Fast Surface Reconstruction and Segmentation with Terrestrial LiDAR Range Data

by Matthew Carlberg

Research Project

Submitted to the Department of Electrical Engineering and Computer Sciences, University of California at Berkeley, in partial satisfaction of the requirements for the degree of **Master of Science, Plan II**.

Approval for the Report and Comprehensive Examination:

Committee:

Avideh Zakhor
Research Advisor

Date

* * * * * *

Carlo Séquin
Second Reader

Date

Fast Surface Reconstruction and Segmentation with Terrestrial LiDAR Range Data

Matthew A. Carlberg

Video and Image Processing Lab

University of California, Berkeley

`carlberg@eecs.berkeley.edu`

May 8, 2009

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 through, urban planning, disaster management, object recognition, training, and simulations. In this thesis, we present general methods for surface reconstruction and segmentation of 3D colored point clouds, which are composed of partially ordered terrestrial range data. Our algorithms can be applied to a large class of LiDAR data acquisition systems, where terrestrial data is obtained as a series of scan lines. We develop an efficient and scalable algorithm that simultaneously reconstructs surfaces and segments the data. For surface reconstruction, we introduce a technique for setting local, data-dependent distance thresholds, and we present post-processing methods that fill holes and remove redundant surfaces in the generated meshes. We demonstrate the effectiveness of our results on datasets obtained by two different terrestrial acquisition systems. The first dataset contains 94 million points obtained by a vehicle-borne acquisition system during a 20 km drive. The second dataset contains 17 million points obtained by a stationary LiDAR sensor in a stop-and-go fashion over a 0.2 km^2 area.

Contents

1 Introduction

Construction and processing of 3D models of outdoor environments is useful in applications such as urban planning and object recognition. Light detection and ranging (LiDAR) scanners provide an attractive source of data for these models by virtue of their dense, accurate sampling. For example, height information from airborne LiDAR scanners is used extensively for generating digital elevation maps [17] and extracting building footprints in urban and suburban areas [15]. Efficient algorithms also exist to register airborne and terrestrial LiDAR data and merge the resulting point clouds with color imagery [8].

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 [5, 14, 16]. 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 [8]. Their system uses two 2D laser scanners and a single color camera. They triangulated a portion of downtown Berkeley using about 28 million range points obtained during an 8 km drive. Zhao and Shibasaki also present a vehicle-borne, urban modeling system that uses three 2D laser scanners and six color cameras [19]. They created a 3D model of a Tokyo apartment complex, using about 8 million range points obtained during a 1.6 km drive.

In this thesis, 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 terrestrial LiDAR systems and demonstrate how it can be exploited to enable fast algorithms for large scale urban modeling. In particular, we introduce a fast surface reconstruction algorithm, as well as a technique for setting local, data-dependent distance thresholds for use in the meshing process. We also present post-processing methods that remove redundant surfaces and fill holes in the generated meshes. In addition to surface reconstruction, we also present a mesh segmentation algorithm that extracts multi-part objects such as individual buildings, cars, and trees from the scanned scene. We use a simple 3D connectedness criterion to extract such objects. We argue that segmentation is computationally quite cheap once surface reconstruction is complete.

We demonstrate our algorithms on two datasets obtained by two different acquisition systems. The first dataset contains 94 million terrestrial points obtained by a vehicle-borne acquisition system during a 20 km drive. We believe that this is the largest urban dataset reported in the literature. The second dataset contains 17 million points obtained by a stationary laser scanner in a stop-and-go fashion over an area of 0.2 km^2.

The scan line structure we identify for terrestrial LiDAR data can be thought of as a series of adjacent range images that are each a single pixel wide. Fig. 1 illustrates a point cloud, ordered as three scan lines. While Fig. 1 shows vertically-oriented scan lines, we make special note that our algorithms do not require scan lines to be oriented in any particular direction. By making this scan line assumption, 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 [11]. A number of surface reconstruction algorithms triangulate unstructured point clouds [1, 7, 10]. In particular, Gopi and Krishnan report fast results by preprocessing data into a set of depth pixels for fast neighbor searches [10]. 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 obtained, and do not require any preprocessing to "reorder" the point cloud.

In their work on merging range images, Turk and Levoy consider the problem of identifying and removing redundant triangles from meshes [16]. Given two triangular meshes M_a and M_b,

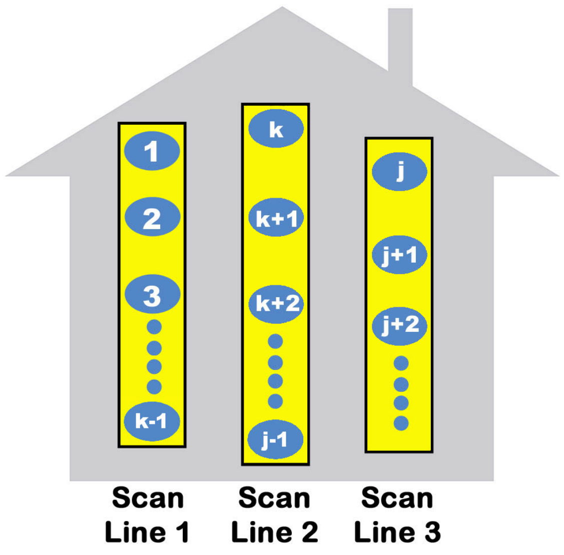


Figure 1: *Scan line illustration. Each blue dot represents a single LiDAR return. We assume that the point cloud is ordered as a series of scan lines as exhibited by the highlighted columns of points. The S1 and S2 datasets were acquired as a series of scan lines; however, we do not have explicit knowledge of where the scan lines begin and end in the data.*

they define a triangle in M_a to be redundant if all three of its vertices are particularly close to non-boundary points on M_b, as determined by a user-defined threshold. They use an algorithm that iteratively removes redundant, boundary triangles on each mesh until both meshes remain unchanged. In this work, we define redundant triangles in a similar manner as Turk and Levoy; however, since urban LiDAR data is usually not in the form of semi-overlapping, gridded range images, we do not directly use their redundant surface removal technique. Rather, we use a voxel-based scheme to ensure that the vertices of every triangle within a voxel are obtained by the same laser sensor at approximately the same time. This approach accounts for the main types of redundancy in urban modeling.

Davis *et al*. present a volumetric method for hole filling in [6]. They take as input any 3D surface model and convert it to a volumetric representation. Namely, they define a signed distance function $d_s(x)$, which assigns a number to each voxel based on how close it is to the surface. Voxels that are not close enough to the original mesh are assigned an “undefined” value. In this context, a hole is understood as an “undefined” region of the voxel space that is surrounded by areas of mesh. Holes are filled by convolving the distance function with a lowpass filter, in order to extend the function into voxels that were previously undefined. The diffusion process propagates inward across holes, eventually spanning them. The zero set of the function in the voxel space is defined as the final surface of the object, and fast marching cube methods are used to re-triangulate the surface without holes. However, the algorithm presented in [6] is not viable for urban modeling, because of the memory and computation constraints imposed by convolving a signed function in a large, 3D voxel space. Instead, we use an efficient technique for identifying mesh boundaries and incrementally fill small holes in the mesh.

Significant work in mesh segmentation has focused on iteratively clustering co-planar faces [9] and grouping triangles that are bound by high curvature [12]. Unlike these works, our segmentation focuses on extracting multi-part objects from the terrestrial mesh. These include objects such as individual buildings, trees, and cars, which are all composed of many connected parts and shapes. Because of the connectedness criterion we use, our algorithm often provides a coarse under-segmentation of a scanned scene. For example, our algorithm typically identifies each individual building as its own segment. However, if two buildings are connected by a fence, our algorithm will identify the two connected buildings as a single object. Thus, our algorithm is complementary with existing work on bottom-up segmentation, and may be combined with more computationally intensive segmentation schemes that use the normalized cut framework [18].

