
Politecnico di Torino


## Politecnico di Torino

Master's Degree in DATA SCIENCE AND ENGINEERING


# Real-Time 3D Reconstruction of Indoor Environments with NVIDIA NvBlox


**Supervisor:**
Marina Indri

**Candidate:**
Vito Perrucci

**Advisors at LINKS Foundation:**
Gianluca Prato
Enrico Ferrera


**March 2026**

*A mia sorella Elena.*
*Oltre l'affetto e il supporto.*
*Oltre il legame e la stima reciproca.*
*Grazie per la tua presenza e la tua amicizia.*

**Abstract**

Although 3D reconstruction may seem useful only for graphic and remote visualization of spaces on electronic devices, its application extends beyond aesthetics, especially when it can be performed in real time. In the field of autonomous robotics, volumetric reconstruction plays a fundamental role: by transforming the complexity of physical space into a structured digital model, it allows robots to interpret the environment to carry out critical tasks, such as navigation and path planning. However, generating detailed volumetric maps imposes significant computational challenges, which often exceed the capabilities of standard CPU-based architectures. The need to operate in real time also requires a delicate balance between the quality of the reconstruction and the processing speed.

This work presents the design and implementation of a 3D reconstruction system capable of responding to these needs, developed in collaboration with the LINKS Foundation, which provided the support and equipment necessary for the experimental validation. The framework analyzed is NvBlox, a solution distributed by NVIDIA and specifically designed to bring three-dimensional reconstruction in real time by exploiting the power of parallel computing on GPU. Specifically, the algorithm adopts Truncated Signed Distance Fields (TSDF), preferred to classic SDFs to limit memory use and have greater numerical robustness; it integrates the calculation of Euclidean Signed Distance Fields (ESDF), whose usefulness is crucial in the field of autonomous navigation for calculating the distance from obstacles.

The proposed solution integrates this framework within the ROS ecosystem, managing the configuration of dependencies through the use of a containerized environment (Docker), essential for the correct functioning of the various software modules. The development of the project follows a rigorous software engineering approach: starting from simulation environments, ideal for the exploration of sensory configurations and safe development, culminating with deployment in the real world on a quadruped robot and experimental validations to assess both reconstruction quality and GPU computational load.

# Table of Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

In the modern era, computerized digitalization is a fundamental pillar. The process is not limited to simply convert data into a format that can be read by a computer, but aims to create a format of information that can be used, shared, and viewed via computer devices.

While a photograph is sufficient to share simple information, complex situations demand a higher level of detail. For example, to allow an architect to assess a construction site remotely, more than simple images are required. This is where 3D reconstruction comes into play.

3D reconstruction allows for the advanced digitalization of reality, transforming physical spaces (such as construction sites or models) into three-dimensional models that can be viewed anywhere. Its applications go beyond mere remote visualization, becoming essential even for the autonomous understanding of the environment by a robot. By mapping the surrounding space in real-time, the robot can better understand how to navigate into it.

## 1.1 Objectives

The work conducted at the LINKS Foundation aims to achieve practical 3D reconstruction of indoor environments. The collaboration with the research center also starts as a challenge: integrating a more software-based engineering approach into a project where practical robotics is the core of the activity. Therefore, it is important to highlight that, before defining the specific objectives, extensive initial research in the field was undertaken. Specifically, the thesis pursues the following objectives:

1. **Analysis of the state of the art**: Study and understanding of current 3D reconstruction technologies, evaluating the advantages and disadvantages of

different approaches, with the aim of identifying the most suitable choice for the specifics of our work.

2. **Development of a preliminary environment**: Creation of an intermediate project useful and necessary for understanding the ROS[1] software ecosystem and configuring the operating environment. This work offers a foundational starting point for modeling and integrating the reconstruction algorithm, facilitating comprehension of ROS project structures and environment setup for broader applications.

3. **Integration in a simulated environment**: Integration of 3D reconstruction into a simulated environment to analyze initial issues and safely evaluate its performance.

4. **Leap into the real world**: Transferring the system to the hardware available at the LINKS Foundation. The ultimate goal of this thesis is to verify the functionality of the 3D reconstruction with a physical robot and real data.

## 1.2 Thesis Structure

The thesis structure reflects the entire study and development process. The chapter organization was designed so that each Section adds value to the previous one, completing a concept while leaving room for further development.

- **Chapter 2**: This chapter covers the literature review conducted during the preliminary study phase. Specifically, three different approaches to 3D reconstruction are described and analyzed, along with their specific frameworks.

- **Chapter 3**: This chapter discusses the preliminary design in ROS and the necessary technical background. This chapter is essential for understanding the subsequent Sections: it summarizes the basics of the ROS software environment, which is useful for understanding the project's operation and the architecture developed.

- **Chapter 4**: This chapter delves into the heart of 3D reconstruction. It describes how it was integrated into the project presented in the previous chapter, analyzing its implementation, the various configurations, and the experiments conducted. At this stage, the analysis will remain limited to the simulation context.

[1]ROS (Robot Operating System) is an open-source framework that acts as a software operating system for robotic applications. It is discussed in detail in Section 3.1.1

- **Chapter 5**: It covers real-world deployment: setup configurations, field results, and potential future experiments to test on the hardware are described.
- **Chapter 6**: It shows some conclusions, with an analysis of the initial objectives compared to the results obtained, the limitations of the work performed, a final discussion, and possible future developments based on this foundation.

# Chapter 2

# Literature Review

In recent decades, thanks to continuous technological advances that have combined computer science and robotics, it has become increasingly important for robots to be able to perceive, model, and store a three-dimensional representation of their environment. Different ideas and techniques have emerged over the years: each has been explored to overcome the limitations of the previous ones, improve them or, in some cases, reinvent the goal itself, observing it from a different perspective and redefining the task.

In fact, as we have seen, reconstructing an environment in 3D is often a means of enabling the robot to understand it better; it is not always an end in itself, but rather opens up new scenarios for behavior and analysis. This chapter investigates different reconstruction methods that use the same type of sensors, analyzing their fundamental principles, disadvantages, advantages, and complexities.

## 2.1 Volumetric Occupacy Mapping

The first analyzed approach is volumetric occupancy mapping, which can be considered a three-dimensional extension of two-dimensional occupancy grid mapping [1].

### 2.1.1 2D Occupancy Grid Mapping

To introduce the concept, it is useful to start with the 2D case: imagine, for example, a robot vacuum cleaner that, before performing its task, must map the floor (2D plane) by detecting obstacles and clear passages, so that it can move autonomously and reach all corners of the room.

This method is based on a pre-initialized grid map, composed of square cells of known side length, which discretize the plane into small elementary regions.

The goal is to use sensor measurements to label each grid cell as occupied, free (unoccupied), or unknown, according to a probabilistic occupancy model that represents the estimated presence of obstacles in the space [2].

In this approach, the resulting map represents regions with a high probability of being occupied, regions with a high probability of being free, and areas not yet observed, explicitly maintaining uncertainty within the concept of probability itself. This leads to several advantages: for example, since perfect sensors are not available, it is possible to combine multiple measurements of the same cell to reduce the effect of noise and obtain more robust estimates over time [2, 3]. Another important aspect concerns sensor fusion: it is useful to be able to integrate, in the same occupancy grid scheme, not only repeated measurements from the same sensor acquired from different points of view, but also measurements obtained from different sensors, for example by combining a LiDAR with a depth camera [4]. Furthermore, in more advanced scenarios, the probabilistic approach allows the integration of data collected from multiple robots exploring different areas of the same environment, improving the coverage and reliability of the overall map [5].

This framework is detailed in [6], in which the maps (called sonar maps, because they are made using sonar sensors) consist of two-dimensional arrays of cells corresponding to the horizontal grid defined on the area to be explored. The grid is typically of size $M \times N$, with each cell covering an area of size $\delta \times \delta$ , and the final map assigns each cell an occupancy value in the range $(-1,1)$: values less than 0 represent cells considered likely to be free, values greater than 0 indicate cells likely to be occupied, while a value exactly equal to 0 corresponds to cells that are still unknown. In addition, this work introduces a decision threshold that allows the map to be "cleaned up" and a more accurate representation to be obtained: a threshold value Ru is used for thresholding, discarding measurements below a certain range in order to reduce spurious reflections and noise in the final map. The authors show, through two images, a 2D map obtained with this method before thresholding (Figure 2.1) and the corresponding map after this filtering process (Figure 2.2) , highlighting the qualitative improvement in the representation.

The maps use the following legend for occupancy estimation: white space corresponds to empty areas with high certainty, while empty areas with a lower certainty factor are displayed with symbols '+' of progressive thickness. Occupied areas are marked with '**x**', and unmapped areas with '.'. Finally, circles indicate the points where the robot acquired the scans, and solid lines mark the outlines of the main objects.

It can therefore be observed that the map in Figure 2.2 is cleaner, the consistency of the data taken is not compromised in any way, but the accuracy of the reconstruction is increased.

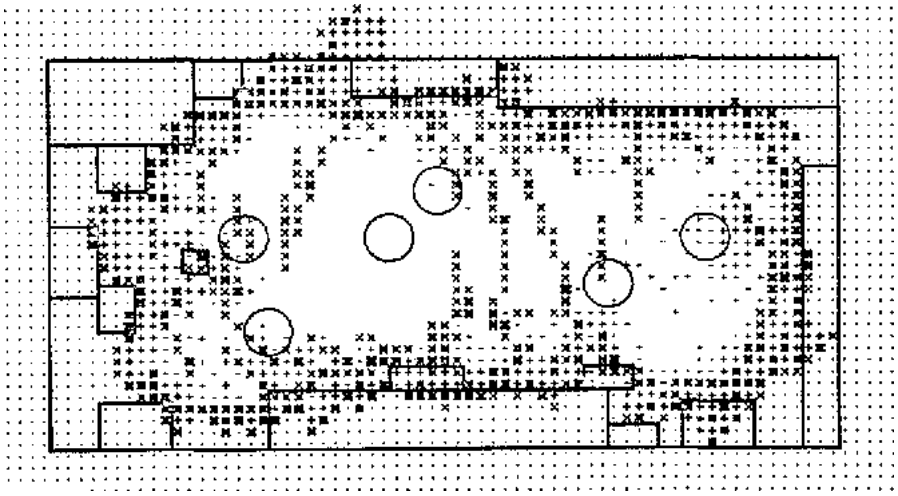


**Figure 2.1:** Two-dimensional sonar map before thresholding [6].

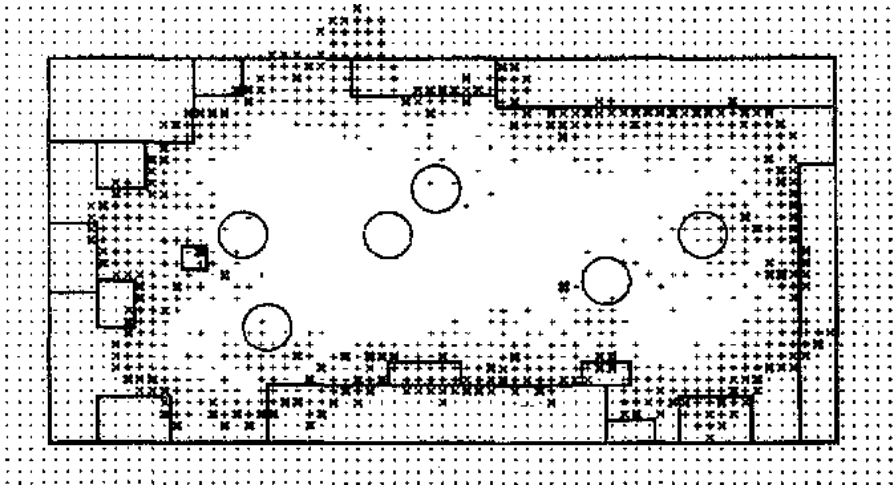


**Figure 2.2:** Two-dimensional sonar map after thresholding [6].

### 2.1.2 3D Extension of the Occupancy Grid

Once the concept has been clarified on the 2D plane and it has been understood how measurements are accumulated and filtered to obtain a consistent occupancy grid, it is natural to extend the discussion to the three-dimensional case, where, with appropriate substitutions of the components used (for example, switching from square cells to voxels and from grids to tree structures), it is possible to move from the 2D plane to 3D space, while maintaining the same basic probabilistic framework [1, 7].

To move into a three-dimensional space, we need to extend the concept of the

grid and the cells it is composed of, introducing voxels, or volume elements that represent a value on a regular 3D grid; they are the three-dimensional equivalent of pixels, the name "voxel" (volume element) is in analogy with "pixel" (picture element) [8]. We will therefore talk about volumetric occupancy mapping, in which the key object is no longer the cell area but the volume associated with each voxel, while maintaining the same objective as the 2D approach: to build a model that allows to distinguish between free, occupied, and unknown space [1, 7].

Even in the volumetric case, the formulation is probabilistic: each voxel is assigned an occupancy value, updated using probability models. The process is carried out by performing ray casting starting from the sensor; for each ray, the voxel hit is marked as occupied, while the voxels in between are marked as free, obviously leaving unknown the regions that have not sbeen observed yet. In many implementations, each voxel stores a log-odds occupancy value that is updated using a recursive Bayesian filter, making the estimates more stable as the number of observations increases.

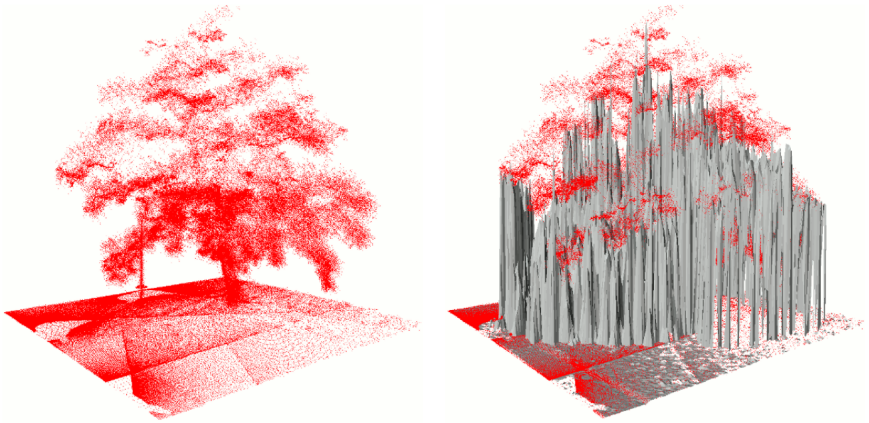


**Figure 2.3:** Comparison between point cloud and Occupancy-Elevation Grid mapping a tree [1]

.

However, storing these values for all voxels in a dense 3D grid is costly in terms of memory and computation, especially when the mapped area is large [1, 7, 9]. An example of an alternative solution is proposed in [9], in which information is stored on a 2D plane limited to the height of the observed obstacle: each cell is described by a tuple of the type $m_n = \langle o_{0:t}, u_{0:t}, \sigma^2_{0:t} \rangle$, where $o_{0:t}$ is the probability of occupancy of the cell, $u_{0:t}$ is the estimate of the height, and $\sigma^2_{0:t}$ is the variance of that estimate. This approach significantly reduces memory consumption, but discretizes the information along a single vertical dimension and is strongly influenced by the height of the sensor. Figure 2.3 illustrates how limited the information provided

by the described reconstruction method is when compared with a standard point cloud.

An alternative representation, less heavy than a true volumetric grid, consists of maintaining a list of occupied voxels "above" each 2D cell: this reduces the total number of stored elements, but loses information about free and unknown voxels [10].

### 2.1.3 OctoMap: 3D Mapping Framework based on Octrees

The OctoMap framework addresses the problem of memory and scalability by introducing a different data structure, the octree, to represent the space. An octree is a hierarchical structure of 3D spatial subdivision: each node represents a cubic volume (voxel) that can be recursively subdivided into eight smaller subvolumes of equal size, until the desired minimum resolution is reached (Figure 2.4).

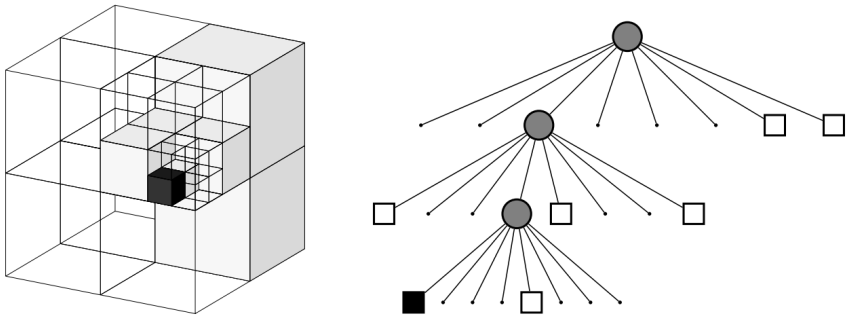


**Figure 2.4:** The volumetric representation of space is on the left, illustrating the principle of spatial subdivision, where a voxel is recursively divided into eight sub-voxels. The black blocks indicate occupied cells and the white blocks indicate free cells. The corresponding tree structure (Octree) is on the right, which hierarchically encodes this spatial subdivision [1].

As in the grid-based case, during ray casting, the voxel hit is initialized as occupied, while the voxels crossed by the ray are initialized as free; the states explicitly represented are therefore "free" and "occupied", thus allowing the use of compact encoding, for example, on log-odds with thresholds for binary classification [1, 7].

In detail, the probability $P(n \mid z_{1:t})$ that a leaf node $n$ is occupied, given the sequence of sensor measurements $z_{1:t}$, is computed as follows:

$$P(n \mid z_{1:t}) = \left(1 + \frac{1 - P(n \mid z_t)}{P(n \mid z_t)} \frac{1 - P(n \mid z_{1:t-1})}{P(n \mid z_{1:t-1})} \frac{P(n)}{1 - P(n)}\right)^{-1} \tag{2.1}$$

where $P(n)$ is the prior probability, and $P(n \mid z_{1:t-1})$ is the previous estimate.

As mentioned earlier, one of the problems solved by the tree structure proposed by the Octomap framework is relative to memory usage. The tree structure allows spatial redundancy to be fully exploited. If all children of a node have the same state (e.g., all free or all occupied), they can be merged into a single parent node representing a larger voxel, drastically reducing the number of nodes needed to represent homogeneous regions, and thus memory consumption.

Moreover, the tree representation makes it natural to perform multi-resolution queries. By dividing the nodes into "inner" nodes and "leaf" nodes, the leaves contain the most detailed local information, while the inner nodes can compactly summarize the overall state of their respective children. Since each internal node has eight children, it is possible to recursively construct a more “coarse” description of the map by applying simple aggregation rules. In particular, two main aggregation methods can be used:

- $\bar{\ell}(n) = \frac{1}{8}\sum_{i=1}^{8} L(n_i)$: average occupancy of child nodes
- $\hat{\ell}(n) = \max_i L(n_i)$: maximum occupancy of child nodes

This allows to obtain low-resolution maps, which are less detailed but much faster to query, particularly useful for path planning or for analyses that require a large-scale overview.

The 3D reconstruction of a tree using OctoMap is shown in Figure 2.5. The same reconstruction is also shown at lower resolutions to observe the functioning of multi-resolution queries.

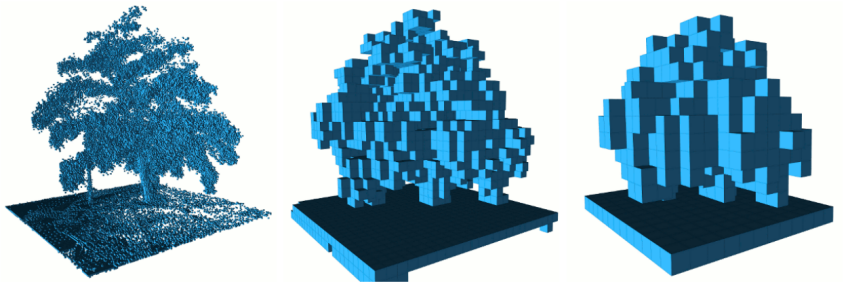


**Figure 2.5:** 3D reconstruction of a tree using OctoMap. From left to right, the map resolution decreases as the voxel size increases [1].

## 2.2 Neural Rendering Mapping: NeRF & 3D Gaussian Splatting

In recent years, with the advent and rapid progress of neural networks, their use has also been extended to rendering for image and video generation. Neural rendering is defined as an approach to image or video generation that allows implicit or explicit control over scene properties [11, 12, 13, 14]. Instead of reasoning from a purely geometric point of view, this approach works on a plane of visualization and photorealism: it is no longer a matter of "stacking bricks," but of "painting" the image according to the point of observation. The matter is not only whether there is an obstacle at a certain point in space, but how that point would appear when viewed from a certain direction [15, 16] .

To understand how this works, let's imagine we are working in the world of computer graphics to create scenes for a movie. First, we start with 3D geometry, add lighting and the camera, and finally obtain a 2D image. This process is called forward rendering: we start with a three-dimensional geometric model to obtain a two-dimensional product. In our case, however, we want the opposite (Figures 2.6): given a set of 2D images, we want to reconstruct a three-dimensional geometric representation of the space, along with the lighting properties and camera parameters. This process, which is the basis of neural rendering, is called *inverse rendering* [14].

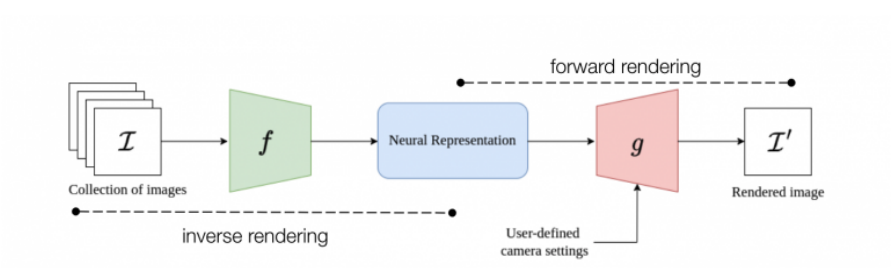



**Figure 2.6:** Inverse Rendering and Forward Rendering.

This approach is based on machine learning concepts: the goal is to use gradient descent to teach the neural network to predict the visualization and properties of a three-dimensional scene from a specific observation origin [11, 15].

Space is modeled as a mathematical function which, given the observer's origin and direction of observation, returns the color of the framed point (and other physical properties) as output, according to a function that governs the entire process [11, 12, 17].

This type of rendering works differentially: the entire image formation pipeline is designed to be derivable. This allows the photometric error gradient to be calculated

and backpropagated to update the scene parameters. With this approach, 3D reconstruction becomes a continuous optimization problem. The scene is seen as a radiance field, in which each point in space not only has matter, but also emits color depending on the direction of observation, allowing reflections, transparencies, and details that are impossible to represent with a simple rigid voxel grid to be modeled [11, 14, 17].

This principle of photometric optimization can be applied using two main strategies [11, 13]:

- using a continuous implicit function parameterized by neural networks;
- using an explicit discrete representation based on optimizable primitives.

### 2.2.1 Using an implicit continuous function: NeRF

We explore this strategy in relation to a specific approach called NeRF (Neural Radiance Fields). NeRF aims to represent the entire scene as a mathematical learnable function [11, 12, 17].

The scene is represented by a single multilayer perceptron (MLP) neural network that learns to map a spatial position $\mathbf{x} = (x, y, z)$ and an observation direction $\mathbf{d} = (\theta, \phi)$ to a color $\mathbf{c} = (r, g, b)$ and a volumetric density $\sigma$. In compact form, the input is $(\mathbf{x}, \mathbf{d})$ and the output is $(\mathbf{c}, \sigma)$.

These input values are obtained from images as follows: for each pixel in the image, a ray $r(t)$ is cast from the camera position and extends in the direction associated with that pixel. Often, not all pixels are sampled for efficiency reasons. The ray is defined as

$$r(t) = \mathbf{o} + t\mathbf{d},$$

where $\mathbf{o}$ is the origin of the ray (i.e., the center of the camera), $t$ is a scalar parameter that identifies points along the ray, and $\mathbf{d}$ is the direction of observation, with

$$t \in [t_{\text{near}}, t_{\text{far}}].$$

The ray is sampled in $N$ discrete intervals

$$\{t_1, t_2, \ldots, t_N\},$$

often uniformly spaced. For each sample, the 3D point is calculated as

$$\mathbf{x}_i = r(t_i) = \mathbf{o} + t_i\mathbf{d},$$

with

$$\mathbf{x}_i = (x_i, y_i, z_i).$$

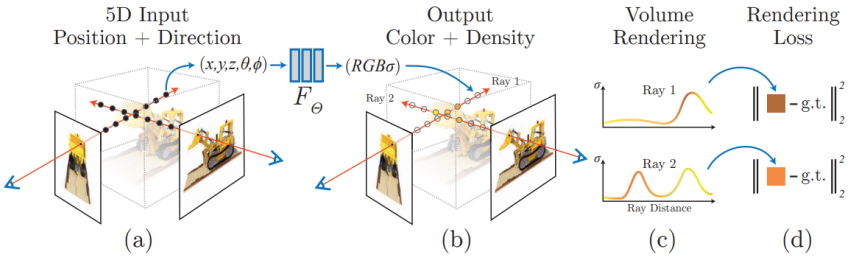



**Figure 2.7:** Neural Radiance Fields pipeline [12].

Each point $\mathbf{x}_i$, together with the direction $\mathbf{d}$, is passed to the MLP, which returns a density $\boldsymbol{\sigma}_i$ and a color $\mathbf{c}_i = (r_i, g_i, b_i)$ (Figures 2.7).

In this way, we can associate each "piece" of the ray with its density and color. By combining all the information present along the single ray, we determine the final color $\hat{C}(r)$ assigned to the pixel:

$$\hat{C}(r) = \sum_{i=1}^{N} T_i \, \alpha_i \, \mathbf{c}_i,$$

where the transmittance $T_i$ is defined as

$$T_i = \exp\left( - \sum_{j=1}^{i-1} \boldsymbol{\sigma}_j \, \delta_j \right),$$

$\alpha_i$ is the opaque contribution of the sample

$$\alpha_i = 1 - \exp(-\sigma_i \, \delta_i),$$

and

$$\delta_i = t_{i+1} - t_i$$

indicates the distance between two consecutive samples along the beam. The transmittance represents the probability that light will reach sample $i$ without being absorbed by the previous samples, ensuring that more opaque contributions reduce the weight of subsequent samples.

The parameters are updated by minimizing a photometric loss function. Let $\hat{C}_p$ be the predicted color for pixel $p$ and $C_p$ be the ground truth color. The loss is defined as

$$\mathcal{L} = \frac{1}{N_{\text{pixel}}} \sum_{p=1}^{N_{\text{pixel}}} \left\| \hat{C}_p - C_p \right\|_2^2.$$

An important observation is the model's poor ability to capture high-frequency details when working directly in the low-dimensional input space. In NeRF, the

raw dimensionality is 5: 3 dimensions for the position $\mathbf{x}$ and 2 for the direction $\mathbf{d}$. It has been seen that by effectively increasing this dimensionality through a nonlinear transformation, the model is able to better represent the fine variations in the observed signals. This is where positional encoding comes into play.

Positional encoding is an input pre-processing technique that transforms low-dimensional data into a high-dimensional vector space using sinusoidal functions at increasing frequencies. In this approach, for each scalar component $\mathbf{x}_i$ of the position $\mathbf{X} = (x, y, z)$, it is defined:

$$\begin{aligned}\lambda(x_i) = \Big[ & \sin(2^0 \pi x_i), \cos(2^0 \pi x_i), \\ & \sin(2^1 \pi x_i), \cos(2^1 \pi x_i), \\ & \qquad \cdots \\ & \sin(2^{L-2} \pi x_i), \cos(2^{L-2} \pi x_i), \\ & \sin(2^{L-1} \pi x_i), \cos(2^{L-1} \pi x_i) \Big].\end{aligned} \tag{2.2}$$

The same scheme is applied to the components of the direction $\mathbf{d} = (\theta, \phi)$. In NeRF, $L = 10$ is chosen for the spatial coordinates and $L = 4$ for the direction. Each $\lambda(x_i)$ will therefore have 10 sine terms and 10 cosine terms, for a total of 20 dimensions; each $\lambda(d_i)$ will have 4 sine terms and 4 cosine terms, for a total of 8 dimensions [11, 12, 17].

With positional encoding, we therefore move from an input of size 5,

$$[x, y, z, \theta, \phi],$$

to a vector of size 76 (20 + 20 + 20 for the three spatial coordinates and 8 + 8 for the two direction components). The coordinates are thus represented in a richer way, "broken down" into different frequencies, and the network can learn high-frequency variations in the color and density signal.

Ultimately, the Multi-Layer Perceptron can be summarized as:

$$\text{MLP} : \Big(\lambda(\mathbf{X}), \lambda(\mathbf{d})\Big) \longrightarrow (\sigma, \mathbf{c}).$$

Figure 2.8 shows the difference between the ground truth, the complete model, and the model without positional encoding: the image obtained without positional encoding is less accurate and less defined because high-frequency dependencies with respect to the inputs are not captured effectively.

### 2.2.2 Using an explicit discrete representation: 3D Gaussian Splatting

After analyzing how the implicit continuous function proposed by NeRF works, a second possibility is to use an explicit representation based on optimizable

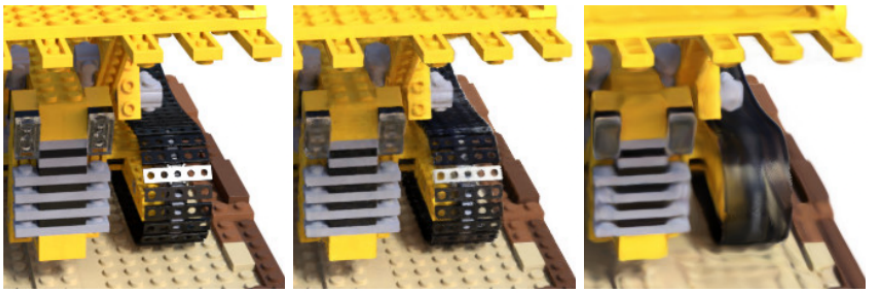



**Figure 2.8:** Comparison Between the Complete Model and the Model Without Positional Encoding on NeRF [17].

primitives. This is where 3D Gaussian Splatting comes in, which lies halfway between volumetric neural representations and explicit methods based on points or meshes [18].

