Fast Surface Reconstruction and Segmentation with Ground-Based and Airborne LIDAR Range Data

Matthew Carlberg
James Andrews
Peiran Gao
Avideh Zakhor

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2009-5
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-5.html

January 14, 2009

Fast Surface Reconstruction and Segmentation with Ground-Based and Airborne LIDAR Range Data[1]

Matthew Carlberg James Andrews Peiran Gao Avideh Zakhor
University of California, Berkeley; Video and Image Processing Laboratory
{carlberg, jima, p_gao, avz}@eecs.berkeley.edu

Abstract

Recent advances in range measurement devices have opened up new opportunities and challenges for fast 3D modeling of large scale outdoor environments. Applications of such technologies include virtual walk and fly through, urban planning, disaster management, object recognition, training, and simulations. In this paper, we present general methods for surface reconstruction and segmentation of 3D colored point clouds, which are composed of partially ordered ground-based range data registered with airborne data. Our algorithms can be applied to a large class of LIDAR data acquisition systems, where ground-based data is obtained as a series of scan lines. We develop an efficient and scalable algorithm that simultaneously reconstructs surfaces and segments ground-based range data. We also propose a new algorithm for merging ground-based and airborne meshes which exploits the locality of the ground-based mesh. We demonstrate the effectiveness of our results on data sets obtained by two different acquisition systems. We report results on a ground-based point cloud containing 94 million points obtained during a 20 km drive.

1. Introduction

Construction and processing of 3D models of outdoor environments is useful in applications such as urban planning and object recognition. LIDAR scanners provide an attractive source of data for these models by virtue of their dense, accurate sampling. Efficient algorithms exist to register airborne and ground-based LIDAR data and merge the resulting point clouds with color imagery [4].

Significant work in 3D modeling has focused on scanning a stationary object from multiple viewpoints and merging the acquired set of overlapping range images into a single mesh [2,9,10]. However, due to the volume of data involved in large scale urban modeling, data acquisition and processing must be scalable and relatively free of human intervention. Frueh and Zakhor introduce a vehicle-borne system that acquires range data of an urban environment while the acquisition vehicle is in motion under normal traffic conditions [4]. They triangulate a portion of downtown Berkeley using about 8 million range points obtained during a 3-kilometer-drive.

In this paper, we develop a set of scalable algorithms for large scale 3D urban modeling, which can be applied to a relatively general class of LIDAR acquisition systems. We identify a scan line structure common to most ground-based LIDAR systems, and demonstrate how it can be exploited to enable fast algorithms for meshing and segmenting ground-based LIDAR data. We also introduce a method for fusing these ground-based meshes with airborne LIDAR data. We demonstrate our algorithms on two data sets obtained by two different acquisition systems. For surface reconstruction and segmentation, we show results on a point cloud containing 94 million ground-based points obtained during a 20 km drive. We believe that this is the largest urban dataset reported in the literature.

The scan line structure we identify for ground-based LIDAR data can be thought of as a series of adjacent range images that are each a single pixel wide. By making this assumption about point ordering, we can incrementally develop a mesh over a large set of data points in a scalable way. Other surface reconstruction algorithms, such as streaming triangulation, do not identify any explicit structure in their data, but instead take advantage of weak locality in any data [7]. A number of surface reconstruction algorithms triangulate unstructured point clouds [1,3,6]. In particular, Gopi and Krishnan report fast results by preprocessing data into a set of depth pixels for fast neighbor searches [6]. Since they make no assumptions about point ordering, they must alternatively make assumptions about surface smoothness and the distance between points of multiple layers. In contrast, we make assumptions about how the data is

[1] This work is in part supported with funding from the Defense Advanced Research Projects Agency (DARPA) under the Urban Reasoning and Geospatial ExploitatioN Technology (URGENT) Program. This work is being performed under National Geospatial-Intelligence Agency (NGA) Contract Number HM1582-07-C-0018, which is entitled, 'Object Recognition via Brain-Inspired Technology (ORBIT)'. The ideas expressed herein are those of the authors, and are not necessarily endorsed by either DARPA or NGA. This material is approved for public release; distribution is unlimited. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited). This work is also supported in part by the Air Force Office of Scientific Research under Contract Number FA9550-08-1-0168.

obtained, and do not require any preprocessing to "reorder" the point cloud.

Significant work in mesh segmentation has focused on iteratively clustering co-planar faces [5] and grouping triangles that are bound by high curvature [8]. Unlike these works, our segmentation focuses on extracting full objects, which may be composed of many connected parts and shapes, from the ground-based mesh. Our algorithm, which usually provides a coarse under-segmentation, is complementary with existing work on segmentation of 3D objects, and may be combined with more computationally intensive segmentation schemes that use the normalized cut framework [11].

Merging airborne and ground-based meshes can be seen as a special-case version of merging range images, and our method in particular resembles the seaming phase of [9]. However, our algorithm is tailored to the problem of merging ground-based and airborne LIDAR data. We achieve fast, low-memory merges, which favor the higher resolution geometry in the ground-based mesh.

In Section 2, we review our assumptions about data acquisition. We then demonstrate a new meshing technique for terrestrial LIDAR data in Section 3. Section 4 presents an algorithm for segmenting the generated mesh, and Section 5 introduces an algorithm for merging our ground-based mesh with airborne LIDAR data.

