

# Merging Surface Reconstructions of Terrestrial and Airborne LIDAR Range Data

*James Andrews*



Electrical Engineering and Computer Sciences  
University of California at Berkeley

Technical Report No. UCB/EECS-2009-70

<http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-70.html>

May 19, 2009

Copyright 2009, by the author(s).  
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

### Acknowledgement

This work is in part supported with funding from the Defense Advanced Research Projects Agency (DARPA) under the Urban Reasoning and Geospatial Exploitation Technology (URGENT) Program. This work is being performed under National Geospatial-Intelligence Agency (NGA) Contract Number HM1582-07-C-0018, which is entitled, 'Object Recognition via Brain-Inspired Technology (ORBIT)'. The ideas expressed herein are those of the authors, and are not necessarily endorsed by either DARPA or NGA. This material is approved for public release; distribution is unlimited.

Distribution Statement "A" (Approved for Public Release, Distribution Unlimited). This work is also supported in part by the Air Force Office of

Scientific Research under Contract Number FA9550-08-1-0168.

---

# Merging Surface Reconstructions of Terrestrial and Airborne LIDAR

## Range Data

by James Andrews

---

### 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:

---

Professor Avidesh Zakhor  
Research Advisor

---

(Date)

\* \* \* \* \*

---

Professor Carlo H. Séquin  
Second Reader

---

(Date)

## Abstract

Recent advances in range measurement devices have opened up new opportunities and challenges for fast 3D modeling of large scale outdoor environments. Applications of such technologies include virtual walk and fly through, urban planning, disaster management, object recognition, training, and simulations. We note that the scan line ordering of terrestrial LIDAR range data is conducive to stream processing, and we show how a fast surface reconstruction algorithm for this data generates the locality necessary to be efficiently represented with a streaming mesh format. We show algorithms for converting such meshes to a streaming mesh format, and for efficiently computing statistics about connected components of these meshes with a memory use that grows linearly as only 0.1% of the number of triangles in the mesh, plus a small constant overhead. Finally, we show how to exploit the locality of streaming meshes to efficiently merge terrestrial meshes with data from airborne sensors. Our merge algorithm requires only a tiny, linearly-increasing amount of additional memory as the amount of terrestrial data increases. We demonstrate the effectiveness and generality of our results on data sets obtained by three different acquisition systems, including a merge of a mesh generated from 129 million airborne points with a terrestrial mesh with over 200 million triangles generated from 570 million terrestrial range points. The algorithm scales linearly in time with the amount of data and is able to merge this large data set in 2.5 hours.

# Contents

<b>1. Introduction</b> .....	4
<b>2. Data acquisition</b> .....	7
<b>3. Terrestrial surface reconstruction</b> .....	8
<b>3.1 Surface reconstruction algorithm</b> .....	8
<b>4. The streaming mesh format</b> .....	11
<b>4.1 Review of the format</b> .....	12
<b>4.2 Suitability of terrestrial reconstruction for streaming</b> .....	15
<b>4.3 Converting to a streaming mesh</b> .....	17
<b>4.4 Computing non-local statistics on a streaming mesh</b> .....	19
<b>5. Merging airborne and terrestrial data</b> .....	25
<b>5.1. Creating and triangulating the height field</b> .....	26
<b>5.2. Finding boundary edges</b> .....	29
<b>5.3. Merging boundary edges</b> .....	32
<b>5.4. Merging results</b> .....	34
<b>6. Conclusions and future work</b> .....	38
<b>7. Acknowledgements</b> .....	39
<b>8. References</b> .....	40

# 1. Introduction

Construction and processing of 3D models of outdoor environments is useful in applications such as urban planning and object recognition. LIDAR (**L**ight **D**etection **a**nd **R**anging) scanners provide an attractive source of data for these models by virtue of their dense, accurate sampling.

Significant work in 3D modeling has focused on scanning a stationary object from multiple viewpoints and merging the acquired set of overlapping range images into a single mesh [2,9,10]. However, due to the volume of data involved in large scale urban modeling, data acquisition and processing must be scalable and relatively free of human intervention. Frueh and Zakhor introduce a vehicle-borne system that acquires range data of an urban environment, while the acquisition vehicle is in motion under normal traffic conditions, and merges that data with airborne data [4, 16, 17, 18]. They model an area in Berkeley by merging a height map constructed from airborne range points with a mesh of building facades constructed from terrestrial range points. Although they collected 60 million airborne range points and 85 million terrestrial range points over a 24.3 kilometer drive, they only presented full airborne/terrestrial merge results for an 8 km drive in downtown Berkeley corresponding to an area of  $.16 \text{ km}^2$ , consisting of 12 city blocks. In processing their airborne data, they applied the QSLIM mesh simplification algorithm to reduce the number of airborne triangles from about 1.3 million to about 100,000 triangles per square kilometer. Thus, they merged a total of 16,000 airborne triangles with approximately 28 million terrestrial triangles to create a merged model for a 12 block area of downtown Berkeley. Their methods take advantage of their ability to carefully design and control the entire pipeline, from data collection through to rendering, and they focus specifically on merging building facades with height maps.

In this report, we introduce a scalable algorithm for merging an existing terrestrial surface reconstruction with an airborne data set. We assume the terrestrial surface reconstruction has already been generated, and our goal is to generate a surface from the airborne data and merge it with the terrestrial reconstruction. We attempt to minimize our assumptions about the nature of the input data: the airborne data may come from any unordered point cloud, while the terrestrial surface reconstruction is only restricted by the requirement that it exhibits the locality necessary to be efficiently represented by a streaming format [14]. By streaming format, we mean a format that lends itself to stream processing: it will be read sequentially from disk and only a small portion of the data will be kept in memory at any given time. We show that range acquisition systems exhibiting a common scan line structure naturally produce this locality, especially when processed by an efficient algorithm introduced by Carlberg *et al.* [13]. We demonstrate our algorithms on three data sets obtained by three different acquisition systems.

We assume our terrestrial LIDAR data sweeps a laser in a line to collect a set of coplanar data points which can be thought of locally as 2.5D range images. By assuming points are ordered in scan lines, we can incrementally develop a mesh over a large set of data points in a scalable way. Other surface reconstruction algorithms, such as streaming Delaunay triangulation, do not identify any explicit structure in their data, but instead take advantage of weak locality in any data [7]. A number of surface reconstruction algorithms triangulate unstructured point clouds [1,3,6]. In particular, Gopi and Krishnan report fast results by preprocessing data into a set of depth pixels for fast neighbor searches [6]. Since they make no assumptions about point ordering, they must alternatively make assumptions about surface smoothness and the distance between points of multiple layers. In contrast, we make assumptions about how the data is obtained, and do not require any preprocessing to reorder



the point cloud. Note that these assumptions only ensure the suitability of our mesh for efficient streaming: any algorithm that meshes with good locality can be used.

Streaming mesh formats were introduced by Isenburg and Lindstrom for efficient out of core processing of large data sets [14], and have shown to be an efficient choice for LIDAR data due to the spatial locality that occurs naturally due to the physical acquisition process [7]. These formats provide finalization information that specifies when a point will not be referenced by any more triangles, and so need not remain in memory. Our usage of the format capitalizes on the same strengths, and uses these strengths for a new application. We also rely on even stronger locality than previous work, which we can assume thanks to the aforementioned scan line structure we identify in terrestrial LIDAR data.

Merging airborne and terrestrial meshes can be seen as a special-case version of merging range images, and our method in particular resembles the seaming phase of [9]. However, our algorithm is tailored to take advantage of the locality and asymmetry inherent to the problem of merging terrestrial and airborne LIDAR data. We achieve fast, low-memory merges, which use the higher resolution geometry in the terrestrial mesh whenever it is available.

In Section 2, we review our assumptions about data acquisition. We explain the method for generating the terrestrial surface reconstruction in Section 3. Section 4 provides a review of the streaming mesh format, and introduces new algorithms for creating and processing them. Section 5 introduces the algorithm for merging the terrestrial mesh with airborne LIDAR data and presents results. Finally, Section 6 presents our conclusions and future work.

## 2. Data acquisition

Our proposed algorithm accepts as input a terrestrial surface reconstruction and a point cloud of airborne data, preprocessed to exist in the same coordinate frame. Each point is specified by an  $(x,y,z)$  position in a global coordinate system and an  $(r,g,b)$  color value. We do not make any assumptions about the density or point ordering of the airborne LIDAR data. However, we do assume the terrestrial reconstruction leads to a mesh that is suitable for streaming, as described in detail in Section 4. To justify this assumption, in Section 3 we show an algorithm that generates suitable meshes by relying on the common scan line structure of the terrestrial acquisition systems used to collect our test data. All three systems possess this structure, which can be described by two key properties. First, they collect point data as a series of scan lines, as illustrated in Fig. 1, enabling us to incrementally extend a surface across a set of data points with only a local search. Second, the length of each scan line is significantly longer than the width between scan lines. This is important for identifying neighboring points in adjacent scan lines, as there are a variable number of data points per scan line, and the beginning and end of each scan line is not known *a priori*. This ambiguity occurs because a LIDAR system does not necessarily receive a data return for every pulse that it emits. We believe these properties are widespread, as LIDAR data is often obtained as a series of wide-angled swaths, obtained many times per second [4,12].

