Implementation of 3D Object Recognition Based on Correspondence Grouping Final Report

Jamison Miles, Kevin Sheng, Jeffrey Huang

May 15, 2017

Instructor: Jivko Sinapov

I. Abstract

Currently, the Segbots in the BWI lab lack the capability of object recognition even with additional input from the user. The purpose of our project, therefore, is to implement 3D object recognition by working with point cloud data and correspondence grouping by giving the Segbot PCD files containing models of the objects we want recognized as input and having it return whether there was a match as a result of the point-to-point correspondences. In order to achieve this task, we developed an algorithm that utilized PCL functions to create keypoints found in the scene seen by the Segbot and compared them to keypoints on models that were also in the scene. In this paper, we outline the details of the 3D correspondence grouping algorithm, the associated tests and results, and our hopes for future development and research regarding it.

II. Introduction and Problem Formulation

The Segbots have a camera (a Kinect for some) that can be used for many purposes. These include self-localization on a map, detecting certain colors, and other functions. However, one of the biggest flaws of the current robot was the lack of ability for the robot to recognize objects. It saw them only as numbers that needed to be translated into an image. Thus, we attempted to solve this problem with pre-rendered images of 3D objects and give the robot the ability to see and “recognize” the object. There are many applications to this project. It could also be used to teach people the names of certain objects. Someone who wants to learn English could use one of these robots to learn the names of certain objects from the robot. However, the most possible application of this program is that it provides a good starting ground for possible future robot developments. If the robot can recognize a basketball, it could show on screen the word “basketball” or just have the robot “know” it is a basketball and determine another action such as “throw it”. The task is to give the robot the ability to “see” a 3D object, have it search for it in its memory and actually return what the object is onscreen. For example, if we added a basketball 3D object in the robot’s memory, the robot should have the ability to see, recognize the basketball, and show on-screen the “basketball” being displayed.

III. Related Work

We reviewed the literature on 3D object recognition to help us with our project. Some of them were helpful enough that we incorporated their concepts and some we used to get a better idea of the different ways this feat could be achieved. Many articles that we found interesting but did not incorporate into our final project included edge detection, which involved involved looking for “a significant local change in the image intensity” [1]. Edge detection had many issues such as false positives, where objects seemingly blended into the background, and inaccuracies at times, so this idea was scrapped. Some other ideas we got from scientific articles but never incorporated into the final version included the attempt to find objects via color. [2] Such an approach involves taking into account the RGB value, saturation, hue and lighting on the object for possible identification from a database of images. This approach contained errors with reading RGB values from the PCD files as shown by the colorless pictures below of the PointCloud2 data, so ultimately this approach was not implemented. Another concern with the color approach was a practicality. This approach could most likely identify basketballs quite well, as it only came in one form. It would not work well on objects such as hats that could come in multiple colors. It was obvious that if we took a color approach, we could not base objects based on color alone and also had to implement shape recognition. There was even more complex ideas such as local-scale invariant features. [3] An example might be a simple picture of an airplane in a background of a blue sky. By being able to pick out a very small portion of the airplane such as the wings, the computer might be able to deduce the object. Due to a potential for a high margin of error with this approach, this idea was also scrapped.

Thus, due to the flaws of the above, we decided to use PointCloud2 data and keypoint recognition as the basis of the project instead. Although it is not possible to have a 0% margin of error, this approached seemed to be the most likely path to reduce the margin of error. One article that we followed a bit more closely was the concept of pose clustering. [4] Pose clustering is selecting points based on what the camera reads and seeing if clustering them together could make a recognizable object. Although our method was not 100% pose clustering, it helped form the framework for our project for retrieving keypoints for object recognition. Our approach was different because we used PointCloud2 data and converted them to PCD files. Ultimately, we did use some of the ideas about pose clustering but with more of a correspondence grouping approach and used altogether a different method for 3D object recognition given by the PCL library, which also involved 3D models captured by either utilizing PCD files from online databases, such as the PCL website, or extracting our own Euclidean clusters. We felt this method had much more practical use outside of a controlled setting.

