3D Object Tracking Using the Kinect

Michael Fleder, Sudeep Pillai, Jeremy Scott - MIT CSAIL, 6.870

Abstract

Tracking an object's 3D location and pose has generally been limited to high-cost systems that require instrumentation of the target object and its environment. We investigate the possibility of using only the Microsoft Kinect's depth and RGB image data to track objects at a low cost, without any object instrumentation. First, we describe our GUI and viewer for 3D Kinect data that allows the user to quickly segment the target object's point cloud. Next, a 2D tracking algorithm computes a color histogram of the segmented point cloud and searches for it in live image data to guess the object's location as it moves. Finally, seeded by this guess, the iterative closest point (ICP) algorithm is used to estimate the transformation of the object's original point cloud in the live point cloud. Along with other optimizations, we show that combining 2D tracking and ICP makes it possible to perform real-time, accurate 3D object tracking using the Kinect.

1. Background and Motivation

Object tracking, or video tracking, is the task of capturing the 3D position and pose of an object from frame to frame. It has become an important tool in many applications, ranging from HCI to robotics. In the film industry, producers have been able to generate realistic character animations that mirror an actor's tracked movement in the real world. In robotics, a robot can not only follow a tracked object over time, but can also learn about 3D manipulations that can be performed on that object. In many more sub-fields of AI, 3D information arguably offers a new modality for learning, analysis and action.

Until recently, the benefits of 3D object tracking have been reserved for those who can afford high-cost motion tracking systems, such as those offered by Vicon, which are priced upwards of $10,000, These systems involve placing reflective markers on the tracked object, describing its 3D structure in a GUI, and then placing many IR cameras around the room to capture and synthesize various perspectives on the object to compute the position and pose of the tracked object. Although the tracking data produced by these systems is precise and reliable for rigid models, the calibration process is fragile and needs to be repeated anytime a camera is accidentally moved. Furthermore, requiring a user to manually mark and describe the 3D shape of an object decreases the usability of the system and limits the number of scenarios in which objects can be tracked. Finally, as mentioned before, the cost of these systems is above-budget for most individuals and many small R&D labs, making the world of 3D object tracking largely inaccessible.

With the advent of the Microsoft Kinect priced cheaply at around $130, depth data in indoor scenes has been made available to a much larger development community. While most projects utilizing the Kinect have focused on direct human-computer interaction involving human skeleton tracking and gesture recognition, relatively little research has focused on 3D object tracking. This is an exciting prospect, not only because of the Kinect's low cost, but because the tracking methodology would require little instrumentation of the environment (one camera vs. >3 cameras) and no physical marking of the tracked object. In this project, we show that it is possible to combine data from the Kinect's depth and RGB cameras to track objects in real time, at a low cost and without any object instrumentation.

2. System overview

The system has three main components that work in a pipeline to track an object's 3D position and pose. The first is a graphical user interface (GUI) that allows the user to pan, rotate and zoom around the 3D scene captured by the Kinect, quickly identify (*segment*) objects of interest, and control which objects are being tracked by the system. The second is a 2D tracker that searches for the segmented object in live RGB images from the Kinect, using a color histogram mean-shift algorithm. The third is a 3D object tracker, which uses the iterative closest point (ICP) algorithm to search for the original point cloud in the live depth image. The 3D tracker is seeded by the 2D tracker's result, greatly reducing the confusion of searching for the object in the entire live point cloud.

The pipeline is outlined below.

1. The Kinect continuously broadcasts LCM messages, each containing data for a single frame: a 640x480 depth and RGB image.
2. The GUI receives live Kinect frames and renders the 3D scene. The user can pan, rotate and zoom around the scene.
3. When the user wants to track an object, the GUI stops rendering live Kinect frames and pauses the scene. The user segments the object's point cloud, which is published over LCM to the 2D tracker.
4. The 2D tracker computes a color histogram based on the RGB values of the segmented point cloud. A mean-shift algorithm searches for this histogram in live RGB images from the Kinect and returns an ellipse surrounding the most likely position of the tracked object. The indices of the ellipse's points are published over LCM to the 3D tracker.
5. The 3D tracker translates the ellipse into 3D points in the live point cloud. Iterative closest point (ICP) is used to search for the original segmented point cloud in the subset (ellipse) of the live point cloud. ICP returns the most likely transformation of the object.

