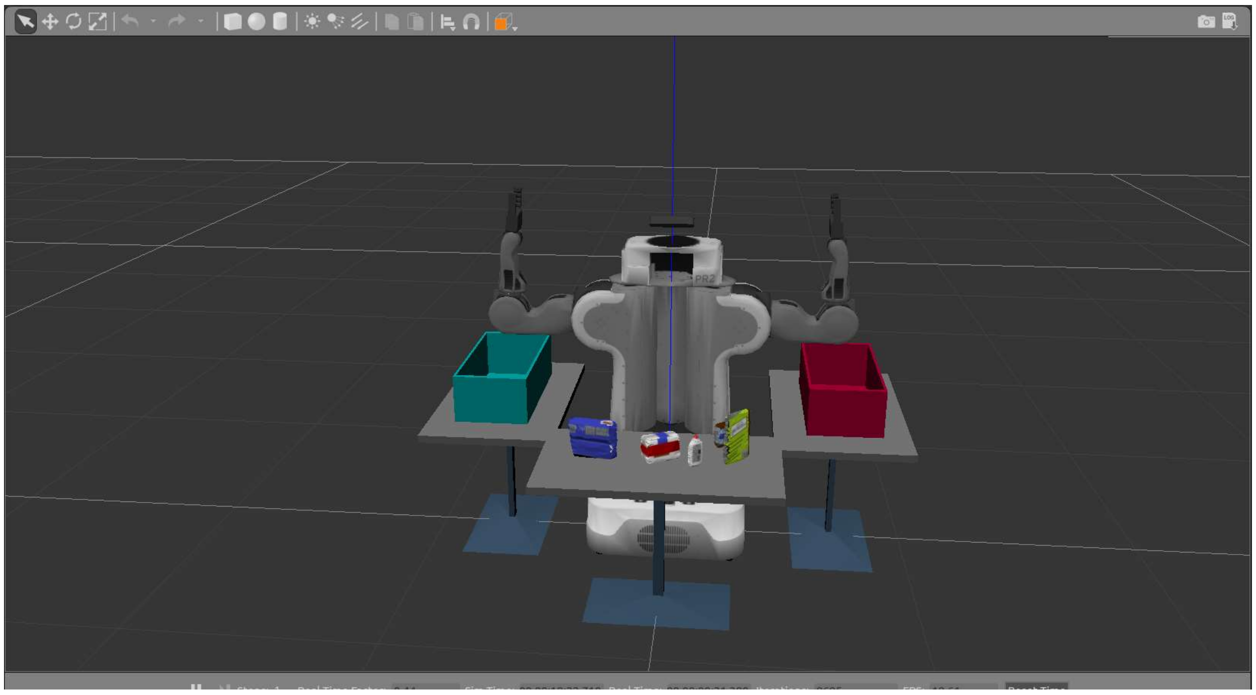
3D Perception

1 OVERVIEW

Our primary goal in this project was to identify specific objects in the PR2 robots field of view and generate "pick and place" commands to retrieve them in the form of yaml files. We were given the secondary challenges of recognizing all objects in a single "megaworld", and of generating the collision matrices and related pick place commands for the actual PR2 robots pick and place operations. (This is contrasted to simply identifying where the objects where and the pick / place poses associated with those locations).

This was achieved by implementing a perception pipeline. Our perception pipeline can be split into three basic parts (which will be described in detail below): Filtering, Segmentation, and Recognition. The results of this pipeline are then sent to a movement pipeline which translates the results into pick and place poses for the PR2 robot which it send back to the robot as an ROS message. It also records the pick and pose commands as yaml files for further analysis.



"The PR2 Robot"

2 PERCEPTION PIPELINE

As mentioned in the overview, the perception pipeline is split into three main parts. In part 1, we look at filtering the data from the PR2's RGBD camera. This step is done to reduce unwanted noise, remove irrelevant information, and sub sample the input to a more manageable size (thus reducing computation time). Once the data is filtered, it is then passed through the segmentation step, which takes the resulting point cloud, removes those points associated with the table, and splits the remaining points into clusters using a Euclidean clustering method. Finally, these clusters are pushed into a pre-trained classification algorithm that provides the label for each cluster with varying degrees of accuracy. The details of these three steps are outlined below.

2.1 FILTERING

2.1.1 Noise Reduction

The input provided to the RGBD camera on the PR2 via the simulation is contaminated with odd pixels of noise that distort the image. Our first task in working with the camera's input is to clean up the noise as best as possible. The method chosen for this task was to use a statistical outlier filter as provided in the course materials. Essentially this filter looks at the mean distance of each point from its neighboring points and culls any point which is more than a single standard deviation (times a scale factor, but the scale factor has been left as 1.0 in this implementation) away from its neighboring points. This removes the few stray points that may have caused confusion for any steps further down the pipeline.

```
#clean up noise in the camera data
def statistical_outlier_filter(point_cloud):
    # Much like the previous filters, we start by creating a filter object:
    outlier_filter = point_cloud.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()

    return cloud_filtered
```

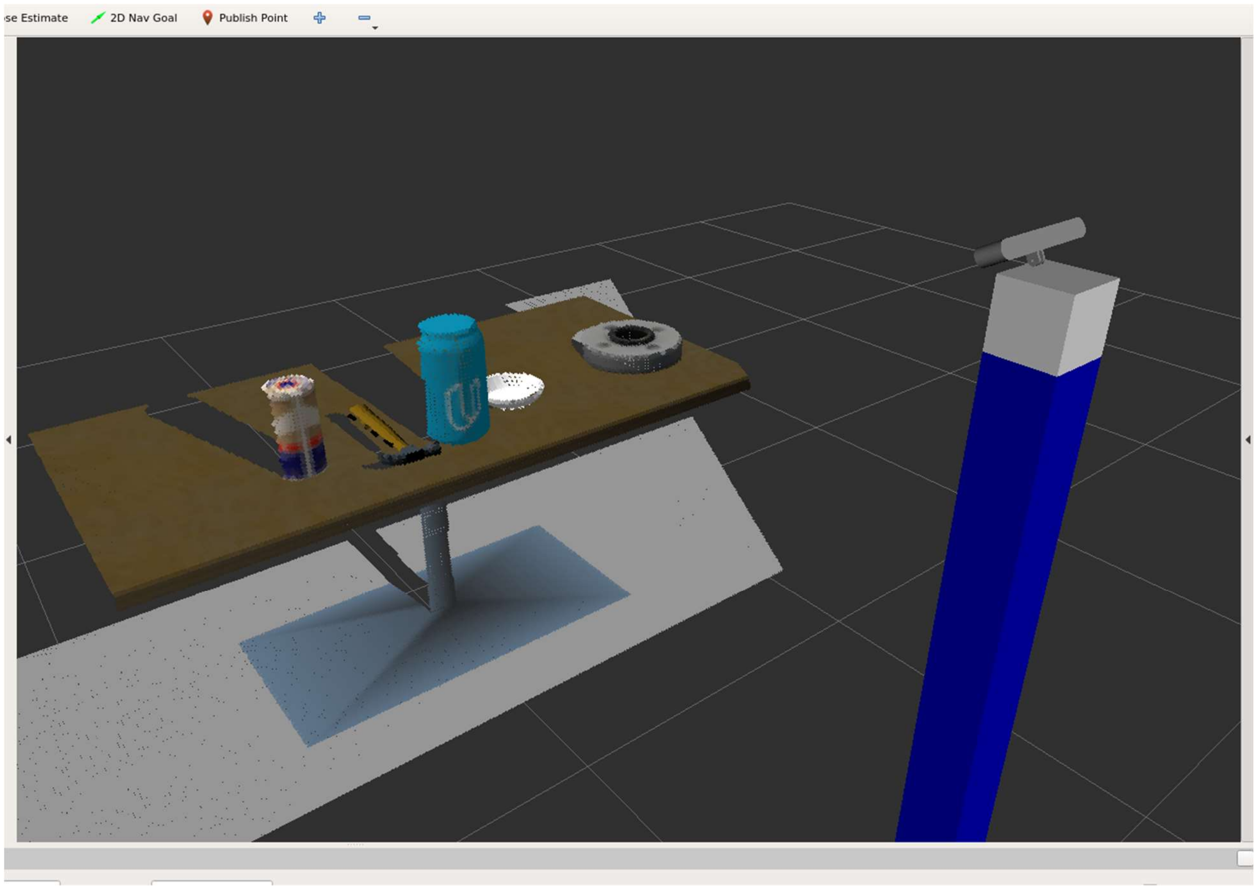
2.1.2 Down sampling

With the noise filtered out (at least to a reasonable level), the next step in the pipeline is Voxel Down sampling. The number of pixels provided by the RGBD camera is extremely large, and processing them all is not only not required to get good results, but computationally expensive. To alleviate this issue, the points are down sampled into voxels.

Voxels can be thought of as three-dimensional pixels – pixels with volume if you will. Down sampling is achieved by first subdividing the space the voxels of a given size (provided as the LEAF_SIZE parameter, for which this implementation uses a value of 0.01) and assigns a value to the voxel based on the average value of all the pixels that would fall inside of its volume.

