Passive Object Recognition using Intrinsic Shape Signatures

by

Kenneth M. Donahue

B.S., Massachusetts Institute of Technology (2009)

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

May 2011

Author . . . .
Department of Electrical Engineering and Computer Science
May 20, 2011

Certified by . . . .
Seth Teller
Professor
Thesis Supervisor

Accepted by . . . .
Dr. Christopher J. Terman
Chairman, Masters of Engineering Thesis Committee

Passive Object Recognition using Intrinsic Shape Signatures

by

Kenneth M. Donahue

Submitted to the Department of Electrical Engineering and Computer Science
on May 20, 2011, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

The Agile Robotics Group (AR) at MIT's Computer Science and Artificial Intelligence Laboratories (CSAIL) has an autonomous, forklift capable of doing task planning, navigation, obstacle detection and avoidance, focused object detection, etc. The goal of the project is to have a completely autonomous robot that is safe to use in a human environment.

One aspect of the project which would be very beneficial to moving on to more complicated tasks is passive object recognition. The forklift is capable of doing a focused scan and looking for very particular things. The forklift is constantly scanning its vicinity with its Light Detection and Ranging (LiDAR) sensors to ensure that it avoids obstacles; instead of only using that information for hazard avoidance, that information can be used to not only passively notice objects but also classify them. This will be useful later when the team starts implementing various higher-level processes, such as localization and/or mapping.

This paper will discuss various modules that were integrated into the Agile Robotics infrastructure that made object recognition possible. These modules were 1) a data segmentation module, 2) an object recognition module using Intrinsic Shape Signature[10] (ISS) to find feature points in our LiDAR data, and 3) various visualization modules to ensure that each module was behaving properly.

Thesis Supervisor: Seth Teller
Title: Professor

Acknowledgments

I would like to express my sincere gratitude to Prof. Seth Teller for guiding me through this process.

I would also like to thank my parents, Craig and Kathleen Donahue, and family for their unconditional love and support.

Thanks also to Sherry Gao for putting up with me.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In 2008, CSAIL's Agile Robotics Group began work on an autonomous military fork-lift with the desire to create a robot that operates in human-occupied, uneven outdoor environments.[9]

Many of the people on this project previously worked on the MIT team for the Defense Advanced Research Projects Agency's (DARPA) Urban Challenge (DUC). This project was to create an autonomous vehicle that could operate in an uncertain environment while still obeying traffic rules.

The forklift project was able to reuse some code from the DUC codebase because the projects had a fair amount of overlap. The existing libraries made implementing a passive object recognition system possible.

The rest of this paper is structured as follows. First, we more fully describe the problem of real-time object recognition from LiDAR data and how this problem has been approached in the past. We then discuss our proposed passive object recognition system at a high-level. We go on to step through the details of the recognition algorithm. Afterwards, we show discuss our results. Finally, we describe how the system can be extended in the future.

1.1 Related Work

LiDAR-based object recognition is a highly researched topic. The research has been focused on detection of hazard avoidance for autonomous systems[4][5], hazard detection for passenger vehicle safety, and with detection of buildings for arial surveillance. Other more complicated recognition using LiDAR require slow, focused scans of regions of interest to create very dense information about the object. For example, pallet recognition in Teller, et al. requires a gesture from the user on a tablet. The gesture provides a Volumne of Interest, which the forklift can then slowly scan, looking for features that indicate a pallet.[9]

1.2 Motivation

Object recognition is a hard problem. Many systems are able to perform object recognition by first being given a region of interest (ROI). Once the ROI is given, a focused scan of the region can be performed. Because the search is limited to that region, recognition can usually be done in a reasonable amount of time. In this case, we do not know a priori the ROI, so we need to store data over the whole space. More data with less a priori information means more search. Creating a system that is able to do this matching in real time is quite a challenge.

Additional problems such as self-occlusions and missing data make this recognition even more challenging.[6] The Intrinsic Shape Signature approach is robust to occlusions;[10] we chose to use it to mitigate this problem.

1.3 Approach

The object recognition system will be composed of three main parts:

1. a data segmentation module,
2. a matching module, and
3. many renderers displaying graphical information about the other modules.

1.3.1 Data Segmentation

The point of the data segmentation module is to focus our processing on a volume of interest instead of the whole space. This module will filter the LiDAR data such that instead of looking at all of the hits, we are only looking at points which are within our volume.

1.3.2 Matching

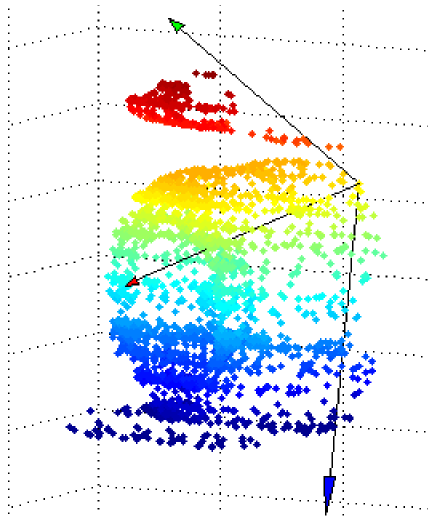


Figure 1-1: Intrinsic Shape Signatures: This algorithm relies on finding the eigen basis of each point and the finding the corresponding feature vectors for each interesting point. This is the eigen basis of a feature point on a truck tire. The LiDAR data has been false-colored by height.

The matching module will do the actual object recognition. This module is largely based on Zhong's paper on Intrinsic Shape Signatures ([10]). In it, Zhong describes a system for object recognition using a robust *signature* construct which is based on the geometry of points in a neighborhood. The Intrinsic Shape Signature has two main parts: an orthonormal right-handed basis (a rotated xyz-coordinate frame, as depicted in Figure 1-1) and a feature vector used to describe the points in the neighborhood around that point in that orthonormal basis.

The main element that was changed from Zhong's paper was the saliency calculation. This will be discussed in greater detail in Chapter 4.

1.3.3 Renderers

In order to visually verify that the modules were working as well as to debug the modules at various stages through their respective algorithms, many renderers were created to work inside of the existing Agile Robotics *viewer*. These renderers rely on Lightweight Communications and Marshalling (LCM) to receive data and use OpenGL commands to render the data in the viewer's 3D rendering environment. Each renderer will be discussed in Chapter 7.

Chapter 2

Code Framework

In this section, we will discuss the existing infrastructure that was used to create this project. Agile Robotics provided quite a bit of the codebase that made this project possible, including the message passing system, the rendering environment, and many useful libraries.

2.1 Lightweight Communications and Marshalling (LCM)

The forklift, like its DARPA vehicle predecessors, uses Lightweight Communications and Marshalling (LCM) as its message passing system to pass messages between modules. LCM follows the publish/subscribe message passing archetype and has bindings in Java, C, and Python. This system affords many benefits, the least of which is modularity. Processes become independent of one another, meaning that they can run on one machine or a network and faults do not propagate from one process to another. The modules can also be written in a number of languages, meaning that the user can choose to code in whichever language he or she is most comfortable.[3] We will be using LCM as our message passing system so we can not only communicate with existing processes but also communicate between our own processes and visualize the recognition algorithm at various steps.

