Using Geometry Maps For Camera-Only Robot Localization

Matthew Fisher
mdfisher@cs.stanford.edu

Roy Frostig
froystig@cs.stanford.edu

Max Libbrecht
maxl@cs.stanford.edu

August 14, 2009

Abstract

We present an algorithm for geometric mapping from range scans captured over significant spatial extents and augmented with calibrated images. Because the input range scans and images are taken over a large area in complex environments, significant errors may be introduced during their calibration and we present several techniques to account for this. The output is a textured mesh that can then answer location queries given input images taken in the scanned area. In the query phase, we use a hidden markov model to localize a camera in the scanned environment by matching it against the geometric map. We present the results of our work on a dataset gathered in a very cluttered environment.

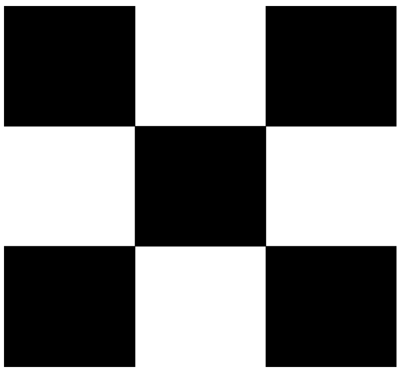
1 Introduction

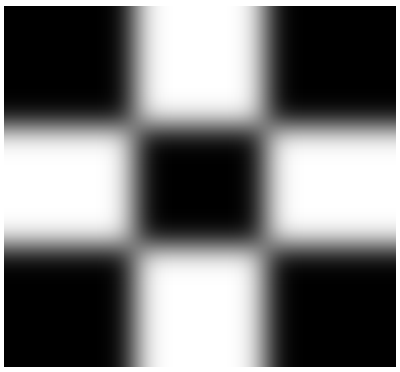
Spatial awareness is critical for autonomous robots to interact effectively with their environments. Although humans are able to perform well given only color information, robots have traditionally needed to rely upon either depth information or some form of global positioning to achieve good spatial awareness. While sensors that can capture this information are readily available, they are considerably more expensive than standard image-based cameras. We demonstrate that, by augmenting a camera-only robot with prior geometric information about its environment, it can accurately determine and track its position. We also show how to obtain this geometric information from a set of range scans and images. This form of tracking has the advantage that it does not suffer from the accumulated error inherent in integrating odometers or accelerometers over time; the location of the robot is determined absolutely each frame. Furthermore, using our technique the image-only agent can determine objects in its view that are inconsistent with its prior view of the world.

Our method consists of two phases: The *Map* phase and the *Query* phase. In the *Map* phase, we build a map of the environment with both geometry and color information. In the *Query* phase, we use the map generated in the previous phase to localize an image. The two phases can be thought of as performed by different robots, with the Map robot being significantly more complex than the Query robot. The Map robot has both range sensors, cameras, and possibly a localization sensor (such as an odometer or GPS.) The Query robot has a set of cameras (generally just one) as its only sensor.

2 Previous Work

Many effective techniques for localization have been introduced using range scanners [Eliazar and Parr, 2003]. These techniques often use a belief-update system using Hidden Markov Models [Ng, 2009]. There has also been success with localizing images with respect to each other and combining them into a shared coordinate frame [Microsoft, 2009, Goesele, 2008]. For simplicity, we will defer a complete overview of existing approaches as we talk about each component.


(a) An original checkerboard pattern.


(b) A reconstruction of the checkerboard where too much averaging has taken place. With small errors in calibration, this results in an image with blurred edges. Sharp edges are very valuable for navigation in the *Query* phase, so blurring them hurts performance.

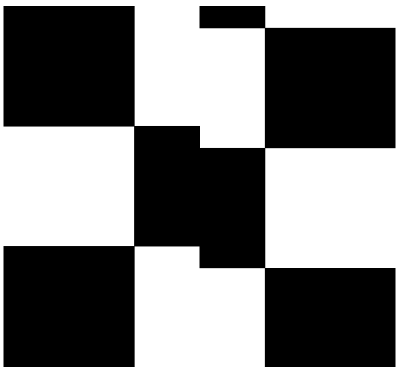

(c) We can do better by taking just a few samples from our video. As you can see, in this case errors create small misalignments, but this turns out to be a much more minor problem. On the other hand, it preserves sharp edges, which is more important.

Figure 1: Image alignment

3 Map

The goal of the *Map* phase is to produce a map of the environment to be used in the *Query* phase. This is performed with data from the Map robot, which takes color and depth data, and can be equipped with some form of positioning, such as an odometer or GPS. Due to the efficient rasterization capabilities of modern graphics cards, we represent the geometry as a textured triangle mesh. The construction of the mesh is done in three phases: the SLAM localization phase, in which we use range data augumented with an odometer determine the location of the Map robot in the global coordinate frame as a function of time; the meshing phase, where we use range data to construct a mesh representing the geometry of the environment; and the texture phase, in which we use color data to add texture to the mesh.

3.1 Sensors

The Map robot is responsible for taking depth and color data about the environment. Therefore it must be equipped with some form of depth sensor, such as a laser range scanner, and one or more image cameras. The Map robot is also greatly aided by some form of positioning, such as an odometer or GPS. We assume all these inputs are calibrated manually beforehand; that is, the extrinsic and intrinsic properties for each camera and scanner relative to the robot's position and orientation are both known.

3.1.1 Video

The *Map* phase requires image data taken of the environment. This could be best performed by a fisheye lens taking a full 360° view from the robot's perspective. We use a slightly less complete set of sensors. We have data from three cameras, pointing up, forward, and to the right of the robot. We texture our geometry using the video from these cameras. We assemble a set of images C_t for each time step t by sampling from the video. We found that the framerate of a normal video turns out to be much higher than is needed to texture the geometry. As well as being redundant, too much image data can actually be detrimental if used incorrectly. A naive way of combining many images is to align them and then average the colors incident on a point together. However, small errors in the alignment will create blurred edges in the resulting image (Figure 1). In Section 3.4.2, we will propose several ways to robustly merge the color information from multiple cameras. We also find it more effective to texture our mesh with a small number of images. While this creates discontinuities in the resulting texture, it preserves sharp edges, which are important in the *Query* phase. Therefore, we sample our video just at a very low frame rate (2 fps.)

Barrel distortion

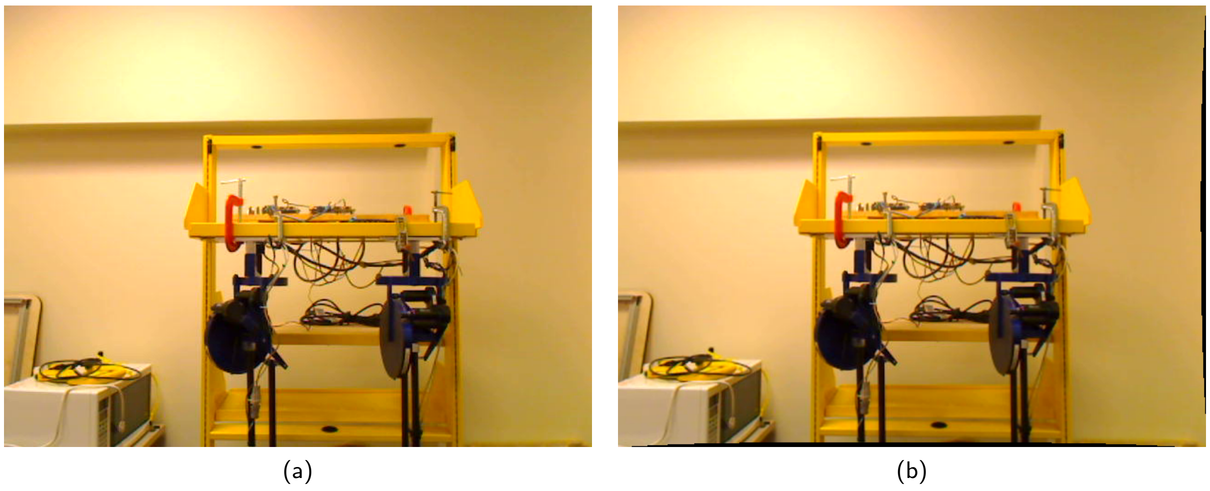


Figure 2: A camera frame before and after barrel distortion correction.

Our video data experiences mild barrel distortion. We use a standard checkerboard calibration pattern [Bouguet, 2008] to compute the coefficients of barrel distortion and other intrinsic camera properties (Figure 2). However, some error remains after this correction. Since barrel distortion is most prominent near the edges of an image, we throw out data that appears within a few pixels of the edge of an video frame (we use 25 pixels).

3.1.2 Depth

Our method requires depth information about all three dimensions of the environment's geometry. Ideally, this would involve a 3D range scanner taking data in a sphere. However, we have had good results taking depth data in a single plane, which moves around the environment as the robot moves.

We have data from a robot with two range scanners (Figure 3). We acquire most of our geometric data from an infrared range scanner mounted on the top of the robot. It sweeps in a vertical plane perpendicular to the direction of the robot (that is, it scans the ceiling above the robot and the walls to its left and right). It only scans about 270° of this plane, centered along the vertical axis. By making some assumptions about the remaining geometry, we complete this to make the full 360° (see below).

The other range scanner is a SICK laser scanner mounted on the front of the robot facing forward. It continually sweeps a laser in a horizontal plane a few feet off the ground. This is used during the localization phase to obtain accurate position information for the vertical scanner and cameras. However, since it only ever sees points in one two dimensional plane, it gives us very limited information for reconstructing the full three-dimensional geometry. We use this scanner solely to determine our position.

Floor completion

Because the vertical range scanner only samples from about 270° of the vertical plane, we need a way of adding the geometry under the robot. Fortunately, we know the robot is rolling on a floor at a distance from the range scanner equal to the height of the robot. We add walls extending down from our first and last point in a scan (that is, those with the smallest and largest θ respectively) and add a floor extending to the base of the walls. We could add the floor when constructing our geometry, but for simplicity we add points to our depth data that correspond to the floor and walls to our 270° data scan, resulting in a full 360°. Obviously, this approach fails

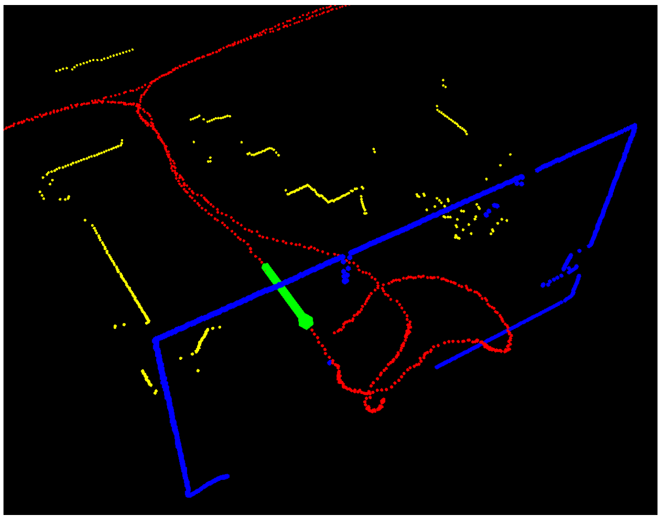


Figure 3: A depiction of all the range data taken by the Map robot at one time-step. The green shape marks the position and orientation of the Map robot. Each yellow dot is a range point taken by the horizontal scanner, and each blue dot is a range point taken by the vertical scanner. The red line is the path taken by the robot.

when the floor isn't flat or the robot is traveling close to an irregularly shaped wall. However, both of these cases are unlikely, particularly in the environments where this robot can operate. If this method were to be applied to a robot running in terrain with irregularly shaped walls and floors, it would be wise to put sensors on the sides and bottom.

