INTER-UNIVERSITY MASTER'S DEGREE PROGRAMME

Università degli Studi di Milano

Università degli Studi di Milano - Bicocca

Università degli Studi di Pavia

Master's Degree Programme
Artificial Intelligence for Science and Technology

Real-Time 3D Semantic Mapping with Multimodal Data for Autonomous Robotic Navigation

Supervisor:
Prof. Paolo Napoletano
Co-supervisor:
Prof. Volker Krueger

Candidate name:
Nicolò Agostara
Registration number:
897237

Academic Year 2023/2024

Abstract

Effective navigation in complex environments is crucial for autonomous robots. Three-dimensional semantic maps, which incorporate object identities, volumes, and spatial positions, provide robots with a comprehensive understanding of their surroundings. This thesis aims to address the challenge of maintaining an accurate, real-time 3D semantic map despite computational constraints and sensor noise.

The research develops an efficient 3D semantic mapping system by leveraging pre-trained deep learning models for object segmentation using RGB images, fused with depth data for multimodal representation. For each frame, segmentation maps and tracking IDs are extracted, while cleaned depth data projects segmented objects into 3D space.

A KD-tree stores objects locations, enabling rapid spatial queries to re-identify previously mapped objects, updating existing point clouds instead of redundantly adding new ones. Additionally, point clouds are down sampled using voxel grids, optimizing both time complexity and memory efficiency, achieving an average throughput of 11 frames per second (FPS), meeting real-time performance requirements.

The system was evaluated in an indoor lab environment using a custom dataset of RGB-D image pairs and robot positional data. Multiple 3D maps of the lab were recreated using the RTAB-MAP SLAM framework and manually annotated with 3D-oriented bounding boxes. The system achieved a mean Average Precision (mAP) score of 0.8 at 40% objects overlap threshold, demonstrating high precision and recall in object mapping.

This approach illustrates that multimodal data fusion can significantly enhance both the accuracy and efficiency of real-time 3D semantic mapping, providing a robust tool for semantic scene understanding.

Contents

List of Figures

List of Algorithms

Introduction

In recent years, autonomous robotic systems have advanced significantly, particularly in their ability to navigate and understand complex environments. Central to these capabilities is the field of 3D semantic mapping, which enables robots to create detailed maps that incorporate not only the physical layout but also the semantic meaning of objects within their surroundings. The primary objective of this thesis is to develop a real-time 3D semantic mapping system that utilizes multimodal data fusion, combining RGB and depth data to achieve both accurate and efficient mapping suitable for environment understanding.

The objective of this thesis is to develop and evaluate a real-time 3D semantic mapping system that efficiently integrates RGB and depth data to create an accurate and detailed representation of an environment. The system tackles challenges related to objects localization in a three-dimensional space, object re-identification and processing speed by employing advanced object detection and tracking methods, along with robust point cloud processing. Pre-trained deep learning models are used for object segmentation and identification, while KD-Trees are implemented for rapid spatial queries, enabling efficient mapping that avoids redundancy. This approach is specifically designed to support robotic applications, where precise knowledge of object positions and identities is essential for effective navigation and scene understanding.

The following chapters provide an in-depth examination of the components, methodologies, and results of the thesis. The discussion begins with an overview of the hardware setup in **Chapter 1: Robot Overview**, which introduces the Heron Robot and the Intel RealSense L515 camera, detailing their roles in capturing multimodal data essential for semantic mapping. This is followed by **Chapter 2: Technical Background**, which reviews the key theoretical aspects utilized in this system, such as the YOLOv8 model for instance segmentation, the ByteTrack algorithm for consistent object tracking, and KD-Trees for rapid spatial querying.

The frameworks supporting the system are discussed in **Chapter 3: Tools and Framework for Robot Navigation**, with an emphasis on the modularity and flexibility offered by the Robot Operating System (ROS) framework. This chapter also explores how the system handles real-time data processing, which is essential for maintaining high performance. Subsequently, **Chapter 4: Point Cloud Processing and Noise Removal** delves into the specific challenges associated with point cloud processing, focusing on noise removal techniques essential for accurate object representation. Techniques such as voxel downsampling and statistical outlier removal are covered to demonstrate how they contribute to the overall system accuracy. In addition, this chapter introduces a methodology used to estimate the overlapping volume between two objects, a crucial aspect for determining object relationships and ensuring precise semantic mapping.

The core methodology is presented in **Chapter 5: Proposed Real-Time 3D Semantic Mapping System**, where the fusion of RGB and depth data for object detection is detailed. This chapter explains how KD-Trees are employed for efficient object re-identification, allowing the system to merge information about objects collected at different timestamps, ultimately building a comprehensive 3D semantic map that reconstructs full object representations. The creation and annotation of a custom dataset used to evaluate the system's performance are described in **Chapter 6: Custom Dataset Creation**, which involves capturing and labeling 3D data in an indoor environment.
The results of the system's evaluation are discussed in **Chapter 7: Results**, where metrics such as mean average precision (mAP), precision, recall and the Dice coefficient are used to assess mapping accuracy. These results demonstrate the system's effectiveness in generating reliable 3D semantic maps with high localization precision. Finally, **Chapter 8: Conclusions** provides a summary of the key findings and explores potential avenues for future research, aimed at further enhancing the precision and efficiency of 3D semantic mapping for robotic applications.

Chapter 1

Robot Overview: Heron

In this thesis, the Heron robot plays a crucial role in facilitating the detection and mapping of three-dimensional (3D) objects in real-time. The Heron system integrates various state-of-the-art components, including the MIR200 mobile industrial robot and the UR5e industrial manipulator, both of which contribute to the mobility and adaptability of the robot in an indoor environment. The configuration used in this project leverages the on-board sensors and computational resources of the Heron robot for object detection and mapping tasks, utilizing its camera system as the primary sensor for semantic mapping.

Figure 1.1: Heron Robot

1.1 Heron Robot Overview

The Heron robot, shown in figure 1.1, consists of two main components: the MIR200 mobile base, which ensures accurate and stable movement, and the UR5e robotic arm, which can be equipped with a variety of grippers for manipulation tasks. For navigation, the Heron robot employs multiple 2D SICK LiDAR sensors, which enable the robot to avoid obstacles and navigate autonomously. While these LiDAR sensors are used primarily for mobility and localization, the primary sensor used for object detection in this thesis is the Intel RealSense L515 camera, mounted on the UR5e arm. The reason why LiDAR is not integrated in the semantic mapping system is that, due to their bi-dimensional nature, they cannot provide a representation of objects in the scene. Figure 1.2 an example of the 3D projected data coming from LiDAR and RGB-D is shown.

1.1.1 MIR200 Mobile Base and Positioning System

The MIR200 mobile base provides precise and autonomous navigation capabilities. It uses a suite of sensors and an advanced positioning system to understand its location in space, combining both LiDAR and wheel odometry, making it ideal for dynamic environments. For the purposes of this thesis, the positioning system is utilized to establish the robot's precise pose relative to the environment, which is critical for projecting 3D objects detected by the camera into the semantic map. The MIR200 enables the robot to move efficiently across the indoor lab setting while maintaining a real-time understanding of its spatial position, which is a prerequisite for the accurate fusion of depth and visual data in 3D space.

1.1.2 UR5e Manipulator and Sensor Suite

The UR5e manipulator offers flexibility and range of motion, allowing for various sensor and gripper configurations depending on the task. In this project, the UR5e is equipped with an Intel RealSense L515 camera. While the UR5e arm has the potential for manipulation tasks, its role in this research is limited to providing an optimal viewpoint for capturing RGB-D data.

1.2 Intel RealSense L515 Camera

The Intel RealSense L515 camera is a key component of the Heron system, as it is responsible for capturing both RGB images and depth data, which are essential for

constructing 3D semantic maps. The L515 is a solid-state LiDAR camera, specifically designed for high-precision depth sensing in indoor environments. It captures high-quality depth images with single millimeter accuracy, making it well-suited for the demands of real-time 3D mapping in this research. Below, the main features and advantages of the L515 camera are discussed in detail.

1.2.1 Depth Sensing Capabilities

The Intel RealSense L515 operates based on Time-of-Flight (ToF) technology, emitting a laser beam with known frequency and measuring the time it takes for the light to return after reflecting off objects in the environment as the phase shift between the transmitted and received signal. This method allows the camera to generate highly accurate depth maps, which are crucial for creating detailed 3D models of the environment. The L515 is capable of capturing depth images at a range of up to 9 meters, with a depth resolution of up to 1024x768 pixels at 30 frames per second (FPS), making it ideal for real-time applications.

The camera's depth accuracy reaches one millimeter for objects within a short range, typically up to 2 meters. Furthermore Intel provide a series of post-processing filters, like spatial and temporal filters, directly inside the camera software that can be configured with a proprietary SDK [2].
In practice, the L515's depth data is fused with RGB images to generate a multimodal representation of the scene, which is then used to project segmented objects into 3D space.

1.2.2 RGB Camera Integration

The L515 also features a 1920x1080 resolution RGB camera, which captures high-quality color images synchronized with the depth data. The SDK [2], which can also be integrated within ROS, provides all the needed camera calibration packages that will define intrinsics and extrinsics parameters of the camera. This allow to align RGB-D image pairs and project point clouds in the 3D space.

This RGB-D pairing is crucial for the semantic segmentation task, where objects are identified and labeled based on both their visual appearance and their 3D geometry. In the proposed system, these RGB images are passed through a pre-trained deep learning model to produce segmentation masks, which are then combined with the depth data to generate 3D projections of the detected objects.

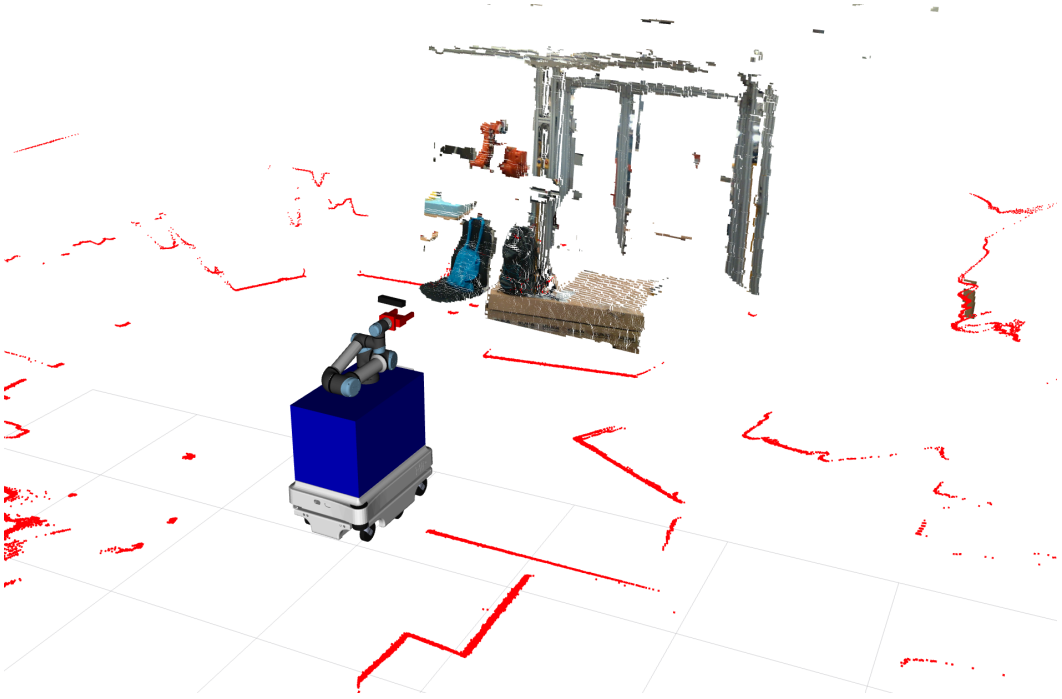


Figure 1.2: Frame shot of projected LiDAR and RGB-D scans. Red dots comes from SICK LiDAR sensors, the colored point cloud is the projected RGB-D image

1.3 Computational Resources

Heron is equipped with powerful computational resources, ensuring that it can handle the demands of real-time 3D object detection, segmentation, and mapping tasks. The system features the following hardware components:

- AMD Ryzen 7 3700X 8-Core Processor
- NVIDIA RTX 2070 GPU with 8GB of VRAM
- 32 GB RAM

These specifications provide more than enough computational power for running deep learning models for object segmentation, processing RGB-D data, and performing spatial queries in real-time.

Chapter 2

Technical Background

2.1 YOLOv8 - Instance Segmentation Model

2.1.1 Overview of YOLO Architecture

The You Only Look Once (YOLO) model is a *single-shot detector*, meaning it predicts bounding boxes, class probabilities, and object masks in a single pass through the network. This single-stage design allows for real-time performance, unlike two-stage detectors that involve a separate region proposal step. The basic components of the YOLO architecture include:

- **Backbone:** Extracts multi-scale features from the input image using convolutional layers, identifying patterns at various resolutions.
- **Neck:** Enhances and fuses features using structures like the Feature Pyramid Network (FPN) or Path Aggregation Network (PAN) for improved multi-scale detection.
- **Head:** Outputs the final predictions, which include bounding boxes, class labels, and confidence scores. In the instance segmentation task, the head also predicts pixel-wise object masks.

2.1.2 Advancements in YOLOv8

YOLOv8 builds on earlier YOLO versions by introducing several key innovations:

- **Anchor-Free Detection:** YOLOv8 eliminates the need for anchor boxes, predicting bounding boxes directly from object centroids. This reduces computational cost and improves generalization across different datasets.

