Department of Computer Science, University of Maryland

Exploration of Objects with the Baxter Research Robot Using an RGB-Depth Sensor and Point Cloud Library

By: Ishaan Parikh
Advisors: Aleksandrs Ecins, Cornelia Fermuller, Yiannis Aloimonos

May 2016

Abstract

Computer vision has many applications with household robots. One of the main problems associated with this kind of robot is the ability to recognize and segment arbitrary objects. In recent years, a variety of new algorithms have been proposed for object detection. However, these are limited by the viewpoint from which the object is captured. Characteristic parts of the objects could be missing from the detection because of the camera's viewpoint. This research introduces an approach that utilizes the moveable arm of the Baxter Research Robot to amass more data from different viewpoints and eventually distinguish objects.

First, the ASUS xTion Camera is used to obtain point cloud information. Using Point Cloud Library (PCL), the cloud is segmented, and the plane of the table is identified. Following segmentation for tabletop objects, the robot's arm is moved to another point so the camera is able to collect more information about the same objects. This reduces total occlusion and maximizes the chances of being able to recognize distinct objects. By collecting from three different viewpoints in total, the robot has learned much more about the structure of the object(s) at hand.

The algorithm is able to learn more about the object, and hence offer a more accurate detection. This will hopefully advance the eventual goal of engineering robots to interact with arbitrary objects in cluttered environments.

Background

ROS (Robot Operating System)

ROS (Robot Operating System) is an open-source operating system for different kinds of robots. It provides many features commonly found in an operating system, such as hardware abstraction, message passing between processes, package management, and low-level device control. It also allows users to build, write, and run code across multiple computers [1].

The ROS model is a peer-to-peer network of many processes (that can be distributed across machines if needed) that are joined through the ROS communication schema of topics, nodes, and servers [1].

Figure 1: In this classic example of ROS communication, RQT Graph plots the method of messaging across nodes. The */teleop_turtle* is a node that publishes across the topic */turtle1/command_velocity*. The subscriber node */turtlesim* then picks up this data, and is free to use it however. Source: http://wiki.ros.org/ROS/Tutorials/UnderstandingTopics

The filesystem contains many packages, which are the software organization units of ROS code. Individual packages can contain executables, scripts, libraries, etc. Within each package is also a manifest (*package.xml)*, which serves as a description of the package. It defines dependencies and information like version, license, etc [2].

One type of executable inside a package is called a node. Nodes are unique because they utilize ROS to communicate with other nodes across the filesystem. Nodes publish messages to topics, which serve as collectors or distributors for whatever information a node may send out. Declaring a node within a script gives it an "identity" within ROS such that it can now communicate with the rest of the

system. In order for two nodes to communicate (one publisher and one subscriber), each must be using the same data type within the message (figure 1) [2, 3].

Point Clouds

A point cloud is a list of data points measuring X, Y, and Z coordinates of objects in the camera frame. As opposed to 2D images that give you RGB information, XYZ coordinates give you 3D geometric information about the scene. This makes it much easier to segment objects, versus doing it only off color information. The camera used for this research (Asus Xtion PRO) is an RGB-D camera, which means it reads not only color values, but also depth cloud information [4].

For this research, only depth cloud information is going to be analyzed. The depth cloud will be segmented with functions provided by the Point Cloud Library (PCL). PCL makes it simple to accurately segment flat surfaces and objects on top of tables, amongst others tasks [4, 5].

Point Cloud Library (PCL)

The Point Cloud Library (PCL) is an open source library for 3D image and point cloud processing. The library has algorithms for solving problems such as surface reconstruction, registration, model fitting, and segmentation. PCL is also easily integrated into ROS software packages, which is why it is heavily utilized in this project [6, 7].

Baxter Research Robot

The Baxter Research Robot is a versatile, affordable research machine that is widely used across labs and classrooms nationwide. There are applications in numerous computing fields including human-robot interaction, machine learning, and planning. This robot is used in this project because of its movable arms and its previous integration with planning [8].

Approach

The process that was followed can be broken down into six steps and is detailed in figure 2.

