GIGAVOXEL PAGED TERRAIN GENERATION AND RENDERING

BY
Alexander N. Pecoraro

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
awarded by DigiPen Institute of Technology
Redmond, Washington
United States of America

December
2015

Thesis Advisor: Pushpak Karnick

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDIES PROGRAM

DEFENSE OF THESIS

THE UNDERSIGNED VERIFY THAT THE FINAL ORAL DEFENSE OF THE MASTER OF SCIENCE THESIS TITLED

GigaVoxel Paged Terrain Generation and Rendering

BY

Alexander N. Pecoraro

HAS BEEN SUCCESSFULLY COMPLETED ON February 10, 2016.

MAJOR FIELD OF STUDY: COMPUTER SCIENCE.

APPROVED:

name date	name date
Graduate Program Director	Dean of Faculty
name date	name date
Department Chair, Computer Science	President

DIGIPEN INSTITUTE OF TECHNOLOGY

GRADUATE STUDIES PROGRAM

THESIS APPROVAL

DATE: February 10, 2016

BASED ON THE CANDIDATE'S SUCCESSFUL ORAL DEFENSE, IT IS RECOMMENDED THAT THE THESIS PREPARED BY

Alexander N. Pecoraro

ENTITLED

GigaVoxel Paged Terrain Generation and Rendering

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE IN COMPUTER SCIENCE AT DIGIPEN INSTITUTE OF TECHNOLOGY.

Pushpak Karnick date	Xin Li date
Thesis Committee Chair	Thesis Committee Member
Gary Herron date	Leo Salemann date
Thesis Committee Member	Thesis Committee Member

ABSTRACT

The paper investigates the feasibility of using the volume rendering technique described in the paper, "GigaVoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering" by Cyril Crassin, as the basis for a large scale terrain visualization system. Today's 3D terrain visualization systems are predominantly textured polygon mesh based rendering systems. These systems require geometric surfaces and their textures to be modeled at ever increasing resolution in order to achieve photo-realism. However, as textured polygon mesh resolution increases so does the amount of aliasing in the rendered image. A point is reached where an increase in polygonal and/or texture fidelity results in lower quality output. To overcome these shortcomings a new approach is needed. The introduction of programmable graphics processing units and the rapid increase in their processing power makes it possible to explore the use of other rendering data formats that do not have the shortcomings of textured polygonal meshes. One such format, that has promise, is a volume based format called voxels (e.g. volumetic pixels). This is because voxel data and voxel rendering techniques are less prone to aliasing issues than polygonal mesh techniques. Voxel rendering, however, requires significantly more memory and is generally more difficult to render at high frame rates. The GigaVoxel voxel rendering technique solves both of these problems. However, the GigaVoxel technique, as described by Cyril Crassin, concentrates on rendering only a single highly detailed object. Adapting and extending the GigaVoxel technique to support the rendering of massive outdoor environments that consist of multiple highly detailed voxel objects, which is a problem domain to which the GigaVoxel technique has yet to be evaluated against, is the focus of this thesis. In addition, in order to demonstrate the GigaVoxel Paged Terrain rendering system, this paper describes a unique out-of-core GigaVoxel Terrain generation system, which at this time is the only known example of an out-of-core GPU voxelization and sparse-voxel oct-tree generation system, implemented with NVIDIA's CUDA GPU programming platform, capable of converting large scale polygonal terrain data into a pageable runtime format optimized for rendering with the GigaVoxel rendering technique

TABLE OF CONTENTS

Page

LIST OF TABLES

Table Page

LIST OF FIGURES

CHAPTER 1

Introduction

Volume rendering is a set of techniques by which a 3D discretely sampled data set is displayed as a 2D projection. Volume rendering techniques generally fall into two categories, indirect and direct rendering[70]. Indirect volume rendering techniques rely on the ability to extract surfaces, such as polygons, from the volume and rendering them directly using typical 3D surface rasterization techniques. The Marching Cubes algorithm is a well known indirect volume rendering algorithm [36]. Direct volume rendering generates an image without converting the data to geometric primitives. This makes it possible for the entire volume data set to contribute to the image. Volume slicing, splatting, shear warp, and volume ray casting fall into this category of volume rendering [22, 42, 64, 66].

Volume ray casting, in particular, is interesting because, of all the techniques, it generates the highest quality images [13], but it is also the slowest. However, rapid increase in processing power of programmable GPUs over the past decade plus years now makes it possible to achieve very high quality volume rendering at interactive to real-time frame rates with volume ray cast rendering techniques [12, 13]. However, dealing with the increased memory demands of volume rendering is still an unsolved issue that is an active area of research [22, 3, 12, 13, 41].

1. Why Volume Rendering?

As the demand for ever more realistic computer generated images in movies, video games, simulation and the like increases traditional textured polygonal mesh based rendering systems become limited because in order to increase scene fidelity, the polygons must become smaller, to better model the surface geometry and, furthermore, the surface detail textures must increase in resolution. However, increasing scene fidelity by decreasing polygon and texel size causes a higher prevalence of aliasing in the output images [15]. There are several methods for dealing with aliasing in textured mesh scenes. To deal with texture aliasing texture MIP-mapping is used. However, prefiltered MIP-maps do not deal with aliasing

on the geometry's silhouette [15]. Eliminating aliasing on the silhouette or edge of geometry requires either multi-sampling or mesh simplification or both. Multi-sampling scales poorly because effectively anti-aliasing smaller and smaller polygons requires increasing the number of samples, which increases the memory required and decreases the frame rate. Furthermore, multi-sampling, is a static solution that does not adapt to the view or scene being rendered, which means that time and memory is wasted over-sampling or quality is compromised due to under-sampling [15]. Mesh simplification has its own drawbacks, such as the loss of essential details when a mesh's polygons are simplified, especially when simplified in an automated fashion. Automated mesh simplification algorithms also do not deal well with non-connected or very thin (i.e. trees, fur on an animal, etc.) mesh geometry making them not useful for many types of geometry [45]. Non-automated (i.e. artist controlled) mesh

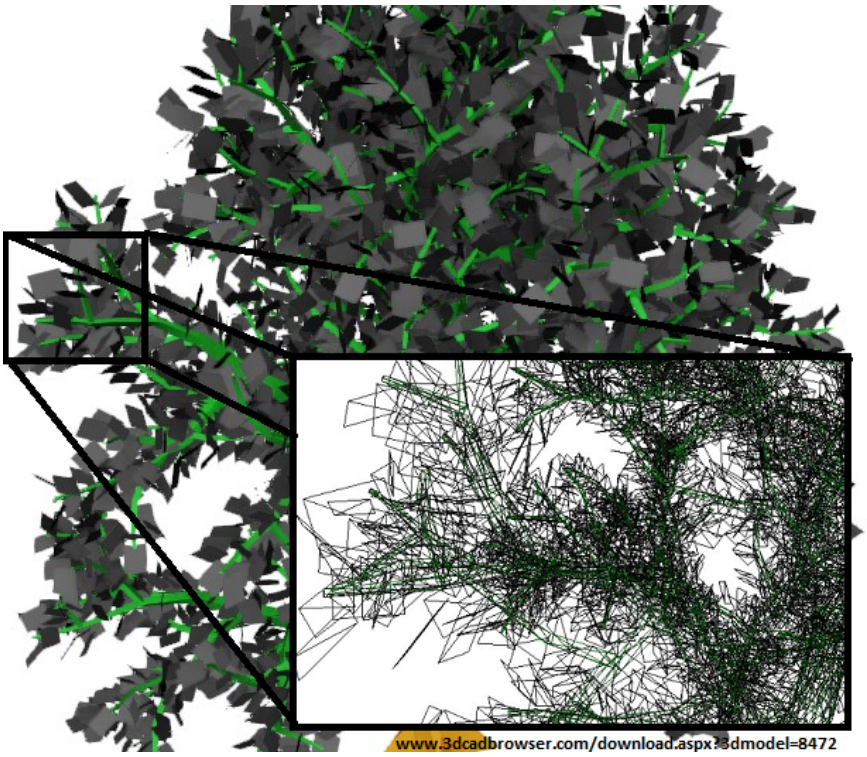

Figure 1.1. Example of high resolution tree whose non-connected leaf polygons are nearly impossible for automatic mesh simplification algorithms to handle.

simplification is time consuming, but is often the only solution for many polygonal mesh scenes if photo-realistic quality is desired [13]. Additionally, artist controlled mesh simplification, unlike automated-progressive mesh simplification algorithms, suffers from level of detail popping artifacts that show up when the mesh switches between the different levels of detail (i.e. non-linear detail reduction) [45]. For these

reasons, mesh simplification alone results in rendered images that are less than photo-realistic. To combat this, extra texture maps are introduced such as bump maps, normal maps, high frequency detail maps, etc. These measures help improve quality, but at the cost of even more memory consumption.

Voxel scenes, on the other hand, due to their uniform grid structure are well suited to automated simplification, even in cases where mesh simplification fails such as disconnected and thin geometry. Voxel scenes can be pre-filtered using the same MIP-mapping technique used to reduce aliasing in 2D texture images because voxels are essentially a 3D texture. Pre-generated, pre-filtered representations (i.e. a 3D MIP-map) of the voxel data can be generated and the best resolution for the current view dynamically picked at run-time in much the same way that it is done when picking a MIP-map level or levels during rasterization of a textured polygon mesh [12]. However, because the voxel structure represents the geometry's structure as well as the geometry's surface details, the 3D MIP-mapping method also effectively deals with anti-aliasing the geometry's silhouette reducing or eliminating the need for multi-sampling [12].

The challenge with voxel rendering is memory and bandwidth management. Memory usage is a problem for textured polygonal mesh based rendering too, but it is even more pronounced when dealing with voxel data because voxel data models the entirety of an object as opposed to just the surface of the object. Video memory is generally less plentiful than the system's main memory, as such, a brute force upload of high resolution voxel data from main memory to the video memory is not a scale-able solution. One method for optimizing video memory usage when working with voxel data is to encode the data into a Sparse Voxel Oct-Tree (SVO) [2, 27, 41]. An SVO optimizes memory by taking advantage of the fact that voxel scenes quite often consist of small dense clusters of non-homogeneous space surrounded by large expanses of homogeneous or empty space [41]. An SVO hierarchically partitions the space so that only the non-homogeneous partitions are stored in memory [41]. Furthermore, an SVO optimizes voxel rendering by enabling various other acceleration techniques such as empty space skipping and early ray termination [35].

Along with being relatively limited in size as compared to the main system memory, access to the GPU's video memory from the CPU is also relatively slow. The CPU cannot directly read or write from or to video memory. Changes to the video memory data by the GPU must be made in the main system memory and then transferred to the video memory. The transfer speed of the data is limited by the bus bandwidth available between the two memory systems and as such it is important to be very efficient when uploading or downloading data to the video memory. Cyril Crassin's GigaVoxel rendering system is an SVO based rendering implementation

designed to deal with the memory consumption and the memory bandwidth issues via an LOD management and video memory management system driven by GPU ray casting. This innovative technique makes it possible to efficiently render high resolution, out-of-core, voxel data sets in order to produce photo-realistic results at interactive to real-time frame rates on modern GPU hardware [12, 13].

Figure 1.2. Example of GigaVoxel scene.

The basic supposition that this thesis aims to test is that the GigaVoxel rendering technique's unique ability to support real-time rendering of out-of-core voxel data makes it a good fit as the basis of a large-scale voxel terrain rendering system. In addition, a GigaVoxel terrain would have the added benefit of being able to seamlessly combine the terrain details such as, trees, grass, street signs, buildings, etc. with the terrain itself, simplifying the LOD management of the scene and allowing the scene's terrain and terrain details LOD to linearly transition, in lock-step, again resulting in higher quality rendered images. Thus, the exploration of the GigaVoxel technique, as a basis for a volume terrain rendering system, is the focus of the research and software developed for and described by this thesis.

2. Objectives

The goal of this thesis is to build a large-scale terrain rendering system driven by the algorithm and data structures described in Cyril Crassin's paper, "Gigavoxels: Ray-guided Streaming for Efficient and Detailed Voxel Rendering.", and in doing so, answer the following questions:

1. Is the GigaVoxel technique a feasible basis for the implementation of a large-scale (i.e. out-of-core) terrain rendering system where the terrain consists of the terrain surface and the terrain details?

2. What extensions to the base GigaVoxel rendering algorithm are required to support large-scale terrain rendering?

3. How would a voxel terrain format that is optimized for use in a GigaVoxel based terrain rendering system be designed?

4. How would a system for generating such a terrain format be designed and implemented?

In answering the questions above, this thesis makes several contributions to the field of volume rendering.

3. Contributions

The contributions to the field of volume rendering provided by this thesis include::

1. Design of a GPU/CUDA software system that generates a page-able terrain format, optimized for rendering in a GigaVoxel terrain rendering system, from an out-of-core polygonal terrain data set.

2. Design of a GigaVoxel page-able terrain format.

3. Design of GigaVoxel paged terrain rendering system.

4. Performance results and analysis of prototype implementations of said software designs.

Implementation of a large-scale voxel terrain rendering system based on the GigaVoxel rendering algorithm requires the design of a GigaVoxel terrain format that supports paging, from disk, multiple oct-trees from a 3D grid of GigaVoxel oct-trees representing the terrain and terrain details as well as a system for generating the

oct-tree terrain grid. Furthermore, rendering multiple oct-trees, as opposed to a single oct-tree, requires extending the GigaVoxel rendering algorithm.

The design of the generation system and the extensions to the base GigaVoxel rendering system described in this paper are based on the latest research in the fields of GPU based voxelization and SVO generation and large scale terrain rendering. The GigaVoxel terrain generation system incorporates and extends prior work, to implement dual out-of-core (i.e. main memory and video memory), GPU based voxelization and SVO generation. The core GigaVoxel rendering algorithm is extended to also support dual out-of-core terrain rendering using techniques borrowed from the latest research in large-scale terrain rendering.

Furthermore, the performance results of a prototype implementation of the GigaVoxel paged terrain rendering system are presented and analyzed. From this analysis conclusions about the suitability of the GigaVoxel rendering algorithm as a basis for large-scale terrain rendering now and into the future are presented. Additionally, this thesis includes a discussion of ways that this work can be extended and improved.

4. Organization

This thesis consists of five chapters. Chapter 2 provides a survey of the research in the fields relevant to this thesis. Chapter 3 provides a detailed descriptions of the GigaVoxel terrain database format, GigaVoxel terrain generation system, and the GigaVoxel terrain rendering system respectively. Results of the prototype implementations are presented in Chapter 4. Analysis of and conclusions based on the results as well ideas for future work are presented in Chapter 5.

CHAPTER 2

Survey

This chapter provides a survey of the relevant research on volume rendering, volume rendering optimizations, terrain rendering, scene graphs, and sparse voxel oct-tree generation that was used as the basis for the software designs described in chapter 3.

1. Volume Rendering Techniques

The initial research into volume rendering was driven by the need for detailed renderings of volumetric data generated by medical devices such as CT and MRI scanners drove the initial research into volume rendering techniques [16, 60, 65, 70]. Additionally, geophysical exploration and geologic mapping processes produce large 3D data sets that require visualization [49, 61]. More recently, the movie and special effects industry, in particular, has found value in modeling and rendering highly detailed objects with volumetric data, especially objects whose surfaces are not well defined and are thus not easily modeled with textured polygons, such as fur, clouds, and highly detailed vegetation [17, 34].

Volume rendering techniques fall into two categories indirect volume rendering and direct volume rendering. An indirect volume rendering algorithms works by attempting to fit geometric primitives to the voxels or to generate geometric primitives from the voxels and rendering those using conventional rendering techniques. Direct volume rendering techniques do not attempt to convert the voxel data to geometric primitives and, thus, support semi-transparent voxels, which, unlike indirect methods, allow for every voxel to potentially be considered in the final rendered image. Direct volume rendering algorithms traverse the volume data directly and employ a transfer function to translate the voxel data values to a color and opacity value that are blended into the final rendered image, often taking into account a gradient and illumination [22].

1.1. Volumetric Data Models. Typically the data produced by CT, MRI, and other such 3D scanner devices is a stack of regularly spaced and regularly sized

images. Other types of volume generation processes, such as LIDAR, produce a 3D point cloud [69]. These types of data are mapped to a 3D array of regularly spaced voxels, generally referred to as a voxel data set, voxel grid, or simply voxels. Each cell in the voxel data set is a hexahedral (typically cubical) area surrounding a central gridpoint [21]. Historically, the voxel data model approach considers the data value stored at a single grid point in the grid to be homogeneous within the bounds of the grid point's voxel [21]. Another approach to model 3D volumetric data sets is as a 3D array of cells. Cell arrays model the volume data as a collection of hexahedra whose corners are grid points and whose values, as opposed to voxels, vary (i.e. can be interpolated) between the grid points [21]. The choice of whether to model the source data with a voxel grid or a 3D cell array comes down to whether or not it is appropriate to be able to interpolate when sampling from the data. However, it is common practice in the literature to refer to regularly spaced volumetric data as simply voxels and to be explicit about the extent to which the data in question can vary between grid points. This thesis will use that convention as well.

1.2. Indirect Volume Rendering Techniques. The best known indirect volume rendering algorithm is an isosurface extraction algorithm known as the Marching Cubes algorithm [36]. The Marching Cubes algorithm is a variation of the Cuberilles or opaque cubes algorithm, which was the first widely used algorithm for visualizing volume data [30]. The basis of the Marching Cubes algorithm is that the eight neighbor voxels in a voxel grid can be converted to a planar surface by comparing the values of those voxels to an isovalue. If one or more of the voxels in the group have a value that is greater than the isovalue and one or more of the voxels in the group have a value that is less than the isovalue then a portion of the isosurface is constructed by selecting from a group of 15 predefined polygons based on which of the voxels are greater than and which are less than the isovalue. Other examples of indirect volume rendering algorithms are the Dividing Cubes and the Marching Tetrahedra algorithms [11]. Both of these algorithms are enhancements to the Marching Cubes algorithm. The Dividing Cubes algorithm eliminates potential aliasing issues caused by sub-pixel sized polygons by drawing them as points. The Marching Tetrahedra algorithm prevents the erroneous holes sometimes seen with the Marching Cubes algorithm due to ambiguities in the pre-defined polygon selection process [11].