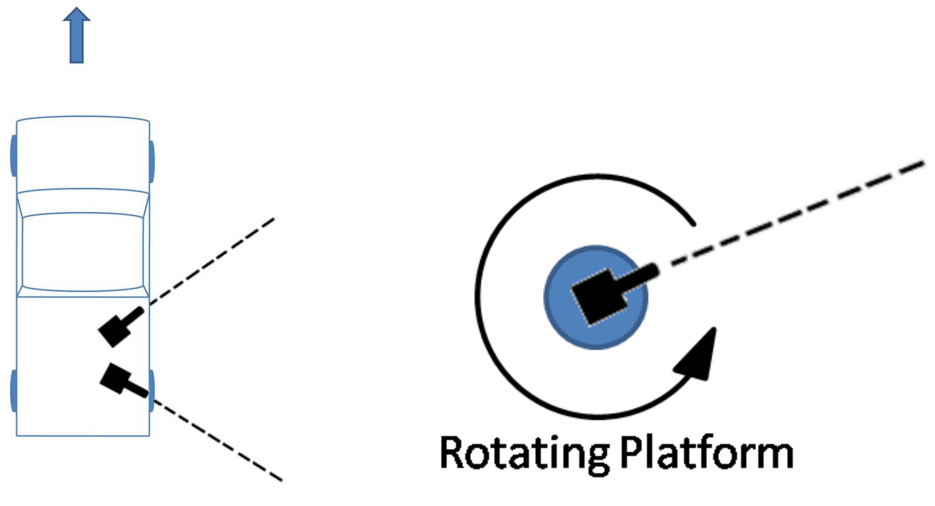
Data from airborne LiDAR is typically much more sparse and noisy than terrestrial data. However, it covers areas which terrestrial sensors often do not reach. In order to produce full 3D urban models, it is necessary to merge airborne and terrestrial data into a single surface model. Carlberg *et al.* [3] and Andrews [2] discuss in detail the problem of merging our terrestrial meshes with airborne data.

In Section 2, we review our assumptions about data acquisition. In Section 3, we demonstrate our meshing technique for terrestrial LiDAR data, and we describe our techniques for adaptive thresholds, redundant surface removal, and hole filling. We present an algorithm for segmenting the generated mesh in Section 4, and we conclude in Section 5.

2 Data Acquisition

Our proposed algorithms accept as input terrestrial LiDAR point clouds. The point clouds have been preprocessed such that each point is specified by an (x, y, z) position in a global coordinate system and an (r, g, b) color value. We make a number of assumptions about the terrestrial 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 significantly longer than the width between scan lines; this helps to identify neighboring points in adjacent scan lines. These requirements for the acquisition system are not particularly constraining, as LiDAR data is often obtained as a series of wide-angled swaths, obtained many times per second [8, 19].

We test our algorithms on two datasets, S1 and S2. In the first dataset, 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. To minimize occluded surfaces in the data, the two vertical scanners are offset $45°$ from each other in the horizontal plane, as illustrated in Fig. 2(a). An S1 input file lists range points from each scanner separately. The S1 terrestrial 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. The second dataset, S2, uses a single 2D laser scanner to obtain terrestrial data in a "stop, scan, and go" manner. The scanner is mounted on a stationary platform, rotates about its vertical axis, and incrementally scans the environment until it has obtained a $360°$ field of view. A top-down illustration of this scanner is shown in Fig. 2(b). For each $360°$ rotation, the scanner acquires approximately 5000 scan lines. The terrestrial data in S2 contains a total of about 19 million points over a 0.2 km^2 area with approximately 700 data points per scan line. To fully cover the 0.2 km^2 area, the S2 scanner acquired data at 6 different locations. S1 and S2 were acquired by different companies as part of projects independent from the UC Berkeley Video and Image Processing Laboratory. As such, neither dataset contains certain desirable attributes, such as scan line numbers or end of scan markers, which are typically used to aid in identifying neighboring points in adjacent scan lines.



(a) (b)

Figure 2: *(a) The S1 dataset is acquired using two 2D vertical laser scanners, mounted on a truck that moves through traffic at normal speed. (b) The S2 dataset is acquired using a single vertical laser scanner mounted on a rotating platform. As opposed to S1, the S2 data is collected in a "stop, scan, and go" fashion.*

3 Terrestrial Surface Reconstruction

3.1 Surface Reconstruction Algorithm

In this section, we propose an algorithm for triangulating terrestrial 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. Since our data does not contain scan line numbers, we use local searches to identify neighboring points in adjacent scan lines. 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. The flow diagram of our surface reconstruction algorithm is shown in Fig. 3.

Since we do not have explicit knowledge of where scan lines begin and end, we use nearest neighbor searches to identify neighboring points in adjacent scan lines. Each nearest neighbor search begins from a point we call the reference point R, as shown in Fig. 3. As indicated by the flow diagram, 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. Beginning from R, we search forward in the point cloud in order to find R's nearest neighbor in the next scan line, and we call this point N. Because we do not know explicitly which points are even in the next scan line, we instead define two user-specified thresholds–*search start* and *search end*–which define the length of each search. The *search start* and *search end* parameters are specified in number of points. The nearest neighbor search begins *search start* points forward of R and ends *search end* points forward of R. The search finds the point within the search space

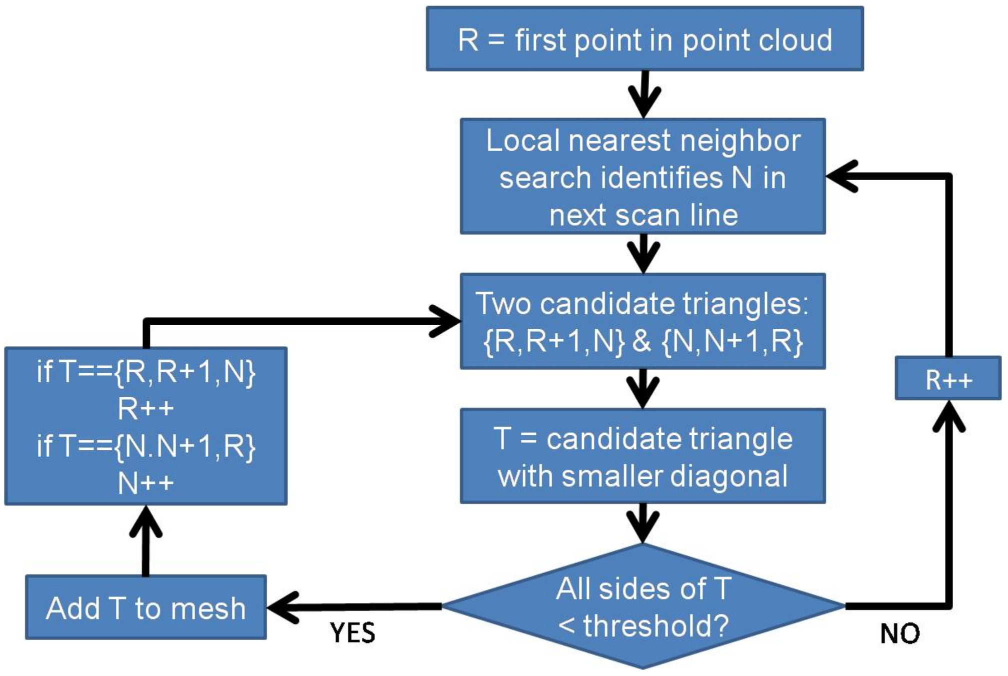

R = first point in point cloud
Local nearest neighbor search identifies N in next scan line
Two candidate triangles: {R,R+1,N} & {N,N+1,R}
T = candidate triangle with smaller diagonal
All sides of T < threshold?
YES
NO
Add T to mesh
if T=={R,R+1,N} R++ if T=={N.N+1,R} N++
R++

Figure 3: *Surface reconstruction flow diagram. The variable R is the current reference point, the variable N is R's neighbor in the adjacent scan line, and the variable T is the current candidate triangle that may be added to the the mesh.*

