DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

GPGPU Out-of-core Ray Tracing for unprocessed Airborne LiDAR Point Clouds

Jean Paul Vieira Filho

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics: Games Engineering

GPGPU Out-of-core Ray Tracing for unprocessed Airborne LiDAR Point Clouds

GPGPU Out-of-core Ray Tracing für unverarbeitete Airborne LiDAR Punktwolken

Author:	Jean Paul Vieira Filho
Supervisor:	Prof. Dr. Rüdiger Westermann
Advisor:	M.Sc. Henrik Masbruch
Submission Date:	15.03.2017

I confirm that this bachelor's thesis in informatics: games engineering is my own work and I have documented all sources and material used.

Munich, 15.03.2017 Jean Paul Vieira Filho

Abstract

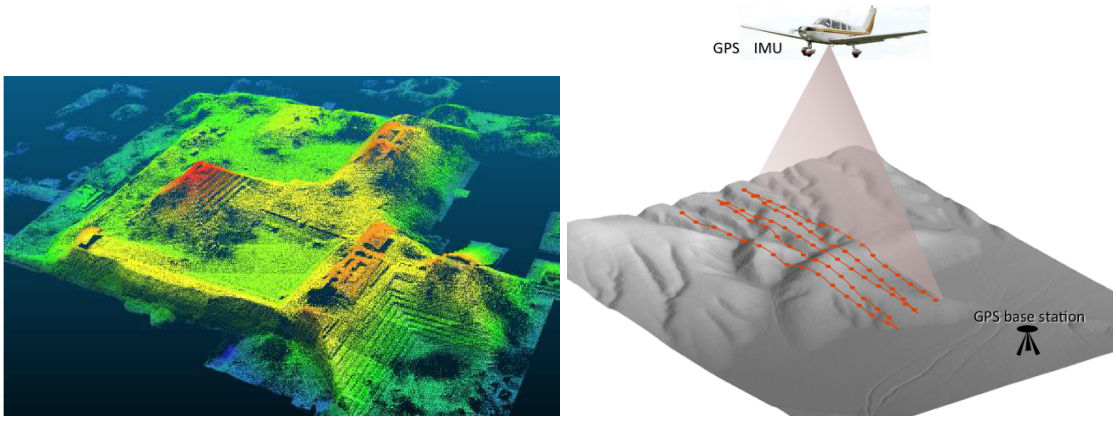
The primary purpose of this thesis is to experiment a visualization method of LiDAR Point Clouds larger than the amount of primary memory without recurring to any sort of data preprocessing. This allows flexibility when changing raw LiDAR files, e.g. adding new points, since no accelerating structure reconstruction is necessary. The proposed method uses CUDA C/C++ interoperability with OpenGL to provide an out-of-core, real-time, grid-based ray tracing engine to render the point cloud files. The ray tracing aspect occurs in the GPU, while the out-of-core and grid management parts are performed by the CPU. Furthermore, in each rendering frame a portion of the grid containing an amount of points around the camera is passed to the GPU and discarded after all pixel color values have been computed.
The results provided by this method show that it is possible to visualize raw LiDAR data at interactive framerates, with the view distance and frame rate limited by the resolution of the grid portion that is passed to the GPU.

Keywords: LiDAR, airbone laser scanning, ray tracing, point cloud, real-time, grid, unprocessed data

Contents

1 Introduction

Point clouds are increasingly being used as a representation method for topography data due to its high accuracy in current scanning technologies. However, rendering point clouds can be a challenging task for the current generation of graphics cards on account of the large amount of points and file size, which oftentimes outgrows the primary memory capacity in most computers. Therefore, a variety of data structures have been researched to visualize point clouds while streaming data from a slower secondary or tertiary memory and still maintain acceptable frame rates for real-time applications.



(a) LiDAR Point Cloud [Von+16] (b) Airborne LiDAR scanning [Nis15]

Figure 1.1

1.1 LiDAR and .LAS file

LiDAR, which stands for *Light Detection and Ranging* [Lun13], is a popular technology that enables the acquisition of an object's information, such as range, through the scattering of light. Based on it, instruments have been developed to allow airborne topography laser scanning, commonly called *Airbone LiDAR*, capable of mapping more

than hundred square kilometers a day [Sla+07]. As a result, point clouds are generated containing information per point such as: X,Y and Z coordinates w.r.t. an origin, Color values, among others.

The simplest way to store these point clouds is in an ASCII file. Nevertheless, the *American Society for Photogrammetry and Remote Sensing* (ASPRS) provides the more efficient LAS public file format to facilitate LiDAR point data exchange [ASP11], which can be read by the open source *libLAS: ASPRS LAS LiDAR Data Toolkit.* [Ise+16].

This thesis was developed using libLAS in order to read Airborne LiDAR point clouds in .LAS format.

1.2 CUDA and OpenGL

Compute Unified Device Architecture (CUDA)[NVi] is a *General-purpose computing on graphics processing units* (GPGPU) programming language developed by *NVidia*. It allows programmers to use the company's proprietary hardware for parallel computing without any knowledge of a graphics API. Nonetheless, in this experiment OpenGL is used, a widely known graphics API, combined with CUDA, in order to display the rendered image on the screen. This is possible by using the CUDA-OpenGL interoperability [Wil13, Chapter 3.10], which allows manipulation of OpenGL's objects, such as buffers and textures, from inside the CUDA environment.

1.3 Ray Tracing and Grids

Ray Tracing is a rendering process in which light rays trajectory is simulated. The *Pinhole Camera Model* [Gla89], for example, traces a ray backwards, i.e. from a camera, through pixels in an image and towards scene objects and/or light sources. In comparison to Rasterization, where shapes are projected onto pixels, Ray Tracing is far more computational intensive since intersection has to be calculated against all objects in the scene. To mitigate this disadvantage, Fujimoto, Tanaka and Iwata [FTI86] proposed a method called *3D Digital Differential Analyzer* (3DDDA) in which the objects are discretized into a 3-Dimensional grid and the pixel rays are casted through it until a filled cell is intersected.

Other methods include tracing rays in continuous space [MB90], where 3D objects are used instead of grids, and rays with thickness [HH84], where an area intersection is calculated instead of infinitesimally thin rays.

This research is based on a more recent study performed by Dick, Krüger and Westermann [DKW09], which uses a similar technique to [FTI86] with a 2D grid of height values that is more suitable for terrain visualization.

1.4 Related Work

Existing research to visualize point clouds rely on different structures to accelerate the Ray Tracing process. Balciunas, Dulley and Zuffo [BDZ06] proposed a method using a Bounding Volume triangle mesh that encloses the higher resolution height field. Combined with Rasterization performed in the GPU, each ray can start almost intersecting with the geometry and thus be performed in parallel by the CPU in a Pipeline.

