grid_map` dauert dieser Vorgang aufgrund des höheren Rechenaufwands länger. Jedoch sind im Gegensatz zu den anderen beiden getesteten Paketen keine der vorherigen Probleme mehr präsent und es kann eine adäquate Karte des Parcours erstellt werden. Steuert man den Roboter einmal entlang des Parcours und wieder zurück, so erhält man eine Karte wie in Abbildung 20.

Obwohl die Oberfläche des Parcours zunächst präzise erfasst wird, offenbart sich bei genauerer Betrachtung der Karte ein zusätzliches Problem. Es sind auffällige Bereiche an den Rändern des Parcours zu erkennen, in denen die Zellen keine Werte enthalten. Diese Situation tritt auf, wenn die Punktwolke auf eine Kante trifft und dahinter eine Zone entsteht, in der keine Punkte erfasst werden. Ein visuelles Beispiel für dieses Problem ist in Abbildung 21 dargestellt. Der grün markierte Bereich des Bodens wird erkannt, während der rot markierte Bereich nicht erfasst wird, da die Kante die Sicht blockiert. Dieses Problem gilt es nun zu lösen.

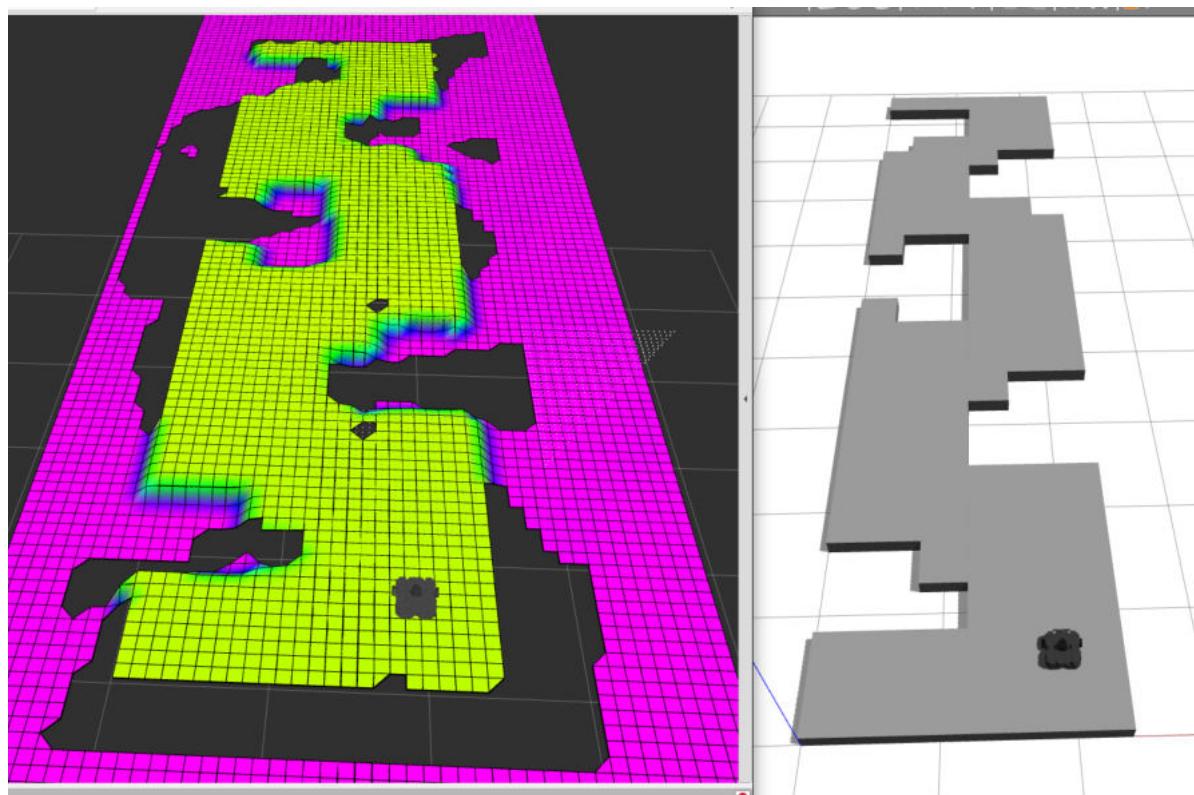


Abbildung 20: Durch `elevation_mapping` erstellte Karte

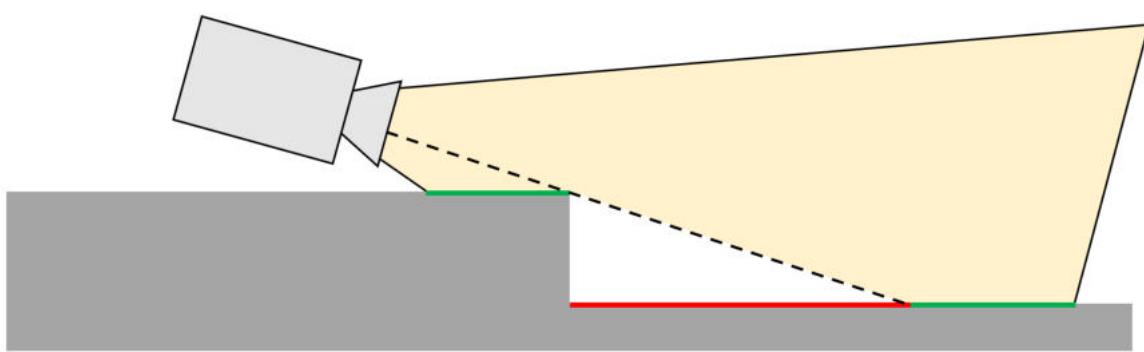


Abbildung 21: Problemzone an Kanten

3.4 Erkennen negativer Hindernisse

Um die im vorherigen Kapitel beschriebene Herausforderung der lückenhaften Kartenerfassung an den Parcoursrändern anzugehen, stehen verschiedene Ansätze zur Verfügung. Ein möglicher Ansatz besteht darin, das Problem auf der Hardwareseite zu lösen, indem die Kamera strategisch angebracht wird. Beispielsweise könnte man die Kamera so hoch wie möglich platzieren und unter einem möglichst steilen Winkel in Richtung des Bodens ausrichten. Dies hätte zur Folge, dass die Zone, in der keine Punkte erfasst werden, minimiert wird. Diese Lösung kann aufgrund des kompakten Aufbaus des Rettungsroboters allerdings nicht realisiert werden. Aus diesem Grund ist eine Lösung auf Softwareebene erforderlich.

Das zugrunde liegende Prinzip des angewandten Lösungsansatzes besteht darin, dass alle Felder der Karte, die sich im Sichtfeld der Kamera befinden und keinen Wert aufweisen, automatisch auf Bodenhöhe gesetzt werden. Diese Vorgehensweise wird durch die Screenshots in den folgenden Abbildungen verdeutlicht. Abbildung 22 dient dabei lediglich einer besseren Veranschaulichung der Situation. In Abbildung 23 wird die Punktewolke dargestellt, die aufgrund der Situation eine Zone aufweist, in der keine Punkte erfasst werden. Zusätzlich ist als Trapez das Sichtfeld der Kamera in grün eingezzeichnet. In letzteren Abbildung, Abbildung 24, ist nun der punktlose Bereich in rot markiert, welcher mittels Software in der Karte automatisch auf Bodenhöhe gesetzt werden soll.

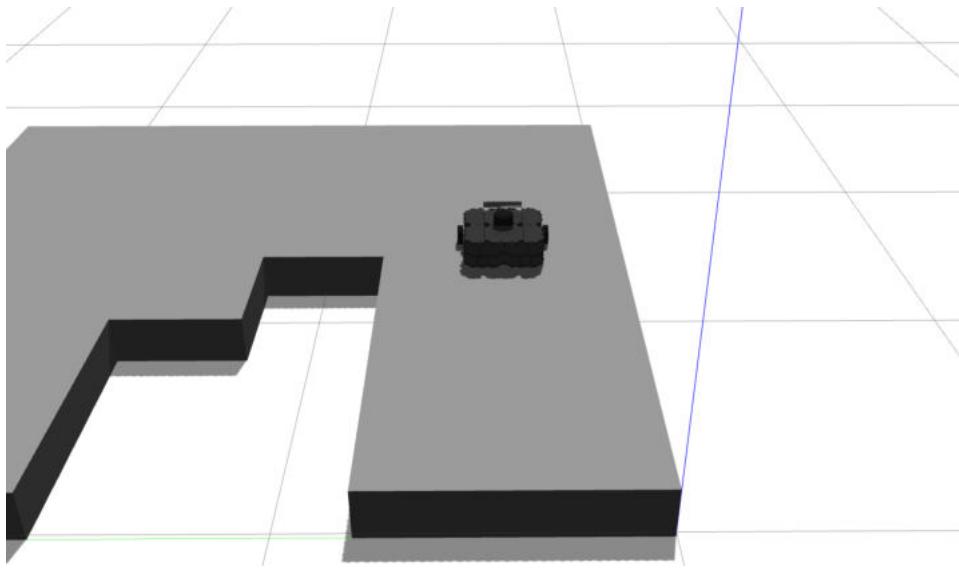


Abbildung 22: Beispiel einer Problemsituation in Gazebo

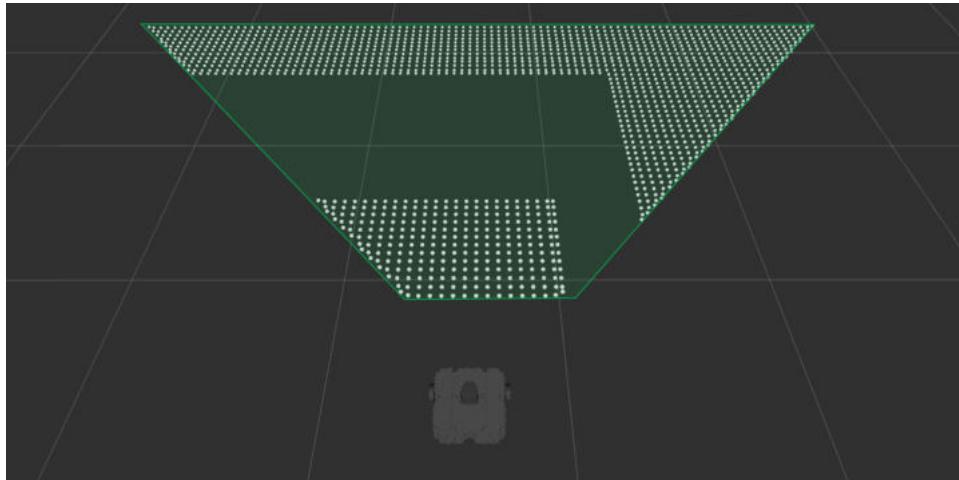


Abbildung 23: Punktwolke und Sichtfeld der 3D-Kamera

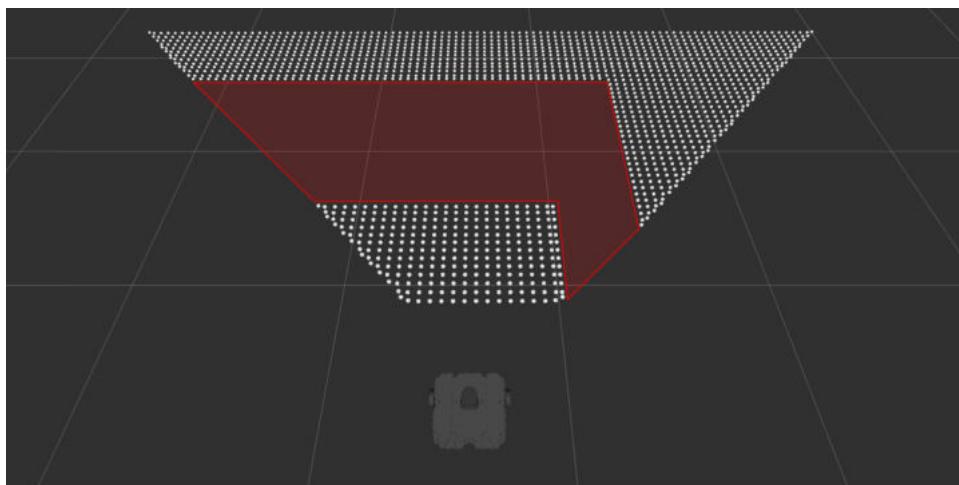


Abbildung 24: Verdeckter Bereich der Punktwolke

Vorbereitung

Basierend auf den gestellten Anforderungen wird ein Knoten entwickelt. Um den Umfang dieser Arbeit angemessen zu halten, wird auch hier lediglich der Pseudocode des Algorithmus erläutert. Dieser Pseudocode findet sich auf der nachfolgenden Seite.

Der Algorithmus zur Kartenerstellung operiert mit drei Eingangsvariablen, wovon zwei als `GridMap` vorliegen. Die erste stammt vom `elevation_mapping`-Knoten, während die zweite zur Manipulation dient und aus drei verschiedenen Ebenen besteht. Die dritte Eingangsvariable repräsentiert die Roboterpose. Die Ausgangsvariable dieses Algorithmus ist eine manipulierte Karte im `GridMap`-Nachrichtentyp.

Zu Beginn des Algorithmus wird die Höhenkarte des `elevation_mapping`-Knotens in

Algorithm 2 Algorithmus zum Manipulieren der Karte

Input: gridMap["elevation"]
 gridMapManipulated["elevation", "ground_manipulated", "elevation_fused"]
 robotPose

Output: gridMapManipulatedMsg

```

1: gridMapManipulated["elevation"]  $\leftarrow$  gridMap["elevation"]
2: robotOrientationYaw  $\leftarrow$  convert(robotPose)
3:
4: // calculating the corner points of the camera trapezoid
5: trapA_x  $\leftarrow$  robotPose.x + cos(robotOrientationYaw + cameraAngle) * d_min
6: trapA_y  $\leftarrow$  robotPose.y + sin(robotOrientationYaw + cameraAngle) * d_min
7: trapB_x  $\leftarrow$  robotPose.x + cos(robotOrientationYaw - cameraAngle) * d_min
8: ...
9: trapD_y  $\leftarrow$  robotPose.y + cos(robotOrientationYaw + cameraAngle) * d_max
10: polygon  $\leftarrow$  add((trapA_x, trapA_y), ... , (trapD_x, trapD_y))
11:
12: // adding ground in the trapezoidal area
13: for (polygonIterator it(gridMapManipulated, polygon); !it.isPastEnd(); ++it) do
14:     if (gridMapManipulated.at("elevation", it) = NaN then
15:         (gridMapManipulated.at("ground_manipulated", it) = ground
16:     end if
17: end for
18:
19: // fusing the layers of the entire map
20: for (GridMapIterator it(gridMapManipulated); !it.isPastEnd(); ++it) do
21:     if (gridMapManipulated.at("elevation", it) != NaN then
22:         gridMapManipulated.at("elevation_fused", it)  $\leftarrow$ 
23:             gridMapManipulated.at("elevation", it)
24:     else if (gridMapManipulated.at("ground_manipulated", it) != NaN then
25:         gridMapManipulated.at("elevation_fused", it)  $\leftarrow$ 
26:             gridMapManipulated.at("ground_manipulated", it)
27:     end if
28: end for
29:
30: gridMapManipulatedMsg  $\leftarrow$  convert(gridMapManipulated)

```

die `elevation`-Ebene der zu manipulierenden Karte kopiert. Anschließend erfolgt die Berechnung des Orientierungswinkels um die z-Achse des Roboters im Raum. Mithilfe dieses Winkels werden die Eckpunkte des Kamera-Trapezes ermittelt. Die `grid_map`-Bibliothek ermöglicht es, aus diesen vier Eckpunkten einen Trapez-Iterator zu generieren. Über diesen Iterator werden die Zellen schrittweise überprüft. Dabei wird festgestellt, ob die einzelnen Felder Werte aufweisen. Falls dies nicht der Fall ist, werden die Felder in der `ground_manipulated`-Ebene auf die Höhe des Bodens gesetzt.

Schließlich erfolgt die Fusion der einzelnen Ebenen der Karte in der `elevation_fused`-Ebene. Die gesamte Karte wird durchlaufen, und für jedes Feld erfolgt eine Fallunterscheidung. Falls ein Feld in der `elevation`-Ebene einen Wert aufweist, wird dieser in die `elevation_fused`-Ebene übertragen. Andernfalls wird überprüft, ob in der `ground_manipulated`-Ebene ein Wert vorhanden ist. Wenn ja, wird dieser in die `elevation_fused`-Ebene übertragen, andernfalls bleibt das Feld unverändert. Abschließend wird die manipulierte Karte in den erforderlichen Datentyp umgewandelt, um sie zu veröffentlichen.

Systemaufbau

Abbildung 25 veranschaulicht den vereinfachten Systemaufbau des durchgef hrten Tests. Dieser Aufbau weist eine deutliche hnlichkeit zum Aufbau des Tests f r den `elevation_mapping`-Knoten auf. Allerdings wird dieser nun um einen weiteren Knoten erweitert. Dieser zustzliche Knoten abonniert die durch den `elevation_mapping`-Knoten generierte Karte, manipuliert sie anhand der Roboterpose und gibt anschlieend die modifizierte Karte uber das Topic `/elevation_map_manipulated` aus.

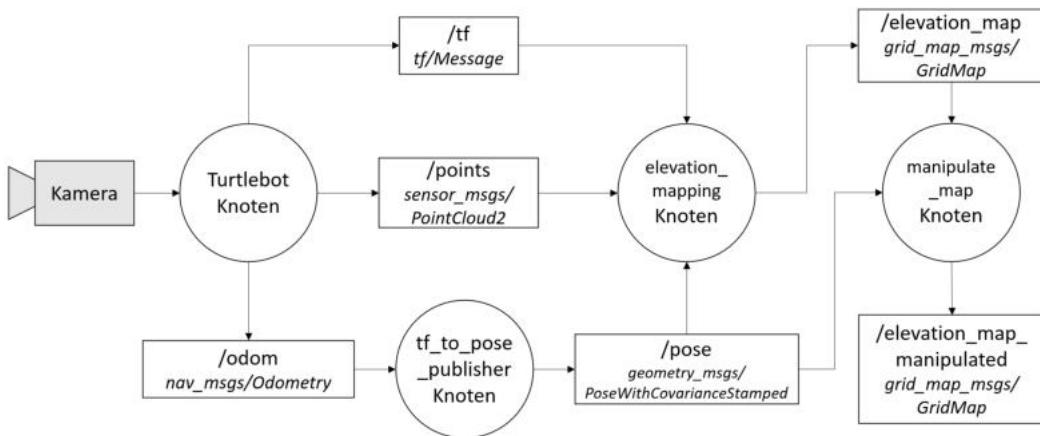


Abbildung 25: Vereinfachter Systemaufbau `manipulate_map`

Test

Zu Beginn des ersten Tests wird der Roboter gezielt an eine Kante gesteuert, woraufhin die relevanten Knoten gestartet werden. Anschließend werden drei Abbildungen aufgenommen. Abbildung 26 zeigt wie gewohnt rechts einen Bildschirmausschnitt der Simulation und links die entsprechende Darstellung in RViz. In dieser ersten Abbildung ist die `elevation`-Ebene zu sehen, die direkt der Karte entspricht, welche vom `elevation_mapping`-Knoten erstellt wird. In Abbildung 27 wird die `ground_manipulated`-Ebene dargestellt, bei der die Zellen gesetzt sind, die im Bereich der Punktfolke keinen Wert aufweisen. Schließlich ist in der letzten Abbildung, Abbildung 28, die `elevation_fused`-Ebene abgebildet, die aus einer Fusion der beiden anderen Ebenen hervorgegangen ist.

Bei Betrachtung dieser Abbildungen lässt sich festhalten, dass der Knoten seine beabsichtigte Funktion erfüllt und der erste Test erfolgreich verlaufen ist.

Der nächste Test ist in Abbildung 29 zu sehen. In diesem Test erfolgt eine komplette Durchfahrt des Parcours sowohl vorwärts als auch rückwärts. Danach erfolgt ein Vergleich der `elevation`- und `elevation_fused`-Ebenen. Erneut zeigt sich, dass die `elevation_fused`-Ebene eine erheblich verbesserte Karte liefert. Dieser Vergleich dient dazu, die Wirksamkeit des Knotens erneut zu bestätigen.