The goal remains to render scenes from calibrated images in order to produce new, high-quality photorealistic views but with a much lower computational cost than NeRF. While NeRF stores the scene in the weights of a neural network, 3D Gaussian Splatting describes the space using an explicit set of 3D (Gaussian) primitives, whose geometric and radiometric parameters are optimized using gradient descent.

The basic data structure is a set of 3D Gaussians, initialized from a sparse point cloud obtained from Structure-from-Motion. Each Gaussian is defined by the following main parameters:

- a position in space $\mu$ (mean);
- a covariance matrix $\mathbf{\Sigma}$ that controls shape and orientation;
- an opacity value $\alpha$, which weights the contribution of the Gaussian in the blending.

The density of the Gaussian at a point $\mathbf{x}$ is given by

$$G(\mathbf{x}) = e^{-\frac{1}{2}(\mathbf{x})^\top \mathbf{\Sigma}^{-1}(\mathbf{x})}$$

which is then multiplied by $\alpha$ in the blending process. The use of anisotropic Gaussians allows adaptation to the local geometry of the scene: elongated ellipsoids can represent thin structures compactly, achieving a relatively dense representation only where needed.

In order to perform rendering, such 3D Gaussians must be projected onto the image plane. Given a view transformation $\mathbf{W}$ of the coordinates from the world

system to the camera system; the covariance matrix in camera coordinates $\mathbf{\Sigma}'$ is obtained as:

$$\mathbf{\Sigma}' = \mathbf{JW\Sigma W}^\top \mathbf{J}^\top,$$

where $\mathbf{J}$ represents the Jacobian of the projection transformation and describes how small 3D displacements near the center of the Gaussian $\boldsymbol{\mu}$ are transformed into displacements on the image plane. Removing the third row and column of $\mathbf{\Sigma}'$, we obtain a $2 \times 2$ matrix that describes the shape of the ellipse projected onto the sensor, i.e., the 2D "splat" used by the rasterizer [19, 18].

In theory, we could optimize $\mathbf{\Sigma}$ directly, but there is a problem: a covariance matrix only makes sense if it is positive semidefinite, and with gradient descent it is easy to end up with invalid matrices. To avoid this, we rewrite $\mathbf{\Sigma}$ in a more convenient way: we interpret the Gaussian as an ellipsoid defined by a scale matrix $\mathbf{S}$ and a rotation matrix $\mathbf{R}$, and use

$$\mathbf{\Sigma} = \mathbf{RSS}^\top \mathbf{R}^\top.$$

In practice, instead of updating $\mathbf{\Sigma}$ directly, we update a 3D vector $\mathbf{s}$ (the three scales) and a quaternion $\mathbf{q}$ (the rotation). From $\mathbf{s}$ and $\mathbf{q}$, the matrices $\mathbf{S}$ and $\mathbf{R}$ are reconstructed, normalizing $\mathbf{q}$ to always have a valid rotation. This allows to work on more intuitive parameters (position, scale, orientation, opacity, color), but the resulting covariance remains automatically physically correct [18].

At this point, the actual optimization comes into play: the Gaussian parameters (position, scale, rotation, opacity, and color) are updated using gradient descent. The views corresponding to the training images are rendered, and a photometric loss between the synthesized image and the real image is minimized. During this phase, the method can also add, move, or remove Gaussians, so as to increase density only in areas where the scene requires more detail and keep the overall representation compact and efficient.

Figure 2.9 shows the steps of the 3D Gaussian Splatting highlighting Operation Flow and Gradient flow.

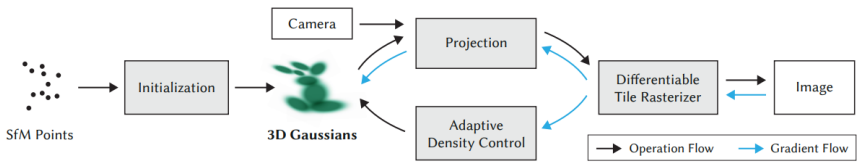



**Figure 2.9:** 3D Gaussian Splatting pipeline [18].

## 2.3 TSDF/ESDF Volumetric Fusion Mapping: NVIDIA NvBlox

The methods discussed so far aim at faithful and visually realistic 3D reconstruction, but in robotics, image quality is often not the ultimate goal: instead, a simple geometric representation is needed, queryable in real time, compatible with tasks such as motion planning and obstacle avoidance.

Signed Distance Functions (SDF) are continuous and regular functions that associate each point in space with the distance from the nearest surface. For application reasons, truncated versions, called Truncated Signed Distance Fields (TSDF), are often used, which concentrate the representation near the surfaces of interest, ensuring limited memory and numerical robustness. SDFs are excellent for integrating (via sensor fusion) heterogeneous data from robotic sensors. In addition, meshes or cost fields for navigation can be generated from these volumes. Figure 2.10 shows an example of a Signed Distance Function, as taken from Wikipedia [20].

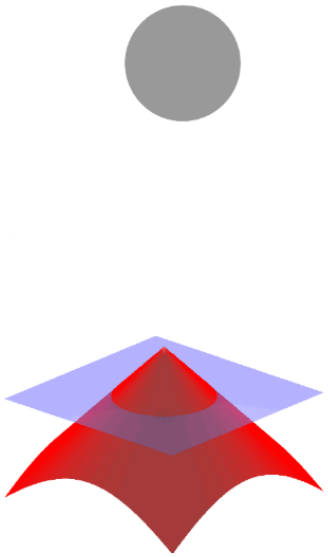


**Figure 2.10:** Signed Distance Function (red) between a surface (blue) and a disk (gray) [20].

In TSDFs, each voxel contains the distance from the nearest surface, saturated above a maximum threshold. Weighted averaging algorithms and Bayesian filters allow for stable data fusion, with little influence from sensor noise.

However, Euclidean Signed Distance Field (ESDF) is used for robotic planning and control, which extends TSDF by calculating the exact Euclidean distance to

all obstacles. TSDF and ESDF often coexist in the same system: TSDF integrates sensory data and reconstructs geometry, while ESDF supports planning [21].

SDF volumetric maps do not replace methods such as NeRF or Gaussian Splatting, but complement them. They are chosen when robustness, simple geometric queries, and real-time updating are required.

### 2.3.1 NVIDIA NvBlox Framework

NvBlox, developed by NVIDIA, was created to bring volumetric reconstruction to real-time by leveraging GPU parallel computing [22]. It integrates the advantages of TSDF/ESDF systems with the speed of the GPU, thus overcoming the scale and resolution limitations typical of CPU-only solutions. NvBlox's goal is to provide a complete pipeline that can be used directly in robotic planning. Figure 2.11 shows the architecture of NvBlox.

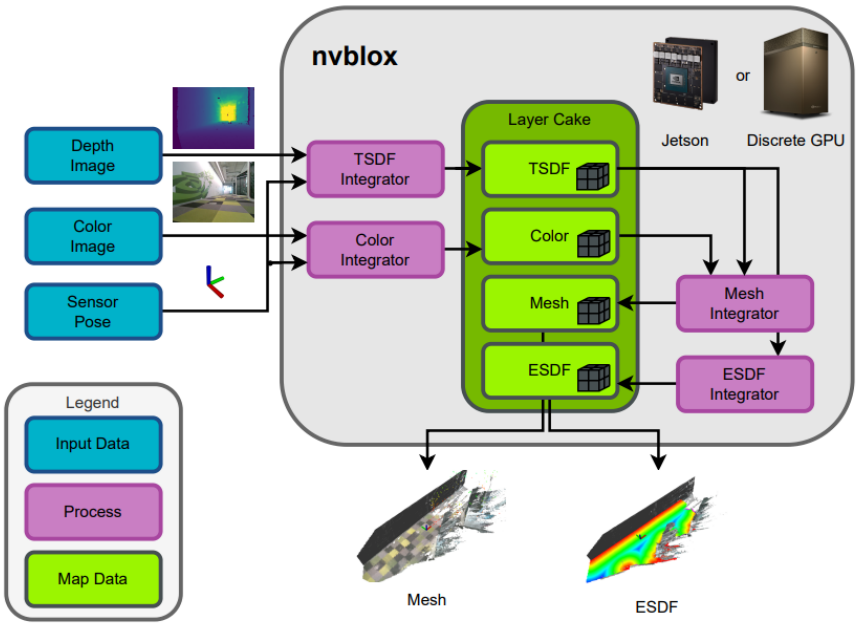



**Figure 2.11:** NvBlox Architecture [22].

Reconstruction in NvBlox is based on functions

$$\Phi : \mathbb{R}^3 \rightarrow \varphi$$

which, given a point, return a quantity useful for voxelization. Samples are allocated only in the regions of the scene that are actually observed.

The observations are depth maps

$$D : \Omega \to d$$

with $d \in \mathbb{R}$ measuring depth in meters, or color images

$$C : \Omega \to \mathbf{c}$$

with $\mathbf{c} \in \mathbb{R}^3$ an array of primary colors. $\Omega$ is the image plane (depth camera) or the angle space (LiDAR). Each observation has an associated sensor pose

$$\mathbf{T}_{C_i}^{L} \in SE(3)$$

in a global frame $L$, which is essential for transforming the data into the map system. Estimating the robot's pose in space is crucial for the final reconstruction task. It provides the fundamental prerequisite for accurately placing obstacles in the map around the robot, which presupposes knowledge of its own position within the environment.

The map in NvBlox is composed of overlapping layers of 3D voxels: voxels are allocated only where the scene is observed (sparse occupancy), implementing a hash-table $\to$ VoxelBlock ($8 \times 8 \times 8$ voxel) hierarchy, which is efficient in terms of memory and GPU access.

Data integration takes place in three main stages:

1. **VoxelBlock allocation:** identification, via ray tracing, of visible blocks and their allocation if they are not mapped.

2. **Voxel projection:** projection of the center $\mathbf{p}^L$ in the sensor frame and then in the image plane using the view transformation $\mathbf{T}_L^C$ and the projection model $\pi_{\text{sensor}}$. For a camera:

$$\pi_{\text{camera}}(\mathbf{p}_C) = \frac{1}{p_{C,z}} \begin{bmatrix} f_u & 0 & c_u \\ 0 & f_v & c_v \end{bmatrix} \begin{bmatrix} p_{C,x} \\ p_{C,y} \\ 1 \end{bmatrix},$$

   where $f_u, f_v, c_u, c_v$ are the intrinsic parameters of the camera. For a rotating LiDAR, the projection is performed in angular coordinates:

$$\pi_{\text{lidar}}(\mathbf{p}_C) = \begin{bmatrix} (\tan^{-1}(p_{C,y}/p_{C,x}) - \theta_{\text{start}})\, \alpha_\theta \\ (\cos^{-1}(p_{C,z}/r) - \phi_{\text{start}})\, \beta_\phi \end{bmatrix},$$

   where $\theta_{\text{start}}, \phi_{\text{start}}$ are the minimum FoV angles and $\alpha_\theta, \beta_\phi$ are the pixel-per-rad factors.

   The depth is sampled from the projection

$$d = D[\pi_{\text{sensor}}(\mathbf{T}_L^C \mathbf{p}^L)].$$

3. **Per-voxel update:** calculate for each voxel the depth of the voxel relative to the sensor $d_v$ and obtain the projective distance

$$d_p = d - d_v.$$

Based on $d_p$, a different update function is applied depending on the layer.

For the **TSDF**, each voxel stores a truncated distance $d_{\text{tsdf}}$ and a weight $w$. The projective distance $d_p$ is limited to the interval $[-\varepsilon, +\varepsilon]$ and weighted by a function $w = f_w(d_p)$ (constant or derived from the sensor error model). The update is performed using a weighted average:

$$d_{\text{tsdf}}^{\text{new}} = \frac{w^{\text{old}} d_{\text{tsdf}}^{\text{old}} + w d_p}{w^{\text{old}} + w}, \qquad w^{\text{new}} = \min(w^{\text{old}} + w, w_{\max}).$$

For the **occupation map**, each voxel stores a log-odds value $l_o$. The update is defined in log-odds space as a running sum:

$$l_o^{\text{new}} = l_o^{\text{old}} + \Delta l,$$

where $\Delta l$ is positive if $d_p < 0$ (voxel in front of the measurement, therefore more likely to be occupied) and negative if $d_p > 0$ (voxel behind the measurement, therefore more likely to be free). This implements Bayesian fusion in log-odds in a numerically stable way.

The TSDF provides truncated projective distances, which are excellent for reconstructing local surfaces, but insufficient for path planning applications that require Euclidean distances to obstacles even beyond the truncation band. For this reason, NvBlox constructs a Euclidean Signed Distance Field (ESDF) from the TSDF (and/or occupancy map).

The ESDF algorithm in NvBlox is based on the Parallel Banding Algorithm (PBA) [23] and is organized at the VoxelBlock level. ESDF voxels are divided into two main categories:

- *sites*: voxels near the edge of the surface, for example with TSDF close to zero, which act as distance sources;
- regular voxels: these can be unknown, free (positive distance), or occupied (negative distance), and record the distance from the nearest site.

Figure 2.12 shows an example with a legend of voxel types based on the reference surface.

The update pipeline can be summarized in the following steps:

1. **ESDF allocation:** modified TSDF blocks are identified, allocating the ESDF structure only where the map has changed.

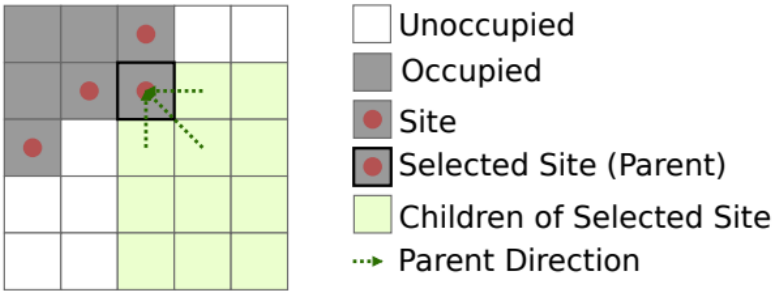



**Figure 2.12:** Site and Regular Voxels representation [22].

2. **Site marking and consistency:** in updated blocks, voxels with $|d_{\text{tsdf}}| \leq \varepsilon$ are marked as sites or, in occupancy, those occupied adjacent to free ones. Indices are updated and child distances are invalidated, if necessary.

3. **Invalid cleaning:** each voxel is checked to see if it still has a valid parent; if not, the distance is invalidated and the block is reinserted for updating.

4. **Lower ESDF (relaxation):** parallel sweeps are performed within the block along all axes, followed by propagation of the new distances to neighboring blocks. This cycle is repeated until the distances converge.

This procedure, which is completely parallel on the GPU, keeps the ESDF constantly updated with respect to changes in the TSDF, quickly providing the Euclidean distance from the nearest surface throughout the map. The result is a distance field that can be used for costmaps, robotic planning, and high-frequency collision checking [22].

# Chapter 3

# ROS2 Project Development

The implementation of a project capable of performing **three-dimensional reconstruction** requires the acquisition of solid theoretical and operational foundations. This chapter describes the preliminary development phase, which is necessary to understand and model a system whose mechanical and functional characteristics are known, as well as its actual potential. Before addressing implementation issues, the technical context is outlined and the algorithmic principles that form the basis of the project are explored in depth. This is followed by a description of the design choices made and a demonstration of the functionality of the sensors.

## 3.1 Technical Background

This Section presents the key concepts, software architecture, and main tools used in the development and visualization of the robotic systems described in the project.

### 3.1.1 ROS 2 Ecosystem

The **Robot Operating System 2 (ROS 2)** is an *open-source framework* entirely dedicated to robotic programming. Contrary to what its name suggests, ROS is not an operating system in the traditional sense, but rather a middleware architecture that provides a set of libraries, tools, and communication standards for creating modular and distributed robotics software. The architecture of ROS is based on the concept of a **node**, which is an executable process that independently performs a single function. The strength of ROS lies in the communication mechanism between nodes, which mainly occurs according to three paradigms:

- **Topics (Publish/Subscribe):** Asynchronous, unidirectional communication mechanism. A *Publisher* node sends messages on a channel (Topic), and any

listening *Subscriber* node receives such messages. The Publisher is unaware of how many Subscribers will actually receive its messages.

- **Services (Client/Server):** Synchronous, bidirectional communication, suitable for operations that require an immediate response. A *Client* node sends a *Request* to a *Server* node, which responds with a *Response* message before continuing execution.

- **Actions:** Extension of previous paradigms, designed for long-term activities (e.g., navigation to a destination). The *Client* sends a goal to the *Server*, which provides periodic feedbacks until the action is completed or interrupted.

Figure 3.1 illustrates the interaction flow between nodes using Topics and Services.

ROS is therefore the basis of our development and provides many nodes that are already written and directly applicable to the project. For example, with pre-existing nodes, we can define the robot model, then generate it on the map in simulation, and finally let it move. With an additional node, we can manage movement control via a joystick, and with another node, we can let it move in space by assigning it a destination. Each implementation in the code deals with the integration of nodes and the management of their data and information communications.

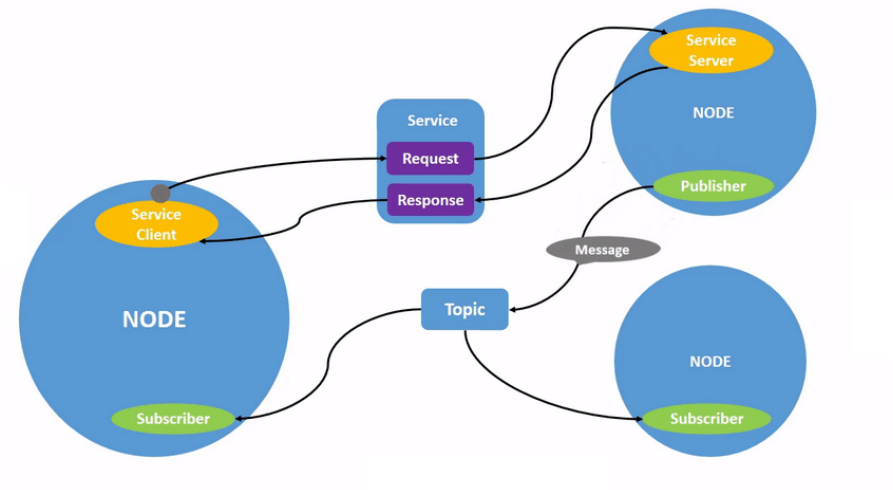



**Figure 3.1:** ROS 2 Architecture: representation of the interaction between nodes via Topics and Services [24].

The 3D reconstruction will be integrated through a dedicated NVIDIA node, which will receive information about the surrounding environment from the sensors

and produce ROS-compatible data representing the reconstruction. In terms of communication, we will primarily use Topic-based interfaces, since the system needs to handle continuous data streams, while in specific cases, such as motion planning, we will instead rely on Services.

Another tool already integrated into the ROS architecture is **ROS Bags**, playing a fundamental role in development and debugging. They allow the message flows exchanged between nodes to be recorded and saved to disk, making it possible to reproduce the same operating scenarios at a later time. It is also possible to choose whether to record all messages or only those related to specific Topics, thus enabling targeted and modular analysis. This approach makes it possible to perform tests and experiments in offline mode, without keeping the physical robot or the simulator constantly active, thereby accelerating the development and validation of the algorithms.

### 3.1.2 Robot Modeling and Coordinate Frames

Before conducting real experiments, it is necessary to perform simulations, so that potential issues can be identified without risking damage to the hardware. To work in a simulated environment, the system must be provided with a formal description of the robot's **kinematics**, **dynamics**, and **geometry**. For this purpose, the **URDF** and **XACRO** formats are used, which are fundamental for an accurate and maintainable robot model.

- **URDF (Unified Robot Description Format):** A description format based on XML that models the robot as a tree of rigid elements, called **Links** (containing inertial, visual, and collision properties), connected by **Joints** that define the degrees of freedom and motion limits. Through URDF it is also possible to attach sensors to the robot, deciding which sensors to use, where to place them on the robot, and how to orient them in space.
- **XACRO (XML Macros):** Since URDF files can easily become long and repetitive, XACRO is used as a macro language built on top of XML, which allows the introduction of macros, variables, constants, and modular file inclusions. This approach improves readability and reuse of code, and makes the robot description easier to maintain and extend when new components or sensors are added.

#### The Static TF Tree

URDF modeling is the basis for generating the robot's static **TF Tree (Transform Tree)**. Each component has its own local coordinate system, and the transformation tree allows coordinates to be converted from one reference to another. These are

mathematical transformations (rotations and translations) that link the different parts of the robot together. Figure 3.2 shows an example of a TF Tree, indicating only the hierarchical relationships.

Defining a *joint* between the robot base link (center), the chassis and the laser sensor sets the exact position of the sensor with respect to the center of the robot in the ROS system. These relationships are managed by the `robot_state_publisher` node, which reads the XACRO file and publishes the static transformations on the `/tf_static` channel. Without this kinematic structure, the robot would not be able to contextualize the data coming from the sensors in space.

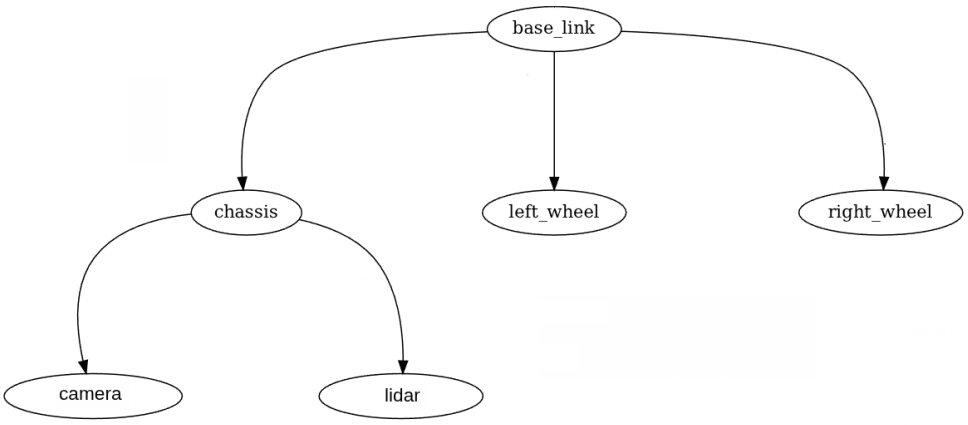



**Figure 3.2:** Example of TF Tree, representation of the hierarchy of coordinate systems defined in URDF/XACRO. It illustrates the fixed transformations from `base_link` to components such as wheels, `chassis`, and sensors (camera, Lidar).

### 3.1.3 Simulation and Visualization Environments

Simulation is a necessary step in the validation of complex robotic systems, as it allows problems to be anticipated, avoiding risks or damage to physical components. It is useful to distinguish between two complementary environments:

1. the **simulation environment**, which generates the physical dynamics of the virtual world.
2. the **visualization environment**, which shows the robot's perceptions, used in both simulation and real-world execution.

#### The Physics Engine: Gazebo vs Isaac Sim

The simulator has the task of replicating the physical phenomena (dynamics, gravity, collisions, friction) and emulating the sensor hardware. In the context of this project, two main environments were analyzed:

- **Gazebo:** Reference simulator for the ROS community. It offers a reliable physics engine and native integration with ROS 2. It is not of high quality from the graphic point of view, but it guarantees efficiency, stability, and full control of simulation parameters.

- **NVIDIA Isaac Sim:** Based on NVIDIA Omniverse, it uses ray tracing and GPU acceleration (RTX), providing high visual and lighting fidelity. It is particularly suitable for realistic studies of advanced sensor technology.

A graphical comparison of the two simulators is shown in Figure 3.3.

Despite Isaac Sim's graphics capabilities — and the fact that the reconstruction framework used (*NvBlox*) is also an NVIDIA product — the choice fell on **Gazebo**. This decision was driven by the simulator's lower computational requirements, which enable faster and more frequent development iterations, as well as by the fact that the primary objective is geometric and dimensional accuracy rather than visual realism. In addition, integrating NvBlox into a simulator different from the one officially proposed by NVIDIA introduces a non-trivial design challenge, broadening the experimental scope and demonstrating the flexibility of the overall system architecture.

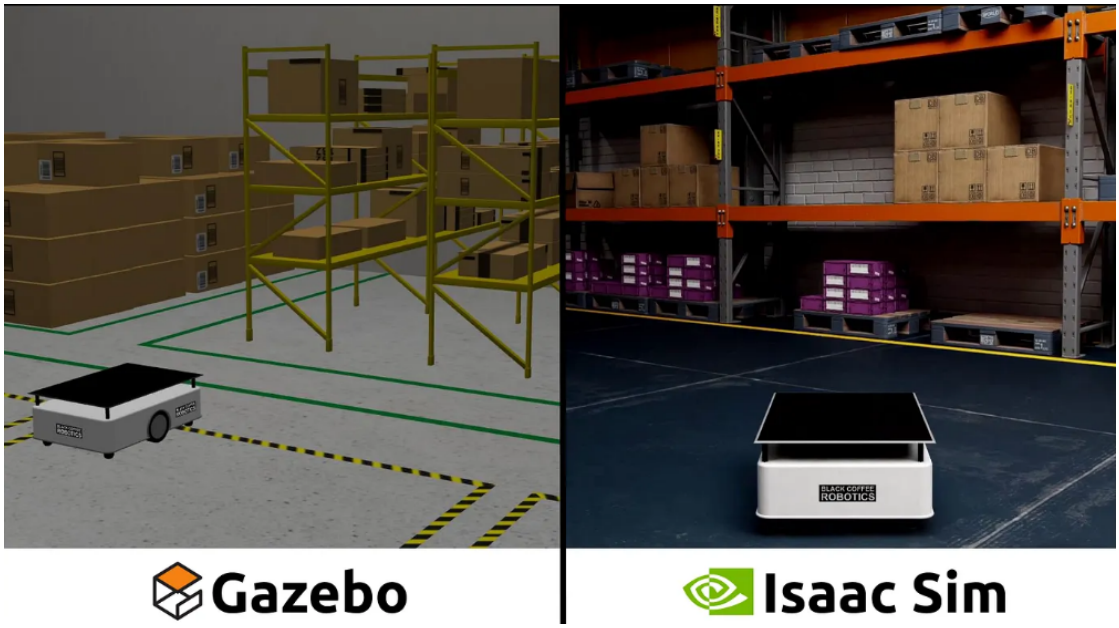



**Figure 3.3:** Side-by-Side Rendering Comparison: Gazebo vs. Isaac Sim Simulation [25].

#### The Data Visualizer: RViz

**RViz (ROS Visualization)** is a 3D viewer that does not perform physical calculations, but allows to represent ROS data flows coming from various nodes in the system.

RViz acts as a listener for Topics: if a node publishes on a channel, RViz can receive the transmitted data, interpret it according to its type, and represent it graphically in a consistent manner.

A clear example is the visualization of the robot itself. As described in Subsection 3.1.2, the **URDF** file provides the description of the robot, which is processed and published in graphical description by the `robot_state_publisher` node on the topic `/robot_description`. RViz can read this topic and visually reconstruct the model.

Similarly to the robot model, RViz is essential for visualizing the raw data acquired by the sensors. It is also the environment where the 3D volumetric reconstruction generated by NvBlox will be rendered, allowing to observe in real time both the movement of the robot within the map and the progressive construction of the three-dimensional spatial representation of the environment.

The combined use of Gazebo and RViz is essential: by comparing the position of the robot in the two environments, it is possible to identify odometry errors (see Subsection 3.1.4), sensory drifts, or inconsistencies in mapping. This is because Gazebo represents the objective simulated reality, while RViz depicts the reality perceived and reconstructed by the robot through its sensors, which is obviously affected by errors. Figure 3.4 shows an example of correspondence between the simulation and its visualization on RViz.

### 3.1.4 Position, Localization, and Navigation Fundamentals

An accurate 3D reconstruction system requires the robot to know its position and be able to correctly represent its surrounding environment. Therefore, it is essential to understand the principles of **positioning**, **localization**, and **autonomous navigation**.

URDF modeling (Subsection 3.1.2) describes the *static* spatial relationships between the robot's internal components, while odometry and localization modules model its *dynamic* relationships with respect to the external environment. The system must be able to determine, instant by instant, its pose (position and orientation) in a global reference frame.

