Volume rendering

Marcus Jonsson

October 12, 2005
Master's Thesis in Computing Science, 20 credits
Supervisor at CS-UmU: Berit Kvernes
Examiner: Per Lindström

Umeå University
Department of Computing Science
SE-901 87 UMEÅ
SWEDEN

Abstract

Volume rendering has become a large part of scientific visualization during the last twenty years. Three-dimensional scalar fields are generated within a wide array of scientific areas, most commonly today by scanners in the medical field, but geophysics, metrology and fluid flow simulations are a few example areas which generate this type of data. Visualization of data is important to quickly and accurately gain insight to enormous amounts of information. This type of data cannot be rendered with conventional rendering techniques, which is why volume rendering has created its own field within scientific visualization.

The purpose of this thesis is to introduce the reader to the field of volume rendering with a comprehensive overview, presenting methods from past and present. Interactive volume rendering has traditionally been an area where users and developers must have access to workstations with powerful special purpose hardware. With ever increasing computational power within the consumer computer field, driven on by interactive entertainment such as computer games, interactive volume rendering is now available at a low cost.

Given the relatively new possibility of hardware acceleration for volume rendering, this thesis will make comparisons between implementations of software and hardware techniques. As speed and flexibility is increasing rapidly within the field of consumer level graphics cards, hardware acceleration using consumer hardware is quickly becoming the fastest growing research field within volume rendering.

Acknowledgements

I would like to thank my supervisor Berit Kvernes for the original idea for this thesis as well as her technical support.

I also want to thank my family and friends for the great support during the time I worked on this thesis, it certainly made it so much more enjoyable.

Contents

1 Introduction **1**
1.1 Background . . . 1
1.2 Objective . . . 2
1.3 Structure of thesis . . . 2

2 Volumetric data **3**
2.1 Data acquisition . . . 4
2.2 Reconstruction and filtering . . . 5
2.2.1 Filtering . . . 6

3 Basic optic theory for volume rendering **7**
3.1 Optical models . . . 7
3.1.1 Absorption . . . 8
3.1.2 Emission . . . 9
3.1.3 Emission with absorption . . . 9
3.1.4 Emission-absorption with shadows . . . 9
3.1.5 Discretization of the volume rendering integral . . . 10
3.2 Illumination model . . . 10
3.3 Gradient computation . . . 11

4 Classification **13**
4.1 Pre and post classification . . . 13
4.1.1 Opacity weighted colour interpolation . . . 14
4.2 Transfer functions . . . 15
4.2.1 One-dimensional transfer functions . . . 15
4.2.2 Multidimensional transfer functions . . . 16
4.3 Pre-integrated classification . . . 17
4.4 Compositing . . . 18

5 Software volume rendering methods **19**
5.1 Image order . . . 19
5.2 Object order . . . 19

5.3 Domain based 20
5.4 Generic render pipeline 20
5.5 Raycasting 21
5.5.1 Levoy's raycasting 21
5.5.2 Optimization techniques 22
5.5.3 Early ray termination 22
5.5.4 Empty space skipping 23
5.5.5 Progressive refinement 23
5.6 Frequency domain rendering 23
5.7 Splatting 24
5.8 Shear-Warp 25

6 Hardware acceleration techniques 29
6.1 Consumer level hardware 29
6.1.1 How the graphics hardware works 29
6.1.2 Graphics hardware models 30
6.1.3 Graphics hardware issues 30
6.2 Techniques 31
6.2.1 2D texture slicing 31
6.2.2 3D texture slicing 32
6.2.3 Problems with slicing 32
6.3 Classification 33
6.4 Lighting 33
6.4.1 Per-pixel 33
6.4.2 Light maps 34
6.5 The future of VR hardware acceleration 34
6.5.1 Hardware acceleration of complex optical models 35
6.5.2 Hardware raycasting 36
6.5.3 Hardware raycasting with pre-integration 37
6.5.4 Splatting 37
6.5.5 Frequency domain 38

7 Implementation 39
7.1 Implementation details 39
7.2 Software raycasting 39
7.2.1 Pre-classification renderer 39
7.2.2 Post-classification 40
7.2.3 Computational complexity 40
7.3 Hardware Implementation 41
7.3.1 2D texture slicing 41
7.3.2 3D texture slicing 42
7.3.3 CG - high level shading language 43

8 Results **45**
8.1 Image quality 45
8.1.1 Quality factors 46
8.1.2 Raycasting 46
8.1.3 Hardware 47
8.2 Performance 47
8.2.1 Raycasting 47
8.2.2 Hardware 49

9 Conclusions **51**

References **53**

List of Figures

2.1 On the right is a typical data volume consisting of structured, rectilinear data. The figures to the left shows the two interpretations of a voxel with the lower being the more common representation. 4
2.2 A discrete sampling (shown on the left) is reconstructed into a continuous signal. 5

3.1 In volume rendering, attenuation and light scattered in the eye (ray) direction is calculated per step along the ray and then integrated to form a single brightness value per screen pixel. 8
3.2 The gradient of the voxel in the centre can be computed by central difference using the six closest neighbouring voxels. 11

4.1 The new pipeline feeds the separately classified opacities over to the shading path before the reconstruction stage. 14
4.2 Four voxels with varying opacity and colour. In Levoy's implementation, voxel D contributes to samples, while Wittenbrink's does not. 15

5.1 These are the common steps for any pipeline, implementation details and order of execution is specific however. 20
5.2 The standard pre-classification raycasting pipeline with separated paths for classification and shading. 22
5.3 The simple case where the rays are perpendicular to the volume is shown in the figure to the left. When the view is rotated, the slices are sheared (center). Alias-free perspective projection is possible by scaling the slices (right). 26

List of Tables

8.1 Results from the engine dataset, $256^2 * 109$, set with high transparency and with early ray termination enabled. 48
8.2 The engine dataset, $256^2 * 109$, early ray termination is enabled with more opaque transfer function settings. 48
8.3 Once again the engine dataset, $256^2 * 109$, here early ray termination is disabled, set with high transparency. 48
8.4 The engine dataset, $256^2 * 109$, with no shading. 49
8.5 The engine dataset, $256^2 * 109$, this time with shading turned on. 50

Chapter 1

Introduction

Visualization of three-dimensional scalar fields, commonly called volume rendering, is an area that have seen developments since the early 1980s. The nature of the data is such that the interior can be rendered as well as the surfaces bounding the object. Volume rendering techniques concentrate on visualizing internal features of choice, and surfaces, at the same time.

Volume rendering is a large field within scientific visualization. While most of the efforts have been geared towards medical imaging, some of the methods can just as well be used for similarly constructed scalar fields obtained from other sources.

Many different basic theories from a wide array of engineering fields and physics are combined under the name volume rendering. For example, medical data is collected with magnetic resonance and then later reconstructed by using sampling theory. A basic light transport algorithm is needed for accurate lighting, which is realized by employing basic optics and physics. Samples from the volume are finalized on screen by two-dimensional image compositing algorithms. Other areas include frequency analysis, linear algebra for transformations and scientific computing for much of the calculations.

1.1 Background

Conventional surface rendering methods are not well suited for volume rendering, researchers realized this during the early 80's which led to the development throughout the 80's and 90's of a number of new techniques. The benchmark approach regarding picture quality, and the one technique that defined modern volume rendering, is Raycasting, a method whose roots in its modern incarnations can be traced back to Blinn[4], and then refined by Levoy[18]. As the name suggests, it works by casting rays into a volume of data and then sample each ray along intervals. Its main advantage is that no intermediate geometry is produced during the process, thus minimizing aliasing. Approaches taking the opposite route by exploiting intermediate geometry for increased rendering speed include Shear-warp[17].

The last few years has seen revitalization within the field with the arrival of programmable consumer graphics cards. The majority of software based approaches can not produce interactive frame rates, but by using consumer graphics cards, most implementations can achieve good frame rates. This cheap and fast hardware has generated a number of new techniques, most notably [9][38][15]. New and groundbreaking techniques are currently being developed and many have been released continuously during

the past year. This trend will most likely continue as the hardware matures.

1.2 Objective

This thesis will concentrate on giving an overview of the volume rendering field. As well as presenting the theoretical background, showing possibilities and limitations, much emphasis are put on the main techniques that have dominated the area for the past ten years. The latest interactive techniques involving consumer graphics hardware is also a large part of this thesis.

1.3 Structure of thesis

Chapter 2 introduces the basics of volume rendering, what the role of the data is, how it has formed the techniques and why acquisition methods have an impact on the choice of rendering technique. It also probes deeper into how the data must be treated, the reconstruction phase is very important. Chapter 3 covers theoretical issues with the optical models that are used by volume rendering, they are explained in detail. The common shading model is explained as well.

Chapter 4 discusses the core of volume rendering, the classification stage. The classification procedure, why it is so important and when it should be performed are the primary goals of this section. The role of the transfer function and how it can affect the image quality, primarily in the reconstruction phase.

Chapter 5 focuses on the most popular software techniques that all were developed around the beginning of the 1990s. The three big paradigms that the methods are using are covered as well. This is a survey that should give a good understanding of what strengths and weaknesses each technique possesses. The developments that have been done within each field are described also.

Chapter 6 is an in-depth look at how general purpose graphics hardware can be used to accelerate volume rendering. The traditional software techniques have all been converted to take advantage of the hardware rendering, the changes that had to be done and what they lead to are explained.

Chapter 7 describes the implementation part of this thesis. It deals mostly with the issues that surround the implementations since the actual techniques are explained in chapter 5 and 6.

