

Implementierung einer Roboterkontrollarchitektur für autonome mobile Roboter im industriellen Umfeld

Masterarbeit

im Fachgebiet
Robotik

vorgelegt von: MARCO MASANNEK

Matrikelnummer: 26 71 56 0

Studiengang: Master Applied Research
in Engineering Sciences

Erstprüfer: Prof. Dr. Stefan May

Zweitprüfer: Prof. Dr. Jörg Arndt

Betreuer: Daniel Ammon

© Sommersemester 2021

Schlüsselwörter : Robotik, Autonomie, Logistik, Softwarearchitektur

Abstract

This document describes a concept for a software control architecture that may be used on autonomous mobile robots targeted towards usage in industry-related scenarios. Serving as a benchmark scenario, the concept is implemented on the competition robot „Ohmnibot“, which is used for participation in Robocup Industrial - @Work tournaments by the team AutonOHM.

The team AutonOHM@Work has been very successful in the league, as they are the champions of the german open and the world cup in both 2017 and 2018. Since most members finished their studies in late 2018, the former team has disbanded and a new team was formed early 2019. Due to mechanical issues on the previously used robot, the team built a new system called „Ohmnibot“, which is based on the Evocortex R&D platform. With the experience of past robocups, the platform was upgraded in terms of sensors, electronics, computation unit and manipulator to establish a robotic system that can fulfill the tasks required in the @Work league.

Participating robots must be capable of fulfilling transportation tasks fully autonomously, including the navigation between different locations, avoiding obstacles and detecting and picking objects. As multiple tasks have to be executed in a limited amount of time, the robots also have optimize their order to increase overall efficiency.

As the complexity of such systems requires their software control architecture to be lucid and maintainable, the concepts basic idea is to separate the various tasks by „what must be done?“ and „how is a problem solved?“. Therefore, the overall software architecture classifies individual programs into four layers, which all hold different levels of remits. These are the hardware integration, algorithms and modelling, hardware and model control, and decision making and data management, concluding in the driver-, model-, control- and brain-layer. In addition to that, the robot’s control architecture is separated into task management (what?) and task execution (how?), which ensures that software components may be used on various robot types with only slight adjustments necessary. The resulting software control architecture is implemented on the „Ohmnibot“ using the Robot Operating System (ROS), which enables loose coupling and flexible interprocess communication between layers and programs.

In this document, the general challenges and solution proposals are presented, followed by an explanation of Ohmnibot’s hardware components. Afterwards, the software components are described regarding their functionality and interfaces, with a more detailed explanation of the task management and execution. In conclusion, the resulting autonomous mobile robot system is presented and tested in an official @Work task, proving the concept’s applicability to such industrial scenarios.

Inhaltsverzeichnis

Abkürzungsverzeichnis	I
1 Einleitung	1
1.1 Einleitung	1
1.2 Motivation	1
1.3 Ziel der Arbeit	2
1.4 Projektorganisation	2
1.5 Struktur	4
2 Aufgaben und Lösungsansätze	6
2.1 Kartierung und Lokalisierung	6
2.2 Physische Hinderniserkennung	7
2.3 Optische Hinderniserkennung	8
2.4 Objekterkennung und -lokalisierung	9
2.5 Taskplanung und -ausführung	10
3 Ohmibot als Entwicklungsroboter	12
3.1 Hauptrechner	12
3.2 Antrieb	13
3.3 Manipulator	14
3.4 Sensorik	15
3.4.1 Laserscanner	15
3.4.2 Hinderniskameras	15
3.4.3 3D-Kamera	16
3.5 Beleuchtung	16
3.6 Testumgebung	16
3.7 Simulation	17
4 Entwurf der Softwarekontrollarchitektur	18
5 Hardwareanbindung: Driver-Ebene	21
5.1 Laserscanner	21
5.2 Hinderniskamera	22
5.3 3D-Kamera	23
5.4 Dynamixel Motoren	24
5.5 Grundplattform	25
5.6 Kamerabeleuchtung	26
5.7 Unterbodenbeleuchtung	27
5.8 Montagepunkte und Robotergeometrie	27

6 Algorithmitk und Modelle: Model-Ebene	29
6.1 Laser Filter	29
6.2 SLAM	31
6.3 Lokalisierung	33
6.4 Pfadplanung	34
6.5 Kamerabildprojektion	37
6.6 Kamerabildentzerrung	38
6.7 Neuronale Netze	39
7 Hardware- und Modellkontrolle: Control-Ebene	41
7.1 Joy Controller	41
7.2 Barriertape Controller	42
7.3 Perception Controller	43
7.4 Navigation Controller	45
7.5 Approach Controller	47
7.6 Gripper Controller	47
7.7 Arm Controller	48
8 Entscheidungsfindung und -ausführung: Brain-Ebene	50
8.1 Eigene Messages	50
8.2 Status Monitor	52
8.3 Worldmodel	53
8.4 Task Manager	54
8.5 Task Executioner	66
9 Gesamtsystem	70
9.1 Kartierung	70
9.2 Navigation	71
9.3 Manipulation	72
9.4 Autonomer Betrieb	72
9.5 Roboterverhalten an einem einfachen Beispiel	74
9.6 Offizieller Testlauf	76
10 Zusammenfassung und Ausblick	78
Abbildungsverzeichnis	79
Literaturverzeichnis	82

Abkürzungsverzeichnis

AMR	Autonomer mobiler Roboter
CAD	Computer-aided Design (rechnergestütztes Konstruieren)
DHCP	Dynamic Host Configuration Protocol
FOV	Field of View (Sichtbereich)
GIMP	GNU Image Manipulation Program
GPU	Graphics Processing Unit (Grafische Recheneinheit)
HMI	Human Machine Interface (Mensch-Maschine-Schnittstelle)
LIDAR	Light Detection and Ranging
LiPo	Lithium Polymer
PCL	Point Cloud Library
PNG	Portable Network Graphic
RGB-D	Red Green Blue - Depth (Rot Grün Blau - Tiefe)
ROS	Robot Operating System
SLAM	Simultaneous Localization and Mapping
TCP	Tool Center Point (Effektiver Arbeitspunkt des Werkzeugs)

1 Einleitung

1.1 Einleitung

Das Team AutonOHM der Technischen Hochschule Nürnberg nimmt seit 2015 an der Robocup@Work Liga teil. Die Liga legt den Fokus auf industrienahe Anwendungen, wobei die mobilen Roboter vollständig autonom agieren müssen, um so potentiell mit menschlichen Arbeitern in einer Fabrik zu kooperieren. Unter anderem müssen die Roboter selbstständig durch einen Parcours navigieren, Objekte erkennen und greifen und Logistikaufgaben wie den Transport von Objekten erfüllen [1, 2].

In den Jahren 2017 und 2018 erreichte das Team seinen Höhepunkt und konnte sowohl die Deutsche als auch die Weltmeisterschaft gewinnen. Ende 2018 stand das Team jedoch vor dem Aus, da einerseits fast alle Mitglieder ihr Studium abgeschlossen und somit die Hochschule verlassen haben, und andererseits der in Abb. 1.1 dargestellte Roboter mechanische Verschleißerscheinungen aufwies, die sich aufgrund der mangelnden Verfügbarkeit von Ersatzteilen auch nicht reparieren ließen.



Abbildung 1.1: AutonOHM
youbot 2018

1.2 Motivation

Über die Jahre konnte ein großer Erfahrungsschatz über die Entwicklung von autonomen Robotern aufgebaut werden. Damit dieses angesammelte Wissen fortgeführt und erweitert werden kann, wurde Anfang 2019 der Entschluss gefasst, mit dem alten System als Grundlage und dem letzten verbliebenen Mitglied als Teamleiter ein neues Team aus Studenten aufzubauen. Dabei galt es, die Methodik und die Lösungsansätze des erfolgreichen Youbots sowohl Hard- und Softwaretechnisch zu analysieren und daraus ein Konzept für einen neuen Roboter zu entwickeln.

Ein Teil der finanziellen Hürden der Neubeschaffung eines Roboters konnte durch die Kooperation mit der Firma Evocortex bewältigt werden. Diese entwickelt Lösungen in

verschiedenen Themengebieten rund um die mobile Robotik, so auch eine omnidirektionale Roboterplattform, die skalierbar ist und Kundenwünsche angepasst werden kann. Evcortex erklärte sich dazu bereit, dem Team eine solche Plattform zur Verfügung zu stellen, um so auch den Fortbestand des Teams zu ermöglichen. Die Zusammenarbeit wirkt sich dabei positiv für beide Parteien aus, da das Team die Plattform, welche sich noch in der Entwicklungsphase befindet, intensiv testet und den Entwicklern positives Feedback und Verbesserungsvorschläge gibt. Außerdem knüpfen die Studenten im Team Kontakte in ein lokales Robotikunternehmen.

1.3 Ziel der Arbeit

Auch wenn der alte Roboter die Saisons 2017 und 2018 gewinnen konnte, waren viele der Ideen und Lösungen aufgrund der langjährigen Entwicklung durch wechselnde Teammitglieder relativ prototypisch umgesetzt und daher schlecht erweiterbar. Bei dem Versuch, den Softwarestand auf den neuen Roboter zu übertragen, stellte sich heraus, wie sehr ein Großteil der Komponenten auf den „youbot“ (Abb. 1.1) zugeschnitten war. Außerdem wurde das System immer Stück für Stück erweitert, ohne vorher eine Architektur festzulegen, die die Wiederverwendbarkeit der Komponenten sicherstellt, sodass der Aufwand der nötigen Anpassungen teilweise dem einer Neuentwicklung gleichkommt.

Das Ziel dieser Arbeit ist es daher, auf Grundlage der Ideen und Lösungsansätze und mit den Erfahrungen des alten Systems eine Roboterkontrollarchitektur zu entwickeln, die die Wiederverwertbarkeit der Systemkomponenten sicherstellt und außerdem durch klare Trennung des „Was?“ und „Wie?“ die komplexen Zusammenhänge der Komponenten verständlicher macht.

1.4 Projektorganisation

Dieser Arbeit gehen zwei Projektberichte voraus, die im Rahmen der Forschungsarbeiten im Master of Applied Research erstellt wurden. Darin wurden die Analyse des Youbots [3] sowie das neue Roboterkonzept [4] behandelt, welches dem Team AutonOHM@Work (Abb. 1.2) als Blaupause für 2019 und die Folgejahre dient.

Bei dessen Umsetzung wurden viele Komponenten von neuen Teammitgliedern übernommen, die ihre Aufgabenpakete in Form von studentischen Arbeiten (z.B. Abschlussarbeiten) und freiwilliger Mitarbeit im Team entwickelt haben. Die Rolle des Teamleiters und Systemarchitekten war dabei, die Aufgabenpakete erstellen, deren Bearbeitung zu betreuen und fertige Komponenten ins System zu integrieren.



Abbildung 1.2: Team AutonOHM 2020

Um die Unterstützung des Teams zu würdigen sind die vergebenen Verantwortlichkeiten im Organigramm 1.3 bzw. 1.4 dargestellt. Darin stehen jeweils die Hauptentwickler einer Komponente, wobei ein Stern (*) jene Komponenten markiert, deren Entwicklung durch den Teamleiter besonders unterstützt wurde.

Ehemalige Mitglieder des alten Teams, deren Komponenten auch auf dem neuen Roboter verwendet werden konnten, stehen in grauen Kästen. Kleinere, ergänzende Arbeiten sind nicht detailliert im Organigramm aufgeführt.

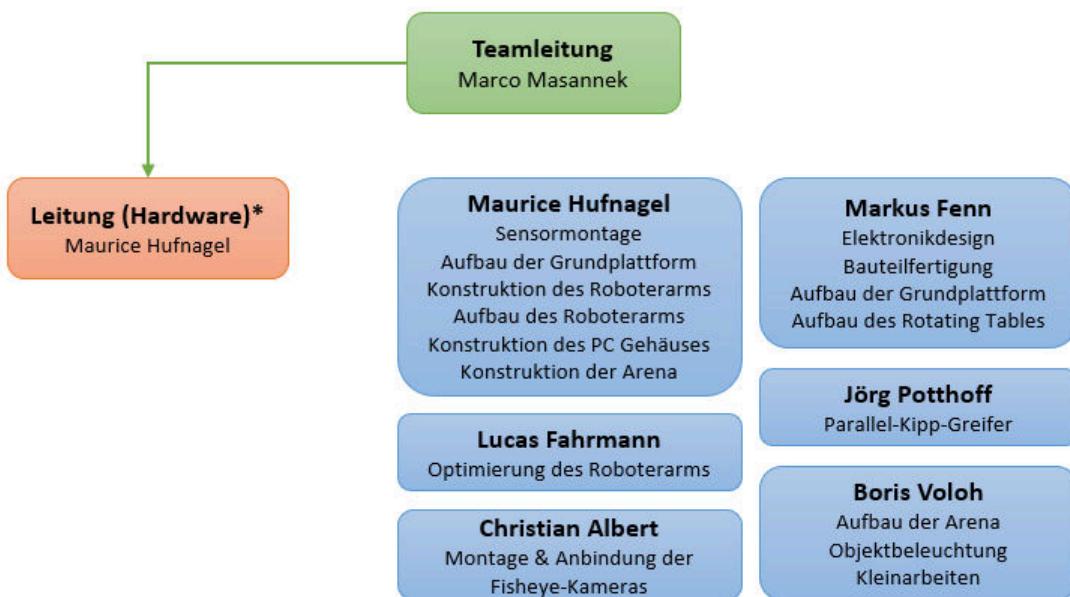


Abbildung 1.3: Organigramm des Teams AutonOHM@Work - Hardware

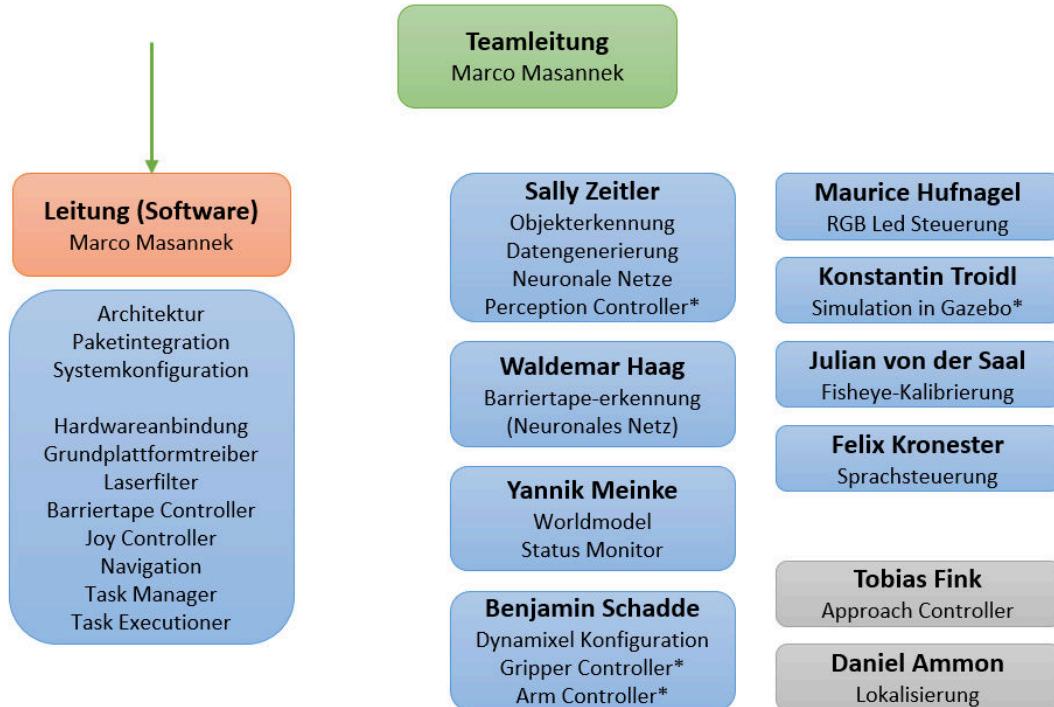


Abbildung 1.4: Organigramm des Teams AutonOHM@Work - Software

Abbildung 1.4 zeigt links die Pakete, welche im Rahmen der vorliegenden Masterarbeit entwickelt wurden, darunter auch der Task Manager und Task Executioner. Diese bilden den Kern der neuen Strategie zur Roboterkontrolle, da hier das „Was?“ und „Wie?“ für das autonome Verhalten des Roboters implementiert sind.

1.5 Struktur

In dieser Arbeit werden zunächst die Aufgaben und möglichen Lösungsansätze zusammengefasst. Dadurch werden die groben Zusammenhänge vorab erläutert, die später durch verschiedene Softwarekomponenten umgesetzt werden. Anschließend wird der neue Roboter vorgestellt, der vom Team in den letzten beiden Jahren aufgebaut und für die Teilnahme an den Wettkämpfen eingesetzt wurde.

In Kapitel 4 wird die neue Softwarearchitektur beschrieben, die die grundlegende Struktur der Programme vorgibt. Die Architektur arbeitet dabei mit verschiedenen Fähigkeitsebenen, die von der einfachen Hardwareanbindung bis zum „Gehirn“ des Roboters reichen, in dem Entscheidungen getroffen werden.

Im Folgenden werden die Komponenten der einzelnen Ebenen beschrieben, wobei überwiegend deren Funktion und Schnittstellen im Fokus liegen. Die genauen Implementierungen bedürfen meist einer Dokumentation in einer gesonderten Abschlussarbeit, weshalb in dieser darauf verzichtet wird.

Da Task Manager und Task Executioner das Verhalten des Roboters bestimmen, bzw. dessen Hardware kontrollieren, werden diese detaillierter behandelt. Dadurch wird auch die Grundlage für Kapitel 9 gelegt, in dem das resultierende Gesamtsystem und einige Beispiele dokumentiert sind, wodurch die Anwendbarkeit der Architektur auf autonome mobile Roboter bewiesen wird.

Abschließend folgt eine Zusammenfassung des aktuellen Stands und ein Ausblick auf Arbeiten, die sich noch in der Entwicklungsphase befinden und daher noch nicht auf dem Roboter zum Einsatz kommen.

2 Aufgaben und Lösungsansätze

Die Roboter in der @Work-Liga müssen vollautonom eine Liste mit Aufgaben abarbeiten. Hierfür müssen verschiedene Schwerpunkte der Forschung gelöst werden. Dabei kann teilweise auf bestehende Lösungen zurückgegriffen werden, die für den eigenen Fall angepasst werden.

2.1 Kartierung und Lokalisierung

Die Arena ist so aufgebaut, dass die Roboter auf einem festen Boden zweidimensional manövrieren können. Das Layout einer Arena beinhaltet Wände und die Workstations (dt: Arbeitsstation), welche beide beliebig aufgestellt werden können. Die einzige Regel hierbei ist, dass zwischen den festen Arenaelementen ein Abstand von mindestens 80 cm bestehen muss. Damit sich ein Roboter sicher durch den Parcours bewegen kann, ohne mit Wänden oder Workstations zu kollidieren, muss zunächst eine Karte der Arena erstellt werden. Mit Hilfe dieser Karte kann sich der Roboter dann in seiner Umgebung lokalisieren, also seine eigene Position innerhalb des Parcours bestimmen. Außerdem können Zielpositionen in dieser Karte eindeutig definiert werden, zu denen unter Berücksichtigung der Wände eine sichere Route geplant werden kann.

Da sich die Roboter lediglich zweidimensional bewegen müssen, und außerdem keine Elemente wie z.B. Schreibtische existieren, bei denen ein Roboter beachten muss, ob er beim darunter durchfahren mit seinem Aufbau kollidieren würde, kann eine Top-Down Karte der Arena verwendet werden. In dieser sind die Umrisse der Hindernisse und Freiflächen eingezeichnet. Zur Generierung einer solchen Karte wird in der Liga überwiegend auf 2D-LiDAR-Sensoren zurückgegriffen.

Wird ein solcher Laserscanner so über dem Boden montiert, dass die ausgesendeten Lichtsignale parallel zum Boden verlaufen, können die Abstände zu Wänden und den Workstations gemessen werden. Wenn diese Daten in einem Koordinatensystem mit dem Scanner als Ursprung aufgetragen werden, können die Raumumrisse in den Abtastpunkten erkannt werden. Bei einer ausreichend hohen Auflösung, also einem möglichst geringen Winkelabstand zwischen zwei Messstrahlen, können gerade Flächen wie z.B. Wände, aber auch Raumecken eindeutig erkannt und markiert werden. Dann können die einzelnen Punkte zu einer Linie fusioniert werden.

Dieses Prinzip kann genutzt werden, um ein Abbild des Raums zu erstellen. Bei einer einzelnen Messung gilt dieses Abbild jedoch nur für die Position, an der sich der Laserscanner zum Aufnahmezeitpunkt befindet. Wird der Sensor bewegt, so ändern sich nicht nur die

Abstände zu den Umgebungselementen, sondern es können unter Umständen auch neue Bereiche erschlossen werden. Da es in einem typischen Arenalayout nicht möglich ist, jeden Blickwinkel und jedes Element von einer einzelnen Position aufzunehmen, muss der Sensor durch die Arena bewegt werden. Dadurch entstehen mehrere Aufnahmen von Teilbereichen, die zu einem großen Ganzen zusammengefügt werden müssen. Ein Verfahren, mit dem dies möglich ist, wird als SLAM (Simultaneous Localization and Mapping, dt: Gleichzeitige Lokalisierung und Kartierung) bezeichnet.

Dabei wird ein Sensor durch einen Raum bewegt und die Aufnahmen miteinander verglichen. Bei einer hohen Abtastrate (z.B. 5 Hz) ist es bei geringer Bewegungsgeschwindigkeit sehr wahrscheinlich, dass die gleichen Elemente in zwei aufeinander folgenden Abbildern enthalten sind. Durch die Identifikation von besonderen Merkmalen und dem Suchen und Übereinanderlegen dieser Merkmale kann die Verschiebung zwischen den beiden Aufnahmen bestimmt werden. Diese Verschiebung kann als Sensorbewegung zwischen den Aufnahmen interpretiert werden, wodurch dessen neue Position ermittelt werden kann. Wird dies nun für den gesamten Datenstrom des Sensors durchgeführt, können die Aufnahmen stets im richtigen Kontext in die Gesamtkarte eingeordnet werden.

Die so erstellte Gesamtkarte kann dann zur Lokalisierung im autonomen Betrieb genutzt werden. Hierfür werden jeweils die aktuellen Sensorwerte mit den zu erwartenden Werten verglichen, es wird also in den Aufnahmen nach bekannten Strukturen gesucht. Wenn sich der Sensor z.B. in einem rechteckigen Raum befindet, so sind die Wände und Ecken in der Karte gespeichert. Wenn nun diese Strukturen im Laserscan wieder auftauchen, kann die Position des Sensors geschätzt werden, indem versucht wird, die Aufnahme und die Karte übereinander zu legen. Daraus ergibt sich eine wahrscheinliche Sensorposition, für die die Übereinstimmung am höchsten ist. Mit dieser Sensorposition kann dann auch auf die Position des Roboters, an dem der Sensor montiert ist, rückgeschlossen werden.

Da sowohl SLAM als auch die einfache Lokalisierung in der bestehenden Karte anfällig für Fehler sind, die durch Messfehler oder Kartenungenauigkeiten entstehen, wird üblicherweise die Roboterodometrie mit einbezogen. Diese wird aus den Radumdrehungen und dem kinematischen Modell des Roboters berechnet und beinhaltet dessen aktuelle Geschwindigkeit sowie eine daraus aufsummierte Position. Unter Zuhilfenahme dieser Hinweise auf die Bewegung des Roboters können z.B. die Kartenteile sicherer zusammengefügt werden, da die Verschiebung schon teilweise bekannt ist. So kann die Genauigkeit erhöht und der Rechenaufwand verringert werden.

2.2 Physische Hinderniserkennung

Das Regelwerk der @Work-Liga sieht in einigen Läufen physische Hindernisse vor, die erst kurz vor dem Wettkampf in der Arena platziert werden. Diese Hindernisse fehlen dann

in der zuvor aufgenommenen Karte, dürfen aber dennoch nicht unbeachtet bleiben, da der Roboter sonst mit ihnen kollidieren könnte. Sie können aber auch nicht direkt in die bestehende Karte eingetragen werden, da sonst die Originalkarte verfälscht würde, was in anderen Läufen, in denen die Hindernisse fehlen oder an anderen Positionen stehen, zu Problemen führt.

Daher werden für die Navigation mehrere Schichten der Karte erstellt, die zu einer Navigationskarte kombiniert werden können. Zusätzlich zur Grundsicht (der Originalkarte) kann eine Hinderniskarte erstellt werden, in der nur die aktuellen Laserscandaten eingetragen werden. Überall dort, wo in der Karte bereits Wände mit dem Laser aufgenommen wurden, können die Laserdaten unbeachtet bleiben. In freien Bereichen werden sie jedoch als blockierend markiert, sodass sie für die Pfadplanung berücksichtigt werden können.

Durch diese zusätzliche Schicht können nicht nur statische Objekte erkannt werden, sondern auch bewegte Hindernisse, z.B. eine durch die Arena laufende Person. Diese würde eine blockierende Spur hinter sich herziehen, wenn blockierende Elemente nur hinzugefügt, jedoch nie entfernt würden. Daher werden die Laserstrahlen des Scanners auf ihrer gesamten Bahn beobachtet. Trifft ein Strahl „durch“ ein zuvor erkanntes Hindernis auf eine dahinter liegende Wand, so ist der Bereich zwischen Sensor und Wand frei und kann auch dementsprechend in der Hinderniskarte markiert werden. Dadurch ist nicht nur sichergestellt, dass bewegte Objekte nur in ihrer aktuellen Position erfasst werden, sondern auch, dass zuvor blockierte Bereiche nach dem Entfernen eines Hindernisses aktualisiert werden.

Dies kann z.B. bei einem Tor vorteilhaft sein. Ist dieses zu Beginn geschlossen, so wird der Zugang als blockiert vermerkt. Wenn der Roboter nach einer gewissen Zeit wieder an der Stelle vorbei kommt, an der auch dieses Tor abgetastet wird, und es wurde zwischenzeitlich geöffnet, so ergibt sich womöglich ein neuer Pfad für den Roboter.

2.3 Optische Hinderniserkennung

Hindernisse, die nicht mit dem Laserscanner erfasst werden können, weil sie z.B. wie das Barriertape (dt: Absperrband) nur auf dem Boden aufgeklebte Markierungen sind, müssen anderweitig erkannt und markiert werden. Hierfür können Kameras verwendet werden, die am Roboter angebracht sind und dessen Umgebung aufnehmen. Anders als bei einem Distanzsensor, für den jeder aufgenommene Punkt ein potentielles Hindernis darstellt, müssen die Sensordaten jedoch zunächst auf zu erwartende Hindernisse untersucht werden. Nur so kann in einem Kamerabild zwischen Hindernis und normaler Umgebung unterschieden werden.

Hierfür wird eine Bildverarbeitungsalgorithmik benötigt, die diese Aufgabe robust und schnell ausführen kann. Der aktuelle Stand der Technik hierfür sind neuronale Netze. Diese werden mit realistischen Aufnahmen aus der Perspektive des Roboters trainiert, in denen die Bereiche markiert sind, die die gesuchten Objekte beinhalten. Durch eine sorgfältig gewählte Kombination aus Netzarchitektur, Parametern und Trainingsdaten kann ein Netz trainiert werden, welches mit hoher Genauigkeit die Zielobjekte erkennen kann, und dabei wenig Fehldetections produziert.

Im Betrieb werden in dieses Netz dann die rohen Kamerabilder eingegeben und als Ausgabe erhält man eine binäre Bildmaske, in der nur noch die erkannten Pixel markiert sind. Mit der Kameraposition auf dem Roboter und dem Lochkameramodell [5] kann die Bildmaske auf den Boden projiziert werden, sodass die Position des Barriertapes in Relation zum Roboter bestimmt werden kann.

Damit das erkannte Barriertape auch als Hindernis für die Navigation berücksichtigt werden kann, muss der Hinderniskarte eine weitere Schicht hinzugefügt werden. Diese beinhaltet ausschließlich die optischen Hindernisse, wird jedoch final mit den beiden anderen Schichten kombiniert. Da es bei den Barriertapes nicht möglich ist, die freien Bereiche mit der selben Strategie wie die Laserstrahlverfolgung zu überprüfen, muss eine gesonderte optische Hinderniskarte erstellt werden. Diese trägt erkannte Objekte mit den entsprechenden Koordinaten in eine Karte ein und überprüft anschließend für jede neue Messung, ob der Aufnahmebereich auch zuvor erkannte Objekte beinhaltet. Ist dies der Fall, so kann die Karte aktualisiert werden, falls sich z.B. eine genauere Position des Objekts ergibt. Wird im neuen Bild nichts erkannt, so kann der Bereich der Karte wieder als frei markiert werden. Dadurch wird auch hier sichergestellt, dass die Umgebung stets möglichst realitätsnah abgebildet wird.

2.4 Objekterkennung und -lokalisierung

Im Wettkampf werden aktuell 13 verschiedene Objekte verwendet, die vom Roboter gegriffen und transportiert werden müssen. Diese können flach oder stehend ausgerichtet sein, sodass sich verschiedene Blickwinkel ergeben. Außerdem werden in manchen Läufen rote und blaue Sichtlagerboxen auf den Workstations platziert, in die die Objekte abgelegt werden sollen. Beim Precise Placement Test müssen außerdem Negativformen für die Objekte erkannt werden.

Anders als bei der Erkennung von Barriertape muss hier also eine Vielzahl von Zielobjekten korrekt erkannt und außerdem zugeordnet werden. Auch hier kann ein neuronales Netz zum Einsatz kommen. Dieses markiert jedoch nicht nur Pixel, sondern ganze Bildbereiche

mit zugehörigem Label. Durch Rückgabe einer Liste mit erkannten Objekten samt Wahrscheinlichkeit können mit nur einem einzigen Bild einer Tischoberfläche theoretisch alle nötigen Informationen für das Greifen gesammelt werden.

Da es jedoch seit der Einführung der Arbitrary Surfaces nicht mehr möglich ist, die Tischhöhen bei der Kartenerstellung fest abzuspeichern, muss vor jeder Manipulationsaufgabe die Oberfläche der Workstation detektiert werden. Dafür kann eine 3D-Kamera verwendet werden, die zusätzlich zum Farbbild eine 3D-Punktwolke der aufgenommenen Szene liefert, in der nach Ebenen gesucht werden kann. Wenn der Bildausschnitt überwiegend eine Tischoberfläche zeigt, liegt auch eine Mehrzahl der 3D-Punkte auf dieser Oberfläche. Mittels Algorithmen wie RANSAC kann dann eine Ebene gemittelt werden.

Durch die Kombination der markierten Bildbereiche eines Objekts und der Tischoberfläche kann auf die Position des Objekts rückgeschlossen werden. Je nach Objektgeometrie kann daraus der Objektmittelpunkt und eine optimale Greifposition bzw. der Ablagepunkt ermittelt werden.

2.5 Taskplanung und -ausführung

In den Läufen erhält der Roboter eine Liste mit Transportaufgaben, die in der vorgegebenen Zeit zu erledigen sind. Dabei steigt die Anzahl und der Schwierigkeitsgrad der Aufgaben mit dem Fortschreiten des Wettkampfs, sodass immer mehr Workstations angefahren werden müssen. Da jedoch manche Workstations doppelt belegt sind, also dass mehr als eine Manipulationsaufgabe (Greifen bzw. Ablegen) an derselben Workstation absolviert werden muss, kann es sich lohnen, zunächst die gesamte Aufgabenliste zu betrachten und gegebenenfalls umzusortieren. Dadurch kann die Zeit, die insgesamt benötigt wird, erheblich reduziert werden, da Wege eingespart werden können.

Bei der Optimierung müssen mehrere Faktoren berücksichtigt werden. Zunächst ist natürlich die Zielworkstation im Vordergrund. Hier muss die geschätzte Navigationszeit betrachtet werden, die der Roboter von seiner aktuellen Position bis zum angedockten Zustand benötigt, in dem dann manipuliert werden kann. Außerdem darf der Roboter maximal drei Objekte gleichzeitig transportieren. Es können also z.B. nicht vier Objekte nacheinander gegriffen werden, sondern es muss zwischenzeitlich mindestens einmal abgelegt werden. Ein Objekt, das noch garnicht gegriffen wurde, kann aber logischerweise auch nicht abgelegt werden. Daher muss stets das Roboterinventar überwacht werden.

Zusätzlich zur Navigationszeit wird die Zeit für die Manipulation eingerechnet. Das Greifen bewegter Objekte dauert beispielsweise länger als das simple Ablegen. Außerdem wird für die Objekterkennung auch Zeit eingeplant, da es unter Umständen etwas dauern kann, bis die ganze Tischoberfläche gescannt und das Zielobjekt gefunden wurde.

Vor allem bei Aufgabenlisten mit vielen schwierigen Aufgaben kann es sich außerdem lohnen, nicht nur nach der geschätzten benötigten Zeit zu optimieren, sondern auch das Fehlerrisiko und die dafür vergebenen Punkte mit einzubeziehen. Je nachdem, wie diese Variablen mit einbezogen werden, können schwierige Aufgaben in der Priorität nach hinten rutschen, da die Erfolgswahrscheinlichkeit und damit der erwartete Nutzen geringer ist. Dies ist vor allem dann relevant, wenn der Entwicklungsstatus des Roboters noch nicht soweit ist, dass alle Aufgaben zuverlässig gemeistert werden können. Dann sollte die Aufgabenreihenfolge so gestaltet werden, dass innerhalb der Wettkampfzeit möglichst viele Punkte gesammelt werden können, anstatt die minimale Gesamtzeit zu ermitteln.

Für n Transportaufgaben, bei der jede aus einmal Greifen und einmal Ablegen besteht, existieren $(2 * n)!$ theoretisch mögliche Reihenfolgen. Davon sind jedoch einige unmöglich, da bei anfangs leerem Inventar nicht mit dem Ablegen begonnen werden kann. Dennoch muss aus einer Vielzahl von Kombinationen die Beste ermittelt werden. Wenn stumpf jede Kombination erstellt und bewertet wird, benötigt ein moderner PC je nach Implementierung bereits ab sechs Transportaufgaben mehrere Minuten, da sich daraus knapp 480 Millionen Möglichkeiten ergeben. Bei einer Prozessorgeschwindigkeit von 4 GHz und geschätzten 1000 Prozessorzyklen pro Möglichkeit würden zwei Minuten nur für die Planung benötigt. Dieser Wert steigt exponentiell mit der Anzahl der Transportaufgaben, weshalb eine effiziente Algorithmitik entscheidend für die Nutzbarkeit eines Taskplanners (dt: Aufgabenplaner) ist. Andernfalls geht wichtige Wettkampfzeit verloren, die dem Roboter dann nicht mehr für das eigentliche Absolvieren der Aufgaben zur Verfügung steht.

Um eine Aufgabenliste abzuarbeiten, muss der Roboter wissen, welche Schritte hierfür nötig sind. Dafür muss jeweils der aktuelle Zustand und die zu erledigende Aufgabe korrekt interpretiert werden, sodass Aktionen wie z.B. die Navigation zur Zielworkstation, das Abscannen der Tischoberfläche und das Greifen des Objekts in der richtigen Reihenfolge ausgeführt werden. Außerdem muss berücksichtigt werden, ob sich der Roboter z.B. bereits an der richtigen Workstation befindet und deshalb nicht gefahren werden muss. Es sollte also zusätzlich zu dem globalen Aufgabenplan (Was?), der die Reihenfolge der Transportaufgaben verwaltet, einen Aktionsplan geben, in die dem für diese Aufgaben benötigten Aktionen vermerkt sind (Wie?).