We test our algorithms on three data sets, shown in Table 1, which include both terrestrial and airborne data. In the first data set, S1, terrestrial data is obtained using two vertical 2D laser scanners mounted on a vehicle that acquires data as it moves under normal traffic conditions. The second dataset, S2, uses one 2D laser scanner to obtain terrestrial data in a stop-and-go manner. The scanner rotates about the vertical axis and incrementally scans the environment until it has obtained a  $360^\circ$

field of view. The third data set, S3, is similar to S1 and also uses two vertical truck-mounted scanners, but it additionally uses two horizontally-oriented scanners. The scale and resolution of each data set is listed in Table 1.

Data set	Area covered	Terrestrial points per scan line	Airborne point density per square meter	Total number terrestrial points	Total number airborne points
S1	7.3 km <sup>2</sup>	120	300	94 M	32 M
S2	2.0 km <sup>2</sup>	700	2	19 M	354 K
S3	6.6 km <sup>2</sup>	200	65	570 M	129 M

**Table 1. Size of the three test data sets. Note that airborne points are not uniformly distributed; airborne point densities are measured in a representative well-covered regions of the data set. Airborne data is typically much sparser in less-covered regions.**

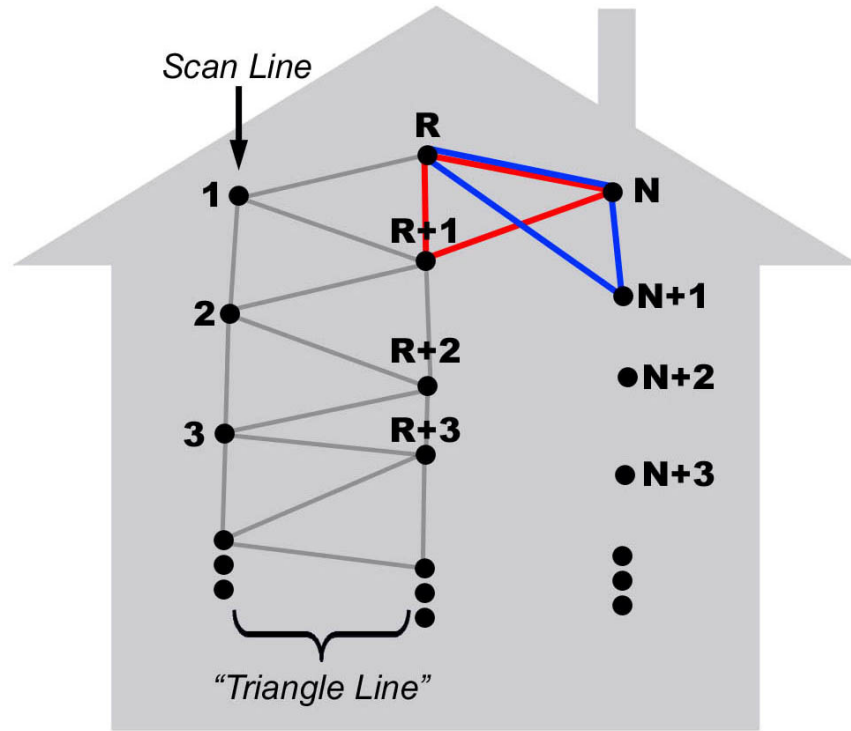
### 3. Terrestrial surface reconstruction

#### 3.1 Surface reconstruction algorithm

In this section, we briefly describe the algorithm proposed by Carlberg et al [13] to generate our terrestrial meshes. The contribution of this report is not the algorithm itself, but identifying the properties of the algorithm that can be exploited to efficiently merge the resulting mesh with airborne data. In this report, we review the basic details of the algorithm, and describe the way it permits the efficient merge operation. For details on how to make the algorithm automatically choose thresholds, remove redundant surfaces, and fill holes in surfaces, refer to [13].

The terrestrial surface reconstruction algorithm processes data points in the order in which they are obtained by the acquisition system, allowing fast and incremental extension of a surface over the data in linear time. We only keep a subset of the input point cloud and output mesh in memory at any given time, so the 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.



**Figure 1. Surface reconstruction illustration.**

Each nearest neighbor search begins from a point we call the reference point  $R$ , as illustrated in Fig. 1.  $R$  is initialized to the first point of an input file, typically corresponding to the first point of the first scan line, and is incremented during triangulation until we reach the end of the file. We perform a search to find  $R$ 's nearest neighbor in the next scan line and call this point  $N$ . The search requires two user-specified parameters—*Search Start* and *Search End*—which define the length of each search. Specifically *Search Start* defines how many points after  $R$  in the input file we skip before starting the

search, while *Search End* defines how many points after  $R$  we traverse before ending the search. The search finds the point within the search space that is closest in distance to  $R$  and defines it as  $N$ .

For proper surface reconstruction, we must ensure that  $N$  is indeed one scan line away from  $R$ . Ideally scan line boundaries would be marked exactly in the data, but in practice none of the vendors who collected our data were able to provide these annotations. Since we process points chronologically in the order in which they were obtained, this criterion can easily be enforced if each data point has a timestamp and the frequency of the LIDAR scanner is known. However, in the general case without any timing information, we must choose the search start and search end parameters carefully. We choose the *Search Start* parameter as an estimate of the minimum number of data points in a scan line. This ensures that  $N$  and  $R$  are not in the same scan line, a situation which can lead to triangles with zero or nearly zero area. We choose the *Search End* parameter as an estimate of the maximum number of points in a scan line to ensure that  $N$  is not multiple scan lines away from  $R$ , a situation which can lead to self-intersecting geometry. In practice, we manually analyze the distance between adjacent points in a dataset's point cloud. Semi-periodic distance discontinuities, while not reliable enough to indicate the precise beginning and end of a scan line, provide a rough estimate for our two search parameters.

Once we have identified an  $R$ - $N$  pair, triangles are built between the two corresponding scan lines, as shown in Fig. 1. We use  $R$  and  $N$  as two vertices of a triangle. The next points chronologically, i.e.  $R+1$  and  $N+1$ , provide two candidates for the third vertex and thus two corresponding candidate triangles, as shown in red and blue in Fig. 1. We choose to build the candidate triangle with the smaller diagonal, as long as all sides of the triangle are below a distance threshold, which can be set adaptively as described later. If we build the triangle with vertex  $R+1$ , we increment  $R$ ; otherwise, if we build the candidate triangle with vertex  $N+1$ , we increment  $N$ . By doing this, we obtain a new  $R$ - $N$

pair, and can continue extending the mesh without a new nearest neighbor search. A new search is only performed when we detect a discontinuity based on the distance threshold.

An advantage of this algorithm is that the resulting mesh has strong locality. In other words, if we consider the mesh in index-buffer format, as a list of triangles referencing indices into an array of vertices, then triangles only use vertices within a small, monotonically increasing range of that array. In Section 4, we show how to exploit this property to represent the mesh in an efficient streaming format.

Results of the algorithm, discussion of its performance, and extensions such as redundant surface removal and hole filling can be found in [13].

## **4. The streaming mesh format**

A strength of the algorithm described in Section 3 is that its output is suitable for representation with the “streaming mesh” format proposed by Isenburg and Lindstrom [14], which enables efficient processing. In contrast to a traditional indexed mesh format, a streaming mesh format interleaves vertices and triangles in one stream, and includes “finalization” information to indicate when a vertex will no longer be referenced by future triangles. We will give a brief review of the details of the streaming mesh format, which we exploit for efficient mesh processing; for full details on the streaming mesh format, refer to [14].

Indexed Mesh	Streaming Mesh
Vertex array: <0.4, 1.3, 2.3> <5.4, 1.2, 2.2> <1.2, 3.4, 6.7> <2.3, 2.4, 6.5> <2.1, 3.9, 6.1>  Triangle Array: <1, 2, 3> <3, 2, 4> <3, 5, 4>	Vertex: <0.4, 1.3, 2.3> Vertex: <5.4, 1.2, 2.2> Vertex: <1.2, 3.4, 6.7> Triangle: <-3, 2, 1> Vertex: <2.3, 2.4, 6.5> Triangle: <2, -4, 1> Vertex: <2.1, 3.9, 6.1> Triangle: <-3, -1, -2>

**Listing 1. The same example mesh in indexed mesh and streaming mesh formats. Note that in practice we use a binary data format; the markup used here is explanatory. For clarity, the ordering of the triangles and vertices in each format is identical.**

#### ***4.1 Review of the format***

A traditional indexed mesh format stores all vertices in one array and all triangles in a second array which references the indices of the first. This is shown in the left column of Listing 1. The streaming mesh format we use, shown on the right column of Listing 1, is the same except for three key changes:

- 1. The vertices and triangles are interleaved into one stream.** Listing 1 shows this difference, as the two separate arrays are joined into one list.
- 2. The triangle indices are negative when the corresponding vertex is not used by any future triangles.** For example, in the first triangle in the right column of Listing 1, the first vertex is listed with a negative index as it will not be referenced by any of the triangles which follow. Similarly all indices of the last triangle are negative, as no vertices can be referenced after the last triangle in the stream. We refer to vertices that will no longer be referenced as ‘finalized’ vertices.