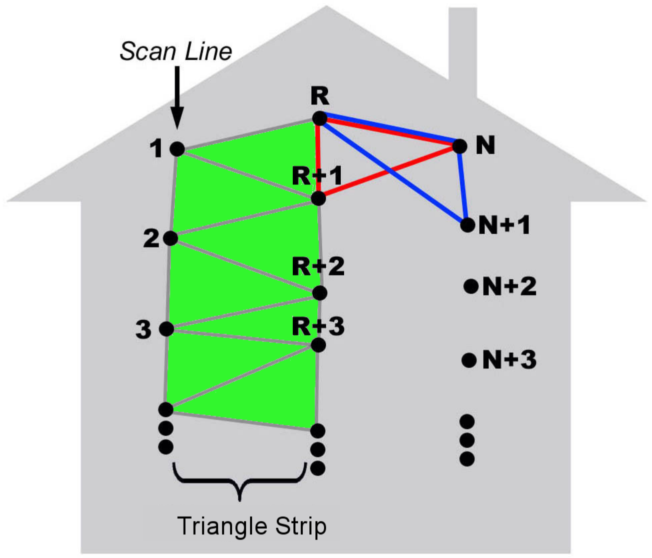


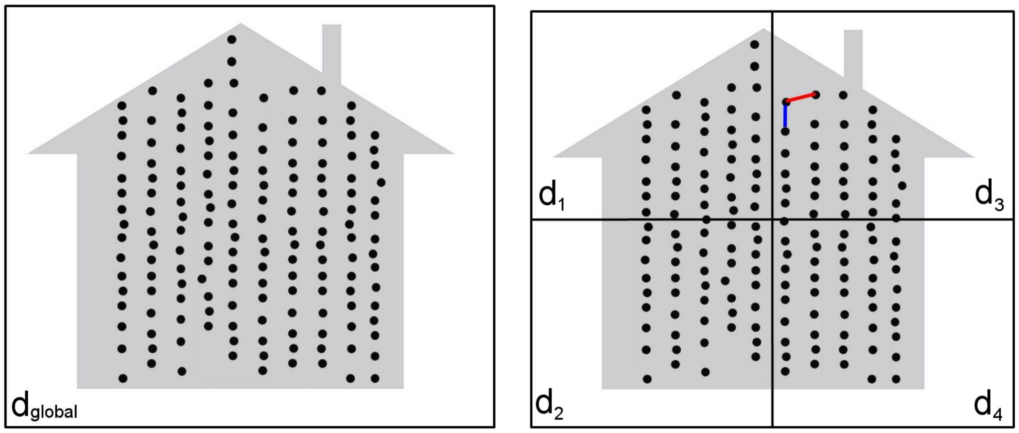
Figure 4: *Surface reconstruction illustration. The triangles shaded green are assumed to have been previously created by the algorithm in such a way that we are currently processing reference point R and neighbor point N. The triangles in blue and red represent the two candidate triangles.*

that is closest in distance to R and defines it as N.

Our nearest neighbor search typically ensures that N is indeed one scan line away from R. Ideally, scan line markers can trivially ensure that R and N are in adjacent scan lines. Similarly, if each data point has a timestamp and the frequency of the LiDAR scanner is known, we can also ensure such a criterion. However, for the general case where the exact scan line structure and timing information are not known, we set our search space parameters in such a way that N is likely one scan line away from R. We choose the *search start* parameter as an estimate of the minimum number of data points in a scan line. This ensures that R and N 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 analyze a small subset of a dataset in order to extract a rough estimate for our search parameters. In particular, distance discontinuities between adjacent points in a point cloud assist in identifying the end of each scan line. Other methods such as analyzing height discontinuities between adjacent points can be used similarly. However, neither of these two heuristics has proven successful in reliably detecting the end of each scan line over the entire dataset.

Once we have identified an R-N pair, triangles are built between the two corresponding scan lines with the goal of generating well-connected surfaces composed of non-skinny triangles. 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. 4. We choose to build the candidate triangle with the smaller diagonal, as long as all sides of the triangle are below a distance threshold d_{global}, which can be set adaptively as described in Section 3.2. If we build the triangle with vertex $R + 1$, we increment the reference point; otherwise, if we build the candidate triangle with vertex $N + 1$, we increment the 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 d_{global}.

3.2 Adaptive Thresholding



(a) (b)

Figure 5: *Illustration of adaptive thresholds on a densely sampled building. (a) For the non-adaptive case, we use a single global distance threshold d_{global} (see Section. 3.1) to triangulate the entire scene. (b) With our adaptive threshold method, the space is subdivided into a set of voxels, and each voxel has its own distance threshold d_i used for triangulation. The threshold chosen for a particular voxel is based on point spacing within that voxel.*

In order to avoid manually hand-tuning a global distance threshold for surface reconstruction, we have developed a voxel-based method to adaptively choose a set of distance thresholds based on local estimates of average point spacing. In particular, in Section 3.1, we described a global distance threshold d_{global} that can be used for triangulating the entire scene. This section describes an alternative method that uses local, data-dependent distance thresholds for each voxel, as illustrated in Fig. 5.

The volume that bounds the input point cloud is first divided into uniformly spaced sub-volumes, or voxels. Each point is sorted into its corresponding voxel, with the goal of calculating a threshold d_i for each voxel i, where $i \in 1, 2, ...L$ and L is the total number of voxels. d_i serves as our distance threshold for triangulating all points within a particular voxel; that is, our surface reconstruction algorithm enforces that all triangles within voxel i must have edge lengths less than d_i. To determine d_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. For example, the blue line in Fig. 5(b) shows one example of the distance between two chronological points in the point cloud. We also find the distance between P and P's nearest neighbor in the adjacent scan line, using the nearest neighbor search described in Section 3.1. For example, the red line in Fig. 5(b) shows one example of the distance between a point and it's nearest neighbor in the next scan line. 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:

$$d_i = \alpha\sqrt{\mu_{i,chron}^2 + \mu_{i,neigh}^2}. \tag{1}$$

Since d_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 impose an upper and lower bound on d_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 our point spacing estimates may not be accurate enough.

The local thresholds depend on the chosen voxel size. If the voxel size is too large, the local threshold does not properly reflect local variations in point spacing. Conversely, if the voxel size is too small, there might not be enough points in each voxel to produce a reliable estimate of point spacing. Smaller voxels also increase computational complexity and memory usage. We have empirically chosen the voxel size in order to capture local variations in point spacing while still being memory efficient for most point clouds. Based on the spatial resolution of datasets S1 and S2, we use a voxel size of $1m \times 1m \times 1m$, resulting in approximately 50-100 points per voxel on average.

3.3 Redundant Surface Removal

Including redundant surfaces in 3D models leads to unpleasant color variations and Z-fighting 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 developed two voxel-based methods for redundant surface removal, which will henceforth be referred to as voxel-based redundant surface removal (VRSR) and distance-based redundant surface removal (DRSR). In general, VRSR and DRSR account for both types of redundancy in urban point clouds by requiring that the vertices of proximal triangles in the mesh come from the same sensor at approximately the same time. Thus, as we are adding triangles to a mesh, we define a new triangle T_{new} as redundant if (1) there is a nonempty set Φ of triangles proximal to T_{new} and (2) any of the triangles in Φ has vertices that originate from a different sensor or at a sufficiently different time from the vertices of T_{new}. We note that condition (1) is effectively a spatial constraint, and condition (2) is effectively a temporal/sensor constraint. VRSR and DRSR differ in how they determine Φ and, thus, how they evaluate condition (1).

For VRSR, every time we attempt to build a new triangle T_{new}, we identify the voxels through which T_{new} passes and examine the triangles previously made in those voxels. If we find that any of these triangles has vertices from a different sensor than the vertices of T_{new} or were acquired at a sufficiently different time than the vertices of T_{new}, we identify T_{new} as redundant 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 this in terms of a maximum allowed point index difference. For example, we expect the first and last points of a point cloud, i.e. the two points with maximal difference in point index, to be far apart in space. If they are close in space, they will typically correspond to a redundant surface in the data.

