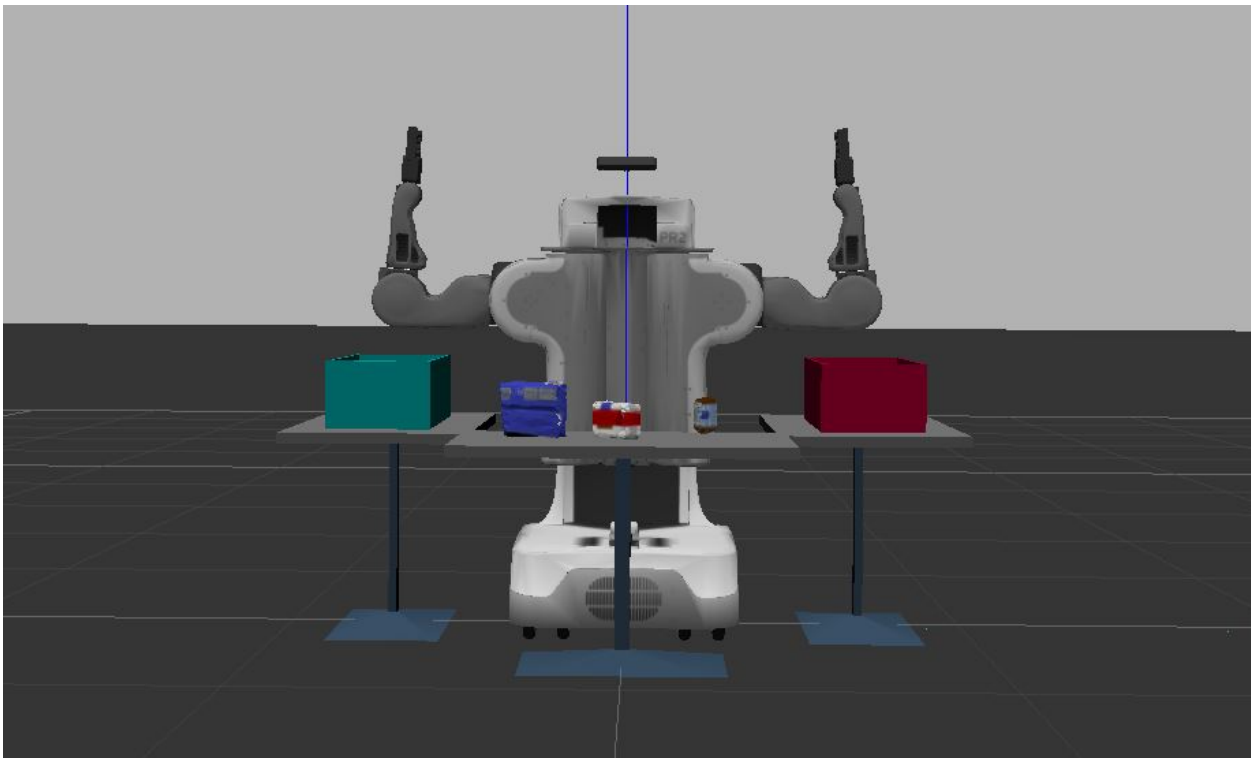
3D Perception Project

By

Leroy Friesenhahn



The 3D perception project combined all the process from 3D perception exercises 1, 2 and 3 to perform a pick and place operation using a PR2

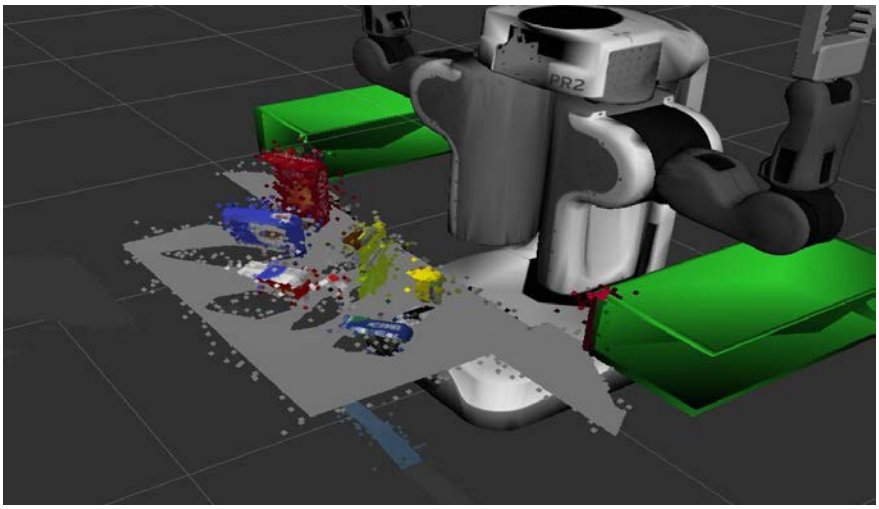
Robot Simulation. The process of identifying an object consist of several step as explained in Calibration, Filtering, and Segmentation

