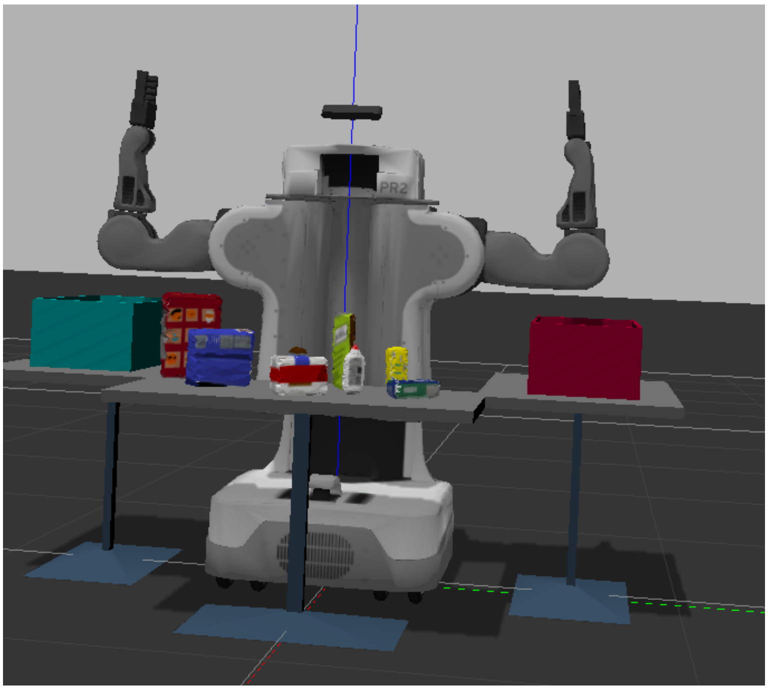
Project 3 of Robotics Nanodegree
Object Detection



October 14, 2017

In this project, we use point cloud data (from topic */pr2/world/points*) and a perception pipeline to detect and identify objects on a table. The implementation of the pipeline involves multiple steps that are outlined in the sections below. The pipeline is used to identify a set of 3, 5 and 8 objects.

0.1 Perception pipeline

0.1.1 Downsampling

The first step consists in using a Voxel Grid Filter. The voxel size (*LEAF_SIZE*) is set to 0.01 m. A higher number results in a loss of resolution.

```
cloud = ros_to_pcl(pcl_msg)
vox = cloud.make_voxel_grid_filter()
LEAF_SIZE = 0.01
vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)
cloud_filtered = vox.filter()
```

0.1.2 Filter noise

Next, we use a statistical outlier filter to *clean up* the point cloud: *i.e* remove the noise. For each data point, we compute the distance from that point to all the k neighboring points. Assuming a Gaussian distribution, the mean μ and standard deviation σ are computed. Any points that are outside the range $\mu \pm x \times \sigma$ are labeled as outliers and removed from the point cloud. x is a scaling factor of the standard deviation..

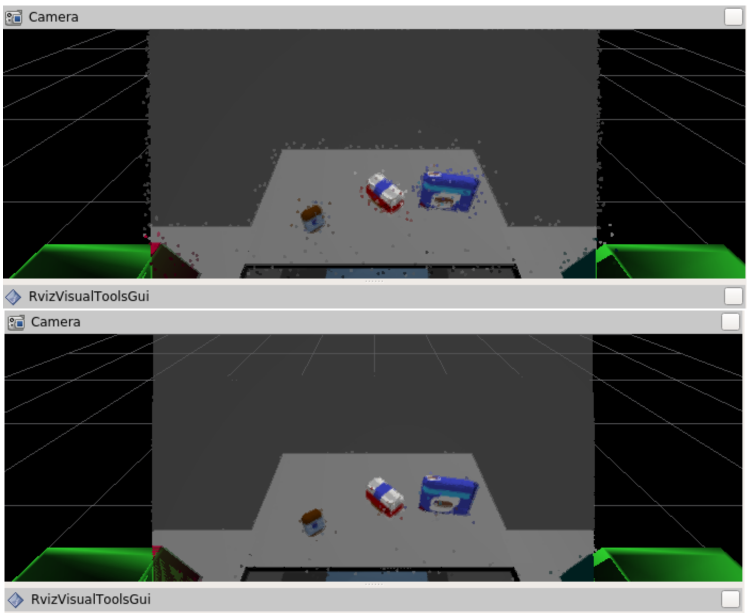


Figure 1: Top image is the original point cloud, and bottom image is the point cloud after statistical outlier filtering.

