

POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

**Human-Robot Collaboration  
through a new touch emulation  
system for assembly tasks**



**Supervisor**  
Prof. Marina Indri

**Candidate**  
Danilo Di Prima

December 2020

Turin, 18th December 2020

# Acknowledgements

In primis un ringraziamento va ai miei genitori, che mi hanno dato la possibilità di affrontare questo percorso di crescita. Ringrazio le mie nonne, che mi hanno sostenuto economicamente con affetto.

Un ringraziamento speciale va a Daniele, per essere stato sempre al mio fianco, anche nei momenti più difficili, e per avermi trasmesso le sue competenze linguistiche. Ringrazio Simona, che mi è stata vicino anche se da lontano. Ringrazio Calogero, che mi ha sempre accolto nonostante le mie lunghe assenze.

Ringrazio la Prof. Indri, per la sua disponibilità e puntualità. Ringrazio Fiorella e David che sono stati una mano d'aiuto, un consiglio immediato e ottima compagnia.

Infine ringrazio me stesso, per aver avuto sempre la forza di rialzarmi e lottare, per essermi posto obiettivi molto alti e per la mia determinazione che mi ha permesso di raggiungerli.

GRAZIE. ♡

## **Abstract**

Collaborative robots have been increasingly used in industries. This master thesis proposes a new mode of interaction to perform an industrial assembly task that employs an LCD, a Leap Motion controller and a camera as hardware and lets them interact with a Niryo One robot over ROS. The robot is an educational one, 3D printed with 6 degrees of freedom. The camera is a webcam, placed above the robot, focusing the workspace and displaying images on the screen. A new touch emulation system was proposed using the display and the Leap Motion controller. It consists of an initial calibration step which creates a virtual representation of the screen. Using the position and the orientation of the LCD, two virtual touch panels are created, parallel to the screen, one representing the touching panel that is very close to the screen, the second representing the farther hovering panel. Every time the index tip crosses a panel, a corresponding event is sent. A human operator, with the tip of his/her index, touches an object on the screen to perform a pick and place operation with that object, while the hovering panel was used to send feedback on the screen to show the current position of the tip of the index. Moreover the Find-Object application was used to recognise pieces on the workspace. This information is used by the robot to avoid any collision, and by the touch emulation system to recognise if an object is touched on the screen.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Background . . . . .	1
1.2	Overview . . . . .	3
<b>2</b>	<b>State of the art</b>	<b>5</b>
2.1	Human-Robot Interaction . . . . .	6
2.1.1	Categorisation . . . . .	6
2.1.2	Human Robot Collaboration . . . . .	8
2.2	Related works . . . . .	9
2.2.1	Teleoperation, step-by-step commands and learning by demon- stration . . . . .	10
2.2.2	Mixed reality . . . . .	12
2.2.3	Multimodalities . . . . .	15
2.3	New generation control devices . . . . .	15
2.3.1	Gestures . . . . .	15
2.3.2	Eye-tracking . . . . .	18
2.4	Interaction with feedback . . . . .	18
2.4.1	Asynchronous feedback . . . . .	19
2.4.2	Synchronous feedback . . . . .	20
2.4.3	Layered Touch Panel . . . . .	20
<b>3</b>	<b>ROS — Robot Operating System</b>	<b>23</b>
3.1	Concepts . . . . .	24
3.1.1	ROS Filesystem Level . . . . .	24
3.1.2	ROS Computation Graph Level . . . . .	26
3.1.3	ROS Community Level . . . . .	27
3.2	rqt . . . . .	28
3.2.1	rqt_graph . . . . .	28
3.2.2	rqt_console and rqt_logger_level . . . . .	29
3.2.3	rqt_join_trajectory_controller . . . . .	29
3.2.4	rqt_tf_tree . . . . .	30
3.3	Robot Model . . . . .	31

3.3.1	URDF . . . . .	31
3.3.2	Xacro . . . . .	33
3.4	Coordinate Frames and TransForms . . . . .	34
3.4.1	Why transforms . . . . .	34
3.4.2	static_transform_publisher and robot_state_publisher . . . . .	35
3.5	MoveIt . . . . .	36
3.5.1	Architecture . . . . .	36
3.5.2	Planning Scene . . . . .	37
3.5.3	Motion Planning . . . . .	38
3.5.4	Pick and place . . . . .	40
3.6	OpenCV . . . . .	41
3.6.1	find_object_2d . . . . .	41
3.7	Rviz . . . . .	42
3.7.1	Displays . . . . .	42
3.7.2	View . . . . .	45
3.8	Gazebo . . . . .	46
3.8.1	Components and SDFFormat . . . . .	46
3.8.2	ROS integration . . . . .	47
<b>4</b>	<b>Hardware</b> . . . . .	<b>51</b>
4.1	Leap Motion . . . . .	51
4.1.1	Architecture . . . . .	52
4.1.2	Coordinate frame . . . . .	52
4.1.3	Tracking model . . . . .	53
4.1.4	Touch emulation . . . . .	57
4.1.5	ROS integration . . . . .	58
4.2	Niryo One . . . . .	61
4.2.1	Niryo One ROS stack . . . . .	64
4.2.2	Simulation . . . . .	64
<b>5</b>	<b>System development</b> . . . . .	<b>71</b>
5.1	The concept . . . . .	71
5.2	System architecture . . . . .	72
5.3	Workspace . . . . .	74
5.3.1	Hardware . . . . .	74
5.3.2	Other components . . . . .	75
5.4	Touch emulation . . . . .	76
5.4.1	Screen definition . . . . .	78
5.4.2	Interaction system . . . . .	80
5.4.3	Feedback marker . . . . .	82
5.5	Object recognition . . . . .	84

5.6	Assembly task . . . . .	85
5.6.1	Planning scene . . . . .	86
5.6.2	Pick and place operation . . . . .	87
5.7	Simulation . . . . .	88
<b>6</b>	<b>Practical implementation and experimental tests</b>	<b>91</b>
6.1	Workspace . . . . .	91
6.2	Niryo One . . . . .	93
6.2.1	Niryo One Studio . . . . .	93
6.2.2	Programming . . . . .	96
6.3	Raspberry . . . . .	96
6.3.1	Camera . . . . .	97
6.3.2	Leap Motion . . . . .	98
6.4	Pick and place . . . . .	99
6.5	Experimental test . . . . .	100
<b>7</b>	<b>Conclusion</b>	<b>103</b>
7.1	Final remarks . . . . .	103
7.2	Future developments . . . . .	105
	<b>Bibliography</b>	<b>107</b>



# Chapter 1

## Introduction

The word “robot” comes from a Czech term that means “hard work” and it was used for the first time by the writer Karel Čapek in his play R.U.R. (Rossum’s Universal Robots). At Čapek’s times the term was used in many fields and since then it has always been used to indicate a machine which substitutes humans in their job.

The term “robotics” was used for the first time by the famous writes Isaac Asimov to indicate the science which studies robots, based on three laws (also known as Asimov’s Laws) [4]:

**first law** A robot may not injure a human being or, through inaction, allow a human being to come to harm.

**second law** A robot must obey the orders given it by human beings except where such orders would conflict with the First Law.

**third law** A robot must protect its own existence as long as such protection does not conflict with the First or Second Law.

### 1.1 Background

Nowadays robots are considered as machines that interact with the external environment and are able to modify it based on predefined laws and information about the environment and the robot itself. They are composed of a mechanical structure, which can be based on locomotion (wheels) or manipulation (manipulators, grippers), actuators, which enable the movement of the mechanical structure (motors), and sensors, which can be internal or external, depending on whether information comes from the robot itself or from the environment. The ability to combine the information, coming from sensors, and the execution of a task is given by the controller system, which represents the brain of the robot.

Robotics can be subdivided in industrial robotics and service (and advanced) robotics. Industrial robotics is based on some strong qualities like versatility, adaptability, strength, repeatability, accuracy, etc. It includes industrial manipulators and Automated Guided Vehicles (AGV). The former are mainly used to pick and place, palletise, measure, weld, cut, paint, etc., the latter are mainly used to move objects (also big and heavy ones) from one cell to another in an industrial process. Service robotics, instead, is based mainly on autonomy. This last allows the robots to move, interact and modify the environment without any a priori knowledge. Advanced robotics is mainly composed of mobile robots. They can be field robots, which work in environments dangerous for humans, and service robots, which work for humans to enhance their life quality.

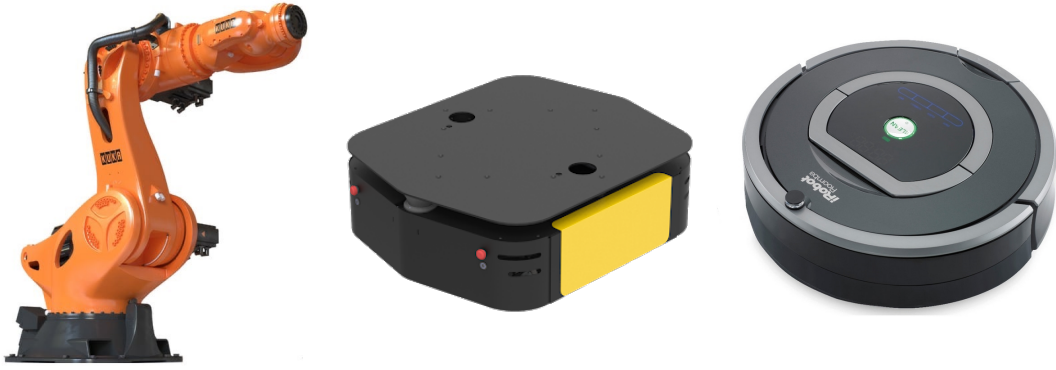


Figure 1.1: From left to right examples of robotic arm, AVG and service robot

The focus of this master thesis is to implement an assembly task using a new way of interaction. The pick and place operations, needed to perform the assembly task, are executed using a robotic arm that substitutes a generic industrial manipulator during the training phase.

An industrial manipulator is characterised by an open kinematic chain. This chain is composed by a sequence of bodies (links) connected by joints. In an open kinematic chain, each joint provides a degree of freedom. To determine the position and the orientation (pose) of an object in the 3D space, 6 degrees of freedom are needed: 3 for the position and 3 for the orientation.

Joints are of two types: prismatic, when they determine a translation between two links, and revolute, when they determine a rotation between two links.

The mechanical structure of a manipulator can be divided into two parts: the arm, which is responsible for the position of the end effector, and the wrist, which is responsible for the orientation of the end effector. This is mounted on the tip of the kinematic chain, and it depends on the task to be accomplished.

Depending on the sequence of joint types used in the arm, robots can be classified

in different typologies. The arm of industrial manipulators are often of anthropomorphic type, i.e. composed of three revolute joints, with the first one having vertical axis perpendicular to the horizontal parallel axes of the two remaining joints. Also the wrist is composed of three revolute joints. There are many types of wrists but the most common is the spherical one, in which the three joint axes intersect in one point. The spherical wrist is preferred despite it is difficult to realise, because it allows the kinematic decomposition of the position and the orientation of the end effector.

## 1.2 Overview

This master thesis aims at proposing a new methodology of Human-Robot Collaboration for pick and place operations to perform an assembly task. The interaction between the robot and the human has been realised implementing a new touch emulation system. The robotic system has been developed using the Niryo One robotic arm and some inexpensive devices like Leap Motion, in conjunction with a camera and an LCD.

The camera, placed above the robot, focusing the workspace, allows a human operator to visualise the pieces around the robot. The user, guided by a virtual feedback marker, can touch the image of a piece on the screen to select the piece to accomplish the assembly. The touch emulation system, thanks to the Leap Motion controller, is able to recognise which piece has been touched by the user and it sends the command to the robot to perform a pick and place operation for the specified piece.

**Chapter 1** This chapter contains the background and some notions of robotics, from the historical definition of a robot, Asimov's laws and the definition of robotics to the mechanical structure of a robot, its classification and the description of a robotic arm.

**Chapter 2** This chapter starts with the definition of Human-Robot Interaction. Then it continues with the definition of four criteria to divide Human-Robot Interaction into three subgroups and examines in depth Human-Robot Collaboration. Then the chapter continues with an overview of past studies classified according to the different interaction techniques and control devices used. The last part deals with the interaction with synchronous and asynchronous feedback and a new generation touch screen device is presented.

**Chapter 3** This chapter contains a detailed description of the Robot Operating System (ROS) and other components that work with it, from the motivation and basic concepts to some very useful tools to the programmer. It describes the models used for robot representation, how useful coordinate frames are,

and images manipulation. Finally, two softwares used for planning and simulation are presented.

**Chapter 4** The first part of this chapter focuses on Leap Motion. It describes the device from the physical point of view, its architecture and tracking model, used by the software to recognise humans hands. Finally, it describes the package used for the integration with ROS.

The second part describes the Niryo One robotic arm from its physical characteristics to the division in different layers of the Niryo One ROS stack. This part also shows the package used for the ROS integration.

**Chapter 5** This chapter contains the description of the development process step by step. First, it describes the elements that compose the workspace and how it is organised, second, it shows the interaction among Leap Motion, camera and the screen to realise the touch emulation with feedback, third, it describes how Find-Object application is used as a link between the touch emulation and the assembly task, finally it shows how the robot behaves during the pick and place operations in the simulated environment.

**Chapter 6** This chapter shows the practical implementation of the system. First, it highlights components used to compose the workspace, in comparison with the simulation with Gazebo. Second, it describes the ROS network created by the robot and Niryo One Studio application to command it. Third, it evidences the differences in the code between the virtual and real tests and how they were treated. Finally, it shows the result obtained through an experimental test.

**Chapter 7** The first part of the final chapter analysed the qualities of the system and some encountered issues. The pros are described through some examples, while, the methods used to avoid some cons are described. In the second part, possible future improvements are listed.



## Chapter 2

# State of the art

In the last 30 years industrial manipulators have been seen as autonomous machines that work separately from humans, surrounded with cages. They have been considered as substitutes to replace humans in hazardous and tedious manufacturing tasks with high accuracy and repeatability [29]. Year after year the technological improvement and the scientific research have brought to the development of logical capabilities in the robotic field to develop some robots able to coexist, cooperate and collaborate with humans. Moreover some tasks are too difficult and too expensive to be fully automated and, in the same way, too strenuous, tedious or dangerous for a human. Thanks to that humans and robots started to work and interact together in a safe way, developing the field called Human-Robot Interaction (HRI).



Figure 2.1: Examples of robots: Racer-7-1.0 by Comau [13] as example of industrial manipulator on the left and UR5e by Universal Robots [73] as example of collaborative robots on the right

## 2.1 Human-Robot Interaction

Human-Robot Interaction has been defined as “the process that conveys the human operators’ intention and interprets the task descriptions into a sequence of robot motions complying with the robot capabilities and the working requirements” [20] or as “a general term for all forms of interaction between human and robot” [62]. In general it is defined as Human-Machine Interaction because a robot is defined as a machine, or simply the interaction between actors which can be both humans and robots.

### 2.1.1 Categorisation

There are different levels of interaction, as Fang et al. explain in [20], and the identification of this level depends on two principles: the level of autonomy (LOA) which the robotic system is able to achieve and the proximity between the human and the robot during the task. Parasuraman et al. define automation as “the full or partial replacement of a function previously carried out by the human operator” [53], which is not all or none, but can vary across different levels. Table 2.1 shows 10 possible levels of automation from fully manual (lowest level) to fully automated (highest level). Thrun in [67] defines autonomy as “the robot’s ability to accommodate variations in its environment”. This explains that industrial robots have a low level of autonomy, while, service robots, which work in close proximity to people, have a high level of autonomy. This is because, on one hand, they have to guarantee human safety and, on the other hand, humans are little (or no) predictable. As expected, autonomy and proximity are related to each other.

LOW	1. The computer offers no assistance: human must take all decisions and actions.
	2. The computer offers a complete set of decision/action alternatives, or
	3. narrows the selection down to a few, or
	4. suggests one alternative
	5. executes that suggestion if the human approves, or
	6. allows the human a restricted time to veto before automatic execution,
	or
	7. executes automatically, then necessarily informs the human, and
	8. informs the human only if asked, or
HIGH	9. informs the human only if it, the computer, decides to.
	10. The computer decides everything, acts autonomously, ignoring the human.

Table 2.1: Levels of automation of decision and action selection [53]

As explained in [62], Human-Robot Interaction is considered a very large category which can be subdivided in subsets according to four criteria:

**workspace** defined as the common space between the human and the robot;

**working time** defined as the time the human spends in the workspace;

**aim** defined as the goal the actors want to achieve;

**contact** defined as the physical interaction which can be occasional or by accident.

Both humans and robots can have the same or different workspace, working time and aim. Human-Robot interaction, therefore, can be subdivided into the follow categories (figure 2.2):

**Human-Robot Coexistence (HRCoex or HRCx)**, also called coaction, is when humans and robots share the same workspace and the same working time but they have a different aim [2]. They don't have a common task and no contact is required. Usually the coexistence is limited to collision avoidance.

**Human-Robot Cooperation (HRCoop or HRCp)** is a specialisation of the HRCx. It is when humans and robots not only share the same workspace and working time but also have a common task to achieve [74]. To do that more sensors are needed, for example force-feedback and vision.

**Human-Robot Collaboration (HRCollab or HRC)** is a specialisation of HRCp. It is when humans and robots exchange information during the execution of a task [14] so they collaborate one over another. There can be collaboration, using two different modalities: physical collaboration and contactless collaboration. The first one is when there is physical contact, one of the four criteria to divide HRI into its subsets. Physical collaboration is intended as an intentional contact with an exchange of forces between the robot and the human. The robot can measure or estimate the exchanged force, predict the human intention and react in an appropriate way. The second modality is when the exchange of information is not directly with the robot itself but with the robotic system, using some sensors. It can be consequently divided into direct communication, like gestures and voice, and indirect communication, by identifying human intentions, for example through eye gaze or facial expressions.

Human-Robot Collaboration enables the achievement of very complex tasks, based on the fact that humans can contribute the decision-making, flexibility and the possibility to react to something that behaves in an unexpected way in the system, instead robots can perform what humans tell them to do without being too much complex, bringing strength, high precision and productivity to the system. In this way robots and humans become co-workers, able to work shoulder-by-shoulder

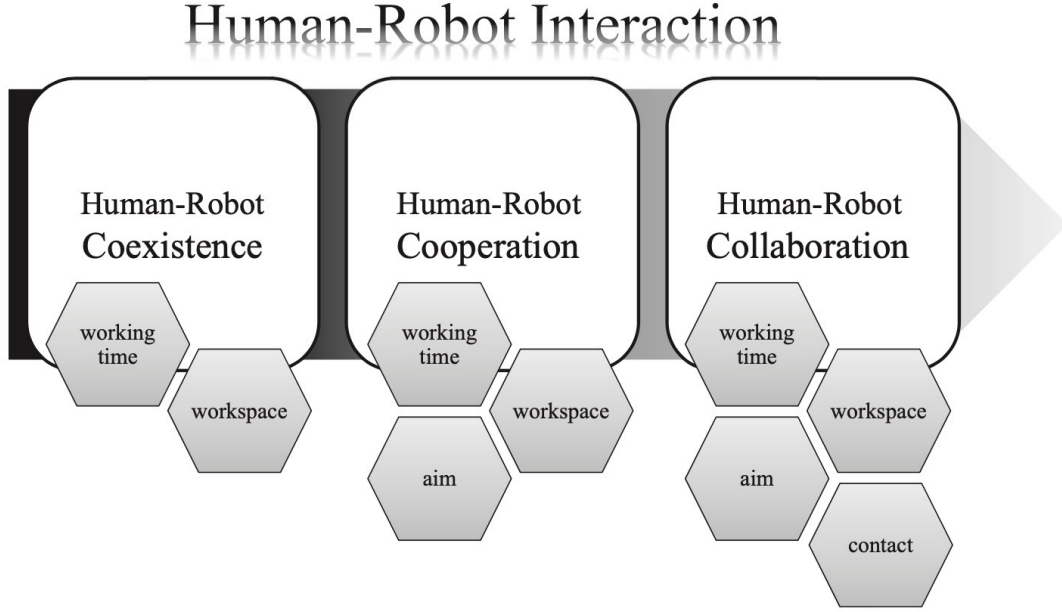


Figure 2.2: Division of Human-Robot Interaction in subsets accordingly to workspace, working time, aim and contact [62]

in a safe environment, in order to reach the same goal, combining the best qualities of each one (table 2.2).

Humans		Robots	
Advantages	Disadvantages	Advantages	Disadvantages
Dexterity	Weakness	Strength	No Process Knowledge
Flexibility	Fatigue	Endurance	Lack of Experience
Creativity	Imprecision	Precision	Lack of Creativity
Decision Making	Low Productivity	High Productivity	No Decision Power

Table 2.2: Comparison between human and robot qualities [64]

### 2.1.2 Human Robot Collaboration

The definition of collaborative robots — “cobots” — was coined to define a robot which directly interacts with a human worker [55] and “cobotics”, the neologism made up of collaborative and robotics, to indicate the science which involves cobot systems. These have a large use in industrial applications and each system adapts itself to a specific job, basing on three main aspects [46]:

**task characterisation** which is defined by many variables like the domain of application (domestic, industrial, military, medical, etc.), the visibility, the adaptation to new applications and the risk to damage the system or humans.

**role of operator** which depends on the complexity of the interaction. These roles are: operator, which pilots the robots, coworker, supervisor, which provides instructions to the robot, bystander, which is present in the workspace without interacting, maintenance operator or programmer. In the past, computer science knowledge was needed to program the robot to execute a specific task, instead, now, a lot of people can use robots without specific training or, at most, with a very short one.

**human-system interaction** which is based mostly on the proximity concept. Ergonomic design must be taken into account. The operator can be in contact with the robot, either nearby or very far; the interaction feedback can be immediate or can be differed, the interaction can be brief or continuous, moreover sensors play an important role in the remote communication. The interaction can be: physical (buttons, joystick or handling the robot), using touch-sensitive surfaces, visual (screens or glasses with virtual or augmented reality), using motion capture (eye tracking, fingers, hands, arms or body tracking) and sounds (alarms or voice detection and recognition).

Since the beginning of 1970s robots have been used in the industrial sector but at the beginning they were very expensive and only few people were able to program them. Since the industrial revolution 4.0, cobots have been very popular and essential in production in many fields [64]. They have been used in the manufacturing industry, automotive industries and assembly lines, especially for the manipulation of small and lightweight pieces. They are used first of all for picking, packing and palletising items, secondly for welding, thirdly for assembling items, fourthly for handling materials and finally for product inspection (figure 2.3). Artificial intelligence, computer vision and speech recognition are some techniques which allow a high degree of HRI, although these systems are not robust enough to be used in industrial applications. For this reason, also nowadays, robots support humans, who are in charge of the perception of the environment.

## 2.2 Related works

As we have seen previously, cobots are robots which can be moved from one task to another without a lot of effort, thanks to their reprogrammability. No experts are needed and almost everyone can use them without specific knowledge.

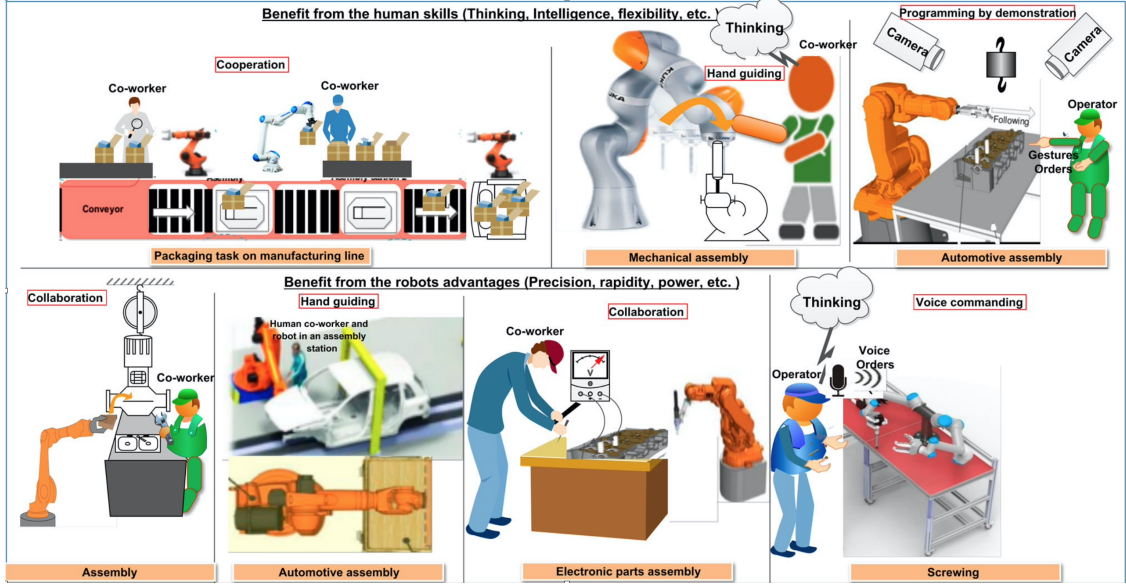


Figure 2.3: Different examples of Human-Robot Interaction [29]

### 2.2.1 Teleoperation, step-by-step commands and learning by demonstration

A human can command a robot using teleoperation, or using previously programmed simple motions and then combine these motions one after another, or the robot can learn for the first time how to accomplish a certain task.

Teleoperation, or telerobotics, is when a human operates on a robot from a distance [60] and the robot moves in real time. The operator sends a command to the robot to control its movement and then the robot sends back a signal which corresponds to the robot's state. There are two methods to teleoperate a robot: vision based method and sensor based method [12]. The first method is based on cameras. It can be a single inexpensive camera or a depth camera as Kinect [30] or Leap Motion [72] equipped with image processing algorithms in order to detect fingers, hands, arms or the whole body (figure 2.4). The robot is programmed in order to follow the movement of the body [30] or the movements of the hands and fingers (gestures) [56] or it can be teleoperated moving it, holding the end effector, in a virtual environment as if it were real [33].

The second method, which is based on sensors, uses them to send command to the robot. Using, for example, an haptic-based device [10], the users hold it as they were holding the end effector and they can retrieve information about the environment by using, for example, a haptic display and a tactile display for feedback [68] or simply a force sensor on the end effector of the robotic arm [16], in order to have a bilateral interface.

The user can also employ other sensors, for example the IMU sensor that contains



Figure 2.4: Examples of depth camera: Kinect [30] by Microsoft on the left and Leap Motion by Ultraleap [72] on the right

a gyroscope and an accelerometer, to control a robotic arm using the head movements. Therefore they take advantage of the three degrees of freedom of the human head to change only the position of the end effector [51] or use not only the head motion but also head gestures to control completely a robotic arm [32].

Users can also utilise other sensors, like magnetometer and electromyography sensors (EMG), together with gyroscope and accelerometer in order to obtain more information. Armbands, for example Myo armband, are used to control a robotic arm reading information about not only the orientation (very accurately) and position (not very precise) of the band but also reading the surface electromyography (sEMG) signals which allow the recognition of few simple hand gestures.

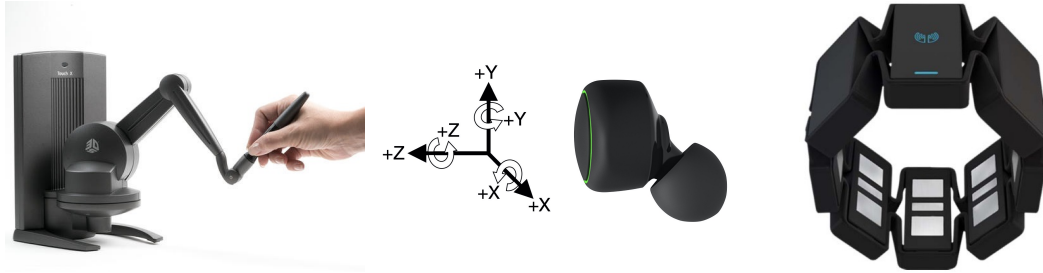


Figure 2.5: Examples of sensors used to teleoperate a robot, from left to right: Touch X by 3D SYSTEM [1] as haptic-based device, eSense [18] as IMU sensor and Myo armband [12] as EMG sensor.

Another possible way to collaborate with a robot is by guiding it through a task execution, not continuously like in teleoperation, but by following a step-by-step procedure. Of course the robot must know how to complete each single step but usually these stages in each process are made by simple actions, so they can be



reused even if the new task is very different from the previous one. Examples of these steps can be, pick an object, manipulate the object and then place it. In this case the cobot is able to perform a pick operation but it does not know which object to pick, how to manipulate it and where to place it. This missing information is given by the human who has the decision making skill to lead the robot.

Also in this case, pick and place operations can be made using different techniques and sensors, for example depth camera and gestures can be used to pick a piece and place it in the user's hand [22], or a smartphone touch display can be used to identify the pose in the 3D space where to place an object, using augmented reality [8] or using a 3D gaze tracking to select the object to pick, simply staring at it [5].

Last but not least, another important method frequently used in HRC is learning by demonstration. In this case the cobot is not able to perform any task without a learning process. There are mainly two techniques to perform the learning procedure. The first method is when the human moves the cobot with his/her hands and shows it exactly the same movements it must perform to accomplish that task. This process is called kinesthetic teaching and can be made using some force sensors on the robot [65] (figure 2.6) or Virtual reality. During this process the robot records the position of each joint to go from one point to another. In a recent study [28], a haptic sensor applied to the robotic arm was developed. This sensor is able to recognise and follow the movements of the human hand and also some gestures drawn on its surface.

The second technique uses observation. The robot collects data on what the human operator is doing using markers [66], cameras or other kinds of sensors, and then, after the training process, using some neural networks, it is able to perform the learned task. In this way no pre-programmed steps are needed and the robot is able to support the human in almost every task [38] (figure 2.6).

### 2.2.2 Mixed reality

Since humans and robots collaborate in the same workspace, safety reasons require to improve the communication between the operator and the cobot. One of the most critical issues is the difficulty of a human to interpret a robot's intent. Mixed Reality helps humans to do this.

Milgram in [45] introduced the concept of Virtuality Continuum (VC) (figure 2.7) in which real environment and virtual environment, or Virtual Reality (VR), are the extremes. The real environment is represented by real objects, while the virtual environment is represented by virtual objects, in other words computer images. Between these two environments there exists the Mixed Reality (MR). It is the general word which includes Augmented Reality (AR) and Augmented Virtuality (AV). The first consists in the vision of the real world with the addition of virtual objects, conversely the former consists in the vision of a virtual environment with the addition of real objects.





Figure 2.6: Examples of learning by demonstration: a man performs kinesthetic teaching moving the robotic arm with his hands [65] on the left and a woman shows the robot which trajectories it has to follow in order to perform a specific task [38] on the right.

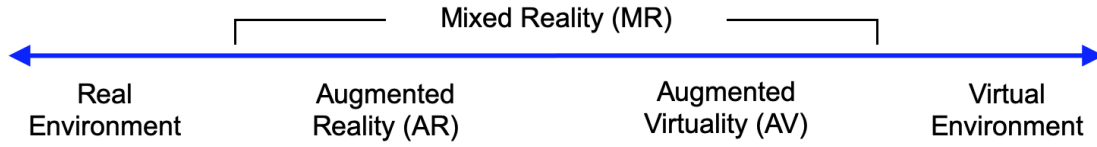


Figure 2.7: Representation of the Virtuality Continuum (VC) [45]

Virtual Reality is usually used when something is too risky for humans, for example a robotic arm is remotely teleoperated with a joystick because grit blasting creates an hazardous working environment for human operators [11]; virtual reality is also used to teach the robot to execute a task without actually stopping it or to teleoperate it in real time from another place (figure 2.8).

Augmented Reality is usually used to add information to the real environment. This can be done by adopting different techniques, for example using a smartphone display to select the place where to perform the pose action [7], or to visualise the point used by the end effector to pick the object, projected by an head-mounted-display (HMD) or a finger [39]. Another way to use AR is to add some virtual cues about the correspondent position of the end effector for the pick operation or a ghost image of the object after the place operation [3]. For safety reasons, instead, AR is usually used to show the planned path and possible contact with the human operator [59] (figure 2.8).

A study [9] uses a camera to detect markers which are then converted into objects, using Augmented Reality, and displayed on an LCD display. After that the user can select a virtual object with their hands and perform a virtual assembly task. Apart from this case, AR is usually used paired to a Head Mounted Display (HMD).

HMDs can be of two types: Optical See-Through (OST) and Video See-Through (VST). The first is when the user can see directly the environment (through a glass), the former is when the real environment is shown to the human's eyes through a display.

In most cases AR is used to add 3D virtual objects in a 3D environment. In other

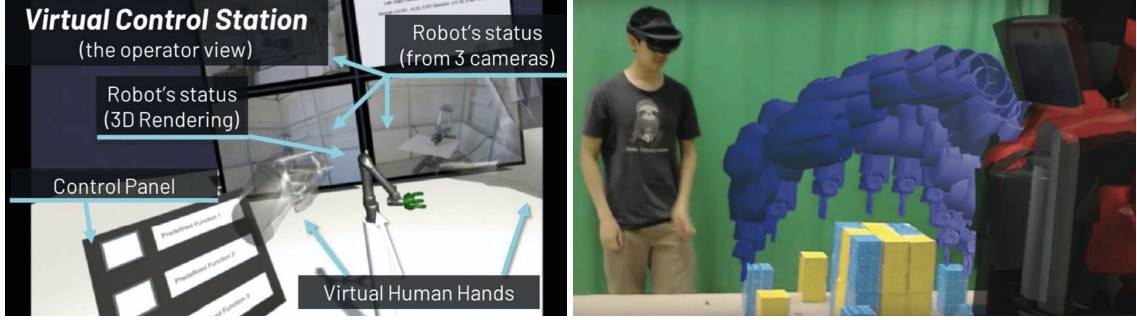


Figure 2.8: Examples of VR and AR with HMD: on the left a man teleoperates a robot using Virtual Reality [33] and on the right a man watches the robot planning path using Augment Reality [59]

cases AR is used to add 2D information on a plane (figure 2.9), usually a table, creating, therefore, a projection-based interactive system [27]. In these cases visual cues are about the position of objects on the plane and the operations to perform with them. A study [23] used this technique to develop an illustration-based language for robot programming in order to develop an easier way of interaction with the cobot.



Figure 2.9: Examples of AR used to add 2D virtual images on a table top [27] [23]

### 2.2.3 Multimodalities

In some cases one single modality for interaction is enough to accomplish the task in a correct way with enough accuracy, in other cases one modality is not sufficient because the system should know more information about the environment. To solve this problem multimodality was developed by adding more sensors, leading the system to a more complex state.

Multimodality can be done in different ways. The most common multimodality is the combination of gestures and voice to perform object manipulation [6], for example the voice is used to indicate which object to manipulate using specific characteristics, while gestures are used to communicate where to place the object. A study [54] used many sensors for the interaction: a glove with a sensor module and IMU sensors to indicate to the robot, respectively, the finger position measurement and the arm position, HMD and vibrators on the tips of the glove were used to receive visual and haptic feedback from the robot. In these two studies the human operator used different sensors to communicate different information to the robot. In other cases, different sensors are used for the same purpose (sensor fusion), in order to increase the accuracy, for example using an ocular interface and a haptic interface to better indicate the path the end effector has to follow [70].

## 2.3 New generation control devices

There are situations in which the communication between the human and the robot cannot be through physical interaction, for example, because of the transmission of microorganisms in a medical environment [15] or simply because the task is too hazardous to be accomplished by a human. To solve this problem tablets and voice commands can be used, but they are not effective in an industrial environment: here operators usually use gloves which do not allow the interaction with a touch screen or the high noise does not allow the operator to use vocal communication.

