Pick and place application using ROS and Moveit!

authors

Alessio Saccuti

Vito Filomeno

UNIVERSITÀ
DI PARMA

DEPARTMENT OF COMPUTER ENGINEERING

UNIVERSITY OF PARMA

AUGUST 2020

Abstract

This project aims to develop a pick and place application using simulation environment as Gazebo and Rviz, together with Moveit!. The goal is to simulate an industrial environment, with multiple input location and output location for targets.

The entire project can be divided in two parts: perception and action. In the perception part we used a Kinect camera for recognizing object on a table, using computer vision algorithms to find their coordinates and send it to the manipulator. In the action part the manipulator takes the objects coordinates and implements motion planning algorithms to grasp them and place them arrayed in a row on another table.

The whole system appears to work well. Moveit! often reports error of unsuccessful motion planning but this can be avoided giving it more details of the scene and targets.

TABLE OF CONTENTS

Chapter 1

Introduction

In the future, in particular for 4.0 industries, robotic arms will be an essential part of the production process. Pick and place is the ability of an arm to pick up objects and place them in particular position in autonomous way, obviously the automation requirements of the future will imply an increasing number of robotic arms in industries as shown in figure 1.1.

Figure 1.1: Assembling pipeline in automotive industries

A manipulator can be designed with anthropomorphic style or not, with more or less number of joints and actuators. In general a robotic arm can perform actions unfeasible for humans, this is a big advantage. The environment around the arm must be studied in all details because the arm can't reach all the possible target in the space. Moreover if there are humans at work nearby manipulators, the risk of dangerous movement must be considered.

This project can't be implemented in a practical way, it will be based on simulation environments. The scenario proposed is just a simple world without many details and obstacles. Each chapter of this paper will explore a particular phase: simulation, perception and execution. We will include portion of code related to the main points of the project, the whole code can be found in **this repository**. This paper gives an overview of arm-ROS-Moveit! interaction, but it can be used as a start point for future works!

Chapter 2

Simulation

Simulation is the core part of this project, unfortunately ROS doesn't provide a complete simulation environment in which we can simulate perception and manipulation simultaneously. To cope with this problem the simulation build up on two different parts, each part uses a different environment and communicates with a project's phase:

- **Gazebo → Perception**
- **Rviz → Manipulation**

The main problem was that with Gazebo is possible to insert a Kinect camera that observe the objects in the scene, but is not possible to use motion planning with Moveit. In Rviz, instead, is possible to use Moveit features, but unfeasible to do perception.

2.1 Gazebo simulation

We built on Gazebo a world with 4 type of objects: arm, kinect, tables and targets. Some of them have unique properties, as shown below. The built scene can be see in figure 2.1.

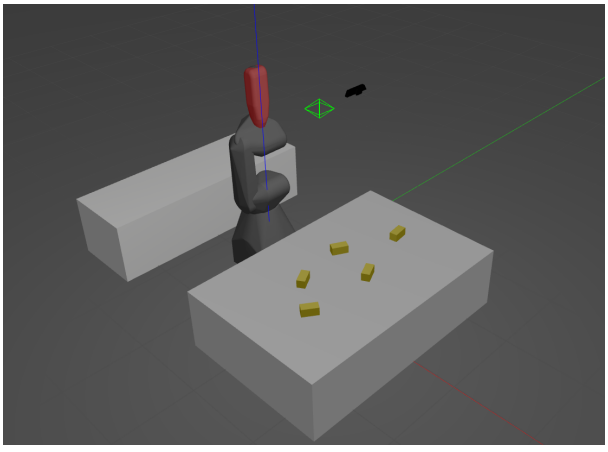


Figure 2.1: Simulation scene in Gazebo

2.1.1 Kinect

The scene is an XML file generated using the gazebo editor. To add the kinect as an active sensor, that publish informations over some topics, we add a plugin into the file:

```
...
<plugin name='camera_plugin' filename='libgazebo_ros_openni_kinect.so'>
            <baseline>0.2</baseline>
            <alwaysOn>1</alwaysOn>
            <updateRate>0.0</updateRate>
            <cameraName>camera_ir</cameraName>
            <imageTopicName>/camera/color/image_raw</imageTopicName>
            <cameraInfoTopicName>/camera/color/camera_info</cameraInfoTopicName>
            <depthImageTopicName>/camera/depth/image_raw</depthImageTopicName>
            <depthImageCameraInfoTopicName>/camera/depth/camera_info</
    depthImageCameraInfoTopicName>
            <pointCloudTopicName>/camera/depth/points</pointCloudTopicName>
            <frameName>camera_link</frameName>
            <pointCloudCutoff>0.5</pointCloudCutoff>
            <pointCloudCutoffMax>3.0</pointCloudCutoffMax>
            <distortionK1>0</distortionK1>
            <distortionK2>0</distortionK2>
            <distortionK3>0</distortionK3>
            <distortionT1>0</distortionT1>
            <distortionT2>0</distortionT2>
            <CxPrime>0</CxPrime>
            <Cx>0</Cx>
            <Cy>0</Cy>
            <focalLength>0</focalLength>
            <hackBaseline>0</hackBaseline>
</plugin>
...
```

