

BSDSAP - Projekt:
Entwicklung eines KI-gesteuerten autonomen Roboters
für die Navigation in einer häuslichen Umgebung

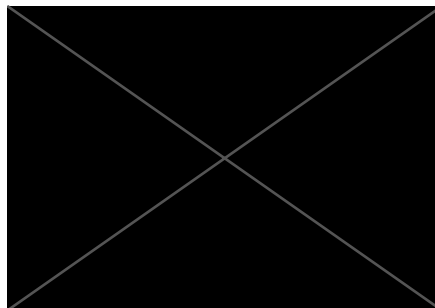
Projektbericht

eingereicht bei



von

Justin Stange-Heiduk



Datum: 20.06.2025

Abbildungsverzeichnis	ii
Abkürzungsverzeichnis	iii
1. Einleitung	1
2. Technische und methodische Grundlagen	2
2.1 Simulation in der Robotik: Isaac Sim und ROS2	2
2.2 Middleware-Kommunikation mit ROS2: Architektur und Rolle im System	3
2.3 Kartenerstellung und Autonome Navigation mit SLAM bzw. Nav2	4
2.4 Objekterkennung mit YOLO	5
3. Aufbau der Simulationsumgebung, des Roboters und der Sensorik	7
3.1 Erstellung der Wohnumgebung in Isaac Sim	7
3.2 Aufbau des Roboters	9
4. Action Graph und ROS2-Datenfluss	12
4.1 Konfiguration des Action Graphs in Isaac Sim	12
4.1.1 Action Graph: ROS2-Zeit	13
4.1.2 Action Graph: Transformationsdaten	14
4.1.3 Action Graphs: Sensordaten	15
4.1.4 Action Graphs: Steuerungslogik	16
4.2 ROS2-Kommunikation: Publizieren und Subscriben von Odometrie, Scan- und Bilddaten	17
4.3 Visualisierung in RViz2 und Validierung der Datenflüsse	20
5. Navigation und Objekterkennung im Systemkontext	22
5.1 Erstellung einer Umgebungskarte mit slam_toolbox	22
5.2 Nutzung der Karte zur autonomen Navigation mit Nav2	24
5.3 Integration der YOLO-Objekterkennung als Proof of Concept	25
6. Ergebnisse und Diskussion	26
6.1 Funktion und Qualität der Karte und Navigation	26
6.2 Funktion und Qualität der Navigation	26
6.3 Funktion und Qualität der Bildererkennung	27
6.4 Probleme und technische Einschränkungen	28
7. Fazit und Ausblick	30
Literaturverzeichnis	iii
Eidesstattliche Versicherung	v

Abbildungsverzeichnis

Abbildung 1: Vogelperspektive auf die in Isaac Sim modellierte Wohnumgebung	9
Abbildung 2: Frontansicht des simulierten Roboters in Isaac Sim	11
Abbildung 3: Transformationsstruktur	19
Abbildung 4: RVIZ	21
Abbildung 5: Karte durch SLAM in rviz2	23
Abbildung 6: Bilderkennung	25
Abbildung 7: Vergleich Erstellte Karte mit SLAM vs Original	26
Abbildung 8: Vergleich YOLO Inferenzschwelle 0.6 vs 0.3	28

Abkürzungsverzeichnis

ROS2	Robot Operating System 2
WSL	Windows-Subsysteme für Windows
DDS	Data Distribution Service
Nav2	Navigation 2
SLAM	Simultaneous Localization und Mapping
TF	Transformation Tree
AMCL	Adaptive Monte Carlo Localization
YOLO	You Only Look Once
QoS	Quality of Service
PoC	Proof of Concept

1. Einleitung

Autonome Robotersysteme gewinnen zunehmend an Bedeutung, insbesondere im Kontext häuslicher Assistenz, Überwachung und Unterstützung alltäglicher Aufgaben. Der Einsatz solcher Systeme verspricht nicht nur eine Entlastung bei wiederkehrenden Tätigkeiten, sondern bietet auch Potenzial für sicherheitsrelevante Anwendungen, etwa in der Pflege oder beim Monitoring von Haustieren. Damit ein mobiler Roboter jedoch zuverlässig in einer privaten Wohnumgebung agieren kann, muss er in der Lage sein, sich autonom zu orientieren, auf Hindernisse zu reagieren und relevante Objekte zu erkennen. Ziel dieses Projekts war es daher, eine Simulationsumgebung für ein solches System aufzubauen, die zentrale Aspekte der Lokalisierung, Navigation und visuellen Objekterkennung abbildet.

Die häusliche Umgebung stellt für autonome Systeme eine besondere Herausforderung dar: enge Raumverhältnisse, unstrukturierte Objekte, variable Lichtbedingungen erfordern eine hohe Anpassungsfähigkeit. In praktischen Entwicklungsprozess ist es notwendig, kostspielige Iterationen an realer Hardware zu vermeiden, insbesondere wenn Sensorik, Algorithmen und physische Einschränkungen noch nicht vollständig integriert sind. Eine realitätsnahe Simulationsumgebung stellt daher ein essenzielles Werkzeug zur prototypischen Entwicklung, Validierung und Fehleranalyse dar.

Im Rahmen dieses Projekts wurde ein virtueller, mobiler Roboter in einer digital nachgebildeten Wohnumgebung simuliert. Die technische Umsetzung erfolgte auf Basis der Simulationsplattform Isaac Sim, kombiniert mit dem Robotik-Framework ROS2. Die Simulation umfasste dabei sowohl die physikalische Modellierung eines Roboters mit zwei ansteuerbaren Rädern und einem Stützrad als auch die Integration von Sensoren wie LIDAR und einer RGB-Kamera. Ergänzend wurde die Navigationssoftware Nav2 zur autonomen Pfadplanung und das Deep-Learning-Modell YOLOv8m zur visuellen Objekterkennung verwendet. Die Datenübertragung zwischen den Modulen erfolgte über ROS2-Nodes und Action Graphs, wobei das System in zwei getrennten Anwendungsfällen getestet wurde: einmal zur Navigation, einmal zur Objekterkennung. Ein kombiniertes, multimodales Verhalten war in dieser Phase nicht implementiert, wurde jedoch als Erweiterungsperspektive identifiziert.

Der vorliegende Bericht gliedert sich in sieben Kapitel. Nach dieser Einleitung folgen in Kapitel 2 die methodischen und technischen Grundlagen, welche die Funktionsweise der eingesetzten Softwarekomponenten, Sensorik und Algorithmen beschreiben. Kapitel 3 erläutert den Aufbau der Simulationsumgebung, die Konstruktion des Robotermodells sowie die Integration der Sensorik. Kapitel 4 widmet sich dem Datenfluss und der Steuerungslogik, insbesondere der Konfiguration des Action Graphs und der ROS2-Kommunikation. Kapitel 5 behandelt die Navigation und die visuelle Objekterkennung als zentrale Anwendungsszenarien. Kapitel 6 diskutiert die erzielten Ergebnisse sowie erkannte Probleme und Einschränkungen. Das abschließende Kapitel 7 zieht ein Fazit und skizziert zukünftige Erweiterungsmöglichkeiten.

2. Technische und methodische Grundlagen

Ziel ist es die Entwicklung eines autonomen Roboters, der in einer häuslichen Umgebung navigieren und dynamisch auf Hindernisse reagieren kann. Um dieses Ziel umzusetzen, ist ein tiefes Verständnis der eingesetzten Technologien, Software-Frameworks und methodischen Ansätze notwendig. Die technische Umsetzung basiert nicht nur auf einer geeigneten Hardwareplattform, sondern erfordert insbesondere eine sorgfältig ausgewählte Simulationsumgebung sowie leistungsfähige Werkzeuge zur Lokalisierung, Kartenerstellung, Navigation und Objekterkennung.

Dieses Kapitel liefert die methodische und technologische Grundlage für das Projekt. Dazu werden zunächst die eingesetzte Simulationsplattform NVIDIA Isaac Sim und die unterstützende Hardwareumgebung beschrieben. Anschließend wird die Rolle von ROS2 als Middleware erläutert, gefolgt von einer detaillierten Vorstellung der beiden zentralen Module SLAM Toolbox und Nav2, die für die Kartierung und Navigation des Roboters verantwortlich sind. Den Abschluss bildet eine kompakte Darstellung des eingesetzten Objekterkennungssystems YOLO, das als Proof of Concept zur visuellen Erkennung von Objekten, insbesondere Katzen, integriert wurde.

Die in diesem Kapitel vorgestellten Grundlagen bilden das Fundament für die spätere Implementierung und Durchführung der Simulation. Sie verdeutlichen, welche technologischen Entscheidungen getroffen wurden, auf welchen Konzepten diese beruhen, und wie sie zusammenwirken, um die Anforderungen an ein autonom agierendes System im Innenraum zu erfüllen.

2.1 Simulation in der Robotik: Isaac Sim und ROS2

Die Simulation robotischer Systeme ist ein wesentlicher Bestandteil moderner Entwicklungsprozesse, da sie reale Testsituationen unter kontrollierten und reproduzierbaren Bedingungen ermöglicht. Für dieses Projekt fiel die Wahl auf Isaac Sim, eine Simulationsumgebung von NVIDIA, die speziell für die Entwicklung und das Testen autonomer Systeme entwickelt wurde. Die Integration mit ROS2 (Robot Operating System 2) erlaubt dabei eine realitätsnahe Steuerung und Kommunikation des simulierten Roboters mit typischen Softwarekomponenten aus der Robotik.

Isaac Sim basiert auf der NVIDIA Omniverse-Plattform, die als modulare Umgebung APIs, Dienste und Software Development Kits (SDKs) für industrielle Digitalisierungsprozesse bereitstellt. Sie ermöglicht die Entwicklung KI-gestützter Tools sowie die physikalisch akkurate Simulation komplexer 3D-Workflows in virtuellen Welten. Im vorliegenden Projekt wurde jedoch ausschließlich Isaac Sim als eigenständige Anwendung genutzt, sodass keine direkte Interaktion mit den weiterführenden Omniverse-Komponenten wie der Collaboration Engine oder dem Nucleus-Server notwendig war. (NVIDIA, 2025a) Isaac Sim unterstützt realistische Physiksimulationen, fotorealistische Darstellungen und bietet native Unterstützung für ROS2.

NVIDIA positioniert sich zunehmend als führender Akteur im Bereich KI-Infrastruktur und autonomer Systeme. Die Wahl von Isaac Sim ist daher nicht nur technisch, sondern auch strategisch begründet, da NVIDIA seine Plattform verstärkt auf Anwendungen in den Bereichen künstliche Intelligenz, digitale Zwillinge und synthetische Datengenerierung ausrichtet. (NVIDIA, 2025a)

Die Plattform kommt bereits in industriellen Kontexten zum Einsatz: So nutzt Amazon Robotics Isaac Sim in Kombination mit Adobe Substance 3D, um virtuelle Lagerumgebungen zu simulieren und logistische Prozesse zu optimieren. (NVIDIA, 2022) Auch die BMW Group zählt zu den Unternehmen, die auf die NVIDIA Omniverse-Technologie setzen, um digitale Zwillinge ihrer Produktionsstätten zu erstellen. In

einer Fallstudie beschreibt NVIDIA, wie BMW mit Hilfe der Plattform eine holistische Fabrikplanung realisiert, von der Robotersteuerung bis zur Layoutoptimierung, um Produktionsprozesse frühzeitig zu simulieren und zu verbessern (BMW Press, 2023).

Trotz der Stärken von Isaac Sim ist es wichtig, auch alternative Simulationsumgebungen kritisch zu betrachten. Gazebo, ein langjährig etabliertes Open-Source-Tool im ROS-Ökosystem, bietet eine breite ROS2-Unterstützung und eine aktive Community. Dennoch weist es Einschränkungen hinsichtlich Grafikqualität, Echtzeitverhalten und der Eignung für computer vision-basierte Forschung auf. Eine wissenschaftliche Vergleichsstudie, in der die Objektklassifikation in Gazebo und Isaac Sim untersucht wurde, zeigt, dass Gazebo die Erkennungsgenauigkeit signifikant überbewertet, während Isaac Sim in Bezug auf Detektionsleistung wesentlich realistischer mit echten Umgebungsbedingungen übereinstimmt. Auch wurde festgestellt, dass die Ergebnisse in Gazebo teils stark variieren, während Isaac Sim konsistenter und stabiler arbeitet. (Duarte et al., 2025, S. 339ff)

Die Installation und Ausführung von Isaac Sim erfolgte gemäß der offiziellen Dokumentation von NVIDIA (NVIDIA, 2025b) auf einem Windows-11-System. Eingesetzt wurde ein Desktop-PC mit einer NVIDIA RTX 5070 Ti (16 GB GDDR7 VRAM), einem AMD Ryzen 9 7900X, 48 GB DDR5-Arbeitsspeicher (6000 MHz) sowie einer 1 TB PCIe 4.0 NVMe SSD. Als Simulationsplattform kam Isaac Sim in Version 4.5.0 zum Einsatz. Isaac Sim lief dabei nativ unter Windows, während ROS2 Humble innerhalb einer WSL2-Umgebung (Windows-Subsystem für Linux) mit Ubuntu 22.04 betrieben wurde.

2.2 Middleware-Kommunikation mit ROS2: Architektur und Rolle im System

In modernen autonomen Systemen wie mobilen Robotern spielt die Middleware eine zentrale Rolle für die Kommunikation zwischen Sensoren, Aktoren und logischer Steuerung. Das in diesem Projekt eingesetzte Framework ROS2 stellt dabei die technische Grundlage für den gesamten Datenfluss zwischen den Komponenten dar.

