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

Abstract

Advances in range measurement devices in recent years have opened up new opportunities and challenges for fast 3D modeling of large scale 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 a general framework for surface reconstruction and segmentation using partially ordered 3D point clouds composed of registered ground-based and airborne range and color 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 reconstructs surfaces and segments ground-based range data simultaneously. 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 multiple acquisition systems.

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. They are also useful because efficient algorithms exist to register airborne and ground-based range data, and to merge the resulting point clouds with color imagery [4].

A wide variety of general techniques currently exist to process 3D scan data [1, 3, 7]. However, as sensors improve and ever larger data sets are collected, faster processing may be required. And as different sources of sensor data become more common, techniques which work on the most general input -- just a point cloud with color data -- are also desirable. To achieve both goals, it is important to exploit the inherent structure of the data. Researchers have previously explored techniques, such as streaming triangulation [5], that do take advantage of weak locality in any data to improve performance of such algorithms. Other work, such as the Berkeley City Modeling project [4], has developed new algorithms directly for specific hardware of their own design, allowing them to take full advantage of the structure of their own data, but making it difficult to accommodate any modifications to the method of data acquisition.

In this paper, we take advantage of a scan-line structure common to most ground-based LIDAR systems to develop fast and relatively general algorithms for meshing and segmentation of ground-based range data. We also introduce a method for fusing these ground-based meshes with airborne LIDAR data. We first discuss our assumptions about data acquisition in Section 2, and then demonstrate a new meshing algorithm for creating 3D models of outdoor environments in Section 3. In Section 4 we present an algorithm for 3D segmentation of objects. Section 5 includes an algorithm for merging our ground-based meshes with data from airborne LIDAR. We conclude and discuss future work in Section 6.

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 assume that the ground-based data is ordered as a series of scan lines. In order to keep the algorithms general for different acquisition systems, we assume that there are a variable number of points per scan line, and that the beginning and end of each scan line are not known *a priori*. We make these assumptions because a LIDAR system does not necessarily receive a data return for every pulse that it emits. We also require the length of each scan line to be significantly longer than the distance between adjacent scan lines, in order to help identify neighboring points in adjacent scan lines. These requirements for the acquisition system are not particularly constraining, as LIDAR range data is often obtained as a series of wide-angled swaths, obtained many times per second [4, 8]. No assumptions are made about the ordering or density of the airborne data.

We test our algorithms on two data sets, which include both ground and airborne data. In the first data set, S1, ground range information is obtained using two vertically oriented laser scanners mounted on the back of a truck

[1] 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 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)

that drives under normal traffic conditions. The scanners are aimed off of the same side of the truck, offset from each other by 45º. An S1 input file lists range points from each scanner separately. The S1 ground data on average has 120 samples per scan line, and the S1 airborne data has roughly 300 points per square meter. The second dataset, S2, uses one ground-based laser scanner that is stationary and vertically oriented. The scanner rotates about an axis and incrementally scans the environment until it has obtained a 360º field of view. The ground data in S2 contains approximately 700 data points per scan line, and the airborne data has roughly 2 points per square meter. Although both ground data sets contain vertically oriented scan lines, this is not a requirement for our algorithms.

3. Ground-based surface reconstruction

In this section, we propose an algorithm for triangulating point clouds that are structured as a series of scan lines. The algorithm requires only one linear pass over all data points, allowing it to scale to large point clouds on the city scale. Our goal is to create well-connected surfaces, advantageous for use in object recognition applications. The algorithm has two basic steps. In the first step, a nearest neighbor search finds two points likely to be in adjacent scan lines. In the second step, the algorithm propagates down the two scan lines, building triangles as long as there are no significant depth discontinuities. Once a depth discontinuity is detected, we perform a new nearest neighbor search, and the process continues.