IV. **Technical Approach**

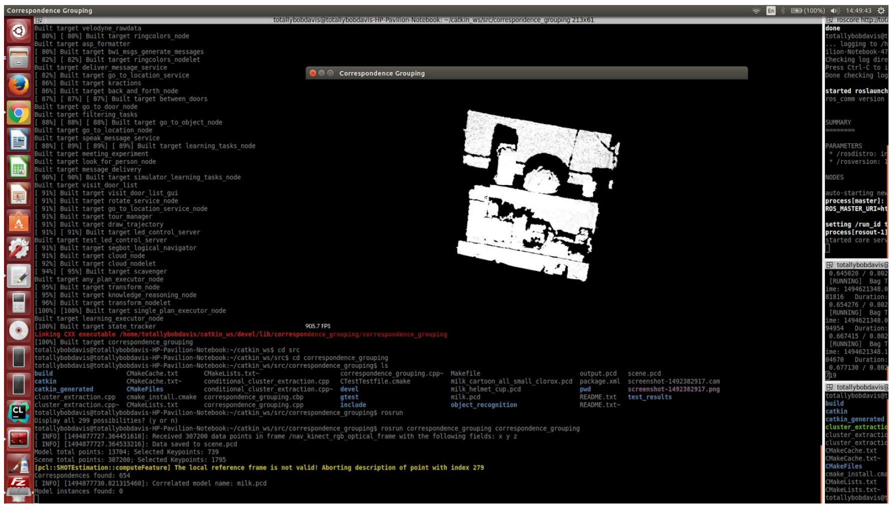
The first step to approaching this problem was to read and understand the documentation of the algorithm provided by the Point Cloud Library tutorials. In the beginning, we misread how to program the PCL functions and did not understand what a PCD file was, believing it to be like a 3D model render made in an program like Blender. It is nothing like that at all. A PCD is a point cloud data file that is made up of many keypoints. To utilize PCD files, we would need to develop an algorithm that would take in as input PCD files that consisted of the test models were looking for in the scene. Then, the algorithm had to be able to compare the robot’s current vision to the keypoints in the PCD file stored on the robot. We needed to subscribe to one of the topics on the Kinect in order to retrieve the right form of data for our algorithm to take in.

In order to integrate ROS with the PCL functions necessary for the algorithm to work, we realized that we would need a way to retrieve PointCloud data from the Kinect. By subscribing to **/nav_kinect/depth_registered/points**, we were able to retrieve the PointCloud2 data directly from the Kinect, then store it as a PCD file for later. After we were able to retrieve the data, we had to make sure that the program was reading the keypoints properly. Using **pcl_viewer**, a visualization of the scene being recorded was shown and allowed us to verify that the scene was correct. [Figure 2] From there, we could add in PCD files containing the models we wanted to search for in the PCD file containing the scene. For the PCD models, we utilized some from online databases while also extracting our own Euclidean clusters to form our own PCD models, though each individual model tended to be separated into numerous PCD files that contained parts of the model's keypoints, but not all together. [Figure 3] As a result, we utilized PCD files more from online databases. In addition, the orientations of the test models was always the front of the objects due to the limitation of our PCD files, which were extracted from a single-point perspective (the Kinect), and therefore only had one view to see the models: the front.

The 3D correspondence grouping algorithm had a number of functions that allowed it to process the PCD files it was given. First, the PCD models were placed in the **catkin_pkg** directory, and the command to **rosrun** the algorithm would be run from that directory to allow access to the models. With the PCD input models, normals of both that and the scene were computed using the PCL function **NormalEstimationOMP**, which estimated local surface properties at each 3D point. The PointCloud2 data retrieved from the PCDs of the model and scene was then downsampled through the PCL function **UniformSampling** (which forms a 3D grid over the PointCloud2 data) to find keypoints that would be given a 3D descriptor to allow for point-to-point correspondences through the PCL function **SHOTEstimationOMP**. To determine point-to-point correspondence, the PCL function **KdTreeFLANN** was used to compare keypoints in the scene to keypoints in the model in order to find the most similar keypoint based on Euclidian distance, and placed into a vector of correspondences. Clustering of correspondences was done with the PCL function **Hough3DGrouping** (which forced the algorithm to create its own local reference frames for each keypoint), though the PCL function **GeometricConsistencyGrouping** was used as an alternative from time to time. In addition, we created two thresholds to determine the status of a model in a scene; These two thresholds defined which models had correlation and which models had full model recognition. For correlation, the minimum correspondences necessary was set to 500 correspondences: this was due to the fact that the test objects we worked with had, on average, 1000 key points to work with. If half of those keypoints managed to be grouped, then there was proof that a version of the model did exist in the scene. On the other hand, full model recognition required nearly one hundred percent exact keypoint to keypoint correspondence: this was to assure that the model recognized was without a shadow of a doubt the input model.