1.3. Direct Volume Rendering Techniques. Direct volume rendering techniques are further categorized into object-order and image-order algorithms. Object-order algorithms treat each voxel as a separate geometric point primitive that is projected and rasterized into the final image in much the same way that a triangle

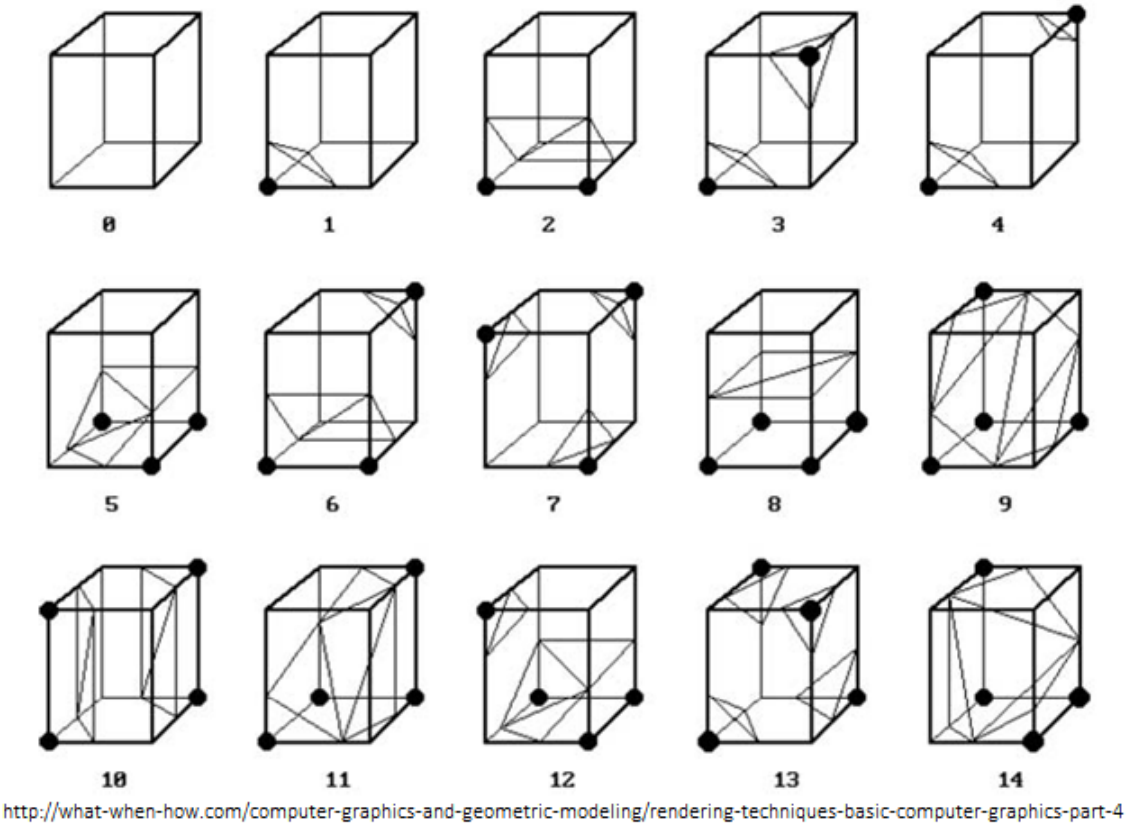

Figure 2.1. Marching Cubes Predefined Polygon Cases

is projected and rasterized in traditional 3D graphics applications [22]. Object-order techniques such as the Westover Back to Front Visibility Ordering and the V-Buffer Traversal Ordering are worth noting for historical perspective as they were some of earliest volume rendering algorithms [8]. The most advanced object-order volume rendering algorithm is the volume splatting algorithm [8]. The splatting algorithm works by more or less throwing each traversed voxel into the output image. The color and opacity are accumulated in the output image as the voxels are traversed in front to back order. The splatting algorithm starts by traversing from the closest voxel on to the farthest voxel and splatting each into the output image. The splatting algorithm is summarized below:

1. Project the voxel's object-space centroid into image-space.

2. Project a reconstruction kernel, such as a round Gaussian, into image-space and place it at the center of the voxel's image-space projection.

3. Using a transfer function, compute the resultant color and opacity of each pixel within the kernel by summing all the voxel contributions for that pixel.

When the splatting algorithm was originally introduced it provided better performance than any other direct volume rendering algorithm, but at the cost of quality in the final image output [8].

Image-order rendering algorithms, on the other hand, are fundamentally different than object-order rendering algorithms in that they start from a pixel in the output image and traverse the voxel data set to determine which voxels affect that pixel, as opposed to object-order techniques, which start at a voxel and attempt to determine which pixels the voxel affects. In general, image order voxel rendering algorithms cast a ray, starting at the pixel, in the direction that the camera is pointing and march the ray through the voxel data, sampling at regular intervals. Each data sample is converted to a color and opacity either at run-time or in pre-processing step via a transfer function [37]. The color and opacity is then used in a light interaction computation to compute the voxel's contribution to the final color of the pixel. As the ray progresses, the color contributed by each sample is accumulated until the ray exits the voxel grid [22]. The general flow of the algorithm is described below:

1. Calculate starting position and direction of each ray.
2. Step ray through voxel grid at regular intervals, sampling the voxel grid at each step.
3. Convert voxel data value to color and opacity with a transfer function.
4. Compute gradient.
5. Compute shading based on gradient and local light sources.
6. Composite shaded color and opacity into output image.

The ray casting technique produces the highest quality output images because it models the effect of illumination on the volumetric data more accurately than object order techniques, such as splatting [22]. This is because the regularly spaced stepping and sampling of the voxel data performs a Reimann sum to compute the volume rendering integral [22]. The volume rendering integral is the basis for the lighting model, known as the emission, absorption model of light transport through a participating medium, which is commonly accepted as the most accurate method to render volumetric data [22].

The paper, "Display of Surfaces from Volume Data." by Levoy [37] was one of the earliest described ray-casting algorithms capable of rendering locally illuminated,

high quality images of volumetric data that exhibited smooth silhouettes and very little aliasing artifacts. The key component of Levoy's paper was the introduction of a pre-processing step where the voxel data values are converted to a color and opacity via a transfer function prior to run-time. By pre-converting the voxel data values to colors and opacity it enabled higher order interpolation of the samples taken from the voxel grid by a ray during its traversal and thus smoother looking output [37].

Image-order rendering techniques produce more accurate results, but are generally slower than splatting and other object-order algorithms [13, 70]. However, the performance gap between object order and image order techniques is shrinking due to the introduction of and continued improvement of hardware accelerated rendering via dedicated graphics processing units. The early graphics processing hardware, unlike today's programmable GPU hardware, was designed for the purposes of rendering textured polygonal data only. As such, volume rendering researchers wanting to utilize the power of this hardware, had to fit their voxel algorithms into a textured polygonal mesh paradigm. The volume slicing algorithm is one such approach that simulates volume ray casting by using textured polygonal primitives in order to harness the power of the graphics hardware [64]. This algorithm works by

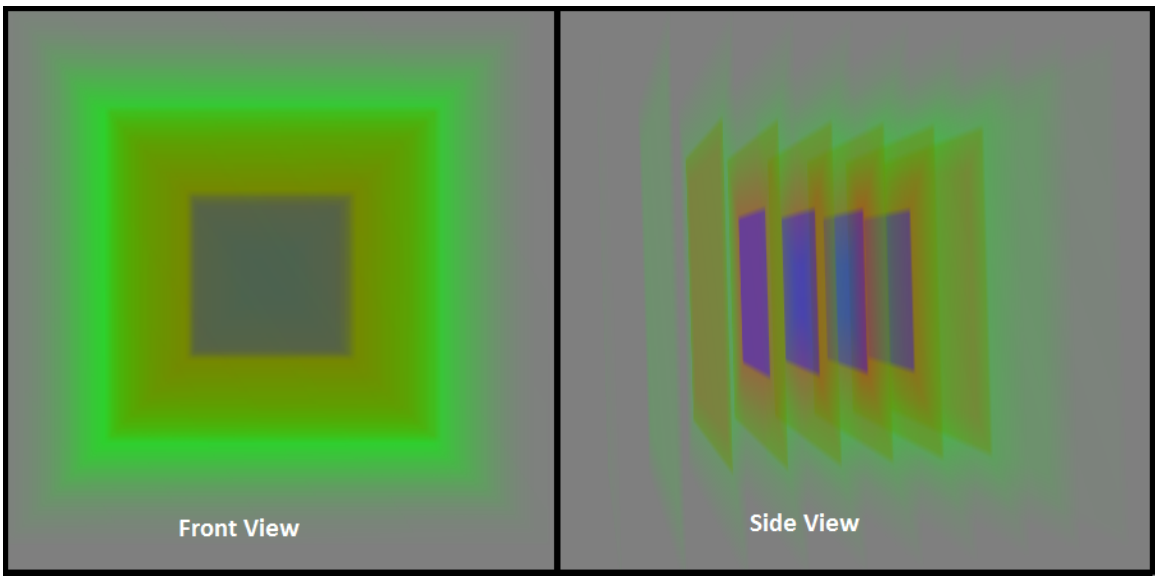

Figure 2.2. Volume Slicing

generating a stack of parallel planes aligned with the viewport that cut through the volume at regular intervals. The voxel grid is stored in a 3D texture map and each plane in the stack utilizes texture mapping coordinates to sample from the 3D texture. Each plane represents a step taken by a ray marching through the voxels. The left side of figure 2.2 shows the volume slicing technique applied to a voxel grid that models a

simple box structure. The left side of figure 2.2 shows a side view of the slice planes after disabling the dynamic update of the volume slice planes, which prevents the slice planes from rotating with the camera making the plane primitives that perform the volume texture sampling visible. The volume slicing approach is capable of producing very high quality images in real-time, but it is not a perfect replication of ray casting and thus does not produce as high quality of output because of the fact that the planes do not sample the voxel data at regularly spaced intervals (see figure 2.3).

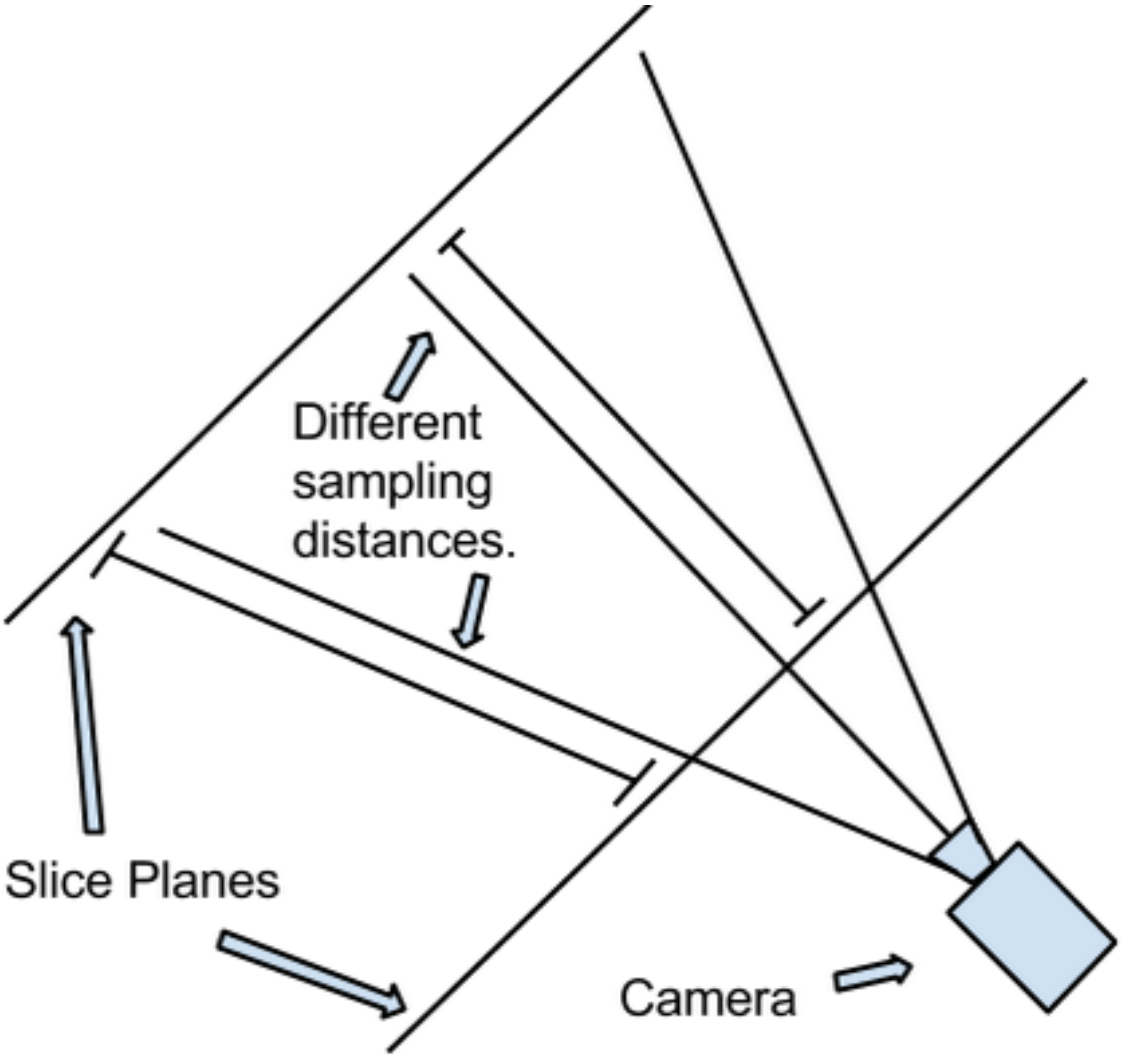

Figure 2.3. Volume slicing samples volume at irregularly spaced step sizes

An improvement on the volume slicing technique is GPU ray casting. GPU ray casting algorithms are only feasible to implement on fully programmable graphics processing units. Additionally, because of the rapid increases in processing power of

GPU hardware it has become possible to render non-out-of-core volumetric data sets via GPU ray casting at interactive or even real-time frame rates. Some of the earliest implementations of GPU ray casting were demonstrated by Kruger and Westermen [35], Rottger et al. [56], and Scharsach [59]. For this technique, the voxel grid is again stored in a 3D texture in GPU memory. Proxy geometry, representing either the near plane or the eight planes surrounding the voxel grid's bounding box, is transformed along with the camera's position from world space into the voxel's 3D texture space using a vertex shader program. The pixel shader then computes each ray's direction vector by using the camera's texture space position and the texture space position of the pixel shader's current pixel. The pixel shader then marches the ray through the voxel grid sampling the 3D texture at regular intervals to obtain a color and opacity that are composited into the frame buffer. Interaction with local lighting sources can be computed with gradient vectors, computed on-the-fly via central differencing or in a pre-processing step [22] A ray's exit from the volume is detected when any of the components of its position are no longer in the range zero to one.

2. Volume Rendering Optimizations

GPU ray casting is very effective at producing high quality output and, on smaller data sets, can produce these images at a high frame rate. However, as the size of the data set grows, the performance tends to decrease and the amount of memory required to hold the larger data sets is often more than what is available on even high end GPUs. The next sections describe the latest research into volume rendering algorithms that attempt to overcome the performance and memory related challenges associated with rendering large volumetric data sets.

2.1. Performance Optimization. Optimizing the run-time performance of volume rendering algorithms, in particular volume ray cast rendering, has lead to the development of several optimization techniques including, early ray termination [35], empty space skipping, adaptive sampling, and occlusion culling [22, 43]. Early ray termination optimizes ray casting by ending a ray's traversal of the voxels before it has fully exited the volume when the pixel shader determines that a sufficient amount of opacity has been reached. This optimization can greatly improve rendering on some data sets. Occlusion detection, empty space skipping, and adaptive sampling is made possible by first sub-dividing the voxel data into non-homogenous, homogenous, and nearly homogenous regions. Sub-dividing and classification of the sub-regions of the voxel data is usually accomplished with hierarchical space dividing data structures such as BSP trees or oct-trees, which are constructed pre-runtime [22]. Occlusion detection queries executed on the GPU can be used to eliminate the need to ray cast an entire sub-region of the volume if the sub-region is determined to be completely

or substantially occluded. Occlusion queries work by drawing a proxy geometry, consisting of a few planes or triangles, representing the voxel's bounding box or the internal structure of the volume (possibly pre-generated by marching cubes) to the frame buffer and then using a graphics driver API (i.e. OpenGL, DirectX, etc.) to perform occlusion queries. The result of the queries provide a way to determine if the level of occlusion is large enough to warrant skipping the more expensive ray cast rendering of the voxel sub-region in question. This can be an effective optimization if the voxel data set has lots of self occlusions or when mixing volume rendering with polygonal rendering for applications such as games or simulations [22]. Empty space skipping works by allowing completely homogenous regions to be skipped entirely if they are completely translucent, or their contribution can be computed procedurally [22]. Adaptive sampling is an optimization that enables regions that are nearly homogenous to be down sampled to a lower resolution and traversed in fewer steps without affecting the accuracy of the results provided that for each step the color and opacity is corrected for the larger step size [22]. These optimization techniques when applied to rendering on modern GPU hardware further help to make interactive and real-time rendering of volume data via GPU ray casting possible. However, they do not necessarily address the challenge of dealing with out-of-core volume data, i.e. data too large to fit entirely into video memory.

2.2. Memory Optimization. The biggest obstacle to GPU ray cast rendering of large voxel data sets is that the size of the data set often does not even fit into the GPU memory. This type of data is referred to as out-of-core. There are several techniques described in the literature for dealing with out-of-core volumetric data. Scharsach introduced the idea of a texture cache or pool to deal with out-of-core volume data[59]. The texture pool is filled with only the regions of the voxel grid that are needed in order to render the current view point. A secondary 3D texture is used to keep track of which portions of the overall voxel grid are stored in the texture pool and where. Additionally, the Sparse Voxel Oct-tree (SVO) data structure is a data structure adapted to deal with out-of-core volume data. The SVO concept grew out of oct-trees used for solid texturing, which is a 2D texture mapping technique designed to deal with distortions caused by surfaces that have no natural texture map parameterization [52, 53]. Perlin's and Peachey's oct-trees were eventually adapted to the GPU [2, 27, 40]. From there voxel researchers realized the potential of oct-tree textures as an optimization for voxel rendering [12, 41, 47]. The SVO reduces the memory required to store a voxel grid by taking advantage of the fact that voxel data often consists of dense patches of non-homogenous space surrounded by large swathes of homogeneous space. The SVO thus allows homogenous portions of the voxel grid to be stored as a single constant value. SVOs have been shown to have a compression

ratio of 600,000 to 1 in data sets consisting of large expanses of terrain [58]. the concept of the GPU based SVO was further enhanced by the integration of it with the brick map technique first developed by Christensen and Batali [6]. Combining the concept of a brick map with an SVO allows the oct-tree nodes to be stored separately from the voxel content, which improves the coherency of the memory accesses.

