UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

DEPARTMENT OF INFORMATION ENGINEERING

MASTER DEGREE IN

COMPUTER ENGINEERING

"Real-Time Object Tracking and Pose Prediction for Robotic Manipulation using ROS"

Supervisor: Prof. Giovanni Boschetti

Candidate: Michele Zadro

Academic Year 2023 – 2024

Graduation date 25-11-2024

Abstract

This thesis explores the use of an eye-in-hand camera system, where the camera is mounted on the end-effector of a robotic arm, to perform a pick-and-place routine. The objective is to develop techniques that enable the robotic arm to pick objects from a conveyor belt efficiently. The key steps involved in this process include camera calibration, object detection and tracking, as well as the implementation of an object following and picking algorithm. Each of these steps is addressed in detail, providing a comprehensive approach to robotic manipulation using this camera system.

Dedication

To my family, whose support and belief in my abilities have uplifted me throughout this journey. Your encouragement has strengthened my confidence and made this achievement possible.

To all my friends, always there to bring a smile, share a laugh, and walk this journey with me, making every step more engaging and meaningful.

Contents

Chapter 1

Introduction

1.1 Problem Statement

The objective of this project was to develop a software system capable of controlling a robotic arm to pick up random bottles from a moving conveyor belt, using a camera mounted on the robot's wrist for guidance.

However, due to challenges in acquiring essential materials, such as the conveyor belt, and limitations in testing the software on physical hardware, the project was ultimately implemented using Gazebo, a physics-based simulation software for robotics. This allowed for realistic testing and development despite the lack of physical components.

1.2 Objectives of the Project

For a similar task, a more reliable approach would have been to use a laser distance sensor mounted perpendicular to the axis of the conveyor belt at a known distance from the robotic arm. This sensor would measure the distance to the closest object at various points along the line, providing the positions of the bottles. By integrating this with the conveyor belt encoder (which tracks both speed and position), the system could obtain the precise positions of the bottles in the 2D plane. This approach is ideal for systems that must guarantee high accuracy and speed, though it requires expensive components such as a 2D laser scanner, an infrastructure mounted with high precision (the laser scanner has to be placed above the conveyor belt at a known and precise distance with the base of the robot) and a reliable communication link between the robotic application and the conveyor belt motor encoder.

Given the disadvantages of this setup, such as cost and complexity, it was decided to develop a robotic application that would not require modifications to the existing system. Instead, with only minor adjustments, the developed system can be easily configured to operate in various environments where the conveyor belt is located near the robot platform.

While developing the project, there were several key principles that had to be followed in order to ensure that the final result could be adapted to real-world systems with the necessary precautions:

- **Modularity**: the software should be divided into separate components or modules that can be easily modified or replaced without affecting the entire application.
- **Configurability**: the software should be easy to configure for different applications or environments with minimal adjustments.

1.3 Tools used

The system is composed of two main hardware devices: the Universal Robots UR5e (the robotic arm) and the Robotiq 2f-85 (the gripper). The choice of these components was related to the fact that their manufacturers provide all the necessary resources, including detailed CAD models and control libraries to import them into the simulation software.

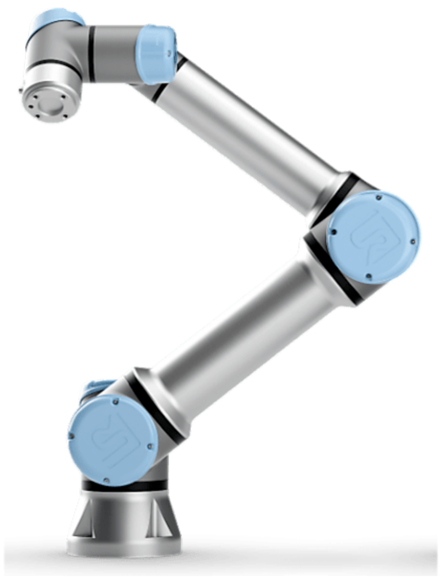


Figure 1.1: Universal Robots UR5e

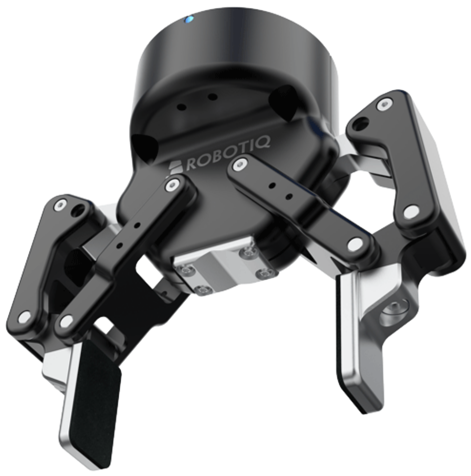


Figure 1.2: Robotiq 2f-85

In addition to the programming language, IDE, and the version control software employed for the project, the two primary components were ROS and Gazebo.

1.3.1 ROS

ROS (Robot Operating System) is an open-source framework designed for robot software development. It provides a collection of libraries and tools that facilitate the creation of robotic systems. ROS is organized into modular units called packages, which are the fundamental blocks of a ROS application. A package can contain various nodes which are executable programs that perform specific tasks, such as reading from sensors, controlling actuators, or performing different algorithms. Nodes can publish information to a topic or subscribe to topics to receive data from other nodes. This structure allows for nodes to operate independently but still communicate effectively. In a way, ROS pushes the user to design modular applications.

There are three main ways ROS nodes can communicate:

- **Messages**: these are the most basic form of communication. Nodes can publish messages to specific topics, and other nodes can subscribe to these topics to receive the messages. Each message has a predefined structure and can contain different types of data (integers, strings, arrays, etc.).

  Example: A camera node publishes image data to the topic */camera/image_raw*, and the main node subscribes to this topic to retrieve and visualize the images.

- **Services**: these use a request-response mechanism to communicate in a synchronous way (the client waits for the server response being blocked). A service client sends a request and the service server processes it and responds. As in messages, the request-response structure is predefined.

  Example: The main node, using the provided service *get_poses*, sends a request to the object detection node, passing an image as input, which will respond with the list of pixel coordinates of the detected objects.

- **Actions**: these are used for tasks that take time and allow for feedback while they are executed. Unlike services, actions support asynchronous communication, allowing the client to receive feedback during the task and cancel it if necessary.

  Example: The main node uses the *pick_action*, of the node in charge to perform the pickup movement, passing in the request field the position where to pick the object and the timestamp when to close the gripper to grasp that object. While performing the motion, the pick node sends feedback to the main node (moving, grasping etc.).

1.3.2 Gazebo

Gazebo is an open-source robotics simulation software that integrates with ROS, providing a rich environment for testing and developing robotic applications in a virtual space. It allows developers to create complex 3D worlds and simulate there the physical behavior of robots, including their interactions with objects and the environment. Gazebo supports advanced features like a physics engine for realistic motion, sensor simulation (e.g. cameras, LiDAR), and customizable plugins for extending functionality (e.g. conveyor belt plugin).

In this project, Gazebo played a crucial role by allowing the simulation of several key components, including:

- the UR5e robotic arm from Universal Robots, to test movement and pick-up operations

- the camera mounted in the wrist of the robot, to simulate what a real camera would observe and thus perform object detection and pose reconstruction

- the platform on which the robot was placed, to verify proper positioning and workspace

- the conveyor belt, to simulate real-world scenarios where objects are transported

- the bottles that were spawned on the conveyor belt, to test the robot's interaction with objects, including detecting, tracking, and manipulating them

Through Gazebo, various scenarios can be simulated to evaluate the robot's performance and accuracy, which significantly reduced development time and removed the risks of real-world testing.

1.4 Chapter descriptions

- **Chapter 2** will explain which are the main components of the application, how they work from a general point of view and how they are connected to form the system chain. Also, this chapter will illustrate how all these components are distributed across the nodes in the ROS project.
- **Chapters 3, 4 and 5** will explore deeply the components discussed in Chapter 2, showing how they work in detail, their limitations and their advantages.
- **Chapter 6** will conclude with the final analysis of the system as a whole, summarizing the key findings from the previous chapters. It also presents discussions on potential improvements and future directions for development, offering a conclusive reflection on the project outcomes.

Chapter 2

System Architecture

This chapter will address the following questions:

- How should the system work in general?
- What are the main components of the system?
- How are these components imported into the simulation?
- How is the ROS project structured?

2.1 System Overview

There are two main components that are part of the system:

- The conveyor belt with the objects (bottles) that slide on top of it
- The robotic arm, with the camera and the gripper mounted on top of it, used to pick up the objects

The robot should be able to catch the desired object (or a random one) without using any external information such as conveyor speed, distance of bottles from a known point etc. Therefore, it has to retrieve all the necessary information to grasp the object using only the camera mounted on the robot's wrist.

The main steps to reach the goal are the following:

1. **Camera calibration**. The camera plays a crucial role in this type of systems, as it is used to reconstruct the pose of an object using only its pixel coordinates. In order to reconstruct the 3D poses (written with respect to a fixed base frame), the camera needs to be calibrated, which means to evaluate the extrinsic and intrinsic parameters.
2. **Object detection**. Before converting pixel coordinates into 3D poses, the objects visible to the camera must first be identified and their pixel coordinates extracted. Object detection is the process by which, given an image of the observed scene, it outputs a list

of pixel coordinates representing the locations in the image where the target objects are situated. In the proposed solution, this process makes use of a famous deep convolutional neural network: YOLO v8.

3. **Pose reconstruction**. This is the process of converting pixel coordinates into 3D poses using previously extracted intrinsic and extrinsic parameters. Knowing the 3D pose of the objects, rather than just their position in the image, is beneficial in many ways. For instance, calculating the velocities of the objects becomes straightforward by using consecutive poses and the time interval between them.

4. **Object tracking**. Having a list of detected object poses for each image is not sufficient for this application; it is necessary to have a system that matches each pose to a unique object, for example, by assigning an integer ID. Without doing so, it would not be possible to answer many useful questions, such as: ”Given two images and the time interval between them, what is the average velocity of bottle X?” There would be only two lists of poses, with no information about how they correspond to each other.

   To achieve this, the application includes a module that continuously reads the list of detected object poses and attempts to find the best match with the previously observed poses. This task is called object tracking.

5. **Pose prediction**. To successfully pick up a continuously moving object, it is not enough to know only its current and past positions. Instead, it is necessary to predict where the object will be after x seconds. This allows the robot to begin moving towards the predicted position and arrive there before the time limit has passed, so it can grab the object when it is directly under the gripper.

   The pose prediction module has been implemented using a custom algorithm that analyzes the trajectories of past bottles to estimate where the target bottle will be.

6. **Object picking**. This part of the application is responsible for moving the robot to the predicted position and waiting there until the object is under the gripper. To determine when to close the gripper, the system needs to calculate the time stamp when the object is expected to reach the predicted position subtracted by the time required for the gripper to close. Once the current time stamp matches the previously calculated one, the gripper is activated to close. Afterwards, the arm should lift and return to its designated position, where it can open the gripper and release the object.

2.2 Starting the Simulation

To develop the application effectively, it is beneficial to have a method for testing the functionality of each component. This is where the simulation software becomes valuable. With such a software, it is possible to simulate the bottles sliding along a conveyor belt, allowing the emulation of what the camera would capture in real-world conditions.

Even though modern simulation softwares can achieve graphics that closely resemble reality, they will never fully replicate real-world conditions. Therefore, it is important to keep in mind that the object detection module may need to be adapted when transitioning the application to a real-world system.

To start a simulation, ROS launch files are highly useful because they allow you to launch multiple nodes and configure parameters simultaneously. By using a launch file, ROS can automatically initialize Gazebo with the specified settings, simplifying the setup and saving time.

Before writing the launch file, it is necessary to define the description files of the robotic station and the conveyor belt. A description file, called URDF (Unified Robot Description Format), is a text file that describes the system's physical design in terms of its links, joints, and properties like shape, size, and movement constraints. This type of file can include other URDF files and thus be composed of several stand alone components that can be interconnected through joints of various types (fixed, prismatic, revolute). In this application, these components were:

- The **Universal Robots UR5e** description file, provided by the manufacturer (as URDF).
- The **Robotiq 2f-85** URDF which provides the description of the gripper that will be mounted on the robot's wrist.
- The **camera** description file, composed of a simple invisible link where the camera sensor (provided by Gazebo) is attached.
- The **base station** description file, in which the visual and collision components are designed using a 3D CAD software and simple links are then specified for the base and surface frames.
- The **conveyor belt** description file, composed of a simple rectangular box in which a gazebo plugin is imported to allow the sliding motion of any object placed on top.

At this point the necessary components can be included in the launch file that runs the simulation. These components include:

- The empty world launch file. This is responsible for launching a Gazebo simulation with no objects or models inside. (Note that any launch file can include other launch files).
- The final customized *robot_description* file (still in URDF format) in which almost all the previous description files are included and their links are joined in a meaningful way:
  - The *base_link* of the base station is attached with a fixed joint to the *world* frame.
  - The *base_link* of the robot is attached with a fixed joint to the *surface_link* of the base station.
  - The *camera_link* is attached to the *tool0* link of the robot (its last link) at a distance of 10cm along the z-axis.

  - The *base_link* of the gripper is attached to the *tool0* link of the robot.

- The *controller_manager* node: responsible for setting up the controllers for the robot. These controllers include the one for the robotic arm and the one for the gripper, both are provided by the manufacturer and are needed to simulate the real behavior of these components.

- The *robot_state_publisher* node: responsible for publishing the state of each link in the robot based on its joint positions. This includes creating the TF tree that defines the spatial relationships between all parts of the robot, which is essential for visualizing the robot's current pose and enabling accurate motion planning.

- The MoveIt components: responsible for the motion planning of both the manipulator and the gripper, ensuring smooth and efficient movement without collisions with the surrounding environment.

- The conveyor belt description file, used to spawn the object (in front of the robot) where elements slide from right to left.

2.3 ROS Project Structure

2.3.1 Overview

A ROS project is organized into packages: folders that hold everything that is needed for a specific part of the robot's application. A package typically contains:

- **Code (nodes)** (Python or C++) for specific tasks (like controlling the robot or reading from the camera sensor), placed in the *src* or *script* folder (depending on the programming language).

- **Launch files** to start parts of the robot system, placed in the *launch* folder.

- **Dependencies** on other packages it relies on to work, specified in the *CMakeLists.txt* file of the package.

If the package defines its own custom communication formats for its nodes, it may contain also:

- **msg** folder which contains custom message definition files using a predefined format. Considering the example of a node that has to publish information about the position of an object, there will be a message file in the path: *package_name/msg/Position.msg* with the following content:

```
float64 x     # X-coordinate in meters
float64 y     # Y-coordinate in meters
float64 z     # Z-coordinate in meters
```

- **srv** folder which contains service definition files, allowing nodes to request a task and wait for a response. For example, a robot node may need to perform a rotation. The service file located at *package_name/srv/Rotate.srv* could have the following structure:

```
# Request
float64 angle   # angle in radians to rotate
---
# Response
bool success    # true if rotation was completed
   successfully
```

- **action** folder which defines action files for tasks that take time and may require ongoing feedback. For instance, a robot may need to move to a specific location with feedback provided along the way. The action file could be stored in *package_name/action/MoveTo.action* and would contain:

```
# Goal
float64 x       # target x-coordinate
float64 y       # target y-coordinate
---
# Result
bool success    # true if the robot reached the target
---
# Feedback
float64 distance_remaining   # distance left to reach
   target
```

Sometimes ROS packages are used to describe models to be used in gazebo, like the robot station, the bottles, the gripper etc. In this case, it is uncommon to have nodes in the package, instead, there are the following folders:

- **meshes**: where the 3D models of the object are stored. Typically, these are in the *.stl* or *.dae* format.
- **urdf**: where the URDF files that describe the model (with extension *.xacro*) are stored.

