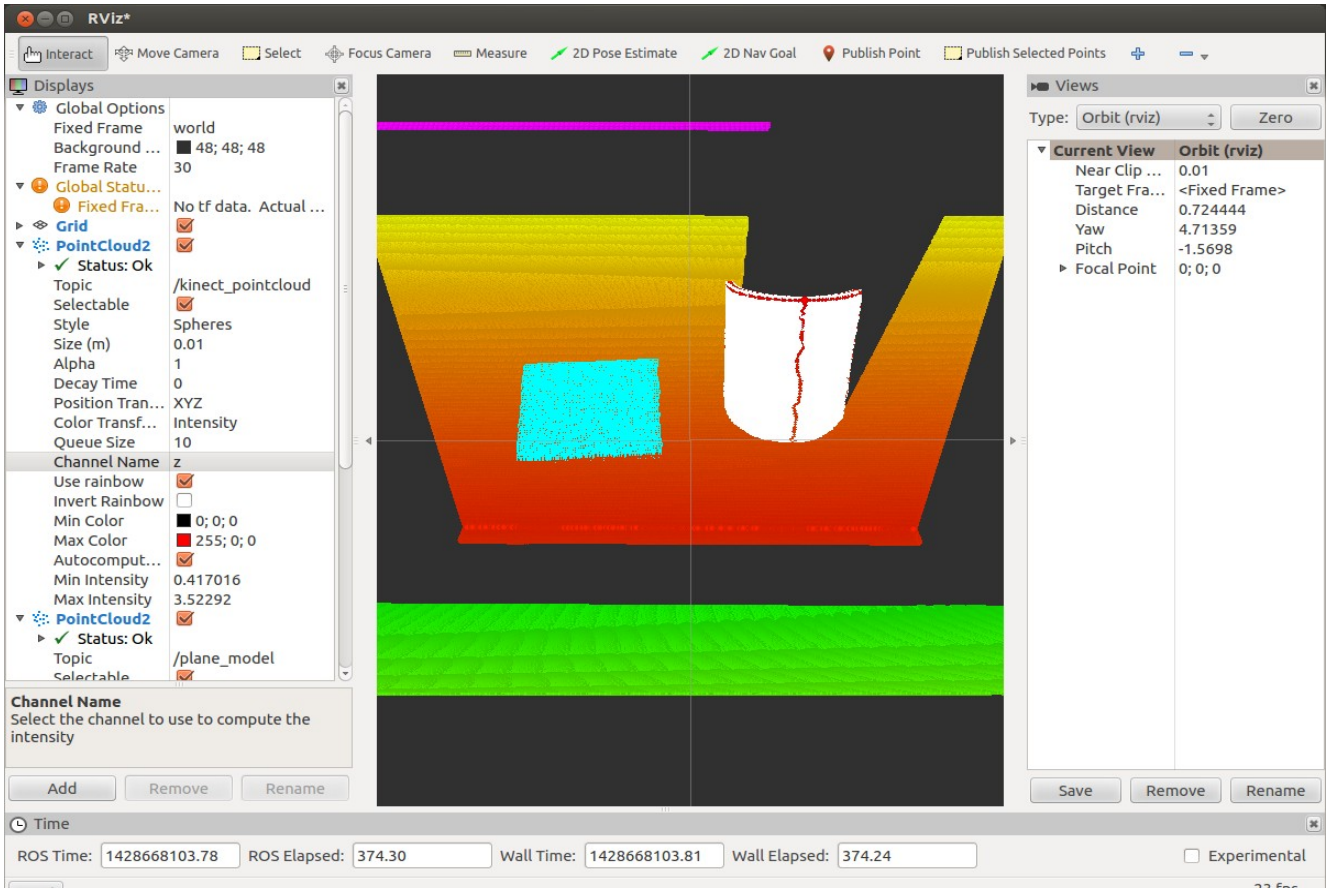
Team Gamma 4/10/15
PS7 : PCL smh150

PS7 : Point cloud processing
Github: https://github.com/ShaunHoward/cwru-ros-pkg-hydro-gamma/tree/master/catkin/src/cwru_376_student/point_cloud