Self-scanning
It's not uncommon for the vertical range scanner to hit the robot itself. This is interpreted as a point in the geometry very close to the robot. If not accounted for, this results in a large false discontinuity in the mesh. To account for this, we discard all points within a certain distance from near the beginning or the end of the range scan (i.e. close to 0° or 270°).

Low intensity depth points
The range scanner also reports the "intensity" of each depth data point. This corresponds to the intensity of the signal received by the sensor after bouncing a laser off the environment. Most surfaces reflect the laser at nearly full intensity, so this value is high. However, certain anomalies can cause the laser not to reflect properly and cause the range scanner to receive a low intensity response. Depth points with low intensity are likely to be error-prone, so we discard them and interpolate the point from nearby points. This is very rare.

3.1.3 Odometer

The results of our method are greatly improved with the addition of some form of positioning, such as GPS or an odometer. In our data set, the robot's wheels are equipped with an odometer. We can integrate the odometer readings to calculate our position at any point. However, odometers are prone to accumulating errors over time (Figure 5, Section 3.2), so we only use this information as input to our localization algorithm.

3.1.4 Calibration

Our data set has large errors in calibration. Many of the techniques we develop, such as k-means clustering of color and tolerance of repeated geometry (both described later), are aimed at accounting for these errors. It is possible that many of these techniques might not be necessary on a data set with better calibration.

3.2 Localization via SLAM

To find our position at time t, we must solve the Simultaneous Localization and Mapping problem (commonly referred to as SLAM). This problem is well-understood, so we do not attempt to develop new methods for it, and instead use a state-of-the-art method [Eliazar and Parr, 2003]. We have a great deal of data that could go towards solving our specific instance of this problem (color data, vertical range scanners), but SLAM is traditionally performed with only horizontal range scanner and odometer data, so we do not use the rest of our data in this step.

This gives us a very accurate estimate of $X_t = (x, y, \theta)$, the robot's position at time t (Figures 4 and 5). Note that since our sensors are calibrated, all data can now be expressed in the global coordinate system: depth data are geometry points on objects in the environment, and we know the position and perspective of every image frame.

While these estimates are very good, they are not perfect. Specifically, even though orientation errors are small, they still result in significant error during the color reprojection phase.

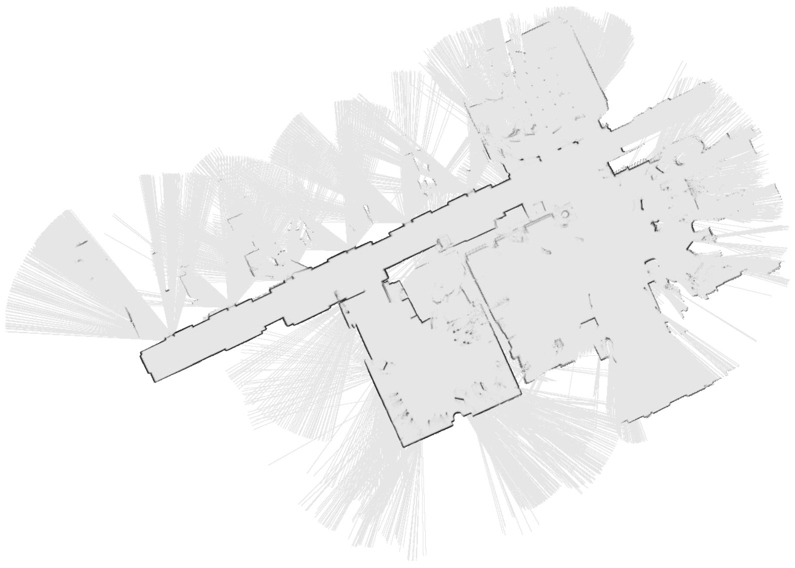


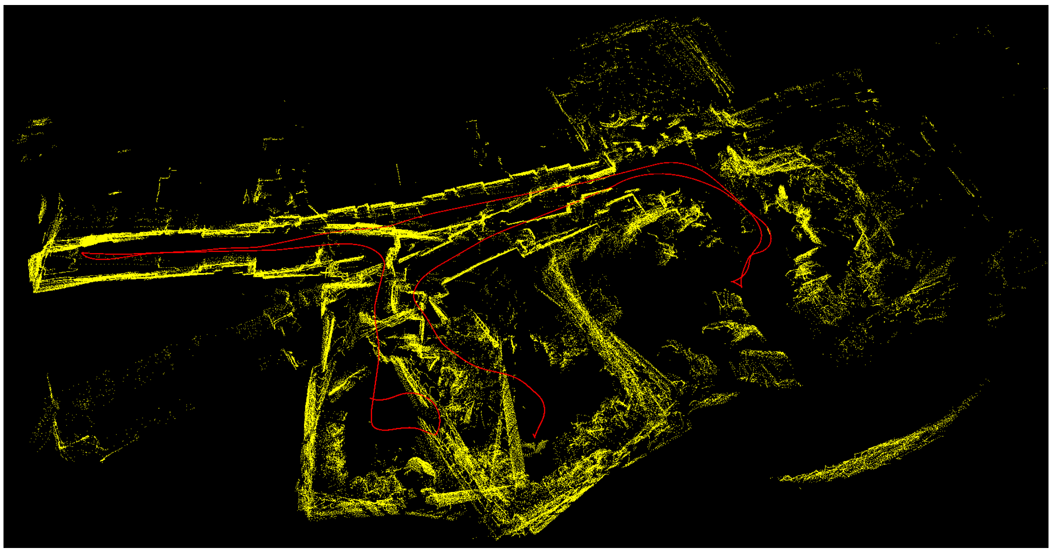
Figure 4: The map of the environment generated by SLAM.

3.3 Geometric mesh construction

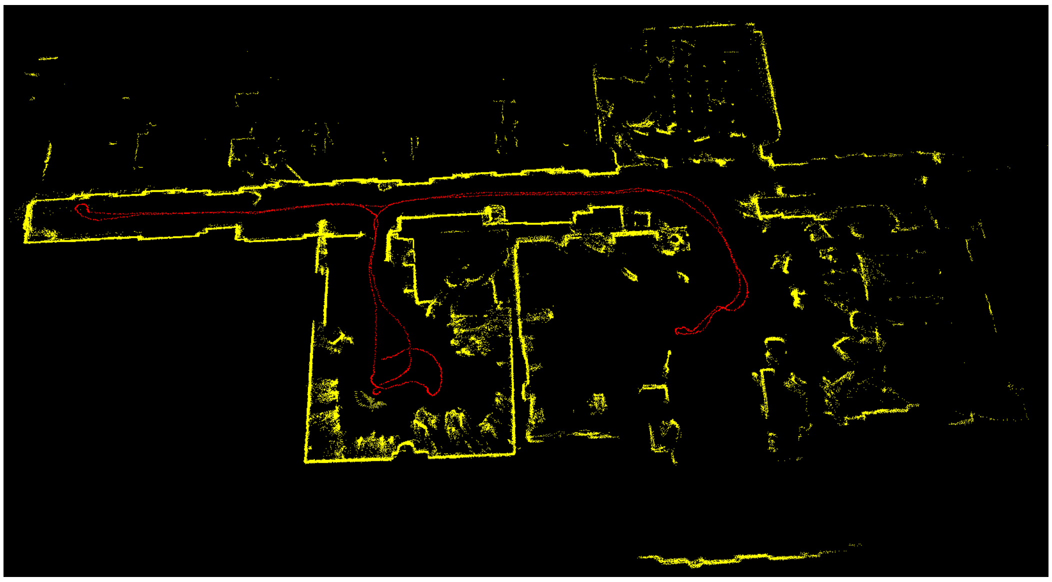
Once we have computed X_t using SLAM, our range data can be interpreted as a cloud of points in the global coordinate frame. Our task in this stage is to produce a triangle mesh of the surface of the environment (Figures 6 and **??**). The primary advantage of a textured triangle mesh is that rasterization of a single frame is extremely fast which we can exploit for real-time performance in the query phase. While it might be advantageous to merge geometry representing the same object in the environment, we find that this is too error-prone, so we leave duplicate geometry in our mesh and develop methods to account for the existence of this duplicate geometry (see below).

3.3.1 Constructing the mesh

To construct the geometry mesh, we take every range point from the vertical scanner to be a vertex. We add an edge between a vertex's neighbors in a single scan (at θ_{i-1} and θ_{i+1}), and between the vertices at the same value



(a) The point cloud created by the horizontal scanner where localization is computed by integrating the odometer, as seen in Section 3.1.2. Each yellow dot is a range data point taken by the horizontal scanner, The red line is the path of the Map robot.



(b) The same point cloud after computing localization with SLAM.

Figure 5: Map created by the horizontal scanner, before and after computing localization with SLAM.

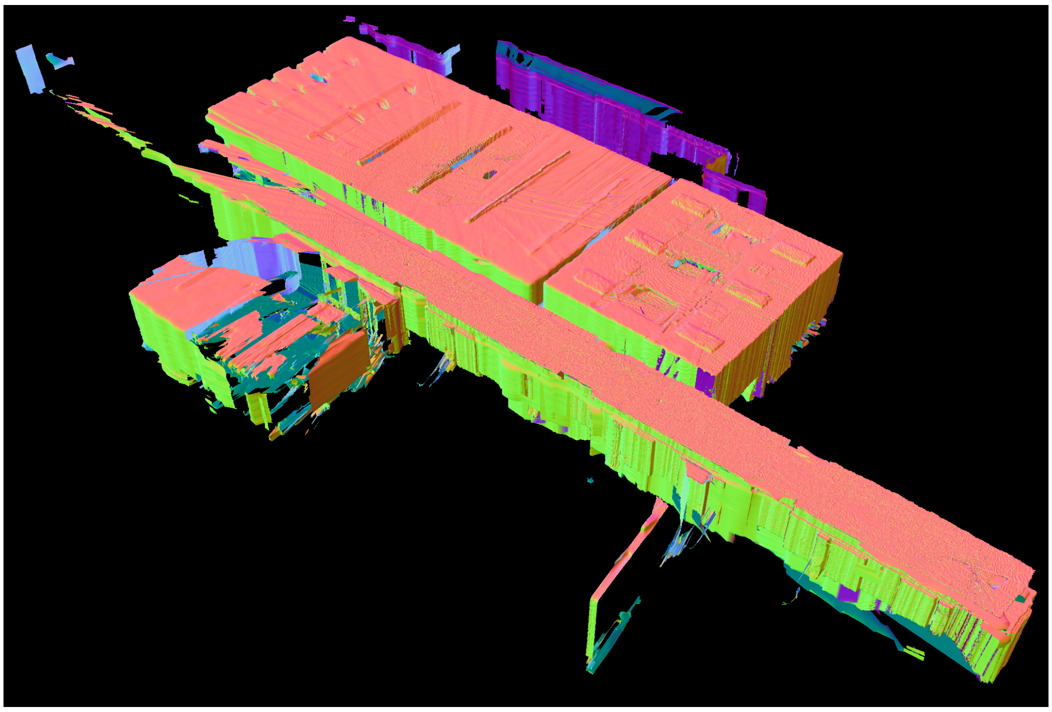


Figure 6: The product of the mesh construction phase: a 3D geometric map of the environment.

of θ in the previous and next scans ($t-1$ and $t+1$) (Figure 8) [Sun et al., 2002]. This gives us a quad mesh (that is, a mesh in which every face is a quadrangle). We add an edge across each face to create a triangle mesh (we choose between the two possible edges arbitrarily) (Figure 9). To reduce noise, we also apply a smoothing step [Desbrun et al., 1999] (Figure 10).