### 2.3.1 Gestures

Nowadays gestures are considered the new form of interaction which are suitable in the cases described above. No practice is required and they allow the operator to communicate with the robot in a manner similar to Human-Human interaction. Duvenhage says “gestures are a normal human way of communicating”, “it comes very naturally to people” [24].

Many different approaches to the use of gestures have been developed thanks to specific studies in this field. Gestural technologies can be based on handheld devices or on cameras. The former are usually wearable devices like gloves or bands which, however, could be bulky, unwieldy and they can limit the hand movements due to the presence of wires and sensors. The latter are depth cameras such as Kinect

[30] and Leap Motion [72] which detect the hand in a cloud of points generated by different IR cameras and powerful image processors. Using depth cameras, users are able to move their hands in a natural way. The drawback, however, is that movements are detected only in the restricted field of view near the camera itself.

Gestures can be classified in three groups [44]: locator, valuator and imager (table 2.3). The first group includes only simple pointing actions which indicate locations, the second group includes gestures which indicate extents of quantity, the last group includes gestures which indicate general images.

Group	Meaning	Example
Locator	Indicate location	Pointing
Valuator	Indicate extents of quantity	"this size" or "rotate this much"
Imager	Indicate general images	sign language, body language, geometric figures

Table 2.3: Gestures classification

Gleeson et al. in [26] studied gestures for industries. As a reliable verbal communication between humans is very difficult, due to the noisy industrial conditions, similarly also the communication between a human and a robot can be complicated. After the analysis of different tasks, they were grouped in three main categories: firstly, part acquisition, which includes the selection of a part, the location in the supply area, the acquisition and the transfer in the work area; secondly, part manipulation, which includes part insertion, feature location and alignment and general part placement; finally, part operation, which includes all actions done to a part after it has been placed.

During the experiment two operators communicated only using gestures, with their mouths and eyes covered. One operator showed the other one which action to perform using body language.

The result was that 276 communication events were coded, reduced to eleven different terms and nine basic gestures (figure 2.10), each one executed using one single hand with the unique exception of the two-handed "swap" gesture. Figure 2.10 shows that the correspondence term-gesture is not one-to-one but many-to-many. Moreover "point at part" gesture is connected with six distinct terms. It is the most used gesture because it is very simple, but its meaning depends on the task context.

Sakr et al. in [60] based their study on the comparison of the orthographic vision-based interface, developed in this work, with respect to the standard vision, in the teleoperation of a robotic arm.

The experiment was based on a pick and place task in which each participant had to gently pick a toy and place it in a target container. The toy was placed on a box, identified by a QR code, used by an inexpensive camera to better estimate

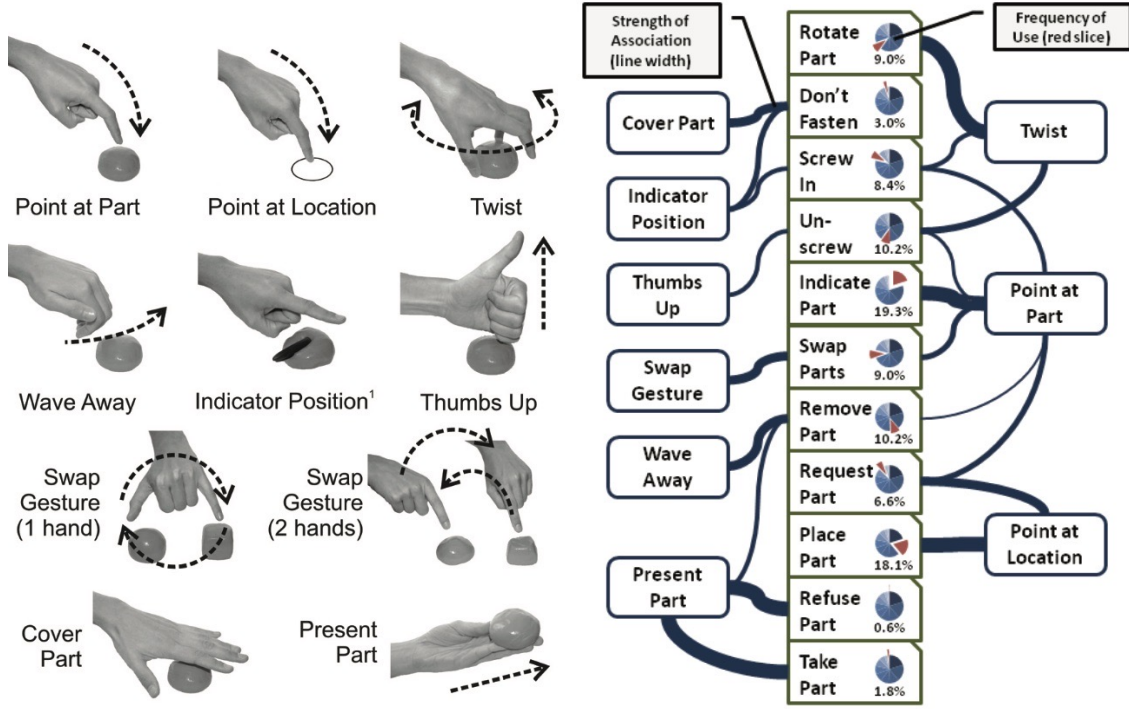


Figure 2.10: On the left, nine basic gestures identified in the study and on the right the association among terms and gestures [26]

the position of the toy. Moreover the box could be pressed by the participant in such a way as to estimate how much gently it was picked. In addition, the study was not based only on the two different vision systems but also on three different modalities, because each participant had to perform the task three times using a joystick, a keyboard and Leap Motion.

The result of this study showed a significant reduction in task completion times using the new orthographic vision system, respectively of 50%, 44%, and 35% using the joystick, the keyboard, and Leap Motion. Moreover Leap Motion has the smallest completion time of  $37 \pm 19$  s compared to  $42 \pm 18$  s and  $51 \pm 23$  s using the keyboard and the joystick, respectively. These results can be interpreted saying, not only that this new developed vision system is better than the standard one, but also that Leap Motion has the smallest completion time independently of the vision system used, maybe because it allows humans to interact in a more natural way, even when they are not accustomed with this device.



### **2.3.2 Eye-tracking**

Sometimes human operators, especially in some industrial applications, can have busy hands, so they cannot interact with the robot system using gestures. Moreover gestures nowadays are not very precise and they usually need more complex systems with sensor fusion abilities. A new possible technology which can be used as way to interact leaving both hands free is eye-tracking.

Dünser et al. in [17] based their study on a comparison between old and new generation control mechanisms, for a pick and place task. As old generation control systems a mouse and a touch screen were used, while eye-tracking was used, as new generation one, in combination with dwell time or mouse click. It was thought that the combination eye-tracking-dwell was the most natural and the most useful when the human operator, who interacted with the robot, had busy hands and could not use them. On the other hand, it was not possible to distinguish when a human was gazing at an object or simply glancing at it, so eye-tracking-click interaction was taken into account in the comparison. The comparison was done to know if gaze interaction is effectively an advantage in a control system or it is simply a novelty which performs well only in simple tasks.

The study was performed in a simulated environment for two reasons: firstly, because in this way all variables were measurable and possible uncertainties, due to an inexpensive system, were avoided, and secondly, because they wanted to detach the comparison from the specific system and, therefore, generalise the study.

The variable to measure was the movement time. The experiment was composed of two circles: one, in red, which represented the target, had to be selected first; the second, a black one, represented the destination and it had to be selected after the first. The time was measured from the selection of the first circle to the selection of the second

The result of this experiment was that touchscreen interaction had the smallest movement time, followed by mouse, eye-tracking-dwell and eye-tracking-click. Despite the touchscreen interaction was evaluated as the one which required the highest physical demand, it was also perceived, like the mouse interaction, as the most precise technique. Results, therefore, showed that using eye-tracking devices where they are not needed is not useful, maybe because so far they have not been good enough to be used in a quite complex task, or maybe because people use touchscreen devices everyday and they are not so used to new generation devices as to eye-tracking.

## **2.4 Interaction with feedback**

Year after year cobots interact better with humans and different techniques have been developed to improve the communication. However Human-Human Interaction is better than Human-Robot Interaction principally because it is based on

a bidirectional channel. In a dialogue between two speakers, while one is speaking, the other uses words, facial expressions or gestures to answer, in other words humans use various forms of feedback to interact.

### 2.4.1 Asynchronous feedback

Quintero et al. [57] performed an experiment using the pointing gesture in a pick and place task. The study used this gesture because it is a form of non-verbal communication and it can be used for Human-Human, Human-Robot and Robot-Human Interaction. Moreover this gesture is very simple and universally understandable.

The experiment focused on the picking of one of many objects placed on a table and placing it in the appropriate container and was conducted for the three types of interaction.

When the experiment was conducted for Human-Robot Interaction, a depth camera was used to detect the body gesture, the pointed object and the positions of the objects on the table. The gesture can be “stand by”, “pointing”, “yes” or “no”, each one represented by a specific human body position. The pointed object is selected as the closest object to the projected point, obtained by the intersection between the plane on which objects are laid and the straight line between the human head and the hand.

For each kind of interaction, three configurations were used:

- non-feedback normal, in which the human indicates one of the objects randomly placed on the table and the robot picks it up;
- non-feedback line, in which the human indicates one of the objects placed along a line on the table and the robot picks it up;
- feedback line, in which the human indicates one of the objects placed along a line on the table and the robot moves the end effector on it. If the feedback is positive the robot picks the object up, otherwise it moves the hand effector to the grasping pose of the nearest object, waiting for new feedback. This procedure is repeated until the right object is selected.

Results (figure 2.11), in term of success rate, show that Robot-Human Interaction is the most difficult to perform without misinterpretations; Human-Human Interaction is very difficult when objects are organised in a line; finally, when the feedback is available, the success rate is the highest, reaching the maximum, independently of the type of interaction.

In this last article the human operator knows which object the robot has selected only after it moves on top of it and the robot picks the right object moving the end effector from the grasping pose of one object to another one until the answer to the feedback is positive. This can last a lot and the communication is not so quick.

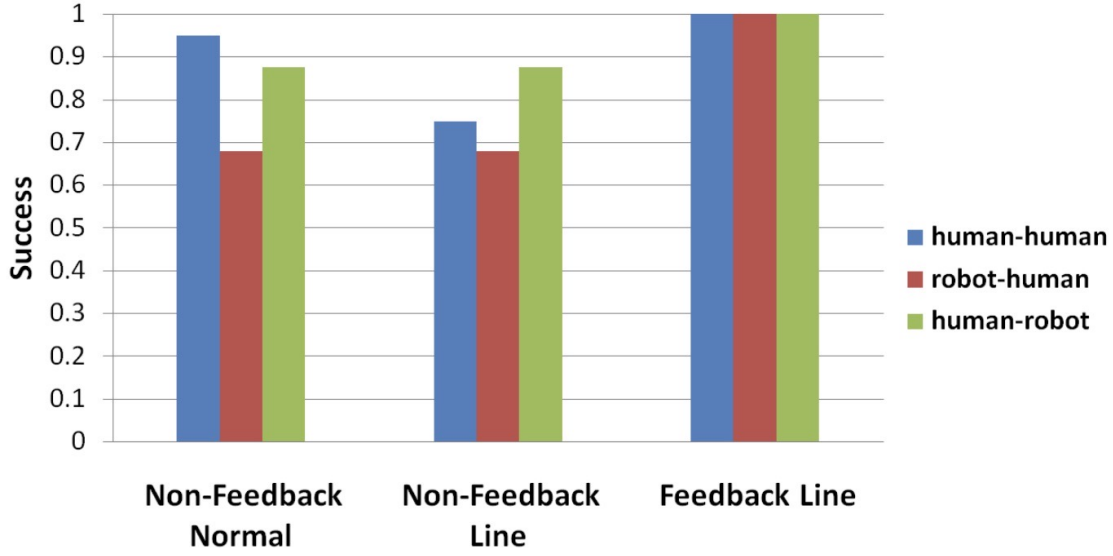


Figure 2.11: Commander and assistant experiment success rate [57]

### 2.4.2 Synchronous feedback

In order to have immediate feedback, Sato and Sakane proposed an interface which uses an Interactive Hand Pointer (IHP) that projects a mark in the real workspace [61].

This system uses the concept of Augmented Reality: two cameras detect the pointing gesture, using a Tracking Vision System (TVS), and an LCD projector is used to project the mark on the workspace. The marker position in the workspace is obtained by the intersection between the horizontal plane and the straight line determined by two points: the “tip point” and the “base point”. The tip point is identified by the tip of the finger, and the base point, in this case, by the base of the finger. In other cases the base point can be identified by the head position [57] or by a Virtual Projection Origin (VPO) [44] (figure 2.12). Using the VPO an assumption is made: all projection lines converge into one single point. This point is identified during a calibration procedure while the user points to some predefined targets.

### 2.4.3 Layered Touch Panel

Tsukada and Hoshino proposed an improvement of the classical touch screen used as input device. They defined the Layered Touch Panel as a touch screen with two levels [69].



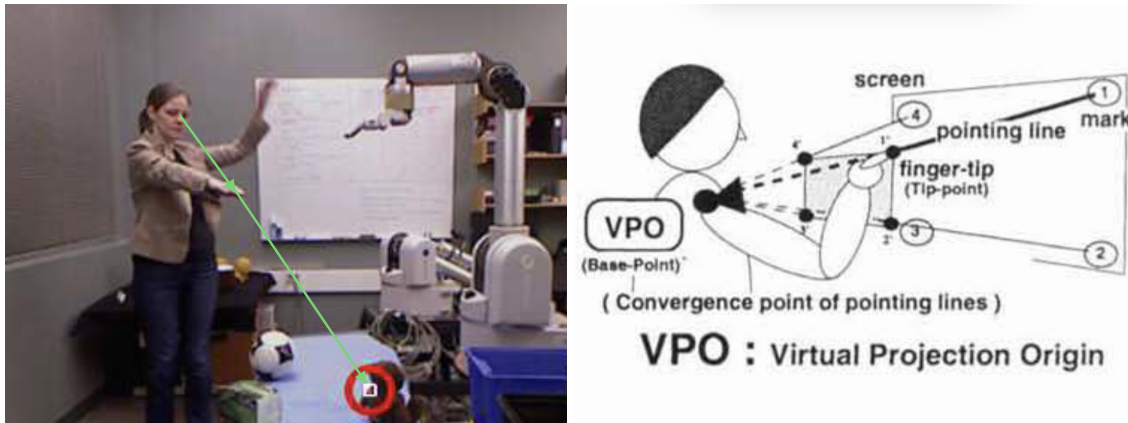


Figure 2.12: Two examples of different usages of the base point: on the left it is defined by the human head [57], on the right it is defined by the Virtual Projection Origin [44]

Their work shows a “Screen Layer Touch Panel (Screen Layer TP)”, which represents the screen surface, and a “Infrared Rays Layer Touch Panel (IR Layer TP)”, which is above the screen, parallel to it, at a small distance (figure 2.13)

The user is able to interact with this device without any physical contact or by

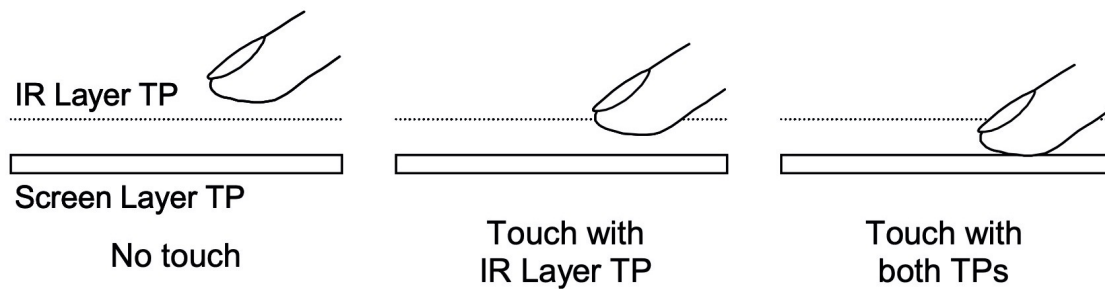


Figure 2.13: Touch states of Layered Touch Panel [69]

touching the surface crossing the IR Layer. This allows it to have not only two states, as a common input device, but rather three. For example, a mouse only has the pointing event and the clicking event but there is no possibility of a nothing state; a touchscreen, instead, has only the touch event or none, but no pointing event is possible; conversely, the Layer Touch Panel has all these three kinds of events (table 2.4). The pointing event added to the Layered Touch Panel is some new further information during the interaction with the device. These data can be used as feedback in order to implement a communication from the machine to the human.

Input state	Normal touch panel	Mouse	Layered Touch Panel
Nothing	o	x	o
Pointign event	x	o	o
Touching or click- ing event	o	o	o

Table 2.4: Available input states [69]

## Chapter 3

# ROS — Robot Operating System

ROS is the acronym for Robot Operating System and it is a meta-operating system that runs on Unix-based platforms [36]. It provides not only all services that an operating system usually does, but also libraries and tools that allow the user to run ROS across multiple machines. In other words ROS is fundamental for robots: it is the core, that interconnect each part of the robot, and the brain, that allows the robot to reach the assigned goals.

ROS, as other operating systems do, allows the user to have different types of communication: synchronous over services and asynchronous over topics. Moreover it is possible to store data on the Parameter Server.

ROS is composed of some elementary parts called nodes [58]. They are processes that build a peer-to-peer network, possibly also among different machines, which represent the ROS computational graph level. Nodes should be designed in order to be executed independently of others but also in such a way that they can be easily integrated in a more complex system. In this way the code uploaded in the ROS community can be reused by researches and developers.

There are many robotics software platforms but ROS has many benefits compared to other. Some advantages are:

**Thin** ROS is designed in order to be used with other robot software frameworks and easily integrated with them.

**ROS-agnostic libraries** ROS libraries are developed with a clean functional interfaces in order to be used without restriction.

**Language independence** Each ROS project can be implemented in any modern programming language. It was full implemented for C++, Python and Lisp; moreover there are experimental libraries for Java and Lua.

**Easy testing** ROS has its own test framework (roctest) that helps the developer to check and analyse the system.

**Scaling** ROS was developed to be modular. It works well on a single machine but its strength is to be easily integrated in a wide network of nodes.

Thanks to the fact that ROS is open source, researches and developers can implement their own code and share it in the ROS community in order to allow other people to use it. The ROS core is very basic, but there are a lot of libraries and packages that can be downloaded from the ROS repository that help the user to reuse the code.

## 3.1 Concepts

ROS can be split in three main levels: the Filesystem Level, the Computation Graph Level and the Community Level. Each level is described below.

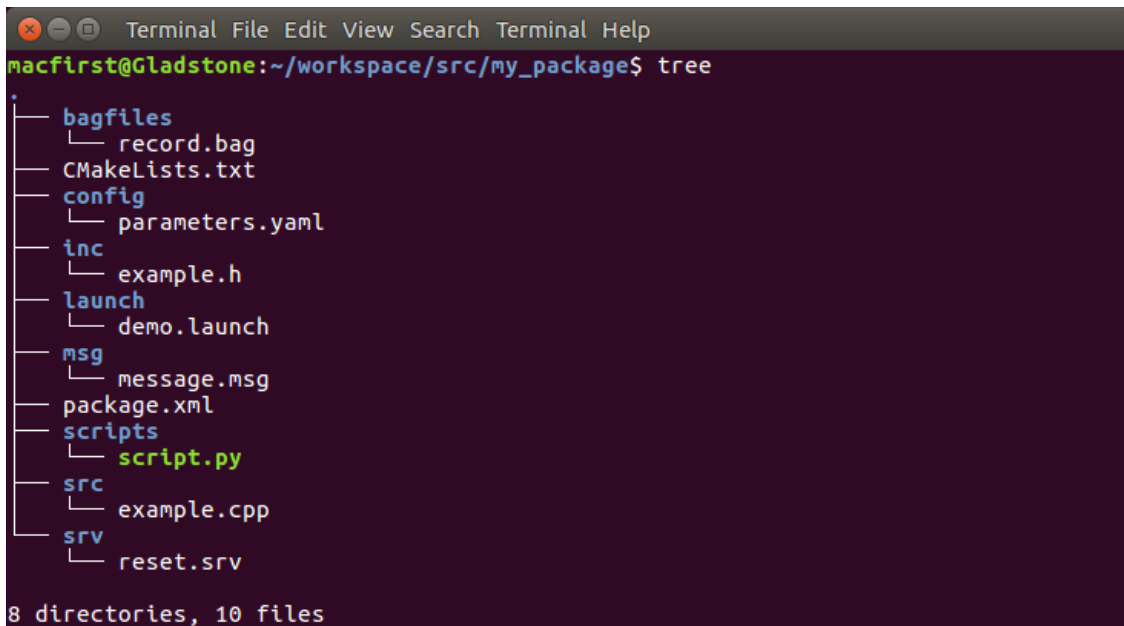
### 3.1.1 ROS Filesystem Level

The Filesystem Level includes all kinds of resources that you use working with ROS and they are:

**Packages** ROS Filesystem is organised, at the first level, in packages. A package is defined as the most atomic build item in ROS. In other words it is the smallest thing that can be built and released. A package is, therefore, an independent unit that provides some useful functionalities, able to work by itself but that can be also interconnected in a more complex system in order to reuse the code. Packages can be created or downloaded from the ROS repository (see section 3.1.3). They can contain code developed by the user and third-party code and they usually reflect the following structure (figure 3.1):

- package.xml is an XML file and represents the Package Manifest (described below).
- CMakeLists.txt represents the CMake build file to build packages. It contains dependencies and describes how to build the code and where to install it;
- scripts/ is a folder which contains executable scrips (usually Python);
- inc/ is a folder containing header files;
- src/ is a folder containing C++ source files;
- msg/ is a folder containing message type files (see section 3.1.2);
- srv/ is a folder containing service type files (see section 3.1.2);

- `bagfiles/` is a folder which contains bag files (see section 3.1.2);
- `launch` is a folder containing launch files. This kind of file is written using XML language and it allows the user to run many nodes (see section 3.1.2) at a time and also to include other launch file. These files are very used in ROS in order to reuse the code;
- `config` is a folder which contains all configuration parameters needed during the execution. These files have extension `.yaml` and they are usually used inside launch files to load parameters on the Parameter Server.



```
macfirst@Gladstone:~/workspace/src/my_package$ tree
.
├── bagfiles
│   └── record.bag
├── CMakeLists.txt
├── config
│   └── parameters.yaml
├── inc
│   └── example.h
├── launch
│   └── demo.launch
├── msg
│   └── message.msg
├── package.xml
├── scripts
│   └── script.py
├── src
│   └── example.cpp
├── srv
│   └── reset.srv
└──
```

8 directories, 10 files

Figure 3.1: Tree visualisation of a generic package structure.

**Metapackages** Metapackages are a special type of packages which do not contain files or code but only references to other packages so that different packages are related to one another.

**Package Manifests** Package manifest is a `.xml` file which describes a package. It contains the package name, version, description, maintainer, licence and dependencies on other packages. A package manifest is automatically generated whenever a new package is created with the `catkin` command.

**Message (msg) types** Messages are `.msg` files stored in `package_name/msg/` folder. They describe the message type that a node can publish. Each message contains a list of fields and each field consists of a pair: type and name. The field type can be a built-in type or another message type and it can be a single variable or an array.

**Service (srv) types** Services are .srv files stored in `package_name/srv/` folder. They are based on messages and implement a request/response communication between nodes. They are formed by a request and a response message type divided by “---”.

### 3.1.2 ROS Computation Graph Level

The Computation Graph Level includes all elements that create the peer-to-peer network of processes. These elements are:

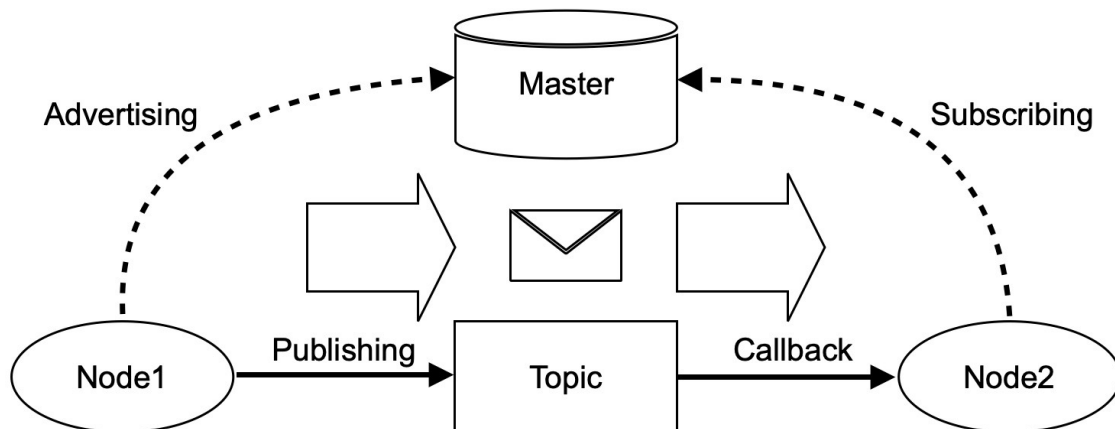


Figure 3.2: Example of computational graph in which two nodes communicate each other through a topic. First of all, nodes contact the master to communicate it that they want to advertise and subscribe to a topic, then node1 can start publishing messages on the topic and node2 receives a callback every time a new message is available on the same topic.

**Master** The ROS Master is the core of the Computation Graph which assigns a unique name to each node. It enables the communication between nodes, recording publishers and subscribers to topics and services. The Master also provides the Parameter Server.

**Parameter Server** The Parameter Server is a dictionary placed inside the Master and it provides a way to store data. As well as the Master, all nodes can contact the Parameter Server and they can save and retrieve parameters through the network. The dictionary is composed of a couple of elements: the first is the name of the parameter, that follows the ROS naming convention, and the second is the value, that can be a built-in type or a struct.

**Nodes** Nodes are processes that perform some functionalities in the system. One single node is the smallest part of the Computation Graph and many nodes

together constitute a network.

As shown in figure 3.2, nodes can communicate each other through topics, services and can contact the Parameter Server in the Master.

Each node has a unique name and it must contact the Master before being launched. If a node is launched with the same name of a running node, the latter is shut down.

Nodes are usually written using C++ or python. They should be developed and implemented so as to concentrate their work on a specific functionality. They should be independent from other nodes so that the code is modular and it can be easily integrated in a complex system.

**Messages** Messages are what nodes send and receive to communicate. As previously described in the Filesystem Level (see section 3.1.1, messages are composed of fields and each field has a type and a value.

Messages can also contain a special type: the Header, which allows the definition of some metadata as a timestamp and a frame ID.

**Topics** Topics are asynchronous unidirectional buses that connect nodes and correspond to the edges of the Computation Graph. They have a name that identifies the content of the message and a type given by the message itself. A node can use a topic as publisher in order to send messages to other nodes and can use it as a subscriber in order to receive messages from others. Nodes can publish or subscribe to many topics, moreover each node does not know if other nodes are reading or writing on the same topic.

**Services** Services implement a request/reply communication. They are synchronous bidirectional buses with a name, which represent the service, and a type, given by the service itself. Each service has a message request type, used by a client node to send a request to a server node, and a message reply type, used to receive the reply.

**Bags** Bag is a file format that ROS uses to record and playback messages that nodes exchange among them. Usually bag files are stored in the same folder (bagfiles) in a package and they are useful not only to simulate how messages are sent on topics but also for an offline study, for example to plot data or to display images.

### 3.1.3 ROS Community Level

The ROS Community Level contains all resources that allow separate communities to exchange software, code and knowledge. These resources are:

**Distributions** Distributions are releases of the Operative System. ROS has several distributions in order to let developers work on a stable system. This project

was developed using ROS Kinetic Kame. Nevertheless it is compatible with previous and future distributions.

**Repositories** Repositories are used in ROS by institutions to develop and release their own software components. In this way people can easily access them.

**ROS Wiki** ROS community Wiki is a forum in which documentation about ROS can be found. Anyone in the community can contribute to the forum adding documentation and corrections and writing tutorials.

## 3.2 rqt

The ROS platform was built to be as agnostic as possible. For example users can use topics, services or the Parameter Server to communicate data; but the architecture does not force you to use one over another nor to assign a specific name. This allows the end user to integrate ROS with other external architectures but, in order to do this, higher-level concepts are necessary.

rqt is a framework for GUI development for ROS based on Qt. It provides the rqt\_gui, which is a kind of main window, and many plugins that can be dockable in the rqt\_gui or used in a traditional standalone method. Moreover the layout can be customised and the perspective saved and restore for future uses. If users do not find the plugin that they want they can create their own.

rqt is divided in three main parts:

- rqt — Core modules
- rqt\_common\_plugins — Tools that can be used on/off of robot runtime
- rqt\_robot\_plugins — Tools for interacting with the robot at runtime

### 3.2.1 rqt\_graph

rqt\_graph is one of the most important GUI plugins from the rqt\_common\_plugins. It is used to display the ROS computation graph showing interconnection of nodes that are topics. More information like the publish frequency and the average age of the message for each topic are displayed. Moreover these two piece of information are displayed in the graph using visual elements, as shown in figure 3.3. The colour of each topic represents the age of the topic itself (red for old and green for young) and the line thickness represents the throughput (the thicker the line, the larger the throughput).



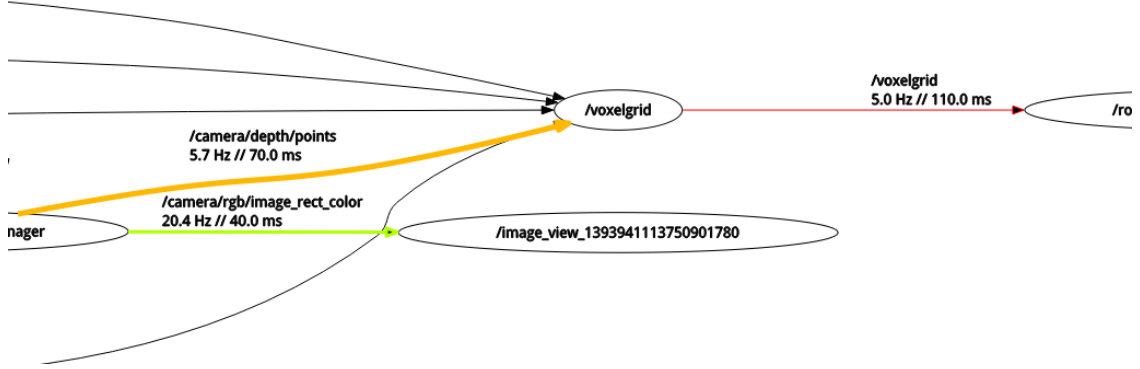


Figure 3.3: ROS graph using rqt\_graph tool with statistics [58]

### 3.2.2 rqt\_console and rqt\_logger\_level

Both rqt\_console and rqt\_logger\_level plugins come from the rqt\_common\_plugins.

The first plugin displays and filters ROS messages that are published to rosout. Figure 3.4 shows in the first box a list view in which are listed all messages collected over time and updated in real time as they arrive. For each message it is possible to retrieve the content of the message, the severity, the node which published it, the topic on which it was published, the time of the publishing and the location. In the second part of the window it is possible to filter messages by excluding or highlighting filters.

The second plugin allows the configuration of the logger level of each node. The main window (figure 3.5) shows the nodes on the left, their associated loggers in the middle and different levels on the right (Fatal, Error, Warn, Info and Debug). Logging levels are prioritised using Fatal as the highest priority and Debug as the lowest. By choosing a certain level, only messages with the same priority level or higher are visualised in the message list in the rqt\_console.

### 3.2.3 rqt\_join\_trajectory\_controller

rqt\_join\_trajectory\_controller is a GUI designed to interact with joint\_trajectory\_controller instances in an easy way (figure 3.6). It allows the user to choose the controller manager namespace, the controller itself and then, after the controller is started, it allows the user to visualise the current value of each joint in the controller and to modify it using the slider or the spinner. Moreover, using the same way of interaction, the user can choose the scaling factor.

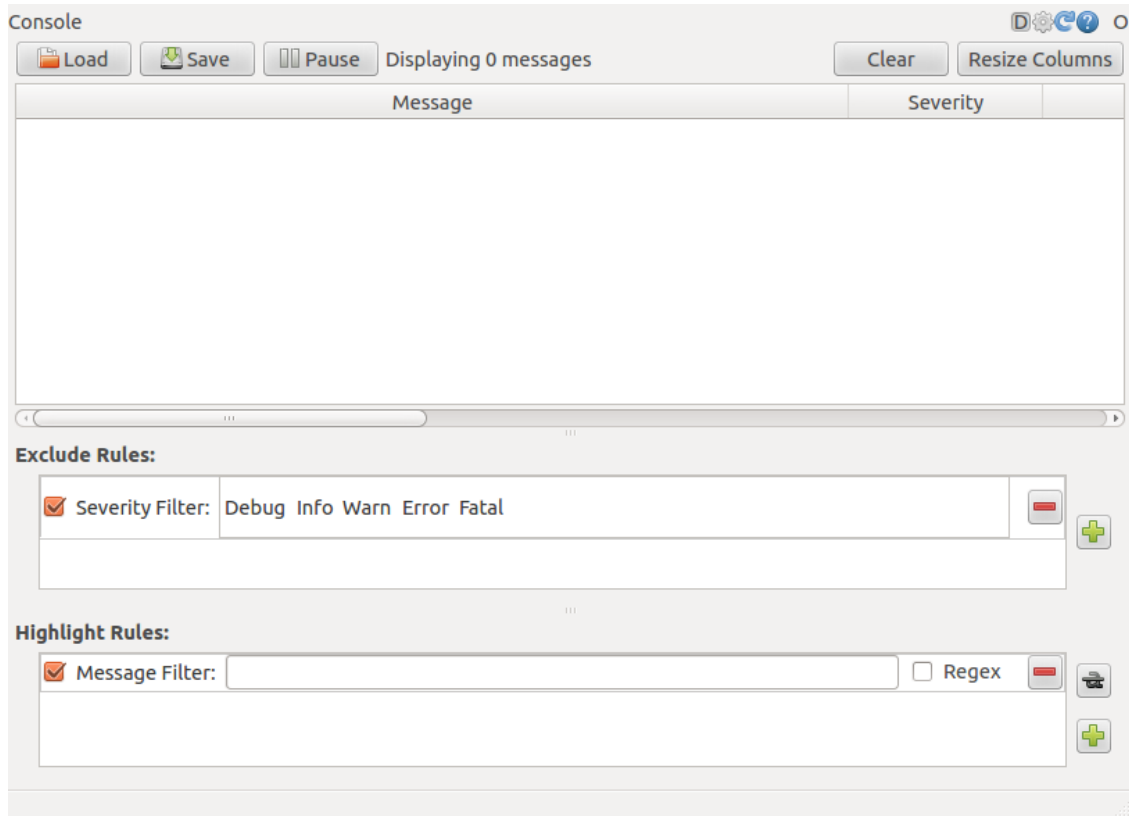


Figure 3.4: rqt\_console window [58]

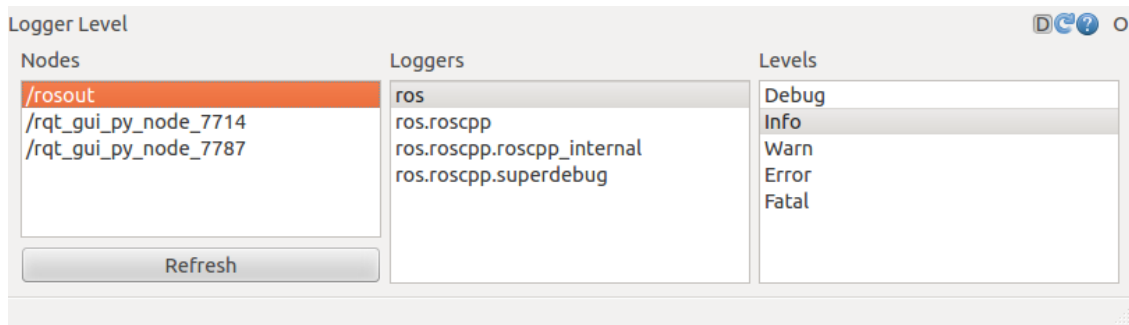


Figure 3.5: rqt\_logger\_level window [58]

### 3.2.4 rqt\_tf\_tree

**rqt\_tf\_tree** is a runtime tool which can be used to visualise the tree of frames (see section 3.4). It works as a listener of frames, which are broadcasted at that specific moment, and then it creates a tree and displays it (figure 3.7). It does not show only the relationship between a parent frame and a child frame but it also reports some diagnostic information like the broadcaster, average rate, most recent

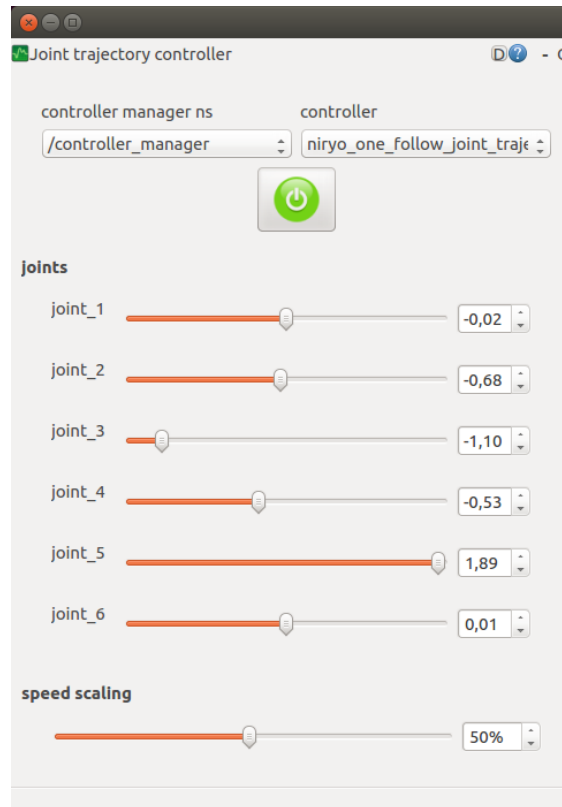


Figure 3.6: Example of `rqt_join_trajectory_controller` used to control a robot manipulator

transform and buffer length.

