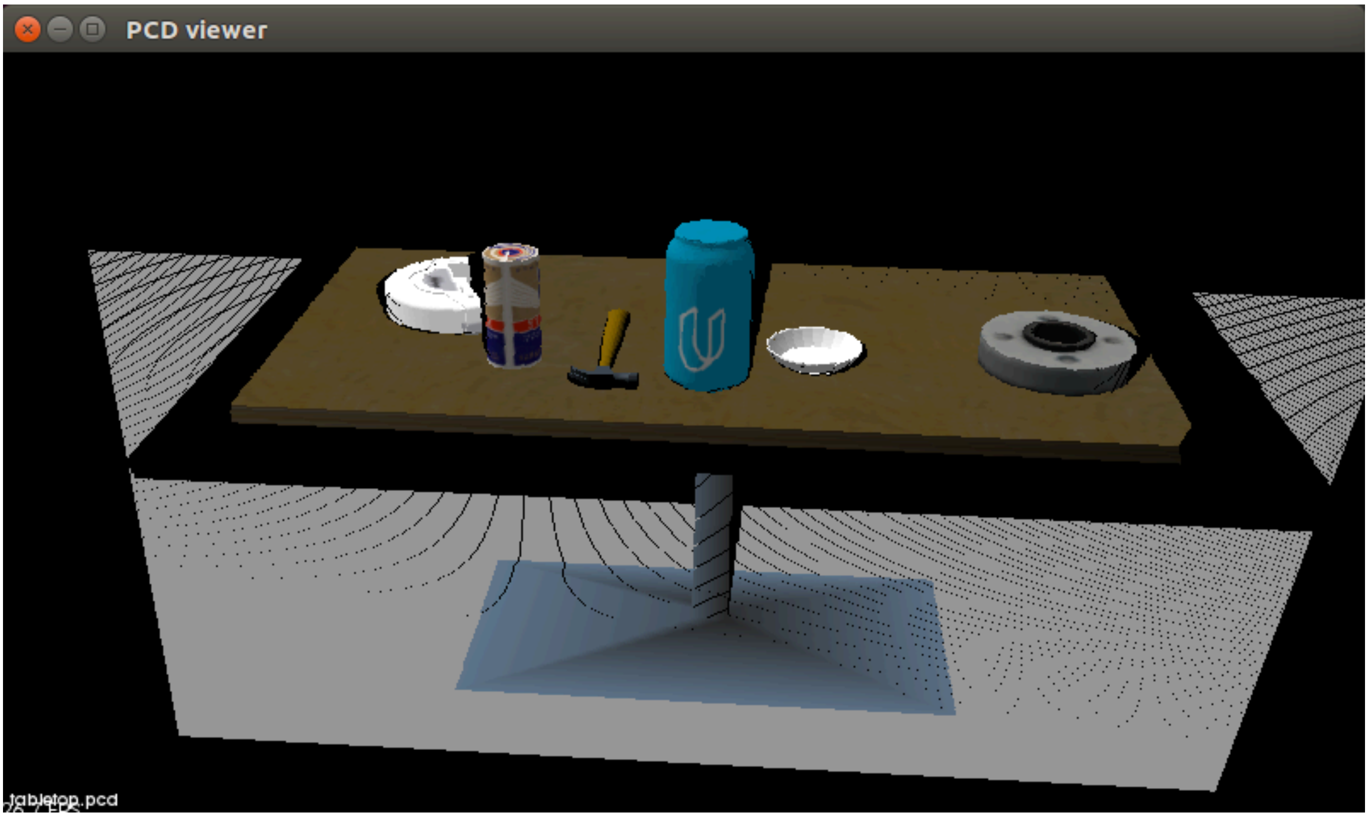
Write-up for the Project 3: 3D Perception

Exercises

Exercise 1

We document here the code used in the implementation of the Exercise 1. This is the starting image as provided in the exercise (tabletop.pcd):

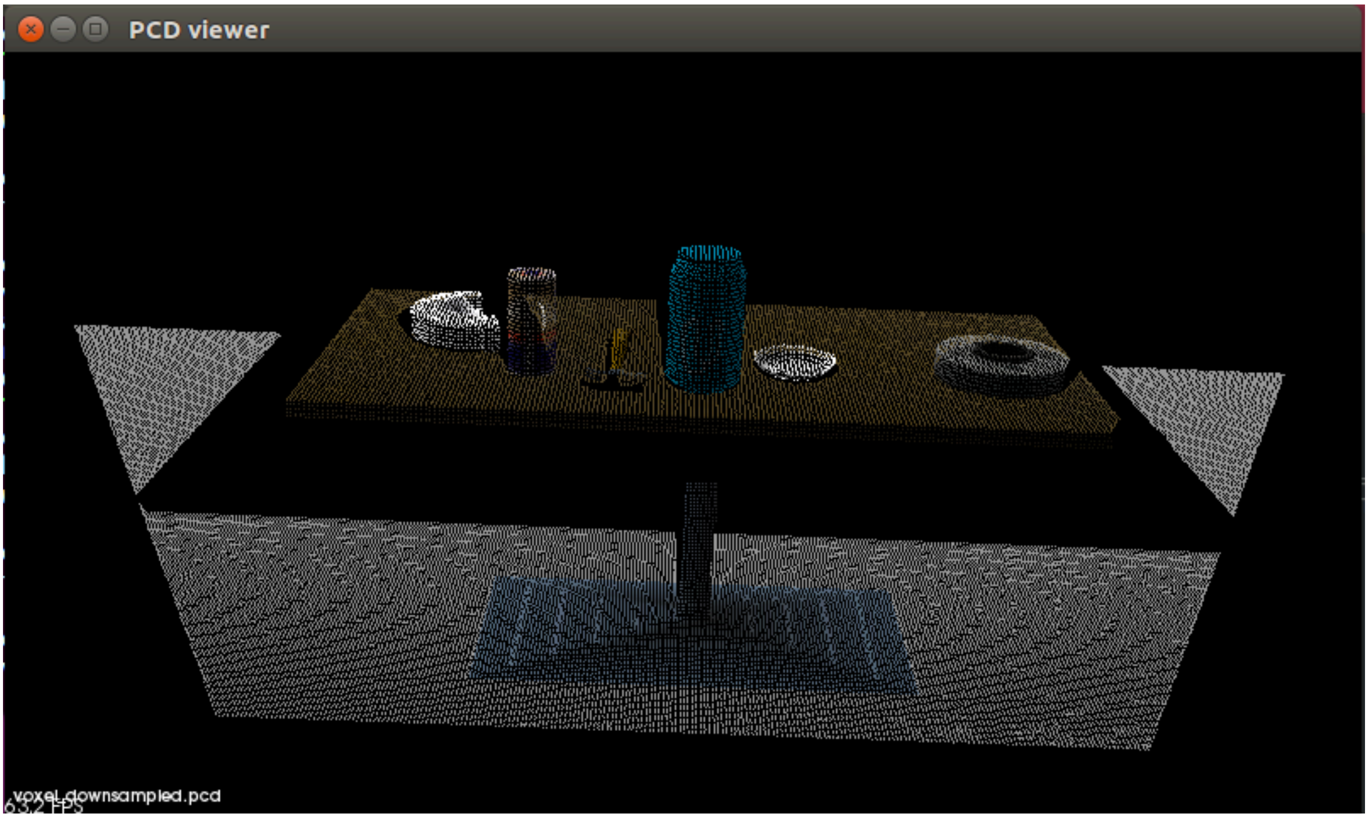


Voxel Downsampling

```
# Voxel Grid filter
# ==========================================================================
# Create a VoxelGrid filter object for our input point cloud
vox = cloud.make_voxel_grid_filter()

# Choose a voxel (also known as leaf) size
# Note: this (1) is a poor choice of leaf size
# Experiment and find the appropriate 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()
filename = 'voxel_downsampled.pcd'
pcl.save(cloud_filtered, filename)
```

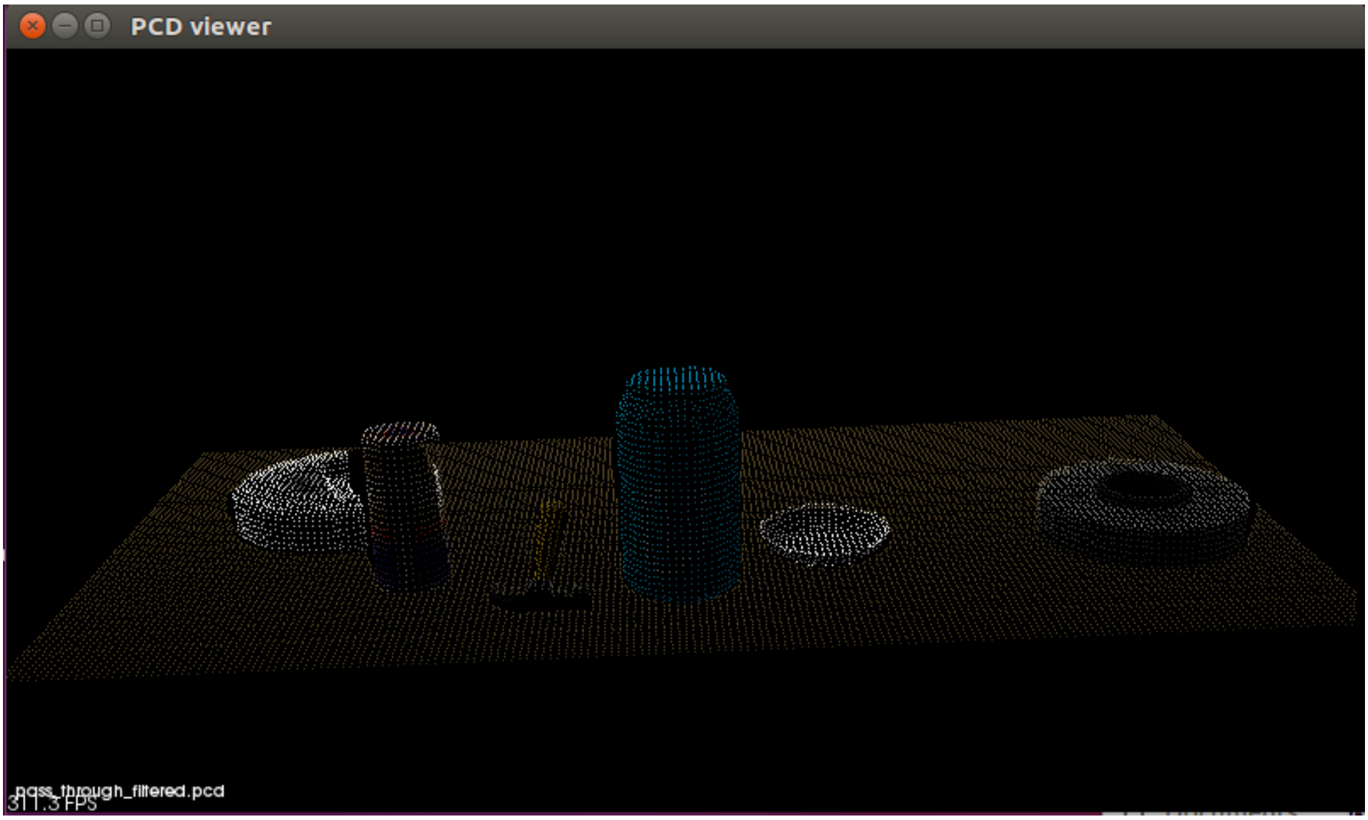
A LEAF size of 0.01 seems to provide the best balance between the size of the resulting point cloud and the accuracy of the data. The resulting downsampled point cloud is shown in the following image



Pass-Through Filter

```
# PassThrough filter
# ==========================================================================
# PassThrough filter
# 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 = 0.77
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()
filename = 'pass_through_filtered.pcd'
pcl.save(cloud_filtered, filename)
```

A filter on Z axis with min = 0.77 and max = 1.1 seems to provide a good separation extraction of the table top and the items:



Outlier Filter

Before implementing RANSAC we add an outlier filter to remove noise:

```
 Outlier Removal Filter
# ===========================================================================
# Much like the previous filters, we start by creating a filter object:
outlier_filter = cloud_filtered.make_statistical_outlier_filter()

# Set the number of neighboring points to analyze for any given point
outlier_filter.set_mean_k(50)

# Set threshold scale factor
x = 1.0

# 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)

# Finally call the filter function for magic
cloud_filtered = outlier_filter.filter()
```