The nearest neighbor search begins from a point called *mainPoint*, which is initialized to the first point of an input file, typically corresponding to the first point in the first scan line. During triangulation *mainPoint* is incremented until we reach the end of the input file. We perform a search to find *mainPoint*'s nearest neighbor in the next scan line and call this point *neighborPoint*. 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 *mainPoint*. The search finds the point within the search space that is closest in distance to *mainPoint* and defines it as *neighborPoint*. Fig. 1 demonstrates our nearest neighbor search. Triangles shown in red in Fig. 1 are assumed to have been built in such a way that the current *mainPoint* is Point 5. With a search start parameter of 3 and a search end parameter of 7, the nearest neighbor search will begin from Point 8 and end at Point 12; 3 and 7 points away from *mainPoint* respectively. Within the search space, Point 11 is found as Point 5's nearest neighbor, so Point 5 and Point 11 form a *mainPoint-neighborPoint* pair.

For triangulation to work properly, we must ensure that *neighborPoint* is indeed one scan line away from *mainPoint*. In order to do this, we first require that the distance between *mainPoint* and *neighborPoint* be below a distance threshold parameter. We also set our search parameters carefully to make the search as short as possible and to guarantee that points directly below *mainPoint*, which could be very close to *mainPoint*, are not considered in the neighbor search. With these goals in mind, the search end parameter is set to the maximum number of points allowed in a scan line, and the search start parameter is set to the minimum number of points allowed in a scan line. If the nearest neighbor search were to erroneously result in a *neighborPoint* that is in the same scan line as *mainPoint*, triangles with zero or nearly zero area could be built. In practice, we chose the search start and search end parameters by manually analyzing the distance between adjacent points in a dataset's point cloud. Particularly large depth discontinuities indicate the end of a scan line, allowing us to estimate an upper and lower bound on the number of points in a scan line for a dataset.

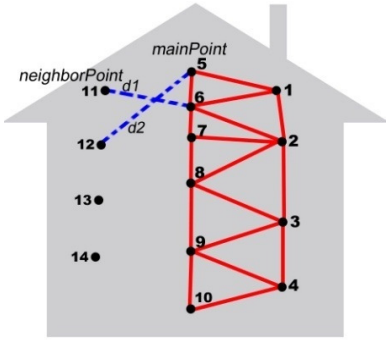

Figure 1. Surface reconstruction illustration.

Once we have identified an initial *mainPoint-neighborPoint* 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 *mainPoint* and *neighborPoint* as two vertices of a triangle. The points directly below them, i.e. *mainPoint+1* and *neighborPoint+1*, provide two candidates for the third vertex and thus two corresponding candidate triangles. We choose to build the candidate triangle with the smaller diagonal, as long as the diagonal is below the distance threshold. In doing so, if we choose to build the candidate triangle with *mainPoint+1*, we increment *mainPoint*; otherwise, if we choose to build the candidate triangle with *neighborPoint+1*, we increment *neighborPoint*. By doing this we have a new *mainPoint-neighborPoint* pair, and can continue our triangle building process without performing a new nearest neighbor search. We only have to perform a nearest neighbor search when we detect a depth discontinuity based on our

distance threshold. Fig. 1 further illustrates how the algorithm builds triangles. The two current candidate triangles are Δ5,11,6 and Δ5,11,12. We build Δ5,11,6 because its diagonal d1 is shorter. Because Δ5,11,6 includes *mainPoint+1*, we increment *mainPoint*. Points 6 and 11 become the new *mainPoint-neighborPoint* pair.

3.1. Surface reconstruction results

All models are created on a 2.66GHz dual-quad core Intel Xeon CPU, with 2.75 GB of RAM. The results of running our surface reconstruction on four different point clouds are specified in Table 1. Fig. 2(a) shows the triangulated mesh of Point Cloud 1 from S1. For the S1 point clouds, the parameters chosen are as follows: 0.5 m distance threshold, 50 point search start, and 200 point search end. Fig. 2(b) shows the triangulated mesh of Point Cloud 3 from S2. For the S2 point clouds, the parameters we chose are as follows: 0.15 m distance threshold, 150 point search start, and 600 point search end.