Chapter 8 analyses and describes the results of the implementation. The factors that influence image quality and how it can be judged is a large part of this section. The performance of each technique is compared and analysed likewise.

Chapter 9 is a wrap up of the thesis and a look ahead to the future of volume rendering.

Chapter 2

Volumetric data

Volume data is in its simplest form a quadruple (x,y,z,w) where x, y, z is a scalar describing a position in three-dimensional space and w is the actual data value, the quadruple is often called a voxel. A volume, or dataset as it is sometimes called, is hence a collection of volume data generated from some sort of measurement apparatus or simulation. A voxel can have an arbitrary number of data values associated with it, the dataset is said to be multivariate when more than one data value is used. Multivariate data is common with data flow simulations or when the result of a number of data sources is combined.

The internal structure of a dataset is very important, Speray and Kennon[42] suggested a number of data structure categorisations in 1990 that are by and large used today, and was summarized by Watt[47] in the following way:

- Cartesian: The data elements are structured, and arranged in a cubic grid
- Regular: As Cartesian, but the grid can be rectangular as well.
- Rectilinear: As Regular, with the acceptance of non-uniformly sized grids
- Structured / Curvilinear: Non-linear grids. Possible for the grid to wrap around objects.
- Block structured: Several structured systems are allowed.
- Unstructured: Unstructured data, data points can be anywhere within the volume.
- Hybrid: A combination of unstructured and structured data.

The most important distinction is between structured and unstructured data since they often require different visualisation methods, for example, shear-warp is a common technique used for structured data while projected tetrahedras is used for unstructured data. One should keep in mind also that these categorisations are not used to the letter by researchers.

The big driving force within the volume rendering field is without a doubt medical imaging. Medical scanners such as MR (magnetic resonance) measure three-dimensional space as a structured three-dimensional grid, and this has led researchers to focus mostly on volume rendering techniques based on structured, rectilinear grids. The grids are often called slices.

It should be noted that voxels can be defined in two different ways:

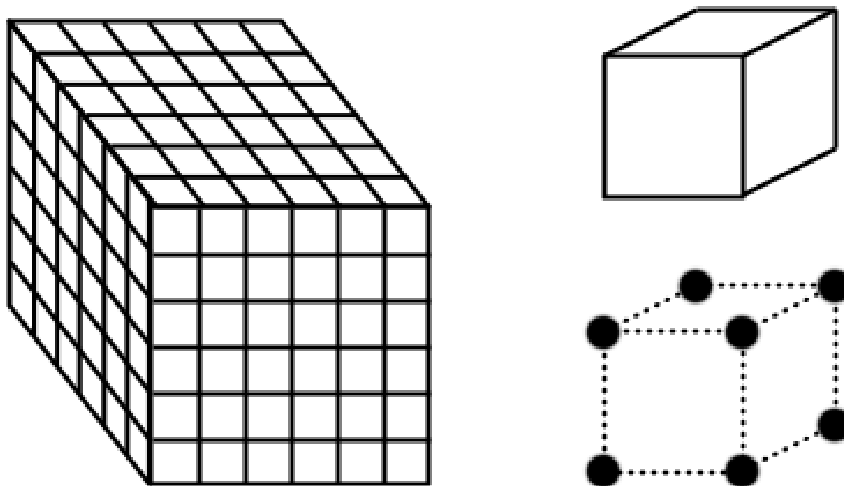

Figure 2.1: On the right is a typical data volume consisting of structured, rectilinear data. The figures to the left shows the two interpretations of a voxel with the lower being the more common representation.

– It can be considered to be a cube formed by four neighbouring grid points in two different slices and the whole voxel is covered by one single data value interpolated from all eight grid points. This definition is sometimes used in older publications.

– More commonly in the modern volume rendering literature as well as in this thesis, is that a voxel is a three-dimensional coordinate paired with a data value. See figure 2.1.

2.1 Data acquisition

Volumetric data is generated from many sources, and it is a good idea to at least know the basics behind how they were acquisitioned. Some methods suffer from shortcomings that must be taken into account in the reconstruction phase.

Computed tomography (CT) is the oldest of the three-dimensional medical scanners. There are a few variants of it, such as positron emission tomography (PET) and single-photon emission computerized tomography (SPECT). With CT, x-ray images are taken from different angles around the test subject. The data from the images are then combined and reconstructed slice by slice to three-dimensional data using radon and fourier transforms. The x-rays are very good at finding boundaries between bones and tissue, but are not very good at finding boundaries within tissue. The resolution is not as good as MRI data, and it contains more noise.

Magnetic Resonance Imaging (MRI) is a modern medical scan method. The MRI scanner works by measuring signals produced by protons within hydrogen atoms. The apparatus consists of a large, circular magnet surrounding the object, usually a patient, and a radio wave emitter. To make the protons send a signal, a magnetic field needs to

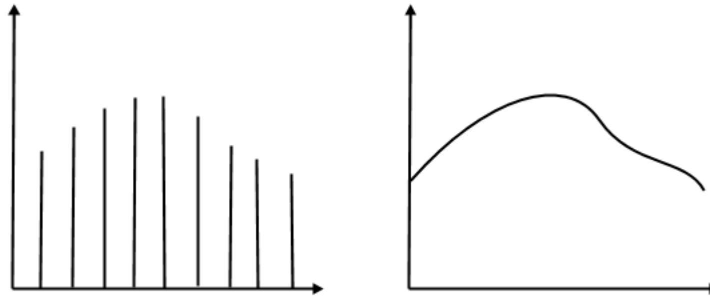

Figure 2.2: A discrete sampling (shown on the left) is reconstructed into a continuous signal.

be generated, and it has to be strong enough to align protons within hydrogen atoms. Radio waves with a certain frequency are then used to spin the protons. This makes them emit a sufficiently strong signal to be recorded a receiver. The datasets originating from a MRI scanner require less data preparation and are generally of a higher quality than other types of medical data.

Other sources include satellite (radar) and weather station data which can be used to render animations of complete weather systems. Seismic information gathered by geophones or similar techniques can be reconstructed to three-dimensional data using several sources. The renderings of such data are for instance used to guide geophysical prospecting.

2.2 Reconstruction and filtering

When the data acquisition stage is done, the result is a number of discrete data points, voxels, forming a volume. The next step to visualize the volume is to apply an algorithm that transforms the discrete values into an image on the screen. But before we do this, a few theoretical issues regarding discrete signals must be considered.

When the volume was created, it was done by *sampling* of the original signal, either by measurements or by sampling of a simulation. In essence, the original continuous signal has been converted into a discrete signal, see figure 2.2. To visualize this sampled volume, the signal has to be *reconstructed* back into its original continuous form. The goal of this section is to show how properties of the data, such as how it was acquisitioned, affect the rendering and the final picture.

A prefect reconstruction of the original signal is possible to achieve according to the sampling theorem[10], but only under certain conditions. First, the signal must be *band limited*, and secondly, the signal must be sampled at more than twice that of the maximum frequency, called the *Nyquist frequency*, and at even intervals. Band limitation means that the signal carries no frequencies above a certain limit, for example, no infinite frequencies are allowed.

There are a few problems with these limitations for a perfect reconstruction. The three-dimensional data produced by measuring or simulations almost always contain infinite frequencies[1], and thus fails one of the sampling theorem requirements. The

infinite frequencies are a result of sharp boundaries, between materials or at edges, which creates discontinuations in the signal.

Furthermore, medical scanners such as MRI machines does not take band limiting into account[18]. CT scanners have anisotropic sensitivity in the spatial domain [18], which produces striping in the rendered image. This means that it is almost guaranteed that a perfect reconstruction from real-world volumetric data is not possible, but it can theoretically be done if the data acquisition is done correctly.

Reconstructing data that is not band limited will result in aliasing effects in the final image. This is an important property to be aware of, since aliasing of this kind will be visible regardless of the rendering method that is used.

2.2.1 Filtering

To reduce the aforementioned aliasing during the reconstruction, a number of filtering and anti-aliasing techniques can be employed. Reconstruction of the discrete volume requires a filter, and the filter that must be used for a perfect reconstruction is the sinc filter. Although the sinc filter does have the ability to perfectly reconstruct a signal that meets the requirements of the sampling theorem, the filter itself is difficult to make practical use of. The sinc filter extends to both negative and positive infinity and carries negative amplitude at times as well.

Other common filters are the box filter (also called, nearest neighbour) and the tent filter (also called linear filter). These filters including the sinc filter are all lowpass filters, which means that they filter out all frequencies above a certain limit set by the filter, thus provides smoothing to the signal.

Using the tent filter is usually considered to provide a satisfactory result especially for techniques that needs quick computation as well as good picture quality. This filter is implemented by doing *trilinear interpolation* of the neighbouring samples. Higher order filters such as the *b-spline* filter, or *Blackman windowed sinc* can produce a very good picture quality with few aliasing artefacts resulting from signal reconstruction, but at the price of long evaluation times.

Anti-aliasing, as the name suggests, is a method that is used to reduce aliasing artefacts. A common method in volume rendering is to run a filtering pass over the volume as a pre-processing step to remove all high frequencies. There is a drawback to this approach though, fine detail is lost and blurring introduced instead.

Chapter 3

Basic optic theory for volume rendering

As with any rendering model, a mathematical model must first be defined in order to be able to create an image with volume rendering. The data can be densitity values for instance, and these need to somehow be mapped to optical properties. Depending on how realistic the final image must be, the model can include a number of physical properties, such as emission, absorption, reflection, scattering of light and occlusion of light. Today, the most common optical model used in volume rendering was developed by Blinn[4], and was later improved by Kajiya[12]. Blinn initially developed the model so that he could render objects defined as volume densities with photo realism. He was specifically interested in rendering clouds and the rings of Saturn.