**3. The triangles use relative indices rather than absolute indices.** Index one means “the most recent vertex in the stream” rather than “the first vertex in the mesh.” For example in Listing 1 triangle  $\langle 1,2,3 \rangle$  in indexed mesh format becomes triangle  $\langle -3,2,1 \rangle$  in streaming mesh format, as the ordering is reversed when viewed relative to the third vertex instead of the first. Note that we are specifically using the “pre-order” streaming format identified by [14], meaning the vertices always precede all triangles which reference them; this is in contrast to the “post-order” format in which vertices always follow all triangles which reference them.

These three changes allow the mesh triangles to be understood from local information: we do not need the full vertex array in memory to understand an indexed triangle. In addition, they maintain the connectivity information and efficiency of the indexed mesh format, unlike a format which simply lists triangles by their vertex positions directly.

Although not mandated by the format, our streaming meshes are “triangle-compact,” meaning that triangles are positioned in the stream immediately after their last vertex. As a consequence of this ordering, one of the relative indices of each triangle is always 1 or -1, indicating the most recent vertex in the stream. Listing 1 is written in triangle-compact ordering, and this property may be verified by observing the triangle indices. Therefore, being triangle-compact means we do not need to store this one vertex’s index; for this vertex all we need is the single bit of finalization information. However, for code simplicity we have not taken advantage of this optimization yet.

For this format to be useful, most vertices should be finalized soon after being introduced to the stream, so that we need not keep them in memory. The set of vertices required to be in memory at any given time is called the “active set,” and the size of this set is called the “front width.” Given a fixed ordering of the vertices in the stream, Isenburg and Lindstrom [14] define the “triangle span” as the difference between the position of the triangle’s last vertex in the stream and its first vertex in the



stream. The “front span,” illustrated in Fig. 2, is the difference between the most recent vertex in the stream and the oldest vertex that is not yet finalized; this is an upper bound on the “front width” and will dictate the memory requirements in practice for our algorithms. In our triangle compact stream ordering, the maximum front span is also the maximum triangle span and maximum front width: no vertex further back in the stream than the maximum triangle span can be reached by future triangles.

Isenburg and Lindstrom [14] suggest two algorithms for processing a streaming mesh. Most generally, a hash table of vertices may be used, where elements are evicted from the hash table as they are finalized. Alternatively, a circular buffer, which is the length of the maximum front span, may be used: vertices are introduced on the end of the buffer; once a vertex must be overwritten because the buffer has wrapped all the way around, this can be done safely since the overwritten vertex is already outside of the active set. The circular buffer algorithm is more efficient for meshes with a low maximum front span, which is the case for our terrestrial surface reconstructions; so we present and implement our algorithms in the context of using a circular buffer. Note also that in this case explicit finalization is not needed, as eviction from the circular buffer guarantees finalization. Typically we process a vertex immediately before overwriting it, on the eviction step, because this guarantees (a) the vertex is finalized, and (b) all vertices are processed in the order in which they were placed in the stream. For meshes that cannot guarantee a low maximum front span, but which still have a low front width, it is possible to translate our algorithms to use the hash table method instead.

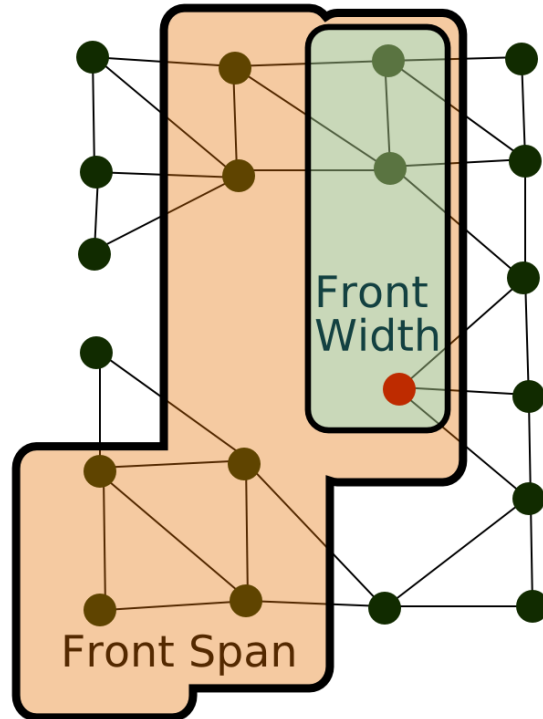


Figure 2. With meshes generated by Carlberg et al. [13], the front span is twice the maximum scan line length, which provides an upper bound on the front width. An example front span and width are shown relative the red vertex, assuming the vertices are ordered in vertical scan lines from left to right.

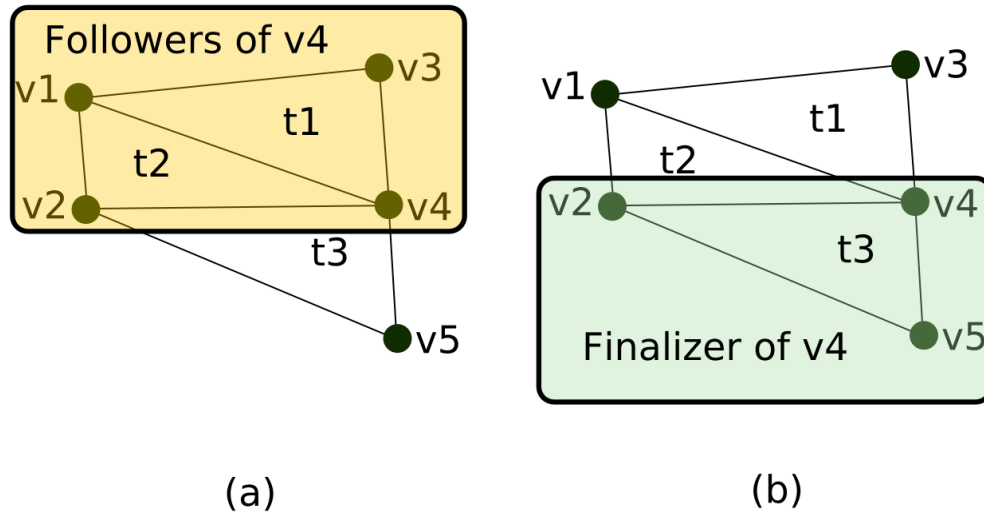
#### 4.2 Suitability of terrestrial reconstruction for streaming

Though the methods proposed in this paper could work with any streaming meshes, the surface reconstruction algorithm presented by Carlberg et al. [13] is especially well suited to stream processing. It provides streaming meshes with maximum front spans of less than one thousand vertices in practice for all tested data sets. To understand why this works, observe that the distance between the indices  $R$  and  $N$  of the algorithm described in Section 3 can not exceed *Search End* after the first search. After this search, the algorithm can then increment  $R$  to increase the triangle span, however, this should never happen more than an additional *Search End* times, as *Search End* defines the maximum length of a scan line. Therefore, the front span should never exceed  $2 \times \textit{Search End}$

vertices, or equivalently twice the maximum scan line length. In practice the span stays well below this value.

We expect a similar low maximum *front span* to be attainable for most surface reconstructions from terrestrial LIDAR, even when the algorithm described in Section 3 is not used; this is because building facades tend to occur in strips where a natural processing order is to simply follow the direction of data collection. We note two cases where this front span cannot be attained, however. First, we find it useful in some cases to post-process the surface reconstruction with a simple hole-filling algorithm: in this case, the new triangles are allowed to have an arbitrarily large vertex span. Second, although the surface reconstruction we used did not do so, it would be reasonable for a surface reconstruction algorithm to close loops during the triangulation process. For instance, if the buildings in a block form a continuous façade, and data is collected on all sides of the block, we would expect a continuous loop of façade. This may prevent a successful conversion to a low-span streaming format, as the end of the stream will need to connect to the beginning. Fortunately, for both the hole filling and closed loop cases, it will be acceptable for our purposes to simply mark the vertices references by any hole-filling or loop-closing triangles, and then not process these triangles in the streaming mesh. This solution is described and justified in Section 5.2.

As an alternative to this simple solution, a hash table may be used instead of a circular buffer, as mentioned in Section 4.1; as long as the large-span triangles are relatively few, the front width does not dramatically increase despite the increase in the front span. The mesh vertices and triangles can also be optimized for streaming, as discussed in greater detail by Isenburg and Lindstrom [14].



**Figure 3.** In this simple five vertex mesh, triangles t1 and t2 are followers of v4, because their last vertex in the stream is v4. Triangle t3 is the finalizer of v4, because no subsequent triangles reference v4.

### 4.3 Converting to a streaming mesh

Our implementation of the terrestrial surface reconstruction, described in Section 3, outputs an indexed mesh rather than directly generating a streaming mesh. Although it would be ideal to implement a surface reconstruction algorithm which directly generates a streaming mesh, it is a simple post-process to convert the indexed mesh output to a streaming mesh format, because the triangles already have low span under the current ordering. The algorithm only needs to interleave the vertex and triangle arrays into one stream.

