Real-Time Rendering of Procedurally Generated Volumetric Models

Jakob Udsholt

Kongens Lyngby 2013
IMM-M.Sc-2013-56

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Building 303 B, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253031, Fax +45 45881399
compute@compute.dtu.dk
www.compute.dtu.dk IMM-M.Sc-2013-56

Summary

This thesis presents a volume based approach to real-time rendering of highly detailed procedurally generated scenes. This approach is based on existing methods for sparse voxel octrees and cache based data management, which aims to solve the problem of maintaining large volume data sets in memory.

Two extensions, designed to provide an efficient framework for procedural content generation in a parallel environment, are suggested: The first is a three step method for subdivision of sparse voxel octree nodes, when the contents of the node is unknown prior to evaluation. The second is a simplification to cache invalidation of unused octree nodes.

A hardware accelerated volume rendering implementation is presented. It uses NVIDIAs CUDA platform to perform volume ray casting in real-time on consumer graphics hardware. The implementation is evaluated based on performance, memory requirements and image quality. Performance bottlenecks related to procedural content generation and scalability are identified, and possible solutions are presented. Finally, potential areas for further work are outlined.

Acknowledgements

I would like to express my appreciation to everyone who provided me with assistance during the course of this project. A special thanks to my supervisors Jakob Andreas Bærentzen and Jeppe Revall Frisvad for their guidance and feedback throughout the process of writing this thesis. I would also like to thank Christian for his valuable input. Finally I would like to thank Darlene for her continued support whenever the voxels were misbehaving.

Contents

Chapter 1

Introduction

This thesis presents a method for producing images of large three dimensional environments, such as landscapes, in a way that makes it suitable in interactive simulations. While the title *"Real-Time Rendering of Procedurally Generated Volumetric Models"* is quite a mouthful, it highlights the four key elements that are relevant to this thesis.

Rendering refers to the process of generating an image on a computer, of some object represented by a virtual model. *Real-time* indicates that several images are rendered continuously to the screen, with a speed that provides the user with an immediate visual feedback as he/she interacts with the program. One example of this is computer games. *Procedural generation* is the process of creating content mathematically and automatically, without the explicit need of an artist. Figure 1.1 shows the Mandlebrot set which is a popular example of procedurally generated shape. *Volumetric models* is related to the representation of the virtual model used in the rendering process. Medical data acquired from CT or MRI scanners are both examples of volumetric models.

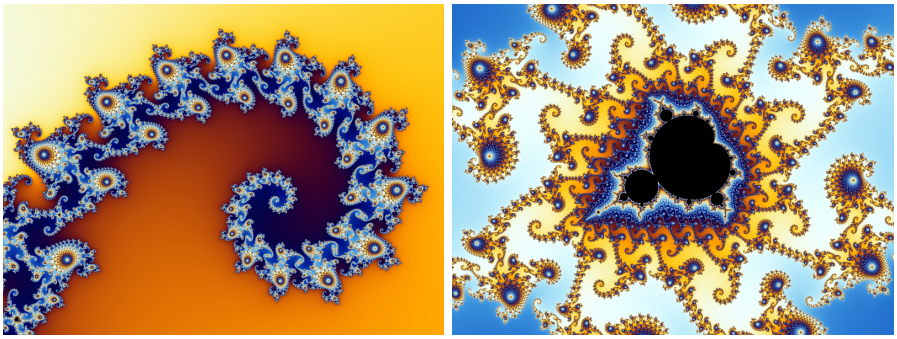


Figure 1.1: The Mandlebrot set is a popular example of a procedurally generated fractal shape.
Source: http://en.wikipedia.org/wiki/Mandelbrot_set

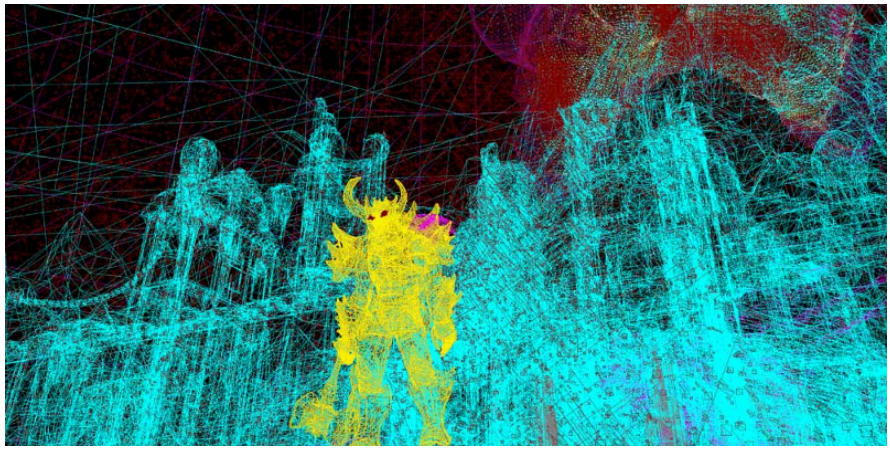
1.1 Background

Creating visual appealing and realistic images often requires rendering highly complicated scenes, such as the one seen in figure 1.2. The complexity grows with the number of objects in the scene and how detailed they are. In interactive simulations, such as computer games, the speed at which images can be produced and displayed is a major concern. Low frame rates, or fluctuations in the frame rate, means that the illusion of interactivity breaks down. Consequently, maintaining a high and constant frame rate is important for interactive simulations.

In traditional rendering methods, the surface of an object in a scene is represented by a triangle mesh. In order to render the object to the screen, the mesh is converted to pixels through a process known as triangle rasterisation. These methods tend to be inefficient for complex scenes because the computational cost of rendering the final image, is proportional to the number of triangles on the screen [Cra11]. Volume graphics takes a different approach to rendering. Originally motivated by the need to visualize data sets in scientific visualization and medical imaging, it has several benefits when dealing with large data sets [EHK+06]. One clear advantage is that data can be accessed in a structured way, that is decoupled from the scene complexity. Recent advancements in consumer graphics hardware have made it possible to apply volume rendering techniques at interactive frame rates. A downside to volume rendering is that it typically requires a lot of memory to store the complete data sets [Cra11].

Problems with scene complexity manifests it self in other ways than just the computational cost of rendering. Growing demands for visual realism and de-

(a) Rendered image from Epics Unreal Engine 4 Elemental demo.



(b) Wireframe model showing the same scene.

Figure 1.2: Screenshots from Epics Unreal Engine 4 Elemental Demo. Figure (a) shows the final image while (b) hints to underlaying complexity of the scene [Images are property of Epic Games, Inc.].

tails in computer games requires an increased amount of graphical content to fill the virtual worlds. Graphical content in computer games includes elements like models for characters and objects, surface textures and terrains. However, manually creating every piece of graphical content can potentially have a negative impact on production time and cost. This impact can be remedied through the use of procedural content generation. And while not all aspects of game content lend themselves naturally to a procedural approach, because they might be hard

to describe by a mathematical process, elements like terrains can benefit from being procedural generated as will be shown in chapter 2.

1.2 Objectives

The goal of this thesis is to investigate how large, procedurally generated, volume data sets can be compactly stored in memory, and how they can be rendered efficiently. Consumer graphics hardware will be used as a platform as the solution should be usable in interactive simulations, such as computer games. This can be broken into two parts:

- Investigate how large volume data sets can be stored compactly and rendered efficiently, using current hardware accelerated methods.
- Building upon these methods, investigate how a framework for on-the-fly procedural content generation could be implemented.

1.3 Overview

This thesis is divided into chapters that aims to familiarize the reader with volume rendering, procedural content generation and how the two can be combined into a hardware accelerated solution that allows large volume data sets to be generated and visualized in real-time.

First, a procedural approach to content generation for volumetric terrains is presented. Next follows an introduction to the physical foundation for volume rendering and iterative rendering methods, such as ray casting. Then a group of data structures, used to compactly store large volume data sets, called sparse voxel octrees are presented. Next is an introduction to parallel computing, on consumer graphics hardware using NVIDIAs CUDA platform. This leads to an analysis and implementation of procedural content generation combined with ray casting using sparse voxel octrees. Finally the results of the implementation are evaluated and discussed, based on the original goals of the thesis. Evaluation criteria involves rendering performance, memory requirements and image quality.

The reader is assumed to be familiar with data structures, such as lists and trees, and basic concepts related to real-time rendering including: mipmapping, shaders, transformation matrices and frame rate.

Chapter 2

Procedural Generation

This chapter serves as an introduction to the history of procedural content generation in games. It will describe one possible approach to generating three dimensional terrains suitable for use in volume graphics, by evaluating a density function. As the term procedural generation can refer to anything from music to animations, this chapter should by no means be considered as an exhaustive look at procedural generation in general.

2.1 Procedural Content Generation in Games

Procedural content generation is used to describe the generation of content algorithmically, rather than manually. It was used extensively in the video game industry at a time where games where severely limited by storage constraints, as there was simply not enough room to store large amounts of levels and artwork [Rud09]. Instead content could be generated on-the-fly using some deterministic algorithm. Determinism is an important factor because the algorithm should always produce the same content, given a particular input, unless the objective is to provide the player with a completely random experience.

The game The Elder Scrolls II: Daggerfall, from 1996, used a mostly procedural approach to generate a massive game world of roughly 200.000 km^2 in size

[Rud09]. Later games in the series moved away from this approach, and used procedural methods to generate and store landscapes and vegetation during game development, which were then loaded at run-time. This allowed artists to manually alter details and make the world more interesting to the player, see figure 2.1.

(a) (b)

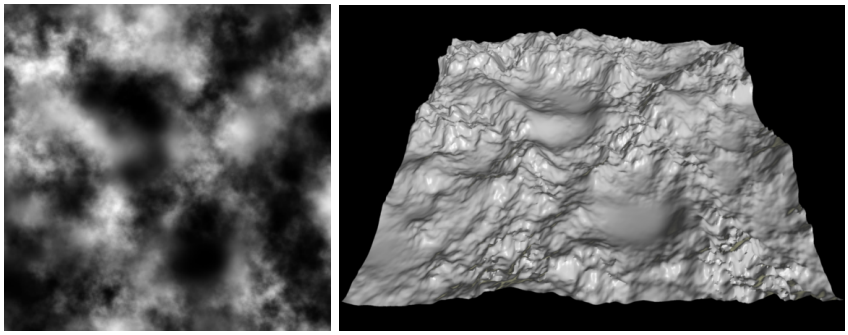
Figure 2.1: Examples of procedurally generated content in video games. (a) an enviroment screenshot from The Elder Scrolls II: Daggerfall where trees and other details are procedurally placed. (b) a screenshot from Oblivion, a later game from the same series, where the content relies more on manual generation [Both games are the property of ZeniMax].

Entirely procedural worlds are still encountered in the games industry. Recent independent games with relatively small production budgets, such as Minecraft and Terraia, uses procedural approach to generate a completely unique world that gives the games a high level of replay value.

2.2 Procedural Terrain

Traditionally, terrain is represented using a two dimensional height field that encodes elevation data. This can be an image, where the intensity of a single channel is interpreted as the height of the terrain at a given point. An example of a height field, and its corresponding rendered landscape can be seen in figure 2.2. Height fields can either be created manually by "painting" the elevation data or they can be procedurally generated.

The notion of a height field can be extended to three dimensions by using a 3D scalar field, typically represented as a volume texture. In this context the intensity is no longer interpreted as a height value, but rather as a measurement of how dense the medium is at a specific location in the volume. One clear



(a) (b)

Figure 2.2: Example of a two dimensional height field. (a) the elevation data encoded as intensity and (b) the height field rendered as a displacement map.
Source: http://en.wikipedia.org/wiki/Heightmap

advantage of working in three dimensions is the built-in ability to express cliff overhangs and caves, as this information cannot be encoded in a height map.

2.3 Density Function

Procedurally generating three dimensional terrains can be done by evaluating a density function [Ngu07]. Such a function $f(\mathbf{x})$ simply takes volume coordinate $\mathbf{x} = (x, y, z)$ and returns the density corresponding a location in the volume as a scalar value.

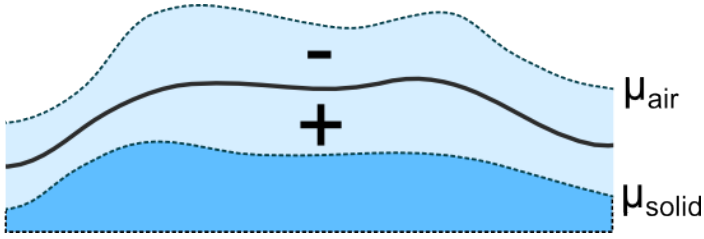


Figure 2.3: Density values between the thresholds $\mu_{air} < 0 < \mu_{solid}$ can be interpreted as a surface.

Applying $f(\mathbf{x})$ to the whole volume it results in a 3D scalar field. As will be described in section 3.6.1, it is possible to interpret a set of points sharing the same scalar value as a surface (or a level set). By picking some threshold values

$\mu_{air} < 0$ and $\mu_{solid} > 0$, representing the density of completely empty and solid space respectively, a surface around $d(\mathbf{x}) = 0$ can be described. This is illustrated in figure 2.3.

2.4 Noise Based Density

A good density function is necessary to create believable and aesthetically pleasing terrains. This will be a brief examination of possible ways a density function can produce interesting results, based on [Ngu07]. Significantly more advanced ways of generating procedural terrains are possible, but this examination will be limited to a function taking a single volume coordinate, and evaluating the density without requiring any other contextual information. This simplification will make the function more suitable when a large amount of densities must be evaluated in parallel.

Noise functions are a common method for introducing seemingly random details in procedural content generation. Noise can be generated in a number of ways, but two key requirements (from [MPP+94]) of a noise function, intended for procedural content generation, are: For a given input the result is always the same, i.e. the result is repeatable and deterministic. The values returned by the noise function are in a known range, namely $[-1..1]$.

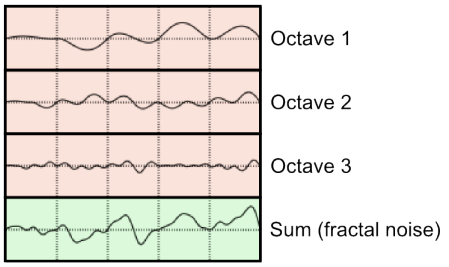
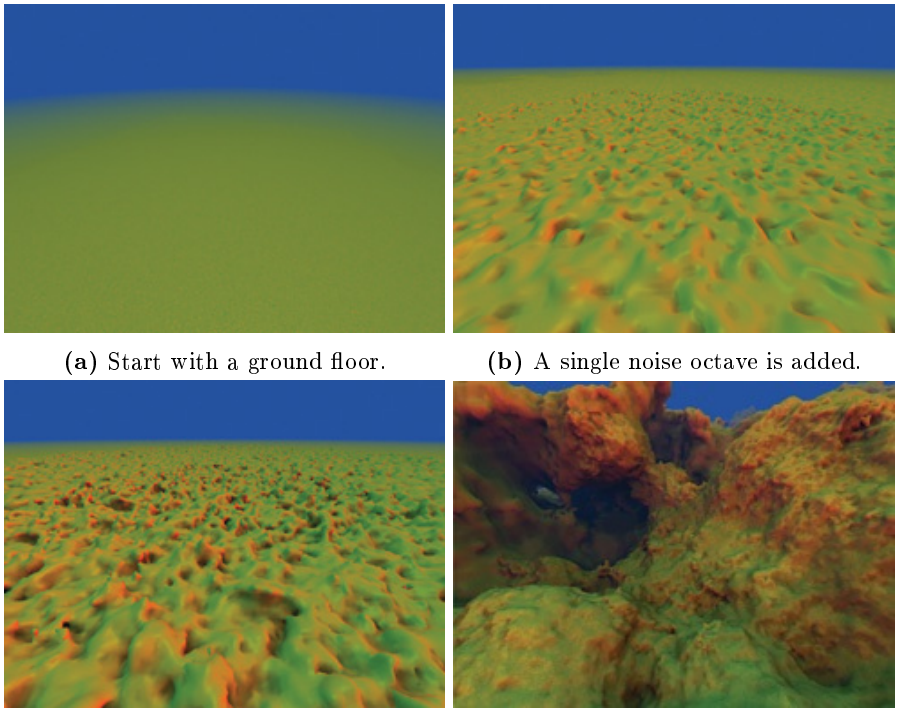


Figure 2.4: A one dimensional example of fractal noise. The noise function is sampled over a number of iterations at increasing frequency (octaves) and summed together.

The frequency of a noise function refers to the number of cycles per unit length, and is a measure of how fast the function changes, see figure 2.4. When a noise function is sampled at different frequencies over a number of iterations and summed together, the result is called fractal noise [MPP+94]. Starting from a specific sampling frequency, the frequency is double with each iteration. The number of iterations is called octaves, which is a term adopted from music,

where it means to double frequency (or pitch) of a tone. See figure 2.4.