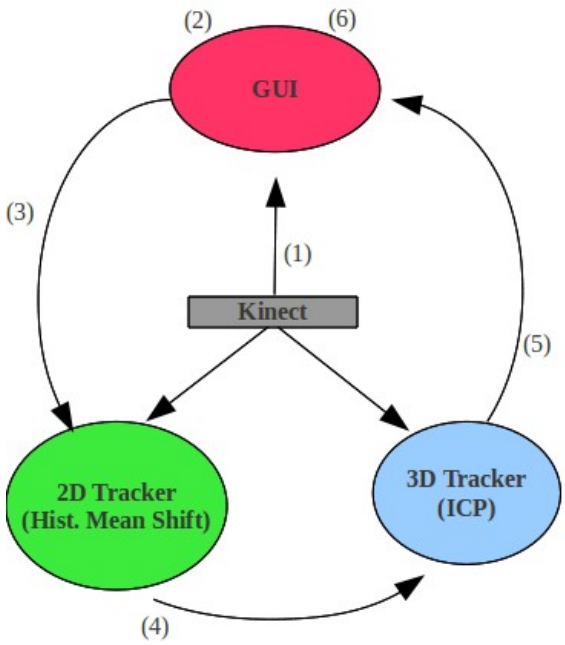

Figure 1: The tracking pipeline

6. Steps 4 and 5 repeat as new frames are received from the Kinect. The GUI resumes rendering the live scene and overlays the 3D tracker's predicted object transformation in bright green.

3. GUI and User-Assisted Segmentation

The GUI is the only point of interaction between the user and the object tracking pipeline. It is responsible for two main functions: *i*) rendering the 3D scene from the Kinect and *ii*) allowing the user to segment objects to be tracked.

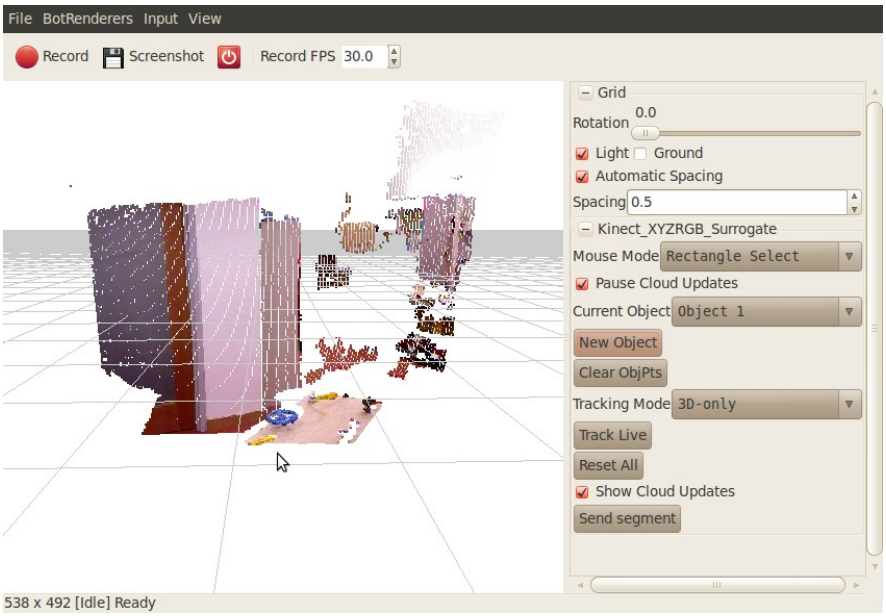

Figure 2: Screenshot of the GUI before any segmentation has been performed.

The user can pan, rotate and zoom the camera by left-clicking, right-clicking and scrolling the mouse wheel, respectively. When the user wants to track an object, they first click the *New Object* button to create a new tracked object. The user then presses the *'S'* key to switch the mouse from *Camera Movement* to *Segmentation* mode. In *Segmentation* mode, the left-mouse acts as a selection tool for 3D points, rather than pan control. Figure 3 shows a user segmenting the blue wheel in the above scene. Points are added to the object's segmented point cloud by holding *Shift* and selecting a rectangle of points with the left-click button (similar to selecting multiple files on a desktop). While points are being selected, they are highlighted in green (Fig. 3a). Once selected, the points are added to the object's point cloud and highlighted in an object-specific color (Fig. 3b).

