OBJECT DISCOVERY WITH A MICROSOFT KINECT

A Major Qualifying Project Report

submitted to the Faculty

of the

WORCESTER POLYTECHNIC INSTITUTE

in partial fulfillment of the requirements for the

Degree of Bachelor of Science

by

Paul Malmsten

Date: December 15, 2012

Approved By:

Professor Sonia Chernova, Advisor

Professor Charles Rich, Advisor

1 Abstract

I introduce a simple yet robust algorithm for discovering objects and level planes from depth information produced by a Microsoft Kinect. In a controlled indoor environment, this algorithm discovers objects on multiple surfaces at varying heights and distinguishes between separate objects on the same surface. The algorithm includes eliminating large planes from the input, clustering objects by spatial proximity, and by filtering results using domain knowledge of the environment.

2 Acknowledgements

I would like to thank my colleagues Richard Kelly, Tim Jenkel, and Paul Shepanski who worked alongside me throughout this project. Their ideas, advice, and contributions to the background section of my report are much appreciated.

I would also like to thank my advisers Sonia Chernova and Charles Rich for their support and direction. It was a pleasure working with them, and I am grateful for the opportunity to learn about computer vision, artificial intelligence, and robotics.

Finally, I wish to thank my mentor Russel Toris. His experience and advice were invaluable while learning to use ROS and our robot.

3 Table of Contents

1 Abstract 2
2 Acknowledgements 3
4 Table of Figures 5
5 Introduction 6
6 Background 7
6.1 Tools 7
6.1.1 KUKA youBot 7
6.1.2 Robot Operating System (ROS) 8
6.1.3 Gazebo Simulator 10
6.1.4 Vision 12
6.1.5 Arm 15
6.2 Algorithms 17
6.2.1 Vision / Point Cloud Processing 17
7 Objectives 21
8 Methodology 22
8.1 Downsampling the Input Cloud 24
8.2 Plane Extraction 25
8.3 Point Clustering with a Kd-Tree 27
8.4 Filtering Results by Measured Attributes 28
9 Results 31
10 Limitations and Future Work 42
11 Conclusion 43
12 Bibliography 44

4 Table of Figures

Figure 1: Laboratory Environment with IKEA Children's Furniture [1] 6
Figure 2: The KUKA youBot 7
Figure 3: Dimensions of the KUKA youBot arm [2] 8
Figure 4: Graph of ROS application 9
Figure 5: Gazebo Simulation of youBot Environment 11
Figure 6: The Microsoft Kinect [5] 12
Figure 7: Microsoft Kinect IR Projection [6] 13
Figure 8: Result of PCL cluster extraction [11] 15
Figure 9: Voxel Grid Example [14] 18
Figure 10: Example k-dimensional tree, where k = 2 [17] 19
Figure 11: RGB View of Example Environment 23
Figure 12: RGBD View of Example Environment 24
Figure 13: Downsampled RGBD Capture of Example Environment, Dilated 25
Figure 14: Discovered Planes, Artificial Color, Dilated 26
Figure 15: Downsampled Cloud of Example Environment with Planes Removed, Artificial Color, Dilated 26
Figure 16: Clustered Objects from Example Environment, Artificial Color, Dilated 27
Figure 17: Clustered Objects from Example Environment, Real Color, Dilated 28
Figure 18: Objects with a Spherical Bounding Radius in [3cm, 5cm] from Example Environment, Real Color, Dilated 29
Figure 19: Level Planes, Artificial Color, Dilated 30
Figure 20: RGBD View of Example Environment 31
Figure 21: Objects with a Spherical Bounding Radius in [3cm, 5cm] from Example Environment, Real Color, Dilated 32
Figure 22: Level Planes, Artificial Color, Dilated 32
Figure 23: RGBD View of Three Cups on Small Table 33
Figure 24: Objects in Three Cups on Small Table Example, Artificial Color, Dilated 34
Figure 25: Three Cup Object Point Clouds, View Moved Forward and Tilted Down, Artificial Color, Dilated 35
Figure 26: Three Cup Example Objects with Minimum Bounding Sphere Radius in [5cm, 13cm], Artificial Color, Dilated 36
Figure 27: Three Cup Example Level Planes, Artificial Color, Dilated 37
Figure 28: RGBD View of WPI Interaction Lab 38
Figure 29: Objects in Interaction Lab, Artificial Color, Dilated 39
Figure 30: Objects in Interaction Lab with Minimum Bounding Sphere Radius in [3cm, 8cm], Artificial Color, Dilated 40
Figure 31: Level Planes in Interaction Lab, Artificial Color, Dilated 41

5 Introduction

This project began as a component of an existing research project whose goal is to provide an always-on robotics laboratory available remotely via the internet. This larger project aims to expand the population that researchers may leverage during human-robot interaction studies by allowing users to participate over the internet. The proposed laboratory includes a robot in a small room with a collection of objects it can manipulate, as well as a corresponding simulation environment. A photo of this is environment is included in Figure 1.

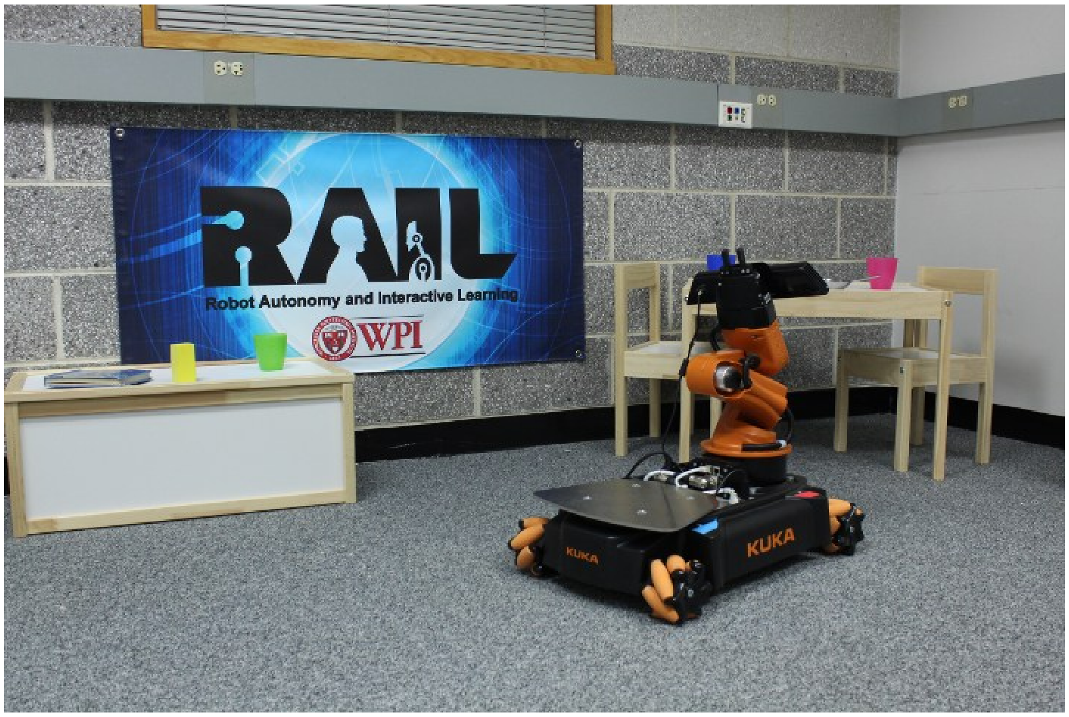


Figure 1: Laboratory Environment with IKEA Children's Furniture [1]

To provide a rich user experience for remote users such that they can effectively manipulate objects remotely, a need was defined for the robot to be able to discover and localize objects in the environment. After evaluating possible sensors for this purpose, it was decided that a Microsoft Kinect would be used to measure the topology and color of the environment.

This project addressed the need for software that can effectively discover objects given the information collected from a Microsoft Kinect.

6 Background

To achieve our objectives, we built upon a collection of existing knowledge and implementation effort provided by the robotics community. In the sections to follow, we will briefly introduce the concepts and tools that the reader should be familiar with. These have been organized by two overarching themes:

- Tools: existing hardware and software that we investigated for this project
- Algorithms: the concepts employed by our software and how it produces useful results

6.1 Tools

In this section, we will briefly introduce each of the tools that we researched throughout the project:

- The KUKA youBot, a 4-wheel mecanum drive robot with a small industrial manipulator
- Robot Operating System (ROS), a robot software development framework and library collection
- The Gazebo simulation environment
- Tools related to vision processing and object detection
- Tools related to arm operation and object manipulation

Each of these will be discussed in the sections below.

6.1.1 KUKA youBot

We are using the KUKA youBot mobile manipulator, as shown in Figure 2. The KUKA youBot base uses an omnidirectional drive system with mecanum wheels. Unlike standard wheels, mecanum wheels consist of a series of rollers mounted at a 45° angle. This allows the robot to move in any direction, including sideways, which makes the robot much more maneuverable in tight areas.

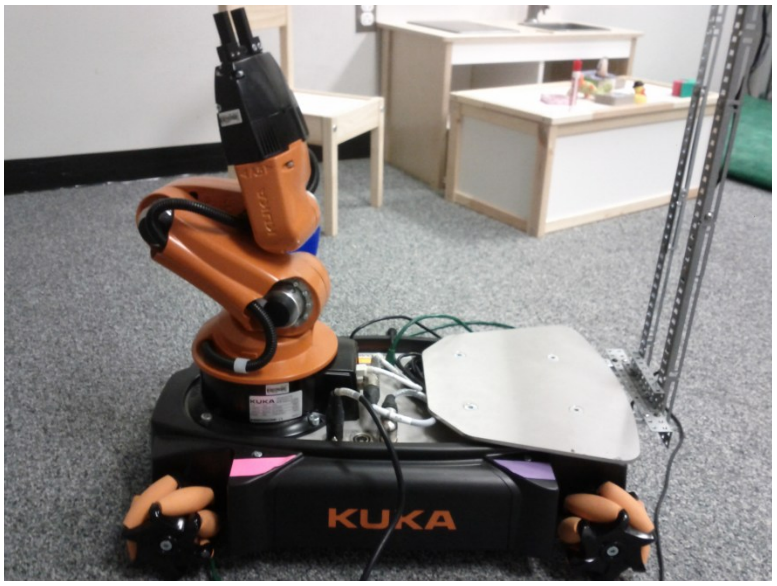


Figure 2: The KUKA youBot

The KUKA youBot is controlled by an onboard computer running a version of Ubuntu Linux. The youBot's onboard computer has many of the features of a standard computer, including a VGA port to connect an external monitor, several USB ports for connecting sensors and other peripherals, and an Ethernet port for connecting the youBot to a network.

Connected to the base of the youBot is a 5 degree-of-freedom arm. The arm is a 5 link serial kinematic chain with all revolute joints. The arm is sturdy and non-compliant, similar to KUKA's larger industrial robot arms. The dimensions of each link of the arm, as well as the range of each joint are shown in Figure 3 below. The rotation of each joint in the youBot's arm is measured by a relative encoder; therefore, users must manually move the arm to a home position before the arm is initialized. The youBot's wrist is equipped with a two finger parallel gripper with a 2.3 cm stroke. There are multiple mounting points for the gripper fingers that users can choose based on the size of the objects to pick up.

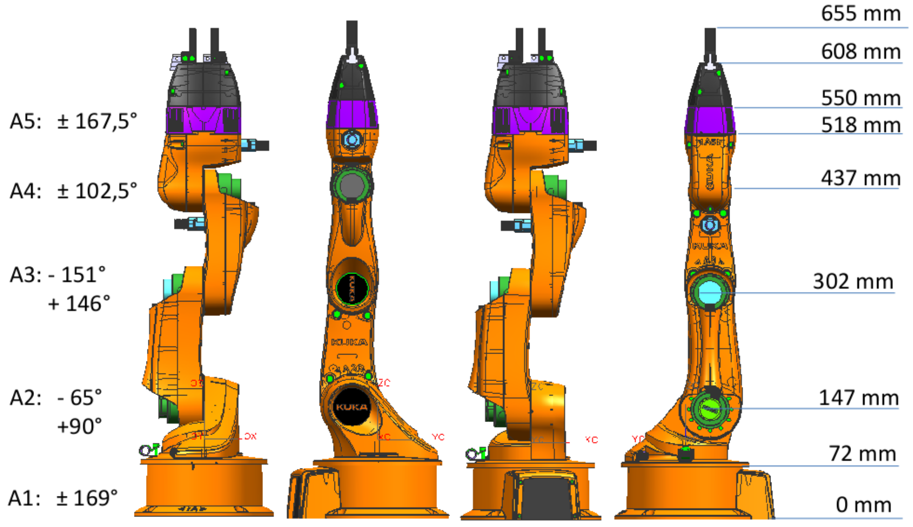


Figure 3: Dimensions of the KUKA youBot arm [2]

6.1.2 Robot Operating System (ROS)

ROS (Robot Operating System) is an open source software framework for robotic development. The primary goal of ROS is to provide a common platform to make the construction of capable robotic applications quicker and easier. Some of the features it provides include hardware abstraction, device drivers, message-passing, and package management [3]. ROS was originally developed starting in 2007 under the name Switchyard by the Stanford Artificial Intelligence Laboratory, but since 2008 it has been primarily developed by Willow Garage and is currently in its sixth release.

The fundamental purpose of ROS is to provide an extensible interprocess communication framework which simplifies the design of distributed systems. The building blocks of a ROS application are *nodes*; a node is a named entity that can communicate with other ROS nodes on behalf of an operating system process. At this time, ROS provides support for nodes written in C++ and Python, and experimental libraries exist for a handful of other languages.

There are three ways that nodes may communicate in a ROS environment:

1. By publishing *messages* to a *topic*.
2. By listening to messages published on a topic.
3. By calling a *service* provided by another node.

Messages represent data structures that may be transferred between nodes. Messages may contain any named fields; each field may be a primitive data type (e.g. integers, booleans, floating-point numbers, or strings), a message type, or an array of either type. ROS provides a mechanism for generating source code from text definitions of messages.

ROS *topics* represent named channels over which a particular type of message may be transferred. A topic provides one-directional communication between any number publishing and consuming nodes. Figure 4 provides a visualization of a ROS graph; ovals represent nodes, and the directed arrows represent publish/subscribe relationships between nodes via a particular topic.

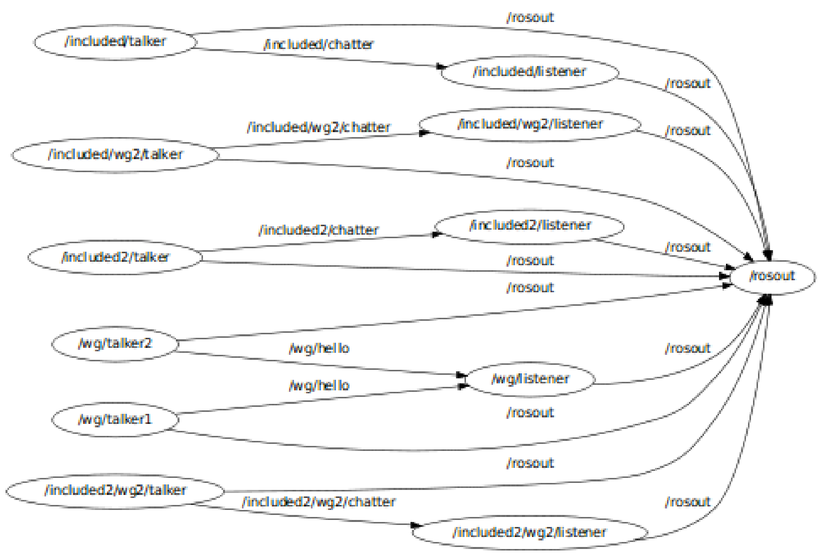


Figure 4: Graph of ROS application

A node that publishes to a topic is called a *publisher*. Publishers often run continuously in order to provide sensor readings or other periodic information to other nodes; however, it is possible to write a publisher that publishes a message only once.