In our implementation, we set the number of neighboring points to be $k = 50$, and the scaling factor $x = 0.1$.

```
outlier_filter = cloud_filtered.make_statistical_outlier_filter()
```

```
outlier_filter.set_mean_k(50)
x = 0.1
outlier_filter.set_std_dev_mul_thresh(x)
cloud_filtered_outliers = outlier_filter.filter()
```

0.1.3 Passthrough filter

The passthrough filter is a crop-like method that enables to reduce the field of view. The first passthrough filter is along the z axis, where only the points with z coordinates in the range [0.6, 1.1] are preserved.

```
passthrough = cloud_filtered_outliers.make_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)
cloud_filtered = passthrough.filter()
passthrough = cloud_filtered.make_passthrough_filter()
```

However, a single passthrough filter is not sufficient. In fact, the front edges of the 2 drop-boxes remains in the view point of the camera and can alter the object recognition. Therefore, in addition of the z axis passthrough, we also use a passthrough filter along the y axis in the range [-0.35, 0.35].

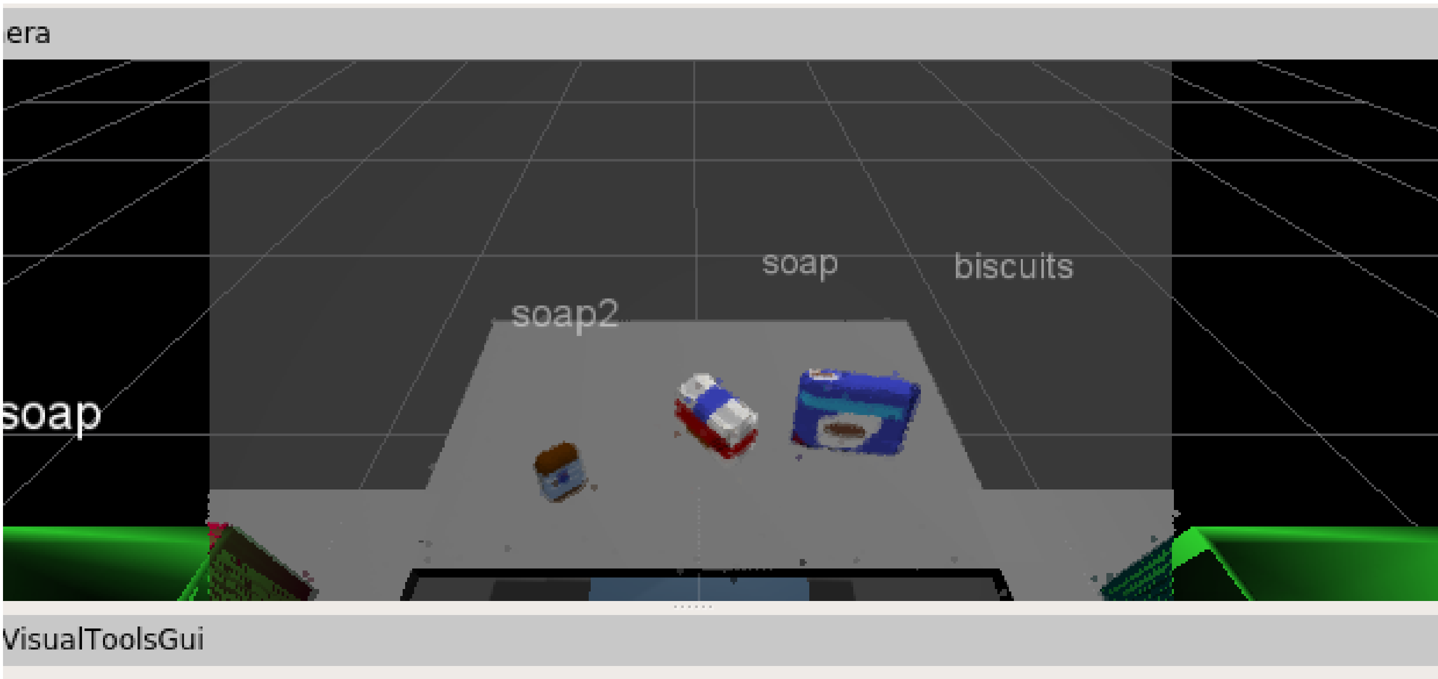


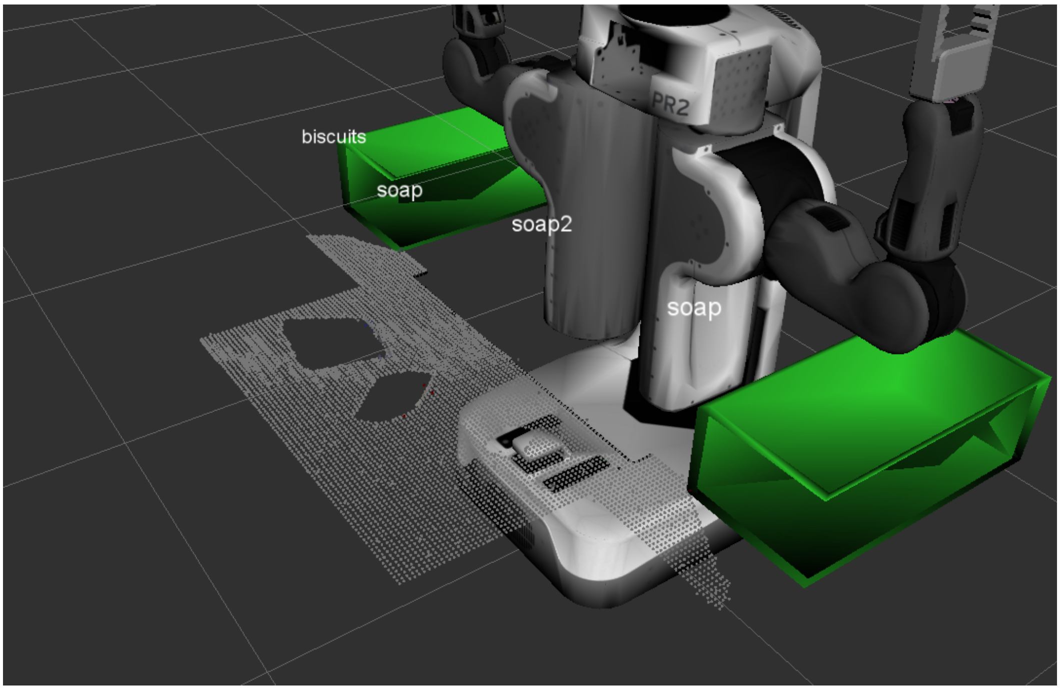
Figure 2: With a passthrough filter along z direction, there remains point clouds because related to the edges of the box (left and right). Those points can lead to erroneous recognition.

```
passthrough = cloud_filtered.make_passthrough_filter()
filter_axis = 'y'
passthrough.set_filter_field_name(filter_axis)
```

```
axis_min = -0.35
axis_max = 0.35
passthrough.set_filter_limits(axis_min, axis_max)
cloud_filtered = passthrough.filter()
```

0.1.4 RANSAC

In this step, we use RANSAC (Random Sample Consensus) algorithm to separate the objects and the table. The table is modeled as a plane to which we fit the entire dataset. We can then extract 2 datasets: the inliers and the outliers. The inliers are the data points that follow the models and as a consequence are most likely associated with the table.