Dick, Krüger and Westermann [DKW09] provided a hybrid Raster/Ray Trace method to visualize large height maps composed of tiles containing each a 2D height field organized in a Quadtree. Because of this structure, only points that can appear in the screen are loaded, the Ray Tracing only occurs inside of areas where an intersection might occur and large spaces can be skipped faster.

Similarly, an Octree structure is used by Richter and Döllner [RD10] to group points together, thus allowing out-of-core by specifically loading currently visible portions of the data and faster projection on the screen using Rasterization.

On a different field, Fogal, Schiewe and Krüger [FSK13] presented a method of Volume Rendering in which the data, in this case not necessarily a point cloud, is spatially divided into bricks that are loaded into the GPU on-the-fly.

In contrast to the previously described approaches, this thesis focuses on raw point cloud data filling a Quadtree structure, based on the Maximum Mipmaps method introduced in [TIS08]. The ray tracing is built on the algorithm presented by [DKW09] and a dynamic grid allocation process inspired by [FSK13].

1.5 Outline

2 **Method**: this chapter presents the problem studied, the main research questions and subquestions, and the proposed solution for the problem.

3 **Implementation**: this chapter explains the data structure used and describes how the rendering process works.

4 **Analysis**: this chapter contains the results of the used method, along with comments and benchmarks. Furthermore, drawbacks are also presented

5 **Conclusion**: this chapter discusses the results achieved and the complications found during the research. It also describes limitations of the proposed solution.

6 **Outlook**: this chapter presents new research questions that were not answered in this project.

2 Method

This chapter defines the theoretical framework and gives an insight into the proposed solution.

2.1 Problem

Current technology to visualize point clouds relies on preprocessed data structures. Although its methods provide better performance when rendering, they usually require hours or even days to build a structure, depending on the point cloud size. Furthermore, adding or removing data often requires complete or partial reconstruction of the data structure.

2.2 Research Questions and Goal

This research answers the following main questions:

- How can unstructured LiDAR point cloud data be visualized?
- How can an acceptable real-time frame rate be achieved?

And the subquestions:

- How can the Ray Tracing process be optimized?
- What discretization structure can be used?
- How can the data be streamed in real-time from the secondary memory?

2.3 Proposed Solution

This research presents a method that employs an array of multi-level grid to stream the unstructured LiDAR data into. From this array, a subsection is sampled and passed to GPU for Ray Tracing in each frame. The results are stored in a texture, which is then displayed on the screen. Furthermore, each part of the array is created or destroyed based on the camera position during runtime through multi-threading.

3 Implementation

This chapter describes the method that was developed in this thesis. It includes the data structure used and the detailed pipeline implementation.

Figure 3.1 shows an overview of the main rendering loop and the steps that occur before entering it. The program starts by allocating the data structure and starting I/O threads. These threads read points from the .LAS file and store the height and color values into the data structure. The main thread then starts, parallel to the I/O threads, the rendering loop. First, it handles the camera rotation and translation. Then, if necessary, it rearranges the data structure by loading and removing portions of it based on camera position. Afterwards, it fills a buffer with data close to the camera and sends it to the GPU. The main thread then starts GPU ray tracing threads and waits for the result. After receiving the pixel color values from the GPU, the main thread copies them into a texture and maps it to a Quad, which covers the entire viewport. It then starts the rendering loop again.

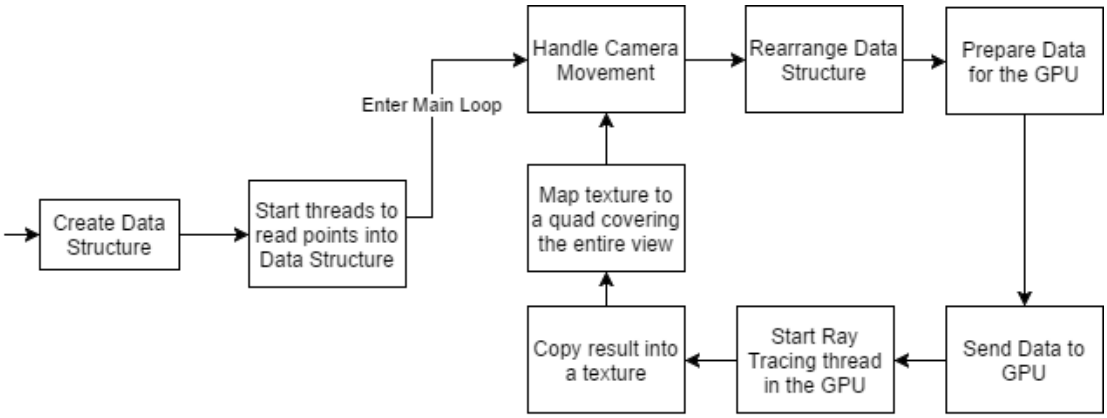


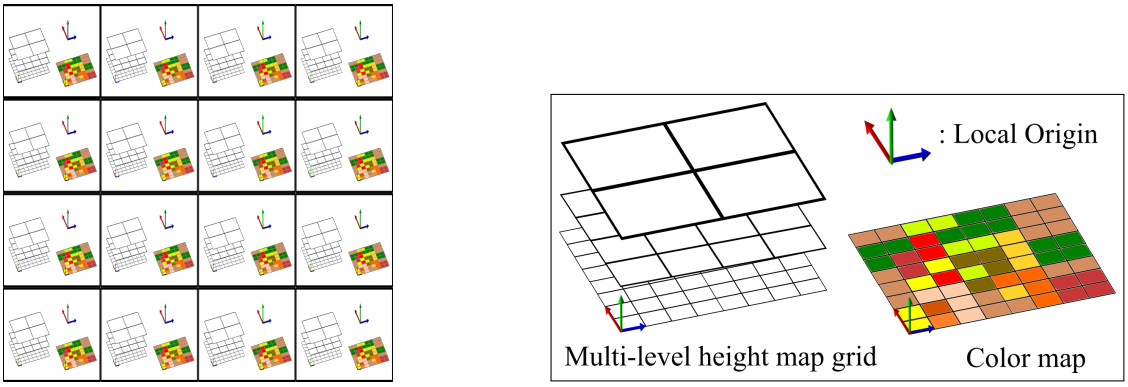
Figure 3.1: The main thread execution steps and rendering loop

Main thread: this pseudo-code represents the main rendering thread and the rendering loop

```
allocate_memory()
initialize_parameters()
initialize_setions()
while(render)
        move_camera();
        rotate_camera();
        manage_sections();
        extract_section();
        start_CUDA_ray_trace();
        synchonize_with_GPU();
        display_texture();
endwhile
free_memory()
```

3.1 Data Structure

This section describes the data structure used by the researched system. It is composed of an object called "section", two buffers for components of this object located in the CPU and GPU memory, and two OpenGL objects: a texture and a pixel buffer object.



