Alignment-Based Descriptors for Classification of Point Clouds in Urban Environments

Fall 2010 Princeton University

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. Background and Project Aim

With the advent of laser range scanning technology in the past few years, many efforts have been made to collect massive data sets of environments in the form of point clouds. For instance, our group has obtained LIDAR scans of Paris and Ottawa done by Google. These data sets can be used in applications including photo tourism with labeling, emergency response planning, trip planning, traffic density analysis, etc. The potential is huge, though it is unreasonably tedious to expect a user to manipulate this massive point cloud point-by-point. Hence, automation techniques are sorely needed to take advantage of data sets of this form.

There are two very important processing techniques that need to be performed on the data set before it is tractable for a human to manipulate. First, we would like to do automatic segmentation of clusters of points that likely belong to the same object, so that the user does not have to separate objects by hand. A fairly successful technique has been presented in [3] for doing this using min-cut techniques by creating a nearest neighbor graph of points and encouraging points that are close to each other to stick together, and creating a penalty for including points in the perceived "background." Thus, I will take it as a given in my work that segmentation is already performed.

A second important problem is automatic classification of segmented objects, to the extent possible. Our group has done work on this specific problem in [2] already using a variety of techniques. For instance, that paper uses some of the results from a segmentation phase as a descriptor for classification. It also trains on some contextual cues about location and surrounding objects, with the idea being that it's less likely to find a car that's suspended 10m above ground level, for instance. These descriptors work well in practice, but relying on contextual cues makes them specific to this application, and the feature vectors are often very high-dimensional (which is not desirable because of the curse of dimensionality). Another recent attempt at this problem in urban environments is presented in [5] using a Markov Random Field model (conditioning on nearby points but keeping independence from far points), using the same idea about cues of nearby objects in mind. This approach also works well in practice, but the authors only demonstrate results using a very small set of classes (wire, pole, ground, scatter). Finally, classical results that are well-known in the field to create 3D object descriptors [4] could

be used, but the disadvantage here is that relying solely on 3D point information in a segmented cloud will lead these descriptors to pick up on noise and occlusion (from partial scans only having the scanner on one side of the object, most commonly).

Last spring, Professor Funkhouser and I began to investigate a new descriptor to address some of the problems with the above descriptors. This new descriptor should be designed to be occlusion-resistant, low-dimensional, and discriminative enough to differentiate between several dozen different classes of small objects that can occur on in a city environment. To address these problems, we decided to create a database of commonly-seen city objects and to align each point cloud to the mesh models in this database using Iterative Closest Points [7]. The expectation here is that occluded and noisy models will still align to the object to which they are the most similar (the same object that complete models will align to), and creating a "score" of alignment for each model leaves us with a feature vector that is very low-dimensional (simply the size of the number of models in the database, 42 in our case). We hope to both speed up the classifications and improve their accuracy in urban environments with this technique, but a careful analysis needs to be done this semester.

An initial implementation of the alignment technique last year yielded upwards of 78% accuracy solely using the alignment descriptor on manually segmented data, which is comparable to classification results with the other techniques presented in [2]. However, a much more detailed analysis of precision, recall, and other objective classification analysis tools will need comparison with the other methods.

III. Code Framework and Dataset

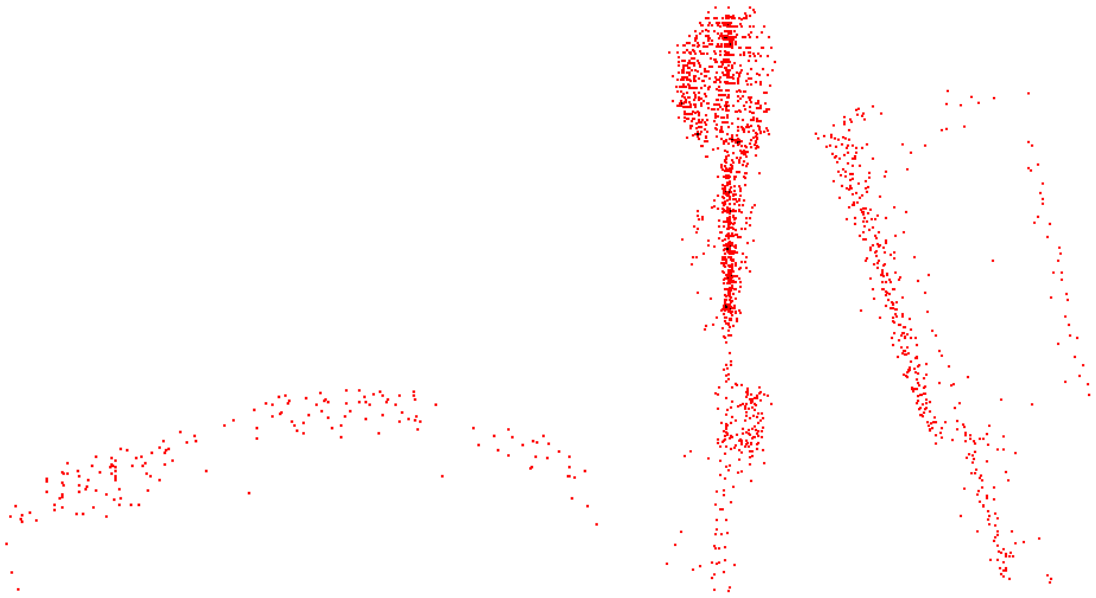
A. Development Environment

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. 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 Perl to run batch tests .

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

There are two sets of point cloud data that I use as training data in this project. The first is a set of 1161 point clouds that have been manually segmented from the city by-hand, which I call the "manual training set." This is the "cleaner" data set, in the sense that the segmentation noise is minimal (although it can still be affected by occlusion). There are 42 different classes of objects in the manual training set. The other data set was obtained by using the mincut segmentation techniques presented in [3]. This data set contains 1087 points in 39 different classes and is plagued by segmentation noise. I refer to this data set as the "mincut training set."

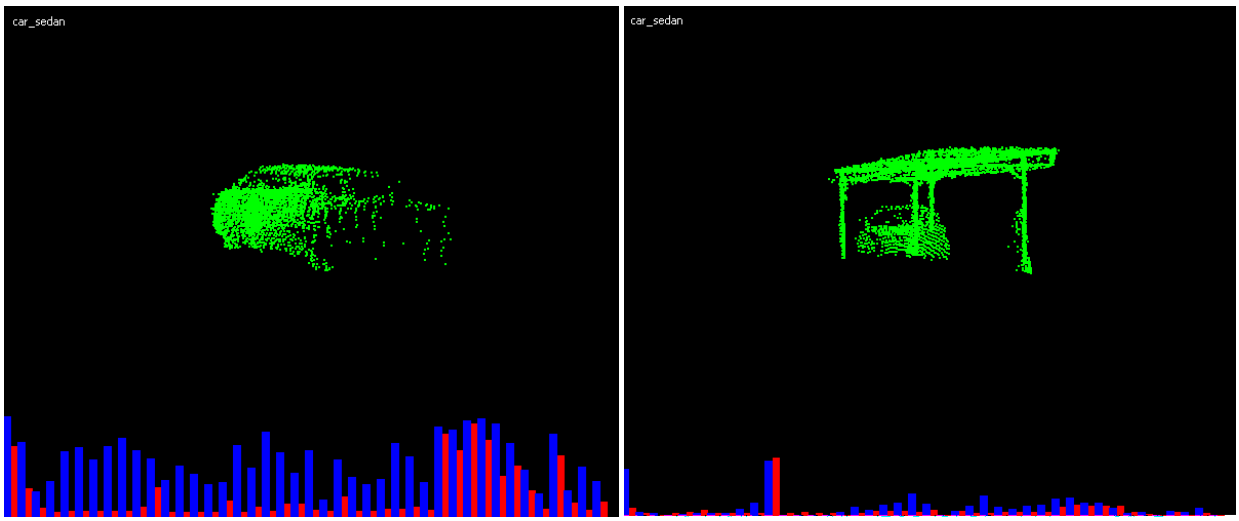


Figure 2: The left pictures shows a cleanly segmented car from the manual training set, while the right examples shows an example of segmentation noise around a car in the mincut training set

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 in the manual training set 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." Note also that after my work last year, I ensured that these models were at ground position so that 3D alignment will work (explained more later).

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 3: List of All objects in the database

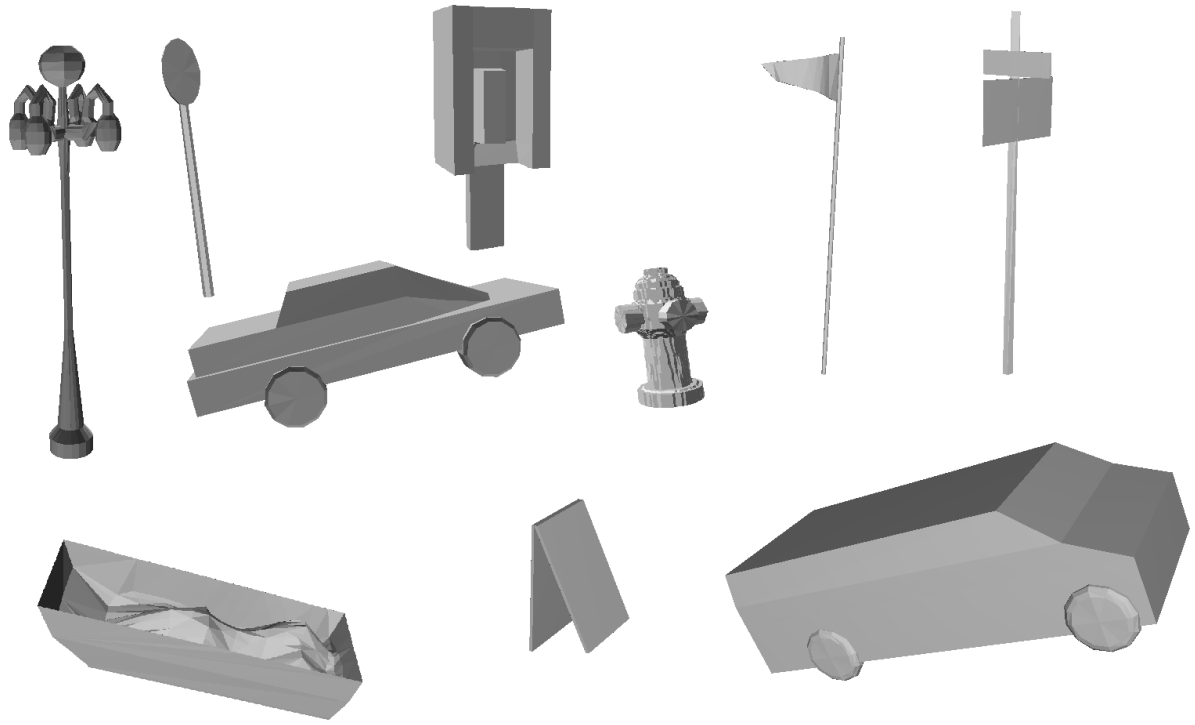


Figure 4: A subset of the triangular mesh models in the database. Here is displayed (from left to right) a lamp post, a stop sign, a sedan, a telephone booth, and a fire hydrant. These meshes were displayed using the "mshview" program mentioned in III A.