Our results demonstrate high quality models generated in linear time. In particular, we note that the façades of the house in Fig. 2(a) and the building in Fig. 2(b) are nearly watertight. While the cars in both figures have a number of small holes and extraneous triangles connecting them to the ground plane, they are still visually recognizable. For the S1 data set, we notice some discoloration in the models, especially on the pickup truck in Fig. 2(a). This is due to the redundancy in the S1 data acquisition system. Because there are two scanners mounted on the data acquisition vehicle, most objects in the S1 data set are actually represented by two surfaces. We have not attempted to merge these two surfaces in any way; we simply render both surfaces, making the algorithm significantly faster. For the S2 data set, it is important to understand the reason the algorithm takes significantly longer to generate a mesh for Point Cloud 4 from S2 than it does for Point Cloud 2 from S1, even though both are of similar size. Scan lines in S2 are significantly longer than those in the S1 data set, requiring nearest neighbor searches to be longer. Specifically, there are about 700 points per scan line for S2 versus 120 points per scan line in S1. Also, data points are significantly more spread out in the S2 data set, requiring more frequent nearest neighbor searches due to natural depth discontinuities. Both of these factors contribute to the algorithm running slower on the S2 data set. Using spatial data structures could potentially improve our search speed.

In our experiments, we have used absolute distance thresholds for both S1 and S2 data sets. Because the distance between sampled points in a scan line grows with depth in any LIDAR scanning system, a large threshold is often necessary to create well-connected surfaces at a distance, but causes over-zealous connections on objects close to the scanner. We plan to implement adaptive distance thresholds to solve this problem. During each neighbor search, we plan to choose our distance threshold based on a local estimate of point density.

(a)

(b)

Figure 2. (a) Surface reconstruction of Point Cloud 1. (b) Surface reconstruction of Point Cloud 2.

4. Ground-based segmentation

We present a segmentation algorithm that is an extension of our triangulation algorithm. First, we perform the triangulation as described in Section 3, with a minimal amount of extra computation. Second, we remove ground triangles. Third, we perform region growing on the remaining triangles based on local proximity.

Point Cloud	# Points	Data Set	# Triangles	Processing Time (Triangulation)	# Segments	Processing Time (Segmentation)
1	237,567	S1	440,765	6 s	15	8 s
2	2,872,155	S1	5,199,574	40 s	268	85 s
3	1,186,486	S2	2,269,242	20 s	22	86 s
4	3,000,000	S2	5,312,065	96 s	67	235 s

Table 1: Results of Surface Reconstruction and Segmentation on Four Point Clouds

During triangulation, we perform a number of extra computations that are not described in Section 3 in order to aid in the segmentation process. First, to assist in ground removal, we calculate the unit normal vector of each triangle. Triangles with unit normal vectors that have a particularly large z-component, as specified by a user-defined z-threshold, are tagged as ground candidates. Second, during triangulation, we perform region growing on the vertices of all ground candidate triangles. This allows us to determine the size of potential ground regions for use in our ground removal step. For simplicity, we choose to perform region growing on vertices of triangles rather than the triangles themselves. Finally, we also compute the centroid of each triangle to aid in determining triangle proximity for the last step of the algorithm once all ground triangles have been removed.

After triangulation is completed, we perform a pass over the list of triangles to remove ground triangles. For each ground candidate triangle, we check the size of the region to which its vertices belong. If the region is larger than the number of vertices specified by the user-defined ground size parameter, we declare the triangle to be part of the ground and remove it from the mesh. Ground is thus defined as any large region with normal vectors pointing upwards. We choose this ground definition because hills in urban and residential neighborhoods rarely exhibit more than 10% grade. We specify a minimum size for ground so that we do not wrongly identify a surface, such as the hood of a car, as ground.

With ground triangles removed, the algorithm passes over the remaining triangles one last time, performing region growing on triangles. In this step, we determine local proximity on the triangle level. Two regions are only joined if they have two triangles whose centroids are close together, as defined by a distance threshold parameter. The ground triangulation algorithm described in Section 3 has preserved the ordering of triangles in a useful way. Namely, the list of triangles forms a series of triangle lines, where each line corresponds to a pair of scan lines. Thus, our region growing algorithm, which exploits this inherent structure, is extraordinarily similar to our ground triangulation algorithm, except all calculations are performed with the centroids of the triangles. More specifically, starting from a particular triangle, which is called *mainTriangle*, we find its nearest neighbor in the adjacent triangle line, which is called *neighborTriangle*. Just as before, we define two parameters, segmentation search start and segmentation search end, which define the length of the nearest neighbor search. In general, the segmentation search space should be twice as long as the triangulation search space, because the maximum number of triangles in our models is twice the number of points. Once we have found a *mainTriangle-neighborTriangle* pair, we join the regions to which both triangles belong if the distance between their centroids is below the specified threshold. Similarly, we also check the distance between *mainTriangle* and the triangle directly below it and merge these two regions if their centroids are within the threshold distance. Just as before, we are now latched onto two triangle lines, and can move down the pair only performing a new nearest neighbor search if the threshold criterion is violated. 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.