LCM also comes with `lcm-logger` and `lcm-logplayer`, which allow for LCM message logging and LCM log playing respectively. The `logger` was used to store LiDAR, camera, odometry, and other messages on various runs on the forklift, all targeting a rental truck. The `logplayer` was used to play back those logs on other computers not connected with the forklift so that the recognition code could be tested without having to run the forklift every time.

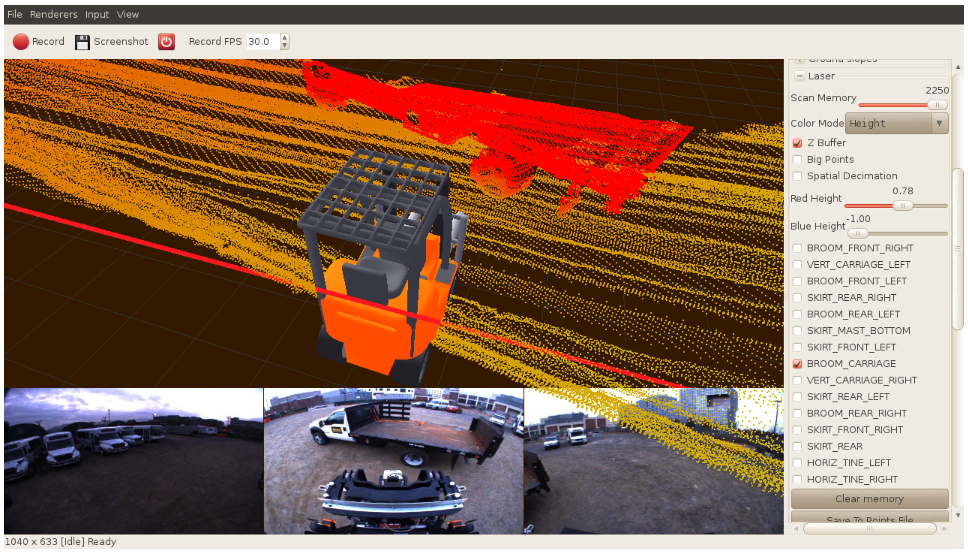
2.2 Viewer

The Agile Robotics code comes equipped with a home-made OpenGL-based rendering environment for 3D displaying of information: the *viewer*. It easily allows for new Renderers to be added to visualize whatever information is pertinent. Below is a screen-shot of the visualizing environment. The viewer, like LCM, is modular in that each renderer can be run independent of the other renderers. The viewer allows the user to choose which renderers are on, set renderer-specific parameters, and interact with each renderer. Each renderer can have mouse and keyboard event listeners allowing the user to change its internal state. As stated previously, we plan create a rather elaborate renderer to visualize the algorithm at various steps.

2.3 Other Agile code

The Agile Robotics repository also has some home-grown libraries for fast geometry computation including standard math operations as well as rotation conversions. These libraries proved invaluable for computing pose estimations as well as quickly finding the proper transform to perform for rendering purposes.

The (unit) quaternion transformations were invaluable as they allowed us to represent the rotation in a unique manner, meaning that each rotation had exactly one representation. That fact allowed us to more accurately estimate the rotation between a LiDAR cloud and a model, as will be discussed later.



Scan Memory 2250
Color Mode Height
[✓] Z Buffer
[ ] Big Points
[ ] Spatial Decimation
Red Height 0.78
Blue Height -1.00
[ ] BROOM_FRONT_RIGHT
[ ] VERT_CARRIAGE_LEFT
[ ] BROOM_FRONT_LEFT
[ ] SKIRT_REAR_RIGHT
[ ] BROOM_REAR_LEFT
[ ] SKIRT_MAST_BOTTOM
[ ] SKIRT_FRONT_LEFT
[✓] BROOM_CARRIAGE
[ ] VERT_CARRIAGE_RIGHT
[ ] SKIRT_REAR_LEFT
[ ] BROOM_REAR_RIGHT
[ ] SKIRT_FRONT_RIGHT
[ ] SKIRT_REAR
[ ] HORIZ_TINE_LEFT
[ ] HORIZ_TINE_RIGHT

Figure 2-1: Agile Robotics Viewer: Forklift approaching a truck. LiDAR data is false-colored by height; the red-orange LiDAR hits correspond to the truck. At the bottom are displayed the Left, Front, and Right camera images to better visually convey what the forklift sees.

2.4 Tina5 Linear Algebra Package

In order to handle matrix multiplication and eigen-basis decomposition, we opted to use an existing linear algebra package called TINA5[4]. This package was chosen mainly because of its implementation of eigen-decomposition; this functionality was required as eigen-bases are used to represent the signatures.

2.5 KDTree Implementation

Much of the recognition algorithm relies on storing the LiDAR data in such a way that the neighbors of each point are quickly accessible. John Tsiombikas' implementation of a *k*d-tree was used, mainly because it has been used with other modules in the Agile codebase; it has been tested and is known to work.

[4]TINA5: http://www.tina-vision.net/tina5.php

Chapter 3

LiDAR Segmentation

In an attempt to filter through the the LiDAR hits for those that represented objects of interest, we created a few segmentation modules.

3.1 Static 3D Grids

The first module subscribed to LiDAR messages and also to the navigation module's obstacle lists. The nagivation module creates the obstacle list and uses it to avoid obstacles while planning paths. From the list of obstacles, we created a list of 3D grids. We then took each LiDAR hit in the messages and determined whether each points fell inside the bounding boxes for each obstacle, and if it did, we put that point into the corresponding cell in the 3D grid.

This proved to be a decent first crack at the segmentation problem, as it did give us volumes of interest to focus on. However, creating a 3D grid for each obstacle and processing each obstacle separately lead to some problems. For starters, the obstacles in the obstacle lists did not necessarily span the entire obstacle. For example, a truck might be composed of many, many small volumes. This worked well for the navigation module as avoiding each small volume of an obstacle was the same as avoiding the entire obstacle. Segmenting this way, however, would mean that we were only looking at a very small volume of the obstacle at a time. This could lead to many inaccuracies.

In an attempt to remedy this problem, we grew the volumes a fixed amount in each dimension. This lead to a different problem: overlapping boxes lead to artificial edges in the lidar data as some boxes would steal the points from others.

To combat this problem, we created a new structure which was designed with resizing and merging in mind.

3.2 Dynamic 3D Grids

As the forklift moves through space, the navigation module publishes many messages about obstacles that lie in its path. By expanding those the obstacle bounding boxes and merging those bounding boxes with existing bounding boxes, we were able to get more complete views of objects in our LiDAR data, instead of only looking at a small part of the object at a time. However, some of these messages are false positives. In addition to the false positives, because we are growing the bounding boxes a fixed size, there are many bounding boxes that have little or no data in them. The Dynamic 3D Grids are periodically resized such that instead of spanning there whole space, they span only the sections that have at least n points in the cells. After resizing, the grids are recentered and ready to continue receiving LiDAR input.

