UNIVERSITÀ DEGLI STUDI DI PADOVA

DIPARTIMENTO DI TECNICA E GESTIONE DEI SISTEMI INDUSTRIALI
CORSO DI LAUREA MAGISTRALE IN INGEGNERIA MECCATRONICA

TESI DI LAUREA MAGISTRALE

DEPTH-SENSORED AUTONOMOUS SYSTEM FOR TABLE CLEARING TASKS

Relatore: GIOVANNI BOSCHETTI

Co-relatori: DAVID MARTÍNEZ MARTÍNEZ
GUILLEM ALENYÀ RIBAS

Laureando: NICOLA COVALLERO
Matricola 1103120

ANNO ACCADEMICO 2016-2017

Abstract

Manipulation planning is a field of study with increasing interest, it combines manipulation skills and an artificial intelligence system that is able to find the optimal sequence of actions in order to solve manipulation problems. It is a complex problem since it involves a mixture of symbolic planning and geometric planning. To complete the task the sequence of actions has to satisfy a set of geometrical restrictions.

In this thesis we present a planning system for clearing a table with cluttered objects, which tackles geometrical restrictions within symbolic planning with a backtracking approach.

The main contribution of this thesis is a planning system able to solve a wider variety of scenarios for clearing a table with cluttered objects. Grasping actions alone are not enough, and pushing actions may be needed to move an object to a pose in which it can be grasped. The planning system presented here can reason about sequences of pushing and grasping actions that allow a robot to grasp an object that was not initially graspable.

This work shows that some geometric problems can be efficiently handled by reasoning at an abstract level through symbolic predicates when such predicates are chosen correctly. The advantage of this system is a reduction in execution time and it is also easy to implement.

This Master Thesis is an extension of [12] initially done at the *Universitat Politècnica de Catalunya* at the **Institut de Robòtica i Informàtica Industrial** (IRI) in the Perception and Manipulation laboratory with the supervision of *David Martínez Martínez* and *Guillem Alenyà Ribas*. The extensions include some improvements of the system carried out in the same laboratory in the scope of publishing a conference poster paper [13]. Furthermore, for the sensing part a Microsoft Kinect sensor has been compared with a stereo vision system based on two frontal parallel cameras in order to assert the best sensor for the task. This has been developed at the *Università degli studi di Padova* at the **Dipartimento di tecnica e gestione dei sistemi industriali** (DTG) in the Robotics laboratory with the supervision of *Giovanni Boschetti*.

Acknowledgements

I want to thank my supervisors David Martínez Martínez and Guillem Alenyà Ribas for guiding me and teaching me along this thesis and during the writing of the paper. I am very grateful for all your help that you gave and still are giving to me. I have learnt so much about robotics, and not just robotics, thanks to both of you.

All the staff of IRI, everyone was so kind and open that made me feel at home. In particular my colleagues of the perception laboratory: Alejandro, Sergi, Enrique. And of course "los trabajadores" Gerard and Maite.

Ringrazio il professore Giovanni Boschetti per avermi dato la possibilità di "giocare" con la visione stereo, e per avermi offerto delle belle opportunitá e per tutto il supporto datomi.

A todos mis compañeros de Master, compañeros de trabajos, de fiesta, de parilladas y de mucho màs: Abraham y su mujer María, Esteban y su mujer Vero, Daniél y su mujer María, Eduardo, Eugenio. Gracias por haberme acompañado en este camino, cada uno tenía el suyo pero todos compartimos mucho. Cada uno de vosotros, con sus experiencias y sabiduría me ha enseñado mucho. En particular quiero agradecer a mis amigos y compañeros de muchos trabajos: Nacho y Eric. Suerte en vuestros caminos de investigadores, el mundo está pendiente de lo que vais a contar.

Un grandissimo ringraziamento alla mia famiglia per avermi sempre supportato durante tutti questi anni e per avermi permesso di intraprendere la meravigliosa esperienza di studiare due anni a Barcellona. Tutti i risultati li devo a voi.

E ovviamente a tutti i miei amici italiani, in particolare Davide, Mattia e Daniele che hanno condiviso le mie pene e le mie gioie durante questo percorso.

Un agradecimiento a los 4 fantasticos: Adeline, Angie y Yann. Mi estancia en Barcelona ha sido fantastica gracias a vosotros, os llevo en mis recuerdos. Os agradezco todos los intento de enseñarme que habia algo màs que el Master, espero que hayáis salido del centro de vez en cuando.

Y un agradecimiento va a mi amigo Samuél por aguantar mis quejas diarias por todo un año. Y a Sara, la mejor amiga que se podia encontrar en Barcelona.

A cada uno os deseo mucha suerte aunque os mereceis mucho màs.

Contents

1. Introduction

This Chapter introduces the project's motivation, its objectives, an insight of the approach used to tackle the problem, the experimental set up, and finally the outline of the thesis.

1.1. Project motivation

Robotic manipulation of objects is an increasing field of research which has captured the interest of researches from many years ago. In several industrial environments robots can be easily programmed when the objects are known a priori, i.e. the manipulation is always the same, and robot operations avoid cluttered scenes, but the workspace has to be designed in a manner to provide to the robot a non cluttered scene. However, there are situations in which robots with enhanced intelligence can be useful. An example in which a robot could face a cluttered scenario is the one of the *Amazon Picking Challenge* [10], which provides a challenge problem to the robotics research community that involves integrating the state of the art in object perception, motion planning, grasp planning, and task planning to manipulate real-world items in industrial settings such the ones human operators face in Amazon's warehouse. Joey Durham from Amazon Robotics describes the challenges of this competition as follows:

> "A robot for picking purposes must possess a lot of skills: the selected item must be identified, handled and deposited correctly and safely. This requires a certain level of visual, tactile and haptic perceptive skills and great handling capabilities."

Moreover, we introduce also the concept of *Industry 4.0*[1]. This term derives from the fact that we stand on the cusp of a fourth industrial revolution in which the factories will be smarter. The principles of this revolution are the ability of machines to communicate with each other, to create a virtual copy of the physical world, to support humans in tasks that are unpleasant and the ability to make decisions on their own.

With these considerations we think that this thesis is well motivated since it proposes

[1] http://www.gtai.de/GTAI/Content/EN/Invest/_SharedDocs/Downloads/GTAI/Brochures/Industries/industrie4.0-smart-manufacturing-for-the-future-en.pdf

a planning system to enhance the self-decision making step of robots for picking tasks exploiting a *Barret WAM* collaborative robot.

1.2. Objectives

The objective of this thesis is designing and implementing an artificial intelligence system which can reason about how to interact with the objects in the scene in order to clear a table with cluttered objects. To clear a table means grasp all the objects and remove them by dropping them, for example, into a bin. The cluttered hypothesis on the kind of scene introduces advanced difficulties in the manipulation because it may happen that to grasp a specific object the surrounding objects need to be manipulated as well (e.g. Figure 1.7b). Thus, the robot needs to be enhanced by a system which decides how to interact accordingly to the observed scene. To interact with the objects a 7 degree of freedom (DoF) robotic manipulator is used. The idea is to design the system by human-inspired actions, that is the intelligence system we want to develop tries to solve the task similarly as a human would do.

To locate the objects the system utilizes a depth sensor that captures a 3D image of the scene. To this aim, two different depth sensor systems are analysed to assert the more adapt for the task. These are the Microsoft Kinect sensor and a stereo vision system developed using two Basler cameras (see Appendix A).

The stereo vision system was developed at the DTG[2] in the Robotics laboratory. Here, there was not a suitable set up to perform the same experiment we did at IRI, thus the stereo system is tested performing real experiments to assert the quality of the 3D image obtained and it is integrated within the presented robotic system only in a simulated environment. Regardless, the design of the robotic system is very modular and the sensing part can be performed by any method able to give a 3D representation of the scene.

1.3. Problem Approach

The strategy to solve the problem is inspired by the way humans solve it. A human would use mainly two actions: grasping and pushing. When it is not possible to grasp an object,

[2] http://www.gest.unipd.it/en

because other objects prevent the human to put the hands in the proper way to grasp the desired one, he/she interacts with the scene to make the object graspable. However, humans are characterized by a high dexterity and therefore they have a lot of freedom in the way to grasp objects. Robots, normally, do not have such a dexterity and grasping in cluttered scenes could be very hard without moving the surrounding objects.

Based on these considerations, the actions the robot has to use are grasping and pushing. Grasping is the most important action since it lets to take an object and drop it somewhere, for instance into a bin, clearing in this way the table. There exist different works facing the same task by focusing only in grasping [18, 53]. The pushing is useful when two adjacent objects could not be grasped if they are so close such that the robot's gripper, when attempting to grasp an object, would collide with the adjacent one, making the object ungraspable. The pushing action can separate adjacent objects that mutually exclude themselves from being grasped. This requires to put the end-effector in a certain pose and then push the object by moving the end effector. However, it is difficult to push an object and ensure that it follows the desired path since the action's outcome depends on its shape. Moreover a controller would be needed in order to correct the pose of the end effector along the path. We assumed that all objects had basic shapes, so that a simple pushing action performs well. For simplicity we only consider actions that interact with a single object, whereas humans may push several objects together in some cases.

To do so the robot must be enhanced with an artificial intelligence system to make decisions. The decision maker uses a planner that returns a sequence of actions to achieve the goal of clearing the table. The robot reasons in an abstraction level by considering only symbolic predicates with limited geometrical information. Then the plan is checked to see if it is feasible, and if it isn't, it replans accounting that such an action cannot be executed (*backtracking* [34, 5]).

To include all the geometrical constraints into the symbolic planning system we propose to divide them into two groups: *relational* and *reachability* geometrical constraints. *Relational* constraints are those generated between objects when executing pushing and grasping action and they give a knowledge of how the objects are collocated with respect the others. They are computed by simulating pushing and grasping actions and checking all the collisions between the objects themselves and with the robot. *Reachability* constraints are generated when computing the robot motion path, possibly taking into account

obstacle avoidance.

We assume the world is perfectly known, that is the planner has all the information regarding any object. The actions are actually non-reliable and the outcomes could differ from the expected one with a certain probability. To handle that, instead to introduce probabilities in the planning stage, we supposed the actions to be deterministic and the uncertainty factor is handled by replanning after the execution of each action[27]. If the outcomes of an action differs from the expected, the planner will next return an updated plan accordingly to the scene.

The pipeline of the planning system (Figure 1.1) is divided in three subsystems:

- A **perception subsystem** devoted to locate the objects through a depth-sensor and to generate the state.
- A **planning subsystem** which is the core of the artificial intelligence that takes decisions depending on the state.
- An **execution subsystem** devoted to the actions execution .

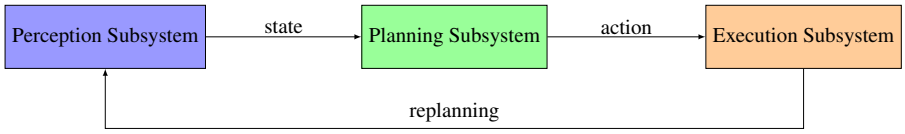


Fig. 1.1: Simplified pipeline of the planning system.

The design of planning subsystem is the most important part because it defines what actions can be executed and what information it needs from the perception subsystem to carry out the decision making stage.

In order to provide a graphical scheme to the reader the perception and planning pipeline is depicted in detail in Figure 1.2. The graphical scheme should be read together to the Algorithm 1. At each step, the perception subsystem (Sec. 4) takes an image from a depth sensor and generates the symbolic state, possibly containing errors due to perception noise. We propose to embed the *relational* and *reachability* constraints in the state definition and thus are tackled naturally by the planner (Sec. 3.4.2). The planning subsystem takes the state of the scene and uses the action model (Sec. 3.4.3) to compute a plan. In our current implementation, the whole plan until the goal is fulfilled is computed. If the plan cannot be found and replanning is not effective, the system cannot continue.

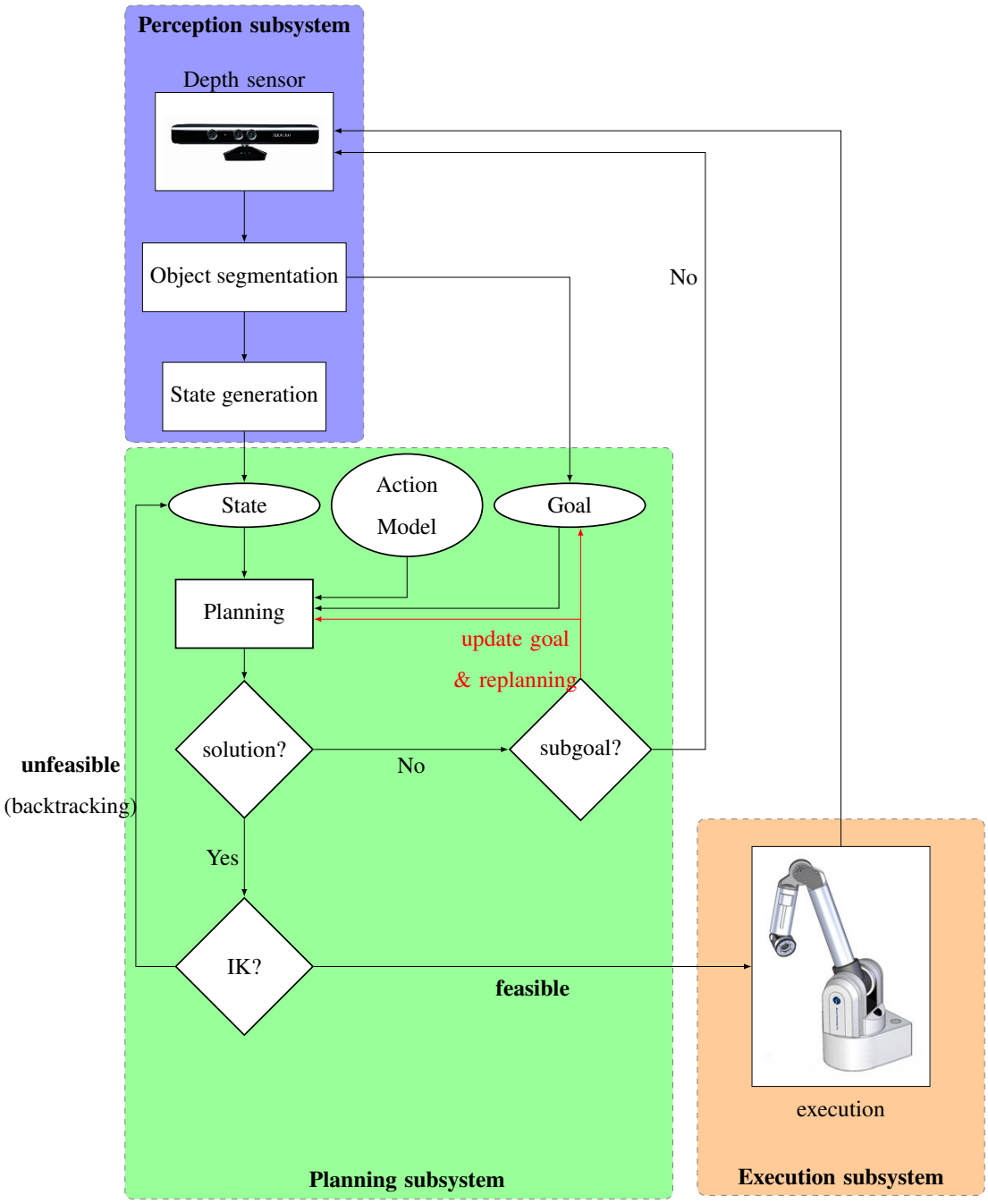


Fig. 1.2: The system is divided in three subsystems: the perception subsystem generates the state with relational restrictions, the planning subsystem finds a feasible plan with no reachability restrictions that are included afterwards through backtracking, and the execute subsystem executes the first action of the plan.

To solve this limitation, sub-goals [39] could be used to find a feasible sub-plan that allows the robot to continue towards the goal. The planning subsystem, after having found a plan, computes the *reachability* constraints to check for feasibility. If they can be fulfilled, the next action is executed; otherwise, backtracking (Sec. 3.5) is triggered to find an alternative solution. In our implementation we validate only the robot inverse kinematic (IK) for the next action, but the same scheme holds for more elaborated strategies, like using heuristics to compute only some key actions [37], compute all actions [34], or use complex robot motion planners [52]. After the execution of an action the system replans, that is it captures a new image and all the process is iterated until the goal is reached.

Notice that replanning is triggered either when an action is executed, in this way the uncertainties regarding the actions outcomes are handled, or whenever it is impossible to find a plan, this could be due to a wrong segmentation which is sensitive to the noise and it could be fixed by capturing a new image. To avoid occlusions, we wait until the robot finishes to execute an action and moves away before taking new images.

The pipeline of our current implementation is described in more detail in Algorithm 1. First a 3D image is taken, the objects are segmented, the goal is set and the state is generated (lines $2-5$). In our experiments the goal is removing all the objects, thus, if there are no objects onto the table the task is considered finished (lines $6-8$). Then, if the goal involves at least one object to interact with (line 9), a plan is obtained without limits regarding the *reachability* (line 10), i.e. all the actions can be executed. If a plan is found (line 11), *reachability* constraints are evaluated. In our implementation the *reachability* constraints refer to the inverse kinematic (IK) of the actions, so we evaluate the robot IK for the first action in the plan; if it is not feasible the state is updated and backtracking takes place. This is repeated until a feasible action is obtained or there exists no solution (lines $13-17$). If there exists a plan and the first action is feasible, it is executed and replanning is triggered (lines $18-20$). Otherwise, if there exists no plan, the system updates the goal (lines $22-24$) removing from the goal the objects the robot cannot interact with because of *reachability* limitations, these information are included into the state. Finally, if the robot has executed an action or cannot find a feasible plan, it replans. The task is considered finished when there are no longer objects on the table.

Algorithm 1 Planning system's pipeline.

```
1:  while ¬finished do
2:      image ← captureImage( );
3:      objects ← segment(image);
4:      goal ← objects;
5:      state ← stateGeneration(objects);
6:      if goalEmpty(goal) then
7:          finished ← true;
8:      end if
9:      while ¬goalEmpty(goal) do
10:         plan ← planning(state);
11:         if hasSolution(plan) then
12:             action ← IK(plan[0]);
13:             while ¬isFeasible(action) ∧ hasSolution(plan) do
14:                 state ← updateState(action);
15:                 plan ← planning(state);
16:                 action ← IK(plan[0]);
17:             end while
18:             if isFeasible(action) ∧ hasSolution(plan) then
19:                 execute(action);
20:                 break
21:             end if
22:             if ¬hasSolution(plan) then
23:                 goal ← updateGoal(state);
24:             end if
25:         end if
26:     end while
27: end while
```

(a) Barrett WAM arm **(b)** Microsoft Kinect sensor **(c)** Stereo vision system

Fig. 1.3: Robot and depth vision sensors.

1.4. Contributions

The contributions of this thesis are:

- A planning system that thanks to the combination of pushing and grasping actions is capable to solve a wider variety of scenarios than considering only grasping action.
- A symbolic planning system which finds the best sequence of pushing and grasping actions.
- A perception system which translates geometrical restrictions into symbolic predicates. A lazy approach is used to evaluate if actions are feasible in order to improve the efficiency of the system.

1.5. Set up

In order to make the problem clear to the reader, the set up of the environment the robot will work in is presented here.

The robot used is a Barret WAM arm, which is a 7 degree of freedom (DoF) manipulator arm (Figure 1.3a). The WAM is noteworthy because it does not use any gears for manipulating the joints, but cable drives, so there are no backlash problems and it is both fast and stiff. Cable drives permit low friction and ripple-free torque transmission from the actuator to the joints. To detect the objects a Kinect camera, a RGB-D sensor, is employed (Figure 1.3b). In section 4.2 also a stereo vision system (Figure 1.3c) is build and tested to assert the best sensor for the task. However, in the experimental set up of the whole planning system only the Kinect was employed.

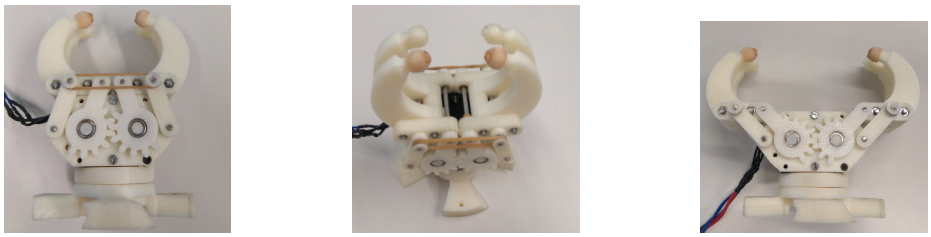


Fig. 1.4: Gripper used for the experiments

Fig. 1.5: Gripper and WAM.

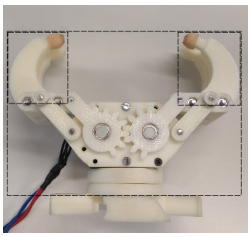
To manipulate the objects the robot has a gripper designed in the IRI institute and actuated by Dynamixel motors. Such a gripper is depicted in Figure 1.4 from several point of views. Its closing width[3] is 3 centimetres while its opening width[4] is of 7 centimetres, therefore we are constrained to grasp objects with a width in the range $[3 \div 7]cm$.

For the task planner, as the reader will see in Chapters 3 and 4, the model of the gripper is an important resource in order to compute the state. The gripper is modelled measuring some principal elements such as: finger's width and height, gripper's width, height and deep, closing and opening width. The modelling procedure is depicted in Figure 1.6. The resulting model is a simple triangle mesh which includes all the important geometric information of the gripper. Such a simple model allows the collision algorithm commented in Chapter 4 to check for collision in just a few milliseconds. A more detailed and complex model would have higher precision, but such a high accuracy is not needed, and it would slow down the algorithm. The gripper is mounted in the end effector of the robot as shown in Figure 1.5.

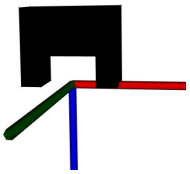
The scenario the robot is going to work in is composed of a table and the objects will lay on top of it. In Figure 1.7a the main elements of the set up are highlighted. In this figure the depth sensor is the Microsoft Kinect sensor. The WAM arm's base is in a fixed position with respect the table and the depth camera is located on top of the table

[3] Distance between the fingers when the gripper is closed.

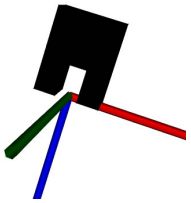
[4] Distance between the fingers when the gripper is open.



Elements measured

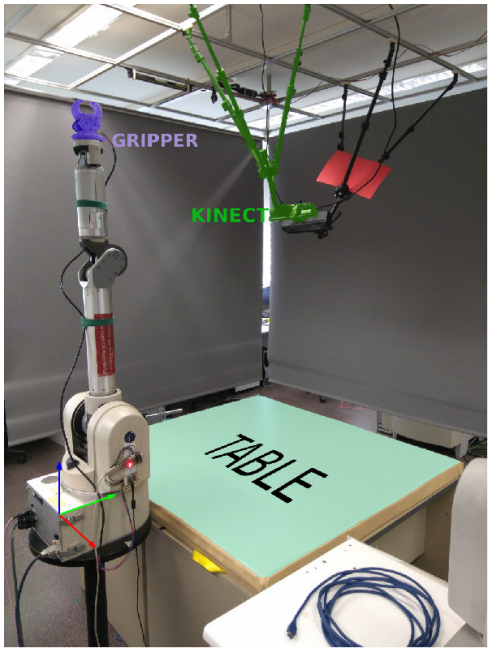


Opened gripper mesh model



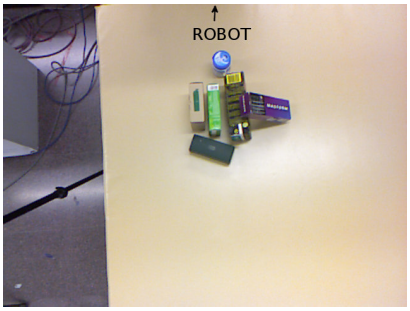
Closed gripper mesh model

Fig. 1.6: At the left the principal elements measured are highlighted for the opened gripper model. The gripper mesh model is here shown in the PCL visualizer. The green, red and blue axis are respectively the x, y and z axis.



(a) Principal elements of the experimental set up.

(b) Example of a cluttered scene.



(c) Kinect's view.

Fig. 1.7: Experimental set up with an example of a cluttered scene the robot is going to interact with. The depth sensor used is the Microsoft Kinect sensor.

pointing downward. The camera is calibrated and the fixed transformation between the camera's frames and the base frame of the robot is known, so all the points measured can be expressed in coordinates with respect the robot's base frame. The transformation is given by the translation vector $\overrightarrow{t} = [0.9014760, -0.0594, 0.725172]m$ and the rotation vector $\overrightarrow{r} = [0.28, 1.379, -2.921]rad$ which represents the roll, pitch and yaw angles respectively. Figure 1.7b shows an example of a cluttered scene the robot is going to deal with, and Figure 1.7c shows the same scene seen by the camera.

1.6. Outline of the Thesis

The document starts by analysing in Chapter 2 some relevant previous works in manipulation planning comparable to ours to assert the contributions of this work. Then, since the proposed system is organized in three subsystems, we describe them by starting from the planning subsystem. Although it is the second one in a logical order, the others are designed in function of the planning subsystem. The planning subsystem is the one which defines what information are required from the scene and what are the possible actions to execute. The planning, perception and the execution subsystems are explained in Chapters 3, 4 and 5 respectively. Chapter 4 includes the comparison, supported by three experiments, between the Kinect and the stereo system to assert the best depth sensor for the task. Then, we proceed discussing the design of the software in Chapter 6. Finally in Chapter 7 the experiments performed are discussed and the document concludes presenting the conclusions and future works in Chapter 8.

2. Previous works

In this chapter we present some previous works regarding manipulation planning and analysing them for their contribution with respect this work.

Many manipulation planning approaches[35] assume that the task can be treated as a geometric problem with the goal to place the objects in their desired positions. Planning is essentially done with a mixture of symbolic and geometric states. They require to obtain the symbolic predicates that represent geometric features, which are very time consuming. Therefore, these hybrid planners can be too slow in real applications.

Dogar and Srinivasa [16] proposed a framework for planning with cluttered scenes using a library of actions inspired by human strategies. They designed a planning system that decides which objects to move, the order, where to move them, and the appropriate manipulation actions. Moreover, it accounts for the uncertainty in the environment all through this process. The planning system first attempts to grasp the goal object, and if it is not possible, it identifies what is the object that prevents the action and adds it to a list of objects that have to be moved. Afterwards, those objects are moved in whatever position that makes the goal feasible. Their work is the most similar to our, but their planning system cannot be directly applied to a table clearing task. The goal is a single object at a time, then to grasp another object they need to replan. Our approach performs better with goals that involve more than one object. We plan sequence of actions considering all objects in the goal. The actions they use to move objects that were in the way may actually hinder future goals.

A recent alternative proposed by Mösenlechner and Beetz [41] is to specify goals symbolically but evaluate the plan geometrically. The idea is to use a high-fidelity physics simulation to predict the effects of actions and a hand-built mapping from geometric to symbolic states. Planning is conducted by a forward search, the effects of actions are determined by simulating them, and then the mapping is used to update the symbolic state. However, their method requires to know the physic of the manipulated objects to simulate them. Moreover the authors didn't test their planning system with a complex scene like the ones faced in this thesis. Our planning system doesn't use any simulator, instead it relies on a prediction algorithm to represent how the objects can be manipulated, leading to a faster and easier to implement solution.

In [9] the authors address a problem similar to the one of this thesis. The authors blended pushing and grasping actions for a table manipulation task. They use the concept of reachability [54] to exclude impossible poses of the gripper at the planning stage, creating a reliable plan suitable for real-time operation. The authors model the planning problem through a Markov Decision Process (MDP), discretizing the world in grid cells and assigning each one a push and grasp vector defined by the reachability concept. Their advantage is that they plan a trajectory level so they can consider more details. In contrast, we plan at an action level, so we can consider more complex goals involving several objects, and will optimize the sequence of actions for completing the whole task. Moreover, while their method needs to be adapted to each robot, to build a reachability map, our method can be directly integrated in any robotic manipulator.

Symbolic planning requires knowledge about the preconditions and effects of the individual actions and such a knowledge can be obtained through machine learning techniques. In [2] the authors proposed an approach to learn manipulation skills, including preconditions and effects, based on teacher demonstrations. With just a few demonstrations the method learns the preconditions and effects of actions. This work looks promising since it allows to resolve planning problem by learning the model, but it is suitable only for simple actions. Having a hand-built model, like the one of our work, lets to solve more complex problems and also it is more straightforward.

In [14] Dearden and Burbridge proposed an approach for planning robotic manipulation tasks which uses a learned bidirectional mapping between geometric states and logical predicates. First, the mapping is applied to get the symbolic states and the planner plans symbolically, then the mapping is applied to generate geometric positions which are used to generate a path. If this process fails they allow the system a limited amount of purely geometric backtracking before giving up and backtracking at the symbolic level to generate a different plan. However, this method cannot tackle complex scenes, such as cluttered objects, since in those cases learning a good mapping would be very hard.

Compared to the state of the art, we propose a planning system for clearing cluttered objects. Our approach plans at a symbolic level, which is efficient and is low time consuming (the time to get a plan is usually less than 0.5 seconds). As far as we know, previous approaches haven't tackled very cluttered scenes, such as the one in Figure 1.7b. We will also show that the lack of geometric constraints introduces some limitations to

the system, but the general results obtained are good.

3. Planning subsystem

In this chapter, after an introduction to task planning and a review of the current state of the art of task planners, a proper planner is chosen and then a suitable description to the table clearing problem is discussed.

3.1. Introduction to task planning

Task planning is the branch of artificial intelligence that concerns strategies to achieve a determined goal. There exists a variety of planners that can involve different set of informations at the planning stage. This introduction focuses on the simplest case which is given by a fully observable, deterministic, finite (with a countable set of possible states and actions) and static (change happens only when the agent, the robot in our case, acts) environment [47]. To describe this classical planning we use the state space model [35]. The fundamental idea is that every possible and interesting situation that can happen in the world is called a *state*, denoted by s, which is a multidimensional variable, and the set of all possible states is called *state space*, denoted by S. For discrete planning S is countable and it is up to the developer designing the possible states in order to account all the relevant information. Aside the state, the model needs to include an *action model* which defines what actions can be executed by the agent and the rules of these actions. The rules define if, given s, action a can be executed and the result is the new state s' specified by a *state transient function*, T. In formulas, the outcome of action a applied to s is the state s':

$$s' = T(s,a). \tag{3.1.1}$$

For each state s there exist a countable set of actions that can be executed, this set is called *action space* and is denoted as $A(s)$.

The general planning problem is formulated as a 6-tuple $\Pi = \langle S, s_o, G, A, T, c \rangle$, where:

- S is the finite set of states;
- $s_o \in S$ is an initial state;
- $G \in S$ is a goal state;

- A is a finite set of applicable actions;
- $T(s,a) : S \times A \times S$ is a deterministic transition function;
- $c(a)$ is the cost to applying action a.

A plan τ_i is a sequence of actions applicable from the initial state to the goal state. The cost of a plan $C(\tau_i)$ is the sum of the cost of the actions of the plan $C(\tau_i) = \sum_{a \in \tau_i} c(a)$. The optimal solution τ^* is the solution with less cost: $\tau^* = \min_{\tau_i} C(\tau_i)$.

When S is finite the aforementioned formulation appears almost identical to a *finite state machine*. Thus, the discrete planning problem could be interpreted as the sequence of input that makes a finite state machine eventually report a desired output.

It is often convenient to express this formulation as a directed *state transition graph* where the vertices, also called nodes, are all the states in S. In this graph, a directed edge from s to s' exists if there exists an action $a \in A(s)$ such that $s' = T(s,a)$. The state transition graph is usually built iteratively during the planning process exploiting the information of the state and action model in order to connect the initial state to the goal state. The graph built until a certain iteration is called *search graph*. Whereas the state transition graph includes all the possible transitions among all the possible states, the search graph includes only the visited transitions and states. The aim of the search graph is not the one to explore the whole state transition graph but to connect the initial state to the goal state finding a path with the smallest cost.

Given the 6-tuple Π the planner have to find the sequence of actions to achieve the goal. To do so, the simplest solution would be to explore iteratively the whole state space but this would require too much time. Instead, there exist search algorithms that tries to optimize this search by focusing on exploring interesting solutions and avoiding solutions that are known not to solve the task. These methods are called *heuristics search methods*. For this aim the searching algorithms are systematic, that is they keep track of the visited states in order to focus on exploring unvisited states and they use heuristics to prioritize the next state to expand the graph.

In the followings the general scheme on which these search methods are based on is described:

1. **Initialization:** the search graph is first initialized with the initial state marked as visited. The search graph will incrementally grow to reveal more and more of the

state transition graph.

2. **Select a state:** choose another state, for expansion. The state is typically chosen from a priority queue which is governed by some *heuristics*.

3. **Apply an action:** virtually apply the action required to obtain the new state, given the current one, accordingly to the state transition function.

4. **Insert a directed edge:** if the action can be executed it is inserted in the graph and the new state is marked as visited.

5. **Check for solution:** if the new state is the goal then the task is terminated.

6. **Return to step 2:** iterate unless a solution is found or a certain termination criteria is satisfied.

To guide the search the states are labelled as visited or unvisited. The unvisited ones are the ones that can be select for expansion from a visited state. A visited state $\underline{s}$ can be mark as open node if the exploration of the state transition graph can continue from such a state, or as closed node if all the possible applicable actions to $\underline{s}$ have already been tested.

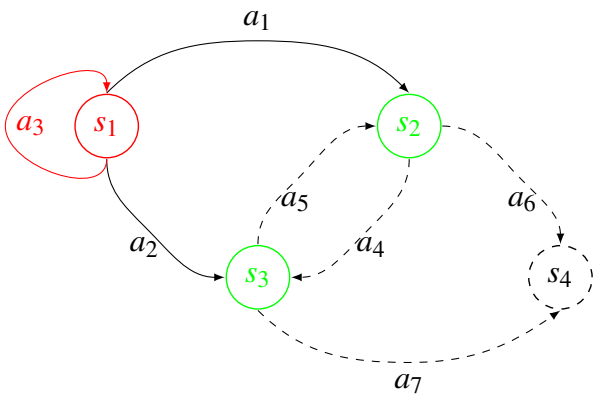


Fig. 3.1: Example of the construction of the search graph. The initial and goal states are s_1 and s_4 respectively. The dashed states and lines refers to unvisited states, while the non-dashed ones refer to visited states. Green circles refers to open node from which the graph can extend, while the red one refers to a closed node. Action a_3 is an useless action because next state is a previously visited state, thus it is not consider by the search algorithm.

Let's focus on the example of Figure 3.1 where s_1 and s_4 are the initial and goal

states respectively. The search graph is rooted in s_1 since we consider a forward search[1]. In the particular iteration considered, the search graph has labelled s_1 as a visited and closed node because all the possible reachable states from s_1 have been visited. States s_2 and s_3 are labelled as visited and open nodes, thus the exploration can continue from them. In this case, to explore from s_2 an heuristics could select for expansion the states s_4 or s_3. The selection depends on an estimate of the cost to reach the goal, such an estimate is done by a heuristic function in order to bias the exploration to speed up the searching. Let consider the following costs for the actions $c(a_1) = 1, c(a_2) = 4, c(a_3) = 1, c(a_4) = 2, c(a_5) = 3, c(a_6) = 7, c(a_7) = 4$. In this scenario the optimal plan is given by the sequence a_1, a_4, a_7, thus a good heuristic would select s_3, instead of s_4, to expand the graph from s_2, although s_4 is the goal. For this simple example there are no clear benefits in using heuristics but for bigger state spaces they may be fundamental to make the problem tractable.