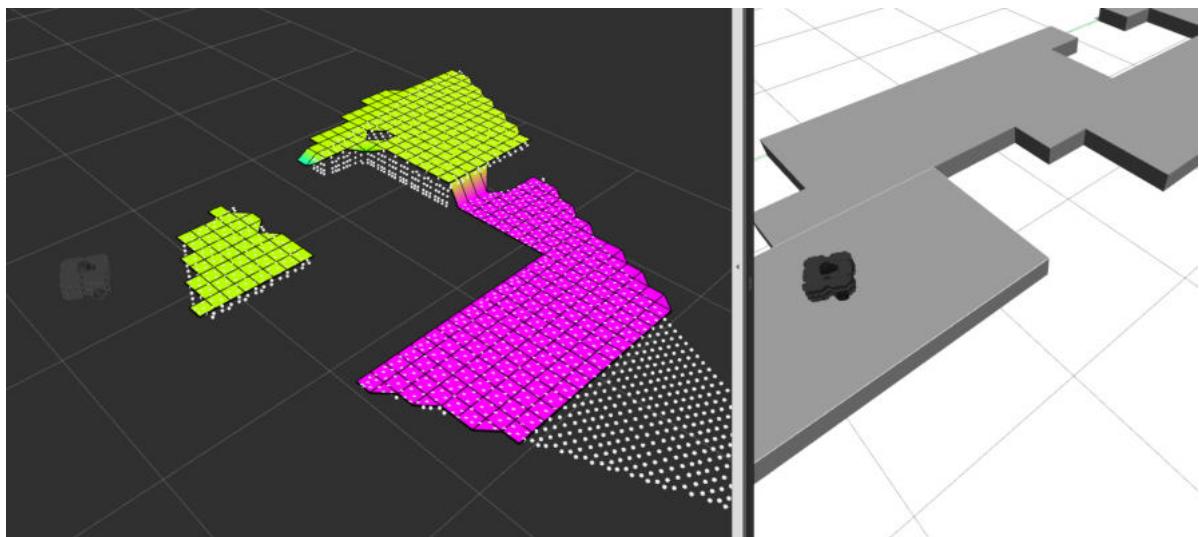


Abbildung 26: `elevation`-Ebene

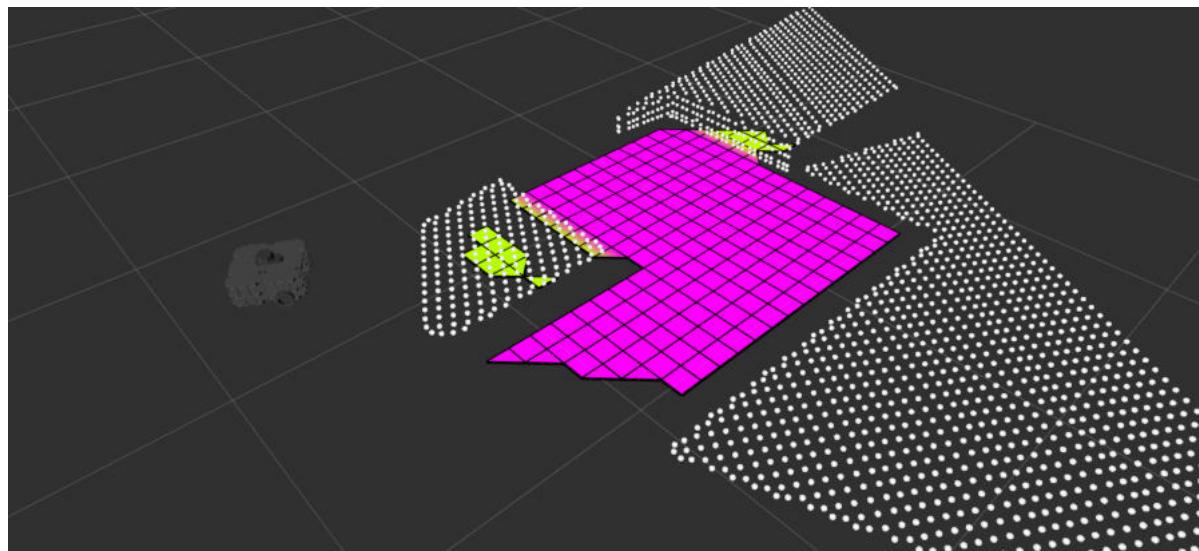


Abbildung 27: `ground_manipulated`-Ebene

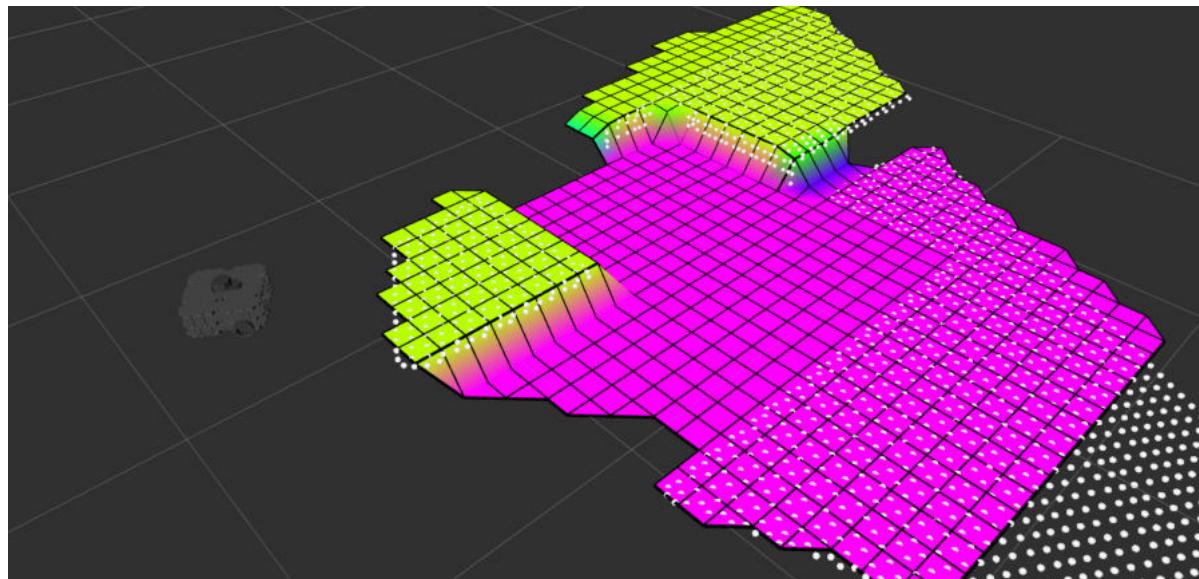


Abbildung 28: `elevation_fused`-Ebene

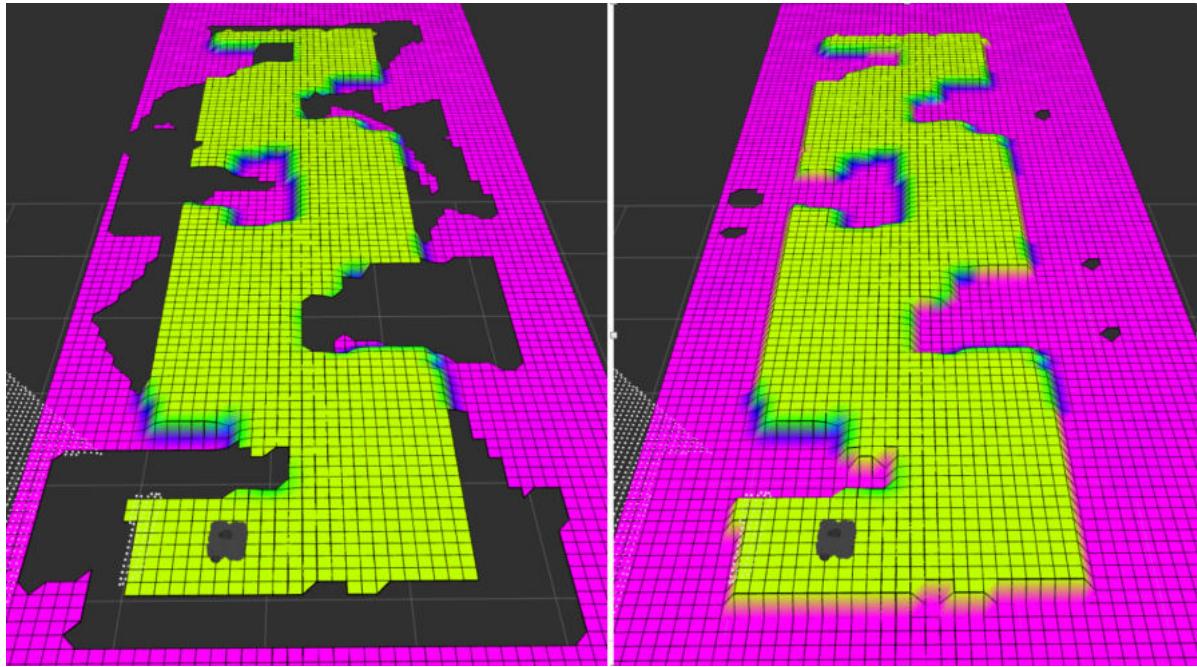


Abbildung 29: Vergleich der elevation- (links) und elevation_fused-Ebene (rechts)

3.5 move_base

Das ROS-Paket `move_base` ist ein wesentlicher Bestandteil des ROS Navigation Stacks und spielt eine zentrale Rolle in der autonomen Robotik. Das Paket wurde entwickelt, um autonomen Robotern die Fähigkeit zur intelligenten Pfadplanung und Hindernisvermeidung zu verleihen, wodurch sie in komplexen und dynamischen Umgebungen sicher navigieren können. [27]

Die Hauptaufgabe des `move_base`-Pakets besteht darin, eine optimale Bewegungsbahn für den Roboter zu planen, um von einem Startpunkt zu einem Zielpunkt in der Umgebung zu gelangen, während Hindernisse vermieden werden. Dies geschieht durch die Integration von globaler und lokaler Pfadplanung, um sicherzustellen, dass der Roboter sowohl langfristige Navigationsziele als auch unmittelbare Hindernisse effektiv berücksichtigt. [27]

In Abbildung 30 ist eine Übersicht über das `move_base`-Paket und dessen Funktionen innerhalb des ROS Navigation Stack zu sehen. Im Folgenden wird kurz auf die einzelnen Komponenten und ihr Zusammenspiel eingegangen.

`global_costmap`: Die globale Costmap beinhaltet umfangreiche Informationen bezüglich der Kosten und Traversierbarkeit verschiedener Bereiche der Umgebung. Ihre primäre Funktion liegt darin, als Basis für den globalen Pfadplaner zu dienen, um den optimalen

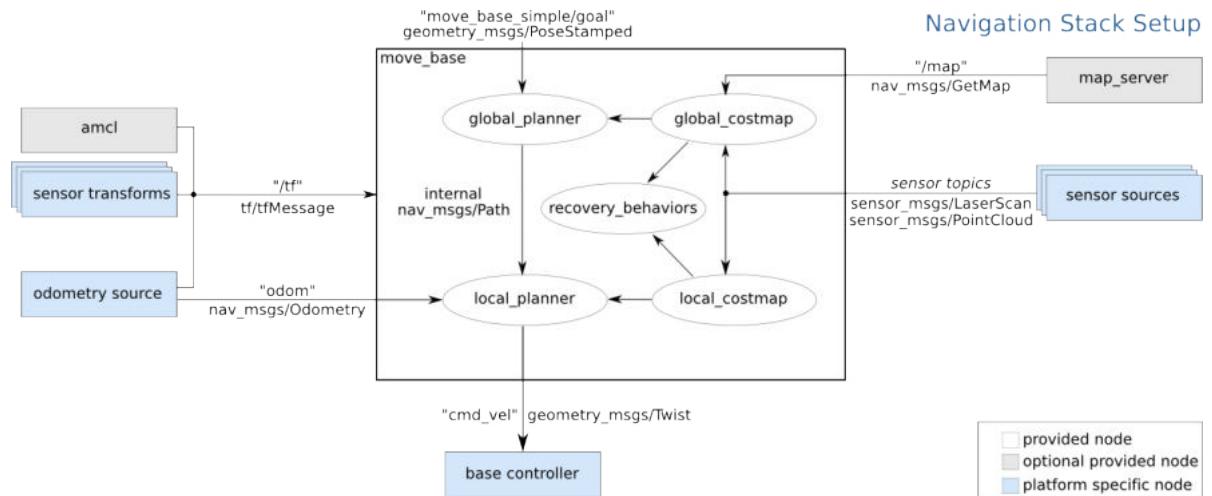


Abbildung 30: Übersicht `move_base` [27]

Pfad durch die Umgebung zu ermitteln. Um diese Karte zu erhalten, gibt es diverse Ansätze. Einerseits ist es möglich, die Karte durch Sensordaten zu generieren, wie es bereits im Kapitel des `costmap_2d`-Pakets illustriert wurde. Andererseits besteht die Option, die Karte von einem Map Server bereitstellen zu lassen. Hierbei können sowohl zuvor aufgezeichnete statische Karten als auch dynamische Karten unterschiedlicher Quellen verwendet werden. [14]

local_costmap: Die lokale Costmap beinhaltet in ähnlicher Weise wie die globale Costmap umfangreiche Informationen über die unmittelbare Umgebung des Roboters. Dieses Kartenmodell wird vom lokalen Pfadplaner genutzt, um geschickt Hindernissen auszuweichen und sich sicher durch das Gelände zu bewegen. In puncto Erhalten der Karte eröffnen sich auch hier dieselben Möglichkeiten wie bei der globalen Costmap. [14]

recovery_behaviors: Die `recovery_behaviors` werden aktiviert, wenn der Pfadplaner auf ein Hindernis oder eine nicht erwartete Situation stößt. Diese Verhaltensmuster sind darauf ausgerichtet, den Roboter aus solchen Engpässen zu befreien und ihn in einen sicheren Zustand zurückzubringen. Beispiele für solche Verhaltensweisen sind das Zurücksetzen des Roboters auf eine vorherige Position oder das Durchführen von Umkehrmanövern. [27]

global_planner: Die Hauptaufgabe des `global_planner` besteht darin, auf Grundlage der globalen Costmap einen optimalen Pfad von einem gegebenen Startpunkt zu einem Ziel zu berechnen. Dabei berücksichtigt er die Traversierbarkeit verschiedener Bereiche der Umgebung sowie die zugrunde liegenden Kostenwerte, die in der globalen Costmap gespeichert sind. [27]

local_planner: Die primäre Aufgabe des `local_planner` liegt darin, einen reaktiven Pfad zu generieren, welcher geschickt Hindernissen ausweicht und simultan den vorgeesehenen Kurs des `global_planner` beibehält. Diese reaktive Methodik erweist sich als essenziell, um auf unerwartete Veränderungen der Umgebung während der Roboterbewegung adäquat zu reagieren. Die Informationsquelle für den `local_planner` bildet hierbei sowohl die lokale Costmap als auch die gegenwärtige Position des Roboters. [27]

Nachdem im vorherigen Abschnitt die Struktur und die zentralen Komponenten des ROS-Pakets `move_base` behandelt wurden, richtet sich der Fokus nun auf die vorbereitenden Schritte, den Systemaufbau sowie den eigentlichen Test dieses Pakets.

Vorbereitung

In diesem Abschnitt wird erläutert, welche Schritte und Überlegungen vor dem eigentlichen Test des `move_base`-Pakets inklusive Handhabung der Karte unternommen wurden.

ElevationMap in Costmap

Wie bereits erwähnt wurde, verwenden sowohl `move_base` als auch `explore_lite` eine Costmap vom Typ `nav_msgs/OccupancyGrid`. Daher ist es notwendig, die vom `manipulate_map`-Knoten erstellte Karte in diesen Nachrichtentypen zu transformieren. Hierbei zeigt sich erneut eine der Stärken des `grid_map`-Pakets. Dieses Paket beinhaltet einen bereits vorgefertigten Knoten, um die Nachricht des Typs `grid_map_msgs/GridMap` in verschiedene andere Nachrichtentypen zu konvertieren. Die Aufgabe besteht nun darin, diesen Knoten unter entsprechender Konfiguration zu starten. Die hierbei wichtigsten Konfigurationsparameter sind in Abbildung 31 zu sehen. Die meisten Parameter in dieser Datei sind selbsterklärend. Eine kurze Erläuterung ist jedoch für die Parameter `data_min` und `data_max` erforderlich. Diese Parameter dienen dazu, den Wertebereich festzulegen, auf dessen Grundlage die Höhenwerte in Kostenwerte umgewandelt werden. In diesem speziellen Fall fällt jedoch auf, dass der `data_min`-Parameter größer ist als der `data_max`-Parameter. Dies ergibt sich aus der Tatsache, dass die Hindernisse negativ sind. Die gewählten Zahlenwerte für diese Parameter basieren auf der Tatsache, dass der Parcours in der Simulation eine Höhe von 0,15 m aufweist. Der Bereich, in dem die Höhenwerte in Kostenwerte umgewandelt werden, erstreckt sich nun von 0,14 m bis 0,13 m. Alle Werte, die kleiner als 0,13 m sind, erhalten automatisch den maximalen Kostenwert und werden somit als Hindernisse gekennzeichnet.

Zur Durchführung eines Tests dieses Knotens wird dieser mit entsprechenden Parametern gestartet und der Roboter entlang des Parcours bewegt. Das Ergebnis ist in Abbildung 32

```
grid_map_topic: /elevation_map_manipulated
grid_map_visualizations:
  - name: elevation_grid
    type: occupancy_grid
    params:
      layer: elevation_fused
      data_min: 0.14
      data_max: 0.13
```

Abbildung 31: Ausschnitt der Konfigurationsdatei für die Konvertierung der GridMap

dargestellt. Auf der linken Seite der Abbildung ist die aufgenommene Höhenkarte zu sehen, während auf der rechten Seite die durch Konvertierung erzeugte Costmap abgebildet ist. Der rosa Bereich repräsentiert Hindernisse, also den Boden. Dies bestätigt die erfolgreiche Funktionalität der Konvertierung.

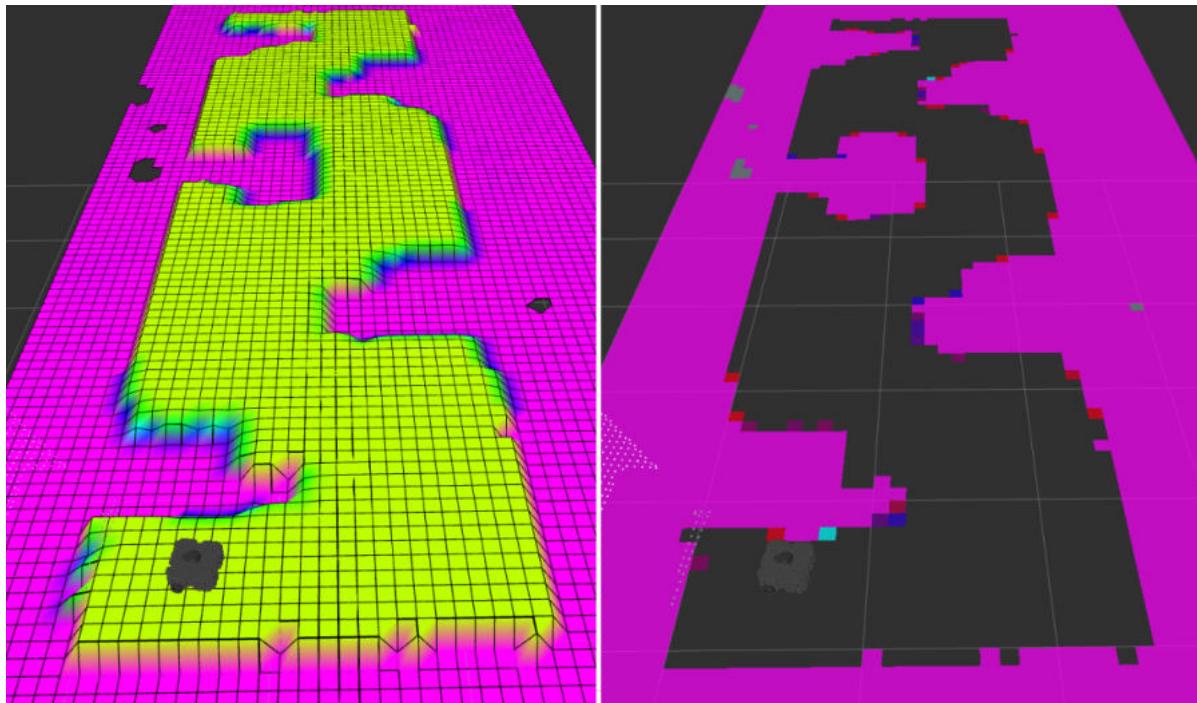


Abbildung 32: Konvertierung der Höhenkarte (links) in eine Costmap (rechts)

Konfiguration move_base

Obwohl `move_base` einen vorgefertigten Knoten besitzt, setzt sich dieser aus verschiedenen Komponenten zusammen, wie in Abbildung 30 dargestellt. Um den `move_base`-Knoten erfolgreich zu initiieren, bedarf es einer erheblichen Konfigurationsarbeit. Diese Aufgabe

wird durch die Erstellung mehrerer YAML-Files bewerkstelligt, wobei jede Datei für eine spezifische Komponente zuständig ist. Im folgenden Abschnitt wird ein kurzer Überblick über die einzelnen Dateien, ihre entsprechenden Komponenten und die wesentlichen Parameter gegeben.

Die erste anzulegende Datei, zu sehen in Abbildung 33, ist die `costmap_common_params`-Datei. In dieser Datei werden Parameter definiert, die sowohl für die globale als auch für die lokale Costmap gelten. Dies umfasst Einstellungen wie Kartenauflösung oder die Festlegung des Radius der Pufferzone um Hindernisse.

```
obstacle_range: 2.5          # maximum range for obstacle being put into the costmap
raytrace_range: 3.0          # range to which the robot attempt to clear out space
robot_radius: 0.23          # robot footprint
resolution: 0.1
inflation_radius: 0.1
```

Abbildung 33: Ausschnitt der Konfigurationsdatei `costmap_common_params`

Die nächste Datei ist die `global_costmap_params`-Datei, zu sehen in Abbildung 34. Diese Datei dient der Spezifikation von Parametern, die ausschließlich die globale Costmap betreffen. Sie erfordert grundlegende Einstellungen, darunter die Definition des Koordinatensystems, in dem die Costmap betrieben wird, sowie die Festlegung des Bezugskoordinatensystems für die Roboterbasis. Des Weiteren wird entschieden, ob die Karte selbstständig erstellt oder vom Map Server bereitgestellt wird, wobei hier die Nutzung des Map Servers gewählt wird. Zusätzlich wird festgelegt, dass die Karte in stationärer Position verbleibt und sich nicht mit dem Roboter bewegt.

Neben den Grundkonfigurationen wird in dieser Datei auch der Map Server mithilfe von Plugins eingerichtet. Das erste Plugin steuert unter anderem, aus welchem Topic der Server die Karte bezieht, in diesem Fall die konvertierte Karte. Das zweite Plugin dient dazu, die Pufferzone um die Karte des ersten Plugins zu erzeugen.

Nach der Erstellung der globalen Costmap-Datei wird nun auch eine Datei für die lokale Costmap mit dem Namen `local_costmap_params` erstellt. Diese Datei ähnelt den Einstellungen und Parametern der vorherigen Datei sehr, jedoch hat der Parameter `static_map` nun den Wert `false` und `rolling_window` den Wert `true`. Diese Unterschiede ergeben sich daraus, dass die lokale Karte nur einen Ausschnitt der gesamten Karte verwendet und sich mit dem Roboter mitbewegen soll. Dies ist notwendig, da die lokale Karte die Hindernisvermeidung übernimmt und dabei alle Hindernisse der

```

global_costmap:
  global_frame: odom
  robot_base_frame: base_link
  static_map: true           # use map server
  rolling_window: false      # map remains in place
  width: 5
  height: 10

  plugins:
    - {name: static_layer, type: "costmap_2d::StaticLayer"}
    - {name: inflation_layer, type: "costmap_2d::InflationLayer"}

  static_layer:
    enabled: true
    map_topic: "/grid_map_visualization/elevation_grid"

  inflation_layer:
    enabled: True

```

Abbildung 34: Ausschnitt der Konfigurationsdatei `global_costmap_params`

Karte berücksichtigt. Hierbei zählt jedes einzelne Feld, das als Boden erkannt wird, als Hindernis. Wenn die gesamte Karte auch hier verwendet würde, entstünde ein enormer Rechenaufwand.

Anschließend folgt die Möglichkeit, die Datei `global_planner_params` zu erstellen. In dieser Datei können Einstellungen für den `global_planner` vorgenommen werden. Da jedoch in diesem Fall keine Änderungen an der Grundkonfiguration erforderlich sind, bleibt diese Datei leer und kann bei Bedarf auch weggelassen werden.

Abschließend sollte eine Konfigurationsdatei für den `local_planner` erstellt werden, die den Namen `base_local_planner_params` trägt. Ein Auszug aus dieser Datei findet sich in Abbildung 35. In der Datei werden verschiedene Einstellungen für den `local_planner` vorgenommen, darunter Parameter zur Begrenzung der maximalen und minimalen Geschwindigkeit sowie zur Steuerung der Beschleunigung des Roboters.

```
TrajectoryPlannerROS:  
  max_vel_x: 0.2  
  min_vel_x: 0.1  
  max_vel_theta: 0.4  
  min_in_place_vel_theta: 0.1  
  
  acc_lim_theta: 3.2  
  acc_lim_x: 2.5  
  acc_lim_y: 2.5
```

Abbildung 35: Ausschnitt der Konfigurationsdatei `base_local_planner_params`

Systemaufbau

In Abbildung 36 ist der vereinfachte Systemaufbau dargestellt, der für den anschließenden Test verwendet wurde. Dieser Aufbau baut auf dem Test des `manipulate_map`-Knotens auf und erweitert ihn um zusätzliche Knoten. Die erste Erweiterung umfasst den `grid_map_visualization`-Knoten, der die Aufgabe hat, die Karte in den erforderlichen Nachrichtentyp umzuwandeln. Die konvertierte Karte wird anschließend an den nächsten hinzugefügten Knoten, `move_base`, weitergeleitet. Dieser Knoten benötigt neben der Karte auch die Informationen aus den Topics `/odom` und `/tf`. Der Systemaufbau schließt mit dem Knoten für RViz ab, der bereits zuvor zur Visualisierung der Daten genutzt wurde, nun jedoch aktiv mit dem Gesamtsystem interagiert. Über RViz wird der Zielpunkt der Navigation mittels des Topics `/move_base_simplegoal` veröffentlicht. `move_base` verarbeitet diesen Zielpunkt zusammen mit den anderen Informationen und wandelt sie in Bewegungsbefehle um. Diese Befehle werden über das Topic `cmd_vel` an den Roboter übertragen, um seine Steuerung zu ermöglichen.