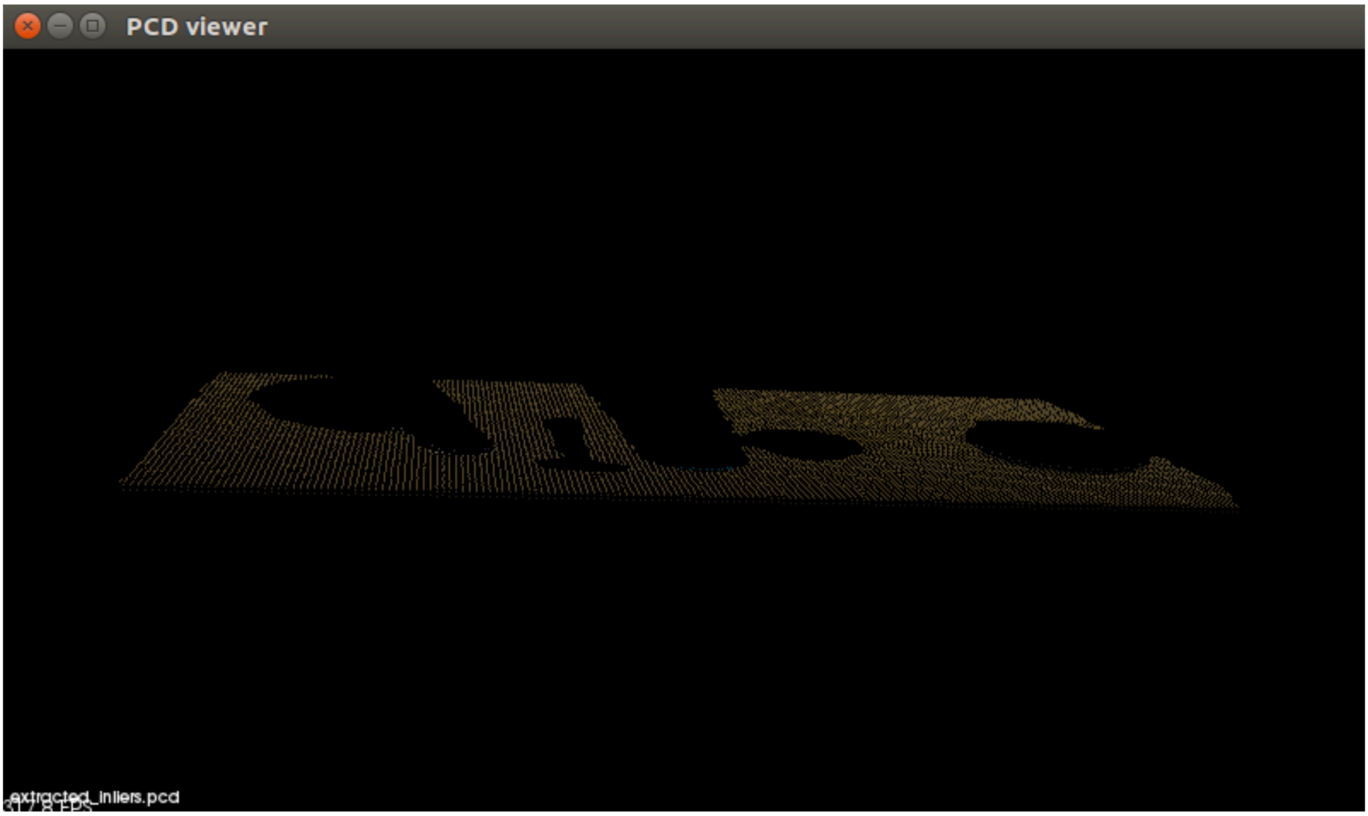
As for Exercise 1 there is no noise in the image I have not provided a pcl_viewer image of the filtered point cloud (it would look the same as the one before).

RANSAC Filtering

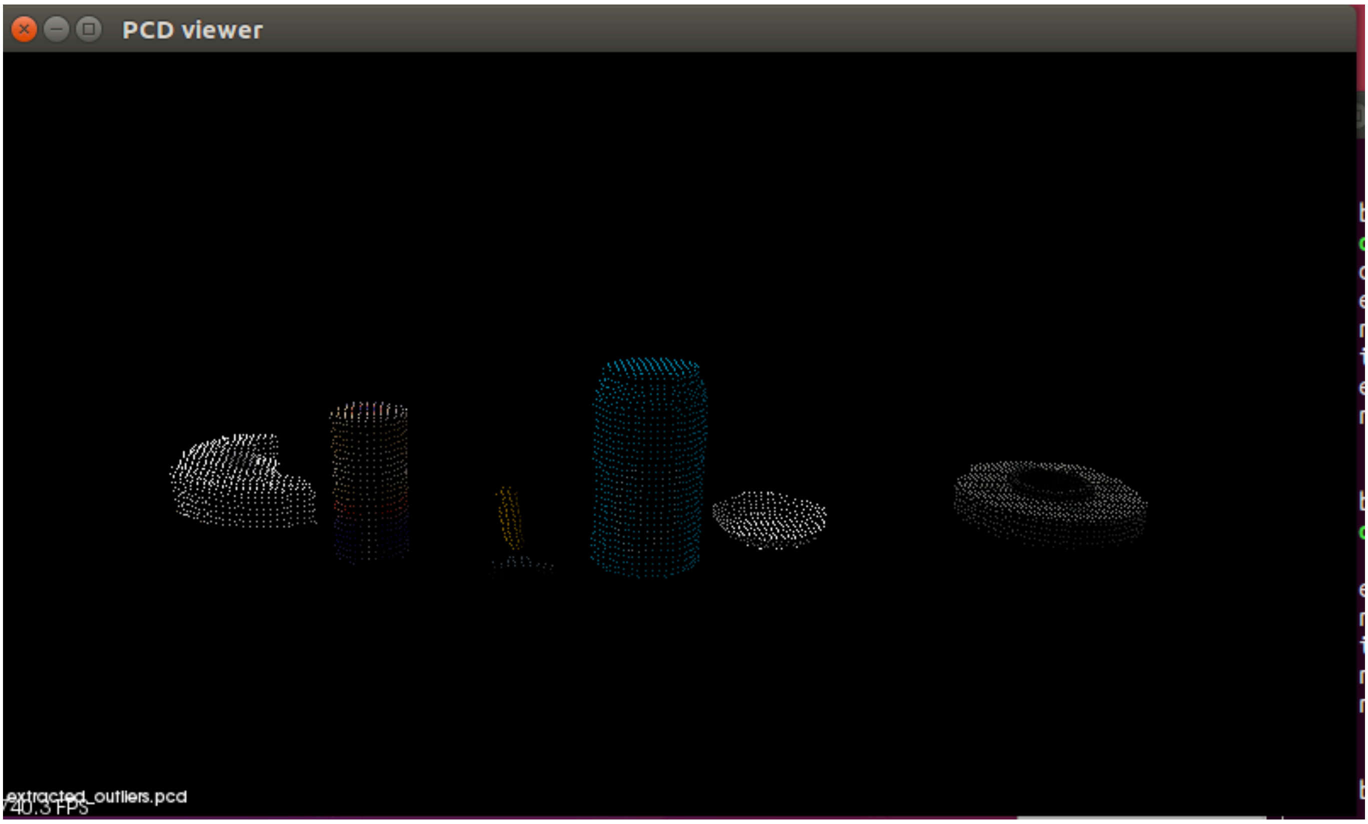
We apply a RANSAC filter to separate the objects from the table:

```
# RANSAC plane segmentation
# ===========================================================================
# 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()
filename = 'inliers_segment.pcd'
pcl.save(cloud_filtered, filename)


# Extract inliers
# ===========================================================================
extracted_inliers = cloud_filtered.extract(inliers, negative=False)
filename = 'extracted_inliers.pcd'
pcl.save(extracted_inliers, filename)



# Save pcd for table
# pcl.save(cloud, filename)


# Extract outliers
# ===========================================================================
extracted_outliers = cloud_filtered.extract(inliers, negative=True)
filename = 'extracted_outliers.pcd'
pcl.save(extracted_outliers, filename)

# Save pcd for tabletop objects
```

The table will be extracted as “inliers” as it is the dominant feature in the point cloud:



While the rest of the objects will be filtered as “outliers”:



This concludes the Exercise 1 code.

Exercise 2

Initialisation of the ROS node

In the `__main__` method of the perception.py we perform the initialisation of the ROS node, the setup of the subscribers and publishers of messages and the main processing loop:

```
if __name__ == '__main__':

    # ROS node initialization
    rospy.init_node("clustering", anonymous=True)

    # Create Subscribers
    pcl_sub = rospy.Subscriber("/sensor_stick/point_cloud",
                               pc2.PointCloud2,
                               pcl_callback,
                               queue_size=1)

    # Create Publishers
    pcl_objects_pub = rospy.Publisher("/pcl_objects", PointCloud2, queue_size=1)
    pcl_table_pub = rospy.Publisher("/pcl_table", PointCloud2, queue_size=1)
    pcl_cluster_pub = rospy.Publisher("/pcl_cluster", PointCloud2, queue_size=1)

    # Initialize color_list
    get_color_list.color_list = []

    # Spin while node is not shutdown
    while not rospy.is_shutdown():
        rospy.spin()
```

Main Processing

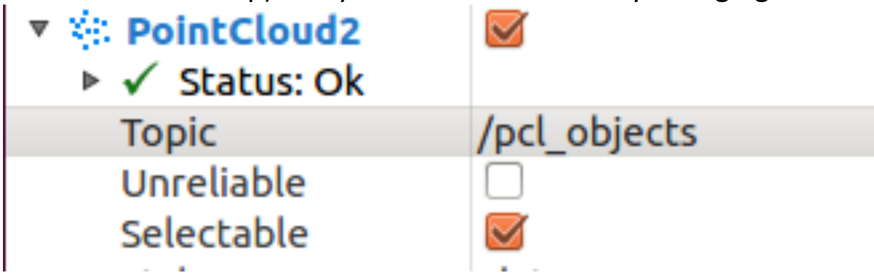
The main processing in `pcl_callback()` is as follows.

First we perform the same filtering as in Exercise 1, specifically voxel downsampling, pass through, RANSAC from which we extract the inliers (table) and the outliers (objects):