3.3 KDTree

Because the Dynamic 3D Grids took longer than expected to debug, the Instrinsic Shape Signature module was implemented using information from the Static 3D Grids. Because the Static 3D Grids do not have any notion of overlapping one another, the union of all their LiDAR hits was pushed into a kd-ree. The kd-tree is a structure which does not have a preset boundaries, but finding neighbors is still a fast computation. The reason this was not ideal was the fact that instead of segmenting out individual objects in the lidar data and processing each separately in an attepmt to match them to known objects, taking the union of the LiDAR data and pushing it into a kd-tree meant that we were not really doing any segmentation. As the Dynamic 3D Grids did work, we eventually were performing segmentation; however, the

ISS module still takes this input and pushes the LiDAR hits from each segmented obstacle into a kd-tree. This step is no longer necessary, meaning that future work with the ISS module would entail moving the structure it uses internally to quickly compute neighbors from a kd-tree to a Dynamic 3D Grid. Using the Dynamic 3D Grid internally is better because finding the neighbors of a points becomes a constant operation instead of a search.

John Tsiombikas' implementation has been tested on other Agile modules which is why it was chosen as a placeholder as the Dynamic 3D Grids were being developed.

Chapter 4

Compute Salient Points

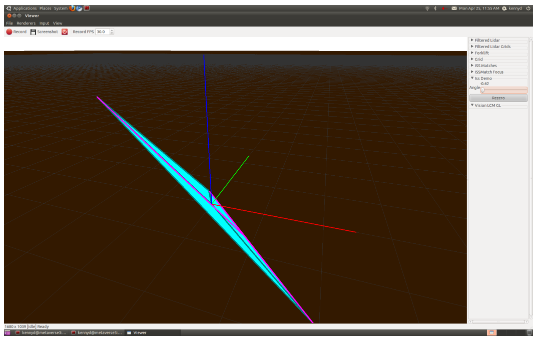
While the saliency computation outlined in [10] used the ratios of eigenvalues to find points that “stuck out” relative to their neighbors, we propose a different saliency calculation that can be done without first calculating the eigenvalues. Through communications with Dr. Zhong, we determined that a saliency calculation that instead looked for “cube-like” points would be faster and, hopefully, as informative.

To find “cube-like” points, we define saliency to be the volume-surface-area ratio; the neighborhood of points around each point define both the volume and surface area.

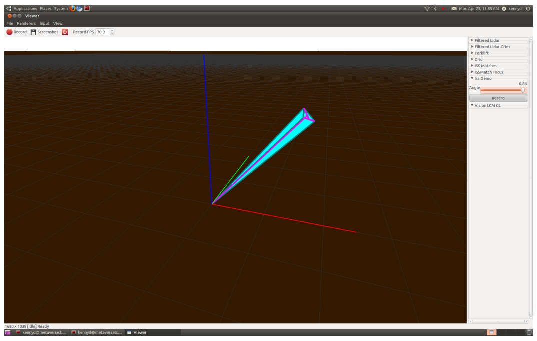
In this section, we will first give an example of the saliency computation, which comprehensively demonstrates why the saliency definition works as it does. We will then move through the math to show how we can compute the values of interest directly from the covariance matrix instead of first computing the eigenvalues.

4.1 Example of Saliency Computation

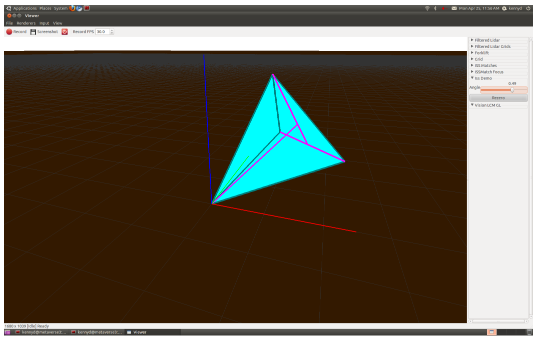
Imagine three vectors v_x, v_y, v_z, each initially aligned with the axes x, y, and z. Imagine the triangluar-based pyramid formed between then ends of these vectors and the origin (where the base is the face away from the origin)). Now, imaging that the vectors are each rotated a fixed angle from their initial positions: v_x rotates in the



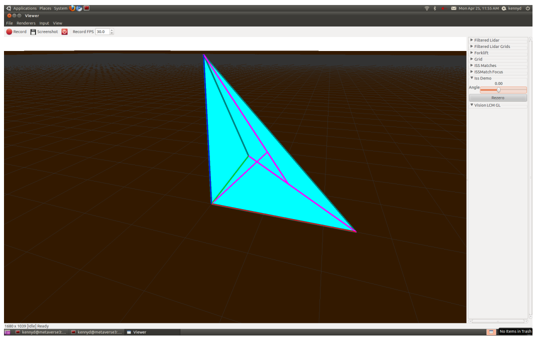
(a) Pancake



(b) Stick



(c) Intermediate



(d) Ideal

Figure 4-1: Visualization of volume-surface-area ratio changing as a function of angle. As will be shown later, this volume-surface-area ratio can also be described as a function of the eigenvalues. Here, values proportional to the eigenvalues are drawn in magenta. The eigenvectors fall along the principal directions of the solid, and the eigenvalues are proportional to the length of the solid along those directions.

plane $y = z$, v_y rotates in the plane $x = z$, and v_z rotates in the plane $x = y$. If the vectors are rotated $\approx -36°$, we end up with a flat pancake which has a high surface area and a volume of zero. This case is shown in Figure 4-1(a). If the vectors are rotated $\approx 54°$, we end up with a stick which has a surface area of zero and a volume of zero. This case is shown in Figure 4-1(b). By plotting the volume, surface area, and the ratio between the two as we move between those two extremes, we can see how the ratio changes. As can bee seen in Figure 4-2, the ratio peaks at a rotation of 0°. Therefore, by thresholding on this ratio, we should be able to determine which points are locally cube-like.

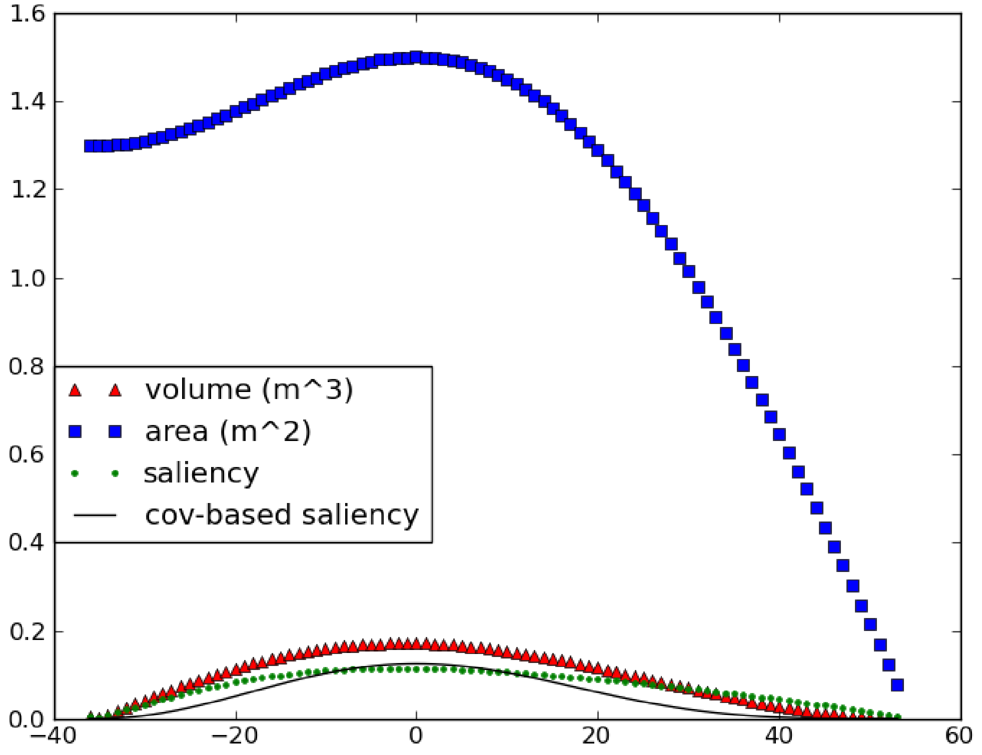


Figure 4-2: Saliency Calculation as a function of Angle (degrees): As we rotate a set of xyz axis-aligned vectors towards $\{1, 1, 1\}$, the surface area and volume of the triangular-based pyramid formed by those vectors change from a pancake at one extreme to a stick at the other. The peak in the volume-surface-area ratio occurs at $0°$. The "covariance-based saliency" calculation will be discussed in the subsequent section. It acts as a fast computation of the same saliency calculation.