The GigaVoxel rendering algorithm combines the texture pool, SVO, and brick map concepts into an innovative algorithm that performs real-time ray cast rendering of voxel data. To optimize GPU memory usage the SVO structure and the bricks are stored in two texture pools [12]. The texture pools guarantee a consistent amount of texture memory usage. They are loaded with only the portions of the voxel data that are needed for the current viewpoint. As the viewpoint changes the stale, no longer needed, SVO nodes and bricks are replaced by new nodes and bricks in an LRU fashion. The needed nodes and bricks are identified by the GPU rays themselves without any CPU based traversal of the SVO required [12]. This feature is unique to the GigaVoxel technique. Other research has been conducted into efficient use of the SVO for real-time rendering by Gobetti et al. and Laine et al. [26, 41]. However, the GigaVoxel technique, despite being very similar to Gobetti's technique, is more efficient in terms of memory usage and also performs better overall because the GigaVoxel technique does not require a CPU based traversal of the SVO to determine which nodes and bricks to load like Gobetti's. In addition, the GigaVoxel technique is more flexible than Laine's technique because it focuses solely on dealing with opaque object surfaces and is not capable of dealing with objects that consist of both partially transparent and opaque voxels [12]. The GigaVoxel technique is described in more detail in the 3 chapter.

3. Terrain Rendering

Terrain rendering research covers a lot of ground in the 3D visualization world. The areas of particular interest to the goals of this thesis are level of detail research and out-of-core research. Research on level of detail (LOD) management and, in particular, LOD management strategies that support out-of-core paging of terrain data from disk to system memory and from system memory to video memory are especially relevant. Managing the paging and LOD of large outdoor terrain is very complex because terrain data consists of the terrain surface, the static terrain details or models (i.e. building models, vegetation models, etc.), the terrain surface texture maps, and the static terrain models' texture maps. All of these data types require a different paging and LOD management strategy. Researchers over the years have developed some very complex solutions to the problems associated with this domain. These solutions usually start with the basic premise of dividing the terrain up into

separately page-able tiles [45]. From that basic premise others have developed more complex schemes to optimize paging performance such as Lindstrom and Pascucci's use of terrain data paging via memory mapped files [44] or Gao's technique involving compression and complex viewer motion prediction with pre-fetching of tiles via a background thread [25].

In addition to terrain data paging, terrain rendering systems manage terrain surface LOD. The earliest algorithms for managing terrain surface LOD involved the triangulated irregular network (TIN) data structure and algorithm initially developed for terrain rendering by W. Randolph Franklin in 1973 [71]. A TIN of polygons is generated from a heightmap such that it models the surface as closely as possible, within some error threshold, using as few vertices as possible. Continuous level of detail algorithms attempt to do at runtime what TINs do pre-runtime [45]. The best known CLOD method are, most likely, the ROAM algorithm developed by Duchaineau[19], of which there are many variants [44, 25], as well as Hoppe's progressive mesh refinement technique [31]. CLOD methods are so named because they compute and generate the terrain surface mesh at runtime from a heightmap or digital elevation map (DEM) and continuously update the number of vertices uses to model the surface based on the current viewpoint.

Run-time paging and LOD management of a terrain's texture maps are another highly active area of research. A basic approach, conceptually the same as the TIN method, is to pre-generate multiple different resolutions of a texture and then use a view dependent calculation to pick a specific resolution for paging [45]. A more complex solution to this problem, known as the texture clipmap, dynamically streams in only the needed portions of a large out-of-core texture based on a view centered rectangular region. This method, originally developed by Tanner et al. [63], has been enhanced and extended by many others, including [32, 10]. The texture clipmap technique has also been extended to the domain of terrain surface management as well, in the form of method known as the geometry clipmap [39], which involves streaming the heightmap instead of a texture map.

As far as LOD management of the terrain detail models like vegetation, buildings, etc., the most common practice is to generate multiple representations of the models at various levels of detail, either automatically or by hand, and to use a distance or projected size metric to switch between the different resolutions at run-time [45, 73]. An individual model's texture map LOD tends to be tightly bound to the model's geometry LOD, which makes them easier to manage than the terrain surface geometry and textures LODs, which are often not directly tied to each other.

More recently, the idea of managing the LOD of the shader program used to render a terrain surface or a terrain detail model has been studied. For instance,

the shader program used to render a highly detailed tree model, one that supports complex light interactions with leaves, branches, etc. is not necessary and obviously too costly to execute when the tree is viewed at a great distance and in great numbers. Meyer et al. has proposed use of multiple different shaders where each shader is designed to render at different resolution from needles on the branches on up to the collection of branches and trees that form the canopy of a forest[46]. Clearly, given all the research it has generated, large scale terrain paging, LOD management, and rendering is a challenging problem.

4. Scene Graph Based Rendering

Closely related to the field of terrain rendering research is the field of scene graph research. A scene graph is data structure commonly used in 2D and 3D graphics to organize and optimize the rendering of a scene and the objects in it [68]. One of the earliest descriptions of a scene graph comes from Clark in which a scene graph is presented as mechanism for LOD management, polygon clipping, data paging based on visibility information, and the ability to tightly couple rendering performance with scene complexity [9]. In addition to those initially identified benefits, modern scene graphs designed for today's graphics rendering technology can also benefit from the ability of a scene graph to spatially and hierarchically organize a 3D scene. This organization makes possible the implementation of common optimization techniques such as culling, and render state inheritance, encapsulation, and sorting [1]. By spatially organizing the elements of a 3D scene in a scene graph, a culling systems can easily identify and collect the currently active set of visible objects in order to deliver them to the rendering system for efficient drawing on screen. Furthermore, a culling system can identify portions of the scene graph that should be loaded from disk because it can predict that a portion of the scene graph will soon be coming into view [1]. The primary mechanisms used to determine the potential visibility of a scene graph branch or leaf node is to detect an overlap between the node's bounding sphere and the view frustum or to detect that the bounding sphere is within some specific distance of the camera. Alternatively, the projected size of a scene graph node's bounding sphere (in pixels) is computed and when the projection becomes larger than some pre-configured amount, the node is collected for rendering or triggered for paging [1].

Scene graphs also optimize rendering performance of large 3D scenes by enabling graphics system state inheritance and encapsulation. Inheritance and encapsulation of graphics system state means that the changes to the graphics system state, needed to render a particular branch of the scene graph, can be stored in a state structure at various levels of the graph and inherited by and

overridden by lower levels. This architecture makes it possible to sort the list of scene graph nodes collected for rendering such that graphics state switching is reduced to only the essential changes and redundant changes are eliminated, which greatly optimizes rendering performance on modern hardware [1]. An example of a scene graph that supports culling, paging, state encapsulation, and sorting, among other things, is the OpenSceneGraph, which is a widely used opensource scene graph based rendering toolkit [5]. The OpenSceneGraph provides inspiration for the scene graph implemented by the rendering system described in this thesis, in particular the OpenSceneGraph's Group node, the PagedLOD node, and the Drawable node types. The Group node provides a way to spatially organize the scene graph by storing a bounding sphere of the child nodes of the Group. The PagedLOD node is special type of Group whose children are not loaded into the scene graph until the culling system determines that they are potentially in view [5]. The culling system initiates the loading of the PagedLOD node's child nodes by submitting a request to a database pager thread [5]. A Drawable node is a leaf node that encapsulates a draw-able entity [5]. The OpenSceneGraph culling system collects these items in its list of potentially visible draw-able items and then sorts them for rendering based on their encapsulated graphics state [5]. Furthermore, each PagedLOD node's last accessed time-stamp is tracked so that stale data can be unloaded after a configurable amount of time [5]. These methods are put to use in the paging and culling system for the GigaVoxel terrain rendering system described by this thesis.

5. Sparse Voxel Oct-Tree Generation

In order to test the feasibility of using the GigaVoxel and related large scale terrain rendering research on a large scale voxel terrain, a terrain generation system capable of producing a GigaVoxel SVO based terrain is needed. Typically, a real world terrain data set designed for 3D visualization is generated by processing the Earth measurements obtained from multiple different types of scanners (Satellite imagery, LIDAR, etc.) into a set of catographic data sources (imagery, height map, etc.). The cartographic sources are then compiled, by a terrain generation system, into a 3D visualization optimized run-time format consisting of the terrain skin, terrain texture, and terrain details (building models, tree models, etc.) [60, 58]. In general, most 3D terrain visualization systems are designed to run on data that is in a textured polygon format and most of the existing software tools that generate 3D terrain visualization data target textured polygons as the output format. Ideally, a voxel based terrain generation system would go directly from Earth measurements to voxel data, but developing the tools to do this is beyond the scope of this thesis. As such, the input to my GigaVoxel terrain generation system is a textured polygon mesh terrain

generated by a typical 3D terrain generation system.

5.1. Voxelization. The first step in generating GigaVoxel terrain is to voxelize the polygonal terrain. Fortunately there is a fair amount of research on voxelization of polygonal data. Hardware accelerated voxelization executed on the GPU is of particular interest because voxel data, being essentially a 3D array, maps nicely to the parallel architecture of a GPU making it an optimal computing platform for generation and computation of voxel data. Pantaleon's VoxelPipe is one of the best examples of a GPU based voxelization system [50]. VoxelPipe is based on previous work by Schwarz and Seidel [62], Zhang et al. [74], Eisemann and Decoret [20], Dong et al. [18], and Fang and Chen [23], among others. The key building block of the VoxelPipe software is its triangle/box overlap test that determines if a polygon should contribute to a particular voxel. This test, which stems from the 2D version of the test first described by Moller and Aila [47] and later extended to 3D by Schwarz and Seidel [62], works by first executing an inexpensive test to determine whether the triangle plane intersects the box. Then a more expensive test is performed. The expensive test comes in two variants, a thin voxelization variant, and a conservative voxelization variant. The thin variant checks for intersection of the 2D projections of the triangle and the voxel's bounding box along the dominant axis of the triangle's normal. The conservative variant checks for intersection along all three axis [50]. The thin variant is faster, but can incorrectly produce holes in certain situations. The conservative test is more accurate, but takes more time and can produce a voxelization that appears to be more chunky. Prior to executing the fine grained overlap test the VoxelPipe

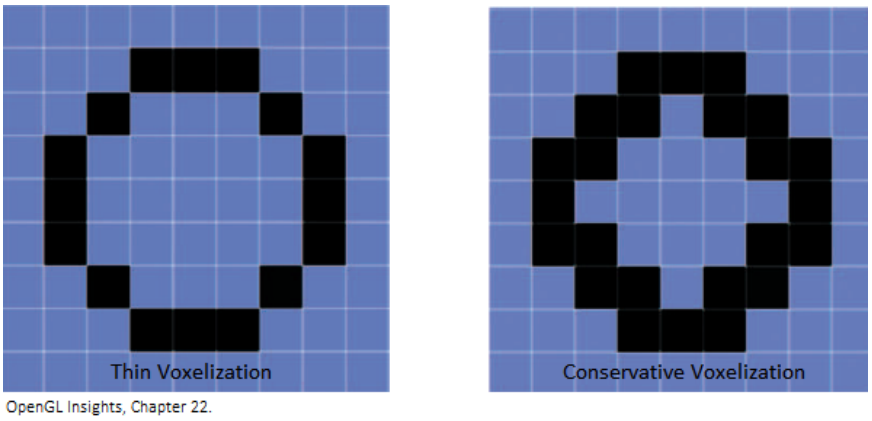

Figure 2.4. Thin and conservative voxelization example

algorithm performs several pre-processing steps, including:

1. Coarse Grained Triangle Prep - these steps assign a single GPU thread to each

triangle in order to obtain a coarse grained sort of the triangles assigned to tiles (tiles are 3D sub-region within the full voxel grid).

(a) Pre-compute integer bounding box and dominant axis (these values are used in box/triangle overlap test) generating a list of triangles and their associated data.

(b) Perform coarse sorting to assign trangles to voxel tiles. This process generates an unsorted list consisting of the triangle-id and tile-id.

(c) Sort the triangle-id, tile-id list by tile-id.

(d) Split the per tile lists up into smaller chunks (this is done for better load balancing).

2. Fine Grained Process - these steps perform the voxelization with a single GPU thread per triangle where triangles are grouped by tile.

(a) Execute triangle plane overlap test.

(b) Execute triangle axis test.

(c) Write data to tile/voxel output.

5.2. SVO Generation. After voxelization of polygonal data is complete an SVO can be generated from the voxel grid. Parallel to my own implementation of GPU based SVO generation Crassin et al. and Baert published papers describing similar techniques for SVO generation. Crassin et al. describes an algorithm for GPU based SVO generation in an article written for the book OpenGL Insights [14]. Crassin's algorithm makes use of the latest features of OpenGL, including image load/store and atomic counters, to generate an SVO from a high resolution mesh in real-time. This technique is somewhat similar to the approach of my implementation, but less memory intensive and faster. Their algorithm is described here:

1. Compute voxelization of highest resolution nodes (i.e. leaf nodes).

2. Identify non-homogenous nodes using voxelization data from step 1.

3. Compute MIP-maps for lower resolution levels using node data from step 2 and voxelization data from step 1.

The first step voxelizes each triangle of the input mesh. Crassin does not use the triangle/box overlap test and instead uses the GPUs rasterization capabilities to orthographically project each triangle along the dominant axis of its normal in order to maximize it's projected area into the output image. The output image is configured

to be the size of the lateral size of the desired voxel grid's resolution (i.e. 512^2 if the desired voxel resolution is 512^3). The projected triangle produces a set of 2D fragments that indicate the potential for an intersection between the triangle and a particular voxel. The pixel shader then uses the interpolated depth of each 2D fragment and the screen space derivatives to compute the list of voxels in the 2D fragment region that are actually intersected. This list of voxels is then added to a pre-allocated buffer using the image load/store and atomic operations introduced by OpenGL 4.2 capabilities. The list of voxel intersections is then used to generate an oct-tree by traversing from the root of the tree down to the leaf node that contains each of the intersected voxels. Finally, the brick for each node is created by traversing back up the tree and filling in each brick by computing a MIP-map from the higher resolution node's brick. The only shortcoming with this algorithm is the requirement for a pre-allocated buffer to store the initial list of voxel intersections. If a particular set of triangles generates more voxel intersections than can be held by the list then they will be lost and the voxelization will be incomplete [14].

The algorithm described by Baert et al. in a paper entitled, Out-of-Core Construction of Sparse Voxel Octrees, makes the insight that a full voxel grid is not needed to compute a branch of the SVO that would contain and model it [3]. As such the software is designed to process the source data in chunks corresponding to the regions that represent the bounding boxes of the leaf nodes in the SVO. The algorithm is in fact very similar to the algorithm that this paper describes for GPU SVO generation, but has a few key differences, foremost of which is that Baert's algorithm is not a GPU algorithm [3].

CHAPTER 3

Implementation

This chapter describes the design of the GigaVoxel terrain renderer and a system capable of generating a pageable grid of GigVoxel sparse voxel oct-trees from a large polygonal texture terrain. The rendering system is implemented with OpenGL and is based on the GigaVoxel rendering algorithm with a few extensions in order to support paged SVO terrain. The terrain generator is implemented using NVIDIA's CUDA (Compute Unified Device Architecture) and is based on the techniques for voxelization and SVO generation described in chapter 2.

1. CUDA

CUDA is NVIDIA's parallel computing platform and programming model that streamlines the implementation of GPU powered applications on NVIDIA GPUs [67]. CUDA was used to implement the terrain generation system because voxelization and sparse voxel oct-tree generation from polygonal data, due to the 3D grid nature of the voxel and SVO data, is well suited to take advantage of the massively parallel architecture of the GPU and CUDA makes harnessing that power for non-graphics applications, like a terrain generation system, much easier to accomplish. The CUDA programming model is designed around the execution of GPU kernel programs. Multiple instances of a kernel program are executed on the GPU in parallel. Each kernel program can uniquely identify itself by its thread indexes and its block indexes [48]. A kernel instance's block and thread indexes are essentially global constants that a running kernel can obtain via the blockIdx and threadIdx built-in symbol names. They are both defined to be of type uint3 aka X, Y, and Z [48]. The dimension or number of blocks is referred to as the grid size and the dimension or number of threads is referred to as the block size. A GPU, depending on it's capabilities, can support up to a certain number of thread blocks and a certain number of threads per block for a single kernel execution. The grid size times the block size equals the total number of threads that can execute a particular kernel concurrently. When executing a kernel the number of blocks and threads that are spawned is specified as part of the

execution syntax [48]. For example, to execute a kernel with 2 blocks of 2 threads one would do:

- MyKernelFunction<uint3(2, 0, 0), uint3(2, 0, 0)>(...)

The example above would result in the execution of MyKernelFunction on 4 concurrent GPU threads.

