Visually-Complete Aerial LiDAR Point Cloud Rendering

Zhenzhen Gao
University of Southern California
3737 Watt Way, PHE 108
Los Angeles, CA 90089
zhenzheg@usc.edu

Luciano Nocera
University of Southern California
3737 Watt Way, PHE 108
Los Angeles, CA 90089
nocera@usc.edu

Ulrich Neumann
University of Southern California
3737 Watt Way, PHE 108
Los Angeles, CA 90089
uneumann@usc.edu

ABSTRACT

Aerial LiDAR (Light Detection and Ranging) point clouds are gathered by a downward scanning laser on a low-flying aircraft. Due to the imaging process, vertical surface features such as building walls, and ground areas under tree canopies are totally or partially occluded, resulting in gaps and sparsely sampled areas. These gaps produce unwanted holes and uneven point distributions that often produce artifacts when visualized using point-based rendering (PBR) techniques. We show how to extend PBR by inferring the physical nature of LiDAR points for visual realism and added comprehension. More specifically, the class of object a point is related to augments the point cloud in pre-processing and/or adapts the online rendering, to produce visualizations that are more complete and realistic. We provide examples of point cloud augmentation for building walls and ground areas under tree canopies. We show how different types of procedurally generated geometry can be used to recover building walls. These methods are generic and can be applied to any aerial LiDAR data set with buildings and trees. Our work also incorporates an out-of-core strategy for hierarchical data management and GPU-accelerated PBR with extended deferred shading. The combined system provides interactive visually-complete rendering of virtually unlimited-size LiDAR point clouds. Experimental results show that our rendering approach adds only a slight overhead to PBR and provides comparable visual cues to visualizations generated by off-line pre-computation of 3D polygonal urban models.

Categories and Subject Descriptors

I.3.3 [**Computer Graphics**]: Picture/Image Generation; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism

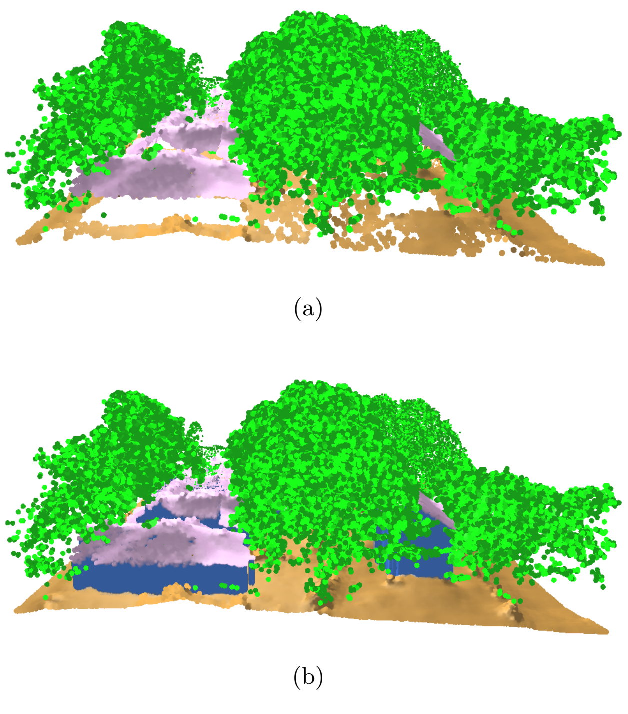
General Terms

Algorithms

Keywords

aerial LiDAR, 2.5D, point cloud, point-based rendering, PBR, procedural geometry, visually-complete, GPU

1. INTRODUCTION


(a)

(b)

Figure 1: Shaded views of a point cloud color coded by classes. Green trees, brown ground and pink buildings. (a) Using basic PBR, white holes are visible where wall points and ground points under trees are missing. (b) Using our extended PBR with added blue point-walls.

Light Detection and Ranging (LiDAR) is a terrain and urban information acquisition technique based on laser technology. LiDAR data sets consist of 3D points without connectivity that can be complemented by intensity, RGB color values and classification labels. Aerial LiDAR is gathered by mounting a downward-scanning laser on a low-flying aircraft while terrestrial LiDAR is gathered by mounting a portable scanner on a tripod and scanning horizontally. The use of aerial LiDAR data for urban modeling [30, 33] and visualization [20, 22] has received increased attention in recent years, primarily due to its use in conjunction with Global Positioning System technology [27]. The interactive visualization of LiDAR data sets is relevant to a number of areas including quality control, security, land management and urban planning [10, 13, 27].

This paper is primarily concerned with the visualization of aerial LiDAR point cloud data. Importantly, aerial LiDAR data can be characterized as being 2.5D in nature, in that the sensor is only able to capture dense details of the surfaces facing it [34]. Only a few points typically appear on vertical surfaces such as building walls and such areas are either completely occluded or severely under-sampled, as is the case for ground areas under tree canopies. To provide useful point-based visualizations of this type of data, one needs to fill in or augment missing or under-sampled areas that often produce artifacts like holes as in Figure 1(a). Our work focuses on augmenting scenes with buildings and trees, as these objects are predominant in urban areas where the occlusions and under-sampling is most severe.

We extend the PBR framework to augment aerial LiDAR point cloud in the offline pre-processing and/or procedurally generate geometries in the online GPU-accelerated rendering for more complete and realistic visualizations, by informing the system with cues relating to the physical properties of the scanned points, inferred from readily available classification information of point clouds.

The *2.5D characteristic of building structures* is formally observed and defined in [34], as "building structures being composed of detailed roofs and vertical walls connecting roof layers". Inspired by exploitation of this characteristic in building reconstruction [23, 26, 30, 31, 34, 35], we introduce the characteristic to the visualization problem. To reconstruct building walls, for any building point on building boundaries, we attach a piece of vertical wall connecting to the ground. Three distinct geometries are explored to use as wall primitive: point, quadrangle and axis-aligned rectangular parallelepiped (ARP). Point-wall is produced by augmenting the point cloud in the pre-processing. Quadrangle-wall and ARP-wall are procedurally generated in the rendering.

For under-sampled ground areas under tree canopies, we augment the point cloud with occluded ground points directly under tree points in the pre-processing.

The large size of aerial LiDAR datasets usually exceeds the memory capacity of a computer system. Our framework also applies an *out-of-core* scheme that processes only a manageable subset of the data at any one time. For performance consideration, it is coupled with a quadtree as the multi-resolution data structure to provide view-dependent *level-of-detail* (LOD). Objects near the viewer are displayed with more detail than those that are farther away.