DRSR is similar to VRSR, except that we introduce a distance threshold d_{drsr} that allows greater flexibility in the composition of Φ. In DRSR, T_{new} is only considered redundant if there are any mesh vertices within a distance of d_{drsr} that come from a different sensor or were acquired at a sufficiently different time than the vertices of T_{new}. Thus, in VRSR we define Φ based on the triangles that fall into a particular voxel, and in DRSR we define Φ via the user-specified parameter d_{drsr}. DRSR is more computationally complex than VRSR due to the extra required distance comparisons, but especially in situations where occlusions are present, DRSR can often provide a better triangulation of the scene. The resulting differences between VRSR and DRSR are discussed in greater detail in Section 3.5.

We now discuss the redundancy present in dataset S1, in which two sensors obtain information about the same area. Two sensors are used in this dataset in order to obtain maximal coverage of the scene, even in the presence of occlusions. As described in Section 2, each sensor is specified separately in an S1 input file. If we blindly triangulate each sensor seperately using the steps described in Section 3.3, our models will effectively contain two separate meshes that overlap each other over a majority of the scene. Using our redundant surface removal technique, data from the second sensor is only used to "fill-in" those areas which the first sensor missed, e.g. due to occlusion. Currently, we do not use techniques for merging the data from the two sensors into a single, connected mesh.

3.4 Hole Filling

In practice, we have found that non-uniform point spacing can result in holes in the 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 consists of 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. These "boundary edges" are of interest because the set of triangle edges that encircle all holes in the mesh E_{holes} is a strict subset of the set of edges on the mesh borders E_{mesh}. The identification of boundary edges is described in detail in [2], and we provide only a brief summary here.

Any triangle edge that falls onto a mesh border, *i.e.* an edge that is part of E_{mesh}, by definition is used in one and only one triangle. Therefore, as we create the mesh, we add triangle edges to a hash table. The hash table allows us to "match up" all triangle edges that are used in two triangles, and thus not part of the mesh border. The remaining triangles that are not matched are indeed part of the mesh boundary. Based on the data locality properties of our surface reconstruction algorithm, the hash table used to identify mesh boundaries only adds a constant memory requirement for hole filling. In fact, at any time, we need only keep $(2 \times search\,end \times maxValence)$ edges in memory at any one time, where *search end* is described in Section. 3.1 and $maxValence$ is the maximum vertex valence in the mesh. For more details on processing mesh boundaries and its applications to both hole filling and, more importantly, merging terrestrial and airborne meshes, we refer the reader to [2].

All boundary edges in the mesh are stored as a set of circularly linked lists, that allow us to easily walk the mesh borders. An example mesh with two sets of boundary edges is shown in Fig. 6(a). Our meshes typically 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 and can approximately be fit by a plane.

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. In order to avoid

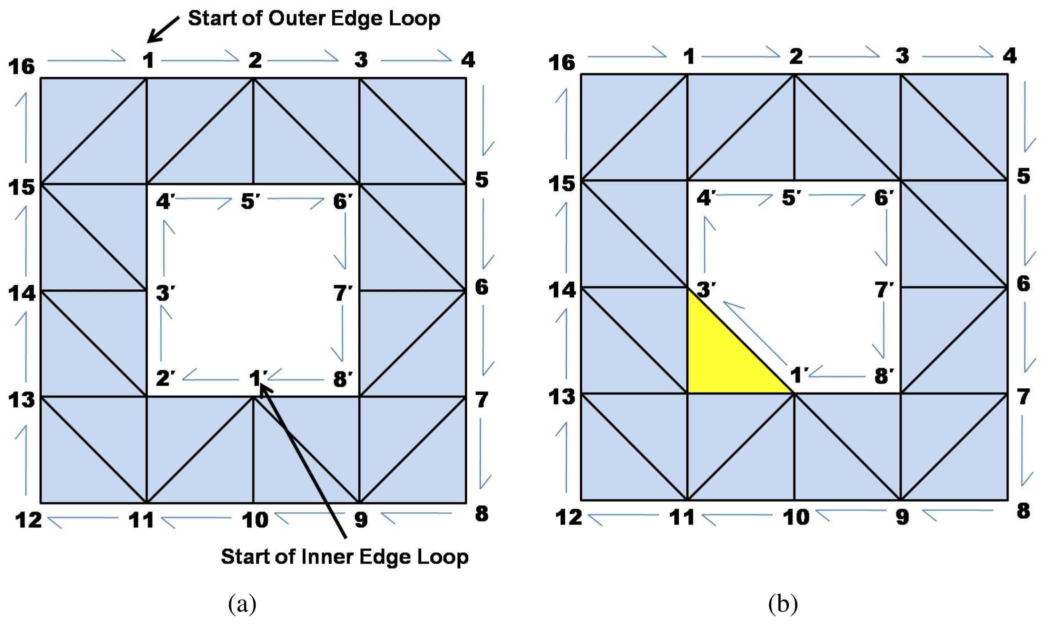


Figure 6: *(a) Illustration of circularly linked lists of boundary edges. (b) To fill holes, we incrementally create triangles and relink the boundary edges.*

skinny triangles, we only build the candidate triangle if its minimum angle is above a threshold. Then, we re-link the circularly linked list, as show in Fig 6(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 and easy to implement, and is reliable on the typical holes observed in urban meshes. However, it has two drawbacks. First, hole identification can fail for small patches of geometry in the mesh. For the example in Fig 6(a), 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, our hole filling method can fail on certain types of holes. For example, some convex holes may not be fillable using our minimum angle criteria; however, we find that this is atypical for the types of holes that appear in our meshes. Also, filling certain types of non-convex holes with our method can result in self-intersecting geometry.

3.5 Surface Reconstruction Results

Point Cloud	Data Set	# Points	Reconstruct Time without Adaptive Threshold (secs)	# Triangles without Adaptive Threshold	Adaptive Threshold Preprocessing Time (secs)	Reconstruct Time with Adapt Threshold (secs)	# Triangles with 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*	-	-	-

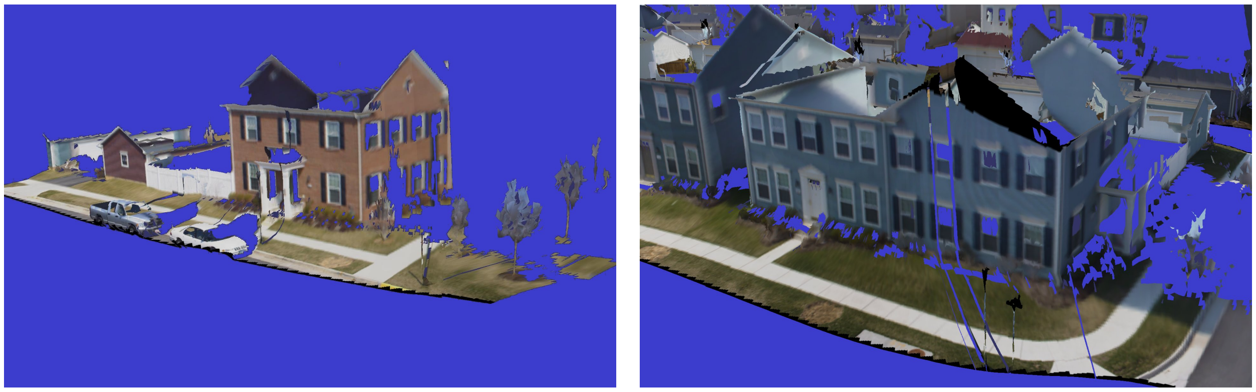
Table 1: *Surface reconstruction results. *Point clouds 3 and 5 do not use adaptive thresholds as text explains.*

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 use adaptive thresholds, because our implementation currently keeps all voxels in memory and cannot scale to point clouds that cover such a large area. Methods could be developed to improve the scalability of our algorithm by only keeping a subset of the voxels in memory at any particular time. In doing so, one must take into account that the data acquisition vehicle can possibly return to the same area in a large voxel space at different times during acquisition. 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 and convert it from an inefficient ASCII format to a more efficient binary format. This initial conversion takes approximately 20 minutes (1200 seconds) for point cloud 3, which contains over 94 million points. The times quoted in Table 1 also do not include the time required to write the final mesh to a renderable scene file format. In particular, for viewing our mesh we use the IVE file format supported by OpenSceneGraph [13], and it takes less than 5 minutes (300 seconds) to generate an IVE file for point cloud 3. 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 writing 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.

