Project: Perception Pick & Place

Brad Huang

Required Steps for a Passing Submission:

1. Extract features and train an SVM model on new objects (see `pick_list_*.yaml` in `/pr2_robot/config/` for the list of models you'll be trying to identify).
2. Write a ROS node and subscribe to `/pr2/world/points` topic. This topic contains noisy point cloud data that you must work with.
3. Use filtering and RANSAC plane fitting to isolate the objects of interest from the rest of the scene.
4. Apply Euclidean clustering to create separate clusters for individual items.
5. Perform object recognition on these objects and assign them labels (markers in RViz).
6. Calculate the centroid (average in x, y and z) of the set of points belonging to that each object.
7. Create ROS messages containing the details of each object (name, pick_pose, etc.) and write these messages out to `.yaml` files, one for each of the 3 scenarios (`test1-3.world` in `/pr2_robot/worlds/`). See the example `output.yaml` for details on what the output should look like.
8. Submit a link to your GitHub repo for the project or the Python code for your perception pipeline and your output `.yaml` files (3 `.yaml` files, one for each test world). You must have correctly identified 100% of objects from `pick_list_1.yaml` for `test1.world` , 80% of items from `pick_list_2.yaml` for `test2.world` and 75% of items from `pick_list_3.yaml` in `test3.world` .
9. Congratulations! Your Done!

Extra Challenges: Complete the Pick & Place

7. To create a collision map, publish a point cloud to the `/pr2/3d_map/points` topic and make sure you change the `point_cloud_topic` to `/pr2/3d_map/points` in `sensors.yaml` in the `/pr2_robot/config/` directory. This topic is read by Moveit!, which uses this point cloud input to generate a collision map, allowing the robot to plan its trajectory. Keep in mind that later when you go to pick up an object, you must first remove it from this point cloud so it is removed from the collision map!
8. Rotate the robot to generate collision map of table sides. This can be accomplished by publishing joint angle value(in radians) to `/pr2/world_joint_controller/command`
9. Rotate the robot back to its original state.
10. Create a ROS Client for the “pick_place_routine” rosservice. In the required steps above, you already created the messages you need to use this service. Checkout the PickPlace.srv file to find out what arguments you must pass to this service.
11. If everything was done correctly, when you pass the appropriate messages to the `pick_place_routine` service, the selected arm will perform pick and place operation and display trajectory in the RViz window
12. Place all the objects from your pick list in their respective dropoff box and you have completed the challenge!
13. Looking for a bigger challenge? Load up the `challenge.world` scenario and see if you can get your perception

pipeline working there!

Rubric Points

Here I will consider the rubric points individually and describe how I addressed each point in my implementation.

Writeup / README

1. Provide a Writeup / README that includes all the rubric points and how you addressed each one. You can submit your writeup as markdown or pdf.

You're reading it!

Exercise 1, 2 and 3 pipeline implemented

1. Complete Exercise 1 steps. Pipeline for filtering and RANSAC plane fitting implemented.

See `project_template.py` for the implementation.

The original point cloud, after being converted to the format of the PCL library, goes through three steps of processing here:

1. The voxel grid filter is created from the point cloud to downsample the points to the specified leaf size of 0.01 (m).
2. The downsampled point cloud is then filtered with a pass through filter, removing every point with z coordinate higher than 1.1m or lower than 0.6m.
3. Lastly, a Segmenter object is created from the filtered point cloud. The segmenter would use the RANSAC method to extract inlier points that belong to the plane of the table, and the outliers would be the points belonging to the objects.

2. Complete Exercise 2 steps: Pipeline including clustering for segmentation implemented.

See `project_template.py` for the implementation.

Based on the positions of the points in the object cloud, a Euclidean clustering is performed with a distance threshold of 0.02. Each cluster can be visualized using the generated color list to show different clusters.