Downsampling

The first step is to reduce the amount of data (pixels) being processed. This can be done without reducing the amount of information available.

```
# TODO: Voxel Grid Downsampling
vox = cloud.make_voxel_grid_filter()
# chose a voxel alos known as a leaf
# change from 0.01 to 0.005
LEAF_SIZE = 0.005

# call the filter function to obtain the resultant downsample point cloud
vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
cloud_downsample = vox.filter()
```

The LEAF_SIZE is in meters. This value is changed to meet the criteria of the image

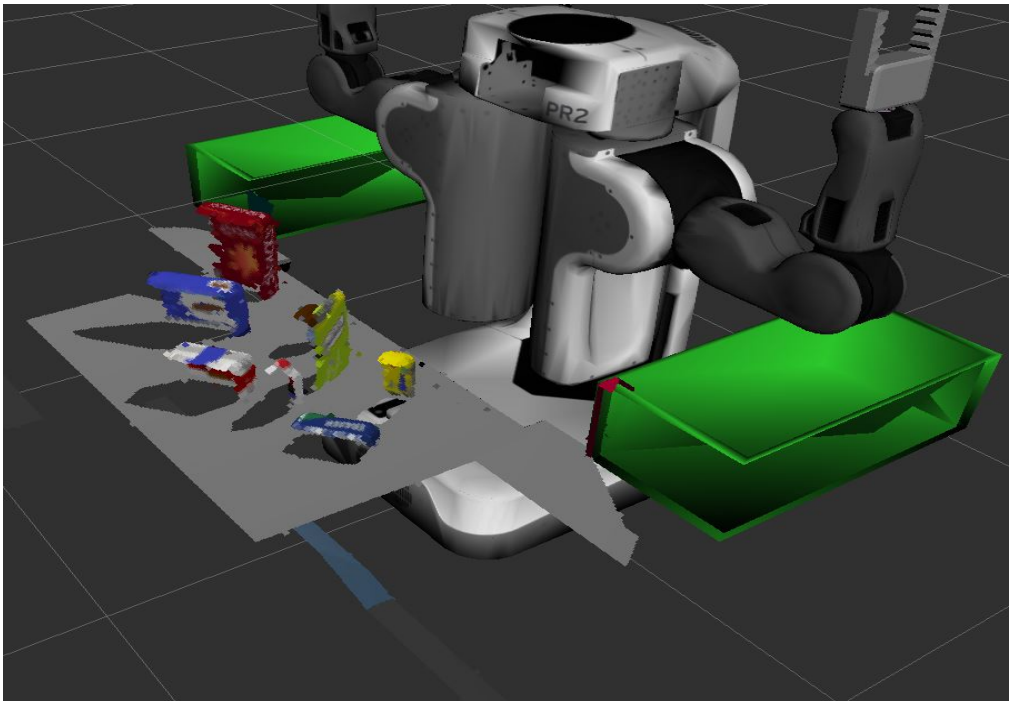


Downsampling Results

Outlier Removal Filter

The outlier removal filter is applied to remove noise and irregularities by comparing a point with points located within a mean distance.