Contributions: To the best of our knowledge, we are the first to utilize the 2.5D characteristic of building structures to improve visualization.

- We show that the demonstrated framework successfully fills the gaps between ground and roofs as well as under-sampled or missing points under tree canopies, thus providing comparable visual cues to visualizations generated by off-line pre-computation of 3D polygonal urban models. Our method has low overhead and achieves interactive frame-rates.

- We introduce three distinct primitives to reconstruct building walls.

- We demonstrate how procedural geometry can be implemented at the rendering level of a PBR framework.

- We extend the deferred shading of PBR to handle both point splats and polygonal geometries.

The rest of the paper is organized as follows. Section 2 summarizes the related literature. A brief overview of the framework is given in Section 3 followed by implementation details. Section 4 introduces data structures and the out-of-core strategy. Section 5 explains the operations that take place during the pre-processing stage while Section 6 details the operations carried out in the rendering stage. Finally, we present experimental results in Section 7 and conclusions in Section 8.

2. RELATED WORK

LiDAR point clouds are often visualized by using triangles, points or splats.

2.1 Triangles

Many Geographical Information Systems such as ArcGIS [5] and GRASS GIS [3], and a majority of the systems listed in [14] support triangular irregular network (TIN) rendering of LiDAR data. Because a TIN is generated by directly triangulating point clouds, missing or under-sampled areas are recovered by the space-filling nature of triangle meshes. TIN visualizations also provide the user with important cues such as surface orientation, lighting and occlusion. However, TIN visualizations have known drawbacks. (i) Triangulated surfaces emphasize the noisy nature of LiDAR points so these visualizations are often visually misleading. (ii) TIN methods are unsuited to vegetation since they are not well-modeled using continuous surfaces. (iii) Triangulation is compute-intensive due to the lack of connectivity between points in raw sensor data. (iv) In the case of very dense point clouds, triangles often cover only one or a partial pixel, making the rendering very inefficient and highly prone to aliasing.

Alternatives to TIN visualizations are the 3D model extraction methods [17, 35, 36], where a point cloud is transformed into a collection of approximating polygonal surfaces. These methods have the advantage of providing data reduction for large urban scenes, but they commonly discard points for objects and details that are not modeled such as fine architectural details, vegetation, street lights, and power cables.

2.2 Points

To overcome the disadvantages of triangle-based visualizations, there are methods for directly displaying points. Points without connectivity offer unaltered visualizations of scanner data. Recent systems such as Exelis ENVI [2], Exelis E3De [1], LViz [4], and most software listed in [14] allow direct rendering of raw 3D points using simple primitives like pixels, squares or smoothed circles. These primitives are fix-sized, non-oriented and non-blended, thus the implementations are fast, but often result in aliased images with holes lacking important 3D cues.

2.3 Splats

With the advance of programmable shaders, PBR techniques that represent points as oriented discs, commonly referred to as *splats*, are now practical for interactive use. A comprehensive survey of PBR is found in [19]. The book [15] offers more detailed comparisons of various PBR algorithms. In the area of LiDAR visualization, the work of [20] applies a two-pass rendering to efficiently blend splats and the system of [22] combines splatting with deferred shading [11] in a three-pass rendering algorithm.

Overlapped oriented splats can create a visually-continuous surface without the need for costly triangulation to form surfaces among points. Terrain and rooftops are well suited to PBR because dense splats form good approximations of continuous surfaces. Rendering vegetation as splats also produces a more realistic image than the TIN representation because branches, stems and leaves are represented without the artifacts of surfaces created by triangulation. However, an important drawback of splatting appears in areas with sparse or no points due to occlusions (facades and areas under tree canopies). These areas appear as holes, often impairing the comprehension of the rendered image as illustrated by Figure 1(a). Our methods address this problem by extending the PBR with deferred shading [11] framework which achieves the best trade-off between performance and rendering quality [15].

3. FRAMEWORK OVERVIEW

An overview of the framework is presented in Figure 2 where the yellow path represents a basic PBR pipeline. Our framework consists of two parts: an offline pre-process and an online GPU-accelerated rendering. Pre-processing performs one time operations that are needed to augment the point cloud and to set-up rendering. The rendering stage provides the standard rendering passes (detailed in Section 6) as well as functions to procedurally generate wall geometries where needed. Users can select from three geometries (point, quadrangle and ARP) to reconstruct walls. The workflow varies depending on the choice of geometry, as shown in Figure 2.

The input is a 3D LiDAR point cloud, as produced by most scanners. The preliminary classifier attaches labels to each point for three primary classes of ground, tree and building. The point cloud is then augmented by adding missing ground points under trees. Next, building points that lie on the perimeter of rooftops are labeled as boundary points. The positions of such points are later used to anchor building walls. After this step, we have four valid classes: ground, tree, building and building boundary. Normal vectors are then computed for all points, this is the last common step shared by three wall reconstructions as follows.

Point-wall After the additional step to compute normals for the wall surfaces, point-walls are created by adding points connecting the building boundaries and the ground. In the rendering stage, augmented points are rendered as splats, like all other points, using the standard three PBR passes.

Quadrangle-wall With the information acquired from the wall normal estimation step, all points are sent to the rendering. After the visibility pass that renders points of all classes as splats, boundary points are processed to create walls by generating a quadrangle for each boundary point.