What heuristics do is to estimate the cost of a path to reach the closest goal state (there could exist more goal states when the user is interested in satisfying only a certain part of conditions). An example of a very basic heuristic is the *Hamming* heuristic which counts the number of variables of the state that differ from the goal. Let recall that the cost of a path is the sum of the cost of the actions belonging to such a path. These heuristics are just estimates of the cost, thus, if a goal is reached by a path obtained exploring always the state with minimum heuristics this does not mean that it is the optimal plan, but a suboptimal one. In fact, there exist three main properties for a heuristic function h:

- **admissibility**: a heuristic is admissible if it never overestimates the real cost of reaching the closest goal state from state s_i. Formally:

  $$h(s_i) \leq h^*(s_i)$$

  where $h^*(s_i)$ is the real cost to reach the goal from s_i.

- **consistency**: a heuristic function is consistent, or monotone, if its estimate is always less than, or equal to, the estimated distance from any neighbouring state to the goal, plus the cost of the action to reach such a state. Formally, let s_2 be a state reachable

[1] There exist backward search algorithms whose search graphs are rooted in the goal state. Some planners also use a bidirectional search such the two trees are rooted one from the initial state and another one from the goal state.

from state s_1 by applying action a_1, the estimated cost to reach the goal s_g from s_1 is not greater than the cost to execute the action to get state s_2 plus the cost of reaching the goal from s_2:

$$h(s_1) \leq c(a_1) + h(s_2)$$
$$h(s_g) = 0.$$

Consistency implies admissibility but the vice versa is not true.

- **safety**: a heuristic function is safe if its estimate is infinite if and only if the actual cost to reach the goal is infinite.

The ideal properties for optimal planning are admissibility, consistency and accuracy, i.e. the heuristic estimate is close as possible to the real cost. Thus a heuristic biases the expansion of the search graph by minimizing the visited states and speeding the process up.

Having a basic understanding of task planning, although restricted to classical planning, we move to analyse what are the available planners to finally decide what suits better to our application.

3.2. Task planners review

To choose the proper planner for the task we evaluated three main categories of planners:

1. classical planners,
2. hierarchical planners,
3. probabilistic planners.

Classical planners are characterized by environments which are fully observable, deterministic, finite and static (changes happen only when the agent acts) and discrete (in time, actions, objects...) [47]. A very well known classic planner is the Fast Downward planner [21].

Hierarchical planning, also called *Hierarchical Task Network*(HTN), works in a similar way to how it is believed that human planning works [38]. It is based on a reduction of the problem. The planner recursively decomposes tasks into subtasks, stopping when it

reaches primitive tasks that can be performed directly by planning operators. This kind of planner needs to have a set of methods, where each method is a schema for decomposing a particular kind of task into a set of subtasks. For this kind of planning technique a well known planner is SHOP [42].

Probabilistic planning is a planning technique which considers that the environment is not deterministic but probabilistic. So the actions have a probability to obtain a certain state, and given an initial state and a goal state, the planner finds the solution path with the highest reward, which depends also on the probability. Probabilistic problems are usually formalized as Markov Decision Processes (MDP). In this category two probabilistic planners that performed good in planning competitions are Gourmand [33] and PROST [31].

3.3. Planner selection

The problem involves a big amount of uncertainty due to the interaction of the robot with the environment. When the robot interacts with the objects, it is hard to predict correctly the position of the object after the execution, that is the next state. A probabilistic planner considers the probability associated with every effect to predict the state after executing an action, such a probability has to be specified or learned for each type of object.

Martinez et al. [40] faced the problem of cleaning a surface entirely by probabilistic symbolic planning. The problem they solved was characterized by a strong uncertainty, and a unexpected effect of an action would require to replan and therefore to slow down the system. Another way to face the problem of probability is replanning after each executed action [27], or whenever the current state deviates from the expected one, generating a new plan from the current state to the goal. In this case the actions are considered deterministic, and only the first action is executed before replanning again.

Little et al. discussed in [36] the problem of when is more useful the probabilistic planning with respect a simple replanning system. They defined a planning problem *probabilistic interesting* if dead ends can be avoided, exist multiple goal trajectories and there is at least one pair of distinct goal trajectories, τ and $\tau^{'}$ that share a common sequence of outcomes for the first $n-1$ outcomes, and where τ_n and $\tau_n^{'}$ are distinct outcomes of the same action.

They assert that unless a probabilistic planning problem satisfies all of the conditions to be *probabilistic interesting* then it is inevitable that a well-written replanner will outperform a well-written probabilistic planner. Moreover the authors do not negate the possibility that a deterministic replanner could perform optimally even for probabilistically interesting planning problems.

To conclude, in many problems it is more efficient to replan with a deterministic planner rather than directly using a probabilistic planner.

Taking into account such considerations and that, except for rare cases, our planning problem is not *probabilistic interesting*, the problem has been thought to be solved by a deterministic planner.

A hierarchical planner would be a good choice if the problem presented some hierarchies, for instance in the case the goal was to clear several tables. Since the problem is about cleaning a single table it is more straightforward to use a classical planner.

The planner chosen was the **Fast Downward** planner [21], a very well know classic one. This planner is feature-wise complete, stable and fast in solving planning problems. The planning problem we handled has not a wide state space and action space, thus, in order to ensure the optimality of the plan, we used a *blind heuristic* that works as a brute force algorithm by selecting randomly the next possible state since it has no information to prioritize any state to expand the search. We experienced with other type of heuristics that the plan was not always optimal, whereas with the blind heuristic the plan is optimal and the increasing of elapsed time is negligible compared to other heuristics.

3.4. State space and action model

Before to proceed to the description of the state and action model for the table clearing problem, the general representation of states and actions is described.

3.4.1. Representation

The *Fast Downward* planner needs the problem to be formulated in *Problem Domain Description Language* (PDDL) [1]. This is a widely used representation for planning domain whose state and action representation is here briefly described. The PDDL formulation is based on two representation: one for the domain, which specifies the state space and the

action model, one for the problem to solve, which specifies what are the initial and goal states.

States The states are represented as a conjunction of *literals*, also called *symbolic predicates*, for example `Happy`$\wedge$`Graduated` may represents the state of a lucky and recently graduated agent. Literals can be either positive, when true, or negative, when false. An extension of this representation is based on *first-order literals* such as `onTable(obj1)`$\wedge$`onTable(obj2)` that might represent a state where objects 1 and 2 are both onto the table. The *close-world* assumption is used, meaning that every omitted condition in the state is assumed false. All the possible literals to describe the state are typically defined in the domain formulation.

The goal is a state, thus its representation is equivalent to the states and every omitted literal means that it can be either positive or negative. In such a case the goal is defined by multiple states.

Actions An action is specified in terms of the preconditions that must hold in order to be able to execute that action, the effects subsequent the action and the cost to apply it. An example of the action of grasping an object and removing it from the table could be:

`Action(grasp(obj)):`

`PRECONDITIONS: GraspFree(obj)` $\wedge$ `onTable(obj)` $\wedge$ $\neg$`removed(obj)`

`EFFECTS: removed(obj)` $\wedge$ $\neg$`onTable(obj)`

`COST: 10`

In words, the action grasps the object labelled as `obj`, with a cost equal to `10`, if it is collision-free when grasped, if it is on the table and if it has not already been removed. The effects are that `obj` is removed and not on the table. We see here that the effects can be also negative, meaning that some literals are set to false (e.g. $\neg$`onTable(obj)`). Notice also that the set of actions defines the transition function we defined in the introductory section.

More properly, the above action notation refers to an *action schema* because it represents a set of different actions that can be derived by varying the variable `obj`.

An action is considered *applicable* when all the preconditions are satisfied. If in the preconditions some literals are omitted, it means that they can be either positive or negative. If in the effects some literals are omitted, it means that they remain unchanged.

3.4.2. State space

The task this thesis faces is a common task done by humans, who think in order to find a feasible sequence of actions. Such a sequence is normally composed of actions that avoid collision between the manipulated objects and the other ones, whenever possible. To do this we, as humans, think on what is going to happen if we manipulate an object in a certain way accordingly to our prediction. The design of the problem has been inspired by such a reasoning way and symbolic predicates are used so that the planner can reason about collisions and geometrical constraints.

As described in the introduction, the system will be able to perform two types of actions: **pushing** and **grasping**. Grasping action is a necessary action in order to grasp an object and drop it into a bin, while the pushing action is an auxiliary action which has the aim to move an object in a pose that does not obstacle the solution of the problem. The pushing action is said to be auxiliary because it is not strictly necessary to solve every kind of problem, depending on the cluttered scene the grasping action may suffice. The combination of these two actions makes wider the range of problem this planner can solve.

The symbolic predicates are designed accordingly to the available actions trying to answer the following questions:

- When can an object be grasped?
- When can a object be pushed? In which direction?

Answering these questions the following predicates are defined:

- **removed**: `(removed o1)` means that object `o1` has been grasped and removed from the table. The goal is reached when all the objects have been removed.
- **on**:`(on o1 o2)` means that object `o1` stands on top of object `o2`. This predicate is defined since we don't want to interact with an object that has objects on top of itself. If we would grasp it, the object above would likely fall corrupting in this way the scene. That behaviour is undesired since a human, normally, would not grasp the bottom object without first grabbing the one on the top. Similarly for the pushing action, when an object with objects on top of itself is pushed, they could fall or collide with other objects. Vice versa if it was on top of other objects.

- **block_grasp**: `(block_grasp o1 o2)` means that object `o1` prevents object `o2` to be grasped. Once we are sure that an object has no objects on top of it we have to check if it is graspable, that is if the gripper would collide with adjacent objects attempting to grasp the desired one. With this predicate the planner knows whether the robot has first to interact with those objects before to grasp the target one.

- **block_push**: `(block_push o1 o2 dir1)` means that object `o1` prevents object `o2` to be moved along direction `dir1`. We considered 4 possible pushing directions (defined in Chapter 4) per object. Being observant to the philosophy of human-inspired actions, we avoid collisions when we push an object. To do so we translate the object along the pushing direction, for a certain length until the object is graspable. and check for collision. Moreover, to push an object the end effector has to be put in the opposite side with respect the pushing direction, so an object cannot be pushed along a certain direction even in the case the gripper collides with an object. Therefore if an object cannot be moved along a certain direction it is because the object would collide, or the end effector would collide, with other objects.

- **ik_unfeasible_dir**, **ik_unfeasible_grasp**: to consider the geometrical constraints regarding the working space of the robot, a predicate which states whether the inverse kinematic has solution is added for each action. For instance, `(ik_unfeasible_dir o1 dir1)` means that the inverse kinematic to push the object `o1` along pushing direction `dir1` has no solution or that `o1` would be pushed outside the working space such that it cannot be grasped afterwards. Whereas `(ik_unfeasible_grasp o1)` means that the grasping pose of `o1` is outside the working space. These predicates are added in order to consider the *reachability* constraints regarding the working space of the robot.

Predicates `on`, `block_grasp`, `block_push` represent the *relational* geometrical constraints mentioned in the introduction, whereas `ik_unfeasible` predicates represent the *reachability* geometrical constraints. The firsts are computed by the perception subsystems and the seconds by the planning subsystem when an action is needed by a plan.

3.4.3. Action model

In this section the actions preconditions and effects are described.

Grasping Action The **preconditions** to grasp a certain object are:

- no object stands on top of it,
- no object collides with the gripper attempting to grasp the desired object,
- the inverse kinematic has solution.

The **effects** of the grasping action are:

- the grasped object is removed and dropped into the bin,
- the grasped object no longer prevents other object from being pushed or grasped,
- if the grasped object was on top of other ones, it is no longer on top of them.

The cost is 1 since all the grasping actions help to remove objects, thus to reach the goal state of a cleared table.

Pushing Action The **preconditions** to push a certain object, along a certain direction, are:

- no object stands on top of it,
- the manipulated object is not on top of other objects,
- no object collides with the manipulated one, or with the gripper, when pushed,
- the inverse kinematic has solution.

In particular we defined 4 pushing actions, one per pushing direction. The symbolic planner is not able to capture all the geometrical information of the problem through symbolic predicates, therefore it is not able to predict the future position of the manipulated object, and so the future states. This problem was handled by considering the object to be pushed until it is graspable in such a way it can be removed afterwards (Figure 3.2).

Therefore, the **effects** of this action are:

- the manipulated object no longer prevents other objects from being pushed or grasped,
- the other objects no longer prevent the manipulated object from being pushed or grasped.

Fig. 3.2: The pushing length l is function of the neighbour objects. The first image shows the configuration before moving the purple box. The second one shows the execution of pushing the purple box. In the third one we see the purple box is moved to a graspable pose.

When the object is pushed the end-effector is moving close to the objects and it has to avoid collisions with them otherwise the hypothesis of a careful manipulation falls. To do so we weight the cost of the action of pushing a certain object along a certain direction by considering the collision-free range for the end-effector. In detail, the cost is an exponential function of the minimum distance d_{min} between the other objects and the end-effector:

$$c = \lceil e^{k(n-d_{min})} \rceil$$

where $k = 100$ is the gain factor and $n = 0.05$ refers to the minimum distance, in meters, to consider the pushing action as safe. When $d_{min} \geq n$ the cost is 1. In order not to be too conservative this is done only for the pushing pose, that is the pose of the end-effector from where it starts to push (see Chapter 5). The results is that the pushing actions with wider collision-free range, to locate the end-effector for pushing, have high priority. This is done only for the pushing pose, hence the collisions along the pushing path with the other objects are allowed. Despite this, these collisions rarely happen and they results just in slight movements of the other objects. In cluttered scenes as the ones we are going to deal with some collisions must be permitted otherwise the task would be likely unsolvable.

3.4.4. PDDL syntax

For clarity purposes, the PDDL syntax of the domain (state space and action model) is shown in Listing 3.1. For the reader who is not experienced with PDDL see [1]. The domain specifies two types to differentiate between objects and directions, the aforementioned predicates and the actions. Notice that the notation of these actions differ from

an action schema but we define a particular action for each object and pushing direction. Despite this, the notation used to represent an action in this Thesis is of the type `(push o0 dir1)` instead of `(push-o0-dir1)`, since the meaning is the same and the notation is cleaner. This is due to an implementation issue because it was not possible to include costs that vary depending on the parameters. The actions of the domain are all similar, they differ only for the objects of the first-order literals in the preconditions and effects. Thus, only the grasping action and pushing action, along the first direction, for object `o0` has been reported. This means that the PDDL domain file is generated on-line at every iteration, after having detected the objects and computed the cost of pushing actions.

```
(define (domain domain_table_clearing)
(:requirements :adl :existential-preconditions :universal-preconditions
    :action-costs )
(:types obj direction )
(:predicates
(block_push ?o1 - obj ?o2 - obj ?d - direction )
(on ?o1 - obj ?o2 - obj )
(block_grasp ?o1 - obj ?o2 - obj )
(removed ?o - obj )
(ik_unfeasible_dir ?o - obj ?d - direction )
(ik_unfeasible_grasp ?o - obj )
)
(:functions (total-cost) - number)
(:action grasp-o0
:parameters ()
:precondition
(and
  (not (exists (?x - obj)(on ?x o0)))
  (not (exists (?x - obj)(block_grasp ?x o0)))
  (not (ik_unfeasible_grasp o0))
)
:effect
(and
  (removed o0)
  (forall (?x - obj)
    (when (on o0 ?x) (not (on o0 ?x)))
  )
  (forall (?x - obj)
```

```
    (when (block_grasp o0 ?x) (not (block_grasp o0 ?x)))
  )
  (forall (?x - obj)
    (and
      (forall (?d - direction)
        (when (block_push o0 ?x ?d)(not (block_push o0 ?x ?d)))
      )
    )
  )
  (forall (?x - obj)
    (and
      (forall (?d - direction)
        (when (ik_unfeasible_dir ?x ?d)(not (ik_unfeasible_dir ?x ?d)))
      )
      (when (ik_unfeasible_grasp ?x)(not (ik_unfeasible_grasp ?x)))
    )
  )
  (increase (total-cost) 1)
)
)


(:action push-o0-dir1
:parameters ()
:precondition
(and
  (not (exists (?x - obj)(block_push ?x o0 dir1)))
  (not (exists (?x - obj)(on ?x o0)))
  (not (exists (?x - obj)(on o0 ?x)))
  (not (ik_unfeasible_dir o0 dir1))
)
:effect
(and
  (forall (?x - obj)
  (and
  (forall (?d - direction)
  (and
  (when (block_push o0 ?x ?d) (not (block_push o0 ?x ?d)))
  (when (block_push ?x o0 ?d) (not (block_push ?x o0 ?d)))
```

```
67    (when (ik_unfeasible_dir ?x ?d)(not (ik_unfeasible_dir ?x ?d)))
      )
69    )
      (when (block_grasp o0 ?x) (not (block_grasp o0 ?x)))
71    (when (block_grasp ?x o0) (not (block_grasp ?x o0)))
      (when (ik_unfeasible_grasp ?x)(not (ik_unfeasible_grasp ?x)))
73    )
      )
75    (increase (total-cost) 75)
      )
77  )
    )
```

Listing 3.1: Domain PDDL syntax.

An example of a PDDL problem file for a problem with three objects is showed in Listing 3.2 and it refers to the example in Figure 6.8. This file, which is generated online, specifies the interesting elements for the problem (objects and pushing directions), the initial state and the goal. What differs from different problem files is only the initial state and the number of objects.

```
(define (problem problem_new_full)
2 (:domain domain_new_full)
(:objects
4 o0 - obj
o1 - obj
6 o2 - obj
o3 - obj
8 o4 - obj
dir1 - direction
10 dir2 - direction
dir3 - direction
12 dir4 - direction
)
14 (:init
(= (total-cost) 0 )
16 (block_push o1 o0 dir2 )
(block_push o2 o1 dir1 )
18 (block_push o4 o1 dir1 )
(block_push o2 o1 dir2 )
```

```
(block_push o4 o1 dir2 )
(block_push o0 o1 dir3 )
(block_push o3 o1 dir3 )
(block_push o3 o1 dir4 )
(block_push o3 o2 dir2 )
(block_push o4 o3 dir1 )
(block_push o4 o3 dir2 )
(block_push o1 o3 dir3 )
(block_push o1 o3 dir4 )
(block_push o1 o4 dir3 )
(block_push o3 o4 dir3 )
(block_push o1 o4 dir4 )
(block_grasp o3 o1 )
(block_grasp o1 o3 )
(block_grasp o1 o4 )
(on o0 o1)
)
(:goal
  (not (exists(?x -obj)(not (removed ?x))))
)
(:metric minimize (total-cost))
)
```

Listing 3.2: Problem PDDL syntax

3.5. Backtracking

The *reachability* constraints related to the inverse kinematic of the robot are computationally expensive. Computing it for each possible action (we have 5 actions in total, one grasping action and 4 pushing actions) for each object would make the computation of the predicates too expensive making the planning system quite slow. Usually the objects are inside the working space of the robot and the computation of the `ik_unfeasible` predicates is usually unnecessary.

To overcome this we used the **backtracking** technique [34, 5]. Backtracking is based on a two-fold strategy:

1. planning and checking if the plan is feasible,

2. if it is not, the state is updated with the new information and the system repeats from point 1 until a feasible plan is obtained.

Planning symbolically is very fast therefore replanning several times is not a problem. With this method the inverse kinematic will be solved only for the action we want to execute, the first one of the plan, and no time is wasted in computing the inverse kinematic for unnecessary actions. If executing the first plan's action is not possible the equivalent `ik_unfeasible` predicate is updated. This allows us to include *reachability* geometrical constraints within symbolic planning. The pseudo algorithm to get a plan is shown in Algorithm 2.

Algorithm 2 Planning procedure with backtracking.

Inputs: initial state s_0 and goal state G.

Outputs: a feasible plan or not plan at all.

```
procedure GETPLAN(s0,G)
    repeat
        plan ← GETFASTDOWNWARDPLAN(s0,G)
        if ¬EXISTSSOLUTION(plan) then return NULL
        end if
        action ← GETFIRSTACTION(plan)
        success ← HASIKSOLUTION(action)
        if ¬success then
            s0 ← UPDATEINITIALSTATE(action)
        end if
    until success return plan
end procedure
```

Observe that in this approach, the planner first considers that the actions have no restrictions regarding the reachability of the robot. Afterwards, the first action in the plan is validated. This is a lazy approach that makes the planning stage fast because the computational expensive information, defined by the *reachability* geometrical constraints, are computed if and only if required by a plan.

It is possible that an object cannot be grasped in a certain pose, because the inverse kinematic has no solution, but it can be moved in a new pose in which it can be grasped.

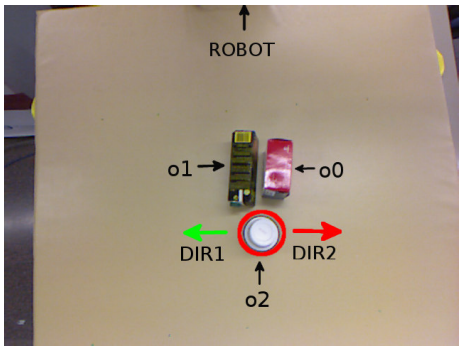


Fig. 3.3: Unfeasible plan due to reachability constraints. In this case the planner returns as action to execute grasping or pushing away the white cup (highlighted by a red circle) but it is out the configuration space of the robot and there exist no plan for that problem.

Therefore the pushing actions also include the effect that for the object of interest the grasping action is feasible after being pushed. This is usually a rare case but it might happen. In the case the object is in a pose where it cannot be neither grasped or pushed because of the inverse kinematic, first the planner will return as a solution to grasp it, then it will replan and the solution will be to push it in one direction and grasp it, and so until no action can be executed and there exist no solution for the plan.

Backtracking is used also to include advanced information when the robot must execute a pushing action. It would be useless to push an object outside the working space, therefore the robot must avoid that situation. To this aim, when the action to perform is pushing, we compute the IK for the pushing action and also for the grasping pose of the object after being pushed up to the estimated final pose. Thus, if the robot can execute the action `(push o1 dir1)` but it would push `o1` outside the working space the predicate `(ik_unfeasible_dir o1 dir1)` is set to true.

An example of how backtracking works is shown in Figure 3.3. Accordingly to our strategy, the robot cannot grasp the black or red box because the gripper would collide, and the same for pushing them. It has to interact with the white cup in order to make space to move the other objects and then grasp them. For this case the system would perform the following set of operations:

1. It first gets the following plan: `(grasp o2)`, `(push o0 dir1)`, `(grasp o1)`, `(grasp o0)`,

2. It solves the inverse kinematic for the `(grasp o2)` action, but it finds no solution and adds to the states the predicate `(ik_unfeasible_grasp o2)`,

3. It replans and gets the following plan: `(push o2 dir1)`, `(grasp o2)`, `(push o0 dir1)`, `(grasp o1)`, `(grasp o0)`,

4. It solves the inverse kinematic for the `(push o2 dir1)` action but it finds no solution, so the predicate `(ik_unfeasible_dir o2 dir1)` is added to the states,

5. It continues until there exists no solution for the planning problem.

It may happen that the object outside the working space of the robot blocks the execution of the task because it is impossible to achieve the goal to grasp all the objects since one is outside the working space. This happens when all the `ik_unfeasible` predicates for an object are set to true. When this situation occurs the object is removed from the goal so that the rest of the goal can be completed if possible. That is, *sub-goals* are introduced whenever the original goal is not achievable.

3.6. Replanning

It is important to point out again that the system is deterministic, meaning that all the actions are supposed to give the resultant state with a probability of 1. Clearly the biggest uncertainty is related to the pushing action; the method used to select the pushing directions does not take into account reliably the geometry of the object and the trajectory will be unlikely the desired one but a similar one. Overall, the planner is considering to push the manipulated object in a position in which it is graspable and isolated, but the may be not isolated and it may be not graspable if the trajectory is different from the expected one. This is another uncertainty of the pushing action due to the lack of geometrical information. Also the grasping action has its uncertainties due to perception noise and difficulties in grasping particular shapes (e.g. cylindrical objects may be hard to grab if the grasp is not stable).

To handle the uncertainty **replanning**[27] is used. After the execution of an action the planner gets a new depth image from the depth sensor, it segments the scene, it recomputes the states and obtains a new plan. In this way the planner considers a totally new problem and all the uncertainties associated to the previous plan are solved by the current one.

4. Perception subsystem

This chapter describes in detail the perception subsystem. First, the software tools used to develop it are described. Second, the depth vision system is discussed with a comparison between the Microsoft Kinect sensor and the stereo vision system implemented at DTG. Third, the object segmentation to locate the objects onto the table is discussed. Finally, how the state is generated is described.

4.1. Software tools

The perception subsystem was developed using the following open-source libraries:

- **OpenCV:** The *Open Source Computer Vision Library* [24] is a popular open-source library, released under the BSD license, that implements several recent state of the art algorithms. This library is used for the implementation of the stereo system. The version used in this Thesis is the last stable version (OpenCV 3.2).

- **PCL:** The *Point Cloud Library* [48] is an open-source project, released under the BSD license, for 3D images processing and it implements several state of the art algorithms. This project has been developed at *Willow Garage*[1] from 2010. The version used in this Thesis is the 1.8 which is currently an unstable release. This library is used to locate objects and model them.

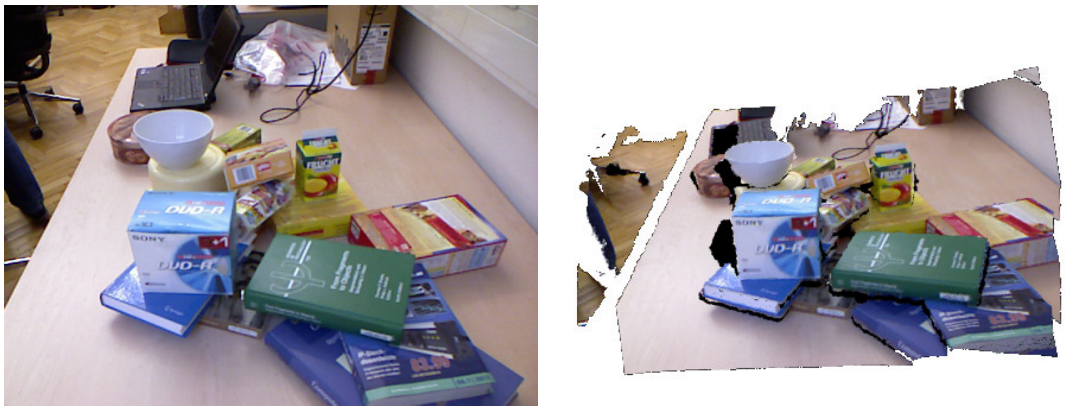
- **FCL:** The *Flexible Collision Library* [44] is an open-source library, released under the BSD license, that implements several algorithms for fast collision checking and proximity computation. This library is used to compute the *relational* geometric constraints.

4.2. Depth vision

Several computer vision algorithms, based on two-dimensional RGB images, can precisely detect objects on the images, and if the camera is calibrated and the object's model

[1] http://www.willowgarage.com/

known a priori, they can also locate it in the space. With unknown objects this task becomes hard, overall for what it concerns the estimation of the object pose. Eventually, this scenario could be easier whether the grabbed image of the scene has spatial information. To this aim, several techniques have been developed to provide a three-dimensional representation of the observed scene.



(a) 2D RGB image of the scene **(b)** 3D image/pint cloud of the scene

Fig. 4.1: Point cloud example: in 4.1b is shown the point cloud corresponding to 4.1a which is the RGB image grabbed by the Kinect's color sensor. This point cloud has been capture using the Microsoft Kinect sensor.

In this extension, 3D images are denoted as *point cloud* (Figure 4.1), this is a vector of points with physical coordinates and eventually with rgb information. The mainly used technologies to obtain such a 3D representation are:

- Laser scanning: these scanners project a laser beam in a particular direction and then analyse either the elapsed time or the phasing of the reflect beam. This is done for each direction of interest. They are particularly precise but cannot provide a matrix sampling of the scene but they generally provide a line sampling. This make them unsuitable for real-time applications whether a matrix sampling is needed.

- Structured Light: by illuminating the scene with a specially designed light pattern a single camera can determine the depth by analysing how the pattern is deformed in the scene. This kind of sensors are not expensive and can work with good frame rates but experience problems with particular materials because of light reflection.

- Time-of-flight: similarly as lasers, this technology uses a light source and the camera gathers the reflected light. The incoming light has a delay with respect the light

source depending on the distance of the surface of reflection. They can grab images with a high rate but the resulting point cloud has generally small resolution.

- Stereo triangulation: this technology is based on multiple monocular cameras that, by triangulation, can measure the depth of the points in the overlapping field of vision of the two cameras. This system can return a very dense point-cloud accordingly to the cameras resolution and can be easily designed to estimate the depth even for objects far from the cameras. In contrast it is computational expensive.

In the initial version of this thesis [12] a structured light sensor (Microsfot Kinect) was employed. In the extension of the thesis we wanted to explore the alternative of a stereo system to compare the results with the Kinect.

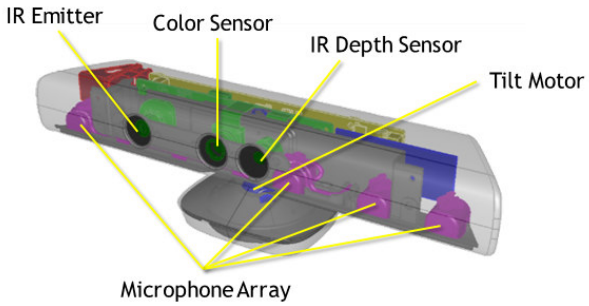


Fig. 4.2: The main parts that conform the Microsoft Kinect sensor.

Let briefly introduce the Microsoft Kinect sensor to analyse its benefits and drawbacks. In Fig. 4.2 some relevant parts of the sensor are highlighted; the most notable ones are the IR emitters and depth sensor. Although the depth-sensing technology is not disclosed it is known to be based on the structured light principle [58]. The IR emitter is an IR laser that passes through a diffraction grating and turns into a set of IR dots. These dots are observed by the IR sensor and, since the relative transformation between the emitter and the sensor are known, by triangulation the depth is computed.

The Kinect sensors can work approximatively up to 30 frame per seconds making it suitable for real time applications. On the other hand, it has drawbacks regarding the quality of depth estimation. In fact, for small objects the estimation is poor and some surfaces does not reflect the infra-red pattern as well as others, as results these objects could be not seen by the Kinect (e.g. the plastic bottle in Figure 4.3). Moreover, it cannot measure points that are too close or too far the camera. These are the main reasons why not all the points of the 2D image of Fig. 4.1 appears in the point cloud.

(a) Image RGB **(b)** Point cloud (top view) **(c)** Point cloud (lateral view)

Fig. 4.3: Point cloud of a plastic bottle captured by the Kinect. The sensor was able to correctly estimate the depth olny for the bottle's label.

Instead, a stereo system provides a dense and precise point cloud and have no restrictions on the materials of the observed objects. In contrast the computational effort is not negligible and make this system not suitable for real time applications and the field of vision may be smaller than the Kinect's. Nevertheless, the robotic system presented in this thesis is not a real-time system thus, there are no restrictive limitations on time requirements. Hence, the employment of a stereo system is beneficial because it would extent the set of objects the system can deal with. The stereo system, depicted in Figure 4.4, was built using two Basler ACE USB 3.0 scan cameras (see Appendix A) which are characterized by a high manufacture quality and are widely used for industrial processes.

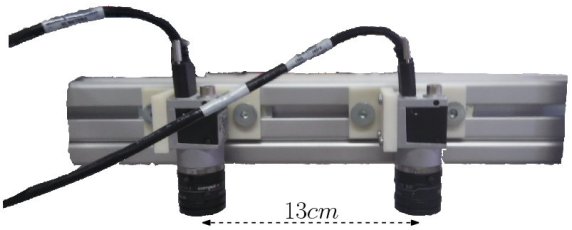


Fig. 4.4: Stereo rig: vertical arrangement composed of two Basler ACE UBS 3.0 cameras distanced by 13cm.

We highlight that in the robotics laboratory of the DTG was not possible to reproduce a suitable set-up for the robotic system here presented. Thus, the stereo system has been developed and tested with real-case scenarios but not integrated within the real robotic system. However, the stereo system has been reproduced in a simulated environment in Gazebo and integrated within the simulated robotic system (Chapter 6).

In this section the theory behind the stereo vision is introduced supported by some

real-world experiments. First, the pinhole camera model, distortions and calibration are discussed in order to give to the reader a required knowledge to understand the stereo vision. Second, the stereo vision itself is discussed presenting all the geometrical concepts and algorithms needed by the stereo vision. Finally, a comparison between the point clouds obtained with the Kinect and the stereo system is carried out to state the best sensor for our application.

4.2.1. Camera model and distortions

There exist different models for the monocular cameras, in this thesis we focus on the simplest one; first considering the ideal model then introducing non-idealities. This model is the so-called **pinhole** model [56].

Pin hole model

This model is characterized by a single light ray entering the pinhole from any point in the scene. In a physical camera this point is projected onto an imaging surface called *imager*, forming an inverted image. The plane to which the image belongs is called *image plane*. The capture of an image is basically a projection of the physical point onto the imager through *perspective projection*. As result, the size of the image depends only on one camera's parameter called *focal length*. This is the distance between the pinhole aperture and the image plane.

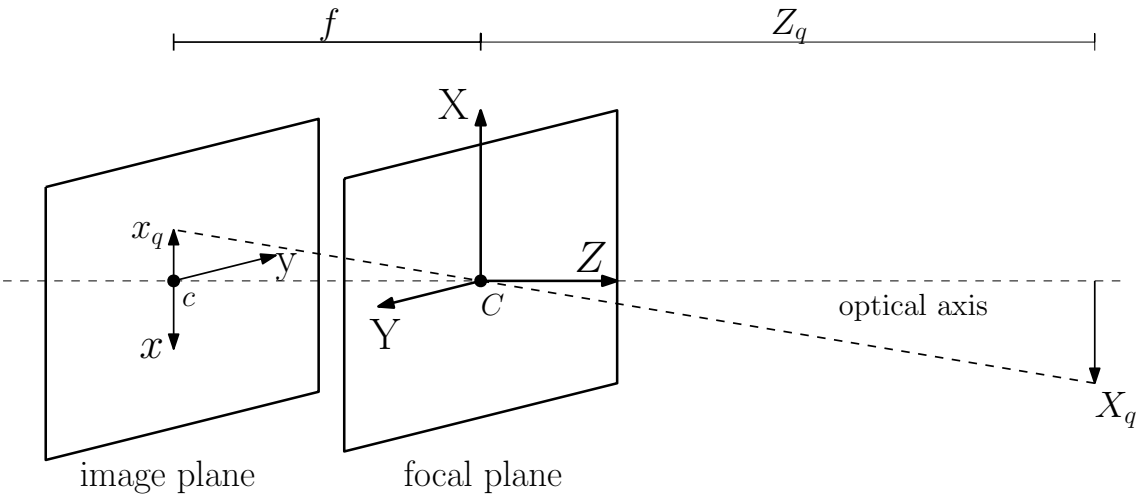


Fig. 4.5: Pinhole camera model: a pinhole (the pinhole aperture, *C*) lets through only those light rays that intersect a particular point in space; these rays then form the image by projecting the point onto the image plane.