Figure 2.5 shows the application of noise within a density function. By sampling the noise function at the volume position with different octave parameters, interesting effects can be obtained. Lower frequencies of noise can be used to simulate mountains, while high frequencies creates surface details.



(a) Start with a ground floor.

(b) A single noise octave is added.

(c) Three noise octaves is added.

(d) Lower frequencies of noise is added.

Figure 2.5: Examples of noise based density in the NVIDIA Cascades Demo from 2007.
Source: [Ngu07]

Chapter 3

Volume Rendering

This chapter serves as an introduction to volume rendering. Specifically it will present a physical model for volume rendering that makes iterative rendering methods, such as volume ray casting possible.

3.1 Physical Model

In volume rendering, light is assumed to propagate along straight lines, unless it interacts with a participating medium. Participating medium is a general term used to describe any material that participates in the propagation of light together with air.

When interaction between light and participating medium occurs, the radiative energy along the light ray is changed. Typically this energy is referred to as radiance $I(\mathbf{x}, \omega)$, which describes the radiation at a point $\mathbf{x}$ on a surface given a light direction ω. In [Cra11] radiance is defined by:

$$I(\mathbf{x}, \omega) = \frac{\mathrm{d}\, Q}{\mathrm{d}\, A \cos\theta \,\mathrm{d}\, \Omega \,\mathrm{d}\, t} \tag{3.1}$$

where Q is the radiant energy (in Joules), and A is some surface area (in m^2). θ is the angle between the light direction ω and the normal vector $\mathbf{n}$ of the surface A, Ω the solid angle (in steradians) and t is time seconds. See figure 3.1.

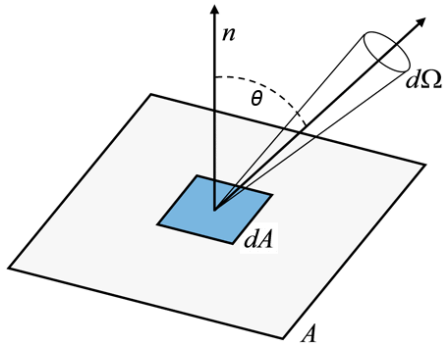


Figure 3.1: Radiance is radiant energy per projected area per solid angle per unit time.

Normally the three following categories of interaction between light and participating medium are considered:

Emission occurs when the medium actively emits light, this increases the radiative energy.

Absorption occurs when the medium absorbs radiative energy converting it to heat, this causes a reduction in radiative energy.

Scattering occurs when the medium scatter light and causes it to change direction. This effect can be divided into two cases: in-scattering and out-scattering. In-scattering happens when light arriving from elsewhere is scattered in the direction of the light propagation, thereby increasing the radiative energy. Out-scattering occurs when light already propagating along the direction is scattered into another direction. See figure 3.2.

Absorption reduces the radiance, while emission increases the radiance. Scattering can both increase or decrease the radiance, depending on whether in- or out-scattering occurs. In total the radiative transfer equation is defined by:

$$\omega \cdot \nabla_{\mathbf{x}} I(\mathbf{x}, \omega) = -\chi(\mathbf{x}, \omega) I(\mathbf{x}, \omega) + \eta(\mathbf{x}, \omega) \tag{3.2}$$

where $\omega \cdot \nabla_{\mathbf{x}} I(\mathbf{x}, \omega)$ describes the dot product between the light direction ω and the radiance gradient, with respect to position $\mathbf{x}$. This dot product is a scalar

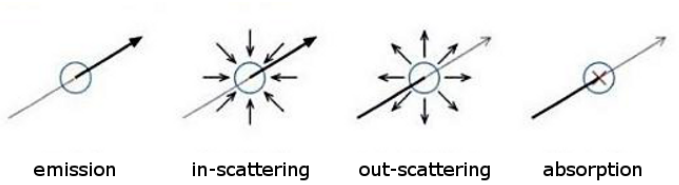


Figure 3.2: Interactions between light and participating medium.
Source: [EHK+06]

value expressing the change in radiant energy at point $\mathbf{x}$ in direction ω. The term $\chi(\mathbf{x}, \omega)$ is the total absorption, and $\eta(\mathbf{x}, \omega)$ is the total emission. Both absorption and emission depends on the participating medium, and can therefore not be considered constant throughout the volume. The interpretation of equation 3.2 is that the change in radiance in a specific direction, can be expressed as sum of the total radiance absorbed and emitted.

The total absorption χ can be broken into two parts: the true absorption κ, and the scattering coefficient σ which represents energy loss due to out-scattering. Likewise the total emission η can be expressed as the addition of true emission q, and the energy gain due to in-scattering j.

$$\begin{aligned} \chi(\mathbf{x}, \omega) &= \kappa(\mathbf{x}, \omega) + \sigma(\mathbf{x}, \omega) \\ \eta(\mathbf{x}, \omega) &= q(\mathbf{x}, \omega) + j(\mathbf{x}, \omega) \end{aligned} \tag{3.3}$$

While κ, σ and q are optical material properties, the in-scattering coefficient j is more complicated to deal with. It represents all incoming energy contributions, from all directions over the sphere:

$$j(\mathbf{x}, \omega) = \frac{1}{4\pi} \int_{sphere} \sigma(\mathbf{x}, \omega') p(\mathbf{x}, \omega', \omega) I(\mathbf{x}, \omega') d\omega' \tag{3.4}$$

Radiance incident from $I(\mathbf{x}, \omega')$ are weighed by both the scattering coefficient σ, and a phase function $p(\mathbf{x}, \omega', \omega)$ that describes the quantity that light will be scattered from the incoming direction w' into the direction w.

Including in-scattering $j(\mathbf{x}, \omega)$ complicates rendering because it requires evaluating the full radiative transfering equation for all directions incident on the sphere. So the typical approach is to chose an optical model that only considers emission and absorption [EHK+06]. If only emission and absorption is

considered the equation 3.2 can be rewritten as:

$$\omega \cdot \nabla_{\mathbf{x}} I(\mathbf{x}, \omega) = -\kappa(\mathbf{x}, \omega) I(\mathbf{x}, \omega) + q(\mathbf{x}, \omega) \tag{3.5}$$

In [EHK+06] equation 3.5 is referred to as the volume rendering equation. Specifically it is the volume rendering equation in its differential form, because it describes the differential change in radiance at a point in a given direction.

If only a single light ray is considered it can be parameterized as:

$$\frac{\mathrm{d}\, I(s)}{\mathrm{d}\, s} = -\kappa(s) I(s) + q(s) \tag{3.6}$$

where the parameter s is the length along some line expressed by $\mathbf{x} = \mathbf{p} + \mathbf{s}\omega$ and $\mathbf{p}$ is some arbitrary point along the ray.

Equation 3.6 can be solved by integrating for radiance along the ray from the starting point $s = s_0$, at the back of the volume, to the endpoint $s = D$, at the front of the volume. This results in the volume rendering integral:

$$I(D) = I_0 e^{-\int_{s_0}^{D} \kappa(s)\, \mathrm{d}\, t} + \int_{s_0}^{D} q(s) e^{-\int_{s_0}^{D} \kappa(t)\, \mathrm{d}\, t}\, \mathrm{d}\, s \tag{3.7}$$

where the term I_0 is the light entering the volume from the background, and $I(D)$ is the light arriving at the front of the volume. This means that the light, arriving at the front of the volume, is the light from the background attenuated by the medium in the volume, and any light contributed within the volume, attenuated by the remaining medium along the ray.

By introducing a transparency function that describes the absorption of light along the line segment from $s = s_0$ to $s = s_1$:

$$T(s_0, s_1) = e^{-\int_{s_0}^{s_1} \kappa(t)\, \mathrm{d}\, t} \tag{3.8}$$

equation 3.7 can be rewritten as:

$$I(D) = I_0 T(s_0, D) + \int_{s_0}^{D} q(s) T(s, D)\, \mathrm{d}\, s \tag{3.9}$$

which will serve as the foundation for a discretization in the next section.

3.2 Discretization

In practice, it is rarely possible to solve the volume rendering, equation 3.9, analytically. Instead a discretization is necessary to numerically approximate the solution.

The integration in equation 3.9 can be split into n intervals described by $s_0 < s_1 < ... < s_n = D$, this is shown in figure 3.3. These intervals does not have to be of equal size.

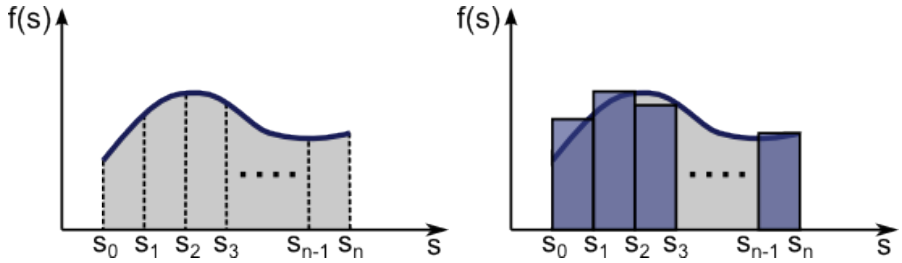


Figure 3.3: Integration along a ray split into several intervals $[s_0..s_n]$.
Source: [EHK+06]

If only the light contribution of the ith interval $[s_{i-1}, s_i]$ is considered, and introduce the notation T_i, for the transparency in the interval, and c_i for the radiance contribution:

$$T_i = T(s_{i-1}, s_i), \qquad c_i = \int_{s_{i-1}}^{s_i} q(s)T(s, s_i)\,\mathrm{d}\,s \tag{3.10}$$

equation 3.9 can be discretized as:

$$I(D) = \sum_{i=0}^{n} c_i \prod_{j=i+1}^{n} T_j, \qquad \text{where } c_0 = I(s_0) \tag{3.11}$$

which states that the radiance leaving the volume at $s = D$, is the sum of the contribution from all intervals, attenuated by the product of the transparency

of the remaining intervals. It is common to replace transparency of the interval T_i by opacity defined as $\alpha_i = 1 - T_i$.

3.3 Compositing

The discrete version of the volume rendering integral, in equation 3.11, can be evaluated iteratively using a method known as compositing [EHK+06]. Two basic compositing schemes are common: back-to-front and front-to-back, here only the latter will be presented.

Radiance is now substituted with a color representation C, typically three components RGB (red, green, blue). For a detailed explanation of the relationship between radiance and color see [MHH08]. Front-to-back compositing requires that rays are traversed from an origin point into the volume, this leads to the following equations:

$$\hat{C}_i = \hat{C}_{i+1} + \hat{T}_{i+1} C_i, \qquad \hat{T}_i = \hat{T}_{i+1}(1 - \alpha_i) \tag{3.12}$$

with $\hat{C}_n = C_n$ and $\hat{T}_n = 1 - \alpha_n$. This is an iterative definition where $\hat{C}_i$ and $\hat{T}_i$ are the results of the current iteration step, and $\hat{C}_{i+1}$ and $\hat{T}_{i+1}$ are the accumulated results of the previous steps. In [EHK+06] it is shown that this can be rewritten into the the following compositing scheme:

$$\begin{aligned} C_{dst} &\leftarrow C_{dst} + (1 - \alpha_{dst}) C_{src}, \\ \alpha_{dst} &\leftarrow \alpha_{dst} + (1 - \alpha_{dst}) \alpha_{src} \end{aligned} \tag{3.13}$$

where the subscript $_{src}$ denotes optical parameters from the volume data set, and $_{dst}$ denotes the resulting accumulated quantities.

The important observation from equation 3.13 is that it is possible to traverse a ray from the eye into the volume, accumulating and updating color C_{dst} and opacity α_{dst} by sampling the source parameters from the volume, and terminating as soon as the α_{dst} becomes fully opaque, $\alpha_{dst} \geq 1$.

3.4 Volume Representation

Volume rendering assumes that data is represented as a continuous three dimensional scalar field: $\mathbb{R}^3 \rightarrow \mathbb{R}$. However, in practice volume data is typically represented as a three dimensional regular spaced grid [EHK+06]. The individual elements in the grid are called voxels, which is short for volumetric elements. Each voxel is associated with a number of scalar values such as density, colors or flow. The exact data associated with a voxel will depend on the application. The rigid structure, provided by the regular grid, ensures that individual voxels can easily be accessed by their three dimensional coordinate within the grid.

This definition of a voxel gives rise to two possible interpretations. It can be viewed as either: a small cube occupying some small volumetric region of space, or as a single point where an interpolation scheme is used to fill the space between the points. In this thesis the latter interpretation is used as it makes it easier to sample the grid, when the sampling location does not match an exact voxel coordinate.

3.5 Rendering Methods

Volume rendering methods can generally be divided into two top level categories: direct- and indirect-volume rendering.

With indirect volume rendering, or surface rendering, the volume data is transformed into another domain before visualization. This will typically be polygons representing a level surface (or iso-surface) which is extracted using a method such as marching cubes [Ngu07]. Direct volume rendering methods renders the volume data directly without any transformation to an intermediate surface representation. The three main classes of direct volume rendering are: ray casting, texture slicing and splatting. Texture slicing and splatting are older methods that tends to be either inflexible, or produce poor approximations of the volume rendering integral [Cra11]. While the basic concepts in ray casting is not new, it is more flexible, and the recent evolution of graphics hardware has made it feasible to perform volume ray casting in real-time. For this reason volume ray casting will be the focus of this thesis. For an in depth description of texture slicing and splatting see [EHK+06].

3.5.1 Volume Ray Casting

Volume ray casting, also called volume ray tracing, is a popular method for volume rendering. For each pixel in the final image, a single ray originating from the eye is traversed into the volume, see figure 3.4. The volume data is sampled at discreet steps along the ray, accumulating the final color of the pixel. The traversal order is front-to-back which means that the compositing scheme, from equation 3.13, can be applied.

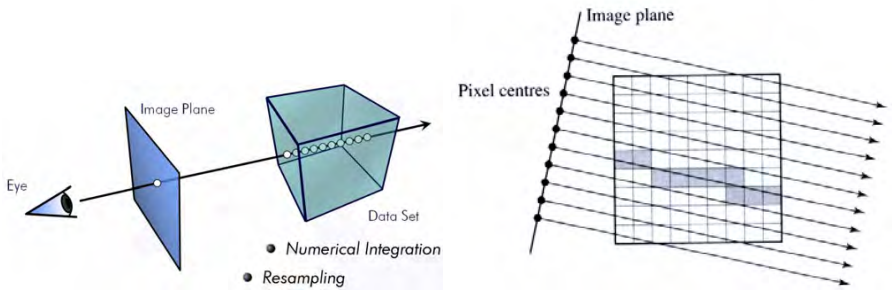


Figure 3.4: Volume rendering using ray casting. For each pixel on the image plane a ray originating at the eye is traced through the volume.
Source: [EHK+06]

The basic outline of a volume ray casting algorithm is outlined in algorithm 1. The pseudocode can be broken into 2 major components, ray setup and the traversal loop:

Ray setup Rays needs to be setup according to the viewing parameters. The rays originates at the eye and their direction is determined by the position of the pixel on the image plane, a plane somewhere in front of the eye upon which the volume is projected. The image plane is also sometimes called the near plane.

For each pixel in the final image a ray r, is created from the eye point p_{eye} to the point p_{near} corresponding to the pixel location on the near plane. The details will depend on the implementation, but this step typically involves transforming the ray into world space using the inverse view matrix.

The ray r is then intersected with the volumes axis aligned bounding box to determine if the ray hits, and where along the ray it enters the volume t_{near}, and where it exits the volume t_{far}.

Traversal Loop Each ray is traversed iteratively, evaluating the volume rendering integral at discreet positions. This consists of the following sub-

components:

Data Access The volume data is accessed at the current ray position p_{sample}, which is a point t_{sample} distance along the ray r. As the sample positions typically does not match an exact grid point in the original volume data, the function `sample` yields the color C_{src} and the opacity α_{src} using some form of interpolation, on the original volume data.

Compositing The color and opacity is accumulated using front-to-back scheme as described in equation 3.13.

Advance Ray Position The current sampling distance along the ray t_{sample} is advanced by some step value t_{step}.

Ray Termination The traversal loop is ended when the distance along the ray t_{sample} is outside the volume. This happens when $t_{sample} > t_{far}$. It is also possible to perform early ray termination when $\alpha_{dst} \geq 1$ as described in section 3.3.

Algorithm 1: Pseudocode for volume ray casting

```
1  foreach p_near ∈ image plane do
2      r ← ray from p_eye to p_near
3      ⟨hit, t_near, t_far⟩ ← intersectVolume(r)
4      if hit then
5          t_sample ← t_near
6          while t_sample ≤ t_far do
7              p_sample ← r_origin + r_direction * t_sample
8              ⟨C_src, α_src⟩ ← sample(p_sample)
9              C_dst ← C_dst + (1 − α_dst)C_src
10             α_dst ← α_dst + (1 − α_dst)α_src
11             t_sample ← t_sample + t_step
12         end
13     end
14     write the final color C_dst into the image at the pixel location
15 end
```

3.6 Local Illumination Model

In section 3.1 the light transfer equation 3.2 was simplified by ignoring in-scattering. One implication of this, is that all external light contributions are ignored, with the exception of the background light I_0. External lights add a

great deal of realism to the final rendered image, and as such completely ignoring the contributions from light sources, results in unappealing images.