#### Odometry

**Odometry** estimates the variation in the robot's pose over time, publishing the dynamic transformation that connects the `odom` reference system to the center of

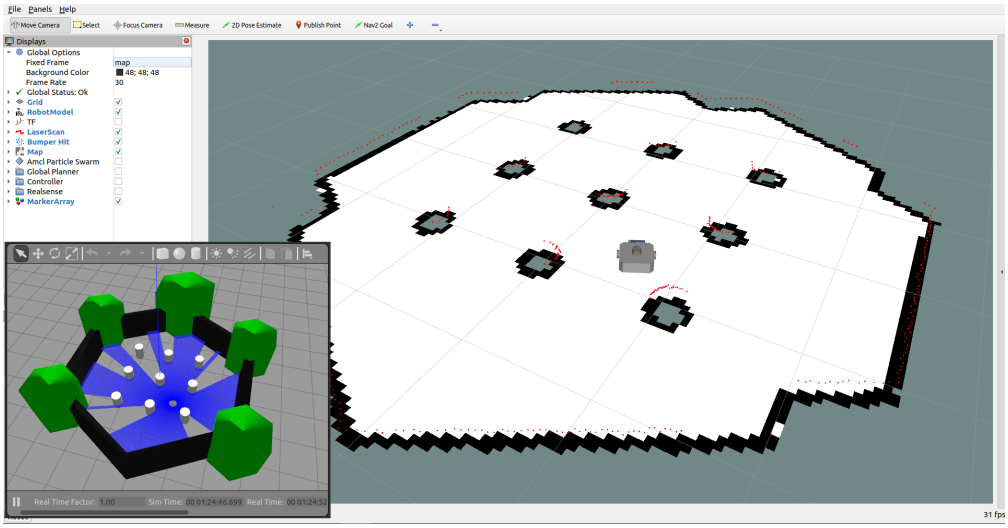


**Figure 3.4:** Gazebo-RViz integration. The Gazebo window (bottom left) shows the simulated scene: the robot equipped with a 2D LiDAR (blue rays) moving inside a simple environment with black walls, green pillars, and white obstacles. The RViz window (background) shows the robot's perception: localization on the 2D map and the point cloud (red) of obstacles detected by the LiDAR.

the robot (`base_link`) or its projection on the floor (`base_footprint`). Figure 3.5 shows how the movement of the robot from one point to another changes the dynamic transformation between its coordinate system and the fixed odometry frame. The variations are calculated based on movement; by integrating linear and angular velocities, a continuous estimate of the position is obtained. Although simple and effective, this technique is subject to measurement errors and wheel slip, causing a gradual accumulation of error (*drift*) with respect to the actual position.

### Localization and Mapping (SLAM and AMCL)

To compensate for drift, the robot must use external references to correct its position estimate. The main approaches available are:

- **SLAM (Simultaneous Localization and Mapping):** Used in unknown environments, it allows the robot to simultaneously construct a map of the environment (typically 2D occupancy) and estimate its position within it. Loop closure allows the errors accumulated by odometry to be corrected.
- **AMCL (Adaptive Monte Carlo Localization):** Used in already mapped environments. It is based on a particle filter that estimates the robot's pose by continuously comparing sensory readings with the known map.

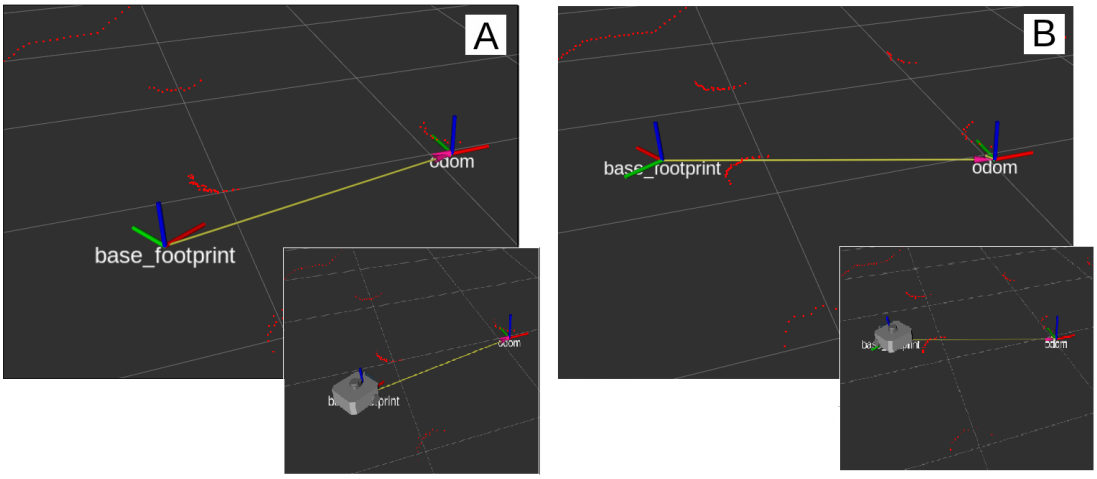



**Figure 3.5:** Representation of the Dynamic Transformation: the robot moves around the map; images A and B show the two positions taken by the robot. We can see how the odometry frame (`odom`), which acts as a static reference point, allows to track the robot's movements through a transformation (yellow arrow) that links `odom` to `base_footprint` (the frame representing the robot's base). The same scene is shown in small format but with a graphic representation of the robot, allowing to understand the robot's direction and compare it with the transformation.

### Autonomous Navigation (Nav2)

As mentioned above, the concepts of localization and mapping, which allow the robot to understand both the surrounding space and its position in it, can be immediately applied to a task that has been extensively explored in robotics: autonomous navigation. In this context, we describe autonomous navigation as the process by which the robot calculates and follows a path to reach a goal that we have provided. We therefore exclude autonomy in the choice of destination, a more complex operation that would require other specific algorithms. The entire process we are interested in is managed by the Nav2 navigation stack. Nav2 is a modular framework based on the Behavior Tree architecture to orchestrate the various navigation tasks. It operates mainly through a double-layer planning organization:

- **Global Planner:** Uses the Global Map (or Static Map), previously created using SLAM or another mapping algorithm, to calculate the optimal and most efficient path from the robot's current position to the final goal.
- **Local Planner (or Controller):** Focuses on executing the short-term plan. Unlike the Global Planner, the Local Planner uses a Local Map (or Dynamic

Cost Map), which is a subset of the global map that is constantly updated with new sensor readings. This allows the robot to perform local and reactive maneuvers, such as braking, steering, or circumnavigating dynamic obstacles not present in the static map, thus ensuring safe movement.

In summary, what Nav2 does is translate the final goal to be achieved into a sequence of movement commands (`cmd_vel`) that allow the robot to reach its destination efficiently and safely. The entire process of planning and executing the movement is illustrated in Figure 3.6, which shows the sequential steps from the user command to the achievement of the goal.

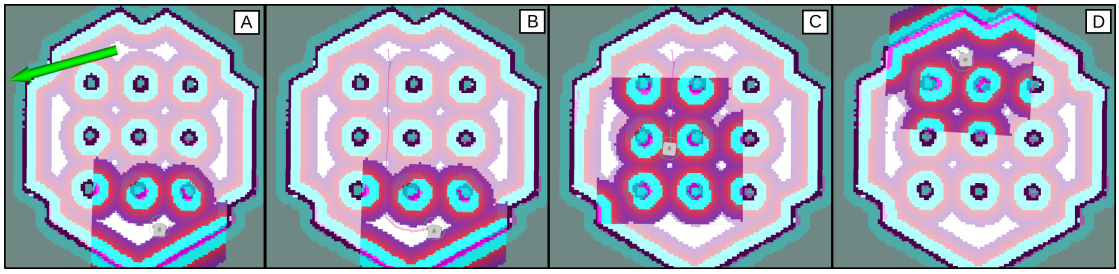



**Figure 3.6: Operating sequence of the Nav2 navigation stack.** **A**: Definition of the goal by releasing a Pose Goal (green arrow) indicating the desired position and orientation. **B**: Generation of the Global Path by the Planner; note the trajectory calculated based on the Global Costmap. **C**: Active navigation phase; the robot follows the trajectory while the Controller dynamically updates the Local Costmap (highlighted box) to manage any immediate obstacles. **D**: Successful arrival at destination and stopping of the robot in the target pose.

The main data structure shared by the global and local planners is the `Costmap2D`. Instead of a binary grid that simply marks cells as free or occupied, the costmap encodes the environment as a grid in which each cell stores a numeric cost expressing how suitable that area is for traversal. Nav2 constructs this map through multiple layers: a static layer derived from an initial map already created in YAML format, an obstacle layer updated from live sensor readings, and an inflation layer that expands the cost around obstacles to reflect the robot's footprint and to maintain a safety margin. A visual representation of the overlapping layers is shown in Figure 3.7.

### 3.1.5 Docker Containerization and Project Sharing

The use of the tools just described is not straightforward but requires substantial installation, configuration, and resolution of incompatibilities between different library and application versions. Each component may be compatible only with

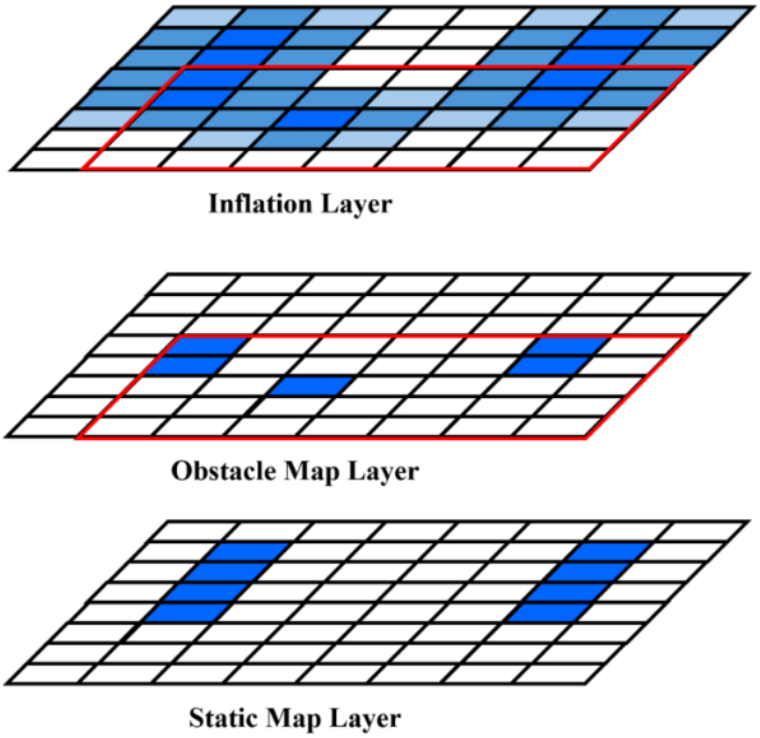



**Figure 3.7:** Layered costmap schema [26].

specific versions of other tools, creating a chain of dependencies and potential errors that should be avoided during *deployment*, especially if the goal is to reproduce the project—along with related tests and experiments—on any sufficiently performant computer.

This is where **Docker** and the concept of **containerization** come into play: Docker is a platform that packages an application and all its dependencies (libraries, environment variables, configurations) into a standard unit called a **container**. The creation of a container begins with writing one or more **Dockerfile**s—text files containing a sequence of commands needed to build an **image**. The image can be thought of as a sort of "disk" constructed to properly launch the actual container (Figure 3.8). Thus, to run the application on a new machine, it suffices to share the Dockerfiles used to build the image, or the image itself, simplifying and accelerating the distribution of the execution environment.

Containers can be considered a lightweight alternative to traditional virtual machines[1], with the key difference that they share the host operating system's

[1]**Virtual Machines** (*VMs*) are complete environments that emulate an entire operating

kernel, making them more resource-efficient. Since the container is effectively a separate environment from the host computer, there is a need to properly manage the project files being worked on. To this end, **mounting** is used: a folder on the host computer is linked to a folder inside the container, creating a correspondence where modifications made on either side are immediately synchronized. This allows working inside the container while keeping the code always updated on the host as well.

In summary, **Docker** simplifies software infrastructure management, ensuring **portability** and **reproducibility** of the execution environment. In the context of this thesis project, Docker will be revisited in a later phase, as it was not needed during the initial development stages.

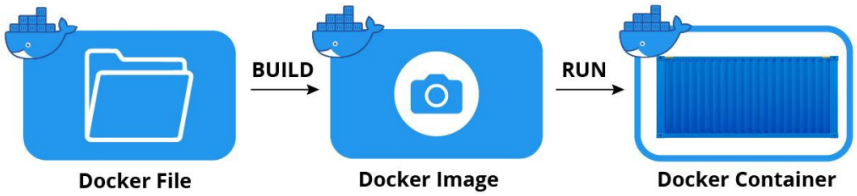



**Figure 3.8:** Docker containerization process [27].

## 3.2 Robot Description and Configuration within the ROS2 Project

After explaining the basic concepts for understanding the project structure, the following Sections describe the implemented code. It is useful to reiterate that the project explained in this chapter is aimed at serving as an initial structure. This structure enables a clear understanding of the correct functioning of a ROS project and provides a solid foundation for future integrations. The project was built based on the 'my_bot' template developed by Josh Newans, an Australian mechatronics engineer. Through his YouTube channel 'Articulated Robotics' [28], he provides numerous useful insights in the field of robotics.

His template is available for free use on his GitHub [29] and was selected due to its convenient and easily comprehensible structure.

The ROS project follows a standardized workspace based on `colcon` (COmmand Line CONsolidator), a modular build system that automates the construction of

system, including a dedicated kernel and full software *stack*.

environments for multiple software packages, manages dependencies, and configures the environment for use.

After creating a directory for the workspace and a ROS package within it, the starting repository 'my_bot' can be cloned. The next Subsection analyzes the repository structure, followed by a description of the robot and the organization of the node launch files.

### 3.2.1 Package Structure and Directory Organization

The preconfigured structure of the cloned package includes four distinct directories, each with specific files and a well-defined role. The main folders are:

- `config/`: contains configuration files, including node parameters and settings for simulation (Gazebo) and visualization (RViz) programs.
- `description/`: collects the robot description files in the formats described in Subsection 3.1.2; the content of this directory is detailed in Subsection 3.2.2.
- `launch/`: includes node launch files, including the Python file that aggregates them; Subsection 3.2.3 illustrates this directory in detail.
- `worlds/`: contains Gazebo simulator maps in `.world` format.

The `description` and `launch` directories are described in detail in the subsequent Sections, while `config` and `worlds` are introduced here.
The `config` directory collects parameter files, separating them from launch files and making the structure clearer through the use of `.yaml` files. The `worlds` directory contains various obstacle configurations saved from Gazebo, used later to compare simulated scenarios with 3D reconstruction results.

### 3.2.2 Personal Robot Description

To make the robot easily manageable and simple to understand, it was implemented in the basic, standard way, including the following components:

- chassis: parallelepiped to which all components are attached
- drive wheels: one per side, cylindrical in shape
- caster wheel: implemented as a hemispherical shape
- sensors: any sensors added based on various configurations

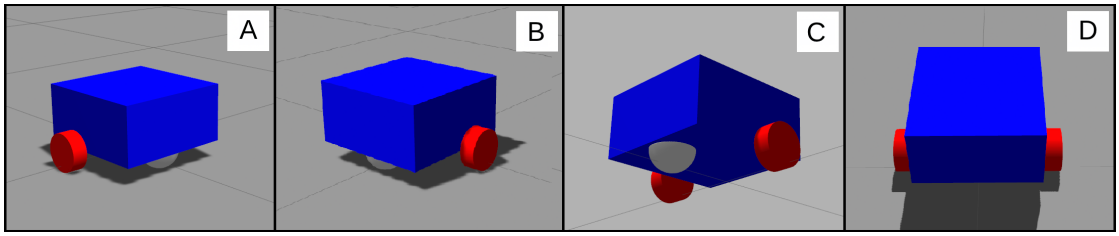



**Figure 3.9:** Visual representation of the robot developed for the project, shown from multiple viewpoints. The blue parallelepiped represents the chassis, the flattened red cylinders are the drive wheels, and the white sphere inscribed in the chassis represents the caster wheel.

Figure 3.9 shows a representation of the robot in its base configuration, without sensor integration.

To delve deeper into the specifics, it is necessary to discuss how this description occurs in code format. As already discussed in Subsection 3.1.2, a detailed description of the robot must be provided to perform simulations, and URDF and XACRO formats are used for this purpose.

The main robot description file, `robot.urdf.xacro`, serves as a modular orchestrator, allowing selective inclusion of the robot's components and control systems.

```
1  <?xml version="1.0"?>
2  <robot xmlns:xacro="http://www.ros.org/wiki/xacro" name="robot">
3
4      <xacro:arg name="use_ros2_control" default="true"/>
5      <xacro:arg name="sensors" default="depth_camera"/>
6
7      <xacro:include filename="robot_core.xacro" />
8
9      <xacro:if value="$(arg use_ros2_control)">
10         <xacro:include filename="ros2_control.xacro" />
11     </xacro:if>
12     ...
13     <xacro:if value="${'$(arg sensors)' == 'depth_camera'}">
14         <xacro:include filename="depth_camera.xacro" />
15     </xacro:if>
16 </robot>
```

**Listing 3.1:** Main robot.urdf.xacro file

From this file, the robot is controlled using the *ros2_control* framework through the `GazeboSystem` plugin, which exposes the hardware interfaces of the drive wheels. Regarding the robot's physical structure, the `robot_core.xacro` file is invoked, which is responsible for describing all the basic robot components, defining a total

of 5 links:

- `base_link`: main reference frame useful as a generic attachment point for all components

```
<link name="base_link"> </link>
```

**Listing 3.2:** Base link definition in robot_core.xacro

- `chassis`: parallelepiped $0.3 \times 0.3 \times 0.15$m with mass 0.5 kg and blue color, connected via joint to `base_link`

```
<link name="chassis">
    <visual>
        <origin xyz="0.15 0 0.075"/>
        <geometry>
            <box size="0.3 0.3 0.15"/>
        </geometry>
        <material name="blue"/>
    </visual>
    <collision>
        <geometry>
            <box size="0.3 0.3 0.15"/>
        </geometry>
    </collision>
    <xacro:inertial_box mass="${chassis_mass}" x="0.3" y="0.3" z="
    0.15">
        <origin xyz="0.15 0 0.075" rpy="0 0 0"/>
    </xacro:inertial_box>
</link>
```

**Listing 3.3:** Chassis definition in robot_core.xacro

- `left_wheel`: differential drive wheel, visually shaped as a red cylinder with mass 0.1 kg, length 0.04 m and radius 0.05 m, but implemented with a collision shape equal to a sphere of the same radius. This modification was necessary to minimize wheel sliding, as cylindrical wheels have a higher ground contact point and cause issues during simulation [30]. The wheels were decided to be connected via joint to `base_link`

```
<link name="left_wheel">
    <visual>
        <geometry>
            <cylinder radius="0.05" length="0.04"/>
        </geometry>
```

```
6           <material name="red"/>
7       </visual>
8       <collision>
9           <geometry>
10              <sphere radius="0.05"/>
11          </geometry>
12      </collision>
13      <xacro:inertial_cylinder mass="${wheel_mass}" length="0.04"
        radius="0.05">
14          <origin xyz="0 0 0" rpy="0 0 0"/>
15      </xacro:inertial_cylinder>
16  </link>
```

**Listing 3.4:** Left wheel definition in robot_core.xacro

- `right_wheel`: implemented identically but mirrored with respect to the left one
- `caster_wheel`: support wheel sized as a white sphere with radius 0.05 m, protruding halfway, with mass equal to the wheels 0.1 kg

```
1   <link name="caster_wheel">
2       <visual>
3           <geometry>
4               <sphere radius="0.05"/>
5           </geometry>
6           <material name="white"/>
7       </visual>
8       <collision>
9           <geometry>
10              <sphere radius="0.05"/>
11          </geometry>
12      </collision>
13      <xacro:inertial_sphere mass="${caster_wheel_mass}" radius="
        0.05">
14          <origin xyz="0 0 0" rpy="0 0 0"/>
15      </xacro:inertial_sphere>
16  </link>
```

**Listing 3.5:** Caster wheel definition in robot_core.xacro

Once the base of the robot and its movement components have been built, attention can shift to the sensory components. Four types of sensors were designed for the task, initially defined and thus available when needed:

- **camera**: front-mounted RGB camera on the chassis at $(0.305, 0, 0.08)$ m, resolution $640 \times 480$ px, FOV 1.089 rad ($\sim 62°$), visual range $0.05 - 8$ m,

update rate 10 Hz. Optical frame `camera_link_optical` aligned with ROS convention via rotation $(-90°, 0, -90°)$. Plugin `libgazebo_ros_camera.so` publishes `sensor_msgs/Image` on topic `/image_raw`.

```
<sensor name="camera" type="camera">
    <update_rate>10</update_rate>
    <camera>
        <horizontal_fov>1.089</horizontal_fov>
        <image>
            <width>640</width>
            <height>480</height>
        </image>
        <clip>
            <near>0.05</near>
            <far>8.0</far>
        </clip>
    </camera>
    <plugin name="camera_controller" filename="
   libgazebo_ros_camera.so">
        <frame_name>camera_link_optical</frame_name>
    </plugin>
</sensor>
```

**Listing 3.6:** RGB camera Gazebo sensor configuration

- **depth_camera**: identical to the RGB camera in terms of position/orientation, resolution, range, and field of view. Same plugin `libgazebo_ros_camera.so` but with parameters `<min_depth>` and `<max_depth>` set equal to the camera range for consistency. The plugin publishes `sensor_msgs/PointCloud2` on topic `/lidar_points`.

```
<sensor name="camera" type="depth">
    ...
    <plugin name="camera_controller" filename="
   libgazebo_ros_camera.so">
        <frame_name>camera_link_optical</frame_name>
        <min_depth>0.1</min_depth>
        <max_depth>100.0</max_depth>
    </plugin>
</sensor>
```

**Listing 3.7:** Depth camera key differences with RGB camera

- **2D lidar**: 2D laser scanner positioned at $(0.1, 0, 0.175)$ m (height 17.5 cm), 360 horizontal rays ($-180°$ to $+180°$), range $0.3 - 12$ m, 10 Hz. Cylinder r=0.05 m $\times$ l=0.04 m. Plugin `libgazebo_ros_ray_sensor.so` publishes `sensor_msgs/LaserScan` on topic `/scan`.

```
<joint name="laser_joint" type="fixed">
    <origin xyz="0.1 0 0.175" rpy="0 0 0"/>
</joint>
<sensor name="laser" type="ray">
    <ray>
        <scan>
            <horizontal>
                <samples>360</samples>
                <min_angle>-3.14</min_angle>
                <max_angle>3.14</max_angle>
            </horizontal>
        </scan>
        <range>
            <min>0.3</min>
            <max>12</max>
        </range>
    </ray>
</sensor>
```

**Listing 3.8:** 2D lidar ray configuration

- **3D lidar**: 3D laser scanner with same shape and in the same position as the 2D one, 360×32 rays (−15° to +15° vertical), identical range 0.3−12 m. Plugin `libgazebo_ros_ray_sensor.so` publishes `sensor_msgs/PointCloud2` on topic `/lidar_points`.

```
<sensor name="laser" type="ray">
    <ray>
        <scan>
            <horizontal>
                <samples>1060</samples>
                <min_angle>-3.14</min_angle>
                <max_angle>3.14</max_angle>
            </horizontal>
            <vertical>
                <samples>32</samples>
                <min_angle>-0.26</min_angle>
                <max_angle>0.26</max_angle>
            </vertical>
        </scan>
        ...
    </ray>
</sensor>
```

**Listing 3.9:** 3D lidar vertical scan

Considering that the two cameras and the two sensors share the same shape and position on the robot in pairs, the different configurations can be observed in Figure 3.10.

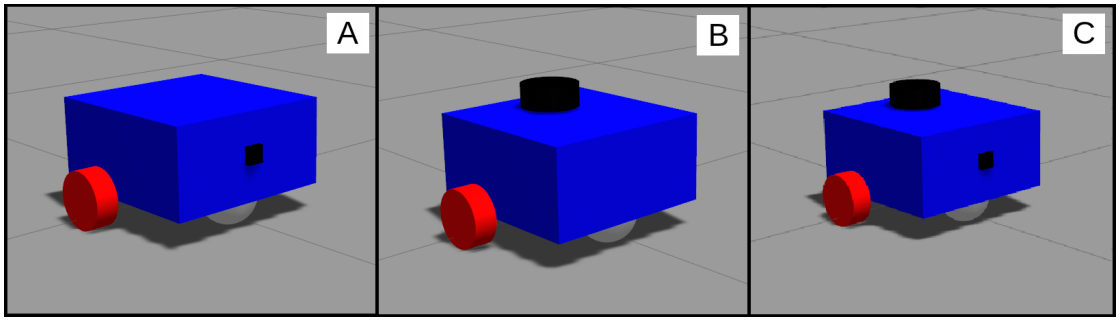



**Figure 3.10:** Sensor configurations on the robot: (A) with front-mounted camera, (B) with lidar depth sensor above, and (C) with a combination of the two

### 3.2.3 Project Launcher

The `launch/` folder contains Python files that enable coordinated startup of all project nodes, following the modular architecture of ROS2 Launch. The main file, or at least the one that starts the entire project, is named `launch_sim.launch.py` and integrates fundamental components:

1. **Robot State Publisher** (`rsp.launch.py`): loads the `robot.urdf.xacro` file via `xacro` to generate the complete robot description (URDF). The `robot_state_publisher` node publishes the /robot_description topic and the complete TF tree (See Listing 3.10).

```
robot_description_config = Command([
    'xacro ', xacro_file,
    ' use_ros2_control:=', use_ros2_control,
    ...
])

node_robot_state_publisher = Node(
    package='robot_state_publisher',
    executable='robot_state_publisher',
    output='screen',
    parameters=[{'robot_description': robot_description_config, '
   use_sim_time': use_sim_time}]
)
```

**Listing 3.10:** RSP node in rsp.launch.py

```
rsp = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
        get_package_share_directory(package_name),'launch','rsp.launch.py'
    )]),
    launch_arguments={
        'use_sim_time': 'true',
        'use_ros2_control': 'true',
        ...
    }.items()
)
```

**Listing 3.11:** RSP launch in launch_sim.launch.py

2. **Gazebo Simulator**: starts the simulator via `gazebo.launch.py` from the `gazebo_ros` package, loading custom parameters from `gazebo_params.yaml` including a map of obstacles (`world`) (See Listing 3.12). The `gazebo_ros/spawn_entity.py` spawner positions the robot in the world with name `tizio_bot` using the /robot_description topic(See Listing 3.13).

```
gazebo_params_file = os.path.join(get_package_share_directory(package_name),'config','gazebo_params.yaml')

gazebo = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
        get_package_share_directory('gazebo_ros'), 'launch', 'gazebo.launch.py')]),
    launch_arguments={'extra_gazebo_args': '--ros-args --params-file ' + gazebo_params_file}.items()
)
```

**Listing 3.12:** Gazebo simulator launch in launch_sim.launch.py

```
spawn_entity = Node(package='gazebo_ros', executable='spawn_entity.py', arguments=['-topic', 'robot_description', '-entity', 'tizio_bot'], output='screen')
```

**Listing 3.13:** Gazebo robot spawning in launch_sim.launch.py

3. **ROS2 Control Stack**: two spawners from the `controller_manager` activate the controllers configured in `my_controllers.yaml`:

   - `diff_cont`: differential controller for robot movement
   - `joint_broad`: joint state broadcaster, responsible for publishing the current states of all robot joints on standard ROS topics

```
diff_drive_spawner = Node(
    package="controller_manager",
    executable="spawner",
    arguments=["diff_cont"],
)

joint_broad_spawner = Node(
    package="controller_manager",
    executable="spawner",
    arguments=["joint_broad"],
)
```

**Listing 3.14:** Controller spawners in launch_sim.launch.py

4. **Teleoperation**: `joystick.launch.py` starts `joy_node` (joystick reading) and `teleop_twist_joy/teleop_node`, which converts joystick commands to /diff_cont/cmd_vel_unstamped, configured via `joystick.yaml`. This node was added mostly for convenience; the alternative is to launch existing ROS nodes for keyboard control.

```
joystick = IncludeLaunchDescription(
    PythonLaunchDescriptionSource([os.path.join(
        get_package_share_directory(package_name),'launch','
    joystick.launch.py'
    )]),
    launch_arguments={'use_sim_time': 'true'}.items()
)
```

**Listing 3.15:** Joystick teleoperation launch in launch_sim.launch.py

5. **Visualizator**: RViz2 is automatically launched with a specific configuration file that pre-configures all necessary displays (robot model, TF tree, sensor data visualization, maps).

```
def launch_rviz(context, *args, **kwargs):

    rviz_config = os.path.join(
        get_package_share_directory(package_name), 'config',
    rviz_file
    )

    rviz_node = Node(
        package='rviz2',
        executable='rviz2',
        arguments=['-d', rviz_config],
```

```
        output='screen'
    )
    return [rviz_node]

rviz_launcher = OpaqueFunction(function=launch_rviz)
```

**Listing 3.16:** RViz launch in launch_sim.launch.py

Once all components have been defined and enabled, synchronizing them using the `use_sim_time:=true` parameter and Gazebo's simulator clock, all that remains is to launch them through ROS2's main launch file.

```
return LaunchDescription([
    rsp,
    joystick,
    gazebo,
    spawn_entity,
    diff_drive_spawner,
    joint_broad_spawner,
    rviz_launcher
])
```

**Listing 3.17:** Final launch sequence in launch_sim.launch.py

## 3.3 Sensor Integration and Perception Validation

Before proceeding to the study of 3D reconstruction using NvBlox and its implementation in the project, it is essential to verify that all data required by the algorithm can be correctly acquired in real-time. The sensors described earlier must therefore function properly, not only collecting the appropriate data but also publishing it via topics in a format suitable for visualization in RViz. This Subsection examines the robot's sensor features to confirm their correct operation from both perception and visual fidelity perspectives.

First, the obstacle map used in this Section must be introduced. As specified in Subsection 3.2.1, the `worlds` directory contains the obstacle maps used in simulation. Initially, for testing, the map shown in Figure 3.11 is used, where the robot is spawned as described in Subsection 3.2.3; later, for 3D reconstruction, a slightly different map enclosed by walls will be employed to have a closed environment.