- **Instance Segmentation Integration:** YOLOv8 seamlessly integrates object detection and instance segmentation tasks. It shares the backbone and neck components between the two tasks, maintaining low computational overhead while predicting pixel-level masks.
- **Enhanced Feature Fusion:** Similar to previous YOLO versions, YOLOv8 employs feature fusion from different scales in the backbone. However, YOLOv8 refines this approach with more efficient methods, improving the detection of objects of various sizes and enhancing segmentation accuracy.

2.1.3 YOLOv8 Loss Functions

YOLOv8 uses a combination of specialized loss functions to optimize object detection and instance segmentation tasks. These losses are combined to form the total loss, ensuring the model predicts accurate bounding boxes, class labels, and masks. Below are the main loss components used in YOLOv8 found in the official repository at [12].

Bounding Box Regression Loss

Bounding Box Regression Loss combines the Complete Intersection over Union (CIoU) Loss, which measures the overlap between the predicted and ground truth bounding boxes, and the Distribution Focal Loss (DFL) to improve the precision of bounding box predictions.

Complete Intersection over Union (CIoU) Loss CIoU Loss penalizes the model when the predicted bounding boxes do not match the ground truth boxes, considering overlap area, center point distance, and aspect ratio [15].

$$\mathcal{L}_{\text{CIoU}} = \frac{1}{N_{\text{pos}}} \sum_{i=1}^{N_{\text{pos}}} \left(1 - \text{CIoU}(B_{\text{pred}_i}, B_{\text{gt}_i})\right)$$

where:

- N_{pos} is the number of positive samples.
- B_{pred_i} is the predicted bounding box for the i-th positive sample.
- B_{gt_i} is the ground truth bounding box for the i-th positive sample.
- CIoU is the Complete Intersection over Union between the predicted and ground truth boxes, defined in [15].

Distribution Focal Loss (DFL) Distribution Focal Loss is designed to improve bounding box regression by modeling the bounding box coordinates as a probability distribution over discrete bins [5]. It minimizes the difference between the predicted and ground truth distributions, helping the model learn more precise box coordinates. This loss function aligns with the formulation in the Generalized Focal Loss paper [5].

Classification Loss

Classification Loss is computed using Binary Cross-Entropy (BCE) Loss because YOLOv8 treats each class as a separate binary classification problem. This approach allows the model to handle multi-label classification effectively, where each class prediction is independent of the others.

$$\mathcal{L}_{\text{cls}} = \frac{1}{N_{\text{pos}}} \sum_{i=1}^{N} \sum_{c=1}^{C} \left[-y_{i,c} \log\left(\sigma(p_{i,c})\right) - (1 - y_{i,c}) \log\left(1 - \sigma(p_{i,c})\right) \right]$$

where:

- N is the total number of samples.
- N_{pos} is the number of positive samples.
- C is the number of classes.
- $p_{i,c}$ is the predicted logit for class c of sample i.
- $y_{i,c}$ is the ground truth label for class c of sample i (1 if the object belongs to class c, 0 otherwise).
- $\sigma(p_{i,c})$ is the sigmoid activation of $p_{i,c}$.

Segmentation Loss

Segmentation Loss is applied to YOLOv8 models that perform instance segmentation. It is computed as the pixel-wise Binary Cross-Entropy (BCE) loss over each predicted mask, averaged over the number of positive masks.

$$\mathcal{L}_{\text{seg}} = \frac{1}{S_{\text{pos}}} \sum_{i=1}^{S_{\text{pos}}} \text{BCE}(M_{\text{pred}_i}, M_{\text{gt}_i})$$

where:

- S_{pos} is the number of positive masks (number of objects with masks).

- M_{pred_i} is the predicted mask for the i-th object.
- M_{gt_i} is the ground truth mask for the i-th object.
- $\text{BCE}(M_{\text{pred}_i}, M_{\text{gt}_i})$ is the Binary Cross-Entropy loss computed pixel-wise between the predicted and ground truth masks.

Total Loss for YOLOv8

The total loss in YOLOv8 is a weighted sum of the various individual losses. These weights are hyperparameters that control the importance of each loss component during training.

$$\mathcal{L}_{\text{total}} = \lambda_{\text{box}}\mathcal{L}_{\text{box}} + \lambda_{\text{cls}}\mathcal{L}_{\text{cls}} + \lambda_{\text{seg}}\mathcal{L}_{\text{seg}}$$

where:

- $\lambda_{\text{box}}, \lambda_{\text{cls}}, \lambda_{\text{seg}}$ are hyperparameters that adjust the relative importance of each loss component.

2.1.4 Real-World Applications

YOLOv8's efficient, anchor-free architecture and advanced loss functions make it suitable for real-time applications such as autonomous driving and robotics. The inclusion of CIoU Loss and Distribution Focal Loss enhances the model's ability to predict precise bounding boxes, improving detection accuracy. The Segmentation Loss enables precise instance segmentation, providing detailed scene understanding through pixel-level object delineation.

The YOLOv8 model, developed and released by Ultralytics [3], has been trained on the Common Objects in Context (COCO) dataset [6], which includes 80 different object categories, making the model suitable for common object segmentation tasks.

2.2 ByteTrack Algorithm

ByteTrack is a multi-object tracking (MOT) algorithm that significantly improves upon traditional tracking-by-detection methods by incorporating all detection boxes, including those with low confidence scores, into the data association process. Most previous approaches eliminate low-scoring detection boxes to avoid false positives,

which often results in missed detections, particularly for occluded or fast-moving objects. ByteTrack addresses this issue by associating low-confidence detections with tracklets using motion and appearance cues, recovering objects that would otherwise be lost [14].

The algorithm operates in two main stages. In the first stage, ByteTrack associates high-confidence detection boxes with existing tracklets based on motion similarity, which is typically calculated using the Kalman Filter. The Intersection-over-Union (IoU) between the predicted and detected bounding boxes is used as the similarity metric. Any unmatched tracklets and detection boxes from this stage proceed to the second stage, where ByteTrack associates low-confidence detections with the remaining tracklets. This second matching stage ensures that occluded objects or those experiencing motion blur are not prematurely discarded [14].

By leveraging low-confidence detection boxes in this way, ByteTrack achieves higher tracking accuracy without introducing significant false positives. The algorithm outperforms several state-of-the-art trackers across standard benchmarks, including MOT17 and MOT20, achieving first-place rankings in both detection accuracy and identity consistency. ByteTrack is also computationally efficient, running at 30 FPS on a V100 GPU, making it suitable for real-time applications [14].

Algorithm 1: Pseudo-code of ByteTrack

Input: Video sequence V, Object detector $YOLO$, Detection score threshold τ

Output: Object tracks T

for *each frame* f_k *in* V **do**
- Predict detection boxes $D_k \leftarrow YOLO(f_k)$;
- Separate D_k into high-confidence D_{high} and low-confidence D_{low} based on τ;
- Predict new tracklet locations using Kalman Filter;
- Associate D_{high} with tracklets using IoU as similarity;
- Perform secondary association between remaining tracklets and D_{low};
- Update tracklets and initialize new tracks for unmatched detections;

return T;

2.3 K-Dimensional Trees (KD-Trees)

KD-trees are a binary tree data structure that efficiently organizes points in a k-dimensional space, enabling fast nearest-neighbor and range searches. First introduced by Jon Louis Bentley in 1975 [1], KD-trees are used widely in applications such as robotics, where 3D point clouds from sensors like LiDAR need efficient querying. In this thesis they are used to store all the mapped objects centroids. This enables spatial queries to determine if newly detected objects overlap with previously mapped ones, facilitating efficient re-identification and reducing redundancy in the mapping process.

2.3.1 Construction of KD-Trees

A KD-tree is built by recursively splitting the dataset along one dimension at a time, alternating between dimensions at each level. The process begins by selecting the median value of a dimension (starting with the first dimension) and using it to split the data into two subsets. This median point becomes the node, and the process repeats for the remaining points and dimensions. This approach ensures that the data is hierarchically partitioned into nested regions, allowing for efficient spatial querying. The construction time is $O(n \log n)$, where n is the number of data points.

2.3.2 Nearest-Neighbor Search

KD-trees are particularly efficient for nearest-neighbor search. Given a query point, the tree is traversed to find the region containing the point. After locating the nearest region, backtracking through the tree helps find other points that may be closer. The search complexity is $O(\log n)$ for balanced KD-trees, making this approach highly suitable for real-time applications such as robotics navigation.

2.3.3 KD-Trees in SciPy

KD-trees are implemented in Python's SciPy [13] library under the `scipy.spatial.KDTree` class. This implementation supports nearest-neighbor queries and range searches. The following code snippet demonstrates how to create and query a KD-tree in SciPy:

```
from scipy.spatial import KDTree
points = [(1, 2, 3), (3, 5, 6), (9, 6, 9), (8, 8, 8)]
tree = KDTree(points)
```

```
# Query nearest neighbor to point (7, 7, 7)
dist, index = tree.query((7, 7, 7))
dist, index
#1.73, 3
```

Chapter 3

Tools and Framework for Robot Navigation

3.1 Robot Operating System (ROS)

Robot Operating System (ROS)[7] is a middleware framework that provides the tools and libraries required to develop, simulate, and deploy robotic applications. Despite its name, ROS is not a standalone operating system but a flexible communication layer that allows various robotic components to exchange information in a distributed manner. In this project, ROS Noetic is utilized due to the preconfiguration of the Heron robot1 and its compatibility with Python 3, which is crucial for integrating modern machine learning and data processing techniques into robotic systems.

3.1.1 Core Concepts of ROS

ROS operates by structuring the robot's software into individual processes called *nodes*, each responsible for performing a specific task such as sensor data acquisition, control, or decision making. Nodes communicate with one another through a messaging system, which is central to ROS architecture.
The core components of ROS include:

- **Nodes**: These are individual processes that perform computations. Each node typically handles a specific function, such as processing sensor data or controlling an actuator.
- **Messages**: Nodes communicate by sending messages. A message is a data structure consisting of fields like integers, floats, arrays, or custom data types.

These messages are transmitted through topics.

- **Topics**: ROS uses a publish/subscribe model for message exchange. Nodes publish messages to specific topics, and other nodes subscribe to these topics to receive the messages. This decouples the sender and receiver, enabling asynchronous communication.
- **Services**: While topics allow for continuous message passing, services enable synchronous communication. A service allows a node to send a request to another node and wait for a response. This is useful for tasks that require immediate feedback, such as commanding a robot arm or querying sensor information.
- **Actionlib**: For tasks that require feedback or longer execution times, ROS provides *actionlib*. This allows nodes to execute goals asynchronously, providing real-time feedback on the progress and the ability to preempt tasks if necessary.

3.1.2 ROS and Python Integration

In this project, Python is used as the primary programming language to interact with ROS. The *rospy* library provides Python bindings for ROS, enabling developers to create nodes, publish and subscribe to topics, and utilize services and actions with ease.

The typical workflow of a ROS node in Python involves:

- Initializing the node using `rospy.init_node`.
- Publishing or subscribing to topics via `rospy.Publisher` or `rospy.Subscriber`.
- Defining service calls using `rospy.Service` and responding to them as required.

3.1.3 Modularity and Communication

One of the key strengths of ROS is its modularity. Each node is an independent process that communicates with other nodes via topics or services. This modular approach makes the system highly scalable and allows for easy testing and maintenance of individual components. Nodes can be run independently or in groups, facilitating debugging and system development.

Furthermore, ROS provides tools for managing and configuring the robot's software stack:

- **roslaunch**: This tool allows the user to launch multiple nodes simultaneously, including configuration and parameter setting.
- **rviz**: A powerful visualization tool for displaying sensor data, robot state, and the environment in 3D.
- **rqt**: A graphical interface for inspecting and debugging the communication between nodes, topics, and services during runtime.

3.2 Open3D for Point Cloud Computation

Open3D [16] is a powerful open-source library designed for 3D data processing, with a particular emphasis on point cloud computation. In this thesis, Open3D is utilized extensively for managing, processing, and visualizing 3D point clouds generated by a depth camera 1.2. The ability to perform these computations efficiently is crucial for real-time applications in robotics.

3.2.1 Point Cloud Operations

Point clouds are sets of 3D points that represent objects in space. These data are typically dense and noisy, requiring various preprocessing techniques for accurate analysis and manipulation. Open3D provides a comprehensive set of functions for handling point cloud data, such as:

- **Voxel Downsampling**: This function reduces the density of the point cloud by grouping points into voxels (3D grid cells) and replacing all points within a voxel with their centroid. This process greatly improves computational efficiency without significantly affecting the accuracy of the 3D representation. It is particularly useful for reducing memory and computational overhead when working with large-scale point clouds.
- **Statistical Outlier Removal**: Point cloud data can contain noise or erroneous points due to sensor inaccuracies. The statistical outlier removal function eliminates points that are far from their neighbors by evaluating the distance of each point to its surrounding neighbors and filtering out those that deviate from the local distribution. This technique helps in cleaning up noisy data, improving the overall quality of the point cloud.

- **Radius Outlier Removal**: Similar to statistical outlier removal, radius outlier removal removes points that have fewer neighbors than a specified threshold within a given radius. This method is especially effective when dealing with sparsely distributed noise points.

- **Minimal Oriented Bounding Box**: Computes the smallest bounding box enclosing a 3D point cloud, aligning with the data's principal axes for a minimal volume fit.

3.2.2 GPU-Accelerated Computations

In this thesis, point cloud computations are accelerated using GPU resources. Open3D provides support for GPU-accelerated operations via its integration with CUDA, which is leveraged to handle large datasets and real-time processing demands. By utilizing GPU for voxel downsampling, outlier removal, and other geometric transformations, the computational performance is significantly improved, enabling faster data processing and more efficient handling of high-density 3D point clouds.