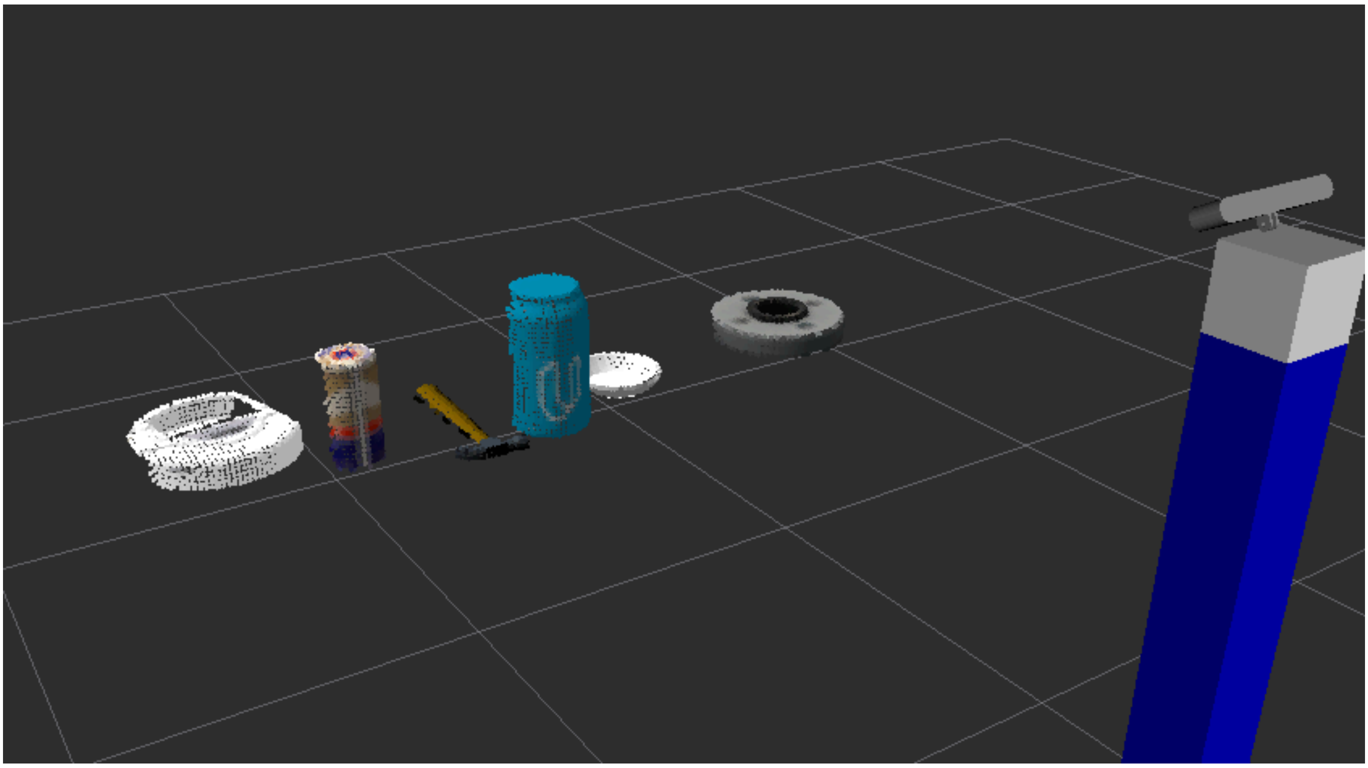
```
# Callback function for your Point Cloud Subscriber
def pcl_callback(pcl_msg):

    # Convert ROS msg to PCL data
    pcl_data = ros_to_pcl(pcl_msg)

    # Voxel Grid Downsampling
    vox = pcl_data.make_voxel_grid_filter()
    LEAF_SIZE = 0.01
    vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
    pcl_filtered = vox.filter()

    # PassThrough Filter
    passthrough = pcl_filtered.make_passthrough_filter()
    filter_axis = 'z'
    passthrough.set_filter_field_name(filter_axis)
    axis_min = 0.77
    axis_max = 1.1
```

```
passthrough.set_filter_limits(axis_min, axis_max)
pcl_filtered = passthrough.filter()

# RANSAC Plane Segmentation
seg = pcl_filtered.make_segmenter()
seg.set_model_type(pcl.SACMODEL_PLANE)
seg.set_method_type(pcl.SAC_RANSAC)
max_distance = 0.01
seg.set_distance_threshold(max_distance)
inliers, coefficients = seg.segment()

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

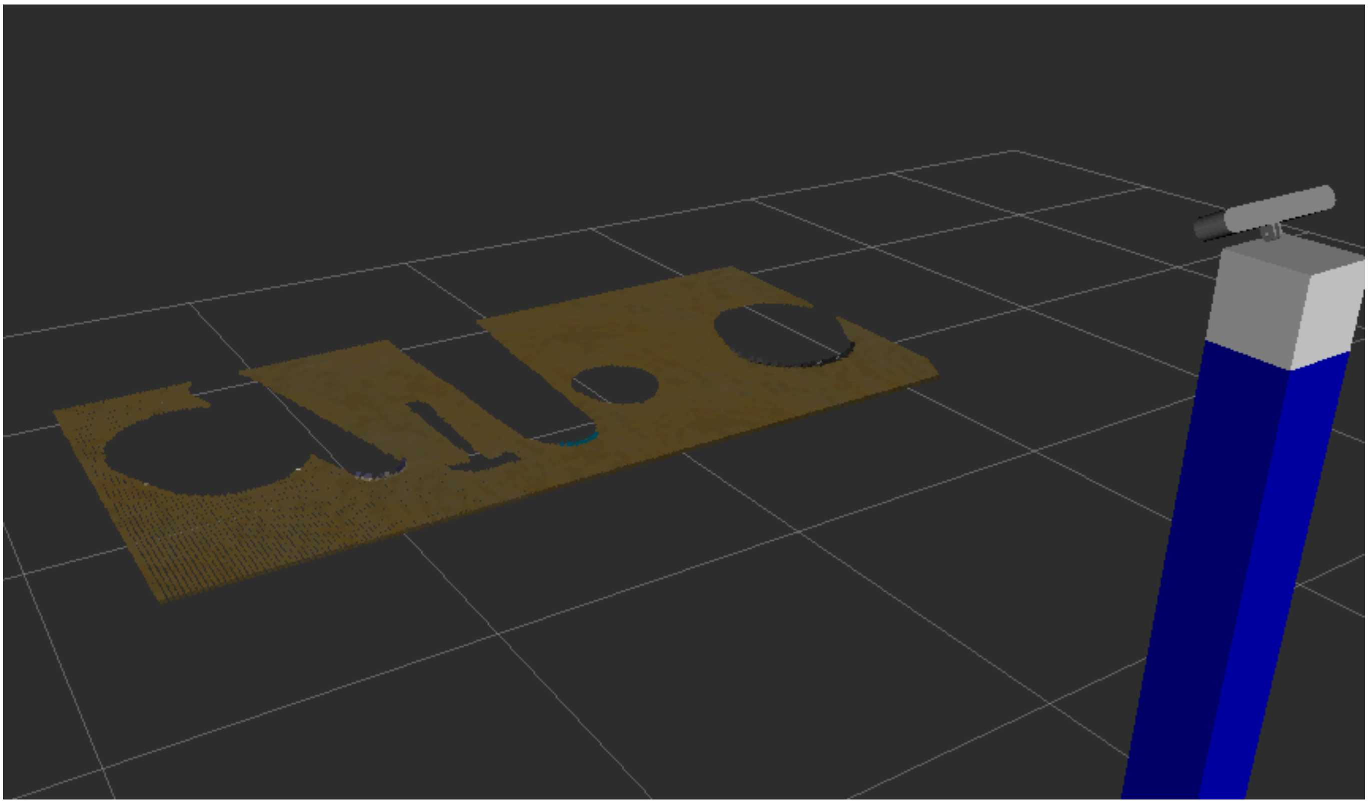
These are later in the method published under the topic `/pcl_objects` and `/pcl_table` (will be shown later in the write-up). They can be seen in RViz by changing the PointCloud2 topic shown:



And they look like this (first the objects):



And the table:



We now perform the segmentation based on Euclidian distance:

```
# Euclidean Clustering
white_cloud = XYZRGB_to_XYZ(cloud_objects)
tree = white_cloud.make_kdtree()
ec = white_cloud.make_EuclideanClusterExtraction()
ec.set_ClusterTolerance(0.02)
ec.set_MinClusterSize(10)
ec.set_MaxClusterSize(20000)
# Search the k-d tree for clusters
ec.set_SearchMethod(tree)
# Extract indices for each of the discovered clusters
cluster_indices = ec.Extract()

# Create Cluster-Mask Point Cloud to visualize each cluster separately
cluster_color = get_color_list(len(cluster_indices))
color_cluster_point_list = []
```

We use a relatively small tolerance (0.02) and set relatively wide ranges for the minimum amnd maximum number of points in the cluster (10 and 20000 respectively).

We now use this information to colour the clusters with dedicated colours taken randomly:

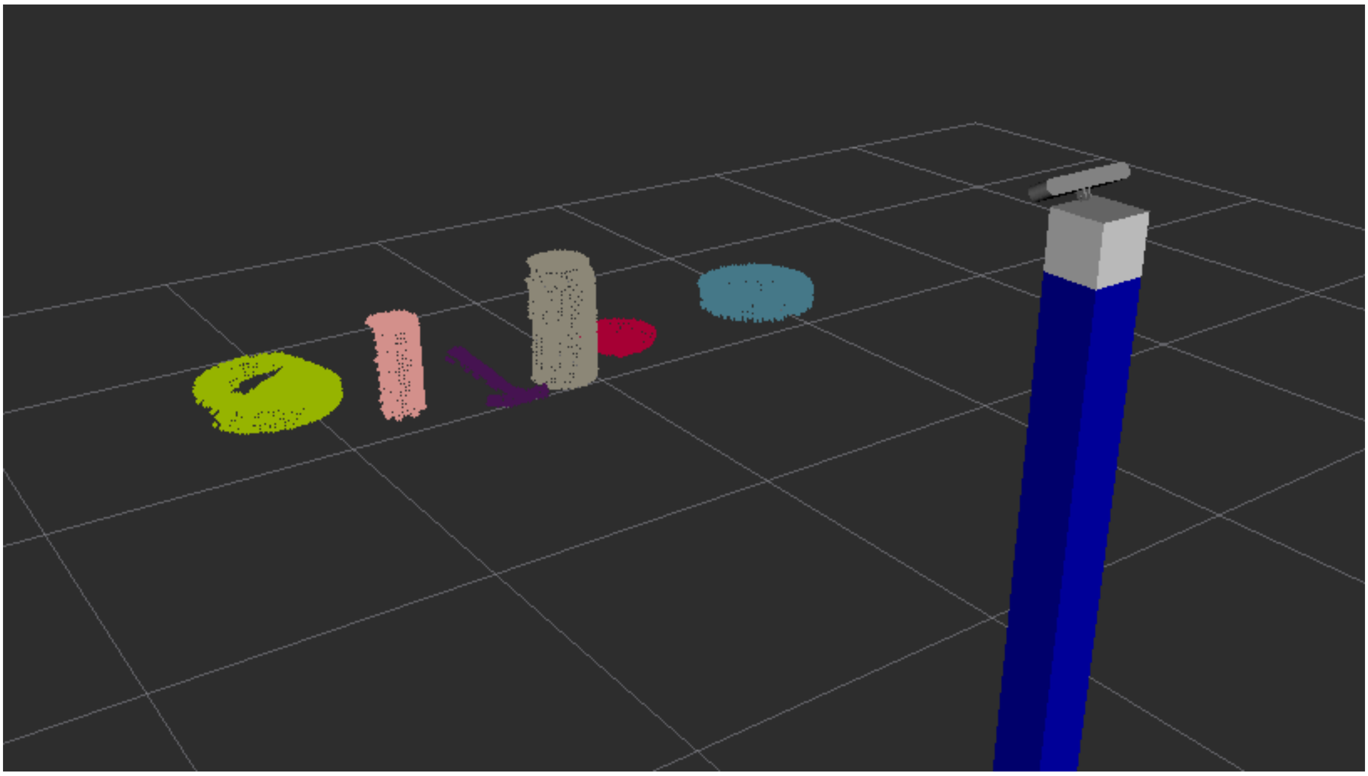
```
# Create Cluster-Mask Point Cloud to visualize each cluster separately
cluster_color = get_color_list(len(cluster_indices))
color_cluster_point_list = []

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)
```

In the end we convert the point clouds to ROS message structures and publish them:

```
    # Convert PCL data to ROS messages
    ros_cloud_objects = pcl_to_ros(cloud_objects)
    ros_cloud_table = pcl_to_ros(cloud_table)
    ros_cluster_cloud = pcl_to_ros(cluster_cloud)

    # Publish ROS messages
    pcl_objects_pub.publish(ros_cloud_objects)
    pcl_table_pub.publish(ros_cloud_table)
    pcl_cluster_pub.publish(ros_cluster_cloud)
```

The way the objects are reflects in RViz on the topic `/pcl_cluster` is the following:



Exercise 3

Histogram Generation

After the first run of `capture_features.py` and `train_svm.py` the results are as expected very bad since the feature detections are outputting random values.

I implemented the histogram methods in `features.py` (in `sensor_stick/src/sensor_stick/`) as follows:

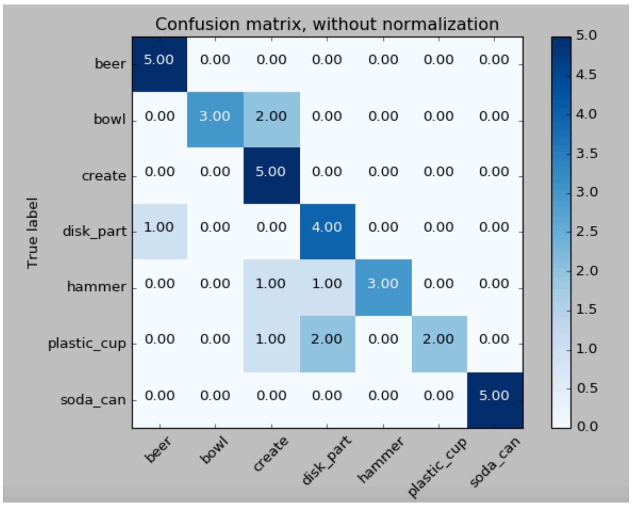
```
def compute_color_histograms(cloud, using_hsv=False):