Figure 4.5 shows a pinhole camera model, where f is the focal length of the camera, Z_q is the distance of a given point $\overrightarrow{Q}$ from the camera, X_q is the coordinate of the point orthogonal to the optical axis and x_q is the coordinate of that point in the imager.

To derive the equations for the perspective projection, the coordinate system (c,x,y) is defined such that the origin is at the point c (intersection of the optical axis and the image plane) and the axes are as the ones in the Figure 4.5. This coordinate system takes the name of *image coordinate system*. We use an extra coordinate system (C,X,Y,Z) to express coordinates in the space, where its origin is in pinhole C which is at a distance f from c, and the axes are chosen as in the figure with the Z axis parallel to the optical one. This second coordinate system is called *camera coordinate system*.

From the aforementioned definitions of camera and image coordinate systems, the relationship between the 2D image coordinates and 3D space coordinates for a given point $\overrightarrow{Q}$ can be obtained by similar triangles as:

$$\frac{x_q}{f} = \frac{X_q}{Z_q} \tag{4.2.1}$$

Notice that Eq. 4.2.1 holds for the X-Z plane, an analogous relationship is obtained for the Y-Z plane.

From a geometric point of view, there is no difference to replace the image plane by a virtual image plane located on the other side with respect the focal plane and changing the orientation of the x and y axes (Figure 4.6). In the camera coordinate system, an image point (x,y) has 3D coordinates (x,y,f).

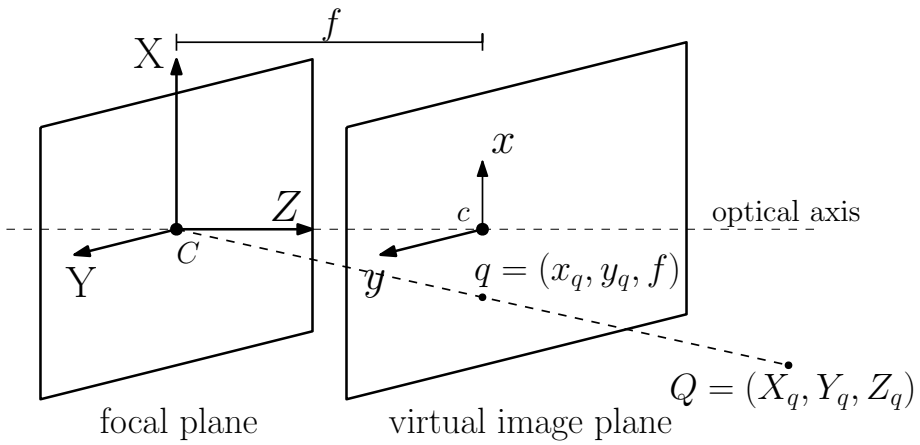


Fig. 4.6: Pinhole camera model with a virtual image plane

The point c of the intersection of the image plane and the optical axis is referred to as the *principal point*, while the point C is also called *center of projection*. For a complete

and robust model we should consider that the principal point could not be exactly at the center of the imager. This is mainly due to the production process that would require a micron accuracy in posing the imager. Thus, there could be some offset in the coordinates of a point in the image plane due to the displacement of the principal point. This offset is modelled by parameters c_x and c_y. A generic point $\overrightarrow{Q}$ in the physical world, whose coordinates are (X_q, Y_q, Z_q), is projected via a *perspective transform* onto the imager at some pixel location, given by $\overrightarrow{q} = (x_q, y_q)$, in accordance to the following relationships [26]:

$$\begin{aligned} x_q &= f_x \frac{X_q}{Z_q} + c_x \\ y_q &= f_y \frac{Y_q}{Z_q} + c_y \end{aligned} \tag{4.2.2}$$

Note that (x_q, y_q) are pixel coordinates of the projected point onto the imager, and not the physical coordinates in image coordinate system. Thus, let's introduce the *pixel coordinate system* that represents a point of the imager as pixel of the image rather than a 2D point belonging to the image plane.

In the previous equations two focal lengths have been introduced. The reasons is that the individual pixel on a typical low-cost imager are rectangular rather than square. The focal length f_x is actually the division of the physical focal length f of the lens, whose measurement unit is *mm*, and the size s_x of the imager pixels, whose measurement unit is $mm/pixels$. Hence, the focal length f_x is measured in $pixels$, in accordance to Eq. 4.2.2. So even the principal point is measured in $pixels$.

The perspective transform that maps $\overrightarrow{Q}$ to $\overrightarrow{q}$ can be denoted by a matrix using the homogeneous coordinates. These coordinates associated with a point in a projective space of dimension n are typically expressed by a $(n+1)$-dimensional vector and all the points that have proportional homogeneous coordinates are equivalent. In this case, the image plane is the projective space and it has two dimensions; thus, we represent points on that plane as a three dimensional vector $\overrightarrow{q} = (x_q, y_q, z_q)$. Given that all the points in the projective space having proportional values are equivalent, we can obtain the pixel coordinates by dividing the vector by z_q. This notation allows as to rearrange the parameters f_x, f_y, c_x, c_y into a single 3×3 matrix called *camera intrinsics matrix*. The perspective projection is then given by the following formula:

$$\overrightarrow{q} = M \cdot \overrightarrow{Q} \tag{4.2.3}$$

where

$$\overrightarrow{q} = \begin{bmatrix} x_q \\ y_q \\ z_q \end{bmatrix} \qquad M = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \qquad \overrightarrow{Q} = \begin{bmatrix} X_q \\ Y_q \\ Z_q \end{bmatrix}$$

Performing the multiplication we obtain

$$\overrightarrow{q} = \begin{bmatrix} x_q \\ y_q \\ z_q \end{bmatrix} = \begin{bmatrix} f_x X_q + c_x Z_q \\ f_y Y_q + c_y Z_q \\ Z_q \end{bmatrix},$$

dividing $\overrightarrow{q}$ by z_q the result is the relationship of Eq. 4.2.2.

Distortions

So far the ideal pinhole model has been considered, but this model differs from the reality because in the ideal model only very small amount of light passes through the pinhole. In practice, in order to capture an image the exposure time should be very long and unsuitable for real applications. To capture images at a faster rate the camera needs to gather more light over a wider area and focus it to converge to the point of projection. This is done by a lens, which can increase the image rate focusing a large amount of light on a point but this introduces distortions that need to be corrected whenever we want a correct projective transformation in accordance to the pinhole model.

In theory, it is possible producing a perfect lens but this is hard because of manufacturing. It is easier to produce a spherical lens rather than a parabolic ones, and also it is hard to perfectly align the lens and the imager. There exist different types of distortions, we only consider the two main ones, that are the ones typically considered by the literature and they are the *radial distortion* and the *tangential distortion*.

Radial distortion This type of distortion is due to the particular shape of the lens.

Lenses usually distort the image near the edges of the imager. Fig. 4.7 shows a scheme of this distortion. Rays farther from the center of the lens are bent more than those closer in, thus in the optical center the distortion is absent but it increases as we approach the edges of the imager. Given that the distortion is small it can be modelled by using the first few terms of the Taylor expansion around the center of the lens. In literature usually only the first three terms are accounted and the radial location of a point in the imager will be

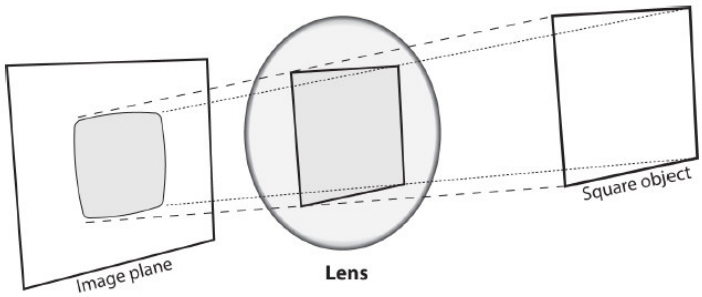


Fig. 4.7: Radial distortion: the square appears to bow out in the image plane because of accentuated bending of the rays farther from the center of the lens. (source [26])

rescaled according to the following equations:

$$\begin{aligned} x_{corrected} &= x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \\ y_{corrected} &= y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \end{aligned} \tag{4.2.4}$$

Here, (x, y) is the original location on the imager of the distorted point, $(x_{corrected}, y_{corrected})$ is the location as result of the correction and r is the euclidean distance from the optical center.

Tangential distortion This distortion is due to manufacturing process of the lens resulting from the lens not being exactly parallel to the imager. This is characterized by two additional parameters p_1 and p_2 [26] such that:

$$\begin{aligned} x_{corrected} &= x + [2p_1 xy + p_2(r^2 + 2x^2)] \\ y_{corrected} &= y + [p_1(r^2 + 2y^2) + 2p_2 xy] \end{aligned} \tag{4.2.5}$$

Thus, in total there are five distortion coefficients needed to correct the image[2]. Nevertheless, due to the high manufacturing quality the cameras lenses we used introduced very low distortion to the images.

Calibration

The calibration of a camera aims to estimate both the intrinsic and distortion parameters of the camera.

The method we use for calibration is based on viewing a known pattern from different angles [57]. The pattern is designed in manner to have many easily identifiable points.

[2]There could be more parameters if we consider more terms of the Taylor expansions for the radial distortions. Moreover, there exist different models of distortions.

Having a different set of points per angle of view we can obtain the distortion parameters by means of optimization. Knowing where the points are projected onto the imager for a certain view, and knowing the structure of the pattern we can estimate where this points should be projected onto the imager.

The pattern used in this work is a flat chessboard with white and black squares, whose point of interest are the inner corners of the squares. In total we used 14 views to calibrate the camera, few of them are shown in Figure 4.8. With such a number of views a better estimate, robust to noise, is achieved.

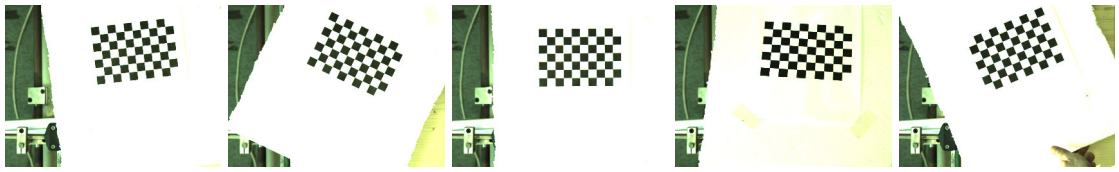


Fig. 4.8: Some views of the chessboard used for calibration.

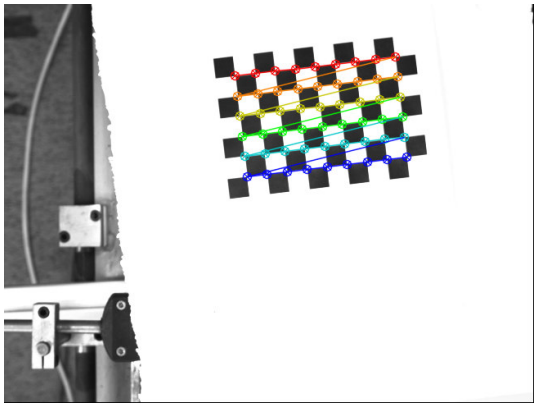


Fig. 4.9: Detected inner chessboard corners in the first view of Figure 4.8.

The detected corners (Figure 4.9) undergo a *perspective transformation* when viewed through a pinhole, this transformation is described by a 3×3 *homography* matrix. Since we work with planar surfaces, the homography transformation is actually a *planar homography* because the points, which belong to the chessboard plan, are projected onto the image plane. In computer vision, a *planar homography* is a projective mapping from one plane to another. This mapping can be expressed in terms of matrix multiplication if homogeneous coordinates are used to express both the point $\overrightarrow{Q'} = (X'_q, Y'_q)$, expressed with respect a frame attached to the chessboard plane, and the relative point on the imager $\overrightarrow{q}$.

Thus, the mapped point $\overrightarrow{q}$ is obtained from $\overrightarrow{Q'}$ by the following formula:

$$\overrightarrow{q} = s \cdot H \cdot \overrightarrow{Q'} \qquad \overrightarrow{Q'} = \begin{bmatrix} X'_q \\ Y'_q \\ 1 \end{bmatrix}, \overrightarrow{q} = \begin{bmatrix} x_q \\ y_q \\ 1 \end{bmatrix} \tag{4.2.6}$$

The term s is an arbitrary scale of factort and it has been introduced to make explicit that the homography is defined up to scale factor s. Notice that points $\overrightarrow{Q'}$ are straightforward to compute because the square size and the number of corners to detect are known a priori. Then, points $\overrightarrow{q}$ are detected in the image as shown in Figure 4.9 and only the map must be estimated.

The matrix H embeds two parts: the physical transformation W between the chessboard plane and the image plane, and the projection which introduces the camera intrinsic matrix M. Then, Eq. 4.2.6 can be reformulated as:

$$\overrightarrow{q} = s \cdot M \cdot W \cdot \overrightarrow{Q'}. \tag{4.2.7}$$

The homography relates the positions of a point $\overrightarrow{P_{src}}$ on the chessboard (expressed with respect to a 2D frame attached to the chessbaord) to a point $\overrightarrow{P_{dst}}$ on the imager by the following equations:

$$\overrightarrow{P_{dst}} = \begin{bmatrix} x_{dst} \\ y_{dst} \\ 1 \end{bmatrix} = H \cdot \overrightarrow{P_{src}} \qquad \overrightarrow{P_{src}} = \begin{bmatrix} x_{src} \\ y_{src} \\ 1 \end{bmatrix} = H^{-1} \cdot \overrightarrow{P_{dst}} \tag{4.2.8}$$

Fixing the scale factor s, the H matrix has 8 DOF [3]. Each 2D point generates two constraints on H, one per coordinate. The results is that we need at least 4 points to estimate the homography; further points bring redundant information useful for the robustness of the estimate.

The rotation is described by three angles and the translation is defined by three displacements, hence there are six geometrical unknowns per view. This unknowns are called *extrinsic* because they do not refer to the camera model.

Since we want to estimate the camera intrinsic parameters and not just the homography, it turns out that we have 10 unknowns (6 for the extrinsic ones plus 4 for the camera intrinsic ones). Thus a single view would not suffice, and each new view would add new 6 extrinsic parameters to estimate given that the chessboard is in a different pose.

In addition to intrinsics parameters also the distortions ones must be estimated. In total there are 4 camera intrinsics unknowns (f_x, f_y, c_x, c_y) and 5 distortions unknowns $(k_1, k_2, k_3, p_1, p_2)$, this set of 9 parameters is called *intrinsics*.

Supposing to capture K images of the chessboard in different poses, how many views are needed to solve the calibration problem? We have to consider that:

- K images of the chessboard provide $2 \cdot 4 \cdot K = 8 \cdot K$ constraints, this because only 4 corners are worth of information and they add two constraints each one.
- There are 9 intrinsics and 6 extrinsic per view to estimate.

Thus, the number of view to solve the calibration problem has to suffice the following inequality:

$$\begin{aligned} 8 \cdot K &\geq 6 \cdot K + 9 \\ K &\geq \frac{9}{2} = 4.5 \end{aligned} \tag{4.2.9}$$

Thus, the minimum number of views to solve the calibration problem is 5. Ignoring the distortion parameters 2 views would suffice.

In our experiment we took 14 views of the chessboard in order to have a good robustness. This disparity between the theoretically minimal 5 images and the practically required 10 or more views is a result of the very high degree of sensitivity that the intrinsic parameters have on even very small noise.

To estimate the parameters the function `cv::calibrateCamera()` [24] of OpenCV has been used, which is based on the algorithms presented by Zhang [57] and Bouguet [7] that solve an analytical solution followed by non-linear optimization. The estimated intrinsic camera matrix, for the first camera, is:

$$M_1 = \begin{bmatrix} 7375.4928 & 0 & 1293.689 \\ 0 & 7368.388 & 961.901 \\ 0 & 0 & 1 \end{bmatrix}$$

The data here reported are in pixel coordinates. It is worth to mention that the pixel coordinate system has its origin at the top left corner of the image, the x coordinate refers to the columns and the y coordinate refers to the rows accordingly to a matrix representation of the image. The physical focal length can be obtained by $f = f_x \cdot 2.2\mu m = 0.01622m$, where $2.2\mu m$ is the pixel width (see Appendix A). A similar results can be obtained with

f_y. Notice that the lens that mount the camera has a focal length of $16mm$, accordingly to the constructor information, thus the estimate matches with the real focal length. Next, the offset of the principal point is $c_x = 1293.68 pixels$ and $c_y = 961.901 pixels$ which approximatively corresponds to the center of the image in pixels coordinates that is given by $(1295, 971)$.

Undistortion With the estimated distortion coefficients we can now remove their effects obtaining a correctly mapping between a 3D point and its projection onto the imager. Then, the image is ready to be used for stereo vision as described in the next section.

When performing undistortion we use a map that specifies where each pixel of the input image goes to in the output image. Such a map is called *undistortion map* [3].

An example of the effect of the distortion is shown in Fig. 4.10. In the image on the left is possible appreciating the effect of the radial distortion since the mirror on the top looks bent. On the right image the same image undistorted is shown; it can be appreciated that after correction the mirror's edge is straight and the image looks more natural and similar to what we would see by naked eye.

Fig. 4.10: Camera image distorted (left) and undistorted (right). (source [26])

4.2.2. Stereo Vision

Stereo vision works on the principle of binocular vision. That is, having two frames, one captured by each camera, points are matched between the two frames and, knowing the relative transformation between the two cameras, it is possible retrieving the depth

[3] http://docs.opencv.org/2.4/modules/imgproc/doc/geometric_transformations.html#initundistortrectifymap

information of a given point by means of similar triangles. Notice that in Eq. 4.2.2 there are two equations and three unknowns (X_q,Y_q,Z_q), thus it is not possible determining the coordinates in the space of point $\overrightarrow{Q}$. To this aim a second camera is used.

We refer to the two cameras as left and right camera[4], such an arrangement of the cameras is called horizontal. In this work, however, the arrangement was vertical such that the left camera is actually the below camera and the right camera is the top one. Despite this, all the theory here described works exactly the same for both arrangements and it is more intuitive using a horizontal arrangement.

Having the cameras calibrated it is possible unambiguously project points in the physical world onto the image in pixel coordinates. This is a fundamental process in order to retrieve the depth. Having a certain pixel in the left image, described by a feature, we can look for a match (i.e. we look for the same feature) in the right image, then by triangulation the depth is retrieved. Generally, the feature relative to a pixel captures information of surrounding pixels to provide a description metric of such a pixel.

This process is complex and involves many expedients (Figure 4.11):

1. Remove radial and tangential distortions from each image to get the undistorted images.

2. Project the two frames on the same plane and make them parallel (i.e. the pixels on a row in the left image belong to the same row on the right image), this process is known as *rectification*. As results, the images are row-aligned for a horizontal arrangement or column-aligned for a vertical arrangement.

3. Find the same features in the left and right camera views, this process is called *stereo matching*. As result, a *disparity map* is produced, which specifies the difference between the pixel coordinates of a point from left and right images. This difference holds part of the depth information.

4. Knowing the geometric transformation between the right camera with respect to the left one, by triangulation, the depth of the pixels can be computed using the disparity map. This final step is called *reprojection* and it returns a point cloud.

[4]In this thesis, the left and right cameras may also be called first and second cameras, respectively.

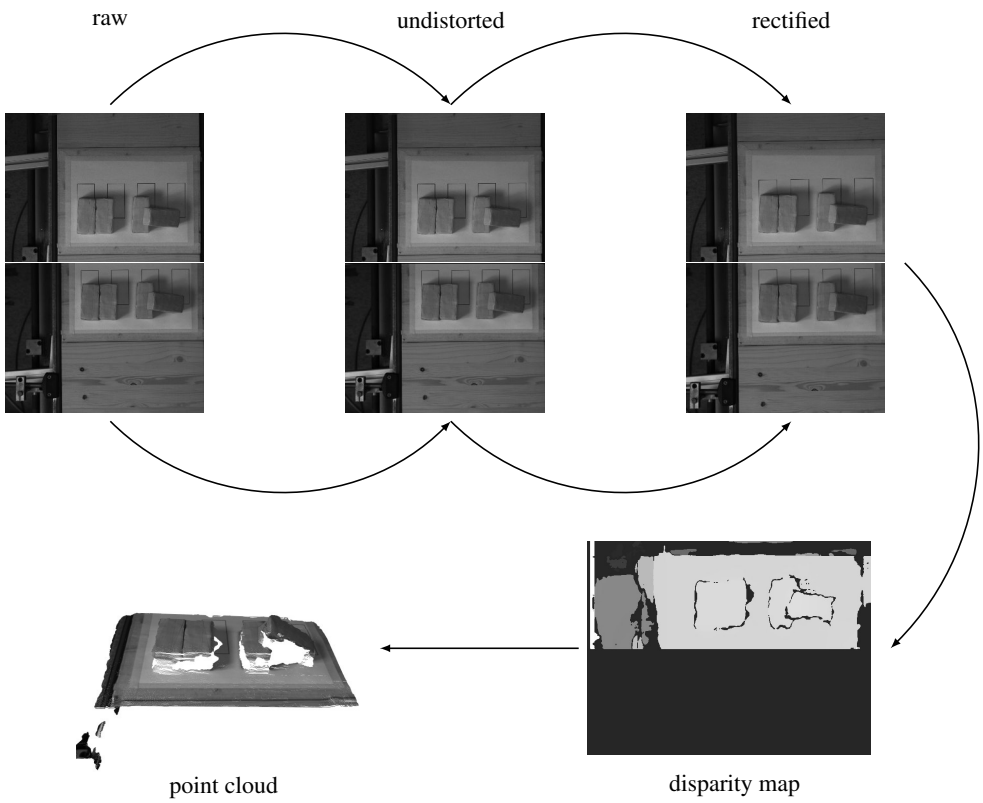


Fig. 4.11: Stereo vision stages for a vertical cameras arrangement: the input raw images are undistorted (here it is not appreciable since the cameras mounted lenses with ultra-low distortion), rectified and a disparity map is computed by matching the pixels in the rectified images. Finally, the pixels of the disparity map are reprojected onto the 3D space forming a point cloud.

Triangulation

Let's assume a perfectly undistorted, aligned and measured stereo system as shown in Figure 4.12. The two cameras have perfectly coplanar image planes with exactly parallel optical axes and are at a known distance apart and have the same focal length. Assume also that the *principal points* c_x^{left} and c_x^{right} have the same pixel coordinates and that the images are row-aligned (this arrangement is called *frontal parallel*). The pixel coordinate systems of the left and right images have origins at upper left corner in the image, and pixels are denoted respectively by (x_l, y_l) and (x_r, y_r). The center of projections are at C_l and C_r with principal ray intersecting the image plane at the principal point. After rectification, the two images are row-aligned, that is they are coplanar and horizontally aligned, one displaced from the other by T.

A point $\overrightarrow{Q} = (X_q, Y_q, Z_q)$ in the first camera coordinate system, that can be seen by both cameras, is projected as $\overrightarrow{q^l}$ and $\overrightarrow{q^r}$ onto the two images plane, whose the horizontal pixel

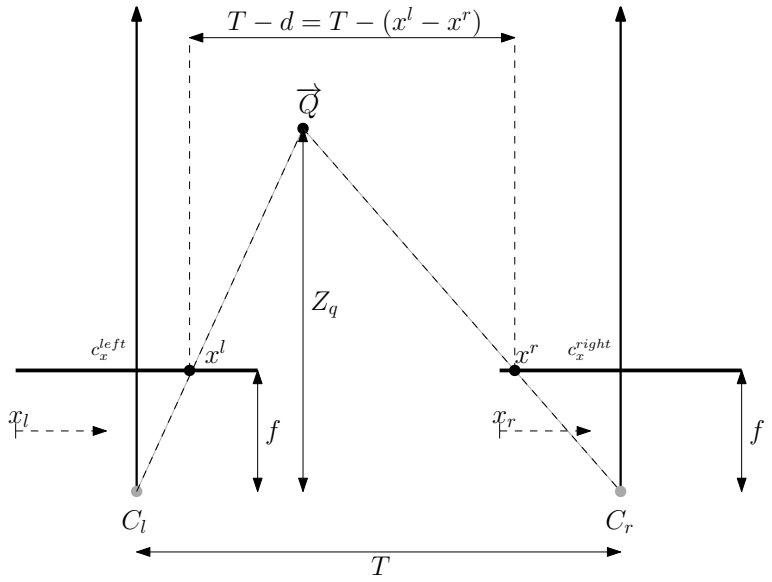


Fig. 4.12: Perfectly undistorted and aligned stereo rig and known correspondence, the depth Z_q can be calculated by similar triangles.

coordinates are x^l and x^r respectively. In this simplified case the disparity is $d = x^l - x^r$; note that it is always positive since a point on the right image will be always projected onto the right image plane to a smaller coordinate than x^l. By triangulation the following equations holds:

$$\frac{T-d}{Z_q - f} = \frac{T}{Z_q} \tag{4.2.10}$$

Thus, the depth can then be computed as follows:

$$Z_q = \frac{fT}{d} \tag{4.2.11}$$

The disparity tends to zero as the observed point tends to move away from the camera. Since the depth is inversely proportional to the disparity the depth is severely affected by an erroneous disparity value when this is small and vice versa. That is, stereo vision systems have high depth resolution only for objects close to the camera and it decreases as these get far. To improve the precision the distance T must be increased.

The general idea is operating in the ideal case of a perfectly undistorted, aligned and known cameras' displacement. The undistortion was discussed in the previous subsection. The alignment of the imagers is a surrealistic hypothesis but it can be achieved after *rectification*. The measurement of the cameras' displacement is done by calibrating the stereo system, here not only the two cameras are calibrated in manner to have the same focal distance but also the geometrical transformation between the two camera

frames is estimated. Despite rectification, when designing a stereo system it is important to arrange the cameras approximatively frontal parallel as possible and as close to horizontally aligned as possible. Second, the cameras need to be synchronized in order to capture simultaneously the image of the scene, otherwise the stereo system will work only in static environments. Last but not least, the stereo system has to be designed in manner that there is sufficient overlapping area between the field of visions of the cameras; thus, further they are from each other, and closer to the working plane, better the depth measurement is, because the disparity is larger, but the overlapping area is smaller.

Stereo calibration

A fundamental information for the stereo system is the relative pose of the second camera with respect to the first one. This has to be estimated through *stereo calibration*. The geometrical transformation is composed by a rotation matrix R and a translation vector T that can be estimated by using the same chessboard used for the calibration of each single camera. To this aim, the two cameras have to capture the same view of the chessboard. Then knowing the position of the inner corners in the left and right images, using the prior knowledge of the calibration of single camera (i.e. the image are undistorted), it is possible estimating R and T.

Given K views of the chessboard we obtain K rotation matrices and translation vector, then the median values are extracted as the initial approximation of the true solution, then the value is refined running a Levenberg-Maquardt iterative algorithm to find the local minimum of the reprojection error of the chessboard corners for both camera views. This is automatically done by the function `cv::StereoCalibrate()` [24]. This function can also calibrate the single cameras forcing them to have the same focal length.

For our vertical arrangement, the two cameras were displaced by 13 centimeters and the transformation estimate of the calibration were composed of a practical identity matrix for the rotation matrix and the translation vector $T = [-0.06cm, 12.988cm, -0.145cm]$. These results match with the real arrangement.

Stereo rectification

Stereo rectification is the process to correct the images of each camera so that they are row-aligned. The geometry it is based on is called *epipolar geometry*. This combines two

pinhole models that have different center of projection, C_l and C_r, and different projective planes, Π_l and Π_r. Epipole $\overrightarrow{e_l}$ on image plane Π_l is defined as the image of the center of projection of the other camera C_r [26]; analogously for epipole $\overrightarrow{e_r}$. The plane in space formed by a generic point $\overrightarrow{Q}$ and the two epipoles is called *epipolar plane* and the lines $\overrightarrow{q_l}\,\overrightarrow{e_l}$ and $\overrightarrow{q_r}\,\overrightarrow{e_r}$ are called *epipolar lines* (where $\overrightarrow{q_l}$ is the projection of $\overrightarrow{Q}$ on Π_l).

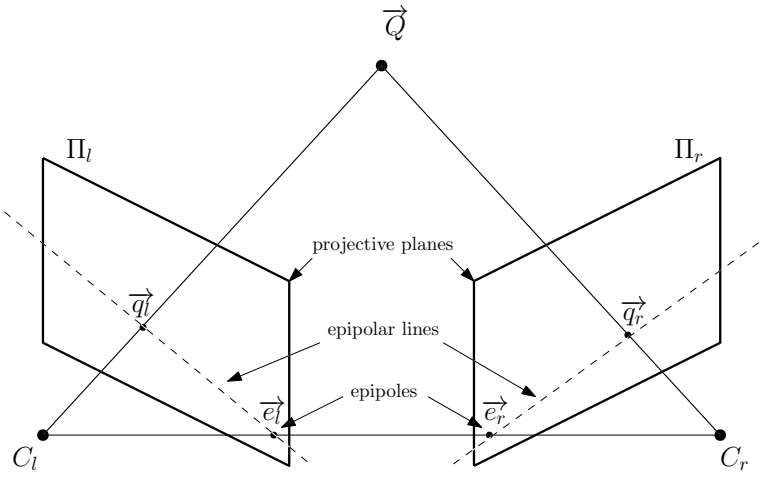


Fig. 4.13: Epipolar geometry entities. Epipoles are located at the intersection of the line joining C_l and C_r and the two projective planes, the epipolar plane is formed by points $\overrightarrow{Q}, \overrightarrow{e_l}, \overrightarrow{e_r}$.

Epipolar lines are very useful to address the problem of triangulation. In fact, considering $\overrightarrow{Q}$ projected onto Π_r, the right camera does not have any information about the depth of $\overrightarrow{Q}$ and it can be any point laying on the line formed by C_r and $\overrightarrow{q_r}$. If that line is projected onto Π_l the epipolar line formed by $\overrightarrow{q_l}$ and $\overrightarrow{e_l}$ is obtained. In other words, the image of all the possible locations of a point seen in one imager is the line that intersects the corresponding point and the epipole on the other imager.

Thus, the epipolar geometry is fundamental for stereo vision for three main reasons:

- Every 3D point, in the overlapped field of vision, belongs to an epipolar plane that intersects each image in an epipolar line.

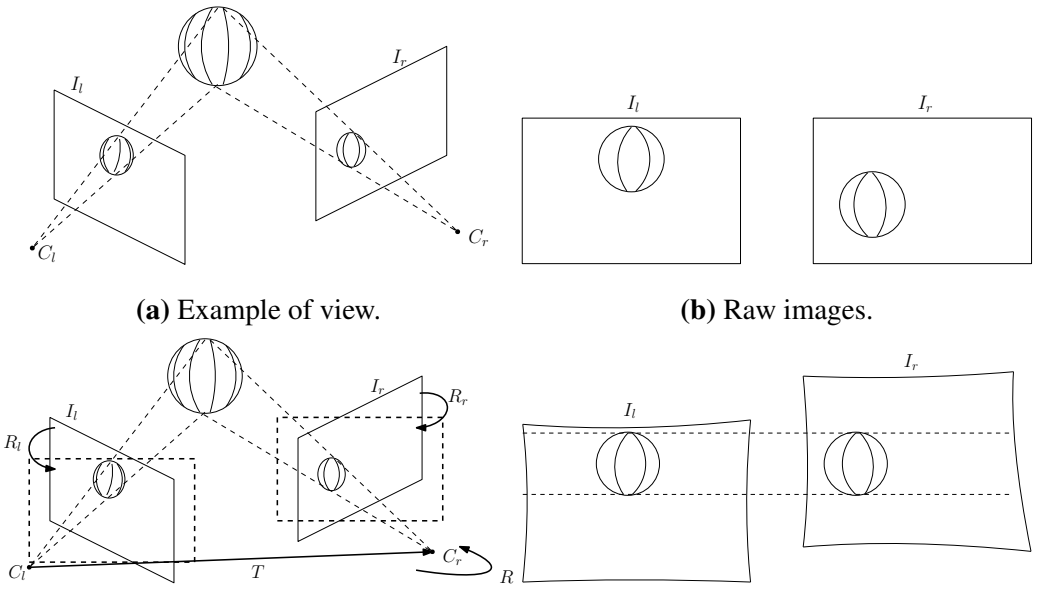
- Given a feature in one image, the corresponding feature in the other image must lay on the corresponding epipolar line (*epipolar constraint*).

- The *epipolar constraint* narrows the search space for all the possible correspondences for a given feature since looking along the corresponding epipolar line suffices. This helps to make the algorithm faster and more robust against false matches.

For a parallel frontal stereo system, since the image planes of the cameras are almost coplanar the results is that the epipoles do not belong to the image and they tend to be at infinity. Thus, the epilines that we expect are horizontal and different from the one in Figure 4.13.

Having the transformation matrices obtained by the stereo calibration the next step is to rectify the images, the result of this operation is that epipolar lines are the same on both images. The images are row-aligned[5], so that the algorithm devoted to look for a correspondence of a pixel in the i-th row can look for them on the other image in the same row and not over all the image.

Rectification In the general case, different from the parallel cameras arrangement, the corresponding points are not defined in a common reference frame and the disparity cannot directly be measured.



(a) Example of view. **(b)** Raw images.

(c) Projective plane. **(d)** Rectified undistorted images.

Fig. 4.14: Rectification accordingly to Bouget's algorithm: the ball is captured by both imagers I_l and I_r (a), but since they are not parallel (their projective planes Π_l and Π_r are different) and row-aligned the ball appears to be at the bottom of the right image (b). With rectification the images are projected onto a parallel plane (c) and they are now row aligned (d). Notice that the images have been undistorted before rectification.

The goal of rectification is to project the two images such that they belongs to the same plane and they are row aligned in a way the disparity can straightforwardly be measured

[5]For a vertical stereo system the result of rectification is that the images are column-aligned.

(Figure 4.14). The result of this alignment is that the epipoles itself are located at infinity, as introduced in the previous paragraph. Since there is an infinity number of possible parallel planes to choose from, more constraints need to be added. These constraints depend on the algorithm used. In this work the Bouget's algorithm[6] is used, which is the most common.

The algorithm attempts to minimize the reprojection distortions while maximizing the common viewing area. To minimize image reprojection distortion, the rotation matrix R that describes the rotation of the second camera with respect to the first one is split in half between the cameras, that is in matrices r_l and r_r. In other words, the two images are both rotated by half rotation in order to lay on an intermediate plane. Each camera's principal axis ends up parallel to the vector given by the sum of the two original principal rays. Such a rotation makes the images coplanar but not row-aligned. When two images a row aligned it means that the epipoles go to inifinity at that the epipolar lines are horizontal (or vertical in case of a vertical arrangement). To achieve this situation a second rotation matrix R_{rect} is used and created as follows:

- The first row e_1 of R_{rect} is given by a normalized vector with the same orientation of the translation vector $T = [T_x, T_y, T_z]$ that expresses the translation of the second camera:
$$e_1 = \frac{T}{||T||}$$
- The second row must be an unit vector orthogonal to e_1, to this aim we choose a normalized vector given by the cross product of the principal ray $pr = [0,0,1]$ and e_1:
$$e_2 = \frac{[-T_y, T_x, 0]}{\sqrt{T_x^2 + T_y^2}}$$
- The third row is given by the vector e_3 orthogonal to both e_1 and e_2, thus:
$$e_3 = e_1 \times e_2$$

[6]Jean-Yves. Bouguet never published this algorithm although he implemented it in his well-known Camera Calibration Toolbox for Matlab.