A node which listens to messages on a topic is called a *subscriber*. A subscriber specifies which topic it wants to listen to as well as the expected message type for the topic, and registers a callback function to be executed whenever a message is received. Similar to publishing a single message, a subscriber may instead block until a message is received if only a single message is required.

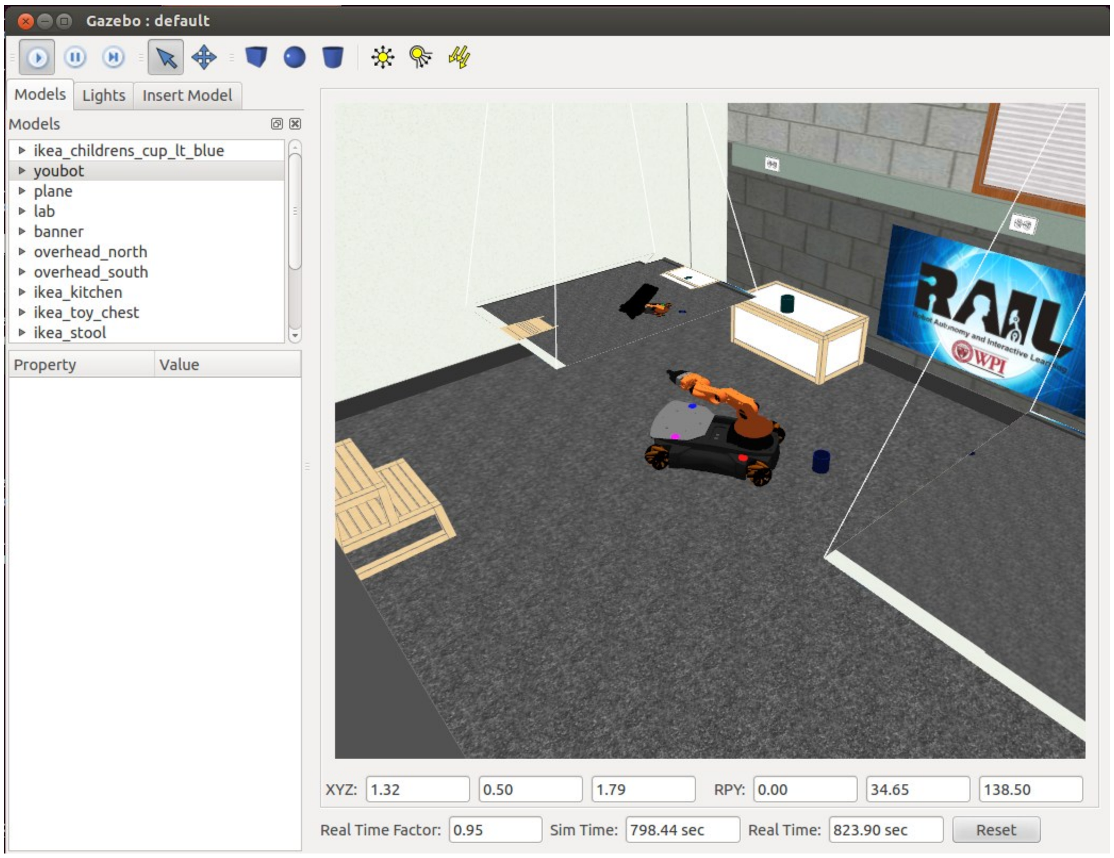
Finally, a node may provide a *service* to other nodes. A service call in ROS resembles a remote procedure call workflow: a *client* node sends a request to the service provider node, a registered callback function in the service provider node performs the appropriate action(s), and then the service provider sends a response back to the client.

Since a service call is an exchange between one client node and one service provider node, they do not employ topics. Service request and response messages, however, are defined in a similar manner as standard messages, and they may include standard messages as fields.

Any node can perform any combination of these actions; for example, a node could subscribe to a topic, call a service on a message it receives, and then publish the result to another topic. This allows a large amount of flexibility in ROS applications. ROS also allows applications to be distributed across multiple machines, with the only restriction that nodes which require hardware resources must run on a machine where those resources are available. More information about ROS including tutorials, instillation instructions, and package information can be found on their website: www.ros.org/wiki.

6.1.3 Gazebo Simulator

The Gazebo simulator [4] is a multi-robot simulator, primarily designed for outdoor environments. The system is compatible with ROS, making it a good choice for representing the robot's environment. Gazebo also features rigid body physics simulation, allowing for collision detection and object manipulation. Finally, Gazebo allows for the simulation of robot sensors, allowing us to incorporate the Kinect's full functionality and the overhead cameras into the simulation.



XYZ: 1.32 0.50 1.79 RPY: 0.00 34.65 138.50

Real Time Factor: 0.95 Sim Time: 798.44 sec Real Time: 823.90 sec

Figure 5: Gazebo Simulation of youBot Environment

The room as it is envisioned in the final product is modeled in the Gazebo simulation. An example image of the simulation is included below. The flat images projected in the air represent the overhead cameras' point of view, which is used in the web interface. Both the robot and arm are fully represented and capable of being manipulated within the simulation. All of the objects that the robot is expected to interact with in the room are also present and can be added and moved. Some of the more obvious objects present:

- IKEA table
- IKEA chest
- Plastic cups
- Stool

As we determine what objects should be present in the environment based upon the design decisions that we make and the experience that we gain working with the robot, the gazebo simulation and the objects included have to be updated to reflect the changes.

6.1.4 Vision

We investigated a handful of tools to help us discover objects in our robot's environment and produce suitable representations of them in software. These include:

- The Microsoft Kinect, a structured-light 3D camera designed for the Xbox game console.
- The Object Recognition Kitchen (ORK), an effort by Willow Garage to develop a general-purpose object detection and recognition library for ROS.
- The Tabletop Object Detector, an object detection and recognition library for objects on tables
- The Pont Cloud Library (PCL), a general purpose library for working with point cloud data structures

Each of these will be introduced in the following sections.

6.1.4.1 Microsoft Kinect

The Microsoft Kinect is a consumer device originally designed by Microsoft as a peripheral for the Xbox game system. It was designed to compete against the motion-sensitive controller introduced by Nintendo for the Wii game system.

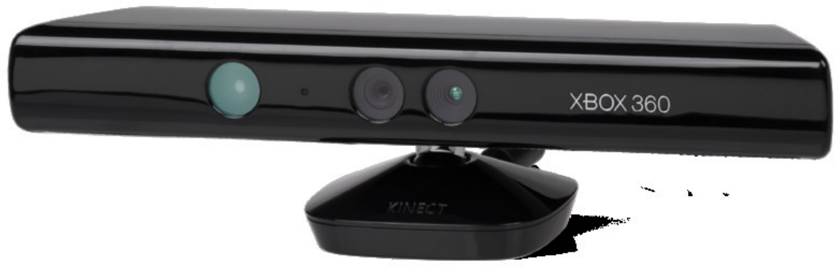


Figure 6: The Microsoft Kinect [5]

The Microsoft Kinect is composite device which includes the following components:

- A color digital camera
- A structured-light infrared projector
- An infrared digital camera
- A microphone array
- A tilt motor, which can adjust the pitch of the attached cameras and projector

The Kinect derives depth information from an environment by projecting a grid of infrared points in a predictable pattern. The resulting projection on the environment is viewed by the integrated infrared camera and interpreted to produce depth information for each point. An example of this projection is illustrated in Figure 7.

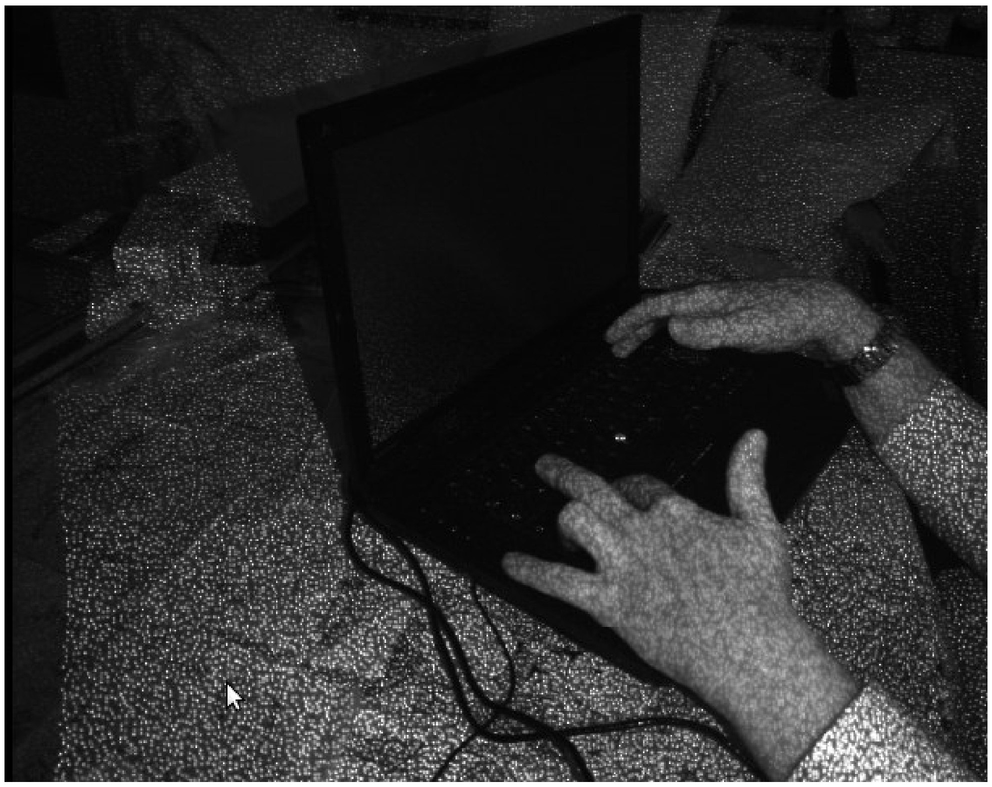


Figure 7: Microsoft Kinect IR Projection [6]

This structured light approach poses challenges for objects particularly near and objects particularly far from the sensor. For objects closer than 0.8 meters, the projected points appear too closely together for the sensor to measure; this results in a short-range blind spot that can have implications for where the sensor is mounted, particularly for small robots. For objects farther than 4 meters, the projected points fade into the background. This maximum range is also adversely affected by the presence of infrared noise in the environment. As such, the Kinect's range is significantly degraded outdoors.

To support game development, Microsoft has developed a software library for interpreting a human figure in this point cloud. This library allows users of the Microsoft tool chain for C++ and C# to easily discover human figures in an environment and measure determine the 3D position of the figure's extremities.

The impressive quality of the depth information produced by the Kinect and the Kinect's low price make it a very attractive sensor for robotics research. As a result, a variety of open-source drivers have been developed for the Kinect which allow one to process the information produced by a Kinect as a cloud of points located in 3D space. In addition, some of these libraries also provide depth registration, wherein each depth point is annotated with the RGB color of that point in the environment.

We have investigated the openni_camera [7] package for ROS. This package produces ROS point cloud message data structures which makes it easy to use a Kinect with many existing ROS packages and infrastructure. In addition, openni_camera also supports depth registration.

6.1.4.2 Object Recognition Kitchen

The Object Recognition Kitchen (ORK) is a tool chain for object recognition that is being developed by Willow Garage [8]. It is independent of ROS, although it provides an interface for working with ROS. The ORK is designed such that object recognition algorithms can be modularized; it implements a few different object recognition approaches, and provides an interface, called a pipeline, that new algorithms can conform to. Each pipeline has a source, where it gets data from, the actual image processing, and a sink, where the data is output. When performing object detection, the ORK can run multiple pipelines in parallel to improve results. The built-in pipelines are LINE-MOD, tabletop, TOD, and transparent objects. Tabletop is a ported version of the ROS tabletop object detector package. TOD stands for textured object detection and matches surfaces against a database of known textures. The transparent objects pipeline is similar to the tabletop object detector, but works on transparent objects such as plastic cups or glass.

We were not able to successfully test the ORK in our environment; it appears that the project is under active development, but not yet complete.

6.1.4.3 Tabletop Object Detector

The tabletop object detector is a software library originally written by Willow Garage for its flagship research robot, the PR2 [9]. The purpose of this package is to provide a means of recognizing simple household objects placed on a table such that they can be manipulated effectively.

Given a point cloud from a sensor, the tabletop object detector first discovers the surface of the table it is pointed at through a process called segmentation (Section 6.2.1.2). Once the table surface has been discovered, the algorithm filters the original point cloud to remove all of the points that do not lie directly above the surface of the table. Finally, the remaining points are clustered into discrete objects using nearest-neighbor clustering with a Kd-tree (Section 6.2.1.3).

This process produces a sequence of point clouds, one for each object. As an additional optional step, the tabletop object detector also provides rudimentary object recognition. Given a database of household object meshes to compare against, the tabletop object detector provides an implementation of iterative closest point (ICP), a relatively simple algorithm for registering a sensor point cloud against a model point cloud (Section 6.2.1.4). If a point cloud successfully compares against a model in the database, the tabletop object detector also provides the more detailed model mesh as a part of the detection result.

Although this package works well for the PR2 in a constrained environment, it has some noteworthy limitations:

- It cannot detect objects on the floor, because it expects the presence of a table plane.
- It is very sensitive to changes in perspective of an observed table.

During our testing, we found that our Microsoft Kinect sensor must be positioned at just the right height and angle with respect to our IKEA children's table in order for the tabletop object detector to properly detect the table. However, even when our table was successfully detected, the library was unable to detect the IKEA children's cups that we placed upon it.

It is worth noting that the tabletop object detector was originally designed to work with point clouds produced by computing the disparity between two cameras, and we chose to use a Microsoft Kinect instead; in addition, our IKEA children's table and children's cups are notably smaller than the standard rectangular table and household objects that the library was designed to work with. These factors likely influenced our results.

6.1.4.4 Point Cloud Library (PCL)

The Point Cloud Library (PCL) [10] is an open source project for image and point cloud processing. PCL was originally developed by Willow Garage as a package for ROS; however, its utility as a standalone library quickly became apparent. It is now a separate project maintained by the Open Perception Foundation, and is funded by many large organizations. The PCL is split into multiple libraries which can be compiled and used separately. These include libraries include support for: filtering, feature finding, key point finding, registration, kd-tree representation, octree representation, image segmentation, sample consensus, ranged images, file system I/O and visualization. This project relies on reading and writing point clouds to files, point cloud visualization, downsampling point clouds using a filter, plane segmentation, and object segmentation using Euclidean Cluster Extraction. Euclidean Cluster Extraction works by separating the points into groups where each member of a group is within a specified distance of at least one other member of the group. Figure 8 shows the result of plane segmentation removal and then Euclidean Cluster Extraction on a sample table scene. Note how the top of the table and floor have been removed and that different colored clouds represent separate clusters.