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 [7]. 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. In other words, the root mean squared distance is minimized at each iteration. 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. Here is some pseudocode to summarize this process:

Align a point cloud **P** to a model **M** with the following algorithm
1. Do PCA and find centroid to guess at initial alignment. Store 3 axes from PCA
2. Point sample **M** and create a KD tree to accelerate correspondence finding
3. For each flip and permutation of PCA axes, create an initial guess that aligns the flipped and permuted coordinate system of the mesh to the principle coordinate system of the point cloud
    A. While not converged and the number of iterations is less than 120
    B. Find closest points on point sampled version of **M** to the current transformation of **P** and create correspondences
    C. If the correspondences haven't changed since last time, then **converged**
    D. If they have changed, find new optimal transformation and go back to step A
4. Return affine transform with smallest RMSD

Convergence is detected when the same correspondences are calculated between the point cloud and the mesh two iterations in a row.

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 root mean square distance between the point cloud and the mesh. 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). One modification from the version of ICP I had last year is that in addition to randomly sampling 100 points from the point cloud, I also randomly sample 100 points from the mesh model, turning this into a point cloud to point cloud optimization problem. In practice, at the loss of some accuracy, this performs almost two orders of magnitude faster than matching the point cloud to the mesh directly, since in that case there are technically an infinite number of points on the mesh that could correspond to the point cloud.

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

3D alignment is more physically realistic. In real life, for instance, highway signs are always oriented on the ground; they never go at a funny angle diagonally from the ground, but 6D would allow for this.

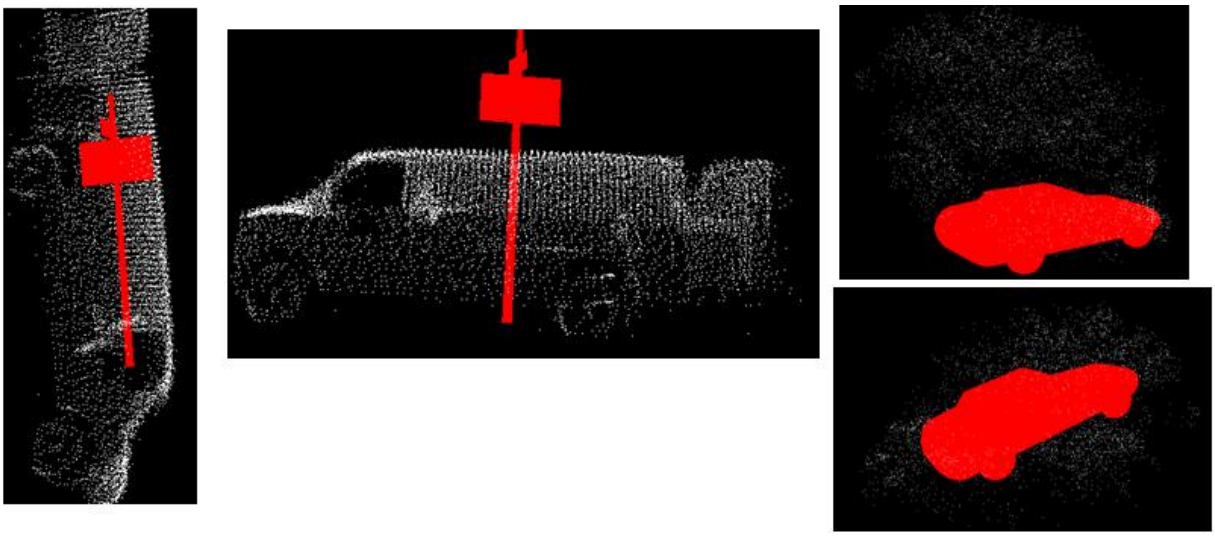


Figure 5: The occasional pitfall of using 6D alignment. The left image shows aligning a point cloud of a car to a street sign model in the database using 6D. The car is lifted off the ground and flipped so that it's facing down, and it may get an erroneously high score this way. Using 3D alignment, the car remains on the ground and is placed at the center of the sign, where it will receive a low score. On the right shows the result of aligning a point cloud of a bush to a model of a car, on the top with 3D and on the bottom with 6D. The bush remains above the car with 3D and gets a low score, but is moved to surround the car with 6D, which will give it an erroneously high score.

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 in this application

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, so that the full 3D information of the point cloud and mesh is still incorporated into the alignment.

I also discovered this semester that within the MasterCity framework, the blobs are reported along with their ground positions and their two principle components along the ground (their markers). So I don't actually need to do PCA on those point clouds to find an initial guess when I'm doing 3D alignment, nor do I need to estimate the ground position of any of the blobs, because 3D PCA is reported along with each point cloud. Also, even when I do have to do 3D PCA for the mesh models, there are many fewer flips and permutations since there are only two principle axes. NOTE also that all of the mesh models reside at their Z ground position, which I

figured out last year, so that 3D alignment can work well with no Z translation (the point cloud and the mesh should be at their proper ground position).

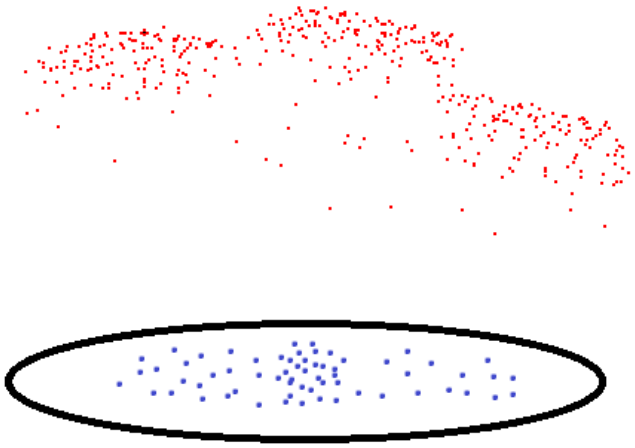


Figure 6: Projecting a 3D point cloud onto the XY plane to do 3DOF alignment

NOTE: Every once in a while because of numerical stability issues, the (3, 3) entry of the matrix (z-scale) gets optimally computed to be -1, causing the entire mesh to be flipped upside down. This won't really be detected by ICP since the meshes have been flattened before that step, so I need to explicitly look for this after each iteration. If the (3, 3) entry ends up being negative, I simply change it back to 1. This is somewhat of a hack but it works in practice, and it significantly improved results once I found it and fixed it. It seems to make the convergence time of the of the 3D alignment take much longer, however. My guess for this is that whenever the (3, 3) entry is -1, it can give rise to oscillatory behavior in the alignment chosen.

C. Giving a Score to an Alignment: PointsFrac vs MeshFrac

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 simply 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. I refer to this score as "PointsFrac" in the remainder of the writeup, since it is the fraction of points that fall within a certain distance of the mesh.

An alternative score is to calculate the dual of "PointsFrac"; that is, to report what fraction of the mesh is falls within a certain distance of the point cloud after alignment. I will refer to this as "MeshFrac." This is no longer resistant to occlusion, but it does make up for some pitfalls of PointsFrac.

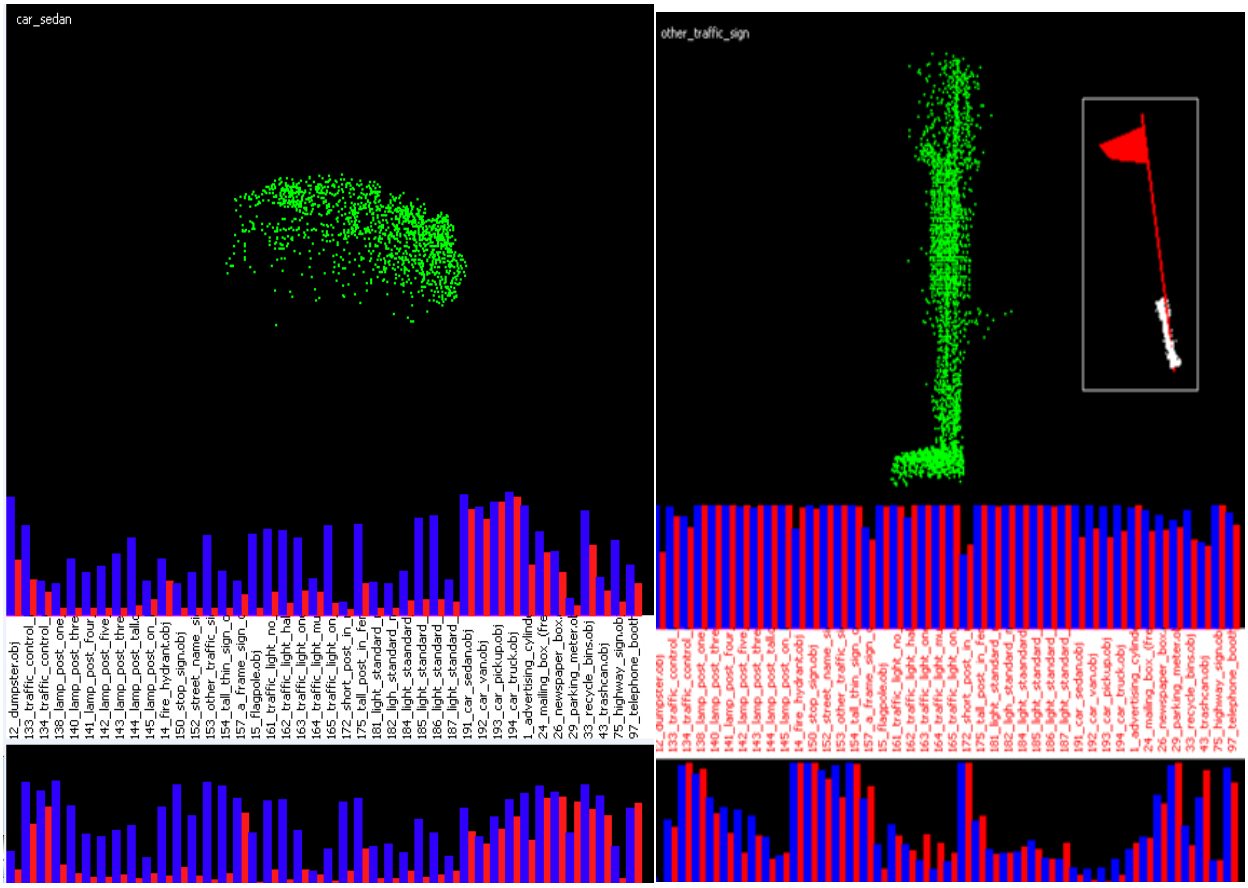


Figure 7: PointsFrac a versus MeshFrac in practice

For small objects, PointsFrac can give an erroneously high score to many mesh models that are very dissimilar to the point cloud. For example, when the traffic sign (green point cloud) on the right in figure 7 is aligned to models in the database, it has a score near 1 for both 3D and 6D for almost all of them (as can be seen by the bar graphs indicating scores per class on top). In the

