Environment Setup:

1- Create active ROS workspace:
$ mkdir -p ~/catkin_ws/src
$ cd ~/catkin_ws/
$ catkin_make

2- Clone the project repository into the **src** directory of the workspace:
$ cd ~/catkin_ws/src
$ git clone https://github.com/udacity/RoboND-Perception-Project.git

3- Install missing dependencies:
$ cd ~/catkin_ws
$ rosdep install --from-paths src --ignore-src --rosdistro=kinetic -y

4- Build the project:
$ cd ~/catkin_ws
$ catkin_make

5- Add the following to .bashrc file at the end:
export
GAZEBO_MODEL_PATH=~/catkin_ws/src/RoboND-Perception-Project/pr2_robot/models:$GAZEBO_MODEL_PATH

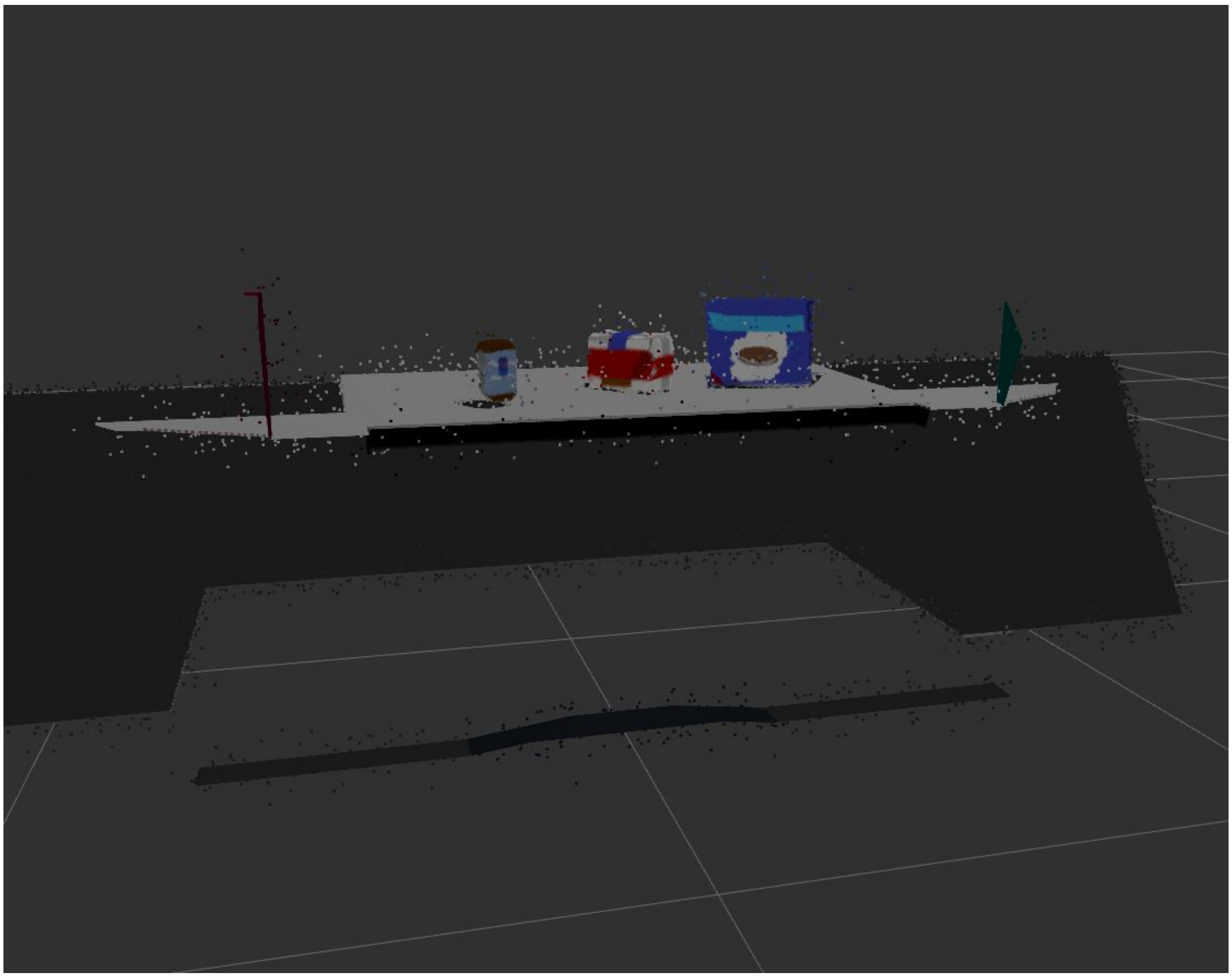
source ~/catkin_ws/devel/setup.bash

7- Save the .bashrc file.

Perception Pipeline:

The first step in the perception pipeline is to subscribe to the camera data (point cloud) topic from which we will get an initial point cloud with noise.

Here is the initial point cloud with noise:



Filtering:

We need to apply various filters on the raw point cloud to keep only the essential data.