2.3.2 Structure of the developed application

Overview

The developed project consists of 11 packages where:

- six of these are just for describing the components of the system to be imported in the simulation (the packages containing the URDF files): camera, Robotiq gripper, robot base station, Universal Robots model (ur5e), final description file (containing all the previous description packages), conveyor belt.

- *model_spawner* package: used to spawn regularly the bottles on top of the conveyor belt. This package contains both the *urdf* folder with the *bottle.urdf* description file, and the python script *spawn_model.py* for spawning the objects.
- *moveit_calibration* package: used to calibrate the precise relationship between the camera and the robot's end effector.
- *moveit_config* package: used to store the parameters that MoveIt requires to ensure the movement of the arm and the gripper without collisions.
- *ur5e_gazebo* package: responsible for launching the gazebo simulator, including all the previously defined packages components to load the simulation.
- *ur5e_controller* package: this is the core part of the application. In this package there are the following essential nodes:
  - **object_detect_node**: responsible for performing the inference on the camera images returning the pixel positions of the detected objects.
  - **ur5e_controller_node**: the main part of the system. This node has to provide to the user a panel where the images of the robot's camera can be seen, the bottles are marked (as detected), and, by clicking on the desired one, the application predicts where it will be after x seconds and performs the pick & place routine.
  - **movement_action_server**: the node that acts as a server in the communication with the previously defined node and, by using MoveIt libraries, performs all the motions required to perform the pick & place routine.

Details of ur5e_controller

While the other packages are mainly used only to start the simulation environment, this package is responsible for performing all the operations that were described in the system overview section (2.1), therefore it will be explored deeply. As previously explained, this package is composed of three major components and their functionality is explained as follows:

- ***ur5e_controller_node*** has an infinite loop where at each iteration:
  1. Evaluates the perpendicular distance between the camera and the conveyor belt.
  2. Subscribes to the */ur5e/camera/image_raw* to retrieve the image of the robot camera at that precise moment.
  3. Saves the current time stamp (the one the image is referred to).

4. Acts as a client in the communication with *object_detect_node* using a service, passing to it the image captured and receiving back the array of pixel coordinates of the bottles detected in that image.
5. Using the camera parameters, converts the pixel coordinates into 3D coordinates with respect to a fixed frame (the base_link of the robot base station).
6. Uses the tracking algorithms, which will be explained in section 4.1, to assign the correct ID number to each detected bottle so that the same one, located in consecutive frames, keeps the same ID.
7. Creates and displays a new image, starting from the one retrieved before, where the bottles detected are marked with colored circles.
8. If the user clicks on one of those detected bottles:
    (a) Using the prediction algorithm (explained in section 4.2), it predicts where the bottle will be after x seconds (where x is a parameter of the program).
    (b) It acts as a client in the communication with *movement_action_server* (using actions), sending to that node the goal position to be reached before the time x (previously defined), the time stamp when the object is expected to be under the gripper, and the boolean variable *pick* set to true. The server node will provide feedback messages during the execution, in order to alert the main node if there are problems or not, and, if the motion performed correctly, will return a success boolean value.
    (c) It communicates (again) with *movement_action_server*, sending to that node the goal position to be reached and the boolean variable *pick* to false. This way the server node will know that it has to perform the place routine (and not the pick one), thus, the gripper will be opened only when the target position is reached.

- ***object_detect_node*** is a simple server node that loads the trained weights of the detection for the YOLO network, performs the inference on the image passed using service communication and returns in output the list of pixel coordinates of the detected objects. The structure of its **Request & Response** method is the following:

```
# Request
sensor_msgs/Image image  # The input image
---
# Response
geometry_msgs/PoseArray pixel_points # The array of pixel
    coordinates
```

- ***movement_action_server*** is the action server node that, using MoveIt libraries (discussed in section 5.1), performs the pick & place routines. It uses actions as a communication method with the following structure:

```
# Goal
string frame_id      # the reference frame for the
   following coordinates
float64 x            # target x-coordinate
float64 y            # target y-coordinate
float64 z            # target z-coordinate
bool pick            # true for pick motion, false for
   place motion
float64 max_duration    # the maximum time the motion can
    take to arrive to the goal position
float64 close_gripper_time  # the time stamp when to
   close the gripper
---
# Result
bool success    # true if the robot completed the routine
---
# Feedback
int32 progress   # progress code of the execution: 1:
   motion planned, 2: motion is executing, 3: target
   position reached, 4: gripper closed / opened, 5:
   moving to home position
```

The next chapters provide an in-depth explanation of how the ***ur5e_controller*** package components work.

Chapter 3

Object Detection and Pose Reconstruction

3.1 Object Detection

Object detection is the process that, given an image of a scene, outputs a list of pixel coordinates indicating where the target objects are located. Typically, object detection identifies the smallest bounding box that contains each target object, with the box defined by its four corner points. However, in our system, where the objects of interest are the bottle lips, it is sufficient to return the central point of the bounding box as the detection result. By definition, object detection also involves classifying each bounding box with a label corresponding to the identified object. However, this aspect is not relevant for our application since there is no need to distinguish between different types of objects (our only targets are the bottles).

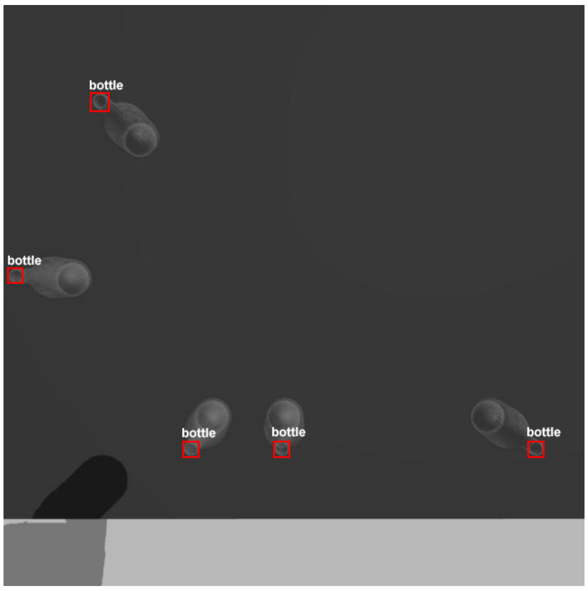


Figure 3.1: General usage of Object Detection

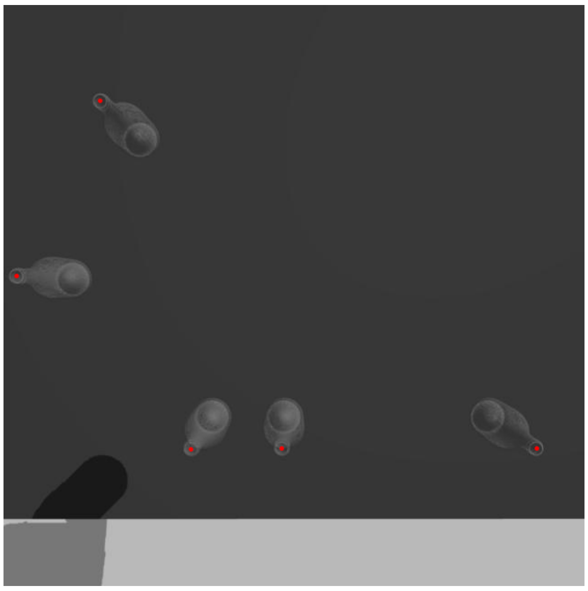


Figure 3.2: Custom usage of Object Detection

3.1.1 The choice of YOLO approach

YOLO (You Only Look Once) is an open-source algorithm that revolutionized the way objects are detected in images and videos by using a single neural network to predict bounding boxes and class probabilities simultaneously. It was first introduced in 2016 and gained popularity due to its speed and accuracy, enabling object detection in real-time applications like the current one.

Since this application was developed in a simulation environment, the images provided by the virtual camera are relatively simple, repetitive, and not very realistic. It would have been easy to create an algorithm optimized for these conditions using standard machine vision techniques. However, this approach would not have been successful in practice, as the algorithm would only work in the simulated environment. When applied to a real system with entirely different and more complex images, modifying the solution would be extremely difficult, if not impossible.

A neural network system, like YOLO, must be trained on images captured by the robot camera. In our case, these are still images with poor realism due to the simulation environment. As a result, when transitioning to a real-world setup, the neural network will likely fail to detect objects accurately. However, this issue can be resolved by retraining the network using a sufficient number of images from the actual environment. This approach enhances configurability, since the core algorithm remains unchanged and only needs to be adjusted for the new conditions, thus requiring minimal effort.

These considerations were the key reasons behind the decision to implement the YOLO algorithm (a neural network) in the system rather than relying on a traditional machine vision approach.

3.1.2 Training the network

Training YOLO involves feeding the neural network with a large dataset of labeled images, where each image contains objects of interest with corresponding bounding boxes and class labels. The network uses this data to learn to predict the location and class of objects in new, unseen images.

This process can take some time as all the following steps need to be implemented with the right amount of precision:

Acquiring the images

The first step involves obtaining a huge number of images from the robot camera of the scene viewed. In this case the camera was pointing to the conveyor belt where the bottles were sliding.

To acquire images, a simple ROS node was created that continuously reads from the topic where the image data is published (in this case */ur5e/camera/image_raw*). Upon user input, the node captures the current image and saves it to a designated folder.

To achieve higher precision from the neural network, it is important to follow these tips when acquiring images:

- Capture images from different angles.
- Vary the lighting conditions.

- Change the distance between the camera and the plane.
- Use the same camera settings that will be applied during actual usage.

Labeling the images

To train the neural network, each image must be accompanied by a text file containing the bounding boxes of the target objects and their class ID. For the YOLOv8 model, this information follows a specific format.
Each line in the label file corresponds to one object in the image and follows this structure:

```
<class_id> <x_center> <y_center> <width> <height>
```

Where:

- `<class_id>`: Integer representing the class of the object.
- `<x_center>`: X coordinate of the center of the bounding box (normalized between 0 and 1).
- `<y_center>`: Y coordinate of the center of the bounding box (normalized between 0 and 1).
- `<width>`: Width of the bounding box (normalized between 0 and 1).
- `<height>`: Height of the bounding box (normalized between 0 and 1).

Note that all coordinates and dimensions are normalized relative to the image size.
Obtaining all these .txt files manually can be very time-consuming, but there are tools that make the process easier. The one used for this project was Roboflow: a web-based platform (or web application) that provides tools for image labeling, data augmentation, and exporting in various formats for training machine learning models (including YOLOv8). It runs entirely online, allowing users to upload, process and download datasets directly through its website.

In the end, the user will obtain two separate folders:

- **images**: containing all the images captured in the previous step, each with a unique name.
- **labels**: containing all the text files representing the annotations, formatted according to the specified structure.

Performing the training

Before proceeding in the training it is necessary to create the *data.yaml* file: a configuration file used in YOLOv8 to define the structure and paths of the dataset that the model will be trained on. It provides essential information such as class names, dataset paths, and dataset splits (train/validation/test).

In order to train the YOLO neural network, it is only necessary to run the script provided by Ultralytics dedicated for this purpose. The command has the following structure:

```
yolo train model=<model> data=<data> epochs=<epochs> imgsz=<imgsz>
```

Where

- `<model>` specifies the model architecture. YOLOv8 comes with various pre-trained weights such as yolov8n.pt (Nano), yolov8s.pt (Small), yolov8m.pt (Medium) etc. Larger models tend to have more layers and parameters, which allows them to learn more complex patterns but increases their resource requirements.
- `<data>` is the path to the dataset configuration file defined above.
- `<epochs>` is the number of training cycles. A higher number of epochs typically leads to a more accurate network on the training data, but it also increases the time required for training.
- `<imgsz>` is the image size used for training. YOLOv8 commonly uses 640x640.

When launching the command, the training process can take a long time depending on:

- the number of training images
- the size of the images
- the performance of the machine
- the number of epochs.

A factor that significantly impacts performance, and thus the time required for training, is whether the machine has a GPU or not. GPUs are designed for parallel processing, allowing them to handle the large-scale matrix operations and computations required for training neural networks much more efficiently than CPUs.

3.1.3 Performing inferences

Training a YOLO neural network results in a PyTorch file, which contains the weights of the trained model and can be accessed using Python modules. This file can be loaded into a ROS service node that acts as a server, receiving an input image for inference and returning a list of points (normalized to the image size) as output.

Again, the performance of inferences is significantly influenced by the presence of a GPU. In the case of the machine used, equipped with an Nvidia GTX 1060 6GB GPU, each image was processed in approximately 130 ms.

Performance analysis

Having measured the inference time on more than 100 consecutive images, these were the result in term of milliseconds:

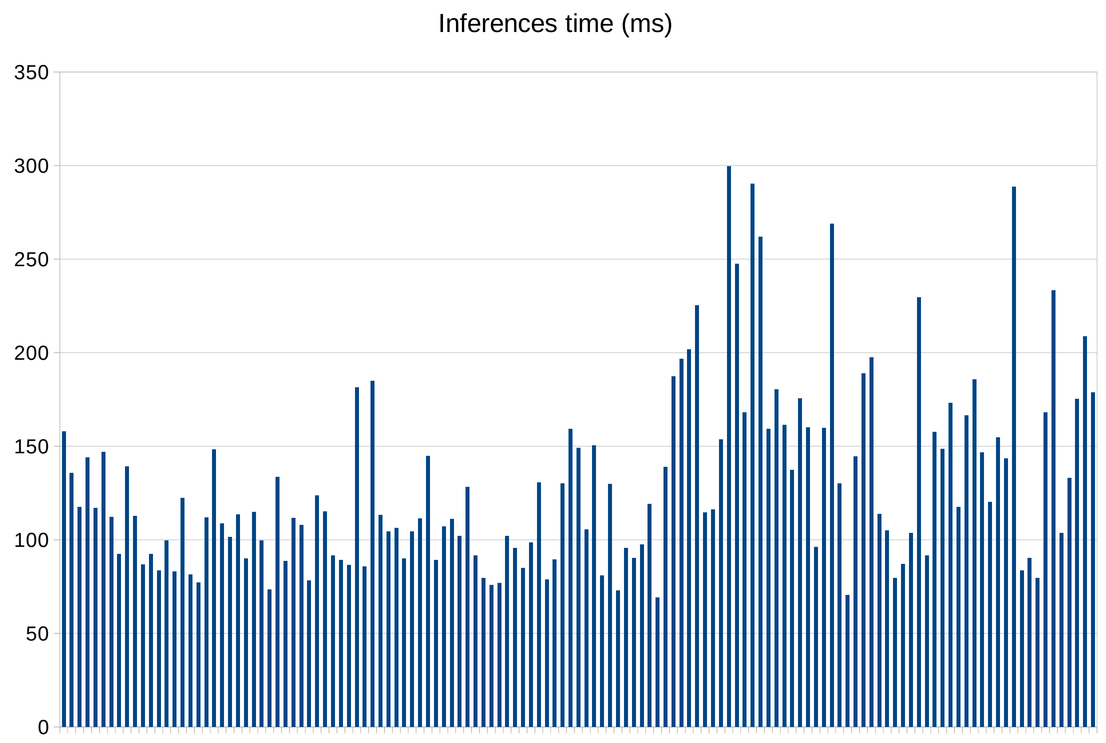


Figure 3.3: Time measures of the inference of 100 images

- mean value of inference time: **130ms**.
- minimum value of inference time: **69ms**.
- maximum value of inference time: **300ms**.

