Classification of 3D Object Scans in Urban Environments

Spring 2010 Princeton University

(Due: 5/11/2010)

Christopher Tralie

Class of 2011 Electrical Engineering

Faculty Adviser: Thomas A. Funkhouser

Professor of Computer Science

Submitted in partial fulfillment of the requirements for the degree of Bachelor of Science in Engineering:
Department of Electrical Engineering, Princeton University
This paper represents my own work in accordance with University regulations

I. Project Aim

The purpose of this project is to explore classification of 3D objects extracted from a scan of a city into different categories. This has many applications, such as in photo tourism where objects in a city could be labeled for the user, or in the military where small objects need to be identified correctly after scanning them.

We start with an enormous data set: a 3D scan of the city of Ottawa, merged from laser scans of the city. Work has been done already to segment this enormous point cloud into smaller objects of interest, the point cloud "blobs," which are denser collections of points separated from the background that likely make up one continuous object (e.g. a fire hydrant, a car, or a mailbox) [3]. My main work this semester expands the classification techniques of these segmented blobs by adding a novel 3D shape descriptor based on alignment of these point clouds and analyzing the extent to which it improves classification.

II. Background and Basis for Project

3D shape descriptors have been explored before in different contexts. The idea is to compress information about a shape into a feature by using some objective function, and in turn to transform classification of 3D shapes into a learning problem. Certain methods such as spherical harmonic decomposition and the "D2" descriptor are known to work well for this problem in general [4].

The main challenge of this research comes from incomplete, unstructured object scans due to high noise (from the scanner and improper segmentation) and occlusion (due to the limited viewpoint of the scanner). Even if a very good general 3D descriptor such as the D2 descriptor exists, it may not deal well with these particular types of noise. That is, the feature may very well summarize the 3D distribution of points in an accurate, succinct manner, but it may also bring along with it the corruptions that occur from occlusion. With this specific city application in mind, we form an entirely new descriptor from scratch that is based on alignment. The program will align each point cloud to known 3D models of city objects in a database and report the accuracy of the alignment. The idea is that even in the face of occlusion, a point cloud should align well with a model that is similar to the full object, and so it should get a good score for models to which it is similar in real life had the full scan been taken. The hypothesis is that this will allow partial scans from objects of the same class to cluster together in feature space in a way that they may not be able to using other feature descriptors.

Our group here at Princeton has been one of the main groups working on small, city objects [2]. In addition to some of the classical feature descriptors discussed in [4], the group is also exploring contextual cues (i.e. what other objects are nearby), among other things, for classifying the objects [2]. My work will add to this by exploring a unique feature descriptor that is constructed aligning the point clouds in this manner.

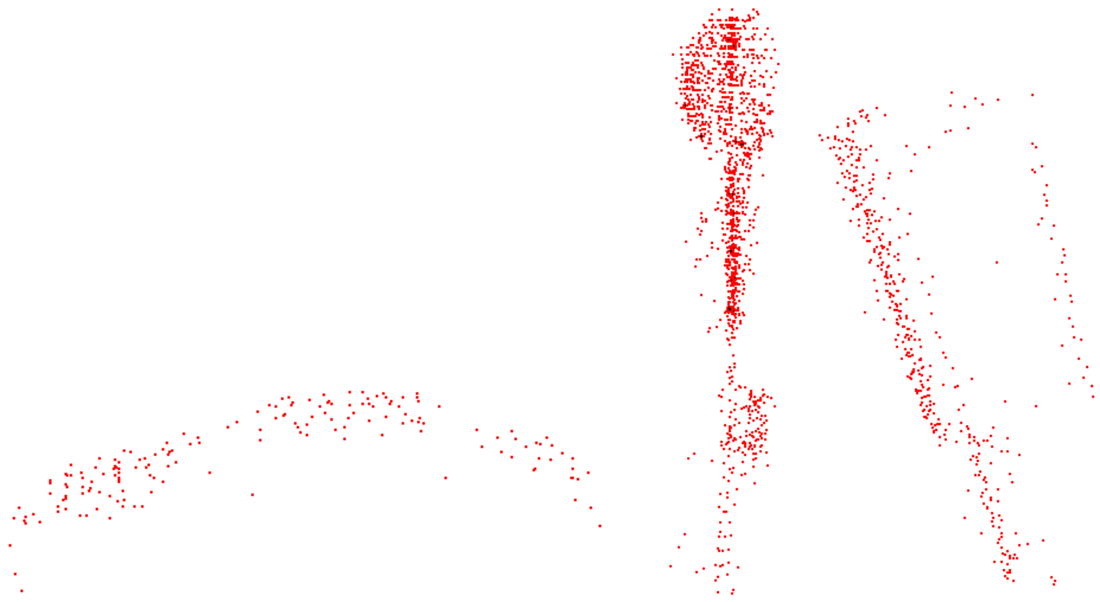
III. Code Framework and Dataset

A. Development Environment

The majority of the code is written under C++ using the "Gaps" graphics library written primarily by Tom Funkhouser [1]. This library facilitates processes such as Principle Component Analysis and 3D triangle mesh construction. It also has functions that help with ICP (explained in IV. A). Most of the code was developed using Microsoft Visual Studio 2008, but the code is cross-platform and the final tests were run under Linux. There were also various intermediate scripts written in Java to facilitate testing and visualization of the alignments, and several previously-developed programs such as "simpleModelFittingViewer" and "mshview" were used to view a point cloud and a mesh together and any number of meshes, respectively.

B. Training Set of Point Clouds

Each object to be classified is a "point cloud," or an unstructured collection of 3D (x,y,z) coordinates in space that were taken by a laser scanner and segmented by techniques presented in [3]. Because of the nature of the scanner, however, many of the point clouds are occluded. In other words, there are only partial scans of many of the objects of interest in the city. In addition, there is much background noise from error in the scanners, in addition to some noise from the segmentation process in [3]



(a) Car Scan (b)Stop Sign (c) Telephone booth

Figure 1: Typical objects of interest scanned from the city. The occlusion problem can be seen in (a), where the bottom of the car is missing, and in (c), where the left half of the telephone booth was scanned better than the right half. Segmentation noise is visible in (b) where points were made part of the stop sign around the base that don't actually belong to it

The training data in this project is composed of 1163 object scan segmentations that have been labeled beforehand. There are 44 different types of point clouds in this training set.

C. 3D Mesh Database

There are 42 objects in the database, all of which were modeled from a specific point cloud in the data set. The most representative point cloud in each of the 42 used classes was picked, with the idea that all of the other point clouds in the city should align well to at least one of the models in the database. In this manner, the database is composed of a collection of "principle models."