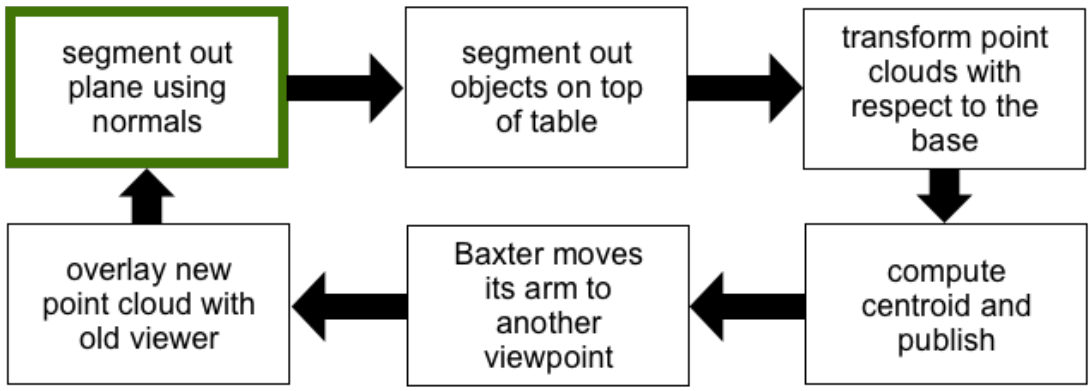

Figure 2: These are all the steps needed to amass more information about object point cloud information by moving the Baxter Research Robot's arm. The entire process had to be repeated three times for the sake of this project; in practice, this number can be increased with a more flexible arm to minimize occluded space.

Step 1: Segmenting Planes with Normal Vectors

PCL allows us to define the following objects:

1. `pcl::SACSegmentationFromNormals<pcl::PointNormal, pcl::Normal> seg`
2. `pcl::NormalEstimation<pcl::PointNormal, pcl::Normal> ne`

The first object requires parameters such as the input point cloud, normal vectors, model type, and distance thresholds. Basically, this allows us to control the parameters of the segmentation so the largest plane in the cloud is returned. Another similar object exists called `pcl::SACSegmentation<pcl::PointNormal>` which does not use normal vectors to assist with the segmentation. It was found that doing this segmentation with normals (the `pcl::NormalEstimation<pcl::PointNormal, pcl::Normal>` object) increases the chances of segmenting the largest plane with the first iteration of segmentation. However, the drawback is the large time it takes to compute the normal vectors. We create an object to store the plane function coefficients (represented by `coefficients` below), and another array consisting of all the indices of points of the original point cloud that exist in a plane (represented by `inliers` below). One call is made to the segment function:

```
seg.segment(*inliers, *coefficients);
```

and all the points of the plane and its function model are determined.

Because there are multiple flat surfaces in the scene, this process would return multiple planes so the table could be identified incase the first segmentation was incorrect, but this proved to be unnecessary when the camera was mounted. The table was simply too large a portion of the cameras view frame such that other planes were rarely given preference by the `SACSegmentationFromNormals` object.

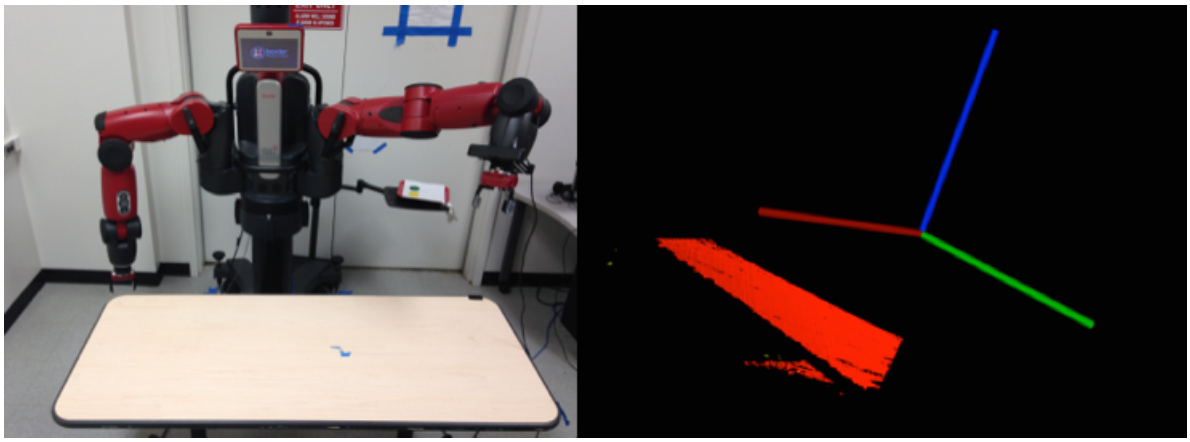

Figure 3: The two images are corresponding images from the real setup to the PCL visualizer. The plane is visualized without the rest of the point cloud in this visualization. The coordinate frame represents the base of the robot, which makes sense as the **z** axis (blue) seems to be perpendicular to the table.

