3D Object Detection with Kinect

Tian Li (tl268)
Prabhat Putchakayala (psp26)
Mike Wilson (mew232)
Keywords: Kinect, vision, object detection, segmentation
Robot: Telepresence

1. Abstract

The goal of our project is to develop a general machine learning framework for classifying objects based on RGBD point cloud data from a Kinect. Using this framework, a robot equipped with a Kinect will take the name of an object as input, scan its surroundings, and move to the most likely matching object that it finds. As a proof of concept, we demonstrate our algorithm on an office/school environment. The 5 major components of this project are as follows. 1) Gather RGBD images of the environment and stitch them together into a 3D map. 2) Implement a 3D image segmentation algorithm building on well known 2D image segmentation algorithms. 3) Create a program to label and extract feature values from the segmented objects; this serves as the training data. 4) Choose a set of features, baselines, and a machine learning model to use. 5) Implement the automated planning and control for the robot.

2. Data Collection

Since our algorithms are capable of working in any environment, there are many types of objects that can be found once trained. We decided to focus on office and school type settings, both because these environments are readily accessible and because a robot capable of finding and possibly retrieving objects in these environments has high potential (for example, locating an object in a cluttered workspace).

To collect data, we obtained a tripod and a wheeled dolly to provide a steady yet mobile base for our Kinect sensor. We then manually moved the setup through many environments and recorded video at 3 frames per second, where each frame was a picture of the environment with RGB values, as well as a depth value for each pixel.

After the fact, the frames from these videos were stitched together into a large point cloud using ROS's RGBD-Slam algorithm. In total, we obtained 20 point clouds using this method, and we received another 7 point clouds from another group working on a similar project, resulting in a total of 27 large point clouds for use in our training phase. These clouds can be added to existing data sets to benefit future projects on 3D analysis of office or school environments.

To limit the scope of our project and prevent having to spend too much time training on many of the objects that can be found in a school or work environment, we limited

our objects to just the following eleven: umberellas, floors, walls, wallets, cell phones, keyboards, trashcans, tables, chairs, water bottles, cd cases . Note that there are more occurrences of objects than point clouds because many of our point clouds contained multiple objects listed here.

3. Segmentation

The segmentation algorithm, at a high level, takes a point cloud as input and, using several metrics, breaks the point cloud into segments, outputting a file representing the point cloud broken into color coded segments.

The first step in executing this segmentation is to read in the data and build an adjacency matrix to localize the data. This matrix is constructed under the direction of our distance metric. The distance metric dictates how close two points must be in order for us to compare them. Using this, we can break our point cloud into an adjacency matrix with buckets of length, width, and height based on the distance metric. The adjacency matrix makes the algorithm feasible from a time perspective, since it allows for efficient lookup of points based on locality.

Once the points are in the adjacency matrix, for each cell in the matrix we calculate an average normal. This is done using RANSAC[1]. because RANSAC returns the “best fit” plane given a cloud of points, to calculate average normal we simply input an adjacency cell into RANSAC and output the plane’s normal as the “average normal”.

Once the adjacency matrix is populated and each cell (and by extension, each point) is given an average normal, we iterate through each point in the data set. For each point, we compare it to each point in all neighboring adjacency cells (we do not need to look at points outside of neighboring adjacency cells since they cannot be part of the same segment by our distance metric). For each of these pairs of points, we run the following heuristic:

```
color = Color Difference[2] < Color Cutoff
norm = Angle Formed(Point1 Norm, Point2 Norm) < Angle Cutoff
dist = distance(Point1, Point2) < Distance Cutoff
return color && norm && dist;
```

If that heuristic returns true for a given pair of points, those points are considered to be in the same segment. In order to represent segments using data structures, a disjoint set[3] data structure is used. This allows for segment merging in a time efficient manner.

Once all the segments are merged together, we run through a loop for each merged segment. Given a segment, first drop all segments that are only a few points large. We then calculate a color for each segment by linearizing the color space and dividing it into equal parts for each segment. Finally, we output a file with each point in the cloud and a “segment color” which is either (128, 128, 128) if the point is not part of any segment (or is part of a dropped segment) or the color as determined by the algorithm.

