An approach to visualization of large data sets from LIDAR

Boštjan Kovač*

Faculty of Electrical Engineering and Computer Science
University of Maribor
Smetanova 17, SI-2000 Maribor
Slovenia

Abstract

Rapid development of laser scanning technology in past decades has resulted in a wide area of its applications. LIDAR is a system that uses this technology to gather information about distant targets. Gathered data are stored into large data sets that are further processed, visualized and analyzed. Fast and accurate visualization is the key factor when working with LIDAR point clouds. The main problem that arises is that vast amount of data can easily exceed memory and processing capacities of modern day computers.

In this paper we present an approach to visualization of large LIDAR point clouds in real time entirely on graphical processing unit using a point-based rendering technique. Our method is based on dynamic data loading and efficient two-pass rendering utilizing approximation of elliptical weighted average splatting with rotated splats. Expensive rendering tasks are delegated to programmable graphics unit to save CPU resources. The proposed system offers realistic visualization of LIDAR point clouds in real time that is visually and performance wise comparable to other solutions, while not requiring any comprehensive preprocessing such as TIN generation beforehand.

Keywords: LIDAR, terrain visualization, point-based rendering

1 Introduction

Light Detection and Ranging (LIDAR) is a remote sensing technology that detects range and other properties of distant objects. Three main components of a LIDAR system are laser scanner, inertial navigation system (INS) and global positioning system (GPS). The entire system is usually mounted on an aircraft that flies in multiple overlapping patches over the terrain. The laser scanner emits infrared laser beams at high frequency toward the ground. Properties of the returned scattered beam are measured and recorded along with data from positioning (GPS) and orientation (INS) subsystems. Gathered data is processed after the flight and every measured point is referenced with its geospatial position [9]. Processed point clouds are then written into output files of different formats, with the preferred format being the open LAS file format [11]. These files usually contain millions of points which amount to a few gigabytes of data.

LIDAR systems are known to achieve 15 cm vertical and 30 cm horizontal accuracy [9], which along with their low cost makes them a technology of choice in various fields. These include forestry, geology, meteorology, coastal engineering, urban planning, etc. Fast and representative visualization of gathered data is necessary for further analysis and verification of results. Objective of this paper is to present an approach to visualization of LIDAR data sets of arbitrary sizes in a true interactive 3D environment in real time. To achieve this, we need to overcome the obstacle of limited computer resources that are at our disposal. Computer systems have been advancing rapidly, but their memory and processing capacities are still not enough to handle virtually unlimited size of LIDAR data in real time.

3D scenes usually consist of 3D meshes which in turn contain millions of triangles [5]. This is possible because of the evolution of graphical processors in past decade and revolution that programmable shaders have brought to the field of computer graphics. But as the capabilities of GPUs are constantly increasing, display resolution has been lagging behind. This exposes an interesting phenomenon, where a single projected triangle covers just a few pixels, but nonetheless brings the whole cost of a triangle mesh with it. In case of LIDAR data, cost of a triangle mesh is mapped to triangulation of the whole point cloud. This is inefficient, since points are by default independent entities. A much more intuitive way to visualize LIDAR data is therefore to treat each point as an independent primitive and render it as such. Additional benefit of point-based geometry is that hierarchical encoding schemes provide compact storage and efficient transmission of these data sets.

The main problem that point-based rendering techniques encounter is that sampled points do not have infinite resolution, resulting in subsequent gaps between them [5]. These gaps can be avoided by applying filters, increasing sampling density or by surface splatting. The latter technique associates each point with a normal vector and a radius. Every point is thus represented as a small disc in 3D

*bostjan.kovac@gmail.com

space which can span over many pixels when projected on the screen. Use of points as rendering primitives has first been proposed by Levoy and Whitted [14], who were followed by Grossman and Dally [12]. Hardware graphical accelerators and introduction of programmable graphical pipeline resulted in renewed interest and subsequent advances in point-based rendering methods. Rusinkiewicz and Levoy were the first to use hardware acceleration in QSplat [16] in order to render large datasets of the Digital Michelangelo Project. They used a tree of bounding spheres to determine visibility, to control levels of detail and for rendering. They also proposed two rendering passes to properly blend overlapping splats. It was the first system designed to interactively render large data sets gathered by modern scanning devices.