Starting with the non-depth RGB camera, positioned frontally on the robot as previously mentioned, Figure 3.12 displays screenshots from both the simulator and visualizer when launching the program with only the camera mounted. In RViz, the captured image appears in a separate window, as it cannot be represented in the robot's 3D space. The simulator also shows the captured image; rays emanating from the camera's optical frame indicate its field of view, and a spatial projection of

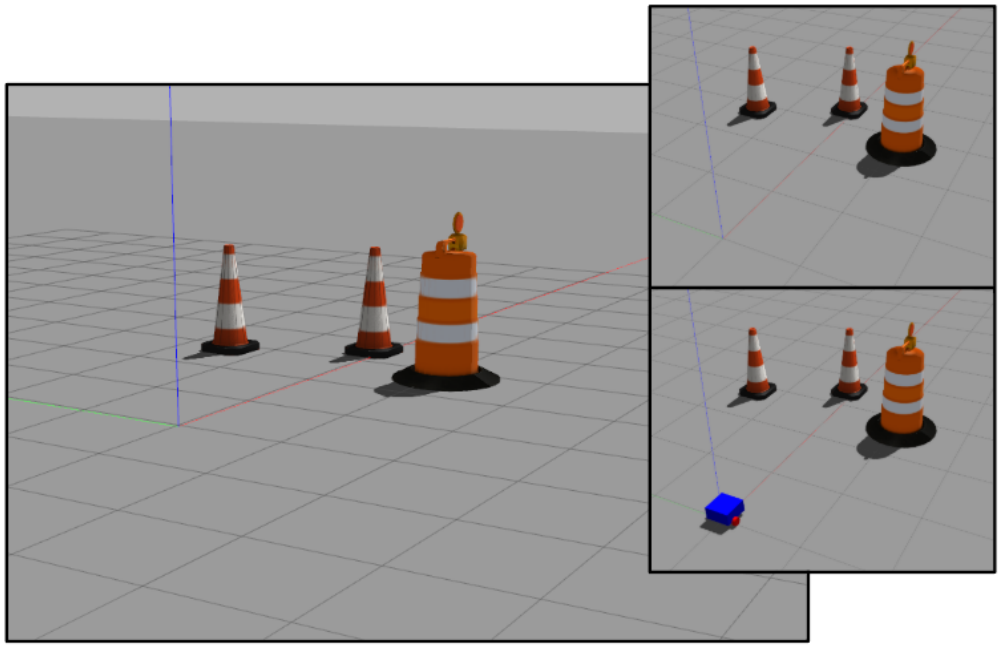


**Figure 3.11:** Simple map ("cones.world") with 3 obstacles used for initial tests. The side smaller images show higher view and robot spawn.

the image ahead of the robot can be viewed to understand the sensor's observation directly in simulation.

The depth camera, on the other hand, captures information beyond simple 2D spatial observation. Figure 3.13 shows only the visualizer screen during depth camera use. The depth data is represented in RViz as a point cloud positioned in space at the distances and directions detected by the camera. Notably, the image reveals a wall behind the cones, which appears due to the `<max_depth>` parameter set to 8 meters, limiting the shared sensor data accordingly. As observed in the figure, the depth camera adds depth information without replacing the standard camera's output; the 2D image in the bottom-left corner of the visualizer confirms that the original data persists and is used to color the point cloud, which would otherwise be monochromatic as shown in the smaller picture.

Considering the true depth sensors, the 2D lidar, operating in a plane parallel to the floor at the sensor's height on the robot, is analyzed first. Figure 3.14 illustrates the points on its plane corresponding to obstacles as displayed in the visualizer. The image also includes a version overlaying these lidar points with a reduced-size point cloud from the previous depth camera, rendered blurred and in the background. The same points are visible, but their positions are clearer, revealing they are not at floor level but at the depth sensor's mounting height.

The last examined sensor is the 3D lidar . Figure 3.15 clearly shows the 32

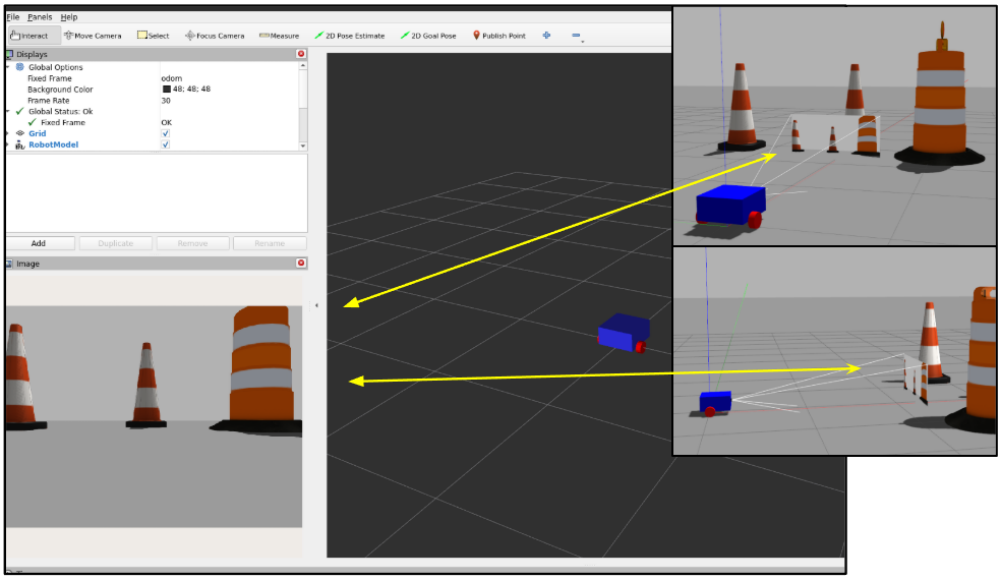


**Figure 3.12:** Sensor Configuration: RGB camera. Visualizer in background, simulator in foreground from two viewpoints. 2D images match in both.

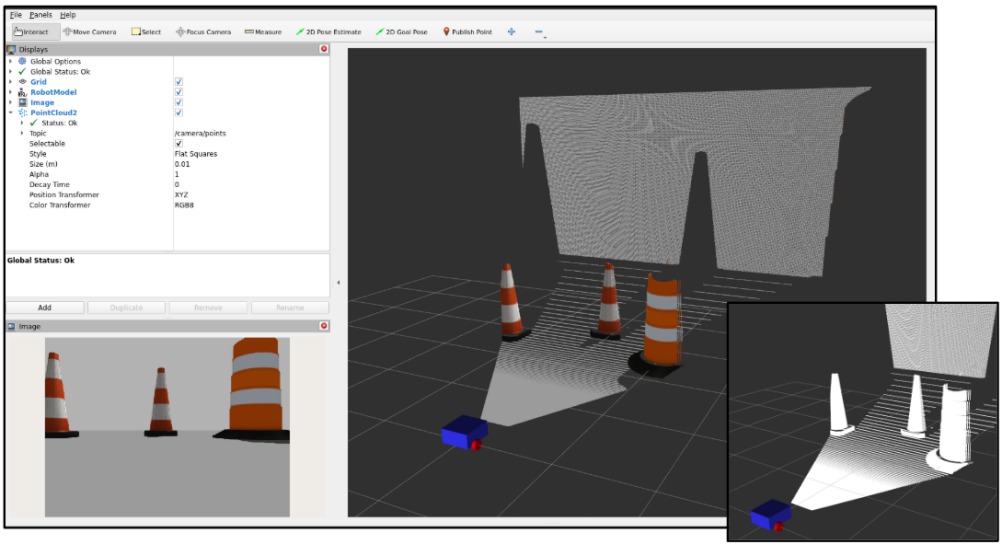


**Figure 3.13:** Sensor Configuration: depth camera. Visualizer shows 2D image bottom-left and colored 3D point cloud. Monochromatic representation reveals depth-only data without color.

concentric rings of data observed by the sensor, represented as a monochromatic point cloud, similar to the depth camera data but spanning 360° horizontally and 30° vertically. The information is more sparse but covers a much larger space than

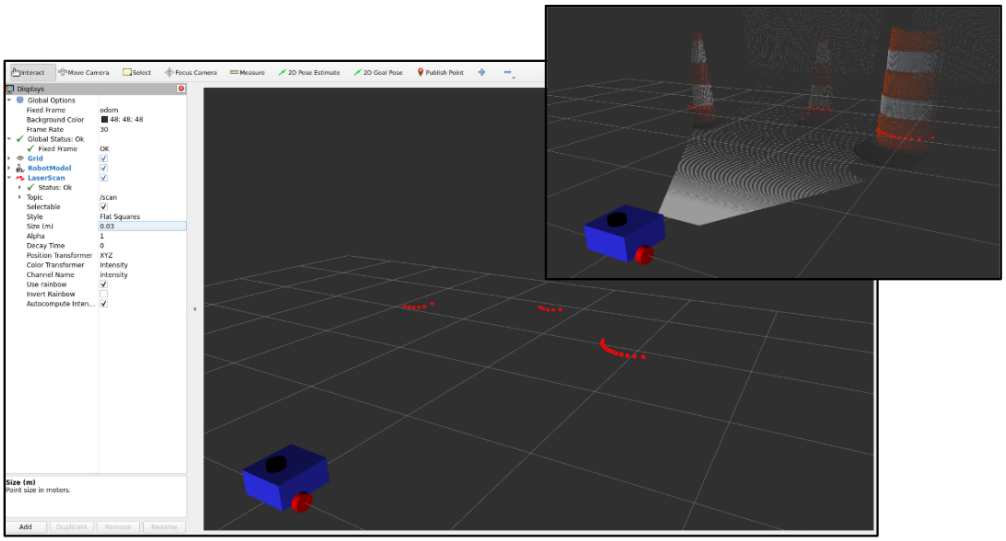


**Figure 3.14:** Sensor Configuration: 2D lidar. Visualizer displays lidar points; blurred full depth camera point cloud in small image clarifies obstacle positions and sensor capabilities.

that of the depth cam, where the rays are all concentrated in the same space of the 2D image produced.

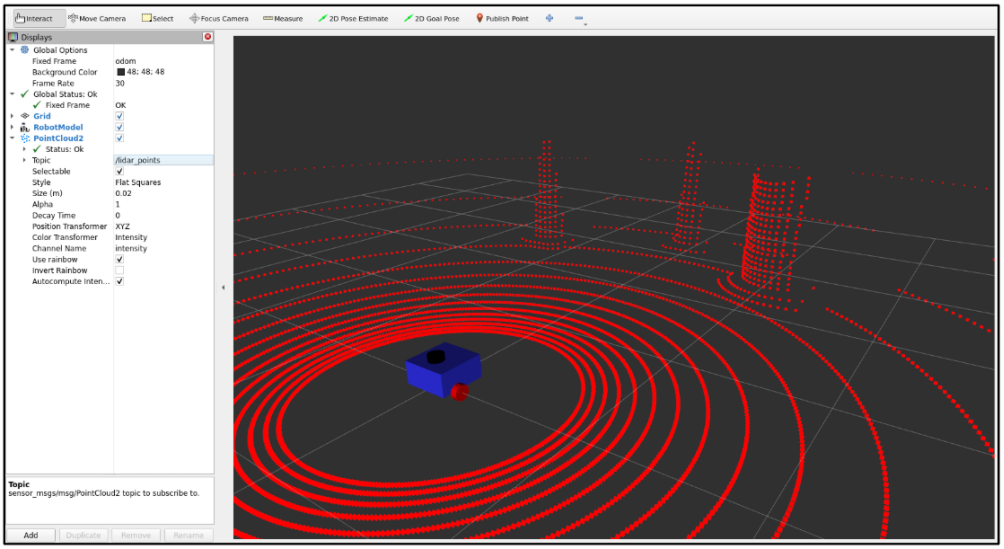


**Figure 3.15:** Sensor Configuration: 3D lidar. Point cloud shows 32 vertical rings. The central blind spot (`min_depth` parameter) prevents the sensor from detecting the robot's own chassis as an obstacle.

### Autonomous Navigation Demo with SLAM and Nav2

Following the technical study presented at the beginning of this chapter and the verification of the individual sensors, it is possible to use the data collected by them to tackle a first navigation task. The goal is to assess whether the robot has a consistent perception of the surrounding space and of its own motion within it. The first step consists in building a two-dimensional map of the obstacles, obtained by using SLAM (see Subsection 3.1.4) in *mapping* mode. Figure 3.16 shows an initial map generated with the robot equipped with only the 2D lidar, controlled via joystick. This preliminary result makes it possible to verify both the consistency and the synchronization between the depth data acquired by the sensor and the motion information provided by odometry (see Subsection 3.1.4).

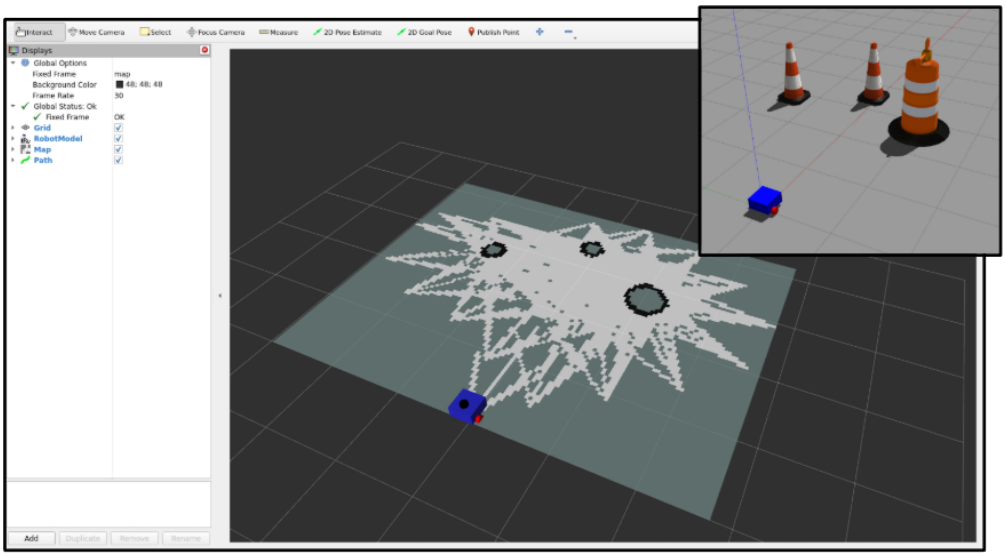


**Figure 3.16:** Initial static map obtained with SLAM Toolbox in mapping mode using a 2D lidar.

As described in Subsection 3.1.4, the data structure underlying motion planning is the *costmap*. Starting from the static map obtained with the SLAM Toolbox in *mapping* mode, Nav2 can build the corresponding costmap. Figure 3.17 shows, inside RViz, the computed costmap using, in this case, the 3D lidar configuration, in order to relate the 2D map on the plane with the point cloud that represents the obstacles in three-dimensional space.

The resulting map can finally be used to allow the robot to plan and follow a safe path toward a given goal. Figure 3.18 shows an example of a trajectory computed from a target pose (position and orientation). It can be observed that the costmap leads to a path that goes around the obstacle while maintaining a safety margin, significantly reducing the probability of collision along the route.

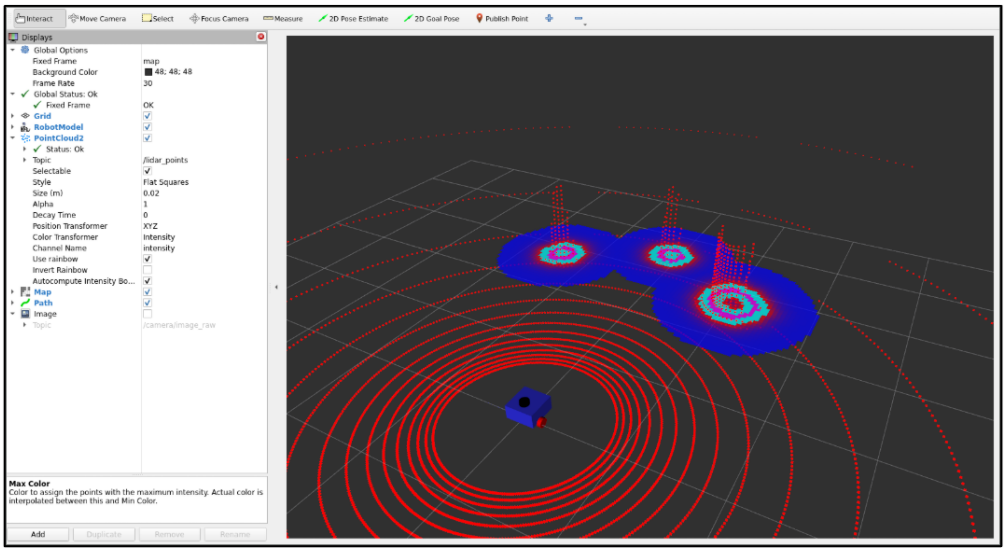



**Figure 3.17:** Nav2 Costmap and 3D lidar point cloud.

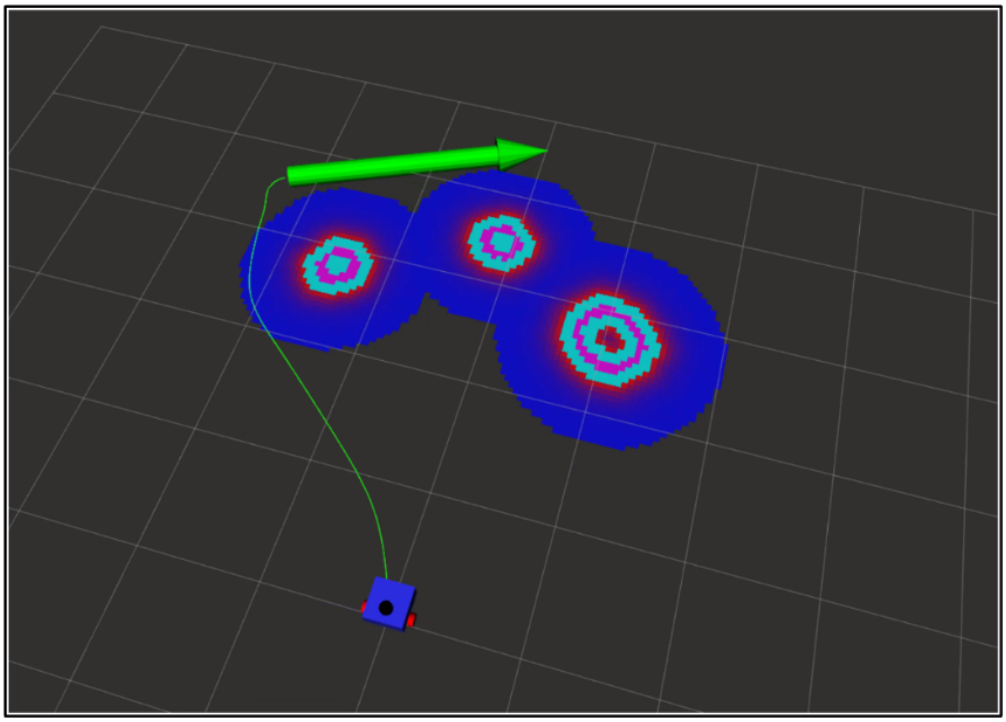


**Figure 3.18:** Demonstration of autonomous navigation with Nav2: the green arrow represents the position and orientation target assigned by the user, while the green path is the trajectory computed by Nav2 for safe navigation.

# Chapter 4

# Selected 3D Reconstruction Framework: NvBlox

In the previous chapters, both the theoretical background of the adopted architectures and the detailed description of the project developed in **ROS 2** have been presented, with particular focus on the software architecture, robot modeling, and sensor integration. This groundwork, which is essential for approaching the advanced integration of **3D reconstruction**, has been strongly challenged by the computational demands of the **NvBlox** algorithm, which is studied in this thesis and used for volumetric 3D reconstruction. The first Subsection of the present Section is devoted to the design choices that made NvBlox practically usable within the project, while the subsequent ones will focus on code modifications and the simulation experiments.

## 4.1 Container-based Integration of NvBlox in the ROS 2 Project

During the integration process, two main strategies were considered to achieve a correct and functional implementation of **NvBlox**. The first option was to use the **NvBlox** library directly as a standalone framework for GPU-accelerated volumetric 3D reconstruction, without relying on ROS-specific components. At first glance, this solution represented the lowest-level approach, offering maximum control over the calls to the library.

However, direct adoption of the library quickly revealed several issues: complex management of **CUDA** dependencies, sensitivity to NVIDIA library versions, and the lack of native integration with **ROS 2**. These difficulties are consistent with what is reported in the official documentation and in various technical discussions,

and they would have required a significant additional effort to develop custom ROS nodes, message adapters, and dedicated build procedures . For these reasons, this approach was deemed inconvenient with respect to the goals and time constraints of the project.

As an alternative, the solution proposed directly by NVIDIA through the `isaac_ros_nvblox` package was considered. This package fully encapsulates the **NvBlox** library and exposes it as a ready-to-use component within the **ROS 2** ecosystem. The **Isaac ROS** environment is officially supported and tested by NVIDIA, making it one of the most recommended options for advanced robotic applications that require GPU acceleration and integration with heterogeneous sensors. In this way, NvBlox can be used through preconfigured nodes and launch files, significantly reducing the complexity of the integration.

It is nevertheless important to underline the trade-offs of this choice. Isaac ROS is distributed and used through a **Docker container** (see Subsection 3.1.5) that includes a large set of libraries, tools, and dependencies, **NvBlox** being only one of many modules provided by the package. Consequently, the environment is substantially heavier in terms of memory usage and disk space, with image sizes that can easily reach tens of gigabytes. Even if, during development, only a subset of the available packages is actually employed, the container image still contains everything needed to install and support the full Isaac ROS stack, with a non-negligible impact on resources.

Despite selecting this implementation, careful attention must still be paid to the available computational capabilities. The development setup utilized in this project is an NVIDIA GeForce RTX 2060 with 6 GB of VRAM, which met minimum requirements but neared practical limits for larger scenes—NvBloxdocumentation recommends 8+ GB for optimal performance with high-resolution maps [31]. This configuration operates at the edge of supported limits, requiring moderation in reconstruction resolution and map sizes to avoid excessive processing demands that could result in project crashes and to generally ensure stable performance on this hardware.

### 4.1.1 Container Setup, NvBlox Installation and Project Sharing

The development environment was therefore based on a **NVIDIA Isaac ROS** container, which includes a compatible **ROS 2** distribution, the **CUDA** support required for GPU acceleration, and the main tools needed to work with Isaac ROS packages. By importing the `isaac_ros-dev` package folder, the container can be built and launched, while preserving access to configuration files and build scripts, which remain fully editable and customizable.

Before building the container, some operational parameters are configured, such

as the container name and its persistence mode, in order to prevent automatic removal at shutdown and to support an iterative development workflow. Using the same volume-mounting mechanism described in Subsection 3.1.5, the ROS project folder introduced in the previous chapter is mounted inside the container, ensuring that the project files and their modifications are visible and synchronized both on the host and within the containerized environment. This setup allows the code to be version-controlled and managed on the host machine, while still benefiting from the execution environment provided by Isaac ROS.

The installation of **NvBlox** is then performed directly inside the Isaac ROS container, which supplies all the dependencies required for the correct operation of the package, significantly reducing the risk of incompatibilities between drivers, CUDA, and NVIDIA libraries. Once the environment configuration is completed (including the installation of the simulator, the robot control nodes, and the joystick management nodes), the project is compiled using `colcon`, yielding a fully functional ROS 2 workspace inside the container. From a logical and functional perspective, the robot behaves as if it were on the host system, with the important difference that all the computational load related to **NvBlox** and GPU processing is handled in an isolated and controlled container environment.

With this container-based setup in place, the integration of **NvBlox** at the code level can now be addressed, detailing how the existing project has been extended and adapted to incorporate volumetric 3D reconstruction into the overall ROS 2 architecture.

## 4.2 Configuration and Launch Arguments

Before integrating **NvBlox** into the project, it was useful to introduce two configuration arguments to make the system more flexible and reduce the number of manual modifications required for each new simulation.

These arguments, passed to the various *launch files*, allow compact selection of both **NvBlox**'s operating mode and the robot's sensor configuration. The two parameters introduced are `nvblox_mode` and `sensors`, referring respectively to **NvBlox**'s operating mode and the sensors activated on the robot.

As shown in Listing 4.1, the `nvblox_mode` argument enables selection among four operating modes of **NvBlox** [32]:

1. **Static**: **NvBlox** performs reconstruction assuming the scene is completely static, without moving obstacles or people. In this base mode, the `TSDF`, `color`, `mesh`, and `ESDF` layers are built independently but perfectly co-localized. This is the only mode actually used in this project.

2. **People Segmentation**: **NvBlox** integrates a semantic segmentation network capable of producing a pixel mask associated with the "person" class. This

mask can be used to exclude pixels corresponding to people during integration, reducing their impact on the map.

3. **People Detection**: In this mode, image regions corresponding to human subjects are identified and treated separately in the reconstruction process, increasing map robustness in the presence of moving people.

4. **Dynamic**: **NvBlox** maintains an explicit free space layer used to detect moving objects. When an object enters a region previously classified as free, it is labeled as dynamic and integrated into the dynamic objects layer, similar to the people case.

```
1     nvblox_mode_arg = DeclareLaunchArgument(
2         'nvblox_mode',
3         default_value='static',
4         description='Mode for NvBlox (static, people segmentation,
      people detection, or dynamic)'
5     )
6
7     sensors_arg = DeclareLaunchArgument(
8         'sensors',
9         default_value='depth_camera',
10        description='Which sensors to enable (depth_camera,
     lidar_3d)'
11    )
```

**Listing 4.1:** Arguments Declaration

The `sensors` argument, instead, controls the active sensor configuration at project launch.

The currently supported options are:

1. **depth_camera**

2. **lidar_3d**

The last configuration, based on LiDAR sensor, also includes a front-mounted RGB camera on the robot, used to acquire color information on obstacles. This choice was introduced primarily to enable more immediate visual comparisons between different reconstruction configurations, adding color to reconstructions from non-RGB depth sensor that would otherwise remain monochromatic. It is worth specifying that the RGB camera is not used for reconstruction itself, but only to color the voxels reconstructed via LiDAR.

The `nvblox_mode` and `sensors` arguments do not merely control algorithms or reconstruction modes, but are also used to coordinate the behavior of various project components.

For example, the `sensors` value is passed to the `robot_state_publisher` (see Subsection 3.1.2) to generate the robot description with the correct set of mounted sensors; the same argument is used in the **NvBlox** launcher to select the appropriate topic remapping (see Subsection 4.3.1), which depends on the sensor and is necessary to achieve a correct reconstruction.

## 4.3 NvBlox Integration in the ROS2 Project

To integrate **NvBlox** into the ROS 2 project, a dedicated *launch file* was created in the `launch/` directory (described in Subsection 3.2.3) and subsequently included in the project's main launch file.

This way, the **NvBlox** node starts automatically along with the other system components.

The dedicated launch file is named `nvblox.launch.py` and handles three main aspects:

- management of **NvBlox** configuration parameters based on the selected `nvblox_mode` argument;
- definition of topic remapping between the project and what **NvBlox** expects to receive;
- declaration and launch of the `nvblox_node`.

Regarding parameters, a `nvblox/` subfolder was added to the `config/` folder, containing:

- a base file `nvblox_base.yaml` with configuration shared across all modes;
- a `specializations/` subdirectory including files `nvblox_detection.yaml`, `nvblox_dynamics.yaml`, and `nvblox_segmentation.yaml`, used in addition to extend configuration for the corresponding modes.

In `static` mode, only the base parameters are loaded, while advanced modes also aggregate the specific parameters from the `specializations/` directory.

### 4.3.1 Topic Remapping for NvBlox Integration

**NvBlox** expects a predefined set of *topic* names to receive input data, such as point clouds, RGB or depth images, odometry, and calibration parameters.

Since the *topic* names used in the project do not match those required by **NvBlox**, explicit remapping was introduced, associating each project *topic* with the corresponding name expected by the reconstruction node.

Moreover, the *topic* names used by **NvBlox** are defined as *relative* paths (e.g., `pointcloud` instead of `/pointcloud`),[1] which makes the adaptation phase between the two naming schemes even more critical.

Two remapping strategies were defined, corresponding to the two configurations actually used for 3D reconstruction:

- reconstruction based on **3D LiDAR**;
- reconstruction based on **depth camera**.

### 3D LiDAR-Based Reconstruction

In the case of 3D reconstruction based on LiDAR sensor, **NvBlox** expects to receive the point cloud on the relative *topic* `pointcloud`, while the project's 3D LiDAR publishes data on the absolute *topic* `/lidar_points` (see Subsection 3.2.2). Thus, `/lidar_points` must be remapped to `pointcloud`.

As anticipated, the LiDAR configuration includes a front RGB camera, requiring additional remapping of image and `camera_info` *topics*. **NvBlox** handles multiple cameras by numbering them (e.g., `camera_0`, `camera_1`, etc.); this work uses only `camera_0`.

The adopted remapping is as follows (left: names expected by **NvBlox**, right: *topics* actually used by sensors):

```
remappings_lidar_3d = [
    ('pointcloud', '/lidar_points'),
    ('camera_0/color/image', '/camera/image_raw'),
    ('camera_0/color/camera_info', '/camera/camera_info'),
    ('odom', '/odom'),
]
```

**Listing 4.2:** Topic remapping for 3D LiDAR-based reconstruction (left: NvBlox topics, right: project topics)

### Depth Camera–Based Reconstruction

The alternative configuration, using a depth camera, requires slightly different remapping. In this case, the depth camera provides both depth images and RGB images; the color-related *topics* remain unchanged from the LiDAR case, while

[1] In ROS 2, *topics* can be specified as absolute names (prefixed with `/`) or relative. Relative names are resolved from the namespace and node name using them. **NvBlox** adopts this convention for greater flexibility in the context where it is inserted; in the project described here, sensor-exposed *topics* are instead defined as absolute names (e.g., `/lidar_points`), making explicit remapping necessary even when the name does not change (e.g., `/odom` becoming `odom`).

depth *topics* are added and remapped to the relative names expected by **NvBlox**. Here too, the left column shows *topic* names in **NvBlox**'s convention, while the right column lists the sensors' default *topics*:

```
remappings_depth = [
    ('camera_0/depth/image', '/camera/depth/image_raw'),
    ('camera_0/depth/camera_info', '/camera/depth/camera_info'),
    ('camera_0/color/image', '/camera/image_raw'),
    ('camera_0/color/camera_info', '/camera/camera_info'),
    ('odom', '/odom'),
]
```

**Listing 4.3:** Topic remapping for depth camera–based reconstruction (left: NvBlox topics, right: project topics)

The remapping presented in this Section thus adapts **NvBlox**'s interface to the project's *topic* scheme without modifying the internal logic of the sensor publishing nodes.