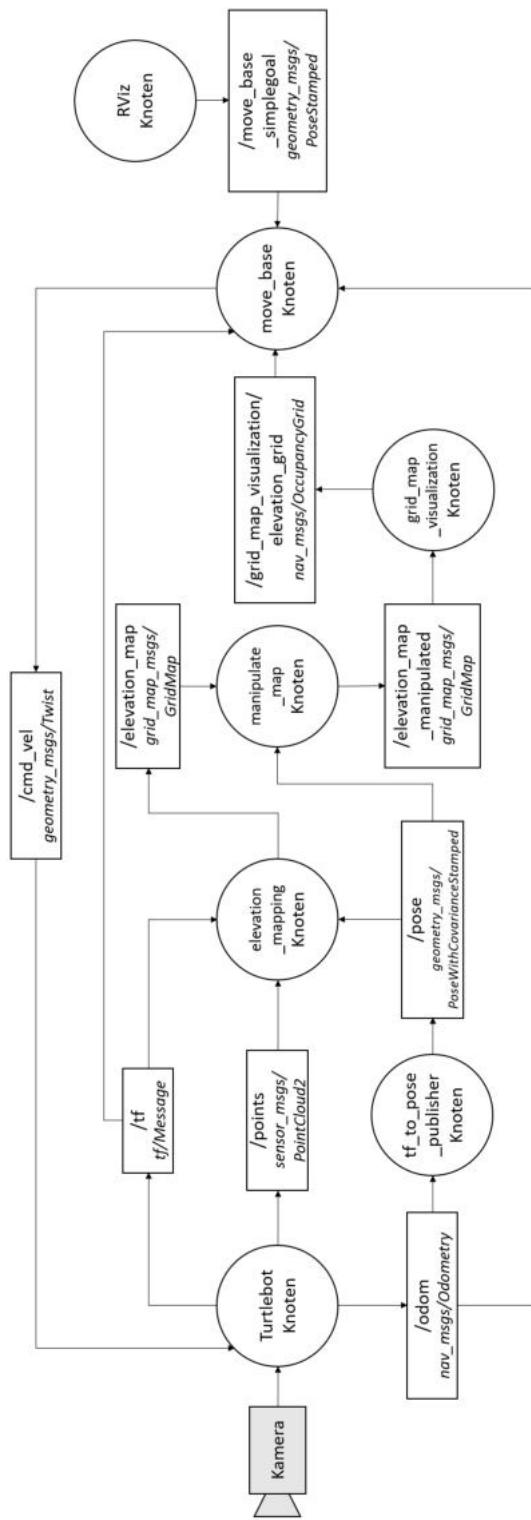


Abbildung 36: Vereinfachter Systemaufbau `move_base`

Test

Nachdem alle erforderlichen Konfigurationen abgeschlossen sind, kann mit dem Testen begonnen werden. Das Ziel dieses Tests ist es, dem Roboter über RViz einen Zielpunkt anzugeben, zu welchem er autonom navigieren soll. Der Testablauf gestaltet sich wie folgt: Zunächst wird das gesamte System gestartet. Daraufhin wird der Roboter manuell gesteuert, um eine vollständige 360-Grad-Drehung durchzuführen. Dies ermöglicht einen umfassenden Überblick über die unmittelbare Umgebung. Anschließend wird über RViz ein Zielpunkt im noch nicht kartierten Bereich der Karte festgelegt. Der Roboter sollte nun eigenständig zu diesem Ziel navigieren.

In Abbildung 37 wird der Beginn des Tests veranschaulicht. Die globale Costmap repräsentiert die gesamte erkennbare Karte, während die lokale Costmap nur bei genauerem Hinsehen erkennbar ist. Letztere bildet das quadratische Feld um den Roboter herum, dass aufgrund der Überlagerung heller erscheint. Zudem ist die unmittelbare Umgebung des Roboters bereits durch die zuvor ausgeführte 360-Grad-Drehung kartiert worden. Der grüne Kreis um den Roboter herum zeigt die Fußabdruckfläche des Roboters, die bei der Pfadplanung des `local_planner` berücksichtigt wird. Der obere rote Pfeil kennzeichnet das Ziel, dass in einem noch unerforschten Teil der Karte platziert wurde. Die Richtung des Pfeils gibt an, wie der Roboter im Zielpunkt ausgerichtet sein soll. Ebenfalls sichtbar ist eine grüne Linie, die den Pfad des `global_planner` darstellt. Auch der Pfad des `local_planner` ist hier als kurze orangene Linie sichtbar. Die fortlaufende Testsequenz ist in mehreren Screenshots in Abbildung 38 dargestellt.

Der erste Screenshot zeigt den Roboter bei der Durchfahrt der ersten Kurve. Im zweiten Bild ist der Roboter nach der ersten Kurve zu sehen. Ein größerer Zeitsprung führt zum dritten Bild, das den Roboter nach der zweiten Kurve zeigt. Hier wird eine der Herausforderungen der Navigation in unbekannten Umgebungen deutlich. Der `global_planner` besitzt keinerlei Informationen über die Umgebung um den Zielpunkt herum und plant daher eine direkte Strecke zum Ziel. Allerdings führt dieser Pfad durch mehrere Hindernisse (Bodenflächen), wie im Bild ersichtlich. Im darauffolgenden vierten Screenshot wird gezeigt, wie `move_base` dieses Problem automatisch löst und einen neuen Pfad berechnet. Der nächste Screenshot präsentiert den Roboter auf dem neu errechneten Pfad. Im letzten Bild ist der Roboter schließlich im Zielpunkt angekommen, wobei auch die korrekte Ausrichtung beachtenswert ist.

Dieser Test wird mehrfach wiederholt, um ein umfassendes Bild von der Konsistenz, den Stärken und Schwächen des `move_base`-Knotens zu erhalten. Dabei fallen mehrere

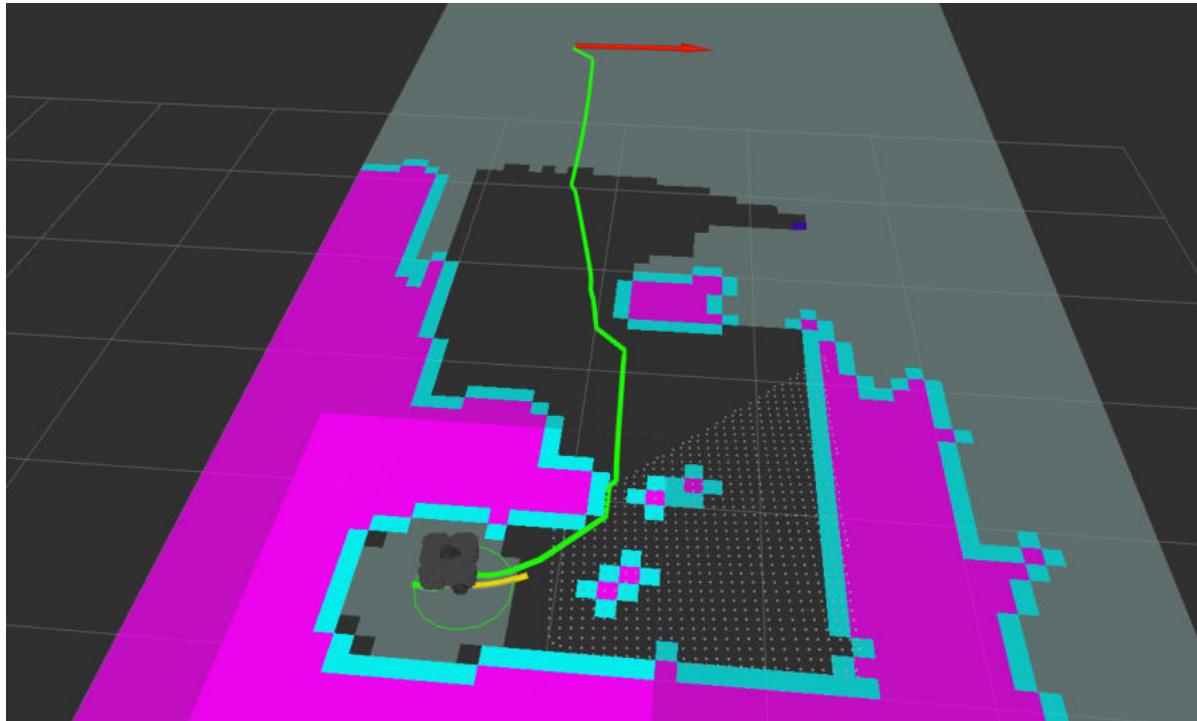


Abbildung 37: Start des Tests von `move_base`

Punkte auf. Grundsätzlich funktioniert die Navigation gut. Es kommt jedoch gelegentlich vor, dass sich der Roboter an Kanten kurzzeitig festfährt und eine gewisse Zeit benötigt, um sich wieder zu stabilisieren. Bemerkenswert ist auch, dass die Navigation gut mit Störungen umgehen kann. In Abbildung 37 oder im zweiten Screenshot von Abbildung 38 ist zu erkennen, dass einige Störungen als Hindernisse erkannt wurden. Trotzdem gerät die Navigation dadurch nicht aus der Bahn. Zum Abschluss ist zu beachten, dass die Wahl des Fußabdrucks erheblichen Einfluss auf die Navigationseffizienz hat. In einem der Tests wurde anstelle eines runden Fußabdrucks ein viereckiger gewählt, was zu häufigeren und stärkeren Blockaden des Roboters führte.

Zusammenfassend lässt sich feststellen, dass der `move_base`-Knoten wie erwartet funktioniert und zufriedenstellende Ergebnisse liefert.

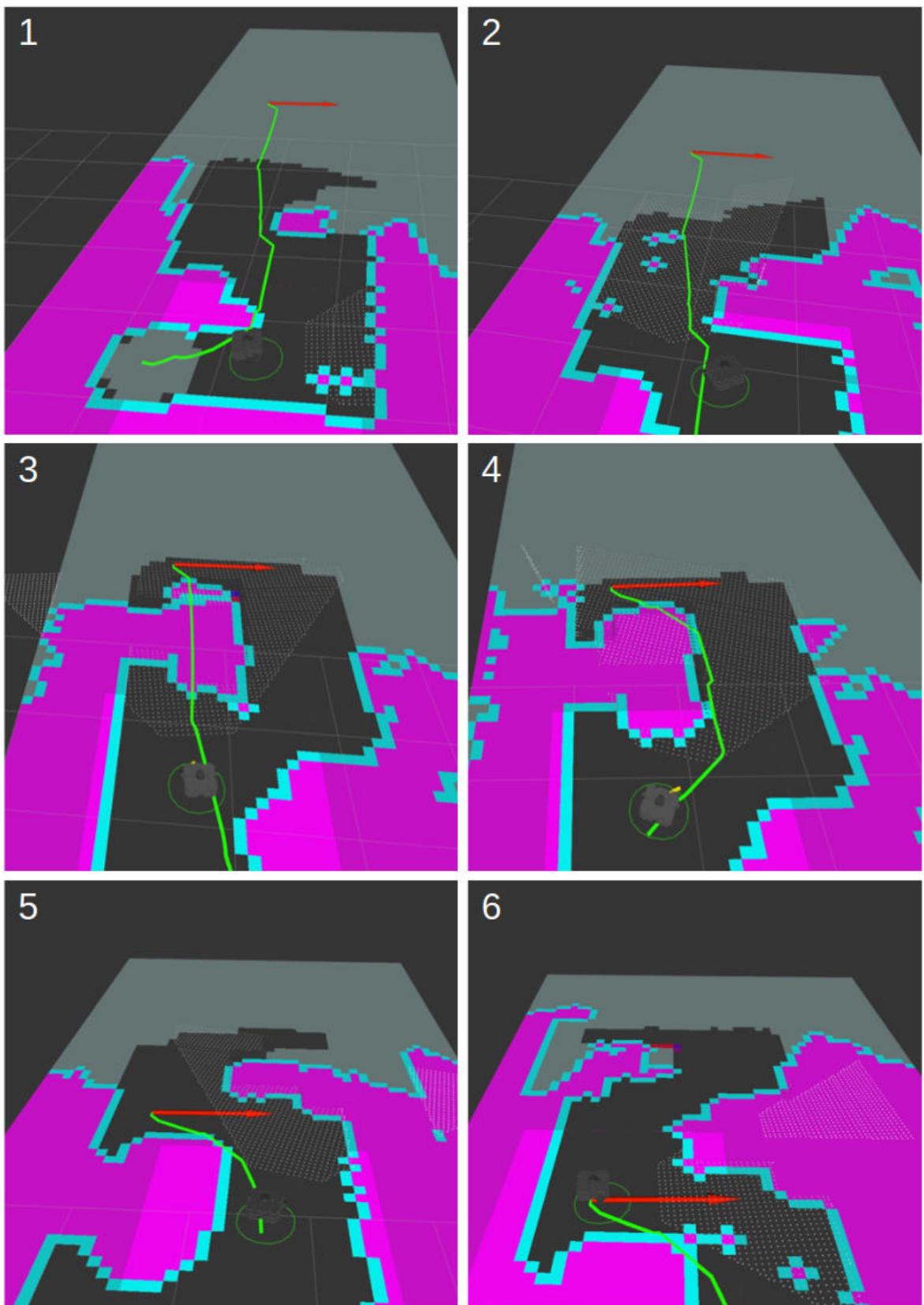


Abbildung 38: Ablauf des Tests von `move_base`

3.6 explore _ lite

Das ROS-Paket `explore_lite` bietet eine effiziente und anpassbare Lösung für die autonome Erkundung von unbekannten Umgebungen. Die Erkundungsstrategie des Pakets basiert auf dem „greedy frontier exploration“-Prinzip. Dieses Prinzip zielt darauf ab, dass ein Roboter eine unbekannte Umgebung systematisch und effizient erkundet, indem er potenzielle „Frontiers“ identifiziert und darauf zusteuer. Frontiers sind Grenzbereiche zwischen bekannten und unbekannten Gebieten, die eine vielversprechende Möglichkeit bieten, neue Informationen über die Umgebung zu sammeln.

Bei der „greedy“, also gierigen Frontier-Exploration, wählt der Roboter stets den nächstgelegenen oder den am meisten vielversprechendsten Grenzbereich als Ziel aus und plant seine Bewegungen, um dieses Ziel zu erreichen. Sobald ein Grenzbereich erreicht ist oder nicht mehr erreichbar erscheint, wird ein neuer Grenzbereich ausgewählt und der Prozess wiederholt sich. [26]

Im Gegensatz zu ähnlichen Paketen zeichnet sich `explore_lite` durch seine Ressourceneffizienz aus. Statt eine eigene Costmap zu erstellen, abonniert der Knoten Nachrichten vom Typ `nav_msgs/OccupancyGrid`. Die Bewegungsbefehle für den Roboter werden an den `move_base`-Knoten gesendet, was eine schlankere und effizientere Arbeitsweise gewährleistet. [26]

Vorbereitung

Wie bereits `move_base` ist `explore_lite` ein fertiges Paket mit einem bereits vorhandenen Knoten. Dies bedeutet auch hier werden lediglich Parameter für die Konfiguration des Knotens belegt. Im Vergleich zu `move_base` ist dieser Schritt jedoch weniger aufwendig, da keine YAML-Files erstellt werden müssen. Es genügt, einige Parameter im Launch-File anzupassen. Der relevante Ausschnitt des Launch-Files ist in Abbildung 39 zu sehen.

Der erste Schritt umfasst die Zuweisung des Parameters für das Roboter-Koordinatensystem. Ein weiterer wichtiger Parameter ist das Topic der Costmap, auf dem die Exploration durchgeführt werden soll. Hier wird das Topic gewählt, welches die von `move_base` erstellte Costmap repräsentiert. Die übrigen Parameter dienen vor allem der Feinabstimmung und können in der Regel auf ihren Standardwerten belassen werden.

```

<node pkg="explore_lite" type="explore" respawn="false" name="explore" output="screen">
  <param name="robot_base_frame" value="base_footprint"/>
  <param name="costmap_topic" value="/move_base/global_costmap/costmap"/>
  <param name="costmap_updates_topic" value="/move_base/global_costmap/costmap_updates"/>
  <param name="min_frontier_size" value="0.01"/>
</node>

```

Abbildung 39: Ausschnitt der Konfigurationsdatei für `explore_lite`

Systemaufbau

In Abbildung 40 wird der relevante Ausschnitt des Systemaufbaus für den Test des `explore_lite`-Knotens dargestellt. Der übrige Systemaufbau entspricht dem des Tests zu `move_base`, welcher in Abbildung 36 zu sehen ist. Der Unterschied zu dem vorherigen Test liegt darin, dass der RViz-Knoten durch den `explore_lite`-Knoten ersetzt wurde. Dieser Knoten empfängt nun über `move_base` die zu verwendende Costmap. Die Zielkoordinaten werden nicht wie bei RViz über ein Topic übertragen, sondern stattdessen über eine Action. Dies ist notwendig, da die Exploration ein zeitaufwendiger Prozess ist, der eine kontinuierliche Kommunikation zwischen den beiden Knoten erfordert.

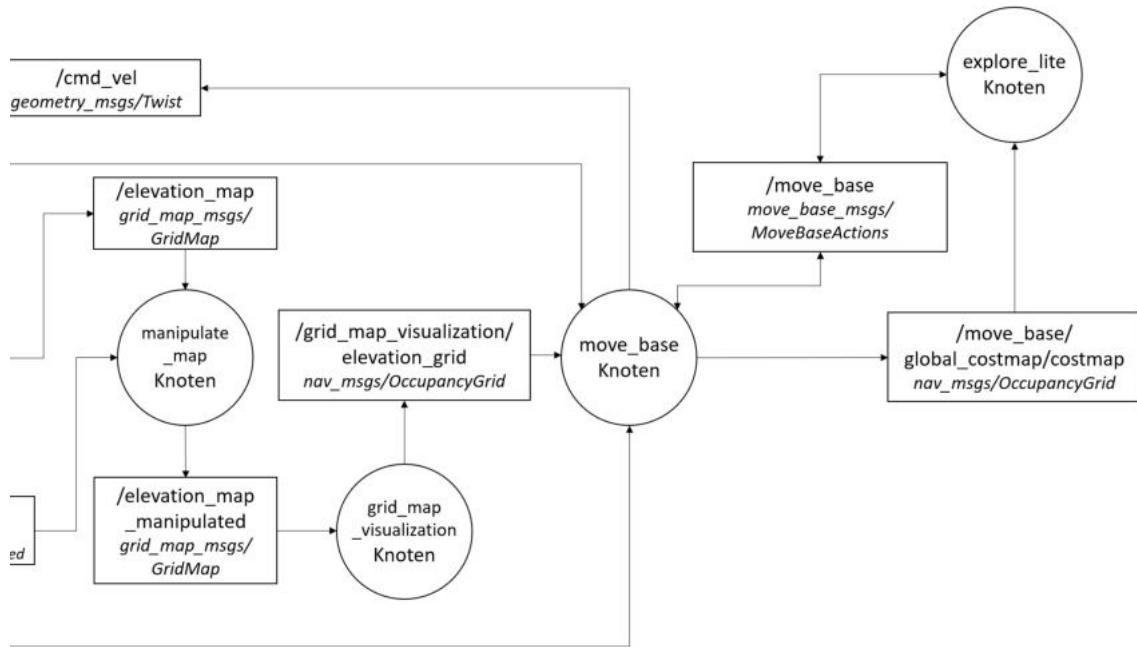


Abbildung 40: Vereinfachter Systemaufbau `explore_lite`

Test

Nachdem alle erforderlichen Konfigurationen abgeschlossen sind, kann mit dem Testen begonnen werden. Das Ziel dieses Tests ist es, die Exploration zu starten, sodass der Roboter den gesamten Parcours abfährt und dabei eine Karte aufzeichnet. Der Testablauf gestaltet sich wie folgt: Zunächst wird wie bei `move_base` das gesamte System bis auf den Explorationsknoten gestartet. Auch hier wird einfacheitshalber eine 360-Grad-Drehung zum Überblick über die nähere Umgebung ausgeführt. Anschließend kann nun der `explore_lite`-Knoten gestartet werden. Dieser sollte direkt starten und mit der Exploration beginnen.

In Abbildung 41 ist nun der Moment kurz nach dem Start des Explorationsknoten zu sehen. In blau sind hier die Frontiers zu erkennen. Die grünen Kugeln sind die Frontiers welche als Zielauswahl für den `global_planner` in Frage kommen. Auf der nachfolgenden Seite sind auch hier wieder einige Screenshots des Tests zu sehen.

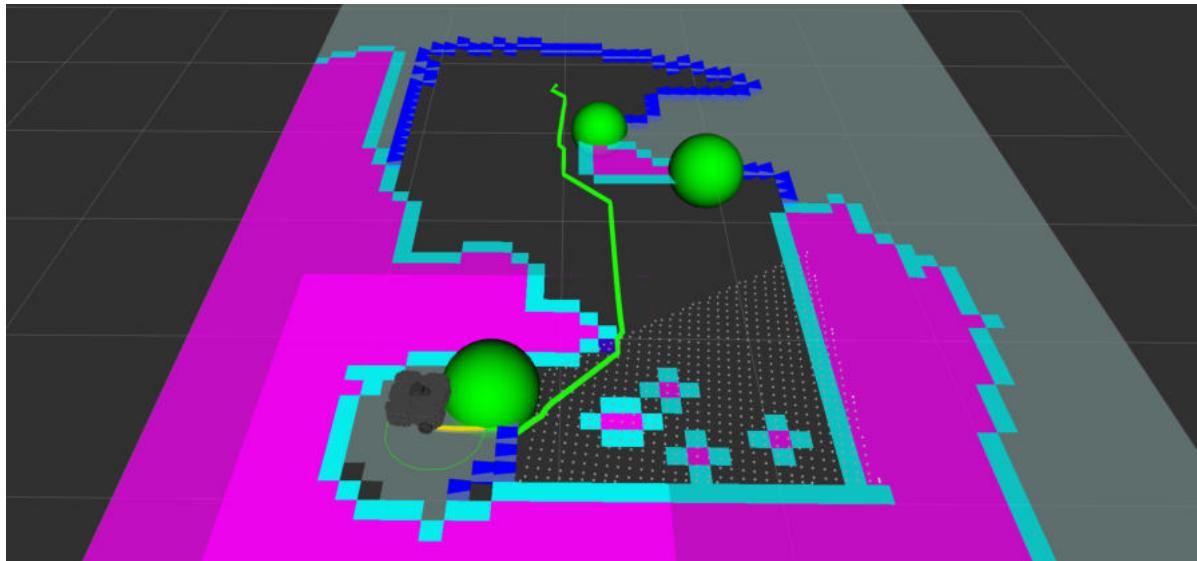


Abbildung 41: Start der Exploration

Der Ablauf der Exploration ist in Abbildung 42 in mehreren Screenshots dargestellt. In den ersten beiden Screenshots kann man erkennen, wie sich der Roboter von seinem Startpunkt weg bewegt, sich umdreht und den noch ungemappten Startbereich kartiert. Anschließend setzt der Roboter in den folgenden Screenshots den Mappingvorgang fort.

In Abbildung 43 kann man nun die finale Karte erkennen, bei der der Roboter das Mapping als beendet angesehen hat. Dieses Ergebnis ist zufriedenstellend, da der gesamte Parcours aufgezeichnet wurde.