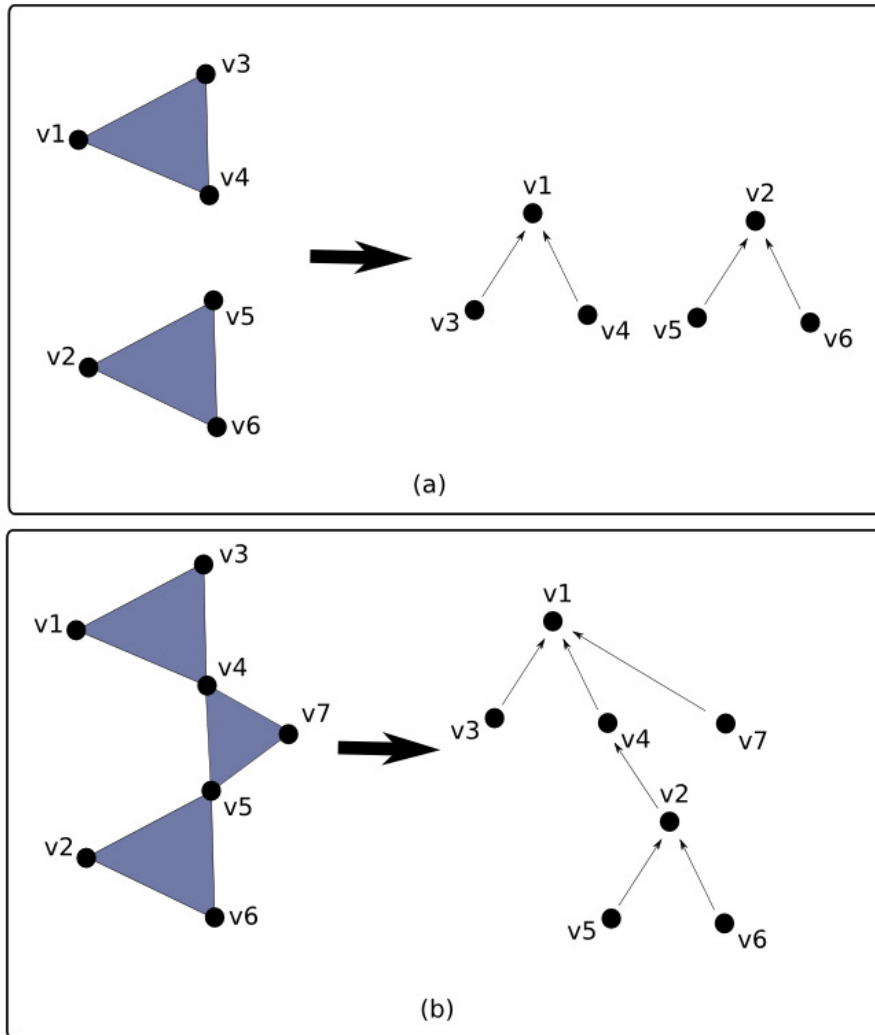
We introduce a simple algorithm to perform this conversion in a single pass using a circular buffer of *front span* vertices. This algorithm also guarantees our stream is pre-order and triangle-compact. In addition to its vertex, each element of the circular buffer will contain a reference to the last triangle to have included it, which we refer to as a ‘finalizer,’ and a reference to the list of triangles whose last vertex in the stream is this one, which we refer to as the ‘followers.’ Examples of followers and a

finalizer are shown in Fig. 3(a) and (b) respectively. For each triangle in the mesh, we perform the following steps:

First, we advance the circular buffer, overwriting any vertices as needed, so that the vertices of the new triangle may be placed in the buffer. Advancing requires the eviction of older vertices, which we write to the output file. On evicting a vertex, we first write the vertex itself, and then we write each triangle in its list of followers to the stream as well. When writing the vertex indices of each triangle, we negate the index to indicate finalization if the vertex's circular buffer element lists the triangle as the finalizer. To compute relative indices, we additionally record the total number of points written to the file in each circular buffer element, at the time the circular buffer element is created. The relative index is simply the difference between the current total number of points written and the total written when the vertex was originally placed in the circular buffer.

Second, we process the triangle by updating the appropriate list of followers in the circular buffer, to ensure the triangle will eventually be written to the output stream. To update the appropriate list of followers, we simply find the triangle's last vertex in the circular buffer, and add the triangle to that element's list.

Third, for each vertex of the triangle, we update the finalizer. To do this, we visit the vertex, and check its listed finalizer. If that finalizer triangle's last vertex is before the current triangle's last vertex in the circular buffer, then we change the vertex's finalizer to the current triangle, as the ordering in the circular buffer dictates which goes in the stream last.



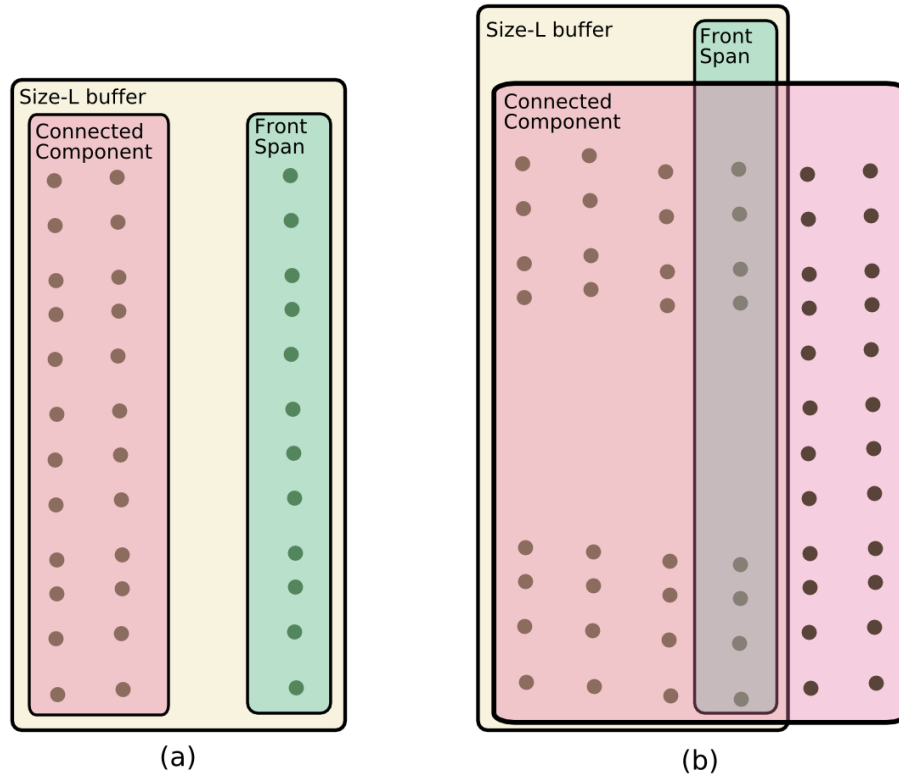
**Figure 4. The union find algorithm. (a) Triangle vertices become nodes in a forest of union find trees, with one tree per connected component. (b) Once triangle (v4,v5,v7) is added, two previously disconnected components are joined into one connected component.**

#### ***4.4 Computing non-local statistics on a streaming mesh***

It is often convenient to determine non-local information about the stream. For example, a simple estimate of the size of the connected mesh component to which a triangle belongs, or its bounding box, could help classify the triangle as noise, a small detail, or as a substantial surface. However, these statistics are difficult to obtain from a streaming format, because they cannot always be computed with only the local information that is kept in memory during stream processing. To

compute a statistic about a full connected component of the mesh, we must process the entire connected component, which could be arbitrarily large. While we could compute them before converting to a streaming mesh, we would ideally like to minimize our use of the non-streaming format or remove it entirely, as it requires significant out-of-core storage as the data size grows, and it is therefore highly inefficient to access. In addition, our non-streaming mesh implementation uses a simple out-of-core data structure, which does not work well with algorithms such as union find [15]. Union find is a standard algorithm for identifying a connected component in a graph and therefore a natural choice when computing statistics on connected components in a mesh. The union find algorithm works by forming a forest of trees on the nodes of the graph, shown in Fig 4(a). The root of each tree represents the whole component: nodes are in the same component if they share the same root. When two nodes are connected, we ensure they have the same root in the union find tree using a ‘union’ operator, which makes one of the nodes adopt the other’s root as its parent, an example of which is shown in Fig. 4(b). Our non-streaming out-of-core format breaks our vertex array into large chunks, and only keeps two or three chunks in memory at one time. Since the union-find algorithm chooses tree roots based on the rank of components, not on any locality criteria, the tree roots are arbitrarily distributed through the array; therefore, without a finer granularity of out-of-core storage, accessing them can cause significant thrashing. For these reasons, we introduce an alternative method for performing a union find and extracting statistics on connected components of a streaming mesh.

The goal of this algorithm is to compute statistics on the full connected component, such as the number of vertices in the component or its bounding box, without storing much of the stream in memory. We furthermore do not wish to re-order the points in the stream, because the original order is likely to have spatial locality, which helps avoid thrashing when performing the ground-air merge described in Section 5.



