3D Perception Project Report

◆ Perception Pipeline

1. Pipeline for filtering and RANSAC plane fitting implemented

The raw RGB-D camera data from the PR2 robot contained noise which could have lead to erroneous values at various stages in the perception pipeline. In order to reduce this noise and remove outliers, PCL's statistical outlier filter was implemented as the first filter in the pipeline. Setting the number of neighboring points to analyze for any given point as 100 and threshold scale factor of 0.003 provided a much cleaner camera output dataset.

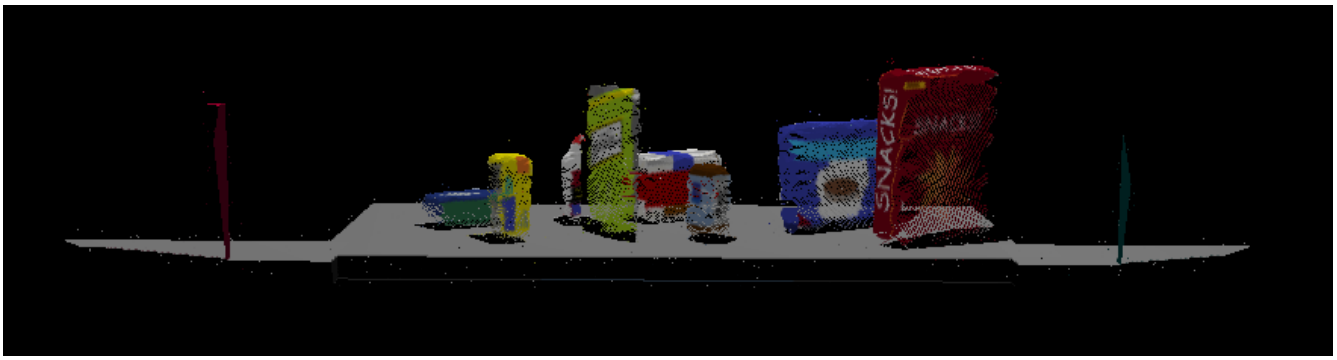


Illustration 1: Statistical Outlier Filtering

In order to reduce computation times on feature rich and dense point clouds from the camera, a VoxelGrid Downsampling filter was applied to the point cloud. The filter works by setting the leaf size (voxel size in cubic meters) along each dimension so as to adjust the sampling size. After experimenting with different values, setting the leaf size to 0.002 downsampled the original point cloud data to much fewer points while retaining its important features.

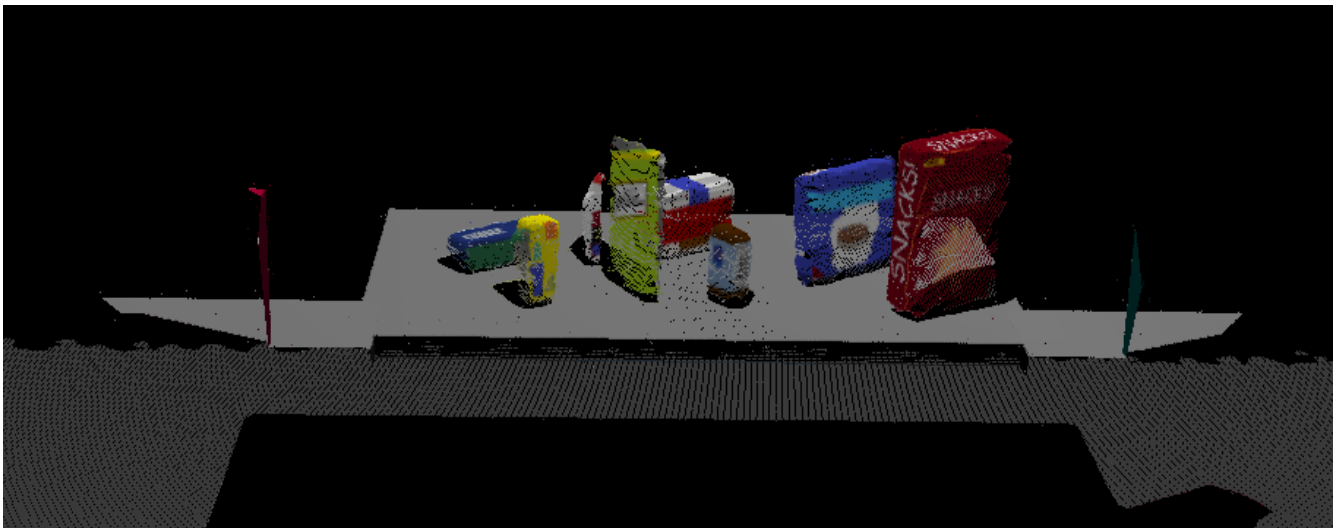


Illustration 2: Voxel Grid Downsampling

The next step was to remove useless data from the point cloud. In this case the objects on the table were the region of interest and the table was useless data which was removed by implementing a