Damit diese vorgemerkteten Aktionen auch tatsächlich in die Tat umgesetzt werden können, muss eine Verbindung zu allen benötigten Roboterkomponenten hergestellt werden, damit z.B. eine Armbewegung ausgelöst werden kann. Dafür müssen Schnittstellen zu den entsprechenden Treibern und Programmen implementiert werden, auf die dann in den jeweiligen Fällen zugegriffen wird. Außerdem muss eine Feedbackschleife zum Taskplanner bestehen, über die der (Miss-)Erfolg der Aufgabe zurückgemeldet werden kann, damit gegebenenfalls reagiert und umgeplant werden kann.

3 Ohmnibot als Entwicklungsroboter

Als Testroboter für die Architektur und die Softwarepakete wird der vom Team AutonOHM entwickelte Ohmnibot verwendet. Dieser wurde in den vergangenen zwei Jahren für den Einsatz in der @Work-Liga optimiert. Da sowohl dessen Hardware als auch die Inbetriebnahme im letzten Forschungsbericht [4] detailliert behandelt wurden, soll an dieser Stelle lediglich ein kurzer Überblick über die verfügbaren Komponenten gegeben werden. Dabei sind z.B. Eigenschaften wie die kinematischen Modelle der Antriebe oder die Sensortypen wichtig, da die Einbindung und Verwendung dieser Komponenten später entscheidend für das Roboterverhalten sind.



Abbildung 3.1: Ohmnibot - Aktueller Stand

3.1 Hauptrechner

Als Hauptrecheneinheit ist ein leistungsstarker Desktop-Prozessor von AMD (Modell Ryzen 3700X) verbaut. Dieser bietet mit seinen acht physischen Kernen eine großzügige Grundlage für die teilweise extrem rechenintensive Algorithmik bei vergleichsweise geringem Stromverbrauch. Für einen möglichst geringen Platzverbrauch ist er auf einem mini-ATX Mainboard angebracht, an das eine passiv versorgte Nvidia GTX 1650 Grafikkarte von MSI angebunden ist. Diese kann für den effizienten Betrieb von neuronalen Netzen genutzt werden.

Die Komponenten sind in ein eigens dafür angefertigtes Gehäuse eingebaut, welches aufgrund des flachen Formfaktors ähnlich wie bei einem Server-Rack in den Roboter

geschoben werden kann und dadurch zwar im Roboter geschützt, aber dennoch relativ einfach zu warten ist.



Abbildung 3.2: Hauptrechner im Ohmnibot

Der PC wird über ein LKW-Netzteil versorgt, das eigentlich für die Montage von Computern in z.B. Führerhäusern konzipiert ist. Das Netzteil ist daher auf den Betrieb mit Batteriespannung statt haushaltsüblichem Wechselstromanschluss ausgelegt und somit gut für den Einsatz auf dem Roboter geeignet.

Als Betriebssystem wird eine Ubuntu 18.04 LTS Distribution verwendet, die einfachen und direkten Zugriff auf Hardwarekomponenten ermöglicht. Außerdem kann dadurch das Robot Operating System (ROS) verwendet werden, welches die Grundlage für die Softwarearchitektur bildet.

3.2 Antrieb

Die Grundplattform (Evorobot R&D) ist mit einem Mecanum-Antrieb ausgestattet und kann sich dadurch omnidirektional auf flachem Untergrund fortbewegen. Durch die Kraftübertragung von den Motoren auf die Räder mittels Zahnrädern und -riemen ist die maximale Geschwindigkeit bzw. die maximal bewegbare Last variabel und kann für den Einsatzzweck angepasst werden. Die hier verwendete Standardkonfiguration ist auf 1 m/s und 120 kg ausgelegt.

Die Motoren werden mittels Motorshields von Evocortex gesteuert, welche in Kombination mit den verbauten Encodern entweder auf Position oder Geschwindigkeit regeln können. Die Shields werden über eine CAN-Schnittstelle angesprochen, über die Kommandos geschickt, Drehwerte empfangen und außerdem die Regelparameter eingestellt werden können.

3.3 Manipulator

Der Manipulator ist eine Eigenentwicklung des Teams. Der Roboterarm kombiniert verschiedene Ansätze der Industrie (SCARA, Gelenkarm) zu einem Kinematikkonzept, welches für das Manipulieren von Objekten auf einer Tischoberfläche optimiert ist. Durch den Einsatz von zwei Linearantrieben für die beiden Hauptachsen (Z und X) ist das Bewegungskonzept auch für unerfahrene Entwickler leicht verständlich. Dabei realisiert es einen variablen Arbeitsraum (Längenanpassung der Linearantriebe), ohne dass sich dabei die Leistungsanforderungen der Antriebsmotoren ändern würden. Dies ist bei klassischen Gelenkarmen nicht der Fall, da bei diesen die Hebelwirkung bei größeren Abständen das erforderliche Leistungsniveau beeinflusst. Durch die konstante Kraftübertragung aufgrund der festen Übersetzung zwischen Motorzahnrad und Linearstange ist die Kraft, die z.B. für das Anheben einer Last benötigt wird, stets konstant. Dies vereinfacht die Regelung des Arms im autonomen Betrieb, da die Bahnberechnungen weniger komplex werden. Dadurch kann die aktuelle Konfiguration einfacher verwendet werden, mit der es möglich ist, ohne zusätzliches Neupositionieren der Plattform alle Bereiche einer Workstation zu erreichen.



Abbildung 3.3: Ausgefahrener Roboterarm



Abbildung 3.4: Kippende Greiffinger

Der Greifer realisiert eine anpassbare Fingerbahn bei größtenteils konstanter Kraftübertragung. Auch hier wird ein Linearantrieb aus Motorzahnrad und in diesem Fall doppelter Zahnstange verwendet, um die Finger anzutreiben. Die Öffnungsweite des Greifers beträgt knapp 6 cm für den parallelen Bahnabschnitt und erweitert sich durch den entwickelten Klappmechanismus [6] (siehe Abb. 3.4) auf bis zu 14 cm im maximal geöffneten Zustand. Dadurch ist die Toleranz der Greifposition erhöht, bei der immer noch zuverlässig gegriffen werden kann. Aufgrund der konstanten Kraftübertragung zwischen Fingern und Motor kann außerdem die Griffkraft eingestellt werden. Mittels Motorfeedback kann außerdem darauf zurückgeschlossen werden, ob ein Objekt tatsächlich gegriffen wurde.

3.4 Sensorik

3.4.1 Laserscanner

Für die Lokalisierung und Kartierung stehen an der Grundplattform insgesamt drei Laserscanner zur Verfügung, die einen kombinierten Abtastbereich von 360° aufweisen. Durch die Überlappung und gegenseitige Ergänzung der einzelnen Scans existieren, bis auf den unmittelbaren Nahbereich der Räder, keine toten Winkel. So können stets alle physischen Hindernisse erkannt werden. Außerdem stehen so die maximal verfügbaren Rauminformationen zur Verfügung, die für die Lokalisierung genutzt werden können.



Abbildung 3.5: Ohmnibot Unterboden

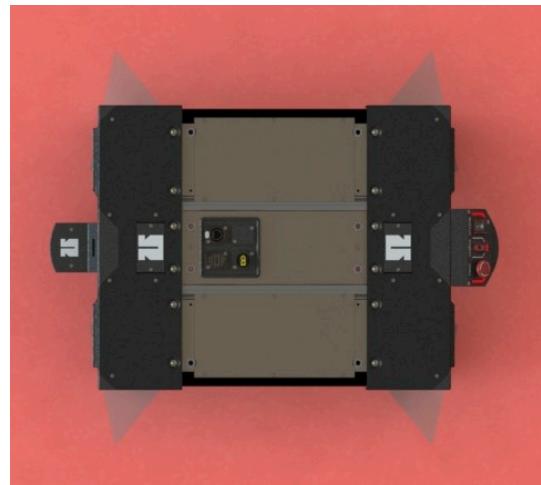


Abbildung 3.6: Laserscanbereich

Die Sensoren sind so angebracht, dass sie die Umgebung in einer Höhe von 8 cm über dem Boden erfassen. Dadurch kann ein Großteil der Arenaelemente mit ihnen erfasst werden, wodurch ein sauberes Abbild der Arena erstellt werden kann. Für die 0 cm und 5 cm können im Kartierungsprozess z.B. zusätzliche Schachteln angebracht werden, die währenddessen zur Umrissgenerierung dienen.

3.4.2 Hinderniskameras

Aktuell wird an einer Methode geforscht, bei der die Bilder von mehreren Fisheye-Kameras zu einem 360° Umgebungsscan fusioniert werden sollen, in dem dann z.B. nach Barriertape gesucht wird. Diese sind zwar noch nicht auf dem realen Roboter montiert, die Technik konnte aber bereits in der Simulation getestet und verifiziert werden, weshalb aktuell die optimalen Montagepunkte erprobt werden, an denen der erfasste Bereich am größten ist.

Bis zum Abschluss dieser Forschungsarbeiten wird eine einzelne Kamera verwendet, die auf dem vorderen Laserscanner in Fahrtrichtung montiert ist.

3.4.3 3D-Kamera

Am Roboterarm ist eine RGB-D Kamera angebracht, die zusätzlich zum Farbbild ein Tiefenbild der Szene aufnimmt. Beide Bilder können zu einer Farbpunktwolke fusioniert werden, welche zur Workstation- und Objekterkennung genutzt werden kann. Die Kamera ist so ausgerichtet, dass sie stets Orthogonal zum Boden bzw. zu einer Tischoberfläche steht. Außerdem ist sie so angebracht, dass der Mindestabstand, der für die fehlerfreie Funktion des Tiefensensors (Infrarot-Stereo) benötigt wird, auch bei der höchsten Workstation noch erreicht werden kann.

3.5 Beleuchtung

Um die 3D-Kamera ist eine steuerbare LED-Beleuchtung montiert (Abb. 3.4), die verschiedene Lichtverhältnisse erzeugen kann. So kann einerseits eine womöglich schwache Raumbeleuchtung ausgeglichen und außerdem eine „Standardbeleuchtung“ eingestellt werden, bei der die Objekterkennung erfahrungsgemäß am besten funktioniert.

Außerdem sind RGB-Streifen an der Unterseite der Grundplattform angebracht, die als HMI für Zuschauer genutzt werden können (siehe 3.7). Mittels verschiedener Blinkmodi kann der aktuelle Zustand des Roboters oder die nächste geplante Aktion signalisiert werden. Dies kann vor allem im vollautonomen Betrieb von Vorteil sein, da die Entwickler so leichter verstehen können, in welchem Abschnitt eventuelle Fehler passieren.



Abbildung 3.7: Beispielmodi der Unterbodenbeleuchtung

3.6 Testumgebung

Damit auch abseits der Wettkämpfe mit einer wettkampfähnlichen Arena getestet werden kann, wurden Workstations und Wände gefertigt und daraus ein Parcours errichtet. Das Layout ist relativ kompakt, da nicht unbegrenzt Fläche zur Verfügung steht. Dennoch können alle entscheidenden Faktoren des Wettkampfs getestet werden. Diese beinhalten z.B. das Blockieren von einzelnen Passagen, der Einsatz verschiedener Tischhöhen und das Greifen bewegter Objekte am Rotating Table.



Abbildung 3.8: Wettkampfarena im Labor

3.7 Simulation

Aufgrund der Kooperation mit Evocortex stehen dem Team die CAD-Daten der Grundplattform zur Verfügung. Da die eigenen Modifikationen in das Grundmodell integriert wurden, existiert ein aktuelles 3D-Modell des Roboters, welches für Simulationszwecke genutzt werden kann. Hier können nicht nur Sensorpositionen schneller und einfacher erprobt werden, sondern auch Softwarepakete getestet werden, ohne dass der eigentliche Roboter bei Fehlern beschädigt wird. In Abbildung 3.10 schweben beispielsweise sechs Kameras (weiß) über dem Roboter, mit denen die optische Hinderniserkennung erprobt wird.

In der Simulation ist es zudem möglich, die Wettkampfarenen vergangener Robocup nachzubilden und damit zu testen, ohne dass deren Platzanforderungen in der Realität verfügbar sein müssen (Abb. 3.9). Außerdem können verschiedene Arenen gespeichert und geladen werden, sodass Spezialfälle einfacher erprobt werden können.

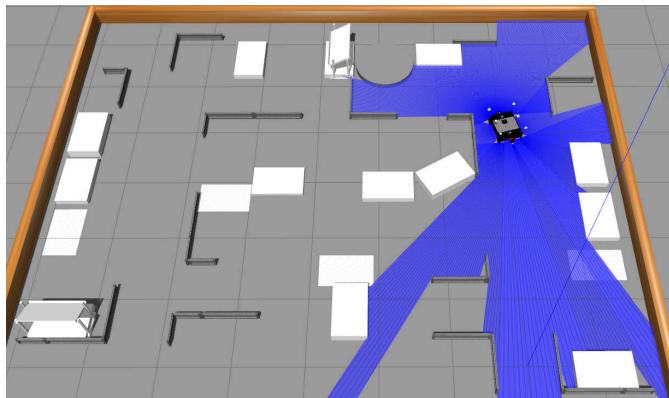


Abbildung 3.9: Simulation der Arena aus Sydney 2019

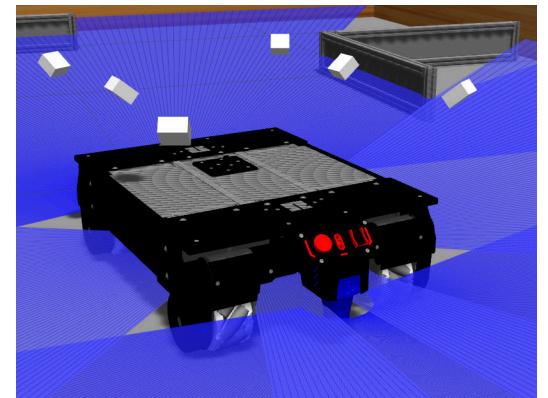


Abbildung 3.10: Simulierte Sensorik

Im Simulationsmodell fehlt aktuell der selbst entwickelte Roboterarm, dessen CAD-Modell erst noch aktualisiert werden muss, bevor er in die Simulation integriert werden kann.

4 Entwurf der Softwarekontrollarchitektur

Große Softwareprojekte bedürfen einer Architektur, die sowohl übersichtlich als auch wartbar ist [7]. Außerdem sollte sie übertragbar sein, wodurch sichergestellt ist, dass eine mögliche Lösung nicht nur ein Nischenproblem abdeckt. Daher bildet eine sorgfältig dimensionierte Architektur die Grundlage für ein erfolgreiches Projekt.

In der Robotik ähneln sich die Aufgabengebiete der einzelnen Softwarekomponenten häufig. Es müssen Sensoren eingebunden und ausgewertet und die Ergebnisse dann so verarbeitet werden, dass die Aktorik dementsprechend angesteuert werden kann. Je mehr Funktionen ein Roboter dabei umsetzen soll, desto mehr Algorithmen werden zwischen Sensorik und Aktorik eingesetzt. Bei einem Forschungsprojekt wie dem OhmniBot ist es außerdem durch Änderungen im Regelwerk immer möglich, dass sich die Anforderungen an ein System relativ kurzfristig ändern. Damit der Entwicklungsstand dabei nicht verloren geht, muss die Architektur offen für Veränderungen bleiben, da sonst alle Komponenten überarbeitet werden müssen.

Dafür muss zunächst die generelle Strategie festgelegt werden. Da es durch die Verwendung verschiedener Hardwarekomponenten vorkommen kann, dass einzelne Produkte durch neuere oder leistungsstärkere ersetzt werden sollen, die dann womöglich andere Treiber einsetzen, darf das Ziel nicht eine einzelne, riesige Applikation sein. Stattdessen sollten einzelne Funktionen jeweils in einem eigenen Programm umgesetzt sein, das mit anderen Programmen kommunizieren kann. Wenn die Schnittstellen zwischen zusammengehörigen Programminstanzen so definiert werden, dass der Datenaustausch allgemein gültig ist, können z.B. die Treiber verschiedener Laserscanner untereinander getauscht werden, solange beide einen Laserscan für weitere Programme zur Verfügung stellen.

Ein weiterer Vorteil von dedizierten Programmen ist, dass nicht alle Softwarekomponenten im selben Prozess laufen, sondern vom Betriebssystem als jeweils eigener Prozess verwaltet werden können. Dadurch können sie parallel ausgeführt werden, wodurch die Rechenlast auf alle Prozessorkerne verteilt werden kann. Da jeder Prozess jedoch einen eigenen Speicherbereich zugewiesen bekommt, auf den andere Prozesse nicht ohne weiteres zugreifen können, muss jedes Programm eine Form der Interprozesskommunikation implementieren, damit die Daten global verfügbar sind.

Diese kann entweder mittels Shared Memory gelöst werden, wobei die Kommunikation durch die Nutzung eines gemeinsamen Datenspeichers implementiert ist (z.B. den Zugriff auf gemeinsame Arbeitsspeicherbereiche), oder durch sogenannte Message Queues.

Bei diesen „werden Nachrichten von einem Prozess an eine Nachrichtenschlange geschickt, von wo diese von einem anderen Prozess abgeholt werden können,“ [8]. Da es in der Robotik vorkommen kann, dass viele Prozesse auf dieselben Sensordaten warten, und es dadurch bei geschütztem Zugriff auf gemeinsamen Speicher zu einer Deadlock-Situation kommen kann, bei der mehrere Prozesse auf die Daten warten und daher den weiteren Programmablauf blockieren, sind Message Queues vor allem für die offene Architektur eines

Forschungsprojekts besser geeignet.

Diese werden im Robot Operating System (ROS) mittels Sockets umgesetzt. ROS ist ein Framework, welches überwiegend von forschenden Entwicklern für die Programmierung von Robotern eingesetzt wird. Es ist Open-Source und kann auf einem beliebigen Linux-Computer verwendet werden. Programme werden darin als sogenannte „Nodes“ gestartet und können dann über „Topics“ miteinander kommunizieren. Die Topics definieren dabei den Socketnamen und den Messagetypen, der über dieses Topic versendet oder empfangen werden kann. Eine „Message“ (dt: Nachricht) kann dabei aus mehreren Variablen bestehen, beispielsweise einem Zeitstempel und einer Distanzmessung.

ROS definiert eine Vielzahl von Standardnachrichten, durch deren Benutzung die Austauschbarkeit von Nodes gewährleistet wird. Außerdem werden wichtige Grundregeln wie die verwendeten Einheiten oder die Koordinatenachsen im Raum festgelegt, wodurch die Verwendung von öffentlich zugänglichen Softwarepaketen erleichtert wird. Die Community stellt dabei z.B. eigene Ansätze für Lokalisierungsalgorithmen, Navigation oder generische Kameratreiber zur Verfügung, die in eigenen Projekten eingesetzt werden können. Dadurch können viele anfängliche Hürden schneller überwunden werden, da nicht jede benötigte Komponente selbst entwickelt werden muss.

Damit die fremden Pakete an das eigene Projekt angepasst werden können, sind die meisten Nodes über den Parameterserver konfigurierbar. In diesen können z.B. mittels Kommandozeilenbefehl eigene Parameter eingespeichert werden, welche beim Start eines Nodes abgefragt und in das Programm integriert werden. So können nicht nur Topics umbenannt, sondern auch Robotereigenschaften wie die Radgröße oder die Spurweite eingestellt werden. Damit ein Node nicht vor jedem Start aufwändig händisch per Kommandozeile konfiguriert werden muss, können in ROS sogenannte Launchfiles erstellt werden. Dabei handelt es sich um Startskripte im XML-Format, in denen eine beliebige Anzahl von Nodes gestartet und Parameter übergeben werden können. Solche Launchfiles können außerdem wiederum andere Launchfiles integrieren, sodass z.B. ein Startskript für den gesamten Roboter erstellt werden kann. Dies wird mit der steigenden Anzahl von insgesamt verwendeten Nodes immer relevanter, da sonst allein der Startprozess mehrere Minuten dauern könnte. Mithilfe der Launchfiles ist es daher möglich, mit einem einzigen Befehl einen vordefinierten Softwarezustand herzustellen.

ROS bildet durch diese Funktionalitäten die Grundlage einer flexible Softwarearchitektur für Roboter und wurde deshalb bereits auf dem alten System verwendet. Durch das stetige Wachstum über die Saisons sind jedoch die Struktur und somit der Überblick verloren gegangen. Daher soll in dieser Arbeit eine Kontrollarchitektur entwickelt werden, in die die verschiedenen Nodes eingeordnet werden können. Diese beinhaltet eine Funktionshierarchie, mithilfe derer eindeutig festgelegt wird, auf welchem logischen Level ein Programm bzw. Algorithmik arbeitet. In Abbildung 4.1 ist der Entwurf einer BCMD-Architektur zu sehen. Dabei steht das Akronym BCMD für die Anfangsbuchstaben der Funktionsebenen Brain, Control, Model und Driver (dt: Gehirn, Kontrolle, Modell und Treiber).

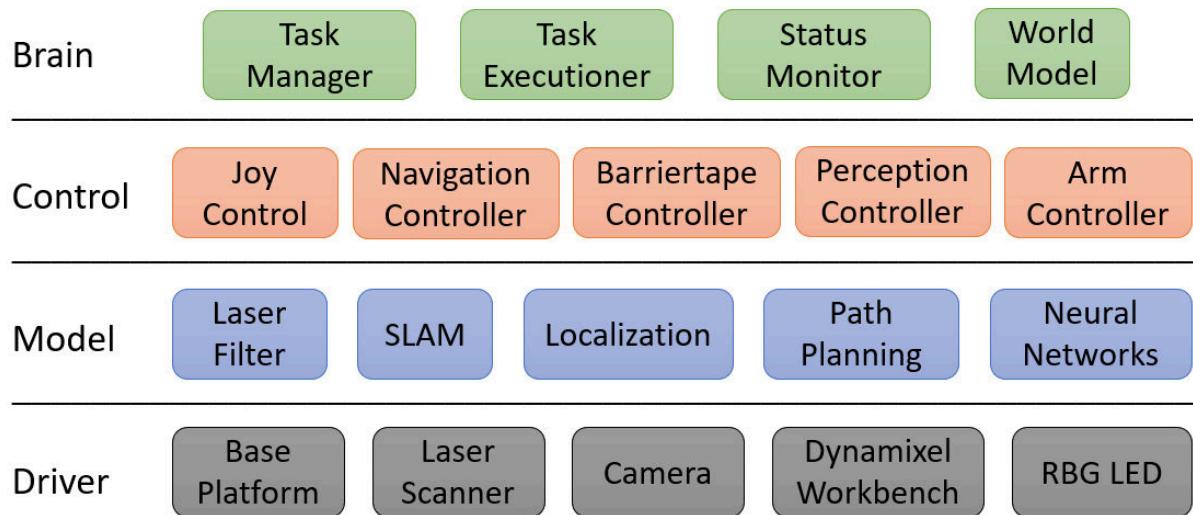


Abbildung 4.1: Softwararchitektur - BCMD

Die Architektur ist an das Model-View-Controller-Designpattern (MVC) aus der Anwendungsprogrammierung angelehnt, in dem die Algorithmik (Model) von der grafischen Oberfläche (View) getrennt ist und über einen verbindenden Controller (Kontrolle) gesteuert werden kann. Die Idee dabei ist, dass sowohl View als auch das Model getauscht werden können, ohne dass die jeweils andere Komponente angepasst werden muss.

In der BCMD-Architektur wird dieser Gedanke aufgegriffen und für die Robotikprogrammierung erweitert. So beinhaltet die Driver-Ebene alle Programme, die für die Einbindung von Hardware genutzt werden. Bei gleichbleibenden Schnittstellen können die individuellen Sensoren bzw. die dazugehörigen Treiber einfach ersetzt werden. Außerdem kann die komplette Ebene getauscht werden, wenn der Roboter nur in der Simulation betrieben werden soll. Dann ersetzen virtuelle Sensoren und Treiber die reale Hardware.

Die Model-Ebene wird aus dem MVC Pattern übernommen, da auch in der Robotik komplexe Algorithmik eingesetzt wird. Die Modelle beinhalten dabei die Lösungen für die in Kapitel 2 beschriebenen Problemstellungen, mit Ausnahme der Aufgabenbehandlung. Diese wird in der Brain-Ebene gelöst, in der nicht nur Aufgaben entgegengenommen und umgeplant werden können, sondern auch der Zustand des Roboters und der Umgebung gespeichert und verwaltet wird.

Zwischen Brain und Model ist außerdem die Control-Ebene. Da es für viele Programme oder Algorithmen vorteilhaft sein kann, wenn für deren Benutzung einfache Interfaces bereitgestellt werden, werden alle Kontrollprogramme hier eingeordnet. Dabei kann das Ziel der Interfaces variieren, wobei das Hauptziel ist, einem übergeordneten Programm aus der Brain-Ebene eine simple Bedienoberfläche zu bieten, damit keine Detailinformationen über z.B. die Armsteuerung im generellen Programmablauf für eine Aufgabenausführung implementiert werden müssen.

5 Hardwareanbindung: Driver-Ebene

Die Treiber für die auf dem Roboter verbaute Hardware wurden bereits im letzten Projektbericht beschrieben. Da die bereitgestellten Topics jedoch von übergeordneten Programmen weiterverwendet werden, werden im Folgenden die Konfiguration des Roboters und die daraus resultierenden Interfaces erläutert. Außerdem werden jeweils die ersetzenen Simulationstreiber erwähnt, falls sie für die Komponente verfügbar sind.

5.1 Laserscanner

Für die drei Laserscanner wird jeweils eine Instanz des „sick_tim551_2050001“ [9] ROS Treibers verwendet. Dieser kommuniziert über die Netzwerkschnittstelle mit den Sensoren und liest deren Messwerte aus, welche im Treiber mit einem Zeitstempel versehen und anschließend als ROS Message gepubliziert werden. Die Laserdaten werden zwar von der Auswertelektronik vorgefiltert, es sind also keine groben Fehler in den Messungen enthalten, jedoch müssen vor der Verarbeitung der Daten noch Filter zwischengeschaltet werden. Damit die Topics und die darin versendeten Daten eindeutig identifiziert werden können, werden die Informationen in den Topicnamen geschrieben. Daraus ergeben sich die folgenden Nodes (blaue Rechtecke) und Topics (grüne Kreise) mit den Messagetypen (lila Pfeile):

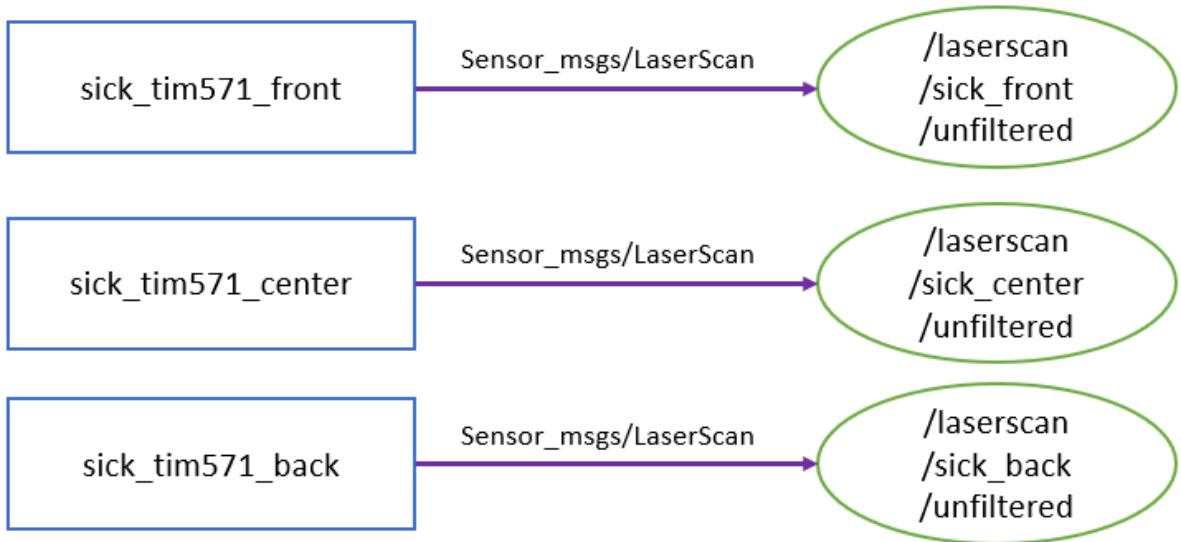


Abbildung 5.1: Treiber für die Laserscanner

Für jeden Treiber existiert ein eigenes Launchfile, welches die IP-Adresse, den Scanbereich und den Topicnamen festlegt. Alle drei Launchfiles werden in „laser_complete.launch“ aufgerufen, wodurch auch die drei Scanner gestartet werden.

In der Simulation werden die realen Treiber durch das Gazebo-Plugin „gpu_ray“ ersetzt, welches mittels einem über Parameter einstellbaren Sensormodell und Raytracing die

Distanzwerte in der virtuellen Umgebung errechnet. Das Plugin wird mit den Spezifikationen der realen Laser parametriert und die Messwerte durch die Erweiterung für ROS „libgazebo_ros_gpu_laser.so“ [10] veröffentlicht. Hier kann auch der Topicname angegeben werden, sodass aus der Simulation exakt dieselben Topics bereitgestellt werden können. Das Messrauschen durch verschiedene Reflexionen realer Materialien kann zwar nicht exakt nachgebildet werden, die Daten sind jedoch für Tests der Softwarestruktur und der Algorithmik unter optimalen Bedingungen ausreichend.

5.2 Hinderniskamera

Zum aktuellen Zeitpunkt existiert zwar keine realisierte Hardwarelösung für die kombinierten Fisheye-Kameras am Roboter, es wurden aber bereits Treiber für die USB Kameras getestet, z.B. der „usb_cam“ Treiber [11]. Dieser greift über die Gerätedatei der Kamera auf die USB Schnittstelle zu und startet einen Bilderstream im angegebenen Format. Da es sich hierbei um einen generischen Treiber handelt, müssen die Informationen über den jeweiligen Kameratyp, die über das Topic „camera_info“ veröffentlicht werden sollen, selbst eingestellt werden. Darin enthalten sind z.B. die Linseneigenschaften und das Verzerrungsmodell. Diese können mit dem „camera_calibration“ Tool aus dem Paket „image_pipeline“ [12] ermittelt werden, falls sie nicht durch den Hersteller angegeben sind. Die eingestellten Werte müssen möglichst akkurat sein, da sonst Fehler bei der Pixelprojektion (siehe 6.5) entstehen können. Daher muss die Kalibrierung sorgfältig durchgeführt und überprüft werden. Im Betrieb veröffentlicht der Treiber über zwei separate Topics die Kamerainformationen und die Farbbilder:

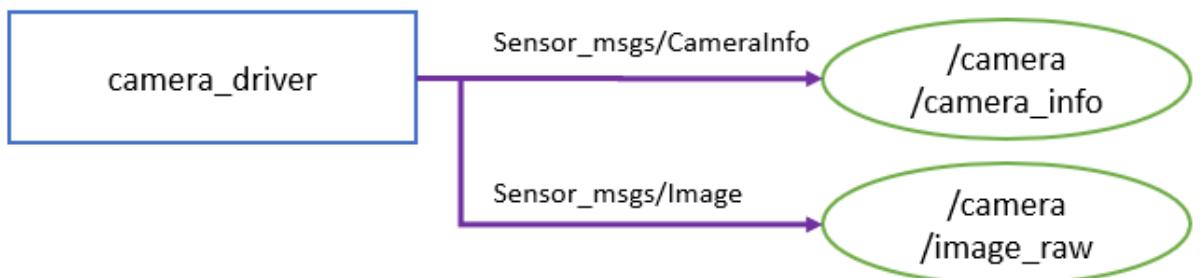


Abbildung 5.2: Treiber für eine USB Kamera

Über ein Launchfile können sowohl dem Node als auch den Topics eindeutige Namen zugewiesen werden. Bei der Einbindung von sechs Kameras können diese Namen den Montagepunkt beinhalten, z.B. „cam_front_left“. In der Simulation konnte dieses Szenario bereits getestet werden (siehe Abb. 3.10). Dafür werden über das Gazebo Plugin „libgazebo_ros_camera.so“ [10] sechs virtuelle Sensoren in die Simulationsumgebung geladen. Auch dieses Sensormodell kann so konfiguriert werden, dass es eine realen Kamera nachbildet. Es müssen lediglich die Kamerainfos und das gewünschte Bildformat als Parameter übergeben werden. Die Simulation generiert dann entsprechende Bilder der Umgebung

und versendet diese über dieselben Schnittstellen wie der Hardwaretreiber in Abb. 5.2. Die Bildqualität der gerenderten Szene ist jedoch nicht sehr realitätsnah (siehe Abb. 5.4). Daher können die simulierten Bilder nur bedingt zum Test der neuronalen Netze genutzt werden.

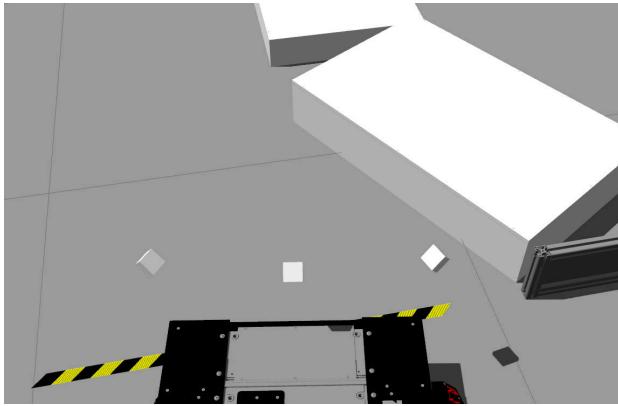


Abbildung 5.3: Szene in der Simulation

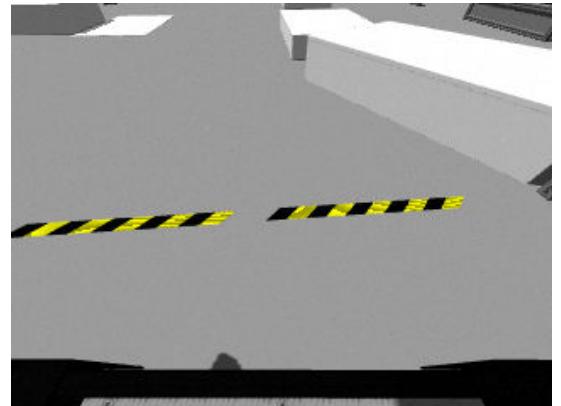


Abbildung 5.4: Generiertes Kamerabild

5.3 3D-Kamera

Die Intel Realsense D435 wird über den dazugehörigen ROS Treiber „realsense2_camera“ [13] ins System eingebunden. Hier können die gewünschte Auflösung und Bildwiederholfrequenz für den Farb- und Tiefensensor eingestellt werden. Außerdem kann die Punktwolgengenerierung aktiviert werden, wodurch Farb- und Tiefenbild zu einer 3D-Farbpunktewolke fusioniert werden. Dabei muss der Parameter „align_depth“ auf „true“ gesetzt werden, da sonst der Versatz der beiden Bilder, der sich aus der Sensoranordnung ergibt, nicht kompensiert wird. Wenn die Daten nicht aufeinander abgestimmt werden, entsprechen die Pixelindizes des einen Bildes nicht den dazugehörigen Pixeln im zweiten Bild, was zusätzlichen Aufwand in der Auswertung erfordern würde. Der Versatz wird zwar zusätzlich zu den Topics auch über den ROS Transformation Server (orange) veröffentlicht, was den Zugriff auf diese Information in anderen Nodes ermöglicht, es ist jedoch trotzdem einfacher, diesen Schritt direkt im Treiber zu erledigen. Es ergeben sich die in Abb. 5.5 dargestellten Interfaces.