ROS2 ist der direkte Nachfolger des weit verbreiteten ROS1 und wurde als vollständige Neuentwicklung konzipiert, um den gestiegenen Anforderungen moderner Robotik gerecht zu werden. Während ROS1 primär für akademische Prototypen gedacht war, wurde ROS2 für professionelle und industrielle Anwendungen entworfen, insbesondere mit Fokus auf Echtzeitfähigkeit, Skalierbarkeit, Multi-Roboter-Unterstützung und Plattformunabhängigkeit. Im Gegensatz zu ROS1, das auf einem zentralen Master-Knoten basierte, verwendet ROS2 den DDS-Standard (Data Distribution Service) als Kommunikations-Backbone. DDS ist ein industrieller Kommunikationsstandard, der den Austausch von Nachrichten über ein Publish-Subscribe-Modell regelt. Dadurch wird die Kommunikation robuster, ausfallsicherer und unabhängig von einem zentralen Knoten organisiert. (Liao, 2020)

Im vorliegenden Projekt war es notwendig, verschiedene Softwarekomponenten darunter Lidar, Kamera, Navigation (Nav2) und Mapping (SLAM Toolbox) miteinander zu verknüpfen. ROS2 bot dafür die ideale Infrastruktur, da (Liao, 2020):

- Echtzeitdaten (z. B. Lidar-Scans, Kamerabilder, Odometrie) zuverlässig zwischen Isaac Sim und Visualisierungstools wie RViz2 ausgetauscht werden konnten.
- Der modulare Aufbau eine klare Trennung zwischen Simulationsdaten, Steuerungslogik und Sensorik ermöglichte.
- Die DDS-basierte Kommunikation auch in der Verbindung zwischen Windows (Isaac Sim) und WSL2 (ROS2) reibungslos funktionierte.

- Der Navigationsstack Nav2 sowie SLAM Toolbox vollständig auf ROS2 aufbauen.

Um die Funktionsweise und Stärken von ROS2 in diesem Projekt besser zu verstehen, lohnt sich ein Blick auf die zentralen Konzepte, die diese Middleware auszeichnen und zu ihrer hohen Flexibilität und Robustheit beitragen.

Die wichtigsten Bausteine des ROS2-Kommunikationsmodells sind (Liao, 2020):

- Node: Eine eigenständige Funktionseinheit (z. B. Kamera- oder Navigationseinheit), die Daten sendet, empfängt oder verarbeitet.
- Topic: Kommunikationskanal für den asynchronen Nachrichtenaustausch nach dem Publish-Subscribe-Prinzip (z. B. /scan, /odom, /rgb).
- Publisher / Subscriber: Komponenten, die Nachrichten zu einem Topic senden bzw. empfangen.
- Service / Client: Synchrone Kommunikation für gezielte Anfragen (z. B. Karten-Reset, Bewegungskommandos).
- tf2: Transformationsframework zur Verwaltung der räumlichen Beziehungen zwischen Sensoren und Roboterteilen.
- QoS (Quality of Service): Feineinstellbare Parameter zur Steuerung der Kommunikationszuverlässigkeit, z. B. ob verlorene Daten erneut gesendet werden oder nicht.
- DDS Layer: Implementiert den Transport der Nachrichten über ein dezentrales, fehlertolerantes Netzwerk.

Durch dieses Architekturmodell ist ROS2 besonders gut geeignet für autonome Systeme, in denen viele Komponenten parallel arbeiten, miteinander kommunizieren und auf Sensorereignisse in Echtzeit reagieren müssen. Für das vorliegende Projekt war ROS2 daher unverzichtbar, um die Echtzeitintegration von Simulationsdaten, die Navigation auf einer SLAM-basierten Karte und die Visualisierung in RViz2 zu ermöglichen.

2.3 Kartenerstellung und Autonome Navigation mit SLAM bzw. Nav2

Ein zentrales Ziel dieses Projekts war es, dem Roboter die Fähigkeit zur autonomen Navigation in einer häuslichen Umgebung zu ermöglichen. Dazu mussten zwei essenzielle Teilfunktionen realisiert werden: die Erstellung einer Karte der Umgebung (SLAM) sowie die Pfadplanung und Hindernisvermeidung innerhalb dieser Karte (Navigation). Beide Aufgaben wurden mit Hilfe von ROS2-kompatiblen, modularen Werkzeugen umgesetzt: SLAM Toolbox für die Kartenerstellung und Nav2 für die Navigation.

Die ROS2-Architektur erlaubt eine strikte Trennung einzelner Funktionseinheiten, wie Kartenerstellung, Pfadplanung, Steuerung, Sensorik oder Datenvisualisierung. Diese Einheiten laufen als sogenannte *Nodes*, die über das DDS-basierte Kommunikationsmodell lose miteinander verbunden sind. Dies ermöglicht es, einzelne Module unabhängig zu konfigurieren, zu testen oder auszutauschen. Im vorliegenden Projekt wurde diese Modularität konsequent genutzt: SLAM Toolbox und Nav2 wurden unabhängig voneinander gestartet, kommunizierten jedoch über gemeinsame ROS2-Topics (z. B. /tf, /map, /scan).

Die SLAM Toolbox ist ein modernes ROS2-Paket zur Erstellung und Pflege von 2D-Karten aus Lidar-Daten. Sie verarbeitet LaserScan-Nachrichten und TF-Transformationen (odom → base_link), um eine konsistente, optimierte Karte aufzubauen. (Macenski & Jambrecic, 2021, S. 1) Im Unterschied zu älteren

SLAM-Verfahren wie GMapping oder Cartographer nutzt die Toolbox moderne Optimierungsverfahren (z. B. Google Ceres) und bietet erweiterte Funktionen wie (Macenski & Jambrecic, 2021, S. 3ff):

- Asynchronen Mapping-Modus, der Sensordaten auf best-effort-Basis verarbeitet
- Elastic Pose-Graph Localization, um sich an veränderte Umgebungen dynamisch anzupassen
- Pose-Graph-Manipulation zur manuellen Korrektur von Mapping-Fehlern
- Interaktives RViz-Plugin zur Visualisierung und Bearbeitung der Karte

SLAM Toolbox ermöglicht es, sowohl Kartenerstellung als auch Lokalisierung in einer bestehenden Karte (Mapping + Localization Mode) auszuführen. In diesem Projekt wurde der asynchrone Mapping-Modus mit dem Launch-File `online_async_launch.py` verwendet. Dieser Modus verarbeitet Sensordaten nur dann, wenn ausreichend Rechenressourcen verfügbar sind, anstatt jede eingehende Messung sofort zu verarbeiten. Dies erhöht die Systemstabilität bei gleichzeitig hoher Sensordatenfrequenz und reduziertem Ressourcenverbrauch. Die resultierende Karte wurde später in Nav2 eingebunden.

Nav2 ist das offizielle ROS2-Navigationsframework. Es wurde entwickelt, um autonome mobile Roboter in komplexen Umgebungen zuverlässig und sicher zu bewegen. Dabei kombiniert Nav2 verschiedene Subsysteme wie Lokalisierung, Pfadplanung, Bewegungssteuerung und Verhaltenssteuerung auf Basis sogenannter Behavior Trees. (Macenski et al., 2020, S. 1)

Nav2 kann (Macenski et al., 2020, S. 2-6):

- Den Roboter auf einer Karte lokalisieren (mithilfe von AMCL oder SLAM Toolbox)
- Pfade planen und dynamisch anpassen
- Hindernisse erkennen und vermeiden
- Komplexe Aufgaben über Verhaltenstemplates steuern
- Zwischenziele, Wegpunkte und Notfallverhalten definieren

Das Framework ist stark modularisiert: Einzelne Komponenten wie Planner, Controller, Recoveries oder Lifecycles laufen in separaten Nodes, die über ROS Actions und Services gesteuert werden. Diese Architektur erlaubt eine feingranulare Anpassung an verschiedene Robotermodelle und Aufgabenstellungen. Im Rahmen dieses Projekts wurde Nav2 mit dem Launch-File `navigation_launch.py` aus dem Paket `nav2_bringup` verwendet.

2.4 Objekterkennung mit YOLO

In autonomen Robotersystemen spielt die visuelle Objekterkennung eine zunehmend wichtige Rolle, insbesondere in komplexen, dynamischen Innenräumen. Während LIDAR-Sensoren zuverlässige geometrische Informationen liefern, können sie keine semantischen Kategorien unterscheiden: Ein Tier, ein Möbelstück oder ein Mensch sind für einen LIDAR-Punktwolkenalgorithmus zunächst gleichwertige Hindernisse. Kameras hingegen ermöglichen die Klassifikation und Unterscheidung solcher Objekte vorausgesetzt, die Bilddaten werden durch geeignete Verfahren ausgewertet.

Moderne Deep-Learning-Modelle zur Objekterkennung, wie die YOLO-Reihe (You Only Look Once), sind dafür besonders geeignet. Sie vereinen hohe Erkennungsgenauigkeit mit Echtzeitfähigkeit und eignen sich daher auch für eingebettete Systeme mit begrenzten Ressourcen. (Terven & Cordova-Esparza, 2024, S. 1) In diesem Projekt wurde YOLOv8m eingesetzt.

Im vorliegenden Projekt wurde YOLOv8m ausschließlich zur visuellen Objekterkennung in der Simulationsumgebung Isaac Sim verwendet. Ziel war es, einen Proof-of-Concept zur Integration visueller Sensorik zu realisieren. Dabei wurde ein Kamerastream aus Isaac Sim über ROS2 verarbeitet und mithilfe des Python-Pakets ultralytics analysiert. Die erkannten Objekte wurden mit Bounding Boxes überlagert und zur besseren Nachvollziehbarkeit visualisiert. Eine Interaktion mit Navigations- oder Lokalisierungsmodulen fand dabei nicht statt.

Die Entscheidung für das Modell YOLOv8m fiel aufgrund seiner ausgewogenen Eigenschaften zwischen Genauigkeit und Geschwindigkeit. YOLOv8 wurde am 10. Januar 2023 von Ultralytics veröffentlicht und stellt eine der derzeit modernsten Implementierungen der YOLO-Familie dar. Es kombiniert eine ankerfreie Architektur mit einem geteilten Ultralytics-Kopf, der präzisere Vorhersagen erlaubt. Insbesondere bei Echtzeitanwendungen wie der Robotik zeigt YOLOv8 durch seine modernisierte Backbone- und Neck-Struktur eine deutlich verbesserte Merkmalsextraktion. Die Version m („medium“) innerhalb der Modellreihe ist speziell für Anwendungen geeignet, bei denen ein guter Kompromiss zwischen Genauigkeit und Rechenaufwand gefragt ist. Das Modell erreicht laut Benchmarks auf dem COCO-Datensatz eine mittlere durchschnittliche Genauigkeit (mAP) von 50.2 % bei einer akzeptablen Inferenzzeit von 234.7 ms auf der CPU in ONNX. Mit 25.9 Millionen Parametern ist es wesentlich leichter als YOLOv8x oder -l, aber deutlich leistungsfähiger als n und s. Ein besonders interessanter Aspekt unseres Setups ist, dass wir die Bilddaten direkt aus der Isaac-Sim-Umgebung abonnieren konnten. Das heißt: Die Kamera unseres virtuellen Roboters liefert ein kontinuierliches RGB-Bild, das vom YOLO-Node verarbeitet wird, ohne weitere Brückenkonfiguration. Diese Kompatibilität zwischen Simulationsumgebung und Echtzeit-Objekterkennung ermöglicht einen nahtlosen Testablauf für spätere reale Einsätze. (Ultralytics, 2025)

Ein zentrales Merkmal des in diesem Projekt eingesetzten YOLOv8m-Modells ist, dass es auf dem weit verbreiteten COCO-Datensatz (Common Objects in Context) trainiert wurde. Dieser umfasst 91 Objektklassen, von denen 80 im Standardmodell von YOLOv8m für die Erkennung vorgesehen sind, darunter auch einige Möbelstücke wie Stühle, Sofas, Betten und Tische. Allerdings deckt der COCO-Datensatz nicht alle für die häusliche Robotik relevanten Objekte ab: Schränke, Fenster, Bücherregale, Näpfe oder spezifische Gegenstände wie Kratzbäume und Hanteln fehlen. (Lin et al., 2014, S. 6-7, 14)

3. Aufbau der Simulationsumgebung, des Roboters und der Sensorik

Nach der theoretischen und technologischen Fundierung in Kapitel 2 folgt nun die praktische Umsetzung der Simulation. Um ein autonomes Robotersystem realitätsnah testen zu können, bedarf es einer möglichst genauen digitalen Repräsentation der Zielumgebung sowie eines funktionsfähigen Robotermodells mit realitätsnaher Sensorik und Kinematik. Die Qualität dieser Modellierung ist ausschlaggebend für die Aussagekraft der späteren Ergebnisse im Bereich der Kartenerstellung und Navigation.

In diesem Kapitel werden daher die zentralen Elemente der Simulation systematisch aufgebaut. Zunächst wird die Erstellung einer physikalisch plausiblen Wohnumgebung in Isaac Sim erläutert. Anschließend folgt die technische Beschreibung des verwendeten Robotermodells, das auf dem bewährten TurtleBot3-Burger basiert. Dabei wird auf die Strukturierung des Roboters, die Integration von Gelenken und Sensoren sowie auf die manuelle Erweiterung durch zusätzliche Komponenten eingegangen.