(a) 4x4 Section Array

(b) Components of a section

Figure 3.2: The data structure is composed of a 4x4 2D pointer array (a), whose cells point to what is termed as sections in this experiment. Each section (b) has a height map, a color map and a local origin.

3.1.1 Sections

A section (Figure 3.2b) is composed of a height map, a color map and a local origin. The local origin is a 3D coordinate assigned to its left lower corner, which represents its position relative to the scene. Additionally, cells in the height maps across all sections have the same fixed size that can be set at initialization.

The height map is a multi-level grid arranged in a Quadtree structure, where each cell in the coarsest level is split in 4 equally sized cells in the next finer grid [DKW09]. The grid levels are represented by variable l, where 0 is the finest level and $l_{max} \geq 0$ is the coarsest level. The color map is a 2D grid that has the same resolution as the finest level of the height map and stores RGB values.

A 4x4 section pointer array (Figure 3.2a) is used to provide out-of-core functionality, and 2 section buffers, called camera sections, are additionally allocated, one in the CPU memory and another in the GPU, for the ray tracing process.

3.1.2 Section Buffer

As mentioned in 3.1.1, a section buffer is allocated on the CPU-side. The whole data is firstly copied into the CPU buffer and then transferred to the GPU buffer using only two memory copy calls per frame: one for the height map and another for the color map. During implementation, this process was compared to having multiple memory copy calls. The first approach provided faster speeds when transferring data between the GPU and CPU.

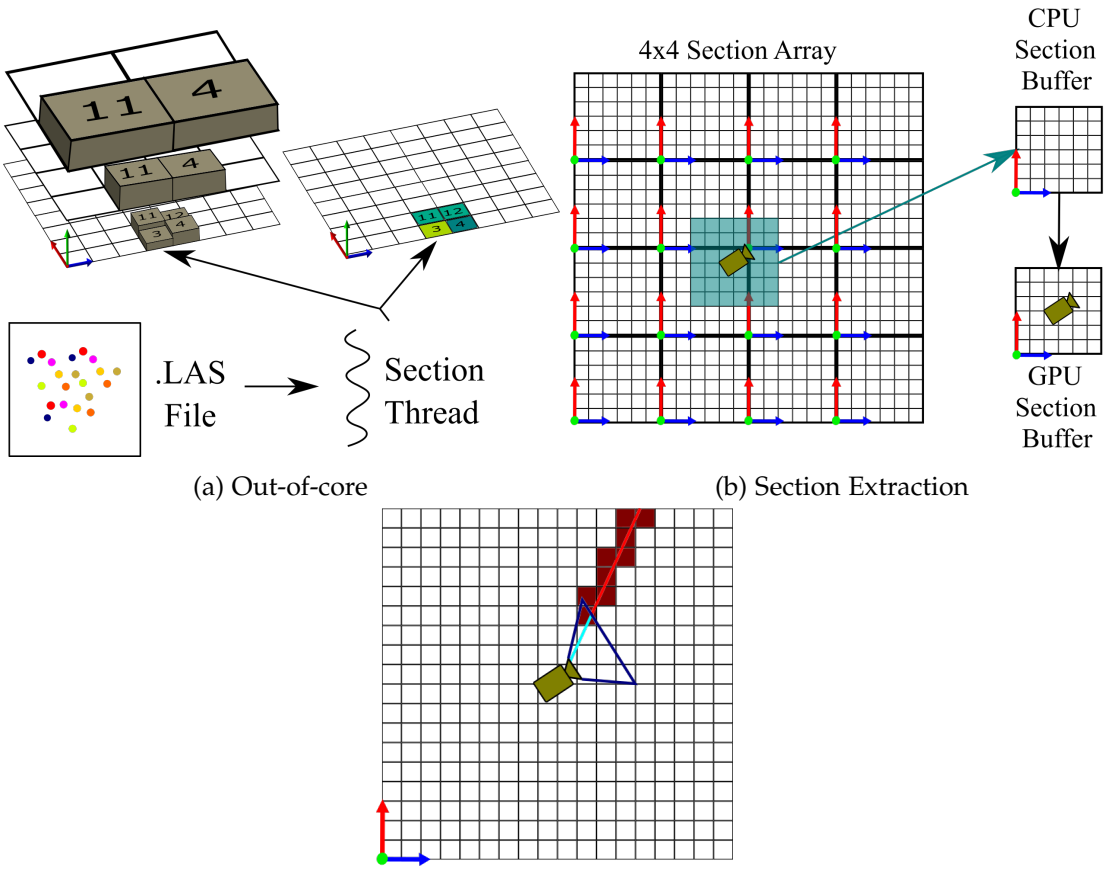
3.1.3 Texture and Pixel Buffer Object

A texture and a pixel buffer object (PBO[1]) are created to display the results after the ray tracing process is complete. The PBO holds the color information for each pixel per frame and the texture is mapped to a quad that covers exactly the entire view frustrum.

[1] A pixel buffer object in OpenGL is a buffer used for pixel transfer operations - see `https://www.khronos.org/opengl/wiki/Pixel_Buffer_Object`

3.2 Pipeline

This section describes the system that operates on top of the presented data structure.



(a) Out-of-core (b) Section Extraction

(c) Ray Tracing

Figure 3.3: The rendering pipeline has 3 major components: the Out-of-core (Figure a), which spawns threads that read the point cloud file and store values into sections, the Section Extraction (Figure b), where the camera section is created from different sections and passed to the GPU, and the Ray Tracing (Figure c), in which the image to be displayed is rendered.