2. Data acquisition

Our proposed algorithms accept as input preprocessed point clouds that contain registered ground and airborne data. Each point is specified by an (x,y,z) position in a global coordinate system and an (r,g,b) color value. We do not make any assumptions about the density or point ordering of the airborne LIDAR data. However, we make a number of assumptions on the ground-based data. First, it is ordered as a series of scan lines, as illustrated in Fig. 1, allowing us to incrementally extend a surface across a set of data points in a fast way. Second, there are a variable number of data points per scan line, and the beginning and end of each scan line are not known *a priori*. Since a LIDAR system does not necessarily receive a data return for every pulse that it emits, this assumption keeps our algorithms general and effective, especially when little information is known about the data acquisition system. Finally, the length of each scan line is assumed to be significantly longer than the width between scan lines, in order to help identify neighboring points in adjacent scan lines. These requirements for the terrestrial acquisition system are not particularly constraining, as LIDAR data is often obtained as a series of wide-angled swaths, obtained many times per second [4,12].

We test our algorithms on two data sets, which include both terrestrial and airborne data. In the first data set, S1, terrestrial data is obtained using two vertical 2D laser scanners mounted on a vehicle that acquires data as it moves under normal traffic conditions. An S1 input file lists range points from each scanner separately. The S1 ground-based data is composed of approximately 94 million points over an area of 1.1 km^2, obtained during a 20 km drive. On average, there are 120 samples per scan line for the ground-based data, and roughly 300 points per square meter for the airborne data. The second dataset, S2, uses one 2D laser scanner to obtain ground-based data in a stop-and-go manner. The scanner rotates about the vertical axis and incrementally scans the environment until it has obtained a 360º field of view. The ground-based data in S2 contains a total of about 19 million points with approximately 700 data points per scan line, and the airborne data has roughly 2 points per square meter. Although both S1 and S2 use vertical ground-based scanners, this is not required for our algorithms.

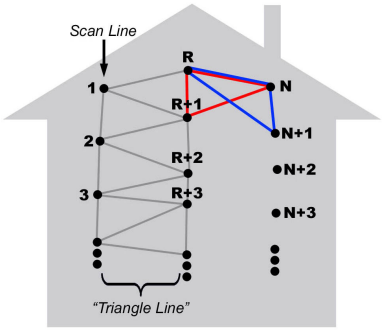

Figure 1. Surface reconstruction illustration.

3. Ground-based surface reconstruction

3.1 Surface Reconstruction Algorithm

In this section, we propose an algorithm for triangulating point clouds that are structured as a series of scan lines. We process data points in the order in which they are obtained by the acquisition system, allowing the algorithm to quickly and incrementally extend a surface over the data in linear time. We only keep a subset of the input point cloud and output mesh in memory at any given time, so our algorithm should scale to arbitrarily large datasets. The algorithm has two basic steps. First, a nearest neighbor search identifies two points likely to be in adjacent scan lines. Second, the algorithm propagates along the two scan lines, extending the triangular mesh until a significant distance discontinuity is detected. At this point, a new nearest neighbor search is performed, and the process continues.

Each nearest neighbor search begins from a point we call the reference point *R,* as illustrated in Fig. 1. *R* is initialized to the first point of an input file, typically corresponding to the first point of the first scan line, and is incremented during triangulation until we reach the end of

the file. We perform a search to find *R*'s nearest neighbor in the next scan line and call this point *N*. The search requires two user-specified parameters—search start and search end—which define the length of each search by specifying where a search begins and ends relative to *R*. The search finds the point within the search space that is closest in distance to *R* and defines it as *N*.

For proper surface reconstruction, we must ensure that *N* is indeed one scan line away from *R*. Because we process points chronologically in the order in which they were obtained, this criterion can easily be enforced if each data point has a timestamp and the frequency of the LIDAR scanner is known. However, in the general case without any timing information, we must choose the search start and search end parameters carefully. We choose the search start parameter as an estimate of the minimum number of data points in a scan line. This ensures that *N* and *R* are not in the same scan line, a situation which can lead to triangles with zero or nearly zero area. We choose the search end parameter as an estimate of the maximum number of points in a scan line to ensure that *N* is not multiple scan lines away from *R*, a situation which can lead to self-intersecting geometry. In practice, we manually analyze the distance between adjacent points in a dataset's point cloud. Semi-periodic distance discontinuities, while not reliable enough to indicate the beginning and end of a scan line, provide a rough estimate for our two search parameters.

Once we have identified an *R-N* pair, triangles are built between the two corresponding scan lines, as shown in Fig. 1. We use *R* and *N* as two vertices of a triangle. The next points chronologically, i.e. *R+1* and *N+1*, provide two candidates for the third vertex and thus two corresponding candidate triangles, as shown in red and blue in Fig. 1. We choose to build the candidate triangle with the smaller diagonal, as long as all sides of the triangle are below a distance threshold, which can be set adaptively as described later. If we build the triangle with vertex *R+1*, we increment reference point; otherwise, if we build the candidate triangle with vertex *N+1*, we increment neighbor point. By doing this, we obtain a new *R-N* pair, and can continue extending the mesh without a new nearest neighbor search. A new search is only performed when we detect a discontinuity based on the distance threshold.

3.2 Adaptive Thresholding

In order to avoid manually hand-tuning a global distance threshold for surface reconstruction, we have implemented a voxel-based method to adaptively choose a set of distance thresholds based on local estimates of point spacing.