### 4.3.2 NvBlox Node

Once **NvBlox** is installed within the Isaac ROS container, the reconstruction node can be launched directly from NVIDIA's `nvblox_ros` package. The node declaration in the *launch file* `nvblox.launch.py` is as follows:

```
    nvblox_node = Node(
        package='nvblox_ros',
        executable='nvblox_node',
        name='nvblox_node',
        output='screen',
        parameters=parameters,
        remappings=chosen_remappings
    )
```

**Listing 4.4:** NvBlox Node Declaration

The node receives both parameters (loaded from `.yaml` files) and the *topic* remapping list (`chosen_remappings`), selected based on the sensor configuration chosen via the `sensors` argument.

For the node to actually start with the rest of the system, the `nvblox.launch.py` file must be included in the main *launch file* `launch_sim.launch.py`, within the `LaunchDescription`, also passing the previously defined arguments:

```
    nvblox = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory(package_name), 'launch', '
    nvblox.launch.py'
        )]),
```

```
        launch_arguments={
            'mode': LaunchConfiguration('nvblox_mode'),
            'sensors': LaunchConfiguration('sensors')
        }.items()
    )
    ...
    ...
    ...
    return LaunchDescription([
        nvblox_mode_arg,
        sensors_arg,
        rsp,
        gazebo,
        spawn_entity,
        ...
        nvblox,
        ...
    ])
```

**Listing 4.5:** NvBlox launcher inclusion

In this way, **NvBlox** is automatically started every time the complete project is launched, maintaining a configuration consistent with the other nodes of the system.

### 4.3.3 Key NvBlox Parameters

Inside the `nvblox_base.yaml` file there are numerous **NvBlox** configuration parameters. In this work, particular focus has been placed on the following:

- `voxel_size`: defines the voxel size (side of the cube) in meters; smaller values increase the resolution of the reconstruction but also the computational cost.
- `num_cameras`: specifies the number of cameras considered by **NvBlox**; in the project, only one camera (`camera_0`) is used.
- `global_frame`: indicates the global reference frame to which the reconstruction is anchored. By default it is `odom`, but if some other nodes want to work in a global map reference, it is possible to connect the **NvBlox** reconstructions to `map` too.
- `decay_tsdf_rate_hz` and `decay_dynamic_occupancy_rate_hz`: have been set to `0.0` to prevent the reconstruction from "fading" over time, instead maintaining the static map once built.
- `use_color`, `use_depth`, `use_lidar`: enable the use of color information, depth images, and LiDAR data in the reconstruction, respectively, allowing **NvBlox** to adapt to the selected sensor configuration.

- `lidar_min_valid_range`, `lidar_max_valid_range`, `lidar_width` and `lidar_height`: define the LiDAR sensor characteristics from **NvBlox**'s perspective, so that the internal sensor model matches the simulated device.

The correct setting of these parameters is fundamental to obtain a reconstruction consistent with the geometry of the environment and the real capabilities of the considered sensors. In the following Sections it will be shown how these choices reflect on the experimental results obtained in simulation.

### 4.3.4 Real-Time and Offline Reconstruction Visualization

Once the integration of the **NvBlox** algorithm within the project described in the previous chapters is completed, it becomes necessary to have a visualization capable of updating in real time with the generated reconstruction. As already explained in Subsection 3.1.3, the **RViz** visualizer allows displaying simultaneously the robot's movement in space and the 3D reconstruction updated based on data coming from the sensors. **NvBlox** continuously processes such data and, with each new frame, adds or updates the voxels that compose the volumetric reconstruction.

The information produced by **NvBlox** is published on several *topics* that share the prefix `/nvblox_node`, covering both temporary and global representations: from voxel (cubes) reconstructions to mesh (surfaces) representations, from monochromatic versions to colored versions, with the possibility of selecting different color maps. The various visualization modes were analyzed using, as a case study, the simple map with cones shown in Figure 3.11, in which the robot is initialized already oriented towards the obstacles to be mapped. In this first analysis, focus is placed on the reconstruction obtained from the robot's "first glance", that is, without movement and therefore without integration from multiple viewpoints, to highlight the different representation forms rather than the completeness of the map.

Figure 4.1 shows the standard visualization mode provided by **NvBlox** using a depth camera. The reconstruction is shown through voxels with size equal to the value set in the parameter file (see Subsection 4.3.3), colored based on the information acquired from the RGB camera. Compared to the simple point cloud obtained from the **depth_camera** configuration (Figure 3.13), here what the camera "sees" instant by instant is not represented directly, but through a persistent volumetric map: the camera data is integrated by **NvBlox**, which inserts the cubes in the positions corresponding to the observed surfaces and keeps them in space, so that, with the robot's movement, the environment can be progressively reconstructed even if it is not always in view. This representation is published, for example, on the *topic* `/nvblox_node/color_layer` and is viewable in RViz through the *display* `NvbloxVoxelBlockLayer`.

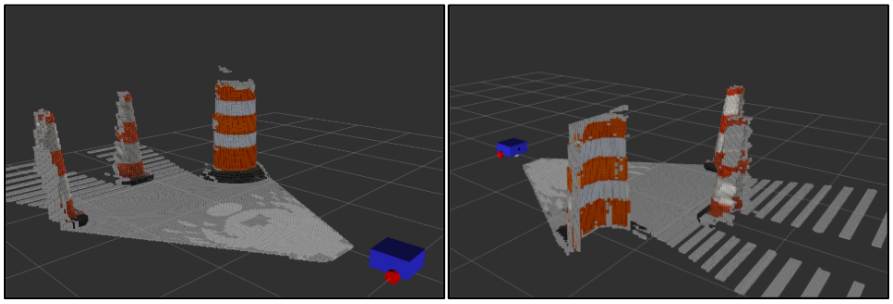


**Figure 4.1:** Standard voxel-based reconstruction in RViz: front and back views of the volumetric map generated by NvBlox.

Other variants of the representation in RViz are shown in Figure 4.2, where both a monochromatic version, obtained for example from the *topic* `/nvblox_node/tsdf_layer`, and a version in the form of a **mesh** can be observed. The mesh is a surface representation obtained by replacing the voxels with small "flat sheets" that occupy the same space, but are connected to each other contiguously. This way, the reconstruction looks closer to the real geometry of the scene, with sharper and less "step-like" edges, especially when the voxel size is small enough to ensure a good level of detail. This reconstruction is published on the topic `/nvblox_node/mesh` viewable in RViz through the *display* `NvbloxMesh`.

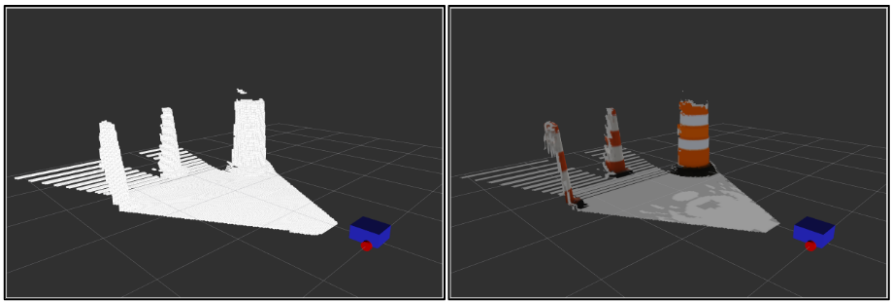


**Figure 4.2:** Alternative visualizations in RViz: TSDF mono-layer (left) and mesh-based surface reconstruction (right) generated from the same scene.

Once the robot has completed its path and the reconstruction phase, it is useful to save the obtained map. RViz provides dynamic real-time visualization, but does not directly handle the storage of results. For this task **NvBlox** provides terminal commands that allow exporting the reconstruction in `.ply` format (*Polygon File Format*), compatible with numerous 3D visualization tools, such as **MeshLab**, a lightweight but very flexible software for inspecting 3D models. MeshLab offers different visualization modes, with parameters easily modifiable by the user; to explore them, the saved reconstructions just visualized in RViz will be used,

imagining the reconstruction as completed. Figure 4.3 shows the main MeshLab window with the loaded reconstruction: the cubic composition can be observed in detail, while in the side panel different visualization variants are available, including monochromatic or grayscale schemes.

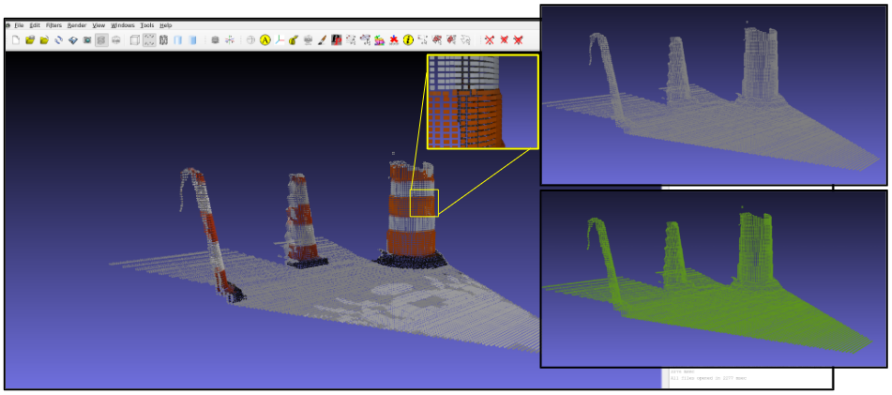


**Figure 4.3:** Loaded NvBlox voxel-based reconstruction in MeshLab with different shading and color options.

An additional visualization mode is **wireframe**, in which the reconstruction is represented as a network of segments that outline the surface polygons. Figure 4.4 shows the level of detail achievable with this representation, which can be useful, for example, to analyze the geometric structure of the mesh in view of a future extension towards collision modeling or object analysis in dynamic scenarios.

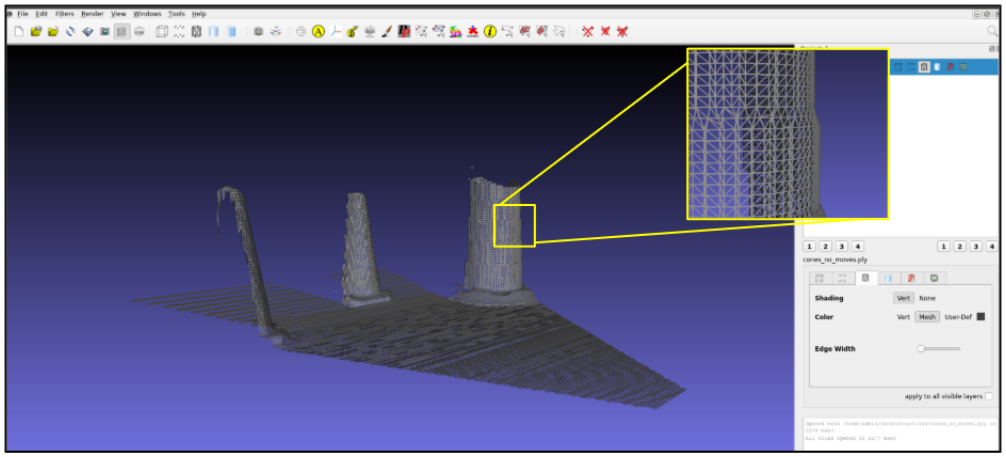


**Figure 4.4:** Wireframe visualization of the reconstructed scene in MeshLab, highlighting the polygonal structure of the mesh.

Finally, MeshLab also allows a full **mesh** visualization, as illustrated in Figure 4.5,

possibly combined with black and white representations or different color palettes. This mode highlights continuous surfaces more effectively and is particularly effective evalulating the overall quality of the reconstruction.

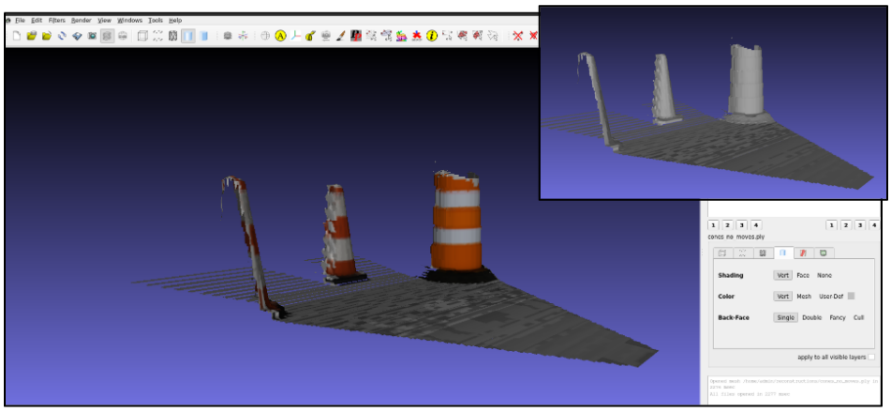


**Figure 4.5:** Surface mesh visualization of the reconstructed environment in MeshLab, including grayscale rendering options.

The availability of multiple visualization modes in an external environment like MeshLab is therefore very useful, as it allows analyzing the same reconstruction from different perspectives, without imposing constraints on the mode in which the map was saved: all necessary information is contained in the `.ply` file, and the various representations are obtained by modifying only the rendering options.

Having concluded this overview of real-time visualization modes (via **RViz**) and *offline* (via **MeshLab**), the next Section will present more extensive and complex reconstructions, obtained in dedicated maps. In particular, the performance of the different sensor configurations will be compared during the robot's movement, analyzing the fidelity of the 3D reconstructions compared to the simulated environment and the GPU effort.

## 4.4 Simulation results

For this Section, a simulated environment in **Gazebo** was created that reproduces the structure of a closed room. Compared to the maps previously used for qualitative demonstrations of sensors and reconstruction, here it is necessary to have an environment delimited by walls, with a higher density of internal objects, so as to be able to evaluate reconstruction performance in a more structured scenario, but still of limited dimensions and manageable in simulation. The memory required by **NvBlox** to maintain all voxels of the reconstruction is not negligible: resource consumption depends directly on the voxel size and the extent of the mapped

environment. To be able to make comparisons even with relatively small voxels, it is therefore appropriate not to exceed the complexity and size of the scene, so as to avoid overloading the **NvBlox** node.

The map used in this Section is shown in Figure 4.6 from different angles. The lower part, corresponding to the robot's spawn point, is intentionally freer, while in the upper area there is a higher concentration of obstacles and adjacent objects, so as to resemble the layout of a real room and provide sufficient variety of elements for reconstruction.

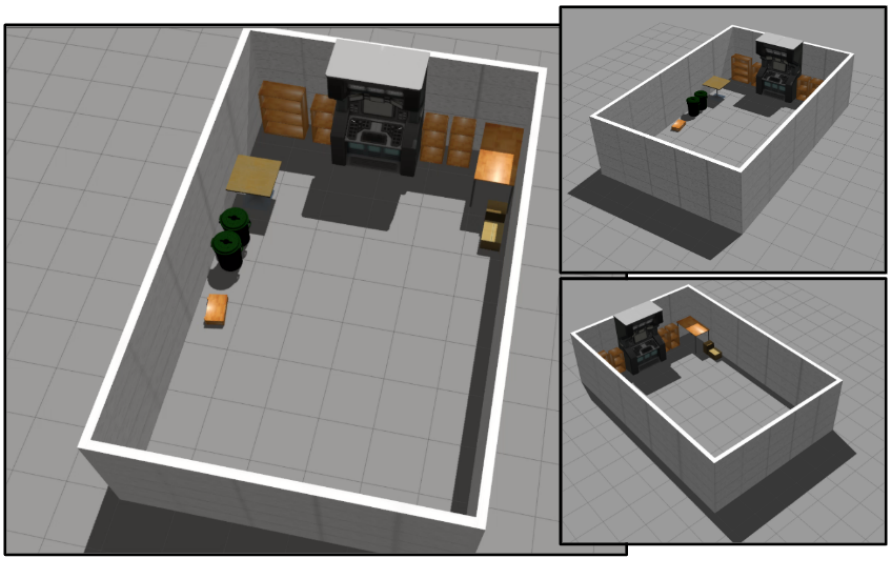


**Figure 4.6:** Simulated closed-room environment in Gazebo with sparse obstacles near the spawn point and a denser object arrangement in the upper area.

Before presenting the reconstruction results obtained with the different sensor configurations, it is useful to observe the mapping process in its temporal development. As described in the previous Subsection, it is possible to follow in real time the progressive addition of voxels as objects enter the robot's field of view. Figure 4.7 shows some significant instants along the path, in which it is possible to appreciate how the space around the robot is gradually reconstructed and consolidated in the volumetric map.

By performing a complete data collection from the robot's sensors, accurately teleoperating it through the entire map, it is possible to obtain a reconstruction that covers almost the entire environment. Figure 4.8 shows a complete reconstruction visualized in **MeshLab**, accompanied by the Gazebo simulation of the environment and different angles of the mesh, which highlight the fidelity of the volumetric representation compared to the original scene.

To perform a systematic comparison of performance varying the sensor used

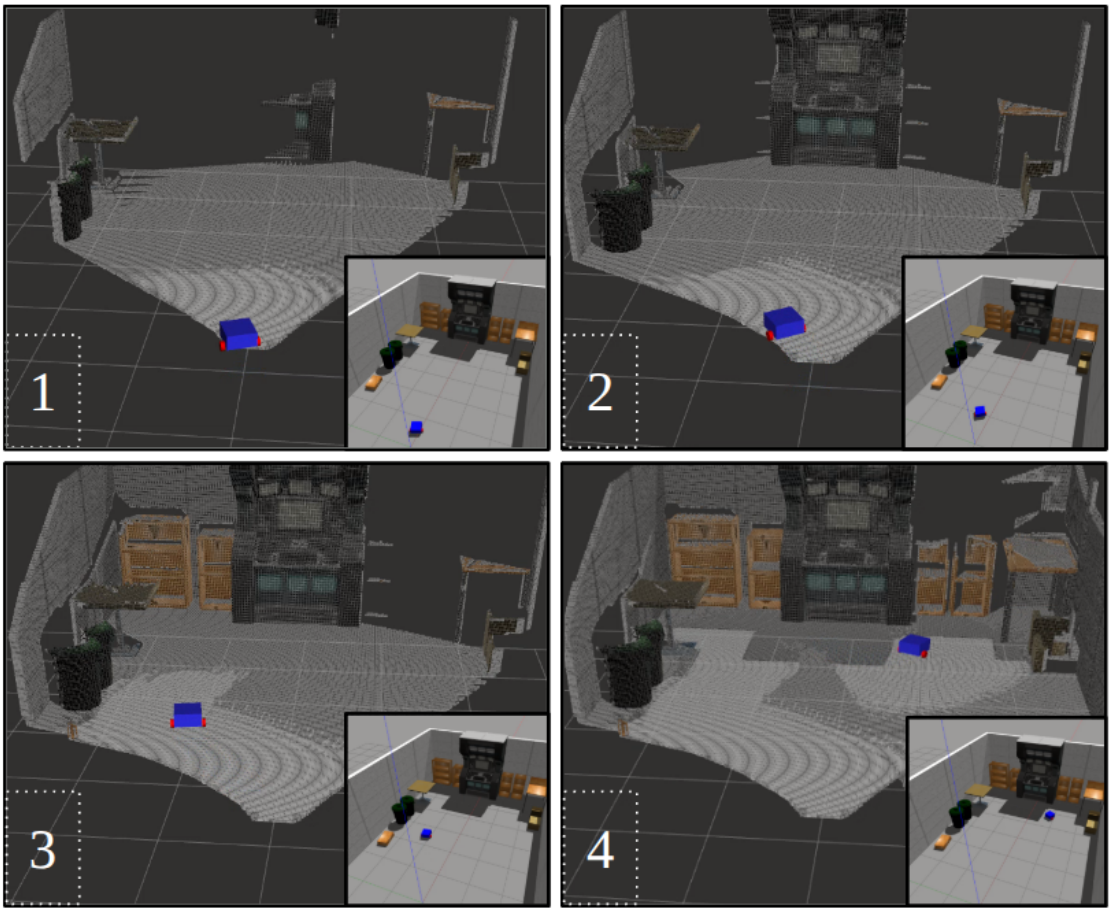



**Figure 4.7:** Step-wise snapshots of the real-time 3D reconstruction along the robot path, showing the progressive accumulation of voxels around the obstacles and room structure.

and the voxel size, it is fundamental to ensure consistency between measurements. Manual use of the joystick to teleoperate the robot is not feasible, as it would prevent reproducing exactly the same trajectory in each experiment. As already discussed in Subsection 3.1.1, ROS **rosbags** provide the ideal solution: by recording the *topics* related to the robot's movement, it is possible to define a reproducible reference trajectory and send it identically to each simulation.

This methodology guarantees not only to save time compared to manual teleoperation but, above all, rigorous comparability of results: varying sensors, **NvBlox** parameters or other configurations, the comparison remains correct since the robot always follows the same path. Figure 4.9 illustrates the trajectory recorded within the room, with the starting point clearly identifiable at the robot's spawn and the origin of the Gazebo coordinate system.

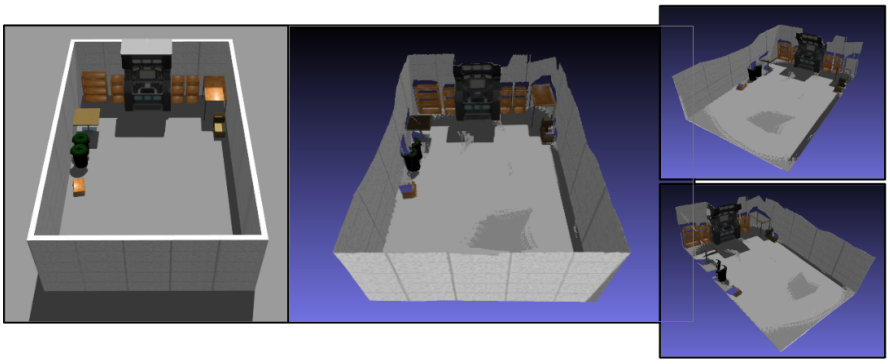


**Figure 4.8:** Complete 3D reconstruction result using a depth camera with voxel size 0.02m: MeshLab rendering alongside Gazebo simulation reference.

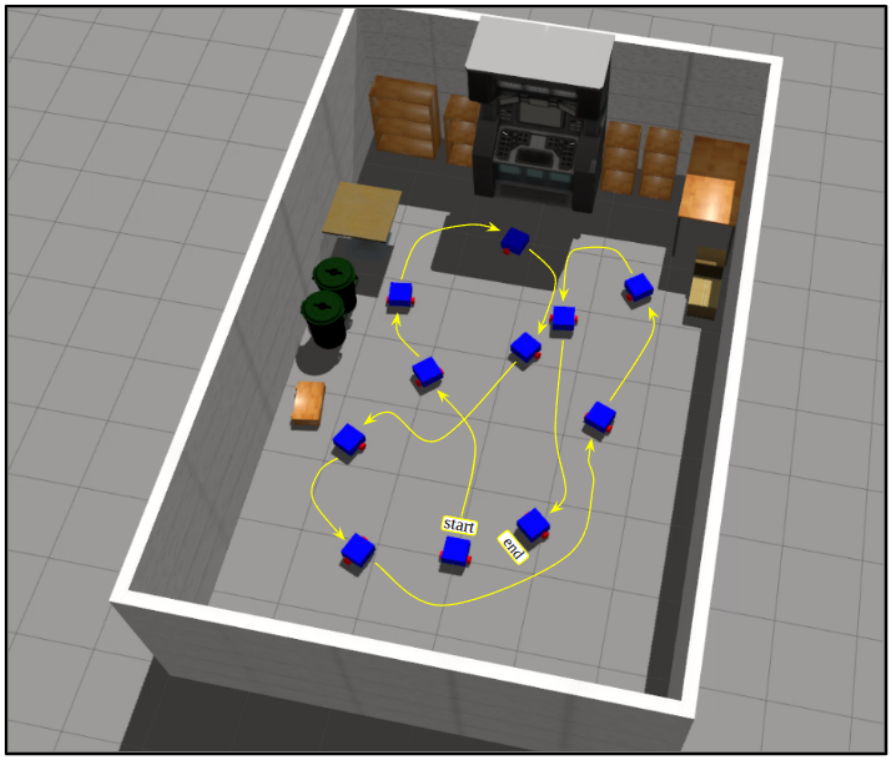



**Figure 4.9:** Robot trajectory within the simulated room, recorded using rosbag and replayed identically across different experimental configurations.

The path was designed to ensure adequate visual coverage, while remaining relatively simple. For a truly exhaustive reconstruction — that included for example the back surfaces of obstacles or the full height of the walls — a much more articulated exploration would be necessary, bringing the robot to visit all

accessible corners and observe every surface from multiple perspectives. Such completeness also depends on the intrinsic limitations of the sensors: the vertical field of view of a depth camera sets a maximum limit on the altitude reachable by the reconstruction.

Table 4.1 reports the numerical results obtained from the different configurations, highlighting the impact of voxel size on the reconstruction output metrics (vertices, faces and occupied memory). It is observed how a higher resolution (smaller voxels) produces significantly more detailed meshes, but at the cost of an exponential increase in memory consumption.

**Table 4.1:** 3D Reconstruction Results with Different Voxel Sizes

| Method | Voxel Size | Vertices | Faces | Memory Size |
|---|---|---|---|---|
| Depth camera | 0.02 | 436,227 | 681,434 | 42.0M |
| | 0.05 | 63,928 | 97,271 | 5.8M |
| | 0.10 | 17,270 | 26,434 | 1.5M |
| 3D LiDAR | 0.02 | 357,952 | 539,017 | 33.0M |
| | 0.05 | 63,133 | 97,572 | 5.7M |
| | 0.10 | 16,393 | 24,801 | 1.4M |

For a direct visual comparison, Figure 4.10 shows the reconstructions obtained from the different configurations, all processed along the same trajectory recorded in the previously described rosbag.

Being **NvBlox** a strongly GPU-centric algorithm, the most significant analyses concern the computational load imposed on the graphics card. During the execution of the movement rosbag (duration about 90 seconds), GPU usage data was recorded at one-second intervals for each configuration.

Figure 4.11 shows the temporal evolution of the GPU **Streaming Multiprocessors (SM)** usage, which represents the main computational load metric. Each line corresponds to a different configuration, with the x-axis indicating time (in seconds) and the y-axis the SM usage percentage. For better readability, **depth camera** configurations are represented with solid lines (in shades of orange), while **3D LiDAR** uses dashed lines (in shades of blue).

It is observed that GPU usage mainly depends on voxel size: finer resolutions require greater computational effort, as expected given the greater quantity of voxels to process. However, the **depth camera** with 0.02m voxels shows more marked oscillations compared to the equivalent **3D LiDAR**. This difference can be attributed to the fact that **NvBlox** shows usage peaks when receiving information related to completely new regions of the scene: with the depth camera, the robot's rotation can bring into view areas never seen before (with all informative data)

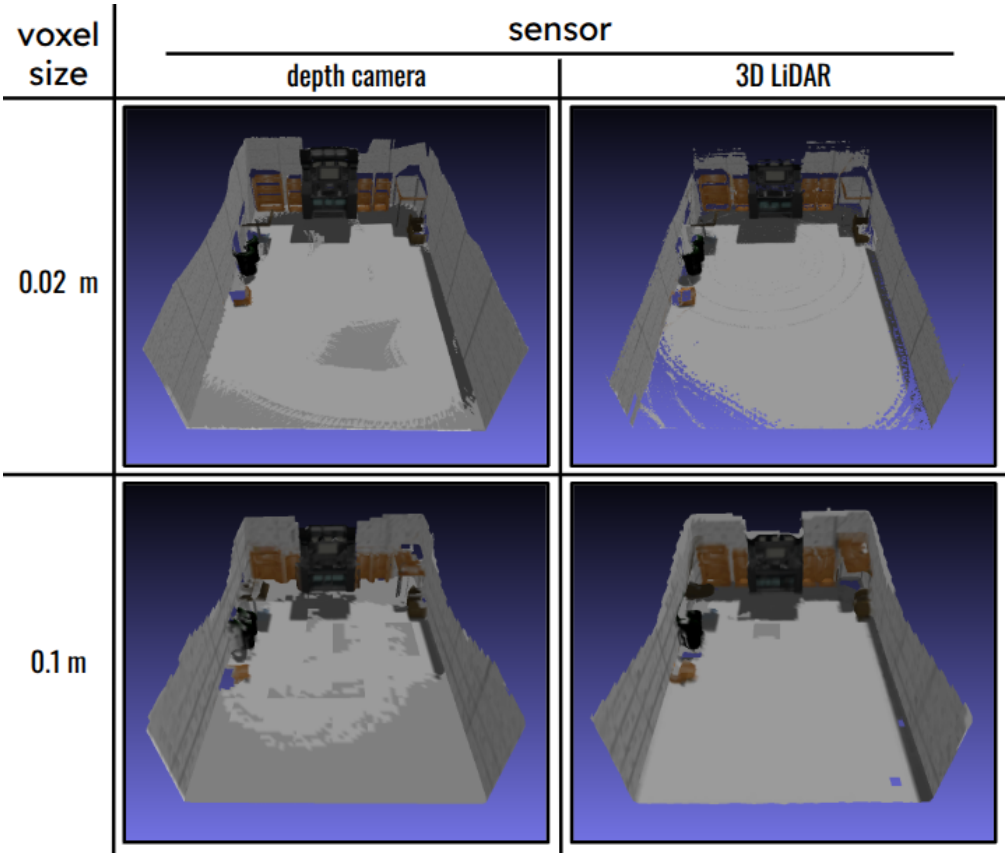



**Figure 4.10:** Visual comparison of 3D reconstructions obtained with different sensor configurations and voxel sizes, following the same robot trajectory.

or already mapped areas (with redundant data), generating fluctuations. The 3D LiDAR, with its wider and more uniform field of view, better distributes the informative flow over time, avoiding such pronounced peaks.