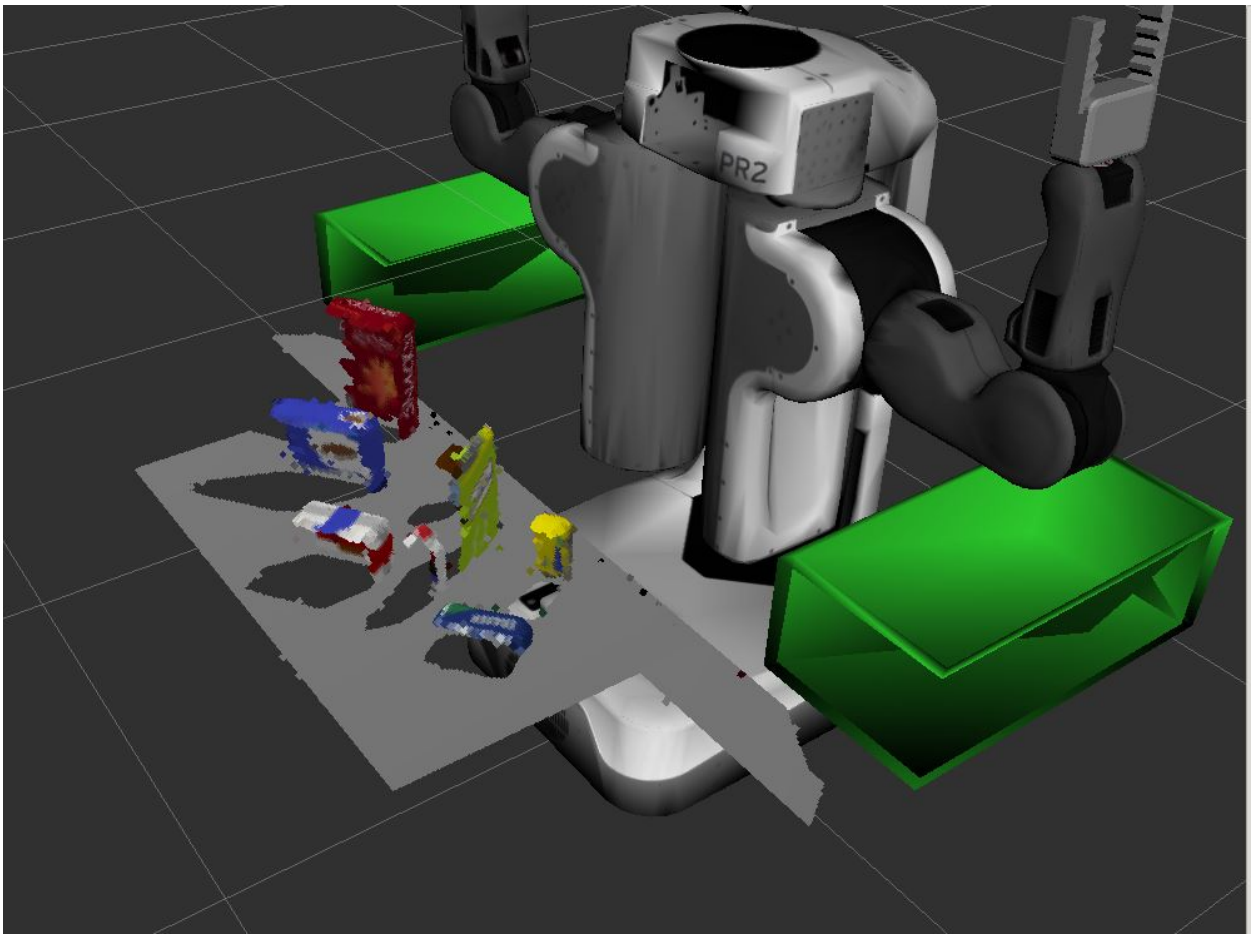
```
# Outlier Removal Filter
# Creating a filter object:
outlier_filter = cloud_downsample.make_statistical_outlier_filter()
# Set the number of neighboring points to analyze for any given point
# previous  from 5 to 8
outlier_filter.set_mean_k(8)
# Set threshold scale factor
# changed from 0.1 to 0.3
x = 0.3
# Any point with a mean distance larger than global (mean distance+x*std_dev) will be considered outlier
outlier_filter.set_std_dev_mul_thresh(x)
# Ccall the filter function for magic
outliers_removed = outlier_filter.filter()
```