gray box is shown the alignment of this model to a tall flagpole, as an example. It aligns very well with the base of the flagpole so most points are close to the mesh, so PointsFrac would give it a very high score. A similar thing happens for most of the other models. MeshFrac, on the other hand, is much more discriminating in this case.

MeshFrac fails when there is occlusion, however. On the left in figure 6 I picked a point cloud of a car that's only partially scanned. The scores graph on the top shows a higher peak in the distribution around all of the car models than the graph on the bottom. This example is a bit more subtle but can have an effect on the final classification results.

It is tempting to think that PointsFrac is the best choice because of the occlusion problem, but this parameter needs to be varied and comparisons need to be done to be sure.

D. Varying Epsilon

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.

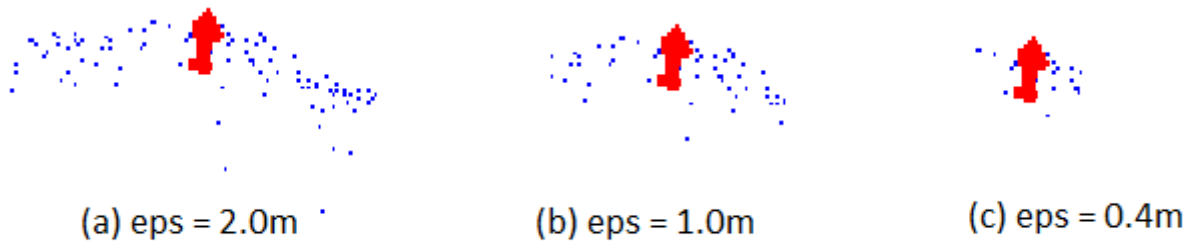


Figure 8: Results of varying epsilon aligning a car to a fire hydrant with 3DOF.

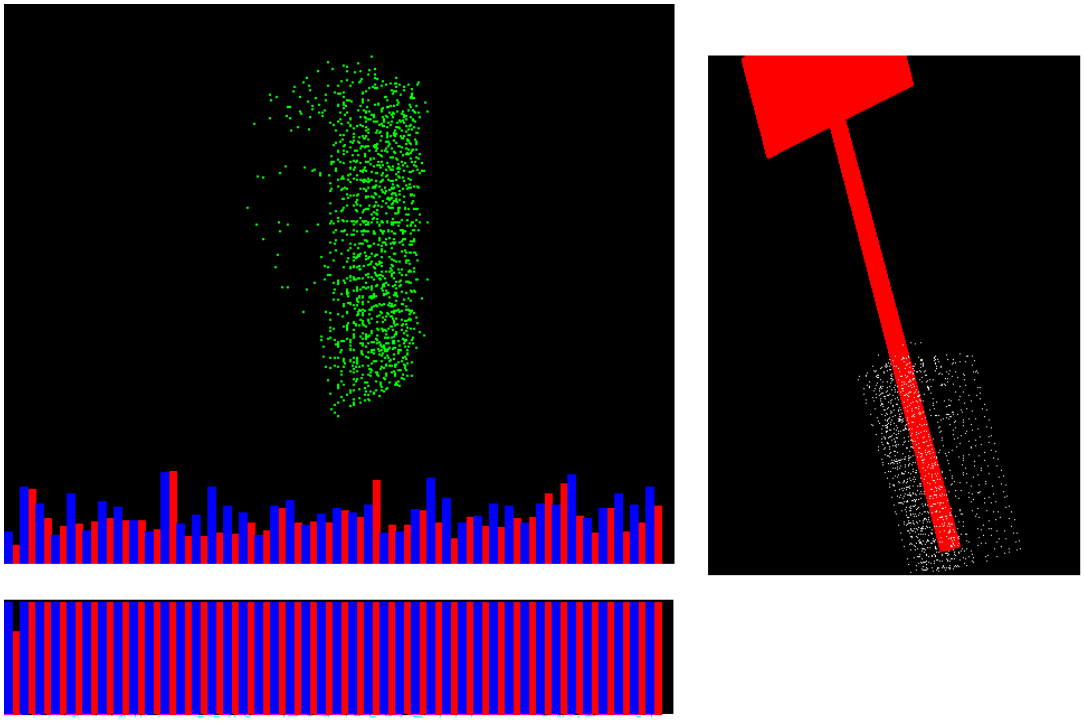


Figure 9: Aligning part of a newspaper box to all of the models in the database. The bar graphs on top show the result of aligning with an epsilon of 0.1, while the bar graphs on the bottom show the result of aligning with an epsilon of 0.35. Clearly, an epsilon of 0.1 is more discriminating since the ratios saturate at 1 for an epsilon of 0.35

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). At the extremes, 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.

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 point cloud scans in a given segmentation training set. The ICP alignment is the bottleneck of this whole process, so the results of the alignments are cached for future test, so that when the batch process goes back and recomputes all of the feature vectors with a new "epsilon" for cutoff, it doesn't have to recalculate all of the alignments as well. I do this by storing individual alignments in a text file every time they are done. But instead of last year where I spit out the entire aligned point cloud each time (which made for tens of thousands of files), I simply stored the affine matrices all in the same file indexed by some hash key that makes each alignment unique (e.g. aligning point cloud 12 to model "lightpost" will have the following entry in the file: lightpost_12 a00 a01 a02 a03 a10 a11 a12 a13 a20 a21 a22 a23 a30 a31 a32 a33). To differentiate 6D from 3D in that database, I append the string "3D" on the end for a unique entry that was done with 3D alignment. At the beginning of each batch test, I load this file and store all of the results in a hash table for quick lookup when I need them. In the even that I stopped a batch process prematurely, the program is able to use the hash table to figure out which alignments it has already done and which ones it has yet to do when I resume the process. 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 can take up to 3 hours when doing alignments from scratch, but then only take from 10-15 minutes when the alignments are cached ahead of time..

The batch testing was performed for both 6D and 3D on all of the 1163 objects in the manual training set, and for all 1087 objects in the mincut dataset. The cutoff value epsilon was varied from 0.05 meters to 0.75 meters, in increments of 0.05. The difference between PointsFrac and MeshFrac (as described in section IVd) was also explored. Choosing between 6D and 3D, manual and mincut segmentation, and PointsFrac and MeshFrac led me to do 8 series of epsilon varied batch tests. I then computed statistics on each of the batch tests to analyze their performance, which is described more below. This variation of parameters should rigorously encompass all of the scenarios I wanted to test this semester.

B. Creating Summarizing Statistics from Distance Matrices

Once the 42-dimensional feature vectors have been computed for every point cloud in the training set for a particular batch test (choosing epsilon, 6D/3D, segmentation technique, and PointsFrac/MeshFrac), a Euclidean distance matrix is calculated over all of the point examples. That is, a symmetric matrix is created, where the ij^{th} is the Euclidean distance in feature space between the i^{th} point cloud and the j^{th} point cloud. A leave-one-out test can then be carried out for every object by sorting all other objects in ascending order of Euclidean distance (most similar to least similar by whatever metric is in use). A bunch of statistics can then be computed based on how many models in the same class end up close to the beginning of the recalled models for a particular object.

One simple visual of the distance matrix looks like this:

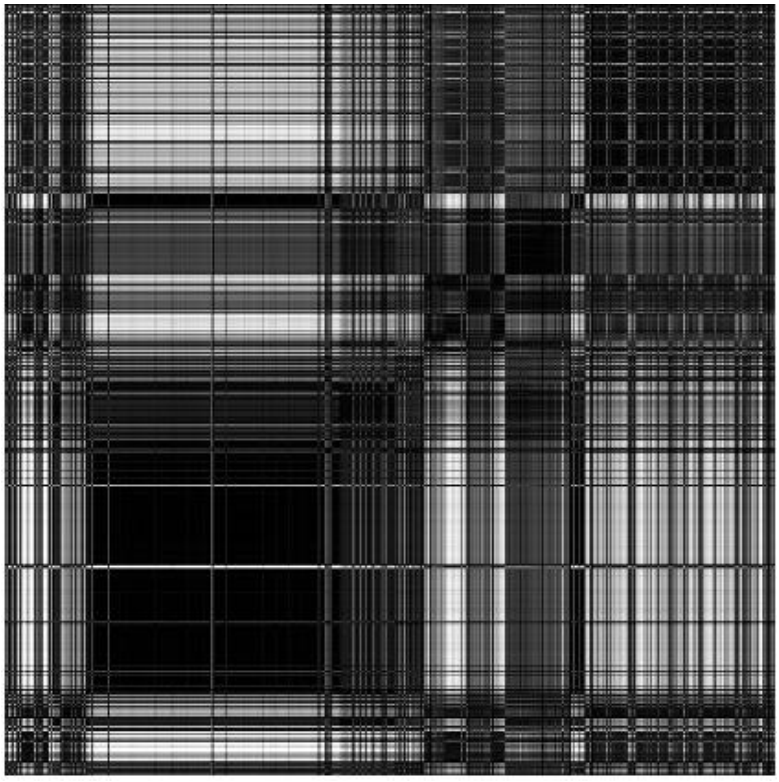


Figure 10

The black means very small distance, while the white means a very large distance. So we would hope to see the most black along the diagonal and within class lines.

