DEVELOPMENT OF OBSTABLE AVOIDANCE SYSTEM FOR 3D ROBOT NAVIGATION

ER KAI SHENG

UNIVERSITI TUNKU ABDUL RAHMAN

DEVELOPMENT OF OBSTABLE AVOIDANCE SYSTEM FOR 3D ROBOT NAVIGATION

ER KAI SHENG

A project report submitted in partial fulfilment of the requirements for the award of Bachelor of Mechatronics Engineering with Honours

Lee Kong Chian Faculty of Engineering and Science
Universiti Tunku Abdul Rahman

May 2024

DECLARATION

I hereby declare that this project report is based on my original work except for citations and quotations which have been duly acknowledged. I also declare that it has not been previously and concurrently submitted for any other degree or award at UTAR or other institutions.

Signature	:	Sheng
Name	:	ER KAI SHENG
ID No.	:	20UEB00934
Date	:	26 APRIL 2024

APPROVAL FOR SUBMISSION

I certify that this project report entitled **“DEVELOPMENT OF OBSTACLE AVOIDANCE SYSTEM FOR 3D ROBOT NAVIGATION”** was prepared by **ER KAI SHENG** has met the required standard for submission in partial fulfilment of the requirements for the award of Bachelor of Mechatronics Engineering with Honours at Universiti Tunku Abdul Rahman.

Approved by,

Signature	:	[illegible]
Supervisor	:	IR DR DANNY NG WEE KIAT
Date	:	16 MAY 2024

Signature	:	[illegible]
Co-Supervisor	:	DR KWAN BAN HOE
Date	:	16 MAY 2024

ACKNOWLEDGEMENTS

I would like to thank everyone who had contributed to the successful completion of this project. I would like to express my gratitude to my research supervisor, Ir. Dr. Danny Ng Wee Kiat and Dr. Kwan Ban Hoe for their invaluable advice, guidance and their enormous patience throughout the development of the research.

In addition, I would also like to express my gratitude to my loving parents and friends who had helped and given me encouragement to complete this project.

ABSTRACT

In this project, an algorithm was developed to implement an obstacle avoidance system in 3D robot navigation. Before project implementation, extensive research was conducted to explore the current state of obstacle avoidance systems for 3D robot navigation. Since 3D robot navigation systems require 3D environmental data, the study focused on 3D Simultaneous Localization and Mapping (SLAM) to obtain environmental information and generate a suitable 3D map for navigation. Two popular 3D SLAM methods, OctoMap and RTAB-Map, were studied, and RTAB-Map was chosen for its ability to directly create 3D maps from depth camera data and its incorporation of odometry error correction, potentially leading to more accurate 3D maps and occupancy grids. To prepare for obstacle avoidance algorithm development, a differential drive robot was constructed, and a URDF description was prepared to ensure correct odometry data conversion from sensor coordinate frames to the robot coordinate frame. Rviz2 was utilized for visualizing coordinate frames. The algorithm was tested in both simulation and on the physical robot. Gazebo simulation software was used to build a virtual world for testing the obstacle avoidance system. RTAB-Map was employed to construct the essential 3D map for navigation. To obtain a good 3D map in RTAB-Map SLAM, it is important to use a LiDAR to refine the odometry of the robot and improve map quality. In this project, robot navigation was implemented using packages provided by Nav2. The voxel layer in the layered cost map provided in Nav2 was utilized to detect the 3D obstacles that cannot be detected by 2D LiDAR. Additionally, the planner and controller modules from Nav2 were employed for path planning and obstacle avoidance. Once the algorithm was fully tested and proven functional in simulation, it would be implemented on the physical robot. The performance of the algorithm is discussed in the results and discussion section. In conclusion, the developed algorithm enables the robot to detect obstacles that are not on the same plane as the 2D LiDAR and cannot be detected using depth camera data.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS / ABBREVIATIONS

API	Application Programming Interface
CLI	Command Line Interface
AGV	Automated Guided Vehicle
ROS2	Robot Operating System 2
tf	Transform Functions (Library in ROS)
Nav2	ROS2 navigation stack
URDF	Unified Robot Description File
YAML	Yet Another Markup Language (Programming language)
XML	Extensible Markup Language (Programming language)
SLAM	Simultaneous Localization and Mapping
3D	Three-dimensional
2D	Two-dimensional

LIST OF APPENDICES

CHAPTER 1

INTRODUCTION

1.1 General Introduction

Industry 4.0 represents the fourth industrial revolution, characterized by the transformation of manufacturing methods from manual labor to machine automation. This transformation significantly enhances productivity, quality, and the reproducibility of goods through sophisticated digital control systems. One of the most vital applications in the industry is the Automated Guided Vehicle (AGV). An AGV is a vehicle capable of transporting goods from one designated position to another. Nowadays, AGVs are equipped with various end effectors, such as robot arms and forklifts, enabling them to perform more complex tasks beyond simple goods transfer.

Today, trackless AGVs are preferred over tracked AGVs in the industry. Trackless AGVs do not require pre-installed tracks, unlike their tracked counterparts. Moreover, trackless AGVs offer greater flexibility, as tracked AGVs can only follow and travel on pre-installed tracks. However, trackless AGVs necessitate more sophisticated detection sensors, such as Light Detection and Ranging devices (LiDAR), Inertial Measurement Units (IMUs), and depth cameras, to determine their position in an environment. The integration of these devices is crucial to ensure accurate AGV navigation.

One of the most popular methods for AGVs to detect their environment and create maps is by using two-dimensional (2D) LiDAR. A 2D LiDAR is a sensor that employs a rotating pulsed laser beam to measure distances between the device and its surroundings. This technology allows a 2D LiDAR to generate precise information about its environment in the form of point clouds, providing valuable insights into its surroundings (Wang et al., 2020). Since the 2D LiDAR emits a rotating pulsed laser beam on a single plane, it can generate a 2D point clouds map. This map enables easy identification of the distances between the robot and its surroundings. However, the information provided by a 2D LiDAR may not be sufficient for a robot to navigate in a complex environment where obstacles may not be on the same plane as the 2D LiDAR. Consequently, the robot may miss information about obstacles and potentially collide with them.

To address this issue, a 3-dimensional (3D) obstacle avoidance system has been proposed.

The 3D obstacle avoidance system consists of several crucial components, including sensors to detect the surroundings and output three-dimensional spatial information in point clouds, such as 3D LiDAR and depth cameras. Middleware facilitates the exchange of information between the sensors and the computer (ROS2), and there is a post-processing step for the data obtained from the sensors. The robot's current surrounding environment is represented in a cost map. A cost map is a standard 2D grid composed of cells, each containing information regarding costs associated with unknown, free, occupied, or inflated areas. This cost map is subsequently analyzed during the search process to calculate a global plan or sampled to determine local control efforts. (Macenski, Moore, et al., 2023) In this project, a 3D map will be established to implement the 3D obstacle avoidance system.

1.2 Importance of the Study

In the industry, a 2D obstacle detection and avoidance system is widely used due to its popularity and stability. However, the information provided by a 2D LiDAR may not be sufficient for a robot to navigate in a complex environment where obstacles may not be on the same plane as the 2D LiDAR. The 3D obstacle detection and avoidance system has become increasingly important in the industry to overcome the challenges that robots must face in complex working environments. A 3D obstacle avoidance system provides more options for robots to avoid obstacles blocking their path and travel to the target destination without human interference. Since a 3D obstacle avoidance system can offer more information about the environment around the robots, conducting studies on 3D obstacle avoidance systems for robots to enable them to operate effectively in complex environments is crucial.

The results of this present study may have significant implications for both robot navigation and understanding the theory behind:

- The construction of a 3D map using the 3D environmental information provided by sensors.
- The algorithm for detecting obstacles from the cost map.

- Global and local path planning to generate a new path when the original path is blocked by obstacles.

1.3 Problem Statement

Problem statement for the current study of obstacle avoidance system for 3D robot navigation is summarized below:

A cost map is a standard 2D grid composed of cells, each containing information about the environment. Consequently, cost maps are essential for robots to perform obstacle detection and path planning. However, when dealing with an obstacle avoidance system in 3D robot navigation, complications arise due to the inherent 2D nature of generated cost maps.

- The information provided by a 2D LiDAR may not be sufficient for a robot to navigate in a complex environment where obstacles may not be on the same plane as the 2D LiDAR and cannot be detected by the robot.
- In the Nav2 library, the 2D cost map is generated using 3D environmental data sent from sensors. The algorithm compresses the z-axis data projects the z-axis data to a planar cost map. If any voxel along the z-axis is occupied, the corresponding grid cell in the 2D cost map is marked as occupied. However, when robots need to traverse specific terrains such as tunnels, the compressed and missing z-axis data causes the robots to mistakenly perceive the entire tunnel as an obstacle, hindering their ability to navigate through the terrain.

1.4 Aim and Objectives

The main aim of this study is to propose an algorithm to implement the obstacle avoidance system for 3D robot navigation. The specific objectives of this research were to:

- Investigate and find a method to construct a 3D map.
- Implement the 3D obstacle detection system from 3D map.

- Integrate the 3D obstacle detection system and path planning system in Nav2 to produce the obstacle avoidance system for 3D robot navigation.

1.5 Scope and Limitation of the Study

The scope of the study is to design a 3D obstacle avoidance system for land wheeled autonomous robots. Hence, the robot will move in a 2D plane and equipped with sensors that can provide information of the environment in 3D. The limitations of the study are:

- The algorithm designed for the 3D obstacle avoidance system is only portable for land wheeled autonomous robots. The obstacle avoidance system is not applicable to unmanned aircraft vehicles (UAV).
- The planned path to avoid the obstacle is different with the type of robots. For example, car-type robots and differential drive robots have different paths to avoid obstacles due to their type of motions. Hence, the obstacle avoidance system is only designed to the robots with specific motion type.

1.6 Contribution of the Study

This study proposed a potential solution for implementing an obstacle avoidance system for 3D robot navigation. Nowadays, the most popular and mature robot navigation system is a model that uses only 2D LiDAR. The robot with a pure 2D LiDAR system can only detect obstacles that are in the same plane as the LiDAR. The solution proposed addressed the limits of the robot's navigation by presenting a 3D visual SLAM method that can store 3D environmental data and an obstacle avoidance system that can detect 3D obstacles that 2D LiDAR cannot. As a result, the suggested obstacle avoidance algorithm allows the robot to safely explore a complicated 3D environment.

1.7 Outline of the Report

Chapter 1 introduces the project by outlining the AGV's existing navigation system and its constraints. The chapter also discusses the aim, objectives and problem statements of this project.

Chapter 2 provides a literature review of existing research related to the navigation system of a robot. This chapter presents a comprehensive overview of the components required to develop an obstacle avoidance system, such as the mapping process, cost map, controller and behaviour tree.

Chapter 3 outlines the methodology of this project. In this chapter, the process of starting a robot navigation will be explained in detail. The methodology consists of five main parts: Construction of a differential drive robot, preparation of the URDF file for the constructed robot, preparation of the drivers required for simulation and real-world sensors, setup of RTAB-Map SLAM to get a 3D map and setup of Nav2 for the obstacle avoidance system.

Chapter 4 presents the results and discussions for this project. The results section is divided into two sections, which are the evaluation of the mapping process and the evaluation of the navigation outcome. Both simulation and real-world navigation results will be presented in the results section. In the discussion section, the possible errors that will cause the imperfection in the mapping and navigation process and their possible solutions have been discussed.

Chapter 5 concludes the project by discussing the findings and their significance. Additionally, it proposes several enhancements for improving the performance and applicability of the RTAB-Map 3D SLAM and obstacle avoidance algorithm in the future.

CHAPTER 2

LITERATURE REVIEW

2.1 Introduction

A map is required for robots to efficiently recognise obstacles and plan their paths. It offers the information required for a robot to detect navigable zones and obstacles to avoid. By analyzing the map, the robot can determine permissible paths and mark areas to be avoided to facilitate safe and efficient navigation. The concept of a 3D mapping methods and its related algorithms has gained the attention of researchers to enable robots to adapt to complex environments. This literature review has two primary objectives. Firstly, it aims to provide a deeper understanding of the algorithms and techniques used in creating a 3D map. Secondly, it investigates the process of integrating the constructed 3D map into the obstacle avoidance system for 3D robot navigation. The outline is structured as follows: Firstly, the review provides an overview of the theory behind constructing a 2D cost map and an obstacle avoidance system. Secondly, it explores the various methods that researchers have employed to create the 3D map. Finally, the review offers insights into the obstacle avoidance system by utilizing the constructed 3D map.

2.2 Nav2

Nav2 is the primary library that provides all the necessary tools and functions for robot navigation. The tools in Nav2 enable users to create a 2D cost map from sensor data, utilize various planners and controllers for robot navigation, and provide a workspace for users to customize their plugins for specific applications. Therefore, it is essential to study the working principles of Nav2 and the algorithms it uses, as the 3D obstacle avoidance system will be developed based on Nav2 concepts.

2.3 Mapping

In robot navigation, the robot must first understand its environment through mapping before it can perform navigation tasks. The sensor data is used to estimate the state of the robot and create an accurate representation of the environment in a mapping process. The map generated through mapping process is crucial for tasks like path planning and localization. In ROS2, the generated map is represented as an occupancy grid, consisting of cells that represent the probability of occupancy. (Macenski, Moore, et al., 2023)

When conducting mapping, the robot requires knowledge of its own position to create an accurate global map. Simultaneous Localization and Mapping (SLAM) is the process in which the robot simultaneously maps and localizes itself (Macenski et al., 2023). There are two conventional tools for robots to perform SLAM: Cartographer and SLAM Toolbox.

Real-time SLAM capabilities are offered by Cartographer. It continually integrates laser scans into submaps over short intervals via pose-graph optimisation, producing locally accurate maps that show the most recent measurements. These localised submaps are used for scan matching, whereby scans are added to the submap at their best-estimated locations. A submap is improved by going through a loop closure procedure once it has been finalised, which compares it with other submaps and local scans. However, professional optimization and high-quality odometry hardware platforms are required in Cartographer to achieve the accuracy up to 3-5 cm in the mapping process. It's important to note that development and support for Cartographer have ceased, making it not recommended for use in SLAM for robots (Macenski et al., 2023).

On the other hand, SLAM Toolbox is one of the 2D SLAM methods that is designed based on the OpenKarto SLAM library. Like Cartographer, the pose-graph optimization is used in SLAM Toolbox to match current sensor readings with previous scans and provide a corrected pose of the robot. The OpenKarto SLAM library used in SLAM Toolbox has been updated to enhance scan matching speed and offer greater flexibility in optimization parameters (Macenski et al., 2023). Figure 2.1 provides an example of a map generated by SLAM Toolbox. However, it's important to note that SLAM Toolbox is limited to planar mapping and may not be suitable for 3D navigation in environments such as climbing ramps between floors.

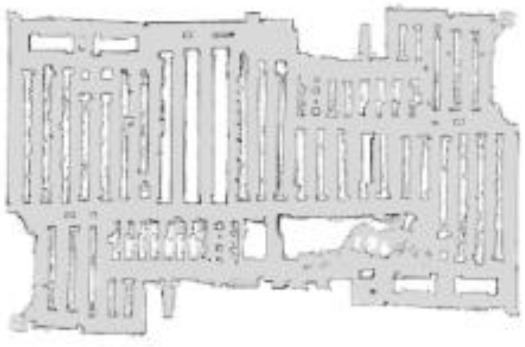


Figure 2.1: Map generated by SLAM Toolbox (Macenski, Moore, et al., 2023)

The information provided by a 2D map obtained from the 2D SLAM Toolbox is insufficient for a robot to perform 3D navigation tasks, such as climbing ramps or navigating through tunnels. Additionally, sensors that provide 3D environmental information, like stereo and RGB-D cameras, have become more affordable. Therefore, it is essential to implement 3D SLAM to represent the environment in three dimensions. Two popular 3D mapping techniques adopted in autonomous robotics systems are OctoMap and RTAB-Map.

OctoMap, a 3D modeling technique, offers a volumetric representation of space. This approach relies on octrees and employs probabilistic occupancy estimation. OctoMap is created under a 3D occupancy grid mapping framework that recursively divides the information in 3D space into smaller cube-shaped spaces called octants. These octants indicate whether the space within them is occupied or free, allowing the construction of a 3D occupancy grid without prior knowledge of the environment's extent. Octrees and OctoMaps can be updated dynamically (K. T. D. S. De Silva et al., 2018).

Hornung et al. (2013) compared 3D maps generated using different methods, including point clouds, elevation maps, multi-level surface maps, and the OctoMap method. The evaluation was based on probabilistic representation, modeling of unmapped areas, and efficiency in terms of access time and memory consumption. Point clouds are memory-inefficient due to the storage of a large number of measurement points. Furthermore, they cannot distinguish between areas without obstacles and unmapped regions, nor do they provide a method for probabilistic fusion of multiple measurements (Hornung et al., 2013). While elevation maps and multi-level surface maps are memory-efficient, they fail to accurately represent complex 3D environments, such as the example shown in Figure 2.2. In contrast, OctoMap allows efficient and probabilistic updates for both occupied and unoccupied spaces while maintaining minimal memory consumption (Hornung et al., 2013)

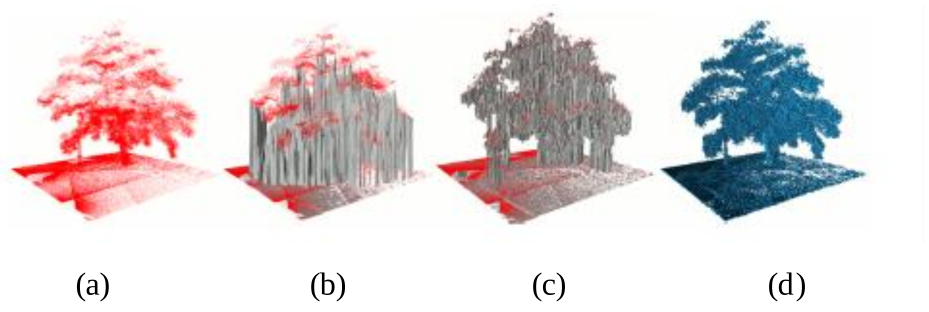


Figure 2.2: The comparisons among different mapping method: (a) Point Cloud, (b) elevation map, (c) multi-level surface map and (d) OctoMap (Hornung et al., 2013)

The Real-Time Appearance-Based Map (RTAB-Map) one of the 3D SLAM methods by using the data from a depth camera. RTAB-Map can represent objects as they appear, preserving the real shapes, features, and colors of the environment to a significant extent. Additionally, the 3D map generated from RTAB-Map will record the path travelled and current position of the robot. The 3D SLAM approach provided by RTAB-Map can support the creation of incremental maps and facilitates loop closure detection. Loop closure detection is a process to determine if the robot has previously visited a particular location in an environment. The loop closure detection is achieved by comparing the probability of image data originating from the same location and subsequently updates the map (K. T. D. S. De Silva et al., 2018).

RTAB-Map offers extensive compatibility with various input sensors, including stereo, RGB-D, odometry and 2D/3D lidar data. Within the Robot Operating System (ROS), RTAB-Map has long served as a versatile alternative to 2D SLAM and the 3D map generated from RTAB-Map can be used for 3D mobile robot navigation. (Merzlyakov and MacEnski, 2021) Unlike conventional SLAM methods that create feature-based maps, RTAB-Map generates dense 3D and 2D representations of the environment. These representations can be utilized similarly to 2D SLAM by using 2D lidar only, making it a seamless and valuable replacement for the conventional methods without requiring additional post-processing. (Merzlyakov and MacEnski, 2021) Consequently, RTAB-Map is a prevailing mapping method used for research purpose and service robot applications as an "alternative" to 2D SLAM. The construction of RTAB-Map is often in conjunction with LiDAR or the data of depth camera. RTAB-Map offers all the essential features required for the navigation of a robot. (Merzlyakov and MacEnski, 2021)

K. T. D. S. De Silva et al. (2018) conducted a comparison between two 3D mapping methods, OctoMap and RTAB-Map. The hardware and software used in the research conducted by K. T. D. S. De Silva et al. are listed in Table 2.1. In addition to the required hardware and software, the research environment for 3D mapping was set up with features such as inclined surfaces, vertical surfaces, and horizontal surfaces. The setup of the environment is shown in Figure 2.3.