This filter is implemented using the provided pcl library.

```
#downsample the point cloud into voxels to save processing time.
def voxel_grid_downsample(pointCloud):
    # voxel size
    LEAF_SIZE = 0.01

    # configure filter
    vox = pointCloud.make_voxel_grid_filter()
    vox.set_leaf_size(LEAF_SIZE, LEAF_SIZE, LEAF_SIZE)

    return vox.filter()
```

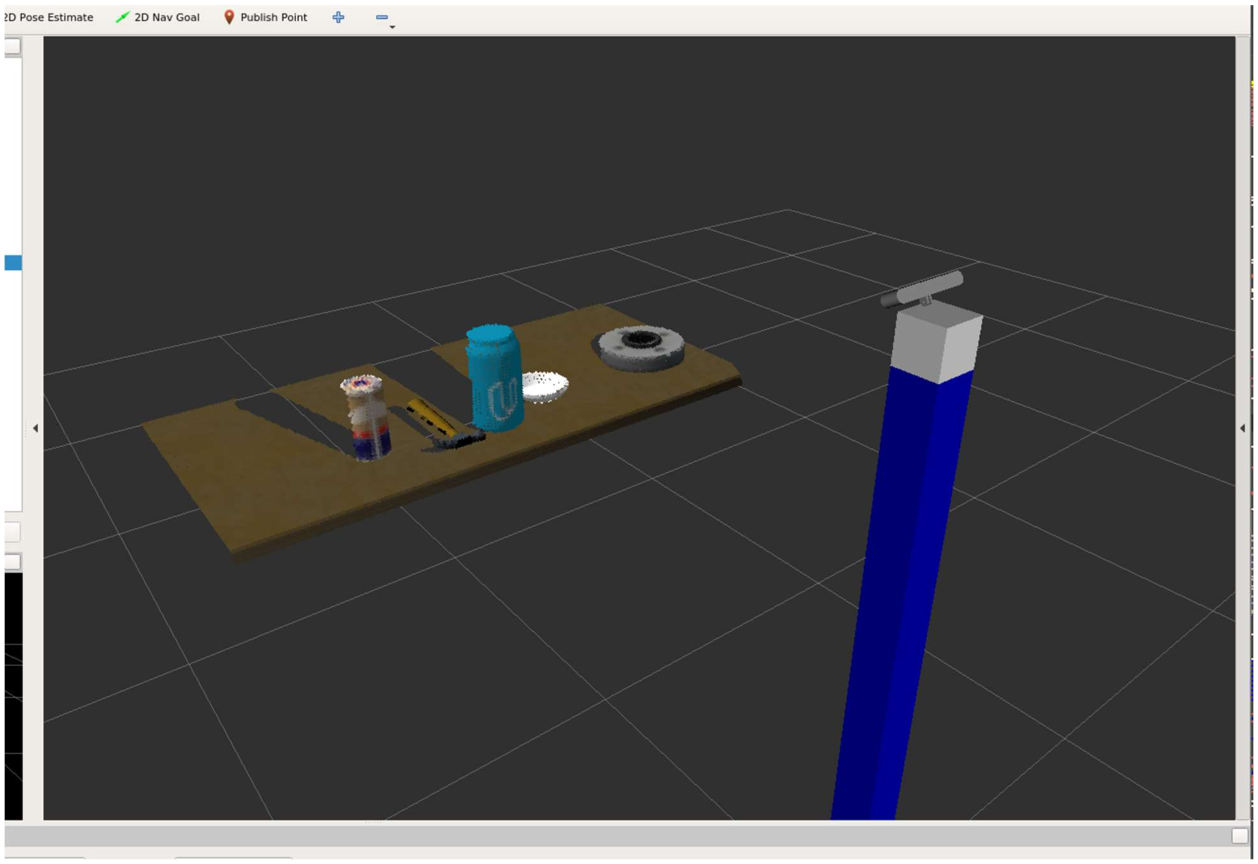


"Voxel Down Sampled Image"

2.1.3 Passthrough Filter

With the input size reduced to a manageable level, the next step in the pipeline removes any extraneous data from the image. For the task we are given, the PR2 robot is only concerned with the actual table top, and the objects on it. Everything below the table top can be discarded, which is achieved using a passthrough filter. This filter works by only allowing those voxels whose z-axis values lay between 0.6 and 1.1 though the filter, suppressing any voxels outside this range. The end effect of this is that only the voxels associated with the items on the table, and the table itself, remain in the point cloud after the filter has done its work.

```
#filter out parts of the image outside the table.
def passthrough_filter(voxels):
    # pass through settings
    filter_axis = 'z'
    axis_min = 0.6
    axis_max = 1.1

    # configure filter
    passthrough = voxels.make_passthrough_filter()
    passthrough.set_filter_field_name(filter_axis)
    passthrough.set_filter_limits(axis_min, axis_max)

    return passthrough.filter()
```



“Passthough filter results”

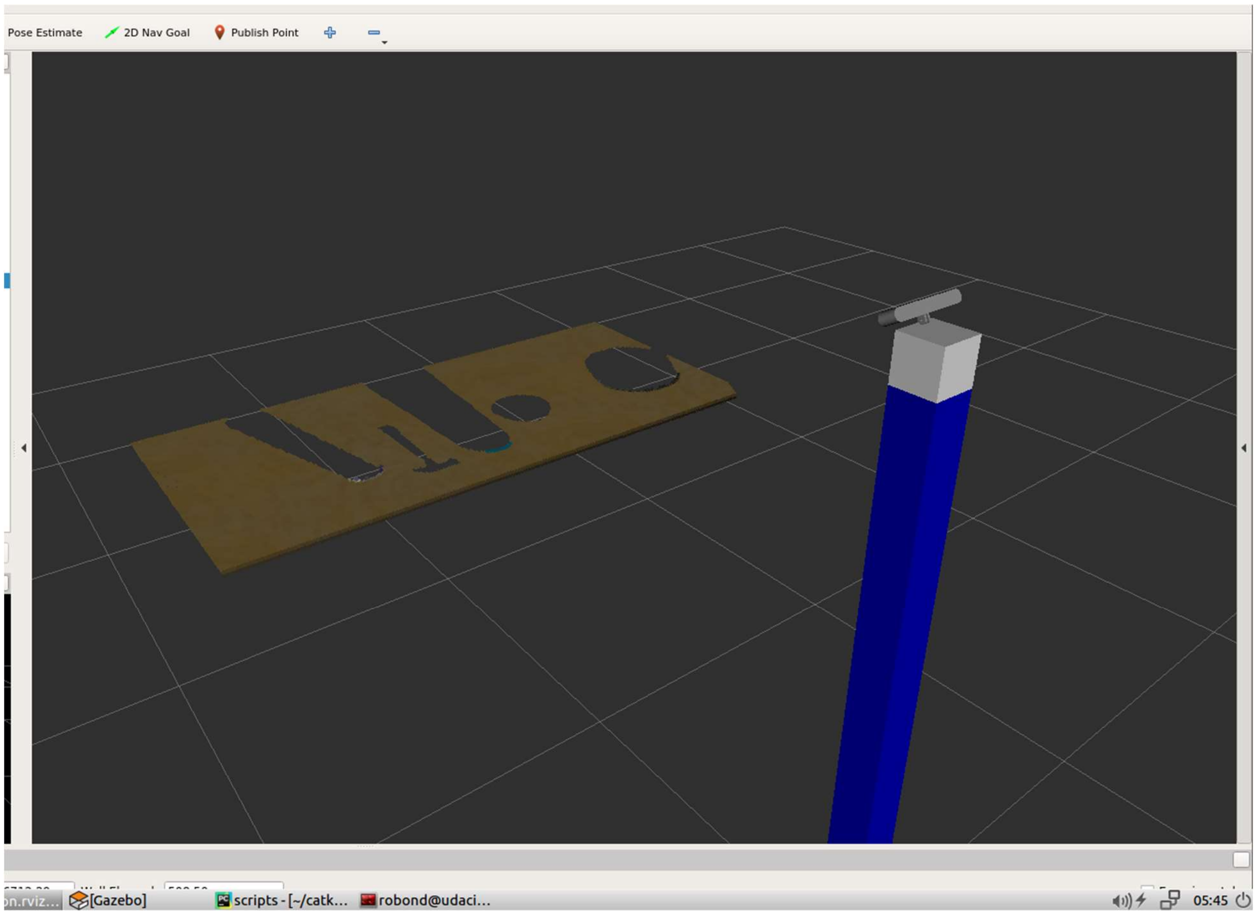
2.2 Clustering and Segmentation

2.2.1 RANSAC

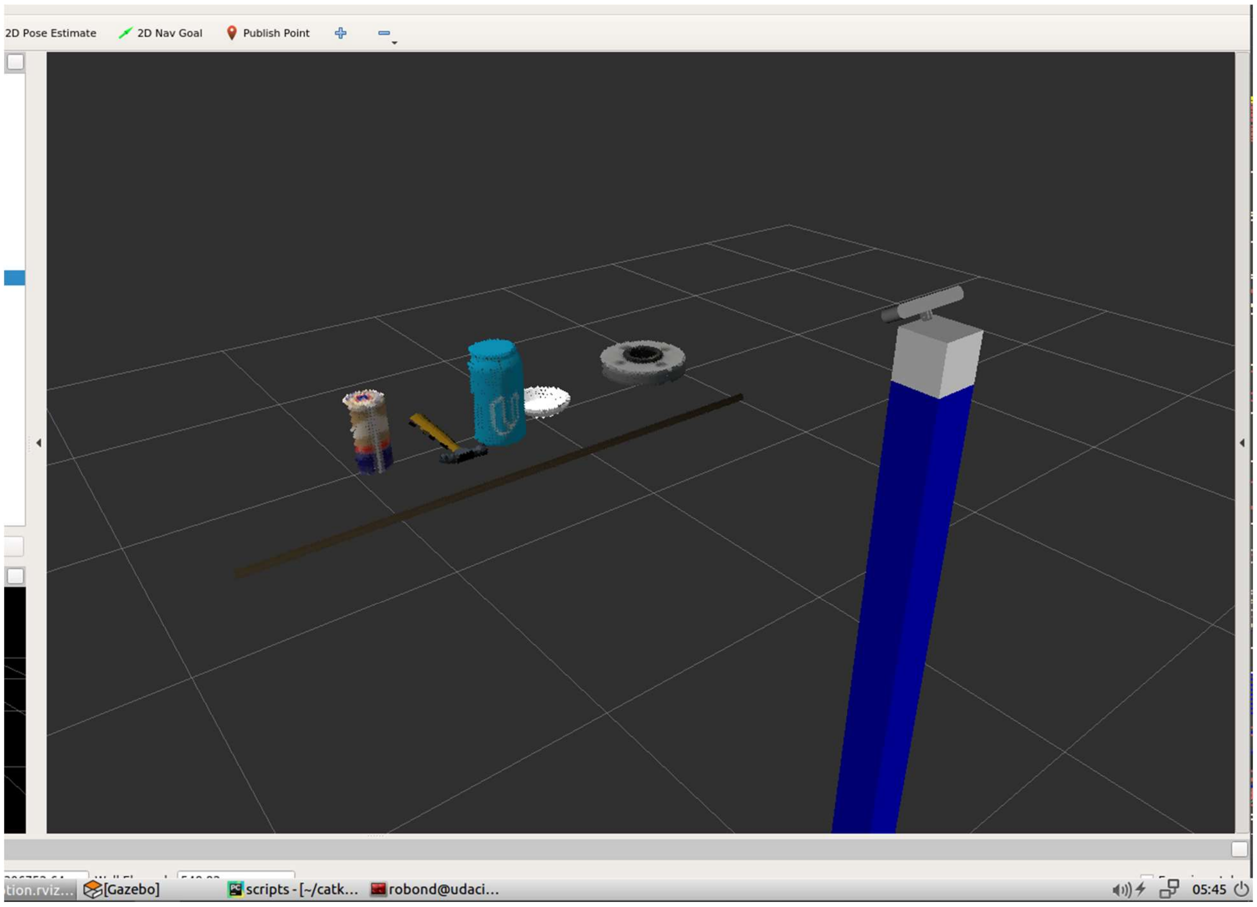
Our first step in the clustering process is to separate the table voxels from the object voxels. To do this we will use a clustering method known as RANSAC (Random Sample Consensus). The idea here is to use the RANSAC algorithm to fit the table voxels to a plane. Once done, we can split the data into two sets – those voxels which are part of the “plane” and thus can be considered part of the table, and everything else – assumed to be the objects of interest.

