Project Requirements

Requirements for a Passing Project Submission

In the PR2 Pick and Place simulator, there are three different tabletop configurations. For a passing submission, your code must succeed in recognizing:

- 100% (3/3) objects in test1.world
- 80% (4/5) objects in test2.world
- 75% (6/8) objects in test3.world

A successful pick and place operation involves passing correct request parameters to the `pick_place_server`. Hence, for each test scene, correct values must be output to `.yaml` format for following parameters (see `/pr2_robot/config/output.yaml` for an example):

- Object Name (Obtained from the pick list)
- Arm Name (Based on the group of an object)
- Pick Pose (Centroid of the recognized object)
- Place Pose (Not a requirement for passing but needed to make the PR2 happy)
- Test Set Number

Name your output files `output_1.yaml`, `output_2.yaml`, and `output_3.yaml`, respectively.

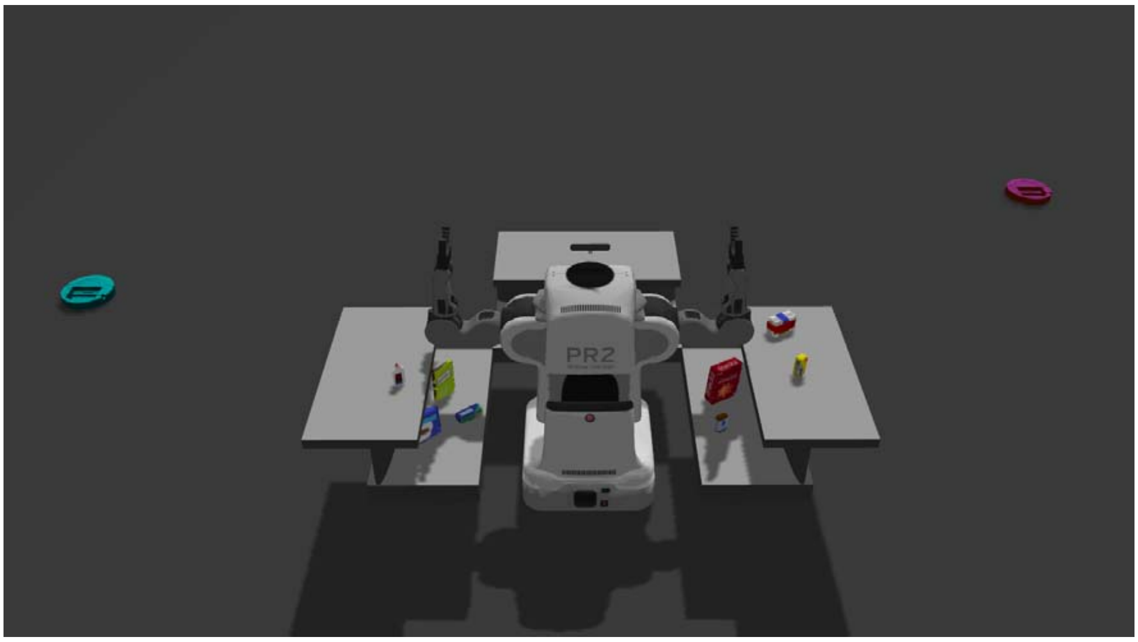
Once you have output `.yaml` files containing the correct message contents for your service request to `pick_place_server` you are done! But if you want the additional challenge, follow the steps described in the following sections lesson to complete the pick and place operation.

Additional Challenges

Follow the upcoming sections regarding collision mapping and using the `pick_place_server` to execute the pick and place operation!

Perception Hero Challenge:

For a heavier challenge, test the might of your perception pipeline against this complex tabletop configuration. Make yourself a pick list and decide where you want to put the items... maybe on another table? Maybe somewhere else? Have some fun with it!



To have a standout submission, you must submit a gif of your RViz window after successfully recognizing and labeling all the objects in this scene.

You think your perception pipeline is the best? Check out how others did in the #udacity_perception slack channel and share your results to find out who is the true Perception Hero!

PR2 Collision Avoidance

NOTE: If you have successfully output .yaml files containing your ROS messages for the service request for all three scenarios (test1.world, test2.world and test3.world) you have already achieved a passing submission of this project! Go ahead and submit now, or keep reading to take on further challenges.

To make PR2 aware of collidable objects like the table and objects on top of it, you must publish a point cloud which the motion planning pipeline can subscribe to and use it for a 3D collision map.

First you need to tell the simulator, which topic you will be publishing this point cloud to.