Zwicker et al. [18] have mathematically formulated elliptically weighted average filtering (EWA) for point-based rendering by introducing fuzzy splats. Ren et al. [15] have reformulated EWA approximation for implementation entirely on GPUs, since exact implementation is not possible due to technical limitations of graphical hardware. This drastically improved visual quality of the output, so others [5, 6, 8, 10, 4, 19] followed with their methods on improving visual quality and speed of rendering.

We have implemented TerraForm application to display, process and analyze LIDAR data. In this paper we discuss problems that we have encountered while rendering large LIDAR data sets and how we overcame them to produce an interactive 3D environment that is able to output the results of our calculations in real time. In the following section we go into the preprocessing stage of our algorithm, where we create our data structure. In the next section we discuss how dynamic data management and visibility are handled. At last, we explain the rendering algorithm. Finally, we discuss results.

2 Data structure

One of our primary objectives was achieving real time visualization of amounts of data that can considerably surpass modern computer's system and graphics memory. We have chosen a quadtree of bounding rectangles as a data structure because it allows simple access, management and visibility culling of smaller subsets of data. Terrain is sampled from aircraft in multiple flyovers and the points are saved into LAS file sequentially in order they were sampled. This means that adjacency in the file does not necessarily correspond to adjacency in the real world. In order to load these points on demand we therefore need to index them.

In the following subsections initialization of our data structure and point indexation are described.

2.1 Initialization

LAS file format contains a lot of meta information besides raw point data. This data is used during the initialization phase to construct the quadtree and determine the bounding boxes of nodes. Tree construction begins at the root node, which covers the whole area. Terrain is then equally divided among its children down the tree until the minimum predetermined number of points per bounding box is reached. 5.000 points per node turns out to be a good tradeoff between size of the entire quadtree structure and the speed of algorithms that are used to load and render data. Much smaller values can result in considerably larger memory requirements when working with very large or dense data sets. Each node contains average colour, average normal vector of the points it encapsulates and four pointers to its children. Leafs on the other hand contain much more data, which are mainly required for dynamic data management described in the next section. Data contained in the leaf structure are presented in Table 1.

Field	Size (bytes)
Leaf state	1
Number of designated points	4
Id of array in GPU memory	4
Point lists	Depends on number of loaded points
Meta info about point lists	8*number of point lists
First point index	4
Number of points	2
List index	2
Bounding box	16

Table 1: Leaf structure

2.2 Indexing

Initialization phase divided 3D space into a grid of rectangles that can be referenced through leafs. In this step points are referenced with their designated leafs by filling leaf's meta information about contained point list arrays.

Indexing starts by sequentially going through the input file and sending read points down the tree. No points are actually stored in this step, only information about them is gathered and used to index their position in the file. Successive points in the file usually belong to the same scan line that sampled them from the terrain. This means that a subset of consecutive points that belong to the same leaf probably exists in this array. All we need to store in the leaf node is therefore the index of the first point of that subset and the number of consecutive points, which we find by traversing the input array of points with a constant step and retracting with bisection when we encounter the first point outside the bounding box. Bisection is repeated until the last point contained in the leaf is found.

3 Dynamic data management

Our data structure is created during the initialization and indexing phase, but points are not inserted into it. Amount of data we can store in our data structure is limited by available system RAM and video memory. Since we have already established that quantity of LIDAR points can surpass system's memory limit, we store only interesting points, i.e. visible and nearly visible points. Visibility is determined by user's interaction with the 3D world, so the interesting set of points is constantly changing, making point management a dynamic and continuous process. Therefore we implemented it asynchronously in a separate thread not to interfere with other important tasks, such as rendering and interaction. The point loading thread contains a queue of requests that works on FIFO principle.