What we care about this plugin is where the data are published:

- **depth point cloud**: `/camera/depth/points`
- **depth image**: `/camera/depth/image_raw`

2.1.2 Targets spawn

To random spawn multiple targets we added to the world a XML portion of code:

```
...
<population name="population">
      <model name="target">
        <include>
          <static>false</static>
          <uri>model://target</uri>
        </include>
      </model>
      <pose>1.2 0 0.855 0 0 0</pose>
      <box>
        <size>1 2.0 0.002</size>
      </box>
      <model_count>5</model_count>
      <distribution>
        <type>uniform</type>
      </distribution>
    </population>
...
```

It allows to uniform spawn target in a box shaped range. To give a random orientation to the boxes we use information on published topics by the gazebo simulation, automatic

generated:

- `/gazebo/model_states` allows to get the objects pose in the scene.
- `/gazebo/set_model_state` allows to set new poses.

The dedicated script `spawn.cpp` uses these topics:

```
ros::NodeHandle ps;
ros::Subscriber pose_sub = ps.subscribe("/gazebo/model_states", 1000, getModelStates);
...
ros::NodeHandle n;
ros::ServiceClient client = n.serviceClient<gazebo_msgs::SetModelState>("/gazebo/
    set_model_state");
...
gazebo_msgs::SetModelState objstate;
/* manipulate objstate from actual model state */
...
client.call(objstate);
```

2.2 Rviz simulation

The actuation of the arm can be directly simulated in Rviz, Moveit! offers the planning interface API that allow to add objects in the scene and plan a sequence of states for the robotic arm. Each object is added in an initialization phase by the arm's script `application.cpp`. The final scene can be seen in the figure 2.2.

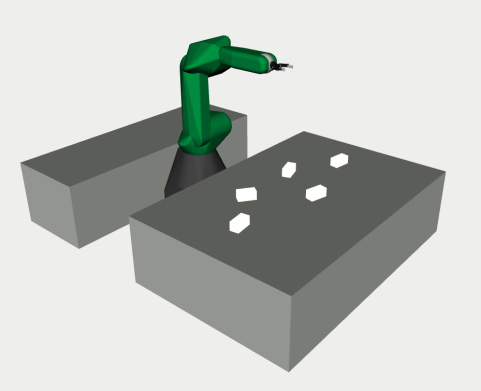


Figure 2.2: Simulation scene in Rviz

The Moveit! simulation can be easily generated using an integrated tool called **setup assistant**, as shown in figure 2.3 and in the appendix A. The used end effector is the **Robotiq 2F-140** (figure 2.4) and the used arm is the **Fanuc cr135ia** (figure 2.5) because in the future the robotic lab of our univerity will buy a similar manipulator.