The volume that bounds the input point cloud is first divided into *N* uniformly spaced sub-volumes, or voxels. Each LIDAR return is sorted into its corresponding voxel, with the goal of calculating a per-voxel threshold T_i where $i \in 1,2,...N$. To calculate T_i, we iterate through each point *P* in voxel *i*. We find the distance between *P* and the next point chronologically *P+1* so long as *P+1* is in the same voxel. We also find the distance between *P* and *P's* nearest neighbor in the adjacent scan line, using the nearest neighbor search described above. For each voxel, we then separately average the set of distances between chronological points and the set of distances between neighboring points to obtain $\mu_{i,chron}$ and $\mu_{i,neigh}$ respectively. We desire to make right triangles, so we choose our local threshold as

$$T_i = \alpha\sqrt{\mu_{i,chron}^2 + \mu_{i,neigh}^2} \quad (1)$$

Because T_i is a rough estimate of point spacing, we are not particularly constrained by this right triangle assumption, so long as we choose α carefully. In practice, we choose α between 1 and 2, and we put an upper and lower bound on T_i to keep it from getting too large or too small. If a voxel has more than 100 points, we only perform our averaging on the first 100 points to increase computational speed. Similarly, if a voxel has less than 10 points, we assign it a default threshold, because we have little confidence in our point spacing estimates.

Point Cloud	Data Set	# Points	Surface Reconstruct Time w/o Adaptive Threshold (in secs)	# Triangles w/o Adaptive Threshold	Adaptive Threshold Preprocessing Time (in secs)	Surface Reconstruct Time w/ Adapt Threshold (in secs)	# Triangles w/ Adapt Threshold
1	S1	237,567	4	274,295	5	3	281,329
2	S1	2,798,059	49	2,769,888	66	39	2,859,635
3	S1	94,063,689	2422	146,082,639*	-	-	-
4	S2	3,283,343	80	6,004,445	201	54	6,122,694
5	S2	19,370,847	919	32,521,825*	-	-	-

Table 1: Surface reconstruction results. *point clouds 3 and 5 do not use surface removal as text explains.

The local thresholds are dependent on the chosen size of the voxel. If the voxel size is too large, the local threshold does not properly reflect local variations in point spacing. Similarly, if the voxel size is too small, there are not enough points in each voxel to produce a reliable estimate of point spacing. Smaller voxels also increase computational complexity and memory usage. Empirically we choose a voxel size of 1m x 1m x 1m in order to catch local variations in point spacing while still being memory efficient for most point clouds.

3.3 Redundant Surface Removal

Including redundant surfaces in 3D models leads to unpleasant color variations and Z-fighting issues during rendering. In urban modeling, there are two main sources of redundancy—either multiple sensors obtain information about the same area or the same sensor passes over an area multiple times. We have implemented a voxel-based means of redundant surface removal. In order to account for both types of redundancy, we impose the constraint that every triangle within a particular voxel is composed of points from the same sensor obtained at similar times.

To impose this constraint, we keep track of the most recent triangle made in each voxel as we are building the mesh. Every time we attempt to build a new triangle T, we identify the voxels through which T passes and analyze the most recent triangle made in each voxel. If we find that one of these previous triangles has vertices that are from a different sensor than T or has vertices that were acquired a significant amount of time before the vertices of T, we identify T as a redundant triangle and do not include it in the mesh. If the LIDAR data has timestamps, the maximum allowed discontinuity in time is specified in actual units of time; however, in the more general case with no timestamps, we specify it in terms of a maximum allowed point index difference. We choose a voxel size of 1m x 1m x 1m for the same reasons as those discussed in Section 3.2.

3.4 Hole Filling

Non-uniform point spacing can cause holes in a triangular mesh. In urban modeling, non-uniform point spacing is most often caused by occlusions or by the non-uniform motion of an acquisition vehicle. Because adaptive thresholds do not account well for local irregularities in point spacing, we propose a post-processing step for filling small holes in the mesh. Our hole filling method has two steps—hole identification and hole triangulation.

In order to identify holes in the mesh, we first identify triangle edges that are on the borders of the mesh. Identification of these 'boundary edges' is described in significantly more detail in Section 5.2. For the purpose of this discussion, we assume that we have as input to our hole filling step a set of circularly linked lists that describe the edges along the mesh boundaries. An example mesh that has two loops of boundary edges is shown in Fig. 2(a). Our meshes usually have thousands of these circular lists of boundary edges. In this context, we define a hole as a region of the mesh that is enclosed by a small number of boundary edges that can approximately be fit by a plane.

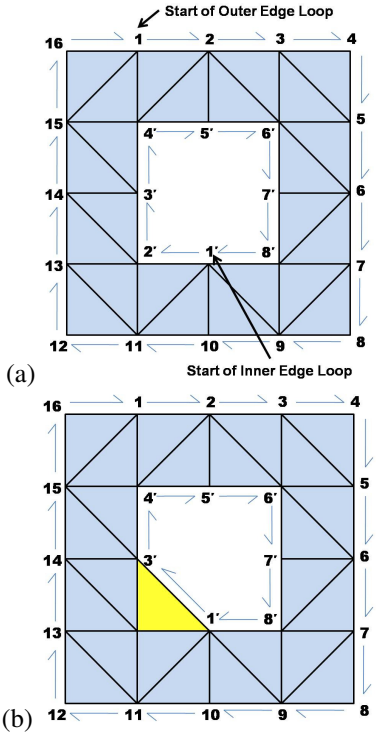

Figure 2. Illustration of (a) circularly linked lists of boundary edges used in (b) incremental hole filling

Once we have identified a hole, we incrementally fill it. Specifically, three adjacent points in the circularly linked list are chosen as three vertices of a candidate triangle. We build the candidate triangle if its minimum angle is above a threshold. Then, we re-link the circularly linked list, as show in in Fig 2(b). If a triangle does not fulfill the minimum angle criteria, we attempt to use three different candidate vertices along the edge of the hole. We continue this process until the hole is filled.