3.2 Camera Calibration

Before diving into what camera calibration is, it is important to understand how a simple pinhole camera is modeled and thus how 3D points are projected into 2D pixels.

Pinhole Camera Model

A pinhole camera is a simple camera with a small aperture where light rays can pass through and project an inverted image on the opposite side of the camera as shown in Figure 3.4. In order to understand the math behind this projection, it is helpful to work on the virtual image plane, placed at focal length distance from the focal point.

Figure 3.5 shows the side view of the pinhole camera scheme, which can be used to derive the transformation between a point P in the 3D space and its projection p onto the virtual image plane. The scheme in 3.5 shows the relation between y_p and Y_P, but the same reasoning is valid

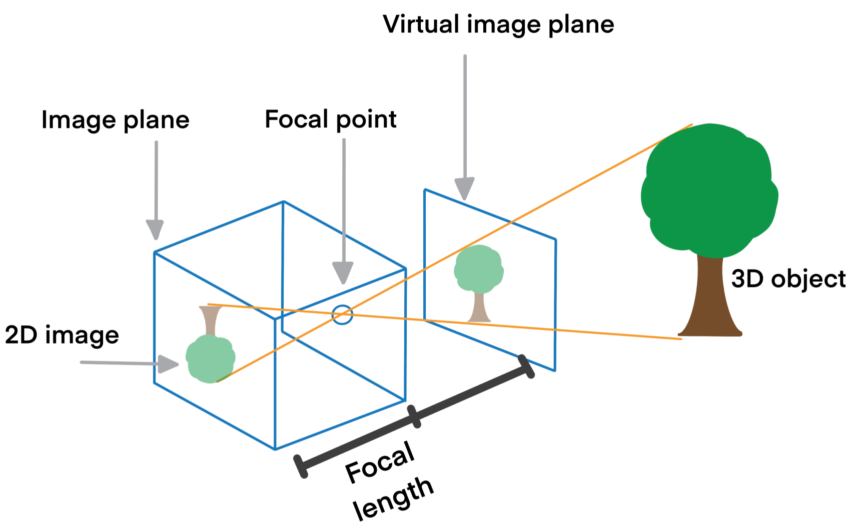


Figure 3.4: Pinhole camera scheme

also for x_p and X_P, therefore:

$$\frac{Y_P}{y_p} = \frac{Z_P}{f}, \qquad \frac{X_P}{x_p} = \frac{Z_P}{f} \tag{3.1}$$

$$y_p = \frac{Y_P \cdot f}{Z_P}, \qquad x_p = \frac{X_P \cdot f}{Z_P} \tag{3.2}$$

Once the coordinates in the image plane are obtained, they can be converted into pixel coordinates. The parameters needed are: the pixel width (w) and height (h) and the coordinates of the principal point: (c_x, c_y). The origin of the pixel coordinates is at the top left corner, thus the resulting equations are:

$$u = c_x + \frac{x_p}{w} = c_x + \frac{f}{w} \cdot \frac{X_P}{Z_P} = c_x + f_x \cdot \frac{X_P}{Z_P} \tag{3.3}$$

$$v = c_y + \frac{y_p}{w} = c_y + \frac{f}{h} \cdot \frac{Y_P}{Z_P} = c_y + f_y \cdot \frac{Y_P}{Z_P} \tag{3.4}$$

In the end the whole projection can be expressed as:

$$Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.5}$$

Since then, it was assumed that the point $P = (X_P, Y_P, Z_P)$ was written with respect to the camera frame (with the origin in the focal point), but if it were given with respect to a different frame, then it would have been necessary to apply to P a rototranslation using the following

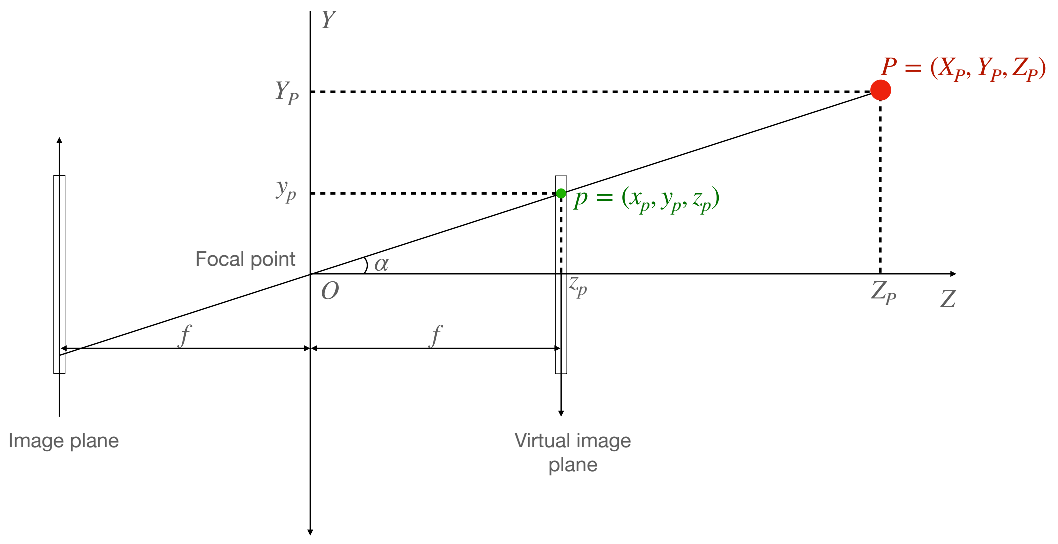


Figure 3.5: Pinhole scheme side view

matrix:

$$\begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{3.6}$$

In the end, the mapping between the 3D world scene and the image pixel coordinates is given by the following equation (composed of both the intrinsic and extrinsic parameters).

$$Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.7}$$

Thus:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \frac{1}{Z} \begin{bmatrix} f_x & 0 & c_x & 0 \\ 0 & f_y & c_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} R & t \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} \tag{3.8}$$

The points w.r.t. (with respect to) the world frame (or another object) are transformed into coordinates w.r.t. the camera frame using the extrinsic parameters. Then, the camera coordinates are mapped into the image plane using the intrinsic parameters.

Real model

In reality, cameras are not just pinhole cameras, they mount lenses to acquire more light. This can cause some unwanted lens distortion effects such as:

- Radial distortion where, due to the lens shape, straight lines appear curved, especially near the edges. They are mainly classified into:
  - barrel distortion
  - pincushion distortion.

  Radial distortion is often corrected considering the polynomial approximation:

  $$x_{\text{corrected}} = x \cdot \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \tag{3.9}$$

  $$y_{\text{corrected}} = y \cdot \left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) \tag{3.10}$$

  where r is the radial distance from the image center.

- Tangential distortion where, due to the lens not perfectly aligned with the image sensor, the image appears skewed or shifted. The tangential distortion correction is applied as follows:

  $$x_{\text{corrected}} = x + \left[2p_1 xy + p_2(r^2 + 2x^2)\right] \tag{3.11}$$

  $$y_{\text{corrected}} = y + \left[p_1(r^2 + 2y^2) + 2p_2 xy\right] \tag{3.12}$$

All the necessary distortion coefficients $(k_1, k_2, k_3, p_1, p_2)$ are evaluated through the camera calibration process.

3.2.1 What is Camera Calibration?

Camera calibration is referred to as the process of finding both internal and external parameters of the camera. More precisely:

- **internal parameters**, typically composed of:
  - **intrinsic parameters**: focal length (f), pixel width (w), pixel height (h), principal point coordinates (c_x, c_y).
  - **distortion coefficients**: (k_1, k_2, k_3) for radial distortion; (p_1, p_2) for tangential distortion.

  Internal parameters help to understand how the camera captures the real world and translates it into a digital image.

- **external parameters**, also called **extrinsic parameters**, describe the camera's position and orientation (its "pose") relative to the external world or an object. These extrinsic parameters are essential for mapping the 3D world to the camera's coordinate system, allowing for spatial understanding.

3.2.2 Finding Internal parameters

In order to obtain the internal parameters, the most used approach involves capturing multiple images of a known calibration pattern, such as a checkerboard, from different angles and positions. The steps required are:

1. **Produce a calibration pattern**. A common approach is to print a checkerboard pattern with known dimensions. A useful tool for this is the website calib.io, which allows users to generate a calibration pattern while also specifying the final dimensions.

2. **Capture multiple images**. Take several images (from 10 to 20) of the calibration pattern from different viewpoints. Move the camera to capture the pattern at different angles and covering various parts of the image.

3. **Detect the Pattern in Each Image**. Using computer vision functions in OpenCV (such as *cv::findChessboardCorners*), detect the points of interest on the pattern (the corners of the checkerboard). These points will be used as reference locations for the calibration process.

4. **Estimate Internal Parameters**. OpenCV provides libraries and methods that, given the previously obtained information, calculate the camera internal parameters. All the code that is needed to perform the calibration is well explained in the OpenCV camera calibration tutorial web page.

3.2.3 Finding External parameters

Determining external parameters is slightly more complex than finding internal ones, as it depends on the specific application. In this scenario, where the camera is mounted on the robot gripper (eye-in-hand configuration), the external parameters are used to define the camera pose relative to the gripper reference frame. This process is known as **eye-in-hand calibration**, and its step are:

1. **Produce a calibration pattern**. Again, like when finding internal parameters, a calibration pattern is required. This time, a better approach is to print a *ChArUco board* pattern. ChArUco is the combination of a chessboard and an ArUco board:

    - ArUco markers and boards are very useful due to their fast detection and their versatility. However, the accuracy of their corner positions is not too high.
    - Chessboard corners can be identified more accurately. However, using a chessboard pattern is not as versatile as using an ArUco board: it has to be completely visible and occlusions are not permitted.

ChArUco boards try to combine the benefits of these two approaches.

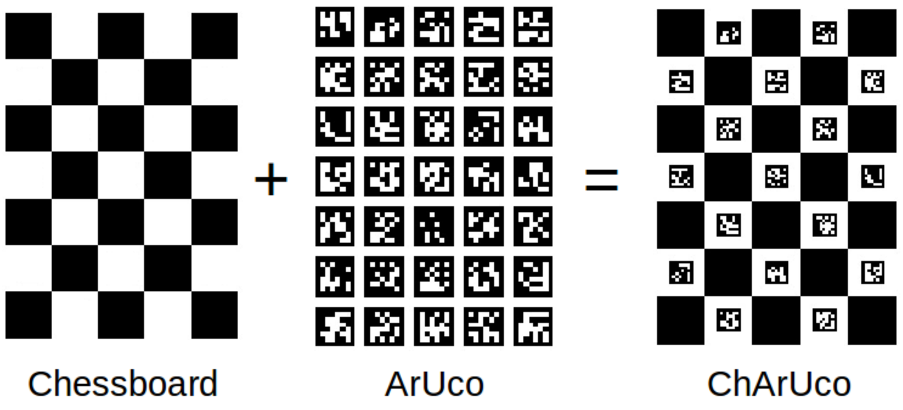


Figure 3.6: ChArUco board

2. **Capture multiple images and correlate them with poses**. With the calibration pattern fixed to the work environment, which ensures that its pose relative to the robot base is constant during the calibration process, the next step is to acquire multiple images of the target and, for each one, associate the end-effector pose (relative to the base frame) to that image.

3. **Find the camera pose relative to the end-effector frame**. The calibration object (of known geometry) can be detected from the camera image and, knowing the internal parameters, it is possible to obtain its pose relative to the camera (H_{OBJ}^{CAM}). The goal is to find the transformation between the camera and the end-effector (H_{CAM}^{EE}); to do that, it is necessary to close the chain of transformations. The transformation between the end-effector and the base frame (H_{EE}^{ROB}) is given (knowing the kinematics of the robot). The only remaining transformation is the one between the target and the base frame (H_{OBJ}^{ROB}) which is known to be constant.

   By acquiring enough images (10 to 20), it is possible to solve the optimization problem with high accuracy to obtain H_{CAM}^{EE}. OpenCV provides the function *calibrateHandEye()* that can be used for this purpose, but for the application, the ROS package *moveit_calibration* has been used, which receives the following basic information as input:

   - camera image topic name (where the image data flows)
   - camera parameters topic (where the internal properties of the camera flows)
   - calibration target properties (size, ArUco dictionary, etc.)
   - end-effector frame
   - base frame
   - list of robot poses (desired poses where to move the end-effector where the target is visible)

   and automates the process of:

(a) moving the robot to the desired position

(b) acquiring the image and all the necessary poses

(c) solving the optimization problem.

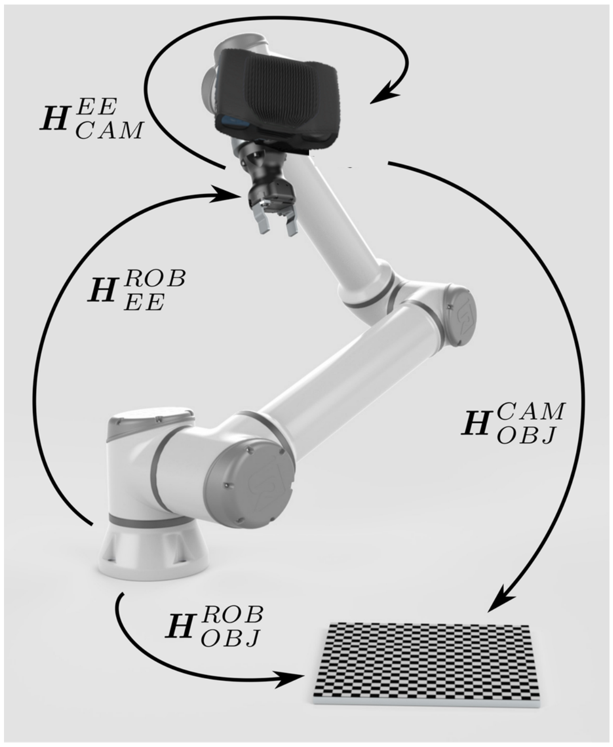

$\boldsymbol{H}_{CAM}^{EE}$
$\boldsymbol{H}_{EE}^{ROB}$
$\boldsymbol{H}_{OBJ}^{CAM}$
$\boldsymbol{H}_{OBJ}^{ROB}$

Figure 3.7: Scheme of the homogeneous transformations.

3.3 Pose Reconstruction

The goal of pose reconstruction is to determine the 3D positions of objects detected with respect to the robot's base frame. Pixel coordinates and camera calibration parameters alone are not enough to evaluate the final pose. In a single-camera system, depth information is lost because the camera captures only a 2D projection of the scene. As a result, given only the pixel coordinates (u,v), there are infinitely many possible 3D positions (X,Y,Z) that could lie along the same line extending from the camera's focal point through the image plane as shown in Figure 3.8.

The solution to this problem is to evaluate the perpendicular distance between the camera and the objects observed in the vertical axis. During all the detection process, it is assumed that the camera points perfectly perpendicular to the conveyor belt where the bottles slide and that the heights of the bottles, the conveyor belt and the platform where the robot is fixed are known. Assuming that, finding the depth becomes straightforward. It is sufficient to find the height of the camera with respect to the robot base frame (h_{camera}^{robot}), using the extrinsic parameters (evaluated during camera calibration). Then the height of the bottles with respect to the robot base frame is: $h_{bottle}^{robot} = (h_{bottle} + h_{conveyor} - h_{robot})$. Finally the depth can be calculated as: $Z_C = h_{camera}^{robot} - h_{bottle}^{robot}$.