One of the unique features of direct volume rendering is the lack of actual surfaces. This is completely different to traditional computer graphics where polygonal or other types of surfaces with material properties are used. Instead of surfaces, Blinn regarded the density values within the data volume as small spherical particles. The particles scatter and attenuate light depending on their density.

3.1 Optical models

Before we continue, we must first describe the terminology and units that are used to describe optical properties. The intensity of light at a point in space is determined by its radiance, which has a frequency and a direction. Scattering describes the change in direction and frequency for the radiance. Absorption of light is the process where energy of light (radiance) is turned into heat.

$$Absorption = pure\ absorption + scattering$$

Emission of light is where light is introduced to a scene from an external source.

$$Emission = source + scattering$$

Both absorption and emission have a scattering term, and these terms cause problems that are very difficult to solve as we will see.

Blinn made a number of assumptions in his model for volume rendering, most of them are made to simplify the equations and thus reduce the computational complexity.

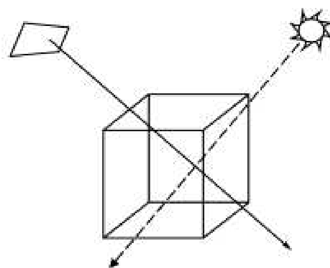

Figure 3.1: In volume rendering, attenuation and light scattered in the eye (ray) direction is calculated per step along the ray and then integrated to form a single brightness value per screen pixel.

The most important assumption is that the volume is in a low albedo environment and is consequently called "the low albedo assumption". The albedo of a particle is a measurement of how much it reflects incoming light, which means that a high albedo environment is going to scatter light a lot between particles.

The low albedo assumption means that only one scattering is used in the calculation, and that is the scattering along the viewing ray. For most purposes, this is a fair assumption, but if visual effects like shadows are needed, then the line from the light must be evaluated as well.

Calculating the scattering along the line from the light to a particle inside the volume is computationally expensive unfortunately, which is why Blinn's model does not take multiple scattering into account. Kajiya[12] shows how the scattering equation can be solved in a high albedo environment, but generally the computational complexity is too high for it to be reasonable to evaluate however (at least if speed is of any importance).

By not evaluating the scattering of light throughout the volume, Blinn's volume rendering lighting model is essentially an emission-absorption model. The emission-part of the equation determines the intensity and colour of a particle inside the volume while the absorption decides the opacity. How the emission and absorption works individually and together to form the general volume rendering integral will be shown next.

3.1.1 Absorption

In the special case of particles inside the volume (voxels) having the characteristic of absorption only, with no emission or scattering, the following differential equation holds

$$\frac{dI}{ds} = -\tau(s)I(s) \tag{3.1}$$

where s is the distance along the ray, $I(s)$ is the light intensity at s, $\tau(s)$ is the light extinction coefficient (or opacity as it is often called) at s. The solution to this equation is

$$I(s) = I_0 \exp^{-\int_0^s \tau(s)\,dt} \tag{3.2}$$

where I_0 is the intensity at the point the ray enters the volume.

3.1.2 Emission

We only consider the effects of emission, no absorption or scattering is accounted for. This yields the differential equation

$$\frac{dI}{ds} = c(s)\tau(s) \tag{3.3}$$

where $\tau(s)$ is a infinitely small absorption and $c(s)$ is the intensity with which the light radiates. The solution to this equation is

$$I(s) = I_0 + \int_0^s c(t)\tau(t)dt \tag{3.4}$$

3.1.3 Emission with absorption

Combining emission with absorption gives a more realistic lighting model. Here every particle absorbs incoming light and emits light on its own, but there is no scattering between particles, other than in the ray direction.

$$\frac{dI}{ds} = c(s)\tau(s) - \tau(s)I(s) \tag{3.5}$$

The solution to this differential equation is

$$I(D) = I_0 \exp^{-\int_0^D \tau(s)\,dt} + \int_0^D c(t)\tau(t)dt \exp^{-\int_s^D \tau(s)\,dt} \tag{3.6}$$

where D is the position at the eye, and s on the edge of the volume. The first term is essentially equation 3.2 and describes light coming from the background and being multiplied by the volumes transparency. The last term describes how the volume emits and absorbs incoming light.

This equation is the low albedo volume rendering integral used by most common volume rendering methods.

3.1.4 Emission-absorption with shadows

Volume shadows are generally not even considered with most traditional methods since it is a very computationally expensive operation to perform. Nevertheless, shadows add a lot of information to an image and with ever increasing speeds among graphic accelerator cards, shadows are bound to be included in many upcoming methods.

Shadows are calculated by sending a ray towards the lightsource for every sample along the viewing ray. This light-ray is then used to calculate how much each sample is occluded from the lightsource by the materials that exist between the sample and the light.

The light ray can be written as:

$$T(s) = \exp^{-\int_s^L \tau(s)\,ds} \tag{3.7}$$

where L is the position of the lightsource. This equation is then appended to the standard volume rendering integral.

3.1.5 Discretization of the volume rendering integral

According to Siegel et.al [41] evaluating the 3.6 integral analytically on a computer is very difficult even though it can be done under certain circumstances. Computers in general are however much better suited to solve problems numerically. This calls for a discrete version which can approximate the volume rendering integral. This approximation is commonly made by a Riemann sum where the viewing ray is divided into equally long line segments.

The absorption part can be rewritten

$$\exp^{-\int_0^D \tau(s)\,dt} \approx \exp^{-\sum_{i=1}^{n} \tau(i\Delta x)\Delta x} \approx \prod_{i=1}^{n} exp(\tau(i\Delta x)\Delta x)$$

And the emission part becomes

$$\int_0^D c(t)\tau(t)dt \approx \sum_{i=1}^{n} c_i\tau_i$$

Together the discrete version of the volume rendering integral is

$$I(D) = I_0 \prod exp(\tau(i\Delta x)\Delta x) + \sum c_i\tau_i \prod exp(\tau(i\Delta x) \tag{3.8}$$

For complete solutions to all stages of computation, see [23].

3.2 Illumination model

With the volume rendering equation established, it remains to describe how the terms for colour and opacity in the equation are going to be evaluated. Shading is the process of calculating a lighting model where the result is the colour for an area (polygon) or for individual pixels. A lighting model calculates the behaviour of light, specifically how lightsources, materials and geometry interacts.

Illumination models are partitioned into two categories depending on how far reaching it is. Local illumination is only concerned with how the light behaves at one specific surface position, while global illumination deals with shadows and indirect light as well.

The most common local illumination model which calculates light per pixel is the phong[32] lighting model. The phong model is made up of three parts, ambient, diffuse and specular.

$$I_{phong} = I_{ambient} + I_{diffuse} + I_{specular}$$

The ambient term is a constant which simulates indirect light on the parts of geometry where there is no direct light, they would otherwise be entirely black.

$$I_{ambient} = Ka \tag{3.9}$$

The diffuse term calculates the reflection of light from a surface without shininess, called a matte surface. The light is reflected at the same angle as it is striking the surface relative to the surface normal.

$$I_{diffuse} = Ip * Kd * cos(\delta) \tag{3.10}$$

where Ip is light source intensity, Kd is a material constant describing colour and δ is the angle between lightsource and surface normal.

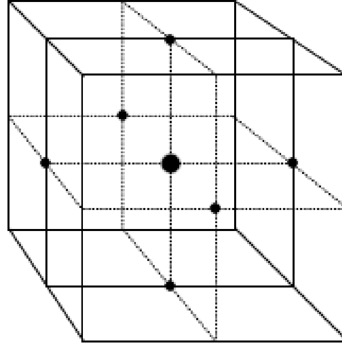

Figure 3.2: The gradient of the voxel in the centre can be computed by central difference using the six closest neighbouring voxels.

Finally, the specular term is added to simulate the shininess of surfaces, it creates highlights which give the viewer cues of lightsource positions and geometry details.

$$I_{specular} = Ip * Ks * cos^n(a) \tag{3.11}$$

where Ks is the specular material constant and n determines the size of the highlights.

3.3 Gradient computation

An essential part of the illumination model is the surface normal. Many of the volume rendering techniques are based on not having a clearly defined surface which of course means that there is no defined surface normal either. This is undoubtedly a problem, but it can fortunately be solved by calculating the gradient vector at all points of interest and use it instead of a surface normal. The gradient vector is mathematically the first order derivate of a discrete volume [3]. The derivate can be defined as:

$$df(x, y, z) = (\frac{df}{dx}, \frac{df}{dy}, \frac{df}{dz})$$

There are several methods for calculating the gradient vector, the most common method is the central difference which estimates the derivative by calculating the first terms of a Taylor expansion [25]. In practice, the six neighbouring voxels are used in the following equation, see figure 3.1 as well.

$$\begin{aligned} df(x, y, z) = &(1/2) * (f(x + 1, y, z) - f(x - 1, y, z)), \\ &(1/2) * (f(x, y + 1, z) - f(x - 1, y - 1, z)), \\ &(1/2) * (f(, y, z + 1) - f(x - 1, y, z - 1)) \end{aligned}$$

For an even better estimation of the gradient vector, the 26 closest neighbours can be evaluated, the Zucker-Hummel operator is such an implementation, see [30] for details.

Another alternative which will most likely yield the best results of these suggested methods is the Sobel 3D operator. It applies a $3 * 3 * 3$ filter kernel.