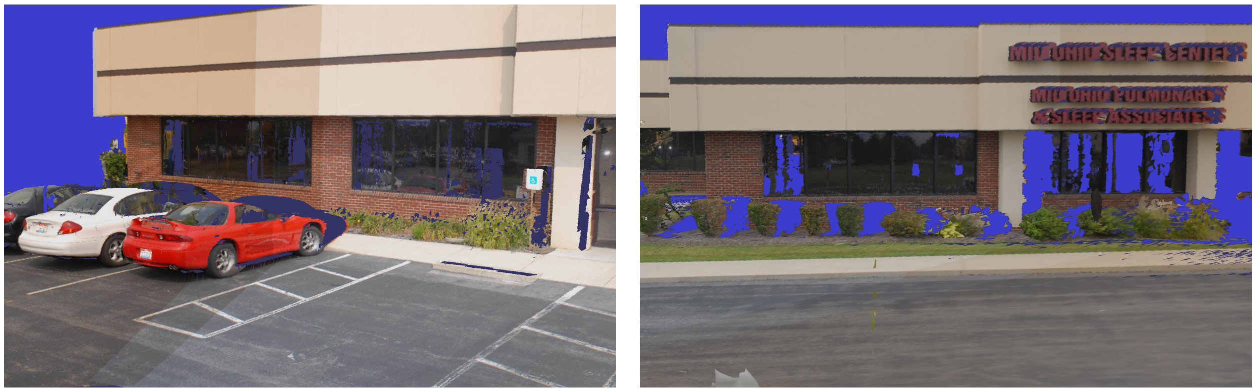
Setting adaptive thresholds is essentially a pre-processing step and takes longer than surface reconstruction because it requires identifying every point's nearest neighbor in the next scan line. However, once adaptive thresholds are set, we observe a speed up in the triangulation portion of the algorithm because we need not repeat the nearest neighbor searches that were performed during the adaptive threshold pre-processing step. 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.

Fig. 7(a) shows the entire 3D surface model of point cloud 1 from S1, and Fig. 7(b) shows a portion of the 3D surface model of point cloud 2 from S1. For S2, two portions of point cloud 4 are shown in Fig. 8. Since point clouds 3 and 5 are so large, there is no commercially available viewer that will display these models without the use of level of detail. Mesh simplification techniques have been extensively studied [4], but we have empirically found that off-the-shelf mesh simplification software does not work on our meshes. In particular, global techniques that identify the best vertex to remove from the mesh do not scale to our large meshes. Further, multi-resolution image processing techniques are not suitable, because our point clouds cannot be easily sorted into a regular grid of points. The meshes shown in Figs. 7 and 8 use a constant threshold of 0.5 m and 0.21 m respectively. For S1 point clouds, the *search start* and *search end* parameters are chosen as 50 and 200 points respectively. For the S2 point cloud, the *search start* and *search end* parameters are chosen as 300 and 800 points respectively.



(a) (b)

Figure 7: *Surface reconstruction result from S1. (a) The entire 3D surface model of point cloud 1. (b) A portion of the 3D surface model of point cloud 2.*



(a) (b)

Figure 8: *Surface reconstruction result from S2. (a) and (b) are both portions of the 3D surface model of point cloud 4.*

Fig. 9(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. 9(b). The voxel size used in our adaptive threshold algorithm is 1m x 1m x 1m.

Fig. 10 shows the results of redundant surface removal using the VRSR technique. The data

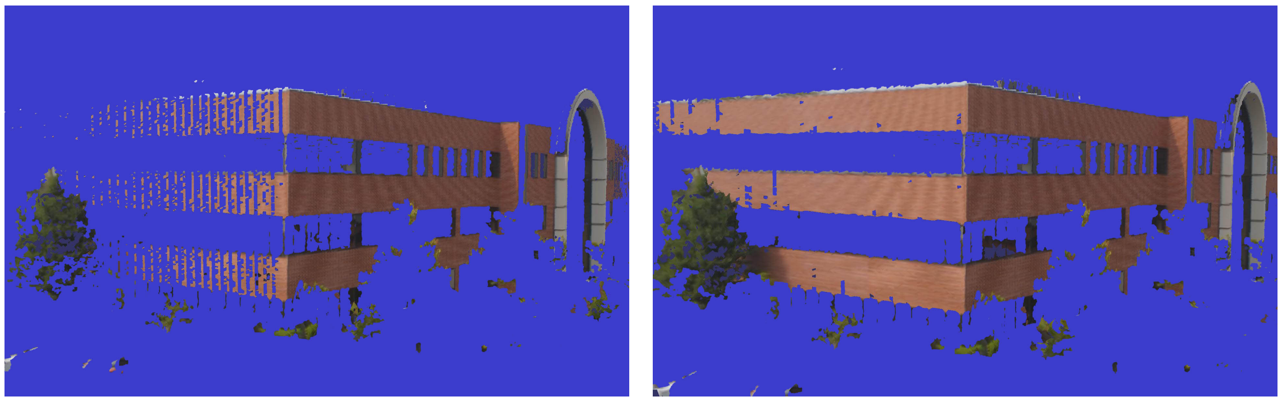
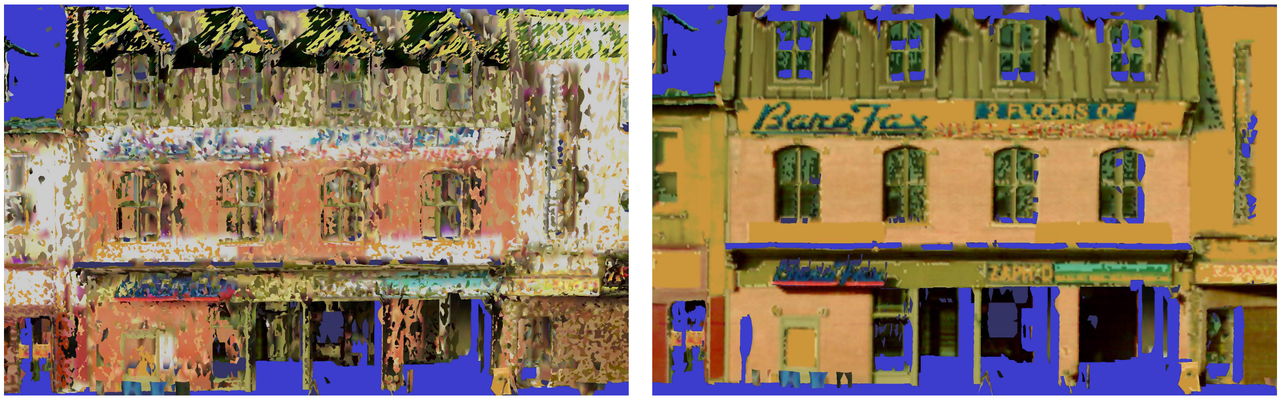

(a) (b)

Figure 9: *Portion of mesh from point cloud 5 (a) with and (b) without adaptive thresholding. In (b), the threshold increases data-dependently resulting in a better triangulation of the building.*

used to generate the meshes in Fig. 10 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. From Fig. 10(a), the drawbacks of including redundant surfaces in urban models is apparent. Because of slight differences in illumination and noise in the sensors, overlapping triangles intersect leading to color inconsistencies and Z-fighting during rendering. For example, the white triangles in Fig 10(a) were most likely obtained when the building was under particularly bright illumination conditions. Intuitively, our redundant surface removal techniques keep "one pass" of sensor data, avoiding these unpleasant visual effects, as shown in Fig. 10(b). The voxel size for redundant surface removal is chosen as 1m x 1m x 1m.