1. **Outlier removal filter:**

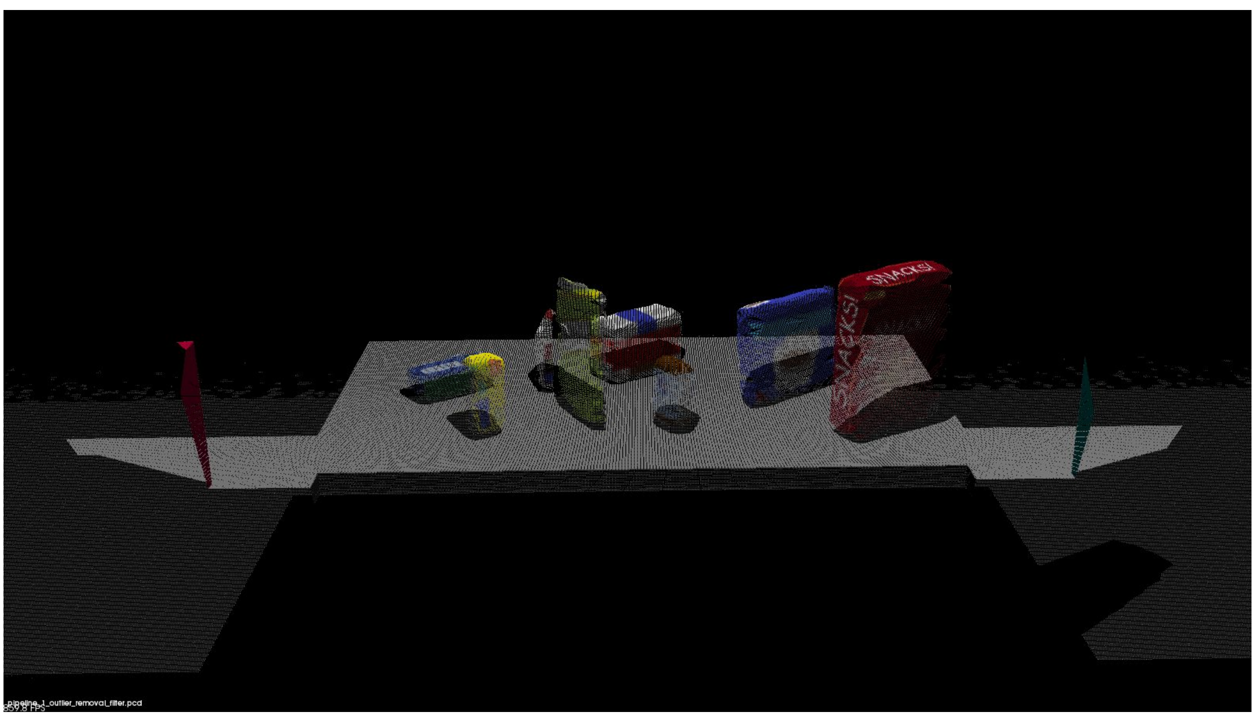
   The outlier removal filter is used to remove noise from the data by performing a statistical analysis in the neighbourhood of each point and remove those points which do not meet a certain criteria. For each point in the point cloud, it computes the distance to all of its neighbours and then calculates a mean distance. By assuming a gaussian distribution, all points whose mean distances are outside of an interval defined by the global distances mean + std-deviation are considered to be outliers and are removed from the point cloud.

   Code is as following:

```
# Statistical outlier filtering
outlier_filter = cloud.make_statistical_outlier_filter()
# Set the number of neighboring points to analyze for any given point
outlier_filter.set_mean_k(20)
# Any point with a mean distance larger than global will be considered out
outlier_filter.set_std_dev_mul_thresh(0.1)
cloud_filtered = outlier_filter.filter()
```

   A mean k value of 20 and a standard deviation threshold of 0.1 is used.

   Here is the cloud after performing the outlier removal filter:



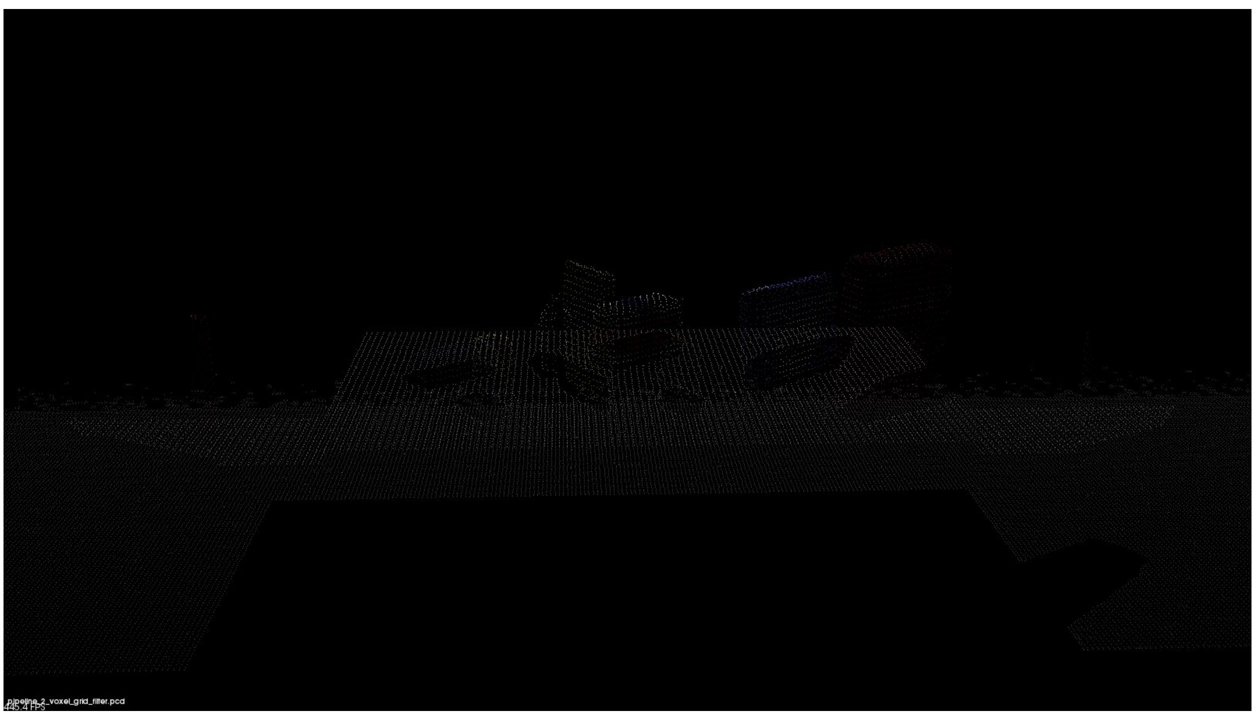
2. Voxel grid Downsampling filter:

A voxel grid filter down-samples the data by taking a spatial average of the points in the cloud confined by each voxel. The set of points which lie within the bounds of a voxel are assigned to that voxel, and are statistically combined into one output point.

Code is as following:

```
# Create a VoxelGrid filter object for our input point cloud
vox = cloud.make_voxel_grid_filter()
# Choose a voxel (also known as leaf) size
LEAF_SIZE = 0.01
# Set the voxel (or leaf) size
vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
# Call the filter function to obtain the resultant downsampled point cloud
cloud_filtered = vox.filter()
```

Here is the cloud after performing the Voxel grid Downsampling filter:



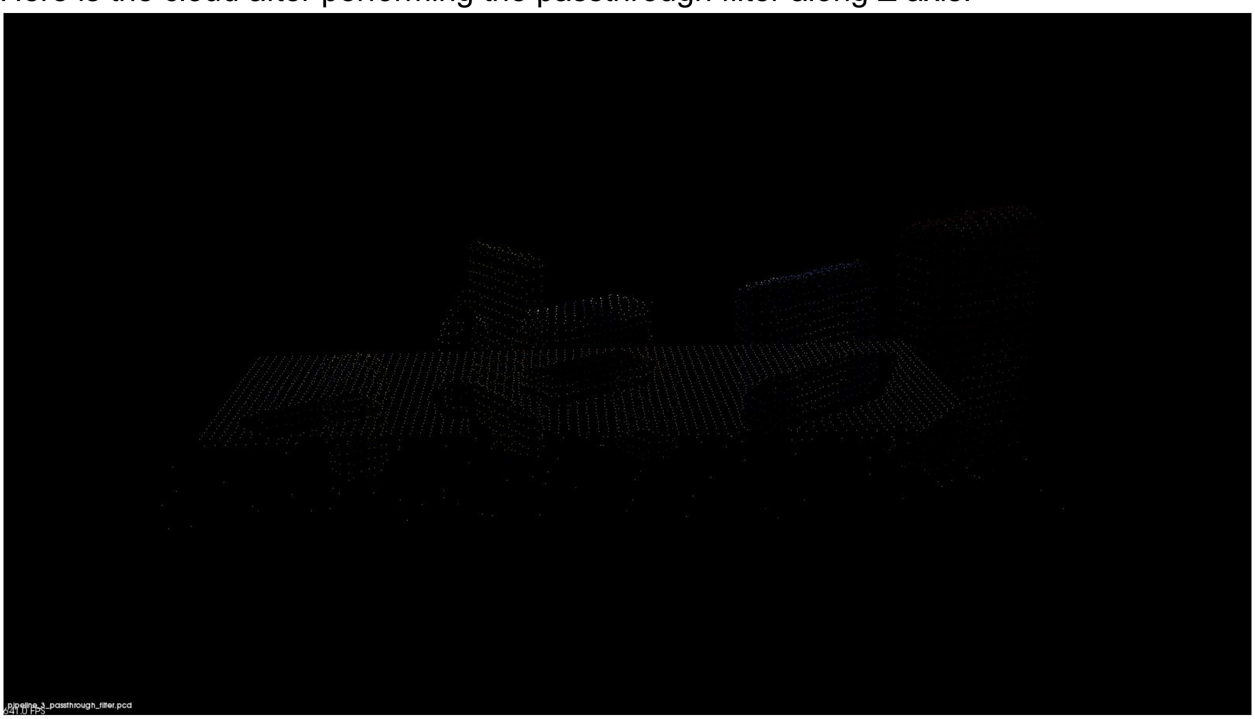
3. Passthrough filter:

The passthrough filter works much like a cropping tool, which allows us to crop 3D point cloud by specifying an axis with cutoff values along that axis. The regions we allow to passthrough, are often referred to as "regions of interest". We will apply the filter for Z axis.

Code is as following:

```
# Create a PassThrough filter object.
passthrough = cloud_filtered.make_passthrough_filter()
# Assign axis and range to the passthrough filter object.
filter_axis = 'z'
passthrough.set_filter_field_name(filter_axis)
axis_min = 0.6
axis_max = 1.1
passthrough.set_filter_limits(axis_min, axis_max)
# Finally use the filter function to obtain the resultant point cloud.
cloud_filtered = passthrough.filter()
```

Here is the cloud after performing the passthrough filter along Z axis:



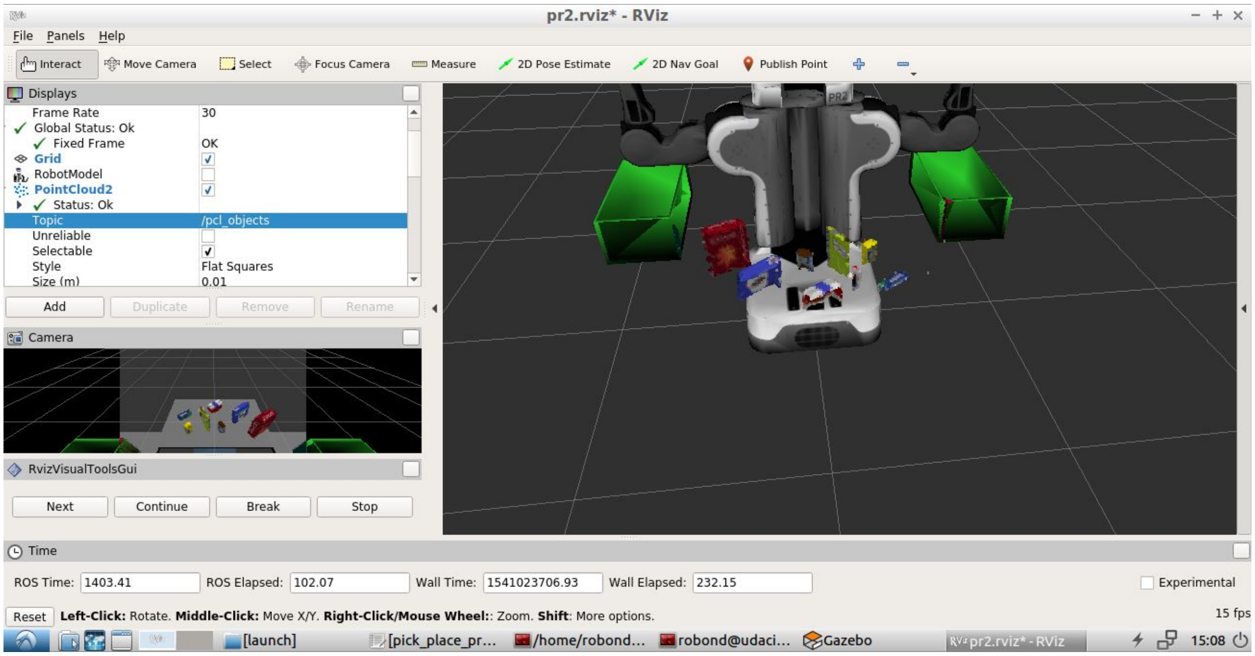
4. RANSAC Plane Segmentation:

RANSAC is used to remove the table itself from the scene. RANSAC algorithm will be used to identify points in dataset that belong to a particular model. The RANSAC algorithm assumes that all of the data in a dataset is composed of both inliers and outliers, where inliers are defined by a particular model with a specific set of parameters, while outliers do not fit that model and hence can be discarded.
By modelling the table as a plane we can remove it from the point cloud.

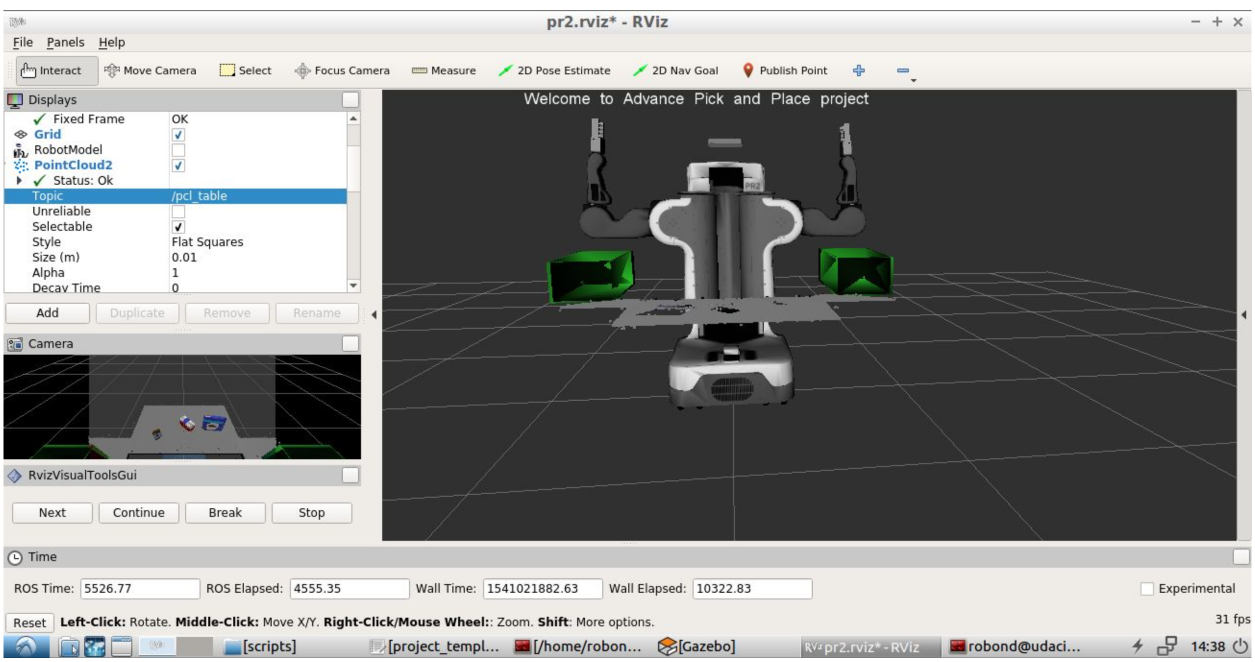
Code is as following:

```
# Create the segmentation object
seg = cloud_filtered.make_segmenter()

# Set the model you wish to fit
seg.set_model_type(pcl.SACMODEL_PLANE)
seg.set_method_type(pcl.SAC_RANSAC)

# Max distance for a point to be considered fitting the model
# Experiment with different values for max_distance
# for segmenting the table
max_distance = 0.01
seg.set_distance_threshold(max_distance)

# Call the segment function to obtain set of inlier indices and model coefficients
inliers, coefficients = seg.segment()

# TODO: Extract inliers and outliers
cloud_table = cloud_filtered.extract(inliers, negative=False)
cloud_objects = cloud_filtered.extract(inliers, negative=True)
```