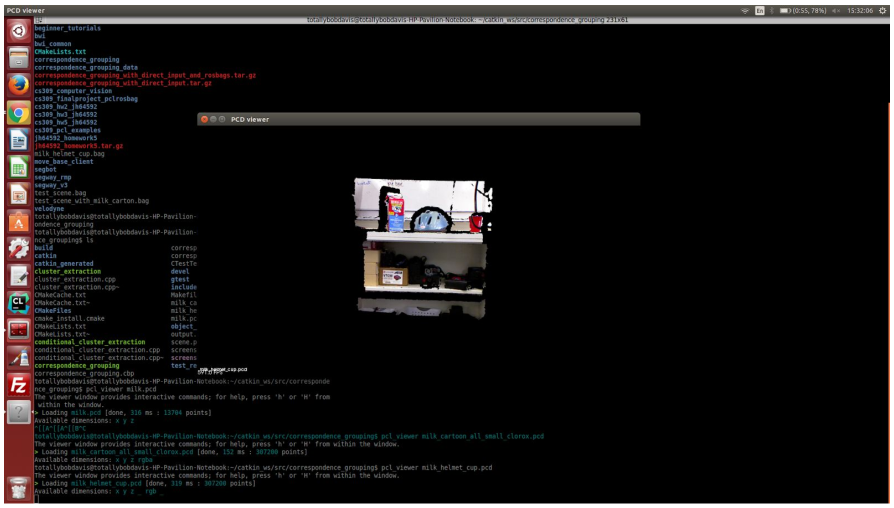
V. Evaluation and Example Demonstration

In order to evaluate the correctness of our algorithm, we focused on having the Segbot recognize three simple objects: a milk carton, a bike helmet, and a red plastic cup. The main test scene was the BWI lab room where we placed the test objects for easy access to the v2 Segbot without having to maneuver it to a different area. Among the first many tests we ran was the capturing of the scene, and then immediately using that captured PCD scene as a model to determine if the correspondence grouping algorithm was at least recording the scene successfully. That test was immediately successful: the scene itself was recognized as a full model recognition, which meant that the algorithm was certain that the input file was the model in the scene. Afterwards, the test objects were individually placed onto a table and their combined Euclidean clusters were sent in as a single PCD input file. Unfortunately, the results did not lead to full model recognitions. The combined models were only seen as "correlated", meaning that the algorithm believed that there were enough correspondences to assume the model was there, but not enough to be considered as a full model recognition. The same results happened even after the models' Euclidean clusters were separated into individual PCD models and sent in one after the other. Throughout all of this, the scene was constant: there were no changes to the orientations or positions of the test objects. The lighting did not seem to affect the algorithm: this was due to a bug that resulted in the algorithm being unable to take in RGB values. [Figure 1] To bypass this, the algorithm used geometric consistency for correspondence grouping. Ultimately, we came to a conclusion from our test results that although our 3D correspondence grouping algorithm was able to understand that input PCD models also in the scene were present, it failed to realize that the two models were one in the same nearly all of the time.