Images illustrating the results of this step are included below.

```
#segement the point cloud data using ransac
def ransac_segmentation(voxels):
    max_distance = 0.035
    seg = voxels.make_segmenter()

    # Set the model you wish to fit
    seg.set_model_type(pcl.SACMODEL_PLANE)
    seg.set_method_type(pcl.SAC_RANSAC)
    seg.set_distance_threshold(max_distance)

    return seg.segment()
```



"Table voxels extracted via RANSAC"



"Object Voxels remaining after RANSAC"

2.2.2 Euclidean Clustering

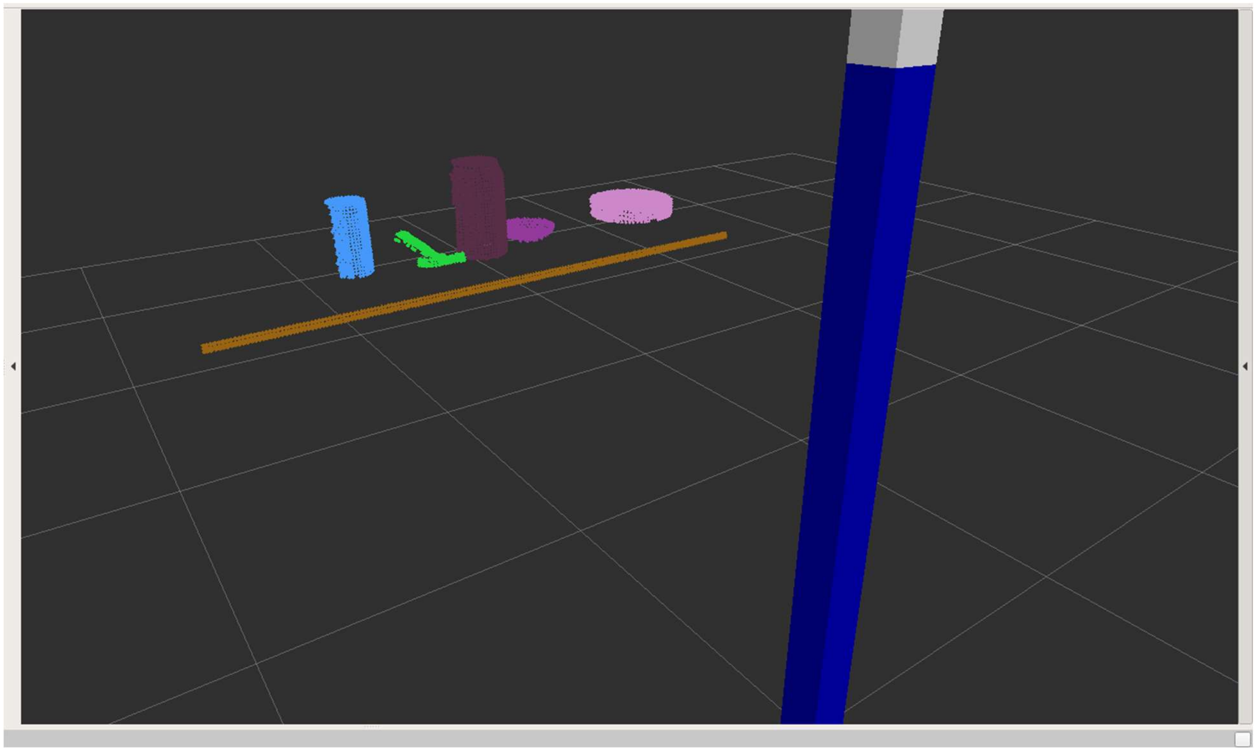
Now that we have separated out the object voxels from the table, we can proceed to separate the individual objects from each other in preparation for the object recognition step. This will be achieved using the Euclidean Clustering algorithm provided by the pcl library.

To perform the clustering step, we first need to convert the point clouds from the RGBXYZ format into an XYZ format. This strips all the color information from the points to make the input compatible with the make_EuclideanClusterExtration() method.

We provide several parameters to the algorithm, which were found via trial and error in the simulation. These are the cluster tolerance (or allowed distance between points in a cluster), and the minimum / maximum cluster sizes. These have been set to 0.05, 50, and 2000 respectively and seem provide a reasonable clustering solution.

A color-coded example of the clustering steps output is available in the image attached below.

```
#extract object clusters from a white point cloud
def extract_clusters(white_cloud):


    # create kd-tree
    tree = white_cloud.make_kdtree()

    # Create a cluster extraction object
    ec = white_cloud.make_EuclideanClusterExtraction()

    # Set tolerances for distance threshold
    # as well as minimum and maximum cluster size (in points)
    ec.set_ClusterTolerance(0.05)
    ec.set_MinClusterSize(50)
    ec.set_MaxClusterSize(2000)

    # Search the k-d tree for clusters
    ec.set_SearchMethod(tree)

    # Extract indices for each of the discovered clusters
    cluster_indices = ec.Extract()

    return cluster_indices
```



"Object Segmentation"

2.3 Object Recognition

The object recognition phase of the perception pipeline was built in two steps. Step one was to collect features representing each of the objects we might see in the scene. Step two was to train a classifier (in the case of the exercises used as the examples in this section a Support Vector Machine (SVM) was used) to recognize the objects from their cluster data.

2.3.1 Feature Collection

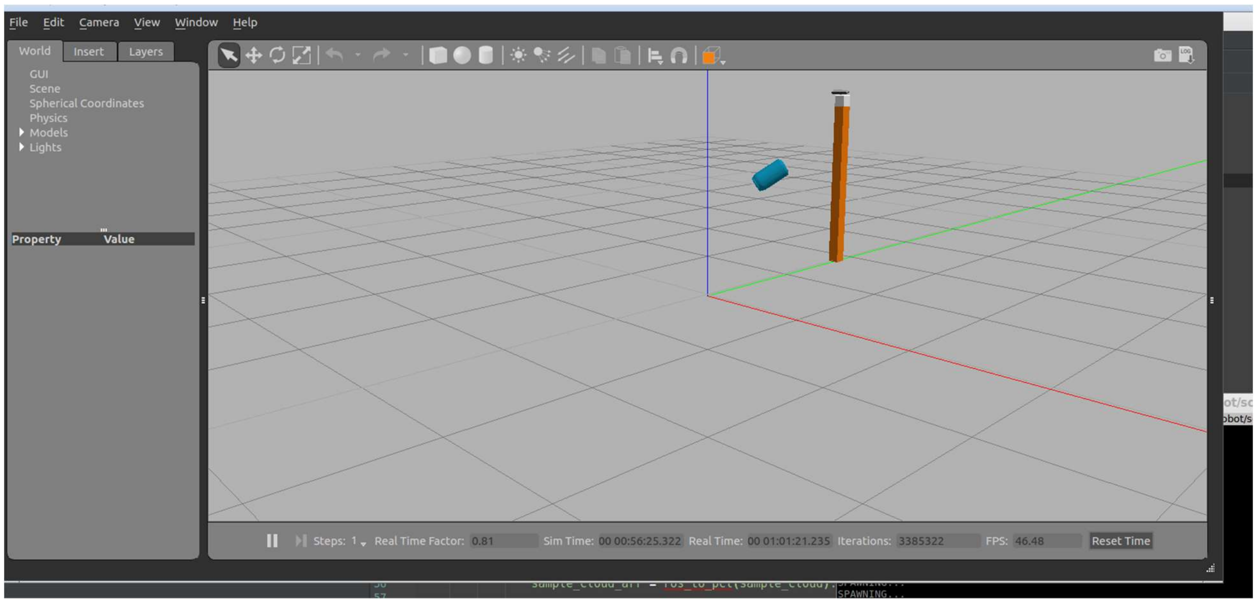
As we are using a supervised learning algorithm to perform object recognition, we require labeled examples of the objects we wish to find to train the system. This achieved by making use of the sensor_stick robots training configuration (an example of which is shown in the associated image for this section).