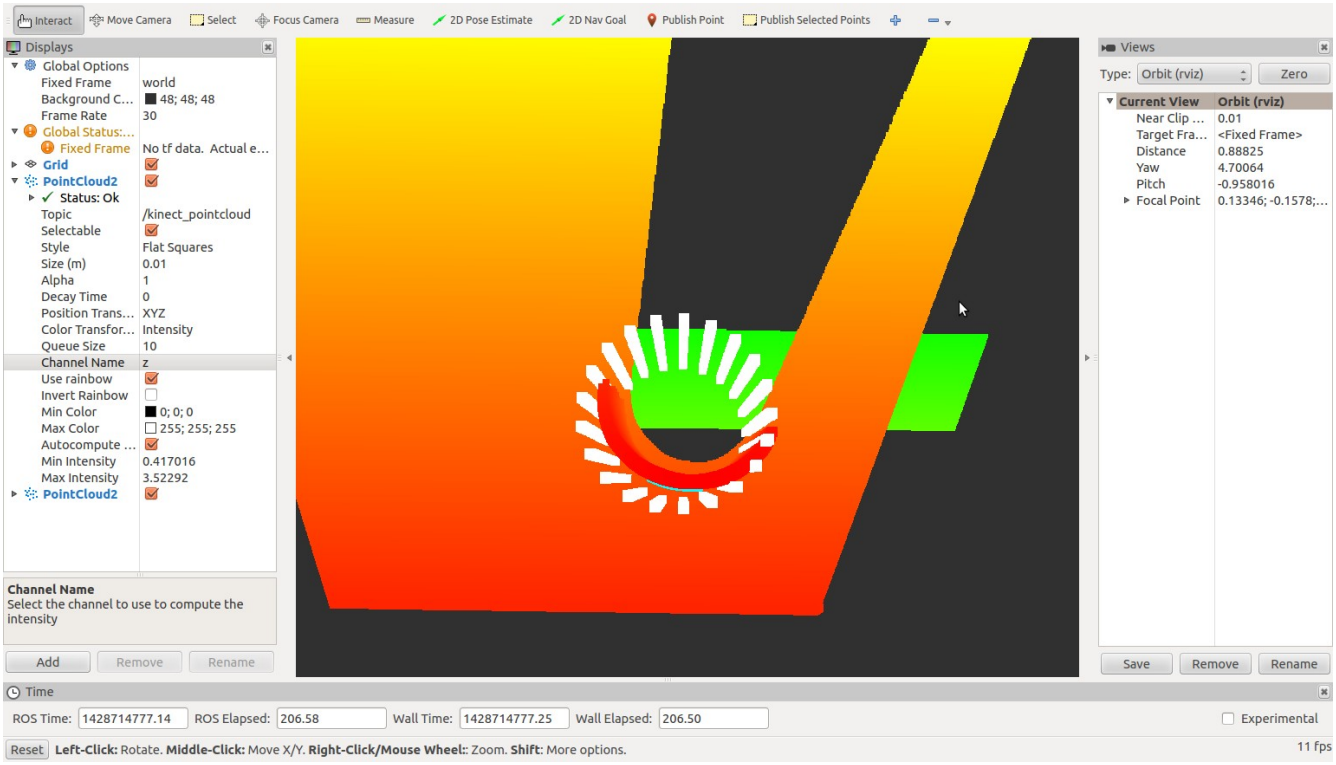
The program, “find_can.cpp”, outlined within this document is located in the “point_cloud” package in the GitHub student directory of the repository. The purpose of this project is to find a can on a table with a Kinect point cloud sample. This means that we want to find the can on the table and model it with our own point cloud. Thus we can estimate the coordinates of the entire can instead of just half of the can since the Kinect image is unidirectional. This sample is read in from the hard disk to the memory in order to process, step by step, and determine the best registration for a can point cloud model created by our program. To initiate each step in processing the point cloud, the “process_mode” service of the program published by the program must be called. Essentially, each step in processing the point cloud gets us closer to our goal of having our point cloud model lined up with the can points according to the Kinect's image. The first step is to identify a plane for which the can should be on, via user-assistance by selecting a patch of points on the plane. This is done with the “Publish selected points” tool in Rviz. Then the second mode is called, to find the points above the given plane. The user must select a patch of points on the can and run this mode to identify points above the plane according to the patch. The next mode will make an estimated 3-D point cloud model based on the data gathered so far from the user and patch selection.