4.1. Segmentation results

We have run segmentation on the same point clouds as the triangulation algorithm and on the same computer. The segmentation results for the four point clouds are shown in Table 1. The quoted processing times include triangulation, segmentation, and writing all output files. Fig. 3 shows the triangulated mesh of Point Cloud 1 with ground removed. All 15 segments obtained from Point Cloud 1 are shown in Fig. 4. For dataset S1, the use of two scanners produces two different surfaces on many objects, leading to multiple segments for the same object. For instance, Figs. 4(d) and 4(m) clearly correspond to the same house as scanned by two different laser scanners. For the S1 point clouds, we use the same parameters for triangulation as before, and the segmentation parameters are chosen as: 0.8 z-threshold, 800 vertex minimum ground size, 100 triangle search start, 400 triangle search end, 1 m centroid distance threshold, and 1000 triangle minimum region size. For the S2 point clouds, the triangulation parameters are chosen as before, and the segmentation parameters are chosen as: 0.8 z-threshold, 1500 vertex minimum ground size, 400 triangle search start, 1600 triangle search end, 0.3 m centroid distance threshold, and 2000 triangle minimum region size.

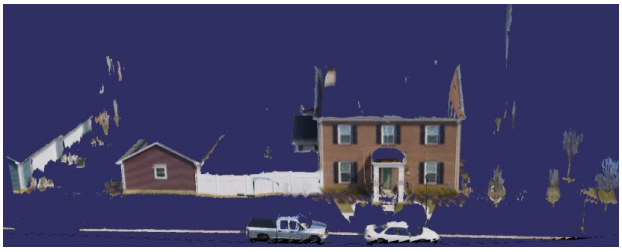

Figure 3: Mesh of Point Cloud 1 with Ground Removed.

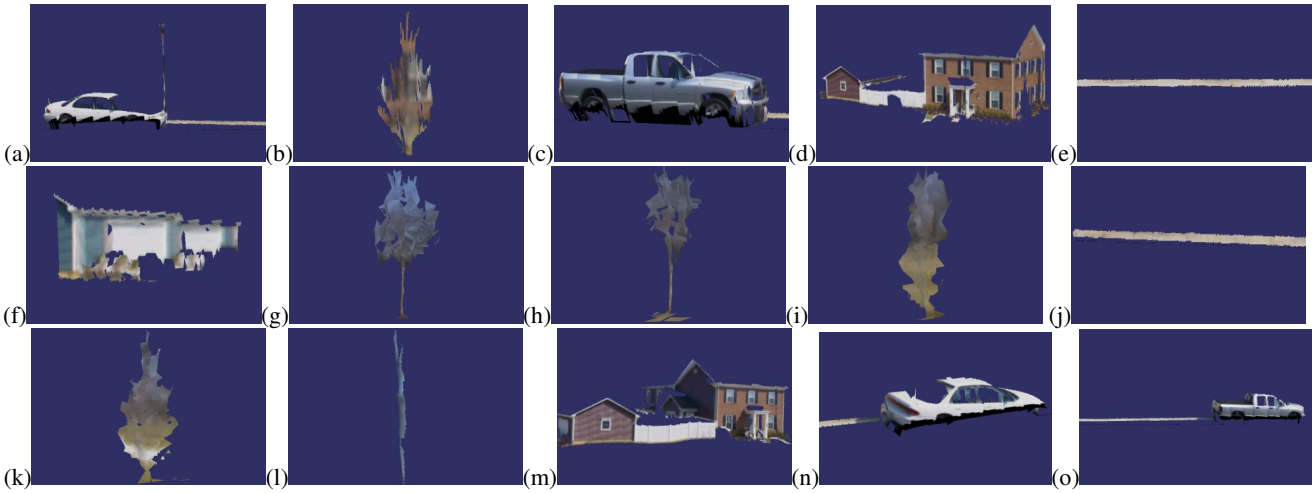

Figure 4. All 15 segments from Point Cloud 1. Duplicate segments exist due to multiple scanners.