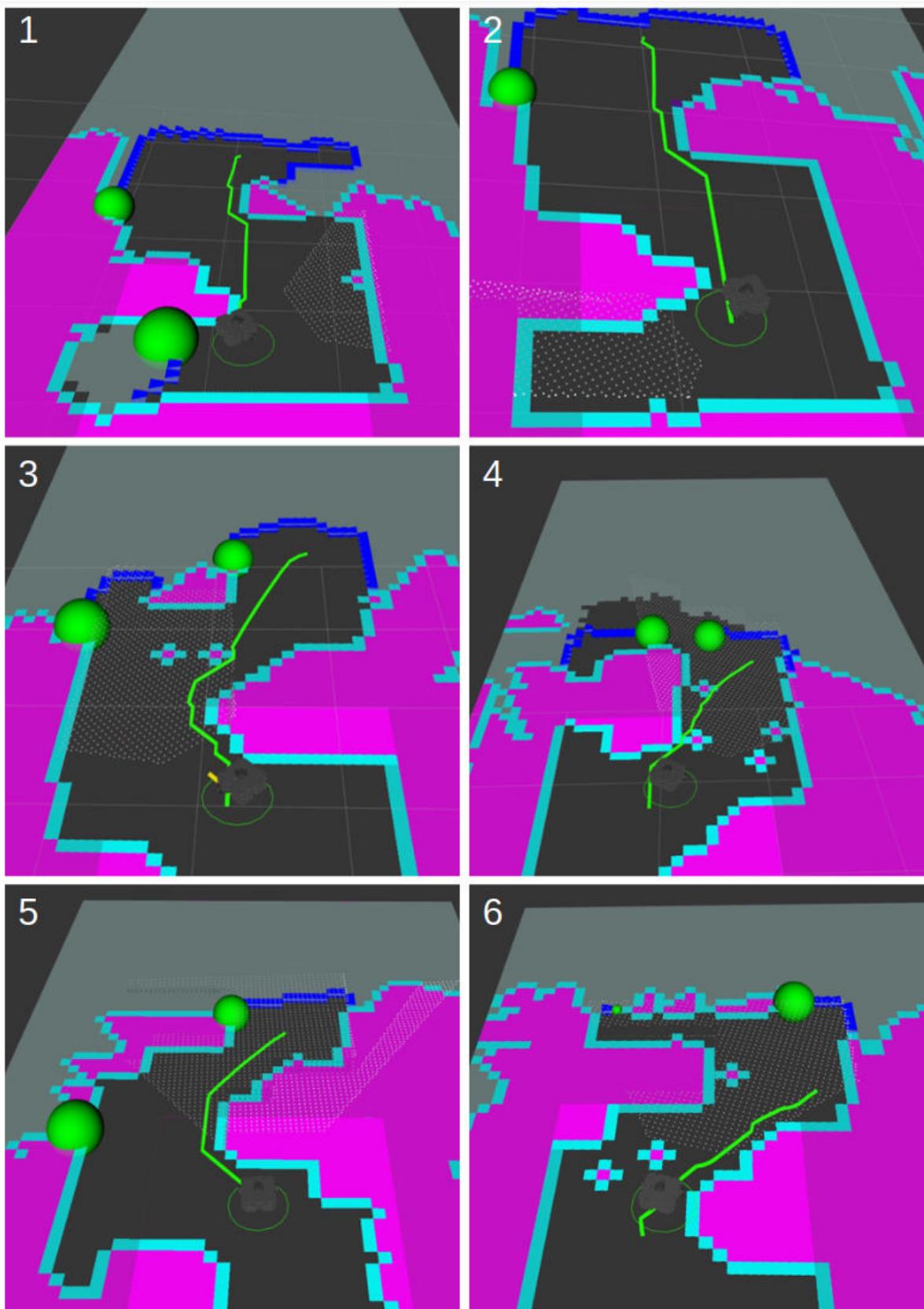


Abbildung 42: Ablauf der Exploration

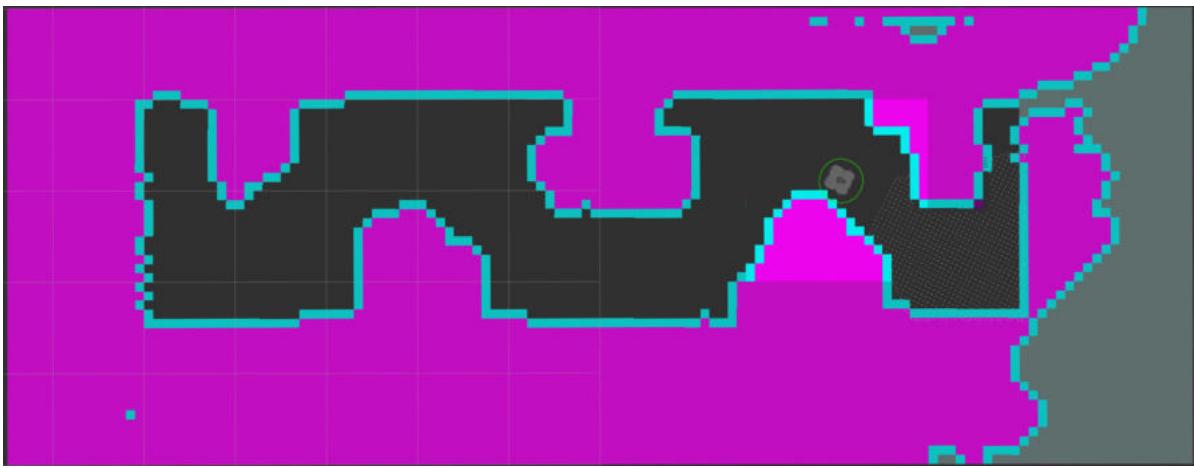


Abbildung 43: Von `explore_lite` erstellte Karte

Dieser Test wird ebenfalls mehrfach wiederholt, um ein umfassenderes Verständnis zu erlangen. Dabei zeigt sich, dass die Exploration in den meisten Fällen zuverlässig funktioniert. Gelegentlich kann es jedoch vorkommen, dass die Navigation der Exploration stockt, was nur durch einen Neustart des Explorationsknotens behoben werden kann. Hierbei wird auch das größte Problem dieses Knotens sichtbar. Aufgrund fehlender Kommunikationsmöglichkeiten mit dem Knoten muss dieser bei Problemen manuell über das Terminal neu gestartet werden.

Zusammenfassend lässt sich dennoch feststellen, dass der `explore_lite`-Knoten wie erwartet funktioniert und zufriedenstellende Ergebnisse liefert.

4 Testaufbau für Vorversuche

Nach der erfolgreichen Simulation der Navigationssysteme steht nun die praktische Erprobung auf echter Hardware an. In diesem Kapitel wird sich darauf konzentriert, die entwickelten Navigationsalgorithmen und -strategien auf dem Kobuki-Roboter in Verbindung mit der Astra 3D-Kamera zu testen. Die Übertragung der Simulationsergebnisse auf die reale Welt stellt einen entscheidenden Schritt dar, der mit einer Vielzahl von Herausforderungen und unbekannten Variablen verbunden ist. Im Verlauf dieses Kapitels wird detailliert auf die Testumgebung, den Parcours und dessen Aufbau sowie auf die durchgeführten Tests mit dem Kobuki-Roboter eingegangen. Dies ermöglicht, die Robustheit, Effizienz und Verlässlichkeit der Navigationslösungen unter realen Bedingungen zu überprüfen und wertvolle Erkenntnisse für zukünftige Optimierungen zu gewinnen.

4.1 Parcours

In diesem Kapitel wird der Aufbau eines Testparcours beschrieben. Dabei sind verschiedene Faktoren zu berücksichtigen. Zum einen steht lediglich ein Dachboden als Testumgebung zur Verfügung. Zum anderen ist das Ziel, dass der Testparcours die gleichen Schwierigkeiten aufweist wie der Wettbewerbsparcours. Hierbei gilt es, die realen Gegebenheiten bestmöglich nachzustellen.

4.1.1 Material

Für den Aufbau eines Parcours sind verschiedene Materialfaktoren von entscheidender Bedeutung. Zunächst sollte das Material in einer geeigneten Form vorliegen, idealerweise in Form von quadratischen oder rechteckigen Blöcken, um einen modularen Aufbau des Parcours zu ermöglichen.

Die Stabilität des Materials ist ebenfalls von großer Wichtigkeit, da es während der Befahrung durch den Roboter nicht beschädigt werden sollte. Die Oberflächenbeschaffenheit des Materials sollte einerseits ausreichend Griffigkeit bieten, um ein Durchdrehen der Roboterreifen zu verhindern, andererseits jedoch nicht zu grob sein, um eine reibungslose Manövriertfähigkeit des Roboters zu gewährleisten.

Das Gewicht der verwendeten Materialien spielt eine entscheidende Rolle, da der Parcours auf einem Holzdachboden aufgebaut wird. Dabei ist darauf zu achten, dass das Material nicht zu schwer ist und somit die Stabilität des Dachbodens gefährden würde.

Auch muss der Aufwand berücksichtigt werden, der für den Transport und den Aufbau der Materialien erforderlich ist. Hierbei ist zu beachten, dass das Material durch ein Treppenhaus und anschließend über eine ausziehbare Leiter durch eine Luke in den Dachboden befördert werden muss. Zudem sollte der Aufbau des Parcours möglichst einfach sein, idealerweise durch das direkte Zusammenfügen der Materialien ähnlich wie Bausteine.

Abschließend sind auch finanzielle Überlegungen von Bedeutung. Dank der großzügigen Unterstützung eines örtlichen Baumarktes können die Kosten jedoch begrenzt werden. Während der Testdurchführung wird daher aus Gründen der Nachhaltigkeit besonders darauf geachtet, das bereitgestellte Material nicht zu beschädigen, damit es später in gutem Zustand zurückgegeben werden kann.

Aufgrund dieser unterschiedlichen Aspekte werden nun im Folgenden mehrere Materialien in Betracht gezogen und bewertet.

Betonblöcke: Als mögliche Option für den Parcours-Aufbau werden zuerst Betonblöcke in Betracht gezogen, wie sie in Abbildung 44 zu sehen sind. Diese Blöcke werden auch für den Aufbau des Parcours beim RoboCup verwendet. Obwohl Betonblöcke gute Eigenschaften wie passende Form, geeignete Oberflächenbeschaffenheit und Stabilität aufweisen, erweisen sie sich für den Einsatz auf dem Dachboden als ungeeignet. Das hohe Gewicht der Blöcke spielt hierbei eine entscheidende Rolle. Zum einen erfordert es einen erheblichen Aufwand, jeden einzelnen Stein in den Dachboden zu transportieren. Noch bedeutender ist jedoch, dass das Gesamtgewicht des Parcours zu hoch für die Tragfähigkeit des Dachbodens ist. Dies könnte zu Beschädigungen oder Instabilität führen.

Holz: Eine weitere Option ist der Bau des Parcours aus Holz. Obwohl Holz ebenfalls Eigenschaften wie Stabilität und passende Oberflächenbeschaffenheit aufweist, erweist es sich in den meisten anderen Aspekten als ungeeignet. Holz wird hauptsächlich in Form von Brettern angeboten und nicht als Blöcke, dies bedeutet, dass der Parcours aus



Abbildung 44: Betonblock für den Aufbau eines Parcours [22]



Abbildung 45: Porenbetonblock [31]

mehreren Brettern zusammengesetzt werden müsste, entweder durch Schrauben oder Kleben. Dies würde jedoch einen erheblichen Aufwand bedeuten. Auch in Bezug auf die Kosten ist Holz keine ideale Wahl. Da die Möglichkeit des Ausleihens von Holzmaterialien nicht gegeben ist, würden erhebliche Kosten für den Kauf anfallen.

Porenbetonblöcke: Ähnlich wie Betonblöcken werden auch Porenbetonblöcke, in Abbildung 45 zu sehen, als Option in Betracht gezogen. Auch diese Steine besitzen eine geeignete Form und Stabilität. Zusätzlich sind sie wesentlich leichter als Betonblöcke und daher besser für den Dachboden geeignet. Trotzdem bleibt der Transport eine Herausforderung. Obwohl diese Blöcke leichter sind, erfordert es dennoch erheblichen Aufwand, die Steine durch das Haus auf den Dachboden zu transportieren. Ein weiteres Problem betrifft die Oberflächenbeschaffenheit der Porenbetonblöcke. Sie weisen eine relativ raue Oberfläche auf, wodurch das Fahren des Roboters unrund sein könnte. Ein zusätzliches Problem ist die Porosität der Blöcke. Dies macht sie anfällig für Beschädigungen. Da es wahrscheinlich ist, dass das Material ausgeliehen wird, wäre eine solche Anfälligkeit äußerst nachteilig.

Styroporblöcke: Ebenso wie die zuvor genannten Materialien weisen Styroporblöcke, zu sehen in Abbildung 46, eine geeignete Form auf. Zusätzlich ist das Gewicht dieser Blöcke erheblich geringer im Vergleich zu den bisher betrachteten Materialien. Jedoch ergibt sich eine Herausforderung hinsichtlich der Stabilität des Materials. Obwohl der Block an sich ausreichend stabil ist, um das Gewicht des Roboters zu tragen, ist die Oberfläche des Styropors anfällig für Eindrücke. Nach mehreren Tests mit dem Roboter könnten die Blöcke ausreichend beschädigt sein, um sie nicht mehr in einwandfreiem Zustand zurückgeben zu können. In Bezug auf den Aufwand wäre Styropor wiederum eine vorteilhafte Wahl, da er leicht ist und sich beim Aufbau des Parcours modular zusammensetzen lässt.



Abbildung 46: Styroporblock



Abbildung 47: Gummimatte

Gummimatten: Abschließend werden Gummimatten, wie in Abbildung 47 dargestellt, in Betracht gezogen. Hier fällt sofort auf, dass diese eine ungeeignete Form aufweisen. Obwohl sich die Matten hervorragend für einen modularen Parcoursaufbau eignen würden, sind sie in Bezug auf ihre Höhe zu niedrig. Man müsste entweder einen Unterbau anfertigen oder mehrere Matten übereinanderstapeln, um die gewünschte Höhe zu erreichen. Da die Anzahl der verfügbaren Matten aus eigenem Bestand begrenzt ist, scheidet die Option des Stapelns zusätzlich aus. Dennoch sind die Gummimatten in Bezug auf ihre restlichen Eigenschaften vielversprechend. Sie sind leicht im Gewicht, sehr stabil und weisen eine geeignete Oberflächenbeschaffenheit auf.

Aufgrund der Analyse verschiedener Materialoptionen für den Parcoursaufbau wird sich letztendlich für eine Kombination von Styroporblöcken und Gummimatten entschieden. Diese Entscheidung wurde getroffen, um die Vorteile beider Materialien optimal zu nutzen und gleichzeitig deren jeweilige Nachteile zu minimieren.

Styroporblöcke wurden aufgrund ihres geringen Gewichts und der Möglichkeit eines modularen Aufbaus ausgewählt. Die leicht zusammensetzbaren Blöcke ermöglichen eine flexible Gestaltung des Parcours und erleichtern den Auf- und Abbau erheblich. Dennoch stellte die geringe Oberflächenstabilität des Styropors eine Herausforderung dar, da wiederholte Durchfahrten des Roboters das Material beschädigen könnten.

Um diesem Problem entgegenzuwirken, werden Gummimatten als Ergänzung eingesetzt. Gummimatten verfügen über eine stabilere Oberfläche, die den reibungslosen Betrieb des Roboters gewährleistet. Durch die Wahl von Gummimatten kann die Gefahr von Beschädigungen während der Testläufe minimiert werden.

Die Kombination aus Styroporblöcken und Gummimatten ermöglicht somit eine Balance zwischen Gewicht, Stabilität und Oberflächenbeschaffenheit. Auf diese Weise kann ein Parcours aufgebaut werden, der den Anforderungen der Navigationsalgorithmen und -strategien gerecht wird und gleichzeitig die Sicherheit und Integrität der verwendeten Materialien gewährleistet.

4.1.2 Layout

Im Anschluss an die Auswahl des geeigneten Materials im vorherigen Kapitel wird nun ein passendes Layout für den Testparcours entwickelt. Aufgrund der begrenzten Fläche auf dem Dachboden ist es nicht möglich, den exakt gleichen Parcours aus der Simulation zu replizieren. Daher ist es erforderlich, sorgfältig zu überlegen, welche Merkmale und Herausforderungen der Parcours aufweisen sollte.

Kurven: Kurven im Parcourslayout dienen dazu, die Fähigkeit des Roboters zur präzisen Steuerung und zur Anpassung seiner Bewegung an die Streckenführung zu testen. Diese Kurven können unterschiedliche Radien aufweisen, was bedeutet, dass der Roboter seine Bewegung je nach Kurvenradius variieren muss. Die Anforderungen an die Navigation in Kurven sind vielfältig. Der Roboter muss nicht nur die geeignete Geschwindigkeit und den richtigen Winkel für die Kurvendurchfahrt wählen, sondern auch die Pfadplanung und die Trajektorienanpassung entsprechend angeleichen. Dabei sollte der Roboter in der Lage sein, die Kurven so zu durchfahren, dass er die Bewegungsbahn bestmöglich einhält und keine Kollisionen mit Hindernissen oder Parcoursbegrenzungen auftreten.

Durchfahrbare Engstellen: Durchfahrbare Engstellen im Parcours simulieren Situationen, in denen der Roboter enge Bereiche durchqueren muss, die möglicherweise nur wenig Platz zum Manövrieren bieten. Solche Engstellen können beispielsweise schmale Gänge, Türrahmen oder andere begrenzte Durchfahrtsöffnungen sein. Die Bewältigung dieser Engstellen erfordert eine besonders sorgfältige Steuerung und Anpassung der Bewegung, um Kollisionen zu vermeiden und den Parcours sicher zu durchqueren. Die Herausforderung bei der Navigation durch solche Engstellen liegt darin, die richtige Geschwindigkeit und Ausrichtung beizubehalten, um den Durchgang ohne Berührung der Wände oder Hindernisse zu schaffen. Der Roboter muss in der Lage sein, seine Bewegung präzise zu kontrollieren und gleichzeitig den verfügbaren Platz optimal zu nutzen. Dies erfordert eine effiziente Pfadplanung und eine genaue Trajektorienanpassung, um den Roboter sicher und reibungslos durch die enge Passage zu steuern.

Nicht durchfahrbare Engstellen: Nicht durchfahrbare Engstellen, die darauf abzielen, den Roboter zu verwirren, sind ein anspruchsvolles Element im Parcourslayout. Im Gegensatz zu den durchfahrbaren Engstellen, die dem Roboter eine enge Passage bieten, sind diese Hindernisse darauf ausgelegt, seine Navigation und Entscheidungsfähigkeiten auf die Probe zu stellen. Solch verwirrende Engstellen erfordern von den Navigationsalgorithmen eine hohe Anpassungsfähigkeit und die Fähigkeit, zwischen tatsächlich befahrbaren

Durchgänge und optischen Täuschungen zu unterscheiden. Der globale Pfadplaner muss in der Lage sein, alternative Wege zu finden, die den Roboter aus solchen Situationen herausführen, ohne dass er in Sackgassen gerät oder unerwünschte Manöver ausführt.

Löcher: Die Einbindung von Löchern als Hindernisse ermöglicht es, die Fähigkeit des Roboters zur Erkennung und Einschätzung von Gefahren zu testen. Der Roboter muss in der Lage sein, zwischen durchfahrbaren Flächen und gefährlichen Bereichen zu unterscheiden und entsprechende Entscheidungen zu treffen. Dies erfordert eine präzise Wahrnehmung der Umgebung sowie die Fähigkeit, die richtigen Bewegungsentscheidungen zu treffen, um die Löcher sicher zu umfahren. Die Herausforderung bei der Implementierung von Lochhindernissen liegt darin, den Roboter so zu programmieren, dass er in der Lage ist, die tatsächliche Passierbarkeit der Flächen korrekt einzuschätzen. Dies erfordert eine genaue Erfassung der Geometrie der Löcher sowie eine klare Kommunikation zwischen den Wahrnehmungssensoren des Roboters und den Navigationsalgorithmen.

Unter Berücksichtigung der verschiedenen Hindernisse, die in den vorherigen Abschnitten besprochen wurden, wird nun ein Parcours entworfen, der eine geeignete Herausforderung widerspiegelt. Der Parcours wird eine Kombination aus Kurven, engen Durchfahrten, nicht durchfahrbaren Engstellen und strategisch platzierten Löchern beinhalten. Diese Elemente wurden ausgewählt, um die Navigationsalgorithmen und -strategien auf die Probe zu stellen und die Robustheit des Roboters zu testen. Zusätzlich zur Berücksichtigung der verschiedenen Hindernisse ist bei der Planung des Parcours auch die Größe der Styroporblöcke zu berücksichtigen. Diese Blöcke haben Abmessungen von 1000 x 500 mm. Da der verfügbare Platz auf dem Dachboden begrenzt, muss der Parcours so gestaltet werden, dass er innerhalb dieser Fläche Platz findet.

Basierend auf den verschiedenen Überlegungen und Hindernissen wurde so das Layout wie in Abbildung 48 dargestellt, entworfen. In der Abbildung ist oben ein einzelner Styroporblock mit seinen Abmessungen zu sehen. Darunter ist der gestaltete Parcours abgebildet. Auf den ersten Blick sind verschiedenen Kurven im Parcours zu erkennen, einige sind weiter und andere enger gestaltet, wobei diese nicht speziell hervorgehoben sind. Die anderen Arten von Hindernissen sind jedoch entsprechend markiert. Die grün markierten Bereiche repräsentieren durchfahrbare Engstellen, die der Roboter passieren kann. In rot ist eine Engstelle dargestellt, die für den Roboter nicht passierbar ist. Als letztes Hindernis ist in blau ein Loch markiert.

Wie ersichtlich, wurde ein Layout entwickelt, dass eine Vielfalt von Hindernistypen beinhaltet und gleichzeitig sicherstellt, dass die Gesamtgröße des Parcours im vorgesehenen Rahmen bleibt.

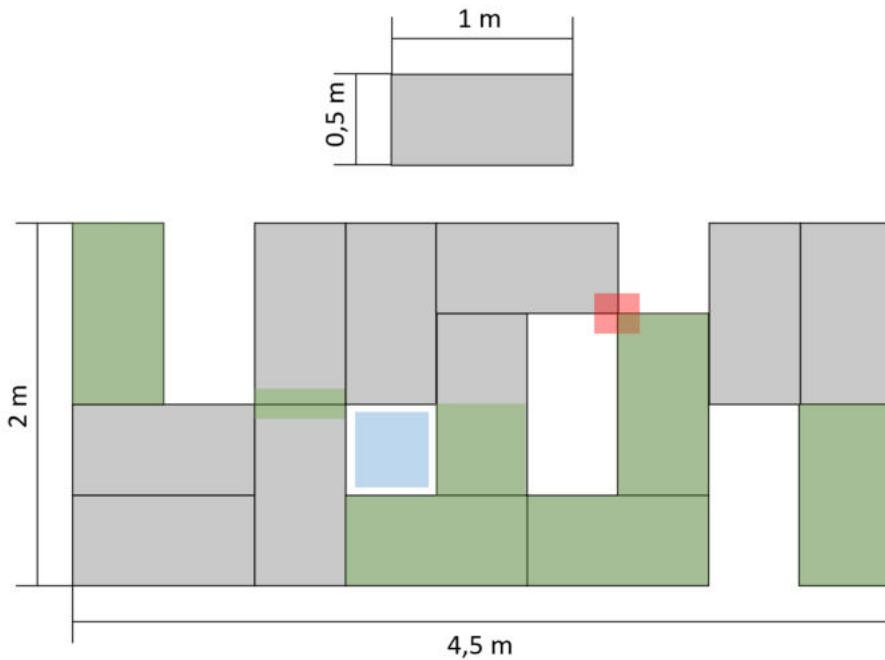


Abbildung 48: Layout des Dachbodenparcours

4.2 Tests und Ergebnisse

Nach der erfolgreichen Planung des Parcours folgt die Übertragung der Tests aus der Simulation in die reale Welt. Dieses Kapitel widmet sich der praktischen Erprobung der Navigationsalgorithmen und -strategien auf dem realen Kobuki-Roboter in Verbindung mit der Astra 3D-Kamera. Der Übergang von der Simulation zur Realität birgt eine Vielzahl von Herausforderungen und Variablen, die berücksichtigt werden müssen.

4.2.1 Test Kobuki

Im Rahmen eines ersten Tests steht die Erprobung der Materialien für den Parcours und die Evaluation des Kobuki-Roboters samt der 3D-Kamera an. In diesem Test werden zwei Hauptaspekte überprüft. Erstens wird getestet, ob der Kobuki manuell über den Parcours gesteuert werden kann. Dies dient dazu, die Grundsteuerung und die physische Bewegungsfähigkeit des Roboters zu überprüfen. Zweitens wird die Funktionalität der 3D-Kamera geprüft, indem überprüft wird, ob sie eine geeignete Punktwolke erzeugt.