4.2 Eigenvalue Definition of Saliency

We can defined the saliency as follows:

$$S_i = \frac{\text{Volume}}{\text{Surface Area}} = \frac{\prod \lambda}{\lambda_1\lambda_2 + \lambda_2\lambda_3 + \lambda_1\lambda_3} \tag{4.1}$$

where all λ-s are the eigenvalues of the covariance matrix COV_i. This volume-surface-area is valid because each of the eigenvalues represent a length in eigen-space, as depicted in Section 4.1. So, the product of all three eigenvalues would represent a rectangular prism volume, and the pairwise product would represent the area of the three visible sides of that rectangular volume.

We know that the eigenvalues represent lengths in this case by extending a similar two-dimensional problem, as described by Strang ([8]). If we have a 2D point cloud of roughly ellipsoidal nature in any orientation, and we calculated the covariance matrix at the centroid, the eigen-decomposition of that matrix would yield two eigenvectors: one along each of the major and semimajor axes. These eigenvectors would have eigenvalues with magnitudes proportional to the length of the major and semimajor axes respectively.

When we extend to a 3D problem, the eigenvectors still represent the principal directions of a neighborhood of the point cloud, and the eigenvalues still represent lengths. As the eigenvalues correspond to lengths, their product would be a volume and the sum of their pair-wise products would be a surface area (as stated previously).

4.3 Computing Saliency from the Covariance Matrix

In order to calculate this saliency value, we must first calculate the covariance matrix of a point neighborhood. The reason for using the covariance matrix is twofold. First, the covariance matrix allows us to directly compute the saliency. This will be shown in Equations 1-4. Secondly, it allows us to compute an eigen-basis representation of that neighborhood, which provides a reference frame in which to represent data about that neighborhood (similar to what happens in Principle Component Analysis[8]).

The covariance matrix can be calculated as follows:

$$COV_i = \frac{\displaystyle\sum_{j}^{N_i} w_j (p_j - p_i)(p_j - p_i)^T}{\displaystyle\sum_{j}^{N_i} w_j} \tag{4.2}$$

where w_j is the weight of the j-th point, p_j, such that

$$w_j = \frac{1}{size\{p_k | \|p_k - p_j\| < r\}} = \frac{1}{N_j} \tag{4.3}$$

where r is experimentally determined, though its value should be inversly proportional to the density of points in the point cloud. In this case, r was set at 0.3 meters. N_j is the number of points within a radius r of p_j, including p_j itself, so the weight is always guaranteed to be a positive number less than or equal to 1.

From Equation 4.1, we can go backwards, using definitions of eigenvalues, to a simpler expression that does not require taking the eigen-decomposition of the covariance matrix, but can instead be computed directly from the covariance matrix.

We can rewrite the numerator as

$$|COV_i| = \prod \lambda$$

by the eigenvalue definition of the determinant; and we can rewrite the denominator as

$$\frac{1}{2}\left(\left(\sum \lambda\right)^2 - \sum \lambda^2\right) = \lambda_1\lambda_2 + \lambda_2\lambda_3 + \lambda_1\lambda_3$$

by algebra. This leads us to a simpler expression for saliency:

$$S_i = \frac{2|COV_i|}{(\sum \lambda)^2 - \sum \lambda^2}$$

This expression can be simplified further by the eigenvalue definition of the trace:

$$\text{trace}(COV_i) = \sum \lambda$$

The last expression involving eigenvalues can be eliminated via the following:

$$\sum_r \sum_c COV_i[r][c]^2 = \sum \lambda^2$$

This leaves us with the following expression for saliency.

$$S_i = \frac{2|COV_i|}{\text{trace}^2(COV_i) - \sum_r \sum_c COV_i[r][c]^2} \tag{4.4}$$

This expression is better than the original expression (Equation 4.1) because it involves only the covariance matrix and simple operations on it. It does not require the eigen-decomposition of the covariance matrix, which is computationally expensive. By setting a saliency threshold S_T, we can filter through the points and focus our analysis on only the ones that are most important.

4.4 Basic Algorithm for Computing Saliency

Using the previously defined equations, we can express the pseudocode for computing saliency as the following.

```
$salient\_list \leftarrow []$
for $p_i$ in $all\_points$ do
    $w_i \leftarrow$computeWeight($p_i$,$all\_points$) // (Equation 4.3)
end for
for $p_i$ in $all\_points$ do
    $cov_i \leftarrow$generateCov($p_i$,$all\_points$) // (Equation 4.2)
    $s_i \leftarrow$computeSaliencyFromCov($cov_i$) // (Equation 4.4)
    if $s_i \geq S_T$ then
        $eigs_i \leftarrow$computeEigenDecomposition($cov_i$)
        $salient\_list \leftarrow salient\_list$+makeFeaturePoint($p_i$,$eigs_i$)
    end if
end for
```

Chapter 5

Computing Feature Vectors

In Chapter 4, we discussed how we determined which points were cube-like and of enough importance to process further. In this chapter, we will discuss how we use the eigenbasis of a point and the points in its neighborhood to construct a unique *signature* which will be used to compare a neighborhood of points in a point cloud to a signature from a model.

5.1 Discretizing the Space around a Point

The feature vector of a salient point fp_i is a vector whose cells represent a partition of the sphere around that point and whose cell values represent the weighted sum of neighbors in that partition. We chose to use Horn's model[2] to discretize the space around a salient point over other gridding options because the other gridding options to not afford much rotational tolerance. A rectangular grid has an issue with rotations: because the cells do not vary in size with distance from the origin, the cells furthest from the origin restrict the rotational tolerance of the grid. A latitude-longitude-altitude grid also has a rotational problem: many cells converge at the poles, meaning that a small rotational error could cause points to be in many different cells, which is a big problem when trying to compute the distance from one feature vector to the next.

To maximize the amount of rotational error allowed by the grid, we chose to use

Horn's model starting with a base octahedron. This caused us to discretize the space into 66 cone-like regions. We chose to partition each of those radially into 10 bins, with the maximum distance being 1.5 meters. This scheme partitions the space into 660 bins. However, because many cells converge at the origin, there is the potential for a small translational error leading to problems with distance calculation. To minimize this possibility, we merge the cells that are in the bins closest to the origin into a single spherical bin.

Our feature vectors are therefore $N_{fp} = 66 * (10 - 1) + 1 = 595$ partitions of space that should provide us with the best translational and rotational tolerances possible.