Diese Modellierung bildet das Fundament für die weiteren Simulationsschritte, insbesondere für die Kartierung mit SLAM, die Navigation mit Nav2 sowie die visuelle Objekterkennung. Ohne eine konsistente physikalische Umgebung und einen korrekt konfigurierten Roboter sind reproduzierbare und realitätsnahe Tests in der Simulation nicht möglich.

3.1 Erstellung der Wohnumgebung in Isaac Sim

Um dem Roboter eine realistische Erkundungs- und Navigationsumgebung bereitzustellen, wurde in diesem Projekt eine maßstabsgetreue Nachbildung der eigenen Wohnung innerhalb von NVIDIA Isaac Sim erstellt. Der zentrale Grund für diese Entscheidung liegt in der Verwendung eines 2D-LIDARs, mit dessen Sensordaten eine Karte generiert werden soll.

Die digitale Rekonstruktion der Wohnung dient somit nicht nur der Visualisierung, sondern bildet die grundlegende Datenquelle für das Mapping: Alle erfassten LIDAR-Messpunkte spiegeln reale Abstände zu Wänden, Möbeln und sonstigen Strukturen wider. Durch die exakte Repräsentation dieser Strukturen im Simulationsraum kann der SLAM-Algorithmus eine präzise 2D-Occupancy-Map erzeugen, welche wiederum die Basis für die spätere autonome Navigation mittels Nav2 bildet. Zudem erlaubt die Simulation die vollständige Kontrolle über die Umgebung, wodurch Testreihen wiederholbar und systematisch durchgeführt werden können.

Zur Erstellung der Szene in Isaac Sim wurde zunächst eine neue Welt definiert, die als physikalische Grundlage für die Simulation dient. Dabei wurde eine Physik-Szene vom Typ „PhysicsScene“ eingebunden, welche für die Gravitation und physikalische Interaktionen innerhalb der Welt verantwortlich ist. Diese Szene sorgt unter anderem dafür, dass sich der Roboter realistisch verhält, beispielsweise bei Kollisionen oder dem Fahren über unebene Flächen.

Im nächsten Schritt wurde ein Umgebungslichtsystem hinzugefügt, um die Szene realistisch auszuleuchten. Dazu wurde ein XForm-Container „Environment“ angelegt, in dem ein Himmel vom Typ „DomeLight“ sowie eine gerichtete Lichtquelle („DistanceLight“) eingebunden wurden. Diese Kombination aus gleichmäßigem Umgebungslicht und direktem Licht gewährleistet eine natürliche Ausleuchtung der Szene, was vor allem für visuelle Sensoren wie Kameras relevant ist.

Die eigentliche Wohnung wurde anschließend modular aufgebaut und in vier Hauptkategorien unterteilt:

- Wände: Alle tragenden Strukturen, Zwischenwände und Raumgrenzen, modelliert als einfache Quader mit physikalischen Eigenschaften.
- Türen: Eingefügt als statische Objekte mit Öffnungsrahmen, jedoch in der Simulation nicht beweglich.
- Böden: Jede Fläche wurde mit realistischer Höhe, Textur und Kollisionseigenschaften versehen.
- Möbel: Platziert an den realen Positionen im Raum, ebenfalls mit physikalischen Eigenschaften zur Interaktion.

Diese Komponenten wurden über XForm-Nodes hierarchisch gruppiert und organisiert, um Übersichtlichkeit und einfache Bearbeitbarkeit zu gewährleisten. Die Struktur wurde so angelegt, dass spätere Ergänzungen unkompliziert möglich sind.

Die gesamte Szene wurde unter einem zentralen World-Node zusammengeführt. Diese Root-Instanz repräsentiert die Gesamtumgebung und stellt sicher, dass alle Bestandteile korrekt geladen und physikalisch miteinander interagieren können.

Für die physikalische Interaktion wurden die Wände, Türen und Böden jeweils mit einem Collider Preset versehen. Diese Einstellung erzeugt eine unsichtbare Kollisionshülle um die Objekte, die dafür sorgt, dass der Roboter z. B. nicht durch eine Wand oder Tür hindurchfahren kann und durch ein Lidar Sensor als Hindernis erkannt wird. Da diese Elemente in der Szene statisch bleiben, war kein zusätzlicher physikalischer Körper erforderlich.

Im Gegensatz dazu wurden die Möbelstücke nicht nur mit einem Collider Preset, sondern zusätzlich mit einem Rigid Body versehen. Dadurch erhalten sie physikalische Eigenschaften wie Masse, Trägheit und können im Bedarfsfall beweglich gemacht oder in dynamischen Simulationen einbezogen werden. Für das aktuelle Projekt bleiben die Möbel jedoch stationär. Der Rigid Body erlaubt hier vor allem realistischere Interaktionen mit dem Roboter, insbesondere bei Stößen oder engen Navigationsmanövern.

Die Möbel selbst wurden als 3D-Modelle im .obj-Format von verschiedenen kostenlosen Plattformen (wie z. B. Sketchfab oder CGTrader) heruntergeladen und in Isaac Sim importiert. Diese 3D-Modelle wurden in ihrer Position, Orientierung und Skalierung an die reale Wohnumgebung angepasst, um eine möglichst genaue Abbildung der späteren Einsatzumgebung des Roboters zu erhalten.



Abbildung 1: Vogelperspektive auf die in Isaac Sim modellierte Wohnumgebung

Die Abbildung zeigt die vollständig rekonstruierte Wohnung in Isaac Sim aus der Vogelperspektive. Zu erkennen sind alle wesentlichen räumlichen Strukturen der realen Wohnsituation, einschließlich mehrerer Zimmer, Türen, Möbel und Verbindungsgänge. Die Szene wurde in Anlehnung an den tatsächlichen Grundriss modelliert, wobei alle Elemente mit physikalischen Eigenschaften ausgestattet wurden, um realistische LIDAR-Messungen und Navigationsmanöver zu ermöglichen.

3.2 Aufbau des Roboters

Für die Navigation in der simulierten Wohnumgebung wurde ein mobiler Roboter benötigt, der in der Lage ist, LIDAR- und Kameradaten aufzunehmen, Bewegungen durchzuführen und in der ROS2-Umgebung vollständig adressierbar zu sein. Anstelle eines aufwendigen Eigenbaus wurde ein existierendes, bewährtes Robotermodell als Grundlage verwendet: der TurtleBot3 Burger von ROBOTIS. Dieses Modell ist ROS-kompatibel, gut dokumentiert und in Isaac Sim bereits als modular strukturierter URDF-Baum implementiert. Es eignet sich durch seine geringe Baugröße, sein Differentialantriebssystem und seine Unterstützung typischer Sensoren ideal für die Navigation in engen Innenräumen.

Die Robotermodellierung erfolgte als hierarchisch strukturierter XForm-Baum. Der Aufbau des Roboters gliedert sich wie folgt:

- `base_footprint`: Die unterste Referenzebene des Roboters, die als Wurzel im Transformationsbaum (TF) dient. Sie wird durch ein festes Gelenk (`base_joint`) mit dem eigentlichen Roboterkörper (`base_link`) verbunden.
- `base_link`: Der zentrale Strukturknoten des Roboters, an dem alle wesentlichen Komponenten befestigt sind. Dieser enthält:
 - Visuals und Collisions: Grafische Darstellung und Kollisionskörper zur physikalischen Interaktion.
 - Feste Gelenke (`PhysicsFixedJoint`) zu:
 - `caster_back_link`: Das hintere Stützrad mit passivem Rollkörper.

- imu_link: Träger für ein IMU-Modul (ohne Funktion in diesem Projekt, nur strukturell vorhanden).
- base_scan: Träger für den LIDAR-Sensor Visual.
- Drehgelenke (PhysicsRevoluteJoint) zu:
 - wheel_left_link und wheel_right_link: Die beiden Hauptantriebsräder des Differentialantriebs.
- caster_back_link: Enthält einen einfachen Cube-Kollisionskörper, der die passive Rolle des hinteren Stützrades modelliert.
- imu_link: Ein struktureller Platzhalter für inertielle Messeinheiten, die in dieser Arbeit jedoch nicht verwendet wurden.
- base_scan: Enthält standardmäßig ein LIDAR-Modul Visual und die Collision dafür. Die tatsächlichen 2D-Lidar und der RGB-Kamera Sensoren mussten manuell den Roboter hinzugefügt und konfiguriert werden. Die Kamera wurde hier erstmal auf Standardkonfiguration gelassen. Die Lidar relevanten Konfiguration wurde nach den technischen Details von SICK RPLIDAR A1, 360-Grad 2D-Laserscanner bearbeitet:
 - minRange = 0.15: Entspricht der minimalen Messdistanz des realen Sensors.
 - maxRange = 12.0: Maximale Reichweite gemäß technischem Datenblatt.
 - horizontalFov = 360: Simuliert ein vollständiges 360°-Scanfeld für Rundumsicht.
 - horizontalResolution = 0.5: Fein aufgelöste Winkelmessung für präzise 2D-Kartierung.
 - verticalFov = 0.0: Da es sich um einen 2D-LIDAR handelt, entfällt vertikale Erfassung.
 - verticalResolution = 1: Notwendig als Pflichtfeld, obwohl nur eine Ebene gescannt wird.
 - rotationRate = 10: Maximale Drehgeschwindigkeit in Hz, wie beim realen Sensor.
- wheel_left_link und wheel_right_link: Enthalten jeweils eigene Visual- und Collision-Meshes zur Simulation des Antriebssystems. Diese Komponenten sind direkt über rotierende Gelenke mit dem base_link verbunden.

Diese Struktur erlaubt es, den Roboter in Isaac Sim vollständig zu steuern und in ROS2 zu adressieren z. B. zur Visualisierung von Sensoren in RViz2 oder zur Einbindung in den Navigationsstack Nav2. Die gewählte Konfiguration bietet darüber hinaus eine hohe Flexibilität für spätere Erweiterungen, etwa durch zusätzliche Aktoren oder Sensoren, da das zugrunde liegende Modell modular aufgebaut ist.

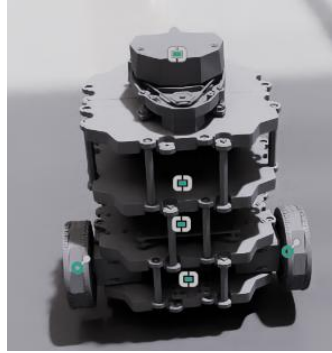


Abbildung 2: Frontansicht des simulierten Roboters in Isaac Sim

Zu sehen ist das auf dem TurtleBot3-Burger basierende Modell mit LIDAR-Sensor auf der obersten Ebene, den seitlich montierten Antriebsrädern und der mehrstufigen Aufbauweise zur Trennung von Elektronik, Sensorik und Mechanik.

4. Action Graph und ROS2-Datenfluss

Nach der Modellierung der physikalischen Umgebung und des Roboters in Kapitel 3 folgt in diesem Kapitel die Verknüpfung der Simulation mit dem ROS2-Ökosystem. Erst durch diese Integration wird es möglich, die in Isaac Sim erzeugten Datenströme für Kartenerstellung, Navigation und Objekterkennung nutzbar zu machen und sie mit realen ROS2-Komponenten zu verbinden.

Zentrales Bindeglied zwischen Simulation und ROS2 bildet der Action Graph, ein visuelles Programmierwerkzeug innerhalb von Isaac Sim. Über diesen lassen sich gezielt Ereignisse etwa Simulations-Frames mit ROS2-spezifischen Operationen verknüpfen. So können etwa Sensordaten publiziert, Steuerbefehle empfangen oder Transformationsdaten generiert werden.

In diesem Kapitel wird zunächst erläutert, wie über Action Graphs verschiedene ROS2-Kommunikationsflüsse eingerichtet wurden, darunter die Publikation der Simulationszeit (/clock), die Übertragung von Sensordaten (LIDAR, Kamera) sowie die Transformationen im TF-Tree. Anschließend wird beschrieben, wie unterschiedliche Steuerungsarten des Roboters, manuell über Tastatur oder automatisch über `cmd_vel`, über dedizierte Szenen und Action Graphs realisiert wurden.

Darüber hinaus wird dargelegt, wie die RViz2-Visualisierung und die Validierung der Topics sowie Transformationsdaten dazu beitrugen, die Korrektheit der Integration sicherzustellen. Die hier beschriebenen Konfigurationen sind entscheidend für den reibungslosen Ablauf der folgenden Kapitel zur Kartierung und Navigation. Sie gewährleisten, dass alle Komponenten im ROS2-Netzwerk in Echtzeit und korrekt miteinander interagieren.

4.1 Konfiguration des Action Graphs in Isaac Sim

Ein Action Graph ist in Isaac Sim ein visuelles Skripting-Werkzeug basierend auf dem OmniGraph-Framework. Er ermöglicht die Erstellung ereignisgesteuerter Abläufe, die gezielt auf Simulationsereignisse reagieren, beispielsweise auf jeden Simulations-Frame via dem Knoten „On Playback Tick“. (NVIDIA, 2025c)

In diesem Projekt wurden Action Graphs gezielt eingesetzt, um zentrale ROS2-Komponenten in der Simulation zu aktivieren und zu steuern. Konkret wurden damit folgende Funktionen realisiert:

- Publikation der Simulationszeit (Clock) an das ROS2-Netzwerk, sodass ROS2-Nodes synchron zur Simulation arbeiten konnten.
- Publikation der Transformationsdaten (/tf), um eine korrekte Einbindung des Roboters in den TF-Tree zu ermöglichen.
- Übertragung von Sensordaten:
 - LIDAR-Daten wurden aus dem simulierten Sensor an ein ROS2-Topic publiziert.
 - Kameradaten wurden analog verarbeitet und an ein ROS2-Topic publiziert.