Figure 3: User segmenting the wheel from above (a, b), then refining the point cloud from the side (c, d)

Point selection is performed in OpenGL by examining each point in the scene and determining if its projection onto the user's viewport is within the user's drawn selection rectangle. Inevitably, selecting a rectangle of points will include points not part of the target object. In this example, parts of the table are included in the segmented point cloud (red in Figure 3b), along with the wheel's points. In order to deselect these points, the user changes their perspective by switching back to camera mode (*C* key) and

rotating to a side perspective. Now, when the user selects the wheel's points, they are distinct from the table's points (Figure 3c). The GUI takes the intersection of the previously selected point cloud with the newly selected point cloud to determine the final segmented point cloud for the wheel (Figure 3d). In this fashion, the user can add and intersect groups of points from different perspectives to quickly segment the object they want to track.

Once the object has been segmented, the user presses *Send Segment* to publish the segmented point cloud to the 2D tracker over LCM. Upon pressing *Track Live*, the GUI resumes rendering the live 3D scene. The GUI also overlays the 3D tracker's output, highlighting the transformed object in bright green.

4. 2D Object Tracking

4.1 Preliminary Experiments

Our initial approach in tracking the 2D objects was to build 2D descriptors for different view points for each object. FAST [1] (Features from Accelerated Segment Test) features essentially look at a patch (specifically, a Bresenham circle of radius 16) to see if the nucleus is a minima or maxima with respect to the gray-scale intensity of pixels that lie on the circle. In order to be computationally efficient it also builds short decision trees to solve this problem. Since they have proven to be reasonably stable features with some lighting invariance, our algorithm first looks for saliency in the image by running the FAST algorithm on the live gray-scale image. Essentially, by determining regions of high saliency using FAST, the algorithm reduces the search space for finding and building feature descriptors by providing a saliency mask image.

The choice of feature descriptors was purely based on empirical results, as we were uncertain about the resulting computational expense of each feature descriptor on the mask image. We integrated various feature descriptors, specifically SIFT, SURF and ORB [2] (Oriented FAST and Rotated BRIEF). Among the 3 descriptors, SIFT produced the most consistent results requiring the least number of descriptors (on the order of 25) per viewpoint. ORB on the other hand required many more descriptors for the same reliability in detection of the features. SURF fell in between in terms of the consistency in detecting and recognizing the features. In terms of computational speed, SIFT was computationally most expensive and ORB was the least, leaving SURF to be fairly fast.

4.2 Preliminary Analysis and Results

Using the above algorithm, our system recognized features on the object fairly well. During the training phase, the object would be rotated so as to build descriptors for each viewpoint, and later the features would be matched across all the descriptors that described each viewpoint. Further filtering could have helped eliminate viewpoint discrepancies, by only training the object on a select few viewpoints that are significantly different from the others. The feature detection scheme did however have several outliers that were hard to manage. Furthermore, the RGB sensor on the Kinect had a low signal-to-noise ratio which led to the detection of spurious points in the image on running the FAST algorithm. Adding more features to the image implied that the search space on which the feature descriptors ran their algorithm was much larger. This further reduced the overall efficiency of the algorithm. The difficulty in dealing with false positives and inefficiency of our initial object tracking system led us to believe that tracking color distributions in a low-resolution image may be more feasible and efficient for our motion capture system.

4.3 Appearance-based tracking

Since matching feature descriptors across images that were inherently noisy turned out to be a difficult task, we looked in to taking advantage of the color distribution of an object as a metric for matching across frames. One approach that seemed promising was to utilize histogram-based appearance models primarily since they are more representative of a colored object over multiple viewpoints. Furthermore, it is computationally efficient while also providing a sufficiently strong belief of the location of the object in the image.

The algorithm we employed is mostly derived from the original Continuously Adaptive Mean Shift (CAMSHIFT) algorithm [3] implementation by Bradski. G. The algorithm is based on a robust non-parametric technique for climbing density gradients to find the mode of probability distributions called the mean-shift algorithm. In this case, the mean-shift algorithm is modified to deal with dynamically changing color probability distributions.