The combination of Open3D's point cloud processing functionalities with GPU acceleration makes it ideal for robotics applications that require real-time performance, such as navigation, obstacle avoidance, and mapping.

3.3 Eye-in-Hand Calibration

Eye-in-Hand calibration is essential in robotics for determining the spatial relationship between a robot's end-effector (the "hand") and an attached vision sensor (the "eye"). Accurate calibration ensures precise perception and manipulation, enabling the robot to interact effectively with its environment.

3.3.1 Tools and Software

For this calibration, the *MoveIt!* motion planning framework is utilized, along with the `moveit_calibration_plugins` package. The calibration process follows the guidelines provided in the MoveIt! hand-eye calibration tutorial[1].

[1] `https://github.com/moveit/moveit_tutorials/blob/master/doc/hand_eye_calibration/hand_eye_calibration_tutorial.rst`

3.3.2 Prerequisites

Before performing the calibration, ensure the following:

- **MoveIt! Setup**: MoveIt! is installed and properly configured with the robot.
- **Aruco Marker Board**: A physical Aruco marker board is available and detectable by the vision sensor.
- **Robot Control**: The robot's end-effector can be moved to various positions and orientations.

3.3.3 Calibration Procedure

The calibration involves capturing multiple poses where the robot's end-effector and the Aruco marker board are in different relative positions:

1. **Data Collection**: Use the `moveit_calibration_gui` to capture images at various poses where the marker board is visible to the sensor. Move the end-effector to different positions and orientations, capturing at least 10–15 poses for improved accuracy.
2. **Calibration Computation**: Initiate the calibration computation within the GUI. The software calculates the transformation between the end-effector and the vision sensor using the collected data.

3.3.4 Mathematical Formulation

The calibration problem aims to find the transformation $\mathbf{X}$ from the end-effector to the camera and the transformation $\mathbf{Y}$ from the robot base to the marker board, satisfying:

$$\mathbf{A}_i \mathbf{X} = \mathbf{Y} \mathbf{B}_i, \tag{3.1}$$

where $\mathbf{A}_i$ is the known transformation from the robot base to the end-effector, and $\mathbf{B}_i$ is the observed transformation from the camera to the marker board for each pose i. By collecting data over multiple poses, this set of equations can be solved using optimization techniques, such as least squares fitting, to determine the unknown transformations $\mathbf{X}$ and $\mathbf{Y}$.
In figure 3.1 a screenshot of the calibration procedure is shown where the computed reprojection error is around 1 cm.

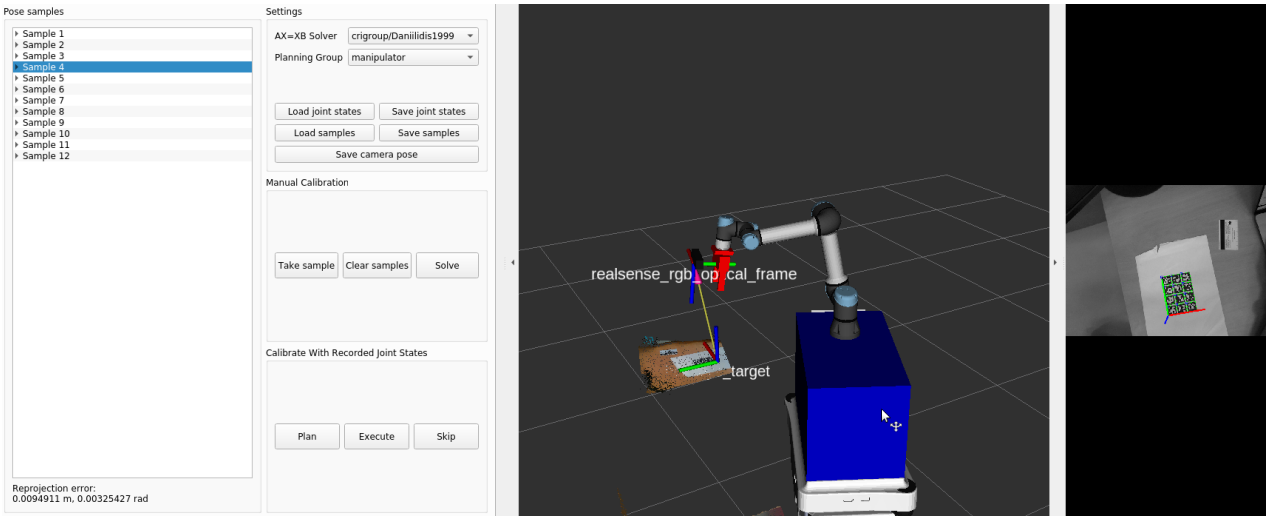


Figure 3.1: Eye-in-hand calibration with MoveIt!

Chapter 4

Point Cloud Processing and Noise Removal

4.1 Object Point Cloud Formation

Accurate formation of object point clouds is essential for reliable 3D semantic mapping. This process involves generating a 3D representation of detected objects from 2D image data and corresponding depth information.

4.1.1 Mask-Depth Overlap

After instance segmentation is performed in the Detection Node, binary segmentation masks for each detected object are obtained. These masks are overlapped with the depth image to isolate the depth information corresponding to each object. This step effectively segments the object's depth data from the rest of the scene as shown in figure 4.1 where annotations are performed thanks to the Supervision library [8].

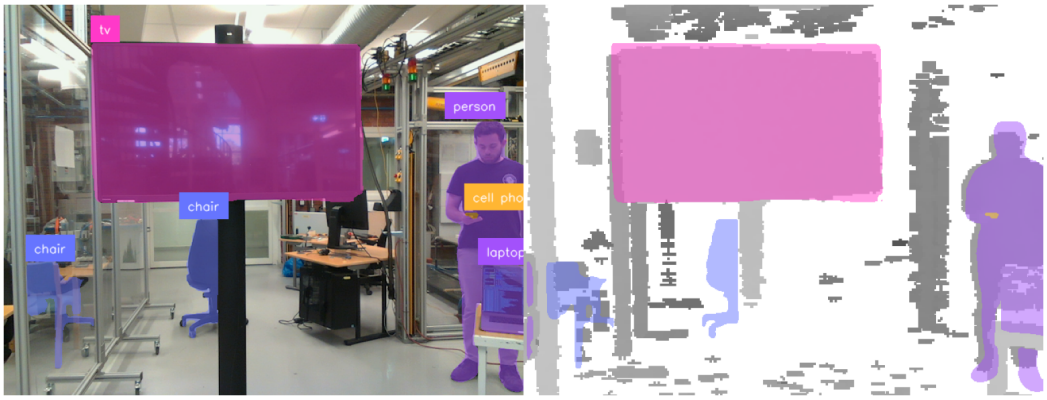


Figure 4.1: RGB-D pair annotated with segmentation masks

4.1.2 Point Cloud Projection

The pixel coordinates and depth values are transformed into 3D coordinates to form the point cloud of the object. Using the intrinsic parameters of the camera focal lengths f_x, f_y and optical center coordinates c_x, c_y the transformation from pixel coordinates (u, v) to camera coordinates (X_c, Y_c, Z_c) is performed using the following equations:

$$X_c = \frac{(u - c_x) \cdot Z_c}{f_x}$$
$$Y_c = \frac{(v - c_y) \cdot Z_c}{f_y}$$
$$Z_c = \text{depth}(u, v)$$

Here, $\text{depth}(u, v)$ is the depth value from the masked depth image at pixel coordinates (u, v). This transformation maps the 2D pixel coordinates into 3D space in the camera's coordinate frame.

To complete the transformation to world coordinates, the extrinsic parameters of the camera, which describe the camera's position and orientation in the world frame, are needed. These parameters are given as a 3×3 rotation matrix R and a translation vector t.

Using the tf2_ros package, we can retrieve the projection matrix $P = [R|t]$ at a given timestamp ts.

Then we compose the rototranslation matrix as:

$$P = \begin{bmatrix} R_{11} & R_{12} & R_{13} & t_x \\ R_{21} & R_{22} & R_{23} & t_y \\ R_{31} & R_{32} & R_{33} & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

and finally we compute the point cloud in the map reference frame with:

$$\mathbf{P}_{\text{map}} = P \cdot \mathbf{P}_{\text{camera}}$$

where $\mathbf{P}_{\text{camera}}$ is the point cloud in the camera frame, augmented with homogeneous coordinates, i.e.,

$$\mathbf{P}_{\text{camera}} = \begin{bmatrix} X_c \\ Y_c \\ Z_c \\ 1 \end{bmatrix}$$

and $\mathbf{P}_{\text{map}}$ is the transformed point cloud in the map frame.

Due to segmentation masks inaccuracies and noisy depth data, a raw projection is not enough to represent an object in 3D space, as shown in figure 4.2, therefore further processing is needed.

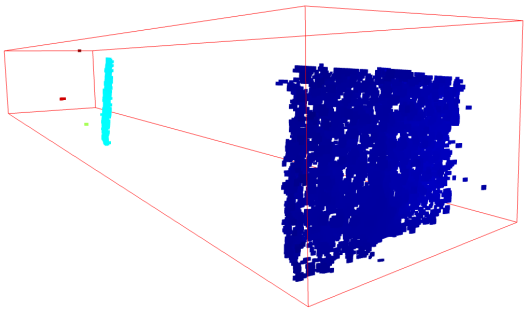


Figure 4.2: Raw point cloud projection of TV object and bounding box

4.1.3 Voxel Downsampling

Point clouds can contain a large number of points, which may be computationally intensive to process in real time. Voxel downsampling reduces the number of points while preserving the overall structure of the object. The space occupied by the point cloud is divided into a 3D grid of voxels of a specified size `VOXEL_SIZE`. Points within each voxel are approximated by a single point, typically the centroid of the points in that voxel. This reduces computational load and accelerates subsequent processing steps. In figure 4.3 one can observe a voxel-downsampled version of the raw point cloud shown in figure 4.2. Going from 61170 to 532 points with a `VOXEL_SIZE` parameter of 0.03 meters.

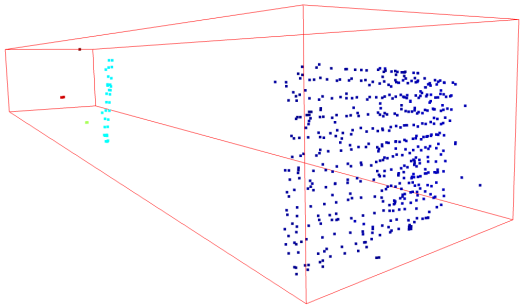


Figure 4.3: Voxel downsampled point cloud

4.2 Noise Removal Techniques

To improve the quality of the point cloud data, noise removal techniques are applied. These techniques help eliminate outliers and artifacts that can adversely affect mapping accuracy.

4.2.1 Statistical Outlier Removal

Statistical outlier removal analyzes the distribution of point neighborhoods to identify and remove points that are significantly different from the majority. Points with a mean distance to their neighbors that is much higher than the average are considered outliers and are removed. This method enhances the point cloud by eliminating anomalies caused by sensor noise or erroneous measurements.

4.2.2 Radius Outlier Removal

Radius outlier removal focuses on spatial density by removing points that have fewer neighboring points within a specified radius than a given threshold. This technique effectively eliminates isolated points that do not contribute meaningful information about the object's structure. Figure 4.4 shows the result of the previously shown pointcloud (figure 4.3) after radius and statistical outlier removal. It can be noticed how the new computed minimal oriented bounding box is now around the correct TV point cloud, not considering outliers.

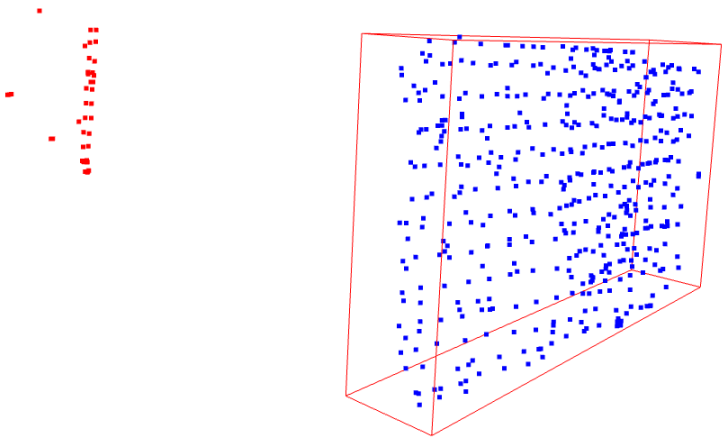


Figure 4.4: Resulting Pointcloud after statistical and radius outlier removal. Blue points are inliers, red points are outliers.

4.2.3 Mitigating Depth Bleeding

Depth bleeding is an artifact where the edges of objects appear larger due to inaccuracies in depth measurement, causing object boundaries to bleed into surrounding areas, as shown in Figure 4.5.