For that, open the sensors.yaml file under /RoboND-Perception-Project/pr2_robot/config and change the parameter named **point_cloud_topic** from /pr2/voxel_grid/points to /pr2/3d_map/points.
Important Note: In order to run the project in ***demo*** mode, you must change the **point_cloud_topic** parameter back to /pr2/voxel_grid/points
Next, you need to publish a point cloud to /pr2/3D_map/points. By publishing this point cloud on this topic you are telling the robot where objects are in the environment in order to avoid collisions. Finally, you need to decide what should be passed as a collidable point cloud to the motion planning pipeline.

As we discussed earlier, the table should certainly be passed as we do not want PR2 to karate chop it, but what about the objects on top of it?

If the object to be picked is a part of the point cloud published over /pr2/3d_map/points, the motion planning pipeline will consider that object as collidable and will fail to produce a pickup plan.

On the other hand, if rest of the objects on top of the table are not published as part of that point cloud, the robot essentially won't be able to see them and most likely knock them over during it's motion.

Keeping all that information in mind, let's go through a quick quiz to determine what your point cloud should consist of for optimum collision avoidance.

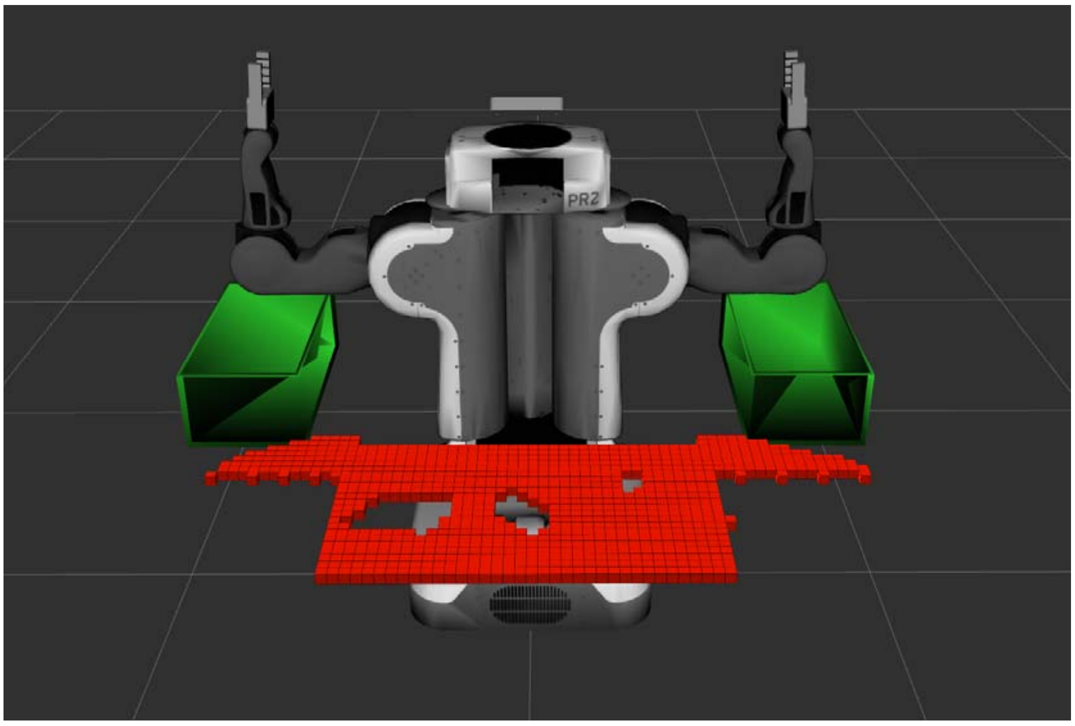
Robot Motion

At this point you have setup your perception pipeline, identified target object and obtained its centroid. In this section, we will explore the interface to control PR2's motion and further develop the pipeline to accomplish the pick and place task.

PR2 Base Joint

Since the robot picks up objects from the table and places them in boxes on it's sides, it is important to create a 3D collision map of this particular area for collision avoidance during trajectory execution.

Since the table and objects are right in front of the robot, the motion planning framework is able to create a collision map for that area.