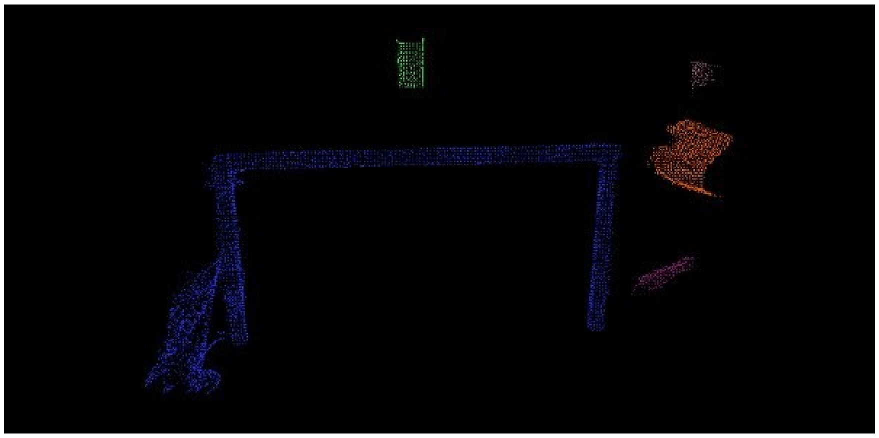


Figure 8: Result of PCL cluster extraction [11]

PCL also provides tutorials and examples for most of its features allowing easy implementation and modification of the PCL algorithms. ROS package is also provided to allow convenient use of the PCL in a ROS node. This package provides functions for converting between ROS and PCL point cloud types and many other features.

6.1.5 Arm

To provide control over the youBot's integrated 6-DOF arm, we focused on two ROS stacks:

- The Arm Navigation stack [12]
- The Object Manipulation stack [13]

Originally developed for the PR2, these stacks comprise most of what is called the object manipulation pipeline. This pipeline provides a robot-independent set of interfaces, message types, and tools to help one implement common object manipulation actions for a particular robot.

Each of the aforementioned stacks will be introduced in the sections to follow.

6.1.5.1 Arm Navigation Stack

The arm navigation stack was developed by Willow Garage to provide for the collision-free motion of a multiple degree of freedom robot arm. While only implemented originally for the PR2 robot arm, the stack was designed in such a way that it could be used for any arm, with the proper setup. Once that setup is complete, the stack handles collision avoidance, inverse kinematics, and publishes status updates on the arm's progress. In order to move the arm to a given position and orientation, only a relatively simple message is required to set the arm's goal. Once that is received, the stack plans a collision-avoiding route to the target location, and produces a set of joint positions to create a smooth path to the destination.

Oddly enough, the arm navigation stack did not provide anything to actually run through that path. Despite this, we chose to use this stack for the built-in features, the relative ease to set up, and the support for further arm tasks. As previously mentioned, the motion path that is created takes collisions, both with the arm itself and objects discovered in the environment, into account. That would be very difficult to program in a timely manner, sparing us significant development time. There were only two major sections of the program that had to be created for the arm navigation stack to operate properly: the above mentioned program to take the path and execute it, and a description of the arm to be used. The first was easy to create, and the second was generated based on the robot model, which made it simple to implement once the requirements were understood. Finally, the arm navigation stack is used by the PR2 to feed directly into picking up an object with one of its arms, so if we wished to also leverage that code, it would be of a great benefit to follow the same process. In fact, arm navigation actually contains both of the kinematics models used by the object manipulation stack below.

We looked into a few other possible kinematics models and arm controllers, but none of them provided the level of functionality or support that the arm navigation stack did. The official KUKA youBot arm manipulation software was designed for a previous version of ROS, and had not been updated, in addition to not containing collision avoidance capabilities. Compared to all other options, the arm navigation stack provided the most useful features and the easiest implementation.

6.1.5.2 Object Manipulation Stack

The object manipulation stack provides the framework for picking up and placing objects using ROS. The stack is designed to be fairly robot independent, but requires some robot specific components to work properly. These robot specific components include a grasp planner, a gripper posture controller, an arm/hand description configuration file, and a fully implemented arm navigation pipeline. The object manipulation pipeline is fully implemented for the PR2 and some other robots, but not for the youBot.

The object manipulation pipeline was designed to work either with or without object recognition. For unknown objects, the grasp planner must select grasp points based only on the point cluster perceived by the robot's sensors. If object recognition is used, grasps for each item in the object database are pre-computed, and should be more reliable than grasps planned based only on a point cluster.

There are two types of motion planners used by the object manipulation pipeline. The standard arm navigation motion planner is used to plan collision free paths. However, this motion planner cannot be used for the final approach to the grasp point since it will most likely think the grasp point will be in collision with the object being picked up. For the final grasp approach, an interpolated inverse kinematics motion planner is used, which will move the gripper linearly from the pre-grasp point to the final grasp point.

The object manipulation pipeline also has the option of using tactile feedback both during the approach to the grasp point and during the lift. This is especially useful to correct errors when executing a grasp that was planned based only on a partial point cluster. Unfortunately, the youBot does not have tactile sensors on its gripper.

Picking up an object using the object manipulation pipeline goes through the following steps [13]:

- The object to be picked up is identified using sensor data
- A grasp planner generates set of possible grasp points for the object
- Sensor data is used to build a collision map of the environment
- A feasible grasp point with no collisions is selected from the list of possible grasps
- A collision-free path to the pre-grasp point is generated and executed
- The final path from the pre-grasp point to the grasp point is executed using an interpolated IK motion planner
- The gripper is closed on the object and the object model is attached to the gripper
- The object is lifted using the interpolated IK motion planner to a point where the collision free motion planner can take over

6.2 Algorithms

Given the complexity of the complexity of a robotic system, it is important to understand the fundamental algorithms employed by the tools we use. In the following sections, we will briefly introduce the approaches taken by the tools we researched. These will be discussed in the contexts of object discovery and arm manipulation respectively.

6.2.1 Vision / Point Cloud Processing

The tools we researched employed a handful of algorithms as a part of processing point clouds. The most significant of these include:

- Spatial Downsampling using a Voxel Grid
- Plane Discovery through Random Sample Consensus (RANSAC)
- Nearest-Neighbor Classification and Kd-Trees
- Iterative Closest Point (ICP)

These algorithms will be briefly introduced in the following sections.

6.2.1.1 Spatial Filtering using a Voxel Grid

Large data sets can create performance issues for computationally expensive processing algorithms. When maximum resolution is not required, a simple way to address this issue is to downsample the information, such that fewer data points remain, but the important information in the data set is preserved.

For spatial data sets, such as a cloud of points in three dimensional space, a voxel grid is an effective data structure that can assist downsampling. A voxel grid is simply a representation of three dimensional space as a grid of cubes. An example of a voxel grid is shown in Figure 9.

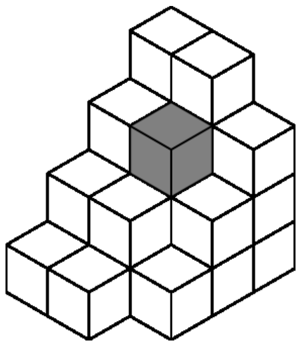


Figure 9: Voxel Grid Example [14]

The general approach for downsampling a point cloud using a voxel grid is:

1. Construct a voxel grid based on the state space of the input cloud and the specified dimensions for each voxel.
2. Sort each input point into a voxel based on its location
3. For each voxel, compute the centroid of the points contained within it
4. Return a point cloud consisting of the centroid point for each voxel.

By configuring the dimensions of each voxel in the grid, one can effectively control the resolution of the resulting point cloud in each dimension.

6.2.1.2 Plane Discovery through Random Sample Consensus (RANSAC)

When searching for objects in an indoor environment with a 3D sensor, there is a lot of noise in the input which has no relevance to the desired result. The most significant sources of this noise are the walls and floor of a room.

In order to discover and subsequently eliminate the noise points produced by the walls and floor, a plane detection algorithm such as RANSAC may be employed. RANSAC, which stands for RANdom SAmple Consensus, searches for consensus among a group of points; in this case, it searches for a best-fit plane [15]. In order to find a hypothetical best-fit plane, a random subset of points is selected from an input cloud, a planar regression model is calculated which fits the random subset well, and the total error between this planar model and all points in the input is calculated. This process is repeated many times, and the planar model with the minimum total error is persisted after each iteration. Finally, the resulting planar model and the set of points that lie within a maximum distance from the resulting model are returned.

Since RANSAC employs randomness, there is no guarantee that the solution will be optimal, or even good; however, this optimality is traded for performance, particularly for large datasets. Instead of testing all possible 2^n subset planar regression models, where n is the number of input points, one selects a fixed number of random subsets to try. Raising the number of iterations improves the probability that a good model will be discovered, although it also lengthens the time required to run the algorithm.

6.2.1.3 Nearest-Neighbor Classification and Kd-Trees

Once an input point cloud has been appropriately filtered to remove background noise, only points that describe potentially interesting objects remain. However, this filtered cloud does not provide any means of distinguishing the points that belong to one object from the points that belong to another.

To determine which points belong to which objects, one can make the assumption that points belonging to the same object will share similar properties, such as a similar location in 3D space, or a similar color. Once a comparison on such an attribute and an appropriate similarity threshold are defined, one can group similar points into clusters.

A common classification algorithm which is often used for clustering is nearest-neighbor. Given a point to classify and sets of already clustered points, this algorithm assigns the unclassified point the classification assigned to the point nearest to it [16]. In the context of clustering, this is modified slightly to allow for the creation of clusters; if the nearest neighbor is too dissimilar according to a given threshold, a new classification is assigned, and the point under consideration is assigned to it.

Since this algorithm performs many searches for points based upon their location in n-dimensional space, it is desirable to index these points in a data structure which optimizes such lookups. A common data structure for this purpose is the k-d tree [16]. A k-d tree optimizes spatial lookups by dividing nodes into two groups along each dimension; the left side of the tree contains points with a value less than the median for the dimension, and the right side of the tree contains points with a value greater than the median for the dimension. An example of such a tree is depicted in Figure 10.

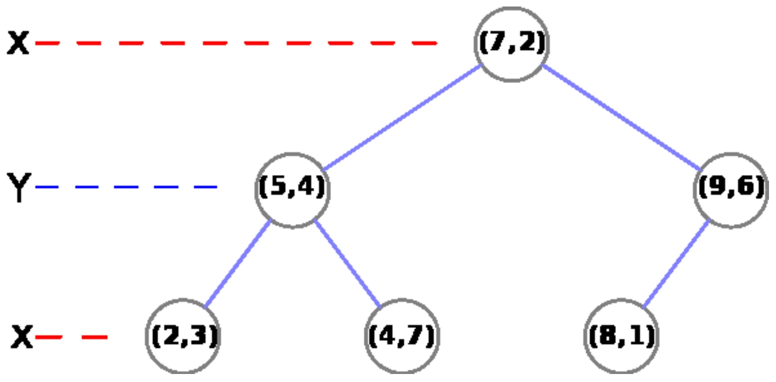


Figure 10: Example k-dimensional tree, where k = 2 [17]

Due to this spatial structure of the tree, a nearest neighbor algorithm may skip nodes that cannot possibly be of interest. For example, during a tree traversal, the algorithm may find that the current nearest neighbor is distance m away along axis A from the node of interest. As the search backtracks up the tree and visits a node that defines a split along axis A, the best distance m is compared against the current distance d along A. If $m < d$, then it is guaranteed that no point at or on the other side of the split could possibly be closer than the current nearest neighbor, and the other branch may be ignored.

6.2.1.4 Iterative Closest Point (ICP)

Iterative Closest Point (ICP) is a simple algorithm for comparing the points in one point cloud to another. It is often used as a simple way to match a point cloud from a sensor reading against meshes in a database of object models.

The algorithm works by iteratively computing a transformation between the points in one cloud to the points in another [18]. The following steps are performed in each iteration:

1. Points are associated between the two point clouds
2. The total error between all associated points is estimated.
3. The transformation is updated in order to minimize the total error.

The first step of the algorithm is match points from one point cloud with the points in the other, such that the error in the transformation between the point clouds can be estimated. Nearest-neighbor classification is a popular choice for this process.

Once the points are matched up with their likely counterparts, the error between each paring can be estimated. A simple function like mean squares is commonly employed.

Finally, the error measurement is used in the previous step to estimate an adjustment to the transformation between the two point clouds, such that the error between the associated points is minimized. The current transformation between them is updated, and the algorithm repeats using the updated transform.

Due to the approximate nature of matching points between point clouds with nearest-neighbor classification, this algorithm cannot guarantee that the resulting transformation is correct. However, its simplicity and performance make it an attractive choice for applications where high precision is not a priority.

7 Objectives

Our goal of providing an always-available online human-robot interaction laboratory poses a series of challenges. Operators should still be able to effectively control the robot regardless of the initial state of the environment the robot is in. In addition, our laboratory presents multiple surfaces that a user may wish to interact with objects on, each of which presents a surface of a unique size and height from the floor.

This project's goal was to satisfy these needs using a Microsoft Kinect mounted on our KUKA youBot's arm. More specifically, the following objectives were identified:

- *The algorithm must be robust.* Object discovery should work from most viewing angles of an object or surface, and should not be affected by the initial pose of the objects or the robot. Object discovery must also not be affected by the height or size of a viewed surface.
- *The algorithm must distinguish between objects and surfaces.* Objects sitting on a surface, like the floor or a table, must be reported as distinct entities.
- *The algorithm must be able to identify surfaces that could support an object.* The vision software must be able to identify suitable locations for placing previously-grasped objects.
- *Captures of discovered objects must be reasonably correct.* Internal representations of discovered objects must be correct enough to allow for effective grasping with the youBot gripper.

I searched for existing ROS packages that could discover objects given depth information, and found the tabletop_object_detector [9] package and the Object Recognition Kitchen [8]; unfortunately, neither of these packages proved robust enough to satisfy our requirements. As a result, I devised an algorithm that would satisfy our requirements and implemented it as a ROS stack.

The following sections will discuss the methodology used to accomplish these objectives and how well the method works in our laboratory environment.

8 Methodology

Given the relatively short duration of this project, I needed a simple algorithm that would work well enough and be simple to maintain. To this end, my work is based off of the Point Cloud Library (PCL) Euclidian Cluster Extraction tutorial, provided by the Open Perception Foundation on pcl.org [11].

This example segments a given point cloud into a collection of point clouds that represent objects in the original point cloud. This is achieved though the following high-level process:

1. The input cloud is downsampled using a voxel grid.
2. Large planes are filtered out of an image, to eliminate building walls and floors.
3. The remaining points are clustered into object point clouds using nearest-neighbor classification on a kd-tree data structure.

This provided promising results, but the results yielded some objects that were not relevant to our scenario. For example, the side face of a table, table legs, and other large objects were discovered. To eliminate this noise, I define a few simple measurements for a point cloud which allow one to filter out unexpectedly large or small objects.

In addition, I extended the example to report the surfaces that it had eliminated from the original point cloud. To make this surface information more useful for placing objects on them, I also define a method to eliminate planes that do not appear parallel to the xy-plane; i.e. planes that are not level with respect to the floor.

