**03-Environment Setup**

It is highly recommended to complete all three **[RoboND-Perception-Exercises](https://github.com/udacity/RoboND-Perception-Exercises" \t "_blank)** before starting any work on this project..

**For the rest of this setup, catkin\_ws is the name of active ROS Workspace, if your workspace name is different, change the commands accordingly.**

If you do not have an active ROS workspace, you can create one by:

$ mkdir -p ~/catkin\_ws/src

$ cd ~/catkin\_ws/

$ catkin\_make

Now that you have a workspace, clone or download [**the project repository**](https://github.com/udacity/RoboND-Perception-Project) into the src directory of your workspace:

$ cd ~/catkin\_ws/src

$ git clone https://github.com/udacity/RoboND-Perception-Project.git

**Important Note:** If you have the Kinematics Pick and Place project in the same ROS Workspace as this project, please remove the gazebo\_grasp\_plugin directory from this project.

Next install missing dependencies using rosdep install:

$ cd ~/catkin\_ws

$ rosdep install --from-paths src --ignore-src --rosdistro=kinetic -y

Build the project:

$ cd ~/catkin\_ws

$ catkin\_make

Add the following line to your .bashrc file:

export GAZEBO\_MODEL\_PATH=~/catkin\_ws/src/RoboND-Perception-Project/pr2\_robot/models:$GAZEBO\_MODEL\_PATH

This allows **Gazebo** to find custom models used in this project.

If you haven’t already, the following line can be added to your .bashrc to auto-source all new terminals:

source ~/catkin\_ws/devel/setup.bash

You have now successfully built the project and are ready to explore it further!

# 04-Project Demo

Once your environment is setup correctly and built the project, you can launch the project demo from a sourced terminal by:

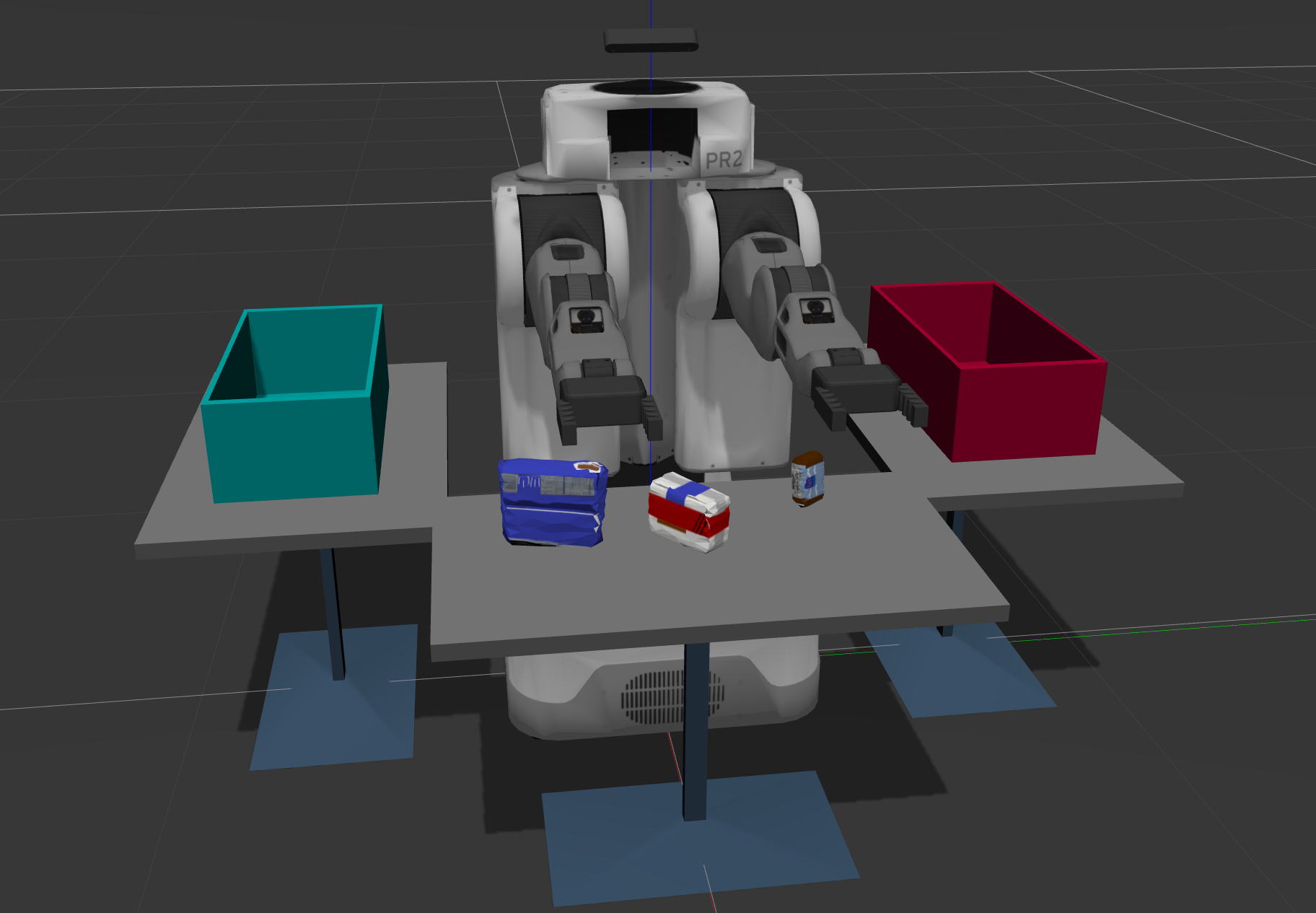
$ cd ~/catkin\_ws/src/RoboND-Perception-Project/pr2\_robot/scripts