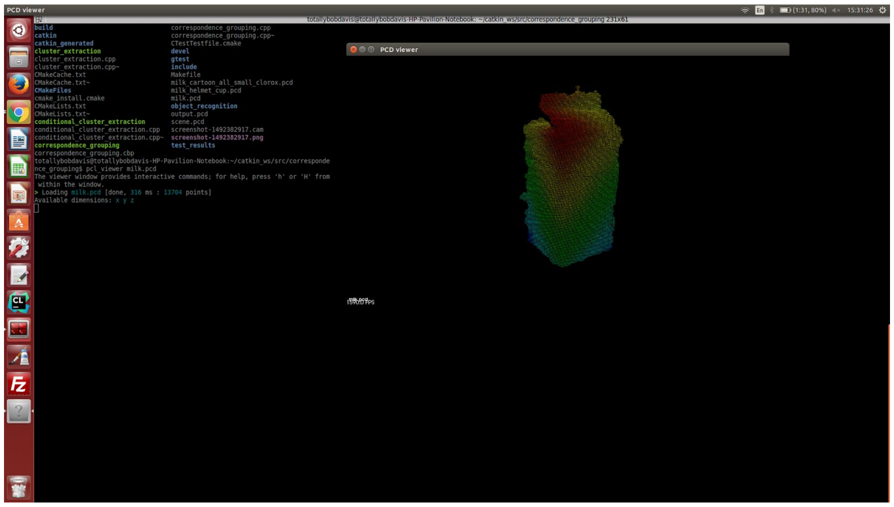
```
Built target velodyne_rawdata
[ 80%] [ 80%] Built target ringcolors_node
Built target asp_formatter
[ 80%] Built target bwi_msgs_generate_messages
[ 82%] [ 82%] Built target ringcolors_nodelet
Built target deliver_message_service
[ 82%] Built target go_to_location_service
[ 86%] Built target kractions
[ 86%] Built target back_and_forth_node
[ 87%] [ 87%] [ 87%] Built target between_doors
Built target go_to_door_node
Built target filtering_tasks
[ 88%] [ 88%] [ 88%] Built target go_to_object_node
Built target go_to_location_node
Built target speak_message_service
[ 88%] [ 89%] [ 89%] [ 89%] Built target learning_tasks_node
Built target meeting_experiment
Built target look_for_person_node
Built target message_delivery
[ 90%] [ 90%] Built target simulator_learning_tasks_node
Built target visit_door_list
[ 91%] Built target visit_door_list_gui
[ 91%] Built target rotate_service_node
[ 91%] Built target go_to_location_service_node
[ 91%] Built target tour_manager
[ 91%] Built target draw_trajectory
[ 91%] [ 91%] Built target led_control_server
Built target test_led_control_server
[ 91%] Built target segbot_logical_navigator
[ 91%] Built target cloud_node
[ 92%] Built target cloud_nodelet
[ 94%] [ 95%] Built target scavenger
Built target any_plan_executor_node
[ 95%] Built target transform_node
[ 95%] Built target knowledge_reasoning_node
[ 96%] Built target transform_nodelet
[100%] [100%] Built target single_plan_executor_node
Built target learning_executor_node
[100%] Built target state_tracker
Linking CXX executable /home/totallybobdavis/catkin_ws/devel/lib/correspondence_grouping/correspondence_grouping
[100%] Built target correspondence_grouping
totallybobdavis@totallybobdavis-HP-Pavilion-Notebook:~/catkin_ws$ cd src
totallybobdavis@totallybobdavis-HP-Pavilion-Notebook:~/catkin_ws/src$ cd correspondence_grouping
totallybobdavis@totallybobdavis-HP-Pavilion-Notebook:~/catkin_ws/src/correspondence_grouping$ ls
build                    CMakeCache.txt       CMakeLists.txt~                     correspondence_grouping.cpp~   Makefile                            output.pcd    scene.pcd
catkin                   CMakeCache.txt~      conditional_cluster_extraction.cpp  CTestTestfile.cmake            milk_cartoon_all_small_clorox.pcd   package.xml   screenshot-1492382917.cam
catkin_generated         CMakeFiles           conditional_cluster_extraction.cpp~ devel                          milk_helmet_cup.pcd                 pwd           screenshot-1492382917.png
cluster_extraction.cpp   cmake_install.cmake  correspondence_grouping.cbp         gtest                          milk.pcd                            README.txt    test_results
cluster_extraction.cpp~  CMakeLists.txt       correspondence_grouping.cpp         include                        object_recognition                  README.txt~
totallybobdavis@totallybobdavis-HP-Pavilion-Notebook:~/catkin_ws/src/correspondence_grouping$ rosrun
Display all 299 possibilities? (y or n)
totallybobdavis@totallybobdavis-HP-Pavilion-Notebook:~/catkin_ws/src/correspondence_grouping$ rosrun correspondence_grouping correspondence_grouping
[ INFO] [1494877727.364451618]: Received 307200 data points in frame /nav_kinect_rgb_optical_frame with the following fields: x y z
[ INFO] [1494877727.364533216]: Data saved to scene.pcd
Model total points: 13704; Selected Keypoints: 739
Scene total points: 307200; Selected Keypoints: 1795
[pcl::SHOTEstimation::computeFeature] The local reference frame is not valid! Aborting description of point with index 279
Correspondences found: 654
[ INFO] [1494877730.821315460]: Correlated model name: milk.pcd
Model instances found: 0
```