The models are stored as triangular meshes. Here is a full list of the mesh database:

A = 1_advertising_cylinder
C = 12_dumpster
D = 14_fire_hydrant
E = 15_flagpole
F = 24_mailing_box(free_standing)
G = 26_newspaper_box
H = 29_parking_meter
I = 33_recycle_bins
J = 43_trashcan
K = 75_highway_sign
L = 97_telephone_booth
M = 133_traffic_control_box
N = 134_traffic_control_unit
O = 138_lamp_post_one_bulb
P = 140_lamp_post_three_bulbs
Q = 141_lamp_post_four_bulbs
R = 142_lamp_post_five_balls
S = 143_lamp_post_three_and_two_balls
T = 144_lamp_post_tall
U = 145_lamp_post_on_light_standard
W = 150_stop_sign
X = 152_street_name_sign
Y = 153_other_traffic_sign
Z = 154_tall_thin_sign_on_pole
AA = 157_a_frame_sign_on_ground
AB = 161_traffic_light_no_arm
AC = 162_traffic_light_half_arm
AD = 163_traffic_light_one_arm
AE = 164_traffic_light_mult_arm
AF = 165_traffic_light_on_light_standard
AG = 172_short_post_in_row
AH = 175_tall_post_in_fence
AI = 181_light_standard_no_arm
AJ = 182_light_standard_mid_arm
AK = 184_light_staandard_T_with_sign
AL = 185_light_standard_no_arm_with_sign
AM = 186_light_standard_top_arm_with_sign
AN = 187_light_standard_mid_arm_with_flag
AO = 191_car_sedan
AP = 192_car_van
AQ = 193_car_pickup
AR = 194_car_truck

Figure 2: List of All objects in the database

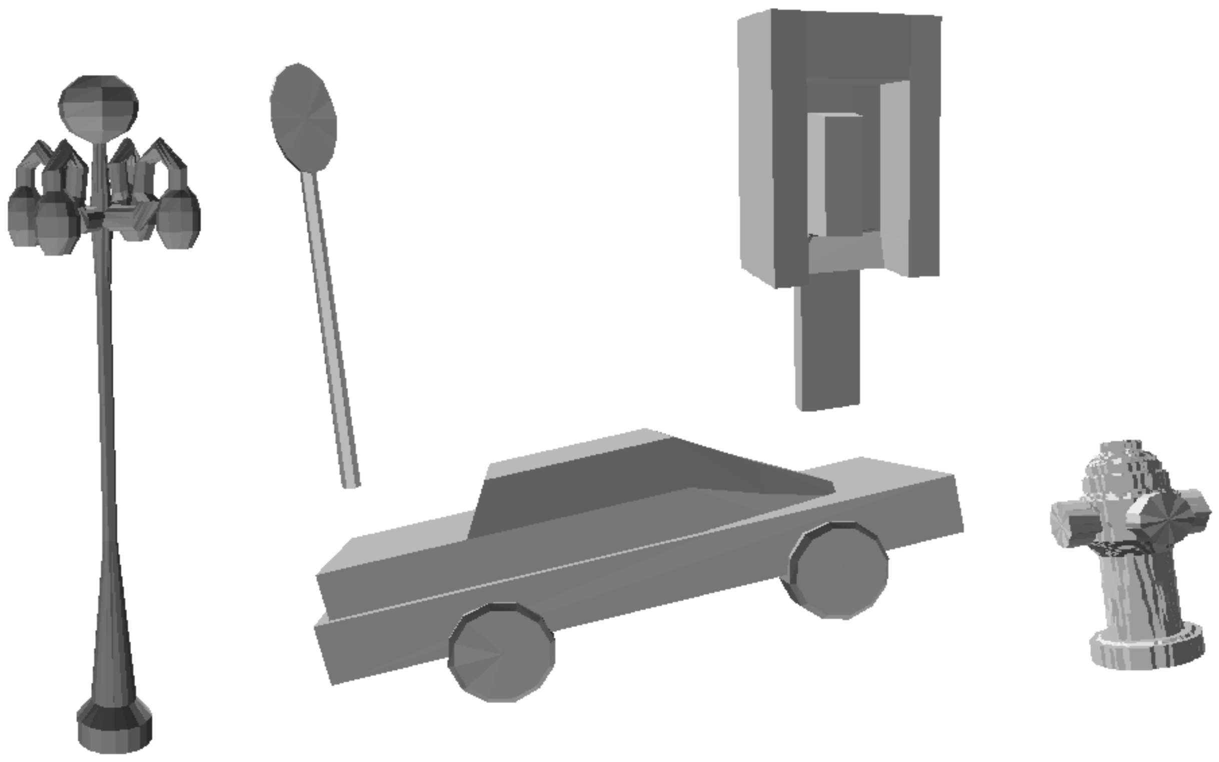


Figure 3: A subset of the triangular mesh models in the database. Here is displayed (from left to right) a lamp post, a stop sign, a sedan, a telephone booth, and a fire hydrant. These meshes were displayed using the "mshview" program mentioned in III A.

IV. Procedures for Generating Feature Vectors

A. ICP Point Cloud to Mesh Alignment

In order to get a score for a point cloud against a model in the mesh database, the point cloud must first be lined up to that model as closely as possible. Iterative Closest Points (ICP) is a known technique to bring different objects close to each other using some distance metric [5]. In this case, a point cloud is aligned to a mesh model using Euclidean distance, where distance is calculated between each point and the closest point on the mesh (accelerated with a spatially partitioned mesh search tree). At each step, an optimal transformation is calculated to bring the point cloud as close as possible to the mesh with this overall distance metric between all of the points. Then another optimal transformation can be calculated, and this process is repeated until convergence. In this manner, the point cloud "snaps" into place at some local minimum of Euclidean distance. Note that at each step, the optimal transformation is calculated between the two point sets: the point cloud and the set of closest points on the mesh to each one of the points in the point cloud.

Like most iterative techniques, ICP isn't guaranteed to find a global min, but it will likely find a good local min in closeness for the alignment. Note also that good initial alignments with Principal Component Analysis techniques (using SVD) can not only help the convergence rate, but they also tend to end up at good local mins. Currently the program starts with the principle components and the centroid as a guess for the initial alignment. It then tries all flips (either the positive or negative direction of the vector) and all permutations of the principle axes. Out of this set of transformations, it picks the final transformation that converged to the smallest distance. This scheme maximizes the chances that a good overall alignment is found.

In this application 100 random points are sampled from the point cloud at the beginning, and this is taken as a representation of the point cloud throughout the alignment. Since the program goes through this resampling and re-aligning process for so many different permutations and flips, this smaller subset is necessary to make the speed of the program feasible, and also to make the process uniform over all different examples (i.e. some point clouds may contain more samples than others).

B. 6DOF vs 3DOF Alignment

i. Tradeoffs between 6D and 3D

Now that the general algorithm for ICP has been established, it is necessary to devise a scheme to compute optimal transformations between the point cloud and the closest points on the mesh at each iteration. One initial scheme could be to optimize using 6 degrees of freedom; to come up with the best affine matrix to take the points to the mesh. The linear transformation matrix would look like this:

$$\begin{pmatrix} r_{12} & r_{12} & r_{13} & T_x \\ r_{22} & r_{22} & r_{23} & T_y \\ r_{32} & r_{32} & r_{33} & T_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Equation 1: The 6DOF matrix that's computed at each step of ICP

On the other hand, perhaps it would be better only to allow for XY translation and rotation about the z-axis, for 3 degrees of freedom total. That matrix would look as follows:

$$\begin{pmatrix} r_{12} & r_{12} & 0 & T_x \\ r_{22} & r_{22} & 0 & T_y \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Equation 2: The 3DOF matrix that can be computed at each step of the ICP

There are clear trade-offs between choosing either 6DOF or 3DOF alignment during ICP. On the one hand, 3D alignment is more physically realistic. In real life, highway signs are always oriented on the ground; they never go at a funny angle diagonally from the ground, but 6D would allow this. On the other hand, using 3D falls into danger if the point scans and the mesh models that would normally be very similar to each other are on different hills. Here are a few examples illustrating this point:

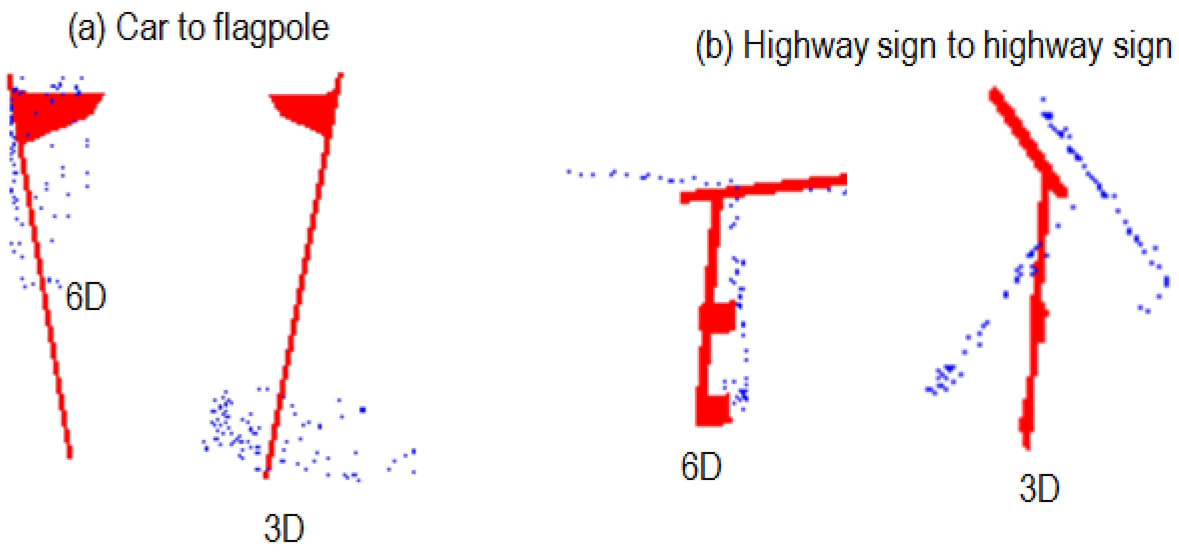


Figure 4: The trade-offs between using 6D and 3D alignment

In figure 4 (a), the potential benefits of using 3D alignment are seen. In this example, ICP is used to align a point cloud of a car to a model of a flagpole in the database. Using 6D alignment, the car is lifted up off of the ground and rotated facing the ground, which is a physically impossible orientation. As such, it gets an erroneously good match. Using only xy translation and z-rotation, on the other hand, the car remains on the ground and gets centered at the flagpole. Since most of the points are much further away from the flagpole, it will (correctly) get a lower score as not matching well with the flagpole.

In figure 4(b), the potential dangers of using 3D alignment are seen. In this example, a scan of a slightly different highway sign is aligned to a model of a highway sign in the mesh database. However, it appears as if these two highway signs were taken on hills with slightly different normals, so they are rotated in different directions not purely about the z-axis. 6D is able to get them back into nearly the proper orientation, but 3D cannot. So it receives an erroneously low score for 3D alignment even though it is of the same object class as the model in the database.

Overall, therefore, 6D is expected to be better for aligning models of the same class to each other, but it risks being too liberal about aligning very different objects. 3D is expected to have fewer matches but good matches in most cases, but is prone to problems like the one demonstrated in figure 4b. Both will need to be tested to assess this trade-off.

NOTE: Regardless of whether 6D or 3D is being used, scale is disabled, because it makes no physical sense to scale objects. If scaling were enabled, this could, for instance, scale a fire hydrant to match up with a tall flagpole, which we certainly wouldn't want.

ii. Computation of 3DOF Using 6DOF and Ground Position Info

There is a way to reduce 3DOF alignment as described above to 6DOF alignment. If the points from the sampled point cloud and the closest point set on the mesh are both projected on the XY plane (i.e. their Z-coordinate are set to zero), then 6DOF alignment can be used to accomplish solely XY translation and rotation about the z-axis. This is because an optimal 6D transformation would never take the points off of the plane (no z-translation), and it would never rotate the planes so that they were no longer parallel (no rotation other than about the z-axis). One other caveat here is that the point correspondences to minimize distance are still always taken before that projection.

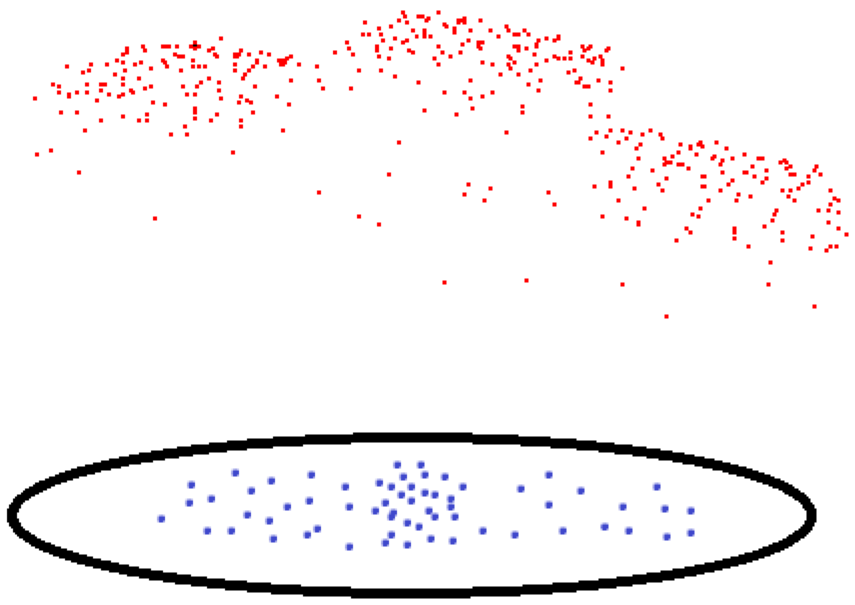


Figure 5: Projecting a 3D point cloud onto the XY plane to do 3DOF alignment

One point of caution occurs with determining the actual z translation of sensible alignment now that this is no longer taken care of by the optimal transformation. One way to do this would be to use the bounding box of both the mesh model and the point cloud to estimate the ground position of both of the objects. However, this is prone to error for a variety of reasons, not the least of which is noise. Instead, I take advantage of information that comes from the segmentation phase; namely, each segmented 3D object has a ground position reported along with it.

Unfortunately, none of the mesh models have ground positions recorded. However, as I mentioned before, each mesh is modeled off of one of the 3D object scans. So if I manage to find the scan off of which each mesh was modeled, I can recover the ground positions of all of the models and record them in a text file. To do this, I go through all of the scans of a particular class and find the one (without alignment) which is closest to the mesh model. I then assume that the ground position of the model is the ground position of that closest scan. Once I have the ground positions of both the object scan and the mesh model, I translate the z coordinates of all of the points of the point cloud scan up by [mesh ground position – scan ground position] before the 3D alignment is performed. This ensures that they both have the same ground position during alignment.

C. Giving a Score to an Alignment

Once the best alignment has been found between a point cloud scan and a model in the database using either 6DOF or 3DOF, a score needs to be computed to make a new element of the feature vector. In this way, the feature vector for each point cloud is a **42-dimensional descriptor**, where each element of the descriptor reports how well the alignment worked for the corresponding model in the database. Motivated once again by the problem of occlusion, this number is chosen to be the fraction of points that fall within some distance, **epsilon**, of the mesh model. A point's distance is calculated as the distance between the point on the scan and the closest point on the mesh. Thus, the number ends up being a score on the interval [0, 1], where 1 reports a good match and 0 reports a bad match.

Varying epsilon has an enormous impact on the feature vector and the overall flavor of the classification. A large epsilon means that many points are accepted, leading to a better score. Correspondingly, a small epsilon means that fewer points are accepted. Figure 5 shows one example of varying epsilon from 2 meters, to 1 meter, to 40cm.