Ähnlich wie die normale Farbkamera kann auch die 3D-Kamera simuliert werden. Mit dem Plugin „libgazebo_ros_openni_kinect.so“ wird das virtuelle Sensormodell einer Microsoft Kinect Kamera erstellt, welches mittels Raytracing ein Tiefenbild bzw. die Farbpunktewolke generiert wird. Die Tiefendaten sind in der Simulation fehlerfrei und können damit für Tests der Workstationerkennung unter optimalen Bedingungen genutzt werden. Da die Bilderqualität aufgrund der unrealistischen Grafik der Simulation zu wenig Details für eine realitätsnahe Erkennungsalgorithmitik bietet, müssen die Bilder, die während eines Tests in der Simulation verwendet werden sollen, anderweitig generiert werden, z.B. durch das Laden von realen Kamerabildern aus einer Datei.

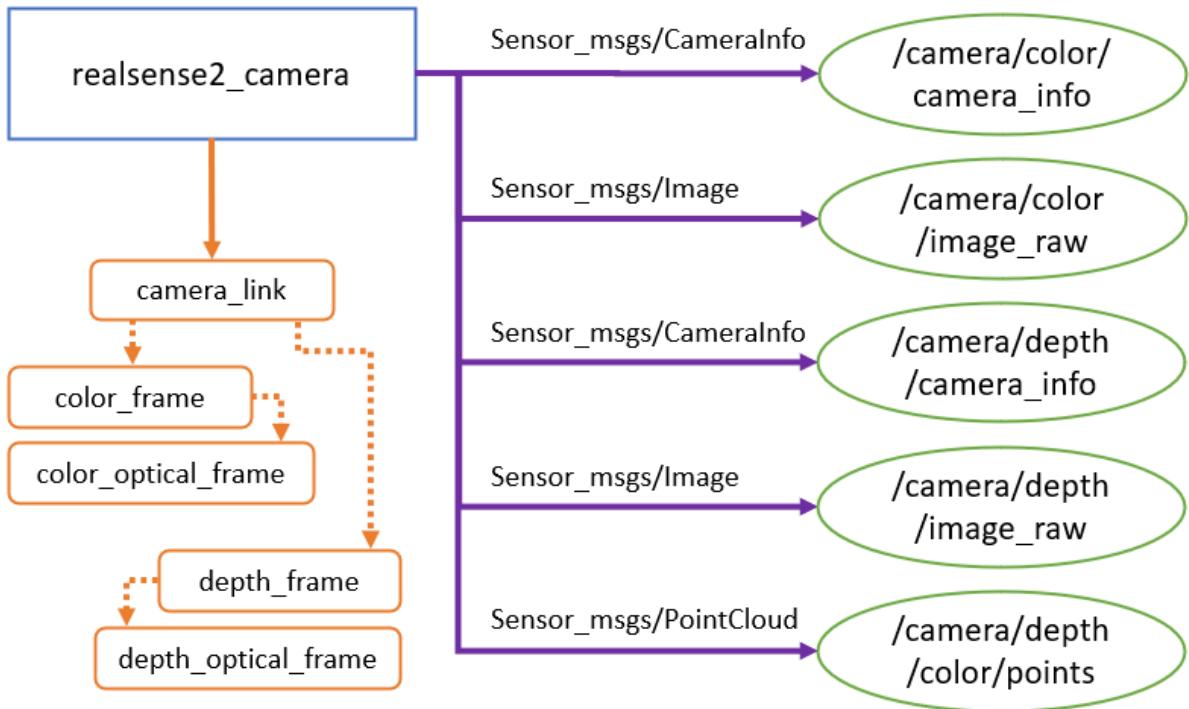


Abbildung 5.5: Treiber für die 3D-Kamera

5.4 Dynamixel Motoren

Dynamixel Motoren können über einen USB2Dynamixel-Converter an einen PC angebunden werden. Der Treiber „dynamixel_workbench_controllers“ [14] schreibt und liest über die Gerätedatei auf den Dynamixel Bus. So können die einzelnen Motoren angesprochen und gesteuert werden. Jeder Motor besitzt dafür eine eigene Bus-ID, die über das Launchfile des Treibers angegeben wird. Den IDs wird jeweils eine Motorbezeichnung zugewiesen, die auch für die Referenzierung in den Topics „trajectory“ und „joint_states“ verwendet werden.

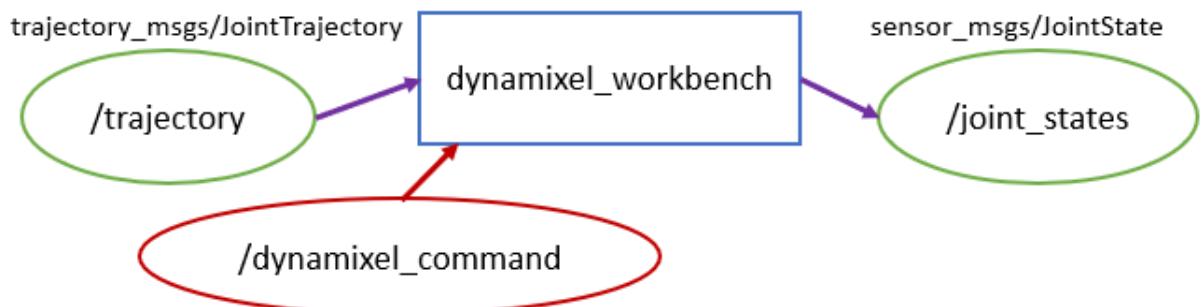


Abbildung 5.6: Treiber für die Dynamixel Motoren

Über die Eingangstrajektorie können den Motoren verschiedene Positionen vorgegeben werden, die der Treiber dann jeweils anfährt. Da die Positionen im Motorsystem angegeben werden, also keine Informationen über den Montagepunkt beinhalten, muss die

Umrechnung zwischen Armkoordinaten und Motorstellungen in dem Node erfolgen, der die Trajektorie versendet. Außerdem müssen für synchrone und flüssige Armbewegungen die Geschwindigkeits- und Beschleunigungswerte der Zwischenpunkte selbst angegeben werden, da der Treiber sonst jede Position Punkt-zu-Punkt anfährt, wobei er an jeder Position zum Stillstand kommt.

Die ausgehenden „joint states“ (dt. Gelenkzustände) beschreiben für jeden Motor die aktuelle Position und Geschwindigkeit. Da diese Werte während der Ausführung einer Trajektorie nicht kontinuierlich aktualisiert werden, sondern nur an den Zwischenpositionen, können einzelne Motoren auch direkt mittels ROS Service (rot) gesteuert werden. Ein Service ist im Gegensatz zu Topics keine einfache Message Queue, sondern agiert als eine Art globale Funktion, auf die zur Laufzeit aus der Kommandozeile oder einem anderen Userspaceprogramm zugegriffen werden kann. Dies macht vor allem für kurze Aktionen Sinn, da ein Serviceaufruf den normalen Programmablauf eines Nodes blockiert. Im Treiber für die Dynamixelmotoren kann über den Service „dynamixel_command“ eine Direktnachricht für einen Motor formuliert werden. Diese beinhaltet die ID des Zielmotors und die gewünschten Stellgrößen, welche über den Dynamixel Bus übertragen werden und eine kontinuierliche Rückmeldung der Statusinformationen über Position und Last auslösen.

Sowohl der Greifer als auch der Towerarm setzen Dynamixel Motoren ein. Daher werden für beide Aktoren jeweils ein USB-Converter und ein Treiber benötigt, wenn die logische Trennung beibehalten werden soll. Aufgrund der Implementierung des Treibers müssen nämlich in den Messages, die über die Topics verschickt werden, immer alle Motoren behandelt werden. Es wäre demnach bei nur einem Treiber für Greifer und Arm nicht möglich, nur eine Komponente zu steuern, ohne die Werte für die andere auch zu setzen bzw. zu lesen. Daher werden zwei Treiber mit den Präfixen „Parallelgripper“ und „Towerarm“ eingesetzt.

Eine Simulation der Motoren macht aufgrund der unrealistischen Physik in Gazebo keinen Sinn. Vorallem weil sie nicht direkt als Gelenk agieren, sondern noch zusätzliche Mechanik verbaut ist. Diese müsste aufwendig nachgebildet werden, was mehr Aufwand als Nutzen bietet. Um in der Simulation dennoch Armbewegungen nachzubilden zu können, kann ein Modell des Arms aus Lineargelenken nachgebaut werden. Dies ist jedoch aktuell noch nicht gelöst, weshalb es nicht weiter behandelt wird.

5.5 Grundplattform

Zur Steuerung der Grundplattform wird der Treiber von Evocortex verwendet. Dieser schreibt und liest über den CAN2USB-Adapter auf den CAN Bus und kommuniziert so mit den Motorshields, an die die Motoren und Encoder angeschlossen sind. Im Node „evo_base_controller_ros“ [15] wird über Parameterdateien eine Roboterkonfiguration eingelesen und daraus ein kinematisches Modell für den Antrieb erstellt. Dieses wird dafür verwendet,

die Zielgeschwindigkeiten für die Plattform, die über das Topic „cmd_vel“ empfangen werden können, in Motorstellgrößen umzuwandeln, die die gewünschte Bewegung umsetzen. Außerdem kann mit den Encoderdaten die Roboterodometrie errechnet werden, welche als Transformation und über das Topic „odom“ veröffentlicht wird. In der Message ist die aktuelle Geschwindigkeit und die geschätzte Roboterposition enthalten, die auf Grundlage der addierten Teilstrecken gebildet werden kann. Da diese Schätzung jedoch aufgrund von z.B. Radschlupf driftet, kann mit dem Service „reset_odom“ die Odometrieposition zurückgesetzt werden.

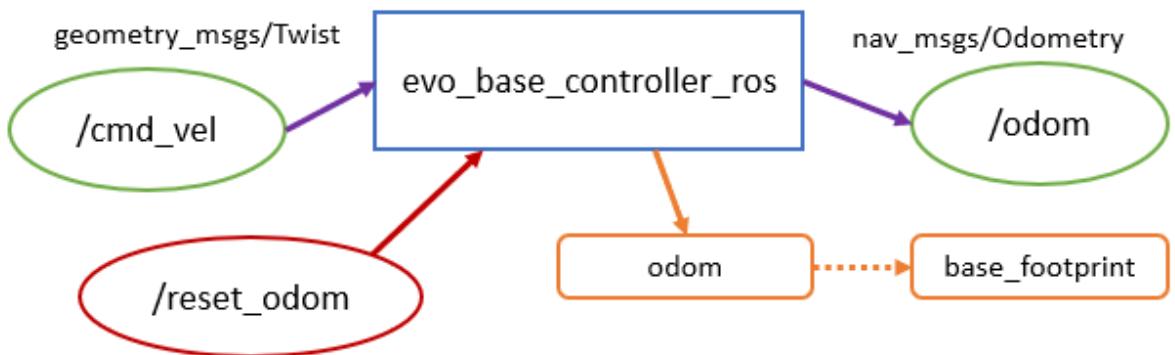


Abbildung 5.7: Treiber für die Grundplattform

Da es für eine realitätsnahe Simulation nötig wäre, die physikalischen Eigenschaften der Räder zu ermitteln und deren Reibung am Boden zu simulieren, wird eine deutlich vereinfachte Variante umgesetzt. Über das „Planar Move“ -Plugin von Gazebo kann ein Körper zweidimensional im Raum bewegt werden. Dafür wird ebenfalls ein „cmd_vel“ Topic abonniert, welches die Zielgeschwindigkeit auf den Körper anwendet. Die Odometrie, die das Plugin ausgibt, entspricht der exakten Position in der Simulationsumgebung. Da deshalb kein Drift entsteht und außerdem keine Massenträgheit auf die Beschleunigung wirkt, kann mit dem Plugin getestet werden, wie sich die Lokalisierung und Navigation unter optimalen Bedingungen verhalten.

5.6 Kamerabeleuchtung

Die Kamerabeleuchtung wird über einen Node implementiert, der auf einem Mikrocontroller läuft, welcher mittels „rosserial“ [16] über USB eingebunden wird. Im Programmcode ist die Umrechnung der gewünschten Leuchttensitität, welche als float-Wert zwischen 0 und 1 angegeben werden kann, auf die entsprechende Analogspannung am GPIO-Pin des Mikrocontrollers implementiert. So kann die Helligkeit zwischen 0% und 100% von anderen Nodes geregelt werden, ohne dass Detailinformationen über die eingesetzte Hardware nötig sind.



Abbildung 5.8: Treiber für die Kamerabeleuchtung

5.7 Unterbodenbeleuchtung

Die einzelnen Leds auf den Streifen am Unterboden des Roboters können jeweils einzeln angesteuert werden. Dadurch können verschiedene Blinkmuster in einer gewünschten Farbe umgesetzt werden. Damit die Farben nicht schon bei der Programmierung des Mikrocontrollers festgelegt werden müssen, können über das Topic „rgb_led_control“ der gewünschte Modus und die Farbwerte mittels „rosserial“ [16] übermittelt werden. Auf dem Mikrocontroller sind die Blinkmodi durch verschiedene Wartezeiten und entsprechende Ansteuerung der Leds auf den Streifen implementiert.

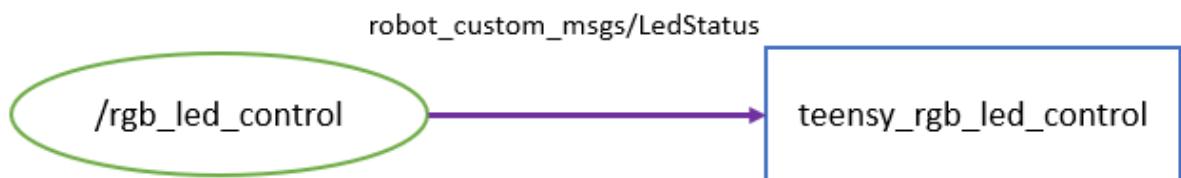


Abbildung 5.9: Treiber für die Unterbodenbeleuchtung

Da für die Steuerung des Unterbodenlichts keine Standardmessage vorhanden ist, wird im Paket „robot_custom_msgs“ (siehe auch 8.1) eine eigene Nachrichtenstruktur definiert, in der die Leuchttintensität, der Blinkmodus und die RGB Farbwerte angegeben werden können. Dadurch kann eine Anwendung beliebige Blinkmuster generieren, speichern und jederzeit nach Bedarf anpassen.

5.8 Montagepunkte und Robotergeometrie

Damit die Daten verschiedener Sensoren kombiniert werden können, müssen deren Montagepunkte auf dem Roboter definiert sein. Der feste Bezugspunkt für alle Komponenten eines Roboters ist der Rotationsmittelpunkt, der sogenannte „base_link“. Dieser liegt über dem „base_footprint“, welcher die Mitte der Grundfläche des Roboters auf dem Boden markiert. Alle geometrischen Beziehungen, die sich nicht durch Aktorik (z.B. ein Drehgelenk) verändern lassen, können als statische Transformation mit dem Node „static_transform_publisher“ veröffentlicht werden. Darunter fallen alle fest verbauten Sensoren wie die Laserscanner oder die Hinderniskameras, aber auch z.B. die Montageposition des Roboterarms. Auch die Gliederlängen des Arms und die Position der 3D-Kamera auf dem Arm können als statische Transformation angegeben werden, da diese sich auch bei

Armbewegungen nicht ändern. Damit dennoch die richtige Raumposition für z.B. die Greifervinger bestimmt werden kann, werden die statischen Transformationen mit dynamischen kombiniert (siehe auch 7.7).

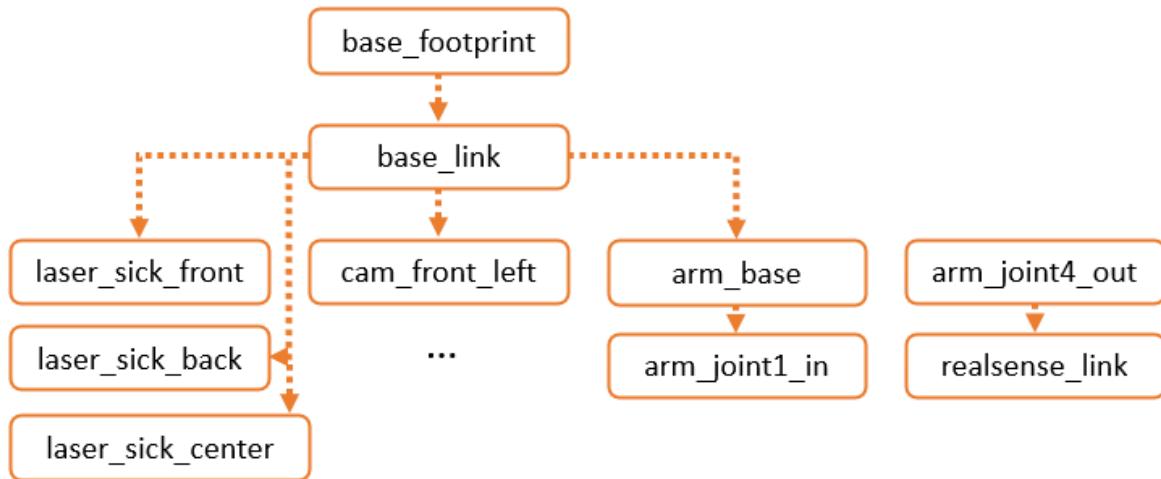


Abbildung 5.10: Statische Transformationen

Falls verfügbar, können die Transformationen auch aus einer „URDF“ Datei geladen werden, die die CAD-Daten als Grundlage dafür nimmt. Dann kann das Robotermodell auch in der Visualisierung „RVIZ“ von ROS angezeigt werden, die außerdem Sensordaten wie z.B. Laserscans oder Kamerabilder anzeigen kann.

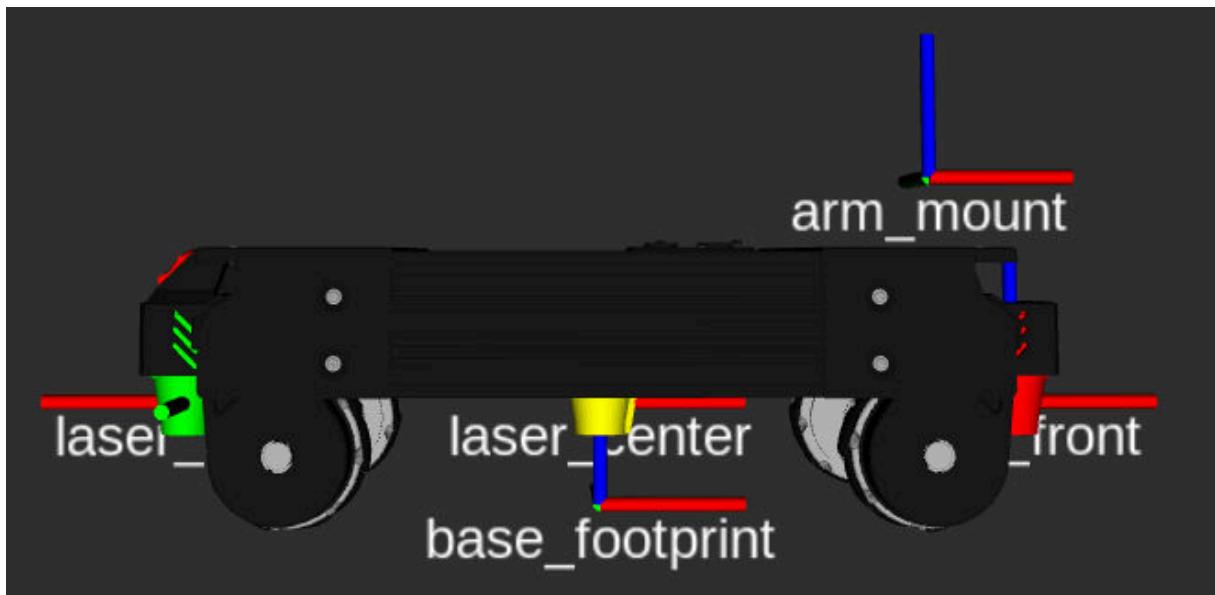


Abbildung 5.11: Robotermodell samt Transformationen

6 Algorithmik und Modelle: Model-Ebene

Algorithmik, die wissenschaftliche Problemstellungen am Roboter löst, wird in die Modell-ebene eingeordnet. Durch die offenen Interfaces ist es so möglich, auf neue Erkenntnisse zu reagieren und diese relativ einfach in das bestehende System zu integrieren, ohne auch die Treiber oder Controller anpassen zu müssen. Dadurch ist sichergestellt, dass das System bei unzureichender Performance oder bei Regeländerungen angepasst werden kann.

6.1 Laser Filter

Die Laserscanner arbeiten nach dem LiDAR-Prinzip, welches die Entfernung zu Objekten über die Signallaufzeit der Reflexion eines zuvor ausgesendeten Lichtsignals ermittelt. Oberflächen, die dieses Licht durch ihre Struktur brechen oder streuen, können daher fehlerhafte Messungen verursachen. Die Auswertelektronik der eingesetzten Sensoren filtert zwar bereits einige dieser Fehler heraus, die Aluprofile, welche als Teil der Wände in der Arena vorkommen, verursachen jedoch häufig einen „Sprungkanteneffekt“. Damit werden einzeln Messpunkte bezeichnet, die aufgrund der Streuung durch die Aluprofilkanten nicht als fester Raumpunkt interpretiert werden, sondern in einem Bereich springen, also keine stabilen Werte liefern. Da diese Punkte meist näher am Sensor eingeordnet werden als sie tatsächlich sind, werden dadurch falsche Hindernisse erkannt. Diese behindern die autonome Navigation des Roboters und müssen daher aus dem Originalscan herausgefiltert werden, bevor dieser weiterverwendet wird.

Daher wurde das Paket „laser_shadow_filter“ entwickelt, welches den Scan nach solchen Punkten durchsucht. Dafür wird für jeden Messpunkt überprüft, ob unter n Nachbarn (beidseitig wenn möglich) mindestens m Punkte innerhalb des Radius r vom aktuell betrachteten Messpunkt liegen. Ist dies nicht der Fall, so wird der aktuelle Punkt aus dem Scan „gelöscht“. Gelöscht steht deshalb in Anführungszeichen, weil die Messpunkte nicht aus der Scanmessage entfernt, sondern mit NaN überschrieben werden. Dadurch werden die Punkte für verarbeitende Programme so markiert, dass sie ignoriert werden können, das Format der eigentlichen Scanmessage jedoch beibehalten werden kann.

In den Tests am Roboter erwies sich die Konfiguration $n = 2, m = 2, r = 0.05$ m als sinnvoll, da aufgrund der hohen Auflösung der Scanner selbst bei kleinen Objekten mindestens drei Messpunkte innerhalb dieses Radius liegen. Die Sprungkanten sind meist nur einzelne Punkte, welche vom Algorithmus gefiltert werden können (siehe Abb. 6.1).

Neben der Filterung der Sprungkanten muss der Messbereich der Laserscanner limitiert werden, damit diese nicht Teile des Roboters erkennen und markieren. Für den vorderen und hinteren Laser kann dies über eine simple Winkelbegrenzung im Scannertreiber gelöst werden. Damit die Räder nicht erkannt werden, wird der Scancbereich von den maximal

möglichen 270° auf ca. 181° begrenzt.



Abbildung 6.1: Ungefilterte Laserscans

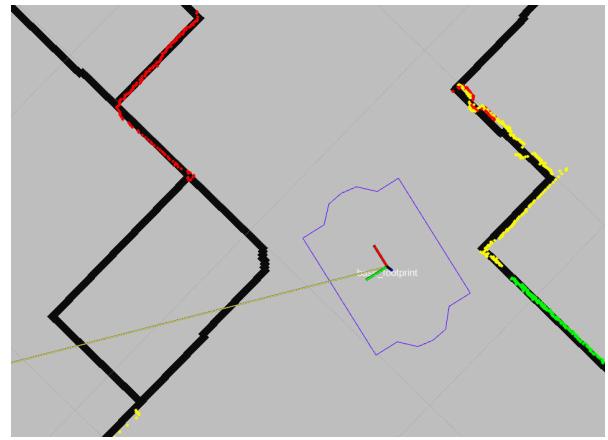


Abbildung 6.2: Gefilterte Laserscans

Da der mittlere Scanner nur seitlich zwischen den Rädern abtasten soll, muss der Scanbereich so angepasst werden, dass links und rechts jeweils ein 60° Öffnungswinkel entsteht. Es müssen also vorne und hinten 120° „gelöscht“ werden, damit die Räder und Scanner nicht in der Messung auftauchen. Dafür wird zunächst der Aufnahmebereich auf 240° reduziert, wodurch eine Seite ausgeschlossen wird. Mit dem Winkelfilter aus dem Paket „laser_filters“ [17] kann ein Winkelbereich aus einem Scan entfernt werden, wodurch die vorderen 120° überschrieben werden können. Der Winkelfilter (laf) wird noch vor dem Sprungkantenfilter (lsf) eingesetzt, sodass sich die folgenden Schnittstellen ergeben:

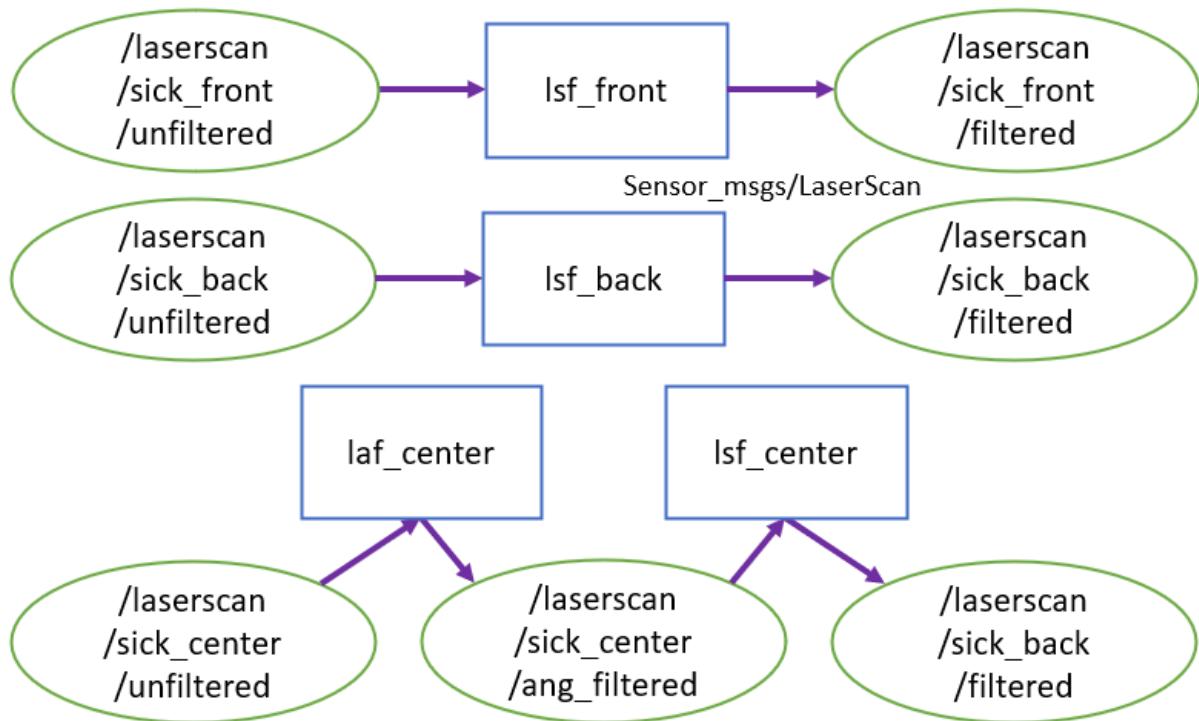


Abbildung 6.3: Laser Filter

6.2 SLAM

Das ROS Paket „gmapping“ [18] implementiert einen hocheffizienten Rao-Blackwellized Partikelfilter, der aus einem Laserscan und der Roboterodometrie eine grid map (dt: Gitterkarte) erstellt [19, 20]. Dieser erstellt für jeden Partikel eine eigene Karte, sodass die Anzahl der Partikel reduziert werden muss, um die am besten passende Karte zu erhalten. Dafür werden Roboterbewegung und Umgebungsaufnahmen so kombiniert, dass die Gesamtanzahl der Partikel, die für eine genaue Karte untersucht werden müssen, niedrig gehalten werden kann. Dadurch wird die benötigte Rechenleistung reduziert, sodass der Algorithmus auf dem verbauten PC mit bis zu 10 Hz Aktualisierungsrate ausgeführt werden kann. Die Frequenz ist dabei durch die 15 Hz der Laserscanrate limitiert, wobei ein kleiner Puffer vorgehalten wird, damit es bei Verzögerungen der Übertragung nicht zu Problemen kommt.

Der Filter selbst kann nur einen Laserscan verarbeiten, was bedeuten würde, dass z.B. nur der vordere Sensor verwendet werden kann. Da dadurch jedoch der Aufnahmebereich deutlich verkleinert würde, was zur Folge hätte, dass weniger Referenzpunkte zur Verfügung stünden, müssen die drei Laserscans vor der Verarbeitung fusioniert werden. Dafür wird der Node „laserscan_multi_merger“ aus dem Paket „ira_laser_tools“ [21, 22] verwendet. Dieser fusioniert unter Angabe der Montageposition mittels Transformation beliebig viele Laserscanner zu einer Punktwolke, die dann unter Angabe des Zielframes (in diesem Fall der mittlere Scanner) wieder in einen Laserscan umgewandelt wird. Dadurch kann ein 360° Scan aus den drei einzelnen Messungen erstellt werden, der als Eingang für den gmapping Node verwendet wird.

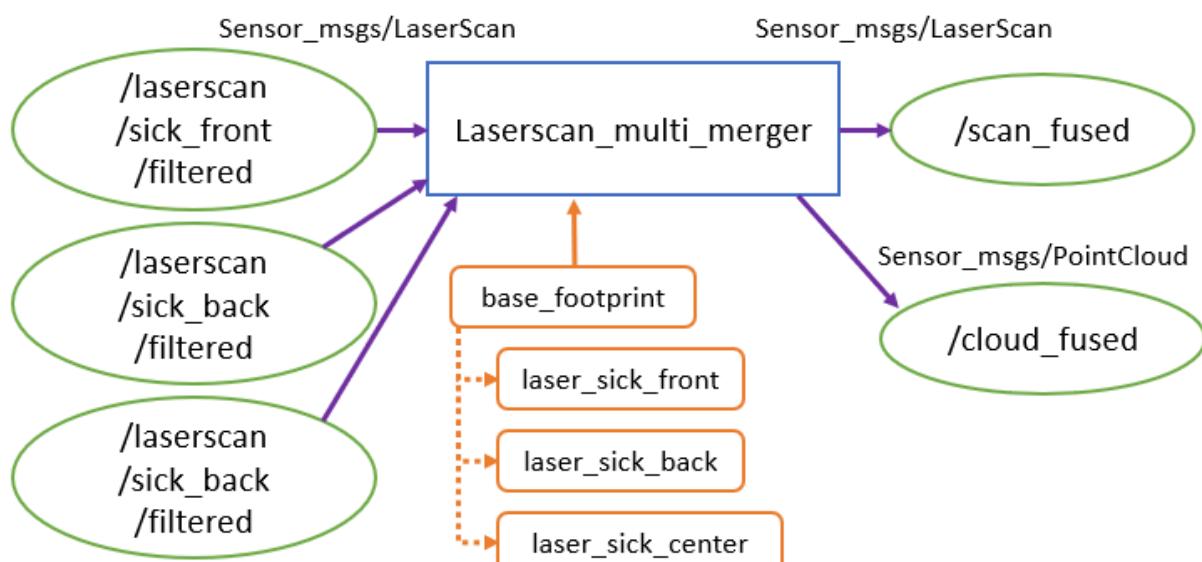


Abbildung 6.4: Laserscan Multimerger

Im Betrieb erweitert gmapping sukzessiv mit den Roboterbewegungen die Karte, welche als OccupancyGrid gespeichert wird. Dieses „Belegungsgitter“ unterteilt einen Raum in

Zellen festgelegter Größe, welche entweder als frei oder belegt markiert werden. Dafür wird jeder Zelle ein Wert zwischen 0 (frei) und 255 (belegt) zugewiesen, je nachdem, wie sicher der Algorithmus den Zellenstatus bestimmen kann.

Sobald der Kartierungsprozess abgeschlossen ist, kann die Karte, die bis dahin nur im Arbeitsspeicher des gmapping-Nodes liegt, über das Topic „map“ abgefragt werden. Mit dem Programm „map_saver“ aus dem ROS Navigation Stack [23] wird die Karte als PNG mit dazugehöriger YAML-Datei auf der Festplatte abgespeichert.

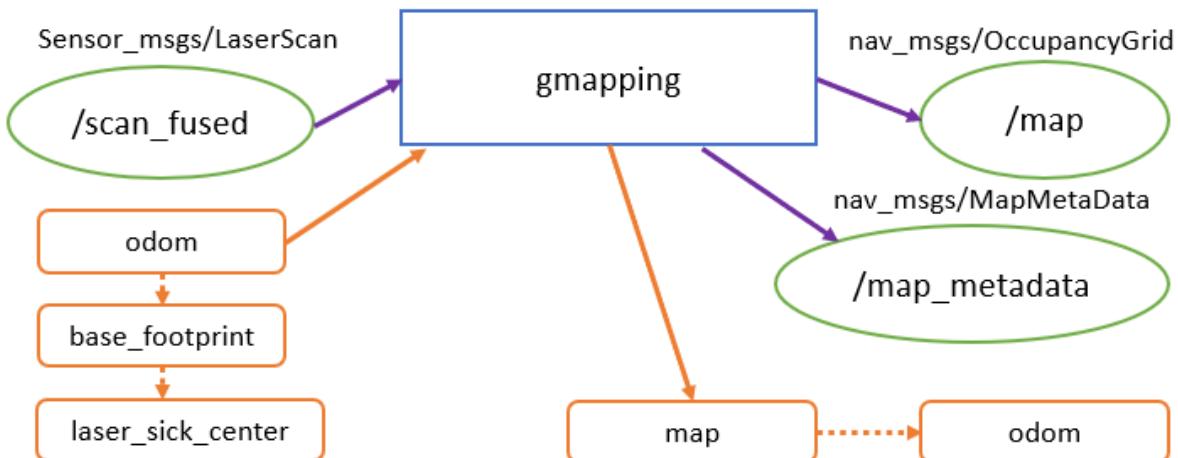


Abbildung 6.5: Gmapping

Das Schwarz-Weiss-Bild, welches aus dem OccupancyGrid erstellt wird, kann dann mit einem Bildbearbeitungsprogramm wie z.B. GIMP [24] nachbearbeitet werden, sodass kleinere Fehler entfernt oder Umrisse hinzugefügt werden können. Beim Kartieren einer @Work-Arena werden so die Markierungen für die Workstations entfernt, die eigentlich unterhalb der Scanhöhe enden und somit nicht erfasst werden können. Außerdem kann so eine zweite Hinderniskarte erstellt werden, die für die Navigation verwendet wird (siehe auch 7.4).