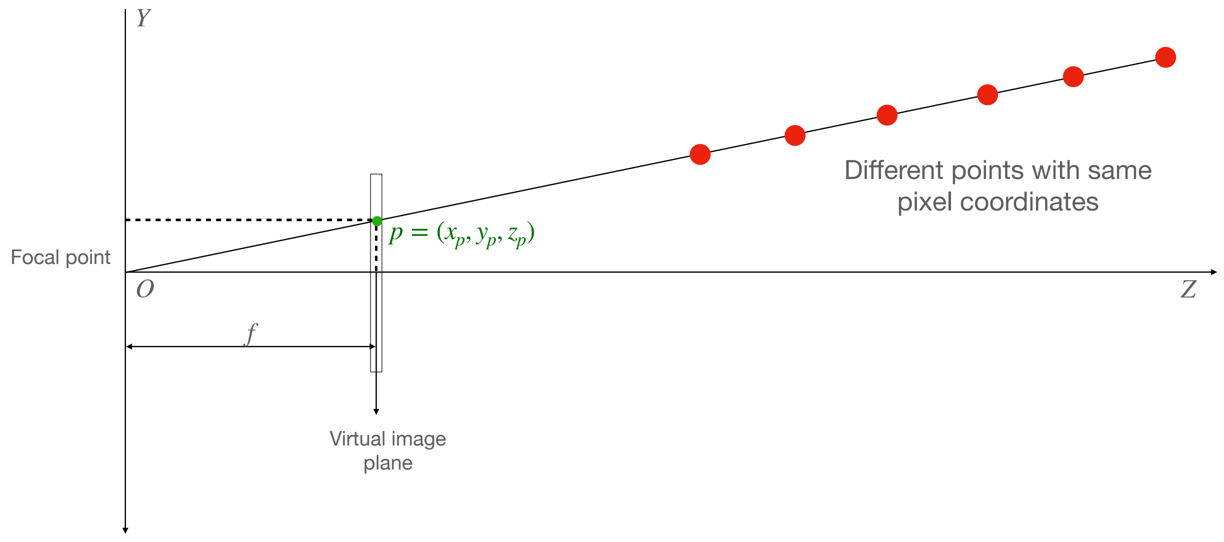


Figure 3.8: Representation of loss of depth information

Now there is all the necessary information to find the position of a detected bottle w.r.t. the robot base frame, knowing its pixel coordinates in the captured image. The process is obtained with the following steps:

1. **Pixel coordinates to 3D position w.r.t camera**. This transformation is done using the internal parameters of the camera and its vertical distance with the object (Z_C) as follows:

$$\begin{bmatrix} u \\ v \end{bmatrix} := \text{pixel coordinates} \qquad \begin{bmatrix} X_C \\ Y_C \\ Z_C \end{bmatrix} := \text{position w.r.t. camera frame} \tag{3.13}$$

$$Z_C = h^{robot}_{camera} - h^{robot}_{bottle} \quad \text{(already calculated)} \tag{3.14}$$

$$X_C = x_{undistorted} * Z_C \tag{3.15}$$

$$Y_C = y_{undistorted} * Z_C \tag{3.16}$$

Where:

$$x_{undistorted} = x_{distorted}\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) + 2 p_1 x_{distorted} y_{distorted} + p_2\left(r^2 + 2x^2_{distorted}\right) \tag{3.17}$$

$$y_{undistorted} = y_{distorted}\left(1 + k_1 r^2 + k_2 r^4 + k_3 r^6\right) + 2 p_2 x_{distorted} y_{distorted} + p_1\left(r^2 + 2y^2_{distorted}\right) \tag{3.18}$$

$$r^2 = x^2_{distorted} + y^2_{distorted} \tag{3.19}$$

$$x_{distorted} = \frac{u - c_x}{f_x} \qquad y_{distorted} = \frac{u - c_y}{f_y} \tag{3.20}$$

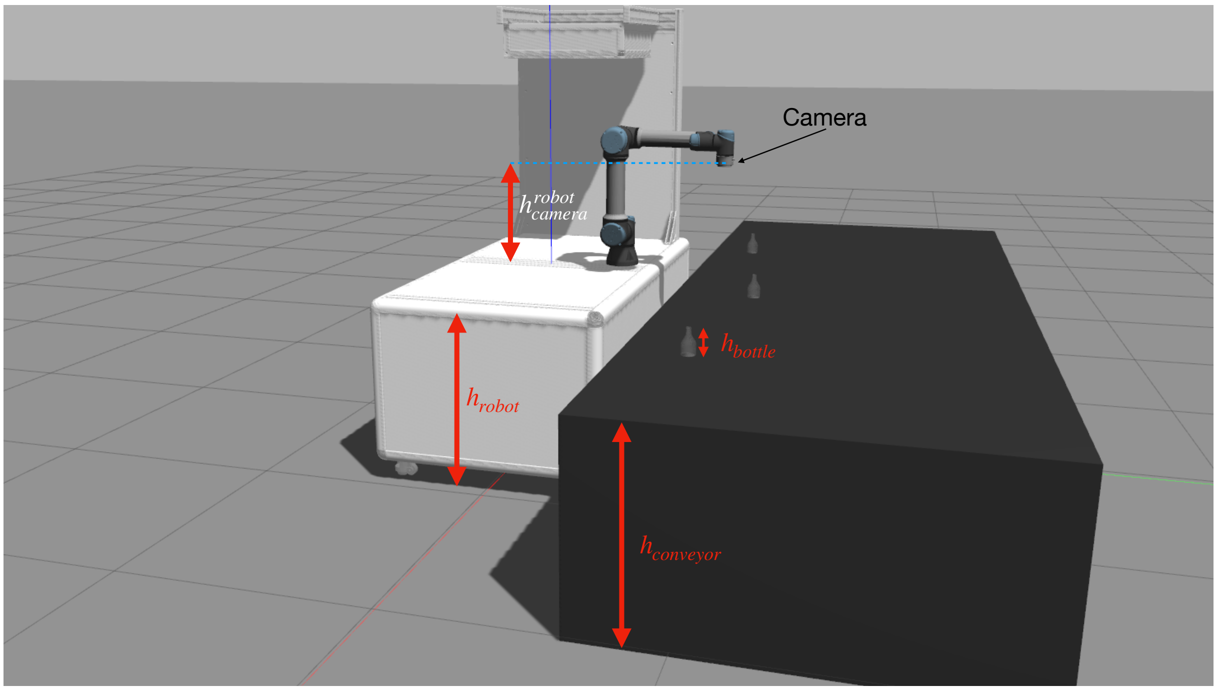


Figure 3.9: Heights scheme

2. **3D position w.r.t. the camera to the robot base frame**. Once the transformation matrix H_{CAM}^{EE} is obtained from the extrinsic calibration, and given the transformation matrix H_{EE}^{ROB}, the overall transformation $H_C^R = H_{EE}^{ROB} H_{CAM}^{EE}$ can be computed. This matrix can then be used to convert coordinates from the camera reference frame to the robot base frame.

$$P_R = H_C^R \cdot P_C \tag{3.21}$$

$$\begin{bmatrix} X_R \\ Y_R \\ Z_R \\ 1 \end{bmatrix} = \begin{bmatrix} R_{3x3} & t_{3x1} \\ 0 & 1 \end{bmatrix} \begin{bmatrix} X_C \\ Y_C \\ Z_C \\ 1 \end{bmatrix} \tag{3.22}$$

Chapter 4

Object Tracking and Pose Prediction

4.1 Object Tracking

The system is now capable of analyzing camera images, detecting objects (identifying the pixel locations of bottles within the images), and translating those pixel coordinates into 3D poses using the camera parameters. However, for each image, the application generates an independent list of poses, with no correlation with those from previous frames. As a result, it is not possible to track the movement of the bottles as they slide, making it difficult to predict the position of a specific bottle at a future time t.

The solution to this problem is to design a system capable of identifying each pose (with an ID number) in a way that the same bottle, captured in different consecutive poses, keeps the same ID. The system will maintain the history of all poses for each ID (thus for each unique bottle).

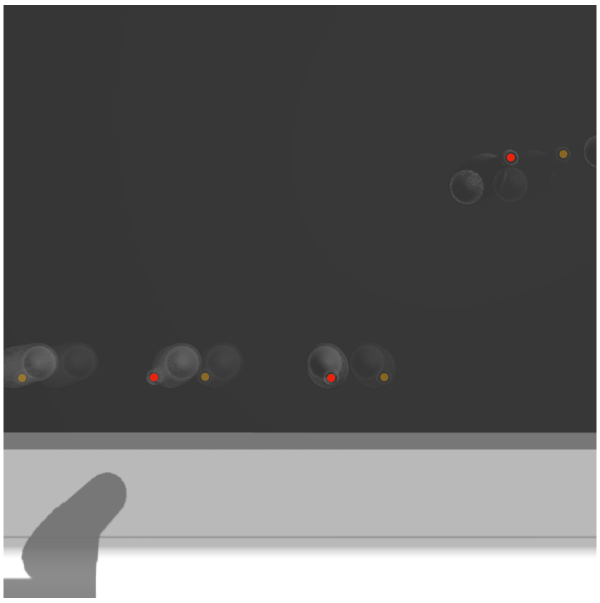


Figure 4.1: Without object tracking

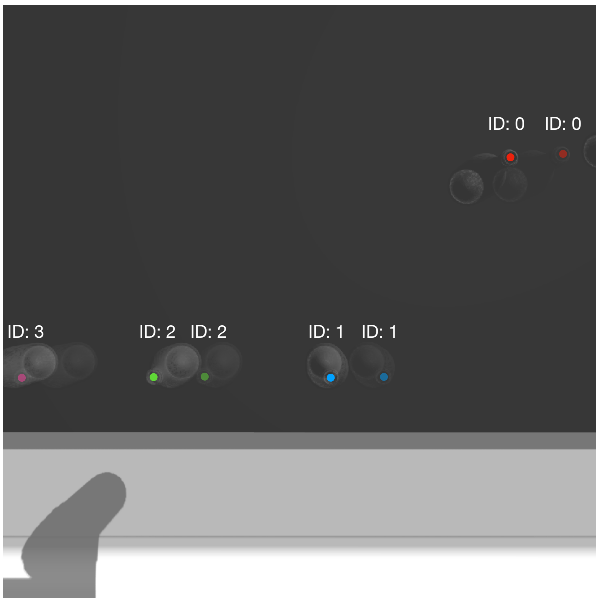


Figure 4.2: With object tracking

4.1.1 General Overview

At each frame, the system must determine whether to assign the position of each bottle to an existing ID or generate a new one. As the bottles move in one direction, any bottle that enters the camera view area is assigned a new ID, and its subsequent positions are tracked and matched to that ID.

The algorithm that tracks the new detected objects takes as input a list of 3D positions representing the bottles and outputs for each position an ID number that represents the corresponding object. This algorithm can be described with the following steps:

- **If** no objects have been tracked yet, create a new object for each input position, assigning incrementing ID numbers, and save them.
- **Else**
  1. For each new detected position, find its closest saved object using the Euclidean distance.
  2. If multiple detected objects are associated with the same saved object, keep only the closest pair.
  3. For each unmatched detected object, create a new object with a unique ID number and save it to begin tracking.

4.1.2 Details of the Tracking System

In details, the object tracking system is composed of two classes:

- **Object**: the class that represents a single bottle (or a generic object), composed of these members:

  ```
  int id;
  std::vector<std::tuple<std::tuple<float, float, float>,
    int>> history_poses;
  ```

  where `id` is the unique integer identifier and `history_poses` the vector of past positions of that object. A single history position is expressed with the tuple `<position, time_stamp>`. This class facilitates the modularity of the application, making easier its developing.

- **Tracking**: the class responsible for storing and maintaining the vector of objects (of type `Object`) making possible to keep track of the detected bottles in input. The members of this class are the following:

  ```
  std::vector<Object> objects;
  int last_time_stamp = 0;
  int last_id = 0;
  ```

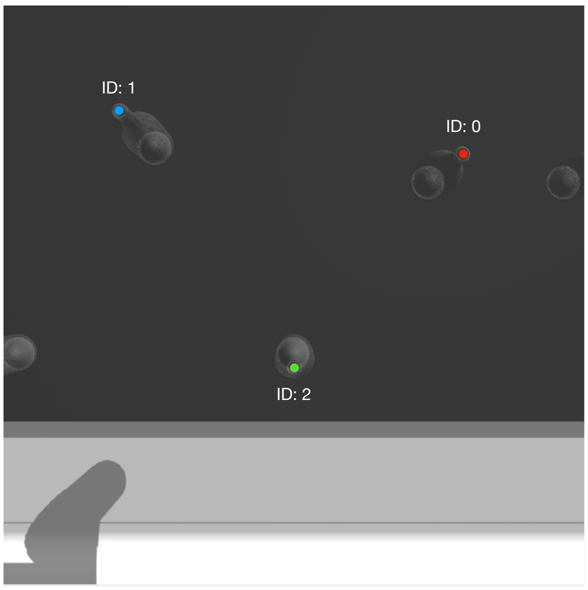


Figure 4.3: Frame n captured by the camera (objects already identified).

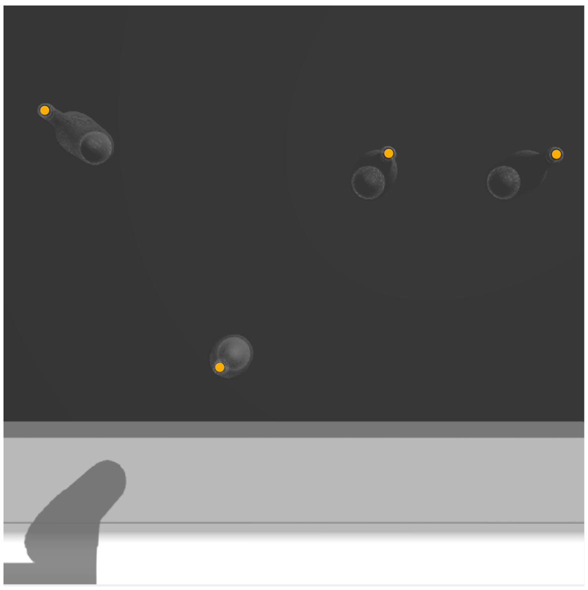


Figure 4.4: Frame $n+1$ captured by the camera (objects have to be identified).

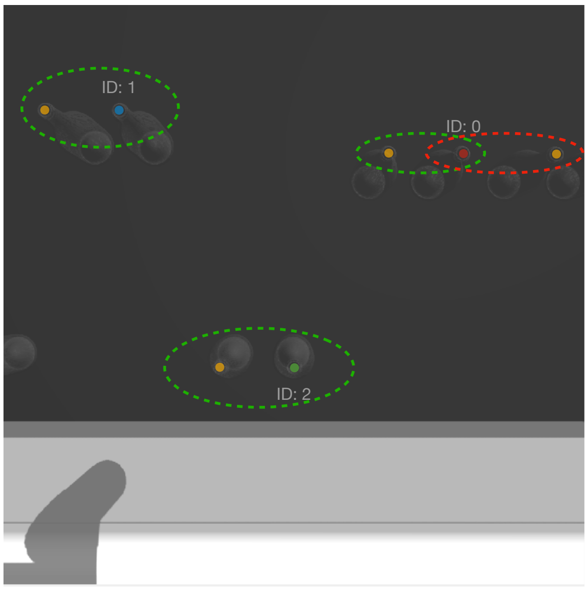


Figure 4.5: Closest pair matching. Green: closest match from both sides; Red: closest match only from one side.