Fig. 11(a) shows a portion of point cloud 2 from S1, which was obtained with two sensors and includes a redundant surface. Fig. 11(b) and Fig. 11(c) show the results of VRSR and DRSR respectively. With VRSR, a portion of the house in Fig. 11(b) remains untriangulated due to occlusion; however, with explicit control of the d_{drsr} parameter (see Section 3.3), DRSR successfully fills in the occluded portion of the house in Fig. 11(b). If the voxel size for VRSR were chosen small enough, it too would reconstruct this occluded surface, which was only "seen" by one sensor. However, decreasing the voxel size also increases the memory and computational complexity of the algorithm.

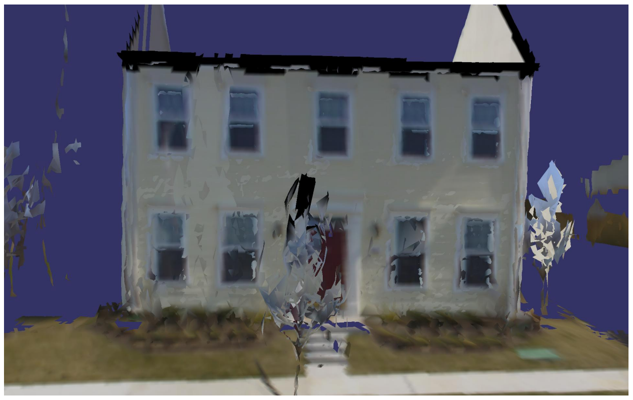


(a) (b)

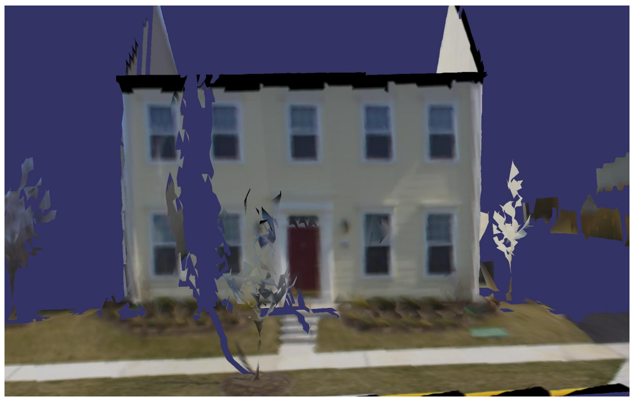
Figure 10: *Mesh (a) with redundant surfaces and (b) without redundant surfaces. The point clouds used to generate these meshes are not from S1 or S2. Instead, they are from a dataset which was obtained using multiple sensor passes.*

We demonstrate hole filling on a different building from the same dataset as used in Fig. 10. The small convex holes exhibited in Fig. 12(a) are filled by our algorithm, as shown in Fig. 12(b). We believe techniques described in [10] for unstructured point clouds may be used to triangulate more complex holes.

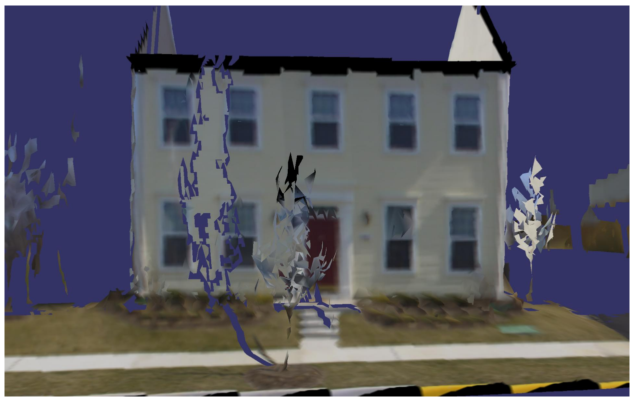
Fig. 13 shows one application of our algorithms, in which our terrestrial mesh from point cloud 2 is merged with corresponding airborne range data to form a single unified 3D urban model. The process of merging terrestrial and airborne meshes is discussed in more detail in [2].



(a)



(b)



(c)

Figure 11: *(a) Mesh generated from a portion of point cloud 2 from S1 without redundant surface removal. Result of redundant surface removal using (b) VRSR technique and (c) DRSR technique. DRSR is computationally more complex but better reconstructs a portion of the house that suffers from an occlusion.*

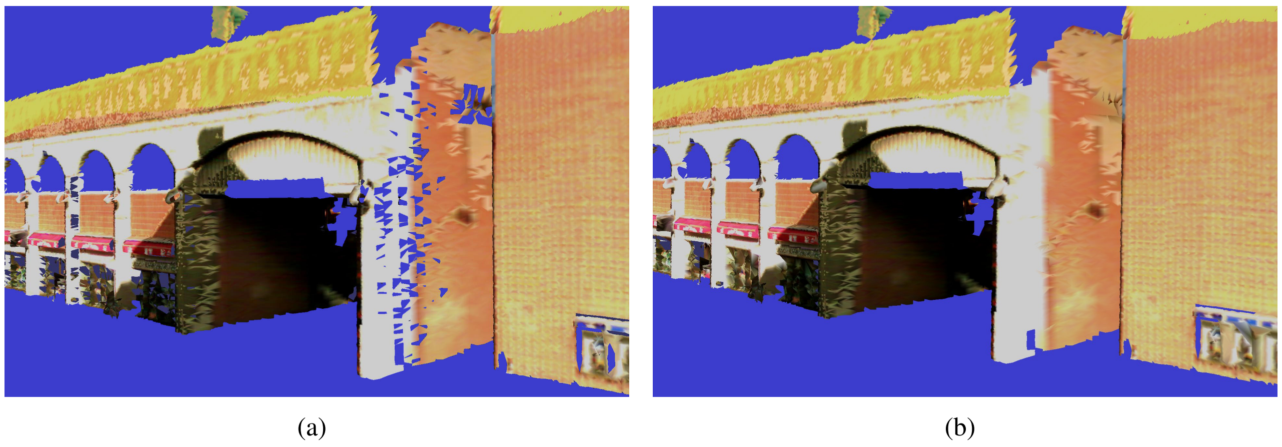


Figure 12: *Mesh (a) with holes and (b) with holes filled. The point clouds used to generate these meshes are not from S1 or S2. Instead, they are from a dataset which was obtained using multiple sensor passes.*

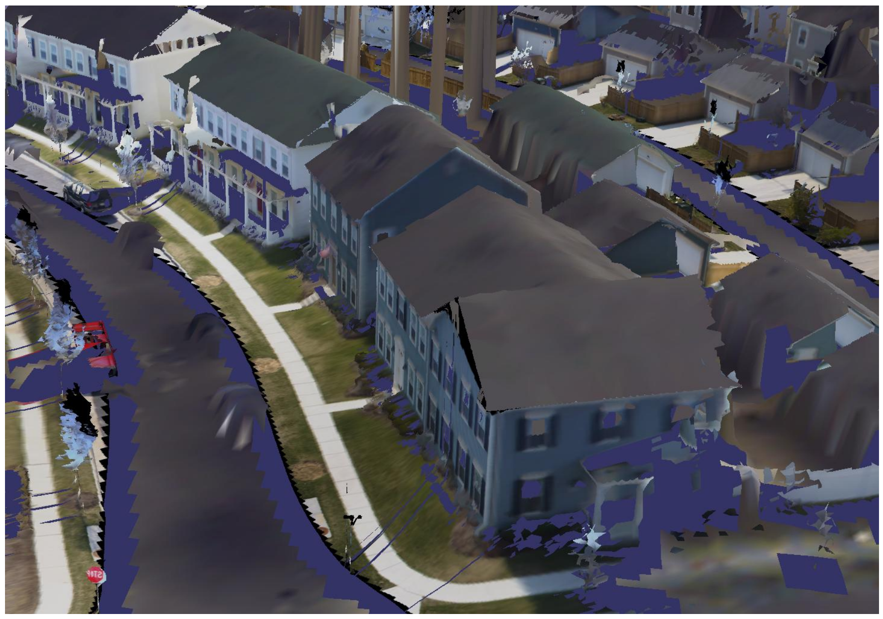


Figure 13: *Combined terrestrial and airborne mesh of a portion of point cloud 2. For details on the ground-air merge, please see [2].*

4 Segmentation Using Terrestrial Data

4.1 Segmentation Algorithm

We now present a mesh segmentation algorithm that is an extension of surface reconstruction. Our goal is to extract multi-part objects such as individual buuildings, cars, and trees from the scanned scene. The algorithm assumes that an object of interest in 3D is attached to the ground and separate from other objects of interest. Our approach is to first identify ground triangles in the reconstructed mesh. Once ground triangles are removed from the scene, we use fast region growing techniques to extract segments containing non-ground, proximate triangles. These segments typically correspond to the objects of interest in our scene. Algorithm 1 describes the segmentation process in detail. Similar to surface reconstruction, we process triangles in order, so as to only keep a subset of them in memory at a given time.