5.2 Computing the Vector Values

Now that the physical structure of the feature vector has been explained, we will discuss what the values in those elements represent. The values in the feature vector are calculated as follows:

$$vec_i[l] = \sum_{j}^{N_i} w_j \text{ such that } p_j \in l$$

meaning that the l-th element in the feature vector is the sum of the weights of the neighboring points that fall into partition l.

To determine which bin l a point p_j falls into, we must first compute the vector from p_i to p_j in the reference frame of p_i. The reference frame of p_i was calculated in Chapter 4: the orthonormal eigen basis allows us to directly compute the projection of any vector into p_i's reference frame.

$$p_{proj} = eig_i^T (p_j - p_i)$$

Once we have p_{proj}, we can determine which bin l it falls into by computing the spherical and radial indices separately. The spherical index will be the closest of the 66 points that discretize the sphere in Horn's model (see Section 5.1). The radial index can be computed from the magnitude $\|p_{proj}\|$.

In an attempt to smooth over noise that may exist in the LiDAR data, we additionally use add-λ smoothing. This smoothing technique adds a small amount of probability, λ, to each cell and then renormalizes. The effect of this is a minor reallocation of probability: bins with high probability will decrease slightly while the bins with small probability will be given a bit more.

There are two main benefits of using add-λ smoothing. The first is that we smooth out some of the noise that may exist in our LiDAR data. The second is that each partition has a non-zero probability associated with it. This is good because then we can use the standard χ^2 distance metric for calculating the distance between two feature vectors. This will be discussed in more detail in Section 6.1.

Chapter 6

Matching

In Chapter 4, we discussed how to compute which points were "cube-like" and therefore if importance for futher processing. In Chapter 5, we discussed how to construct a signature for each point. In this chapter we will discuss the next step in the recognition process: taking a list of salient-model matches and computing a pose estimation between the point cloud and a model.

6.1 Feature Vector Distance Metric

There are a number of distance metrics we could have used, but we chose to use the standard χ^2 distance metric because it is similar to the distance metric used in [10].

$$distance(p_i, p_j) = \underset{k}{\operatorname{argmin}} \sum_{l=0}^{N_{fp}-1} \frac{(p_i[l] - p_j[rotate(l,k)])^2}{p_j[rotate(l,k)]} \tag{6.1}$$

Our metric differs slightly because we used add-λ smoothing to reallocate probabilities throughout all the N_{fp} bins such that there is no possibility of there being a zero-weighted bin.

6.2 Avoiding One-to-Many Matches

In order to avoid the one-to-many matching problem in which a single model point matches many salient points in the point cloud (or vice versa), we create a list containing all matches and filter through the list to ensure that each point is used only once. To make sure that each point is only used in its minimum-distance match, we first sort the list of matches by distance.

The result of this sorting and filtering is a list of the matches such that each match represents the minimum-distance match for each signature and each signature is used only once.

6.3 Computing Model Transformations

Once we have a list of matching signatures, we can compute the transformation from the model coordinate system to local coordinates. If there are enough matches that agree on one specific transformation, then we can declare a match and publish it, along with its associated transformation, so that information can be used in other (higher-level) processes.

6.3.1 Using a Histogram to Compute Average Rotation

To start, we create a histogram of possible rotations. For each match in the list of matches, we calculate the Euler Angles (roll, pitch, and yaw) associated with the rotation between the model signature and the query signature. We then add that match to the histogram, using a discretization of roll, pitch, and yaw as indices. Once the initial frequency histogram has been fully populated, we gaussian blur the histogram to make our solution robust to noise. The average rotation is computed by iterating over the gaussian-blurred histogram and finding the cell with the highest vote.

One issue with this method is that there are an infinte number of equivalent Euler Angle (roll-pitch-yaw) combinations. By discretizing over the roll-pitch-yaw values, we risk spreading out equivalent votes into many bins.

Instead of discretizing over Euler Angles, a discretization of the quaternion representation of the rotation was used. A *quaternion* is a 4-tuple which is an elegant representation of a 3-D rotation; even with only these four numbers, rotation transformations can be performed, whereas with the Euler Angles, the rotation matrix ($3x3$) must be computed first before applying any transformations).

A quaternion is defined as

$$\mathbf{R}(\theta, \vec{r}) \rightarrow \hat{\mathbf{q}} = \{q_w, q_x, q_y, q_z\}\{\cos\theta, r_x \sin\theta, r_y \sin\theta, r_z \sin\theta\}$$

In this representation, we can discretize the range of each element $[-1.0, 1.0]$ into many bins. Moreover, it is possible to restrict the value of q_w to a solely positive value using the following rule:

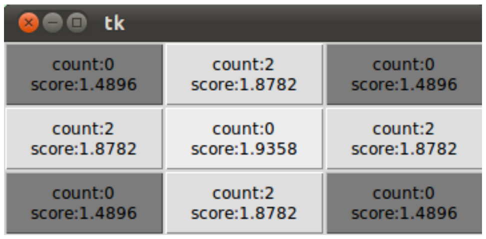
$$\hat{\mathbf{q}} = s\hat{\mathbf{q}} = -\hat{\mathbf{q}}$$

All nonzero scalar multiples of a quaternion are equivalent, so if q_w is negative, we can negate the whole quaternion, giving us an equivalent representation.[7] By doing this, we can avoid the problem we had with Euler Angles: each cell in the histogram must represent a unique rotation.

6.3.2 Computing Average Translation

Computing average translation is done in a different manner and is completely dependent on getting a good rotation estimate. Because we gaussian blurred the histogram, there was the potential for a cell with no votes to win, as would be the case in Figure 6-1.

As can be seen, the unblurred raw frequency histogram has no votes in the center cell while the blurred histogram would choose it as a winner over its neighbors. To anticipate this problem, the translation estimate was edited slightly from [10] to the



count:0 score:1.4896	count:2 score:1.8782	count:0 score:1.4896
count:2 score:1.8782	count:0 score:1.9358	count:2 score:1.8782
count:0 score:1.4896	count:2 score:1.8782	count:0 score:1.4896

Figure 6-1: Demonstrative Gaussian Blurring Example: In this figure, you will note that by gaussian blurring the frequency counts, the central bin ends up with the highest score. Using that bin would not allow us to compute a translational estimate as there are no actual counts in the bin itself.

following:

$$\mathbf{T} = \frac{\sum_{k}^{K} w_k \sum_{l}^{L_k} \frac{p_{li} - \mathbf{R} p_{lj}}{L_k}}{\sum_{k}^{K} w_k} \tag{6.2}$$

Where K is the number cells within the blurring radius of the winning cell $\mathbf{R}$, and L is the number of hits in each of those cells. Instead of only looking at the cell with the hightest vote to compute the translation, we instead use the rotation estimate associated with that cell, but do a translational estimate which is a weighted sum over the translational estimate from all the hits in the neighborhood of that cell. The effect is that we get a translational estimate even if there are no hits in that particular cell.

6.3.3 Confidence

Although there are many possible metrics for determining the confidence of a match, we used the value in the winning cell of the blurred histogram. This value is a function of the number of matching points that agree on a pose estimate. A high number indicates many matches, which should indicate high confidence; likewise, a low number indicates few matches, which should indicate low confidence.

