PR2 PROJECT: 3D PERCEPTION

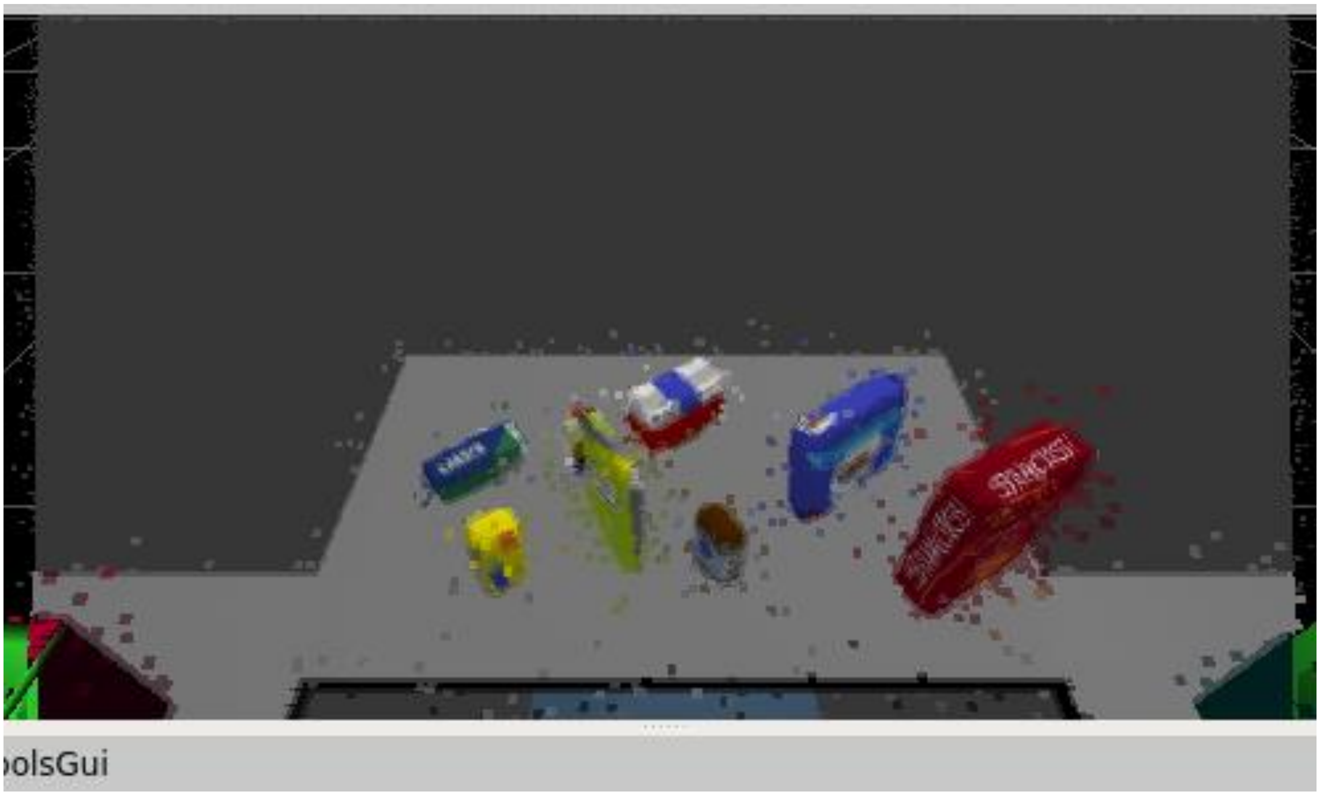
This is the project three (3) of Nanodegree Robotics Software Engineering at Udacity. This project uses PR2 Robot simulation outfitted with RGB-D camera to perceive world around it. This is accomplished by implementing perception pipeline from point cloud filtering, segmentation and clustering to object recognition.

1.0 Accessing the RGB-D Camera Data:

The first step in perception pipeline is to read data from RGB-D camera for manipulation. This is achieved by creating node that subscribed to the **/pr2/world/points** topic. Each time **/pr2/world/points** publishes ROS point cloud it is receive and handle by **pcl_callback** function where the perception pipeline is implemented:

```
# TODO: ROS node initialization
   rospy.init_node("clustering", anonymous=True)
   # TODO: Create Subscribers
```

```
pcl_sub = rospy.Subscriber("/pr2/world/points", pc2.PointCloud2, pcl_callback, queue_size=1)
```