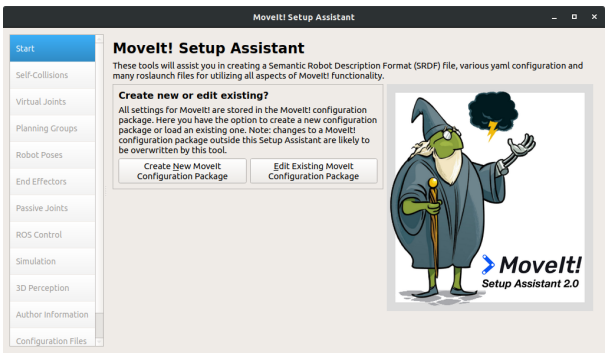


Figure 2.3: Setup assistant initial screen

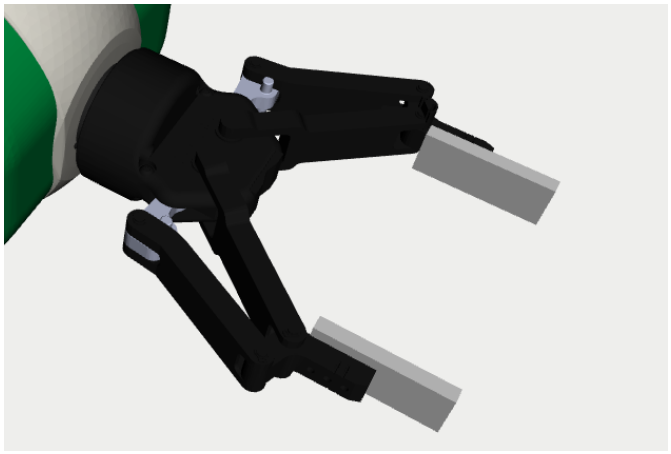


Figure 2.4: Robotiq 2f-140

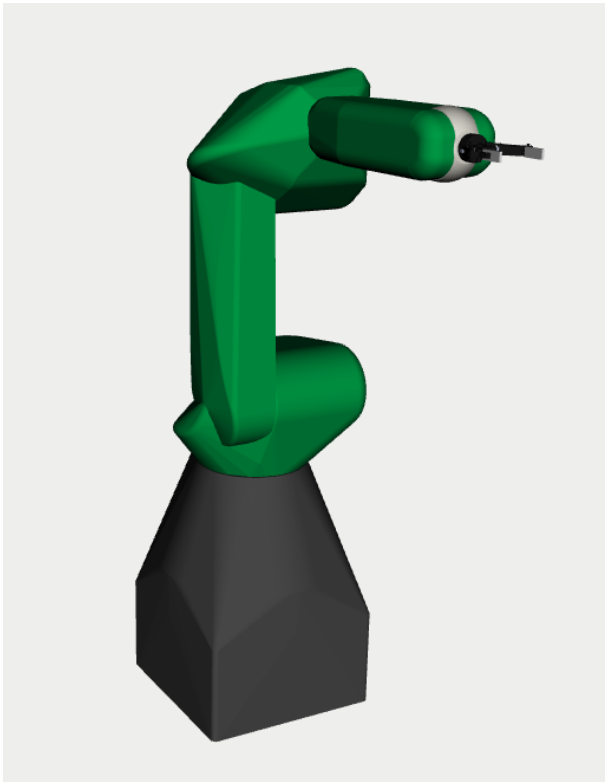


Figure 2.5: Fanuc cr35ia

Chapter 3

Perception

The first step of the simulation is the perception phase, this part is completely focused on Gazebo environment. The goal is to take the 2D image and point cloud provided by Kinect and process them in order to obtain pose and orientation of the objects in the scene. Once we get this information, compress them in a quaternion and publish it on a specific topic.

3.1 Input and Output

First of all we need to obtain the image published by Kinect, to do that we used the `image_transport` ROS package which allow us to create a subscriber who receives images.

```
#include <image_transport/image_transport.h>

ros::NodeHandle nh;
image_transport::ImageTransport it(nh);
image_transport::Subscriber image_sub;
image_sub = it.subscribe("/camera/color/image_raw", 1, &ImageConverter::imageCb,
this);
```

Each time an image is received on `/camera/color/image_raw` topic the `imageCb` callback is executed. In this function the ROS image message is converted into OpenCV image using the `CV_bridge` package.

```
#include <cv_bridge/cv_bridge.h>

...

void imageCb(const sensor_msgs::ImageConstPtr& msg)
{
cv_bridge::CvImagePtr cv_ptr;
try
{
  cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::RGB8);
}
catch (cv_bridge::Exception& e)
{
  ROS_ERROR("cv_bridge exception: %s", e.what());
  return;
}

cv::cvtColor(cv_ptr->image, gray, cv::COLOR_RGB2GRAY);
```

```
19  if (this->cloud != NULL)
20    findCoordinates(gray);
21  }
```

In order to obtain the point cloud we only need to create a normal subscriber on `PointCloud 2` topic, and a callback that stores the updated point cloud in memory.

```
1    #include <sensor_msgs/PointCloud2.h>
2
3    ...
4    ros::Subscriber point_sub;
5    point_sub = nh.subscribe<sensor_msgs::PointCloud2>("/camera/depth/points", 1, &
     ImageConverter::processCloud, this);
6
7    void processCloud(const sensor_msgs::PointCloud2ConstPtr& cloud)
8  {
9    this->cloud = cloud;
10 }
```

As we'll see later the output of the perception phase is an array of poses $[x, y, z, w]$ one for each object. In order for the robot to manipulate them, we need to share this information. We create a publisher of `PoseArray` message on `/poses` topic.

```
1    #include <geometry_msgs/PoseArray.h>
2
3    ...
4
5    ros::Publisher pose_pub;
6    pose_pub = nh.advertise<geometry_msgs::PoseArray>("/poses",1);
```

3.2 Image Preprocessing

Once we get raw image from camera, we need to process it to ensure good poses extraction, in `ImagePreProcessor.cpp` file we have defined a class for this purpose. First we need to separate objects from background, we take advantage from the fact that the table's color is uniform and light colored. Our approach consists in finding a range of intensity values of the background pixels that can be seen with the help of an histogram:

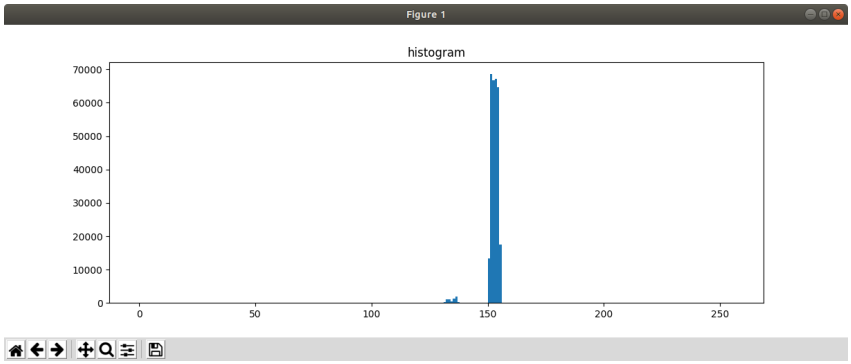
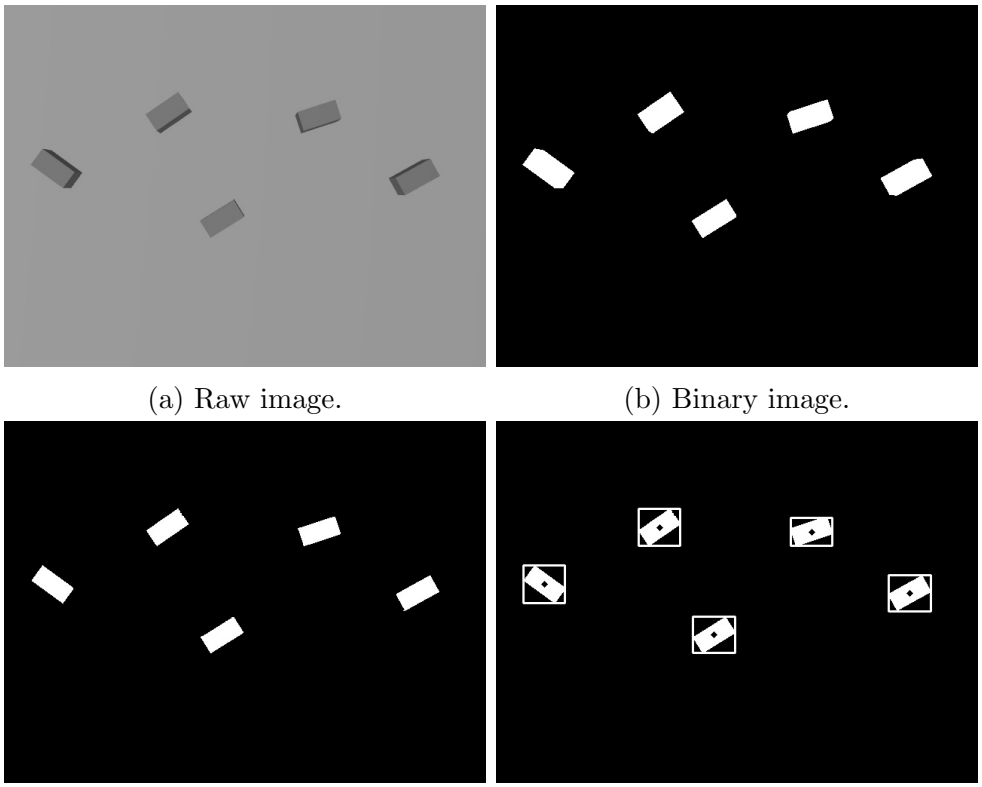


Figure 3.1: Image histogram.

We'll use them in the threshold operation to obtain a binary image with value 0 for the background and value 255 for objects, as shown in figure 3.2 b.



(a) Raw image.

(b) Binary image.

(c) Binary image without lateral surfaces.

(d) Binary image with centers and bounding boxes.

Figure 3.2: Sequence of operations leading to finding the centers.

```
#include <opencv2/opencv.hpp>
#include "ImagePreProcessor.cpp"
ImagePreProcessor pr;
...
int threshold = pr.find_peak(gray, 0);
cv::threshold(img, thres, threshold-10, 255,cv::THRESH_BINARY_INV);
```

Object's centroids can be found using `findCentroids` method. It takes as input a canny filter of 3.2 b, and returns a structure containing centers and bounding boxes for each object.

```
struct obj
{
 vector<cv::Point> centroids;
 vector<cv::Rect> boundRects;
};
...
cv::Canny(img, canny, 100, 10);
obj objects;
objects = pr.findCentroids(canny);
```

At this stage we got a good approximation of object position, but not good enough. Due to prospective, an object which is not exactly under the camera shows also its lateral face in the image, this surface is counted as contour and can lead to errors in center calculation up to 5mm, as shown in figure 3.3.

A good approximation to overcome this issue is to consider only the top surface of the

object in centroids calculation. The `filter` method was implemented for this scope, it takes gray image, binary image and bounding boxes as input and then perform the following steps for each object in the image:

1. Using boxes and binary image it creates a filter useful to isolate a single object in the gray image.
2. Applies the `find_peak` method to find average intensity, and removes darker pixels correspond to lateral surfaces.
3. Stores this found surface in a new image in the same position.
4. Iterates for each objects to obtain image shown in 3.2 c.

Now applying again `findCentroids` on this new image we get a more accurate centers coordinates with an error lower than $1\,mm$. The final result of this process can be seen in figure 3.2d.