Darüber hinaus wurden zwei Steuerungspfade über den Action Graph implementiert:

1. Manuelle Steuerung via Tastatur (WSAD):
Mittels eines *Keyboard Input Nodes* wurde eine einfache manuelle Steuerung des Roboters umgesetzt, um vor der Integration mit Nav2 erste Tests durchzuführen und für den YOLO PoC.

2. Autonome Steuerung über cmd_vel-Topic (Nav2):

Die vom Navigationsstack berechneten Geschwindigkeitsbefehle (geometry_msgs/Twist) wurden über einen ROS2 Subscribe Twist Node empfangen und mithilfe eines Differential Controller Nodes an den Roboter weitergeleitet.

Um verschiedene Steuerungskonzepte unabhängig voneinander testen und evaluieren zu können, wurden zwei separate Simulationsszenen in Isaac Sim erstellt:

1. Szene mit WSAD-Steuerung (scene_camera.usd):

Diese Szene diente primär der initialen Validierung der Roboterkinematik, der Kollisionsverarbeitung und der korrekten Einbindung von Sensorik. Über einen Action Graph wurde hier die Steuerung des Roboters direkt über die Tastatur realisiert, um Bewegungsabläufe und Sensorreaktionen in einer isolierten Testumgebung zu beobachten.

Auf Basis dieser Szene wurde die YOLO-basierte Objekterkennung eingesetzt. Da die gleichzeitige Publikation von LIDAR- und Kameradaten eine hohe Rechenlast und Datenaustausch verursachte, kam es in Kombination mit Nav2 zu Performanceeinbußen. Die Navigation war unter diesen Bedingungen nicht mehr zuverlässig durchführbar. Um die Bildverarbeitung dennoch unter realistischen Bedingungen evaluieren zu können, wurde in Szene ausschließlich der YOLO-Inferenz aktiviert. Der Roboter wurde manuell gesteuert, wodurch eine stabile Bilderkennung bei gleichzeitig reduzierter Systemlast möglich war.

2. Szene mit cmd_vel-Abonnent (scene_nav.usd):

In dieser Szene wurde der Roboter vollständig über ROS2 und den Nav2-Stack gesteuert. Der Action Graph empfing die von Nav2 berechneten Geschwindigkeitsbefehle über das ROS2-Topic /cmd_vel und leitete diese über einen Differential Controller Node an die Radantriebe weiter. Diese Szene bildete den zentralen Kern für alle Experimente zur autonomen Navigation auf Basis der zuvor erstellten Karte.

Beide Szenen wurden auf Basis desselben Wohnungsmodells realisiert und unterscheiden sich lediglich in der Konfiguration des Action Graphs. So konnte der Einfluss der Steuerungsart isoliert untersucht und gezielt zwischen manuellem und autonomem Betrieb gewechselt werden.

4.1.1 Action Graph: ROS2-Zeit

Damit externe ROS2-Nodes wie z. B. RViz2 korrekt mit der Simulationszeit arbeiten können, muss das Topic /clock regelmäßig mit der aktuellen Simulationszeit versorgt werden. In Isaac Sim wurde dafür ein spezieller Action Graph eingerichtet, der die Zeit bei jedem Simulationsframe publiziert. (NVIDIA, 2025d)

Folgende Nodes wurden verwendet (ebd.):

- On Playback Tick: Dieser Event-Node wird bei jedem Simulationsframe ausgelöst und dient als Trigger für nachgelagerte Aktionen.
- ROS2 Context: Stellt die Verbindung zwischen Isaac Sim und dem ROS2-Netzwerk her. Hier wird u. a. der Domain-ID-Kontext gesetzt.
- Isaac Read Simulation Time: Liest die aktuelle Simulationszeit aus dem Physik-Simulator. Optional kann die Zeit bei jedem Neustart der Simulation zurückgesetzt werden (resetOnStop).
- ROS2 Publish Clock: Sendet die gelesene Simulationszeit an das ROS2-Topic /clock.

Damit ROS2-Nodes diese Zeit nutzen, muss der Parameter use_sim_time (z. B. in Launch-Dateien oder per Kommandozeile) aktiviert sein.

4.1.2 Action Graph: Transformationsdaten

Zur Integration in das ROS2-Transformationssystem wurde ein eigener Action Graph erstellt, der kontinuierlich die Position und Orientierung des Roboters berechnet und in Form von Transformationsdaten (/tf) sowie Odometrieinformationen (/odom) publiziert. Dies ist insbesondere für die Lokalisierung, die Navigation mit Nav2 und das SLAM-Modul essenziell. (NVIDIA, 2025e)

Der Aufbau orientiert sich am offiziellen Isaac-Sim-Tutorial zur TF-Integration, wurde jedoch für das Projekt wie folgt angepasst (ebd.):

- On Playback Tick
- Context
- ReadSimTime: Synchronisation der Zeitstempel mit der Simulationszeit.

Transformationspfade:

- TFWorld2Odom
 - childFrameId = "odom"
 - parentFrameId = "world"
 - topicName = "tf"
 - Definiert den statischen Übergang von der globalen Welt in den Odometrie-Raum.
- TFOdom2Robot
 - childFrameId = "base_footprint"
 - parentFrameId = "odom"
 - topicName = "tf"
 - Repräsentiert die Bewegung des Roboters im Odometrie-Koordinatensystem.

Odometrie-Berechnung:

- ComputeOdometry
 - chassisPrim: Verknüpft mit dem Prim-Pfad zu base_footprint.
 - Berechnet die Odometrie auf Basis der Transformationen.
- PublisherOdometry
 - chassisFrameId = "base_footprint"
 - odomFrameId = "odom"
 - topicName = "odom"
 - Publiziert die resultierende Odometrie-Nachricht auf dem /odom-Topic.

Statische TF-Relationen:

- TFRobot
 - parentPrim: verweist auf base_footprint

- targetPrims: base_link und base_scan
- Erstellt die fixen Transformationsbeziehungen zwischen den statischen Elementen des Roboters.

Diese Konfiguration ist nicht nur grundlegend für Visualisierung und Steuerung, sondern auch essentiell für alle nachfolgenden ROS2-Module, insbesondere für die Kartenerstellung mit slam_toolbox und die Navigation mit Nav2. SLAM-Algorithmen benötigen ein korrekt definiertes TF-Netzwerk, um Sensordaten (z. B. LIDAR-Scans) räumlich richtig einordnen zu können. Ohne die klar strukturierte Transformation vom Weltkoordinatensystem bis hin zum Sensormodul würden sowohl Kartierung als auch Lokalisierung fehlschlagen. Daher bildet dieser Action Graph eine zentrale Grundlage für alle höherliegenden Navigationsfunktionen des Systems.

4.1.3 Action Graphs: Sensordaten

Damit ROS2-Nodes auf die in Isaac Sim erzeugten Bilddaten zugreifen können z. B. für die Visualisierung in RViz2 oder zur Objekterkennung durch ein YOLO-Modell wurde ein spezieller Action Graph eingerichtet, der RGB-Bilddaten in jedem Simulationsframe an ein ROS2-Topic publiziert. (NVIDIA, 2025f)

Folgende Nodes wurden verwendet (ebd.):

- On Playback Tick
- ROS2 Context:
- Isaac Create Render Product: Erzeugt ein Render Product aus dem angegebenen Kamera-Prim das als Quelle für die Bildausgabe dient.
- ROS2 Camera Helper: Definiert den Kameratyp (rgb), das ROS2-Topic (/rgb) sowie die zugehörige Frame-ID (base_scan). Intern erstellt dieser Node automatisch eine temporäre Post-Processing-Pipeline, die Bilddaten vom Renderer an das ROS2-Netzwerk überträgt.

Der Graph stellt sicher, dass bei jedem Frame ein aktuelles Kamerabild generiert und über das definierte Topic bereitgestellt wird. Die erzeugten Bilder konnten daraufhin über einen ROS2-Node empfangen und weiterverarbeitet oder visualisiert werden.

Da die Kamera-Pipeline zur Laufzeit dynamisch erstellt wird, ist sie nicht im USD-Stage-Baum sichtbar und wird nicht persistent gespeichert. Für den Einsatz im Projekt genügte dies jedoch vollständig zur Umsetzung eines stabilen Bilddatenstroms für die YOLO-Inferenz.

Zur Übertragung der simulierten LIDAR-Daten an das ROS2-System wurde ein Action Graph eingerichtet, der klassische LaserScan-Nachrichten publiziert. Dies ist insbesondere für SLAM, Navigation und Hinderniserkennung relevant.

Folgende Nodes wurden verwendet (NVIDIA, 2025g):

- On Playback Tick
- ROS2 Context
- Isaac Run One Simulation Frame: Führt einmalig bei Simulationsstart initiale Schritte aus, um z. B. das Render Product effizient zu initialisieren.

- Isaac Create Render Product: Verknüpft das RTX LIDAR-Prim mit einem Render-Ausgabekanal. Dieser dient als Eingang für die nachfolgenden Helper-Nodes.
- ROS2 RTX Lidar Helper (LaserScan): Publiziert die LIDAR-Daten als `sensor_msgs/LaserScan` auf ein definierbares ROS2-Topic (`/scan`). Die zugehörige Frame-ID wurde auf `base_scan` gesetzt.

4.1.4 Action Graphs: Steuerungslogik

Zur Integration mit dem ROS2-Navigations-Stack (z. B. Nav2) wurde in Isaac Sim ein Action Graph aufgebaut, der das Topic `/cmd_vel` abonniert. Darüber empfängt der Roboter kontinuierlich Bewegungsbefehle vom Typ `geometry_msgs/Twist`, die in Translation (linear) und Rotation (angular) unterteilt sind. Diese werden in Echtzeit verarbeitet und an einen Differential Controller übergeben, der das Fahrverhalten des Roboters steuert. (NVIDIA, 2025h)

Aufbau des Action Graphs:

- On Playback Tick
- ROS2 Context
- ROS2 Subscribe Twist: Abonniert das Topic `/cmd_vel` und stellt die empfangenen Geschwindigkeitsdaten (linear und angular) zur Verfügung.
- Break Vector3 (linear): Zerlegt die lineare Komponente des Twist-Messages. Die x-Achse (Vorwärts-/Rückwärtsbewegung) wird extrahiert und direkt an den Differential Controller weitergeleitet.
- Break Vector3 (angular): Zerlegt die Rotationskomponente und extrahiert die z-Achse (Rotation um die Hochachse).
- Constant Float: Dient zur Skalierung der Drehgeschwindigkeit. So kann z. B. die maximale Rotationsgeschwindigkeit des Roboters angepasst werden.
- Multiply: Multipliziert die extrahierte `angular.z` mit dem konfigurierten Skalierungsfaktor aus dem Constant Float Node. Dadurch wird nur die Drehbewegung (nicht die Translation) angepasst.
- Differential Controller: Dieser Node erhält sowohl:
 - die nicht skalierte lineare Geschwindigkeit (`linear.x`), und
 - die skalierte Rotationsgeschwindigkeit (`angular.z`).
 - Daraus wird dann die Motorbefehle für das differenzialgesteuerte Fahrwerk berechnet und ausgeführt.

Damit ist eine nahtlose Integration des Roboters in das ROS2-Navigationssystem gegeben. Der Roboter reagiert in Echtzeit auf von Nav2 berechnete Steuerkommandos und bewegt sich entsprechend durch die simulierte Umgebung.

Zur manuellen Steuerung des Roboters wurde ein separater Action Graph implementiert, der Tastatureingaben auswertet und in Bewegungsbefehle übersetzt. Dies ermöglichte die frühe Validierung der Roboterkinematik sowie Tests der Sensorintegration, unabhängig vom ROS2-Stack.

Zentrale Komponenten:

- W / A / S / D: Diese Nodes registrieren den Zustand der jeweiligen Tastaturtaste (vorwärts, links, rückwärts, rechts).
- ToDoubleX: Wandelt das Tastensignal in einen numerischen Wert um.
- AddLinear / AddAngular: Kombiniert die Richtungseingaben zu einem vollständigen Bewegungsbefehl (Translation & Rotation).
- Negate / Scale: Dienen zur Umkehrung oder Begrenzung der Eingabewerte (z. B. für Rückwärtsfahrt).
- Articulation Controller: Steuert direkt die Gelenke bzw. Antriebseinheiten des Roboters basierend auf den berechneten Bewegungsvektoren.

4.2 ROS2-Kommunikation: Publizieren und Subscriben von Odometrie, Scan- und Bilddaten

Nachdem die einzelnen Sensoren und Steuerungskomponenten in Isaac Sim über Action Graphs mit ROS2 verbunden wurden, war die Überprüfung der tatsächlichen Datenkommunikation ein wichtiger Schritt. Dabei wurde kontrolliert, ob die erwarteten Nachrichten im ROS2-Netzwerk verfügbar sind, korrekt publiziert werden und die Zeitsynchronisation funktioniert.

Bevor die Kommunikation mit ROS2-Nodes erfolgen kann, muss zunächst die Simulation in Isaac Sim gestartet werden. Nur bei laufender Simulation werden die Action Graphs ausgeführt und die entsprechenden ROS2-Topics aktiv bedient.

Im Anschluss daran erfolgt der Wechsel in ein Terminal unter Ubuntu 22.04 (z. B. über WSL2), in dem die ROS2-Umgebung initialisiert wird. Dabei sind folgende Befehle notwendig:

- `chmod 0700 /run/user/1000/`
 - Stellt sicher, dass der ROS2-Daemon korrekt auf das Nutzerverzeichnis zugreifen kann (insbesondere unter WSL2).
- `source /opt/ros/humble/setup.bash`
 - Lädt die ROS2-Umgebungsvariablen für ROS Humble.
- `ros2 topic list`
 - Lässt sich anschließend überprüfen, ob die Simulation erfolgreich ROS2-Topics veröffentlicht.