## 3.3 Robot Model

In a real project it is necessary to have a description of the robot system to display, control, plan, etc. This description is made using two formats to describe some characteristics as kinematics, dynamics, visual and collision of the robot itself. The first one is the URDF and it is usually used for simple models, the second one is the xacro and it is used when a robot system is complex or to make it modular.

### 3.3.1 URDF

The Unified Robot Description Format (URDF) is a model used to describe a robot and it is represented by a file written using XML. It is made to be as general as possible, thus causing some limitations: for example it can represent only robots

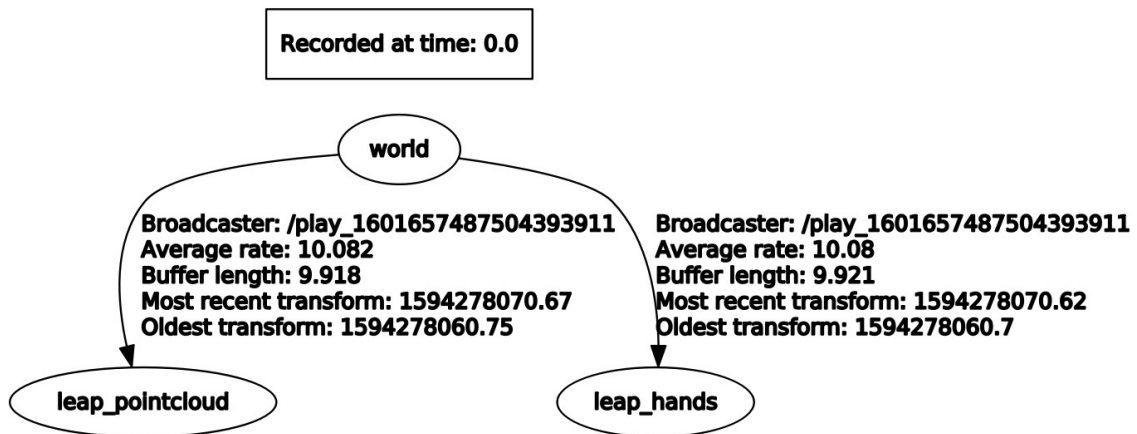


Figure 3.7: rqt\_tf\_tree window shows a tf tree with a parent frame (world) and two child frame (leap\_pointcloud and leap\_hands)

with a tree structure and rigid links connected by joints (neither parallel robots nor flexible elements can be represented). The file consists of a set of link elements and joint elements connected in an appropriate way as shown in figure 3.8.

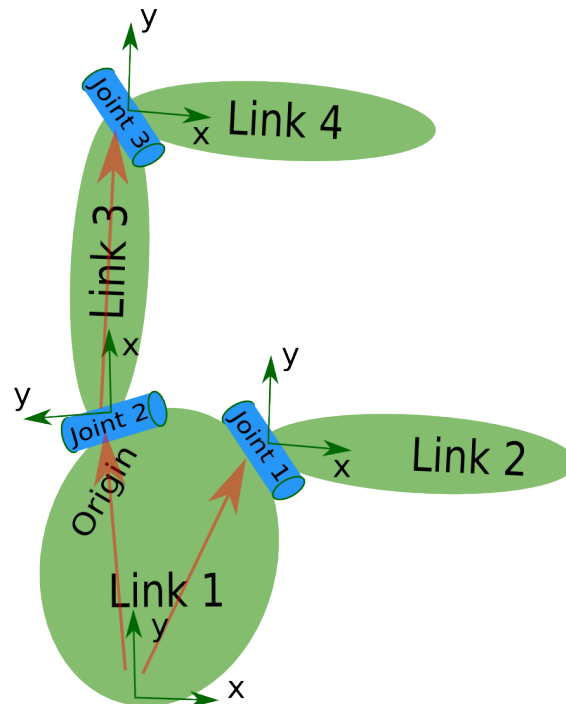


Figure 3.8: URDF structure

A link has only one mandatory attribute, the name of the link itself, and some

optional attributes such as vision, collision and inertial. Each instance of visual and collision has an optional name, the origin of the reference frame with respect to the link, the geometry (box, cylinder, sphere or a mesh file specified by a name and located in a different folder) and material (name, colour and texture). For complex elements more visual or collision instances can be used and the final instance is given by the union of all elements of geometry. Collision and visual properties can be different and the geometry of the collision property is usually simpler than the geometry of the visual property, so as to reduce the computation complexity in order to better perform the collision check. Usually in a robot description .dae files are used for the visual attribute, in order to have a robot representation as close as possible to the reality, while the .stl files or elementary geometry figures described above are used for the collision attribute. Finally, the inertial instance has the origin as specified for visual and collision properties, the mass and the inertia (represented by the 6 elements above diagonal of the inertia matrix due to it is symmetric).

A joint element has some mandatory attributes such as name, type (revolute, continuous, prismatic, fixed, floating or planar), parent link name, child link name, axis of rotation and its limits in terms of lower and upper limits, velocity and effort. Some other attributes are optional such as origin in terms of xyz and RPY, calibration in terms of rising and falling, dynamics in terms of damping and friction, mimic used to define a joint mimics another joint with a multiplier and an offset, and safety controller.

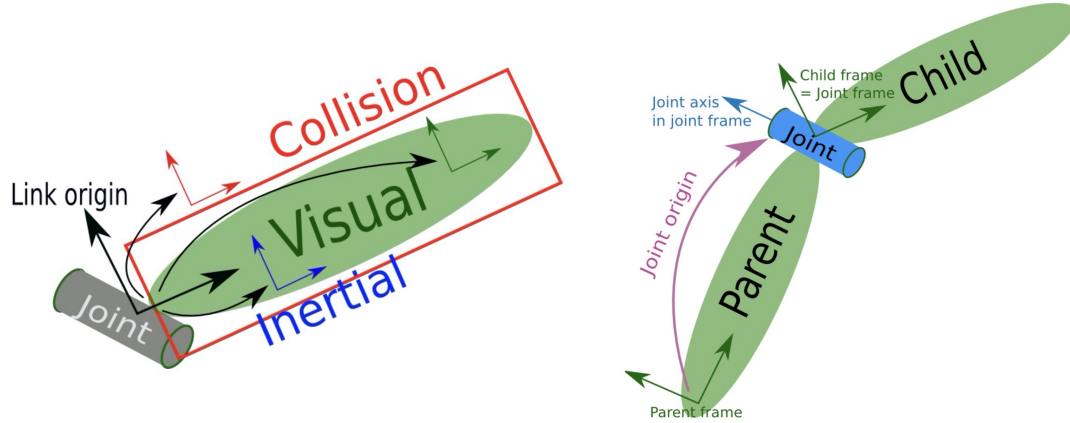


Figure 3.9: On the left, general representation of a link, on the right, general representation of a joint

### 3.3.2 Xacro

Xacro (XML Macros) is another format generally used to describe a robot structure. This format is very useful when the file is very large because it provides many

functionalities in order to make the file more compact. It is possible to define properties and property blocks using a name and a value. Properties are like variables that can be defined at the beginning of the file and then reused in any part of the code. In the same way macros can be defined using a name and a list of parameters. This format also allows the inclusion of other xacro files, insert some simple math expressions and add some conditional blocks, which are very useful when the robot can be used in different configurations.

## 3.4 Coordinate Frames and TransForms

Usually, for the user it is simpler to work inside a local frame; for this reason a new frame is defined for each single piece that composes the robot system. For example, when a sensor is added to the robot, all the data collected by this sensor are related to its coordinate system located in the centre of the sensor. Then these data need to be converted into the robot coordinate frame.

### 3.4.1 Why transforms

From an abstract point of view, a transform defines an offset in terms of translation and rotation between two coordinate frames. We can decide to manage this offset by ourselves, applying it to each couple of adjacent links or sensors, but it could be a little difficult when the number of coordinate frames increases. tf2 stands for TransForm and it is a tool that helps the user to keep track of multiple coordinate frames over time. tf2 allows the definition of a local frame for each link, sensor, etc., and it helps the user to transform points, vectors, etc., between any two coordinate frames. tf2 is also able to maintain the relationship of different coordinate frames in an oriented graph without loops, represented by a tree structure as shown in figure 3.10. Each node in the tree represents a frame. Each frame has only one single parent but it can have many children. Each tree has one single root usually called world frame. Each edge represents the transform that needs to be applied to move from the parent coordinate frame to the child coordinate frame.

Usually a robot system has several 3D coordinate frames changing over time. Considering a manipulator arm each link has a coordinate frame, from the base to the end effector, moreover there usually exists a fixed frame (world frame) used as reference and a coordinate frame for the end effector. tf2 helps the user to know, for example, not only the current pose of the gripper relative to the base or relative to the world but also the previous poses of a certain link. In each transform the direction of conversion is specified. It transforms the coordinate frame “frame\_id” into “child\_frame\_id”. This is the same transform which will take data from “child\_frame\_id” into “frame\_id”. This tool is very simple to use because its high level allows the users to compute transforms, only knowing the name of each “frame\_id”, that needs to be globally unique. In addition it is

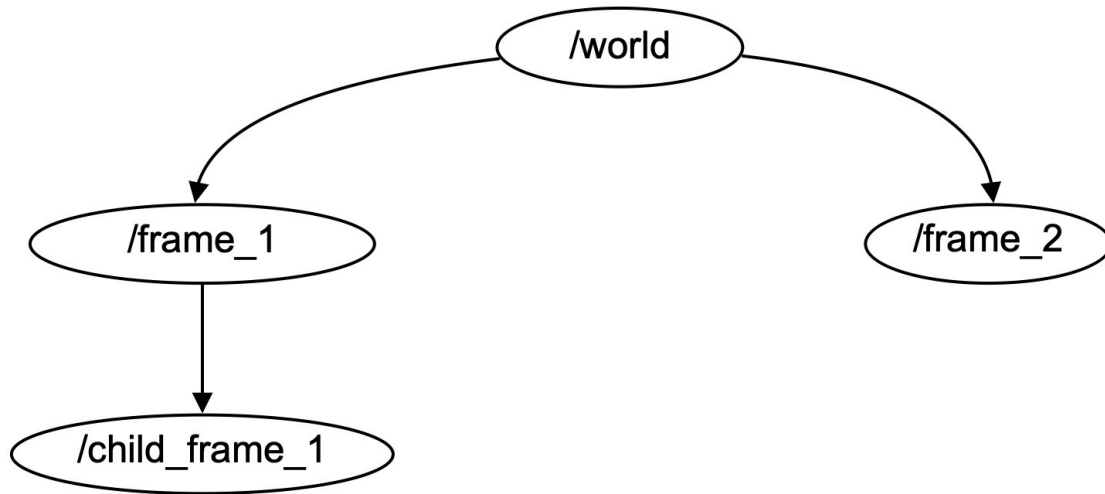


Figure 3.10: Generic example of a transform tree

efficient to use because it transforms data between coordinate frames only at the time of use and it also works well in a distributed system because all the coordinate frames are available in all the components of the system.

Its main functionalities are listening for transforms and broadcasting transforms. The first means receiving and storing in a buffer (up to 10 seconds) all coordinate frames and querying for specific transforms, the latter stands for publishing the relative pose of coordinate frames. A system can have several broadcasters, each one publishing information about a different part of the robot.

### 3.4.2 `static_transform_publisher` and `robot_state_publisher`

Some objects attached to the robot do not change their own position in time so it is not needed to broadcast the transform. `static_transform_publisher` comes to our help in order to not communicate again and again things which do not change in time. Compared to a regular transform, the biggest difference is that it does not keep a time history.

In a real development process an executable program (`static_transform_publisher`), provided by the `tf2_ros` package, can be used as a commandline tool or as a node in a launch file, indicating the offset in terms of pose.

`robot_state_publisher`, instead, is an executable that publishes the state of the robot to tf. This package uses the URDF (XML format) and the publish rate stored in the Parameter Server, respectively under the name “`robot_description`” and “`publish_frequency`”, and the position of each joint, shared in the topic `joint_states`. By doing this, it has all information about the structure of the robot and current position of the joints in order to compute the forward kinematics of the robot and

to publish it over tf (figure 3.11).

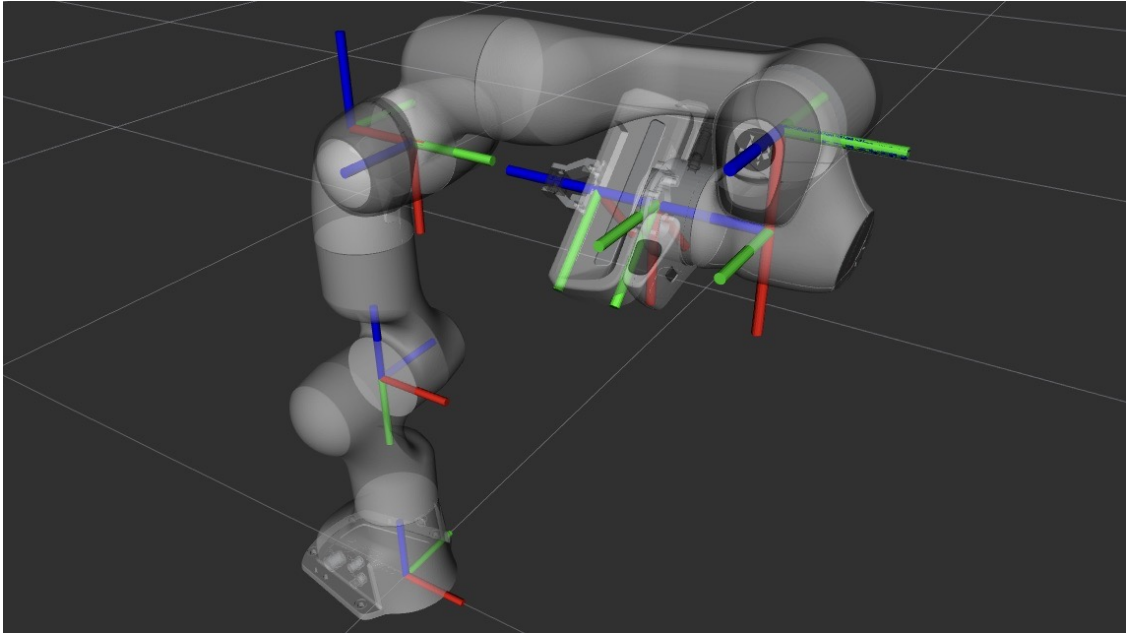


Figure 3.11: Example of a generic robotic arm using `robot_state_publisher` to calculate the coordinate frame of each link

## 3.5 MoveIt

MoveIt is an open source robotics manipulation platform [47]. It is used to generate high-degree of freedom trajectories using motion planning, analyze and interact with the environment with grasp generation, solve the inverse kinematics for a given pose, control, have a 3D perception of the world and collision checking. Users can interact with MoveIt by programming or using a plugin interface in Rviz (figure 3.12) and it is easily integrable with Gazebo (see section 3.8) and ROS Control to have a powerful robotics development platform.

### 3.5.1 Architecture

The primary node provided by MoveIt is `move_group`. This is a ROS node capable of interacting with other nodes in the ROS environment. In figure 3.13 it is possible to see all interconnections between the `move_group` node and the system.

This node mainly interacts with the User Interface. The User Interface allows the user to send actions and services to the `move_group` node and let MoveIt do its job. This interaction can be done using C++ or Python interfaces or through the GUI on Rviz.



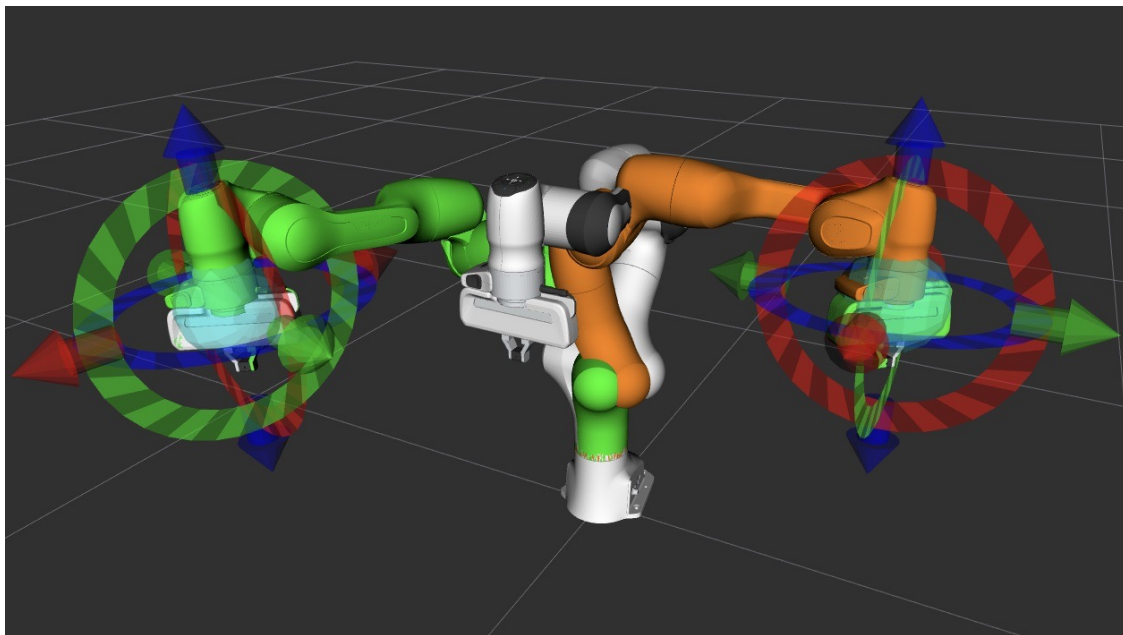


Figure 3.12: Example of interaction with MoveIt using the Rviz plugin. Adding the MotionPlanning display to Rviz, the user can choose the initial pose of the robot (green) and the final pose (orange).

The `move_group` node interacts also with the Param Server. In this way the main node can retrieve all information about the robot system: the URDF from the `robot_description` parameter, the SRDF from the `robot_description_semantic` parameter and other MoveIt configurations like joint limits, kinematics, motion planning, perception etc. Configuration files and SRDF files are generated automatically from the MoveIt Setup Assistant when the user setups the robot to work with MoveIt.

Finally, the `move_group` node interacts with the robot using topics and actions to retrieve information about the current robot state (joints position from `/joint_states` topic) and data collected from sensors.

### 3.5.2 Planning Scene

MoveIt uses the Planning Scene Monitor (inside the `move_group` node) to create the Planning Scene. It is an internal representation of the world and the robot itself. As described in figure 3.14, the Planning Scene Monitor uses:

- the scene monitor to retrieve information about the current joint positions and to reconstruct the robot state (the robot state can include also objects that can be considered rigidly attached to the robot);

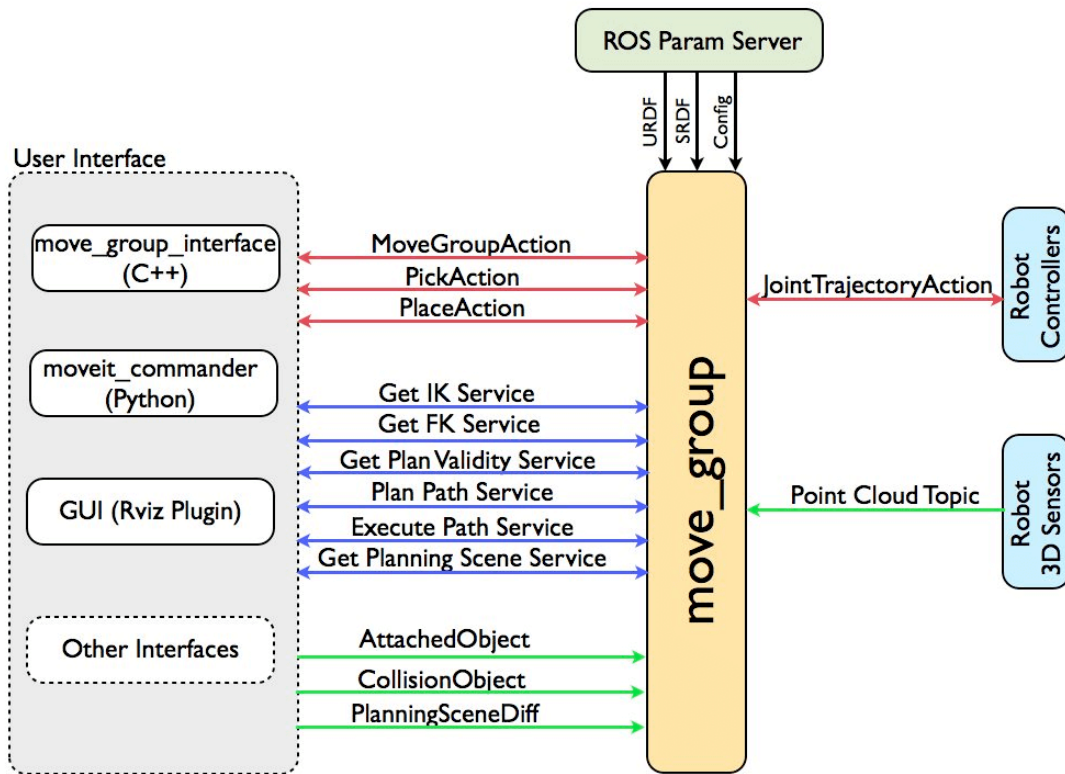


Figure 3.13: Interconnection between the `move_group` node by MoveIt and other parts of the system [47]

- the state monitor to retrieve information about the data collected from the robot sensors;
- the world geometry monitor to build world geometry using information about the robot, sensors and user input, listening to the `planning_scene` topic, to add object information.

In order to do this, the world geometry monitor uses the occupancy map monitor to construct an octomap that is a 3D occupancy grid map based on an octree designed to generate an updatable full 3D model in a flexible and compact way.

### 3.5.3 Motion Planning

MoveIt interacts with a motion planner through a plugin interface. This makes MoveIt easily extensible with different motion planners from multiple libraries. The default motion planners used by MoveIt are configured using OMPL (Open Motion Planning Library), which implements randomised motion planners.

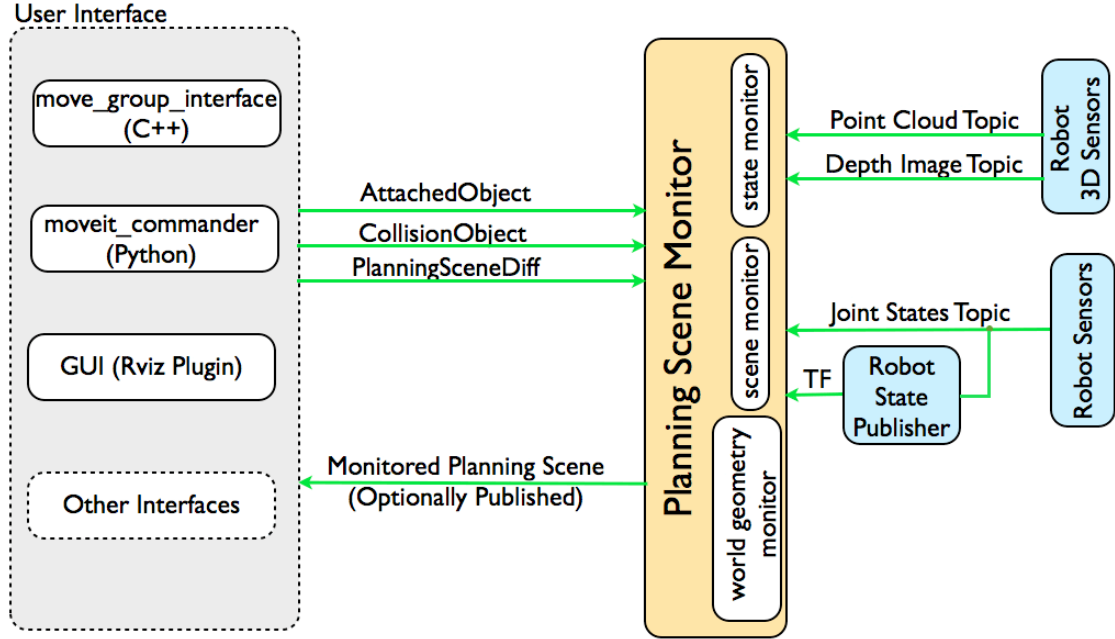


Figure 3.14: MoveIt Planning Scene Monitor and its interconnections [47]

Figure 3.15 shows the entire flow from motion plan request, as input, to motion plan response, as output.

The motion plan request usually asks the robotic arm to move to a specific location in joint space or to a specific pose of the end-effector. Moreover some kinematic constraints can be specified, like the position and orientation constraints of a link, joint constraints or more complex customized constraints by the user. Collisions and self collisions are checked by default. If an object is attached to the end-effector (or any part of the robot) MoveIt takes care of it and makes collision checking and planning considering the object as a momentary part of the robot.

The motion plan response contains the trajectory (not simply a path) that moves the robotic arm to the desired pose. `move_group` uses the maximum velocity and acceleration if specified.

Before the request arrives to the motion planner and before the final motion plan response is generated, there are some planning adapters that carry out pre-processing motion plan request and post-processing motion plan response. Pre-processing is used, for example, to fix the start state bounds, when the joint positions is slightly outside the joint limits, then joints are moved to the joint limits. Another example is when the start state is in a collision state then joints are perturbed using a small amount such as to find a collision free state. Post-processing is instead used for example to generate a time-parameterized trajectory, applying velocity and acceleration constraints. This is because the “kinetic path” generated by the motion

planner does not obey any velocity or acceleration constraints and it is not time parameterized.

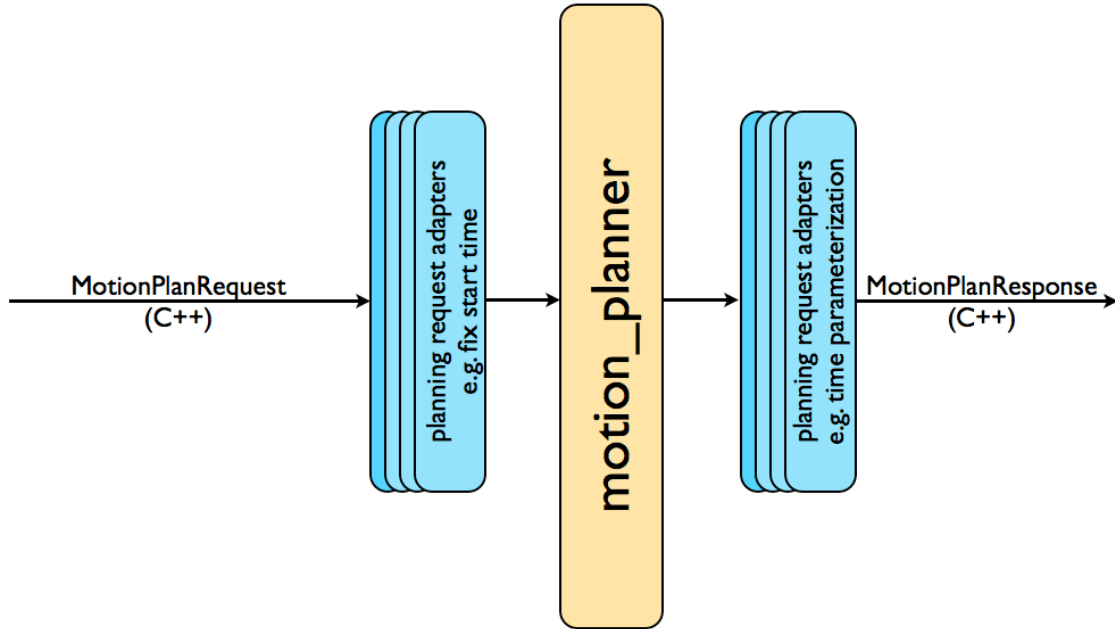


Figure 3.15: MoveIt Motion Planning pipeline [47]

### 3.5.4 Pick and place

MoveIt website provides many tutorials to better explain the MoveIt package behaviour. One of the most used feature is the pick and place functionality. It is performed using `PlanningSceneInterface` and `MoveGroupInterface`. The executable is composed by three main functions: `addCollisionObjects`, `pick` and `place`.

The `addCollisionObjects` function uses the `PlanningSceneInterface`. Through this interface it is possible to add collision object to the Planning Scene using a vector of messages of type `moveit_msgs/CollisionObject`. Each object has a unique identifier, a type, which can be a primitive shape, a mesh file or a plane, and its pose.

The `pick` function uses the `MoveGroupInterface` to call the `MoveGroupInterface::pick` function with the object id and a vector of messages of type `moveit_msgs/Grasp`, which defines the possible ways to grasp the object. The grasp message is defined by five relevant fields:

- `pre_grasp_approach` defines the direction the end effector has to follow and the distance from which to approach the object;

- `pre_grasp_posture` defines the trajectory position of the joints of the end effector before the grasp;
- `grasp_pose` defines the pose of the end effector during the grasp;
- `grasp_posture` defines the trajectory position of the joints of the end effector for grasping the object;
- `post_grasp_retreat` define the direction the end effector has to follow and the distance to travel after the object is grasped.

The `place` function uses the `MoveGroupInterface` to call the `MoveGroupInterface::place` function with the object id and a vector of messages of type `move-it_msgs/PlaceLocation`, which defines the possible ways to place the object. As in the case of the grasp message, the place message is defined by four relevant fields: `pre_place_approach`, `place_pose`, `post_place_posture` and `post_place_retreat`. In this case the pose is defined with respect to the center of the object.

## 3.6 OpenCV

Open Source Computer Vision Library (OpenCV) is an open source computer vision and machine learning software library [52]. It was built in order to have a common infrastructure for computer vision applications. It has a lot of algorithms (more than 2500) based on state-of-the-art computer vision and machine learning algorithms. They can be used to detect and recognise faces and objects, classify human actions and track objects movements in a video, extract 3D models of objects, produce 3D point clouds when a stereo camera is available, find similar images from an image database, follow eye movements, add virtual markers using Augmented Reality, etc. OpenCV was implemented for Windows, Linux, Android and Mac OS and it provides C++, Python, Java and MATLAB interfaces to develop OpenCV applications.

From ROS Kinetic Kame, OpenCV3 can be used by default. To do this, the package `vision_opencv` is included in the ROS distribution and it allows the user to use OpenCV as it were outside ROS [34]. It contains `cv_bridge` and `image_geometry` packages. The first is used as an interface between ROS messages and OpenCV (figure 3.16), the second provides a lot of functions to manipulate images and pixels.

### 3.6.1 `find_object_2d`

`find_object_2d` [21] is a package which corresponds to Find-Object application [40] in ROS. It uses OpenCV and an inexpensive webcam to implement SIFT, SURF, FAST, BRIEF and other feature detectors and descriptors to detect and recognise images from a database.

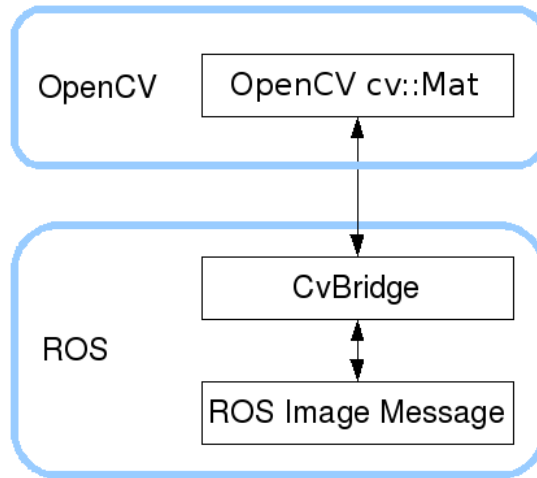


Figure 3.16: Graphical representation of the interconnection between ROS and OpenCV using `cv_bridge` [58]

The executable listens the topic in which the camera images are published, detects and recognises objects and then publishes `DetectionInfo` and `ObjectsStamped` message on two topics. Both message types contain the ID of detected objects, their position in the image in pixels and other information.

Using the Qt based GUI (figure 3.17), the user can add some objects the executable has to recognise, loading them from a folder or taking new photos to the current video, and change some parameters at runtime [35].

## 3.7 Rviz

Rviz is a 3D visualiser for ROS. The main window (figure 3.18) is divided into three main parts: on the left, a list of displays and their own properties, the 3D view in the middle and, on the right, a list of views.

