Easy Object-Conscious SLAM with YOLO and LIDAR

Project: Learning Robots

Esteban Padilla Cerdio EPC

March 31, 2024

Abstract

Simultaneous Localization and Mapping (SLAM) allows robots to understand where they currently are in relation to an environment, and at the same time create a map of their surroundings. Object-Conscious SLAM (OCSLAM) introduces an additional layer of knowledge by allowing the robot to locate and classify objects within this world. I propose a simple method that leverages LIDAR information already being used for regular SLAM, in combination with the You Only Look Once (YOLO) network for locating, classifying and tracking objects.

EPC

1 Introduction

Robots equipped with Simultaneous Localization and Mapping (SLAM) capabilities can navigate and map their surroundings. However, a robot that is to interact with its environment should have a syntactical understanding of the elements that compose it, instead of only physical. Integrating Object-Conscious SLAM (OCSLAM) enhances this by enabling object detection and classification in a three-dimensional world. This project aims to leverage existing LIDAR data used in SLAM alongside the You Only Look Once (YOLO) network for efficient object localization, classification, and tracking.

By combining these technologies, the robot gains the ability to not only understand its environment but also identify and categorize objects within it. This report sets the stage for the proposed method, which optimizes the use of minimal sensors to enhance robotic perception and interaction with its surroundings.

2 Method

The project revolves around creating an OCSLAM system with minimal and inexpensive components. These include a camera, a two-dimensional LIDAR, and a differential-drive chassis with encoders. The camera provides image information to a pre-trained YOLO network, which outputs a series of classified bounding boxes. The LIDAR outputs an array of distance readings separated by an angle increment. Meanwhile, the wheels' encoders output absolute position information that can be used to estimate the pose of the robot via odometry algorithms.

The LIDAR and odometry information is used by a SLAM node to generate the map of the environment. This node also uses feature matching to occasionally correct the pose estimation. Finally, the robot pose, the LIDAR information and the YOLO bounding boxes are consumed by a Consciousness node that uses projection matrices, point clustering and object tracking to generate a list of classified objects with their corresponding coordinates within the generated map.

Due to the node-based architecture of the method, which is illustrated in Figure 1, ROS2 is used as the operating system of the robot, with topics being used to share information between the nodes.

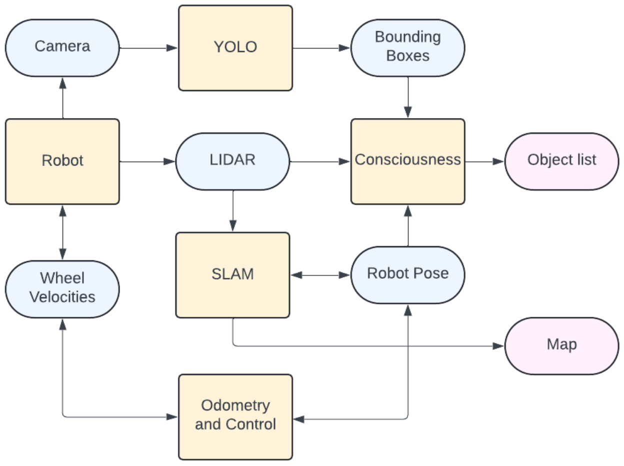

Figure 1: System Architecture

2.1 Webots and Turtlebot3

A commercially-available robot that meets the previously stated requirements is the Turtlebot3 by Open Robotics and ROBOTIS [4]. For this project, the Burger version is used. For ease of implementation and testing, I use a digital clone inside Cyberbotics' Webots [10] simulation environment.

To create a bridge between the Webots simulation and the ROS2 environment, the *webots_ros2_driver* plugin is implemented as described by [11]. This allows the creation of a ROS wrapper that can read the data from the simulated LIDAR and camera and output it as a ROS topic. Likewise, the C++ *webots* library is used to read the encoders' positional data directly, and to set each wheel to a specific velocity. Additionally, as part of this plugin, an URDF file describing the links and joints of the robot is required to be later used for visualization and odometry.