The S2 dataset is significantly noisier than the S1 dataset, especially for points near the scanner. In order to properly remove ground, we smooth the normal vectors of the triangles to remove high frequency noise by applying an averaging filter over a window of nine triangles. Failing to remove ground can result in our region growing algorithm undesirably connecting multiple segments together into one through the ground region.

The segmentation algorithm is significantly more sensitive to parameters than the triangulation algorithm, especially with respect to the minimum region and centroid threshold parameters. For example, if in Point Cloud 1, the minimum region is set to 500 instead of 1000 and the centroid threshold is set to 0.5 m instead of 1 m, our algorithm produces 17 rather than 15 segments. Changing the parameters in this way causes an over-segmentation of the scene, whereas the parameters we chose for Fig. 4 provide a slight under-segmentation. To illustrate this under-segmentation, we observe that roadside curbs are often segmented together with cars, because they are so close together. We have observed similar situations with trees and street lamps that are very close to parked cars or the façades of buildings. Further application of pruning algorithms, such as normalized cuts [6], could separate such objects producing a more intuitive segmentation.

It is also important to note that the current definition of ground as a "large" region with upward pointing normals is slightly ambiguous and can be improved. As seen in Fig 4(a), the hood of the white sedan has been removed because it is large enough to be considered a ground region. Histogram analysis on the z values of our points could potentially improve ground detection.

5. Merging with ground 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. 5.

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 triangulation method, we perform this merge with only a constant additional memory requirement with respect to the size of the ground data, and linear memory growth with respect to the air data.

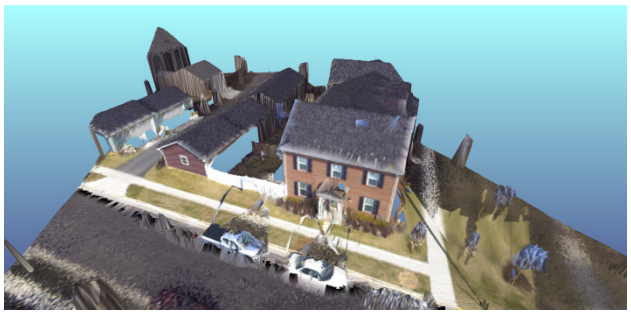

Figure 5. The dense ground-based mesh is merged with a coarser airborne mesh.

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. We 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 mesh. We therefore mark the portions of the height field which are likely to be problematic, and avoid using those portions when we regularly tessellate the height field.

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. 6, the airborne mesh may slice through or occlude the ground-based mesh details. In practice, this often happens on building facades.

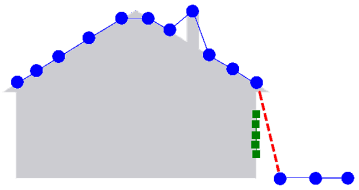

Figure 6. A side view of the height field, in blue circles, and the ground data, in green squares. The dashed line obscures the ground data.

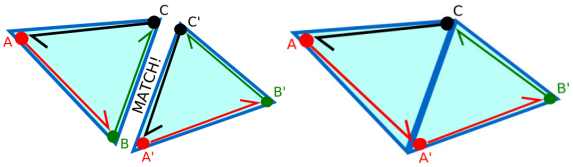

Figure 7. By removing the shared edges and re-linking the circular linked list, we have a list of loose edges encompassing both triangles.

5.2. Finding loose edges

Now that we have created disconnected ground 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 borders in both the ground and airborne meshes. We refer to the triangle edges on a mesh border as *'loose edges.'* Loose edges are identifiable as edges which are used in only one triangle. We post-process the ground and air mesh data to find the loose edges. Each mesh is stored as a list of triangles; each triangle is specified by three unique integer vertex indices referencing a global array of all vertex positions. Any edge can be uniquely expressed by its two integer vertex indices. To simplify finding loose edges, we assume that each triangle has its vertices listed in counterclockwise order, and that each edge is used in at most two triangles. To capture edge adjacency information, we wish to place the loose edges in a circular, doubly-linked list of edge nodes, ordered such that adjacent edges in the list share one vertex.