PassThrough filter. Most of the table was cropped out from the scene with the PassThrough filter applied along the z-axis. However, a visible front edge of the table was later removed with an additional filter implemetation along the y-axis.

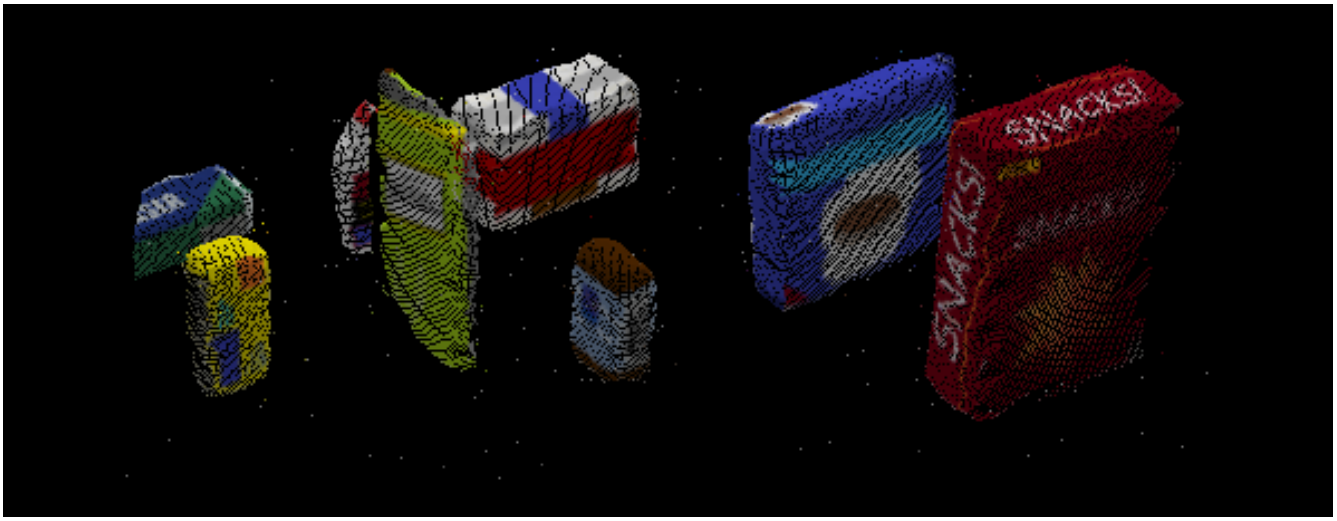


Illustration 3: PassThrough Filtering

After removing extraneous data and noise, a RANSAC algorithm was run with the presence of outliers and inliers in the data set to identify points that belong to a particular model. PCL library's implementation of RANSAC plane fitting algorithm involved creating the segmentation object, setting the model to fit, choosing a maximum distance for a point to be considered fitting the model (max_distance) and then calling the segmentation function to obtain set of inlier indices. An Extractindices filter was then implemented to extract points from the point cloud by providing a list of indices. The inliers correspond to the point cloud indices that were within the max_distance of the best fit model and represented the objects. Two point clouds were obtained containing objects (*cloud_objects*) and table (*cloud_table*) separately.