This hole filling method is fast, easy to implement, and reliable on the typical holes observed in urban meshes. However, it has two drawbacks. First, hole identification can fail for the case of small patches of geometry in the mesh. For the example of Fig 2, if we had chosen the maximum number of edge used in hole identification as 16, we would have wrongly identified two holes, one

bound by the outer edge loop and one bound by the inner edge loop. Second, while our triangulation method works for any convex hole, it can fail on certain non-convex holes and create self intersecting geometry.

3.5 Results

We generate models that demonstrate our surface reconstruction algorithm on a 64 bit, 2.66GHz Intel Xeon CPU, with 4 GB of RAM. The results for five different point clouds from S1 and S2 are shown in Table 1. For point clouds 1, 2, and 4, we run our algorithm twice—first using a constant global distance threshold and second setting adaptive local thresholds. For point clouds 3 and 5 we do not implement redundant surface removal or adaptive thresholds, because these memory-intensive, voxel-based schemes are not practical for point clouds that cover such a large area. As shown in Table 1, the complexity of surface reconstruction without adaptive threshold is linear with the number of points in the point cloud. The algorithm can process approximately 100 million points in about 45 minutes.

The times quoted in Table 1 do not include the time required to initially read in the point cloud or to write the final output file to disk. Our point locality assumptions allow us to copy blocks of intermediate data to disk during surface reconstruction in order to avoid running out of memory. For large point clouds, such as point clouds 3 and 5, about 16% of the processing time involves streaming intermediate data to and from disk. Since the pattern of data use is predictable, it should be possible to reduce this cost by using multithreading to prefetch data.

Setting adaptive thresholds is essentially a pre-processing step and takes longer than surface reconstruction because it requires a large number of nearest neighbor searches. However, when we use adaptive thresholds, there is a noticeable speed up in the surface reconstruction algorithm because many of the nearest neighbor searches have already been performed and need not be repeated. We do not currently copy intermediate voxels to disk, making point clouds 3 and 5 prohibitively large for use with adaptive thresholds. However, the strong spatial locality of the data indicates that improved memory management is possible.

Figs. 3(a) and 3(b) show a portion of the triangulated mesh of point cloud 2 from S1 and point cloud 5 from S2 respectively. When using a constant global distance threshold, as is the case in Fig. 3, we use a 0.5m threshold for S1 point clouds and a 0.21 m threshold for S2 point clouds. For S1 point clouds, the search start and search end parameters are chosen as 50 and 200 respectively. For the S2 point cloud, the search start and search end parameters are chosen as 300 and 800 respectively.

(a)

(b)

Figure 3. Surface reconstruction result from (a) point cloud 2 and (b) point cloud 5

Fig. 4(a) is a portion of mesh generated from point cloud 5 using a manually chosen threshold of 0.21 m. The building is a significant distance away from the scanner. As a result, points on the building are naturally more spread out than points closer to the scanner. While our manually tuned threshold fails to capture the building structure and results in gaps on the side of the building, our adaptive thresholding method compensates for the lower point density, as shown in Fig. 4(b). Fig. 5 shows the results of the redundant surface removal algorithm. The data used to generate the mesh in Fig. 5 is not from data sets S1 or S2. Instead, it is generated from a point cloud that is obtained by an acquisition vehicle that drives past the same building multiple times. By applying our redundant surface removal technique, we obtain Fig. 5(b), which has no overlapping triangles and consistent color. We demonstrate hole filling on this same dataset. Fig. 6(a) shows a triangulated mesh before hole filling, and Fig. 6(b) shows the triangulated mesh with hole filling as a post process.

4. Segmentation using terrestrial data

We now present a mesh segmentation algorithm that is an extension of surface reconstruction. The algorithm assumes that an object of interest in 3D is attached to the ground and separate from other objects of interest. Similar to surface reconstruction, we process triangles in

order, so as to only keep a subset of them in memory at a given time. Our segmentation algorithm has two steps: identifying ground triangles during surface reconstruction followed by grouping non-ground, proximate triangles.

The first step in segmentation is to identify ground triangles that are created during surface reconstruction. As a preprocessing step before surface reconstruction, we estimate the height of the ground over a grid in the x-y plane. For each cell, the ground height is estimated as the lowest ground-based LIDAR return in that cell. Next, we perform our surface reconstruction algorithm as described in Section 3. For each triangle built, we project all three vertices onto the grid in the x-y plane. For each vertex, we find the height difference between the vertex and the minimum ground estimate over a 3x3 window of the grid around the vertex. If the height distance for all three vertices is less than a specified ground distance threshold, the triangle is tagged as ground.

Once surface reconstruction is complete, we pass over the list of triangles and perform region growing on non-ground, proximate triangles. The surface reconstruction algorithm has preserved the ordering of triangles as a series of "triangle lines," as illustrated in Fig. 1, where each triangle line corresponds to a pair of scan lines. Therefore, our segmentation algorithm is very similar to our surface reconstruction algorithm described in Section 3. Beginning from a reference triangle, we iterate through a search space to find the triangle in the adjacent triangle line whose centroid is closest to the reference triangle. The algorithm then propagates along the pair of triangle lines performing region growing on pairs of triangles, so long as the distance between their centroids is below a threshold. We only perform a new search when we encounter a distance discontinuity between centroids. Once region growing on the triangles is complete, we render all segments that contain a large number of triangles, as specified by the region size parameter.

We have chosen the distance between triangle centroids as our metric of proximity during segmentation. It is possible to choose other metrics for proximity such as triangles sharing an edge. However, this criterion fails on objects such as trees, which in our meshes are not guaranteed to be composed of sets of connected triangles. Thus, centroid distance provides a simple and relatively general measure of proximity.