$ chmod u+x pr2\_safe\_spawner.sh

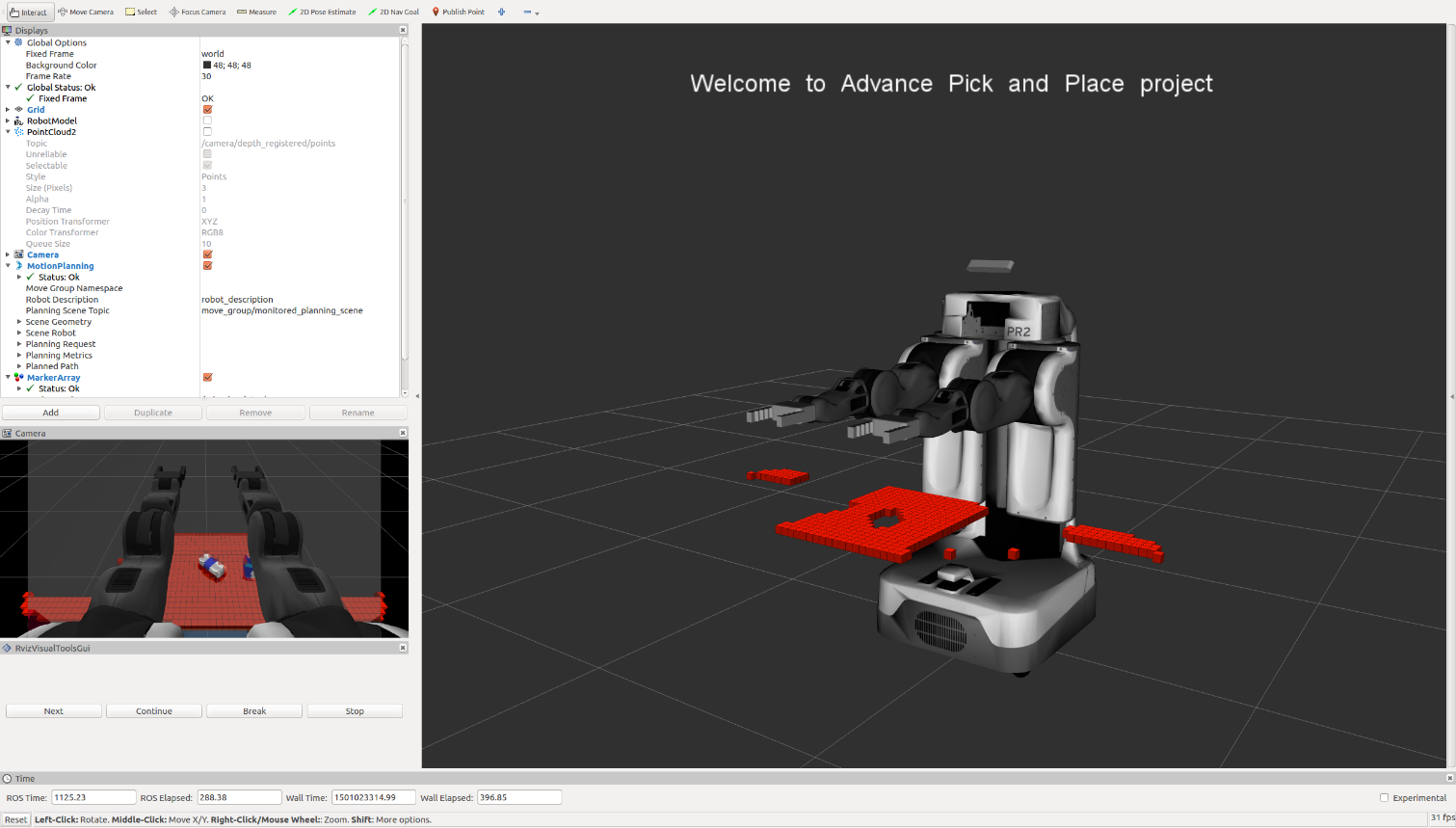
$ ./pr2\_safe\_spawner.sh

Once Gazebo is up and running, make sure you see the following objects in the gazebo world:

* Robot
* Table arrangement
* Three target objects on the table
* Dropboxes on either sides of the robot

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)**

On the RViz window, you should be able to see PR2 along with a preliminary collision map and a welcome message.

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)**

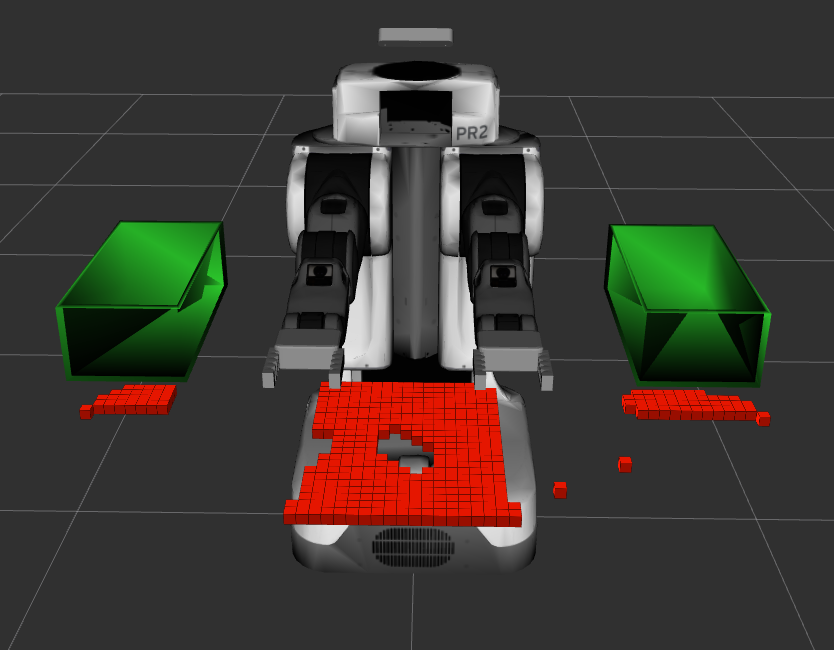
If there seems to be a problem with either gazebo or RViz, consider closing the demo by pressing **ctrl+c** in the terminal you used to launch the demo and restarting it.

If everything worked out fine, you can now begin the demo.

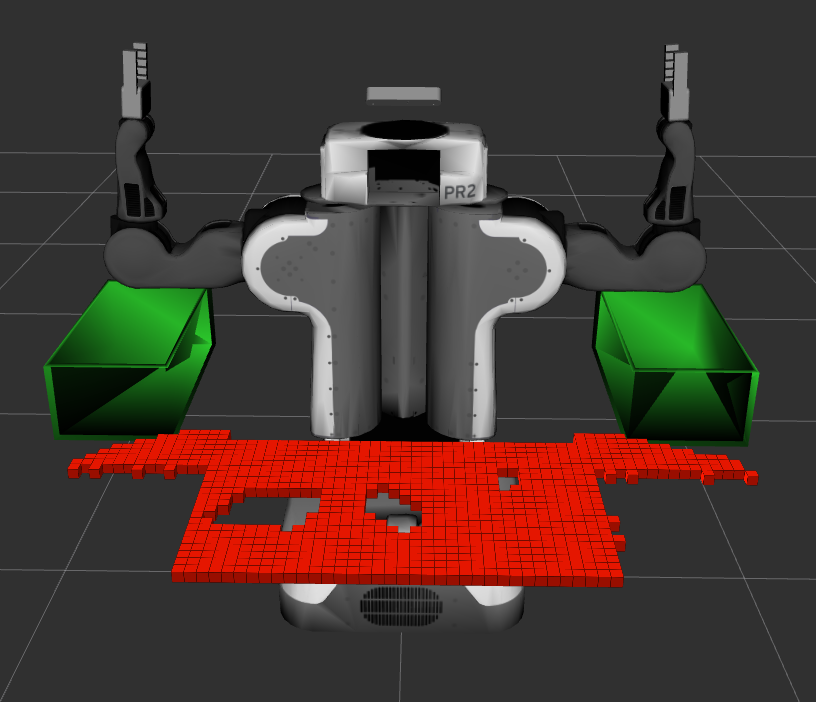
Just click on the Next button on the bottom left corner of the RViz window to proceed from one state to another.

Notice how a neat little status message is displayed about the simulator state on top of the robot in RViz window. This status message changes as we move through the different stages of simulation.