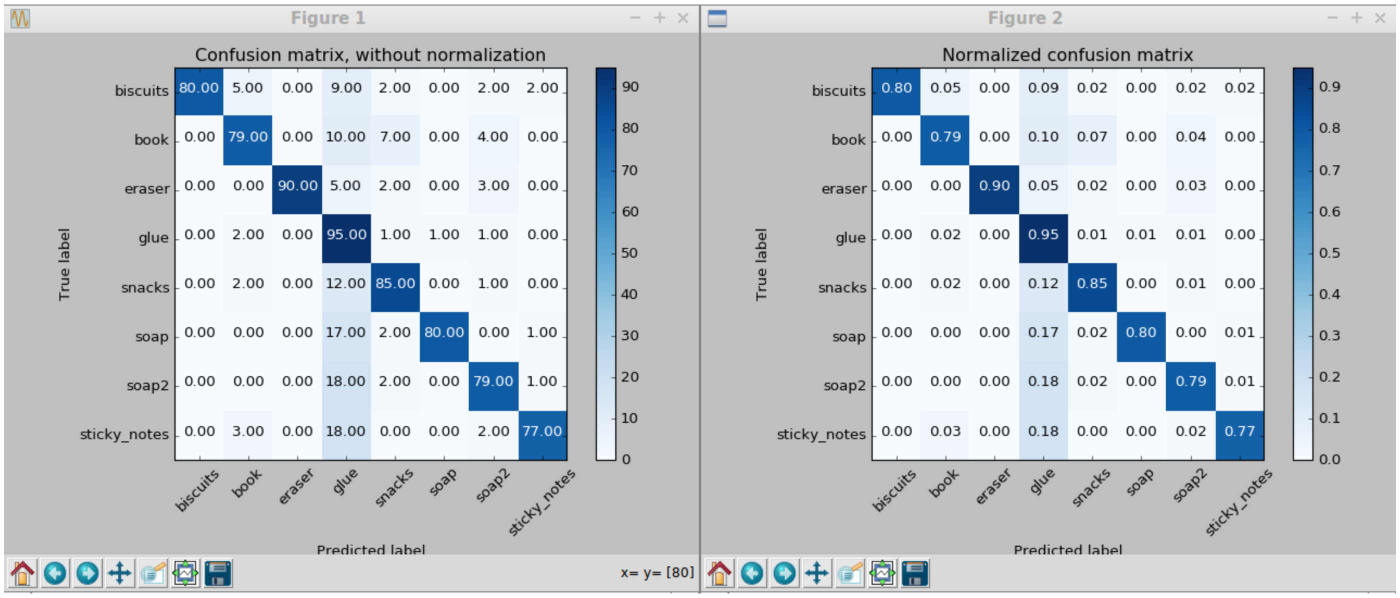
Three point clouds, including the table cloud, the object cloud and the clustered cloud, are then publised as ROS messages after being converted to the ROS formats.

3. Complete Exercise 3 Steps. Features extracted and SVM trained. Object recognition implemented.

See `sensor_stick_copy/features.py` , `capture_features.py` and `train_svm.py` for the implementation.

The features of each point in the point cloud include the color and the normal histograms of the cloud created from `numpy.histogram` . The RGB color values range from 0 to 256, and are binned into 32 bins. The XYZ normal values range from -1 to 1, and are also binned into 32 bins. The color and normal features are then separately normalized into a total sum of 1.0 for each cloud.

During the training phase, each model is created with 100 random orientations, and the resulting point clouds' features are extracted and saved (not included in the repo due to size). The features are then used to train an SVM model with sigmoid kernel. I found the sigmoid kernel to give the best results compared to linear, poly and rbf kernels from my dataset. The model is saved in `results/model.sav` . For the pick and place project, all the models in world 3 are used to collect the features and train the classifier, so the model is an 8-class SVM classifier.



Confusion matrix, without normalization

	biscuits	book	eraser	glue	snacks	soap	soap2	sticky_notes
biscuits	80.00	5.00	0.00	9.00	2.00	0.00	2.00	2.00
book	0.00	79.00	0.00	10.00	7.00	0.00	4.00	0.00
eraser	0.00	0.00	90.00	5.00	2.00	0.00	3.00	0.00
glue	0.00	2.00	0.00	95.00	1.00	1.00	1.00	0.00
snacks	0.00	2.00	0.00	12.00	85.00	0.00	1.00	0.00
soap	0.00	0.00	0.00	17.00	2.00	80.00	0.00	1.00
soap2	0.00	0.00	0.00	18.00	2.00	0.00	79.00	1.00
sticky_notes	0.00	3.00	0.00	18.00	0.00	0.00	2.00	77.00