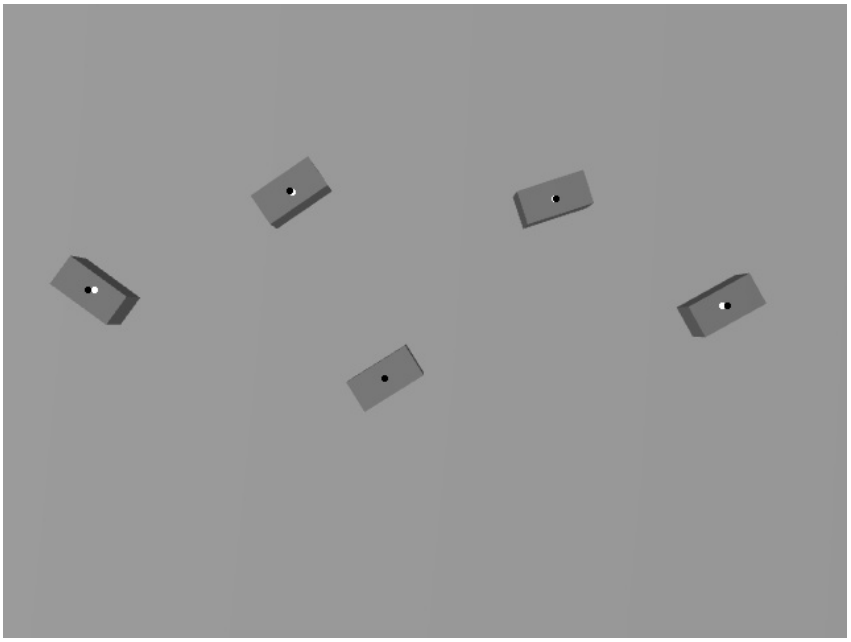


Figure 3.3: Centroids before (white) and after (black) lateral surface removal. As we can see this leads to a better approximation of the center, especially when the objects are close to the edge.

3.3 Pose and orientation

So far we found objects centroids in two-dimensional image, we need to convert them in global 3D coordinates and find the yaw angle for each object. Kinect is a depth camera, so it returns a depth image that is treated as a pointcloud, this greatly facilitates the task of finding the points in the global coordinates starting from pixels of the two-dimensional image. In fact there is a 1 to 1 correlation between pixels and points of the point cloud. The point cloud is managed as a byte array, so we can find coordinates starting from center pixel (u,v) as shown below:

```
1  int arrayPosition = v*this->cloud->row_step + u*this->cloud->point_step;
2
3   // compute position in array where x,y,z data start
4   int arrayPosX = arrayPosition + this->cloud->fields[0].offset; // X has an offset
    of 0
5   int arrayPosY = arrayPosition + this->cloud->fields[1].offset; // Y has an offset
    of 4
6   int arrayPosZ = arrayPosition + this->cloud->fields[2].offset; // Z has an offset
    of 8
```

This piece of code allows us to find the placeholders where information is stored in the array, and allows us to access them to get the coordinates. After that we need to convert coordinates from camera frame to global frame. Since Kinect's coordinates in global frame are $x = 0, y = 1.5, z = 3$ we don't need any traslation matrix. With a little effort, looking at figure 2.1, we can find the relationship between frames shown in rows 6-8 below.

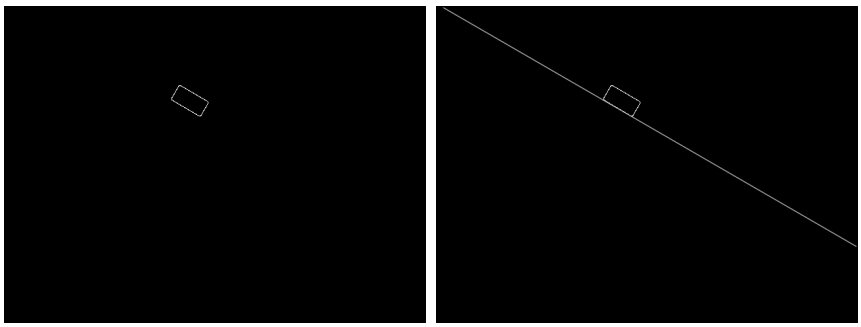
```
1   memcpy(&X, &this->cloud->data[arrayPosX], sizeof(float));
2   memcpy(&Y, &this->cloud->data[arrayPosY], sizeof(float));
3   memcpy(&Z, &this->cloud->data[arrayPosZ], sizeof(float));
4
5   //float x_camera = 1.5;
6   Xg = Y+this->x_camera; // X in global frame
7   Yg = X; // Y in global frame
8   Zg = z_surface; //static value, objects are all the same size
```

In order to complete the poses, we need to find the objects angle. The algorithm in charge of doing this is defined in `findOrientation` function. It needs two parameters to perform:

- canny filter of figure 3.2c.
- bounding boxes for all objects.

The algorithm iterates through each object and perform the following steps:

1. Use bounding box to isolate the object in the image.
2. Computes Hough transform to find the lines passing through the lateral side of the object. Lines are expressed in ρ, θ coordinates referred to the upper left corner of image.
3. Use θ angle to find orientation in global frame.



(a) Isolated object. (b) Result of Hough transform.

Figure 3.4: Find angle algorithm.

Chapter 4

Manipulation

The execution phase is possible thanks to MoveIt! A documentation that explain how it works can be found **here**. What we used to build the application node is:

- **Move group interface**: used to attuate the motion.
- **Planning scene interface**: allows to generate a scene to manipulate.