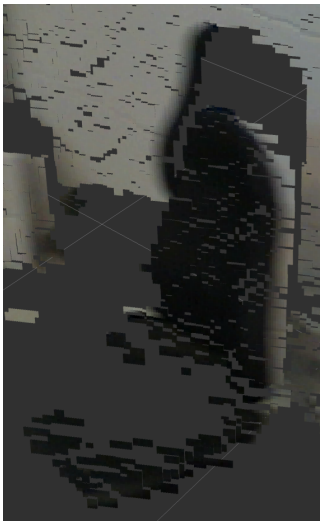
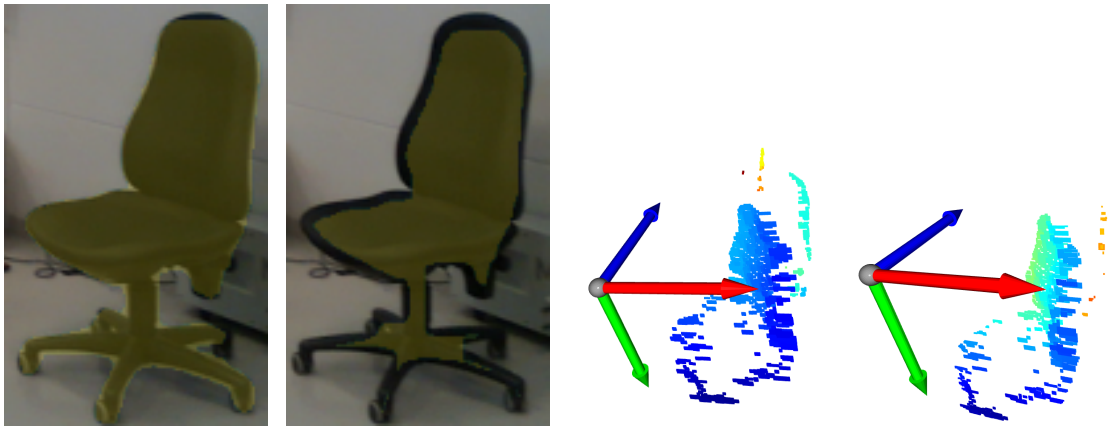


Figure 4.5: Example of depth bleeding artifact.

This artifact leads to distorted object boundaries and can degrade the accuracy of the point cloud representation.

Mask Erosion

To mitigate depth bleeding, a morphological operation called erosion is applied to the segmentation masks. Mask erosion reduces the size of the masks by removing pixels from the edges, effectively eliminating the pixels most affected by depth bleeding. This results in more accurate object boundaries and improves the quality of the point cloud.
Figure 4.6 illustrates the effect of mask erosion on both the segmentation mask and the resulting point cloud.



(a) Mask without erosion **(b)** Mask with erosion **(c)** Point cloud without mask erosion **(d)** Point cloud with mask erosion

Figure 4.6: Comparison between computed point clouds with and without mask erosion

4.3 Overlap Between Two Point Clouds

Quantifying the overlap between two point clouds in 3D space is crucial for accurate object re-identification. Consider the example shown in Figure 4.7, where the red points represent the stored point cloud of a couch, and the blue points represent a new detection. While it is visually apparent that both point clouds likely belong to the same object, a quantitative measure is needed to confirm this overlap.

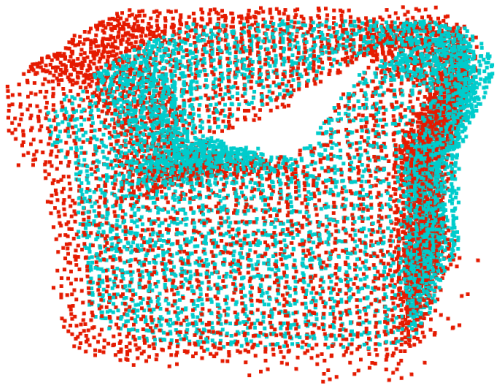


Figure 4.7: Overlap of two point clouds: red for the stored object, blue for the new detection.

4.3.1 Occupancy Grid Approximation

To quantify the overlap, we approximate each point cloud with a 3D occupancy grid. This grid divides the 3D space into fixed-size cubic cells (voxels), similar to the voxelization process discussed earlier. For this purpose, we assign a value to each voxel based on whether it contains points from the point cloud.

For each point $\mathbf{p}_i = (x_i, y_i, z_i)$ in the point cloud P, we map it to a grid cell (i, j, k) using the cell size s:

$$i = \left\lfloor \frac{x_i}{s} \right\rfloor, \quad j = \left\lfloor \frac{y_i}{s} \right\rfloor, \quad k = \left\lfloor \frac{z_i}{s} \right\rfloor$$

The occupancy grid function $O_P(i, j, k)$ is defined as:

$$O_P(i, j, k) = \begin{cases} 1, & \text{if cell } (i, j, k) \text{ contains at least one point from } P \\ 0, & \text{otherwise} \end{cases}$$

This function effectively indicates whether a grid cell is occupied by the point cloud P creating its approximation.

An illustration of the occupancy grid approximation is shown in Figure 4.8. In this visualization, green cells represent matching occupied cells between the two point clouds, while red cells represent non-matching cells.

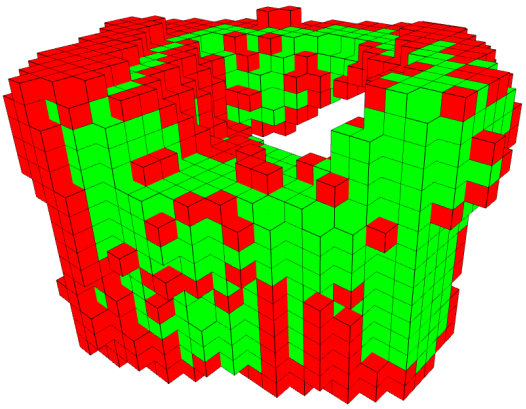


Figure 4.8: 3D Occupancy Grid visualization: green cells represent matching occupied cells, red cells represent non-matching cells.

4.3.2 Dice Coefficient

To quantify the similarity between the two occupancy grids, we use the Dice coefficient, a statistical measure of overlap between two sets. The Dice coefficient D between two occupancy grids O_P and O_Q is defined as:

$$D = \frac{2|O_P \cap O_Q|}{|O_P| + |O_Q|}$$

The Dice coefficient ranges from 0 to 1, where:

- $D = 1$ indicates perfect overlap between the two point clouds.
- $D = 0$ indicates no overlap between the two point clouds.

A higher Dice coefficient suggests a greater degree of similarity between the two point clouds, indicating that they likely represent the same object, for example, the dice coefficient of the occupancy grids shown in figure 4.8 is 0.65.

4.4 Computing Minimal Oriented Bounding Box from a Point Cloud

For visualization purposes we want to visualize the minimal oriented bounding box that encloses a point cloud. This is a problem with 9 Degrees of Freedom (DoF) since we need to define translations, rotations and extends of all the x, y, z axis. This procedure is carried on from the Open3D[16] library in the following way:

1. Convex Hull Computation Given a set of points $\{\mathbf{p}_i \in \mathbb{R}^3\}$, compute the convex hull to focus on the outer boundary points, yielding vertices $\{\mathbf{v}_j\}$.

2. Principal Component Analysis (PCA) Compute the centroid $\boldsymbol{\mu}$ and the covariance matrix $\mathbf{C}$ of the convex hull points:

$$\boldsymbol{\mu} = \frac{1}{M}\sum_{j=1}^{M}\mathbf{v}_j, \quad \mathbf{C} = \frac{1}{M}\sum_{j=1}^{M}(\mathbf{v}_j - \boldsymbol{\mu})(\mathbf{v}_j - \boldsymbol{\mu})^\top$$

Perform eigenvalue decomposition on $\mathbf{C}$:

$$\mathbf{C} = \mathbf{R}\boldsymbol{\Lambda}\mathbf{R}^\top$$

where $\boldsymbol{\Lambda} = \text{diag}(\lambda_1, \lambda_2, \lambda_3)$ are eigenvalues (sorted in descending order) and $\mathbf{R} = [\mathbf{r}_1, \mathbf{r}_2, \mathbf{r}_3]$ are the corresponding eigenvectors. Ensure $\mathbf{R}$ forms a right-handed coordinate system by setting $\mathbf{r}_3 = \mathbf{r}_1 \times \mathbf{r}_2$.

3. Transformation to Local Coordinates Transform the convex hull points into the local coordinate system aligned with the principal axes:

$$\mathbf{v}'_j = \mathbf{R}^\top (\mathbf{v}_j - \boldsymbol{\mu})$$

4. Axis-Aligned Bounding Box in Local Coordinates Compute the minimum and maximum coordinates along each local axis:

$$x_{\min} = \min_j v'_{j1}, \quad x_{\max} = \max_j v'_{j1}, \quad \text{and similarly for } y \text{ and } z$$

Calculate the extents $\mathbf{e} = [e_x, e_y, e_z]^\top$ with $e_x = x_{\max} - x_{\min}$,
and the center $\mathbf{c}' = \left[\frac{x_{\min}+x_{\max}}{2}, \frac{y_{\min}+y_{\max}}{2}, \frac{z_{\min}+z_{\max}}{2}\right]^\top$.

5. Transformation Back to World Coordinates Transform the bounding box center back to the original coordinate system:

$$\mathbf{c} = \mathbf{R}\mathbf{c}' + \boldsymbol{\mu}$$

The OBB is defined by center $\mathbf{c}$, orientation $\mathbf{R}$, and extents $\mathbf{e}$, it tightly encloses the point cloud with minimal volume, aligned along the data's principal directions as shown in figure 4.9

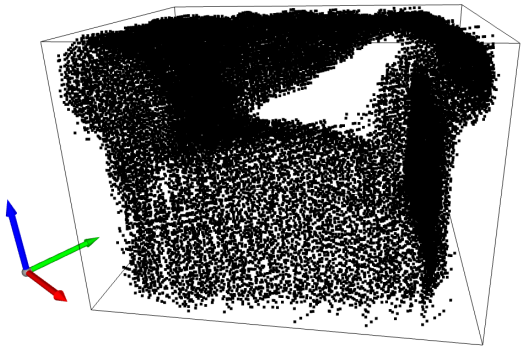


Figure 4.9: Minimal Oriented Bounding Box

Chapter 5

Proposed Real-Time 3D Semantic Mapping System

5.1 Introduction

This chapter presents an in-depth analysis of the software architecture implemented for the 3D semantic mapping system, which is structured around multiple ROS nodes, each with a specific functionality. The system ensures efficient real-time updates and accurate 3D object representations by modularizing tasks such as object detection, mapping, and world management. Figure 5.1 represents an abstract view of the message flow between the nodes. The following nodes are included:

- **Camera Node**: Responsible for capturing RGB-D image pairs from the Realsense L515 camera and publishing them to the ROS network.

- **Detection Node**: Subscribes to the RGB and depth topics, performs instance segmentation, and publishes the results to the ROS network.

- **Mapping Node**: Subscribes to the detection result topic and creates a 3D point cloud of the detected objects. It acts as an intermediate node between the Detection Node and the World Node, containing information about different reference frames.

- **World Node**: Manages the state of the environment by maintaining and updating object entries, removing outdated or redundant objects, and tracking objects over time, with object re-identification.

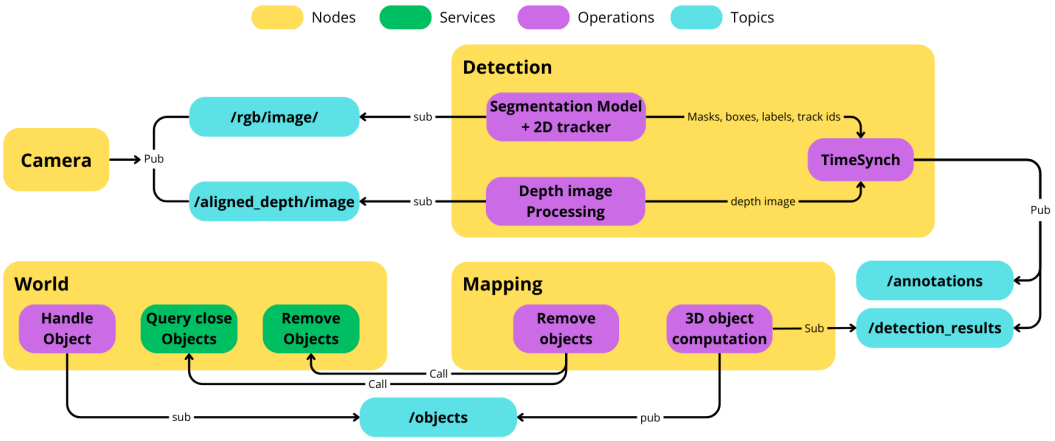


Figure 5.1: Software Structure

5.2 Detection Node

The Detection Node utilizes the YOLOv8 model for real-time instance segmentation on the robot's sensor inputs. It subscribes to synchronized RGB and depth topics from the Camera Node, performs objects instance segmentation, and publishes the results to the ROS network.

To ensure consistent object identification across frames, the ByteTrack algorithm (see Section 2.2) is integrated to assign unique IDs to each detected object in the 2D RGB images. Additionally, a custom 3D re-identification procedure is employed in the World Node.

From the model's output, the following information is extracted for each detected object:

1. **Class Labels**: The class labels of the detected objects.
2. **Segmentation Masks**: Binary masks indicating the pixels belonging to each object.
3. **Track IDs**: Unique IDs used to track objects across frames.
4. **Prediction Confidence Scores**: The confidence scores for each detected object.

Upon receiving RGB and depth images, the Detection Node's callback function processes the data and prepares it for the Mapping Node. Detection results are

published on the `/detection_results` topic using a custom message containing header information, IDs, scores, masks, corresponding depth images, and labels. This message provides all necessary information for the Mapping Node to generate 3D point clouds.

5.3 Mapping Node

The Mapping Node is responsible for projecting detected objects from two-dimensional to three-dimensional space by integrating depth, and segmentation data. It then transforms the resulting 3D coordinates from the camera reference frame to the world reference frame. Serving as an intermediary between the Detection Node and the World Node, the Mapping Node subscribes to the detection result topic, processes the data to generate 3D point clouds in the world frame, and subsequently publishes these objects for management by the World Node.