4.4 Implementation and Analysis

In order to use CAMSHIFT to track the colored object in the scene, a probability distribution image of the desired color in the live scene had to be created. In order to do this, we first create a model of the desired hue using a color histogram. The HSV (Hue, Saturation, Value) color space was chosen as it parametrizes the color space such that the hue tells us enough about the shade of a specific colored object. Thus, the algorithm is initialized with a 1D color histogram of the object in the H space. Once the color model of the object is built, the back-projection of the object color model histogram on the live image's hue channel is calculated. Bhattacharya Coefficient can be used as a metric to compute this backprojection/overlap between histograms. The resulting back-projection image is a grayscale image representing the belief of whether the pixel in the image belongs to the histogram of the object. It is on this gray-scale image that the mean-shift algorithm is computed to find the modes of high probability distributions and consequently the tracker is updated towards such high probability distributions.

Our algorithm worked especially well by having a histogram size of approximately 10 bins. Thus, we employed a technique where we only considered a single highest mode of the histogram distribution and its neighboring bins to represent the color model of the object. This helped especially when the colored object had other colors excluding its base color.

Tracking in the color histogram realm also helped deal with motion blur issues that is usually difficult to solve for when dealing with feature detection. Since the back-projection computed is a gray-scale image of the belief of whether the pixel in the image belongs to histogram of the object, it is quite robust to perspective views and one doesn't have to actively learn new features over multiple viewpoints.

5. 3D Object Tracking Using Iterative Closest Point (ICP)

5.1 ICP Description

A common algorithm for tracking objects of arbitrary shape in point clouds is Iterative Closest Point (ICP). The setup is:

Input: Point-sets $A \subseteq \mathrm{R}^3$. A represents the segmented object to track.
$B \subseteq \mathrm{R}^3$. B represents the latest point cloud from the Kinect.

Output: One-to-one matching function $\mu : A \rightarrow B$ that minimizes the root mean squared distance (RMSD) between A and B. That is, μ indicates

how the object *A* is embedded in the latest point cloud *B* (from the Kinect).

In particular, we are minimizing: $RMSD(A, B, [?][?]) = \sqrt{\frac{1}{n} * \sum_{a \in A} \|a - [?][?](a)\|^2}$

ICP allows for the object *A* to be translated and rotated; the object *A* might have moved since we received the previous point cloud from the Kinect. So letting *R* be a rotation matrix and *t* a translation vector, we want to find:

$$\min_{\mu,\, t,\, R} \sum_{a \in A} \|Ra - t - [?][?](a)\|^2$$

That is: given that we knew the location of *A* in the previous point cloud, we want to search for a possibly rotated-and-translated version of it in the latest point cloud. ICP proceeds as follows [4]:

ICP(A,B): A, B point sets. R = rotation matrix. t = translation vector.
A is the object we want to track
Find μ one-to-one matching function.
1. Initialize R = [the identity matrix], t = 0.
2. Matching Step: Given R and t, compute optimal μ by finding min_{μ} *RMSD(A, B, μ).*
3. Transformation Step: Given μ, compute optimal R and t by finding
$min_{R,t}$ *RMSD(RA − t, B, μ).*
4. Go to step 2 unless μ is unchanged.

ICP runs with a maximum number of iterations (a parameter); and if the error never diminishes below threshold, we declare that ICP has not converged / object tracking fails..

5.2 ICP Results

Using a 2.53 GHz, quad-core laptop with 4GB of RAM, we found that ICP worked well for low frame rate (< 3Hz), small-displacement tracking with a full point cloud. The algorithm typically took 180-220ms to converge. In our final implementation, we downsampled the point cloud by a factor of 4; resulting in ICP runtimes of 30ms, without noticeable impact on tracking performance. We were subsequently able to run at up to 12Hz on the downsampled point cloud.

ICP usually failed when the target was displaced significantly between frames. To handle significant displacements, we incorporated the 2D tracker as described in the following section.

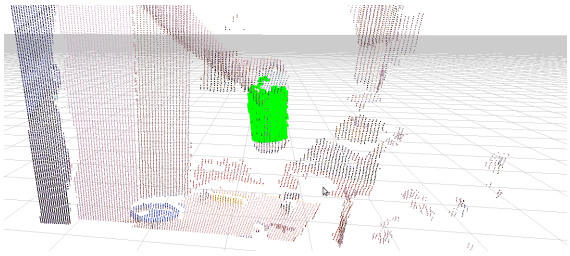

Figure 4: ICP successfully tracking a cylinder in a down-sampled cloud

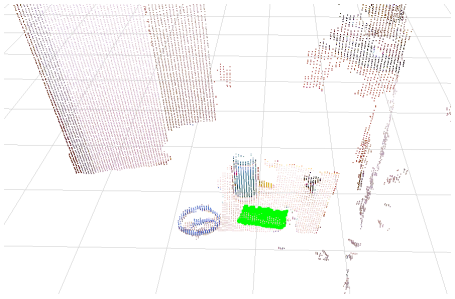

Figure 5: ICP failure after big displacement between frames

6. Combining 2D and 3D Tracking: Using ICP with hints from 2D Tracking

To handle large object-displacements between frames, we used the 2D tracker to seed the 3D tracker.