Following the segmentation, a point cloud is not returned, just the location of the points in the original point cloud. PCL provides the object `pcl::ExtractIndices <pcl::PointNormal>`, which removes all the points corresponding to (inputted) indices and places them in a new `pcl::PointCloud<pcl::PointNormal>::Ptr`.

We can then add the plane point cloud to the PCL visualizer object as shown in figure 3.

Step 2: Segmenting Objects Above Plane

There are two important objects involved with this step:

1. `pcl::PointCloud <pcl::PointNormal>::Ptr convexHull(new pcl::PointCloud <pcl::PointNormal>);`
2. `pcl::ExtractPolygonalPrismData <pcl::PointNormal> prism;`

First, a convex hull (see figure 4) is defined. This is the smallest possible space to envelop all of the points in the table given from the plane segmentation. Second, the `ExtractPolygonalPrismData` object is similar to the `SACSegmentationFromNormals` object described above, except it does not look for a plane. Rather, this object is given a convex hull on an input point cloud (original point cloud) and a height threshold. For this project, the height restrictions were set from 1cm to 60cm off the table. By then calling the function:

```
prism.segment(*objectIndices);
```

The indices of all the points corresponding to the objects above the table is returned in `objectIndices`.

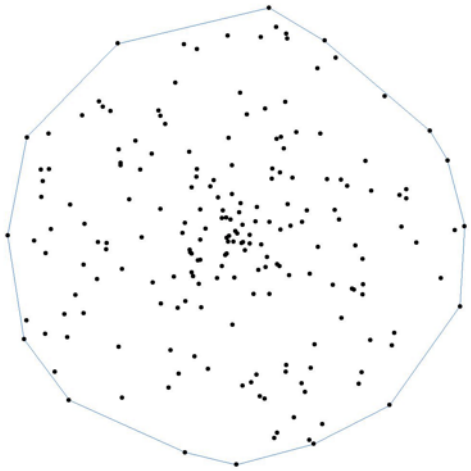

Figure 4: A convex hull of a set of points S in n dimensions is defined as the intersection of all convex sets containing S. For N points p_1, ... p_n, the convex hull C is given by

$$C \equiv \left\{ \sum_{j=1}^{N} \lambda_j \, p_j : \lambda_j \geq 0 \text{ for all } j \text{ and } \sum_{j=1}^{N} \lambda_j = 1 \right\}.$$

Working with a convex object makes it far easier to extract objects above the plane. It translates a set of points that are close to each other to a contiguous object. Source: http://mathworld.wolfram.com/ConvexHull.html, http://xlr8r.info/mPower/examples.html

By then extracting these points from the original point cloud (similar to first step), objects can then be place on the PCL visualizer as seen in figure 5.

Step 3: Transform Point Clouds with Respect to Baxter Robot

Because the camera has no reference frame, it will display consecutive incoming point clouds as if the objects are moving (opposed to the reality that the camera is moving). Since there is no way for the camera to know where it is going, transforms are used to alter the point clouds in such a way to visualize them consistently.