[Figure 1]
A visualization of the scene along with the result of the correspondence grouping (the lack of RGB properties made it so white meant key points not fully recognized and red meant fully recognized models)



```
beginner_tutorials
bwi
bwi_common
CMakeLists.txt
correspondence_grouping
correspondence_grouping_data
correspondence_grouping_with_direct_input_and_rosbags.tar.gz
correspondence_grouping_with_direct_input.tar.gz
cs309_computer_vision
cs309_finalproject_pclrosbag
cs309_hw2_jh64592
cs309_hw3_jh64592
cs309_hw5_jh64592
cs309_pcl_examples
jh64592_homework5
jh64592_homework5.tar.gz
milk_helmet_cup.bag
move_base_client
segbot
segway_rmp
segway_v3
test_scene.bag
test_scene_with_milk_carton.bag
velodyne
totallybobdavis@totallybobdavis-HP-Pavilion-
ondence_grouping
totallybobdavis@totallybobdavis-HP-Pavilion-
nce_grouping$ ls
build                                 corresp
catkin                                corresp
catkin_generated                      CTestTe
cluster_extraction                    devel
cluster_extraction.cpp                gtest
cluster_extraction.cpp~               include
CMakeCache.txt                        Makefil
CMakeCache.txt~                       milk_ca
CMakeFiles                            milk_he
cmake_install.cmake                   milk.pc
CMakeLists.txt                        object_
CMakeLists.txt~                       output.
conditional_cluster_extraction        scene.p
conditional_cluster_extraction.cpp    screens
conditional_cluster_extraction.cpp~   screens
correspondence_grouping               test_re
correspondence_grouping.cbp
totallybobdavis@totallybobdavis-HP-Pavilion-Notebook:~/catkin_ws/src/corresponde
nce_grouping$ pcl_viewer milk.pcd
The viewer window provides interactive commands; for help, press 'h' or 'H' from
 within the window.
> Loading milk.pcd [done, 316 ms : 13704 points]
Available dimensions: x y z
^[[A^[[A^[[B^C
totallybobdavis@totallybobdavis-HP-Pavilion-Notebook:~/catkin_ws/src/correspondence_grouping$ pcl_viewer milk_cartoon_all_small_clorox.pcd
The viewer window provides interactive commands; for help, press 'h' or 'H' from within the window.
> Loading milk_cartoon_all_small_clorox.pcd [done, 152 ms : 307200 points]
Available dimensions: x y z rgba
totallybobdavis@totallybobdavis-HP-Pavilion-Notebook:~/catkin_ws/src/correspondence_grouping$ pcl_viewer milk_helmet_cup.pcd
The viewer window provides interactive commands; for help, press 'h' or 'H' from within the window.
> Loading milk_helmet_cup.pcd [done, 319 ms : 307200 points]
Available dimensions: x y z _ rgb _
```

[Figure 2]
A PCD file of the scene with a milk carton, a bike helmet, and a red plastic cup. These are the three models that are being tested for correspondence grouping by the PCL algorithm.

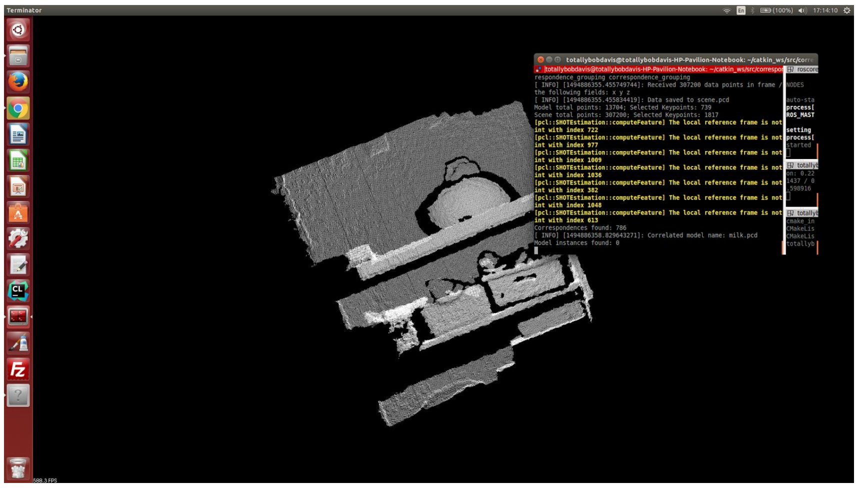


```
build                                 correspondence_grouping.cpp
catkin                                correspondence_grouping.cpp~
catkin_generated                      CTestTestfile.cmake
cluster_extraction                    devel
cluster_extraction.cpp                gtest
cluster_extraction.cpp~               include
CMakeCache.txt                        Makefile
CMakeCache.txt~                       milk_cartoon_all_small_clorox.pcd
CMakeFiles                            milk_helmet_cup.pcd
cmake_install.cmake                   milk.pcd
CMakeLists.txt                        object_recognition
CMakeLists.txt~                       output.pcd
conditional_cluster_extraction        scene.pcd
conditional_cluster_extraction.cpp    screenshot-1492382917.cam
conditional_cluster_extraction.cpp~   screenshot-1492382917.png
correspondence_grouping               test_results
correspondence_grouping.cbp
totallybobdavis@totallybobdavis-HP-Pavilion-Notebook:~/catkin_ws/src/corresponde
nce_grouping$ pcl_viewer milk.pcd
The viewer window provides interactive commands; for help, press 'h' or 'H' from
 within the window.
> Loading milk.pcd [done, 316 ms : 13704 points]
Available dimensions: x y z
```