Different memory chips have varying speeds and transfer rates. When considering speed of visualization, it is important to note that video memory of graphics card provides the fastest way to access data that need to be rendered. Therefore all points that are currently visible or are near the visible frustum are stored in very fast graphics card memory as shown in Figure 1. Because user interaction can be rapid and unpredictable, points near the viewing band have to be uploaded to the video memory to ensure smooth movement and interaction with the world. Points that fall just outside of nearly visible terrain are stored in system RAM so they can be quickly transferred to the video memory. Points in video memory are kept in vertex buffer objects, which provide the quickest access during rendering. Transfer of points from system to video memory is very fast compared to reading from disk, so no queuing mechanism is necessary.

Memory location of points is kept in leaf's 'leaf state' field. Possible states are defined as:

- Unloaded - points are left in file
- Requests loading - points need to be read from LIDAR file/data source
- Loaded in RAM - stored in system memory
- Loaded in GPU ram - stored in video memory

Primary reasons to implement points loading thread asynchronously are slow disk speed and the lack of normal vectors in the input file. Loading process occurs in this order: load points from disk into memory, select appropriate subset of points, insert points into leaf, calculate normal vectors. While loading and inserting are straightforward operations, we will describe point selection and normal calculation in the following two subsections.

3.1 Point selection

Selection algorithm is needed because older video cards may not have enough memory to contain visible portion

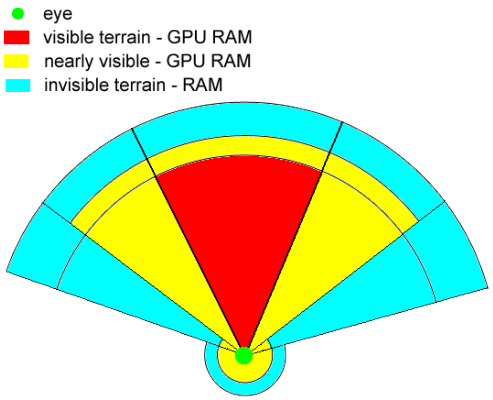

Figure 1: Leaf visibility and states

of the terrain. In this case only a subset of all designated points have to be inserted into leaf's point lists.

Points on the same list are more or less equidistant, as are the adjacent point lists. Reason lies in constant angle velocity of mirror that reflects the laser beam and the constant speed of aircraft. Also, when points are indexed, the lists first points often fall very near the border of the bounding box as seen in Figure 2a. Patterns are not visible due to sufficient density of points when rendering a full set of points. However, if we cannot afford to render all points and choose them with an equidistant step, we get a pattern shown in Figure 2a. Points are therefore chosen with a random step (Figure 2b) to assure pattern-free rendering.

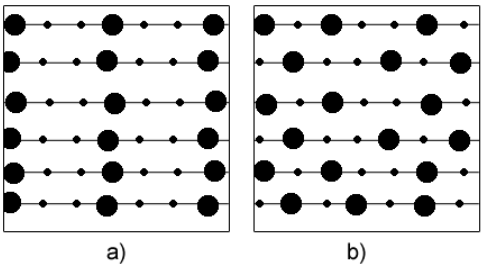

Figure 2: Equidistant (a) and random step (b) point indices

3.2 Normal vectors

Normal vectors for each point have to be calculated because LIDAR systems do not measure them. We need two neighboring points besides the original one to calculate the normal vector. The closest two points are probably the one before and the one after it, but those points tend to be collinear, so we can use only one of them. We search for the closest valid third point by utilizing the data structure of our quadtree's leaf. Since a list of point lists is kept, it can be assumed that one of point lists adjacent to the list containing the original point contains the closest third point. Because aircrafts fly over the terrain in patches this is not always true. Points from a very distant point list

could in fact be the closest, but the margin of scanning error and approximate nature of our normal calculation do not encourage us to spend significant amount of time searching for them. After the required points are found the normal vector is calculated as a cross product of two edges of the triangle they form.

4 Rendering algorithm

A lot of rendering methods based on points have been proposed in recent years. Since LIDAR data is large and not very dense when compared to scanned 3D models, our rendering algorithm is based on Botsch and Kobbelts algorithm [5]. It is known as very fast while providing decent visual quality [17]. It was also designed to be implemented entirely on graphic cards with vertex and fragment shaders.