### 3.7.1 Displays

Displays are what is shown in the 3D view. A display has some properties such as the status (OK, Warning, Error and Disabled) with a short explanation, in order to let the user understand if it is behaving in the correct way, and the subscribed topic where data are taken from. There is a very long list of different kinds of displays and each display has its own kind of properties. The most common built-in displays types are:

**Grid** This display shows a 2D or 3D grid of lines along a plane, centered at the

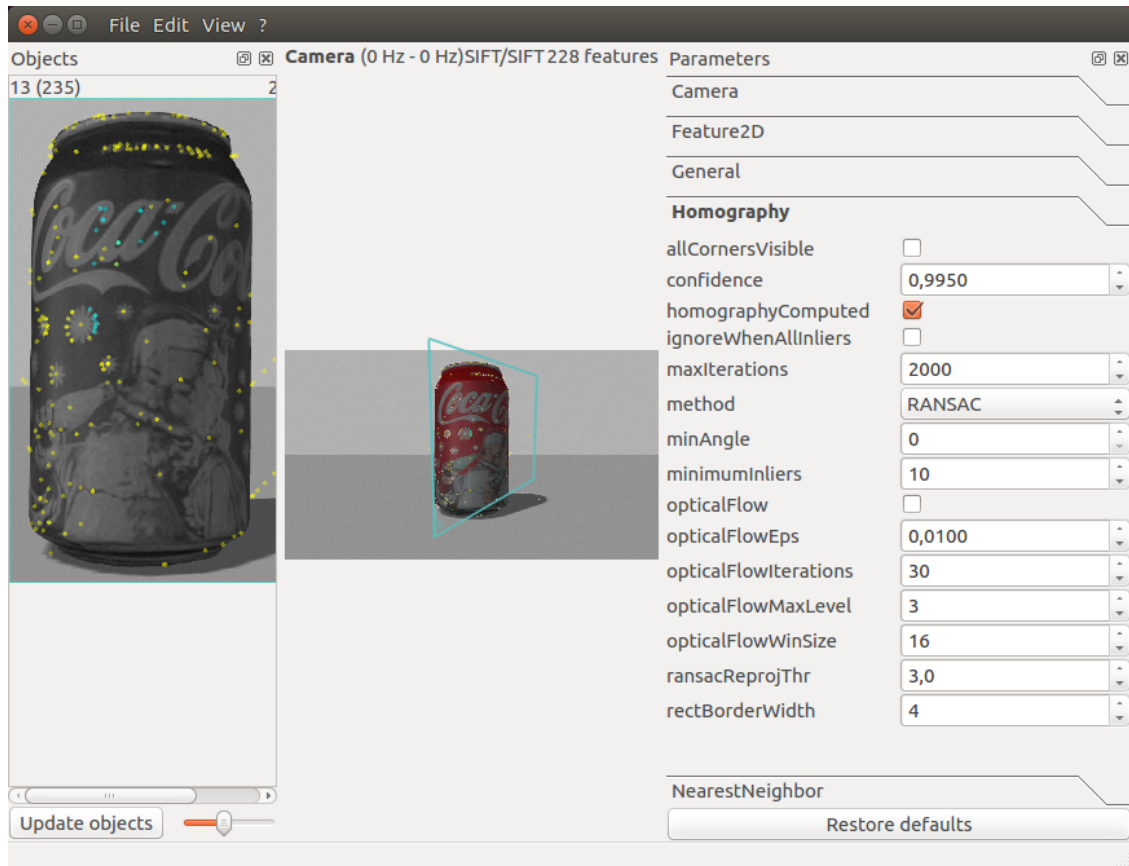


Figure 3.17: Example of object recognition. on the left, the object to recognise which detectors and descriptors (yellow and light blue points), in the middle there is the image from the camera and the detected object has a square around it, on the right, the list of parameters.

origin of the target frame. It is generally used to represent the XY plane as a surface on which to locate the robot. The Grid display is included by default when a new Rviz window is opened.

**RobotModel** This kind of display shows the pose of each link and reconstructs the entire robot. Links information (visual and collision) is given by the `robot_description` parameter in the Parameter Server and it retrieves the position of each joint using the tf tree. The model is updated with a given rate.

**TF** This kind of display shows the tf transform tree published by the `robot_state_publisher` node (see section 3.4.2). It allows to display the frame name, the frame axes and the arrow from the child frame to the parent frame. Axes are indicated: in red the X axis, in green the Y axis and in blue the Z axis. The fixed frame is always available (usually world frame) and it is



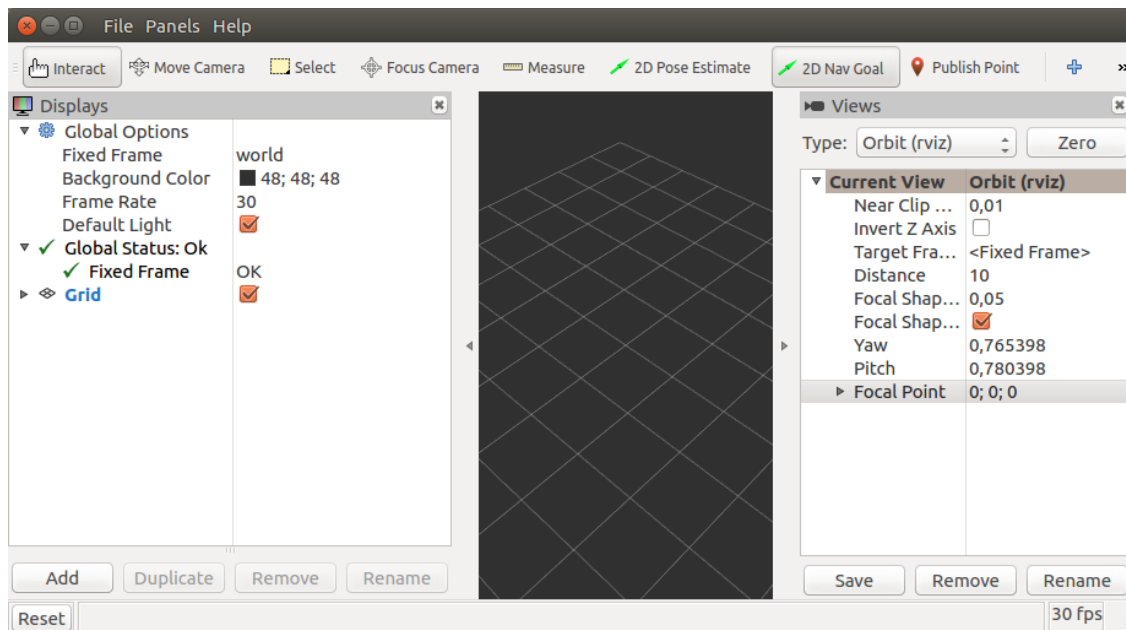


Figure 3.18: Rviz main window with the displays section on the left, views section on the right and the 3D view in the middle

located in the center of the coordinate system.

**Image** This kind of display generates a new display window with an image inside. This image is updated with a given rate and it takes information from the `sensor_msgs/Image` topic. This kind of display is generally used to show images coming from a camera.

**Marker** This display adds a primitive shape to the 3D view, reading information from the `visualization_msgs/Marker` message on `visualization_marker` topic. The available shapes are arrow, cube, sphere, cylinder, line strip, line list, cube list, sphere list, points, view-oriented text, mesh resource and triangle list. In case of a single object, it is displayed using the center of the object or using the start and end point (only in the case of the arrow), while in case of list objects a set of points is used. Obviously the visualisation of a single marker is less expensive than many markers, so it is better to use an object list when many shapes of the same type have to be displayed.

When this kind of display is created, it automatically subscribes to the same topic with extension “`_array`” using `visualization_msgs/MarkerArray` messages. With this kind of messages it is possible to display many markers at once.

**PointCloud2** This kind of display is used to show data from `sensor_msgs/PointCloud2` messages in order to generate some shapes in the 3D view, according to



the depth generated from a sensor, for example, a depth camera. This display offers four different rendering styles (points, billboards, billboard spheres and boxes), giving the possibility to choose the best one that fits the application.

If the users do not find a desired display in the provided list they can use a plugin to add a new customised display with some specific properties so as to make it behave as they want.

### 3.7.2 View

Views are different types of camera available in the visualiser. Each camera has a different way to be controlled and a different type of projection (Orthographic or Perspective). The different types of cameras are:

**Orbital Camera** This camera rotates around a focal point that can be everywhere in the space.

**XY Orbit** This camera has the same behaviour and the same way of controlling the Orbital Camera but the focus point is restricted to be in the XY plane.

**Third Person Follower** This camera maintains a constant viewing angle towards the tangent frame but it turns if the tangent frame yaws.

**FPS (first-person) Camera** This camera gives you a first-person perspective. Rotating the camera is like rotating one's head up and down or left and right. Zooming in and out is like moving forwards and backwards.

**Top-down Orthographic** This camera provides a top to bottom perspective, along the Z axis in the robot frame using an orthographic view therefore if the camera moves farther the object does not get smaller.

Given a certain type of camera, its pose and the target frame, it is possible to save the view in the view panel and use it in the future.

Many different configurations are possible using displays and views. A configuration is made up of displays and their properties, tool properties and a camera type with its settings. It can be saved in a folder and reused again. It can be simply loaded from the Rviz windows or it is generally used inside a launch file.

When a display is used, rviz uses the tf to transform data from the coordinate frame into the global reference frame. There are two very important reference frames: the fixed frame and the target frame. The former is used to identify the world, in fact it is usually called “world”, it is fixed and it does not move. The latter is the frame for the camera view. For example, considering a mobile robot, if the target frame is the map, you will see the robot moving around, conversely if the target frame is the robot, you will see the robot fixed and the map moving with respect to it.

## 3.8 Gazebo

Gazebo is a free simulator which enables the user to design robots and simulate them using a realistic scenario [25]. It can be used from the command line or using a graphical interface, it has a robust physics engine and high quality graphics. It is also able to generate sensor data optionally with noise, and users can also write plugins for robots, sensors and environments.

### 3.8.1 Components and SDFFormat

The simulator is composed of two executables: the Gazebo Server and the Graphical Client. The first is the core of the simulator, it simulates the world using a physics and sensor engine without any graphical interface, the latter is a visualiser based on Qt user interface which is connected with the server and displays elements (figure 3.19). The running simulation can also be controlled and modified. When the Gazebo Server runs, some files are uploaded. These files are formatted using SDF [63]. SDF (Simulation Description Format) is an XML file which can be used to describe objects and environments for robot simulation, visualisation and control. It gives the possibility to accurately describe static and dynamic objects, any kind of robot not only using kinematics and dynamics but also using sensors and surface properties as textures, friction etc.

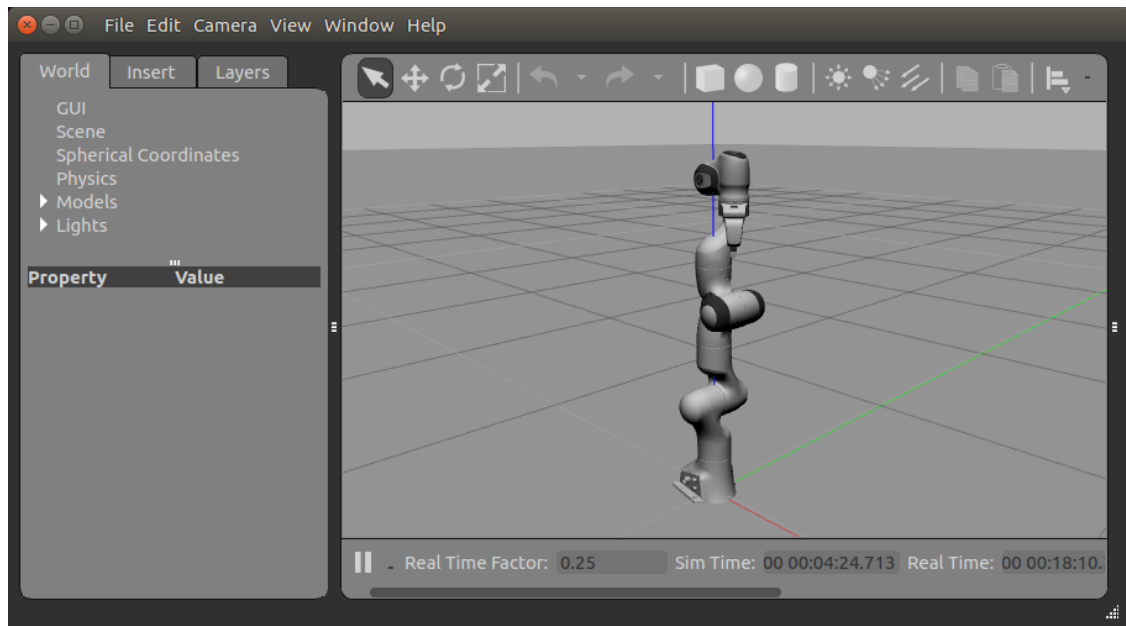


Figure 3.19: General robotic arm simulation using Gazebo Server and the Graphical Client

There are four different types of elements and all of them can be represented by

an SDF file:

**world** The world element describes the whole environment and it can contain models, actors, light and plugins. It is used by the Gazebo Server to populate an environment.

**model** The model elements describe a robot or a sensor or any other physical object. An SDF file can contain only one single model but many models can be included in a world. This is mainly done for code reuse and to reduce the dimension of world files. Models can be dynamically loaded into simulation either programmatically or through the GUI. They can be downloaded from the repository or created by the user.

**actor** The actor element is a special kind of model which can have a scripted motion that can be global waypoint type animations and skeleton animations.

**light** The light element describes a light source.

### 3.8.2 ROS integration

In order to use stand-alone Gazebo with ROS a set of packages named `gazebo_ros_pkgs` need to be downloaded. These packages work as a wrapper between the two systems in order to let them communicate with each other, as shown in figure 3.20. In addition these packages have the main characteristic to treat an URDF file as close as possible to a SDF file and improve controllers using `ros_control`.

An URDF file can only specify the kinematic and the dynamic characteristic of a robot, moreover it can only describe things that are robots, and not sensors, lights and so on. For this reason the SDF was developed and it was made in order to reuse more code as possible from the URDF files.

First of all Gazebo needs to know some physical properties to better simulate the environment, so the optional inertial parameters in the URDF file become mandatory for each link. The mass, the center of mass and the moment of inertia matrix must be specified. Since the 3x3 inertia matrix is symmetric positive-definite, with 3 diagonal elements and 3 unique off-diagonal elements, only 6 numbers are sufficient to represent it. Additionally, with the URDF it is not possible to specify the position of the robot with respect to the environment. To solve this problem a new link, usually defined as “`world_link`”, is added and the base of the robot is connected with the world link using a fixed joint.

Secondly, Gazebo needs to know more information about the controllers in order to simulate the hardware interfaces. The `ros_control` package receives as input the joint state and use a generic control loop feedback (PID controller) to move the robot.

Some controllers are available in the `ros_controllers` package:

GAZEBO + ROS + ros\_control

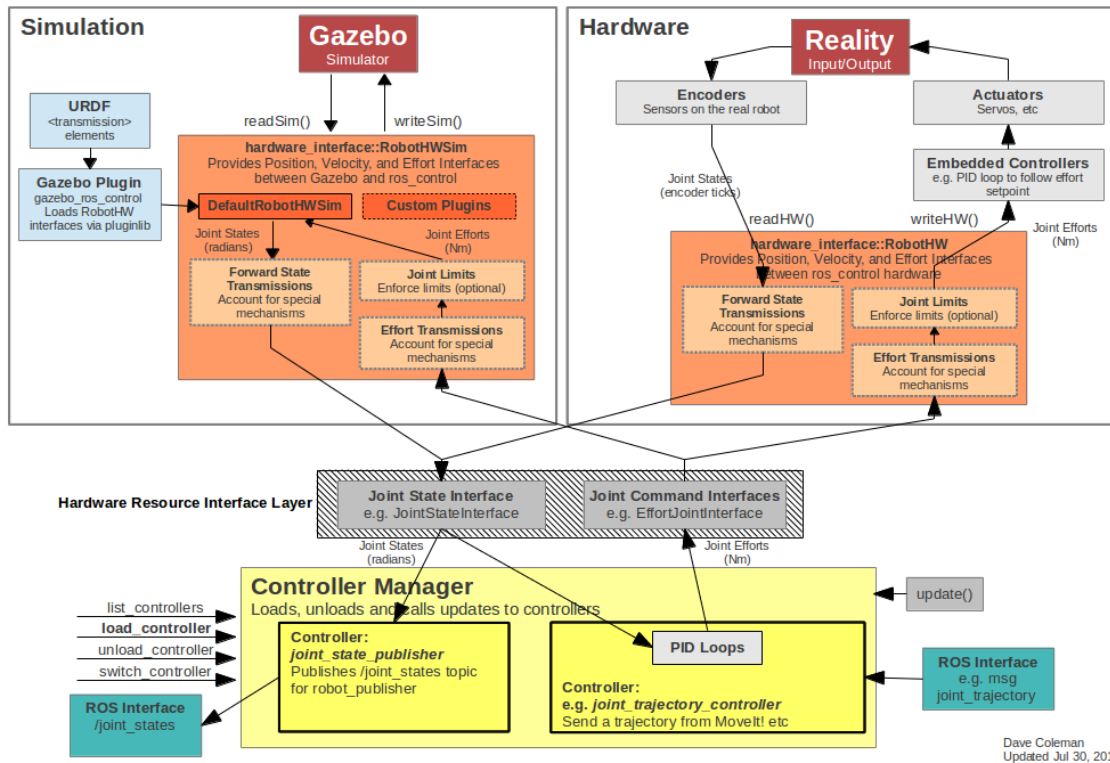


Figure 3.20: Interconnection between Gazebo and ROS using `ros_control` package [25]

- `joint_state_controller` — Publishes the state of all joints.
- `position_controllers` — Set one or multiple joint positions at once.
- `velocity_controllers` — Set one or multiple joint velocities at once.
- `effort_controllers` — Command a desired force/torque to joints.
- `joint_trajectory_controllers` — Extra functionality for splining an entire trajectory.

If the above list of controllers does not satisfy the user requirement, they can autonomously create the one which better fits their needs.

In parallel with the controllers, ROS needs some hardware interfaces in order to read and send commands to the hardware. The most important hardware interfaces are:

- **Joint State Interface** — Used for reading the state of an array of named joints.

- Joint Command Interface — Used for commanding an array of joints.
- Actuator State Interfaces — Used for reading the state of an array of named actuators.
- Actuator Command Interfaces — Used for commanding an array of actuators.

Also in this case users can create their own hardware interface if they do not find the right one in the provided list.

Finally a transmission tag needs to be added for each link in the URDF file. Transmission is the element that interconnects an actuator to a joint. It transforms the force or the velocity between the two components maintaining their product (power) constant.

Once all hardware is specified, Gazebo is ready to simulate the defined system. To do this, the `gazebo_ros_control` plugin is added to the URDF in order to load the appropriate hardware interfaces and controller manager [37]. After this procedure the simulated robot system can be treated as a real robotic system.



# Chapter 4

## Hardware

### 4.1 Leap Motion

Leap Motion Controller is an optical hand tracking module, developed by Ultraleap, able to capture the movement of your hands and fingers [41]. It is a very small sensor (80 mm x 30 mm x 11.3 mm), and it is simple but also fast and accurate. It contains three IR LEDs that illuminate the interaction zone with a frequency of 100 times a second using infrared light, which is not visible to humans but visible to its two IR cameras which capture images and send data back to the computer via USB. The interaction zone goes from 10 cm up to 80 cm (preferred 60 cm) in a 120°x 150° field of view.

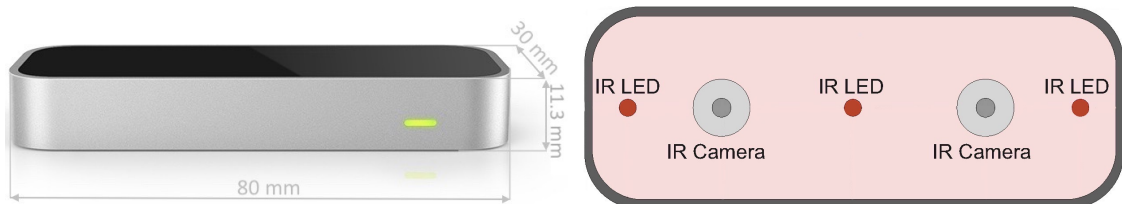


Figure 4.1: On the left, the Leap Motion controller dimensions are shown, on the right, a representation of IR LEDs and IR cameras

The software works in a powerful and robust way to reconstruct the hand shapes and generate a virtual model of the hands from images. It recognises 27 distinct hand elements among bones and joints, even when some pieces are hidden, from the controller point of view. Moreover it is able to track the complexity of natural movements in a smooth way thanks to its high frame rate. Cameras work at 120 Hz and they are capable of capturing an image within 1/2000th of a second. Thanks

to its good qualities, the software processes images with a very low latency, which is not perceptible to the human brain.

### 4.1.1 Architecture

The Leap Motion software runs as daemon on Linux. The Leap Motion service, as shown in figure 4.2, receives data from the Leap Motion controller via USB. It processes data and sends them to the Leap-enabled application, which can be one at a time. The Leap Motion SDK provides two kinds of API for getting Leap Motion tracking data: a WebSocket interface and a native interface. The native interface is a dynamic library which the user can use to create new Leap-enabled applications. When a Leap-enabled application is in foreground it can connect the Leap Motion service using the Leap motion native library to receive tracking data, while, if the Leap-enabled application is in background, it stops receiving data from the Leap Motion service, unless the configuration settings are configured differently. From the Leap setting app, which runs separately from the service, the user can configure the Leap Motion installation.

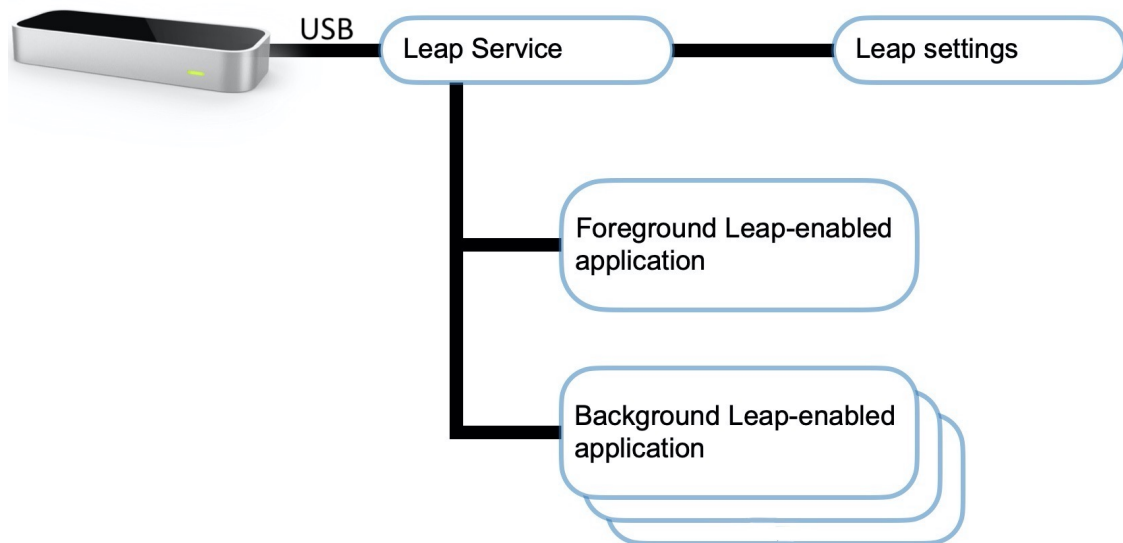


Figure 4.2: Leap Motion architecture for native application interfaces

### 4.1.2 Coordinate frame

The Leap Motion uses a right-handed cartesian coordinate system (figure 4.3). The origin of this system is in the centre of the top of the Leap Motion controller.



Considering the Leap Motion controller on a desk with the user on one side and the computer on the other side, the horizontal plane is represented by the x-z plane with the positive z axis pointing to the user and the positive x axis pointing to the right. The y axis is consequently vertical and its value increases going upwards. By using this coordinate frame, only positive values can be obtained from the y axis. Moreover, as shown in figure 4.3, the field of view is an inverted pyramid, so the available range on the x and z axes is very small when closer to the device while it increases when going farther. The consequence is that users need to maintain their hands in the field of view if they want the controller to properly track them.

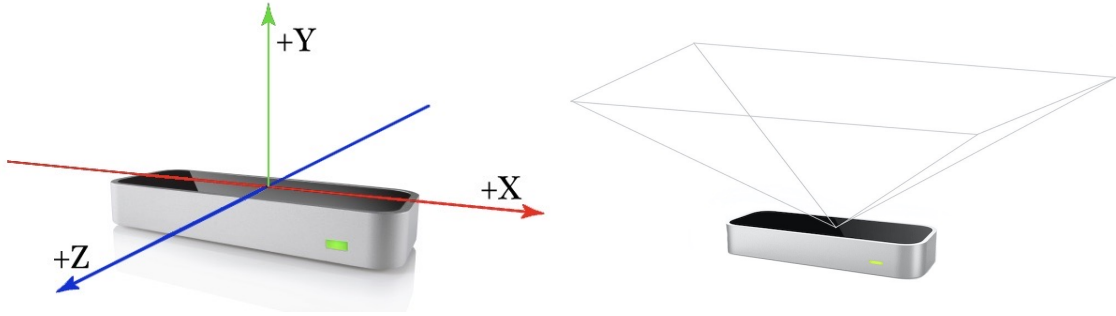


Figure 4.3: On the left, the representation of the Leap Motion coordinate system [41], on the right, the field of view represented by the inverted pyramid

### 4.1.3 Tracking model

The Leap Motion API provides a class for each tracked object. The tracking model is organised using a tree structure in which the root is given by the Frame class and the other classes are a specialisation or a component of the above class (figure 4.4). A new Frame object is created at each update interval and from this class the user can access all other tracked elements. It contains classes of the “-List” type, which allow the user to retrieve many elements all at once and other functions for filtering. These “-List” classes contain classes of the Image, Hand, Pointable, Finger, Tool and Gesture types. Moreover there are some classes, like Vector and Matrix classes, which are not included in the tracking model but which provide several useful mathematical functions to manipulate data, for example, directions and normals.

A Frame object contains a snapshot of the scene recorded by cameras. Frames can be retrieved in two ways: by polling and with callbacks. The first method is the simplest and works very well when the application has a constant frame rate. At each iteration a frame is taken from the controller but, in this situation, if the application frame rate is greater than the Leap Motion controller frame rate, the same frame is taken twice; conversely, if the application frame rate is lower, some

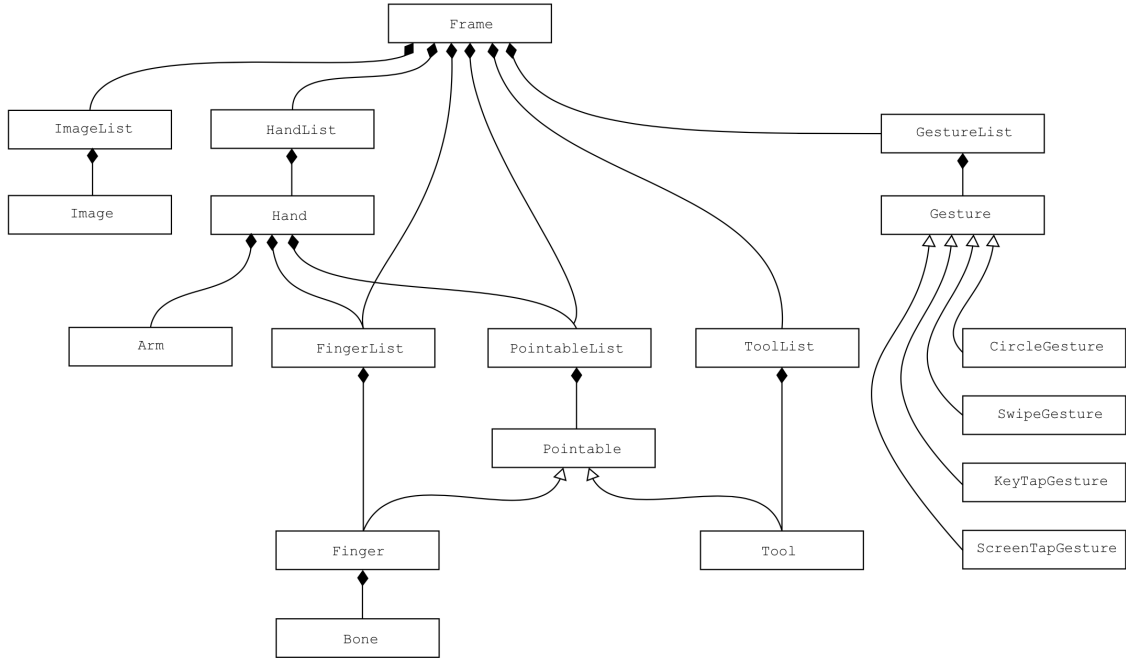


Figure 4.4: Leap Motion tracking model [41]

frames are skipped. The second method uses listener callbacks, so the controller calls a function each time a new frame is received. Callbacks are more complex because they are multi-thread and each callback is invoked on an independent thread, so the user must ensure data access is made in a thread-safe manner. Also in this case some frames can be skipped: if the callback takes too much time to complete its process, the frame is added to the history; if the computer has many tasks to run, instead, the Leap Motion software may not finish to process some frames in time and these frames are discarded.

Hands are the main element tracked by the Leap Motion controller. More than one hand can be tracked but the quality decreases if more than two hands are present in the scene. The Leap Motion software has an inner model of the human hand and it uses this model to compare data received from sensors and validate them. This also allows the software to reconstruct the position of fingers even when they are not completely visible, for example, when a finger is behind the palm, from the controller point of view. From a **Hand** object it is possible to access its information like palm position, direction, norm, etc. but it is also possible to access the **Arm** object and **Pointable** objects (fingers).

An **Arm** object can be accessed only from a **Hand** object. It provides information like orientation, length, width, and end points of an arm. If the elbow is not visible by the controller, its position is estimated based on past observations as well as typical human proportions.

Pointable objects are Finger and Tool objects. They have some common properties deriving from the Pointable class and other more specific properties deriving from their own specialised class. A Tool object is represented by a cylinder object longer and thinner than a human finger; instead, a Finger object is more complex.

A Finger object has a type (thumb, index, middle, ring and pinky), other properties, like the direction, the length, the width, etc. and a list of bones. As shown in figure 4.5, all fingers have four bones. Even if a real thumb has three bones, for a practical realisation, the metacarpal is present but has zero length.

Each bone is represented by an object of the Bone class. Also in this case it has a type (metacarpal, proximal phalanx, intermediate phalanx and distal phalanx) and other properties like the position, the direction, etc.

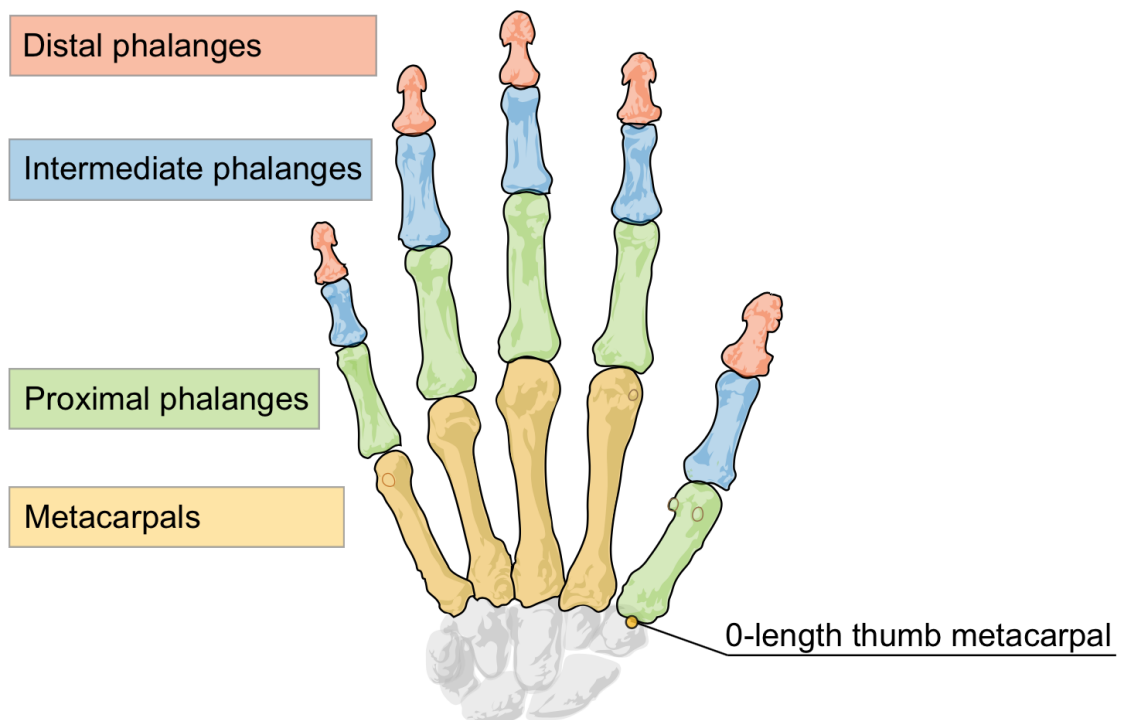


Figure 4.5: Leap Motion hand model

A Gesture object can be accessed directly from a Frame object and it represents a specific movement of the hand. When the software recognises that a gesture is present, a new object is created and added to the Frame object. If a gesture continues in time, it is updated and inserted in future Frame objects. A Gesture object contains the type of gesture, the duration and other information. Leap Motion recognises four gestures (figure 4.6):

- CircleGesture — a finger movement as it traces a circle;
- SwipeGesture — a finger movement as it traces a line;

- KeyTapGesture — a tapping movement as a finger taps a keyboard key;
- ScreenTapGesture — a tapping movement as a finger taps a vertical screen.

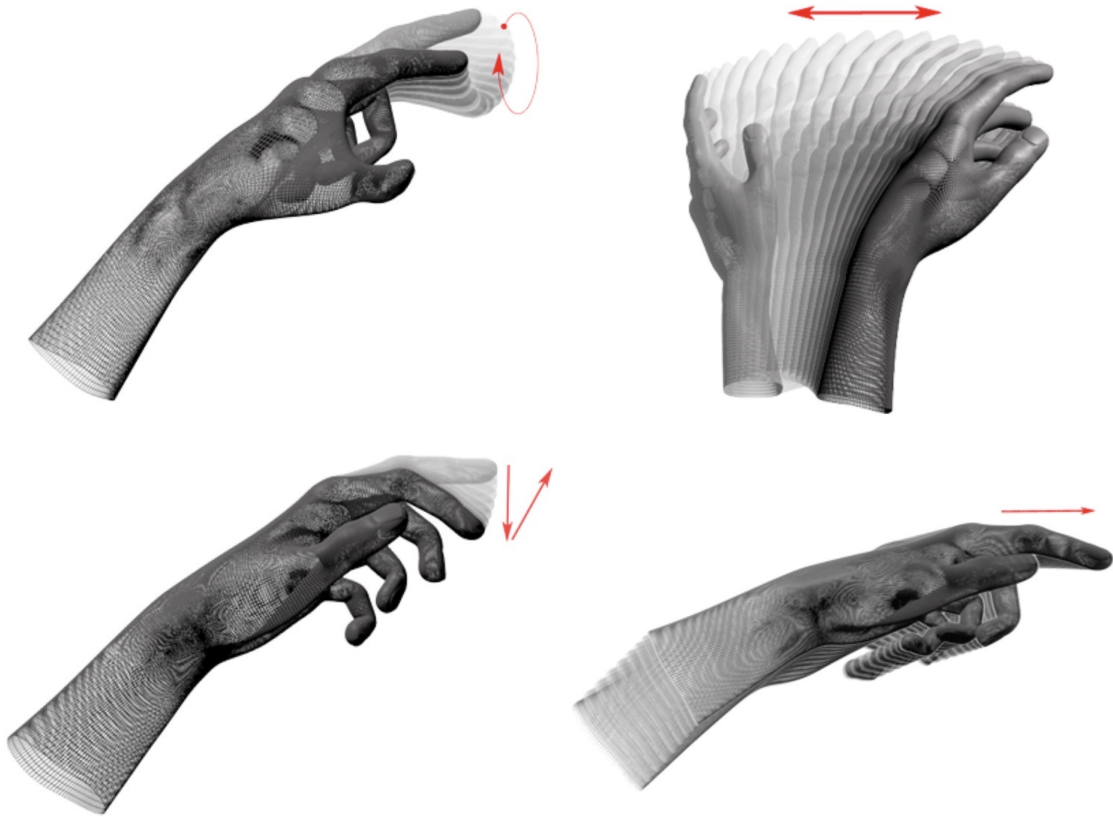


Figure 4.6: Representation of Leap Motion gestures [41]. On the top left, circle gesture, on the top right, swipe gesture, on the bottom left, key tap gesture, on the bottom right, screen tap gesture

The ImageList class contains the two raw images coming from the left and right infrared cameras. Image objects can be obtained both from a Frame object or directly from the controller. The difference is that the synchronisation between the user's hands and a Frame object is slightly lagged, while images obtained directly from the controller are real time but do not correspond to the hand represented on the screen. An Image object contains also the calibration data required to correct the lens distortion.