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 in the grid, 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, using our ground height estimates to tag ground triangles. For each triangle built, we project all three vertices onto the grid of ground height estimates in the x-y plane. For each vertex, we find the height difference between the vertex and the ground estimate over a 3×3 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. Typically this ground distance threshold is approximately 0.5 m, and our ground height estimates are made over a grid with cell size 1m x 1m. Therefore, if the ground height varies by more than 0.5m over a 3m x 3m window, our ground identification can fail.

Once surface reconstruction is complete, we pass over the list of triangles and perform region

Algorithm 1 Mesh segmentation algorithm

```
**INPUT:** Unsegmented mesh of triangles ordered as a series of “triangle strips”
**OUTPUT:** Segmented mesh

$T_R \leftarrow$ first triangle;
$T_N \leftarrow$ first triangle;
searchNeeded $\leftarrow$ true;
while $T_N \neq$ last triangle do
    if $T_R$ is a ground triangle then
        $T_R++$;
        searchNeeded $\leftarrow$ true;
        continue;
    end if
    if searchNeeded == true then
        $T_N \leftarrow$ nearest non-ground neighbor of $T_R$ in adjacent triangle strip;
        searchNeeded $\leftarrow$ false;
    end if
    $d1 \leftarrow$ distance between centroids of $T_R$ and $T_N$;
    $d2 \leftarrow$ distance between centroids of $T_R$ and $T_{R+1}$;
    $d3 \leftarrow$ distance between centroids of $T_R$ and $T_{N+1}$;
    $d4 \leftarrow$ distance between centroids of $T_N$ and $T_{R+1}$;
    if $d1 <$ threshold then
        join $T_R$ and $T_N$ into same segment;
    end if
    if $d2 <$ threshold then
        join $T_R$ and $T_{R+1}$ into same segment;
    end if
    if $d1 >$ threshold OR $d2 >$ threshold then
        searchNeeded $\leftarrow$ true;
        $T_R++$;
    else if $d3 < d4$ then
        $T_N++$;
    else
        $T_R++$;
    end if
end while
```

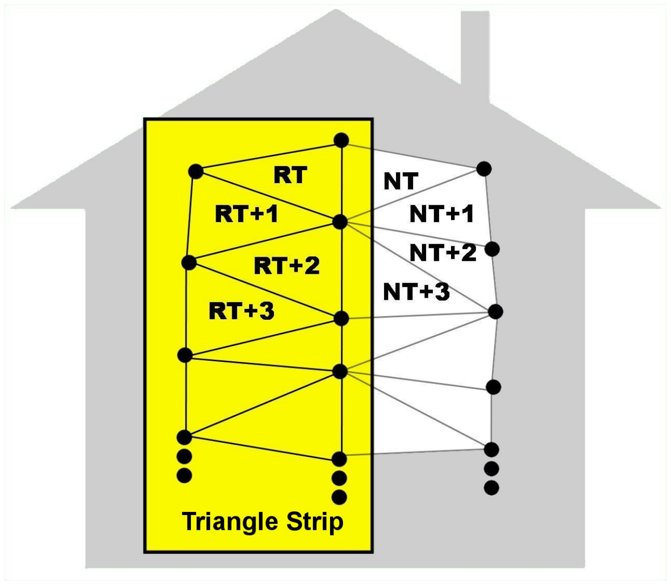


Figure 14: *Illustration of "triangle strips," the partial ordering that is present in the mesh based on how we have performed surface reconstruction.*

growing on non-ground, proximate triangles. The surface reconstruction algorithm has preserved the ordering of triangles as a series of "triangle strips," as illustrated in Fig. 14, where each triangle strip 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 strip whose centroid is closest to the reference triangle. The algorithm then propagates along the pair of triangle strips 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, each segment that contains a large number of triangles, as specified by the region size parameter, is rendered individually.

We have chosen the distance between triangle centroids as our metric of proximity for region growing. It is possible to choose other metrics, 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.

4.2 Segmentation Results

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 computationally cheap byproduct of surface reconstruction. However, segmentation does not scale quite as well as surface reconstruction because it requires copying more data, *i.e.* both points and triangles, to and from disk and because there is extra computation associated with region growing. Fig. 15 shows the mesh of point cloud 1 with ground triangles removed. All 10 segments obtained from point cloud 1 are shown in Fig. 16. 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 Fig. 16(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 of our algorithm is that we do not combine segments that correspond to the same object as obtained from different scanners. For example, Fig. 16(i) is part of the white fence in Fig. 16(a). It is segmented separately because this portion of the fence originates from a different sensor, than the portion of the fence in

Fig. 16(a), due to an occlusion hole.

Fig. 16(f) includes some ground triangles that are not correctly identified as ground, with our relatively simple criterion. If ground triangles are not properly tagged, numerous objects can be erroneously connected together during region growing. Exploring more reliable means of ground identification would make our algorithm significantly more robust.

Point Cloud	# Points	Segmentation Time (secs)	# Segments
1	237,567	1	10
2	2,798,059	13	130
3	94,063,689	2,195	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.*

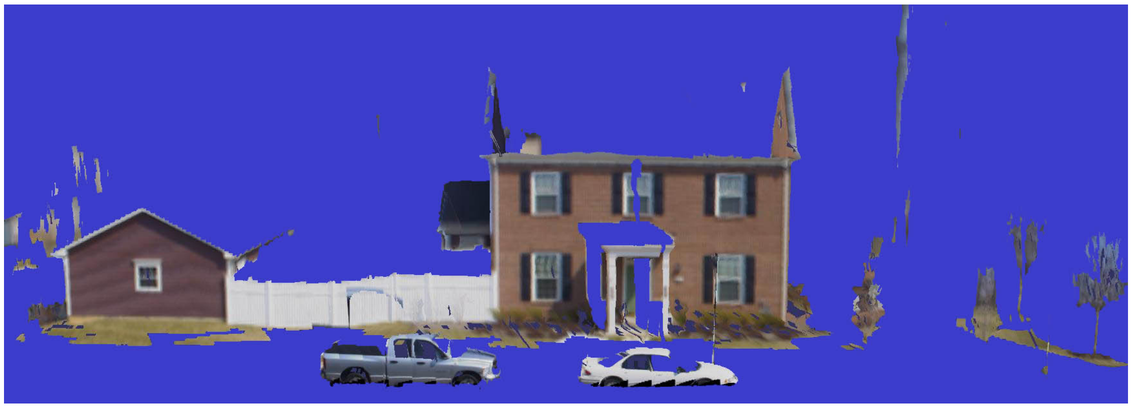


Figure 15: Mesh of point cloud 1 with ground triangles removed.