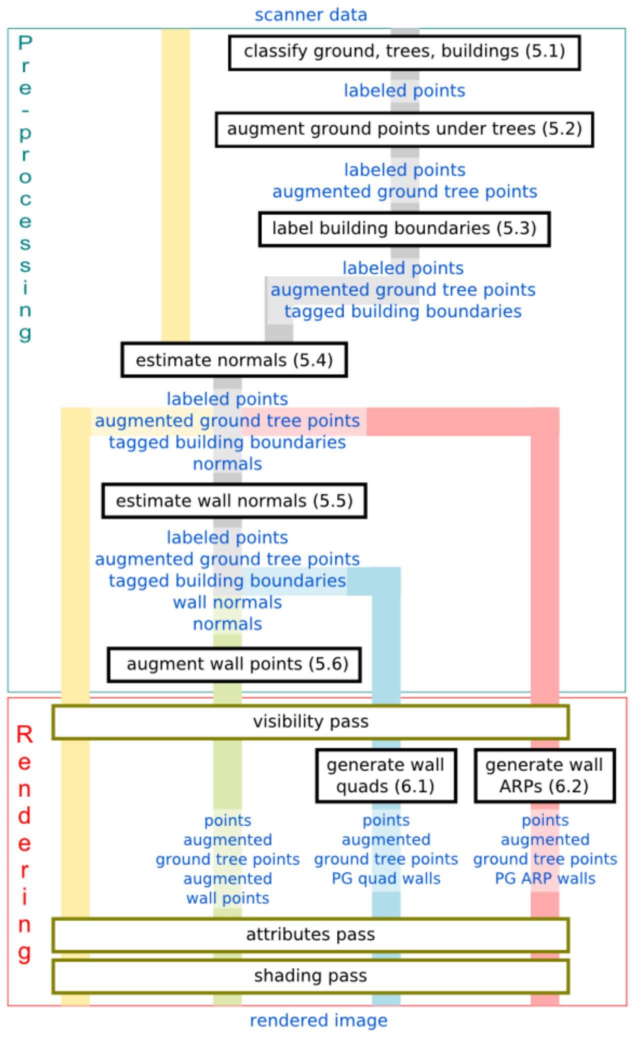


Figure 2: Pre-processing and rendering for basic PBR (yellow), and for our completion methods: augmented wall points (green), procedurally generated quadrangle walls (blue) and procedurally generated ARP walls (red). Corresponding section numbers are listed in parentheses.

Both quadrangles and splats are sent to the attribute pass to accumulate per-pixel normals and colors. And finally the shading pass provides the Phong shading [29].

ARP-wall In many cases it is appropriate to consider the vertical wall primitive as axis-aligned, so the step to estimate wall normals is skipped. The remaining rendering steps are similar to the quadrangle case.

Note that the view-dependent PBR processes will potentially render point splats and quads surfaces each frame.

4. DATA MANAGEMENT

A LiDAR data set is often too large to load and process in main memory. A common solution is to partition the data into several manageable *subsets* and to process these subsets in sequence in an out-of-core manner [20, 21, 22]. Our system utilizes a quadtree hierarchy to manage subsets as [20]. The quadtree construction is inexpensive and the domain of quadtrees is effectively 2.5D, which is a good match for aerial LiDAR. The generated structure is light-weight so it can reside in main memory and be efficiently traversed.

We predefine a maximum number of points to be stored in each quadtree node and the input LiDAR point cloud is spatially partitioned as a hierarchical quadtree. The root node covers the whole area and each interior node contains its bounding box and four pointers to its children. Leaf nodes contain the bounding box of each subset of the original point cloud. The quadtree of [20] only stores index information of points that belong to the bounding box in a leaf node. It does not apply to our case where indices are dynamic as the input point cloud is augmented. We write the point data of each leaf node in sequence to storage as a single file and attach index information in the file to corresponding leaf nodes. Although this takes additional time for the one-time file write operations, our per-frame read operations are faster than that of [20]. Because our accesses are spatially coherent, but read operations of [20] extract sparsely located points from the large original input data file.

The quadtree is also used as the multi-resolution structure for managing LOD during rendering. Hierarchical traversal of the bounding boxes enables efficient visibility testing for the rendering stage. The LOD implementation of [20] loads points of a visible leaf node into multiple OpenGL Vertex Buffer Objects [8] (VBOs) and distributes VBOs uniformly among levels of the quadtree. An interior node does not have complete data until the collection of VBOs from all descendants are accessed. Our multi-resolution method does not propagate VBOs from leaf nodes to interior nodes. Rather, for a visible leaf node, given a current camera position and the user-defined LOD factor f, we compute the number of screen pixels n by projecting the bounding box of the leaf node onto the screen. We then randomly select $f \times n$ points from the leaf node, load them into VBOs and send them directly to the rendering pipeline. Since n decreases as the distance between the leaf node and the camera increases, the LOD image is created by displaying more points near the camera and fewer points for areas far from the camera. To avoid frequent points selection and update, we define an interval around n and reload points only when the current calculated n' is beyond the interval defined by the previous n. In the experiments, we use $[\frac{n}{\sqrt{2}}, n\sqrt{2}]$ as the interval.

When processing a subset of points, a k-d tree is built for each class of points to enable rapid in-class neighborhood selection. To facilitate rapid between-class neighborhood selection, the k-d trees are partitioned on the x-y plane to enable fast accessing of all points with similar (x, y) values. Given a point (P_x, P_y, P_z) in class A, its neighbors in class B are located by seeking points near (P_x, P_y) in B's k-d tree regardless of the difference in z values.

5. PRE-PROCESSING

Pre-processing handles operations that only need one-time offline computations.

5.1 Classify Ground, Trees and Buildings

The input data are assumed in LAS [24] format. The LAS standard defines several standard LiDAR classes including ground, building, water, low vegetation, etc. Our test data and many other LiDAR point clouds come with classification information. In cases where classification tags are not available, third-party software such as LAStools [18] or those listed in [14] can be used to generate class information.

Our framework requires points in classes of ground, building, and all levels of vegetation. For simplicity, we treat all vegetation as tree points regardless of their sub-classes.

5.2 Augment Ground Points Under Trees

Trees do not possess the 2.5D characteristic, so laser scans often include both canopy points and ground points under the tree crown. However, sampling of the ground under the tree canopy is often sparse as in Figure 3(a). To provide a hole-free ground visualization, we augment the point cloud with new ground points in under-sampled areas.

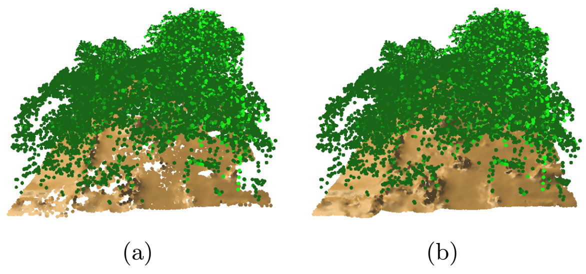


Figure 3: The demonstration of augmenting ground points under trees. Tree points are rendered in green and ground points in brown. (a) Before. (b) After.