Table 2.1: Hardware and software required in the research (K. T. D. S. De Silva et al., 2018)

Hardware	Software
Microsoft Kinext for Xbox 360 (RGB-D sensor)	ROS Kinetic Kame
Kobuki, Yujin Robot’s mobile research base	Kobuki, OctoMap and rtabmap_ros (software libraries)
Laptop with intel-i7 processing power	RVIZ (ROS Visualizer)

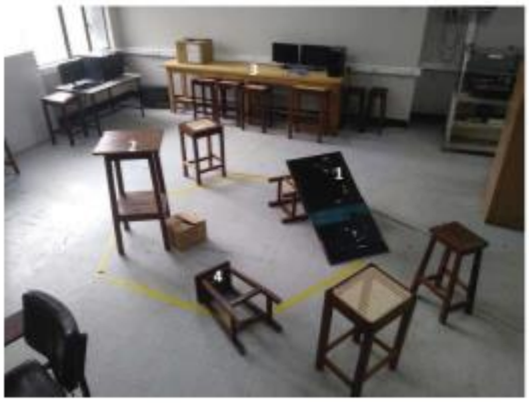


Figure 2.3: The environment setup (K. T. D. S. De Silva et al., 2018)

In the research conducted by K. T. D. S. De Silva et al., several criteria were used to evaluate the performance of these 3D mapping methods. These criteria include portability of the software used in the project, real-world representation in generated 3D map, navigation and path planning, the ability to handle dynamic obstacles, sharing built maps among other robots and the potential for odometry error corrections. The performance of these two 3D mapping methods under different criteria is summarized in Table 2.2 (K. T. D. S. De Silva et al., 2018).

Table 2.2: Performance of the two 3D mapping methods under different criterion (K. T. D. S. De Silva et al., 2018)

	OctoMap	RTAB-Map
Portability	Required extra procedure	Easier to setup compared to OctoMap
Real-world representation	No	Yes
Navigation and path planning	Yes	Yes
Capability to handle dynamic obstacle	Yes	Yes
Sharing built maps among other robots	Provides better method for detecting the profile of obstacle	-

Potential of odometry error corrections	-	As RTAB-Map is a SLAM with loop closure detection, it can be easily used for odometry error correction

In summary, there are benefits and drawbacks to both the OctoMap and RTAB-Map 3D mapping techniques for mapping and navigating, particularly in uneven terrain. When it comes to obstacle recognition, collision-free navigation, and sharing mapping data among robots, OctoMap performs exceptionally well. This is because the octree structure of the OctoMap provides a lightweight mathematical solution for these applications. On the other hand, RTAB-Map offers significant advantages when it comes to real-world representation in the map. RTAB-Map's real-world representation can play a crucial role in the decision-making processes of intelligent robots and operators.

Both systems can be integrated to create a robust 3D mapping algorithm. Initially, RTAB-Map is created directly from the Kinect camera using the SLAM approach to build the 3D map. Subsequently, the RTAB-Map data can be used to generate an OctoMap. The OctoMap generated by RTAB-Map can be utilized while preserving the real-world representation provided by RTAB-Map as the default 3D map during the robot navigation process and the data sharing operations. (K. T. D. S. De Silva et al., 2018) The conversion of RTAB-Map into OctoMap could be achieved as a standalone application for a single robot. Additionally, the incorporation of odometry error correction by RTAB-Map may contribute to more precise 3D maps and occupancy grids. Therefore, it is essential to integrate both mapping methods to generate a robust 3D map (K. T. D. S. De Silva et al., 2018).

2.4 Cost Map

When a robot navigates in a dynamic environment, it needs to obtain information about the environment to react effectively. This is referred to as the reactive behaviour of the robot. In ROS2 (Robot Operating System 2), cost maps, which is also called risk maps are employed as environmental models to consolidate results from potentially numerous algorithms. Since the conventional mapping method for robots is in two dimensions, the cost map is also in 2D, as it is constructed based on the map generated by different mapping methods. A cost maps is derived from probability-based occupancy grids and representing the world model as a 2D grid with associated costs. They allow planning algorithms to select low-cost paths over high-cost ones and include special values for unknown and occupied cells to ensure robust robot behaviour (Macenski et al., 2023).

Cost maps serve as the configuration space for planning algorithms, balancing fidelity and efficiency across multiple computer platforms. However, they result in linear memory usage as the number of cells increases. Quad-trees have been proposed for multi-resolution data storage, although they may not yield significant memory benefits in some scenarios. In practice, cost map cells are typically 0.05 meters by 0.05 meters in size, striking a compromise between displaying the environment with appropriate fidelity and managing computing resources efficiently (Macenski et al., 2023).

Traditionally, the cost map is structured as a monolithic cost map where all data is stored in a single grid of values. This approach has been popular due to its simplicity, as there's only one location for reading and writing values. However, it results in a loss of semantic information about the values in the cost map, making it challenging to properly maintain the cost map across cycles. Hence, a new approach to constructing the cost map called layered cost maps was proposed by Lu et al. The Layered cost map, which provides a systematic and extendable approach, is used to update and maintain the cost map in ROS 2. (Lu et al., 2014) It consists of a list of dynamically loaded layers that represent different data sources, methods, or outcomes. The primary cost map is successively updated by polling these layers to incorporate new information into the grid during each update cycle. Some important layers in the cost map include the static layer, obstacle layer, voxel layer, spatio-temporal voxel layer, inflation layer, keepout layer, and speed limiting layer (Macenski et al., 2023).

The static layer provides information about known obstacles from the map generated after a mapping process. It serves as the base layer, offering initial environmental information. Although named "static," it can change over time due to map sharding or updates to the environment. The obstacle layer stores data recorded from sensors such as LiDAR and depth cameras in a two-dimensional grid. The obstacle layer is suitable for 2D rangefinder such as LiDAR but may have limitations when processing 3D data from depth camera. Pre-filtering may be important for noisy sensor data to reduce noise in the cost map. The voxel layer is similar to the obstacle layer. It can capture the 3D information of the environment and 3D ray-traces measurements. It is suitable for 3D sensors such as depth camera and 3D laser scanners. The voxel layer uses a voxel grid model to represent the environment in a non-probabilistic manner, with a limited maximum height. While it conducts 3D raytracing, it is more computationally intensive than the obstacle layer. Other layers such as the range layer, inflation layer, keepout layer, and speed limiting layer have their filters to restrict the regions that the robot can travel to and control the speed of the robot when traveling through specific regions (Macenski et al., 2023).

The conventional cost map for robots is a 2D cost map. However, when a robot needs to perform 3D navigation, the information provided by a 2D cost map is insufficient. To address this limitation, Withey and Matebese (2021) proposed a method to construct a 3D cost map based on OctoMap. The OctoMap-based 3D cost map offers several advantages, including the ability to classify cells into distinct categories such as Drivable, Obstacle and Unsafe regions. Furthermore, the features of the OctoMap such as efficient memory consumption is retained to enable the compression of octants with the same cell type in all child cells. This hierarchical representation allows for rapid retrieval of specific cell types within the cost map octree hierarchy.

The construction of the 3D cost map is based on OctoMap, which employs an octree-based environment representation. In octree structure, a 3D cubic region is recursively divided to octants and finally form a tree structure. Starting from the root node, which encompasses the entire volume, and extending to the finest resolution leaf nodes, each node in the tree represents an octant. Every parent node has eight child nodes, one for each octant (Withey and Matebese, 2021).

The construction of a cost map involves several steps. Initially, the binary OctoMap is parsed, converting occupied cells into Obstacle cells and free cells into Drivable cells. Subsequently, within a safety radius, Unsafe regions are delineated by identifying Drivable cells near Obstacle cells. Drivable cells adjacent to unexplored areas are then reclassified as Frontier cells. Lastly, cells susceptible to WiFi loss are marked as WiFi Loss cells. The resulting cost map can be used for 3D path planning (Withey and Matebese, 2021).

In summary, the OctoMap-based cost map offers a practical method for segmenting a 3D environment into discrete regions with consistent attributes, facilitating tasks such as 3D path planning. It is computationally efficient and can be greatly compressed for inter-process communication, such as in a ROS framework. Similar to its 2D counterpart, the 3D cost map represents various cell types, including Obstacle, Unsafe zones surrounding obstacles, Drivable or unoccupied cells, Frontier cells at the interface of scanned and unscanned regions, and WiFi Loss status. These attributes are stored as a pair of bit fields (Withey and Matebese, 2021).

2.5 Planners and Controllers

When the information of the surrounding environment has been captured and represented in a cost map, the robot needs to use the constructed map to do the navigation from beginning to destination. The algorithm to plan the path for the robots in a pre-constructed map is called planner and controller. According to the size of region considered during path planning, path planning for a robot can be classified as global path planning and local path planning. The planner in Nav2 is the global path planner while the controller in Nav2 is the local trajectory planner.

Global path planning seeks to determine the best paths through an area while considering various levels of accuracy, ranging from grid cells to continuous sequences. Global route planning in Nav2 is focused on kinematic limitations, whereas local trajectory planning considers system dynamics and other factors. The goal is to establish a global route through the environment, which will then be fine-tuned by a local trajectory planner. The optimal path in global path planning varies depending on the application, however these techniques are often based on extensions of Dijkstra's algorithm, which seeks paths with the lowest cost. The described cost map is the primary determinant of cost in Nav2. Typically, the shortest path that avoids obstacles is the result.

There are two types of global path planners: search-based and sampling-based. In traditional path planning algorithms, the information of the environment must be known before the path search can be performed. However, this increases the amount of computation exponentially when doing the path planning in the high-dimensional and spending more time to generate a path. Rapidly-exploring random tree (RRT) and other sampling-based planners are good for higher-dimensional planning issues since they produce nearly optimum paths. However, the paths generated from sampling-based path planning method typically require post-processing. In 2D state spaces search-based planners are faster in creating really optimum pathways with acceptable and consistent heuristics. They provide predictable execution times, which is critical for dynamic replanning in large-scale complicated systems. The path planners in Nav2 mainly use search-based global planning methods, as these planners are primarily designed for the 2D cost map (Macenski et al., 2023).

The Local Trajectory Planner is used to translate a global path into velocity commands for a mobile platform. This planner creates collision-free paths and sends velocity commands to the microcontroller to control the speed of motors. The local trajectory planners are referred as controllers in Nav2 as the planners are highly related to the mobile robot base and dependent on the kinematics of the robot. (Macenski, Moore, et al., 2023) Trajectory planners vary according to robot type and application. They frequently necessitate extensive customization to fit the kinematic and dynamic properties of a certain robot. A variety of local trajectory planners are provided in Nav2 to accommodate diverse vehicle types, such as differential drive, omnidirectional drive, and legged robot. The Dynamic Window Approach (DWA) and Model Predictive Path Integral (MPPI) are some of the popular controllers that are used for local trajectory planners. (Macenski et al., 2023).

2.6 Behaviour Tree

In Nav2, the planning, control, high-level behaviours, recoveries, and other internal robot navigation duties are all orchestrated via configurable behaviour trees (BTs). This differs from earlier methods that utilized state machines and hard-coded logic. Behaviour trees have grown in popularity thanks to user surveys and offer more versatility. The BT Navigator is in charge of handling requests and carrying out actions listed in the behaviour tree. Behaviour trees are displayed as XML files containing BT nodes that are dynamically loaded at runtime. This enables the system to change to accommodate various behaviours or tasks as required. For modularity, the majority of BT nodes connect to a distant server via a ROS 2 interface, but this is not required. In addition to the rudimentary BT nodes provided by Nav2, users can also build their own nodes and trees. (Macenski, Moore, et al., 2023)

The three main categories of behaviour trees offered by Nav2 are "navigate to pose," "navigate through poses," and "task-specific applications," each with configurable options to deal with different circumstances. Basic pose-to-pose navigation is made easier by the "navigate-to-pose" behaviour trees. These trees provide reliable replanning at fixed intervals, proportionate to robot speed, based on distance travelled, or if the goal is modified. Recovery behaviours are frequently included in practical BTs to address common navigation problems brought on by changing situations, ambiguity, or constrained surroundings. For component failures, ROS 2 BTs can launch context-specific operations, and for system-level failures, they have a global recovery branch. Advanced BTs integrate characteristics like goal patience and mixed replanning to overcome problems like oscillations and momentary blocks. (Macenski, Moore, et al., 2023)

2.7 Summary

In conclusion, the literature concerning essential components in the 3D obstacle avoidance system, such as mapping methods, types of cost maps, planners, and controllers, has been thoroughly analyzed. In the mapping method, conventional 2D mapping approaches, such as the SLAM Toolbox, are insufficient for generating a 3D map. OctoMap and RTAB-Map have emerged as popular mapping methods for generating 3D maps. These 3D maps need to be transformed into cost maps to enable the robot to plan paths based on the cost of each cell in the cost map. To generate paths within an environment, the planner and controller in Nav2 will be used for global path planning and local trajectory planning respectively. The BT navigator will also be employed to handle requests and execute actions listed in the behaviour tree, thereby facilitating the moderation of navigation tasks. Future research should continue to explore and develop algorithms capable of constructing 3D maps and local trajectory planners to realize the obstacle avoidance system for 3D robot navigation.

CHAPTER 3

METHODOLOGY AND WORK PLAN

3.1 Introduction

In this section, the methodology employed for the creation of the 3D obstacle avoidance system will be presented. The primary objective of this study is to develop a method for the generation of a 3D cost map and the subsequent implementation of a 3D obstacle avoidance system based on this cost map. Understanding the underlying principles of cost map formulation and robot avoidance systems is considered vital for the successful development of our 3D obstacle avoidance system. The methodology includes a series of preparatory steps, which include prototype preparations, the setup of the software environment, and the readiness of sensors for the system.

3.2 Requirements

The development of the cost map and the obstacle avoidance system in three-dimensional requires:

- A computer with Ubuntu 22.04 LTS (Jammy Jellyfish) operating system
- Installation of ROS2 Humble
- Installation of library for robot navigation (Nav2)
- A sensor that can generate the 3D environmental data
- Simulation software such as Gazebo and RViz
- A differential drive robot with the computer that preinstalled ROS2 Humble on it.

3.3 Implementation

The ROS2 (Robot Operating System 2) is a collection of software libraries and tools designed for various robot applications. The fundamental structure of ROS2 comprises multiple nodes, each serving as a program for specific tasks, such as sending commands to a microcontroller. These nodes communicate with one another through "communication protocols" provided by ROS2, such as topics, services, and actions, to facilitate robot applications. In this project, the 3D obstacle avoidance system will be developed on ROS2 Humble. Each version of ROS2 is best suited for specific operating systems. ROS2 Humble is highly compatible with the Ubuntu 22.04 Jammy Jellyfish Linux operating system. Therefore, to implement ROS2 on the robot, the robot must be equipped with a computer running the Ubuntu 22.04 Jammy Jellyfish Linux operating system. Figure 3.1 shows the example node graph generated by using the rqt tool in ROS2.

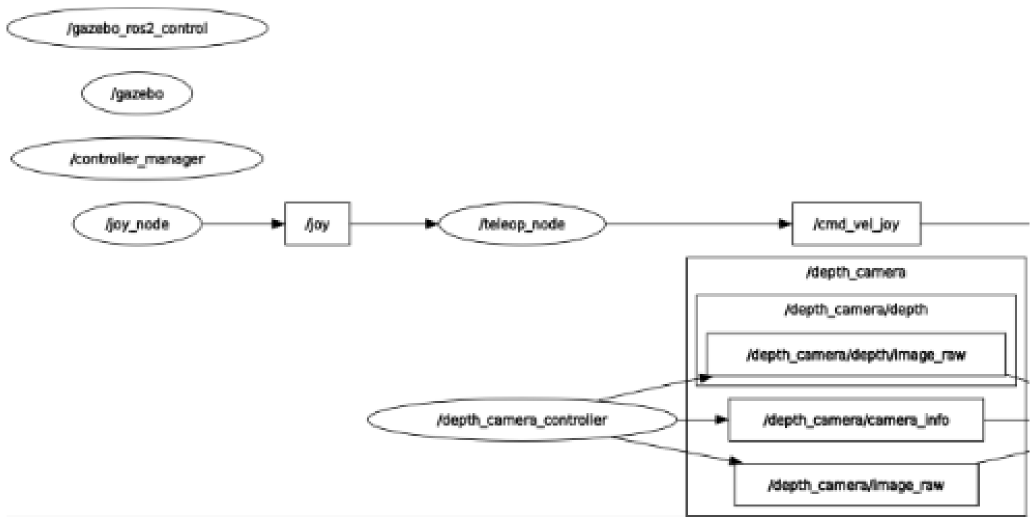


Figure 3.1: The example node graph in ROS2 network

Similar to ROS2, Nav2 is a collection of software libraries and tools designed for robot navigation, specifically tailored for ROS2. Nav2's navigation algorithms are implemented as ROS action server plugins. The primary components of the navigation stack are the two action servers: the planner and the controller servers. Each algorithm plugin on these servers is configured for specific tasks or robot states. The behaviour tree of the navigation system or application server allows for the selection of planning methods, providing flexibility in adapting to various robot tasks and environments. Besides the plugins for planner and controller servers, Nav2 also offers tools for users to develop plugins for cost maps (2D), behaviour trees and navigation. (Macenski, Singh, et al., 2023) Each distribution of ROS2 has its own set of Nav2 dependencies. In this project, the Nav2 version compatible with ROS2 Humble was installed on the computer for the development of the 3D obstacle avoidance system.

Once ROS2 and Nav2 are installed, simulators such as Gazebo and Rviz in ROS2 will be utilized to simulate robot operations, including mapping, creating cost maps, and navigation within the created map. Developers can test algorithms in these simulators and make instant adjustments when issues arise during simulations. To ensure that simulation results closely resemble real-world testing results, several crucial components must be configured in the simulation, including sensors, Unified Robot Description Format (URDF) for the robots, and navigation plugins. Figure 3.2 shows the flowchart depicting the implementation of this study.

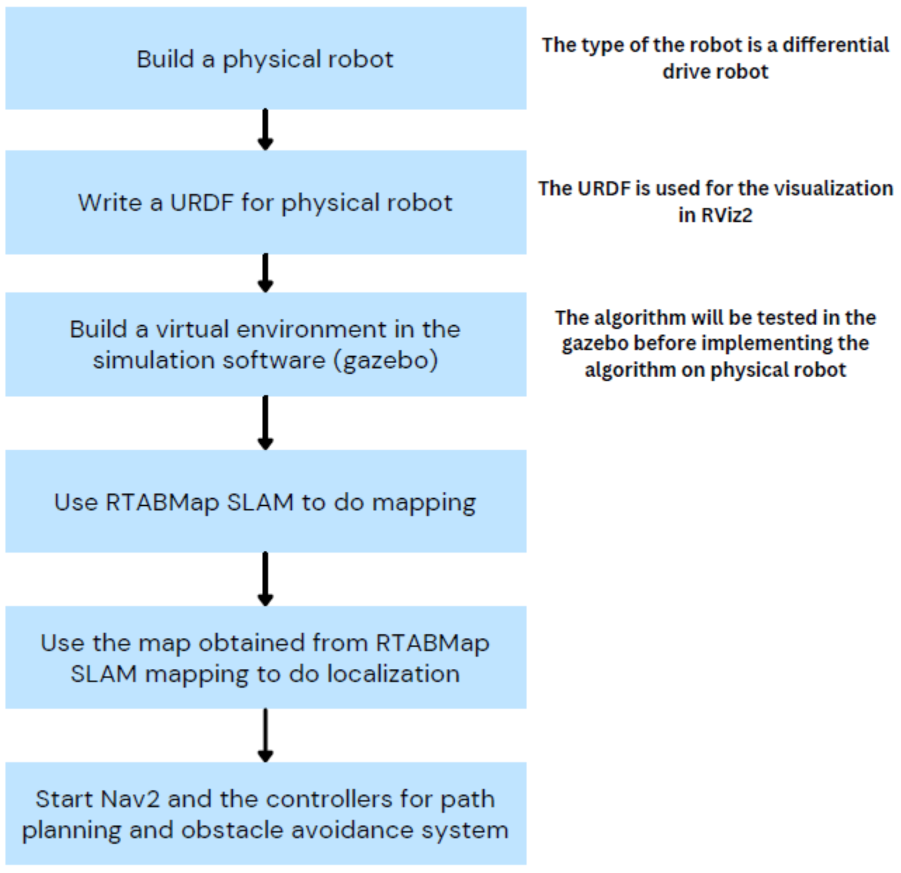

Build a physical robot
The type of the robot is a differential drive robot
Write a URDF for physical robot
The URDF is used for the visualization in RViz2
Build a virtual environment in the simulation software (gazebo)
The algorithm will be tested in the gazebo before implementing the algorithm on physical robot
Use RTABMap SLAM to do mapping
Use the map obtained from RTABMap SLAM mapping to do localization
Start Nav2 and the controllers for path planning and obstacle avoidance system

Figure 3.2: The flowchart of the implementation of this project

Initially, a differential drive robot was constructed for the study. Subsequently, a Unified Robot Description File (URDF) was created to define the physical attributes of the real robot and enable visualization of the robot in

Rviz2. The URDF contained detailed specifications such as the robot's dimensions, wheel diameter, and the relative positions of sensors to the robot base. The robot base, serving as a virtual reference point on the robot's body, was established to accurately interpret sensor data in relation to the robot's movements. This ensured precise representation of the robot's motion within Rviz2. After that, a simulation world was created to evaluate 3D robot navigation algorithms. Before initiating the robot navigation process, the robot underwent a mapping procedure to gather environmental data necessary for navigation. The sensor data obtained during the mapping process was then utilized to generate a map that will be used in the path planning for robot navigation. As the robot navigates in an environment, its position on the map changes over time. Sensors such as the Inertial Measurement Unit (IMU) and encoders are employed to continuously update the robot's position on the map. Additionally, LiDAR and depth cameras are utilized to compare the current environment with the mapped data, enabling the robot to adapt to changes and accurately localize itself on the map. This iterative process of updating the robot's position on the map is known as robot localization. Once the map is generated, it serves as a basis for robot navigation tasks such as path planning and obstacle avoidance. The functions provided by Nav2, such as the packages to construct a 2D cost map and controller for obstacle avoidance systems, will be implemented in the robot navigation process.