We first find the loose edges of the air mesh. Some triangle data structures provide a simple way to extract and walk these edges directly [2]; however we benefit from not using these structures, especially on the ground mesh, since we use less memory by only temporarily storing such connectivity information. For consistency, we use similar triangle data structures for both the ground and air mesh. Since loose 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. For each triangle, we perform the following steps. First, we create a three node, circular, doubly-linked list with each node corresponding to an edge in the triangle. For example, the data structure associated with triangle ABC in Fig. 7 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 by re-linking the previous and next nodes of each edge node to point to the previous and next nodes of their "paired" edge node. An example of this is shown in Fig. 7. In general, since these "paired" edges are removed, the nodes remaining in the resulting hash table and linked lists at any time correspond to the loose edges of the mesh if it were to only include the triangles processed up to that time. Thus, after we traverse through all edges of all triangles, the hash table and linked lists both contain all the loose edge nodes of the full mesh.

The key to the fast execution of the above algorithm is the hash table, which is used as a tool to quickly detect whether an incoming edge has been encountered before. As mentioned earlier, the hash table is populated with edge nodes. Specifically, we use a pair of integer vertex indices, which define an edge, as the key to the hash table, and we use the associated edge node of a circular, doubly-linked list as the value. These linked lists are continuously evolving as we keep creating new lists and merging with old lists during the traversal of all mesh triangles. Thus, in

traversing edge *i* of triangle *k*, we are essentially checking whether there is an edge with the same key, i.e. same pair of integer vertex indices, existing in the hash table. If such an edge is found, we retrieve its associated circularly linked edge node in order to merge it with the circularly linked edge node *i* of triangle *j*. This merging is done in such a way that at the end, after all edges have been visited, the resulting circular doubly-linked list contains only loose edges; furthermore, it is ordered so that adjacent edges in the list share one vertex.

One additional detail that needs to be taken care of during the pairing up process is the detection of edge matches regardless of the order in which their vertices are used to describe them. For example key AB for an edge should result in a match to another edge with key BA. To ensure that we consider edges with the same vertices listed in opposite order as equivalent edges, we refer to the edges first by their lower vertex index, i.e. "*lowIndex,*" then by their higher vertex index, "*highIndex*."

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

To achieve this, we exploit the locality of vertices inherent to our ground triangulation algorithm: specifically, the nearest neighbor search performed during ground triangulation in Section 3 cannot find a *neighborPoint* further than *Search End* points away from *mainPoint*, where *Search End* is a user-defined parameter described in Section 3. Since any discontinuity prompts a new search, and the ground triangulation algorithm should never process more than a scan line of points before hitting a discontinuity, the distance between the indices of *mainPoint* and *neighborPoint* can never exceed 2 × *Search End*. Therefore, the range of vertex indices for any given triangle in the ground mesh should similarly never exceed 2 × *Search End*. In practice the range stays well below this value. Furthermore, as the algorithm processes ground mesh triangles in order, the minimum vertex index in these triangles monotonically increases because it corresponds to *mainPoint* in Section 3. Therefore, as we traverse through ground triangles, once that minimum index exceeds the *lowIndex* of an edge under consideration, the algorithm will never see the same edge referenced again by the ground mesh.

These two locality attributes—that the range of vertex indices in a given triangle will never exceed 2 × *Search End* and that *mainPoint* is monotonically increasing—allow us to choose a fixed-size circular buffer of 2 × *Search End* small hash tables as the edge-lookup structure for our ground data. As we traverse through all triangles in the ground mesh, we place each circularly linked edge node of each triangle in to this data structure. The *lowIndex* of the corresponding edge is used as the index into the circular buffer to retrieve a hash table of all edge nodes which share that *lowIndex*. The *highIndex* 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 air 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 loose edge. We can then remove the edge node and its pair from the hash table and merge their linked lists as with the air mesh.

Whenever the *lowIndex* of an edge of the triangle that is being traversed is too large to fit in the circular buffer, we advance the circular buffer's starting index forward until the new *lowIndex* 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 loose 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 no more edges with the same *lowIndex* will be observed. 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 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 data.