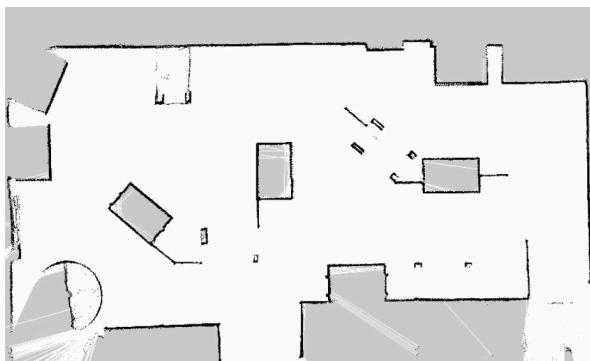


Abbildung 6.6: Originalkarte

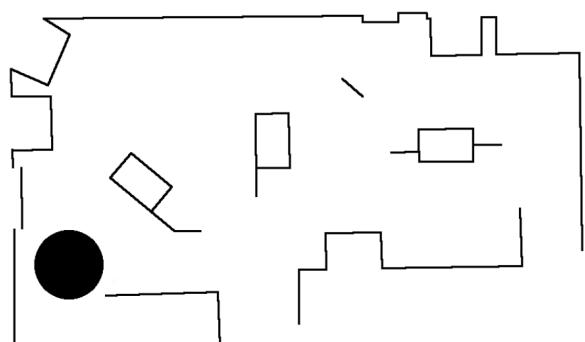


Abbildung 6.7: Modifizierte Laserkarte

6.3 Lokalisierung

Für die Lokalisierung wird zunächst die zuvor aufgenommene Karte mit dem Node „map_server“ von der Festplatte geladen und als ROS Message veröffentlicht. Dabei ist es wichtig, dass die Karte keine falschen Punkte (z.B. die Workstationmarkierungen) enthält, sondern die realen Messwerte auf der Scanhöhe möglichst akkurat wiederspiegelt. Dann funktioniert die Monte-Carlo-Lokalisierung, welche in einer studentischen Eigenimplementierung im Paket „ohm_pf“ [25,26] enthalten ist, am besten, da die Übereinstimmungen so am höchsten sind. Der darin implementierte Partikelfilter fusioniert verschiedene Sensormodelle, wie in diesem Fall die drei Laserscans und die Roboterodometrie, um so eine Roboterposition mit einer Genauigkeit von bis zu 2cm zu errechnen (abhängig von der Qualität der Karte, Odometrie und Messdaten).

Dabei handelt es sich um eine passive Lokalisierung, die für die Initialisierung keine aktiven Roboterbewegungen ausführt. Stattdessen können entweder die Partikel zufällig über die ganze Karte gestreut werden, wodurch die aktuelle Pose in der Arena mit einer zufälligen Laufzeit bestimmt werden kann, oder eine Positionsschätzung manuell eingegeben werden, die als Initialpose für die Partikelstreuung verwendet wird.

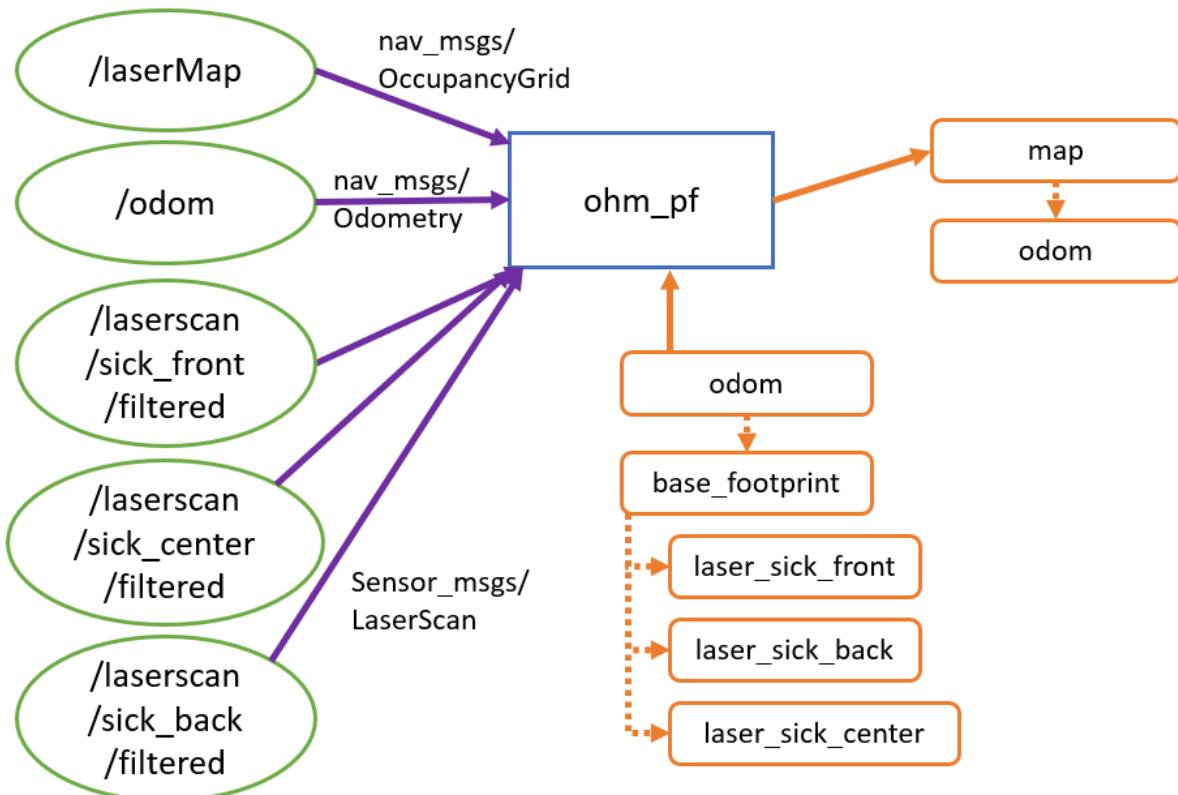


Abbildung 6.8: Ohm Partikelfilter

Der Filter startet den Betrieb sobald die erste Roboterbewegung ausgeführt wurde (z.B. manuell mit einem Joystick, siehe 7.1). Außerdem werden die Karte und die Sensordaten über ROS Topics abgefragt, bei deren Fehlen eine Warnmeldung ausgegeben wird. Aufgrund

der effizienten Implementierung ist es möglich, eine Aktualisierungsrate von 45 Hz zu erreichen, wodurch auch bei Zeitverschiebungen der Laserscandaten stets eine akkurate Lokalisierung möglich ist. Diese wird über die Transformation zwischen „map“ und „odom“ veröffentlicht, die den Odometriefehler ausgleicht und so eine korrekte Position innerhalb der Karte in der Transformation zwischen „map“ und „base_footprint“ bereitstellt. Zusätzlich wird die direkte Verbindung zwischen „map“ und der errechneten Roboterpose veröffentlicht (siehe Abb. 6.9).



Abbildung 6.9: Lokalisierung

6.4 Pfadplanung

Bei der autonomen Navigation wird zwischen lokaler und globaler Pfadplanung unterschieden. Die globale Pfadfindung sucht in einer Karte einen Weg von einer Start- zu einer Zielposition. Dabei variieren die Ergebnisse abhängig vom verwendeten Algorithmus hinsichtlich z.B. dem Abstand zu Hindernissen, der Kurvendimensionierung und der benötigten Rechenzeit. Ein Pfad besteht dabei aus mehreren Zwischenpositionen mit einem definierten Mindestabstand, die vom lokalen Pfadplaner abgefahren werden. Während der globale Planer die Umgebung zum Planungszeitpunkt berücksichtigt und dementsprechend versucht, mit dem aktuellen Wissensstand den besten Pfad zu generieren, betrachtet der lokale Planer während der gesamten Fahrtzeit alle verfügbaren Informationsquellen (Sensoren, Odometrie, Karte) und prüft, ob das Anfahren der nächsten Zwischenposition eine Kollision verursachen könnte. Dafür wird ein Ereignishorizont generiert, in dem verschiedene Geschwindigkeitskommandos simuliert und der Zustand von Roboter und

Umgebung analysiert werden. Dabei werden Kollisionsfreiheit, die Nähe zum globalen Pfad und notwendige Richtungswechseln berücksichtigt. Im Anschluss wird das Kommando umgesetzt, welches für die Weiterfahrt am besten geeignet ist.

Da beide Planer dieselben Daten verwenden, sind beide als Plugin in „move_base“ (siehe 7.4) integriert. Für die Datenkonfiguration wird daher auf den entsprechenden Abschnitt verwiesen. Da die ausgewählten Algorithmen bzw. Pakete jedoch Roboterspezifisch sind, werden sie im Folgenden näher betrachtet.

Als globaler Planer wird der „mcr_global_planner“ aus dem Paket „mas_navigation“ [27] verwendet. In Tests in der Simulation zeigte sich, dass es für eine saubere Navigation entscheidend ist, dass jeder Zwischenposition eine optimale Orientierung für den Roboter zugeordnet wird. Andernfalls steht es dem lokalen Planer frei, wie er einen Pfad abfährt, was in Engstellen aufgrund der nicht punktförmigen Chassisform zum Steckenbleiben führen kann. Der „mcr_global_planner“ optimiert daher den Pfad für ein omnidirektionales RobotermodeLL so, dass überwiegend geradeaus gefahren wird. Dafür wird ab einer gewissen Mindestdistanz am Pfadbeginn die Geradeaus-Rotation integriert, welche kurz vor dem Ziel für die Zielorientierung wieder aufgegeben wird. Da dies bei kurzen Streckenabschnitten zu unnötigen Rotationen führt, die zum Erreichen der Endposition aufgrund des omnidirektionalen Antriebs eigentlich nicht nötig sind, werden Pfade unter der Mindestlänge direkt angefahren.

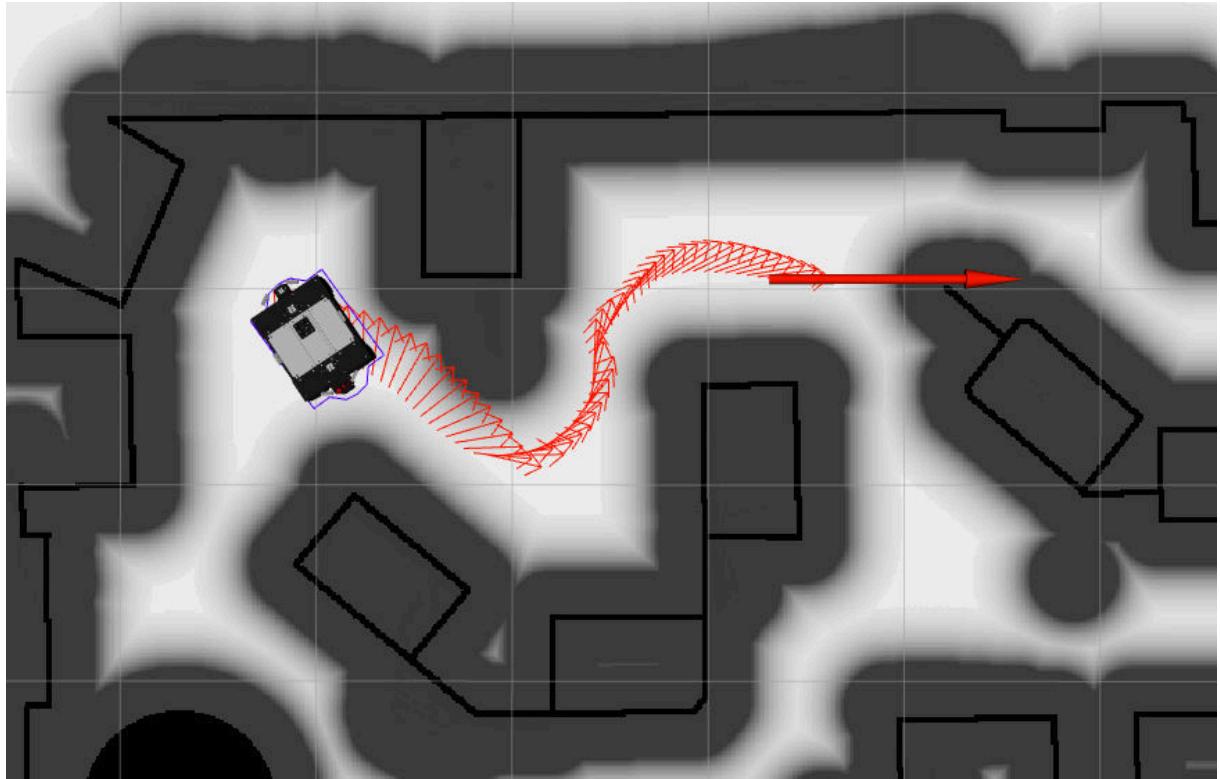


Abbildung 6.10: Globale Pfadplanung

Die eigentlichen Fahrbefehle werden mit dem „teb_local_planner“ [28–30] generiert. Dieser nutzt für die lokale Pfadoptimierung das „Time Elastic Band“ [31] Verfahren, welches die Zeit für die Ausführung der Trajektorie, den Abstand zu Hindernissen, die Fahrzeugkinematik und maximale Geschwindigkeiten bzw. Beschleunigungen berücksichtigt.

Der Anwender kann mittels Gewichtungen für diese Variablen das Verhalten des Algorithmus beeinflussen, sodass der Planer für die eigene Anwendung optimiert werden kann. Außerdem kann mit einem Parameter angegeben werden, ob die Rotationsoptimierung deaktiviert werden soll, sodass die im globalen Pfad optimierten Orientierungen eingehalten werden.

Zur Reduzierung der Rechenzeit werden die Hindernisse, welche in Form von belegten Zellen in der Hinderniskarte vermerkt sind, zu geometrischen Formen zusammengefasst, deren Kollisionsberechnung dann schneller durchgeführt werden kann. Diese sind in Abb. 6.11 als rote Quader dargestellt.



Abbildung 6.11: Lokale Pfadplanung

6.5 Kamerabildprojektion

Für die Hinderniskameras, die keinen Tiefensensor integriert haben, kann das Lochkamera-modell verwendet werden, um die Raumkoordinaten zu einem Bild zu erhalten. Dabei kann über die Linsenparameter und Bilddimensionen für jeden Pixel ein Raumvektor bestimmt werden, dessen Schnittpunkt mit der Ebene der aufgenommenen Szene die Pixelposition im Raum repräsentiert. Da die Kameras primär auf den Boden gerichtet sind und deren Montageposition auf dem Roboter bekannt ist, kann das Bild von der Sensorposition auf den Boden projiziert werden. Im Node „cam_image_to_pcl“ ist diese Funktion implementiert, indem aus dem Kamerabild und den Linseninformationen eine Punktwolke erstellt wird, deren Ursprung die Kameraposition und deren Projektionsebene der Boden ist.

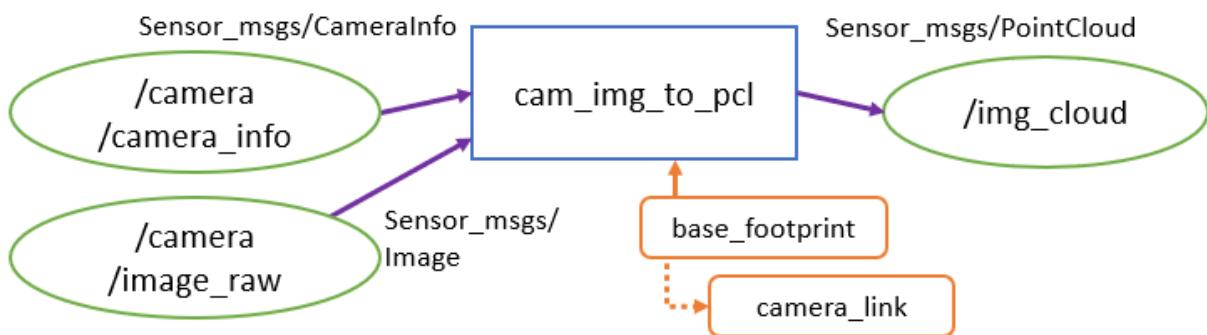


Abbildung 6.12: Camera Image to PCL

Dabei ist jedoch zu beachten, dass die einzelnen Pixel nur dann korrekt dargestellt werden können, wenn die Szene tatsächlich nur den Boden zeigt. Die Workstations werden aufgrund der fehlenden Höheninformation falsch dargestellt, da ausschließlich die Schnittpunkte der Pixelvektoren mit dem Boden bestimmt werden können. Da diese simple Version des Algorithmus jedoch primär für die Detektion von Barriertapes verwendet wird, die ausschließlich auf dem Boden angebracht werden, kann so auch mit einfachen Kameras die optische Hinderniserkennung erfolgen.

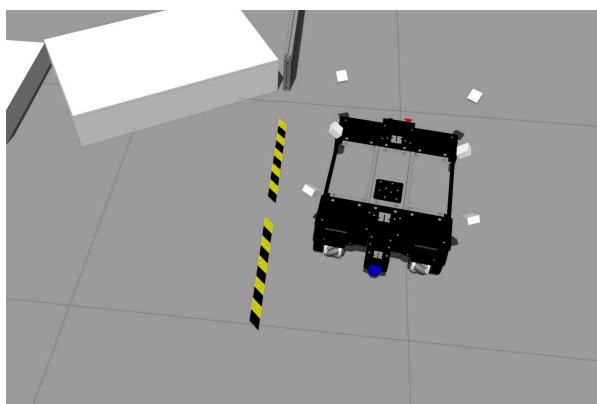


Abbildung 6.13: Barriertape in der Simulation

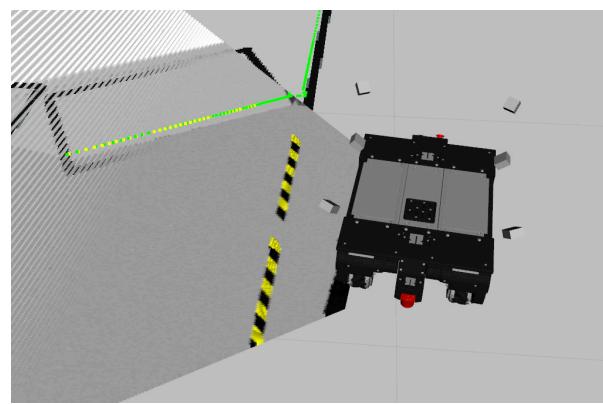


Abbildung 6.14: Aus dem Kamerabild generierte Punktwolke

6.6 Kamerabildentzerrung

Das Lochkameramodell kann nicht ohne weiteres für Fisheyeobjektive verwendet werden, da dieser Kameratyp keine eindeutige Projektionsfläche liefert. Daher muss das Kamerabild entzerrt werden, bevor es mit dem Algorithmus aus Kapitel 6.5 auf den Boden projiziert werden kann.

Im dem Fork des Pakets „image_pipeline“ [32] ist im Node „image_proc_fisheye“ eine Beispielimplementierung enthalten, die das Bild einer Fisheye-Kamera unter Zuhilfenahme der Kamerainfo, die mit dem Kalibriertool bestimmt wurde (siehe 5.2), entzerrt und dieses als Topic veröffentlicht.

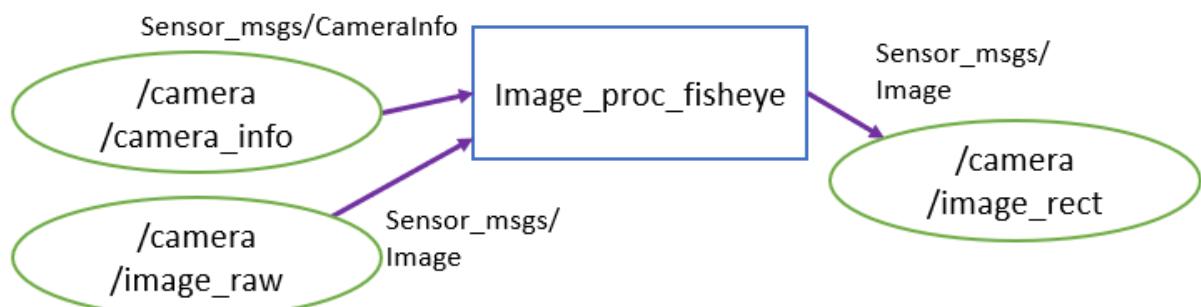


Abbildung 6.15: Fisheye Image Rectification

Damit wird aus dem Originalbild (links) der Bildbereich (rechts) extrahiert, für den das Lochkameramodell verwendet werden kann. Außerdem wird dieser Bereich so gestreckt, dass das Format beibehalten wird. Dabei gehen zwar Informationen am Bildrand verloren, der Aufnahmebereich ist gegenüber herkömmlichen Kameras dennoch etwa doppelt so groß (ca. 130°).

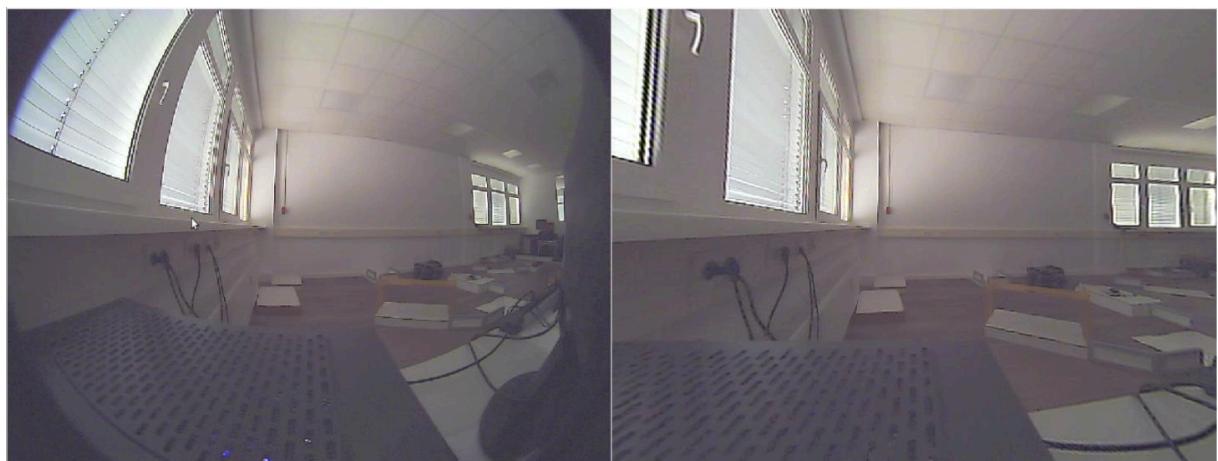


Abbildung 6.16: Bilder einer Fisheye-Kamera: Original (links) und Entzerrt (rechts)

6.7 Neuronale Netze

Auf dem Roboter werden zwei verschiedene neuronale Netze eingesetzt. Da laut Regelwerk [2] das Barriertape immer Schwarz-Gelb ist und nicht zwischen verschiedenen Versionen unterschieden werden muss, reicht es aus, als Ergebnis eine Bildmaske zu erhalten, in der erkanntes Barriertape markiert wird. Hierfür kann die U-Net Architektur [33,34] verwendet werden, die sich durch eine schnelle und präzise Bildsegmentierung auszeichnet. Für das Training eines solchen Netzes reicht außerdem ein vergleichsweise kleiner Datensatz von ca. 30 gelabelten Bildern aus, um eine stabile Erkennung des Barriertapes zu generieren.



Abbildung 6.17: Kamerabild mit Barriertape



Abbildung 6.18: Bildmaske

Dafür wurden verschiedene Keras Implementierungen des U-Nets [35,36] kombiniert und für die Verwendung und Anwendung im ROS System erweitert. Es hat sich zudem gezeigt, dass eine Bildauflösung von 320x240 ausreicht um das Barriertape zu erkennen. Im Vergleich zu Netzen für 640x480 Bildern ist die Dateigröße eines Bildes somit um den Faktor 4 reduziert, was einen Geschwindigkeitsschub von ca. 2 Hz auf bis zu 9 Hz mit dem verwendeten Python Skript erbracht hat.

Der Node für das neuronale Netz der Barriertapeerkennung erhält als Eingang das farbige Kamerabild, welches durch das Netz verarbeitet wird. Als Ausgang liefert das Netz ein monochromes Bild, in dem die erkannten Pixel je nach Sicherheit mit einem Wert von 0 (keine Erkennung) bis 255 (sichere Erkennung) markiert werden.



Abbildung 6.19: Barriertapeerkennung mit neuronalem Netz

Die Detektion der Objekte auf den Workstations erfordert eine komplexere Netzarchitektur, da hier zwischen verschiedenen Objekttypen unterschieden werden muss. Dafür wird ein

neuronales Netz mit YOLO-Architektur (You only look once) [37] eingesetzt. Diese Netze zeichnen sich aufgrund der Funktionsweise durch eine hohe Erkennungsrate und einer hohen durchschnittlichen Genauigkeit aus, womit sie für die Anwendung auf einem mobilen Roboter gut geeignet sind. Je nachdem, welche Trainingsdaten verwendet werden, können beliebige Bildformate analysiert werden. Das Netz liefert eine Liste mit Bounding Boxes (dt: Begrenzungsrahmen), Objektklassen und zugehörigen Wahrscheinlichkeiten, die für die Weiterverarbeitung genutzt werden können. In dem C++ Node, der zur Einbindung des Netzes erstellt wurde, werden diese Ergebnisse aus der Bildanalyse in eine eigene Nachricht geschrieben und veröffentlicht. Dabei erreicht der Node eine Erkennungsrate von bis zu 60 Hz, was vor allem für die Bahnerkennung von bewegten Objekten vorteilhaft ist.

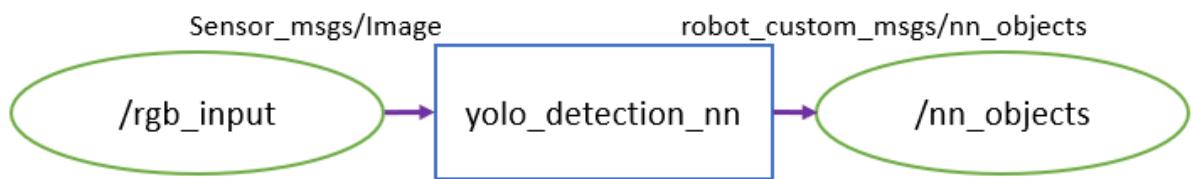


Abbildung 6.20: Objekterkennung mit neuronalem Netz

Mit der Einführung der Arbitrary Surfaces (dt: willkürliche Oberfläche) werden die Tischoberflächen teilweise verändert, sodass der Hintergrund statt weiß nun ein zufälliges (Farb-)Muster haben kann. Damit das Netz die Objekte trotz der daraus resultierenden Kontrastunterschiede zuverlässig erkennen kann, sollten schon beim Training Bilder mit verschiedenen Oberflächen verwendet werden. Da die manuelle Generierung solcher Trainingsdaten sehr zeitaufwändig ist wurde eine automatisierte Datensatzgenerierung implementiert. Diese modelliert mittels eines Python Skripts eine Beispielszene in Blender [38], in der automatisch verschiedene Objektmodelle, Hintergründe und Belichtungsszenarien verwendet und durchgewechselt werden und jeweils ein Bild gerendert wird, in dem die eingefügten Objekte automatisch markiert werden. In Kombination mit einem Datensatz, der händisch generiert wurde, können die meisten Objekte mit einer Genauigkeit von 90% erkannt werden.

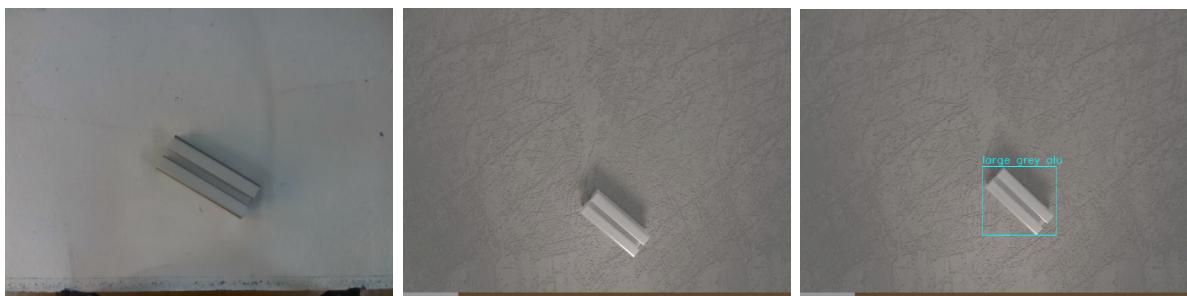


Abbildung 6.21: Datengenerierung: Kamerabild (a) Gerendertes Bild aus Blender (b) mit Label (c)

7 Hardware- und Modellkontrolle: Control-Ebene

In die Controllerschicht werden Programme eingeordnet, die dem Benutzer oder einer weiteren Anwendung Schnittstellen zur Verfügung stellen, die die Benutzung bzw. die Kontrolle von Treibern und Modellen vereinfachen. Dabei werden z.B. leicht verständliche Eingaben in weniger intuitive Datentypen umgewandelt oder der Ablauf einer Aktion geregelt.

7.1 Joy Controller

Damit der Roboter von einem Anwender händisch gesteuert werden kann, um z.B. während der Kartierung durch die Arena zu fahren, wurde ein Programm entwickelt, das die Joystickeingaben eines PS3 Controllers in Zielgeschwindigkeiten für den Roboter umwandelt. Damit der Roboter nicht unerwartet losfährt, wenn die Sticks versehentlich bewegt werden, z.B. durch Ablegen des Controllers auf einem Tisch, ist eine Totmannschaltung implementiert. Damit Bewegungskommandos gesendet werden, müssen die beiden hinteren Schultertasten gedrückt gehalten werden. Außerdem ist ein Präzisionsmodus eingebaut, bei dem nur sehr feine Bewegungskommandos geschickt werden. Dies ist z.B. für die genaue Positionierung vor einer Workstation wichtig. Der Modus kann über die beiden vorderen Schultertasten aktiviert werden.

Da der PS3 Controller nativ von Linux unterstützt wird, kann der generische „joy“ Node aus dem Paket „joystick_drivers“ [39] zur Einbindung verwendet werden. Dieser veröffentlicht die Knopf- und Hebelzustände über ein Topic, welches von dem selbst entwickelten „robot_joy_control“ Node abgefragt und in eine „cmd_vel“ Message umgewandelt wird. Diese kann dann vom Plattformtreiber (siehe 5.5) umgesetzt werden.

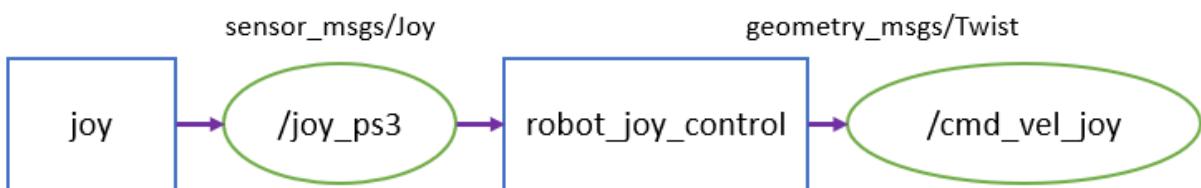


Abbildung 7.1: Steuerung mit einem PS3 Controller

Da der Roboter überwiegend autonom fahren soll und deshalb auch andere Programme Bewegungskommandos erzeugen und versenden, muss der Datenfluss zum Plattformtreiber geregelt werden, damit die Motoren und Getriebe nicht beschädigt werden, wenn sich die Kommandos gegenseitig überlagern. Außerdem muss sichergestellt werden, dass die autonome Navigation jederzeit durch manuelle Befehle überschrieben werden kann, damit in Fehlerfällen der Roboter gestoppt werden kann. Dafür kann im Node „twist_mux“ [40] für

mehrere Eingangstopics jeweils eine Priorität vergeben werden, mit der die Eingaben an den Ausgang weitergeleitet werden. Auf dem Ohmnibot werden dadurch die Controllereingaben (7.1) und die Befehle der Navigation (7.4) und Feinpositionierung (7.5) zusammengefasst.

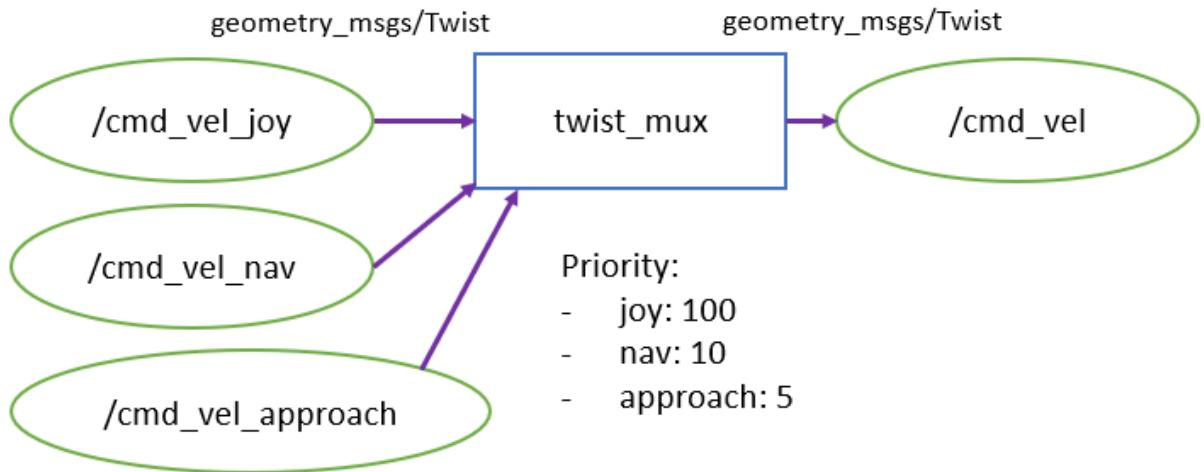


Abbildung 7.2: Prioritätssteuerung der Bewegungskommandos

7.2 Barriertape Controller

Im Barriertape Controller ist die optische Hinderniserkennung und -verwaltung implementiert (vgl. 2.3). Dafür werden die Bilder von beliebig vielen Kameras zunächst auf ihr Format überprüft und dann nacheinander an das neuronale Netz geschickt. Da das neuronale Netz für die Erkennung des Barriertapes nur mit 9 Hz läuft (siehe 6.7), eine normale Kamera jedoch 30 fps liefern kann, muss die Bildzufuhr zum Netz gedrosselt werden. Daher wird für jedes Kamerabild auf die zugehörige Bildmaske gewartet, bevor ein neues Bild zur Erkennung verschickt wird.

Die empfangene Bildmaske wird dann mit dem Algorithmus aus Kapitel 6.5 in eine Punktwolke umgewandelt, um die jeweilige Raumposition der Pixel zu erhalten. Außerdem wird jedem Punkt eine Intensität hinzugefügt, über die zwischen belegten und freien Bereichen unterschieden werden kann.

Anschließend wird die Punktwolke in die bestehende Karte integriert. Diese liegt in Form einer Gridmap vor (siehe 6.2). Da die optischen Hindernisse innerhalb der Arena erwartet werden, kann die Gridmap über die Originalkarte initialisiert werden. Dafür werden die Kartendimensionen und Zellengrößen übernommen, der Inhalt jedoch gelöscht, sodass zu Beginn eine freie Karte in Arenagröße vorliegt.

Zur Integration der Punktwolke wird jeweils die Zelle der Gridmap bestimmt, in der ein Punkt liegt. Je nachdem, ob der Punkt einen belegten oder freien Bereich markiert, wird der Belegungswert der Zelle erhöht oder reduziert, sodass Erkennungsfehler nach wenigen Messungen korrigiert werden können.

Die Gridmap wird zyklisch und unabhängig von der Erkennung über das Topic „barriertape_map“ veröffentlicht, sodass stets die aktuellste Version für die Navigation verfügbar ist. Zu Debugging-Zwecken kann außerdem eine Farbpunktwolke veröffentlicht werden, in der die hinzugefügten und gelöschten Punkte einer Bildmaske visualisiert werden. Die Karte kann über den Service „clear_memory“ zurückgesetzt werden, falls die Hindernisse Probleme für die Navigation verursachen (vgl. 7.4).

Die Erkennung kann über einen Service (de-)aktiviert werden, sodass sie z.B. während Manipulationsaufgaben an einer Workstation abgeschaltet werden kann um Rechenleistung zu sparen. Zudem ist die Bildentzerrung aus 6.6 integriert, da diese sehr rechenaufwändig ist und sowieso nur dann durchgeführt werden muss, wenn das neuronale Netz neue Daten annehmen kann. Die Entzerrung kann optional für jede Kamera einzeln aktiviert werden, wenn es sich dabei um ein Modell mit Fisheye-Linse handelt.