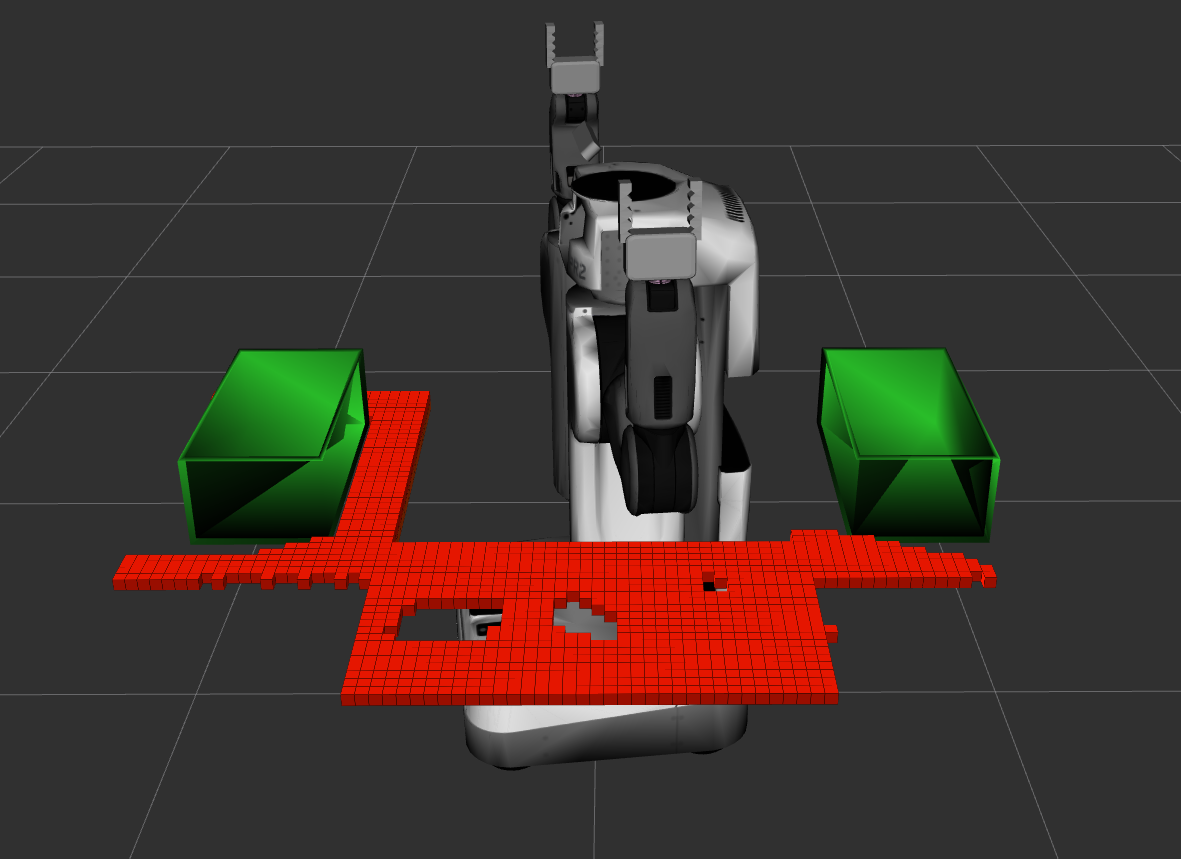
* The robot starts in an idle state, with both arms extended forward.
* Clicking on the Next button will add two dropboxes as collision elements.

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)**

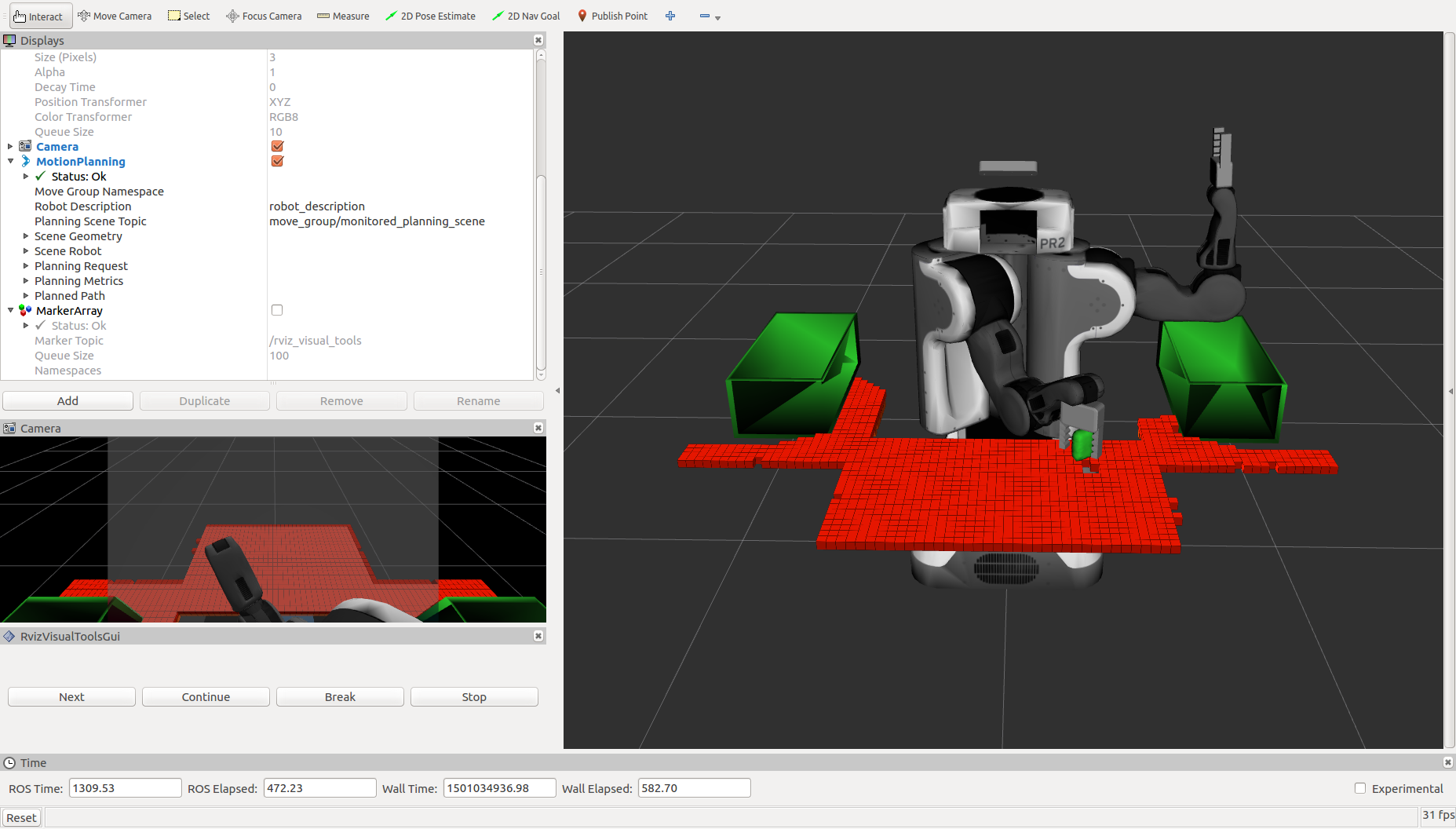
* Next, the robot raises its arms to enter the so-called "Ready State". This allows the robot to have an unobstructed vision in front of it, evident by a growing collision map.

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)**

* Further, the robot rotates in-place, first clockwise then anti-clockwise, finally settling at the center "Ready State". This maneuver allows the robot to expand the collision map and include the side tables as well.

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)**

* Next the robot identifies three tabletop objects (marked by insertion of green object meshes) and starts planning pick and place trajectories.