The callback function for the `/detection_results` topic works as follow:

- Extract timestamp ts from header
- Query tf2_ros for $[R|t]$ matrix between camera frame and world frame at ts.
- Project point cloud as explained in section 4.1 obtaining points in the world reference frame.
- Create custom `Object` message with label, point cloud, confidence score and detection time.
- Publish the message on the `/objects` topic.

5.4 World Node

The World Node acts as the central component in maintaining and updating the state of the environment. It manages object entries, removes redundant objects based on time thresholds and tracks objects over time. This node interacts with other nodes through topics and services, providing an up-to-date global map view.

5.4.1 Object Representation

Each detected object is represented by an instance of the `Obj` class, which encapsulates all the necessary information, including the object's point cloud, label,

prediction confidence score, 3D bounding box, and timestamps indicating when it was first and last observed.

For the purpose of simplifying the system's operation, a distinction between static and dynamic object categories must be established prior to execution. This classification allows the system to update the representation of moving objects using only their most recent detection, ensuring that transient entities are accurately tracked in real time. Conversely, for static objects, the system aggregates and merges all detections over time, enhancing the completeness and accuracy of their representation under different point of views. This approach optimizes the processing of dynamic objects while building a comprehensive model of static objects through accumulated observations. From now on we will talk about mapping static objects since, with the assumption made before, mapping dynamic entities becomes a straightforward task.
Key functionalities of the `Obj` class include:

- **Updating Object**: Merging a new detection with the existing representation of the object, updating its point cloud, confidence score and label.
- **Point Cloud Cleaning**: It applies statistical and radius outlier removal in order to keep a clean object representation as explained in section 4.2.
- **Computing Bounding Boxes**: Computing the minimum oriented bounding box for the object's point cloud as explained in section 4.4.
- **Resource Management**: Ensuring that GPU memory and other resources are properly managed.

Merging Objects

When a new detection corresponds to an already stored object, the two objects are merged to enhance the overall object representation. The final label is selected based on the most frequent assigned class and, in the eventuality of ties, with the highest prediction confidence score. The combined point cloud incorporates data from both the previous and the new detection, allowing for a more complete representation of the object's volume from multiple perspectives. Following this merge, the resulting point cloud undergoes voxelization and is then cleaned as detailed in Section 4.2. Finally, an updated oriented bounding box is computed as described in Section 4.4.

5.4.2 Services Provided

The `World` node maintains a collection of all detected objects and uses a KDTree for efficient spatial querying. It provides the following ROS services:

- **Remove Objects**: Removes objects with specified object IDs from the world model.
- **Query Objects**: Using a KDTree efficiently query objects within a certain distance threshold from a specific point.
- **Get All Objects**: Retrieves all objects currently stored in the world model.
- **Clean Up**: Removes outdated or invalid objects based on certain criteria.

And subscribes to the topic `/objects` to asynchronously receive newly detected objects.

A locking mechanism is applied to control access to the world model, which is the only resource requiring a lock. This ensures that only one operation can modify the world model at a time, thereby preventing inconsistencies in object representation. Since there is only a single locked resource, deadlocks are inherently avoided. Without this lock, concurrent operations could alter the number of objects in the world model during execution, potentially leading to Python exceptions and compromising data integrity.

Object Management Workflow

When a new object is published on the `/objects` topic, a callback function to manage that instance is triggered. The object management workflow is executed as outlined in the following algorithm.

Algorithm 2: Object Management Workflow

Input: New object data `obj_data`

Output: Updated world state

```
obj_data ← Receive data from Mapping Node via /objects topic;
new_obj ← Create Obj instance from obj_data;
existing_obj ← Query world for object matching new_obj;
if existing_obj exists then
    Update existing_obj with data from new_obj;
end
else
    Add new_obj to world;
    Rebuild world's KDTree to reflect updated state;
end
```

Where, the retrieval of the existing object is a crucial step.
Here object re-identification is performed with the following heuristic:

Algorithm 3: Re-identification

Input: `new_obj`

Output: `obj_id` if found else `new_obj.id`

```
centroid ← Compute new_obj centroid;
close_objects ← Query objects within a distance threshold of centroid;
foreach close_obj in close_objects do
    score ← Compute 3D Dice score between new_obj and close_obj;
    if new_obj.label = close_obj.label and score > dice_thr then
        return close_obj.id;
    end
    if new_obj.label ≠ close_obj.label and score > 2× dice_thr
     then
        return close_obj.id;
    end
end
return new_obj.id;
```

In the detection process, it is possible for objects to be misclassified. Therefore, it is crucial to provide the system with the opportunity to match objects even when their labels differ. To address this, a more stringent overlap threshold is applied for objects

with different labels. Specifically, the Dice similarity coefficient is used to quantify overlap as explained in section 4.3, and in cases where labels differ, the threshold is set to twice the normal value to ensure robustness in distinguishing separate objects.

For instance, consider two close objects, such as a laptop and a cup, whose point clouds may slightly overlap due to sensor noise. By enforcing a more stringent threshold for these cases, we effectively prevent erroneous merging of distinct objects. Conversely, if an object has been misclassified but is otherwise similar to a correctly labeled detection, the stricter threshold still allows the objects to be correctly identified as the same.

In this implementation, the baseline Dice threshold (`dice_thr`) is set to a low value of 0.2 since most of the time we are comparing a partial cloud with a complete and reconstructed one. When objects have matching labels, the threshold allows for sufficient overlap to confirm identification, while the doubled threshold for differing labels prevents misalignment between distinct entities. The final label for a merged object is determined by selecting the label with the highest prediction confidence score, ensuring that the most likely classification is assigned.

However, it is important to note that this heuristic may not perform optimally in situations where two objects with the same label are in close proximity but we can also consider that, for close and non overlapping objects, achieving an overlap of 0.2 is rare. This limitation may result in occasional difficulties in confirming the identity of closely positioned objects.

Cleaning and Maintenance

Since the segmentation model, in certain sporadic frames, sees objects that are not really there, we put a `is_confirmed` boolean flag on each object. An object is confirmed if it has been detected for more than N frames withing a certain amount of time, otherwise will be discarded. The World Manager spins a thread that periodically cleans up the world model to maintain accuracy and efficiency. The clean-up process involves:

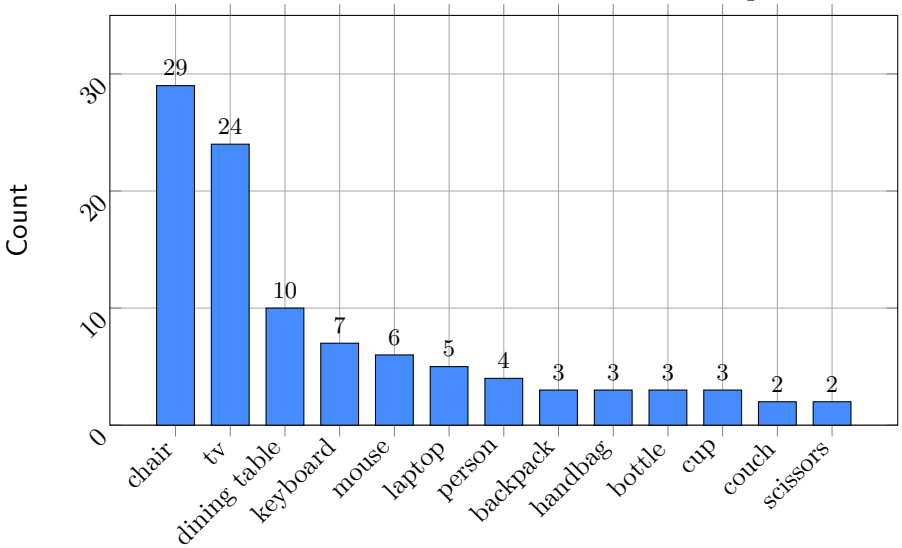
- Removing non confirmed or dynamic objects that have not been seen for a certain time threshold.
- Managing resources by releasing memory occupied by outdated or redundant objects.

Chapter 6

Custom Dataset Creation

This chapter outlines the process of creating a custom dataset specifically designed to evaluate the real-time 3D semantic mapping system. Collected in a controlled indoor environment, the dataset comprises 101 objects across various categories, including chairs, TVs, tables, and smaller items like keyboards and cups. Although the dataset distribution is skewed,with common objects such as "chair" and "tv" represented more frequently,this composition mirrors real-world indoor environments.

Due to the need for precise annotation, dataset collection was a time-intensive process, with each object carefully labeled from multiple perspectives to establish a reliable ground truth. Despite the dataset's imbalanced nature, using a pre-trained YOLO model mitigates potential issues. YOLO's extensive training on large datasets enables effective generalization across object categories, allowing for consistent detection and re-identification of both frequent and rare objects.



6.1 Data Collection

The dataset comprises seven distinct environments point clouds captured within an indoor lab setting, where objects such as chairs, laptops, televisions, mugs, bottles, couches, and handbags were randomly distributed. To collect the point clouds, the robot was manually navigated through the environment, with attention given to recording all visible objects from multiple viewpoints.

During the data collection process, the RTAB-MAP[4] tool, integrated with ROS, was employed to generate real-time point cloud reconstructions of each environment. Simultaneously, the developed 3D semantic mapping system was actively detecting and mapping objects within the environment. This approach allowed for the concurrent generation of both raw point clouds and corresponding object predictions.

6.2 Evaluation of Mapping Accuracy

Accurate environment reconstruction is pivotal for assessing the performance of the real-time 3D semantic mapping system. This section describes the methodologies employed to evaluate mapping accuracy, leveraging the MiR200 robot's localization system as ground truth (GT) and analyzing the performance of RTAB-MAP [4] when configured to use only visual odometry. By using visual odometry alone, we effectively reduce the correlation between the RTAB-MAP reconstruction and the mapping system's predictions, allowing for a more unbiased evaluation of the system's performance.

6.2.1 Ground Truth Verification with Robot Odometry

To establish a reliable ground truth for evaluation, we utilized the MiR200 robot's built-in localization system as the reference trajectory. The MiR200 combines data from multiple sensors—including 3D front cameras (not the Realsense L515 1.2), a SICK 360° LiDAR scanner, wheel odometry, and ultrasonic sensors—to achieve high positional accuracy, specified by the manufacturer to be approximately ± 10 *mm*.

In contrast, RTAB-MAP was configured to use only visual odometry from the Realsense L515 camera without incorporating the robot's odometry data. This con-

figuration decouples the RTAB-MAP reconstruction from the robot's localization system, reducing potential biases in the evaluation of the mapping system's predictions.

We computed the Absolute Trajectory Error (ATE) and Relative Pose Error (RPE) between the robot's odometry (serving as ground truth) and the RTAB-Map's visual odometry estimates using the TUM Benchmark Suite [11]. The results are presented in Tables 6.2 and 6.1.

Table 6.1: Relative Pose Error (RPE) between robot odometry and RTAB-MAP visual odometry estimates

Metric	**Value (m)**
RMSE	0.030850
Mean	0.028143
Median	0.025616
Standard Deviation	0.012637
Minimum	0.001583
Maximum	0.067017

Table 6.2: Absolute Trajectory Error (ATE) between robot odometry and RTAB-Map visual odometry estimates

Metric	**Value (m / deg)**
Translational RMSE	0.051299 m
Translational Mean	0.040192 m
Translational Median	0.032806 m
Translational Std Dev	0.031878 m
Translational Min	0.000086 m
Translational Max	0.173246 m
Rotational RMSE	2.754485 deg
Rotational Mean	2.152562 deg
Rotational Median	1.748910 deg
Rotational Std Dev	1.718623 deg
Rotational Min	0.002152 deg
Rotational Max	10.155773 deg

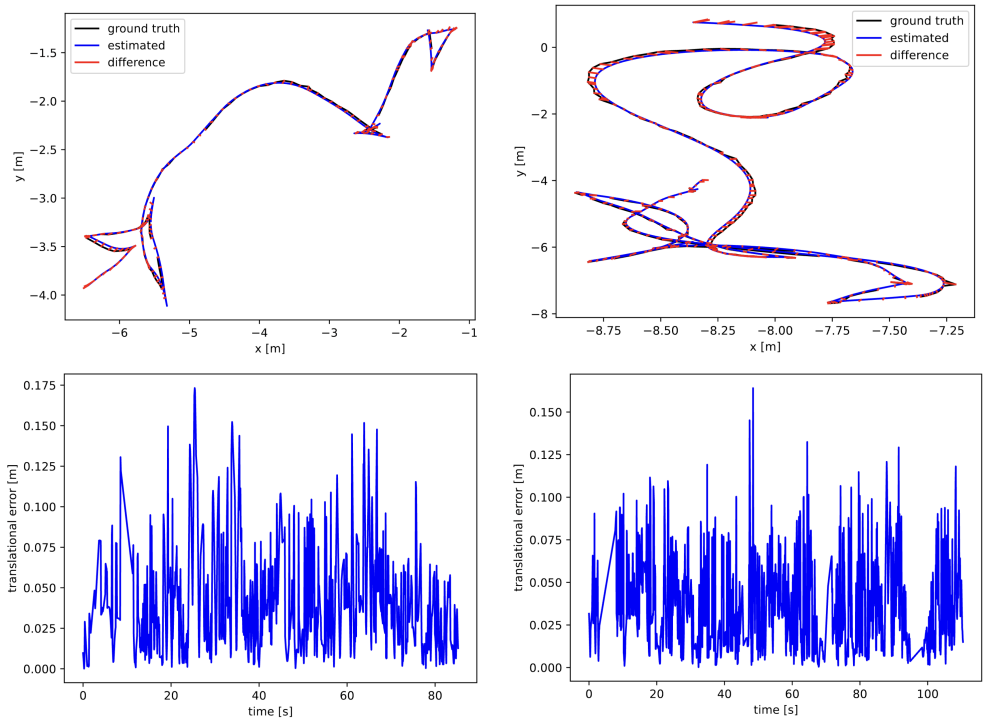


Figure 6.1: ATE and RPE visualization of 2 out of 7 reconstructed scenes

By decoupling the RTAB-MAP reconstruction from the robot's odometry, we ensure that the ground truth map used for evaluation is not directly influenced by the same positional data sources as the mapping system's predictions.