Outlier Removal Filter

Pass Through Filter

Pass through Filters are used to remove unnecessary data from your point cloud. This is used when the information of interest is located within a certain area of the image



Pass through filter rssults

```
# TODO: PassThrough Filter
passthrough = outliers_removed.make_passthrough_filter()
# assign axis and range to the passthrough filter
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)
passthrough_z = passthrough.filter()
# Limiting on the Y axis too to avoid having the bins recognize
passthrough = passthrough_z.make_passthrough_filter()
# Assign axis and range to the passthrough filter object.
# change from y to x
filter_axis = 'x'
passthrough.set_filter_field_name(filter_axis)
# change min from -0.45 to 0.34
# change max from +0.45 to 1.0
axis_min = 0.34
axis_max = 1.0
passthrough.set_filter_limits(axis_min, axis_max)

#use the filter function to obtain the resultant point cloud
cloud_passthrough = passthrough.filter()
```

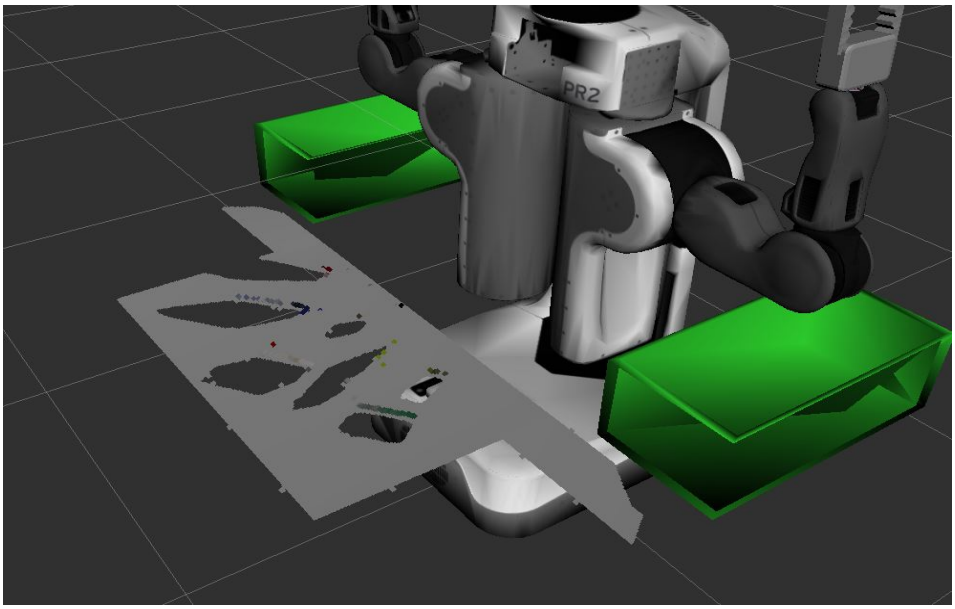
The filter can be modified to climate data in the x, y or z directions. The area of the filter can be change by applying a min and max value in one of the selected directions.

RANSAC Plane Fitting

The RANSAC algorithm involves performing two iteratively repeated steps on a given data set. A minimal subset containing the smallest number of points required to uniquely a model. In this case, 3 non-collinear points are needed to determine a plane.

```
# TODO: RANSAC Plane Segmentation
seg = cloud_passthrough.make_segmenter()
seg.set_model_type(pcl.SACMODEL_PLANE)
seg.set_method_type(pcl.SAC_RANSAC)

# Set max distance for a point to be considered fitting the model
max_distance = 0.01
seg.set_distance_threshold(max_distance)
inliers, coefficients = seg.segment()
```

A max distance for a point to be considered for the model can be set using the max_distance value.