Daraufhin sollten alle relevanten Topics in der YOLO Szene angezeigt werden:

- `/clock`: Simulationszeit
- `/tf`: Transformationsdaten
- `/odom`: Odometrie
- `/rgb`: Kamera
- `/parameter_events`: Parameterüberwachung
- `/rosout`: Konsolenlog

In der Navigations Szene gibt es außerdem noch:

- `/scan`: Lidar
- `/cmd_vel`: Bewegungsbefehle

Um die Inhalte zu prüfen, wurden unter anderem folgende Kommandos verwendet:

- `ros2 topic echo /scan`: zeigt die Rohdaten des LIDAR in Textform. Es sollten kontinuierlich Datenpakete erscheinen, die jeweils eine Liste von Distanzwerten (float-Array) enthalten. Diese zeigen den 2D-Umriss der Umgebung aus Sicht des LIDAR. Ein Ausbleiben oder eine Null-Liste würde auf ein Problem mit der Sensorintegration oder der Positionierung hindeuten.
- `ros2 topic echo /odom`: überprüft die Bewegungsdaten (Position, Geschwindigkeit). Die Odometrie-Nachrichten sollten sich bei jeder Bewegung des Roboters aktualisieren und sowohl Position (pose) als auch Geschwindigkeit (twist) enthalten. Wenn der Roboter sich bewegt, sollten sich insbesondere die x- und y-Werte innerhalb von pose und twist.linear verändern. Bei einem Stillstand sind stabile Werte zu erwarten.
- `ros2 topic hz /rgb`: misst die Frequenz der Kameraübertragung (wichtig für YOLO). Die Frequenz der Bildübertragung sollte stabil je nach Simulationsgeschwindigkeit und Renderleistung liegen. Ein Wert von 0 Hz oder eine stark schwankende Frequenz würde auf Probleme im Action Graph, der Renderpipeline oder der Kamera-ID hindeuten.

Diese Prüfungen dienten nicht nur zur Validierung der Funktionsfähigkeit, sondern halfen auch bei der Fehlersuche beispielsweise bei Performanceproblemen durch hohe Bildraten oder falsch gesetzten Frame-IDs.

Mithilfe des Befehls `ros2 topic list` konnte die Gesamtheit aller verfügbaren Topics im laufenden ROS2-System überprüft werden nachdem die Simulation in Isaac Sim gestartet wurde. Dabei war insbesondere relevant, ob die folgenden Topics erfolgreich publiziert wurden:

- `/clock`: Simulationszeit (vgl. Abschnitt 4.1.1)
- `/tf` und `/tf_static`: Transformationsdaten (vgl. 4.1.2)
- `/odom`: Odometrieinformationen
- `/scan`: LIDAR-Daten
- `/rgb`: Kamerabilder (für YOLO)
- `/cmd_vel`: (in Szenen mit Nav2: Eingangs-Topic für Steuerung)

Anzeigen und Überprüfen einzelner Topics

Um die Inhalte zu prüfen, wurden unter anderem folgende Kommandos verwendet:

- `ros2 topic echo /scan`: zeigt die Rohdaten des LIDAR in Textform
- `ros2 topic echo /odom`: überprüft die Bewegungsdaten (Position, Geschwindigkeit)
- `ros2 topic hz /rgb`: misst die Frequenz der Kameraübertragung (wichtig für YOLO)

Diese Prüfungen dienten nicht nur zur Validierung der Funktionsfähigkeit, sondern halfen auch bei der Fehlersuche beispielsweise bei Performanceproblemen durch hohe Bildraten oder falsch gesetzten Frame-IDs.

Ein sehr wichtiger Punkt war die Überprüfung der TF-Struktur. Ein zentrales Werkzeug zur Überprüfung der Transformationsstruktur war das Tool `tf2_tools`:

- `ros2 run tf2_tools view_frames`
 - Es erstellt ein PDF-Diagramm der aktuell bekannten Transformationsbeziehungen (TF-Tree).

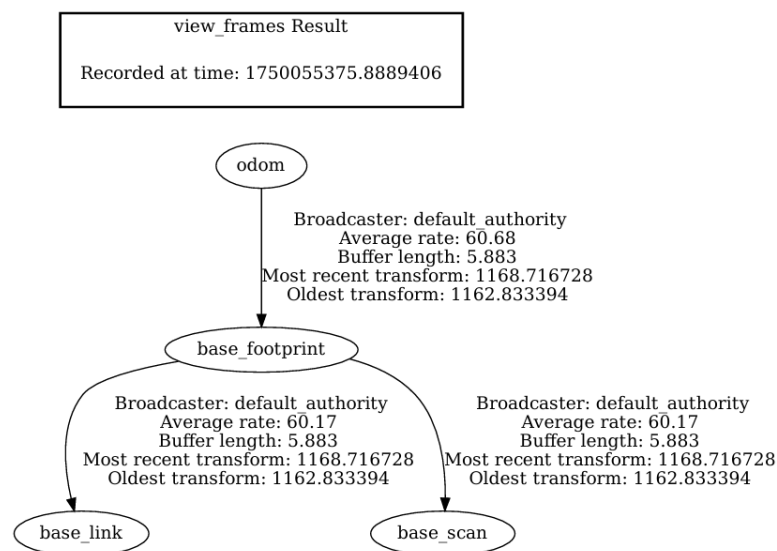


Abbildung 3: Transformationsstruktur

Abbildung 3 zeigt das Ergebnis des ROS2-Kommandos `ros2 run tf2_tools view_frames`, das eine Visualisierung des aktuellen Transformationsbaums (TF-Tree) generiert. Die Struktur stellt dar, wie die einzelnen Koordinatenrahmen (Frames) des Roboters zueinander in Beziehung stehen und wie die Transformationen in der Simulation korrekt weitergegeben werden.

Im dargestellten TF-Baum ergibt sich die folgende Hierarchie:

- Der Ursprung ist der `odom`-Frame, der die geschätzte Position des Roboters im Raum über die Zeit beschreibt.
- An `odom` ist der `base_footprint`-Frame gekoppelt, der die projizierte Aufstandsfläche des Roboters auf den Boden repräsentiert.
- Von `base_footprint` zweigen zwei weitere Frames ab:
 - `base_link`: die zentrale mechanische Referenz des Robotermodells, meist Mittelpunkt der Sensorik und Dynamik.
 - `base_scan`: die Position des LIDAR-Sensors, der für SLAM und Navigation essenziell ist.

Jeder Knoten zeigt darüber hinaus wichtige Diagnoseinformationen wie:

- Broadcast-Rate (z. B. ~60 Hz): wie häufig der Transform veröffentlicht wird.
- Buffer-Länge: wie lange vergangene Transformationsdaten vorgehalten werden.
- Zeitstempel der aktuellsten und ältesten Transform.

Die Struktur entspricht exakt den Erwartungen für ein korrekt konfiguriertes Mobilrobotersystem mit LIDAR-Sensorik und ist damit grundlegend für die fehlerfreie Funktion von SLAM, Nav2 und RViz2. Besonders wichtig ist dabei die konsistente Weitergabe der Transformationen entlang der Kette, um Sensor- und Steuerdaten korrekt räumlich einordnen zu können.

4.3 Visualisierung in RViz2 und Validierung der Datenflüsse

Nach erfolgreicher Überprüfung der aktiven ROS2-Topics sowie der Transformationshierarchie erfolgte im nächsten Schritt die visuelle Validierung in RViz2, dem zentralen Visualisierungswerkzeug im ROS2-Ökosystem. Ziel war es, die Datenflüsse aus der Simulation nicht nur technisch, sondern auch inhaltlich und räumlich korrekt darzustellen.

Hierzu wurde eine RViz-Konfiguration erstellt für die Navigation mit Nav2. In dieser Navigation-Szene wurde das Topic `/scan` visualisiert, um die LIDAR-Daten während der autonomen Fortbewegung des Roboters zu überprüfen. Zusätzlich konnte über das RViz-Tool die aktuelle Transformationsstruktur (`/tf`) in Echtzeit kontrolliert und mit der erwarteten TF-Hierarchie abgeglichen werden.

Im Rahmen dieser Validierung wurde erwartet, dass sich der Roboter in korrekter Orientierung entlang der Karte bewegt und dass die Positionsdaten mit den tatsächlichen Bewegungen übereinstimmen. Die über das Topic `/scan` übertragenen LIDAR-Punkte sollten dabei optisch exakt mit den simulierten Wänden und Hindernissen übereinstimmen. Die Odometrie über das Topic `/odom` musste sich synchron zur Bewegung aktualisieren und realistische Positionen und Geschwindigkeiten liefern.

Ein zentrales Element der Überprüfung war die Konsistenz der Transformationspfeile (`/tf`), insbesondere entlang der Kette `odom` → `base_footprint` → `base_link`. Nur bei korrekt aufgebauter und stabil übertragener TF-Struktur ist eine sinnvolle räumliche Verknüpfung von Sensordaten und Steuerkommandos möglich.

Diese umfassenden visuellen Prüfungen waren unerlässlich, um die technische Umsetzung in Isaac Sim mit den tatsächlichen semantischen Anforderungen und Erwartungen eines ROS2-Systems abzugleichen. Sie stellten sicher, dass Navigation, Sensorik und Darstellung korrekt ineinandergreifen können.

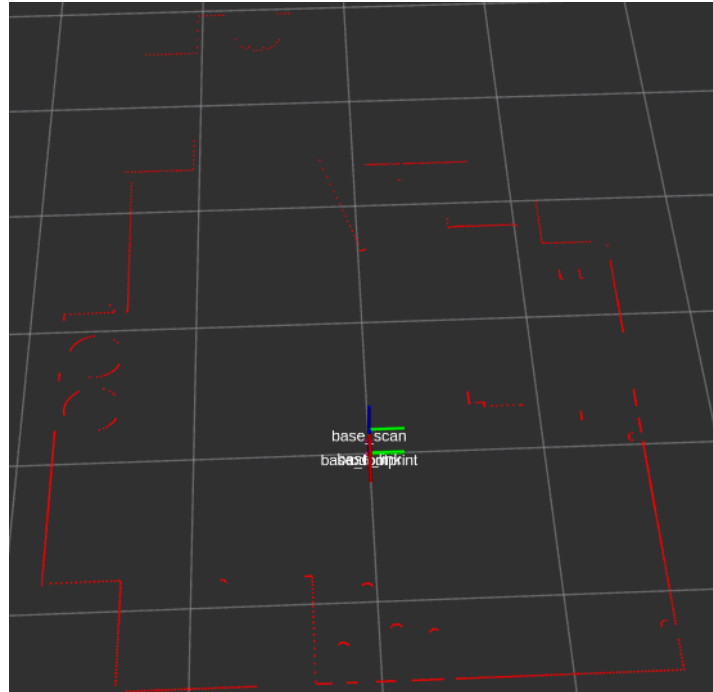


Abbildung 4: RVIZ

In Abbildung 4 sind die LIDAR-Daten des Roboters in RViz2 dargestellt (rote Punktwolke), die in Echtzeit über das Topic /scan empfangen werden. Die Punkte repräsentieren die von den LIDAR-Sensoren detektierten Entfernungen zu Wänden, Möbeln und anderen Objekten in der simulierten Wohnung. Im Zentrum ist der Roboter mit den beiden TF-Frames `base_footprint` und `base_scan` zu erkennen. Diese Koordinatensysteme definieren die physikalische Basis (Aufstandsfläche) des Roboters sowie die Position des LIDAR-Sensors relativ zur Basis. Die Transformationspfeile (blau, grün, rot für z/y/x) zeigen die jeweilige Orientierung dieser Frames im dreidimensionalen Raum.

Die Visualisierung zeigt, dass sowohl die Transformationen als auch die LIDAR-Daten korrekt miteinander verknüpft sind, ein wichtiger Validierungsschritt für SLAM und Navigation.

5. Navigation und Objekterkennung im Systemkontext

In diesem Kapitel werden die praktischen Resultate der entwickelten Komponenten des Robotersystems vorgestellt. Nach erfolgreicher Einrichtung der Hardware-in-the-Loop-Simulation mit Isaac Sim, der Integration der Sensorik sowie der Umsetzung zentraler ROS2-Module, konnten verschiedene Funktionalitäten systematisch getestet und validiert werden.

Zunächst wurde mithilfe von `slam_toolbox` eine Karte der simulierten Wohnumgebung erstellt, die als Grundlage für die spätere Navigation dient. Die resultierende Karte zeigt eine klare, strukturierte Darstellung der Umgebung mit erkannten Wänden, Hindernissen und freien Bereichen. Sie konnte erfolgreich gespeichert und in das Navigationssystem eingebunden werden.

Auf dieser Basis wurde das Nav2-Framework genutzt, um autonome Navigation zu realisieren. Die Navigation war in der Lage, zuverlässig Zielpunkte innerhalb der Umgebung anzufahren, Hindernisse zu umgehen und bei Bedarf Alternativrouten zu wählen.

Zusätzlich wurde ein YOLOv8-Modell zur Objekterkennung integriert. Durch den simulierten Kamerastream konnten Objekte wie Möbelstücke korrekt erkannt und klassifiziert werden. Die visuelle Ausgabe der erkannten Objekte wurde in Echtzeit angezeigt, wobei Bounding Boxes und Objektlabels über die Kamerabilder gelegt wurden.