6.2.2 Comparison of Reconstructed Map with 2D LiDAR Scans

To further validate the accuracy of the reconstructed 3D maps generated using visual odometry, we compared them with 2D LiDAR scans from the MiR200's SICK 360° laser scanner. The 2D LiDAR scans offer precise distance measurements at a fixed height of 20 cm above ground level, serving as an independent ground truth reference.

We sliced the reconstructed 3D point cloud at the 20 cm height with a slack of 2 cm to obtain a 2D cross-sectional map and overlaid the LiDAR scans collected at multiple timestamps. Figure 6.2 illustrates on the left the alignment between the reconstructed map and LiDAR scans, on the right the alignment between the sliced 3D map (in green) and the aggregated LiDAR scans (in red).

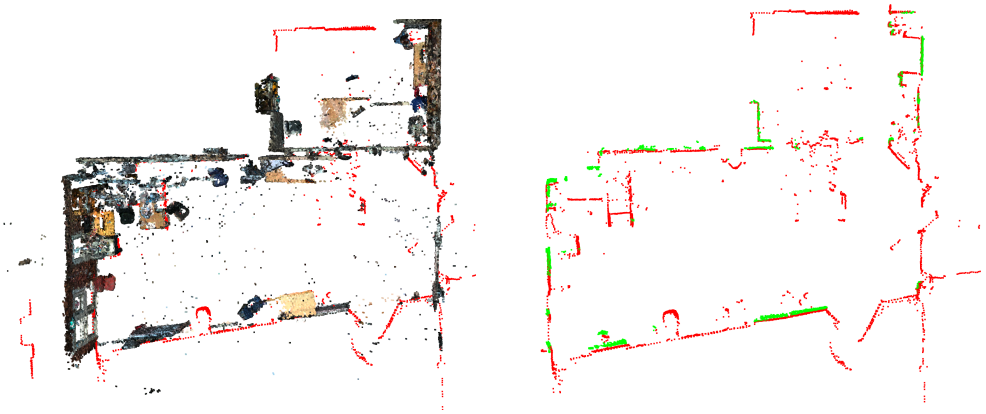


Figure 6.2: Reconstructed map and superimposed LiDAR scans alignment.

By computing the nearest-neighbor distances between the points in the sliced map and the LiDAR scans, we found a median distance of 0.0057 meters. This close alignment indicates that the reconstructed map is accurate enough for evaluating the mapping system.

6.3 Data Annotation

Post-collection, each environment's point cloud was manually annotated using the **labelCloud**[10][9] tool. This tool facilitated the precise labeling of objects within the 3D point clouds, enabling the creation of accurate ground truth annotations for subsequent evaluation. Labels were assigned to objects based on their visual and spatial characteristics, with each object class corresponding to a unique identifier.

The annotation process involved detailed inspection of each point cloud, with bounding boxes carefully drawn around each object to ensure precise alignment with the object boundaries. Furthermore, labels are also added to objects with partially reconstructed point clouds, as shown in figure 6.3, because we should still be able to detect them. This manual annotation step was crucial for establishing a robust ground truth against which the system's predictions could be rigorously assessed.

6.4 Data Structure and Organization

The dataset is organized as follows:

- **Point Cloud Files**: Each environment's point cloud is stored in PLY, a standard format compatible with popular 3D processing tools, facilitating easy

integration with ROS and other visualization tools.

- **Annotation Files**: For each point cloud, corresponding annotation files are provided, containing the 3D oriented bounding box coordinates and object labels.

- **System Predictions**: The final objects representation, generated by the 3D semantic mapping system during data collection, is saved separately for each environment, enabling direct comparison with the ground truth annotations.

6.5 Sample Annotated Environments

To provide a visual representation of the annotated dataset, Figures 6.3 and 6.4 presents sample images of the labeled environments. These images demonstrate the diversity of objects within the dataset and the accuracy of the manual annotations.

Figure 6.3: Example of annotated environment.

Figure 6.4: Sample annotated environments

Chapter 7

Results

The evaluation of the 3D semantic mapping system focuses on three main aspects: mean Average Precision (mAP), Mean Localization Precision (mLP), and execution time analysis. mAP assesses the system's ability to correctly detect and classify objects, providing a summary of precision and recall across multiple classes. Meanwhile, mLP evaluates the spatial accuracy of these detections by measuring the overlap between predicted and ground truth objects. Finally, an execution time analysis measures the system's real-time performance, crucial for applications where timely updates are essential. Together, these metrics offer a comprehensive view of the system's effectiveness, capturing both its classification accuracy and localization precision, as well as its suitability for real-time deployment.

7.1 Evaluation Methodology

To assess the accuracy of object positioning and labeling in the developed semantic mapping system, we establish a method for matching predicted objects with ground truth annotations. Instead of relying solely on bounding box overlaps, we use point cloud overlaps, which provide a more granular and precise measure of similarity between objects (see Section 4.3).

For each annotated oriented bounding box in the dataset, we extract the enclosed point cloud using Open3D [16]. This results in a set of ground truth point clouds $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$ and a set of predicted point clouds $\mathcal{P} = \{P_1, P_2, \ldots, P_M\}$, where N is the number of ground truth objects and M is the number of predicted objects.

7.1.1 Matching Predicted Objects with Ground Truth

The matching process involves computing a similarity score between each pair of ground truth and predicted point clouds using the Dice coefficient (see Section 4.3.2). We construct a similarity matrix $\mathbf{D} \in \mathbb{R}^{N \times M}$, where each element D_{ij} represents the Dice coefficient between ground truth point cloud G_i and predicted point cloud P_j:

$$\mathbf{D} = \begin{bmatrix} \text{Dice}(G_1, P_1) & \text{Dice}(G_1, P_2) & \dots & \text{Dice}(G_1, P_M) \\ \text{Dice}(G_2, P_1) & \text{Dice}(G_2, P_2) & \dots & \text{Dice}(G_2, P_M) \\ \vdots & \vdots & \ddots & \vdots \\ \text{Dice}(G_N, P_1) & \text{Dice}(G_N, P_2) & \dots & \text{Dice}(G_N, P_M) \end{bmatrix}. \tag{7.1}$$

To determine the best matches between ground truth and predicted objects, we perform the following steps:

1. For each ground truth object G_i, identify the predicted object P_j with the highest Dice coefficient:

$$j^* = \arg\max_j D_{ij}. \tag{7.2}$$

2. If the maximum Dice coefficient D_{ij^*} exceeds a predefined threshold τ, we consider G_i and P_{j^*} as a matching pair:

$$\text{If } D_{ij^*} > \tau, \text{ then } (G_i, P_{j^*}) \text{ is a match.} \tag{7.3}$$

The choice of threshold τ affects the strictness of the matching criteria, influencing the subsequent evaluation metrics. It will be considered in the following evaluation.

7.1.2 Algorithm for Object Matching

The matching process is formalized in Algorithm 4.

Algorithm 4: Object Matching Based on Dice Coefficient

Input : Ground truth point clouds $\mathcal{G} = \{G_1, G_2, \ldots, G_N\}$;
Predicted point clouds $\mathcal{P} = \{P_1, P_2, \ldots, P_M\}$;
Threshold τ

Output: Matching pairs $\mathcal{M}$;
Unmatched predicted objects $\mathcal{U}_\mathcal{P}$;
Unmatched ground truth objects $\mathcal{U}_\mathcal{G}$

Initialize matching pairs $\mathcal{M} \leftarrow \emptyset$
for $i \leftarrow 1$ **to** N **do**
  for $j \leftarrow 1$ **to** M **do**
    Compute Dice coefficient $D_{ij} = \text{Dice}(G_i, P_j)$
  end
  Find $j^* = \arg\max_j D_{ij}$
  if $D_{ij^*} > \tau$ **then**
    $\mathcal{M} \leftarrow \mathcal{M} \cup \{(G_i, P_{j^*})\}$
  end
end
$\mathcal{U}_\mathcal{P} \leftarrow \{P_j \in \mathcal{P} \mid P_j \text{ not in any matching pair}\}$
$\mathcal{U}_\mathcal{G} \leftarrow \{G_i \in \mathcal{G} \mid G_i \text{ not in any matching pair}\}$
return $\mathcal{M}, \mathcal{U}_\mathcal{P}, \mathcal{U}_\mathcal{G}$

7.1.3 Interpretation of Results

The outcome of the matching process yields:

- **Matching Pairs** ($\mathcal{M}$): Ground truth and predicted object pairs with a Dice coefficient exceeding the threshold τ, indicating potential correct detections.
- **Unmatched Predicted Objects** ($\mathcal{U}_\mathcal{P}$): Predicted objects that do not match any ground truth object, which may be considered false positives in further analysis.
- **Unmatched Ground Truth Objects** ($\mathcal{U}_\mathcal{G}$): Ground truth objects without corresponding predicted objects, which may be considered false negatives in further analysis.

These results provide the basis for calculating evaluation metrics by categorizing predictions and ground truths into true positives, false positives, and false negatives, as defined in the next section.

7.2 Computation of Mean Average Precision (mAP)

The mean Average Precision (mAP) metric is widely used in object detection to assess both precision and recall across multiple classes and varying confidence thresholds. By summarizing the precision-recall curve into a single value, mAP provides a comprehensive view of the system's ability to accurately detect and classify objects. High mAP values indicate that the system is effectively balancing precision—minimizing false positives—with recall—minimizing false negatives. In the context of this 3D semantic mapping system, mAP evaluates how well the system recognizes, locates and labels objects correctly, which is crucial for reliable environment understanding in robotic navigation.

7.2.1 Determining True Positives, False Positives, and False Negatives

Before computing the Average Precision, it is essential to determine the true positives (TP), false positives (FP), and false negatives (FN) for each class:

1. **True Positives (TP):** A predicted object is considered a true positive for class c if:
   - It matches a ground truth object (as determined by the matching process with Dice coefficient exceeding τ).
   - The predicted label matches the ground truth label (both are class c).
2. **False Positives (FP):** A predicted object is considered a false positive for class c if:
   - It does not match any ground truth object (from $\mathcal{U}_{\mathcal{P}}$) and is labeled as class c.
   - It matches a ground truth object but the predicted label is class c while the ground truth label is not class c (misclassification).
3. **False Negatives (FN):** A ground truth object of class c is considered a false negative if:
   - It has no corresponding predicted object (from $\mathcal{U}_{\mathcal{G}}$).
   - It matches a predicted object but the predicted label is not class c (misclassification).

By accurately identifying TP, FP, and FN, we can compute the precision and recall values necessary for calculating the Average Precision.

7.2.2 Computing Average Precision (AP) for Each Class

The Average Precision (AP) for each class c is computed using the precision-recall curve derived from the TP and FP determinations. The steps are as follows:

1. **Collect Predictions for Class c:** Gather all predicted objects labeled as class c, including both matched and unmatched predictions.
2. **Label Predictions as TP or FP:**
   - Label a prediction as TP if it correctly matches a ground truth object of class c.
   - Label a prediction as FP if it does not match any ground truth or if it matches a ground truth of a different class.
3. **Sort Predictions by Confidence Score:** Arrange the predictions in descending order of their confidence scores.
4. **Compute Cumulative Sums:** Calculate the cumulative sums of TP and FP. The cumulative sums are used to compute precision and recall at each threshold defined by the confidence scores.
5. **Compute Precision and Recall for each confidence threshold:**

$$\text{Precisions} = \frac{\text{Cumulative TP}}{\text{Cumulative TP} + \text{Cumulative FP}}, \tag{7.4}$$

$$\text{Recalls} = \frac{\text{Cumulative TP}}{\text{Total Ground Truths of Class } c}. \tag{7.5}$$

   where the last values of these vectors correspond to the overall precision and recall of the system.
6. **Adjust Precision Curve:** Modify the precision values to ensure the precision-recall curve is non-increasing. This is done by iterating from the highest recall to the lowest and replacing each precision value with the maximum of itself and the following precision value.
7. **Compute Average Precision:** Calculate the area under the precision-recall curve using the trapezoidal rule or by summing the products of changes in recall and precision at each step.

We use cumulative sums in this process to account for all predictions up to each confidence threshold, allowing us to compute precision and recall at varying levels of confidence. By evaluating the model across all confidence thresholds, we obtain a comprehensive precision-recall curve.

7.2.3 Algorithm for Computing Average Precision

The computation of Average Precision for class c is formalized in Algorithm 5.

Algorithm 5: Compute Average Precision (AP) for Class c

Input :

- List of predicted objects for class c, each with a confidence score and TP/FP label.
- Total number of ground truth objects of class c, N_{gt}.