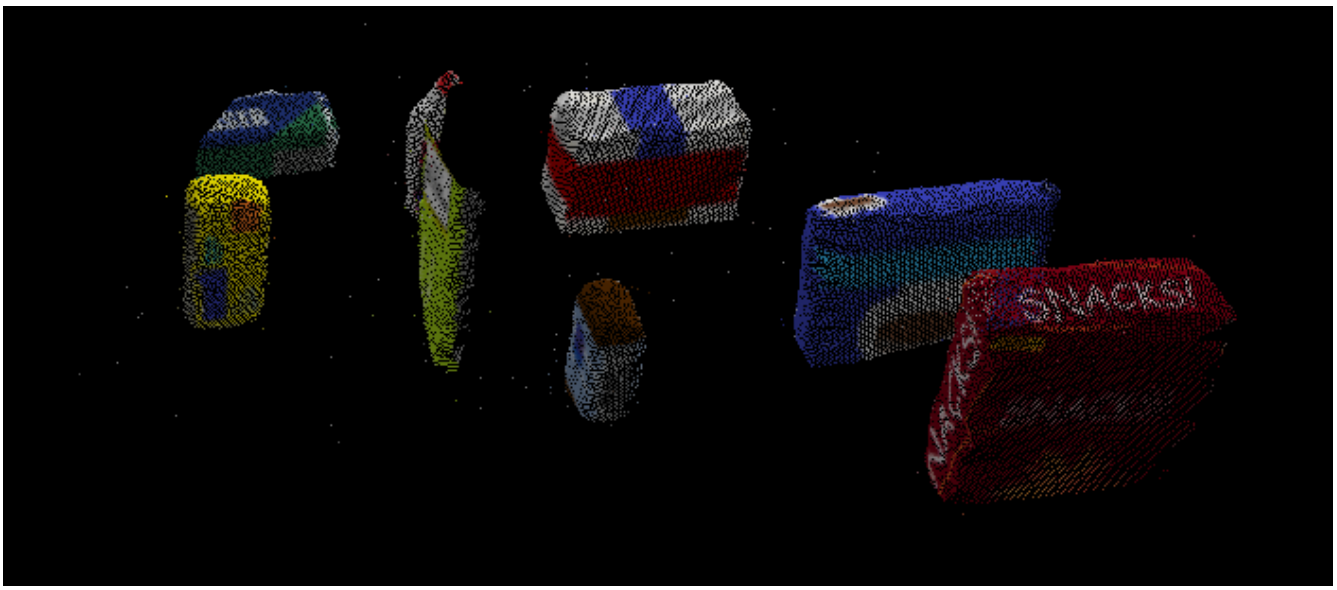


Illustration 4: Extracted inliers from RANSAC plane fitting

```
    # TODO: Statistical Outlier Filtering
    outlier_filter = cloud.make_statistical_outlier_filter()
    outlier_filter.set_mean_k(100)
    outlier_filter.set_std_dev_mul_thresh(0.003)
    cloud_filtered = outlier_filter.filter()
    pcl.save(cloud_filtered, "project_table_scene.pcd")

    # TODO: Voxel Grid Downsampling
    vox = cloud_filtered.make_voxel_grid_filter()
    LEAF_SIZE = 0.002
    vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
    cloud_filtered = vox.filter()
    pcl.save(cloud_filtered, "voxel_downsampled.pcd")

    # TODO: PassThrough Filter
    passthrough_z= cloud_filtered.make_passthrough_filter()
    passthrough_z.set_filter_field_name("z")
    passthrough_z.set_filter_limits(0.62, 1.2)
    cloud_filtered_z = passthrough_z.filter()
    passthrough_y= cloud_filtered_z.make_passthrough_filter()
    passthrough_y.set_filter_field_name("y")
    passthrough_y.set_filter_limits(-0.50, 0.3)
    cloud_filtered = passthrough_y.filter()
    pcl.save(cloud_filtered, "pass_through_filtered.pcd")

    # TODO: RANSAC Plane Segmentation
    seg = cloud_filtered.make_segmenter()
    seg.set_model_type(pcl.SACMODEL_PLANE)
    seg.set_method_type(pcl.SAC_RANSAC)
    max_distance = 0.08
    seg.set_distance_threshold(max_distance)
    inliers, coefficients = seg.segment()

    # TODO: Extract inliers and outliers
    cloud_objects = cloud_filtered.extract(inliers, negative=False)
    pcl.save(cloud_objects, "extracted_inliers.pcd")

    # Extract outliers
    cloud_table = cloud_filtered.extract(inliers, negative=True)
    pcl.save(cloud_table, "extracted_outliers.pcd")
```

Illustration 5: Code block for implementing filters and RANSAC plane-fitting

2. Pipeline including clustering for segmentation implemented

Next step in the pipeline, the *cloud_objects* point cloud was further segmented into meaningful pieces using Euclidean Clustering. Solely using RANSAC for segmenting the data would have lead to erroneous results since a particular model fitting algorithm is not good enough to differentiate objects with similar shapes, sizes or other features. On the contrary, to cover all shapes and sizes, running different RANSAC algorithms on the same point cloud data for multiple model fitting objectives would have been computation heavy and not optimal.

