3D Perception Report

Introduction

In this project, we used pr2 robot and its RGBD camera to build a perception pipeline for picking target objects on the table and dropping them into corresponding box that the target should belong to.

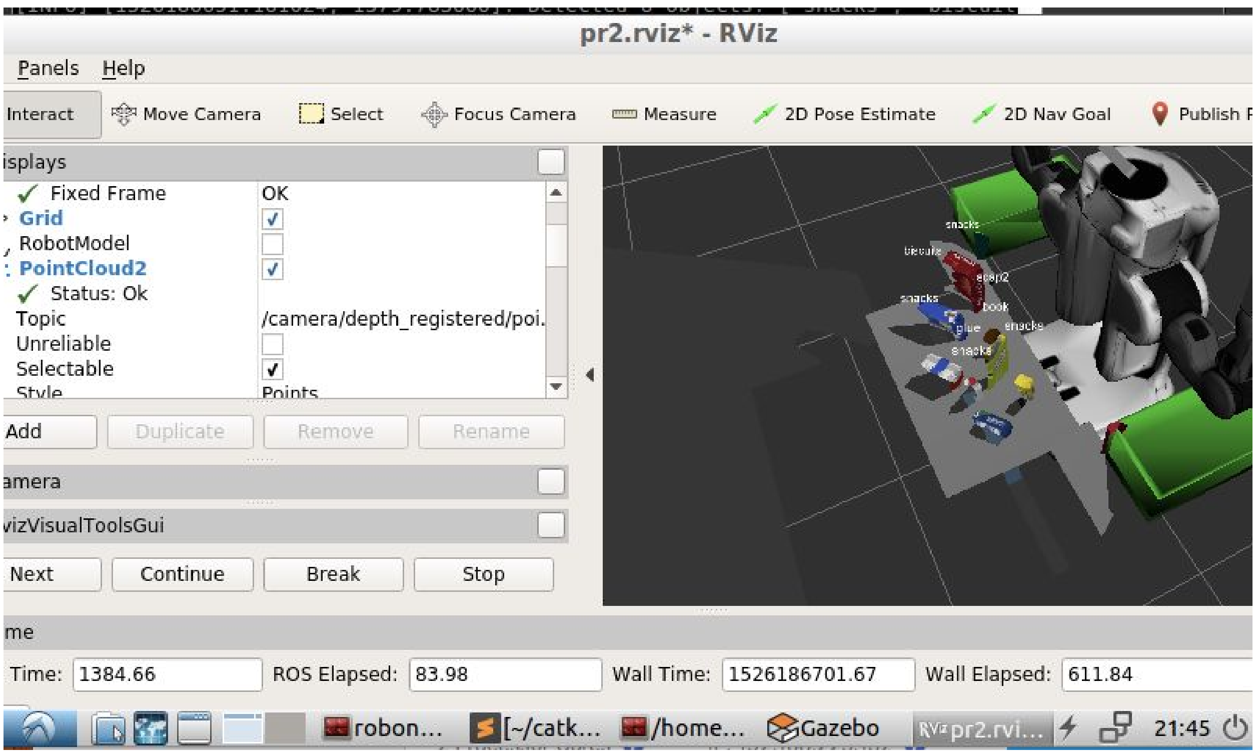

Figure 1: objects labelling results from pr2 robot’s RGBD camera

We applied clustering for segmentation technique to let pr2 robot recognize the object on the table. We will use the following steps to build pr2 perception pipeline.

1. Convert ROS msg to PCL data
2. Voxel grid downsampling
3. Statistical outlier filtering (denoise the camera data)
4. Table Segmentation (separate the objects from the table)
5. Euclidean clustering (group different objects into distinct clusters)
6. Object recognition
    a. Generate a training set of features for the objects in the pick lists
    b. Train your SVM with features from the new models

c. Apply object recognition code to pick and drop targets

Exercise 1, 2, and 3 Pipeline Implementation:

Exercise 1's purpose is to finish step 1 - 4 above.

```
# TODO: Convert ROS msg to PCL data
cloud = ros_to_pcl(pcl_msg)

# TODO: Voxel Grid Downsampling
vox = cloud.make_voxel_grid_filter()
LEAF_SIZE = 0.003
vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
cloud_filtered = vox.filter()

# TODO: Statistical Outlier Filtering
outlier_filter = cloud_filtered.make_statistical_outlier_filter()
outlier_filter.set_mean_k(50)
x = 1.0
outlier_filter.set_std_dev_mul_thresh(x)
cloud_filtered = outlier_filter.filter()

# TODO: PassThrough Filter
passThroughZ = cloud_filtered.make_passthrough_filter()
filter_axis = 'z'
passThroughZ.set_filter_field_name(filter_axis)
axis_min = 0.6
axis_max = 1.1
passThroughZ.set_filter_limits(axis_min, axis_max)
cloud_filtered = passThroughZ.filter()
passThroughY = cloud_filtered.make_passthrough_filter()
filter_axis = 'y'
passThroughY.set_filter_field_name(filter_axis)
axis_min = -0.5
axis_max = 0.5
passThroughY.set_filter_limits(axis_min, axis_max)
cloud_filtered = passThroughY.filter()

# 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 = 0.01
seg.set_distance_threshold(max_distance)
inliers, coefficients = seg.segment()
```

First, we took a spatial average of the points in each voxel with 0.003m side length to downsample the 3D point cloud. Second, use Statistical Outlier Filtering to remove the white noise from the camera input data. Third, use passthrough filter to select a region of interest from the Voxel Downsample Filtered point cloud. we know that the table is roughly in the center of our robot’s field of view. Hence by using a Pass Through Filter we can select a region of interest to remove some of the excess data. Then, applying a Pass Through filter along z axis (the height with respect to the ground) to our tabletop scene in the range 0.6m to 1.1m, and along y axis with range -0.5m and 0.5m. Lastly, with max distance 0.01 RANSAC plane fitting algorithm, we can complete segment the table object, which is shown in Figure 2.

Figure 2: table segmentation with passThrough filter.

Exercise 2 is for using Density-Based Spatial Clustering of Applications with Noise (DBSCAN) algorithm, also called Euclidean Clustering algorithm to segment the outliers into individual objects. We chose to use DBSCAN Algorithm because we don't know how many clusters to expect in the camera data, but we know that table objects should be clustered in terms of density.

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

    # TODO: Create Cluster-Mask Point Cloud to visualize each cluster
separately
    ec = white_cloud.make_EuclideanClusterExtraction()
    ec.set_ClusterTolerance(0.005)
    ec.set_MinClusterSize(150)
    ec.set_MaxClusterSize(50000)
    ec.set_SearchMethod(tree)
    cluster_indices = ec.Extract()
    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])])
    cluster_cloud = pcl.PointCloud_PointXYZRGB()
    cluster_cloud.from_list(color_cluster_point_list)
```

By using the above algorithm, we can obtain the result shown in Figure 3.

Figure 3: result for applying DBSCAN algorithm

In exercise 3, object features need to be got extracted and SVM got trained. Applying an SVM to table objects point cloud allows us to characterize each object into discrete classes. The divisions between classes in parameter space are known as "decision boundaries", shown here by the colored polygons overlaid on the data (shown in figure 3). In exercise 3, we applied both color histograms and normal histograms of the objects to generate features to train our SVM.

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

    # 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])

    # TODO: Compute histograms
    r_hist = np.histogram(channel_1_vals, bins=32, range=(0, 256))
    g_hist = np.histogram(channel_2_vals, bins=32, range=(0, 256))
    b_hist = np.histogram(channel_3_vals, bins=32, range=(0, 256))

    # TODO: Concatenate and normalize the histograms
    hist_features = np.concatenate((r_hist[0], g_hist[0],
b_hist[0])).astype(np.float64);
    normed_features = hist_features / np.sum(hist_features)
    # Generate random features for demo mode.
    # Replace normed_features with your feature vector

    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])

    # TODO: Compute histograms of normal values (just like with color)
    x_hist = np.histogram(norm_x_vals, bins=32, range=(-1,1))
    y_hist = np.histogram(norm_y_vals, bins=32, range=(-1,1))
```

```
    z_hist = np.histogram(norm_z_vals, bins=32, range=(-1,1))
    # TODO: 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)

    # Generate random features for demo mode.
    # Replace normed_features with your feature vector

    return normed_features
```