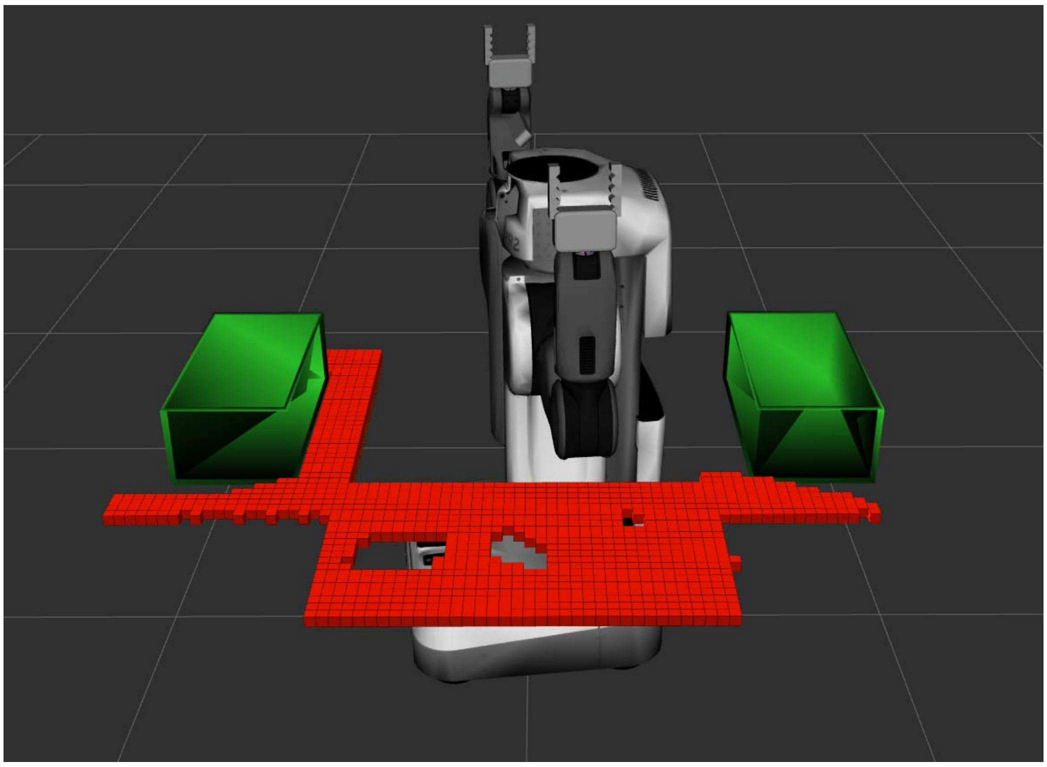


But to represent side tables with boxes in the collision map, you must rotate the robot in place.

This can be achieved by publishing joint_angle values directly to the world_joint_controller.
This world_joint_controller controls the revolute joint world_joint between the robot's base_footprint and world coordinate frames.
To control this joint publish joint_angle values (in radians) to the following topic:

```
/pr2/world_joint_controller/command
```

For a reminder on how to publish joint angles to a topic you can refer back to [this lesson]().



PR2 Arms

The left and right arms of PR2 are controlled using Moveit! motion planning framework. You can learn more about Moveit! [here]().
Once your perception pipeline has successfully recognized a target object from the pick list, you can invoke the pick and place functionality by means of pick_place_routine Service.
For a quick refresher on ROS Services, follow [this link]()
Essentially, pick and place operation is implemented as a request-response system, where you must write a ros_clientto extend a request to the pr2_pick_place_server. Have a look at PickPlace.srv in pr2_robot/srv. This script defines the format of the service message:

```
# request
```

```
std_msgs/Int32 test_scene_num
std_msgs/String object_name
std_msgs/String arm_name
geometry_msgs/Pose pick_pose
geometry_msgs/Pose place_pose
---
# response
bool success
```

The request your ros_client sends to the pr2_pick_place_server must adhere to the above format and contain:

Name	Message Type	Description	Valid Values
test_scene_num	std_msgs/Int32	The test scene you are working with	1,2,3
object_name	std_msgs/String	Name of the object, obtained from the pick-list	-
arm_name	std_msgs/String	Name of the arm	right, left
pick_pose	geometry_msgs/Pose	Calculated Pose of recognized object's centroid	-
place_pose	geometry_msgs/Pose	Object placement Pose	-

You already handled generating these messages for the .yaml output, but in reality, the argument **place_pose**, is a bit tricky.
At this point, you know which arm you are going to use for a given object based on its group, but clearly more than one object may need to be placed in either one of the drop boxes.

The robot needs to be efficient in its use of drop box space and not place multiple objects on top of each other.

Moreover, if all objects that belong to the same group were dropped at the same location, instead of stacking like they did in Kinematics project, here they will fall out of the drop box (making PR2 a sad robot).

Stack of cylinders created by Richie Muniak, a RoboND student, in the Kinematics Project.

For each pick and place operation, adjust the place_pose you send by a little bit so that your objects don't pile up but land side by side nice and cozy inside the box. As long as all the fields of your request message are valid and contain a pick_pose within a small tolerance of actual object location, the pr2_pick_place_server will use the designated arm to pick and place the object at a location specified by you.