In surface rendering, such as triangle rasterisation, the interaction of light from light sources at material boundaries can be described using the bidirectional reflectance distribution function (BRDF). In volume rendering a BRDF can be applied by assuming, that light is reflected at surfaces inside the volume data [EHK+06]. This means a computationally inexpensive method for doing local illumination can be implemented, using a local illumination model.

3.6.1 Gradient Estimation

Traditionally, local illumination models depend on the notion of a normal vector, a vector perpendicular to the surface (of unit length), that describes orientation of the surface. In volume rendering it is assumed that light is reflected at iso-surfaces inside the volume data [EHK+06]. A iso-surface can be defined as a set of points in the volume data, that share some scalar value e.g. density, pressure, etc. Extracting iso-surfaces from a volume, using the marching cubes algorithm, is a common way to convert volume data to a polygon mesh that can be rendered using triangle rasterization [JC06] [Ngu07].

If a point $\mathbf{x}$ inside the volume, and its associated scalar value $f(\mathbf{x})$, is considered, then from [EHK+06] we get the gradient of the scalar field $\nabla f(\mathbf{x})$:

$$\nabla f(\mathbf{x}) = \begin{pmatrix} \frac{\partial f(\mathbf{x})}{\partial x} \\ \frac{\partial f(\mathbf{x})}{\partial y} \\ \frac{\partial f(\mathbf{x})}{\partial z} \end{pmatrix} \tag{3.14}$$

This gradient will point in the direction of the steepest ascent, which is vector perpendicular to the iso-surface. As the gradient is typically not of unit length, it must be normalized to be used as a surface normal in lighting calculations:

$$\mathbf{n}(\mathbf{x}) = \frac{\nabla f(\mathbf{x})}{\|\nabla f(\mathbf{x})\|} \tag{3.15}$$

Equation 3.15 breaks down if $\|\nabla f(\mathbf{x})\| = 0$. This can happen in regions of the volume where the scalar field is either homogeneous, or has a local extremum. In this case the normal will considered to be zero, which means that illumination terms become zero.

Evaluating the gradient can be done in a number of ways. A very common approach is to use a method called central differences [EHK+06]. Central differences simply computes the averaged difference of values along each axis around the neighborhood of some point $\mathbf{x} = (x, y, z)$:

$$\nabla f(x,y,z) \approx \frac{1}{2h} \begin{pmatrix} f(x+h,y,z) - f(x-h,y,z) \\ f(x,y+h,z) - f(x,y-h,z) \\ f(x,y,z+h) - f(x,y,z-h) \end{pmatrix} \tag{3.16}$$

where h is some step size, typically the grid size. Central differences is fast to compute and can be done either on-the-fly during rendering, or pre-computed and stored for later use.

Chapter 4

Sparse Voxel Octrees

Sparse voxel octrees is a class of spatial subdivision data structures, used to efficiently store volume data in memory. Generally this means compressing constant regions of the volume and storing only the volume data needed for rendering the current view of the scene. Figure 4.1 shows examples of sparse voxels octrees.

There are a number of different ways to implement sparse voxel octrees, and the details depends on the application requirements and the algorithms used to traverse the tree. The information presented in this chapter is based mainly on the GigaVoxels engine from [Cra11][Eng10], because the data producers described in section 4.3.3 provides a natural extension point for procedural content generation. GigaVoxels uses a combination of cone tracing, which is a derivative of volume ray casting, and out-of-core data management to update a sparse voxel octree in a view dependent manner. The method relies on parallelization on graphics hardware using NVIDIAs CUDA platform, described in chapter 5, but the concepts can be applied in any parallel environment.

This chapter will first present the basic data structures, followed by the algorithms used to navigate the octree. The concept of cone tracing is defined, and finally the out-of-core update mechanisms are explained.

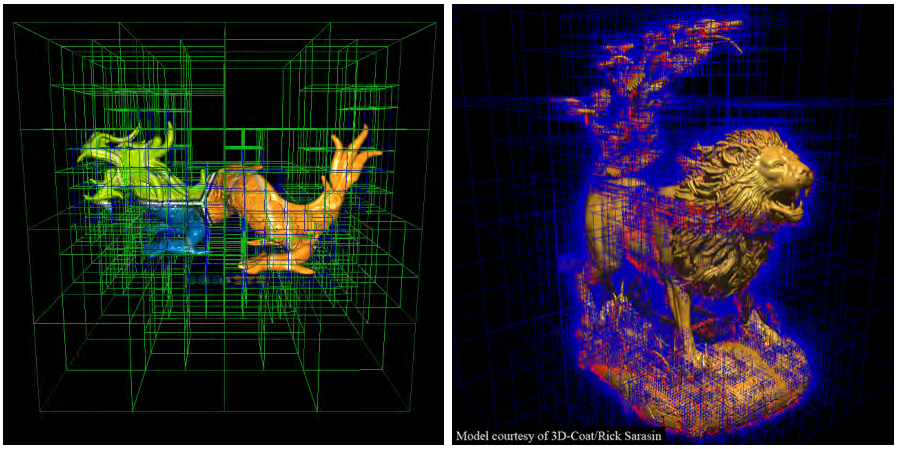


Figure 4.1: Examples of sparse voxel octrees. Notice the compact representation of empty regions of volume.
Source: [Cra11]

4.1 Data structures

The central component in GigaVoxels rendering approach is a hierarchical representation of the voxel data. This structure allows for dynamic updating of the volumetric data to fit the current rendering requirements, and a compact representation of empty space.

The volumetric data is encoded in a sparse voxel octree. Each node in the octree contains a region of the volume data. The root node contains the entire volume and subsequent levels subdivides the volume into eight equally sized regions. The structure is outlined in figure 4.2. The data associated with each node depends on the content of the region in the original volume data. Regions of homogeneous material can be efficiently represented with a constant color, while all other regions are stored as voxel volumes of some fixed resolution, called bricks. Each subdivision of non constant nodes further refines the level of detail in the bricks, similar to mipmap pyramid. On the other hand constant nodes requires no subdivision as the result would be eight new constant nodes, so constant nodes are considered terminal.

The octree is stored in two memory areas which are pre-allocated with some fixed size. This is commonly called a memory pool and is used to improve performance when a number individual elements, of some fixed size, needs to be managed. The nodes are stored linearly, as a pointer based tree, called the node pool. The bricks are stored in a volume texture called the brick pool.

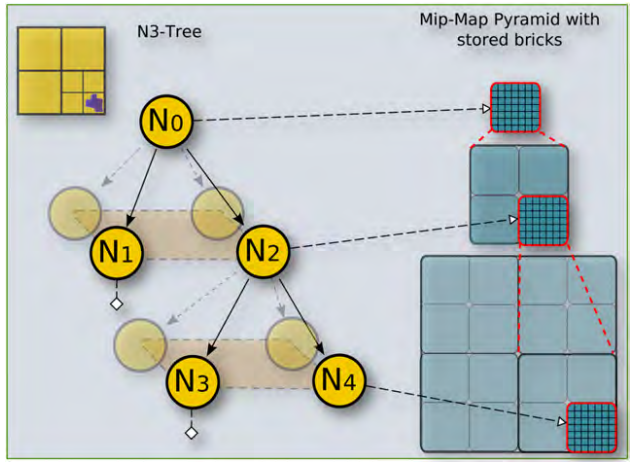


Figure 4.2: The GigaVoxels sparse voxel octree data structure. This structure stores a volumetric scene at multiple resolutions, where bricks are referenced by octree nodes creating a sparse mipmap pyramid.
Source: [Cra11]

Node Pool

The basic entity in the node pool is a grouping of eight nodes, called a node tile. Subdividing a single node will result in new node tile, containing the eight children stored contiguously in memory. This allows each node to point to all its children using a single node tile address and an offset in the range: [0..7]. This property is important during octree navigation, which will be described in section 4.2.1. As stated in section 4.1 the volume data associated with a node is either a constant color or a pointer into the brick pool.

Because memory needs to efficiently managed, node tiles may be recycled whenever the view of the scene changes, when this happens the least recently used node tiles are replaced with new data. For this reason a nodes location in the node pool, has no relation to its spatial location in the volume.

Brick Pool

The brick pool contains small cubic regions of volumetric data called bricks. All bricks have the same resolution, typically 16^3 or 32^3, but represents volume data at different levels of detail. In order to take advantage of hardware accelerated trilinear interpolation, the brick pool is stored in a volume texture. However

this presents a problem at the boundaries of the bricks where data will "bleed" from one brick to another when interpolated. The solution is to add a border all around the brick, effectively reducing the resolution of the brick.

4.2 Rendering

The rendering approach is based on volume ray casting as outlined in section 3.5. The main difference is that volume data can no longer be sampled directly from a voxel grid, but must be located in the octree. This means that the function `sample` is replaced with a descent into the octree that locates the node, containing the brick or constant. The node is then traversed using a local ray cast within its bounds, commonly called brick marching. This is outlined in figure 4.3.

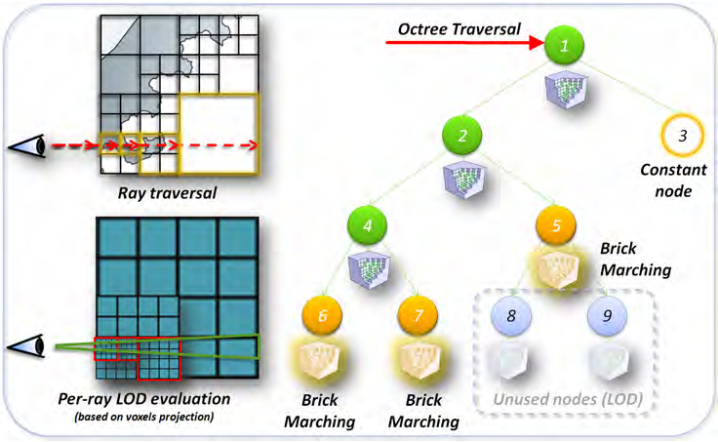


Figure 4.3: Illustration of octree traversal during ray casting. Bricks are located from the octree and marched.
Source: [Cra11]

4.2.1 Octree Descent

A number of different octree navigation algorithms exists. In [LK11] a stack of recently visited nodes is maintained. This keeps the cost of locating nodes in the same region of the octree low, at the cost of maintaining a stack for each ray cast. [Cra11] argues that stack based traversal tend to be inefficient because it requires additional hardware registers, which is an important factor in performance on current generation graphics hardware.

Another possible way to navigate a sparse voxel octree is to use an algorithm similar kd-restart presented in [HSHH07], which is simple to implement and requires no auxiliary data structures. When the node at position $\mathbf{p}$ needs to be located a search is initiated from the root node, as outlined in algorithm 2.

Algorithm 2: Octree descent using kd-restart

```
node ← getRootNode(octree)
while hasChildAddress(node) do
    offset ← (int(p_x * 2), int(p_y * 2), int(p_z * 2))
    index ← getChildTileAddress(node) + offset_x + 2 * offset_y + 4 * offset_z
    node ← getNodeByIndex(octree, index)
    p ← 2 * p − offset
end
```

The simplicity of the descent is made possible by the node tile layout in the node pool, discussed in section 4.1. Using the child address, which points to the first node tile, and an offset in the range [0..7], the correct child can be located. The position $\mathbf{p}$, which is updated to match the nodes local bounding box for each decent, can be converted to an offset by converting its integer components to a linear offset. This can be seen in figure 4.4. In practice, a full descent is rarely necessary as demonstrated in the next section.

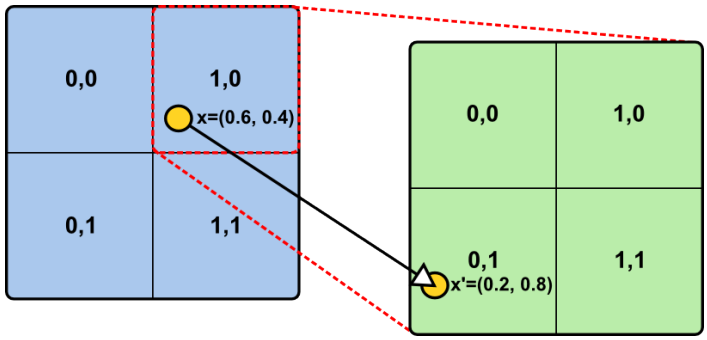


Figure 4.4: 2D example of offset calculating for the point $\mathbf{x} = (0.6, 0.4)$. The point $\mathbf{x}$ translates to child node $(1, 0)$ in the first node tile, and $(0, 1)$ one level deeper.

4.2.2 Volume Mipmapping

In classical ray casting each pixel on the screen is associated with a single ray in space. This can be a source of aliasing problems because the volume is sampled

exclusively along this ray, regardless of distance. Ray casting can counter the aliasing problems by casting multiple rays for the same pixel, this is called multi-sampling. However casting additonal rays incurs a negative impact on performance.

GigaVoxels uses an derivative of ray casting called cone tracing. It differs in the sense that each pixel corresponds to a cone, expanding from the eye into the volume, and as such is an area of the volume. The size of this area can be used to select a node of appropriate depth, and in turn volume data of a detail level matching the pixel size. This is similar to the mipmapping technique used in triangle rasterisation.

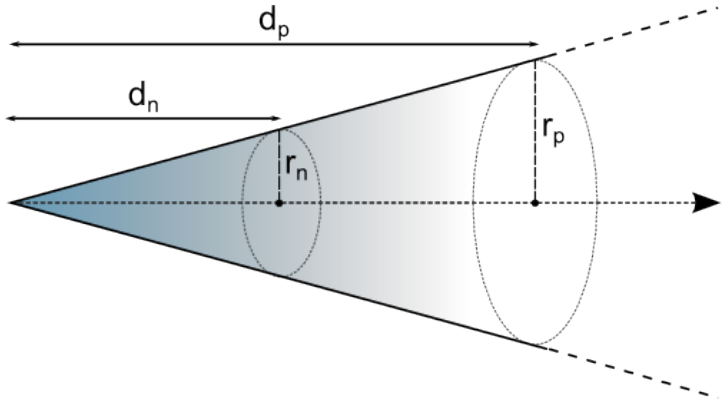


Figure 4.5: Radius of a cone at the near plane r_n and at a sample position r_p.

As the octree already encodes are sparse mipmap pyramid, it becomes a problem of selecting a brick of sufficient resolution from the octree. This must be done in relation to the size of a pixel on the screen. If we consider the cone in figure 4.5, we know from similar triangles that the following equation must be true:

$$r_p = \frac{d_p r_n}{d_n} \tag{4.1}$$

where r_p is the radius of the cone at sample position and r_n is the radius of the cone at the nearplane, d_p and d_n denotes the respective distances. r_n depends on the size of the screen S:

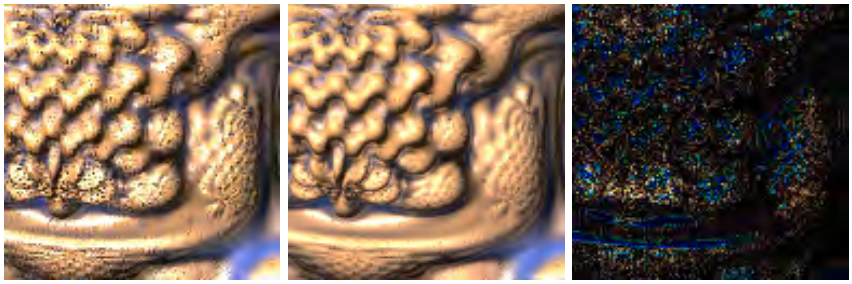
$$r_n = \frac{1}{2 \ \min(\mathrm{S}_{width}, \mathrm{S}_{height})} \tag{4.2}$$

The radius of a voxel, r_v in the octree depends on the depth in the octree D and the brick resolution B:

$$r_v = \frac{1}{2}^D \frac{1}{B} \tag{4.3}$$

which means that the size of a voxel at depth D is half the size of a voxel at size $D-1$. This means that during octree navigation, using algorithm 2, the descent can be stopped when $r_v < r_p$, as this indicates that size of single voxel will be less than the size of pixel on the screen.

While this approach is elegant it can lead to artifacts caused by discontinuity at locations where nearby pixels use different mipmap levels, see figure 4.6. The solution is to interpolate between the current mipmap level D and the immediate parent $D-1$. But since the octree is navigated using the kd-restart algorithm, the parent will always be available at no additional cost.