Normalized confusion matrix

	biscuits	book	eraser	glue	snacks	soap	soap2	sticky_notes
biscuits	0.80	0.05	0.00	0.09	0.02	0.00	0.02	0.02
book	0.00	0.79	0.00	0.10	0.07	0.00	0.04	0.00
eraser	0.00	0.00	0.90	0.05	0.02	0.00	0.03	0.00
glue	0.00	0.02	0.00	0.95	0.01	0.01	0.01	0.00
snacks	0.00	0.02	0.00	0.12	0.85	0.00	0.01	0.00
soap	0.00	0.00	0.00	0.17	0.02	0.80	0.00	0.01
soap2	0.00	0.00	0.00	0.18	0.02	0.00	0.79	0.01
sticky_notes	0.00	0.03	0.00	0.18	0.00	0.00	0.02	0.77

Pick and Place Setup

1. For all three tabletop setups (`test*.world`), perform object recognition, then read in respective pick list (`pick_list_*.yaml`). Next construct the messages that would comprise a valid `PickPlace` request output them to `.yaml` format.

- In `pcl_callback` :

For each camera frame, the point cloud is processed, filtered, and clustered. The features of the clusters are then used by the saved 8-class classifier to classify the cluster into one of the eight classes. The label and the cluster cloud is then saved into a `DetectedObject` message type.

- In `pr2_mover` :

The list of `DetectedObject` provided from `pcl_callback` is compared against the pick list to identify the pick poses of each items in the pick list. From the parameter server, I acquire the `object_list` parameters, which contain the items to be picked and the arm/group to pick