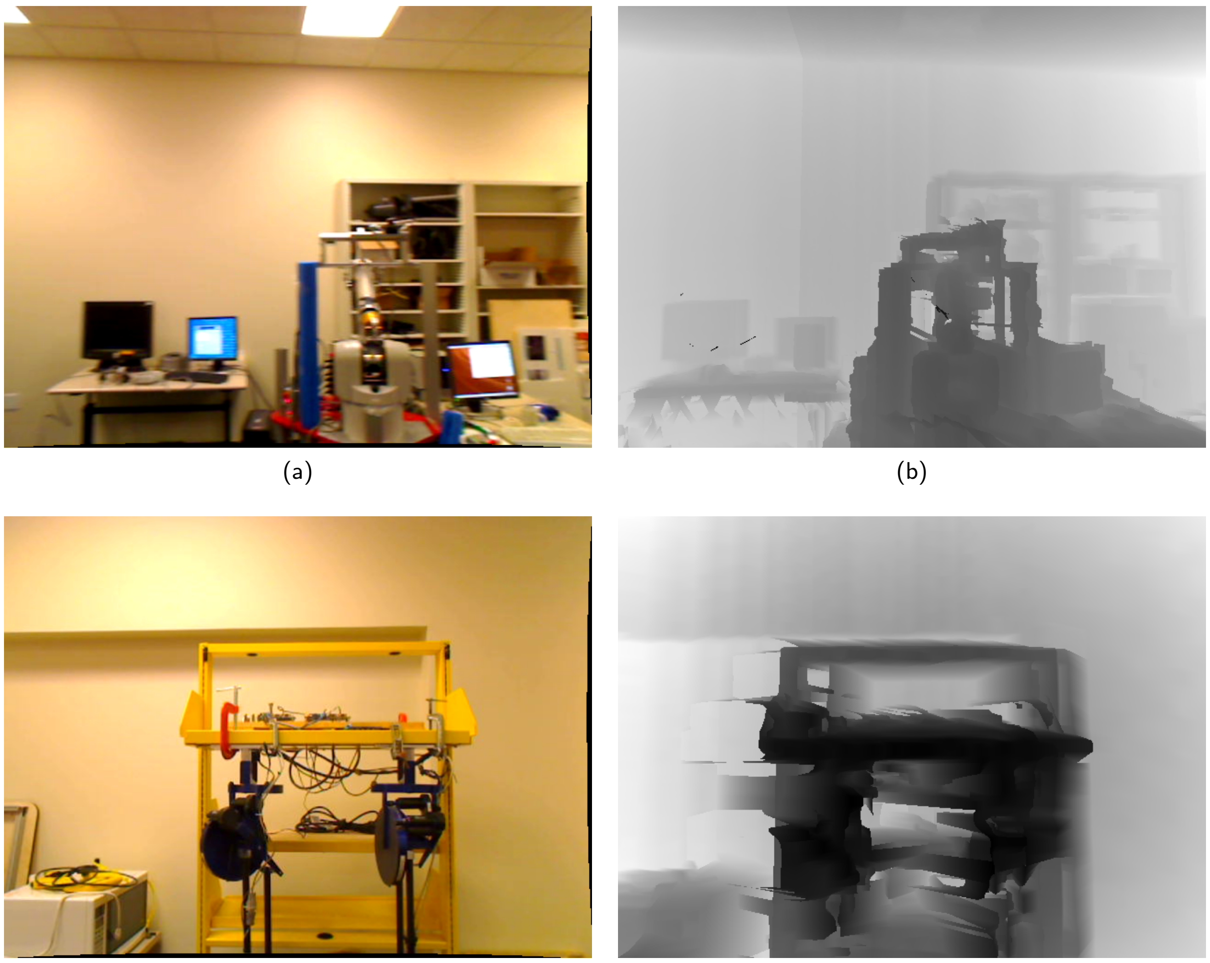
The resolution of the resulting mesh depends on the frequency of sweeps of the laser scanner and the speed the robot is moving and turning. When the robot is moving slowly in a straight line, we get a very dense mesh. When the robot is turning while scanning geometry far away, we get a very sparse mesh.

The way we build the mesh requires the range data to have a very strict structure—every point must have a predecessor and a successor in two dimensions. We could not apply this method on a data set without this structure, such as an unordered cloud of points. Our data set has this structure, but other techniques might need to be developed to apply our method to another data set.

We do not use data from the horizontal range scanner in this stage. While we could use this data, it is likely to have calibration errors with respect to the vertical range scanner, and it only scans points in a single plane, so it does not contribute very much useful information.

3.3.2 Area weighted normal

It is important in the *Query* phase that the meshes are labelled with the side that points into free space and the side that points into the obstacle. We know which side is which by the position of the range scanner when the point was taken. This information taken together with its neighbors gives us the directionality of the mesh. We use the area weighted normal method to compute the direction, which defines the normal at a vertex to be the area-weighted average of the normals of all the faces incident to the vertex.



(a) (b) (c) (d)

Figure 7: Examples of images an their corresponding depth maps (assembled from depth data). As can be seen from (d), objects with complex geometry can be hard to reconstruct, so a map is very approximate. This is a source of error in the *Query* phase.

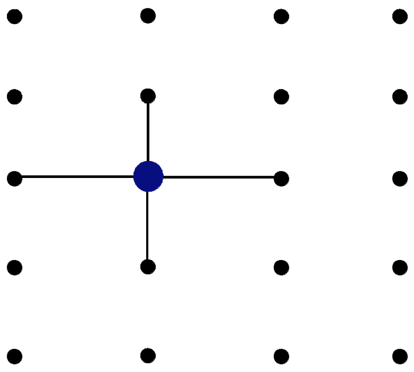


Figure 8: Each black dot represents a range data point, which we will use as vertices in our geometry mesh. Each column of dots corresponds to sweeps of the laser range scanner, at $t-1$, t and $t+1$ respectively. We connect each point to its precursor and successor in both time and θ (the angle of the laser scanner).

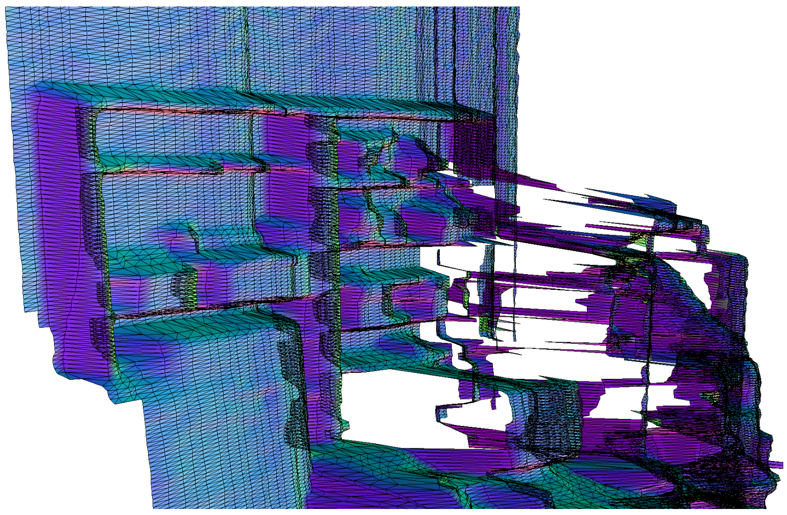
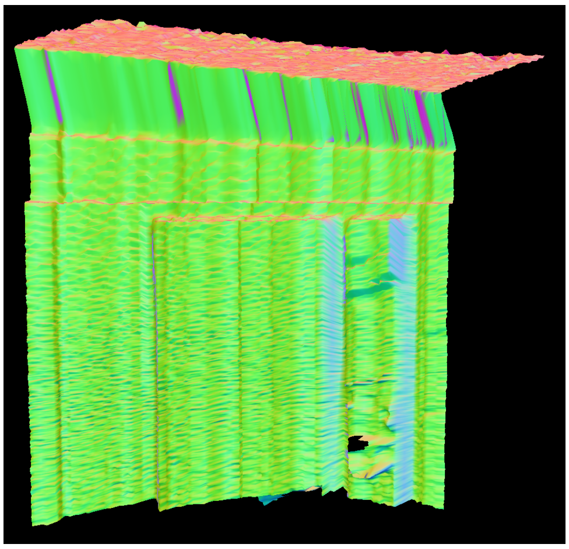


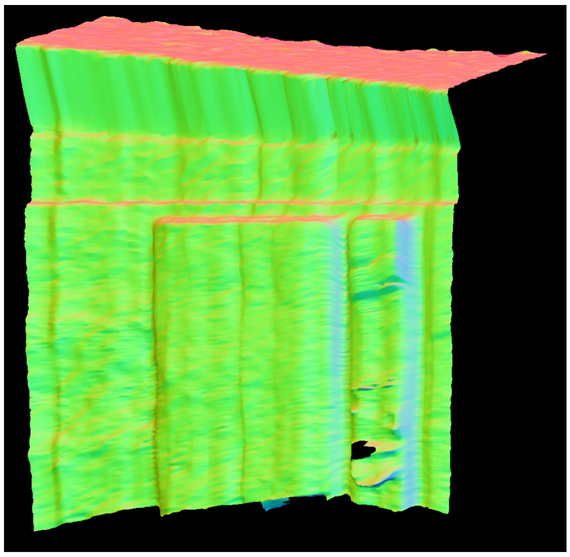
Figure 9: A mesh of a bookshelf with lines showing the edges between vertices explicitly. Each intersection is a vertex created from a range data point.

3.3.3 Ribbon of geometry

In the above method we connect nearby depth points, but if an object is scanned twice, two copies of its geometry will exist in the mesh. Optimally, we would find a correspondence between all points that represent the same geometry and merge them. Our original plan was to use Poisson Surface Reconstruction [Kazhdan et al., 2006] to build the geometry this way [Brown and Rusinkiewicz, 2004]. This would produce a contiguous mesh representing the explored geometry. However, due to extensive calibration errors in our data set, we found this approach to be too error-prone. Corresponding geometry often failed to align, which produced large errors in the *Query* phase (Figures 11 and 12). We decided to abandon this approach. On another data set, Poisson Surface Reconstruction might be possible, and many of the techniques below may not be necessary.

We found it more effective to simply leave the repeated geometry in the mesh and develop techniques to account for it, including adding textures to every copy. These techniques are described in detail below, but in general


(a) An unsmoothed mesh of a door. Our range data is very noisy. This can cause our directionality estimation (normal calculation) to vary wildly because we compute normals in the 1-ring neighborhood


(b) To address this, we use a standard geometry processing technique (cite implicit fairing paper) which uses implicit mean curvature flow to smooth the position function at mesh vertices. This method has a single parameter λ which controls the degree of smoothing. This image uses $\lambda = 3 \times 10^{-4}$.

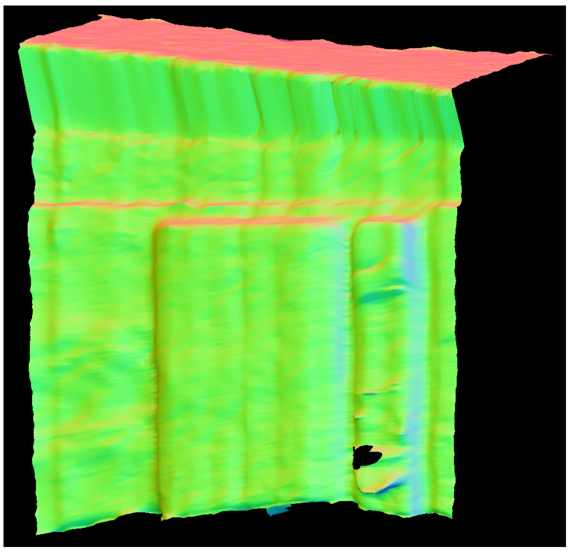

(c) The same mesh after two smoothing iterations. This is visibly too smooth.