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)**

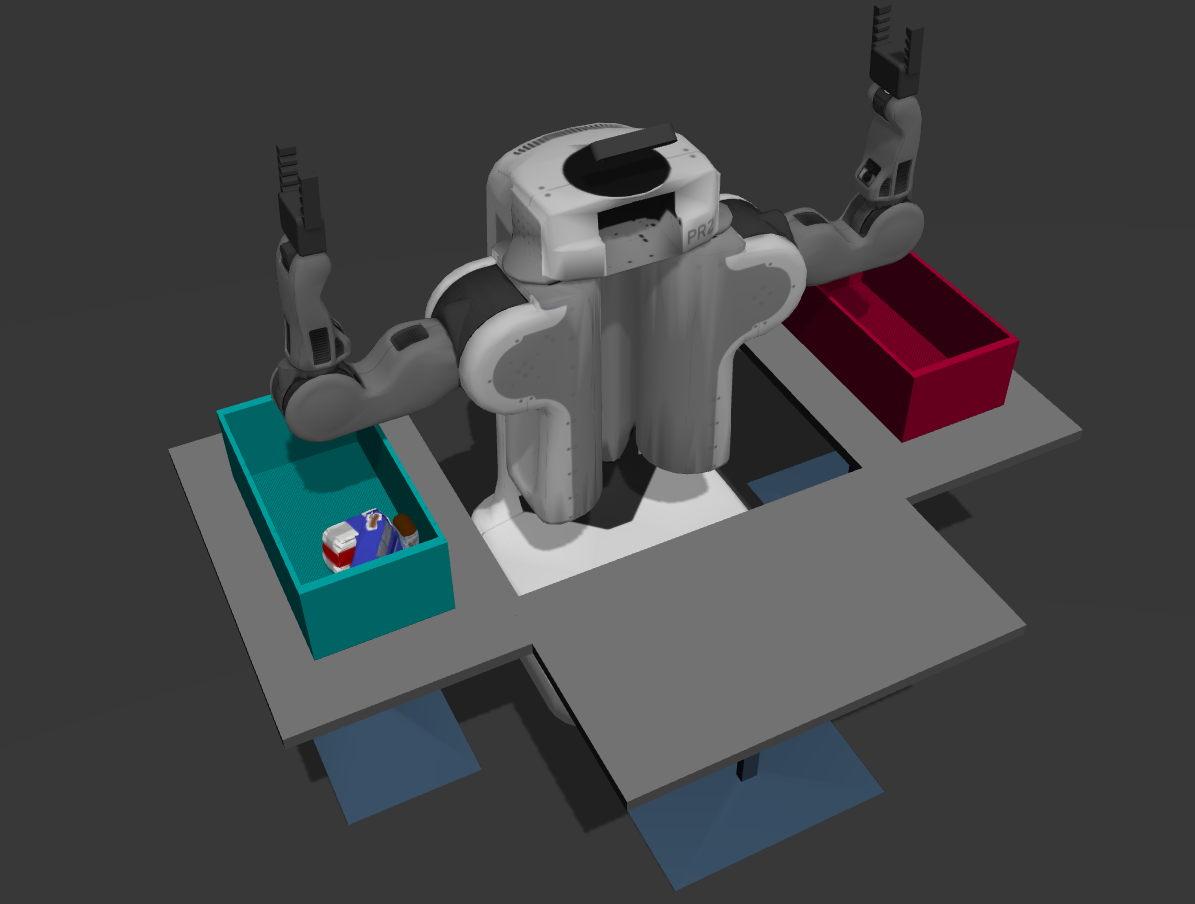
* From here on, just sit back, relax and watch as the robot swiftly accomplishes the pick and place task by depositing all three items.

This signifies the completion of one pick and place operation cycle.

Note that the planning framework creates a new plan with different trajectory points every time, even though the three objects are spawned at the same location.

Play around with this demo and don't forget to have fun!

#### Important Note: The robot is a bit moody at times and might leave objects on the table or fling them across the room.

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/ed92d363-b701-43b7-ac3e-e8c2fd771610)**

# 05-Perception Pipeline

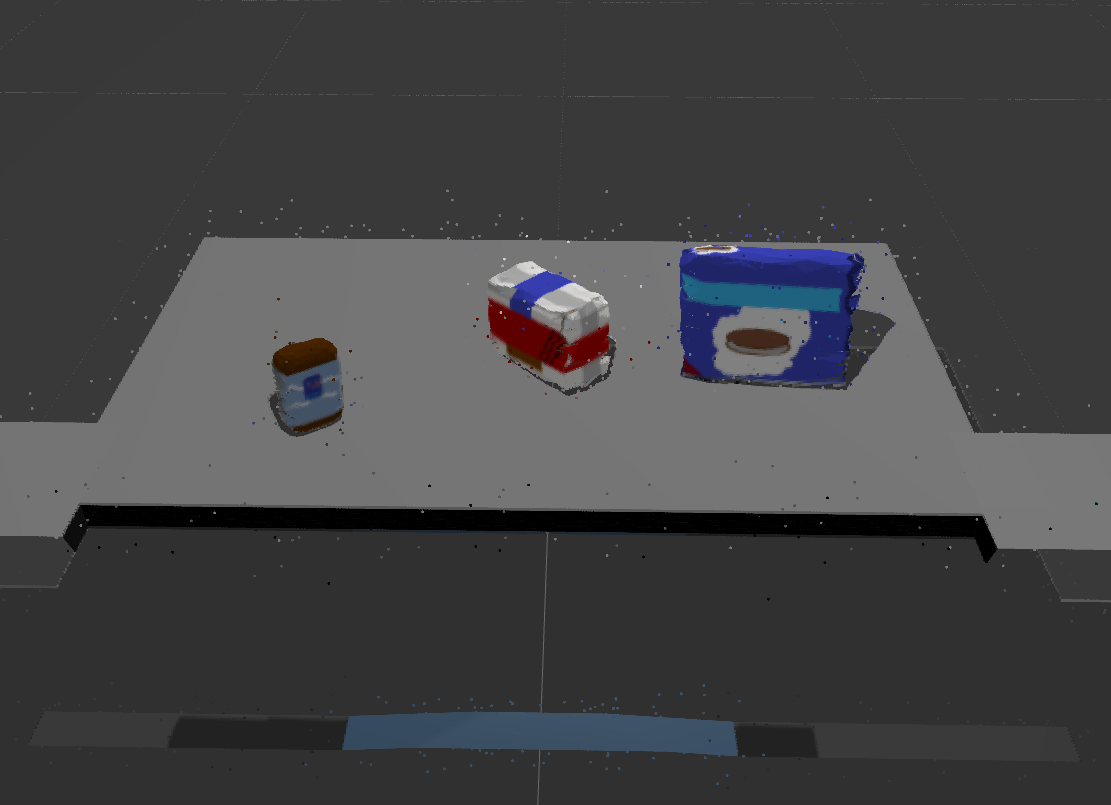
You will be pleased to know that you have already completed about 75% of this project by finishing the **[RoboND-Perception-Exercises](https://github.com/udacity/RoboND-Perception-Exercises" \t "_blank)**! All you need to do for a passing submission on this project is to implement your perception pipeline in a new environment with new objects and output some ROS messages telling the robot where to find them.

### Creating the perception pipeline

Start out by creating a ROS node just like you did in the exercises. In the pr2\_robot/scripts/ directory you'll find a file called project\_template.py, where you can move over all your code from Exercise-3 (or start from scratch if you like).

First, you'll need to change your subscriber (or create a new subscriber) to subscribe to the camera data (point cloud) topic /pr2/world/points.

To make sure that everything worked fine, try publishing the same data on your own topic and view it in RViz. To refresh your memory on how to do this, follow the tutorial in the [**Publish Your Point Cloud**](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/2cc29bbd-5c51-4c3e-b238-1282e4f24f42/concepts/57f2601e-7f58-4873-aead-70ba96da270f) section.

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)**

**[Initial point cloud with noise.](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)**

### Filtering

Once you have your camera data, start out by applying various filters you have looked at so far. Keep in mind you'll have to tweak the parameters to accommodate this new environment. Also, this new dataset contains noise! To clean it up, the first filter you should apply is the [**statistical outlier filter**](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/8d51e0bf-0fa1-49a7-bd45-e062c4a2121f/concepts/fdb3a445-43e0-4a02-81e2-0448432c156f?contentVersion=1.0.0&contentLocale=en-us).