Another even more informative visualization looks like this:

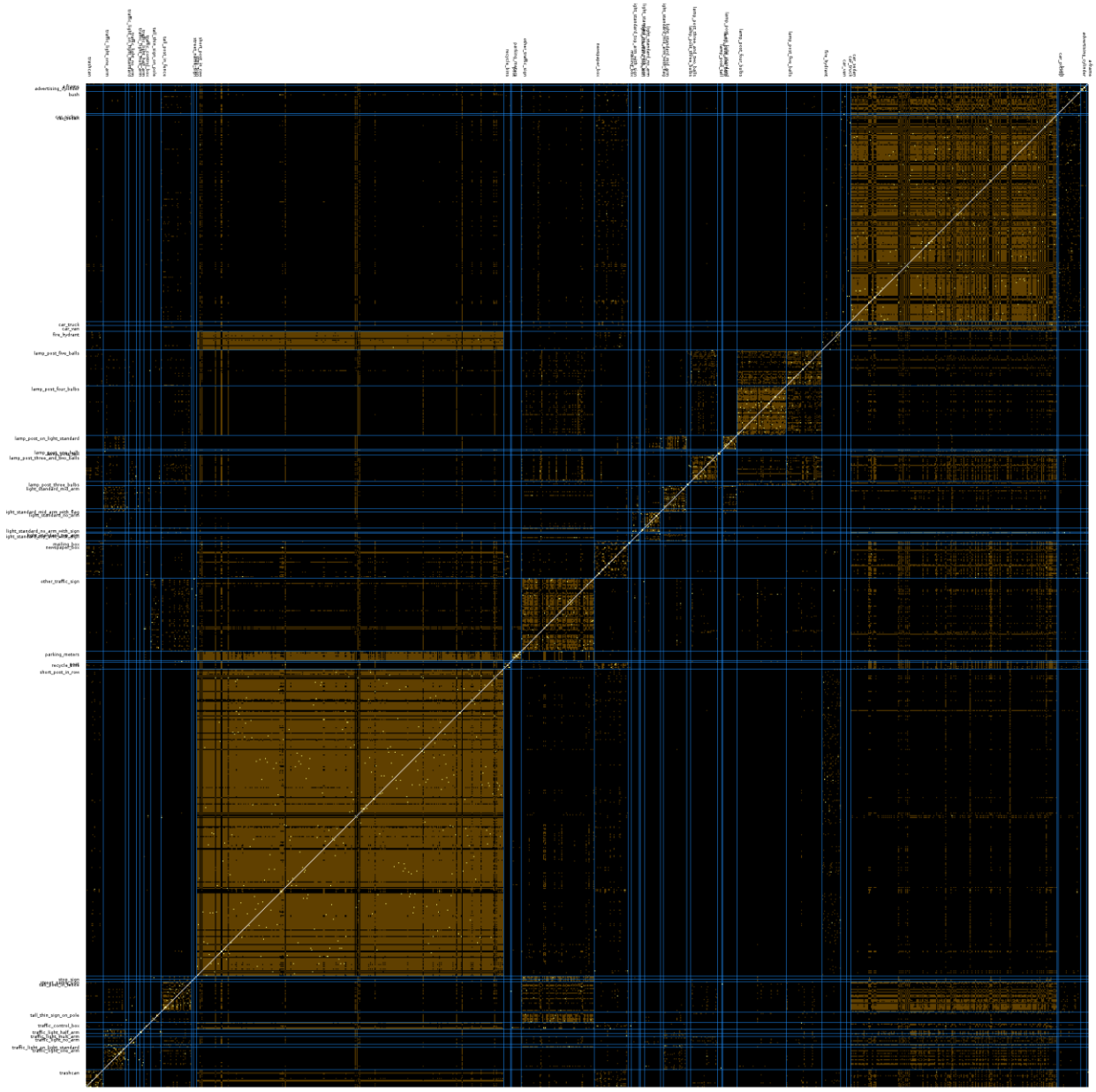


Figure 11: First Frontier Matrix Visualization

The brown marks how a point cloud got classified, and the white dots in a column signify the top 10-ranked point clouds by distance in the matrix. We would hope to see lots of brown within a square (meaning that point clouds within a class got classified within that class), and lots of white dots within a square (meaning that the top 10 most similar objects to an object are within the same class). Off-diagonal brown or white dots show class confusions. Patterns in these confusions can be used to find systematic flaws in the algorithm, so this plot is very useful.

The statistics that can be computed for each distance matrix are as follows:

- NN1 (nearest neighbor 1) is the fraction of first-ranked (closest in feature space aside from the object itself) point clouds that are of the same class as the query point cloud
- NN3 and NN5 (nearest neighbor 3 and 5) are the fraction of point clouds from the highest 3 and highest 5-ranked, respectively sorted entries that are of the same class as the query point cloud
- The "first tier" is defined as follows: If there are N point clouds in a particular class, the first tier is the fraction of the first N point clouds sorted by distance that are in that class.
- The second tier is the same as the first tier, except it's the fraction of the real point clouds that have been recalled in the first 2N sorted by distance
- The DC gain is another summarizing statistic that corrects for a lot of noise and is supposed to be the most reliable indicator of a method's accuracy

C. Generated HTML Files Showing Distance Rankings

A bunch of HMTL files are generated to show the ranking of different point clouds against each other based on distance. I generate a set of images to show the point clouds head on with bar graphs showing the feature vectors (process described in more detail in section VIII E). We would expect images closer to each other in feature space to have similar feature vectors.

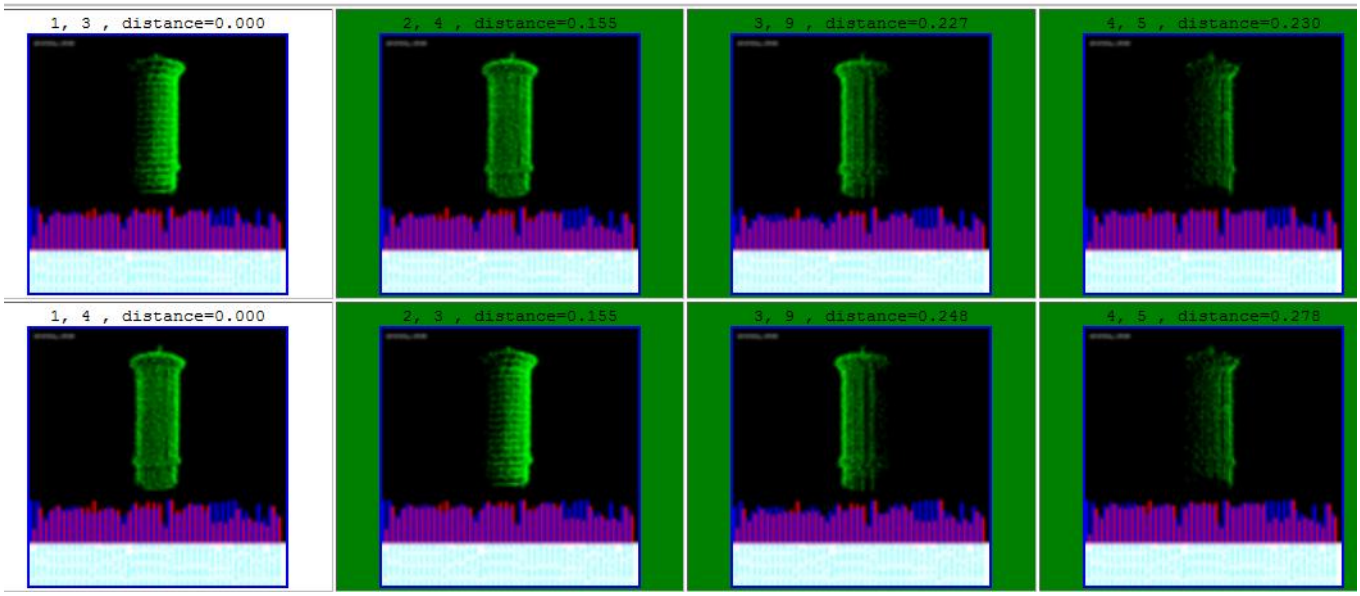
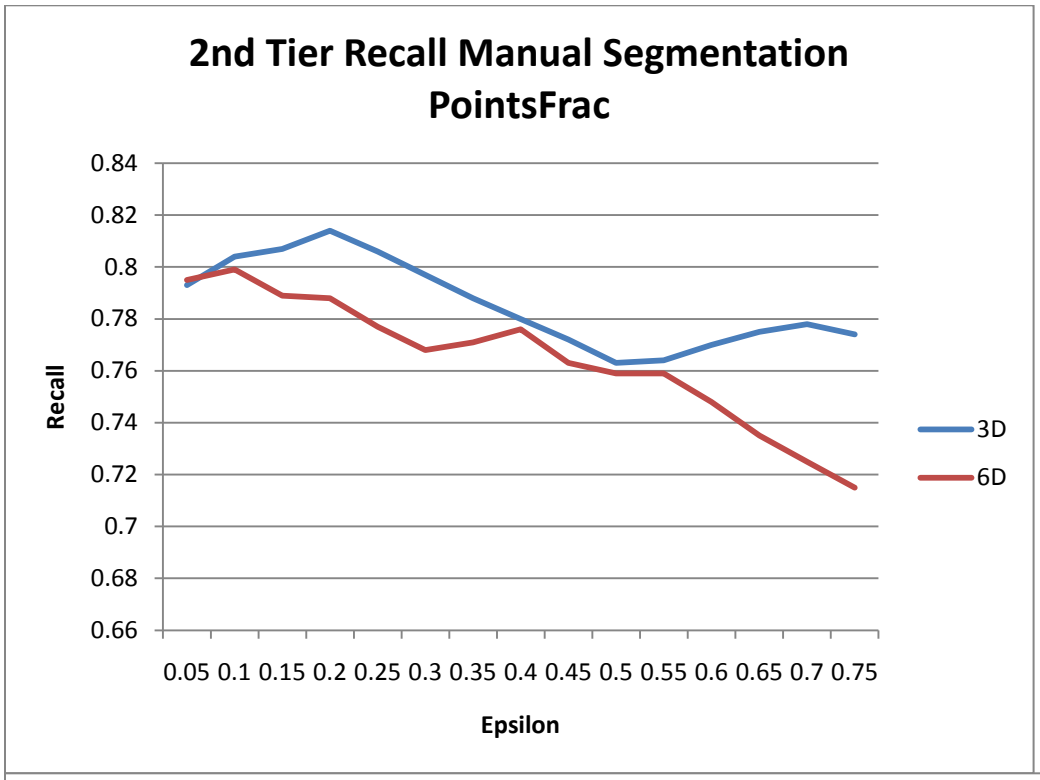
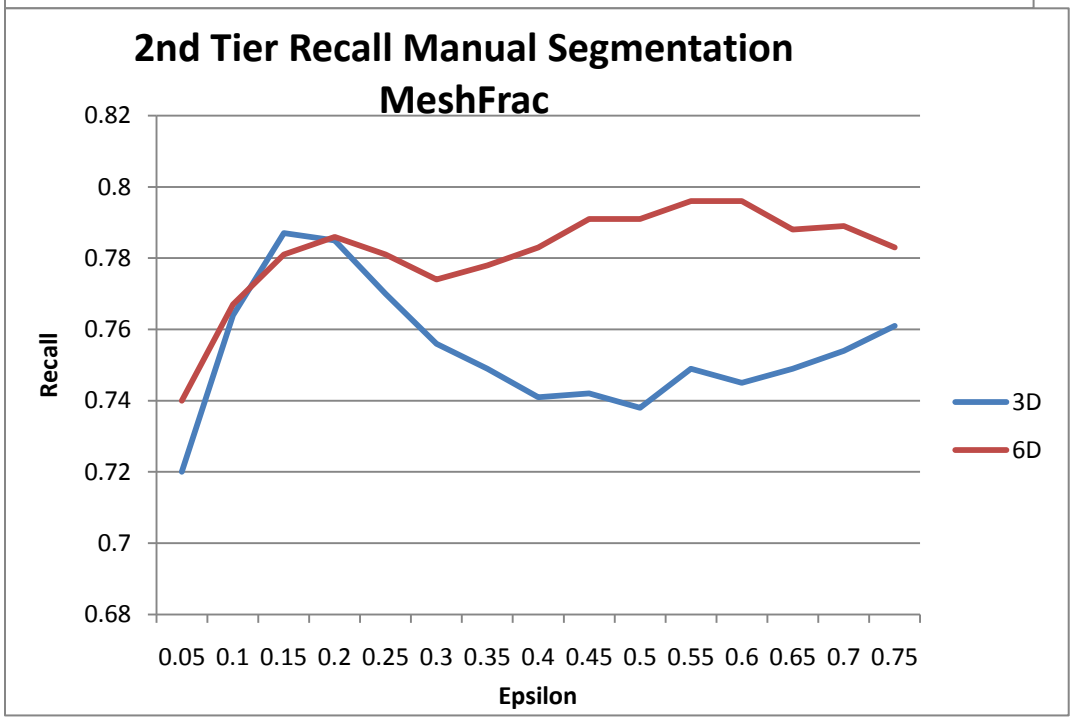