    # Compute histograms for the clusters
    point_colors_list = []

    # Step through each point in the point cloud
    for point in pc2.read_points(cloud, skip_nans=True):
        rgb_list = float_to_rgb(point[3])
        if using_hsv:
            point_colors_list.append(rgb_to_hsv(rgb_list) * 255)
        else:
            point_colors_list.append(rgb_list)

    # Populate lists with color values
    channel_1_vals = []
    channel_2_vals = []
    channel_3_vals = []

    for color in point_colors_list:
        channel_1_vals.append(color[0])
        channel_2_vals.append(color[1])
        channel_3_vals.append(color[2])

    # Compute histograms
    ch1_hist = np.histogram(channel_1_vals, bins=32, range=(0, 256))
    ch2_hist = np.histogram(channel_2_vals, bins=32, range=(0, 256))
    ch3_hist = np.histogram(channel_3_vals, bins=32, range=(0, 256))

    # Concatenate and normalize the histograms
    hist_features = np.concatenate((ch1_hist[0],
                                    ch2_hist[0],
                                    ch3_hist[0])).astype(np.float64)
    normed_features = hist_features / np.sum(hist_features)

    return normed_features


def compute_normal_histograms(normal_cloud):
    norm_x_vals = []
    norm_y_vals = []
    norm_z_vals = []

    for norm_component in pc2.read_points(normal_cloud,
                                          field_names = ('normal_x', 'normal_y',
'normal_z'),
                                          skip_nans=True):
        norm_x_vals.append(norm_component[0])
        norm_y_vals.append(norm_component[1])
        norm_z_vals.append(norm_component[2])

    # Compute histograms of normal values (just like with color)
    x_hist = np.histogram(norm_x_vals, bins=32, range=(0, 256))
    y_hist = np.histogram(norm_y_vals, bins=32, range=(0, 256))
    z_hist = np.histogram(norm_z_vals, bins=32, range=(0, 256))

    # Concatenate and normalize the histograms
    hist_features = np.concatenate((x_hist[0],
                                    y_hist[0],
                                    z_hist[0])).astype(np.float64)
    normed_features = hist_features / np.sum(hist_features)
```

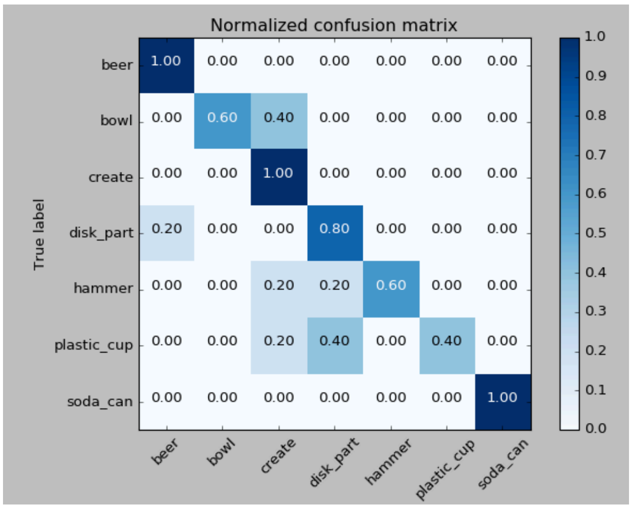
```
    return normed_features
```

I have re-run `capture_features.py` and `train_svm.py` the results are as follows:

```
Features in Training Set: 35
Invalid Features in Training set: 0
Scores: [ 0.71428571  0.85714286  0.57142857  1.          0.71428571]
Accuracy: 0.77 (+/- 0.29)
accuracy score: 0.771428571429
```

And the confusion matrices are:





They are better but we can improve them.

<u>Improve the SVM training</u>

We will perform some changes to the capture_features.py so that we generate more samples for training. Having only 5 training samples for each object is unsatisfactory.

```
    for model_name in models:
        spawn_model(model_name)

        for i in range(50):
            # make five attempts to get a valid a point cloud then give up
            sample_was_good = False
            try_count = 0
```

We will generate 50 samples for each item.

We will also use HSV histograms that are better suited for similarity comparison and less subjected to errors due to differences in ilumination.

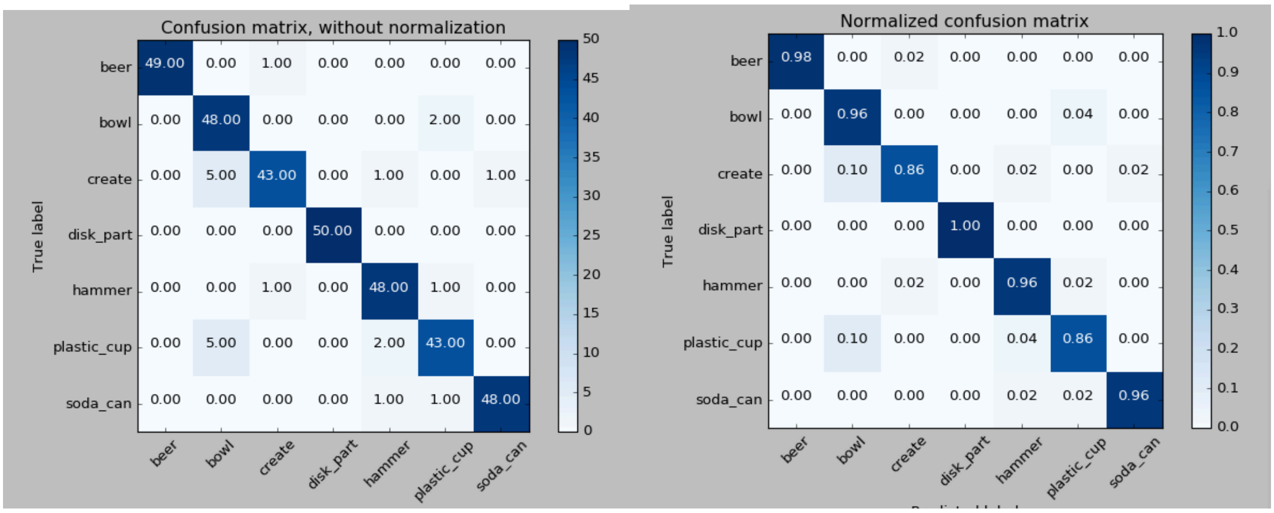
```
            chists = compute_color_histograms(sample_cloud, using_hsv=True)
```

After these changes, training the SVM model produces:

```
Features in Training Set: 350
Invalid Features in Training set: 0
```

```
Scores: [ 0.9         0.92857143  0.95714286  0.92857143  0.98571429]
Accuracy: 0.94 (+/- 0.06)
accuracy score: 0.94
```

With the confusion matrices:



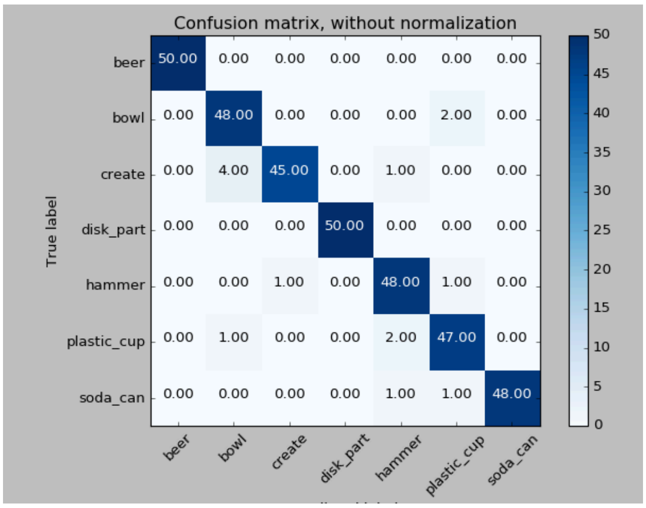
Can we improve on this result? Since we have more training samples (350) we can try to increase the number of folds that we use in the cross-validation:

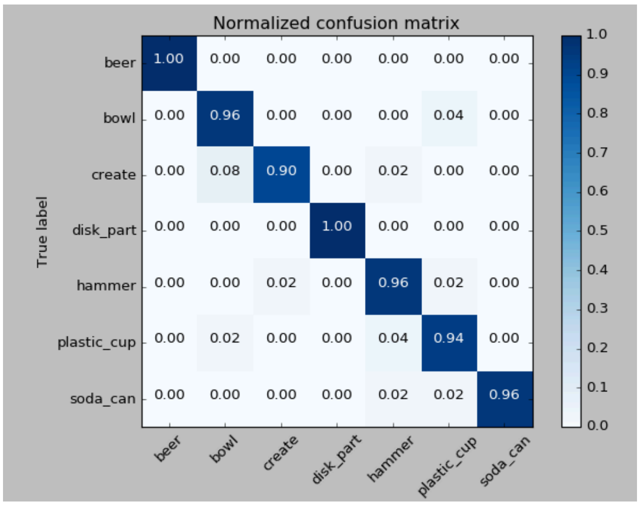
```
# Set up 5-fold cross-validation
kf = cross_validation.KFold(len(X_train),
                            n_folds=10,
                            shuffle=True,
                            random_state=1)
```

When running `train_svm.py` we obtain:

```
Features in Training Set: 350
Invalid Features in Training set: 0
Scores: [ 0.94285714  0.97142857  0.94285714  0.88571429  0.97142857  0.97142857
  0.94285714  1.          1.          0.97142857]
Accuracy: 0.96 (+/- 0.06)
accuracy score: 0.96
```

With the confusion matrices with the following shape:





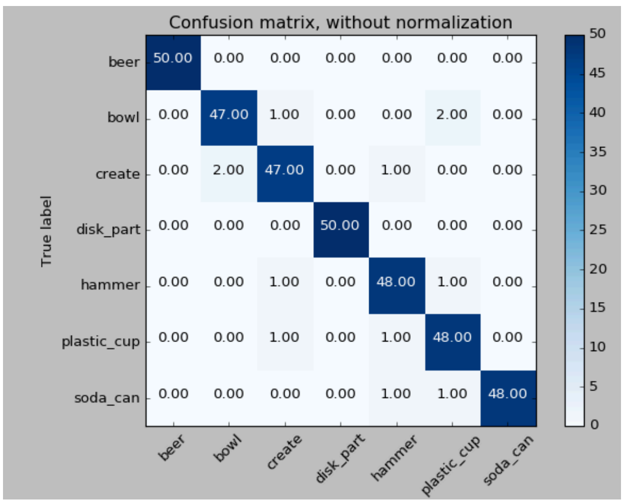
This certainly improved the recognition of the plastic cup, but we still have some errors on the “create”. One additional change we can make is to setup the clasisifier with a different “C” parameter (by default it is 1) so that we introduce a slighty better regularisation of the data before training.

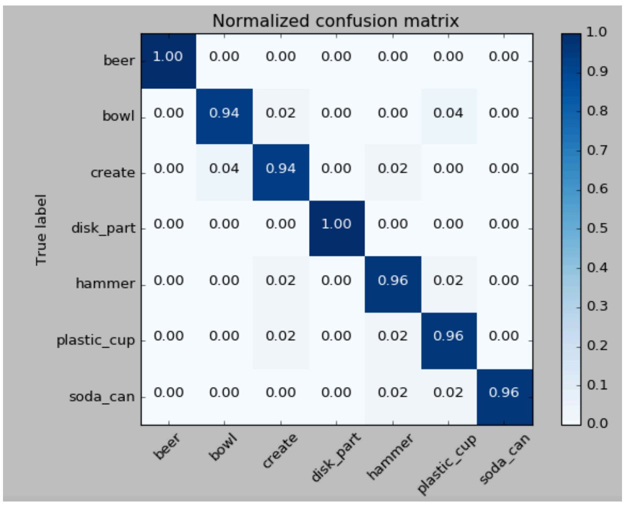
```
# Create classifier
clf = svm.SVC(kernel='linear', C=0.3)
```

Running the training now results in this:

Features in Training Set: 350
Invalid Features in Training set: 0
Scores: [0.97142857 0.94285714 0.94285714 0.91428571 0.97142857 1.
0.94285714 1. 1. 0.97142857]
Accuracy: 0.97 (+/- 0.06)
accuracy score: 0.965714285714

And the confusion matrices:





The average score is just slightly up but most importantly the recognition of “create” object is significaly improved (from 90% to 94%) albeit with a slight worsening of th recognition for “bowl” (that decreases from 96% to 94%). Overall though we are happy with this training and we will use it in the next section of the excersise.

We can now move to the recognition of the object within ROS.

We create the file `object_rocognition.py` in `sensor_stick/scripts/` and update the following:

In the __main__ we add the following code:

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

    # Create Subscribers
    pcl_sub = rospy.Subscriber("/sensor_stick/point_cloud",
                               pc2.PointCloud2,
                               pcl_callback,
                               queue_size=1)

    # Create Publishers
    pcl_objects_pub = rospy.Publisher("/pcl_objects", PointCloud2, queue_size=1)
    pcl_table_pub = rospy.Publisher("/pcl_table", PointCloud2, queue_size=1)
    pcl_cluster_pub = rospy.Publisher("/pcl_cluster", PointCloud2, queue_size=1)
    object_markers_pub = rospy.Publisher("/object_markers", Marker, queue_size=1)
    detected_objects_pub = rospy.Publisher("/detected_objects", DetectedObjectsArray,
queue_size=1)

    # Load Model From disk
    model = pickle.load(open('model.sav', 'rb'))
    clf = model['classifier']
    encoder = LabelEncoder()
    encoder.classes_ = model['classes']
    scaler = model['scaler']

    # Initialize color_list
    get_color_list.color_list = []

    # Spin while node is not shutdown
    while not rospy.is_shutdown():
        rospy.spin()
```

In addition to the initialisation of the ROS node and the subscribers and publishers setup, we read the SVM recognition model that we trained earlier and we add an additional Publisher.