**Figure 5. Local and non-local statistics on connected components. (a) The connected component is entirely finalized when the last vertex must be evicted from the size-L buffer, so all statistics are known from local information. (b) The connected component is not finalized when its first vertex is evicted from the size-L buffer, so the statistics are not known from local information alone and require our two-pass algorithm.**

The core idea of this algorithm is to create an extra circular buffer of some small size  $L$  and to store vertices in this buffer as they are read from the stream. The goal of this buffer is to keep the vertices in memory until all vertices in their connected component have been finalized. In practice, most components are relatively small and will fit entirely within this buffer; therefore our statistics are fully computed by the time the first vertex of the component need be evicted from this buffer. This is the case illustrated in Fig. 5(a). However, for a few components, we must evict some vertices before their statistics are computed, as shown in Fig. 5(b). For these cases we perform an initial pass over the data, compute the statistics ahead of time, and store them in a lookup table. For any future traversal of the data set, we therefore have this non-local information available in the lookup table, while all locally-



computable statistics are recomputed per traversal. In all our experiments we choose the size  $L$  of the circular buffer to be  $10 \times \text{front span}$ , and we observe that our table of pre-computed statistics has a size less than 0.1% of the number of triangles in the ground mesh for all ground meshes in our test set.

The initial pass works as follows. First, vertices are read from the streaming mesh in to a size- $L$  circular buffer of union find objects. This defines the local portion of the stream in which we hope most of the connected components are resolved. Each vertex is given a union find tree node in the circular buffer; these are all initialized to root nodes, indicating that we have not yet found any other vertices in the same connected component. For each root node, the algorithm tracks the index of the newest vertex introduced to the corresponding connected component; initially this is the index of the root node itself. The algorithm processes the triangles of the stream in order, and advances the circular buffer to fit all the vertices of the most recent triangles. Triangles are processed by performing a union operation on the vertices, which causes all vertices of the triangle to share a single root node, ensuring they are all part of the same component. In addition to the standard union implementation, the union function incrementally combines the statistics computed on separate components into one set of statistics for the whole component.

Once a union find object must be evicted from the circular buffer, the algorithm must decide whether to place the component's statistics in the lookup table. Note, it only needs to make this decision if no previous vertices from this component were already evicted, as the decision must be the same for the whole component. To decide, the algorithm checks if the component is finalized: in other words, it checks whether the newest vertex in the component is within the current front span of the stream. It is only in this case that the statistics for the connected component cannot be computed locally using only the current circular buffer, so the algorithm must store the statistic in a lookup

table. If it cannot yet compute the statistic, instead it simply makes a note in the root union find node, storing a lookup table index where the algorithm should eventually store the computed statistic. It uses a simple increasing index – the first component to need an index gets 0, the next gets 1, and so on. Once the component is finalized, the algorithm will be able to compute the statistics and leave them in the requested index.

When performing the union operation on two nodes, we may merge what previously appeared to be two separate components into one larger component. It is possible that, while separate, the components have each already had their first eviction, and have therefore decided on different indices in the lookup table. In this case, the final statistics must be stored at both places in the table. Therefore, instead of storing a single table index per component, the algorithm stores a linked list of table indices. Since a destructive list concatenation takes only constant time, the union operation still runs in constant time. Once the newest vertex in the component is the vertex being evicted, the whole component is finalized and the algorithm traverses the list once and places a reference to the final computed statistics at each requested table index.

Once the first pass is finished, the algorithm has a lookup table for all the statistics that cannot be computed locally. During that first pass, this information was not available – the size of large components, for example, was unknown at the time of processing – so any algorithm requiring that information must make a second pass through the stream. During the second pass, the algorithm simply needs to process the data in the exact same way as the first pass. The only difference is that whenever the first pass would have made a note to store information in the lookup table, instead it simply pulls the information from the lookup table. Once a vertex is evicted from this processing, the algorithm therefore knows all non-local as well as local statistics.

Typically, non-local statistics are computed to aid a mesh processing algorithm, such as the merging algorithm of Section 5. Therefore, during our second pass over the data we will not just want to compute the statistics, but will also want to provide them to the processing algorithm which would make use of them. In order to send these statistics-laden vertices and the corresponding triangles to our mesh processing algorithm, we keep a queue of triangles that have been processed for the statistics computation but not yet output as part of the streaming mesh. Once the last vertex of a triangle is fully processed, the triangle may be pulled from the queue and sent to the processing algorithm. In order to ensure the vertices that are evicted from the size- $L$  circular buffer are still remembered by the triangles referencing them, until those triangles can be output to the stream, the vertices can either be stored in the triangle, or in an additional circular buffer of size  $2 \times \textit{triangle span}$ . Therefore, non-local statistics are computed in constant memory proportional to  $L + 2 \times \textit{triangle span}$ , plus the linear but very small size of the table.

Note that it is possible to optimize the computation of statistics on connected components if we know that these statistics will only be compared against a threshold; for example, in the algorithm described in Section 5, we will only check if a component is larger than some threshold value and will not use the exact size. This fact may be used to reduce the size of the table of non-local statistics, or to remove the need for a second pass entirely at the cost of increasing  $L$ . However, these optimizations were not explored, because computing the full table of non-local statistics accounted for less than 5% of the processing time in all our tests.



Figure 6. The dense terrestrial mesh is merged with a coarser airborne mesh.

## 5. Merging airborne and terrestrial data

Data from airborne LIDAR is typically much more sparse and noisy than terrestrial data. However, it covers areas that terrestrial 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. 6.

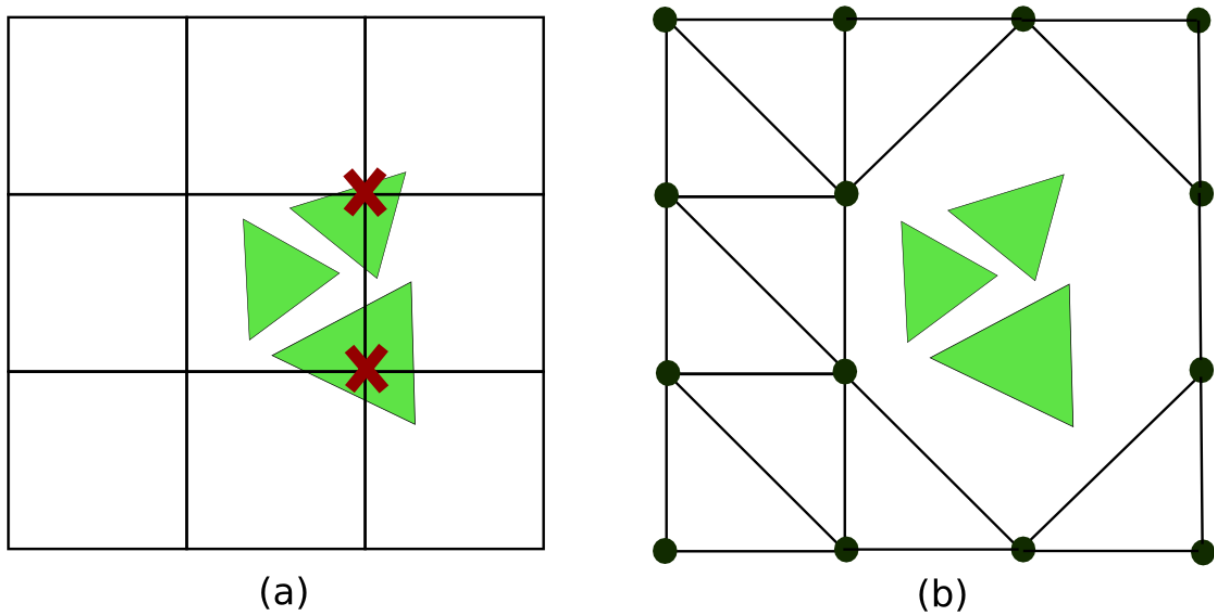
To accomplish this kind of merge, we present an algorithm to (1) create a height field from the airborne LIDAR, (2) triangulate that height field only in regions where no suitable ground data is found, and finally (3) fuse the airborne and terrestrial meshes into one complete model by finding and connecting neighboring boundary edges. By exploiting the locality of the streaming mesh format for the terrestrial triangulation, we perform this merge with only a constant amount of the ground mesh in

memory at any given time, plus the 0.1% for non-local statistics, and with linear memory use with respect to the air data. The algorithm is linear in time with respect to the sum of the size of the airborne point cloud and the size of the terrestrial mesh. This kind of merge is highly ambiguous near ground level, because of the presence of complex objects, such as cars and street signs. Therefore, we only merge with terrestrial mesh boundaries that are significantly higher than ground level, e.g. roofs, where geometry tends to be simpler.

### ***5.1. Creating and triangulating the height field***

We choose to model the airborne data with a simple height field. We do not attempt to use a streaming format for airborne surface reconstruction. Processing the ground mesh in a streaming order already dictates the order in which we must process the airborne mesh, and stream processing would not allow us to process the mesh in that order. We could alternatively stream the airborne data and not stream the terrestrial mesh, but the typically sparser airborne data and simpler 2.5D structure make it the natural choice for non-streaming. Storing the data in a regular grid makes the correspondence between the terrestrial mesh and the airborne data simply a constant-time query. To deal with the large size of the data, which does not all fit in memory, we divide the height map in to large rectangular portions, and keep least recently used portions on disk. Since the ground mesh stream tends to have good spatial locality, and not jump around in space too rapidly, we find having these large rectangular portions works well and does not tend to thrash.