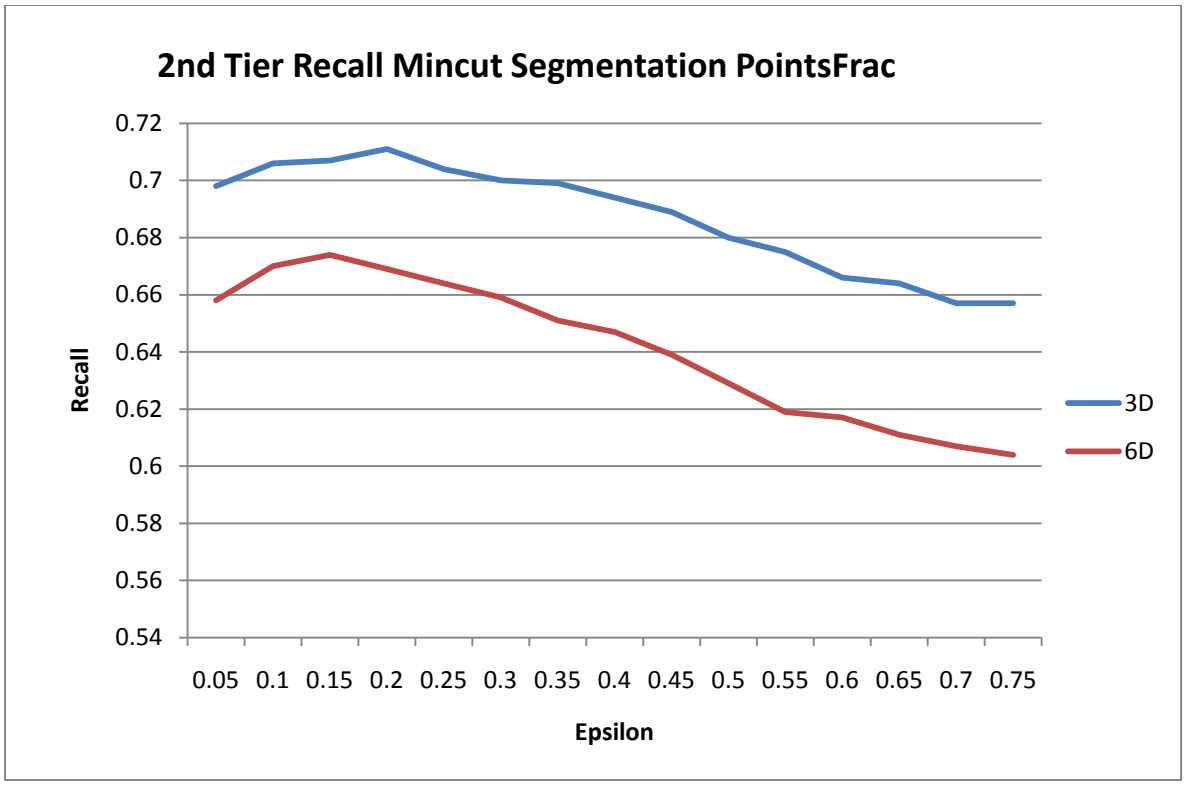
Figure 12: Returning similar objects ranked by distance. Note how similar the feature vectors are

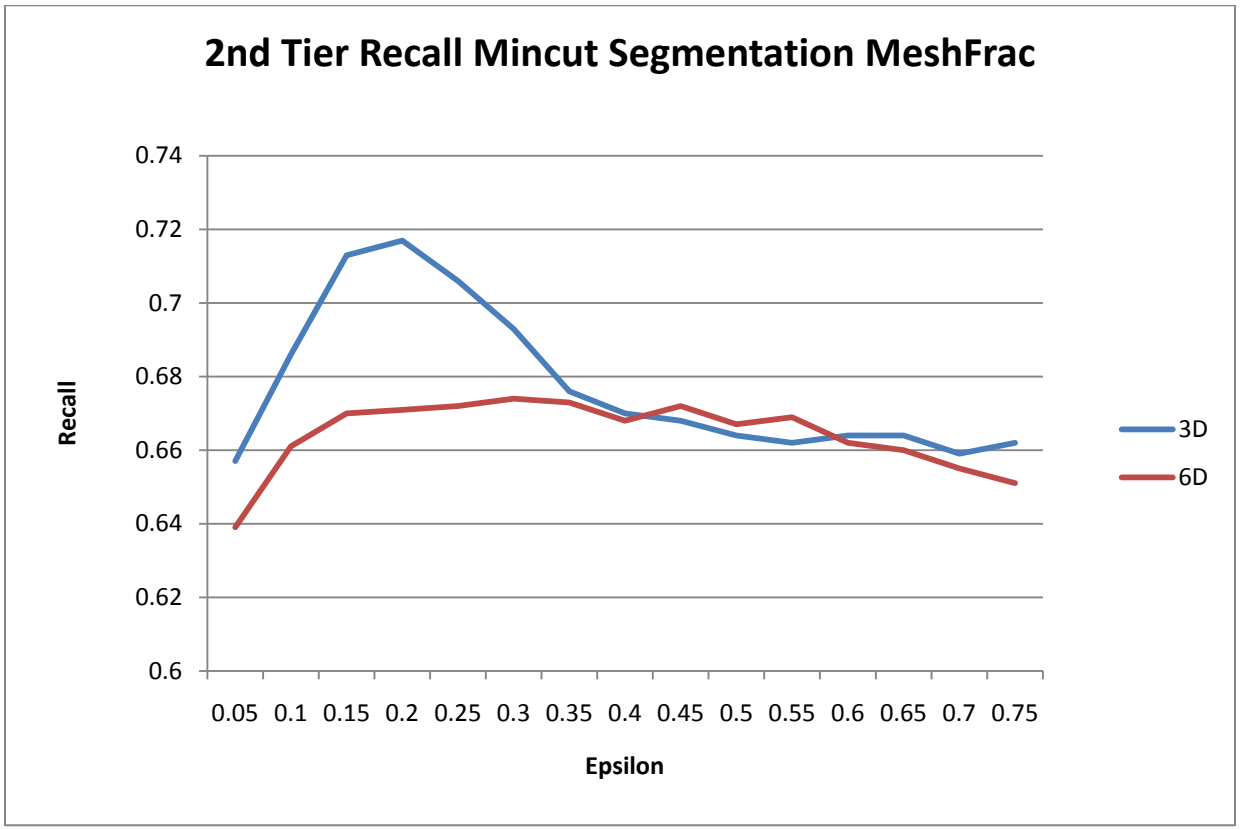
VI. Results / Analysis

A. Global Results of Batch Tests Varying Parameters






2nd Tier Recall Mincut Segmentation PointsFrac
Recall
0.72
0.7
0.68
0.66
0.64
0.62
0.6
0.58
0.56
0.54
0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5 0.55 0.6 0.65 0.7 0.75
Epsilon
3D
6D


2nd Tier Recall Mincut Segmentation MeshFrac
Recall
0.74
0.72
0.7
0.68
0.66
0.64
0.62
0.6
0.05 0.1 0.15 0.2 0.25 0.3 0.35 0.4 0.45 0.5 0.55 0.6 0.65 0.7 0.75
Epsilon
3D
6D

- In almost all variation of parameters, 3D outperforms 6D, which is encouraging since 3D was motivated by intuition. The only case where it doesn't is with manual segmentation with large epsilon for 6D

- The overall trend varying epsilon is to start off low and reach a global maximum, and then to decrease; as expected, a balance was struck between an epsilon that was too small and one that was too large. The distribution was skewed further towards a small epsilon than I expected, however. With PointsFrac especially, small (but not too small) epsilons lead to a much more discriminating classifier, which makes sense based on the examples shown before. Increasing epsilon has much less of an effect when MeshFrac is used, however, since models that are very different will still have regions beyond epsilon even for large epsilon (e.g. flagpole example). Overall, according to the batch tests, a choice of epsilon to be about 0.2 meters appears to strike the best balance between a cutoff that is too harsh and a cutoff that isn't discriminating enough.