Visualizing LIDAR points as single pixels is not acceptable as they are too sparsely spread around the terrain and the resulting image would contain too many holes even at very large viewing distances. A preferred way to visualize points is therefore by using splats, which are disk shaped objects possibly rotated in 3D space according to the point's normal vector. The splat is therefore defined by its centre's position in 3D world, radius, normal vector and colour. Benefit of using splats is that they are not piecewise constant but are represented with linear geometry. Therefore they exhibit the same quadratic approximation order as triangle meshes [5]. Another benefit of choosing splats over simple one-pixel points is splat filtering, which removes high frequency noise that is often produced by rendering unfiltered points.

Rendering points as splats requires several tasks to produce a hole-free image of acceptable quality. First, we calculate splat's size, which depends on its distance from the viewer. Secondly, splat's shape has to be determined based on its rotation and position. These two steps render dynamically sized and shaped splats that represent surface with good quality, but the third step is required to filter the unwanted artifacts from the final image. Filtering is performed by blending overlapping splats and thus eliminating alias. A quick normalization step is required at last to normalize gathered colour values. These steps are explained in the following subsections.

4.1 Splat size

Calculating splat size accurately is the key to avoiding holes in the image. Splat's position, radius and position of the viewer are used to make the necessary calculations. The procedure of OpenGL's transformation from 3D eye space coordinates to the final 2D image coordinates is shown in Figure 3. The visible portion of space is bound by a frustum defined by distances to its near (n) and far (f) planes, while parameters t (top) and b (bottom) control its opening angle. The frustum is projected on the unit cube $[-1,1]^3$, which is then projected into the $[-1,1]^2$ with a simple parallel projection that discards the Z-coordinate. Translation and scaling of these 2D coordinates map them into the window coordinates $[0,w]x[0,h]$. This is equivalent to projecting the viewing volume on the near plane z = -n and scaling the resulting coordinates by $h/(t-b)$ [5].

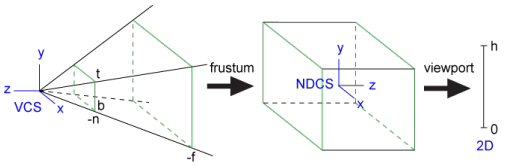

Figure 3: OpenGL's transformation pipeline

Splat's size is determined by the size of its projection in the window coordinates. Exact calculation requires the use of splat's position and normal vector in the process, which can result in a quite expensive calculation. Therefore we have used the approximation proposed by Chen and Nguyen [7], where the splat's bounding sphere is projected neglecting its offset from the optical axis and normal vector. Final image space splat size is therefore calculated by equation 1, where z_{eye} is splat's distance from the camera, r its radius and n,t,b,h projection parameters shown in Figure 3.

$$size_{win} = r\frac{n}{z_{eye}}\frac{h}{t-b} \tag{1}$$

Adjusting splat's size renders $size_{win} \times size_{win}$ large square on the image.

4.2 Splat shape

Splat is a small disc in object space, whose projection on screen is a rotated ellipse shown in Figure 4. Radius and orientation of the projected ellipse depend on the splat's normal vector transformed to eye coordinates. Modification of splat's size renders square that always faces the camera. To draw a rotated disc, we have to determine for every pixel inside that square whether it is a projection of a point from inside or outside the splat. A pixel lies within the rotated ellipse if the distance between its corresponding 3D point and splat's centre is smaller than the radii of the splat.