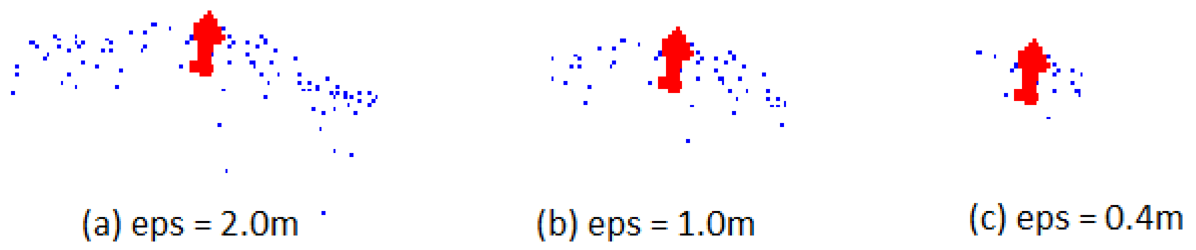


Figure 5: Results of varying epsilon aligning a car to a fire hydrant with 3DOF.

Since epsilon is a variable, there should be some global optimum for epsilon for classification. It is unclear what the best choice of epsilon is, however, because of the tradeoff between being too accepting (large epsilon) and too discriminative (small epsilon). An epsilon of 0 will accept nothing, while an epsilon of infinity will accept everything. A small epsilon is very discriminative and will have mostly zero entries in the feature vector, but the nonzero entries really mean something.

D. A Few Bugs Encountered During Development

i. Super slow alignment bug

This wasn't so much of a bug as it was a lack of oversight during the ICP process. At first, I was using every point in the scan during the alignment, which was overkill. Typical alignments with all flips and permutations during ICP would take about 20 minutes each, which obviously would not be feasible for the 42 alignments needed for each of 1163 object scans. To remedy this problem, I decided to randomly sample a subset of only 100 points for each scan, where the random part was to ensure equal distribution (same idea behind the D2 descriptor presented in [4]). The alignments may be a bit coarser overall, but this step is essential for making computation feasible.

ii. Erroneous Good Scores

As I was getting initial results I noticed that some really good scores were being reported for some very non-similar object scans and mesh models. This was leading to some pretty abysmal classification results (only about 12% correct at first). After a little digging, I discovered that for some points, the closest point on the mesh was set to be the original query point (not a point on the mesh). This was bad, because then the distance of these points to the mesh would be calculated as zero, leading the program to erroneously believe these points were extremely close to the mesh (and giving the scan a better score). It took a while to discover why closest points were being returned as the original query point in some cases, but eventually it was discovered (with the help of Tom Funkhouser) that some faces had zero area, which violates an assumption of the closest points function in the R3MeshSearchTree. My guess is that this must have occurred during the mesh construction phase where the person who created them had to delete a few faces, and it was most convenient for the program to keep the data in this format. To fix this problem, the meshes were patched up and all faces of zero area were eliminated. Alignment accuracy shot up after this.

NOTE: This bug was also affecting alignment before since closest points also need to be computed at each step of ICP

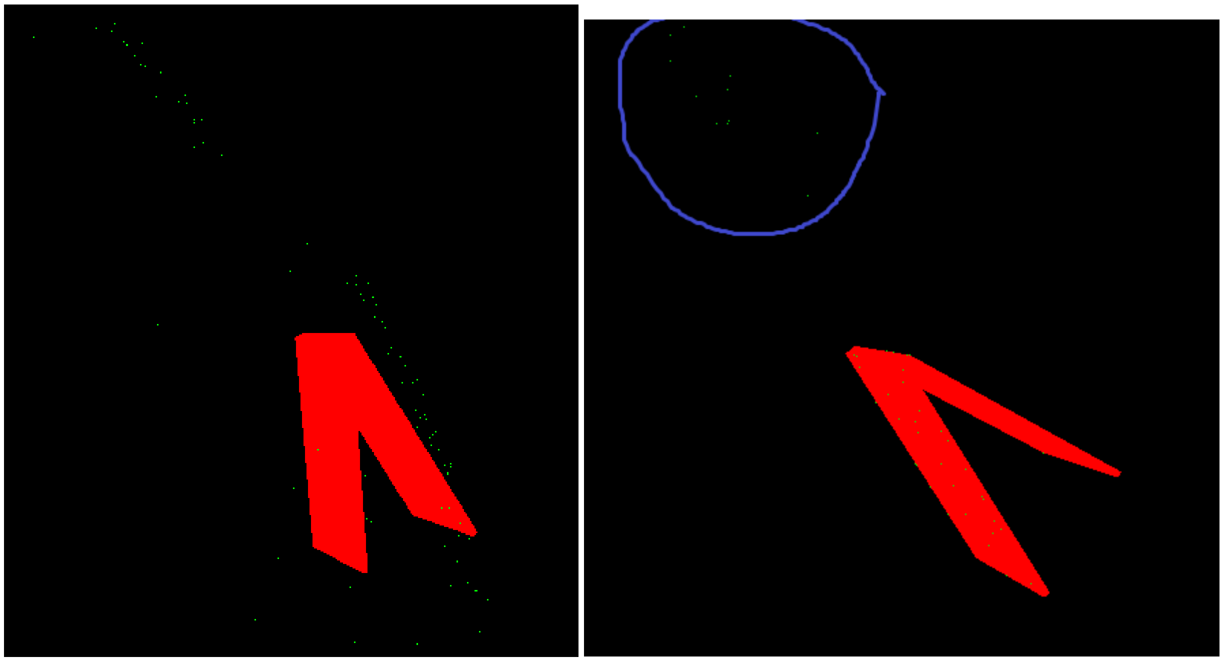


Figure 6: The closest points bug. On the left is shown a mesh model (in red) and an aligned object scan (in green). On the right is shown what the program calculated to be the closest points on the mesh. Note how many of the closest points did fall on the mesh, but how there is a cluster of points (circled in blue) that remained the original query point

V. Testing for Success

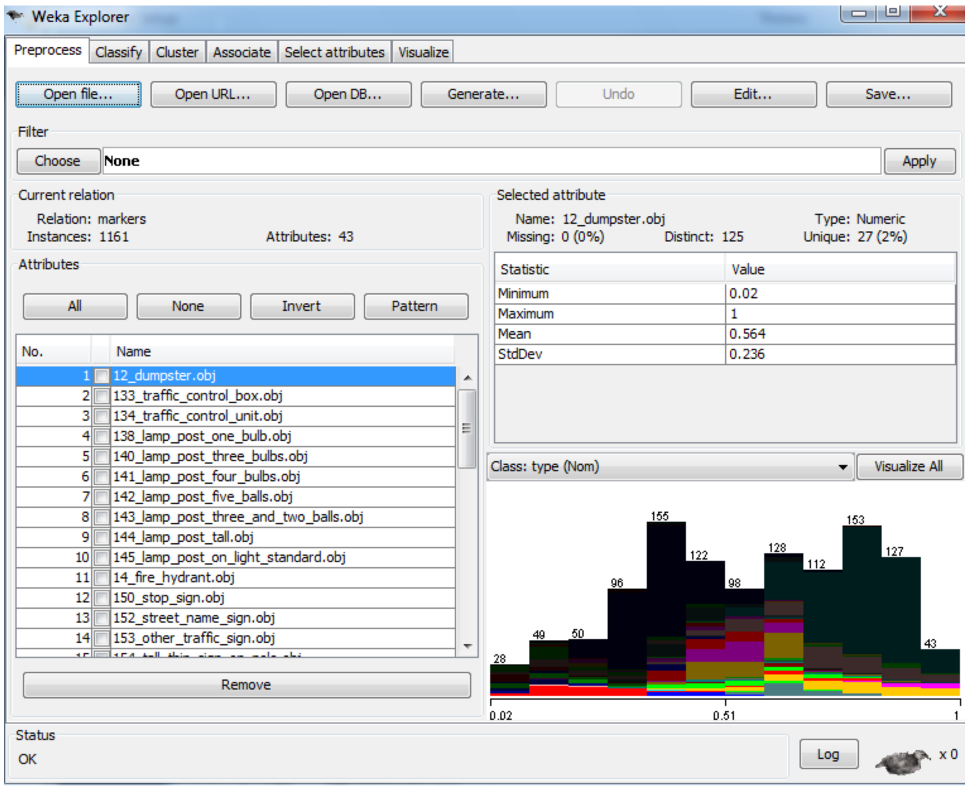
A. Batch Testing and Caching for Future Tests