The Leap Motion software not only generates Frame objects, one independently from the other, but also keeps track of the correlation between a couple of frames. For example, if the user makes a rotation of his/her hand and Leap Motion recognises a frame in which the palm normal has a value and, later, another frame in

which the palm normal has a different value, the software compares these two frames and recognises the rotation. Motions are representations of these movements and the software is able to recognise them both with one single hand (considering the movement of fingers and properties of a Hand object) and two hands (considering the position of one hand with respect to the other).

There are three types of motions (figure 4.7):

- translation — linear movement;
- rotation — angular movement;
- scale — relative expansion or contraction.

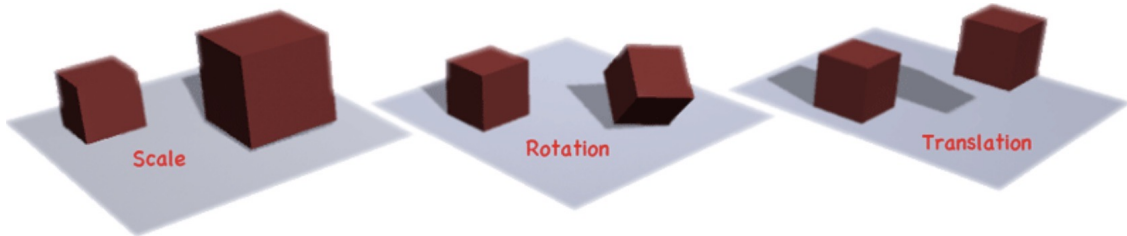


Figure 4.7: Representation of motions [41]. On the left, scale, in the middle, rotation, on the right, translation

#### 4.1.4 Touch emulation

Leap Motion API, through the `Pointable` class, provides information which can be used to implement a touch emulation.

Leap Motion defines a touch surface, roughly parallel to the x-y plane, which is used as a virtual surface. This surface divides the space into two zones: the hovering zone, between the user and the plane, and the touching zone, beyond the plane (figure 4.8). If the `Pointable` object, the finger or the tool, is too far from the touch surface, it is in the none zone.

The `Pointing` class declares the `Zone` enumeration composed by the hovering, touching and none states, which can be retrieved with the `touchZone` attribute. If the `Pointable` object is in the hovering zone or in the touching zone, it is possible to access to the `touchDistance` attribute, which returns a value in the range  $[+1, -1]$ . When the `Pointable` object enters the touching zone this quantity is  $+1$ ; then it decreases, becoming 0 when the `Pointable` object touches the virtual plane; when the `Pointable` object crosses the plane and is in the touching zone it continues to decrease without reaching  $-1$ . Of course the `touchDistance` attribute does not report the real distance from the plane but a normalised distance in the range  $[+1, -1]$ .

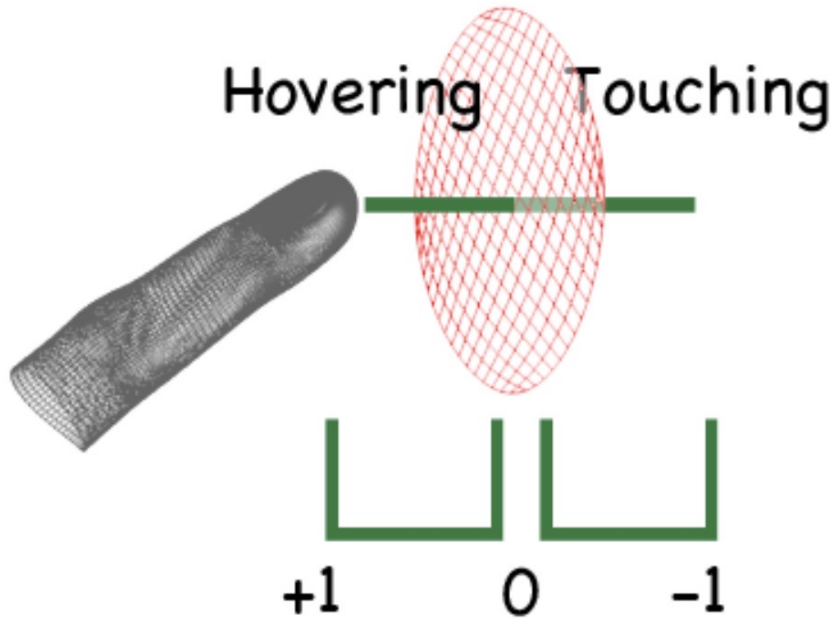


Figure 4.8: Representation of Leap Motion touch emulation with hovering and touching zones and  $[+1, -1]$  interval [41]

#### 4.1.5 ROS integration

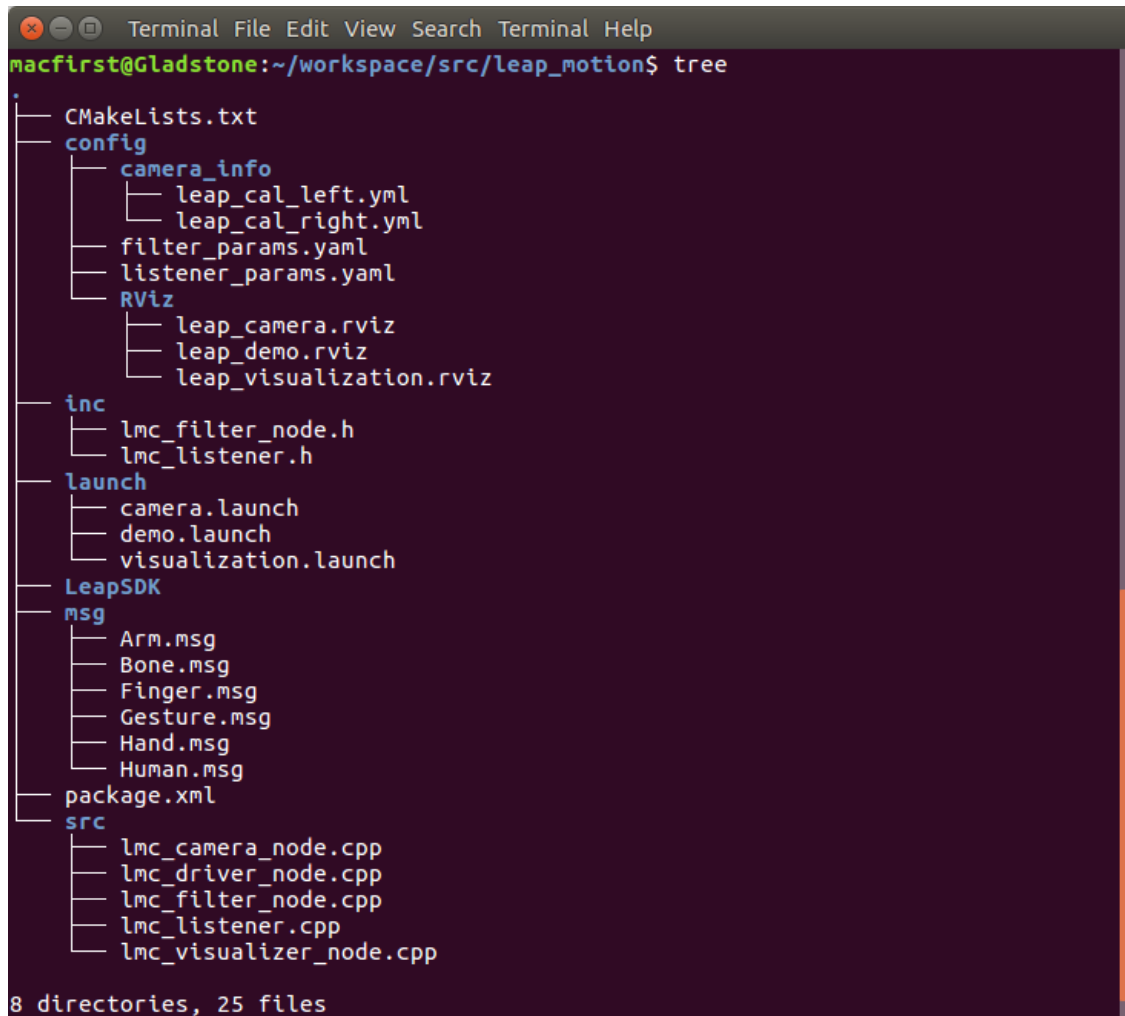
`leap_motion` [42] is a ROS package used as a wrapper for interfacing ROS nodes with the Leap Motion controller. This ROS package, apart from `CMakeLists.txt` and `package.xml`, also contains five folders (figure 4.9): `LeapSDK`, `config`, `msg`, `inc`, `src` and `launch`.

The `LeapSDK` folder contains all the files needed to interact with the Leap Motion controller. It includes some header files which can be used when Leap Motion functionalities are required.

The `config` folder contains two configuration files (`.yaml`) used to setup the controller and the filter, a folder which contains cameras parameters (both left and right), and a folder containing characteristics for Rviz visualisation.

The message folder contains message files, one for each message type. Messages were designed to be consistent with the tracking model of the Leap Motion controller. Message types are: `Bone`, `Finger`, `Hand`, `Arm`, `Gesture` and `Human`. Also in this case each message has its own properties and some message types compose other message types. For example, a `Finger` message contains a `Bone` list, an `Hand` message contains an `Arm` message, a `Gesture` list and a `Finger` list, and a `Human` message contains two messages of `Hand` type, one for the left hand and one for the right hand. Consequently from the definition of the `Human` message it is possible to deduce that `leap_motion` package supports one single person.

The `src` folder contains five source files in C++ language linked to header files



```

macfirst@Gladstone:~/workspace/src/leap_motion$ tree
.
├── CMakeLists.txt
├── config
│   ├── camera_info
│   │   ├── leap_cal_left.yml
│   │   └── leap_cal_right.yml
│   ├── filter_params.yaml
│   ├── listener_params.yaml
│   └── RViz
│       ├── leap_camera.rviz
│       ├── leap_demo.rviz
│       └── leap_visualization.rviz
├── inc
│   ├── lmc_filter_node.h
│   └── lmc_listener.h
├── launch
│   ├── camera.launch
│   ├── demo.launch
│   └── visualization.launch
├── LeapSDK
├── msg
│   ├── Arm.msg
│   ├── Bone.msg
│   ├── Finger.msg
│   ├── Gesture.msg
│   ├── Hand.msg
│   └── Human.msg
├── package.xml
└── src
    ├── lmc_camera_node.cpp
    ├── lmc_driver_node.cpp
    ├── lmc_filter_node.cpp
    ├── lmc_listener.cpp
    └── lmc_visualizer_node.cpp

8 directories, 25 files

```

Figure 4.9: Tree representation of leap\_motion package

in the inc folder. These files are:

**lmc\_driver\_node.cpp** This file generates a ROS node, contacts the Parameter Server to retrieve some Leap Motion controller configuration parameters (some for debug and some to enable and disable gestures), generates an instance of type `lmc_listener` (described below) and uses it to publish messages of type `Human` on a topic.

**lmc\_listener.cpp** This is the main file which allows nodes to receive tracked elements and their information. It contains the constructor and a list of callback functions which are:

- `onConnect` — called when a connection is established between the controller and the Leap Motion software;

- `onDisconnect` — called when the Controller object disconnects from the Leap Motion software or the Leap Motion hardware is unplugged;
- `onInit` — called when an instance of a Listener object is added to a Controller object;
- `onExit` — called when the Listener object is removed from the Controller object or the Controller instance is destroyed;
- `onFrame` — called when the Controller object receives a new frame. This function retrieves all tracked elements from the controller and puts them inside a message of type `Human`. Then this message is published on a topic to let all tracked data be available to any node on the ROS network.

**`lmc_filter_node.cpp`** This file creates a ROS node which implements a 2nd-order Butterworth lowpass filter [31]. This node receives `Human` messages from the `lmc_driver_node` and publishes them on another topic after filtering the position of each tracked element. The low pass filter uses the cutoff parameter to define the cutoff frequency of the filter. The higher its value is, the more trust is in the filtered data.

**`lmc_visualizer_node.cpp`** This file is used to create a visual representation of the hands. It creates a node that receives data from the filter node if it is enabled, otherwise directly from the driver node; then, it creates some `Marker` object and send them to Rviz using the `visualization_marker_array` topic (see section 3.7.1). The shape of the hand is generated using elementary shapes: `line_list` is used to draw the hand outline and fingers, spheres are used to draw joints and the centre of the palm.

**`lmc_camera_node.cpp`** This file creates a ROS node that contacts the controller and retrieves images generated by the cameras. The node publishes on different topics, both for left and right camera, raw images and camera information.

The launch folder contains three launch files:

- `visualization.launch`, first of all, loads some parameters, some used by the driver node for debug reason and for gestures configuration, others used by the filter node; it creates one instance of `leap_motion_driver_node`, `leap_motion_filter_node` and `leap_motion_visualizer_node`. These three nodes work in a pipeline mode: data are generated by the controller, then they are filtered and finally used to create a visualisation of hands. The launch file also launches the Rviz visualiser to display the shapes of the hands (figure 4.10). `static_transform_publisher` is used (see section 3.4.2) to rotate the Leap Motion coordinate system in order to match the world coordinate system (static).



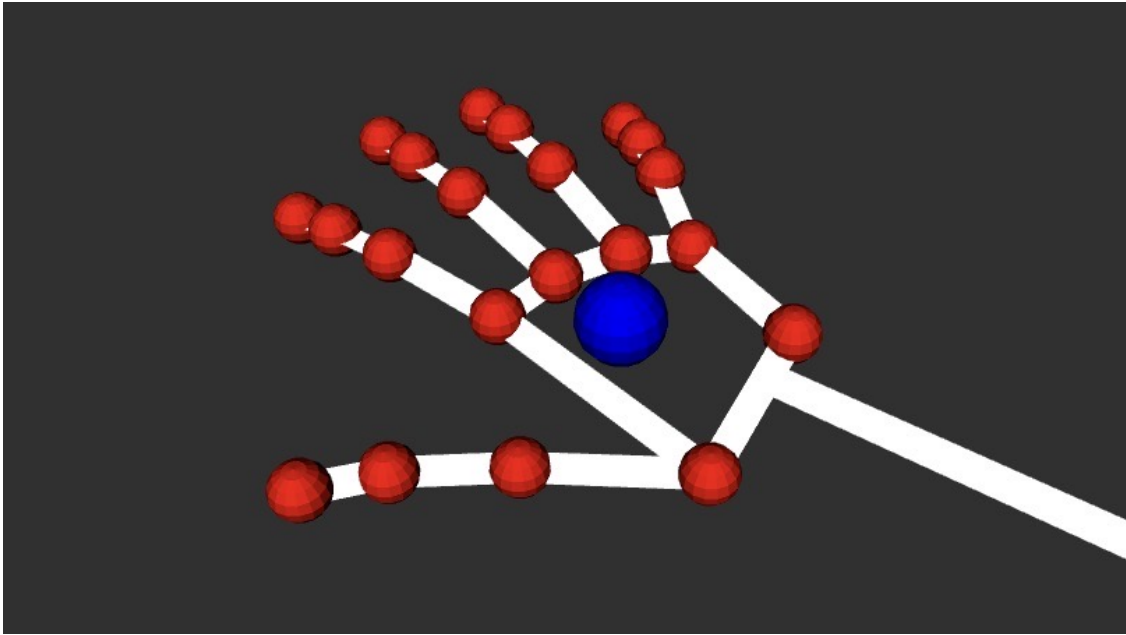


Figure 4.10: Representation of one hand in Rviz using MarkerArray display: line\_list for the hand outline and fingers, spheres for joints and the centre of the palm

- camera.launch creates an instance of leap\_motion\_camera\_node, which is used to display the raw image of a camera, and an instance of stereo\_image\_proc from stereo\_image\_proc package, which performs the duty of image\_proc for both cameras so as to remove camera distortion from the raw images. Also this launch file launches an instance of Rviz visualiser, to display the raw image of one camera, using the Image display, and the point cloud (figure 4.11) generated by stereo\_image\_proc node, using the Point Cloud 2 display (see section 3.7.1). Also this launch file creates an instance of static\_transform\_publisher to rotate the point cloud coordinate system in order to match the world coordinate system.
- demo.launch is a combination of the two previous launch files that allows the visualisation of the shapes of the hands using MarkerArray display, to fill the 3D view using PointCloud2 display and to visualise the raw image of the camera using Image display (figure 4.12).

## 4.2 Niryo One

Niryo One is a 6-axis collaborative robotic arm, which means that it has 6 degrees of freedom. It is an educational robot, used to train industry 4.0, which is open source



Figure 4.11: Representation of one hand in Rviz using PointCloud2 display

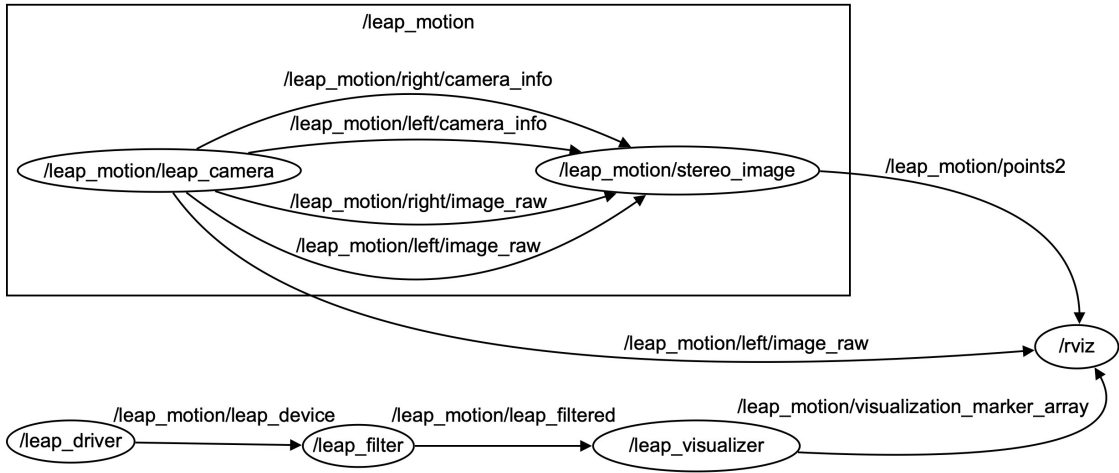


Figure 4.12: Graph which represents the interconnection among nodes when the demo.launch file is launched

and 3D printed, so its cost is very low with respect to other collaborative robots. Its weight is 3.3 Kg and its reach is 440 mm. The company provides STL files that allow buyers to print the robot by themselves. It is composed of seven pieces: base, shoulder, wrist, arm, elbow, forearm, wrist and hand (figure 4.13). They also provide STL files for five different interchangeable end-effectors: three grippers,

one vacuum pump and one electromagnet. Each tool has the same mechanical connector and the company also provides instructions that enable users to design their own end-effectors if they want. In this project “Gripper 1” was used. It has a weight of 70 g, a length (gripper closed) of 80 mm and a maximum opening width of 27 mm. A representation of this gripper can be seen in figure 4.14

1 - Base	Orange
2 - Shoulder	Yellow
3 - Arm	Green
4 - Elbow	Turquoise
5 - Forearm	Blue
6 - Wrist	Purple
7 - Hand	Red



Figure 4.13: Niryo One structure composed of seven pieces, each one reported in the table on the left [48]

Weight	70 g
Length (gripper closed)	80 mm
Max opening width	27 mm
Picking distance from end effector base	60 mm



Figure 4.14: Niryo One Gripper 1 with its specific in the table on the left [48]

It is possible to easily connect the robot with other devices. The user can integrate Arduino and/or Raspberry Pi boards to create a controlled environment in which more Niryo One robots can be synchronously controlled or where an

assembly line or any other robotic arm application can be simulated. The company also provides a free desktop application to easily interact with the robot: Niryo One Studio. The software has a graphical interface not only to control each joint separately but also to create programs using blocks, so no robotic knowledge is needed. The robot can be controlled using an ethernet cable or Wi-Fi. In the latter case the robot can work as a hotspot mode or it can be connected to a local network.

### 4.2.1 Niryo One ROS stack

Figure 4.15 is the representation of different layers that compose the Niryo One ROS stack [50]. Going from the bottom up, it is possible to define the hardware layer. This layer is the communication channel between the hardware itself and the software. One part is inside the `niryo_one_rpi` package (mostly Python) and it is responsible for the top button, the LED, the digital I/O panel, the Wi-Fi, the fans, etc.; the other part is the motor driver and it is responsible for handling the CAN and Dynamixel buses to control motors. This driver works as an interface between the motors and ROS and it is able to both send commands to all motors and read their position.

The above layer is the control layer. It uses the `ros_control` package next to the `joint_trajectory_controller`. This controller is a position controller that receives a trajectory and runs a control loop: it reads the current position from the driver, interpolates the trajectory (using a quintic spline) to get the next position command and sends it to the driver.

On top of the control layer there is the motion planning layer. This layer uses the well-known ROS MoveIt package (see section 3.5) to find the inverse kinematics and build a path for the robot. The path is composed of a series of points and for each one a specific position, velocity, and acceleration are given. Then the path is sent to the `joint_trajectory_controller`.

Next there is the command and user interface layer. This layer is the link between the ROS system and the client (the user or another machine). All commands go through this layer, coming from the command line, code or graphical user interface.

The last level is the external communication level and it is used to communicate with what is outside the system. For example, the user can use a joystick, through a USB bus, to send commands to the robot.

### 4.2.2 Simulation

The company also provides the `niryo_one_ros_simulation` metapackage [49], which contains three packages: `niryo_one_description`, `niryo_one_gazebo` and `niryo_one_moveit_config`.

The `niryo_one_description` package provides all the material needed for the

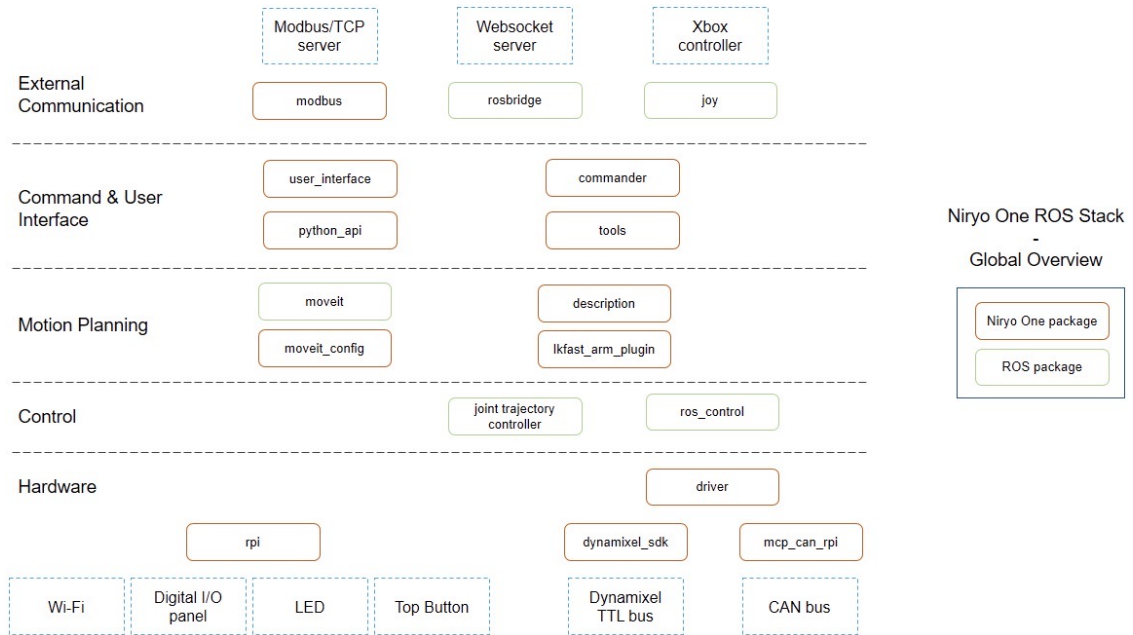


Figure 4.15: Representation of the division in different layers of the Niryo One ROS stack [50]

```

macfirst@Gladstone:~/workspace/src/niryo_one_ros_simulation/niryo_one_description
n$ tree
.
├── CMakeLists.txt
├── config
│   └── default_config.rviz
├── launch
│   └── display.launch
├── meshes
│   ├── collada
│   │   ├── base_link.dae
│   │   ├── ...
│   │   └── hand_link.dae
│   └── stl
│       ├── base_link.stl
│       ├── ...
│       └── hand_link.stl
├── package.xml
├── urdf
│   ├── gazebo_niryo_one.urdf.xacro
│   └── niryo_one.urdf.xacro
└── 6 directories, 20 files

```

Figure 4.16: Tree structure of niryo\_one\_description package

visualisation of the robot. It contains (figure 4.16): mesh files, in the meshes folder,

of the seven pieces that constitute the robot, both collada and stl formats, used in the URDF file for the visual and collision properties, respectively (see section 3.3.1); two xacro files, in the urdf folder, one used simply to display the robot from the base link to the head link (the gripper is not included) and the other containing also the inertial parameters and transmission tags for the simulation with Gazebo (see section 3.8); a launch file, in the launch folder, which loads the robot description (the simpler version) into the Parameter Server and uses `joint_state_publisher` and `robot_state_publisher` and Rviz to display the robot (figure 4.17); a configuration file, in the config folder, which is used to configure the Rviz window.

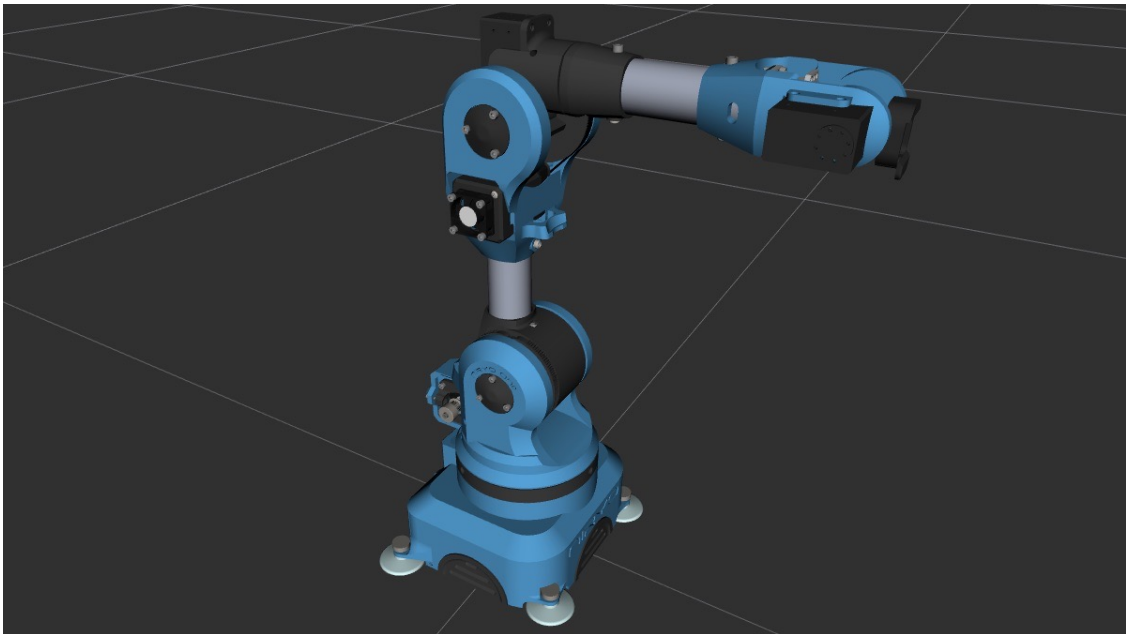


Figure 4.17: Representation in Rviz of Niryo One using RobotModel display

The `niryo_one_gazebo` package is used for the simulation. It contains a configuration file (yaml file), in the config folder, with the description of the controllers (`joint_state_controller` and `joint_trajectory_controller`) and two launch files, in the launch folder (figure 4.18). One launch file loads the robot description (Gazebo version) into the Parameter Server, runs Gazebo and populates it with the robot model (figure 4.19); the other file loads into the Parameter Server the configuration file containing the controller parameters, uses `robot_state_publisher` node, to communicate the state of the robot, and a spawner node from the `controller_manager` package which loads and starts the two controllers.

The `niryo_one_moveit_config` package is used by MoveIt to plan the motion. As figure 4.20 shows, this package contains some configuration files, in the config folder, generated by the MoveIt Setup Assistant, which contain some information

```

macfirst@Gladstone:~/workspace/src/niryo_one_ros_simulation/niryo_one_gazebo$ tree
.
├── CMakeLists.txt
├── config
│   └── niryo_one_controllers.yaml
├── launch
│   ├── niryo_one_control.launch
│   └── niryo_one_world.launch
└── package.xml

2 directories, 5 files

```

Figure 4.18: Tree structure of niryo\_one\_gazebo package

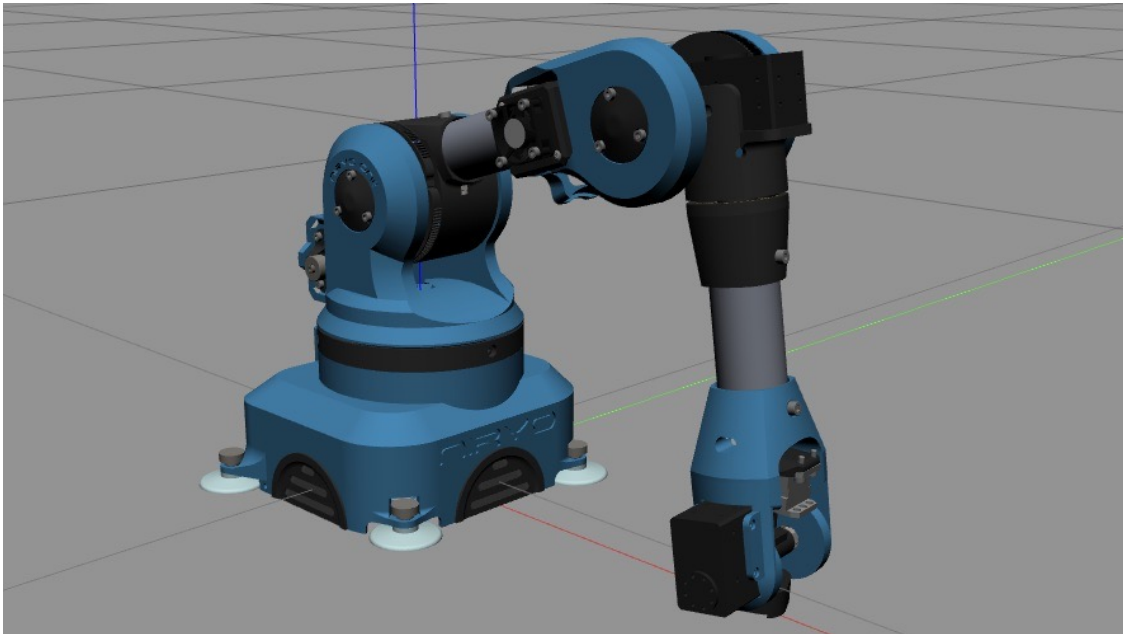
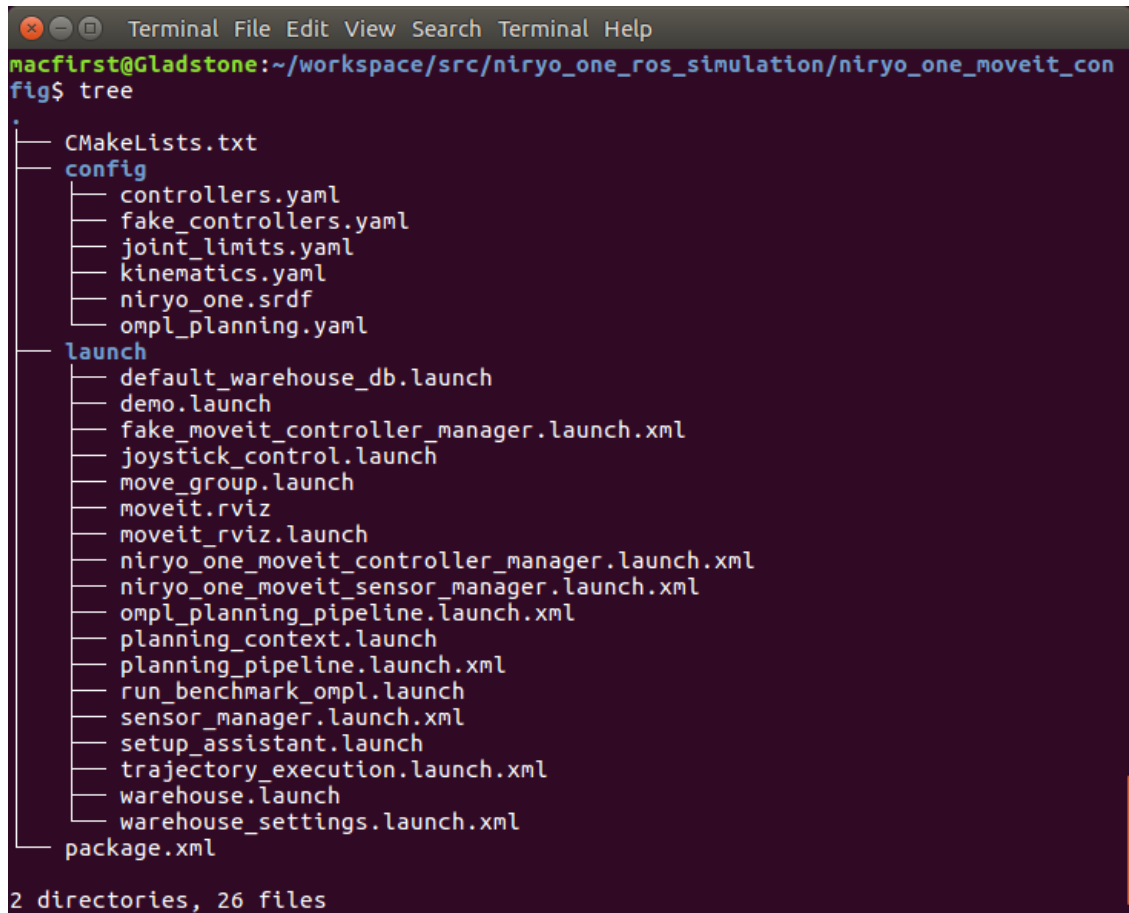


Figure 4.19: Simulation of Niryo One using Gazebo

like joint limits, kinematics, semantic robot description, ompl planning configuration, fake and real controllers parameters. The package also contains some launch files, in the launch folder. The most used are `move_group.launch` and `demo.launch`.