Matrix R_{rect} is then

$$R_{rect} = \begin{bmatrix} e_1 \\ e_2 \\ e_3 \end{bmatrix}, \tag{4.2.12}$$

this matrix rotates the first camera to the center of projection so that the epipolar lines become horizontal (or vertical) and epipoles are located at infinity. Therefore, the row-alignment is then given by the combination of R_{rect}, r_l and r_r matrices:

$$\begin{aligned} R_l &= R_{rect} r_l \\ R_r &= R_{rect} r_r \end{aligned} \tag{4.2.13}$$

Next, also the projection matrices P_l and P_r, which embed the camera matrices, for the left and right cameras, are computed:

$$\begin{aligned} P_l &= \begin{bmatrix} f_{x_l} & 0 & c_{x_l} & 0 \\ 0 & f_{y_l} & c_{y_l} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \\ P_r &= \begin{bmatrix} f_{x_r} & 0 & c_{x_r} & T_x f_{x_r} \\ 0 & f_{y_r} & c_{y_r} & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \end{aligned} \tag{4.2.14}$$

In case of vertical stereo P_r is:

$$P_r = \begin{bmatrix} f_{x_r} & 0 & c_{x_r} & 0 \\ 0 & f_{y_r} & c_{y_r} & T_y f_{y_r} \\ 0 & 0 & 1 & 0 \end{bmatrix} \tag{4.2.15}$$

These projection matrices take a 3D point in homogeneous coordinates to a 2D point in homogeneous coordinates as follows:

$$P_l \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ w \end{bmatrix} \tag{4.2.16}$$

where the screen coordinates can be calculated as $(x/w, y/w)$. Image points can also be projected to 3D points in homogeneous coordinates given their screen coordinates and the

camera intrinsics matrix. Recalling the following relationships:

$$\begin{aligned} x &= f\frac{X}{Z} + c_x \\ y &= f\frac{Y}{Z} + c_y \\ Z &= \frac{fT_i}{d} \end{aligned} \tag{4.2.17}$$

where T_i is either equal to T_x for horizontal stereo or T_y for vertical stereo. Putting these equations in matrix form we obtain the reprojection matrix Q:

$$Q = \begin{bmatrix} 1 & 0 & 0 & -c_x \\ 0 & 1 & 0 & -c_y \\ 0 & 0 & 0 & f \\ 0 & 0 & \frac{1}{T_i} & 0 \end{bmatrix}. \tag{4.2.18}$$

So, given a 2D dimensional point in screen coordinates and its disparity, which holds the depth information, we can project such a point to a 3D point in the space as follows:

$$Q \begin{bmatrix} x \\ y \\ d \\ 1 \end{bmatrix} = \begin{bmatrix} X \\ Y \\ Z \\ W \end{bmatrix} \tag{4.2.19}$$

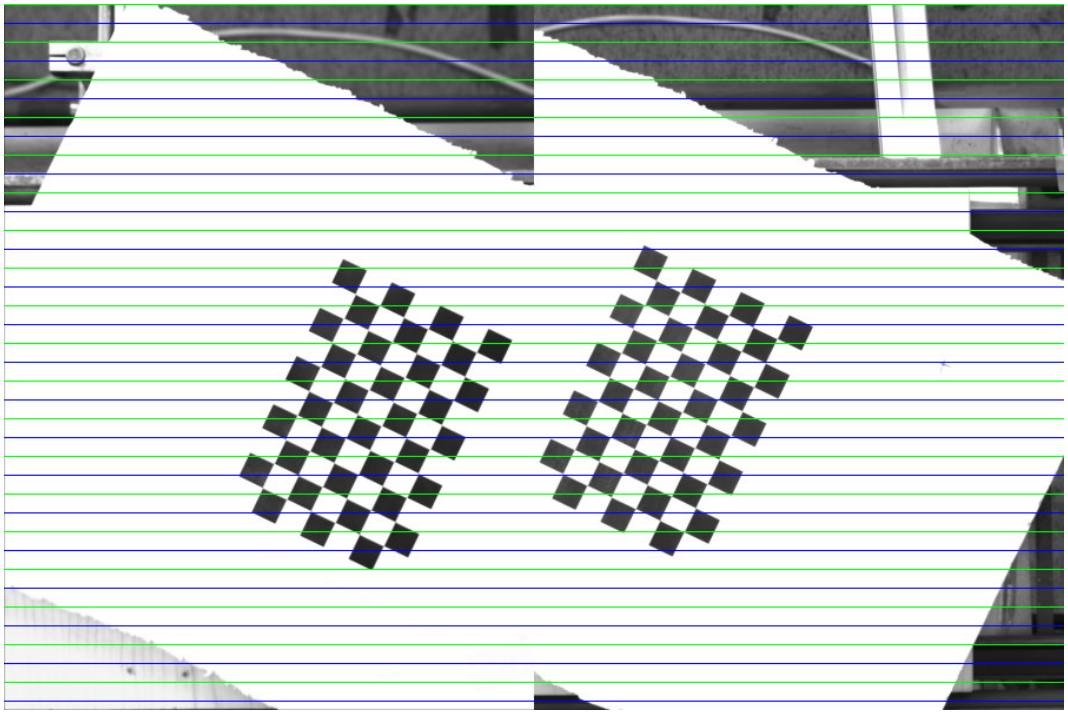
where the 3D coordinates are then $(X/W, Y/W, Z/W)$.

In Fig. 4.15 an example of the rectification on the proposed stereo system is shown. The horizontal lines serve to highlight the row-alignment of the rectified images. For a better understanding Figure 4.16 provides a 3D representation of the stereo system that is observing objects onto a conveyor belt. The image shows that the rectified images, if projected onto the conveyor belt plane, perfectly overlaps.

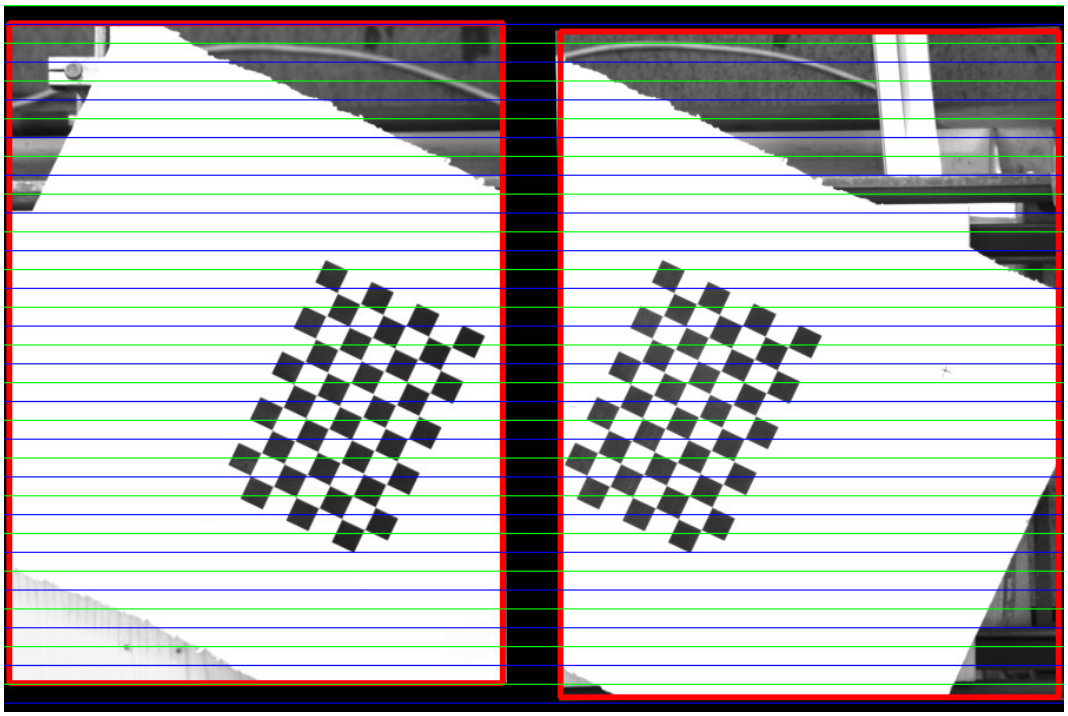
Stereo matching

Stereo matching aims to match a 3D point, in the overlapping volume of the field of visions, in two different views (i.e. look for correspondences) to compute the disparity. Knowing the physical transformation between the cameras it is possible retrieving the depth from the disparity $d = x^l - x^r$ between the corresponding points in the two different views.

Two algorithms are tested to perform this matching. The more robust one is the semi global block matching (SGBM) stereo algorithm implemented in OpenCV, which is based



(a) Unrectified images.



(b) Rectified images.

Fig. 4.15: Unrectified and rectified images.

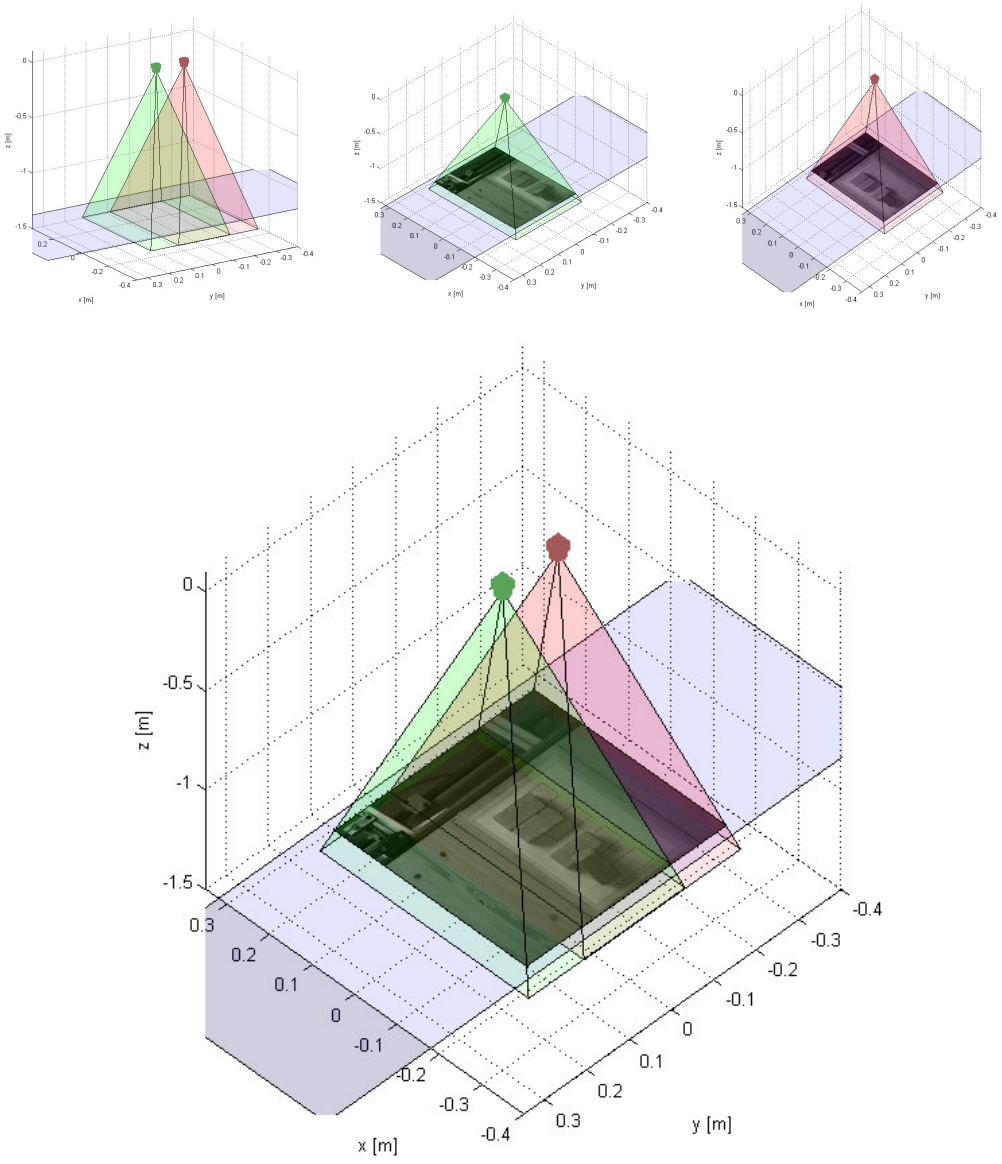


Fig. 4.16: Camera's field of vision in the 3D space. The green pyramid is the field of vision of the first camera, and the red pyramid is the field of vision of the second one. The blue plane is the plane of the conveyor belt our system is working with. This plot highlights that only for the points in the overlapping volume the depth can be measured and how the rectified images overlap, despite some occluded parts. Notice that in this plot the stereo rid is supposed frontal parallel, thus the translation vector is $T = [0, 13cm, 0]$ and the rotation is $R = I_{3\times3}$. This differs from reality but after rectification we can refer to the case of frontal parallel. The images are the ones of Figure 4.17.

on a more refined revision of the block matching algorithm (BM). BM relies on the "sum of absolute difference" (SAD) of neighbour pixels for finding correspondences between the views of the two camera. This SAD is a measure of similarity between image blocks: it is computed taking the sum of the absolute difference between each pixel in the original block and the corresponding pixel's block used for comparison. Because of the SAD metric used for matching, the algorithm is reliable only on high textured scenes.

The BM algorithm is based on three main steps:

1. Prefiltering to normalize the image brigthness and enhance texture.

2. Looking for correspondences along horizontal epipolar lines using the SAD window.

3. Postfiltering to eliminate bad correspondeces.

The correspondences are looked by sliding a SAD window[7]. For each feature in the left image the algorithm searches the corresponding row for the best match. The search is limited to the same line because after rectification the images are row-aligned. If the feature cannot be found in the other image, because the scene is too poorly textured or the feature is occluded, the algorithm neglects such a feature.

To speed up the search it can be specified the minimum disparity to start from and the maximum disparity to stop to. Recall that large disparities mean closer objects, thus, if we work with object close to the camera the search can be specified to start from a high number of disparity. These two parameters narrow down the search over the same row speeding the process up and making it more robust against false matches. In the physical world, this means that the 3D region covered by the depth estimation is controlled by the minimum and maximum disparity consider, this 3D volume is called *horopter*. Horopters can be made larger decreasing the distance of the two cameras in such a way the overlapping volume increases, reducing the focal length (i.e. increasing the field of vision of the cameras), increasing the disparity search range or by increasing the pixel width.

In our experiment, there was a conveyor belt at a distance of $Z = 1.3m$ from the cameras with blocks of size $4 \times 3 \times 8cm$ laying on it. Thus, a reasonable depth range is

[7]This metric works on image blocks, i.e. it is a window of arbitrary $n \times m$ pixels.

from $1.2m$ to $1.4m$ whose correspondence disparity values are:

$$\begin{aligned} d_{max} &= \frac{fT}{Z_{min}} = \frac{7327.73 pixels \cdot 0.13m}{1.2m} = 799 pixels \\ d_{min} &= \frac{fT}{Z_{max}} = \frac{7327.73 pixels \cdot 0.13m}{1.4m} = 680 pixels \end{aligned} \tag{4.2.20}$$

The smallest depth range resolution ΔZ directly depends on the increment of disparity Δd the algorithm uses to look for correspondences:

$$\Delta Z = \frac{Z^2}{fT} \Delta d. \tag{4.2.21}$$

Thus, higher the displacement of the camera is better the resolution ΔZ is.

To neglect outliers, there is an *order constraint* which states that the features on the left image mush have the same order of the ones in the right image. In formulas, if a point $\overrightarrow{q_1^l} = (x_1^l, y_1^l)$ has a correspondence to $\overrightarrow{q_1^r} = (x_1^r, y_1^r)$, a point $\overrightarrow{q_2^l} = (x_2^l, y_2^l)$ such that $x_2^l > x_1^l$ cannot have a correspondence to $\overrightarrow{q_2^r} = (x_2^r, y_2^r)$ such that $x_2^r < x_1^r$.

After correspondences, the algorithm post-filters the found matches in order to reject false matches. Each feature, in order to be considered as positive, must have a SAD value low enough such that it is similar to the neighbour features. If neighbour features have a small matching metric value whereas the current feature has a high one this is index of a likely false match.

SGBM algorithm relies on mutual information as a superior metric of local correspondences and the enforcement of the *consistency constraints* along directions other than epipolar lines. As result, a much greater robustness against light and false matches is provided. The two main improvements are:

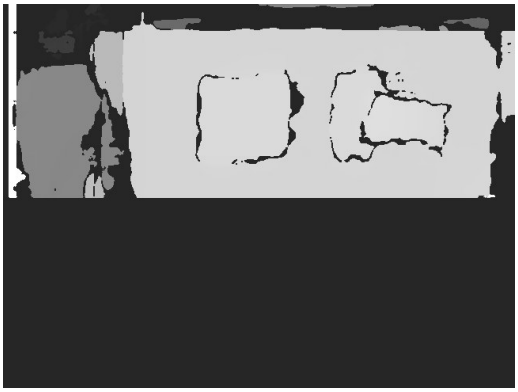
- Use of new Birchfield-Tomasi metric to compare pixels which is insensitive to image sampling thanks to linearly interpolated intensity functions of surrounding pixels[6].

- Use of the *consistency constraints*, this assumes there is some continuity along certain directions on the disparity map. In other words, if the cameras see a diagonal stick the points belonging to the stick are neighbours and have similar disparity values. If this is not true it means that such points are lonely points in the space, and this indicates they are false matches.

The introduction of this superior metric allows the algorithm to use smaller window than BM. For BM, bigger window means more information to rely on for matching but

(a) Top view

(b) Bottom view



(c) Disparity map

Fig. 4.17: Example of the disparity map(c) obtained by the SGBM algorithm given the rectified images (a) and (b). The whiter a point is closer to the camera it is.

it creates problems near the edges of objects because the cameras may see things that are occluded to the other and this results in a high SAD value. Instead, SGBM uses a small window that is less affected by this phenomena and relies on the information of neighbour matches for validation.

SGBM is more sophisticated and more robust but it has a considerable computational cost that may make it not suitable for real time application. In contrast, block matching stereo algorithm is faster. The computation of correspondences with SGBM took up to 2 seconds for the images captured with our stereo rig that have a resolution of 5M pixels. To speed up the process a scaled model of the cameras can be used, thus, the images are scaled by a factor s, and so are the focal length and the center of the image (c_x, c_y). This allows to speed up the process but in contrast the point cloud is less dense and could be smoother near the edges. However, we typically run the stereo match algorithm on a four times downscaled version and the results are good and shown in the next subsection. On such a downscaled image SGBM took about 0.25 seconds and BM took about 0.17

seconds to perform the stereo matching for the example of Figure 4.17. Notice that if the process of calibration is run on a downscaled version of the capture images, the calibration refers to a downscaled version of the camera and the rectification and stereo matching can work directly on the downscaled images without the need to manipulate the reprojection matrix Q.

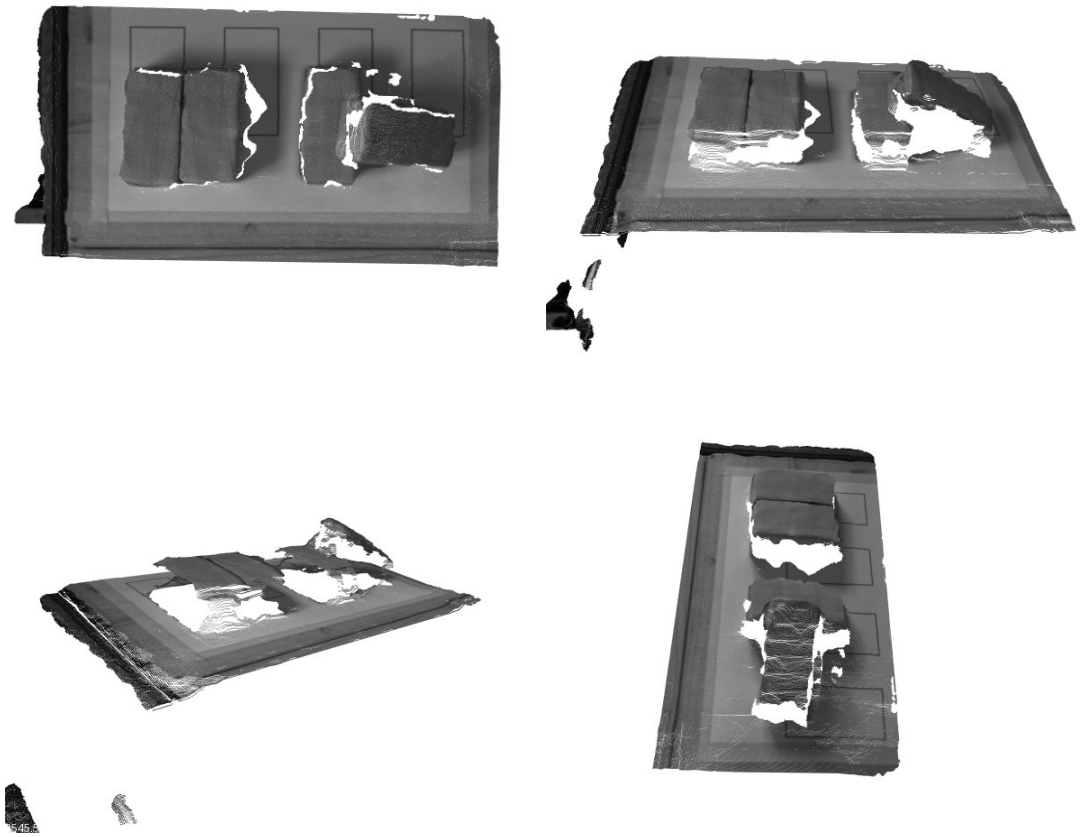


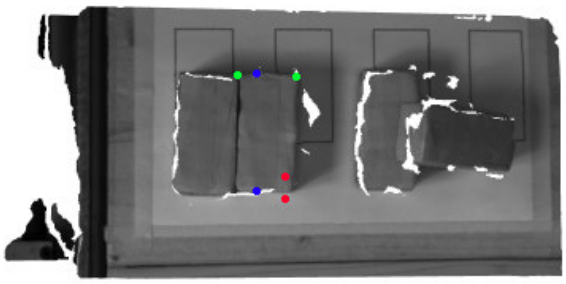
Fig. 4.18: Point cloud obtained by reprojecting the points given the disparity map of Figure 4.17. This has been obtained downscaling four times the input rectified images.

Reprojection

The point cloud can finally be computed by reprojecting all the points of the disparity map accordingly to the Equation 4.2.19. Then the RGB info can be obtained by the first rectified image. The resulting point cloud of Figure 4.17, downscaling the images by a factor 0.25, is shown in Figure 4.18. The depth has been filtered otherwise all the black points of the disparity map, that are the ones without matches, are reprojected to infinity. The example shows four blocks upon a conveyor belt and the dimensions of interests are:

- Distance of the conveyor belt from the cameras: $\approx 1.3m$
- Block dimension: $4 \times 3 \times 8cm$

Tab. 4.1: Measured points coordinates to compare the dimension with the real ones. The euclidian distance is used as metric to compare with the real width and length. For the thickness we used the difference only along the z coordinate.



x	y	z
$0.022304m$	$-0.100604m$	$1.271616m$
$-0.017364m$	$-0.100298m$	$1.276024m$
$width$	$0.0399m$	
$-0.003263m$	$-0.022619m$	$1.275622m$
$-0.001954m$	$-0.102918m$	$1.276024m$
$length$	$0.0803m$	
-	-	$1.266445m$
-	-	$1.295624m$
$thickness$	$0.0292m$	

To verify the correctness of the point cloud the viewer of PCL (`pcl_viewer`) was used enabling the option to display the coordinates of a selected point of the point cloud. In Table 4.1 some relevant points are highlighted and so their coordinates. It is appreciable as the relative distances match with the real dimensions. Notice also that the z coordinate of conveyor belt's points match with the measured distance between it and the cameras. For such a distance from the cameras ($\approx 1.3m$) the depth resolution can be computed using Eq. 4.2.21:

$$\begin{aligned} \Delta Z &= \frac{1.3m^2}{0.016m \cdot 0.13m} = 812.5pixels \\ \Delta Z &= 812.5pixels \cdot 2.2\frac{\mu m}{pixel} = 1.79mm \end{aligned} \tag{4.2.22}$$

where $\Delta d = 1$, and ΔZ has been converted from pixels to millimetres. The resolution is good enough to work with those blocks in the scene.

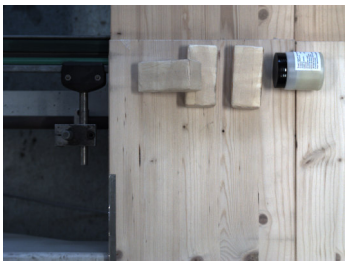
4.2.3. Depth sensor selection

Three different experiments are discussed in order to assert the best depth system for the problem tackled by this thesis. The experiments were designed in order to assert the pro and cons of the two systems, and they are commented in details in the following.

1^{st} **experiment: cluttered objects** This is the most interesting one for the problem we are tackling and is showed in Figure 4.19. The are three small blocks and a flipped over jar. One block is on top of another, and the remaining objects are very close each other. It is appreciable that Kinect can see with a good quality, with small noise, all the objects above the table although the resolution of the depth image is not very high because of the higher field of vision of the Kinect. In fact, Figure 4.19c shows only a small portion of the cloud captured by the Kinect that was approximatively located 1 meter far away the table. Figure 4.19d shows that the edges are poor noisy, thus a good segmentation is expected.

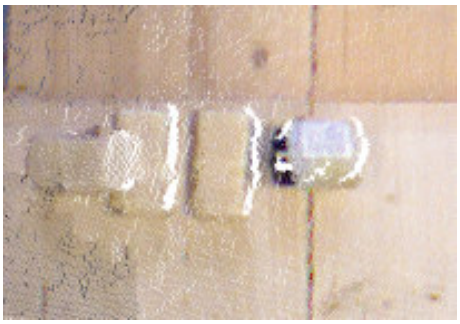
Figures 4.19e and 4.19f show the point cloud captured by the stereo system. The resolution is very high, this because of the small field of vision. The objects are easily identifiable although Figure 4.19f highlights the higher noisy content of the point cloud. Not all the table plane is detected, and for some points the depth is wrongly estimated although they do not differ too much from the real depth. To assert if this point cloud is good enough for our task the segmentation algorithm, explained in the next section, is used. Figures 4.19g shows the detected objects. The segmentation is good except for the block sustained by the other block because of the noise at the edges. By the way this is not a problem as shown in Figure 4.19h where the objects models and the table plane, in gray, are shown. Accordingly to our strategy, explained in the Section 4.4, the perception subsystem is capable of understanding that the purple object is on top of the yellow one; thus the purple one will be grasped before the yellow one and the problem will be correctly solved.

2^{nd} **experiment: transparent plastic bottle** This experiment is intended as proof of the incapability of the Kinect to detect transparent objects such as a plastic bottle. This is



(a) Raw bottom image

(b) Raw top image



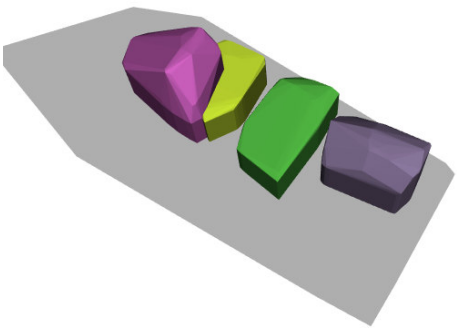
(c) Kinect top view

(d) Kinect lateral view

(e) Stereo top view

(f) Stereo lateral view

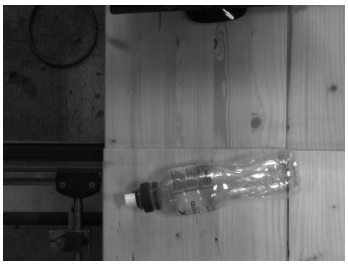
(g) segmentation



(h) objects models

Fig. 4.19: Cluttered scene experiment. In (a) and (b) the bottom and top images are respectively shown. In (c) and (d) the Kinect's point cloud of the objects is shown. In (e) and (f) the point cloud of the stereo system is shown (the point cloud is obtaining using the block matching stereo algorithm). In (g) and (h) the segmentation and the models are shown (see Sections 4.3 and 4.4.1).

(a) Raw bottom image



(b) Raw top image

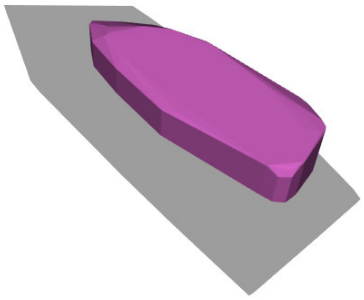
(c) Kinect top view

(d) Kinect lateral view

(e) Stereo top view

(f) Stereo lateral view

(g) segmentation



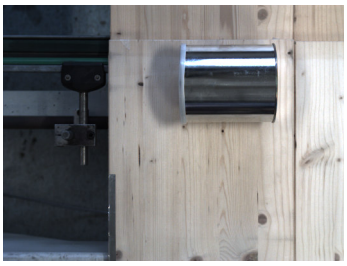
(h) object model

Fig. 4.20: Transparent plastic bottle experiment. In (a) and (b) the bottom and top images are respectively shown. In (c) and (d) the Kinect's point cloud of the objects is shown. In (e) and (f) the point cloud of the stereo system is shown (the point cloud is obtaining using the block matching stereo algorithm). In (g) and (h) the segmentation and the models are shown (see Sections 4.3 and 4.4.1). The gray plane in (h) is the convex hull of the table.

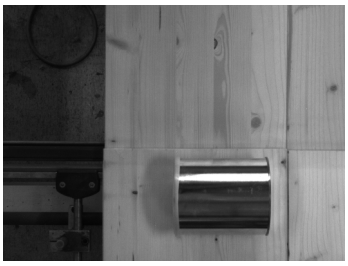
because infra-red light suffers of reflection problem on transparent, shiny, or very matte and absorbing surfaces[4]. Figures 4.20c and 4.20d show that the transparent surface of the bottle is not seen by the Kinect's IR camera but it sees directly the table below. Instead the stereo system can see it because it relies only on RGB information. Figures 4.20e and 4.20f show the point cloud captured by the stereo system. It is appreciable how the image is still noisy and not all the points of the plastic are detected but just few of them. This is because of the light reflection that makes it hard for the matching algorithm to find correspondences. However, the segmentation algorithm is able to segment nicely the bottle (Figures 4.20g and 4.20h).

3^{th} **experiment: flipped over metallic jar** In this experiment it is notable how the Kinect sees nicely the object but the circular edges are missed (Figures 4.21c and 4.21d). Instead, the stereo system is able to see nicely the whole top surface (Figure 4.21e). In the other hand, it gave some problems on the estimation of depth. As appreciable from Figure 4.21f, the top surface has not a circular shape but it is seen as a flat surface. An insight of the problem is that this is due to false matches detected at the part of the jar that reflects the light source. That reflection causes some pixel to be saturated, and it obstacles the matching algorithm. Then, the flat surface is given by the ordinary constraint applied to the pixels that follows the wrong matches. Despite this, the object was nicely segmented as shown in Figures 4.21g and 4.21h.

Conclusions To conclude, Table 4.2 summarizes the pros and cons of both methods for the specific task of this thesis. For the problem we are tackling we state that the Microsoft Kinect is a good depth sensor because its estimation is not particularly noisy with big objects (sizes of some centimetres) as the ones the robot is going to deal with, the limitation of the kind of materials it can detect is not of interest because all the objects we supposed to work with are not transparent, it is a cheap sensor, easy to use and has a wide field of vision compared to our stereo system. The wide field of vision is important because, since the robot is supposed to work with object with dimensions of $15cm$ for instance, the view of the sensor must be wide enough in order to sufficiently cover the working space. To achieve a suitable field of vision with our stereo system, the cameras should be located at a very high distance from the table, this also results in a low resolution of the measured depth. Otherwise other cameras with different characteristics should be

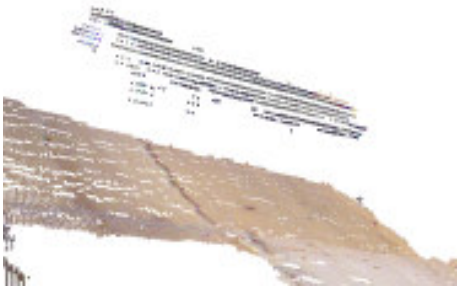


(a) Raw bottom image

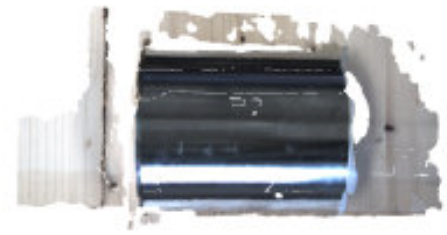


(b) Raw top image

(c) Kinect top view



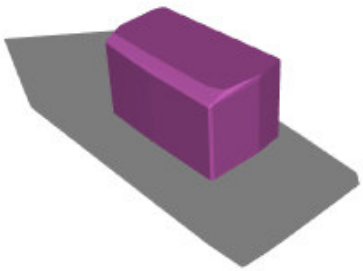
(d) Kinect lateral view



(e) Stereo top view

(f) Stereo lateral view

(g) segmentation



(h) object model

Fig. 4.21: Plastic bottle experiment. In (a) and (b) the bottom and top images are respectively shown. In (c) and (d) the Kinect's point cloud of the objects is shown. In (e) and (f) the point cloud of the stereo system is shown (the point cloud is obtaining using the semi-global block matching stereo algorithm). In (g) and (h) the segmentation and the models are shown (see Sections 4.3 and 4.4.1). The gray plane in (h) is the convex hull of the table.

used. As final comment, from these results the stereo system should be enhanced by images pre-processing to handle the cases of high brightness and saturation (maybe with High Dynamic Range[15]) or by point cloud post-processing to reduce the noise.

Tab. 4.2: Pros and cons of the Microsoft Kinect sensor and the stereo system.

	Pros	Cons
Kinect	easy to use wide field of vision (58.5×46.6 degrees) good precision good frame rate cheap	problem with some materials poor measurements for small/high distances poor measurements for tiny objects
Stereo	all the surfaces can be detected high dense point cloud	problems with saturated images small field of vision (19.8×14.9 degrees each camera) possibly noisy at objects boundaries computationally expensive, low frame rate expensive

However we highlight that this comparison focuses on the task of this thesis, and this does not state that Kinect is better than a stereo system. It depends on the specific task. A stereo system can be easily designed to get a good point cloud of tiny objects such as bolts whereas the Kinect would give a point cloud that make it difficult to distinguish the bolt from the table.

4.3. Object Segmentation

One of the most interesting applications of computer vision is the location of objects in the scene. There are two main categories of methods that deal with locating objects: *object detection* and *object segmentation*. The former is the branch of computer vision that focuses on locating certain classes of objects in the scene. Hence, a prior knowledge of the searched object is required. In a similar but different fashion, the latter is a group of methods to segment the objects, that is to assign a certain part of the image to each object. Object segmentation is a particular case of *image segmentation* which is the problem of localizing regions of an image relative to content.

Generally, object detection algorithms rely on knowledge of the objects that can be given by a 3D or 2D model of the objects (e.g. a CAD model), or relevant features that can distinguish it and that can be learnt using machine learning methods (e.g. face

detection). To make the robotics system be more general as possible we decide to work with no prior knowledge of the objects. Then, the object can be located by segmenting them.

This task is quite hard to perform on a RGB image but easier to do on point cloud since the point representation has important information regarding the geometry of the objects.