OpenGL's point sprite extension is used to project a texture instead of a single point. Pixels generated in this way receive texture coordinates in $[0,1]^2$ space. Simple subtraction and division transforms these coordinates to $[-1,1]^2$ space, where the centre point (0, 0) is the projected centre of the splat. Depth offset δz from the splat centre is computed for every generated pixel with coordinates $(x,y) \in [-1,1]^2$ by equation 2, where $(nx,ny,nz)^T$ represents the eye space normal vector.

$$\delta z = \frac{n_x}{n_z}x - \frac{n_y}{n_z}y \tag{2}$$

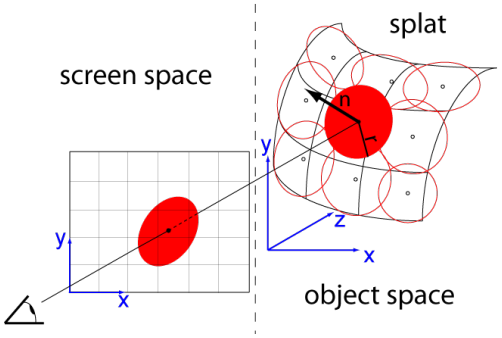

Figure 4: Splat projected to screen space

δz is then used to calculate point's distance from the splat's centre and correct its depth, as shown in Figure 5. Pixel $p(x,y)$ is inside the ellipse, if $\|(x,y,\delta z)\|^2 \leq 1$. Again, this is just an approximation which neglects the angle between the optical axis and splat's normal vector. This approximation can lead to very narrow ellipses, so we bound the maximum foreshortening of the ellipses as proposed in [2, 5, 8].

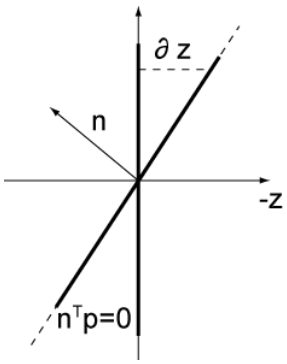

Figure 5: Depth correction

4.3 Filtering

Elliptical weighted average filtering is a method of filtering that is equivalent in quality to anisotropic filtering when dealing with polygons. It has been formulated for point-based rendering by Zwicker et al. in [18] and its approximations have been widely used by many authors. EWA consists of an object space reconstruction kernel and screen space low-pass filter. We have chosen to sacrifice visual quality in favour of higher rendering performance by not implementing the screen space filter. Overall visual quality would not be significantly improved to justify the performance hit it would cause.

To implement splat filtering, each splat is associated with a radially decreasing Gaussian weight function, thus forming our object space reconstruction kernel. The projection of such a splat produces an image space elliptical Gaussian that can be blended with other splats. Image space weight $\alpha(x,y)$ of a pixel is calculated from the distance between pixels corresponding 3D point and splat's centre in object space. We already calculated this distance in the previous step, when we calculated splats shape. Pixels weight is calculated by passing the distance from its centre to the Gaussian function:

$$\alpha(x,y) = Gauss[\|(x,y,\delta z)\|^T] \quad (3)$$

Splats should only be blended if their z-distance is small enough; otherwise the splats in front have to overwrite the splats behind. This is done in two rendering passes. In the first pass splats are rendered only to the depth buffer with ε-offset that determines the maximum distance between two blended splats. During the second pass we do not clear the depth buffer and disable writing to it. This way only splats within ε-distance are blended together. For this to work correctly each of splat's projected pixels must have its correct depth value set. Without correcting this value, each pixel gets its depth from the centre pixel. This is because point sprite extension renders image-plane aligned rectangles instead of points, which means that all rectangles' pixels have the same depth. Incorrect depth values result in blending artifacts near contours when viewing from flat angles [5]. The required window z-coordinate can be computed from adjusted eye space z-coordinate and δz by applying frustum and viewport mapping to it [5].

4.4 Normalization

Normalization step is necessary because the number of fragments blended into the same pixel can vary significantly throughout the image. This causes bright pixels and can be easily avoided by dividing each pixel's accumulated colour by its accumulated value of $\alpha(x,y)$.

5 Implementation

Our TerraForm application was implemented using C# language and .NET 2.0 environment. OpenGL 2.1 was used for rendering tasks through the use of open source Tao library [1]. Vertex and fragment shaders were implemented in Cg language. Results were measured in Windows XP running on AMD XP 3200+ processor and ATI Radeon x800 256MB graphics card.