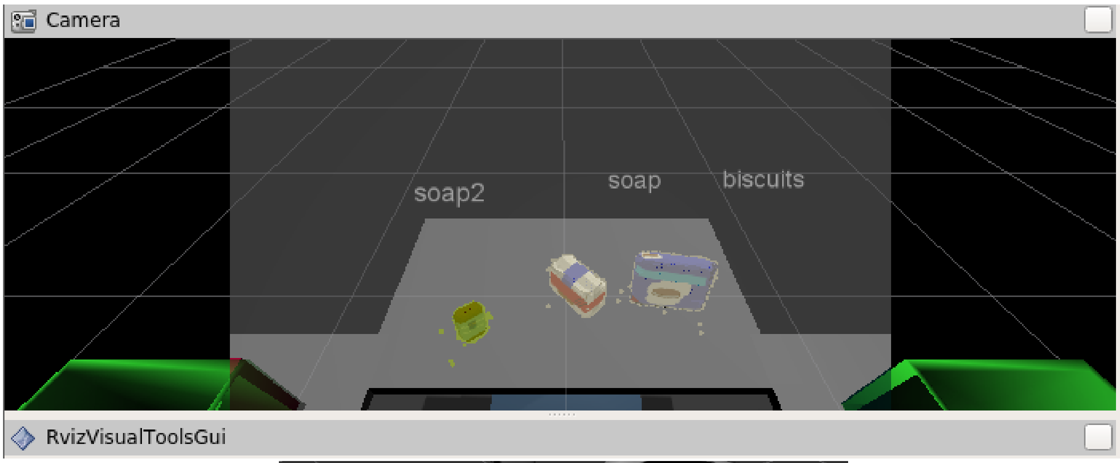


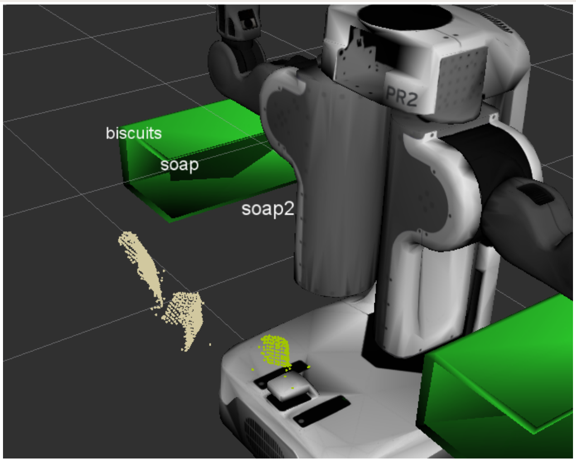

```
seg = cloud_filtered.make_segmenter()
seg.set_model_type(pcl.SACMODEL_PLANE)
seg.set_method_type(pcl.SAC_RANSAC)
max_distance = 0.01
seg.set_distance_threshold(max_distance)
inliers, coefficients = seg.segment()
cloud_table = cloud_filtered.extract(inliers, negative=False)
cloud_objects = cloud_filtered.extract(inliers, negative=True)
```

The hyper-parameter is the *max_distance*: it represents the maximum distance that the datapoint can be from the model best fit and still be considered to be an inlier. We set *max_distance*= 0.01.

0.1.5 Clustering (DBSCAN)

The *DBSCAN* (Density Based Spatial Clustering of Applications with Noise) is a unsupervised learning algorithm used for segmentation. The 2 hyper-parameters are *min_samples*, the minimum number of points that comprise a cluster, and *max_samples* is the maximum distance between cluster points.






```
white_cloud = XYZRGB_to_XYZ(cloud_objects)
tree = white_cloud.make_kdtree()
ec = white_cloud.make_EuclideanClusterExtraction()
ec.set_ClusterTolerance(0.05)
ec.set_MinClusterSize(100)
ec.set_MaxClusterSize(1550)
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)
```

We set the tuning parameters as follows: *cluster_tolerance*= 0.05, *MinClusterSize*= 100, *MaxClusterSize*= 1550.

0.1.6 Object Recognition

Now that we have segmented the point cloud, we need an algorithm to recognize the objects/segments. First, we need to define a set of features to characterize the objects, and then use a supervised learning algorithm to classify the clusters.

Create features

The features are composed of the color histogram and the normal surface histogram. The color histogram is constructed from the HSV channels, with the pixel intensity range [0, 255] splitted into 16 bins. The normal surface histogram has a range [0, 512] and 32 bins. The 2 histograms are then concatenated into a single vector and normalized.

```
chists = compute_color_histograms(sample_cloud, using_hsv=True)
normals = get_normals(sample_cloud)
nhists = compute_normal_histograms(normals)
feature = np.concatenate((chists, nhists))
```