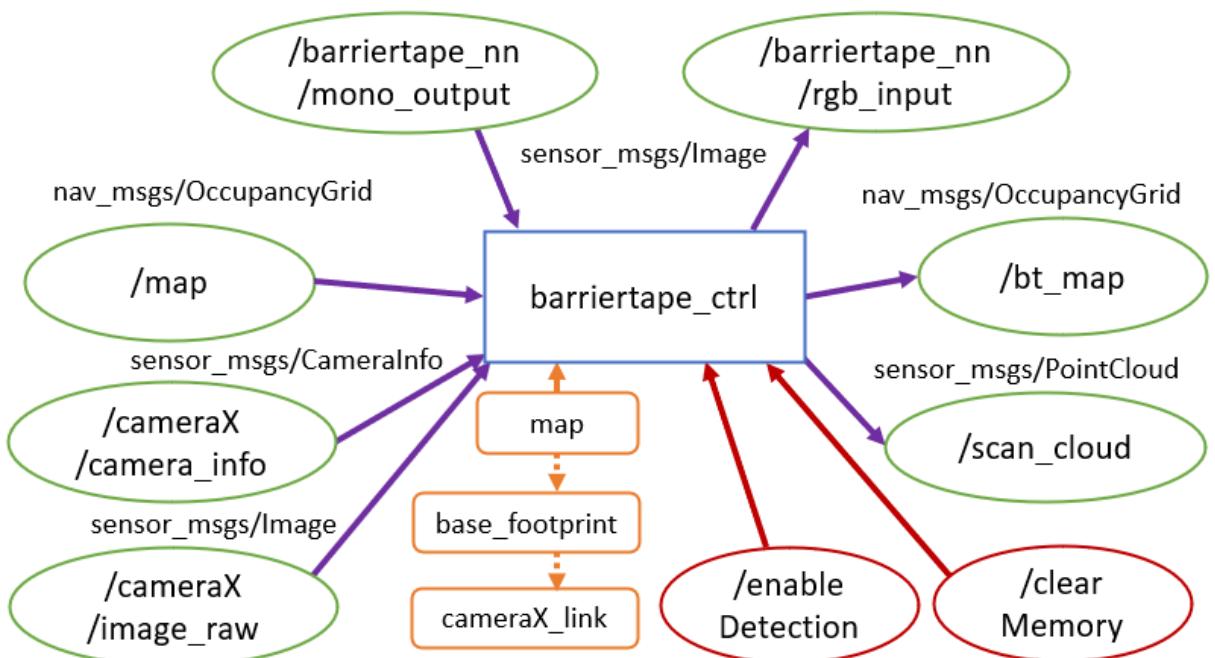


Abbildung 7.3: Barriertape Controller

7.3 Perception Controller

Da es für die Erkennung eines Objekts relevant ist, welches gesucht wird und ob die Aktion erfolgreich war, ist das Interface für die Objekterkennung als ROS Action (gold) implementiert. Diese ist vom selbst erstellten Typ „robot_custom_msgs/executeTaskAction“ (siehe 8.1) und beinhaltet die Informationen über den aktuellen Subtask, das gesuchte Objekt und die Workstation.

Zu Anfang eines Tasks ist nur bekannt, um welches Objekt und welche Workstation es sich handelt. Nähere Informationen wie die Ebene der Tischoberfläche und die Objektposition müssen während der Action ermittelt werden.

Dafür wird zunächst in der Punktewolke der 3D-Kamera, welche auf die Tischoberfläche gerichtet ist, mit der Point Cloud Library und der Ransac Methode die Hauptebene segmentiert, die die Messpunkte bilden. Da die Tischoberfläche überwiegend eben ist, wird sie durch die Hauptebene dargestellt und kann daher als Grundlage für die Positionsbestimmung der Objekte verwendet werden.

Diese werden mit dem neuronalen Netz erkannt, welches eine Liste aus Bounding Boxen und dazugehörigen IDs zurückgibt (siehe Kap. 6.7). Falls eine der IDs mit der ID des gesuchten Objekts übereinstimmt wird innerhalb der Bounding Box der Objektmittelpunkt und die Rotation bestimmt. Mit den daraus resultierenden Pixelkoordinaten und der Ebene der Tischoberfläche kann die Objektposition mit dem Algorithmus aus Kapitel 6.5 bestimmt werden.

Bei der Erkennung bewegter Objekte werden solange Objektpositionen ermittelt, mit einem Zeitstempel versehen und in einen Vektor gespeichert, bis genügend Positionen vorhanden sind, um daraus die Bewegungsstrajektorie zu errechnen. Für den Rotating Table kann diese durch den Tischmittelpunkt, also der Rotationsachse, dem Bahnradius und der Winkelgeschwindigkeit beschrieben werden. Mit diesen Informationen kann außerdem die vorraussichtliche Position des Objekts zu einem beliebigen Zeitpunkt geschätzt werden, was für das Greifen des Objekts relevant ist.

Die errechneten Informationen werden abschließend in die Antwort der Action eingefügt, sodass die Details auch für andere Aktionen verfügbar sind. Durch die Speicherung der Daten im Worldmodel (siehe 8.3) kann außerdem bei weiteren Aktionen am selben Tisch z.B. auf die bereits bestimmte Tischoberfläche zugegriffen werden, anstatt diese erneut zu bestimmen.

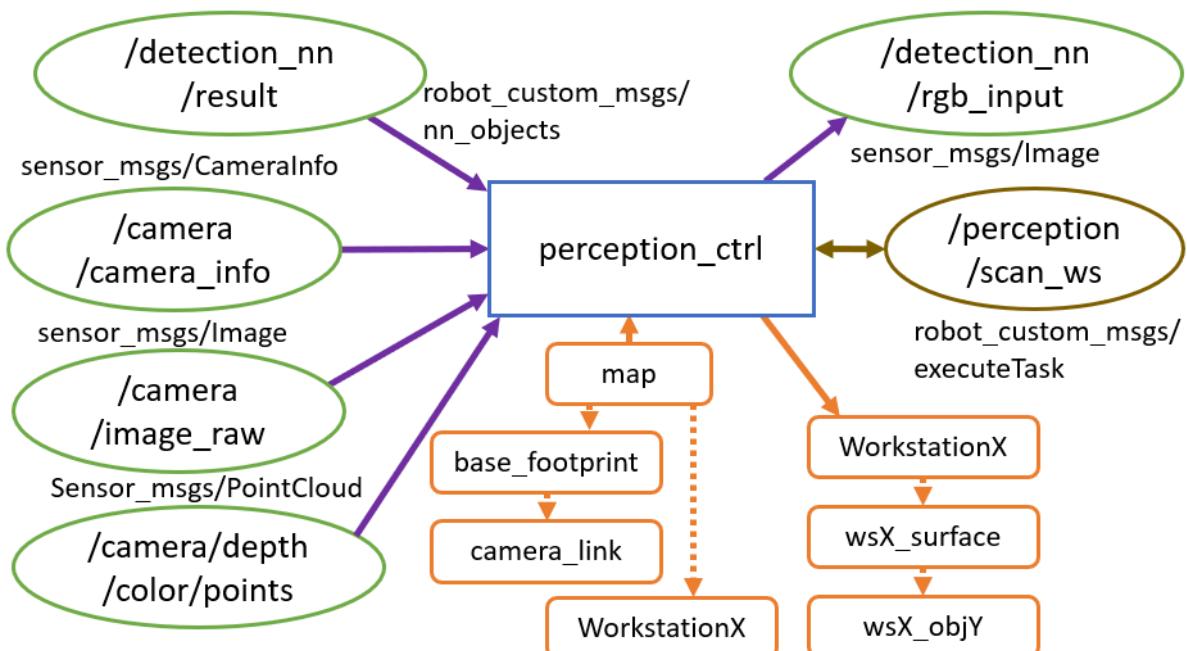


Abbildung 7.4: Perception Controller

7.4 Navigation Controller

Im Node „move_base“ aus dem Navigation Stack [23] ist eine Action implementiert die versucht, ein übergebenes Ziel durch Ansteuerung einer mobilen Roboterplattform zu erreichen. Dafür werden lokale und globale Planner integriert (vgl. 6.4), welche jeweils eine eigene Hinderniskarte in Form einer Costmap (dt: Kostenkarte) verwalten. Diese ermitteln für jede Zelle einer Karte einen Kostenwert, der die Belegung der Zelle und damit die Durchfahrtskosten darstellt (siehe auch „OccupancyGrid“ aus 6.2). Costmaps können verschiedene Datenquellen kombinieren, um so die freien und belegten Flächen in der Umgebung zu bestimmen. Auf dem Ohmnibot werden dafür eine angepasste Navigationskarte, die Barriertapekarte und die aktuellen Laserscans kombiniert. Im Gegensatz zur Laserkarte (vgl. 6.2) müssen die Workstations, die nicht mit dem Laserscanner erkannt werden können, in die Navigationskarte bzw. Hinderniskarte eingetragen werden, damit sie für die Pfadplanung berücksichtigt werden können.

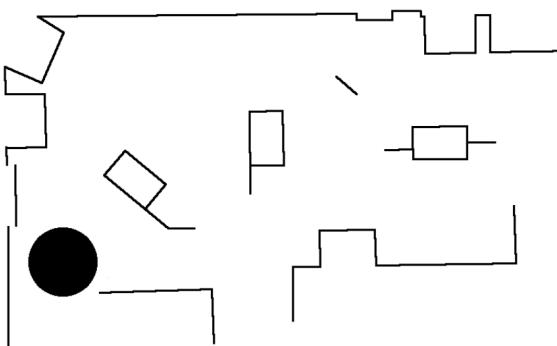


Abbildung 7.5: Laserkarte

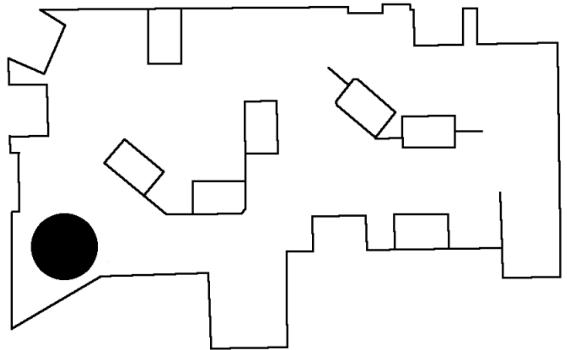


Abbildung 7.6: Hinderniskarte

In Kombination mit den anderen Datenquellen ergibt sich eine gesamte Hinderniskarte, in der auch die dynamischen Hindernisse eingezeichnet sind. Diese können Passagen enger machen oder Wege blockieren. Damit festgestellt werden kann, ob ein Roboter vorsichtig fahren muss weil Hindernisse in der Nähe sind, wird die kombinierte Karte abschließend in einem sogenannten Inflation Layer (dt: Aufblasende Schicht) zusammengefasst. Darin werden die Hindernisse abhängig von der Robotergröße durch eine Abstandsfunktion künstlich vergrößert, die die Nahbereiche von Hindernissen teurer als Freiflächen macht (siehe Abb. 6.10 und 6.11). Die Pfadplaner beziehen die Kosten jeder Zelle auf einem Pfad mit in die Gesamtberechnung ein, sodass z.B. ein enger Korridor, in dem langsam und vorsichtig gefahren werden muss, teurer ist, als ein längerer Weg durch eine freie Fläche. Dabei wird jedoch nur die Robotermitte beachtet. Für die Kollisionsberechnung wird der Grundriss der Roboterplattform als Polygon angegeben, dessen Schnittpunkte mit Hindernissen berechnet werden können.

Der globale Pfad, also der Weg von einer Start- zu einer Zielposition, kann mit dem Service „get_path“ ermittelt werden, ohne dass er ausgeführt wird. Dann wird nur der globale Planner genutzt, um diesen Weg zu bestimmen und zurückzugeben. Diese Funktion wird

z.B. zur Effizienzplanung im Task Manager (siehe 8.4) genutzt, um die Navigationszeiten zwischen Workstations zu ermitteln. Wenn es keinen ausführbaren Pfad gibt, weil der Weg z.B. durch ein dynamisches Hindernis blockiert ist, meldet der Service einen Fehler.

In der Action für die tatsächliche Navigation werden in diesem Fall sogenannte Recovery Behaviours (dt: Genesungsverhalten) ausgeführt, die z.B. Teile der Costmap löschen und so zusätzliche Freifläche schaffen, indem potentielle Fehldetections entfernt werden. Dabei sind verschiedene Level implementiert, die immer drastischere Maßnahmen ausführen und zwischenzeitlich überprüfen, ob diese erfolgreich waren. Zunächst wird nur die unmittelbare Umgebung gelöscht, sodass z.B. Hindernisse entfernt werden, die kurzzeitig mit dem Laser erkannt wurden. Der Radius wird dabei stückweise erhöht.

Anschließend wird über die „ForceFieldRecovery“ aus dem Paket „mas_navigation“ [27] versucht, durch ein Force Field (dt: Kraftfeld), welches von den Hindernissen ausgeht, einen Richtungsvektor für den Roboter zu erzeugen, mit dem dieser ein kleines Stück von z.B. einer Wand weg bewegt werden kann. Dies kann dann sinnvoll sein, wenn lokale Pfadplaner eine Kollision erkannt hat, aus der er sich aufgrund der Implementierung, die dann keine Bewegungskommandos mehr zulässt, nicht selbst befreien kann. Falls auch das nicht hilft, wird der Speicher der Barriertapekarte gelöscht, der durch den Barriertape Controller extern verwaltet wird (vgl. 7.2).

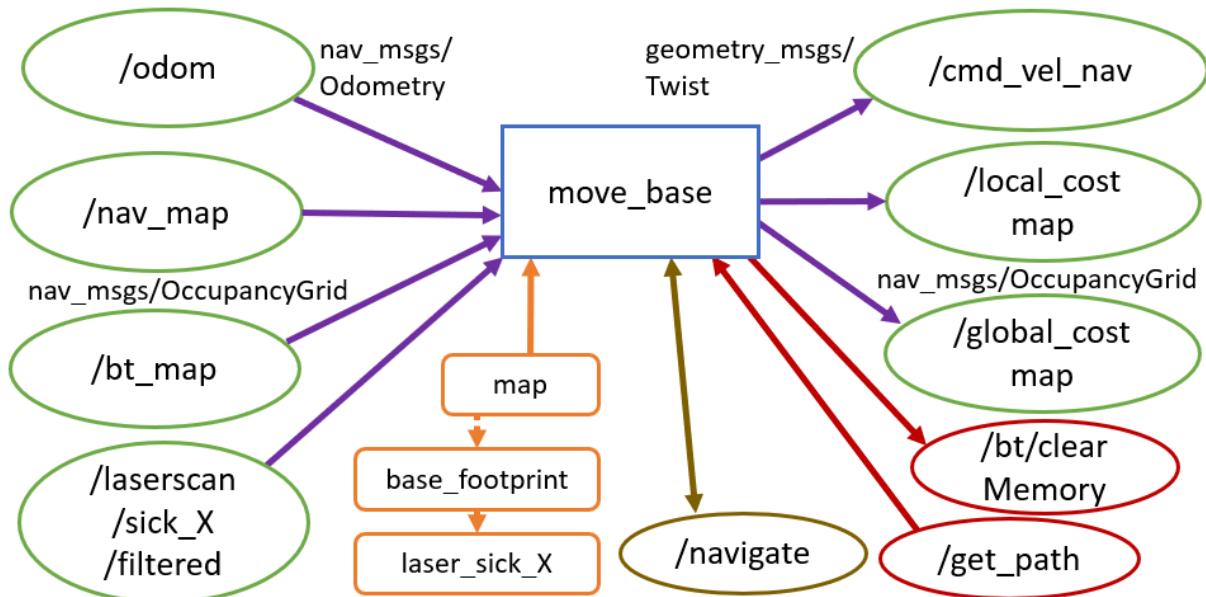


Abbildung 7.7: Navigation Controller

Wenn trotz der Maßnahmen das Ziel nicht erreicht werden kann, weil die Hindernisse z.B. jedesmal sofort wieder eingetragen werden weil sie tatsächlich existieren, bricht „move_base“ die Navigation ab und meldet einen Fehler zurück.

Das Ziel gilt als erreicht, wenn der Roboter dieses mit einer Toleranz von 20 cm erreicht hat, da der lokale Pfadplaner kleine Regelabweichungen nur schlecht ausgleichen kann. Dieser letzte Schritt wird durch den Approach Controller ausgeführt (siehe 7.5).

7.5 Approach Controller

Im Approach Node ist ein simpler Positionsregler implementiert, der dazu verwendet wird, den Roboter nach der Navigation final zu positionieren. Dafür wird der Regelfehler zwischen Zielposition und aktueller Roboterposition bestimmt und als Eingang für einen PID-Regler verwendet, der Bewegungskommandos für die Plattform generiert. Dabei werden keine Hindernisse beachtet, weshalb der Node zwar wenig Rechenleistung benötigt, aber dadurch auch nicht für die globale Navigation verwendet werden kann.

Gesteuert wird der Node über einen Service, über den die Zielposition übergeben wird. Diese ist überwiegend eine Workstation, kann aber auch z.B. die Dockingposition sein, die der Roboter vor einer Workstation einnimmt, um besser manipulieren zu können. Da diese Dockingpositionen meist so nah an der Workstation sind, dass die Navigation sie aufgrund der Kollisionsberechnungen nicht mehr anfahren würde, ist die fehlende Hindernisberücksichtigung hierbei sogar von Vorteil. Sie erfordert jedoch eine höhere Sorgfalt bei der Auswahl der Zielpositionen bzw. beim Anlernen und Einspeichern dieser.

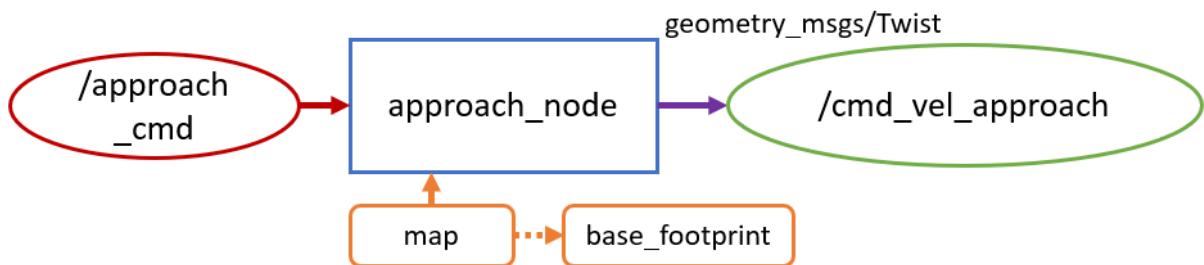


Abbildung 7.8: Approach Controller

7.6 Gripper Controller

Der „parallel_gripper_node“ steuert den im Greifer verbauten Motor. Um die Motorstellgröße für z.B. die geschlossene oder ganz geöffnete Stellung herauszufinden, kann über „Dynamic Reconfigure“ ein Stellwert übergeben werden, der vom Motor dann sofort umgesetzt wird. Sobald diese beiden Werte ermittelt sind, können relativ dazu weitere Positionen eingespeichert werden. Über den Service „gripper_control“ kann dann der Name einer solchen Zielstellung übergeben werden, sodass z.B. der Arm Controller den Greifprozess steuern kann.

Bei der Ausführung über den Service wird die Last am Motor überprüft, sodass während dem Schließen bei zu hohem Widerstand die Finger angehalten werden, um Schäden am Greifer vorzubeugen. Durch die flexiblen Finger reicht die Griffkraft dann trotzdem noch aus um ein Objekt sicher zu greifen. Über voreingestellte Motorstellbereiche kann dann festgestellt werden, ob das richtige Teil gegriffen wurde. Dies ist bei der autonomen Manipulation wichtig, da so einerseits eine doppelte Absicherung beim Greifprozess implementiert ist,

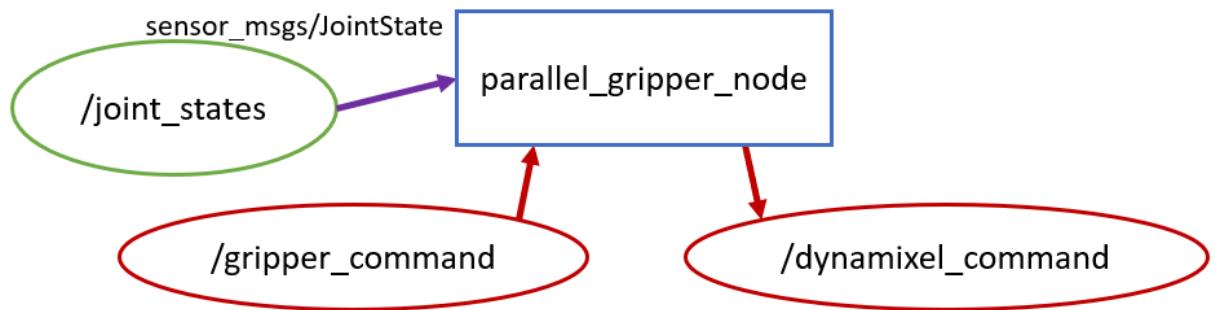


Abbildung 7.9: Gripper Controller

aber auch während der Armbewegung zum Inventar überprüft werden kann, ob das Teil zwischenzeitlich verloren ging.

7.7 Arm Controller

Im Node „tower_arm_ctrl“ ist die aktuelle Armsteuerung implementiert. Dieses Paket wurde vom Team entwickelt und basiert auf der alten Armsteuerung für den KUKA youbot. Es beinhaltet Klassen für Gelenke und Glieder, die jeweils für die aktuelle Armkonfiguration angepasst werden können. Die Glieder laden ihre geometrischen Eigenschaften aus den statischen Transformationen (siehe Kap. 5.8), sodass diese jederzeit angepasst werden können, wenn die Mechanik des Arms geändert wird. Die Gelenke werden aus einer Konfigurationsdatei geladen, in der Winkelbegrenzungen, das Übersetzungsverhältnis und die maximale Drehgeschwindigkeit angegeben sind. Außerdem kann die Rotationsachse bzw. die Richtungsachse angegeben werden, die das Gelenk durch Motorbewegungen verändert. Dadurch kann prinzipiell jedes kinematische Modell abgebildet werden, auch das des neuen Armkonzepts (siehe 3.3).

Die Gelenke und Glieder werden in einer Armmodell-Klasse zusammengefasst, in der die Vorfärtskinematik implementiert ist. Diese nutzt die Motorstellungen und Gliederlängen, um so die Raumposition des TCP zu berechnen und den Armstatus mittels Transformationen zu veröffentlichen. Zusätzlich zur aktuellen Armstellung, die aus den tatsächlichen Motorwerten gebildet wird, kann die simulierte Stellung einer neuen Position dargestellt werden, sodass diese zunächst virtuell getestet werden kann, bevor der Arm sie ausführt. Für das Greifen von Objekten ist es zusätzlich notwendig, dass die Gelenkstellungen zum Erreichen einer Raumposition mittels der inversen Kinematik berechnet werden. Diese muss zunächst für das jeweilige Armmodell bestimmt und dann in der dafür vorgesehenen Funktion implementiert werden. Aufgrund der vergleichsweise einfachen Kinematik des aktuellen Arms kann diese händisch ermittelt werden. Zur Lösung der inversen Kinematik für eine beliebige Greiferstellung werden die nötigen Gelenkwinkel nacheinander angenähert, indem unter Berücksichtigung der Armmechanik zunächst die Stellungen der Rotationsgelenke und dann die der Lineargelenke bestimmt werden.

Zur Steuerung des Arms können sowohl gewünschte Gelenkstellungen als auch eine TCP-Position mittels „Dynamic Reconfigure“ übergeben werden. Der Arm fährt diese Positionen dann direkt an. Durch Ausgabe der aktuellen Gelenkwinkel können diese abgelesen und als Standardpositionen eingespeichert werden, die dann über eine Action unter Angabe des Positionsnamens angefahren werden können. Da diese Action z.B. von einer State machine genutzt wird, ist eine Trajektorienplanung integriert, die zunächst (falls nötig) Zwischenpositionen einfügt, damit der Arm auf seiner Bahn nicht mit dem Roboter kollidiert. Die Trajektorie wird dann so dimensioniert, dass eine synchrone und flüssige Bewegung bis zur finalen Zielposition entsteht.

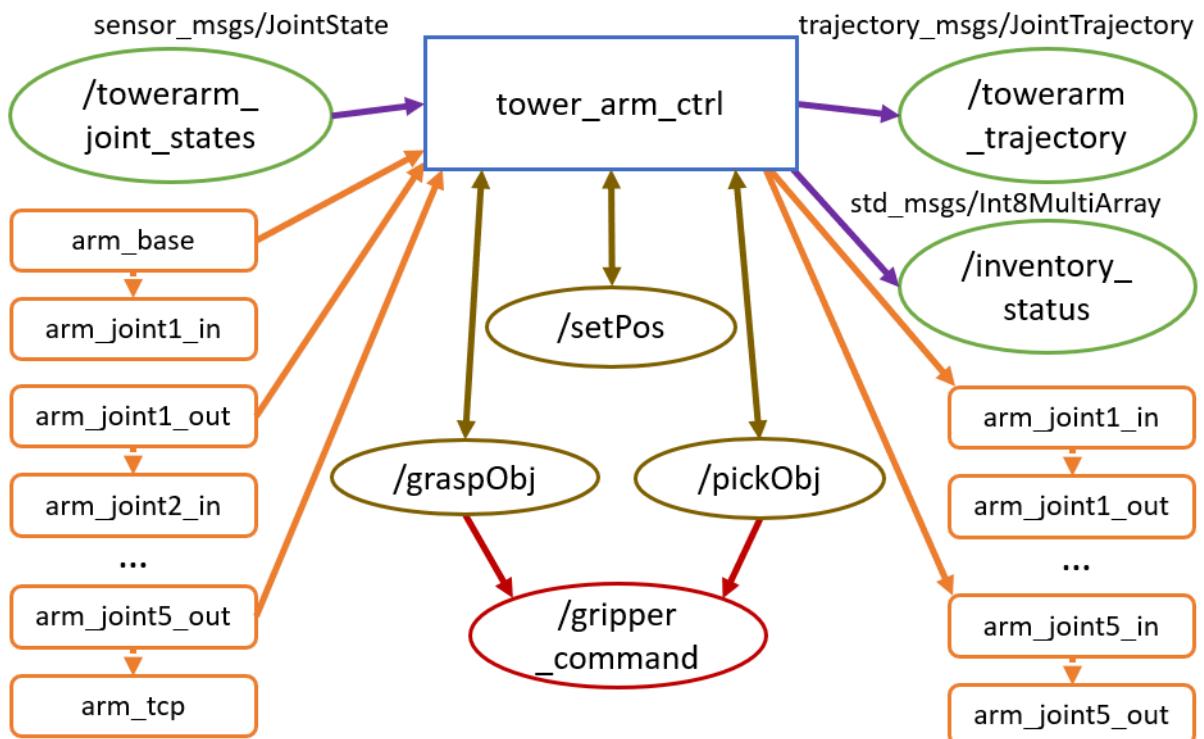


Abbildung 7.10: Arm Controller

Zusätzlich zu den Interfaces für die Positionssteuerung sind Actions für das Greifen und Ablegen von Objekten implementiert. Diese nutzen die Informationen, die über den Actiontyp „robot_custom_messages/executeTask“ übergeben werden. Die darin enthaltenen Objektpositionen werden für die Berechnung der nötigen Armstellungen genutzt, um das Objekt zu greifen, zum Ziel (z.B. Roboterinventar) zu transportieren und dort abzulegen. Abhängig vom Typ der Aufgabe werden spezifische Aktionen ausgeführt, um z.B. eine Kollision mit den Regalen zu verhindern, oder beim Greifen von bewegten Objekten einen optimalen Greifzeitpunkt abzuwarten. Aktuell ist in dem Ablauf sowohl die Greiferansteuerung als auch die Inventarverwaltung integriert, da beide Interfaces sehr häufig genutzt werden. Für beide Komponenten sind Schnittstellen implementiert, sodass der Inventarstatus auch im „status_monitor“ (siehe 8.2) verfügbar ist.

8 Entscheidungsfindung und -ausführung: Brain-Ebene

In der Brain-Ebene sind die Programme eingeordnet, die direkt mit der Ausführung eines Tasks zusammenhängen. Hier werden die Informationen über den Roboter und seine Umgebung gesammelt, aufbereitet und für die Aufgabenplanung verwendet. Analog zum menschlichen Gehirn befindet sich hier außerdem die Schaltzentrale, die die verschiedenen Komponenten des Roboters anspricht und so Aktionen und Reaktionen auslöst.

8.1 Eigene Messages

In ROS sind bereits einige Messagetypen definiert, die häufig in Robotikprojekten vorzufinden sind. Dabei handelt es sich vor allem um Sensordaten (z.B. Distanzmessungen) oder Aktorikvorgaben (z.B. Zielwinkel). Beide Typen sind meist eine Komposition aus Standardnachrichten, die simple Standarddatentypen wie Integer-, Double- oder Stringwerte implementieren. Durch die Serialisierung im ROS System können diese über die Kommunikationsschnittstellen versendet werden und so die Daten zwischen Programmen ausgetauscht werden.

Für Datenstrukturen, die noch nicht vordefiniert sind, bietet ROS die Möglichkeit, diese aus den bereits bekannten Nachrichtentypen selbst zusammenzustellen. Dafür werden Message-Files erstellt, die dann durch das Compilieren des Pakets, in dem diese enthalten sind, gebaut werden und damit auch für andere Programme verfügbar werden. Auf dem OhmniBot wurde dafür das Paket „robot_custom_msgs“ erstellt, was keine Nodes, sondern ausschließlich solche Message-Files enthält. Dadurch ist die Algorithmik und Datensemantik voneinander getrennt, sodass z.B. spezielle Bibliotheken, die in einem Bildverarbeitungsprogramm verwendet werden, nicht für das Compilieren der Armsteuerung bekannt sein müssen.

Neben den Nachrichtentypen, die bereits zuvor erwähnt wurden (siehe z.B. Kap. 5.7 und 6.7), hat sich durch die frühere Entwicklung des Taskplanners und der StateMachine herausgestellt, dass es sinnvoll ist, direkt mit ROS Nachrichtentypen als interne Datenstruktur zu arbeiten, statt dieselben Informationen in normalen C++ Klassen „doppelt“ zu definieren. Dies hat den Vorteil, dass die Daten auch zwischen verschiedenen Nodes ausgetauscht werden können, ohne dass dort jedesmal erneut die Klassen erstellt werden müssen. Außerdem erspart dies das Kopieren der Daten zwischen fast identischen Strukturen.

Die übergeordnete Datenstruktur, die von den Controllern für den Arm und die Perception verwendet wird, ist ein „Subtask“. In diesem ist der Aufgabentyp definiert, über den verschiedene Verhaltensmuster ausgelöst werden können. Der Typ ist als uint8 mit zugehörigen Standardvariablen kodiert, sodass im Programmcode einfach darauf zugegriffen werden kann. Zusätzlich kann dafür ein String angegeben werden, der für den Menschen besser lesbare Ausgaben ermöglicht.

```

1 Subtask.msg
2 -----
3
4 ##### subtask types (Ausschnitt)
5 uint8 PICK = 10
6 uint8 PICK_SHELF = 11
7 uint8 PICK_MOVING = 12
8
9 uint8 PLACE = 20
10 uint8 PLACE_PRECISE = 22
11 uint8 PLACE_CONTAINER = 23
12
13 ##### key
14 uint8 type_id
15 string type
16
17 ##### task info
18 robot_custom_msgs/Workstation workstation
19 robot_custom_msgs/Object object
20 robot_custom_msgs/Object container

```

Die Informationen über die Workstation, das Objekt und den optionalen Container sind jeweils in einer eigenen Message definiert, die in der Subtaskmessage integriert sind. Jedes Objekt besteht aus einer „ObjectSimple“ und einer „ObjectInfo“ Message, die für unterschiedliche Zwecke genutzt werden können. In „ObjectSimple“ wird lediglich der Typ und Status gespeichert, da diese Informationen für das Verhalten des Roboters relevant sind. In „ObjectInfo“ können alle Informationen über das tatsächliche Objekt gespeichert werden, die z.B. im Laufe der Aufgabenausführung durch die Detektion oder Manipulation gesammelt werden.

```

1 ObjectSimple.msg
2 -----
3
4 ##### key
5 uint8 id
6 string name
7
8 ##### delivery status
9 uint8 UNKNOWN = 0
10 uint8 LOST = 1
11 uint8 ESTIMATED = 2
12 uint8 DETECTED = 3
13 uint8 LOADED = 4
14 uint8 DELIVERED = 5
15
16 uint8 status

```

Auch die Workstation ist in „WorkstationSimple“ und „WorkstationInfo“ unterteilt. In „WorkstationSimple“ steht jedoch nur der Name, da die Workstations für jede Arena unterschiedlich heißen können. Außerdem ist der Tischtyp und der Status integriert. Auf

die Darstellung der Detailinformationen wird hier verzichtet und auf das Repository verwiesen, da für die Robotersteuerung und -logik nur die einfachen Informationen verwendet werden.

```
1 WorkstationSimple.msg
2 -----
3
4 # basic info
5 string name
6
7 ### status
8 uint8 UNKNOWN = 0
9 uint8 KNOWN = 1
10 uint8 SCANNED = 2
11
12 uint8 status
13
14 ### type
15 uint8 GROUND = 0
16 uint8 TABLE = 1
17 uint8 ROTATING_TABLE = 2
18 uint8 PRECISE_PLACE = 3
19 uint8 SHELF = 4
20 uint8 ARBITRARY = 5
21
22 uint8 type
```

Neben den Messagetypen sind außerdem eigene Services und Actions implementiert. In der Action „executeTask“ wird ein Subtask übergeben, den der ausführende Node als Datengrundlage für seine Aktionen verwenden kann. Dafür kann z.B. auf den Subtasktypen zugegriffen werden, der mittels der globalen Standardvariablen eindeutig identifiziert werden kann, sodass dementsprechende Programmteile ausgeführt werden können. Während des Programmablaufs wird der Subtask dann weiter vervollständigt bzw. den Geschehnissen angepasst (Teil gegriffen oder verloren) und abschließend zurückgegeben. So gelangen die Informationen stets zum Absender, der dann entsprechend reagieren kann. Dies ist vor allem für den Task Manager relevant, da dieser die Subtasks dann als erledigt oder gescheitert markieren kann und dadurch die weitere Aufgabenplanung beeinflusst wird (vgl. 8.4).

8.2 Status Monitor

Im „status_monitor“ wird der aktuelle Zustand des Roboters gespeichert. Dafür sind verschiedene Services implementiert, über die der Status gesetzt und teilweise auch abgefragt werden kann. Außerdem sind darin die Farbcodes für verschiedene Statusmeldungen hinterlegt, die dann an die Unterbodenbeleuchtung (siehe 5.7) gesendet werden. Die Codes werden beim Start aus einem Ordner geladen, sodass sie jederzeit angepasst und neue Modi hinzugefügt werden können.