The move group interface offers high level method, but they aren't suitable for our project. We had to generate this methods ourself declaring for example mid steps for the motion. In general our application can be seen as a finite number of steps:

1. Scene generation
2. Target reception
3. Pick
4. Place

4.1 Scene generation

When the code is launched it spawns an empty Rviz scene with only the arm, the two tables and the objects to manipulate are added separately:

- `arm::generate_scene()` adds the two tables using coordinates known a priori.
- `arm::generate_objects()` adds every single object that we will use as target getting information from the gazebo scene, using the topic introduced in the simulation chapter `model_states`.

Each method generates a **collision object**, which can be mainly described using position and translation quantities as quaternions. For example the support table on which we spawn the target can be added as shown:

```
1    moveit_msgs::CollisionObject collision_object;
2    collision_object.header.frame_id = move_group->getPlanningFrame();
3
4    /* ORIGIN TABLE */
5    collision_object.id = "origin_table";
6    shape_msgs::SolidPrimitive primitive;
7    primitive.type = primitive.BOX;
```

```
    primitive.dimensions.resize(3);
    primitive.dimensions[0] = 2;
    primitive.dimensions[1] = 3;
    primitive.dimensions[2] = 0.8;

    geometry_msgs::Pose origin_table;

    origin_table.orientation.w = 1.0;
    origin_table.position.x = 1.5;
    origin_table.position.y = 0;
    origin_table.position.z = 0.4;

    collision_object.primitives.push_back(primitive);
    collision_object.primitive_poses.push_back(origin_table);
    collision_object.operation = collision_object.ADD;

    collision_objects.push_back(collision_object);
    planning_scene_interface.addCollisionObjects(collision_objects);
```

We can also explicit rotation values different from the default, for this table it's not necessary. `collision_objects` is a private vector of the application class, it can store different objects that can finally be added to the scene using `planning_scene_interface.addCollisionObjects(collision_objects)`.

4.2 Target reception

The targets can be taken directly from a topic published by the perception node using the application method `arm::set_targets(std::string topic)`:

```
ros::NodeHandle ps;
ros::Subscriber pose_sub = ps.subscribe(topic, 10, &cr35ia::getPoses, this);

std::cout << "waiting for targets.." << std::endl;
while (detected_poses == false) usleep(500000);

ps.shutdown();
```

Where the callback is:

```
/* callback kinect topic */
void getPoses(const geometry_msgs::PoseArray &msgs)
{
    pose_array = msgs;
    detected_poses = true;
}
```

The main problem was to associate each detected poses to an object in the scene, we have done it using a simple data association based on a pose error valutation:

```
/* data association */
...
for (int j = 0; j < model.pose.size(); j++)
{
  int pos = 0;
  if (model.name[j].find("target", pos) != std::string::npos)
  {
    for (int i = 0; i < pose_array.poses.size(); i++)
    {
```

```
      double err_x = abs(model.pose[j].position.x - pose_array.poses[i].position.x);
      double err_y = abs(model.pose[j].position.y - pose_array.poses[i].position.y);
      double err_z = abs(model.pose[j].position.z - pose_array.poses[i].position.z);

      if (err_x < 0.1 && err_y < 0.1 && err_z < 0.1)
      {
        ...
        targets.push_back(pose_array.poses[i]);
        ...
        break;
      }
    }
  }
}
```

4.3 Pick

Both pick and place operations needs a multiple-step configuration. The pick method that can be found in the move group interface can take as input a vector of **grasp**. As can be seen in the documentation, we had to explicit:

- **pre grasp approach**: pose before picking the target, approach direction and how much to approach toward the target.
- **grasp**: pose to pick the target.
- **post grasp retreat**: pose after picked the target, retreat direction and how much to retreat.

For this project and the related scene we configured that the arm will take all the targets in a vertical direction. All the possible grasps rotation, considering the avilable joints, can be generated using as input the yaw of the target given by perception node:

```
for (int k = 1; k <= 2; k++)
    {
      for (int j = 1; j <= 2; j++)
      {
        tf::Quaternion orientation;
        quaternionMsgToTF(targets[target_id].orientation, orientation);
        tf::Matrix3x3 matrix(orientation);
        double yaw, pitch, roll;
        matrix.getEulerYPR(yaw, pitch, roll);
        orientation.setRPY(0, pow(-1, j) * M_PI / 2, pow(-1, k) * M_PI / 2 - yaw);
        quaternionTFToMsg(orientation, grasp.grasp_pose.pose.orientation);
        grasps.push_back(grasp);
      }
    }
```

Then if a successful sequence of state for the motion is found, the target can be picked up using: `move_group->pick(std::to_string(target_id), grasps)`.

4.4 Place

As the pick method needs grasps information, the place method needs place location informations divided into:

- **pre place location approach**: how to approach the place location.
- **place location**: where to place the target.
- **post place location retreat**: how to retreat after placed the target.