Vorbereitung

Anstatt den gesamten Parcours auf einmal aufzubauen, wird zunächst nur ein kleiner Teil davon errichtet, zu sehen in Abbildung 49. Dieser Ansatz verhindert, dass unnötig viel Arbeit investiert wird, falls es aufgrund der Materialien oder anderer Probleme zu Schwierigkeiten kommt. Zudem reduziert dies den Aufwand für den Umbau, falls Änderungen oder Anpassungen erforderlich sind.

Vonseiten der Software verfügen sowohl der Roboter als auch die 3D-Kamera über vorkonfigurierte ROS-Pakete. Das bedeutet, dass zum Starten der entsprechenden Knoten lediglich die passenden Launch-Files ausgewählt werden müssen. Allerdings werden Informationen zur Koordinatentransformation zwischen dem Roboter und der Kamera benötigt. Diese Informationen werden mithilfe des `static_transform_publisher`-Knotens übermittelt. Dieser Knoten ist Teil des `tf`-Pakets, das bereits in der Standardinstallation von ROS enthalten ist. Für die Konfiguration müssen lediglich beim Starten des Knotens zwei bereits vorhandene Koordinatensysteme angegeben werden sowie deren gegenseitige Position zueinander. In diesem Fall werden die Koordinatensysteme `/base_footprint` des Roboters und `/camera_link` der Kamera ausgewählt und zugeordnet. [32]

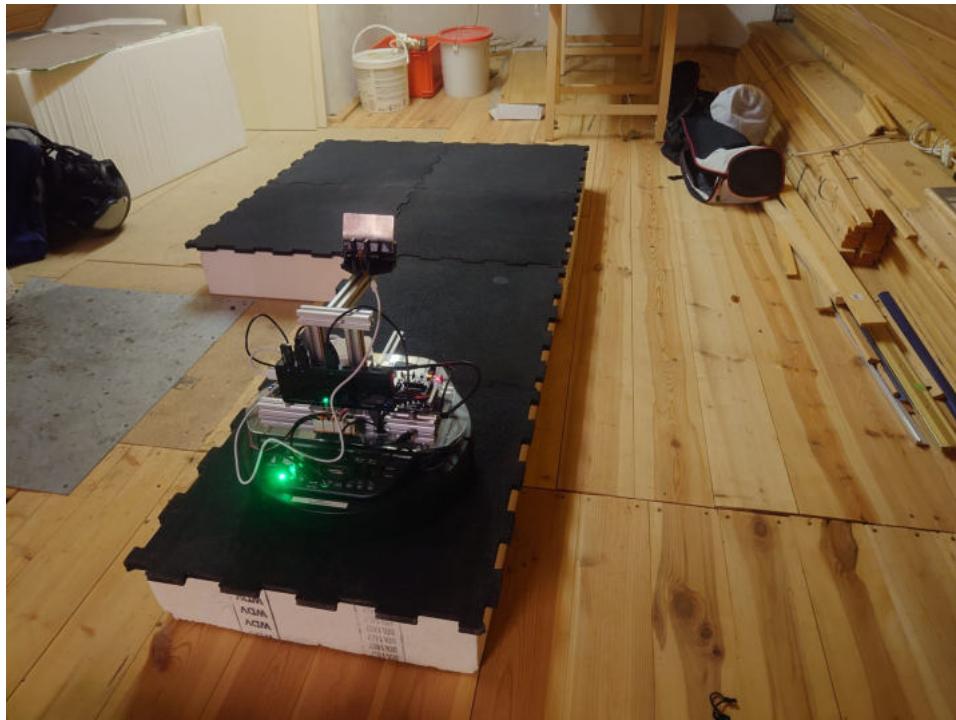


Abbildung 49: Teilaufbau des Testparcours

Systemaufbau

Im anschließenden Abschnitt wird der vereinfachte Systemaufbau der Knoten beschrieben, die auf dem ODROID-Rechner ausgeführt werden, welcher der Hauptrechner des Kobuki-Roboters ist. Auf diesem Rechner sind die Knoten für den Kobuki-Roboter, die 3D-Kamera und die `tf`-Transformation aktiv. Die Interaktion zwischen diesen Komponenten wird in Abbildung 50 veranschaulicht.

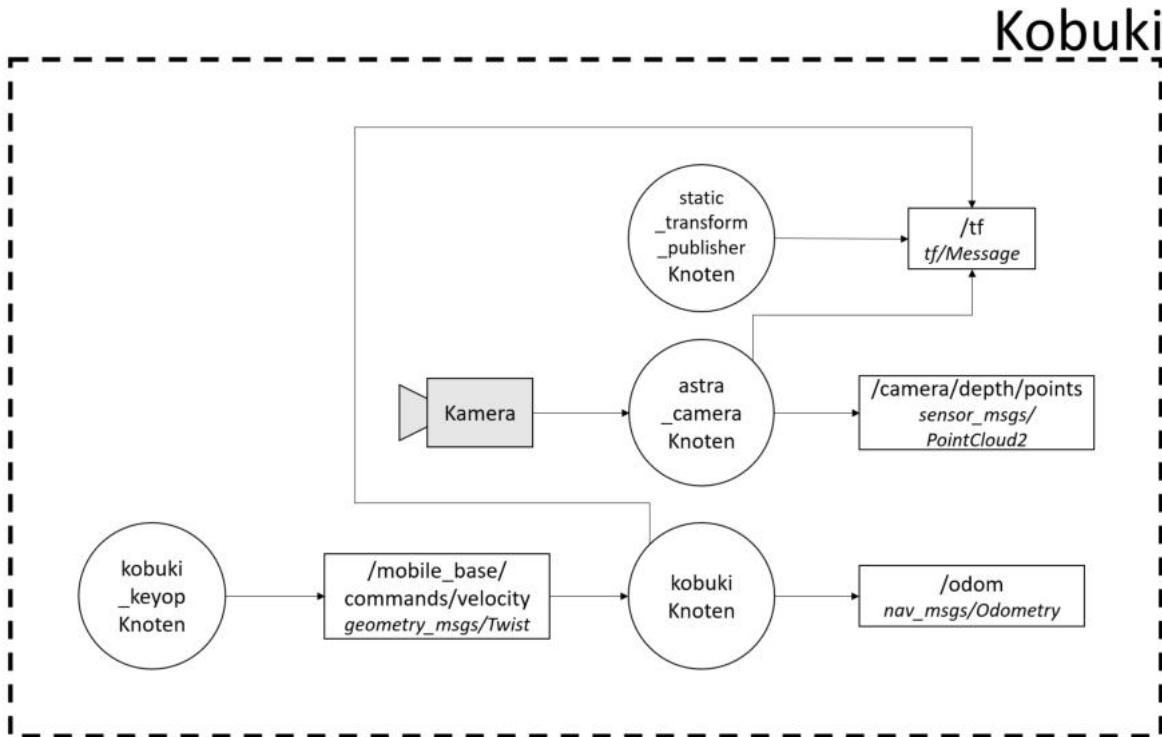


Abbildung 50: Vereinfachter Systemaufbau Kobuki

In den ersten Tests wird der Kobuki-Roboter über den `kobuki_keyop`-Knoten gesteuert, der die Eingaben von Pfeiltasten in Geschwindigkeitssignale umwandelt. Diese Signale werden dann über das Topic `/mobile_base/commands/velocity` an den Kobuki-Knoten gesendet. Der Kobuki-Knoten verarbeitet diese Geschwindigkeitssignale und steuert die Motorcontroller des Roboters entsprechend an. Gleichzeitig gibt der Kobuki-Knoten Transformations- und Positionsinformationen über die Topics `/tf` und `/odom` aus. Um die Beziehung zwischen dem Koordinatensystem der Kamera und dem des Kobuki-Roboters zu definieren, wird der `static_transform_publisher`-Knoten verwendet. Der `astracamera`-Knoten erfasst die Daten der 3D-Kamera und stellt sie als Punktwolke über das Topic `/camera/depth/points` zur Verfügung. Gleichzeitig sendet auch der Kamera-Knoten Informationen über seine eigenen Koordinatensysteme.

Test

Für den Test werden die erforderlichen Knoten über eine SSH-Verbindung („Secure Shell“-Verbindung) gestartet, wobei die entsprechenden Launch-Files verwendet werden. Das Hauptziel dieses Tests besteht darin, den Roboter über das Terminal zu steuern und gleichzeitig die erzeugte Punktwolke zu überprüfen.

Die Steuerung des Roboters über das Terminal verläuft problemlos und die Verzögerung zwischen der Eingabe der Bewegungsbefehle und ihrer Ausführung ist kaum wahrnehmbar. Bei Betrachtung der Situation in RViz, wie in Abbildung 51 dargestellt, zeigt sich, dass die Transformation zwischen den verschiedenen Koordinatensystemen erfolgreich ist. Jedoch fällt auf, dass der aufgebaute Teil des Parcours in der Punktwolke nicht komplett erfasst wird. Da der Boden rechts neben dem Parcours sichtbar ist, lässt sich vermuten, dass die verwendeten Gummimatten das Problem verursachen. Diese Vermutung wird durch Abbildung 52 bestätigt. Hier wird ein Blatt Papier auf den Matten platziert, welches deutlich erkennbar ist.

Aufgrund dieser Erkenntnis werden die Matten mit Geschenkpapier überzogen. Das Ergebnis der Punktwolke nach dieser Anpassung ist in Abbildung 53 zu sehen. Dieses Resultat ist zufriedenstellend, was darauf hinweist, dass der gesamte Parcours mit Papier überzogen werden sollte, um eine korrekte Erfassung zu gewährleisten. Zusätzlich ist die Auflösung der Punktwolke noch zu hoch, was eine höhere Rechenleistung erfordert. Für kommende Tests ist es notwendig, diese Auflösung zu verringern, um die Leistungseffizienz zu verbessern.

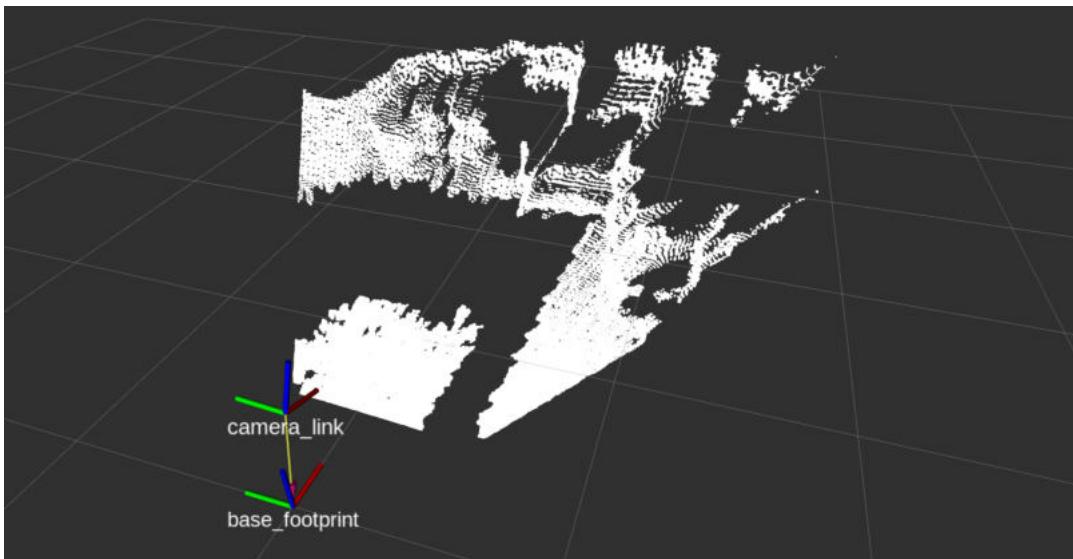


Abbildung 51: Aufgenommene Punktwolke des Parcours

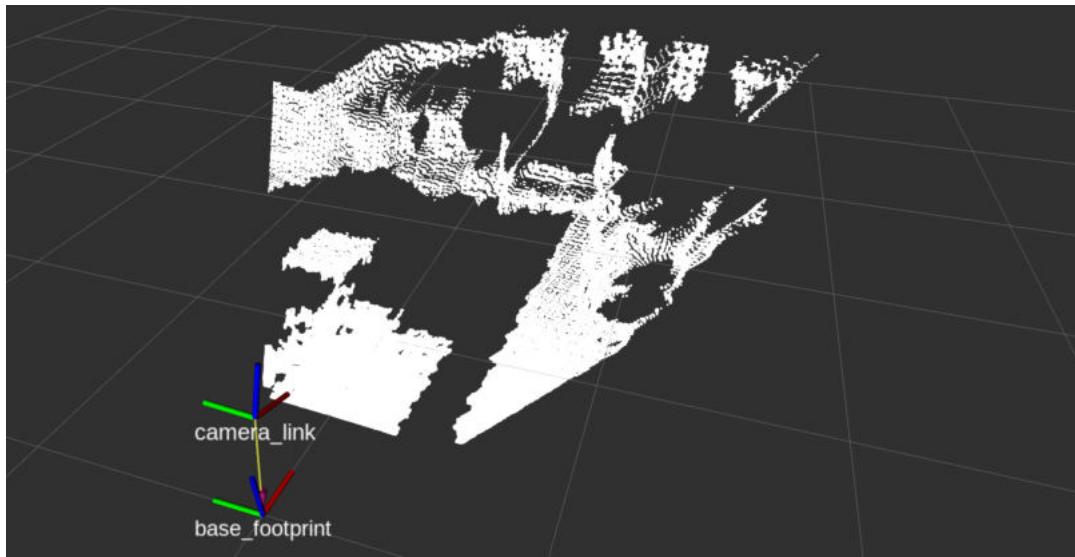


Abbildung 52: Aufgenommene Punktfolke des Parcours inklusive eines Blatt Papiers

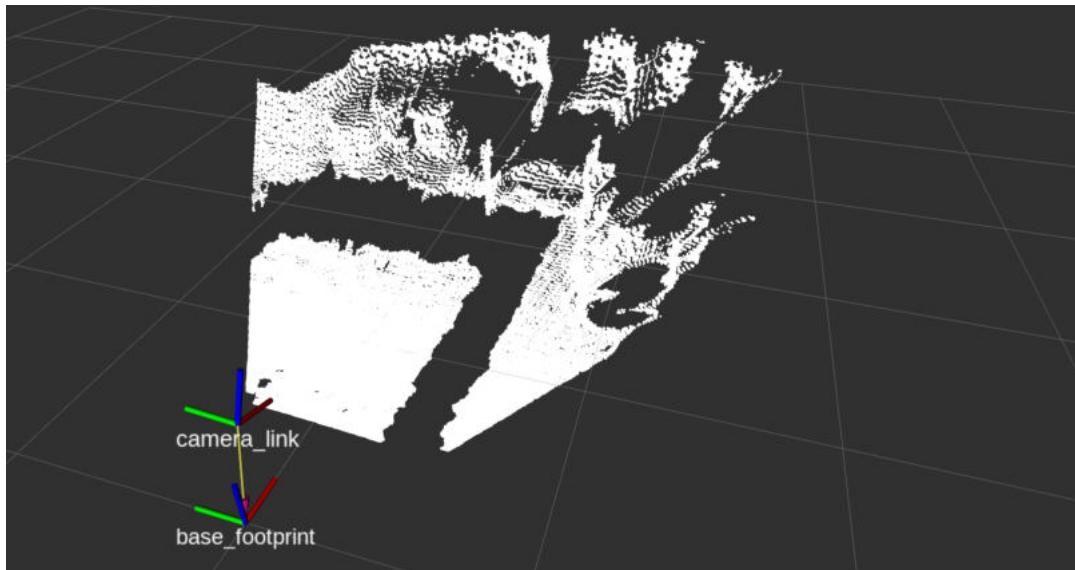


Abbildung 53: Aufgenommene Punktfolke des mit Papier überzogenen Parcours

4.2.2 Test Mapping

Im Rahmen des zweiten Tests steht die Erprobung des Mappings inklusive des `manipulate_map`-Knotens in der realen Welt an. Ziel dieses Tests ist es, die Fähigkeit des Roboters zu überprüfen, eine Karte der Umgebung zu erstellen und gleichzeitig Manipulationen daran vorzunehmen.

Vorbereitung

Vor Beginn dieses Tests wird der Parcours entsprechend dem zuvor erstellten Plan des vorherigen Kapitels aufgebaut. Ein Vergleich zwischen dem Plan und der realen Umgebung ist in Abbildung 54 dargestellt. Auf dieser Abbildung ist auch erkennbar, dass die Matten, wie zuvor beschrieben, mit Papier überzogen wurden. Ein weiterer Aspekt, der noch berücksichtigt werden muss, ist die Notwendigkeit, die Auflösung der Punktfolge zu reduzieren. Hierfür wird ein Filter-Knoten namens `voxel_grid` aus dem `pcl_ros`-Paket verwendet. Die Konfiguration dieses Filters erfolgt über ein Launch-File, in dem die entsprechenden ROS-Parameter festgelegt werden. Zusätzlich zur Parameterkonfiguration müssen die Input- und Output-Topics des Filters über einen remap der Topics angegeben werden. [33]

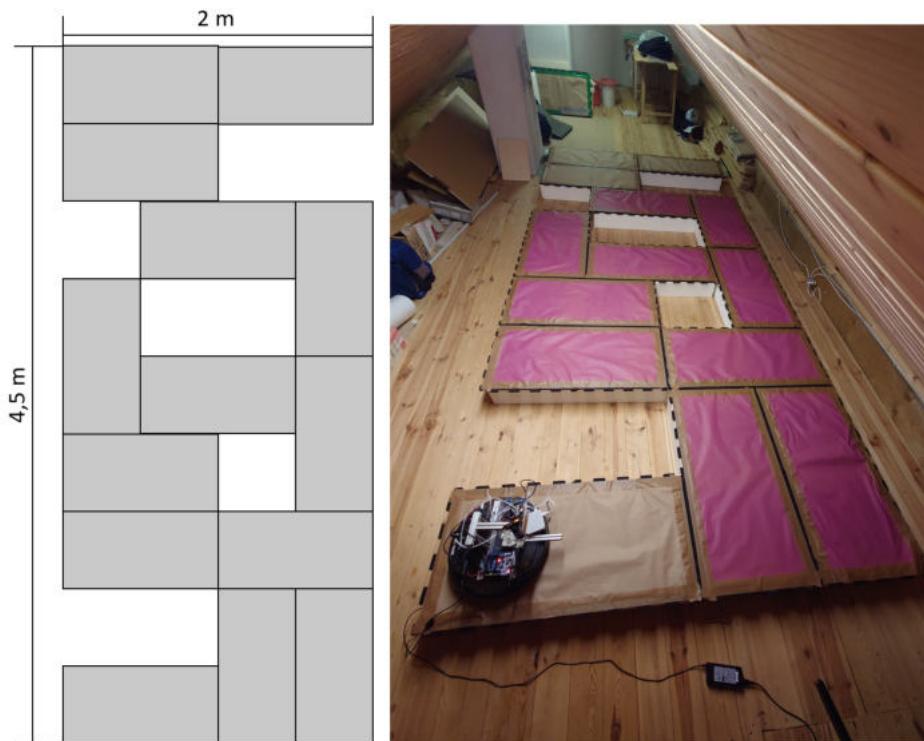


Abbildung 54: Gegenüberstellung des Parcoursplans und dem aufgebauten Parcours

Systemaufbau

Der vereinfachte Systemaufbau ist in den Abbildungen 55 und 56 dargestellt. Dieser ähnelt in logischer Hinsicht dem Systemaufbau des Mapping-Tests in der Simulation. Dennoch ergeben sich einige Änderungen, da der reale Kobuki-Roboter anstelle des simulierten Roboters verwendet wird. In Abbildung 55 sind die Knoten aufgeführt, die auf dem Kobuki-Roboter laufen. Diese Knoten wurden bereits im vorherigen Test beschrieben. Die wesentliche Änderung betrifft die Notwendigkeit eines zusätzlichen Filters am Entwicklungsrechner, der die Auflösung der Punktwolke reduziert. Dieser Filter empfängt die Punktwolke über das Topic `/camera/depth/points`, verarbeitet die enthaltenen Sensordaten und gibt die gefilterte Punktwolke über das Topic `/camera/depth/points_downsampled` an den Mapping-Knoten weiter.

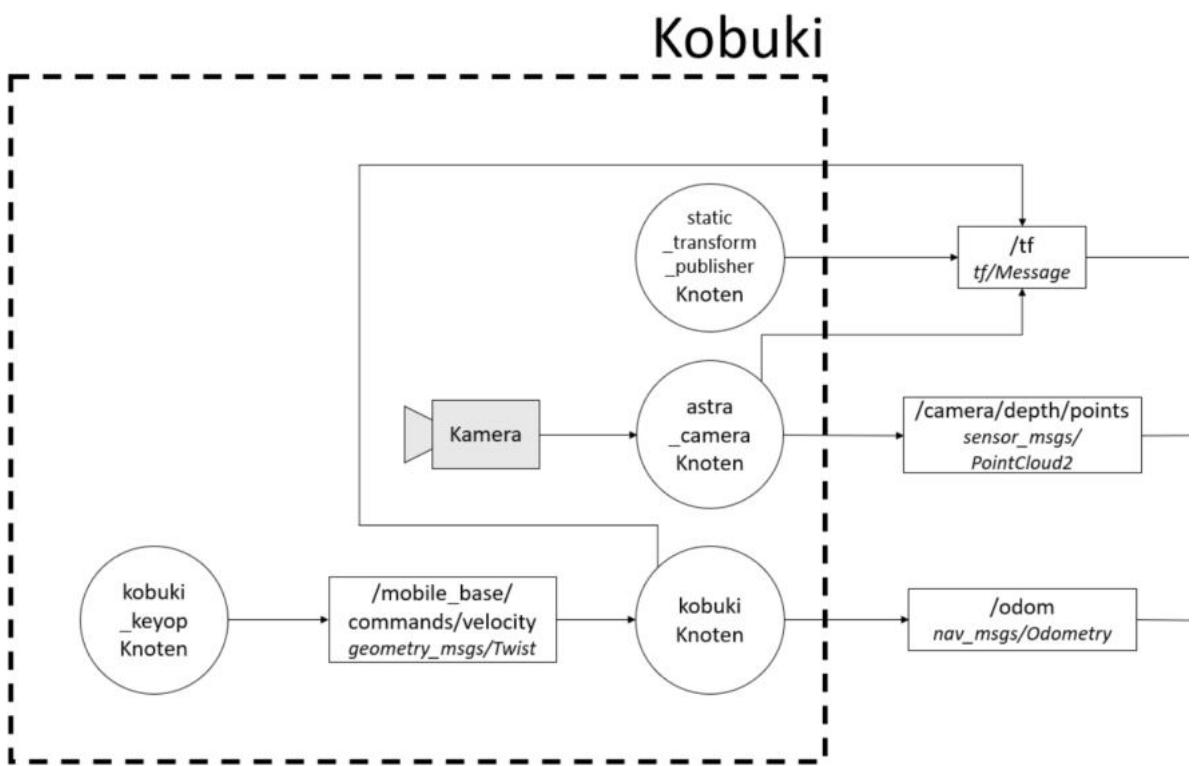


Abbildung 55: Vereinfachter Systemaufbau des realen Mapping-Tests Kobuki

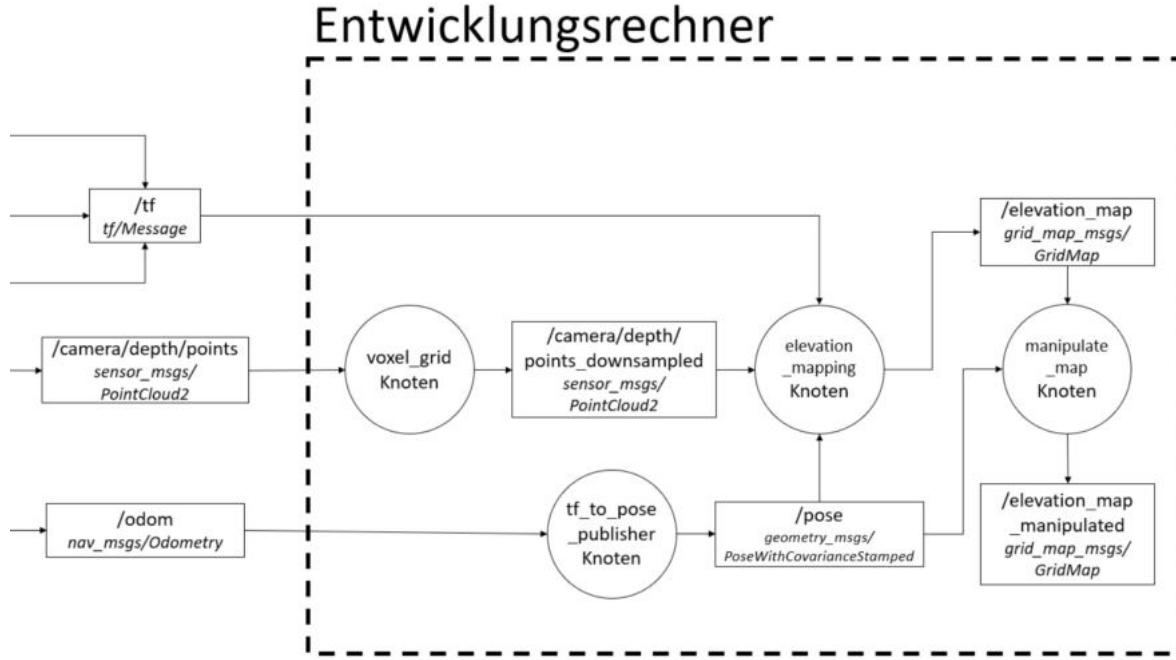


Abbildung 56: Vereinfachter Systemaufbau des realen Mapping-Tests Entwicklungsrechner

Test

Mit dem entsprechend vorbereiteten Systemaufbau wird das Mapping in der realen Umgebung getestet. Ziel dieses Tests ist es, den Roboter über das Terminal durch den vorbereiteten Parcours zu steuern und währenddessen eine präzise Karte der Umgebung zu erstellen.