In the `pcl_callback()` function we first copy the code we has in previous exercise:

```
# Callback function for your Point Cloud Subscriber
def pcl_callback(pcl_msg):

    # Convert ROS msg to PCL data
    pcl_data = ros_to_pcl(pcl_msg)

    # Voxel Grid Downsampling
    vox = pcl_data.make_voxel_grid_filter()
    LEAF_SIZE = 0.01
    vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
    pcl_filtered = vox.filter()
```

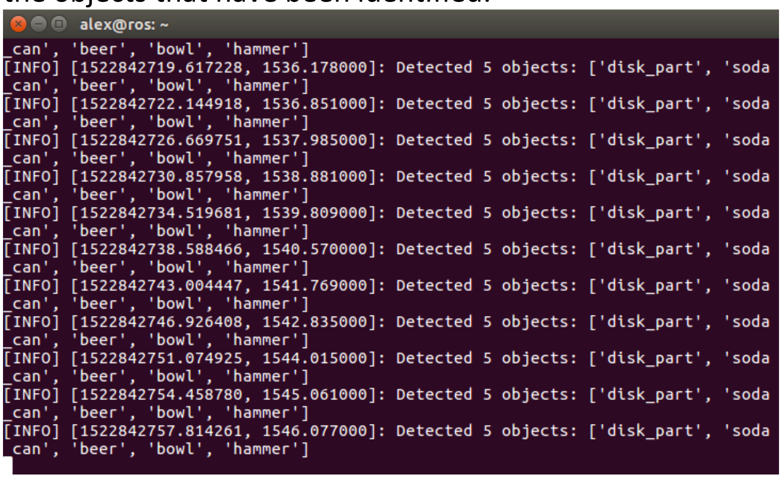
```
# PassThrough Filter
passthrough = pcl_filtered.make_passthrough_filter()
filter_axis = 'z'
passthrough.set_filter_field_name(filter_axis)
axis_min = 0.77
axis_max = 1.1
passthrough.set_filter_limits(axis_min, axis_max)
pcl_filtered = passthrough.filter()

# RANSAC Plane Segmentation
seg = pcl_filtered.make_segmenter()
seg.set_model_type(pcl.SACMODEL_PLANE)
seg.set_method_type(pcl.SAC_RANSAC)
max_distance = 0.01
seg.set_distance_threshold(max_distance)
inliers, coefficients = seg.segment()

# Extract inliers and outliers
cloud_table = pcl_filtered.extract(inliers, negative=False)
cloud_objects = pcl_filtered.extract(inliers, negative=True)

# Euclidean Clustering
white_cloud = XYZRGB_to_XYZ(cloud_objects)
tree = white_cloud.make_kdtree()
ec = white_cloud.make_EuclideanClusterExtraction()
ec.set_ClusterTolerance(0.02)
ec.set_MinClusterSize(10)
ec.set_MaxClusterSize(20000)
# Search the k-d tree for clusters
ec.set_SearchMethod(tree)
# Extract indices for each of the discovered clusters
cluster_indices = ec.Extract()

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

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

# Publish ROS messages
pcl_objects_pub.publish(ros_cloud_objects)
pcl_table_pub.publish(ros_cloud_table)
pcl_cluster_pub.publish(ros_cluster_cloud)
```

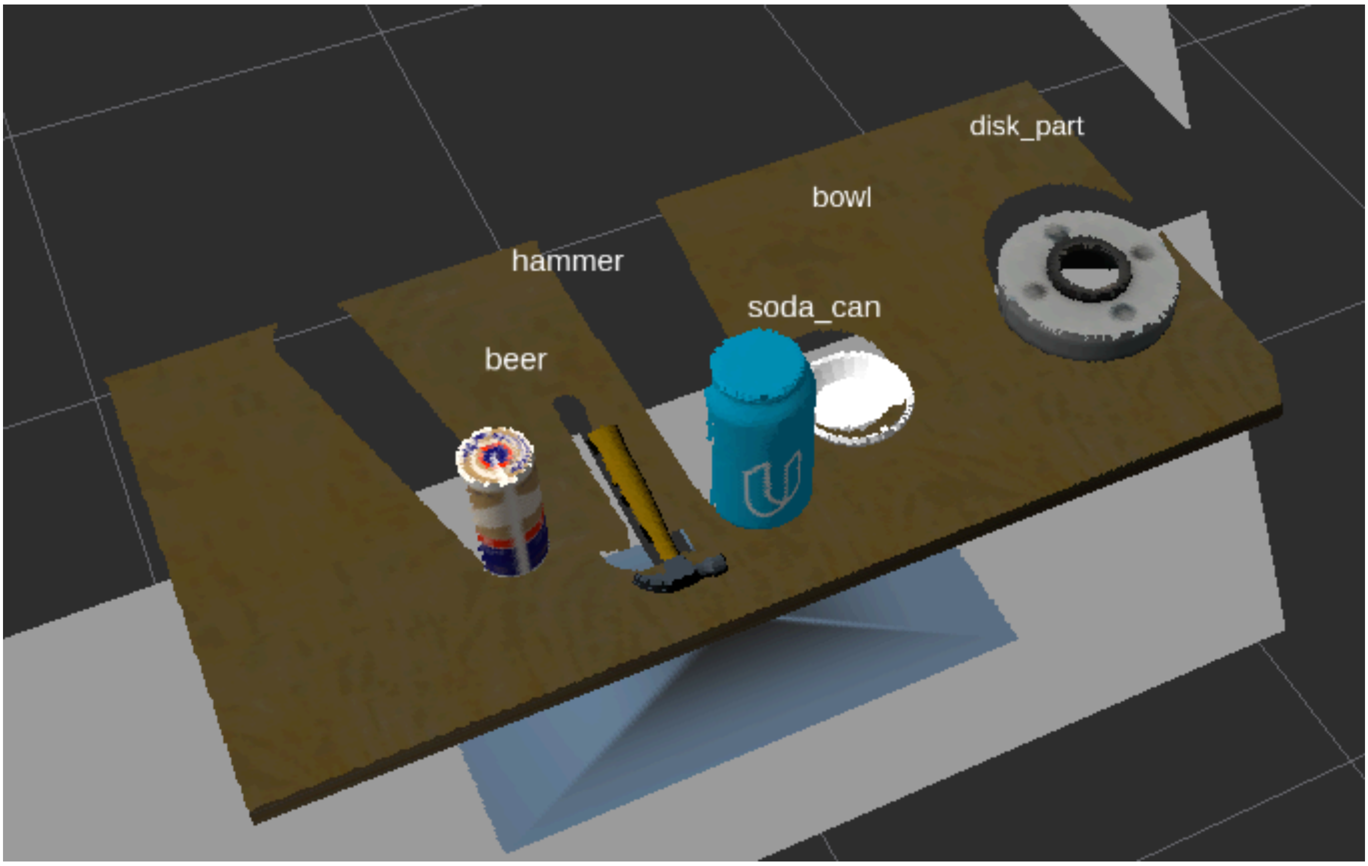
And we include the code that will produce the ROS message with the labelled objects:

```
# Classify the clusters!
```

```
    detected_objects_labels = []
    detected_objects = []

    for index, pts_list in enumerate(cluster_indices):
        # Grab the points for the cluster from the extracted outliers (cloud_objects)
        pcl_cluster = cloud_objects.extract(pts_list)
        # TODO: convert the cluster from pcl to ROS using helper function
        ros_cluster = pcl_to_ros(pcl_cluster)

        # Extract histogram features
        # TODO: complete this step just as is covered in capture_features.py
        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, retrieve the label for the result
        # and add it to detected_objects_labels list
        prediction = clf.predict(scaler.transform(feature.reshape(1,-1)))
        label = encoder.inverse_transform(prediction)[0]
        detected_objects_labels.append(label)

        # Publish a label into RViz
        label_pos = list(white_cloud[pts_list[0]])
        label_pos[2] += .4
        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)

    rospy.loginfo('Detected {} objects: {}'.format(len(detected_objects_labels),
detected_objects_labels))

    # Publish the list of detected objects
    # This is the output you'll need to complete the upcoming project!
    detected_objects_pub.publish(detected_objects)
```

We need to run this program from the same directory where the SVM model was saved using rosrun. Once this is done the items are shown with the labels on top and the console issues messages indicating the objects that have been identified:





This concludes the Exercise 3.

Pick And Place

We will now implement the pipeline for the pr2_robot.

Training the SVM model

Similar to the previous exercises we have copied the script capture_features.py into the pr2_robot/scripts (to have a clean version for this project) and made the following changes:

The models have been changed to reflect the ones that are in the project's worlds:

```
if __name__ == '__main__':
    rospy.init_node('capture_node')

    models = [\
        'sticky_notes',
        'book',
        'snacks',
        'biscuits',
        'eraser',
        'soap',
        'soap2',
        'glue']

```

Since the models are included in the sensor_stick package too we do not need to worry about the access to the models.

Also, since we are looking for a high accuracy in the recognition and because we know that the data will have noise we have decided to increase the number of examples to 100 for each object:

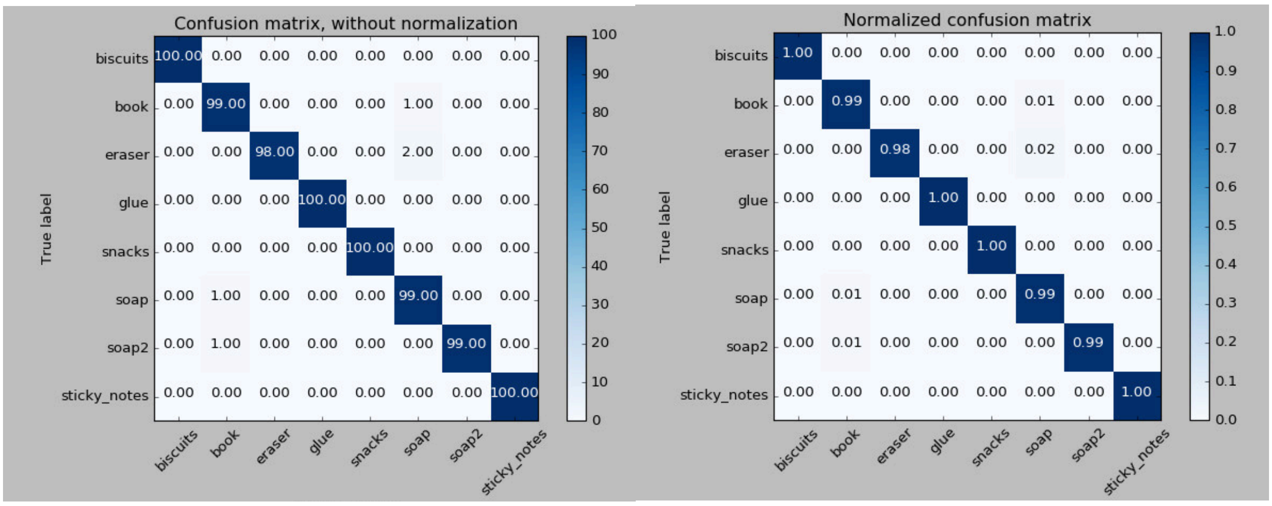
```

    for i in range(100):
        # make five attempts to get
        sample_was_good = False
        try_count = 0
        while not sample_was_good an
```

This in principle should give us a good training model for the recognition. Since I am running this on a server with 32 cores the generation of the training data is also not a big issue.