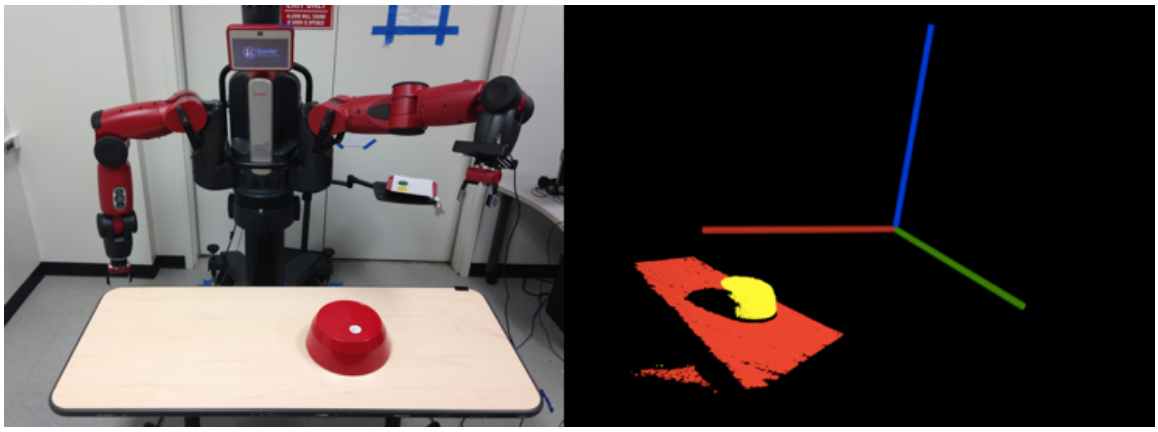

Figure 5: The two pictures are corresponding images from the real setup to the PCL visualizer. The plane and the object are shown without the rest of the point cloud in this visualization. The coordinate frame represents the base of the robot as in figure 3. The perspective has been slightly altered in the PCL visualized to show the occlusion that happens behind the object. The goal of this project is to use the mobility of Baxter's arm to decrease the unknown area.

The camera is mounted on the robot's left arm, specifically very close to the gripper. There are certain points on the robot where ROS can return the transformation from the *base* (static) to said frame (non-static). There are two of these frames close to the mounted camera: *left_hand_camera*, and *left_hand_camera_axis*. Although *left_hand_camera_axis* is closer to the camera, *left_hand_camera* is used because it has the same orientation as the camera. This means that the *left_hand_camera*'s frame points its **x**, **y**, and **z** axes in the same orientation as the camera's (versus the *left_hand_camera_axis* does not).

It is important to discuss the reference frames and how exactly a "transform" is returned. We use the object `Eigen::Affine3f eigen3f`, which is a 3 by 3 matrix of floats that contain a transformation. This matrix has two parts: the translation matrix, and a quaternion. The translation matrix is a 3 by 3 matrix that maps the *base* frame's origin to the *left_hand_camera*'s origin (assumed to be the camera's

origin). The quaternion is a four dimensional vector that represents a rotation in 3D space; this is the format ROS uses for transforms.

Baxter keeps track of transformations between any two of its reference frames, and so the following function is called:

```
listener.lookupTransform("left_hand_camera", "base", ros::Time(0), transform);
```

The objects are as follows:

1. `tf::TransformListener listener` – Object that is able to access the transforms sent over a ROS topic from Baxter to the node.
2. `tf::StampedTransform transform` – Object that stores the transform message from Baxter received by `listener`.

The function stores the transform in the `tf::StampedTransform` object, and the function `tf::transformTFToEigen(transform, eigen3d)` is called (where `Eigen::Affine3d eigen3d` is a matrix of doubles rather than floats – we convert this into an `Eigen::Affine3f` immediately after). To transform the point clouds to be with respect to the base, we call the function:

```
pcl::transformPointCloud(*not_transformed_cloud, *transformed_cloud, eigen3f);
```

to transform everything.

This is helpful to this project because now when Baxter moves its arm to a different position, the PCL visualizer appears to “learn more” about the object, versus blindly inserting point clouds into a view frame.

Step 4: Computing and Publishing Centroid

This step deals with a simple function to find the centroid of a point cloud, and ROS mechanics to send a `pcl::PointStamped` across a topic. The difference between a regular `pcl::PointNormal` and a `pcl::PointStamped` is that the first only has x, y, and z coordinates. The second has all of the above, but also a frame, which we pass in as "*base*" so it is in the same place with respect to the base of Baxter.

When we want to visualize this in a program like RViz (ROS' visualization tool), the stamped feature makes it display in the correct relative position.

The original goal was to use the centroid to compute the next position for the robots arm to move to. However, this was not accomplished due to uncertainty with the arm movement.

Step 5: Movement of Baxter's Arm

In order to move Baxter's arm, a subscriber node listens to a topic that sends a notification every time the camera finishes collecting data. When this signal is received, the script knows which step (out of three) the program is in, and moves the robot to the necessary location.