2. GigaVoxel Sparse Voxel Oct-Tree

2.1. GigaVoxel Run-time Format. In order to better understand the algorithms used to generate a GigaVoxel oct-tree, a detailed description of the GigaVoxel oct-tree run-time data structure is required. First off, the tree structure as described in [12], is not necessarily an oct-tree. The tree structure is actually

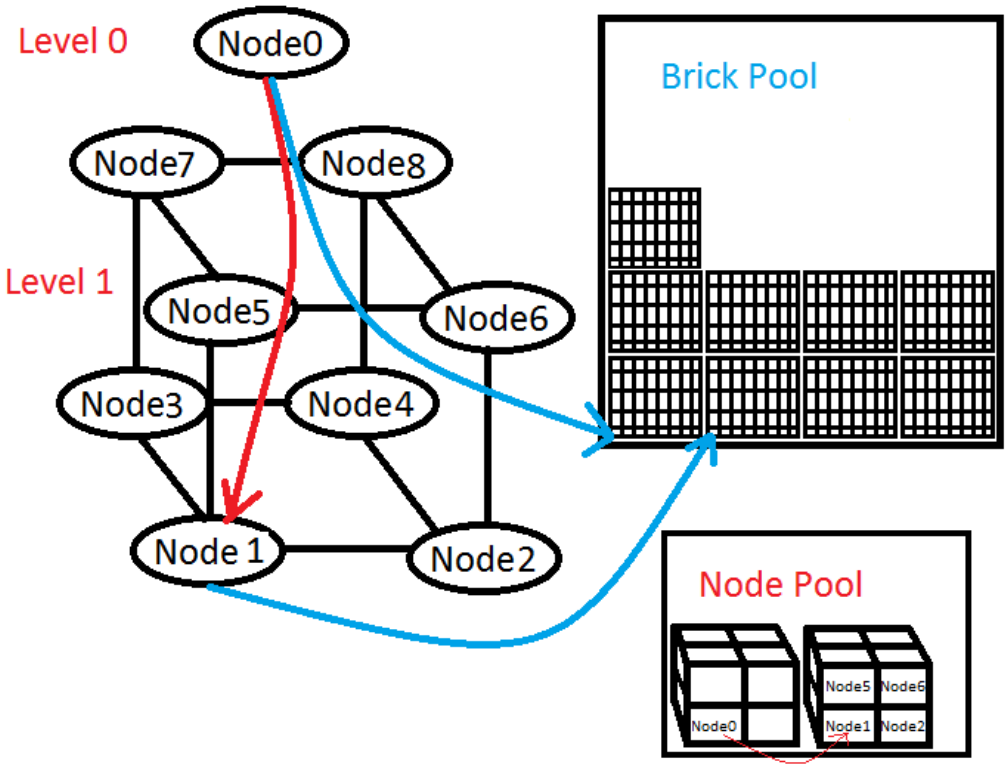

Figure 3.1. GigaVoxel Oct-Tree and Texture Pool Structure

generalizable into an N^3 tree (i.e. each node has N^3 uniformly sized children). The choice of N offers a trade off between memory efficiency and traversal efficiency. A smaller N results in a deeper tree, whereas a larger N results in a shallower tree. However, in order to simplify the implementation of both the generation system and

the rendering system support for N=2 trees is implemented (i.e. an oct-tree). The GigaVoxel oct-tree data structure is stored by the rendering system in both main system memory and in video/GPU memory. The GPU version of the oct-tree is stored in two pool textures. The Node Pool texture stores the nodes of the tree and the Brick Pool texture stores the brick for each node in the Node Pool (refer to figure 3.1). The Node Pool texture is a 32 bit unsigned integer luminance alpha texture.

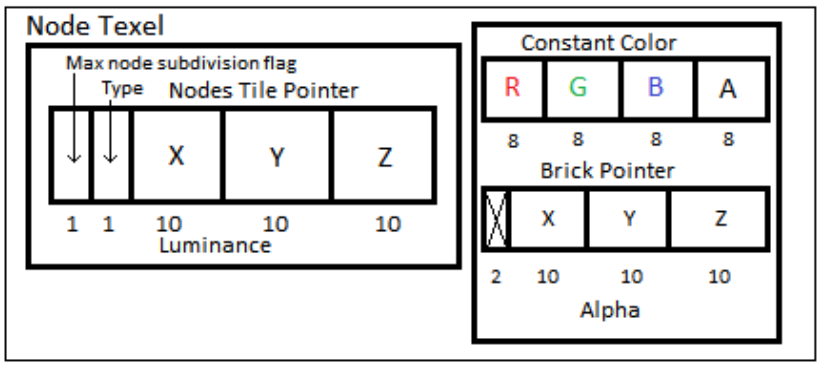

Figure 3.2. Node Pool Texel

Each node in the tree is stored in a single texel of the Node Pool texture. The first 32 bits of the texel are used to store a max subdivision flag, a node type flag, and a child pointer. The max subdivision flag, which is stored in the highest order bit is used to indicate whether the node is refined to a maximum or whether the node has children that are yet to be loaded into the Node Pool. The node type flag stores whether the node is a constant or non-constant node. The first 30 bits, with 10 bits per each XYZ component, are used to encode a pointer to the node's children (i.e. an XYZ offset within the Node Pool texture). Each node only needs a single child pointer because its children are stored in a 2x2x2 contiguous block of the Node Pool, thus the siblings of the base child can be addressed via a simple XYZ offset from the base child. The second 32 bit portion of the node (i.e. the alpha component of the luminance alpha texture) stores either a constant color or a brick pointer. The constant color is encoded as an 8 bit RGBA value. The brick pointer, like the child pointer, is encoded as three 10 bit integers that specify the XYZ offset of the brick in the Brick Pool texture (refer to figures 3.1 and 3.2) .

A node's brick stores the portion of the voxel grid that corresponds to the area within the node's bounding box. The leaf node bricks contain the color and opacity from the highest resolution voxel grid (post transfer function translation from scalar to color and opacity). The data stored in the non-leaf node bricks comes from a voxel grid generated by, MIP-map style, down sampling of the highest resolution voxel

grid. For example, in a two level oct-tree that represents a 16x16x16 voxel grid, the root node brick is an 8x8x8 voxel grid generated by computing the MIP-map of the 16x16x16 voxel grid. In turn, the 16x16x16 voxel grid would be divided amongst the 8 children of the root node, giving each of them an 8x8x8 brick. Also, note that for proper interpolation when sampling, each brick needs to store an extra layer of border voxels to ensure that linearly interpolated samples taken around voxels at the edge of a node's extent pull from the voxels that are actually neighboring the node's bounding region, as opposed to whatever voxels happen to be stored in the Brick Pool next to the node's brick.

2.2. Paged GigaVoxel Scene Graph File Format. Ultimately, the generation system needs to generate a 3D grid of GigaVoxel oct-trees, where each oct-tree represents a chunk of the terrain and terrain details. The oct-trees need to be in a format that can be quickly loaded from disk when requested by the rendering system. The generation system encodes the 3D grid of page-able oct-trees in the Paged GigaVoxel Scene Graph XML file. The Paged GigaVoxel Scene Graph XML file consists of an "OctTrees" node, which is similar in function to the OpenSceneGraph's "Group" node, and a "PagedOctTree" node, which is similar in function to the OpenSceneGraph's "PagedLOD" node [5]. A detailed description of each node type and its parameters is described below.

1. OctTreeGroup - element node that contains OctTree element nodes.
   (a) Parameters:
      i. Compressed - indicates whether or not the OctTrees referenced by this XML file contain compressed brick data. The possible values are YES or NO.
      ii. BrickXSize, BrickYSize, BrickZSize - indicate the size in voxels of the bricks referenced by the OctTrees in the grid of oct-trees.
   (b) Child Elements: OctTree
2. PagedOctTree - element node that provides a reference to GigaVoxel oct-tree tree and brick data.
   (a) Parameters:
      i. CenterX, CenterY, CenterZ - specifies the center position of the OctTree.
      ii. Radius - specifies the radius of the bounding sphere of this OctTree.

iii. TreeFile - specifies the path to the XML file that defines the nodes of the actual OctTree.

(b) Child Element Nodes: None

The OctTreeGroup element's Compressed and BrickXSize, BrickYSize, and BrickZSize indicate to the rendering system the dimensions and data type required for the Brick Pool texture. The PagedOctTree element's parameters specify a bounding sphere for the oct-tree referenced by the TreeFile parameter, both of which are used by the rendering system for on-demand loading of oct-tree data. The TreeFile parameter specifies the path to the GigaVoxel Oct-Tree File.

2.3. GigaVoxel Oct-Tree File Format. The GigaVoxel Oct-Tree File specifies the structure of the oct-tree nodes in a single GigaVoxel oct-tree. The generation and rendering systems support either an XML format or a more compact binary format. The structure of the XML version of the GigaVoxel file is described below.

1. GigaVoxelsOctTree - contains Node element nodes and specifies oct-tree parameters.

   (a) Parameters:

      i. X, Y, Z - specify the X, Y, and Z location in world coordinates of the minimum corner of the oct-tree's bounding box.

      ii. DeltaX, DeltaY, DeltaZ - specify the size in meters of a single voxel in the oct-tree.

      iii. VolumeXSize, VolumeYSize, VolumeZSize - specify the size in meters of the bounding box of the oct-tree.

      iv. BrickXSize, BrickYSize, BrickZSize - specify the number of voxels in the bricks referenced by the nodes of this oct-tree.

      v. Binary - specifies whether or not the brick data referenced by the nodes in this oct-tree is stored in a binary format or a text format. Possible values for this parameter are YES or NO.

   (b) Child Element Nodes: Node

2. Node - specifies the parameters for a node in the oct-tree.

   (a) Parameters:

i. Type - specifies whether the node is a CONST (i.e. contains a constant color), or a NON-CONST (i.e. contains a brick), or a LEAF-CONST node. A LEAF-CONST node is a node that is leaf node, but is not at the MaxDepth of the oct-tree because it is a CONST and all of its children are CONST nodes.

ii. Brick - specifies the index of this node's brick in the file that contains the brick data for the node's at the current level of the tree.

(b) Child Element Nodes: None

The GigaVoxelsOctTree element's X, Y, Z, DeltaX, DeltaY, DeltaZ, VolumeXSize, VolumeYSize, and VolumeZSize parameters specify the parameters for the bounding box "proxy geometry" that serves as a mechanism for ray generation in the rendering system. The BrickXSize, BrickYSize, BrickZSize, and Binary parameters specify parameters that describe the brick data referenced by the nodes in the oct-tree.

The binary format of the GigaVoxel Binary File is optimized to facilitate quick loading from disk into the node pool texture, as such, its format closely resembles the format of the Node Pool texture. The header of the file consists of the parameters defined by the GigaVoxelsOctTree XML element node, specifically:

1. Bytes (0 - 96]: three 32 bit floating point values encoding the X, Y, and Z parameters.

2. Bytes (96 - 192]: three 32 bit floating point values encoding the DeltaX, DeltaY, and DeltaZ parameters.

3. Bytes (192 - 288]: three 32 bit unsigned integers encoding the VolumeXSize, VolumeYSize, and VolumeZSize parameters.

4. Bytes (288 - 1056]: twenty four 32 bit floating point values encoding the vertexes of the bounding box of the oct-tree.

5. Bytes (1056 - 1057]: one bit to encode the Binary parameter.

6. Bytes (1057 - 1089]: one 32 bit unsigned integer encoding the size of the oct-tree node data that follows.

From there bytes 1089 through the size specified by the last header parameter contain the data for the pixels of the node pool texture starting with the root node. The root node inhabits the first 64 bits of the first 512 bits starting at offset 1089. Recall that we need 512 bits for each node because nodes are stored in 2x2x2 blocks in the Node Pool texture. The next 512 bits encode the children of the root node. Following that

is the 512 bits to encode the first child of the root node's children. The children are laid out in the file such that they can easily be loaded into the Node Pool texture via a call to glTexSubImage3D.

The data that is stored starting at offset 1090 is loaded by the rendering system as a single chunk of contiguous memory. After loading this "oct-tree chunk" into main memory, the rendering system, or more specifically the data paging system, builds an oct-tree of nodes where each node has a pointer to the offset within the chunk corresponding to the node (i.e. the root node points to offset zero, the root's first child points to offset 512). Refer to section 3 for more details on the run-time data structures used by the rendering system.

The brick data is stored separately in the Brick Data File and each level of the oct-tree has its own Brick Data File. The content of the Brick Data File are the RGBA colors of the voxels and XYZ components of the voxel gradients. The brick file contains the bricks in the same order that the nodes are ordered in the node file so that is easy to determine the offset within the Brick Data File that a particular node's brick, which is again used at load time by the paging system to build an oct-tree node structure in main system memory that indexes into the Brick Data File and the GigaVoxel Binary File. Each brick is stored such that it can easily be loaded into the Brick Pool texture with glTexSubImage3D. The format of the file's header is as follows:

1. Bytes (0 - 192]: encodes the dimensions of each brick (width, height, depth).

2. Bytes (192 - 193]: a bit indicating if the data in the brick is compressed or not.

After the header each node's brick data is encoded in the file with the RGBA portion leading and the XYZ gradient portion following. Each RGBA portion is preceded by a 32 bit integer specifying the size of the RGBA brick data to follow. The XYZ gradient portion is also preceded by a 32 bit integer specifying the size of the XYZ brick data to follow. The bricks are in the same order that the NON-CONST nodes appear in the node definition file.

2.4. GigaVoxel Paged Terrain Generation System Overview. The generation system takes as input both the terrain skin and terrain details, such as tree and building models, in the form of an OpenSceneGraph runtime terrain database and outputs a 3D grid of GigaVoxel oct-trees and a scene graph with references to the oct-trees that are used for paging/loading the oct-trees at run-time by the rendering system. The primary challenge of the system is to maintain a minimal memory footprint while dealing with out-of-core terrain data and to quickly convert large amounts of polygonal terrain and imagery into GigaVoxel oct-trees. Figure 3.3 is a

flow chart showing the major components of the system and the inputs and outputs of each. The major components and the inputs and outputs are described in more

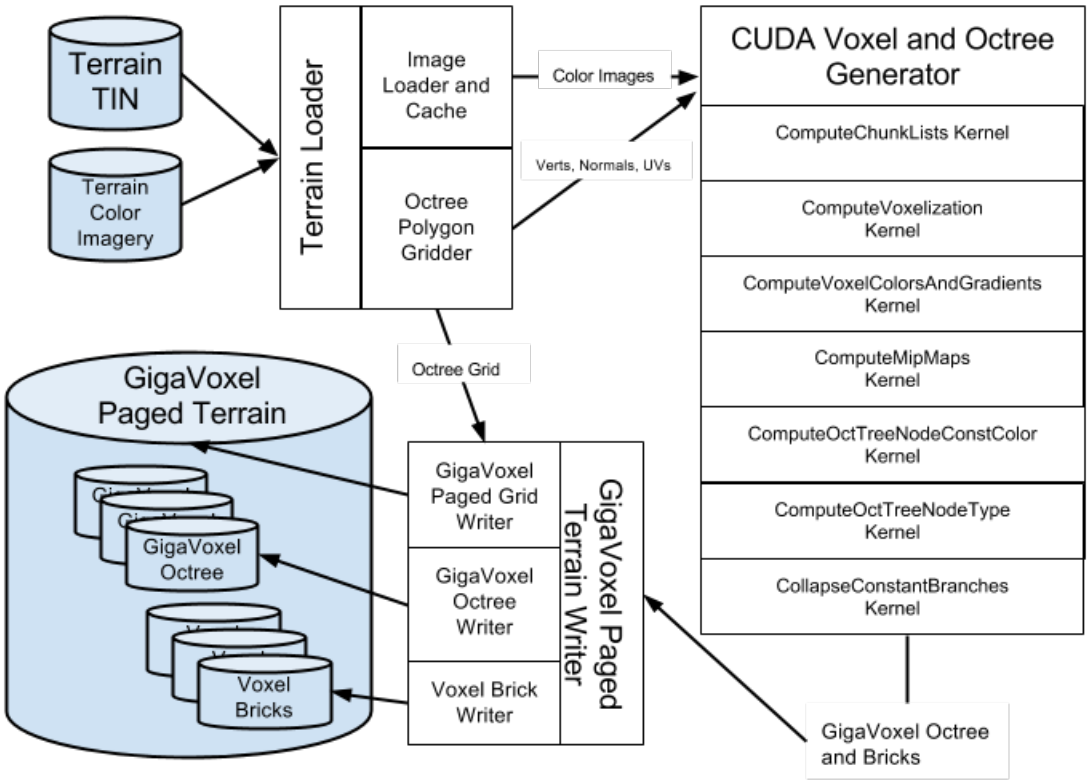

Figure 3.3. GigaVoxel Paged Terrain Generation System Data Flow

detail below:

1. Terrain Loader - this component loads the polygonal terrain and it's imagery and maintains a grid/list of the parameters describing the oct-trees that have been generated by the CUDA Generator component.

2. CUDA Voxel and Octree Generator - this component consists of a series of CUDA kernels responsible for converting polygonal data into GigaVoxel oct-tree data (listed below).

3. GigaVoxel Paged Terrain Writer - this component manages the file output operations.

2.5. GigaVoxel Paged Terrain Generation. The first step of the generation process is to determine the 2D layout of the 3D grid of GigaVoxel oct-trees, by dividing up the bounding box of the input data by the desired oct-tree output dimensions in order to compute the number of oct-trees, in the X and Y directions, required to encompass the input terrain's extents.

Input terrain that uses round Earth coordinate systems are handled by transforming the input polygons into a local coordinate system at the centroid of the input data with the the Y axis pointing towards the North pole, the X axis pointing due West, and both of them tangent to the Earth's gravity vector at the centroid.

Once the X and Y position and the width and height of each oct-tree grid cell has been determined then the Z dimension for the grid and the Z position for the oct-tree in each XY grid cell is computed based on the input data's height in that cell. The Z position is chosen such that the number of oct-trees required to encompass the height of the input terrain's polygons is minimized and such that the oct-tree is centered on the portion of the terrain that it encompasses. Figure 3.4 shows a side view of an example terrain to illustrate the dynamic nature of the Z positioning of each oct-tree in the grid. The positioning algorithm is described in more detail below.