(a) (b) (c)

Figure 4.6: Comparison between rendering quality without mipmap interpolation (a) and with (b). The difference between the two images, with 2× amplification can be seen in (c).
Source: [Cra11]

4.3 Out-of-core Data Management

The GigaVoxels approach is designed to handle arbitrarily large voxel data sets. The main use case, as described in [Cra11] and [Eng10], is to stream data from disc or main memory, so that only the parts contributes to the current view of the scene are present in graphics memory. In very general terms this is done by issuing requests for missing data during ray casting. The data is then uploaded, replacing any data in the octree that has not been used recently.

4.3.1 Global Scheme

Figure 4.7 shows an overview of the GigaVoxels rendering engine. This figure highlights the three main components in the method. The ray caster, which is responsible to rendering the final image, updating usage data, and emitting requests for needed data that is currently not present in the sparse voxel octree. The cache manager is responsible for sorting these requests, and prioritizing what data should be recycled. Finally the producer updates the octree by placing the new data in node and brick pools.

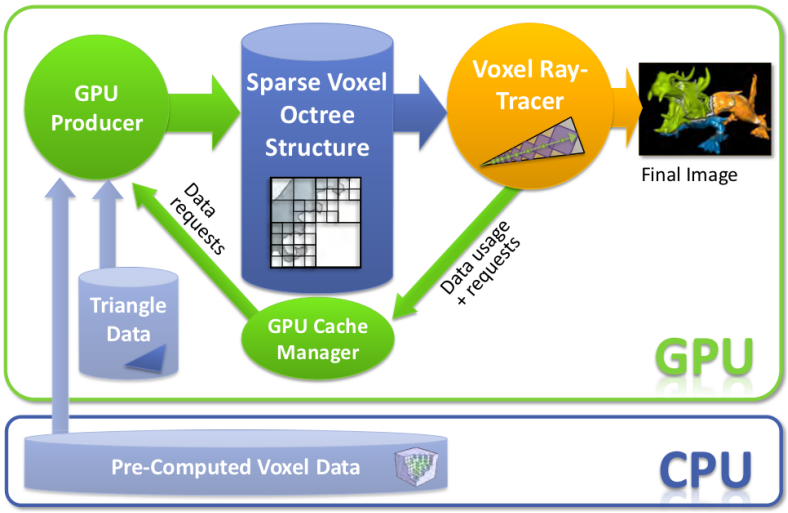


Figure 4.7: Overview GigaVoxels rendering engine. The voxel raytracer emits data requests while travsering the octree structure. Data is then generated using producers, and the octree structure is updated.
Source: [Cra11]

4.3.2 Node Requests

The ray caster uses a combination of the ray casting algorithm, outlined in algorithm 1, and octree decent from algorithm 2. While descending in the octree structure, the ray caster might encounter a node that does not yet have the sufficient level of detail. When this happens it emits a subdivision request for the node.

As this can happen in parallel for all rays, multiple threads might request the same node index. Following principals in parallel programming this must be

done in a way where no race conditions occur, but for performance reasons it must be done as a non locking operation. By writing the index of the requested node into a buffer, of the same size as the node pool that is cleared before each rendering pass, in the following way: `requestBuffer`$[index] \leftarrow index$, it is guaranteed that a specific request will appear at most once, and simultaneous request will not interfere with each other. After the request buffer has been filled, it is sorted so all non requested node indexes are moved to the front of the buffer.

4.3.3 Data Producers

Data producers responds to requests, emitted by the ray caster, by updating the octree structure with data from some source. Typically, this source is a pre-filtered voxel data set stored on disk or main memory. When a node is requested, the corresponding data should be located and uploaded to graphics memory. This presents a problem because the node index has no relation to the nodes spatial location in the volume. GigaVoxels solves this with an auxiliary data structure, the localization buffer, which maps the node index to its spatial location.

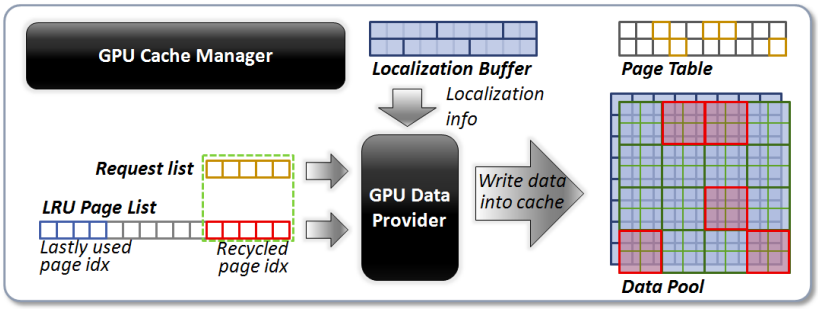


Figure 4.8: Data producers take the request, the least recently used elements in the pools and the localization buffer in order to update the pools.
Source: [Cra11]

The localization buffer encodes the octree location for each node in the node pool. It consists of the node depth and three bit vectors, one for each axis, corresponding the navigation choice in the octree at a certain depth. Given the original bounding box of the volume, this information is also sufficient to calculate the bounding volume of the octree node.

Data producers are not limited to loading pre-filtered voxel data sets. Given

the bounding volume it is also possible to procedurally generate data on-the-fly. Neither [Cra11] or [Eng10] supplies any specific implementation details on procedural content generation using data producers, besides stating that it can be a slow operation.

4.3.4 Cache Mechanism

In order to allocate space for new nodes and bricks, both memory pools are managed as caches by recycling the least recently used elements. To maintain a list of candidate elements for recycling, the ray caster provides usage information collected during rendering. This information is tracked using two timestamp buffers, indicating when the node tile or brick was last used.

Using the timestamps buffers alone would require data producers to sort all timestamps each time elements needs to be recycled. With the additional constraint, that this needs to happen in massively parallel environment it becomes infeasible. The solution is to introduce an additional list, called a usage buffer, for each pool. An entry in the usage buffer consists of an index into its respective pool, i.e. the node tile index or the brick index, and a flag that indicates if the element was used during this frame. This flag is updated after the ray caster has updated the timestamps. If the elements entry in the timestamp buffer does not match a current global timestamp, it has not been used during this frame.

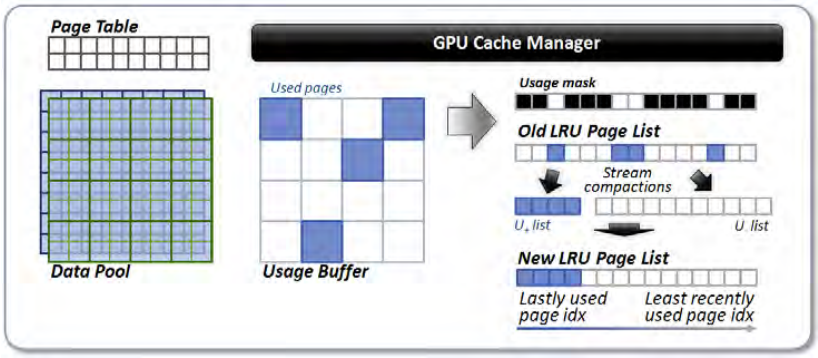


Figure 4.9: An incremental update of the usage mask, based on timestamp information.
Source: [Cra11]

The final part of the process is to ensure that the least recently used elements are moved to the front of usage buffers, where they can easily be located by the data producers. This can be done efficiently by a method called stream compaction.

By continuously partitioning the updated usage buffers, into flagged and unflagged elements, a list of least recently used elements emerge. The process can be seen in figure 4.9.

4.3.5 Cache Invalidation

As described in the previous section, the node and brick usage buffers will be sorted according to timestamp information, with the least recently used elements at the front of the list. These elements will be reused and when the octree structure needs to be updated. But as these elements might still be in use, it is necessary to invalidate any part of the octree structure that still references these elements. This is a non trivial problem because GigaVoxels supports recursions and instancing, which means that a single node or brick could potentially be referenced by more than one parent, see figure 4.10.

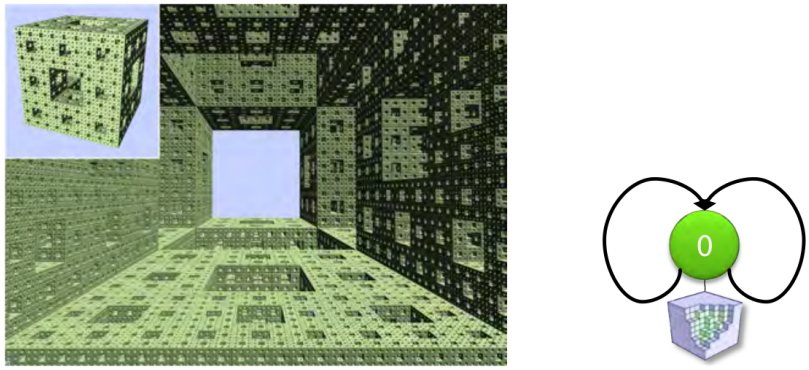


Figure 4.10: GigaVoxels supports recursivity in the octree, as shown in this example of a Sierpinski sponge fractal. This complicates the invalidate procedure.
Source: [Cra11]

This problem is solved with a two step procedure. After the requests from the ray caster has been collected, the number of node tiles and bricks that needs to be recycled can be determined. The first step is to flag all elements that needs to be recycled. This is done by setting their value in the timestamp buffers to some special value (zero is reserved for this purpose). The second step is to test every single node in the octree to see if it references a node or brick with a timestamp of zero. If this is the case, the corresponding pointer is nulled.

After the invalidation procedure is complete, the elements at the front of the usage buffers can safely be reused.

Chapter 5

CUDA Programming Model

This chapter serves as a brief introduction to the parallel computation using consumer graphics hardware with NVIDIAs Compute Unified Device Architecture, also nown as CUDA.

5.1 A Brief History of Graphics Hardware

Graphics hardware has been in steady development for many years. The development has, at least in part, been driven by a demand for consumer graphics hardware capable of rendering 3D graphics in real-time. In the late 1990s NVIDIA launched the GeForce 256, which was the first consumer level graphics card that supported transform and lighting operation directly on the graphics processing unit (GPU) [SK10]. Since then GPUs has continuously been evolved to extend their flexibility. Programmable shaders allowed parts of the rendering pipeline, the transformation from triangles to pixels on the screen, to be replaced with application specific code.

In 2006 NVIDIA launched the GeForce 8 series, which was the first range of GPUs that was built on the CUDA architecture. Unlike the previous generations of GPU hardware, where the computing resources were partitioned into

specific steps in the rendering pipeline (vertex and pixel shaders), the CUDA architecture was constructed as a uniform shader pipeline [SK10]. This meant that every arithmetic logic unit could be used for general purpose computing. Furthermore, the execution units were allowed read and write access to arbitrary memory on the hardware, which was not possible before. Methods such as ray casting, that were previously only possible to do in software, could now be implemented using hardware acceleration.

5.2 Programming Model

At the top level CUDA operates with two concepts. The host, which is the CPU and the main memory, and the device, referring to the graphics hardware. The host executes code on the device by launching kernel code which is then run in parallel on all the streaming multiprocessors (SMs) on the graphics hardware. The SMs handles the actual execution, and each of them has their own control units, registers, caches and cores. The amount of cores determines the number of instructions that can be executed in parallel. In the third generation of the CUDA architecture from 2010, codenamed Fermi, each SM has exactly 32 cores [Cor09].

The basic unit of execution for a CUDA kernel is a thread. Threads are organized into blocks, which then again are divided into a grid. Grids and blocks can be one-, two- and three-dimensional, depending on how the kernel is executed. This hierarchy can be seen in figure 5.1.

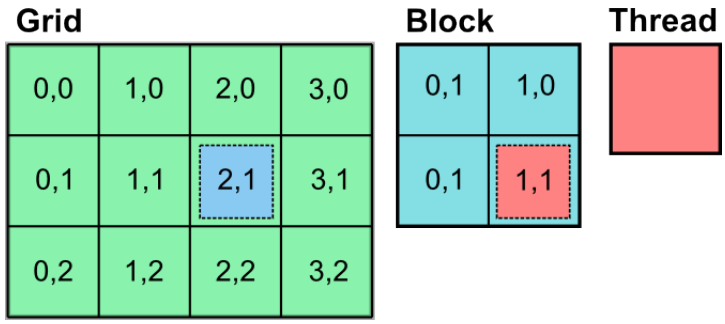


Figure 5.1: The CUDA Thread hierarchy. The grid defines a number of blocks, and the blocks are divided into threads.

Grids and blocks serves as a tool for organizing the relationship and dependency between threads. When a kernel is executed, the blocks are distributed across

the available SMs, which allows the execution to scale with more powerful hardware [Cor09]. This is illustrated in figure 5.2. The number of simultaneous threads a single SM can handle in parallel depends on the amount of cores, but this might not match the exact number of threads in a block. For this reason the actual thread execution happens in groups called warps, which is a subset of the threads in the original block. Every thread in a warp executes the same instruction on different data. This implies that if one thread takes a divergent branch, the entire warp execution has to wait, a process called serialization [Cor12]. This means that in order to optimize performance, threads allocated in the same warp should take similar paths through the kernel code.

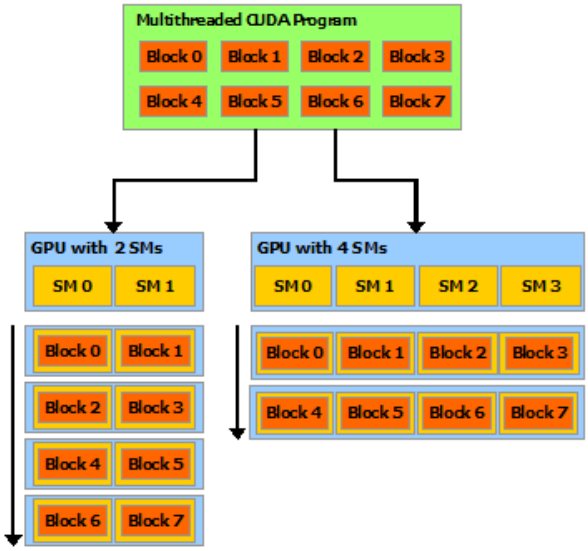


Figure 5.2: Thread blocks are distributed across the available streaming multiprocessors (SMs). This allows CUDA programs to scale with more powerful hardware.
Source: http://docs.nvidia.com/cuda/cuda-c-programming-guide

5.3 Memory Model

CUDA kernels work exclusively with device memory, which is divided into a number of different memory areas[1]. Global memory is accessible from all threads and can be allocated from the host as either linear memory or as CUDA Arrays.

[1] For a full overview of the CUDA memory model, see: [Cor12]

Linear memory exists in 32/40 bit address space[2], allowing separate entities to reference each other using pointers. This makes linear memory suitable to for containing tree structures. CUDA Arrays are optimized for texture fetching, and supports different interpolation schemes for data reads. They can also be accessed as surface memory which allows writing to texture memory from kernels.

Finally it is possible to map selected resources from graphics libraries (OpenGL, Direct3D) to CUDA device memory. Using this technique it is possible for CUDA to read and write directly to buffers allocated by the graphics library.

[2] The address space depends on the compute mode capabilities of the GPU, see [Cor12]

Chapter 6

Analysis and Implementation

This chapter presents an analysis and implementation details, of a CUDA based ray caster, for rendering large completely procedural volumetric scenes in real-time. The GigaVoxels rendering engine, outlined in chapter 4, has shown good results when dealing with complex volumetric scenes and serves as a foundation for the method described in this chapter. The implementation has been made from scratch based on information from [Cra11] and [Eng10].

Two contributions has been made that distinguish this work from the standard GigaVoxels engine. The first contribution is a three step method for efficiently subdividing nodes and updating the octree structure, in a purely procedural environment, and is described in section 6.2.1. The second contribution is a simplification of the invalidation procedure which is described in section 6.3. Together the contributions helps to form a framework for procedural content generation using a density function, as described in chapter 2.

6.1 Data Formats

The following is a description of the data formats and their interpretation, which will be helpful in the remaining sections of this chapter.

6.1.1 Octree Representation

The otree consists of two data structures: the node pool and the brick pool. As described in section 4.1, the node pool is divided into groups of 8 nodes, called a node tile, that represents all children of a single node. During a decent into the octree only the nodes structural information, such as child addresses and if the node is terminal, is typically of interest. To exploit cache coherency, octree nodes are split into two data structures each encoded in a 32 bit value. They are stored linearly in memory as a structure of arrays, meaning that all data used for traversal is stored contiguously. This increases the chance that threads, which takes a similar path in the octree, will have access to a cached version of the data.

Node Pool

The main data structure of the octree node pool is called `NodeChildData` and is shown in figure 6.1. This data structure contains the address of the first child in the node tile, `childAddress`, and the flag `maxSubdivision` signaling if the node is terminal. It also contains a flag `dataType`, that determines that interpretation of the second data structure. If the flag is `DATA_TYPE_CONSTANT` it means that the second 32 bit value should be interpreted as a `NodeConstantColor` otherwise it is a `NodeBrickPointer`.