Here is the objects:



Here is the table:

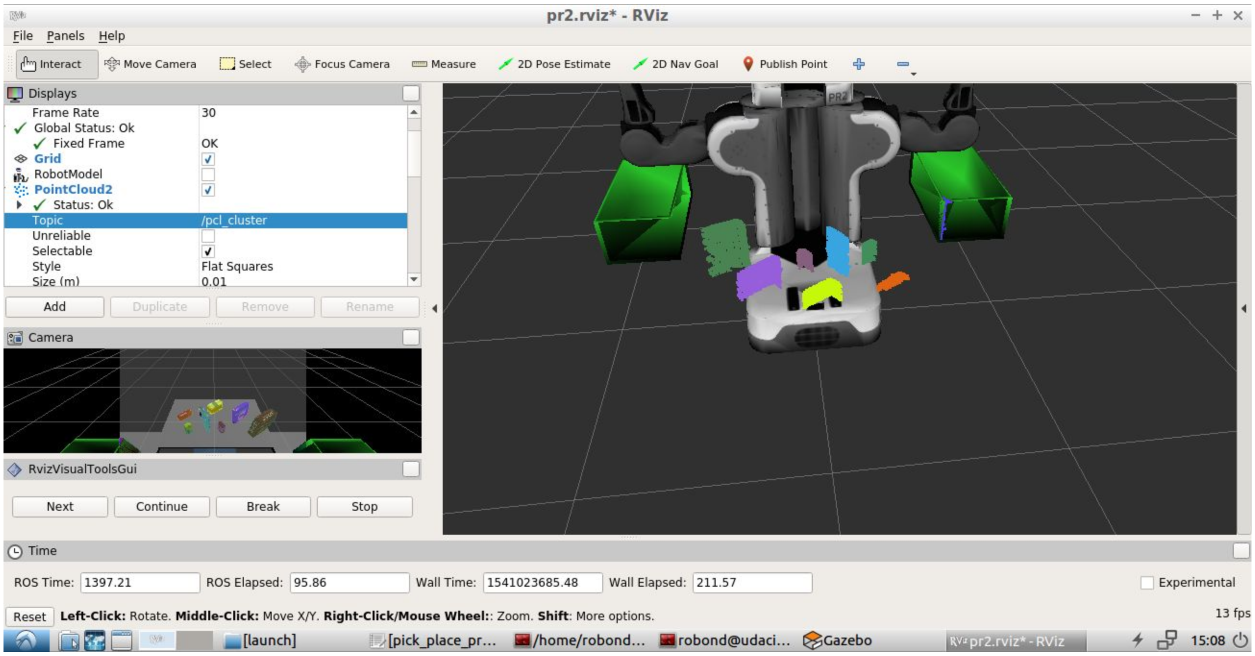


Clustering:

The Euclidean Clustering technique is used to separate the objects into distinct clusters. Code is as following:

```
 # Euclidean Clustering
 white_cloud = XYZRGB_to_XYZ(cloud_objects)# Apply function to convert XYZRGB to XYZ
 tree = white_cloud.make_kdtree()
 # TODO: Create Cluster-Mask Point Cloud to visualize each cluster separately
 ec = white_cloud.make_EuclideanClusterExtraction()
 # Set tolerances for distance threshold
 # as well as minimum and maximum cluster size (in points)
 ec.set_ClusterTolerance(0.05) #0.05
 ec.set_MinClusterSize(100) #100
 ec.set_MaxClusterSize(3000) #3000
 # Search the k-d tree for clusters
 ec.set_SearchMethod(tree)
 # Extract indices for each of the discovered clusters
 cluster_indices = ec.Extract()

 #Assign a color corresponding to each segmented object in scene
 cluster_color = get_color_list(len(cluster_indices))

 color_cluster_point_list = []

 for j, indices in enumerate(cluster_indices):
     for i, indice in enumerate(indices):
         color_cluster_point_list.append([white_cloud[indice][0],
                                          white_cloud[indice][1],
                                          white_cloud[indice][2],
                                          rgb_to_float(cluster_color[j])])

 #Create new cloud containing all clusters, each with unique color
 cluster_cloud = pcl.PointCloud_PointXYZRGB()
 cluster_cloud.from_list(color_cluster_point_list)

 # TODO: Convert PCL data to ROS messages
 ros_cloud_objects =  pcl_to_ros(cloud_objects)
 ros_cloud_table = pcl_to_ros(cloud_table)
 ros_cloud_object_cluster = pcl_to_ros(cluster_cloud)

 # TODO: Publish ROS messages
 pcl_objects_pub.publish(ros_cloud_objects)              # original color objects
 pcl_table_pub.publish(ros_cloud_table)                  # table cloud
 pcl_objects_cloud_pub.publish(ros_cloud_object_cluster) # solid color objects
```

Here is the objects after clustering:



Object Recognition:

The object recognition code allows each object within the object cluster to be identified. In order to do this, the system first needs to train a model to learn what each object looks like. Once it has this model, the system will be able to make predictions as to which object it sees. The goal is to find the features that best describe the object we are looking for. The better the description of the object we are looking for, the more likely the algorithm is to find it.