Features were built by systematically loading each model into 100 random orientations. These were then scanned by the sensor sticks RGBD camera and converted to a point cloud. Once the point cloud was obtained, we created two histograms. The first histogram collected all the surface normal and binned them into 1024 bins (seen in the code snippet provided). Next the RBG values where transformed into the HSV colour space (done to reduce light dependencies, and because the system seemed to perform better in HSV space than RGB space) which was then converted to a histogram with 64 bins. These histograms where then normalized and concatenated together to create a feature vector.

Once a feature vector had been generated for each of the 100 random position on all the objects in the scene, the resulting data was saved in a training_data.sav file so it could be used to train the SVM classifier in the next step.

```python
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])
    # TODO: Compute histograms
    ch_1_hist = np.histogram(channel_1_vals, bins=64, range=(0, 256))
    ch_2_hist = np.histogram(channel_2_vals, bins=64, range=(0, 256))
    ch_3_hist = np.histogram(channel_3_vals, bins=64, range=(0, 256))

    # TODO: Concatenate and normalize the histograms
    feature_vector = np.concatenate( (ch_1_hist[0], ch_2_hist[0], ch_3_hist[0])
).astype(np.float64)
    normalized_feature_vector = feature_vector / np.sum(feature_vector)

    # Generate random features for demo mode.
    # Replace normed_features with your feature vector
    normed_features = normalized_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)
    norm_x_hist = np.histogram(norm_x_vals, bins=1024, range=(0,1))
    norm_y_hist = np.histogram(norm_y_vals, bins=1024, range=(0,1))
    norm_z_hist = np.histogram(norm_z_vals, bins=1024, range=(0,1))

    # TODO: Concatenate and normalize the histograms
    feature_vector = np.concatenate( (norm_x_hist[0], norm_y_hist[0],
norm_z_hist[0])).astype(np.float64)
    normed_features = feature_vector / np.sum(feature_vector)

    # Generate random features for demo mode.
    # Replace normed_features with your feature vector
    #normed_features = np.random.random(96)

    return normed_features
```



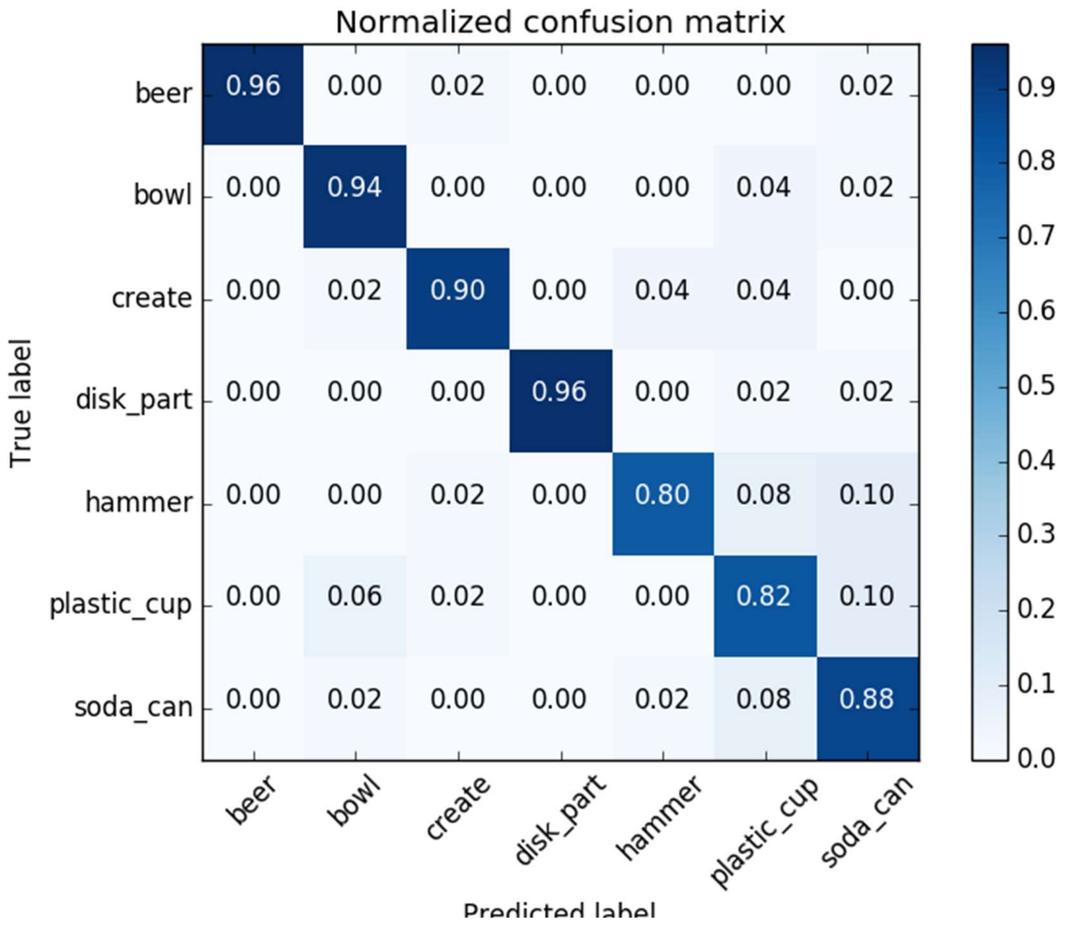
"Sensor Stick capturing Soda Can features"

2.3.2 Training the Classifier

Using the features collected in the previous section, we now create a model using a SVM. The SVM was obtained from the sklearn library. Parameters for the SVM where chosen via the GridSearchCV api, searching though the following parameters:

Parameter	**Values**
Kernel	Linear, rbf, poly
C	0.01, 0.1, 1, 10, 100, 1000
gamma	0.01, 0.1, 1, 10, 100, 1000

Once the SVM is fit against the collected features, an accuracy score and confusion matrix where generated. The confusion matrix resulting from training the SVM for the objects in the provided exercises is provided below



"Confusion Matrix for SVM Classifier"

3 Pick and Place

We make use of the perception pipeline described in the previous section to control the PR2 robot simulation to do a pick and place operation. To do this, we must add some additional support code to the perception pipeline. A basic outline of the system is below, details of each step will be provided in the following subsections.

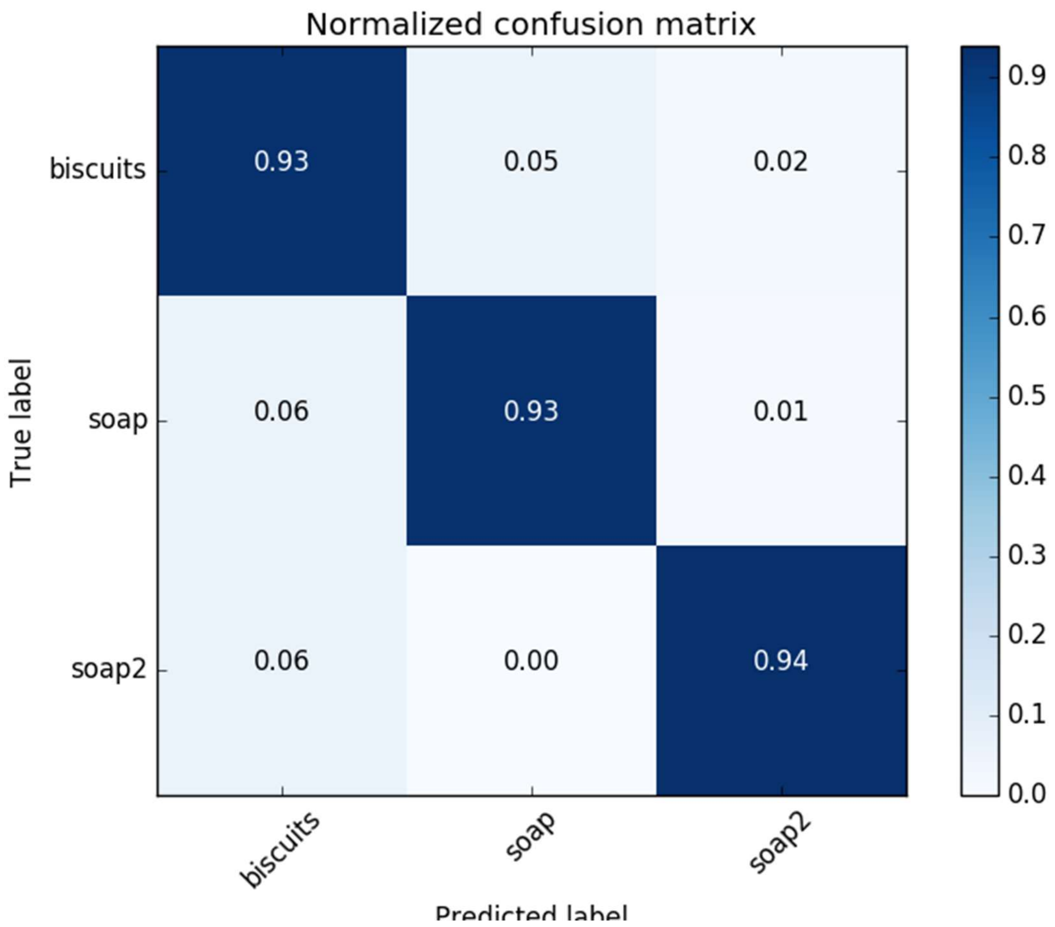
1. Build a collision map
2. Process the camera data through the perception pipeline.
3. Calculate the centroids of all detected objects.
4. Get the pick list from the parameter server.
5. For each object in the pick list
    a. Determine the correct arm.
    b. Build a pick pose for the object
    c. Build a place pose for the object
    d. Send pick place command