Figure 6.1: `NodeChildData` is encoded in a 32 bit int. It contains the address of the first child in the node tile, and two flags signaling: if the node is at max subdivision level, and the data type of the volume data associated with the node.

Figure 6.2 shows the two possible interpretations of the second 32 bit value. The first possibility is that the node contains a region of volume data in the form of a brick. In this case it should be unpacked as a `NodeBrickPointer` that

points into the brick pool using (x, y, z) coordinates. The second possibility is that the node is entirely homogeneous. This is represented by a RGBA color, where completely empty regions have a zero alpha component.

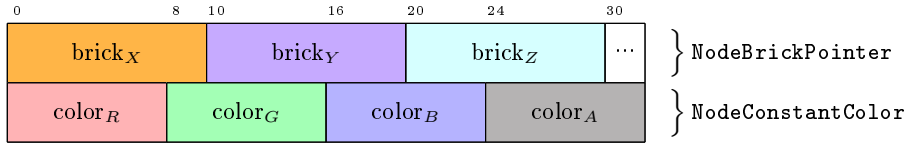


Figure 6.2: The two possible interpretations of the second 32 bit value. `NodeBrickPointer` encodes a pointer into the brick pool. `NodeConstantColor` encodes a constant color.

Brick Pool

The brick pool is implemented using CUDA Arrays, which are configured to act as writable volume textures. The arrays are allocated at startup, for some fixed volume dimension, and then divided into a number of smaller cubic regions that serves as bricks. Bricks are referenced from nodes using a `NodeBrickPointer` as described in the previous section.

The number of arrays and their layout depends on the application requirements. This implementation uses two CUDA Arrays as shown in figure 6.3. The first array describes the voxel color in RGB format, while the second contains the density of the voxel. Storing the density as a full 32 bit float might seem excessive, but as normals are not explicitly stored they must be calculated using central differences during rendering. On the other hand, if normals were pre-calculated they would have to be stored with sufficient precision to avoid artifacts. The last 8 bit in the color array could be used to store material parameters, but is currently unused.

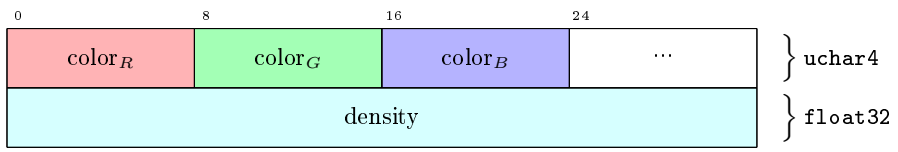


Figure 6.3: Voxel format used in bricks.

6.1.2 Usage Data

Usage data is stored as a combination of timestamp and usage buffers as described in section 4.3.4. All timestamps are stored as 32 bit ints. As nodes are managed on a tile basis, only one timestamp per tile needs to be stored in the array `tileTimestamp`. Brick timestamps are stored in the array `brickTimestamp` with one element per brick.

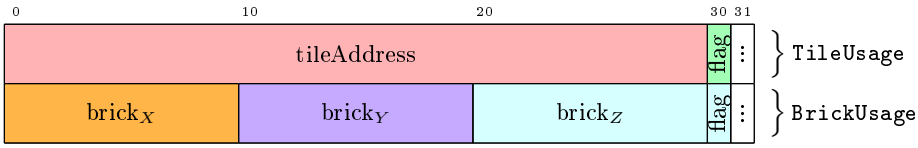


Figure 6.4: Usage information encoded in 32 bit values. Both `TileUsage` `BrickUsage` contains a 1 bit usage flag which is updated according to timestamp information, and is used to sort the usage buffers as described in section 4.3.4.

The usage buffers contains a list of all node tile addresses and brick pointers, sorted in a least recently used fashion according to the timestamp information. Figure 6.4 shows the encoding of the usage elements `TileUsage` and `BrickUsage`. As elements in the usage buffers are constantly moved around, their indexes does not correspond with the pools or timestamp buffers. Instead, the address information contained in the usage elements is used to map back to the correct index. For example the `tileAddress` in `TileUsage` maps directly to the index in the node pool, and `tileAddress`/8 maps to index in the `tileTimestamp` buffer.

6.1.3 Localization Data

Localization data is used during procedural content generation to determine the spatial extent of a nodes bounding box. `NodeLocalizationCode`, shown in figure 6.5, contains three bit vectors representing the choice along each axis in an octree descent. An 8 bit value `NodeLocalizationDepth` determines the depth of the node, and therefore how many bits of the choice vectors should be used.

The node pool has the potential to scale to 1073741823 elements[1], which is roughly the number of nodes in a fully populated octree of depth 10. The 10 bit

[1] 1073741823 is the maximum number of nodes addressable by 30 bit node address

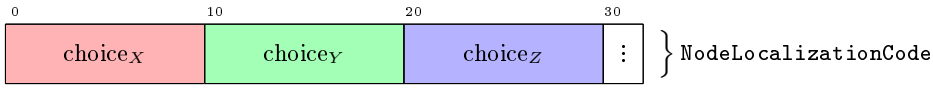


Figure 6.5: `NodeLocalizationCode` encodes localization information in 32 bit value. The contents is three bit vectors of 10 bit each, that represents the choice along each axis in an octree descent.

choice vectors restricts the depth to a maximum of 10. Exploiting the full size of the node pool, to accommodate deeper sparse voxel octrees, could be done by extending the choice vectors to some appropriate size.

6.2 Procedural Content

As described in section 4.3.3 the GigaVoxels octree structure is updated using data producers. These producers responds to node requests emitted by the ray caster during rendering. The GigaVoxels rendering engine makes the distinction between subdivision requests and data requests. This is done by analyzing the state of the requested node. This process is described in [Eng10]. The idea is that nodes can be subdivided, without necessarily uploading the associated brick data to the GPU, which can be a relatively costly operation.

While the distinction between subdivision and data requests works well for precomputed voxel data, where it is known in advance whether a node is constant and terminal, this is problematic for completely procedural volume data. There is no way to know if an octree node is terminal without evaluating all voxels insides its corresponding brick. Given this observation, a node request should always perform a complete subdivision and brick evaluation. This also simplifies the update process as no preprocessing step has to be performed prior to the node requests.

6.2.1 Three step method

Using a single kernel for subdivision and brick evaluation would mean that a single thread is responsible for calculating the contents of every voxel in all eight resulting bricks. In order to effectively exploit parallelization with CUDA during processing of node requests, a three step method is used.

The first step is to subdivide the nodes. This is followed by the evaluation of

all voxels in the resulting bricks and lastly the nodes are finalized. Finalization means to check if the nodes should be considered constant. At each step the number of threads running simultaneously should be maximized.

Node Subdivision

The first step is to handle subdivision of all nodes that the ray caster has requested during rendering. After a single rendering pass the request buffer, `raymarcherRequests`, contains the indexes of all nodes that did not contain a sufficient level of detail for the current view of the scene. As described in section 4.3.2, multiple rays may potentially request the same node, which was solved by writing node index to the request buffer in a way that is robust against race conditions. Each requested node index appears exactly once in the request buffer, but scattered with gaps between them corresponding to the node indexes that were not requested. These gaps are removed by using a stream compaction operation[2], that that pulls all non empty node indexes to the front of the buffer.

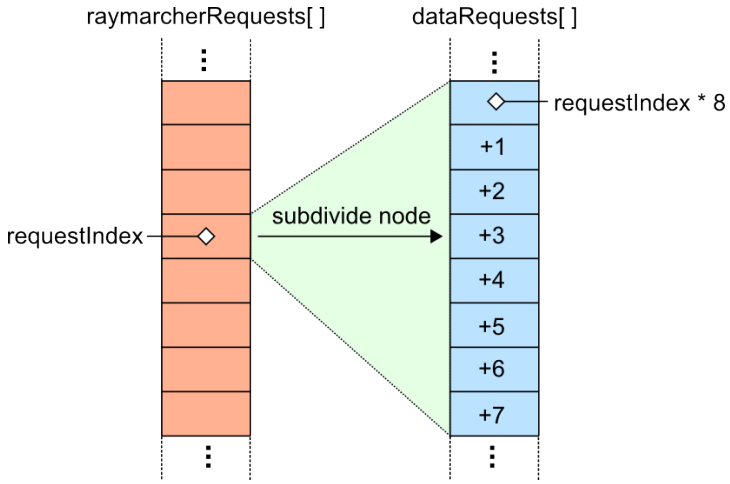


Figure 6.6: The node subdivision process. The array `raymarcherRequests` contains a list of all nodes that must be subdivided. A single CUDA thread handles subdivision for a single node. Each node is allocated a node tile to contain its children and the indexes of these nodes are placed into the array `dataRequests`.

Once all node requests are located at the front of `raymarcherRequests` they can

[2]All stream compaction operations are implemented using CUDA Thrust which is essentially a collection of parallel algorithms that resembles the C++ Standard Template Library (STL). More information is available at https://code.google.com/p/thrust/.

be processed. Subdividing a node requires allocating a new node tile that will contain all eight children. This is done by executing a kernel with one thread per requested node. The maximum number of requests considered should be limited, in order to avoid the process consuming to much time and causing low frame rates. One brick must be evaluated for each child in the new node tile, and this can be tracked using a second request buffer called `dataRequests`. For each child node a single brick is reserved from the brick pool, and the child nodes index is placed in `dataRequests`. This array is allocated to contain the maximum number data requests per frame, which is eight times the maximum number of subdivide requests per frame. This is shown in figure 6.6.

Data Requests

The next step is to evaluate all voxels associated with the bricks of the newly created nodes placed in `dataRequests`. Parallelization with CUDA can be used to optimize this process. Each element in `dataRequests` maps to a brick that must be processed, and each brick contains N^3 voxels where N is the brick resolution. This translates to launching N^3 threads per element in `dataRequests` with the sole responsibility of evaluating a single voxel.

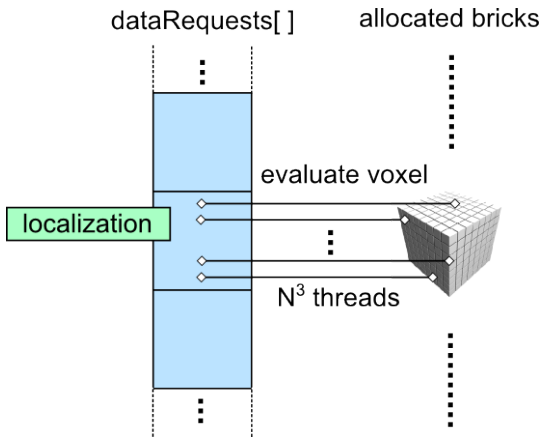


Figure 6.7: Handling data requests. For each element in `dataRequests` N^3 threads are launched where N is the brick resolution. Each thread evaluates a single voxel based on its world space position.

A three dimensional CUDA kernel is launched, which maps to a specific voxel position $x \in [0..1]^3$ within a brick associated with a specific child node from `dataRequests`, see figure 6.7. Using the localization buffer, described in section 6.1.3, the coordinate x can be multiplied with the bounding volume of the

child node yielding the exact location of the voxel in the volume. The volume coordinate and the depth of the node is passed to a function, that returns the density and color of the voxel.

Finalize Child Nodes

The final step is to determine if any of the newly created child nodes should be flagged as constant and terminal. Based on the density model described in section 2.3 this can be the case in one of two situations: Either the density of all voxels in the brick is below the threshold μ_{air}, indicating that the node is completely empty, or all densities are above μ_{solid}, which means that the node is completely solid.

This requires the context of the entire brick which means a kernel, with a thread per element of `dataRequests`, is launched. It loops over each voxel in the newly evaluated brick and continuously updates the minimum and maximum density values encountered. If the maximum density value, for all voxels, is below the threshold μ_{air}, then the brick is recycled and the brick pointer is converted to a constant color with a alpha value of zero. Analogously if the minimum threshold is above μ_{solid}, then the constant color is set to the average color of the voxels, and the brick is recycled. A different strategy could be to keep the brick, and flag the node as terminal. But completely solid bricks will typically not be visible, as the nodes that make up boundary of the level set would occlude them.

6.2.2 Voxel Evaluation

Voxel evaluation is done with a density function, as described in chapter 2, with the addition that a color for the voxel is also returned. As the CUDA Array that stores the color and density values is configured to support trilinear interpolation, care must be taken at the borders of the bricks. Figure 6.8 shows a 2D case where values are sampled near the border.

If the implementation used pre-computed normals it would be sufficient to have a single voxel border surrounding the brick. But since normals are computed during rendering, using central differences and triliear interpolation, the sampling neighborhood extends two voxels beyond the voxel center. For this reason it is necessary to use a two voxel wide border around the brick. Alternatively, it would be possible to treat normal estimation at brick borders as a special case. But this would introduce divergent branches in a central part of the implementation, and incur a negative performance impact.

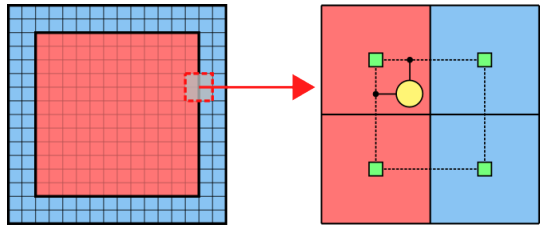


Figure 6.8: Voxel interpolation requires a border around the brick. Because normals are estimated with central differences, this border must be two voxels wide.

This means that before a voxel can be evaluated, the octree nodes bounding box must be expanded to account for the brick border. During brick marching the reverse operation must be performed.

6.3 Cache Invalidation

As described in section 4.3.5, the invalidation process in GigaVoxels is non trivial to perform, because it supports recursive octree definitions. While recursive structures and instancing could make sense in the context of entirely procedural volume data, for example instancing of trees or rocks, for procedural generation of terrain, it complicates the framework and adds a potentially very costly scan of the entire node pool during each frame.

Invalidation can be simplified by disallowing multiple references to node tiles and bricks, while tracking where in the octree they are referenced. To do this two additional buffers are introduced, `tileAttachment` and `brickAttachment`. When a node tile or a brick is allocated from the pools, the address of the node pointing to the element is put into the respective attachment buffer. When a node tile or a brick is about to be recycled, the attachment buffer can then be checked to see if the element is in use, and what node in the octree has a pointer to it. This node may then be invalidated.

6.4 Rendering

Ray casting in CUDA can be done by launching a two dimensional kernel, sized to fit the target image resolution, as seen in figure 6.9. Block and thread indexes

are used to map the particular thread instance to x, y coordinates in the image. This thread is responsible for casting the corresponding ray. A nice property is that all treads in a block, will have rays with similar directions and typically require the same octree nodes. This will keep warp serialization due to divergent execution paths, as described in section 5.2, as low as possible.

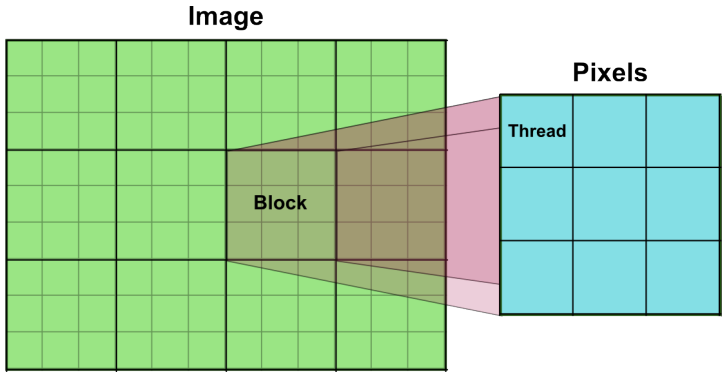


Figure 6.9: CUDA Raycasting. The image is divided into a grid of blocks, and each block is divided into a grid of threads. Each thread handles, casting a single ray for its corresponding pixel.

The mapping from the thread pixel coordinates x, y in the image, to the near plane coordinates u, v is illustrated in figure 6.10. Image coordinates x, y are normalized by dividing with the width and height of the image, and mapped to the range $[-1..1]$. They are then converted to near plane coordinates u, v by multiplying them with the view frustum values for top and left (shown in figure 6.11b), taking into account that the y axis in the image is inverted.