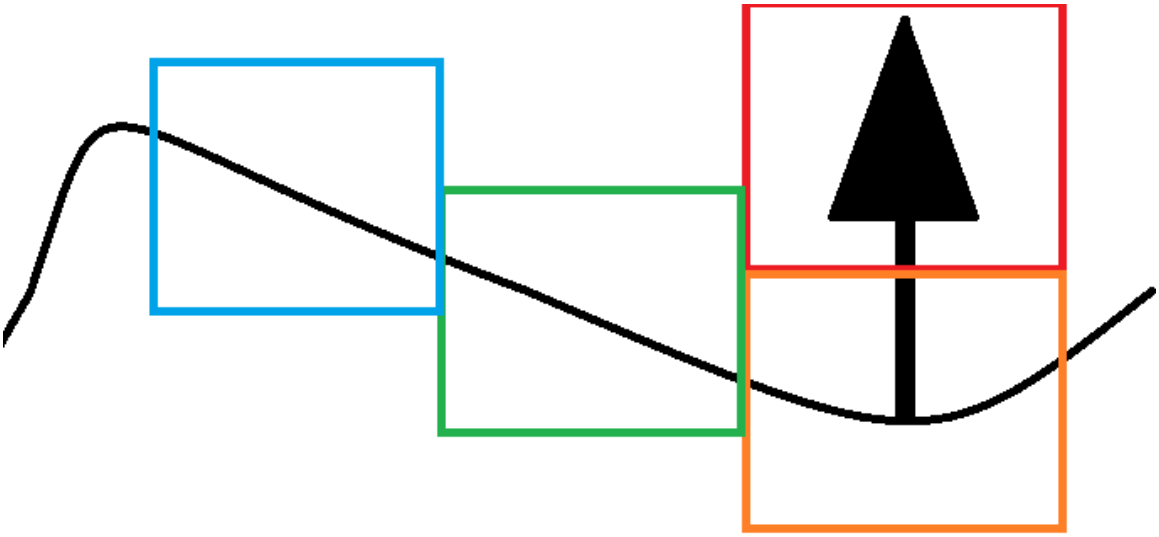

Figure 3.4. Terrain Profile Showing Stacking of Oct-Trees

1. Load required subset of input data - the 2D bounding region of the grid cell is used to load only the triangles, and the textures used by the triangles, contained or overlapped by the cell. The loading system maintains a fixed size LRU cache of previously loaded data to reduce redundant loading of data for neighboring

oct-tree's that require an overlapping set of input files. The triangles and texture color imagery are loaded into GPU memory in groups organized by texture.

2. For each oct-tree grid cell determine the number of oct-trees in the Z dimension that are required to encompass the input data's height.

3. For each oct-tree determined by step 2.

   (a) Compute Z component of centroid of oct-tree.

   (b) Generate GigaVoxel Oct-tree.

The final step, Generate GigaVoxel Oct-tree, is implemented as a series of CUDA kernels. The first kernel performs some pre-processing to setup the triangle groups for further processing by the voxel and oct-tree kernels. After the pre-processing step, the full resolution voxel grid for each oct-tree is generated in N^3 chunks, where N can be any power of 2 that is less than (or equal) to the full resolution voxel grid. The value chosen for N depends on the amount of memory that the GPU provides.

1. ComputeChunkLists - for each triangle group assign each triangle to a chunk list (i.e. the list of triangles that overlap each chunk).

2. For each N^3 chunk of the total voxel grid.

   (a) For each triangle group.

      i. ComputeVoxelization

      ii. ComputeVoxelColorsAndGradients

   (b) ComputeMipMaps

   (c) ComputeOctTreeNodeConstColor

   (d) ComputeOctTreeNodeType

3. CollapseConstantBranches

The ComputeChunkList kernel uses a single GPU thread per triangle to compute the bounds (in voxels) of each triangle and assign the triangle to one or more chunk lists. The bounds of each triangle are written to an array corresponding to the order of the triangles in the triangle group list. The triangle bounds are used in the ComputeVoxelization kernel for the voxel-triangle overlap test. The chunk lists consist of one array of triangle indexes per N^3 chunk of the full voxel grid. The chunk lists ensure that only the triangles that are definitely overlapping a chunk

are processed by the voxelization kernels. The number of items in each chunk list is computed via the CUDA atomic add function and stored in a chunk list count array, that, which is read back to main system memory prior to execution of the voxelization kernels in order to drive the dimensional configuration and launching of the voxelization kernels.

After the bounds and chunk list setup, the voxel and oct-tree data is generated one N^3 chunk at a time by the two voxelization kernels, which are named ComputeVoxelization and ComputeVoxelColorsAndGradients.

The ComputeVoxelization kernel performs the triangle voxel overlap test for each triangle and each voxel in the voxel grid chunk. The dimensions of the ComputeVoxelization kernel are configured such that the cube root of the max number of threads per block supported by the GPU is used for all three dimensions of the block size, in order to assign one GPU thread to each voxel. The grid size is configured such that there are total of N^2 threads spread across as many blocks as required in the XY dimension. The Z dimension of the grid is set to the number of triangles in the current chunk list. For example, if the GPU supports 512 max threads per block and there are 128 triangles in the chunk list then ComputeVoxelization kernel would be invoked like so:

$$ComputeVoxelization < uint3(32, 32, 128), uint3(8, 8, 8) > (...);$$

Each instance of the ComputeVoxelization kernel is responsible for testing for overlap of a single triangle on the voxel at cell X, Y, and Z, which are computed:

$$TriangleIndex = CurrChunkList[blockIdx.z]$$

$$X = (blockIdx.x * blockDim.x) + threadIdx.x$$

$$Y = (blockIdx.y * blockDim.y) + threadIdx.y$$

and the column of eight voxels starting at Z:

$$Z = zOffset + threadIdx.z$$

where zOffset is an outer loop variable that is input to the ComputeVoxels kernel and is used to iterate the entire N^3 voxel grid from the minimum Z, in voxels, of the triangle group's bounding box to its maximum Z. In order to optimize for triangle groups whose bounding regions are larger in the Z dimension, for instance tree or building models, the algorithm selects the smallest of the three dimensions to use for the loop (i.e. zOffset could be yOffset if the smallest dimension of the triangles being processed is in the Y dimension). The triangle index within the triangle group for

each GPU thread is the blockIdx.z parameter, that is, each thread block processes a single triangle.

Each ComputeVoxelization kernel instance uses the box overlap test described previously to detect if a voxel overlaps a triangle. If it does then the triangle's index is added to the triangle overlap list for the voxel. The triangle overlap list is stored in 12 bits of a 3D array of 64 bit integers depending on the number of triangles in the chunk list. Each element of the array represents a list that can encode triangle indexes of up to 5 triangles. Supporting only 4 to 8 overlaps per voxel might seem like not enough to store all of the overlaps for each voxel, but the fact that the triangles are grouped into small groups by texture, which are further paired down by the chunk list, and the fact that the voxel size is very small, in practice, this results in 1 to 2 overlaps per triangle group. Each thread synchronizes writes to the list by using a second 3D array, the triangle overlap count list, to keep track of the number of items in each voxel's overlap list. When a GPU thread detects an overlap, it uses the CUDA atomicAdd function to obtain the write index for the current overlap and to increment the count for the next overlap write index.

The triangle overlap count list and the 3D array that tracks the list of triangle indexes that overlap each voxel is used in the next kernel, ComputeVoxelColorsAndGradients, to compute the voxel color and gradient. For this kernel a single GPU thread is assigned to each voxel in the N^3 chunk by spreading the threads evenly across the available blocks in the grid and maximizing the number of threads per block using the same cube root method used by the ComputeVoxelization kernel. The X, Y, and Z indexes of each thread's voxel is computed as such:

$$X = (blockIdx.x * blockDim.x) + threadIdx.x$$

$$Y = (blockIdx.y * blockDim.y) + threadIdx.y$$

$$Z = (blockIdx.z * blockDim.z) + threadIdx.z$$

Each thread loops through the the list of overlapping triangles stored in the triangle overlap list. For each triangle the color is looked up from the color texture using the input texture map and texture map coordinates. The interpolated texture coordinates are computed by projecting the overlap triangle along the dominant axis of the its face normal and then computing the barycentric coordinates of the voxel's center with respect to the 2D projection of the triangle. The barycentric coordinates of the voxel's center are then used to compute the interpolated texture coordinates to use for texture lookup via the following equation:

$$V = (uvB - uvA) * baryY$$

$$U = (uvC - uvA) * baryX$$

$$interpUV = uvA + U + V$$

Where uvA, uvB, and uvC are the texture coordinates associated with the three vertices of the overlapped triangle and baryX and baryY are the barycentric coordinates of the voxel's center. The same computation is used to compute the voxel's gradient if the triangle has per vertex normals. The color from each triangle is added to a N^3 array of RGBA values. Similarly the voxel gradient is added to a N^3 array of XYZ values. After adding the contribution of color and gradient for each overlapping triangle the triangle overlap count is used to compute the average color and gradient respectively. One thing to note is that if the texture lookup results in a color that is completely translucent (i.e. alpha is zero) then it is does not contribute to the color or gradient computation.

After the contribution of all the triangles in every triangle group is computed and stored in the two N^3 arrays, then, for each level of the oct-tree, a MIP-map is computed. This is handled by a kernel function called, ComputeMipMap, that takes as input the parent level's color and gradient data and writes it's output to a 3D voxel grid that is half the dimensions of the parent's. Each GPU thread is assigned a single voxel in the ouput voxel grid and uses a custom box filter to compute a weighted average from the parent voxels. The weight for the weighted average computation comes from the alpha component of the parent voxels, that is, a higher alpha component carries more weight than a lower alpha. Furthermore, for MIP-map levels 2 and above, the output is actually computed from the parent level that is two levels higher than the current level. This results in much better accuracy in the MIP-map computation. So for level N=1 the weighted average comes from the 8 (2x2x2) voxels at N=0 (i.e. root level). However, for N >= 2 the weighted average comes from the 64 (4x4x4) voxels of the level N-2 voxel grid. Additionally, the alpha average is computed in a way that prevents the translucent voxels from causing the object modeled by the voxels from fading inordinately fast:

$$alphaAvg = alphaSum/sumCount$$

$$alphaAvgPow = pow(alphaAvg, 1 - alphaAvg)$$

Where the alphaSum is the sum of the alpha components from the parent or grand-parent voxels and sumCount is the number voxels from the parent (i.e. 8) or grand-parent (i.e. 64). By using the pow function the alpha component fades out more

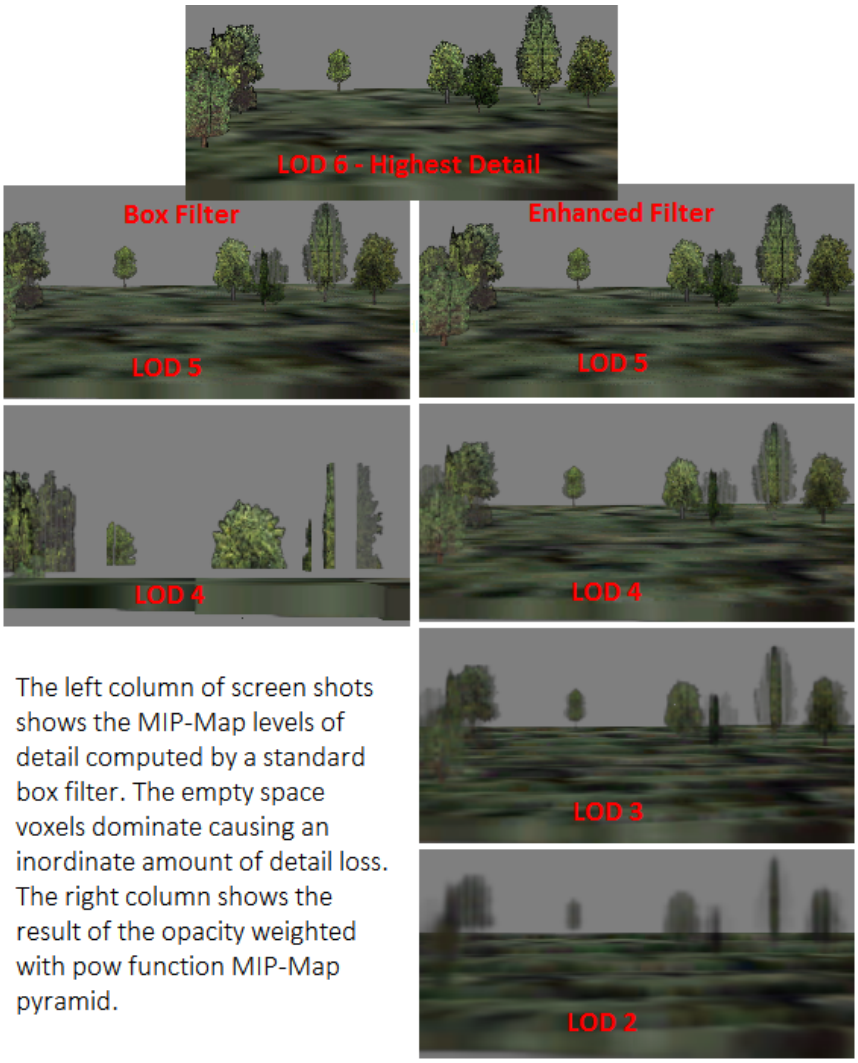

Figure 3.5. Difference between standard box filter and enhanced box filter

slowly as the amount of completely translucent voxels (i.e. voxels with no overlaps) mix with the non-translucent voxels. If this pow() function is not used then the effect is that the voxelized object appears to shrink and become inordinately translucent in the lower resolution MIP-maps. By applying the pow() function it causes the lower resolution MIP-maps to appear fuzzier, which is the desired effect.

The last operation performed on each N^3 chunk is to compute the node type and constant color for each node in the portion of the oct-tree corresponding to the current chunk of full voxel grid. The levels of the oct-tree are traversed one at a time from leaf to root using recursion on the CPU. At each recursion/oct-tree level a ComputeOctTreeNodeConstColor kernel function is used to determine

the constant color that would be used to represent each node if it is determined to be a constant node. Then the ComputeOctTreeNodeType kernel uses this value to determine whether a node is or is not a constant node. For the ComputeOctTreeNodeConstColor, each GPU thread is assigned a node in the current level of the oct-tree. The X, Y, and Z index of the node in the oct-tree is computed as such:

$$X = nodeXOffset + ((blockIdx.x * blockDim.x) + threadIdx.x)$$

$$Y = nodeYOffset + ((blockIdx.y * blockDim.y) + threadIdx.y)$$

$$Z = nodeZOffset + ((blockIdx.z * blockDim.z) + threadIdx.z)$$

where nodeXOffset, nodeYOffset, and nodeZOffset are computed based on the offset of the current N^3 chunk within the full voxel grid. The ComputeOctTreeNodeConstColor kernel function uses the X, Y, Z of the oct-tree node to sample a single value from the set of voxel colors corresponding to the area covered by the oct-tree node. The thread stores that value in 3D array of RGBA values whose dimensions correspond to the dimension of the current level of the oct-tree (i.e. 1x1x1 at level 0, 2x2x2 at level 1, etc.). This array then becomes the input to the ComputeOctTreeNodeType kernel function, which, like the ComputeVoxelColorsAndGradients kernel function, assigns a single GPU thread to each voxel in the N^3 chunk. In this case each kernel instance is responsible for detecting that its oct-tree node is non-constant by comparing the node's constant color to the voxels assigned to it. Each GPU thread determines which oct-tree node it is responsible for by the following equation:

$$brickX = fullVoxXYZ.x/brickDim.x$$

$$brickY = fullVoxXYZ.y/brickDim.y$$

$$brickZ = fullVoxXYZ.z/brickDim.z$$

where fullVoxXYZ is equal to the XYZ offset of thread's assigned voxel relative to the full voxel grid. Each GPU thread/kernel instance uses the brickX, brickY, and brickZ to look up the constant color computed by ComputeOctTreeNodeConstColor to compare it to its assigned voxel. If the values are different then the node is classified as a non-constant node by writing a 1 to 3D array whose dimensions, like the constant color 3D array, correspond to the dimensions of the oct-tree at the current level of the oct-tree. Thus, if any of the GPU threads detect a difference between the constant color and the color of the voxel that they are assigned, then they mark the oct-tree

node as non-constant. There is a special case for brick border voxels. Voxels at the border of a brick also require checking for non-const-ness for the oct-tree nodes whose bricks share the particular border. So in those cases the GPU thread compares the constant color of those neighboring nodes to its assigned voxel's color as well. After the node type is computed then each non-constant node's brick is dumped into the brick file.

After the full voxel grid has been traversed in N^3 chunks then the final CUDA kernel, CollapseConstantBranches, identifies constant branches of the oct-tree that can be eliminated. A constant branch is one where the root of the branch is a constant node and the root's children, grand-children, etc. are all constant nodes. The input to this function is the node type 3D array computed by ComputeOctTreeNodeType. The output is another 3D array of integers, where like the constant color 3D array, each element corresponds to a node in the oct-tree. The integer for each node indicates if a particular node is a constant branch, where a value of zero indicates that the node is the root of a constant branch and a value of one indicates that it is not. The constant branch array is initialized to zero to indicate that all nodes in the tree start off as constant branch root nodes. The CollapseConstantBranches kernel is called recursively at each level of the oct-tree starting at the bottom of the tree (i.e. leaf node level). Each GPU thread is assigned to a particular node in the oct-tree at the current level of the tree. The algorithm is described below.

1. Look up my node's type in the node type array.
2. If my node's type is constant then do nothing
3. Else if it is non-constant then write 1 to my parents node's cell in the constant branch 3D array.

The CollapseConstantBranches kernel produces a list of nodes whose children can be omitted from the oct-tree because they can be accurately represented as a single constant color. The final step in the generation process is to use the 3D array of node types, constant colors, and the constant branch array to dump the oct-tree node data to a file.

3. GigaVoxel Paged Terrain Rendering System

The GigaVoxel Paged Terrain Rendering System design builds off the data structures and algorithms described by Cyril Crassin in the paper "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering." [12] with enhancements adapted from the fields of large scale terrain rendering and scene graph rendering (see chapters 3 and 4. This section gives an overview of the rendering system as a

whole. The next section reviews, in detail, the base GigaVoxel algorithms and data structures. Finally, the last few sections detail the extensions to the base algorithms and data structures.

3.1. GigaVoxel Paged Terrain Rendering System Overview. Figure 3.6 depicts the major components of the rendering system along with the data flow through the rendering system components. The components and data flow are described in more detail below.