6. Save the pick and place commands in a yaml file.

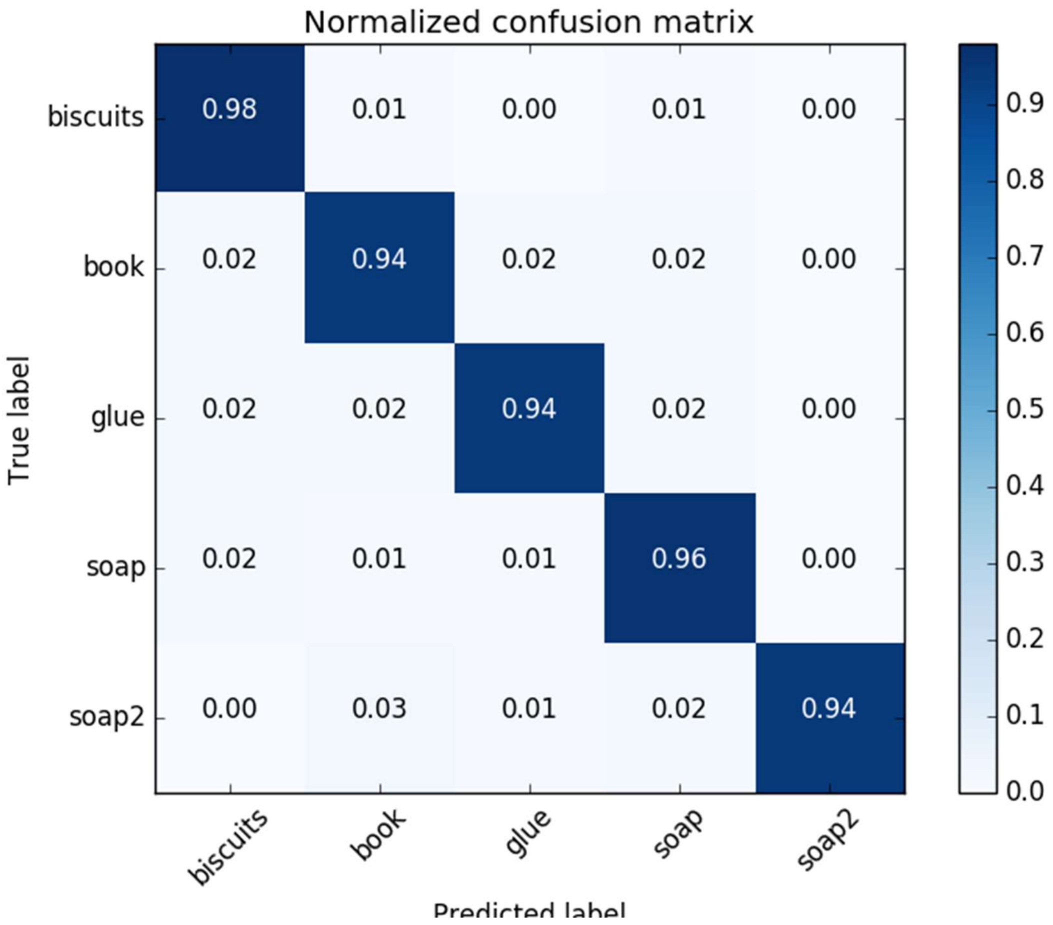
3.1 Build Model

To make use of the perception pipeline, it was necessary to build a model for each of the three worlds we were required to work with. Training of these models was done in much same way as was done for the perception pipeline exercises – however one key change was introduced.

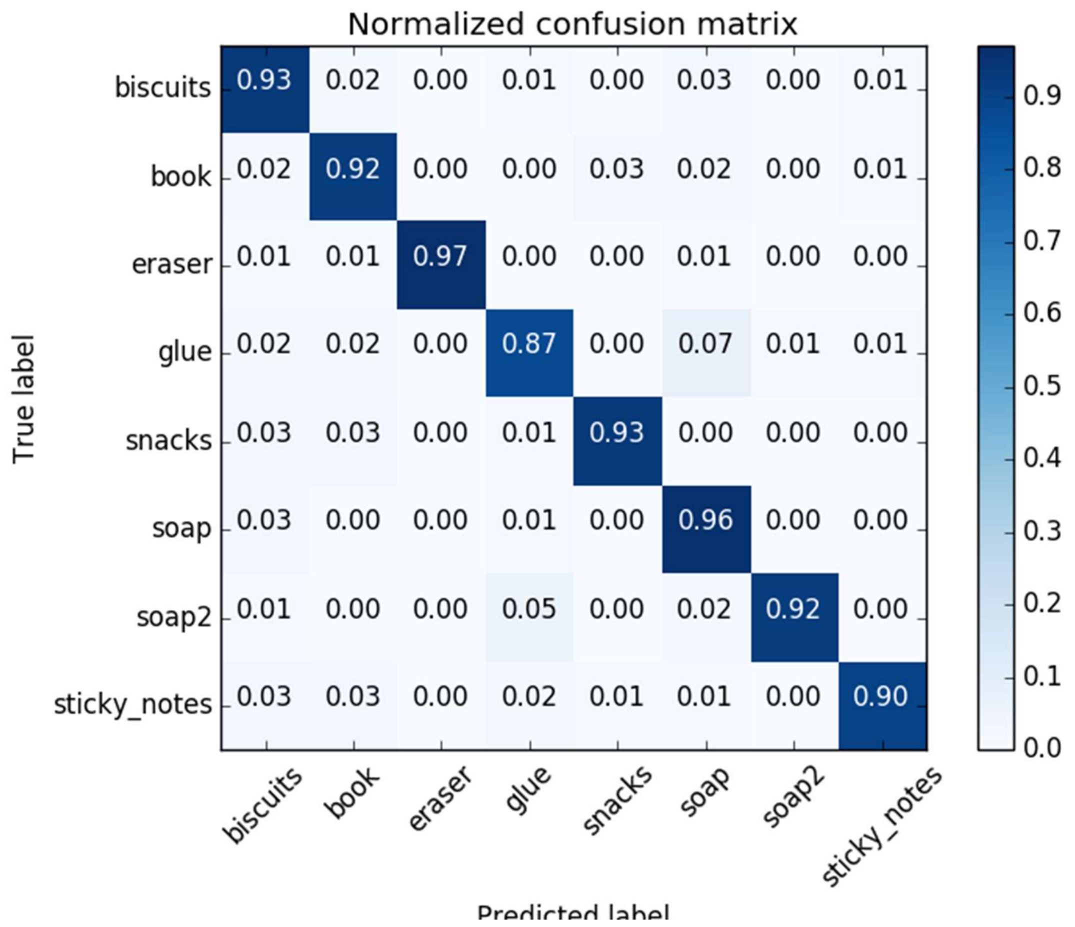
Rather than use a support vector machine, it was decided to change to a Logistic Regression algorithm (obtained from the SKLEARN library). This proved to be easier to train and seemed to get slightly better results than the SVM approach used in the exercises. Confusion matrices from the training data collected are provided below:



"World 1 Confusion Matrix"



"World 2 Confusion Matrix"



	biscuits	book	eraser	glue	snacks	soap	soap2	sticky_notes
biscuits	0.93	0.02	0.00	0.01	0.00	0.03	0.00	0.01
book	0.02	0.92	0.00	0.00	0.03	0.02	0.00	0.01
eraser	0.01	0.01	0.97	0.00	0.00	0.01	0.00	0.00
glue	0.02	0.02	0.00	0.87	0.00	0.07	0.01	0.01
snacks	0.03	0.03	0.00	0.01	0.93	0.00	0.00	0.00
soap	0.03	0.00	0.00	0.01	0.00	0.96	0.00	0.00
soap2	0.01	0.00	0.00	0.05	0.00	0.02	0.92	0.00
sticky_notes	0.03	0.03	0.00	0.02	0.01	0.01	0.00	0.90

"World 3 Confusion Matrix"

3.2 Build Collision Map

Building the collision map requires we rotate the PR2 to get a good look at the tables on the left and right with the drop off boxes. As we do not want the main perception pipeline to run while the robot is moving (otherwise the object recognition would fail and send nonsense pick and place commands), the movement system is controlled by a state machine which controls which behavior should be active at any given time.

The following states were used to build the initial collision map:

State	Action
LEFT	PR2 Robot rotates world joint to the pi / 2 position
READ_LEFT	PR2 Robot merges the currently detected point cloud into the collision map
RIGHT	PR2 Robot rotates world joint to the -pi / 2 position.
READ_RIGHT	PR2 Robot merges the currently detected point cloud into the collision map

CENTER	PR2 Robot rotates to the zero position.
PERCEPTION	PR2 Robot commences pick and place operations

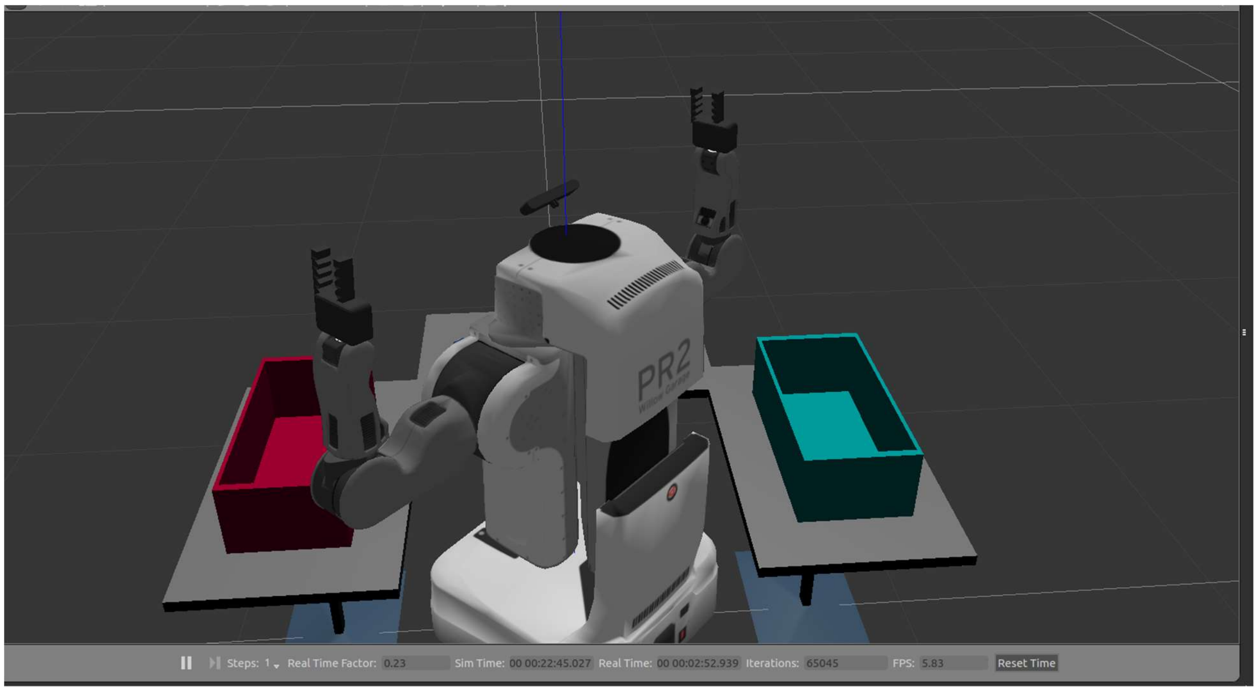
When the perception node is started up, the PR2 begins in the "LEFT" state. As it completes each action, the current state is transitioned down the list until it reaches the perception state. Once the PR2 reaches the perception state, it remains in that state until the simulation is restarted.