Once these 800 training samples are produced we are running the `train_svm.py` script that is unchanged from the version that we used in Example 3. The results of the training are:

```
Features in Training Set: 800
Invalid Features in Training set: 0
Scores: [ 0.9875  0.9875  1.      1.      0.9875  1.      0.9875  0.9875  1.
  1.    ]
Accuracy: 0.99 (+/- 0.01)
accuracy score: 0.99375
```

As expected the training accuracy is high, also reflected in the confusion matrices:



We are now ready for the perception pipeline and

Perception Pipeline

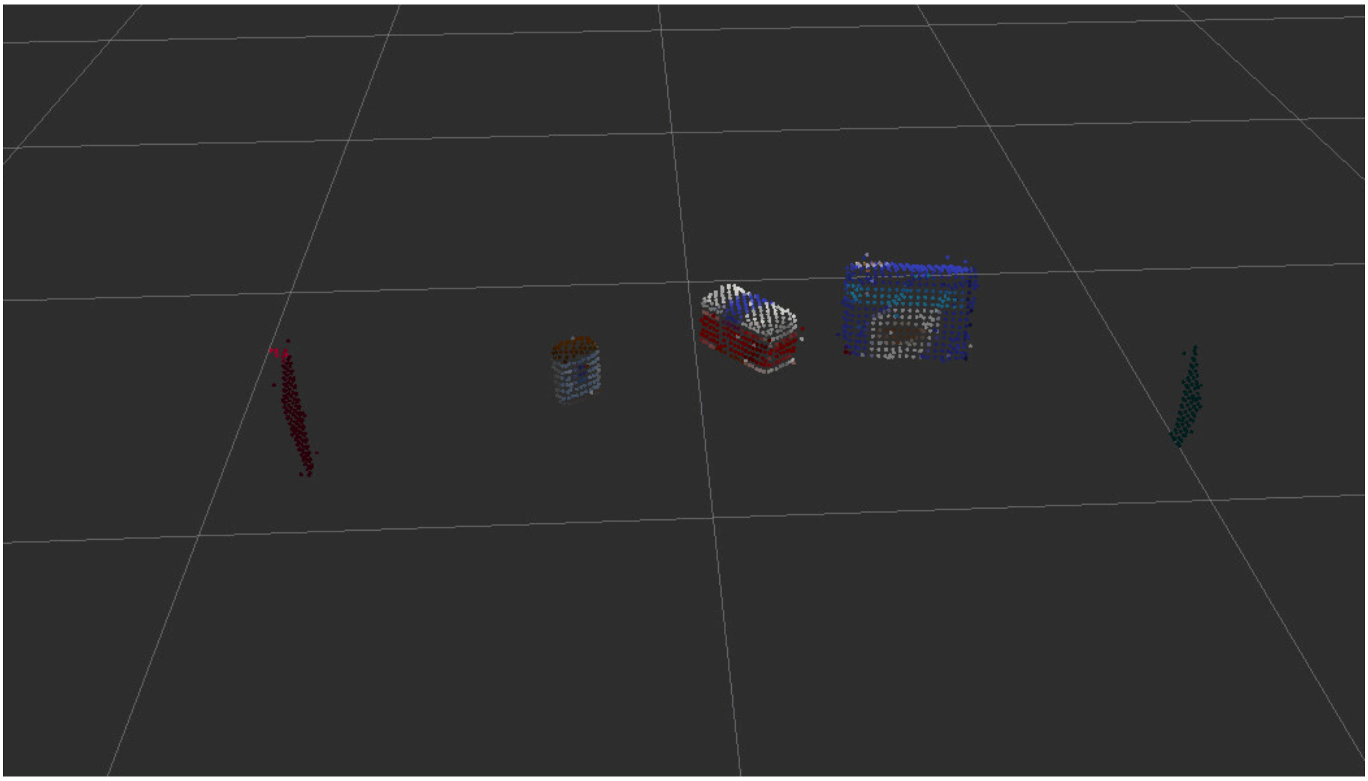
As suggested we will create a Python script in `pr2_robot/scripts` named `perception.py` that will implement the perception pipeline.

In the __main__ function we implement the same functionality as in the Exercise 3 (initialising the ROS node, creating the subscribers and publishers and loading the trained model):

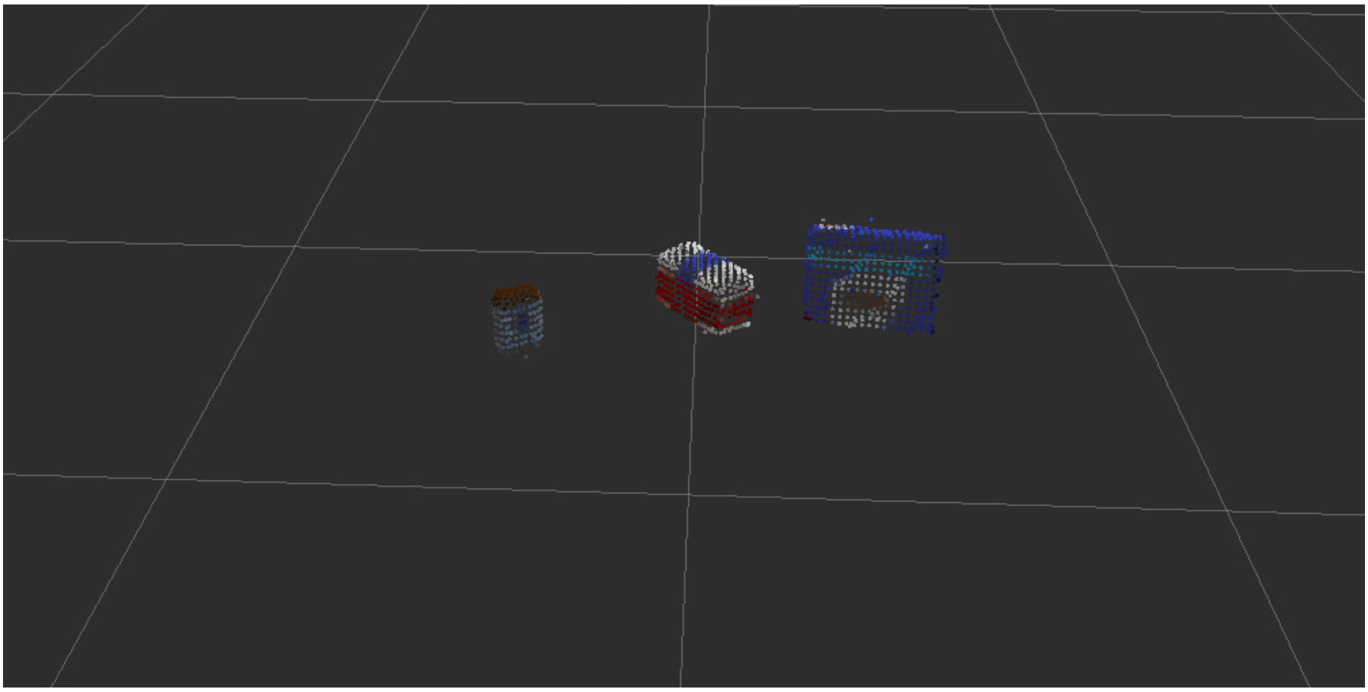
```
if __name__ == '__main__':

    # ROS node initialization
    rospy.init_node("perception", anonymous=True)

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

    # Create Publishers
    pcl_objects_pub = rospy.Publisher("/pcl_objects", PointCloud2, queue_size=1)
    pcl_table_pub = rospy.Publisher("/pcl_table", PointCloud2, queue_size=1)
    pcl_cluster_pub = rospy.Publisher("/pcl_cluster", PointCloud2, queue_size=1)
    object_markers_pub = rospy.Publisher("/object_markers", Marker, queue_size=1)
    detected_objects_pub = rospy.Publisher("/detected_objects", DetectedObjectsArray,
queue_size=1)

    # Load Model From disk
    model = pickle.load(open('model.sav', 'rb'))
    clf = model['classifier']
    encoder = LabelEncoder()
    encoder.classes_ = model['classes']
    scaler = model['scaler']

    # Initialize color_list
    get_color_list.color_list = []

    # Spin while node is not shutdown
```

```
    while not rospy.is_shutdown():
        rospy.spin()
```

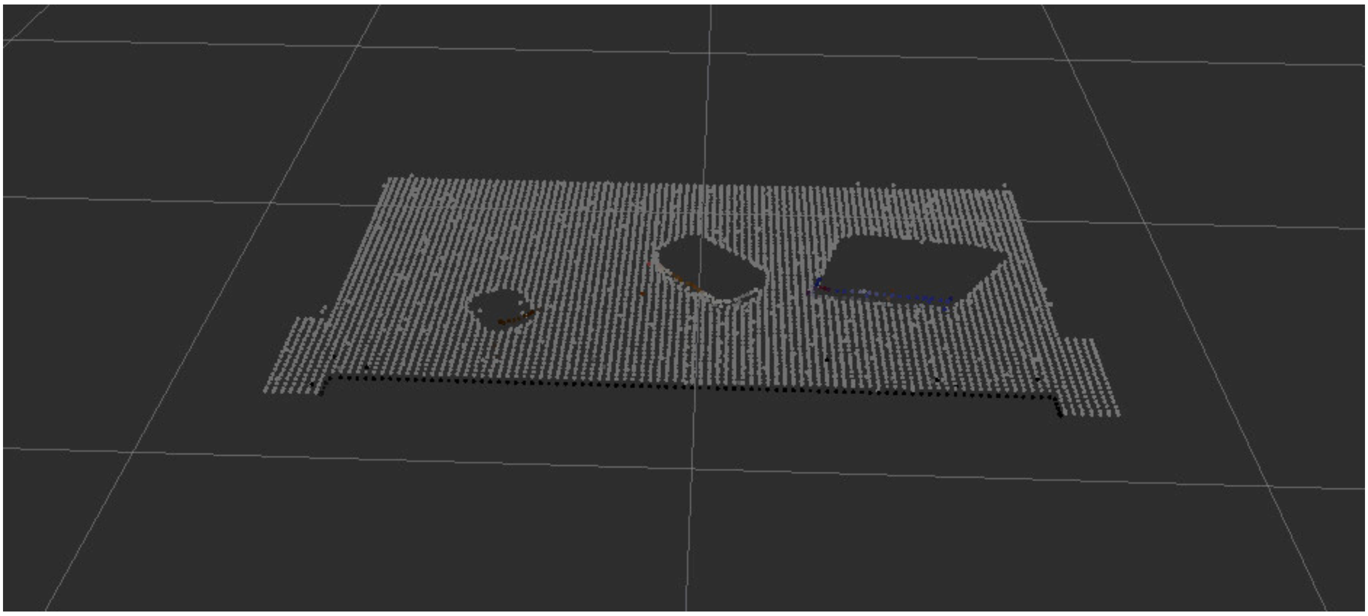
The pipeline processing is similar, except of some exceptions: one of the problems with the current pipeline is that the edges of the bins are visible on the left and right side of the field of view and will be considered as objects to be recognized:



So we'll need to add a second pass-through filter that will remove along the Y axis the items outside the main table.