Noisy Point Cloud from RGB-D Camera

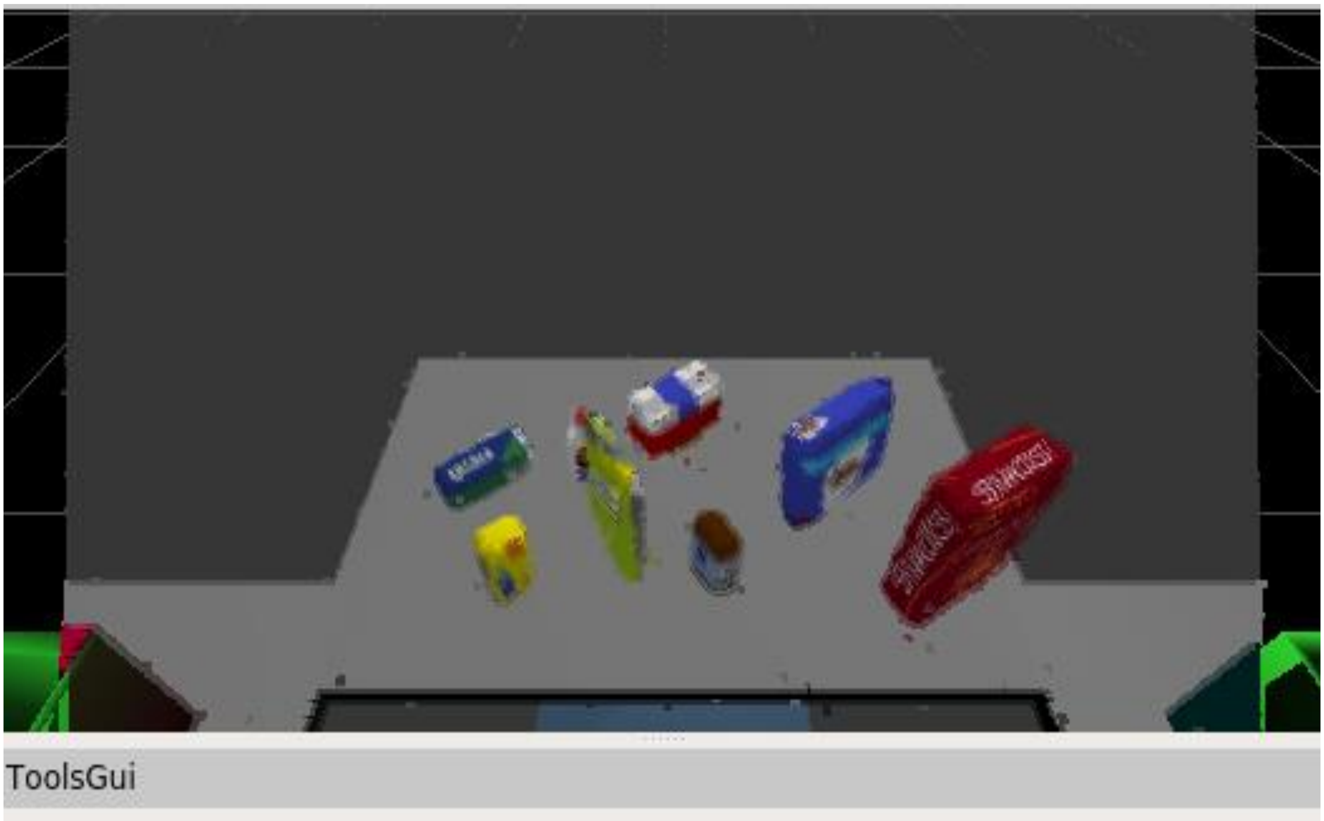
2.0 Point Cloud Filtering

Majority of the point cloud data are not useful for identifying the target object. Therefore, can be considered as noise. To remove such unnecessary and excessive data points as well as adversarial data I used point cloud filtering techniques which includes outlier removal filter, Voxel Grid filter, and pass through filter.

2.0.1 Outlier removal filter

Because the point cloud data from RGB-D comes with a lot of external noise due weather conditions such as dust, humidity, temperature and etc. In this project I alleviate this noise by using outlier removal filter technique.

```
# TODO: Statistical Outlier Filtering
    #create filter object
    outlier_filtered = pcl_cloud.make_statistical_outlier_filter()
    # Set number of neighboring points to analyse for a given points
    outlier_filtered.set_mean_k(20)
    # Set threshold scale factor
    x = 1.0
    # Consider any point with a mean distance than global mean (mean + x * std_dev)
    outlier_filtered.set_std_dev_mul_thresh(x)
    # Finally, call the filter function
    cloud_filtered = outlier_filtered.filter()
    filename = 'outlier_filtered.pcd'
    pcl.save(cloud_filtered,filename )
```



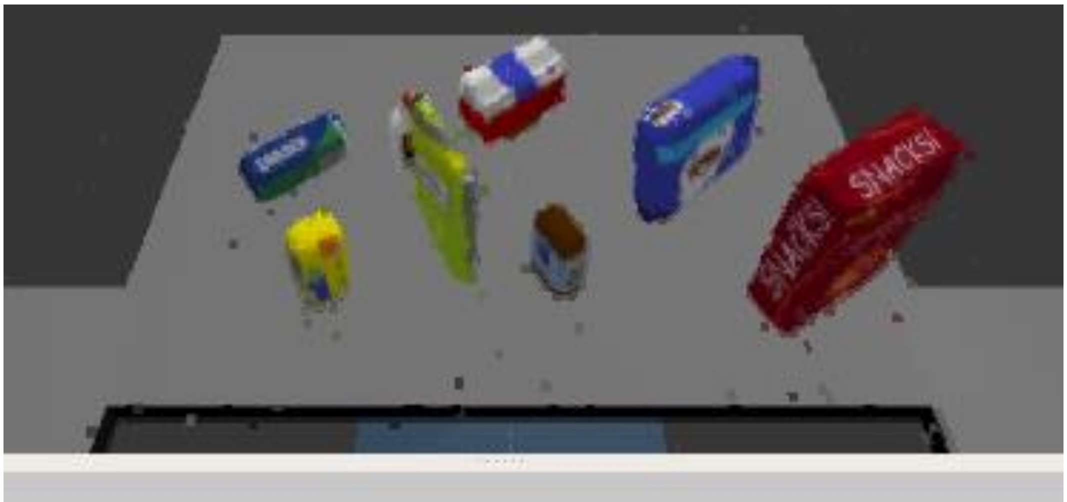
Point Cloud with noise or outlier removed