1. Capture Object Features:

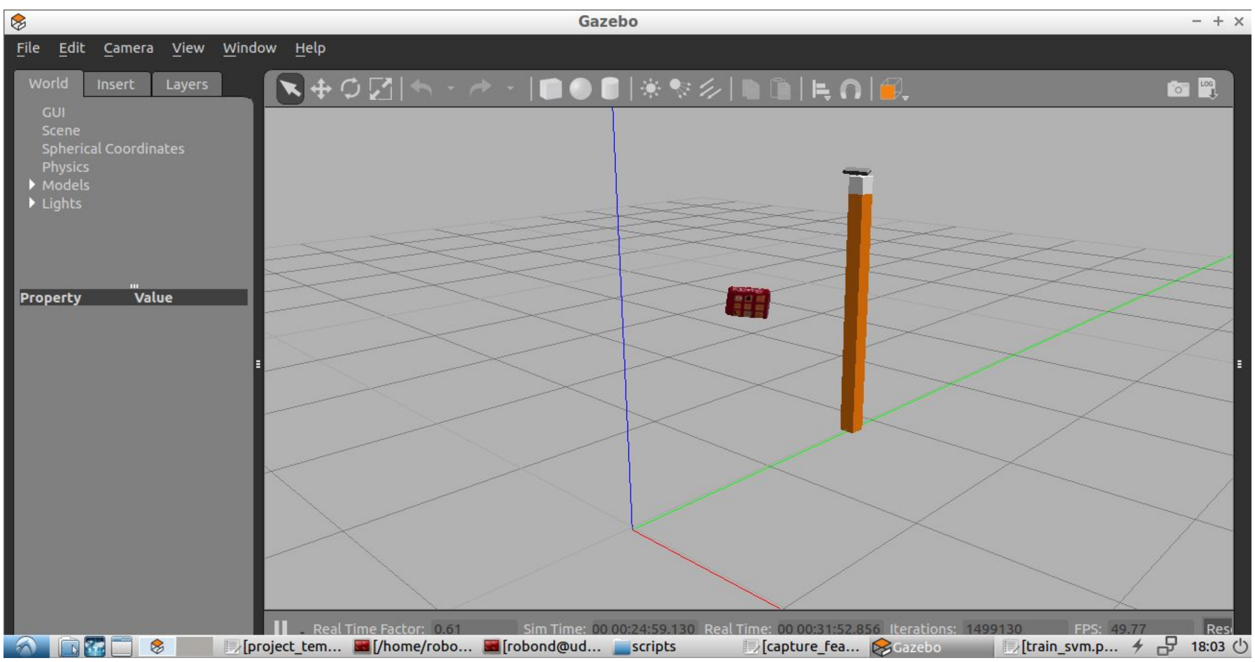
To capture the point cloud features, I used the sensor_stick model to analyze and record each object of the PR2 project. In order to do this, I copied the models folder from the PR2 project into the models folder of the sensor stick folder (the one used for Exercise 3). Once the models were stored there, I just had to alter the model names in the capture_features.py file in the sensor_stick/scripts folder to match the PR2 model names. I saved this file under the name capture_features_pr2.py.

With the file prepared, the next step is to launch the Gazebo environment:

- roslaunch sensor_stick training.launch

Then the capture_features_pr2.py script could be run:

- rosrun sensor_stick capture_features_pr2.py



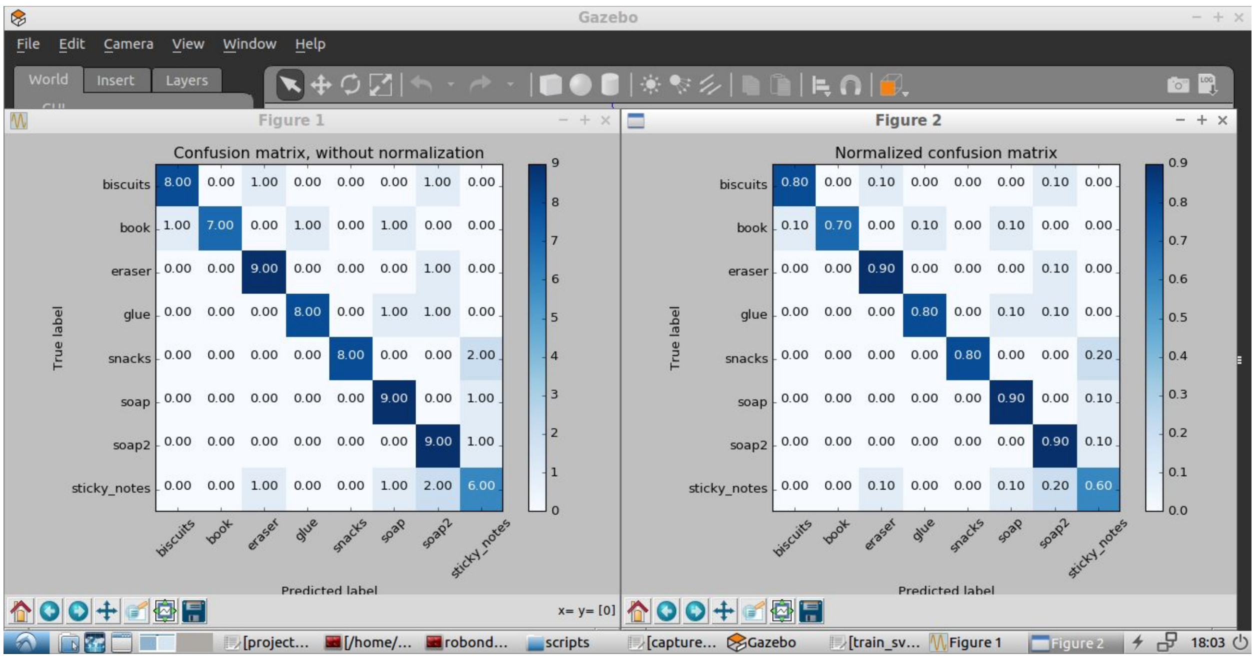
2. **Train SVM Model:**

To start the training, run:

- rosrun pr2_robot train_svm.py

This will train the SVM and save the final model to model_pr2.sav.

The SVM loads the generated training set, and prepares the raw data for classification.



Running the 3 tests:

Test 1:

To run test 1

1- set ‘test_num’ variable in ‘object_recognition.py’ file to be equal 1

2- check number of test in line 13 in ‘pick_place_project.launch’ is set to test1.world

3- check line 39 in ‘pick_place_project.launch’ is set to pick_list_1

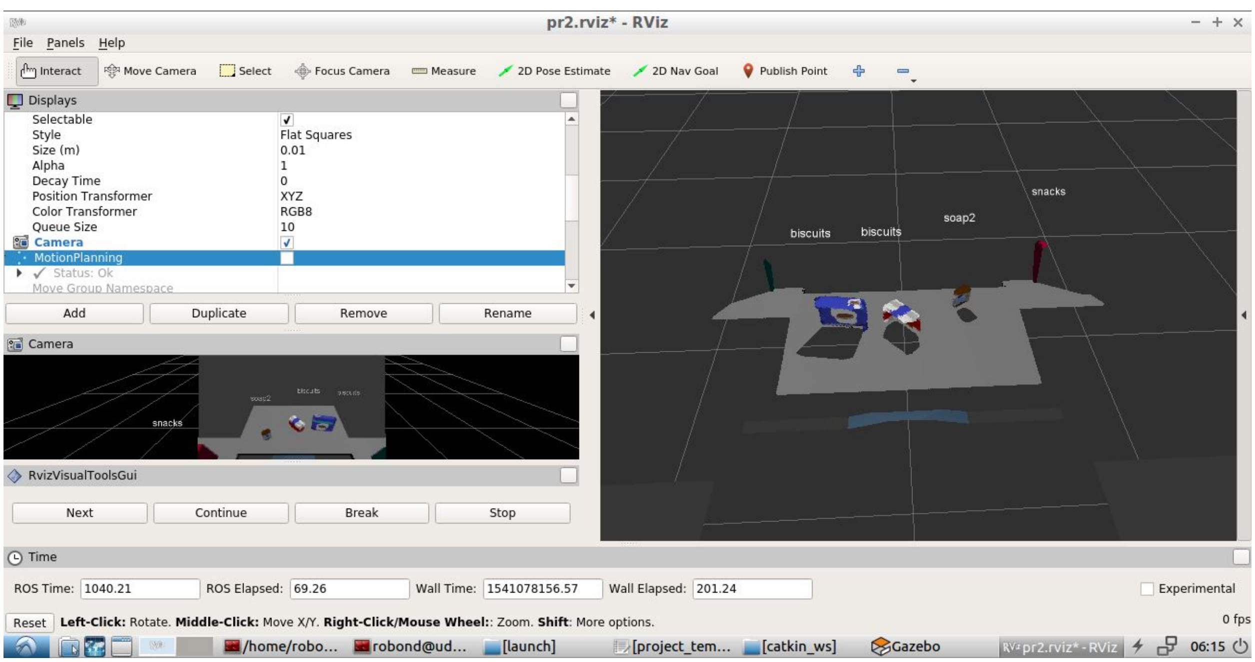
4- launch the Gazebo environment:

roslaunch pr2_robot pick_place_project.launch

5- run the object recognition script:

rosrun pr2_robot object_recognition.py

Image of predicted objects:



Test 2:

To run test 2

1- set ‘test_num’ variable in ‘object_recognition.py’ file to be equal 2

2- check number of test in line 13 in ‘pick_place_project.launch’ is set to test2.world