[Figure 3]
A PCD file containing a milk carton's key points by extracting its Euclidian cluster. Notice how the keypoints are shaded a different color: the difference in colors represents the distance from the camera.

VI. Conclusion and Future Work

In conclusion, we were able to have the robot successfully identify the desired objects with high accuracy and consistency. The robot was able to identify a bike helmet, a red plastic cup, and a milk carton in a cluttered scene. However, there is much to be done in terms of future work. We seem to have a bug in which the program is not able to read-in RGB values. This does not affect the program at all, but is simply a portion of the program that could use some polishing. In addition, we will need to improve on the correspondence grouping as keypoints of one model may be mistaken for another model's, possibly resulting in misassociation. [Figure 4]



```
totallybobdavis@totallybobdavis-HP-Pavilion-Notebook: ~/catkin_ws/src/corre
respondence_grouping correspondence_grouping
[ INFO] [1494886355.455749744]: Received 307200 data points in frame /
the following fields: x y z
[ INFO] [1494886355.455834419]: Data saved to scene.pcd
Model total points: 13704; Selected Keypoints: 739
Scene total points: 307200; Selected Keypoints: 1817
[pcl::SHOTEstimation::computeFeature] The local reference frame is not
int with index 722
[pcl::SHOTEstimation::computeFeature] The local reference frame is not
int with index 977
[pcl::SHOTEstimation::computeFeature] The local reference frame is not
int with index 1009
[pcl::SHOTEstimation::computeFeature] The local reference frame is not
int with index 1036
[pcl::SHOTEstimation::computeFeature] The local reference frame is not
int with index 382
[pcl::SHOTEstimation::computeFeature] The local reference frame is not
int with index 1048
[pcl::SHOTEstimation::computeFeature] The local reference frame is not
int with index 613
Correspondences found: 786
[ INFO] [1494886358.829643271]: Correlated model name: milk.pcd
Model instances found: 0
```

[Figure 4]
A PCD file containing a milk carton with the bike helmet in front of it. The correspondences of the helmet have been mistaken for the milk carton's, resulting in the algorithm believing that there are more correspondences for the carton than there should be.

There is also room for future improvement in terms of communicating that the object has been successfully identified. For now, there is only an output to the terminal that states that the object has been identified. It could be easier to communicate that the object has been discovered with flashing LEDs. Another possible improvement would be to make sure this program works with the V3's. Currently our program subscribes to the Kinect, and we are not sure if the V3's camera uses the same subscriber as the kinect does. If it does not, then we will have to improve our program by including the correct subscriber. Another improvement, and possibly the most important, would be to have our ROS node return the x, y, and z coordinates of the identified object. This would allow robots to know the exact location of the identified object. This way, if someone wanted a robot with an arm to pick up a specific object. Then the robot could pick up the object that it was specifically requested, and have the robot identify the object itself. Then it would pick it up with the returned coordinates after the identification.

VII. **Works Cited**

[1] Ramadevi, Y., T. Sridevi, B. Poornima, and B. Kalyani. "Segmentation And Object Recognition Using Edge Detection Techniques." International Journal of Computer Science and Information Technology 2, no. 6 (2010): 153-61.

[2] Gevers, Theo, and Arnold W.m. Smeulders. "Color-based object recognition." Pattern Recognition 32, no. 3 (1999): 453-64.

[3] Lowe, D.g. "Object recognition from local scale-invariant features." Proceedings of the Seventh IEEE International Conference on Computer Vision, 1999.

[4] Stockman, George. "Object recognition and localization via pose clustering." Computer Vision, Graphics, and Image Processing 39, no. 3 (1987): 393.