A second node, called *Robot*, receives the */cmd_vel* topic, which contains angular and linear velocity information, and translates it into Webots commands. This allows the use of controllers like Teleop Twist Keyboard [12] for manual control, or Nav2 [9] for autonomous control.

2.2 Odometry and control

Because this robot does not have an IMU, its pose can only be estimated from the velocity of the wheels. This is done using the classical equations described by [1]. Since the Turtlebot3's encoders provide data in the form of absolute position, I use the change in position for the right and left wheels, Δp_r and Δp_l respectively, in combination with the wheel radius r and the distance between the wheels l to obtain the linear v and angular ω velocities. Δt is calculated on each step, since it is dynamic and depends on the node's computational performance.

$$v = \frac{(\Delta p_r + \Delta p_l) r}{\Delta t \cdot 2}$$

$$\omega = \frac{(\Delta p_r - \Delta p_l) r}{\Delta t \cdot l}$$

With these velocities, I update the current heading θ_i and subsequently the current coordinates x_i and y_i.

$$\theta_i = \theta_{i-1} + \Delta t \cdot \omega$$

$$x_i = x_{i-1} + \Delta t \cdot v \cdot cos(\theta_i)$$

$$y_i = y_{i-1} + \Delta t \cdot v \cdot sin(\theta_i)$$

An inverse of the initial equations is used to transform the */cmd_vel* command into the desired right and left wheel angular velocities, which can then be sent back to the simulation to control the robot.

$$\omega_r = \frac{v + \omega \cdot l}{r}$$

$$\omega_l = \frac{v - \omega \cdot l}{r}$$

All this done by the same Robot node created earlier, which outputs the robot's pose on the */odom* topic, as well as a transform of the same name that is later used by the SLAM node to make the correlation between the robot and the map. The architecture so far is illustrated by Figure 2.

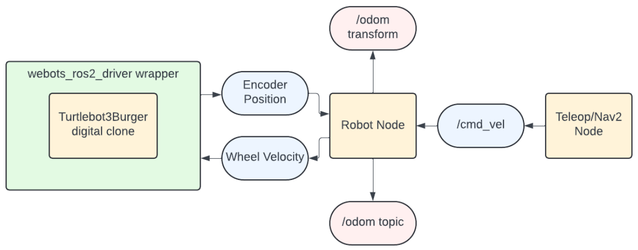

Figure 2: The Robot node

2.3 SLAM

As to not reinvent the wheel, I leverage Steve Mackenski's SLAM node found in the *slam_ toolbox* package [8]. This node consumes a valid transform that defines the position of the robot in relation to the world, as well as the raw LIDAR data produced by the robot, to generate the map and the estimated robot pose. By using feature matching, the node can also occasionally correct the pose estimation, which comes in handy when there is no IMU to support it. The final transform tree, which includes the links and joints defined in the first section, as well as the relationship between the LIDAR sensor and the base of the robot, looks like Figure 3.

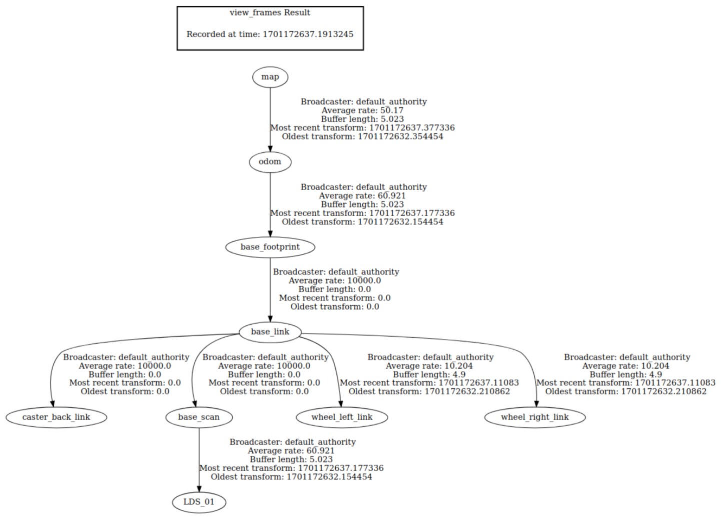
view_frames Result
Recorded at time: 1701172637.1913245
map
Broadcaster: default_authority Average rate: 50.17 Buffer length: 5.023 Most recent transform: 1701172637.377336 Oldest transform: 1701172632.354454
odom
Broadcaster: default_authority Average rate: 60.921 Buffer length: 5.023 Most recent transform: 1701172637.177336 Oldest transform: 1701172632.154454
base_footprint
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
base_link
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10000.0 Buffer length: 0.0 Most recent transform: 0.0 Oldest transform: 0.0
Broadcaster: default_authority Average rate: 10.204 Buffer length: 4.9 Most recent transform: 1701172637.11083 Oldest transform: 1701172632.210862
Broadcaster: default_authority Average rate: 10.204 Buffer length: 4.9 Most recent transform: 1701172637.11083 Oldest transform: 1701172632.210862
caster_back_link
base_scan
wheel_left_link
wheel_right_link
Broadcaster: default_authority Average rate: 60.921 Buffer length: 5.023 Most recent transform: 1701172637.177336 Oldest transform: 1701172632.154454
LDS_01