2.0.2 Voxel Grid filter Downsampling

Voxel Grid Filter Downsampling is a filtering algorithm use to improve the quality of a point cloud data and to derive a point cloud that has fewer points but still do a good job of representing the input point as a whole.

```
# TODO: Voxel Grid Downsampling

# Create a voxelGrid filter object the point cloud

vox = cloud_filtered.make_voxel_grid_filter()

# Choose a voxel or leaf size

LEAF_SIZE = 0.01

# Set the leaf size

vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
```

```
# call the filter function to obtain the resultant downsampled point cloud
cloud_downsampled = vox.filter()
filename= 'voxel_downsampled.pcd'
pcl.save(cloud_downsampled,filename)
```



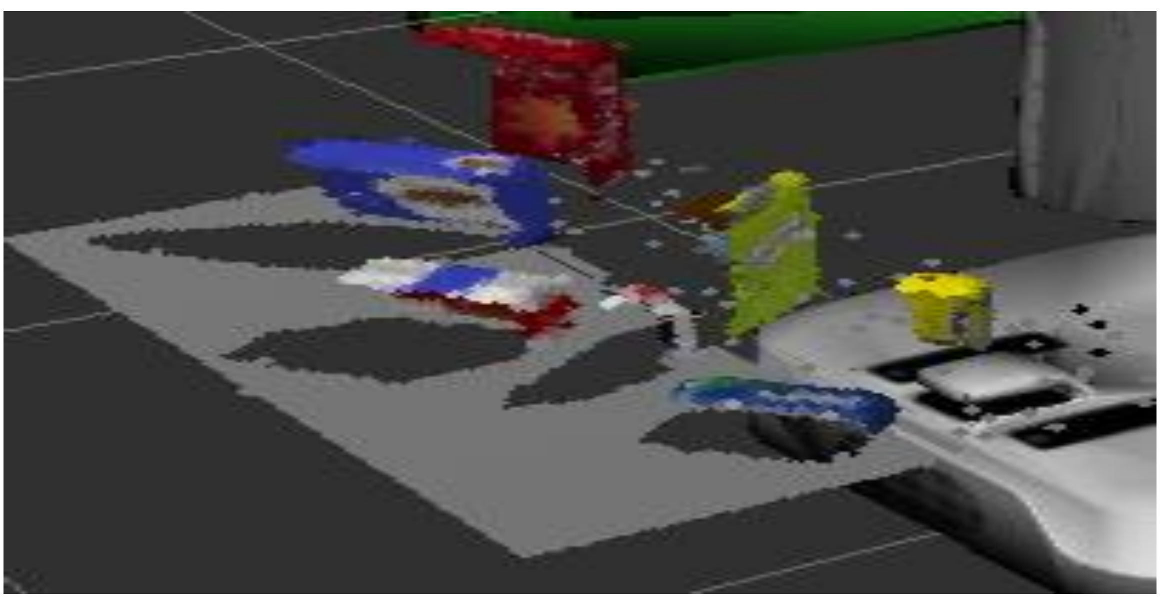
Downsampled Point Cloud

2.0.3 Pass through Filter

This is like cropping tool that remove useless data from the point cloud by specifying axis with cut-off values along that axis that allow only the region of interest to pass through. The region of interest in this scenario is just the table and the objects on the top of the table. The PR2 robot simulation required pass through filters for both Y and Z axis. I used a range of -0.4 and 0.4 and 0.61 and 0.9 for Y and Z axis respectively.

```
# TODO: PassThrough Filter
    # Create passthrough filter object
    passthrough = cloud_downsampled.make_passthrough_filter()
    # Assign axis and range to the passthrough filter object for z axis
    filter_axis='z'
```

```
passthrough.set_filter_field_name(filter_axis)

axis_min = 0.61

axis_max = 0.9

passthrough.set_filter_limits(axis_min,axis_max)

cloud_passed = passthrough.filter()

# Assign axis and range to the passthrough filter object for y axis

passthrough = cloud_passed.make_passthrough_filter()

filter_axis='y'

passthrough.set_filter_field_name(filter_axis)

axis_min = -0.4

axis_max = +0.4

passthrough.set_filter_limits(axis_min,axis_max )

cloud_passed = passthrough.filter()

filename ='pass_through_cloud_filtered.pcd'

pcl.save(cloud_passed, filename)
```