To test this whole scheme for success, the 42-dimensional feature vector (one dimension for the score of a scan against each model in the database) is computed for each of the 1163 object scans. The alignment is the bottleneck of this whole process, so the results of the alignments are cached for future test. That is, a new file is created to contain the aligned point cloud after alignment is complete between a point cloud and a database model. The alignments are not likely to change much from test to test (only subject to random sampling of the points). Therefore if the alignment has already been done once, these cached results can be used instead of re-aligning when I want to try a new epsilon. Finding closest points still takes time, but not nearly as long as repeating the alignment from scratch. To put numbers on it, the batch tests take about 36 hours for 6D and 3D starting from scratch, but they only take 1 hour using pre-computed alignments.

The batch testing was performed for both 6D and 3D on all of the 1163 objects. The cutoff value for closest points, epsilon, was varied from 0.05m, to 0.1m, to 0.2m, to 0.4m, to 0.8m.

B. Machine Learning Using Weka

The focus on this research is not on machine learning techniques, but rather computing feature vectors that are generally discriminative between object classes. Therefore, I made use of pre-existing software to do classification of the 3D point scans into different categories once the feature vectors had been computed. Our team uses the open-source machine learning program called "Weka" to do classification. The results need to be put into "arff file" format, where the feature vector components are defined first and then an arbitrary number of labeled (or unlabeled) examples are defined with the feature vector components filled in.



Filter: None

Current relation
Relation: markers
Instances: 1161
Attributes: 43

Selected attribute
Name: 12_dumpster.obj
Type: Numeric
Missing: 0 (0%)
Distinct: 125
Unique: 27 (2%)

Statistic	Value
Minimum	0.02
Maximum	1
Mean	0.564
StdDev	0.236

Attributes

No.	Name
1	12_dumpster.obj
2	133_traffic_control_box.obj
3	134_traffic_control_unit.obj
4	138_lamp_post_one_bulb.obj
5	140_lamp_post_three_bulbs.obj
6	141_lamp_post_four_bulbs.obj
7	142_lamp_post_five_balls.obj
8	143_lamp_post_three_and_two_balls.obj
9	144_lamp_post_tall.obj
10	145_lamp_post_on_light_standard.obj
11	14_fire_hydrant.obj
12	150_stop_sign.obj
13	152_street_name_sign.obj
14	153_other_traffic_sign.obj

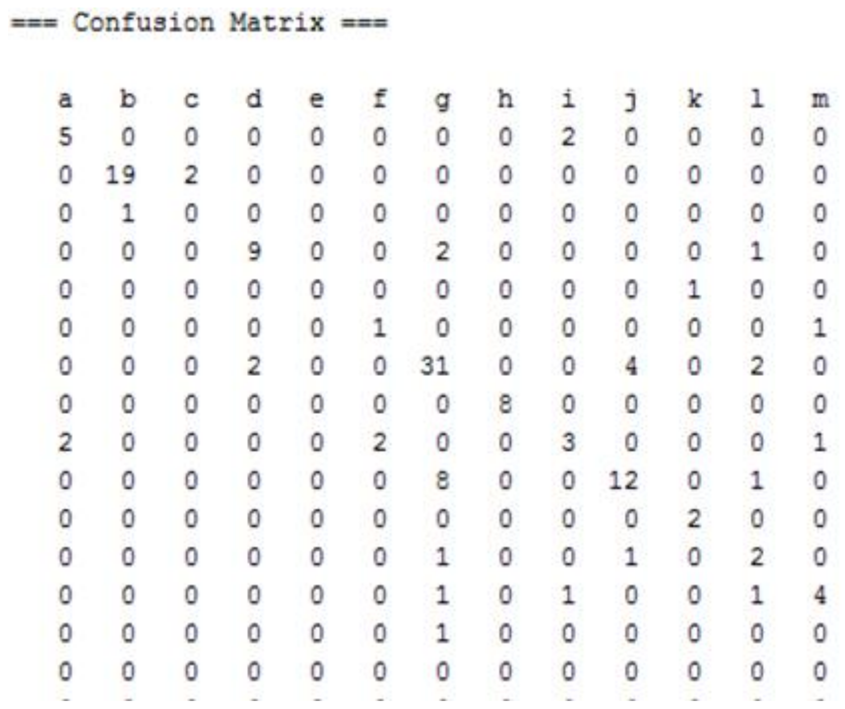
Class: type (Nom)

Status
OK

Figure 8: Weka GUI showing the histogram of values for matching the "dumpster" mesh model in the database over all examples

Once the data is in the proper format, any number of advanced analysis tools and classification techniques can be performed to test the effectiveness of different features. I decided to use the J48 decision tree classifier to run the main classification tests, and the Naïve Bayes classifier to compute probabilities that were used to create precision-recall graphs. For each of these tests, "cross-validation" was used on the training data to test classification. That is, a random 10% of the data was taken out and trained against the remaining 90%, and this was repeated 10 times.

Mistakes often happen during classification. To visualize the overall correctness of the in more detail than a mere percent accuracy, an nxn "confusion matrix" (where n is the number of object classes, 44 in this case: see III B) can be drawn, where the rows represent a particular class and the columns represent classifications of objects within that class. As such, diagonal entries represent correct classifications, and off-diagonal entries represent "confusions," or mistakes made by the classifier.



```
=== Confusion Matrix ===

   a   b   c   d   e   f   g   h   i   j   k   l   m
   5   0   0   0   0   0   0   0   2   0   0   0   0
   0  19   2   0   0   0   0   0   0   0   0   0   0
   0   1   0   0   0   0   0   0   0   0   0   0   0
   0   0   0   9   0   0   2   0   0   0   0   1   0
   0   0   0   0   0   0   0   0   0   0   1   0   0
   0   0   0   0   0   1   0   0   0   0   0   0   1
   0   0   0   2   0   0  31   0   0   4   0   2   0
   0   0   0   0   0   0   0   8   0   0   0   0   0
   2   0   0   0   0   2   0   0   3   0   0   0   1
   0   0   0   0   0   0   8   0   0  12   0   1   0
   0   0   0   0   0   0   0   0   0   0   2   0   0
   0   0   0   0   0   0   1   0   0   1   0   2   0
   0   0   0   0   0   0   1   0   1   0   0   1   4
   0   0   0   0   0   0   1   0   0   0   0   0   0
   0   0   0   0   0   0   0   0   0   0   0   0   0
```

Figure 9: Part of a confusion matrix reported after one of the batch classifications. Entries along the diagonal are correct classifications, while off-diagonal entries are confusions

C. Precision Recall Graphs

Another common visualization of classification effectiveness is the "precision-recall graph." Precision and recall are two numbers that summarize information about true and false positives. Training examples are taken back from the set in order of most likely to least likely according to the classifications. At any point taking back models, the recall is the fraction of the number of models of a particular class that have been returned, to the total number of models in that class. The precision is the ratio of the number of models that are of the correct class to the total number of models that have been returned. An inverse relationship is normally expected between the two variables; that is, a higher recall normally leads to a lower precision (recall is normally plotted on the x-axis). In other words, by the time all of the objects of the class have been recalled, it is likely that many false examples have also been recalled.