Figure 3: Transform Tree

2.4 YOLO

While the SLAM and Robot nodes are focusing on locating the robot and mapping the environment, the camera information is concurrently fed into the YOLO node, which uses Ultralytics' YoloV8 [7] network to locate and classify the objects within the frame. YOLO is a Convolutional Neural Network capable of identifying objects of pre-established classes within an image, and surround them with a bounding box. I chose it due to its superior speed and accuracy compared to similar models like Single Shot Detection, and the simplicity of the training required for model customization.

To improve the processing speed, the image is first scaled down using OpenCV 2 (CV2) [6] up to a point before it starts affecting performance, which is obtained experimentally. The image is then fed into the model, which has been previously trained with the desired custom classes. The model can also receive a customizable confidence threshold that allows only objects with a confidence score above it to be considered. Using CV2 once more, the bounding boxes are drawn over the original image, which is posted to the */yolo_image* topic. Finally, the boxes are published as a matrix over the */yolo_boxes* topic to be consumed by the Consciousness node as seen in Figures 1 and 4.

$$\begin{bmatrix} top\_left\_x & top\_left\_y & bottom\_right\_x & bottom\_right\_y & class\_index & confidence \cdot 100 \\ top\_left\_x & top\_left\_y & bottom\_right\_x & bottom\_right\_y & class\_index & confidence \cdot 100 \\ ... & ... & ... & ... & ... & ... \end{bmatrix}$$

Figure 4: The */yolo_boxes* matrix

2.5 Consciousness

All the previous information (the bounding boxes, LIDAR data and robot pose) is combined in the Consciousness node in order to, as accurately as possible, create a list of objects found in the world and assign them a class and position within the generated map. The node does this by identifying which points of the LIDAR cloud fall within a projection of each of the bounding boxes detected by the camera. From these points, clusters are extracted and the most probable cluster is chosen as an object. The object is then tracked across time with its position, size and confidence being updated on each frame.

2.5.1 LIDAR point cloud

The LIDAR sensor works by rotating around its axis, performing a distance measurement with a LASER after each angle step, starting from a specified angle. The sensor therefore sends the data in a ROS message that contains an array of measurements of size $N = \frac{2\pi}{angle\_step}$, the *angle_min*, and the *angle_step*. With this information, I obtain the x and y coordinates of each point at index i, relative to the center of the LIDAR.

$$x(i) = distances[i] \cdot cos(angle\_min + angle\_step \cdot i)$$

$$y(i) = distances[i] \cdot sin(angle\_min + angle\_step \cdot i)$$

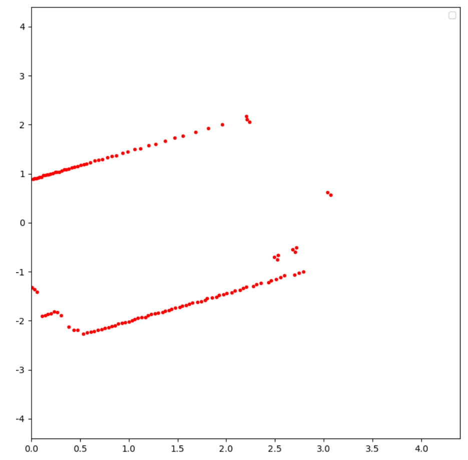

Figure 5: LIDAR Data in 2D coordinate system

2.5.2 Bounding area projection

To understand which of these points are being observed by the camera, and which fall specifically within the bounding boxes detected by the YOLO network, I use a projection from the camera's 2D coordinate system to the world's 3D system, with the equations described by [3].