We implemented our own library for reading LIDAR data with support for LAS file formats 1.0 and 1.1. Preprocessing, dynamic point management and visibility culling were implemented on CPU, while all rendering related tasks were delegated to GPU to free processing resources.

Point data consisting of positions, normal vectors and colours is stored in video memory's vertex buffer objects. They require data to be transferred from RAM to GPU RAM only once for static geometries, so they are the optimal choice for our type of data. Only a few commands are required to display geometric data stored in vertex buffer objects, so the overhead is also minimal. We used OpenGL's *GL_POINT_SPRITE_ARB* extension to render

splats. This let us draw textures instead of points, thus gaining access to all splats projected pixels in the fragment shader. Our texture contained only alpha channel in which a 2D Gaussian function was encoded to avoid expensive computation.

Vertex program calculates each point's projection and its normal vector in eye space coordinates. It also uses equation 1 to calculate splat size according to its distance from the viewer. Data common to all fragments generated from the same vertex is also calculated in vertex shader, since computation at this stage is not so costly.

Each fragment's depth offset is computed in fragment program using data passed from vertex shader and fragment's distance to the centre of the splat. Fragment's position in regard to the centre of the splat is derived from point sprite's texture coordinates. Equation 2 is applied to calculate depth offset. Then depth offset is used to calculate pixel's 3D distance from the centre of the splat in object space and test if the fragment is inside or outside of projected ellipse. Gauss function stored in point sprite's texture is used to calculate the weight of contained fragment and thus its contribution to the blended pixel. Fragment's depth correction has to be calculated using equation from [5] for blending to work correctly.

We store fragment's colour and weight contribution in an off-screen buffer in form of a texture. This texture is drawn after the second pass as a single rectangle spanning over the whole screen. A simple fragment shader program at last divides accumulated weighted RGB values by accumulated weight to produce a normalized image.

6 Results

Table 2 shows that we have achieved interactive frame rates. These results have to be taken with some consideration to the point clouds they were measured on and the varying point distance to the camera. Performance can vary significantly due to the dynamically changing point sizes, because larger points produce much more fragments and therefore require more processing in the fragment shader.

	Resolution	
# points	800x600	1280x1024
500 k	25	25
1 M	25	19.596
2 M	14.14	10.87
2.5 M	11.25	8.6

Table 2: Rendering performance in frames per second (fps). Note: Maximum fps is limited at 25.

Points within the viewing range that are not loaded into the video memory are loaded in the background process and displayed when they become available. This provides user with interactive 3D environment that is not handicapped by slow hard and optical disk speeds.

Much more important than speed is the visual quality of the application, as shown in Figure 6. The image is antialiased and generally of a very good quality. The number of holes is relatively small, thanks to the dynamic splat sizes. Point based rendering seems to be a perfect solution for terrain visualization, especially because ellipses are very good at approximating curved surfaces. Vegetation and trees are mostly rendered as scattered points, but this was expected because of insufficient sampling rate. Accuracy and density of LIDAR points have to be taken into account when evaluating visual quality. Therefore we cannot directly compare TerraForm's rendering quality to similar visualizations of high quality 3D models that were sampled with specialized scanners or generated by computer software.

As can be observed in Figure 7, point based rendering does not prove to be a very good choice when rendering urban areas. Because they contain tall buildings, vertical walls cannot be sampled densely enough by downward pointing laser beam. Big gaps are imminent in these cases and only triangulation would help to remove them.