Using the local gradient vector works well for shading, but there is a problem with regions inside a material. One property of the gradient is that it only possesses a magnitude when going from a material to empty space or in between different materials, it has no magnitude inside a material. With no magnitude, there should not be any shading either in these regions. However, due to noise and small differences inside a material, a faulty gradient is likely to be calculated.

There are a number of ways this problem can be solved. The easiest method is to use a multidimensional transfer function that considers both the current data value and the gradient magnitude. In case that is not possible, gradient magnitude modulation can be used. This technique scales the opacity of a sample according to its magnitude, thus making the noise disappear. Another, more complex solution is to include a surface scalar to the standard volume rendering integral.

Chapter 4

Classification

The most important operation during all volume rendering techniques is without a doubt the classification stage. It is at this point geometric features are chosen to be emphasized to varying degrees, or be discarded. This stage is the key to make the most of the data. Classification is a process where the data values are mapped into a colour $c(t)$ and opacity value $\tau(s)$, this is usually done by what is called a transfer function. The goal of the classification is to make the interesting features of the underlying data be shown in the best way possible without introducing any aliasing. Even though classification is very important for the final picture and ultimately important for the usefulness of volume rendering as a whole, there is no general method that can do this automatically. Levoy [18] devised a method for automatic surface extraction in medical data and there are a few more semi-automatic methods as well. Finding the sought-after features manually from a three-dimensional dataset is difficult which is why methods that make this easier are a big research area at present.

4.1 Pre and post classification

The classification of values can be performed before or after the colour and opacity of the point in space has been determined by interpolation from the surrounding voxels. At first glance it might seem like it makes little or no difference, but in practice it does have a big impact on the quality of the final image. A classic render pipeline that uses pre-shading as well as pre-classification is the raycasting method proposed by Levoy, see figure 5.2.

Pre-classification produces faulty colours, this will be shown in the next section, and in addition to this it also introduces blur to the image as high frequencies within the transfer function are not possible to reproduce. The only time pre- and post-classification will produce the same results when a non-linear interpolation method is used (as tri-linear for example) is when the transfer function is constant[9].

Pre-classification does still have its uses though. If the final picture quality is not of utmost importance, it can be used to speed up the calculations in software raycasting for instance since the whole volume can be pre-shaded at program start-up. In hardware rendering, graphics cards must have support for dependent texture reads to be able to use post-classification, which makes pre-classification the only method usable by cards lacking this feature. In contrast to pre-classification, post-classification produces the correct results which results in better picture quality, however it also requires more

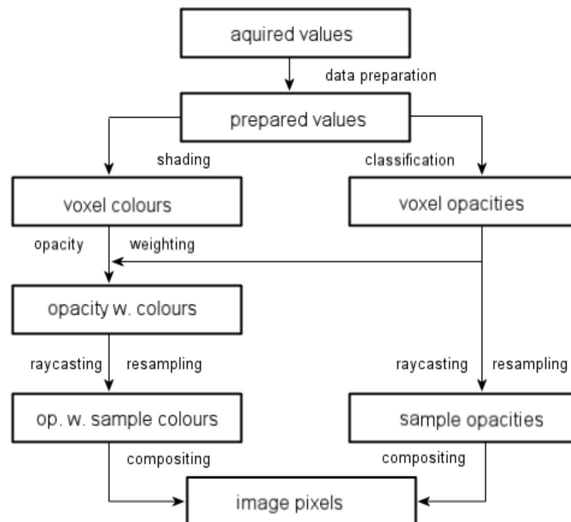

Figure 4.1: The new pipeline feeds the separately classified opacities over to the shading path before the reconstruction stage.

calculations and more advanced graphics hardware generally.

4.1.1 Opacity weighted colour interpolation

Post-classification always produces the correct result (with certain exceptions, see section about pre-integration), but this comes at a higher computational cost and with higher hardware requirements in the case of hardware assisted rendering. Using pre-classification is a good way to speed up any method since both classification and shading is done as a pre-processing stage.

Fortunately, Wittenbrink et al.[50], discovered that many of the aliasing effects produced by rendering pipelines like the one Levoy proposed, see figure 5.2, could be alleviated if opacity weighted colours were used. Levoy issued a statement[20] in 2000 admitting that his figure depicting the rendering pipeline is faulty.

In Levoy's pipeline, colours and opacities are determined in separate stages. Wittenbrink proposed a change to the rendering pipeline, voxel colours should be multiplied by their corresponding voxel opacity, the colours are then said to be associated. See figure 4.1.

One of the more obvious problems this change solves is in the case depicted in figure 4.2. If a sample is taken within the area of the four voxels, the colour will be interpolated from all four voxels, even though voxel D has an alpha value of zero. The contribution should have been zero, but since the pipelines for shading and classification are separated, this result is inevitable. Wittenbrinks solution to this problem yields the correct result since multiplication between colours and opacity will automatically make sure that each voxel colour is only contributing as much as it should.

This example problem often occurs when a ray exits a volume and enters open space. Empty space contributes to the surface colour which causes aliasing effects (colour banding) on the surface of the volume. Using opacity weighted colour interpolation does

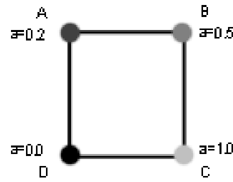

Figure 4.2: Four voxels with varying opacity and colour. In Levoy's implementation, voxel D contributes to samples, while Wittenbrink's does not.

not solve all problems relating to pre-classification, but it eliminates some of the most striking. Wittenbrinks pipeline should be used for any pre-classification method.

4.2 Transfer functions

The job of the transfer function is to map data values of interest for the user into colour and opacity, properties that are used by the renderer. The design of the transfer function is critical in many respects, and it is foremost detrimental in deciding the usability of the rendering program. The transfer function is the only part of the rendering pipeline where the end user can change and manipulate the output. It is simply put the interface by which the end user can bring out interesting features from the data. Unfortunately, the task of finding and isolating areas is something the user must perform by hand usually, and this means that the transfer function has to strike a balance between ease of use and technical complexity. Research in the transfer function area has not yet come far enough to present a fully automatic solution. There exists a number of semi-automatic variants, such as Levoy [18] region boundary surfaces which works fairly well on medical data, a genetic algorithm approach by He. et al.[11], and another automatic method by Durkin[13].

In summary, the fundamental requirements for the transfer function are:

– The transfer function must be able to isolate and present the areas that the end user of the program is interested in.

– The transfer function mechanics must be simple to understand for the end user.

Unfortunately, the means of accomplishing these requirements are often conflicting with each other, as we will see.

4.2.1 One-dimensional transfer functions

The simplest and most common transfer function is the one-dimensional transfer function. It is called one-dimensional since it only uses one value in its mapping function. This value is the voxel density at the sample location. Step functions are common in order to accentuate differences between regions. One-dimensional transfer functions are often used because they fulfill one of the fundamental requirements, manipulation of the transfer function is easy to understand for the end user. The user only has to consider the density and what corresponding opacity and colour to use at different levels. In addition, it is very fast and easy to implement. However, this simple transfer function

design has several problems, both with aliasing and with how well it isolates areas. Using a step function is equivalent to binary classification which is always producing aliasing effects such as holes. Area isolation is often difficult to achieve with only density as a guide, since there is no link between density and spatial position within the volume. An area of interest might have the same density as another area surrounding it, making visualization of the area problematic. A tree-dimensional dataset can have multiple boundaries with the same density value, this is yet another situation a one-dimensional function cannot solve correctly.

4.2.2 Multidimensional transfer functions

In order to overcome the limitations of one-dimensional transfer functions, such as the difficulty in isolating areas, an obvious solution is to expand the transfer function to include more dimensions. One of the interesting properties to include in the transfer function in order to increase its capability in isolating areas is the first order derivative of the volume, since it is a good indication of where areas of change are. Areas of change are in turn a good indication where edges between surfaces are located. The magnitude of the local gradient is the equivalent of the first order derivative, and it is therefore used in most two-dimensional transfer functions. For even better results, the second derivative can be included as well, yielding a three-dimensional transfer function.

Levoy [18] introduced two different two-dimensional transfer functions. The first was Isovalue contour surfaces and it was intended to be used on arbitrary structured data. The algorithm starts by assigning the wanted opacity α to all voxels carrying the density f_v. In order to enhance the visual appearance and make the transition between areas smoother, voxels with values close to f_v are faded out with the inverse proportional to the gradient.

$$\alpha(x_i) = \alpha_v \begin{cases} 1 & \text{if } |\Delta f(x_i)| = 0 \text{ and } f(x_i) = f_v \\ 1 - \frac{1}{r}\left|\frac{f_v - f(x_i)}{\Delta f(x_i)}\right| & \text{if } |\Delta f(x_i)| > 0 \text{ and } f(x_i) - r\,|\Delta f(x_i)| \leq f_v \leq f(x_i) + r\,|\Delta f(x_i)| \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

where r is the speed by which the opacity falls off, $f(x_i)$ is the density of the current voxel, $\delta f(x_i)$ is the gradient.