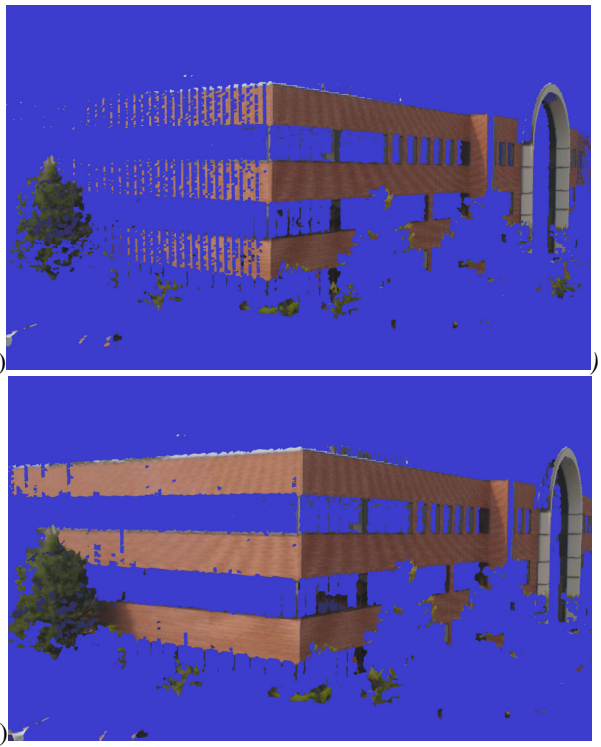

Figure 4. Portion of mesh from point cloud 5 (a) with and (b) without adaptive thresholding.

Figure 5. Mesh (a) with redundant surfaces and (b) without redundant surfaces

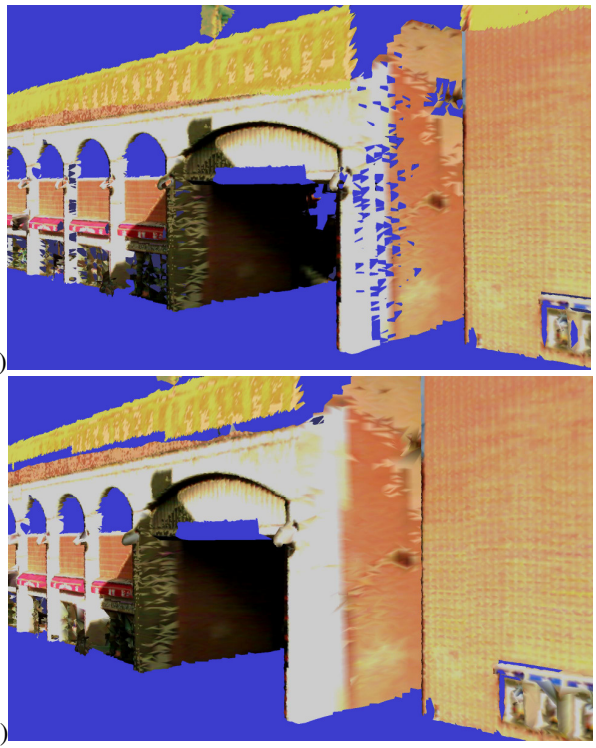

Figure 6. Mesh (a) with holes and (b) with holes filled

We have run segmentation on the same point clouds as the surface reconstruction algorithm, as reported in Table 2. We quote the extra time beyond surface reconstruction required for segmentation. Segmentation times are lower than the surface reconstruction times for the corresponding point clouds. Thus, segmentation can be thought of as a byproduct of surface reconstruction. However, segmentation does not scale quite as well as surface reconstruction because it requires streaming more data, i.e. both points and triangles, to and from disk and because there is extra computation associated with region growing. Fig. 7 shows the mesh of point cloud 1 with ground triangles removed. All 10 segments obtained from Point Cloud 1 are shown in Fig. 8. For the S1 point clouds, we use the same parameters for triangulation as before, and the segmentation parameters are chosen as: 0.5m ground distance threshold, 100 triangle search start, 400 triangle search end, 1 m centroid distance threshold, and 1500 triangle minimum region size. For the S2 point clouds, the triangulation parameters are chosen as before, and the segmentation parameters are chosen as: 0.5 m ground distance threshold, 400 triangle search start, 1600 triangle search end, 0.3 m centroid distance threshold, and 2000 triangle minimum region size.

As shown in Figure 8(a), the house gets segmented together with the white fence and adjoining garage. We argue that this is actually quite an intuitive segmentation, because all of these objects are physically connected. If one were to segment based on some other feature such as color or planarity, this one segment could be split into many different segments, a situation that could be non-ideal in certain applications, such as object recognition. One drawback to our algorithm is that we do not combine segments that correspond to the same object as obtained from different scanners. For example, Fig 8(i) is part of the white fence in Fig. 8(a). It is segmented separately because this portion of the fence originates from a different sensor, than the portion of the fence in Fig. 8(a), due to an occlusion hole.

Fig. 8(f) includes some ground triangles that were not identified correctly as ground, with our relatively simple criterion. If ground triangles are not properly tagged, numerous objects can be wrongly connected together during region growing. Exploring more reliable means of ground identification would make our algorithm significantly more robust.

Figure 7. Mesh of point cloud 1 with ground triangles removed.