Utilizing PCL's EuclideanClusterExtraction() function, DBSCAN cluster search was performed on the point cloud data in a ROS and Gazebo environment. DBSCAN groups together points in a cluster based on a threshold distance from the nearest other point in the data. The technique does not need a convergence criteria and works best on point cloud data where the number of clusters are unknown. A

list of indices (*cluster_indices*) for each cluster was obtained after playing around with different values for *cluster tolerance*, *minimum cluster size* and *maximum cluster size.* To see these clusters separately, each segmented object was assigned a color and a new point cloud () was created to then visualize the results in Rviz.

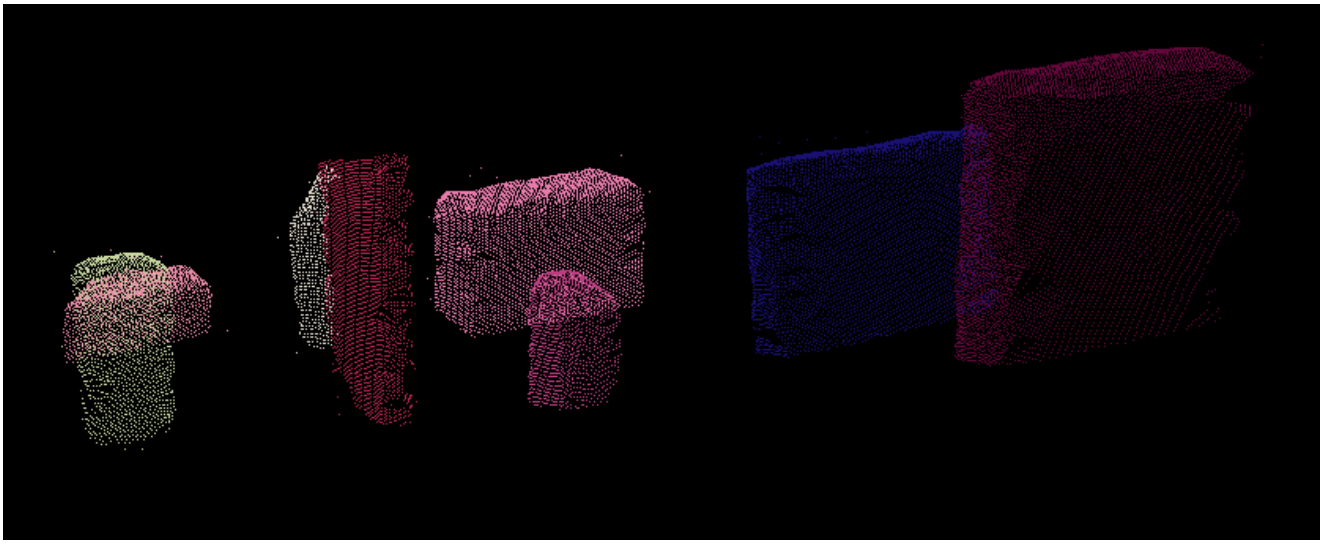


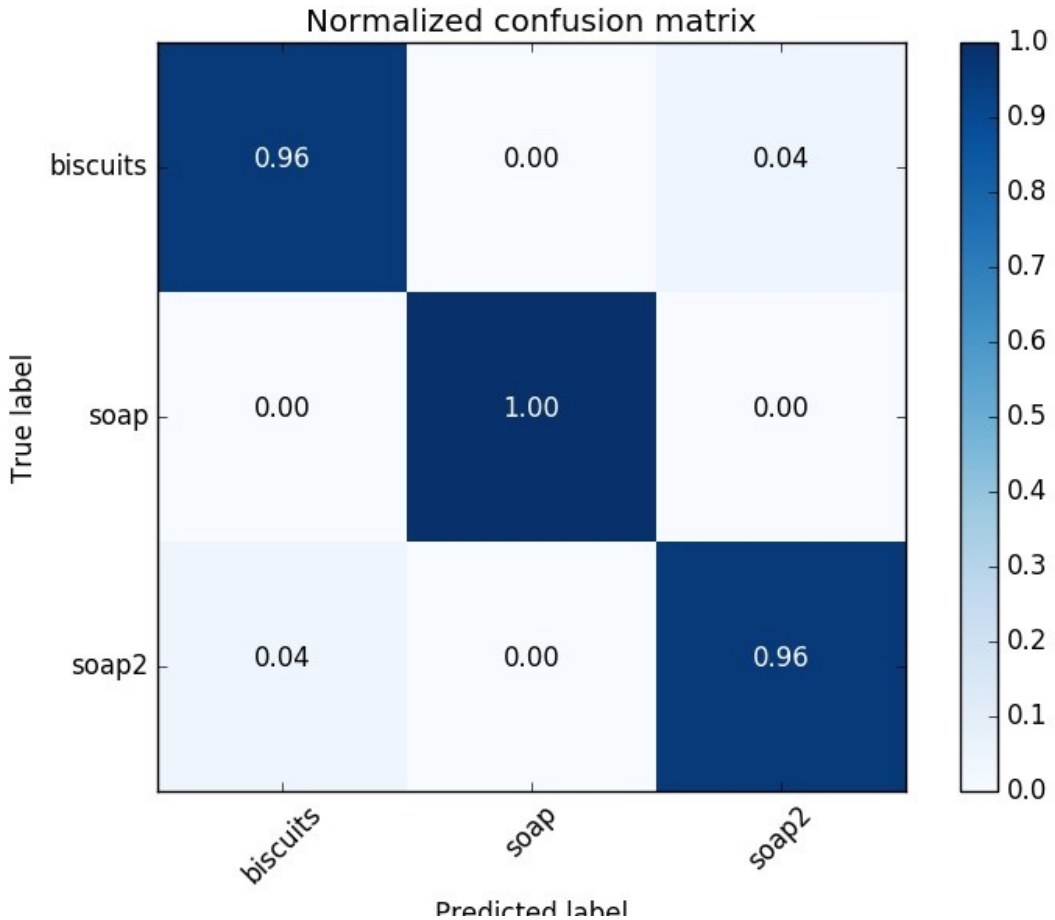
Illustration 6: Euclidean Clustering

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

    # TODO: Create Cluster-Mask Point Cloud to visualize each cluster separately
    ec = white_cloud.make_EuclideanClusterExtraction()
    ec.set_ClusterTolerance(0.02)
    ec.set_MinClusterSize(300)
    ec.set_MaxClusterSize(10000)
    # 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_cluster_cloud = pcl_to_ros(cluster_cloud)

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

Illustration 7: Code block to implement Euclidean Clustering

3. Features extracted and SVM trained. Object recognition implemented

For object recognition in each scene of the given 3 different world scenarios, a training set of features and labels was generated for the objects present in that particular world's pick list. Utilizing Gazebo environment with *capture_features.py and features.py* script in sensor stick module, histogram features and shapes for each object were extracted given bin size and pixel intensity range. The generated training set was then used to train a Support Vector Machine (SVM) classifier with the *train_svm.py* script. SVM algorithm allowed to characterize the entire parameter space into discrete classes. After experimenting with various kernels (*rbf, poly, sigmoid, precomputed and callable*), a basic SVM with a *linear* kernel was implemented in this case. The linear kernel turned out to be faster to train the classifier and better in it's classification job. The classifier was then used to predict objects in the segmented point cloud. The overall accuracy of the classifier was provided in form of a confusion matrix which represented the matching of predicted label with the true label of all the objects in the scene. The accuracy of classifier was improved by tuning the parameters of feature extraction (number of bins), by choosing the most optimal kernel for SVM and by increasing the number of feature capture iterations to get a larger set of random orientations of the objects.



```
    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)
```