We generate multiple random place position on a support table:

```
/* generate different poses for placing */
    for (int i = 0; i < 1000; i++)
    {
      place_location.place_pose.pose.position.x = -1;
      place_location.place_pose.pose.position.y = fRand(-1.24, 1.24);
      place_location.place_pose.pose.position.z = 0.85;
      places.push_back(place_location);
    }
```

The target can be placed using:

```
move_group->setSupportSurfaceName("place_location");
move_group->place(std::to_string(target_id), places);
```

4.5 End effector grasp and place location

Both pick and place methods requires to set the opening and the closing of the gripper. It can be done setting the finger joints value with `openGripper(bool pick) and closedGripper (bool pick)` functions, which uses:

```
grasp.pre_grasp_posture.points[0].positions[0] = X;
...
grasp.grasp_posture.points[0].positions[0] = Y;
...
place_location.post_place_posture.points[0].positions[0] = Z;
...
```

Where x, y, z mean that:

- $x|y|z \rightarrow 0.699$ the joints of the gripper are set in a closed pose.
- $x|y|z \rightarrow 0$ the joints of the gripper are set in an opened pose.

Chapter 5

Conclusion

The final system works well for the implemented scenario. Sometimes Moveit! reports error of unsuccessful motion planning but often they're related to unreachable objects in the space. An example of execution can be seen in this video. Possible extensions and future related works can be:

- **Try different configuration**: by default Moveit! motion planning uses RTT algorithm to find a sequence of state towards the goal state, if it fails to find a solution is possible to change it and change the planning time:

```
moveit::planning_interface::MoveGroupInterface::setPlanningTime(double seconds)
moveit::planning_interface::MoveGroupInterface::setPlannerId(const std::string &
    planner_id)

/* ## Possible planner id ##
    - SBLkConfigDefault
    - ESTkConfigDefault
    - LBKPIECEkConfigDefault
    - BKPIECEkConfigDefault
    - KPIECEkConfigDefault
    - RRTkConfigDefault
    - RRTConnectkConfigDefault
    - RRTstarkConfigDefault
    - TRRTkConfigDefault
    - PRMkConfigDefault
    - PRMstarkConfigDefault */
```

- **Hardware execution**: MoveIt! setup assistant offers a setup panel if physical hardware is available, as shown in figure 5.1.

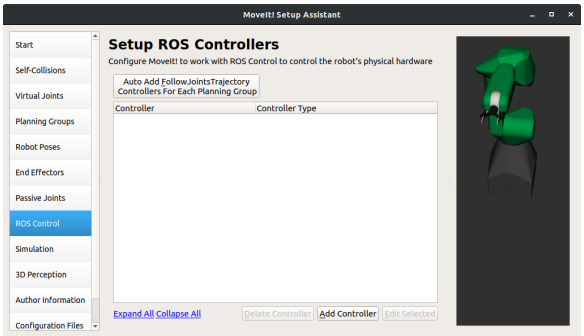


Figure 5.1: S.A. hardware's window

- **Add octomap to Rviz**: using pointclouds is possible to generate an octomap that better characterizes the C space of the scene. An example can be seen in figure 5.2.

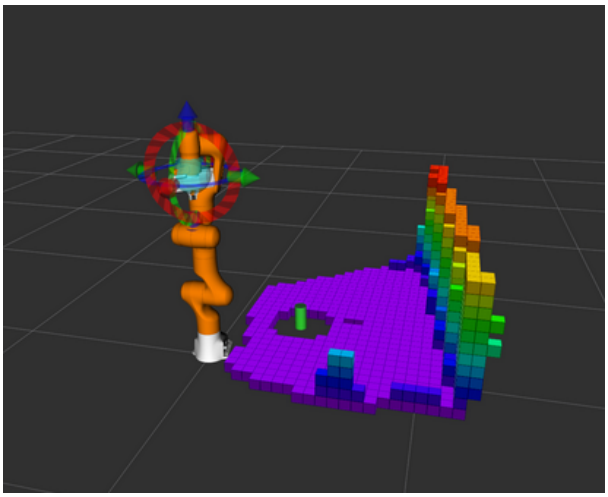


Figure 5.2: Octomap example

Appendix A

Setup assistant configuration

MoveIt! setup assistant is a tool to configure a simulation node. In the following we'll show which part of this tool we used for our project. It can be launched typing in a terminal: `roslaunch moveit_setup_assistant setup_assistant.launch`. The first screen, as shown in the simulation chapter, will look like:

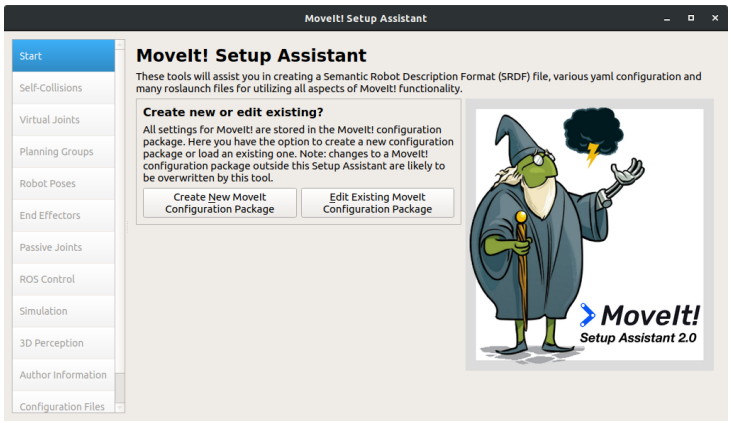
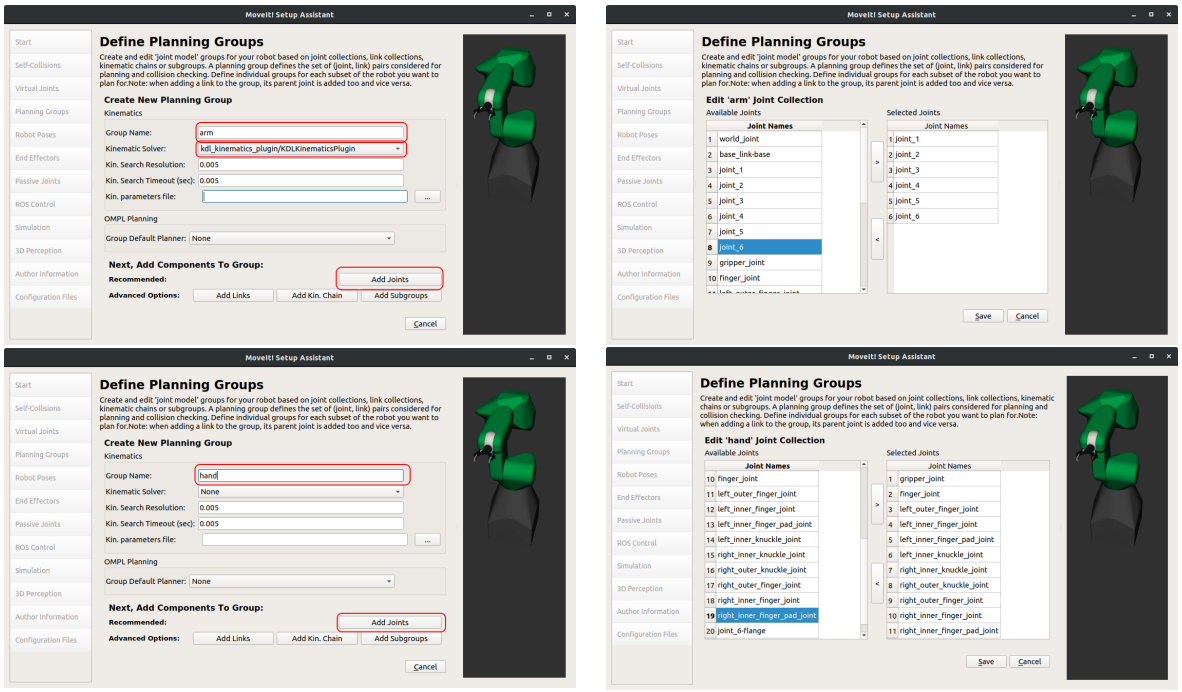


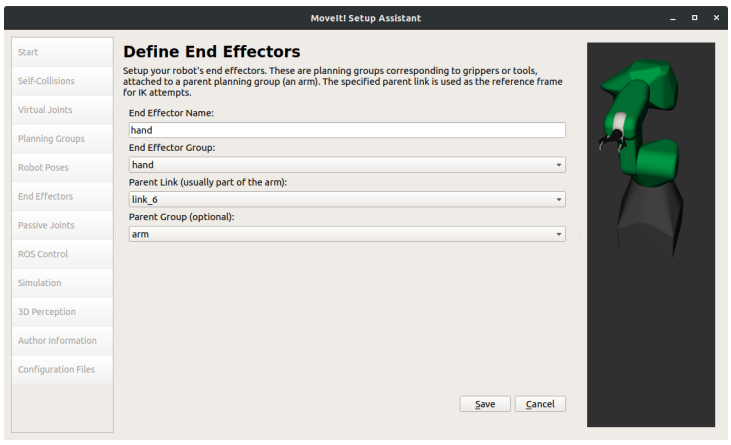
Figure A.1: Setup assistant initial screen

From this window we can update a node already generated with this tool or create a new one, for this tutorial we'll generate a new node. We must select a proper urdf file, we have used: `pick_and_place/src/fanuc_description/fanuc_cr35ia_support/urdf/assembled_arm_2.0.xacro`. After loading the urdf file we will see our manipulator and a list of possible panels to use:

- **Self collision**: allows to checks a number of possible self collision by the manipulator, we used the default value of sampling. In this panel it's sufficient to click on `Generate Collision Matrix`.
- **Virtual joints**: it's useful to link the robot with the scene, in general it's sufficient to add a simple virtual joint called *world*, with a parent called *world*. We just inserted these names and clicked on `save`.
- **Planning groups**: it's the most important part, we have to specify which joints compose the arm and which joints compose the gripper. We have added two groups that will be used directly in the script of our node, called `arm` and `hand`. For each group we have used these settings:



- **Robot poses**: it's possible to declare some particular poses, for example we saved the vertical pose of the arm as a pose named `default`.
- **End effector**: we have to explicit which planning group previously inserted is the end effector:



- **Author information**: insert some information about who generated the node.
- **Configuration file**: save all the files in a specific folder.

The generated node offers multiple launch files useful for the simulation, the most used is the `demo.launch` file that recalls all the other necessary files.