5.1 Erstellung einer Umgebungskarte mit `slam_toolbox`

Um eine autonome Navigation zu ermöglichen, muss der Roboter zunächst eine Karte der Umgebung erstellen. Dies erfolgt über das Verfahren des Simultaneous Localization and Mapping (SLAM), bei dem die gleichzeitige Lokalisierung und Kartierung durch Sensordaten, insbesondere LIDAR, realisiert wird. In diesem Projekt wurde dafür die ROS2-kompatible `slam_toolbox` verwendet, die auf 2D-LIDAR-Daten basiert.

Für die Bewegung des Roboters während des SLAM-Prozesses kam bereits die Navigationsarchitektur Nav2 zum Einsatz. Wichtig ist dabei, dass Nav2 hier nicht zur Pfadplanung auf einer vorhandenen Karte, sondern lediglich zur gezielten Fortbewegung innerhalb der sich dynamisch entwickelnden SLAM-Karte genutzt wurde. Die Karte war also während der Bewegung im Aufbau, was einen realistischen Anwendungsfall simuliert, bei dem sich ein mobiler Roboter durch unbekannte Umgebungen bewegt.

Sowohl `slam_toolbox` als auch Nav2 sind modular aufgebaut und ermöglichen es, die zugrunde liegenden ROS2-Launch-Dateien und Parameterkonfigurationen an die eigene Simulationsumgebung anzupassen. Für dieses Projekt wurden eigene Launch-Skripte erstellt, in denen z. B. Topics wie `/scan`, die Frame-IDs (z. B. `base_link`, `map`) und die Simulationszeit berücksichtigt werden. Die Anpassungen wurden in separaten ROS2-Packages abgelegt, sodass die Komponenten unabhängig voneinander gestartet und flexibel kombiniert werden können.

Zur besseren Anpassung an die Projektstruktur wurden sowohl die Launch- als auch die Konfigurationsdateien für SLAM und Navigation dupliziert und in eigene ROS2-Packages überführt. In der Konfigurationsdatei der `slam_toolbox` lassen sich zahlreiche Einstellungen vornehmen, um das Verhalten der Kartenerstellung an die eigene Hardware und Umgebung anzupassen. Dazu gehören unter anderem die Definition der genutzten Frames und Topics, wie etwa `/scan`, `odom` und `base_footprint`, sowie die Angabe des Modus, ob eine neue Karte erstellt oder eine vorhandene zur Lokalisierung genutzt werden soll. Besonders wichtig ist auch die Anpassung der minimalen und maximalen LIDAR-Reichweite an die technischen Eigenschaften des verwendeten Sensors. Darüber hinaus können

Parameter zur Häufigkeit von Updates, zur Transformationsverarbeitung und zur Optimierung von Kartendaten durch Scan-Matching oder Loop-Closing verändert werden. Diese Optionen ermöglichen es, die SLAM-Performance präzise auf die Anforderungen des Roboters und der Umgebung abzustimmen.

In diesem Projekt wurde lediglich der Package-Name entsprechend geändert sowie eine einheitliche Ordnerstruktur mit einem config/-Verzeichnis statt params/ gewählt. In der SLAM-Konfigurationsdatei wurden zudem die LIDAR-Grenzwerte (`min_laser_range`, `max_laser_range`) an die verwendete Hardware angepasst (0.12 m bis 12.0 m).

Die übrigen Parameter konnten unverändert übernommen werden, da sie standardmäßig auf das Modell TurtleBot3 Burger von ROBOTIS abgestimmt sind, das in seiner Struktur unserem Roboter entspricht. Wichtig war lediglich, dass die verwendeten Topik Namen bzw. Frame-IDs wie `/scan`, `/odom` und `/base_footprint` in Isaac Sim korrekt gesetzt wurden, insbesondere bei der Erstellung der Action Graphs, um eine korrekte Kommunikation mit der SLAM- und Nav2-Architektur zu gewährleisten.

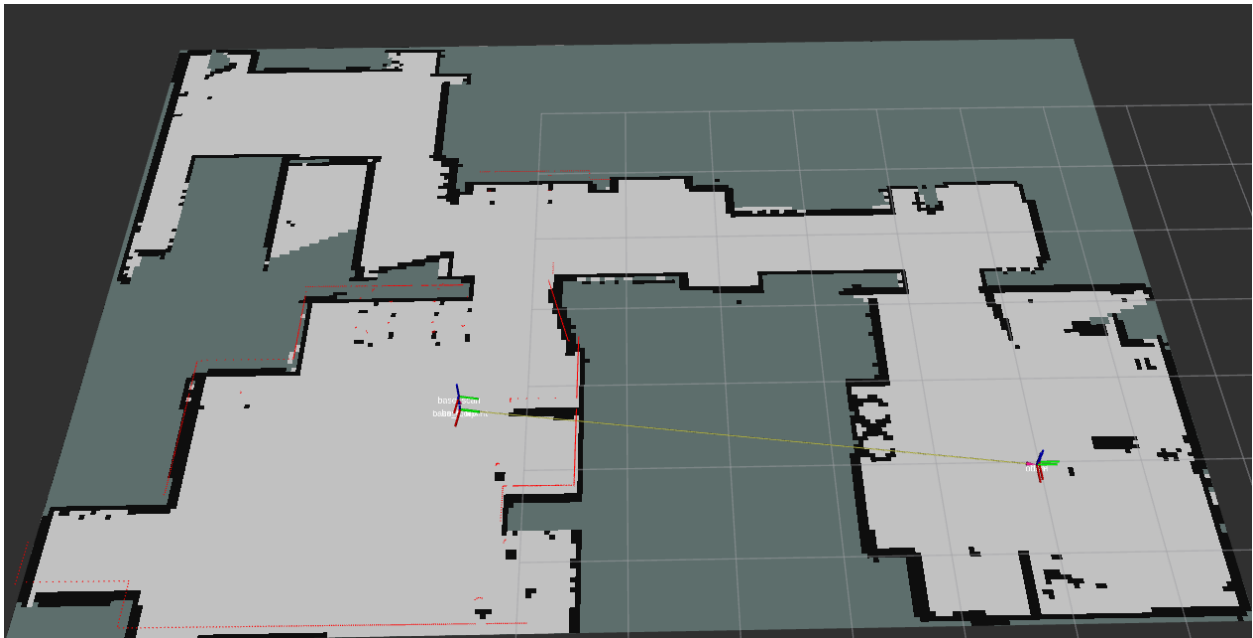


Abbildung 5: Karte durch SLAM in rviz2

Die Abbildung zeigt die fertig generierte 2D-Karte der simulierten Wohnumgebung in RViz2 nach Abschluss des SLAM-Prozesses. Die hellgrauen Flächen repräsentieren dabei die frei befahrbaren Bereiche, während die dunkelgrauen Zonen unbekanntes Terrain darstellen. Schwarze Pixel markieren erkannte Hindernisse oder Wände, die vom LIDAR-Sensor zuverlässig kartiert wurden.

Im Anschluss an die erfolgreiche Kartenerstellung wurde die resultierende Umgebungskarte mithilfe des ROS2-Kommandos `map_saver_cli` aus dem Paket `nav2_map_server` gespeichert. Dieser Befehl erzeugt zwei Dateien: eine Bilddatei im PGM-Format, welche die belegten, freien und unbekannten Bereiche der Umgebung kodiert, sowie eine YAML-Datei, die Metadaten wie die Auflösung, den Ursprung und das zugehörige Bild angibt. Beide Dateien zusammen bilden die Grundlage für die spätere Navigation mit dem Nav2-Stack.

5.2 Nutzung der Karte zur autonomen Navigation mit Nav2

Nach der erfolgreichen Erstellung und Speicherung der Umgebungskarte folgt im nächsten Schritt deren Integration in den Navigations-Stack von ROS2. Mithilfe des Nav2-Frameworks ist es möglich, auf Basis dieser Karte autonome Navigation zu realisieren, das heißt, dem Roboter werden Zielpositionen vorgegeben, die er unter Berücksichtigung von Hindernissen eigenständig ansteuert.

In der Nav2-Konfigurationsdatei lassen sich ähnlich wie bei SLAM eine Vielzahl von Parametern anpassen, um das Verhalten des Navigations-Stacks an die Eigenschaften der Umgebung und des Roboters anzupassen. Dazu gehören unter anderem Parameter für die globale und lokale Kostenkarte (Costmap), den Pfadplaner (Planner), den Controller sowie den Verhaltensteil (Behavior Tree). Beispielsweise können Auflösung, Sensorreichweiten, Inflationsradien und Algorithmen für die Pfadplanung konfiguriert werden.

Nach dem erfolgreichen Abspeichern der Karte kann nun ein neues RViz2-Fenster geöffnet werden, in dem die vorbereitete Umgebungskarte geladen wird. Dazu wird zunächst Nav2 aktiviert, beispielsweise über das eigene Launch-Package, welches die Navigation-Server startet. Anschließend lässt sich die gespeicherte Karte über das `map_server`-Node laden.

Sobald die Karte aktiv ist, kann die LIDAR-Datenanzeige in RViz2 deaktiviert werden. Die Lokalisierung und Navigation funktionieren dennoch weiterhin korrekt, da Nav2 mit Hilfe der zuvor erstellten statischen Karte arbeitet. Das System nutzt die eingehenden Odometrie- und ggf. transformierten Sensordaten, um die Position des Roboters auf der Karte kontinuierlich zu schätzen und geplante Zielpunkte anzusteuern.

Um eine autonome Navigation zu ermöglichen, muss der Roboter zunächst seine eigene Position innerhalb der Umgebung erkennen können. Dazu wird eine zuvor erstellte statische Karte in das System geladen und der Lokalisierungsprozess gestartet.

Im Rahmen dieses Projekts wird die Lokalisierung mit dem AMCL-Algorithmus (Adaptive Monte Carlo Localization) realisiert, der probabilistisch arbeitet und eine Partikelfilter-Methode nutzt. Die dazugehörige Launch-Datei startet sowohl den Kartendienst (`map_server`) als auch den Lokalisierungsknoten (`amcl`). Eine zuvor erstellte YAML-Datei enthält alle relevanten Metadaten zur Karte, wie Auflösung, Ursprung und die zu ladende Bilddatei.

Nach dem Start des Lokalisierungssystems ist es zwingend notwendig, die initiale Position des Roboters manuell festzulegen. Dies erfolgt über das Tool *2D Pose Estimate* in RViz2. Erst danach ist eine sinnvolle Positionsschätzung durch AMCL möglich, da der Roboter sonst keine Kenntnis über seine Lage innerhalb der Karte hat. Sobald die initiale Pose gesetzt wurde, erscheint die Karte in RViz2, und der Roboter ist korrekt auf dieser visualisiert. Bewegungen des Roboters innerhalb der Simulation werden dann konsistent auf der Karte dargestellt. Dadurch ist der Übergang zur autonomen Navigation vorbereitet.

Nach erfolgreicher Lokalisierung ist der Roboter in der Lage, sich autonom in der bekannten Umgebung zu bewegen. Das Navigationssystem wird in einem separaten Prozess gestartet. Es nutzt die bestehende Karte, die bereits im Lokalisierungsschritt geladen wurde. Neben dem globalen und lokalen Planer werden hier auch der Controller-Server und weitere Lifecycle-Knoten aktiviert. Voraussetzung für den Start ist, dass die Karte bereits geladen und die Position des Roboters durch AMCL korrekt geschätzt wurde.

In RViz2 kann nun über das Tool *2D Nav Goal* ein Zielpunkt gesetzt werden. Dieser Punkt definiert die gewünschte Zielposition sowie die Ausrichtung des Roboters. Nach dem Setzen beginnt das Nav2-

System automatisch mit der Pfadplanung unter Berücksichtigung der statischen Karte und der Echtzeit-Sensordaten.

In der Simulation zeigt der Roboter ein stabiles Navigationsverhalten. Er berechnet auf Basis der statischen Karte und dynamischer Sensordaten eine sichere Route zum Ziel und passt seine Bewegung kontinuierlich an auftretende Hindernisse an. Die Steuerung erfolgt dabei über einen Local Planner, der Echtzeitentscheidungen trifft.

5.3 Integration der YOLO-Objekterkennung als Proof of Concept

Für die visuelle Objekterkennung in unserer simulierten Umgebung implementierten wir ein eigenes ROS2-Node-Skript, das mithilfe der Bibliothek ultralytics das Modell YOLOv8m verwendet. Dieses Skript abonniert das Kamerabild direkt aus Isaac Sim über das Topic `/rgb`, konvertiert es mit `cv_bridge` und führt anschließend die Objekterkennung durch. Die erkannten Objekte werden inklusive ihrer Labels und Konfidenzwerte, die über 60% liegen, in einem Live-Fenster über OpenCV angezeigt.

Die Verarbeitung erfolgt auf der CPU, was zwar eine höhere Latenz bedeutet, aber für Testzwecke und bei moderater Szenenkomplexität im Simulationskontext ausreichend ist.



Abbildung 6: Bilderkennung

Die Abbildung zeigt die erfolgreiche Objekterkennung innerhalb der Simulationsumgebung durch das YOLOv8m-Modell. Zu sehen ist ein Stuhl, der mit einer Konfidenz von 0,86 erkannt und durch einen grünen Rahmen visuell hervorgehoben wurde. Die Beschriftung „chair 0.86“ erscheint am oberen Bildrand und verdeutlicht, dass das Modell nicht nur die Objektklasse korrekt identifiziert, sondern auch eine hohe Sicherheit in der Vorhersage besitzt.