Figure 10: Smoothing uses implicit mean curvature flow.

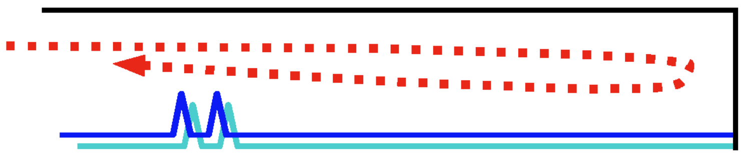


Figure 11: A misalignment error. The robot passed the same geometry twice, creating two copies in the mesh. Due to misalignment errors, we found it difficult to merge copies of the same geometry.

when asking questions about the visibility of a mesh, we report a mesh visible if it is within a margin of the first mesh encountered by a ray trace. In other words, if there is a duplicate copy of an object directly behind the first (slightly misaligned due to noise), we will report the second copy as visible as well.

Observe that since we only ever connect depth points scanned in adjacent scans, the closeness of two points in the mesh isn't related to the physical distance between the two points (or even the surface distance). Instead, we form a long "ribbon" of geometry in which points scanned nearby in time are closely connected in the mesh. This is acceptable because all mesh operations are based on visibility, and not edge distance.

3.3.4 Removing long edges

When connecting the geometry above, we connected every pair of vertices that are nearby in t or θ. However, discontinuities in the scans (usually caused by the edge of an obstacle) create long edges which are unlikely to represent true geometry. This false geometry can be very confusing to the Query robot, because it often occludes true geometry. Therefore, we remove all edges longer then a certain threshold (Figure 13).

This is an imperfect solution, but unless the Map robot explores the discontinuity, we will likely never know its true geometry. Therefore our primary motivation is to avoid errors in later stages of the algorithm, and less to provide useful information to the Query robot.

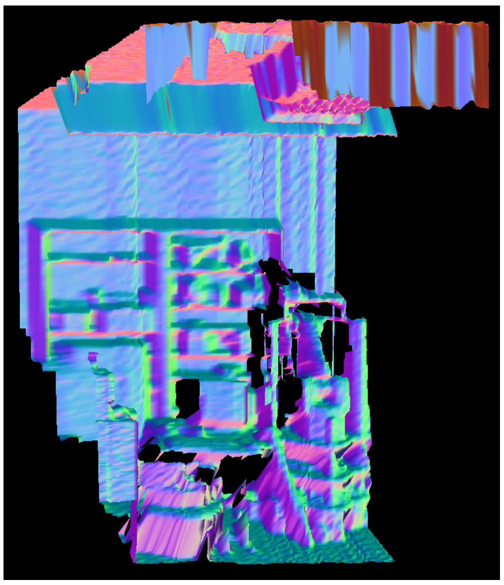
3.4 Texturing the mesh

Given a geometry mesh, the next step is to use color data to add a texture by assigning a color texture to each mesh face [MSDN, 2009]. Each pixel in a face that recieve a color sample is referred to as a textel ("texture pixel"). To help account for errors, we actually assign more than one color to each textel (see below). See Figure 14 for some examples of textured meshes.

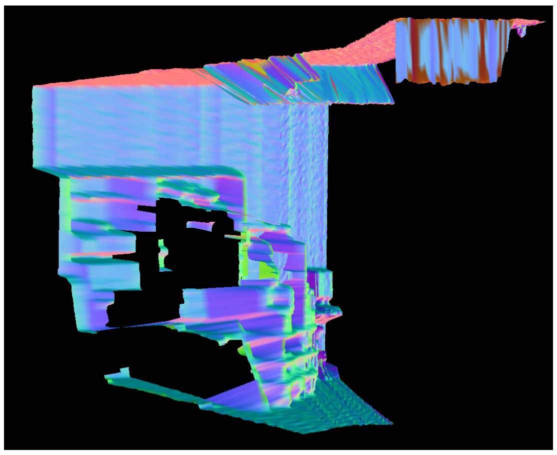
3.4.1 Collecting color data

To color a textel t, we use color data from every camera t is visible from, subject to the constraints described below. We call this set of color points C^*.

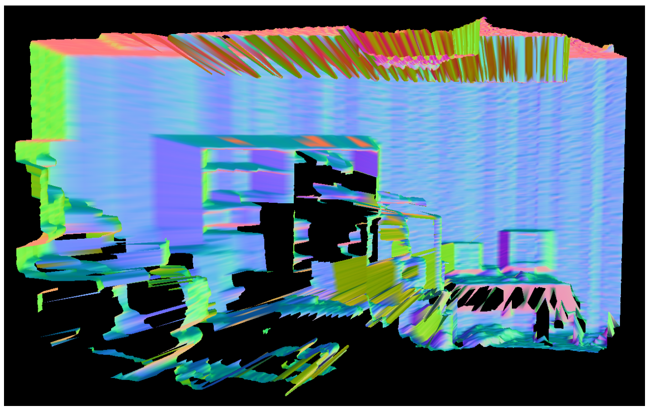
Mesh directionality
As discussed in the Geometric Mesh section, we give the mesh directionality using the area weighted normal. If a camera is facing the same direction as the mesh, the camera is looking at the "back" of the geometry and can't



(a)



(b)



(c)

Figure 12: Three meshes corresponding to the same bookcase. We don't attempt to merge homologous geometry, so these all three copies will exist in our final geometric map

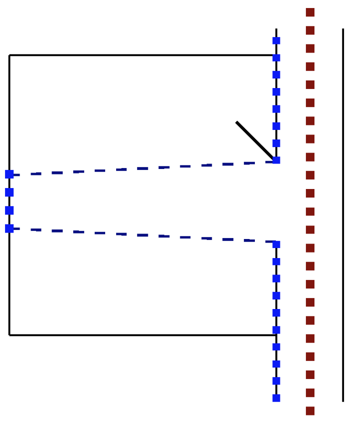


Figure 13: A top-down floorplan view of the Map robot passing an open door as it travels down a hall. The dotted red line is the robot's path, and the blue dots are the scanner sample data points. As it passes the door, the range scanner's view jumps from the doorframe to the wall on the back side of the room. Long edges in the mesh are strong evidence that they don't represent true geometry, so we remove all edges longer than a threshold. In the above case, the presence of the long edge could be very confusing if we later explored the room, since it would be interpreted as a wall blocking sight through the room.

actually see the face of the geometry (Figure 15 (a)). We can calculate this by taking the sign of the dot product of the direction vectors of the camera and the mesh.

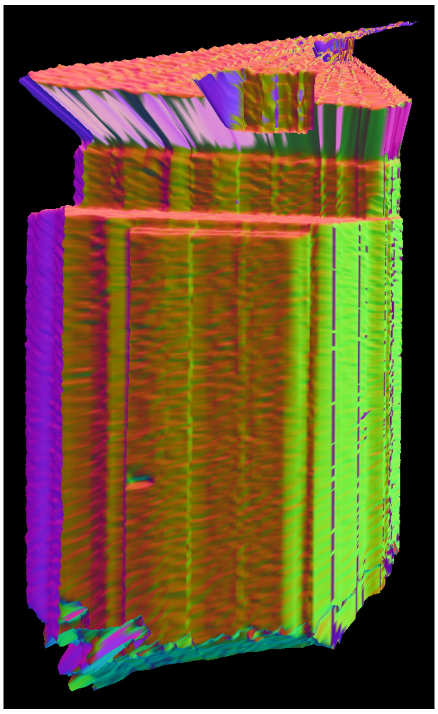
Visibility
We also cull all camera images which have geometry between the camera and the textel. However, because of duplicate geometry, we color all points within a certain margin of the closest visible mesh (Figure 15 (b)). We use this margin to be 10% of the distance between the camera and the textel.

Distance
Since at each time-step we take video in three dimensions and depth data in only a plane, we're likely to see much more geometry than we map. When we have an image that can see a textel but is much farther away (we use 50% farther away than the closest camera) than another images of the same textel, there's a large possibility there's unmapped geometry between the far-away camera and the textel (Figure 16). Since we have other color data, we throw out the farther-away image.

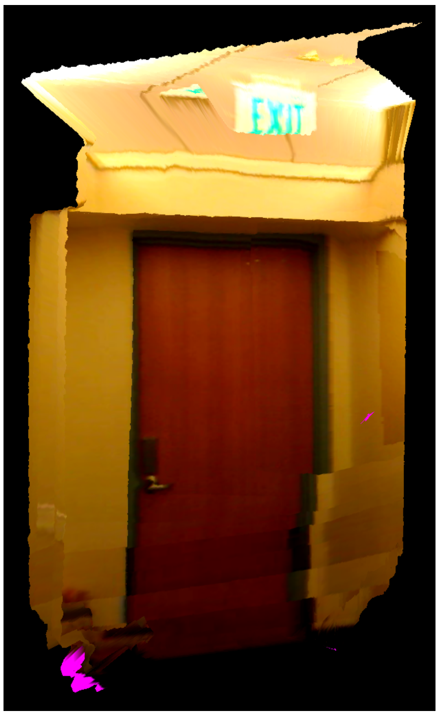
Barrel distortion
We discussed the problem of barrel distortion in Section 3.1. Even after correcting for barrel distortion, the outer edge of each image is more error-prone than the inside. Therefore we don't use an image to color a mesh textel if the textel appears within a few pixels of the edge of the image when projected into the camera's image plane (we use 25 pixels).

3.4.2 k-means clustering

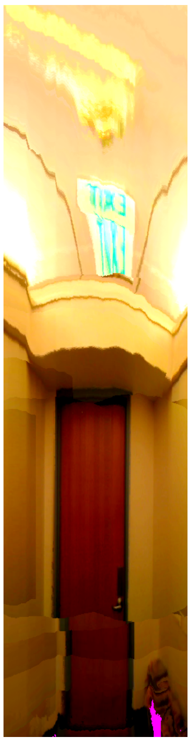
The above process gives us a set of colors C^* which we have observed at the vertex in question. There are many ways of aggregating this data, such as taking the average of observed points. However, because of sources of error discussed below, it's most effective to store more than one color at each point. We use k-means clustering to choose k "mean" colors $\mu_1, ..., \mu_k$ that accurately represent the colors in C^*. See the Appendix, Section 6, for a more detailed treatment of k-means clustering.



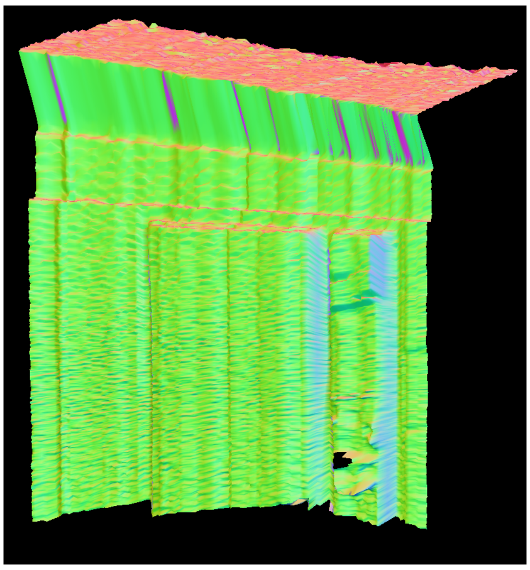
(a) A mesh of a door.