Bei Systemstart wird sofort deutlich, dass am Rand des Parcours das Schrägdach des Dachbodens mit aufgezeichnet wird, was die Kartenqualität beeinträchtigt. Um diesem Problem entgegenzuwirken, wird vor der Fortsetzung des Tests eine Anpassung am `manipulate_map`-Knoten vorgenommen. Während der Fusion der Karten wird in der `elevation`-Ebene geprüft, ob der Höhenwert der aktuellen Zelle größer ist als die Höhe des Parcours. Ist dies der Fall, wird diese Zelle auf die Höhe des Bodens gesetzt. In Abbildung 57 ist ein Vergleich zwischen einem Screenshot vor und nach dieser Korrektur zu sehen. Die obere Abbildung zeigt das Ergebnis ohne die Korrektur, während die untere Abbildung das Ergebnis nach der Korrektur präsentiert. Es wird deutlich, dass die vorgenommene Korrektur erfolgreich ist und das Ergebnis verbessert.

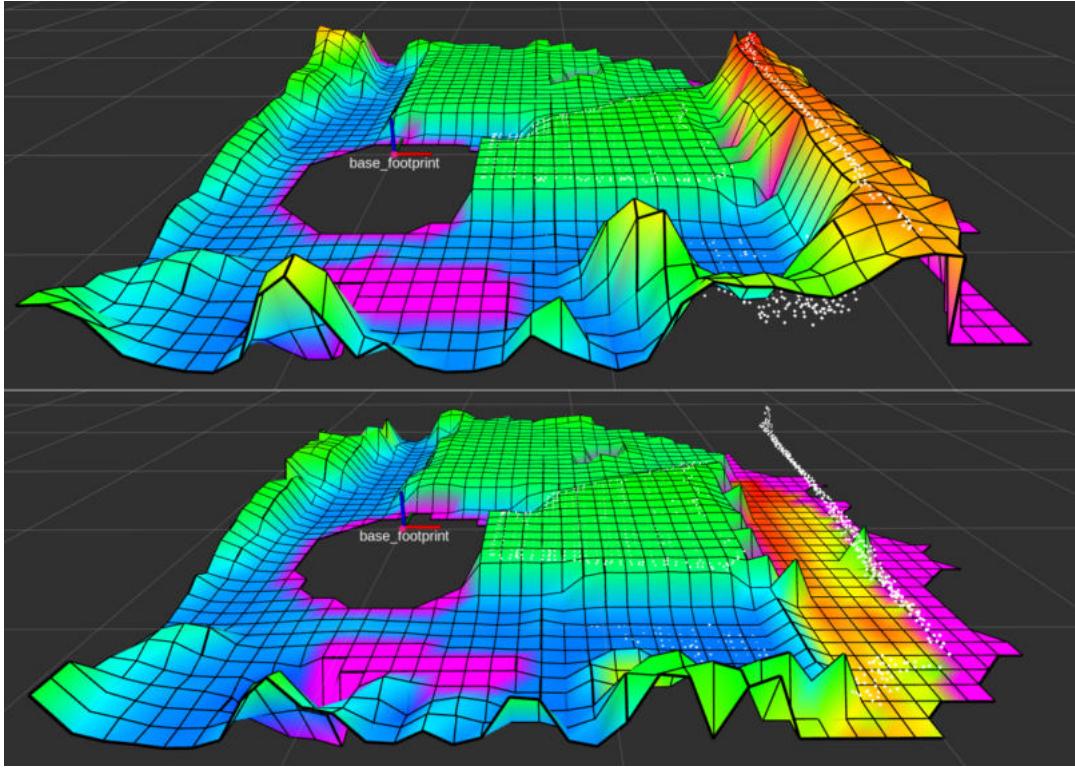


Abbildung 57: Gegenüberstellung des Mappings ohne (oben) und mit (unten) Korrektur des Schrägdachs

Nachdem der `manipulate_map`-Knoten ausgebessert wurde, wird mit dem Test fortgefah- ren. Durch das vollständige Durchfahren des gesamten Parcours und der gleichzeitigen Kartenerstellung ergibt sich eine Höhenkarte, wie sie in Abbildung 58 links dargestellt ist. Auf der rechten Seite der Abbildung ist erneut das Bild des Parcours zu sehen. Ein Vergleich der beiden Bilder verdeutlicht, dass die Karte zu großen Teilen ordnungsgemäß erstellt wurde. Trotz einiger Störeinflüsse, die größtenteils auf die Beschaffenheit des Dachbodens zurückzuführen sind, ist das Ergebnis zufriedenstellend.

Während des Tests fiel ein weiterer bemerkenswerter Aspekt auf, der die Darstellung von Löchern in der Karte betrifft. Es wurde festgestellt, dass diese Löcher etwas kleiner erscheinen, als sie tatsächlich sind. Dieses Phänomen hat zwei Hauptursachen. Zum einen liegt es an der Funktionsweise des Algorithmus des `elevation_mapping`-Knotens. Bereits in der Karte, die von diesem Knoten ausgegeben wird, sind die Löcher kleiner als erwartet. Das grundlegende Problem besteht daher im Algorithmus dieses Knotens, was die Bewäl- tigung dieses Phänomens äußerst herausfordernd macht, da dieses Paket sehr komplex und umfangreich ist. Die zweite Ursache betrifft die Funktionsweise des `manipulate_map`-

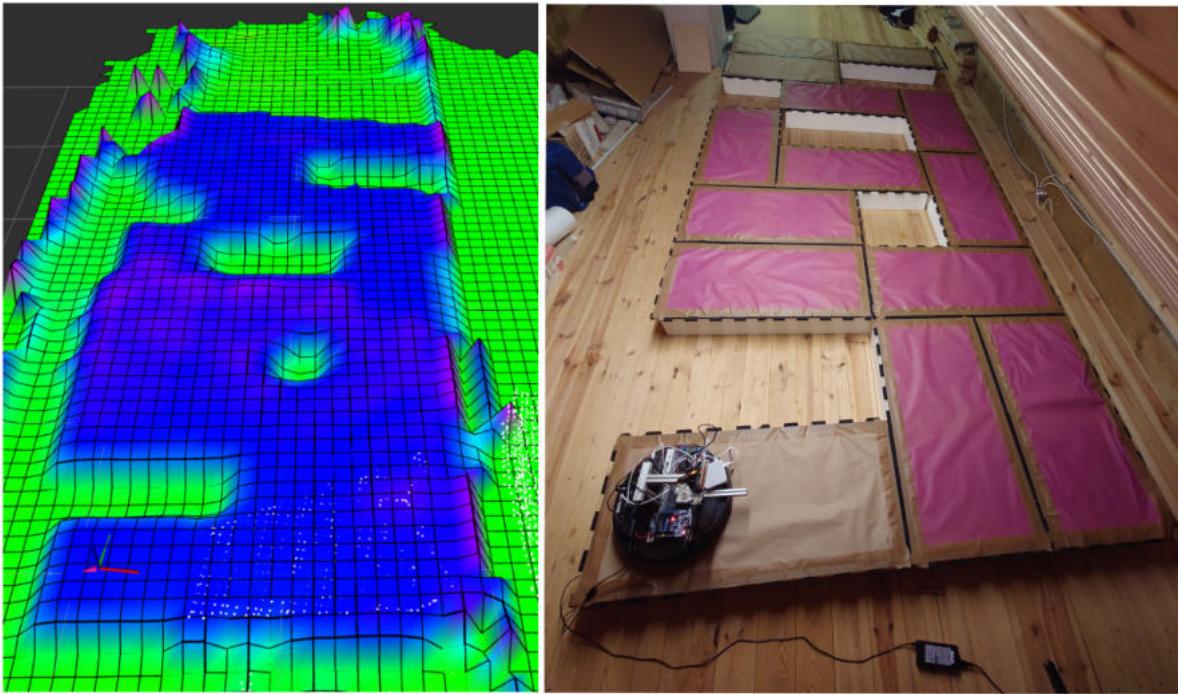


Abbildung 58: Gegenüberstellung der erzeugten Karte (links) und dem realen Parcours (rechts)

Knotens. Einmal erhaltene Werte durch das Mapping des `elevation_mapping`-Knotens werden durch nachfolgende Manipulationen nicht mehr überschrieben.

Trotz dieses Problems wird vorerst mit der aktuellen Konfiguration und dem bestehenden Systemaufbau fortgefahrene, in der Hoffnung, dass sich dieses Problem nicht übermäßig auf die bevorstehenden Tests und die Exploration auswirken wird.

4.2.3 Test autonome Navigation

Als nächster Schritt in der Testreihe steht die Überprüfung der `move_base`-Navigation in der realen Umgebung an. Nachdem die grundlegende Kartenerstellung erfolgreich getestet wurde, wird sich nun auf die Fähigkeit des Roboters konzentriert, mithilfe der `move_base`-Navigation autonom durch den Parcours zu navigieren.

Vorbereitung

Vor der Durchführung des `move_base`-Tests sind einige Anpassungen der Parameter notwendig, um den `move_base`-Knoten an die reale Umgebung und die Eigenschaften des Kobuki-Roboters anzupassen. Zuallererst wird die `costmap_common_params`-

Datei betrachtet. Hier wird der Parameter für den Fußabdruck des Roboters angepasst, um die tatsächliche Geometrie des Kobuki-Roboters korrekt abzubilden. In der `local_costmap_params`-Datei ist es erforderlich, den Parameter für das Roboterkoordinatensystem anzupassen, da das Kobuki-System einen anderen Koordinatensystemnamen verwendet als in der Simulation. Die restlichen Parameter können in dieser Datei unverändert bleiben. Ähnlich verhält es sich mit der `global_costmap_params`-Datei. Hier wird ebenfalls der Parameter für das Koordinatensystem des Roboters angepasst. Die Konfigurationsdateien, die sich mit der Pfadplanung befassen, erfordern keine Änderungen, da sie die gleichen Parameter wie in der Simulation verwenden können. Zum Abschluss der Vorbereitung wird noch ein sogenannter remap durchgeführt, um das Ausgangstopic des `move_base`-Knotens anzupassen. Dieses Topic, das die Bewegungssignale überträgt, wird von seinem Standard-Topic `cmd_vel` auf das Topic `mobile_base/commands/velocity` umgemappt, welches vom Kobuki-Roboter erwartet wird.

Systemaufbau

Abbildung 59 und Abbildung 60 illustrieren den vereinfachten Systemaufbau für den bevorstehenden Test, welcher auf den vorangegangenen Tests aufbaut. Diese gesamte Abbildung findet sich in Originalgröße im Anhang.

Die erweiterten Knoten in dieser Konfiguration sind von besonderem Interesse. Unter diesen Knoten befindet sich der `grid_map_visualization`-Knoten, der dazu dient, die zuvor erstellte Höhenkarte in eine Costmap zu transformieren. Ebenso ist der `move_base`-Knoten eingeführt worden, der für die robotergesteuerte Navigation verantwortlich ist. Der RViz-Knoten vervollständigt dieses Ensemble, indem er ermöglicht, dem `move_base`-Knoten Zielpunkte zuzuweisen.

Test

Der Hauptzweck dieses Tests liegt darin, die Navigationsfähigkeiten des Roboters innerhalb des Parcours zu validieren. Dies wird erreicht, indem Zielpunkte mithilfe von RViz festgelegt werden und der `move_base`-Knoten für die Navigation zum jeweiligen Zielpunkt genutzt wird. Wie zuvor wird auch hier der Roboter vor Beginn des Tests um 360 Grad rotiert.

Während der Testdurchführung wird rasch deutlich, dass der lokale Pfadplaner aufgrund seiner begrenzten Komplexität an seine Leistungsgrenzen stößt. Dieses Szenario wird durch das erneute Auftreten des Problems verschärft, bei dem Kanten aufgrund der Kartenauflösung und der Charakteristika des `elevation_mapping`-Algorithmus nicht exakt genug dargestellt werden. Das Fehlen eines Parameters, der den Randabstand kontrolliert, verschärft die Problematik des verwendeten `local_planner` zusätzlich.

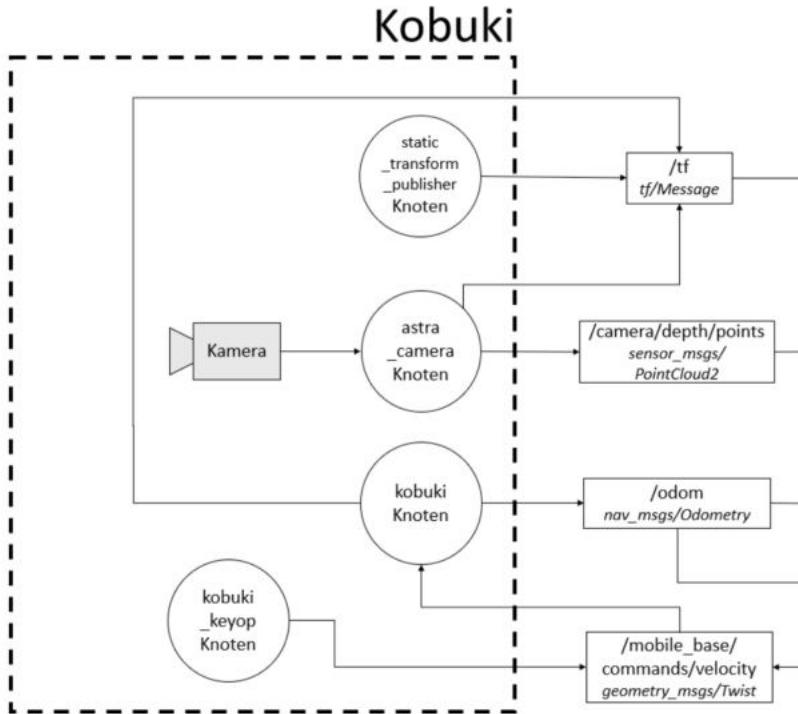


Abbildung 59: Vereinfachter Systemaufbau des move_base-Tests in realer Umgebung Kobuki

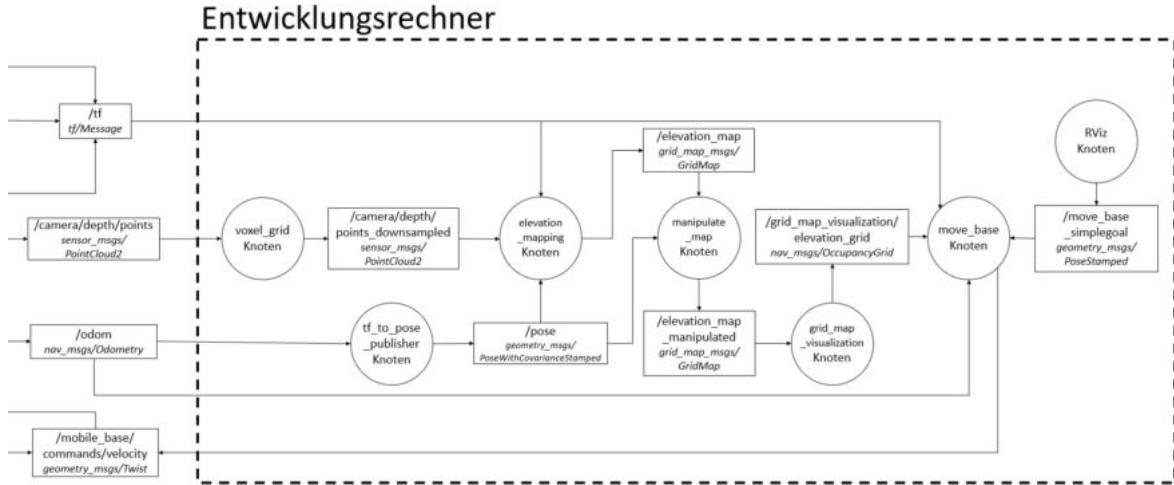


Abbildung 60: Vereinfachter Systemaufbau des move_base-Tests in realer Umgebung Entwicklungsrechner

Angesichts dieser Herausforderungen ist der Wechsel zu einem alternativen `local_planner` unausweichlich, um die Genauigkeit und Zuverlässigkeit der Pfadberechnung zu gewährleisten. Dementsprechend wird der laufende Test vorerst beendet, bis eine geeignete Alternative für den `local_planner` ausgewählt wurde.

Infolge einer Analyse der verfügbaren `local_planner` wird der `teb_local_planner` als präferierte Wahl eruiert. Dieser Entschluss fußt auf seiner bemerkenswerten Konfigurationsflexibilität, die durch eine breite Palette von Parametern ermöglicht wird. Besonders in Umgebungen mit variablen oder unvorhersehbaren Hindernissen manifestiert sich der `teb_local_planner` als adäquate Option. [34]

`teb_local_planner`

Im Kontext der Verwendung des `teb_local_planner` für die bevorstehenden Tests wird ein YAML-File für die Konfiguration benötigt. Die wichtigsten Parameter für die Konfiguration sind in Abbildung 61 zu sehen.

```
TebLocalPlannerROS:
# Robot Configuration Parameters
  acc_lim_x: 0.3
  acc_lim_theta: 0.4
  max_vel_x: 0.05
  max_vel_x_backwards: 0.1
  max_vel_theta: 0.4

  footprint_model:
    type: "circular"
    radius: 0.22

  min_obstacle_dist: 0.1
  enable_homotopy_class_planning: false      # deactivate parallel planning
  weight_kinematics_forward_drive: 1000       # prevent backward driving
```

Abbildung 61: Ausschnitt der Konfigurationsdatei für `teb_local_planner`

In dieser Datei werden die essenziellen Parameter definiert, die im Folgenden erläutert werden. Zu Beginn werden Geschwindigkeits- und Beschleunigungsbegrenzungen eingestellt. Diese Festlegungen dienen dazu, im Falle eines unvorhergesehenen Ereignisses rechtzeitig eingreifen zu können. Weiterhin ist es erforderlich, den Fußabdruck des Roboters erneut anzugeben, da dieser Parameter vom Planer benötigt wird, um korrekt zu operieren. Ein bedeutender Parameter ist der `min_obstacle_distance`-Parameter. Dieser dient dazu,

einen angemessenen Mindestabstand zu den Hindernissen einzuhalten, wenn dies möglich ist. In diesem Kontext wird dieser Parameter auf 100 mm festgesetzt. Diese Distanz soll sicherstellen, dass der Roboter nicht über den Rand des Pfades hinausfährt, wie im vorherigen Test beobachtet. Darüber hinaus werden weitere Parameter so eingestellt, dass der Rechenaufwand des Planers begrenzt wird. Zum Beispiel wird lediglich eine Pfadberechnung durchgeführt, anstatt mehrere Optionen zu berechnen. Zuletzt wird ein Parameter festgelegt, der sicherstellt, dass der Roboter stets versucht, vorwärts zu fahren. Dies hat den Effekt, dass der Roboter nicht entgegengesetzt der Blickrichtung der Kamera fährt, was die Orientierung und Navigation erleichtert.

Test inklusive `teb_local_planner`

Nach der Konfiguration des neuen `tеб_local_planner` kann der Test fortgesetzt werden, wo er zuvor unterbrochen wurde. Der Roboter wird in die Ausgangsposition gebracht, und anschließend wird ein Wegpunkt in unbekanntem Gelände festgelegt. In Abbildung 62 sind einige Screenshots dieses Tests zu sehen. Im ersten Bild wird die Ausgangsposition des Roboters dargestellt. Der rote Pfeil im oberen Bereich markiert den Zielpunkt samt Ausrichtung. Der grüne Pfad repräsentiert den Pfad des `global_planner`, der deutlich durch unbekanntes Terrain verläuft. Der orangefarbene Pfad stellt den Pfad des `tеб_local_planner` dar. Auffällig ist, dass der lokale Pfad in der Kurve wie beabsichtigt einen deutlich größeren Bogen schlägt als der `global_planner`. Im nächsten Bild ist der Roboter auf dem Weg entlang der ersten Kurve zu sehen. In den darauffolgenden Bildern ist zu erkennen, wie sich der Roboter aufgrund des anfänglich berechneten globalen Pfades in einer engen Passage festfährt. Nachdem der Planer jedoch erkennt, dass die zu enge Lücke nicht durchfahrbar ist, wird ein neuer globaler Pfad berechnet. Im letzten Screenshot ist ersichtlich, wie der Roboter dem neuen globalen Pfad folgt, nachdem dieser neu berechnet wurde. Dies verdeutlicht die Anpassungsfähigkeit des `tеб_local_planner` an die Umgebung.

Der durchgeführte Test bestätigt den Erfolg des neuen Planers. Daraus lässt sich folgern, dass der Test des `move_base`-Knotens in der realen Umgebung als erfolgreich betrachtet werden kann.