Given an input data resolution d points/m^2, a tree point with position (T_x, T_y, T_z), if there is no ground point at position (T_x, T_y), we consider the ground as possibly occluded. In this case, we count the number of ground points n at position (T_x, T_y) within a radius r in the ground k-d tree. If $n < \pi r^2 d$, the ground area is considered under-sampled and we augment the input data with a new ground point (G_x, G_y, G_z) under the tree point as follows:

$$G_x = T_x, G_y = T_y, G_z = \frac{\sum_{i=1}^{n} z_i w_i}{\sum_{i=1}^{n} w_i} \tag{1}$$

where each ground point i in the neighborhood of radius r has position (x_i, y_i, z_i), and the weight w_i is based on its 2D distance to the new ground point:

$$w_i = \frac{1}{\sqrt{(x_i - G_x)^2 + (y_i - G_y)^2}} \tag{2}$$

The radius of the neighborhood r is adjusted depending on the resolution d of the data set. Experiments on several data sets with d ranging from 0.5 points/m^2 to 25 points/m^2 show that 16 neighbors yield good results and performance, so we use $r = \sqrt{\frac{16}{\pi d}}$ for our tests. This method effectively fills the ground holes under trees as seen in Figure 3(b).

5.3 Label Building Boundaries

Classification only provides categories of ground, trees, and buildings. To locate walls, points on building boundaries (roof edges) need to be identified. We use a threshold on the number of neighboring ground points to identify if a building point belongs to the boundary.

Figure 4 shows a bird-eye view of various cases where a point lies on building boundaries. A building point may be adjacent to the ground (point A), another building (point

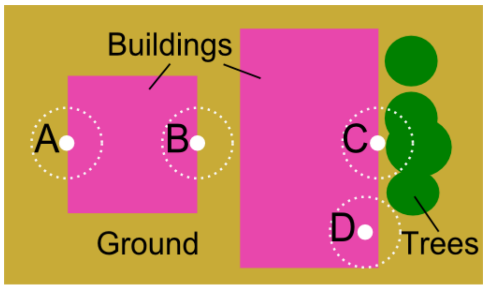


Figure 4: Various cases of boundary points.

B), or trees (point C). We claim that boundary points have half of their neighboring points in radius r as ground points. An ideal example is point A. If r does not overlap with the adjacent building, the claim also holds for point B. Since ground points under trees are recovered prior to this process (in Section 5.2), the claim is still valid for point C. The neighborhood of any point on roof corners contains even more ground points than that of point A, B or C, thus corner points satisfy the claim as well. However, this simple criterion assumes a constant point sample density. For areas with a variable point density, and areas that only the building points near the boundary (for example, point D) in stead of those right on the edge (point A, B and C) are sampled, there may be too few ground points. In these cases, we adjust a tolerance δ to find boundary points. Given a neighborhood radius r and a building point, we count the number of building points N_{bp} and the number of ground points N_{gp} in the neighborhood. The building point is labeled as a boundary point if:

$$\frac{N_{gp}}{N_{gp}+N_{bp}} >= 0.5 - \delta \tag{3}$$

Equation (3) is only valid if N_{bp} refers to points from a single building. In practice, the neighborhood is chosen to be small enough to only contain points from a single building as well as to contain a sufficient number of samples for a valid measurement. Our experiments use a fixed radius $r = 2$, corresponding to 2 meters (the datasets are in UTM coordinates), a distance that is smaller than the distance between any two buildings yet provides enough samples for the estimation of (3) given the datasets. A value of $\delta = 0.1$ produces good results in our experiments.

Once a boundary point is identified, the corresponding ground point directly under it is estimated similarly to augmenting the ground point under a tree point (Section 5.2). Because these two points share the same x-y value, only the z value of the ground point, i.e. G_z, needs to be stored with the boundary point. By processing all building points, the building boundaries are approximated and stored in a separate k-d tree. Note that a point labeled as boundary is also a building point, so it occurs in both k-d trees.

This threshold method is simple but provides good results in our experiments. If needed, more sophisticated methods like [9, 12] can approximate building boundary more precisely with the trade-off of more pre-processing time.

5.4 Estimate Normals

The input of this step is points labeled as trees, ground, buildings and building boundaries. Only neighbors of the same class are used to estimate the normal of a point.

We calculate the normal of a tree point as in [20] by the cross product of vectors formed with two neighbors. Since tree points are usually sparse and randomly sampled, noisy normals computed this way lead to a natural appearance of vegetation.

Since ground points form a large continuous surface, we use Principal Component Analysis [28] over k nearest neighbors of a point for a more precise normal estimation. As in Section 5.2, we use $k = 16$. Since Principal Component Analysis determines a normal up to a sign, we fix the sign so that the ground normal points skyward, i.e., the angle between the normal and the unit vector along z-axis should be smaller than 90 degrees.

Building points corresponding to a single building form a continuous surface, so Principal Component Analysis is also applicable to estimate these normals. However, the segmentation of separate buildings is not available, so we analyze the k points within the neighborhood radius $r = 2$ as in Section 5.3 to ensure that all neighbors of a building point are constrained to the same building as the point. The sign of a building normal is fixed similar to that of a ground normal, as points on building roofs also faces the sky.

Because boundary points are also building points, they share the normals computed as above.

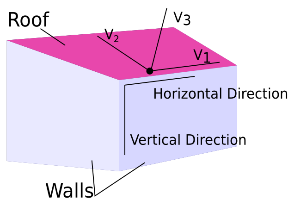


Figure 5: Geometric interpretation of Principal Component Analysis.

5.5 Estimate Wall Normals

When computing the normal of a boundary point as in Section 5.4, Principal Component Analysis also provides axes directions of the underlying tangent plane. Let $l_1 \geq l_2 \geq l_3$ be the sorted eigenvalues of the covariance matrix, and v_1, v_2, v_3 the associated eigenvectors. As shown in the geometric interpretation in Figure 5, v_1 estimates the axis with greatest variance of the tangent plane, v_2 estimates the other axis of the tangent plane, and v_3 the normal at the point.

To attach a piece of wall to a given boundary point, only the vertical direction (unit vector along z-axis) of the wall is assumed and fixed. We use the largest tangent plane axis direction v_1 of the corresponding boundary point to estimate the horizontal direction of the wall, because the two vectors are parallel as depicted in Figure 5. The normal of the wall is then calculated as the cross product of its horizontal direction and its vertical direction.

5.6 Augment Wall Points

When the user selects point as the primitive to reconstruct missing walls, a set of augmented points are created to model the walls.

We know the approximate data resolution d points/m^2 of input LiDAR data. For a boundary point (B_x, B_y, B_z) with

G_z as the ground height at the foot of the corresponding wall, the two end points of a vertical line within the wall are known. Augmented points are created by linear interpolation along z-direction of this line. A number of $(\frac{B_z - G_z}{\sqrt{d}} - 2)$ equidistant wall points are created and assigned the same wall normal calculated in Section 5.5 for the boundary point. The augmented points are then added to a separate k-d tree for walls. Figure 8(b) shows a rendering result for point represented walls.