1. *We convert the 2D ellipse (returned from 2D tracking) into a 3D point cloud B: B is a subset of the full point cloud.*
2. *We run ICP within the sub-cloud B.*

The result is that ICP is seeded with an initial guess: the region *B* corresponding to the ellipse. Since the 2D tracker is unaffected by large, inter-frame displacements, the ICP displacement problem disappears. The 2D/3D hybrid tracker fails whenever the 2D tracker fails: if ICP is not seeded with a sensible guess, it's unlikely to converge. However, if the 2D tracker is able to estimate, even roughly, the object location, ICP works well.

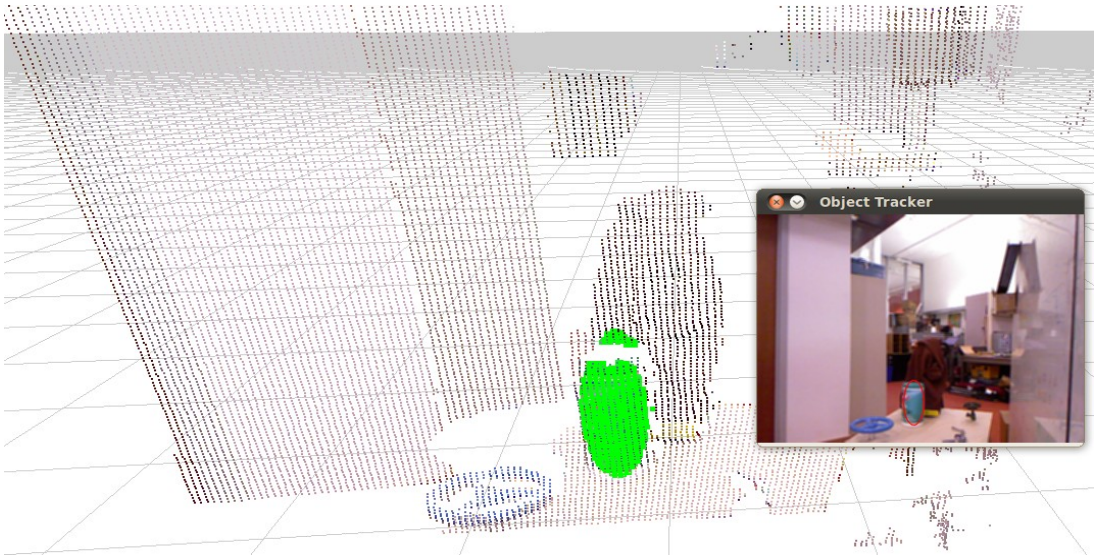

Figure 6:Here we see (a) the 2D ellipse returned by the tracker (right-side, 2D image). (b) The corresponding 3D points (left-side, green). Notice that the ellipse has partially spilled onto the sweater behind the cylinder.