6. Ergebnisse und Diskussion

6.1 Funktion und Qualität der Karte und Navigation

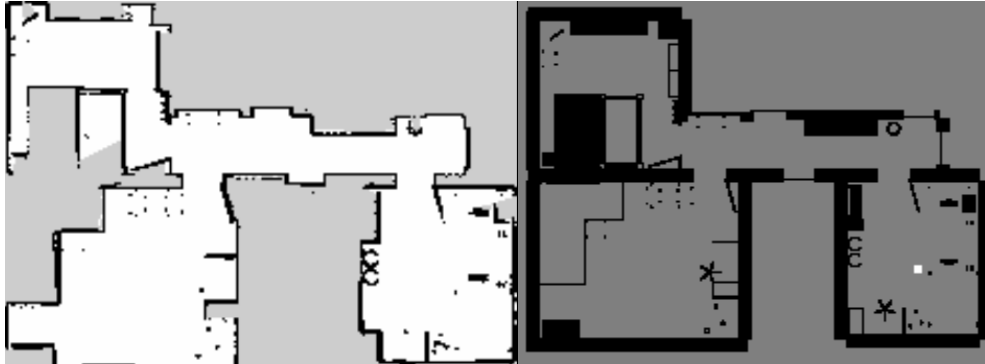


Abbildung 7: Vergleich Erstellte Karte mit SLAM vs Original

Zur Bewertung der Kartengenauigkeit wurde die durch SLAM erzeugte Karte (links) der ideal generierten Occupancy Map aus Isaac Sim (rechts) gegenübergestellt.

Die SLAM-Karte bildet die Umgebung anhand der LIDAR-Daten, die während der Bewegung des Roboters gesammelt wurden. Insgesamt ist die Struktur der Wohnung gut erkennbar. Die Hauptwände, Türen und Verbindungsflure wurden korrekt erfasst. In einigen Bereichen zeigen sich jedoch kleinere Ungenauigkeiten und Auslassungen, beispielsweise an Ecken, schmalen Durchgängen oder in Bereichen mit eingeschränkter Sensorabdeckung. Solche Unschärfen sind typische Effekte von SLAM-Systemen und hängen stark davon ab, wie vollständig und gleichmäßig die Umgebung abgefahren wurde. Auch einige vereinzelte Pixelartefakte lassen sich beobachten etwa in Form von kleinen, nicht vorhandenen Objekten oder Lücken.

Die Occupancy Map aus Isaac Sim zeigt demgegenüber ein vollständig symmetrisches und rauschfreies Abbild der Wohnung. Alle geometrischen Strukturen wurden exakt übernommen. Dies liegt daran, dass diese Karte direkt aus dem Simulationsmodell abgeleitet wurde und somit keine Sensorunsicherheiten oder Bewegungsfehler enthält.

Im direkten Vergleich zeigt sich, dass die mit SLAM erzeugte Karte eine qualitativ ausreichende Grundlage für die Navigation bietet. Die relevanten Raumstrukturen wurden korrekt erkannt und interpretiert. Trotz kleinerer Ungenauigkeiten ist die Karte konsistent genug, um robuste Pfadplanung und Zielnavigation zu ermöglichen. Die ideale Isaac-Sim-Karte dient hier als Referenz und macht sichtbar, wo SLAM-bedingte Abweichungen auftreten können. Für den Einsatz in realitätsnahen Tests ist die SLAM-Karte jedoch ein realistisches und funktionales Ergebnis.

6.2 Funktion und Qualität der Navigation

Die autonome Navigation auf Grundlage der mittels SLAM generierten Karte erwies sich im praktischen Test als zuverlässig und konsistent. Zielkoordinaten konnten innerhalb der RViz2-Oberfläche definiert werden, woraufhin das System erfolgreich eine Route berechnete und diese eigenständig abfuhr. Die Ausführung der Bewegungen einschließlich translatorischer Fahrmanöver sowie rotatorischer Ausrichtungen erfolgte dabei flüssig, stabil und ohne erkennbare Oszillationen oder Pfadabweichungen.

Ein beobachtbarer Aspekt war, dass der Roboter das Ziel in einzelnen Fällen geringfügig vor dem exakten Zielpunkt stoppte. Dies lässt sich durch die definierte Zieltoleranz (xy- und yaw-Toleranz) im goal_checker-Modul der Navigation erklären, welche bewusst eine gewisse Flexibilität hinsichtlich der exakten Endposition einräumt. Das Verhalten ist systemseitig korrekt und entspricht den gewählten Parametereinstellungen.

Besonders positiv fiel auf, dass bei nicht erreichbaren Zielkoordinaten etwa solchen, die sich innerhalb vollständig blockierter Bereiche befanden nach einem kurzen Evaluationszeitraum ein automatischer Abbruch des Navigationsversuchs (Status: aborted) erfolgte. Diese Fähigkeit zur Erkennung nicht lösbarer Planungsprobleme ist ein entscheidender Aspekt robuster autonomer Systeme.

Die Navigation zeigt sich auf Grundlage der SLAM-Karte in Kombination mit der eingesetzten Pfadplanung und Kontrolle als eine verlässliche autonome Bewegung des Roboters innerhalb der erfassten Umgebung. Die Reaktionsfähigkeit auf Zielveränderungen sowie auf planerische Sackgassen bestätigt die funktionale Integration der beteiligten Module und die geeignete Parametrisierung des Gesamtsystems.

6.3 Funktion und Qualität der Bilderkennung

Die Implementierung der Objekterkennung mittels YOLOv8 gestaltete sich aus technischer Sicht als unkompliziert. Das Python-Skript konnte mithilfe der Ultralytics-Bibliothek mit minimalem Aufwand in den ROS2-Stack integriert werden, wobei insbesondere der Zugriff auf das Kamerabild aus Isaac Sim via ROS-Topic /rgb reibungslos funktionierte.

Während die grundsätzliche Einbindung erfolgreich verlief, traten in der Praxis zwei zentrale Herausforderungen auf. Erstens zeigte sich, dass die Übertragungsfrequenz der Bilddaten aus Isaac Sim nicht konstant stabil war. Dies führte dazu, dass einzelne Bilder verzögert oder unregelmäßig verarbeitet wurden, was sich auf die Echtzeitfähigkeit der Objekterkennung negativ auswirkte. Um diesem Problem entgegenzuwirken, wurde die Kameraauflösung gezielt auf 200×200 Pixel reduziert. Diese Maßnahme half zwar, die Verarbeitungsfrequenz auf etwa 2–5 Hz zu steigern, ging jedoch zulasten der Bildqualität und erschwerte dadurch insbesondere die zuverlässige Erkennung kleiner oder detailarmer Objekte.

Zweitens kam es insbesondere bei niedrigem Konfidenzniveau zu falschen Positiverkennungen. Um diesem Umstand zu begegnen, wurde die Inferenzschwelle (conf) manuell auf 0.6 gesetzt, sodass nur Objekte mit ausreichend hoher Modellvertrauenswahrscheinlichkeit visualisiert und berücksichtigt wurden. Diese Maßnahme reduzierte zwar die Erkennungsrate für schwach sichtbare Objekte, erhöhte jedoch signifikant die Zuverlässigkeit der dargestellten Ergebnisse.

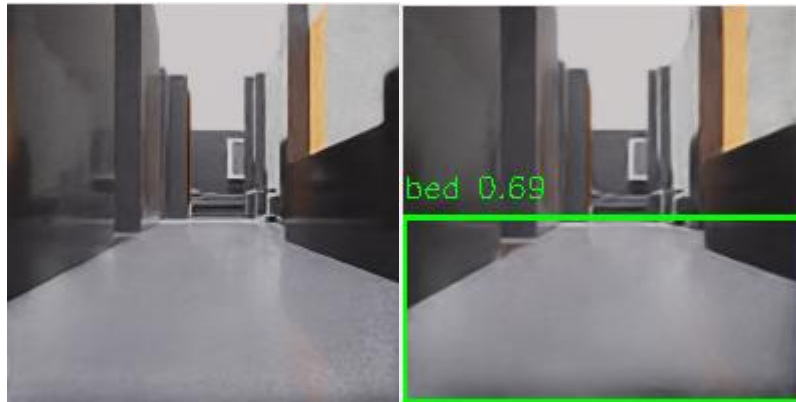


Abbildung 8: Vergleich YOLO Inferenzschwelle 0.6 vs 0.3

In der Abbildung 8 ist deutlich zu erkennen, wie sich die Wahl der Inferenzschwelle auf die Erkennungsergebnisse von YOLOv8 auswirkt. Bei einer niedrigeren Schwelle von 0,3 (rechts) wurde fälschlicherweise der Boden als Bett („bed“) identifiziert, was bei einer konservativeren Schwelle von 0,6 (links) nicht der Fall war. Diese fehlerhafte Klassifizierung trat insbesondere während der Fahrt des Roboters auf, wenn flache Strukturen wie glatte Böden im Sichtfeld lagen.

Darüber hinaus zeigte sich im Praxistest, dass die Erkennungsleistung stark vom Objektbestand des zugrunde liegenden COCO-Datensatzes abhängig war. Möbelstücke wie Stühle, Sofas, Betten oder Esstische, die explizit in den Trainingsdaten von YOLOv8m enthalten sind, wurden zuverlässig erkannt, selbst unter vereinfachten Simulationsbedingungen. Andere für Wohnräume typische Objekte wie Schränke, Kratzbäume, Hanteln oder Futternäpfe hingegen blieben systematisch unentdeckt, da sie nicht zu den standardmäßig trainierten Klassen gehören. Die Erkennung beschränkte sich somit effektiv auf einen Teilbereich der tatsächlichen Umgebung.

Zum anderen ist die visuelle Darstellung in Isaac Sim hinsichtlich Texturtreue, Farbgebung und Schattenwurf stark vereinfacht. Diese Reduktion der Bildtiefe kann dazu führen, dass selbst potenziell klassifizierbare Objekte für das Modell nicht eindeutig identifizierbar sind, insbesondere wenn charakteristische visuelle Merkmale wie Farbe oder Oberflächenstruktur fehlen. In Kombination mit der reduzierten Bildauflösung (200×200 Pixel zur Frequenzsteigerung) erschwert dies die semantische Interpretation zusätzlich. Die Ergebnisse zeigen damit exemplarisch die Grenzen vortrainierter Erkennungsmodelle in simulationsbasierten Kontexten auf, insbesondere dann, wenn keine weiteren Trainingsmaßnahmen zur Domänenanpassung erfolgen.

6.4 Probleme und technische Einschränkungen

Im Verlauf der Umsetzung zeigten sich diverse technische Einschränkungen und Stabilitätsprobleme, die die Entwicklungsarbeit wiederholt erschwerten. Ein zentrales Problem bestand in der gleichzeitigen Verarbeitung der LIDAR- und Bilddatenströme. Zum Beispiel bei der Darstellung der Lidar und Kamera Daten in RVIZ2. Beide Datenquellen verursachten eine erhebliche Auslastung des Systems, was nicht nur zu verringerter Performance, sondern auch zu Instabilität beim Datenempfang führte. Als Konsequenz musste die Verarbeitung der Bilddaten in ein separates Node ausgelagert werden.

Ein weiteres Hindernis bestand in der verzögerten oder inkonsistenten Aktualisierung von Parametern innerhalb der Action Graphs. Änderungen an Parametern etwa der Kameraposition oder der Bildgröße wurden häufig erst nach einem kompletten Neustart der Simulationsumgebung wirksam. Dies erschwerte insbesondere das iterative Testen und führte zu hohem Zeitaufwand bei Konfigurationsänderungen.

Zusätzlich kam es zu regelmäßigen Abstürzen von Isaac Sim, sowohl beim Start der Anwendung als auch während des laufenden Betriebs. Die Ursachen dieser Instabilitäten waren häufig nicht eindeutig erkennbar, da Fehlermeldungen unpräzise oder nicht vorhanden waren. Auch die Aktualisierung von Parametern im Action Graph erfolgte in vielen Fällen nicht in Echtzeit, sodass ein kompletter Neustart der Simulation notwendig war, um Änderungen wirksam zu machen. Besonders kritisch erwies sich dies beim Testen und Optimieren von Sensorkonfigurationen.

Darüber hinaus konnten bestimmte Features wie die "omni.anim.navigation.core" = { version = "106.4.0" } nach Updates die gesamte Simulationsumgebung zum Absturz bringen. Diese Fehler zeigten, dass Isaac Sim trotz seiner Möglichkeiten als Entwicklungsumgebung nicht durchgängig stabil oder robust in der Handhabung ist. Auch in Bezug auf Speicherverbrauch und GPU-Last offenbarte sich eine hohe Abhängigkeit von sehr leistungsfähiger Hardware.

Diese technischen Einschränkungen sollten bei zukünftigen Projekten frühzeitig berücksichtigt werden. Alternativ kann es sinnvoll sein, auf ressourcenschonendere Simulationsumgebungen wie Gazebo auszuweichen, sofern keine komplexen physikalischen Interaktionen erforderlich sind.

7. Fazit und Ausblick

Im Verlauf dieses Projekts wurde ein vollständiges System zur autonomen Navigation und Objekterkennung in einer häuslichen Umgebung erfolgreich aufgebaut und in zwei getrennten Anwendungsszenarien erprobt, jeweils mit Fokus auf eine zentrale Fähigkeit: Navigation bzw. visuelle Erkennung. Dabei wurden moderne Simulationswerkzeuge, SLAM, Navigationsalgorithmen bzw. KI-Bilderkennung und ROS2 zielgerichtet miteinander kombiniert, um eine belastbare, erweiterbare Systemarchitektur zu entwickeln.