3.3.1 Construct a physical robot

In this project, a differential drive robot was constructed. The robot comprises two driven wheels, two motors, one caster wheel and a chassis made of aluminium profile and acrylic plate. Typically, the two driven wheels are positioned at one end of the robot, towards the front, to facilitate climbing slopes. Figure 3.3 illustrates the physical robot built for this project.

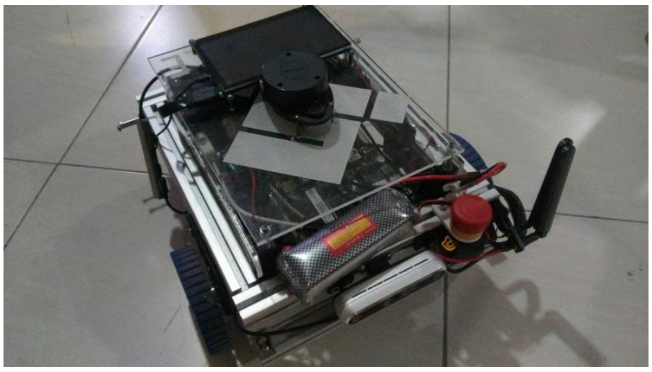


Figure 3.3: The robot built for this project

The physical robot is equipped with various components, including a microprocessor, a microcontroller, a dual-channel motor driver, sensors, and a battery. An emergency button is installed to cut off power supplied to the motor driver in case the robot loses control. For safety reasons, the microprocessor and motor driver are powered separately.

The microprocessor chosen for this robot is the CAPA55R i7, equipped with 32GB of RAM and a 128GB SSD. Before installing the necessary libraries for this project, the microprocessor is set up with the ROS2 Humble operating system, based on Ubuntu 22.04 Jammy Jellyfish Linux. A microcontroller is employed to communicate with peripherals such as the motor driver and encoders. This ensures real-time responsiveness, enabling tasks such as sending PWM signals to motors and receiving encoder pulses. The microcontroller used in this project is Arduino Nano. The dual-channel motor driver allows the microcontroller to control two motors by sending PWM signals to the motor driver.

Sensors play a crucial role in environment detection, mapping, and localization for the robot navigation. For this project, a 2D LiDAR (RPLiDAR A1M8) and a depth camera (Realsense D455) are utilized to gather environmental data. The Realsense D455 depth camera includes a built-in IMU. Both the LiDAR and depth camera are integrated with ROS2 so that the sensors can transmit data over the ROS2 network. The depth camera is mounted at the front of the robot, while the LiDAR is mounted on top. This configuration ensures optimal data collection without obstruction from the robot's body.

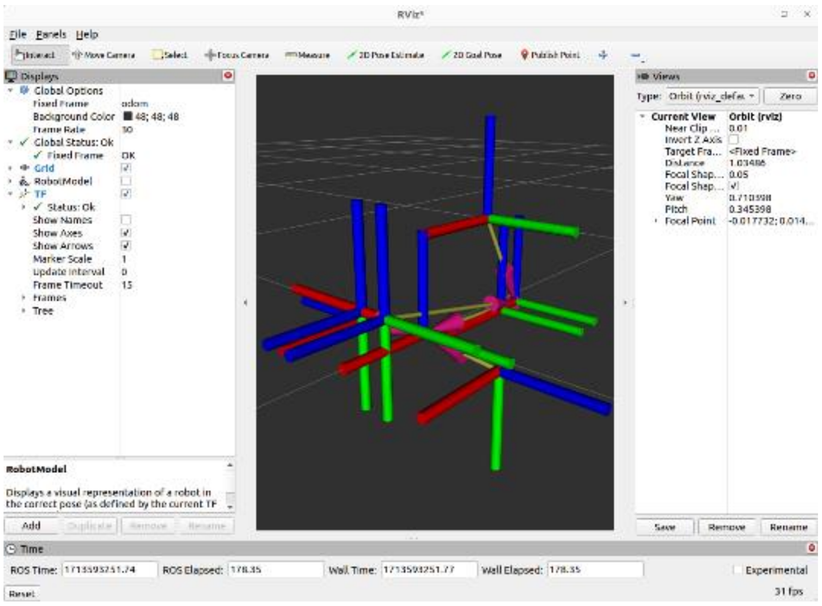
3.3.2 Write Unified Robot Descriptions File (URDF) for the robot

When a robot needs to perform a simple operation, like moving forward by one meter, its location within an environment change. The final position of the robot must be calculated using data from sensors such as motor encoders. Therefore, to accurately determine the robot's location after navigating to a specific point, parameters provided in the URDF file such as wheel diameter and wheel separation must be precise to minimize errors in distance travelled during linear or rotational motion. The placement of sensors, such as depth camera and LiDAR, on the robot relative to its chassis in URDF also affects the robot's localization on the map. If the sensor locations recorded in the URDF do not match the actual locations on the physical robot, errors accumulate over time. The accumulation of errors will ruin the mapping process and hinders navigation processes like path planning and obstacle avoidance as the robot cannot accurately position itself within the environment.

In ROS2, there exists a library known as the tf library, which facilitates the management of coordinate frames for individual components and the transformation of data within the entire robot system. This library enables users to confidently access data within specific frames without needing to understand all the coordinate frames within the system. By utilizing the tf library, users can integrate data from various sensors with different coordinate frames to accurately determine the location of the robot. (Foote, n.d.) The tf library was initially designed for ROS1 and the tf library designed for ROS2 is called tf2.

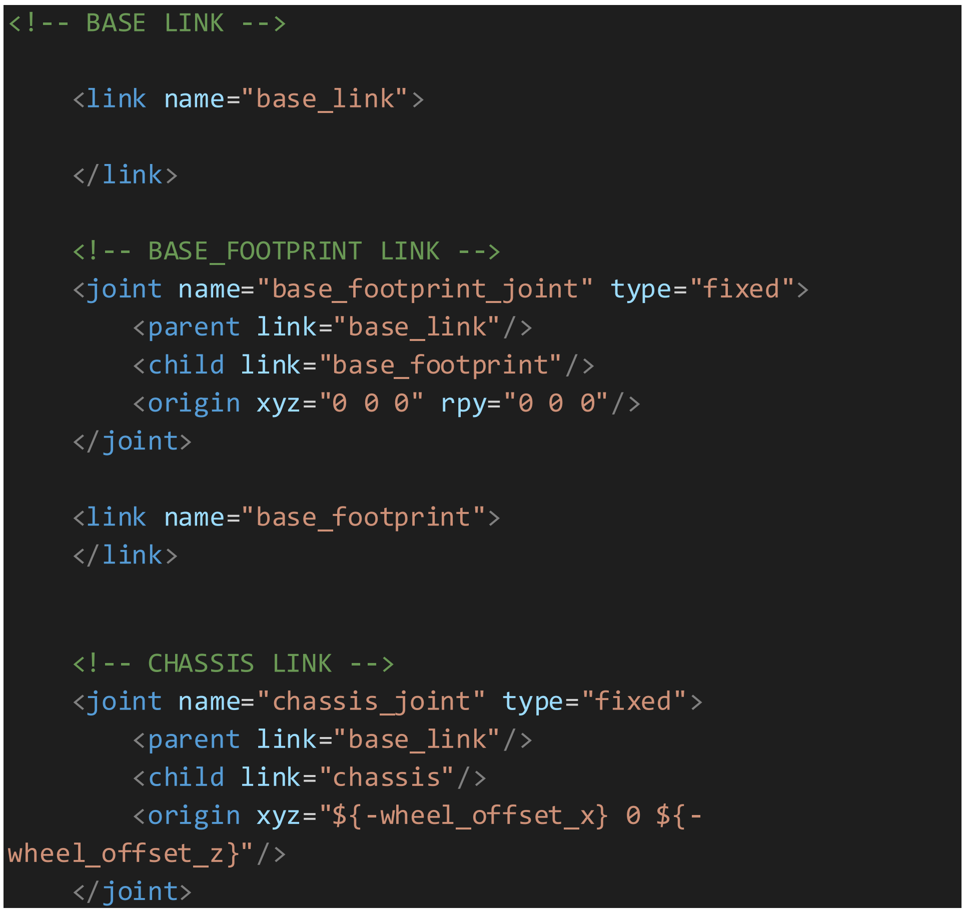
In this project, the robot_state_publisher package is used to publish the state of the entire robot to tf2. A URDF file that defines the coordinate frames of different components must be prepared and provided as a parameter to the robot_state_publisher package. The robot_state_publisher processes the URDF and publishes the state of the robot systems to establish the relationships among coordinate frames in tf2. The robot_state_publisher will subscribe to the joint_states topic to update the state of individual joints. For instance, as the robot moves forward, the state of the chassis relative to the map origin changes and these joint state changes are monitored by the robot_state_publisher to ensure that the location of the robot on the map is continuously updated during navigation. The robot_state_publisher handles two types of joints: fixed and movable. Fixed joints, such as those between the LiDAR and chassis, are published to the /tf_static topic. The movable joints, like those between the chassis and wheels, are published to the regular /tf topic whenever the relevant joint state is updated in the joint_states message. (Sucan et al., n.d.)

URDF files are written in Extensible Markup Language (XML) and contain essential data such as the robot's dimensions and the positions of its wheels and sensors. However, extraneous details such as motor dimensions and the locations of microcontrollers and motor drivers are not required for the URDF as they do not influence the robot's localization during navigation. The URDF focuses only on recording the critical coordinate frames affecting localization, such as those of the depth camera and LiDAR. To simplify the URDF and enhance maintainability, the robot components are represented by simple shapes such as boxes, cylinders, and spheres. For example, the depth camera is approximated by a box, the LiDAR by a cylinder, and the chassis by a rectangular box. Additionally, the two wheels are directly connected to the chassis. Figure 3.4 depicts the coordinate frames of different robot components as outlined in the URDF, while Figure 3.5 displays excerpts of the URDF code.



```
Global Options
  Fixed Frame          odom
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
  Fixed Frame          OK
Grid                   ✓
RobotModel
TF                     ✓
  Status: Ok
  Show Names
  Show Axes            ✓
  Show Arrows          ✓
  Marker Scale         1
  Update Interval      0
  Frame Timeout        15
  Frames
  Tree

RobotModel
Displays a visual representation of a robot in
the correct pose (as defined by the current TF

Type: Orbit (rviz_defa     Zero
Current View    Orbit (rviz)
  Near Clip ...   0.01
  Invert Z Axis
  Target Fra...   <Fixed Frame>
  Distance        1.03486
  Focal Shap...   0.05
  Focal Shap...   ✓
  Yaw             0.710398
  Pitch           0.345398
  Focal Point     -0.017732; 0.014...

ROS Time: 1713593251.74   ROS Elapsed: 178.35   Wall Time: 1713593251.77   Wall Elapsed: 178.35   Experimental
31 fps
```

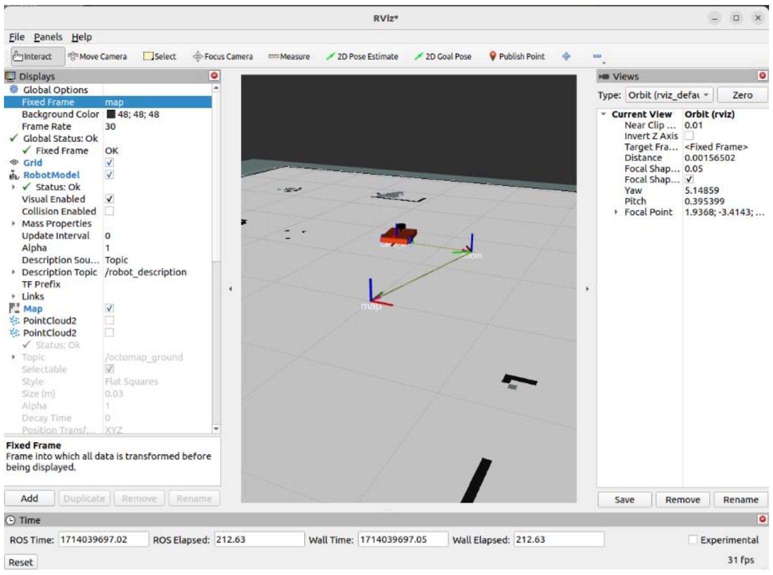
Figure 3.4: The coordinate frames of different components on the robot



```xml
<!-- BASE LINK -->

    <link name="base_link">

    </link>

    <!-- BASE_FOOTPRINT LINK -->
    <joint name="base_footprint_joint" type="fixed">
        <parent link="base_link"/>
        <child link="base_footprint"/>
        <origin xyz="0 0 0" rpy="0 0 0"/>
    </joint>

    <link name="base_footprint">
    </link>


    <!-- CHASSIS LINK -->
    <joint name="chassis_joint" type="fixed">
        <parent link="base_link"/>
        <child link="chassis"/>
        <origin xyz="${-wheel_offset_x} 0 ${-
wheel_offset_z}"/>
    </joint>
```

Figure 3.5: An excerpt of the URDF code (Newans, n.d.)

In addition to the robot's internal coordinate frames, the relationship between the robot and its surroundings is critical for successful mapping and localization operations. The ROS Enhancement Proposal (REP) 105 provides a complete overview of naming standards and semantic interpretations for coordinate frames between the robot and its environment. (Meeussen, 2010) REP 105 helps developers by specifying standards for drivers, models, and libraries, allowing for the integration and reuse of software components across several mobile platforms. REP 105 outlines the definitions and properties of three key coordinate frames: base_link, odom, and map. Base_link is affixed to the mobile robot base and providing a fundamental point of reference. Odom serves as a world-fixed frame suitable for short-term local reference, though it may exhibit some degree of drift over time. Map, another world-fixed frame, is better suited for long-term global reference, despite the potential for discontinuous jumps in position estimations. Figure 3.6 shows the relationship between the three key coordinate frames in a robot navigation process. It is important to adhere to the naming conventions outlined in REP 105 as it ensures compatibility with drivers and libraries developed under this convention, thereby facilitating seamless integration with the obstacle avoidance system being developed in this project.



Global Options	
Fixed Frame	map
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	✓
✓ Status: Ok	
Visual Enabled	✓
Collision Enabled	☐
Mass Properties	
Update Interval	0
Alpha	1
Description Sou...	Topic
Description Topic	/robot_description
TF Prefix	
Links	
Map	✓
PointCloud2	☐
PointCloud2	☐
✓ Status: Ok	
Topic	/octomap_ground
Selectable	✓
Style	Flat Squares
Size (m)	0.03
Alpha	1
Decay Time	0
Position Transf...	XYZ

Fixed Frame
Frame into which all data is transformed before being displayed.

Type: Orbit (rviz_defau

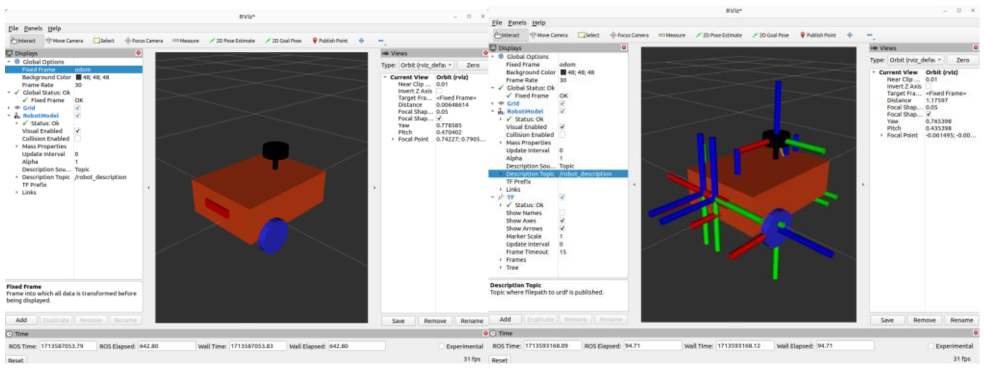
Current View	Orbit (rviz)
Near Clip ...	0.01
Invert Z Axis	☐
Target Fra...	<Fixed Frame>
Distance	0.00156502
Focal Shap...	0.05
Focal Shap...	✓
Yaw	5.14859
Pitch	0.395399
Focal Point	1.9368; -3.4143; ...