3.2.1 PR2 Rotation

The PR2 robots body orientation is controlled by publishing joint angles to the "/pr2/world_joint_controller/command" topic. As publishing messages to the provided topic return immediately, we need to monitor the joint states of the PR2 so we know when it is in the desired location. To do this we are listening in on the "/pr2/joint_states" topic.

Control of the torso joint is then achieved in a similar method to that used in the class notes for writing ROS Nodes (ROS Essentials, Section 3). The basic algorithm is outlined below:

1. Publish provided joint angle to the "/pr2/world_joint_controller/command" topic.
2. While True:
    a. Wait for a response from the "/pr2/joint_states" topic.
    b. Get the last entry from position component of the previous response.
    c. Check if that entry is within 0.05 units of the target response
        i. If it is, exit the routine
        ii. Otherwise, continue



"PR2 Robot rotating"

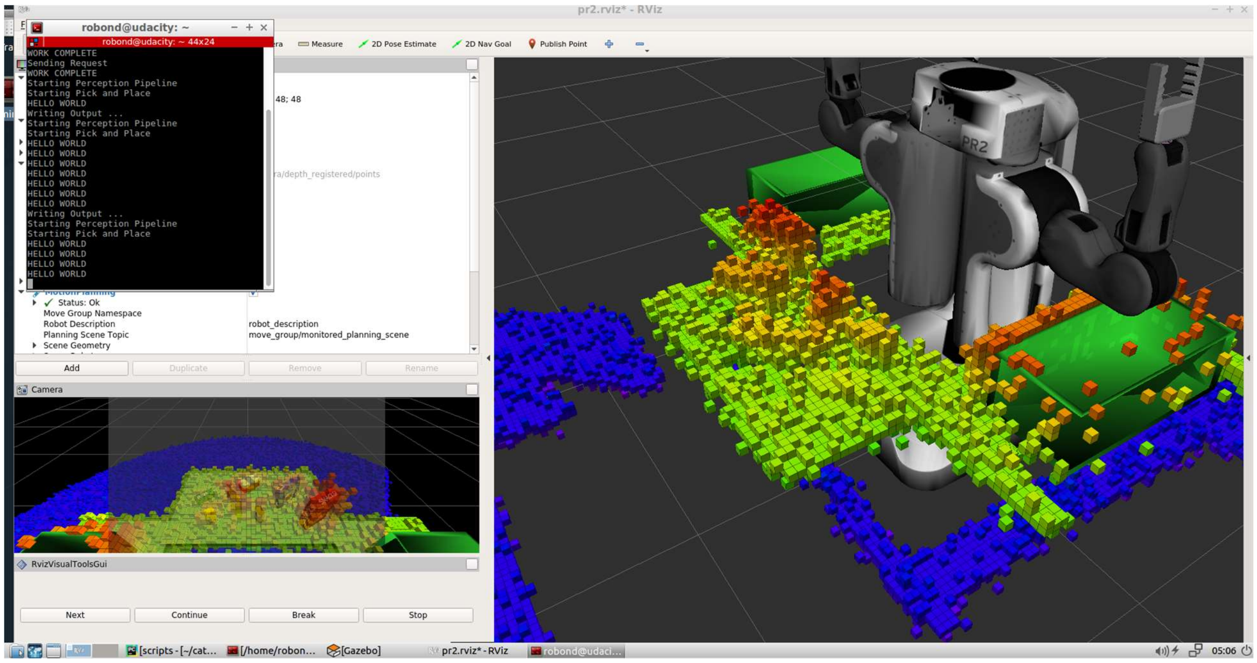
3.2.2 Building the Collision Point Cloud

Our collision map is built in several stages. The first stage of the map is built when the PR2 is in the READ_LEFT, and READ_RIGHT states. In these states, the data from the RGBD camera is cached as left / right collision clouds respectively. These are cached as its unlikely that the area of interest will change significantly and rebuilding them for each operation would require that the PR2 perform the rotation steps yet again. Since this rotation step is rather slow, the decision was made to have the robot read the data for this sections in only once.

Once the two side tables have been mapped, the PR2 returns to its central position and completes the perception pipeline, whose results are then piped into the pick and place routine. As most of the details of this routine will be explained in later sections, we will only look at those parts of the pick and place routine that are pertinent to constructing the collision map here.

Point clouds from the two drop boxes, the table, and the non-target objects are concatenated together into one large point cloud to construct the collision map. This collision map is then published to the '/pr2/3d_map/points' topic before continuing on to the next stage of the pick and place process.

An example of the resulting map is shown below:



```
WORK COMPLETE
Sending Request
WORK COMPLETE
Starting Perception Pipeline
Starting Pick and Place
HELLO WORLD
Writing Output ...
Starting Perception Pipeline
Starting Pick and Place
HELLO WORLD
HELLO WORLD
HELLO WORLD
HELLO WORLD
HELLO WORLD
HELLO WORLD
HELLO WORLD
Writing Output ...
Starting Perception Pipeline
Starting Pick and Place
HELLO WORLD
HELLO WORLD
HELLO WORLD
HELLO WORLD
```

"PR2 3D Collision Map"

3.3 Find Centroids

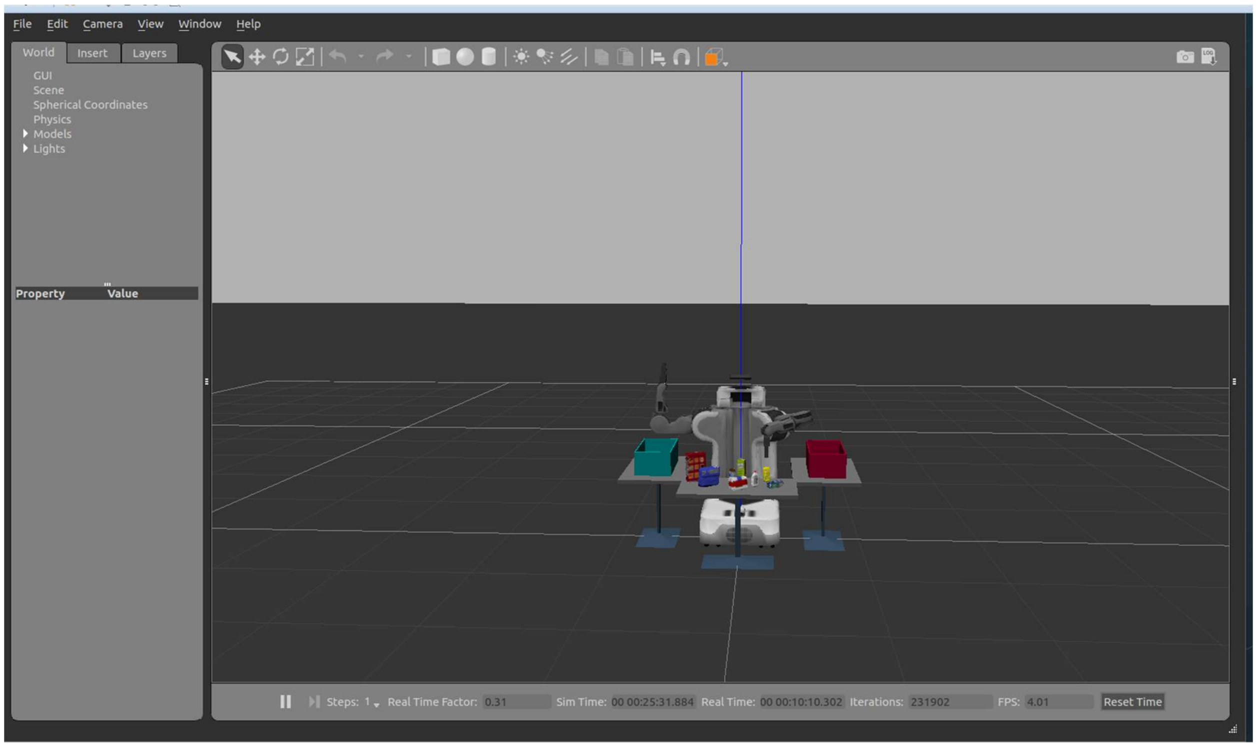
Before we can make use of the Pick and Place routine, we will need to know the centroids of the detected objects. To do this we simply loop through the list of detected objects and calculate the mean value of their point clouds. We then output a list of labels and a list of centroids (in the same order, so for example, index 2 of the labels corresponds to index 2 of the centroids). The code snippet that achieves this was provided in the Udacity course materials under the 3D Perception project in the section on outputting YAML files.