Multiple surfaces can be computed and then combined into one final picture by using the compositing

$$\alpha_{tot}(x_i) = 1 - \prod_{n=1}^{N} (1 - \alpha_n(x_i) \tag{4.2}$$

Levoy's other transfer function, region boundary surfaces, is tailor made for medical data. This is a semi-automatic transfer function that extracts and displays surfaces, suppressing any intermediate data in between. It assumes that the volume may have an arbitrary amount of tissues (materials) and that these tissues have different density values. Furthermore, it assumes that any tissue does not border to more than two other tissues. If this is the case, then it is possible to sort all tissues in a ascending order, and assume that any tissue in the sorted list only touches the tissues directly above and

below it in the list. The equation becomes

$$\alpha(x_i) = \Delta f(x_i) \begin{cases} \alpha_{v_{n+1}} \left(\frac{f(x_i) - f_{v_n}}{f_{v_n+1} - f_{v_n}}\right) + \alpha_{v_n} \left(\frac{f_{v_n+1} - f(x_i)}{f_{v_n+1} - f_{v_n}}\right) & \text{if } f_{v_n} \leq f(x_i) \leq f_{v_n+1} \\ 0 & \text{otherwise} \end{cases} \tag{4.3}$$

The calculated alpha value is multiplied by the gradient magnitude to make surfaces have the highest alpha.

While multidimensional transfer functions can produce very good results in the best case, they are much more difficult to use for the end user. This means that the user must have a very good understanding of both the program and the data that is visualised. With every added dimension, the complexity of the required user interface increases. Kniss[14] shows how a good user interface for multi-dimensional transfer functions can be made.

4.3 Pre-integrated classification

The problems surrounding data reconstruction was discussed earlier in chapter 2. It was established that the volume must be sampled according to the sampling theorem to avoid aliasing. One of the important properties of the theorem states that the sample frequency is required to be at least the Nyquist frequency.

However, the issue of aliasing effects due to undersampling is not limited to just the reconstruction. The transfer function may have non-linear properties which affect the required sampling frequency as well. This was noted by Engel et al.[9] in 2001, and their solution was to use a new classification scheme called pre-integration.

Engel et.al. observed that the required sampling frequency was in fact the product of the Nyquist frequency in the discrete volume and the highest Nyquist frequency contained in either the transfer function for transparency, or the transfer function for colour, when the transfer function is non-linear. This problem is avoided all together by using a linear transfer function (smooth transitions), but this might not be desirable or possible at all times.

It is achievable to avoid this product of Nyquist frequencies and thereby relaxing the sampling requirements if the integration of the scalar field is separated from the integration of the transfer function. The idea of pre-integration is in essence to compute the integration of the transfer function in a separate stage with the classification data and store this information in a look-up table.

The integration of the transfer function is done by rendering slabs instead of slices. A slab is defined as the volume between two slices. The slab is integrated by using the scalar value of the front and back slices that make up the slab together with the intra-slice distance. These equations are then performed for all possible scalar values and are then stored and used instead of the standard transfer function look-up table.

The opacity equation becomes

$$\alpha(s_f, s_b, d) \approx 1 - \exp\left(-\frac{d}{s_b - s_f}\left(\int_0^{s_b} \tau(s_b)ds - \int_0^{s_f} \tau(s_f)ds\right)\right) \tag{4.4}$$

where s_f and s_b is the scalar value of the front and back slice respectively, d is the distance between s_f and s_b.

And similarly for the colour

$$c(s_f, s_b, d) \approx \frac{d}{s_b - s_f}\left(\int_0^{s_b} C(s_b)ds - \int_0^{s_f} C(s_f)ds\right) \tag{4.5}$$

It should be noted that these equations are approximations of the actual analytical equations which can be found in [9].

There is no doubt that this technique has great benefits, it is possible to avoid the reconstruction aliasing brought on by the transfer function. However, there are a number of drawbacks as well. The biggest problem is that it does take a long time to calculate the integration look-up table, and even though this can be done at program start-up, the tables must be recomputed every time the transfer function is changed. Pre-integration can be difficult to implement on some graphic cards also, the increased number of required texture reads means that it is not possible to use in conjunction with other methods.

4.4 Compositing

The discrete version of the volume rendering equation replaces the continuous integral with a Riemann sum as can be seen in chapter 3. When the colour of each sample is determined by classification and shading, and when the transparency is determined by classification, the next step is to evaluate the volume rendering equation.

This evaluation is performed by a process called compositing. Digital image compositing is a general technique for use in the field of two-dimensional image processing that was proposed by Porter et.al.[34] in 1984. It is used to combine several translucent images or colours into one single result.

Compositing can be done either back to front as in [18] or front to back. The back to front equation using porter and duffs under operator is:

$$C_{i+1} = c_i * (1 - \alpha_i) + C_i * \alpha_i \tag{4.6}$$

where C_i is the current composited colour, α_i and c_i are the alpha and colour of the current sample.

Front to back compositing is equivalent to back to front, but it has an added advantage. Since the composition is done towards the back and the current transparency is known at all times, the compositing can be stopped early when the translucency is full (nothing behind that point is going to affect the final image). This is one of the more powerful optimizations that can be done for several rendering techniques.

The front to back equation using the over operator is:

$$C_{i+1} = \alpha_i * c_i + (1 - \alpha_i) * A_i * C_i \tag{4.7}$$

$$A_{i+1} = \alpha_i + (1 - \alpha_i) * A_i \tag{4.8}$$

where C_i is the current composited colour, A_i is the current composited transparency, α_i and c_i are the transparency and respective colour of the current sample. The final rendered image is generally finished when this stage is completed.

Chapter 5

Software volume rendering methods

Over the years since the idea of volume rendering first came into being, a lot of different techniques have been presented that improve the rendering speed or enhance the picture quality. A few of these techniques incorporate ideas from many sources to produce a unification of ideas that later becomes a benchmark method within the field. This chapter will focus on describing how these benchmark methods works, and analyse what their benefits and drawbacks are.

There is no single "complete" algorithm that both deliver the best image quality and the best speed. Not surprising, this means that newer algorithms that build upon raycasting for instance, which had an excellent image quality from its inception, are largely about speed optimizations while other approaches that build on fast algorithms like shear-warp are concentrating on improving picture quality. The lack of a complete algorithm has led researchers to explore a diverse spectrum of methodologies, ranging from wavelet analysis to snowball analogies.

The volume rendering methods can be divided into three paradigms, object order, image order and domain based rendering. This division is based on how the data is handled and in what manner it contributes to the final image.

5.1 Image order

Image order, or backward mapping as it is also known as, is an area within volume rendering where techniques are evaluating each on-screen pixel one at a time. Rays are cast backwards from the screen into the data, where it is then sampled to calculate exactly how much it contributes to that particular pixel. Raycasting and variations of it is an example of methods that works in this way. Image order techniques generally produce the best image quality, but at the price of slow render speed.

5.2 Object order

The idea of object order is to reverse many of the ideas that exist with image order. In object order rendering, which is often called forward mapping, the data is projected in a forward direction, from the object onto the screen. Data coherence can often be exploited

by traversing voxels in storage order. Forward mapping techniques have the advantage of being possible to run in parallel due to the fact that only a small surrounding area around a voxel needs to be examined for shading and classification calculations. The typical behaviour of a method of this type is very good render speed, with average to good picture quality. Shear-warp and splatting are two major methods of this kind.

5.3 Domain based

Methods of this kind operate by switching much of the computations to a different domain, such as frequency, compression or others. The domain switch is usually done because there is something to gain from it, some calculations may quicker or more easily performed in the other domain. Domain based methods, especially those based on the frequency domain, are often very fast with low computational complexity. They are sometimes plagued by limitations caused by the domain swap however, making them less general than image and object order techniques.

5.4 Generic render pipeline

Although techniques based on image, object and domain based paradigms are approaching the concept of volume rendering in different ways, they all share a few important steps in the rendering pipeline. They all deal with data that are in some way collected and represent some property that cannot be immediately rendered.

- Data preparation: Before any render pipeline operation is performed on the data, it may need some sort of preparation first. Filtering, anti-aliasing, contrast enhancement, domain switching are some common operations that are used.

- Reconstruction: The volume is a discrete sampling of a continuous signal. In order to be able to render the volume with as few sampling errors as possible, the signal must be reconstructed by a filter.

- Mapping: This is the heart of any algorithm. Initially the data values in the volume have no meaning in a graphical sense. They have to be converted into material properties somehow. The mapping stage is called classification and the function that maps data to colour and opacity is called a transfer function.

- Rendering: The treated samples are rendered to screen in the final step. A number of optical and illumination models can be used depending on how realistic the final image has to be or how computationally complex the operation is allowed to be.

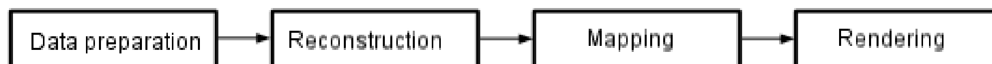

Figure 5.1: These are the common steps for any pipeline, implementation details and order of execution is specific however.

5.5 Raycasting

By the late 80's, a number of surface extraction techniques had been developed, like the marching cubes algorithm[21]. Surface extraction and rendering using polygons works fairly well on arbitrary data, but it does have one big drawback; aliasing effects due to the difficulty of classifying where the actual surface is. This problem is further amplified by the fact that the marching cubes algorithm uses a binary classification scheme which means that a data point is either on the surface, or it is not. Obviously, that kind of classification is not very precise, and it does not work well even with real numbers. Polygons are generally not well suited to display, complex, fine details, especially since a near infinite number is needed. The classification errors produce aliasing artefacts, meaning that surface features that do not exist in the dataset are embedded in the final rendered picture. For applications used in the medical field in particular, aliasing or any artefacts at all are not acceptable.

One solution to the classification problem is to use a technique called raycasting. The basic algorithm is simple, rays are cast into a data volume and samples are taken along each ray by interpolation of the surrounding voxels. This means that no intermediate geometry is constructed and thus the classification problem is solved. Another advantage is that the volume can be rendered semi-transparent, and as a result it is possible to display many surfaces within each other.