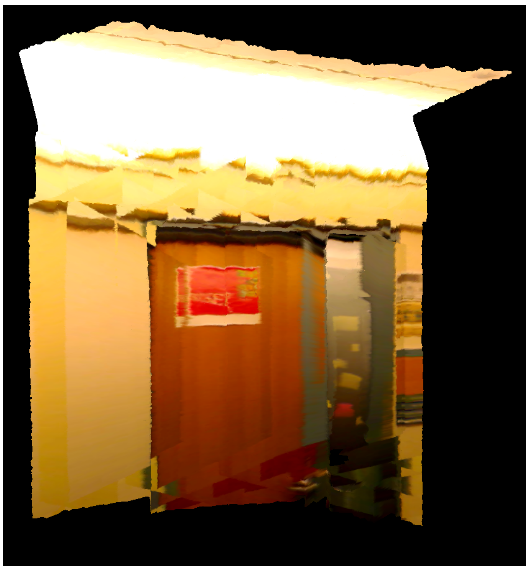
(b) The same mesh with its texture. (We use the middle brightness color cluster. The images composed of other clusters look similar.)



(c) The color domain of the texture. The mesh doesn't actually store the colors itself, just indices into this color domain.

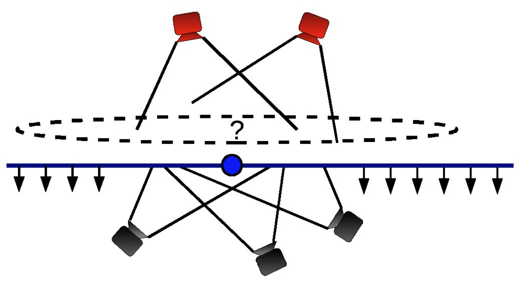


(d) Another mesh.

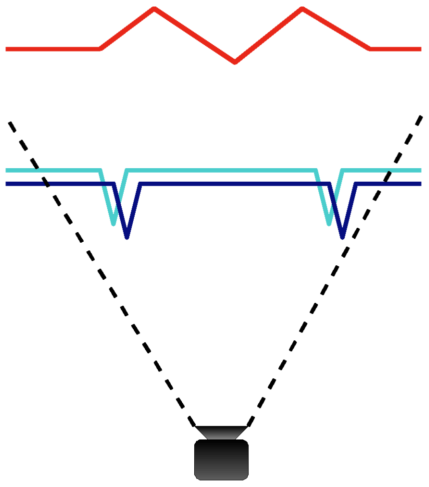


(e) The mesh with texture. The slanting discontinuities are caused by different camera frames as the robot approaches the door from the side.

Figure 14: Mesh texturing



(a) Mesh Directionality: We're trying to color the point on the mesh labeled with a blue dot, which is on a wall. The three black cameras are looking at it from the correct side. The two cameras in red have the point in their field of view, but they're looking at the wall from the wrong side. Whether or not we've sampled the geometry of the other side of the wall, the mesh directionality tells us that we should not consider these cameras in coloring the vertex.



(b) Visibility: We shouldn't use this camera to color the red mesh, since there's other geometry in the way. However, in this case the teal geometry is probably another copy of the blue mesh, so we should use this camera to color it, even though the blue mesh is in the way. We do this by finding the first mesh the a camera ray encounters and coloring all meshes some fixed percent threshold distance beyond the first ray intersection.

Figure 15: Culling images

We order the k clusters in order of brightness (Figure 17). When rendering the mesh in the *Query* phase, we apply anisotropic filtering to achieve a sharper image, but anisotropic filtering works on an entire mesh, not vertex-by-vertex. Therefore we apply anisotropic filtering k times, one for each cluster. Since anisotropic filtering works best on an image with few discontinuities, we attempt to render an image that agrees with itself as much as possible. Ordering by brightness performs acceptably well, and is much better than choosing the ordering arbitrarily.

Like any learning algorithm, there is a trade-off in choosing our value of k. If we have seen more than k separate colors at a point, at least one mean color μ_i is likely to be an outlier (Figure 18 (a)). On the other hand, choosing k to be too large gives us high variance; we're vulnerable to memorizing outliers (Figure 18 (b)). We use $k = 3$ everywhere.

3.5 Sources of error

3.5.1 Non-Lambertian surfaces

A Lambertian surface is one that is the same color when viewed from every angle—the classic example of a non-Lambertian surface is a mirror. Most surfaces have some degree of non-Lambertian property. We use k-means clustering to deal with the problem of non-Lambertian surfaces. We expect that there won't be more than a few colors surrounding a non-Lambertian surface, so each of these will be recorded when we perform k-means clustering. We will never be able to specify a non-Lambertian surface precisely, since to do so we would have to view it from every angle.

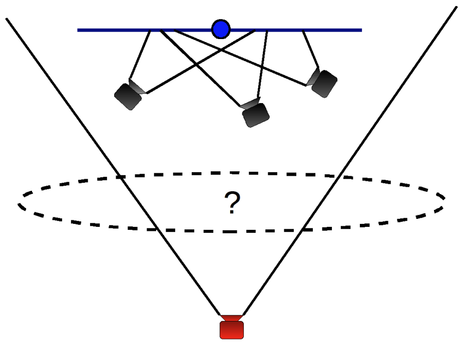


Figure 16: Distance: We're trying to decide whether to use the image from the red camera to color the point on the mesh. Since it is far away from the vertex (relative to the other cameras), it's rather likely that there is geometry in the question mark area that we haven't mapped. Since we have a closer image of the same mesh anyway, we throw out the farther-away image.

3.5.2 Camera calibration

As discussed above, calibration is the source of a great deal of our errors. Our color data and our range data are often misaligned, resulting in assigning the wrong color to our geometry (Figure 19). k-means clustering addresses the problem. We are likely to have the correct color in another image, so we store both colors, and it's unlikely that a point will see more than a few colors as a result of calibration errors. Nevertheless, this texturing misalignment is likely the largest source of our error and would benefit greatly from a depth-map to color image alignment step.

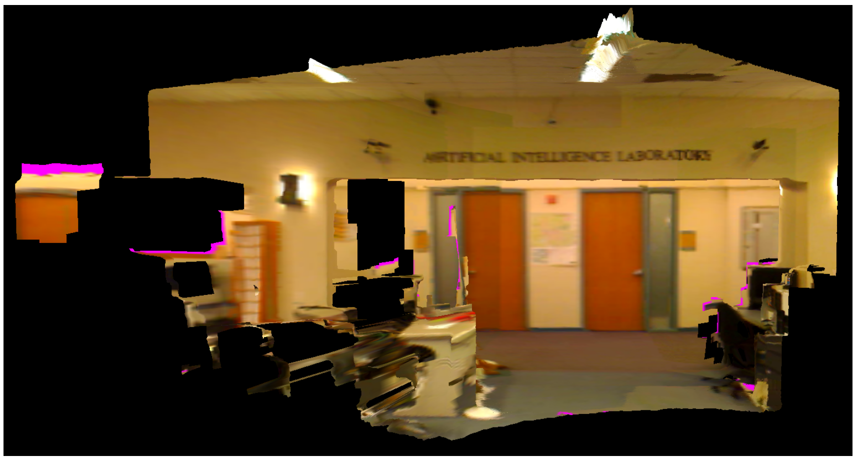
4 Query

Once the *Map* process above is complete, we can use our textured mesh to answer location queries. In general, this means that we can answer the localization question given just an image of the environment. In particular — in terms of autonomous robotics — we can use the mesh and our querying capabilities in conjunction with a Hidden Markov Model (HMM) to allow a camera-only robot can use the system constructed above to self-localize.

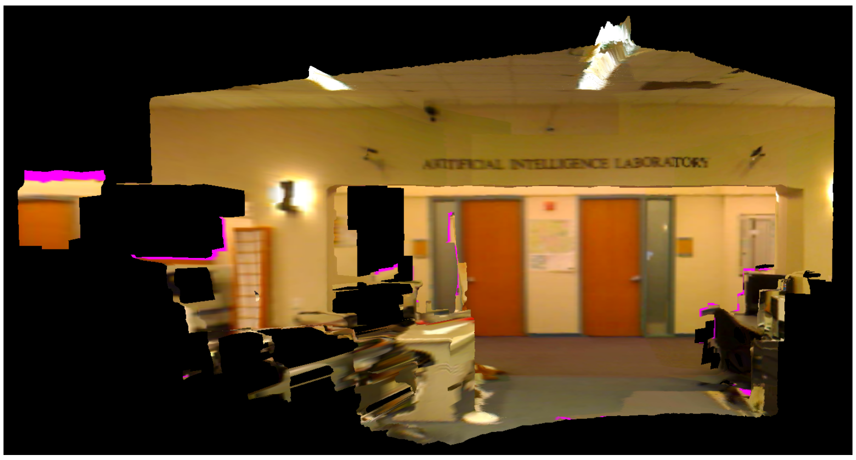
The general querying algorithm takes as input a query image and outputs a probability distribution defined over possible locations, called a “belief map”, which can then be used to infer location. Optionally, the algorithm might also be provided a collection of samples of locations (with repetition) to inspect, which is useful for implementing HMM-based localization for an agent moving within the environment space. Then in the general case, when looking to localize unorganized images, we can think of this optional input collection as being a set of sample points uniformly distributed in the environment space.

4.1 General algorithm

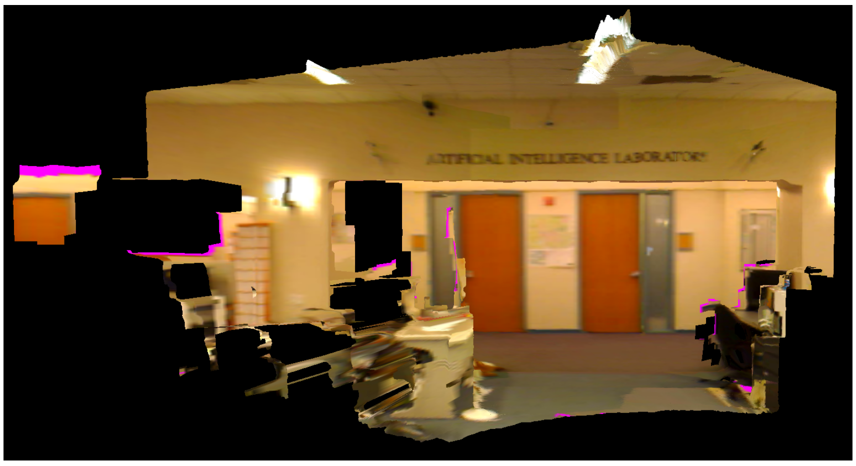
In the most general case, we are given a query image in isolation and wish to find the location $X = (x, y, \theta)$ within the environment from where the image was taken. Informally, the algorithm answers this question by considering a number of locations from which the query image may have been taken. For each such location, it places a virtual camera in the environment model and from there takes a “snapshot”. It then picks the location for which the snapshot was most similar to the query image. We answer the localization query for the single image as follows (the subprocedures Reconstruct-Image-From-Location and Image-Compare are defined further below):



(a)

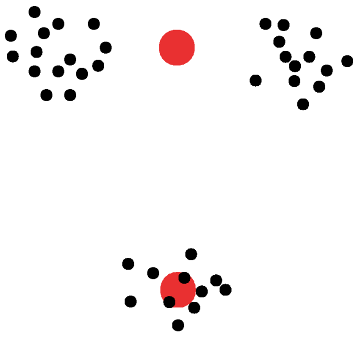


(b)

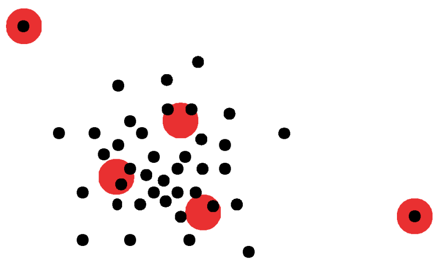


(c)

Figure 17: A room textured with each of the three clusters. We order the clusters by brightness: the top image is textured with the darkest cluster and the bottom image is textured with the brightest cluster.