Passthrough Point Cloud

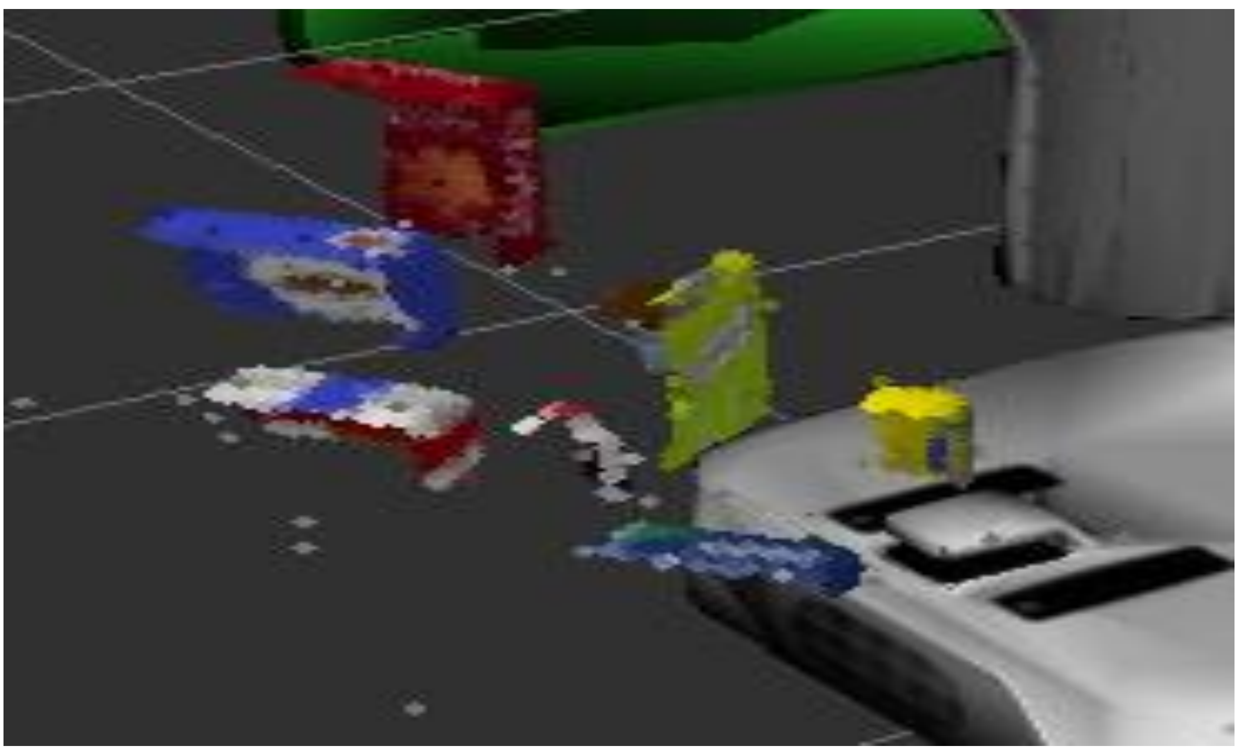
3.0 RANSAC Plane Segmentation

In this step of perception pipeline, there is need to remove the table itself from the scene. To do so, I used technique called Random Sample Consensus. This technique or algorithm used to identify points in the dataset that belong to particular model such as a cylinder, a box, or any other common shape. In this scenario the model I choose is the plane which represent the table.

The RANSAC algorithm assumes that all of the data in dataset is comprised of both inliers and outliers, where inliers can be defined by a particular model with a specific set of parameters, while outliers do not fit that model and hence, can be discarded. I used model of plane and RANSAC maximum distance value of 0.01.

```
# TODO: RANSAC Plane Segmentation
    # Create the segmentation object
    seg = cloud_passed.make_segmenter()
    # Set model you wish to fit
    seg.set_model_type(pcl.SACMODEL_PLANE)
    seg.set_method_type(pcl.SAC_RANSAC)
    # Set maximum distant to be considerred for fitiing
    max_distance = 0.01
    seg.set_distance_threshold(max_distance)
    # Call the segment function to obtain set of inliers indices and model coefficients
    inliers, coefficients = seg.segment()


    # Extract inliers
    cloud_table = cloud_passed.extract(inliers, negative=False)
```

```
filename = 'extracted_inliers.pcd'
# Save pcd for table
pcl.save(cloud_table, filename)


# Extract outliers


cloud_objects = cloud_passed.extract(inliers, negative=True)
filename = 'extracted_outliers.pcd'
# Save pcd for tabletop objects
pcl.save(cloud_objects, filename)
```



Objects Point Cloud (outliers)