ROS Time: 1714039697.02 ROS Elapsed: 212.63 Wall Time: 1714039697.05 Wall Elapsed: 212.63 ☐ Experimental

31 fps

Figure 3.6: The relationship among the base_link, odom and map

After creating the URDF file for the robot, a launch file is necessary to execute the node provided in the robot_state_publisher package and pass the URDF file as a parameter to the node. In ROS 2, users can configure their system and execute it based on that description in the launch file that specifies arguments and configurations to pass to different nodes. Launch files can be written in various languages, including Python, XML, or YAML. In this project, the launch file is written in Python. Upon execution of the launch file, the '/robot_description' topic is published over the ROS 2 network. Users can then subscribe to this topic in Rviz2 to visualize the robot model. Figure 3.7 illustrates the robot model visualized in Rviz2.



```
Global Options
  Fixed Frame          odom
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
  Fixed Frame          OK
Grid                   ✓
RobotModel             ✓
  Status: Ok
  Visual Enabled       ✓
  Collision Enabled
  Mass Properties
  Update Interval      0
  Alpha                1
  Description Sou...   Topic
  Description Topic    /robot_description
  TF Prefix
  Links

Fixed Frame
Frame into which all data is transformed before being displayed.

Type: Orbit (rviz_defa

Current View           Orbit (rviz)
  Near Clip ...        0.01
  Invert Z Axis
  Target Fra...        <Fixed Frame>
  Distance             0.00648614
  Focal Shap...        0.05
  Focal Shap...        ✓
  Yaw                  0.778585
  Pitch                0.470402
  Focal Point          0.74227; 0.7905...

Time
ROS Time: 1713587053.79   ROS Elapsed: 642.80   Wall Time: 1713587053.83   Wall Elapsed: 642.80   Experimental
31 fps
```

```
Global Options
  Fixed Frame          odom
  Background Color     48; 48; 48
  Frame Rate           30
Global Status: Ok
  Fixed Frame          OK
Grid                   ✓
RobotModel             ✓
  Status: Ok
  Visual Enabled       ✓
  Collision Enabled
  Mass Properties
  Update Interval      0
  Alpha                1
  Description Sou...   Topic
  Description Topic    /robot_description
  TF Prefix
  Links
TF                     ✓
  Status: Ok
  Show Names
  Show Axes            ✓
  Show Arrows          ✓
  Marker Scale         1
  Update Interval      0
  Frame Timeout        15
  Frames
  Tree

Description Topic
Topic where filepath to urdf is published.

Type: Orbit (rviz_defa

Current View           Orbit (rviz)
  Near Clip ...        0.01
  Invert Z Axis
  Target Fra...        <Fixed Frame>
  Distance             1.17597
  Focal Shap...        0.05
  Focal Shap...        ✓
  Yaw                  0.765398
  Pitch                0.435398
  Focal Point          -0.061495; -0.00...

Time
ROS Time: 1713593168.09   ROS Elapsed: 94.71   Wall Time: 1713593168.12   Wall Elapsed: 94.71   Experimental
31 fps
```

Figure 3.7: The robot model visualized in Rviz2

3.3.3 Prepare simulation world to test obstacle avoidance system algorithm

Gazebo is a standalone simulation tool that can be used without ROS or ROS2. Hence, to use the ROS2 algorithm in Gazebo simulation, the gazebo_ros_pkgs is required to provide a bridge between Gazebo's C++ API and transport system and ROS 2 messages and services. (Open Source Robotics Foundation, 2014) The gazebo_ros_pkgs metapackage contains numerous subpackages, one of which is gazebo_dev, which provides a cmake configuration for the ROS distribution's default version of Gazebo. In addition, the gazebo_msgs package defines message and service data structures for interfacing with Gazebo in ROS 2. The gazebo_ros package provides useful C++ classes and functions for other plugins, whereas gazebo_plugins package includes a variety of Gazebo plugins that expose sensors and functionality to ROS 2, such as publishing ROS 2 images and providing interfaces for controlling differential drive robots. (Open Source Robotics Foundation, 2014) Hence, the gazebo_ros_pkgs metapackage is installed to test the obstacle avoidance algorithm in Gazebo simulation environment.

Figure 3.8 depicts the simulated world prepared in Gazebo for algorithm testing in simulation. Various obstacles are strategically placed within the simulation world to assess the algorithm's performance. The obstacles positioned lower than the LiDAR detection range is used to test the algorithm's ability to navigate around objects undetected by the 2D LiDAR. Additionally, an office table is included in the simulation world to evaluate the algorithm's capability to navigate beneath obstacles, facilitating the shortest traveling distance.

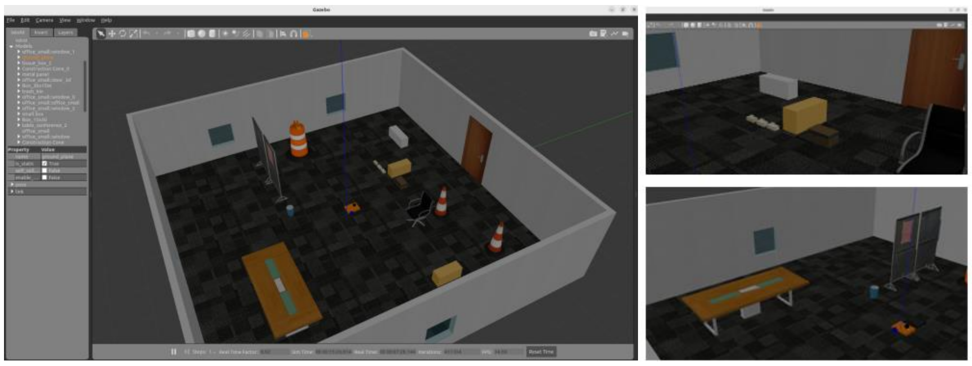
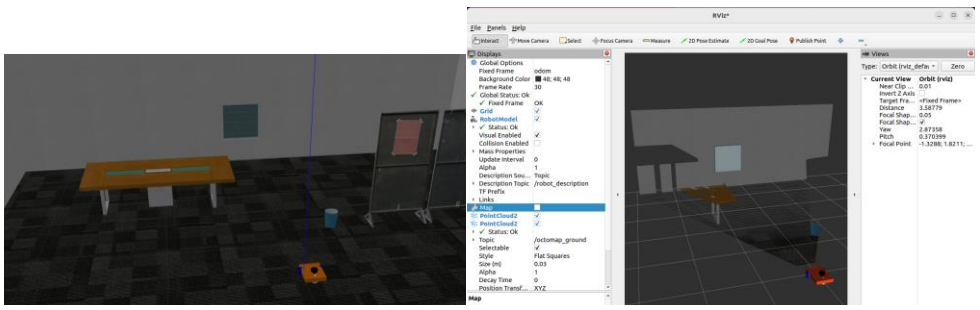


Figure 3.8: The simulation world prepared for this project

3.3.4 Setup driver for the microcontroller, LiDAR and depth camera

In this project, both the simulation and the physical robot testing are carried out to examine the functionality of the obstacle avoidance system for 3D robot navigation. In simulation, there are virtual drivers, which are also called controllers for the LiDAR and depth camera can be found from gazebo_ros_pkgs metapackage. The virtual drivers will allow the devices in Gazebo such as LiDAR and depth camera to function as the real LiDAR and depth camera and capture the data of the virtual world in Gazebo and publish the data to ROS2 network. The type of controller needs to be specified in URDF of the robot to implement the simulation of the LiDAR and depth camera in Gazebo. Figure 3.9 shows the data in PointCloud2 message type collected by the depth camera in simulation and presented in Rviz2.



```
Global Options
  Fixed Frame          odom
  Background Color     48; 48; 48
  Frame Rate           30
✓ Global Status: Ok
  ✓ Fixed Frame        OK
Grid                   ✓
RobotModel             ✓
  ✓ Status: Ok
  Visual Enabled       ✓
  Collision Enabled    ☐
  Mass Properties
  Update Interval      0
  Alpha                1
  Description Sou...   Topic
  Description Topic    /robot_description
  TF Prefix
  Links
Map                    ☐
PointCloud2            ✓
PointCloud2            ✓
  ✓ Status: Ok
  Topic                /octomap_ground
  Selectable           ✓
  Style                Flat Squares
  Size (m)             0.03
  Alpha                1
  Decay Time           0
  Position Transf...   XYZ
```

```
Type: Orbit (rviz_defa
Current View      Orbit (rviz)
  Near Clip ...   0.01
  Invert Z Axis   ☐
  Target Fra...   <Fixed Frame>
  Distance        3.58779
  Focal Shap...   0.05
  Focal Shap...   ✓
  Yaw             2.87358
  Pitch           0.370399
  Focal Point     -1.3288; 1.8211; ...
```

Figure 3.9: The depth image captured by the depth camera and displayed in Rviz2

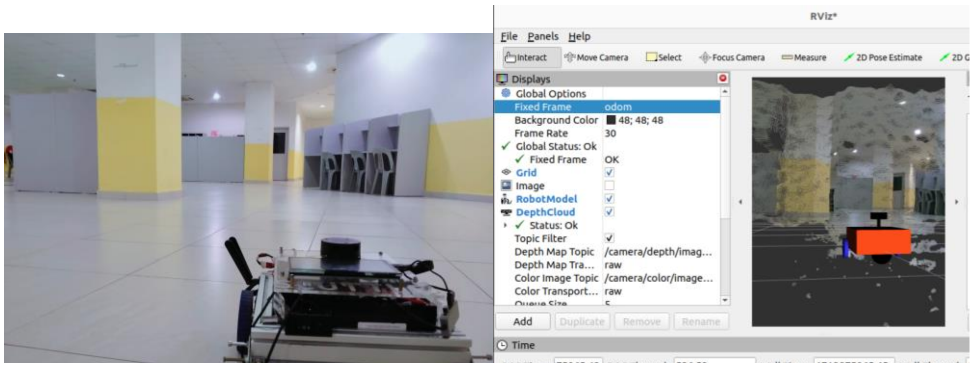
Before physical sensors such as LiDAR and depth camera can send environmental data over ROS2 network, it is important to install the ROS2 driver for each sensor so that the driver can convert the environmental data to the ROS2 message type and publish the data over ROS2 network. A ROS2 message is a single data structure. In ROS2, there are several standard data structures that are commonly used to publish the data obtained from sensors over ROS2 network, such as LaserScan type for LiDAR and PointCloud2 for depth camera. For the sensors that will be used in this project, RPLiDAR A1M8 and Realsense D455 have their own ROS2 driver and the user needs to install it before integrating the sensors' data to ROS2 network. The ROS2 driver of the RPLiDAR A1 M8 and Realsense D455 is constructed as different ROS2 packages. Table 3.1 listed out the drivers of the simulated and real devices that are used in this project. To use the node provided in the driver packages, two launch files have been prepared for RPLiDAR and Realsense D455 sensors respectively.

Table 3.1: The drivers for different hardware in simulation and real-world implementation

	Simulation	Physical robot
Differential drive controller	libgazebo_ros_diff_drive.so	serial, diff_drive_arduino
LiDAR	libgazebo_ros_laser.so	rplidar_ros (RPLiDAR A1 M8)
Depth camera	libgazebo_ros_camera.so	Intel® RealSense™ SDK 2.0, Intel® RealSense™ ROS2 wrapper (realsense D455 depth camera)

In the launch file, it is necessary for the user to specify the coordinate frame of the sensor so that the data of the sensor can be published on the correct coordinate frame. The RPLiDAR and Realsense D455 sensors may publish the data on a fixed coordinate frame name by default if there is no coordinate frame name given. Hence, in this project the coordinate frame of the RPLiDAR and Realsense D455 depth camera have been remapped to name of the coordinate frame stated in URDF sketched. Figure 3.10 shows the image captured by

Realsense D455 depth camera when the coordinate frame of the data published by Realsense 455 depth camera is remapped to the existing coordinate frames stated in URDF file. When the data is collected from sensors and is published to ROS2 network over ROS2 topic framework, regardless of the source of the data either it is from virtual world in gazebo or physical environment, if the message type of sensors' data is one of the standard message types of ROS2 message, the user can use the data to create a map. In this project, the RTAB-Map is used as a tool to organize the ROS2 messages published from the sensors and generate the map.



Global Options	
Fixed Frame	odom
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
Image	☐
RobotModel	✓
DepthCloud	✓
✓ Status: Ok	
Topic Filter	✓
Depth Map Topic	/camera/depth/imag...
Depth Map Tra...	raw
Color Image Topic	/camera/color/image...
Color Transport...	raw

Figure 3.10: The depth images captured by Realsense D455 depth camera

3.3.5 Setup RTAB-Map for 3D SLAM

The SLAM algorithm that will be used in this project is the Real-Time Appearance-Based Mapping (RTAB-Map) will be used in this project. RTAB-Map is an 3D SLAM library that implements loop closure detection using a memory management approach. It limits the size of the map so that loop closure detections are always processed within a fixed time limit to fulfill the requirements for long-term and large-scale environment mapping. (Labbé and Michaud, n.d.) RTAB-Map is a loop-closure solution based on memory management, hence it can be supplied with any odometry approach, including visual, lidar, or wheel odometry. (Labbé and Michaud, 2019) In this project, two different types of odometry source are fed into the RTAB-Map for mapping process, which are the odometry provided by wheel encoder only and the integration of the wheel encoder and LiDAR data as odometry. In this project, the comparison of the mapping method with different odometry sources is recorded in the results & discussion chapter.

RTAB-Map is a graph-based SLAM technique that has been incorporated into ROS as the rtabmap_ros package since 2013. In ROS2, rtabmap_ros has become a meta package and the nodes provided in the ROS1 rtabmap_ros package are moved to several sub packages. (Labbe, 2023) By using RTAB-Map, SLAM can be performed using any type of odometry that is acceptable for the application and robot as the odometry in RTAB-Map is an external input. The map is structured as a graph with nodes and links. A node in the map is generated by the Short-Term Memory (STM) that contains the odometry pose, sensor's raw data and other information that is useful for the modules such as Loop Closure and Proximity Detection. (Labbé and Michaud, 2019) Figure 3.11 shows the example of the 3D map generated by using RTAB-Map in the simulation with nodes (blue dot) and the links (the blue line that connects two nodes) in 3D map to indicate the odometry of the robot. Nodes are formed at a preset rate, "Rtabmap/DetectionRate" in milliseconds, based on the amount of data overlap between nodes. (Labbé and Michaud, 2019) A link represents a stiff transformation between two nodes. The STM adds neighbour links between consecutive nodes using the odometry recorded in a robot. Loop Closure and Proximity linkages are added using loop closure or proximity detection, respectively. (Labbé and Michaud, 2019) To reduce odometry drift, graph optimisation propagates computed errors to the whole graph after adding a new loop closure or proximity connection. OctoMap, Point Cloud, and 2D Occupancy Grid outputs can now be built and published to other modules after the graph has been optimized.

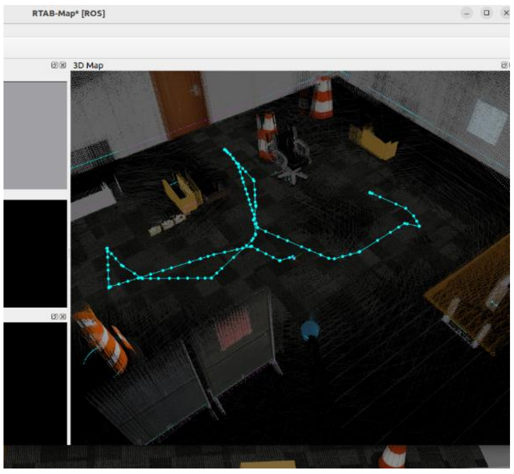


Figure 3.11: The 3D environment generated in the rtabmap_viz

The architecture of the memory management system in RTAB-Map is designed for large-scale and long-term operation. Loop closure detection is an important feature in SLAM for localizing a robot in each environment. The conventional method to perform global loop closure detection involves comparing a new location recorded by sensors with locations previously recorded in the map database. A new location is added to the database if no match is found. However, using this method, loop closure detection is impractical as the time required to process new observations increases with the number of locations on the map. If this processing time exceeds the acquisition time, a delay is introduced, resulting in an obsolete map. Furthermore, the time delay in loop closure detection worsens if the robot operates in large areas for extended periods. (Labbé and Michaud, n.d.) To mitigate delays in loop closure detection, a special memory management system is implemented in RTAB-Map.

A novel approach to enhancing loop closure detection in robotic navigation systems is used in RTAB-Map. Figure 3.12 shows the structure of the memory management in RTAB-Map. In RTAB-Map, the memory to store the data of the nodes and links have been separated into Working Memory (WM) and Long-Term Memory (LTM), prioritizing recent and frequently observed locations within WM. The primary aim is to achieve real-time loop closure detection in expansive environments by dynamically managing the number of locations utilized for detection. The key features of the approach include the selective transfer of less critical locations from WM to LTM when processing time exceeds real-time constraints. This transfer process is guided by the assessment of location importance, which is determined based on the frequency of occurrence. Locations are assigned weights accordingly, facilitating the prioritized transfer of locations within the memory system. Furthermore, the utilization of Short-Term Memory (STM) prevents redundant loop detection on recently visited locations, ensuring efficient operation. The overarching objective of this strategy is to strike a balance between real-time processing demands and the need for comprehensive mapping in long-term robotic operations. Hence, the robotic navigation systems can achieve enhanced efficiency and accuracy in loop closure detection, paving the way for more effective long-term operation in diverse environments.

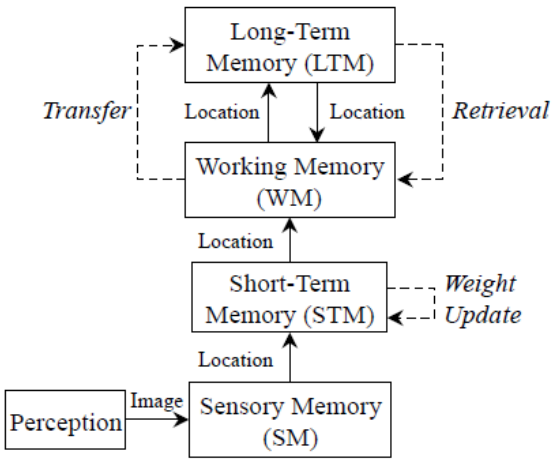


Figure 3.12: The structure of the memory management in RTAB-Map (Labbe and Michaud, 2013)

In RTAB-Map, several essential packages are utilized in this project, including rtabmap_slam, rtabmap_sync, and rtabmap_viz. Figure 3.13 illustrates its main ROS node, rtabmap, within the rtabmap_slam subpackage. When a loop closure is recognized, the map's graph is progressively created and optimized. The node's online output is the local graph, which contains the most recent data contributed to the map. The 3D map generated by RTAB-Map is stored in a .db file format, indicating the database that stores all sensor data and node information of the 3D map. By default, the RTAB-Map database is stored in "~/.ros/rtabmap.db". The rtabmap node can be set to mapping mode or localization mode according to the parameters provided in the YAML file.

Figure 3.14 illustrates the settings of the rtabmap node to initiate SLAM mode and localization mode, respectively.

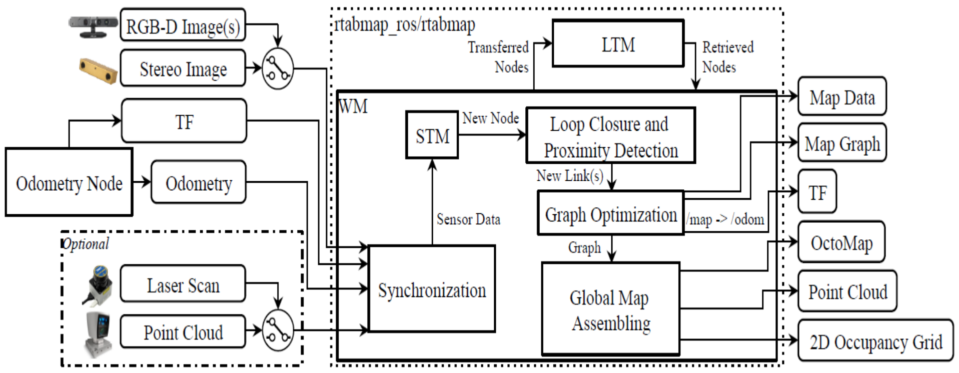


Figure 3.13: The structure of the rtabmap node in rtabmap_slam package (Labbé and Michaud, 2019)

```
# SLAM Mode:
Node(
        condition=UnlessCondition(localization),
        package='rtabmap_slam', executable='rtabmap', output='screen',
name='rtabmap_slam',
        parameters=[rtabmap_config],
        remappings=remappings,
        arguments=['-d']),

# Localization mode:
Node(
        condition=IfCondition(localization),
        package='rtabmap_slam', executable='rtabmap', output='screen',
        parameters=[rtabmap_config,
          {'Mem/IncrementalMemory':'False',
           'Mem/InitWMWithAllNodes':'True'}],
        remappings=remappings),
```

Figure 3.14: The settings of the rtabmap node in launch file

Figure 3.14 shows that the discrepancies between the SLAM and localization modes are due to two parameters: 'Mem/IncrementalMemory' and 'Mem/InitWMWithAllNodes'. By default, the rtabmap node operates in SLAM mode. To enable localization mode with RTAB-Map, set the 'Mem/IncrementalMemory' parameter to 'False' and 'Mem/InitWMWithAllNodes' to 'True'. The 'database_path' argument is added to the launch file to allow the user to specify which database to load into the rtabmap node during localization mode. In SLAM mode, the argument "--delete_db_on_start" or "-d" tells the system to delete the database before starting. Otherwise, the previous mapping session from the database is loaded. In localization mode, rtabmap employs multiple localization techniques: In localization mode, rtabmap utilizes several localization approaches: It assumes it is restarted in the map where it shut down previously. Global visual relocalization is performed using its loop closure detection approach (bag-of-words). Proximity detection (e.g., scan matching) can be used to refine its current position with the closest nodes in the map. Different localization modes provided by the rtabmap node can be used by setting appropriate parameters in the launch file.