Before this script was written, the topic */robot/limb/left/endpoint_state* was listened to when the robot was put in the desired positions. These are saved as the first, second, and third poses. A pose is essentially the transformation from *base* to the *left_hand_camera*. The aforementioned topic gives all the information needed to create a `geometry_msgs.msg.Quaternion quat` and `geometry_msgs.msg.Point loc`. The new joint information is generated by running:

```
limb_joints = test_ik_solver.ik_solve('left', loc, quat)
```

The Baxter SDK gives the above method; it is able to figure out exactly what the joint angles need to reach the desired pose. We then call:

```
arm.move_to_joint_positions(limb_joints,timeout=20,threshold=0.008726646)
```

This moves the arm to the desired position (if the proper joint angles were found). The timeout signifies the maximum amount of time the robot can take to move, and the threshold represents the epsilon around the target (in meters) point the arm can consider accurate.

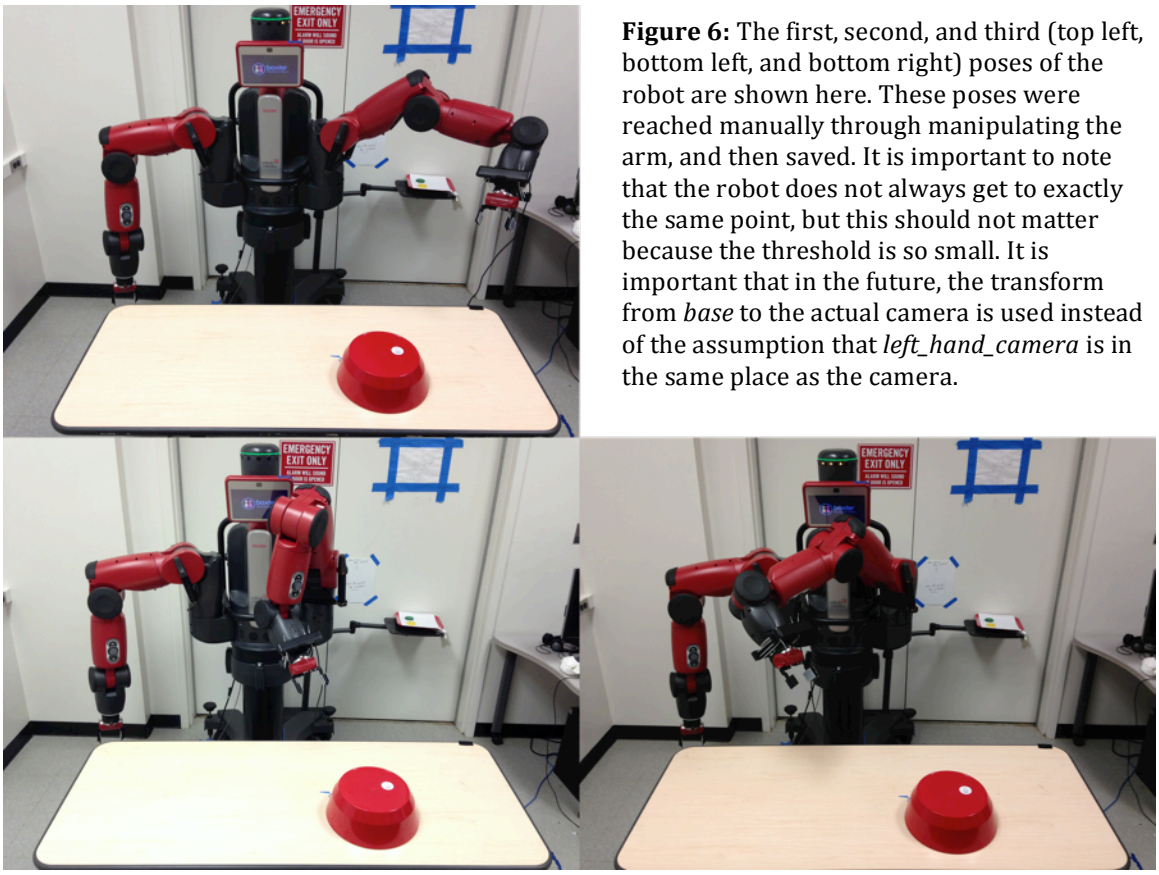

Figure 6: The first, second, and third (top left, bottom left, and bottom right) poses of the robot are shown here. These poses were reached manually through manipulating the arm, and then saved. It is important to note that the robot does not always get to exactly the same point, but this should not matter because the threshold is so small. It is important that in the future, the transform from *base* to the actual camera is used instead of the assumption that *left_hand_camera* is in the same place as the camera.