The first file is the main MoveIt executable. First of all it launches `planning_context.launch`, which is used to upload all needed parameters on the Parameter Server, like robot description, semantic robot description, joint limits and kinematics; it launches `planning_pipeline.launch.xml`, which uploads all parameters needed for planning functionality (`ompl_planning_pipeline.launch.xml`); it launches `trajectory_execution.launch.xml`, which uploads the controller manager





```

macfirst@Gladstone:~/workspace/src/niryo_one_ros_simulation/niryo_one_moveit_config$ tree
.
├── CMakeLists.txt
├── config
│   ├── controllers.yaml
│   ├── fake_controllers.yaml
│   ├── joint_limits.yaml
│   ├── kinematics.yaml
│   ├── niryo_one.srdf
│   └── ompl_planning.yaml
├── launch
│   ├── default_warehouse_db.launch
│   ├── demo.launch
│   ├── fake_moveit_controller_manager.launch.xml
│   ├── joystick_control.launch
│   ├── move_group.launch
│   ├── moveit.rviz
│   ├── moveit_rviz.launch
│   ├── niryo_one_moveit_controller_manager.launch.xml
│   ├── niryo_one_moveit_sensor_manager.launch.xml
│   ├── ompl_planning_pipeline.launch.xml
│   ├── planning_context.launch
│   ├── planning_pipeline.launch.xml
│   ├── run_benchmark_ompl.launch
│   ├── sensor_manager.launch.xml
│   ├── setup_assistant.launch
│   ├── trajectory_execution.launch.xml
│   ├── warehouse.launch
│   └── warehouse_settings.launch.xml
└── package.xml
2 directories, 26 files

```

Figure 4.20: Tree structure of niryo\_one\_moveit\_config package

and controller parameters that can be real (`niryo_one_moveit_controller_manager.launch.xml`) or fake (`fake_moveit_controller_manager.launch.xml`); finally, it starts the `move_group` node/action server that provides MoveIt functionality.

The second file is used to display the robot and to use MoveIt with fake controllers. It launches `planning_context.launch`; it uses `joint_state_publisher` to publish fake joint states because the robot is not connected; it uses `robot_state_publisher`, given the published joint states, to publish `tf` for the robot links; it launches `move_group.launch` using a fake execution; finally it launches `moveit_rviz.launch` which runs Rviz with its configuration file, `moveit.rviz`. The Rviz window allows the user to see the planning scene that MoveIt uses to calculate the trajectory avoiding collision, and the trajectory itself.



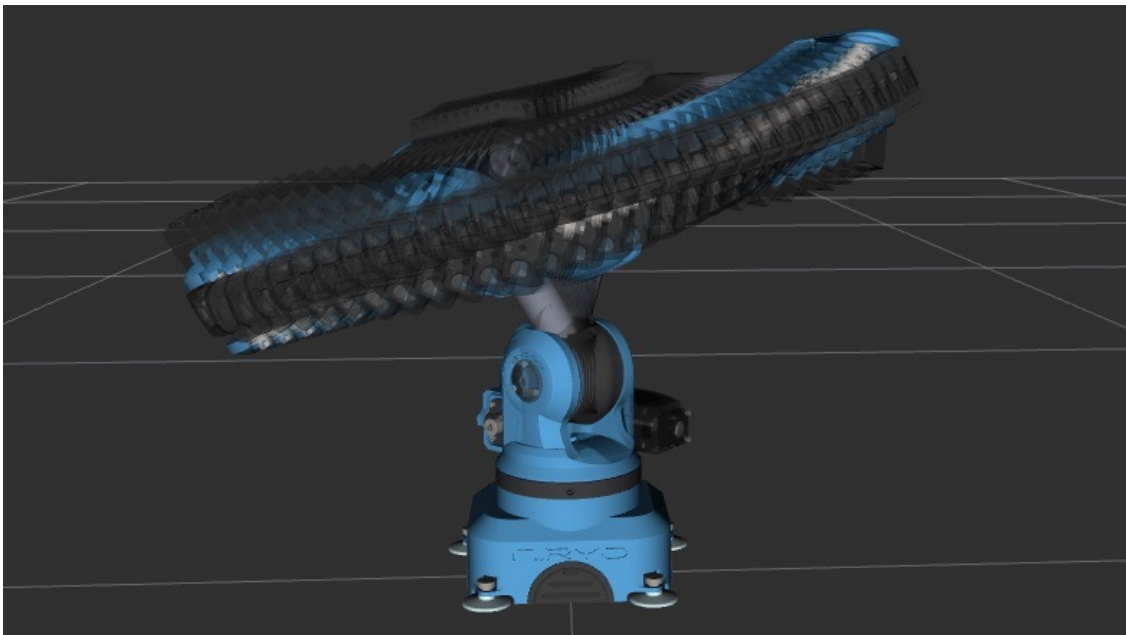


Figure 4.21: Representation of the trajectory in Rviz using Trajectory display



## Chapter 5

# System development

The goal of this study was to implement a collaborative assembly task, using Niryo One robotic arm, two low cost sensors, a Leap Motion controller and a webcam, as input devices, and an LCD as output. Human operators, using gestures, perform a Human-Robot Collaboration in order to carry out a pick and place operation.

### 5.1 The concept

The first innovative idea was to place the camera above the robot, focusing the workspace, and to display the output on the LCD (figure 5.1). In order to perform an industrial assembly task, the user can now see the robot and a set of pieces on the screen. Human operators can use the Leap Motion controller, placed in front of the screen on the same surface. With their index they can indicate some pieces on the screen to pick them up and place them in a given order.

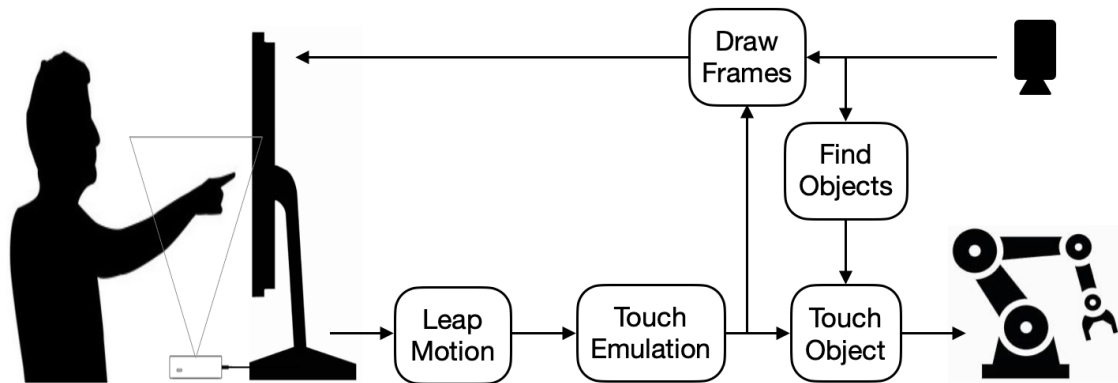


Figure 5.1: Block diagram of the proposed system

As discussed in chapter 2, many studies based their research on Human-Robot Collaboration using gestures, because they are a natural way for humans to interact

(see section 2.3.1).

As explained in [26], the pointing gesture, in particular “point at part” gesture, is universally understandable and very simple; the study [17] shows that people perform better when they use a simple method which is natural for them, as gestures; moreover, [57] shows that the interaction quality is better when there is a feedback in the communication.

The system proposed in this thesis tries to take into account the good results of these previous studies and combines all these features in order to create an innovative way of Human-Robot Interaction using the “point at part” gesture. Moreover there are no studies focusing on a direct interaction between a screen and gestures; so it was decided to develop a system using the LCD not only to visualise the image coming from the camera placed above the robot, but also in conjunction with the Leap Motion controller to develop a touch emulation system.

Gestures are generally used in conjunction with Augmented Reality to recreate a feedback marker but this technique requires expensive devices. In this study feedback was integrated with the touch emulation system employing the already used LCD, adding a virtual marker to the image stream coming from the camera.

Human operators, using the tip of their index, touch the object image on the screen, guided by the feedback marker, and communicate to the robot which object to pick up. This way the robot performs a pick and place operation from the object position to the first placing area in front of it. Then the operation is repeated again to pick and place another object in the second placing area to complete the assembly task.

## 5.2 System architecture

The proposed system is composed of several pieces, which can be represented in a schematic way as shown in figure 5.2. In the extremities (top and bottom) of the scheme, the hardware used in the system is represented, while the software is in the middle (with the exception of Rviz). Information flows from the top (input devices) to the bottom (output devices). The scheme can be divided into 6 layers. They are described below, following the same order of the information flow:

**input devices** The Leap Motion controller was used to recognise the human hands, the camera was used to retrieve images of the workspace from the top view.

**drivers** Although they have been called drivers, they are in reality some ROS nodes used as links between the input devices and the main layer. These nodes perform a set-up phase to adapt raw information in order to be available for the next layer.

**main layer** This layer represents the core of the system. It is responsible for many things: first of all, the visualisation of the hands recognised by the Leap Motion

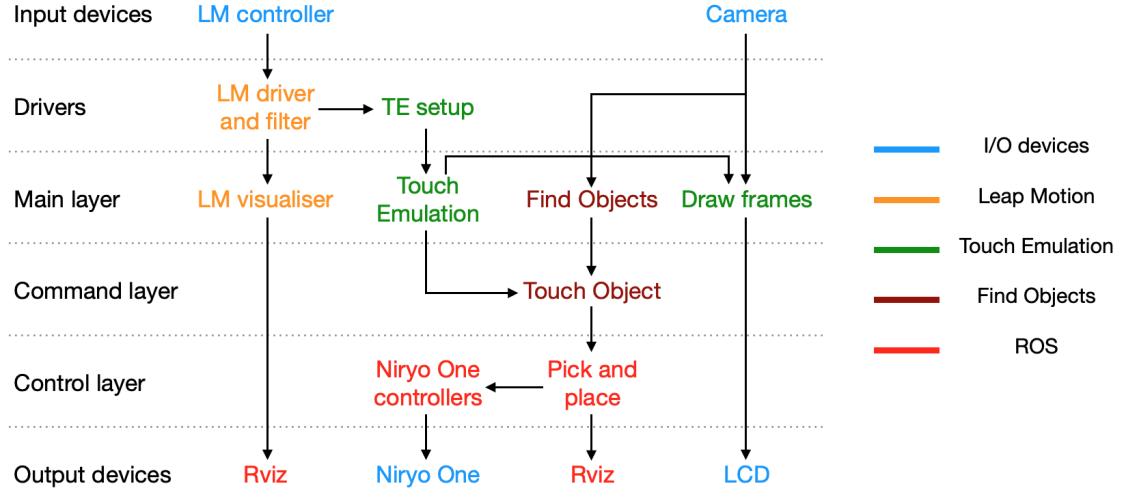


Figure 5.2: Schematic representation of the proposed system and division in layers

controller; secondly, the visualisation and manipulation of images coming from the camera; thirdly, the implementation of the touch emulation system; finally, the detection and recognition of objects in the workspace.

**command layer** This layer represents the crux that fuses together information coming from the user and the robot. In other words, it takes the information coming from the touch emulation block and the one coming from the find objects block and verifies if their combination makes sense. If it does, it sends the command to the next layer.

**control layer** This layer consists of ROS components responsible for the pick and place operation and commands to be sent to the robot controller to perform a given task.

**output devices** The robot itself and the LCD, which shows images coming from the camera, are considered as output devices. Rviz was added to this last layer because it is not possible to interact with it, but only to visualise information about the system. For this reason Rviz was considered as an end point, even though it is constituted by a software part.

The system was developed following a top-down approach, with the exception of the workspace, which was developed firstly, because the camera retrieves the information from there. Each block of the system was developed separately and only at the end, when all components were completed, the whole system was simulated. Firstly the workspace was organised and some pieces were designed, secondly the touch emulation system was developed, then the find-object application was adapted and finally the pick and place operation was performed.

## 5.3 Workspace

The assembly task was performed using the Gazebo simulator (see section 3.8). The workspace is composed of a flat surface on which the robot, the camera and some pieces to pick and place are located.

### 5.3.1 Hardware

NIRYO company provides all files (.stl) needed to print the robot links and the gripper with a 3D printer and other physical parts that are not printable. Then the final user uses these components to mount the robot. On the contrary, the company does not provide all needed files (.dae) used for the simulation, but only those files regarding the robotic arm and not the ones regarding the gripper.

The first problem was encountered in the simulation of a pick and place task because, of course, the gripper is needed. Due to the fact that the company does not provide any file for the simulation of the gripper, files used to print the gripper are also used for the simulation. Differently from the robotic arm, whose .dae files are used for visualisation and .stl files are used for collision, gripper visual and collision tag are both realised using .stl files. This allows the realisation of the same behaviour regarding the motion, while it is not possible to visualise the texture and colours as in the case of the robotic arm.

Similarly to the real gripper, which is equipped with a single motor, also the simulated gripper was developed in the same way. For this reason the `hand.urdf.xacro` file, which describes the Niryo One Gripper 1, contains only one independent joint, while the second joint mimics the first one. To reproduce this behaviour in the Gazebo simulator, the `mimic_joint_plugin` was also included in the file `gazebo_hand.urdf.xacro`, in addition to the `transmission` tag for the independent joint. In reality some connectors allow the conversion of the rotating movement of the rotor into the translational movement of the gripper's fingers. For simplicity, the joint declared in the urdf file was defined as prismatic. Moreover a controller of type `effort_controller/JointTrajectoryController` was added to control the gripper and some files containing its parameters. Finally, all files were adapted to contain `niryo_one_hand.urdf.xacro` and `gazebo_niryo_one_hand.urdf.xacro`, which include the description of both robotic arm and the gripper.

As figure 5.3 shows, the robot was positioned at the center of the world frame while the camera was placed above it, at a distance of 1m from the horizontal plane. The camera is represented in simulation as a 0,1m long white cube which is not affected by gravity (as in reality it was suspended in the air). The camera is simulated with an image that is 1920x1080 pixels large and an update rate of 30 fps, in order to reflect real camera parameters. The image is oriented with the positive x world frame axis going down and the positive y world frame axis going right. In this way the x-y plane of the image matches the x-y plane where the

robotic arm and pieces are placed. Moreover Gazebo simulator gives the possibility to add some white noise, as it was in reality, before publishing the image on the topic `sensor/usb_cam/image_raw` using `sensor_msgs/Image` messages.

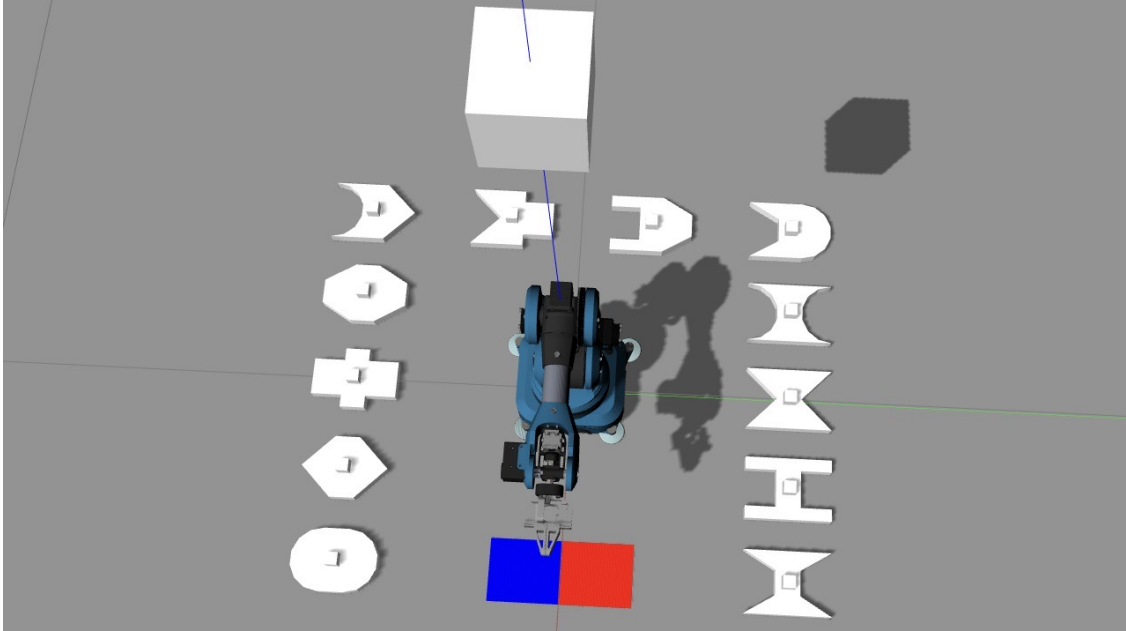


Figure 5.3: Representation of the simulated system composed of Niryo One in the middle, a camera placed above it (white box), white pieces around the robot and two place locations in front of it (blue and red rectangles)

### 5.3.2 Other components

Gazebo does not simulate only gravity and textures, but also light. In the simulation a directional light was used, to recreate the sunlight coming from a window, as in a real environment. This light generates some shadows which are fundamental in object recognition (see section 5.5).

To perform pick and place operations some objects are placed in a horseshoe shape around the robot (figure 5.3). It was decided to organise the workspace in this way because it was thought that in an industrial assembly task, this last is the final step of a chain. Possibly the pieces are placed from other robots after a manipulation task.

These objects are all different from one another but follow the same pattern. As figure 5.4 shows, one object is represented by a parallelepiped having base 10 cm x 14 cm and a height of 2 cm. The middle section (in green) is the common part to all pieces, while the shortest edges of the parallelepiped have a concave shape that matches the convex shape of another object and vice versa. Placed above the

parallelepiped, in the middle, there is a cube with 2 cm long edges, used by the gripper to easily pick the object. The lower part is larger, to be better identified by the find-object software, while the upper part is smaller because the Niryo One Gripper 1 has a small opening width. Moreover the lower part was designed in this way so that it would be too difficult for a robot to perform an assembly task by itself, leaving to the human operators the ability to choose two matching objects, thus exploiting the Human-Robot Collaboration.

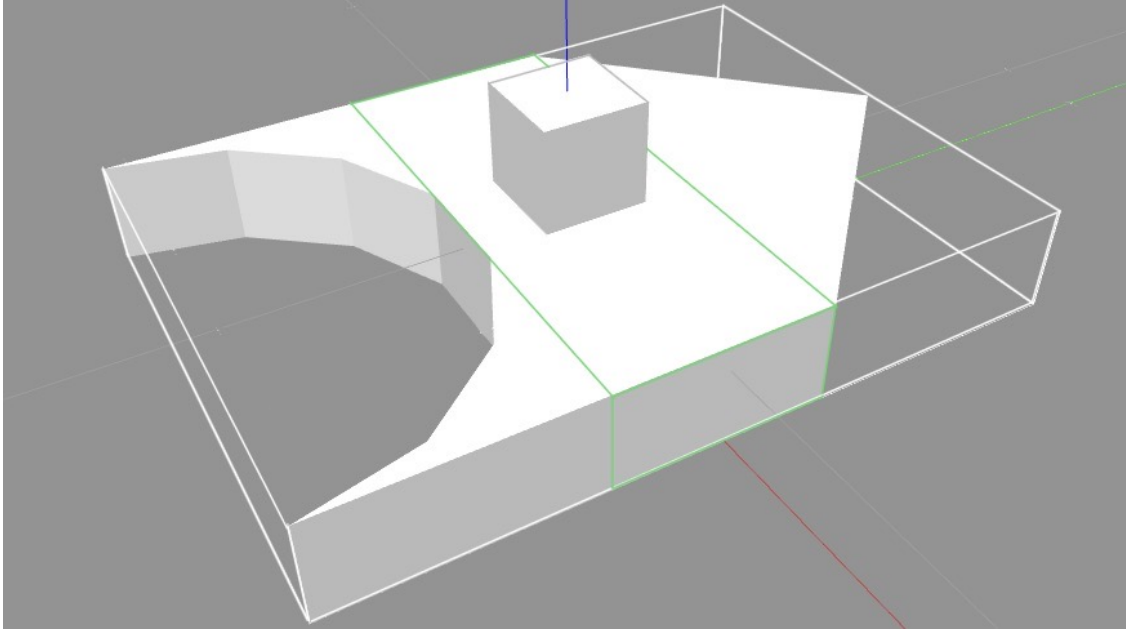


Figure 5.4: Representation of one piece used in the simulation with a concave shape on one side and a convex shape on the other side. The middle section (green) is the common part.

Each single piece was firstly created using an external software and then exported using .stl format using an ID represented by a unique increasing number. Then each piece was added to a .xacro file describing it and named `pieceX.xacro`, where X is the object's ID. Finally, these xacro files are used to spawn the pieces in Gazebo through the `gazebo.launch` file.

## 5.4 Touch emulation

The developed touch emulation combines information retrieved by the Leap Motion controller, using the `leap_motion` package, with the image coming from the camera.

At the beginning the first idea was to use the pointing gesture to indicate an object, but during the developing phase many problems occurred.



The first difficulty occurred in the definition of the pointing finger gesture. This can be identified when the index is straight and the other fingers are closed. But it is difficult to identify how straight the index has to be or how closed other fingers have to be. However, Leap Motion provides a function, which is not reported in the Bone message of the `leap_motion` package, which returns the normalised direction of a bone from the base to the tip. By adding this information to the Bone message it is possible to calculate the angle between two bones, using the function `Vector::angleTo(const Vector & other)`. After a threshold  $\Delta$  has been defined, the two conditions to identify the position of the fingers for the pointing gesture are:

- the angle between each couple of adjacent bones of the index has to be smaller than  $\Delta$ ;
- the angle between metacarpals and proximal phalanges has to be greater than  $\Delta$  for the remaining fingers, with the exception of the thumb.

It was observed that, setting  $\Delta = 0,2\text{rad}$  (about  $11,5^\circ$ ), the position of the fingers is similar to the position of the pointing gesture normally used. Moreover this value allows the recognition of the pointing gesture also when the index is not completely straight.

Another problem that occurred during the development phase was the time needed to define a pointing gesture. An initial idea was to let a timer start each time the position of the fingers verified the previously described conditions, and trigger the pointing gesture event when the timer reached zero. In this case, if the timer was set to a low value, the gesture would be recognised very quickly, but the user would not have enough time to identify the virtual marker on the screen and move it to the desired position; on the contrary, if the timer was set to a high value, then the user would have the possibility to point at an object on the screen without triggering the gesture event, but it would be very unlikely to keep the hand still while indicating an object on the screen.

The last problem, extensively discussed in many studies (see section 2.4.1), was the definition of the “base point” (the “tip point” is defined by the tip of the index) to define the pointing line which intersects the pointed plane. In this case, as the user’s head cannot be detected by the Leap Motion controller, it could not have been used as the base point. Other alternative base points could have been either the base of the finger or a Virtual Projection Origin (VPO). In any case, whatever point had been chosen as the base point, it could not have corresponded to what the user expected, due to a different mental model.

To solve all these problems it was necessary to find a way of interaction independent from the position of the fingers, from a timer to trigger the event and possibly also independent from how the base point is defined. The new way of interaction had to maintain, of course, the possibility to interact using feedback.

In [17], it was seen that people perceive more accurately and prefer a system they are more comfortable with, such as a touchscreen; in [69] a Layered Touch Panel was developed to implement the hovering event. These articles, in conjunction with the touch emulation API provided by Leap Motion (see section 4.1.4), generated the idea to implement a virtual Layered Touch Panel which solved the problem described before.

Leap Motion provides some API for the touch emulation which works using an imaginary vertical plane, centred in the origin of the Leap Motion controller. There is no direct correspondence between the human hand position and the pointer on the screen and no feedback is displayed on the screen.

The touch emulation system proposed in this study generalises the already available touch emulation for a general LCD screen, implementing a direct correspondence using a 1:1 scale and having the touching plane correspond to the LCD screen, whatever its position and orientation. The real time feedback was established by adding a virtual panel to the LCD. For simplicity, an assumption was made for the development: the plane where the LM controller is placed and the LCD screen are perpendicular. This assumption is quite reasonable, because having a vertical screen on an horizontal plane is very common.

In this case the LCD was not used only as an output device where to display images coming from the camera, with the possible addition of the feedback marker, but also as a tool to define the touch panel and the virtual one.

This touch emulation system was developed to be as general as possible. Using the same ROS principles, this system is usable with every type of screen and with every dimension (inside the Leap Motion field of view).

### 5.4.1 Screen definition

After the Leap Motion controller is correctly linked with the computer and working, and the leap daemon is run and the demo.launch file is launched (see section 4.1.5), first of all, the file screen\_edges\_definition has to be run. This file was written to implement a calibration phase, in which the human operator uses LM to define the screen in the virtual space, used later as point of reference for the touch emulation. The screen definition is performed, saving on the Parameter Server some information about the screen, with respect to the world coordinate system. These parameters are the dimensions of the screen, its position and its orientation, with respect to the world frame.

At first, the position of the screen was represented by the centre of the screen, while the orientation was identified by the normal vector coming out from the screen. It was then seen that this definition was useful only for the visualisation of the virtual screen.

Later, the position was redefined to coincide with the top left corner of the screen, while the orientation was defined by the normal vector entering the screen.

The next section of this thesis will illustrate how useful this choice is to realise the screen Transform.

When the executable `screen_edges_definition` is running, users have to touch with their index the top left corner of the screen and save its coordinates by pressing the spacebar or the enter key. Then they repeat the operation with the remaining corners, in a clockwise direction. The screen dimensions are calculated as the mean of the distance between the top and bottom corners for the width, and the mean of the distance between the right and left corners for the height. To calculate the screen orientation, instead, the coordinates of the center are calculated first. This point is calculated as the mean, for each coordinate, of the four corners. The four vectors are calculated, one for each corner. Each vector has the head in one corner and the tail in the previously calculated centre point of the screen. After that, the two top vectors and the two bottom vectors are used to calculate two normal vectors, by making the cross product for each couple of vectors. In order to obtain one single normal vector (red vector in figure 5.5), the mean between the two normal vectors is calculated. Finally this vector is normalised. Before storing these parameters in the Parameter Server, they are converted in the world frame and the vector, which represents the orientation, is converted in a quaternion. For the orientation, only the yaw angle is considered thanks to the previous assumption which imposes to have the screen plane perpendicular to the plane on which the Leap Motion controller is placed.

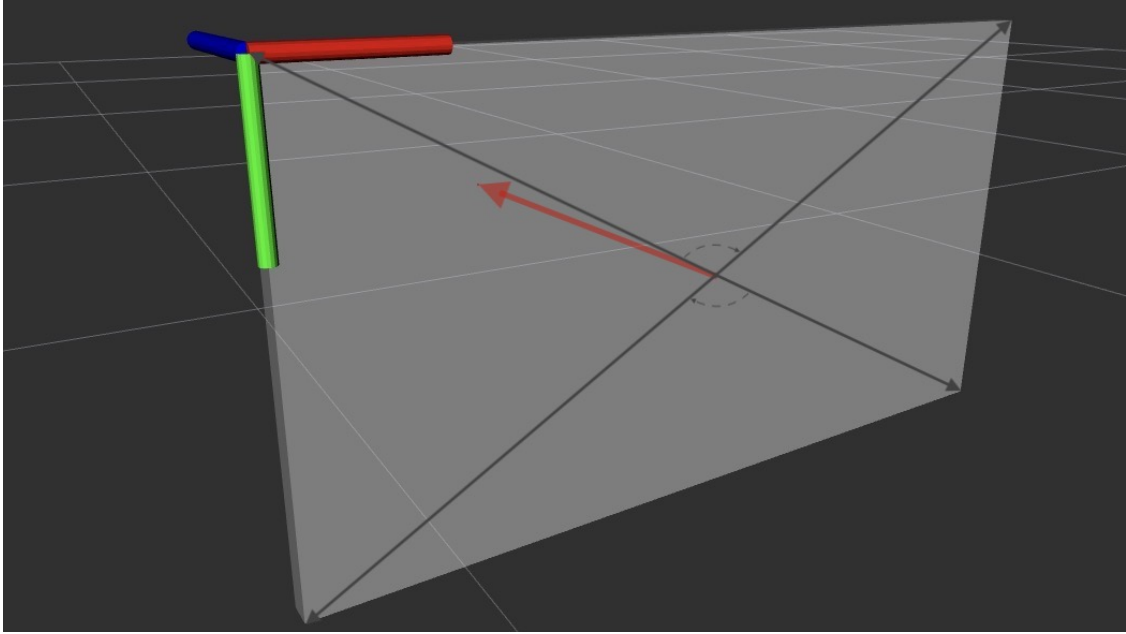


Figure 5.5: Representation of the screen corners as vectors with respect to the center of the screen and the normal vector (red) entering in the screen surface

### 5.4.2 Interaction system

After the calibration phase is completed, in order to make the touch emulation work properly, three source files were created. Some executables and other parameters are included in the `touch_emulation.launch` file in order to start the touch emulation system with one single command. Some parameters regard the camera, as the image dimension in pixels, and some the touch emulation, as the hovering distance, the touching distance and the pointer radius of the marker visualised on the screen (see next section). The three executables in the launch file are described below (figure 5.6):

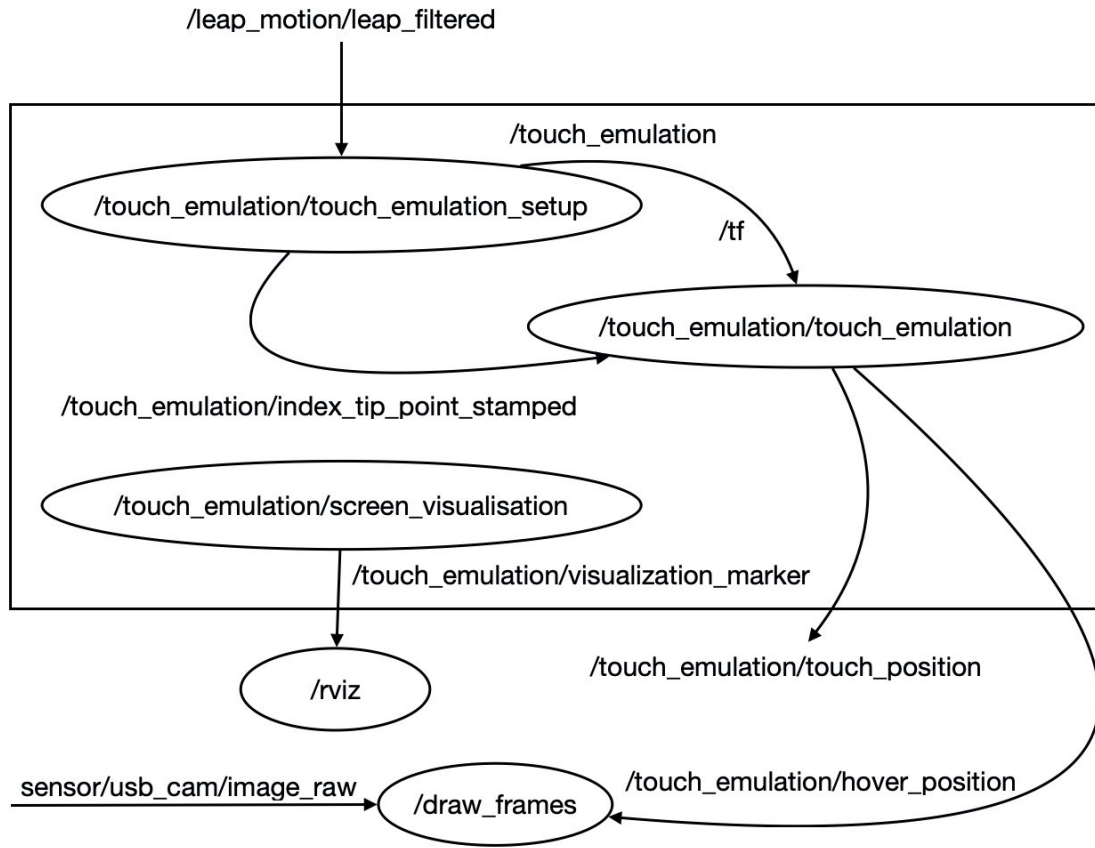


Figure 5.6: Graphical representation of the touch emulation system

**leap\_motion\_touch\_emulation\_setup** This file creates a ROS node and, by reading the screen parameters from the Parameter Server, creates the “screen” TF (see section 3.4) using the parameters previously stored in the Parameter Server during the calibration phase. It means that the TF is positioned in the top left corner of the screen, and the same orientation of the screen, with the positive x axis going to the right and the positive y axis going down along

the screen (figure 5.7). As a consequence, the direction of the positive z axis enters the screen. This reference frame was set to match the image coordinate system, which has the origin in the top left corner, x values growing towards the right and y values growing towards the bottom. This validates the decision to define the position and the orientation as explained before. Therefore it is very simple to transform points from the world frame to the screen frame and scale them from meters to pixels.

In addition, in order not to broadcast useless information, this node filters the position of the tip of the index from the Human message and sends it on the topic `index_tip_point_stamped` using a `geometry_msgs/PointStamped` message.

**screen\_visualisation** This file simply visualises the virtual screen in Rviz (figure 5.7). It uses the Marker display (see section 3.7.1) to show a rectangular box with respect to the screen frame. It means that the center of the box is translated by half screen height and half screen width, but the orientation is the same. The marker is published using a very low rate (a frame each 5 seconds) because it is supposed that the screen does not move with respect to the world frame.

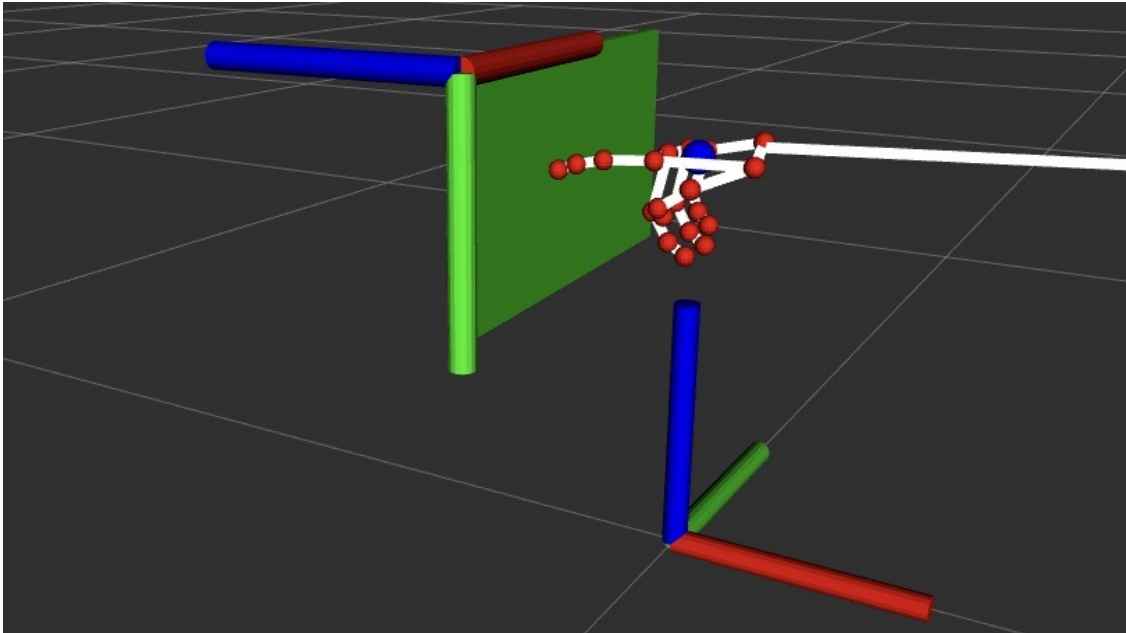


Figure 5.7: Virtual representation of the screen using Marker display in Rviz

**leap\_motion\_touch\_emulation** This executable does the main work of the touch emulation. It receives messages containing the position of the tip of the index with respect to the world frame, converts them into the screen frame,

using screen `Transform`, and then it verifies two conditions. First it verifies if the distance from the screen is less than the previously declared hovering distance (figure 5.8). If the condition holds, it broadcasts the position on the topic `hover_position` using `geometry_msgs/Point` messages (more information in the next section), otherwise it does nothing. Second, if the tip of the index is in the hovering zone then it verifies another condition. If the distance from the screen is smaller than the previously declared touching distance then the same message is broadcast also on the topic `touch_position`. It is of course easier to verify these two conditions, while working in the screen frame, because only values of *z* axis are compared.