Train model

The classification algorithm is a Support Vector Machine. To train the model, we create a balanced dataset of different objects: *soap*, *soap*2 and *biscuits* for example. We trained our model using features from 150 samples per category. The decision boundary can be either linear or non-linear, so the choice of the kernel in the *SVM* affects the performance of the classifier. We ran a Grid search to find the best performing model:

```
parameter_grid = [{'C':[0.1, 1,10,100,1000], 'kernel': ['linear']},
                  {'C': [0.1, 1, 10, 100, 1000], 'gamma': [0.01, 0.001, 0.000
                  {'C': [0.1, 1, 10, 100, 1000], 'gamma': [0.1, 0.01, 0.001,
clf = GridSearchCV(estimator=svm.SVC(), param_grid=parameter_grid, n_jobs=-1)
....
....
clf.fit(X=X_train, y=y_train)
print('Best C:',clf.best_estimator_.C)
print('Best Kernel:',clf.best_estimator_.kernel)
print('Best Gamma:',clf.best_estimator_.gamma)
```

We also use a $K - fold$ cross validation to train the model. Such an approach is particularly useful when dealing with a small dataset.

The best model is with *kernel=linear*, $C = 1$.

Results

We trained 3 models depending on the number of objects t obe detected.

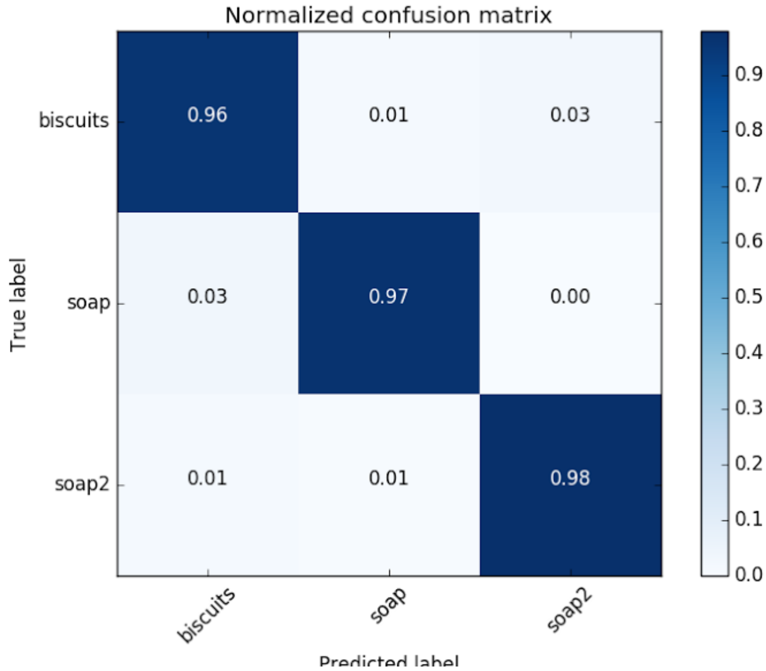


Figure 3: Normalized Confusion matrix for 3 objects (test1). The model has an accuracy of 96.89%

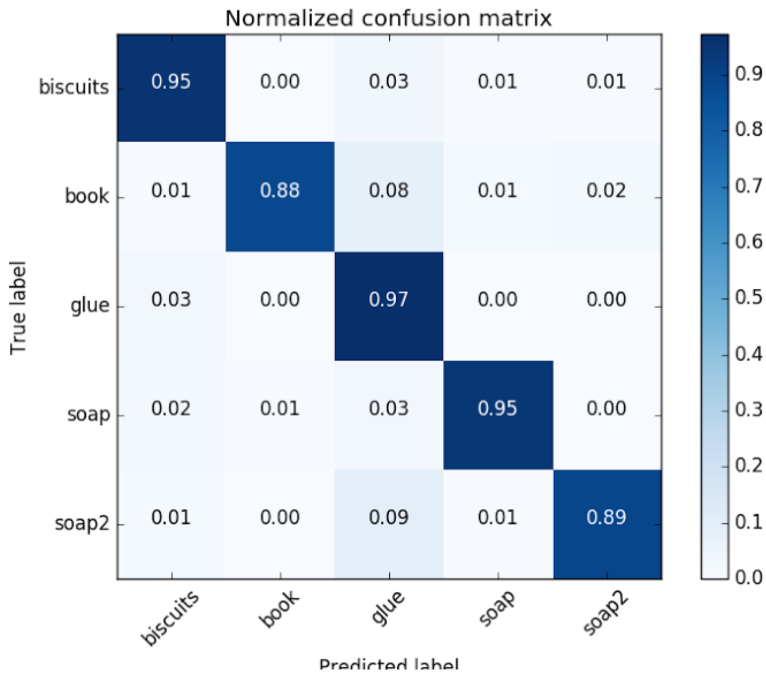


Figure 4: Normalized Confusion matrix for 5 objects (test2). The model has an accuracy of 92.78%