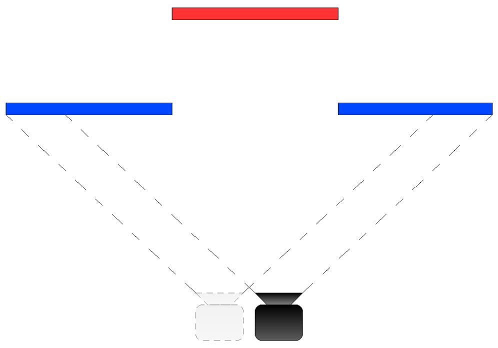


(a) A k-means clustering problem with $k = 2$, with solution points shown in red. This results in the top point, which is a nonsense color between two nearby clusters. The data would be better represented with $k = 3$.

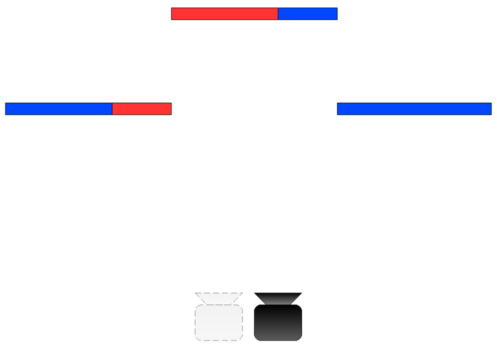


(b) Another k-means clustering problem, this time with $k = 5$. In this case, the data would be better represented with $k = 1$ (the mean). Large k results in the division of the large cluster into three separate colors, each a little bit off from the center of the cluster, which we would guess as the "true" color. Also, it caused us to identify two outliers as their own clusters.

Figure 18: Potential errors in using the k-means clustering algorithm. As with most learning algorithms, there is a trade-off when choosing the number of features to use.



(a) A top-down view of the robot looking at three segments of wall with different colors. The robot believes itself to be at the position in grey, but it's actually in the position in black.



(b) When we reconstruct the color of the wall, a little bit of the color of a wall will "leak" onto the adjacent wall due to imperfect calibration. However, the use of k-means clustering largely corrects this errors. We could also try to align color discontinuities with geometric discontinuities, but we don't take this approach.

Figure 19: An example of an error due to imperfect calibration.

```
LOCATION-QUERY-SINGLE(query image $C$)
1  $B \leftarrow$ a sample set of location points $\{X^{(1)}, ..., X^{(m)}\}$ uniformly distributed within the environment space
2  for $i \leftarrow 1, ..., m$
3        do $C^{(i)} \leftarrow$ RECONSTRUCT-IMAGE-FROM-LOCATION($X^{(i)}, C$)
4  return the location $X^{(i)}$ where $i = \arg\max_j$ IMAGE-COMPARE($C^{(j)}, C$)
```

Notice that here we are interested solely in finding the most likely camera location. Instead of the redundant step of building a belief map by taking samples of locations according to weights proportional to the likelihood of the location, we simply output the location of maximum likelihood. We could also pick a few sample locations $X^{(i)}$ that give a near-maximal IMAGE-COMPARE consistency rating and return the average location among them. This approach is justified by the fact that the distribution has a very high peak near the correct location, as seen in Figure 22. For the same reason, if we have knowledge of the general region in which the true camera location lies, it often suffices to take the sample set B densely within the region rather than in the entire environment space. This allows for denser sampling, and is unlikely to yield a false positive as we do not expect there to have been as strong a local maxima outside of the region as the maximum obtained at samples near the true location. Using LOCATION-SINGLE-QUERY, we obtained a localization error for the query images in Figure 21 as $0.10°$ and 12.5 cm, and $1.13°$ and 6.0 cm, respectively.

Also, given our entire 3-D environment model from the *Map* phase, it is just as possible to consider locations parameterized on all six possible degrees of freedom $(x, y, z, \theta_1, \theta_2, \theta_3)$, but for simplicity we mount our robot such that it has the mentioned three degrees of freedom. Clearly, introducing more possible degrees of freedom increases the expected computation time and the size of the sample space.

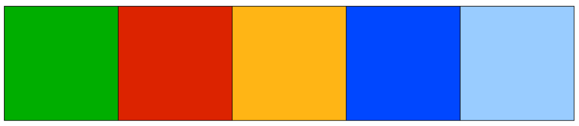
4.1.1 Image reconstruction

The subprocedure RECONSTRUCT-IMAGE-FROM-LOCATION reconstructs an image as close to the query image as possible *from the given location*. Several reconstructions of two query images are shown in Figure 21. To achieve this, RECONSTRUCT-IMAGE-FROM-LOCATION places a virtual camera within the textured environment model at the provided location and uses the textured environment model to select each pixel in the reconstructed image.

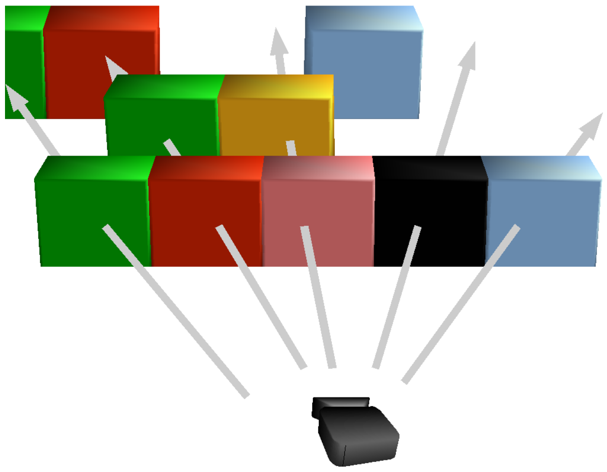
One intuitive approach to pixel selection is as follows. For every pixel in the query image, trace a ray through the environment in the direction of the current pixel until intersecting a face on the mesh. This intersection holds k colors (one for each of the clusters described in Section 3.4). The corresponding pixel location in the reconstructed image is assigned the nearest color across all colors offered by the face of intersection. Here, we choose "nearest color" to mean nearest under the L_1 metric over RGB color vectors.

An equivalent pixel reconstruction can be achieved by splicing out the mesh segment within the environment model that falls within the space visible to the virtual camera. From this mesh segment, we render k proposed images — each image corresponding to a choice of the color cluster index $1, ..., k$ that will be used in coloring all of the image's pixels (cf. Figure 17). We iterate over each such proposed image and update the "best" pixel in our reconstructed image according to the pixels offered by the proposed image. While the ray-tracing-based implementation may be more intuitive, this rasterization approach can leverage the rendering capabilities of a graphics card, and, since it is run inside of the query loop, offers a useful optimization.

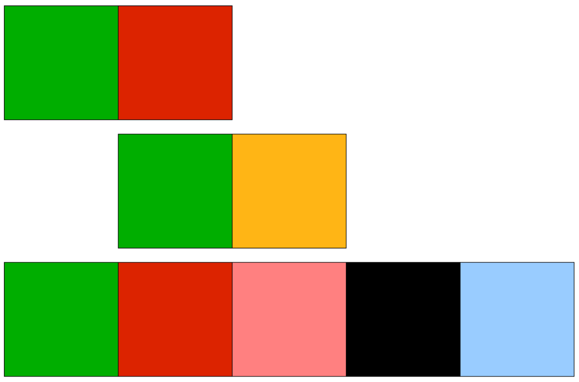
Recall that in creating the environment model, due to calibration error and finite geometric sampling, we may have multiple nearby mesh constructions of the same object corresponding to multiple scans of that object (cf. Section 3.3). For instance, it is possible that the scanner of the *Map* phase went up and down the same corridor of a building while gathering depth data. Thus we may have two nearby "parallel" mesh segments representing the same door, corresponding to the two scans recorded in passing by the door in both directions. In order not to discard the relevant texture information in mesh segments that lie immediately behind the frontmost visible one from the perspective of the virtual camera, we consider visibility up to some percent threshold beyond the first truly "visible" mesh segment. That is, we do not allow mesh segments to obstruct one another in the visibility region of a virtual camera, and collect all the intersections up to some small percent threshold distance beyond that of the first intersection. The image reconstruction algorithm still runs similarly, but now considers a slightly larger set of candidate pixels when selecting the color nearest to the corresponding pixel in the query image. In our implementation, we use a depth threshold of %10 distance beyond the frontmost intersection. An illustration of a simplified version of this final algorithm is shown in Figure 20.



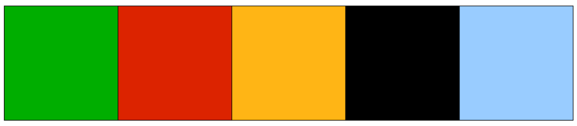
(a) The query image. The reconstruction algorithm will attempt to reconstruct this image from the input location.



(b) A virtual camera is placed in the environment model at the given location. The upright 3-D boxes represent faces of the environment mesh. For each pixel in the query image, we trace a ray from our virtual camera in its direction within the environment model. We gather all colors of all mesh faces intersected by the ray that lie within a depth near the frontmost ray intersection.



(c) The gathered candidate colors. Each column has the colors that we may use in reconstructing the pixel corresponding to that column.



(d) The final reconstructed image. From each column in (c), the color that is nearest that of the query image is chosen for the corresponding pixel in the reconstructed image.

Figure 20: Execution of a simplified version of Reconstruct-Image-From-Location. Here, the height of the world is a single pixel, and the query image is a row of 5 pixels. Also, we use $k = 1$ for illustration purposes. That is, only one color is stored at each mesh face in the environment model. In reality, we usually store $k > 1$ colors for consideration at each face on the mesh.

The reconstructed images in Figure 21 show some pixels colored in magenta. The magenta coloring represents pixels for which there is no "good" color obtained from the intersection of virtual camera ray extending in that pixel's direction within the environment model. This may be caused by one of the following:

- The environment may not have been scanned there, and thus the environment map may not be filled anywhere along the ray (i.e. the ray travels "forever" into a void within the environment model).

- The environment mesh exists and the camera ray does intersect it, but in the mapping and texturing phase there was no original image taken that "saw" this segment of the mesh and hence it is not textured at the intersection point.
- The ray interects a textured part of the map, and the k color clusters have been recorded and considered in Reconstruct-Image-From-Location, but they were all too far from the query image pixel under the color distance metric to be used in the reconstruction, and hence were left as a magenta color for identification.