3.2.1 Out-of-core

The allocation of each section is handled by the main thread, based on the camera position. At initialization, the camera is assumed to be in the center of the sections

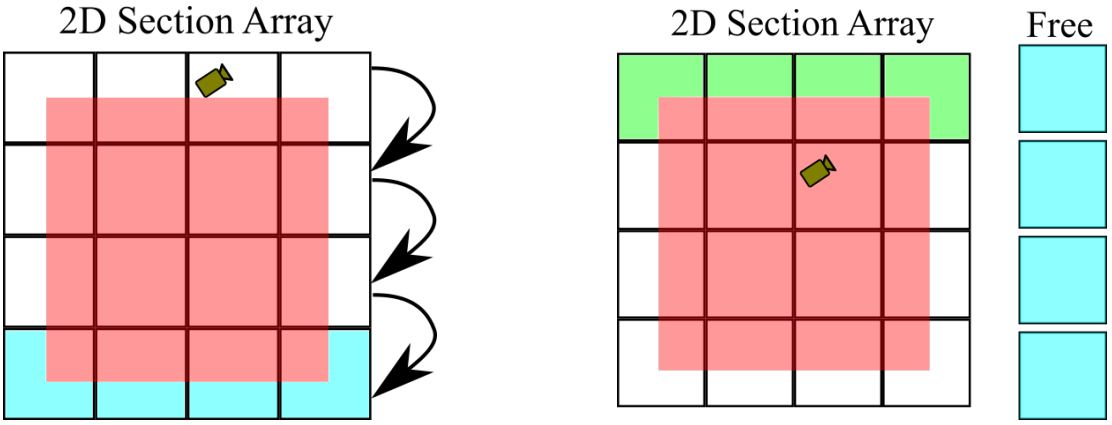
array.

A thread is then spawned and assigned to each section. Their task is to fill a section with points that are located inside its grid. In order to achieve that, each thread searches linearly through the point cloud data. This approach may not be the most efficient way to search for points. However, the points inside a .LAS file are not necessarily aligned according to their position. Furthermore, this thesis is based on the fact that no data structure exists. Even though providing a highly efficient search algorithm for this problem is not in the scope of this research, it may be an interesting topic to study in the future.

If a point is located inside the grid, its index I is calculated for each grid level l using the formula: given a 3D point position Pos and a .LAS file lower bounding box vertex Min:

$$I^l_{x/y} = \frac{\left\lfloor \frac{Pos_{x/y} - Min_{x/y}}{CellSize_{x/y}} \right\rfloor}{2^l} \text{ where } l = 0, ..., l_{max}$$

The height value $(Pos_z - Min_z)/CellSize_z$ is then inserted in the grid from finest to coarsest level using the Maximum Mipmap method from [TIS08]



(a) Each row moves down one position and blue sections are marked to be freed

(b) New green sections are created at the top row and old sections are freed in parallel

Figure 3.4: If the camera moves outside of a safe area, the cells on the opposite side are marked to be deallocated and removed from the grid (Figure a). Each row or column is then moved in the direction of the removed cells. New sections are allocated at the now free cells and the old ones are deallocated in parallel (Figure b).

During the main loop, the camera will most likely move outside of the area covered by the current sections (Fig. 3.4a). To prevent that, the main thread rearranges the 2D section array. The sections farther away from the camera are removed from the array and set to be deallocated (Fig. 3.4b). Any threads running in sections that are marked to be deallocated will stop searching for new points and free the memory area of its height and color maps.

Section thread: this pseudo-code represents a thread reading points from the .LAS file into a section

```
1  open_point_file()
2  foreach (point in point_file)
3          if(point_is_in_section)
4                  insert_point_in_height_map();
5                  insert_color_in_color_map();
6          endif
7          if(terminate_thread)
8                  exit foreach;
9          endif
10 endforeach
11 close_point_file()
12 while(!terminate_thread)
13         yield;
14 endwhile
15 free_section_memory();
```

3.2.2 Section Extraction

To perform the ray tracing, a camera section is passed every time to the GPU. For this purpose, a CPU-side buffer is allocated as mentioned in 3.1.2. The problem is that this camera section is not aligned with those from the 2D section array. Consequently, it needs partial data from multiple sections, as illustrated in Figure 3.3b.

First, the section and its coarsest cell index in which the camera section's origin is located must be defined. Afterwards, all elements inside the top right quadrant relative to the cell index are copied. If the camera section is not yet completely filled, the remaining elements from the right/top right/top neighbors need to be copied. After the buffer is filled, it is passed to the GPU for the Ray Tracing part.

3.2.3 Ray Tracing

Additionally to the camera section other parameters are needed. Through the use of CUDA-OpenGL interoperability [NVi, Chapter 3.10], a mapped referenced to the PBO (Chapter 3.1.3) is passed to the GPU. Further parameters are: the camera forward direction and position, frame dimensions and resolution, and whether the image should be rendered with the color map or height based colors.

A preparation step is performed by calling a single-threaded kernel[2] to store the ray tracing parameters into global variables in the GPU. Afterwards, the ray tracing CUDA kernel is called with exactly a thread per pixel in the frame.

For each thread, the ray starting position is calculated according the *Pinhole Camera Model* [Gla89, Chapter 1] in camera space. This position is then transformed from camera to section space by using a basis change matrix and used as the ray direction R_{dir}.

The ray tracing algorithm used was adapted from [DKW09]. The part that was modified for this thesis is the color sampling when the ray intersects. Instead of retrieving the values from a texture, the color is either calculated at runtime (Fig. 4.5b) or copied from the camera section's color map. The implementation is as follows:

1. **Adjust paremeters:**
   a) The only case handled is $R_{dir_x} > 0$ and $R_{dir_y} > 0$. All other cases can be converted to the previous by inverting the direction, e.g. $R_{dir_x} = -R_{dir_x}$, and mirroring the indexes of the height and color map

2. **Main loop:**
   a) Calculate ray exit point R_{exit} and $Edge$ based on the current ray position R_{pos}:

   The distances Δt_x and Δt_y from R_{pos} to the next axes intersection in a grid level l is defined as

$$\Delta t_{x/y} := \frac{\left(\left\lfloor \frac{R_{pos_{x/y}}}{2^l} \right\rfloor + 1\right) \cdot 2^l - R_{pos_{x/y}}}{R_{dir_{x/y}}}$$

   The closest intersection is thus $\Delta t = min\{\Delta t_x, \Delta t_y\}$ and $R_{exit} = R_{pos} + \Delta t \cdot R_{dir}$.