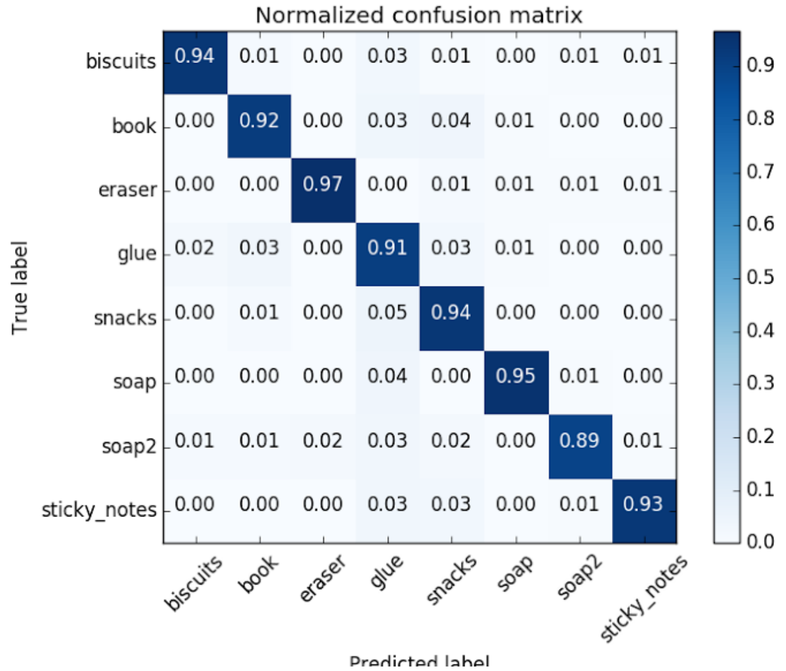


Figure 5: Normalized Confusion matrix for 8 objects (test3). The model has an accuracy of 93%

0.2 Pipeline in action

Below we show the output of the pipeline for the 3 worlds, where 3, 5 and 8 objects must be identified.

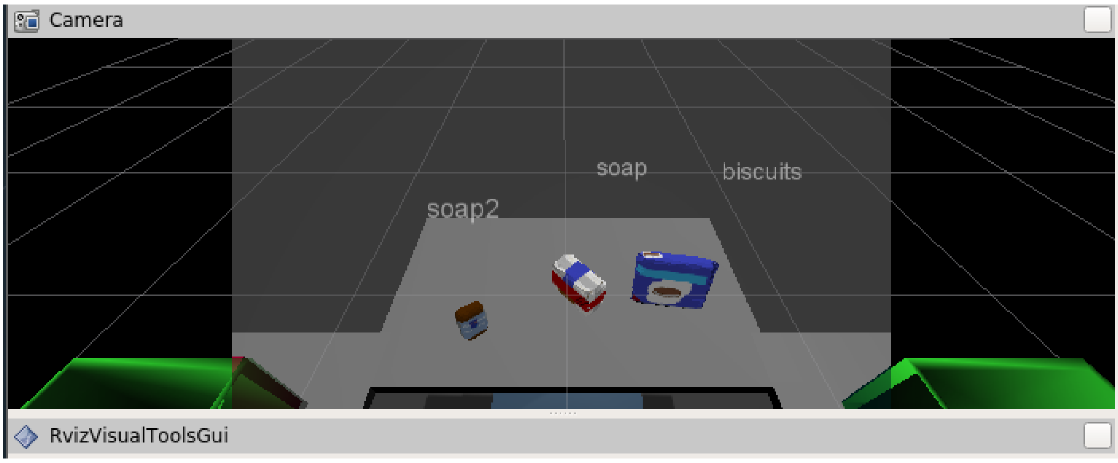


Figure 6: 3 out of 3 objects correctly identified

Figure 7: 5 out of 5 objects correctly identified

Figure 8: 6 out of 8 objects correctly identified. The model cannot separate the glue and the book, and merged into a single cluster that it labels erroneously as soap.