Zunächst wurden im Rahmen einer fundierten theoretischen Einführung die wesentlichen Technologien wie Isaac Sim für die physikbasierte Simulation, ROS2 als Middleware sowie slam_toolbox und Nav2 für die Navigation eingeführt. Ergänzend kam das Objekterkennungsmodell YOLOv8m zum Einsatz, das im späteren Verlauf separat zur visuellen Kontextanalyse genutzt wurde.

Im praktischen Teil des Projekts erfolgte die detailgetreue Nachbildung einer realen Wohnumgebung in Isaac Sim, die als Testumgebung für den virtuellen Roboter diente. Der eingesetzte Roboter besaß zwei aktiv steuerbare Räder sowie ein passives Stützrad zur Stabilisierung. Die Sensorik bestand aus einem 2D-LIDAR und einer RGB-Kamera, die in Isaac Sim eingebunden und über Action Graphs mit ROS2 synchronisiert wurde. Die notwendigen Transformationsdaten, Odometrieinformationen sowie Sensorausgaben wurden erfolgreich in RViz2 visualisiert und zur Steuerung genutzt.

Ein erstes Anwendungsszenario konzentrierte sich auf die autonome Navigation: Mit Hilfe der slam_toolbox wurde eine Karte der Umgebung erstellt, die anschließend durch Nav2 zur Zielpunktnavigation genutzt wurde. Der Roboter konnte dabei präzise durch die simulierte Wohnung manövrieren, auf Zieländerungen reagieren und nicht erreichbare Ziele nach kurzer Zeit korrekt verwerfen. Die Navigation verlief dabei flüssig, mit realistischen Drehungen und angemessener Reaktionszeit, auch wenn der Zielpunkt gelegentlich minimal vorzeitig als erreicht gewertet wurde.

In einem separaten zweiten Anwendungsszenario wurde die Kameraanbindung zur Bildverarbeitung genutzt. Die Integration von YOLOv8m ermöglichte eine visuelle Objekterkennung anhand der RGB-Bilddaten. Um die Rechenlast zu reduzieren und die Verarbeitungsfrequenz zu erhöhen, wurde die Bildauflösung auf 200×200 Pixel reduziert. Die Objekterkennung wurde mithilfe eines Confidence Thresholds von 0.6 stabilisiert, wodurch Fehlklassifikationen, etwa das Erkennen des Bodens als Bett, minimiert werden konnten. Die Erkennungsfrequenz lag bei ca. 2–5 Hz. Es zeigte sich jedoch, dass nicht viele Haushaltsobjekte zuverlässig erkannt wurden, was sowohl auf das Fehlen spezifischer Objektklassen im Trainingsdatensatz (COCO) als auch auf die fehlende Farbinformation in Isaac Sim zurückgeführt werden konnte.

Die Umsetzung des Projekts ermöglichte eine tiefgreifende Auseinandersetzung mit den praktischen und konzeptionellen Herausforderungen bei der Entwicklung eines KI-gestützten, autonomen Robotersystems in einer simulierten häuslichen Umgebung. Obwohl die gesetzten Teilziele erreicht wurden, offenbarte der Projektverlauf auch wesentliche technische Hürden und strukturelle Grenzen der eingesetzten Tools, die in einer kritischen Gesamtschau betrachtet werden müssen.

Ein zentrales Problem bestand in der begrenzten Stabilität und Skalierbarkeit von Isaac Sim. Trotz seiner physikalisch realistischen Simulationen erwies sich das System als absturzanfällig, insbesondere bei komplexeren Konfigurationen mit mehreren Sensoren oder beim Einsatz von ROS2-Extensions wie der Navigations-Extension. Darüber hinaus zeigte sich, dass Parameteränderungen im Action Graph nicht

konsistent oder verzögert übernommen wurden, was häufige Neustarts zur Folge hatte und den Entwicklungsprozess erheblich verlangsamte.

Auch die Trennung der Anwendungsfälle, Navigation und Objekterkennung, war nicht primär konzeptionell motiviert, sondern wurde notwendig, um die Systemlast in Grenzen zu halten. Die parallele Verarbeitung von LIDAR- und Bilddaten führte zu Performance-Einbrüchen, wodurch ein simultaner Betrieb beider Funktionen in Echtzeit nicht mehr praktikabel war.

Im Bereich der visuellen Objekterkennung zeigte sich, dass trotz der einfachen Integration von YOLOv8m die Modellarchitektur und das zugrundeliegende COCO-Dataset für den häuslichen Kontext nur bedingt geeignet waren. Viele relevante Objekte wie Möbel oder spezifische Haushaltsgegenstände wurden nicht erkannt, was auf fehlende Trainingsdaten sowie auf die geringe Bildqualität und Farblosigkeit der Simulationsbilder zurückzuführen ist. Zwar konnte durch die Reduktion der Bildgröße ein Kompromiss zwischen Frequenz und Genauigkeit erreicht werden, doch war die Erkennungsleistung stark schwankend und nicht immer zuverlässig.

Positiv hervorzuheben ist jedoch die klare Modularisierung des Systems. Die Trennung der Navigation und Bildverarbeitung ermöglichte es, beide Komponenten unabhängig zu testen, zu bewerten und potenziell später zusammenzuführen. ROS2 stellte sich dabei als äußerst flexibles Framework heraus, das sowohl einfache als auch komplexe Datenflüsse über Nodes und Topics effizient abbildet, vorausgesetzt, die zugrundeliegende Simulationsplattform unterstützt die Synchronisierung konsistent.

Das vorliegende Projekt legte eine solide technische Grundlage für die Entwicklung KI-gesteuerter, autonom navigierender Robotersysteme in Innenräumen. Aufbauend auf dieser Basis eröffnen sich vielfältige Perspektiven für zukünftige Erweiterungen, sowohl hinsichtlich der Systemfunktionalität als auch der methodischen Tiefe.

Ein erster zentraler Ansatzpunkt besteht in der Integration eines Behavior-Tree-Systems, das dem Roboter ermöglicht, sein Verhalten kontextsensitiv zu steuern. Durch die Definition regelbasierter Entscheidungsbäume ließe sich beispielsweise modellieren, wie sich der Roboter in verschiedenen Raumzonen (z. B. Wohnzimmer, Schlafzimmer) oder bei Begegnung mit bestimmten Objekten (z. B. Katze, Mensch) verhalten soll. Dies wäre ein bedeutender Schritt in Richtung semantisch informierter Navigation.

Des Weiteren stellt die bislang getrennte Verarbeitung von LIDAR- und Kameradaten eine Einschränkung dar. Eine der nächsten Herausforderungen besteht daher in der Entwicklung eines fusionierten Sensorikmodells, das beide Informationsquellen gleichzeitig auswertet. Damit ließe sich die Umgebung deutlich robuster und differenzierter wahrnehmen.

Auch im Bereich der Navigation bieten sich Erweiterungen an. Während bislang auf klassische SLAM- und Pfadplanungsverfahren zurückgegriffen wurde, könnte in einem nächsten Schritt ein Reinforcement-Learning-Ansatz erprobt werden, der es dem Roboter erlaubt, Navigationsstrategien durch Erfahrung zu erlernen und sich an sich verändernde Umgebungen anzupassen.

Im Bereich der Objekterkennung wäre es zudem möglich, das YOLO-Modell durch Feintraining auf projektspezifische Objekte (z. B. Futternäpfe, Kratzbäume, Bücherregale) zu erweitern. Durch Transfer Learning oder das Training auf synthetisch erzeugten Szenen könnten bislang nicht erkannte Objekte zuverlässig klassifiziert werden.

Nicht zuletzt könnte die Simulationsumgebung selbst weiter ausgebaut werden: dynamische Objekte wie sich bewegende Tiere oder Personen sowie komplexere Lichtverhältnisse oder Tageszeiten würden die Realitätstreue der Tests erhöhen und das System besser auf reale Herausforderungen vorbereiten.

Insgesamt zeigen diese Perspektiven, dass das Projekt nicht als abgeschlossenes System, sondern als modular erweiterbare Plattform verstanden werden kann, deren Weiterentwicklung sowohl methodisch als auch praktisch erhebliches Potenzial birgt

Literaturverzeichnis

BMW Press. (2023, März 1). *Die BMW Group nutzt für ihre Produktionsplanung NVIDIA Omniverse*

Enterprise. BMW GROUP.

<https://www.press.bmwgroup.com/deutschland/photo/detail/P90498775/die-bmw-group-nutzt-fuer-ihre-produktionsplanung-nvidia-omniverse-enterprise-eine-plattform-fuer-den-aufbau-und-betrieb-industrieller-3d-metaverse-anwendungen-um-simulationen-mit-digitalen-zwillingen-in-echtzeit-durchzufuehren-und-damit-layouts-robotik-und-logistiksysteme-virtuell-zu-optimieren-maerz-2023?>

Duarte, C., Luo, S., Ojalvo, J., Pasternak, K., & Visser, U. (2025). A Quantitative Comparison of Vision Performance for the HSR: Gazebo vs. Isaac Simulator. In E. Barros, J. P. Hanna, H. Okada, & E. Torta (Hrsg.), *RoboCup 2024: Robot World Cup XXVII* (S. 339–350). Springer Nature Switzerland. https://doi.org/10.1007/978-3-031-85859-8_29

Isaac Sim. (2025, Mai 26). NVIDIA Developer. <https://developer.nvidia.com/isaac/sim>

Liao, Y. (2020, Juli 30). *Autonome Systeme – Roboter-Betriebssysteme: ROS2 bügelt Schwächen aus – Magazin des Fraunhofer-Instituts für Kognitive Systeme IKS*. DE / Safe Intelligence. <https://safe-intelligence.fraunhofer.de/artikel/autonome-systeme-ros2>

Lin, T.-Y., Maire, M., Belongie, S., Bourdev, L., Girshick, R., Hays, J., Perona, P., Ramanan, D., Zitnick, C. L., & Dollár, P. (2014, Mai 1). *Microsoft COCO: Common Objects in Context*. arXiv.Org. <https://arxiv.org/abs/1405.0312v3>

Macenski, S., & Jambrecic, I. (2021). SLAM Toolbox: SLAM for the dynamic world. *Journal of Open Source Software*, 6(61), 2783. <https://doi.org/10.21105/joss.02783>

Macenski, S., Martín, F., White, R., & Clavero, J. G. (2020). The Marathon 2: A Navigation System. 2020 *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2718–2725. <https://doi.org/10.1109/IROS45743.2020.9341207>

NVIDIA (Regisseur). (2022, März 23). *Amazon Robotics Builds Digital Twins of Warehouses with NVIDIA Omniverse and Isaac Sim* [Video recording]. <https://www.youtube.com/watch?v=-VQLqs6s9y0>

NVIDIA. (2025a, Mai 26). *NVIDIA Omniverse*. NVIDIA. <https://www.nvidia.com/de-de/omniverse/>

NVIDIA. (2025b, Juni 10). *Installation—Isaac Sim Documentation*.
<https://docs.isaacsim.omniverse.nvidia.com/4.5.0/installation/index.html>

NVIDIA. (2025c, Juni 11). *OmniGraph—Isaac Sim 4.2.0 (OLD)*.
https://docs.omniverse.nvidia.com/isaacsim/latest/gui_tutorials/tutorial_gui_omnigraph.html?

NVIDIA. (2025d, Juni 16). *ROS2 Clock—Isaac Sim Documentation*. ROS2 Clock.
https://docs.isaacsim.omniverse.nvidia.com/4.5.0/ros2_tutorials/tutorial_ros2_clock.html

NVIDIA. (2025e, Juni 16). *ROS2 Transform Trees and Odometry—Isaac Sim Documentation*. ROS2 Transform Trees and Odometry.
https://docs.isaacsim.omniverse.nvidia.com/4.5.0/ros2_tutorials/tutorial_ros2_tf.html

NVIDIA. (2025f, Juni 16). *ROS2 Cameras—Isaac Sim Documentation*.
https://docs.isaacsim.omniverse.nvidia.com/4.5.0/ros2_tutorials/tutorial_ros2_camera.htm

NVIDIA. (2025g, Juni 19). *RTX Lidar Sensors—Isaac Sim Documentation*. RTX Lidar Sensors.
https://docs.isaacsim.omniverse.nvidia.com/4.5.0/ros2_tutorials/tutorial_ros2_rtx_lidar.html

NVIDIA. (2025h, Juni 16). *Driving TurtleBot via ROS2 messages—Isaac Sim Documentation*. Driving TurtleBot via ROS2 messages.
https://docs.isaacsim.omniverse.nvidia.com/4.5.0/ros2_tutorials/tutorial_ros2_drive_turtlebot.html

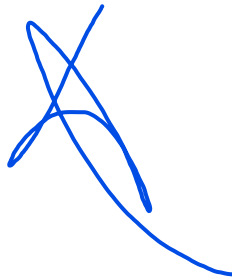
Terven, J., & Cordova-Esparza, D. (2024, Januar 7). *A Comprehensive Review of YOLO Architectures in Computer Vision: From YOLOv1 to YOLOv8 and YOLO-NAS*. <https://arxiv.org/html/2304.00501v6>

Ultralytics. (2025, April 1). *YOLOv8*. <https://docs.ultralytics.com/de/models/yolov8>

Eidesstattliche Versicherung

Ich versichere, dass ich das beiliegende Projektbericht selbstständig verfasst, keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie alle wörtlich oder sinngemäß übernommenen Stellen in der Arbeit gekennzeichnet habe.

Osterdde am Harz, 30.06.2025

A handwritten signature in blue ink, consisting of a large, stylized 'X' shape with a long, sweeping tail that curves downwards and to the right.