In the following sections, I will discuss how each of these parts of the overall algorithm was implemented, and the reasoning behind each approach. I will include visualizations of each step as applied to the following example scenario.

To improve clarity for this report, many of the visualizations have been dilated. This transform effectively replaces each non-black pixel in an image with a cluster of four pixels of the same color, such that they can be more easily distinguished from the background. Visualizations that have been dilated are captioned appropriately.

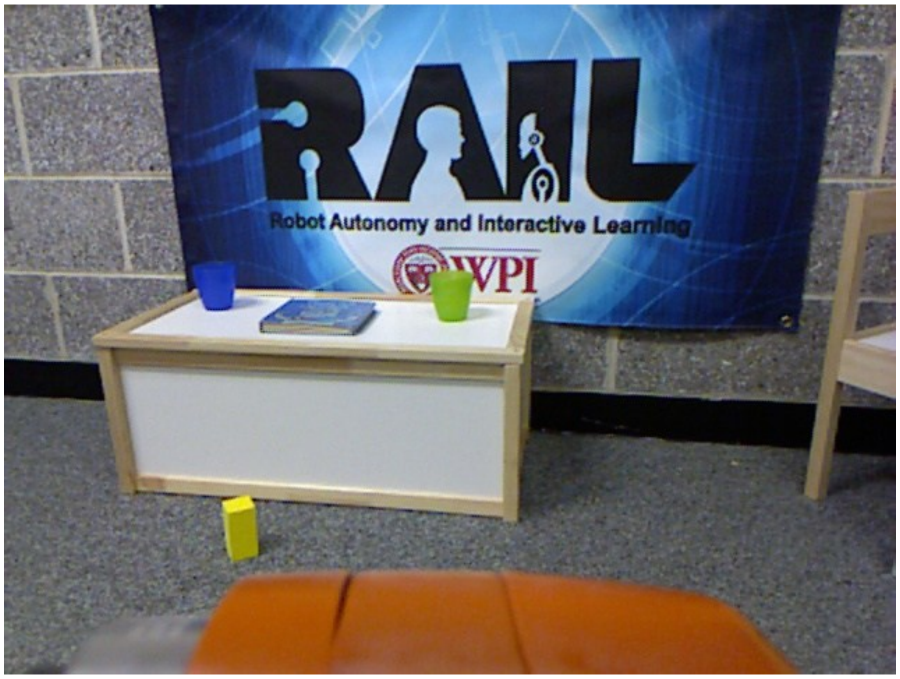


Figure 11: RGB View of Example Environment

Figure 11 is a picture of the example environment as taken from the Microsoft Kinect's RGB camera mounted on our KUKA youBot. This environment includes a few small objects that could be of interest, including a blue cup, a green cup, and a small yellow block. The cups are placed on top of a small storage chest, while the yellow block is placed on the floor. For a sense of scale, each of these objects is child-size; the cups and block are approximately 10 centimeters in diameter, and the storage chest is approximately 3 feet long by 1 foot tall by 1 foot deep.

In addition, the bottom of this image is dominated by a large orange shape. This shape is actually part of the KUKA youBot's arm, and is visible due to the placement of our Microsoft Kinect on the wrist of the arm looking backwards. The arm will not be as pronounced in any of the following figures, and may be ignored.

Figure 12 is a capture of the same environment from the same perspective with depth information added.

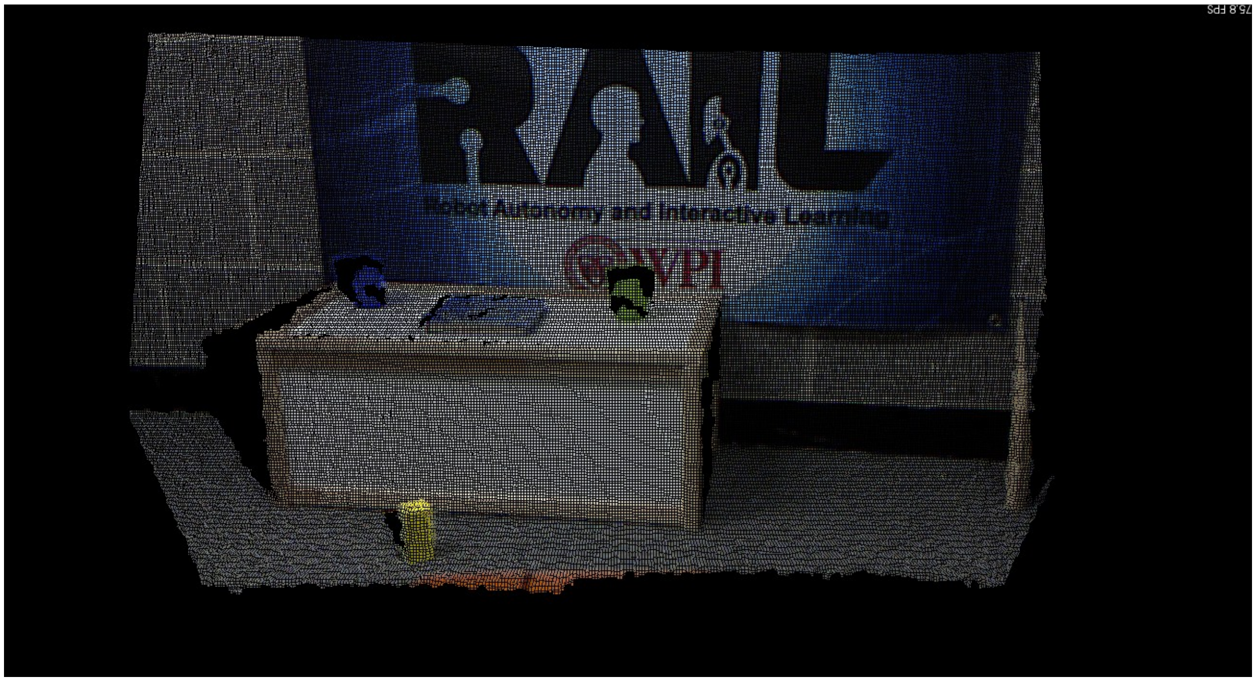


Figure 12: RGBD View of Example Environment

This point cloud was captured using the openni_launch and openni_camera [7] packages for ROS with color registration enabled. This point cloud was saved using the pointcloud_to_pcd tool in the ROS package pcl_ros, and was visualized with the pcd_viewer from the PCL.

All subsequent visualizations are of point clouds like this one.

8.1 Downsampling the Input Cloud

To improve the performance of the complete process, the input cloud is downsampled using a voxel grid (Section 6.2.1.1).

I employ the PCL's implementation of a voxel grid filter with a default grid size of 1 cm^3; the result is shown in Figure 13.

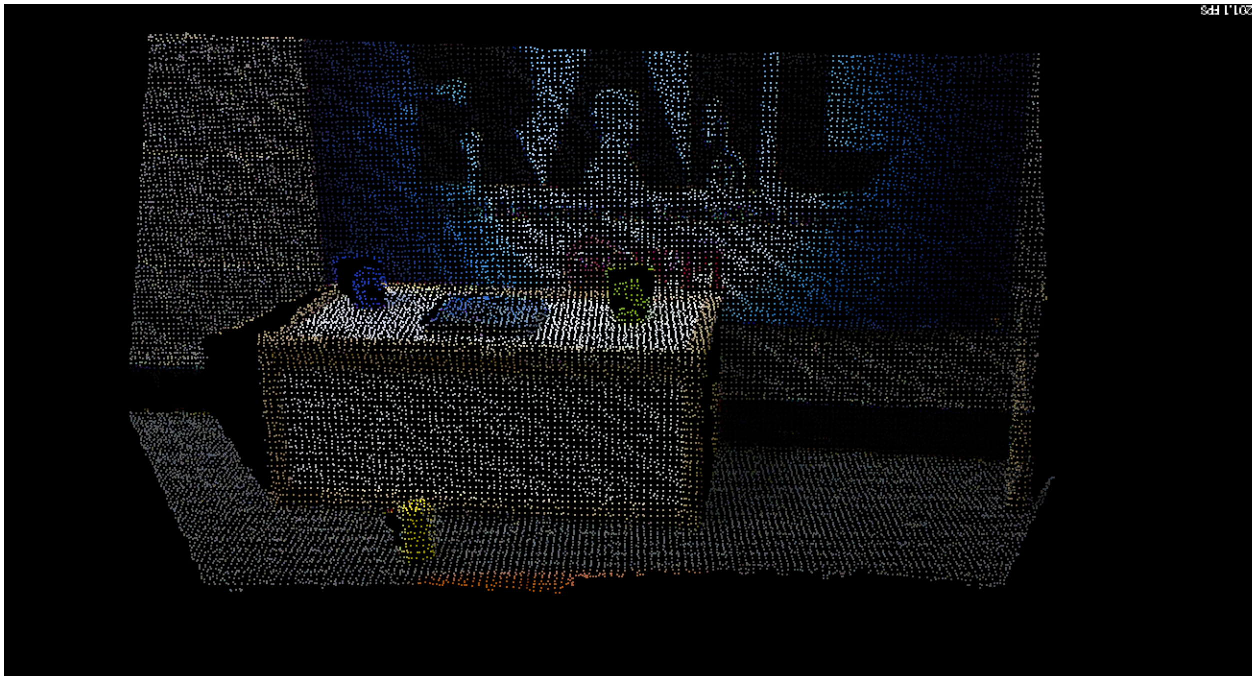


Figure 13: Downsampled RGBD Capture of Example Environment, Dilated

8.2 Plane Extraction

Planes in the input point cloud are discovered using RANSAC (Section 6.2.1.2). The set of inlier points for each discovered plane are then extracted from the input point cloud and returned as a separate point cloud.

The Euclidean Clustering Example provided by the Open Perception foundation demonstrated a different threshold for determining when to stop extracting planes; their example continued to discover and remove planes until the total number of points in the filtered cloud was reduced by a certain factor. I found that this approach made it difficult to tune how aggressively the algorithm removed planes from the input point cloud, particularly for the surface of tables, without encouraging the algorithm to damage object information by removing planes of insignificant size. Directly specifying the minimum satisfactory number of points in a plane made tuning more effective.

I employ the PCL's implementation of RANSAC for plane discovery. A RANSAC pass is computed and the estimated plane is extracted until a plane with fewer inlier points than a default minimum size of 1000 points is discovered. A point is deemed an inlier of a plane if it is no more than 1.8cm from the closest point on the plane. The planes discovered by RANSAC are shown in Figure 14; they have been artificially recolored for clarity.

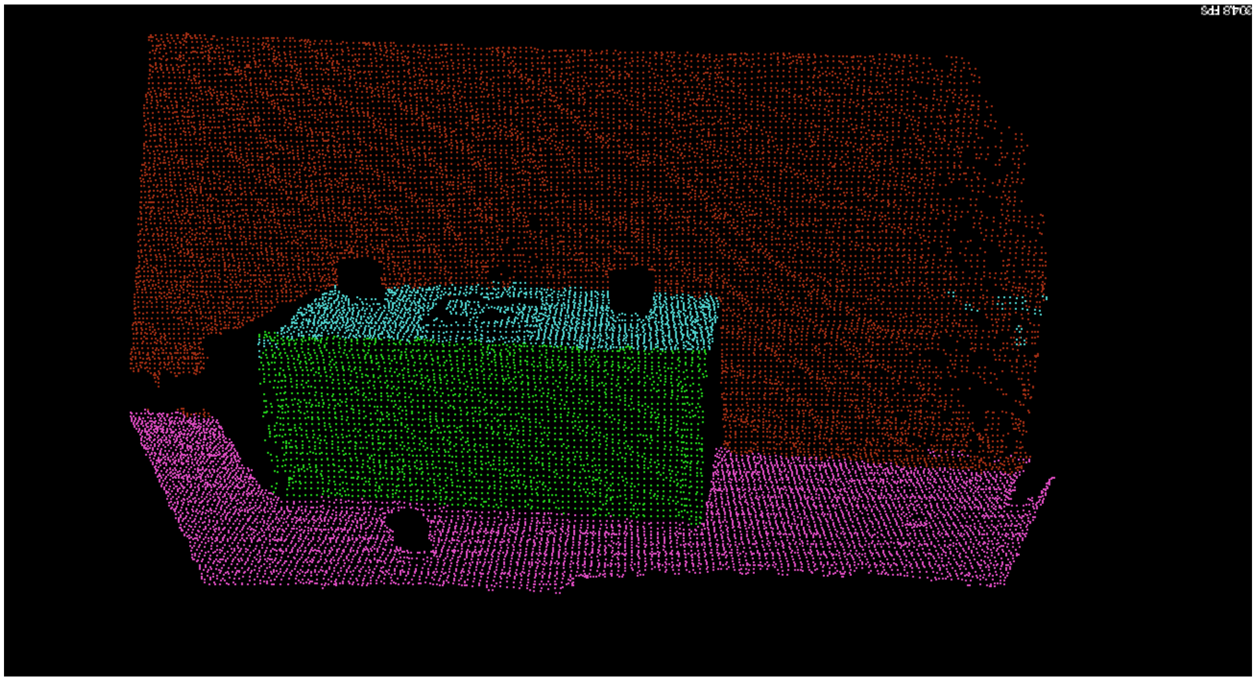


Figure 14: Discovered Planes, Artificial Color, Dilated

The resulting point cloud with planes removed is shown in Figure 15, which has also been artificially recolored.

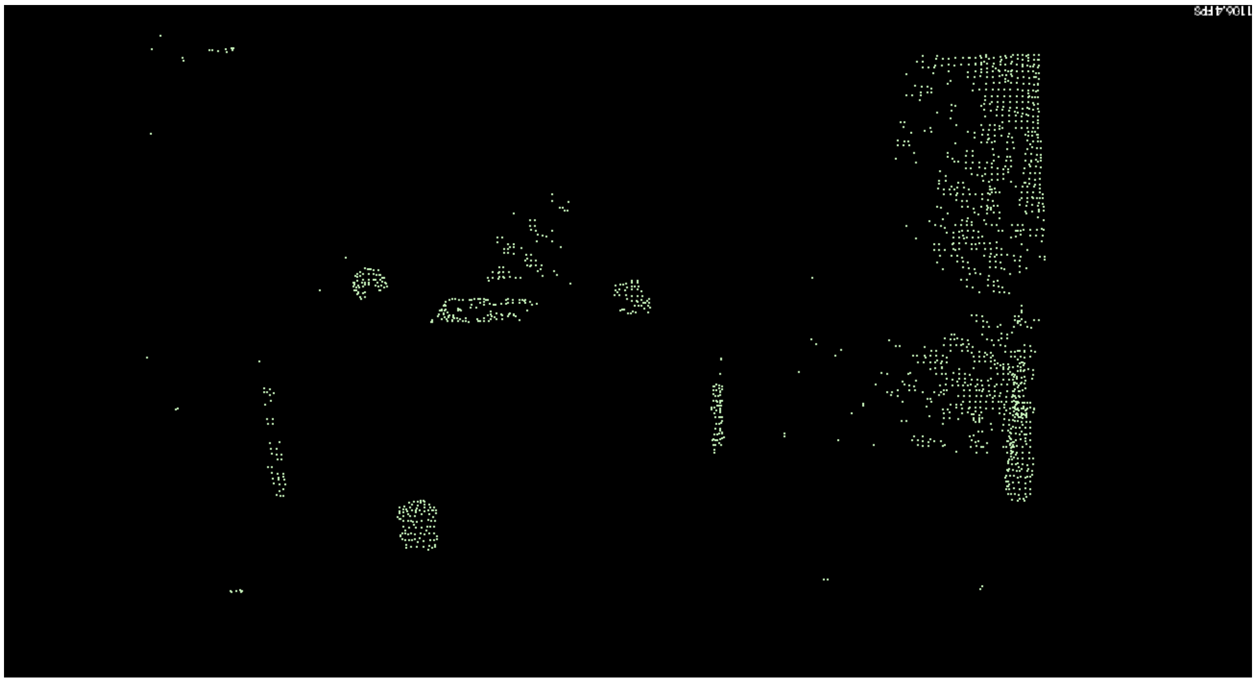


Figure 15: Downsampled Cloud of Example Environment with Planes Removed, Artificial Color, Dilated

The objects in the image now start to become visible; the two circular blobs in the middle of the image are the cups on the chest, and the blob in the bottom center of the image is the yellow block. The computer still represents all of these points as one point cloud, however, hence every point has the same color; the next step in the process is to classify these clusters as distinct.