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 height values and apply a median filter with a window size of 5 to reduce noise.



**Figure 7. Creating the height field. The green triangles represent the terrestrial mesh. (a) We mark problematic cells which are likely to conflict with terrestrial data. (b) The height field is regularly tessellated, skipping over the marked portions.**

We wish to create an airborne mesh that does not obscure or intersect features of the higher-resolution terrestrial mesh. We therefore mark those portions of the height field that are likely to be “problematic,” as in Fig. 7(a), and regularly tessellate the height field, skipping over the marked portions as in Fig. 7(b).

To mark problematic cells, we iterate through all triangles of the terrestrial 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 terrestrial triangle is close to the height field, it is likely that the two meshes represent the same surface; for example, points on the ground are often present in both the terrestrial and airborne meshes. Second, when the height of the terrestrial triangle is in-between the heights of adjacent height field cells, as in Fig. 8, the airborne mesh may slice through or occlude the terrestrial mesh details. In practice, this often happens on building facades. Unfortunately, our assumption that the terrestrial mesh is superior to the airborne mesh does not always hold: in particular on rooftops,

we tend to observe small amounts of floating triangles which cut the airborne mesh but do not contribute positively to the appearance of the roof. Therefore as a preprocessing step we use the union-find-based algorithm of Section 4.4 to compute the size of each connected component of the terrestrial mesh, and then remove components smaller than a threshold size from in the terrestrial mesh.

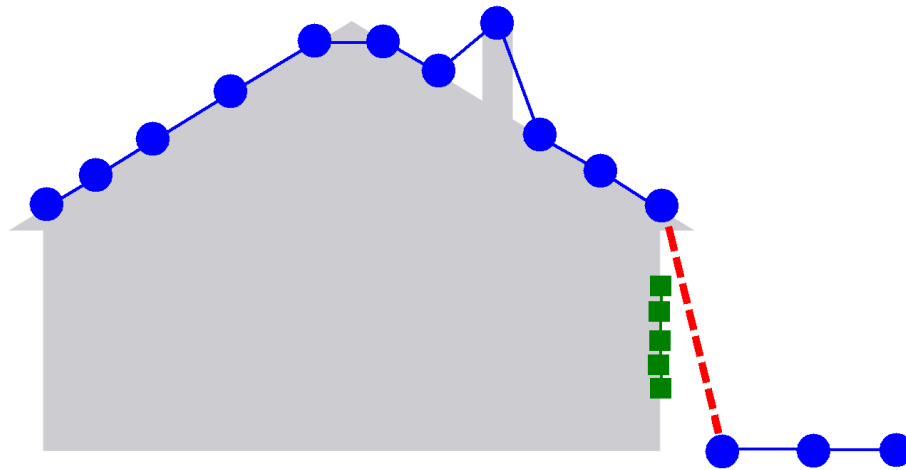


Figure 8. A side view of the height field, in blue circles, and the terrestrial data, in green squares. The dashed line connecting airborne points obscures the terrestrial data.

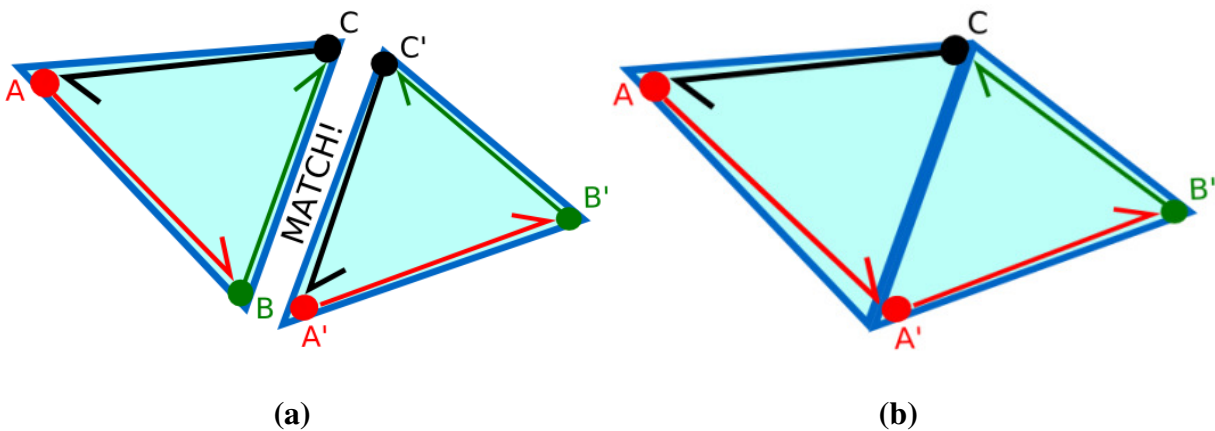


Figure 9. By removing the shared edges and re-linking the circular linked list, we obtain a list of boundary edges encompassing both triangles.

## 5.2. Finding boundary edges

Now that we have created disconnected terrestrial and airborne meshes, we wish to combine these meshes into a connected mesh. Fusing anywhere except the open boundaries of two meshes would create implausible geometry. Therefore, we first find these mesh boundaries in both the terrestrial and airborne meshes. We refer to the triangle edges on a mesh boundary as '*boundary edges*.' Boundary edges are identifiable as edges that are used in only one triangle. Note that these edges are often known at the time of mesh construction, and in these cases it is more efficient to mark them at that time than to rediscover them using the algorithm below. However, we do not wish to rely on a specific algorithm for the terrestrial surface reconstruction, and boundary edges are not marked in the streaming mesh format, so we use the following algorithm to efficiently find boundary edges. For consistency and ease of implementation, we use the equivalent algorithm for the airborne mesh.

We first find the boundary edges of the airborne mesh. Although we store the airborne mesh in an indexed mesh format, and the terrestrial mesh in a streaming format, in both cases we may think of vertices as being ordered by global, absolute integer indices and triangles as three global vertex indices. Any edge can be uniquely expressed by its two integer vertex indices. Since boundary edges are defined to be in only one triangle, our strategy for finding them is to iterate through all triangles and eliminate triangle edges that are shared between two triangles. All edges that remain after this elimination are boundary edges.

In detail, our algorithm for finding the boundaries of the air mesh is as follows: For each triangle, we perform the following steps. First, we create a circular, doubly-linked list with 3 nodes corresponding to the edges of the triangle. For example, the data structure associated with triangle ABC in Fig. 9(a) consists of three edge nodes, namely AB linked to BC, linked to CA, linked back to AB. Second, we iterate through these three edge nodes; in doing so, we either insert them in to a hash



table or, if an edge node from a previously-traversed triangle already exists in their spot in the hash table, we “pair them up” with that corresponding edge node. These two “paired up” edge nodes physically correspond to the exact same location in 3D space, but logically originate from two different triangles. Third, when we find such a pair, we remove the “paired up” edge nodes from the hash table and from their respective linked lists, and we merge these linked lists as shown in Fig. 9(b). After we traverse through all edges of all triangles, the hash table and linked lists both contain all boundary edge nodes of the full mesh.

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

To achieve this, we exploit the bounded front span of our streaming mesh, described in Section 4. Since no edge can reach a vertex once it has been finalized, the algorithm never encounters the same edge again after it has processed a full front span of the subsequent vertices. This locality attribute allows us to choose a front span-sized circular buffer of small hash tables as the edge-lookup structure for our terrestrial data. As we read the streaming terrestrial mesh, we place each circularly linked edge node of each triangle in to this circular buffer data structure. The lower vertex index of the corresponding edge is used as the index in to the circular buffer to retrieve a hash table of all edge

nodes that share that index. The higher index of the edge under consideration is then used as the key to the corresponding hash table. The value retrieved from this hash table is the circularly linked edge node. As with the hash table used in processing our air mesh, we check whether there is already a circularly-linked edge node with the same key existing in this hash table. Again as with the airborne mesh, if such an edge node is found, we know that more than one triangle must contain this edge, and it therefore does not correspond to a boundary edge. We can then remove the edge node and its pair from the hash table and merge their linked lists as with the airborne mesh.

Whenever the lowest index of a new edge is too large to fit in the circular buffer, we advance the circular buffer's starting index forward until the new index fits, clearing out all the existing hash tables over which we advance. The edge nodes of any hash that tables we clear out in performing this step must correspond to boundary edges, because we have removed all edges observed to be shared by multiple triangles in the traversal so far, and the locality attributes dictate that no future edges can use the vertices of this edge, because they are outside the range of the front span. These edge nodes may therefore be processed as described in Section 5.4 and freed from memory. This process makes it unnecessary to consider more than the maximum *front span* terrestrial edges at any one time. Note that since the front span is a small, bounded quantity in practice, this process results in a constant memory requirement with respect to the quantity of terrestrial data.