It remains the problem of defining what an object is; we consider an object any sufficiently big cluster of points on the table. We do not want to segment the entire image, otherwise the table would be segmented as an object, and the floor as well. Thus, the algorithm has first to detect the table, and so the objects that stand on top of it, and then segmenting them.

Thus, this stage of the perception pipeline is composed of 3 steps:

1. Point cloud filtering: point clouds are noisy at the edges, thus to reduce the noise the statistical outlier removal algorithm [49] is used.

2. Tabletop objects detection: at this stage we detect the table and all the objects upon it.

3. Objects segmentation: having a point cloud of all the objects we segment it in order to obtain a cluster of points per object.

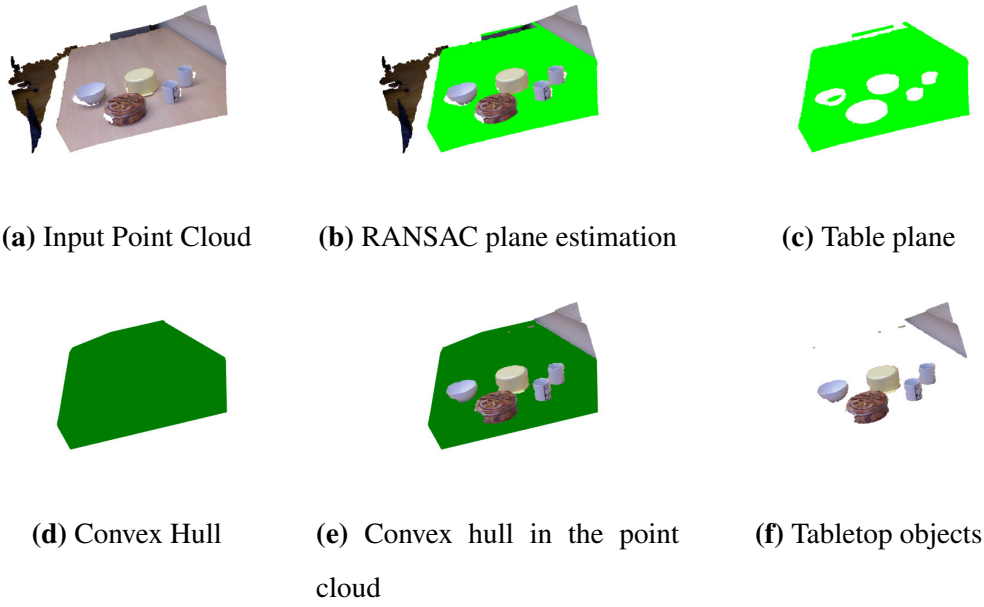
4.3.1. Tabletop Object Detection

The strategy for the tabletop object detection phase is composed of 3 different steps:

1. **Table plane estimation**: the points of the table are detected estimating a plane in the point cloud using the RANSAC algorithm[19], all the points which belong to such a plane are the points of the table.

2. **2D Convex Hull of the table**: having the points of the table a 2D convex hull is computed in order to get a 2D volume containing those points. A 2D convex hull is the smallest convex area that contains all the points of a given set.

3. **Polygonal prism projection**: all the points are projected on the table plane previously estimated and all the points whose projections belong to the 2D convex hull



(a) Input Point Cloud **(b)** RANSAC plane estimation **(c)** Table plane

(d) Convex Hull **(e)** Convex hull in the point cloud **(f)** Tabletop objects

Fig. 4.22: Object Segmentation: Given the point cloud (a), the estimated table's plane is obtained (b and c), its convex hull is extracted (d and e), and the tabletop objects are obtained by a polygonal prism projection (f).

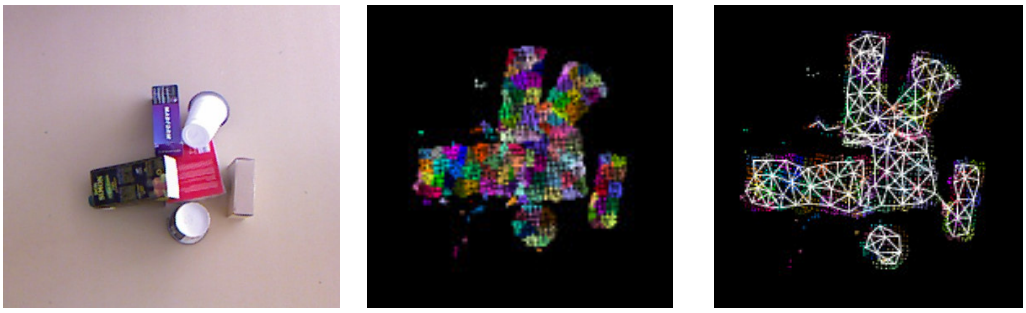
are considered to be points of tabletop objects. The points that do not belong to it are points of non-tabletop objects.

The steps of this tabletop object detection algorithm are described in Figure 4.22 for the point cloud[8] in Figure 4.22a.

4.3.2. Segmentation

The PCL library [48] provides several algorithms to perform image segmentation. Many of them are based on clustering the points by some constraints, they may be both geometrical and color constraints, in such a way that only the points that are close and share similar characteristic are considered as points of the same object. The *Locally Convex Connected Patches* (LCCP) [51] algorithm proved to be very effective thanks to the way it handle those constraints.

[8] Point cloud taken from the Object Segmentation Database (OSD) http://users.acin.tuwien.ac.at/arichtsfeld/?site=4



(a) Input scene **(b)** Supervoxels **(c)** Adjacency graph

Fig. 4.23: Example of supervoxels for the table top objects.

Supervoxel

For their segmentation the supervoxel concept is used. A supervoxel is a group of voxels that share similar characteristics, for instance similar normals.

In this work the supervoxels are computed with the *Voxel Cloud Connectivy Segmentation* (VCCS) algorithm [45], which is able to be used in online applications. An example of the obtained supervoxels is shown in Figure 4.23.

The algorithm works in 3 main steps as a K-nearest variant:

- Voxelizing the point cloud to choose the seeds: it partitions the 3D space in voxels ensuring an evenly distribution of seeds.
- Clustering together all adjacent voxels by means of a distance metric that accounts spatial distance, color and normals.
- Creating an adjacency graph for the supervoxel: the adjacency map connects adjacency supervoxels and is iteratively used to generate and refine the supervoxels.

Local Convex Connected Patches Segmentation

Once the supervoxels of the tabletop objects are computed, they can be clustered in order to segment the objects. Papon et al. [51] also proposed a segmentation algorithm based on their supervoxel technique, called *Local Convex Connected Patches Segmentation* (LCCP). This algorithm allows to segment objects by clustering together adjacent convex supervoxels. In Figure 4.24 the algorithm's pipeline is briefly described. The algorithm is quite simple but very effective for segmentation of objects that have convex shapes.

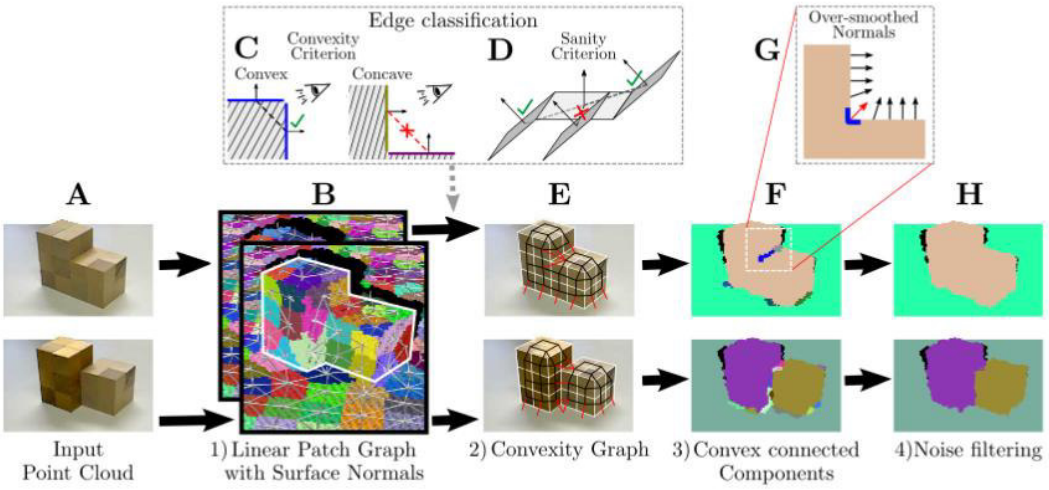


Fig. 4.24: LCCP algorithm's structure. Reproduced from [51]

Fig. 4.25: Example of segmentation results.

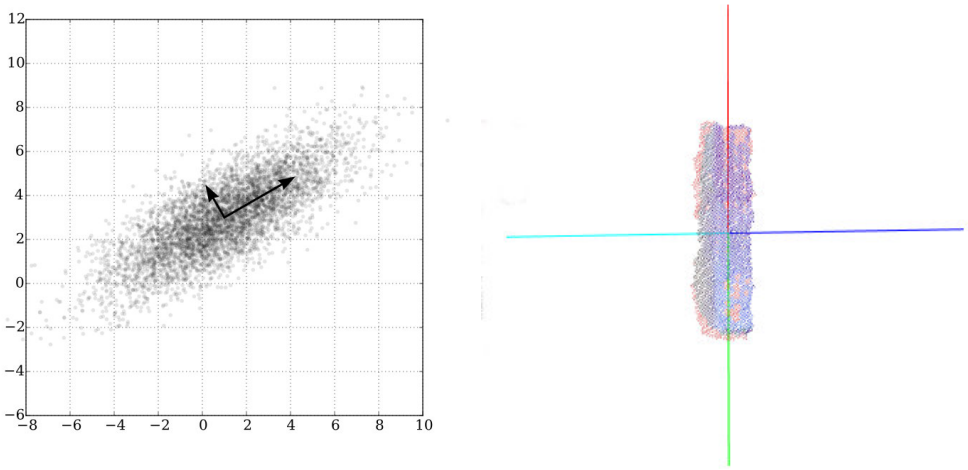
It clusters all the adjacent convex supervoxels (patches) using 2 criterion:

- Convexity criterion: to consider two adjacent patches convex, both must have a connection to a patch which is convex with respect both patches
- Sanity Criterion: check if the adjacent patches which can be considered as convex present geometric discontinuities (see point D of Figure 4.24), in this case they are not considered as valid to form a cluster.

Then, due to the smoothed normals that could appear in some edges of the objects (point G Figure 4.24), the algorithm merges the clusters that are composed of few supervoxels to the biggest adjacent cluster.

By tuning properly the parameters of the segmentation algorithm the objects can be correctly segmented obtaining for each one a point cloud. Two examples of the segmentation algorithm for a cluttered scene are depicted in Figure 4.25.

The algorithm was set in manner to segment accounting only geometric properties, and not the color properties.



(a) **(b)**

Fig. 4.26: Principal Components Analysis: Figure 4.26a shows PCA for a standard 2D set of observations, Figure 4.26b shows the result of the PCA for a rectangular segmented object. The green, red lines refers to different ways of the first principal direction, while blue and cyan lines refers to different ways of the second principal direction. The third one is the cross product of the first two principal directions because of the orthogonality property of principal components.

Note that this segmentation algorithm forces us to distort the initial hypothesis of absence of prior knowledge about the objects. Hence, there is prior knowledge about the convex shape of the objects.

4.4. State generation

In this section the computation of the symbolic predicates introduced in Section 3.4 is described in detail. Before to move in the detail description a brief introduction of some required general concepts is proposed.

4.4.1. Preliminary concepts

Principal Direction The principal direction of an object is its principal axis which is defined as any of three mutually perpendicular axes about which the moment of inertia is maximum. For instance, for a rectangular object its principal direction is the axis aligned with its longest dimension.

To obtain the principal axis the principal component analysis (PCA) [25] technique is

used. This technique is a common statistical procedure that uses orthogonal transformation to convert a set of observations of possibly correlated variables into a set of values of linearly uncorrelated variables, which are called principal components. The transformation is defined in a manner that the first component has the largest variance, the second has the second largest variance and so on. An example of the principal components for a 2D data set is depicted in Figure 4.26a[9].

A generic point cloud can be seen as a set of observations, where the variables are the 3D coordinates, and the PCA can be directly applied to the object's point cloud to retrieve its principal components. In this work we refers to principal components as principal directions, where with the term *first principal direction* we refer to the first principal component, or equivalently to the principal axis. In Figure 4.26b the first two principal directions of a generic object are illustrated. Note that for each principal direction we can obtain two vectors (one per sense).

Projection onto a plane We will see later that the concept of the projections of a point onto a plane will be useful. Considering a point $\overrightarrow{p} = (x_p, y_p, z_p)$ and a plane Π defined by the following equation

$$ax + by + cz + d = 0$$

the projection p_Π of point $\overrightarrow{p}$ onto the plane Π is given by the following set of operations:

1. Calculate the origin point $\overrightarrow{\Pi_O} = (x_O, y_O, z_O)$ of the plane, which can be calculated by arbitrary x_O and y_O coordinates as

   $$z_O = \frac{-1}{c}(ax_O + by_O + d),$$

   then calculate the coordinates of $\overrightarrow{p}$ with respect the point $\overrightarrow{\Pi_O}$:

   $$\overrightarrow{\Pi_O p} = \overrightarrow{p} - \overrightarrow{\Pi_O}.$$

2. Then calculate the projection of $\overrightarrow{\Pi_O p}$ onto the plane normal $\overrightarrow{n} = (a, b, c)$

   $$\lambda_p = \overrightarrow{n} \cdot \overrightarrow{\Pi_O p}.$$

3. Translate point $\overrightarrow{p}$ by λ_p along the normal of the plane $\overrightarrow{n}$

   $$\overrightarrow{p_\Pi} = \overrightarrow{p} - \lambda_p \overrightarrow{n}.$$

   The minus sign is due to the fact that the normal is pointing upwards.

[9] Image taken from https://en.wikipedia.org/wiki/Principal_component_analysis

Rotation Matrices Rotation matrices express a rotation between two reference frames. Given two frames $\{A\}$ and $\{B\}$, and the rotation matrix ${}^{A}_{B}R$ that defines the rotation of $\{B\}$ relative to $\{A\}$, then a point ${}^{A}\overrightarrow{p}$ with respect frame $\{A\}$ is given by ${}^{A}\overrightarrow{p} = {}^{A}_{B}R\,{}^{B}\overrightarrow{p}$, where ${}^{B}\overrightarrow{p}$ is the same point relative to frame $\{B\}$.

Having a frame $\{B\}$ defined by axis ${}^{A}\overrightarrow{X_B}$, ${}^{A}\overrightarrow{Y_B}$ and ${}^{A}\overrightarrow{Z_B}$, where ${}^{A}\overrightarrow{Y_B}$ is the unitary vector, relative to frame $\{A\}$, with same orientation of the y axis of frame $\{B\}$, the rotation matrix between $\{A\}$ and $\{B\}$ is defined as

$$ {}^{A}_{B}R = \begin{bmatrix} {}^{A}\overrightarrow{X_B} \\ {}^{A}\overrightarrow{Y_B} \\ {}^{A}\overrightarrow{Z_B} \end{bmatrix} $$

To transform any object, such as the gripper mesh model, to a pose defined by frame $\{B\}$ then the following homogeneous transform is applied:

$$ H = \begin{bmatrix} {}^{B}_{A}R & {}^{A}B_O \\ \overrightarrow{0} & 1 \end{bmatrix} $$

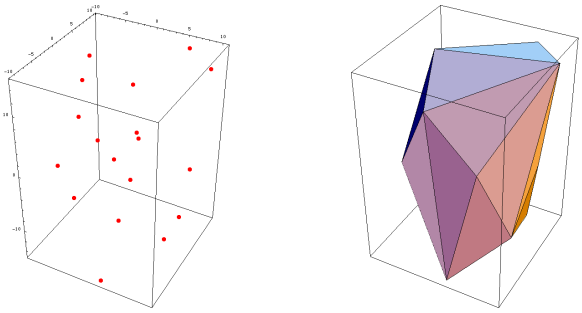
where ${}^{B}_{A}R = {}^{A}_{B}R^{\top}$ and ${}^{A}B_O$ is the origin of frame $\{B\}$ relative to $\{A\}$. In this way, having some axes that define our new reference frame, we can transform the gripper model in such a way its closing point is in the origin of the new frame and its orientation is aligned to the one of the new reference frame.

Bounding Box A bounding box is the smallest cubic volume that completely contains an object[10]. An axis-aligned bounding box (AABB) is a bounding box aligned with the axis of the coordinate system, while an oriented bounding box (OBB) is a bounding box oriented with the object's frame. To compute the OBB the object is transformed from its frame to the world frame and the dimensions of the bounding box are obtained by computing the maximum and minimum coordinates of the transformed object. In this way it is possible to have an approximation of the length, width and height of an object.

Convex Hull A convex hull of a point cloud P is the smallest 3D convex set that contains P. In Figure 4.27[11] an example of the convex hull for a point cloud is shown. The vertices are first detected and then connected among them by means of triangles. In this way a

[10] https://en.wikipedia.org/wiki/Bounding_volume

[11] Images obtained from http://xlr8r.info/mPower/gallery.html



(a) Pointcloud **(b)** Convex hull

Fig. 4.27: Convex hull example

triangle mesh is associated to the convex hull. This can be directly computed using the PCL library[48].

Collision Detection To understand if an object blocks a certain action, such as the pushing along a certain direction, we have to check if along the desired trajectory the pushed object would collide with the others. The collision detection is therefore a crucial step to generate the states. There exist different techniques to assert if two objects are colliding and all of them need a representation of the object, which could be a basic shape or a more complex as an octree.

The mesh shape has been thought to use since it can be directly obtained from a convex hull.

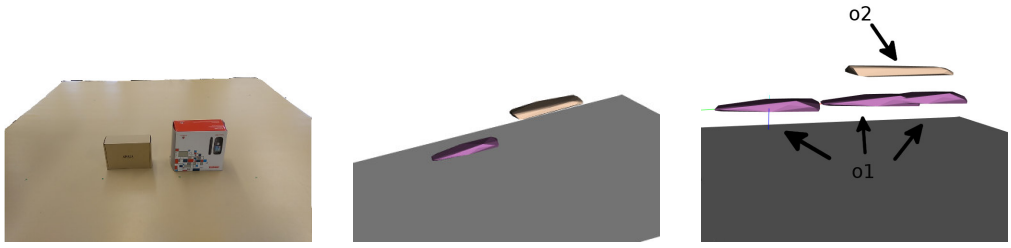
Given two objects A and B and their configurations $\mathbf{q_A}$ and $\mathbf{q_B}$, the collision test returns a boolean value about whether two objects collide or not [44]. Two objects collide if

$$A(\mathbf{q_A}) \cap \mathbf{B}(\mathbf{q_B}) \neq \mathbf{0}$$

The collision detection will be used to understand if in a given pose the object A would collide with the other objects in the scene.

In order to relax the collision detection the majority of collision libraries, before to use complex algorithm to detect collision between two shapes, they first check if the bounding volumes (e.g. AABB) of the objects intersect, if they don't the objects surely don't collide. If their bounding volumes intersect the objects might collide and more complex algorithm are used to assert the collision.

For the collision detection the FCL[44] library is used. Objects are represented as triangular meshes directly formed by the convex hull.

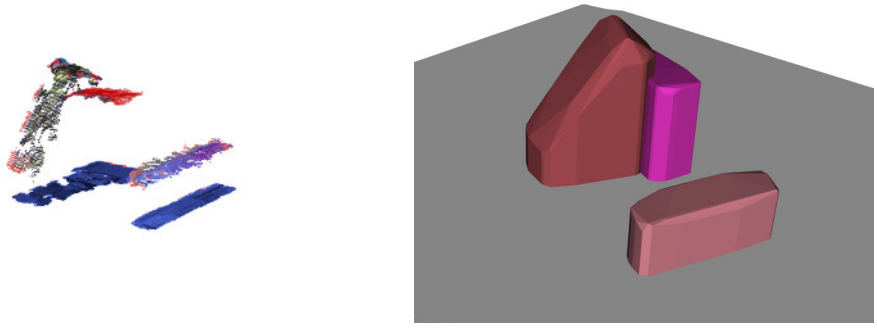


(a) (b) (c)

Fig. 4.28: Convex hulls and collision detection using the segmented objects retrieved by the LCCP segmentation algorithm. The gray surface represents the plane's 2D convex hull. Figure 4.28a shows the scene captured by an external camera, where as the depth sensor sees the scene from above. Figure 4.28b shows the convex hull computed on the segmented object retrieved by the LCCP segmentation algorithm. It is possible appreciating that we miss the information about the hidden part of the object. In Figure 4.28c a collision detection example is depicted. The convex hull of object `o1` is translated along a direction and no collision is detected since the two convex hulls do not intersect.

Objects Modeling The depth sensor can mainly see one face (usually the top) of the objects and therefore we cannot apply directly the convex hull algorithm to the detected surfaces. If we applied the convex hull on an object's observed surface, we would have likely the situation depicted in Figure 4.28c, in which the collision detections would not detect any collision when it should. This is because we are missing the surfaces that cannot be seen from the Kinect's point of view.

From the captured point cloud also the table plane is known because estimated at the segmentation stage, so the information we have are: the table plane model and the segmented objects (mainly the top surfaces). If an human would be in the same pose of the camera, looking at the table, he would imagine that the objects are not floating surfaces, and he/she would deduce the objects shape from the shape of the top surface. The sides of the objects can be deduced by projecting the top surface's edges to the plane and then filling the missing object's sides with points. To do that we have to detect the top surface's edges. A easier method is directly projecting all the points of the surfaces onto the table plane and then apply the convex hull algorithm to the resulting point cloud given by the sum of the top surface and its projection. In this way the missing sides are indirectly retrieved by the convex hull. An example of this method is depicted in Figure 4.29.



(a) Surfaces projection **(b)** Resulting convex hull

Fig. 4.29: Convex hull of the objects using their projections onto the table plane: Figure (a) shows the detected surfaces and their projected surfaces onto the table plane (in blue), (b) shows the retrieved convex hull using also the projected points. The original scene, although difficult to understand in image (a), is composed of a violet box, a red one and a black juice box supported by the red box.

4.4.2. Predicate: `block_grasp`

The `(block_grasp o1 o0)` predicate refers to the fact that object `o1` prevents `o0` from being grasped. The computation of this predicate is straightforward: the mesh model of the opened gripper is transformed to the grasping pose of object `o0`, and checked if it collides with the other objects. If no object collides with the gripper, `o0` is considered to be graspable. Notice that in order to have a certain collision-free range we suppose that a grasping pose is considered to collide if the minimum distance between the gripper and the surrounding objects is less than $2cm$. In other words, `o0` is graspable if all the objects are further than $2cm$ from the gripper when transformed to the grasping pose of object `o0`. In Figure 4.30 such a procedure is shown and in Algorithm 3 the pseudo-algorithm is described in detail.

Notice that this method requires to check for collision between the gripper and objects that might be very far from the interested object, i.e. there is no need to compute the collision detection. Despite this, as explained in Section 4.4.1, the majority of collision detection algorithms first check if the bounding boxes of the objects intersect. This is a computationally cheap operation, and only if they intersect the computationally expensive algorithm are used to check for collision. This makes the Algorithm 3 efficient and computationally not expensive. It has been observed that, in average, to compute this

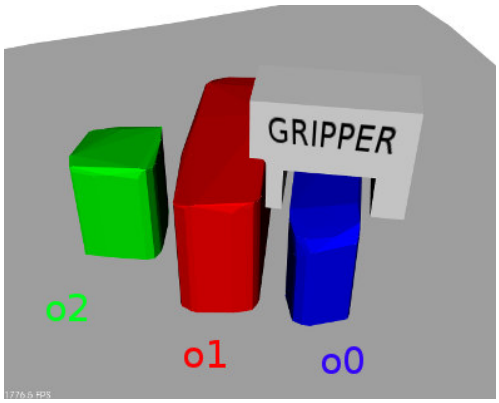


Fig. 4.30: Visualization of the computation of `block_grasp` predicate for object `o0`. The opened gripped model is transformed to the grasping pose for object `o0` and it is tested if the gripper mesh model collides with the other objects. In this case it collides with `o1`.

Algorithm 3 Computation of `block_grasp` predicates.

Inputs: Set of objects O (convex hull retrieved with the projection onto the table plane) and the set of grasping poses G_{poses}.

Outputs: The `block_grasp` predicates.

function COMPUTEBLOCKGRASPPREDICATES(O,G_{poses})
  BLOCK_GRASP_PREDICATES $= \varnothing$
  for all $A \in O$ **do**
    $gripperMeshTransformed \leftarrow$ TRASNFORMGRIPPERMODEL($G_{poses}[A]$)
    for all $B \in O$ **do**
      if $A \neq B$ **then**
        $collision \leftarrow$ ISTHERECOLLISION($gripperMeshTransformed$, B)
        if $collision \vee$ DISTANCE($gripperMeshTransformed$, B) $< 2cm$ **then**
          BLOCK_GRASP_PREDICATES $\bigcup$ `(block_grasp B A)`
        end if
      end if
    end for
  end for
return BLOCK_GRASP_PREDICATES
end function

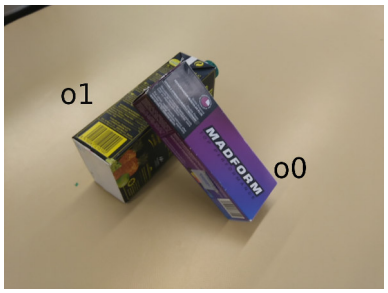
predicate the time is about 10 milliseconds per object, i.e. to check for collision between the opened gripper and all the other objects.

This predicate is evaluated for every possible combinations of grasping pose and objects, therefore the complexity to generate this state is $\mathcal{O}(n^2)$,where n is the number of objects.

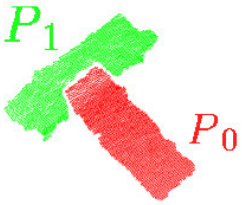
4.4.3. Predicate: `on`

The `(on o0 o1)` predicate, when true, means that object `o0` is on top of object `o1`. With the convex hull of the objects is easy to understand if two objects are one on top of the other one by checking for collision, but in this way we do not know what is above and what is below. To do this, their surface projections onto the table plane are used. The research group of Artificial Intelligence and Robotics Laboratory of Istanbul Technical University, published some interesting researches suitable to the aim of this thesis. In [17, 43] the authors proposed some approaches to enhance 3D recognition and segmentation results to create and maintain a consistent world model involving attributes of the objects and spatial relations among them. Their researched focused on modelling the world for manipulation planning tasks. They do not consider scene like the one of this thesis but simpler ones such as a pile of cubes above each other. What can be directly used from their work is the computation of the `on` predicate. The `on` relation for a pair of objects is determined by checking whether their projections onto the table plane overlap. This predicate was not a relevant part of their work and they did not provide too much information about its computation. Therefore our implementation for the `on` predicate is based on their idea with some modifications.

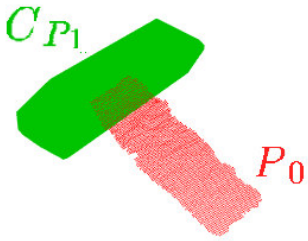
The key idea is based on the fact that an object, which stands on top of another, occludes some parts of the object below. In the other side, the one below does not occlude any part of the top object. Let's consider the scene in Figure 4.31a, the object `o0` occludes a portion of object `o1`. The projections onto the table plane of `o0` and `o1` form respectively the sets of point P_0 and P_1, which are the red and green ones in Figure 4.31b. The set P_0 intersects the convex hull C_{P_1} of P_1, whereas the P_1 does not intersect the convex hull C_{P_0} of P_0 (Figures 4.31c and 4.31d). In this issue the intersection does not refer to the mathematical intersection but to the set resulting from the points, of a given set, that are inside a given convex hull.



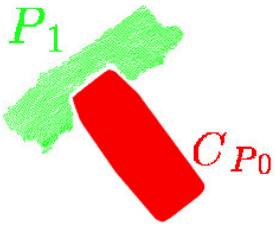
(a) Scene



(b) P_1 & P_0



(c) C_{P_1} & P_0



(d) P_1 & C_{P_0}

Fig. 4.31: Visualization of the computation of the `on` predicate. Figure 4.31a shows a scene where `o0` is on top of `o1`, Figure 4.31b shows the projections of the objects onto the table plane whereas Figures 4.31c and 4.31d represent the two fold strategy to compute the `on` predicate.

Although this method works fine to compute the `on` predicate it has the limitation that its scope is only for objects with a simple shape. Moreover, if the top object occludes too much the object below, the segmentation may see the object below as two different objects and the top object may be detected as being not on top of anything.

It is important to take into account also that actually the edges of the occluded parts of the below object, once projected, could be at a similar position, of some projected edges of the top object. This could be dangerous for the computation of this predicate. Focusing the attention on Figures 4.31c and 4.31d it can be appreciated that the intersection $C_{P_1} \cap P_0$ includes several points. On the other hand, in case the edges projections relative to the occluded and occluding part have similar coordinates, the intersection $C_{P_0} \cap P_1$ would include just few points. Therefore a threshold is added as bound for the cardinality of the intersections. Finally, the `(on o0 o1)` predicate is update accordingly to the following

formula:

$$\texttt{(on o0 o1)} = \begin{cases} True,\ |C_{P_0} \cap P_1| < th \ \wedge \ |C_{P_1} \cap P_0| > th \\ False,\ otherwise \end{cases} \tag{4.4.1}$$

where the threshold th is 100 and is empirically determined.

The formula 4.4.1 is then evaluated for every possible combinations of objects, therefore the complexity to generate this state is $\mathcal{O}(n^2)$,where n is the number of objects. Despite this, its computation is very fast. The example in Figure 4.31 was evaluated just 2 times since there are only 2 objects and it took 3 milliseconds, that is $\approx 1.5 \frac{ms}{pair\ of\ object}$. For instance, for a complex scene with 10 objects the total time devoted to compute this predicate would be approximatively $10 \cdot 9 \cdot 1.5 \approx 135ms$.

4.4.4. Predicate: `block_push`

The `(block_push o1 o0 dir`$_i$`)` predicate, if true, means that object `o1` prevents object `o0` from being moved along its i-th direction because a collision would occur.

Recall that the pushing action pushes the object along a pushing direction up to a graspable pose. Let's evaluate the predicate for object `o0` along its pushing direction `dir1`. Its computation is based on a two-fold strategy :

- `o0` is translated by ρ ($0.03m$ in our experiments, although it can be variable with some heuristics) along `dir1` and checked for collisions with the other objects. If `o0` collides with `o1` the literal `(block_push o1 o0 dir1)` is true. This process is iterated until `o0` is graspable. At this stage, to check for graspability, the surrounding objects are tested for collision with the gripper in the grasping pose of the translated object. Notice that, in order to speed up this graspability checking, the objects that prevent `o0` from being pushed are not considered because they need to be moved anyway to push `o0` along `dir1`.

- The robot needs free space to position the end-effector before pushing `o0`, such a pose is called *pushing pose*. If in the pushing pose the closed gripper model collides with `o1` then `(block_push o1 o0 dir1)` is set to true. This is done only for the pushing pose and not for all the pushing path in order not to be too conservative.

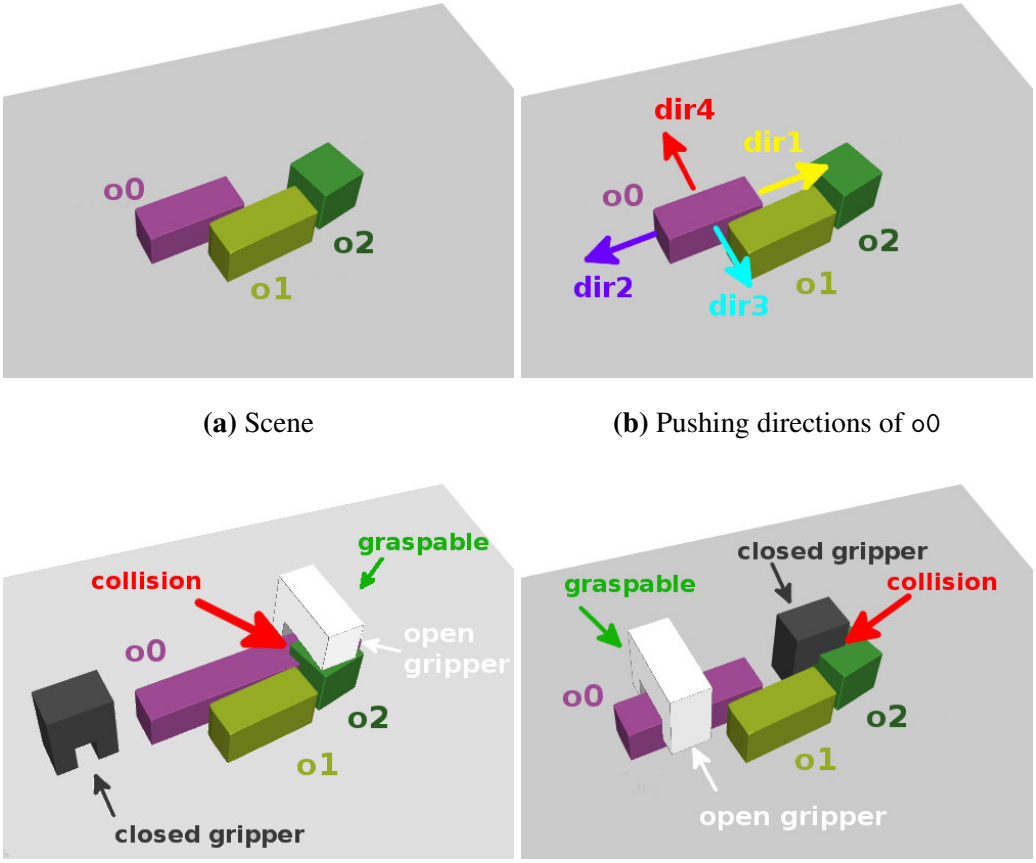
A queue is used to keep track of the objects that are already known to collide with the translated object or the gripper in the pushing pose. In this way, we skip the evaluation, i.e. the collision checking, of predicates that are already true.

Note that also the gripper during the pushing action will move, so the collision checking should be done exactly as done for the object. We decided to neglect this and to check only for the pushing pose in order to relax the planner and not to make it too much conservative. This means that during the pushing action the robot might actually move more than one object. Despite this relaxing strategy the algorithm showed to work fine and cases in which the gripper moved more than one object were rare and correctly handled by replanning.

An example of the computation of the `block_push` predicate is shown in Figure 4.32. In Figure 4.32a a scene with 3 objects is shown, and in Figure 4.32b the pushing directions of `o0` are highlighted. In Figure 4.32c and 4.32d the computation of the predicates for pushing `o0` along `dir1` and `dir2` are graphically shown. We first check if `o0` can be pushed along its first pushing direction. In Figure 4.32c the closed gripper model is collision-free meaning that there is free space to collocate the end-effector at the *pushing pose*. Object `o0` needs to be pushed 0.21 meters along `dir1` to be graspable but it collides with `o2`, so `(block_push o2 o0 dir1)` is set to true. It can be thought that it needs to be moved more than 0.21 meters but `o2` does not intervene in the graspability checking since `o2` has first to be moved away in order to move `o0` along that direction. In Figure 4.32d `o0` is collision-free along `dir2` and it has to be pushed 0.06 meters to be graspable, but `o2` collides with the gripper in the *pushing pose*, so `(block_push o2 o0 dir2)` is set to true. Therefore, the robot will have first to interact with `o2` to move `o0` along either `dir1` or `dir2`. Considering also the directions `dir3` and `dir4`, the predicates `(block_push o1 o0 dir3)` and `(block_push o1 o0 dir4)` are true as well.