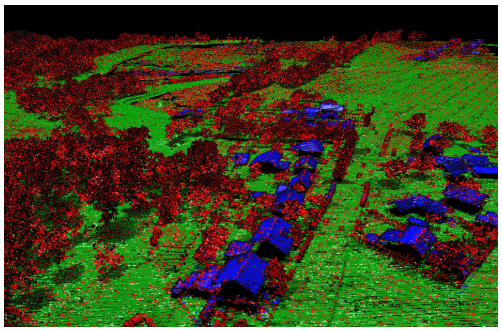

Figure 6: Countryside terrain rendering by our TerraForm application

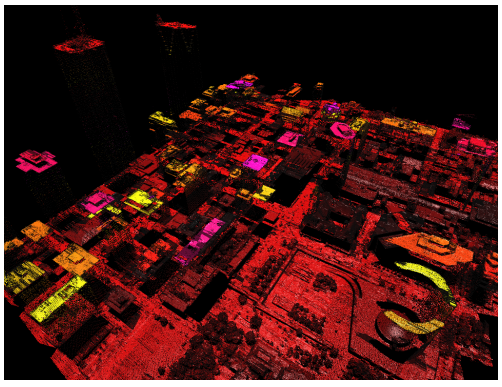

Figure 7: Urban area rendering with TerraForm. Only rooftops of buildings are visible, because laser beam is unable to properly sample vertical features.

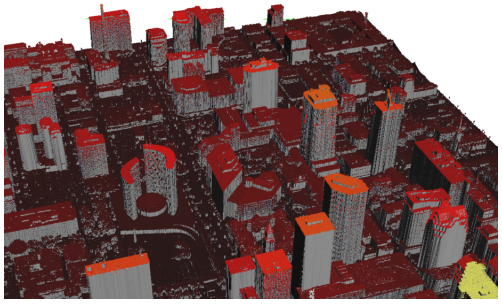

Figure 8: Polygonal rendering of a triangulated urban area rendered by Lasview application [2]

When compared with similar applications[1], the chosen approach appears to work best visually and performance-wise. Reviewed applications were mostly rendered using single pixel points, which amounted to high degrees of alias as well as a lot of holes, especially when viewing points from small distances. Advanced features such as dynamic splat sizes and shapes, blending and rendering of full resolution terrain were not implemented. Most of them have not implemented advanced point management system and were thus unable to display the full density of points, even at greater zoom levels. Some applications also feature an alternative polygon based rendering engine, which provides a hole-free rendering of triangulated points. Polygonal renderings of urban areas prove to be of much higher quality thanks to a lot of flat surfaces they contain, as seen in Figure 8. On the other hand, polygonal terrain renderings produced an unnatural looking surface with distorted vegetation and trees, as seen in Figure 9.

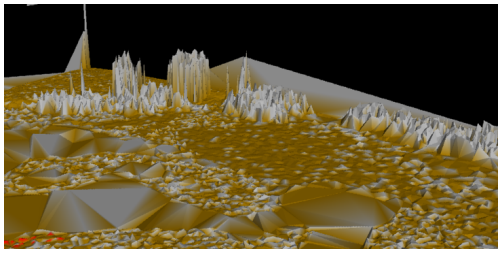

Figure 9: Polygonal rendering of a triangulated terrain area rendered by VG4D application [3]

7 Conclusions

In this paper we presented an approach to visualization of LIDAR data with a point-based rendering method. Our approach uses dynamic data management that enables rendering engine to operate with a full subset of points. Point density has to be lowered only on really old graphics cards, due the smaller amount of available onboard RAM. Since all rendering computations are delegated to the GPU, the central processing unit is available for other tasks. This is especially important while dealing with LIDAR data, where visualization often serves as a tool for analysis and further data manipulation.

While the resulting visual quality is really good when compared with similar applications, the nature of LIDAR data has quite an impact on it. Data gathered by older LIDAR systems can be quite sparsely sampled and therefore inappropriate for use with our system. Splats need to be very large to avoid subsequent holes. In some such cases, interpolation might be useful to fill the empty space for rendering purposes. A hybrid polygon and point based rendering engine, similar to the one proposed in [8], might prove to be a good solution to holes. Urban areas seem to be affected the most. The amount of flat surfaces they contain makes them ill suited for use with point-based rendering, which performs best on curved surfaces.

Splats are also not well suited for visualizing sharp features, such as building's edges and corners. Using elliptical Gaussians as point representations causes them to be soft. We could counter this by specifying one or two clipping lines for each sharp splat and clip the splat against them [13]. Another problem inherent to the LIDAR data is its error margin. While it encourages us to use fast approximations in our algorithms without significant visual penalties, anomalies that lower the output quality still occur. One such case is normal vector calculation, where too infrequently sampled points sometimes cause a splat to be rotated out of place.