Beyond the points we expect around the small objects in the room, there are still some points along the back wall and the sides of the chest. These points are too far from their respective plane models, and therefore are not removed. A larger tolerance would reduce this noise, but would also trim more points from objects on a surface. Ad-hoc testing in our environment appears to show that distance tolerance of 1.8cm provides a good balance between noise rejection and object resolution.

8.3 Point Clustering with a Kd-Tree

Following plane extraction, it is assumed that all remaining points belong to some object in the room. Each of these points is inserted into a Kd-Tree data structure by its 3-dimensional location in the environment, and nearest-neighbor classification is employed to cluster them into distinct objects (Section 6.2.1.3).

Before the plane extraction algorithm was revised to handle table surfaces appropriately, this algorithm tended to cluster the top of a table and all of the objects sitting on it together. We successfully altered the representation of a point such that color information was also taken into account in the kd-tree, although attempting to scale the significance of color differences vs. spatial location differences led to instability in the PCL's nearest-neighbor classification implementation, and the approach was ultimately abandoned.

I employ the PCL's implementation of Euclidean Clustering with a Kd-Tree, and cluster nearby points with a default tolerance of 10 cm. Only clusters with at least 25 points and at most 10,000 points are accepted as useful results.

Figure 16 illustrates the result of this algorithm with objects shown in artificial color for clarity.

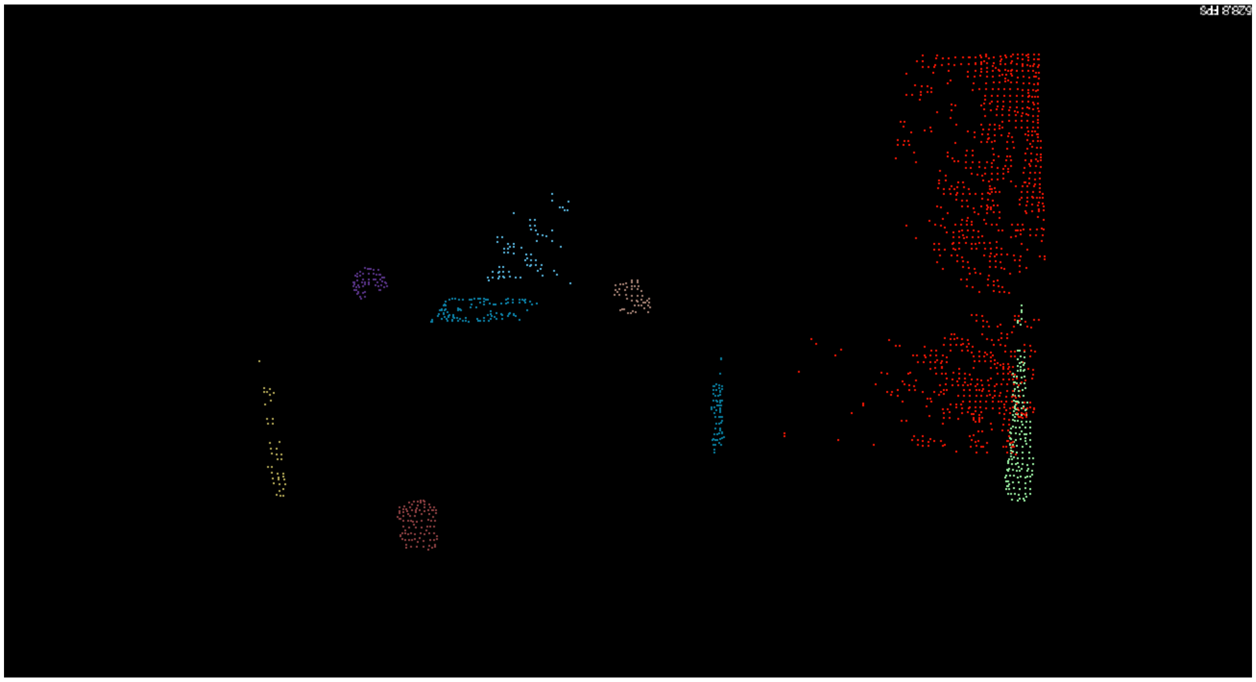


Figure 16: Clustered Objects from Example Environment, Artificial Color, Dilated

All of the major groups of points now appear as distinct objects, and many of the outlier points that did not meet the minimum size cluster criteria were removed.

Figure 17 shows the same object clusters where each point has its original color.

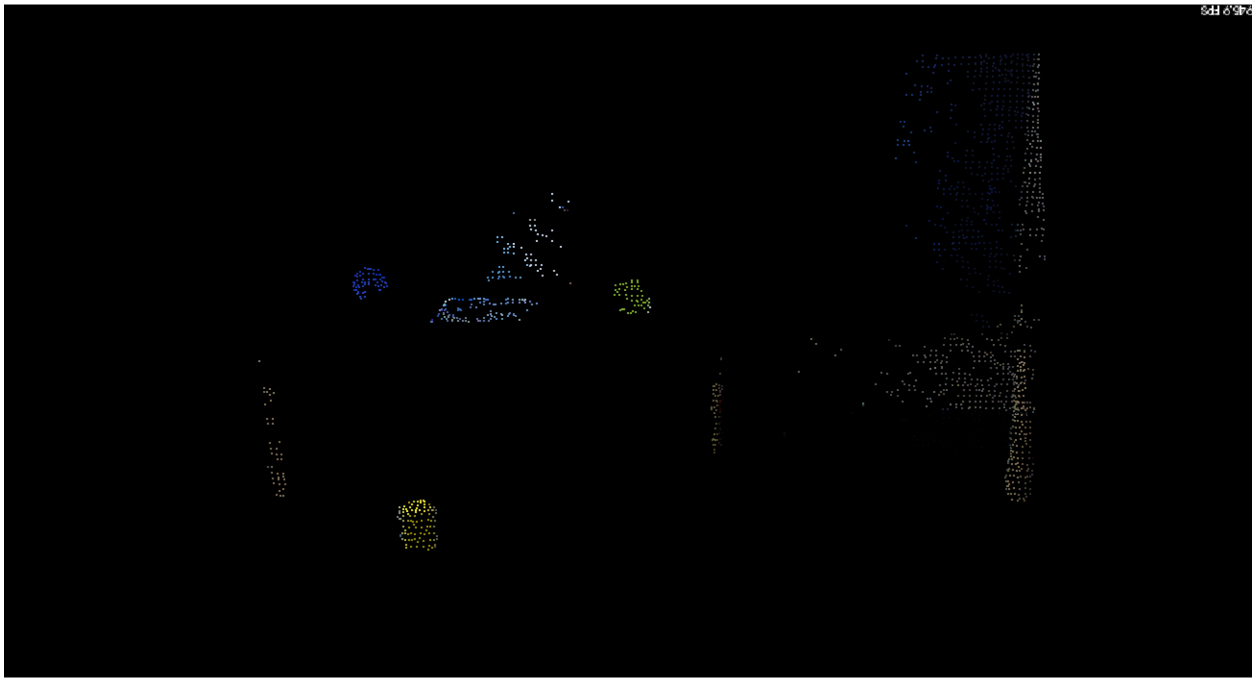


Figure 17: Clustered Objects from Example Environment, Real Color, Dilated

One can now clearly see the green cup, the blue cup, and the yellow block as distinct objects in the world. However, noise still remains, in the form of objects that are not interesting to us, including parts of the wall, parts of the chest, and perhaps the book on the chest. The next step is to define criteria by which one can determine whether a given object is interesting, such that uninteresting objects can be discarded.

8.4 Filtering Results by Measured Attributes

Once all objects are discovered from a point cloud, the results are filtered to remove objects that do not appear relevant to our robot. These criteria include the object's estimated distance away from the robot and the object's estimated size.

The distance of an object away from a robot is computed by first estimating the centroid of the object. The centroid of an object is computed by computing the average point for the object's point cloud; i.e. by adding all of the object's points together and dividing by the number of points for the object. The object's distance away from the robot is then the Pythagorean distance between the origin and the object's centroid in three dimensions.

Given the centroid for an object, I also estimate the size of an object by computing a minimum bounding sphere for the object around its centroid. The radius of this minimum bounding sphere is simply the maximum Pythagorean distance from the centroid of the object to any of the points in its point cloud.

With these two measurements defined for object point clouds, it then becomes trivial to filter out objects that appear too close, too far away, too large, or too small.

Figure 18 shows the remaining objects after filtering by the radius of each object's minimum bounding sphere.

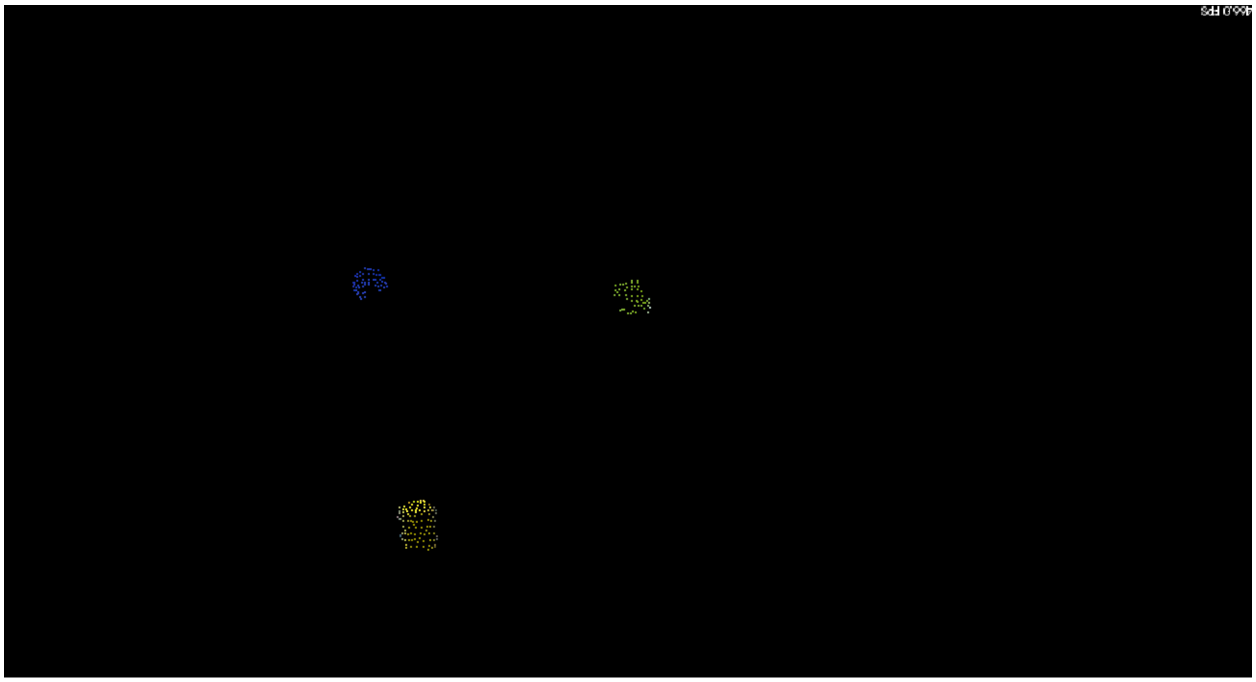


Figure 18: Objects with a Spherical Bounding Radius in [3cm, 5cm] from Example Environment, Real Color, Dilated

For this example environment, only the blue cup, green cup, and yellow block remain, which is the ideal result. It is worth noting, however, that such crisp results tend to be the exception, not the norm. A spherical bounding radius range of [3cm, 5cm] works very well for this particular environment and this particular view, but it does not always completely eliminate noise. The algorithm is particularly susceptible to partially occluded table legs, which appear to be the right size, but are not interesting to manipulate.

The planes discovered during plane extraction are also filtered to discard planes that appear to be tilted, vertical, or otherwise non-flat from the perspective of the robot. Each plane discovered during plane extraction is annotated with the plane model that was estimated for it, in the form of $ax + by + cz + d = 0$.

I assume that the input cloud has been transformed to account for the perspective of the Microsoft Kinect, and therefore planes which appear parallel to the xy-plane are level with respect to the floor. For these planes, the a and b terms go to zero, leaving equations of the form $cz + d = 0$. Therefore, I argue that planes for which the absolute value of a or b is larger than a given tolerance are not level and may be discarded.

Figure 19 illustrates the resulting planes that are classified as level with artificial color for clarity.

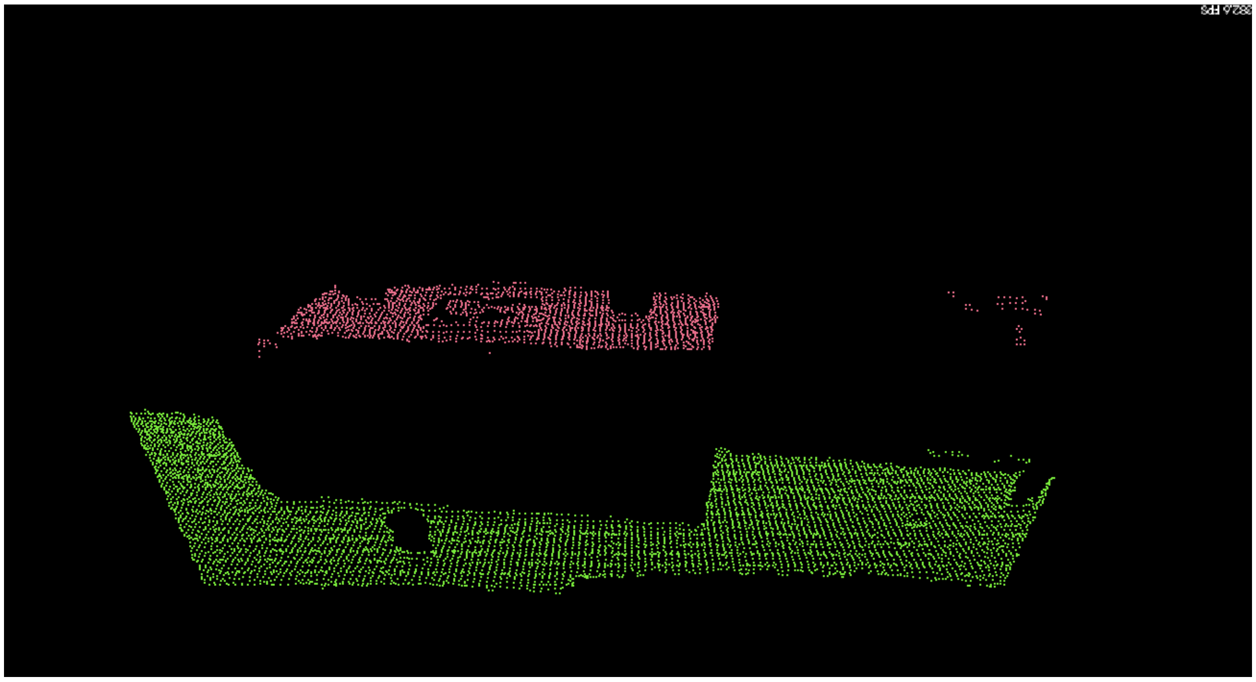


Figure 19: Level Planes, Artificial Color, Dilated

These planes are provided as results alongside discovered objects, such that they may be used by an object manipulation algorithm as possible surfaces on which objects may be placed.