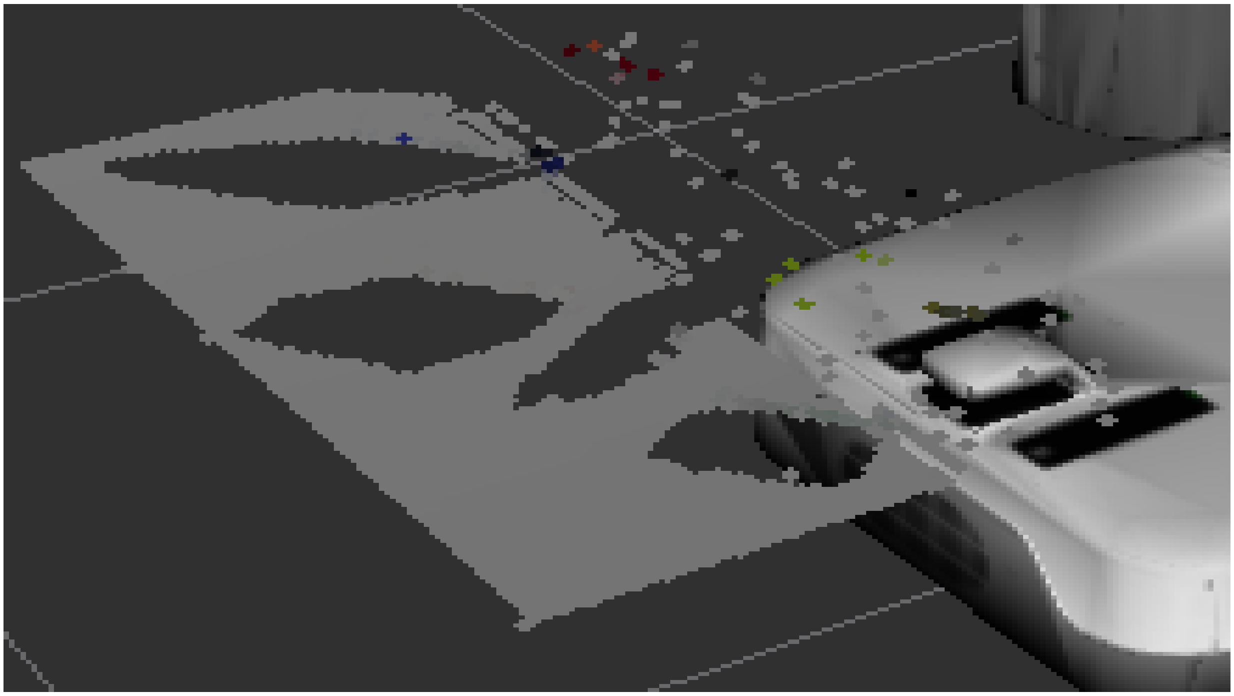


Table Point Cloud (Inliers)

4.0 Clustering for Segmentation

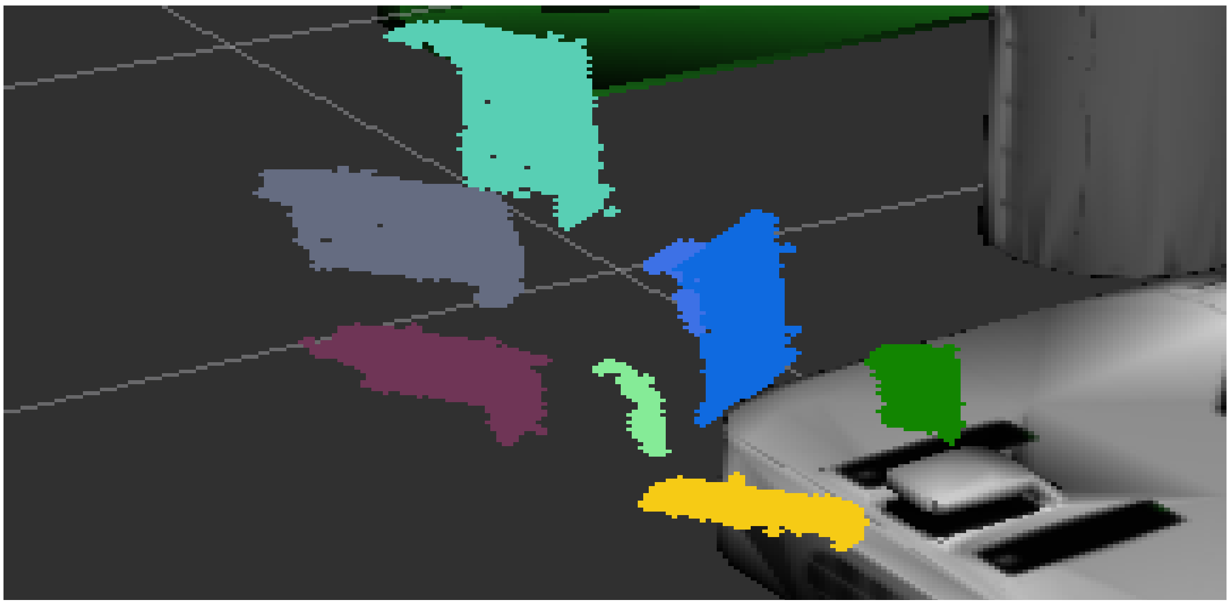
So far, we've used RANSAC Plane Fitting to remove table from the scene where we have been relying solely on object shape to perform segmentation. However, our dataset contain feature rich color information that can be combined with shape information to perform complex segmentation task. Here I've used clustering which allowed us to segment object in our point cloud without having assume a model shape. Clustering finds points in the dataset and group them together based on particular features such as color, position, texture or a combination of many features.

To implement clustering I've used Euclidean clustering (DBSCAN) to identified the object from one another, Euclidean clustering unlike K-means you do not need to know how many clusters to expect from the dataset, all you need to know is the density of the data in question.

```
    # TODO: Euclidean Clustering

        # Apply function to convert XYZRGB to XYZ
```

```
white_cloud = XYZRGB_to_XYZ(cloud_objects)

# Contruct k-d tree

tree = white_cloud.make_kdtree()

# Create a cluster extraction object

ec = white_cloud.make_EuclideanClusterExtraction()

# set tolearance for diistance threshold as well as maximum cluster size

ec.set_ClusterTolerance(0.02)

ec.set_MinClusterSize(10)

ec.set_MaxClusterSize(2800)

# Search the k-d tree for cluster

ec.set_SearchMethod(tree)

# Extract indices for each of the discovered cluster

cluster_indices = ec.Extract()



# TODO: Create Cluster-Mask Point Cloud to visualize each cluster
separately

# Assign a color to each segmented object in the 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 cluster to contain all cluters each with unique color
    cluster_cloud = pcl.PointCloud_PointXYZRGB()
    cluster_cloud.from_list(color_cluster_point_list)
    filename ='cluster_cloud.pcdpcl.save(cluster_cloud,filename)
```



Euclidean Cluster Segmentation

5.0 Object recognition

Now that we have a point cloud broken into individual object. Object recognition allow each object to be identified based on color and spatial information.