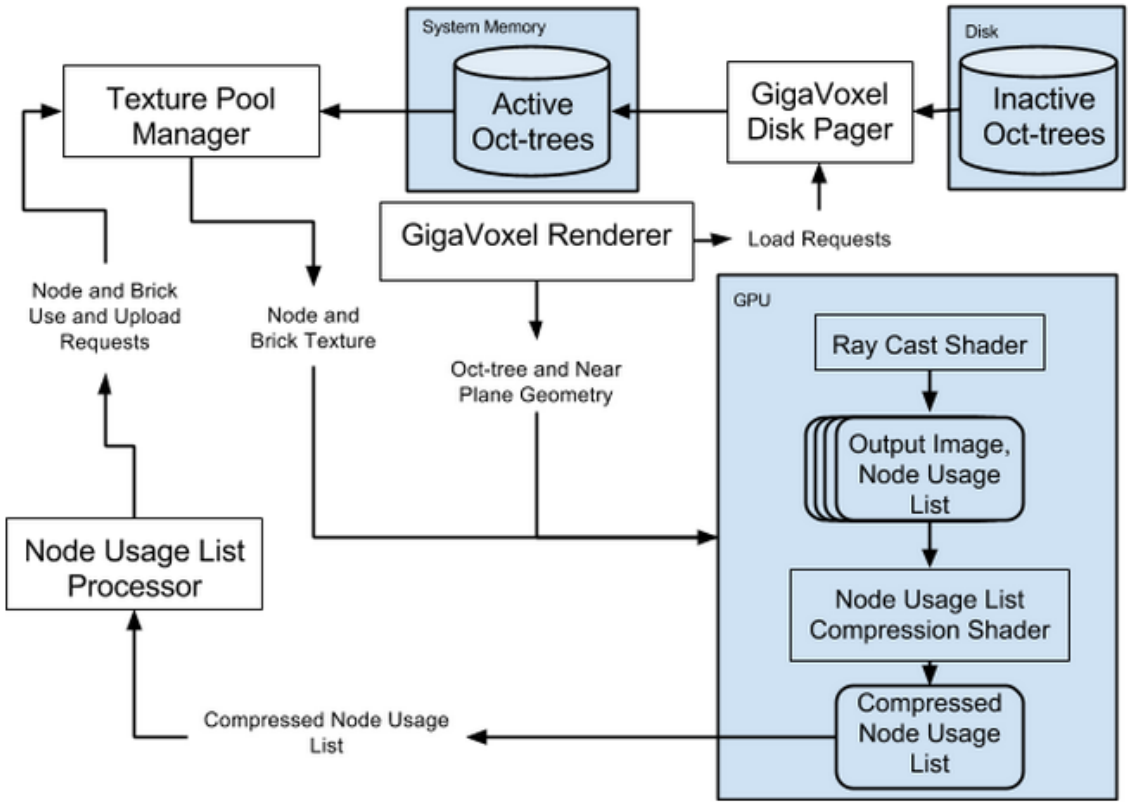

Figure 3.6. GigaVoxel Paged Terrain Rendering System Overview

1. GigaVoxel Renderer
   (a) Identifies the list of oct-trees needed for rendering.
   (b) Manages OpenGL state for execution of pixel shaders and blending of oct-tree output images.
   (c) Manages read back of compressed node usage lists.
   (d) Submits load requests to Disk Pager.

2. Disk Pager - loads individual oct-trees from disk into system memory.
3. Ray Cast Shader
   (a) Renders oct-trees to an image.
   (b) Generates list of traversed oct-tree nodes.
4. Node Usage List Compression Shader - compresses the node usage list generated by the Ray Cast Shader.
5. Node Usage List Processor
   (a) Iterates compressed node usage lists.
   (b) Submits usage and upload requests to Texture Pool Manager.
6. Texture Pool Manager - uploads oct-tree nodes and voxel bricks to the GPU.

The primary components and their logic, data structures, and algorithms are described in more detail in the sections that follow.

3.2. GigaVoxel Oct-Tree Overview. The GigaVoxel rendering algorithm relies on an oct-tree data structure stored in both the main system memory and the GPU memory. The main system memory stores the full oct-tree, that is, the entire oct-tree including all the nodes and all the bricks. The GPU data structure stores only the portions of the oct-tree needed for rendering the current viewpoint. The in-view nodes and bricks of the oct-tree are copied from main system memory into the Node Pool texture and the Brick Pool texture respectively. The pool textures ensure that the amount of GPU memory used by the rendering system is limited to a constant pre-runtime configured amount. Section 2.1 describes in detail the format of the texture pools. The oct-tree and MIP-map brick pyramid data structures stored in the texture pools allows the view rays to drive the loading of the required oct-tree nodes and bricks from main system memory into GPU memory and to render realistic images from out-of-core voxel data. The oct-tree consists of either constant nodes or non-constant nodes (refer to 2.1 for more detail). Non-constant nodes have a pointer to a brick, which is encoded as an XYZ offset into the Brick Pool texture. Constant nodes, on the other hand, do not require a brick and can be represented by a single color and opacity. The non-constant nodes from each level of the oct-tree point to bricks that come from the corresponding level of a 3D MIP-map generated from the full resolution voxel grid upon which the oct-tree is based (see figure 3.1 and figure 3.7).

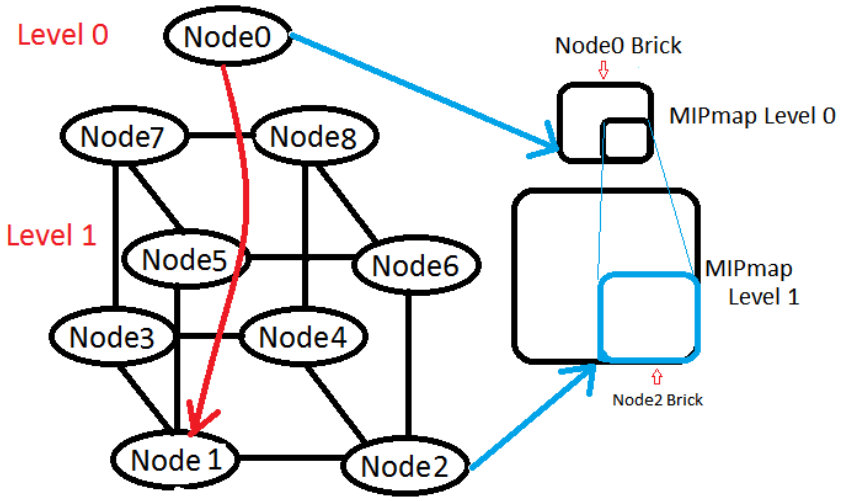

Figure 3.7. GigaVoxel Oct-tree with MIP-map Pyramid

3.3. GigaVoxel Ray-Casting. Rendering an image from a GigaVoxel oct-tree consists of marching a single view ray for each pixel in the output image through the oct-tree structure. Each ray computes the color and opacity of its pixel in the output image by sampling from non-constant node bricks or, for constant nodes, by computing the node's color and opacity contribution computationally. The rays originate from proxy geometry that encompasses the bounding box of the oct-tree or, if the camera is inside the bounding box, a pair of triangles representing the near view plane. The OpenGL stencil buffer is used to determine which of the two ray origination methods takes precedence over the other. This is done by first drawing the bounding box geometry, with back face culling on, and then drawing the near plane geometry. the stencil buffer test allows only the fragments from the closest of two proxy geometries to execute the pixel shader stage and thus only those rays affect the output image.

Each ray marches across the oct-tree's volume by traversing down the oct-tree until it arrives at a node with the correct level of detail. To determine which level of the tree to stop at, the projected size of a voxel at the current oct-tree level is compared to the size of a pixel in the output image. If the projected size is less than or equal to one pixel then the correct level has been reached. The ray can now start computing the node's contribution to the output color and opacity via brick sampling or via constant color computation. Because each level of the oct-tree, from

the root down to the leaf node level, represents the voxel data at progressively higher resolution each ray's traversal across the brick is optimized because the sampling rate is adjusted accordingly based on the voxel size at each level, i.e. lower resolution voxels are sampled at a lower rate [22]. After the ray computes the contribution of the node then the exit point from the node is used as the new starting point for another descent of the oct-tree node hierarchy. This process repeats until the entire oct-tree has been traversed by each view ray or until the opacity of the output pixel has become sufficiently saturated.

Descent of the tree is simple because the ray's current location within a node converts easily into a pointer to the root's child that should be visited next in order to continue the downward traversal. For example, given a ray position in the root node's local coordinate system $P \in [0,1]^3$ the offset to the child node containing P is simply $Q = int(P * 2)$. This is because the children of each node are stored in a contiguous 2^3 block in the node pool texture. Each node has an XYZ index to its lower left child, in the node pool. Given a child pointer C, the XYZ coordinates of the child needed for the descent can be obtained by $P = C + int(P * 2)$. The new starting location within the child node is computed by $P = (P * 2) - Q$. From the new starting location the oct-tree descent continues until the ray reaches the desired oct-tree level.

3.4. GigaVoxel High Quality Filtering. The GigaVoxel rendering technique obtains realistic results by computing the volume rendering integral using a Riemann sum method computed along a simulated cone traced per pixel out in the direction of the camera [22]. Cone tracing is a more accurate method to determine the color of a pixel because a pixel has an area much larger than that of a single ray and the voxels that affect it's color should not be limited to those that are intersected by a single ray, but by the voxels encompassed by the cone extending from the camera through the edges of the pixel [13]. Standard ray tracing algorithms simulate cone tracing via multi-sampling, that is, generation of multiple rays per pixel with slightly different offsets within the pixel and traversing the voxel data with all of the rays. This method, however, has a large negative impact on performance and memory consumption and is still not as accurate as cone tracing [22]. The GigaVoxel technique simulates the cone with a single ray by using quadri-linear interpolation technique when sampling voxels from the oct-tree bricks. Quadri-linear interpolation is accomplished first by representing each level of the oct-tree at a progressively lower resolution, aka a 3D MIP-map pyramid. Each ray samples from a brick using the graphics hardware's built-in bi-linear interpolation capability. Furthermore, the traversal of the oct-tree stops at the level of the tree appropriate for the current view point (see section 3.3). If the projected size of the voxel at the selected level lies

somewhere between the size of a voxel at the current level of the oct-tree and the size of a voxel at parent level then both levels are traversed and the sampled values are averaged together to provide quadri-linear filtering. This quadri-linear filtering results in highly accurate output images devoid of aliasing artifacts that would normally be present in images rendered from extremely high resolution voxel data [13].

3.5. GigaVoxel Ray Guided Cache Updates. Each GPU ray drives the loading of the texture pools. Each ray descends down the oct-tree structure until it reaches the node at the required level of detail. The ray then adds the XYZ index of the final node reached to the node usage list. The node usage list provides a mechanism for the rays to communicate back to the rendering system the list of nodes that were and/or are needed for rendering. If a node is added to the node usage list this communicates to the Texture Pool Manager that the node is active and should be kept in the node pool. If the required level of detail cannot be reached because a node's children have not been loaded into the node pool then, in addition to the node's XYZ index, the node sets a bit in the node usage list indicating that the node's children should be loaded into the Node Pool texture. It is up to the Node List Processor(s) to communicate to the Texture Pool Manager the list of nodes from the node list and the ancestors of these nodes (i.e the parents, grand-parents, etc. of the node list nodes) so that the nodes and bricks needed by the rendering system are available in the appropriate texture pools. Furthermore, during the descent each ray keeps track of the most recently encountered node, whose brick is currently loaded in the Brick Pool, and the node's parent so that if the required LOD cannot be reached or if the brick of the node at the required LOD is not currently loaded into the Brick Pool, then the ray can traverse the brick that came from higher up in the tree instead. This guarantees that some output color will be produced by all rays, while the Texture Pool Manager works to get the needed LOD loaded. Lastly, it should be noted that the Texture Pool Manager ensures that each oct-tree's root node brick is always present in the brick pool so that a ray always has access to at least one brick for generating an output image.

The node list is implemented using three extra render targets (note that the first render target is reserved for the output image). The three extra render targets are RGBA 32 bit unsigned integer textures attached to an OpenGL FrameBufferObject. Using 10 bits per XYZ component each texel of the node list texture can encode four XYZ node indexes along with an extra bit used to encode the need for more detail. Thus the three textures together can encode up to twelve node indexes per ray. However, twelve nodes is not sufficient for tracking the ray's full traversal. In order to encode more than twelve nodes, the algorithm takes advantage of the spatial coherence of neighboring rays by having each block of 2x2 rays encode different sets of

nodes. The upper left ray in each 2x2 block encodes the first twelve visited nodes, the upper right ray encodes the second twelve, the lower left ray encodes nodes 25 to 36, and the final ray encodes nodes 37 to 48. Furthermore, each frame, a rolling update scheme is used, on even frames the first 48 traversed nodes are recorded, then on odd frames nodes 48 to 96 are recorded. Generally, 96 is enough to track the traversal of most oct-trees because the majority of rays do not have to traverse across the entire oct-tree because of early ray termination due to opacity saturation. However, should that not be the case, the rolling window can be extended to more frames.

3.6. GigaVoxel Node List Compaction. After the rays finish generating the node list it would be impractical to read back the list from the video memory into system memory because the amount of data would be too large to transfer while still maintaining a consistent frame rate. Thus, efficient compression of the node list data is essential. The GigaVoxel technique uses a multi-pass compression algorithm utilizing multiple pixel shader invocations driven by near plane proxy geometry. The first compression pass removes duplicate node indexes from neighboring pixels. Removal of duplicate node indexes is optimized to take advantage of spatial and temporal coherence of neighboring pixels/rays, which is possible due to the fact that neighboring texels of the node list texture most likely will have have a significant amount of duplicate indexes because neighboring rays will visit many of the same nodes. Furthermore, neighboring rays will also likely visit each node in nearly the same order. Therefore, the duplication reduction process only compares nearby items in the node list of neighboring texels. Specifically, for the i'th element in the list only the (i-1)'th, the i'th, and the (i+1)'th elements in the neighbor's list are compared. Neighbors are defined as the texels to the left, upper left, directly above, and upper right of the each texel. If, for the i'th element, a match is not found in the neighboring list then the output is a single set bit in the i'th element of the output pixel. The output pixel is a 32 bit unsigned RGBA texture. This output pixel then becomes the bit vector that is used as the selection mask input to the HistoPyramid algorithm, which is a data compaction algorithm specifically designed to take advantage of the highly parallel architecture of the GPU (refer to [75] for a detailed description of the HistoPyramid algorithm). After the duplication removal and the HistoPyramid reduction, the node list usually contains only 2 to 3 entries, which allows the final compacted texture to fit into a single low resolution 32 bit RGBA texture [12]. This smaller texture can then be downloaded from the GPU to main system memory more efficiently.

4. GigaVoxel Extensions

In order to better support rendering of a large scale paged voxel terrain, several enhancements to the base GigaVoxel oct-tree rendering algorithm were implemented. The enhancements included the development of an OpenSceneGraph inspired, scene graph based, view frustum culler and a database pager thread [5]. Furthermore, to compensate for the increased workload of multiple oct-trees the processing of the node lists is handled in multiple background Node Usage List Processor threads. Lastly, the base rendering algorithm was augmented to handle blending of multiple intermediate images, generated from the ray casting of multiple oct-trees, into a single final image (as opposed to the standard GigaVoxel algorithm where there is only a single oct-tree and thus a single output image).

4.1. View Frustum Culler. The view frustum culler's purpose is to ensure that only the visible oct-trees are processed by the rendering system. The culler is implemented according to the visitor pattern described by Gamma, et al. [24] and is similar in implementation to the OpenSceneGraph's culling system [5]. The culling visitor visits each node in the scene graph, which is constructed, by the database pager, from the oct-tree grid XML file described in chapter 5. When it visits an OctTree node whose bounding sphere overlaps or is contained in the current view frustum then it either adds it to its render list or, if the oct-tree referenced by the node needs to be loaded from disk, then it submits a load request to the database pager thread. Furthermore, upon visitation, the culler updates the node's last access frame-stamp. The last access frame-stamp is used to identify and unload stale oct-trees from main memory. At the conclusion of the traversal the render list is passed to the rendering system for drawing.

4.2. Database Pager. The database pager thread's purpose is to assist the culler in loading into system memory the minimal set of oct-trees required for rendering. The culler sends load requests that go into a priority queue sorted by distance from the camera. To compensate for camera movement, the culler continually recomputes and updates this distance while the oct-tree load request is in the load request queue. After loading an oct-tree, the database pager adds it to the pending load list. Every frame the primary (renderer) thread transfers the items in the pending load list to the the scene graph.

4.3. Node List Processors. The rendering system draws multiple oct-trees each frame and each oct-tree generates its own node usage list. In order to process the node usage lists of all them, without impacting the frame-rate, the node list processing is done in several background threads. The number of background processor threads

is dynamically computed at startup such that as many of the CPU cores as possible are used, while still leaving at least two cores for the primary rendering thread and the database pager thread.

At the conclusion of each frame, the render thread reads back each active oct-tree's node usage list from GPU memory and adds each one to one of the Node List Processor thread's pending queue. The Node List Processor threads process their pending queue in LIFO order. LIFO order is used because the most recently added lists are more immediately relevant to the current camera view. Furthermore, in keeping with the strategy of processing newer lists first, the pending queue size is capped so that older lists are kicked out by newer lists.

Each Node List Processor iterates through its current node list and submits load requests to the Texture Pool Manager or if a node in the list has already been loaded into a pool it simply notifies the Texture Pool Manager that the node is active (i.e. needed for rendering). This prevents the node from being replaced by a new node being loaded into the pool.

The Texture Pool Manager maintains a sorted list of the nodes currently loaded in its pool textures. The list is sorted such that the most recently active nodes are first and less active or inactive nodes are at the end of the list. It maintains this sort order by moving a node, when notified by the node list processor, from its current location on the pool list to the front of the list. This keeps the front of the active list populated by the active nodes and the stale nodes will naturally migrate to the end of the list so that their spot in the texture pools can be used for new nodes that need to be loaded.

4.4. Asynchronous Upload and Download. The addition of multiple rendered oct-trees heightened the need for optimal use of the limited bandwidth between the system's main memory and the graphics hardware's memory. In order to maintain a consistent frame rate all uploads to the GPU texture memory and downloads from the GPU memory are implemented using asynchronous transfer via dual and/or triple buffered OpenGL Pixel Buffer Objects (PBO). This increases the amount of main system memory required by the rendering system, but only by a small fraction of the total size of the memory required overall by the rendering system. Dual and tripled buffered, ping pong style asynchronous PBO upload and download is implemented by requesting the upload or download to the target PBO in frame N and then not attempting to access that data directly until frame N+1, for dual buffered PBO, or frame N+2 for tripled buffered PBOs. This prevents the graphics hardware from stalling in order to satisfy an upload or download request.