3.4 Pick And Place Routine

We will be using the "pick_place_routine" service to perform the actual pick and place operation in the simulation. To make use of this service we need to collect a few parameters:

Parameter	Description
Test Scene Number	The number of the test scene being used for the pick and place operation. This can be between 1 and 3 inclusive.

Object Name	The label of the object we are doing the pick and place operation for.
Arm	Left or Right depending on the arm of the PR2 that should be used for the pick and place operation.
Pick Pose	The target pose for the arm to pick up the target object.
Place Pose	The target pose for the arm to drop the target object.


File Edit Camera View Window Help
World Insert Layers
GUI
Scene
Spherical Coordinates
Physics
Models
Lights
Property Value
Steps: 1 Real Time Factor: 0.31 Sim Time: 00 00:25:31.884 Real Time: 00 00:10:10.302 Iterations: 231902 FPS: 4.01 Reset Time

"PR2 during pick and place operation"

3.4.1 Test Scene Number

This is simply a constant. Its value corresponds to the selected World file (as provided in the Udacity Perception project) and the corresponding training data the perception pipeline is working with. Any time the Launch file is updated to use a different world / pick list file, this constant needs to be updated to reflect the new state.

3.4.2 Object Name and Arm

The Object Name and Arm are determined from the values provided by the '/object_list' entry obtained from the ROS parameter server. This provides us with a list of pick and place requests, of which we are interested in two properties.

Property	Description
Name	The name of the object we should pick up next.
Group	Green or Red – determines which box we want to drop the object in once we've picked it up.

The name parameter maps directly onto the object name. For the arm name, we look at the group and choose the arm that corresponds to the dropbox we want to place the object in. In our case, this will mean that the green box corresponds to the right arm, and the red box corresponds to the left arm.

3.4.3 Pick Pose

Now that we have the label of the target object, we can build the pick pose command. To do this we search for the index of the label of the target object in the centroids we generated before proceeded to perform the pick and place operation. Unfortunately, the object detection algorithm used in the perception pipeline cannot identity all the objects all the time. This means that occasionally, the requested object is not found in the list of centroids. When this happens we simply skip the current object and move on to the next.

If we can find the centroid for the requested object, we then create an instance of the Pose object. The Pose object has two parts of interest – the position and the orientation. There does not seem to any information on the correct orientation for the end effector in the notes, but it was observed that the system works if the orientation is set to the zero vector. We use the position of the target objects centroid as the position for the Pose object.

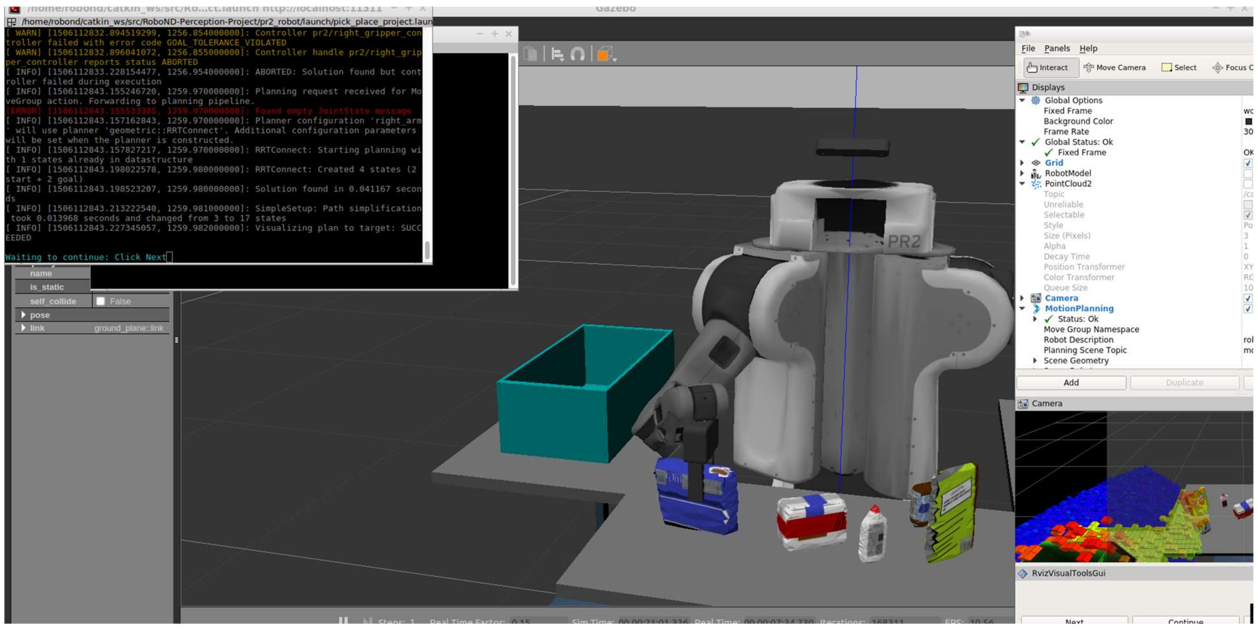
It should be noted, that either do to a coding error which could not be isolated, or some defect in the simulation, the arm doing the pick does not appear to be able to fully grasp the object in question. It can place the end effect in the appropriate position and does appear to close its gripper over the object, but when the arm is moved back upwards the targeted object does not appear to be attached.

In other cases, the gripper simply hovers over the target object, never lowering itself to a point where it can be picked up. Attempts were made to coerce the arm lower in these cases by adding a small perturbation in the z direction but this did not seem to help the situation.

3.4.4 Place Pose

Finally, we can create the place pose for the object. There are two possible place poses for an object, dependent on the choice of arm. For each arm, we choose a position above the closest drop box. So, the right arm's place pose moves to the position above the green box, and the left arms place pose uses the position above the red box. As before, there appears to be no guidance on the choice of an orientation for the place pose but the zero vector appears to work well.

In the class materials, it mentions that we may wish to perturbate the place pose by a small random amount to avoid accidently stacking objects on top of each other. Unfortunately, as for whatever reason the gripper does not appear to be working correctly on my machine I was not able to test this and thus left this step out of the code.

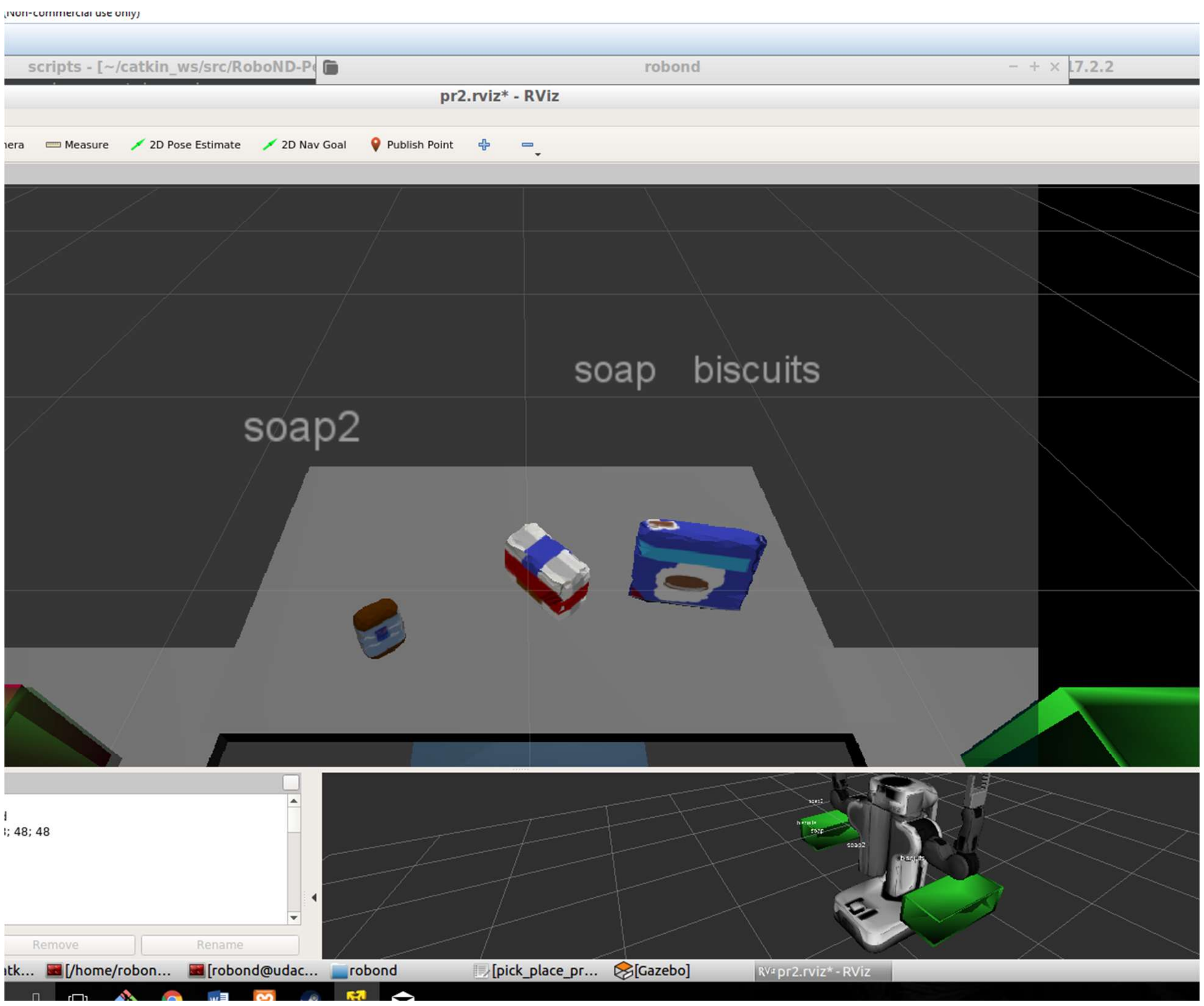


```
/home/robond/catkin_ws/src/RoboND-Perception-Project/pr2_robot/launch/pick_place_project.laun
[ WARN] [1506112832.894519299, 1256.854000000]: Controller pr2/right_gripper_con
troller failed with error code GOAL_TOLERANCE_VIOLATED
[ WARN] [1506112832.896041072, 1256.855000000]: Controller handle pr2/right_grip
per_controller reports status ABORTED
[ INFO] [1506112833.228154477, 1256.954000000]: ABORTED: Solution found but cont
roller failed during execution
[ INFO] [1506112843.155246720, 1259.970000000]: Planning request received for Mo
veGroup action. Forwarding to planning pipeline.
[ERROR] [1506112843.155533385, 1259.970000000]: Found empty JointState message
[ INFO] [1506112843.157162843, 1259.970000000]: Planner configuration 'right_arm
' will use planner 'geometric::RRTConnect'. Additional configuration parameters
will be set when the planner is constructed.
[ INFO] [1506112843.157827217, 1259.970000000]: RRTConnect: Starting planning wi
th 1 states already in datastructure
[ INFO] [1506112843.198022578, 1259.980000000]: RRTConnect: Created 4 states (2
start + 2 goal)
[ INFO] [1506112843.198523207, 1259.980000000]: Solution found in 0.041167 secon
ds
[ INFO] [1506112843.213222540, 1259.981000000]: SimpleSetup: Path simplification
 took 0.013968 seconds and changed from 3 to 17 states
[ INFO] [1506112843.227345057, 1259.982000000]: Visualizing plan to target: SUCC
EEDED

Waiting to continue: Click Next
```