After running process modes 0 and 1, my results were the following:



We can see that the can point cloud was successfully filtered from the rest of the points.

Previously, mode 2 for making the initial point cloud, made a cloud with its origin at the origin of the centroid for the second selected patch. This allowed the can cloud to have a really rough estimate of the center of itself, but this estimate was too far off to register the can model properly with the point cloud by iteratively stepping it into place. To fix this, I still assign the patch centroid to the cylinder model origin, but I decrement the y-coordinate in place by the estimated radius of the cylinder. Essentially, this will move the model very close to the center of the actual can in the point cloud because it is moving back from the centroid of the points on the side of the can in the point cloud that were selected. A screen shot of my results from running process mode 2 is the following:

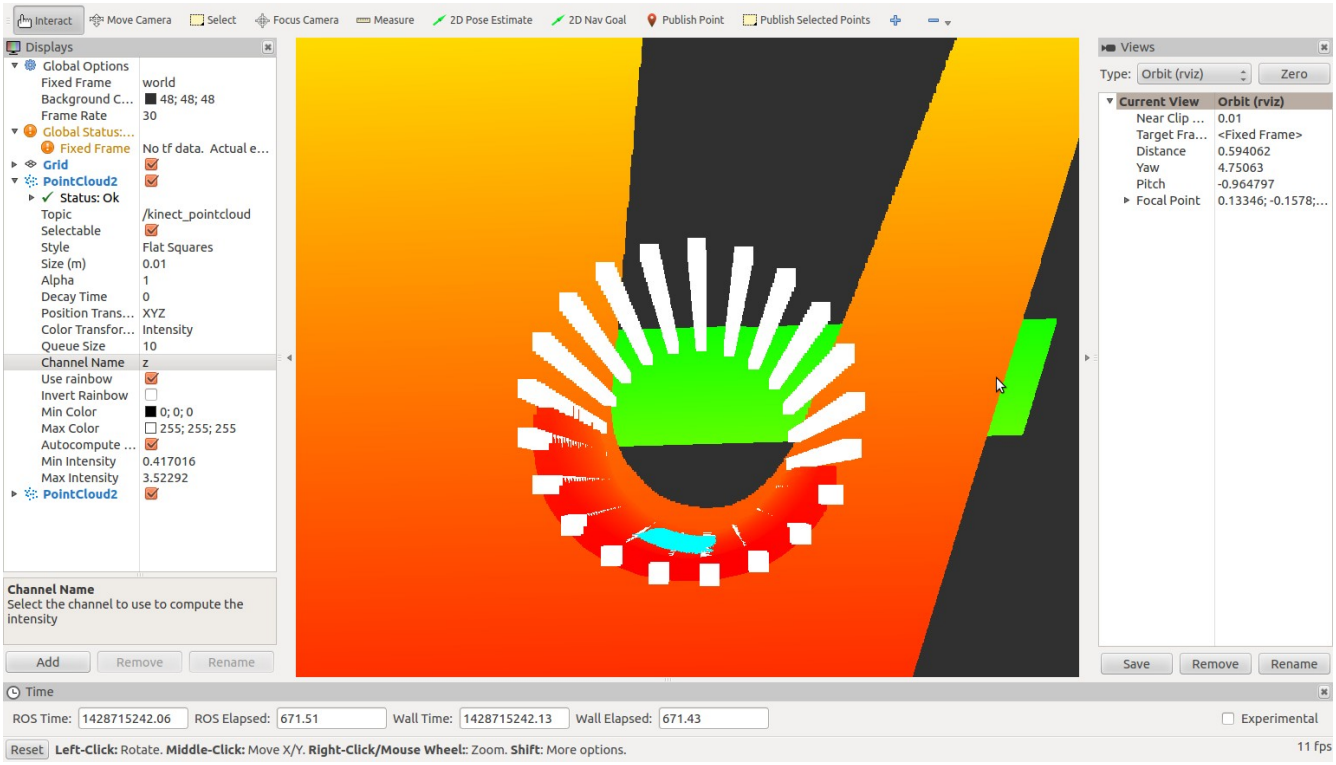