As mentioned in Section 4.2, any post-processing to fill holes or to close loops in the mesh may greatly increase the *front span* of our mesh. However, we note that almost all edges that are referenced by loop closing or hole filling triangles are not boundary edges. Because we are only interested in extracting boundary edges, these new triangles need not be processed. The vertices that they reference are marked with a Boolean value indicating that their incident edges are not boundary edges.

When using the terrestrial reconstruction generated by [13] as described in Section 3, three or more terrestrial triangles occasionally share a single edge. This is because the search for neighbor point  $N$  is not constrained to monotonically increase: after a triangle is formed with some neighbor point  $N$ , a new search can find a point  $N'$  that precedes  $N$ , and as triangles are extended from  $N'$  they may overlap the triangles associated with  $N$ . Assuming this happens rarely, we can recognize these cases and avoid any substantial problems by disregarding the excess triangles involved.

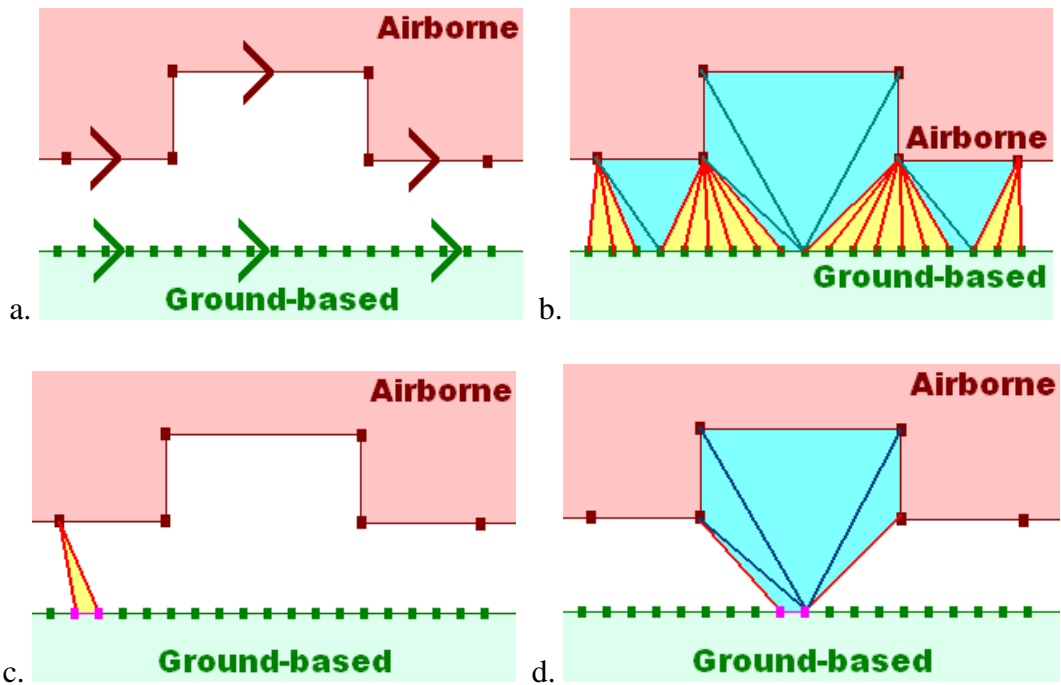


Figure 10. Adjacent terrestrial and airborne meshes are merged. (a) Airborne and terrestrial edges are close enough to be merged. (b) The result of the merge. (c) When both vertices of the terrestrial edge have the same closest vertex in the airborne mesh, one triangle is formed. (d) When the vertices of the terrestrial edge have different nearest neighbors, multiple triangles are formed.

### 5.3. Merging boundary edges

The merge step occurs incrementally as we find boundary edges in the terrestrial mesh, as described in Section 5.2. Given the boundary edges of our airborne mesh and a single boundary edge of the terrestrial mesh, we fuse the terrestrial based boundary edge to the nearest airborne boundary edges, as

shown in Figs. 10(a) and 10(b). As a preprocessing step, before finding the boundary edges of the terrestrial mesh, we sort the airborne boundary edge nodes in to the 2D grid of the height map to facilitate fast spatial queries. For both vertices of the given terrestrial boundary edge, we find the closest airborne vertex from the grid of airborne boundary edges. When the closest airborne vertex is closer than a pre-defined distance threshold, we can triangulate. If the two terrestrial vertices on a boundary edge share the same closest airborne vertex, we form a single triangle as shown in Fig. 10(c). However, if the two terrestrial vertices find different closest airborne vertices, we perform a search from one of the two vertices to find the other through the circular list of airborne boundary edges. If the number of boundary edges we must traverse to find the other vertex is below some threshold, we create the merge triangles that are shown in blue in Fig. 10(d).

Since objects close to ground level tend to have complex geometry, there is significant ambiguity in deciding which boundary edges, if any, should be merged to the airborne mesh. For example, the boundary edges on the base of a car should not be merged with the airborne mesh. However, boundary edges near the base of a curb should be merged to the airborne mesh. Without high level semantic information, it is difficult to distinguish between these two cases. Additionally, differences in color data between airborne and terrestrial sensors create artificial color discontinuities in our models, especially at the ground level. Therefore, we only create merge geometry along edges that are a fixed threshold height above ground level, thus avoiding the merge of ground level airborne triangle with ground level terrestrial triangles. This tends to limit our merges to mesh boundaries that have simple geometry and align with natural color discontinuities, such as the boundary between a building façade and its roof. Ground level is estimated by taking minimum height values from a sparse grid as described in Section 5.1. To further improve quality of merges, we do not merge with small patches in the terrestrial triangulation with less than 200 vertices, computed as a non-local statistic as

described in Section 4, or with loops of less than 20 airborne boundary edges. This avoids merges of difficult, noisy geometry.

#### ***5.4. Merging results***

Our algorithm is tested on a 64 bit 2.66 GHz Intel Xeon CPU with 4 GB of RAM. The results for five different point clouds from S1, S2, and S3 are shown in Table 2. Fig. 6 shows the fused mesh from point cloud 2, and Fig. 11 shows the fused mesh from point cloud 4. Table 2 reports run times for all point clouds, including a break down of the time taken by each processing stage. Table 3 reports the sizes of the input and output data.

The preprocess stage, ‘convert to streaming mesh format,’ converts the non-streaming, indexed mesh, generated as described in Section 3, to a streaming format as described in Section 4.3. From the S1 and S2 data sets, we see the cost of converting to a streaming format is not insignificant, but we note that for the full S1 and S2 data sets the total processing time, with this preprocess time added, is still less than that of previous versions of this algorithm that did not use a streaming mesh: the previously published results of [13] took 8392 seconds to process point cloud 2 compared to the total of 8167 seconds reported here, while point cloud 4 took 2518 seconds compared to the 1262 seconds reported here.

Note that the S3 data set was divided into rectangular regions by the vendor who collected the data, apparently for their post processing. We were unable to re-sort the points to correct these artificial divisions, which cause our mesh to be disconnected along the region boundaries when using the fast algorithm of [13] to generate a terrestrial mesh as described in Section 3. In effect this means our input to ground triangulation and the pre-processing stage was a set of much smaller, separate portions of the scene, that did not require the out-of-core storage that a contiguous input would have required. This artificially reduced the preprocessing time; therefore it does not make sense to report a

preprocess timing for point cloud 5, which is the only point cloud from the S3 data set. Fortunately these issues do not significantly affect the timings of all stages beyond the preprocess stage, since the stream processing memory requirement is the same for the disconnected mesh as it would be for a contiguous mesh. We can see this by noting that the memory requirement is dictated by the maximum scan line length, which should not change significantly due to a small fraction of the scan lines being broken across a region boundary.

**The first processing stage**, ‘Make non-local table,’ is the stage in which the non-local statistics of the streaming mesh are pre-computed. This is where the primary processing cost of the technique described in Section 4.4 is incurred, which is consistently a small percentage of the total processing time. This cost is close to the time required simply to read the ground based triangles from disk.

**The second stage**, ‘Make height map,’ includes reading the airborne data points from disk twice, once for the bounding box and once to generate the height map. This is proportional to both the number of air points, and the size of the height map; in addition, it is affected by the spatial locality of the airborne data set. We note that point cloud 5 performs surprisingly poorly compared to the similarly-large point cloud 2, because it has more airborne points with worse spatial locality and therefore spends significantly more time swapping portions of its height map to and from disk. This stage also accounts for a surprising 51% of the processing time for point cloud 1, because of the large number of airborne points relative to the height map size in that data set.

**The third stage**, ‘Process height map,’ includes hole filling and median filtering. We see it appears to scale quite poorly with the size of the height map. However, the main cost of this stage occurs due to (a) the inefficient hole filling algorithm, which performs expensive searches in regions with no data, and does not exploit coherence that would allow it to avoid much of this search, and (b) the simplistic median filter that temporarily duplicates the entire height map, causing additional strain

on the memory and disk use. These could both be improved substantially, but are not a central focus of this work and so have been left relatively inefficient.