All of the uploads required by the system (i.e. uploading nodes and bricks to a texture pool texture on the GPU) are implemented with dual buffer PBOs, however

analysis of the performance of the system revealed that triple buffering the download PBO for the node usage list data prevented the readback of that data from stalling the rendering pipeline and becoming the bottleneck in the performance of the system. The triple buffered system thus provided enough separation between the download request and the access/reading of the PBO (via glGetBufferSubData) so that the graphics system had time to finish with its rendering tasks before having to perform the transfer.

4.5. Rendering Modifications. The final modification made to the base GigaVoxel rendering algorithm was to implement blending of each oct-tree's individual output image together with the other oct-tree output images. This is accomplished by rendering the oct-trees in front to back order. The sorting of the oct-trees is accomplished by the view frustum culler, which sorts the oct-trees by distance from the near plane while it is building the render list. The oct-trees are rendered in this order using the OpenGL blending mode: glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA) [22].

CHAPTER 4

Results

The results presented in this section describe the run-time rendering performance and resource utilization of the prototype GigaVoxel paged terrain rendering system implemented as described in chapter 3. The tests were performed on two distinct regions of terrain taken from a source data set that consisted of a TIN terrain skin generated from a 0.3m Digital Elevation Model (DEM) and thousands of terrain detail models (buildings, tree, other structures and vegetation, etc.). The first region consisted of terrain and mostly vegetation detail models. The second region consisted of terrain and mostly buildings models (see figure 4.1). Both regions covered

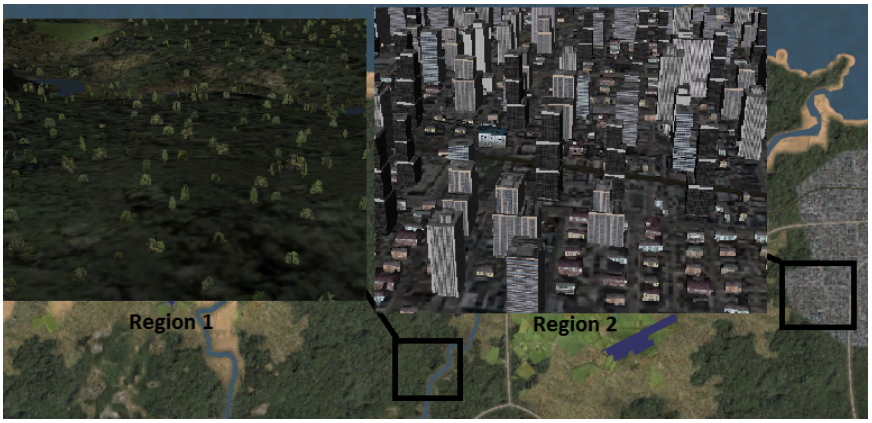

Figure 4.1. Source Data Showing Voxelized Regions

about $2km^2$ and were voxelized using two different resolutions, $1024^3$and 2048^3, for the purpose of comparing performance and data size differences due to voxel resolution. The tests were performed on an Intel Core i7 3.4GHz CPU with 16GB of RAM and an NVIDIA GeForce GTX 760 graphics card with 2GB of VRAM. The source data was an OpenSceneGraph WGS84 geotypical terrain database produced and distributed for non-commercial use by TrianGraphics (triangraphics.de).

1. Data Size Results

The source data was voxelized at a resolution of 0.1m per voxel. The 1024^3 oct-trees resulted in an 18x18 grid of oct-trees in the first region. Several of the cells in the grid required two oct-trees to be stacked on top of each other in order to encompass the full extent of the height of the terrain and terrain detail models in the oct-tree grid cell, which resulted in a total of 357 oct-trees in all. Region 2 required more oct-trees than the first because it consisted of several relatively tall building models. It ended up consisting of 392 oct-trees (18x18 with 68 stacked oct-trees). By comparison region 1 and 2 both required exactly 9x9 oct-trees at the 2048^3 resolution (i.e. zero stacked oct-trees) because the 204.8 meter extents of the oct-tree completely encapsulated the terrain and building models in almost all cases.

The grid of 1024^3 oct-trees averaged about 120MB each on disk for region 1 and 160MB for region 2 with a total size of 36GB and 57GB respectively. In contrast the grid of 2048^3 oct-trees averaged about 420MB per oct-tree in region 1 and 520MB per oct-tree in region 2 for a total of 34GB and 42GB respectively.

2. Rendering Results

The rendering performance results were measured using an 800 by 600 pixel

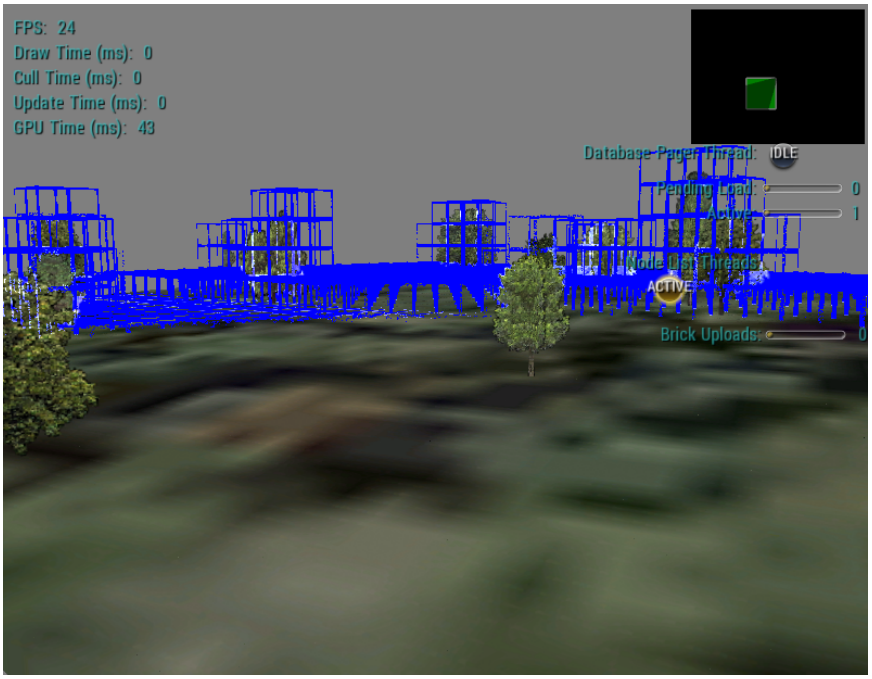

Figure 4.2. Single GigaVoxel oct-tree with majority of rays requiring full traversal

display window. Tests were performed on a small data set initially to determine a

baseline performance expectation and then on the full data sets. Testing a single 2048^3 oct-tree and a nearly equivalent 2x2 grid of 1024^3 oct-trees generated from terrain in region 1 and region 2 revealed that the performance on small data sets depends largely on the data and the viewpoint on the data. The test results showed a performance range between 20 to 30 frames per second in some view configurations on up to 60 frames per second depending entirely on the number of pixels that generate terrain intersecting rays, the depth each ray traversed into the oct-tree(s) before opacity saturation, and the oct-tree depth each ray descended in order to reach the required tree LOD for brick traversal. Figure 4.2 shows the rendering performance on

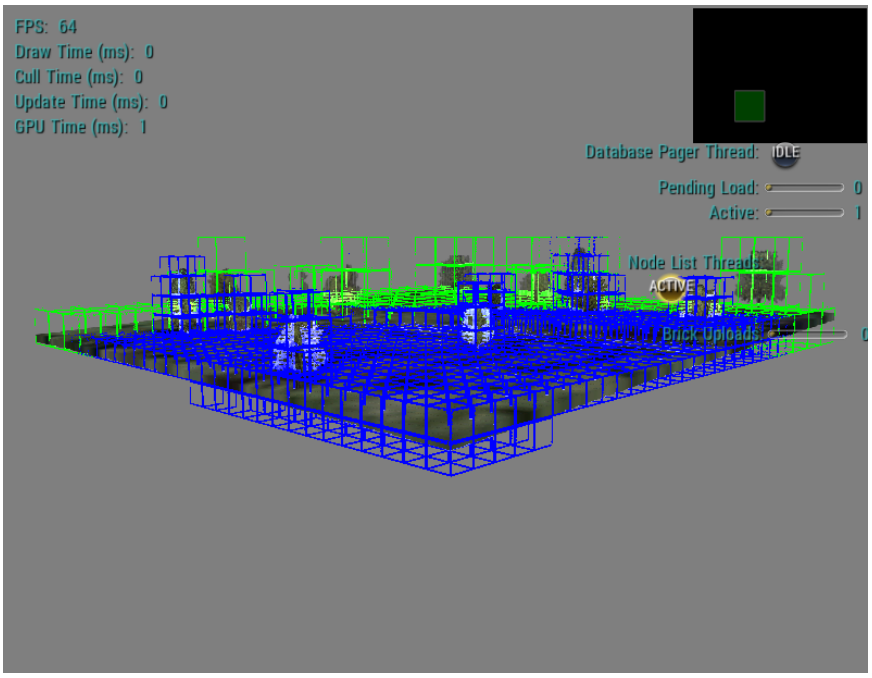

Figure 4.3. Single GigaVoxel oct-tree with no rays requiring full traversal

a single GigaVoxel oct-tree where the majority of the rays intersected the terrain and required a full traversal down to the leaf nodes or one level above the leaf nodes (one level above the leaves depicted with blue boxes). Figure 4.3 shows the same oct-tree rendering at 60 frames per second after moving the camera far enough away such that none of the rays traverse to a leaf node and most of the rays skip through empty space or miss the oct-tree entirely. Figure 4.4 shows a screen shot of the NVIDIA OpenGL debugger after capturing the timing information for a single frame, the thicker blue bar in the frame event graph corresponds to the ray traversal portion of the rendering algorithm, clearly indicating that the ray casting stage, as opposed to the usage list compaction and the usage list download, was the primary bottleneck in the rendering

performance. So it was fairly clear that on smaller data sets the rendering performance

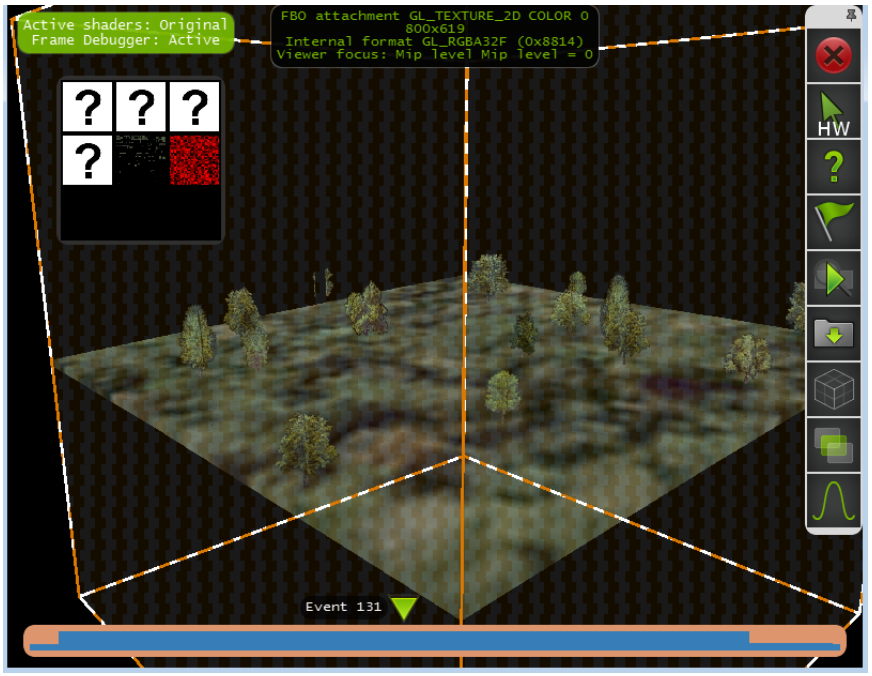

Figure 4.4. Screen shot of the NVIDIA OpenGL frame debugger showing timing of the various stages of the rendering algorithm

is pixel shader bound by the ray cast stage of the pixel shader.

The rendering performance on the full data set consisting of the grid of 1024^3 or 2048^3 GigaVoxel oct-trees revealed that 2048^3 oct-trees were more efficient for longer view ranges than the 1024^3 oct-trees. Essentially, as the view range increased the management, on the CPU-side (i.e. culling, unloading, loading, downloading usage lists, etc.), of the larger number of oct-trees required for the smaller 1024^3 oct-trees became the bottleneck in the performance. At a view range of 600 meters the performance of the 2048^3 data set averaged around 10 to 15 frames per second, again depending on the view and data currently in view. Whereas the 1024^3 data set's performance dropped to less than 10 frames per second at the same view range due to increased CPU time per frame (refer to figures 4.5 and 4.6). The main bottleneck in the performance of the 2048^3 data set varied depending on whether or not the camera was moving or static. If the camera was moving, resulting in lots of node and brick uploads, then the bottleneck was the triggering of data transfer to the GPU by the CPU. If the camera was static, then the bottleneck was, again, the ray casting stage of the rendering algorithm.

Not surprisingly, the main memory usage was very similar for both data sets with a 600 meter view range. The 12 active oct-trees required for a view range of

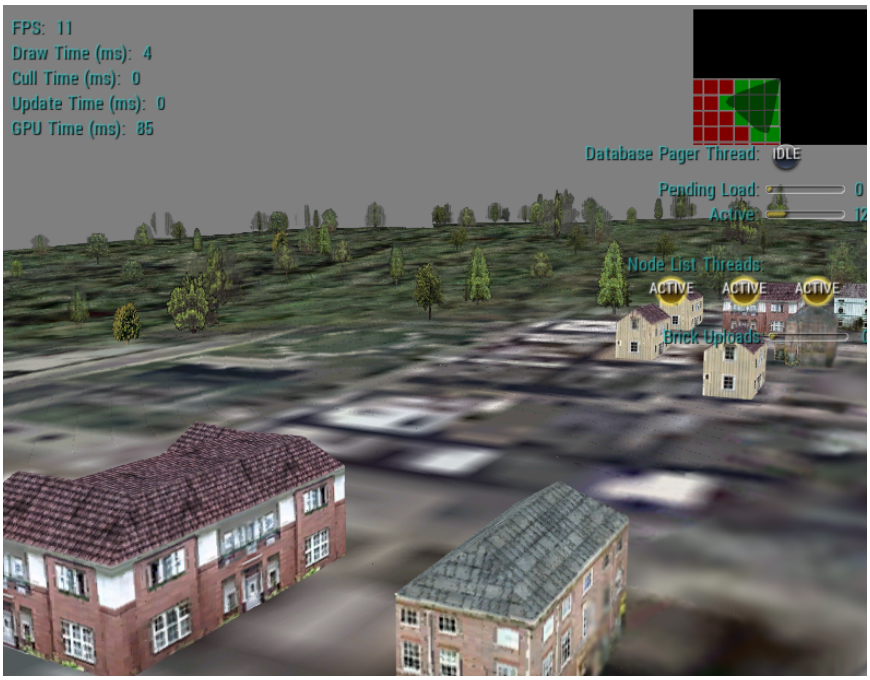

Figure 4.5. 2048^3 Grid of oct-trees with 600 meter view range.

600 meters on the 2048^3 data set resulted in about 4.7GB of RAM usage. Whereas the 1024^3 data set resulted in 47 active oct-trees and about 4.9GB of RAM usage. In addition, the GPU memory was 1.5GB in both cases. This was the expected result because the GPU memory usage is determined primarily by the size of the node pool textures (especially the brick pool texture), whose sizes are specified as a configuration parameter at runtime. This amount of GPU memory usage was actually a little higher than necessary for the 600m view range because the rendering system only required between 15,000 to 20,000 loaded bricks (i.e. active non-constant nodes), but the brick pool texture was configured to hold as many as 32,768 active bricks.

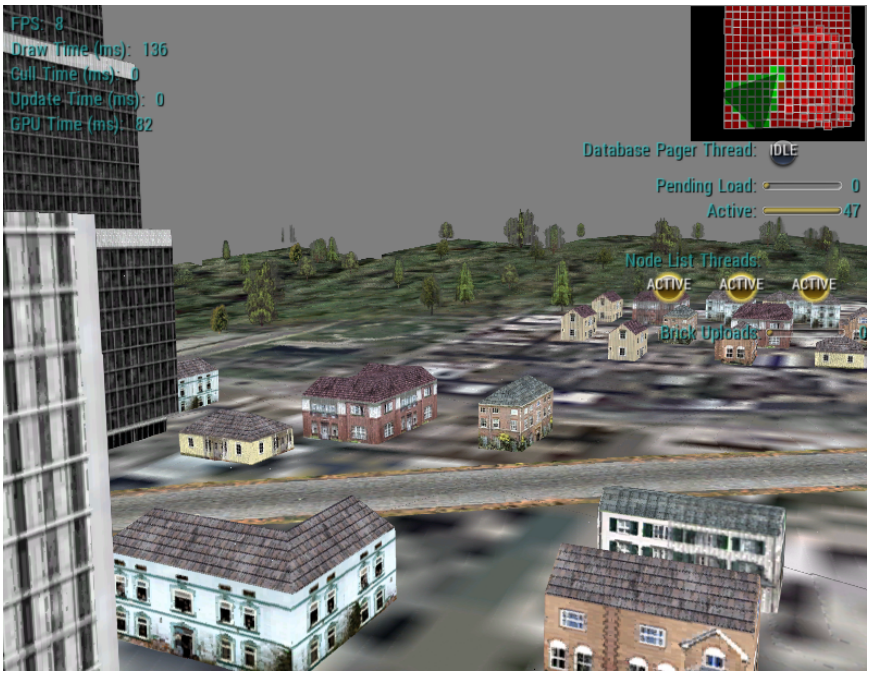

Figure 4.6. 1024^3 Grid of oct-trees with 600 meter view range.