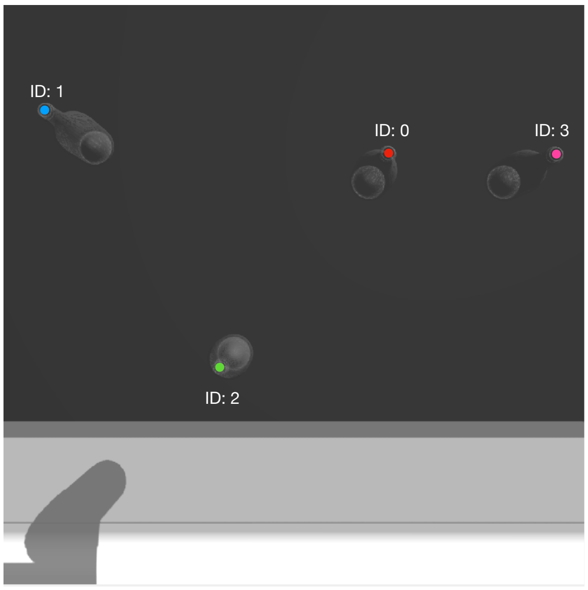


Figure 4.6: Final Matching. The new detections whose closest matches were not from both sides became a new object.

where:

- `objects` is the vector used to store the bottles information of `Object` type.
- `last_time_stamp` is the integer representing the time stamp of the last object inserted into `objects`.
- `last_id` is the integer value that increases every time a new object is tracked: it is used to assign a unique id to the objects.

Tracking class implements the following method:

```
void trackDetections(std::vector<Object> &detections);
```

which is the one used to solve the tracking problem.

At the beginning of the program a `Tracking` variable is declared in order to initialize the tracking class (with empty members) and every time a frame of the camera is analyzed and a list of positions is obtained, the program prepares the `detections` vector to be passed to the `trackDetections` function. `detections` is a variable of type `vector<Object>`, thus, as the program has not already identified the new positions, the IDs of all its members are initialized to -1.

trackDetections function operates as follows:

1. After verifying if the input parameter is valid, it updates the member variable `last_time_stamp` with the time stamp obtained from an element in the `detections` vector (it is assumed that all elements in this vector have the same time stamp as the detections are obtained from the same image).
2. If the tracking variable is empty (if there are no objects saved in its `objects` member variable), then for each element in detections (the input vector):
   (a) set its id to `last_id`
   (b) increment `last_id`
   (c) push this detection of type `Object` into the `objects` member variable.
3. Create an empty map of type: `std::map<int, std::tuple<int, float>>` that will be used to store the matches *(saved object - new detected object - distance between them)*
4. Scan all new detections (input) and, for each one, compare it with all saved objects (stored in the member vector) to find the closest pair in terms of Euclidean distance. It is important to consider the time difference between these two items: a new detection may have zero Euclidean distance from the last known position of an older object, but that does not necessarily mean they represent the same bottle.

   If the older object was previously matched with a closer detected object, replace the match with the new detection. Otherwise just add the new match to the map.

5. Update the `history_poses` of the already saved objects adding the pose of their match and the corresponding time stamp.

6. Add all the input detections that were not assigned in the mapping as new objects.

4.1.3 Limitations of this system

The tracking system described above works as soon as the whole analysis process doesn't require too much time. This process consists of:

1. Object detection

2. 3D reconstruction

3. Object tracking.

3D reconstruction and object tracking do not require much time compared to **object detection**, making it the bottleneck of the system. Even cheaper cameras can still achieve a frame rate of 30 FPS (frames per second), but object detection, which in this hardware configuration takes around 130ms, limits the analysis to $1/0.130 = 7.7$ images per second.

With a given frame rate for the analysis, there are three parameters that can be adjusted in order to have a correct tracking system:

- conveyor belt speed (m/s)

- density of the bottles (minimum distance between two bottles in the conveyor)

- field of view of the camera.

Consider the case of being able to track 7 images per second, obviously the bottle has to be visible from the camera which (is assumed) is able to capture images of a rectangle portion of the conveyor belt of $(1x1)$ meters. Considering for simplicity that the trajectory of the bottles is horizontal, clearly the speed of the bottle cannot be above $7m/s$, otherwise it may not be captured by the camera.

It is less obvious to understand how the distance between bottles (the density) has an impact on the correct operation of the tracking system. This system assigns to each new detected object the closest one found in the history of previous poses, this works as long as the gap between the observed frame and its subsequent one is low, like in Figure 4.5. When the gap increases it may happen that some bottles are incorrectly assigned, as shown in Figure 4.9.

Bottle density and conveyor speed are correlated by the following equations that depend on the frames per second that can be analyzed:

$$\frac{1}{FPS} \cdot \text{speed} = \text{minimum detectable distance} \tag{4.1}$$

which means:

$$\frac{1}{FPS} \cdot \text{speed} = \text{max density} \tag{4.2}$$

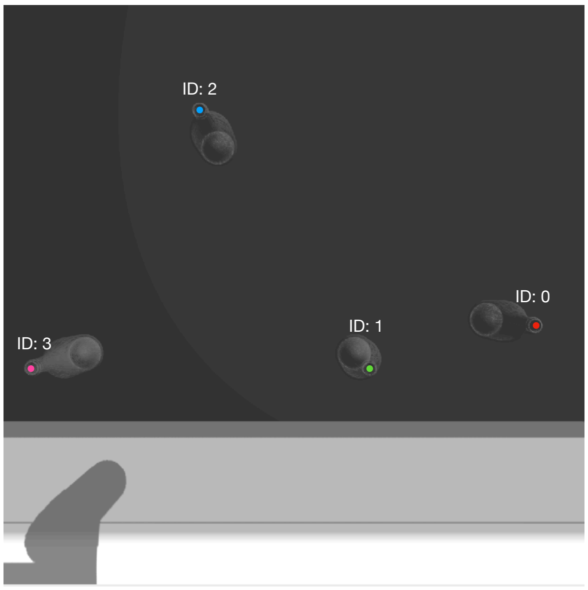


Figure 4.7: Frame n analyzed.

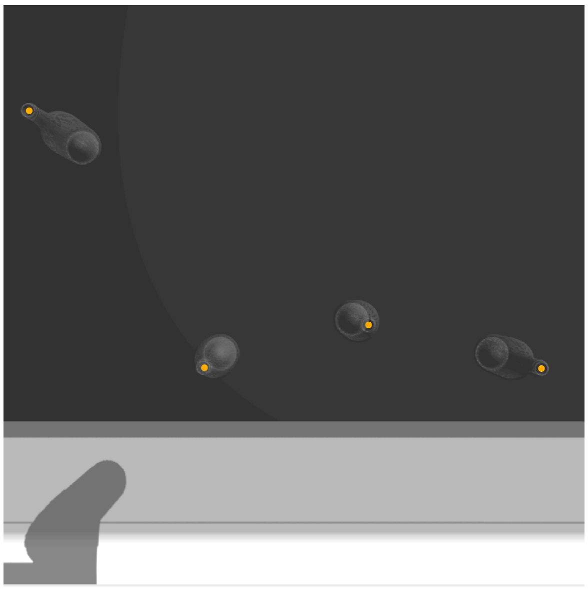


Figure 4.8: Frame $n+1$ analyzed where the gap between the previous frame is too high.

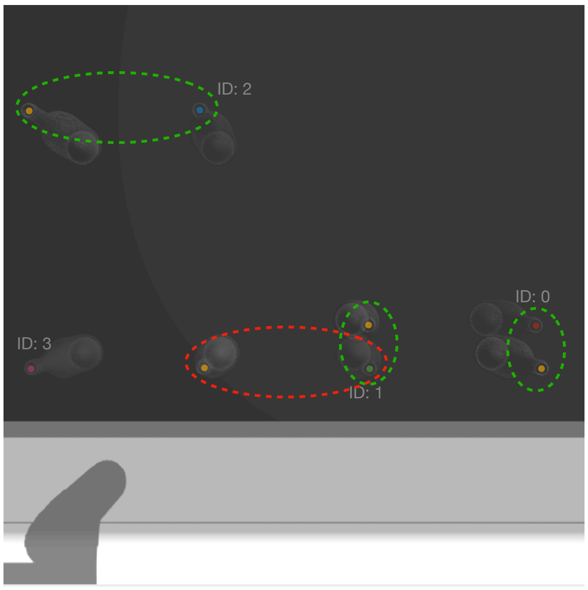


Figure 4.9: Wrong pair matching

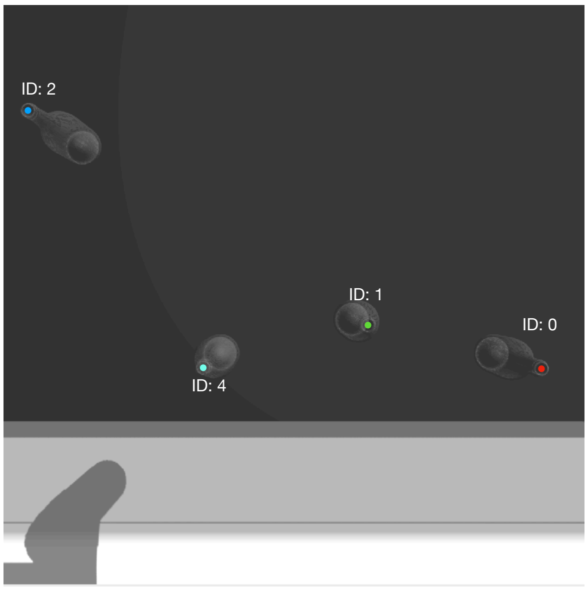


Figure 4.10: Wrong assigned ID's.

At this point, two equations can be expressed:

- Given the density of the bottles (their minimum distance), then the maximum speed of the conveyor belt will be:

$$\text{max speed} = FPS \cdot \text{max density} \quad \left[\frac{m}{s}\right] \tag{4.3}$$

- Given the speed of the conveyor, the maximum density of the bottles can be:

$$\text{max density} = \frac{\text{max speed}}{FPS} \quad [m] \tag{4.4}$$

Figure 4.11 represents a graph of the maximum speed / density that can be achieved at different analysis speeds (FPS).

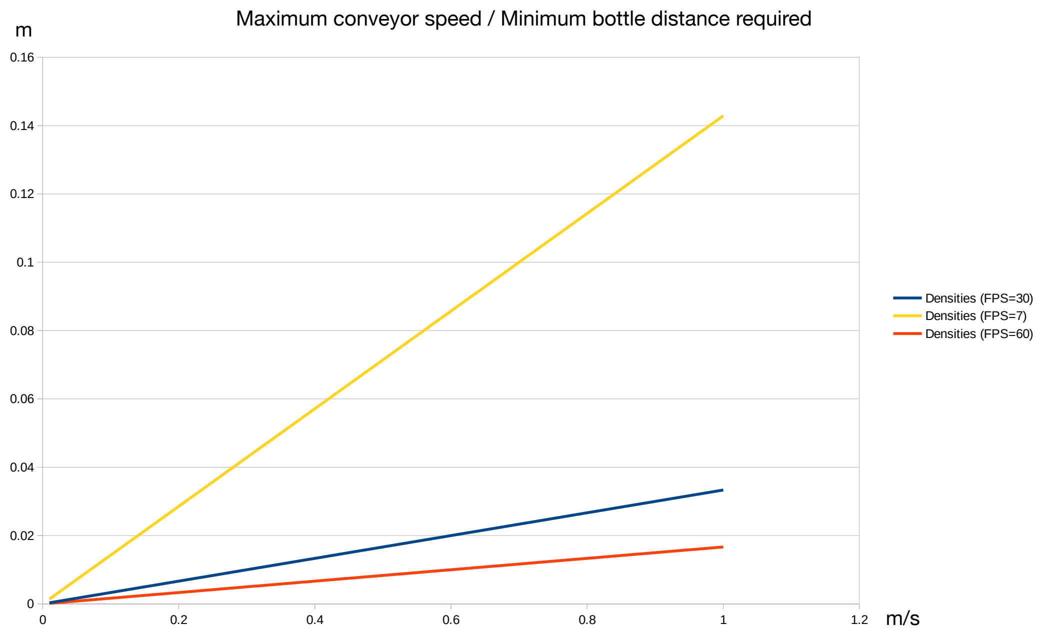


Figure 4.11: Plot of different maximum speed/minimum distance ratios based on the speed of the analysis (FPS)

In order to obtain a reliable system, it is important to select the density of the bottles and the conveyor speed in a way that keeps some tolerance in the tracking system. Otherwise, as soon as the computation time required by the object detection process takes more than expected, the tracking system will fail. In this application the maximum time measured for object detection was: 300ms, so considering $FPS = 2$ for deciding the values of speed and density will guarantee that the system will not fail easily. With $FPS = 2$, $speed = 0.1m/s$, then $max\_density = 0.05m = 5cm$.

4.2 Pose Prediction

At this point the system is capable of storing the poses with relative time stamps of all the objects that passed in the camera view. The goal of the whole application was to use the robotic arm to pick one of the bottles that are sliding on the conveyor belt. In order to do that, the system has to know in advance where the bottle will be after a period of time before starting the movement, otherwise it would not be able to reach the object in time.

Pose prediction is the task in charge of solving this problem: it receives as input the ID of the bottle of interest and a time x, analyzes the stored object poses to understand the motion path, and produces in output the time x' and the guessed position of the object at time $x' \approx x$.

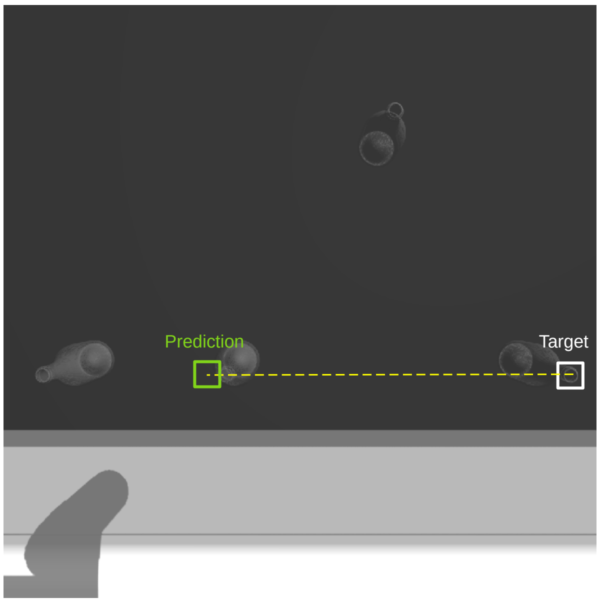


Figure 4.12: The system predicts where the bottles will be at time x.

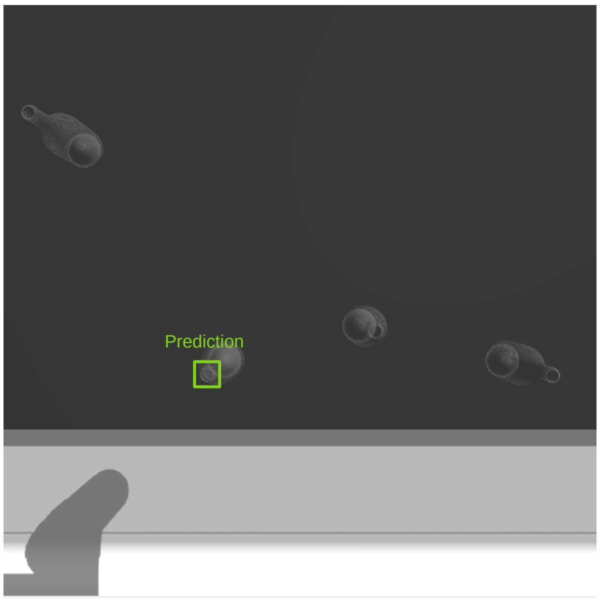


Figure 4.13: Image captured at time x, the target bottle actually passed through the predicted position.