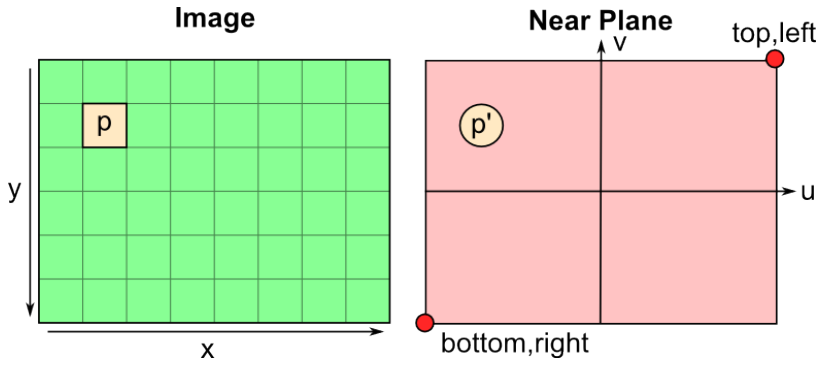
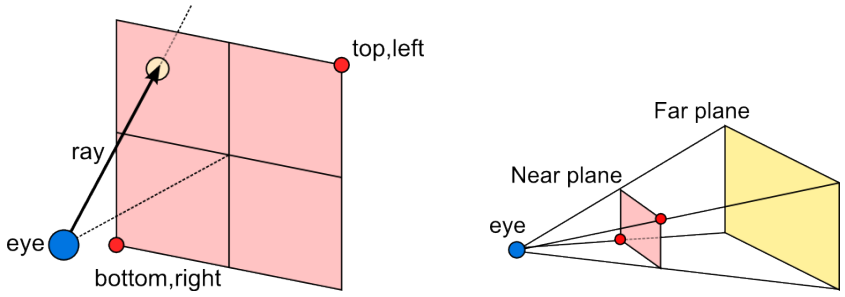


Figure 6.10: The pixel coordinates for p in the image is mapped to a position p' on the near plane.

Using the u, v coordinates, of the point on the near plane, a ray is created from the eye towards the point. This is shown in figure 6.11. The ray originates at the eye, which has coordinates $(0, 0, 0)$ in view space, and has a direction towards the u, v coordinates on the near plane. In order to transform the ray into world space, it is multiplied by the inverse view matrix.



(a) (b)

Figure 6.11: Rays originates at the eye and has a direction towards the pixel position on the near plane, as shown in (a). The view frustum is shown in (b), where the top, left values are shown in relation to the eye.

While ray casting is performed entirely within a kernel, the image needs to be copied to the screen buffer before it is visible in the application. To avoid copying the image rendered by the ray caster multiple times, it should be directly accessible from the graphics library which will display the image. As described in section 5.3 CUDA supports interoperability with existing graphics libraries, in this case OpenGL.

One way of approaching the problem is to create an OpenGL pixel buffer of sufficient size to hold the pixel data written by CUDA. This pixel buffer can be mapped to CUDA device pointer, which can be accessed in the same way as a normal array stored in global memory. This requires remapping the pixels x, y coordinate to a linear offset into the pixel buffer. This device pointer is injected into the kernel, where pixel values are written. When the kernel is done the pixel buffer must be unmapped to return control to OpenGL. Finally the image can be copied to the screen buffer, by rendering a screen filling quad with the pixel buffer mapped as a texture. The approach is outlined in figure 6.12.

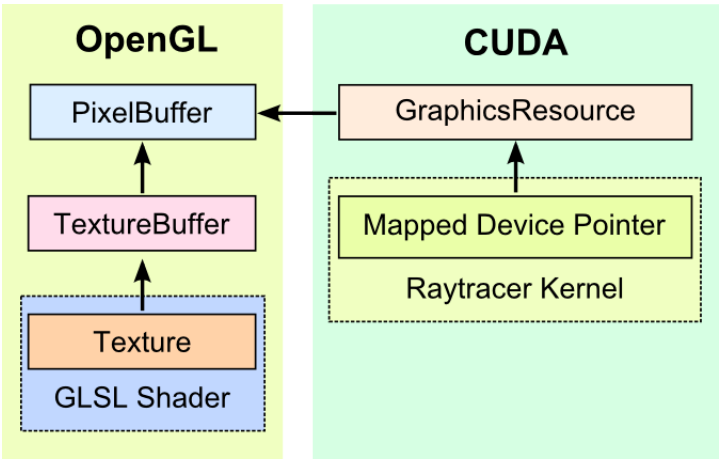


Figure 6.12: The relationship between the pixel buffer, the CUDA device pointer used to access inside the kernel, and the texture used in the GLSL shader.

6.4.1 Ray Casting Algorithm

The ray casting pseudocode used in the implementation is outlined in algorithm 3. It is an extension of the ray casting algorithm presented in chapter 3, that incorporates octree descent and brick marching. The algorithm can be broken into 5 main parts:

Ray setup View rays needs to be correctly setup according to the view parameters. The ray originates at the eye and the direction is determined by the position of the pixel on the near plane, as described in the previous section.

Volume Intersection The ray r is then intersected with the volume bounding box to determine if it hits, and where along the ray it enters the volume, $near_{volume}$, and where it exits the volume far_{volume}. See figure 6.13.

Node Lookup While traversing the rays through the volume, the corresponding octree nodes must be located. The octree node is found by doing an octree descent as described in section 4.2.1 while using the projected voxel size, v_{size}, on the near plane as per section 4.2.2. The function `octreeDescent` is also responsible for emitting subdivision requests, if no node of suitable depth exists.

Node Intersection Once the node has been located, a local intersection with

the nodes bounding box is made to determine the entry and exit points $near_{node}$ and far_{node} of the ray. See figure 6.13.

Node Traversal The traversal complexity depends on the type of node. For constant nodes, i.e. nodes without a brick pointer, the traversal is straight forward, as they are either completely empty or solid. For brick nodes a local ray cast is performed on the volume data referenced by the brick pointer. This ray cast must respect the brick borders, which keeps voxel data from bleeding during interpolation, as described in section 6.2.2.

After the node has been traversed, the distance along the ray t_{volume} is advanced to the exit point of the node far_{node}. In practice it seems to be a good idea to add a small epsilon value to the exit distance. This avoids problems, caused by numerical errors due to glancing ray angles, that would cause the exit point to remain within the bounds of the node.

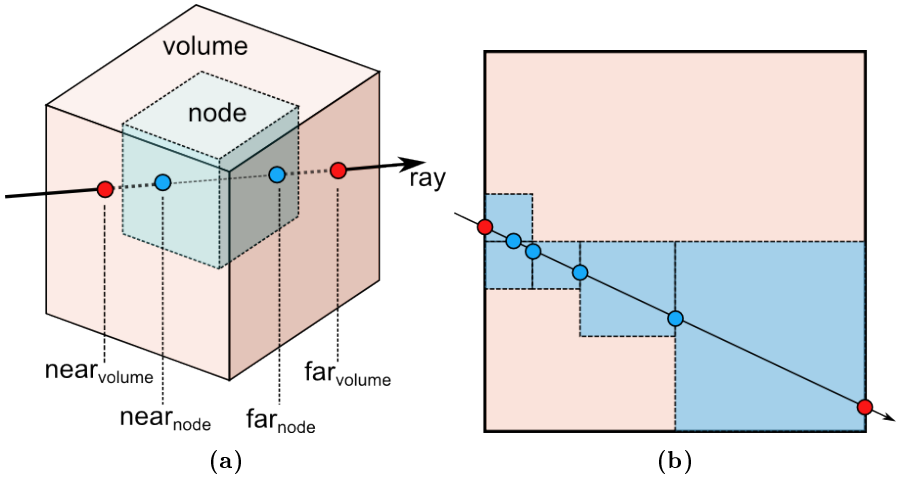


Figure 6.13: Illustration of volume and node intersections with a ray. In (a) the intersections are conceptualized in 3D and (b) shows how this translates to volume mipmapping, where nodes are selected based on the distance to the eye.

Algorithm 3: Pseudocode for octree volume ray casting

```
1  foreach p_near ∈ image plane do
2      r ← ray from p_eye to p_near
3      ⟨hit, near_volume, far_volume⟩ ← intersectVolume(r)
4      if hit then
5          t_volume ← near_volume
6          while t_volume ≤ far_volume do
7              p_volume ← r_origin + r_direction * t_volume
8              v_size ← size of voxel at distance t_volume
9              node ← octreeDescent(p_global, v_size)
10             ⟨near_node, far_node⟩ ← intersectNode(r, node)
11             if node has brick then
12                 t_step ← step size at the depth of node
13                 t_node ← near_node
14                 while t_node ≤ far_node do
15                     p_node ← r_origin + r_direction * t_node
16                     ⟨C_src, α_src⟩ ← sample(brick, p_node)
17                     C_dst ← C_dst + (1 − α_dst)C_src
18                     α_dst ← α_dst + (1 − α_dst)α_src
19                     t_node ← t_node + t_step
20                 end
21             else
22                 ⟨C_src, α_src⟩ ← costantColor(node)
23                 C_dst ← C_dst + (1 − α_dst)C_src
24                 α_dst ← α_dst + (1 − α_dst)α_src
25             end
26             t_volume ← far_node
27         end
28     end
29     write the final color C_dst into the image at the pixel location
30 end
```

Chapter 7

Results and Discussion

This chapter will evaluate and discuss the results of the implementation described in chapter 6. The performance of the individual components will be evaluated quantitatively, as well as their memory consumption. Invalidation and image quality problems related to the three step subdivision process will also be examined and discussed.

7.1 Performance Analysis

In order to study the performance rendering and cache mechanisms the implementation have been tested in some typical usage scenarios. All tests have been performed using a NVIDIA Geforce GTX 670 and a Intel Core 2 Duo CPU running at 2.6 GHz.

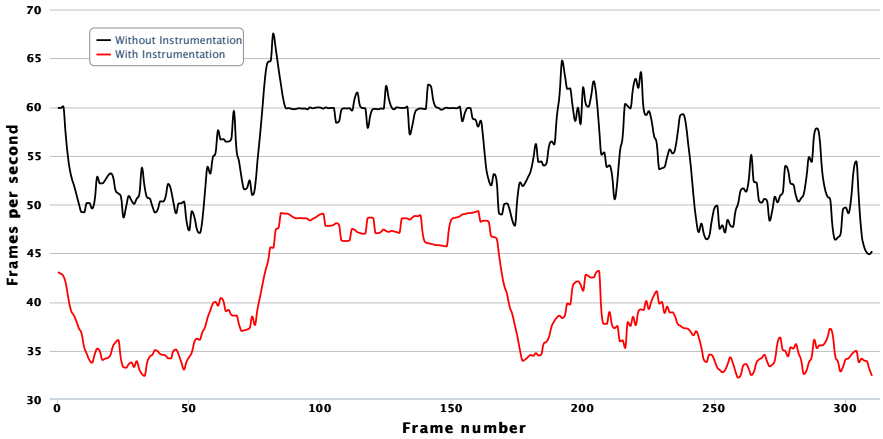


Figure 7.1: The difference between frames per second measured with and without the performance impact incurred by instrumentation. Actual performance, in a typical usage scenario, will be about 15 frames per second better.

7.1.1 Measuring Approach

The measurements have been made with a combination of CUDA event timers[1] and in-kernel timers, that allows to profile specific segments of a CUDA kernel[2]. Both types of measurements requires code instrumentation, and does incur a negative impact on the performance. In the following scenarios the actual number of frames per second will, on average, be around 15 higher if instrumentation was disabled. This can be seen in figure 7.1.

The overall frame rate is measured by the engine and includes all overhead from input processing, OpenGL rendering and animation systems as well as the CUDA kernels.

7.1.2 Global Performance

Due to the nature of the implementation, the performance must be analyzed as a sequence. Each frame consists of a rendering phase and a subsequent subdivision

[1] Full details of the CUDA Event API can be found here: http://docs.nvidia.com/cuda/cuda-runtime-api/.

[2] The in kernel timers are inspired by this thread on stack overflow: http://stackoverflow.com/questions/11209228/timing-different-sections-in-cuda-kernel.

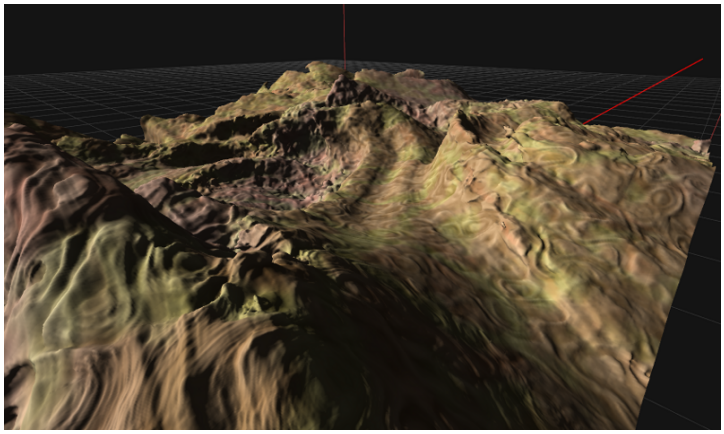


Figure 7.2: This image shows the scene used for the performance analysis.

phase, where the cache mechanism subdivides nodes and produces new volume data in the form of bricks. These bricks will then be available in the next frame and used for rendering. This means that a change in the camera configuration will cause the need for subdivision.

The following analysis is based on data gathered during a key framed animated fly by of the scene shown in figure 7.2, rendered at 800×600. The key frames has been setup to highlight three different scene exploration speeds, which means camera movement and rotation. At the start of the animation only the visible part of the scene is cached, and any new parts of the octree must be subdivided, as part of the exploration.

Figure 7.3 shows the cost, in milliseconds, of the rendering and subdivision phases, as well as the overall frames per second achieved. The animation sequence can be divided into three parts: **A** (frames 0 to 70) which consists of some fast movement, **B** (frames 70 to 170) where the camera is slowly rotated and finally **C** (frames 170 to 310) where the camera is slowly accelerated.

The first thing to observe is the fairly consistent performance of the ray casting phase, even when the layout of the scene changes rapidly. An important part of the steady rendering performance is due to the cone tracing model described in section 4.2.2. Bricks are selected from the octree to match the their projected screen size, meaning that far-off details are cheaper to render then in standard ray casting. A more detailed breakdown of the rendering kernel is covered in section 7.1.3.

While rendering performance looks good, there are major fluctuations in the

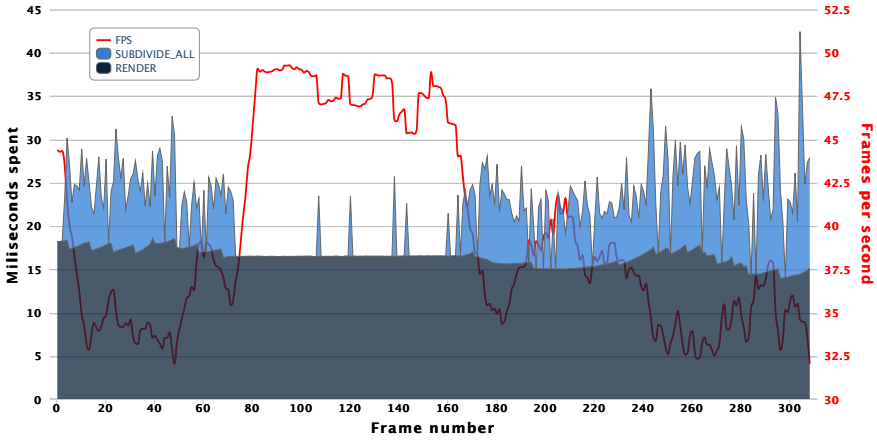


Figure 7.3: Global performance divided into rendering and subdivision.

frame rate. Looking at 7.3 it is clear that the cost of node subdivision varies a lot. In sequence **B** where the camera is only rotated slowly very few nodes needs to be subdivided, and the frame rate remains consistent. However sequence **A** and **C** reveals that cost of subdivision, due of movement, at times exceeds the cost of rendering to the point that the frame rate dips below 30. Section 7.1.4 will cover the individual cost of subdivision phases in detail.

7.1.3 Rendering Performance

Figure 7.4 shows a breakdown of the time spent in different segments of the rendering kernel. Out of the total 7 kernel segments, only 5 have a visible impact on the overall cost.

The segment `KERNEL_LOOKUP` covers octree descent described in algorithm 2. It is not surprising that this, is the most costly operation, as a descent from the root node must be started for each node visited in the octree. The fluctuation observed here can be attributed to the fact that when the camera moves closer to the terrain, the descent needs to be deeper in the octree to retrieve nodes at the lowest level.

`KERNEL_LOCALIZE` is a measurement of the time spent unpacking node localization codes and determining their bounding volumes. This appears to be the second most costly segment in the rendering kernel. An interesting fact is that localization could be omitted, as the bounding volume could easily be calculated

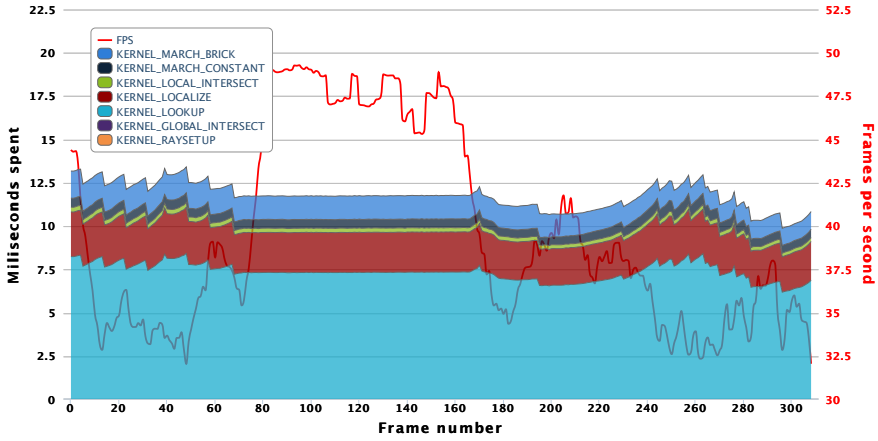


Figure 7.4: A breakdown of the cost of individual kernel segments during ray casting.

during the octree descent.