Output: Average Precision AP_c

if $N_{gt} = 0$ **then**
  return $\text{AP}_c \leftarrow 0$
end
Sort the predicted objects by descending confidence score
Compute TP labels array TP $\leftarrow$ indicator function for true positives
Compute FP labels array FP $\leftarrow$ indicator function for false positives
`// Compute cumulative sums of TP and FP:`
cum_TP $\leftarrow$ cumulative sum of TP
cum_FP $\leftarrow$ cumulative sum of FP
`// Compute precision and recall at each confidence threshold:`
Compute precision P $\leftarrow$ cum_TP/(cum_TP + cum_FP)
Compute recall R $\leftarrow$ cum_TP/N_{gt}
`// Adjust precision to be non-increasing`
for $i \leftarrow$ *length of* $P - 1$ **to** 1 **do**
  $\text{P}[i-1] \leftarrow \max(\text{P}[i-1], \text{P}[i])$
end
`// Compute Average Precision as area under the curve`
Compute $\text{AP}_c \leftarrow \sum_{i=1}^{n} (\text{R}[i] - \text{R}[i-1]) \times \text{P}[i]$
return AP_c

7.2.4 Computing Mean Average Precision (mAP)

The mean Average Precision (mAP) is calculated by averaging the AP values over all object classes:

Algorithm 6: Compute Mean Average Precision (mAP)

Input :

- Set of object classes $\mathcal{C}$.
- For each class $c \in \mathcal{C}$:
  - List of predicted objects for class c, each with a confidence score and TP/FP label.
  - Total number of ground truth objects of class c, N_{gt}.

Output: Mean Average Precision mAP

```
Initialize list of AP values: AP_list ← [ ]
foreach class c ∈ C do
    Compute AP_c using Algorithm 5
    Append AP_c to AP_list
end
Compute mAP ← mean of values in AP_list
return mAP
```

7.2.5 Mean Localization Precision

While mAP provides an overview of classification performance, it does not reflect the precision of object localization. To address this, we introduce Mean Localization Precision (mLP), which quantifies the spatial accuracy of matched objects. The mLP is defined as follows:

$$\text{mLP} = \frac{1}{n} \sum_{(G_i, P_j) \in \mathcal{M}} \text{Dice}(G_i, P_j)$$

where $\mathcal{M}$ represents the set of matched pairs, G_i and P_j are the ground truth and predicted objects, respectively, n is the total number of matches, and $\text{Dice}(G_i, P_j)$ is the overlap metric as defined in Section 4.3.2.

This metric provides a focused assessment of object localization, directly evaluating how well the system positions objects relative to the ground truth.

7.3 Results and Discussion

We now present the evaluation results, focusing on Mean Average Precision (mAP) across different Dice coefficient thresholds, as well as the Mean Localization Precision (mLP).

7.3.1 Mean Average Precision at Various Dice Thresholds

To assess system capabilities, we calculate mAP by evaluating $\mathcal{M}$ (Matching Pairs), $\mathcal{U}_{\mathcal{P}}$ (Unmatched Predictions), and $\mathcal{U}_{\mathcal{G}}$ (Unmatched Ground Truths) sets across different overlap thresholds (as explained in section7.1.1). This gives us insight into the system's precision and recall under varying levels of strictness in object matching. As the threshold increases, the number of matching pairs decreases, indicating that higher overlap requirements make it more challenging to identify correct matches, thus highlighting the system's sensitivity to positional accuracy.
Figure 7.3.1 shows that the system achieves an mAP of 0.8, with performance remaining stable up to a 0.4 overlap. Beyond this point, stricter constraints reduce mAP.

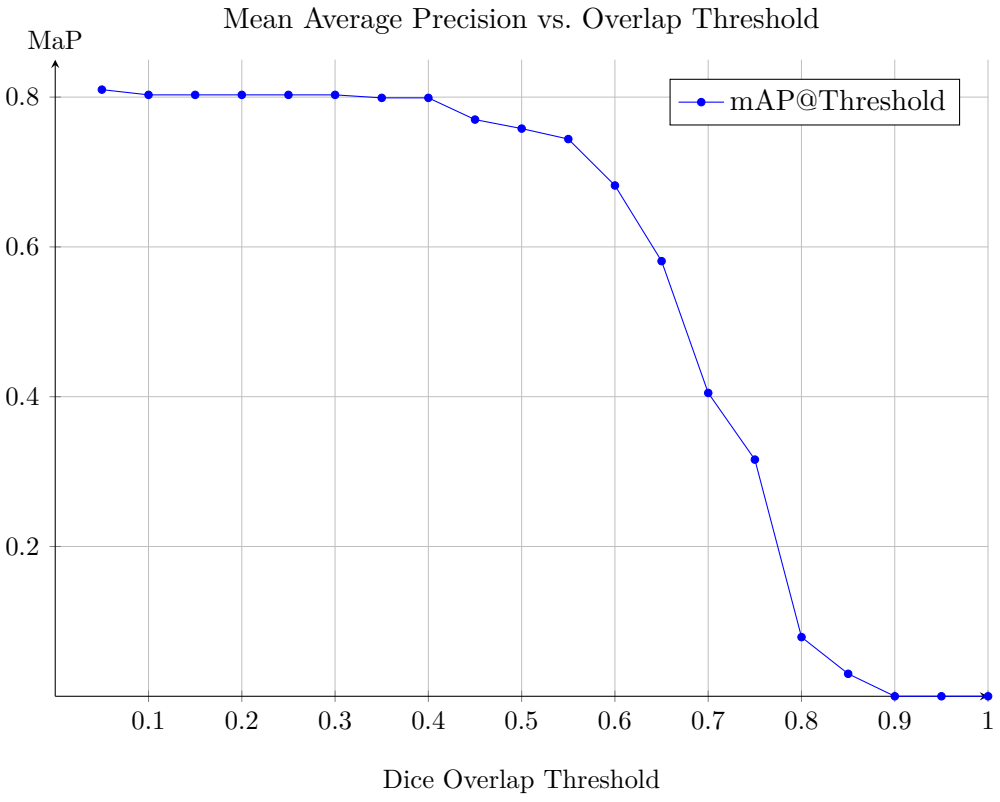


Figure 7.1: Mean Average Precision evaluated at various overlap thresholds.

7.3.2 Precision-Recall-F1 Analysis

This paragraph presents the macro-averaged precision, recall, and F1 scores across different overlap thresholds, as shown in Figure 7.3.2. These metrics differ from the mean Average Precision (mAP), which is the area under the precision-recall curve; here, we use the standard macro-averaging method to provide a comprehensive evaluation of detection performance across categories without integrating over recall.

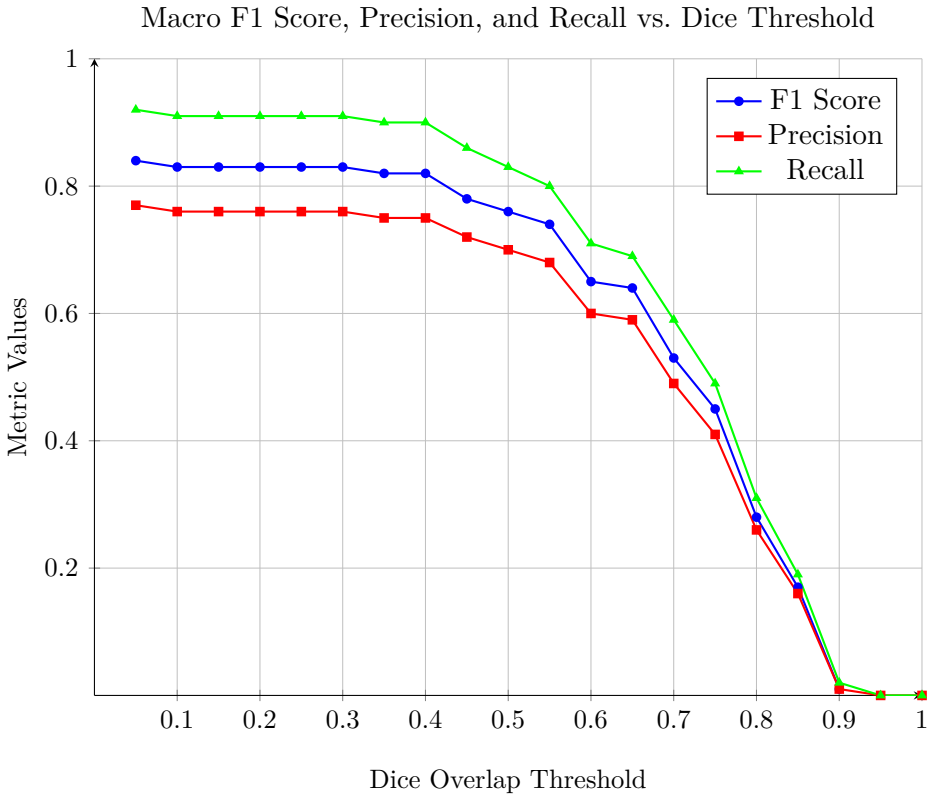


Figure 7.2: F1 Score, Precision, and Recall evaluated at various Dice thresholds.

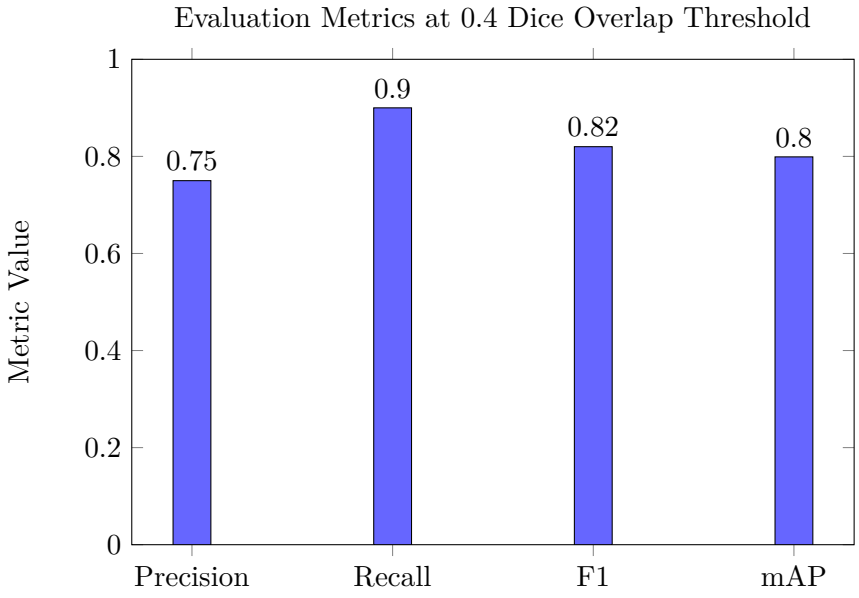
High precision at lower thresholds indicates that many predictions are correct. As thresholds increase, both precision and recall decline, revealing challenges in achieving higher overlaps with ground truth.

7.3.3 Mean Localization Precision Analysis

The system's mLP score of 0.69 demonstrates good localization and reconstruction precision.

7.3.4 Discussion

From the results, we identify 0.4 as an optimal Dice overlap threshold. This threshold offers the best balance of precision, recall, and mAP while keeping a strict constraint for objects overlap.
Figure 7.3.4 summarizes system performance at this threshold.



The evaluation confirms that the system performs well, especially at moderate overlap thresholds. High mAP values up to a 0.4 threshold suggest that the system effectively detects and classifies objects with acceptable spatial overlap.

The mLP of 0.69 indicates satisfactory alignment between predicted and ground truth objects, despite sensor noise.

The decrease in performance at higher thresholds underscores the difficulty of precise alignment due to sensor noise, environmental factors, and algorithmic limitations. Precision-recall trends further illustrate the trade-off between precision and recall as matching criteria tighten.

7.4 System Performance Analysis - Execution Time

To evaluate the performance of the proposed 3D semantic mapping system, we measured the execution times of the nodes' main functions: The Detection node callback,

responsible for performing instance segmentation and tracking, has a mean execution time of 0.009 seconds, making it well-suited to handle the 30 FPS input stream from the camera. The Mapping node's main callback, responsible for merging depth and segmentation data into object instances, has a mean execution time of 0.01 seconds. The World node's object-handling function, which manages detected objects, has a mean execution time of 0.044 seconds. Other functions, such as cleaning and removing objects, are executed periodically and have execution times in the order of milliseconds, making their impact on the overall performance negligible.

Since each ROS node runs asynchronously on a separate thread, computation is distributed, and only the functions directly affecting the Mapping Node's and World Node's operation impact the frame rate of the maps updates. Furthermore, the Detection and Mapping nodes' main functions can process data faster than required, so they do not represent bottlenecks in the system. The key bottleneck arises from the World node's object-handling function, which is called n times per frame, where n is the number of detected objects in that frame. This leads to a linear increase in processing time with the number of objects, making it the primary performance-limiting factor. To prevent the system from becoming overwhelmed, ROS topics are implemented with a queue that only retains the last five submissions, allowing the system to focus on more up-to-date objects and avoid getting stuck if processing falls behind.

As the system employs an asynchronous pub-sub mechanism with separate threads for each node, only the World node's object-handling function determines the effective frame rate. The per-frame processing time, T_{frame}, can thus be calculated as follows:

$$T_{\text{frame}} = n \times T_{\text{Obj\_callback}}, \tag{7.6}$$

where $T_{\text{Obj\_callback}}$ is the execution time of the World node's object-handling function.

The following frame rates were computed based on typical values of n between 1 and 5:

- $n = 1$: $T_{\text{frame}} = 0.044$ s, yielding an FPS of approximately 22.73.
- $n = 2$: $T_{\text{frame}} = 0.088$ s, yielding an FPS of approximately 11.36.
- $n = 3$: $T_{\text{frame}} = 0.132$ s, yielding an FPS of approximately 7.58.

- $n = 4$: $T_{\text{frame}} = 0.176$ s, yielding an FPS of approximately 5.68.