I wrote a program in Java to generate precision recall curves, and then I plotted them in Matlab. My Java program takes as input the object class probabilities of each 3D point scan, and it parses them into a matrix of probabilities. Then within each of the 44 classes, the objects are sorted from most likely to least likely. Going from most to least likely, every time an object of the correct class is encountered, the recall is incremented and the precision is recalculated, leading to a new point on a precision recall curve.

Since each class has a different number of examples, the number of points on the precision-recall graph varies. Classes with more examples have more to recall, and thus have more points along the recall axis (finer resolution precision-recall curve). This is a challenge, though, because I would like an overall precision-recall graph, not just one by-class (because Weka already does per-class precision recall graphs). To accomplish this, I average all of the precision-recall graphs over the 44 classes, and I simply use linear interpolation to fill in the spots on the lower resolution graphs for classes that have fewer examples.

VI. Results

A. Batch Results Varying Epsilon Using J48 Decision Tree

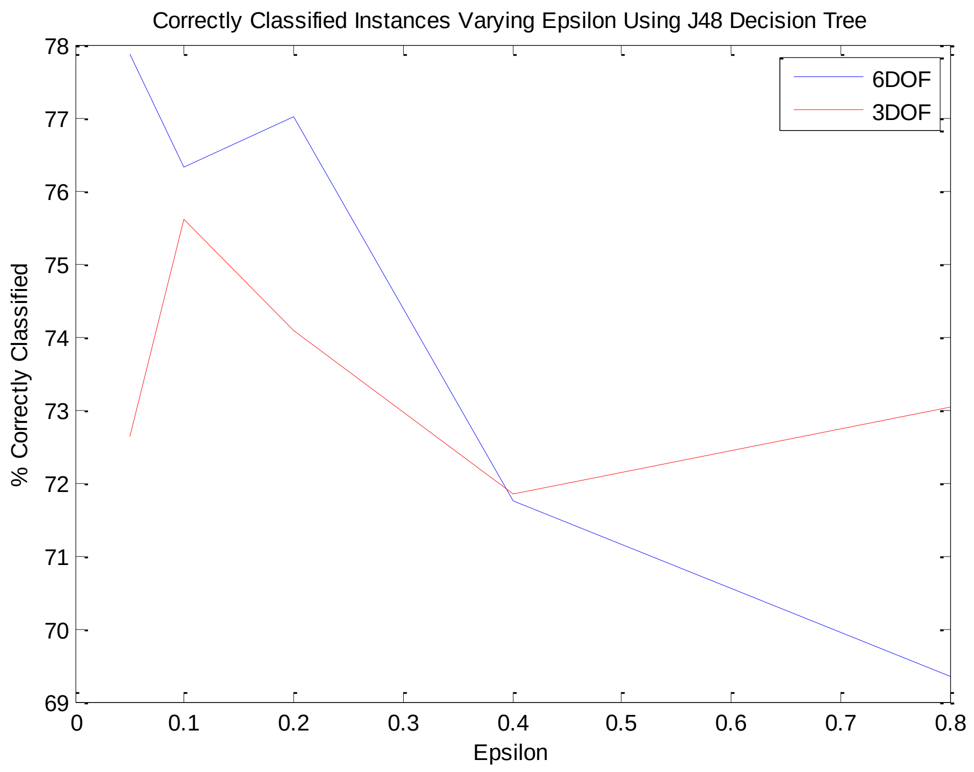


Figure 10: A plot of the percent correctly classified instances for both 3DOF and 6DOF alignment varying the cutoff distance for closest points, epsilon

B. Precision Recall Graphs

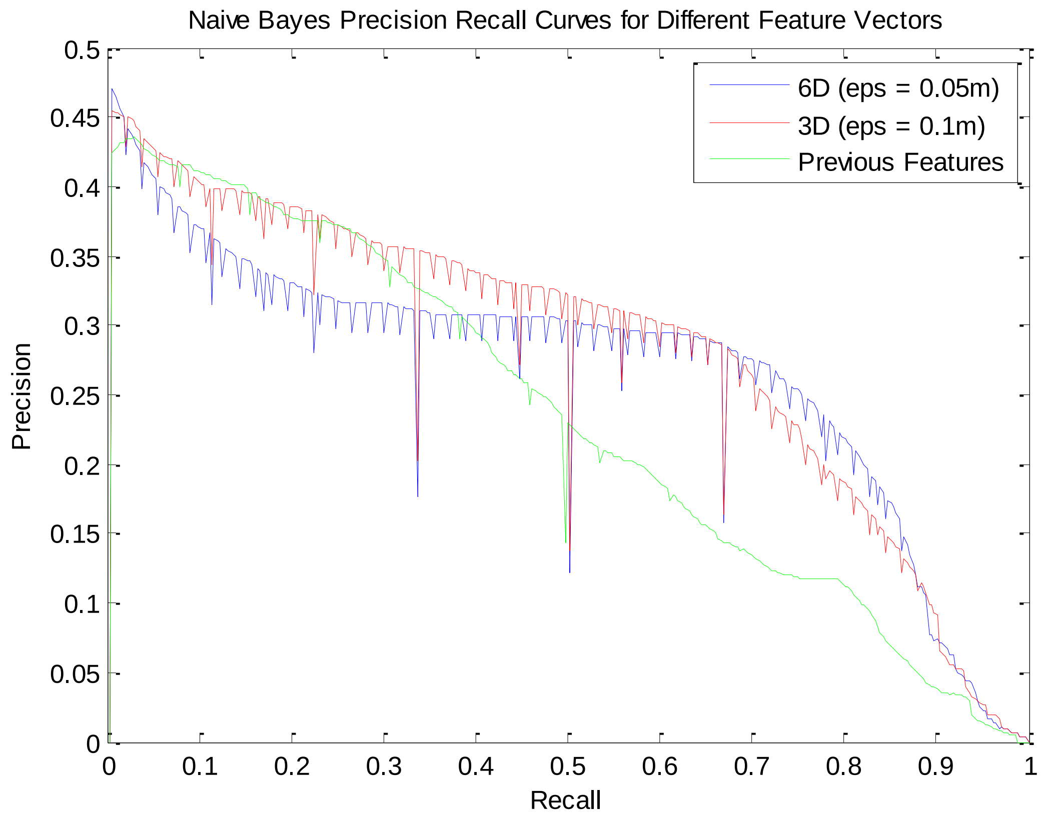


Figure 11: A plot of the precision recall curves generated by my Java program for 6D and 3D blue and red, respectively) using the epsilon that gave the best classification for the J48 tree, plotted against the precision-recall graph of the classification that used the roughly 1200 features from previous research (in green)