Raw LIDAR data contains only information that can be gathered from the reflected laser beam, which does not contain any colour information. We were therefore limited to just a few colouring modes, such as colouring by point classification, elevation, intensity etc. To gain real visual appeal, we should think about associating LIDAR points with digital aerial photos. This would greatly improve visual quality and extend the usability of the application.

[1] Compared with: LPViewer, lasview, LP360 for ArcGis, VG4D, LViz

References

[1] Tao framework. [online] At `http://www.taoframework.com/`, 2007. (November 28, 2008).

[2] Las tools. [online] At `http://www.cs.unc.edu/~isenburg/lastools/`, August 2008. (November 28, 2008).

[3] Vg4d viewer. [online] At `http://www.virtualgeomatics.com/solutions4.html`, November 2008. (November 28, 2008).

[4] Mario Botsch, Alexander Hornung, Matthias Zwicker, and Leif Kobbelt. High-quality surface splatting on today's gpus. In *Eurographics Symposium on Point-Based Graphics*, pages 17–24, Stony Brook, NY, 2005. Eurographics Association.

[5] Mario Botsch and Leif Kobbelt. High-quality point based rendering on modern GPUs. In *11th Pacific Conference on Computer Graphics and Applications*, pages 335–343. Eurographics Association, 2003.

[6] Mario Botsch, Michael Spernat, and Leif Kobbelt. Phong splatting. In *Eurographics Symposium on Point-Based Graphics*, pages 25–32, Zürich, Switzerland, 2004. Eurographics Association.

[7] Baoquan Chen and Minh Xuan Nguye. Pop: A hybrid point and polygon rendering system for large data. In *IEEE Visualization*, pages 45–52, Los Alamitos, CA, USA, 2001. IEEE Computer Society.

[8] Liviu Coconu and Hans-Christian Hege. Hardware-accelerated point-based rendering of complex scenes. In *Proceedings of the 13th Eurographics workshop on Rendering*, pages 43–52. Eurographics Association, 2002.

[9] Airborne 1 Corporation. How lidar works. Technical report, 2006. [online] At `http://www.airborne1.com/technology/how_lidar_works.shtml`, (November 28, 2008).

[10] Carsten Dachsbacher, Christian Vogelgsang, and Marc Stamminger. Sequential point trees. In *ACM Transactions on Graphics*, volume 22, pages 657–662, New York, NY, USA, 2003. ACM.

[11] American Society for Photogrammetry and Remote Sensing. [online] At `http://www.lasformat.org/`, April 2008. (November 28, 2008).

[12] J. P. Grossman and William J. Dally. Point sample rendering. In *Rendering Techniques '98*, Eurographics, pages 181–192. Springer, 1998.

[13] Leif Kobbelt and Mario Botsch. A survey of point-based techniques in computer graphics. *Computers & Graphics*, 28(6):801–814, 2004.

[14] Marc Levoy and Turner Whitted. The use of points as display primitives. Technical report, University of North Carolina at Chapel Hill, 1985.

[15] Liu Ren, Hanspeter Pfister, and Matthias Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. In *Computer Graphics Forum (Eurographics 2002)*, volume 21, pages 461–470, September 2002.

[16] Szymon Rusinkiewicz and Marc Levoy. QSplat: A multiresolution point rendering system for large meshes. In *Proceedings of ACM SIGGRAPH 2000*, pages 343–352, July 2000.

[17] Miguel Sainz and Renato Pajarola. Point-based rendering techniques. *Computer & Graphics*, 28(6):869–879, December 2004.

[18] Matthias Zwicker, Hanspeter Pfister, Jeroen Van Baar, and Markus Gross. Surface splatting. In *Proceedings of ACM SIGGRAPH 2001*, pages 371–378, august 2001.

[19] Matthias Zwicker, Jussi Räsänen, Mario Botsch, Carsten Dachsbacher, and Mark Pauly. Perspective accurate splatting. In *Proceedings of Graphics Interface 2004*, pages 247–254, School of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2004. A K Peters.