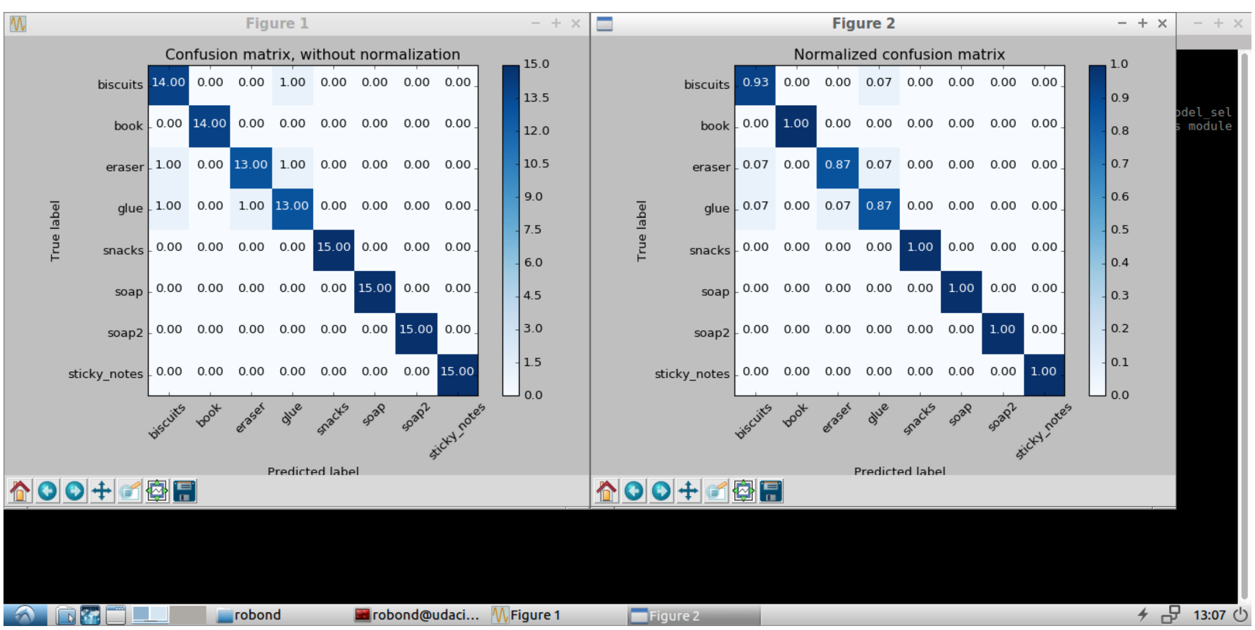
5.0.1 Capture Object Features

Color and normal histogram help the system to convert color and shape information into features that robot used for classification. I ran **capture_feature.py** script for feature extraction. The script spawned each object in random orientation and captured features based on the point cloud resulting from each of the random orientation. The **capture_feature.py** script save the object features in a file **training_set.sav.** It captured each object in 15 random orientations, using HSV color space and 32 bins.

```
# Extract histogram features
        chists = compute_color_histograms(sample_cloud, using_hsv=True)
        normals = get_normals(sample_cloud)
        nhists = compute_normal_histograms(normals)
        feature = np.concatenate((chists, nhists))
        labeled_features.append([feature, model_name])
```

5.0.2 Train SVM Model

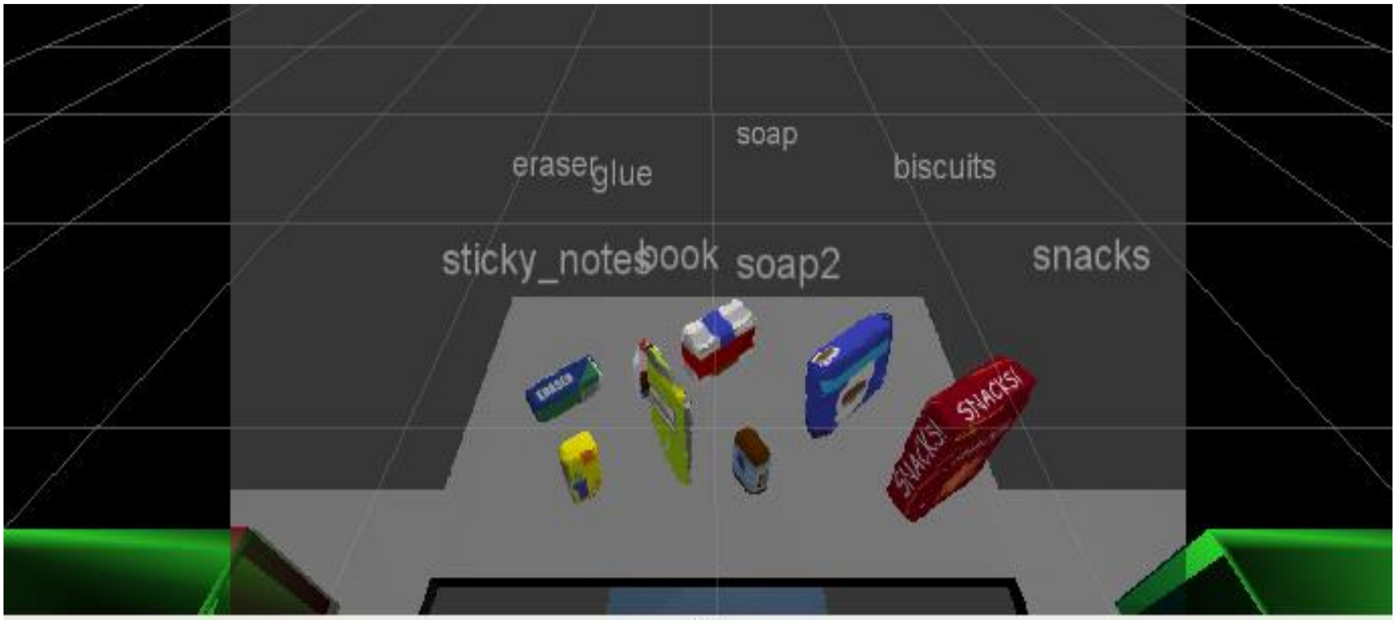
Support Vector Machine (SVM) is utilized to trained the model. The SVM apply an iterative method to a training dataset generated from capture_features.py script. After having train the model by running train_svm.py script I was able to get accuracy score of 95.7% and the model is saved as **model.sav.** Below is confusion matrix for the trained model:



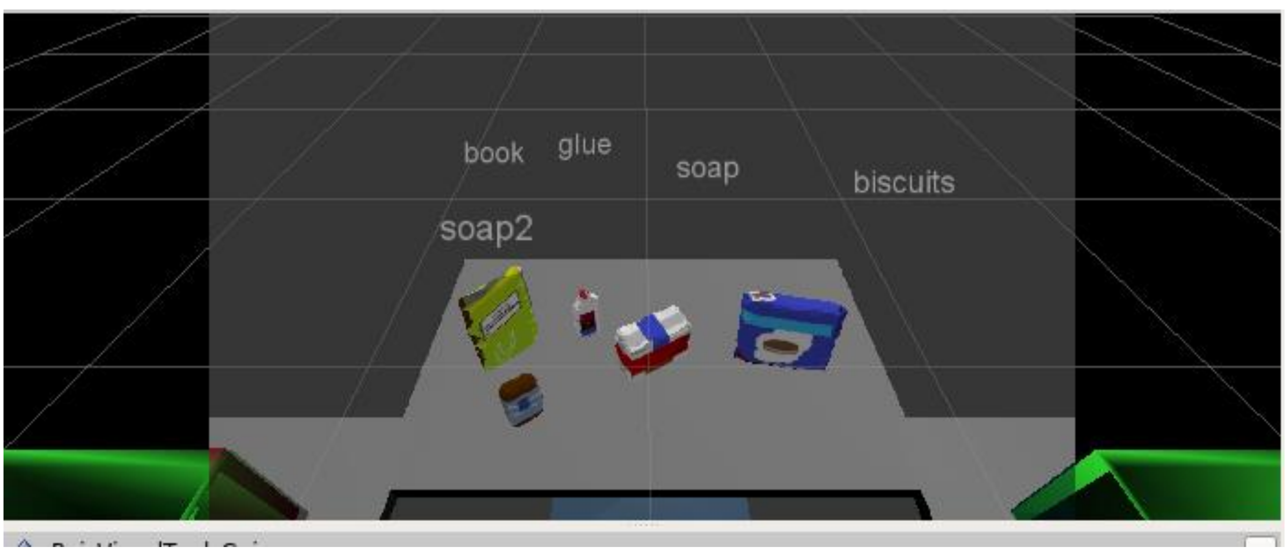
5.0.3 Object recognition Result

The model has done a very good job in predicting the all eight (8) objecst correctly. The recognition is achieved by iterating each cluster then compute its feature vector and make appropriate prediction.