The final three segments: `KERNEL_LOCAL_INTERSECT`, `KERNEL_MARCH_CONSTANT` and `KERNEL_MARCH_BRICK` make up the remaining cost. They correspond to local ray cast within in the bounding volume calculated in `KERNEL_LOCALIZE`. A major factor of the performance in the rendering kernel, is that empty space can easily be skipped as it is stored in constant nodes, this is represented in `KERNEL_MARCH_CONSTANT`. Unsurprisingly `KERNEL_MARCH_BRICK` makes a considerable part of the total cost as this covers all texture sampling and lighting calculations. But without the concept of constant nodes this cost would have been much higher.

7.1.4 Subdivision Performance

Figure 7.5 shows a breakdown of the kernel calls and stream compaction operations used during the subdivision phase.

The first thing to notice is that the cost of the compaction operation on the node requests emitted from the ray caster, `SUBDIVIDE_REQUESTS_COMPACT`, remains steady through out all sequences of the animation.

During the subsequent usage buffer updates, only the compaction operations in `SUBDIVIDE_USAGE_MASK_COMPACT` have a visible impact. Usage mask updates

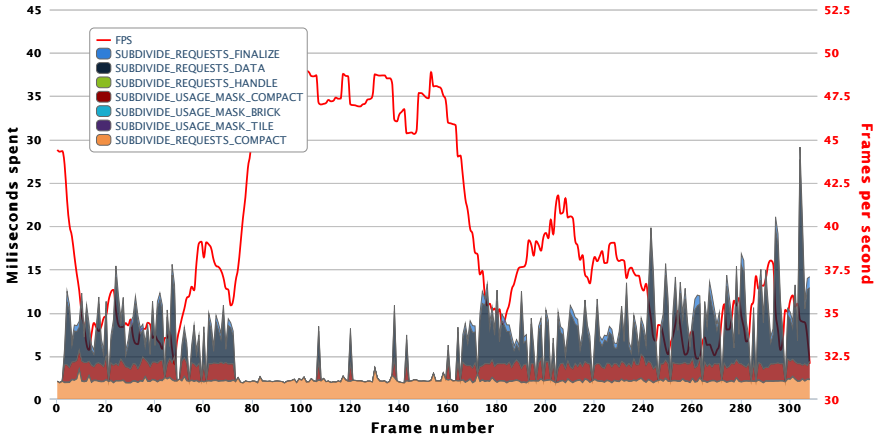


Figure 7.5: A breakdown of costs during the subdivision phase. The majority of the time is spent evaluating voxels in `SUBDIVIDE_REQUESTS_DATA`.

based on timestamp information, `SUBDIVIDE_USAGE_MASK_TILE` and `SUBDIVIDE_USAGE_MASK_BRICK`, have a negligible cost.

This leaves the three step method of node subdivision described in section 6.2.1. At this point it should be clear, that the primary reason for frame rate fluctuations can be traced to the voxel evaluation in `SUBDIVIDE_REQUESTS_DATA`. This is not surprising as the scene used in the profiling example relies on a fairly complex density function. For each voxel evaluated 35 calls to the function `noise3(x,y,z)`, which is an implementation of Perlin noise, are made in a combination with a host of addition, multiplication and trigonometric function calls.

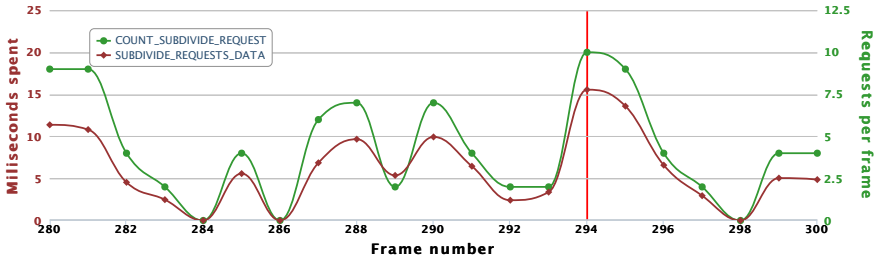


Figure 7.6: At frame 294 the number of node subdivision requests is 10 which translates to the evaluation of 327680 voxels.

The number of voxels evaluated at peak costs, such as frame 294 where 10

node subdivision requests are emitted as per figure 7.6, can be calculated as: `NODES_SUBDIVIDED` $\times 8 \times$ `BRICK_SIZE`^{3}. Using the parameters for the build used in this performance analysis this equals $10 \times 8 \times 16^3 = 327680$ voxels, or 11468800 calls to the fairly costly `noise3` function. The CUDA kernel handles these evaluations in 15.5 milliseconds, which if nothing else is quite impressive.

A Simple Density Function

To confirm that the frame fluctuations are indeed a result of the complexity of the density function, a more simple function is considered. Figure 7.7 shows a simple sphere, with some noise applied to its surface. This density function makes 2 calls to `noise3`.

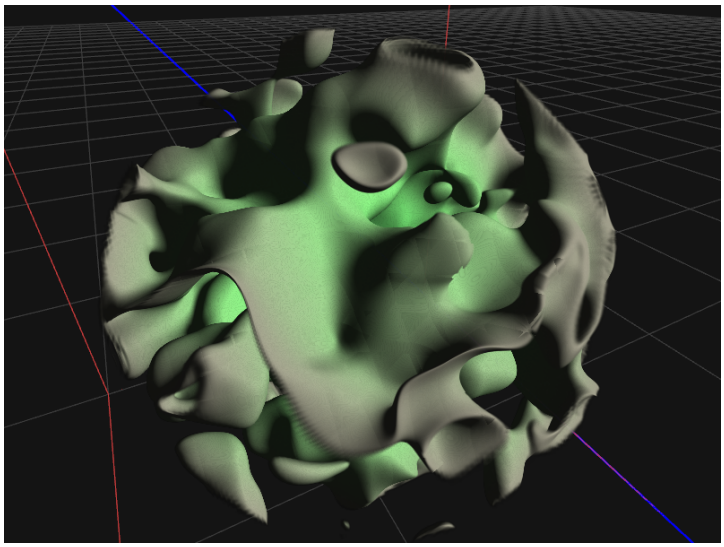


Figure 7.7: A simple density function used for performance analysis.

The analysis is not as extensive as the terrain scene, but merely an examination of the requests per frame and the resulting frame rate. Figure 7.8 shows the profiling data. The maximum observed subdivision requests are 64, which is the limit enforced by the configuration parameters of the implementation. Following the formula for the number of voxels evaluated, this translates to $64 \times 8 \times 16^3 = 2097152$. This is over 6 times the number of voxels evaluated roughly 1.9 ms, as opposed to 15.5 ms in the previous test scene. It should be noted that the increase in subdivisions also causes the subsequent finalization to increase in cost from 1.37 ms to 5 ms.

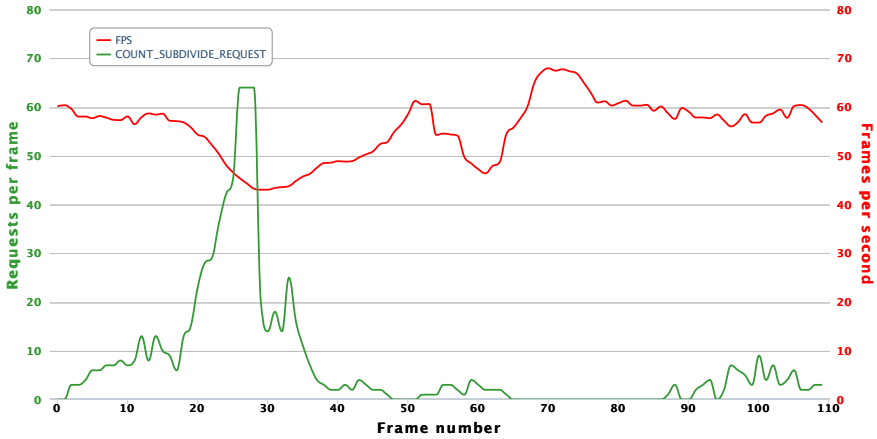


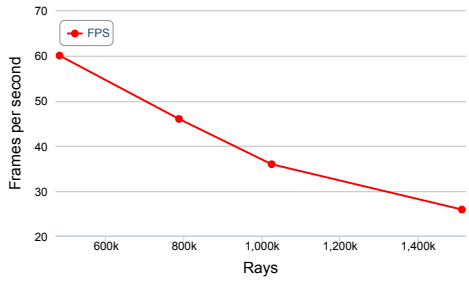
Figure 7.8: The frame rate for the simple test scene is shown together with the number of subdivision requests per frame.

As a side note the density function used in both scenes can arguable be described as unoptimized, and similar visual results could be obtained in a much less costly way. Following the example of [Ngu07] the volume noise could have been pre-calculated and cached in texture memory, which would greatly increase performance.

7.1.5 Image resolution

The performance analysis was done at a resolution of 800×600. But as the number of rays casted are dependent on the resolution of the final image, it makes sense to examine what happens with the frame rate when the resolution is increased.

Figure 7.9 shows the rendering performance at different common image resolutions. From this graph it becomes apparent that there is a linear relationship between the number of rays casted, and the frame rate. Even though 26 frames per second is not terrific at a resolution of 1600×900, the linear relationship is an indication that the rendering should scale with more powerful graphics hardware, as they would have more cores to distribute the work load.



Resolution	Rays	FPS
800×600	480000	60
1024×768	786432	46
1280×800	1024000	36
1600×900	1512000	26

Figure 7.9: A linear relationship exists between the image resolution (the number of rays) and the frame rate.

7.2 Invalidation

As described in section 6.2.1 the procedural evaluation relies on the three step method, which does not make the distinction between subdivision requests and data requests. This means that brick pointers can not be invalidated on their own, but requires the entire node tile to be invalidated in order to recycle the allocated bricks.

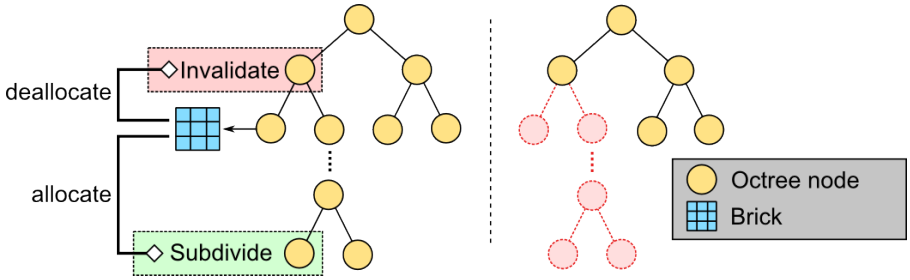


Figure 7.10: When a brick from a top level node is recyled, as part of a subdivsion, the entire node tile where the brick is referenced must be invalidated. This causes the subtree of the parent node to become invalid.

During scene exploration it becomes necessary to recycle the bricks, previously allocated for nodes elsewhere in the octree. This will typically be nodes close to the root, as their bricks are not used frequently when the camera is close to the model. But without the ability to have nulled brick pointers, the only option is to invalidate the entire node tile containing the brick. When a node tile somewhere in the top of the octree is invalidated, all children at the lower levels effectively

becomes invalidated as well. The result is that brick invalidation, caused ray casting requests, often causes an endless chain where the octree never reaches a depth beyond 2 and resulting in flickering images. The problem is illustrated in figure 7.10. A solution to the problem will be presented in section 8.1.

The cache invalidation design using attachments buffers, described in section 6.3, could still be used to simplify the invalidation process as long as recursive octree definitions are not allowed.

7.3 Memory Consumption

Table 7.1 presents an overview of the memory consumption of various pools and buffers in a typical configuration. This specific configuration is the same as used in the performance analysis in section 7.1. The brick pool is stored in a volume texture of dimension 512^3 with a brick resolution of 16^3 giving a total of 32768 bricks.

Name	Size	Elements	Memory	
			MegaBytes	Percentage
Node Pool	64 bit	2097152	16	1.49
Node Localization Data	40 bit	2097152	10	0.93
Node Tile Timestamp	32 bit	262144	1	0.09
Node Tile Usage	32 bit	262144	1	0.09
Node Usage Attachment	32 bit	2097152	8	0.74
Brick Pool	64 bit	512^3	1024	95.84
Brick Timestamp	32 bit	32768	0.125	0.01
Brick Usage	32 bit	32768	0.125	0.01
Brick Attachment	32 bit	32768	0.125	0.01
Raymarcher Request	32 bit	2097152	8	0.74
Raymarcher Data Requests	32 bit	512	0.001	0.00
			1068.376	

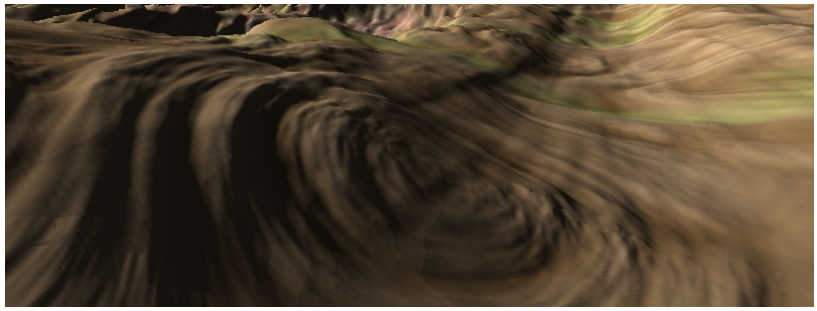
Table 7.1: An overview of the memory consumption used by the various pools and buffers in the implementation. The brick pool is assumed to be stored in a volume texture of dimension 512^3 with a brick resolution of 16^3.

Unsurprisingly, the brick pool is by far the largest buffer accounting almost 96%

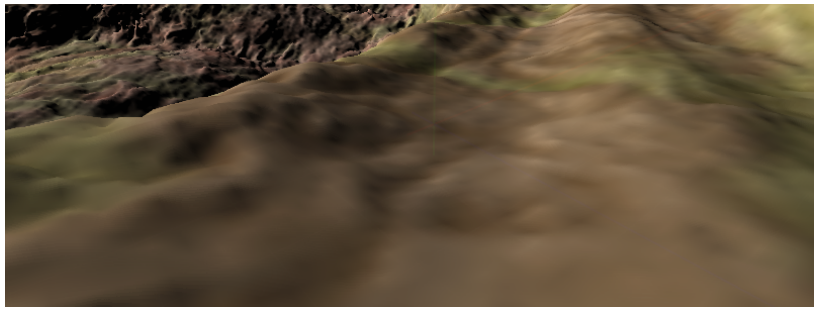
of the total memory used. The size of the pool could be reduced by fixing the brick invalidation described in section 7.2.

7.3.1 Volume Resolution

As stated in chapter 4, one reason for using a sparse voxel octree implementation, is the ability to represent volume data at a much higher resolution, than possible if using a volume texture as data storage. Figure 7.11 shows two images rendered with a sparse voxel octree implementation as described in chapter 6, and a ray caster using a volume texture implemented based on algorithm 1.



(a) Image rendered at 800×600 by a CUDA raycasting using a sparse voxel octree implementation at around 60 fps.



(b) Image rendered at 800×600 by a CUDA raycaster using a static volume texture as source for volume data at around 28 fps.

Figure 7.11: A comparison between the data resolution: (a) sparse voxel octree implementation and (b) a implementation using static volume texture. The raytracing parameters for both implementations have been tweaked to create similar conditions for step length.

The images have been cropped to highlight the resolution of the data in the foreground. Both implementations uses a volume texture, with a resolution of 512^3, as storage for volume data. However the octree version, which in this case has a maximum depth of 8, has allocated the majority of the bricks to represent the details closer to the camera. A very rough estimate[3] of the resolution needed to achieve similar results in the standard ray caster would put the size of the volume texture at around 3000^3, or nearly 100 gigabytes (assuming the volume is represented using a RBGA8 format). Even though this is a crude simplification it does give an idea of the order of magnitude achieved with sparse voxel octrees.