6. RENDERING

Our rendering algorithm is based on a GPU-accelerated PBR with deferred shading framework [11]. The rendering of points is summarized in three basic passes.

1. **Visibility pass** calculates the splat size (radius) and shape of a corresponding point, and renders the splats only to the depth buffer with ϵ-offset that determines the maximum distance between the two blended splats.
2. **Attribute pass** renders the splats again. Because this pass uses the depth buffer from the visibility pass and disables writing to it, only splats within ϵ-distance are blended. Besides depth buffer, other rendering buffers supported by OpenGL Multiple Render Targets are used to store accumulated normals and colors of blended splats.
3. **Shading pass** traverses pixels of the screen to normalize the accumulated normals and colors stored in rendering buffers of the attribute pass and to apply accurate per-pixel Phong shading.

If point-wall representation is selected, the point data is complete when rendering. The above mentioned 3-pass renders, blends, and shades points of all classes together to generate the final image.

If quadrangle or ARP is selected to represent wall geometry, the rendering process is modified as depicted in Figure 2. A procedure runs after the visibility pass to generate wall-surface geometry as detailed in Section 6.1 and Section 6.2. The rendering loop is modified to handle both splats and polygonal primitives as described in Section 6.3.

6.1 Generate Wall Quadrangles

The 2.5D characteristic of building structures assumes vertical walls, so it is natural to use vertically-oriented quadrangles as wall primitives.

For a boundary point (B_x, B_y, B_z) with G_z as the ground height at the foot of the wall, a vertical quadrangle is generated to connect the boundary point to the ground. The quadrangle height is $B_z - G_z$ and its normal is calculated in Section 5.5. The width of the quadrangle is $2r$, where r is the radius of the corresponding rendered splat of the boundary point. Since the splat radius r is calculated in the visibility pass, the generation of quadrangles can occur immediately after that pass. We procedurally generate a triangle fan representing the wall quadrangle in a geometry shader [7] whenever the splat of a boundary point passes the visibility pass. As splat radius is designed to ensure overlapping with neighboring splats, quadrangles created this way also overlap with neighboring quadrangles.

If quadrangles are rendered with a solid color, shading aliasing artifacts often occur when two quadrangles with different normals overlap, as seen in Figure 6(a). To alleviate the problem, we render quadrangles with gradient color, so overlapped quadrangles have smooth color changes as shown in Figure 6(b). Figure 8(c) shows a rendering result for quadrangle represented walls.

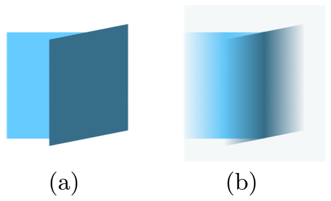


Figure 6: Rendering of two overlapping wall pieces with different normals. (a) Using the same solid color. (b) Using the same gradient color.

6.2 Generate Wall ARPs

For point clouds that have very low data resolution, the horizontal direction and normal of a piece of wall surface estimated in Section 5.5 is often inaccurate. Quadrangles with inaccurate orientations may produce holes in overlapping areas with neighbors, but rectangular parallelepipeds are always watertight when viewed from any direction, as seen in Figure 7. We devise this approach of using ARPs to represent solid walls without computing their orientations based on normals.

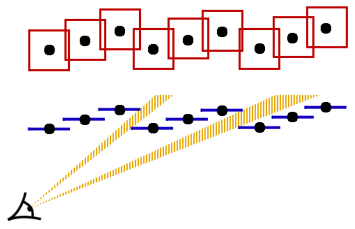


Figure 7: Comparison of ARP-walls (red rectangles) and quadrangle-walls (blue segments) over the same group of boundary points (black dots), viewed from top. When viewed along the directions of yellow segments, gaps are found in quadrangle-walls. While ARP-walls are watertight in all directions.

Because walls stand between the ground and roofs, the top and bottom faces of an ARP will always be hidden by the splats of points at the top and bottom. Therefore, only the vertical faces need to be generated for the ARP. For the four vertical faces, at most two of them can be seen at anytime. Similar to the quadrangle approach of Section 6.1, we use $2r$ as the width of each vertical face, use the geometry shader after the visibility pass to generate a triangle fan representing the two visible faces of a ARP, and use gradient colors for faces to alleviate shading aliasing. Figure 8(d) shows a rendering result for ARP represented walls.

6.3 Rendering Extension

The deferred shading of point splats and polygonal geometries is enabled by sharing the same depth buffer for the visibility pass and sharing the rendering buffers to accumulate normals and colors. The extended renderer runs as follows.

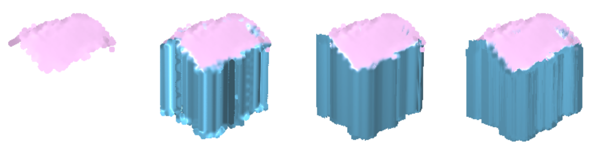


Figure 8: Different wall representations. (a) Original roof-only point cloud. (b) Augmented with point-walls. (c) Augmented with quadrangle-walls. (d) Augmented with ARP-walls.

1. Points in the classes of ground, trees and buildings are rendered as splats using the visibility and attribute passes.

2. Boundary points passing the visibility pass proceed to a procedure to generate wall geometries as detailed in Section 6.1 and Section 6.2. To improve performance, back face culling is implemented for polygons based on the angle between their normals and the viewing ray. The visible polygonal geometries are then rendered using a similar attribute pass as the one used for points.

3. The same shading pass as the basic PBR normalizes the accumulated normal and color buffers, and applies per-pixel Phong shading to generate the final image.

7. EXPERIMENTAL RESULTS

The implementations use the C# language under .NET 4.0 environment, OpenGL 2.1, Tao Library [6] and the Cg [25] language for shaders. Three wall reconstruction approaches are implemented, we also re-implemented the PBR system of [11] for comparison.

A computer equipped with dual-core Intel Xeon 2.0GHz CPU, 4GB RAM, NVIDIA GeForce GTX260 896MB graphics card, Western Digital 7200RPM hard disk and Windows Vista was used for testing. All tests were rendered at 1280×800 pixels.

7.1 Performance

Table 1 lists basic information of various data sets used for the experiment. The data of Los Angeles is mostly urban area consists of buildings and less trees, the other two data sets are residential areas where plenty of trees can be found.

Table 1: Basic information of test data sets.