RANSAC plane fitting

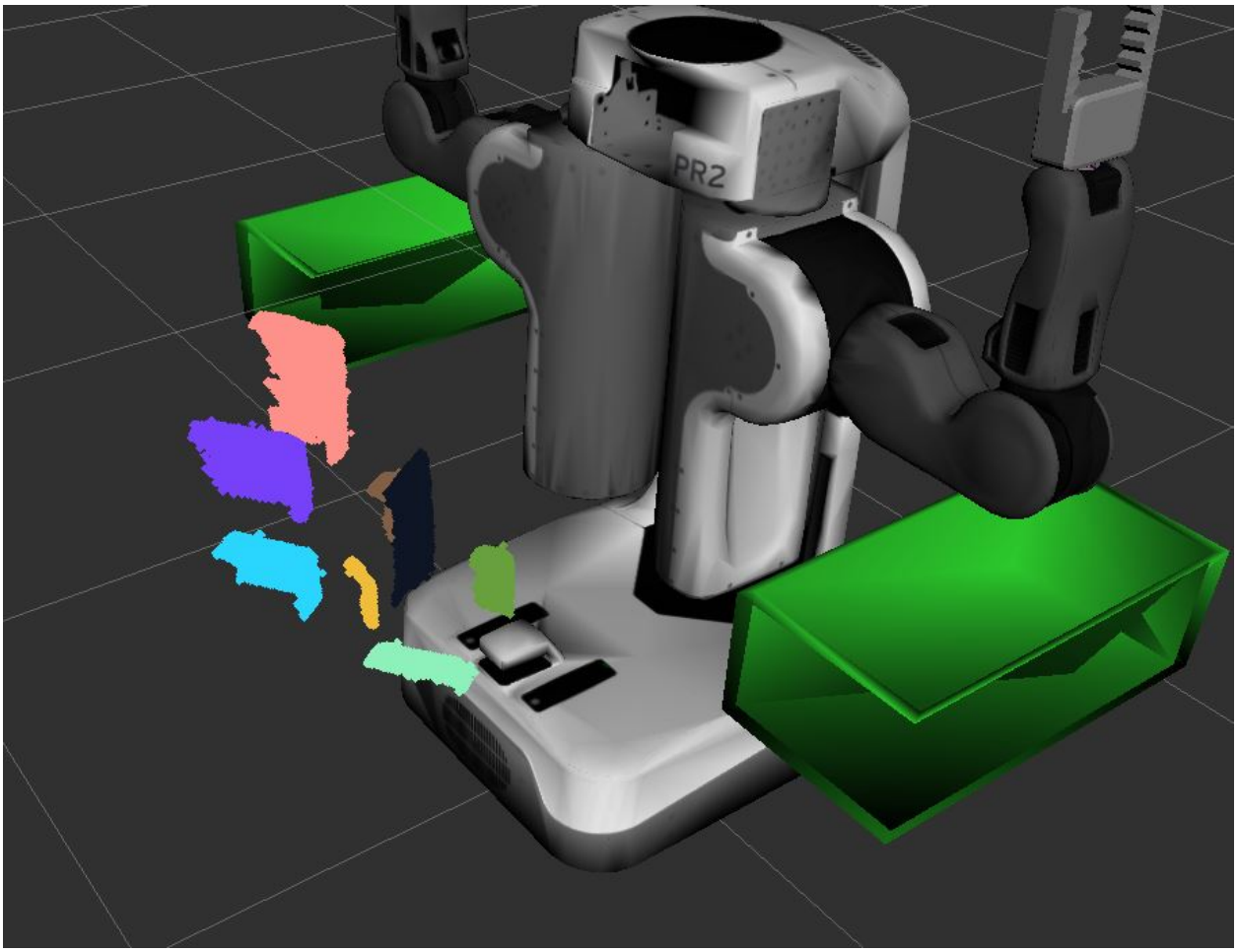
Extracting Indices

Using RANSAC, one has identified the indices in the point cloud. If the Pass Through Filter has been applied correctly, the indices representing the table and the objects have been identified. As a result the objects can be extracted.

```
# TODO: Extract inliers and outliers
# Exctract inliers - tabletop
cloud_table = cloud_passthrough.extract(inliers, negative=False)
# Extract outliers - objects
cloud_objects = cloud_passthrough.extract(inliers, negative=True)
```

Euclidean Clustering

```
# TODO: Euclidean Clustering
white_cloud = XYZRGB_to_XYZ(cloud_objects)
tree = white_cloud.make_kdtree()

# Create a cluster extraction object
ec = white_cloud.make_EuclideanClusterExtraction()

# Set tolerance for extraction
ec.set_ClusterTolerance(0.02)
# change from 10 to 40
ec.set_MinClusterSize(40)
# change 2500 to 4000
ec.set_MaxClusterSize(4000)

# Search the k-d tree for clusters
ec.set_SearchMethod(tree)

# Extract indices for each of the discovered clusters
cluster_indices = ec.Extract()
```