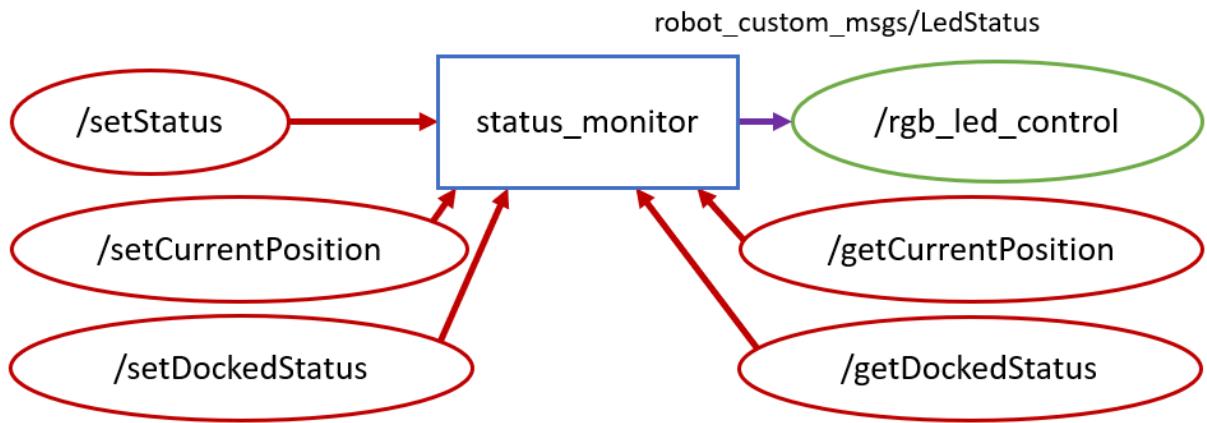


Abbildung 8.1: Status Monitor

8.3 Worldmodel

Das „Worldmodel“ dient als Datenbank für die Umgebung des Roboters. Hier werden die Karten abgelegt und außerdem die Informationen über die Workstations gespeichert. Dafür werden beim Start YAML-Dateien aus einem Ordner geladen, in denen der Name, Typ, die Position und der dazu relative Docking-Offset eingetragen sind. Die Dateien können entweder manuell oder über ein Kommandozeilenprogramm generiert werden, welches auf den Service „`saveWorkstation`“ zugreift. Dieses liest nacheinander die entsprechenden Variablen ein und sendet diese mit dem Speicher-Flag an den Worldmodel Node, sodass eine neue Datei erstellt bzw. eine existierende überschrieben wird. Die Position wird dabei über die Roboterlokalisierung bestimmt, indem der Roboter zuvor manuell vor der Workstation positioniert wird. Die Positionen werden dann über Transformationen öffentlich sichtbar gemacht.

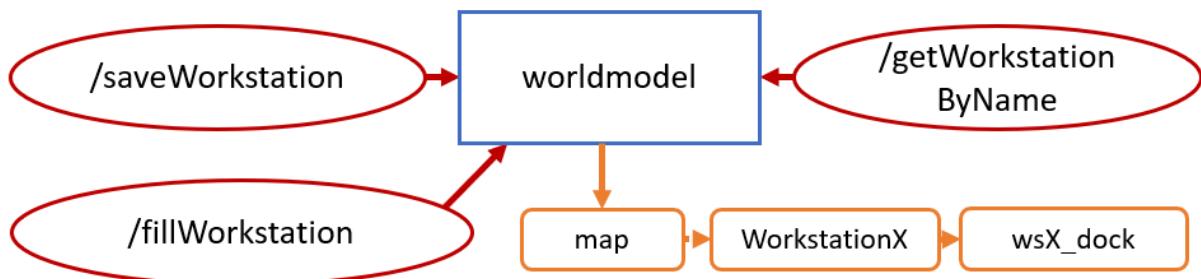


Abbildung 8.2: Worldmodel

Im autonomen Betrieb können diese Daten über den Service „`getWorkstationByName`“ abgefragt werden, der eine Workstationmessage (siehe 8.1) zurückgibt, in der die verfügbaren Daten eingetragen sind. Für das zwischenzeitliche Lesen von Daten, die während der Aufgabenausführung gesammelt werden (z.B. Objekte auf der Workstation), ist der Service „`fillWorkstation`“ implementiert, der fehlende Felder einer übergebenen Workstation auffüllt. Zum Schreiben dieser Erkenntnisse kann der „`saveWorkstation`“ ohne das Flag genutzt

werden. Dann werden die Daten nur im Arbeitsspeicher des Nodes gehalten, was für alle temporären Daten sinnvoll ist.

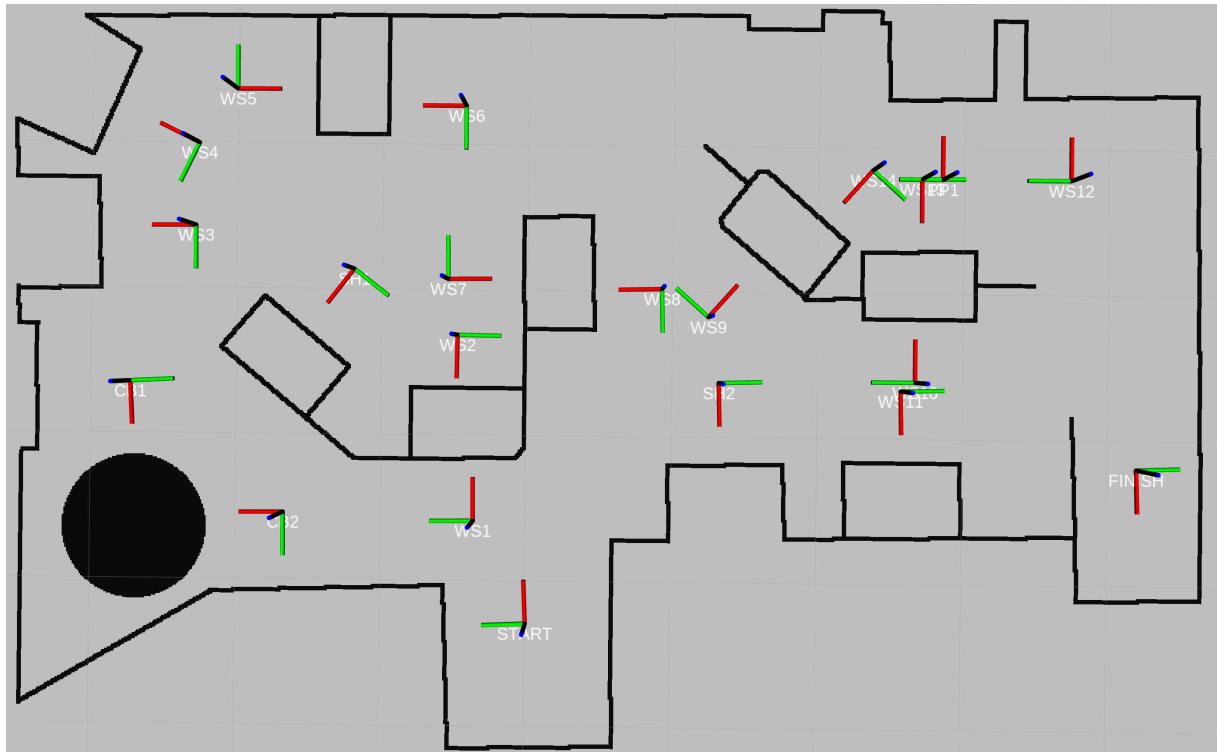


Abbildung 8.3: Workstation Positionen

8.4 Task Manager

Auf dem Ohmnibot sind die Aufgabenplanung und -ausführung (siehe auch 2.5) in getrennten Nodes implementiert, die über eine Action miteinander kommunizieren. Dabei werden die Aufgaben, die der Roboter über verschiedene Kanäle empfangen kann, vom „task_manager“ verwaltet und einzeln an den „task_executioner“ gesendet, der diese ausführt. Es sind also zwei Statemachines definiert, die jeweils ein eigenes Aufgabenlevel bearbeiten. Der Grund dafür ist, dass ein Roboter zwar selbst wissen sollte, wie eine Aufgabe auszuführen ist, jedoch im Kontext einer Fabrik mit mehreren Robotern nicht selbst für die Aufgabenplanung zuständig wäre, da ihm der Status der anderen Roboter womöglich nicht bekannt ist. Außerdem könnte sich die Art, wie ein Roboter eine Aufgabe ausführen muss, je nach Bauart des Roboters unterscheiden. Der Taskmanager könnte dementsprechend als Server zwischen mehreren Robotern agieren, der die Effizienz des Gesamtsystems optimiert, indem er die Aufgaben geschickt verteilt. Da aktuell in der @Work-Liga jedoch nur ein Roboter verwendet werden darf, beinhaltet die aktuelle Implementierung auch lediglich den Prozess für einen Roboter.

Zunächst auf eine neue Aufgabe gewartet. Primär sollen die Aufgabenlisten der @Work Refbox verwaltet werden, zu Testzwecken muss es jedoch auch möglich sein, eigene Aufgaben zu generieren und an den Task Manager zu schicken.

Damit der Messagetypr einer Aufgabenquelle variieren kann, wird für jede Quelle ein eigener Parser erstellt, der aus der Message einen oder mehrere Subtask(s) erstellt.

Sobald eine Quelle neue Daten empfängt, werden diese übernommen und ein Flag gesetzt, sodass die Abfrage „new task ?“ bestätigt wird, worauf in den Schritt der „input preparation“ (dt: Eingabevorbereitung) gesprungen wird.

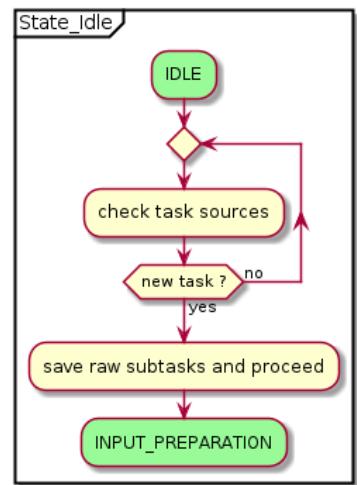


Abbildung 8.4: State Idle

Damit die Aufgaben umgeplant und verarbeitet werden können, müssen zuvor einige Schritte erledigt werden. Zunächst wird dafür das Roboterinventar aktualisiert. Wenn ein Objekt, welches in einem der Subtasks abgelegt werden soll, bereits im Inventar liegt, wird das Objekt des entsprechenden Subtask als „LOADED“ (vgl. 8.1) markiert. Andernfalls bleibt es „UNKNOWN“.

Anschließend werden über das Worldmodel die Informationen über die Workstations geladen. Bekannte Workstations werden außerdem auf eventuell gesuchte Objekte untersucht, deren Status angepasst wird, falls diese bereits gefunden wurden. Subtasks, die an unbekannten Workstations erledigt werden sollen, werden außerdem gefiltert. Die dadurch aus dem Originalvektor gelöschten Subtasks werden in eine „Garbage Collection“-Liste (dt: Müllsammlung) gespeichert und dadurch für einen eventuellen Neuversuch vermerkt (vgl. Abb. 8.13).

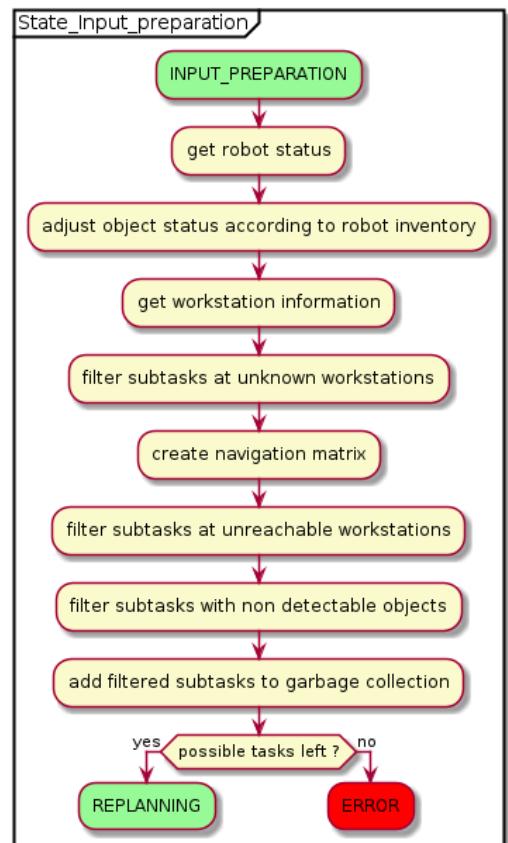


Abbildung 8.5: State Input Preparation

Im Anschluss wird eine Navigationsmatrix erstellt. Dafür werden die Pfade zwischen der Roboterposition und zwischen allen Workstations, die in der Aufgabenliste vorkommen, über den Service „getPath“ (vgl. 7.4) abgefragt und in einer Matrix abgelegt, über deren Indizes (Workstationnamen) auf diese Wege zugegriffen werden kann. Dieser Schritt spart wertvolle Zeit während der Effizienzplanung, da für diese Wege nicht jedesmal erneut der Pfad berechnet werden muss. Außerdem können Subtasks an Workstations, die nicht erreicht werden können, gefiltert und als gelöscht vermerkt werden.

Abschließend werden die Subtasks noch nach Objekten gefiltert, die womöglich nicht von der Objekterkennung erkannt werden können. Dadurch können manche Aufgaben gezielt ignoriert werden, was wertvolle Zeit im Wettkampf erspart, die wahrscheinlich sowieso keine Punkte einbringt.

Wenn am Ende der Vorbereitung noch mindestens ein möglicher Subtask übrig ist, kann in den Planungsschritt übergegangen werden. Falls nicht, springt der Algorithmus in den Fehlerzustand.

Die Effizienzplanung ist das Herzstück des Taskmanagers. Hier wird über die Reihenfolge entschieden, in der die einzelnen Subtasks abgearbeitet werden sollen. Um die Reihenfolge zu ermitteln, die insgesamt am besten ist, stehen zwei Algorithmen zur Verfügung, die beide den individuellen Score (dt: Wertung) für jeden Subtask berechnen und aufaddieren. Dieser kann auf zwei Arten ermittelt werden, wobei die Algorithmen als Abbruchbedingung eine Einheit benötigen, bei der kleine Werte besser und höhere Werte schlechter sind.

Die einfache Variante hierfür ist die angenommene Zeit, die wahrscheinlich für Ausführung des Subtasks benötigt wird. Diese setzt sich aus Zeitschätzungen für die Navigation, Detektion und Manipulation zusammen. Dafür ist jeweils ein eigener Planer implementiert, der über abgespeicherte Erfahrungswerte über z.B. die durchschnittliche Fahrgeschwindigkeit für jeden Subtask eine individuelle Dauer ausrechnet. Dabei werden sowohl der Roboter- als auch der Umgebungszustand mitbetrachtet, sodass z.B. die Detektionszeit für bereits erkannte Objekte auf null gesetzt wird. Durch die Addition der einzelnen Zeiten ergibt sich ein Subtaskscore bzw. in diesem Fall eine Gesamtdauer pro Subtask. Die Addition der einzelnen Subtaskscores ergibt die Dauer für die komplette Aufgabe, deren Optimierung in den meisten Fällen die sinnvollste Variante ist.

Wenn es jedoch unwahrscheinlich ist, dass innerhalb der vorgegebenen Zeit alle Aufgaben erledigt werden können, können zusätzlich zur benötigten Zeit T die Punkte P , die für das erfolgreiche absolvieren eines Subtasks vergeben werden, und die Erfolgswahrscheinlichkeit E mit einbezogen werden. Über die Formel $T/(P * E + 1)$ wird dann die Zeit ermittelt, die voraussichtlich pro Punkt benötigt wird. Dadurch werden die Subtasks, die zwar länger dauern, jedoch auch stärker belohnt werden, höher gewichtet und deshalb vor allem bei der Nutzung des „Nearest Neighbor“-Algorithmus (dt: nächster Nachbar) früher erledigt.

Dieser ist in Abbildung 8.6 dargestellt.

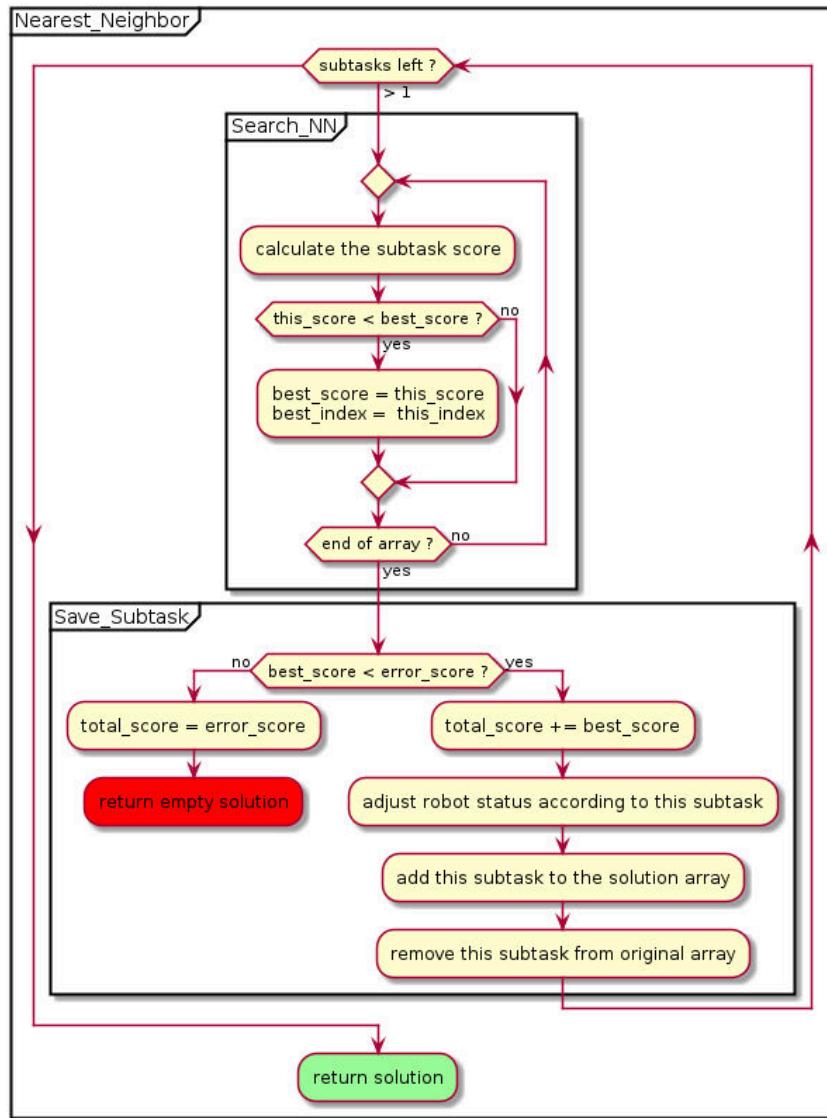


Abbildung 8.6: Nearest Neighbor Algorithmus

Der Nearest Neighbor Algorithmus sucht zunächst unter Berücksichtigung des aktuellen Roboterstatus aus der übergebenen Subtaskliste den Subtask aus, der den niedrigsten Score hat, also z.B. am wenigsten Zeit benötigt.

Dabei werden alle Subtasks, die zum aktuellen Zeitpunkt nicht ausgeführt werden können, weil z.B. das Objekt, welches abgelegt werden soll, noch nicht aufgenommen wurde, mit einem hohen Fehlerscore markiert. Solange immer mindestens ein absolvierbarer Subtask in der Liste verbleibt ist dessen Score im Vergleich zum Fehlerscore deutlich niedriger und der Subtask wird somit als nächster Nachbar identifiziert. Dann kann dieser aus der ursprünglich übergebenen Liste entfernt und in die umgeplante Liste eingefügt werden. Diese Schritte solange wiederholt, bis die ursprüngliche Liste leer ist, worauf die ermittelte Reihenfolge zurückgegeben werden kann.

Es kann jedoch vorkommen, dass trotz der Vorfilterung keine gültige Lösung existiert. Dies ist z.B. der Fall, wenn insgesamt mehr Objekte abgelegt als aufgenommen werden sollen. Dann findet der Algorithmus mindestens einmal keinen gültigen Nachbarn und gibt daher eine leere Reihenfolge zurück. Damit trotz solcher einzelnen Fehler ein Großteil einer Aufgabenliste umgeplant und später absolviert werden kann, müssen solche unmöglichen Subtasks vor der Planung identifiziert und isoliert werden.

Dafür wird der Task logisch untersucht. Dabei werden das Roboterinventar und die Aufgabenliste untersucht und für jeden Objekttypen die Aktionen vermerkt. Objekte im Inventar und Greifaktionen „beladen“ den Roboter, Ablegeaktionen „entladen“ ihn. Anschließend werden in mehreren Schritten die folgenden Fehlerquellen durchsucht:

Blockiertes Inventar Objekte, die während einer vorherigen Ausführung nicht aus dem Inventar entfernt werden konnten, blockieren den Slot permanent. Außerdem blockieren Objekte, die zwar noch normal im Inventar liegen, für die jedoch kein Ablegen geplant ist, auch den jeweiligen Slot für die aktuelle Aufgabenausführung. Wenn kein freies Inventar mehr übrig ist, müssen alle Greifaufgaben gelöscht werden.

Fehlendes Objekt Wenn mehr Objekte eines Typs abgelegt werden sollen, als dem Roboter im Inventar bzw. durch Aufnahme solcher Objekte zur Verfügung stehen werden, müssen Ablegeaufgaben entfernt werden. Wenn dabei mehrere Auswahlmöglichkeiten existieren, wird eine Bubble (dt: Blase) erstellt und in den Cage (dt: Käfig) eingefügt.

Zukünftig Blockiertes Inventar Wenn mehr Be- als Entladeaktionen durchgeführt werden, muss überprüft werden, ob genügend Inventarslots zur Bearbeitung bereitstehen. Dies sind alle Slots, die nicht blockiert sind, was im Schritt „Blockiertes Inventar“ bereits überprüft wurde. Wenn die Anzahl der Beladungen eines Objekttyps die Entladungen überschreitet, werden Greifaufgaben falls möglich gelöscht. Auch hier werden bei mehreren Auswahlmöglichkeiten Bubbles in den Cage eingefügt

Wenn eine Fehlerquelle angesprochen hat und daher Subtasks gelöscht werden müssen, kann es mehr als eine Lösungsmöglichkeit geben. Das einfachste Beispiel hierfür ist, wenn aus zwei möglichen Subtasks zum Ablegen eines Objekts nur einer gelöscht werden muss. Da erst dann entschieden werden kann, welcher davon besser behalten werden soll, wenn das Ergebnis der Effizienzplanung vorliegt, da so beide Möglichkeiten im globalen Kontext verglichen werden können, müssen mehrere Szenarien erstellt und durchgeplant werden. Pro Szenario bzw. Combination (dt: Kombination) werden dann N aus M ausgewählt, bis alle Möglichkeiten erstellt wurden. Die verschiedenen Combinations werden dann in einer Bubble gespeichert, die somit alle möglichen Lösungen für einen Problemraum beinhaltet (siehe Abb. 8.7).

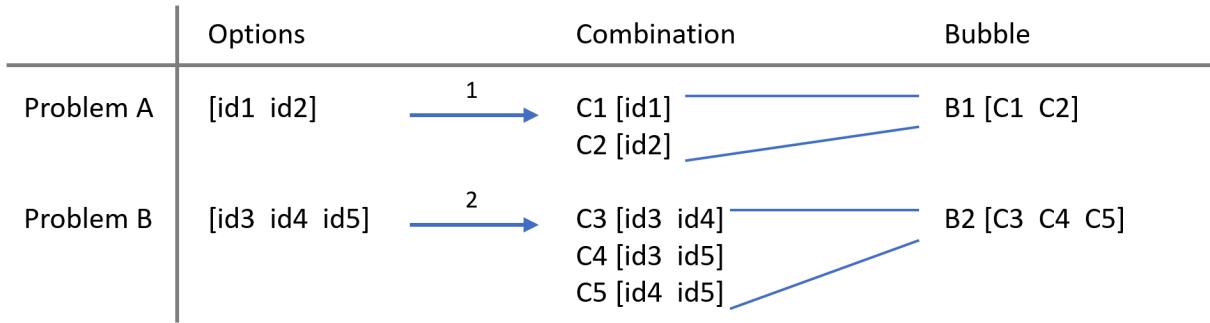


Abbildung 8.7: Problemräume als Bubbles

Falls mehr als eine Fehlerquelle identifiziert wird, wird auch mehr als eine Bubble generiert. Für eine Gesamtlösung müssen dann mehrere Bubbles kombiniert werden. Dafür wird der Cage, in dem die Bubbles abgelegt wurden, „geöffnet“ und aus allen Combinations, die in den Bubbles enthalten sind, jeweils eine Gesamtkombination erstellt. Dadurch entsteht wieder eine Bubble, die jedoch dieses mal den gesamten Problemraum abdeckt (siehe Abb. 8.8). Aus dieser Bubble werden die zu löschen Subtasks extrahiert, sodass die verschiedenen Szenarien generiert und durchgeplant werden können.

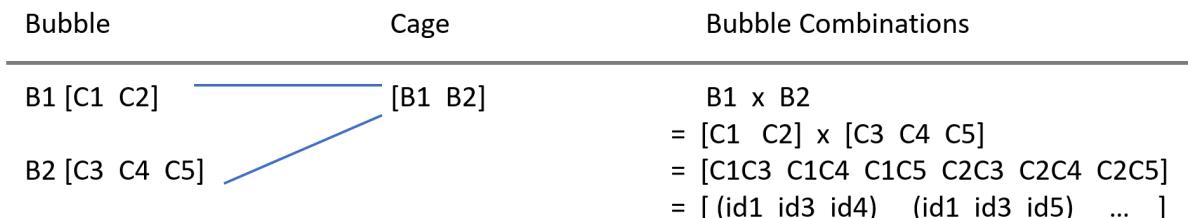


Abbildung 8.8: Kombination der Bubbles zu Lösungen

Die Taskscores aller Szenarien werden dann verglichen und jenes als finale Lösung verwendet, für das der geringste Gesamtscore ermittelt wurde.

Abbildung 8.9 zeigt, wie verschiedene Planungsalgorithmen eine einfache Aufgabenliste abarbeiten. Dabei steht die Pfeillänge zwischen Workstations für deren Distanz zueinander, die Blöcke an einer einzelnen Station zeigen einen Besuch an dieser an.

Die Standardreihenfolge des „atwork_commanders“ (Kürzel AC), also der @Work Refbox, arbeitet nacheinander jeweils eine Transportaufgabe in der vorgegebenen Reihenfolge ab. Dabei fällt auf, dass sowohl viele Wege, als auch viele Besuche entstehen. Da alle Aktionen wertvolle Wettkampfzeit kosten, ist diese Reihenfolge am schlechtesten.

Der Nearest Neighbor Algorithmus (Kürzel NN) optimiert immerhin schon die Wege, sodass keine Umwege mehr gefahren werden. Da der Algorithmus jedoch immer nur den nächstbesten Subtask auswählt, findet keine globale Optimierung der Anzahl der Besuche statt. Das Andocken und die Vorbereitung der Manipulation kosten jedoch auch jedes mal Zeit, was in einem Szenario mit komplexeren Wegen schnell ineffizient wird.

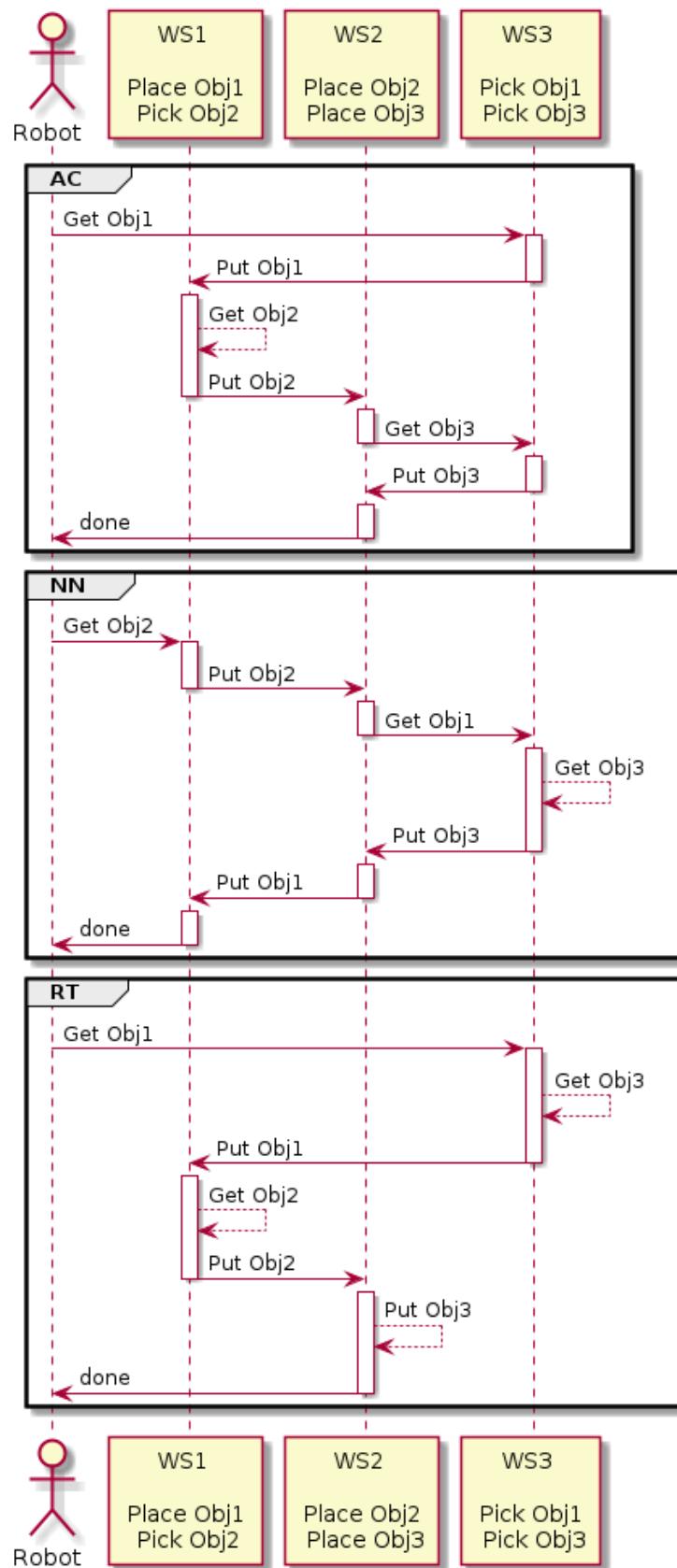


Abbildung 8.9: Vergleiche der verschiedenen Planungsalgorithmen

Die optimale Reihenfolge, bei der die Gesamtdauer bzw. der Gesamtscore global optimiert wird, wird durch den „Recursive Tree“ (dt: Rekursivbaum) (Kürzel RT) ermittelt. Um die absolut beste Lösung zu finden, müssen theoretisch alle möglichen Kombinationen erstellt und bewertet werden. Da dies bei simplen Implementierungen jedoch sehr lange dauern kann (vgl. 2.5) ist ein Algorithmus entwickelt worden, der rekursiv eine Baumstruktur erstellt, in der auf jeder Stufe die noch zu erledigenden Subtasks analysiert werden. Dabei wird für jeden Subtask zunächst überprüft, ob dieser mit dem aktuellen Roboterstatus überhaupt erledigt werden kann. Wenn dies nicht der Fall ist (z.B. Ablegen bei leerem Inventar), kann ein Zweig abgebrochen werden, da diese Reihenfolge nicht ausführbar ist.

Für Subtasks, die erledigt werden können, wird jeweils der Score berechnet und zu den bis dahin entstandenen Kosten (bzw. dem Score) dazu addiert. Wenn dieser Score noch unterhalb des besten Gesamtscores ist, hat dieser Zweig das Potential eine bessere Reihenfolge zu erhalten und soll daher weiter entwickelt werden.

Dann wird die bisherige Reihenfolge kopiert und der aktuelle Subtask angehängt. Außerdem wird der Subtask aus einer Kopie der Liste mit noch zu erledigenden Aufgaben gelöscht und der Roboterstatus dem Subtask entsprechend angepasst. Anschließend wird ein rekursiver Funktionsaufruf mit den für diesen Zweig angepassten Kopien der Daten getätigkt, sodass der Zweig weiter entwickelt werden kann.

Beim erreichen eines Blatts wird ebenfalls der aktuelle Subtaskscore bestimmt und abschließend zu dem bisherigen Score addiert, sodass ein Gesamtscore entsteht. Wenn dieser niedriger als der Score der bisherigen besten Lösung ist, wurde eine neue und bessere Lösung gefunden. Dann können die Daten der besten Lösung aktualisiert werden.

Der Algorithmus ist in Abb. 8.10 dargestellt. Damit nicht bei jedem Rekursionsaufruf der gesamte Subtaskvektor kopiert werden muss, werden die Indizes auf die darin gespeicherten Subtasks verwendet. Dadurch ergeben sich die folgenden Rechenzeiten für N Subtasks und den daraus resultierenden „Total Leafs“, also den theoretisch möglichen Kombinationen. Die Spalte „% visited“ gibt an, welcher Anteil davon nicht abgebrochen, also bis zu einem Blatt entwickelt wurde.

N-Subtasks	Calc Time	Total Leafs	% visited
10	0.02 s	$3.6 * 10^6$	$1 * 10^{-3}$
12	0.5 s	$479 * 10^6$	$2.2 * 10^{-5}$
14	14.7 s	$87 * 10^9$	$1.77 * 10^{-6}$

Tabelle 8.1: Ergebnisse der Planung mit dem Rekursivbaum

Trotz des immer geringeren Anteils der vollständig entwickelten Blätter steigt die Rechenzeit aufgrund der exponentiell wachsenden Anzahl der Möglichkeiten ebenfalls exponentiell, so dass maximal 14 Subtasks (oder 7 Transportaufgaben) durch diesen Algorithmus optimiert werden können.

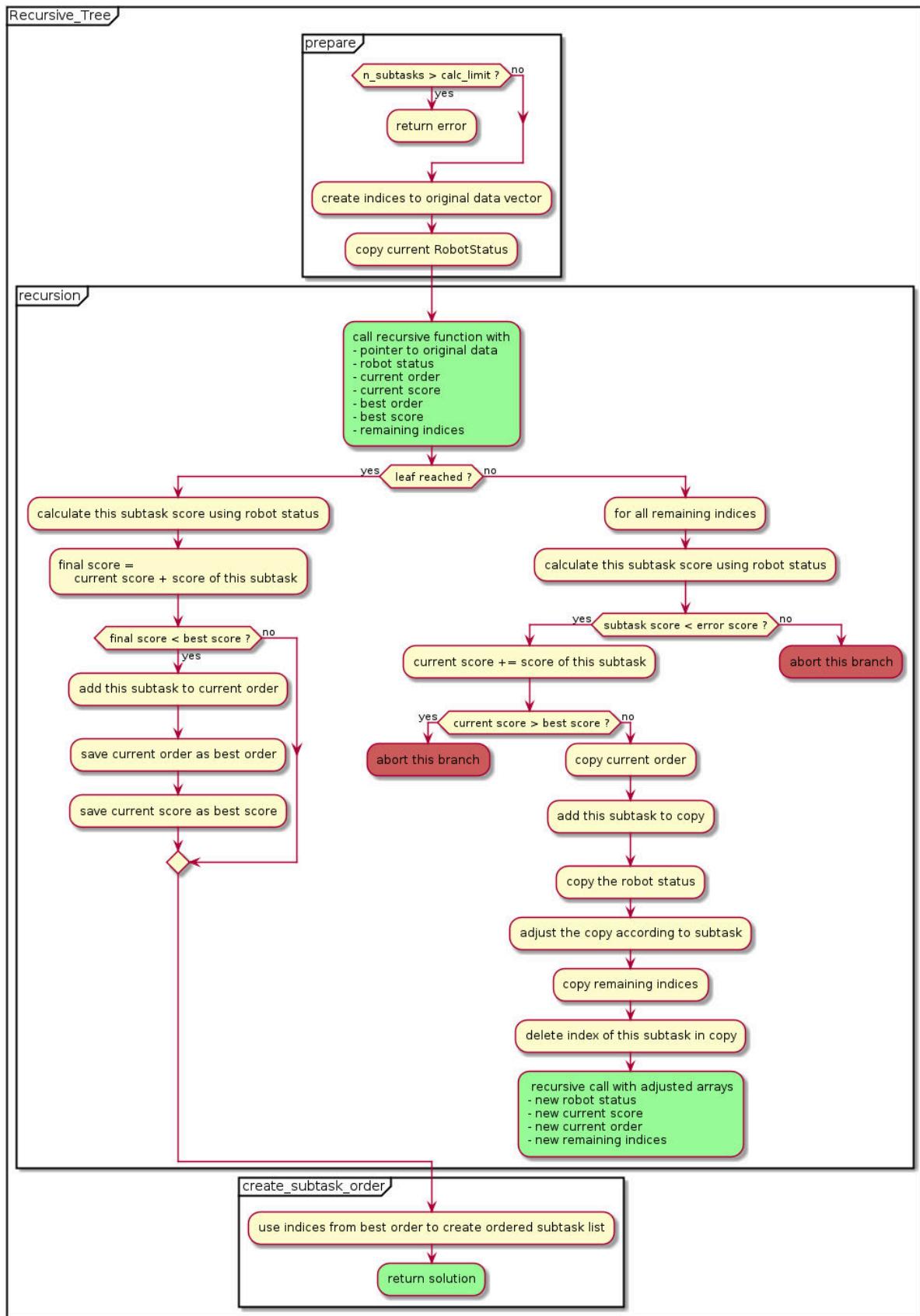


Abbildung 8.10: Rekursive Baumsuche

Da der Rekursionsalgorithmus nicht für alle Aufgabenlisten der @Work-Liga eingesetzt werden kann (Finale mit 20 Subtasks), werden für die Effizienzplanung zunächst der Original- und Nearest Neighbor-Score berechnet. Beide Methoden benötigen nur wenige Millisekunden, sodass deren Ausführung keine negativen Auswirkungen auf die gesamte Planungszeit hat. Wenn die Anzahl der Subtasks unterhalb der akzeptablen Schwelle liegt (14 Subtasks bei einer Bubble, 12 bei mehreren), kann der rekursive Algorithmus verwendet werden, um die absolut beste Lösung zu ermitteln. In allen anderen Fällen wird die Reihenfolge des Nearest Neighbor Algorithmus verwendet. Diese ist in den meisten Fällen soweit teiloptimiert, dass das Gesamtergebnis um ca. 20% von der optimalen Lösung abweicht. Da dann die Planungszeit außerdem sehr gering ist, bleibt mehr Zeit für die Ausführung übrig.