Occasionally, setting the points manually did not work. Depending on how the arm starts (pre-program run), it may or may not next point. This is something that needs to be addressed with future research. Figure 6 shows exactly where we wanted the camera to be at each data collection.

Step 6: Overlaying Point Cloud Data

The script that creates and visualizes the point clouds is actually a custom class. One of the member objects of this class is:

```
pcl::visualization::PCLVisualizer viewer;
```

PCL's visualizer needs only one input before displaying point clouds: a reference frame. Because of the function call:

```
pcl::transformPointCloud(*not_transformed_cloud, *transformed_cloud, eigen3f);
```

this does not have to be manually set. All the clouds are in the reference to the same frame: *base*. By calling "`viewer.addCoordinateSystem(1.0);`", the axes shown in above figures is displayed. These axes are located at the *base* frame of the robot.

Each iteration through the program simply requires one call to:

```
viewer.addPointCloud <pcl::PointNormal> (transf_objs, color, "obj cloud");
```

where `transf_objs` is a point cloud and `color` is a color setting. By default, the clouds are overlain, as shown in in figure 7.

Each iteration slightly compounds errors and discontinuity of point clouds. When the camera moved locations, it could detect something off the table as part of the plane or the objects. In addition, the plane is occasionally thought to be part of the object. Although these are errors with PCL's segmentation, there are filters that can be applied to deal with this effectively and efficiently.

In figure 7, it is clear that in the third iteration, the circular bowl (object) is distorted. This is because of the discrepancy between the pose of the *left_hand_camera* and the ASUS xTion's position. More calibration will be necessary to correct this issue as wall as disregarding the assumption that the camera is at the same spot as *left_hand_camera*.