   To prevent roundoff errors, the coordinate of the intersected edge is explicitly set. Furthermore, the edge index must be calculated for the level

[2] "The GPU executable code..." [NVi, Chapter 3.5]

change to work.

$$R_{exit_a} = \left(\left\lfloor \frac{R_{pos_a}}{2^l} \right\rfloor + 1 \right) \cdot 2^l \text{ and } Edge = \left\lfloor \frac{R_{pos_a}}{2^l} \right\rfloor \text{ where } a = \begin{cases} x, & \text{if } \Delta t = \Delta t_x \\ y, & \text{otherwise} \end{cases}$$

b) Test for intersection:

First, the grid position needs to be calculated $G_{pos} = \left(\left\lfloor \frac{R_{pos_x}}{2^l} \right\rfloor, \left\lfloor \frac{R_{pos_y}}{2^l} \right\rfloor \right)$.
Let $Height(G_{pos}, l)$ be the function that gives the height value of a grid position at level l. An intersection occurs if and only if one of the two conditions are met:

[1] $R_{dir_z} \geq 0$ and $R_{pos_z} \leq Height(G_{pos}, l)$

[2] $R_{dir_z} < 0$ and $R_{exit_z} \leq Height(G_{pos}, l)$

c) If there is an intersection:

If the current level $l > 0$ then $l = l - 1$, otherwise retrieve the color value from the color map at position G_{pos} and $l = 0$ or calculate the color based on $Height(G_{pos}, 0)$

d) If no intersection occurs:

The ray position moves to the ray exit and if the ray leaves the current level 2x2 square, l is increased by 1:
$R_{pos} = R_{exit}$ and $l = min\{l + 1 - (Edge \bmod 2), l_{max}\}$

e) If the ray leaves the bounding box of the camera section, break the loop and return a background color

CUDA Ray Tracing thread this pseudo-code represents the ray tracing process performed by a CUDA thread

```
calculate_ray_position();
calculate_ray_direction();
color_value = color(0,0,0);
while(ray_is_inside_boundaries and !exit_loop)
        calculate_exit_point_and_edge();
        if(is_intersection())
                if (level > 0)
                        level = level - 1;
                else
                        exit_loop = true;
                        if (use_color_map)
                                color_value = get_color_map_value();
                        else
                                color_value = calculate_height_color();
                        endif
                endif
        else
                calculate_new_level();
                ray_position = ray_exit;
        endif
endwhile
pixel_buffer_object[position_x][position_y] = color_value;
```

3.3 Improvements

With each implementation step new complications were introduced. These are listed in the following subsections:

3.3.1 Linear Memory Allocation

As explained in 3.2.2, in each frame the section buffer is copied to the GPU-side camera section. To speedup the process and avoid multiple memory copy calls between the CPU and GPU, the section's multi-level grid is allocated linearly. This requires only one memory copy call to transfer the data into the GPU. At initialization, the offset for each level of the grid is calculated once and used during rendering to read and write height values.

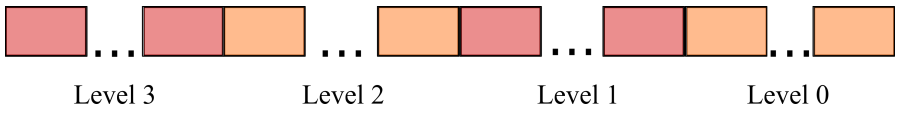


Figure 3.5: A representation of a 4-Level Grid allocated linearly in memory

3.3.2 Ray Coherence

In CUDA, threads inside a *block*[3] of a kernel are grouped together in a *warp*[4]. If a branch divergence occurs, some threads are locked until the others converge into the same instruction again.

To reduce the chances of a branch divergence happening, the camera roll axis is locked and the vertically aligned rays are grouped together in a thread block. Since the rays visit the same cells of a section's grid most of the time, the effectiveness of caching might also have improved.

3.3.3 Smooth Camera Movement

After the camera section is created (Chapter 3.2.2), the camera is assumed to be at the exact center of it. This, however, provides a rough movement through the scene. The camera would "jump" from the center of a grid cell to another.

To correct this, the x and y position $\vec{C}_{section}$ of the camera inside its section is calculated and the ray tracing is performed based on it. Given the camera section origin $\vec{O}_{world}$,

[3] A group of threads assigned to and executed by the same CUDA Core [NVi, Chapter 2.6]
[4] A group of 32 threads that always execute the same instruction at a time [NVi, Chapter 7.3].

the section's origin $\vec{S}_{world}$ in which $\vec{O}_{world}$ is in and the sections' grid coarsest resolution r.

$$C_{section_{x/y}} := O_{world_{x/y}} - S_{world_{x/y}} - \left\lfloor \frac{O_{world_{x/y}} - S_{world_{x/y}}}{2^{l_{max}}} \right\rfloor \cdot 2^{l_{max}} + (r-1) \cdot 2^{l_{max}-1}$$

3.3.4 Height Colors

The color properties of a point were only introduced at LAS version 1.2 [ASP11]. To support older .LAS files, the possibility to visualize data based on the point height was also implemented. Blue is assigned to the lowest height, yellow to half the maximum height and red to the maximum height. Any value located between these intervals are then interpolated. Given a height h, a max height h_{max} and $h_t := 2 \cdot \frac{h}{h_{max}}$, the color values red r, green g and blue b are calculated:

$$r = \begin{cases} 255 & \text{if } h_t \geq 1 \\ 255 \cdot h_t & \text{otherwhise} \end{cases}$$

$$g = \begin{cases} 255 - (h_t - 1) \cdot 255 & \text{if } h_t \geq 1 \\ 255 \cdot h_t & \text{otherwhise} \end{cases}$$

$$b = \begin{cases} 0 & \text{if } h_t \geq 1 \\ 255 - h_t \cdot 255 & \text{otherwhise} \end{cases}$$

A clearer example can be seen in Figures 4.5b and 4.5a

3.3.5 Thread Priority

Since the experiment uses a multi-threaded approach, some complications were introduced. During the testing phase, the main thread would oftentimes not receive any execution time from the thread scheduler, thus freezing the rendering until a section thread had finished loading points. Furthermore, sections farther away from the camera would be loaded first than those in which it was situated.

To solve this problem, each thread priority is set manually based on Microsoft specifications [Mic]. The main thread's priority is set to 2 (highest priority), the 4 section threads in the middle of the 2D Section Array are set to 0 (normal priority) and the rest is set to -2 (lowest priority).

4 Analysis

This chapter discusses the results achieved in this research, a benchmark of the calculation time for the major parts of the frame rendering and the visual drawbacks of the proposed method.

4.1 Results

The computer in which this method was tested was equipped with an Intel i7-3770K 3,50GHz, 16GB of RAM and a NVidia GeForce GTX 970.

The following images were sampled at a resolution of 1920x1080, with sections having a 32x32 grid with 8 levels. The system was able to visualize datasets at an average of 18 frames per second, with camera navigation.

The datasets used were the Autzen Stadium, provided by [Ise+16], and the Utah Wasatch Front, provided by [Cen14].

Figure 4.1: Farmland with 0.25 cell size - Utah Dataset

Houses, roads, vehicles, farmlands and water bodies can be recognized, as seen in Figures 4.1 and 4.2. Therefore, the proposed method can be used to visualize and navigate *Airbone LiDAR* point clouds.