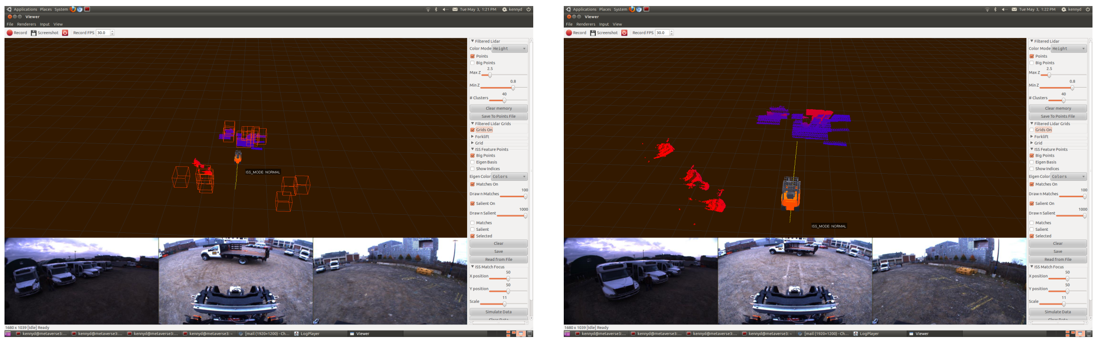
Chapter 7

Renderers

Many renderers were created to ensure that the program was behaving as desired. A few are mentioned here with a description of its function along with a screen-shot.

7.1 Filtered Lidar

The lidar messages were the input into the intrinsic shape signature module. In order to verify that lidar hits were being buffered properly, this renderer was created. Notice the orange boxes which indicate the regions of interest. The buffer only stores points that fall into one of the current orange boxes, which individually represent anything the forklift's obstacle detection module has deemed relevant.



(a) Filtered Lidar Grids

(b) Filtered Lidar

Figure 7-1: Filtered Lidar Renderer: 7-1(a) depicts the grids used to store the LiDAR hits. 7-1(b) depicts the lidar hits that have been stored.

7.2 Feature Detail

On receiving a buffered lidar message, the intrinsic shape signature module creates a *kd-tree* of the points. This information allows for quick access of neighbors. Neighbor information was used to determine the weight of each point and then later to determine the feature vector of each salient point. This process was discussed in great detail in Chapters 4 and 5. Figure 7-2 shows which points factored into one of the entries in the feature vector of this salient point.

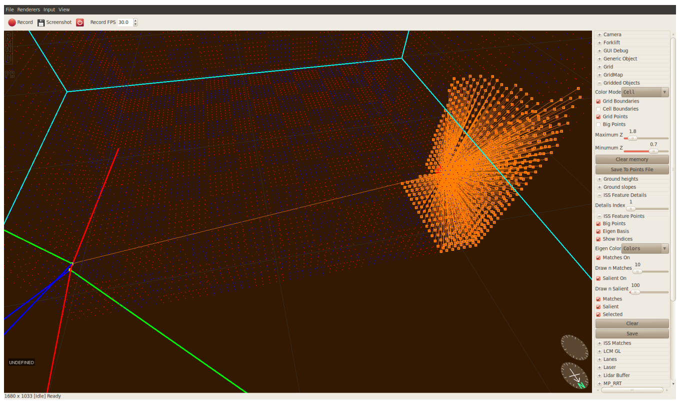


Figure 7-2: Feature Detail Renderer: This image shows how the points in the neighborhood of a point contribute their weight to a feature vector entry.

7.3 Feature Points

In order to visualize the recognition algorithm, it was useful to render points that met our saliency threshold. We created a renderer that draws the points as well as their associated eigenbasis triads. The triads were colored red, green, and blue for the x-, y-, and z-axes respectively. This allowed us to visually verify that the eigen-bases looked reasonable, with the z-axis roughly normal or antinormal to the surface.

In addition, this renderer allowed for the focused rendering of a salient point so that each feature point could be examined individually. Specifically, we could examine which entries were populated in the feature vector associated with each point.

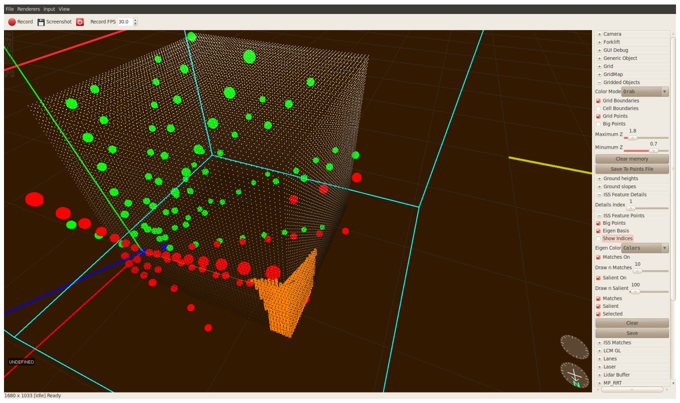
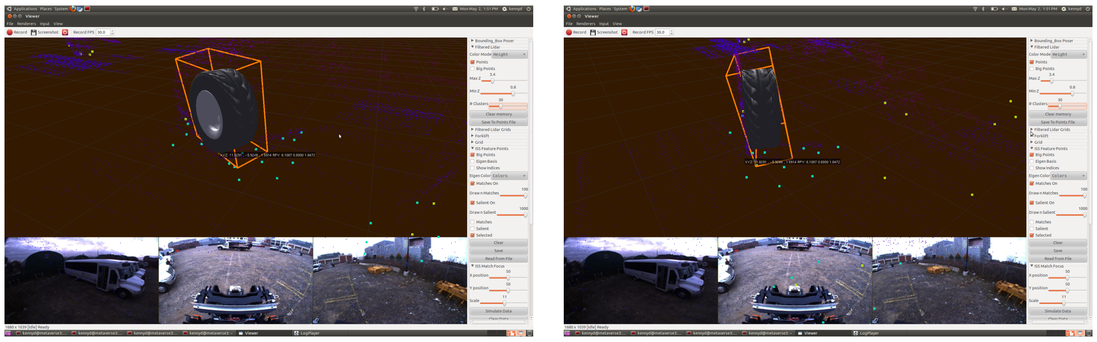


Figure 7-3: All Elements: This image shows the salient point and all of its populated feature vector elements.

7.4 Match Lists

Although matching salients were published individually as Feature Points and rendered much in the same way, they were also amalgamated into lists of matches with common rotational and translational pose estimates. These pose estimates represent actual model matches over many points: the output of the intrinsic shape signature process.