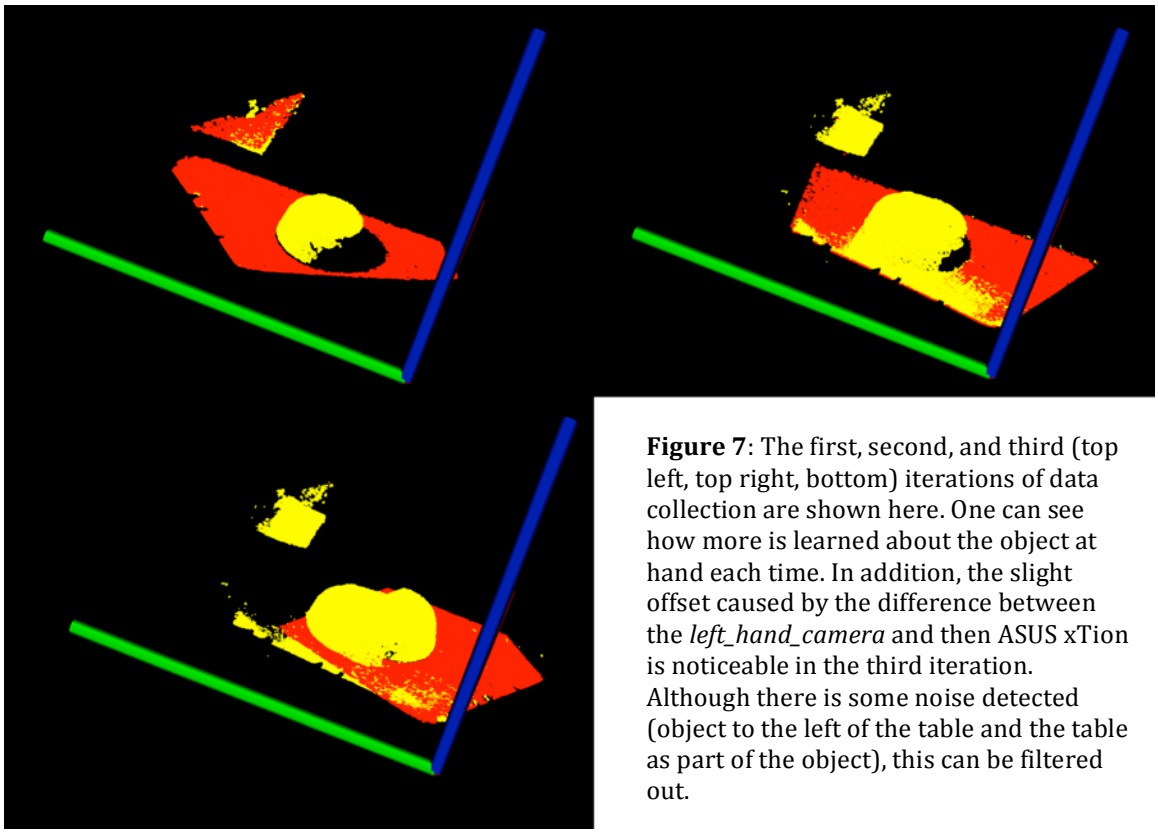

Figure 7: The first, second, and third (top left, top right, bottom) iterations of data collection are shown here. One can see how more is learned about the object at hand each time. In addition, the slight offset caused by the difference between the *left_hand_camera* and then ASUS xTion is noticeable in the third iteration. Although there is some noise detected (object to the left of the table and the table as part of the object), this can be filtered out.

Conclusion/Future Work

Overall, this research was successful on two fronts. First, this successfully uses the arm to obtain more information about the point cloud of the object(s) on the surface. A problem with many algorithms that segment and detect objects is that the viewpoint does not capture enough information about the objects (due to occluded space). This research addresses this problem by fusing the views and providing a larger data set for the same object. A problem with motion planning and movement is engineering a robot to be able to look at a new scene and distinguish/interact with objects. Having a more holistic view of the scene is a key part of solving this problem in the long run.

In addition, the movement of the research robot's arm to different viewpoints is a preliminary step to optimizing arm movement for the same purpose. Eventually, it would be ideal for an algorithm to figure out what the "most occluded" space is, and then move to learn the maximum possible information about the point cloud. Another part to this problem is that with different tasks (reconstruction, segmentation, symmetry identification, etc.), the optimal camera path would change. This requires more research and analysis with the technicalities of specific algorithms, and then combining this knowledge with the Baxter planning.

In the Autonomy Robotics Cognition Lab at the University of Maryland, this research will feed into another project that aims to segment objects based on symmetry to make distinctions. When the projects are combined, the accuracy of symmetry segmentation with an increasing number of data points (as the camera moves) should be observable.

In future related research, it is important that the transform between the *base* of the robot and the Asus xTion camera is calibrated precisely, as the assumption that the camera is at the same location as the *left_hand_camera* can lead to inaccurate point clouds. In addition, filtering of point cloud data would lead to less uncertainty in terms of centroid computation and point cloud stitching.

Acknowledgements

I would like to thank Dr. Cornelia Fermuller and Dr. Yiannis Aloimonos for being my faculty advisors during the span of this project and permitting my use of the Autonomy Robotics Cognition Lab's Baxter Research Robot. In addition, Aleksandrs Ekins was extremely helpful for getting past the steep learning curve associated with ROS and PCL. He provided most of the support across the duration of the project. In addition, Ariyan Kabir and Kanishka Ganguly assisted with the technicalities of the robot.

References

1. Quigley, M., Conley, K., Gerkey, B., Faust, J., Foote, T., Leibs, J., Wheeler, R. and Ng, A.Y., 2009, May. ROS: an open-source Robot Operating System. In*ICRA workshop on open source software* (Vol. 3, No. 3.2, p. 5).
2. Garage, W., 2012. Robot operating system (ROS).
3. Crick, C., Jay, G., Osentoski, S., Pitzer, B. and Jenkins, O.C., 2011, August. Rosbridge: Ros for non-ros users. In *Proceedings of the 15th International Symposium on Robotics Research*.
4. Kramer, J., Burrus, N., Echtler, F., Daniel, H.C. and Parker, M., 2012. Object modeling and detection. In *Hacking the Kinect* (pp. 173-206). Apress.
5. Kramer, J., Burrus, N., Echtler, F., Daniel, H.C. and Parker, M., 2012. Multiple kinects. In *Hacking the Kinect* (pp. 207-246). Apress.
6. Rusu, R.B. and Cousins, S., 2011, May. 3d is here: Point cloud library (pcl). In *Robotics and Automation (ICRA), 2011 IEEE International Conference on*(pp. 1-4). IEEE.
7. Aldoma, A., Marton, Z.C., Tombari, F., Wohlkinger, W., Potthast, C., Zeisl, B., Rusu, R.B., Gedikli, S. and Vincze, M., 2012. Point cloud library. *IEEE Robotics & Automation Magazine*, *1070*(9932/12).
8. Robotics, R., 2013. Baxter Research Robot.