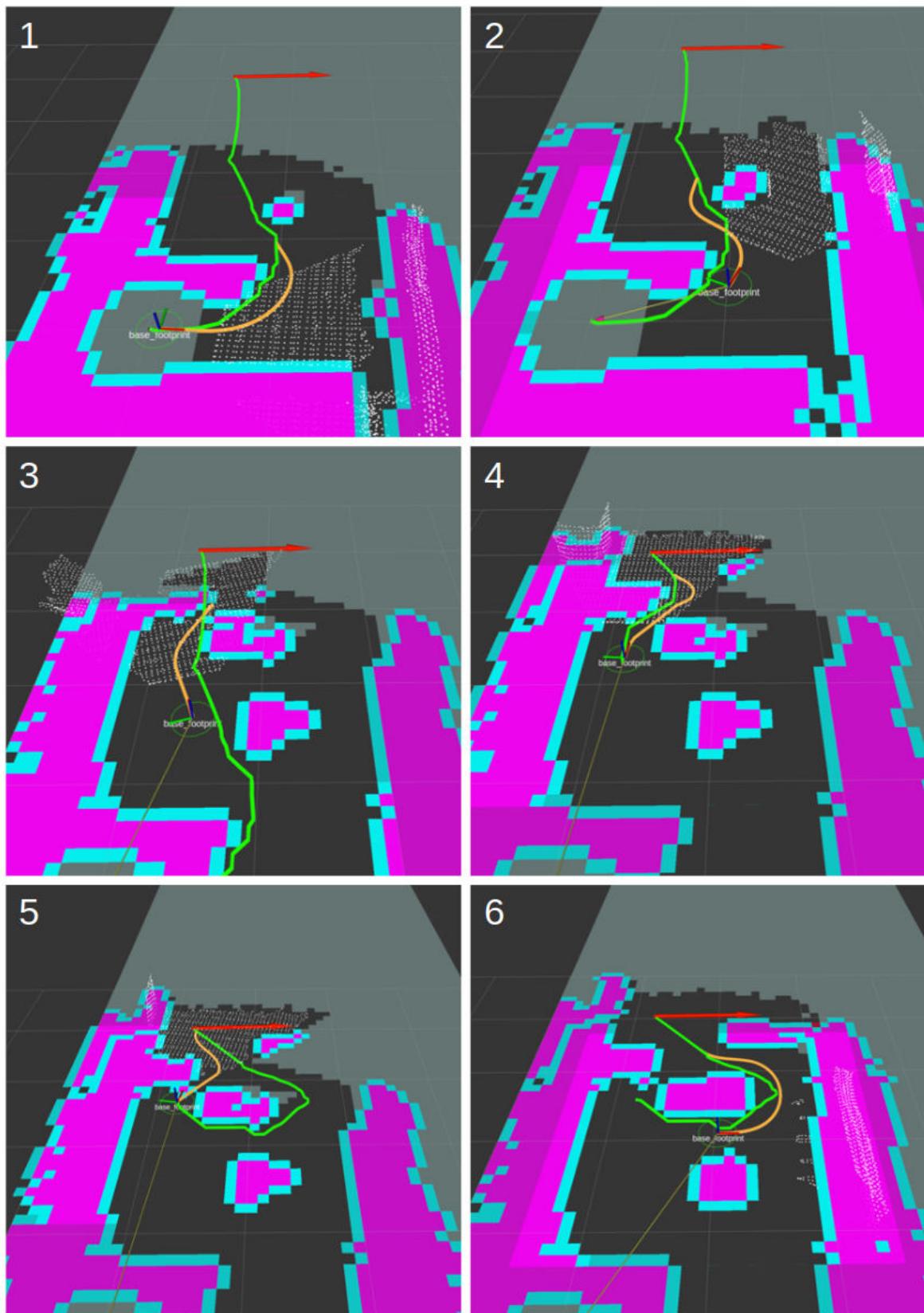


Abbildung 62: Ablauf des Tests mit dem `teb_local_planner`

4.2.4 Test autonome Exploration

Im letzten Testabschnitt wird der Explorationsknoten in der realen Umgebung getestet. Das Hauptziel dieses Tests besteht darin, die Fähigkeit des Roboters zu prüfen, unbekanntes Terrain autonom zu erkunden.

Vorbereitung

Die Vorbereitung für diesen Test entfällt, da der Knoten identisch zur Simulation gestartet werden kann. Dies ermöglicht einen reibungslosen Übergang von der Simulation zur realen Welt, ohne zusätzliche Anpassungen oder Konfigurationen vornehmen zu müssen.

Systemaufbau

Betrachtet man den Systemaufbau des Tests, zu sehen in Abbildung 63, so ist dieser fast identisch zum vorherigen Test. Wie in der Simulation ist der einzige Unterschied, dass die Zielpunkte nicht über RViz vorgegeben werden, sondern über den `explore_lite`-Knoten.

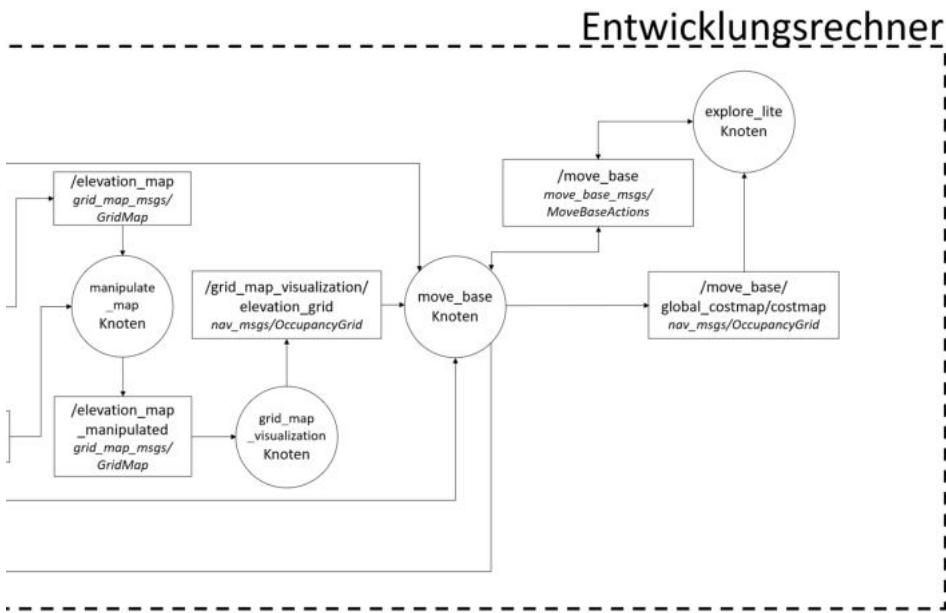


Abbildung 63: Vereinfachter Systemaufbau des Explorationstests in realer Umgebung

Test

Das Hauptziel des letzten Tests besteht darin, den Explorationsknoten in der realen Umgebung zu überprüfen. Die Vorgehensweise folgt dabei dem Muster des Simulationstests. Abbildung 65 zeigt eine Abfolge von Screenshots, die den Versuchsablauf darstellen.

Wie üblich zeigt das erste Bild den Roboter zu Beginn des Tests. In diesem Bild sind auch zwei mögliche Zielpunkte als grüne Kugeln markiert. Im Verlauf der aufeinanderfolgenden Bilder kann man verfolgen, wie die Exploration über den gesamten Parcours hinweg stattfindet.

In Abbildung 64 wird die Karte präsentiert, die nach Abschluss der Exploration entstanden ist. Wie erwartet weicht das Kartenresultat in der Genauigkeit von der Simulation ab. Dennoch lässt sich beobachten, dass der Roboter den kompletten Parcours autonom abgefahren ist und eine akzeptable Karte generiert hat, wenn auch mit leichten Verzerrungen. Dieser letzte Test bestätigt den Erfolg des Explorationsknotens auch in der realen Umgebung.

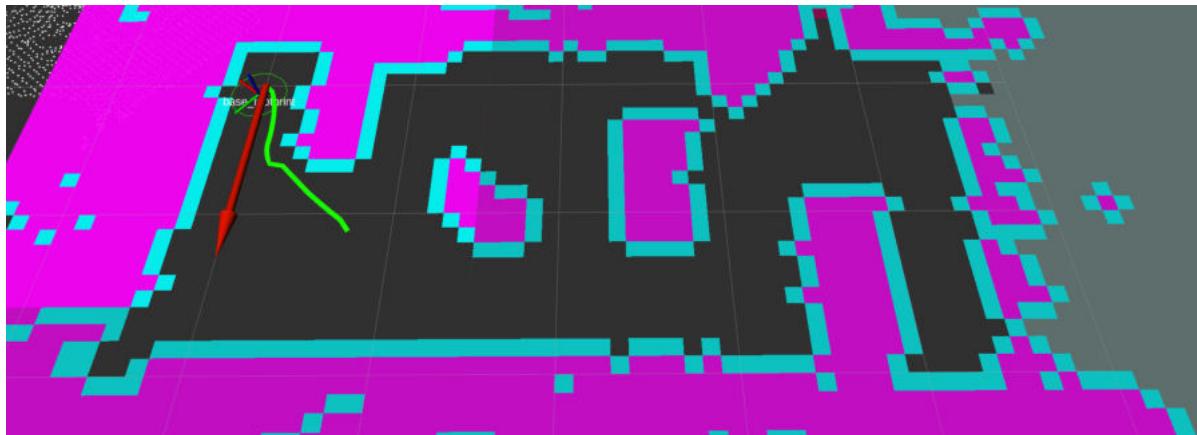


Abbildung 64: Aufgezeichnete Karte des Explorationstests

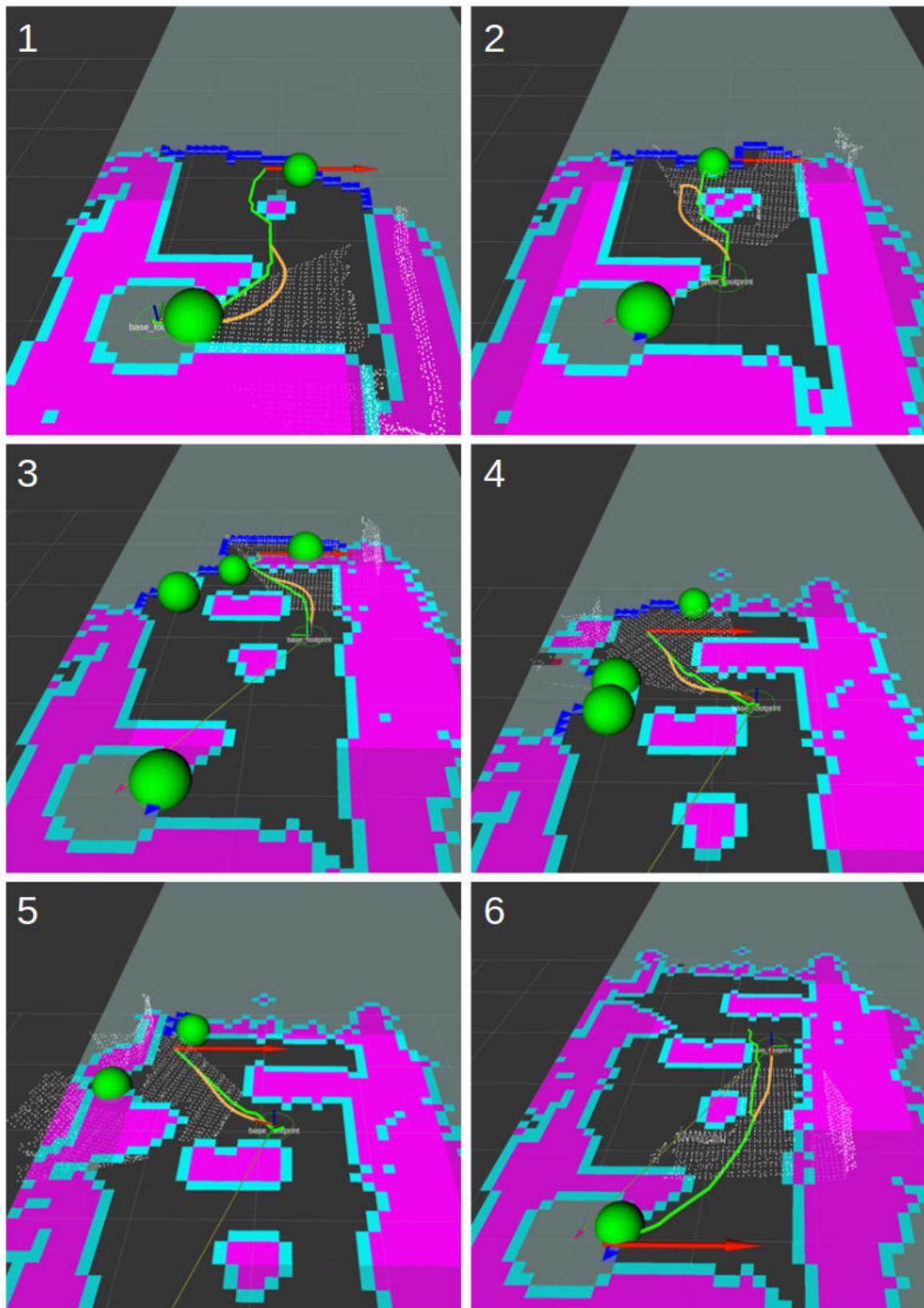


Abbildung 65: Ablauf des Explorationstests

5 Teilnahme und Tests bei Wettbewerben

Da die Tests der Hardware und Software sowohl in der Simulation als auch in einer realen Umgebung erfolgreich verlaufen sind, folgt nun die Übertragung auf den Wettbewerbsroboter Schrödi und die anschließende Testphase. Es ist jedoch zu beachten, dass zum Zeitpunkt der Softwarefertigstellung bereits die Vorbereitungen für die Teilnahme an den Wettbewerben im Gange waren. Aus zeitlichen und kostentechnischen Gründen war es daher nicht möglich, im Labor einen Parcours aufzubauen und die Tests in einer kontrollierten Umgebung durchzuführen. Stattdessen werden die Tests direkt während der Wettbewerbe German Open und Weltmeisterschaft durchgeführt.

5.1 Vorbereitung

Während die verbleibende Zeit vor den Wettbewerben nicht ausreichend ist, um umfassende Tests durchzuführen, bietet sie dennoch ausreichend Gelegenheit, einige Vorbereitungen zu treffen. Hardwareseitig können bereits Maßnahmen ergriffen werden, wie die Befestigung der Kamera am Roboter. Auf der Softwareseite ist es möglich, die entwickelte Software auf den Roboter zu übertragen, was wiederum kleinere Tests ermöglicht.

5.1.1 Hardware

Wie bereits erläutert, besteht vor dem Wettbewerb die Möglichkeit, die 3D-Kamera am Schrödi zu montieren. Aufgrund des begrenzten Platzes steht jedoch nur eine geeignete Position zur Verfügung. Diese Position befindet sich vorne zwischen den Ketten, an der Abdeckung für den Akkuwechsel, wie in Abbildung 66 dargestellt.

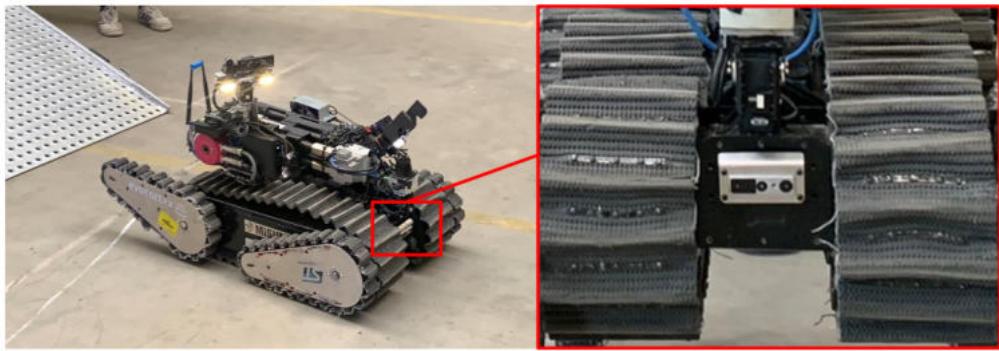


Abbildung 66: Position der 3D-Kamera am Wettkampfroboter Schrödi

5.1.2 Software

Auch auf der Softwareseite lassen sich im Vorfeld des Wettbewerbs Vorbereitungen treffen. Wie bereits im Kapitel der Grundlagen erwähnt, läuft auf dem Wettbewerbsroboter Linux 18.04 mit der ROS-Version Melodic. Daher ist der erste Schritt, die gesamte Software auf diese Version anzupassen. Hierfür wird die zweite Linux-Partition auf dem Dual-Boot-Entwicklungsrechner genutzt. Auf dieser Partition wird zunächst ROS Melodic installiert, und anschließend wird schrittweise versucht, die Tests am Kobuki mit älteren Softwareversionen zu wiederholen.

Der erste durchgeführte Test ist der des Kobuki-Roboters. Dieser Test kann jedoch übersprungen werden, da die dafür benötigte Software auf dem ODROID-Rechner des Kobuki läuft und nicht auf dem Entwicklungsrechner.

Anschließend erfolgt der Test des Mappings. Die hier verwendete Version des `elevation_mapping`-Pakets funktioniert genauso wie die Version für ROS Noetic. Allerdings ergeben sich einige Komplikationen bei der Verwendung der `grid_map`-Bibliothek im Zusammenhang mit dem `manipulate_map`-Knoten. Bei Verwendung der überladenen `:=`-Operation zum Kopieren der empfangenen Karte auf die zu bearbeitende Karte verschieben sich die Indizes, wodurch die gesamte Karte verschoben wird. Daher ist es notwendig, über die zu bearbeitende Karte zu iterieren, für jede Zelle die Position abzurufen und dann den Wert der Position aus der empfangenen Karte zu kopieren. Abgesehen von dieser geringfügigen Anpassung kann der restliche Mapping-Softwarecode eins zu eins übernommen werden.

Die Software für die letzten beiden Tests des `move_base`- und `explore_lite`-Pakets kann ohne Probleme aus den neueren Versionen übernommen werden. Es treten keine Komplikationen auf, und die Tests können erfolgreich wiederholt werden.

Vor der Übertragung der Software auf Schrödi wird ein zusätzlicher Knoten namens `state_controller` entwickelt. Dieser Knoten dient zur Implementierung einer Zustandsmaschine. Obwohl die genauen Regeln des Wettbewerbs nicht bekannt sind, ist es häufig erforderlich, zwischen dem Startpunkt und dem Zielpunkt des Parcours hin und her zu fahren. Zwar ermöglicht der Explorationsknoten eine Aufzeichnung des Parcours, allerdings bleibt der Roboter stehen, sobald die Aufzeichnung abgeschlossen ist. Um dem Roboter bzw. dem `move_base`-Knoten weitere Ziele zu geben, wird die Zustandsmaschine implementiert.

Der erste Zustand ist die Exploration. Nach Abschluss dieses Zustands wechselt die Zustandsmaschine in den zweiten Zustand, in dem ein zuvor programmiertes Ziel angefahren wird. Sobald das Ziel erreicht ist, wechselt die Zustandsmaschine in den dritten Zustand, der die Startposition als neues Ziel setzt. Nach Erreichen dieses Ziels kehrt die Zustandsmaschine erneut in den zweiten Zustand zurück. Auf diese Weise entsteht eine Schleife, in der nach der Exploration abwechselnd das Ziel und der Startpunkt angefahren werden.

Die Feststellung des Abschlusses der Exploration erfolgt durch Überwachung des Topics `/rosout`. Die meisten Knoten geben über dieses Topic ihre Konsolenausgaben aus. Die relevante Ausgabe, auf die hier abgefragt wird, ist die Nachricht des Explorationsknotens mit dem Inhalt „!Exploration stopped.!“.

Nach den getroffenen Vorbereitungen kann die Software auf Schrödi übertragen werden. Um eine erste Prüfung durchzuführen, wird die Funktionalität des `elevation_mapping`-Knotens getestet, die aus den Daten der Kamera eine entsprechende Karte erstellen soll. Hierbei wird festgestellt, dass eine passende Karte erzeugt wird. Allerdings beginnt sich die Karte aufgrund der ungenauen Odometrie des Roboters langsam im Kreis zu drehen. Angesichts des knappen Zeitrahmens ist geplant, dieses Problem erst während der German Open anzugehen.

5.2 RoboCup German Open

Der erste Wettbewerb, an dem Schrödi im laufenden Jahr teilgenommen hat, waren die RoboCup Rescue German Open 2023, die im Deutschen Rettungsrobotik-Zentrum in Dortmund ausgetragen wurde. Dieses Ereignis erstreckt sich über einen mehrere Tage umfassenden Zeitraum. Die Auftaktphase beinhaltete die Anreise und die Aufbauphase, welche am Sonntag stattfand. Am darauffolgenden Montag erhielten die teilnehmenden Teams die Gelegenheit, die verschiedenen Parcours für Trainingseinheiten zu nutzen. Die eigentliche Vorrunde des Wettbewerbs wurde am Dienstag und Mittwoch ausgetragen, wobei die Finalrunden am Donnerstagnachmittag stattfanden. Die abschließende Siegerehrung erfolgte am Donnerstagnachmittag. Insgesamt nahmen sieben Teams an diesem Wettbewerb teil, wobei eines der teilnehmenden Teams von Schülern gebildet wurde.

5.2.1 Vorbereitung und Training

Nachdem der Roboter ausgepackt und erfolgreich aufgebaut wurde, wurden zunächst einige Standardtests durchgeführt, um die grundlegenden Funktionen des Roboters zu überprüfen. Bei der erneuten Durchführung des Elevation-Mappings traten erneut die Odometrie-Probleme auf. Da das gesamte Team keine unmittelbare Lösung für dieses Problem finden konnte, wurden die Testläufe vorübergehend mit beeinträchtigter Odometrie durchgeführt. Als Vorbereitung für das `move_base`-Paket wurde der Fußabdruck des Roboters angepasst. Weitere Anpassungen und Tests werden im Rahmen der kommenden Testläufe durchgeführt.

Der Parcours für diese Testläufe ist in Abbildung 67 dargestellt. Dieser Parcours spiegelt einige der Herausforderungen wieder welche auch beim auf dem Dachboden aufgebauten Testparcours berücksichtigt wurden. Insbesondere enthält der Parcours Kurven, die es zu bewältigen gilt. Zudem sind durchfahrbare Engstellen vorhanden, welche in grün markiert sind. Eine nicht durchfahrbare Engstelle ist in rot gekennzeichnet und zeichnet sich durch einen einzelnen Stein aus. Zusätzlich enthält der Parcours ein größeres Loch, welches in blau markiert ist und sich in der Mitte des Parcours befindet. Somit sind sämtliche Herausforderungen, die im heimischen Testparcours berücksichtigt wurden, auch im aktuellen Testparcours präsent.

In diesem Parcours wurden zunächst Tests mit dem `move_base`-Knoten durchgeführt. Grundsätzlich funktioniert das Paket, und der Roboter versucht autonom, dem in RViz

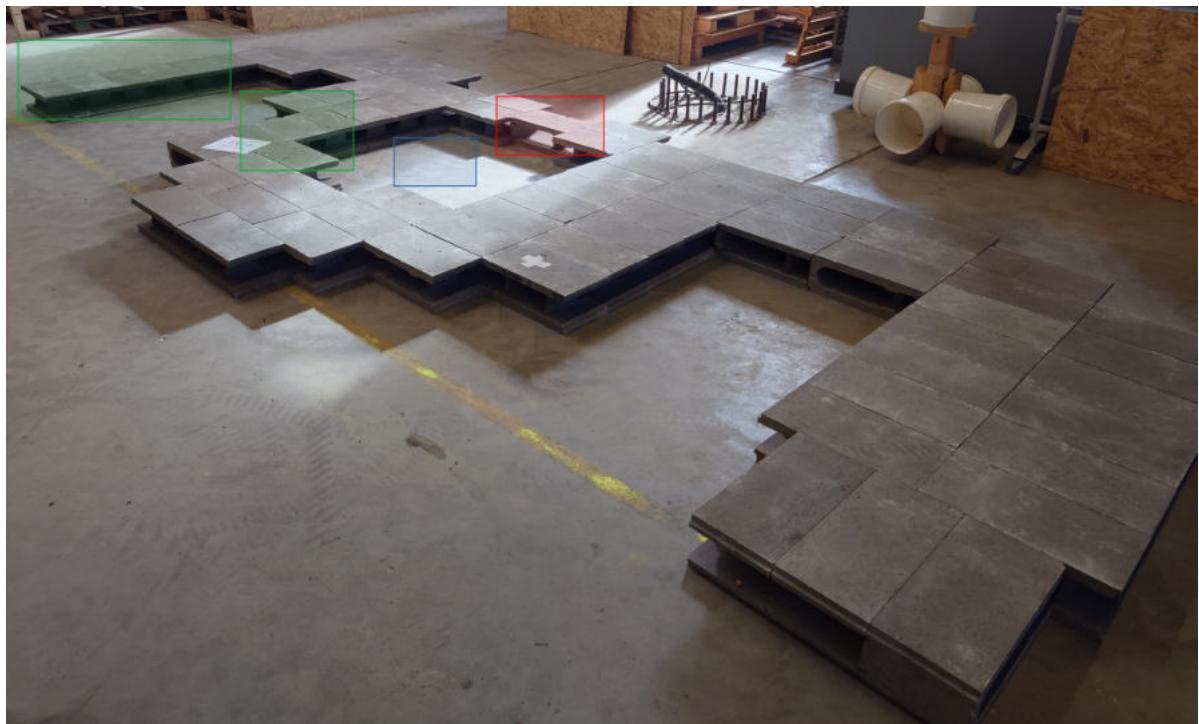


Abbildung 67: Testparcours der German Open

festgelegten Zielpunkt zu folgen. Allerdings kommt es aufgrund der begrenzten Rechenleistung des Schrödi zu einer Verzögerung bei der Aktualisierung der Karte, wodurch dem `move_base`-Knoten die benötigten Kartendaten fehlen. Daher wurden die Geschwindigkeitsparameter drastisch reduziert, um sicherzustellen, dass sich der Roboter nur sehr langsam bewegen kann. Eine zusätzliche Herausforderung besteht darin, dass der `move_base`-Knoten in Anbetracht des nun rechteckigen Fußabdrucks des Roboters während der Pfadplanung Schwierigkeiten hat und dazu neigt, sich leichter in Engpässen zu verfangen.