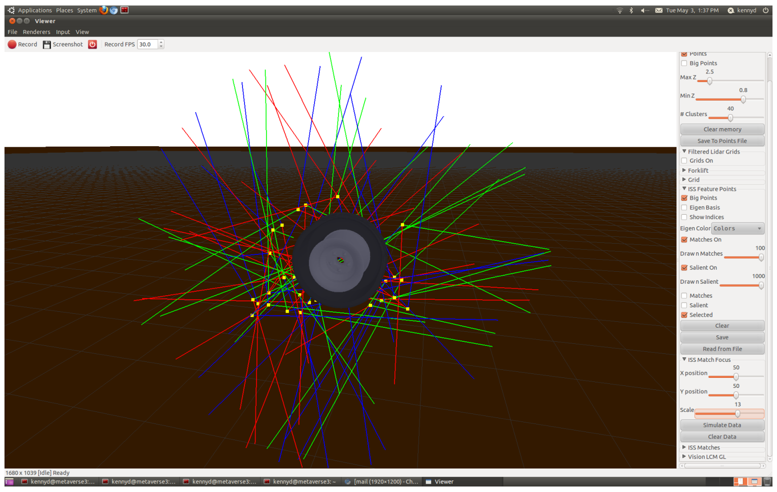
(a) Isometric(-like) View (b) Side View

Figure 7-4: Match List Renderer: In these images, the ground truth pose for a tire is shown by the opaque solid model. The match pose estimate is shown by the orange bounding boxes. 7-4(a) shows a match. 7-4(b) shows the same match at a different angle to make the slight angular error more apparent. This error is due to the angle discretization in the histogram (discussed in Chapter 6).

Although there is much more information in the match lists, it was difficult to visualize the information. Initially, lines were drawn from the model's points to the world points, but that was largely unhelpful. The rotational error information was also difficult to interpret when visualized. The simple bounding box with the solid model was deemed more digestible while still informative.

7.5 Models

A renderer was created to ensure that the models looked as expected, meaning that any transformations being applied to the model and associated intrinsic shape signatures were being applied in the correct order. The renderer allows the user to look at a model at a fixed pixel in the viewframe. The model is oriented to always point towards north, so if as the user moves in the viewer's 3D environment, the model spins, but its center is fixed at the desired pixel. The user can also specify the scalinig of the model, so it could act as a widget in the corner, or it could be enlarged for a more detailed inspection. Figure 7-5 depicts the model used for finding tires.

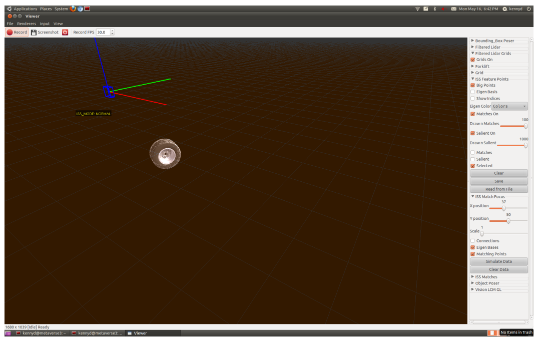


Points [✓]
Big Points [ ]
Max Z 2.5
Min Z 0.8
Clusters 40
Filtered Lidar Grids
Grids On [ ]
Forklift
Grid
ISS Feature Points
Big Points [✓]
Eigen Basis [ ]
Show Indices [ ]
Eigen Color Colors
Matches On [✓]
Draw n Matches 100
Salient On [✓]
Draw n Salient 1000
Matches [ ]
Salient [ ]
Selected [✓]
ISS Match Focus
X position 50
Y position 50
Scale 13

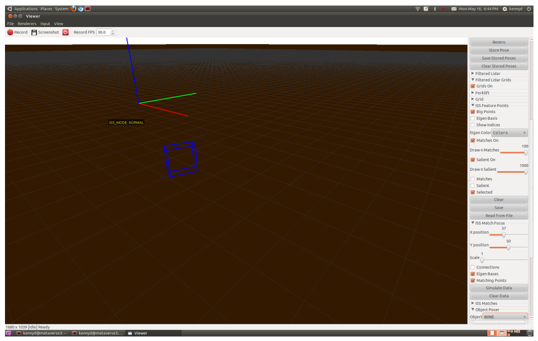
Figure 7-5: Model Renderer: This shows the model used for detecting tires. Note that the solid model is not quite the same as the actual tires being detected and is used notionally. The actual tire has a slightly larger radius, which can be seen in the position of the signatures. Note that a few points in the ground plane near the tire are also included in the model.

7.6 Transformers

In order to ensure that the rotation/translation estimates were consistent across all modules, we created a SolidModelTransformer and BoundingBoxTransformer which were used to translate and rotate objects and bounding boxes from the origin to locations of interest in the world. This way, we could quickly visually determine what the expected transformation should be and verify that our recognition algorithm's pose estimate was producing similar results.



(a) Solid Model Transformer



(b) Bounding Box Transformer

Figure 7-6: Transformers: 7-6(a) shows a truck tire undergoing a pose transformation to fit our model. 7-6(b) shows the bounding box of that truck tire undergoing the same transformation.

The transformers were used as the "ground truth" when running simulations, meaning that we would place the transparent solid model or the blue bounding box at the locations where the objects actually appeared, expecting that our recognition module would give similar estimates. The solid model was rendered as transparent to differentiate it from the recognition module's pose estimates. Similarly, the blue bounding boxes are blue to distinguish them from the recognition module's pink bounding box pose estimates.

Chapter 8

Results

In this chapter, we will discuss our results.

8.1 LiDAR Calibration Limitations

One of the initial problems was that the LiDAR were not perfectly calibrated. This was a problem because instead of gaining more information by using more sensors, the extra data serves only to confound the information provided by any one sensor, giving multiple ground planes and surface approximations.

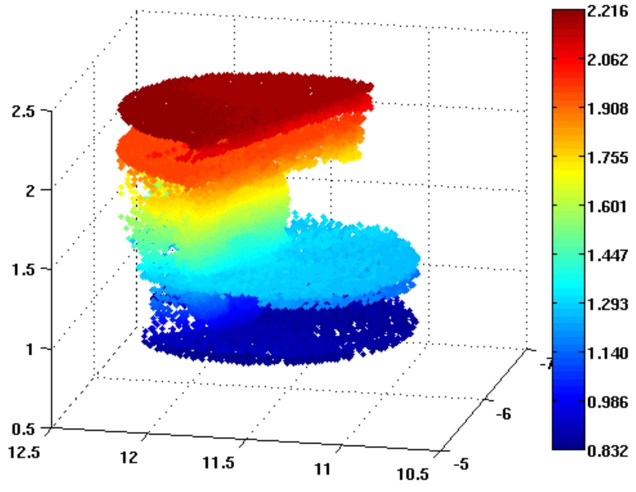


Figure 8-1: Uncalibrated LiDAR sensors: This image shows the fact that the sensors were not perfectly calibrated. Note the different estimates of the ground plane.

As a result, only one top-center mounted sensor was used, but the process was designed such that adding other sensors would be easy so when the sensors are properly calibrated, they can be used together.

8.2 Threshold Optimization

Once the segmentation module and the Intrinsic Shape module were completed, there were many thresholds that needed to be optimized. Initially, the thresholds were too lax which resulted in many false positives and, because each cloud can only match up to one of each type of model, a few false negatives. In an effort to remedy this issue, the thresholds were tweaked slowly until the truck tire in the training LCM log matched perfectly. This worked well on that particular LCM log, but did not work on other logs at all.

We determined that there were really three thresholds which were most important in determining saliency and matches. These three thresholds were

- the Saliency threshold, which determines whether or not a point was a salient (see Chapter 4),
- the Match distance threshold, which determines whether two signatures have a feature vector that is close enough to declare them a match (see Chapter 6), and
- the Confidence threshold, which determines if there are enough matching points that agree on a pose estimate to publish the match to other modules.

We wrote an optimization module for these thresholds which talked back and forth with the Intrinsic Shape Signature module; it worked by tweaking the thresholds, publishing a new LiDAR message, waiting for the Intrinsic Shape Signature to finish processing the data, and then evaluated the results.