In contrast to messages sent over the topic `hover_position`, which are sent continuously, messages on the `touch_position` topic are sent each and every single time the tip of the finger enters the touching zone. Moreover messages contain only values of *x* and *y* and no value for *z*, since the latter are not used in the following steps.

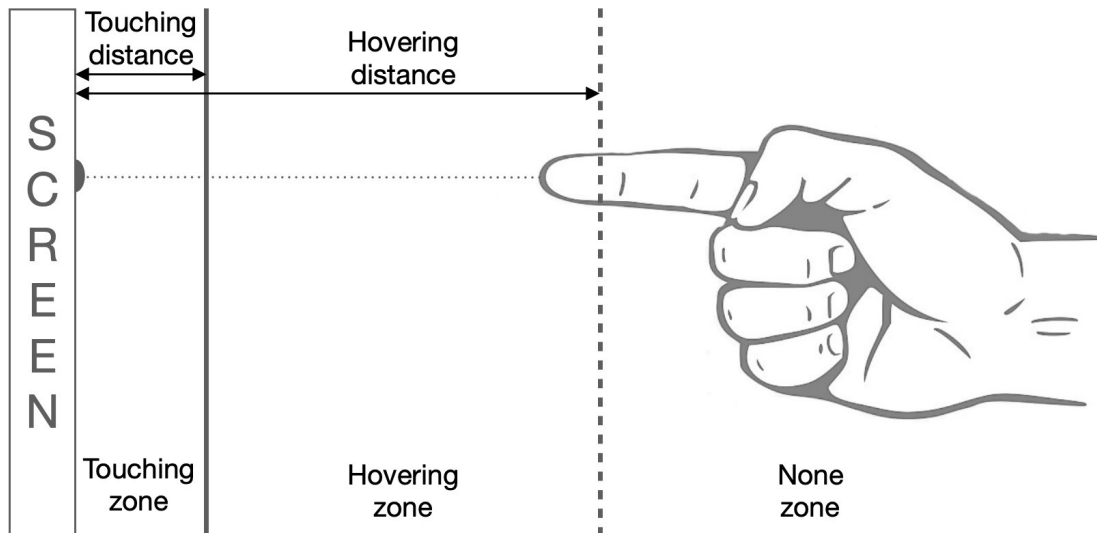


Figure 5.8: Graphical representation of the two virtual panels, the first closer to the screen, the second farther

### 5.4.3 Feedback marker

When immediate feedback is implemented in a system to show what the index is pointing, Augmented Reality is needed to represent a virtual marker in the real workspace. AR has been used a lot recently but very expensive devices are needed like an LCD projectors or Head Mounted Display.



In this study the purpose was to integrate live feedback using low cost devices. Since an LCD is used to see the real workspace and, in conjunction with Leap Motion, to emulate a touchscreen, this screen is also used to display a feedback marker in correspondence with the finger tip.

A study on the Layered Touch Panel [69] uses Screen Layer TP and IR Layer TP to implement a revolutionary touchscreen with the hovering event. Inspired by this Layered Touch Panel, this system simulates both touch panels using Leap Motion: the Screen Layer TP and the IR Layer TP are represented by two virtual planes parallel to the LCD with a distance respectively equal to `touch_distance` for the first virtual panel and `hover_distance` for the second one (figure 5.8). The first virtual panel is used to capture a touching event, the second for the hovering event.

The executable `draw_frames` is executed in the same machine that is connected to the LCD. As figure 5.6 shows, it subscribes to topics `hover_position` and `sensor/usb_cam/image_raw`, which is the topic where the camera publishes images. This file creates a window in which the images coming from the camera can be displayed. When an image is available, the node simply displays it or, if a `geometry_msgs/Point` message was previously published on the `hover_position` topic, a virtual marker is added to the image (figure 5.9). The marker is represented by a black circle having radius equal to the previously defined variable `pointer_radius` and it represents the point that the finger is going to touch. The position of the tip of the index (in meters) is scaled to the screen range and converted into pixels in order to have a direct correspondence.

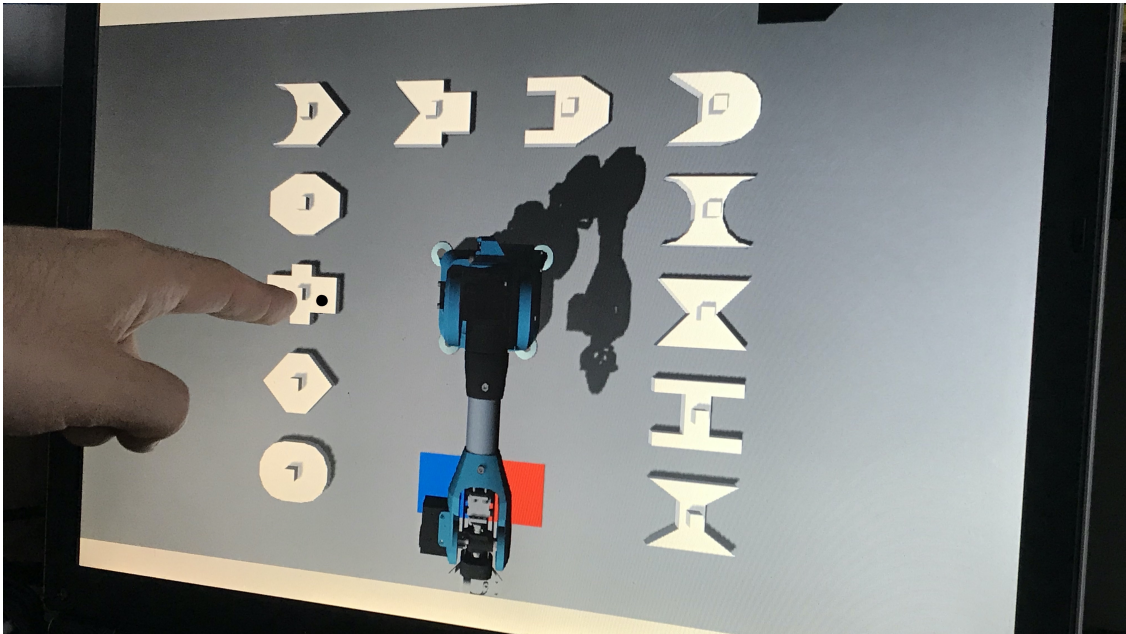


Figure 5.9: Representation of immediate feedback (black circle) used in this system

In this case, the position of the marker is simply the normal projection of the fingertip on the screen, unlike previous studies in which it was given by the intersection of the plane and the pointing direction. This decision was made to give the user the freedom to touch the screen not only using the fingertip but also using the finger pad, which would lead to projecting the virtual marker far from the fingertip if the pointing direction was used.

## 5.5 Object recognition

The object recognition group is used as a link between the touch emulation and the simulated system. All executables are included in the `find_object_2d.launch` file. In this file the user defines the topic from which to receive images, used as an environment from which to extract detected objects, and the folder path, where objects to detect are saved. The launch file is composed of three executables that work in pipeline (figure 5.11):

**find\_object\_2d** This executable is used to simulate the find-object application in ROS (see section 3.6.1). The GUI is used only to take some photos of the environment and save images of objects to detect. Each object has an ID, which is a unique increasing number, and the name of the photo is given by the ID followed by the extension.

At first, the light was set to come from above, such as a light coming from a bulb, in order not to project shadows. It was thought that shadows could be considered as noise for the find-object application.

At a later stage, the attention was put to photos of the pieces, which are taken in a grey scale because the application compares the difference of brightness and not colours. Since pieces do not have images on them but they all are of the same colour, shadows are an advantage because they allow a higher contrast along the shape, which makes an object easily recognisable. As a consequence the sunlight was set as if coming from a window of a building.

When an object is recognised, a coloured polygon is drawn around it (figure 5.10). After this process, it is possible to set the `gui` parameter in the launch file equal to false and not use the graphical interface, as it is not needed.

**find\_object\_2d\_filter** This executable works as a filter to not overload the system with useless information. It subscribes to the topic `objects` and publishes on the topic `objects_filtered` the same type of message, but only when there is a variation in the number of recognised objects. Theoretically one object can be removed and another can be added at the same time without having a variation in the number of recognised objects; practically, this is not possible due to the quite high frame rate (once every two seconds).



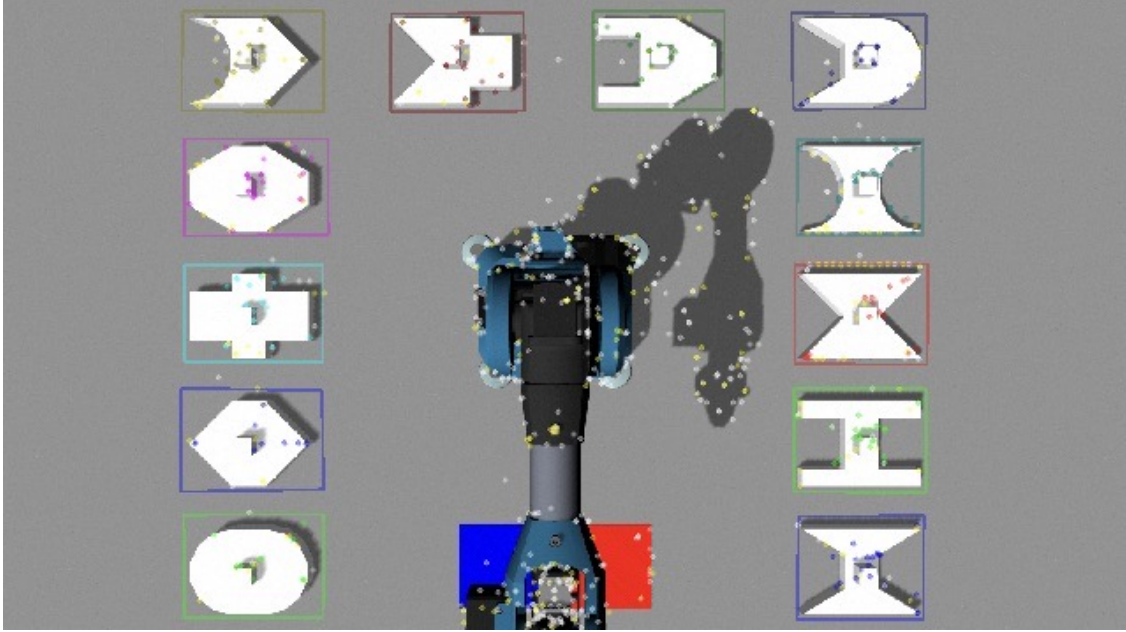


Figure 5.10: Visualisation of recognised objects (coloured polygons) using `find_object_2d` with the GUI

**touch\_object** This executable was designed to work as a funnel; the node which connects the find-object application with the touch emulation system. The ROS node subscribes to the topic `objects_filtered` and to the topic `/touch_emulation/touch_position` and publishes to the topic `pick_piece` the ID of the object, which has been recognised and touched, using a `std_msgs/Int32` message (figure 5.11).

When a message of recognised objects is received, using the homography matrix stored in the `std_msgs/Float32MultiArray` message, this node calculates the position of each recognised object with respect to the whole image and stores the four vertices of the polygon in an internal variable. When a message that indicates the touch position is received, this node verifies if the position of the tip of the index is in correspondence with one recognised object and, if it is true, the node sends the ID of the object that has been touched over the topic `pick_piece`. For simplicity, the check is considered verified if the position of the tip of the index is inside the rectangle inscribed in the polygon.

## 5.6 Assembly task

The assembly task is performed doing the pick and place operation with two objects. MoveIt is responsible to perform the pick and place operation using the Planning

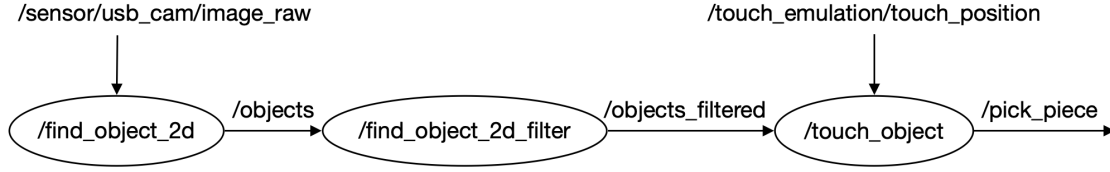


Figure 5.11: Graphical representation of object recognition group

Scene Monitor and the `move_group` node. A new file, `pick_place.cpp`, is developed adapting the `pick_place_tutorial.cpp` source file, provided in MoveIt tutorials (see section 3.5.4), to the current context.

### 5.6.1 Planning scene

The executable `pick_place` creates a ROS node and subscribes to `objects_filtered` topic. Every time a message is received, which means that a variation in the numbers of detected objects has occurred, the position of each detected object is stored in an internal variable. The position is given in terms of  $x$  and  $y$  (the height is the same for all objects) and it is retrieved from the Parameter Server. It was assumed that the initial position of an object with a given ID was always the same because an inexpensive webcam cannot locate an object in the 3D space. This is a reasonable assumption if the assembly task is the last step of an industrial chain, in which other robots are responsible for the location of the pieces.

The planning scene is composed of a horizontal plane, which represents the top surface of the table so that the robot does not move under the  $x$ - $y$  plane, and the representations of pieces placed around the robot in a horseshoe shape (figure 5.12). In contrast to the simulation in Gazebo, in which objects are all different from one another, in the planning scene, for simplicity, objects are all the same.

As explained before, the concave and convex shapes of each piece are inscribed in a parallelepiped. Starting from this, initially, all pieces were represented by a parallelepiped and a smaller cube above it, in order to be a general shape in which each piece can be inscribed. Doing this, MoveIt was not able to place the second piece in the second location, because there was an overlapping part between the two pieces, which led to a collision.

To avoid the collision, each object is represented only by the central part which is the same for all pieces. As a consequence, the additional concave or convex parts are considered like extensions which characterise the piece and they are seen as such only by the human operators. It means that the user alone has the responsibility to check if two pieces match each other, because from the robot point of view, pieces are all identical.

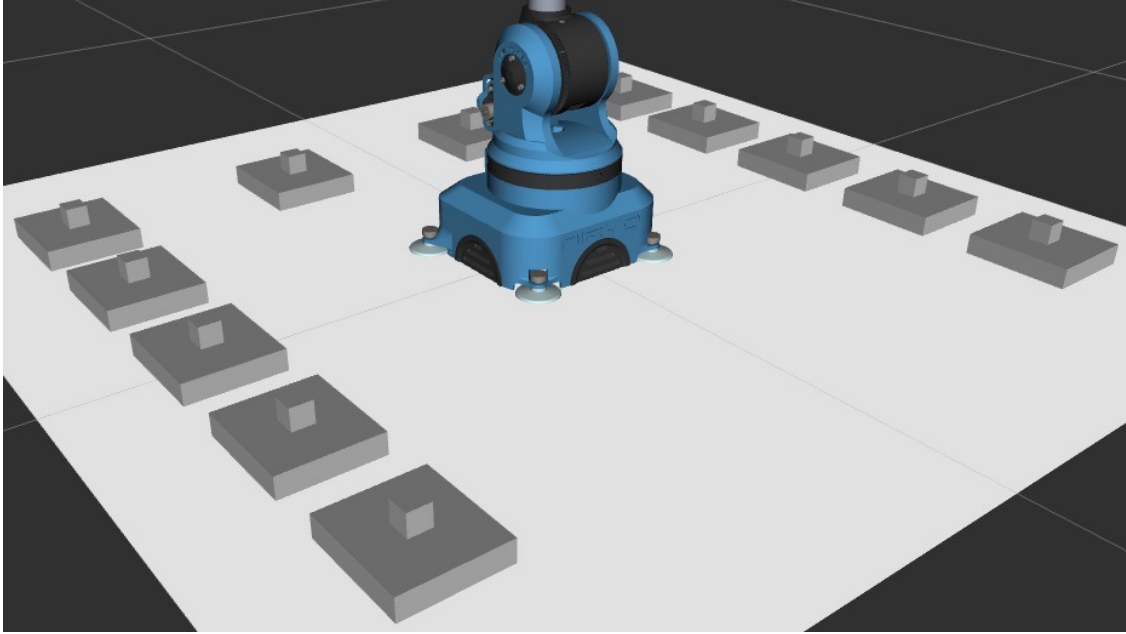


Figure 5.12: Representation of the planning scene (white objects), composed of the horizontal plane and 12 pieces, in Rviz using the PlanningScene display

### 5.6.2 Pick and place operation

Each time an object is touched on the screen a pick signal is emitted. The executable maintains two internal variables in which the ID of the objects can be saved to perform the pick and place operation.

At the beginning, only the horizontal plane is part of the planning scene. Only when the first pick request is received by the node, the pieces are added to the planning scene. These pieces correspond to the last detected objects and they do not change during the assembly task. This was made for two reasons: the first is to avoid a system overload because, even if a real piece is not removed, the robot, during the pick or place operation, can move above that piece which can not be recognised by the find-object application, therefore removed from the planning scene and then added again; the second reason is that when the robot moves on the top of the piece to pick it, the piece is no longer recognised by the find-object application, therefore the robot “does not see” the object and an error is generated.

Successively, the robot grasps the little cube on top of the touched object and lifts it up vertically for a distance of 10 cm.

During the place operation, using the variables in which IDs are stored, if no objects are picked up before, the piece grasped by the end effector is placed in the first location (blue rectangle); otherwise, if only one variable contains the ID, the piece is placed in the second location (red rectangle).

In the final phase, where the two objects are assembled and constitute a single

piece, this has to be removed manually from the workspace.

## 5.7 Simulation

During the simulation all blocks, which were previously developed and executed separately from one another, are now run all together.

For a better result, two machines were used for the simulation, in order to have higher performances.

In the first machine, to which the Leap Motion controller is linked, the ROS core is executed, followed by LeapControlPanel and the leap daemon. Then the demo.launch file from the leap\_motion package is launched in order to recognise hands.

In the meanwhile, in the second machine, the gazebo.launch file is launched. Firstly, this file uploads both the xacro files of the robot, of the camera and of the pieces and two yaml files containing the parameters of the controllers and the position of the pieces on the Parameter Server. Secondly, it launches an instance of Gazebo with an empty environment and populates it with all models previously uploaded. Finally, it loads the controllers to command the robot, and the robot\_state\_publisher to communicate the current state of the robot. Still in the second machine, the draw\_frame executable is run and the find\_object\_2d.launch file is launched. At this moment, the draw\_frame executable shows images coming from the camera in a new window, without having the possibility to add the marker, firstly because the touch\_emulation system is not launched yet, secondly, because screen parameters are not set yet.

Coming back to the first machine, the calibration phase is started by running the screen\_edges\_definition executable. Pointing with the tip of the index, the position of the four screen corners is detected and parameters saved in the Parameter Server. At this point the draw\_frame executable knows how to convert the position (in meters) of the tip of the index to the position of the virtual marker in pixels. As it does not know when the finger is in the hovering zone yet, the touch\_emulation.launch file is launched.

The whole system is almost up. Leap Motion recognises the hands, the touch emulation system divides the space into the three zones, the Gazebo simulator is simulating the workspace and the find-object application recognises the pieces on the screen. It is possible to see the virtual marker on the LCD when the finger is in the hovering zone, but nothing happens if the tip of the finger touches the image of one piece on the display. To react to the touching event, the pick\_place executable is run on the first machine.

In the simulation 12 pieces were placed on the table, numbered from 1 to 12 starting from the bottom left corner and continuing in a clockwise order. The simulation was performed touching the object with ID equal to 9 on the screen, placed in the top right corner. It was picked up and placed on the blu rectangle

(figure 5.13). The second object touched on the screen was the one with ID equal to 5, in the top left corner. This piece was successfully picked up and placed on the red rectangle.

When the assembly task is finished, the objects have to be removed manually to clear the workspace.

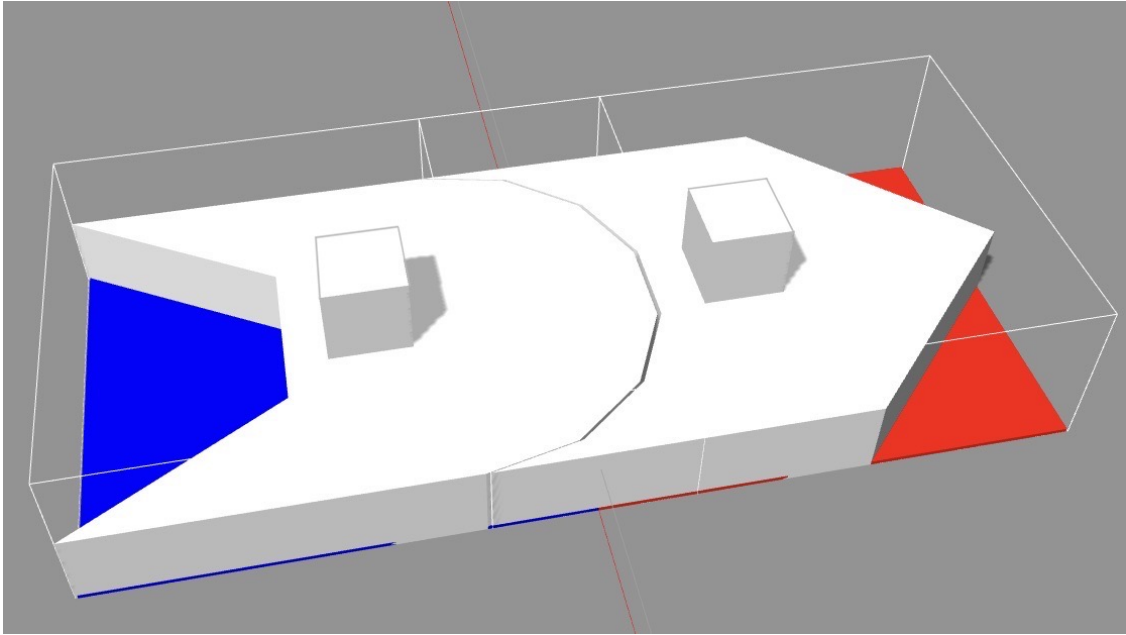


Figure 5.13: Representation of the completed assembly task in Gazebo simulator



## Chapter 6

# Practical implementation and experimental tests

After the simulation in Gazebo, experimental tests were performed employing the Niryo One robot, a computer to which the Leap Motion controller is attached and a Raspberry Pi 3 Model B+ to which a webcam and an LCD are linked.

### 6.1 Workspace

The workspace was organised in the same way as in the simulation with Gazebo. The robot was placed in the middle of the workspace and pieces surrounded it in a horseshoe shape (figure 6.1).

Pieces have particular shapes which were quite difficult to generate. They were made using cardboard, with the highest possible precision, by cutting and assembling them by hands. The error of the dimension of each piece was about 1 mm. As figure 6.2 shows, they were composed of a flat surface having the thickness of the cardboard (about 3 mm) and a square prism placed in the centre of the piece, having a 2 cm long base and a height such that the total height is equal to 4 cm.

Niryo One has four suckers at its bottom to be better fixed to the surface. Despite these suckers, the robot's stability is quite precarious when the arm is completely extended. The position of the pieces was modified so that the straight lines of the horseshoe were 30 cm distant from the center of the robot.

Since the pieces had the color of the cardboard, a white blanket was created with some white paper sheets and it was used as a uniform surface to increase the contrast between the pieces and the background. In the middle of this blanket the shape of the base of the robot was cut to let the robot stick to the surface below the blanket.

The webcam was placed above the workspace with the help of the e.DO collaborative robot by Comau. The height of the camera was set so that the workspace

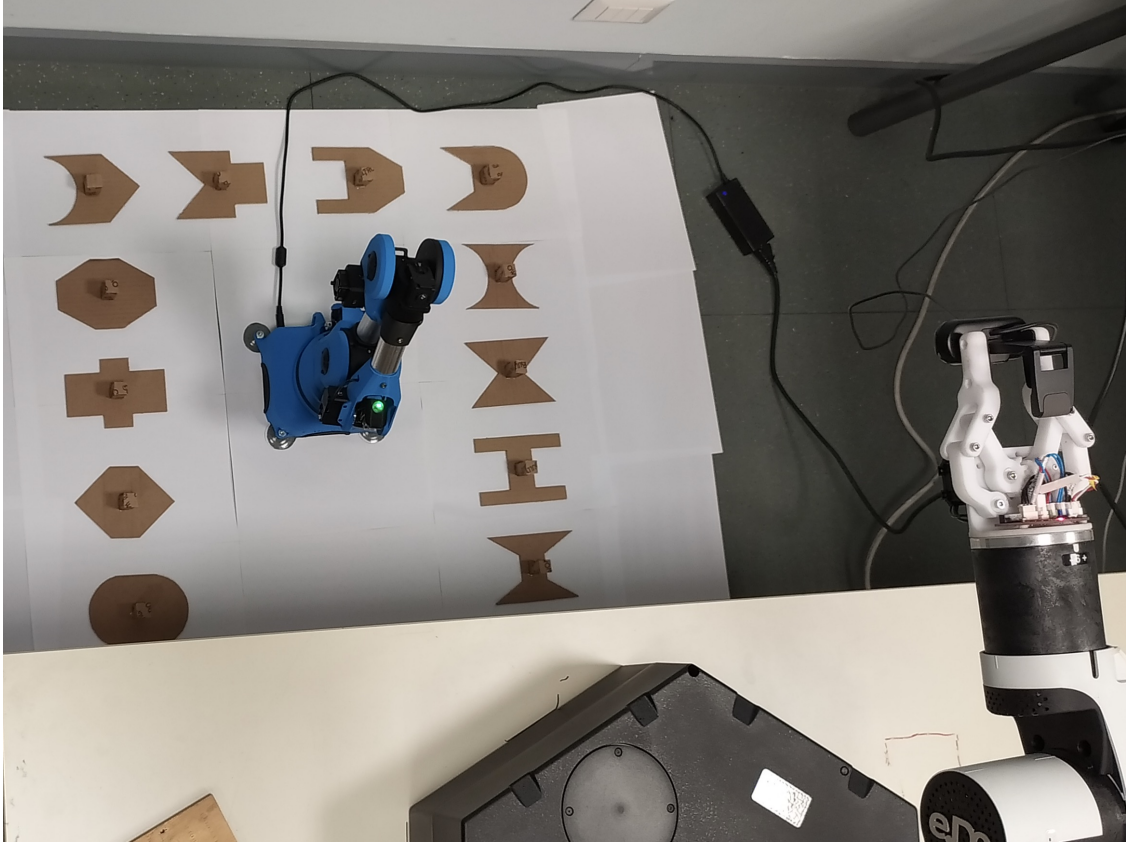


Figure 6.1: Representation of the workspace composed of the Niryo One robot in the middle, 12 pieces surrounding the robot and a webcam placed above it

was entirely visible in the LCD. As in the simulation in Gazebo, images of the workspace on the screen had the x positive axis going down and the y positive axis going to the right.

A lamp was then added to the workspace (see section 6.3.1). It was placed near the camera to generate shadows but in order not to be seen in the image stream.

For the physical simulation, in order to reproduce a realistic robotic system, computers were removed from the system and substituted with a board. In the system, therefore, two boards were present: one was the Raspberry Pi 3 Model B inside the robot and the second was a Raspberry Pi 3 Model B+ outside the robot. From here going ahead, to distinguish the two boards easily, the first one will be called “robot”, the second one will be simply called Raspberry.

Far from the workspace, the Raspberry, linked to the LCD and the webcam, was placed on a stable surface. The screen and the Leap Motion controller in front of it were placed on the same surface (figure 6.3).



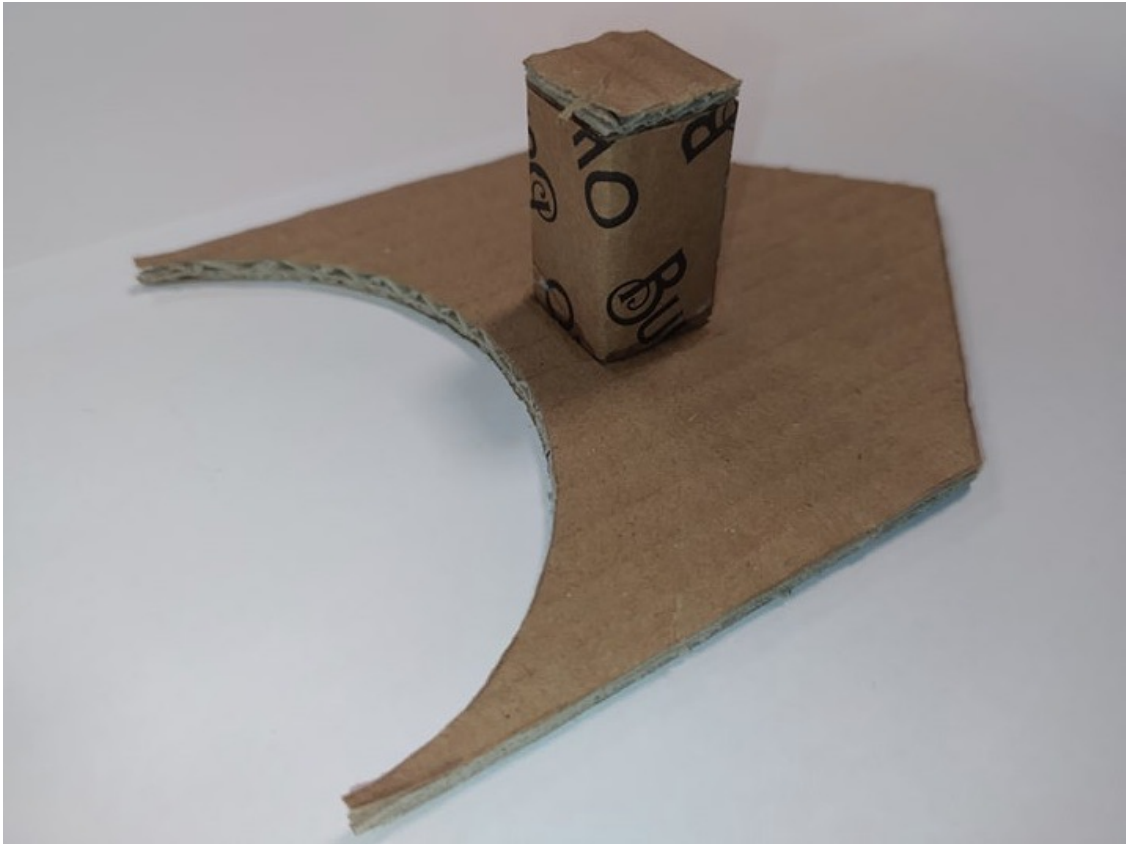


Figure 6.2: Detail of one piece made of cardboard used in the tests

## 6.2 Niryo One

The Niryo One base link, which is fixed to the surface, is already provided with the Raspberry Pi 3 Model B. This board is responsible for the correct mode of operation of all layers of the Niryo One ROS stack, from the hardware (LEDs, Wi-Fi, top button, digital I/O panel, motors, etc.) to the controllers, motion planning (MoveIt), commands (from both the graphical interface and the code) and external communication (websocket server or joystick).

### 6.2.1 Niryo One Studio

After the robot is powered on, it can be controlled through Niryo One Studio. This is a graphical interface the user can use to interact with the robot without any programming knowledge.

First of all the computer has to be connected to the same network of the robot (robot in hotspot mode or local network). Through this application, in the top right part (figure 6.4), the user can know the IP of the robot and connect to it

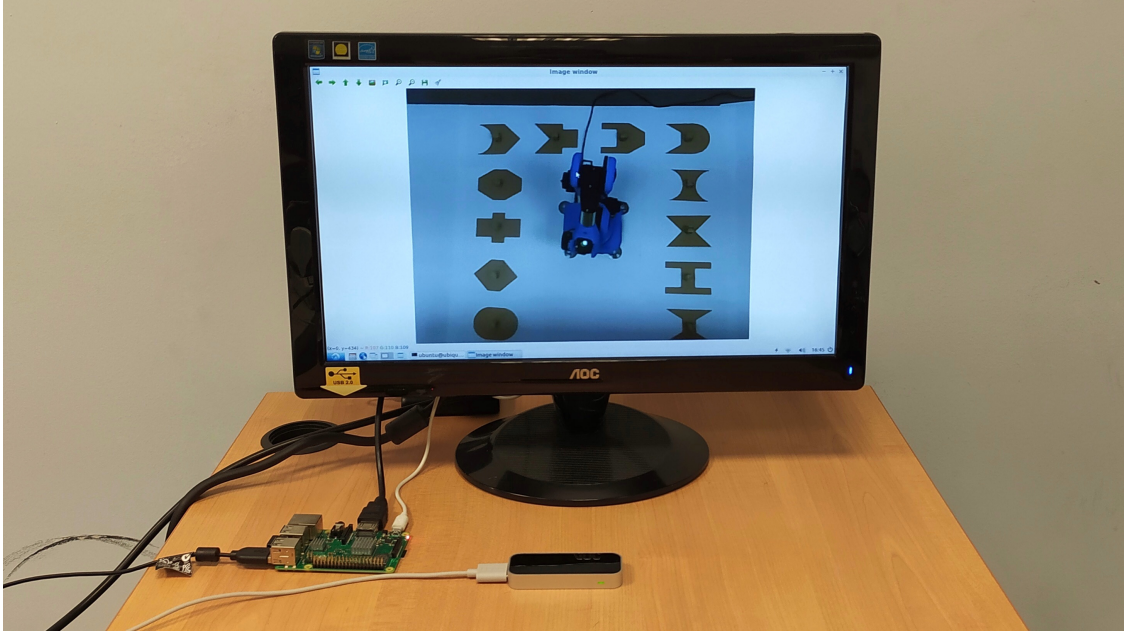


Figure 6.3: Representation of the work station composed of Raspberry Pi 3 Model B+ to which the webcam and the LCD are linked. The Leap Motion controller is placed in front of the screen.

without any difficulties.

Then the robot calibration is required. The calibration can be done in two ways: auto-calibration and manual calibration. The first is when the robot moves by itself and the axes, which need to be calibrated, move until they reach their maximum position, in order to calculate an offset for each motor. The second is when the user moves the axes in order to align the two arrows placed on the fixed and moving parts of the joints. To perform a manual calibration, at least one auto-calibration is needed because the first one uses the same precision of the second one. Manual calibration is recommended, firstly because it allows the robot to reach the same position with the same precision, secondly because it is faster and motors wear off less.

After the calibration, in the right part of the application (figure 6.4), the user can see the current state of the robot in a 3D virtual representation, read the values of all joints, the position and the orientation of the end effector and choose the arm maximum speed. A toggle button is also present to switch on and off the “learning mode”. This modality allows the deactivation of the torque on all motors so it is possible to move the robots by hand.