9 Results

To recap the example discussed in the methodology, the algorithm starts with a point cloud such as the one in Figure 20.

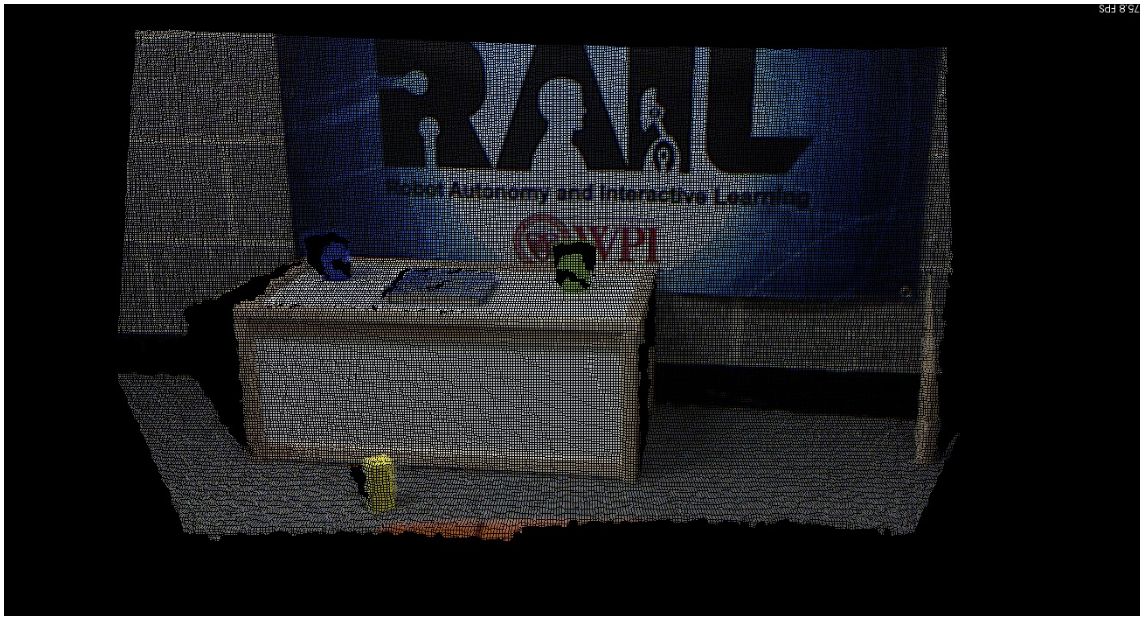


Figure 20: RGBD View of Example Environment

This point cloud is then filtered, classified, and filtered again to produce objects and level planes as in Figure 21 and Figure 22.

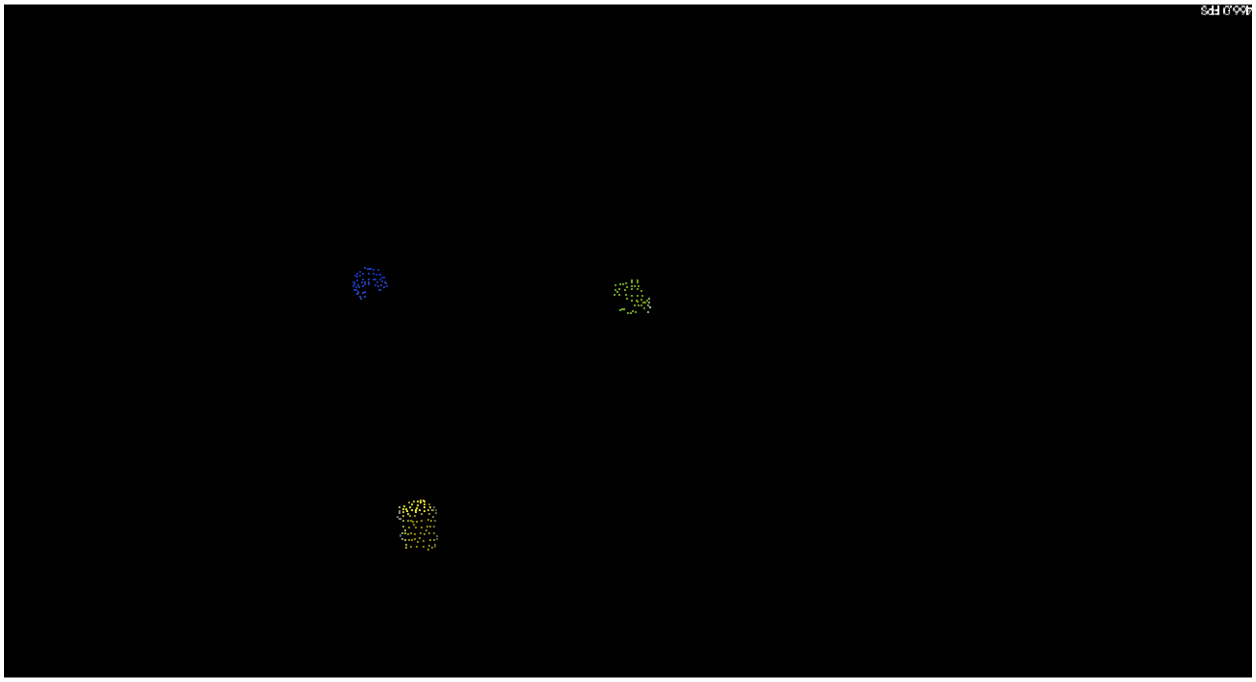


Figure 21: Objects with a Spherical Bounding Radius in [3cm, 5cm] from Example Environment, Real Color, Dilated

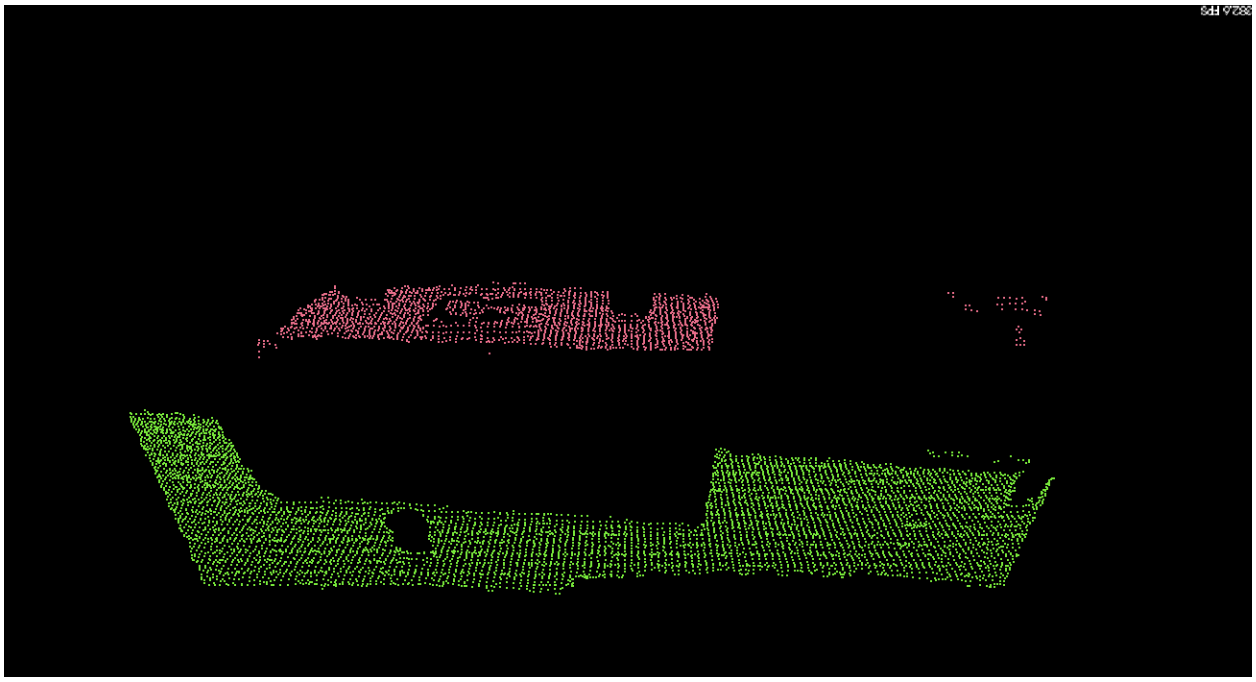


Figure 22: Level Planes, Artificial Color, Dilated

To refresh your memory, the objectives for the project are:

- *The algorithm must be robust.* Object discovery should work from most viewing angles of an object or surface, and should not be affected by the initial pose of the objects or the robot. Object discovery must also not be affected by the height or size of a viewed surface.
- *The algorithm must distinguish between objects and surfaces.* Objects sitting on a surface, like the floor or a table, must be reported as distinct entities.
- *The algorithm must be able to identify surfaces that could support an object.* The vision software must be able to identify suitable locations for placing previously-grasped objects.
- *Captures of discovered objects must be reasonably correct.* Internal representations of discovered objects must be correct enough to allow for effective grasping with the youBot gripper.

Although measuring against these objectives is rather subjective due to their high-level and conceptual nature, I argue that the example input and example results in Figure 20, Figure 21, and Figure 22 satisfy the second, third, and fourth objectives. The results clearly show that distinct objects on surfaces are distinguished appropriately, and that level-appearing surfaces are identified. The resulting objects are of reasonably good accuracy as well; although the resolution of the input point cloud is reduced during processing, the resulting object clouds retain much of their relevant shape.

This example also partially satisfies the first objective, in that objects placed on surfaces at different heights are handled properly. However, I have only considered one environment and perspective so far; the remainder of this section will discuss additional example environments and the results for each.

This next example is of looking down at three small cups on a children's table at a slight angle. The input cloud as captured from the Microsoft Kinect is depicted in Figure 23.

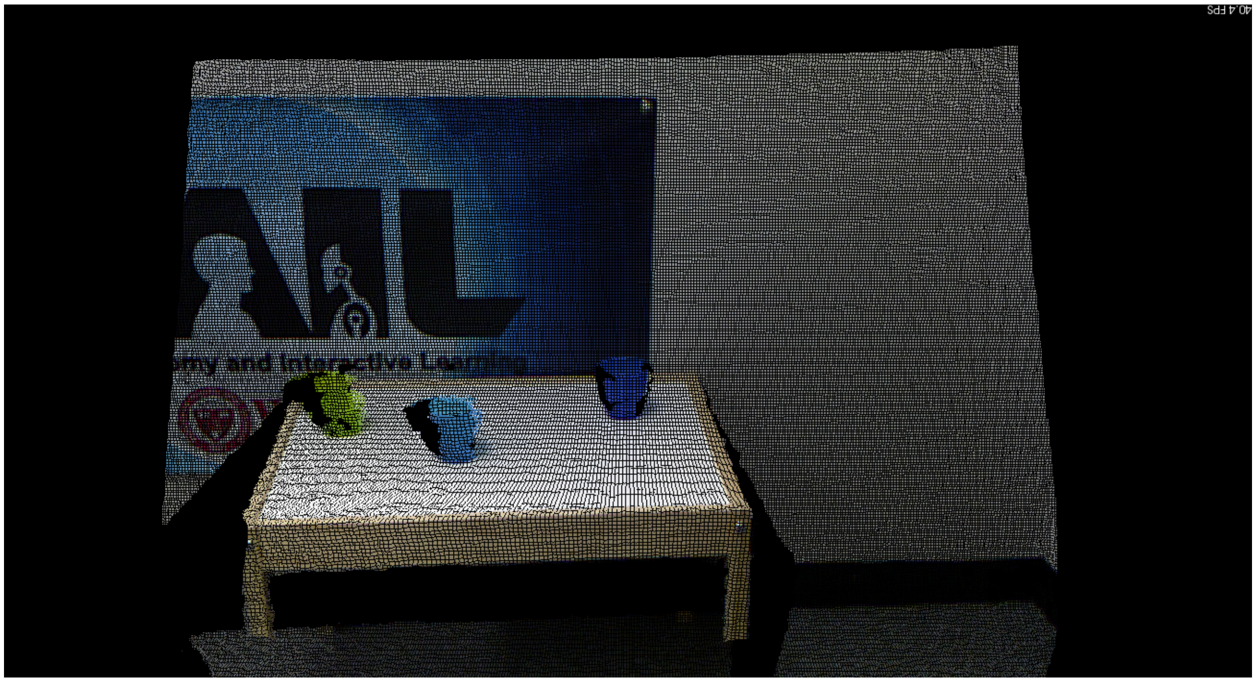


Figure 23: RGBD View of Three Cups on Small Table

The resulting objects, before filtering by size, are shown in Figure 24.