$$\begin{pmatrix} X_{\text{world}} \\ Y_{\text{world}} \\ Z_{\text{world}} \\ 1 \end{pmatrix} = P^{-1} \begin{pmatrix} X_{\text{image}} \\ Y_{\text{image}} \\ 1 \end{pmatrix} \tag{1}$$

Where P is the projection matrix $K \begin{bmatrix} R & T \end{bmatrix}$, in which R and T are the rotation and translation matrices of the camera atop the robot and K is the intrinsic matrix of the camera, defined as

$$K = \begin{bmatrix} f_x & 0 & p_x \\ 0 & f_y & p_y \\ 0 & 0 & 1 \end{bmatrix}$$

Here, f_x and f_y are the focal lengths along the x and y axes respectively and p_x and p_y are the x and y coordinates of the principal point. These values are obtained through camera calibration.

These equations allow the definition of bounding areas on the LIDAR plane, that can tell the robot which points of the cloud fall within each of the YOLO boxes. I also use this to project the boundaries of the camera and eliminate all the points that lie outside the current view of the robot. Once obtained the bounding areas, a simple iteration over the point cloud can assign each point to one (or several) of the projections.

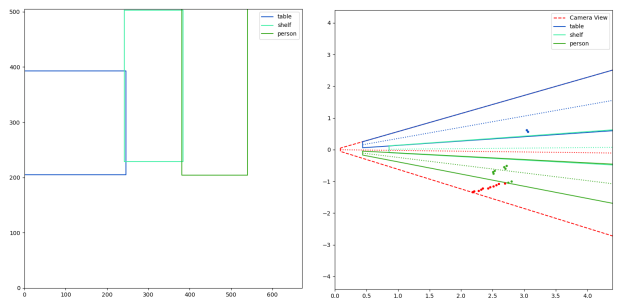

Figure 6: 2D yolo boxes (left). Camera and YOLO bounding area projections (right)

2.5.3 Object identification

The naive method for extracting the objects from the point cloud would be to simply perform some sort of clustering to detect islands and register each island as an object. Although this project does that as an initial step, namely using Shapely's [5] *buffered multipoint* method to group points separated by a customizable distance into *islands*, several edge cases are also taken into account.

The first case, as illustrated by Figures 7 and 8, is the fact that the bounding boxes are not *shrinkwrapped* around the object. This means that the edge of the box is squared, while the objects themselves are not, which leaves a blank space around the object that will contain information from the background. If the background is detected by the LIDAR, its points will invariably lie within the projected bounding area, generating additional, incorrect islands. This would mean that, for instance, the wall behind the flowers in Figure 7 would be detected as two additional objects, or that the points corresponding to "ball" in Figure 8 would be categorized additionally as "dog".

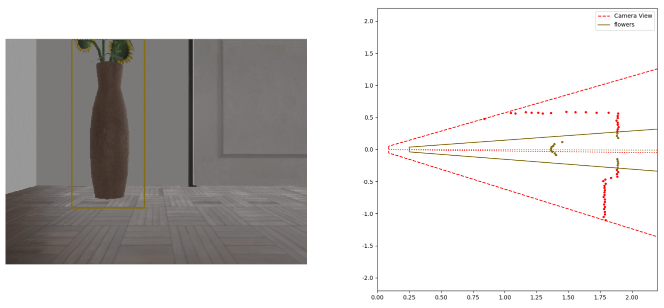

Figure 7: Wall points falling within a "flowers" bounding area

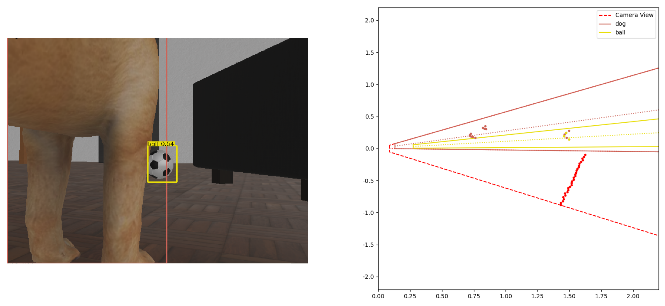

Figure 8: Points falling inside "dog" and "ball"