the item, and the `dropbox` parameters, which contain the group identifier of each arm (`red` for `left` and `green` for `right`) and the final place position of each arm. I converted the `dropbox` parameters into dictionaries of group names to arm names and place poses for further uses.

To process the `DetectedObject` list, I created three lists of the same size as the number of cluster, including the labels and centroids of each detected object. The last list is a list of booleans that indicate whether the entry of `DetectedObject` has been identified as an item to pick.

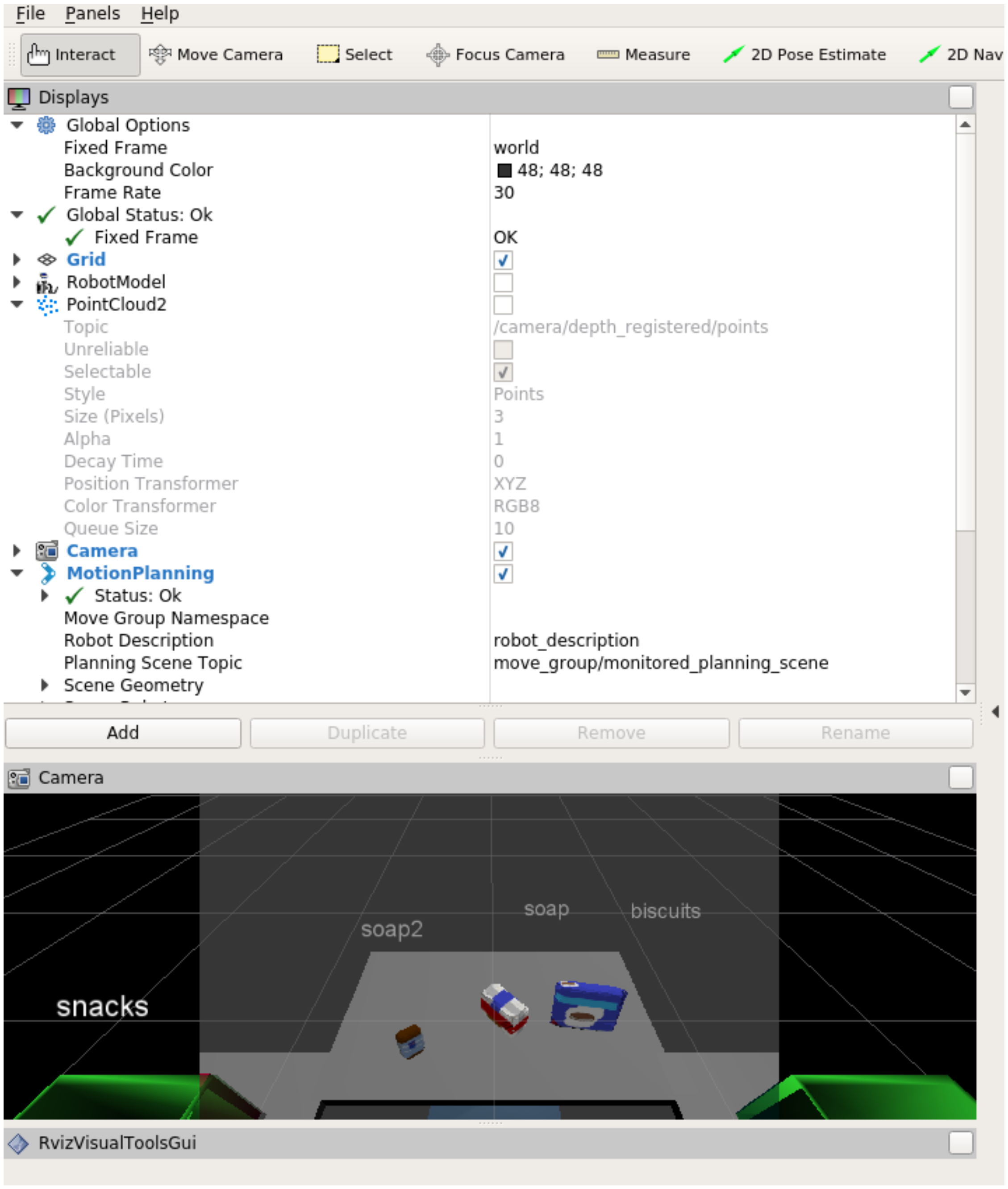
To compare the `DetectedObject` list and the items to pick list from the `object_list` parameter, I iterate over the items to pick, and compare the labels with the labels in the `DetectedObject` list. If a detected object has the same label and has not been used for a previous item, the detected object is identified as the next item to pick. The pick pose would simply be the centroid of the `DetectedObject` , and the place pose would be based on the group name from the `object_list` . The parameters are then wrapped into a dictionary and saved into a list of yaml dictionaries. At the end of the function, the yaml dictionaries are saved as the output yaml file if the number of identified items to pick is sufficient.