Then, we applied these two functions to compute the associated feature vector, make prediction and label the objects.

```
# Exercise-3 TODOs:

    # 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 = extracted_outliers.extract(pts_list)
        ros_cluster = pcl_to_ros(pcl_cluster)

        # Compute the associated feature vector
        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)))
        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)
```

To improve of the accuracy of our SVM model, we need to modify how many times each object is spawned randomly by changing the loop number in capture_features.py that begins with for i in range(5). The strategy for me was that the more objects in the pick list, the higher loop number will be, so for the pick_list_1, 45 times each object will be spawned randomly. For pick_list_2, 50 times each object will be spawned randomly. For pick_list_3, 85 each object will be spawned randomly. The results are shown in Figure 4, 5, 6

Figure 4: normalized confusion matrix for pick_list_1

Figure 5: normalized confusion matrix for pick_list_2

Figure 6: normalized confusion matrix for pick_list_3

Pick and Place Setup

The following code is for guiding pr2 to pick up the recognized objects in the list and place the object into the box it belongs to. At the same time, the code sent PickPlace requests into output_1.yaml, output_2.yaml, and output_3.yaml for each scene_1, scene_2, and scene_3 respectively. The results shown in Figure 8, 9, 10.

```
# function to load parameters and request PickPlace service
def pr2_mover(object_list):
    print("Inside pr2_mover")

    # TODO: Initialize variables
    outputFileName = "output_3.yaml";
    object_list_param = rospy.get_param('/object_list');
    test_scene_num = Int32()
    test_scene_num.data = 3
```

```
    dict_list = []
    # TODO: Get/Read parameters
    box_param = rospy.get_param('/dropbox')
    box_name = []
    box_group = []
    box_position = []
    for i in range(0, len(box_param)):
        # TODO: Parse parameters into individual variables
        box_name.append(box_param[i]['name'])
        box_group.append(box_param[i]['group'])
        box_position.append(box_param[i]['position'])
    # TODO: Loop through the pick list
    for i in range(0, len(object_list_param)):
        labels = []
        # TODO: Get the PointCloud for a given object and obtain it's
centroid
        centroids = [] # to be list of tuples (x, y, z)
        for object in object_list:
            # TODO: Parse parameters into individual variables
            labels.append(object.label)
            points_arr = ros_to_pcl(object.cloud).to_array()
            centroids.append(np.mean(points_arr, axis=0)[:3])
    # TODO: Rotate PR2 in place to capture side tables for the collision
map
        object_name = String()
        object_name.data = object_list_param[i]['name']
        object_group = object_list_param[i]['group']

        # TODO: Create 'place_pose' for the object
        place_pose = Pose()

        # TODO: Assign the arm to be used for pick_place
        arm_name = String()
        if object_group == 'red':
            arm_name.data = 'left'
            place_pose.position.x = box_position[0][0]
            place_pose.position.y = box_position[0][1]
            place_pose.position.z = box_position[0][2]
        else:
            arm_name.data = 'right'
            place_pose.position.x = box_position[1][0]
            place_pose.position.y = box_position[1][1]
```

```
            place_pose.position.z = box_position[1][2]
        # TODO: Create a list of dictionaries (made with make_yaml_dict())
for later output to yaml format
        pick_pose = Pose()
        desired_object = object_list_param[i]['name']
        print "\n\nPicking up ", desired_object, "\n\n"
        print "\n\nPutting in ", object_group, "\n\n"

        try:
            labelPosition = labels.index(desired_object)
            pick_pose.position.x = np.asscalar(centroids[labelPosition][0])
            pick_pose.position.y = np.asscalar(centroids[labelPosition][1])
            pick_pose.position.z = np.asscalar(centroids[labelPosition][2])
        except ValueError:
            continue

        yaml_dict = make_yaml_dict(test_scene_num, arm_name, object_name,
pick_pose, place_pose)
        dict_list.append(yaml_dict)
        # 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)

            # TODO: Insert your message variables to be sent as a service
request
            resp = pick_place_routine(test_scene_num, object_name,
arm_name, pick_pose, place_pose)

            print ("Response: ",resp.success)

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

    # TODO: Output your request parameters into output yaml file
    send_to_yaml(outputFileName, dict_list)
```

Figure 7: result for pick_list_1

Figure 8: result for pick_list_2

Figure 9: result for pick_list_2

Figure 10: result for pick_list_3