4.1.2 Image comparison

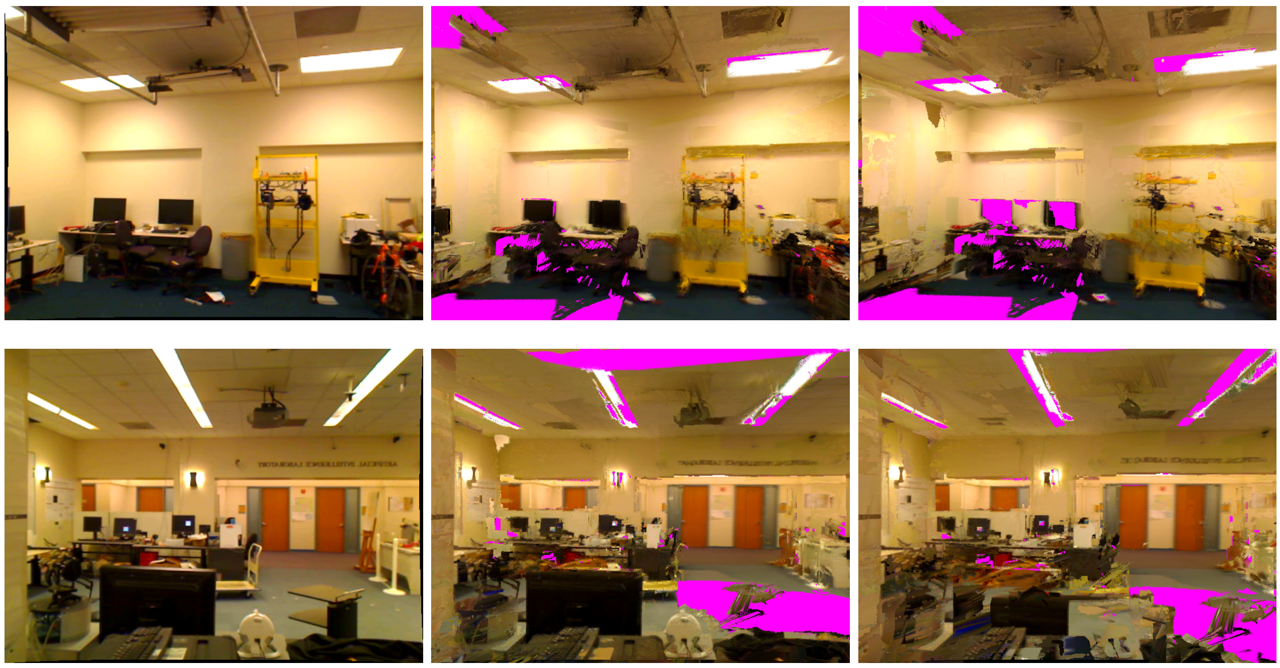


Figure 21: Query images and two of their corresponding reconstructed images from two neighboring locations. Regions colored in magenta correspond to pixels that were not matched, usually because the environment mesh is not textured at the pixel location. **(Left column)** Query images taken by camera in the real world. **(Middle column)** Well-reconstructed images from one sample location. **(Right column)** Reconstructed images from a location slightly offset from those in the middle column. In the top image, the offset is a rotation of 5°. In the bottom image, the offset is a translation of 25 cm. Notice that the images reconstructed from a slight offset location appear far less similar to the query image than those of the middle column.

Once we have a reconstructed image, the Location-Query-Single algorithm calls a subprocedure Image-Compare to compare its reconstructed image to the query image so as to obtain a "goodness" value, which it interprets as the conditional probability of having observed the query image given the image reconstructed from a candidate location [Wang et al., 2005, Rushmeier et al., 1992, Dementhon et al., 2000]. Let $\mathbf{C}_i^{(j)}$ denote the RGB vector of the ith pixel of an image $C^{(j)}$. The image comparison function measures consistency between images by the nearness of corresponding pixels using a capped and normalized L_1 metric on RGB color vectors, given by

$$\textsc{Image-Compare}(C^{(1)}, C^{(2)}) : \sum_i \left(1 - \frac{1}{\delta} \min\left\{\delta,\ \|\mathbf{C}_i^{(1)} - \mathbf{C}_i^{(2)}\|_1\right\}\right)^{\alpha}$$

The term inside the sum of Image-Compare takes on values in $[0, 1]$ and peaks with the value 1 when the two pixels have equal color. The parameter δ has the effect of capping the maximum considered distance between

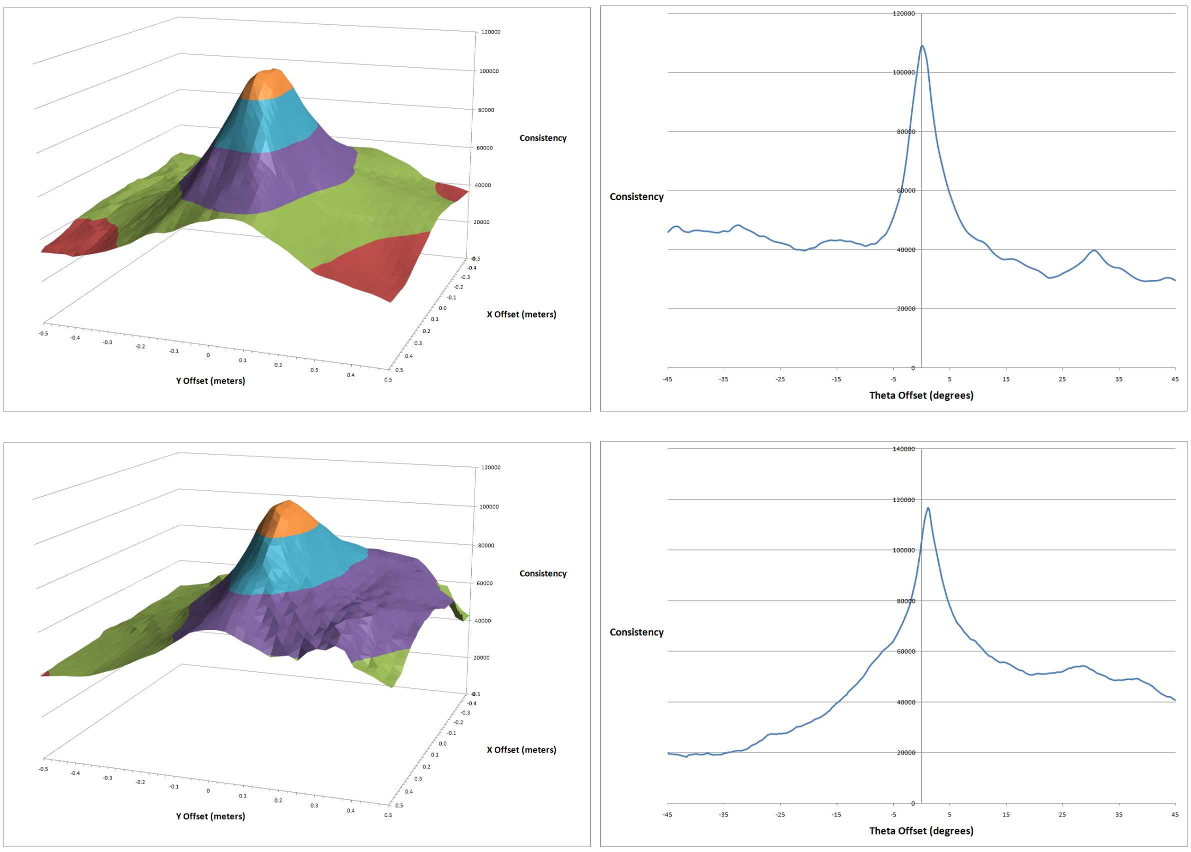


Figure 22: Image-Compare consistency ratings for images reconstructed in a densely sampled region according to the two query images in Figure 21, plotted as a function of: **(left)** varying location parameters x and y, and **(right)** varying location parameter θ. Notice that although the interpreted likelihood of reconstructed images is not necessarily low at far offset locations, the distribution has a very strong peak near the location that gives a qualitatively "good" reconstructed image. Using Location-Query-Single, which outputs the location corresponding to the image for which the graph peaks, the localization error for the top image was $0.10°$ and 12.5 cm, and for the bottom image was $1.13°$ and 6.0 cm.

two colors, so that the consistency of two colors distant beyond some threshold δ is always 0. The parameter α determines the sharpness (or bluntness) of the fall-off curve, as shown in Figure 23.

We have motivation for choosing $\alpha \geq 1$ as follows. We are interpreting the consistency value given by

$$\text{Image-Compare}(C^{(j)}, C)$$

as a conditional probability distribution that ideally peaks when $C^{(j)}$ corresponds to the true location from which C was taken. That is, we would like to have a large distinction between the maximally consistent candidate image $C^{(j)}$ and any other arbitrary candidate image. Hence, having $\alpha \geq 1$ guarantees that having two nearly matching pixels across the compared images contributes a super-linearly greater amount to the image consistency rating than does having two only barely matching pixels. After some experimentation, we found that simply setting $\alpha = 1$ does well to give a sharply peaking consistency between the query image and images reconstructed at locations around the query location. For instance, the consistency plots seen in Figure 22 were observed using $\alpha = 1$.

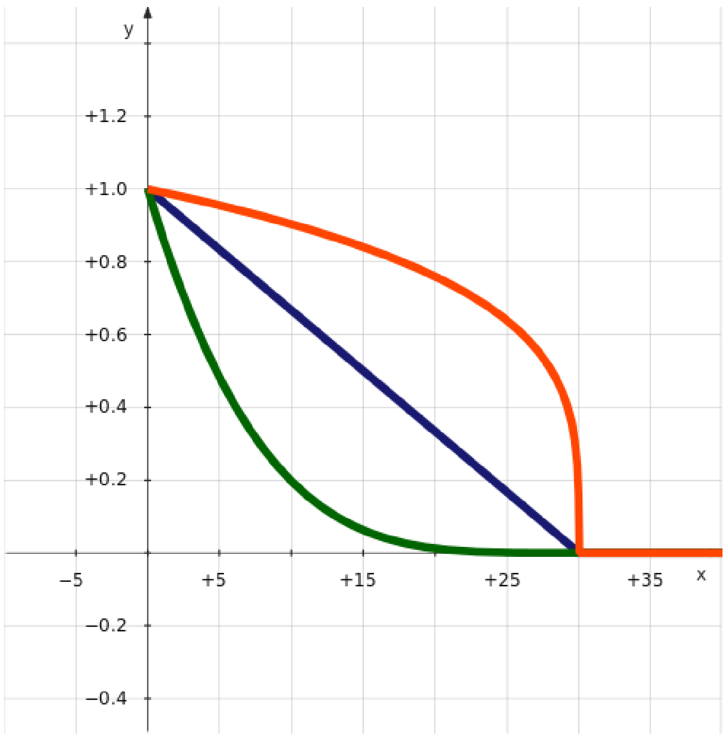


Figure 23: Several plots, for different values of α, of the term within the sum of IMAGE-COMPARE as a function of the distance between the two pixels $\mathbf{C}_i^{(1)}$ and $\mathbf{C}_i^{(2)}$. Here we set $\delta = 30$ for all three curves. **(Green curve)** $\alpha = 4$; **(Blue curve)** $\alpha = 1$; **(Red curve)** $\alpha = \frac{1}{4}$

.

4.2 HMMs for moving-agent localization

4.2.1 The HMM-friendly algorithm

In a common special case, we have data as a stream of images taken over time by an agent travelling through the environment in time that wishes to self-localize. We call such an agent the Query robot. As opposed to the Map robot, which mounts many expensive sensors necessary for the *Map* phase, the Query robot is assumed to be a camera-only robot. Such a robot could self-localize by taking single snapshots and using the LOCATION-QUERY-SINGLE approach to localization, but this discards useful temporal information that could help for more accurate localization.