Illustration 8: Code block for object detection

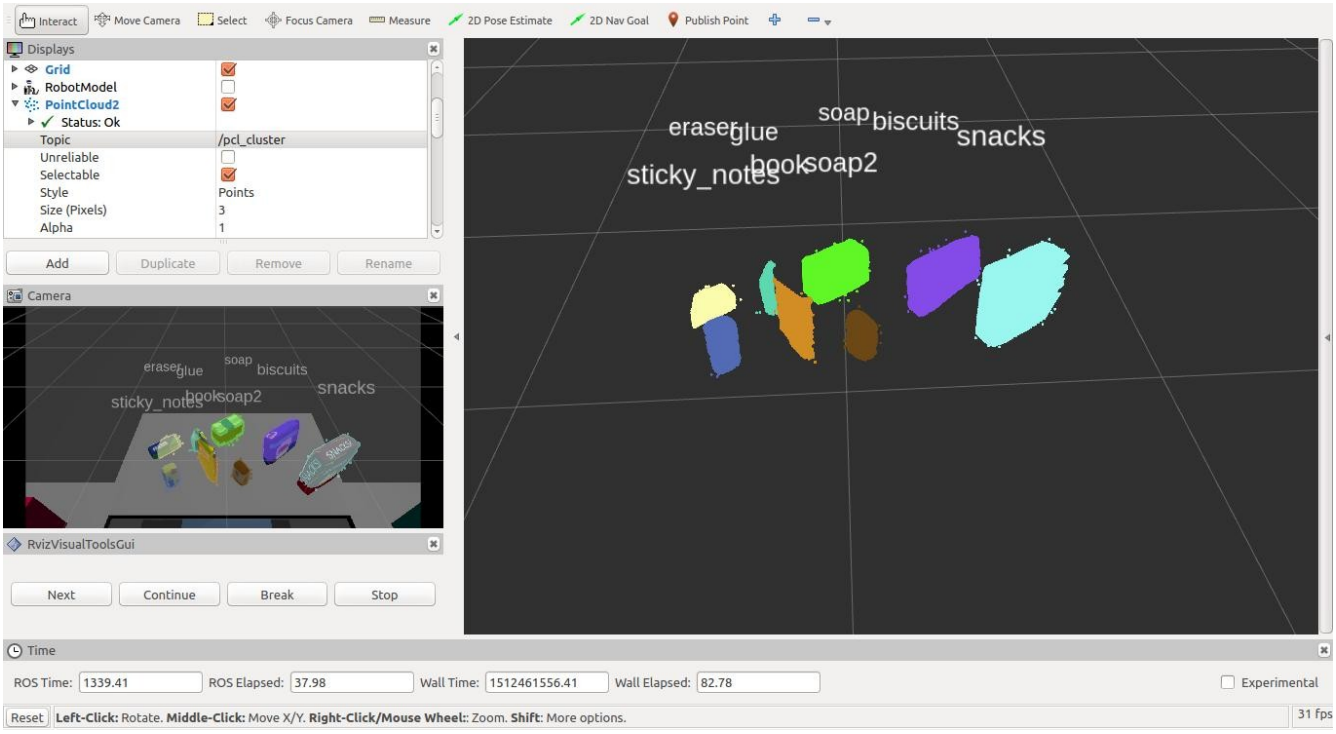
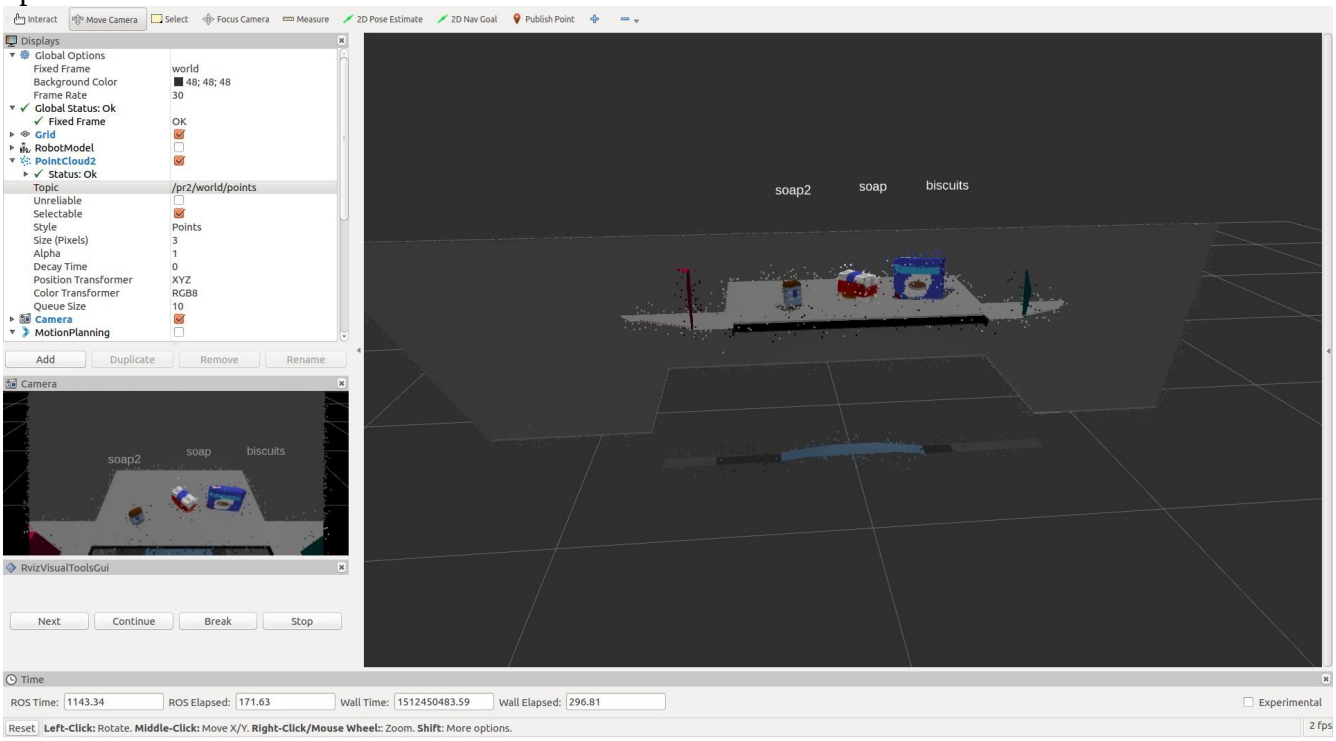