Angesichts dieser Herausforderungen, einschließlich der Probleme mit der rotierenden Odometrie, wurde es für unpraktisch erachtet, einen autonomen Explorationstest durchzuführen. Insgesamt war es nicht möglich, den Parcours aufgrund dieser Schwierigkeiten erfolgreich autonom zu durchfahren. Dennoch hat dieser Test wertvolle Einblicke in die Herausforderungen der Software geliefert und die Identifizierung der auftretenden Probleme ermöglicht.

5.2.2 Wettbewerb

Trotz einiger Schwierigkeiten während des Testlaufs wurde der Punktelauf durchgeführt. Für diesen speziellen Wettbewerbslauf wurde der Parcours leicht modifiziert. Sowohl der Roboter Schrödi als auch der angepasste Parcours sind in Abbildung 68 dargestellt. Einige Änderungen wurden am Parcours vorgenommen, darunter die Abmilderung der befahrbaren Engstellen durch das Hinzufügen von zusätzlichen Steinen. Auch die nicht durchfahrbare Engstelle wurde verändert, indem ein Stein entfernt wurde, um die Engstelle länger zu machen. Des Weiteren wurde im hinteren Bereich des Parcours ein Holzbrett mit verschiedenen Geschicklichkeits- und Erkennungsaufgaben platziert.

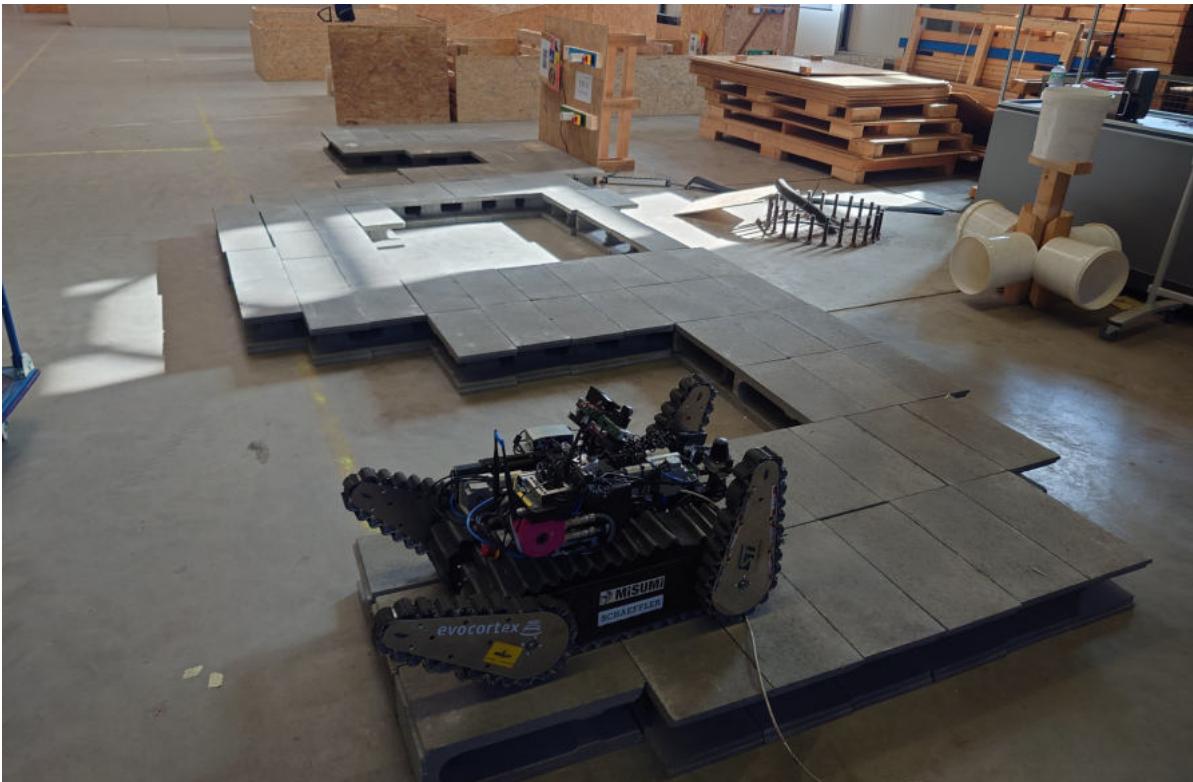


Abbildung 68: Schrödi im Startbereich des Wettbewerbsparcours

Der Ablauf des Punktelaufs gestaltet sich wie folgt: Der Roboter muss den gesamten Parcours einmal durchfahren, um Punkte zu sammeln. Dies wird abwechselnd von Startpunkt zu Zielpunkt und von Zielpunkt zu Startpunkt durchgeführt. Während jeder Durchfahrt besteht die Möglichkeit, Bonuspunkte zu erzielen, indem bestimmte Geschicklichkeitsaufgaben erfolgreich bewältigt werden. Vor dem offiziellen Start des Wettbewerbs ist es im Startbereich, in dem sich der Roboter in der Abbildung befindet,

erlaubt, eine teleoperierte Navigation durchzuführen. Dies ermöglicht es dem Roboter, sich vor dem Start um 360 Grad zu drehen, ähnlich wie es bereits in den Simulationstests oder mit dem Kobuki-Roboter möglich war. Der eigentliche Wettbewerb besteht darin, innerhalb von 20 Minuten so viele Durchfahrten und Geschicklichkeitstests wie möglich zu absolvieren.

Trotz des weniger erfolgreichen Testlaufs am Vortag gelang es im Punktelauf, eine autonome Fahrt durch den Parcours durchzuführen und Punkte zu sammeln. Es war jedoch nicht möglich, den Parcours ein zweites Mal in umgekehrter Richtung zu durchfahren. Dies liegt daran, dass sich die Karte aufgrund der Rotation der Odometrie verändert, was dazu führt, dass die Kartendaten und der Parcours nicht mehr übereinstimmen. Infolgedessen wäre eine erfolgreiche Navigation in entgegengesetzter Richtung unmöglich gewesen.

5.2.3 Resümee

Insgesamt kann die Teilnahme am RoboCup hinsichtlich dieser Arbeit und der entwickelten Software als erfolgreich betrachtet werden. Es markierte das erste Mal, dass Schrödi in dieser speziellen Aufgabe Punkte sammeln konnte. Dies unterstreicht die grundsätzliche Funktionsfähigkeit der entwickelten Software. Für die anstehende Weltmeisterschaft steht die Herausforderung bevor, die Schwierigkeiten des Schrödi, wie Rechenleistung und Odometrie, anzugehen und zu bewältigen.

5.3 RoboCup Weltmeisterschaft

Im Verlauf des aktuellen Jahres nahm das Team AutonOHM an einem weiteren Wettbewerb teil, der RoboCup Weltmeisterschaft 2023 in Bordeaux. Ähnlich der Deutschen Roboter Meisterschaft erstreckte sich auch dieser Wettbewerb über mehrere Tage. Die Anreise und der Aufbau erfolgten an einem Sonntag. Montag und Dienstag waren den Trainingsläufen gewidmet. Die eigentliche Vorrunde fand von Mittwoch bis Freitag statt, während am Samstag das Halbfinale ausgetragen wurde. Schließlich erreichte der Wettbewerb am Sonntag seinen Höhepunkt mit dem Finale und der anschließenden Siegerehrung. Insgesamt waren bei diesem Wettbewerb im Bereich Rescue 14 Teams vertreten.

5.3.1 Vorbereitung und Training

Vor der Teilnahme am RoboCup wurde eines der Probleme, das bei den German Open auftrat, angegangen. Dabei handelte es sich um die begrenzte Rechenleistung des Roboters. Um dieses Problem zu beheben, wurden die veralteten Computer des Schrödi durch modernere Modelle ersetzt. Hinsichtlich der Odometrie-Problematik konnte jedoch nicht rechtzeitig eine Lösung gefunden werden, da Schrödi für eine Verbesserung neue Ketten benötigte. Der Bau dieser Ketten beanspruchte einen erheblichen Teil der Vorbereitungszeit. Es wurde jedoch eine vielversprechende Lösung in Erwägung gezogen, um das Odometrieproblem anzugehen. Die Idee bestand darin, die Odometrie mithilfe des Laserscanners zu erfassen. Diese Methode würde zu einer erheblich präziseren Odometrie führen.

Nach der Ankunft beim RoboCup wurden zunächst grundlegende Funktionsprüfungen am Roboter durchgeführt, gefolgt von mehreren Trainingsläufen. Leider führten zahlreiche kleinere Probleme und die Notwendigkeit umfangreicher Trainingsfahrten dazu, dass die Odometrie nicht rechtzeitig für den Wettbewerbslauf in einen funktionsfähigen Zustand gebracht werden konnte.

5.3.2 Wettbewerb

Abbildung 69 zeigt den Parcours der negativen Hindernisse bei der Weltmeisterschaft. Im oberen linken Bild erhält man einen Überblick über den Parcours. Der Startbereich ist im oberen rechten Bild zu sehen, ebenso wie ein Geschicklichkeitstest in blau. Die folgenden Bilder zeigen die verschiedenen Abschnitte des Parcours.

Neben der Gestaltung des Parcours unterscheiden sich auch die Regeln für diese Aufgabe von denen der German Open. Bei den German Open hatte jedes Team 20 Minuten Zeit, um so viele Wiederholungen wie möglich durchzuführen. Während jeder Wiederholung konnte ein Geschicklichkeitstest absolviert werden. Bei der Weltmeisterschaft wurde dieser Ansatz aufgeteilt. In den ersten 10 Minuten muss der Parcours so oft wie möglich durchfahren werden, bis entweder 10 Wiederholungen erreicht sind oder die Zeit abgelaufen ist. Anschließend stehen den Teams 10 Minuten zur Verfügung, um so viele Geschicklichkeitsaufgaben wie möglich zu bewältigen.

Eine weitere Änderung betrifft die Bedeutung des Mappings bei den negativen Hindernissen. Anders als bei den German Open ist es bei der Weltmeisterschaft gestattet, den Roboter



Abbildung 69: Parcours der Weltmeisterschaft

teleoperiert durch den Parcours zu steuern. Allerdings muss nach dem Lauf eine Karte des Parcours abgegeben werden.

Bedauerlicherweise konnte aufgrund der problematischen Vorbereitung und einem zusätzlichen größeren Problem am Roboterarm dieser Wettbewerb nicht durchgeführt werden. Dennoch soll im Folgenden genauer auf den Parcours, seine Schwierigkeiten und mögliche Lösungsansätze eingegangen werden.

Bereits im Startbereich des Parcours ist zu erkennen, dass neben den Betonblöcken dieses Mal auch Holzpaletten Teil des Parcours sind. Die Holzpaletten sollten jedoch keine zusätzlichen Herausforderungen darstellen. Trotz des gewissen Abstands zwischen den Holzbrettern der Paletten sollte dieser Abstand ausreichend klein sein, um nicht als Hindernis erkannt zu werden.

Bei genauerer Betrachtung des Parcours fallen weitere Besonderheiten auf. Wie im unteren linken Bild der Abbildung zu erkennen ist, ist der Parcours nicht frei stehend. Es

gibt sowohl Wände, die direkt neben dem Parcours aufgebaut sind, als auch Hindernisse, die wie am Auto zu sehen, in den Parcours hineinragen. Doch auch diese Schwierigkeiten sollten keine unüberwindlichen Hindernisse für die Software darstellen. Ähnliche Herausforderungen traten bereits beim Parcours auf dem Dachboden auf, wo Wände in Form des Schrägdachs störten. Diese Störer konnten mithilfe der Software als Hindernisse bzw. Boden erkannt werden.

Schließlich ist zu bemerken, dass die Betonblöcke nicht eng aneinander liegen, sondern eher locker platziert sind. Dieses Problem könnte potenziell schwieriger zu bewältigen sein als die anderen. Die Lücken zwischen den Steinen sind wahrscheinlich groß genug, um als Hindernisse erkannt zu werden. Ein autonomes Durchfahren dieses Bereichs wäre für den Roboter somit nicht möglich. Es gibt mehrere Ansätze, um dieses Problem zu lösen. Eine einfache Lösung wäre, die Karte mit einem Filter, wie zum Beispiel einem Medianfilter, zu bearbeiten. Auf diese Weise könnten kleinere Hindernisse möglicherweise herausgefiltert werden. Eine weitere Option ist es, die Software des `manipulat_map`-Knotens zu erweitern, sodass diese Bereiche der Karte erst ab einer bestimmten Größe als Hindernisse markiert. Die letzte Möglichkeit wäre, einen eigenen Pfadplaner zu entwickeln, der kleinere Hindernisse ignoriert. Dies ist vermutlich die beste, jedoch auch aufwendigste Lösung.

5.3.3 Resümee

Die Teilnahme an der RoboCup Weltmeisterschaft 2023 in Bordeaux war eine bedeutende Erfahrung im Kontext dieser Arbeit. Obwohl die Vorbereitungszeit begrenzt war und einige unerwartete Probleme auftraten, konnte dennoch wertvolles Wissen gewonnen werden, das sowohl die Stärken als auch die Schwächen der entwickelten Software und Hardware aufzeigte.

Der bedeutendste Punkt war die Erkenntnis, wie stark sich die Wettbewerbsparcours von Jahr zu Jahr verändern können. Die Tatsache, dass der Parcours bei der Weltmeisterschaft erhebliche Unterschiede gegenüber dem Parcours der German Open aufwies, verdeutlichte die Notwendigkeit einer hohen Flexibilität und Anpassungsfähigkeit der entwickelten Software. Dies unterstreicht die Wichtigkeit von robusten und generalisierten Algorithmen, die in der Lage sind, sich an unterschiedliche Umgebungen und Hindernisse anzupassen.

6 Zusammenfassung und Ausblick

In dieser Arbeit wurde ein umfassendes Konzept zur Entwicklung und Integration von Software und Hardware für den Rescue-Roboter Schrödi vorgestellt. Die entwickelte Software umfasst Module zur autonomen Navigation, Kartenerstellung und Exploration. Die Hardware-Integration umfasst eine 3D-Kamera zur Umgebungserfassung. Die Konzepte und Lösungen wurden zunächst in Simulationen getestet und anschließend auf einem realen Robotersystem, dem Kobuki, validiert. Schließlich wurden die entwickelte Software und Hardware auf Schrödi übertragen und auf den Wettbewerben German Open und Weltmeisterschaft angewendet.

Die umfassenden Tests in simulierten und realen Umgebungen haben gezeigt, dass die entwickelte Software und Hardware in der Lage sind, den Roboter autonom durch komplexes Terrain zu navigieren, Karten zu erstellen und Explorationen durchzuführen. Trotz einiger Herausforderungen wie der begrenzten Rechenleistung des Roboters und der Ungenauigkeiten in der Odometrie konnten bedeutende Fortschritte erzielt werden.

Obwohl die entwickelte Software und Hardware bereits beeindruckende Ergebnisse erzielt haben, gibt es dennoch Raum für zukünftige Verbesserungen und Erweiterungen. Ein Bereich, der weiter ausgebaut werden kann, ist die Robustheit der Navigation und Pfadplanung. Durch die Optimierung der Algorithmen und die Berücksichtigung von Unsicherheiten in der Umgebung kann autonomes Navigieren in noch anspruchsvolleren Szenarien ermöglicht werden.

Ein interessanter Ansatz für zukünftige Entwicklungen ist der Ersatz der 3D-Kamera durch moderne, kompakte Lidar-Sensoren („Light Detection and Ranging“-Sensoren). Diese könnten eine präzisere und zuverlässigere Erfassung der Umgebung ermöglichen und gleichzeitig die Hardwareanforderungen reduzieren. Die Integration dieser fortschrittlichen Sensoren erfordert jedoch eine sorgfältige Anpassung der Softwarearchitektur und Algorithmen.

Insgesamt bietet die entwickelte Lösung eine solide Grundlage für zukünftige Forschung und Weiterentwicklung im Bereich der autonomen Robotik, sowohl für Wettbewerbe als auch für reale Anwendungen.

Literaturverzeichnis

- [1] Mobile Robotik – Technische Hochschule Nürnberg Georg Simon Ohm, November 2020. [Online]. Available: <https://www.th-nuernberg.de/fakultaeten/efi/forschung/forschungsaktive-labore/mobile-robotik/> (12.09.2023).
- [2] A Brief History of RoboCup, 2023. [Online]. Available: https://www.robocup.org/a_brief_history_of_robocup (12.09.2023).
- [3] Ubuntu release cycle, 2023. [Online]. Available: <https://ubuntu.com/about/release-cycle> (30.08.2023).
- [4] ROS: Home, 2021. [Online]. Available: <https://www.ros.org/> (30.08.2023).
- [5] Packages - ROS Wiki, April 2019. [Online]. Available: <http://wiki.ros.org/Packages> (30.08.2023).
- [6] Florian Weißhardt. ROS introduction. 2023. [Online]. Available: <https://publica-rest.fraunhofer.de/server/api/core/bitstreams/29dd16c5-3ee2-4a90-aff5-c66d5cce5110/content> (30.08.2023).
- [7] Intro to ROS — ROS Tutorials 0.5.2 documentation, 2015. [Online]. Available: <https://www.clearpathrobotics.com/assets/guides/melodic/ros/Intro%20to%20the%20Robot%20Operating%20System.html> (30.08.2023).
- [8] What is a ROS Message?, 2023. [Online]. Available: <https://roboticsbackend.com/what-is-a-ros-message/> (30.08.2023).
- [9] roslaunch - ROS Wiki, October 2019. [Online]. Available: <http://wiki.ros.org/roslaunch> (30.08.2023).
- [10] ROS Param YAML Format, 2023. [Online]. Available: <https://roboticsbackend.com/ros-param-yaml-format/> (30.08.2023).

- [11] rviz - ROS Wiki, May 2018. [Online]. Available: <http://wiki.ros.org/rviz> (30.08.2023).
- [12] Tully Foote. tf: The transform library. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6, Woburn, MA, USA, April 2013. IEEE. [Online]. Available: <http://ieeexplore.ieee.org/document/6556373/> (30.08.2023).
- [13] navigation - ROS Wiki, September 2020. [Online]. Available: <http://wiki.ros.org/navigation> (30.08.2023).
- [14] navigation/Tutorials/RobotSetup - ROS Wiki, July 2018. [Online]. Available: <http://wiki.ros.org/navigation/Tutorials/RobotSetup> (30.08.2023).
- [15] Gazebo, 2014. [Online]. Available: <https://classic.gazebosim.org/> (31.08.2023).
- [16] Kobuki User Guide, March 2017. [Online]. Available: <https://www.manualslib.com/manual/1572953/Yujin-Robot-Iclebo-Kobuki.html> (31.08.2023).
- [17] odroid-xu4:odroid-xu4 [ODROID Wiki], April 2023. [Online]. Available: <https://wiki.odroid.com/odroid-xu4/odroid-xu4> (31.08.2023).
- [18] Team AutonOHM. RoboCup Rescue 2023 Team Description Paper AutonOHM, 2023.
- [19] ORBBEC_astra_stereo_s.pdf, 2023. [Online]. Available: https://cdn-reichelt.de/documents/datenblatt/C300/ORBBEC_ASTRA_STEREO_S.pdf (31.08.2023).
- [20] Orbbec3D Stereo S | MYBOTSHOP.DE, 208,95 €, 2023. [Online]. Available: <https://www.mybotshop.de/Orbbec3D-Stereo-S> (31.08.2023).
- [21] kobuki - ROS Wiki, September 2016. [Online]. Available: <http://wiki.ros.org/kobuki> (31.08.2023).
- [22] RoboCup2022_assemblyguide_final.pdf, 2022. [Online]. Available: https://rrl.robocup.org/wp-content/uploads/2022/05/RoboCup2022_AssemblyGuide_Final.pdf (31.08.2023).

- [23] rrl_rulebook_2019_v2.4.pdf, 2019. [Online]. Available: https://rrl.robocup.org/wp-content/uploads/2019/06/rrl_rulebook_2019_v2.4.pdf (31.08.2023).
- [24] TurtleBot3, 2023. [Online]. Available: <https://robots.ros.org/turtlebot3/> (31.08.2023).
- [25] frontier_exploration - ROS Wiki, March 2017. [Online]. Available: http://wiki.ros.org/frontier_exploration (31.08.2023).
- [26] explore_lite - ROS Wiki, December 2017. [Online]. Available: http://wiki.ros.org/explore_lite (31.08.2023).
- [27] move_base - ROS Wiki, September 2020. [Online]. Available: http://wiki.ros.org/move_base (31.08.2023).
- [28] costmap_2d - ROS Wiki, January 2018. [Online]. Available: http://wiki.ros.org/costmap_2d (12.09.2023).
- [29] Péter Fankhauser and Marco Hutter. A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation. In *In: Robot Operating System (ROS)*, volume 625. January 2016. [Online]. Available: https://www.researchgate.net/publication/284415855_A_Universal_Grid_Map_Library_Implementation_and_Use_Case_for_Rough_Terrain_Navigation (12.09.2023).
- [30] Peter Fankhauser, Michael Bloesch, and Marco Hutter. Probabilistic Terrain Mapping for Mobile Robots With Uncertain Localization. *IEEE Robotics and Automation Letters*, 3(4):3019–3026, October 2018. [Online]. Available: <https://www.research-collection.ethz.ch/bitstream/handle/20.500.11850/272110/fankhauser2018.pdf?sequence=1&isAllowed=y> (12.09.2023).
- [31] Porenbeton Planblock 599x199 mm PPW-2-04 kaufen bei OBI, 2023. [Online]. Available: <https://www.obi.de/porenbetonsteine/porenbeton-planblock-599x199-mm-ppw-2-04/p/8447922> (25.09.2023).
- [32] tf - ROS Wiki, October 2017. [Online]. Available: <http://wiki.ros.org/tf> (11.09.2023).

- [33] pcl_ros/Tutorials/VoxelGrid filtering - ROS Wiki, September 2014. [Online]. Available: http://wiki.ros.org/pcl_ros/Tutorials/VoxelGrid%20filtering (11.09.2023).
- [34] teb_local_planner - ROS Wiki, March 2020. [Online]. Available: http://wiki.ros.org/teb_local_planner (09.09.2023).

Anhang

Der Gesamte Anhang inklusive Code, Abbildungen, Quellen, etc. findet sich unter folgender Adresse:

https://github.com/PinznerLe/autonomous_exploration_negative_obstacles_ros.git