- $n = 5$: $T_{\text{frame}} = 0.220$ s, yielding an FPS of approximately 4.55.

These results were measured using the computational resources of the Heron platform, as outlined in Section 1.3. Tests were conducted across multiple recordings, each involving various numbers of objects detected simultaneously. The FPS values reflect an average across these scenarios, representing cases where up to five objects are detected per frame.

It is important to note that these FPS calculations assume the presence of objects in every frame, which serves as a conservative estimate of the system's performance. In practice, the system often encounters frames with fewer objects—frequently detecting no objects or just one or two. This allows the system to achieve higher frame rates during those instances, ensuring that it operates in real time under the tested conditions. Therefore, while the maximum object count of five presents a challenge to frame rate, the system generally maintains real-time processing in the tested environments.

7.5 Qualitative Results

This section presents qualitative examples illustrating the final 3D semantic map generated by the system and the effectiveness of the object re-identification process.

Figure 7.1 shows the initial detection and annotations, focusing on the "TV" object, which is initially assigned ID 5 by ByteTrack. When this object leaves the camera's field of view (FOV) and later reappears, YOLOv8 detects it and ByteTrack assign a new ID (22). With the re-identification algorithm, explained in section 5.4.2, the system is able to update the already stored "TV" object with information coming from the new detection, as shown in Figure 7.2.

In addition, Figure 7.2 highlights a temporary misclassification by YOLOv8, where a "couch" is erroneously labeled as "chair." To counter such misclassifications, the system applies a label consistency mechanism, assigning each object the label most frequently observed over time. This approach ensures stability and reduces mislabeling within the 3D semantic map, reinforcing the system's robustness.

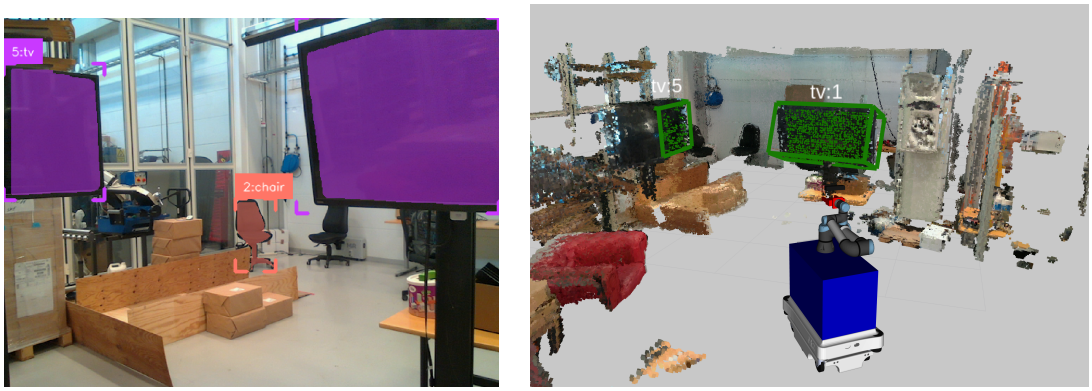


Table 7.1: Initial object identification. Left: YOLOv8 detection showing the initial identification of the TV. Right: Corresponding entry in the world model before re-identification.

Table 7.2: Object re-identification and update. Left: YOLOv8 detection upon re-identification. Right: Updated world model entry after re-identification, reflecting the latest information.

Figures 7.3 and 7.4 present snapshots of the final 3D semantic map generated after the robot's exploration of two different environments. These maps reflect accurate spatial information and robust object representation achieved through incremental updates and effective re-identification.

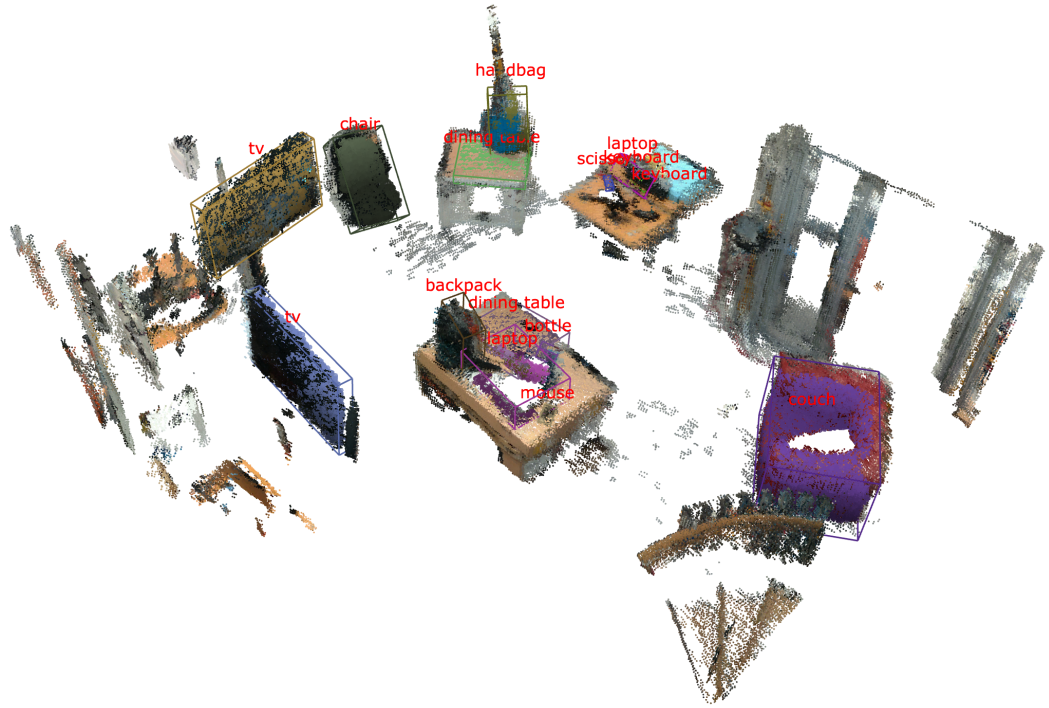


Figure 7.3: Reconstructed 3D semantic map

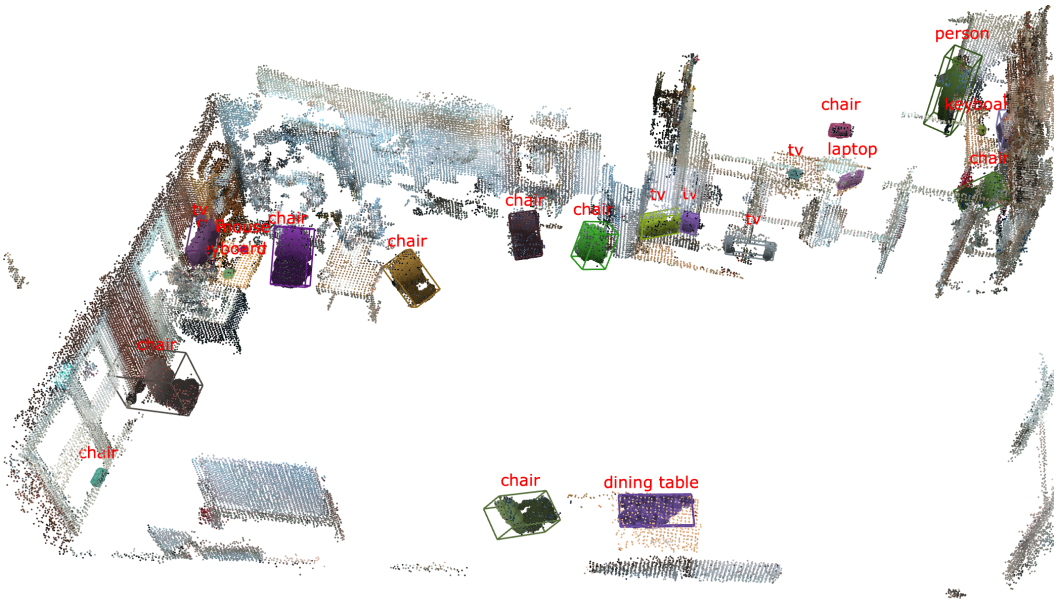


Figure 7.4: Reconstructed 3D semantic map

Chapter 8

Conclusion

In this thesis, we have developed and evaluated a real-time 3D semantic mapping system designed for autonomous robotic navigation in indoor environments. The proposed system effectively integrates multimodal data from RGB and depth images to construct detailed and accurate 3D maps that provide robots with the necessary understanding of their surroundings for efficient navigation and object identification.

The system leverages state-of-the-art object detection models, specifically YOLOv8 for instance segmentation, and utilizes KD-Trees for fast spatial queries and object re-identification. This combination allowed for efficient updating of object representations over time, preventing redundant data and improving both memory usage and time complexity. The voxel downsampling and noise removal techniques, such as statistical and radius outlier removal, further enhanced the system's ability to maintain a clean and manageable point cloud while preserving the accuracy of the mapped environment.

The system was tested in a controlled indoor lab using a custom dataset collected with a robot equipped with an Intel RealSense camera. The evaluation metrics, particularly the mean Average Precision (mAP) and Dice coefficient, demonstrated the effectiveness of the proposed approach. The system achieved a peak mAP of 0.800 at a 40% overlap threshold and consistently performed well across various object categories. The system's real-time performance, with a throughput of 11 frames per second with an average of 2 objects per frame, makes it suitable for practical deployment in real-world scenarios requiring rapid updates to 3D maps.

However, some challenges and limitations remain. The system's assumption of static

environments means it struggles with not pre-determined moving objects. Additionally, while the noise reduction techniques were effective, sensor inaccuracies, especially in depth images, continue to introduce errors in object reconstruction and re-identification. Future improvements could focus on extending the system's capability to handle dynamic environments, integrating better noise filtering methods, and incorporating more advanced object tracking algorithms to maintain object representations over time more effectively, especially with very close objects belonging to the same category.

In conclusion, this research demonstrates the potential of multimodal data fusion and real-time semantic mapping to improve the accuracy and efficiency of 3D mapping systems in robotic applications. The results suggest that this approach can significantly enhance robotic navigation in complex indoor environments, providing a strong foundation for further work in this domain.

8.1 Future Work

Future research can build upon this work in several key areas. One promising direction is the implementation of open-category models where objects are not merely classified into predefined categories, but are instead represented by semantic vectors. These vectors would enable the system to recognize novel objects and utilize their semantic meaning for more flexible and intelligent navigation. However, current open-category models are computationally expensive and not suitable for real-time applications, making their integration into real-time systems an open challenge.

Moreover, improving the system's ability to handle dynamic environments with moving objects, as well as enhancing object tracking accuracy for closely positioned objects belonging to the same category, remains an important focus. The development of more advanced noise filtering and point cloud processing techniques would further improve the reliability of object re-identification and mapping, especially in cluttered or noisy settings.

Bibliography

[1] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

[2] IntelRealSense. Intelrealsense/librealsense: Intel® realsense™ sdk. Available At: `https://github.com/IntelRealSense/librealsense`.

[3] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics YOLO, 2023.

[4] Mathieu Labbé and François Michaud. Rtab-map as an open-source lidar and visual simultaneous localization and mapping library for large-scale and long-term online operation. *Journal of Field Robotics*, 36(2):416–446, October 2018.

[5] Xiang Li, Wenhai Wang, Lijun Wu, Shuo Chen, Xiaolin Hu, Jun Li, Jinhui Tang, and Jian Yang. Generalized focal loss: Learning qualified and distributed bounding boxes for dense object detection, 2020.

[6] Tsung-Yi Lin, Michael Maire, Serge J. Belongie, Lubomir D. Bourdev, Ross B. Girshick, James Hays, Pietro Perona, Deva Ramanan, Piotr Doll'a r, and C. Lawrence Zitnick. Microsoft COCO: common objects in context. *CoRR*, abs/1405.0312, 2014.

[7] Morgan Quigley, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. Ros: an open-source robot operating system. In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*, Kobe, Japan, May 2009.

[8] Roboflow. Supervision. Available at `https://roboflow.github.io/supervision`.

[9] Christoph Sager, Patrick Zschech, and Niklas Kuhl. labelCloud: A lightweight labeling tool for domain-agnostic 3d object detection in point clouds. *Computer-Aided Design and Applications*, 19(6):1191–1206, mar 2022.

[10] Christoph Sager, Patrick Zschech, and Niklas Kühl. labelcloud: A lightweight domain-independent labeling tool for 3d object detection in point clouds, 2021.

[11] J. Sturm, N. Engelhard, F. Endres, W. Burgard, and D. Cremers. A benchmark for the evaluation of rgb-d slam systems. In *Proc. of the International Conference on Intelligent Robot Systems (IROS)*, Oct. 2012.

[12] Ultralytics. Yolov8 loss functions. Available at `https://github.com/ultralytics/ultralytics/blob/main/ultralytics/utils/loss.py`.

[13] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[14] Yifu Zhang, Peize Sun, Yi Jiang, Dongdong Yu, Zehuan Yuan, Ping Luo, Wenyu Liu, and Xinggang Wang. Bytetrack: Multi-object tracking by associating every detection box. *CoRR*, abs/2110.06864, 2021.

[15] Zhaohui Zheng, Ping Wang, Wei Liu, Jinze Li, Rongguang Ye, and Dongwei Ren. Distance-iou loss: Faster and better learning for bounding box regression. *Proceedings of the AAAI Conference on Artificial Intelligence*, 34(07):12993–13000, Apr. 2020.

[16] Qian-Yi Zhou, Jaesik Park, and Vladlen Koltun. Open3D: A modern library for 3D data processing. *arXiv:1801.09847*, 2018.