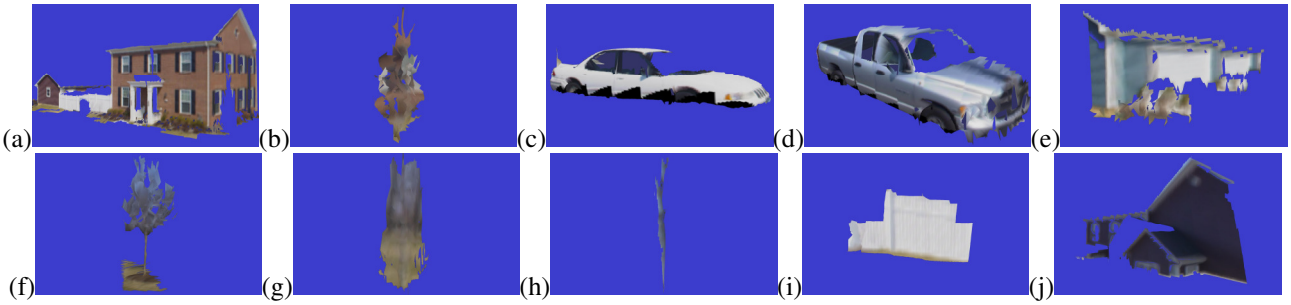

Figure 8. All 10 segments generated from Point Cloud 1.

Point Cloud	# Points	Segmentation Time (in secs)	# Segments
1	237,567	1	10
2	2,798,059	13	130
3	94,063,689	2195	6,197
4	3,283,343	27	56
5	19,370,847	655	463

Table 2: Segmentation results on same point clouds as Table 1

5. Merging airborne and terrestrial data

Data from airborne LIDAR is typically much more sparse and noisy than ground-based data. However, it covers areas which ground-based sensors often do not reach. When airborne data is available, we can use it to fill in what the ground sensors miss, as shown in Fig. 9.

To accomplish this kind of merge, we present an algorithm to (1) create a height field from the airborne LIDAR, (2) triangulate that height field only in regions where no suitable ground data is found, and finally (3) fuse the airborne and ground-based meshes into one complete model by finding and connecting neighboring boundary edges. By exploiting the ordering inherent in our ground-based triangulation method, we perform this merge with only a constant additional memory requirement with respect to the size of the ground-based data, and linear memory growth with respect to the air data. The algorithm is linear in time with respect to the sum of the size of the airborne point cloud and the size of the ground-based mesh. This kind of merge is highly ambiguous near ground level, because of the presence of complex objects, such as cars and street signs. Therefore, we only merge with ground-based mesh boundaries that are significantly higher than ground level, where geometry tends to be simpler and, in the case of roofs, mesh boundaries tend to align with natural color discontinuities.

Figure 9. The dense ground-based mesh is merged with a coarser airborne mesh.

5.1. Creating and triangulating the height field

To create a height field, we use the regular grid structure used by [4] for its simplicity and constant time spatial queries. We transfer our scan data into a regular array of altitude values, choosing the highest altitude available per cell in order to maintain overhanging roofs. We use nearest neighbor interpolation to assign missing cell values and apply a median filter with a window size of 5 to reduce noise.

We wish to create an airborne mesh which does not obscure or intersect features of the higher-resolution ground-based mesh. We therefore mark those portions of the height field that are likely to be "problematic" and regularly tessellate the height field, skipping over the marked portions.

To mark problematic cells, we iterate through all triangles of the ground-based mesh and compare each triangle to the nearby cells of the height field. We use two criteria for deciding which cells to mark: First, when the ground-based triangle is close to the height field, it is likely that the two meshes represent the same surface. Second, when the height of the ground-based triangle is in-between the heights of adjacent height field cells, as in Fig. 10, the airborne mesh may slice through or occlude the ground-based mesh details. In practice, this often happens on building facades. Unfortunately, our assumption that the ground-based mesh is always superior to the airborne mesh does not always hold: in particular on rooftops, we tend to observe small amounts of floating triangles which cut the airborne mesh but do not contribute positively to the appearance of the roof. Therefore as a preprocessing step, we use region growing to identify and remove particularly small patches of triangles in the ground-based mesh.

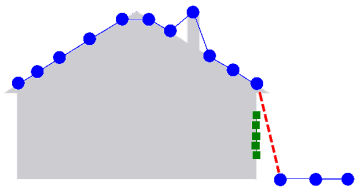

Figure 10. A side view of the height field, in blue circles, and the ground-based data, in green squares. The dashed line obscures the ground-based data.

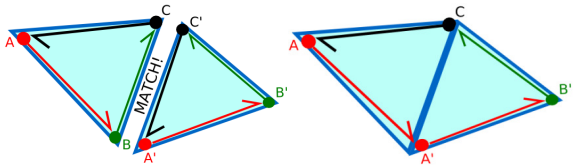

Figure 11. By removing the shared edges and re-linking the circular linked list, we obtain a list of boundary edges encompassing both triangles.

5.2. Finding boundary edges

Now that we have created disconnected ground-based and airborne meshes, we wish to combine these meshes into a connected mesh. Fusing anywhere except the open boundaries of two meshes would create implausible geometry. Therefore, we first find these mesh boundaries in both the ground-based and airborne meshes. We refer to the triangle edges on a mesh boundary as *'boundary edges.'* Boundary edges are identifiable as edges which are used in only one triangle.

We first find the boundary edges of the airborne mesh. For consistency, we use similar triangle data structures for both the ground-based and air mesh: an array of vertices and an array of triangles in which each triangle is specified by three indices into the vertex array. Any edge can be uniquely expressed by its two integer vertex indices. Since boundary edges are defined to be in only one triangle, our strategy for finding them is to iterate through all triangles and eliminate triangle edges which are shared in between two triangles. All edges which remain after this elimination are boundary edges.