Applying the same procedure to `o1`, we obtain that the predicates `(block_push o0 o1 dir3)`, `(block_push o0 o1 dir4)`, `(block_push o2 o1 dir1)` and `(block_push o2 o1 dir2)` are true. This means that if `o1` is not graspable the robot has to push away the other objects before interacting with `o1` since they impede `o1` to be grasped or pushed in any direction.

Regarding `o2`, only the pushing directions `dir3` and `dir4` are impeded. In fact, for `o2` the `block_push` predicates set to true are `(block_push o0 o2 dir3)`, `(block_push`



(a) Scene **(b)** Pushing directions of o0

(c) (block_push o2 o0 dir1) **(d)** (block_push o2 o0 dir2) and (block_push o1 o0 dir2)

Fig. 4.32: Computation of block_push predicate for the first two pushing directions dir1 and dir2 of o0. In (c) and (d) the closed gripper is transformed to the pushing pose of (push o0 dir1) and (push o0 dir2) actions respectively. The white mesh model represents the open gripper model transformed to a collision free grasping pose after having pushed o0 for a sufficient length.

o0 o2 dir4), (block_push o1 o2 dir3) and (block_push o1 o2 dir4).

For this example, having a goal to clear the table, the planner finds as solution this set of actions: (push o2 dir1),(grasp o2),(push o0 dir1), (grasp o1), (grasp o0).

For completeness, Figure 4.33 shows an example of the computation of the predicate for a cluttered scene with 8 objects. From that we can notice:

- o7 is graspable from its pose so there is no need to push it.
- o2 must be pushed away along its dir1 but o3 impedes the action because it collides

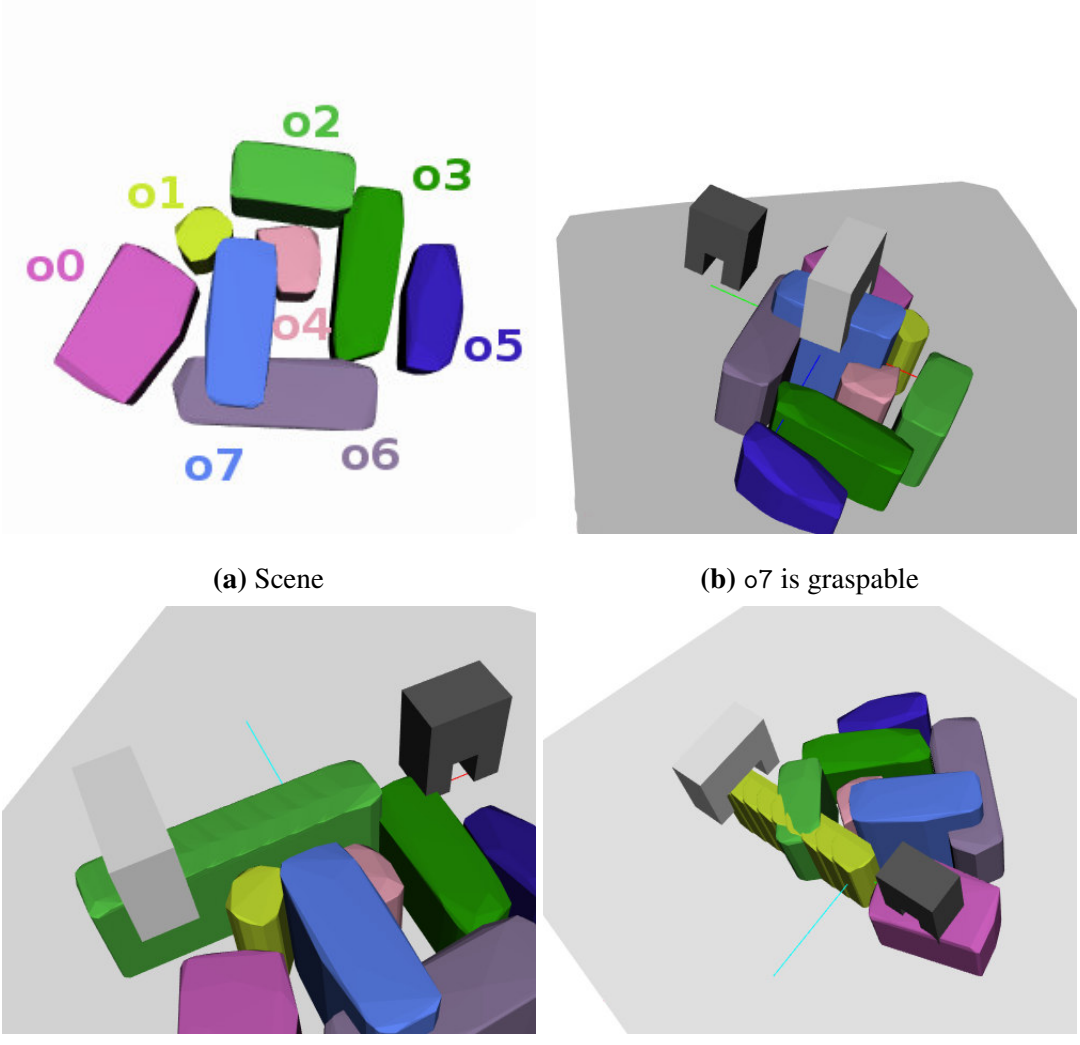
with the gripper in the pushing pose.

- `o1` have to be pushed away along its `dir1` but `o2` collides with `o1` and the gripper collides with `o0` in the pushing pose.

The computation of this predicate can be appreciated in detail in Algorithm 4.

The computation of this algorithm is the most computational expensive of all the planner since it involves many collision detections. In fact the time required to compute this predicate for the example in Figure 4.32 was $\approx 0.221 seconds$, that is not negligible considering the simplicity of the problem.

For the more complex example of Figure 4.33, the system took about $1.9 seconds$,



(a) Scene **(b)** `o7` is graspable

(c) `(block_push o3 o2 dir1)` **(d)** `(block_push o2 o1 dir1)`, `(block_push o0 o1 dir1)`

Fig. 4.33: Example of the computation of `block_push` predicate for a cluttered scene with 8 objects.

0.047*seconds* and 0.027*seconds* to compute the `block_push`, `block_grasp` and `on` predicates respectively. Despite this, the problem is very complex and, after computing the predicates, the planning stage is very fast and it justifies their computational costs.

Algorithm 4 Computation of `block_push` predicates.

Inputs: set of objects O (convex hull retrieved with the projection onto the table plane), set of the pushing directions P_d of all the objects, set of the pushing pose P_{poses} of all the objects, ρ displacement to test the object for collisions when pushed.

Outputs: `block_push` predicates

function COMPUTEBLOCKPUSHPREDICATES(O, P_d, P_{poses}, ρ)

BLOCK_PUSH_PREDICATES $= \varnothing$

for all $A \in O$ **do**

$BO = \varnothing$ ▷ set of objects known to collide either with A or the gripper

for all $dir \in P_d(A)$ **do**

while $\neg$ISGRASPABLE(A) **do** ▷ graspable if no object is closer than 2cm to the gripper

$A \leftarrow$ TRANSLATEOBJECT(A, ρ, dir)

for all $B \in O$ **do**

if $A \neq B \ \wedge \ B \ni BO$ **then**

if ISTHERECOLLISION(A,B) **then**

BLOCK_PUSH_PREDICATES $\bigcup$ `(block_push B A` dir`)`

$BO \bigcup B$

end if

end if

end for

end while

$closedGripperMesh \leftarrow$ TRASNFORMCLOSEDGRIPPERMODEL($P_{poses}(A, dir)$)

for all $B \in O$ **do**

if $A \neq B \ \wedge \ B \ni BO$ **then**

if ISTHERECOLLISION($closedGripperMesh$,B) **then**

BLOCK_PUSH_PREDICATES $\bigcup$ `(block_push B A` dir`)`

$BO \bigcup B$

end if

end if

end for

end for

end for

return BLOCK_PUSH_PREDICATES

end function

5. Execution Subsystem

In this chapter how the actions are executed is described in details. This chapter also helps to have a better understanding of the actions with respect to the action model provided in Chapter 3.

5.1. Pushing

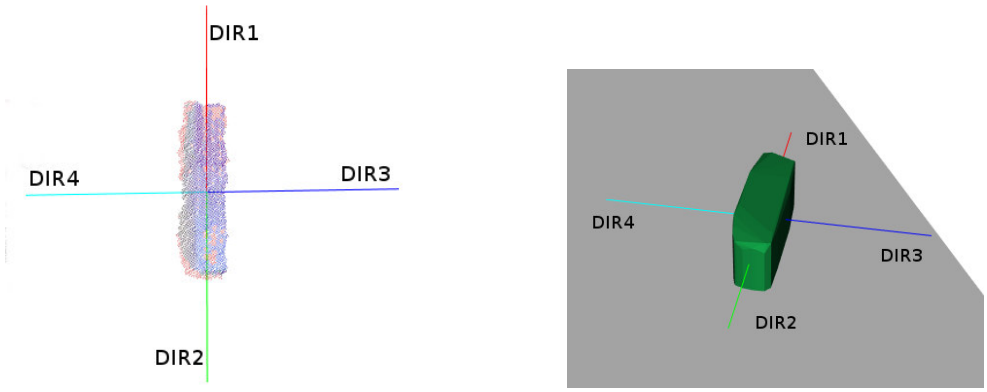
Pushing is a difficult action to execute when the goal is to move one object along a path. The majority of pushing actions in object manipulation have the aim to interact with the objects in order to move them and validate the segmentation [30] [28] [29] , without taking care about the final position of the objects or about eventual collisions. Hermans et al. [22] presented a novel algorithm for object singulation through pushing actions, here the pushing actions have the aim to separate objects in cluttered scenes but they are not interested in singulate tidily the objects but just to find a feasible interaction to singulate them regardless possible collisions.

We considered to work with objects with simple shapes, comparable to parallelepipeds or cylinders. A human would push such objects mainly accordingly its principal axis and the one orthogonal to its (i.e. the first 2 principal components of Figure 4.26b), he/she also could push it along the diagonal thanks to the several degrees of freedom of the hands. Inspired by this consideration, we decided to consider its first two principal directions as possible directions to push an object. In particular, there are two senses for each direction, so in total we have 4 possible pushing directions per object, as depicted in Figure 5.1.

Another thing to take into account is that the principal directions are not always parallel to the table plane. An object which stands on top of a table will be obviously pushed along a direction parallel to the table. For this aim the first two principal components are initially computed onto the segmented surface and then they are projected onto the table plane. So the pushing directions considered are not the principal directions but their projections.

Next, the pose of the gripper is computed accordingly to the gripper shape, to the shape of the objects and the pushing direction.

In Figure 5.2 is possible observing the profile of the gripper mounted to the base of



(a) **(b)**

Fig. 5.1: Example of pushing directions. The principal directions are initially computed on the segmented surface seen by the depth sensor (Figure (a)). Next, they are projected onto to the table plane and they are associated to the object's model (Figure (b)), whose barycentre is calculated as the mean coordinate of the segmented surface plus its projection.

Fig. 5.2: The gripper and its base.

Fig. 5.3: Profile view of a desired pose for pushing an object.

the gripper, highlighted by the blue color. The base has a circular shape and the gripper's depth is less than the one of the base. It is undesirable pushing an object with a circular, or spherical, shape for the end effector because there is more uncertainty on the resulting path of the manipulated object. The gripper has no a circular shape and it is symmetric, this makes it suitable to push an object with a certain stability (i.e. make the object follow the desired path) during the action. Since we want a pushing action as accurate as possible, we don't want that the gripper's base touches the manipulated object.

Knowing also the height of the objects retrieved by its OBB, it is possible having a pose for the gripper is such a way that the gripper's base does not touch the object. The gripper's pose, relative to the object, is computed in manner to locate the red point of Figure 5.2 to be at the same height of the object. In this way the fingers will fully touch

(a) (b)

Fig. 5.4: Possible pushing poses to push an object along its principal axis: In Figure (a) the closing direction of the gripper is orthogonal to the pushing direction, and for the case depicted in the figure the gripper would likely push also the black juice box. In Figure (b) the closing direction of the gripper is parallel to the pushing direction.

the object during the pushing action. Moreover, to make easy for the robot reaching the pushing pose, it was defined to be a certain distance from the object (in our experiment it was set to 5cm). It would be difficult to reach the pose of Figure 5.3 without colliding with the object. Thus the pushing pose is a bit far away from the object to avoid dangerous collisions.

Due to the limited opening width of the gripper (7 centimetres) the objects the robot can manipulate are thin. This means that when pushing along the principal axis, the object's width is likely small (Figure 5.4a). Pushing in such a way the gripper would likely push also the black juice box since the brown box is very thin. Therefore, when pushing, the pushing pose can have either the closing direction[1] parallel to the pushing direction if the object is thin, or orthogonal if the object is wide. This is because the pose of Figure 5.4a, with the gripper's closing direction orthogonal to the pushing direction, is more stable since the contact points (the fingers) are more distant between themselves.

Having the projections of the principal components, the table normal and the desired coordinates for the gripper's closing point it is possible defining a rotation and translation

[1]The closing direction is the direction along with the gripper's fingers move when closing or opening.

matrix that define the pushing pose. To push along direction 1 those matrices are:

$$R = \begin{bmatrix} dir2_X & dir2_Y & dir2_Z \\ dir4_X & dir4_Y & dir4_Z \\ -n_x & -n_y & -n_z \end{bmatrix} \qquad T = \begin{bmatrix} c_x \\ c_y \\ c_z \end{bmatrix} \tag{5.1.1}$$

where $dir2_X$ refers to the x component of the versor that defines the direction 2 with respect to a world frame, $\overrightarrow{n}$ is the table's normal pointing upward and $\overrightarrow{c}$ is the desired tool closing point[2]. The pushing pose is sketched in Figure 5.5.

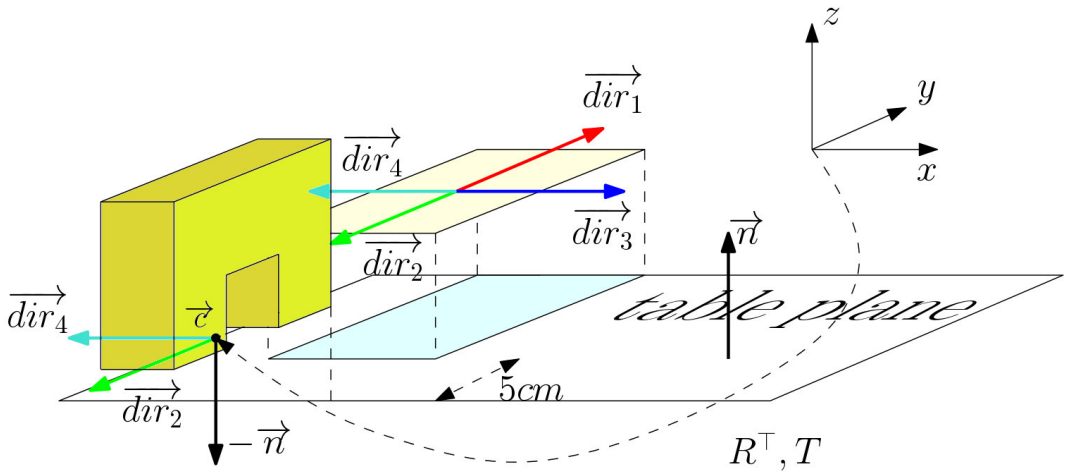


Fig. 5.5: Pushing pose for pushing direction 1. The yellow surface is the detected surface of the object and the blue one its projection. Here, the arrows refer to the the pushing directions of the object centered in the barycentre of the segmented top surface. The world reference frame is represented by x, y and z.

As aforementioned, the planner considers to push the objects for a sufficient length, denoted by the scalar l, to make it graspable. To do so, the length of the pushing is given by the perception subsystem that knows how long it has to push to move it up to a graspable pose.

To retrieve the path we consider the origin of the frame associated to the pushing pose as the initial point, and the barycentre of the object summed to the pushing length l along the associated pushing direction, direction 1 for instance, as final point. The pose is then defined by the orientation expressed by the rotation matrix of Equation 5.1.1.

The path is then given by the discretization of the linear trajectory connecting the initial and final points in the space. In particular, the discretization is given by three

[2]The gripper's closing point is finger's contact point. (Although in this case the fingers do not touch themselves)

poses: one for the pushing pose, one for the final one and one for a pose at the midway. For each pose the inverse kinematic is calculated.

When the robot approaches the pushing pose it could collide with other objects. It would be suitable to use *MoveIt!*[3] which can build an octomap representation of the scene and find a collision-free path. The integration of *MoveIt!* will be a future work. To avoid the collisions we considered a pre-pushing pose which is the same of the pushing pose but translated, accordingly to the table's normal, 30 centimetres upwards. This pre-pushing pose is easier to reach without collisions. After the execution of the pushing action the robot goes to its *home* pose (depicted in Figure 1.7a) in order not to stay inside the depth sensor's view. When it goes to home it might collides with some objects, so also for the final pose we define the post-pushing pose as the final path's pose translated, accordingly the table's normal, 30 centimetres upwards. In this way the pushing trajectory is defined by a total of 5 poses.

Despite this very basic strategy and the absence of a controller to follow the linear trajectory, the robot performs an approximatively linear trajectory that fits well to our purposes. In Figure 5.6 all the poses aforementioned are depicted.

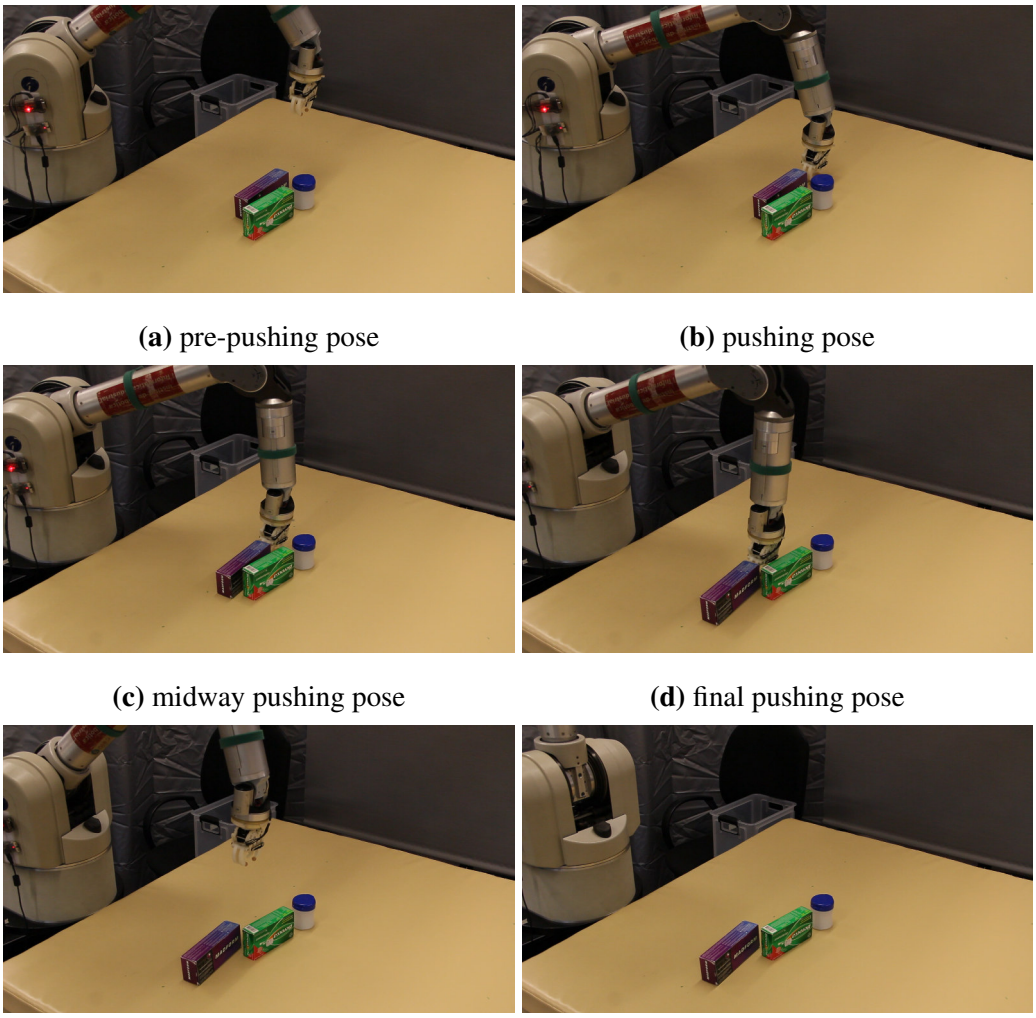
5.2. Grasping

There exists an advanced state of the art regarding grasping. Despite this, all the techniques of grasping are usually computationally expensive. Many of them rely on the identification of the shape of the objects and then a set of pre-built grasping poses is returned [8]. Other techniques rely on the identification of local features which can state if a grasping pose is feasible or not. Two goods grasping planning algorithms of this kind, which deal with novel objects, are AGILE [53] and HAF [18], despite this, they are not so robust and they are computationally expensive and not suitable for this thesis [11]. In order to have a fast planning algorithm we considered a very simple approach to grasp the objects, which is suitable only with the kind of objects we are going to interact with. Despite this, the planner presented by this thesis can be directly integrated with several grasping algorithms.

The idea is to grasp the object in manner that the gripper's closing direction[4] is orthog-

[3] Ioan A. Sucan and Sachin Chitta, "MoveIt!", [Online] Available:http://moveit.ros.org

[4] The gripper's closing direction is the direction along with the fingers move when grasping.



(a) pre-pushing pose **(b)** pushing pose

(c) midway pushing pose **(d)** final pushing pose

(e) post-pushing pose **(f)** Home pose

Fig. 5.6: Execution of the pushing action for the example of Figure 3.2

onal the principal axis of the object. The approaching direction[5] of the gripper is given by the third principal component of the object. Then the gripper's closing point coordinates are given by the centroid, of the object's top surface, translated along the approaching direction by the half of the gripper's fingers height ($2.25cm$). A generic grasping pose is depicted in Figure 5.7. In this manner a single grasping pose is obtained for each object.

Even to grasp the object a approaching pose is needed, otherwise the gripper would collide with the object attempting to reach the grasping pose moving it away, and the grasp would fail. This pose is the pose at which the robot opens the gripper to grasp afterwards the object. The approaching pose is simply defined by the grasping pose translated along its approaching direction by 10 centimetres. Once the object has been grasped it may

[5]The gripper's approaching direction is the direction along with the gripper approaches the grasping pose.

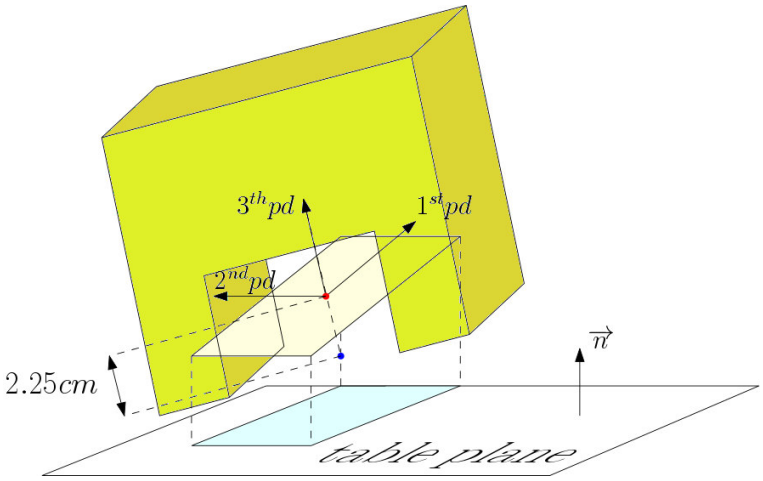
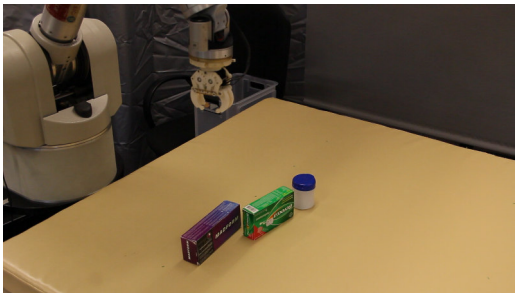


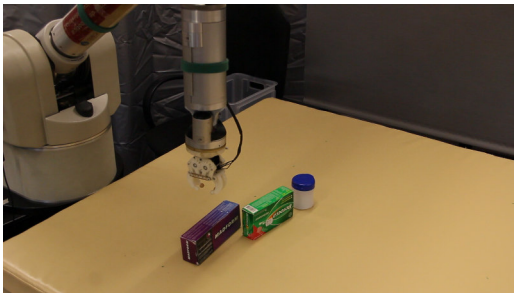
Fig. 5.7: Grasping pose for a generic object. The yellow surface is the detected surface of the object and the blue one its projection. Here, the arrows refer to the the principal directions of the object centered in the barycentre of the segmented top surface. The red point is the barycentre of the segmented top surface and the blue one is that barycentre transleted along the approaching direction by $2.25cm$.

easily collides with the others, therefore also a post-grasping pose is defined by translating the grasping pose for 30 centimetres along the table's normal. This post-grasping pose is used both to reach the approaching pose avoiding collisions and to move the object at a sufficient high to avoid collision when moving to the dropping pose. The dropping pose is the pose at which the robot opens the gripper dropping the object down into a bin.

All the poses of the grasping actions are described in detail in Figure 5.8 for the example of Figure 5.6.

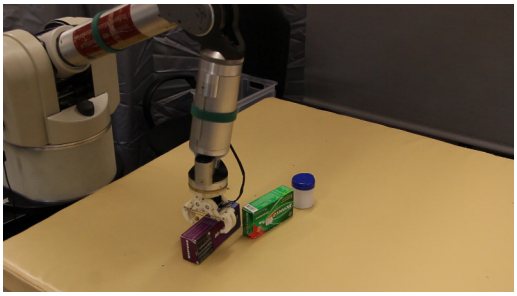


(a) post-grasping pose

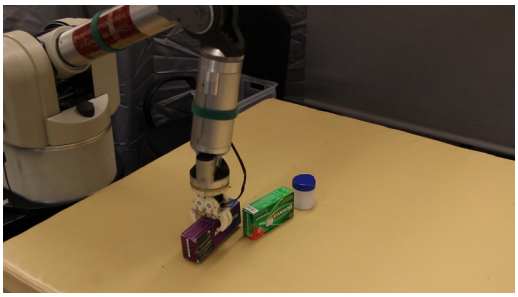


(b) approaching pose

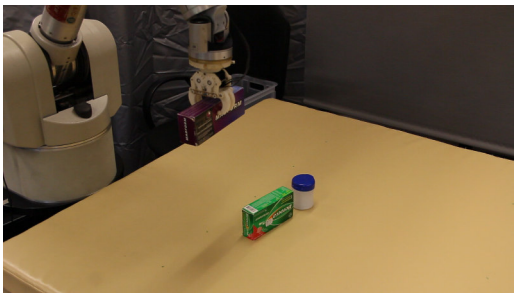
(c) opening gripper



(d) grasping pose



(e) closing gripper



(f) post-grasping pose

(g) dropping pose

(h) opening gripper

Fig. 5.8: Execution of grasping action for the purple box of Figure 5.6

6. Software design

In this chapter the software design[1] and the external libraries used are briefly described. Sections 6.1 and 6.2 describe the libraries developed and the general ROS architecture of the system. In section 6.3 how to use the code is explained. Although that section is a brief tutorial of how to use code for people familiar with ROS, I believe it is interesting for the reader to skim the images to have an insight of the system. Some of the tools used to implement the system have been mentioned in the previous chapters (OpenCV, PCL, FCL, Fast Downward planner), the remaining are:

- **ROS Indigo:** The *Robot Operating System* [46] is a robotics open-source middleware, i.e. a collection of software frameworks, for robot software development. It has infrastructure layer that is shared by all platforms using the operating system. This allows a very modular software programming and the result is the creation of software packages of easy integration. For this reason is widely used for research purposes. The entire pipeline of the thesis is developed in ROS and all the software is developed as C++ libraries that are used in ROS packages.

- **Gazebo:** It is an open source simulator for robotics [32]. It allows to reproduce complex indoor and outdoor environments populated by multiple robots using a robust physic engine. It has an interface to ROS allowing to accurately test the planning system before the implementation on the real platform.

The algorithm relies heavily on the PCL library to do the following operations: filtering, segmentation, plane estimation, principal component analysis, projections onto the table plane and convex hulls. The FCL library was used only for collision detection between the convex hulls of the objects and the gripper as well. Whereas the Fast Downward planner is used for the decision making step feeding it with a coherent problem description.

[1] All the development has been carried out in Ubuntu 14.04 LTS.

6.1. C++ code

All the code was developed as independent C++ libraries that can be easily embedded in a ROS package and they are all available in Git [50] repositories. All the libraries are provided with a documentation[2] done in Doxygen [55], including some examples. However, a basic usage of the codes is described in Section 6.3. The libraries are divided by the specific task they do:

- **Stereo vision**: this library (`stereovision`) include a class to compute the stereo calibration, stereo matching and point cloud computation of a stereo system. The implementation works as a wrapper of OpenCV methods to easily and intuitively calibrate the system and get a point cloud of the scene.

  Git repository: https://github.com/NicolaCovallero/StereoVision.git

- **Object segmentation**: this library (`tos_supervoxel`) performs the object segmentation described in Section 4.3 and it was developed in the scope of a training at IRI institute[11].

  Git repository: https://github.com/NicolaCovallero/tos_supervoxels.git

- **State generation**: this library (`table_clearing_planning`) performs the state generation described in Section 4.4 given a set of segmented objects and the table plane normal, both provided by `tos_supervoxel` library.

  Git repository: https://github.com/NicolaCovallero/table_clearing_planning.git

- **Fast Downward wrapper**: this library (`fast_downward_wrapper`) works as a wrapper for the task planner. This is because the planner is not provided with an API, thus, the only way to use it is writing in two different files the domain and problem files, as mentioned in Section 3.4.4, and launch the executable that reads those files and gives as output the plan. This library is designed in manner to be easily used by any program, not strictly the ones of this thesis.

  Git repository: https://bitbucket.org/NicolaCov/fast_downward_wrapper

[2]The documentation will be subjected to improvements. To use it go in the build folder `trunk/build` and launch the command `make doc`. Then, open the documentation: `firefox ../doc/html/index.html`

The C++ libraries developed need the aforementioned open-source libraries commented in Chapter 4 to be correctly installed. Then, for each developed library it is sufficient to move to its root folder and launch the commands:

```
$ cd trunk/build
$ cmake ..
$ sudo make install
```

Each one is provided with examples to show how to use the library.

6.2. ROS implementation

The ROS code has been separated in different packages in order to make it as modular as possible. The code was developed to be easy and intuitive to use for anyone when possible, such as the code for segmentation or for wrapping the task planner. Despite this, it may be tricky to use because it has been coded using a specific class of the IRI institute following the philosophy of LabRobotica[3]. In the LabRobotica web page is possible also to find many packages to handle the WAM robot that are omitted in the following. However all the ROS code is based on the C++ libraries aforementioned that are easy to embed in any program. The ROS packages developed are the following:

- The package `basler_stereo` which implements a node, using the `stereovision` library, to perform the stereo rectification and stereo matching of the images obtained by two, non synchronized, cameras. The point cloud is published as a PointCloud2 message on a topic whenever there is at least one subscriber.

  Git repository: https://github.com/NicolaCovallero/basler_stereo.git

- The package `iri_tos_supervoxels` to segment the objects and estimating the table plane coefficients using `tos_supervoxels` library. This node is implemented as a service that segments a given point cloud on request.

  Git repository: https://github.com/NicolaCovallero/iri_tos_supervoxels.git

- The meta-package (a collection of more packages) `iri_table_clearing_planning` to manage all the operations specific for the table clearing task. These are divided

[3]LabRobotica: http://wiki.iri.upc.edu/index.php/LabRobotica

in different packages accordingly to their function[4]

- `iri_table_clearing_predicates`: this packages implements a service node that computes the symbolic predicates on request using the `table_clearing_planning` library.
- `iri_table_clearing_execute`: this packages implements a service node that computes the IK of the robot and it executes the action if feasible.
- `iri_table_clearing_decision_maker`: this package implements a node that, given the last received point cloud from the depth sensor, manages all the nodes and calls all the required services following the scheme sketched in Figure 6.1.
- `iri_table_clearing_description`: this package includes an enhanced URDF description of the WAM robot that includes the simplified model of the gripper. This package is used only for simulation together to the following package.
- `iri_table_clearing_gazebo`: this package includes example of worlds for Gazebo and a python script to simulate the grasping. The gripper was model without any actuators, thus the grasping action is supposed to be deterministic and the object is removed from the world of Gazebo when the robot approaches the object to grasp it.

Git repository: https://github.com/NicolaCovallero/iri_table_clearing_planning.git

- The package `iri_fast_downward_wrapper` to work as an API to interface the code to the task planner through the `fast_downward_wrapper` library. This package implements a service node that can handle two different services on request: the first one to write a problem PDDL file and call the executable of the planner, the second one to write both the domain and problem PDDL files and call the planner.

  Git repository: https://bitbucket.org/NicolaCov/iri_fast_downward_wrapper

Here, the perception subsystem is composed of packages `iri_tos_supervoxels` and `iri_table_clearing_predicates`, the planning subsystem is made of packages `iri_fast_downward_wrapper` and `iri_table_clearing_decision_maker` and part

[4]Only the more relevant ones are cited.

of the package `iri_table_clearing_execute`. In the code explanation we drop off the division in subsystems and we consider the planning system as a unique one.

The software architecture is sketched in Figure 6.1. The decision maker node does the following operations:

1. Wait for a point cloud;
2. Call the segmentation service giving as input the point cloud and receiving as result the segmented tabletop objects and the plane coefficients;
3. Call the state generation service giving as input the tabletop objects point clouds and the plane coefficients and receiving as result all the states, as well the poses for the pushing and grasping actions;
4. Call the Fast Downward planner giving as input the states;
5. Call the action execution service. The result of this service is a boolean variable which specifies if the requested action has the inverse kinematic feasible. If it is not feasible, the decision maker adds to the state the `ik_unfeasible` predicate for that action and it requests a new plan going to point 4. If the action's IK is feasible the action is executed and the system starts again to point 1.

6.3. How to use

In this section the usage of the code, which is quite complex to use, is described providing examples of the planning system in the simulated environment. However, only a brief explanation is given. All the libraries are provided with a Doxygen documentation and examples. Moreover the codes might be subjected to changes in the future. This section supposes that all the previously mentioned ROS packages and C++ libraries have been downloaded and correctly installed (see also the installing section of LabRobotica[5]).

Prepare the world Gazebo[6] does not allow to disable the gravity for a specific model during the simulation, this makes the arrangement of the objects very hard because if

[5] http://wiki.iri.upc.edu/index.php/LabRobotica_Software_Installation

[6] The version used is the 2.2.6.