- Not surprisingly, the classifier performs better on results that have been segmented by hand (manual segmentations), which means that this classifier is not resistant to segmentation noise

B. Global Comparison to SpinImages

I took the best epsilon for 3D and for 6D for each variation of parameters and combined the feature with spinimages, another feature of higher dimensionality that is used to classify city objects. Spinimages combined with this feature consistently does better than spinimaes alone, which suggests that these alignment descriptors are bringing new information into the learning.

Mincut Segmentation PointsFrac

Name	NN1	NN3	NN5	1st Tier	2nd Tier	Top Ten	E-Measure	DCGain
spinimagesAnd6D	0.776	0.785	0.789	0.581	0.712	0.694	0.248	0.819
spinimages	0.767	0.786	0.793	0.597	0.746	0.683	0.257	0.825
spinimagesAnd3D	0.757	0.769	0.779	0.603	0.731	0.683	0.245	0.818
3D	0.717	0.727	0.737	0.581	0.711	0.646	0.226	0.798
6D	0.701	0.715	0.719	0.542	0.674	0.622	0.215	0.782
random	0.157	0.165	0.177	0.155	0.310	0.155	0.040	0.559

Mincut Segmentation MeshFrac

Name	NN1	NN3	NN5	1st Tier	2nd Tier	Top Ten	E-Measure	DCGain
spinimagesAnd6D	0.802	0.804	0.802	0.594	0.718	0.695	0.260	0.827
spinimagesAnd3D	0.788	0.801	0.802	0.624	0.761	0.701	0.258	0.832
spinimages	0.767	0.786	0.793	0.597	0.746	0.683	0.257	0.825
3D	0.736	0.753	0.752	0.576	0.717	0.656	0.233	0.805
6D	0.717	0.732	0.735	0.526	0.660	0.632	0.220	0.787
random	0.142	0.142	0.154	0.157	0.311	0.160	0.042	0.561

Manual PointsFrac

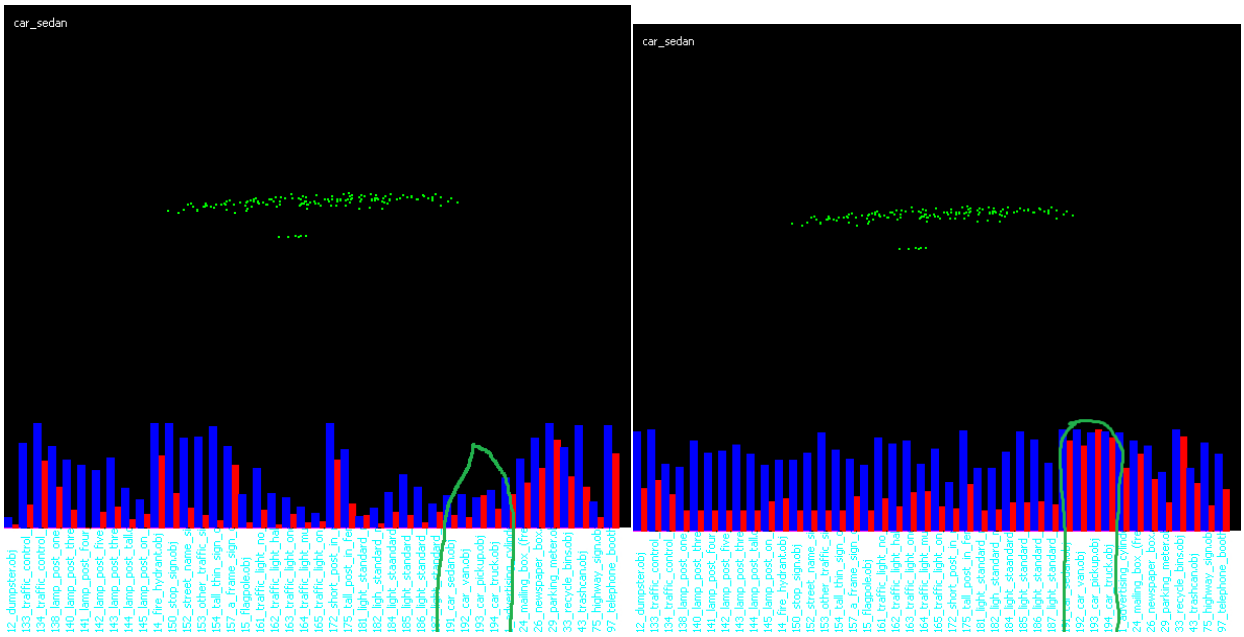
Name	NN1	NN3	NN5	1st Tier	2nd Tier	Top Ten	E-Measure	DCGain
spinimagesAnd6D	0.834	0.848	0.850	0.704	0.824	0.767	0.295	0.875
spinimagesAnd3D	0.814	0.820	0.822	0.692	0.826	0.733	0.287	0.867
6D	0.802	0.812	0.809	0.622	0.767	0.708	0.255	0.833
3D	0.782	0.815	0.817	0.636	0.785	0.710	0.271	0.844
spinimages	0.753	0.778	0.779	0.630	0.771	0.693	0.262	0.834
random	0.143	0.150	0.157	0.146	0.293	0.145	0.037	0.551

Manual MeshFrac

Name	NN1	NN3	NN5	1st Tier	2nd Tier	Top Ten	E-Measure	DCGain
spinimagesAnd6D	0.818	0.838	0.833	0.685	0.822	0.737	0.286	0.865
spinimagesAnd3D	0.801	0.831	0.826	0.679	0.824	0.730	0.291	0.864
3D	0.797	0.814	0.816	0.638	0.787	0.711	0.271	0.844
6D	0.793	0.806	0.815	0.643	0.795	0.720	0.272	0.847
spinimages	0.753	0.778	0.779	0.630	0.771	0.693	0.262	0.834
random	0.151	0.162	0.173	0.147	0.293	0.149	0.037	0.551

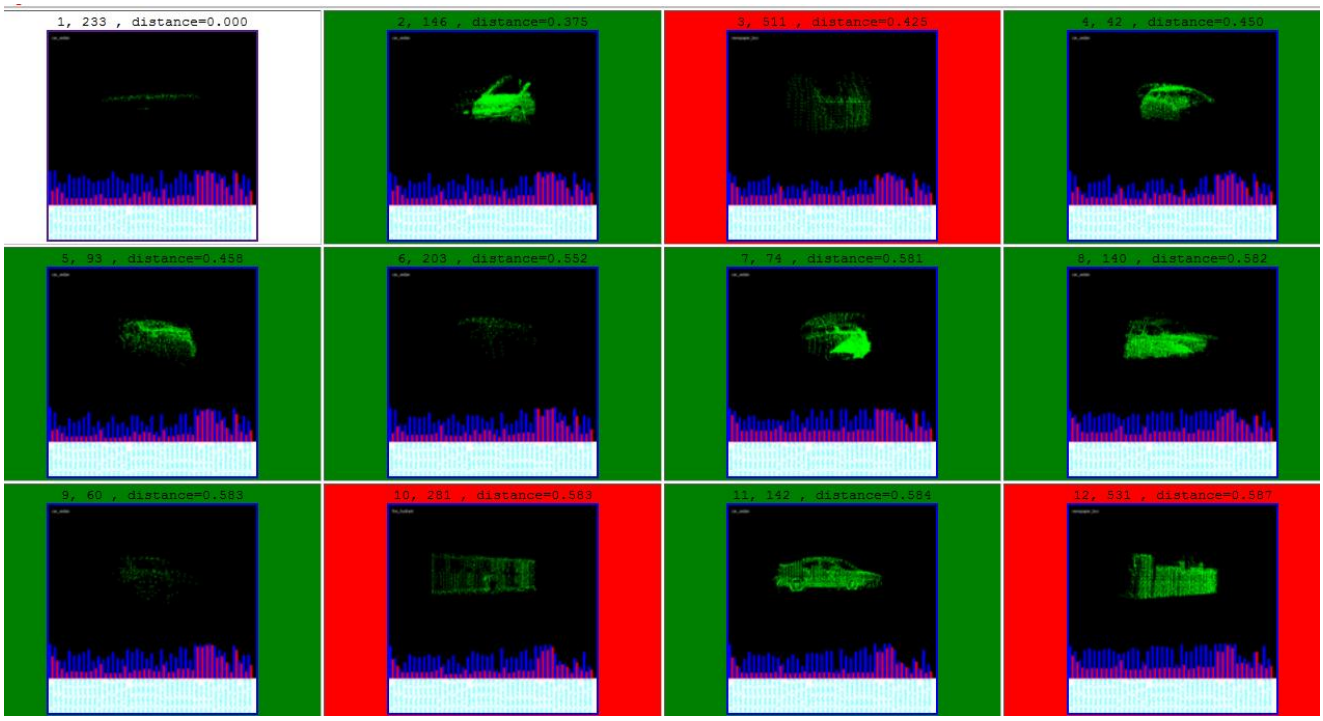
C. Using the Tools to Look More Closely at Some Anomalies