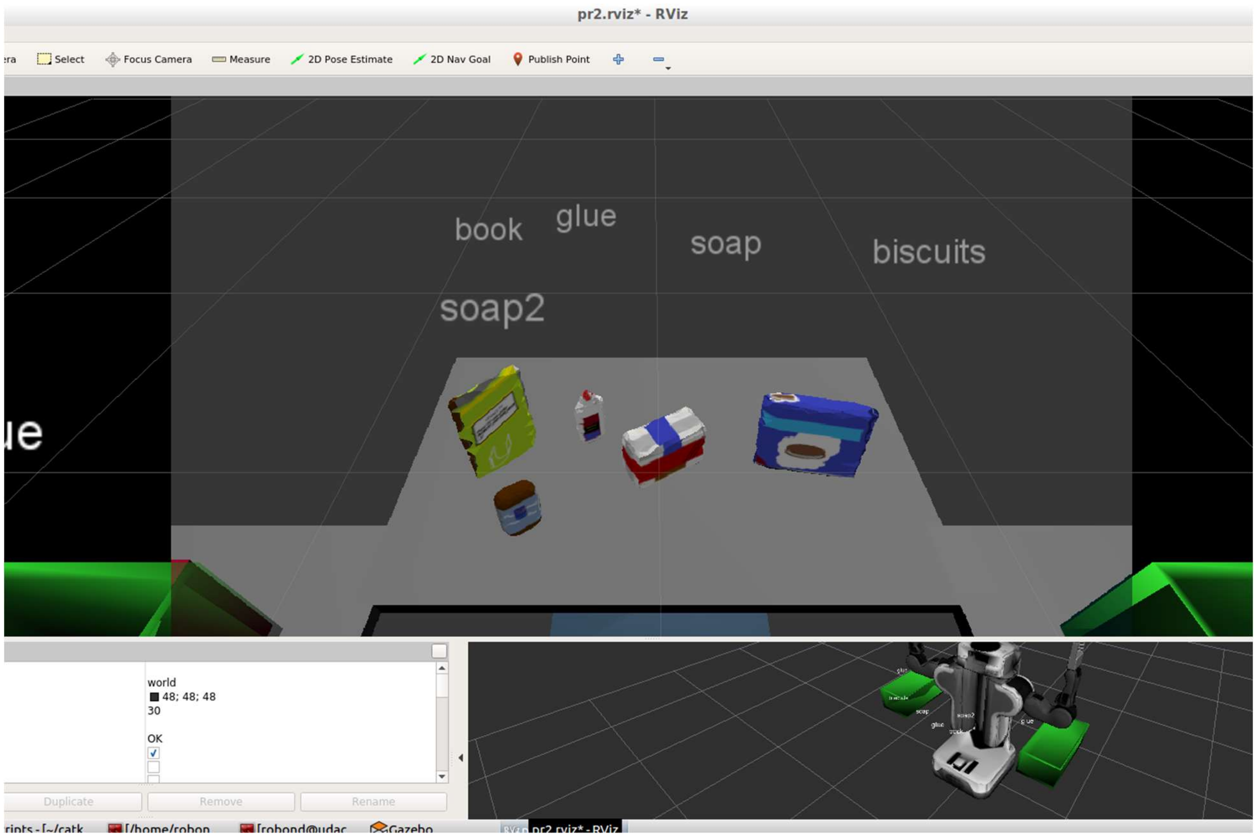
"Getting some tasty biscuits"

4 Discussion

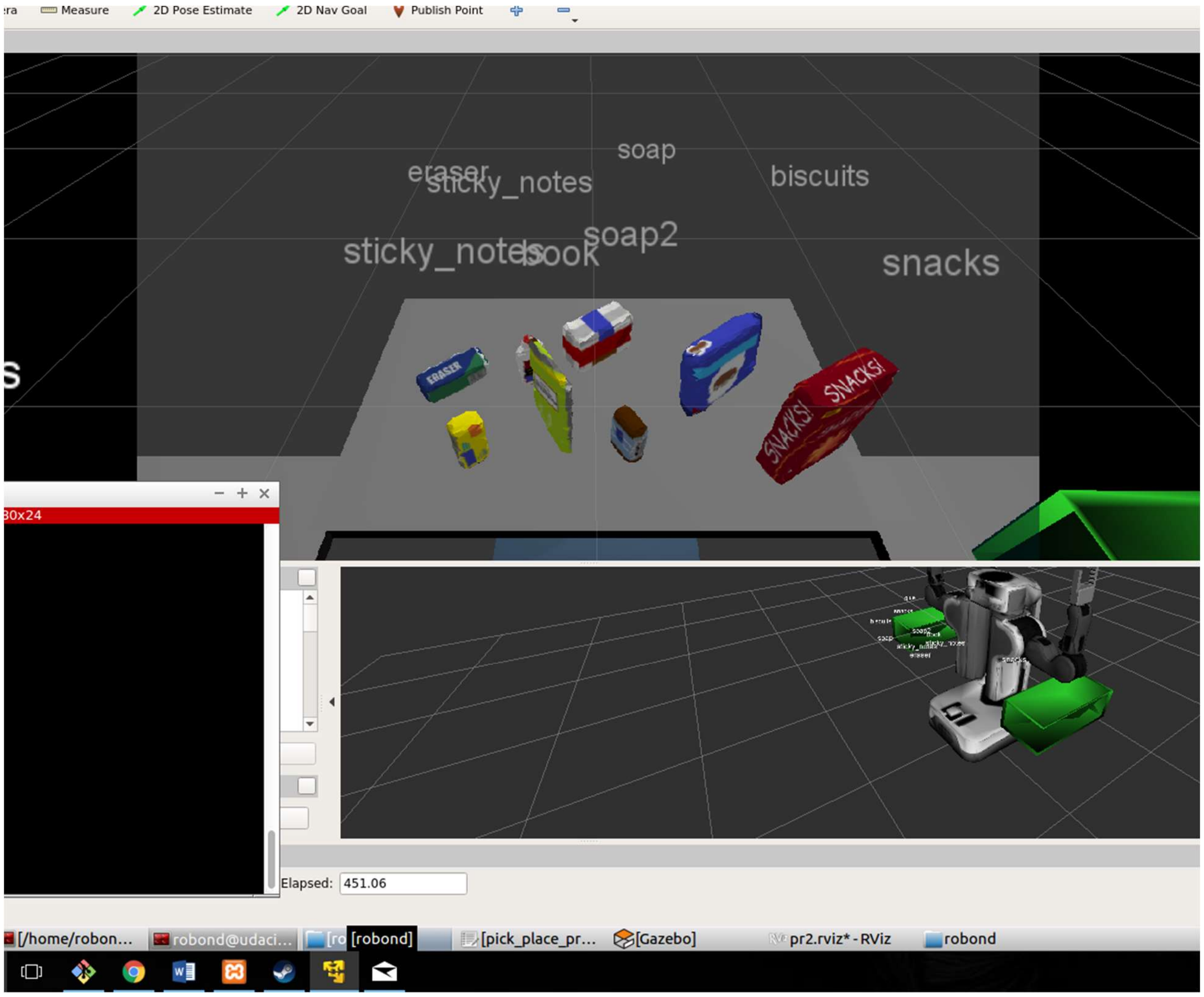
Using the methods outlined in this paper the pick and place task were completed with a reasonable level of accuracy. The system can correctly identify 3/3 objects in the first world, 4/5 objects in the second, and between 6/8 and 7/8 objects in the third world.



"World 1 Labels"



"World 2 Labels"



"World 3 Labels"

Examples of the classifier in action in each of the three worlds are shown above. To maintain clarity, I disabled the collision map for the construction of these images.

As can be seen in the provided images the system is not quite perfect. It occasionally misidentifies a few of the objects which can cause issues with the pick and place system. This might be improved by using a more robust object classifier – perhaps using multiple different classifiers and taking a vote for each of the detected object segments regarding the objects identity. This would slow down the already slow object recognition system however, and as such may not be practical.

The pick and place behavior itself appears to be flawed, and the cause of this could not isolated. It seems that the robot will simply not pick up objects on occasion even when the gripper is directly over them. Attempts to gently push the gripper in the desired direction to solve this problem were to no avail as any significant perturbation of the target angle causes the pick and place routine server to reject the command.