Figure 4.12 confirms the previous observations through the GPU power consumption (in Watts), showing a strong correlation with SM usage. The peaks observed in the **depth camera** 0.02m configuration are also reflected here, confirming that power consumption is directly proportional to the actual computational load.

For a comprehensive quantitative summary, Table 4.2 collects the average and maximum values of the main GPU metrics recorded during the entire reconstruction process. The column of the maximum GPU temperature provides additional indications of operational limits: high values activate hardware safety mechanisms that can automatically interrupt excessively heavy processes.

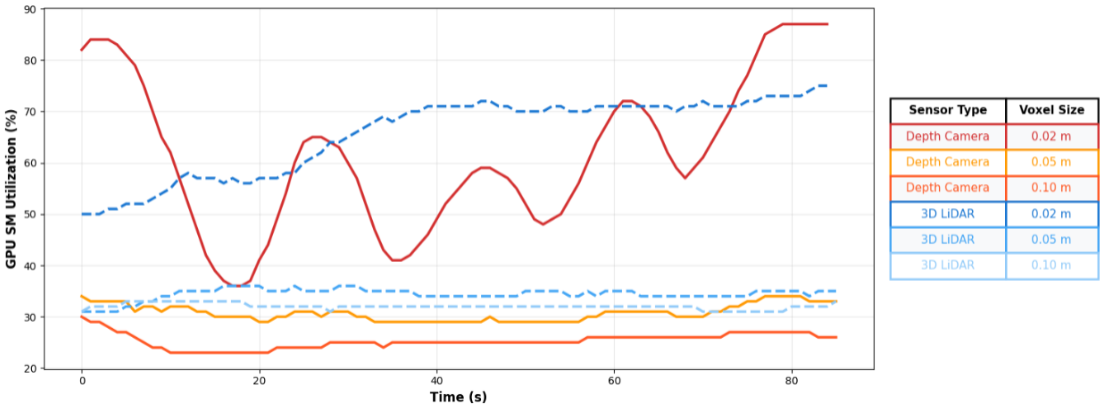



**Figure 4.11:** GPU Streaming Multiprocessor (SM) utilization over time during reconstruction: depth camera (solid lines) vs 3D LiDAR (dashed lines) for different voxel sizes.

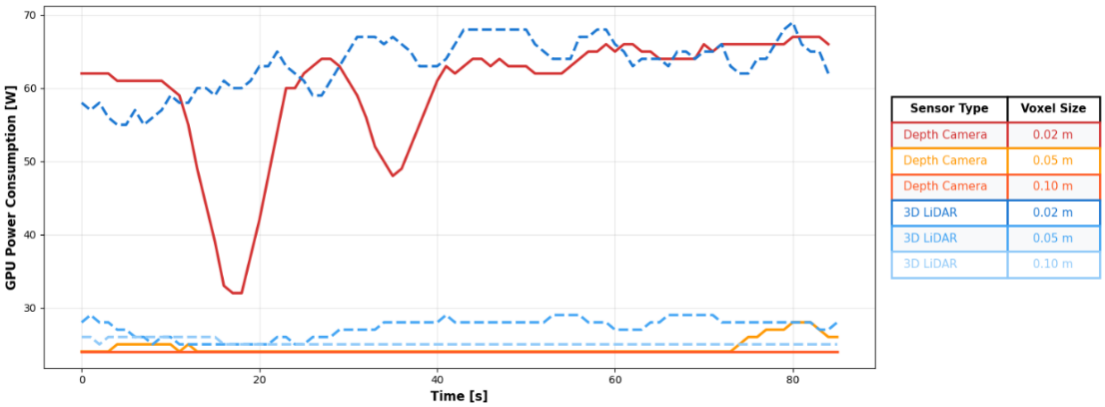



**Figure 4.12:** GPU power consumption over time during reconstruction, showing the same patterns observed in SM utilization.

**Table 4.2:** GPU Performance Metrics During Reconstruction

| Sensor | Voxel | SM avg | SM max | Power avg | Power max | GPU Temp max |
|---|---|---|---|---|---|---|
| Depth Cam | 0.02 | 61.9% | 89% | 60.1 W | 68 W | 69 °C |
| Depth Cam | 0.05 | 31.2% | 40% | 24.9 W | 29 W | 56 °C |
| Depth Cam | 0.10 | 25.7% | 36% | 24.2 W | 26 W | 55 °C |
| 3D LiDAR | 0.02 | 66.1% | 80% | 63.8 W | 70 W | 69 °C |
| 3D LiDAR | 0.05 | 34.8% | 39% | 27.8 W | 30 W | 57 °C |
| 3D LiDAR | 0.10 | 32.5% | 38% | 25.3 W | 28 W | 58 °C |

## 4.5 Enhancing Nav2 Costmaps with NvBlox 3D Reconstruction

Before moving on to the Section regarding the implementation and real-world experiments, it is fundamental to analyze the integration of **NvBlox** with **Nav2**. As introduced in Subsection 3.1.4, Nav2 is a modular framework based on *Behavior Trees* for the management of complex navigation tasks. Through a map built in layers (layered costmap), Nav2 allows the robot to interpret the surrounding space, distinguishing between navigable paths, forbidden zones, and areas to preferably avoid due to the proximity to obstacles. The objective of this integration is to exploit the dense 3D reconstruction provided by NvBlox to enrich the 2D navigation map, offering a superior environmental perception compared to traditional methods.

The implementation of Nav2 in the project requires the installation of the specific package and the creation, in the `/launch` directory, of a file named `nav2.launch.py`. As shown in Listing 4.6, this script manages the inclusion and the launching of the file `bringup_launch.py`, located in the standard Nav2 package.

```
1
2       nav2_params = os.path.join(
3           get_package_share_directory(package_name),
4           'config',
5           'nav2_params.yaml'
6       )
7
8       nav2_bringup_dir = get_package_share_directory('nav2_bringup')
9       nav2_launch = IncludeLaunchDescription(
10          PythonLaunchDescriptionSource(
11              os.path.join(nav2_bringup_dir, 'launch', '
    bringup_launch.py')
12          ),
13          launch_arguments={
14              'use_sim_time': 'true',
15              'params_file': nav2_params,
16              'map': os.path.join(
17                  get_package_share_directory(package_name),
18                  'maps',
19                  'static_map_save.yaml'
20              )
21          }.items()
22      )
```

**Listing 4.6:** Launcher for Nav2 initialization

It is possible to note two crucial arguments passed during the launch. The first one is the configuration file `nav2_params.yaml`, cloned from the `config` directory of the original Nav2 package to facilitate its modification in subsequent phases. This

file defines the structure of the costmap, specifying the order and the parameters of the layers used. Although Nav2 allows configuring the *local map* (for the reactive management of dynamic obstacles) and the *global map* (for long-range planning) separately, in this implementation phase the layer structure is shared and it is as illustrated in Listing 4.7.

```
    plugins: ['static_layer', 'obstacle_layer', 'inflation_layer']

    static_layer:
        plugin: "nav2_costmap_2d::StaticLayer"
        enabled: True
        map_topic: "/map"
        subscribe_to_updates: True
        transform_tolerance: 0.0

    obstacle_layer:
        plugin: "nav2_costmap_2d::ObstacleLayer"
        enabled: True
        max_obstacle_height: 2.0
        min_obstacle_height: 0.0
        laser_scan_sensor:
          topic: /lidar_points
          data_type: PointCloud2
          marking: True
          clearing: True

    inflation_layer:
        plugin: "nav2_costmap_2d::InflationLayer"
        enabled: True
        cost_scaling_factor: 10.0
        inflation_radius: 0.8
```

**Listing 4.7:** Layer configuration for Nav2 maps

The `plugins` array establishes the stacking order of the layers that make up the map. Subsequently, each layer is detailed with its own parameters:

- `static_layer`: visualizes the pre-generated static map, requiring the topic on which it is published.
- `obstacle_layer`: manages obstacles asking on which topic to listen for the sensor readings.
- `inflation_layer`: mathematically calculates the expansion of danger areas around obstacles via the `inflation_radius` parameter, which defines the physical safety radius, and the `cost_scaling_factor` parameter, which determines how quickly the cost decreases moving away from the obstacle; higher values restrict the high-cost zone.

Subsequently, after inserting the Nav2 launch into the project (as shown in Listing 4.8) and after providing an initial map generated via SLAM, it is possible to visualize the complete costmap.

```
    nav2 = IncludeLaunchDescription(
        PythonLaunchDescriptionSource([os.path.join(
            get_package_share_directory(package_name), 'launch', '
    nav2.launch.py'
        )]),
        launch_arguments={'use_sim_time': 'true'}.items()
    )
    ...
    ...
    ...
    return LaunchDescription([
        ...
        nav2,
        ...
    ])
```

**Listing 4.8:** Inclusion of Nav2 in the main launcher

Figure 4.13 shows a comparison between the simulated world in Gazebo and the costmap generated by Nav2 using the three standard layers previously described.

However, the standard approach presents significant limitations. The detection of obstacles is bound to the scan plane of the LiDAR sensor: exclusively the data present at the sensor mounting height are considered. This prevents a volumetric assessment of the obstacle, leaving zones erroneously free that might need to be avoided. Critical situations include obstacles with bases wider than the Section seen by the LiDAR or suspended objects that could impact the upper part of the robot.

Figure 4.14 highlights this issue: by removing the walls for clarity, it is noted how the ray divergence leaves entire portions of objects "invisible". For example, only the legs of the tables are detected, ignoring the tabletop, and some shelves are perceived with partial geometries. Although the `inflation_layer` helps to mitigate the risk by virtually expanding the obstacles, this approach is not sufficient for objects that completely escape the scan plane, as shown in the red box of the figure, where a low obstacle (on the left, under the cylindrical figures) is totally ignored.

This structural problem requires the introduction of an additional level in the costmap, capable of considering the three-dimensionality of objects. The solution lies in the use of the plugin provided by the **NvBlox** library, which generates a layer based on the dense 3D reconstruction. In Listing 4.9 the modifications to the file `nav2_params.yaml` are reported: the definition of the `nvblox_layer` and its insertion into the plugin array, positioning it before the `inflation_layer`.

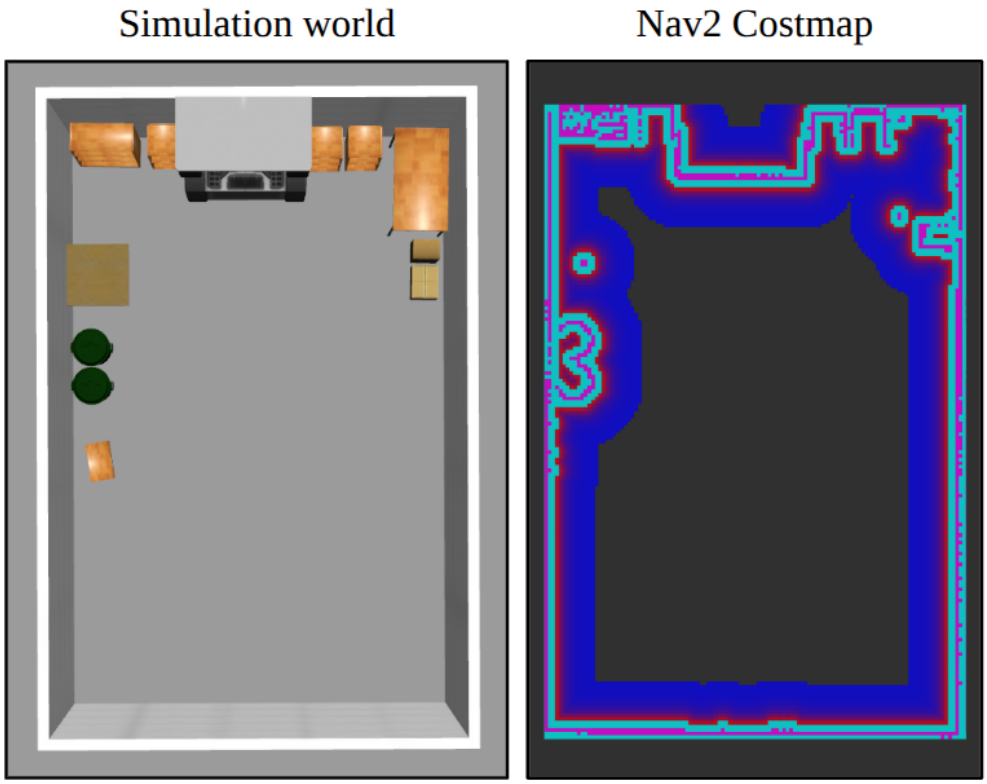



**Figure 4.13:** Visual comparison: simulated world (left) and Nav2 Costmap generated by static, obstacle and inflation layers (right).

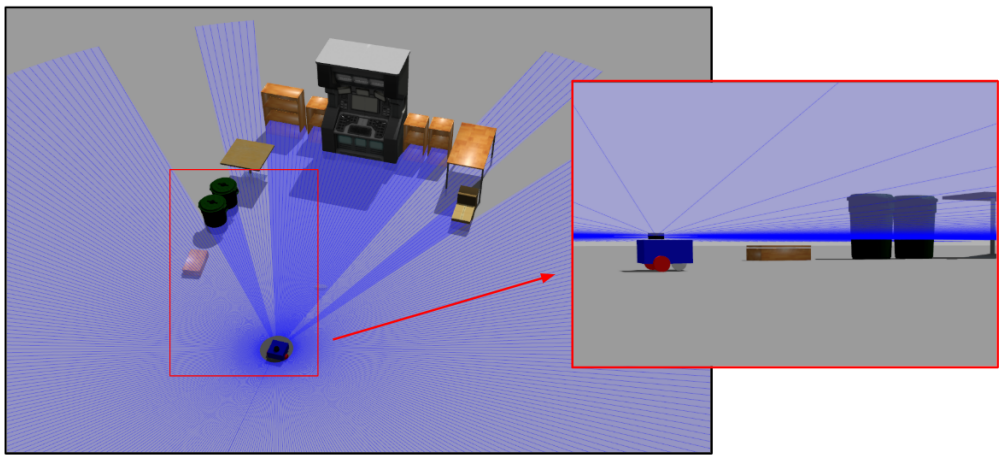


**Figure 4.14:** Analysis of LiDAR ray divergence. The red box highlights a low obstacle not detected since it is located below the scan plane.

```
1
2  plugins: ['static_layer', 'obstacle_layer', 'nvblox_layer', '
       inflation_layer']
3        ...
4        ...
5        ...
6        nvblox_layer:
7          plugin: "nvblox::nav2::NvbloxCostmapLayer"
8          enabled: True
9          nav2_costmap_global_frame: odom
10         nvblox_map_slice_topic: "/nvblox_node/static_map_slice"
11         convert_to_binary_costmap: True
```

**Listing 4.9:** Definition and configuration of the NvBlox layer

This integration allows building a high-fidelity costmap, capable of representing obstacles outside the sensor plane and understanding their complete geometry. Figure 4.15 presents an exhaustive comparison: it is clearly noted how, with the addition of the NvBlox layer, the tables are mapped in their entirety and the obstacles result more defined, eliminating the collision risks due to geometries not completely detected by the 2D LiDAR alone.

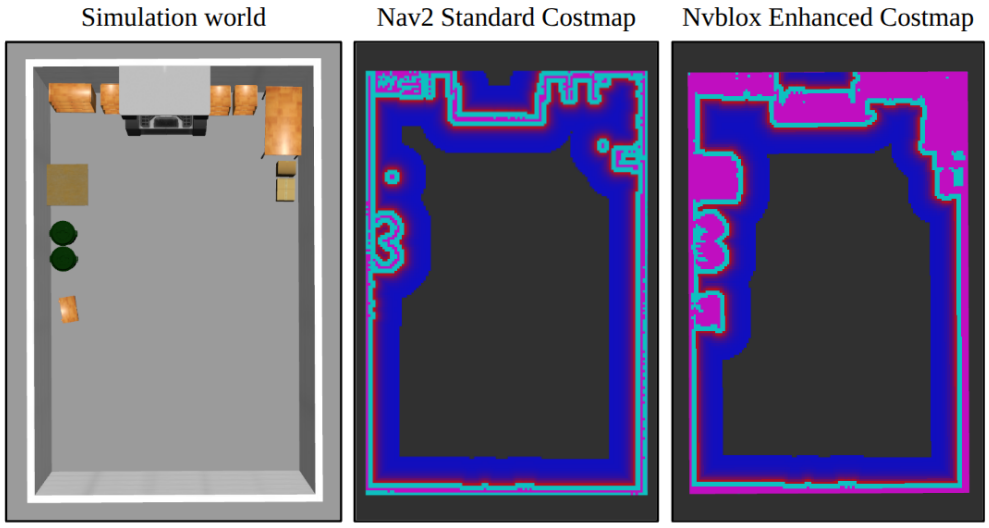



**Figure 4.15:** Three-way comparison: simulated world (left), standard Nav2 Costmap (center) and Costmap enriched with the NvBlox layer (right).

To validate the effectiveness of this solution, a path planning test (*path to goal*) was performed. The low obstacle, invisible to the planar LiDAR, was moved directly onto the robot's trajectory, as shown in Figure 4.16.

Requesting the robot to reach a destination placed behind the obstacle, the

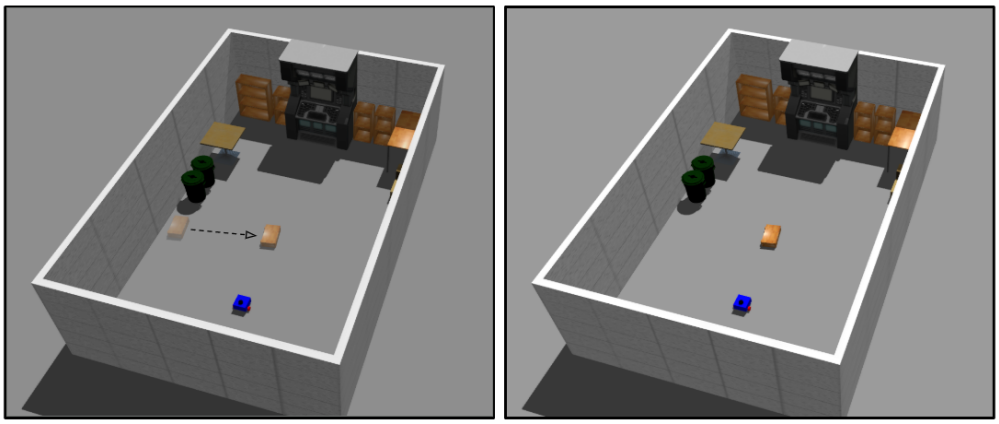


**Figure 4.16:** Experimental setup: repositioning of the low obstacle along the robot's frontal trajectory.

results are evident (Figure 4.17). With the standard approach, the planner traces a straight line, ignoring the obstacle and causing a collision that compromises the pose and stability of the robot. Conversely, in the *enhanced* configuration with NvBlox, the obstacle is correctly detected and mapped, allowing Nav2 to calculate a safe trajectory and bypass it.

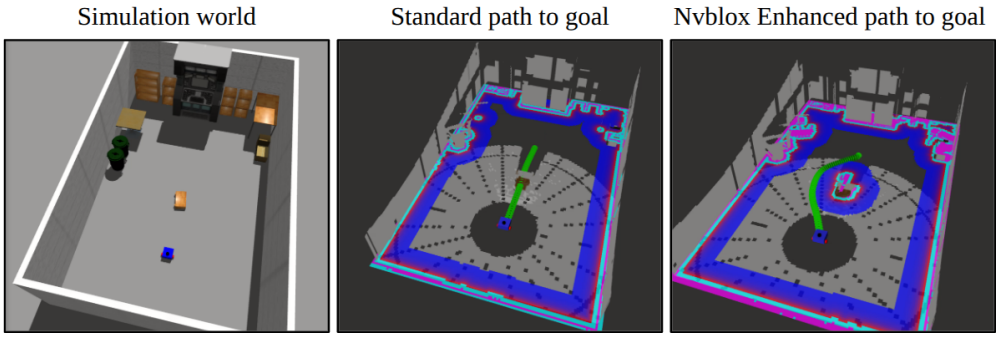



**Figure 4.17:** Planning comparison: path in collision with standard approach (center) vs obstacle avoidance path with NvBlox integration (right).

# Chapter 5

# Real-world Deployment at LINKS Foundation

The final objective of this thesis work consists of transferring the developed system from the simulated environment to the real world, adapting the algorithms for use on a physical robot equipped with a specific sensor suite. Experimental validation in a real-world scenario plays a fundamental role in robotics research: it allows for verifying the robustness of the algorithms against disturbances that cannot be modeled in simulation, such as unpredictable sensor noise, varying lighting conditions, and complex physical dynamics.

This activity aligns with the strategic plans of the LINKS Foundation research center, which aims to make its robotic platforms fully autonomous in navigating company spaces. The intent is to assign these robots future operational roles, which are still being defined but are closely linked to interacting with staff and guests, as well as managing the logistics of the facilities.

Although the simulation phase provided a solid and verified baseline, ensuring smooth preliminary integration, the transition to the physical world inevitably introduces new challenges. Indeed, the ideal approximations typical of virtual environments no longer hold, requiring the management of stringent hardware and environmental constraints.

## 5.1 Hardware Selection and Challenges

In a simulated environment, configuration freedom is almost absolute: it is possible to define sensors with ideal characteristics and position them arbitrarily without considering mechanical constraints related to mounting, wiring, or weight balancing. In the real world, this flexibility is lost, necessitating a careful analysis of the available and compatible hardware architectures.

To replicate the operation of the 3D reconstruction system, it was necessary to identify a minimum configuration consisting of a mobile robotic base (*teleoperable*) and a sensor system capable of collecting geometric data from the environment. The design choices were guided by the equipment available at the research center and the desire to invest in platforms with high potential for future development.

### 5.1.1 Selected Sensor: 3D LiDAR

Regarding the acquisition of environmental data, the choice fell on the only available 3D LiDAR sensor, which offers performance in line with the requirements validated in simulation. The chosen LiDAR is the RoboSense RS-Helios-16P, a rotating **16-channel** sensor with a 360° horizontal field of view and a 30° overall vertical field of view (+15° to -15°). It is based on a 905 nm laser and is classified as **Class 1 eye-safe**. The sensor offers a typical range from 0.2 m up to 150 m (90 m on targets with 10% reflectivity), with a measurement accuracy of approximately ±2 cm and an angular resolution of 0.2°/0.4° horizontally and 2° vertically. It supports scanning frequencies of 10 Hz and 20 Hz (600/1200 rpm), produces up to 288,000 points/s in *single return* mode and 576,000 points/s in *dual return* mode, and communicates via a 100M Ethernet interface using UDP packets containing 3D coordinates, calibrated reflectivity, and timestamps [33].

The bare version of the sensor is shown in Figure 5.1. However, at the LINKS Foundation, the LiDAR was enclosed in a protective cage and equipped with predefined rails for mounting on the selected robots.

The use of a 3D LiDAR, compared to passive vision-based solutions, was preferred for its versatility and intrinsic precision. Given the same computational resources, a video signal cannot resolve geometric details with the same accuracy and stability as a time-of-flight sensor. However, the weight and size of this device narrowed down the suitable robotic platforms for its transport.

To establish communication with the computer and transmit the geometric scan data, the sensor requires a constant physical connection via an Ethernet cable for the entire duration of the acquisition session. This connection enables the high-speed data transfer required for real-time reconstruction.

From a software perspective, integration into the ROS 2 system is ensured by installing two specific packages within the workspace: `rslidar_msg` and `rslidar_sdk`. The first package defines the custom message formats for the sensor, while the second acts as a driver and software development kit for decoding the raw signals.

The actual data acquisition and publication as a ROS topic rely on the activation of the main node, named `rslidar_sdk_node`, from the `rslidar_sdk` package. Drawing on parameters defined in a dedicated YAML configuration file, this software component transforms the incoming data from the Ethernet cable into


**Figure 5.1:** The RoboSense RS-Helios-16P 3D LiDAR sensor utilized for environmental point cloud acquisition [34].

standardized point clouds, which are then published as `sensor_msgs/PointCloud2` messages on the default `/rslidar_points` topic.

A straightforward integration of this node into the launch file is shown in Listing 5.1, which allows calling the node within the Launch Description using its assigned name, in this case `rslidar_node`.

```
    lidar_config_path = os.path.join(
        get_package_share_directory('rslidar_sdk'), 'config', '
    config.yaml'
    )

    rslidar_node = Node(
        package='rslidar_sdk',
        executable='rslidar_sdk_node',
        name='rslidar_sdk_node',
        output='screen',
    )
```

**Listing 5.1:** rslidar node definition

### 5.1.2 Robotic Platform: Unitree Go1

For the choice of the robot, two main alternatives were evaluated: a wheeled rover (e.g., Turtlebot) and a quadruped (Unitree Go1). Despite the easier integration offered by a wheeled system, the final decision fell on the Unitree Go1 quadruped robot, shown in Figure 5.2. This decision was motivated by several strategic and technical factors:


**Figure 5.2:** The Unitree Go1 quadruped robot selected as the mobile platform for the experimental validation [35].

- **Research Continuity:** The research center has been working on quadruped applications for about five years, whereas the use of LiDAR on wheeled vehicles is an already established industrial reality. Applying it to legged robots therefore offers more opportunities for innovation.
- **Dynamics and Robustness:** Unlike a wheeled rover, a quadruped introduces significant disturbances into the measurements during locomotion, such as high-frequency vibrations, roll, pitch, and height variations. Testing the algorithm under these conditions allows evaluating its robustness in much more complex scenarios compared to the planar motion of a Turtlebot.
- **Mobility:** The legged structure guarantees the potential ability to overcome

vertical obstacles and navigate stairs, an essential feature for future full navigation of the multi-level building.

- **Human-Robot Interaction (HRI):** Considering future application scenarios that involve interacting with people within the company, the quadruped's morphology provides significantly higher communicative engagement and impact compared to standard low-profile robotic platforms.

However, it is necessary to highlight a significant technical challenge related to this platform. Although the Unitree Go1 is natively equipped with three depth cameras (front and lateral) —which ideally would have allowed 3D reconstruction without additional external sensors— previous implementations have compromised accessibility to the robot's internal board. This made it impossible both to acquire the data stream from the integrated cameras and to programmatically send velocity commands (or read the internal odometry). Consequently, the robot is essentially used as a transport vehicle for the external LiDAR, manually controlled via joystick.

Since the robot dog is used solely for locomotion—which is already configured by the developers via the provided joystick—it is not necessary to install drivers for its basic operation. Nevertheless, some code modifications are required regarding its visual representation. As explained in Subsection 3.1.2, visualizing the robot requires a URDF description that defines all its components and positions them with the correct joints. Obviously, this description does not include the physics simulation parameters for Gazebo; it serves merely as a graphical representation to be integrated into the visualizer, mapping the robot's spatial movement. While not strictly mandatory—a much rougher approximation could be used just to indicate the robot's current position and movement—LINKS Foundation shared a package called `unitree_ros2` containing a Robot State Publisher (RSP) launch file already configured with the relevant URDF. Thus, it only needs to be integrated into the main experimental launch file. Once the package is imported and built within the workspace, a dedicated RSP can be added to the launch file, as shown in Listing 5.2, setting the necessary arguments for a clean execution.

```
1     rsp = IncludeLaunchDescription(
2         PythonLaunchDescriptionSource([os.path.join(
3             get_package_share_directory('go1_description'), '
    launch', 'load_go1.launch.py'
4         )]),
5         launch_arguments={
6             'use_jsp': 'jsp',
7             'use_rviz': 'false',
8         }.items()
9     )
```

**Listing 5.2:** Unitree robot state publisher definition

The `load_go1.launch.py` file also automatically launches RViz; setting the `'use_rviz'` argument to `'false'` prevents this, as RViz is managed separately in the custom launcher. The `use_jsp` argument supports various modes: setting it to `jsp` hides the Joint State Publisher GUI. Since manual control of the robot's legs is not required and their movement cannot be accessed programmatically, it is preferable to keep the represented legs fixed in a stretched position, letting them intersect with the floor. Figure 5.3 shows the robot model visualized in RViz upon launching the node, alongside a simplified representation of the robot's TF tree. The static transformation between `base` and `trunk` serves to elevate the torso from the ground, with all other robot parts attached directly to the trunk. The `trunk` acts as the central reference frame for components mounted on the robot (such as the LiDAR), while `base` serves as the reference frame for odometry.

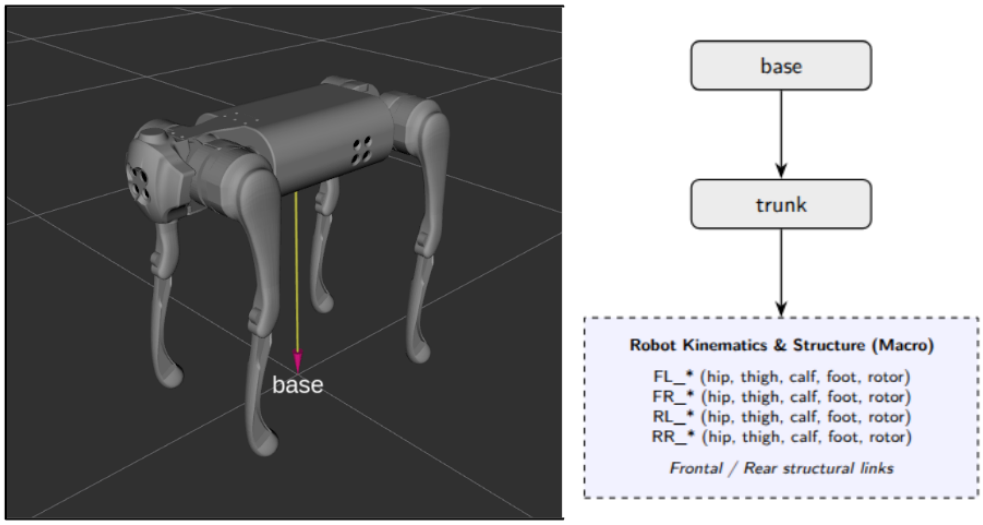



**Figure 5.3:** Visualization of the Unitree Go1 URDF model in RViz (left) alongside a simplified representation of its corresponding TF tree (right), highlighting the structural relationship between the base, trunk, and attached components.