4.2.1 How it works

The pose prediction process is based on using the trajectories of the previously observed objects to estimate the path of the target one. The whole process is described as follows:

1. Once the target bottle is selected (by its ID), find its last pose saved and the corresponding time stamp.

2. Select from the saved bottles the one which matches these conditions:

   - Its last pose saved has not a time stamp that is much distant from the target one, otherwise it may represent a bottle whose trajectory is not the same as the current one.

- In its history of poses, there should be one close to the target pose of the object. In this way the bottle used to make prediction is known to have been passed near the new one, which should increase the precision.
- The difference of the time stamp of its last pose saved with the time stamp of the pose closest to the target bottle should be greater or equal than the time x required to perform the motion. Otherwise this would mean that there is not enough information to estimate the final position.

3. Save its last pose and the closest one to the target bottle with the corresponding time stamps.
4. Calculate the translation vector obtained by the difference of the target last pose (obtained in step 1) with its closest one (obtained in step 3).
5. Sum the translation vector just calculated to the last pose of the nearest object (the one saved in step 3).
6. Find the time interval (difference of time stamps) that passed between the two poses from step 3 and sum it to the time stamp of the last pose of the target.

At this point the prediction is finished: with the last two steps, the predicted pose of the target bottle and and its time stamp when it will arrive there are calculated.

4.2.2 Limitations of this prediction approach

An alternative solution could have been that of observing the trajectories of the bottles over time and estimating the mean velocities (v_x, v_y) of the bottles in x and y directions. Then, using the uniform rectilinear motion formulas, calculating the pose of the target bottle at time t:

$$x_p(t) = x_{p_0} + v_x \cdot (t - t_0) \tag{4.5}$$

$$y_p(t) = y_{p_0} + v_y \cdot (t - t_0) \tag{4.6}$$

Both solutions work only in the case of a straight-line trajectory. Even though the approach proposed before may seem to translate well for any trajectory case, this is not the case for curved trajectories: figure 4.18 shows the problem that can be encountered.

A possible **solution** to this problem could be to use the poses history to estimate the center point (x_c, y_c) of the circle of motion and, knowing that the angular velocity ω between all objects is equivalent, the following equations of motion can be used:

$$x(t) = x_0 + r\cos(\theta_0 + \omega t) \tag{4.7}$$

$$x(t) = x_0 + r\cos(\theta_0 + \omega t) \tag{4.8}$$

Where:

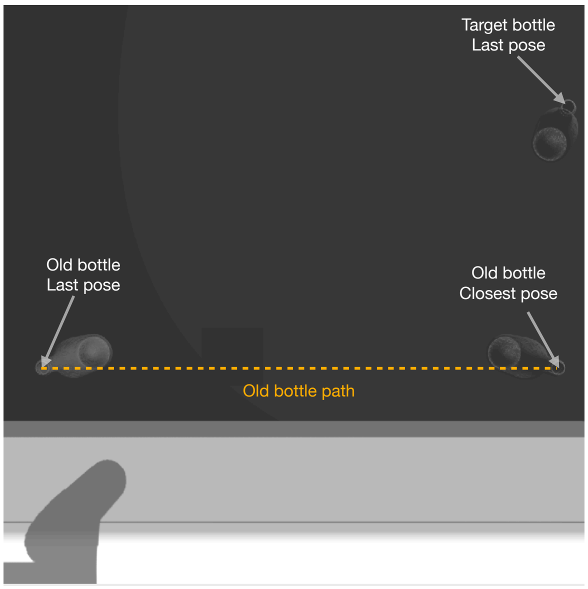


Figure 4.14: Scene containing both the target bottle and the older one satisfying the required conditions.

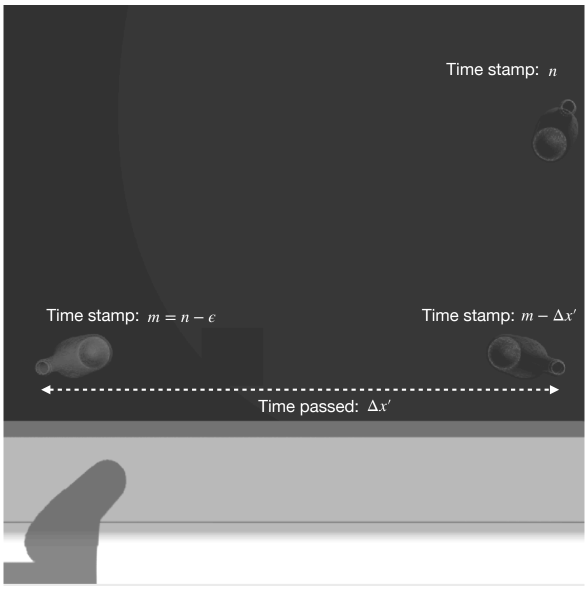


Figure 4.15: The current time stamp is n, the last pose of the saved bottle has time stamp $n - \varepsilon$, indicating that the time passed is small.

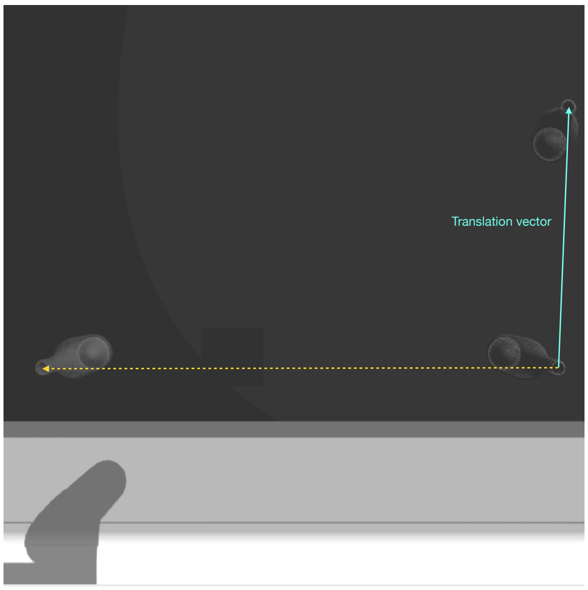


Figure 4.16: Representation of the translation vector that will be used to translate the final pose of the older bottle to predict the final pose of the target one.

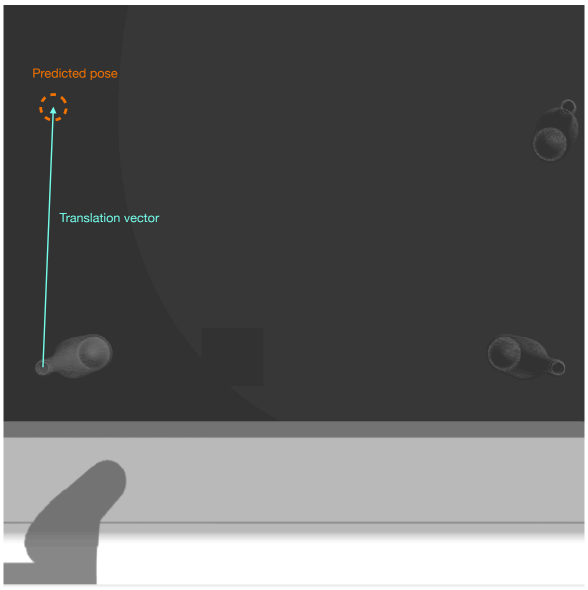


Figure 4.17: Prediction of the target bottle pose after $\Delta x'$ time.

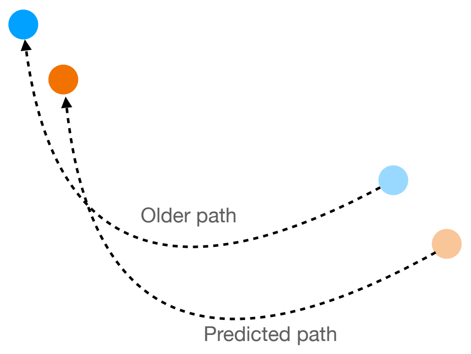


Figure 4.18: Scheme of the prediction that would be obtained with the system used.

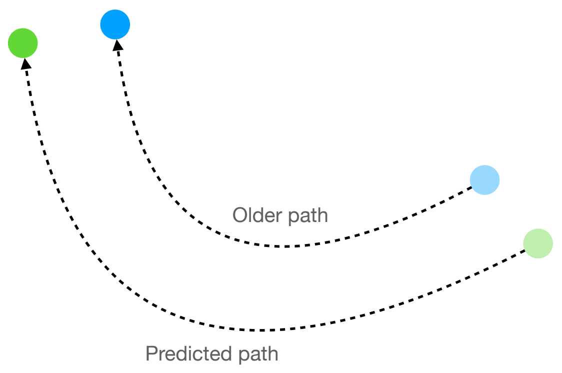


Figure 4.19: Scheme of the prediction that should be obtained instead.

- r is the radius obtained from (x_c, y_c) and (x_0, y_0).
- θ_0 is the initial angle corresponding to the position of the object at: $(x_0 + r, y_0)$.
- ω is the angular velocity.

Chapter 5

Object Picking

At this point, there is all the necessary information to perform the motion routines: grasp the object and place it in a predefined location in the workspace. Since the second part of the movement (the place routine) is very similar to the first one, this chapter will focus more on the pick routine.

Before exploring the main logic of the algorithm that performs the motion, it is necessary to explain what is MoveIt and how it was used in this application.

5.1 MoveIt

5.1.1 Motion planning

When the robot has to grasp something with its arm, it is necessary to move all its joints so that the final part (the end effector) can be at the proper location to pick the object. Moving the arm to achieve that position is a non-trivial task because it is necessary to produce the sequence of values that every joint must follow (in coordination with the other joints). This task, called motion planning, can be easily solved using the planners provided by MoveIt, then executed by sending the obtained trajectory to the robot controllers.

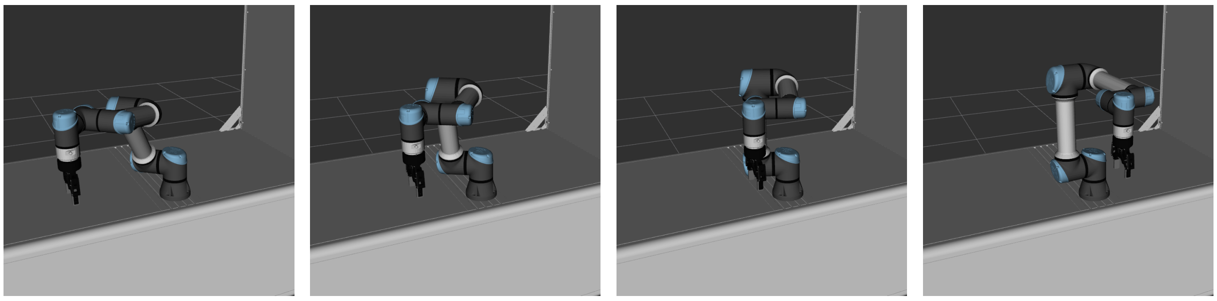


Figure 5.1: Trajectory example

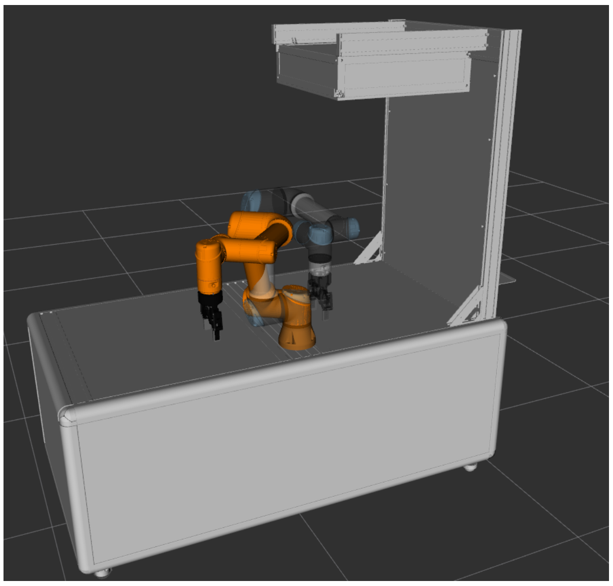


Figure 5.2: Robot's current and planned (in orange) states in RViz

5.1.2 MoveIt overview

MoveIt is an open-source motion planning framework that integrates with ROS to simplify complex robot control tasks such as the ones described before. MoveIt planners can avoid obstacles and optimize the planned path to generate smoother and more efficient collision-free trajectories between the initial position and the target object. This is critical in pick-and-place applications where the robot has to move within confined spaces without colliding with objects or itself.

MoveIt integrates with RViz (the ROS visualization tool) to provide a powerful visual interface for robot motion planning, making it easier to simulate, and debug complex robotic tasks. Through RViz, MoveIt displays the robot's current and planned states as shown in Figure 5.2. The current state represents the actual joint configuration from sensor feedback, while the planned state shows the predicted positions of the joints based on the goal.

RViz allows users to interact with the robot in real-time using the interactive marker or setting manually the joint states as shown in Figures 5.3 and 5.4. The marker allows the users to move and rotate the robot end-effector, then the values of the joints are evaluated automatically. This way it is possible to set new goal positions and test reachability.

5.1.3 Setup with a custom robot

In order to work, MoveIt has to be configured for the specific robot description. Some manufacturers may provide already configured MoveIt packages that work properly on their robots. Even though this is the case (Universal Robots provides moveit_config packages for its robots: UR3, UR3e, UR5, UR5e, etc.), the configuration had to be done from scratch due to the fact that, in this application, the robot description URDF does not include only the UR5e, but also the platform where it is located, the gripper mounted on its last frame, and the camera mounted near the base of the gripper. All these elements added to the robot must be taken into account

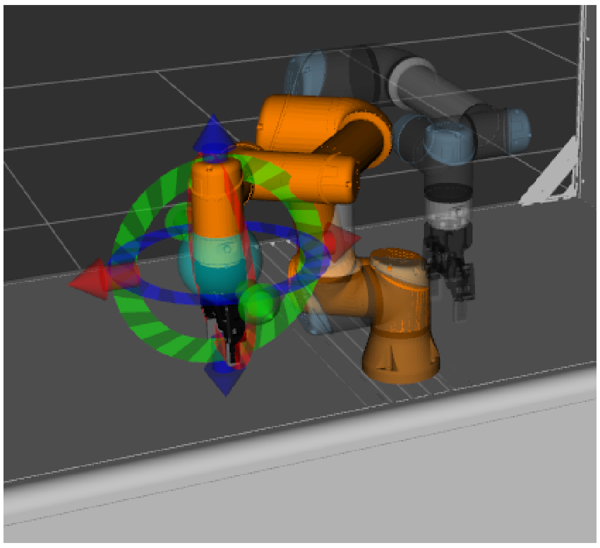
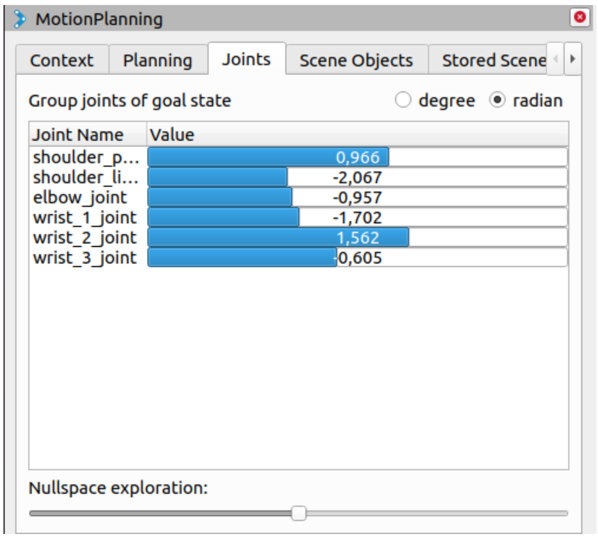


Figure 5.3: Interactive marker



Group joints of goal state   degree [✓ radian]

Joint Name	Value
shoulder_p...	0,966
shoulder_li...	-2,067
elbow_joint	-0,957
wrist_1_joint	-1,702
wrist_2_joint	1,562
wrist_3_joint	0,605

Nullspace exploration:

Figure 5.4: Joint states

to avoid collisions during motion planning.