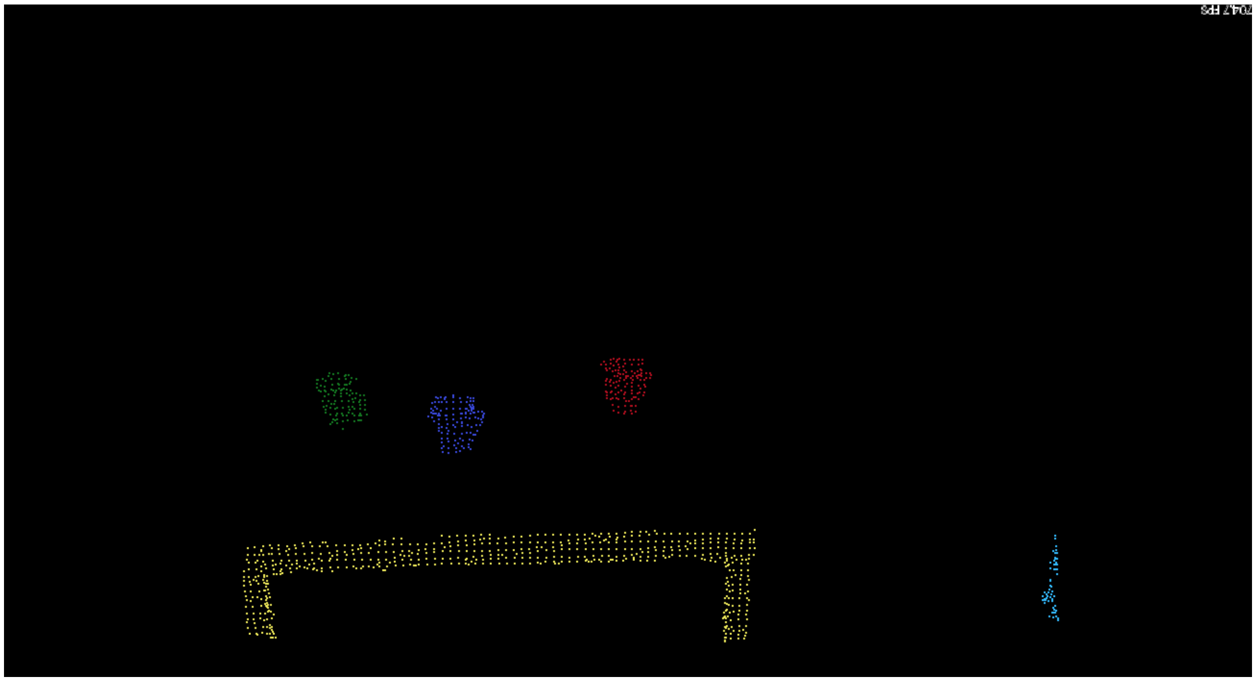


Figure 24: Objects in Three Cups on Small Table Example, Artificial Color, Dilated

In this example, some of the points significantly behind the middle cup were still included in its object cluster, causing it to have a minimum bounding sphere radius of approximately 13cm while the other two cups have a minimum bounding sphere radius of only 5cm. The outlier points that are included in the middle cup are shown in Figure 25.

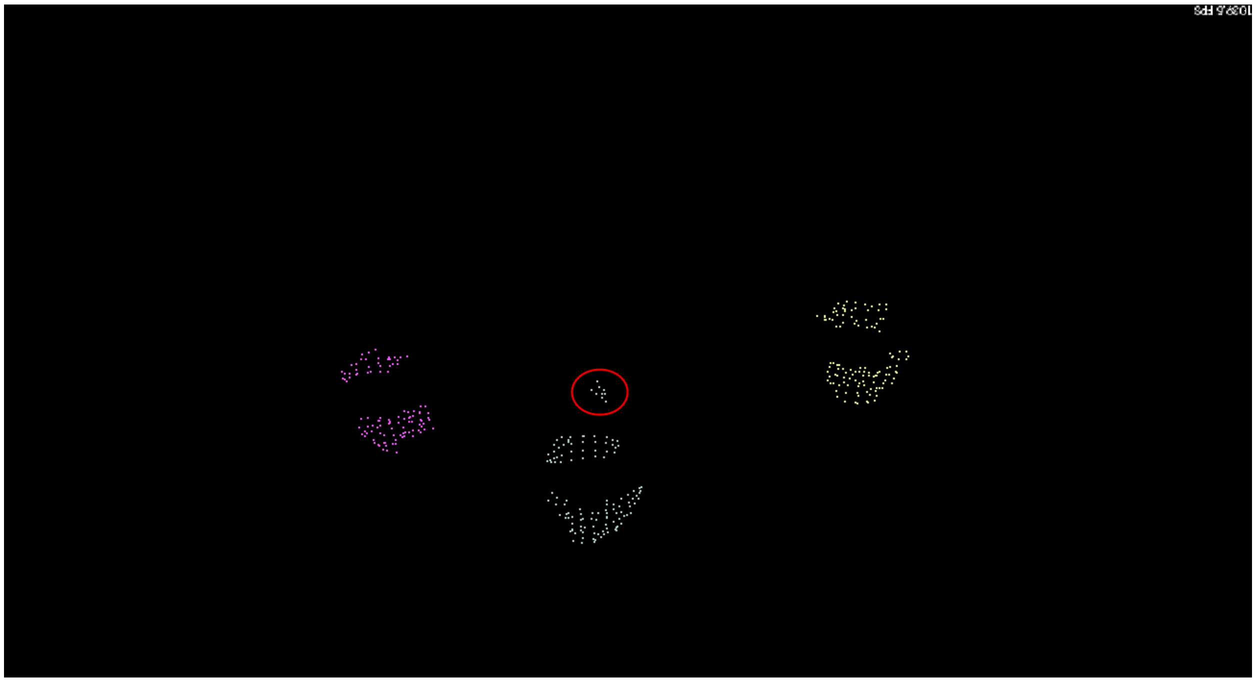


Figure 25: Three Cup Object Point Clouds, View Moved Forward and Tilted Down, Artificial Color, Dilated

The outlier points included with the middle cup are highlighted by the red circle in Figure 25. These points lie approximately 5 centimeters behind the back of the cup.

These points were most likely included in the middle cup due to the clustering algorithm's default tolerance of including points up to 10cm away from other points. Tightening the cluster tolerance would very likely eliminate this issue.

Even with a larger than normal size range to accommodate the middle cup in this example, filtering the object clouds by minimum bounding sphere radius works well. Figure 26 shows the objects which have a minimum bounding sphere radius in the range [5cm, 13cm].

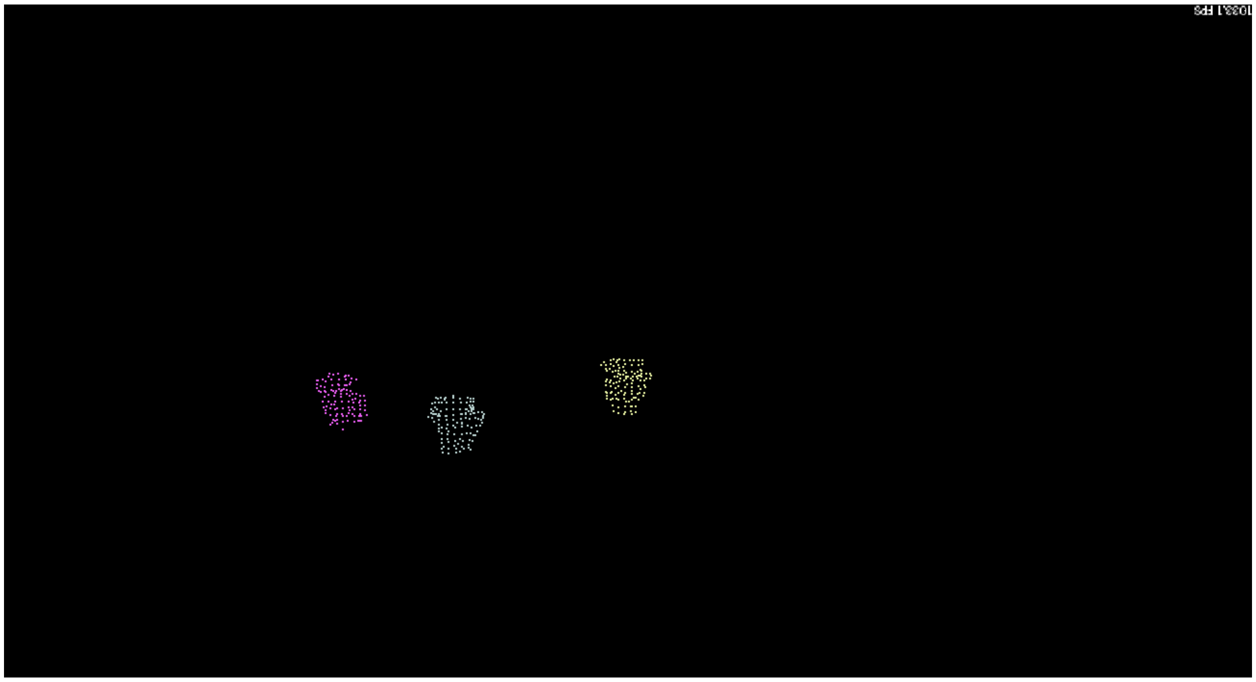


Figure 26: Three Cup Example Objects with Minimum Bounding Sphere Radius in [5cm, 13cm], Artificial Color, Dilated

The level planes discovered in this example are shown in Figure 27.

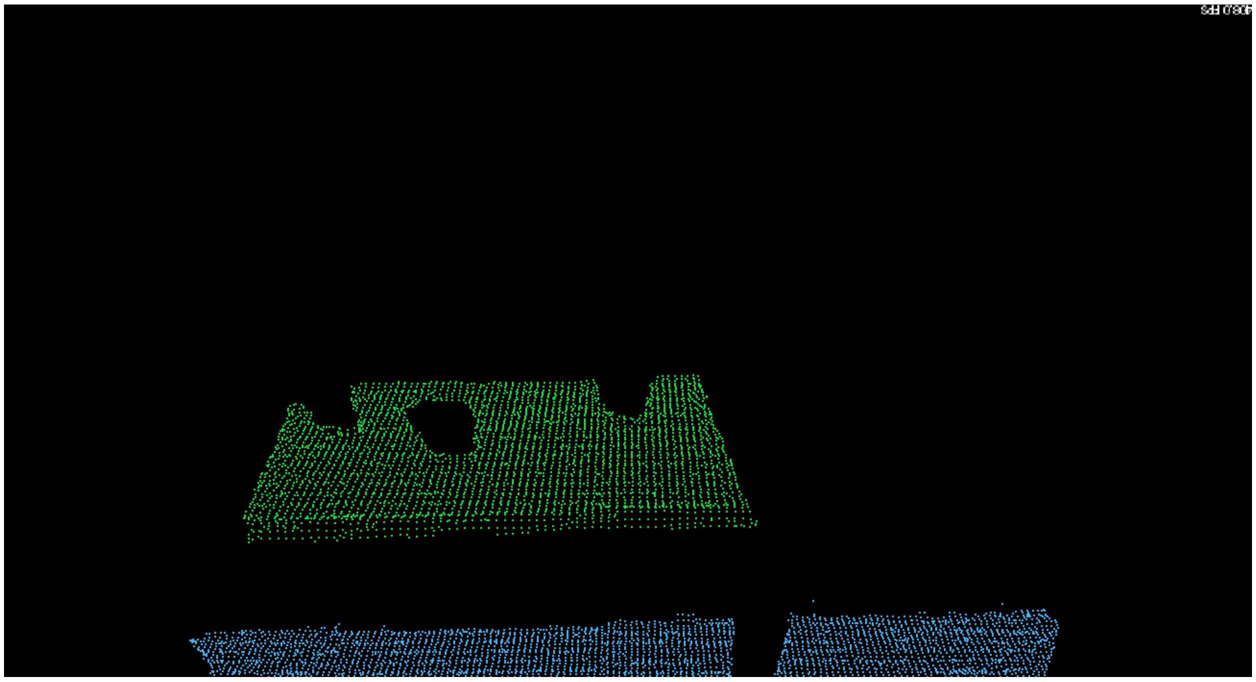


Figure 27: Three Cup Example Level Planes, Artificial Color, Dilated

The surface of the table and the floor are clearly visible.

In this scenario, the only recommended improvement would be to reduce the easily configurable distance tolerance of the object clustering phase.

For my final example, I will show the results of this algorithm on a significantly more complicated environment, depicted in Figure 28.

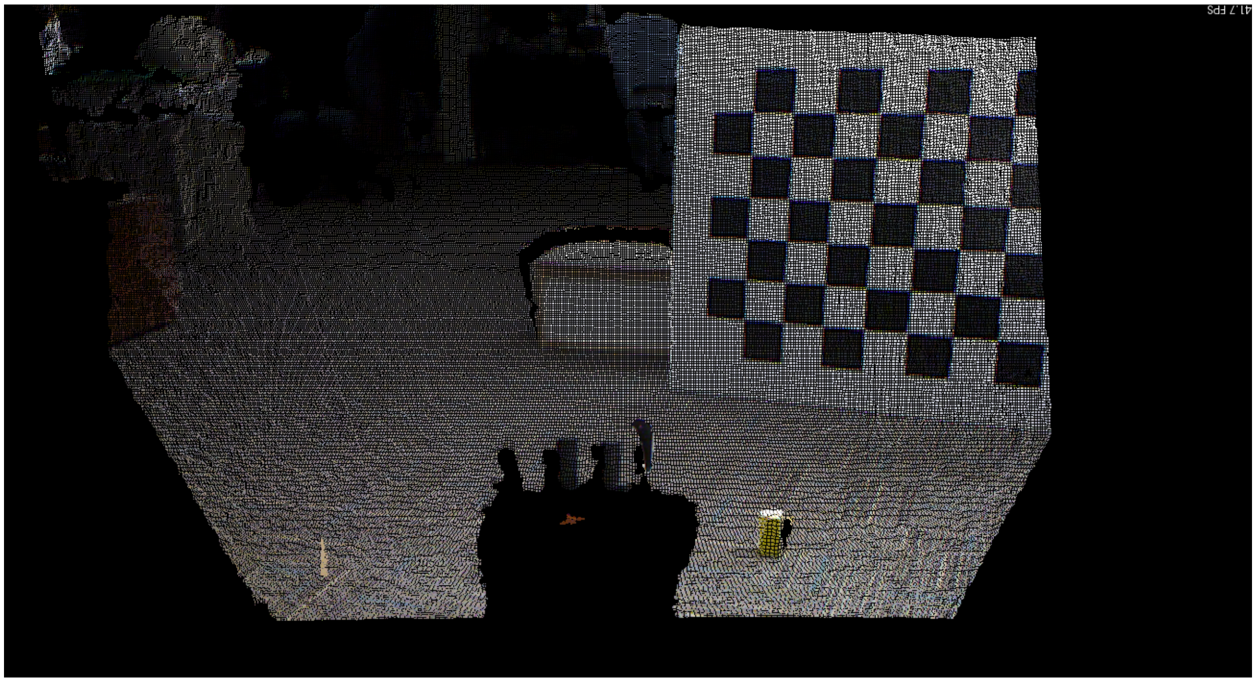


Figure 28: RGBD View of WPI Interaction Lab

This capture is of WPI's Interaction Laboratory in the Fuller Laboratories building. This is a research laboratory for computer science graduate projects, and includes things like tables, chairs, and trash cans in the distance. For this scenario, the goal is to discover the yellow block in the bottom right of the capture.

This capture was taken from a different perspective than either of the prior examples; before the Microsoft Kinect was mounted to the wrist of the KUKA youBot's arm, it was mounted to a raised platform above the rear of the robot looking over the front of the robot. The large black shadow in the bottom center of the image was cast by the youBot's arm in stowed position. As a result, some fragments of the arm were captured by the Kinect in the bottom center of the image.

The discovered objects before filtering by attribute are shown in Figure 29.