The parameters of the rtabmap node used in this project are the default parameters stated in the launch file. By default, the rtabmap node subscribes to the /odom topic for the robot's odometry information. However, the differential drive controller used in this project publishes the robot odometry information over the /diff_cont/odom topic. Hence, it is important to remap the odometry topic that RTAB-Map needs to subscribe to in the RTAB-Map launch file. Additionally, the topic of the image provided by the depth camera also needs to be remapped, as the default topic that RTAB-Map subscribes to is not the same as the topic published by the depth camera. The remapped topics for the simulation in this project are presented in Figure 3.15.

```
remappings=[
        ('/odom', '/diff_cont/odom'),
        ('/rgb/image', '/depth_camera/image_raw'),
        ('/rgb/camera_info', '/depth_camera/camera_info'),
        ('/depth/image', '/depth_camera/depth/image_raw')]
```

Figure 3.15: The code to remap the topics that the rtabmap node subscribes to

The node provided by rtabmap_viz initializes the RTAB-Map visualization interface, serving as a wrapper for the RTAB-Map GUI library. While it shares the same purpose as rviz2, it offers additional settings that aid users in processing the RTAB-Map database. Users can open an RTAB-Map database in the rtabmap_viz node to visualize the 3D map generated by the rtabmap node. The interface of rtabmap_viz is depicted in Figure 3.11.

Synchronization of sensor data is critical for accurate registration and processing of the nodes and links in RTAB-Map database. ROS2 supports two types of synchronizations: accurate and approximate. (Labbé and Michaud, 2019) Exact synchronization necessitates matching timestamps for input subjects that correspond to data from the same sensor (e.g. stereo camera images). Approximate synchronisation compares timestamps and minimises delay faults, making it perfect for combining data from several sensors. However, synchronising issues with varying synchronisation needs, such as cameras and other sensors, might be difficult. The rtabmap_sync package makes this possible by combining camera topics into a single subject before processing with the rtabmap node. The use of the rtabmap_sync package can help to ensure consistent data alignment even when the timestamp changes. (Labbé and Michaud, 2019)

In the rtabmap_demos subpackage, there are three example launch files for starting the mapping process with RTAB-Map using different odometry sources. In this project, two launch files, turtlebot3_rgbd.launch.py and turtlebot3_rgbd_sync.launch.py, provided in the rtabmap_demos subpackage are modified to fit the requirements of 3D mapping. Another launch file, turtlebot3_scan.launch.py, is not considered in this project as it only receives 2D data from the 2D LiDAR and cannot be used to construct a 3D map. The difference between the two launch files provided in the rtabmap_demos subpackage is the odometry source used for the mapping process. In turtlebot3_rgbd.launch.py, the wheel encoder is the sole odometry source, whereas in turtlebot3_rgbd_sync.launch.py, the SLAM node is fed with the integration of wheel encoder and LiDAR data. The 3D map generated by different methods with be discussed in results & discussion section.

3.3.6 Setup Nav2 for robot navigation

In the obstacle avoidance system, the robot needs to first identify the obstacle and update the free space region that the robot can travel. After that, the robot can plan a new path by using the map with the updated free space region. The process to identify the obstacles and update free space region is achieved by using the cost map. While the process to replan a path is achieved by the local trajectory planner in Nav2. The application of the cost map and local trajectory planner in Nav2 will be explained in the sections below.

3.3.6.1 Cost map

There are two types of cost map in provided in Nav2: global cost map and local cost map. The global cost map is used for long-term path planning over the entire environment while the local cost map is used for local planning and obstacle avoidance. (Open Robotics, n.d.) Both the cost maps are layered cost map, which consists of several layer plugins such as static layer, obstacle layer and voxel layer. In this project, the configuration of the local cost map will be emphasized to develop the obstacle avoidance system and the global cost map will be created by using 2D LiDAR source only, which means that the global cost map does not have the ability to record the 3D obstacle and update it in global cost map. Hence, the global path initially planned based on the global cost map might traverse through 3D obstacles. A new path will be planned when obstacles are observed in the local cost map.

The nav2_costmap_2d package is an important package to create the cost map from the map generated by 3D SLAM algorithms such as RTAB-Map. To detect the 3D obstacles that cannot be detected using 2D LiDAR, there are two important plugins called obstacle_layer and voxel_layer plugin in 2D cost map to receive 3D environmental data and update the local cost map.

The obstacle layer organises data from the sensors such as LiDAR and depth cameras into a 2D grid. Each data point is interpreted as a ray extending from the sensor's position and traced over the grid via Bresenham's algorithm. Figure 3.16 illustrates the working principle of the obstacle layer. The ray's end point is marked as obstacle but its path is labelled as empty region due to direct visibility. Bresenham's ray-casting approach is used to outline the empty region between the sensor and the barrier in white, leaving the rest of the grid unexplored (the grey colour region in Figure 3.16). (Macenski, Moore, et al., 2023) While this layer is best suited for planar laser scanners due to its two-dimensional structure, it may be limited when working with 3D data.

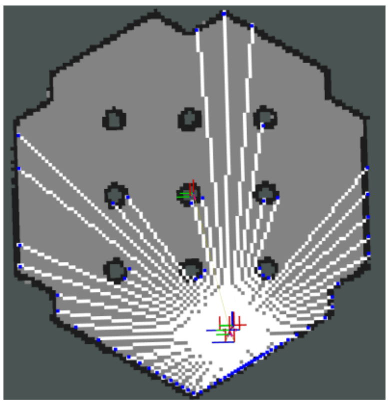


Figure 3.16: The example of Bresenham's ray-casting approach (Macenski, Moore, et al., 2023)

The voxel layer has the function that is similar to obstacle layer, but it functions in three dimensions and is ideal for tracking environments and taking ray-tracing data. It is especially useful for depth camera or 3D LiDAR that can provide the 3D environmental data, as well as sensor streams that are not strictly two-dimensional. As the height dimension of a voxel layer uses unsigned 32-bit integers, the maximum height in voxel layer is limited. The resolution of voxels in vertical direction is often coarser and differs from the resolution of voxels in horizontal direction. The 3D data in voxel layer is projected onto 2D plane and included into the primary cost map. Due to 3D raytracing in voxel layer, this layer requires more processing resources than the obstacle layer. (Macenski, Moore, et al., 2023) As the voxel layer is more suitable for the 3D obstacle detection, in this project the voxel layer will be used for the 3D obstacle detection system.

The voxel layer plugin is one of the built-in cost map plugins provided by the nav2_costmap_2d package. All plugins in nav2_costmap_2d must inherit from the basic class, nav2_costmap_2d::Layer, to be included in the layered cost map. This class defines virtual method APIs used to configure plugins, including onInitialize(), updateBounds(), updateCost(), and reset(). (Nav2, n.d.) The voxel layer overrides these methods to customize them for its specific use case. It receives PointCloud2 messages to iteratively calculate data and update the local cost map. It is crucial to set up the voxel layer correctly to detect 3D obstacles with specific heights. Parameters for the voxel layer and local trajectory planner (controller) are typically stored in a YAML file, which is then loaded into launch files for configuration. The nav2_bringup package provides example launch files for launching ROS2 actions and services related to robot navigation. The example YAML file provided by nav2_bringup package can be modified for different robot system and applications. (ROS Planning, n.d.) Figure 3.17 shows the example of the code in YAML file to enable the voxel layer in local cost map.

```
local_costmap:
  local_costmap:
    ros__parameters:
      update_frequency: 5.0
      publish_frequency: 2.0
      global_frame: odom
      robot_base_frame: base_link
      use_sim_time: True
      rolling_window:  true
      width: 3
      height: 3
      resolution: 0.05
      robot_radius: 0.22
      plugins: ["voxel_layer", "inflation_layer"]
```

Figure 3.17: The code to enable voxel layer in local cost map

3.3.6.2 Local trajectory planner

Navigation has two key components: global path planning and local trajectory planning. After the global path planner has determined a feasible course across the environment, the local trajectory planner takes over. Its function is to construct collision-free trajectories and send velocity commands to the robot's motor controller, guaranteeing smooth navigation while considering collision avoidance and system limits. Nav2's default local trajectory planner, DWB, provides flexibility and configuration via plugin-based critique functions and trajectory generators.

The Collision Monitor plays an important role in preventing collisions by monitoring zones around the robot with sensors such as point clouds or laser scans. It identifies breaches in these zones and takes appropriate actions, such as slowing or stopping the robot to avoid collisions. The Collision Monitor also has an Approach mode that allows the user to reduce the robot speed and keep a safe distance from potential impediments. Overall, these components work together to ensure that the robot can navigate safely and efficiently. Figure 3.18 illustrates the three primitive types of collision monitoring: stop, slowing, and approach polygons. In this project, the default controller, which is DWB controller will be used as the algorithm of the obstacle avoidance system.

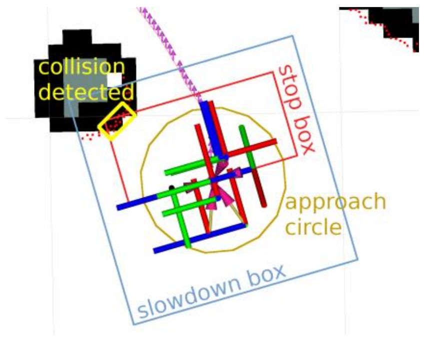


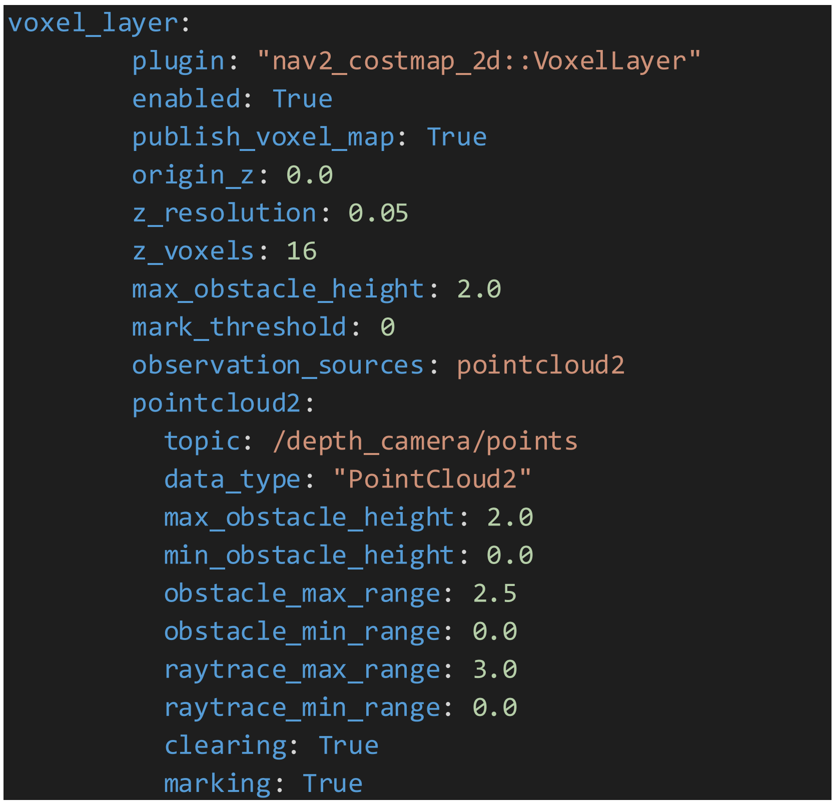
Figure 3.18: Different zones in the Collision Monitor

3.3.6.3 Nav2 implementation

A YAML file with the modified configuration of the local cost map and local trajectory planner is prepared for the obstacle avoidance system. The YAML file is modified from the default YAML file provided in nav2_bringup package. (ROS Planning, n.d.)

There are two main modifications made for this project. First, the parameters relate to Adaptive Monte Carlo Localization (AMCL) have been removed. This is because the localization of the robot navigation is achieved by setting the rtabmap node from rtabmap_slam subpackage to localization mode and the AMCL is no longer required for the robot localization.

Another modification of the YAML file is the voxel layer is enabled for local cost map. The parameters for the voxel layer are adjusted as shown in Figure 3.19. This ensures that 3D obstacles can be detected and projected onto the 2D local cost map.



```
voxel_layer:
        plugin: "nav2_costmap_2d::VoxelLayer"
        enabled: True
        publish_voxel_map: True
        origin_z: 0.0
        z_resolution: 0.05
        z_voxels: 16
        max_obstacle_height: 2.0
        mark_threshold: 0
        observation_sources: pointcloud2
        pointcloud2:
          topic: /depth_camera/points
          data_type: "PointCloud2"
          max_obstacle_height: 2.0
          min_obstacle_height: 0.0
          obstacle_max_range: 2.5
          obstacle_min_range: 0.0
          raytrace_max_range: 3.0
          raytrace_min_range: 0.0
          clearing: True
          marking: True
```

Figure 3.19: The parameters for voxel layer

In this project, the robot detects 3D obstacles using observation sources provided in the ROS2 PointCloud2 message type. The topic indicated in Figure 3.19 refers to the topic name from which users need to subscribe to obtain data published by the depth camera in PointCloud2 message type. In the simulation environment, the depth camera publishes 3D environmental information in

PointCloud2 message type over the /depth_camera/points topic. The voxel layer utilizes this 3D environmental information to update the local cost map when obstacles are detected using the depth camera.

The max_obstacle_height parameter specifies the maximum height of obstacles considered in the voxel layer and labelled as obstacles in the 2D cost map. This is a crucial parameter that allows the robot to navigate through tunnels or underneath tables safely. To ensure the robot's safety during operation, the max_obstacle_height is adjusted to the height of the robot with a certain clearance.

The modifications in the YAML can ensure the features provided by Nav2 libraries are functioning in the environment that is provided by the parameters in the YAML file.

3.4 Work Plan

The project is divided into two phases. The first phase will primarily involve research on the 3D cost map and path planning methods for obstacle avoidance. In the second phase, the 3D obstacle avoidance algorithm will be tested on an actual robot. The detailed work plan for these two phases is outlined below:

First Phase:

- Conduct research on the theory behind the cost map and local trajectory planner.
- Install the operating system, ROS2 and Nav2.
- Familiarize oneself with the Command Line Interface (CLI) in Linux.
- Gain a solid understanding of the basic concepts of ROS2 and how to use it.
- Learn the fundamental concepts of Nav2.
- Fabricate a differential drive robot for hardware testing.

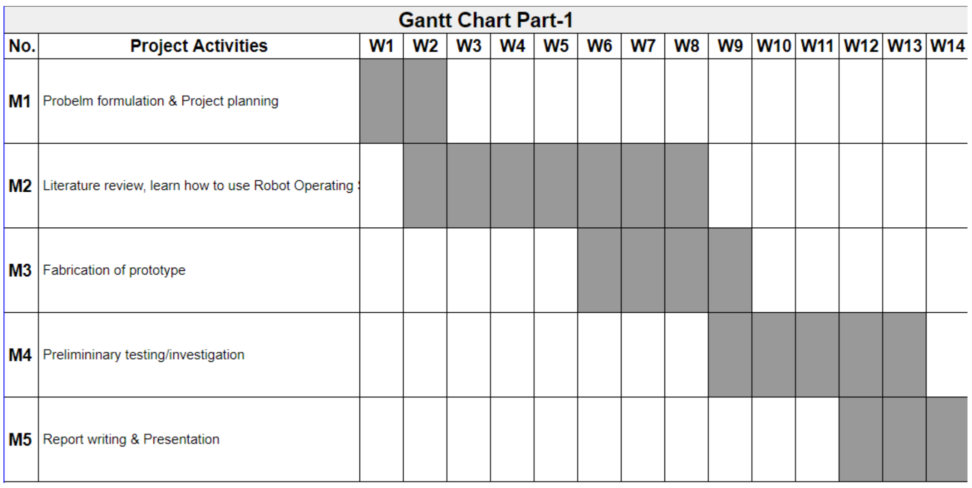
Second Phase:

- Develop the algorithm of the obstacle avoidance system for 3D robot navigation.
- Implement and apply the algorithm to the robot.
- Conduct thorough testing of the algorithm on the robot.
- Document and record the results of the tests.

These phases will encompass the research, software and hardware setup, learning, algorithm development and testing required for the successful implementation of the obstacle avoidance system in 3D environment.

3.5 Gantt Chart

3.5.1 First Phase

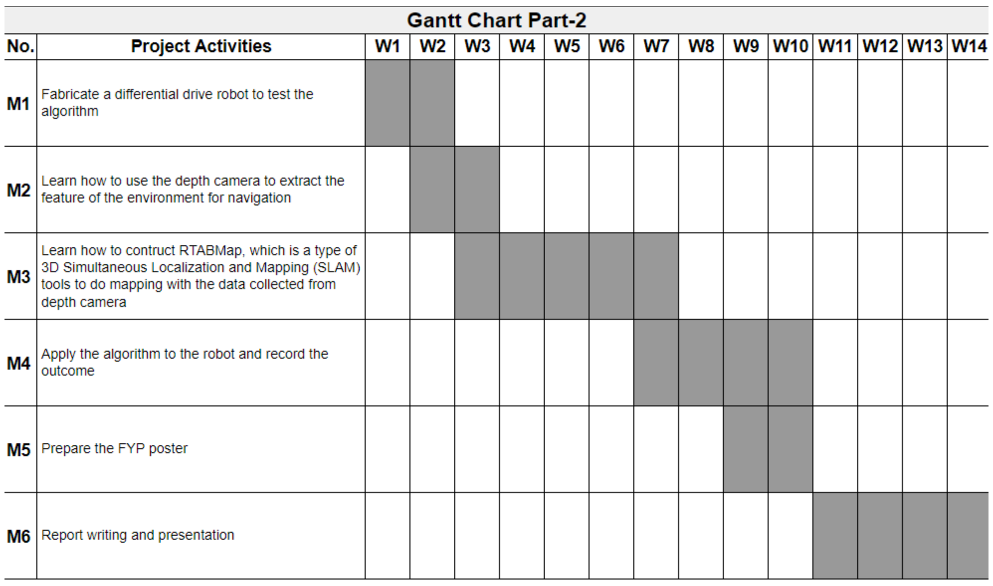


Gantt Chart Part-1

No.	Project Activities	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14
M1	Probelm formulation & Project planning	▨	▨												
M2	Literature review, learn how to use Robot Operating		▨	▨	▨	▨	▨	▨							
M3	Fabrication of prototype						▨	▨	▨	▨					
M4	Prelimininary testing/investigation									▨	▨	▨	▨	▨	
M5	Report writing & Presentation												▨	▨	▨

Figure 3.20: Gantt chart for first phase

3.5.2 Second Phase



Gantt Chart Part-2

No.	Project Activities	W1	W2	W3	W4	W5	W6	W7	W8	W9	W10	W11	W12	W13	W14
M1	Fabricate a differential drive robot to test the algorithm	▨	▨												
M2	Learn how to use the depth camera to extract the feature of the environment for navigation		▨	▨											
M3	Learn how to contruct RTABMap, which is a type of 3D Simultaneous Localization and Mapping (SLAM) tools to do mapping with the data collected from depth camera			▨	▨	▨	▨	▨							
M4	Apply the algorithm to the robot and record the outcome							▨	▨	▨	▨				
M5	Prepare the FYP poster									▨	▨				
M6	Report writing and presentation											▨	▨	▨	▨

Figure 3.21: Gantt chart for second phase

3.6 Summary

In this software-based project, thorough preparations are essential for successfully implementing the obstacle avoidance system for 3D robot navigation. Prior to the implementation of the obstacle avoidance algorithm, several prerequisites for robot navigation must be fulfilled. Firstly, a differential drive robot is constructed for this project. Subsequently, the URDF of the differential robot is created and ROS2 wrapper packages are installed for the sensors to enabling them to publish data on the ROS2 network. Additionally, RTAB-Map is installed to facilitate 3D SLAM and localization of the robot. Finally, Nav2 local trajectory planner is introduced to utilize the map generated by RTAB-Map and produce a cost map for the obstacle avoidance system.

CHAPTER 4

RESULTS AND DISCUSSIONS

4.1 Introduction

There are two main parts in this section: results and discussion. The findings of this project will be presented in the results section while the evaluation of the findings will be presented in the discussion section.