i. Severe Occlusion car_sedan example

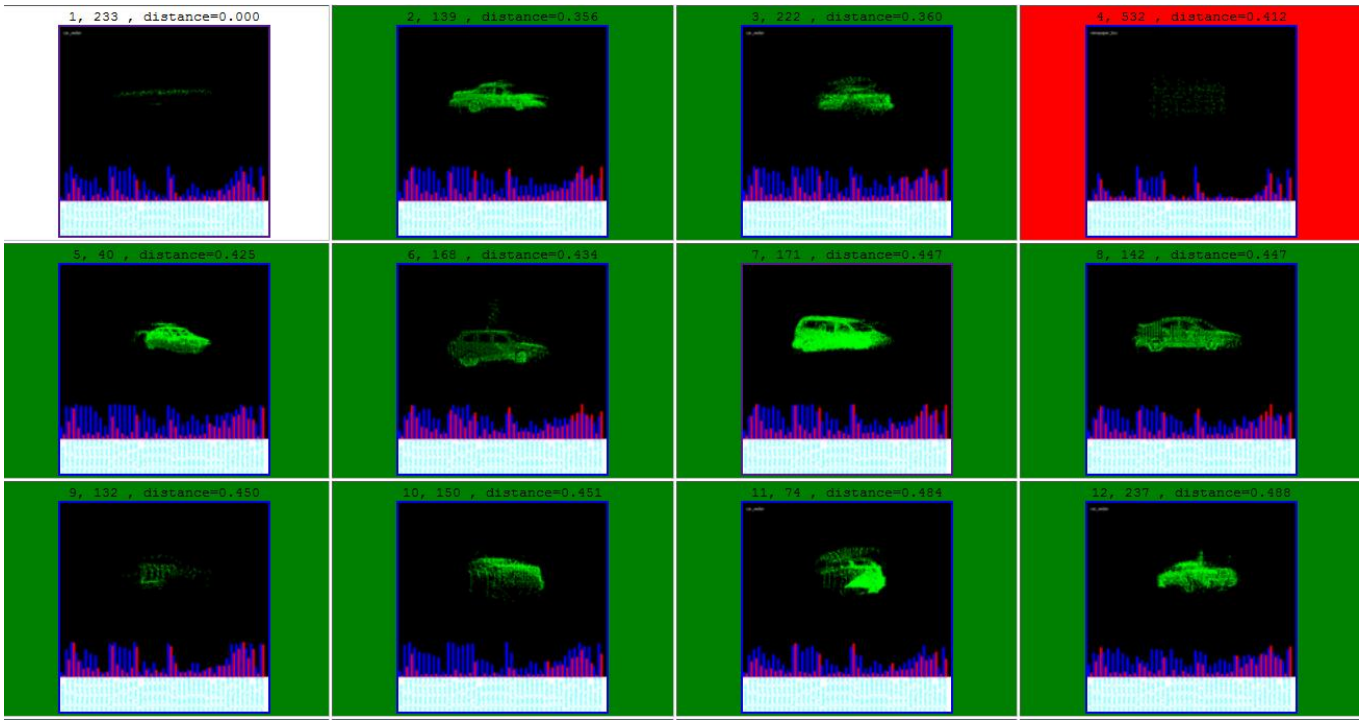


To the left is shown the feature vector of the car using MeshFrac, and to the right is shown the feature vector of the car using PointsFrac. I circled the car models in green.

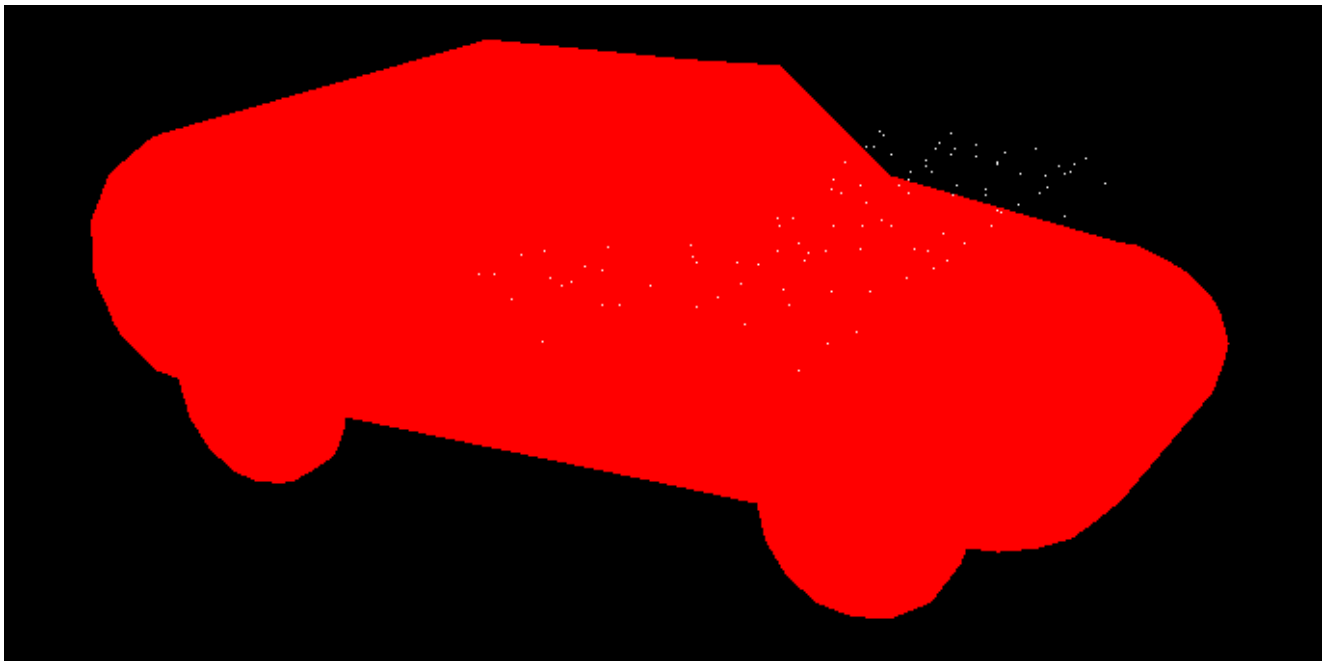
First 12 objects recalled for PointsFrac



First 12 objects recalled for MeshFrac



Below is a picture of the 3D alignment



It is clear that the features match better for the PointsFrac case, but surprisingly the first 12 nearest neighbors have more incorrect examples in PointsFrac than MeshFrac. However, it is possible that even though MeshFrac doesn't score very high for the car features, it scored high

for other features that give important geometric cues that can be learned on like the height and width of the car even in the presence of occlusion. Looking through lots of results, I noticed that most of the meshfrac images have several relatively high peaks in constrast to a more uniform distribution for pointsFrac, and I believe these peaks give geometric cues that aren't necessarily related to the original object but that are available because the database of 42 models is so rich (it has short fat, tall/skinny, and everything in between). In this example, the MeshFrac peaks occur around fire_hydrant, traffic_control box, short_post_in_row, and parking_meter. These objects are all roughly the same height as the part of the car that's there in the occluded model, and MeshFrac also gives them a high score for the models that are recalled next that contain fuller scans of the cars. Hence, MeshFrac is still able to learn an occluded model, even though PointsFrac was designed to do so.

ii. Confusions in tier images

Last year I did a few hand-wavy examples of confusions, but now it's possible to make very definitive claims about confusions by looking at the tier images to determine where the first tier of a class ends up. A lot of the first tier of small_posts_in_row falls into the fire hydrant category, which makes sense since they have a similar geometry. A car_pickup also gets confused with a bush often, which is imaginable looking at figure 5 (the bush is roughly the same height as the car_sedan in that example). The lampposts with different numbers of balls are all confused with each other, and there is some confusion between newspaper boxes, mailing boxes, and traffic signs that are all about the same height.

VII. Conclusions / Further Avenues

The use of 3D and 6D alignment features shows promise for learning object classes in for small point cloud objects in urban environments. This descriptor performs better than spinimages most of the time and is less than half the dimensionality of spinimages, which suggests it is doing a better job at compressing the relevant shape information. Also, it will be easier to model for learning techniques that are susceptible to the curse of dimensionality. However, Spinimages does beat both 3D and 6D when automatic segmentation is done, which means that spinimages is better at resisting segmentation noise. With improved segmentation

techniques the alignment feature would be expected to dominate. Otherwise, an epsilon cutoff of about 0.2 meters with 3DOF alignment appears to be the best way to start using these techniques.

More analysis still needs to be done as to why there isn't a significant difference between PointsFrac and MeshFrac, however, which I was very surprised to find because of the occlusion problem. My initial guess would be that in the cases that I expected MeshFrac to fail, some of the other models were able to pick up on geometric cues that can still work with occlusion (as I explained in my example in the analysis section), but I would have to spend hours looking at many more examples to verify that this is true. This is difficult when I discover that some of the alignments are off and the batch tests need to be redone. It may be easier to create scripts to automatically detect and view anomaly cases like this.

One arguable limitation of this approach is that the mesh database is static and needs to be modeled beforehand. Perhaps an algorithm should be designed to automatically learn what regions of space are discriminating per class and to automatically come up with N volumetric models to which to align, where N is the number of classes in the training set. I have already begun to explore such a method. I took all of the point clouds from the manually segmented data (a nice, clean dataset good to learn classes on) and outputted them aligned to the origin. I would then imagine creating a voxel grid per class that somehow voted on which voxels were the most common within that class. A weighted alignment to each of the learned models could then take place. In any case, alignment-based features seem like a good idea taking the initial look that I did, so the next step would be to automatically learn them.

VIII. Software Contributions / Usage

A significant amount of my time this semester was devoted to rewriting all of the alignment code from scratch to fit within the Master City framework so future contributors to the city project can use this feature. I also created a few visualization tools to break down the large data sets so I could analyze my results more efficiently. I will now briefly summarize my software contributions to the Master City Project in these areas

A. The LPAlignmentFeature Class

The LPFeature class is the parent class to all possible feature vectors that can be computed for a point cloud. The LPFeature interface has a function called "Evaluate" that accepts an LPLidarPointSet object and a coordinate system, and which is responsible for computing the feature vector. It then stores the result of the feature vector in an array of floats. For my research, I created a child class of LPFeature called "LPAlignmentFeature." This class has static methods for loading in all of the mesh models and for loading the cached alignments from previous batch tests. It has a subclass called "AlignmentModel" that is used to store the information for each model to which the points are to be aligned. When an AlignmentModel object is initialized, it is passed a filename. It then loads a mesh from that file, computes and stores all permutations and flips of the PCA axes for both 6D and 3D (so this doesn't have to be recomputed for every new alignment to this model), and samples points from the model and puts them into a KD tree for fast lookup later during correspondence finding (as described in section IV A). An array of AlignmentModel classes is initialized at the beginning of a batch test to save time, and the Evaluate function accepts a new point cloud that it will align to these models each time a new feature vector needs to be computed.