3- check line 39 in ‘pick_place_project.launch’ is set to pick_list_2

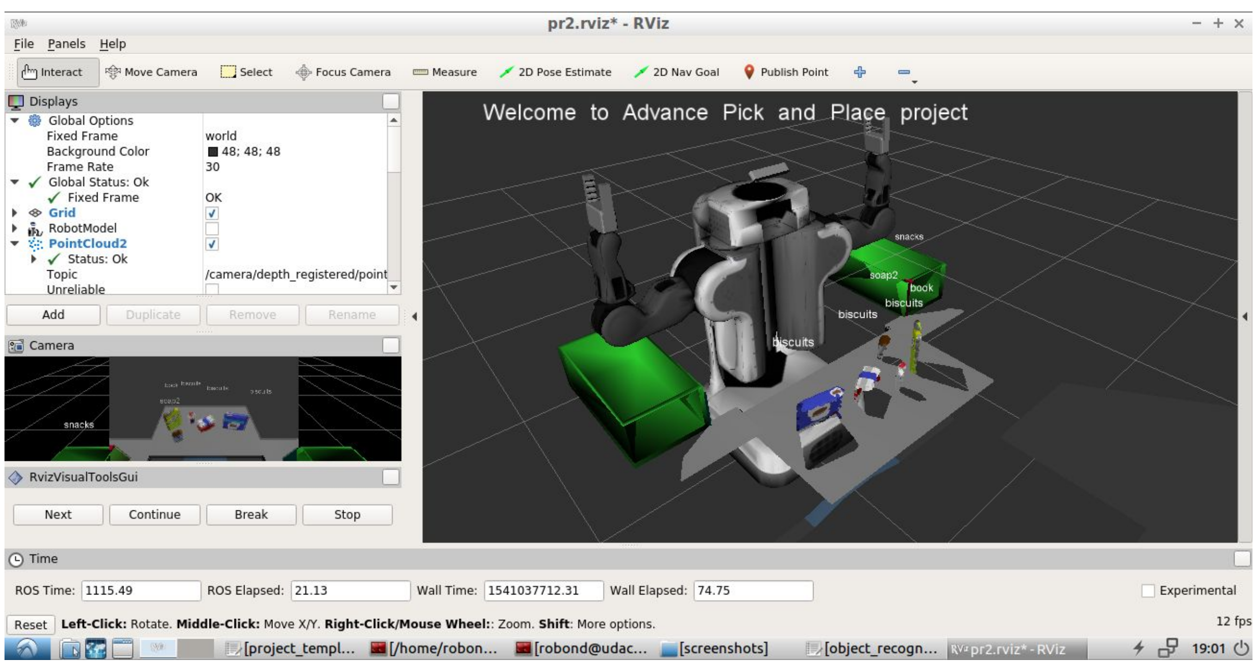
4- launch the Gazebo environment:

roslaunch pr2_robot pick_place_project.launch

5- run the object recognition script:

rosrun pr2_robot object_recognition.py

Image of predicted objects:



Test 3:

To run test 3

1- set ‘test_num’ variable in ‘object_recognition.py’ file to be equal 3

2- check number of test in line 13 in ‘pick_place_project.launch’ is set to test3.world

3- check line 39 in ‘pick_place_project.launch’ is set to pick_list_3

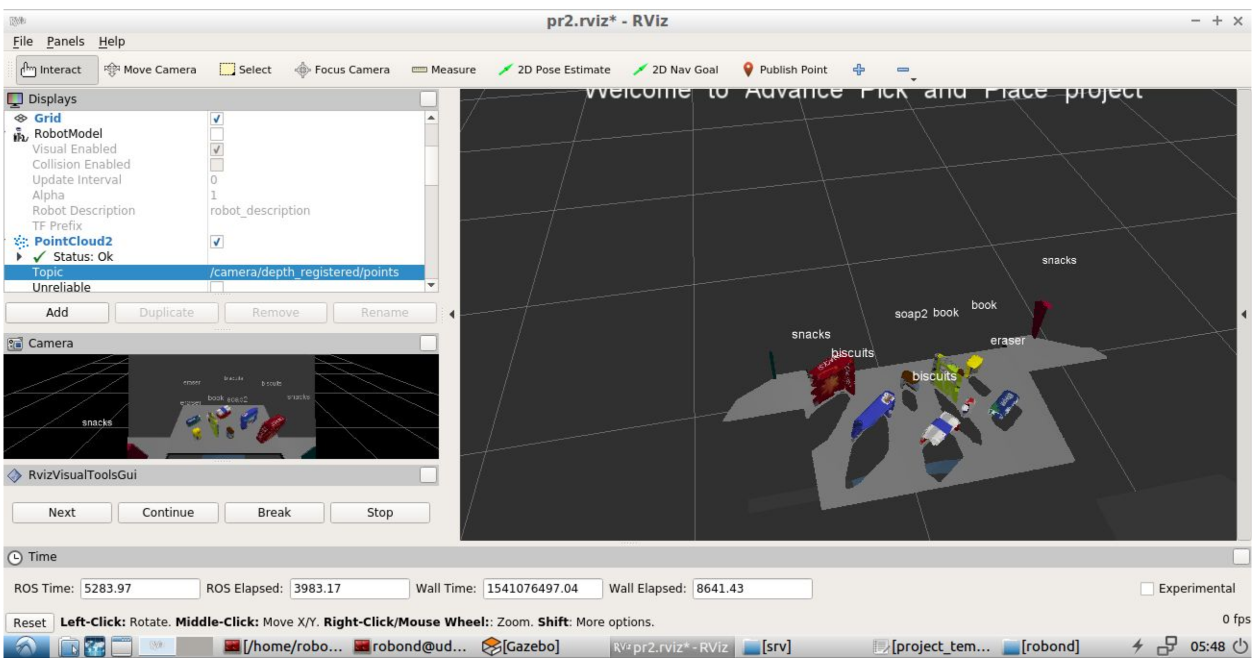
4- launch the Gazebo environment:

roslaunch pr2_robot pick_place_project.launch

5- run the object recognition script:

rosrun pr2_robot object_recognition.py

Image of predicted objects:



Issues faced during project:

1- 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 :

- I added this line to my .bashrc.

    export GAZEBO_MODEL_PATH=~/catkin_ws/src/RoboND-Perception-Project/pr2_robot/models:$GAZEBO_MODEL_PATH"

2- When compiling using catkin_make I used to get error "cannot convert to bool". I resolved it by adding static_cast<bool>().

Future improvements:

- Collision avoidance need to be done as described in project last section. If I have more time I will work on it.
- We can go through the object detection pipeline once after everytime we pick an object, this will give better results in crowded table case like test 3.