4.2 Results

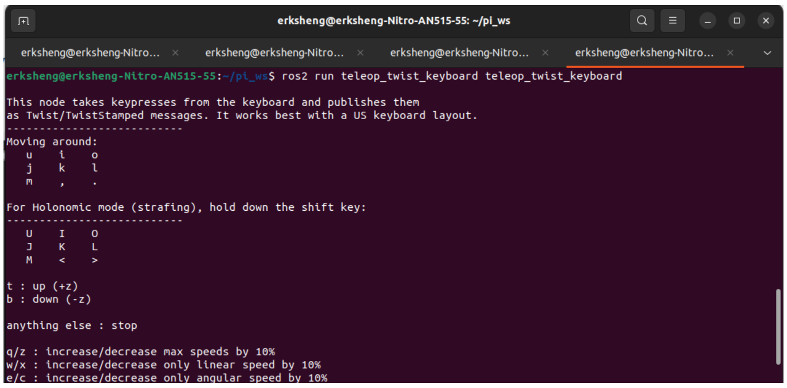
The results for this project are separated into two parts: mapping and navigation. In the mapping section, the 3D map generated by the RTAB-Map SLAM will be analyzed to determine the quality suitable for robot navigation. In the navigation section, the performance of the obstacle avoidance system will be presented. The mapping and navigation results will be provided for both simulation and real-world applications to assess the feasibility of the mapping and navigation systems.

4.2.1 Mapping

Mapping is a prerequisite for initiating the robot navigation process. In this project, the mapping results from both simulation and real-world applications have been recorded in this chapter. Two odometry sources have been fed to the RTAB-Map SLAM tool to generate the 3D map. The quality of the generated 3D map will be analyzed and the map with the best quality will be selected for the robot navigation process. The criteria for evaluating the quality of the generated 3D map include the accuracy of obstacle positioning and the similarity between the real environment and the generated map. The simulation results and actual results will be presented in the following sections.

4.2.1.1 Simulation result

Before starting the mapping process, it is crucial to follow the following steps to avoid conflicts in the ROS2 environment. First, the robot_state_publisher node should be launched to ensure proper setup of coordinate frames in the robot and the drivers for the simulation sensors. After that, Gazebo is launched, where the robot will be spawned in the prepared simulation world. Once Gazebo is ready, the RTAB-Map node is launched to initiate the mapping process. Figure 4.2 shows the environment in which the robot will carry out the navigation process in the simulation. The launch files to start the node for the mapping process are provided in the RTAB-Map meta package and can be obtained from the rtabmap_demos subpackage. After that, the node in the teleop_twist_keyboard package is launched to manually control the robot using the keyboard to navigate around the environment for mapping purposes. Figure 4.1 illustrates the command to launch the node in the teleop_twist_keyboard package in the CLI. During this process, the odometry data that represents the distance travelled by the robot and sensor data are recorded in the RTAB-Map database to construct a 3D map. It is important to ensure that the robot's travel speed is slow enough to prevent the loss of environmental information and inaccurate odometry during the mapping process. In this project, the robot's speed is set to 0.2 m/s throughout the mapping process.


erksheng@erksheng-Nitro-AN515-55: ~/pi_ws
erksheng@erksheng-Nitro...
erksheng@erksheng-Nitro...
erksheng@erksheng-Nitro...
erksheng@erksheng-Nitro...

```
erksheng@erksheng-Nitro-AN515-55:~/pi_ws$ ros2 run teleop_twist_keyboard teleop_twist_keyboard

This node takes keypresses from the keyboard and publishes them
as Twist/TwistStamped messages. It works best with a US keyboard layout.
---------------------------
Moving around:
   u    i    o
   j    k    l
   m    ,    .

For Holonomic mode (strafing), hold down the shift key:
---------------------------
   U    I    O
   J    K    L
   M    <    >

t : up (+z)
b : down (-z)

anything else : stop

q/z : increase/decrease max speeds by 10%
w/x : increase/decrease only linear speed by 10%
e/c : increase/decrease only angular speed by 10%
```

Figure 4.1: The launching process of the teleop_twist_keyboard package

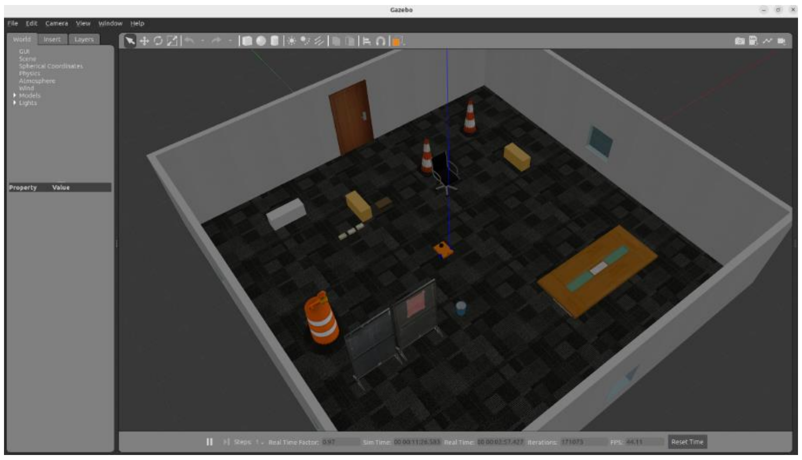


Figure 4.2: The world prepared in simulation

The resultant 3D map generated with only the wheel encoder as the odometry source for the mapping process is shown in Figure 4.3. From Figure 4.3, it can be observed that when the wheel encoder is the sole odometry source for the mapping process, the generated map exhibits very poor quality. The obstacles are misaligned with the environment constructed in the Gazebo simulation. Additionally, there is a noticeable ghosting effect, where the same object appears multiple times around specific positions. The 3D map fails to meet the evaluation criteria, which include the accuracy of obstacle positioning and the similarity between the real environment and the generated map. Consequently, the wheel encoder as the sole odometry source is deemed unsuitable for the 3D mapping process.

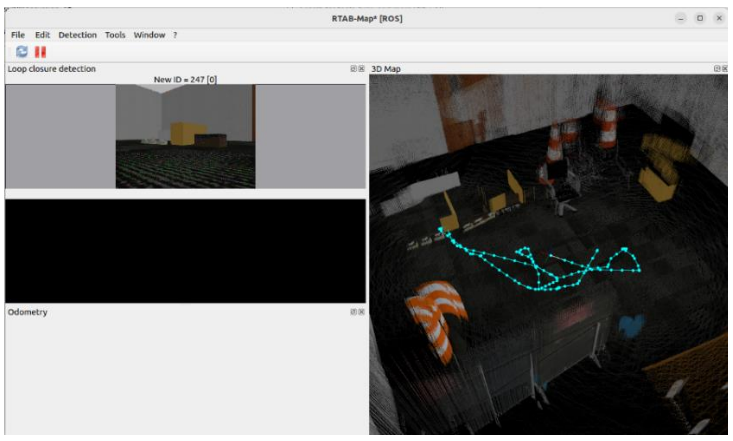


Figure 4.3: The 3D map presented in rtabmap_rviz with the wheel encoder as the only odometry source

The resultant 3D map generated with the integration of the wheel encoder and LiDAR data as odometry sources is shown in Figure 4.4. From Figure 4.4, it can be observed that the quality of the generated 3D map has significantly improved compared to the map generated using only the wheel encoder as the odometry source. Although some ghosting issues persist in the map, the overall quality of the 3D map is acceptable. Besides, the 3D map has higher accuracy in obstacle positioning and greater similarity to the given environment in the Gazebo world. Consequently, the integration of the wheel encoder and LiDAR data will be utilized as the odometry source for the mapping process to generate a 3D map with better quality.

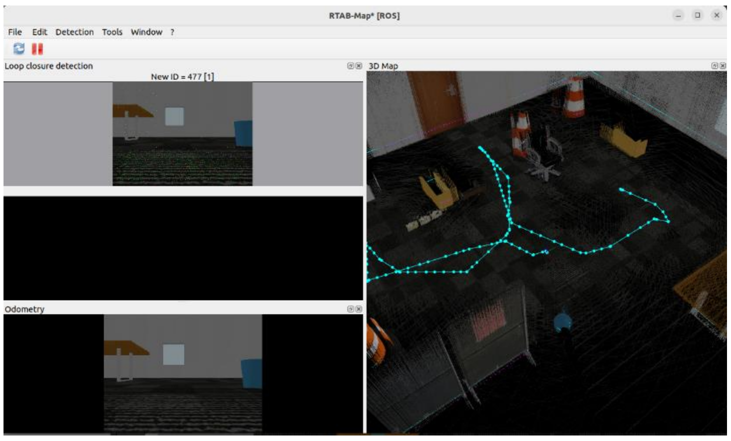


Figure 4.4: The 3D map presented in rtabmap_rviz with the integration of the wheel encoder and LiDAR data as odometry source

4.2.1.2 Real world result

The steps to initiate the mapping process for a real robot are almost identical to those in the simulation. The user must first launch the robot_state_publisher node for the transform function on the robot. After that, rtabmap_sync_node.launch.py is launched to set up the nodes provided in rtabmap_ros metapackage. Finally, the teleop_twist_keyboard node is started to enable manual control of the robot using the keyboard. However, there are some differences in the setup process between the simulation and real-world implementation for the mapping process. For instance, in real-world applications, the user does not need to start the Gazebo simulation application. Additionally, unlike in the simulation, the drivers for the RPLiDAR and

Realsense D455 depth camera have not been integrated into the ros2_control package. Therefore, for real-world applications, the user needs to manually launch the drivers for the sensors one by one in the command-line interface (CLI). Additionally, the parameter ‘use_sim_time’ in the launch file must be set to 'false' since the robot is now navigating in the real-world application. The environment in which the robot will navigate is depicted in Figure 4.5.

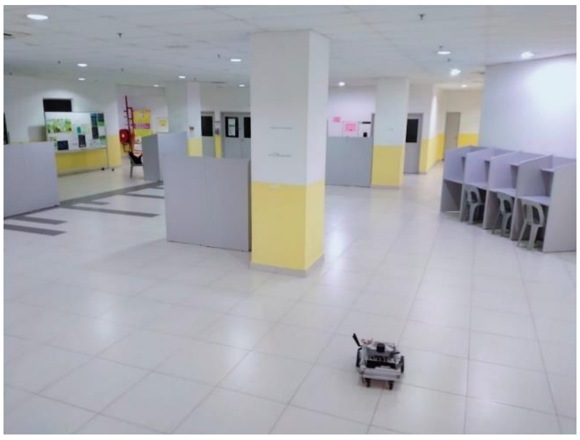


Figure 4.5: The real-world environment for the robot's navigation

The 3D map generated by the RTAB-Map tool is shown in Figure 4.6. It can be observed that the quality of the 3D map generated in the real-world application is sufficient for robot navigation. This is evidenced by the higher accuracy in obstacle positioning and greater similarity to the real-world environment. The robot's speed is set to 0.2 m/s, which is the same as in the simulation to ensure the production of a high-quality 3D map.

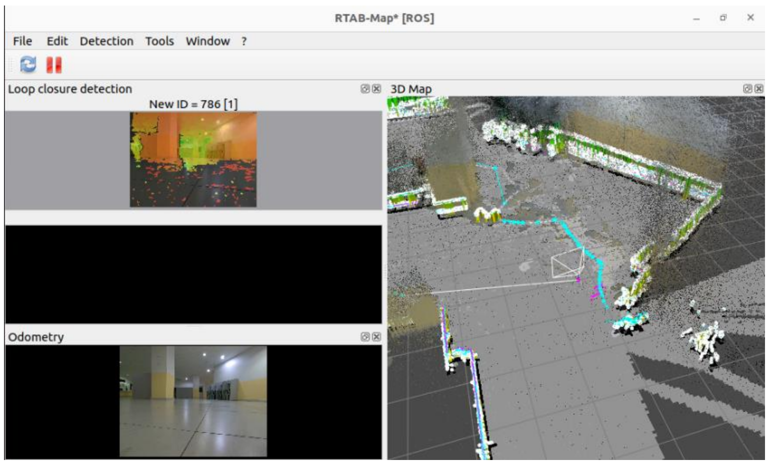


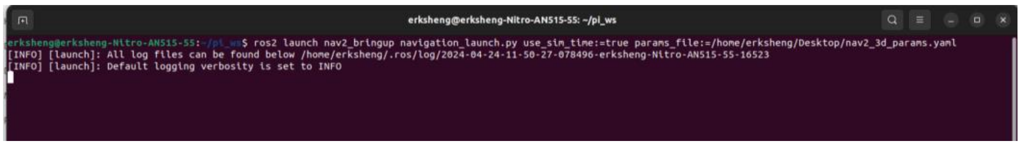
Figure 4.6: The 3D map generated in the real-world environment

4.2.2 Navigation

After generating the 3D map using the RTAB-Map tool, the robot can start the navigation process based on the map recorded in the database. The results of robot navigation will be divided into two parts: simulation and real-world results. The navigation results will primarily focus on detecting obstacles that cannot be detected by the 2D LiDAR. When the Nav2 library detects obstacles, it will update the cost map accordingly and, if necessary, replan the path to avoid the obstacles.

4.2.2.1 Simulation result

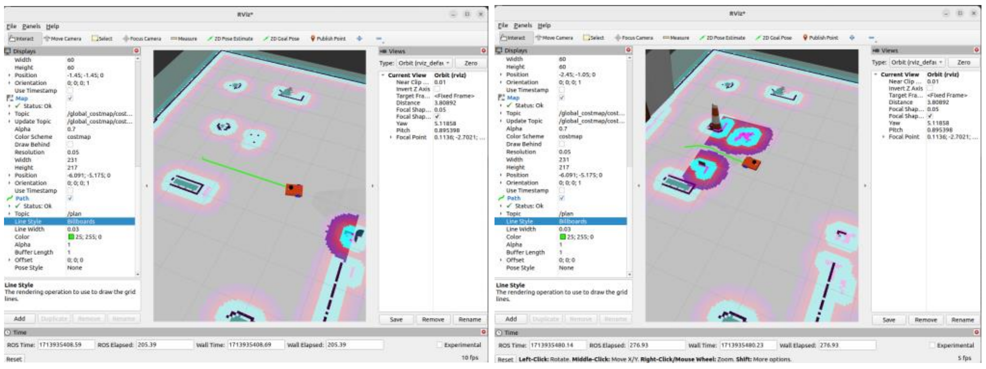
Once the YAML file for the Nav2 launch file, namely navigation.launch file, is prepared, the Nav2 node can be launched in the command-line interface (CLI) using the command shown in Figure 4.7. The 'use_sim_time' parameter is set to true as the robot navigates in the simulation world. The params_file parameter specifies the path to access the YAML file, which depends on the user's chosen location. As long as the path to the YAML file is correct, users can place the YAML file in any location.



```
erksheng@erksheng-Nitro-AN515-55:~/pi_ws$ ros2 launch nav2_bringup navigation_launch.py use_sim_time:=true params_file:=/home/erksheng/Desktop/nav2_3d_params.yaml
[INFO] [launch]: All log files can be found below /home/erksheng/.ros/log/2024-04-24-11-50-27-078496-erksheng-Nitro-AN515-55-16523
[INFO] [launch]: Default logging verbosity is set to INFO
```

Figure 4.7: The command to launch navigation.launch with a given YAML parameter file

The preparation steps before starting robot navigation are same as the steps of the mapping process. Initially, the robot_state_publisher node must be launched to properly set up the coordinate frames in the robot and configure the sensor drivers in the simulation environment. After that, Gazebo is launched to spawn the robot in the prepared simulation world. Once Gazebo is ready, the RTAB-Map node is launched to localize the robot within a map and publish the 2D occupancy grid map over the /map topic in the ROS2 network. The main difference between mapping and navigation lies in configuring the rtabmap_slam node, which needs to be set to localization mode during navigation to utilize the 3D map recorded in the database. Once all configurations are complete, the navigation.launch.py file in nav2_bringup is launched with the parameters recorded in the YAML file. Robot navigation can start once navigation.launch.py is successfully launched. Figure 4.8 illustrates the path before and after obstacles, undetectable by LiDAR, are identified by the depth camera. The green line represents the path planned by using Nav2.



(a) (b)

Figure 4.8: The path of the robot before (a) and after (b) the obstacles is detected by depth camera

The data published by the 2D LiDAR, represented as LaserScan, is the sole observation source in the global cost map. Consequently, obstacles located lower than the position of the 2D LiDAR cannot be directly detected, and thus are not depicted in the global cost map. As a result, the nav2 planner calculates the shortest path, typically a straight line from the starting point to the destination based on the global cost map. When the robot approaches an obstacle undetected by the 2D LiDAR, the local cost map with the voxel layer enabled, can detect the obstacle and projects it onto the 2D cost map. Subsequently, the robot's path is replanned using the controller plugin in nav2 to navigate around the obstacle. Additionally, the robot's navigation underneath a table was tested in the simulation. Figure 4.9 illustrates the robot's operation as it travels beneath an office table. The robot successfully plans its path underneath the table if the available height and space allow for passage to ensure the shortest travel distance.

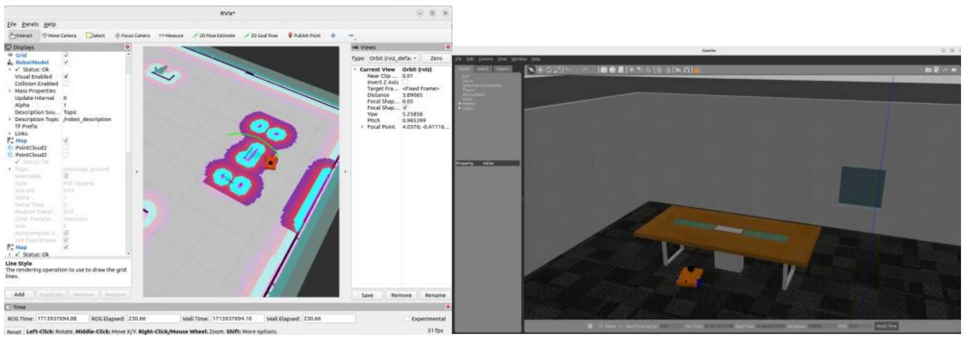
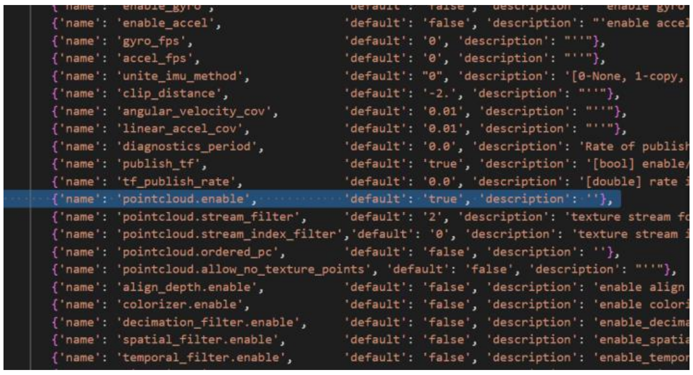

(a) (b)

Figure 4.9: The path planned (a) underneath an office table (b)

4.2.2.2 Real world result

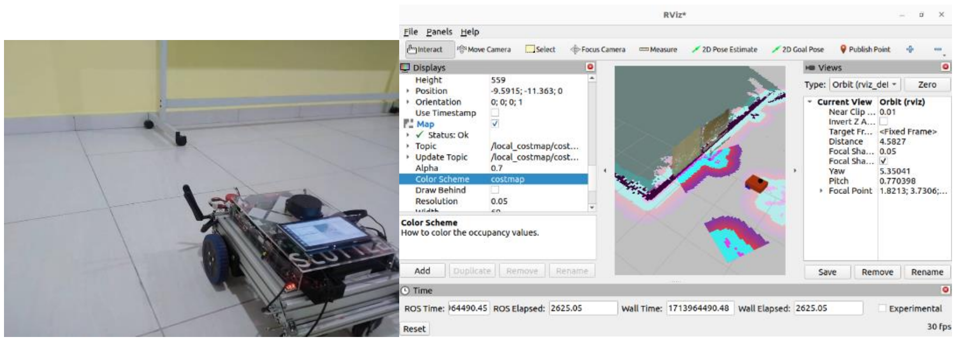
The method of starting navigation for a real robot is very similar to the simulation. Users start by launching the robot_state_publisher node to enable robot transforms, then rtabmap_sync_node.launch.py to enable RTAB-Map localization, and finally the Nav2 server. However, there are variations in setting up navigation for real-world navigation compared to simulations. In real-world circumstances, users do not run the Gazebo simulation application. Furthermore, users must run drivers for RPLiDAR, Realsense D455 depth camera, and microcontroller separately via the CLI as the drivers are not integrated to ros2_control package. Numerous parameters in the Nav2 package's YAML file require adjustment for the real-world navigation. Specifically, the use_sim_time parameter is set to false to simulate real-world navigation. Additionally, the pointcloud.enable parameter in the launch file given by the Realsense ROS2 Wrapper (IntelRealSense, n.d.) is set to true (the default value is false) to allow the Realsense D455 depth camera broadcasts 3D environmental data in PointCloud2 message format over the ROS2 network. The 3D environmental data will be fed to voxel layer and the local cost map will be updated when obstacles are detected using the depth camera. Figure 4.10 displays a portion of the parameters from the example launch file provided by the Realsense ROS2 Wrapper package.



```
{'name': 'enable_gyro',                 'default': 'false', 'description': "'enable gyro
{'name': 'enable_accel',                'default': 'false', 'description': "'enable accel
{'name': 'gyro_fps',                    'default': '0', 'description': "''"},
{'name': 'accel_fps',                   'default': '0', 'description': "''"},
{'name': 'unite_imu_method',            'default': "0", 'description': '[0-None, 1-copy,
{'name': 'clip_distance',               'default': '-2.', 'description': "''"},
{'name': 'angular_velocity_cov',        'default': '0.01', 'description': "''"},
{'name': 'linear_accel_cov',            'default': '0.01', 'description': "''"},
{'name': 'diagnostics_period',          'default': '0.0', 'description': 'Rate of publish
{'name': 'publish_tf',                  'default': 'true', 'description': '[bool] enable/
{'name': 'tf_publish_rate',             'default': '0.0', 'description': '[double] rate i
{'name': 'pointcloud.enable',           'default': 'true', 'description': ''},
{'name': 'pointcloud.stream_filter',    'default': '2', 'description': 'texture stream fo
{'name': 'pointcloud.stream_index_filter','default': '0', 'description': 'texture stream i
{'name': 'pointcloud.ordered_pc',       'default': 'false', 'description': ''},
{'name': 'pointcloud.allow_no_texture_points', 'default': 'false', 'description': "''"},
{'name': 'align_depth.enable',          'default': 'false', 'description': 'enable align
{'name': 'colorizer.enable',            'default': 'false', 'description': 'enable colori
{'name': 'decimation_filter.enable',    'default': 'false', 'description': 'enable_decima
{'name': 'spatial_filter.enable',       'default': 'false', 'description': 'enable_spatia
{'name': 'temporal_filter.enable',      'default': 'false', 'description': 'enable_tempor
```

Figure 4.10: The parameters to launch the ROS2 driver for D455 depth camera (IntelRealSense, n.d.)