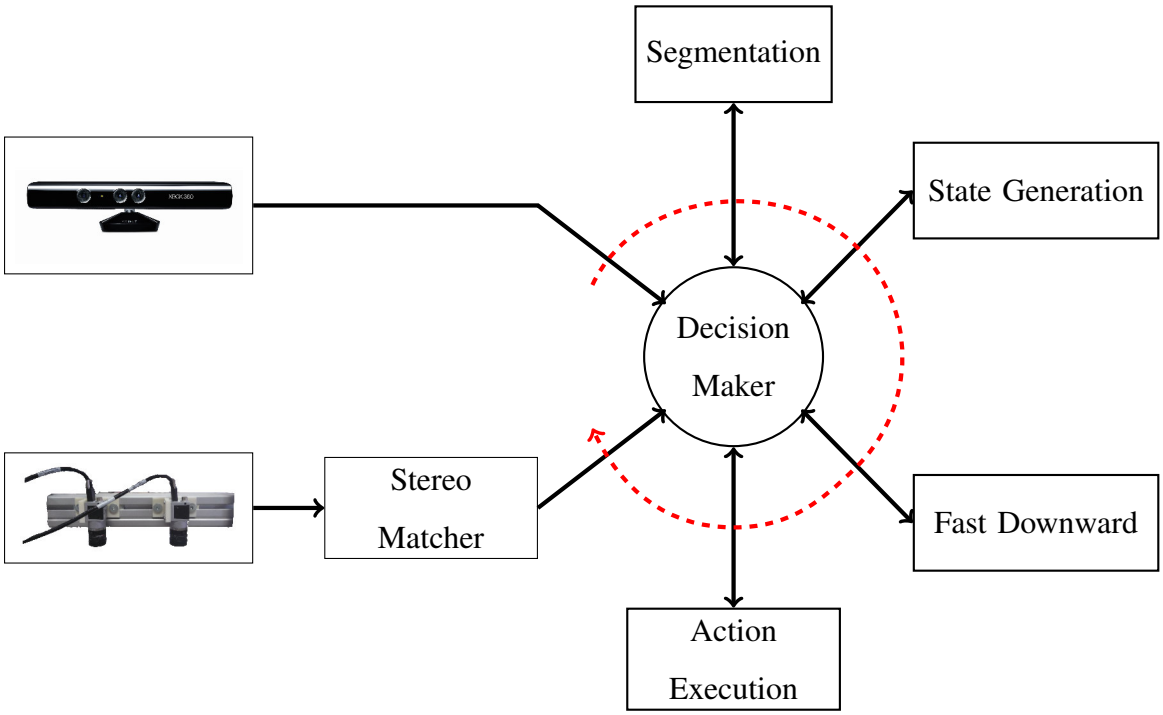


Fig. 6.1: Software architecture. The decision maker waits for a point cloud captured by the Kinect or by the stereo vision system, depending on which one is employed. Next, it calls the services of the other nodes by following the order indicated by the dashed line.

you translate upward the object and then rotate it, moving from the translation tool to the rotation one, the object falls. A trick is creating the world disabling the gravity for all the objects, then the user arranges the objects as he/she pleases. To do so, launch in a terminal the command:

```
roslaunch iri_table_clearing_gazebo world_only.launch world:=
    test_textured4
```

Where `test_textured4` is a previously world defined to use as example and it should look like in Figure 6.2. Then disable the physics by checking `enable physics` box in the `world->physics` panel on the left. Then, add the desired models to the world, enable the physics and save the world in the `worlds` folder of package `iri_table_clearing_gazebo`.

Notice that some models are needed and these include models to simulate the experimental set up at IRI laboratory and some textured objects. These are available at

https://gitlab.iri.upc.edu/perception/gazebo_models.git

https://github.com/NicolaCovallero/stereo_models.git

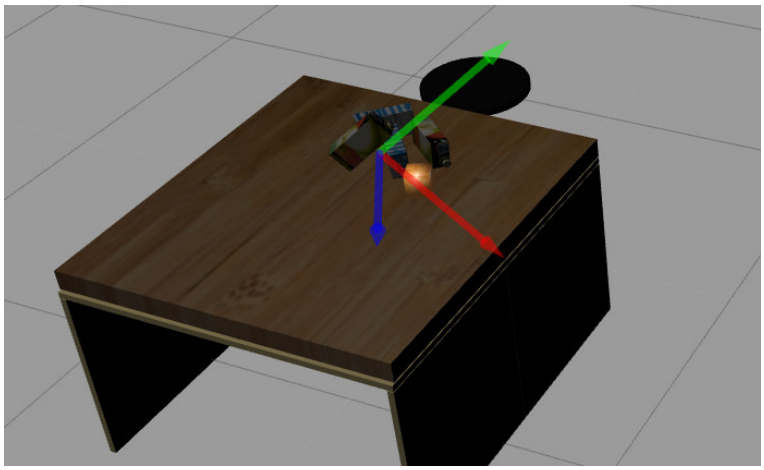


Fig. 6.2: World arrangement.

Fig. 6.3: Rendered textured objects models.

Stereo vision The stereo vision system has been implemented in the simulated environment with a different focal length in order to have a field of vision that covers a sufficient area of the table. The area covered by the implemented stereo system is too small with respect the objects the robot has to manipulate. Repository `stereo_models` includes a chessboard calibration pattern to perform a simulation of the calibration and some texture objects models (e.g. Figure 6.3). Notice that in Gazebo, since the cameras parameters are known, it is not necessary performing a calibration because all the data to estimate are perfectly know. However, to make it as general as possible, we calibrate the system in the simulated environment to show that we are effectively simulating a real case scenario. Next, the textured models are required since for the stereo matching the texture plays a fundamental role. In a non-textured surfaces the features are wrongly matched and a no-sense point cloud is obtained (Figure 6.4).

The models of the stereo system is defined in the xacro files `left_camera.xacro` and `right_camera.xacro` (in package `basler_stereo`), where the relative transformation between the cameras has to be fixed at the beginning of file `right_camera.xacro`. The transformation between a generic reference frame and the left camera has to be set at the

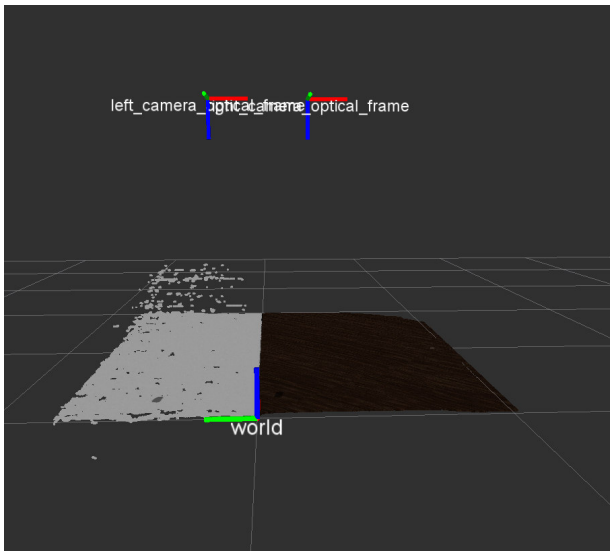


Fig. 6.4: Comparison between point clouds with textured and non-textured surfaces. Here, there is a non-textured gray (i.e. all the points of that surface have the same rgb value) floor whose coordinate points are wrongly estimated because of false matches in the stereo matching stage. The right surface is a textured wood floor and its points are perfectly measured.

beginning of file `left_camera.xacro`. Notice that in this files the cameras are defined as Gazebo plugins and the user can set whatever desired values, such as the focal length playing with the `horizontal_fov` (horizontal field of view, in radians) parameter. Once the user has defined the stereo system as he/she pleases, the calibration procedure begins by launching in a terminal the following command:

```
$ roslaunch basler_stereo basler_stereo_calibration.launch
```

This command opens Gazebo with the stereo system and a calibration chess pattern. The next step is saving the images captured by the cameras. To do so, first create two folders where you are going to save the images. Let's call those folder `calib_left` and `calib_right` and let's locate them in the folder `data` inside the `basler_stereo` package. Then, launch in a terminal, inside the folder `calib_left`, the following command:

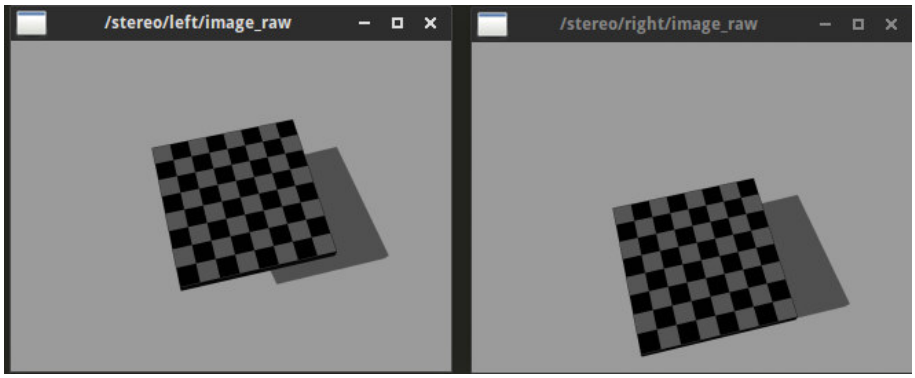
```
$ rosrun image_view image_view image:=/stereo/left/image_raw
```

Then, launch in a different terminal, inside the folder `calib_right`, the following command:

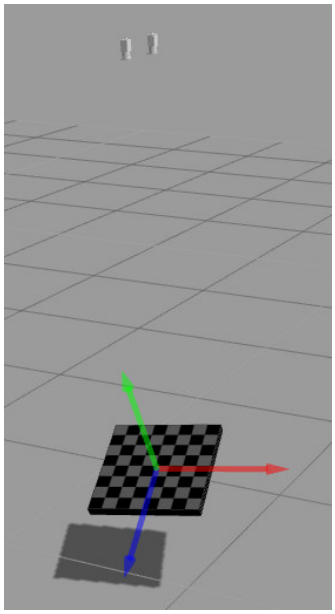
```
$ rosrun image_view image_view image:=/stereo/right/image_raw
```

Two windows appear showing the captured image of the two cameras (Figure 6.5a)[7].

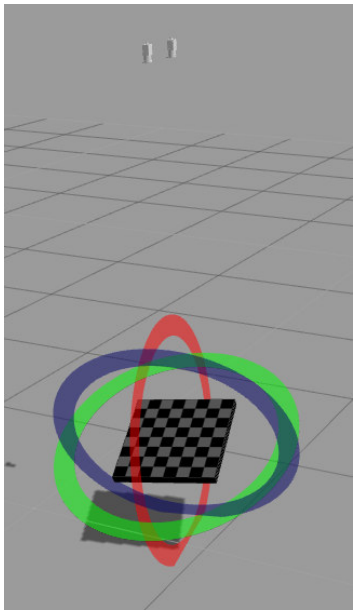
[7]Notice that to easy the calibration process you may want to add lights in Gazebo. This helps the



(a) Cameras views



(b) Translations



(c) Rotations

Fig. 6.5: Calibration of the simualted stereo system.

Next, by right clicking on each image view the images are saved in JPG format. The chessboard can be translated and rotated in Gazebo (Figure 6.5) to capture multiples images, we recommend to grab at least 10 images. Finally, to calibrate the system it is required to run the `stereo` executable of the `StereoVision` project inside the folder that contains the two xml list files, one per camera, that list all the pairs images to use for calibration. For example, the file relative to the left camera has the following format:

```
<?xml version="1.0"?>
<opencv_storage>
<images>
  "./calib_right/frame0000.jpg"
```

identification of the chessboard corners when calibrating.

```
  "./calib_right/frame0001.jpg"
  ...
  "./calib_right/frame0020.jpg"
</images>
</opencv_storage>
```

Notice that the order of images in the two files .xml must be the same otherwise the images pairs are wrong.

To calibrate the system launch the following command in the folder containing the files:

```
$ <path to StereoVision project>/bin/stereo left_list.xml right_list.
    xml
```

a menu will be printed in the terminal, press 'c' to perform the calibration. The program will ask the user for the number of inner corners, by defining the number of corners in width and height (7 and 7 for the chess pattern in Figure 6.5), and for the size of each square (0.0625m for the chess pattern in Figure 6.5). Notice that the measurement unit you choose to use here is fundamental because it will affect all the cameras parameters (focal length, translation, etcetera ...) and the resulting point cloud. Define the square size in metres. Then it will also ask to give a down scale factor to speed up the process. Despite this, it is better to use always a down scale factor of 1, i.e. not downscaling at all, because we obtain more robust result.

At this point the calibration actually starts, this process detects the inner corners for every pair of the images listed in the input files, and then it performs the stereo calibration. Finally, it prints out some information such as the reprojection error as quality metric, it should be around 1, or less, to have a nice calibration. It pops up some windows that show the rectified input images with some lines that serve to visually check if they are effectively aligned (something similar to Figure 4.15). If the rectified images are not column-aligned, or row-aligned, something gone wrong. Press any button to keep on. When all the rectified pair have been showed the program saves the intrinsics and extrinsics parameters on files. This produces the `intrinsics.xml` and `extrinsics.xml` files that contain all the relevant informations regarding the stereo system. The easiest way to check if the calibration has been correctly done is by checking the translation vector T in the `extrinsics.xml` file.

Next, the stereo matching algorithm parameters must be tuned. This is done by using the executable `stereo` of the `StereoVision` project on example images. This may be captured in a similar way as done for calibration using the ros package `image_view`. Pay attention that now the program is going to perform a matching of pixels and the image needs to be textured[8]. The textured set up environment, including the robot, can be launched by the following command:

```
$ roslaunch iri_table_clearing_gazebo estirabot_gripper_stereo.launch
    world:=test_textured4
```

Now grab the images in the same manner we did for calibration, i.e. using the node `image_view`.

To perform the tuning now is necessary to move in the folder where the images have been saved and launch the command:

```
$ <path to StereoVision project>/bin/stereo left_textured_image.jpg
    right_textured_image.jpg
```

The same menu of before is printed on the terminal. Press the 'u' button and enjoy the tuning. The program asks to specify if its a vertical stereo arrangement or not, in our case it is. Next it asks to use a downscale factor, this is important because it severely affects the computation time. In our experiments we set up a down scale factor of 4, meaning that we worked with images 4 times smaller than the grabbed ones. Next, it asks to choose the stereo matching algorithm to use (block matching or semi global block matching). Then, it displays the rectified frames (it also plots lines by pressing 'l') to visually check that everything is fine. By pressing 'q' a new window appears containing two images, the disparity map on the left and the input left image on the right (Figure 6.6). The right image is useful because it allows you to understand which points of the scene have a matched given the displayed disparity map. Another menu is printed that gives several options for tuning the stereo matching algorithm. By pressing 'q' the tuning process terminates and it asks whether to save or not the parameters. It will save the parameters in the file `sgbm_parameters.yml` if you used semi global block matching algorithm, or in the file `bm_parameters.yml` for the block matching algorithm. Notice that the current folder

[8] Any model the user wants to use with the stereo system must be textured. This can be done by having a .stl model and then a texture can be applied using the open-source project Blender. Then export the textured model as .dae and configure a new Gazebo model.

must contain the `intrinsics.xml` and `extrinsics.xml` files otherwise it will return error.

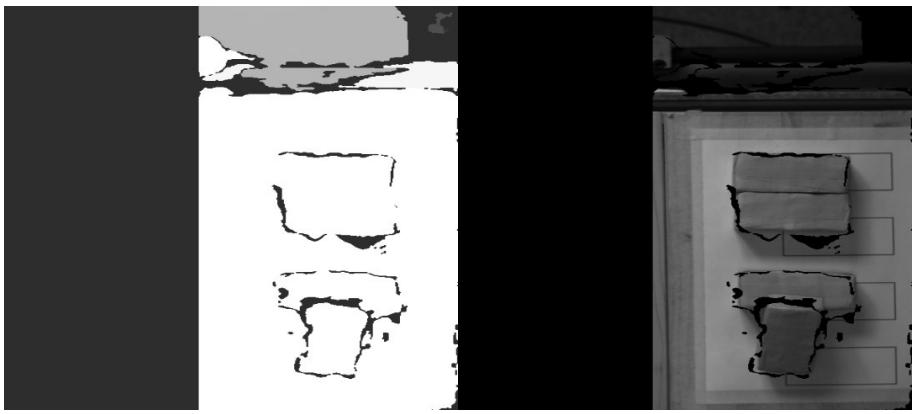


Fig. 6.6: Stereo matching algorithm tuning. This figure refers to the example of Figure 4.17.

The stereo matching tuning can be checked by running the aforementioned command and selecting the option to perform the stereo matching. It asks for the downscale factor (it must be the same used for tuning), whether it is a vertical arrangement, the matching algorithm and then it saves a point cloud in the current folder that can be visualize afterwards with the `pcl_viewer`.

Set up the robot and depth sensor Let's respawn the robot and the stereo vision system:

```
$ roslaunch iri_table_clearing_gazebo estirabot_gripper_stereo.launch
    world:=test_textured4
```

Now, the robot and the stereo system has been spawned in Gazebo and the cameras are grabbing. The next step is performing the stereo vision launching the node of package `basler_stereo`:

```
$ roslaunch basler_stereo stereo.launch
```

Remember that all the files aforementioned (`intrinsics.xml`, `extrinsics.xml`, `sgbm_parameters.yml`, `bm_parameters.yml`) must be located inside the folder `config` of package `basler_stereo`. However this can be changed by modifying the file `stereo.launch`. Now the stereo system is initialized but to compute a point cloud it is waiting for someone to subscribe to the topic `/stereo/pointcloud`. In another terminal launch:

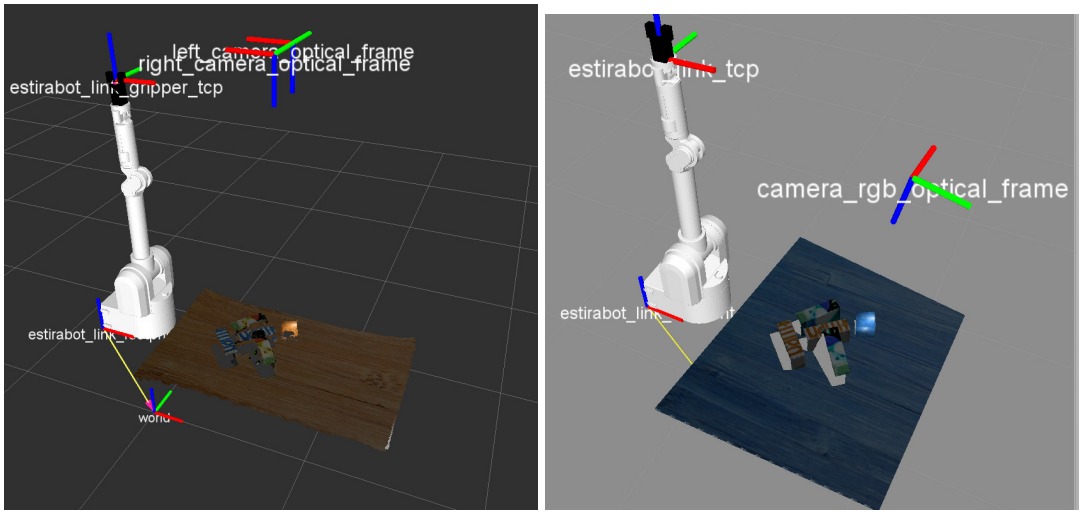
```
$ rviz
```

and add the PointCloud2 voice, subscribe it to the topic `/stereo/pointcloud` and select `world` as reference frame.

To spawm the robot with kinect instead of the stereo system launch:

```
$ roslaunch iri_table_clearing_gazebo estirabot_gripper.launch world:=
    test_textured4
```

What should be seen in Rviz is similar to Figure 6.7.



(a) Stereo **(b)** Kinect

Fig. 6.7: Pointclouds obtained from the simulated sensor. The Kinect was set to have a very low noise, so also the cameras but the stereo system produces a more noisy point cloud because of the stereo matching. Notice also that the simulated Kinect produces erroneous rgb values but our segmentation algorithm does not care about rgb values.

Object segmentation To perform only the object segmentation on the captured point cloud launch in another terminal

```
$ roslaunch iri_tos_supervoxels tos_supervoxels_stereo.launch
```

for the stereo system, or

```
$ roslaunch iri_tos_supervoxels tos_supervoxels.launch
```

if you spawned the Kinect. Then, this node produces a point cloud for tabletop objects and each object is labelled by a different color (Figure 6.8). This message is published in the topic `/segmented_objects/points`.

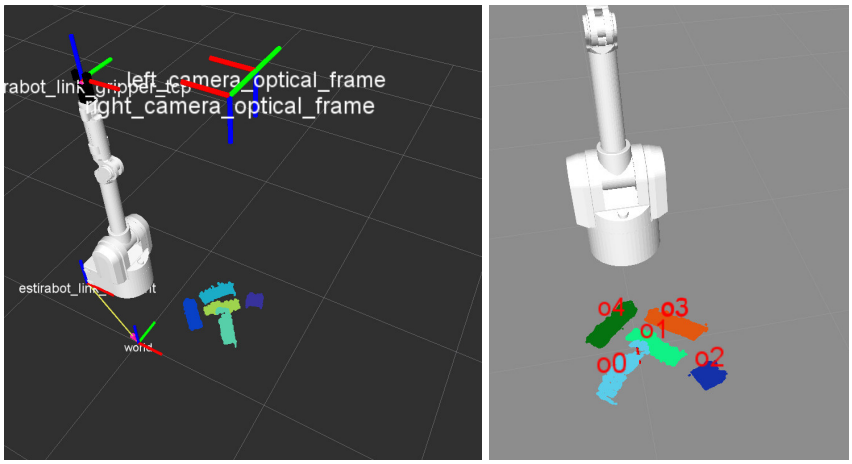


Fig. 6.8: Segmentation of the point cloud produced by the stereo system.

Deploying the whole planning system In this paragraph we briefly explain how to launch the whole planning system. Each command must be launched in a different terminal.

First of all launch:

```
$ roscore
```

Then the system must be spawned, either with the stereo system:

```
$ roslaunch iri_table_clearing_gazebo estirabot_gripper_stereo.launch
    world:=test_textured4
$ roslaunch basler_stereo stereo.launch
```

or with the Kinect:

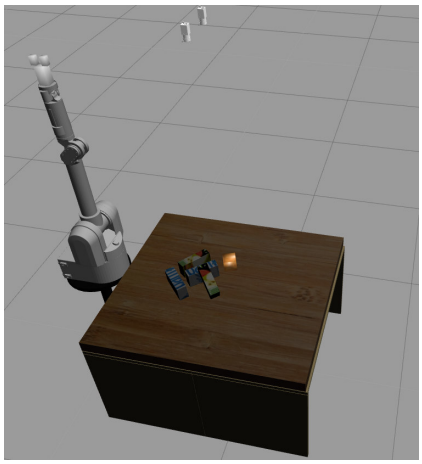
```
$ roslaunch iri_table_clearing_gazebo estirabot_gripper.launch world:=
    test_textured4
```

The spawned robot and world in Gazebo should be similar to Figure 6.9.

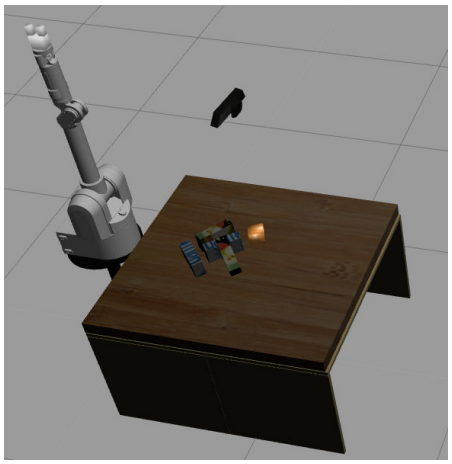
In the following we refer to the objects configuration showed in Figure 6.10.

Next, we have to launch the nodes to compute the predicates, call the planner and execute the robot, accordingly to the scheme of Figure 6.1:

```
$ roslaunch iri_table_clearing_predicates iri_table_clearing_predicates
    .launch
$ roslaunch iri_fast_downward_wrapper fast_downward_server.launch
$ roslaunch iri_table_clearing_execute iri_table_clearing_execute.
    launch
$ roslaunch iri_table_clearing_decision_maker
    iri_table_clearing_decision_maker.launch INPUT_TOPIC:=/stereo/
```



(a) Robot with the stereo vision system



(b) Robot with the Microsoft Kinect

Fig. 6.9: Gazebo world and robot.

Fig. 6.10: Objects configuration

```
pointcloud STEREO:=True
```

where the parameter `INPUT_TOPIC` can be left to the default value, i.e. omitted in the command, when the Kinect is used. `STEREO` is an another argument to specify if the system is using the stereo vision system. Then, the decision maker node asks to the user if he/she wants to repeat each iteration. By pressing 'y' it segments the last received point cloud, computes the predicates, it plans and it gives the action to execute to the execution node which solves the IK and if the action is feasible, by default, it asks to the users step by step if it can proceed to the new pose of the action. By pressing 'y' in the terminal, where the execution node was launched, the robot goes to the next action's pose, otherwise by pressing 'n' the robot goes to its home pose and the decision node performs another iteration of the planning system.

Then, launch in another terminal Rviz because very useful to understand what is going on in the system. The decision node publishes several useful markers, the most important

ones are:

`/estirabot/iri_table_clearing_decision_maker/action_trajectory`

`/estirabot/iri_table_clearing_decision_maker/objects_label`

`/estirabot/iri_table_clearing_decision_maker/pushing_directions`

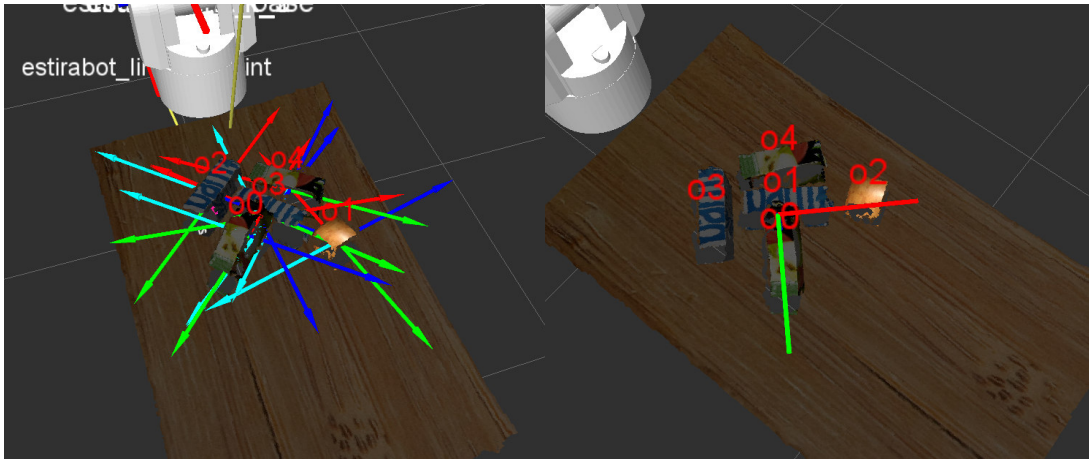
and the pose of the action (pre-pushing pose for pushing and approaching pose for grasping):

`/estirabot/iri_table_clearing_execute/action_pose`

For representing the pose choose the axis shape. The segmented objects are shown in the pointcloud published in the topic

`/estirabot/iri_table_clearing_decision_maker/cloud`

For instance, in Figure 6.11a the pushing directions of the objects are shown and Figure 6.11b shows the approaching pose[9] for grasping `o0`. Notice that in that example the can either grasp `o2` or `o0`. The second is on top of `o1` so it must be grasped before to manipulate the others. Note also that the object labels of Figures 6.11a and 6.11b are different because they refer to different point clouds and for each point cloud the labels may change since the system considers each iteration as a whole new problem.



(a) Pushing directions **(b)** Approaching pose for `o0`

Fig. 6.11: Pushing directions and approaching pose visualised in Rviz.

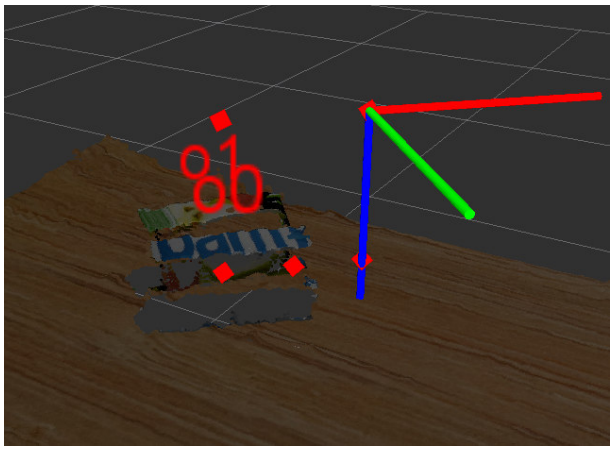
In Figures 6.12 and 6.13 an example of pushing and grasping actions in the simulated environment is shown.

[9]Here, the red, green and blue axes are the x, y and z axes respectively.

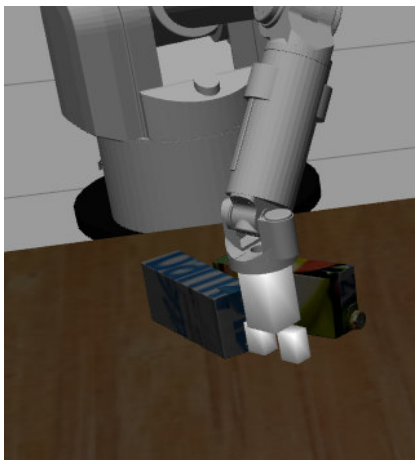
Finally, in order to have a complete visual result a gui based on `rqt_gui` has been developed. Launch in another terminal:

```
$ roslaunch iri_table_clearing_config rqt_gui.launch
```

And then you will have to load the rviz configurations.



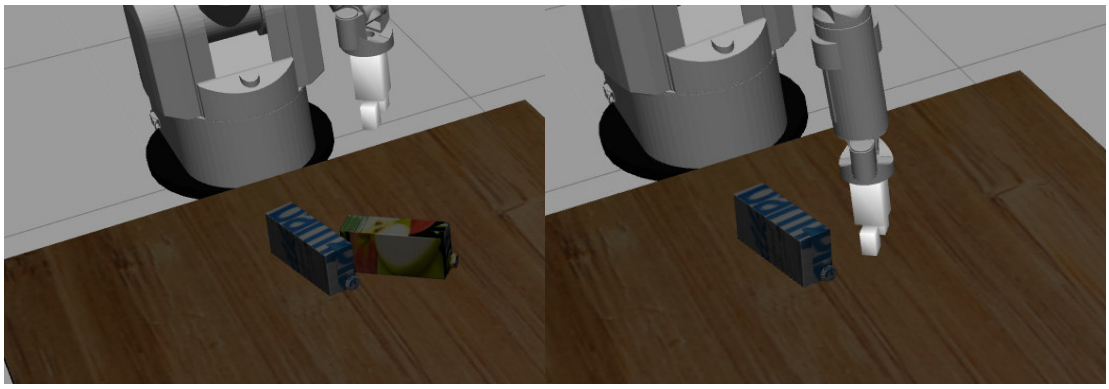
(a) Rviz visualization of the pushing trajectory



(b) Execution in Gazebo

(c) Outcome of the action

Fig. 6.12: Execution of a pushing action to separate `o0` and `o1`. In Rviz (a) the pushing trajectory is used to visually verify the correctness of the trajectory, red marks are the poses of the pushing path. Then the excution node controls the robot to execute such an action in the simulated environment (b), and the outcome is shown in (c) where it is appreaciable that the object's trajectory is not as predicted. The white mesh models are the Collada models of the Basler cameras of the stereo vision system.



(a) Post grasping pose **(b)** Grasping pose

Fig. 6.13: Continuation of the experiment of Figure 6.12. Despite the unexpected outcome of the previous pushing action, `o1` is graspable. So the system decides to grasp it. It initially locates the end-effector at the post-grasping pose, then it moves to the approaching pose. At this point the execution node calls a service to remove `o1`'s model from Gazebo (because the grasping is not simulated and supposed to be deterministic). Then it moves to the grasping pose to finally drop the object into a bin.

7. Experiments

In this chapter three experiments[1] [2] are presented in order to assert the quality and robustness of the proposed planning system. These experiments are intended to:

1. Study the performance of the planning system for scenes with 3, 4, 5, 6 and 7 objects (Figure 7.1). Each scene has been repeated three times changing the configuration of the objects.

2. Analyse the trace of one clearing trial for a scenario with 7 objects.

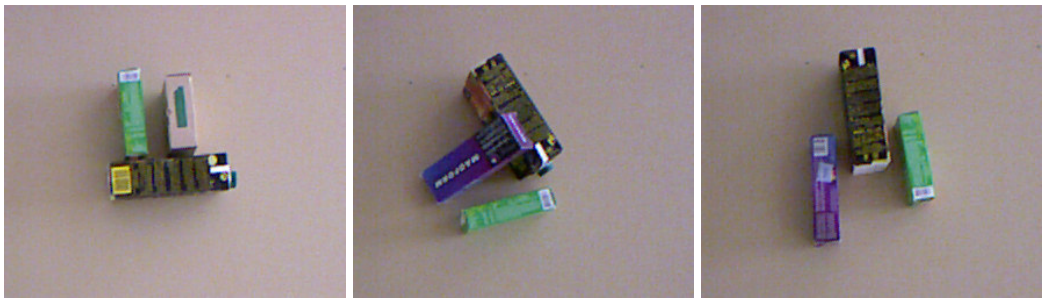
3. Do a comparison between the proposed planning system and the framework presented by Dogar and Srinivasa[16] which is the most similar work to the proposed planning system.

The first experiment aims to assert the robustness of the system and study its performance in terms of the time taken by each of the three subsystems. Table 7.1 shows some numerical results. It can be seen that, as expected, the global time increases with the complexity of the scene. In particular, the perception time used to compute the predicates, because the number of relationships between objects is larger. We recall that each object is modelled using the convex hull of the segmented object and its projection onto the table. This leads to complex mesh models that make the collision checking time consuming. If we accept to work with simple shapes, an alternative is to fit a simple model for each object, considerably reducing the time devoted to the state generation.

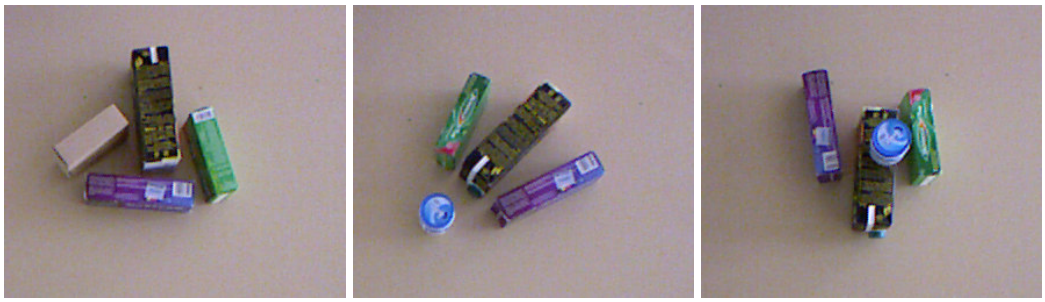
Contrarily, observe that the planning subsystem takes less accumulated time in the case of 5 objects (12.62 sec) than 4 objects (19.84 sec). The explanation is that even if the scenario contained more objects, their position were always reachable even after pushing actions and thus, backtracking was not necessary. Backtracking is time consuming because our implementation uses an optimization method to compute the IK that, naturally, is not constant in time. The evaluation of an unfeasible pose takes the maximum time.

[1] Additional material: www.iri.upc.edu/groups/perception/phasinginplan

[2] Video: https://www.youtube.com/watch?v=P4QRB0bdiGw this video refers to the first version of this thesis [12] whose main difference is that the actions do not have cost and the pushing length is function of the manipulated object and not of the surrounding ones. Despite this, in the video the robot behaves almost the same to the one presented in this thesis.