The second case, as illustrated by Figure 9, is when a small object is detected by YOLO and within the LIDAR's range, but its bounding area overlaps with that of a larger object detected by the camera, but not by the LIDAR. This creates a single set of points with two different possible assignments.

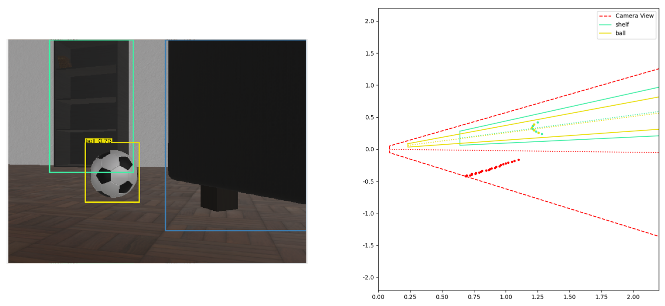

Figure 9: Ball points detected as "ball" and "shelf"

Finally, it is important to take into account that the same objects are being detected across different frames, so it is necessary to ensure that the robot creates only a single instance of each object. Likewise, if the robot moves, due to the errors caused by the inaccuracy of the odometry and the variability in the YOLO bounding boxes, a single object can be detected in slightly different positions and sizes on each frame.

To handle these edge cases, I use the following assumptions:

1. Each bounding projection contains one and only one correct object.
2. Since the actual object lies at the center of its YOLO box, the correct island will be the one closest to the center line (bisector) of the projection.
3. Because we are dealing with a flat surface, closer objects will cast a projection that is closer to the camera, regardless of size.
4. Two objects cannot lie on similar coordinates in a way that makes them overlap.
5. If a new object is detected that is close to an old object of the same class, it is considered the same object.

Algorithm 1 takes into account the edge cases and utilizes the assumptions to create an object detection and tracking system with very high fidelity and very little noise. The customizable tracking threshold distance is used to track the objects across different frames and collision detection is used in addition to a series of conditionals to reduce the amount of incorrect detections. The tracking method also has the side effect that if an object moves slowly while the robot is observing it, its position will be correctly updated. During object merges, the position is updated with a moving average and the size of the largest object is kept.

Object Consciousness Algorithm

```
1: for Every YOLO bounding projection do
2:     Find all the points that lie within it
3:     Apply clustering to its assigned points to generate islands
4:     Find the island closest to the projection's center bisector and define it as the Object $X$
5:     Use translation and rotation matrices to position $X$ in the world
6: end for
7: for Every $X$ and $Y$ objects do
8:     if $X.class == Y.class$ then
9:         if distance($X$,$Y$) $<$ threshold then
10:            Merge the objects
11:        else
12:            Keep both objects
13:        end if
14:    else
15:        if $X$ collides with $Y$ then
16:            if X.projection and Y.projection are at different distances from the camera then
17:                Keep the object whose projection is closest to the camera
18:            else
19:                Keep the object with the highest confidence
20:            end if
21:        else
22:            Keep both objects
23:        end if
24:    end if
25: end for
```

2.5.4 Soft decision

Although Algorithm 1 is very effective at the movement speeds that the Turtlebot3 handles, it is not perfect. As it is, it will sometimes output fake objects, or *ghosts*. These can come from a single incorrect YOLO bounding box, or if a background object is closer to the center than the actual object, which happens usually for a single frame when the object is entering or leaving the camera bound. Leaving the system as it is means that after a while it will be littered with ghost objects coming from single incorrect frames.