We can see that my modifications line the can model up pretty closely with the point cloud representation.

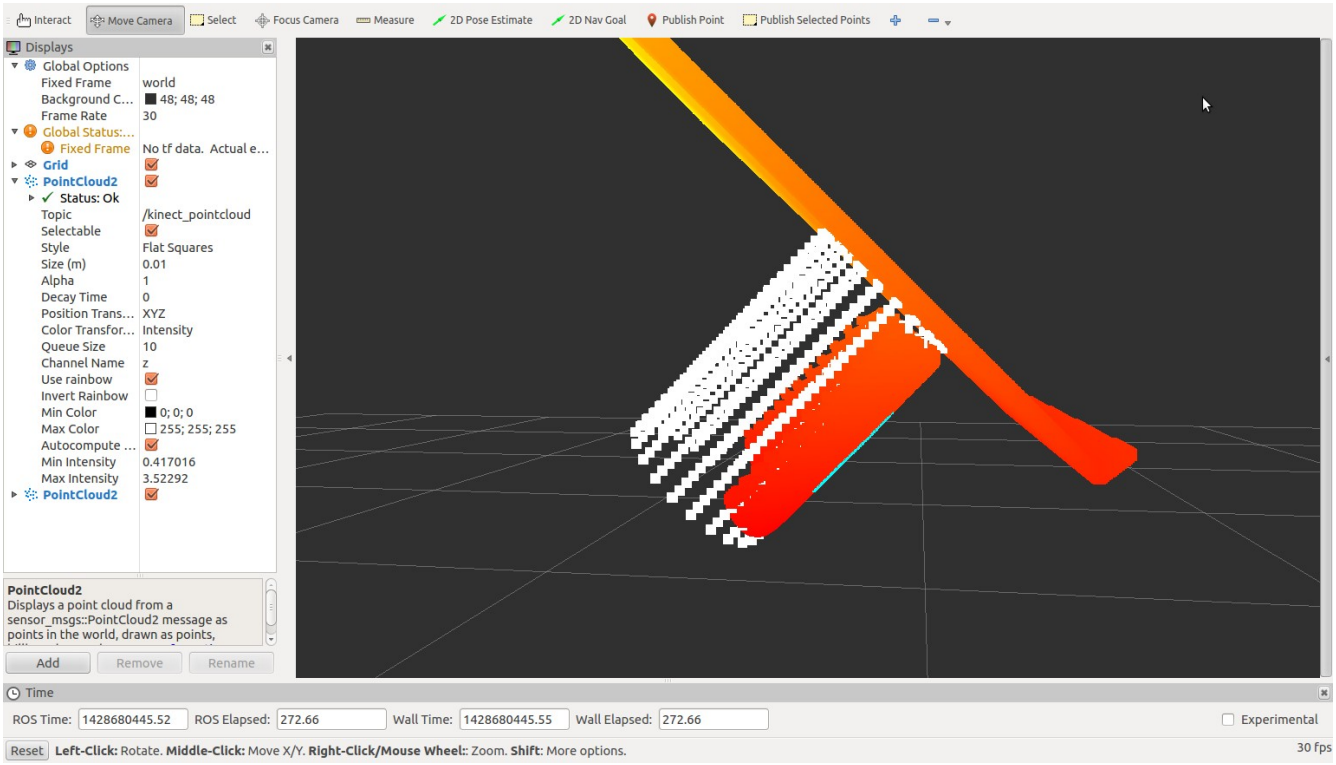
Mode 3 is used for the iterative stepping into place of the can model to align it with the can point cloud points. This mode previously did nothing. I modified this mode to use gradient descent in order to find the local minimum error that would register the can model with the actual can point cloud as close as possible. The secret here is to create a new vector called dEdC from dEdCx, dEdCy, and the can_center_wrt_plane z, since the z-axis is fixed to the table plane's z. Next, we compute the normal of this vector and assign it to dEdC_norm. From this we can improve can_center_wrt_plane x and y values with gradient descent. The following lines are the magic in the iterative stepping mode:

```
//use gradient descent to improve registration values
//from normalized error vector
can_center_wrt_plane[0] -= 0.005 * dEdC_norm[0];
can_center_wrt_plane[1] -= 0.005 * dEdC_norm[1];
```