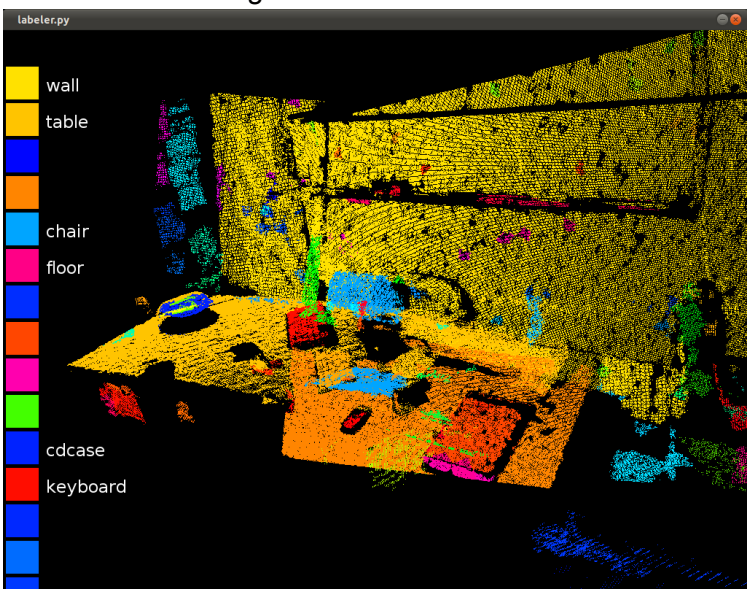
4. Labeling

Once the segmentation algorithm has broken up a point cloud into many segments, a human needs to label the important segments with text and throw out unimportant segments. To facilitate this, we created a 3D point cloud labeler in OpenGL. The input to the program is a list of points in the format <x coordinate, y coordinate, z coordinate, segment's red value, segment's blue value, segment's green value, pixel's red value, pixel's green value, pixel's blue value>, where the segment values refer to the color of the segment the point belongs to as determined by the segmentation algorithm.

Using this as input, the labeling program displays the segmented point cloud. Example:

Point cloud with original colors:

Point cloud with segment colors:



The colors of all the segments are displayed as squares on the left side of the screen. When hovering over a square, the corresponding segment is displayed and all other segments are temporarily hidden. The user can then click on the square and type in an appropriate input. In this image, the user was able to identify and label 6 objects.

Once all the segments have been labeled, the user can output the list of labeled segments along with their labels. This labeler can also be used in future projects related to 3D image segmentation, as long as the points and segments are provided in the described format.

5. Feature Extraction

The feature extraction algorithm takes as input the file output from the segmentation code and the file output by the OpenGL labeling code. It also takes as input “test” or “train”, which determines what the algorithm will do.

First, the code runs through the labels looking for a segment labeled “floor” (as several of our features require knowledge of which segment is the floor segment). If it cannot find the segment labeled “floor” (or if it is in “test” mode, which ignores labels), it attempts to guess which segment is the floor. To do this, it runs through each segment, calculating the segment’s best fit plane using RANSAC[1]. Then, given the plane, the algorithm runs through each point in the segment and calculates distance from each point to that plane. This results in each segment having an “error” value, which is the average distance of a point in a segment from it’s representative plane. This value gives a rough idea of how planar the segment is. The algorithm then combines this knowledge with how “low” the plane is (based on average Z value), and chooses the floor to be the lowest and most planar segment using a weighting of those two values.

Once the code has determined which segment is the floor by reading it or guessing it, the code runs through each segment calculating the segment’s feature vector. The feature vector is as follows:

1. Minimum and Maximum Height: These two values are calculated by calculating the minimal and maximal distance of any given point in the segment from the floor plane.

2. Volume and Surface Area: these are calculated using a straightforward computation of a 3D Bounding box.

3. Angle to Floor: To calculate this, RANSAC is used to calculate this segment’s average normal. Then, the angle between that normal and the floor’s normal is calculated and used as this feature.

4. Average Color: This computation is a straightforward average operation.

Once each of these vectors are computed for each segment, the results are output to a

file to be read in by the machine learning code.