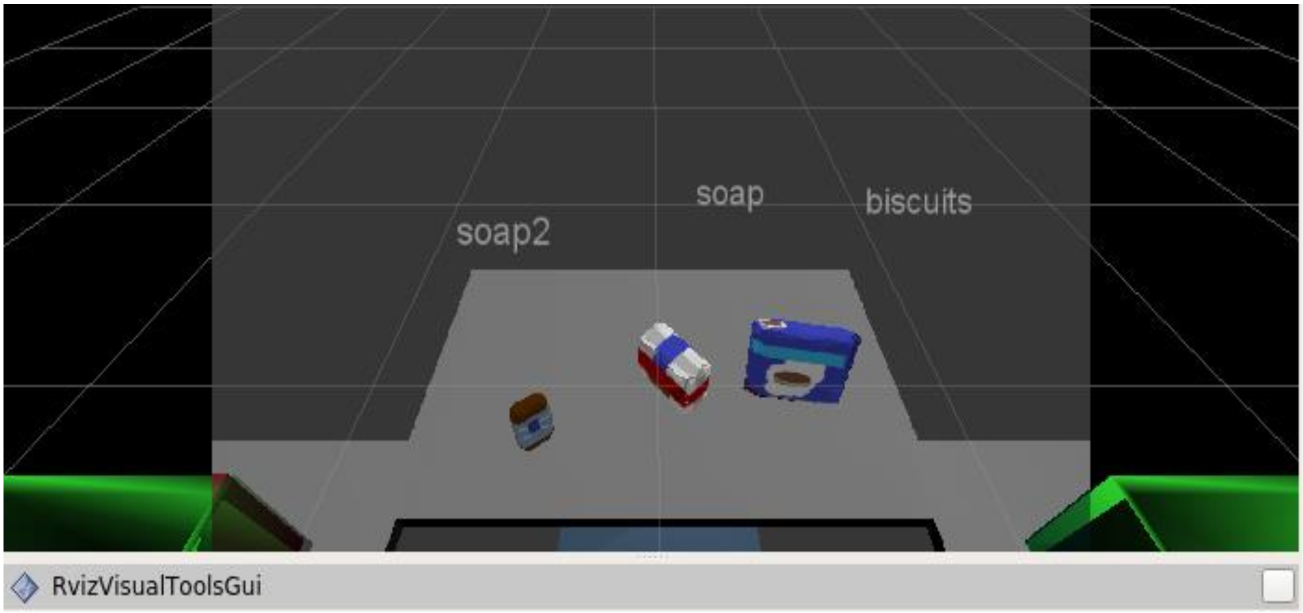
```
# Classify the clusters! (loop through each detected cluster one at a time)
    detected_object_labels = []
    detected_objects = []
    for index, point_list in enumerate(cluster_indices):
        # Grab the points for the cluster from the extracted outlier (cloud_objects)
        pcl_cluster = cloud_objects.extract(point_list)
        # convert cluster from pcl to ROS
        ros_cluster = pcl_to_ros(pcl_cluster)
        # Compute the associated feature vector
        color_hists = compute_color_histograms(ros_cluster, using_hsv=True)
        normals = get_normals(ros_cluster)
        normals_hists = compute_normal_histograms(normals)
        hists_feature = np.concatenate((color_hists, normals_hists))
        # Make the prediction
        # Retrieve the label for the result and add it to detection_objects_label
        prediction = clf.predict(scaler.transform(hists_feature.reshape(1, -1)))
        label = encoder.inverse_transform(prediction)[0]
        detected_object_labels.append(label)
```



World 3



World 2



World 1

6.0 Output Yaml Files

6.0.1 Reading Parameters

The object list and dropbox locations were read form the parameter server and then store in the   dictionary object for easy manipulation.

```
# TODO: Get/Read parameters
    object_list_param = rospy.get_param('/object_list')
    dropbox_list_param = rospy.get_param('/dropbox')


    # TODO: Parse parameters into individual variables
    object_param_dict = {}
    for idx in range(0,len(object_list_param)):
        object_param_dict[object_list_param[idx]['name']] = object_list_param[idx]
        print object_list_param[idx]
    dropbox_param_dict = {}
    for idx in range(0,len(dropbox_list_param)):
```

```
    dropbox_param_dict[dropbox_list_param[idx]['group']] = dropbox_list_param[idx]
```

6.0.2 Calculating Centroid and Pose messages

For each item in the list I identified its associated object in the scene and calculate an object's centroid by averaging positons of all the points in the point cloud and then retrieve its group to determine in which dropbox the object will be dropped.

```
# TODO: Loop through the pick list
  for object in object_list:


      # TODO: Get the PointCloud for a given object and obtain it's centroid
      point_arr = ros_to_pcl(object.cloud).to_array()
      centroid = np.mean(point_arr, axis=0)[:3]
      print object_param_dict[object.label]
      # Get config param for that kind of object
      object_param = object_param_dict[object.label]

      # Get config param for that kind of object
      dropbox_param = dropbox_param_dict[object_param['group']]

      object_name.data = str(object.label)

      # TODO: Create 'pick_pose' for the object
      pick_pose.position.x = np.asscalar(centroid[0])
      pick_pose.position.y = np.asscalar(centroid[1])
      pick_pose.position.z = np.asscalar(centroid[2])
      pick_pose.orientation.x =0.0
      pick_pose.orientation.y =0.0
```

```
        pick_pose.orientation.z =0.0

        pick_pose.orientation.w =0.0


        # TODO: Create 'place_pose' for the object

        position = dropbox_param['position'] + np.random.rand(3)/10

        pick_pose.position.x = float(position[0])

        pick_pose.position.y = float(position[1])

        pick_pose.position.z = float(position[2])

        pick_pose.orientation.x =0.0

        pick_pose.orientation.y =0.0

        pick_pose.orientation.z =0.0

        pick_pose.orientation.w =0.0


        # TODO: Assign the arm to be used for pick_place

        arm_name.data = str(dropbox_param['name'])


        # TODO: Create a list of dictionaries (made with make_yaml_dict()) for later output to yaml
format

        yaml_dict_list = make_yaml_dict(test_scene_num, arm_name, object_name, pick_pose,
place_pose)

        dict_list.append(yaml_dict_list)
```

6.0.3 Create Yaml Output Files

Finally call send_to_yaml helper method to create the yaml output files .

```
# TODO: Output your request parameters into output yaml file

    yaml_filename ="output" + str(test_scene_num.data) + ".yaml"

    send_to_yaml(yaml_filename, dict_list)
```

7.0 Conclusion

I find this PR2 Project very interesting because it give necessary knowledge and tools to implement robotics perception pipeline which I believe apply in future to solve in real world problems.