Objects cluster

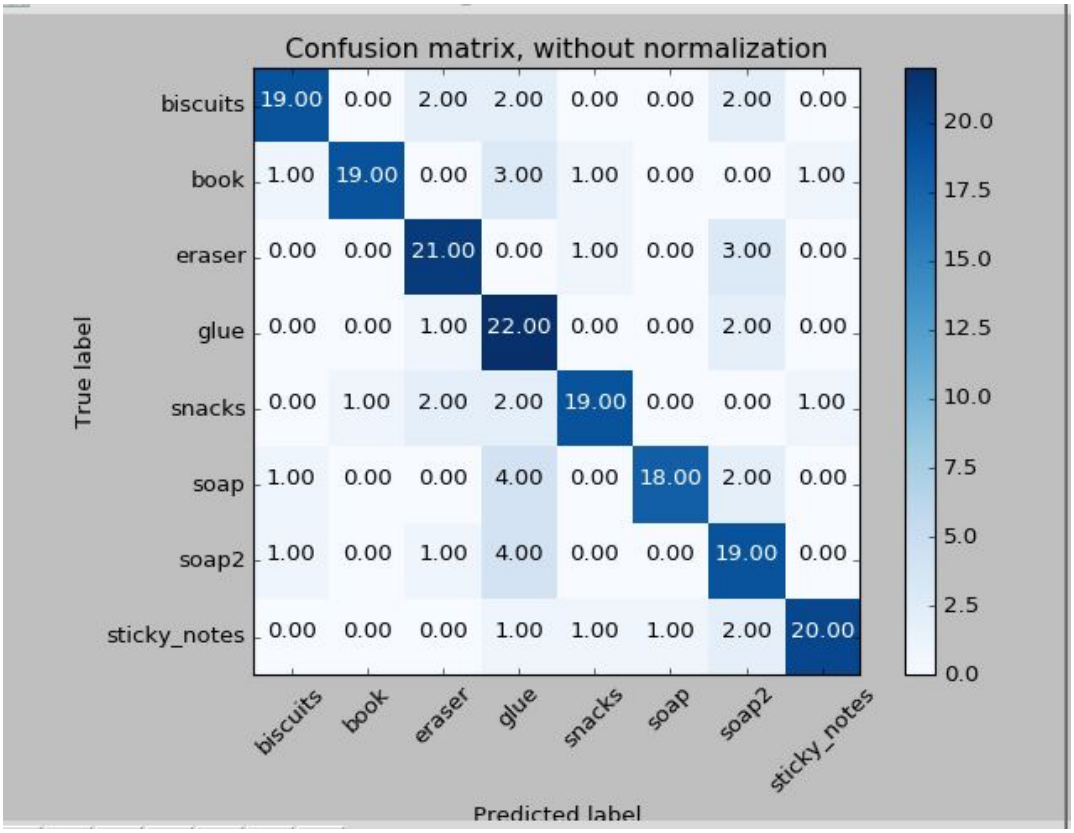
Classify the Clusters

Now that the object have been identified, the process is to use the cluster to predict the type of object. The clf.predict function uses the image dataset (previously generated) to determine the object.