6. Training and Evaluation

a. Baselines

Before implementing our machine learning algorithm, we first created two baselines:

i. Guess a random label: For a given test feature, randomly guess a label from the set of training features. Since this algorithm is not deterministic, the accuracy ranged from about 5% to 15% during cross validation using only the labeled features.

ii. Guess the most common label: For a given test feature, guess the label that appears most frequently in the training data. In our cases, “floor” and “wall” appear most frequently in our training point clouds. The program guessed “floor” for all the test features and was correct 16.5% of the time using only the labeled features.

b. Machine Learning

The machine learning code has no knowledge about point clouds or segments, and indeed can be used for any problem. It takes as input the features extracted from the training set, along with the label of the segment represented by each feature, and trains a multi-class support vector machine[4]. Next, it takes the features extracted from the test set and attempts to classify each one. It also provides a certainty measure for each test feature (in machine learning terms, this is the decision function). Finally, it takes the requested label inputted by the user and outputs the ID of the test feature that has the highest score of all the test features that match the given label. In terms of our project, the program outputs the segment in the test set that it believes is most closely represented by the given label.

To evaluate our algorithm, we performed cross validation by determining one point cloud from the training data to be our test set and then using all remaining point clouds to train. Using only the labeled features as test features, the machine learning algorithm was correct 52.11% of the time. Here is a breakdown of accuracy for each object:

Label: umbrella, Occurrences: 4, Accuracy: 0.000000
Label: floor, Occurrences: 11, Accuracy: 0.545455
Label: wall, Occurrences: 7, Accuracy: 0.428571
Label: wallet, Occurrences: 3, Accuracy: 0.333333
Label: phone, Occurrences: 1, Accuracy: 0.000000
Label: keyboard, Occurrences: 7, Accuracy: 0.714286
Label: trashcan, Occurrences: 2, Accuracy: 0.000000
Label: table, Occurrences: 11, Accuracy: 0.545455
Label: chair, Occurrences: 8, Accuracy: 0.375000
Label: googlebottle, Occurrences: 5, Accuracy: 0.200000

Label: blackbottle, Occurrences: 1, Accuracy: 0.000000
Label: cdcase, Occurrences: 7, Accuracy: 0.714286

Note that during the planning and control stage, the number of test features available will be a lot more than in the training phase because every segment defined by the segmentation algorithm will be used to generate a feature, not just the segments that have been labeled.

7. Planning and Control

We decided to use the Erratic telepresence robot because of the mounted Kinect and ground mobility. The robot is provided a label to find (ex: “keyboard” or “cdcase”), and it starts in the center of a room and turns 360 degrees on the spot, occasionally taking a snapshot with the Kinect. We decided to analyze each snapshot individually rather than stitching them together like we did to create the training data because ROS’s RGBD-Slam algorithm runs very slowly and cannot operate in real time.

Each snapshot is run through the analysis phases above (segmentation → feature extraction → machine learning classification) to determine the best segment in the snapshot (i.e, the segment in the snapshot that received the highest score from the SVM for the given label). The score for the segment is stored as the robot turns and takes additional snapshots.

Once a full circle has been made by the robot, it calculates the position of the segment with the highest score (the position is simply the average position of all the points that make up the segment). Next, it calculates the angle it needs to turn in order to be pointing directly at the segment. Finally, it calculates the distance to the segment and drives towards it, stopping immediately before it makes contact.

8. Demonstration

From the training and evaluation phase, we realized that the keyboard (specifically, we trained using a small, white, iPad keyboard) was very easy for the algorithm to segment and detect, so we decided to use that in our demonstration video.

The robot takes 4 snapshots in 90 degree increments and looks for a keyboard in each one. It does not find any segment in snapshots 1, 3, or 4 with a score higher than 0, but in snapshot 2, it is able to find the keyboard and give it a high score. After taking all 4 snapshots, the robot turns towards the keyboard and drives to it.

A video demonstration of our robot finding a keyboard is available on YouTube here:

http://www.youtube.com/watch?v=rff-0BbZn-U

9. References

[1]: MRPT RANSAC Algorithm: http://www.mrpt.org/RANSAC_C++_examples

[2]: Color Difference Formula: http://www.compuphase.com/cmetric.htm
[3]: Disjoint Set Code: http://www.emilstefanov.net/Projects/DisjointSets.aspx
[4]: PyML: http://pyml.sourceforge.net/