Note, the statistical outlier filter in python-pcl was broken for a time and was only fixed after the exercises came out, so if you're getting an error when running the statistical outlier filter that looks like this:

Error: TypeError: \_\_cinit\_\_() takes exactly 1 positional argument (0 given)

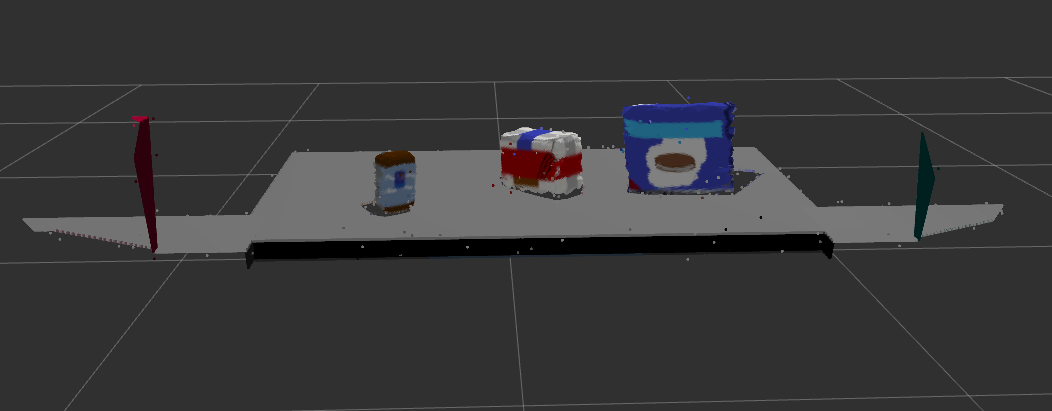
You'll need to git pull the **[RoboND-Perception-Exercises](https://github.com/udacity/RoboND-Perception-Exercises" \t "_blank)** repo to get the latest updates, then re-install python-pcl:

$ cd ~/RoboND-Perception-Exercises/python-pcl

$ python setup.py build

$ sudo python setup.py install

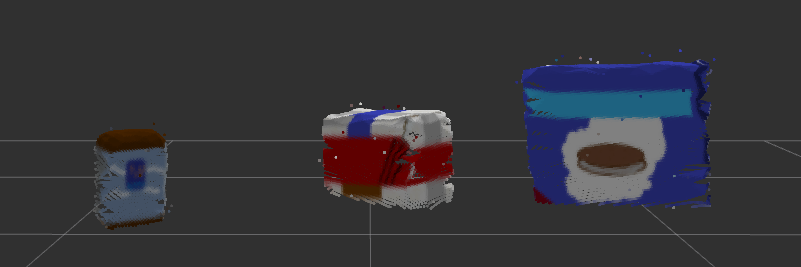
For a reminder on the filters covered in Exercise-1 have a look [**here**](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/8d51e0bf-0fa1-49a7-bd45-e062c4a2121f/concepts/7f0db904-2d61-476d-9f31-f80069150619)

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)**

**[Your point cloud after statistical outlier filtering](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)**

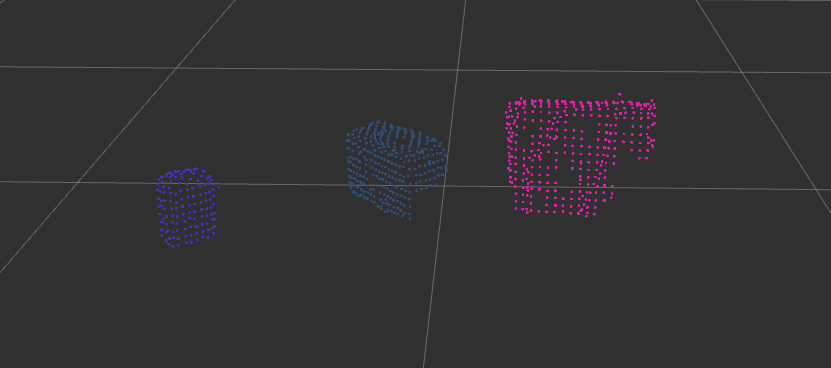
## Table Segmentation

Next, perform RANSAC plane fitting to segment the table in the scene. Much like you did [**in a previous lesson**](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/8d51e0bf-0fa1-49a7-bd45-e062c4a2121f/concepts/4c068d67-16e3-4d21-a8fc-5f13f88f8f29), to separate the objects from the table.

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)**

## Clustering

Use the Euclidean Clustering technique described [**here**](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/2cc29bbd-5c51-4c3e-b238-1282e4f24f42/concepts/02428d63-6f79-40dc-8105-31eda8e0def4) to separate the objects into distinct clusters, thus completing the segmentation process.

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)**

## Object Recognition

For this project, as you already saw in the previous section, you have a variety of different objects to identify. Essentially, there are three different worlds or scenarios that you are going to work with where each scenario has different items on the table in front of the robot. These worlds are located in the /pr2\_robot/worlds/ folder, namely the test\_\*.world files.

By default, you start with the test1.world but you can modify that in the pick\_place\_project.launch file in the /pr2\_robot/launch/ folder:

<!--Launch a gazebo world-->

<include file="$(find gazebo\_ros)/launch/empty\_world.launch">

<!--TODO:Change the world name to load different tabletop setup-->

<arg name="world\_name" value="$(find pr2\_robot)/worlds/test1.world"/>

<arg name="use\_sim\_time" value="true"/>

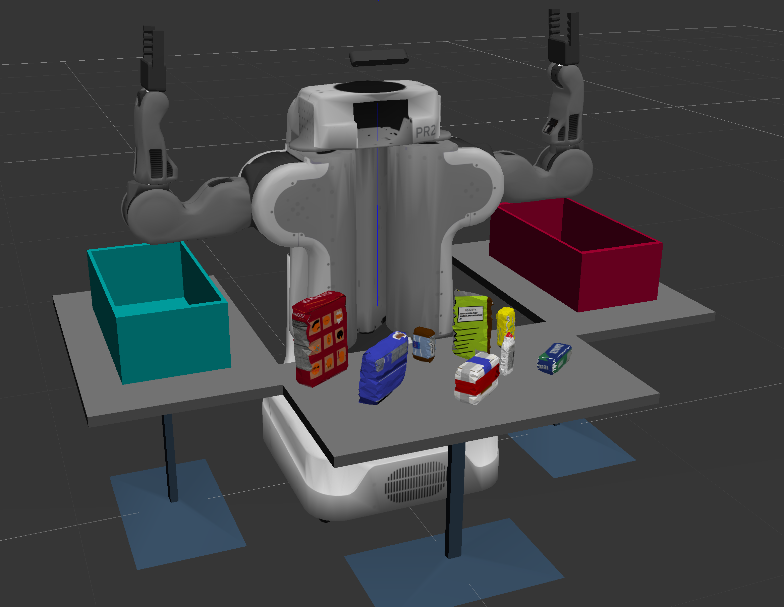
<arg name="paused" value="false"/>

<arg name="gui" value="true"/>

<arg name="headless" value="false"/>

<arg name="debug" value="false"/>

</include>

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)**

**[Test3 world with all 8 items](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)**

You can now complete the object recognition steps you performed in [**Exercise-3**](https://github.com/udacity/RoboND-Perception-Exercises/tree/master/Exercise-3) including:

* Generate a training set of features for the objects in your pick lists (see the pick\_list\_\*.yaml files in /pr2\_robot/config/). Each pick list corresponds to a world and thus indicates what items will be present in that scenario. To generate the training set, you will have to modify the models list in the capture\_features.py script and run it as you did for Exercise-3:
* **if** \_\_name\_\_ == '\_\_main\_\_':
* rospy.init\_node('capture\_node')
* *# Modify following list with items from pick\_list\_\*.yaml*
* models = [\
* 'beer',
* 'bowl',
* 'create',
* 'disk\_part',
* 'hammer',
* 'plastic\_cup',
* 'soda\_can']
* Train your SVM with features from the new models.
* Add your object recognition code to your perception pipeline.
* Test with the actual project scene to see if your recognition code is successful.

To test with the project, first run:

$ roslaunch pr2\_robot pick\_place\_project.launch

and then,

$ rosrun pr2\_robot project\_template.py

You should arrive at a result similar to what you got in Exercise-3 but this time for new objects in a new environment!

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/0f453188-f83b-48fd-b0d3-49f3d5bfed55)**

# 06-Output .yaml files

For a passing submission of this project, all you need to do is implement your perception pipeline in a new environment and correctly identify the majority of objects in three different scenarios. In each scenario, you'll be faced with the same robot environment and a different collection of objects on the table. In each case you'll be provided with a "pick list" of objects that you're looking for.