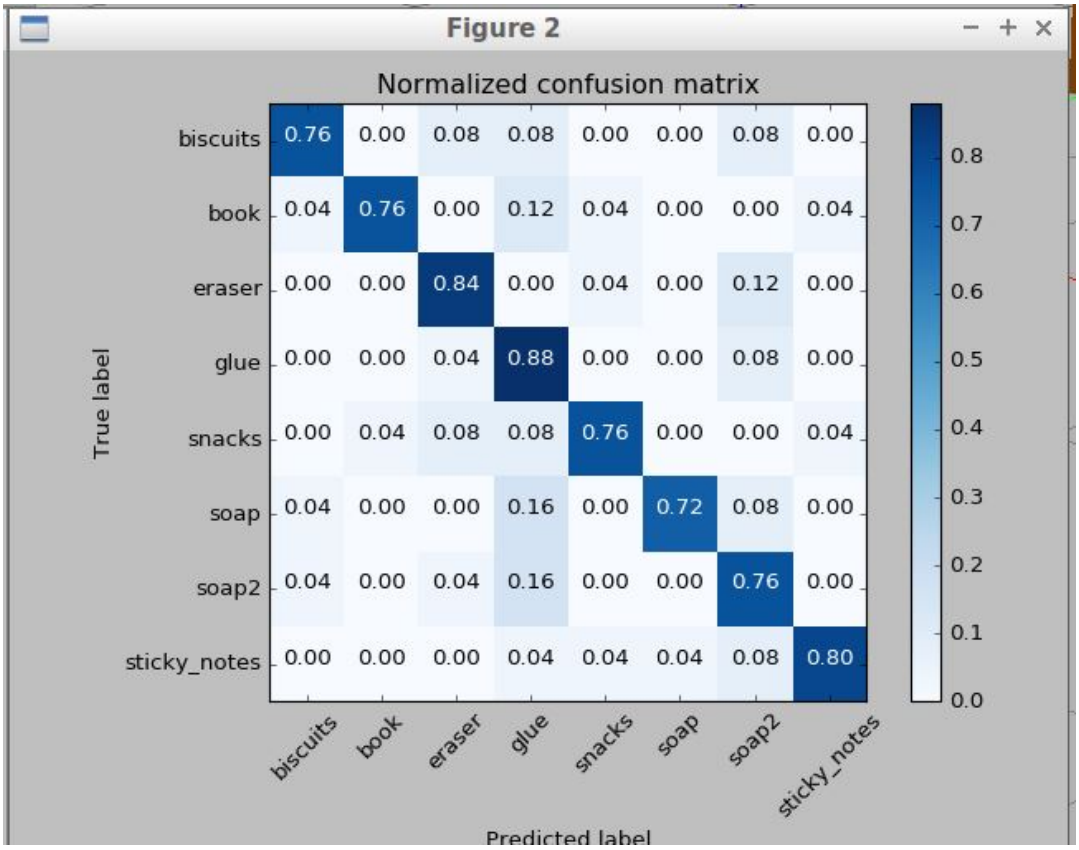
```
# Classify the clusters! (loop through each detected cluster one at a time)
detected_objects_labels = []
detected_objects = []

for index, pts_list in enumerate(cluster_indices):

    # Grab the points for the cluster
    pcl_cluster = cloud_objects.extract(pts_list)

    # convert the cluster from pcl to ROS
    ros_cluster = pcl_to_ros(pcl_cluster)

    # Compute the associated feature vector
    # extract the histogram features
    chists = compute_color_histograms(ros_cluster, using_hsv=True)
    normals = get_normals(ros_cluster)
    nhists = compute_normal_histograms(normals)
    feature = np.concatenate((chists, nhists))

    # Make the prediction
    prediction = clf.predict(scaler.transform(feature.reshape(1, -1)))

    # get the label
    label = encoder.inverse_transform(prediction)[0]
    detected_objects_labels.append(label)
    label_pos = list(white_cloud[pts_list[0]])
    label_pos[2] += .4
    # Publish a label into RViz
    object_markers_pub.publish(make_label(label, label_pos, index))

    # Add the detected object to the list of detected objects.
    do = DetectedObject()
    do.label = label
    do.cloud = ros_cluster
    detected_objects.append(do)
```

Creating the Image Dataset(Training the SVM)

In order to identify each cluster image, a training dataset is created. The Support Vector Machine using the program capture_features.py creates multiple view of images to be identified. The images are converted to histograms and normalized to provide the basis for predicting the image. This program was provided lack any histograms processing to improve the accuracy of identifying the required images. Below is a sample of the added histogram code and a result of the images processed.