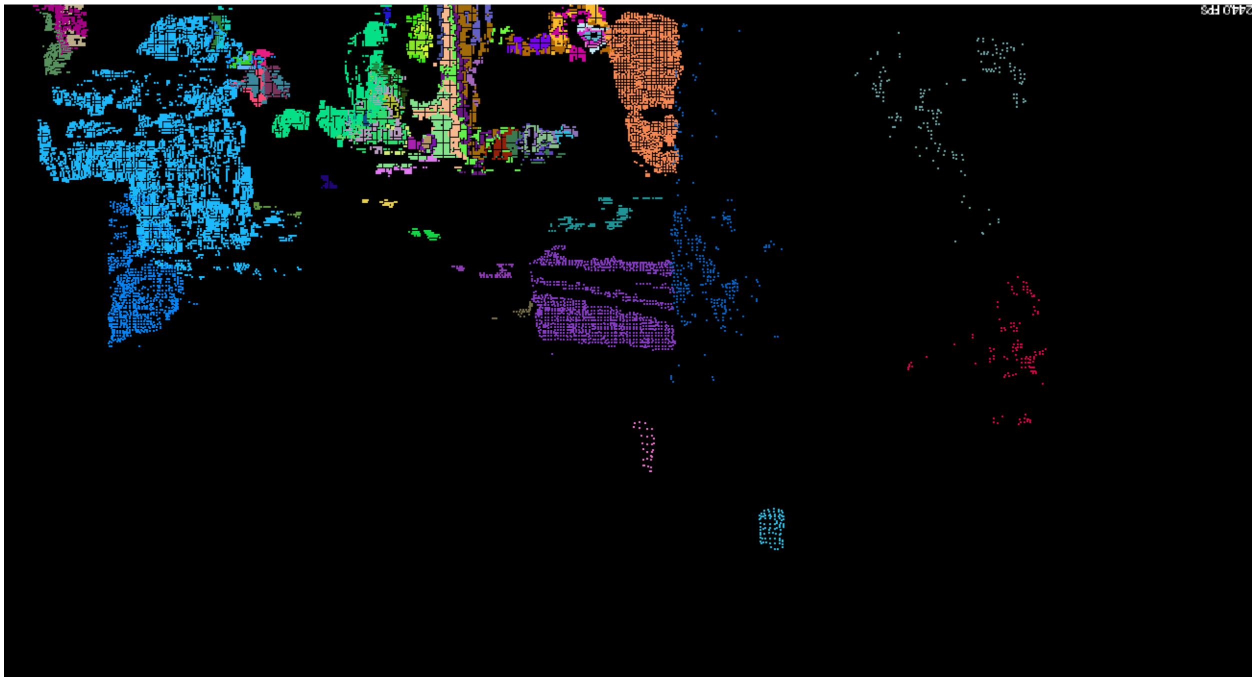


Figure 29: Objects in Interaction Lab, Artificial Color, Dilated

The bottom half of the resulting visualization is pretty clear; the yellow block (shown in turquoise) is clearly visible. In addition, at the bottom center, part of the robot's arm was discovered (shown in light pink).

Farther back, the many objects in the room create noise. The chair appears in the top center (shown in orange), and the garbage cans appear on the left (shown in light blue). In addition, some patches of the floor were not entirely eliminated during plane discovery, and were clustered into objects.

In the spirit of the previous examples, these objects are then filtered by size; only objects with a minimum bounding sphere radius in [3cm, 8cm] are kept. The remaining objects are shown in Figure 30.

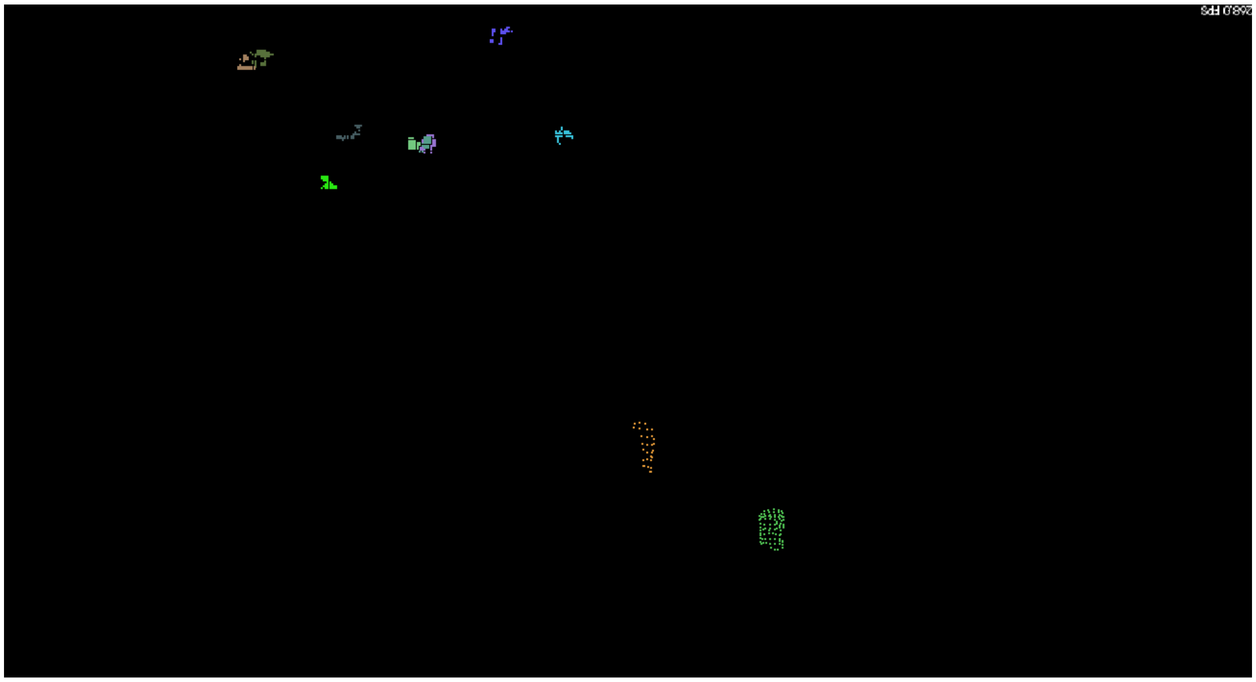


Figure 30: Objects in Interaction Lab with Minimum Bounding Sphere Radius in [3cm, 8cm], Artificial Color, Dilated

Only the two foreground objects and a handful of small background objects remain. Many fewer objects remain, although the noise is not yet satisfactorily eliminated.

By employing additional knowledge of the domain, the remaining noise can be eliminated. One could suggest that objects discovered within a robot are not interesting, because the robot has no need to discover objects on itself. One could also suggest that objects at great distance may not be interesting, since the resolution of their respective point clouds is low and/or the robot must move to reach them. In this example, filtering the resulting objects by a minimum and maximum acceptable distance from the center of the robot leaves only the yellow block as an interesting object.

Finally, the level planes discovered in the Interaction Lab are shown in Figure 31.

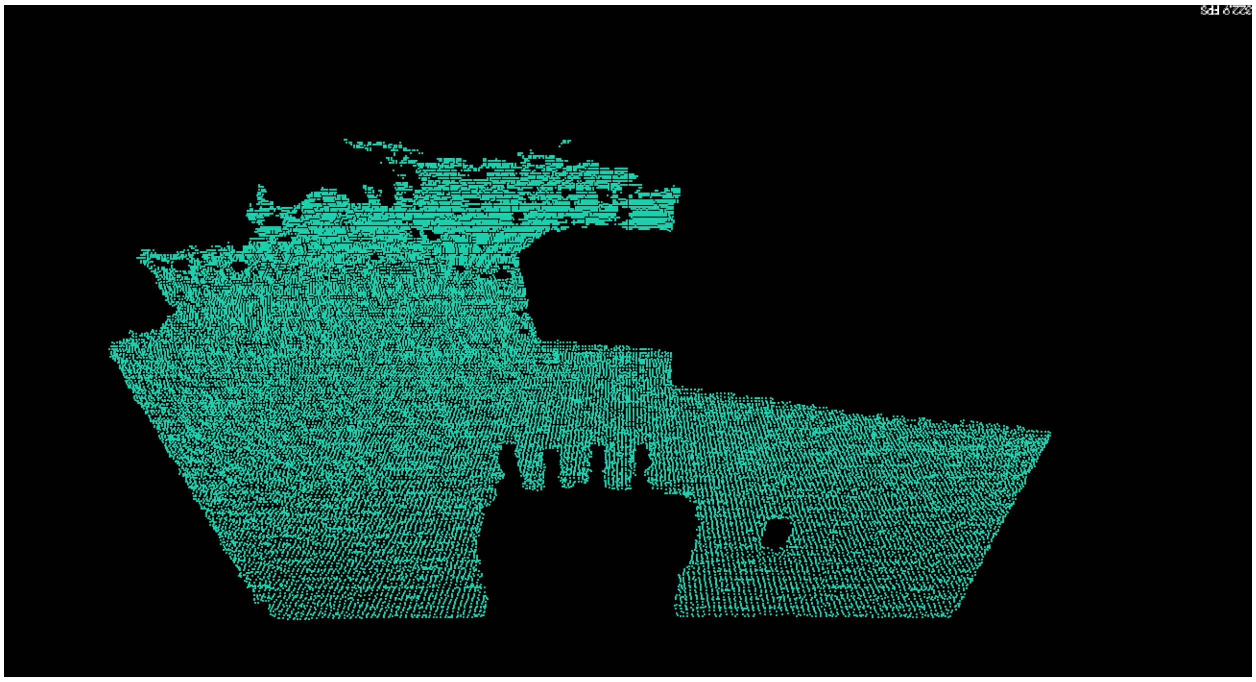


Figure 31: Level Planes in Interaction Lab, Artificial Color, Dilated

The floor was the only level plane discovered in this capture of the Interaction Lab.

These additional scenarios support the conclusion that the proposed object discovery algorithm is suitable for use in new environments and from new perspectives with little tuning required. This fulfills the first objective and satisfies the set of requirements for the project.

10 Limitations and Future Work

The proposed method of discovering objects from RGBD data collected from a Microsoft Kinect sensor device is simple, robust, and easy to tune when required. However, its general simplicity leaves some opportunity for improvement. In this section, I introduce a handful of these.

Level planes are not required to be contiguous. Beyond discovering objects in an environment, the other major deliverable from this algorithm is a set of planes that appear level. This information is particularly relevant for object manipulation tasks; once an object has been discovered and grasped, it naturally follows that one would wish to put it down on some suitable surface.

For simple environments, the algorithm as-is works quite well; the top of a single small table, for example, is modeled well, and a useful bounding box could be derived from the point cloud of the surface. For more complicated environments, however, such as two tables of the same height, the RANSAC plane discovery algorithm would discover both surfaces as a single giant surface which crosses the gap.

This could be addressed by adding a separate clustering stage to process each level plane before it is returned as a result. This would provide an opportunity to split non-contiguous level planes into their contiguous constituents.

Transferring initial point clouds across the network may be a bottleneck. No performance testing has been done at this point. However, from ad-hoc testing during development, it seems that transferring the initial point cloud across the network may be a bottleneck. If further testing verifies this hypothesis, then one could consider separating out the downsampling portion of the process, such that it runs onboard the robot before transferring point clouds across the network for processing.

Filtering by size and distance may not remove all noisy objects. In each of the examples I pose, it is very effective to filter out objects by size and by distance from the robot. However, there are situations where this is not enough. For example, partially occluded legs of a table may appear to be the approximate size of an interesting object. In the event that this becomes a significant issue, it may be necessary to constrain the domain of the objects being discovered further, for example by limiting the interesting objects to be of a specific color and filtering out results that do not present the expected color.

Objects sitting directly next to one another are clustered as a single object. Clustering points together by distance alone can only distinguish between objects that are far enough away from one another. If two objects are directly adjacent to one another, then some additional criteria must be used to separate them. At the moment, this is not a primary use case of the algorithm, and it has been ignored.

This algorithm makes no attempt to recognize objects. Point clouds for each result object only consist of the points observed by the Kinect sensor. No information is known about the occluded surfaces of an object. This project assumes that an implementation of an algorithm to recognize objects and provide an improved model instead would be implemented as a consumer of the results produced by this package.

11 Conclusion

In this report, I present a simple algorithm for discovering objects and level planes using depth information acquired using a Microsoft Kinect. The overall approach is to downsample the an input point cloud, remove large planes, cluster the remaining points into objects by proximity, and use domain knowledge about the features of objects wished to be discovered to eliminate the remaining noise. This algorithm has proven effective in a constrained indoor environment with a set of known objects.

Although simple and effective, this algorithm leaves much to be desired. Discovered planes are not guaranteed to be contiguous, and this may pose a challenge when attempting to place objects on a surface. In addition, multiple objects that are placed too closely together will be discovered as a single large object. Finally, this algorithm makes no attempt to recognize objects and match them to a higher resolution and/or more complete model.

12 Bibliography

[1] R. Toris, 2012.

[2] Locomotec, "KUKA youBot User Manual," 6 October 2012. [Online]. Available: http://youbot-store.com/downloads/KUKA-youBot_UserManual.pdf. [Accessed 11 October 2012].

[3] Willow Garage, "Willow Garage - Software Overview," 2011. [Online]. Available: http://www.willowgarage.com/pages/software/overview. [Accessed 11 December 2012].

[4] Open Source Robotics Foundation, "Home - Gazebo," [Online]. Available: http://gazebosim.org/. [Accessed 8 December 2012].

[5] E. Amos, "Wikipedia - File: Xbox-360-Kinect-Standalone.png," 2 August 2011. [Online]. Available: http://en.wikipedia.org/wiki/File:Xbox-360-Kinect-Standalone.png. [Accessed 9 October 2012].

[6] Kolossos, "Wikipedia - File: Kinect2-ir-image.png," 20 April 2011. [Online]. Available: http://en.wikipedia.org/wiki/File:Kinect2-ir-image.png. [Accessed 8 October 2012].

[7] "ROS Wiki: openni_camera," [Online]. Available: http://www.ros.org/wiki/openni_camera. [Accessed 8 October 2012].

[8] Willow Garage, "Object Recognition Kitchen," 2011. [Online]. Available: http://ecto.willowgarage.com/recognition/. [Accessed 10 October 2012].

[9] Willow Garage, "ROS Wiki: tabletop_object_detector," 2012. [Online]. Available: http://www.ros.org/wiki/tabletop_object_detector. [Accessed 8 October 2012].

[10] Open Perception Foundation, "Point Cloud Library," [Online]. Available: http://pointclouds.org/. [Accessed 8 December 2012].

[11] Open Perception Foundation, "Euclidean Cluster Extraction," [Online]. Available: http://www.pointclouds.org/documentation/tutorials/cluster_extraction.php. [Accessed 6 October 2012].

[12] E. G. Jones, "ROS Wiki: arm_navigation," [Online]. Available: http://www.ros.org/wiki/arm_navigation. [Accessed 8 December 2012].

[13] M. Ciocarlie and K. Hsiao, "ROS Wiki: object_manipulator," 14 January 2011. [Online]. Available: http://www.ros.org/wiki/object_manipulator. [Accessed 11 October 2012].

[14] Vossman, "File:Voxels.svg," 27 October 2006. [Online]. Available:

http://en.wikipedia.org/wiki/File:Voxels.svg. [Accessed 9 December 2012].

[15] M. A. Fischler and R. C. Bolles, "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography," *Communications of the ACM,* vol. 24, no. 6, pp. 381-385, 1981.

[16] R. Stuart and P. Norvig, Artificial Intelligence: A Modern Approach, Upper Saddle River: Pearson Education, Inc., 2010, pp. 738-740.

[17] MYguel, Artist, *Tree_0001.svg.* [Art]. Wikipedia, 2008.

[18] Z. Zhang, "Iterative Point Matching for Registration of Free-Form Curves and Surfaces," *International Journal of Computer Vision,* vol. 13, no. 2, pp. 119-152, 1992.