These pick lists exist as yaml files under /pr2\_robot/config and are loaded to the ros parameter server with the following line in your project launch file:

<rosparam command="load" file="$(find pr2\_robot)/config/{PICK\_LIST\_NAME}.yaml"/>

If you open up the project launch file called pick\_place\_project.launch in pr2\_robot/launch, you'll find the default pick list is set to pick\_list\_1.yaml, which looks like this:

object\_list:

- name: biscuits

group: green

- name: soap

group: green

- name: soap2

group: red

This list dictates the order in which objects are to be collected, along with each object's group, which determines which dropbox the object goes into.

You can retrieve the list from the parameter server and parse it in the following manner. Since the header of the pick list file was object\_list, that is the parameter name under which is loaded. To read this parameter, you can use:

*# get parameters*

object\_list\_param = rospy.get\_param('/object\_list')

The object\_list\_param now holds the pick list and can be parsed to obtain object names and associated group.

object\_name = object\_list\_param[i]['name']

object\_group = object\_list\_param[i]['group']

where i = index used to traverse the list.

For each item in the list you must identify the associated object in the scene and calculate its centroid. To calculate an object's centroid (average position of all the points in the object cloud) you can use a handy pclmethod to convert the cloud to an array. First though, recall that in your perception pipeline, you created a list of detected objects (of message type DetectedObject) like this:

*# Add the detected object to the list of detected objects.*

do = DetectedObject()

do.label = label

do.cloud = ros\_cluster

detected\_objects.append(do)

For each item in the list, you'll need to compare the label with the pick list and provide the centroid. You can grab the labels, access the (x, y, z) coordinates of each point and compute the centroid like this:

labels = []

centroids = [] *# to be list of tuples (x, y, z)*

**for** object **in** objects:

labels.append(object.label)

points\_arr = ros\_to\_pcl(object.cloud).to\_array()

centroids.append(np.mean(points\_arr, axis=0)[:3])

**WARNING: ROS messages expect native Python data types but having computed centroids as above your list centroids will be of type numpy.float64. To recast to native Python float type you can use np.asscalar(), for example:**

>>> import numpy as np

>>> x=np.mean([1.27,2.2,3.99])

>>> print(x)

2.48666666667

>>> type(x)

<class 'numpy.float64'>

>>> type(np.asscalar(x))

<class 'float'>

Now that you have the labels and centroids for target objects in the pick list, you can create the necessary ROS messages to send to the pick\_place\_routine service.

**Note: creating these messages successfully and outputting them to .yaml file format is all you are required to do to complete this project. After that, if you're interested in a challenge you can work on sending these messages and completing the pick and place operations.**

The pick and place operation is implemented as a request-response system, where you must write a ros\_client to 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 can obtain the **test\_scene\_num** by referring to the world file name, loaded in your pick\_place\_project.launch file. The line in your launch file looks like this:

<arg name="world\_name" value="$(find pr2\_robot)/worlds/test1.world"/>

Indicating in this case that the **test\_scene\_num** value is 1.

But be careful, the message type you need to send for this variable is not a simple **integer** but a ROS message of type std\_msgs/Int32.

To get more information on this message type. In a terminal type in the following command:

$ rosmsg info std\_msgs/Int32

and you'll get:

int32 data

Which means that this message type simply contains a data field of int32 type.

Meaning you cannot simply assign a number to std\_msgs/Int32. instead, you must first import this message type into your node:

**from** std\_msgs.msg **import** Int32

Next, initialize the **test\_scene\_num** variable:

test\_scene\_num = Int32()

And then populate the appropriate data field

test\_scene\_num.data = 1

Next, the **object\_name** is directly obtained by reading off of the pick list. But again, this is not a simple string but a std\_msgs/String type ros message. Just like before, you can investigate the contents of this message type by:

$ rosmsg info std\_msgs/String

And find that it contains a single data element:

string data

So in the script for your ROS node import the String message type and populate it with the appropriate value:

*# import the message type*

**from** std\_msgs.msg **import** String

*# Initialize a variable*

object\_name = String()

*# Populate the data field*

object\_name.data = object\_list\_param[i]['name']

Based on the **group** associated with each object (that you extracted from the pick list .yaml file), decide which arm of the robot should be used.

Since the green box is located on the right side of the robot, select the right arm for objects with green group and left arm for objects with redgroup. Like you did with object\_name, initialize the appropriate message type and populate it with the name of the arm.

Previously you wrote code to calculate the centroid of an identified object, this centroid will now be passed as the **pick\_pose** variable.

Since the message type for **pick\_pose** is geometry\_msgs/Pose, just like previous ROS message types, investigate it's contents, import the message type, initialize and then populate the fields with appropriate data.

$ rosmsg info geometry\_msgs/Pose

*# import message type*

**from** geometry\_msgs.msg **import** Pose

*# initialize an empty pose message*

pick\_pose = Pose()

*# TODO: fill in appropriate fields*

Now the final argument to be passed for a pick and place request is **place\_pose**. You can retrieve the place pose from the dropbox.yaml.

Just like the pick\_list.yaml file, dropbox.yaml is also loaded into the parameter server via the pick\_place\_project.launch by this line of code:

<rosparam command="load" file="$(find pr2\_robot)/config/dropbox.yaml"/>

The dropbox.yaml file looks like this:

dropbox:

- name: left

group: red

position: [0,0.71,0.605]

- name: right

group: green

position: [0,-0.71,0.605]

The position parameters give you the (x, y, z) positions of the centers of the bottom of each drop box.

Similar to what you did to parse the pick list .yaml file, parse dropbox.yaml and retrieve the position parameters. Then for each pick and place operation, populate the place\_pose positions with those values.

And that's it! You now have all the messages you need and you're ready to create a .yaml output file. The way it will actually work within your code is that you'll iterate over each item in the pick list, see whether you found it in your perception analysis, then if you did, populate the pick\_pose message with the centroid. Since you'll do this with each object one at a time, a convenient way to save the messages for each object is in a list of dictionaries.

We've provided a make\_yaml\_dict() helper function to help you convert messages to dictionaries. It looks like this:

**def** **make\_yaml\_dict**(test\_scene\_num, arm\_name, object\_name, pick\_pose, place\_pose):

yaml\_dict = {}

yaml\_dict["test\_scene\_num"] = test\_scene\_num.data

yaml\_dict["arm\_name"] = arm\_name.data

yaml\_dict["object\_name"] = object\_name.data

yaml\_dict["pick\_pose"] = message\_converter.convert\_ros\_message\_to\_dictionary(pick\_pose)

yaml\_dict["place\_pose"] = message\_converter.convert\_ros\_message\_to\_dictionary(place\_pose)

**return** yaml\_dict

This function takes advantage of further helper code in the rospy\_message\_converter package to convert ROS messages to dictionary format. With each iteration over the pick list, you can create a dictionary with the above function and then generate a list of dictionaries containing all your ROS service request messages. So for example, your for loop might look like this:

dict\_list = []

**for** i **in** range(0, len(object\_list\_param)):

*# Populate various ROS messages*

yaml\_dict = make\_yaml\_dict(test\_scene\_num, arm\_name, object\_name, pick\_pose, place\_pose)

dict\_list.append(yaml\_dict)

After finishing iterations, you can send your list of dictionaries to a .yamlfile using the yaml Python package and another helper function we provide called send\_to\_yaml() which looks like this:

**def** **send\_to\_yaml**(yaml\_filename, dict\_list):

data\_dict = {"object\_list": dict\_list}

**with** open(yaml\_filename, 'w') **as** outfile:

yaml.dump(data\_dict, outfile, default\_flow\_style=**False**)

**07-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!

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/77b74e0e-9f84-4e84-8e93-e5b27bac66fe)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/77b74e0e-9f84-4e84-8e93-e5b27bac66fe)**

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!