**The fourth stage**, ‘Mark bad cells,’ traverses the terrestrial mesh and marks areas of the height map where conflicts are likely. This requires a traversal of the terrestrial mesh and simultaneous traversal of the airborne mesh in the terrestrial order, and so its cost is proportional to the size of both data sets. In addition the airborne mesh must be traversed in the order of the terrestrial mesh data, which requires more swapping of the airborne height map data to and from disk than other stages which can process the airborne data in the most convenient order instead. Accordingly, this is one of the most expensive stages.

**The fifth stage**, ‘Make air mesh,’ is the process of translating the height map to an indexed mesh format. This requires only a straightforward traversal of the height map, and the generation of an airborne mesh structure.

**The sixth stage**, ‘merge meshes,’ is the final merging phase. It requires a traversal of both terrestrial and airborne structures simultaneously, as with the ‘mark bad cells’ stage, however it only requires boundary edges of the terrestrial mesh be mapped to the airborne mesh and therefore costs a proportional-but-substantially-smaller amount.

Time required to use the result, for example to output it to disk or to render it, is not included.

Overall, we see that processing time increases significantly as the memory requirement pushes us to swap more to disk, and that the time and space requirements of each stage of the algorithm grows with a different set of criteria. The diversity of these criteria make it difficult to cleanly summarize the performance of the algorithm, but we note that we are able to process some of the largest urban data sets reported in literature in just a few hours. Specifically, while previous work reports on merging about 16 thousand airborne triangles with about 27 million terrestrial ones [4], in this paper

we have reported merges of about 200 million terrestrial triangles with 14 million airborne triangles, and 146 million terrestrial triangles with 22 million airborne triangles. Thus, we have developed an algorithm capable of handling 1300 times as much airborne data and 7 times as many terrestrial triangles as in [4].

Point Cloud	(preprocess) Convert to stream format (s)	(first) Make non-local table (s)	(second) Make Height Map (s)	(third) Process Height Map (s)	(fourth) Mark Bad Cells (s)	(fifth) Make Air Mesh (s)	(sixth) Merge mesh (s)	Total processing time (s)	Total preprocess and process time (s)
1 (S1)	282	6 2%	154 51%	8 3%	83 28%	3 1%	43 14%	297 100%	579
2 (S1)	1233	189 3%	411 6%	1610 23%	2631 38%	496 7%	1597 23%	6934 100%	8167
3 (S2)	29	7 2%	2 0%	5 2%	235 78%	2 0%	48 16%	300 100%	329
4 (S2)	282	36 4%	8 1%	35 4%	530 54%	16 2%	354 36%	980 100%	1262
5 (S3)	n/a	257 3%	1380 15%	1518 17%	3737 41%	382 4%	1891 21%	9165 100%	n/a

**Table 2: Merge results for point clouds 1 through 5, listing the processing time for each phase of the algorithm. Percentages listed indicate percentage of the processing time, and do not reflect preprocess time.**

Point Cloud	Terrestrial Tris	# Air Pts	Height Map Size	Airborne Tris	# Merge Tris
1	5 M	17 M	436 × 429	245 K	33 K
2	146 M	32 M	4729 × 6151	22 M	903 K
3	6 M	9 K	215 × 217	79 K	19 K
4	32 M	354 K	1682 × 1203	4 M	92 K
5	200 M	129 M	4390 × 5990	14 M	889 K

**Table 3: Merge results for point clouds 1 through 5, listing the size of the input and output data for each point cloud.**





**Figure 11. A merged model, with vertices from airborne data colored white.**

## **6. Conclusions and future work**

We have shown a technique for merging meshes from high resolution terrestrial sensors with lower-resolution airborne data. By exploiting locality in the terrestrial mesh, we achieve fast merges of very large quantities of data. By defining these locality properties in terms of a streaming mesh format, we generalize the mesh-merge algorithm to work with a broad class of meshes; we do not rely on the specific details of the terrestrial surface reconstruction.

Our technique does over-triangulate if there is detailed terrestrial geometry near an airborne mesh boundary. This occurs because multiple terrestrial boundaries may be close enough to the airborne mesh to be fused with it, creating conflicting merge geometry. This is a result of the fixed threshold we use to determine when merge triangles should be created, and a different heuristic may be used to obtain different artifacts. However, the underlying ambiguities in triangulation cannot be properly

solved without higher level shape analysis. A promising area of avenue of future work is therefore to classify points in the scene according to object type, and only perform merges when it is logically consistent to do so; for example, to only merge ground triangles with other ground triangles.

As an additional direction for future work, we note that this work focused only on connecting and interpolating actual data points. However, to generate appealing models, much could be done to improve the results visually by deviating from this data. For example, color could be adjusted to be more self-consistent across the terrestrial and airborne data sets, as often color varies arbitrarily in the actual data due to the changing time of day, changing sensors, and changing viewing angles. Similarly, the input of multiple sensors could be used to detect and remove ‘ghost’ data which occurs when a LIDAR scanner captures a portion of a moving object.

Finally, we note that the height field interpretation of the airborne data is not always optimal, as, especially in areas of dense airborne coverage from multiple sensor passes, scan points can often capture geometry which is not possible to represent in a height field, such as points on a building façade. A system designed to merge fully 3D airborne data where necessary, while still taking advantage of the often-2.5D nature of the rest of the airborne data, would be ideal.

## **7. Acknowledgements**

This work is in part supported with funding from the Defense Advanced Research Projects Agency (DARPA) under the Urban Reasoning and Geospatial Exploitation Technology (URGENT) Program. This work is being performed under National Geospatial-Intelligence Agency (NGA) Contract Number HM1582-07-C-0018, which is entitled, ‘Object Recognition via Brain-Inspired Technology

(ORBIT)'. The ideas expressed herein are those of the authors, and are not necessarily endorsed by either DARPA or NGA. This material is approved for public release; distribution is unlimited. Distribution Statement "A" (Approved for Public Release, Distribution Unlimited). This work is also supported in part by the Air Force Office of Scientific Research under Contract Number FA9550-08-1-0168.

## 8. References

- [1] N. Amenta, S. Choi, T. K. Dey and N. Leekha. A simple algorithm for homeomorphic surface reconstruction. *Symp. On Comp. Geometry*, pp. 213-222, 2000.
- [2] B. Curless and M. Levoy. A volumetric method for building complex models from range images. *SIGGRAPH 1996*, pp. 303-312, 1996.
- [3] H. Edelsbrunner. Surface reconstruction by wrapping finite sets in space. Technical Report 96-001, Raindrop Geomagic, Inc., 1996.
- [4] C. Frueh and A. Zakhor. Constructing 3D city models by merging ground-based and airborne Views. *Computer Graphics and Applications*, pp. 52-61, 2003.
- [5] M. Garland, A. Willmott, and P. Heckbert. Hierarchical face clustering on polygonal surfaces. *Symp. on Interactive 3D Graphics*. pp 49-58, 2001.
- [6] M. Gopi and S. Krishnan. A fast and efficient projection-based approach for surface reconstruction. *SIBGRAPI 2002*, pp.179-186, 2002.
- [7] M. Isenburg, Y. Liu, J. Shewchuk, and J. Snoeyink. Streaming computation of Delaunay triangulations. *SIGGRAPH 2006*, pp 1049-1056, 2006.

- [8] A. Mangan and R. Whitaker. Partitioning 3D surface meshes using watershed segmentation. *IEEE Trans. on Visualization and Computer Graphics*, 5(4), pp 308-321, 1999.
- [9] R. Pito. Mesh integration based on co-measurements. *IEEE Int. Conf. on Image Processing*, vol. II pp. 397-400, 1996.
- [10] G. Turk and M. Levoy. Zippered polygon meshes from range images. *SIGGRAPH 1994*, pp. 311-318, 1994.
- [11] Y. Yu, A. Ferencz, and J. Malik. Extracting objects from range and radiance images. *IEEE Trans. on Visualization and Computer Graphics*, 7(4), pp. 351-364, 2001.
- [12] H. Zhao and R. Shibasaki. Reconstructing textured CAD model of urban environment using vehicle-borne laser range scanners and line cameras. *Int'l Workshop on Computer Vision Systems*, pp. 284-297, 2001.
- [13] M. Carlberg, J. Andrews, P. Gao and A. Zakhor. Fast Surface Reconstruction and Segmentation with Ground-Based and Airborne LIDAR Range Data. *3DPVT 2008*, Atlanta, Georgia, June 2008.
- [14] M. Isenburg and P. Lindstrom. Streaming Meshes. *Visualization'05*, pp. 231-238, 2005.
- [15] M. A. Weiss. *Data Structures and Problem Solving Using Java*, 3rd edition, Boston: Addison Wesley, 2007, pp. 842-848, 2007.
- [16] C. Früh, S. Jain, and A. Zakhor. Data Processing Algorithms for Generating Textured 3D Building Facade Meshes from Laser Scans and Camera Images. *International Journal of Computer Vision*, 61(2), pp. 159-184, 2005.
- [17] C. Früh and A. Zakhor. An Automated Method for Large-Scale, Ground-Based City Model Acquisition. *International Journal of Computer Vision*. 60(1), pp. 5 – 24, 2004.
- [18] <http://www-video.eecs.berkeley.edu/~frueh/3d/>