To mitigate this, I move towards a Soft Decision output, where the objects go from being definitely there or not there, to having a probability of being there. This probability, or presence confidence, comes from the accumulated confidence of the YOLO boxes, divided by a value that diminishes with relation to the amount of frames that the object has been observed for. This means that objects who are observed for only a couple of frames will have lower presence confidences, even if the YOLO confidence is high. Whenever the robot moves and a new version of the object is tracked, the presence confidence is updated with these criteria as follows:

$$X_{p_0} := X_c/F$$

$$X_{p_i} := max(Xp_{i-1}, Y_c \cdot X_f/F)$$

$$X_f := X_f + 1$$

Where X and Y are the old and new objects, respectively, p_i is the presence confidence at this timestep, c is the confidence of the YOLO detection, f is the number of frames the object has been present for and F is a constant that defines after how many frames should the object be considered actually present.

The result is a set of objects with varying probabilities, and a minimum presence threshold can be set to filter out the improbable ones depending on the use case's criteria.

3 Development

With a defined system, I begin the implementation. For this, a simulated world is created, the YOLO network is trained with a predefined set of classes, the camera is calibrated, and the threshold and presence constants are selected.

3.1 Simulation

As stated in the beginning sections, the Webots environment was selected for the development of this project, with a digital clone of the Turtlebot3 Burger inside. The chosen environment was a simple apartment, with objects of the following set of classes:

$$\text{Classes} = \begin{bmatrix} Ball & Dog & Gnome & Table \\ Door & Rubberduck & Person & Extinguisher \\ Chair & Shelf & Flowers & \end{bmatrix}$$

During experimentation, the rubber duck turned out to be too short to be detected by the LIDAR, and the chairs were too big to be identified as single objects, so they were both removed.

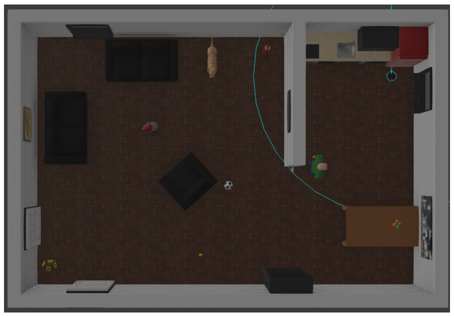
Figure 10: Simulated environment.

3.2 Training

The YOLO network has the advantage that it has been pre-trained with the COCO library, and it is possible to use Transfer Learning to customize the model's head to output a value from the specified classes. For training, I took a total of 782 images from the subjects at varying angles, from the perspective of the robot. On each image, a bounding box was manually drawn over each object using a custom Python script.

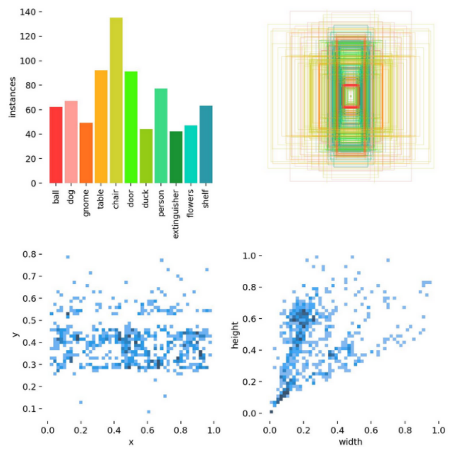

Figure 11: Visualization of training data. Generated by YOLO.

The model was trained using this data with Ultralytics' Python tool, on an NVIDIA GeForce RTX 3050. Several experiments were performed until the best model was obtained, which was the Nano model with a 224 pixel wide input, trained with 64 batches for around 200 epochs. This model obtained an overall precision of 89.9% with a full processing speed of 625 images per second. Figure 12 displays the most significant experiments.

Model	Batches	Image Width (px)	Epochs	Precision	Speed (fps)
Nano	16	640	100	0.895	111
		224		0.897	625
	64			0.890	175
	192			0.865	175
	64		**237 (autostop)**	**0.899**	**625**
XL	64			0.873	103

Figure 12: Training experiments.