And the table:



The full code for the perception pipeline is bellow:

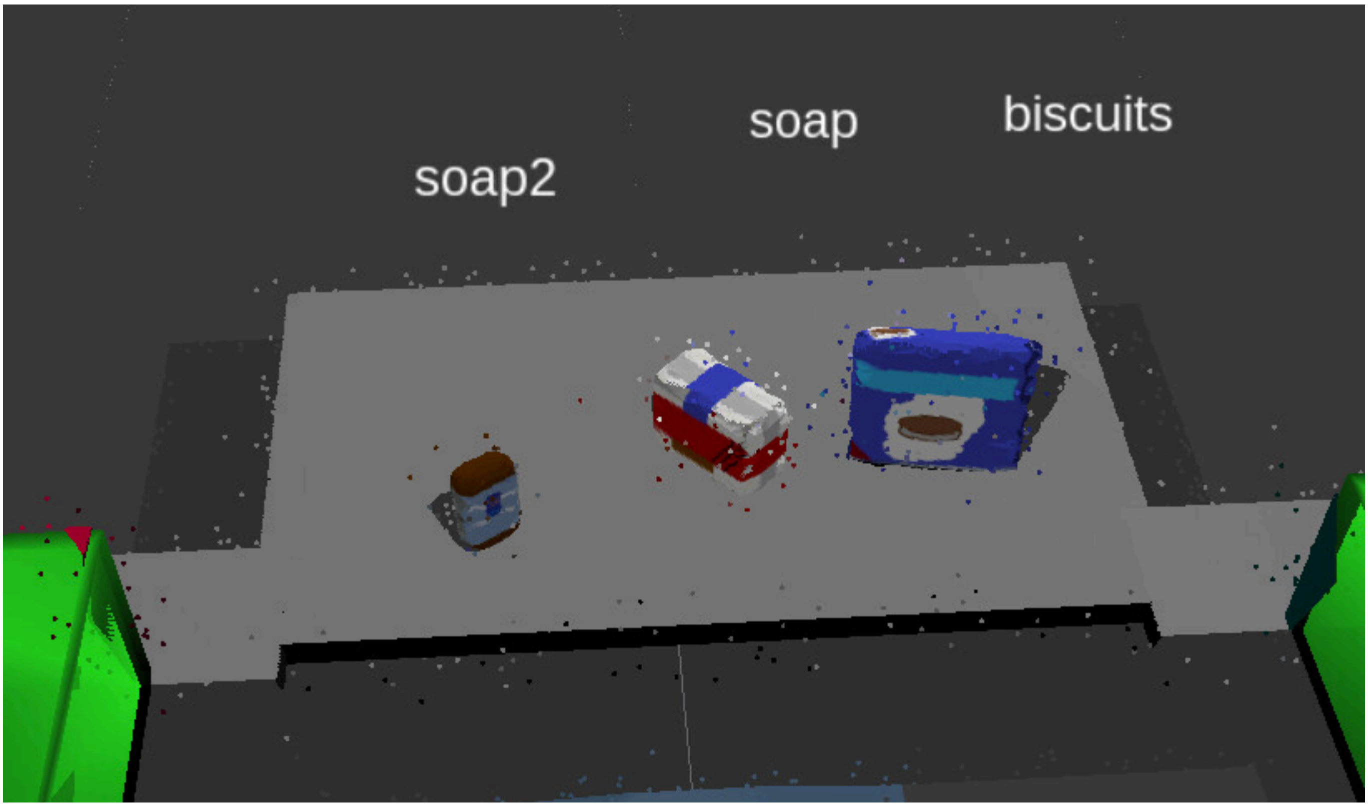
```
# Callback function for your Point Cloud Subscriber
def pcl_callback(pcl_msg):

    # Convert ROS msg to PCL data
    pcl_data = ros_to_pcl(pcl_msg)

    # Statistical Outlier Filtering
    outlier_filter = pcl_data.make_statistical_outlier_filter()
    outlier_filter.set_mean_k(50)     # Set threshold scale factor
    outlier_threshold = 0.1
    outlier_filter.set_std_dev_mul_thresh(outlier_threshold)
```

```
pcl_filtered = outlier_filter.filter()

# Voxel Grid Downsampling
voxel_filter = pcl_filtered.make_voxel_grid_filter()
LEAF_SIZE = 0.01
voxel_filter.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
pcl_filtered = voxel_filter.filter()

# PassThrough Filter
passthrough_filter = pcl_filtered.make_passthrough_filter()
filter_axis = 'z'
passthrough_filter.set_filter_field_name(filter_axis)
axis_min = 0.6
axis_max = 1.2
passthrough_filter.set_filter_limits(axis_min, axis_max)
pcl_filtered = passthrough_filter.filter()

# filter by Y axis to remove the bin edges
passthrough_filter = pcl_filtered.make_passthrough_filter()
filter_axis = 'y'
passthrough_filter.set_filter_field_name(filter_axis)
axis_min = -0.5
axis_max = 0.5
passthrough_filter.set_filter_limits(axis_min, axis_max)
pcl_filtered = passthrough_filter.filter()

# RANSAC Plane Segmentation
segmenter_filter = pcl_filtered.make_segmenter()
segmenter_filter.set_model_type(pcl.SACMODEL_PLANE)
segmenter_filter.set_method_type(pcl.SAC_RANSAC)
max_distance = 0.02
segmenter_filter.set_distance_threshold(max_distance)
inliers, coefficients = segmenter_filter.segment()

# Extract inliers and outliers
cloud_table = pcl_filtered.extract(inliers, negative=False)
cloud_objects = pcl_filtered.extract(inliers, negative=True)

# Euclidean Clustering
white_cloud = XYZRGB_to_XYZ(cloud_objects)
tree = white_cloud.make_kdtree()
ec = white_cloud.make_EuclideanClusterExtraction()
ec.set_ClusterTolerance(0.02)
ec.set_MinClusterSize(10)
ec.set_MaxClusterSize(20000)
# Search the k-d tree for clusters
ec.set_SearchMethod(tree)
# Extract indices for each of the discovered clusters
cluster_indices = ec.Extract()

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

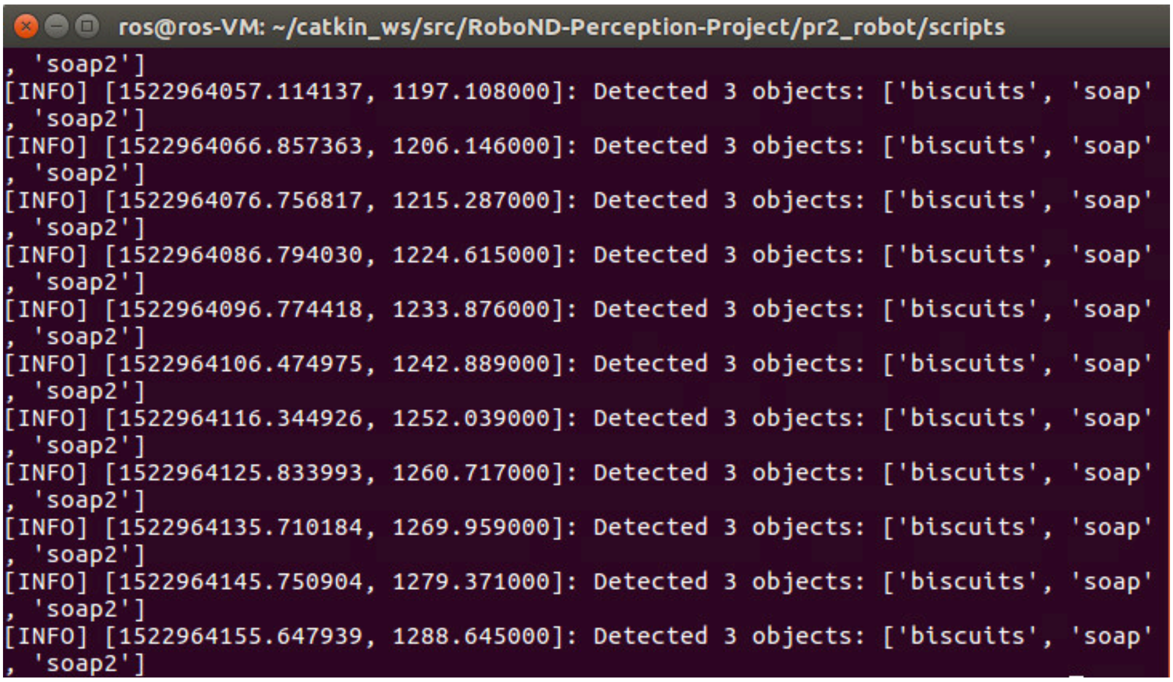
```
    cluster_cloud = pcl.PointCloud_PointXYZRGB()
    cluster_cloud.from_list(color_cluster_point_list)

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

    # Publish ROS messages
    pcl_objects_pub.publish(ros_cloud_objects)
    pcl_table_pub.publish(ros_cloud_table)
    pcl_cluster_pub.publish(ros_cluster_cloud)

# Exercise-3 TODOs:

    # Classify the clusters!
    detected_objects_labels = []
    detected_objects = []

    for index, pts_list in enumerate(cluster_indices):
        # Grab the points for the cluster from the extracted outliers (cloud_objects)
        pcl_cluster = cloud_objects.extract(pts_list)
        # convert the cluster from pcl to ROS using helper function
        ros_cluster = pcl_to_ros(pcl_cluster)

        # Extract 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, retrieve the label for the result
        # and add it to detected_objects_labels list
        prediction = clf.predict(scaler.transform(feature.reshape(1,-1)))
        label = encoder.inverse_transform(prediction)[0]
        detected_objects_labels.append(label)

        # Publish a label into RViz
        label_pos = list(white_cloud[pts_list[0]])
        label_pos[2] += .4
        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)

    rospy.loginfo('Detected {} objects: {}'.format(len(detected_objects_labels),
detected_objects_labels))

    # Publish the list of detected objects
    # This is the output you'll need to complete the upcoming project!
    detected_objects_pub.publish(detected_objects)
```

World 1 Recognition

Running the simulator with the test1.world we obtain the following recognition:



And the ROS node reports also correctly the detection of the 3 objects.

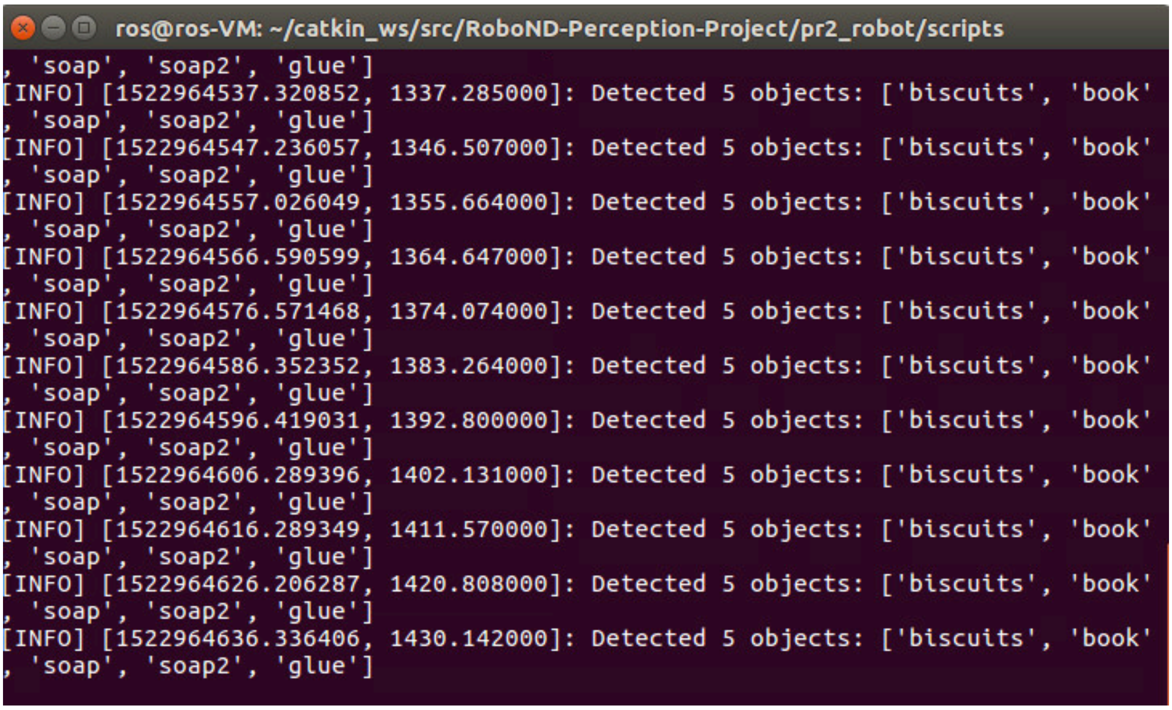