The single-image query algorithm can be extended to support a Hidden Markov Model for Query robot localization. For this, as motivated in Section 4.1, we use location samples as the discretization method for our state space, and use a particle filtering algorithm to update our approximation of a continuous belief map as follows. (The general method for this is given in the last portion of CS221 Lecture 13 notes.)

```
LOCATION-QUERY-HMM(query image $C$, belief map sample collection $B = \{X^{(1)}, ..., X^{(m)}\}$,action $A$)
 1  for $i \leftarrow 1, ..., m$
 2      do $X'^{(i)} \leftarrow$ SAMPLE-TRANSITION-MODEL($X^{(i)}, A$)
 3         $C' \leftarrow$ RECONSTRUCT-IMAGE-FROM-LOCATION($X^{(i)}, C$)
 4         $w^{(i)} \leftarrow$ IMAGE-COMPARE($C', C$)
 5  $W \leftarrow$ a probability distribution over $\{1, ..., m\}$ obtained from normalizing the weights $w^{(i)}$
 6  $B' \leftarrow \emptyset$
 7  for $1, ..., m$
 8      do $j \leftarrow$ sample a number from $W$
 9         add $X'^{(j)}$ to $B'$
10  return $B'$
    ▷ Note: $B'$ is the new belief map, and will be passed as $B$ in the next query.
```

The parameter B is assumed to be kept as state by the Query robot between calls to the algorithm, and represents the belief map as a collection of samples (with repetition) from the distribution. Assume that in the first time step of the Query robot's journey, B is as in LOCATION-QUERY-SINGLE, a sample set of locations uniformly distributed within the environment space. We also assume that we have some procedure SAMPLE-TRANSITION-MODEL which, given the Query robot's location and an action, returns a state sampled from the its state transition model.

4.2.2 Discoveries

After a few experiments with the HMM Query robot localization, we found that the additional benefits in localization accuracy were small relative to the single-image query method, and did not do well to justify the vast amount of additional computational effort required. This can be explained by observing that belief propagation methods are most useful when absolute localization error is much greater than the state transition model error, in which case we benefit from the propagation of beliefs through the relatively reliable state transition model. Our localization error in the single-image query case is quite small, and our assumptions about the Query robot are such we cannot generally assume its state transition model can compete in accuracy.

In particular, we identified the following in our experiments. Relaxing our assumptions on the simplicity of the Query robot's gear, the state transition model is at best provided by data from an odometer or GPS device. Recall that we assume the Query robot is a camera-only robot, and hence it is reasonable to assert that the lowest state transition model error we can expect from the Query robot is indeed that resulting from the use of such an unattainable device. Let ε denote our approximate absolute (i.e. single-image) localization error under the Euclidean distance metric, and let τ denote the approximate computational runtime of a single query using LOCATION-QUERY-HMM. Given access to odometer data on the Query robot, we found that in order to achieve a state transition error low enough relative to ε that it significantly increases localization accuracy would require sampling the Query robot's video at time intervals close to or even less than τ. Hence, at the rate we are willing to sample the Query robot's video stream, our absolute localization error ε is lower (or about the same) as the state transition model error that we would hypothetically obtain using odometers on the Query robot. Nevertheless, we expect that by having a considerably higher computational time budget we could use LOCATION-QUERY-HMM to achieve far better localization accuracy than with the single-image approach.

In the common expected usecase, where the Query robot is camera-only, the state transition model error is far greater. Examples of common Query robots are a film taken with a camcorder, or an autonomous robot with a webcam. In such cases our best state transition model is a guess of probabilities based on the action taken, or, since the Query robot operates in a continuous workspace, a Gaussian distribution which peaks in the location of the movement action's direction. As such, we found that LOCATION-QUERY-HMM would only be really useful in practice for the subset of robots that are slow-moving and/or equipped with fast processing power.

5 Conclusions and Future Work

We have demonstrated a functional technique for creating a textured geoemtry map in a real environment. Although our data acquistion suffers from severe calibration errors in the form of significant structured and unstructured noise, our method has proven sufficently robust to generate a reasonable environment model. We have quantifiably demonstrated that our model is accurate by using our model to localize arbitrary input images to reasonable accuracy. A big drawback in our method is the computational time necessary to use a full particle-based HMM localization technique on a dense set of input images, especially in the case of a lost tracking which requires a vast number of particles to recover from. This could be greatly improved by using a more intelligent method, such as image-similarity search queries, for finding candidate locations. The largest source of errors in our map construction was the alignment of the images to the range scan. These suffer from an extreme dependence on the exact value of the rotation of the robot, and the accuracy of the localization would benefit greatly from a reasoned approach to aligning the geometry with the camera images to improve the rotation estimation. Furthermore, although the query images can easily detect novel regions, we did not explore the natural consequences of this behavior, such as marking regions for future reconstruction and updating the internal environment model [Furukawa and Ponce, 2007]. Nevertheless, the localization results demonstrated in the tens of centimeters are sufficent for many applications.

6 Appendix: k-means clustering

k-means clustering seeks to solve the clustering problem. That is, given a set of points in $\mathbb{R}^n$, group the points in an intelligent fashion such that nearby points are more likely to be in the same group. We formalize this as follows. Let

$$\text{centroid}(S) = \frac{1}{|S|} \sum_{x \in S} x. \tag{1}$$

We define the k-means clustering objective function on a set of k clusters $C = \{S_1, \ldots, S_k\}$, which we seek to minimize, as

$$J(C) = \sum_{i=1}^{k} \sum_{x_j \in S_i} \|x_j - \text{centroid}(S_i)\| \tag{2}$$

The solution to the k-means clustering problem is defined to be the assignment of points to the sets in C such that $J(C)$ is minimized.

6.1 Lloyd's algorithm

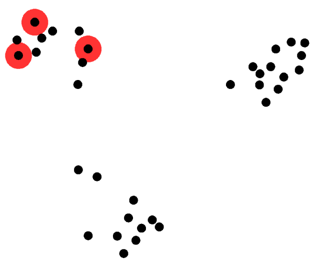
We use Lloyd's algorithm to solve the k-means clustering problem. Lloyd's algorithm works as follows. We define k points $\mu_1 \ldots \mu_k \in \mathbb{R}^n$. These will be the bases for our k clusters. We start by setting each value μ_i arbitrarily. Then we repeat the following until it converges: For each x_j,

$$S_i := S_i \cup \{x_j\}, \text{ where } i = \arg\min_i \|x_j - \mu_i\|$$

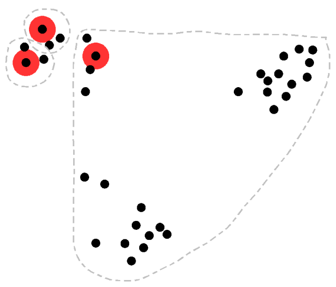
then for each μ_i,

$$\mu_i := \text{centroid}(S_i) = \frac{1}{|S_i|} \sum_{x \in S_i} x.$$

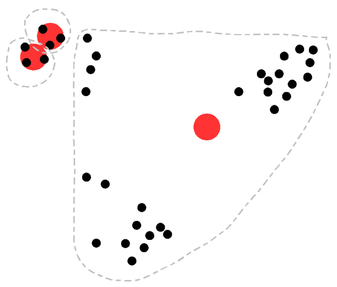
That is, we collect the elements closest to μ_i into S_i, then set μ_i to be the centroid of S_i. See Figure 24 for an example of Lloyd's algorithm being carried out.



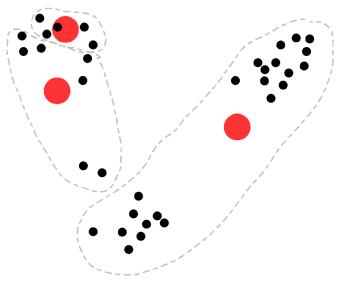
(a) A k-means clustering problem with $k = 3$ in two dimensions. The black points are data observed by the camera. In our method, the data are colors in $\{0, ..., 255\}^3$, but here we're using two dimensions for ease of drawing. The red points are μ values. We initialize each μ value to an arbitrary position—in this case, on top of three data points.



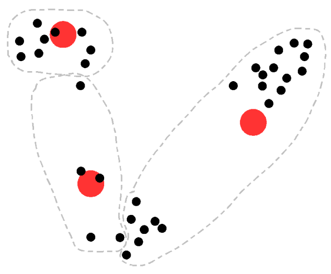
(b) The first step of the first iteration of Lloyd's algorithm. We collect the points into three sets, where each data point goes into the set of the μ value closest to it.



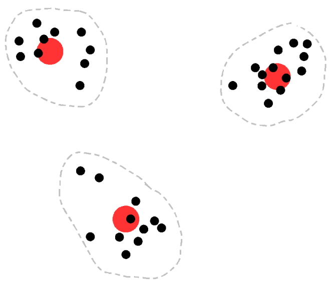
(c) The second step of the first iteration of Lloyd's algorithm. We now assign each μ value to the centroid of its set.



(d) The second iteration of Lloyd's algorithm. We reassign the sets as before, and recalculate the centroids.



(e) The third iteration of Lloyd's algorithm.



(f) The position of the μ values when Lloyd's algorithm converges. This is the solution to the k-means clustering problem.

Figure 24: Running Lloyd's algorithm with $k = 3$ in two dimensions.

References

[Bouguet, 2008] Bouguet, J.-Y. (2008). Camera calibration toolbox for matlab. Retrieved from `http://www.vision.caltech.edu/bouguetj/calib_doc/htmls/parameters.html`.

[Brown and Rusinkiewicz, 2004] Brown, B. and Rusinkiewicz, S. (2004). Non-rigid range-scan alignment using thin-plate splines. In *3DPVT '04: Proceedings of the 3D Data Processing, Visualization, and Transmission, 2nd International Symposium*, pages 759–765, Washington, DC, USA. IEEE Computer Society.

[Dementhon et al., 2000] Dementhon, D., Doermann, D., and Sttickelberg, M. V. (2000). Image distance using hidden markov models. In *in Proceedings 15th International Conference on Pattern Recognition*, pages 143–146.

[Desbrun et al., 1999] Desbrun, M., Meyer, M., Schr oder, P., and Barr, A. (1999). Implicit fairing of irregular meshes using diffusion and curvature flow.

[Eliazar and Parr, 2003] Eliazar, A. and Parr, R. (2003). Dp-slam.

[Furukawa and Ponce, 2007] Furukawa, Y. and Ponce, J. (2007). Accurate, dense, and robust multi-view stereopsis. In *Computer Vision and Pattern Recognition, 2007. CVPR '07. IEEE Conference on*, pages 1–8.

[Goesele, 2008] Goesele, M. (2008). Multi-view stereo for community photo collections.

[Kazhdan et al., 2006] Kazhdan, M., Bolitho, M., and Hoppe, H. (2006). Poisson surface reconstruction. In *SGP '06: Proceedings of the fourth Eurographics symposium on Geometry processing*, pages 61–70, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[Lu and Milios, 1997] Lu, F. and Milios, E. (1997). Globally consistent range scan alignment for environment mapping. *Autonomous Robots*, 4:333–349.

[Microsoft, 2009] Microsoft (2009). Photosynth. Retrieved from `http://livelabs.com/photosynth/`.

[MSDN, 2009] MSDN (2009). Uvatlas. Retrieved from `http://msdn.microsoft.com/en-us/library/bb206321.aspx`.

[Ng, 2009] Ng, A. (2009). Cs221 lecture notes #13: Hidden markov models. Retrieved from `http://www.stanford.edu/class/cs221/notes/cs221-notes13.pdf`.

[Rushmeier et al., 1992] Rushmeier, H., Ward, G., Piatko, C., Sanders, P., and Rust, B. (1992). Comparing real and synthethic images: Some ideas about metrics.

[Sun et al., 2002] Sun, Y., Paik, J. K., Koschan, A., and Abidi, M. A. (2002). 3d reconstruction of indoor and outdoor scenes using a mobile range scanner. In *ICPR '02: Proceedings of the 16 th International Conference on Pattern Recognition (ICPR'02) Volume 3*, page 30653, Washington, DC, USA. IEEE Computer Society.

[Wang et al., 2005] Wang, L., Zhang, Y., and Feng, J. (2005). On the euclidean distance of images. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(8):1334–1339.