When the robot is properly set up by following all the procedures as stated, the navigation of the real robot can begin. Figure 4.11 (a) shows the support leg of the notice board, which cannot be detected by the 2D LiDAR, is captured by the depth camera and updated in the local cost map.

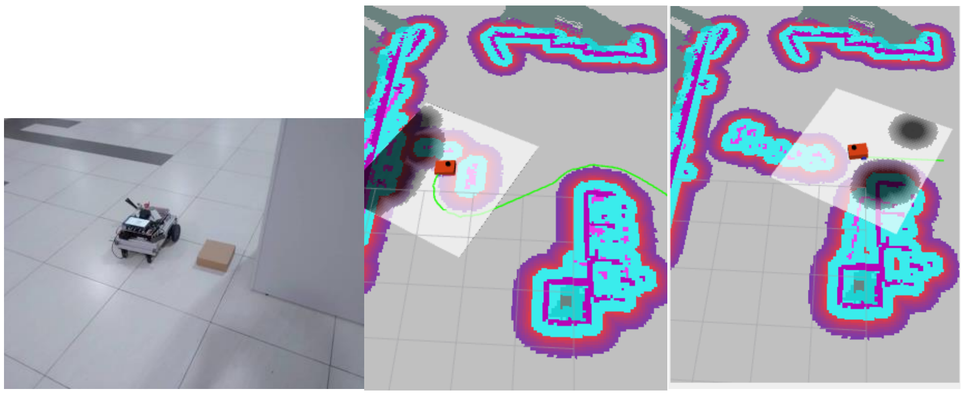


```
Height            559
Position          -9.5915; -11.363; 0
Orientation       0; 0; 0; 1
Use Timestamp     ☐
Map               ✓
Status: Ok
Topic             /local_costmap/cost...
Update Topic      /local_costmap/cost...
Alpha             0.7
Color Scheme      costmap
Draw Behind       ☐
Resolution        0.05

Color Scheme
How to color the occupancy values.

Type: Orbit (rviz_def    Zero
Current View      Orbit (rviz)
Near Clip ...     0.01
Invert Z A...     ☐
Target Fr...      <Fixed Frame>
Distance          4.5827
Focal Sha...      0.05
Focal Sha...      ✓
Yaw               5.35041
Pitch             0.770398
Focal Point       1.8213; 3.7306;...

Time
ROS Time: 64490.45  ROS Elapsed: 2625.05  Wall Time: 1713964490.48  Wall Elapsed: 2625.05  ☐ Experimental
```

(a) (b)

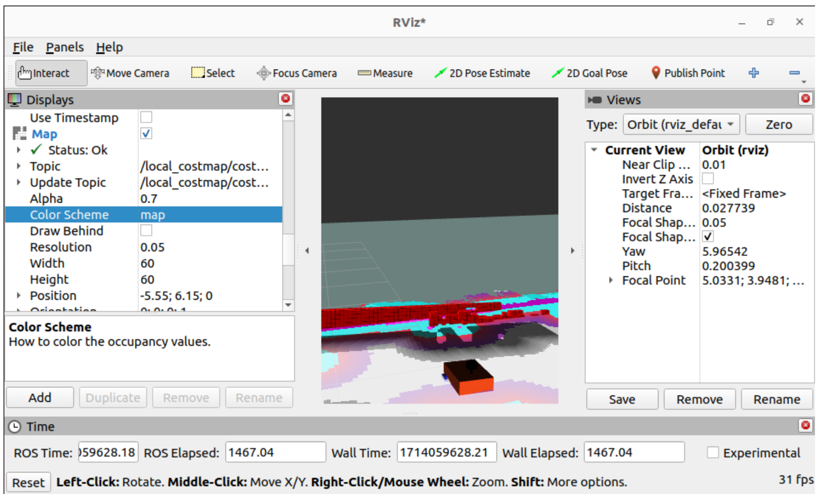
Figure 4.11: The real-world scene (a) and the environment displayed in Rviz2 (b)

Figure 4.12 shows the real-world results of the obstacle avoidance algorithm. Figure 4.12 (a) shows the cardboard obstacle that has been placed on the floor as the obstacle that the 2D LiDAR cannot detect. In the global cost map, the obstacle is not detected as the global cost map is constructed by the 2D LiDAR source only. Hence, the global path planning algorithm plans the path on the obstacle for the shortest distance, as shown in Figure 4.12 (b). When the robot approaches the obstacle, it is captured by the depth camera and updated in the local cost map, as shown in Figure 4.12 (c). The obstacle in the global cost map is represented in colour, while in the local cost map, it is depicted in black and white. It is proven that the algorithm developed for this project can detect obstacles that cannot be detected by 2D LiDAR and replan the path to avoid the obstacle. The obstacle behind the robot represents the examiner who needs to follow the robot and monitor the navigation process of the robot in Rviz2 from the screen installed on the robot. Figure 4.13 shows the visualization of the voxel layer used in the project. The voxels shown in Figure 4.13 are up to the maximum obstacle height set in the YAML file. Hence, the robot can go beneath the obstacle that has the enough clearance.



(a) (b) (c)

Figure 4.12: The real-world navigation results



Displays

```
Use Timestamp   ☐
Map             ☑
  Status: Ok
  Topic         /local_costmap/cost...
  Update Topic  /local_costmap/cost...
  Alpha         0.7
  Color Scheme  map
  Draw Behind   ☐
  Resolution    0.05
  Width         60
  Height        60
  Position      -5.55; 6.15; 0
```

Color Scheme
How to color the occupancy values.

Views

```
Type: Orbit (rviz_defau
Current View      Orbit (rviz)
  Near Clip ...   0.01
  Invert Z Axis   ☐
  Target Fra...   <Fixed Frame>
  Distance        0.027739
  Focal Shap...   0.05
  Focal Shap...   ☑
  Yaw             5.96542
  Pitch           0.200399
  Focal Point     5.0331; 3.9481; ...
```

Time

```
ROS Time: )59628.18  ROS Elapsed: 1467.04  Wall Time: 1714059628.21  Wall Elapsed: 1467.04  ☐ Experimental
```

Left-Click: Rotate. **Middle-Click:** Move X/Y. **Right-Click/Mouse Wheel:** Zoom. **Shift:** More options. 31 fps

Figure 4.13: The visualization of voxel layer of the local cost map in Rviz2

4.3 Discussion

In this section, the performance of the mapping process and the navigation process will be discussed. In the mapping process, the possible factors that might lead to the bad map quality has been discussed. In the navigation process, the performance of the obstacle avoidance system designed for this project will be discussed.

4.3.1 Mapping

From the results, it can be found that the quality of the generated 3D map with the integration of wheel odometry and LiDAR as odometry source has is better than the map generated using only the wheel encoder as the odometry source. There are several possible reasons that might cause the difference in two mapping process. First, the odometry obtained from the wheel encoder will drift over the distance travelled. Hence, when the robot travelled around the map, the error in the odometry provided by wheel encoder will accumulate and causing the drift of the map, which will affect the quality of the map. Besides, the update rate of the rtabmap node in SLAM mode is a possible factor that lead to the ghosting issue in the SLAM process. The RTAB-Map database consists of nodes and links with the 3D environmental data. When the robot travels too fast, the successive node may not have enough time to overlap the current node, which cause the incorrect node information and finally lead to ghosting issue in 3D mapping. (Labbé and Michaud, 2019)

Labbé and Michaud (2018) stated that the data of a LiDAR is used to refine the odometry of the robot by the Proximity Detection module in the SLAM process. When the robot revisits areas in the opposite direction or in environments lacking visual features within the depth range of the RGB-D camera, the appearance-based loop closure cannot be detected. This limitation can hinder path planning capabilities and navigation as the loop closure cannot be detected. The limitation of the appearance-based loop closure detection in rtabmap node is the main reason that cause the ghosting issue in the 3D map as the loop closure cannot be detected when the robot revisits the world in the opposite direction. Hence, the Proximity Detection module is used to overcome the limitation of appearance-based loop closure detection in RTAB-Map. To overcome this issue, the Proximity Detection module uses LiDAR to compensate for odometry drift in places where the camera cannot identify loop closures. To fix the map, proximity linkages are created by aligning laser scans in the reverse direction. (Labbé and Michaud, 2018)

Due to the limitations of the appearance-based loop closure detection in RTAB-Map SLAM, it is necessary for the robot to be equipped with a LiDAR to provide the laser rangefinder data to Proximity Detection module and improve the 3D map quality.

4.3.2 Navigation

From the results, it can be found that the robot navigation by using the default controller in Nav2, which is DWB, has successfully achieved the function to avoid the 3D obstacles. The DWB (Dynamic Window Approach) trajectory planner is now the default local trajectory planner in Navigator 2. The DWB architecture is a development of prior Navigation Stack controllers, providing more flexibility through plugin-based critic functions and trajectory generators. Despite its configurable nature, DWB requires challenging parameter tuning to obtain optimal performance, which can be complex and time-consuming. Poorly calibrated setups can result in inferior robot behaviours, prompting criticism from the ROS mobile robotics community. The maintainers intend to replace DWB with MPPI (Model Predictive Path Integral Control) once it has reached a sufficient degree of maturity, citing its potential for enhanced performance and reliability. (Macenski, Moore, et al., 2023)

MPPI generates a large number of modified trajectory samples based on the prior optimal trajectory, then scores each sample to determine the optimal trajectory. This method eliminates the necessity for non-linear optimisation, giving designers more freedom in designing system behaviour because cost functions are not required to be differentiable or convex. MPPI iteratively refines the trajectory between time steps as the robot travels towards its goal, demonstrating effectiveness in a variety of contexts, including aggressive outdoor driving and commercial robotics. MPPI outperforms reactive approaches by reacting intelligently to dynamic impediments even when not explicitly modelled. It rarely requires active recovery behaviours, because to its predictive back-out manoeuvres, which are especially useful in high-traffic or limited situations to keep the robot from being trapped. (Macenski, Moore, et al., 2023) Hence, the MPPI controller needs to be studied in the future to improve the performance of the controller for the obstacle avoidance system.

4.4 Summary

In this section, the results of the obstacle avoidance algorithm in the simulation and real-world navigation have been presented. It can be found that the quality of the 3D map generated by the RTAB-Map SLAM with the integration of wheel encoder and LiDAR as odometry source is better than the quality of the 3D map generated by the RTAB-Map SLAM with the wheel encoder as the only odometry source. This is because the application of the LiDAR in RTAB-Map 3D SLAM can be a source to refine the odometry of the robot by using the Proximity Detection module. Besides, the simulation and real-world navigation results have proved that the obstacle avoidance system developed is feasible and applicable in real-world usage.

CHAPTER 5

CONCLUSIONS & RECOMMENDATIONS

5.1 Conclusion

In conclusion, the objectives of this project have been successfully achieved, marking significant progress in the development of an effective obstacle avoidance system for 3D robot navigation. Throughout the project, RTAB-Map played a central role in constructing a high-quality 3D map of the environment. A crucial step in achieving this was the incorporation of LiDAR data for odometry refinement within the RTAB-Map SLAM process, ensuring accurate mapping and localization. Additionally, the integration of depth camera data into the voxel layer of the local cost map enabled the detection of 3D obstacles that were not identifiable by the 2D LiDAR alone, further enhancing the system's obstacle avoidance capabilities.

Moreover, the selection of the DWB local trajectory planner as the default controller proved to be effective in meeting the project's requirements. It is also important to emphasize the significance of maintaining a robust transformation tree (tf) throughout the system. A well-configured tf infrastructure ensures accurate spatial relationships between various components to enable seamless coordination and navigation of the robot within its environment.

5.2 Recommendations for future work

Due to the nature of the voxel layer in the local cost map, the obstacle avoidance system developed in this project will consider the ramp as an obstacle. Hence, in the future, the obstacle avoidance system needs to be improved so that the algorithm does not consider the ramp as an obstacle. This improvement is important because if the robot does not consider the ramp as an obstacle, it can plan a path on it so that the robot can navigate on the path.

Possible solutions for improving the obstacle avoidance system in the future include using the waypoint following plugin provided in Nav2 and creating a new cost map plugin for the local cost map. RTAB-Map can be built with OctoMap. In OctoMap, there is a filter called octomap_ground which will produce a point cloud of the ground of the OctoMap (Labbe, n.d.). Hence, the point cloud of the ground can be used to create a new cost map plugin by defining the points in the point cloud of octomap_ground as an empty region that the robot can travel. However, creating the new cost map plugin requires understanding of Object-Oriented Programming (OOP) language and extensive testing to verify the feasibility of this method to detect a ramp.

REFERENCES

Foote, T., *tf: The Transform Library*,

Hornung, A. et al., 2013. OctoMap: An efficient probabilistic 3D mapping framework based on octrees. *Autonomous Robots*, 34(3), pp.189–206.

IntelRealSense (no date) *Realsense-ros/realsense2_camera/launch/rs_launch.py at ros2-development · IntelRealSense/Realsense-Ros, GitHub*. Available at: https://github.com/IntelRealSense/realsense-ros/blob/ros2-development/realsense2_camera/launch/rs_launch.py (Accessed: 28 April 2024).

Labbe, M. (2023) *Rtabmap_ros - Ros Wiki*. Available at: http://wiki.ros.org/rtabmap_ros (Accessed: 28 April 2024).

Labbe, M. (no date) *Wiki*, *ros.org*. Available at: https://wiki.ros.org/rtabmap_slam (Accessed: 28 April 2024).

Labbe, M. and Michaud, F., 2013. Appearance-based loop closure detection for online large-scale and long-term operation. *IEEE Transactions on Robotics*, 29(3), pp.734–745.

Labbé, M. and Michaud, F., 2018. Long-term online multi-session graph-based SPLAM with memory management. *Autonomous Robots*, 42(6), pp.1133–1150.

Labbé, M. and Michaud, F., 2019. RTAB-Map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2), pp.416–446.

Labbé, M. and Michaud, F., *Memory Management for Real-Time Appearance-Based Loop Closure Detection*,

Lu, D. V., Hershberger, D. and Smart, W.D., 2014. Layered costmaps for context-sensitive navigation. *IEEE International Conference on Intelligent Robots and Systems*. 31 October 2014 Institute of Electrical and Electronics Engineers Inc., pp. 709–715.

Macenski, S., Moore, T., et al., 2023. From the Desks of ROS Maintainers: A Survey of Modern & Capable Mobile Robotics Algorithms in the

Robot Operating System 2. Available at: http://arxiv.org/abs/2307.15236.

Macenski, S., Singh, S., Martin, F. and Gines, J., 2023. Regulated Pure Pursuit for Robot Path Tracking. Available at: http://arxiv.org/abs/2305.20026.

Merzlyakov, A. and MacEnski, S., 2021. A Comparison of Modern General-Purpose Visual SLAM Approaches. *IEEE International Conference on Intelligent Robots and Systems*. 2021 Institute of Electrical and Electronics Engineers Inc., pp. 9190–9197.

Nav2 (no date) *Writing a new Costmap2D plugin*, *Writing a New Costmap2D Plugin - Nav2 1.0.0 documentation*. Available at: https://navigation.ros.org/plugin_tutorials/docs/writing_new_costmap2d_plugin.html (Accessed: 28 April 2024).

Newans, J. (no date) *Joshnewans/articubot_one at humble*, *GitHub*. Available at: https://github.com/joshnewans/articubot_one/tree/humble (Accessed: 28 April 2024).

Open Robotics (no date) Setup and Configuration of the Navigation Stack on a Robot, ros.org. Available at: https://wiki.ros.org/navigation/Tutorials/RobotSetup#Global_Configuration (Accessed: 28 April 2024).

Open Source Robotics Foundation (2014) *Ros 2 overview*, *gazebo*. Available at: https://classic.gazebosim.org/tutorials/?tut=ros2_overview (Accessed: 15 April 2024).

ROS Planning (no date) *Navigation2/nav2_bringup at main · Ros-planning/navigation2*, *GitHub*. Available at: https://github.com/ros-planning/navigation2/tree/main/nav2_bringup (Accessed: 28 April 2024).

Sucan, I., Kay, J. and Meeussen, W. (no date) *Ros/robot_state_publisher*, *GitHub*. Available at: https://github.com/ros/robot_state_publisher?tab=readme-ov-file (Accessed: 28 April 2024).

Wang, X. et al., 2020. The evolution of LiDAR and its application in high precision measurement. *IOP Conference Series: Earth and Environmental Science*. 1 June 2020 Institute of Physics Publishing.

Withey, D.J. and Matebese, B.T., 2021. An OctoMap-based 3D CostMap. *2021 Rapid Product Development Association of South Africa - Robotics and Mechatronics - Pattern Recognition Association of South Africa: Digital Manufacturing: Industrialising Africa, RAPDASA-RobMech-PRASA 2021*. 2021 Institute of Electrical and Electronics Engineers Inc.