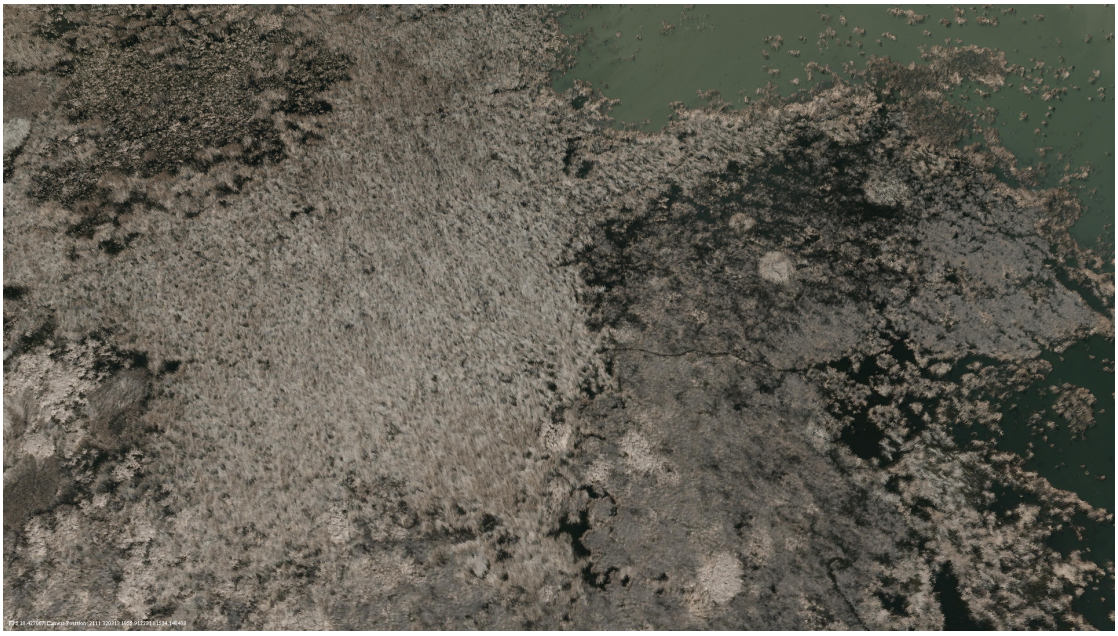
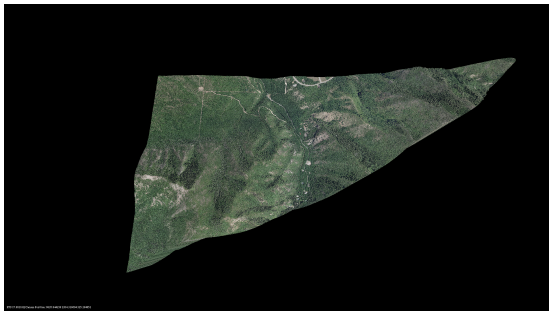


Figure 4.2: Swamp area with 0.25 cell size - Utah Dataset

(a) Mountain area with 0.25 cell size - Utah Dataset



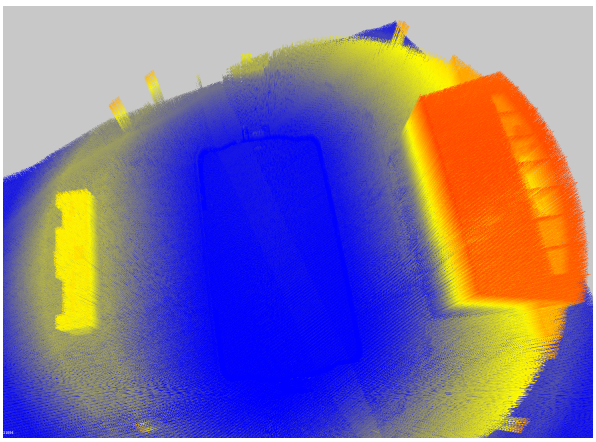
(b) Mountain area with 2.00 cell size - Utah Dataset

Figure 4.3

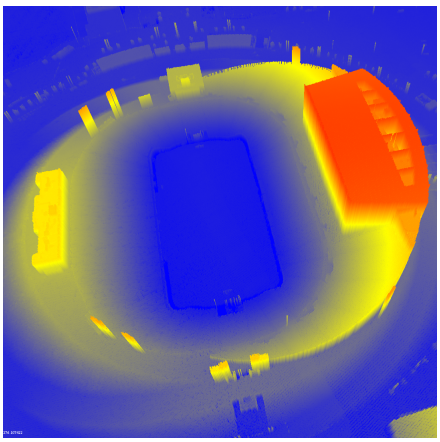
Comparing Figures 4.3a and 4.3b, it can be seen that changing the finest level cell size of the sections' grid allows larger or smaller extents of terrain to be seen. The

major restrictions provided by this method, however, are the level of detail and the grid size at the finest level. The smallest detail in the visualization is the cell itself (a single column). With a cell size of 0.25 units, the largest extent that can be seen in a frame is 1024 units (Fig. 4.4).

Figure 4.4: Top-down view of a 32x32 grid with 0.25 cell size - Utah Dataset



(a) Stadium with 0.25 cell size - Autzen Dataset



(b) Stadium with 1.00 cell size - Autzen Dataset

Figure 4.5

The point cloud density should be considered when setting a cell size. Figure 4.5a of the Autzen Stadium dataset shows empty space between points, thus reducing the quality of the visualization. A more optimal value for the cell size in this particular dataset is 1.0 (Fig. 4.5b)

4.2 Benchmark

The data presented in Figure 4.6 and Table 4.1 are wall clock mean times per frame, collected from random camera paths for over 20 seconds each in the experiment. Ray tracing refers to Chapter 3.2.3, while Prepare Section Buffer and Copy Section Buffer refer each respectively to the second and last paragraphs of Chapter 3.2.2.