The first method using this approach was Kajiya in 1977, then Blinn[4] made many improvements in 1982, among those were the low albedo assumption and the general volume rendering integral. Kajiya[12] continued in 1984, showing that high albedo environments could be used as well. Then in 1988, Levoy[18] published a paper with a raycasting algorithm which since has become the definition for raycasting.

5.5.1 Levoy's raycasting

Levoy's algorithm uses a rendering pipeline where the shading is separated from the classification, see figure 5.2. The reason for this separation is that if it is not done, the shading will depend on weather the classification is successful or not, which will in turn lead to aliasing.

The algorithm starts by preparing the data values, this step could be anything from checking that voxels are aligned correctly to interpolation of grids to increase the number of voxels on one axis.

In the next stage, the entire volume of prepared values is shaded in the shading pipeline, yielding voxel colours. Classification is done at the same time in the classification pipeline. Performing the calculation of colours and opacities prior to resampling does have several drawbacks. Basically, the combination of split pipeline and pre-calculation leads to several aliasing effects. A more thorough explanation of the various problems concerning Levoy's rendering pipeline can be read in section 4.1.1.

Levoy suggested two different methods for classification. Isovalue contour surfaces, for pure volume rendering with the possibility of showing several semitransparent surfaces. This is a two-dimensional classification method using the voxel value as well as the gradient magnitude to calculate the opacity. The other method, region boundary surfaces, is concentrated on rendering medical data and biological tissues in particular. It uses a number of assumptions of how the human body is constructed to extract surfaces automatically. Both these methods are explained in detail in chapter 4.

After shading and classification is done at each voxel, the volume is set up for the

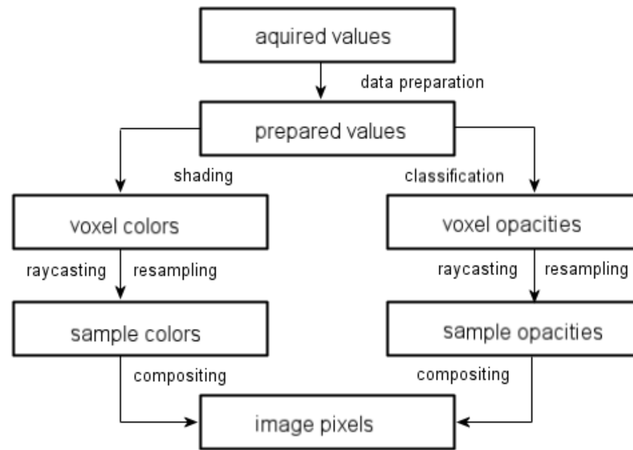

Figure 5.2: The standard pre-classification raycasting pipeline with separated paths for classification and shading.

raycasting stage. Rays are cast into the volume from each pixel on screen. Each ray is sampled at equidistant intervals, every sample is determined using tri-linear interpolation of the eight closest voxels.

Finally, the resulting sample colours and opacities along each ray are fed into the compositing step. The samples are composited back to front to produce the final colour for the on-screen pixel.

5.5.2 Optimization techniques

The picture quality is very good with raycasting and it is therefore one of the most used methods. It does have a great speed disadvantage though because of the enormous amount of calculations that must be performed. Because of this, there have been a number of different approaches to optimize various parts of the raycasting pipeline.

5.5.3 Early ray termination

This is an optimization that is simple to implement and it is very effective. When the rays are cast front to back and are composited at every step, the ray can be terminated as soon as the opacity reaches a threshold value. Integrating beyond the point where the pixel turns opaque will not change the colour or opacity any further, it is a waste of computation time. Early ray termination provides the best results when the opacity threshold is reached quickly, such as when surfaces are rendered. In this case, much of the data will be skipped, and the renderer will receive a big speed increase. A highly transparent volume will on the other hand not see much of a speed gain. It can be said that the transfer function settings are affecting the usefulness of this technique to a high degree.

5.5.4 Empty space skipping

This is another powerful optimization method that complements early ray termination nicely. As was mentioned above, early ray termination does not work well with mostly transparent volumes. Empty space skipping is used to optimize raycasting under just such conditions. A data structure is used to keep track of areas in the volume where the density is zero or close to zero, many implementations use octrees for this task. The nodes in the tree save information such as the highest and lowest density value within a partition. If the space is deemed empty, none of the data within that partition will be computed.

5.5.5 Progressive refinement

Finding good transfer function values are a tedious task of trial and error mostly, requiring the slow raycasting procedure to run in between changes. Progressive refinement is a way to speed up rendering in the early stages when the user is still trying to find a good transfer function setting. The quality is reduced while the user fine tunes the values, letting the raycasting run faster. The means to accomplish this reduction in quality differ, it can be done by simply reducing the sample frequency and resolution, then increasing it step by step. More complicated solutions exist also[19] where a pyramid-like data structure is used.

5.6 Frequency domain rendering

The frequency domain rendering algorithm was proposed in 1993 by Malzbender[22] and was later in the same year extended by Levoy et al.[45]. Its main advantage over image order algorithms is that the computational complexity is close to a magnitude lower, $O(N^3)$ for image order compared to $O(N^2 logN)$ for frequency domain rendering.

Much of the theoretical foundation in this method is based on the Fourier transform slice projection theorem. This theorem states that in the case of a three-dimensional volume, a two-dimensional slice through the spatial volume is equal to a slice that goes through the fourier transformed volume at the same angles, and passes through the origin. This means that the volume can be sliced and computed at any viewing angle.

The core algorithm is done in three steps. First the volume is transformed from the three-dimensional spatial domain into the frequency domain by using the fast Fourier transform. This is a slow process but it can fortunately be done as a pre-processing step. In the second step, the slice through the volume is computed in the same way as was mentioned in the previous paragraph. However, the slice through the spatial volume is only equal to the frequency transformed volume at one single point, the origin. This means that this step is mainly a reconstruction stage performed by a filter. In the third and final step is the reconstructed frequency domain plane translated back to the spatial domain by using an inverse two-dimensional transformation.

The reconstruction contributes $O(N^2)$ together with the inverse transformation $O(logN)$. They are the main computations that are performed during the rendering, resulting in the $O(N^2 logN)$ complexity.

There are a number of drawbacks to this rendering technique. First of all, it uses equation 3.2 which is a slightly different illumination model than the other techniques[53], since only absorption is modelled with this model. This leads to a final image that lacks depth information, making the result look much like x-ray images. A number of methods

have tried to alleviate this problem, among them depth cueing[45], but there is still no real solution.

The biggest drawback of most of the frequency domain methods is that transfer functions cannot be used at all. Any interactivity, such as changing opacity or colours of areas in the volume is impossible after the pre-processing stage. As recent as this year (2004) a new approach has been presented by Nagy et al.[31] which allow a special kind of transfer functions that work under some conditions, but the problem is still not solved.

5.7 Splatting

Splatting is one of the primary object-order rendering techniques. It was first developed by Westover[49] in 1990 and has since then been improved a number of times. A forward rendering technique works by mapping the volume data onto the image plane, as has been mentioned earlier.

Splatting works in a different fashion from raycasting, instead of sampling surrounding voxels to points along a ray, the voxels contribute to screen space via a distribution function, often called a footprint, which spans many pixels in screen space. The footprint is often represented by a Gaussian function. The term splatting is coined from how the voxels spread out over the screen plane by the distribution function, much like a snowball against a surface.

The original rendering algorithm by Westover is divided into four stages, transforming, shading, reconstruction and visibility. It begins by transforming voxels from object space into screen space. The analytical volume reconstruction equation for splatting is:

$$signal = \iiint hv(u-x, v-y, w-z)\rho(x,y,z) \sum \delta(x,y,z) du dv dw$$

where hv is the volume reconstruction kernel, often a sphere, ρ is the density, and δ is the comb function. The discrete equation used to calculate the contribution of all voxels perpendicularly behind a screen coordinate (x, y, z) along a ray going down negative z is

$$signal_{3D}(x,y,z) = \sum_{D \ni vol} hv(x-D_x, y-D_y, z-D_z)\rho(D)$$

where D is a density sample that is within the kernel, and D_x, D_y and D_z are on screen coordinates.

The splat footprint is defined as

$$splat(x,y) = \int hv(x,y,z)dz$$

A Gaussian reconstruction kernel is defined as:

$$hv(x,y,z) = d * \exp\left(\frac{-x^2}{a^2}, \frac{-y^2}{b^2}, \frac{-z^2}{c^2}\right)$$

The volume is shaded in a pre-processing step and is then reconstructed. The first version of splatting[48] only supported one shading model, but since[49], the shading is generalized and common models like phong-shading can be used. It is common for splatting algorithm implementations to restrict the shading model to only make use of local data. This restriction makes it possible to run the rendering in parallel.

How much every single voxel in the volume is contributing to a pixel is determined during the reconstruction step. The simplest possible solution is to integrate the reconstruction kernel for every voxel for every screen pixel. This solution would be far too slow which is why Westover introduced a few enhancements to the brute force algorithm. The footprint is constant with an orthographic projection, it is hence possible to use a lookup table of the footprint function. Westover called this pre-computed footprint function a generic footprint table, because it can be used for any view. The reconstruction process is done on sheets which are slices of the rectilinear volume aligned along the major axis to the current screen plane. Each voxels contribution is calculated on the particular sheet it happens to be on, and then the sheets are composited together in front to back order.