Training SVM Confusion Matrix without normalization



Training SVM Normalized confusion matrix

Project Requirements:

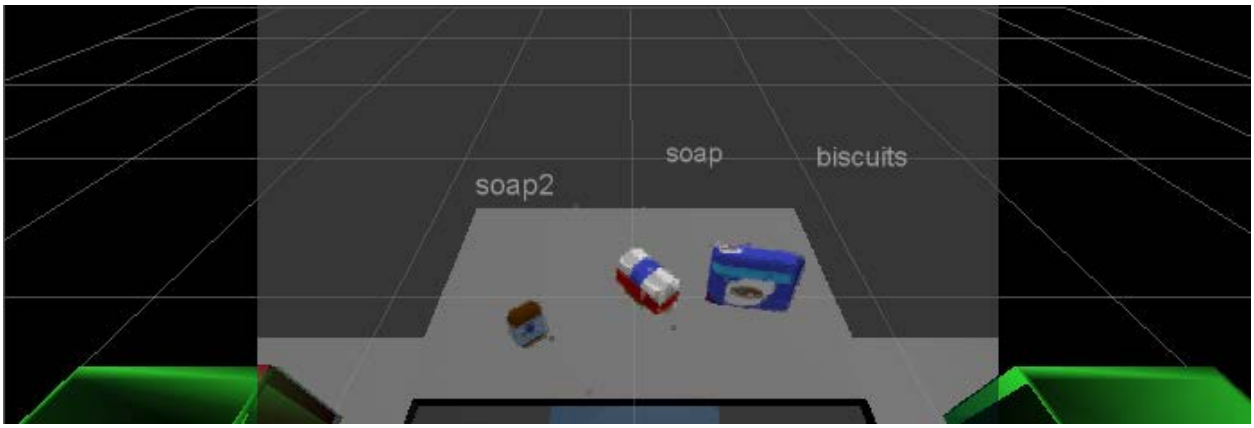
To meet the requirements of this project the following must be accomplished:

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

The first attempt at achieving these results failed. Attempt two improved upon tweaking the different image filters. The third attempt exceeded the required results. This was accomplished by enhancing the SVM program by adding the histogram processes and increasing the number of samples per object to 25.

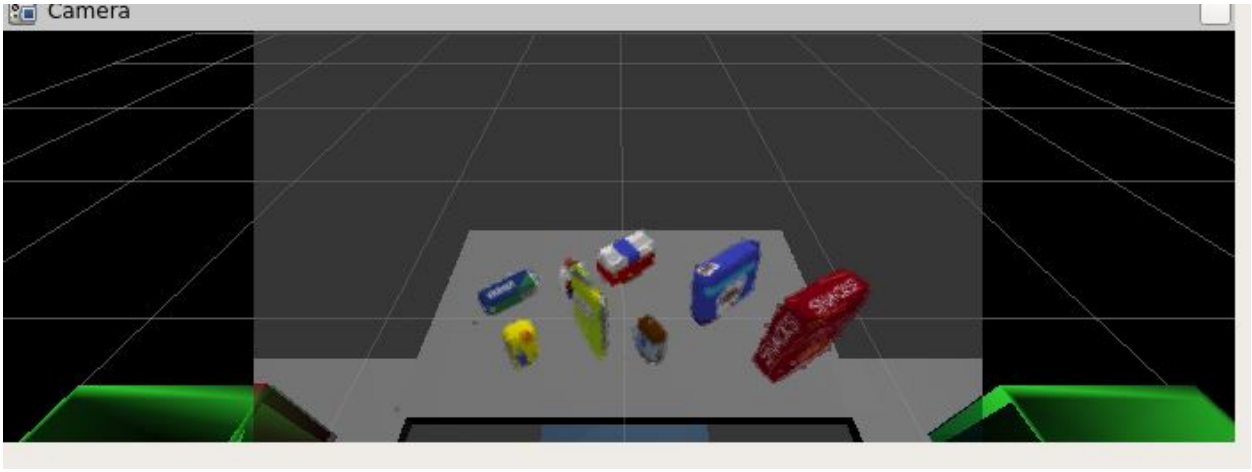
The result of the individual runs can be viewed as follows:

Test1.world output_1.yawl



Test2.world output_2.yawl

Test3.world output_3.yawl



Start of Test3.world

3D perception Program project_template.py

Training Program feature.py

Summary

This was a very interesting and time consuming project. With no absolute correct filter values, time was spent in trial and error find the best values. I have a great interest in computer vision so the time spend was frustrating but rewarding. I hope to find additional time to pursue the extra assignment included with the project.