Data set	Denver	Los Angeles	Atlanta
File size (MB)	44.0	49.6	104.6
Resolution (pts/m^2)	6.6	0.58	22.9
# points (M)	1.64	2.60	5.48
% ground	67	56	42
% building	20	28	16
% tree	13	12	42
% other	0	4	0

The pre-processing is measured in two parts: (i) the time spent on building up data structures, and (ii) the total time for all pre-processing operations except classification shown in Figure 2. Pre-processing is different for different wall reconstruction approaches, so individual times for each choice of wall primitive are listed. Under default setting, we do ten runs for each data set with each wall primitive and record the average time in Table 2. The largest sum of (i) and (ii) among three wall reconstruction approaches is also listed as "Maximum total time" for each data set.

The time of (i) is proportionally increasing with the size of input file and the total number of points. It is also affected by the maximum number of points per quadtree leaf node. The time of (ii) depends on the number of points of each class and the actual scene-architectural layout. For example, given the same number of building points, different layouts will obtain varied numbers of boundary points, thus impacting processing time. Although we cannot define the precise effects of every factor, the "Average time/M points" shows that with our test data, the system takes 8-15 seconds to pre-process one million points.

Table 2: Pre-processing time (in seconds) of three wall reconstruction approaches.

Data set	Denver	Los Angeles	Atlanta
Structure build-up	2.64	4.06	8.56
Point-wall	12.26	18.87	58.23
Quadrangle-wall	11.35	16.64	49.01
ARP-wall	11.16	16.54	48.12
Maximum total time	14.90	22.93	66.79
Avg. time/M points	9.09	8.82	14.91

Ignoring factors like point density and rendering parameters such as blending distance ϵ and splat sizes, the performance of the LOD renderer is mainly dependent on the number of selected points rather than the total number of points. By adjusting the LOD parameter, one can always achieve interactive frame-rates.

For comparisons, we use fixed rendering parameters and render the scene with full detail, i.e. LOD disabled. The rendering performance of each approach is measured by adjusting the viewport to fit the whole scene area horizontally and recording the average frame-rate while orbiting the camera about the scene for a minute. Table 3 demonstrates the rendering performance of three wall reconstruction approaches and basic PBR with deferred Phong shading where no augmentation or procedurally generated geometries are added to the input data. All three wall primitives have comparable performance, the quadrangle representation performs slightly better than the other two. Compared to basic PBR, the overhead of our method is low. For example, using the quadrangle wall impacts performance by less than 35% in these test cases.

Table 3: Rendering performance (in fps) of our methods and the basic PBR.

Data set	Denver	Los Angeles	Atlanta
Point-wall	42.6	34.1	9.6
Quadrangle-wall	46.8	36	10.6
ARP-wall	42.5	32.5	10.7
Basic PBR	53.2	38.8	16.2

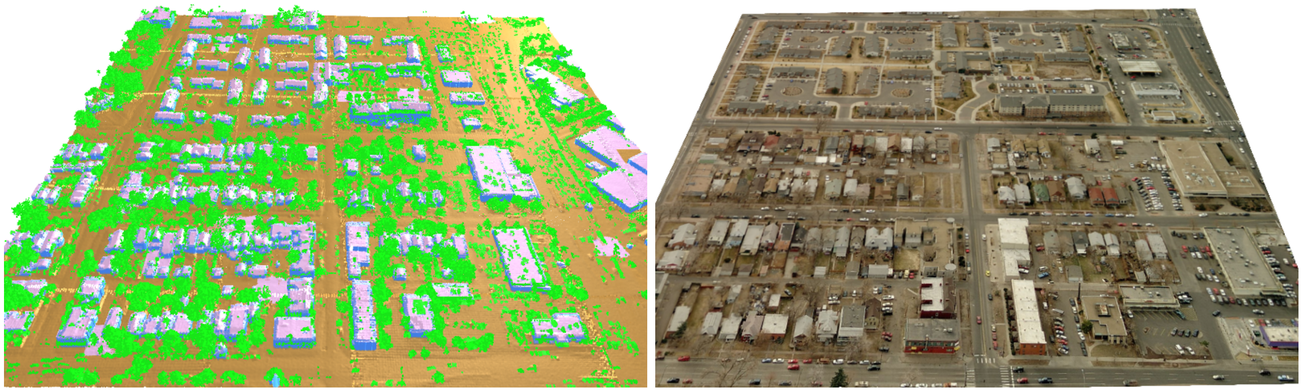

(a) (b)

Figure 9: Denver area. (a) Rendering with quadrangle-walls and LOD factor $f = 4$. (b) Aerial image from Microsoft Bing Maps.

7.2 Image Quality

Our method provides visually complete and appealing renderings for both urban areas (Figure 11(a)) and residential areas (Figure 9(a), Figure 10(a)), despite the variation in data resolution. Figure 9 and Figure 10 compare our visualizations with corresponding aerial images. Since the point clouds and images were not acquired at the same time, the vegetation looks quite different. However, our renderings provide clear building contours that perfectly match those in the images.

Figure 8 compares the rendering quality of three wall primitives. Points provide a unified look of the scene, but as stated in [20], round and fuzzy splats do not approximate flat surfaces and edges as well as curved natural features. The point-wall is often less visually appealing than the other two primitives. ARPs represent the sharp corners of walls best as vertical faces of the primitive form corners naturally, but because of axis-alignment, they may create bumpy walls. Quadrangles make the best wall representation as they provide unified look along vertical direction and smooth appearance on overlapped areas, if gaps caused by inaccurate wall orientations are not present.

7.3 Comparison with Other Methods

The state of the art LiDAR visualization systems [20, 22] were built upon basic PBR with no consideration of occlusions. Figure 1 clearly shows that by adding building walls and filling ground under canopies, our system provide a better visualization than basic PBR.

Figure 11 compares the renderings of our quadrangle-wall approach with the visualization of complete building models created offline by [34] on the same Los Angeles area. Our rendering provides comparable visual information for buildings as the polygon models provide. The bottom right close-up even shows finer details of the stadium than the respective model. Since we only consider local neighbors, the appearance of walls are highly dependent on the quality and sampling of the raw data. As shown in the top right close-ups in Figure 11(a), buildings rendered by our method look less appealing than the corresponding models. However, the goal of this work is not to compete the image quality with the compute-intensive urban modeling but to improve the state of the art interactive city-scale visualization.