Westovers original algorithm does have a few problems though. It only supports pre-classification and pre-shading. The axis-aligned sheets make a sudden, visible change in colour and opacity when the object is rotated past 45 degrees and new sheets are loaded from another axis. Determining the optimal size of the footprint function can be difficult. A too small table results in gaps and a too big table causes extra blurring. Another big problem is that the sampling rate is locked to the inter-grid spacing.

However, the splatting method has seen a lot of improvements over the years, and most of these problems are now solved. Splatting now supports both post-shading and post-classification[28]. The sheet changing problem was solved by Mueller et al.[27] in 1998. They solved the problem by using view-aligned sheets. The sampling distance problem was solved in[27] as well. This was done by integrating slabs (the area between two slices) instead of just point values on slices. Interestingly, Mueller et al.[29] showed that splatting can be implemented as a image-order method as well.

Splatting does have many good features, of great interest is that it can be made parallel which makes it potentially very fast. Another good feature is that it is capable of perspective projection by using an elliptical footprint of varying size, and this can be performed without aliasing effects. The speed can be optimized even further by implementing "early splat-termination", a optimization technique that is very similar to early ray termination. Overall, the picture quality of splatting rivals that of raycasting which was pointed out in a survey by Meissner et al.[24].

5.8 Shear-Warp

The shear-warp technique was introduced by Levoy et al.[17] in 1994 and is a mixture of several prior methods and it offers excellent speed without sacrificing the picture quality. It is still today regarded as the fastest software rendering method and it is the basis for many hardware rendering methods.

The shear-warp method tries to combine the picture quality of image-based techniques such as raycasting with the efficiency of object-order techniques. The main disadvantage of raycasting is that a lot of computational time is used for tracing the viewing rays through the volume and the interpolation of voxels. Shear-warp is based on the idea that if raycasting is always done perpendicularly to the volume, then the data can potentially be traversed much more efficiently by exploiting how data in rectilinear grids are stored. Likewise can interpolation be made easier compared to when the rays are cast into the volume at an angle. This is the core idea of shear-warp rendering. The technique assumes that the volume data is arranged in rectilinear grids, and consequently, it will not work on unstructured data.

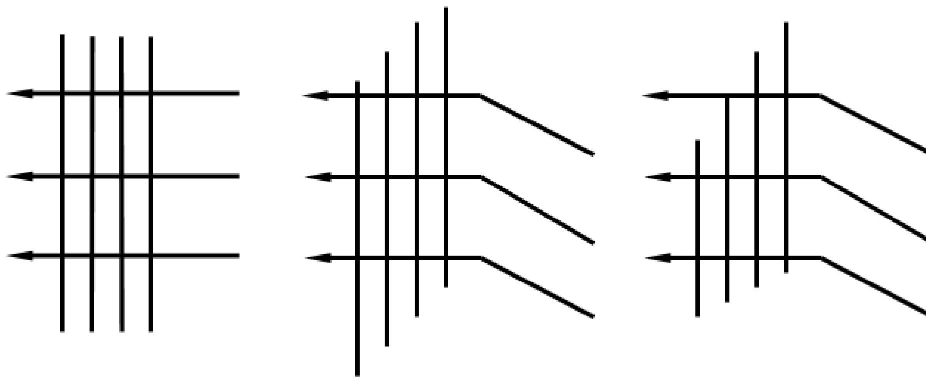

Figure 5.3: The simple case where the rays are perpendicular to the volume is shown in the figure to the left. When the view is rotated, the slices are sheared (center). Alias-free perspective projection is possible by scaling the slices (right).

Basically, shear-warp first transforms the volume by shearing the two-dimensional slices along the main viewing angle, this makes it possible to process the volume perpendicularly to the slices. Then the resulting image of the shearing process is finally warped to the image plane using common image manipulation techniques, see figure 5.3 center. Most volume rendering techniques are only capable of parallel projections, otherwise the volume is not sampled uniformly, but perprective projection can be achieved with shear-warp without aliasing effects by simply scaling the slices figure 5.3 right.

The basis for volume rendering using shear-warp is shear-warp factorisation which was originally used for optimizations of volume rendering in SIMD processors. The shear-warp factorisation is done by evaluating the equation

$$M(view) = P * S * M(warp)$$

where P is a rotational matrix rotating the volume along z-axis, S is the shear matrix and $M(warp)$ warps the viewing image.

The algorithm for performing a simple shear-warp can be summarized as:

1. Shear all slices according to S
2. Composite the sheared slices to form an intermediate image.
3. Warp the intermediate image into image space.

In addition to the basic shearing of the data and the warping of the image plane, Levoy et al.[17] realized that they could exploit the coherence between a scanline in the image and a scanline in the data to efficiently remove transparent voxels. They chose to use run-length encoding (RLE) of each scanline on all three axes as a pre-processing stage. As up to 95% of a classified volume consists of transparent voxels, this saves both computational time and space. The way the scanlines are classified emulates the common raycasting optimization of early ray termination as well, increasing speed even more.

There are a number of disadvantages regarding RLE and the shear-warp technique in general however. The major drawback of removing voxels during pre-processing is

of course that the classification cannot be changed during execution of the program. It also means that pre-classification must be used which reduces the picture quality significantly. Another downside is that, since the scanlines are processed per slice and then immediately composited, there can only be bilinear interpolation of voxels. This is of course not good for the final picture quality. Levoy et al.[17] realised the problem with only using pre-classification though and came up with an alternative way to classify voxels on the fly, instead of using pre-processing and pre-classification. Another problem with the original technique is that the sample distance along the viewing rays is constant and determined by the intra-slice distance. Three copies of the RLE compressed volume (one for each axis) require extra memory. There is also a popping artefact when the view passes 45 degrees, just as with the earlier splatting techniques.

Many of the image quality related problems have been solved though, thankfully. Sweeny et al.[43] shows how post-classification and post-shading can be implemented. They also perform intra-slice interpolation on the fly by. Schulze et al.[39] added pre-integration to shear warp, further increasing the picture quality. Schulze et al.[40] also showed how shear-warp can be parallelised.

Shear-warp is the fastest software rendering technique shown by Meissner et al.[24] who also found that it delivers a picture quality that is not far behind raycasting and splatting.

Chapter 6

Hardware acceleration techniques

Currently, there are two ways to employ hardware acceleration to volume rendering; either specially made hardware, or general purpose hardware. Special purpose graphics cards like the VIRIM has existed since 1994 and provide hardware accelerated raycasting at interactive speeds. The other possibility is to use general purpose consumer level graphics hardware that is mainly used for enhancing rendering speed in computer games. This particular area has grown in popularity among researchers since 2000 when the first programmable general purpose consumer graphics card was released, the Nvidia Geforce 256. Since these graphics cards are used to accelerate rendering of geometry like polygons, many common software techniques like raycasting which are not using intermediate geometry have to be adapted to take advantage of this strength. The most common and simplest brute force techniques are primarily influenced by shear-warp and was among the first published [36], but there was earlier publications on hardware acceleration as early as 1998 by Dachille et al.[8].

6.1 Consumer level hardware

To fully understand why the earliest techniques work the way they do, one must have a basic understanding of how the actual hardware is designed and what the limitations are. The strength of current graphics cards are their sheer speed when rendering large amounts of geometry combined with textures and the programmability of the on-board GPU. The earliest techniques mainly take advantage of the fast geometry rendering and texture capacity, mainly using 2D or 3D texture stacks with only limited use of the GPU. The latest techniques focus more on exploiting the ever improving GPU on the latest cards to do more advanced rendering and improve picture quality.

6.1.1 How the graphics hardware works

The graphics pipeline is the process of going from program assigned vertices and textures in a scene to a final image of coloured and lighted pixels on screen. When programmable hardware is used in conjunction with OpenGL, the standard OpenGL pipeline is preserved with two vital changes. The initial vertices are fed into the geometry processing

unit in the first stage, where they are transformed and lighted. The next step is the rasterisation, where triangles and other primitives from the geometry processing stage are turned into fragments. Fragments are pixels that are not quite ready yet, they must first pass through a number of tests, alpha testing, stencil testing or depth testing. Or, it can be blended together by alpha blending.

The geometry processing stage is said to be programmable. This means that user-made programs can be uploaded to the GPU and be executed. Since the geometry processing stage is only concerned with transformation and lighting of vertices, this stage is not of great interest for the basic volume rendering part. It is useful for the more advanced tasks as we will see.

The fragment stage is programmable as well, and it is here the shading model should be implemented. Since every pixel is shaded, the surface of the model is visually greatly enhanced compared to the lighting provided by the geometry stage. The level of programmability is severely limited however, on all but the latest cards.

6.1.2 Graphics hardware models

There are a few different makes and models of graphics hardware available, and it is important to know the capabilities of each of them. Currently, there are two major manufacturers of programmable graphics chips, ATI and nVidia. The first generation of programmable cards were the Ati Radeon 7000 series and the nVidia GeForce 256, released in 1999. Both these cards are capable of performing the simplest 2D techniques, but the lack of on board memory, generally only 32Mb, makes them very limited regarding object size. They also provide only a few GPU instructions, too few to actually implement a phong shading algorithm.

The following generation of cards appeared in 2001 and were called the Ati Radeon 8500 and the nVidia GeForce 3. The cards can render the most common volume sizes with their 64Mb of onboard memory. Both these cards are quite well suited to volume rendering as they support 3D textures, and have enough GPU power to barely do a phong shading model as well. The most limiting factors for this generation of cards are still the limited programmability of the GPU, the memory size and the lack of more advanced features like 3D dependent textures. The internal precision of floating point variables is also lacking somewhat, which might generate aliasing, especially with multi-pass solutions.