**08-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.

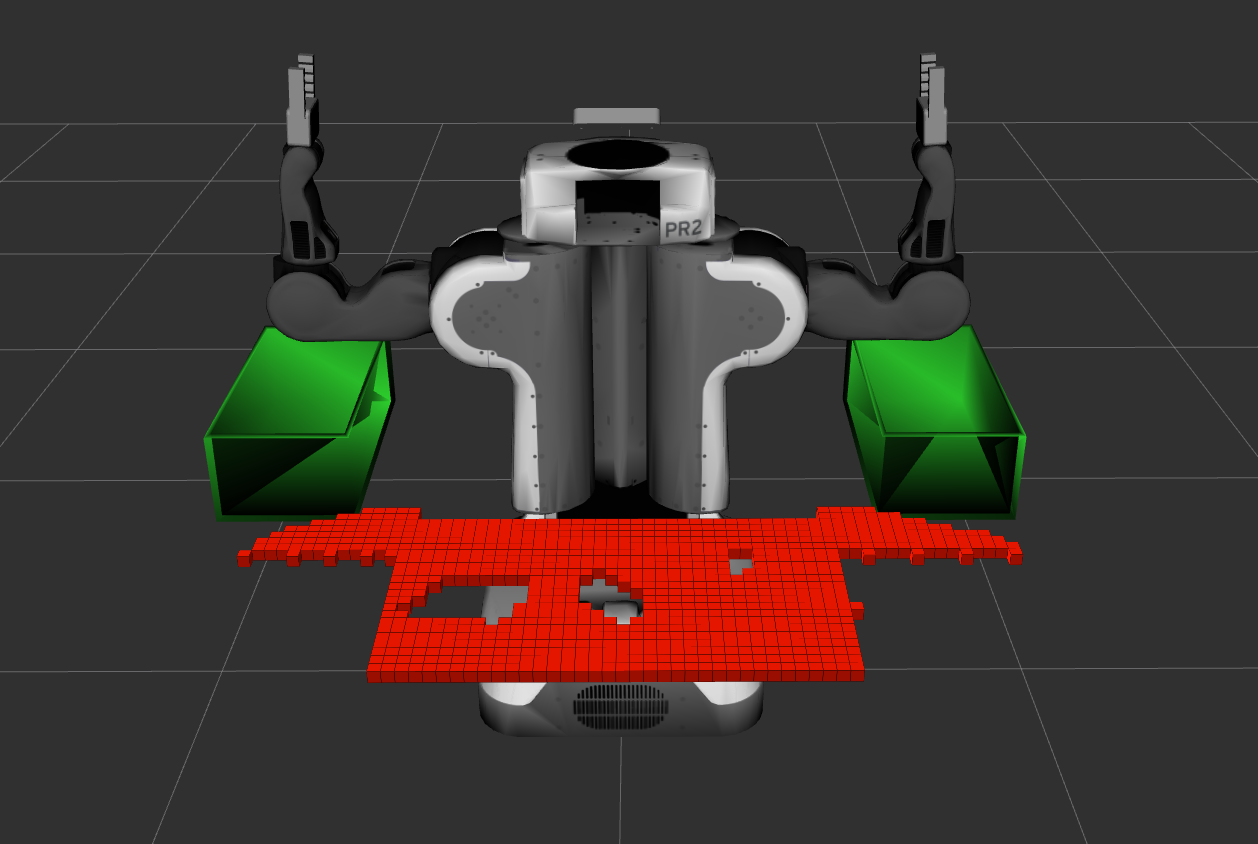
# 09-Robot Motion

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.

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/6946723a-7d12-4c1d-bc16-3707e2373e3d)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/6946723a-7d12-4c1d-bc16-3707e2373e3d)**

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

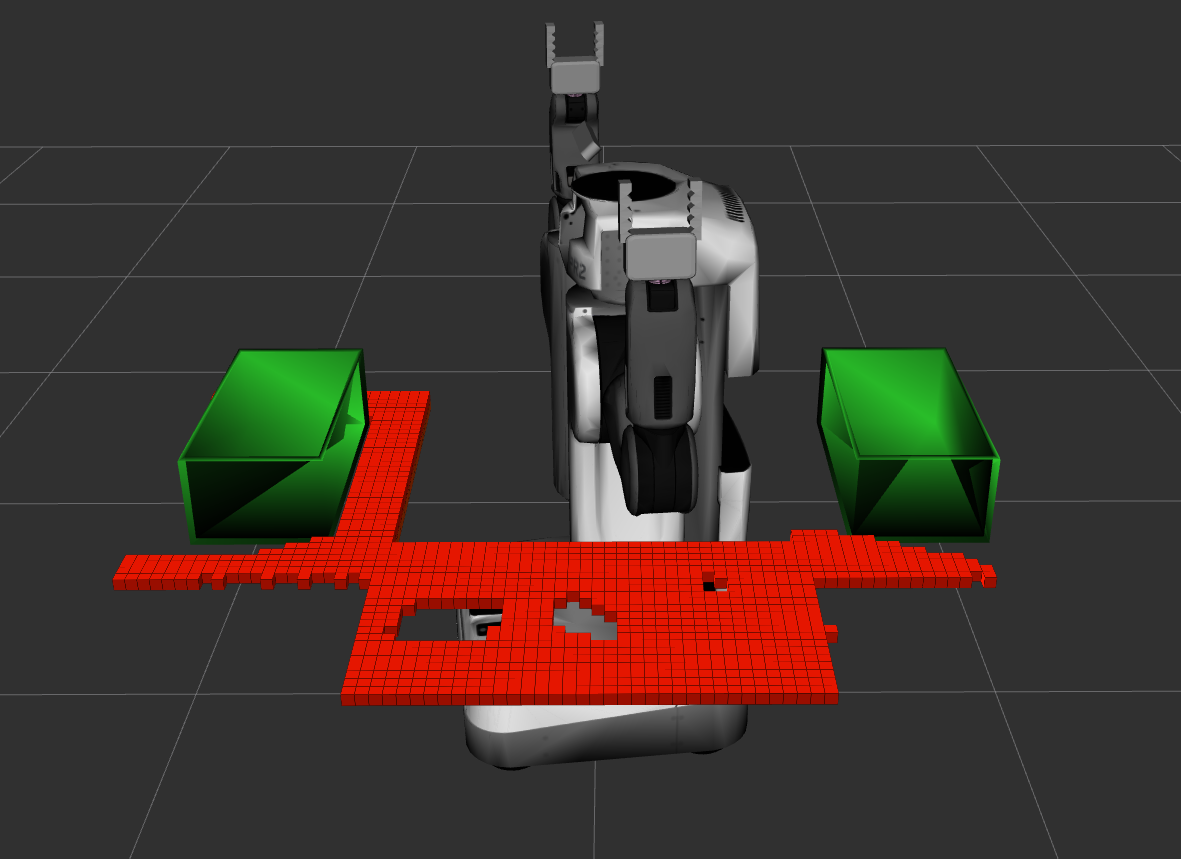
This can be achieved by publishing joint\_angle values directly to the world\_joint\_controller.

This world\_joint\_controller controls the revolute joint world\_jointbetween 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**](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/2919466f-aa2b-4424-b86a-98b0a53ce335/lessons/ade14c05-008c-4802-bfd8-f0271db9facc/concepts/faa4d5c5-54e9-4e35-9fb1-08e17b2bd97e?contentVersion=1.0.0&contentLocale=en-us).

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/6946723a-7d12-4c1d-bc16-3707e2373e3d)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/6946723a-7d12-4c1d-bc16-3707e2373e3d)**

### PR2 Arms

The left and right arms of PR2 are controlled using Moveit! motion planning framework. You can learn more about Moveit! [**here**](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/8855de3f-2897-46c3-a805-628b5ecf045b/lessons/91d017b1-4493-4522-ad52-04a74a01094c/concepts/c656a6e1-c31f-43c8-bff8-668fa412d74a).