In practice, three or more ground triangles occasionally share a single edge. Assuming this happens rarely, we can recognize these cases and avoid any substantial problems by disregarding the excess triangles involved.

5.3. Merging loose edges

This merge step occurs incrementally as we find loose edges in the ground mesh, as described in Section 5.3. Given the loose edges of our airborne mesh and a single

loose edge of the ground based mesh, we fuse the ground-based loose edge to the nearest airborne loose edges, as shown in Figs. 8(a-b). As a preprocessing step, before finding the loose edges of the ground mesh, we sort the airborne loose edge nodes in to a 2D grid to facilitate fast spatial queries. For both vertices of the given ground-based loose edge, we then find the closest airborne vertex from the grid of airborne loose 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 loose edge share the same closest airborne vertex, we form a single triangle as shown in Fig. 8(c). However, if the two ground-based vertices find different closest airborne vertices, we perform a search through the circular list of airborne loose edges to create the merge triangles which are shown in blue in Fig. 8(d).

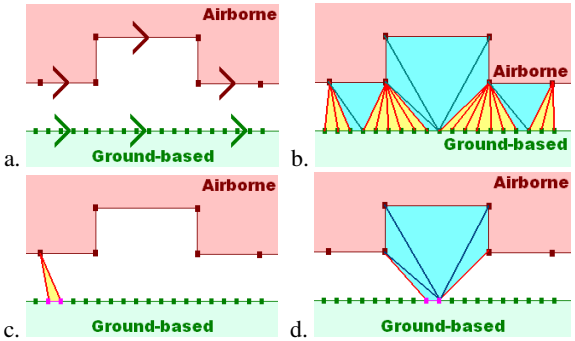

Figure 8. Adjacent ground-based and airborne meshes are merged.

Figure 9. A merged model, with vertices from airborne data colored white.

5.4. Merging results

Figs. 5 and 9 show examples of ground meshes fused with air mesh, using points from the S1 and S2 data sets respectively. The scene in Fig. 9 is created from 5,628,387 ground-based triangles and 4,244 points of airborne data. We identify 441,328, or 3%, of the ground edges as loose edges, and we identify 420, or 3%, of the air edges as loose edges. To fuse the two meshes, 24,717 triangles are added. All together, the airborne triangulation and merge takes 38 seconds; the merge alone takes 13.5, or 36%, of these seconds. The scene in Fig. 5 is created from 442,391 ground-based triangles, and 1,349,273 points of airborne data. It requires 24 seconds to triangulate, 7 seconds of which are used to perform the merge.

Note that, while the previous work of [4] is specifically for merging ground-based building facades with an airborne mesh, our technique works with any ground-based surface. This is particularly valuable for cases such as the triangular holes in the ground mesh behind parked cars in Fig. 5, which we are able to fill with data from the air mesh.

Our technique does over-triangulate if there is detailed ground geometry near an airborne mesh border. This occurs because multiple ground-based borders 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.

6. Conclusions and future work

We have presented new algorithms which use the natural structure of ground-based LIDAR data to enable fast, automatic modeling, segmentation, and merging of outdoor environments. For future work, we would explore making thresholds for the ground-based triangulation and air-ground merge data dependent and using normalized cuts to enhance segmentation.

8. References

[1] N. Amenta, S. Choi and R. Kolluri. The Power Crust. *Symposium on Solid Modeling 2001*, pp. 249-260, 2001.

[2] D. K. Blandford, G. E. Blelloch, D. E. Cardoze, and C. Kadow. Compact Representations of Simplicial Meshes in Two and Three Dimensions. *Int'l Journal of Computational Geometry and Applications*, 15(1), pp. 3-24, 2005.

[3] B. Curless and M. Levoy. A Volumetric Method for Building Complex Models from Range Images. *SIGGRAPH 1996*, pp. 303-312, 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. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming Computation of Delaunay Triangulations. *SIGGRAPH 2006*, pp 1049-1056, 2006.

[6] J. Shi and J. Malik. Normalized Cuts and Image Segmentation. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 22(8), pp. 888-905, 2000.

[7] G. Turk and M. Levoy. Zippered Polygon Meshes from Range Images. *SIGGRAPH 1994*, pp. 311-318, 1994.

[8] H. Zhao and R. Shibasaki1. 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.