In the left part of the application, using a menu, the user can navigate through different panels which enable the user to control the robot, save some positions and sequences, and other specific actions like settings, calibration, hardware status,

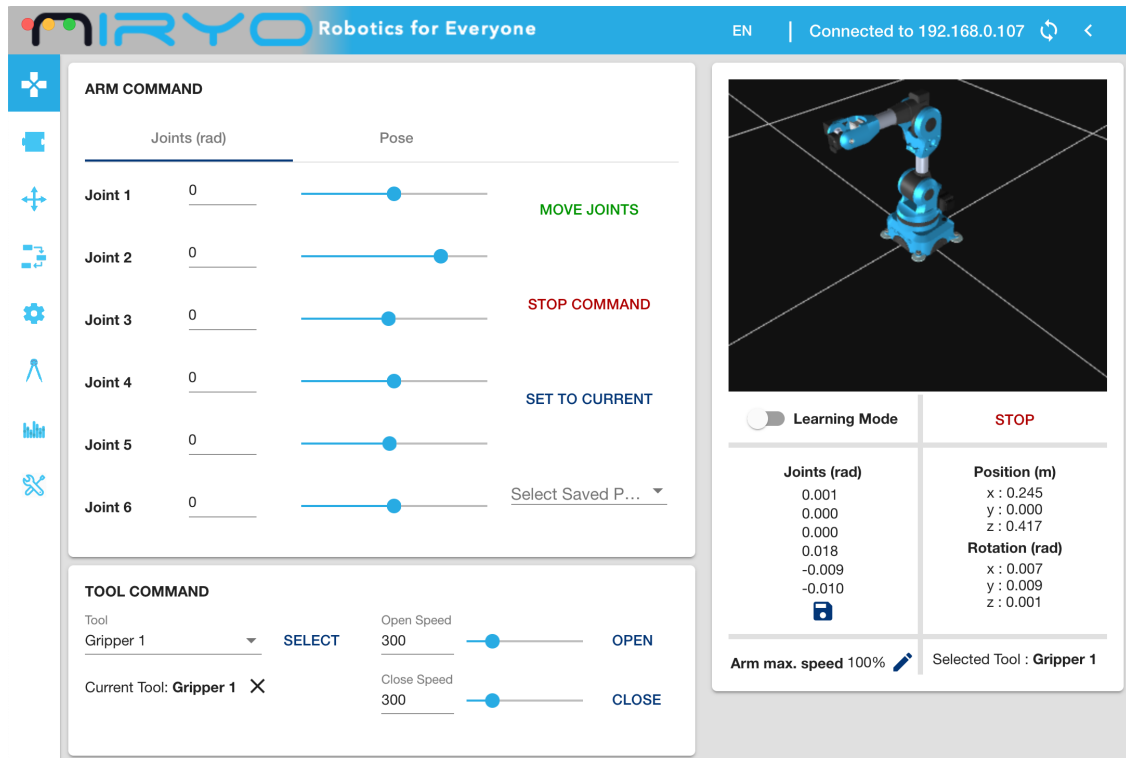


Figure 6.4: Niryo One Studio application showing the robot state on the right and arm and tool command on the left

debug and logs.

Selecting the control section from the left menu, two boxes are displayed. In the first box, at the top, two tabs are presented: the first allows the user to choose the value for each joint, the second allows the user to define the final pose of the robot. In both cases the user can set the current values, change them, send the command to the robot or stop it.

The second box, in the lower part, regards the tool. From this box the user can select which gripper is mounted, choose the open and close speed and send open and close commands.

The user can be completely unaware of the robot internal behaviour and can command it without any effort. This GUI allows the decoupling of robotics and programming knowledge. However this application uses the ROS network described in the next section, the same used for programming. All commands generated by Niryo One Studio are executed using services or implement a client-server protocol using the actionlib package (see section 6.4).

At the beginning of the practical tests, robot movements were hardly imposed, setting the grasping pose of each piece in the Niryo One Studio, in order to be sure that each piece is reachable.

### 6.2.2 Programming

At the startup of the robot, the Raspberry launches the ROS master and creates a network of ROS nodes which allow the robot to work properly.

NIRYO company provides an open source metapackage (`niryo_one_ros`) which contains a series of packages which compose the Niryo One ROS Stack.

The `niryo_one_bringup` package contains all the files needed to start the system. This operation can be done in two ways. The first one is through a service (`niryo_one_ros.service`), which is automatically started when the robot is switched on, works in the background and in this case it is not possible to modify the robot behaviour. The second one is through a launch file which creates the same network of nodes but works in the foreground, and in this case it is possible to modify the Niryo One ROS stack. It is necessary to make a distinction between two launch files: `rpi_setup.launch` and `desktop_rviz_simulation.launch`. The first file runs over the robot and launches `roslaunch`, for the connection with other components; `niryo_one_base`, which loads robot parameters; controllers, which manage the drivers, the tool interface and the state of the robot; `robot_interface`, which is responsible for the robot movements; and `user_interface`, which allows the interaction with Niryo One Studio. The second file is identical to the first one apart from the following aspects: it can be launched only on the computer, it uses Rviz to visualise the robot, it does not have the access to the hardware therefore fake controllers are used.

When the network is up, the robot can communicate with other components in hotspot mode, creating its own local network, or with a direct connection using an Ethernet cable, usb cable or ssh, for example to be linked to other boards, to a joystick or to a local network.

For example, Niryo One Studio is connected with the robot using ssh. It sends commands to Niryo One which uses a Websocket server to receive and process them.

## 6.3 Raspberry

For the experimental tests, it was decided to leave the Raspberry embedded in the robot “cleaned”, by not adding external code or modifying the Niryo One ROS Stack, using the external Raspberry instead to connect the sensors and the LCD with the system and run the required ROS nodes.

On the Raspberry it was decided to mount a robot development image made by Ubiquiti Robotics [71]. This is an image designed to make the development of ROS based robot applications easier, in fact it is a Linux image with ROS Kinetic Kame already installed.

The LCD was linked to the Raspberry, not only to use the terminal to execute commands, but also to visualise images coming from the camera through the `draw_frame` node. The screen type and its size are not a problem, because the

code was developed to be independent of them. However, the screen must be small enough to guarantee that the image of the workspace displayed on it is inside the Leap Motion field of view. During the simulation a 17 inch screen was used.

Also in this case the ssh protocol was used to let the Raspberry and the robot communicate over the ROS network. In both machines `ROS_IP` and `ROS_HOSTNAME` variables were set to have the IP of the current machine, while the `ROS_MASTER_URI` variable was set in all the machines to have the IP of the robot, since the ROS Master runs over it.

Many packages were downloaded to set up the Raspberry. Some of them like OpenCV and MoveIt are used by some nodes like `draw_frame` and `pick_place`, others, like the `niryo_one_ros` package, were downloaded only because they contain the definition of some messages.

### 6.3.1 Camera

Webcam C210 by Logitech [43] was used as a camera. It has a resolution of 640 x 480 pixel, it is very small and inexpensive.

After it was linked to the Raspberry, it worked well in the operating system but it was not able to share images and information on the ROS network. To let the webcam communicate with other nodes, the `usb_cam_node` executable from the `usb_cam` package was run. This node works as a driver which takes images from the camera and publishes them over the topic `/usb_cam/image_raw` topic. During the simulation in Gazebo, images coming from the simulated camera were published on the topic `/sensor/usb_cam/image_raw`. Nodes that subscribed to this topic were `draw_frames` and `find_object_2d`, therefore they were adapted to the new topic name.

Unlike in the simulation with Gazebo in which the camera had a resolution of 1920 x 1080 pixel that allows a good object detection, the camera used in the real simulation has a lower resolution and this causes a not so good behaviour of the Find-Object application. Since the resolution is lower, colours are not very close to the reality and small changes in colour, like light shadows, are not perceived. The light in the laboratory, coming from the windows, was intense enough to illuminate the pieces in the workspace but not enough for the generation of dark shadows. The light coming from the ceiling was more intense than the sunlight but it generated multiple shadows for each object. To solve this problem a lamp was used. The light bulb, positioned close to the workspace, near the webcam, was intense enough to generate shadows. Due to its position, shadows were not only along one direction but they had a radial direction. As in Gazebo, the best solution would have been to have a central light positioned far from the workspace but intense enough to generate dark shadows along one single direction. However the use of the lamp allows the Find-Object application to generate more than 1000 detectors, which are enough for a correct object recognition, compared to about 700 detectors without



the use of the lamp.

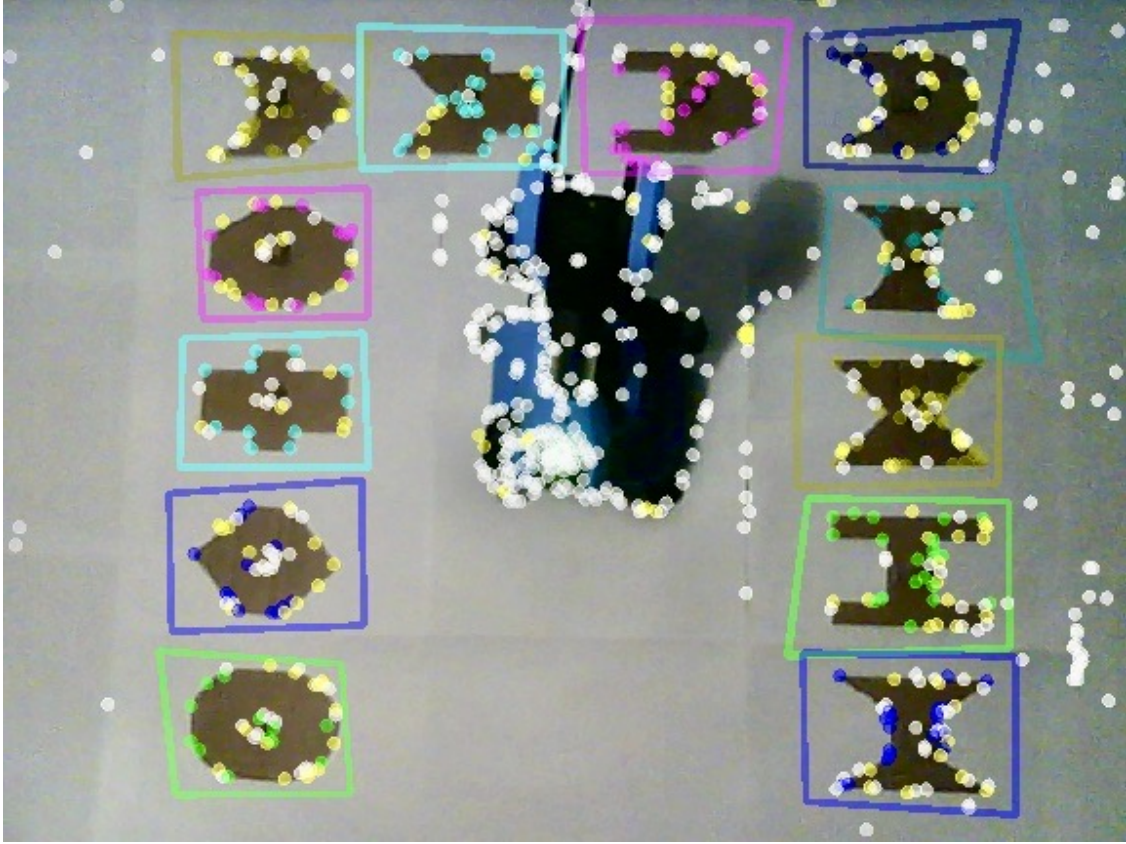


Figure 6.5: Visualisation of the Find-Object application with the real pieces recognition

### 6.3.2 Leap Motion

As mentioned before, initially the idea was to replace the two computers used in the simulation with Gazebo with the Raspberry. This meant that not only the LCD and the webcam had to be connected with the board but also the Leap Motion controller.

After the download of the Leap Motion software, when the installation of this component over the Raspberry was attempted an error was encountered. The problem was that Leap Motion installation is available only for AMD or Intel Core processors, while all Raspberry boards have an ARM processor. The one and only solution available was to insert again the computer in the system and link the Leap Motion controller to it. This was done because only with the installation it is possible to read the leap libraries that are available for Leap Motion. Therefore the `demo.launch` and `touch_emulation.launch` files from the `leap_motion` package

were run on the computer, which has an Intel Core processor, because both run executables that need `leap.h`. Also the `screen_edges_definition` was executed on the computer, even if the screen was linked to the Raspberry.

## 6.4 Pick and place

After the first approach in which Niryo One Studio was used to move the robot, the code to perform a pick and place operation used in the simulation with Gazebo was adapted for the real robot.

Going step by step, the first one was to adapt the previously developed code to the desktop simulation. During this phase it was discovered that the robotic arm moved well but the gripper did not perform the opening and closing actions. This occurred because, unlike the behaviour in Gazebo simulator in which both robotic arm and the gripper used a `joint_state_controller`, the Niryo One ROS Stack uses this kind of controller only to move the robotic arm and it provides an `ActionClient/ActionService` to perform the operation on the gripper.

The `actionlib` package provides the `ActionClient` and the `ActionServer` which implement a client-server communication between ROS nodes. The client and the server create an exchange of information as in a service protocol, but with the possibility not only to send a goal and receive the result, but also to cancel the goal and receive feedback about the task progression during the execution. As for services, a message has to be defined. In this case the message contains three fields: the goal, the result and the feedback, separated by “---”.

In this case the `Tool.action` is already defined and available in the `niryo_one_msg` package. Using this action, the `ActionClient` sends a goal containing the tool ID, the command type (open or close) and the velocity; the `ActionServer` in the robot executes the command.

Consequently the pick and place functions from `MoveGroupInterface`, which enable to open and close the gripper selecting the gripper joint, are not usable yet. To solve this problem a different approach was used. When the node is generated, it creates an `ActionClient` that tries to contact the `ActionServer` of the gripper and, if it does, it uses a service to select the gripper ID. Then the functions `setPoseTarget` and `move from MoveGroupInterface` were used. The first function allows the definition of the final pose of the robotic arm, using the corresponding values with the piece touched on the screen, the second one permits to perform the movement. These two functions substitute the functions `pre_grasp_approach`, `pre_grasp_posture`, `grasp_pose`, `grasp_posture`, `post_grasp_retreat` (see section 3.5.4) used in the pick function. As described above, the `ActionClient` was used to send a message to open the gripper before the robot reaches the grasp pose and a message to close the gripper when it is in position. A similar procedure was used also to adapt the function `place` to put the piece in the first place or in the second one, sending also a message to open the gripper, using the `ActionClient`, to release the piece.

The remaining part of the code, which adds the pieces to the Planning Scene, was left unchanged.

The second step was to recreate the simulation from the desktop to the reality. The Niryo One ROS Stack was run in the robot and the adapted pick\_place executable in the Raspberry. At first it did not work. The adapted code and Niryo One ROS Stack were verified many times far and wide but no solution was found. In a second moment two issues were discovered. First, the calibration is mandatory every time the robot is switched on, otherwise the commands do not reach motors. Second, unlike the Niryo One Studio application, which uses ActionClient to send commands to the robot and it is able to switch off the “learning mode” automatically and move the robot, the move\_group node does not have this ability and, even if it was able to correctly compute the motion plan, it was not able to move the robotic arm because the torque on all motors were deactivated. After these two little issues were discovered, it was seen that the code worked quite well also in the practical implementation.

## 6.5 Experimental test

During an experimental test (figure 6.6) the assembly of piece 5 and piece 10 (counting pieces clockwise) was attempted. Firstly the image of piece 5 was touched on the screen. The robot moved from the start pose to the grasping pose above this piece. Then it opened the gripper, went down following a straight line, closed the gripper to pick the piece and moved up following a straight line. After that the robot moved above the first place position. Following the same pattern the robot went down following a vertical movement, opened the gripper, released the piece and then went up following the same vertical movement. Secondly the image of piece 10 was touched on the screen, with the help of the virtual feedback marker. As a consequence the robot moved on the top of this piece to grasp it. It performed the pick operation and moved the piece on the top of the second location to perform the place operation. The assembly task was concluded correctly with no error as it is shown in the video that was made during the test [19]. Sometimes in the final assembly some pieces overlap. This happens because pieces are imprecise while in the virtual simulation pieces are assembled with high precision. Moreover, as previously explained, the robot is not able to recognise if two pieces match each other: the pick and place operation was tested with a wrong assembly task and the system worked well up to the picking operation of the second piece, making a wrong placing operation because the shapes of the two pieces did not match.



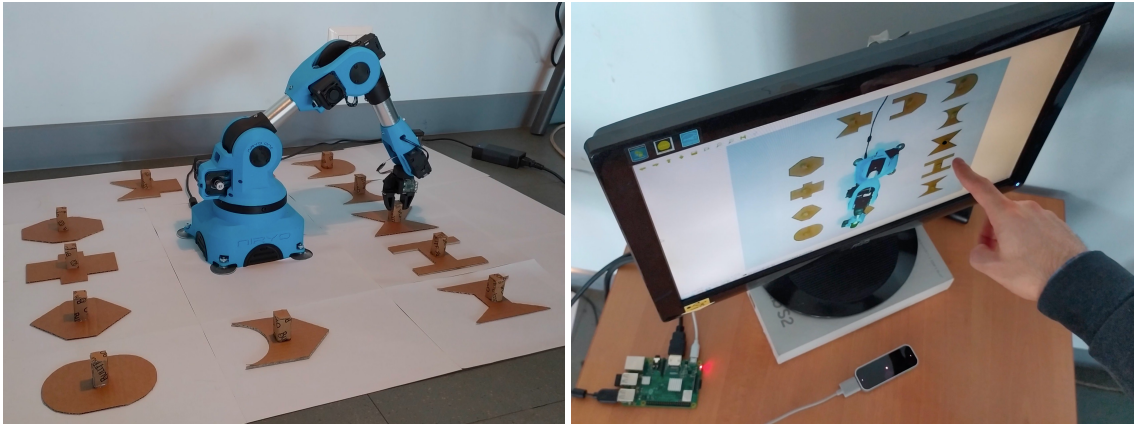


Figure 6.6: On the left, the robot picking the second piece of the assembly task, on the right the hand pointing the same piece on the screen



# Chapter 7

## Conclusion

This master thesis presented a new way for Human-Robot Collaboration, to pick and place objects in an industrial assembly task. The system simulates a touch emulation using inexpensive devices like Leap Motion, for hands recognition, a camera and an LCD.

### 7.1 Final remarks

In this system the following advantages were observed:

**performance** The ROS core is very simple but also very efficient. ROS was essential in this robotic system to let the robot interact with sensors and humans. ROS allowed the reuse of the code developed by third companies and other developers without reducing the performance.

**scalability** This system was developed using Niryo One robotic arm. This robot is an educational robot, it is small, good for prototyping, and it behaves as an industrial robot. Potentially, Niryo One software components could be substituted by another robot and the system should work in the same way.

**inexpensiveness** This advantage was thought as necessary a prerequisite during the developing face. For this reason Leap Motion and a webcam were chosen as sensors. An alternative idea was to use two cameras in place of the Leap Motion controller but this would have led to a frequent calibration phase. The advantages of using Leap Motion is to have a depth camera with an integrated software able to recognise human hands.

In addition Niryo One is very cheap with respect to other collaborative robots and it can be 3D printed, so the cost can be even reduced.

**easy to perform** The system was developed using Gazebo simulator. This allowed the simulation of different scenarios with little effort. Moreover, the

simulation is a solution in a situation in which the environment is not available, like in the case of this project because the robot was not accessible until October due to COVID-19.

In addition ROS provides some tools like rqt, TF and Rviz that allows the user to visualise how the system behaves and easily interact with it using GUIs.

**versatility** The touch emulation developed in this system allows users to interact with the robot using their hands. Usually in industrial processes, human operators have to wear gloves for safety reasons. Gloves do not allow the use of a touchscreen device but of course this system can be used either with or without gloves.

**adaptability** In this system 12 different pieces were used matching one another. These pieces have the same pattern but they are recognised by the find-object application as all different. Each object has an ID and it is identified using an image having the ID as a name. Adding, removing or changing some objects, which have a unique ID, allows the system to adapt to every assembly task.

**modularity** The proposed system can be conceptually divided into five main blocks: Leap Motion, touch emulation, find objects, MoveIt and Gazebo. All these blocks are linked to work together but each block is independent from the others. As a consequence, each block can be potentially substituted with another one which behaves in the same way, using a different technology or a different code.

The following disadvantages were also observed:

**object position** Since a single camera is placed above the robot, the find-object application is not able to recognise the position of each piece in the 3D space but only its location in the whole image. This led to always assigning the same position to the same piece. This is a kind of limitation in the system but it works in situations in which other robots perform the work done before the assembly task. These robots could also be programmed to place a specific piece in the same place.

**LM field of view** The main drawback in this system is given by the small interaction space. Even if the human operator is free to move in the workspace, their hand is recognised only if it is inside the Leap Motion field of view. Moreover, since during the calibration phase the user has to touch the corners of the LCD, also the screen must be inside the Leap Motion field of view.

**LM errors** Even though the Leap Motion controller is very precise in the recognition of human hands and their motion, it is not so precise in measuring distances. Since, during the calibration phase, the Leap Motion controller is used to detect the screen dimensions, this error cannot be considered to be of

little value. The error reported is about 1 cm less every 10 cm. In this application, this error can be avoided by touching the external corner of the screen instead of touching the window corners and, of course, the virtual marker added as feedback is considered essential.

## 7.2 Future developments

The system has been developed in order to be as simple as possible and potentially integrable in a real industrial task. Some improvements are here proposed:

- The system could allow the user to have more power during the assembly task. This improvement could be obtained by recognising not only the pointing at part gesture but also other gestures. For example the circle gesture could be used to rotate the piece to one side or the other, or another gesture could be used to cancel the current operation. Another way could be to give the user the possibility to choose the location where to place the piece.
- The system could be improved by adding more sensors. The camera placed above the robot could be substituted with a depth camera to recognise objects, their position and the possible grasping pose. Another possibility is to add eye tracking sensors to move the robot in the proximity of the object so as to make the pick operation last less time.
- The touch emulation system could be made universally possible. If the Leap Motion controller did not need to be placed on a surface perpendicular to the screen, this would lead to an easier physical setup of the system.
- The system could be equipped with some neural networks. Neural networks could be used in many ways. One possibility is to improve the pick operation of the second piece, moving the robot in the proximity of those pieces that match with the first one.



# Bibliography

- [1] 3D SYSTEMS website: <https://www.3dsystems.com/haptics>.
- [2] Angelo O Andrisano, Francesco Leali, Marcello Pellicciari, Fabio Pini, and Alberto Vergnano. Hybrid reconfigurable system design and optimization through virtual prototyping and digital manufacturing tools. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 6(1):17–27, 2012.
- [3] Stephanie Arévalo Arboleda, Tim Dierks, Franziska Rücker, and Jens Gerken. There’s more than meets the eye: Enhancing robot control through augmented visual cues. In *Companion of the 2020 ACM/IEEE International Conference on Human-Robot Interaction*, pages 104–106, 2020.
- [4] Isaac Asimov. *I, robot*. Spectra, 2004.
- [5] Rowel Atienza and Alexander Zelinsky. Intuitive human-robot interaction through active 3d gaze tracking. In *Robotics Research. The Eleventh International Symposium*, pages 172–181. Springer, 2005.
- [6] HMRT Bandara, MMSN Edirisighe, BLPM Balasooriya, and AGBP Jayasekara. Development of an interactive service robot arm for object manipulation. In *2017 IEEE International Conference on Industrial and Information Systems (ICIIS)*, pages 1–6. IEEE, 2017.
- [7] Sonia Mary Chacko and Vikram Kapila. Augmented reality as a medium for human-robot collaborative tasks. In *2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, pages 1–8. IEEE, 2019.
- [8] Sonia Mary Chacko and Vikram Kapila. An augmented reality interface for human-robot interaction in unconstrained environments. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2019*, pages 3222–3228. Institute of Electrical and Electronics Engineers Inc., 2019.
- [9] Siam Charoenseang and Tarinee Tonggoed. Human-robot collaboration with augmented reality. In *International Conference on Human-Computer Interaction*, pages 93–97. Springer, 2011.
- [10] CJ Chen, SK Ong, AYC Nee, and YQ Zhou. Haptic-based interactive path planning for a virtual robot arm. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 4(2):113–123, 2010.
- [11] Pholchai Chotiprayanakul, Dalong Wang, Ngaiming Kwok, and Dikai Liu. A

- haptic base human robot interaction approach for robotic grit blasting. In *IS-ARC 2008-Proceedings from the 25th International Symposium on Automation and Robotics in Construction*, 2008.
- [12] Mustafa Çoban and Gökhan Gelen. Wireless teleoperation of an industrial robot by using myo arm band. In *2018 International Conference on Artificial Intelligence and Data Processing (IDAP)*, pages 1–6. IEEE, 2018.
- [13] Comau website: <https://www.comau.com/en>.
- [14] Alessandro De Luca and Fabrizio Flacco. Integrated control for phri: Collision avoidance, detection, reaction and collaboration. In *2012 4th IEEE RAS & EMBS International Conference on Biomedical Robotics and Biomechatronics (BioRob)*, pages 288–295. IEEE, 2012.
- [15] Lucio Tommaso De Paolis. A touchless gestural platform for the interaction with the patients data. In *XIV Mediterranean Conference on Medical and Biological Engineering and Computing 2016*, pages 880–884. Springer, 2016.
- [16] Davis Dhivin, James Jose, and Rao R Bhavani. Bilateral tele-haptic interface for controlling a robotic manipulator. In *2017 International Conference on Intelligent Computing, Instrumentation and Control Technologies (ICICICT)*, pages 1614–1620. IEEE, 2017.
- [17] Andreas Dünser, Martin Lochner, Ulrich Engelke, and David Rozado Fernandez. Visual and manual control for human-robot teleoperation. *IEEE computer graphics and applications*, 35(3):22–32, 2015.
- [18] eSense website: <https://www.esense.io>.
- [19] Experimental test link: <https://www.youtube.com/watch?v=Sy5h8Yyz0Jw>.
- [20] HC Fang, SK Ong, and AYC Nee. A novel augmented reality-based interface for robot path planning. *International Journal on Interactive Design and Manufacturing (IJIDeM)*, 8(1):33–42, 2014.
- [21] find-object package: <https://github.com/introlab/find-object>.
- [22] A Freddi, M Goffi, S Longhi, A Monteriú, D Ortenzi, and D Proietti Pagnotta. A gestures recognition based approach for human-robot-interaction. In *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*, pages 27–28. IEEE, 2018.
- [23] Yuxiang Gao and Chien-Ming Huang. Pati: a projection-based augmented table-top interface for robot programming. In *Proceedings of the 24th International Conference on Intelligent User Interfaces*, pages 345–355, 2019.
- [24] Lee Garber. Gestural technology: Moving interfaces in a new direction [technology news]. *Computer*, 46(10):22–25, 2013.
- [25] Gazebo website: <http://gazebo.org>.
- [26] Brian Gleeson, Karon MacLean, Amir Haddadi, Elizabeth Croft, and Javier Alcazar. Gestures for industry intuitive human-robot communication from human observation. In *2013 8th ACM/IEEE International Conference on Human-Robot Interaction (HRI)*, pages 349–356. IEEE, 2013.



- [27] LL Gong, SK Ong, and AYC Nee. Projection-based augmented reality interface for robot grasping tasks. In *Proceedings of the 2019 4th International Conference on Robotics, Control and Automation*, pages 100–104, 2019.
- [28] Ryosuke Hanyu, Toshiaki Tsuji, and Shigeru Abe. Command recognition based on haptic information for a robot arm. In *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 4662–4667. IEEE, 2010.
- [29] Abdelfetah Hentout, Mustapha Aouache, Abderraouf Maoudj, and Isma Akli. Human–robot interaction in industrial collaborative robotics: a literature review of the decade 2008–2017. *Advanced Robotics*, 33(15-16):764–799, 2019.
- [30] Mohammed A Hussein, Ahmed S Ali, FA Elmisery, and R Mostafa. Motion control of robot by using kinect sensor. *Research journal of applied sciences, engineering and technology*, 8(11):1384–1388, 2014.
- [31] Introduction to digital filters website: <https://ccrma.stanford.edu/~jos/filters/>.
- [32] Anja Jackowski, Marion Gebhard, and Roland Thietje. Head motion and head gesture-based robot control: A usability study. *IEEE Transactions on Neural Systems and Rehabilitation Engineering*, 26(1):161–170, 2017.
- [33] Inmo Jang, Joaquin Carrasco, Andrew Weightman, and Barry Lennox. Intuitive bare-hand teleoperation of a robotic manipulator using virtual reality and leap motion. In *Annual Conference Towards Autonomous Robotic Systems*, pages 283–294. Springer, 2019.
- [34] Lentin Joseph. *learning Robotics using python*. Packt Publishing Ltd, 2015.
- [35] Lentin Joseph. *ROS Robotics Projects*. Packt Publishing Ltd, 2017.
- [36] Lentin Joseph. *Robot Operating System (ROS) for Absolute Beginners*. Springer, 2018.
- [37] Lentin Joseph and Jonathan Cacace. *Mastering ROS for Robotics Programming: Design, build, and simulate complex robots using the Robot Operating System*. Packt Publishing Ltd, 2018.
- [38] Dorothea Koert, Susanne Trick, Marco Ewerton, Michael Lutter, and Jan Peters. Online learning of an open-ended skill library for collaborative tasks. In *2018 IEEE-RAS 18th International Conference on Humanoid Robots (Humanoids)*, pages 1–9. IEEE, 2018.
- [39] Dennis Krupke, Frank Steinicke, Paul Lubos, Yannick Jonetzko, Michael Görner, and Jianwei Zhang. Comparison of multimodal heading and pointing gestures for co-located mixed reality human-robot interaction. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 1–9. IEEE, 2018.
- [40] Labbé, M. Find-Object. <http://introlab.github.io/find-object>, 2011. accessed YYYY-MM-DD.
- [41] Leap Motion developer website: <https://developer-archive.leapmotion.com/documentation/v2/cpp/index.html>.
- [42] Leap Motion package: [https://github.com/ros-drivers/leap\\_motion](https://github.com/ros-drivers/leap_motion).

- [43] Logitech. Webcam c210 specifications.
- [44] Masaaki Mase and Kenji Suenaga. Real-time detection of pointing actions for a glove-free interface. In *In IAPR Workshop on Machine Vision Applications*. Citeseer, 1992.
- [45] Paul Milgram and Fumio Kishino. A taxonomy of mixed reality visual displays. *IEICE TRANSACTIONS on Information and Systems*, 77(12):1321–1329, 1994.
- [46] Théo Moulières-Seban, David Bitonneau, Jean-Marc Salotti, Jean-François Thibault, and Bernard Claverie. Human factors issues for the design of a cobotic system. In *Advances in human factors in robots and unmanned systems*, pages 375–385. Springer, 2017.
- [47] MoveIt website: <https://moveit.ros.org>.
- [48] NIRYO. Niryo one mechanical specifications, 2018.
- [49] Niryo One package: [https://github.com/NiryoRobotics/niryo\\_one\\_ros\\_simulation](https://github.com/NiryoRobotics/niryo_one_ros_simulation).
- [50] Niryo One website: <https://niryo.com>.
- [51] Henry Odoemelem, Alexander Hölzemann, and Kristof Van Laerhoven. Using the esense wearable earbud as a light-weight robot arm controller. In *Proceedings of the 1st International Workshop on Earable Computing*, pages 26–29, 2019.
- [52] OpenCV website: <https://opencv.org>.
- [53] Raja Parasuraman, Thomas B Sheridan, and Christopher D Wickens. A model for types and levels of human interaction with automation. *IEEE Transactions on systems, man, and cybernetics-Part A: Systems and Humans*, 30(3):286–297, 2000.
- [54] Sungman Park, Yeongtae Jung, and Joonbum Bae. A tele-operation interface with a motion capture system and a haptic glove. In *2016 13th International Conference on Ubiquitous Robots and Ambient Intelligence (URAI)*, pages 544–549. IEEE, 2016.
- [55] Michael Peshkin and J Edward Colgate. Cobots. *Industrial Robot: An International Journal*, 26(5):335–341, 1999.
- [56] Camilo Perez Quintero, Romeo Tatsambon Fomena, Azad Shademan, Oscar Ramirez, and Martin Jagersand. Interactive teleoperation interface for semi-autonomous control of robot arms. In *2014 Canadian Conference on Computer and Robot Vision*, pages 357–363. IEEE, 2014.
- [57] Camilo Perez Quintero, Romeo Tatsambon, Mona Gridseth, and Martin Jägersand. Visual pointing gestures for bi-directional human robot interaction in a pick-and-place task. In *2015 24th IEEE International Symposium on Robot and Human Interactive Communication (RO-MAN)*, pages 349–354. IEEE, 2015.
- [58] Robot Operating System website: <http://wiki.ros.org>.
- [59] Eric Rosen, David Whitney, Elizabeth Phillips, Gary Chien, James Tompkin, George Konidaris, and Stefanie Tellex. Communicating and controlling robot

- arm motion intent through mixed-reality head-mounted displays. *The International Journal of Robotics Research*, 38(12-13):1513–1526, 2019.
- [60] Maram Sakr, Waleed Uddin, and HF Machiel Van der Loos. Orthographic vision-based interface with motion-tracking system for robot arm teleoperation: A comparative study. In *Companion of the 2020 ACM/IEEE International Conference on Human-Robot Interaction*, pages 424–426, 2020.
- [61] Shin Sato and Shigeyuki Sakane. A human-robot interface using an interactive hand pointer that projects a mark in the real work space. In *Proceedings 2000 ICRA. Millennium Conference. IEEE International Conference on Robotics and Automation. Symposia Proceedings (Cat. No. 00CH37065)*, volume 1, pages 589–595. IEEE, 2000.
- [62] Jonas Schmidtler, Verena Knott, Christin Hölzel, and Klaus Bengler. Human centered assistance applications for the working environment of the future. *Occupational Ergonomics*, 12(3):83–95, 2015.
- [63] SDFORMAT website: <http://sdformat.org>.
- [64] F Sherwani, Muhammad Mujtaba Asad, and BSKK Ibrahim. Collaborative robots and industrial revolution 4.0 (ir 4.0). In *2020 International Conference on Emerging Trends in Smart Technologies (ICETST)*, pages 1–5. IEEE, 2020.
- [65] Michail Theofanidis, Joe Cloud, Ashwin Ramesh Babu, James Brady, and Fillia Makedon. A human robot interaction framework for robotic motor skill learning. In *Proceedings of the 11th Pervasive Technologies Related to Assistive Environments Conference*, pages 110–111, 2018.
- [66] Anand Thobbi and Weihua Sheng. Imitation learning of hand gestures and its evaluation for humanoid robots. In *The 2010 IEEE International Conference on Information and Automation*, pages 60–65. IEEE, 2010.
- [67] Sebastian Thrun. Toward a framework for human-robot interaction. *Human-Computer Interaction*, 19(1-2):9–24, 2004.
- [68] Dzmitry Tsetserukou, Katsunari Sato, Naoki Kawakami, and Susumu Tachi. Teleoperation system with haptic feedback for physical interaction with remote environment. In *2009 ICCAS-SICE*, pages 3353–3358. IEEE, 2009.
- [69] Yujin Tsukada and Takeshi Hoshino. Layered touch panel: the input device with two touch panel layers. In *CHI’02 Extended Abstracts on Human Factors in Computing Systems*, pages 584–585, 2002.
- [70] Andrés Úbeda, Eduardo Iáñez, José M Azorín, José M Sabater, Nicolás M García, and Carlos Pérez. Improving human-robot interaction by a multi-modal interface. In *2010 IEEE International Conference on Systems, Man and Cybernetics*, pages 3580–3585. IEEE, 2010.
- [71] Ubiquity Robotics website: <https://downloads.ubiquityrobotics.com>.
- [72] Ultraleap website: <https://www.ultraleap.com/product/leap-motion-controller/>.
- [73] Universal Robots website: <https://www.universal-robots.com>.
- [74] Nana Wang, Yi Zeng, and Jie Geng. A brief review on safety strategies of

physical human-robot interaction. In *ITM Web of Conferences*, volume 25, page 01015. EDP Sciences, 2019.