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**](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/2919466f-aa2b-4424-b86a-98b0a53ce335/lessons/2cd33882-d29e-43e8-9ff7-10398c8b5351/concepts/eb3bf5a4-112f-4d9b-b631-ccfd10c294b8)

Essentially, pick and place operation is implemented as a request-response system, where you must write a ros\_client to 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).

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/6946723a-7d12-4c1d-bc16-3707e2373e3d)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/6946723a-7d12-4c1d-bc16-3707e2373e3d)**

**[Stack of cylinders created by Richie Muniak, a RoboND student, in the Kinematics Project.](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/6946723a-7d12-4c1d-bc16-3707e2373e3d)**

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.

### 10 – COMMON QUESTIONS

### Question: I'm encountering errors as I try to run the perception exercises, including issues installing pcl, how to fix this?

**Answer:** You should run inside the VM, not your local Mac, Windows, machine, etc. Then you should have no issues.

### Question: I'm trying to run the Perception exercises and I only see a blue stick for the robot inside RViz. I also notice some error message like this in the terminal. HELP!

[ERROR] [1499695108.298622255, 1132.552000000]: Lookup would require extrapolation into the past. Requested time 1128.729000000 but the earliest data is at time 1128.772000000, when looking up transform from frame [camera\_rgb\_optical\_frame] to frame [world] terminate called after throwing an instance of 'tf2::ConnectivityException' what(): Could not find a connection between 'world' and 'camera\_rgb\_optical\_frame' because they are not part of the same tree.Tf has two or more unconnected trees. [cloud\_transformer-8] process has died [pid 6137, exit code -6, cmd /home/robond/catkin\_ws/devel/lib/sensor\_stick/cloud\_transformer \_\_name:=cloud\_transformer \_\_log:=/home/robond/.ros/log/a1c14e10-6577-11e7-9e9f-000c298e62c8/cloud\_transformer-8.log]. log file: /home/robond/.ros/log/a1c14e10-6577-11e7-9e9f-000c298e62c8/cloud\_transformer-8\*.log

**Answer:** Double, triple check that you added this line correctly to your .bashrc.

export GAZEBO\_MODEL\_PATH=~/catkin\_ws/src/sensor\_stick/models

Also inside RViz change the point style from Points to Flat Squares.

Many seemingly unrelated errors come back to the Udacity can model not being loaded correctly. So even if, for example, your Gazebo starts up OK, but you can not see the light blue Udacity can (in addition to the generic soda can) then this line was not added correctly to .bashrc!

### Question: In Lesson 5.12 it says "And now we're ready to shuffle and split the data into training and testing sets. It's always a good idea to test your classifier on a separate dataset from the one you trained on, but first you should always randomize (shuffle) the data"

### Why should the data be randomized? Why can't we go sequential? The object of interest can be anywhere in the list.

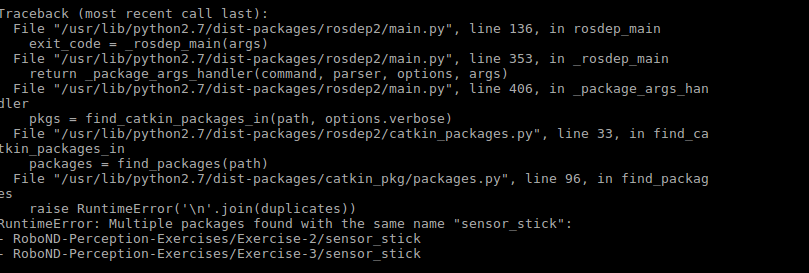
**Answer:** We built the dataset out of Cars and noCars samples which we then glued together sequentially. In this case they were roughly 50% each. We now want to take a section of this data to use for the training set, and another section to use for the testing set.

Now let's say we decide to take a training set of 80% of the data and testing set of the remaining 20%. If we had not shuffled the data then the training set would include all the Car samples and some of the noCar samples. The testing set would only include noCar samples and none for Car.

This is bad for two reasons. The classifier is being trained with proportionally more samples for Car, so it is more likely to start thinking everything is a car than it really should. And the other reason is since the test set contains no Car samples we can never check how good our model is at predicting cars.

If we shuffle the data first we get a better relatively balanced distributed of Cars and notCars and so we can train and test our model properly.

### Question: What to do if I get this error related to two different packages called sensor\_stick

**[[](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/75439d7d-4bd2-4cae-8110-934c282561f9)](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/75439d7d-4bd2-4cae-8110-934c282561f9)**

RuntimeError: Multiple packages found with the same name "sensor\_stick":

**Answer:** This occurs since there are sensor\_stick packages inside both the Exercise2 and Exercise3 folders and so if you placed the whole RoboND-Perception-Exercises repo inside your catkin\_ws you will get this issue.

It is not trivial to rename the package, so it is best to follow the exercise README exactly. So move the RoboND-Perception-Exercises repo outside your catkin\_ws and then just copy the sensor\_stick folder back to the catkin\_ws/src

## **12-[[1]](#endnote-1)Project Submission**

**Steps to complete the project:**

1. Make sure you have already setup your ROS Workspace in the VM provided by Udacity or on your own local Linux/ROS install. If you're not already setup, you can find instructions [**here**](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/2919466f-aa2b-4424-b86a-98b0a53ce335/lessons/658c94f5-f806-4273-9001-9e2838e56856/concepts/a777bc7a-95d4-44ca-b4e3-119718e3a213).
2. Complete Perception Exercises 1, 2 and 3, which comprise the project perception pipeline.
3. Download or clone the [**project repository**](https://github.com/udacity/RoboND-Perception-Project)
4. Follow the steps laid out in [**this lesson**](https://classroom.udacity.com/nanodegrees/nd209/parts/c199593e-1e9a-4830-8e29-2c86f70f489e/modules/e5bfcfbd-3f7d-43fe-8248-0c65d910345a/lessons/e3e5fd8e-2f76-4169-a5bc-5a128d380155/concepts/802deabb-7dbb-46be-bf21-6cb0a39a1961).

In this project, you must assimilate your work from previous exercises to successfully complete a tabletop pick and place operation using PR2. The PR2 has been outfitted with an RGB-D sensor much like the one you used in previous exercises. This sensor however is a bit noisy much like real sensors.

Given a cluttered tabletop scenario, you must implement a perception pipeline using your work from Exercises 1, 2 and 3 to identify target objects from a so-called “Pick-List” in a particular order, pick up those objects and place them in corresponding dropboxes.

Finally, you'll make a brief writeup report. The project repository has a writeup\_template.md that can be used as a guide.

## Evaluation

Once you have completed your project, use the [**Project Rubric**](https://review.udacity.com/#!/rubrics/1067/view) to review the project. If you have covered all of the points in the rubric, then you are ready to submit! If you see room for improvement in **any** category in which you do not meet specifications, keep working!

Your project will be evaluated by a Udacity reviewer according to the same [**Project Rubric**](https://review.udacity.com/#!/rubrics/1067/view). Your project must "meet specifications" in each category in order for your submission to pass.

## Submission

### What to include in your submission

You may submit your project as a zip file or with a link to a GitHub repo. The submission must include these items:

1. All your python code (make sure to add comments at appropriate places in your code)
2. Three output yaml files that contain PickPlace request parameters: output\_1.yaml, output\_2.yaml, output\_3.yaml.
3. Writeup report (md or pdf file)
4. (Optional) output\_4.yaml for the challenge environment along with screenshots/gif of your rviz window with objects labeled.

### Ready to submit your project?

Click on the "Submit Project" button and follow the instructions to submit!

You have not submitted the project yetSUBMIT PROJECT

1. [↑](#endnote-ref-1)