8. CONCLUSION AND FUTURE WORK

We are the first to utilize the 2.5D characteristic of building structures to fill holes in the visualization of aerial LiDAR point cloud. We present a system using a hierarchical and out-of-core approach to data management and GPU-accelerated PBR with deferred shading. The system treats points differently by classification and incorporates three distinct geometries to represent building walls: points, quadrangles and ARPs. The method is generic to any data with buildings and trees regardless of point density. The rendering of our PBR system provides comparable 3D cues to those obtained with polygonal 3D modeling while maintaining interactive frame-rates.

We have found that among the three building wall reconstruction approaches, quadrangle-walls achieve the best trade-off between performance and image quality. Point-walls are useful when the processed point cloud is to be rendered with other rendering algorithms, because the augmented points are added in pre-processing and therefore independent of run-time rendering algorithm. ARP-walls are best employed with noisy or low resolution data as this approach does not require the estimation of a normal vector.

As splats are round and fuzzy, sharp features and corners are not reproduced accurately with PBR. We will investigate alternatives to address this problem in our future work. To improve the visual appearance, texture mapping aerial imagery to points is needed and again part of our future work. The application of non-photorealistic rendering to LiDAR data is also an interesting area to explore. We also recognize that a semantic framework [16, 32] can further generalize and extend the presented approach to other objects than buildings and trees. The development of a semantic object description for a more procedural visualization is another future work.

9. ACKNOWLEDGMENTS

This work is supported by Chevron Corp. under the joint project, Center for Interactive Smart Oilfield Technologies (CiSoft), at the University of Southern California. We thank the reviewers for their valuable suggestions. We are grateful for the programming support from Miao Wang. We thank Airborne 1 Corp. and Sanborn Corp. for providing the

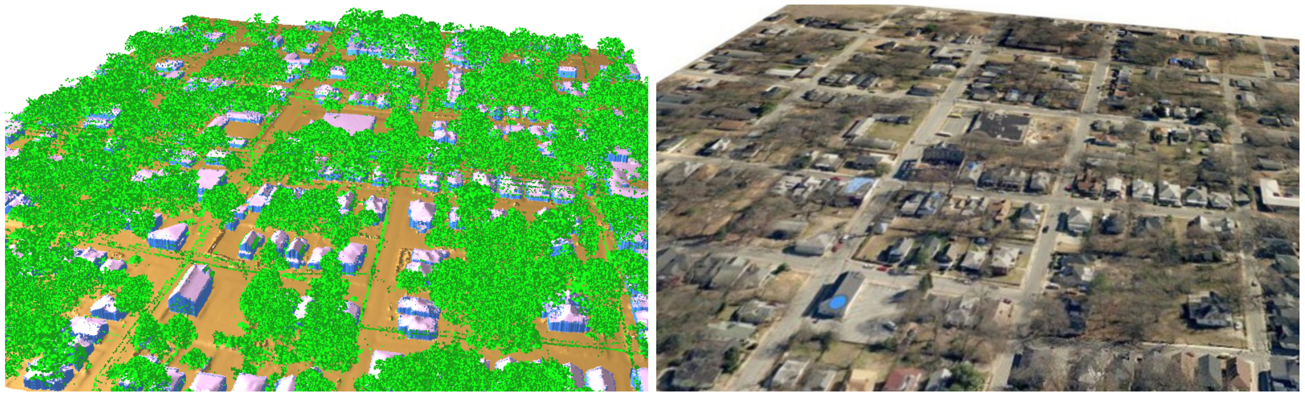

(a) (b)

Figure 10: Atlanta area. (a) Rendering with quadrangle-walls and LOD factor $f = 4$. (b) Aerial image from Microsoft Bing Maps.

LiDAR data. We thank Qian-Yi Zhou for sharing his classification code, providing Figure 11(b), and valuable discussions. We thank Sang-Yun Lee, Jiaping Zhao, Wei Guan and Tanasai Sucontphunt for their useful comments and support.

10. REFERENCES

[1] Exelis E3De. *http://www.exelisvis.com/language/en-US/ProductsServices/E3De.aspx*.

[2] Exelis ENVI. *http://www.exelisvis.com/ProductsServices/ENVI.aspx*.

[3] GRASS GIS. *http://grass.osgeo.org*, 1999.

[4] 3D LiDAR visualization tool. *http://lidar.asu.edu/LViz.html*, 2008.

[5] Esri arcgis the complete geographic information system. *http://www.esri.com/software/arcgis/*, 2008.

[6] The Tao Framework, Open source library. *http://sourceforge.net/projects/taoframework/*, 2008.

[7] Geometry shader. *http://www.opengl.org/wiki/Geometry_Shader*, 2010.

[8] Vertex buffer object. *http://www.opengl.org/wiki/Vertex_Buffer_Object*, 2012.

[9] A. Alharthy and J. Bethel. Heuristic filtering and 3d feature extraction from lidar data. In *In ISPRS Commission III, Symposium 2002 September 9 - 13, 2002*, pages 23–28, 2002.

[10] D. Bhagawati. Photogrammetry and 3-d reconstruction - the state of the art. In *ASPRS Proceedings*, 2000.

[11] M. Botsch, A. Hornung, M. Zwicker, and L. Kobbelt. High-quality surface splatting on today's gpus. In *Symposium on Point-Based Graphics 2005*, pages 17–24, June 2005.

[12] W. Cho, Y.-S. Jwa, H.-J. Chang, and S.-H. Lee. Pseudo-grid based building extraction using airborne lidar data. *International Archives of Photogrammetry and Remote Sensing*, 35(part 3):378–381, 2004.

[13] J. Danahy. Visualization data needs in urban environmental planning and design. In D. Fritsch and R.Spiller, editors, *Photogrammetric Week '99*, pages 351–365, Toronto, 1999. Heidelberg: Herbert Wichmann Verlag.

[14] J. Fernandez, A. Singhania, J. Caceres, K. Slatton, and R. K. M Starek. An overview of lidar processing software. Technical report, Geosensing Engineering and Mapping, Civil and Coastal Engineering Department, University of Florida, 2007.

[15] M. Gross and H. Pfister, editors. *Point-based graphics*. The Morgan Kaufmann Series in Computer Graphics, 2007.

[16] R. Guercke, C. Brenner, and M. Sester. Generalization of semantically enhanced 3d city models. *GeoWeb 2009 Academic Track - Cityscapes*, XXXVIII-3-4/C3:28–34, 2009.

[17] J. Hu, S. You, and U. Neumann. Approaches to large-scale urban modeling. *IEEE Comput. Graph. Appl.*, 23(6):62–69, Nov. 2003.