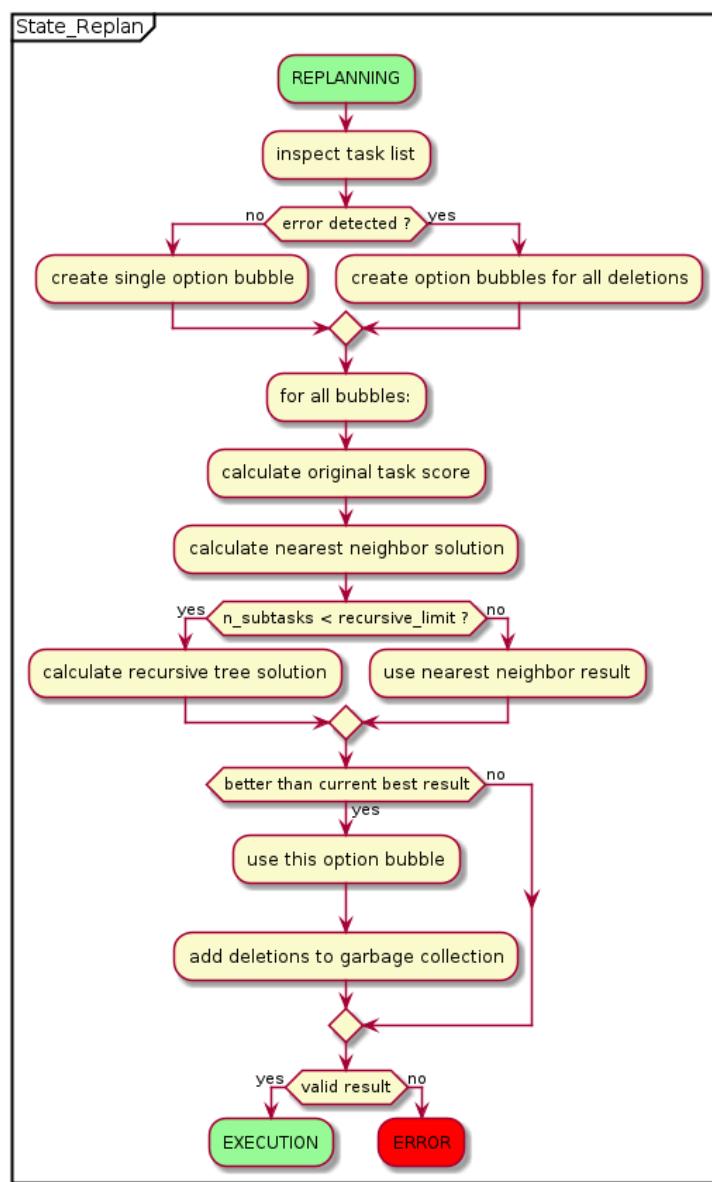


Abbildung 8.11: State Replanning

In der Ausführungsphase werden die Subtasks nacheinander einzeln an den „Task Execu-

tioner“ gesendet. Dieser meldet zurück, ob die Ausführung erfolgreich war oder ein Fehler aufgetreten ist.

Bei Erfolg werden der neue Roboter- und Umgebungsstatus abgefragt und für alle anderen Subtasks übernommen. Dadurch sind alle Informationen, die während der Ausführung gesammelt werden konnten, sofort für weitere Aktionen verfügbar. Der absolvierte Subtask wird dann in eine Liste mit erledigten Subtasks verschoben, die z.B. zur Erstellung von Statistiken verwendet werden kann. Wenn alle Aufgaben erledigt wurden, kann in die „Exit“ -Phase übergegangen werden.

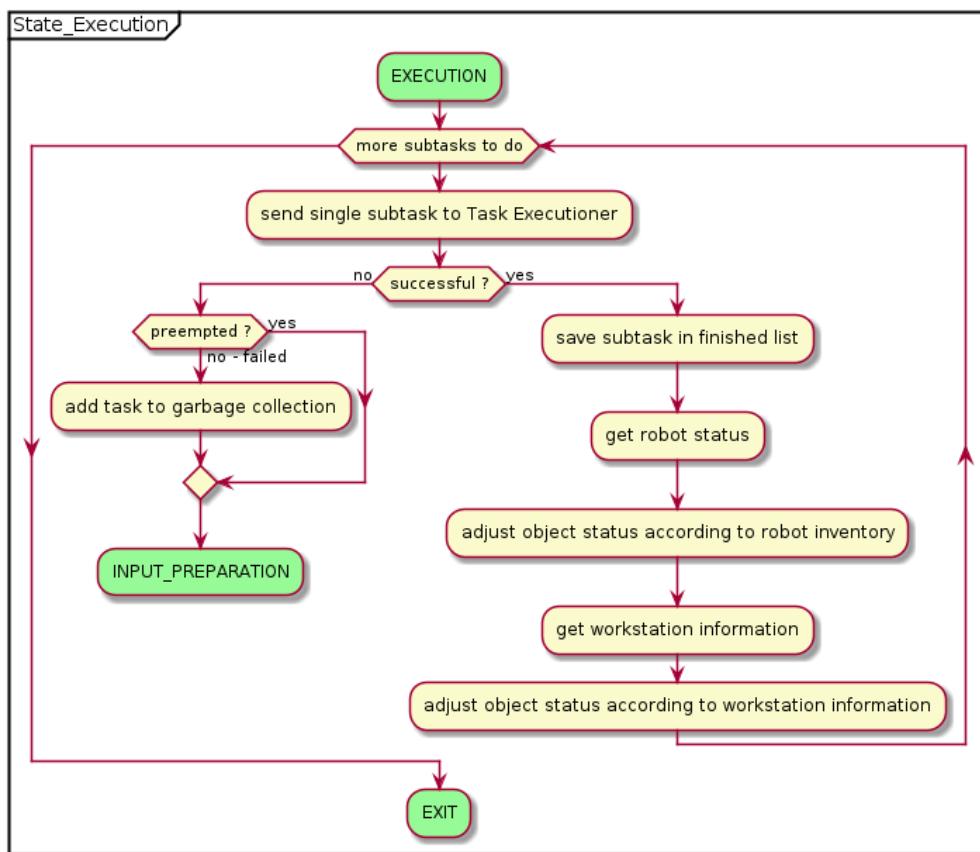


Abbildung 8.12: State Execution

Tritt während der Ausführung ein Fehler auf, weil der Roboter z.B. ein Objekt nicht detektieren konnte, meldet der Task Executioner das zurück. Dann wird der Subtask in die „Garbage Collection“ verschoben, sodass gegebenenfalls später ein neuer Versuch gestartet werden kann.

Manche Aktionen können dabei auch vorzeitig abbrechen, ohne dass sie tatsächlich fehlgeschlagen sind, z.B. wenn die Navigation feststellt, dass der ursprünglich geplante Pfad blockiert ist, und dadurch die Navigationszeit stark von der eigentlichen Planung abweicht. Dann bleibt der Subtask bestehen.

In beiden Fällen wird zurück in den Vorbereitungsschritt gesprungen, sodass alle Daten aktualisiert werden und erneut geplant wird.

Diese Feedbackschleife ist deshalb wichtig, weil sonst der ursprüngliche Plan entweder ineffizient oder sogar unmöglich wird. Damit der Roboter immer das bestmögliche Ergebnis anstrebt, muss der Plan daher immer aktuell sein.

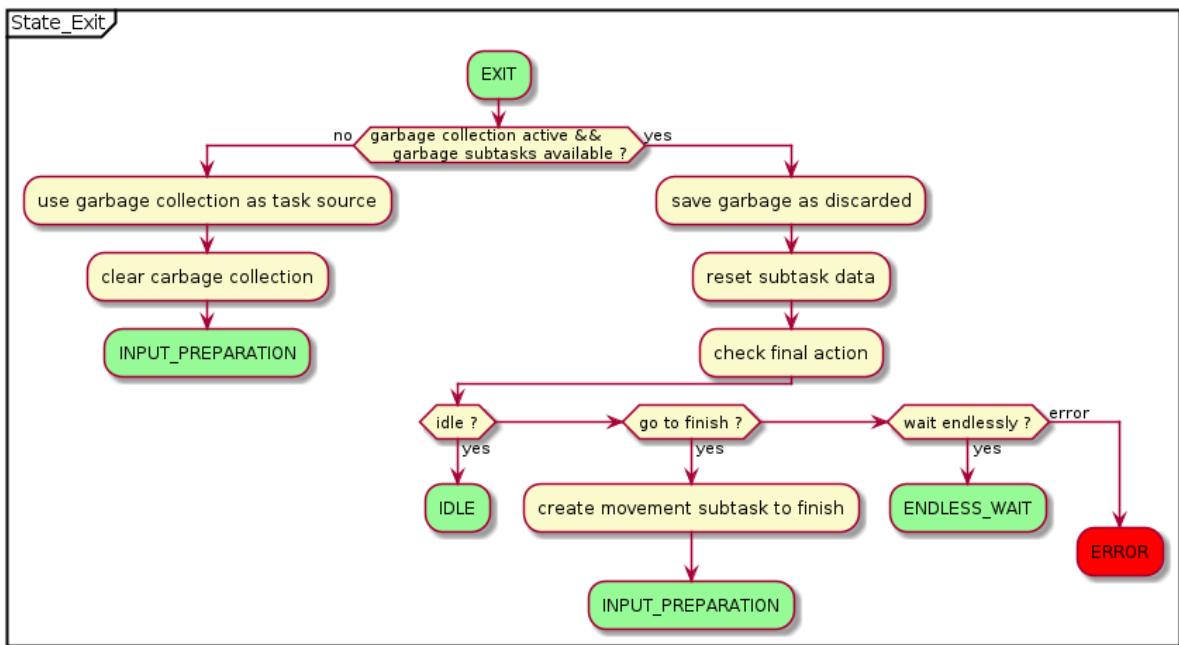


Abbildung 8.13: State Exit

In der Exit-Phase wird entschieden, wie nach der Ausführung einer Aufgabenliste weitergemacht werden soll. Wenn während der Ausführung Subtasks in die Garbage Collection verschoben wurden, kann versucht werden, diese Subtasks erneut zu probieren. Dann wird die Garbage Collection als Aufgabenquelle verwendet und in den Schritt „Input Preparation“ übergegangen, sodass ein neuer Ausführungszyklus beginnt. Das kann im Wettkampf nochmal entscheidende Punkte einbringen, weil alle Aufgaben, die nicht absolviert werden konnten, nochmal versucht und potentiell sogar absolviert werden. Damit jedoch keine Endlosschleife entsteht, falls all diese Tasks schon im Vorbereitungsschritt gefiltert werden, wird nur einmal pro Zyklus die Garbage Collection abgefragt und danach deaktiviert. Wenn die Garbage Collection nicht ausgeführt wird, werden die Subtasks daraus in eine Liste mit verworfenen Subtasks verschoben, die ebenfalls für Statistiken verwendet werden kann. Anschließend werden alle für den Planungsprozess verwendete Listen zurückgesetzt. In einem @Work-Test muss der Roboter nach Abschluss seiner Aufgaben in einen vorgegebenen Zielbereich fahren und dort warten. Daher wird im letzten Schritt überprüft, welche finale Aktion ausgeführt werden soll. Im normalen Betrieb kann in die „Idle“ Phase übergegangen und auf neue Aufgaben gewartet werden. Im Wettkampfmodus wird ein Bewegungssubtask zum Ziel generiert, geplant und ausgeführt. Dabei wird auch vermerkt, dass die nächste Abschlussaktion ist, im Ziel endlos zu warten. Das verhindert, dass der Roboter zurück in die Arena fährt, wenn er diese erstmal verlassen hat, da sonst Strafpunkte vergeben werden.

Die Interfaces des Task Managers beschränken sich auf die Aufgabenquellen, Datenquellen und der Action zum Task Executioner, der die Subtasks ausführt. Aktuell sind zwei Aufgabenquellen implementiert.

Mit dem Kommandozeilenprogramm „robot_direct_command“ können einzelne Subtasks durch Eingabe des Typs, der Workstation und des Objekts erstellt und über das Topic „direct_commander/single_subtask“ an den Task Manager gesendet werden. Außerdem können über das Topic „direct_commander/subtask_vector“ mehrere Subtasks verschickt werden, sodass eigene Szenarien erstellt und getestet werden können.

Die Verbindung zur Refbox wird über das Topic „atwork_commander/object_task“ hergestellt. Die Refbox schickt darüber die Transportaufgaben, die intern in Subtasks umgewandelt werden.

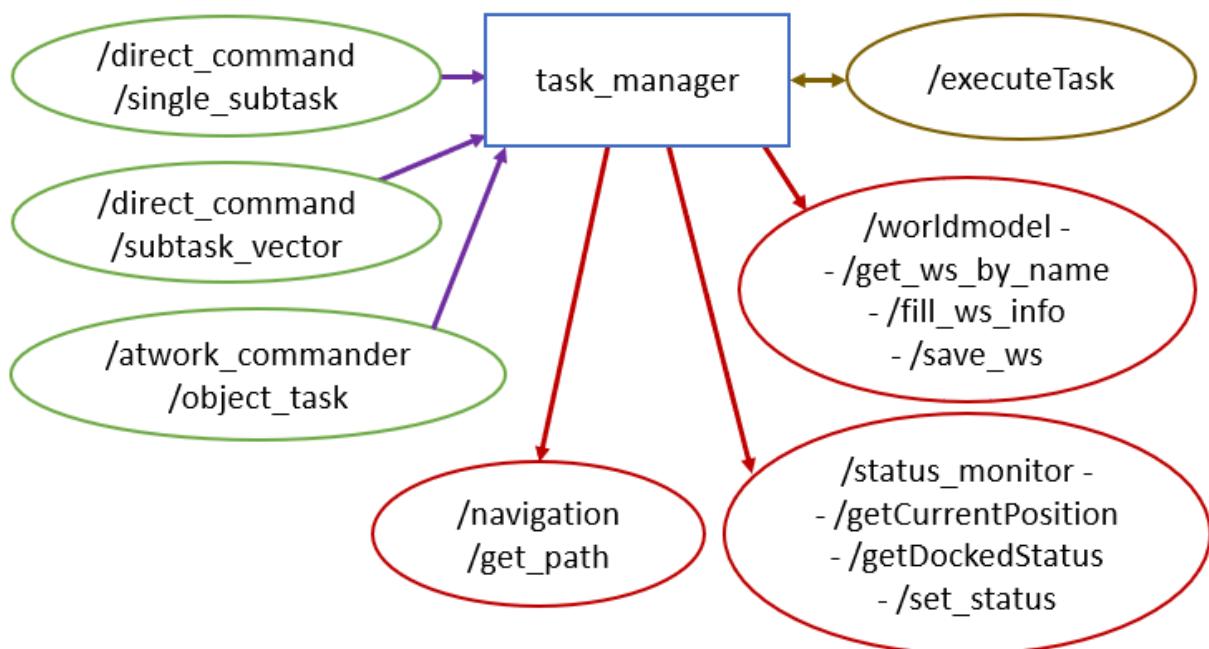


Abbildung 8.14: Task Manager

8.5 Task Executioner

Im „Task Executioner“ ist das „Wie?“ zur Absolvierung eines Subtasks („Was?“) implementiert. Hier werden alle Verbindungen zu den restlichen Systemkomponenten (z.B. Controller) hergestellt, die zur Ansteuerung der Hardware genutzt werden, also zum Auslösen einer tatsächlichen Aktion.

Für die Ausführung eines Subtasks sind überlicherweise die Aktionsbausteine „Navigation“, „Approach“, „Detection“ und „Manipulation“ nötig. Je nach Art des Subtasks und Robotерstatus kommen manche Bausteine mehrmals oder auch garnicht vor. Damit nicht für jede Variante eine eigene Statemachine erstellt werden muss, wird jeder Subtask zunächst untersucht und interpretiert.

Durch die Vorarbeit des Task Managers sind zum Interpretationszeitpunkt alle notwendigen Informationen über Workstations und Objekte im Subtask gespeichert. Daher muss zu Beginn der Interpretation lediglich der aktuelle Roboterstatus abgefragt werden. Anschließend können alle Aktionen, die zur Ausführung notwendig sind, vorgeplant werden, da mit den vorhandenen Informationen bereits alle Entscheidungen dafür getroffen werden können.

Die Aktionen werden in Form von States erstellt, die mittels Übergabeparametern so konfiguriert werden können, dass die Variablen, die Subtaskspezifisch sind, in die Abarbeitung integriert werden. Die vorgeplanten States werden in einem Vektor gespeichert, der nach der Interpretation (siehe Abb. 8.15) State für State abgearbeitet wird.

Diese beginnt mit der Überprüfung der aktuellen Position und der der Zielworkstation bzw. Zielposition. Wenn diese nicht übereinstimmen muss der Roboter navigieren. Dafür wird optional das Abdocken von der aktuellen Workstation eingeplant und anschließend die Navigation zum Ziel sowie die Feinpositionierung hinzugefügt.

Wenn an der Workstation ein Objekt aufgenommen oder abgelegt werden soll, muss der Roboter an die Workstation angedockt werden, wenn dies nicht bereits der Fall ist. Anschließend wird überprüft, ob das Zielobjekt auf der Workstation schon erkannt ist. Wenn ja kann direkt Manipuliert werden, andernfalls muss zunächst Detektiert werden, wobei dafür je nach Subtask ein eigener Timeout gesetzt wird, da z.B. für die Erkennung bewegter Objekte mehr Zeit als für statische benötigt wird.

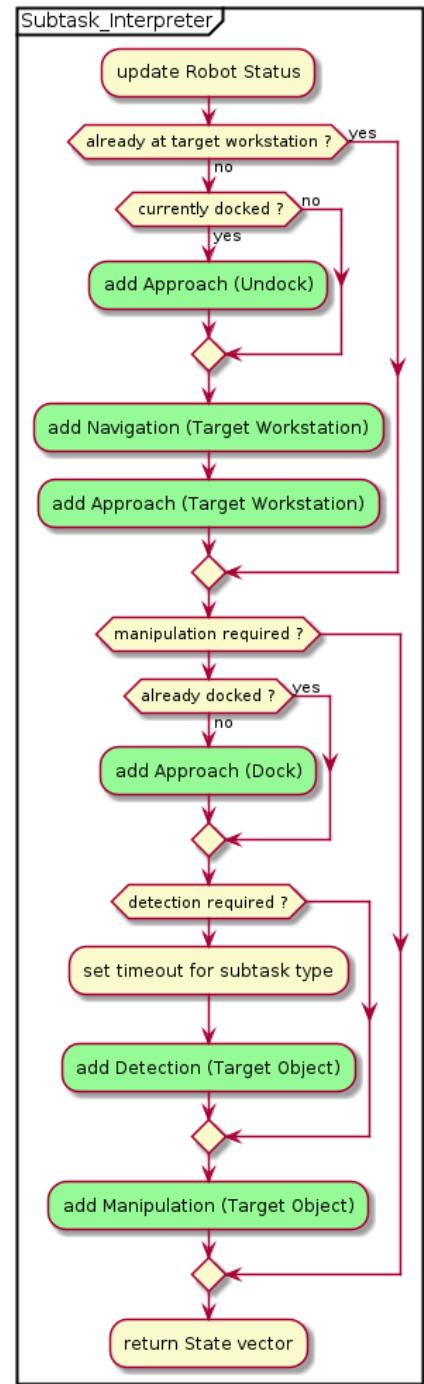


Abbildung 8.15: Subtask Interpreter

Der resultierende Vektor, der abschließend zurückgegeben wird, enthält dann alle Aktionen als States, die für die Ausführung des empfangenen Subtasks nötig sind. Damit diese trotz verschiedener Typen in einer einzelnen Schleife abgearbeitet werden können, ist jeder State nach einer Vorlage implementiert.

```
1 RobotStateBlueprint.h
2 -----
3 enum class STATE_EXIT_CODE : int {OK, PREEMPTED, ERROR};
4
5 class RobotStateBlueprint
6 {
7 public:
8     virtual STATE_EXIT_CODE entry(){return STATE_EXIT_CODE::OK;};
9     virtual STATE_EXIT_CODE run(){return STATE_EXIT_CODE::OK;};
10    virtual STATE_EXIT_CODE exit(){return STATE_EXIT_CODE::OK;};
11};
```

Diese gibt die Funktionen „entry“, „run“ und „exit“ vor, in denen jeweils verschiedene Teilaktionen durchgeführt werden können. In „entry“ (dt: Eingang) wird üblicherweise der aktuelle Status an den „Status Monitor“ (s. 8.2) gesendet, damit der Farbcde für den State gesetzt werden kann. Außerdem können hier falls nötig Vorbereitungen für die Ausführung in „run“ getroffen werden. Wenn nach einem State noch „Aufräumarbeiten“ nötig sind, wie z.B. die Objektbeleuchtung abschließend auszuschalten, kann die „exit“-Funktion in einer Kindklasse überschrieben werden. Die drei Funktionen werden für jeden State nacheinander aufgerufen und der Rückgabewert überprüft. Solange dieser „OK“ ist, wird mit der Abarbeitung weitergemacht. Sobald dieser jedoch entweder „PREEMPTED“ oder sogar „ERROR“ ist, wird Schleife beendet und dem Task Manager die entsprechende Antwort übermittelt. States, die zu diesem Zeitpunkt noch nicht bearbeitet wurden, werden verworfen, da diese nur für den aktuellen Subtask gelten, dessen Ausführung durch den Fehler nicht fortgesetzt werden kann.

Damit in den verschiedenen States auf dieselben Roboterkomponenten zugegriffen werden kann, ist zu jeder Komponente ein eigenes Softwareinterface implementiert, in dem die möglichen Aktionen gekapselt sind. Beim Start des Task Executioners werden alle Interfaces erstellt und in einen „Robot Controller“ geladen, der alle Interfaces mittels Pointern verwaltet. Diese Pointer können dann von anderen Klassen abgefragt werden, sodass die Interfaces nur einmal erstellt und anschließend überall verwendet werden können. Während der Subtaskinterpretation werden den einzelnen States Pointer zu allen Interfaces übergeben, die diese intern für die Ausführung benötigen. Die Interfaces stellen den States dann simple Funktionen zur Verfügung, in denen die spezifische Kommunikation mit einer Komponente umgesetzt ist, sodass bei Änderungen am System lediglich das Interface, jedoch nicht der State angepasst werden muss. Außerdem können jedem Interface die Topic- und Service-Namen übergeben werden, sodass schon beim erstellen eines Interfaces überprüft werden kann, ob eine Verbindung zu einer Komponente hergestellt werden konnte.

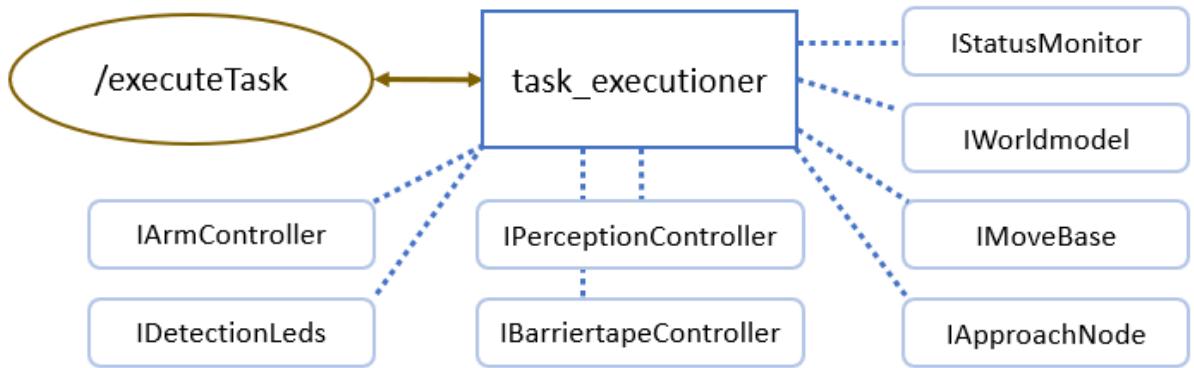


Abbildung 8.16: Task Executioner

Durch die stetige Entwicklung am Roboter kann es vorkommen, dass eine Komponente zwischenzeitlich nicht verwendet werden kann. Das ist z.B. der Fall, wenn der Arm gerade umgebaut wird, und daher nicht auf dem Roboter montiert ist. Damit der Ablauf von Aktionen bzw. das Roboterverhalten dann trotzdem getestet werden kann, ohne dass dafür eine neue Version des Task Executioners erstellt werden muss, ist für jedes kritische Interface ein Simulationsmodus implementiert. Dann werden keine Verbindungen zu externen Knoten (z.B. Arm Controller) aufgebaut, sondern lediglich der logische Ablauf nachgebildet (Kommando senden, warten, Aktion bestätigen). Der Simulationsmodus kann für jedes Interface über einen Parameter aktiviert werden, sodass ohne viel Aufwand zwischen verschiedenen Konfigurationen gewechselt werden kann. Dadurch kann die logische Integrität des „Wie?“ für einen Subtask auch ohne vollständige Hardware getestet werden, was das Debugging und die Suche nach einfachen Fehler im Vorfeld ermöglicht, wenn nötig sogar ganz ohne Roboter.

9 Gesamtsystem

Die Nodes aus Kapitel 5 bis 8 werden für verschiedene Einsatzzwecke zu Systemkonfigurationen zusammengeschalten. In den folgenden Abschnitten werden diese Konfigurationen grafisch dargestellt, wobei die einzelnen Topics nicht mehr betrachtet werden. Stattdessen werden die Verbindungstypen und -richtungen mit verschiedenfarbigen Pfeilen dargestellt. Grüne Pfeile markieren Topics, orangene Pfeile stehen für Transformationen, die roten Pfeile signalisieren Services und die goldenen Doppelpfeile zeigen die Verbindung zweier Nodes über eine Action an.

In den Darstellungen wurde überwiegend die Hierarchie der BCMD-Architektur (vgl. Abb. 4.1) eingehalten, sofern dies möglich war, ohne die gesamte Übersicht einer Grafik dadurch zu verschlechtern. Für Details zu einzelnen Nodes und deren Verbindungen wird an dieser Stelle auf den jeweiligen Abschnitt in den vorigen Kapiteln verwiesen.

9.1 Kartierung

Während der Kartierung der Arena wird der Roboter manuell gesteuert. Daher kommen die Bewegungskommandos aus den Joystickeingaben. Zur Kartenerstellung werden die gefilterten Laserscans fusioniert und zusammen mit der Odometrie des Plattformtreibers im „gmapping“-Node verarbeitet. Die fertige Karte kann abschließend über den „map_saver“-Node auf der Festplatte gespeichert werden. Die Nodes überhalb der lila gestrichelten Linie werden ausschließlich zur Kartierung verwendet, die Nodes unterhalb können auch in anderen Konfigurationen vorkommen.

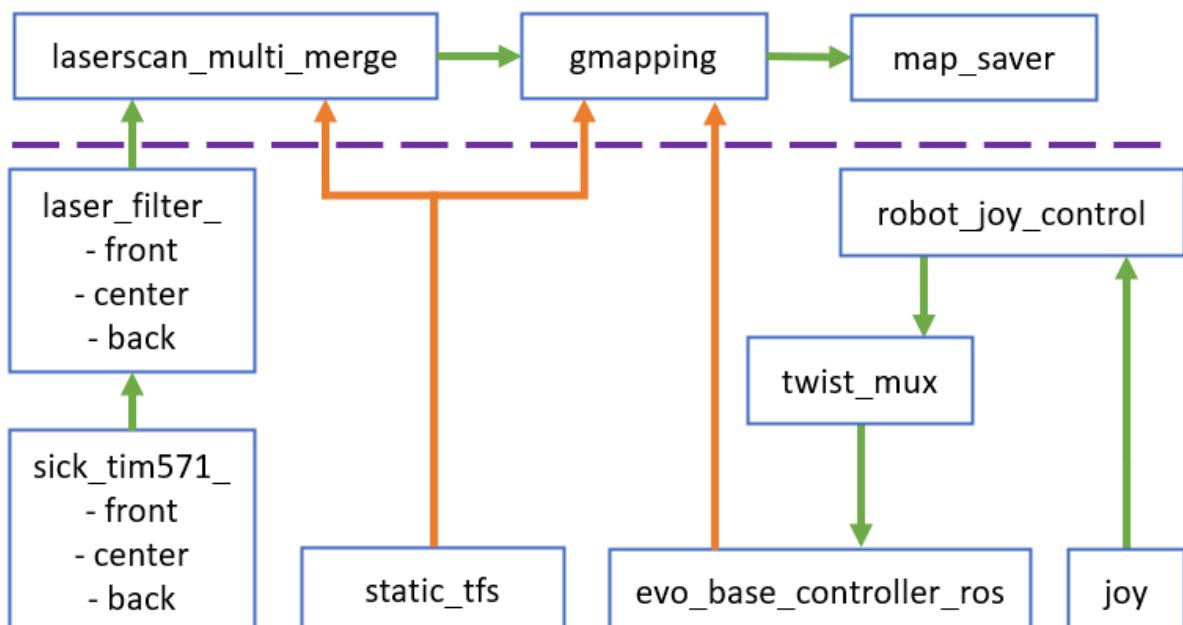


Abbildung 9.1: Systemkonfiguration für die Kartierung

9.2 Navigation

Für die Navigation werden die Laser- und Navigationskarte über „map_server“-Nodes von der Festplatte geladen und veröffentlicht. Die Lokalisierung kombiniert die fusionierten Laserscans und die Plattformodometrie zur Roboterposition innerhalb der Karte, die von „move_base“ zur Pfadregelung verwendet wird. Die Roboterposition wird außerdem vom Barriertape Controller für die Lokalisierung der erkannten Barriertapes innerhalb der Karte verwendet. Die Bilder hierfür kommen aktuell von einer einzelnen USB-Kamera, deren Bildformat an das neuronale Netz angeglichen wird. „move_base“ vereint die Navigationskarte, die Karte des Barriertapes und die Laserscans, um Hindernisse zu erkennen und zu vermeiden und so entsprechende Bewegungskommandos an den Plattformtreiber zu senden. Gestrichelte Pfeile dienen lediglich der Übersichtlichkeit und haben sonst keine zusätzliche Bedeutung zur ursprünglichen Legende.

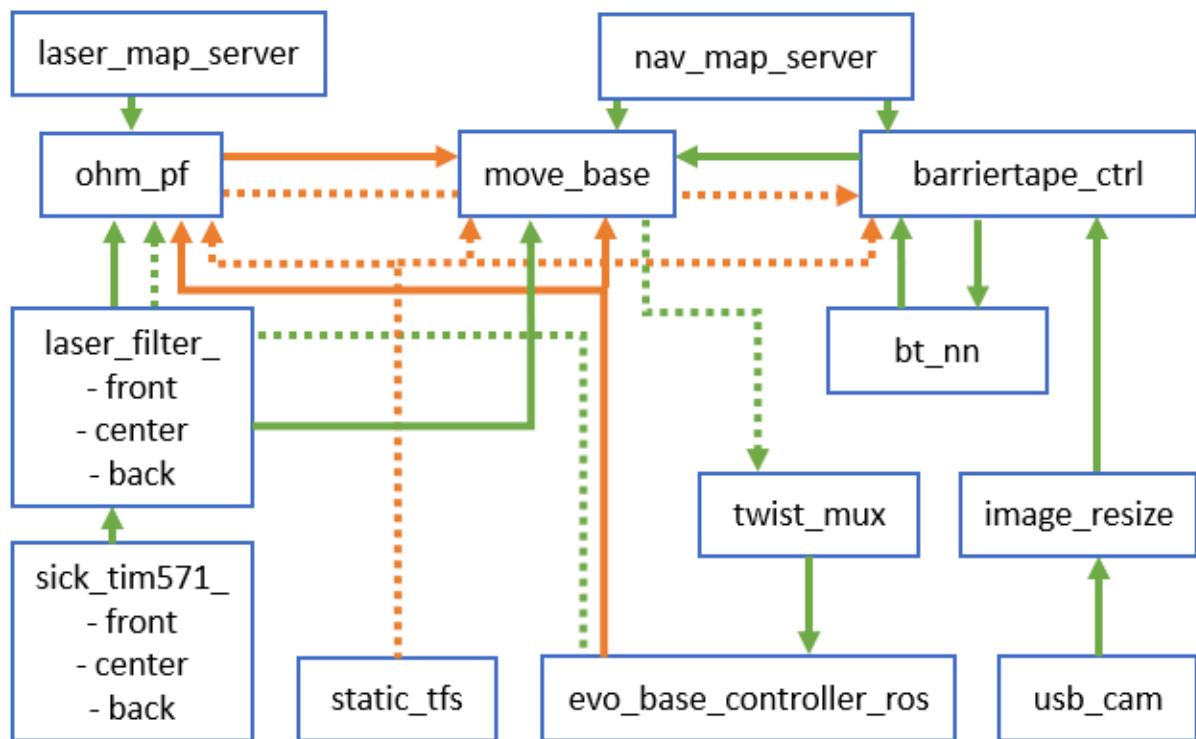


Abbildung 9.2: Systemkonfiguration für die Navigation

Diese Systemkonfiguration ist lediglich zum Testen von „move_base“ konzipiert. Es kann somit zwar die globale Navigation verwendet, jedoch keine Feinpositionierung vorgenommen oder Workstation angefahren werden.

Zudem ist die Steuerung durch den Joystick in der Grafik nicht mehr aufgeführt, auch wenn sie optional aktiviert werden kann.

9.3 Manipulation

Für die Manipulation müssen der Roboterarm samt Greifer gestartet werden. Der Arm vervollständigt dann die Transformationskette des Manipulators, sodass die Position der Realsense Kamera auf dem Roboter bestimmt werden kann.

Dann können die Messwerte der 3D-Kamera in den Arenakontext eingeordnet werden, sodass die Tischoberfläche einer Workstation bestimmt und dadurch auf die Position der Objekte rückgeschlossen werden kann, die mit dem neuronalen Netz erkannt wurden.