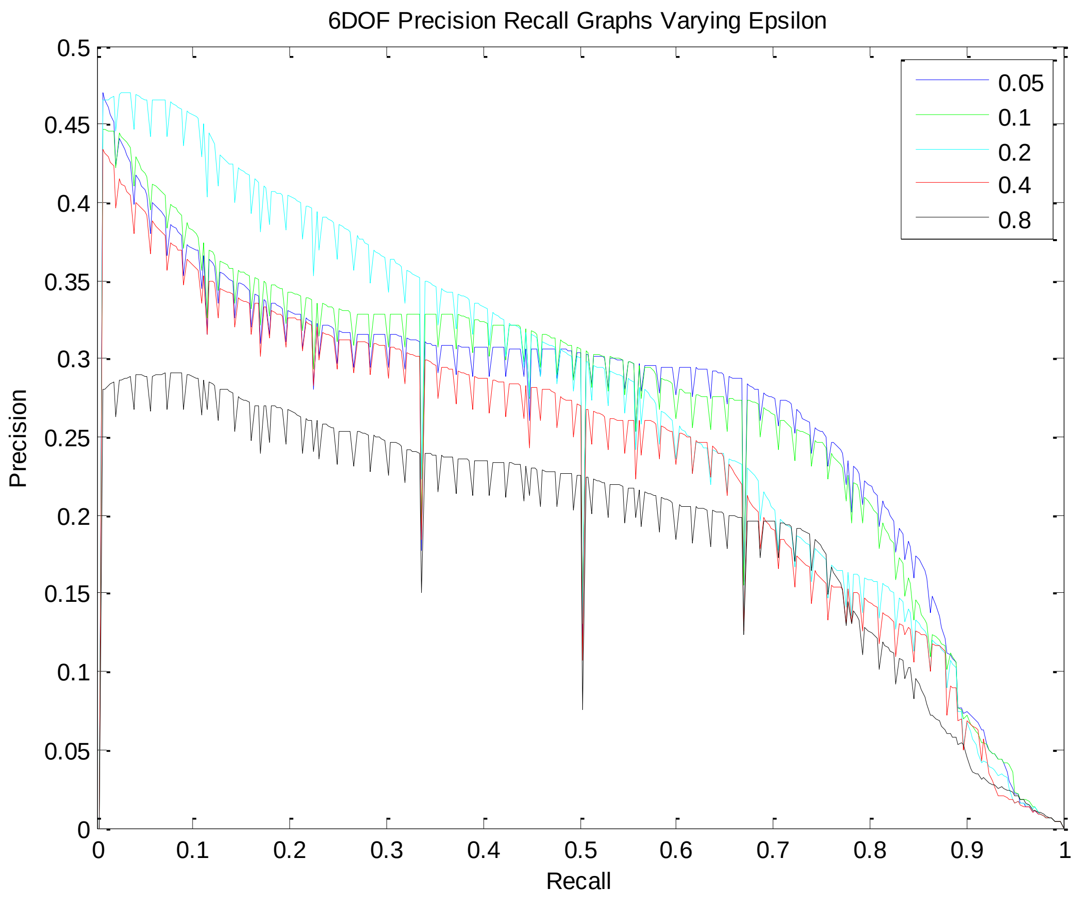


Figure 12: A plot of the precision-recall graphs for 6D using Naïve Bayes varying the cutoff distance epsilon

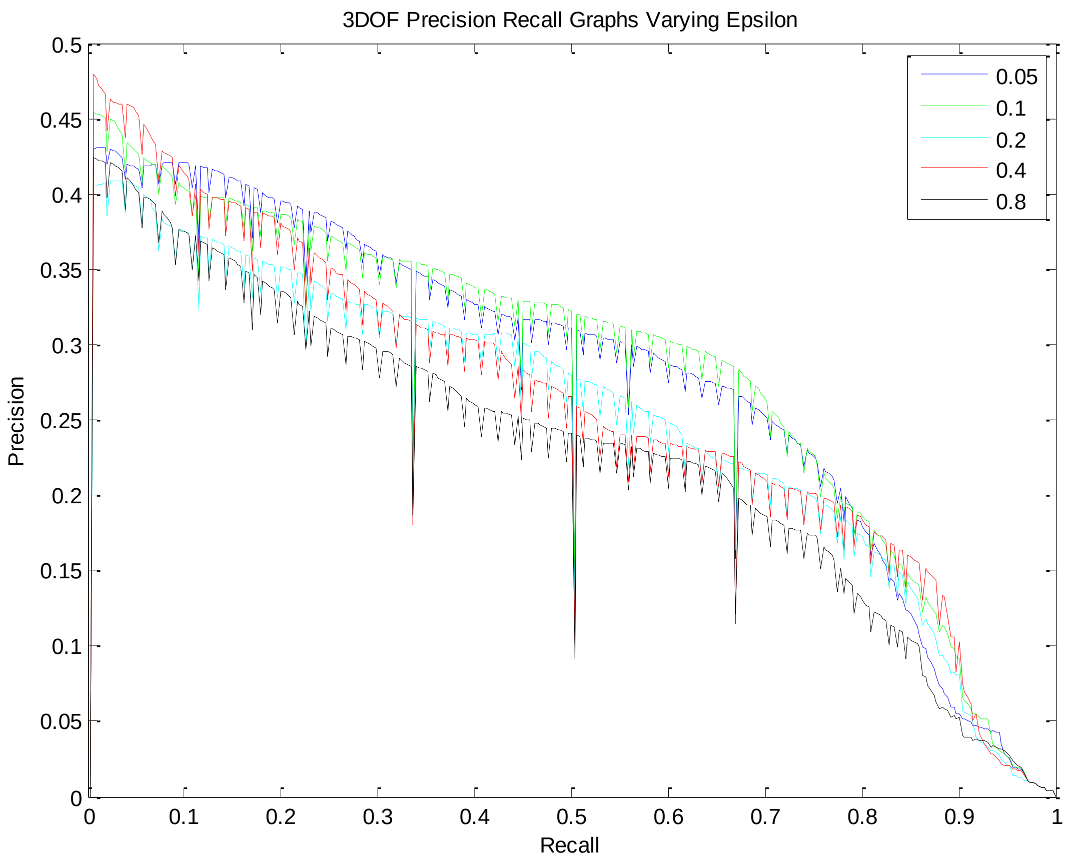


Figure 13: A plot of the precision-recall graphs for 3D using Naïve Bayes varying the cutoff distance epsilon

C. Common Confusions

What the percent accuracy fails to show are the classes for which the classifier failed, but for which the mistakes are very logical, choosing classes that are very similar to the actual class. Here are a few examples of the most common inter-class confusions found in the confusion matrix:

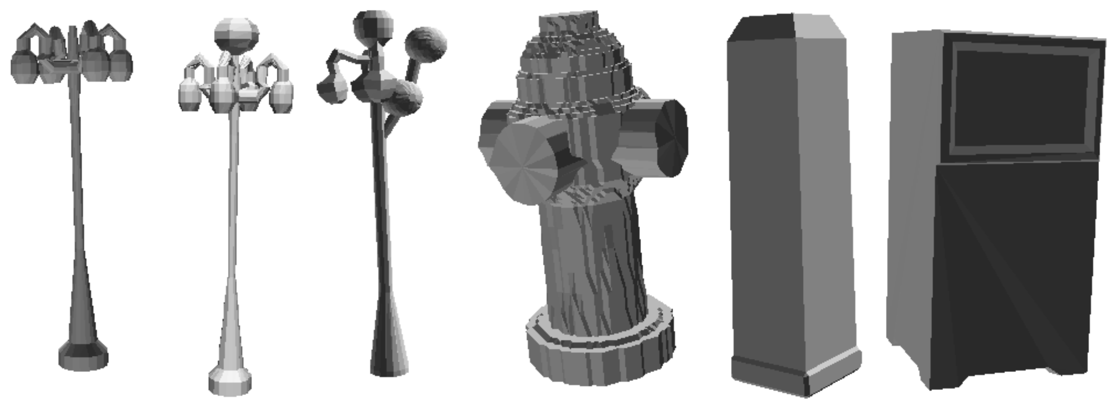


Figure 14: On the left, lamp posts with different numbers of bulbs were often confused with each other. On the right, fire hydrants, short posts, and mailboxes were sometimes confused with each other

VII. Conclusions / Further Avenues

Overall the results were quite good for this new classifier. At best using the J48 tree, both 6DOF and 3DOF alignments give rise to high 70% range correct classifications. 6DOF performs slightly better than 3DOF overall; perhaps the trade-off with objects being oriented on hills made 3DOF less likely to perform well. However, 3DOF alignments can be computed more quickly than 6DOF alignments if I treat the problem independently of 6DOF alignment instead of making the reduction to 6DOF, and they are still pretty accurate. Thus, even if 3DOF is a bit less accurate, choosing a good epsilon may turn it into a quickly-computable, low dimensional classifier that works well.