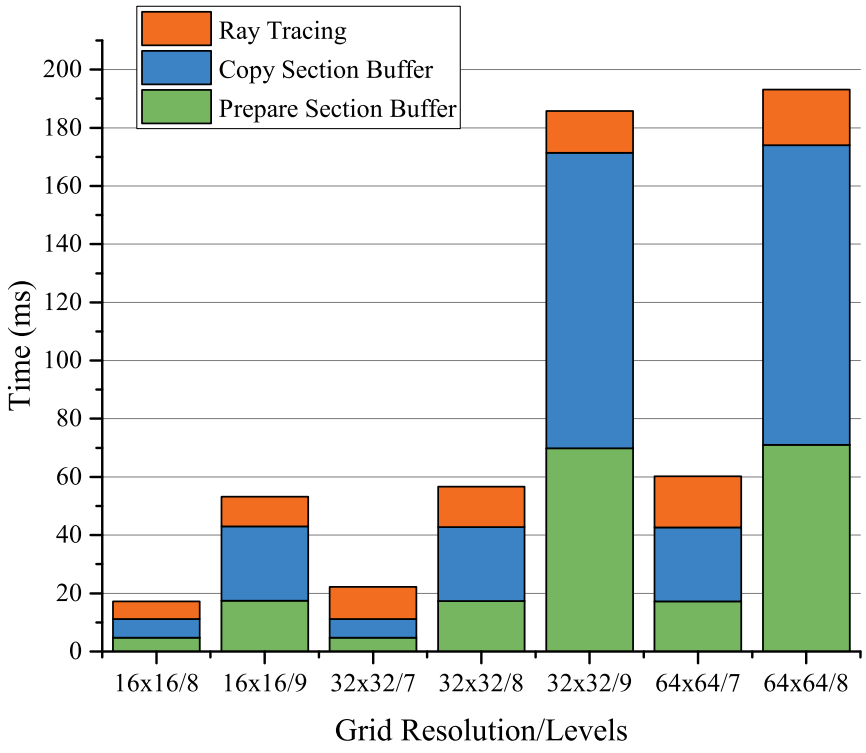


Figure 4.6: Calculation wall clock mean time per frame

Grid Resolution/Levels	Prepare Section Buffer (ms)	Copy Section Buffer (ms)	Ray Tracing (ms)
16x16/8	4.71	6.49	5.98
16x16/9	17.48	25.48	10.29
32x32/7	4.72	6.48	11.06
32x32/8	17.35	25.41	13.88
32x32/9	69.89	101.52	14.33
64x64/7	17.22	25.39	17.61
64x64/8	71.00	102.96	19.19

Table 4.1: Calculation wall clock mean time per frame

The Ray Tracing presented a high variation, most likely due to the disparity of landscapes being rendered at different times [DKW09], e.g. Swamp area (Fig 4.2) and Flat farmland (Fig. 4.1). Prepare and Copy Section buffer, on the other hand, had constant values with low deviation over time because the amount of data being copied in each frame is also constant. This can be seen when comparing the times of the **16x16/9**, **32x32/8** and **64x64/7** configurations.

It can be concluded that the greatest factor influencing the framerate is the size of the section being copied over in each frame. The Ray Tracing calculation time did increase slightly with larger sections but not as much when compared to the other 2 counterparts. Therefore, the rendering speed is bound firstly to the bandwidth between CPU and GPU memory, and secondly to the memory copy process inside the RAM.

4.3 Drawbacks

The data structure and ray tracing method used have also downsides. They cannot represent complex objects (Fig. 4.7) and empty spaces (Fig. 4.8)

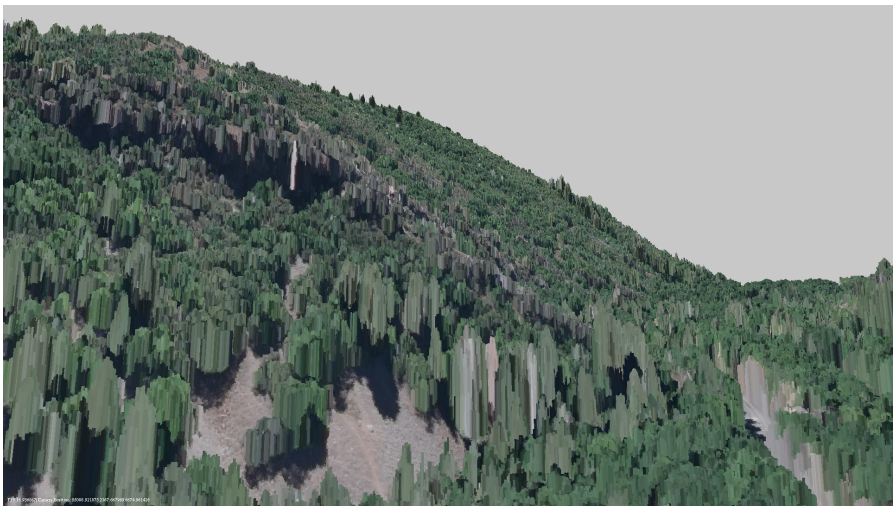


Figure 4.7: When the camera is close to the height map, single columns can be noticed and the objects might become difficult to recognize

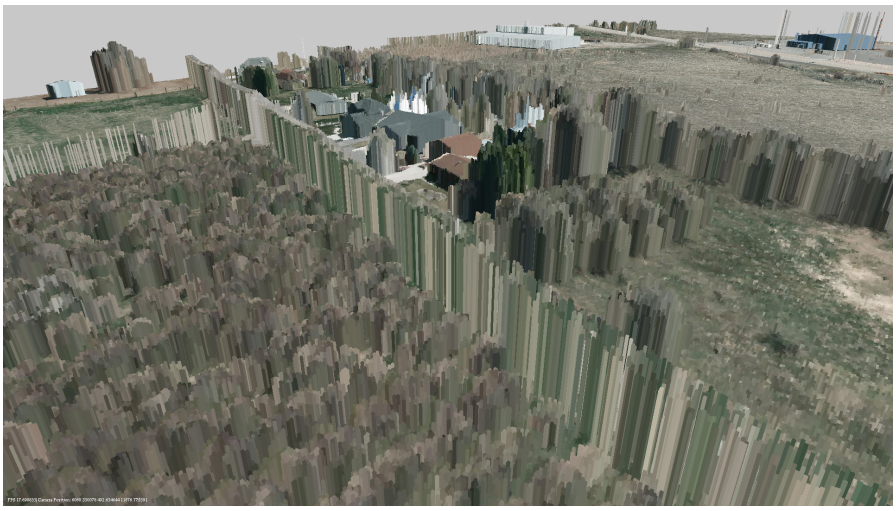


Figure 4.8: Empty space under power lines cannot be drawn in this model

5 Conclusion

This thesis achieved its goal of providing a method to visualize unstructured LiDAR point clouds at interactive framerates. The development process was as follows:

Firstly, a grid ray tracing system for ASCII point clouds using CUDA was developed based on the *3DDDA* algorithm [FTI86]. It provided a simple method to visualize points based on their height at a resolution of 640x480. Then, it was further optimized for height maps based on the algorithm provided by [DKW09]. Afterwards, the libLAS library [Ise+16] was introduced in order to read .LAS files efficiently. The grid structure was still being loaded entirely into the GPU at this point and tests were performed using the Autzen Stadium dataset [Ise+16]. Camera movement was then introduced to navigate through the dataset. Following series of tests and refactoring steps, the out-of-core functionality was added with the section system. Lastly, the point color property visualization was implemented and tested with the Utah Wasatch Front dataset [Cen14].