Another reason why it is needed to re-configure MoveIt is that the configuration provided by the manufacturer is able to control only the robotic arm, not the gripper, as this latter component varies depending on the application.

To perform the setup, MoveIt provides the package ***moveit_setup_assistant*** that facilitates the configuration of a custom system starting from the URDF robot description file (defined in section 2.2). During this process, the self-collision matrix is generated: a matrix where pairs of links that will never collide during the motion are identified to optimize the collision avoidance process.

Setting joint limits

Once the setup is done, a MoveIt configuration package is created, where it is possible to modify the ***joint_limits.yaml*** file to specify the minimum and maximum positions (in radians) of the joints. This is useful to avoid that a desired pose is reached with an unwanted arm configuration as shown in Figure 5.6.

5.2 General operation

Once MoveIt is setup, it is possible to move the robot in the reachable workspace using the C++ *moveit::planning_interface::MoveGroupInterface* class. By using that class, it is possible to:

- Get the state of the current joint values.
- Set the speed of the robot motions (a float value from 0 to 1).
- Plan the motion, both of the manipulator and the gripper, using a target pose.

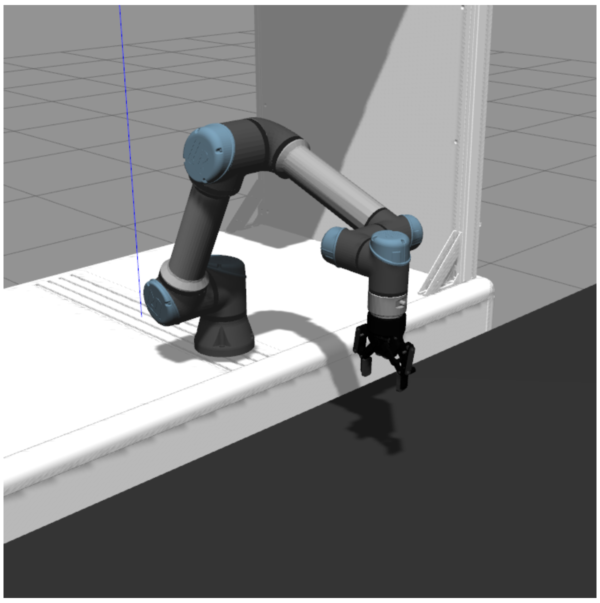


Figure 5.5: Correct arm position

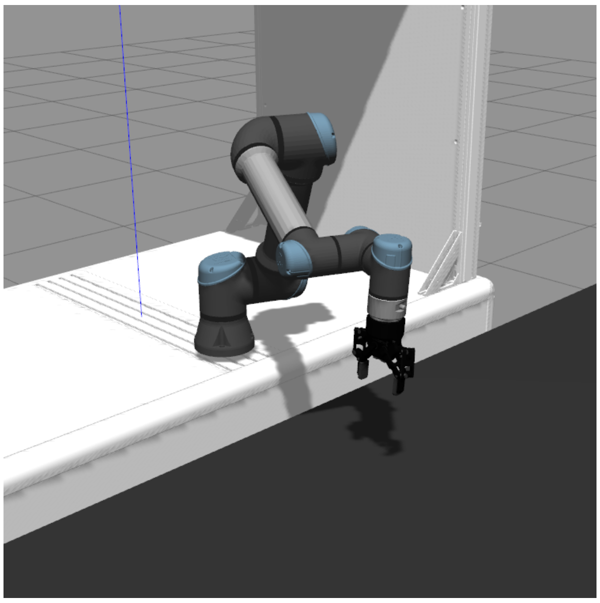


Figure 5.6: Alternative (complex) arm position

- Obtain useful planned trajectory information, like the time required to complete the movement.
- Execute the trajectory.

Even though there are many other functionalities of the class that can be explored, the ones described above are sufficient to perform the pick & place routines. The general pipeline for the pick motion is the following:

- Read the target position and plan a motion to reach it with maximum speed.
- If the time planned to reach the goal position is higher than the requested one, then abort the request and respond with a failure. Otherwise, execute the motion.
- Once the position has been reached, wait until the time-stamp corresponds to the predicted arrival time of the bottle.
- Then close the gripper and move back to the home (predefined) position.

In the target position, the z-coordinate, which essentially represents the distance between the gripper and the top of the bottle, is constant and can be retrieved once before the startup of the system by manually moving the end-effector to the correct height where it is able to grasp the objects.

The pipeline for the place routine is very simple, thus not explored deeply. It is similar to the pick one, except for the fact that the gripper will be opened (and not closed) when arriving to the target position, and the time-stamp information, passed in the request, is ignored as it is not requested to place the object at a precise desired time.

The approach described for the pick routine is valid under the ideal assumption that the gripper closing time is zero. In section 5.4 the alternative method used to solve this task in the application will be discussed.

5.3 Accuracy of predicted positions

At this point, it can be beneficial to test the accuracy of the predicted position of the bottles before actually grasping them. For doing that, the adopted approach was to move the robotic arm with the end-effector in the position where it should pick the object and, at the precise timestamp of when the bottle is assumed to be under the gripper (and the camera), take a screenshot of what the camera is viewing. This way it is straightforward to see if the bottle is centered in the image, and thus if the predicted position is correct or not.

5.3.1 Results

After testing this approach on some samples, the results were quite convincing: almost all bottles were near the center, as can be seen in Figure 5.7. Still, there is a small error that is not easy to avoid, in section 5.4 it will be discussed how the choice of the gripper affects this error tolerance.

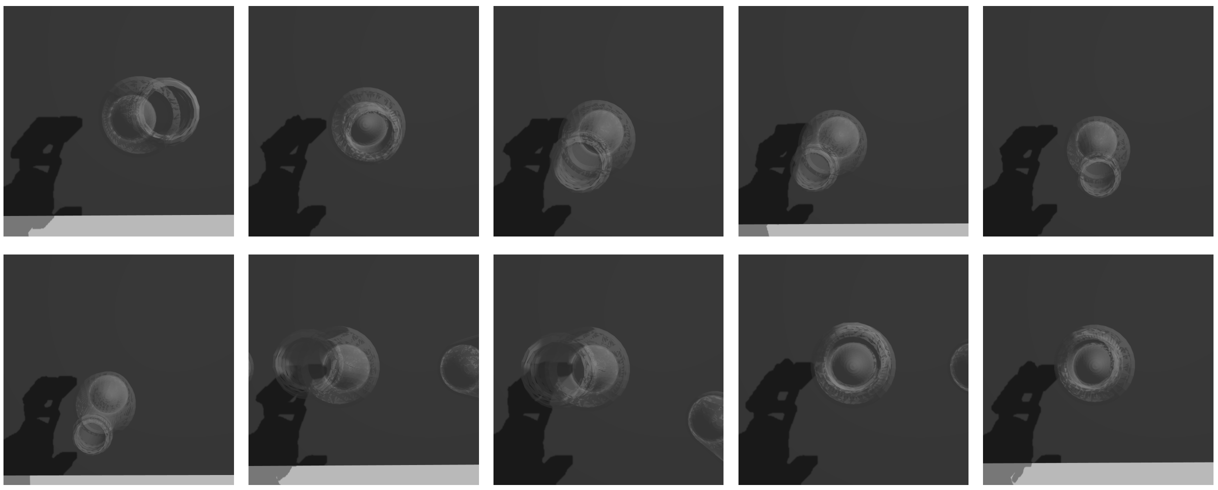


Figure 5.7: Screenshots captured at the time when the bottles were assumed to be under the gripper.

5.4 Limitations of the gripper

As anticipated in section 5.2, starting the closing motion of the gripper, when the object is already in the correct position to be grasped, would not work in a real application nor in the simulated one, as the behavior of the components in the simulation attempts to resemble reality. Depending on the gripper specifications, its closing time may vary from 0.05s upwards; in this case, the Robotiq 2f-85 takes 0.85s to close, starting from its maximum extension (85mm). As shown in Figure 5.8, the gripper reaches its closing position only when the bottle has already passed.

An immediate solution would be to start the motion with the gripper not fully opened, but only the minimum necessary for the bottle to pass through. Then, when the bottle is under the end-effector, close the fingers just enough to grasp the object. The problem with this approach is that it assumes that there is the utmost perfection of the predicted position of the bottle, which, in this application, is not the case.

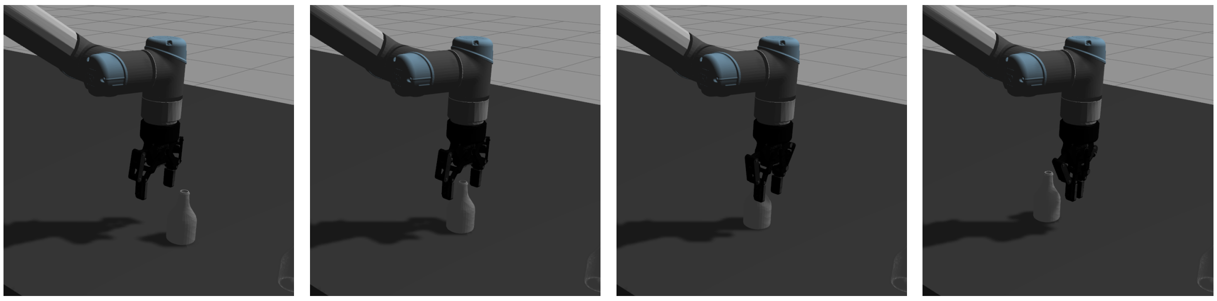


Figure 5.8: The gripper starts to close only when the bottle is under it.

5.4.1 Alternative approach

The adopted approach was to start the closing motion before the bottle was under the gripper. More in detail, MoveIt was used to plan the closing motion, obtaining the whole trajectory and thus the required time, then the execution was started at time: $bottle\_predicted\_time - gripper\_required\_time$. This way, the moment when the gripper reaches its closing position corresponds to the time when the bottle is ready to be grasped, as shown in Figure 5.9.

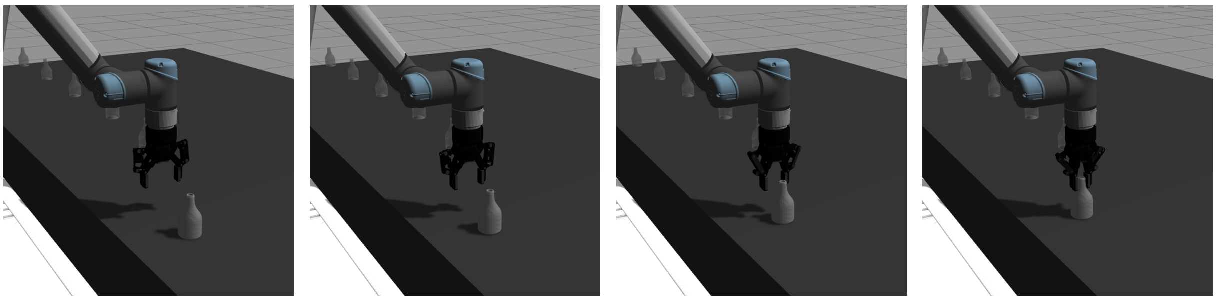


Figure 5.9: The gripper starts to close before the bottle is under it.

The $gripper\_required\_time$ depends, besides its technical specifications, on the initial opened position of the gripper and on the final one, which in turn depends on the section (diameter) of the bottle lip.

Starting the closing motion before the object is under the gripper may cause another problem: when the object has to pass through the fingers of the gripper, it may hit one of them due to the fact that, at that moment, the aperture is tighter than the maximum one. At this point, whether the grasping system works or not depends on the error of the predicted position of the bottle. The gap between the maximum aperture and the closed one, the bottle speed (on the conveyor belt), the finger width and the gripper speed determine the maximum vertical tolerance on the prediction error:

- If the gap increases, then it is more likely that the bottle passes through the gripper fingers without hitting one of them.

- If the speed of the bottle increases, then the gripper closing motion has to start earlier than usual, reducing the distance between the two fingers when the bottle has to pass through. This reduces the tolerance on the prediction error.
- If the finger width increases, then the closing motion can start later than usual. This way, the aperture that the bottle has when entering through the gripper fingers is higher, which increases tolerance.
- If the gripper speed increases, then (as before) the closing motion can start later than usual, thus increasing tolerance.

Below, different tables represent the margin that can be tolerated by the error of the predicted positions of the bottles.

Margin of error with Robotiq 2f-85										
Speed of the bottle (cm/s)	1	2	3	4	5	10	15	20	30	40
Margin when entering (cm)	6.5	5	3.3	2.5	2	1	0.7	0.5	0.3	0.25

Table 5.1: Margin of error depending on the speed of the bottle. Gripper width (stroke): 85mm, closing time: 0.85s, finger width: 20mm, bottle diameter: 20mm

Margin of error with OnRobot RG2										
Speed of the bottle (cm/s)	1	2	3	4	5	10	15	20	30	40
Margin when entering (cm)	9	9	9	9	9	5.2	3.5	2.6	1.8	1.3

Table 5.2: Margin of error depending on the speed of the bottle. Gripper width: 110mm, closing time: 0.21s, finger width: 20mm, bottle diameter: 20mm

Margin of error with Robotiq 2f-140										
Speed of the bottle (cm/s)	1	2	3	4	5	10	15	20	30	40
Margin when entering (cm)	12	12	8.3	6.3	5	2.5	1.7	1.3	0.8	0.6

Table 5.3: Margin of error depending on the speed of the bottle. Gripper width: 140mm, closing time: 0.56s finger width: 20mm, bottle diameter: 20mm

Margin of error with Neobotix AG-105-145										
Speed of the bottle (cm/s)	1	2	3	4	5	10	15	20	30	40
Margin when entering (cm)	12.5	7	4.7	3.5	2.8	1.4	0.9	0.7	0.5	0.4

Table 5.4: Margin of error depending on the speed of the bottle. Gripper width (stroke): 145mm, closing time: 0.9s, finger width: 17.5mm, bottle diameter: 20mm

In the end, with low bottle speeds (under 10cm/s), the Robotiq 2f-85 gripper is sufficient, although one with better performance (like the OnRobot RG2) would be a better choice.