## 5.2 Pose Estimation Problem

The hardware selection and the limitations described above dictate a specific integration strategy with the *NvBlox* framework. For proper operation, the algorithm strictly requires two synchronized data streams:

- **Point Cloud:** The raw environmental data provided by the externally mounted LiDAR sensor.

- **Pose:** The position and orientation of the sensor in space relative to a fixed global reference frame.

While point cloud generation is guaranteed by the proper functioning of the LiDAR, pose estimation represents the primary critical issue in this real-world setup.

In a simulated environment, the pose can be obtained directly from the simulator (*Ground Truth*) or accurately estimated via wheel odometry derived from the issued velocity commands. In the current real-world scenario, no *Ground Truth* is available and, due to the inaccessibility of the robot's control board, it is impossible to read the commands sent via joystick to calculate odometry based on a kinematic model. This scenario therefore requires the implementation of alternative localization techniques capable of tracking the exact movement of the robot to feed into *NvBlox* for reconstruction. Two distinct solutions are proposed: one is a software-based approach, while the other is hardware-based and relies on a highly non-portable system, thereby constraining the reconstruction capabilities to the reach of the localization infrastructure.

### 5.2.1 Pose Estimation via LiDAR Odometry: The KISS-ICP Approach

The first analyzed solution addresses the problem from a purely computational perspective, exploiting the geometric information provided by the LiDAR sensor to estimate the robot's movement.

For this purpose, **KISS-ICP** (*Keep It Small and Simple – Iterative Closest Point*) was employed. This is an efficient and robust LiDAR odometry algorithm capable of operating in real-time without the need for an Inertial Measurement Unit (IMU) or complex kinematic models. Its operating principle is based on sequential point cloud registration (*scan-to-scan matching*): the algorithm calculates the spatial transformation that minimizes the distance between the points of a new scan and the progressively built local map, thereby deriving the robot's pose with high precision.

System integration is achieved through a dedicated node, whose configuration is detailed in Listing 5.3.

```
1     pointcloud_topic = "/rslidar_points"
2     base_frame = "base"
3     lidar_odom_frame = "odom"
4
5     kiss_icp_node = Node(
6         package=package_name_kiss,
7         executable="kiss_icp_node",
8         name="kiss_icp_node",
```

```
9           output="screen",
10          remappings=[
11              ("pointcloud_topic", pointcloud_topic),
12          ],
13          parameters=[
14              {
15                  "base_frame": base_frame,
16                  "lidar_odom_frame": lidar_odom_frame,
17                  "publish_odom_tf": True,
18                  "invert_odom_tf": True,
19                  "use_sim_time": True,
20                  "position_covariance": 0.1,
21                  "orientation_covariance": 0.1,
22              },
23              os.path.join(get_package_share_directory(
    package_name_kiss), "config", "config.yaml")
24          ]
25      )
```

**Listing 5.3:** Configuration of the KISS-ICP node for odometry estimation.

An analysis of the node parameters highlights the strategic choices made to ensure compatibility with the *NvBlox* framework. Primarily, defining `base_frame` and `lidar_odom_frame` allows KISS-ICP to seamlessly integrate into the Transform (TF) tree, acting as a broadcaster of the relationship between the fixed reference frame (`odom`) and the robot's mobile frame (`base`). Automatically, without inserting additional transformations, the TF tree thus becomes as shown in Figure 5.4.

Particularly relevant is the `invert_odom_tf` parameter, set to `True` to correctly align the direction of the published transform with the conventions required by the navigation system. Furthermore, setting `position_covariance` and `orientation_covariance` to 0.1 provides an estimation of the uncertainty associated with the measurement. Although KISS-ICP is highly accurate, this value reflects a methodological prudence necessary to handle potential slipping or vibrations typical of the quadruped's locomotion, which could introduce noise into point registration.

### 5.2.2 Localization via Motion Capture System: Vicon

The second proposed solution for pose estimation relies on a professional **Motion Capture (MoCap)** system provided by **Vicon**. This infrastructure consists of a rectangular "cage" equipped with six overhead infrared cameras designed to map the entire operational volume. The system operates on the principle of **optical triangulation**: the cameras detect the light reflected by retro-reflective markers positioned on the robot's chassis and, by intersecting the visual data, calculate the position and orientation of the rigid body with sub-millimeter precision.

For accurate tracking, it is crucial that the markers are applied to the robot in an

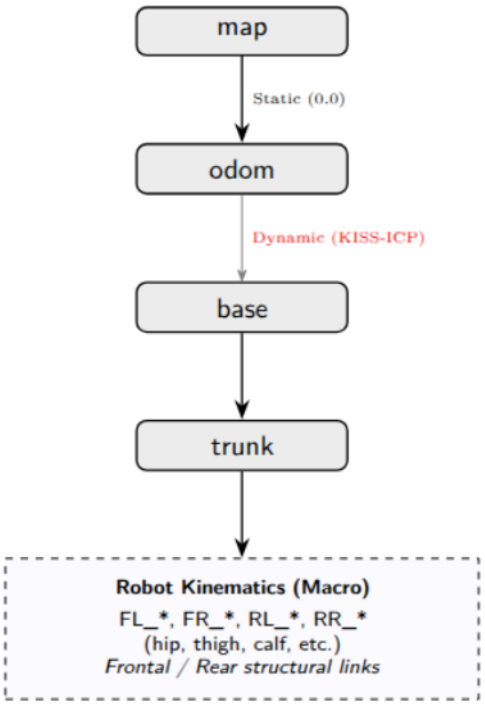



**Figure 5.4:** TF tree for localization via KISS-ICP. The LiDAR odometry directly publishes the dynamic transform between the inertial reference frame (`odom`) and the robot's base.

**asymmetrical** configuration; this arrangement allows the system to unambiguously determine the orientation of the rigid body and avoid geometric ambiguities. Once the object is registered in the Vicon software environment, the data must be transmitted to the ROS 2 system. Since the cameras are connected to a dedicated Windows workstation, the robot's onboard computer must be connected to the same Wi-Fi network to receive the data stream.

Integration is handled by activating a specific client, configured as shown in Listing 5.4.

```
1     vicon_receiver_node = IncludeLaunchDescription(
2         PythonLaunchDescriptionSource([os.path.join(
3             get_package_share_directory(package_vicon), 'launch',
    'client.launch.py'
4         )]),
5         launch_arguments={
6             'hostname': ip,
7             'topic_namespace': 'vicon',
8             'buffer_size': '200',
9             'world_frame': 'map',
```

```
            'vicon_frame': 'vicon',
            'map_xyz': '[0.0, 0.0, 0.0]',
            'map_rpy': '[0.0, 0.0, 0.0]',
            'map_rpy_in_degrees': 'false'
        }.items()
    )
```

**Listing 5.4:** Integration of the vicon_receiver node for external odometry data reception.

The `hostname` parameter identifies the IP address of the Windows station, while the `world_frame` and `map_xyz/rpy` parameters set the name and position of the global reference frame relative to the center of the capture cage. They ensure that the received pose is correctly aligned with the global origin of the virtual world (`map`). Different parameters can obviously allow the reconstruction (which is based on the map frame) to be positioned accordingly at the center of the Vicon setup; however, this is not the case here, as the tests abstract away from such alignments, meaning the `map` and `vicon` values coincide.

The system's operation relies on publishing a transform chain composed of a static and a dynamic component. The `vicon_receiver` node initially generates a *Static TF* between the `map` and `vicon` frames, defining the position of the acquisition cage relative to the origin of the virtual world. Subsequently, a dynamic transform is published between the `vicon` frame and the tracked object's frame, named `dog_dog`. This specific nomenclature stems from the internal logic of the Vicon node, which generates the frame name by concatenating the name of the object registered in the Tracker software (*dog*) with the name of the identified segment, automatically repeating it.

From a physical standpoint, the origin of the `dog_dog` frame was set via the Vicon management software to coincide with the geometric centroid of the infrared marker arrangement. The orientation was calibrated to align with the robot's axes (X-axis forward, Z-axis upward). However, since this point represents an external reference based on the markers and does not necessarily coincide with the origin of the robot's kinematic structure (`base`), a manual connection between the capture system's TF tree and the quadruped's descriptive TF tree is required.

Unlike with KISS-ICP, where the linkage is intrinsic to the odometry calculation, in this case it was necessary to implement a dedicated static transform publisher, as shown in Listing 5.5.

```
    static_tf_cane_to_base = Node(
        package='tf2_ros',
        executable='static_transform_publisher',
        name='static_tf_cane_to_base',
        arguments=['0', '0', '-0.60', '0', '0', '0', 'dog_dog', '
    base'],
        parameters=[{'use_sim_time': False}]
```

```
7         )
```

**Listing 5.5:** Definition of the static transform between the Vicon reference frame and the robot base.

As evident from the node arguments, a vertical offset of $-0.60$ meters along the Z-axis was applied. This measurement compensates for the physical distance between the marker plane (positioned on the top of the robot) and the `base` frame located at the bottom of the quadruped's body. The resulting TF tree for this configuration is shown in Figure 5.5.

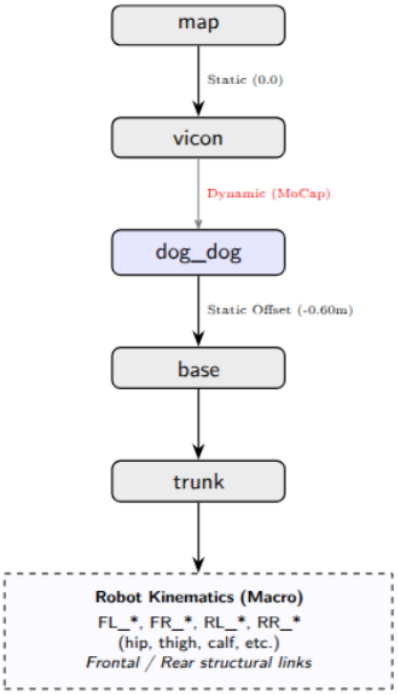



**Figure 5.5:** Complete TF tree for localization via the Vicon system. The figure highlights the distinction between the static calibration transforms and the dynamic transform produced by the Motion Capture system between the cage origin frame (`vicon`) and the markers' centroid (`dog_dog`).

### 5.2.3 Comparative Analysis: LiDAR Odometry vs. Motion Capture

The comparison between the use of KISS-ICP and the Vicon system highlights a clear **trade-off** between operational flexibility and acquisition stability. The two

strategies present substantial divergences that can be summarized in the following key points:

- **Autonomy and Portability:** The **KISS-ICP** approach is intrinsically *on-board*, allowing the robot to move freely in vast or unstructured environments since the pose is estimated by analyzing the environmental geometry. Conversely, the **Vicon** system is bound to the capture "cage" and its physical limits, making it a non-portable solution primarily useful for laboratory validation.
- **Computational Load and Resources:** KISS-ICP is a *compute-intensive* solution that heavily relies on the robot's onboard GPU. Since it must compete with *NvBlox* for the processing power required for point cloud registration, it can cause frame rate drops if not properly balanced. The Vicon system, on the other hand, delegates computation to an external workstation, resulting in an extremely lightweight load for the robot but introducing a critical dependency on Wi-Fi network latency.
- **Mechanical Sensitivity and Vibrations:** During locomotion, the impact of the robot dog's limbs generates vibrations that can be transmitted to the supporting structure of the Vicon cameras, especially when the robot operates near the support poles. The system is extremely sensitive to these stresses: excessive bumps or vibrations can cause momentary optical misalignments that the software interprets as tracking errors, leading to the automatic deactivation of some cameras and the subsequent loss of the tracking signal.
- **Algorithm Robustness:** While Vicon guarantees sub-millimeter precision as long as the markers are visible, KISS-ICP can fail in geometrically degraded environments (such as very long, featureless corridors) or in the presence of excessively abrupt movements that prevent the proper convergence of the point cloud registration algorithm.

Ultimately, while the Vicon system represents the precision baseline for scientific validation in a controlled environment, KISS-ICP constitutes the only solution capable of guaranteeing the portability necessary for real-world exploration missions, provided the higher onboard computational load is acceptable.

## 5.3 Experimental Setup and System Integration

Following the selection of hardware and software architectures, and after defining the methodologies for acquiring the data required by the *NvBlox* algorithm, it was necessary to prepare the system for the actual execution of real-world experiments.

This *deployment* process was divided into two main phases: the physical installation of the sensor suite on the quadruped robot and a profound revision of the ROS 2 *launch* files, which is essential to manage the transition from the simulated environment to the real platform.

### 5.3.1 Robot Hardware Setup

The first intervention involved the mechanical installation of the sensors on the Unitree Go1. The only structurally suitable housing to support the weight and dimensions of the LiDAR proved to be the robot's back (identified by the `trunk` link in the kinematic tree). Although a configuration with the sensor mounted on the head could offer theoretical advantages in terms of field of view, the *payload* limits of that specific joint made the choice of the back mandatory. Figure 5.6 shows the result of the mounting process, with the RoboSense sensor anchored to the `trunk` and protected by the dedicated structural cage, as anticipated in Subsection 5.1.1.

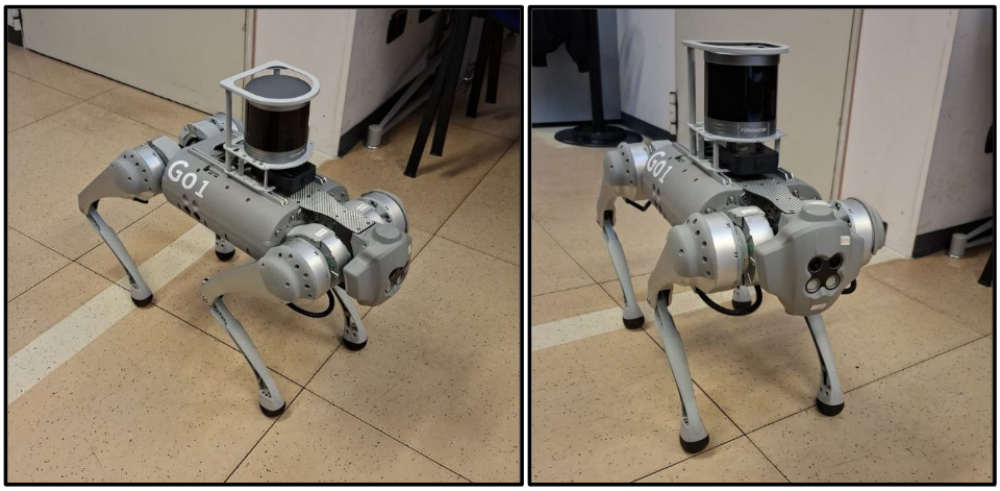



**Figure 5.6:** Hardware configuration of the Unitree Go1 robot: detail of the RS-Helios LiDAR sensor mounted on the back (`trunk`) inside the protective metal structure.

To enable localization via the Motion Capture system, further physical changes were made. As discussed in Subsection 5.2.2, proper tracking requires the application of infrared markers arranged in a rigidly asymmetrical manner. To determine the optimal number and position of the markers, empirical tests were conducted by moving the robot within the capture area and monitoring the Vicon software for any signal loss (occlusions or geometric ambiguities).

The final setup involved the use of 6 spherical markers, positioned exclusively on non-moving parts of the chassis to ensure constant relative distances, a necessary condition for defining the *rigid body*. Furthermore, it was necessary to completely cover the LiDAR's metal cage with dark, opaque adhesive tape: the reflective surfaces of the metal generated **false positives** on the infrared cameras, thereby degrading localization quality.

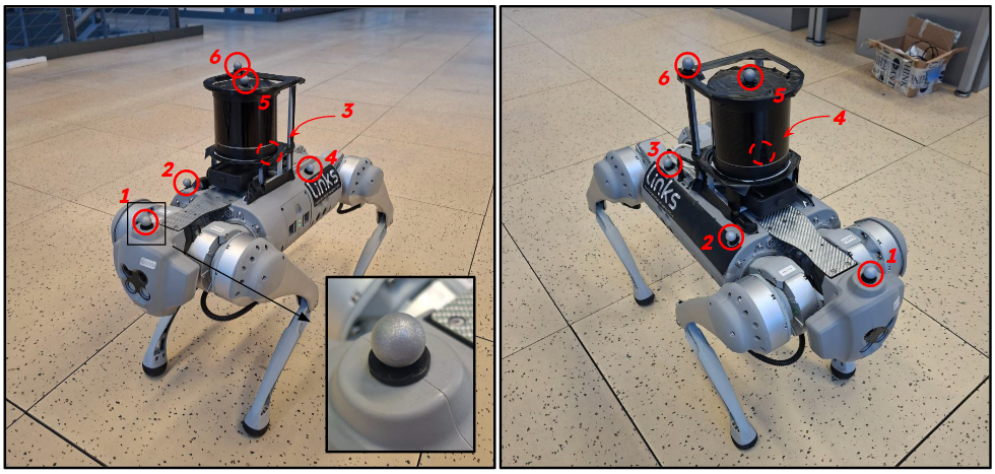



**Figure 5.7:** Setup for Vicon tracking. The 6 retro-reflective markers are highlighted in red; dashed circles indicate markers occluded by perspective. Note the blackout covering applied to the LiDAR cage to suppress optical disturbances. The inset shows an enlargement of the marker positioned on the head.

### 5.3.2 Software Architecture Reconfiguration

The transition from simulation to the real world required a radical cleanup and reorganization of the main launch file. Many of the nodes used during the virtual testing phase proved to be redundant or incompatible with the hardware limits encountered on the real robot. Specifically, the following components were removed:

- **Simulated environment and spawner:** Calls to Gazebo were eliminated, as well as the `diff_drive_spawner` and `joint_broad_spawner` nodes. On the physical system, the kinematics and joint states are managed internally by the robot's proprietary controller.
- **Teleoperation nodes:** Parameters and nodes related to the virtual joystick or ROS controllers were removed, since the quadruped's locomotion is entirely managed via the proprietary remote control provided by Unitree.

- **Navigation stack (Nav2):** Although Nav2 is a standard for autonomy in ROS 2, it was removed from the current pipeline. As previously analyzed, the inability to access the low-level control board prevents sending velocity commands. Nav2 remains an architecturally valid integration, but it is contingent upon resolving the hardware communication limitations in the future.

In parallel with the removals, the launch file was enriched with the nodes necessary for real-world operation. First, the `rslidar_sdk_node` driver was integrated to decode the network packets coming from the sensor (see Subsection 5.1.1).

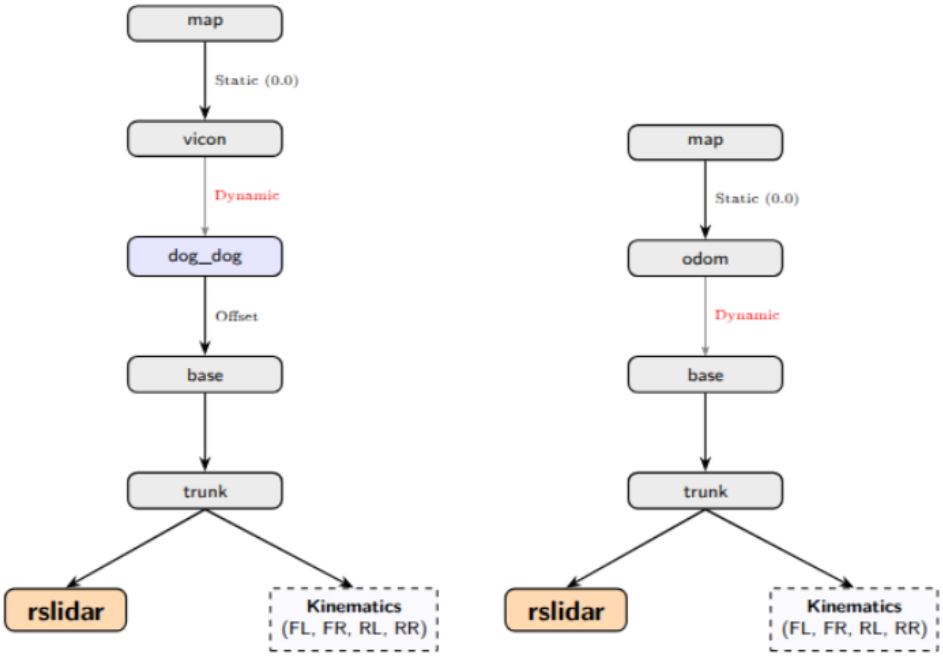
Subsequently, it was crucial to define the spatial position of the LiDAR relative to the robot. Although the visual URDF model of the robot dog does not include the 3D mesh of the sensor (a purely aesthetic absence that in no way affects the performance of *NvBlox*), the Transform (TF) system must know the exact physical offset between the components. Through manual measurement, the height of the LiDAR's optical center relative to the back was found to be 12 centimeters. A dedicated static transform was therefore added, as illustrated in Listing 5.6, adding a component to the previously shown tree, which now becomes as shown in Figure 5.8.

```
1     static_tf_trunk_to_lidar = Node(
2         package='tf2_ros',
3         executable='static_transform_publisher',
4         name='static_tf_trunk_to_lidar',
5         arguments=['0', '0', '0.12', '0', '0', '0', 'trunk', '
    rslidar'],
6         parameters=[{'use_sim_time': False}]
7     )
```

**Listing 5.6:** Definition of the static transform (*Static TF*) for the spatial alignment between the robot chassis and the LiDAR sensor's optical center.

## 5.4 Real-world results

This Section illustrates the experimental activity conducted in the laboratory within a test area equipped with the *Vicon* motion capture system. The goal of the data collection is not limited to demonstrating the correct operation of the volumetric reconstruction and odometry algorithm in a real-world context—characterized by physical sensors and non-simulated dynamics—but specifically aims at a comparative evaluation between the two available positioning systems. Although the Vicon system guarantees sub-millimeter precision, its use is strictly tied to the presence of a pre-configured and calibrated infrastructure, thus making it unsuitable




**(a) Vicon Motion Capture Configuration** **(b) KISS-ICP Odometry Configuration**

**Figure 5.8:** Comparison of the complete TF trees in the two experimental configurations. Both cases highlight the integration of the new LiDAR sensor (`rslidar`, highlighted in orange), statically linked to the robot's trunk (`trunk`) in parallel with the pre-existing kinematic macro-structure.

for *out-of-the-box* applications. The integration of *KISS-ICP*, on the other hand, allows operating in unknown environments without preliminary configurations, albeit introducing critical issues related to the propagation of estimation errors (**drift**), which will be analyzed in detail in the following paragraphs.

To ensure the statistical validity of the analyses, several *rosbags* were recorded following differentiated trajectories. The scenario was enriched with physical obstacles strategically placed at the edges of the Vicon area to keep the quadruped's central movement line clear (Figure 5.9). During the acquisition phase, only LiDAR and Vicon data were saved. This methodological choice provided the flexibility to run the *KISS-ICP* node both *online* and *offline* during the reconstruction phase. Consequently, it became possible to isolate and accurately measure the increase in GPU utilization required by the algorithm compared to the passive Vicon *baseline*, whose computation takes place on a dedicated external workstation.

The robot executed four movement profiles, distinct in complexity and duration:

- **linear_motion**: uninterrupted unidirectional linear trajectory (duration: 20s);
- **linear_with_pitch**: linear movement interspersed with two stops, during which the robot performs anteroposterior *pitch* variations (duration: 35s);
- **linear_multi_maneuvers**: linear path with complex static maneuvers integrating lateral inclinations, rotations, and height variations, aiming to maximize sensor coverage (duration: 75s);
- **bidirectional_cage_tour**: complete and bidirectional tour of the test area with frequent stops and diverse maneuvers (duration: 100s).

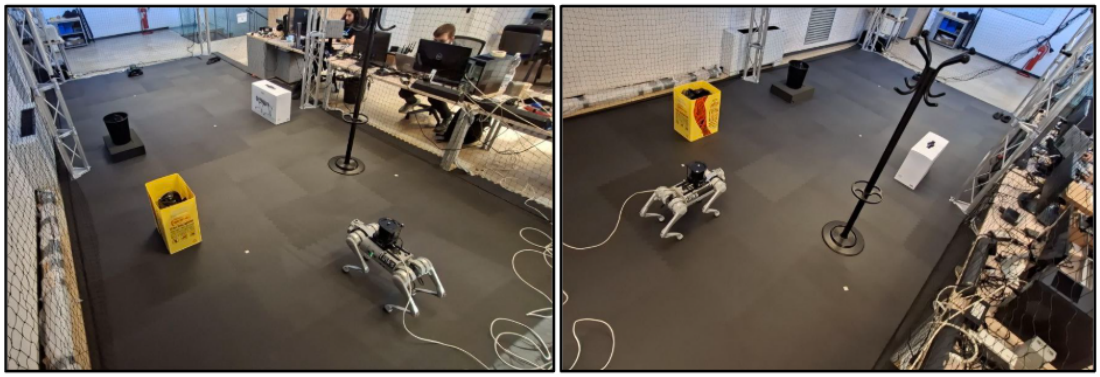


**Figure 5.9:** Experimental configuration of the Vicon cage: positioning of perimeter obstacles and starting point of the quadruped robot.

The intent of such maneuvers is to stress-test the *KISS-ICP* algorithm under limit conditions, to evaluate the extent to which sudden movements and attitude variations affect pose estimation. Before proceeding with quantitative analysis, the system was qualitatively validated in real time, as shown in Figure 5.10, confirming the visual coherence between the robot's kinematics in the physical world and the map update in the 3D visualizer.

Figure 5.11 shows the final reconstruction in the graphical environment. Although it confirms the overall effectiveness of the method, the visualization highlights a lower precision compared to the simulated environment. This discrepancy is attributable both to the physical limits of the used LiDAR sensor and to the absence of chromatic data (*RGB*), which in simulation greatly facilitated surface discrimination. Furthermore, the pose estimation—evaluated according to the two previously discussed methodologies—is severely tested by the complex kinematics of the quadruped: the impulsive movements of the legs induce **significant oscillations** of the sensor during data acquisition.

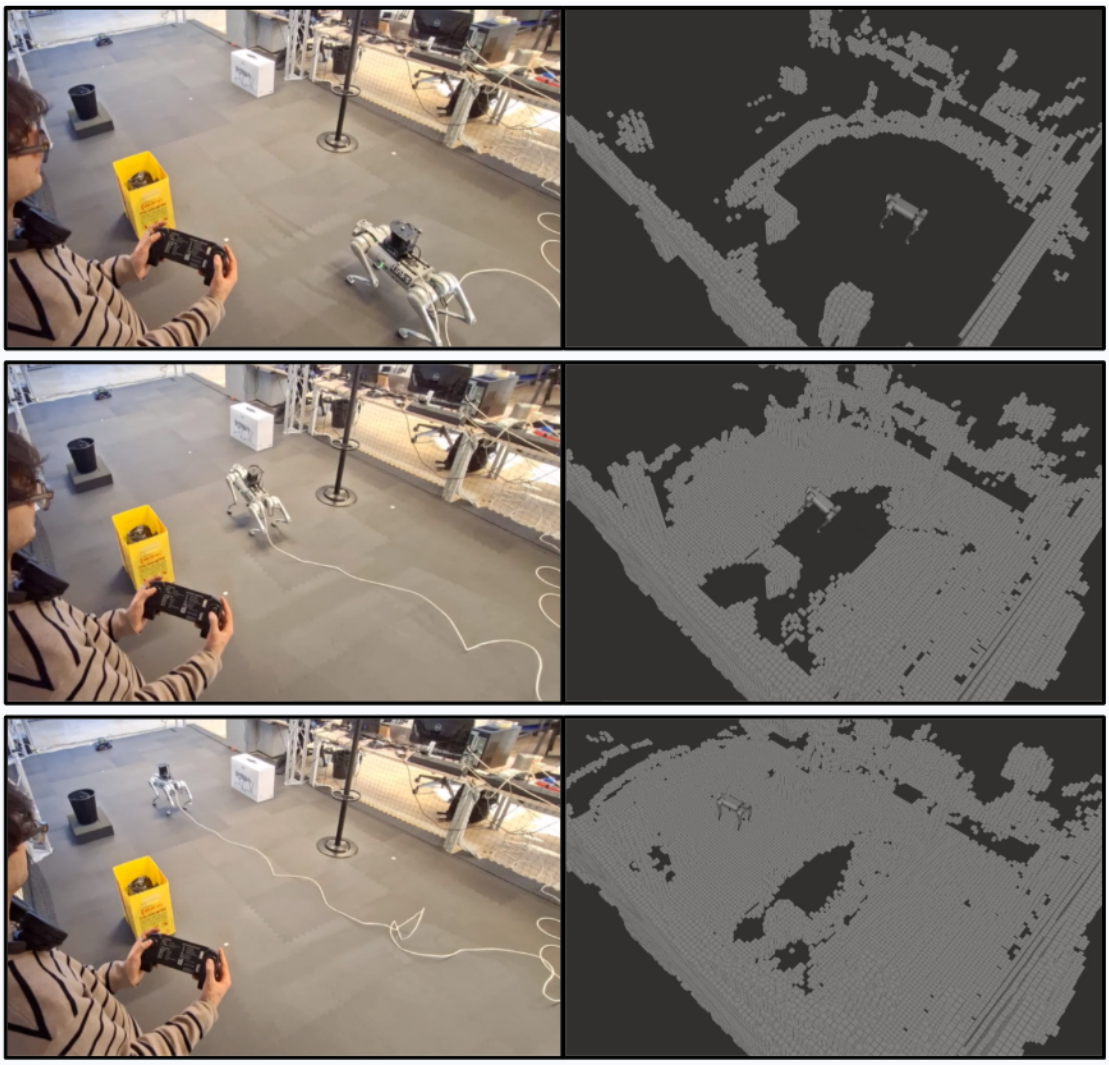


**Figure 5.10:** Real-time visual validation of the reconstruction: comparison between the physical movement of the robot (left) and the real-time updated 3D reconstruction output (right).