The result was a good quality visualization process for .LAS files larger than the primary memory available, alongside an implementation of a fast ray tracing algorithm, a self-managing out-of-core data structure based on grids and a simple camera navigation.

However, there are still many aspects of it which can be improved. The major limitation in this method is the necessary section extraction in each frame (Chap. 3.2.2). New technologies might improve the interface speed between graphics cards and primary memory, though the best solution to this problem is most probably to compress the data and/or prevent having to copy in the first place. Further limitations are the lack of an interpolation function to smooth the close-up rendering (Fig. 4.7), incorrect representation of geometry with empty spaces underneath it (Fig. 4.8) and relatively small total visible area when compared to other existing methods (Fig. 4.4).

In conclusion, rendering unstructured raw point data directly has the advantage of reducing the preprocessing time for the visualization. Nevertheless, it also introduces new problems such as the need of elaborated real-time data structures.

6 Outlook

After finishing the experiment, new research questions arose that could further improve the resulting rendering system. These questions are described in the following sections.

6.1 Multiple sections in the GPU

One possible way to avoid copying a section in each frame is to implement a similar system to the one mentioned in Chapters 3.2.1 and 3.1.1 on the GPU side. This would then lead to sections being copied to the GPU only when needed, instead of every frame. The expected outcome would be the possibility of ray tracing larger sections, thus having more terrain data available to visualize.

6.2 Indexing and reading multiple .LAS files

The .LAS format has limitations in the amount of points that it can hold. Therefore, datasets that are close to Terabytes of file size cannot be stored in a single file. A system to organize and read from multiple .LAS files could remove this limitation and additionally improve the point search used in this method.

6.3 Geometry with Empty Space Underneath

Having the height values stored in a stack inside the multi-level grid cell might introduce the possibility to recognize empty space underneath objects. This would be possible by comparing the height levels inside a stack to check if the ray is currently in an empty space or not. It probably would eliminate the drawback mentioned in Figure 4.8.

List of Figures

List of Tables

Bibliography

[ASP11] ASPRS. *LAS Format Specification 1.4*. The American Society for Photogrammetry Remote Sensing. Nov. 2011. URL: http://www.asprs.org/wp-content/uploads/2010/12/LAS_1_4_r13.pdf (Accessed: 4 Mar. 2017).

[BDZ06] D. A. Balciunas, L. P. Dulley and M. K. Zuffo. 'GPU-assisted ray casting of large scenes'. In: *Interactive Ray Tracing 2006, IEEE Symposium on*. IEEE. 2006, pp. 95–103.

[Cen14] U. A. G. R. Center. *Wasatch Front LiDAR Elevation Data*. 2013–2014. URL: https://gis.utah.gov/data/elevation-terrain-data/2013-2014-lidar/.

[DKW09] C. Dick, J. H. Krüger and R. Westermann. 'GPU Ray-Casting for Scalable Terrain Rendering.' In: *Eurographics (Areas Papers)*. Citeseer. 2009, pp. 43–50.

[FSK13] T. Fogal, A. Schiewe and J. Krüger. 'An analysis of scalable GPU-based ray-guided volume rendering'. In: *Large-Scale Data Analysis and Visualization (LDAV), 2013 IEEE Symposium on*. IEEE. 2013, pp. 43–51.

[FTI86] A. Fujimoto, T. Tanaka and K. Iwata. 'Arts: Accelerated ray-tracing system'. In: *IEEE Computer Graphics and Applications* 6.4 (1986), pp. 16–26.

[Gla89] A. Glassner. *An Introduction to Ray Tracing*. Academic Press. Academic Press, 1989. ISBN: 9780122861604.

[HH84] P. S. Heckbert and P. Hanrahan. 'Beam tracing polygonal objects'. In: *ACM SIGGRAPH Computer Graphics* 18.3 (1984), pp. 119–127.

[Ise+16] M. Isenburg, H. Butler, M. Loskot, P. Vachon, F. Warmerdam, M. Rodriguez, O. M. Rubi and R. Goncalves. *libLAS: ASPRS LAS LiDAR Data Toolkit*. 2007–2016. URL: https://www.liblas.org/ (Accessed: 4 Mar. 2017).

[Lun13] P. B. Lundquist. *Light detection and ranging*. US Patent App. 13/916,081. June 2013.

[MB90] J. D. MacDonald and K. S. Booth. 'Heuristics for ray tracing using space subdivision'. In: *The Visual Computer* 6.3 (1990), pp. 153–166.

[Mic] Microsoft. *Scheduling Priorities*. URL: https://msdn.microsoft.com/en-us/library/windows/desktop/ms686277(v=vs.85).asp (Accessed: 11 Mar. 2017).

[Nis15] E. Nissen. *Airborne LiDAR*. Colorado School of Mines. Dec. 2015. URL: http://serc.carleton.edu/download/images/83475/airborne_lidar.png (Accessed: 11 Mar. 2017).

[NVi] NVidia. *CUDA*. URL: http://www.nvidia.com/object/cuda_home_new.html (Accessed: 5 Mar. 2017).

[RD10] R. Richter and J. Döllner. 'Out-of-core real-time visualization of massive 3D point clouds'. In: *Proceedings of the 7th International Conference on Computer Graphics, Virtual Reality, Visualisation and Interaction in Africa*. ACM. 2010, pp. 121–128.

[Sla+07] K. C. Slatton, W. E. Carter, R. L. Shrestha and W. Dietrich. 'Airborne Laser Swath Mapping: Achieving the resolution and accuracy required for geosurficial research'. In: *Geophysical Research Letters* 34.23 (2007). L23S10. ISSN: 1944-8007. DOI: 10.1029/2007GL031939.

[TIS08] A. Tevs, I. Ihrke and H.-P. Seidel. 'Maximum mipmaps for fast, accurate, and scalable dynamic height field rendering'. In: *Proceedings of the 2008 symposium on Interactive 3D graphics and games*. ACM. 2008, pp. 183–190.

[Von+16] J. Von Schwerin, H. Richards-Rissetto, F. Remondino, M. G. Spera, M. Auer, N. Billen, L. Loos, L. Stelson and M. Reindel. 'Airborne LiDAR acquisition, post-processing and accuracy-checking for a 3D WebGIS of Copan, Honduras'. In: *Journal of Archaeological Science: Reports* 5 (2016). Figure 22, pp. 85–104.

[Wil13] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013. ISBN: 9780133261509.