Result Discussion

For the exercises, I took 100 random orientations of each model and used a sigmoid kernel SVM to classify the training set. The cross validation resulted in a final score of 0.8375. Most of the errors are false negatives of `glue` classifications. Using other kernsl result in lower score, mostly below 0.8.

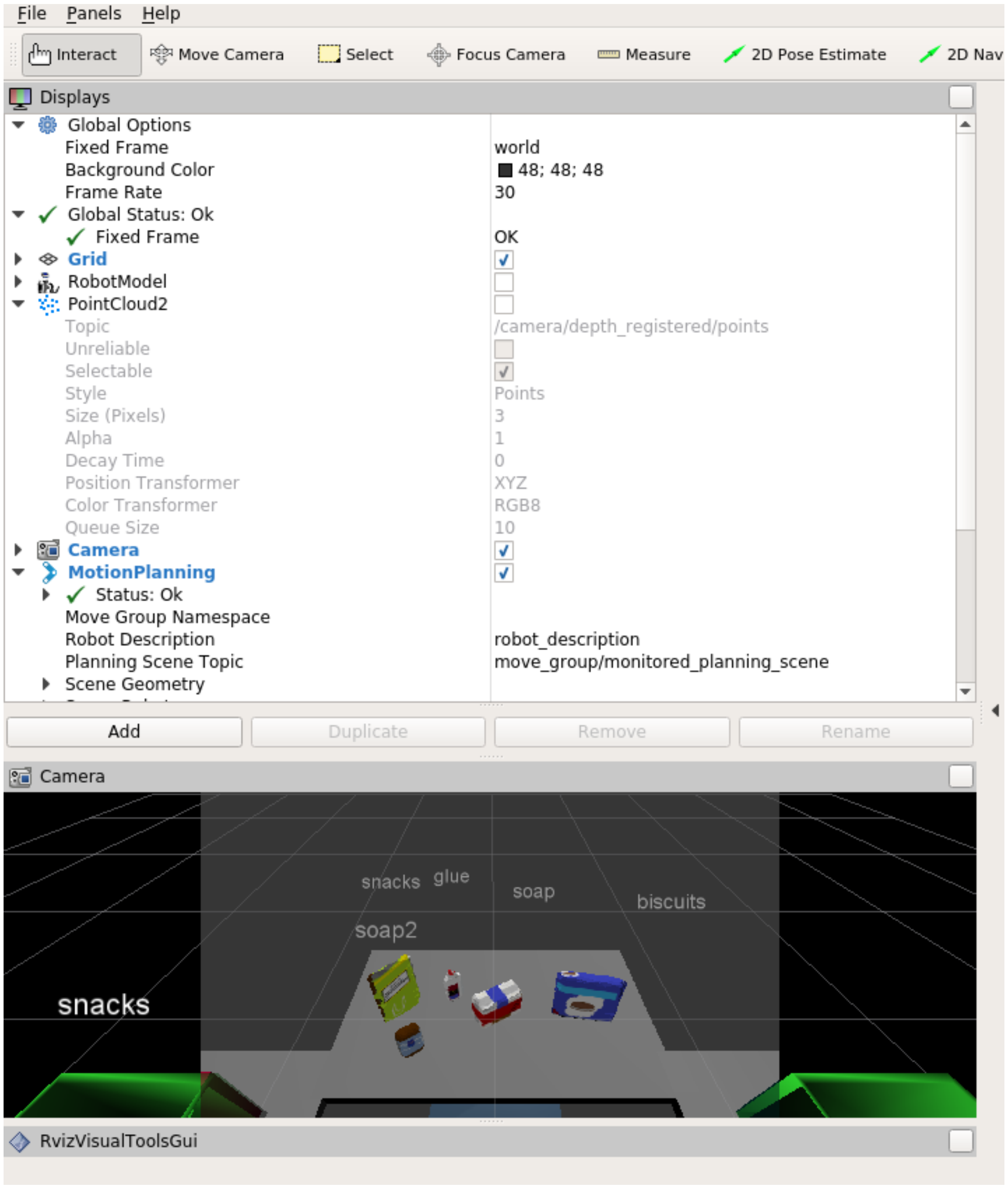
The model performed fairly acceptable in the pick and place environment. For world 1, the model can correctly classify all three objects most of the time. However, for world 2 and 3, the model usually gets 3 or 5 objects correct and sometimes gets more than 4 or 6 objects correct. Notable in world 3, the book and the glue are usually clustered together and thus making neither of them correctly identified.

World 1 results:



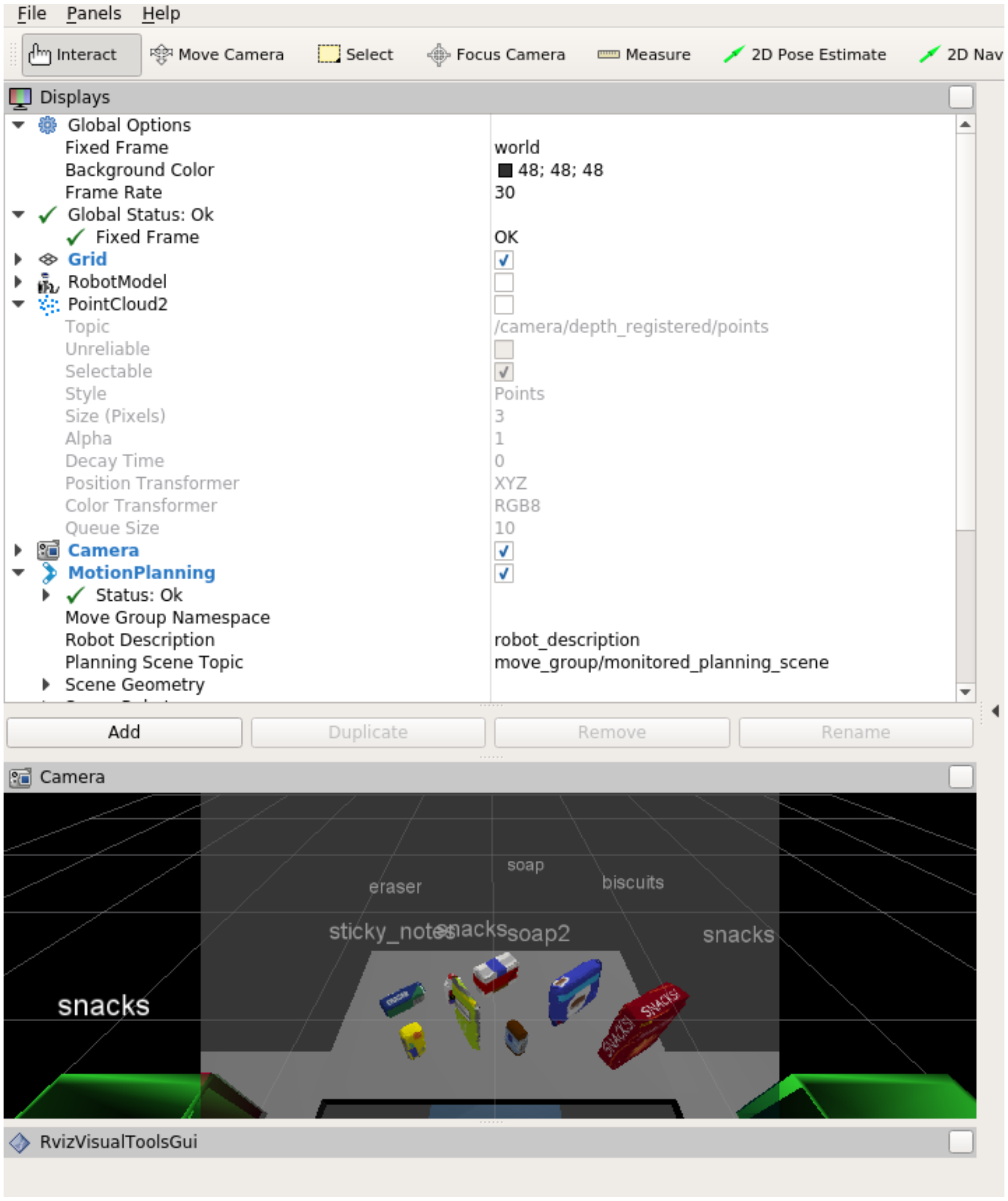
```
Global Options
  Fixed Frame                 world
  Background Color            48; 48; 48
  Frame Rate                  30
Global Status: Ok
  Fixed Frame                 OK
Grid                          ✓
RobotModel
PointCloud2
  Topic                       /camera/depth_registered/points
  Unreliable
  Selectable                  ✓
  Style                       Points
  Size (Pixels)               3
  Alpha                       1
  Decay Time                  0
  Position Transformer        XYZ
  Color Transformer           RGB8
  Queue Size                  10
Camera                        ✓
MotionPlanning                ✓
  Status: Ok
  Move Group Namespace
  Robot Description           robot_description
  Planning Scene Topic        move_group/monitored_planning_scene
  Scene Geometry
```

World 2 results:



Global Options	
Fixed Frame	world
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	☐
PointCloud2	☐
Topic	/camera/depth_registered/points
Unreliable	☐
Selectable	✓
Style	Points
Size (Pixels)	3
Alpha	1
Decay Time	0
Position Transformer	XYZ
Color Transformer	RGB8
Queue Size	10
Camera	✓
MotionPlanning	✓
✓ Status: Ok	
Move Group Namespace	
Robot Description	robot_description
Planning Scene Topic	move_group/monitored_planning_scene
Scene Geometry	

World 3 results:



Global Options	
Fixed Frame	world
Background Color	■ 48; 48; 48
Frame Rate	30
✓ Global Status: Ok	
✓ Fixed Frame	OK
Grid	✓
RobotModel	☐
PointCloud2	☐
Topic	/camera/depth_registered/points
Unreliable	☐
Selectable	✓
Style	Points
Size (Pixels)	3
Alpha	1
Decay Time	0
Position Transformer	XYZ
Color Transformer	RGB8
Queue Size	10
Camera	✓
MotionPlanning	✓
✓ Status: Ok	
Move Group Namespace	
Robot Description	robot_description
Planning Scene Topic	move_group/monitored_planning_scene
Scene Geometry	

I think the model can be improved with a larger training set (capturing 800 took a while on my computer so I haven't tried it yet), and better tuning of the classified. I was trying to make the results consistent, so I did not train separate classifiers for each world, but the performance can definitely be better if we fine-tune the classifier for each world.