APPENDICES

Appendix A Code of the differential drive robot's URDF

```
<?xml version="1.0"?>
<robot xmlns:xacro="http://www.ros.org/wiki/xacro" >

    <xacro:include filename="inertial_macros.xacro"/>


    <xacro:property name="chassis_length" value="0.41"/>
    <xacro:property name="chassis_width" value="0.31"/>
    <xacro:property name="chassis_height" value="0.06"/>
    <xacro:property name="chassis_mass" value="1.0"/>
    <xacro:property name="wheel_radius" value="0.0635"/>
    <xacro:property name="wheel_thickness" value="0.04"/>
    <xacro:property name="wheel_mass" value="0.05"/>
    <xacro:property name="wheel_offset_x" value="0.31"/>
    <xacro:property name="wheel_offset_y" value="0.155"/>
    <xacro:property name="wheel_offset_z" value="0.00"/>
    <xacro:property name="caster_wheel_radius" value="0.0635"/>
    <xacro:property name="caster_wheel_mass" value="0.01"/>
    <xacro:property name="caster_wheel_offset_x" value="0.075"/>
    <xacro:property name="caster_wheel_offset_z"
value="${wheel_offset_z - wheel_radius + caster_wheel_radius}"/>

    <material name="white">
        <color rgba="1 1 1 1" />
    </material>

    <material name="orange">
        <color rgba="1 0.3 0.1 1"/>
    </material>

    <material name="blue">
        <color rgba="0.2 0.2 1 1"/>
    </material>

    <material name="black">
        <color rgba="0 0 0 1"/>
    </material>

    <material name="red">
        <color rgba="1 0 0 1"/>
    </material>

    <!-- BASE LINK -->
    <link name="base_link">

    </link>

    <!-- BASE_FOOTPRINT LINK -->

    <joint name="base_footprint_joint" type="fixed">
        <parent link="base_link"/>
        <child link="base_footprint"/>
        <origin xyz="0 0 0" rpy="0 0 0"/>
    </joint>

    <link name="base_footprint">
    </link>


    <!-- CHASSIS LINK -->

    <joint name="chassis_joint" type="fixed">
        <parent link="base_link"/>
        <child link="chassis"/>
        <origin xyz="${-wheel_offset_x} 0 ${-wheel_offset_z}"/>
    </joint>

    <link name="chassis">
        <visual>
            <origin xyz="${chassis_length/2} 0
${chassis_height/2}"/>
            <geometry>
                <box size="${chassis_length} ${chassis_width}
${chassis_height}"/>
            </geometry>
            <material name="orange"/>
        </visual>
        <collision>
            <origin xyz="${chassis_length/2} 0
${chassis_height/2}"/>
            <geometry>
                <box size="${chassis_length} ${chassis_width}
${chassis_height}"/>
            </geometry>
        </collision>
        <xacro:inertial_box mass="0.5" x="${chassis_length}"
y="${chassis_width}" z="${chassis_height}">
            <origin xyz="${chassis_length/2} 0
${chassis_height/2}" rpy="0 0 0"/>
        </xacro:inertial_box>
    </link>

    <gazebo reference="chassis">
        <material>Gazebo/Orange</material>
    </gazebo>

    <!-- LEFT WHEEL LINK -->

    <joint name="left_wheel_joint" type="continuous">
        <parent link="base_link"/>
        <child link="left_wheel"/>
        <origin xyz="0 ${wheel_offset_y} 0" rpy="-${pi/2} 0 0" />
        <axis xyz="0 0 1"/>
    </joint>

    <link name="left_wheel">
        <visual>
            <geometry>
                <cylinder radius="${wheel_radius}"
length="${wheel_thickness}"/>
            </geometry>
            <material name="blue"/>
        </visual>
        <collision>
            <geometry>
                <sphere radius="${wheel_radius}"/>
            </geometry>
        </collision>
        <xacro:inertial_cylinder mass="${wheel_mass}"
length="${wheel_thickness}" radius="${wheel_radius}">
            <origin xyz="0 0 0" rpy="0 0 0"/>
        </xacro:inertial_cylinder>
    </link>

    <gazebo reference="left_wheel">
        <material>Gazebo/Blue</material>
    </gazebo>

    <!-- RIGHT WHEEL LINK -->

    <joint name="right_wheel_joint" type="continuous">
        <parent link="base_link"/>
        <child link="right_wheel"/>
        <origin xyz="0 ${-wheel_offset_y} 0" rpy="${pi/2} 0 0" />
        <axis xyz="0 0 -1"/>
    </joint>

    <link name="right_wheel">
```

```
        <visual>
            <geometry>
                <cylinder radius="${wheel_radius}"
length="${wheel_thickness}"/>
            </geometry>
            <material name="blue"/>
        </visual>
        <collision>
            <geometry>
                <sphere radius="${wheel_radius}"/>
            </geometry>
        </collision>
        <xacro:inertial_cylinder mass="${wheel_mass}"
length="${wheel_thickness}" radius="${wheel_radius}">
            <origin xyz="0 0 0" rpy="0 0 0"/>
        </xacro:inertial_cylinder>
    </link>

    <gazebo reference="right_wheel">
        <material>Gazebo/Blue</material>
    </gazebo>


    <!-- CASTER WHEEL LINK -->

    <joint name="caster_wheel_joint" type="fixed">
        <parent link="chassis"/>
        <child link="caster_wheel"/>
        <origin xyz="${caster_wheel_offset_x} 0
${caster_wheel_offset_z}"/>
    </joint>


    <link name="caster_wheel">
        <visual>
            <geometry>
                <sphere radius="${caster_wheel_radius}"/>
            </geometry>
            <material name="white"/>
        </visual>
        <collision>
            <geometry>
                <sphere radius="${caster_wheel_radius}"/>
            </geometry>
        </collision>
        <xacro:inertial_sphere mass="${caster_wheel_mass}"
radius="${caster_wheel_radius}">
            <origin xyz="0 0 0" rpy="0 0 0"/>
        </xacro:inertial_sphere>
    </link>

    <gazebo reference="caster_wheel">
        <material>Gazebo/White</material>
        <mu1 value="0.001"/>
        <mu2 value="0.001"/>
    </gazebo>

    <!-- Add in joint_state_publisher so the wheel can be
displayed -->
    <gazebo>
        <plugin name="joint_state_publisher"
filename="libgazebo_ros_joint_state_publisher.so">
            <jointName>left_wheel,right_wheel</jointName>
        </plugin>
</gazebo>

</robot>
```

Appendix B Code of the launch file modified to launch RTAB-Map SLAM

```
import os
from ament_index_python.packages import
get_package_share_directory

from launch import LaunchDescription
from launch.actions import DeclareLaunchArgument,
SetEnvironmentVariable
from launch.substitutions import LaunchConfiguration
from launch.conditions import IfCondition, UnlessCondition
from launch_ros.actions import Node


def generate_launch_description():

    # package_name = 'articubot_one'
    # rtabmap_config =
os.path.join(get_package_share_directory(package_name), 'config',
'rtabmap_sync_node.yaml')

    use_sim_time = LaunchConfiguration('use_sim_time')
    qos = LaunchConfiguration('qos')
    localization = LaunchConfiguration('localization')

    rtabmap_config={
          'frame_id':'base_link',
          'use_sim_time':use_sim_time,
          'subscribe_rgbd':True,
          'subscribe_scan':True,
          'use_action_for_goal':True,
          'qos_scan':qos,
          'qos_image':qos,
          'qos_imu':qos,
          # RTAB-Map's parameters should be strings:
          'Reg/Strategy':'1',
          #'Reg/Force3DoF':'true',
          'RGBD/NeighborLinkRefining':'True',
          'Grid/RangeMin':'0.2', # ignore laser scan points on
the robot itself
          #'Optimizer/GravitySigma':'0' # Disable imu constraints
(we are already in 2D)
    }

    remappings=[
          ('/odom', '/diff_cont/odom'),
          ('/rgb/image', '/depth_camera/image_raw'),
          ('/rgb/camera_info', '/depth_camera/camera_info'),
          ('/depth/image', '/depth_camera/depth/image_raw')]
    return LaunchDescription([

        # Launch arguments
        DeclareLaunchArgument(
            'use_sim_time', default_value='true',
            description='Use simulation (Gazebo) clock if true'),

        DeclareLaunchArgument(
            'qos', default_value='2',
            description='QoS used for input sensor topics'),

        DeclareLaunchArgument(
            'localization', default_value='false',
            description='Launch in localization mode.'),

        # Nodes to launch
        Node(
            package='rtabmap_sync', executable='rgbd_sync',
output='screen', name='rtabmap_slam',
            parameters=[{'approx_sync':True,
'use_sim_time':use_sim_time, 'qos':qos}],
            remappings=remappings),

        # SLAM Mode:
        Node(
            condition=UnlessCondition(localization),
            package='rtabmap_slam', executable='rtabmap',
output='screen', name='rtabmap_slam',
            parameters=[rtabmap_config],
            remappings=remappings,
            arguments=['-d']),

        # Localization mode:
        Node(
            condition=IfCondition(localization),
            package='rtabmap_slam', executable='rtabmap',
output='screen',
            parameters=[rtabmap_config,
              {'Mem/IncrementalMemory':'False',
               'Mem/InitWMWithAllNodes':'True'}],
            remappings=remappings),

        Node(
            package='rtabmap_viz', executable='rtabmap_viz',
output='screen',
            parameters=[rtabmap_config],
             remappings=remappings),])
```

Appendix C The YAML file for the navigation.launch.py launch file in nav2_bringup package

```
bt_navigator:
  ros__parameters:
    use_sim_time: True
    global_frame: map
    robot_base_frame: base_link
    odom_topic: /diff_cont/odom
    bt_loop_duration: 10
    default_server_timeout: 20
    # 'default_nav_through_poses_bt_xml' and
'default_nav_to_pose_bt_xml' are use defaults:
    #
nav2_bt_navigator/navigate_to_pose_w_replanning_and_recovery.xml
    #
nav2_bt_navigator/navigate_through_poses_w_replanning_and_recover
y.xml
    # They can be set here or via a RewrittenYaml remap from a
parent launch file to Nav2.
    plugin_lib_names:
    - nav2_compute_path_to_pose_action_bt_node
    - nav2_compute_path_through_poses_action_bt_node
    - nav2_smooth_path_action_bt_node
    - nav2_follow_path_action_bt_node
    - nav2_spin_action_bt_node
    - nav2_wait_action_bt_node
    - nav2_assisted_teleop_action_bt_node
    - nav2_back_up_action_bt_node
    - nav2_drive_on_heading_bt_node
    - nav2_clear_costmap_service_bt_node
    - nav2_is_stuck_condition_bt_node
    - nav2_goal_reached_condition_bt_node
    - nav2_goal_updated_condition_bt_node
    - nav2_globally_updated_goal_condition_bt_node
    - nav2_is_path_valid_condition_bt_node
    - nav2_initial_pose_received_condition_bt_node
    - nav2_reinitialize_global_localization_service_bt_node
    - nav2_rate_controller_bt_node
    - nav2_distance_controller_bt_node
    - nav2_speed_controller_bt_node
    - nav2_truncate_path_action_bt_node
    - nav2_truncate_path_local_action_bt_node
    - nav2_goal_updater_node_bt_node
    - nav2_recovery_node_bt_node
    - nav2_pipeline_sequence_bt_node
    - nav2_round_robin_node_bt_node
    - nav2_transform_available_condition_bt_node
    - nav2_time_expired_condition_bt_node
    - nav2_path_expiring_timer_condition
    - nav2_distance_traveled_condition_bt_node
    - nav2_single_trigger_bt_node
    - nav2_goal_updated_controller_bt_node
    - nav2_is_battery_low_condition_bt_node
    - nav2_navigate_through_poses_action_bt_node
    - nav2_navigate_to_pose_action_bt_node
    - nav2_remove_passed_goals_action_bt_node
    - nav2_planner_selector_bt_node
    - nav2_controller_selector_bt_node
    - nav2_goal_checker_selector_bt_node
    - nav2_controller_cancel_bt_node
    - nav2_path_longer_on_approach_bt_node
    - nav2_wait_cancel_bt_node
    - nav2_spin_cancel_bt_node
    - nav2_back_up_cancel_bt_node
    - nav2_assisted_teleop_cancel_bt_node
    - nav2_drive_on_heading_cancel_bt_node

bt_navigator_navigate_through_poses_rclcpp_node:
  ros__parameters:
    use_sim_time: True

bt_navigator_navigate_to_pose_rclcpp_node:
  ros__parameters:
    use_sim_time: True

controller_server:
  ros__parameters:
    use_sim_time: True
    controller_frequency: 20.0
    min_x_velocity_threshold: 0.001
    min_y_velocity_threshold: 0.5
    min_theta_velocity_threshold: 0.001
    failure_tolerance: 0.3
    progress_checker_plugin: "progress_checker"
    goal_checker_plugins: ["general_goal_checker"] #
"precise_goal_checker"
    controller_plugins: ["FollowPath"]

    # Progress checker parameters
    progress_checker:
      plugin: "nav2_controller::SimpleProgressChecker"
      required_movement_radius: 0.5
      movement_time_allowance: 10.0
    # Goal checker parameters
    #precise_goal_checker:
```

```
    #  plugin: "nav2_controller::SimpleGoalChecker"
    #  xy_goal_tolerance: 0.25
    #  yaw_goal_tolerance: 0.25
    #  stateful: True
    general_goal_checker:
      stateful: True
      plugin: "nav2_controller::SimpleGoalChecker"
      xy_goal_tolerance: 0.25
      yaw_goal_tolerance: 0.25
    # DWB parameters
    FollowPath:
      plugin: "dwb_core::DWBLocalPlanner"
      debug_trajectory_details: True
      min_vel_x: 0.0
      min_vel_y: 0.0
      max_vel_x: 0.26
      max_vel_y: 0.0
      max_vel_theta: 1.0
      min_speed_xy: 0.0
      max_speed_xy: 0.26
      min_speed_theta: 0.0
      # Add high threshold velocity for turtlebot 3 issue.
      # https://github.com/ROBOTIS-
GIT/turtlebot3_simulations/issues/75
      acc_lim_x: 2.5
      acc_lim_y: 0.0
      acc_lim_theta: 3.2
      decel_lim_x: -2.5
      decel_lim_y: 0.0
      decel_lim_theta: -3.2
      vx_samples: 20
      vy_samples: 5
      vtheta_samples: 20
      sim_time: 1.7
      linear_granularity: 0.05
      angular_granularity: 0.025
      transform_tolerance: 0.2
      xy_goal_tolerance: 0.25
      trans_stopped_velocity: 0.25
      short_circuit_trajectory_evaluation: True
      stateful: True
      critics: ["RotateToGoal", "Oscillation", "BaseObstacle",
"GoalAlign", "PathAlign", "PathDist", "GoalDist"]
      BaseObstacle.scale: 0.02
      PathAlign.scale: 32.0
      PathAlign.forward_point_distance: 0.1
      GoalAlign.scale: 24.0
      GoalAlign.forward_point_distance: 0.1
```

```yaml
      PathDist.scale: 32.0
      GoalDist.scale: 24.0
      RotateToGoal.scale: 32.0
      RotateToGoal.slowing_factor: 5.0
      RotateToGoal.lookahead_time: -1.0

local_costmap:
  local_costmap:
    ros__parameters:
      update_frequency: 5.0
      publish_frequency: 2.0
      global_frame: odom
      robot_base_frame: base_link
      use_sim_time: True
      rolling_window: true
      width: 3
      height: 3
      resolution: 0.05
      robot_radius: 0.22
      plugins: ["voxel_layer", "inflation_layer"]
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 3.0
        inflation_radius: 0.55
      voxel_layer:
        plugin: "nav2_costmap_2d::VoxelLayer"
        enabled: True
        publish_voxel_map: True
        origin_z: 0.0
        z_resolution: 0.05
        z_voxels: 16
        max_obstacle_height: 2.0
        mark_threshold: 0
        observation_sources: pointcloud2
        pointcloud2:
          topic: /depth_camera/points
          data_type: "PointCloud2"
          max_obstacle_height: 2.0
          min_obstacle_height: 0.0
          obstacle_max_range: 2.5
          obstacle_min_range: 0.0
          raytrace_max_range: 3.0
          raytrace_min_range: 0.0
          clearing: True
          marking: True
      static_layer:
        plugin: "nav2_costmap_2d::StaticLayer"
        map_subscribe_transient_local: True
      always_send_full_costmap: True

global_costmap:
  global_costmap:
    ros__parameters:
      update_frequency: 1.0
      publish_frequency: 1.0
      global_frame: map
      robot_base_frame: base_link
      use_sim_time: True
      robot_radius: 0.22
      resolution: 0.05
      track_unknown_space: true
      plugins: ["static_layer", "obstacle_layer", "voxel_layer",
"inflation_layer"]
      obstacle_layer:
        plugin: "nav2_costmap_2d::ObstacleLayer"
        enabled: True
        observation_sources: scan
        scan:
          topic: /scan
          max_obstacle_height: 2.0
          clearing: True
          marking: True
          data_type: "LaserScan"
          raytrace_max_range: 3.0
          raytrace_min_range: 0.0
          obstacle_max_range: 2.5
          obstacle_min_range: 0.0
      static_layer:
        plugin: "nav2_costmap_2d::StaticLayer"
        map_subscribe_transient_local: True
      voxel_layer:
        plugin: "nav2_costmap_2d::VoxelLayer"
        enabled: True
        publish_voxel_map: True
        origin_z: 0.0
        z_resolution: 0.05
        z_voxels: 16
        max_obstacle_height: 2.0
        mark_threshold: 0
        observation_sources: pointcloud2
        pointcloud2:
          topic: /depth_camera/points
          data_type: "PointCloud2"
          max_obstacle_height: 2.0
          min_obstacle_height: 0.0
          obstacle_max_range: 2.5
          obstacle_min_range: 0.0
          raytrace_max_range: 3.0
          raytrace_min_range: 0.0
          clearing: True
          marking: True
      inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        cost_scaling_factor: 3.0
        inflation_radius: 0.55
      always_send_full_costmap: True

map_server:
  ros__parameters:
    use_sim_time: True
    # Overridden in launch by the "map" launch configuration or
provided default value.
    # To use in yaml, remove the default "map" value in the
tb3_simulation_launch.py file & provide full path to map below.
    yaml_filename: ""

map_saver:
  ros__parameters:
    use_sim_time: True
    save_map_timeout: 5.0
    free_thresh_default: 0.25
    occupied_thresh_default: 0.65
    map_subscribe_transient_local: True

planner_server:
  ros__parameters:
    expected_planner_frequency: 20.0
    use_sim_time: True
    planner_plugins: ["GridBased"]
    GridBased:
      plugin: "nav2_navfn_planner/NavfnPlanner"
      tolerance: 0.5
      use_astar: false
      allow_unknown: true

smoother_server:
  ros__parameters:
    use_sim_time: True
    smoother_plugins: ["simple_smoother"]
    simple_smoother:
      plugin: "nav2_smoother::SimpleSmoother"
      tolerance: 1.0e-10
      max_its: 1000
      do_refinement: True
```

```yaml
behavior_server:
  ros__parameters:
    costmap_topic: local_costmap/costmap_raw
    footprint_topic: local_costmap/published_footprint
    cycle_frequency: 10.0
    behavior_plugins: ["spin", "backup", "drive_on_heading",
"assisted_teleop", "wait"]
    spin:
      plugin: "nav2_behaviors/Spin"
    backup:
      plugin: "nav2_behaviors/BackUp"
    drive_on_heading:
      plugin: "nav2_behaviors/DriveOnHeading"
    wait:
      plugin: "nav2_behaviors/Wait"
    assisted_teleop:
      plugin: "nav2_behaviors/AssistedTeleop"
    global_frame: odom
    robot_base_frame: base_link
    transform_tolerance: 0.1
    use_sim_time: true
    simulate_ahead_time: 2.0
    max_rotational_vel: 1.0
    min_rotational_vel: 0.4
    rotational_acc_lim: 3.2

robot_state_publisher:
  ros__parameters:
    use_sim_time: True

waypoint_follower:
  ros__parameters:
    use_sim_time: True
    loop_rate: 20
    stop_on_failure: false
    waypoint_task_executor_plugin: "wait_at_waypoint"
    wait_at_waypoint:
      plugin: "nav2_waypoint_follower::WaitAtWaypoint"
      enabled: True
      waypoint_pause_duration: 200

velocity_smoother:
  ros__parameters:
    use_sim_time: True
    smoothing_frequency: 20.0
    scale_velocities: False
    feedback: "OPEN_LOOP"
    max_velocity: [0.26, 0.0, 1.0]
    min_velocity: [-0.26, 0.0, -1.0]
    max_accel: [2.5, 0.0, 3.2]
    max_decel: [-2.5, 0.0, -3.2]
    odom_topic: "/diff_cont/odom"
    odom_duration: 0.1
    deadband_velocity: [0.0, 0.0, 0.0]
    velocity_timeout: 1.0
```