CHAPTER 5

Conclusions & Future Work

The results of this thesis make it clear that the GigaVoxel oct-tree data structure is a very effective mechanism for compaction of terrain voxel data both on disk and in memory. The two test regions each covered a space about 1840m x 1840m x 204m in dimension, requiring a 0.1m voxel data set of about 18432 x 18432 x 2048 voxels. A voxel grid of this size that encoded the color and gradient with 8 bit RGBA and an 8 bit XYZ volume texture, would require about 8TB of data on disk and in memory. The combination of the GigaVoxel oct-tree data structure and DXT compression effectively compressed the terrain voxel data by two orders of magnitude. This data compression coupled with lower GPU memory requirements from ray driven GPU paging make GigaVoxel terrain rendering at least feasible from a memory usage standpoint, but only when the view range is kept relatively low. A view range of 600m required about 12 active oct-trees for the 2048^3 oct-tree grid and 48 active oct-trees for the 1024^3 oct-tree grid and about 5GB of RAM or about 400MB and 100MB per oct-tree respectively. A more fine grained disk paging scheme, one that pages the oct-trees at the granularity of each tree level would allow for longer view ranges because the majority of the oct-trees beyond 600m would require only the lowest detail levels of the oct-tree and thus substantially less memory than the full oct-tree.

The rendering performance results, on the other hand, were less satisfying than the results of the disk and memory consumption. The test results indicate that the rendering performance on small data sets is almost entirely view and data dependent and bound by the pixel shader/ray casting stage. Essentially, for small data sets the performance depends on how far each ray has to traverse through each oct-tree or oct-trees until it becomes fully opacity saturated and how deep into the tree it has to descend to find the correct LOD node. For the full large scale terrain the performance bottleneck is harder to characterize, sometimes the performance is bound by the CPU, specifically the upload of new nodes and bricks, this is especially apparent when flying through the terrain quickly, and sometimes it is bound by the ray casting stage. However, it should be noted that the performance test hardware (NVIDIA GTX 760)

is not, at the time of this writing, the most powerful available consumer grade graphics hardware. By comparison the NVIDIA GTX 980 performs roughly twice as well on standard benchmark tests as the GTX 760 according to videocardbenchmark.net and has roughly double the number of pixel processing cores (1152 to 2048), doubling the number of pixel processing cores would definitely improve the performance of a pixel bound application such as this. So it is possible that consistent interactive to real-time rendering performance on a GigaVoxel terrain data is not outside the realm of possibility on the latest hardware or, at worst, on hardware released in the near future.

That being said, there are several improvements, in addition to the aforementioned disk paging scheme, that could be added to the implementation described by this thesis. One improvement that was not explored would be to limit the amount of time allotted per frame to triggering node and brick uploads to the GPU in order to prevent the CPU from becoming the bottleneck in the rendering performance. This might result in a more consistent frame rate because the GPU ray casting stage would always be the bottleneck, but it would also have the effect of delaying the update of the scene to the required LOD. Implementation of the GPU paging scheme described by Cyril Crassin's PHd thesis [13], which requires no CPU intervention, would more than likely be a more effective improvement. This advanced GPU paging scheme drives the loading of the texture pools entirely on the GPU by using paged locked unified memory, a feature supported by the latest GPU hardware, and thus could eliminate the node usage list download step and the CPU side processing of it as well. In addition, by reducing the CPU workload it would make it possible to support more active oct-trees and thus longer view ranges. Another possible improvement that remains to be explored is an adaptive TIN inspired approach to generating each oct-tree. This would have the potential to improve both the memory usage and the rendering performance. A TIN inspired approach could allow for variable depth branches of the oct-tree, whereby any particular branch of an N-level tree could have it's leaf nodes at level N-X (where N>X) if it is determined that the resolution at level N-X is sufficient to model the data as determined by a specific error metric, much like TINs are generated by eliminating vertices in the DEM mesh. This type of optimization would especially benefit oct-trees that, for example, model flat terrain with very high resolution tree models. The branches of the oct-tree that encompass the tree models would require very small, perhaps 1cm voxels, but the branches of the oct-tree that encompass the terrain could be modeled by much lower resolution voxels.

Furthermore, the generation system could be improved by adaptive fitting of the extents of the oct-trees to the terrain data within the oct-tree grid. The system

used for this thesis required homogeneous 1024^3 or 2048^3 oct-trees for all oct-trees in the terrain grid. A more efficient solution would allow for variable dimensions in all three axis and even rotation of the axis in order to best fit the oct-tree to the input data's extents. This would prevent unnecessary processing on empty regions of the input and result in memory consumption efficiency gains. Furthermore, by creating a tighter fitting oct-tree it would reduce the ray traversal time, which would result in rendering performance gains. In addition, implementing a node based approach to voxelization, as opposed to the chunk based approach, similar to the algorithm described by [3], and possibly a sparse voxelization approach similar to [14], would make the generation system simpler and more memory efficient. These improvements would allow for larger voxel grids to be used for GigaVoxel SVO generation.

The objective of this thesis work was to illuminate the feasibility of utilizing the GigaVoxel rendering algorithm as a basis for a large scale voxel terrain rendering system. The results revealed some success, specifically in memory and disk space compaction, and some challenges for future improvement, specifically in the rendering performance. Despite the less than stellar rendering performance, the lessons learned as a result of this thesis work provide as much if not more value by showing what not to do and where to go next to eventually make realizing a large-scale real-time voxel terrain rendering system possible.

REFERENCES

[1] Bar-Zeev, Avi. "Scenegraphs: Past, Present, and Future." http://www.realityprime.com/blog/2007/06/scenegraphs-past-present-and-future. 2007.

[2] Benson, David, and Joel Davis. "Oct-tree textures." ACM Transactions on Graphics (TOG). Vol. 21. No. 3. ACM, 2002.

[3] Baert, Jeroen, Ares Lagae, and Ph DutrÃČÂľ. "Out-of-Core Construction of Sparse Voxel Octrees." In Computer Graphics Forum, vol. 33, no. 6, pp. 220-227. 2014.

[4] Burns, Andrew. ÃćâĆňÅŞEpic Reveals Stunning Elemental Demo, & Tim Sweeney On Unreal Engine 4.ÃćâĆňÂİ Geforce.com Web URL. June 2012.

[5] Burns, Don, and Robert Osfield. "Open scene graph a: Introduction, b: Examples and applications." (2004): 265.

[6] Christensen, Per H., and Dana Batali. "An Irradiance Atlas for Global Illumination in Complex Production Scenes." Rendering Techniques. 2004.

[7] Celniker, George, Indranil Chakravarty, and Jan Moorman. "Visualization and modelling of geophysical data." Visualization, 1993. Visualization'93, Proceedings., IEEE Conference on. IEEE, 1993.

[8] Cengiz CELEBI, Omer. "Scientific Visualization and 3D Volume Rendering." http://www.byclb.com/TR/Tutorials/volume_rendering. 2015.

[9] Clark, James H. "Hierarchical geometric models for visible surface algorithms." Communications of the ACM 19, no. 10 (1976): 547-554.

[10] Clasen, Malte, and Hans-Christian Hege. "Terrain rendering using spherical clipmaps." Proceedings of the Eighth Joint Eurographics/IEEE VGTC conference on Visualization. Eurographics Association, 2006.

[11] Cline, H.E., Lorensen, W.E., Ludke, S., Crawford, C.R., and Teeter, B.C. “Two Algorithms for the Three-Dimensional Reconstruction of Tomograms.” Medical Physics, 15(3), 320-327. 1988

[12] Crassin, Cyril, et al. "Gigavoxels: Ray-guided streaming for efficient and detailed voxel rendering." Proceedings of the 2009 symposium on Interactive 3D graphics and games. ACM, 2009.

[13] Crassin, Cyril. “GigaVoxels: A Voxel-Based Rendering Pipeline For Efficient Exploration Of Large And Detailed Scenes”. Diss. PhD thesis, UNIVERSITE DE GRENOBLE, 2011.

[14] Crassin, Cyril, et al. "Oct-tree-based sparse voxelization using the gpu hardware rasterizer." OpenGL Insights (2012).

[15] Crow, Franklin C. "The aliasing problem in computer-generated shaded images." Communications of the ACM 20.11 (1977): 799-805.

[16] Drebin, Robert A., Loren Carpenter, and Pat Hanrahan. "Volume rendering." ACM Siggraph Computer Graphics. Vol. 22. No. 4. ACM, 1988.

[17] Decaudin, Philippe, and Fabrice Neyret. "Rendering forest scenes in real-time."EGSR04: 15th Eurographics Symposium on Rendering. 2004.

[18] Dong, Zhao, et al. "Real-time voxelization for complex polygonal models." Computer Graphics and Applications, 2004. PG 2004. Proceedings. 12th Pacific Conference on. IEEE, 2004.

[19] Duchaineau, Mark, et al. "ROAMing terrain: real-time optimally adapting meshes." Proceedings of the 8th Conference on Visualization'97. IEEE Computer Society Press, 1997.

[20] Eisemann, Elmar, and Xavier DÃČÂľcoret. "Fast scene voxelization and applications." Proceedings of the 2006 symposium on Interactive 3D graphics and games. ACM, 2006.

[21] Elvins, T. Todd. "A survey of algorithms for volume visualization." ACM Siggraph Computer Graphics 26.3 (1992): 194-201.

[22] Engel, Klaus, et al. “Real-time volume graphics.” AK Peters, Limited, 2006.

[23] Fang, Shiaofen, and Hongsheng Chen. "Hardware accelerated voxelization." Computers & Graphics 24.3 (2000): 433-442.

[24] Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. “Design patterns: elements of reusable object-oriented software.” Pearson Education, 1994.

[25] Gao, Yu, Baosong Deng, and Lingda Wu. "Efficient view-dependent out-of-core rendering of large-scale and complex scenes." Proceedings of the 2006 ACM international conference on Virtual reality continuum and its applications. ACM, 2006.

[26] Gobbetti, Enrico, Fabio Marton, and JosÃČĈÂĺ Antonio Iglesias GuitiÃČĈÂąn. "A single-pass GPU ray casting framework for interactive out-of-core rendering of massive volumetric datasets." The Visual Computer 24.7-9 (2008): 797-806.

[27] Gibbs, Jonathan, Devorah DeLeon Petty, and Nate Robins. "Painting and rendering textures on unparameterized models." ACM Transactions on Graphics (TOG). Vol. 21. No. 3. ACM, 2002.

[28] Hayward, Kyle. ÃćâĆňÅŞVolume Rendering.ÃćâĆňÂİ Personal blog graphicsrunner.blogspot.com entries tagged as ÃćâĆňÅŞVolume RenderingÃćâĆňÂİ. 2009 through 2010.

[29] Hege, H. C., T. HÃČĈÂűherer, and D. Stalling. "Volume Rendering." (1994).

[30] Herman, G.T., and Liu, H.K. “Optimal Surface Reconstruction from Planar Contours.” Computer Graphics and Image Processing, 1 - 121. 1979.

[31] Hoppe, Hugues. "Smooth view-dependent level-of-detail control and its application to terrain rendering." Visualization’98. Proceedings. IEEE, 1998.

[32] Hua, Wei, et al. "Huge texture mapping for real-time visualization of large-scale terrain." Proceedings of the ACM symposium on Virtual reality software and technology. ACM, 2004.

[33] KÃČĈÂďhler, Ralf, et al. "GPU-assisted raycasting for cosmological adaptive mesh refinement simulations." Eurographics/IEEE VGTC Workshop on Volume Graphics (Boston, Massachusetts, USA, 2006), Machiraju R., MÃČĈÂűller T.,(Eds.), Eurographics Association. 2006.

[34] Kajiya, James T., and Timothy L. Kay. "Rendering fur with three dimensional textures." ACM Siggraph Computer Graphics. Vol. 23. No. 3. ACM, 1989.

[35] Kruger, Jens, and RÃČĈÂijdiger Westermann. "Acceleration techniques for GPU-based volume rendering." Proceedings of the 14th IEEE Visualization 2003 (VIS’03). IEEE Computer Society, 2003.

[36] Lorensen, William E., and Harvey E. Cline. "Marching cubes: A high resolution 3D surface construction algorithm." ACM Siggraph Computer Graphics. Vol. 21. No. 4. ACM, 1987.

[37] Levoy, Marc. "Display of surfaces from volume data." Computer Graphics and Applications, IEEE 8.3 (1988): 29-37.

[38] Levoy, Marc. "Efficient ray tracing of volume data." ACM Transactions on Graphics (TOG) 9.3 (1990): 245-261.

[39] Losasso, Frank, and Hugues Hoppe. "Geometry clipmaps: terrain rendering using nested regular grids." ACM Transactions on Graphics (TOG) 23.3 (2004): 769-776.

[40] Lefebvre, Sylvain, Samuel Hornus, and Fabrice Neyret. "GPU Gems 2. chapter 37: oct-tree Textures on the GPU." (2005).

[41] Laine, Samuli, and Tero Karras. "Efficient sparse voxel oct-trees." Visualization and Computer Graphics, IEEE Transactions on 17.8 (2011): 1048-1059.

[42] Lacroute, Philippe, and Marc Levoy. "Fast volume rendering using a shear-warp factorization of the viewing transformation." Proceedings of the 21st annual conference on Computer graphics and interactive techniques. ACM, 1994.

[43] Li, Wei, Klaus Mueller, and Arie Kaufman. "Empty space skipping and occlusion clipping for texture-based volume rendering." Proceedings of the 14th IEEE Visualization 2003 (VIS'03). IEEE Computer Society, 2003.

[44] Lindstrom, Peter, and Valerio Pascucci. "Visualization of large terrains made easy." Visualization, 2001. VIS'01. Proceedings. IEEE, 2001.

[45] Luebke, David P., ed. "Level of detail for 3D graphics." Morgan Kaufmann Pub, 2003.

[46] Meyer, Alexandre, and Fabrice Neyret. "Multiscale shaders for the efficient realistic rendering of pine-trees." Graphics Interface. 2000.

[47] MÃČÂÿller-Nielsen, Peter. "Sparse voxel oct-tree ray tracing on the gpu." Diss. Aarhus Universitet, Datalogisk Institut, 2009.

[48] Nvidia, C. U. D. A. "Programming guide." (2008).

[49] Ohno, Nobuaki, and Akira Kageyama. "Scientific visualization of geophysical simulation data by the CAVE VR system with volume rendering." Physics of the Earth and Planetary Interiors 163.1 (2007): 305-311.

[50] Pantaleoni, Jacopo. "VoxelPipe: a programmable pipeline for 3D voxelization." Proceedings of the ACM SIGGRAPH Symposium on High Performance Graphics. ACM, 2011.

[51] Pawasauskas, John. ÃćâĆňÅŞVolume rendering overview.ÃćâĆňÂİ Personal Website. 1997.

[52] Peachey, Darwyn R. "Solid texturing of complex surfaces." ACM SIGGRAPH Computer Graphics. Vol. 19. No. 3. ACM, 1985.

[53] Perlin, Ken. "An image synthesizer." SIGGRAPH Comput. Graph. 19.3 (1985): 287-296.

[54] Rabinovich, Boris, and Craig Gotsman. "Visualization of large terrains in resource-limited computing environments." Visualization'97., Proceedings. IEEE, 1997.

[55] RoupÃČÂľ, Mattias, and Mikael Johansson. "3D-city modeling: a semi-automatic framework for integrating different terrain models." Advances in Visual Computing. Springer Berlin Heidelberg, 2011. 725-734.

[56] RÃČÂűttger, Stefan, Martin Kraus, and Thomas Ertl. "Hardware-accelerated volume and isosurface rendering based on cell-projection." Proceedings of the conference on Visualization'00. IEEE Computer Society Press, 2000.

[57] Royan, JÃČÂľrÃČÂťme, et al. "Network-based visualization of 3D landscapes and city models." Computer Graphics and Applications, IEEE 27.6 (2007): 70-79.

[58] Salemann, Leo, and Roland Cutaran. ÃćâĆňÅŞPolygon-free modeling & simulation.ÃćâĆňÂİ Simulation Interoperability and Standards Organization. November 2010.

[59] Scharsach, Henning. "Advanced GPU raycasting." Proceedings of CESCG 5 (2005): 67-76.

[60] Salemann, Leo, Scott Gebhardt, and Eliezer Payzer. "Polygons, point-clouds, and voxels, a comparison of high-fidelity terrain representations." Simulation Interoperability and Standards Organization. November 2010.

[61] Silva, Pedro MÃČÂąrio, Marcos Machado, and Marcelo Gattass. "3D seismic volume rendering." 8th International Congress of the Brazilian Geophysical Society. 2003.

[62] Schwarz, Michael, and Hans-Peter Seidel. "Fast parallel surface and solid voxelization on GPUs." ACM Transactions on Graphics (TOG). Vol. 29. No. 6. ACM, 2010.

[63] Tanner, Christopher C., Christopher J. Migdal, and Michael T. Jones. "The clipmap: a virtual mipmap." Proceedings of the 25th annual conference on Computer graphics and interactive techniques. ACM, 1998.

[64] Van Gelder, Allen, and Kwansik Kim. "Direct volume rendering with shading via three-dimensional textures." Volume Visualization, 1996. Proceedings., 1996 Symposium on. IEEE, 1996.

[65] Valentino, Daniel J., J. C. Mazziotta, and H. K. Huang. "Volume rendering of multimodal images: application to MRI and PET imaging of the human brain." Medical Imaging, IEEE Transactions on 10.4 (1991): 554-562.

[66] Westover, Lee Alan. Splatting: a parallel, feed-forward volume rendering algorithm. Diss. University of North Carolina at Chapel Hill, 1991.

[67] Wikipedia. CUDA. http://en.wikipedia.org/wiki/CUDA. February 2015.

[68] Wikipedia. Scene Graph. http://en.wikipedia.org/wiki/Scene_graph. January 2015.

[69] Wikipedia. Lidar. https://en.wikipedia.org/wiki/Lidar. July 2015.

[70] Wikipedia. Volume Rendering. http://en.wikipedia.org/wiki/Volume_rendering. September 2013.

[71] Wikipedia. Triangulated irregular network. http://en.wikipedia.org/wiki/Triangulated_irregula May 2014.

[72] Wrenninge, Magnus, et al. "Volumetric methods in visual effects." ACM SIGGRAPH. 2010.

[73] Wimmer, Michael, Peter Wonka, and FranÃČÂğois Sillion. "Point-based impostors for real-time visualization." Rendering Techniques 2001. Springer Vienna, 2001. 163-176.

[74] Zhang, Long, et al. "Conservative voxelization." The Visual Computer 23.9-11 (2007): 783-792.

[75] Ziegler, Gernot, Art Tevs, Christian Theobalt, and Hans-Peter Seidel. "GPU point list generation through histogram pyramids." 2006.