B. Modifications to evaluateObjects

The "evaluateObjects" app in the MasterCity project is used to evaluate batch tests on a set of point clouds obtained from a segmentation phase. I had to make a lot of modifications to this file to vary the parameters that I wanted to vary during my alignment batch tests. Here's a rough list of the modifications I did:

- I added a flag to append an LPAlignmentFeature to the list of features that get computed for each point cloud. The syntax for the feature is as follows:
  `-alignmentFeature <epsilon> <cached alignments text file> <models directory> <alignment feature 6D: 1/0> <alignment feature 3D: 1/0> <features directory name>`
  Adding this flag will append a 42-dimensional 6D alignment feature if <alignment feature 6D> is set to 1, and/or a 42-dimeansion3D alignment feature if <alignment feature 3D> is set to 1. It will save all of the feature vectors for all of the point clouds in the directory <features directory name> for different values of epsilon, which is then used to draw bar graphs later

- I added a function to sort all of the point clouds in alphabetical order by ground-truth class name before running the batch process. I did this because I found that the in the segmentation results, the point clouds were not always continuous by class; sometimes the examples would switch from class A to class B and then back to class A again. I needed the classes to be completely continuous so I could create the .cla files used to analyze the distance matrices.

  NOTE: Sorting is disabled by default to make this code backwards compatible. To enable it, include the flag -sortObjects
- I added a function to compute Euclidean distance matrices between all feature vectors. Include the flag –distanceMatrix <output file> to output the distance matrix to a binary file that's compatible with the distance matrix classification tools.\

These are a lot of parameters to deal with, so I created a Perl file called "batchTests.pl" to vary the parameters and make calls to evaluateObjects for me.

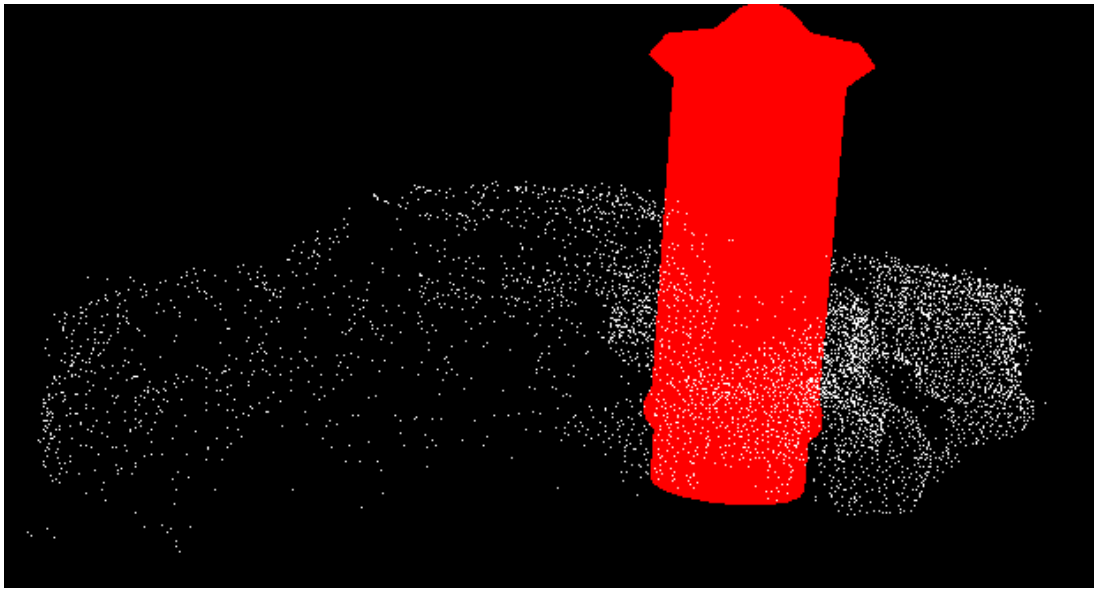
C. simpleAlignmentResultsViewer

I created an app to load a model and a point cloud and display them on top of each other with the alignment between them that's been cached during the batch tests. This way I can view how accurate an alignment was for 3D or 6D. The parameters for this program are as follows:

```
alignmentResultsViewer <model> <pointclouds dir> <point cloud
num> <alignments.txt> <3DOF: 1/0>
```

It is assumed that the point clouds have been outputted by number to some directory (point cloud directory) after the batch tests. Alignments.txt is the test file that should hold all of the alignments. If 3DOF is set to 1, then view the 3D alignment result. Otherwise, view the 6D result. Here is an example execution of this program (and a picture to go along with it; NOTE that the user may have to double click on the center mouse button to get the alignment in view):

```
alignmentResultsViewer models\1_advertising_cylinder.obj
pointClouds 40 alignments.txt 1
```



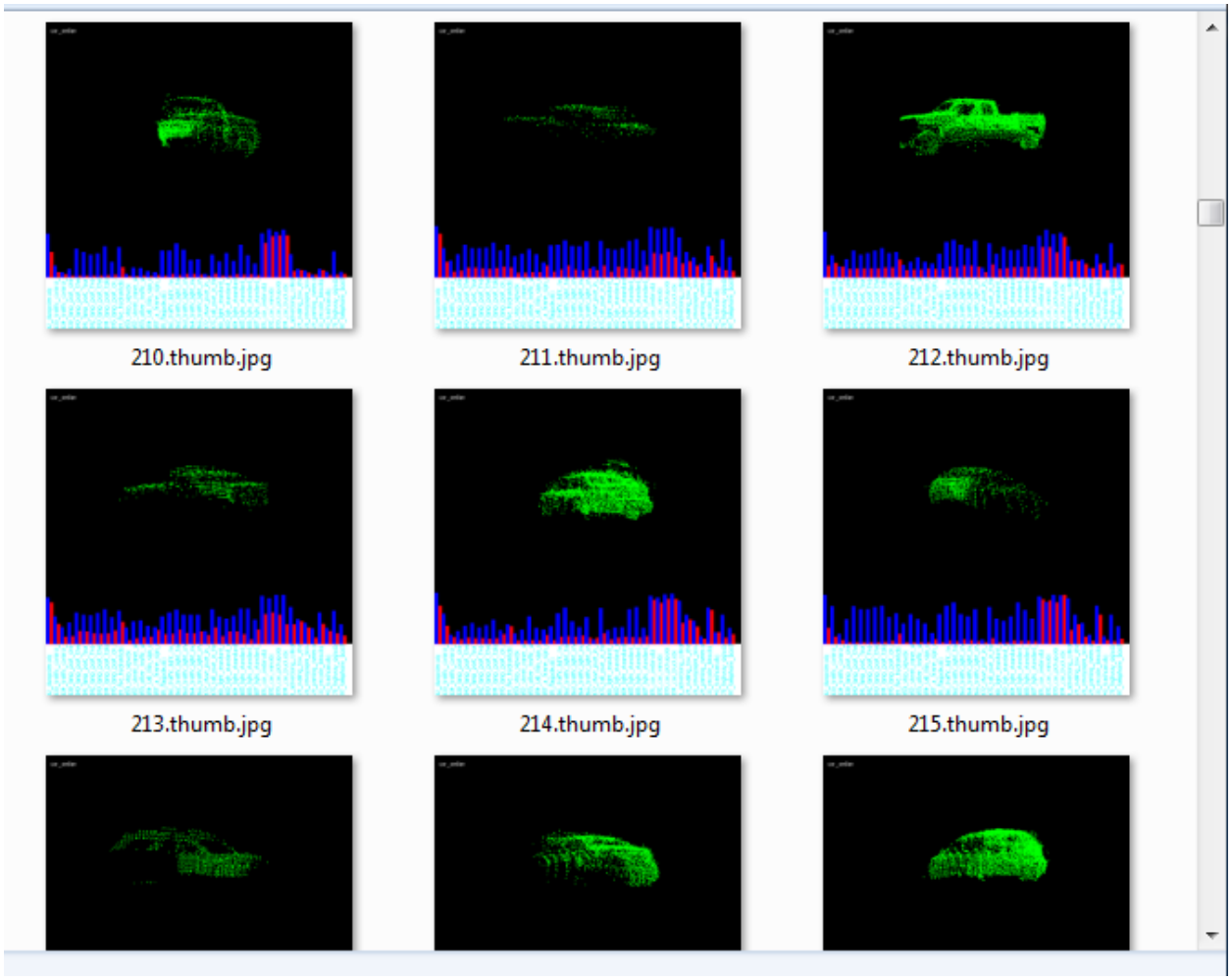
D. Modification to simplePointsViewer

The distance matrix tools create web pages that show models recalled, and they try to find a thumbnail for each model in the distance matrix to display along with the recalled objects. I wanted to get a snapshot of each point cloud so that I could view images on these web pages and see what similar point clouds (by the alignment metric) would look like. To do this, I added some code to the simplePointsViewer app to save snapshots of the point clouds to an image file. The program finds the center of mass and principle components of the point cloud, and then it creates a snapshot by looking head on at the point cloud based on this PCA information.

E. The drawAlignmentBarGraphs App

Once I have completed all of the batch tests and generated images for all of the point clouds using simplePointsViewer, I would like to have a nice way to view the feature vectors that were computed. I decided to do this by drawing bar graphs on top of the image of each point cloud. An epsilon is chosen and the feature vectors are loaded from a file (recall from second B how a feature vectors directory was specified during the batch test), and the features for 3D and 6D are drawn for each of the 42 classes as bar graphs using the CImg library. 6D features are drawn in blue, while 3D features are drawn in red right next to the 6D ones. Here's what a folder full of

this type of data looks like after executing a batch file (makeImages.pl) to run this on all of the point clouds:



F. runtests

I worked with and modified Professor Funkhouser's "runtests" program to do classification results on distance matrices. I created a little C++ script to set up .cla files based on a file that has all of the ground truth class names in alphabetical order (as discussed before). I can then put all of the distance matrices in one folder and have runtests crank away on it to create .HTML files to view the results.

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] D. Munoz, N. Vandapel, and M. Hebert, "Directional associative markov network for 3-d point cloud classification," in Fourth International Symposium on 3D Data Processing, Visualization and Transmission, 2008.

[6] Tralie, Chris. "Classification of 3D Point Cloud Scans in Urban Environments." Junior Independent work 2010. <http://www.princeton.edu/~ctralie/Projects/IWS2010/ctralie_finalIWreport.pdf>.

[7] P.J. Besl, N.D. McKay, "A Method for Registration of 3-D Shapes," IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 14, no. 2, pp. 239-256, Feb. 1992, doi:10.1109/34.121791

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