Illustration 9: Vizualizing clustering and object recognition in Rviz

◆ Pick and Place Setup

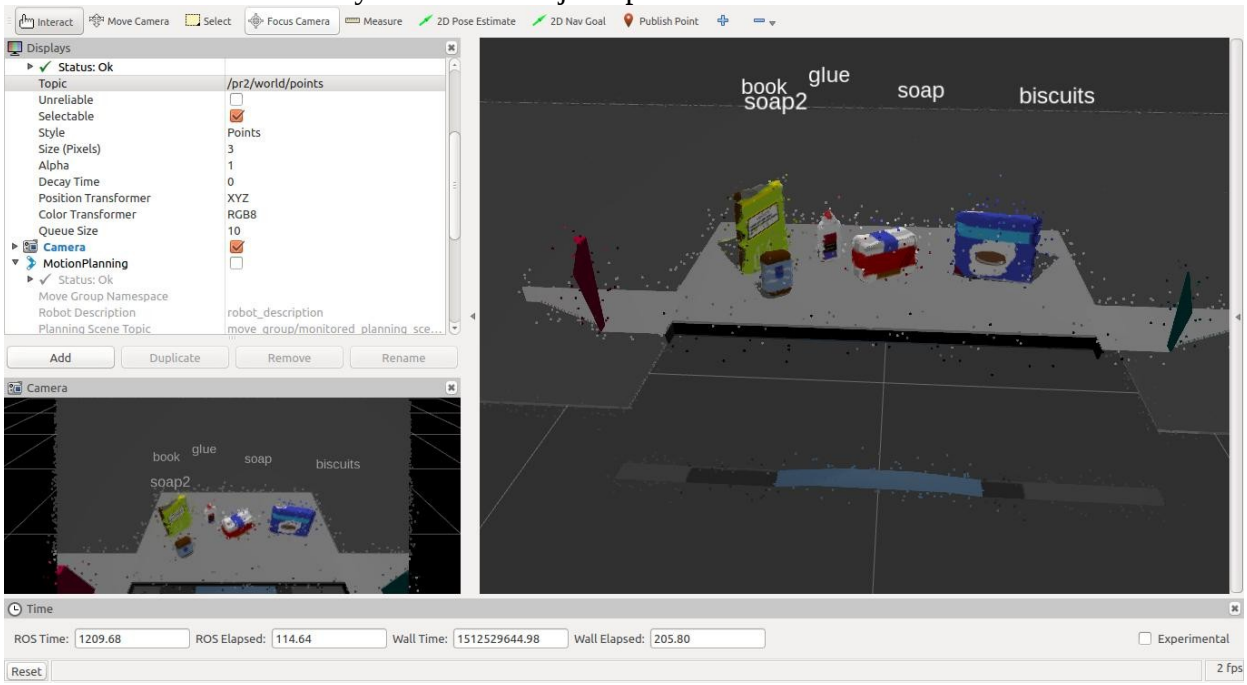
1. World One Scene

The model was able to correctly detect all 3 objects present in this scenario and created .yaml file with correct message contents, including the place pose, for a service request to pick_place_server operation.