Another encouraging aspect of this research is that not only is the percent of correctly classified objects high, but many of the objects that are incorrectly classified are put into similar categories. It is easy to imagine why, for example, the different lamp posts got confused and how a fire hydrant may be confused with a short post if the laser scanner didn't pick out the finer details of the concave regions of each of these objects.

An interesting point to note about the precision recall graphs is that for low epsilon in both 6DOF and 3DOf classification, the precision is much higher overall for recall between 0 and 0.5. This makes sense when the meaning of a small epsilon is considered. A small epsilon forces most of the entries in the feature vector to be zero; only those models that are very similar to the point scan in question get nonzero scores. Therefore, when the classifier ranks probabilities for the objects, the objects are likely to be very similar to a particular class, leading very similar objects to be recalled towards the beginning for small epsilon.

What needs to be explored more about this particular shape descriptor scheme for city objects is the extent to which it impacts classification when it is combined with other features that have been used in the past. In the precision recall graphs, it appears to do just as well, if not better, than feature vectors from previous schemes. Additionally, the J48 tree had an accuracy of 78.2% for the previous feature vectors, which is not significantly better than the new scheme. It is possible, therefore, that the 1200 feature vectors can either be replaced by the much lower-dimensional 42D feature vector from my work, or that they can be combined with the new features to improve classification.

All of this research is headed towards a system that will automate the process of labeling city objects, since this task would be monstrous for a user to go through and do by hand. This feature vector and the others proposed before are supposed to give the program good guesses to start with. Then, the user can give feedback on which ones were correct in some small, local region (perhaps just one city block), and adaptive reclassification can take place. Aleksey Boyko is currently working on the adaptive reclassification, which is a very difficult problem.

VIII. Software Usage

A. Aligning a Single Point Scan to a Mesh

This can be done for testing. The syntax is

```
singleBlobMeshAlign <mesh> <blob> <3D?> <eps>
```

This program assumes that there is a file, ZCoords.txt, in the current working directory that has all of the ground positions of the mesh models. This program will output the result of alignment to the point cloud data file “out.xyz”, and it will launch simpleModelFittingViewer to view out.xyz against <mesh>

B. Visualizing Random Alignments

I created a program in Java to go through and pick out a random model and a random mesh and to align them to each other (it calls the C++ programs from within Java). The program then uses the “Java Robot” class to zoom in on each object individually and then on the results of the alignment, and to take screenshots of all of these items. Then, the original objects and their aligned result are shown side-by-side in a web page This program sped up testing of the alignment and it gave me a rough idea if things were working without the tedium of having to come up with my own examples and type out the long commands by hand in the terminal.

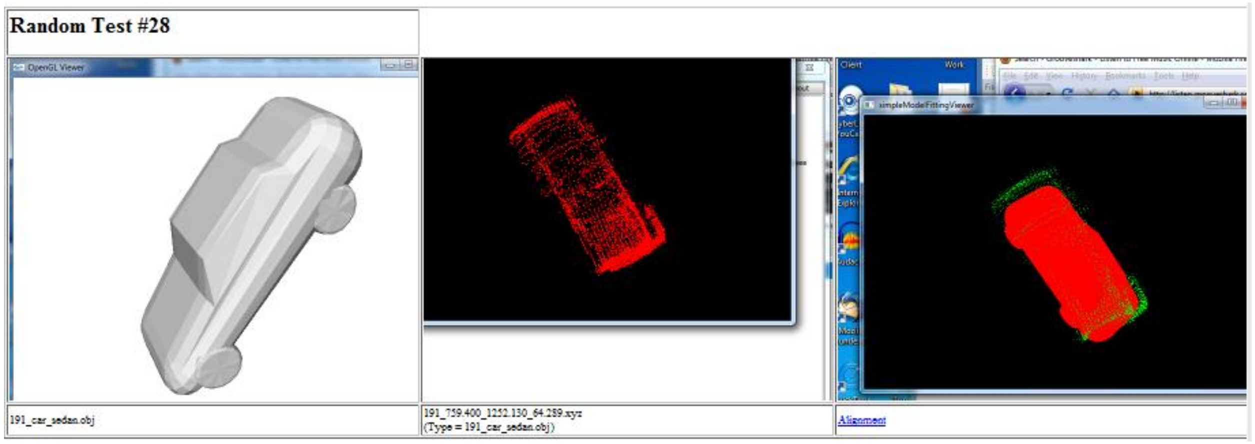


Figure X: One of the random alignment tests from my Java automatic tester

C. Running Batch Tests

This program is run to generate the arff files for Weka on all 1163 blobs. It is normally run under Linux on the CS clusters but it is cross-platform. The syntax is as follows:

```
blobMeshAlign <mesh directory> <blob directory> <results
directory> <arff file> <3DOF?> <groundpos file> <eps>
<maxpoints>
```

- mesh directory: the location of all of the 42 models in the database (stored in obj format)
- blob directory: the location of all 1163 city object scans (training data)

- results directory: a directory which holds the cached results of alignment of the training data. If an alignment has already been done it has been saved here, and the batch tests will use this instead of re-computing the alignment
- arff file: the output file used by weka
- 3DOF?: This should be 1 if 3 degrees of freedom are being used and 0 if 6 degrees of freedom are being used
- groundpos file: A file that holds the ground coordinates of all of the models in the database (used for 3D alignment as explained in IVBii)
- eps: The cutoff distance epsilon for closest points
- maxpoints: The maximum number of randomly sampled points from each 3D point scan. It is always 100 in these tests

D. Generating Precision-Recall Curves

There is a script in Java that can take a probability matrix from the Weka GUI and generate precision-recall curves. This process has been described in detail in part V(c). This program only needs one argument: the name of a text file that holds the matrix in the format that it was presented in the Weka GUI. The program parses that matrix and gets rid of characters that Weka uses to denote different events, such as incorrect classifications (+) and highest probabilities in class (*). It is also able to determine the correct class of each example from that matrix, which is an essential bit of information for generating precision-recall curves.

IX. References

[1] Boyko, Aleksey. "Lidar City Project instructions." *Computer Science Department at Princeton University*. Web. 26 Feb. 2010. <http://www.cs.princeton.edu/~aboyko/city_doc/mainInstructions/>.

[2] Funkhouser, Thomas, Aleksey Govolinsky, and Vladimir G. Kim. "Shape-based Recognition of 3D Point Clouds in Urban Environments." Print.

[3] Funkhouser, Thomas, and Aleksey Golovinsky. "Min-Cut Based Segmentation of Point

Clouds." Print.

[4] Funkhouser, Thomas, Patrick Min, Michael Kazhdan, Joyce Chen, Alex Halderman, David Dobkin, and David Jacobs. "A Search Engine for 3D Models." Print.

[5] Yaroslav Halchenko, Yaroslav. "Iterative Closest Point (ICP) Algorithm. L1 solution. . ." Web.

X. Acknowledgements

- Tom Funkhouser: Faculty adviser, provided codebase, algorithm ideas/advice, and extreme debugging help
- Aleksey Boyko: Graduate student in computer science, started city project and advised me on many technical points along the way
- Cindy Menkes: Undergraduate Program Coordinator for electrical engineering; helped keep me on top of my work this semester by providing intermediate deadlines, and helped print my poster for the poster presentation session