This phenomenon is evident in the reconstruction, where some obstacles appear thinned or, in the case of reduced thickness, completely omitted. During navigation, in fact, minimal errors in the LiDAR pose estimation generate a spatial misalignment of the scans (*misregistration*). Consequently, shifted measurements induce the system to map areas previously registered as *occupied voxels* as *free space*. This erroneous overwriting causes the progressive volumetric erosion of larger obstacles and the deletion of smaller ones. In light of these geometric alterations, which are strictly dependent on localization degradation, the introduction of **objective**

**numerical metrics** became necessary to rigorously evaluate the odometry quality.

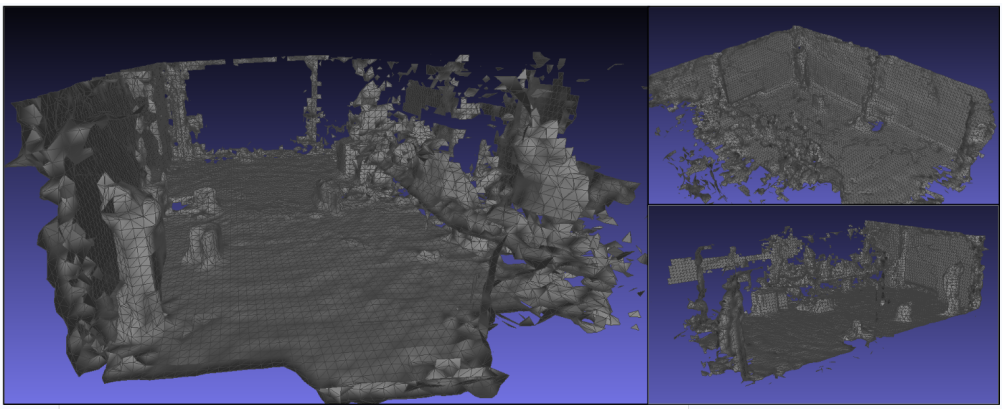


**Figure 5.11:** 3D reconstruction of the test area visualized in Meshlab: detail of the geometry obtained through the integration of LiDAR scans.

The quantitative analysis focused on the point-to-point distance between the trajectories generated via Vicon and those estimated by *KISS-ICP*. As shown in Figure 5.12, by evaluating the **mean distance** and the **standard deviation** as the sequence duration varies, the drift phenomenon clearly emerges. As the recording time increases, the reconstruction performed using *KISS-ICP* diverges from the one created using the *Vicon* system, increasing in both mean and standard deviation. The algorithm's pose estimation is based on the previous position, which, if estimated with an error, will retain and propagate this error into subsequent measurements, resulting in its progressive accumulation.

In parallel, Figure 5.13 shows the analysis as a function of voxel size, highlighting how a **lower spatial resolution** (larger voxels) amplifies the metric error. An imprecise pose estimation, in fact, generates a more pronounced geometric impact if projected onto large voxels, increasing the risk of volumetric discrepancies that could compromise autonomous navigation and trajectory planning (*motion planning*).

The overall results, summarized in the histogram in Figure 5.14 and in Table 5.1, allow for more precise observations through a detailed analysis of the numerical values. By isolating the performance at an equal spatial resolution and considering the optimal results obtained with a voxel size of 0.02 m, it becomes clear that the error does not depend solely on temporal duration, but is strongly linked to the kinematic characteristics of the motion:

- ***linear_motion* (1.87 cm)**: This represents the ideal scenario where the algorithm performs at its best. Given that it is a pure and simple linear translation, the odometry estimates from the two approaches are almost entirely comparable; the recorded error is present but remains strictly marginal.

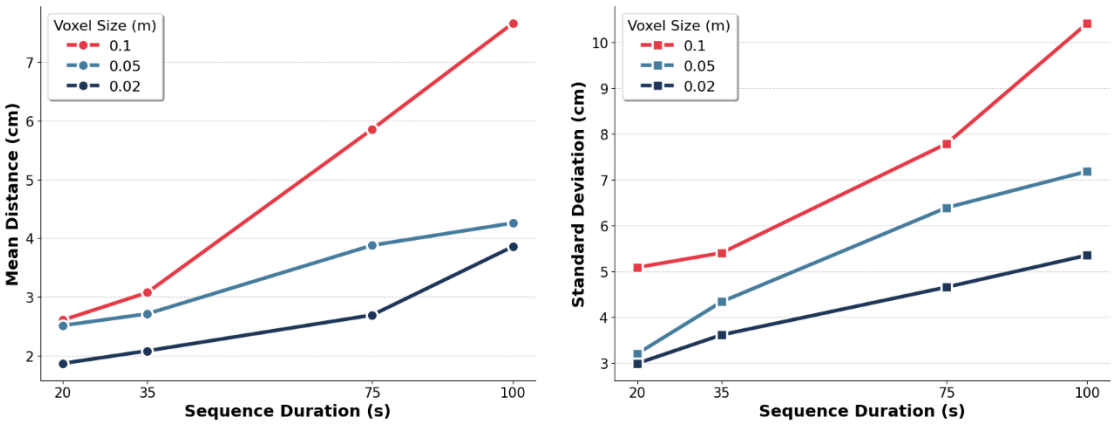



**Figure 5.12:** Mean distance (left) and standard deviation (right) between Vicon and KISS-ICP odometries reconstructions as the recording duration increases.

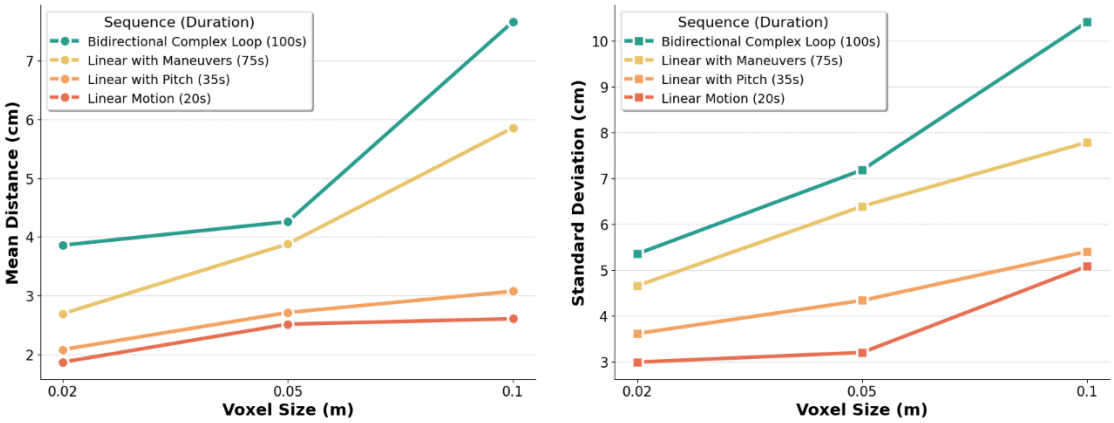



**Figure 5.13:** Mean distance (left) and standard deviation (right) between Vicon and KISS-ICP odometries as voxel size increases.

- ***linear_with_pitch* (2.08 cm)**: The introduction of pitch demonstrates that the drift consists of both a translational and a rotational component. However, although the addition of inclinations might seem kinematically more problematic, the error increase is minimal. This suggests that *KISS-ICP* is able to compensate for and calculate angular variations with considerable stability compared to continuous pure translations.

- ***linear_multi_maneuvers* (2.69 cm)**: The analysis of this sequence confirms the previous deduction. Despite the recording duration being more than double that of the preceding sequences, the metric error does not undergo a proportional increase. This unequivocally highlights how the estimation

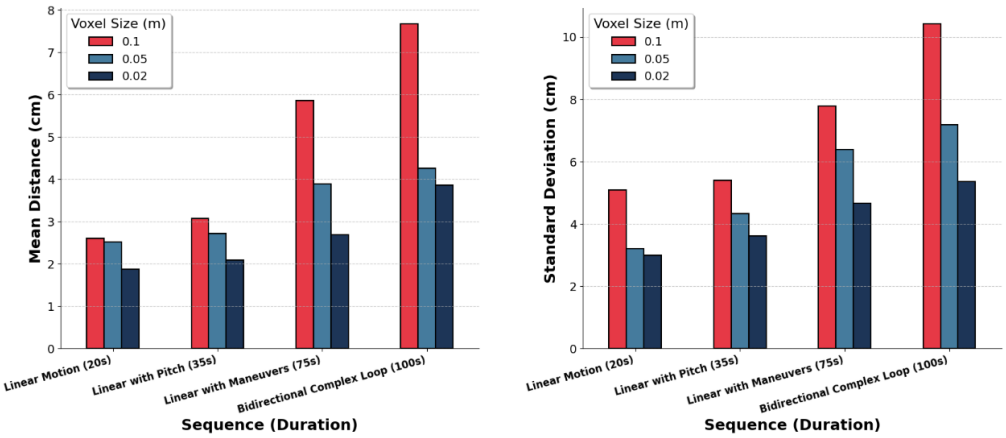



**Figure 5.14:** Global comparison of mean error and variance for all tested sequences and resolutions.

**Table 5.1:** Experimental results of distance metrics (Mean and Standard Deviation) expressed in centimeters.

| Rosbag Sequence | Voxel Size (m) | Avg Dist (cm) | Std Dev (cm) |
|---|---|---|---|
| linear_motion | 0.10 | 2.61 | 5.09 |
| | 0.05 | 2.51 | 3.20 |
| | 0.02 | 1.87 | 2.99 |
| linear_with_pitch | 0.10 | 3.07 | 5.40 |
| | 0.05 | 2.71 | 4.34 |
| | 0.02 | 2.08 | 3.61 |
| linear_multi_maneuvers | 0.10 | 5.86 | 7.79 |
| | 0.05 | 3.88 | 6.39 |
| | 0.02 | 2.69 | 4.66 |
| bidirectional_cage_tour | 0.10 | 7.66 | 10.42 |
| | 0.05 | 4.26 | 7.18 |
| | 0.02 | 3.86 | 5.35 |

degradation is not a strictly linear function dependent solely on execution time, but is mitigated or exacerbated by the specific type of maneuvers performed.

- ***bidirectional_cage_tour* (3.86 cm)**: This trajectory provides the final validation of the drift dynamics. With a time increase of only 25 seconds compared to the *multi_maneuvers* sequence, there is a substantial spike in error that breaks the previous pattern. This degradation is justified by the overall extent of the path: while the maneuvers remain similar, the translated distance doubles, accumulating greater uncertainty in the pose estimation.

Furthermore, a crucial element for volumetric reconstruction, this is the only recording that includes a return path (loop closure). Due to the operational architecture of *nvblox*, the repeated observation of the same space leads to the overwriting and updating of voxels. Since the odometric error has accumulated over time, projecting the "new" readings (affected by a larger drift) onto the previously mapped space generates a pronounced geometric misalignment, severely emphasizing volumetric differences during the metric evaluation phase.

Having understood the potential of the software-only approach in the context of visual reconstruction — which, importantly, allows the system to operate without external hardware infrastructure such as a *Vicon* tracking cage, thus freeing the robot from structured environments — it is useful to extend the analysis to different application scenarios. Independent of the goal of producing a definitive global map, *NvBlox* can be employed exclusively for obstacle awareness. In this context, the accumulation of drift becomes significantly less problematic. Scans can be focused solely on detecting obstacles along the path through local, temporary reconstructions. By utilizing specific decay parameters, as described in Subsection 4.3.3, the map can be made "volatile," ensuring that the reconstructed space continuously updates and is used solely for reactive navigation.

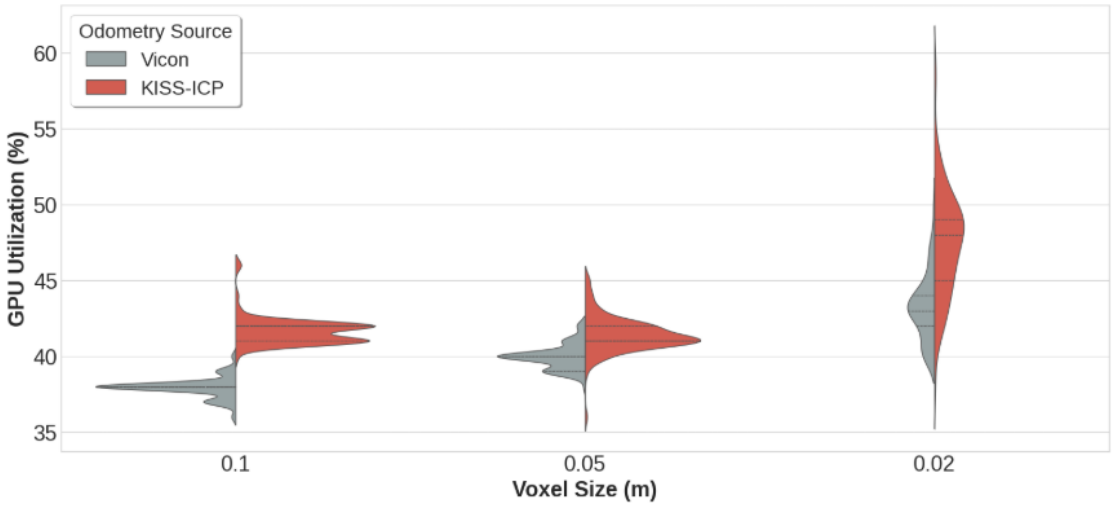



**Figure 5.15:** Statistical distribution (Violin Plot) of GPU load: comparison between the passive baseline and active processing with KISS-ICP.

Given this use case, a second type of analysis is required to determine whether the software odometry processing heavily impacts the GPU load, potentially depriving the reconstruction framework of necessary hardware resources. Figure 5.15 illustrates the distribution of GPU utilization percentages during the reconstruction

phase, varying both the odometry source and the voxel size. Here, the values labeled *Vicon* serve as a passive baseline for comparison: since the motion capture system is processed externally (*off-board*) and does not burden the local PC, this comparison effectively isolates the net computational impact of running *NvBlox* and *KISS-ICP* together compared to running *NvBlox* alone.

The tests, performed on the complex *bidirectional_cage_tour* sequence, indicate that running *KISS-ICP* entails a marginal and entirely sustainable increase in computational load. As shown in Table 5.2, when comparing the average GPU usage, the algorithm adds a negligible burden: at a 0.10 m voxel size, it increases from 37.94% to 41.64% (approximately +3.7%), while at the highest resolution (0.02 m), it goes from 43.21% to 47.49% (only a +4.3% additional load). These data mathematically demonstrate that the primary resource consumer is the volumetric integration process, not the odometry estimation.

**Table 5.2:** GPU computational load: average and maximum peaks recorded during the tests.

| Odometry Method | Voxel Size (m) | Avg GPU (%) | Max GPU (%) |
|---|---|---|---|
| **Vicon** | 0.10 | 37.94 | 40.0 |
| | 0.05 | 39.96 | 42.0 |
| | 0.02 | 43.21 | 50.0 |
| **KISS-ICP** | 0.10 | 41.64 | 46.0 |
| | 0.05 | 41.41 | 45.0 |
| | 0.02 | 47.49 | 59.0 |

Regarding the impact of spatial resolution, the percentage values recorded in the real environment are significantly lower than those measured in the simulated case from the previous chapter. This efficiency stems from the absence of a physics engine: the processing unit does not have to compute the dynamics of actuators and contacts, thus being able to dedicate resources exclusively to spatial perception processes. Finally, analyzing the peak usage as the voxel size decreases, the computational demand logically increases for both methods. *KISS-ICP* reaches a maximum utilization of 59.0% at the 0.02 m resolution (compared to 50.0% for the baseline). However, a peak of 59.0% implies that the GPU retains over 40% of its resources completely free even during moments of maximum stress, definitively confirming the absence of computational *bottlenecks* and validating the robustness of the proposed implementation.

# Chapter 6

# Conclusions and Future Works

This work provides a solid methodological foundation for the integration of 3D reconstruction onto real-world robotic platforms. Rather than an isolated project, this research was conducted as a systematic study to overcome the constraints of complex experimental contexts, such as those provided by the LINKS Foundation. The central objective was the development of a versatile framework: a tool capable of equipping different robotic architectures with 3D spatial awareness, an essential requirement for conscious autonomy.

Given the rapid evolution of NVIDIA technologies and the *nvblox* framework, this study serves as a starting point for future implementations. The following Subsections analyze the goals achieved with respect to the initial objectives, the limitations encountered, and the potential development directions offered by the results obtained.

## 6.1 Achieved Objectives

The objectives set during the initial phase of the research were fully met through a process that combined theoretical analysis with practical experimentation. Specifically, the achievements include:

- **Simulation Environment Development:** The creation of a dedicated scenario in *Gazebo* provided a controlled ecosystem to validate the flexibility of the architecture and the interoperability between various software modules.
- **Framework Integration and Spatial Awareness:** The implementation of *nvblox* in simulation allowed for the generation of the first volumetric

reconstructions, validating the proof-of-concept. In this phase, not only the visual quality of the reconstructions was evaluated, but also the robot's ability to interpret the environment through them was successfully tested, demonstrating how path planning algorithms (e.g., *Nav2*) can utilize the reconstruction to plan safe and optimized paths.

- **Real-World Validation and Pose Analysis:** The transition to the physical platform confirmed the system's robustness in unstructured scenarios. A key objective was the critical comparison between different odometry sources, specifically *KISS-ICP* and the *Vicon* system, analyzing their operational limits and intrinsic divergences to ensure maximum localization precision.

## 6.2 Limitations and Discussion

The implementation of the framework and the subsequent experimental phase highlighted several technical and operational constraints that influenced the system's development. These limitations can be summarized as follows:

- **Computational Requirements and Dependencies:** Adopting *nvblox* imposes stringent hardware constraints, as it is a GPU-accelerated framework requiring specific NVIDIA architectures. Beyond raw computing power, a strong dependency on software compatibility (drivers, CUDA versions, and ROS distributions) emerged. Although the use of *Docker* facilitated the creation of a controlled environment, managing the interfaces between the container and the hardware remains a critical element that limits immediate system portability.
- **Simulation Criticalities:** Despite using a lightweight simulator like *Gazebo*, the virtual testing phase proved paradoxically more demanding than the real-world application. This is due to the computational load required to physically emulate the environment and sensors, saturating the GPU regardless of graphical rendering. Furthermore, physical discrepancies were encountered, such as wheel drift and instabilities in the motion model, which complicated the algorithm's calibration compared to observations on the physical platform.
- **Sensory Quality and Reconstruction:** Experiments confirmed that reconstruction quality is highly sensitive to hardware performance. The use of a LiDAR with limited resolution showed room for improvement in map accuracy, while the absence or limitation of chromatic data (color) made the visual and semantic interpretation of the surrounding space less intuitive.
- **Robotic Platform Accessibility:** A significant limitation concerned the inability to fully interface with the internal components of the quadruped robot.

The lack of access to integrated cameras and the inability to communicate directly with the motor control boards prevented the development of a comprehensive multi-sensor analysis and the execution of autonomous movement commands from the PC. Consequently, it was not possible to test *Nav2* integration in the real environment, limiting the validation of autonomous navigation based on the performed reconstruction.

These elements highlight that while the framework is robust, its practical effectiveness depends strictly on the openness of the robotic architectures used and the balance between available computational resources and the required reconstruction fidelity.

## 6.3 Future Works

The results obtained in this work pave the way for several research lines aimed at increasing system autonomy and fully exploiting the proposed framework's modularity. The main directions for future evolution are as follows:

- **On-board Integration and Hardware Optimization:** Overcoming current physical constraints requires integrating the system directly onto the robot's onboard computer. Local execution of the architecture would eliminate dependence on external cabling and high-latency wireless communication while allowing the use of the platform's pre-installed sensors. This approach would reduce the external mechanical load, improving the quadruped's motion dynamics and stability.
- **Autonomous Exploration and Next-Best-View:** A significant evolution involves moving from passive navigation to active exploration. 3D reconstruction should be viewed not just as an output but as an input for *Active Perception* algorithms. In this scenario, the robot analyzes the partial map to identify low-resolution or unexplored areas, planning optimal movements to complete the reconstruction as efficiently as possible.
- **Multi-Agent Systems and Ground-Air Cooperation:** Extending the framework to heterogeneous systems represents a highly interesting frontier. Integrating data from a drone could fill the perspective gaps of the ground robot, providing overhead coverage of obstacles and allowing for complete volumetric reconstruction of vertical structures or areas otherwise inaccessible from the ground.
- **Dynamic Environment Management:** Leveraging the advanced capabilities of *nvblox*, a natural continuation of this work involves implementing

algorithms for managing moving objects. This would allow the system to distinguish between static scene elements and temporary obstacles (such as people or other robots), making the reconstruction reliable even in crowded or constantly changing real-world contexts.

- **Advanced Multi-Sensor Analysis:** Full access to the robot's interfaces would enable deeper sensor fusion, combining LiDAR data, RGB-D cameras, and inertial information (IMU). Adding chromatic data to the polygonal reconstruction would not only improve the model's aesthetics but also provide a fundamental semantic basis for object recognition tasks and complex interaction with the environment.

In conclusion, the modularity of the developed architecture ensures compatibility with these future extensions, serving as a fundamental stepping stone toward the creation of robotic systems capable of true spatial and operational awareness.

# Bibliography


[1] Armin Hornung, Kai M. Wurm, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. «OctoMap: An Efficient Probabilistic 3D Mapping Framework Based on Octrees». In: *Autonomous Robots* 34.3 (2013), pp. 189–206. DOI: 10.1007/s10514-012-9321-0.

[2] Sebastian Thrun. «Learning Occupancy Grid Maps with Forward Sensor Models». In: *Autonomous Robots* 15.2 (2003), pp. 111–127. DOI: 10.1023/A:1025584807625. URL: https://doi.org/10.1023/A:1025584807625.

[3] Svetoslav Noykov and Christo Roumenin. «Occupancy Grids Building by Sonar and Mobile Robot». In: *Robotics and Autonomous Systems* 55.12 (2007), pp. 915–927. DOI: 10.1016/j.robot.2006.06.004. URL: https://doi.org/10.1016/j.robot.2006.06.004.

[4] Mingkang Li, Zhaofei Feng, Martin Stolz, Martin Kunert, Roman Henze, and Ferit Küçükay. «High Resolution Radar-based Occupancy Grid Mapping and Free Space Detection». In: *Proceedings of the 15th International Conference on Informatics in Control, Automation and Robotics (ICINCO)*. 2018, pp. 70–81. DOI: 10.5220/0006667300700081. URL: https://doi.org/10.5220/0006667300700081.

[5] Sooraj Sunil, Saeed Mozaffari, Rajmeet Singh, Behnam Shahrrava, and Shahpour Alirezaee. «Feature-Based Occupancy Map-Merging for Collaborative SLAM». In: *Sensors* 23.6 (2023), p. 3114. DOI: 10.3390/s23063114. URL: https://doi.org/10.3390/s23063114.

[6] Hans P. Moravec and Alberto Elfes. «High Resolution Maps from Wide Angle Sonar». In: *Proceedings of the 1985 IEEE International Conference on Robotics and Automation (ICRA)*. Vol. 2. 1985, pp. 116–121. DOI: 10.1109/ROBOT.1985.1087316.

[7] Kai M. Wurm, Armin Hornung, Maren Bennewitz, Cyrill Stachniss, and Wolfram Burgard. «OctoMap: A Probabilistic, Flexible, and Compact 3D Map Representation for Robotic Systems». In: *Proc. IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Vol. 2. 2010.

[8] Wikipedia contributors. *Voxel*. 2025. URL: https://en.wikipedia.org/wiki/Voxel.

[9] Anderson A. de Santana Souza and Luiz Marcos G. Gonçalves. «Occupancy-elevation grid: an alternative approach for robotic mapping and navigation». In: *Robotica* 34.11 (2016), pp. 2592–2609. DOI: 10.1017/S0263574715000235. URL: https://doi.org/10.1017/S0263574715000235.

[10] Konstantin Schauwecker and Andreas Zell. «Robust and Efficient Volumetric Occupancy Mapping with an Application to Stereo Vision». In: *Proceedings of the 2014 IEEE International Conference on Robotics and Automation (ICRA)*. Hong Kong, China, 2014, pp. 6102–6107. DOI: 10.1109/ICRA.2014.6907758.

[11] Ayush Tewari et al. «State of the Art on Neural Rendering». In: *arXiv preprint arXiv:2004.03805* (2020). DOI: 10.48550/arXiv.2004.03805. URL: https://arxiv.org/abs/2004.03805.

[12] Ayush Tewari et al. «Advances in Neural Rendering». In: *arXiv preprint arXiv:2111.05849* (2021). DOI: 10.48550/arXiv.2111.05849. URL: https://arxiv.org/abs/2111.05849.

[13] Justus Thies, Michael Zollhöfer, and Matthias Nießner. *Neural Rendering – Tutorial*. Tutorial on Neural Rendering. 2020.

[14] Xinkai Yan, Jieting Xu, Yuchi Huo, and Hujun Bao. «Neural Rendering and Its Hardware Acceleration: A Review». In: *arXiv preprint arXiv:2402.00028* (2024). DOI: 10.48550/arXiv.2402.00028. URL: https://arxiv.org/abs/2402.00028.

[15] RebusFarm. *3D Neural Rendering and Its Real-Time Power*. 2025. URL: https://rebusfarm.net/blog/3d-neural-rendering-and-its-real-time-power.

[16] Aufait Technologies. *An Introduction to Neural Rendering and Its Applications*. 2025. URL: https://aufaittechnologies.com/blog/neural-rendering-web-development-ai-real-time-visuals/.

[17] Ben Mildenhall, Pratul P. Srinivasan, Matthew Tancik, Jonathan T. Barron, Ravi Ramamoorthi, and Ren Ng. «NeRF: Representing Scenes as Neural Radiance Fields for View Synthesis». In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2020. URL: http://arxiv.org/abs/2003.08934.

[18] Bernhard Kerbl, Georgios Kopanas, Thomas Leimkühler, and George Drettakis. «3D Gaussian Splatting for Real-Time Radiance Field Rendering». In: *ACM Transactions on Graphics* 42.4 (July 2023). URL: https://repo-sam.inria.fr/fungraph/3d-gaussian-splatting/.

[19] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar, and Markus Gross. «Surface Splatting». In: *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH 2001)*. 2001, pp. 371–378. DOI: 10.1145/383259.383300. URL: https://dl.acm.org/doi/10.1145/383259.383300.

[20] Wikipedia contributors. *Signed distance function*. 2024. URL: https://en.wikipedia.org/wiki/Signed_distance_function.

[21] Helen Oleynikova, Zachary Taylor, Marius Fehr, Juan Nieto, and Roland Siegwart. «Voxblox: Incremental 3D Euclidean Signed Distance Fields for On-Board MAV Planning». In: *arXiv preprint arXiv:1611.03631* (2017). DOI: 10.48550/arXiv.1611.03631. URL: https://arxiv.org/abs/1611.03631.

[22] Alexander Millane, Helen Oleynikova, Emilie Wirbel, Remo Steiner, Vikram Ramasamy, David Tingdahl, and Roland Siegwart. «nvblox: GPU-Accelerated Incremental Signed Distance Field Mapping». In: *arXiv preprint arXiv:2311.00626* (2024). DOI: 10.48550/arXiv.2311.00626. URL: https://arxiv.org/abs/2311.00626.

[23] Donald G. Bailey. «An Efficient Euclidean Distance Transform». In: *Combinatorial Image Analysis, 10th International Workshop, IWCIA 2004, Auckland, New Zealand, Proceedings*. Vol. 3322. Lecture Notes in Computer Science. Springer, 2004, pp. 394–408. DOI: 10.1007/978-3-540-30503-3_28.

[24] Open Robotics. *Understanding ROS 2 Nodes*. 2024. URL: https://docs.ros.org/en/rolling/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html.

[25] Black Coffee Robotics. *Isaac Sim: Photorealistic Rendering for Next-Gen Robot Development*. 2023. URL: https://medium.com/black-coffee-robotics/isaac-sim-photorealistic-rendering-for-next-gen-robot-development-37ca8186c1d9.

[26] Wikipedia contributors. *Layered costmaps*. 2024. URL: https://en.wikipedia.org/wiki/Layered_costmaps.

[27] Isaac Arogbonlo. *Differences between a Dockerfile, Docker Image and Docker Container*. URL: https://cto.ai/blog/docker-image-vs-container-vs-dockerfile/.

[28] Articulated Robotics. *Articulated Robotics [YouTube channel]*. URL: https://www.youtube.com/@ArticulatedRobotics.

[29] Josh Newans. *my_bot: Template for a ROS package*. URL: https://github.com/joshnewans/my_bot.

[30] Articulated Robotics. *Fix Problem: Wheel Slippage*. YouTube video, section "Fix Problem: wheel slippage". 2022. URL: https://www.youtube.com/watch?v=8ByoP_oSdno.

[31] NVIDIA Corporation. *Getting Started with Isaac ROS*. 2025. URL: https://nvidia-isaac-ros.github.io/getting_started/index.html.

[32] NVIDIA Corporation. *Isaac ROS Nvblox*. 2025. URL: https://nvidia-isaac-ros.github.io/repositories_and_packages/isaac_ros_nvblox/index.html.

[33] RoboSense. *RS-Helios-16P User Manual*. Technical datasheet and user manual for the RS-Helios-16P LiDAR. RoboSense. 2022.

[34] Génération Robots. *LiDAR RS-Helios-16P RoboSense*. URL: https://www.generationrobots.com/it/403970-lidar-rs-helios-16p-robosense.html.

[35] TOD System. *Unitree Go1 Robot quadrupede*. URL: https://store.todsystem.com/en/product/unitree-go1/.