3.3 Calibration

The simulated camera was calibrated using CV2's calibration tool, which uses a series of images of a checkered pattern to identify the values of the intrinsic matrix K.

$$K = \begin{bmatrix} 565.14 & 0 & 322.43 \\ 0 & 566.11 & 255.41 \\ 0 & 0 & 1 \end{bmatrix}$$

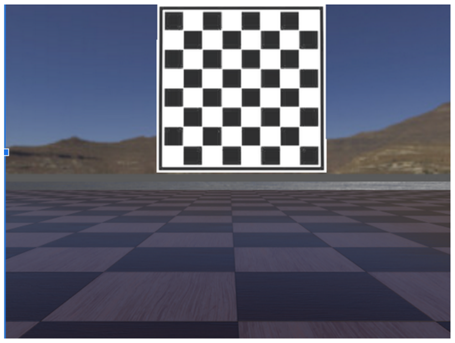
Figure 13: Example of image used for calibration

Once the camera was calibrated, it was tested by selecting points on an image of a checkered floor with squares of size 1m, and it was verified that the projected points laid correctly on the corresponding 3D coordinates.

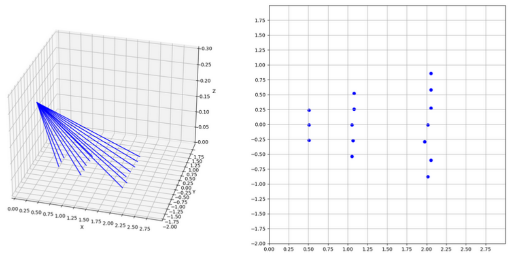
Figure 14: Calibration test

3.4 Constant selection

The necessary constants were selected through experimentation. They are dependent on the performance of the YOLO model, the rotation and translation speed of the robot being used and the performance of the system running the code.

Tracking Threshold	= 30cm
Island Threshold	= 10cm
Presence Confidence Threshold	= 80%
Rolling Avg Weight for Merging	= 90%
YOLO Confidence Threshold	= 40%
Minimum Frames for Presence	= 5

4 Results

A video with the results can be found here [2]. The system proved to have a perfect sensitivity and a very high precision, with minimal false positives and no false negatives. All objects in the scene were correctly placed, tracked and identified. A certain number of ghosts appeared, which are justified by the limitations described in the next section.

Objects in Scene	= 10
Objects in Map	= 16
True Positives	= 10
False Negatives	= 0
False Positives due to Legs	= 3
False Positives due to Corner Problem	= 2
Other False Positives	= 1

This gives a precision of 81% if we allow each leg of legged objects to have a different position or 62% if not. Solving the Corner Problem would increase the precision to 93%. On the other hand, the sensitivity is of 100%.

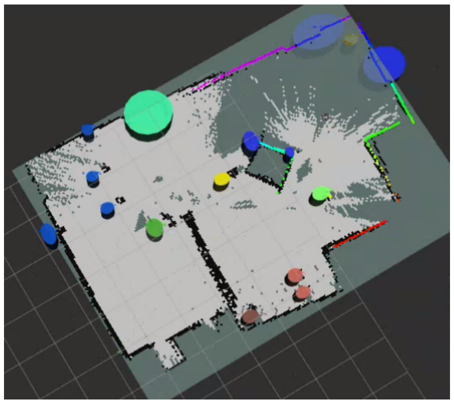

Figure 15: Generated map with objects

4.1 Limitations

Certain edge cases that generate False Positives or False Negatives have not been solved by this project.

4.1.1 Height

Due to the fact that the LIDAR is not flush with the ground, but is rather atop the robot at a specific height, any object shorter than this height, or that is standing on another, higher object, would not be properly detected by the system. This was the case of the rubber duck, as stated at the beginning of the Development section.

4.1.2 Legs

As is the case with the table and the dog, for example, certain objects have legs. If the legs of the object are taller than the LIDAR, and they are separated by a distance larger than the *island threshold*, each leg will be detected as a new object.

To solve this, it would be possible to add additional post-processing that takes into account the characteristics of each class of entity, in order to group each leg object into a single object. This is, however, out of the scope of this project.