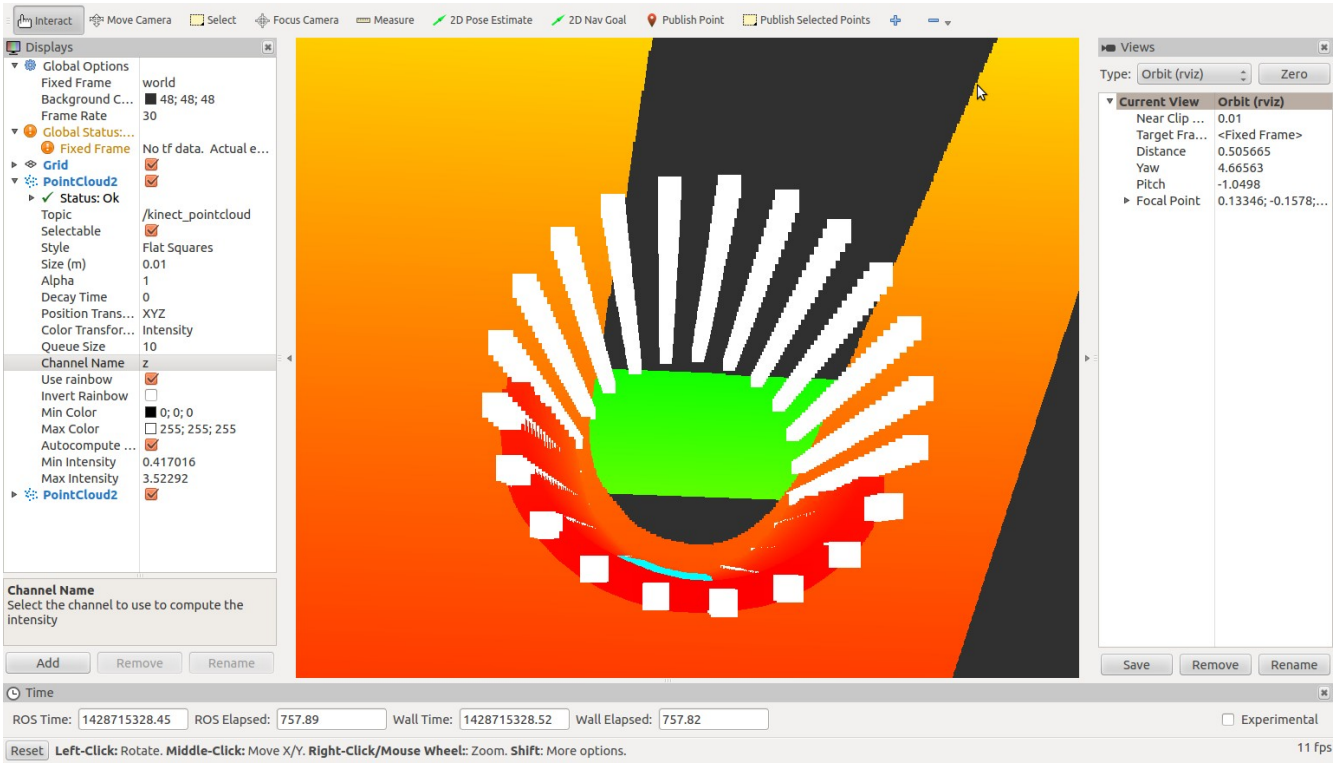
What this does is, based on normalized error, moves the can into place according the the current error that exists in its placement relative to the point cloud can origin. This method of optimization works fairly well. The step size and error can be minimized even further by decreasing the value 0.005 to a value like 0.001, however, more steps would be need to fit the model in place accurately. After running mode 3 approximately 7 times, the can model will be fit into place with the point cloud can. A few screen shots of this fit are below:



Above we see that the can model is aligned fairly well on the x and y-axes of the point cloud can.



In the second screen shot, we see that the z-axis of the can is also aligned with that of the plane of the table, which is assumed from a can standing on the table.



Above is a closer screen shot of the can model registration. Altogether, we see that the fit should be accurate enough for our purposes.

A video of me running the find_can program with all of the process modes will be posted to YouTube shortly. In this video I will demonstrate the accuracy of my iterative optimization algorithm.

In order to generalize this point cloud processing approach, we would have to do several things. Some of the considerations we would have to make would be the following:

- Automation of finding the plane we want based on some initial data given when the program is called (process mode 0)
- Automation of filtering the cloud points above the plane found (process mode 1)
- Automation of determining the objects on the plane
- Determining the shapes of the objects and some of their properties if we want to lift one up.
- Determining which object is the one we want, if not all.
- Constructing a model for that object based on our prior knowledge (process mode 2).
- Using some fuzzy logic or previously given data to determine the best guess for the origin of the model (modify process mode 2)
- Running process mode 3 in a loop until the error from the center of the point cloud points and the model are minimized

Altogether we can see that generalizing this process and making it robust may be difficult. It will require us to implement some better AI in order to determine the objects we want and their locations without much, if any, user input.

Some notes on what works and what does not:

Through experimentation, I noticed that I could implement a gradient ascent algorithm by reversing the – to a + in process mode 3. Running the find_can program with this caused the can model to move away from the desired origin. Thus, although gradient ascent worked, it was not what I wanted for this purpose. Hence, gradient descent and error minimization from the center of the point cloud points is what is desired and what I used in the end.

Another thing that was realized is how modifying vectors between the process modes produced variable results. Some of these results approached the desired goal and others were farther from the goal. In process mode 2, I experimented with moving the cylinder origin to several different points. I tried to use the normal vector from the latest patch of points given (finding the can from the point cloud) and translate the model forward/backward by one radius of the cylinder. This resulted in the can being farther out of place than desired because the can's bottom was the the center of the selected point patch on the can cloud. Then I tried modifying the g_plane_normal to use the normal from the latest patch again, and although the can model was aligned closer on x and y, the z was still too far away from the table. Eventually I came to the conclusion that using my current configuration would yield the best result.

All in all, I think that this program can be made better and more generalized with some better error

calculation (potentially) and automation, as outlined above. However, it does work well for scenarios based on user input. It is very specific to the purpose of finding a can on a table rather than multiple different types of objects, but for our purposes this should allow us to reach our goal of detecting a can on a table with Abby.