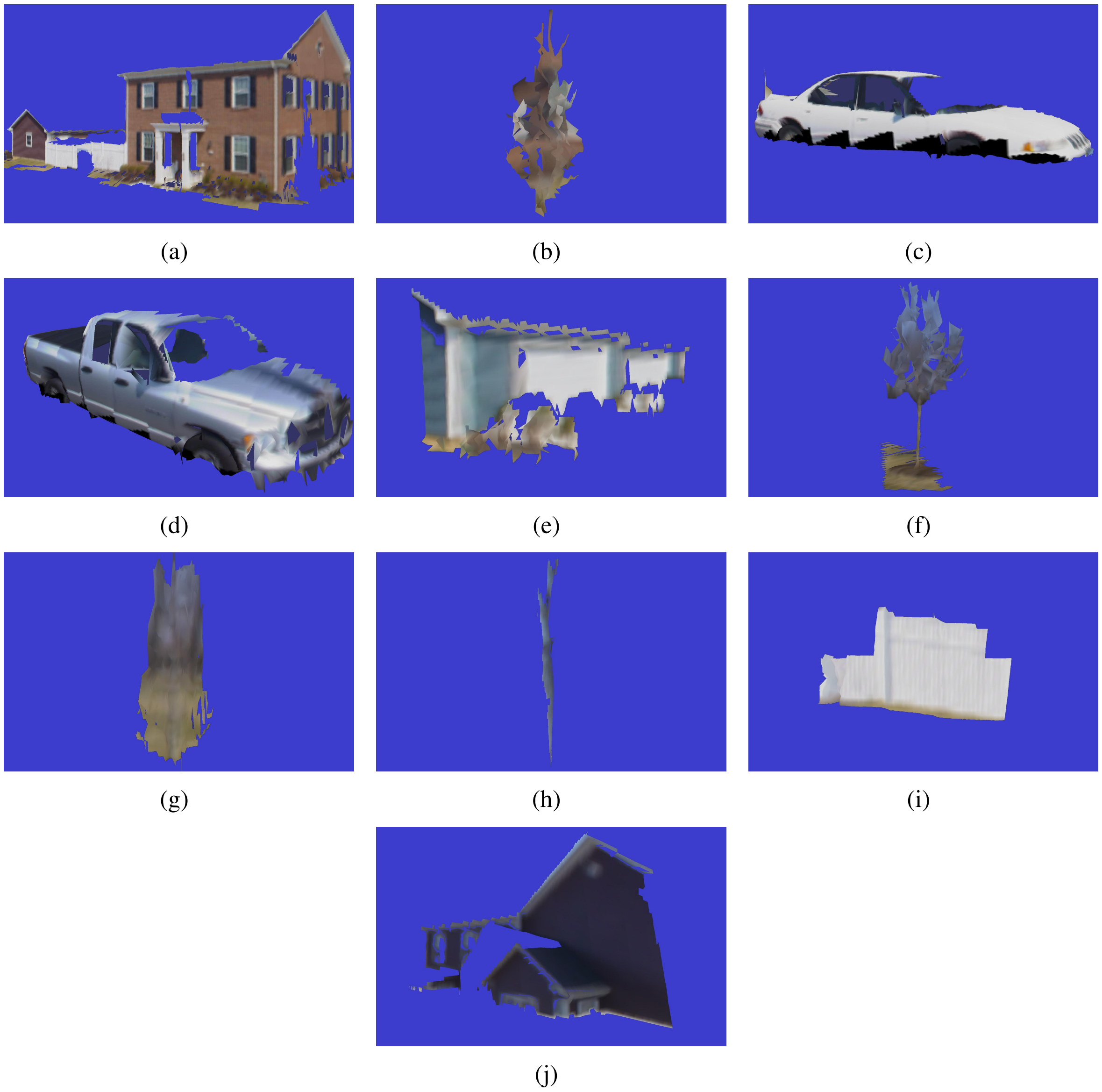


Figure 16: *All 10 segments generated from Point Cloud 1. (a), (e), and (j) are buildings; (b), (f), (g), and (h) are trees; (c) and (d) are cars; (i) is a portion of a fence.*

5 Conclusions

We have presented new algorithms which use the natural structure of many ground-based LIDAR data acquisition systems to enable fast, automatic modeling and segmentation of outdoor environments. Our surface reconstruction and segmentation algorithms scale to point clouds with approximately one hundred million points. We have also introduced techniques for setting data-dependent thresholds, removing redundant surfaces, and filling holes in our generated meshes. For future work, we wish to improve the scalability of our voxel-based adaptive threshold and redundant surface removal algorithms, by taking advantage of the spatial locality in our point clouds and possibly streaming intermediate voxels to and from hard disk. We would also like to explore combining our fast mesh segmentation algorithm with more sophisticated segmentation techniques that use the normalized cut framework [18].

6 Acknowledgments

This work is 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 author, and are not necessarily endorsed by either DARPA or NGA. This material is approved for public release; distribution is unlimited.

References

[1] N. Amenta, S. Choi, T. K. Dey, and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. *SCG '00: Proceedings of the 16th annual symposium on Computational geometry*, pages

213–222, 2000.

[2] J. Andrews. Merging fast surface reconstructions of ground-based and airborne lidar range data. Master's thesis, University of California, Berkeley, 2009.

[3] M. Carlberg, J. Andrews, P. Gao, and A. Zakhor. Fast surface reconstruction and segmentation with ground-based and airborne lidar range data. *3DPVT '08: Proceedings of the Fourth International Symposium on 3D Data Processing, Visualization, and Transmission*, pages 97–104, 2008.

[4] P. Cignoni, C. Montani, and R. Scopigno. A comparison of mesh simplification algorithms. *Computers & Graphics*, 22:37–54, 1998.

[5] B. Curless and M. Levoy. A volumetric method for building complex models from range images. *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 303–312, 1996.

[6] J. Davis, S. R. Marschner, M. Garr, and M. Levoy. Filling holes in complex surfaces using volumetric diffusion. *3DPVT '02: Proceedings of the First International Symposium on 3D Data Processing, Visualization, and Transmission*, pages 428–438, 2002.

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

[8] C. Früh and A. Zakhor. Constructing 3d city models by merging aerial and ground views. *IEEE Computer Graphics and Applications*, 23(6):52–61, 2003.

[9] M. Garland, A. Willmott, and P. S. Heckbert. Hierarchical face clustering on polygonal surfaces. *I3D '01: Proceedings of the 2001 symposium on Interactive 3D graphics*, pages 49–58, 2001.

[10] M. Gopi and S. Krishnan. A fast and efficient projection-based approach for surface reconstruction. *SIBGRAPI '02: Proceedings of the 15th Brazilian Symposium on Computer Graphics and Image Processing*, pages 179–186, 2002.

[11] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of delaunay triangulations. *SIGGRAPH '06: Proceedings of the 33rd International Conference and Exhibition on Computer Graphics and Interactive Techniques*, pages 1049–1056, 2006.

[12] A. Mangan and R. Whitaker. Partitioning 3d surface meshes using watershed segmentation. *IEEE Transactions on Visualization and Computer Graphics*, 5(4):308–321, 1999.

[13] R. Osfield and D. Burns. Open scene graph, 2006. `http://www.openscenegraph.org`.

[14] R. Pito. Mesh integration based on co-measurements. *ICIP '96: Proceedings of the 1996 International Conference on Image Processing*, 2:397–400, 1996.

[15] F. Rottensteiner and C. Briese. A new method for building extraction in urban areas from high-resolution lidar data. *International Archives Photogrammetry and Remote Sensing (IAPRS)*, 34(3A):295–301, 2002.

[16] G. Turk and M. Levoy. Zippered polygon meshes from range images. *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 311–318, 1994.

[17] G. Vosselman. Slope based filtering of laser altimetry data. *International Archives of Photogrammetry, Remote Sensing and Spatial Information Sciences*, 33(B3/2):935–942, 2000.

[18] Y. Yu, A. Ferencz, and J. Malik. Extracting objects from range and radiance images. *IEEE Transactions on Visualization and Computer Graphics*, 7(4):351–364, 2001.

[19] H. Zhao and R. Shibasaki. Reconstructing textured cad model of urban environment using vehicle-borne laser range scanners and line cameras. *ICVS '01: Proceedings of the Second International Workshop on Computer Vision Systems*, pages 284–297, 2001.