4.1.3 The corner problem

This edge case occurs when the corner of an unclassified object that is detectable by the LIDAR falls inside the bounding box of an object that is too far away to generate a point cloud. When this happens, the island that is both closest to the center bisector of the projection and within the projection closest to the camera, is incorrect. Since the true entity is too far away, a correct object is not generated and the collision and position conditionals cannot be applied. So far, I have found no solution to this situation, and this is a point that requires further development.

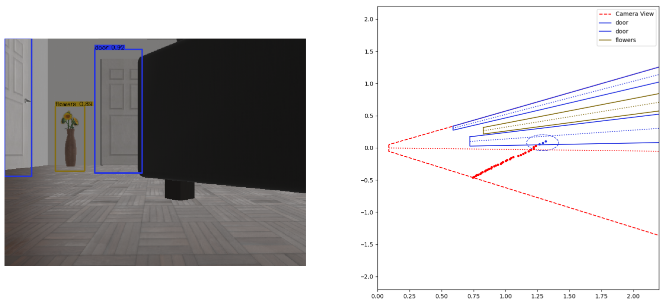

Figure 16: Corner of chair identified incorrectly as door

5 Conclusion

In this work, I presented a system capable of simultaneously mapping the environment, locating the robot within the world and identifying, classifying, locating and tracking a series of objects, with only a LIDAR, a camera, and two encoders as sources of information. This demonstrates that an Easy Object-Conscious SLAM model can be implemented on a robot with minimal components. Taking into account certain limitations, the sensitivity and precision of the system is adequate and its implementation on a real environment should be possible without issues. EPC

References

[1] Mordechai Ben-Ari and Francesco Mondada. "Robotic Motion and Odometry". In: *Elements of Robotics*. Cham: Springer International Publishing, 2018, pp. 63–93. ISBN: 978-3-319-62533-1. DOI: 10.1007/978-3-319-62533-1_5. URL: https://doi.org/10.1007/978-3-319-62533-1_5.

[2] Esteban Padilla Cerdio. *Easy Object-Conscious SLAM — youtube.com*. https://www.youtube.com/watch?v=ok1H8ay-ka0. [Accessed 31-03-2024]. 2024.

[3] Bob Fisher. *3x4 Projection Matrix — homepages.inf.ed.ac.uk*. https://homepages.inf.ed.ac.uk/rbf/CVonline/LOCAL_COPIES/EPSRC_SSAZ/node3.html. [Accessed 29-03-2024]. 1997.

[4] Tully Foote and Melonee Wise. *What is a TurtleBot?* URL: https://www.turtlebot.com/.

[5] Sean Gillies et al. *Shapely*. Version 2.0.2. Oct. 2023. DOI: 10.5281/zenodo.5597138. URL: https://github.com/shapely/shapely.

[6] Itseez. *Open Source Computer Vision Library*. https://github.com/itseez/opencv. 2015.

[7] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. *Ultralytics YOLO*. Version 8.0.0. Jan. 2023. URL: https://github.com/ultralytics/ultralytics.

[8] Steve Macenski and Ivona Jambrecic. "SLAM Toolbox: SLAM for the dynamic world". In: *Journal of Open Source Software* 6.61 (2021), p. 2783. DOI: 10.21105/joss.02783. URL: https://doi.org/10.21105/joss.02783.

[9] Steve Macenski et al. "The Marathon 2: A Navigation System". In: (2020). DOI: 10.48550/ARXIV.2003.00368. URL: https://arxiv.org/abs/2003.00368.

[10] O. Michel. "Webots: Professional Mobile Robot Simulation". In: *Journal of Advanced Robotics Systems* 1.1 (2004), pp. 39–42. URL: http://www.ars-journal.com/International-Journal-of-%20Advanced-Robotic-Systems/Volume-1/39-42.pdf.

[11] Open Robotics. *Setting up a robot simulation (Webots) 2014; ROS 2 Documentation: Galactic documentation — docs.ros.org*. https://docs.ros.org/en/galactic/Tutorials/Advanced/Simulators/Webots.html. [Accessed 30-03-2024]. 2014.

[12] Open Robotics. *teleop$_t$wist$_k$eyboard*. 2015. URL: http://wiki.ros.org/teleop_twist_keyboard.