We were able to determine the best values for these thresholds on our training data.

Variable	Value
Saliency Threshold	0.002
Match Distance Threshold	0.005
Confidence Threshold	2.400

Table 8.1: Optimized Threshold Values

Using these settings, the module was able to determine tires very accurately (no false positves or false negatives), as depicted in Figure 8-2.

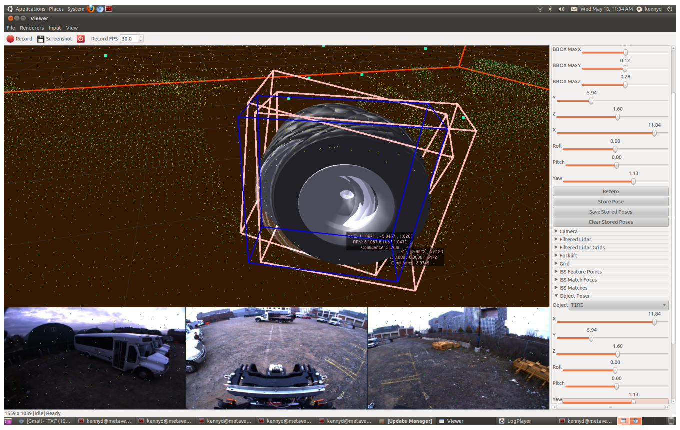


Figure 8-2: Tire Match Results: Tire Matches worked perfectly on training logs using optimized threshold settings. Here we see a cloud that has been processed twice. Each pose estimate (pink bounding box with opaque solid model) falls within the rotational and translational thresholds of ground truth (blue bounding box with transparent solid model).

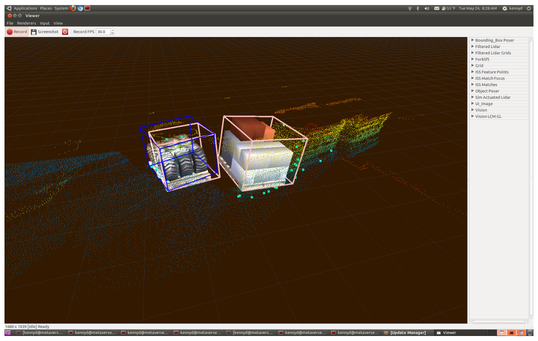
On another log file, under the same settings, the module was able to recognize some pallets with similar results, while completely unable to find others. The results are depicted in Figure 8.2. In the Figure, there are two additional pallets to the right of the JMIC pallet. These are Ammo pallets. They are not detected at all, regardless of thresholds. This may be a result of a number of things:

1. Occlusions and self-occlusions in the data mean that we only see a very small part of the pallet (mostly only one face);

2. Ammo pallets are similar to JMIC pallets, so the JMIC might be matching a few times instead, stealing points away from the ammo pallet such that it cannot

match while either only changing the JMIC pose estimate slightly or not at all if they fall into a different region of the histogram.

3. The models were less developed than the tire, JMIC, and tire pallet models, so it is possible that the signatures being used to match were not specific enough to the Ammo pallet.


(a) Tire Pallet

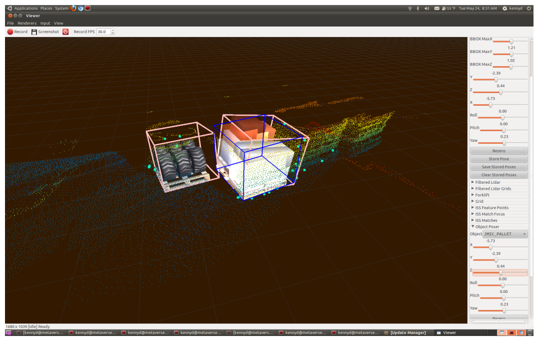

(b) JMIC Pallet

Figure 8-3: Pallet Matching: 8-3(a) depicts the a tire pallet being matched. 7-1(b) a JMIC pallet being matched.

As can be seen in the figure, we are able to match Tire Pallets and JMIC pallets without false positives or false negatives. However, the Ammo pallets to the right do not match at all.

8.3 Runtime

The average time to process a cloud of 20000 points is roughly 30 seconds. The data segmentation module publishes at a rate of one 20000 point message per second. This of course leads to an issue with the Intrinsic Shape Signature queue overflowing. Very few of the messages are actually processed. However, because the segmentation module publishes indiscriminantly, it publishes the same areas over and over again. Although the Intrinsic Shape Signature module only processes $\frac{1}{30}$ of the messages it receives, it does process messages that span the whole space the forklift can see. That is to say that the data segmentation module publishes the same information each second, meaning that much of the information is redundant.

The module does not run in real time in that it cannot process messages faster than it receives them, however, it does process at least a message in each region of interest.

8.4 Contributions

We have implemented a system that does data segmentation and object recognition in real time. The object recognition algorithm was an implementation of that described by Zhong [10]. We showed that the saliency calculation can be done before the eigenbasis calculations and that recognition can still performed using this new metric for saliency.

8.5 Future Extensions

Future extensions of this project include:

1. Editing the ISS module so that it uses a Dynamic 3D Grid interally instead of a *k*d-tree;
2. Improving the segmentation module such that instead of chunking out rather large areas of space, it chunks out individual objects;
3. Storing Salient points over time and see how that affects matching accuracy as well as 6-D pose estimates; and
4. Removing the Tina5 dependency by implementing an eigendecomposition method using arrays that work with the existing Agile rotation code.

Bibliography

[1] P. Cho, H. Anderson, R. Hatch, and P. Ramaswami, *Real-Time 3D Ladar Imaging*, IEEE Applied Image and Pattern Recognition Workshop (AIPR), 2006

[2] B. K. P. Horn, *Robot Vision*, MIT Press. Cambridge, MA, 1986

[3] A. Huang, E. Olsen, and D. Moore, *Lightweight Communications and Marshalling for Low-Latency Interprocess Communication*, MIT,CSAIL Technical Report, MIT-CSAIL-TR-2009-041, 2009

[4] J. Leonard, et al., *Team MIT Urban Challenge Technical Report*, MIT,CSAIL Technical Report, MIT-CSAIL-TR-2007-058, 2007

[5] J. Leonard, et al., *A Perception-Driven Autonomous Urban Vehicle*, Journal of Field Robotics, vol. 25, no. 10, pp. 727-774, June 2008

[6] B. Matei, Yi Tan, Harpreet S. Sawhney, and Rakesh Kumar, *Rapid and scalable 3D object recognition using lidar data*, Proc. SPIE 6234, 623401 (2006), DOI:10.1117/12.666235

[7] T. Möller, E. Haines, and N. Hoffman, *Real-Time Rendering*, A K Peters Ltd, 2008.

[8] G. Strang, *Computational Science and Engineering*, Wellesley-Cambridge Press, 2007.

[9] S. Teller, et. al, *A Voice-Commandable Robotic Forklift Working Alongside Humans in Minimally-Prepared Outdoor Environments*, In Proc. IEEE Int'l Conf. on Robotics and Automation (ICRA), May 2010

[10] Y. Zhong, *Intrinsic Shape Signatures: A Shape Descriptor for 3D Object Recognition*, In Int'l Conf. on Computer Vision Workshops (ICCV), 2009