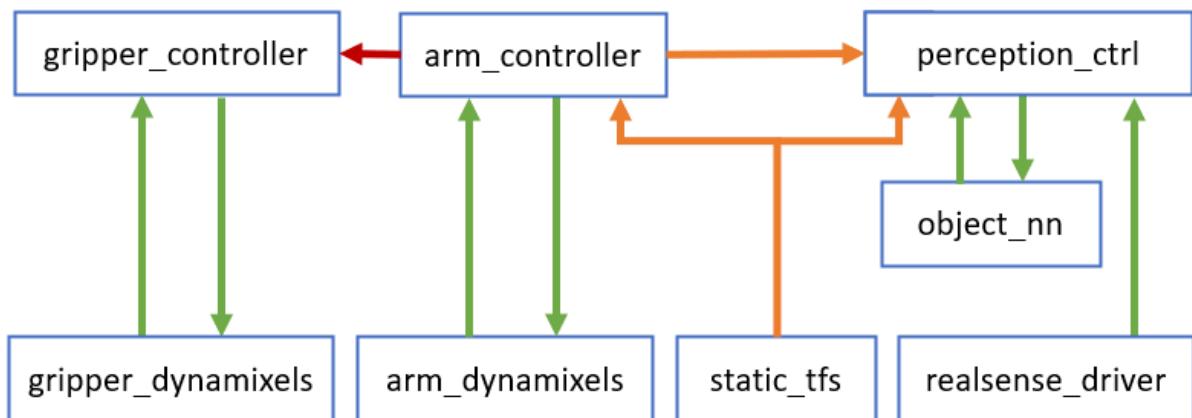


Abbildung 9.3: Systemkonfiguration für die Manipulation

9.4 Autonomer Betrieb

In Abbildung 9.4 ist die Systemkonfiguration für den autonomen Betrieb dargestellt, in der Navigation und Manipulation kombiniert und um die Brain-Ebene mit Worldmodel, Status Monitor, Task Manager und Task Executioner erweitert werden. Zur Visualisierung des aktuellen Roboterstatus wird außerdem die Unterbodenbeleuchtung gestartet, um das Roboterverhalten von Task Manager bzw. Executioner besser nachvollziehen zu können.

In dieser Konfiguration sind außerdem der Approach Controller zur Feinpositionierung und die Objektbeleuchtung enthalten, die im autonomen Betrieb bessere Rahmenbedingungen für die Manipulation bzw. Objekterkennung an einer Workstation schaffen.

Nicht enthalten sind die Nodes, mit denen der Roboter gesteuert werden kann. Diese sind die Joysticksteuerung (vgl. auch 9.2) und der Worldmodel Controller. Diese können bei Bedarf dazugeschalten werden, sodass der Roboter z.B. manuell vor die Workstations gefahren werden und so deren jeweilige Position eingespeichert werden kann. Sobald alle (verwendeten) Workstations in der Arena eingespeichert wurden, können über „robot_direct_command“ bzw. „atwork_commander“ Subtasks an den Task Manager geschickt und anschließend vom Task Executioner absolviert werden.

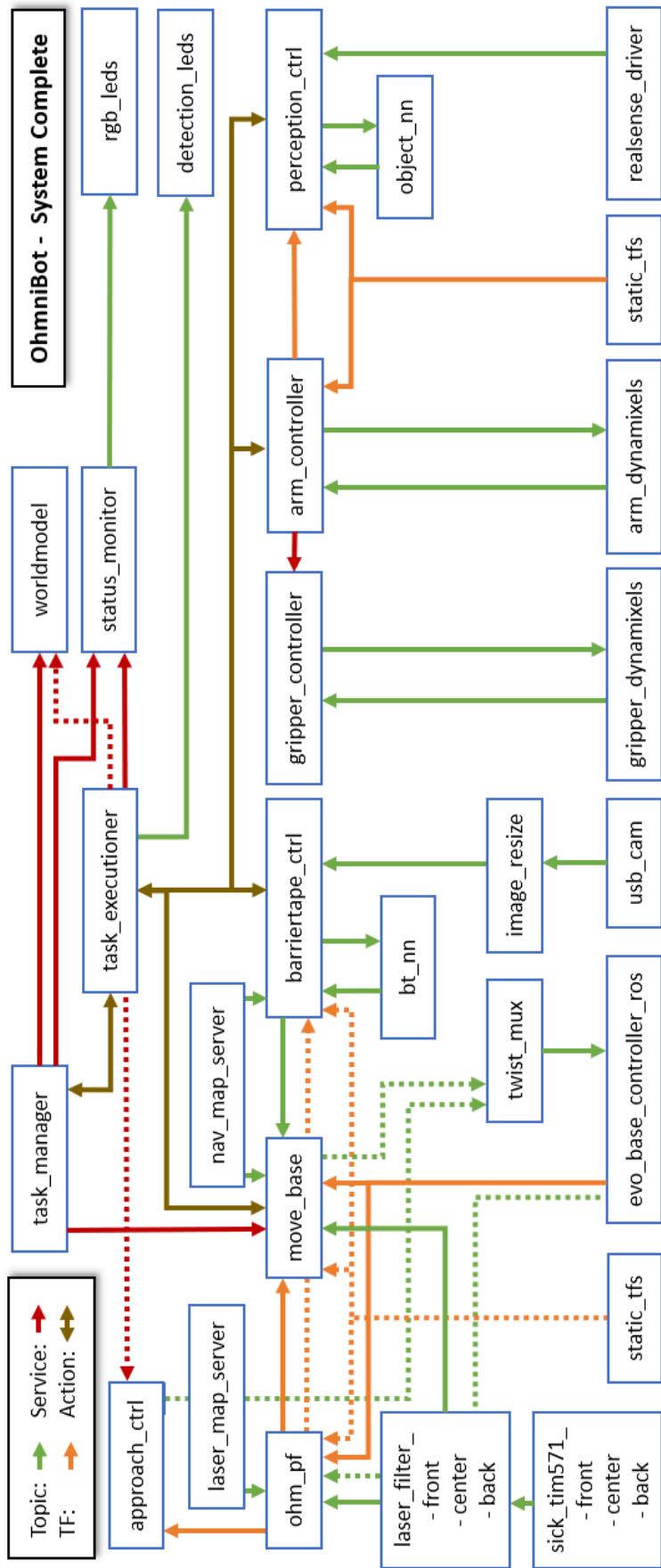


Abbildung 9.4: Systemkonfiguration für den vollautonomen Betrieb

9.5 Roboterverhalten an einem einfachen Beispiel

Die Abbildungen 9.5 und 9.6 zeigen das Roboterverhalten für den in Abb. 8.9 dargestellten Task mit drei Transportaufgaben. Durch die sechs resultierenden Subtasks kann der rekursive Algorithmus verwendet werden, der die optimale Reihenfolge der Subtasks ermittelt. Diese sind oben in schwarz dargestellt.

Bei einem fehlerfreien Durchlauf ergeben sich dadurch die farbig markierten States im Task Executioner. Die verwendeten Farben stehen dabei für den Farocode, der durch die Unterbodenbeleuchtung umgesetzt wird.

Zu Beginn muss zur ersten Workstation (WS3) navigiert werden und anschließend die genaue Position eingenommen werden. Von dort kann an die Workstation angedockt werden, sodass diese mit der 3D-Kamera gescannt und das Objekt (Obj1) gesucht werden kann. Nachdem dieses gefunden wurde, wird es mit dem Manipulator aufgenommen und im Inventar abgelegt.

Da an derselben Workstation ein zweites Objekt (Obj3) gegriffen werden soll, muss nicht gefahren werden. Im Optimalfall wurde dieses Objekt schon beim Suchen zuvor mitdetektiert, sodass direkt gegriffen werden kann.

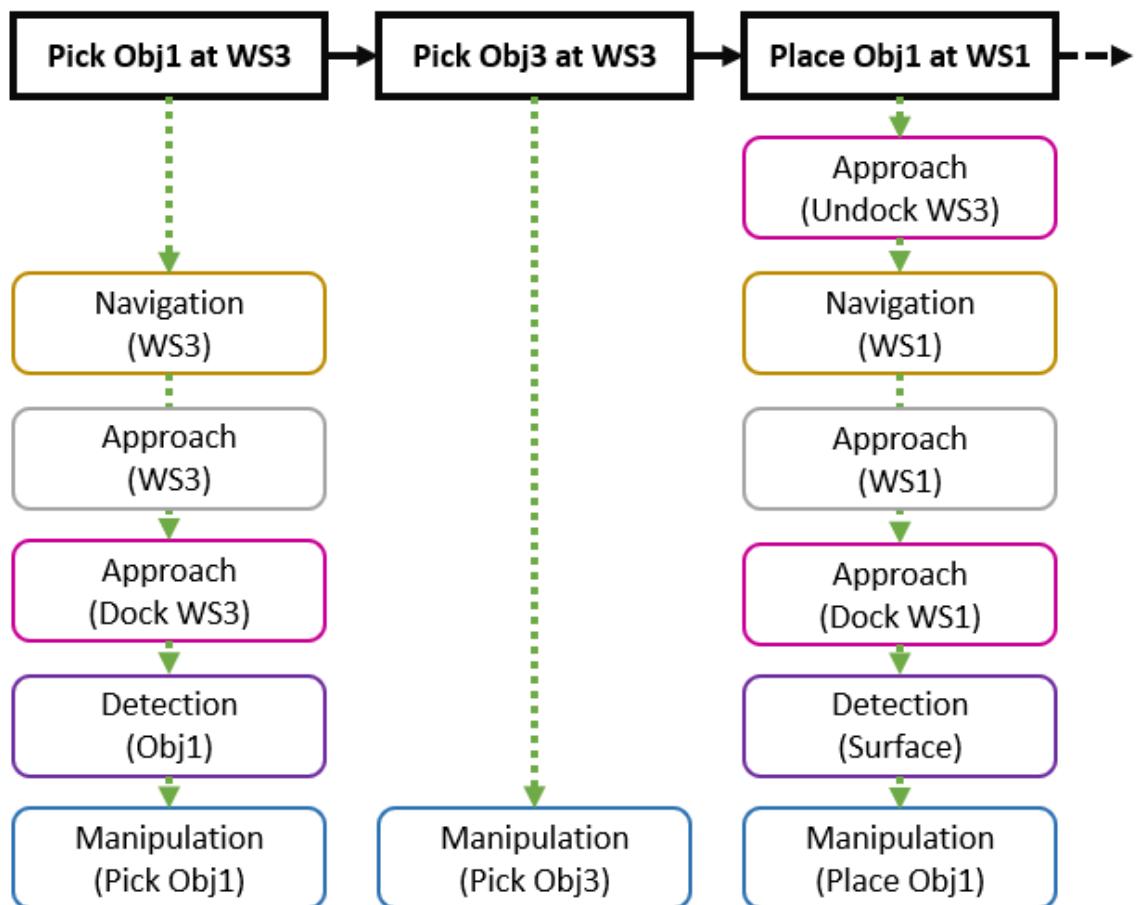


Abbildung 9.5: Roboterverhalten an einem Beispiel (Teil 1)

Damit die aufgenommenen Objekte abgelegt werden können, muss zunächst von der aktuellen Workstation (WS3) abgedockt werden, damit mit der freien Navigation zur nächsten Workstation (WS1) weitergemacht werden kann. An diese muss wieder angedockt werden. Damit das Objekt (Obj1) sicher abgelegt werden kann, muss zunächst die Tischoberfläche erkannt werden, da daraus die Ablegehöhe berechnet wird. Anschließend kann das Objekt aus dem Inventar gegriffen und auf der Workstation abgelegt werden.

Da bei der Detektion der Tischoberfläche nicht nach Objekten gesucht wird, muss für das Greifen des nächsten Objekts (Obj2) der Tisch erneut untersucht werden. Wenn das Objekt gefunden wurde, kann es gegriffen und ins Roboterinventar gelegt werden.

An der letzten Workstation (WS2) müssen die beiden Objekte (Obj2 und Obj3) nun noch abgelegt werden. Nach dem Navigationsprozess dorthin muss also lediglich die Tischhöhe bestimmt werden, damit beide Objekte abgelegt werden können.

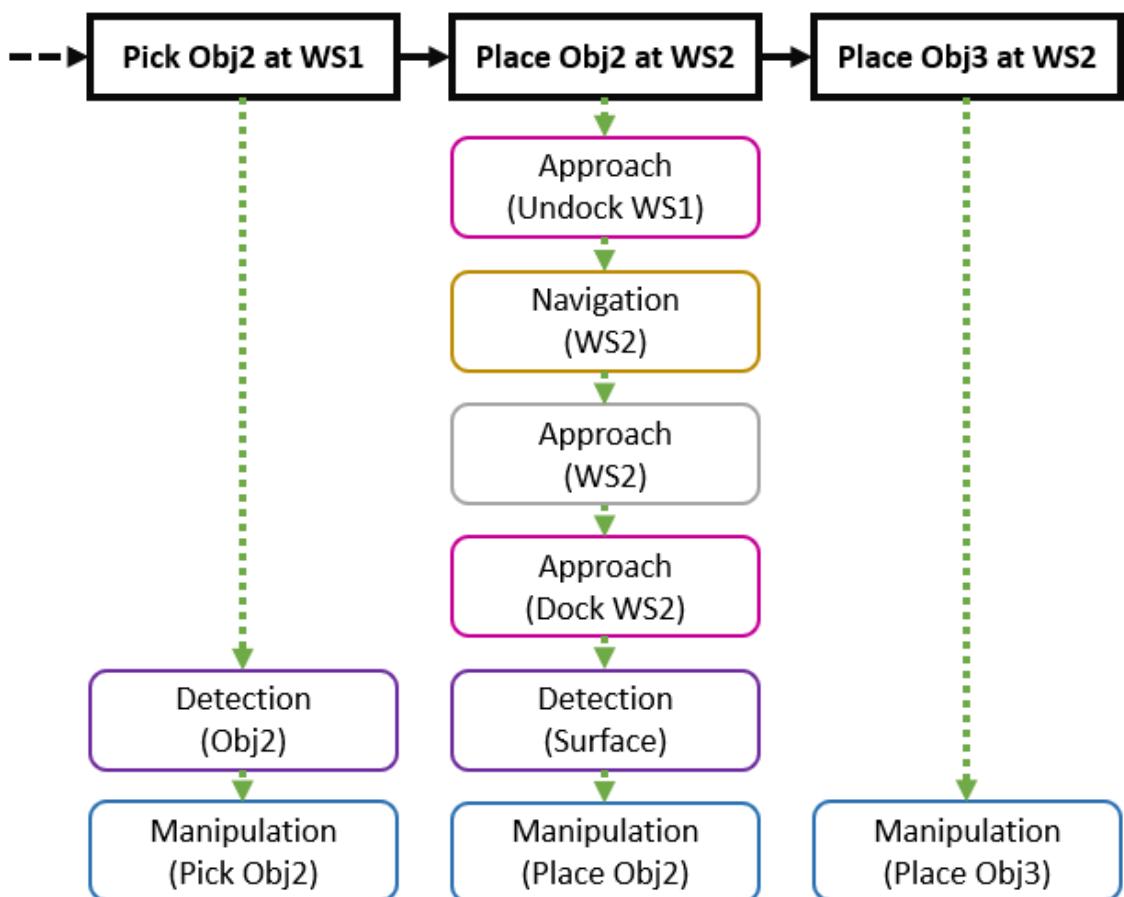


Abbildung 9.6: Roboterverhalten an einem Beispiel (Teil 2)

9.6 Offizieller Testlauf

Über den Link aus [41] kann ein Video aufgerufen werden, in dem ein offizieller Testlauf der @Work-Liga in der Arena im Labor zu sehen ist. Die Aktionen im Video ähneln denen aus Abb. 9.5 und 9.6, wobei die verschiedenen States über die Farbcodes identifiziert werden können.

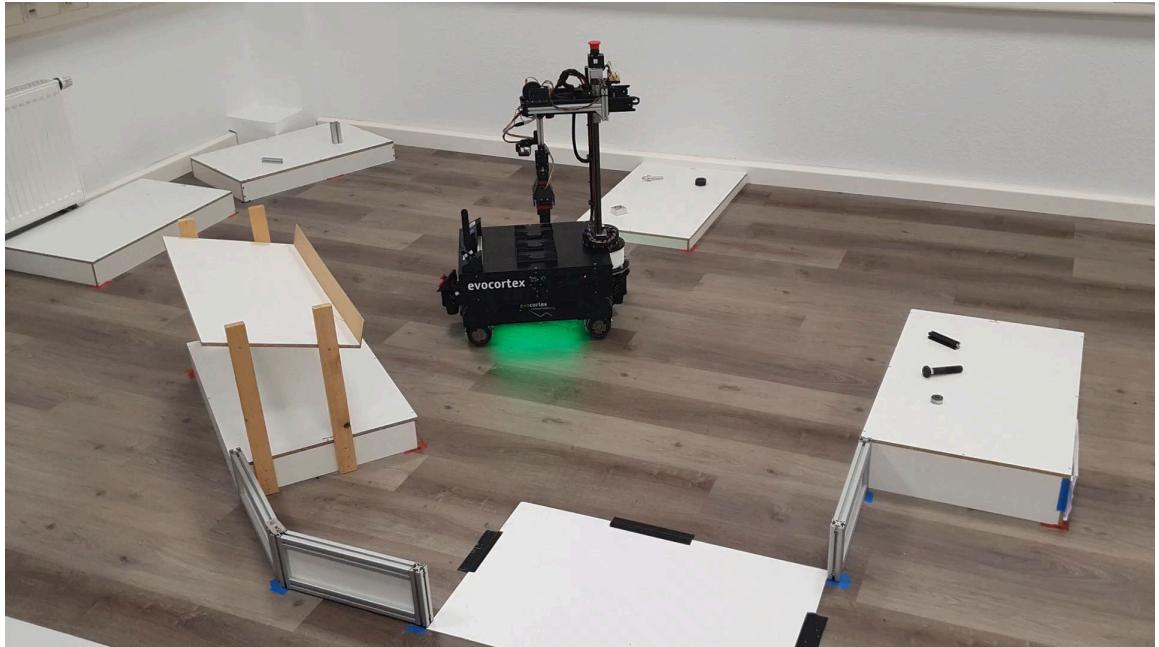


Abbildung 9.7: Testlauf: Umgebung

Der Roboter startet im Idle-Zustand in der Mitte der Tische, auf denen bereits die Objekte liegen, die transportiert werden sollen. Außerdem sind weitere Objekte auf den Workstations platziert, da diese den Roboter irritieren sollen.

Sobald der Ohmnibot den Task empfängt, beginnt er mit der Aufgabenplanung (Abb. 9.8 li.). Anschließend navigiert er zur ersten Workstation (Abb. 9.8 mitte) und fährt deren Position genau an (Abb. 9.8 re.).



Abbildung 9.8: Testlauf: Aufgabenplanung, Navigation und Feinpositionierung

Als Vorbereitung für die Manipulation dockt der Roboter anschließend an die Workstation an (Abb. 9.9 li.) und beginnt, diese zu scannen (Abb. 9.9 mitte). Das gesuchte Objekt (die Schraube) wird erkannt und anschließend gegriffen (Abb. 9.9 re.).

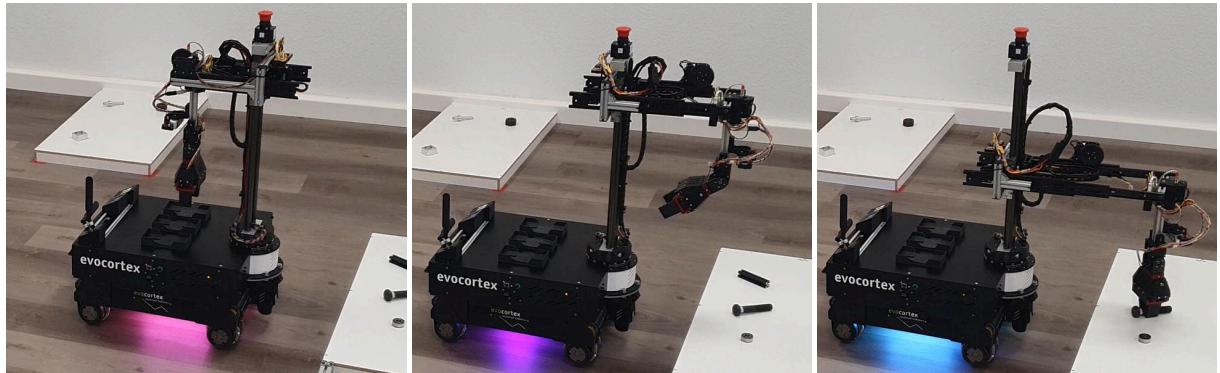


Abbildung 9.9: Testlauf: Andocken, Detektion und Greifen des Objekts

Das erfolgreich gegriffene Objekt wird dann im Inventar abgelegt (Abb. 9.10 li.). Da das zweite Zielobjekt (das Kugellager) bereits während der ersten Detektionsphase erkannt und lokalisiert wurde, kann dieses direkt gegriffen werden (Abb. 9.10 mitte), wodurch die Subtasks an dieser Workstation abgeschlossen sind und zur nächsten gefahren werden kann (Abb. 9.10 re.).

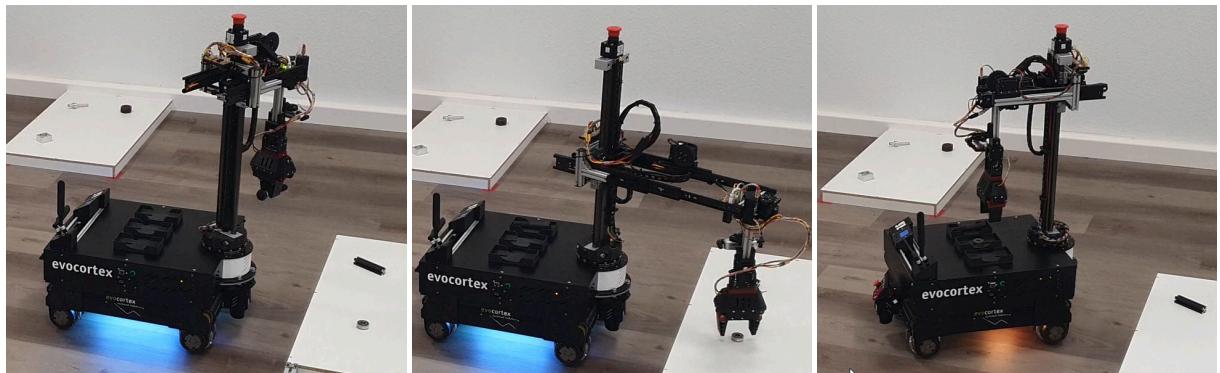


Abbildung 9.10: Testlauf: Ablegen des Objekts im Inventar, Greifen des zweiten Objekts und Navigation zur nächsten Workstation

Dort wird abgelegt und mit dem Rest der Subtasks weitergemacht. Diese Prozesse können im Video nachverfolgt werden, welches dieser Arbeit in reduzierter Auflösung angehängt ist.

10 Zusammenfassung und Ausblick

Der aktuelle Softwarestand, der im hochschuleigenen Repository liegt [42], ist gegenüber dem letzten Stand aus 2018 um ca. 2500 Commits weiterentwickelt worden. Über einen Entwicklungszeitraum von ca. 2 Jahren sind beinahe alle alten Nodes durch Neuentwicklungen ersetzt worden, die die Modularität des Systems erhöhen und vor allem im Bereich der Bildverarbeitung die Analyse von der Auswertung trennen, sodass bei zukünftigen Regeländerungen nicht mehr die komplette Objekterkennung ausgetauscht werden muss.

Die modulare Softwarearchitektur hat sich vor allem während der Covid-19 Pandemie und den damit einhergehenden Lockdown-Maßnahmen bewährt, da trotz des fehlenden Zungangs zum physischen Roboter mit der Simulation viel weiterentwickelt und verbessert werden konnte. Gerade die neue Art, die Subtasks anzunehmen und zu verarbeiten, ermöglichte es trotz des Rückstands in der Hardwareentwicklung große Teile der Navigation und Objekterkennung im Vorfeld zu testen. Durch den Simulationsmodus des Task Executio-ners konnten hier die Systemkomponenten deaktiviert werden, für die keine Simulation in Gazebo verfügbar war, und so das Roboterverhalten am Bildschirm optimiert werden.

Das neue Verhalten, welches deutlich übersichtlicher implementiert und außerdem durch die Unterbodenbeleuchtung für den Zuschauer einfacher verständlich ist, beweist im Video aus Kapitel 9.6, dass damit Aufgaben autonom ausgeführt werden können. Durch das neue Konzept der Rollenverteilung kann außerdem besser auf Fehler reagiert werden, sodass die Effizienz des Roboters stets optimiert werden kann.

Mit dem neuen Ohmnibot und der neuen Kontrollarchitektur konnte im Oktober 2020 der dritte Platz beim Virtual RoboCup Asia Pacific 2020 belegt werden. Das zeigt, dass das System potential hat, auch in Zukunft erfolgreich an den Wettkämpfen teilzunehmen. Für höhere Siegchancen wird das System stetig vom Team weiterentwickelt. Aktuell wird an einer 360° Kameraüberwachung gearbeitet, die bei der frühzeitigen Erkennung von Barriertapes helfen soll. Außerdem wird der Task Manager um ein Interface für eine Spracherkennung erweitert, sodass mehr Schnittstellen für die Mensch-Maschine-Interaktion zur Verfügung stehen.

Abbildungsverzeichnis

1.1	AutonOHM youbot 2018	1
1.2	Team AutonOHM 2020	3
1.3	Organigramm des Teams AutonOHM@Work - Hardware	3
1.4	Organigramm des Teams AutonOHM@Work - Software	4
3.1	Ohmnibot - Aktueller Stand	12
3.2	Hauptrechner im Ohmnibot	13
3.3	Ausgefahrener Roboterarm	14
3.4	Kippende Greiferfinger	14
3.5	Ohmnibot Unterboden	15
3.6	Laserscanbereich	15
3.7	Beispielmodi der Unterbodenbeleuchtung	16
3.8	Wettkampfarena im Labor	17
3.9	Simulation der Arena aus Sydney 2019	17
3.10	Simulierte Sensorik	17
4.1	Softwarearchitektur - BCMD	20
5.1	Treiber für die Laserscanner	21
5.2	Treiber für eine USB Kamera	22
5.3	Szene in der Simulation	23
5.4	Generiertes Kamerabild	23
5.5	Treiber für die 3D-Kamera	24
5.6	Treiber für die Dynamixel Motoren	24
5.7	Treiber für die Grundplattform	26
5.8	Treiber für die Kamerabeleuchtung	27
5.9	Treiber für die Unterbodenbeleuchtung	27
5.10	Statische Transformationen	28
5.11	Robotermodell samt Transformationen	28
6.1	Ungefilterte Laserscans	30
6.2	Gefilterte Laserscans	30
6.3	Laser Filter	30
6.4	Laserscan Multimerger	31
6.5	Gmapping	32
6.6	Originalkarte	32
6.7	Modifizierte Laserkarte	32
6.8	Ohm Partikelfilter	33
6.9	Lokalisierung	34

6.10 Globale Pfadplanung	35
6.11 Lokale Pfadplanung	36
6.12 Camera Image to PCL	37
6.13 Barriertape in der Simulation	37
6.14 Aus dem Kamerabild generierte Punktwolke	37
6.15 Fisheye Image Rectification	38
6.16 Bilder einer Fisheyekamera: Original (links) und Entzerrt (rechts)	38
6.17 Kamerabild mit Barriertape	39
6.18 Bildmaske	39
6.19 Barriertapeerkennung mit neuronalem Netz	39
6.20 Objekterkennung mit neuronalem Netz	40
6.21 Datengenerierung: Kamerabild (a) Gerendertes Bild aus Blender (b) mit Label (c)	40
 7.1 Steuerung mit einem PS3 Controller	41
7.2 Prioritätssteuerung der Bewegungskommandos	42
7.3 Barriertape Controller	43
7.4 Perception Controller	44
7.5 Laserkarte	45
7.6 Hinderniskarte	45
7.7 Navigation Controller	46
7.8 Approach Controller	47
7.9 Gripper Controller	48
7.10 Arm Controller	49
 8.1 Status Monitor	53
8.2 Worldmodel	53
8.3 Workstation Positionen	54
8.4 State Idle	55
8.5 State Input Preparation	55
8.6 Nearest Neighbor Algorithmus	57
8.7 Problemräume als Bubbles	59
8.8 Kombination der Bubbles zu Lösungen	59
8.9 Vergleiche der verschiedenen Planungsalgorithmen	60
8.10 Rekursive Baumsuche	62
8.11 State Replanning	63
8.12 State Execution	64
8.13 State Exit	65
8.14 Task Manager	66
8.15 Subtask Interpreter	67
8.16 Task Executioner	69

9.1	Systemkonfiguration für die Kartierung	70
9.2	Systemkonfiguration für die Navigation	71
9.3	Systemkonfiguration für die Manipulation	72
9.4	Systemkonfiguration für den vollautonomen Betrieb	73
9.5	Roboterverhalten an einem Beispiel (Teil 1)	74
9.6	Roboterverhalten an einem Beispiel (Teil 2)	75
9.7	Testlauf: Umgebung	76
9.8	Testlauf: Aufgabenplanung, Navigation und Feinpositionierung	76
9.9	Testlauf: Andocken, Detektion und Greifen des Objekts	77
9.10	Testlauf: Ablegen des Objekts im Inventar, Greifen des zweiten Objekts und Navigation zur nächsten Workstation	77

Literaturverzeichnis

- [1] “Robocup@Work Homepage,” 2021. [Online]. Available: <https://atwork.robocup.org/>
- [2] A. Norouzi, B. Schnieders, S. Zug, J. Martin, D. Nair, C. Steup, and G. Kraetzschmar, “Robocup@work 2019 - rulebook,” <https://atwork.robocup.org/rules/>, 2019.
- [3] M. Masannek, “Analysis of an existing AMR for Tasks required in Industry 4.0 Applications,” Forschungsbericht, Technische Hochschule Nürnberg, 2020.
- [4] M. Masannek, “Konzepterstellung und Inbetriebnahme eines optimierten Roboterprototyps für Aufgaben der Industrie 4.0,” Forschungsbericht, Technische Hochschule Nürnberg, 2021.
- [5] “Intersection of camera ray and 3D plane,” 2021. [Online]. Available: http://nghiaho.com/?page_id=363
- [6] J. Potthoff, “Konstruktive Optimierung eines Robotergreifers für autonome Manipulationsaufgaben im RoboCup@work-Wettbewerb,” Bachelorarbeit, Technische Hochschule Nürnberg, 2020.
- [7] “Innoq - Grundlagen für Softwarearchitektur,” 2021. [Online]. Available: <https://www.innoq.com/de/articles/2020/01/grundlagen-fuer-software-architektur/>
- [8] “Wikipedia - Interprozesskommunikation,” 2021. [Online]. Available: <https://de.wikipedia.org/wiki/Interprozesskommunikation>
- [9] “Sensortreiber für die LiDARs von SiCK,” 2021. [Online]. Available: https://github.com/uos/sick_tim
- [10] “Gazebo ROS Plugin Tutorials,” 2021. [Online]. Available: http://gazebosim.org/tutorials?tut=ros_gzplugins
- [11] “Treiber für USB Kameras (Github),” 2021. [Online]. Available: https://github.com/ros-drivers/usb_cam
- [12] “Image Pipeline (Github),” 2021. [Online]. Available: https://github.com/ros-perception/image_pipeline
- [13] “Kameratreiber für die Intel Realsense D435 (Github),” 2021. [Online]. Available: <https://github.com/IntelRealSense/realsense-ros>
- [14] “Treiber für Dynamixelmotoren (Github),” 2021. [Online]. Available: <https://github.com/ROBOTIS-GIT/dynamixel-workbench>

- [15] “Plattformtreiber für den EvoRobot (Github),” 2021. [Online]. Available: https://github.com/evocortex/evo_rd_platform
- [16] “Treiber für USB Mikrocontroller (Github),” 2021. [Online]. Available: <https://github.com/ros-drivers/rosserial>
- [17] “Laser Filters (Github),” 2021. [Online]. Available: https://github.com/ros-perception/laser_filters
- [18] “Gmapping (Github),” 2021. [Online]. Available: https://github.com/ros-perception/slam_gmapping
- [19] G. Grisetti, C. Stachniss, and W. Burgard, “Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling,” in *IEEE International Conference on Robotics and Automation (ICRA)*, 2005.
- [20] G. Grisetti, C. Stachniss, and W. Burgard, “Improving grid-based slam with rao-blackwellized particle filters by adaptive proposals and selective resampling,” in *IEEE Transactions on Robotics, Volume 23*, 2007, pp. 34 – 46.
- [21] “Ira Laser Tools (Github),” 2021. [Online]. Available: https://github.com/iralabdisco/ira_laser_tools
- [22] A. L. Ballardini and S. Fontana and A. Furlan and D. G. Sorrenti, “iralasertools: a ros laserscanmanipulation toolbox,” <https://arxiv.org/pdf/1411.1086.pdf>, 2014.
- [23] “ROS Navigation Stack (Github),” 2021. [Online]. Available: <https://github.com/ros-planning/navigation>
- [24] “GNU Image Manipulation Program (Website),” 2021. [Online]. Available: <https://www.gimp.org/>
- [25] “Ohm PF (Github),” 2021. [Online]. Available: https://github.com/amndan/ohm_pf
- [26] D. Ammon, “Entwicklung eines Lokalisierungssystems für fahrerlose Transportsysteme,” Masterarbeit, Technische Hochschule Nürnberg, 2017.
- [27] “MAS Navigation (Github),” 2021. [Online]. Available: https://github.com/b-it-bots/mas_navigation
- [28] “TEB Local Planner (Github),” 2021. [Online]. Available: https://github.com/rst-tu-dortmund/teb_local_planner
- [29] C. Rösmann and F. Hoffmann and T. Bertram, “Integrated online trajectory planning and optimization in distinctive topologies,” in *Robotics and Autonomous Systems, Vol. 88*, 2017, pp. 142 – 153.

- [30] C. Rösmann and W. Feiten and T. Wösch and F. Hoffmann and T. Bertram, “Trajectory modification considering dynamic constraints of autonomous robots,” in *Proc. 7th German Conference on Robotics, Germany, Munich*, 2012, pp. 74 – 79.
- [31] *Timed-Elastic-Bands for Time-Optimal Point-to-Point Nonlinear ModelPredictive Control*, 2015.
- [32] “Image Pipeline Fork by DavidTorresOcana (Github),” 2021. [Online]. Available: https://github.com/DavidTorresOcana/image_pipeline
- [33] *U-Net: Convolutional Networks for BiomedicalImage Segmentation*, 2015.
- [34] “U-Net Website,” 2021. [Online]. Available: <https://lmb.informatik.uni-freiburg.de/people/ronneber/u-net/>
- [35] “U-Net Keras Beispieldimplementierung (Github),” 2021. [Online]. Available: <https://github.com/pietz/unet-keras>
- [36] “U-Net Keras Beispieldimplementierung zum Trainieren (Github),” 2021. [Online]. Available: <https://github.com/zhixuhao/unet>
- [37] J. Redmon and A. Farhadi, “Yolov3: An incremental improvement,” *arXiv*, 2018.
- [38] “Blender Website,” 2021. [Online]. Available: <https://www.blender.org/>
- [39] “Joystick Drivers (Github),” 2021. [Online]. Available: https://github.com/ros-drivers/joystick_drivers
- [40] “Twist Mux (Github),” 2021. [Online]. Available: https://github.com/ros-teleop/twist_mux
- [41] “Video eines Basic Transportation Test im Labor,” 2021. [Online]. Available: https://thnuernbergde-my.sharepoint.com/:v/g/personal/masannekma61828_th-nuernberg_de/EUgLeGqAADVNkICBjhBIyn8Bsat9fwqrhK730et3hAH40Q?e=0RUb0D
- [42] “Ohm Atwork Repository (Github),” 2021. [Online]. Available: https://github.com/autonomohm/ohm_atwork