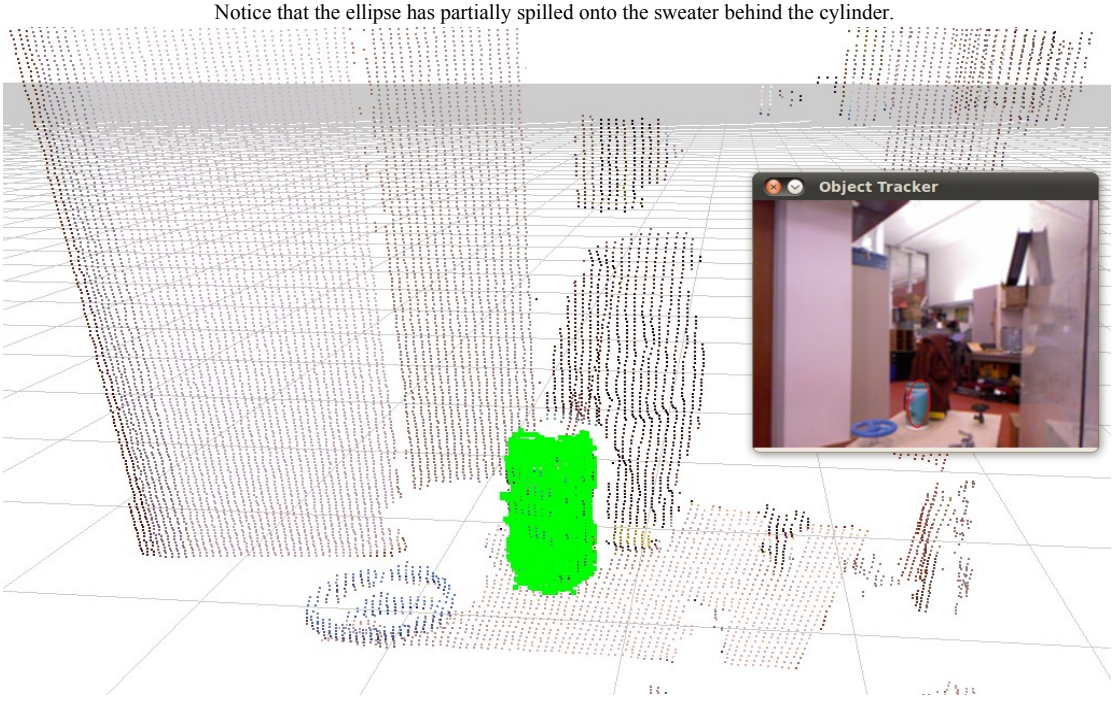

Figure 7: The result of running ICP within the 3D region seeded from the 2D tracker. ICP determines that the sweater is not part of the cylinder.

7. Personal Contributions

The following was my contribution to the final project:

7.1 Feature description/tracking

i) Implemented 2D feature tracking (initial) which employed using FAST features and SIFT/SURF/ORB feature descriptors

ii) Implemented the underlying code to add descriptors to account for multiple viewpoints

iii) Developed and coded up an IDing scheme once the descriptors were matched

7.2 Histogram-based tracking

i) Integrated the CAMSHIFT algorithm and improved upon the algorithm to account for multi-modal histogram matching

ii) Improved upon the mean-shift tracking algorithm to reacquire the object once the track is lost

iii) Added an interface to build color models for the object in a live-feed

iv) Optimized the 2D tracking code and the ICP code for real-time performance

v) Integrated most of the message passing structures between 3D tracking and 2D tracking

7.3 GUI

i) Implemented 3D picking, where one could draw a bounding box around a point cloud and have it return the IDs for each of the points that fall within the bounding box.

8. Summary

In this project, we have shown that it is possible to perform real-time, accurate 3D object tracking using only the Kinect. Replacing the cumbersome need to physically mark the target object in traditional motion capture systems, we introduce a lightweight GUI that allows users to navigate around the 3D scene and quickly segment the point cloud associated with the object. We use both 2D appearance-based tracking and the 3D iterative closest point (ICP) method, combining color and shape information to register the object from frame to frame. Importantly, our pipeline uses the 2D tracker's fast and accurate results to seed the 3D tracker, resolving much of the ambiguity in the ICP step and producing more accurate results. Due to the fact that our system's components communicate over LCM, they can be run on separate machines to increase throughput. Running on a single machine, our system tracks a colorful cylinder's movement between frames in 30 ms, closely matching the frame rate of the Kinect (33 FPS).

9. References

[1] Rosten, Edward. *FAST Corner Detection*. Web. http://www.edwardrosten.com/work/fast.html

[2] Rublee, E. et. al. *ORB: An Efficient Alternative to SIFT or SURF*. Web. www.willowgarage.com/sites/default/files/**orb**_final.pdf

[3] Bradski, Gary. *Computer Vision Face Tracking for Use in a Perceptual User Interface*. Web. opencv.jp/opencv-1.0.0_org/docs/papers/**camshift**.pdf

[4] Phillips, Jeff. "Iterative Closest Point and Earth Mover's Distance." Lecture. *Geometric Optimization*. Web. 7 Dec. 2011. <http://www.cs.duke.edu/courses/spring07/cps296.2/scribe_notes/lecture24.pdf>.