In detail our algorithm for finding the boundaries of the air mesh is as follows: For each triangle, we perform the following steps. First, we create a circular, doubly-linked list with 3 nodes corresponding to the edges of the triangle. For example, the data structure associated with triangle ABC in Fig. 11 consists of three edge nodes, namely AB linked to BC, linked to CA, linked back to AB. Second, we iterate through these three edge nodes; in doing so, we either insert them in to a hash table or, if an edge node from a previously-traversed triangle already exists in their spot in the hash table, we "pair them up" with that corresponding edge node. These two "paired up" edge nodes physically correspond to the exact same location in 3D space, but logically originate from two different triangles. Third, when we find such a pair, we remove the "paired up" edge nodes from the hash table and from their respective linked lists, and we merge these linked lists as shown in Fig. 11. After we traverse through all edges of all triangles, the hash table and linked lists both contain all boundary edge nodes of the full mesh.

We now find the boundary edges of the ground-based mesh. The ground-based mesh may be arbitrarily large, and we wish to avoid the need to store all of its boundary edge nodes in memory at once. Instead, our algorithm traverses the triangles of the ground-based mesh in a single, linear pass, incrementally finding boundary edges, merging them with their airborne counterparts, and freeing them from memory. The merge with airborne counterparts is described in more detail in Section 5.4. In overview, our processing of the ground-based mesh will be the same as the processing of the airborne mesh except that (1) rather than one large hash table, we use a circular buffer of smaller hash tables and (2) rather than waiting until the end of the ground-based mesh traversal to recognize boundary edges, we incrementally recognize and process boundary edges during the traversal.

To achieve this, we exploit the locality of vertices inherent to our ground-based triangulation algorithm, described in Section 3: specifically, we observe that the distance between the indices of R and N can never exceed $2 \times$ *Search End*. Therefore, the range of vertex indices for any given edge in the ground-based mesh should similarly never exceed $2 \times$ *Search End*. In practice the range stays well below this value. Furthermore, since the algorithm processes ground-based mesh triangles in order, the minimum vertex index in each triangle monotonically increases because it corresponds to R in Section 3. Therefore, as we traverse through ground-based triangles looking for boundary edges, the algorithm will never see the same edge referenced again by the ground-based mesh after it has seen a vertex with index $2 \times$ *Search End* beyond the lower vertex index of that edge.

These two locality attributes—that the range of vertex indices in a given triangle will never exceed $2 \times$ *Search End* and that R is monotonically increasing—allow us to choose a fixed-size circular buffer of $2 \times$ *Search End* small hash tables as the edge-lookup structure for our ground-based data. As we traverse through all triangles in the ground-based mesh, we place each circularly linked edge node of each triangle in to this data structure. The lower vertex index of the corresponding edge is used as the index in to the circular buffer to retrieve a hash table of all edge nodes which share that index. The higher index of the edge under consideration is then used as the key to the corresponding hash table. The value retrieved from this hash table is the circularly linked edge node. As with the hash table used in processing our air mesh, we check whether there is already a circularly-linked edge node with the same key existing in this hash table. Again as with the airborne mesh, if such an edge node is found, we know that more than one triangle must contain this edge, and it therefore does not correspond to a boundary edge. We can then remove the edge node and its pair from the hash table and merge their linked lists as with the airborne mesh.

Whenever the lowest index of a new edge is too large to fit in the circular buffer, we advance the circular buffer's starting index forward until the new index fits, clearing out all the existing hash tables over which we advance. The edge nodes of any hash tables we clear out

in performing this step must correspond to boundary edges, because we have removed all edges observed to be shared by multiple triangles in the traversal so far, and the locality attributes dictate that all future edges will use larger vertex indices. These edge nodes may therefore be processed as described in Section 5.4 and freed from memory. This process allows us to avoid ever considering more than (2 × *Search End* × *maxValence*) ground-based edges at any one time, where *maxValence* is the maximum vertex valence in the mesh. Note that since *Search End* and *maxValence* do not grow in proportion to the size of the data set, this results in a constant memory requirement with respect to the quantity of ground-based data.

In practice, three or more ground-based triangles occasionally share a single edge. This is because, during ground-based surface reconstruction as described in Section 3, the neighbor point *N* is not constrained to monotonically increase similar to the reference point *R*. Assuming this happens rarely, we can recognize these cases and avoid any substantial problems by disregarding the excess triangles involved.

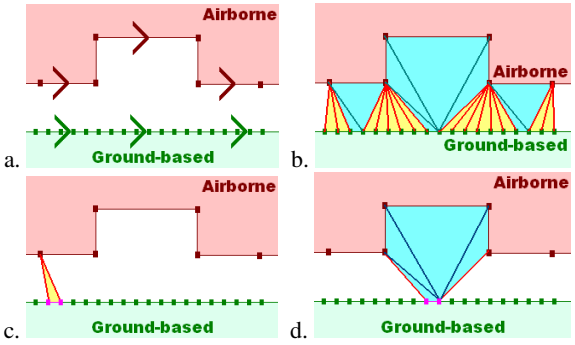

Figure 12. Adjacent ground-based and airborne meshes are merged.

5.3. Merging boundary edges

The merge step occurs incrementally as we find boundary edges in the ground-based mesh, as described in Section 5.2. Given the boundary edges of our airborne mesh and a single boundary edge of the ground-based mesh, we fuse the ground-based boundary edge to the nearest airborne boundary edges, as shown in Figs. 12(a) and 12(b). As a preprocessing step, before finding the boundary edges of the ground-based mesh, we sort the airborne boundary edge nodes in to a 2D grid to facilitate fast spatial queries. For both vertices of the given ground-based boundary edge, we find the closest airborne vertex from the grid of airborne boundary edges. When the closest airborne vertex is closer than a pre-defined distance threshold, we can triangulate. If the two ground-based vertices on a boundary edge share the same closest airborne vertex, we form a single triangle as shown in Fig. 12(c). However, if the two ground-based vertices find different closest airborne vertices, we perform a search through the circular list of airborne boundary edges to create the merge triangles that are shown in blue in Fig. 12(d).