7.4 Image Quality

The volume data in the octree is constructed from the root node and down through the tree, as described in chapter 6. This means that a brick associated with a node at $depth = D$, is normally constructed before a brick at $depth = D+1$. The consequence is that low detail bricks can not be created by downsampling from higher detail bricks. They are instead evaluated from the same density

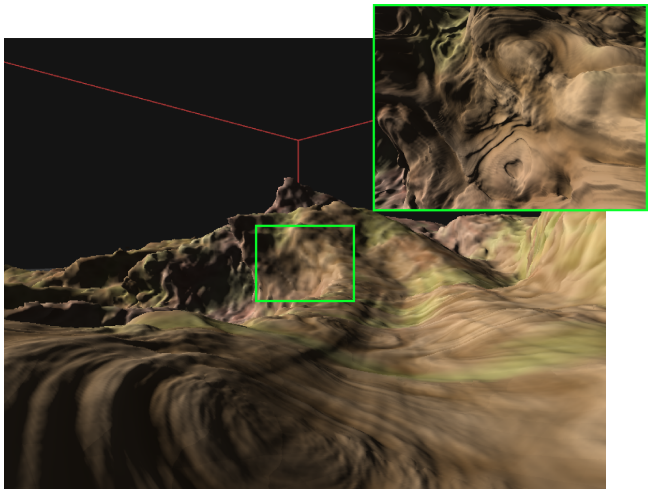


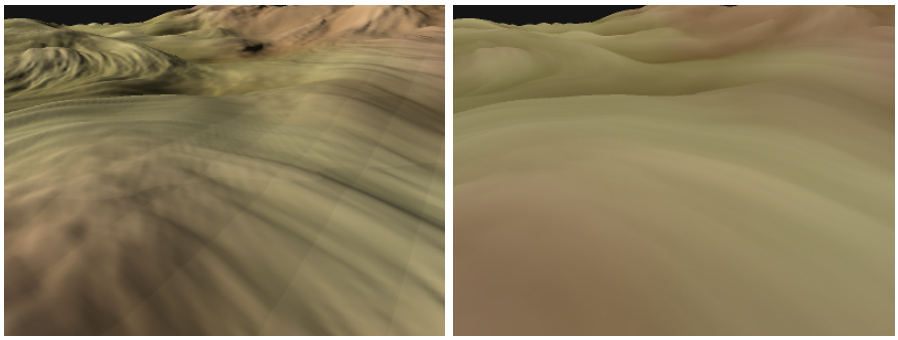
Figure 7.12: A test scene rendered at two different zoom levels. The bricks at a high level in the octree fails to capture the details of the lower levels, due to the lack of downsampling in the three step subdivision method.

[3] This estimate based on the length of a single voxel at a depth of 8 in the octree, with a brick resolution of 16.

function, that may contain high frequency noise which causes aliasing. This results in an inaccurate description of the contents of bricks, that fails to capture subtle details of volume data at a distance. An example of this can be seen in figure 7.12, showing a rendered terrain at two different zoom levels.

GigaVoxels does not normally suffer from this problem, as it uses pre-filtered volume data, where the octree is constructed by downsampling the original volume data as a pre-processing step. Something similar could be done in this implementation if the brick evaluation was done using downsampling. This would however strain the already costly data requests step, so it comes down to a performance/quality trade off. Alternatively, the frequency of the density could be adjusted by choosing the number of noise octaves according on the node depth. But this could potentially lead to problems where a parent node is characterized as solid or empty, and therefore terminal, even though its child nodes would not have been solid or empty. In section 8.2 an alternative possible solution is presented.

Finally another noticeable artifact, related to brick marching, is shown in figure 7.13. This artifact is caused by a bug in the implementation, that occurs when mapping local ray casts into the reduced node bounding volume to account for brick borders, but only affects the normal estimation using central differences. While this could be solved with more work it should be mentioned, because it is very apparent in the screenshots found in this chapter.



(a) With lighting **(b)** Without lighting

Figure 7.13: Lighting artifacts caused by incorrect mapping of local position, within node bounding boxes, to brick coordinates when accounting for brick borders.

Chapter 8

Future Work

This chapter describes a possible solution to the brick invalidation problem discussed in section 7.2, as well as suggest some extensions to the method presented in this thesis to improve its capability to create more varied worlds, improved lighting quality, and integration with existing technology.

8.1 Invalidation

As described in section 7.2, the three step method for node subdivision does not make the distinction between subdivision requests and data requests. This causes problems with brick invalidation, because they can not be invalidated on their own, and requires the full node tile to be invalidated.

This problem could be fixed by reintroducing the distinction between subdivision requests and data requests by having the notion of invalidated brick pointers. If a request from the ray caster targets a node that has children, but an invalidated brick pointer, a data request for a new brick can be sent to the subsequent data producer. If the node on the other hand has no children, it should be processed as normal.

8.2 Varied, Modifiable and Infinite Worlds

The method presented in this thesis could be used to create highly detailed landscapes in interactively simulations, such as computer games. The expressiveness of voxel grids makes it possible to describe environments that contains elements such as caves, tunnels and cliff overhangs that normal 2D methods do not easily describe. It would be interesting to investigate how the technique could be used to create a more varied world. The main problem would be the shear complexity of a single density function describing such a world.

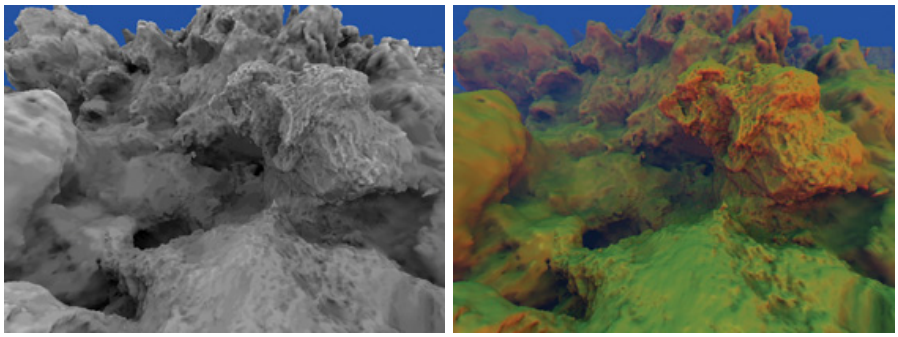
Variation could be introduced by dividing the world into areas, described by their location in the world which could be determined procedurally. Each area could be assigned a specific density function giving each area a unique look. When evaluating voxels, a blended weighting of the density functions associated with nearby areas, could produce interesting results.

However, a blended weighting does not mitigate the computational cost of the voxel evaluation described in 7.1.4. It also does not counter the aliasing problems caused by the lack of downsampling of voxel data described in section 7.4. Both problems could be addressed by procedurally evaluating a coarse pre-filtered version of the world, and storing it on disc and introduce details in the voxel data when it is streamed to the GPU. This would also allow for run-time modifications of the coarse volume data, which could then be written back and stored on disc and used the next time the application is started. If the world was stored in blocks it would be possible to store only the blocks that were modified during run-time, and procedurally generating the rest again when the application is started. This would help to keep storage requirements low.

Finally it would be possible to create an infinite world. The pointer based nature of the octree would make it possible to move the bounding box of the volume, and invalidate the octree nodes that are no longer contained within the new bounding box, by moving child pointers around in the top of the tree. This would also require updating the localization buffer, to make sure that the choice vectors correspond to the new structure. This must be done whenever the eye position has moved some distance, and the visible world no longer fits inside the current bounding volume.

8.3 Ambient Occlusion

The lighting quality could be improved by using a simple global method like ambient occlusion, which is a crude approximation to real global illumination, but tend to produce a nice soft look, see figure 8.1.



(a) (b)

Figure 8.1: Ambient occlusion estimates the visibility of a point on a surface. In (a) the ambient occlusion is shown, and (b) shows the results when combined with a local illumination model.
Source: [Ngu07]

Ambient occlusion approximates the amount of ambient light reaching a point on a surface, by approximating the visibility of the background over the hemisphere of the surface. In ray casting this translates to casting a number of secondary rays, into the hemisphere, to determine the visibility, see figure 8.2. [Cra11] shows that cone tracing can be used to reduce the number of secondary rays and use volume mipmapping to efficiently compute the visibilty factor.

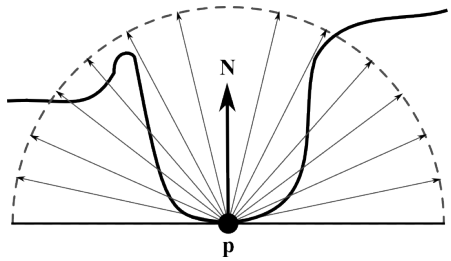


Figure 8.2: Ambient occlusion approximates the visibility of a point p by casting rays into the hemisphere over the surface described by the normal N.

8.4 Combined with Triangle Rasterisation

The rigid structure of voxel based representations makes them very suitable for static models, such as terrains and buildings. But if deformations are needed, when for example character models are animated using skeleton animation and skinning, triangle rasterisation methods tend to be more flexible and mature.

The method presented in this thesis could be used in combination with triangle rasterisation, by including depth information when the pixel buffer is written. This information could be written to the depth buffer before triangle models are rendered to the screen. In the OpenGL Shading Language (GLSL) this could be achieved by writing to `gl_FragDepth`. The process could also be reversed by using depth information from triangle rasterisation for early ray termination in the ray caster. In any case, care must be taken in areas were transparent geometry from the first rendering occludes geometry from the second.

Finally, it is shown in [CR12] that it is possible to leverage the existing rasterisation pipeline to voxelize triangle models in real-time. In [CNS+11] it is shown that voxelization may be integrated into the GigaVoxels engine to incorporate animated triangle based models.

Chapter 9

Conclusion

It has been shown that large volume data sets can be compactly represented in GPU memory using sparse voxel octrees. Benefits such as compression of homogeneous regions and view dependent updating of the tree structure have been discussed. A possible way of implementing sparse voxel octrees, inspired by the GigaVoxles engine, exploiting the parallel nature of consumer graphics hardware has been presented. This includes the central octree data structures, auxiliary buffers and related algorithms.

The physical foundation leading to volume ray casting has been presented. It has been shown that volume mipmapping and cone tracing can improve the standard ray casting method, when the volume data is compressed using sparse voxel octrees. An in-depth performance study of the individual step related to volume ray casting, using cone tracing, has been carried out.

Two extensions to the GigaVoxels engine, aimed at procedural content generation, has been examined: A three step method for subdividing octree nodes and procedurally evaluating their contents, and a simplification to the node and brick pool invalidation. The performance of the three step method has been examined, and a bottleneck was identified in the data request step, when complex density functions are used. Problems related to invalidation caused by a missing distinction between subdivision- and data-requests in the three step method were identified, and a possible solution has been presented.

Appendix A

Appendix

The source code is avaible on Github at:
https://github.com/udsholt/thesis-cuda-sparse-voxel-octree.git

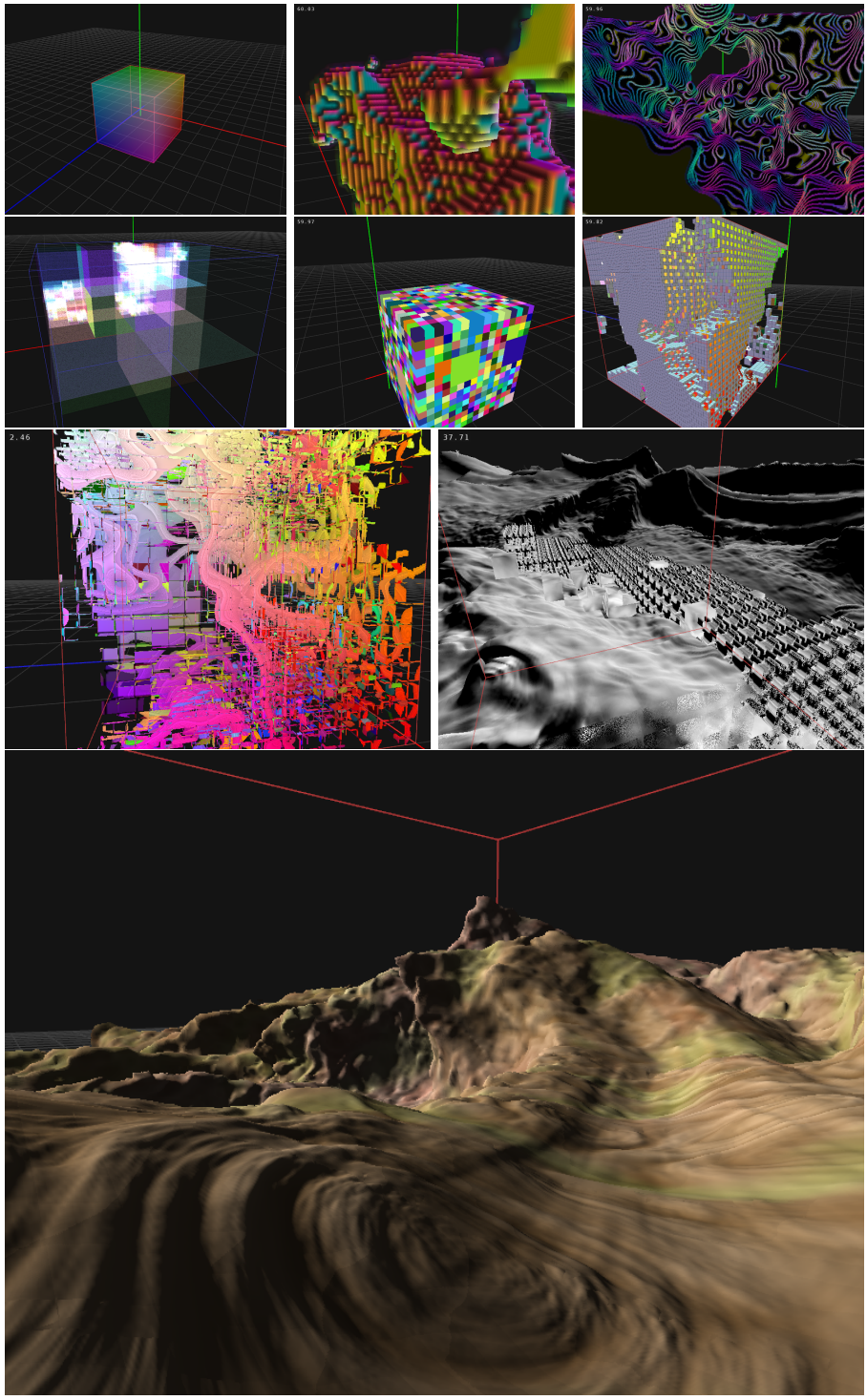


Figure A.1: The long road to real-time rendering of procedurally generated volumetric models...

Bibliography

[CNS+11] Cyril Crassin, Fabrice Neyret, Miguel Sainz, Simon Green, and Elmar Eisemann. Interactive indirect illumination using voxel cone tracing. In *Computer Graphics Forum*, volume 30, pages 1921–1930. Wiley Online Library, 2011.

[Cor09] NVIDIA Corperation. NVIDIA Fermi Compute Architecture Whitepaper. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf, 2009. [Online; last accessed 13-July-2013].

[Cor12] NVIDIA Corperation. CUDA C PROGRAMMING GUIDE. http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf, 2012. [Online; last accessed 13-July-2013].

[CR12] Patrick Cozzi and Christophe Riccio. *OpenGL Insights*. AK Peters Limited, 2012.

[Cra11] Cyril Crassin. Gigavoxels: A voxel-based rendering pipeline for efficient exploration of large and detailed scenes, July 2011. English and web-optimized version.

[EHK+06] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-Time Volume Graphics*. A K Peters Ltd., first edition, 2006.

[Eng10] Wolfgang Engel. Gpu pro, 2010.

[HSHH07] Daniel Reiter Horn, Jeremy Sugerman, Mike Houston, and Pat Hanrahan. Interactive kd tree gpu raytracing. In *Proceedings of the 2007*

symposium on Interactive 3D graphics and games, pages 167–174. ACM, 2007.

[JC06] Gunnar Johansson and Hamish Carr. Accelerating marching cubes with graphics hardware. In *Proceedings of the 2006 conference of the Center for Advanced Studies on Collaborative research*, page 39. IBM Corp., 2006.

[LK11] Samuli Laine and Tero Karras. Efficient sparse voxel octrees. *Visualization and Computer Graphics, IEEE Transactions on*, 17(8):1048–1059, 2011.

[MHH08] Tomas Möller, Eric Haines, and Naty Hoffman. *Real-time rendering*. AK Peters Ltd, 2008.

[MPP+94] F Kenton Musgrave, Darwyn Peachey, Ken Perlin, Steven Worley, and David S Ebert. *Texturing and modeling: a procedural approach*. Academic Press Professional, Inc., 1994.

[Ngu07] Hubert Nguyen. *Gpu gems 3*. Addison-Wesley Professional, first edition, 2007.

[Rud09] Nicholas Eugene Rudzicz. *Arda: A Framework for Procedural Video Game Content Generation*. PhD thesis, McGill University, 2009.

[SK10] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.