3 objects



4 objects

5 objects

6 objects

7 objects

Fig. 7.1: Initial configuration of the performed trials.

Tab. 7.1: Plan's length and elapsed times of the trials of Figure 7.1 that were correctly finished, i.e. the robot was able to grasp all the objects and no object was moved outside the working space. The elapsed times are the average times spent by the three subsystems to achieve the goal of clearing the table. Thus, several iterations are included within these data.

n° objects	3	4	5	6	7
n° actions	4.33 ± 0.58	7 ± 1.73	8 ± 1	9.67 ± 1.53	12 ± 3.46
perception [s]	8.05 ± 0.88	14.62 ± 3.82	17.97 ± 2.26	28.88 ± 4.11	45.67 ± 12.77
planning [s]	7.64 ± 4.42	19.84 ± 16.8	12.62 ± 1.74	28.38 ± 12.10	81.58 ± 36.52
execution [s]	108.70 ± 11.79	170.73 ± 33.43	194.21 ± 14.05	243.19 ± 42.61	292.26 ± 53.59
total [s]	124.40 ± 10.50	205.2 ± 37.53	224.81 ± 16.73	300.46 ± 50.69	419.5 ± 100.71
n° backtracks	0.67	2	0	3	1

The execution of the actions is the most time-expensive subsystem because our robot moves at low speed for safety. Thus, it is important to compute a good plan that contains the minimum number of actions.

As a practical lesson learnt during the experiments, we observed several unexpected outcomes during pushing actions because either the resulting trajectory of the object was slightly different to the expected one or the end-effector slightly touched other objects. One alternative is to generate more robust robot motions. In our case, with simple pushing actions, the replanning technique showed to be very suitable to handle these unexpected outcomes. An example is depicted in Figure 7.2. The robot had to move the green box along the direction of the arrow but the gripper touched the brown box too. As result, both the green and brown boxes have been moved and none of them was graspable. The system replanned and the next action was to push the brown box. The action was correctly executed and the brown box was graspable and the task could be terminated.

Replanning showed to be effective also to handle the unexpected outcomes of grasping. Some shapes could be challenging to grasp and the object could slip out if the grasp is not stable (Figure 7.3). This happened some times when the robot attempted to grasp circular objects. These objects also slipped out the working space twice and the use of subgoals showed to be effective.

We observed also that the top objects are usually the ones that prevent the majority of the actions (see the third trials with 4,5,6 objects) and that the system can recognize them

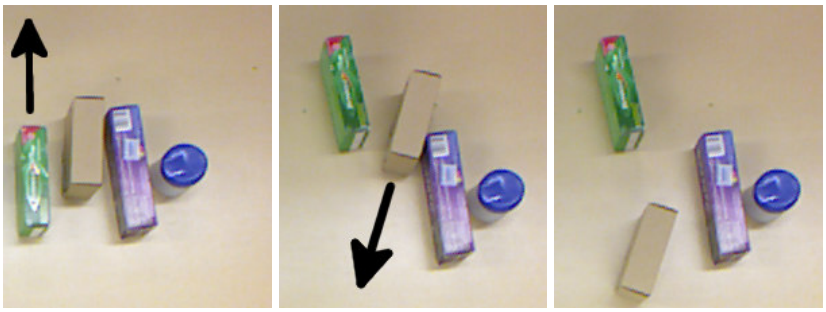


Fig. 7.2: Unexpected outcome of the pushing action. In the first scene the robot had to push the green box away but it accidentally moved also the brown box. The green box was not graspable after being pushed, the robot replanned and decided to move the brown one.

Fig. 7.3: Unstable grasp that makes the object slip out.

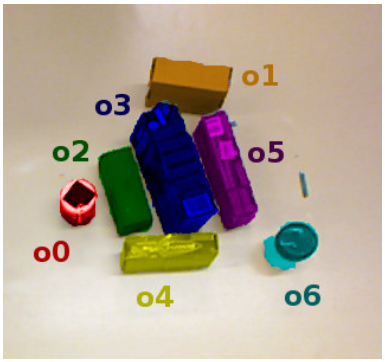
and it grasps them first.

The second experiment presents in more detail the trace of one clearing trial for a scenario with 7 objects. Figures 7.4 and 7.5 represent the actions' outcomes and their execution respectively. It is interesting noting the difference between the initial and executed plan[3]:

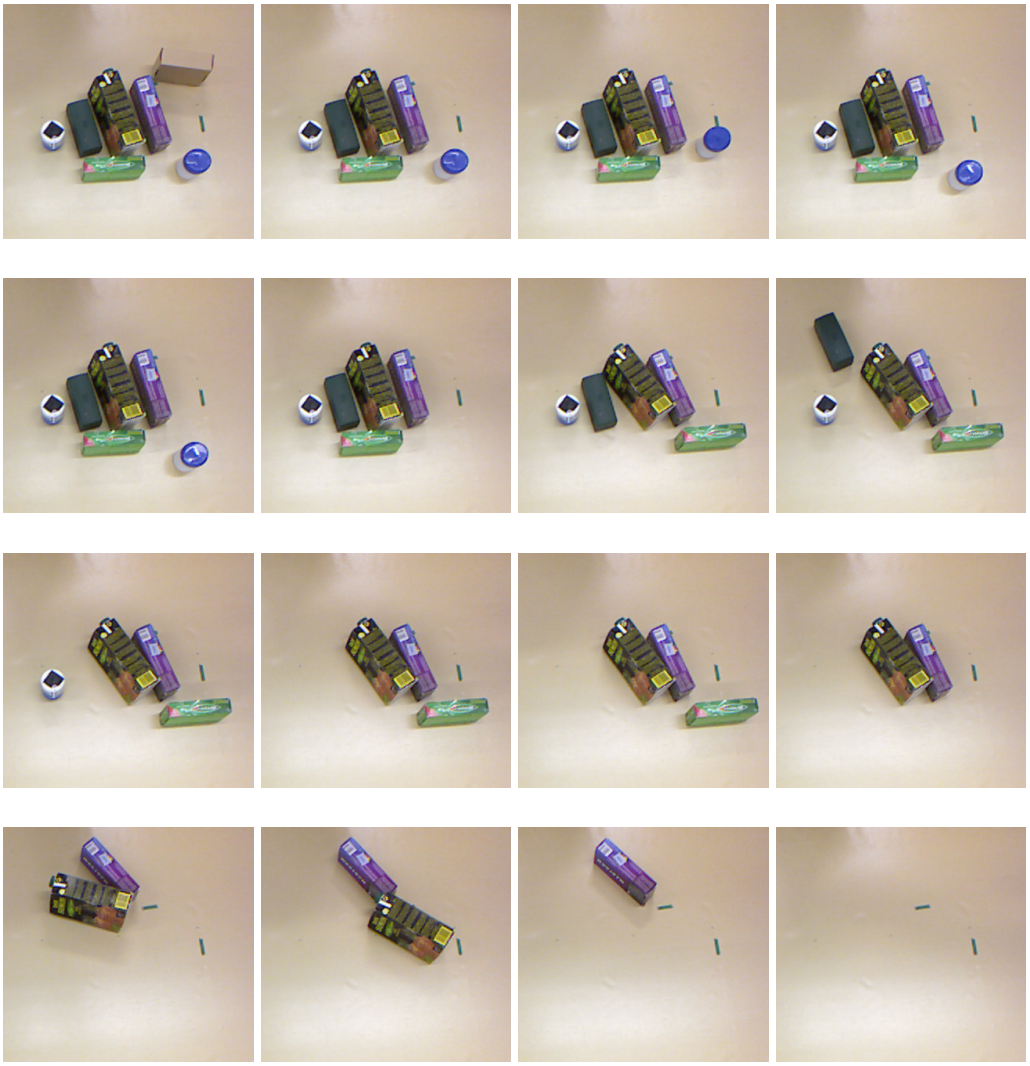
- Initial plan: `(push o1 dir1) (push o5 dir2) (grasp o6) (push o4 dir1) (push o2 dir1) (grasp o0) (grasp o5) (grasp o3) (grasp o2) (grasp o1) (grasp o4)`
- Executed actions: `(push o1 dir1) (grasp o1) (push o6 dir1) (push o6 dir2) (push o6 dir2) (grasp o6) (push o4 dir1) (push o2 dir1) (grasp o2) (grasp o0) (push o4 dir1) (grasp o4) (push o5 dir1) (push o3 dir2) (grasp o3) (grasp o5)`

Where the initial plan is the plan obtained for the first captured image (Figure 7.4a) and the executed one is the sequence of the executed actions that may differ from the initial

[3]Here the objects' label are the same along all the trace but actually, after the execution of each action the robot considers the problem as a new one with no historical information, thus the label for a certain object may change at every iteration.

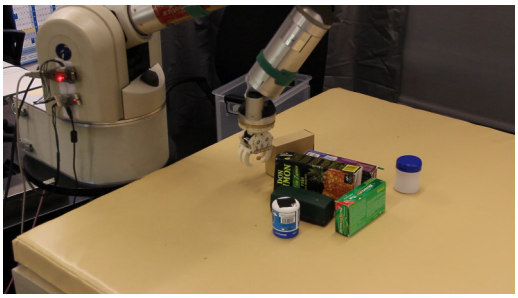


(a) Initial configuration and objects' labels

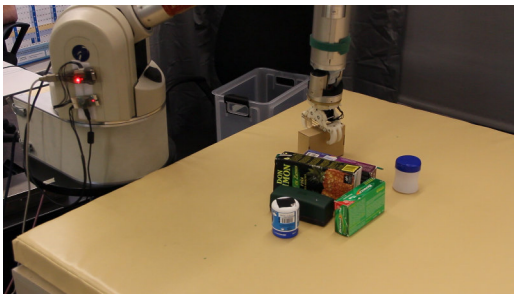


(b) Outcomes of the executed actions

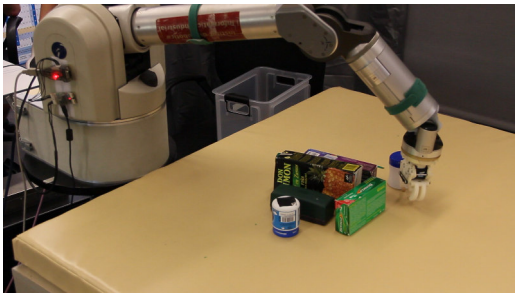
Fig. 7.4: First trial for the scenario with 7 objects.



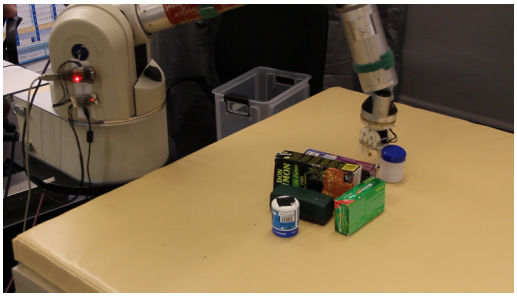
(a) (push o1 dir1)



(b) (grasp o1)



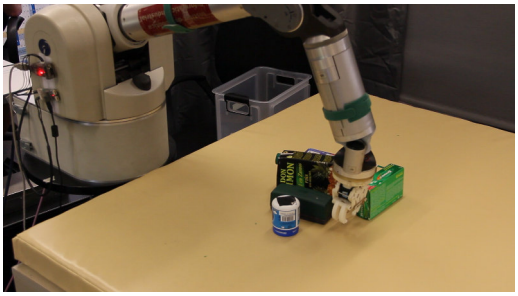
(c) (push o6 dir1)



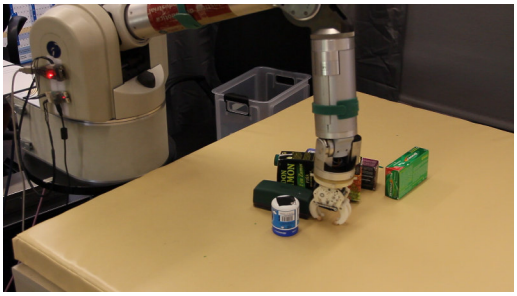
(d) (push o6 dir2)

(e) (push o6 dir2)

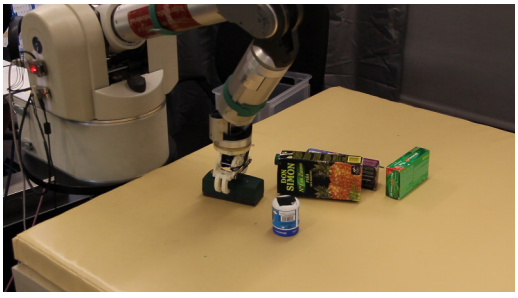
(f) (grasp o6)



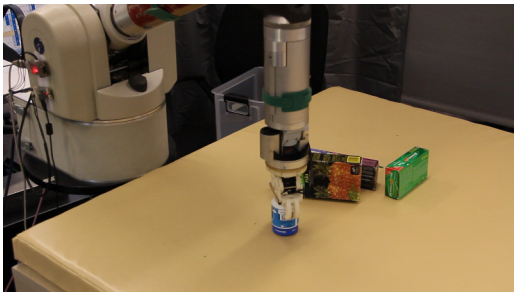
(g) (push o4 dir1)



(h) (push o2 dir1)



(i) (grasp o2)



(j) (grasp o0)

Fig. 7.5: Execution of the first 8 actions of Figure 7.4

one. In fact, the executed plan differs because of two main reasons:

1. the robot had to push some objects more times to make them graspable.

2. the robot, when pushing, might have moved more than one object.

The first case happened when the robot attempted to manipulate objects o6 and o4. This is appreciable in Figure 7.4b. From picture 3 to 6, the robot pushed o6 initially in one direction, but the object was not graspable and needed to be moved again, so it decided to push in the opposite direction and the same problem occurred. This was solved by a further pushing action. Figure 7.6 shows the grasping poses of o6 in the several poses. A similar case happened for o4, the robot had to push it twice in order to put it in a graspable pose. As lesson learnt, the probability of the outcome of the pushing action should be handled also by the pushing length. A simple but effective way is to consider a larger collision-free range for the grasping pose after being pushing.

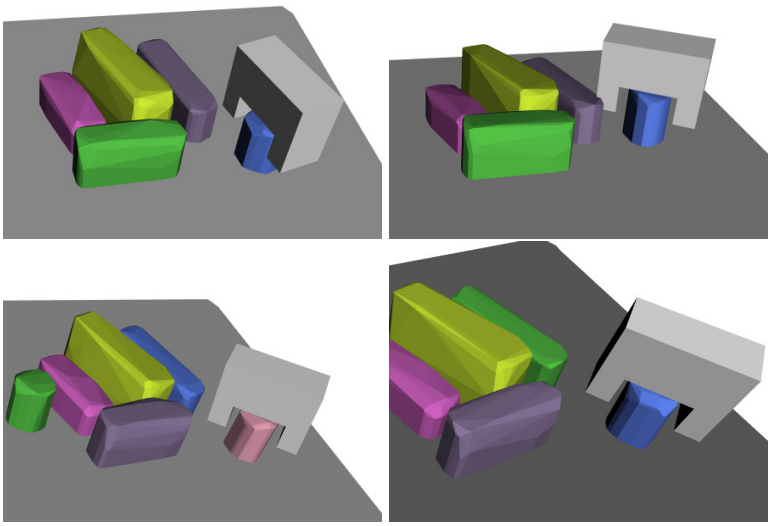
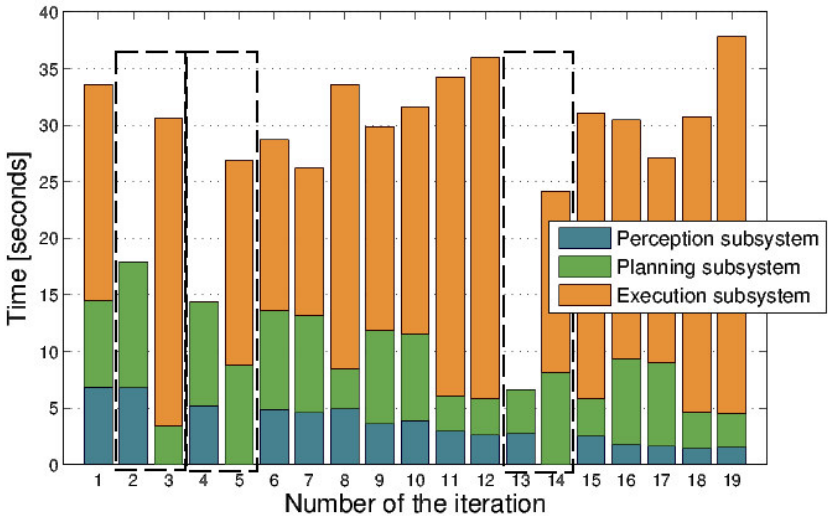


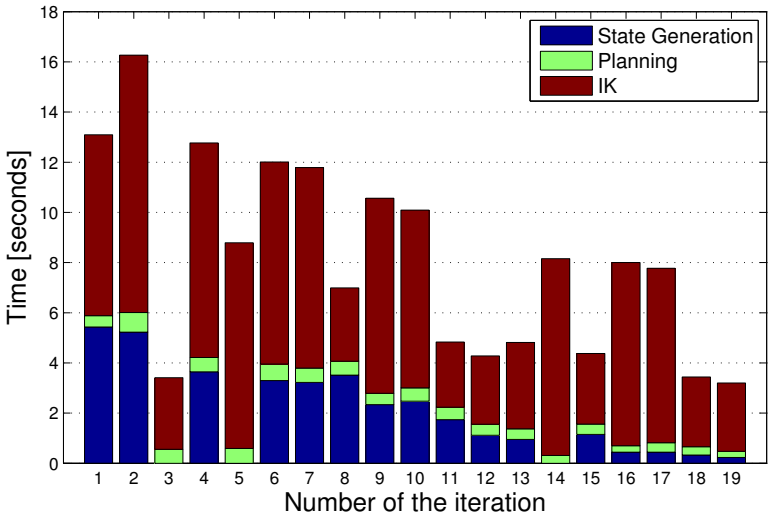
Fig. 7.6: Executed actions to grasp o6 in the experiment depcited in Figure 7.4. From top left, in its initial pose the gripper is too close to the gray box, after pushing the gripper would collide, after pushing again o6 is still too close to the gray. Finally, after an ulterior pushing the gripper in the grasping pose is collision free and sufficiently far from the other objects.

The times devoted to each phase are shown in Fig. 7.7a, where backtracks are represented as black dashed rectangles joining two consecutive iterations. Figure 7.7b concentrates on the three main aspects for planning: state generation, planning and IK evaluation. As it can be seen, IK is the most time consuming operation. The explanation is that each of the pushing and grasping actions are composed by several poses of the end-effector,

and therefore several checks have to be performed. State generation consumes also several seconds, and planning is very fast. These results uphold the benefits of delaying IK computations to evaluate the feasibility of one action only when it is needed in a plan.

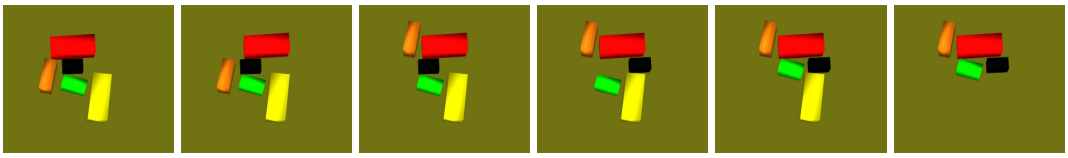


(a) Amount of time spent by the three subsystems.

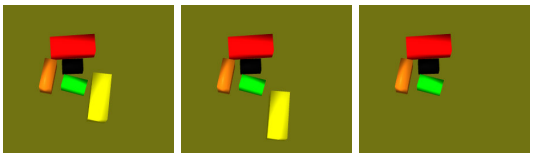


(b) Elapsed times for the state generation, planning and IK.

Fig. 7.7: Elapsed times for each iteration of the experiment in Figure 7.4. (a) elapsed times for the three subsystems. Backtrackings are highlighted using black dashed rectangles. The perception subsystem first filters the point cloud for noise reduction, segments the objects and then generates the state. The planning subsystems computes the plan and the IK of the first action. The execution subsystems just executes the action. (b) elapsed times to generate the state, to get a plan and evaluate the IK of the first action. In the last iteration the robot correctly grasps the last object remaining on the table and the goal is achieved.



(a) Reactive strategy



(b) Proposed system

Fig. 7.8: Experiment to compare the proposed system with the reactive strategy used in [16]. The figures on the left in (a) and (b) are the initial scene.

The third experiment compares the proposed method with an adaptation of the framework presented by Dogar and Srinivasa [16], which is the most similar work to ours to the best of our knowledge. The authors propose a general framework to deal with cluttered scenes, where, if necessary, the objects around the target one are rearranged to make it graspable. Considering that the goal is to clear the whole table, their algorithm behaves as a reactive strategy repeated for each object. We select the target object as the one yielding the less number of objects that would collide with the end-effector if grasped; if there are several candidates, the target object is selected randomly.

We simulated both algorithms in a simulated environment. In simple scenarios both strategies are similar. Differences arise in moderately complex scenes. Figure 7.8 shows one example comparing both solutions for a 5 object scene. In the figure, the lower-right yellow object is the candidate target. Observe that in this case the reactive approach requires 5 actions to grasp it (Fig. 7.8a), while with our approach (Fig. 7.8b) 2 actions suffice.

8. Conclusions

The objectives established have been achieved developing a planning system which is able to solve table clearing tasks with cluttered objects reasoning at a symbolic level. The planning system can choose the best sequences of pushing and grasping actions to solve problems that could be not solved by only considering grasping actions.

The planning system has been inspired by the way humans solve the task and the experiments showed the robot can solve the task with a intelligent sequence of actions that we consider close to the one a human would do.

The system is able to interact with the objects because augmented with a depth sensor that gives a three-dimensional representation of the scene. Two depth sensors are considered: a Microsoft Kinect and a stereo vision system. After a comparison of the two sensors we assert that for the specific task tackled by this thesis the Microsoft Kinect sensor suits better. The developed stereo vision system, although does not suffer for the kind of observed material as Kinect does, it suffers for brightness that may cause saturated images and it is computationally expensive and less robust than Kinect.

The novelty of this thesis regards the tackling of geometrical restrictions within symbolic planning. To do so, a symbolic three-layered system that plans at a deterministic semantic level and accounts for geometrical restrictions is proposed. This is done by handling geometrical constraints both with symbolic predicates and through backtracking. Geometrical constraints are divided into *relational* (collisions between objects or with the robot) and *reachability* (unfeasible actions due to non-solvable IK or path planning failures). *Relational* constraints generate a set of symbolic states that are used by the planner to compute a plan. The evaluation of *reachability* constraints is delayed until the plan has to be executed. This lazy approach allows to accelerate the reasoning process. Thanks to backtracking, unfeasible actions generate new symbolic predicates and replanning produces a new plan.

A simple implementation is presented and evaluated in different scenarios. Experiments show that the strategy of delaying the *reachability* constraints evaluation is beneficial in practical robot applications as it strongly reduces the required computational time. The presented algorithm can handle different strategies, but as we propose to replan after each action, the current implementation only evaluates the first action of the

computed plan. This is a good option to take into account uncertainty when the robot actions are non-deterministic, like the ones used in the experiments. But also has some limitations, for example, pushing objects towards the limits of the robot working space possibly invalidates some future pushing or even grasping actions.

The planning system can be easily adapted to every kind of robotic manipulator and gripper, as it only needs to include the model of the gripper and the inverse kinematic. It can also be easily integrated with more sophisticated grasping methods.

As future work we would like to include the probabilities of the outcomes of the pushing action within predicates, so as to be able to choose between different pushing strategies. Also, one limitation of the presented system is that it considers as errors all types of collisions. Thus, we would like to introduce the collisions as actions' cost accordingly to the kind of collision.

The more restrictive limitation of the system is the segmentation, particularly hard in cluttered environments. A wrong segmentation can make the planner unable to find a feasible plan. A method to contrast this is enhancing the action model with an auxiliary action that allows to grab another image, as done by Hernánzed [23], to provide more information about the objects whether the segmentation is not good enough. Then, to maximize the added value of the new image the point of view should be movable and chosen accordingly to a likelihood function that favours point of views that allow to see previously occluded parts in a similar fashion to the work of Foix et al. [20]. Alternatively, an extra camera can be employed.

Finally, the system can be enhanced by integrating it with the motion planning framework *MoveIt!* [52] in order to execute collision-free trajectories to approach the pushing and grasping poses.

Bibliography

[1] PDDL - The Planning Domain Definition Language. Technical report, CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.

[2] Nichola Abdo, Henrik Kretzschmar, Luciano Spinello, and Cyrill Stachniss. Learning manipulation actions from a few demonstrations. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1268–1275. IEEE, 2013.

[3] Anubhav Agarwal, CV Jawahar, and PJ Narayanan. A survey of planar homography estimation techniques. *Centre for Visual Information Technology, Tech. Rep. IIIT/TR/2005/12*, 2005.

[4] Faraj Alhwarin, Alexander Ferrein, and Ingrid Scholl. Ir stereo kinect: improving depth images by combining structured light with ir stereo. In *Pacific Rim International Conference on Artificial Intelligence*, pages 409–421. Springer, 2014.

[5] Julien Bidot, Lars Karlsson, Fabien Lagriffoul, and Alessandro Saffiotti. Geometric backtracking for combined task and motion planning in robotic systems. *Artificial Intelligence*, 2015.

[6] Stan Birchfield and Carlo Tomasi. A pixel dissimilarity measure that is insensitive to image sampling. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 20(4):401–406, 1998.

[7] Jean-Yves Bouguet. Camera calibration tool-box for matlab. *http://www. vision. caltech. edu/bouguetj/calib_doc/*, 2002.

[8] Peter Brook, Matei Ciocarlie, and Kaijen Hsiao. Collaborative grasp planning with multiple object representations. In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*, pages 2851–2858. IEEE, 2011.

[9] Rui Coelho and Alexandre Bernardino. Planning push and grasp actions: Experiments on the icub robot.

[10] Nikolaus Correll, Kostas E. Bekris, Dmitry Berenson, Oliver Brock, Albert Causo, Kris Hauser, Kei Okada, Alberto Rodriguez, Joseph M. Romano, and Peter R. Wurman. Lessons from the amazon picking challenge. *CoRR*, abs/1601.05484, 2016.

[11] N. Covallero and G. Alenyà. Grasping novel objects. Technical Report IRI-TR-16-01, Institut de Robòtica i Informàtica Industrial, CSIC-UPC, 2016.

[12] Nicola Covallero. Task planning for table clearing of cluttered objects. Master's thesis, Universitat Politècnica de Catalunya, 2016.

[13] Nicola Covallero, David Martínez, Guillem Alenyà, and Carme Torras. Planning clearing actions in cluttered scenes by phasing in geometrical constraints. In *Proceedings of the* 20^{th} *International Conference of the Catalan Association of Artificial Intelligence*, 2017.

[14] Richard Dearden and Chris Burbridge. Manipulation planning using learned symbolic state abstractions. *Robotics and Autonomous Systems*, 62(3):355 – 365, 2014. Advances in Autonomous Robotics — Selected extended papers of the joint 2012 {TAROS} Conference and the {FIRA} RoboWorld Congress, Bristol, {UK}.

[15] Paul E Debevec and Jitendra Malik. Recovering high dynamic range radiance maps from photographs. In *ACM SIGGRAPH 2008 classes*, page 31. ACM, 2008.

[16] Mehmet Dogar and Siddhartha Srinivasa. A framework for push-grasping in clutter. In Nick Roy Hugh Durrant-Whyte and Pieter Abbeel, editors, *Robotics: Science and Systems VII*. MIT Press, July 2011.

[17] Mustafa Ersen, Melodi Deniz Ozturk, Mehmet Biberci, Sanem Sariel, and Hulya Yalcin. Scene interpretation for lifelong robot learning. In *Proceedings of the 9th international workshop on cognitive robotics (CogRob 2014) held in conjunction with ECAI-2014*, 2014.

[18] David Fischinger, Astrid Weiss, and Markus Vincze. Learning grasps with topographic features. volume 34, pages 1167–1194, 2015.

[19] Martin A Fischler and Robert C Bolles. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 24(6):381–395, 1981.

[20] Sergi Foix, Guillem Alenya, and Carme Torras. 3d sensor planning framework for leaf probing. In *Intelligent Robots and Systems (IROS), 2015 IEEE/RSJ International Conference on*, pages 6501–6506. IEEE, 2015.

[21] Malte Helmert. The fast downward planning system. *J. Artif. Intell. Res.(JAIR)*, 26:191–246, 2006.

[22] Tucker Hermans, James M. Rehg, and Aaron F. Bobick. Guided pushing for object singulation. In *IROS*, pages 4783–4790. IEEE, 2012.

[23] Alejandro Suárez Hernández, Carme Torras Genıs, Guillem Alenya Ribas, and Javier Béjar Alonso. Integration of task and motion planning for robotics. 2016.

[24] Itseez. Open source computer vision library. https://github.com/itseez/opencv, 2017.

[25] Ian Jolliffe. *Principal component analysis*. Wiley Online Library, 2002.

[26] Adrian Kaehler and Gary Bradski. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. " O'Reilly Media, Inc.", 2016.

[27] Leslie Pack Kaelbling and Tomás Lozano-Pérez. Unifying perception, estimation and action for mobile manipulation via belief space planning. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 2952–2959. IEEE, 2012.

[28] Dov Katz. *Interactive perception of articulated objects for autonomous manipulation*. University of Massachusetts Amherst, 2011.

[29] Dov Katz, Moslem Kazemi, J. Andrew (Drew) Bagnell, and Anthony (Tony) Stentz . Clearing a pile of unknown objects using interactive perception. In *Proceedings of IEEE International Conference on Robotics and Automation*, March 2013.

[30] Dov Katz, Arun Venkatraman, Moslem Kazemi, J Andrew Bagnell, and Anthony Stentz. Perceiving, learning, and exploiting object affordances for autonomous pile manipulation. *Autonomous Robots*, 37(4):369–382, 2014.

[31] Thomas Keller and Patrick Eyerich. Prost: Probabilistic planning based on uct. In *ICAPS*, 2012.

[32] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE.

[33] Andrey Kolobov, Mausam, and Daniel S Weld. Lrtdp vs. uct for online probabilistic planning. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[34] F. Lagriffoul, D. Dimitrov, A. Saffiotti, and L. Karlsson. Constraint propagation on interval bounds for dealing with geometric backtracking. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 957–964, 2012.

[35] Steven M. LaValle. *Planning Algorithms*. Cambridge University Press, May 2006.

[36] Iain Little, Sylvie Thiebaux, et al. Probabilistic planning vs. replanning. In *ICAPS Workshop on IPC: Past, Present and Future*, 2007.

[37] T. Lozano-Perez and L. P. Kaelbling. A constraint-based method for solving sequential manipulation planning problems. In *IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 3684–3691, 2014.

[38] Bhaskara Marthi, Stuart J Russell, and Jason Wolfe. Angelic semantics for high-level actions. In *ICAPS*, pages 232–239, 2007.

[39] D. Martínez, G. Alenyà, and C. Torras. Relational reinforcement learning with guided demonstrations. *Artificial Intelligence*, 247:295–312, 2017.

[40] David Martínez, Guillem Alenya, and Carme Torras. Planning robot manipulation to clean planar surfaces. *Engineering Applications of Artificial Intelligence*, 39:23–32, 2015.

[41] Lorenz Mösenlechner and Michael Beetz. Using physics- and sensor-based simulation for high-fidelity temporal projection of realistic robot behavior. In *AIPS*, 2009.

[42] Dana Nau, Yue Cao, Amnon Lotem, and Hector Munoz-Avila. Shop: Simple hierarchical ordered planner. In *Proceedings of the 16th international joint conference*

on Artificial intelligence-Volume 2, pages 968–973. Morgan Kaufmann Publishers Inc., 1999.

[43] Melodi Ozturk, Mustafa Ersen, Melis Kapotoglu, Cagatay Koc, Sanem Sariel-Talay, and Hulya Yalcin. Scene interpretation for self-aware cognitive robots. 2014.

[44] Jia Pan, Sachin Chitta, and Dinesh Manocha. Fcl: A general purpose library for collision and proximity queries. In *Robotics and Automation (ICRA), 2012 IEEE International Conference on*, pages 3859–3866. IEEE, 2012.

[45] Jeremie Papon, Alexey Abramov, Markus Schoeler, and Florentin Wörgötter. Voxel cloud connectivity segmentation - supervoxels for point clouds. In *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, Portland, Oregon, June 22-27 2013.

[46] Morgan Quigley, Ken Conley, Brian P. Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, and Andrew Y. Ng. Ros: an open-source robot operating system. In *ICRA Workshop on Open Source Software*, 2009.

[47] Stuart J. Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 2 edition, 2003.

[48] Radu Bogdan Rusu and Steve Cousins. 3d is here: Point cloud library (pcl). In *International Conference on Robotics and Automation*, Shanghai, China, 2011 2011.

[49] Radu Bogdan Rusu, Zoltan Csaba Marton, Nico Blodow, Mihai Dolha, and Michael Beetz. Towards 3d point cloud based object maps for household environments. *Robotics and Autonomous Systems*, 56(11):927–941, 2008.

[50] Ravishankar Somasundaram. *Git: Version control for everyone*. Packt Publishing Ltd, 2013.

[51] S. C. Stein, M. Schoeler, J. Papon, and F. Woergoetter. Object partitioning using local convexity. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[52] Ioan A. Sucan and Sachin Chitta. Moveit! [Online] Available:http://moveit.ros.org.

[53] Andreas ten Pas and Robert Platt. Using geometry to detect grasp poses in 3d point clouds. In *International Symposium on Robotics Research (ISRR)*, September 2015.

[54] Nikolaus Vahrenkamp, Tamim Asfour, and Rudiger Dillmann. Robot placement based on reachability inversion. In *Robotics and Automation (ICRA), 2013 IEEE International Conference on*, pages 1970–1975. IEEE, 2013.

[55] Dimitri Van Heesch. Doxygen: Source code documentation generator tool. *URL: http://www. doxygen. org*, 2008.

[56] Gang Xu and Zhengyou Zhang. *Epipolar geometry in stereo, motion and object recognition: a unified approach*, volume 6. Springer Science & Business Media, 2013.

[57] Zhengyou Zhang. A flexible new technique for camera calibration. *IEEE Transactions on pattern analysis and machine intelligence*, 22(11):1330–1334, 2000.

[58] Zhengyou Zhang. Microsoft kinect sensor and its effect. *IEEE multimedia*, 19(2):4–10, 2012.

Appendices

A. Basler Cameras

The Basler ACE USB 3.0 cameras used are the acA2500-14um and acA2500-14uc models. The two models differ only for the color scale they capture the image, the *uc* model captures a RGB image while the *um* model captures a monochrome image. In Table A.1 the main specifications of the models are reported.

Tab. A.1: Basler ACE USB 3.0

Specification	**acA2500-14um/uc**
Resolution	um: 2592×1944 uc: 2590×1942
Sensor Type	Aptina MT9P031, CMOS
Optical Size	1/2.5”
Pixel size	$2.2\mu m \times 2.2\mu m$
Fps	14

The lenses used are ultra low-distorsion lenses with a focal length of $16mm$, f-number[1] 2.0, for sensor sizes up to 2/3".

Fig. A.1: Basler ACE USB 3.0 acA2500-14um/uc without and with lence.

[1]The f-number F is the ratio between of focal length to the diameter D of of the entrance pupil such that $F = \frac{f}{D}$