Chapter 6

Results and Discussion

6.1 Pick & Place Results

In the end the overall pick & place motion can be described by the following sequence of images:

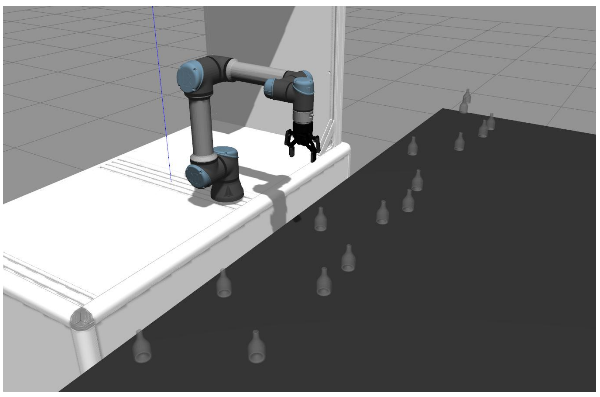


Figure 6.1: Initial position

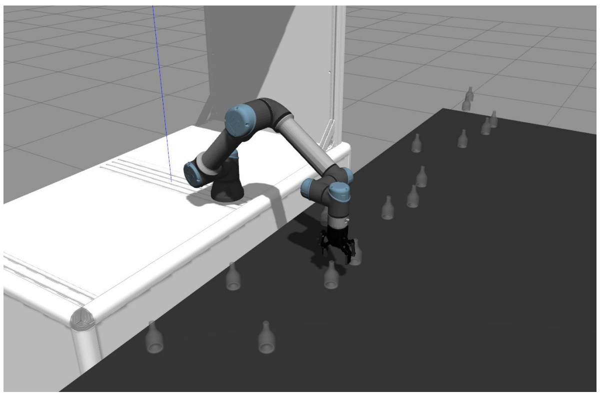


Figure 6.2: Target position (gripper opened)

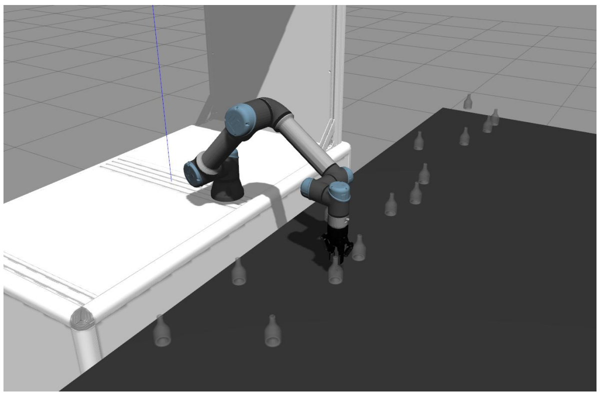


Figure 6.3: Target position (gripper closed)

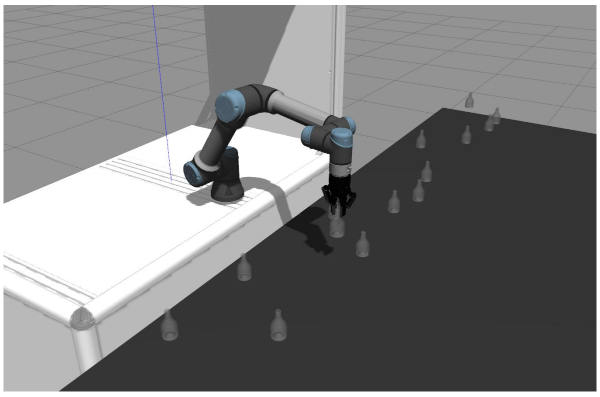


Figure 6.4: Lifted position with the grasped object

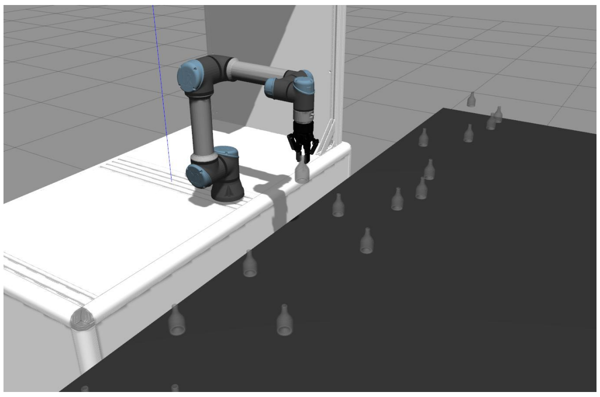


Figure 6.5: Back to home position

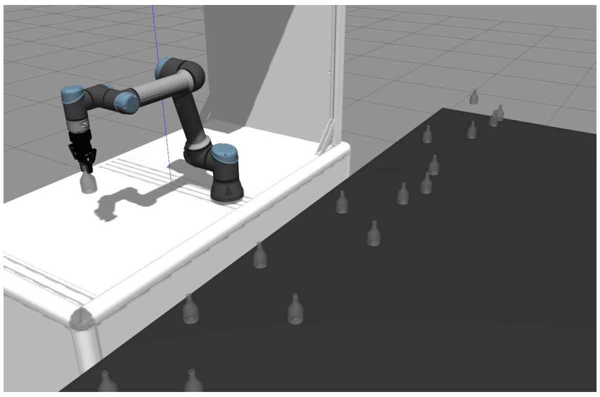


Figure 6.6: Approach position for placing the object

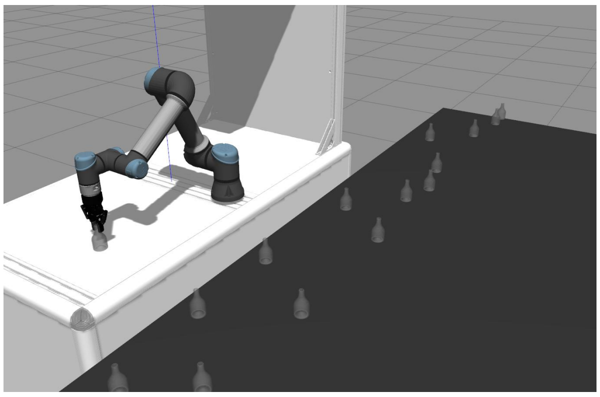


Figure 6.7: Place (final) position

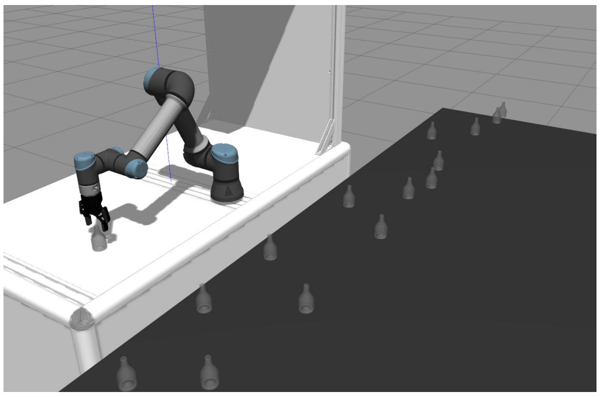


Figure 6.8: Opening gripper

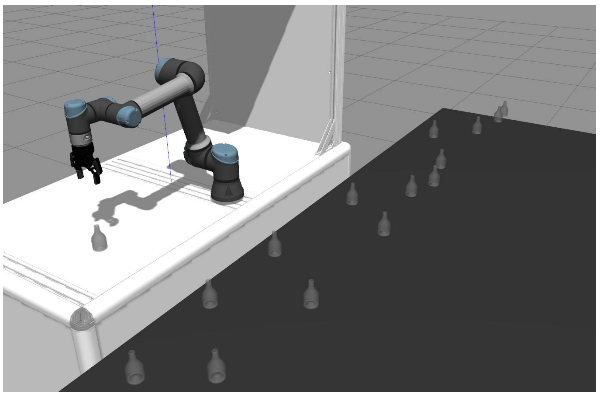


Figure 6.9: Lifted from the place position

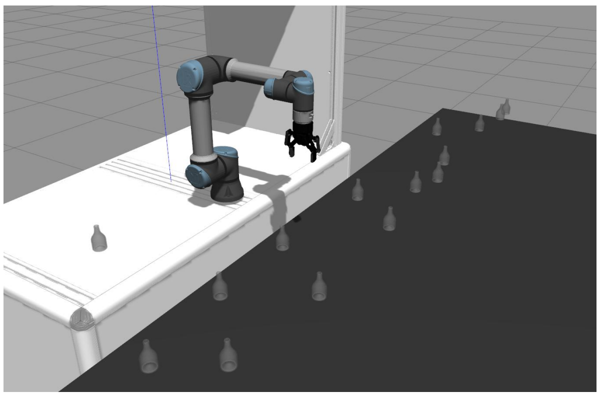


Figure 6.10: Move back to home position to start the routine again.

6.2 Future improvements

There are several points that can be improved to increase the system performance in terms of accuracy and speed.

Object detection speed

The precision that is given by using the YOLO v8 for detecting the positions of the objects in real-time applications is more than sufficient, what can be improved is the speed of this process. As was explained in section 4.1, the time required for the analysis of an image determines the maximum speed of the bottles that can be handled.

In order to reduce the time for inference, a possible solution could be converting the PyTorch weights file (.pt) into ONNX and then optimizing it for TensorRT (if using a compatible Nvidia GPU). TensorRT is a high-performance deep learning inference library developed by NVIDIA, designed to optimize models for deployment on Nvidia GPUs, improving inference speed and efficiency. TensorRT models offer a number of key features that contribute to the efficiency in high-speed inference:

- **Precision calibration**: TensorRT supports precision calibration, allowing models to be tuned to specific accuracy requirements. This includes support for reduced precision formats such as INT8 and FP16, which can further increase the speed of inference while maintaining acceptable levels of accuracy.
- **Layer fusion**: The TensorRT optimization process includes layer fusion, where multiple layers of a neural network are combined into a single operation. This improves inference speed by minimizing memory access and computation.

Prediction algorithm

The algorithm proposed for this application, to predict where a certain bottle will be after x seconds, is very simple and works under the assumption that the trajectory of the objects is linear. This may not be the case in certain applications where the robot is placed near a curved conveyor belt. For these situations, the algorithm has to be redesigned to include curved trajectories, or more complex ones if necessary.

Alternative pick movement

In this application, the robotic gripper is moved to the predicted position where the bottle will be and waits there until the object is ready to be grasped. However, this solution introduces several problems:

- **Collision with other objects**. If the predicted position intersects with another object, then the gripper will attempt to move into that position, potentially causing a collision. This issue is problematic when operating with a conveyor belt full of objects placed closely together.

- **Timing misalignment**. If the timing for closing the gripper is not perfectly synchronized with the arrival of the bottle under the gripper, there can be problems. If the gripper closes too early, the bottle may hit the fingers, and if the gripper closes too late, the bottle may not be grasped at all.

An alternative solution could be to follow the bottle with the gripper above it by using Visual Servoing techniques (controlling the robot's motion based on visual input). While moving sideways, the robot would approach the object, close the gripper and then move upward to complete the pick & place routine.

6.3 Possible applications

In the end, the proposed solution can be suitable for systems where there is already a line with objects sliding, and it is necessary to pick some of them for a random inspection, without meeting stringent requirements in terms of speed and precision.

High speeds cannot be achieved due to the specifications of the robotic arm and its gripper: these components are designed primarily for safe and flexible operations in environments that require interaction with humans.

The precision is limited to the fact that the final pose, where to pick the object, is estimated once, but that may vary due to some external factors, like changing the conveyor speed while performing the pick routine, or other external forces that may deflect the trajectory of the object (like vibrations). In addition, the precision of the predicted position can be affected by changes in the environment, such as lighting conditions: the vision system used to estimate the object's position might misinterpret the object's location.

Despite the fact that the performances are limited, there are many advantages on using such a stand-alone system:

- There is no need to interface with existing encoders or other sensors, as the application can determine the speed of the conveyor and the positions of the bottles just by using the camera.

- Adapting the robotic system to a new environment involves simply acquiring sufficient images of the new objects for the neural network and making minor adjustments to the settings, thus ensuring the system functions properly in the new context.

Glossary

Actuators devices that convert energy into motion, enabling the robotic arm to move.

Array a structured collection of elements used for storing data.

Conveyor belt a moving surface that transports items to and from the robotic arm's workspace.

Encoder a sensor that provides position or speed feedback to ensure precise movements of robotic components.

End-effector the tool at the end of the robotic arm, used for interacting with objects, such as a gripper or suction cup.

Eye-in-hand a vision system mounted on the robotic arm to capture images for precise positioning and guidance.

Fixed joint a type of joint or connection that does not allow movement between linked parts.

Framework a set of tools, libraries, and standards that streamline software development for specific applications.

Gripper an end-effector designed to grasp and hold objects.

Inference the process of applying a trained model to make predictions, like object detection.

Joint a connection between links that allows movement, such as rotation or sliding.

LiDAR a sensor that uses laser light to measure distances, creating detailed 3D maps of the robot's environment.

Link a rigid segment of a robotic arm that connects two joints.

Open-source software or hardware with source code available for modification and distribution by anyone.

Pick and Place a robotic process of picking up objects from one location and placing them in another.

Pixel coordinates the location of a pixel within an image, often used in vision-based tasks.

Prismatic joint a joint that allows linear sliding motion along an axis.

Revolute joint a joint that allows rotational movement around a fixed axis.

Robotic arm a programmable mechanical device that picks, moves and places objects in precise locations.

Spawn the process of generating and placing an object in the environment, such as on a conveyor belt.

TF tree a hierarchical structure that tracks coordinate frames for various parts of a robot.

Timestamp a recorded time associated with data or events, used for synchronization.

Trained weights parameters learned by a neural network model to make accurate inferences.

Wrist the joint of the robotic arm that controls the rotation and angle of the end effector.

Bibliography

[Att] ROS-Industrial Attic. *Robotiq ROS Package*. URL: https://github.com/ros-industrial-attic/robotiq.

[Cal] Calib.io. *Camera Calibration Pattern Generator*. URL: https://calib.io/pages/camera-calibration-pattern-generator.

[Cona] MoveIt Contributors. *MoveIt Calibration Repository on GitHub*. URL: https://github.com/moveit/moveit_calibration.

[Conb] MoveIt Contributors. *MoveIt Hand-Eye Calibration Tutorial*. URL: https://github.com/moveit/moveit_tutorials/blob/master/doc/hand_eye_calibration/hand_eye_calibration_tutorial.rst.

[Mat] MathWorks. *Camera Calibration*. URL: https://it.mathworks.com/help/vision/ug/camera-calibration.html.

[Ope] OpenCV. *Camera Calibration Tutorial*. URL: https://docs.opencv.org/4.x/dc/dbb/tutorial_py_calibration.html.

[Roba] Roboflow. *Roboflow - Computer Vision Platform*. URL: https://roboflow.com/.

[Robb] Open Robotics. *Gazebo Sim*. URL: https://gazebosim.org/home.

[Robc] Open Robotics. *ROS - Robot Operating System*. URL: https://www.ros.org/.

[Robd] PAL Robotics. *Gazebo ROS Link Attacher*. URL: https://github.com/pal-robotics/gazebo_ros_link_attacher.

[Robe] PickNik Robotics. *MoveIt*. URL: https://moveit.ai/.

[Robf] PickNik Robotics. *MoveIt Setup Assistant Tutorial*. URL: https://moveit.picknik.ai/main/doc/examples/setup_assistant/setup_assistant_tutorial.html.

[Robg] Robotiq. *Robotiq - Collaborative Robot Solutions*. URL: https://robotiq.com/it/.

[Robh] Universal Robots. *Universal Robots - Collaborative Robots*. URL: https://www.universal-robots.com/it/.

[Rok] Rokokoo. *Gazebo Conveyor Plugin*. URL: https://github.com/rokokoo/gazebo-conveyor.

[ROS] ROS-Industrial. *Universal Robot ROS Package*. URL: https://github.com/ros-industrial/universal_robot.

[Ult] Ultralytics. *YOLOv8 Object Detection Model*. URL: https://yolov8.com/.

[Vid] Analytics Vidhya. *A Comprehensive Guide for Camera Calibration in Computer Vision*. URL: https://www.analyticsvidhya.com/blog/2021/10/a-comprehensive-guide-for-camera-calibration-in-computer-vision/.

[Wik] ROS Wiki. *URDF Tutorials*. URL: http://wiki.ros.org/urdf/Tutorials.

[Ziv] Zivid. *Hand-Eye Calibration Solution*. URL: https://support.zivid.com/en/latest/academy/applications/hand-eye/hand-eye-calibration-solution.html.