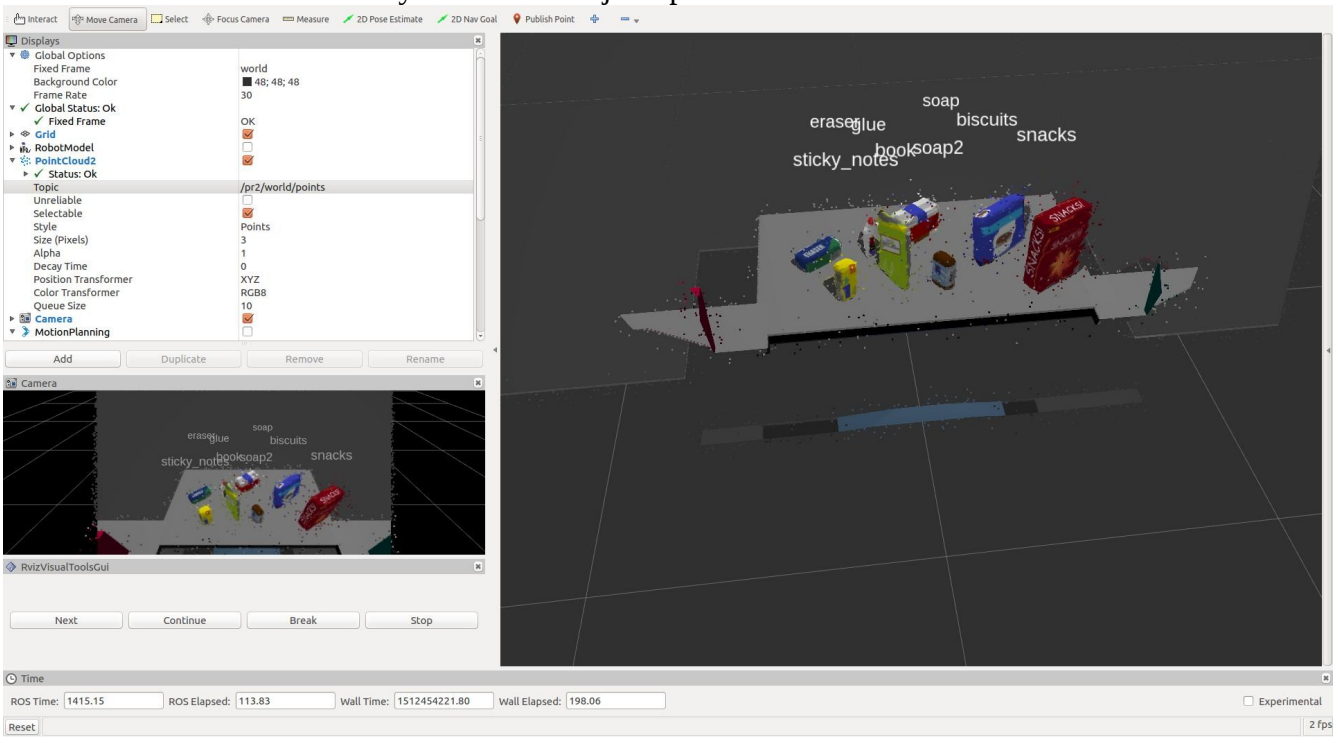
2. World Two Scenario

The model was able to correctly detect all 5 objects present in this scenario.



3. World Three Scenario

The model was able to correctly detect all 8 objects present in this scenario.



Improvements

One of the biggest challenges in object recognition part was to detect the presence of glue bottle in world 3 scene. Since a big part of the glue was behind the book on the table, increasing the number of iterations to capture more orientations of glue bottle helped immensely to improve the accuracy of detection. Several other tune ups in the first phase of process like leaf size selection in voxel grid downsampling and cluster sizing in segmentation also contributed to a more robust detection. In case of pursuing this project further, I would like to investigate different implementations of SVM and also look into other classification algorithms. Increasing the number of bins in the histogram extraction module would be another area to investigate which might contribute to a better detection model as well.