```
ros@ros-VM: ~/catkin_ws/src/RoboND-Perception-Project/pr2_robot/scripts
, 'soap2']
[INFO] [1522964057.114137, 1197.108000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
[INFO] [1522964066.857363, 1206.146000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
[INFO] [1522964076.756817, 1215.287000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
[INFO] [1522964086.794030, 1224.615000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
[INFO] [1522964096.774418, 1233.876000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
[INFO] [1522964106.474975, 1242.889000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
[INFO] [1522964116.344926, 1252.039000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
[INFO] [1522964125.833993, 1260.717000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
[INFO] [1522964135.710184, 1269.959000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
[INFO] [1522964145.750904, 1279.371000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
[INFO] [1522964155.647939, 1288.645000]: Detected 3 objects: ['biscuits', 'soap'
, 'soap2']
```

World 2 Recognition

Running the simulator with the test2.world we obtain the following recognition:

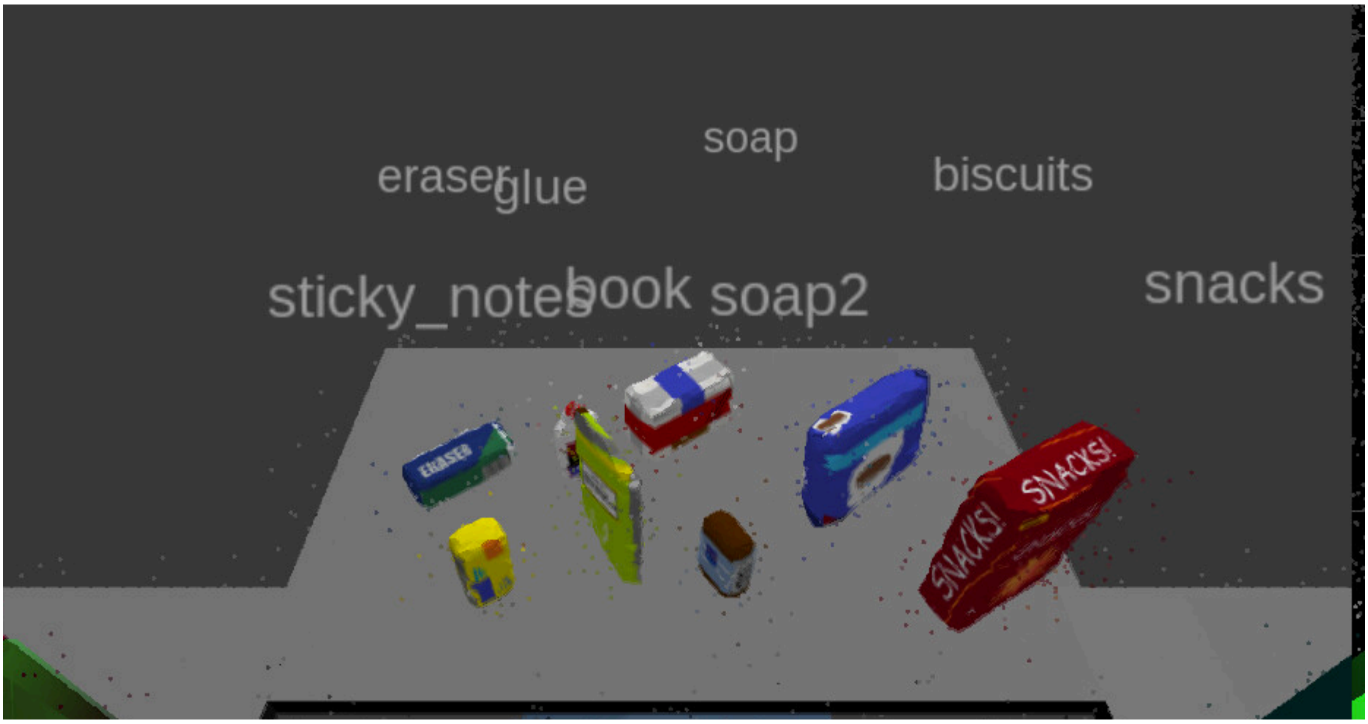
And the console output:



```
ros@ros-VM: ~/catkin_ws/src/RoboND-Perception-Project/pr2_robot/scripts
, 'soap', 'soap2', 'glue']
[INFO] [1522964537.320852, 1337.285000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
[INFO] [1522964547.236057, 1346.507000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
[INFO] [1522964557.026049, 1355.664000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
[INFO] [1522964566.590599, 1364.647000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
[INFO] [1522964576.571468, 1374.074000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
[INFO] [1522964586.352352, 1383.264000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
[INFO] [1522964596.419031, 1392.800000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
[INFO] [1522964606.289396, 1402.131000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
[INFO] [1522964616.289349, 1411.570000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
[INFO] [1522964626.206287, 1420.808000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
[INFO] [1522964636.336406, 1430.142000]: Detected 5 objects: ['biscuits', 'book'
, 'soap', 'soap2', 'glue']
```

World 3 Recognition

Running the simulator with the test3.world we obtain the following recognition:



And the console output:

```
ros@ros-VM: ~/catkin_ws/src/RoboND-Perception-Project/pr2_robot/scripts
r2_robot perception.py
[INFO] [1522964888.532362, 1449.373000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'eraser', 'soap2', 'sticky_notes', 'glue']
[INFO] [1522964899.110547, 1459.069000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'eraser', 'soap2', 'sticky_notes', 'glue']
[INFO] [1522964909.300020, 1468.460000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'soap2', 'eraser', 'sticky_notes', 'glue']
[INFO] [1522964920.411923, 1478.758000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'soap2', 'eraser', 'sticky_notes', 'glue']
[INFO] [1522964931.436046, 1489.104000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'eraser', 'soap2', 'sticky_notes', 'glue']
[INFO] [1522964942.295084, 1499.240000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'eraser', 'soap2', 'sticky_notes', 'glue']
[INFO] [1522964952.808894, 1509.009000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'eraser', 'soap2', 'sticky_notes', 'glue']
[INFO] [1522964962.846448, 1518.252000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'eraser', 'soap2', 'sticky_notes', 'glue']
[INFO] [1522964973.051500, 1527.660000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'eraser', 'soap2', 'sticky_notes', 'glue']
[INFO] [1522964983.535934, 1537.592000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'eraser', 'soap2', 'sticky_notes', 'glue']
[INFO] [1522964994.085549, 1547.603000]: Detected 8 objects: ['snacks', 'biscuit
s', 'book', 'soap', 'eraser', 'soap2', 'sticky_notes', 'glue']
```

Creating the commands for picking and YAML command file

At the end of the call-back function we have included the following code that calls the `pr2_mover` function:

```
if len(detected_objects) > 0:
    try:
        pr2_mover(detected_objects)
```

```
        except rospy.ROSInterruptException:
            pass
```

In the `pr2_mover` we implement the following:

First, we identify the execution parameters: the pick list, the world, the position of the dropboxes and their characteristics. We also create an empty list where we will store the dictionary of commands that we will output to YAML later:

```
    # get parameters:
    #
    # pick list
    object_list_param = rospy.get_param('/object_list')
    object_names = [item['name'] for item in object_list_param]
    object_groups = [item['group'] for item in object_list_param]

    # world; there is no simple way of finding the world_name because it is
    # passed as an argument to the gazebo node; we use the numer of objects
    # to infer the world
    world_dict = {3:1, 5:2, 8:3}
    try:
        test_scene_num_msg = Int32()
        test_scene_num_msg.data = world_dict[len(object_list_param)]
        # also create the name of the yaml output file
        yaml_file = 'output_'+str(world_dict[len(object_list_param)])+'.yaml'
    except KeyError:
        rospy.logwarn('Scene cannot be determined for a pick list with {}
objects.'.format(len(object_list_param)))
        return

    # dropboxes
    dropbox_param = rospy.get_param('/dropbox')
    dropboxes = {}
    for item in dropbox_param:
        dropboxes[item['group']] = item

    # prepare the command list for yaml output
    command_list = []
```

It's good to note here that identifying the world is significantly more complicated as first imagined as this is pass as an argument to the Gazebo node at start and is not available in the parameters server. We could load the .launch file and scan it for the world line, but instead we chose to simply match the world by the number of items we are asked to pick: 3 for world 1, 5 for world 2 and 8 for world 3.
The dropboxes are stored in a dictionary by the colour so that later it will be easier to retrieve the position of the box based on the colour requested in the pick list.

We now go through the list of the items in the pick list (object_names):

```
    # Loop through the pick list
    for index, label in enumerate(object_names):
```

and perform the following actions: first we try to find from the list of recognized objects (passed as a parameter to the pr2_mover function) the object that matches the current item to pick:

```
        # Get the PointCloud for a given object and obtain it's centroid
```

```
        try:
            detected_object = next((obj for obj in detected_objects_list if obj.label
== label))
        except StopIteration:
            # the requested object in the pick list does not exist in
            # the detected objects; issue warning and move to the next in
            # pick list
            rospy.logwarn('Object: {} from pick list was not
detected.'.format(label))
            continue
```

In case the requested object is not in the list of detected objects we issue a warning message and move to the next item in the pick list.

With that detected object found we determine the centroid of the point cloud and prepare a Pose() message that will represent the pick position:

```
        # calculate the centroid
        points_arr = ros_to_pcl(detected_object.cloud).to_array()
        centroid = np.mean(points_arr, axis=0)[:3]
        pick_pose = Pose()
        pick_pose.position.x = np.asscalar(centroid[0])
        pick_pose.position.y = np.asscalar(centroid[1])
        pick_pose.position.z = np.asscalar(centroid[2])
```

For the drop pose we are looking into the dropboxes variable that we read earlier from the parameter server, specifically we pick the dropbox with the colour as indicated in the pick list. We then simply use the position attributes of the dropbox to create the end pose for robot:

```
        # Create 'place_pose' for the object
        place_pose = Pose()
        dropbox_pos = dropboxes[object_groups[index]]['position']
        print('dropbox position: '+str(dropbox_pos))
        place_pose.position.x = dropbox_pos[0]
        place_pose.position.y = dropbox_pos[1]
        place_pose.position.z = dropbox_pos[2]
```

The arm to be used is also determined from the definition of the dropboxes, as the attribe ‘name’ of the dropbox specifies if it is the ‘left’ or the ‘right’ one:

```
        # Assign the arm to be used for pick_place
        which_arm_msg = String()
        which_arm_msg.data = dropboxes[object_groups[index]]['name']
```

Finally, we create a ROS message for the name of the object we are picking, simply using the label provided in the pick list:

```
        # Object name message
        object_name_msg = String()
        object_name_msg.data = label
```

With all these in place we can use the help function make_yaml_dict to create a dictionary with the command for the current item in the pick list and add it to the list of commands that we will dump to a file later.

```
        # Create a list of dictionaries (made with make_yaml_dict()) for later output
to yaml format
        command_yaml = make_yaml_dict(test_scene_num_msg,
                                      which_arm_msg,
                                      object_name_msg,
                                      pick_pose,
                                      place_pose)
        command_list.append(command_yaml)
```

To issue to ROS commands we call the service `pick_place_routine`. We also issue a message with the content of the call and then the result:

```
        # Wait for 'pick_place_routine' service to come up
        rospy.wait_for_service('pick_place_routine')

        try:
            pick_place_routine = rospy.ServiceProxy('pick_place_routine', PickPlace)
            print('Calling pick_place_routine with:')
            print('    scenene number: '+str(test_scene_num_msg.data))
            print('    object name   : '+str(object_name_msg.data))
            print('    which arm     : '+str(which_arm_msg.data))
            print('    pick pose     : '+str(pick_pose.position))
            print('    place pose    : '+str(place_pose.position))
            resp = pick_place_routine(test_scene_num_msg,
                                      object_name_msg,
                                      which_arm_msg,
                                      pick_pose,
                                      place_pose)

            print ("Response: ",resp.success)

        except rospy.ServiceException, e:
            print "Service call failed: %s"%e
```

Before finishing the procedure we save the YAML file (will be saved in the current directory from where the node was started) and issue a notice:

```
    # Output your request parameters into output yaml file
    send_to_yaml(yaml_file, command_list)
    rospy.loginfo('YAML file saved: {}.'.format(yaml_file))
```

The content of the 3 YAML files are included in the submission archive.