[18] M. Isenberg. LAStools: converting, filtering, viewing, gridding, and compressing LIDAR data, 2012.

[19] L. Kobbelt and M. Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28:801–814, 2004.

[20] B. Kovač and B. alik. Visualization of lidar datasets using point-based rendering technique. *Comput. Geosci.*, 36(11):1443–1450, Nov. 2010.

[21] O. Kreylos, G. Bawden, and L. Kellogg. Immersive visualization and analysis of lidar data. In G. Bebis, R. Boyle, B. Parvin, D. Koracin, P. Remagnino, F. Porikli, J. Peters, J. Klosowski, L. Arns, Y. Chun, T.-M. Rhyne, and L. Monroe, editors, *Advances in Visual Computing*, volume 5358 of *Lecture Notes in Computer Science*, pages 846–855. Springer Berlin / Heidelberg, 2008.

[22] M. Kuder and B. Zalik. Web-based lidar visualization with point-based rendering. In *Proceedings of the 2011 Seventh International Conference on Signal Image Technology & Internet-Based Systems*, SITIS '11, pages 38–45, Washington, DC, USA, 2011. IEEE Computer Society.

[23] F. Lafarge and C. Mallet. Building large urban environments from unstructured point data. *Computer Vision, IEEE International Conference on*, 0:1068–1075, 2011.

[24] LAS Specifications. *http://www.asprs.org*.

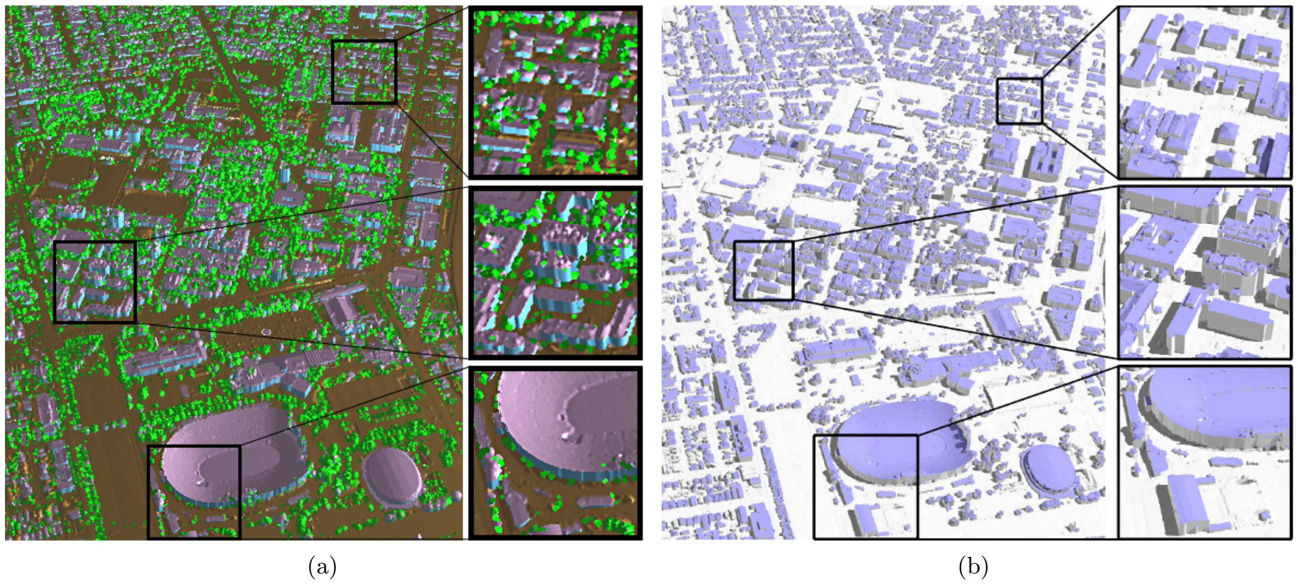


Figure 11: Rendering comparison of Los Angeles area. (a) Using our rendering with quadrangle-walls, LOD is disabled. Trees are colored in green. (b) Visualization of the full geometric models from [34] using a ray tracer and shadow maps, image courtesy to Qian-Yi Zhou.

[25] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard. Cg: a system for programming graphics hardware in a c-like language. In *ACM SIGGRAPH 2003 Papers*, SIGGRAPH '03, pages 896–907, New York, NY, USA, 2003. ACM.

[26] B. C. Matei, H. S. Sawhney, S. Samarasekera, J. Kim, and R. Kumar. Building segmentation for densely built urban regions using aerial lidar data. In *CVPR*. IEEE Computer Society, 2008.

[27] T. C. Palmer and J. Shan. A comparative study on urban visualization using lidar data in gis. *URISA Journal*, 14(2):19–25, 2002.

[28] K. Pearson. Pearson, k. 1901. on lines and planes of closest fit to systems of points in space. *Philosophical Magazine*, 2(11):559–572, 1901.

[29] B. T. Phong. Illumination for computer generated pictures. *Commun. ACM*, 18(6):311–317, June 1975.

[30] C. Poullis and S. You. Automatic reconstruction of cities from remote sensor data. pages 2775–2782, 2009.

[31] V. Verma, R. Kumar, and S. Hsu. 3d building detection and modeling from aerial lidar data. In *Proceedings of the 2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Volume 2*, CVPR '06, pages 2213–2220, Washington, DC, USA, 2006. IEEE Computer Society.

[32] W. Xu, Q. Zhu, and Y. Zhang. Semantic modeling approach of 3d city models and applications in visual exploration. *The International Journal of Virtual Reality*, 9(3):67–74, 2010.

[33] Q.-Y. Zhou and U. Neumann. A streaming framework for seamless building reconstruction from large-scale aerial lidar data. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 0:2759–2766, 2009.

[34] Q.-Y. Zhou and U. Neumann. 2.5d dual contouring: A robust approach to creating building models from aerial lidar point clouds. In K. Daniilidis, P. Maragos, and N. Paragios, editors, *Computer Vision - ECCV 2010, 11th European Conference on Computer Vision, Heraklion, Crete, Greece, September 5-11, 2010, Proceedings, Part III*, volume 6313 of *Lecture Notes in Computer Science*, pages 115–128. Springer, 2010.

[35] Q.-Y. Zhou and U. Neumann. 2.5d building modeling with topology control. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, 0:2489–2496, 2011.

[36] Q.-Y. Zhou and U. Neumann. 2.5d building modeling by discovering global regularities. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference*, 2012.