Since objects close to ground level tend to have complex geometry, there is significant ambiguity in deciding which boundary edges, if any, should be merged to the airborne mesh. For example, the boundary edges on the base of a car should not be merged with the airborne mesh. However, boundary edges near the base of a curb should be merged to the airborne mesh. Without high level semantic information, it is difficult to distinguish between these two cases. Additionally, differences in color data between airborne and ground-based sensors create artificial color discontinuities in our models, especially at the ground level. Therefore, we only create merge geometry along edges that are a fixed threshold height above ground level, thus avoiding the merge of ground level airborne triangle with ground level ground-based triangles. This tends to limit our merges to mesh boundaries that have simple geometry and align with natural color discontinuities, such as the boundary between a building façade and its roof. Ground level is estimated by taking minimum height values from a sparse grid as described in Section 4. To further improve quality of merges, we do not merge with small patches in the ground-based triangulation with less than 200 connected triangles, or with loops of less than 20 airborne boundary edges. This avoids merges of difficult, noisy geometry.

5.4. Merging results

Fig. 9 shows the fused mesh from point cloud 2, and Fig. 13 shows the fused mesh from point cloud 4. Table 3 reports run times for point clouds 2 through 5. For point cloud 3, the airborne triangulation and merge take a total of 8392 seconds. It takes 1522 seconds to perform a union find on the ground mesh vertices to calculate the sizes of ground mesh segments, 462 seconds to read the airborne data in to a height map, 1811 seconds to perform median smoothing and hole filling on the height map, 2121 seconds to iterate through the ground-based mesh and mark problematic cells in the height map, 476 seconds to regularly tessellate the height map, and finally 2001 seconds to perform the merge. Processing time therefore scales with the size of our ground mesh, the number of airborne points, and with the dimensions chosen for the height field. In practice, airborne mesh processing does not scale as well as ground-based surface reconstruction or segmentation because it requires streaming more data: in addition to the ground-based mesh data, we copy intermediate blocks of the height map to and from disk. To scale to even larger data, we would additionally need to copy intermediate portions of the airborne boundary edge hash table to disk.

Our technique does over-triangulate if there is detailed ground-based geometry near an airborne mesh boundary. This occurs because multiple ground-based boundaries may be close enough to the airborne mesh to be fused with it, creating conflicting merge geometry. This is a necessary result of the fixed threshold we use to determine when merge triangles should be created; it can be alleviated by performing a second pass over the data to adaptively adjust that threshold, but this might cause the merge step to take twice as long. It would also not completely solve the problem: ambiguities in triangulation are not always correctly solved by choosing the closest mesh boundary. To ensure correct merges, higher level shape analysis may be needed.

Point Cloud	Ground-Based Triangles	# Air Points	Height Map Dimensions	# Merge Triangles	Triangulate and Merge Time (sec)
2	3 M	17 M	436×429	29 K	112
3	146 M	32 M	4729×6151	1.4 M	8392
4	6 M	9 K	215×217	16 K	38
5	32 M	354 K	1682×1203	82 K	2518

Table 3: Merge results for point clouds 2 through 5.

Figure 13. A merged model, with vertices from airborne data colored white.

6. References

[1] N. Amenta, S. Choi, T. K. Dey and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. *Symp. On Comp. Geometry*, pp. 213-222, 2000.

[2] B. Curless and M. Levoy. A volumetric method for building complex models from range images. *SIGGRAPH 1996*, pp. 303-312, 1996.

[3] H. Edelsbrunner. Surface reconstruction by wrapping finite sets in space. Technical Report 96-001, Raindrop Geomagic, Inc., 1996.

[4] C. Frueh and A. Zakhor. Constructing 3D city models by merging ground-based and airborne Views. *Computer Graphics and Applications*, pp. 52-61, 2003.

[5] M. Garland, A. Willmott, and P. Heckbert. Hierarchical face clustering on polygonal surfaces. *Symp. on Interactive 3D Graphics*. pp 49-58, 2001.

[6] M. Gopi and S. Krishnan. A fast and efficient projection-based approach for surface reconstruction. *SIBGRAPI 2002*, pp.179-186, 2002.

[7] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. *SIGGRAPH 2006*, pp 1049-1056, 2006.

[8] A. Mangan and R. Whitaker. Partitioning 3D surface meshes using watershed segmentation. *IEEE Trans. on Visualization and Computer Graphics*, 5(4), pp 308-321, 1999.

[9] R. Pito. Mesh integration based on co-measurements. *IEEE Int. Conf. on Image Processing*, vol. II pp. 397-400, 1996.

[10] G. Turk and M. Levoy. Zippered polygon meshes from range images. *SIGGRAPH 1994*, pp. 311-318, 1994.

[11] Y. Yu, A. Ferencz, and J. Malik. Extracting objects from range and radiance Images. *IEEE Trans. on Visualization and Computer Graphics*, 7(4), pp. 351-364, 2001.

[12] H. Zhao and R. Shibasaki. Reconstructing textured CAD model of urban environment using vehicle-borne laser range scanners and line cameras. *Int'l Workshop on Computer Vision Systems*, pp. 284-297, 2001.