Current graphics cards like the nVidia FX series and the latest 6800 as well as ATI's 9000 series and x800 are very well suited to volume rendering, especially compared to older generations. The cards are available with up to 512Mb of onboard memory which encompasses very high resolution volumes of 1024x1024x512. The GPU of this generation of cards supports a very large number of instructions and can be programmed by using high level program languages like HLSL, GLSL or CG.

6.1.3 Graphics hardware issues

A number of technical factors are limiting the different hardware architectures. They can be summarized as:

- Available memory (limits the size of the volume)
- Supported texture dimensions (2D- and/or 3D-textures)

– Number of texture units (limits the number of available textures used in advanced methods)

– Dependent textures or similar feature (needed for multidimensional post-classification)

– Number of fragment instructions available (limits more advanced methods)

– Internal precision (might introduce additional aliasing)

The number of texture shaders determines how many textures that can be sampled. The four provided by GeForce3 cards are enough for 2D post-classification and per-pixel lighting, but more would be needed if lightmaps were used instead for example. The number of dependent texture samplings as well as the number of dimensions suppored for these textures is limiting the dimensionality of transfer functions.

6.2 Techniques

6.2.1 2D texture slicing

The idea of using two-dimensional textures for hardware accelerated volume rendering was among the first published hardware rendering techniques, Cabral et al.[5] showed that in 1994. A number of papers covering more current hardware were published in 2000 coinciding with the release of the GeForce level cards, especially of note is [9]. Just as shear warp it is also an object order technique, and overall, the technique borrows many ideas from the software rendering method.

The basic technique is developed around the early programmable graphics card, like the GeForce 256, where the programmable GPU was very limited, but the raw processing power of polygons and textures was abundant. Since the cards are very fast at rendering geometry, the obvious approach was to create proxy geometry, usually polygons to represent the slices in the volume, and then use textures to represent the data values. The volume rendering integral is then evaluated by alpha blending to render the final picture.

The slices containing data values are first converted into textures. Depending on what additional data may be needed for the rendering, the texture can store just the data value, or for instance, the local gradient as well. Exactly how much data that can be stored is determined by the available texture memory on the graphics card. As an example, on a 64mb card, a volume with sides 256^3 complete with gradients is the largest possible volume.

Because we want to be able to look at the volume from an arbitrary position, the slices in the volume need to be converted into textures from each of the major axis, this is why this technique is sometimes called axis aligned as well. If this is not done, we will be looking down the sides of the textures and obviously, the volume would not be visible from a perpendicular angle and it would seem to be distorted and filled with holes from a close to perpendicular angle.

Building three texture stacks means that three copies of the volume is stored in memory, this is clearly quite wasteful and it is one of the biggest disadvantages of the two-dimensional texture technique. Fortunately, this shortcoming is not too disadvantageous since not all three stacks have to be loaded into the graphics card at once, only the stack that is currently facing the camera is needed. The memory requirements are because of that not higher for the actual graphics card, but the system has to be able to store the

two stacks currently not in view in main memory. Additionally, a very noticeable delay is experienced every time the view rotates enough so that a new texture stack has to be loaded in from main memory.

The stack switching brings on one more problem, the sampling points change abruptly and noticeably when texture stacks are changed.

Another problem arises from the requirement of a constant sampling rate for the volume rendering integral. As the distance between the proxy geometry slices is constant, the sampling rate will change when the angle to the axis increases. There are two solutions one can use to solve this problem. A crude, but simple solution is to correct the opacity by taking the current viewing angle into account. An even better, but more complicated solution is to use the graphics cards ability to blend textures on the fly. A constant sampling distance can be calculated and then new textures can be produced anywhere within the volume by blending two adjacent slices. This does affect performance, the speed of the blending operation is dependent on the cards fillrate.

Filtering is generally only bilinear with two-dimensional texturing which can be seen as a significant drawback when it comes to picture quality, but trilinear interpolation is possible as well. A pseudo variant of trilinear filtering is preformed with the above mentioned blending of slices technique and with pre-integration (see classification chapter).

6.2.2 3D texture slicing

3D textures works slightly different from 2D textures. While a 2D texture is assigned to a polygon and then rendered, the exterior of a 3D texture can be rendered by placing polygons at the sides of the texture. The interior of a 3D texture is rendered by putting a number of polygons inside the texture. The polygons can be placed arbitrarily within the texture. This means it is possible to use view aligned polygons at any sample interval, thus eliminating the source for many aliasing effects.

Three-dimensional texturing methods do not have the problem of needing several texture stacks, instead one single 3D texture is filled with all the data from the volume. In fact, three-dimensional texturing solves many of the problems that plague the two-dimensional technique. The texture is in this case view-aligned, calculated on the fly by the hardware. This means that there are no problems with differing sample intervals, the slices are always oriented towards the view. Sample intervals can be set to any lengths and the result is calculated by the hardware. Filtering is trilinear, also calculated by hardware and there are of course no problems with stack switching.

An additional advantage is that perspective projection can be used with 3D textures. As discussed earlier, most rendering techniques require a parallel projection because otherwise the sampling intervals will become uneven. Fortunately, the geometry that slices through the 3D texture can be of any shape and have any orientation. This means that instead of using the standard planar polygon slices, one can use slices that are spherical to solve this problem. It should be noted that the added amount of complicated geometry might in practice not make it worth while.

6.2.3 Problems with slicing

The 2D- and 3D-texture rendering methods produces good results, considering both picture quality and render speed. Although it is possible for the graphic card to deliver interactive frame rates, the techniques are in essence pure brute force implementations. Every sample is contributing to the total workload, it has to be interpolated, lighted and

blended, even though they might not make a difference to the final scene. Recall from earlier that up to 95 percent of a volume might be transparent. See section 6.5 for a closer look at a couple of new ideas that uses different approaches to solve the problem.

6.3 Classification

All the classification procedures mentioned in the classification chapter are applicable to hardware rendering as well. This section will only add a few hardware specific issues regarding classification.

The classification procedure is completely independent from the type of texturing method that is being used, the possible limitations are depending on available hardware features. Pre-classification is supported on all hardware since the volume can be classified before it is converted into textures. Performing classification that way is very inflexible because once the textures are loaded there is no way of changing the classification without reloading all the textures. This inhibits the interactivity with the user severely as being able to change the classification on the fly is very important to the usability of the technique. Only having to reload the transfer function table would be a much more efficient method, and fortunately there is such a feature on GeForce level cards.

The feature is supported by most hardware and is called paletted textures. Paletted textures use a colour table which controls the colours of the textures. This can be used to implement a more flexible version of pre-classification, the colour table becomes the equivalent of a one-dimensional transfer function table as used by many software rendering algorithms.

Post-classification can be performed on all hardware that supports dependent texture lookup, this includes many cards produced from 2001 and onwards such as Geforce3- and Radeon8500-class cards and above. With dependent texture lookups, flexible post-classification with multidimensional transfer functions is possible. Dependent texturing allows the card to evaluate a texture at an arbitrary texture coordinate and use the values at either the alpha and red channel, or the blue and green channel as a texture coordinate in another texture. As described in chapter 4, the post-classification algorithm performs interpolation of data values before the actual classification is done. In the case of two-dimensional textures, bilinear filtering is applied, while tree-dimensional textures uses trilinear. The resulting data value, or fragment, as it is currently being halfway through the graphics pipeline, is then used as a coordinate in a separate texture. This separate texture is either a one- or two-dimensional pre-calculated transfer function table. Dependent texture lookup is slightly slower than the pre-classification method of paletted textures but achieves far better results.

6.4 Lighting

6.4.1 Per-pixel

Phong illumination is of great use as it enhances understanding of the visualized data, a description of its use is in chapter 3. It is possible to implement phong illumination on consumer graphics cards as well by using pixel shaders, described earlier. Pixel shading allows manipulation of individual texels on a texturemap, this is a critical feature since phong illumination is a per-pixel technique.

The gradient is used as a surface normal for the illumination calculations, just as with the software illumination technique. The graphics card needs to be able to fetch the gradient quickly, and for every pixel, so the only place is can be stored at is in the texture itself. Usually, the gradient (x,y,z) is stored in the RGB channels of a RGBA texture, and the voxel value is stored in the alpha channel. Some scaling and biasing must be performed first given that components of a normalized vector are in the range [-1, 1] while a standard RGBA texture expects values of [0, 255].

With gradients available per-pixel and with light vectors and colours supplied to the graphics card by the main program, the phong illumination equation can now be implemented in a pixel-shader program. This program is currently possible to write in either a low-level assembly language, or in a high-level program such as CG or HLSL (they are essentially the same language, co-developed by Microsoft and Nvidia).

6.4.2 Light maps

Using per-pixel lighting is simple and fast to use in the uncomplicated case where no extra effects are used and where there is only one light source. If there is a need for multiple light sources or a demand for more specific optical effects, then light maps, or environment maps as they are sometimes called, can be useful.

A lightmap is essentially a special kind of texture that has six-sides. Each of these texture sides are mapped onto one of the sides of the volume, covering it completely. The light map can be seen as an omnidirectional picture, containing all the incoming light striking the cube. The light maps are constructed in advance by using image manipulation software for instance, where the lights surrounding the volume are setup. The phong shading model can be implemented with lightmaps, in that case, two maps are needed, a diffuse and specular map. These maps are then loaded into texture units and only a few fragment program instructions are necessary to complete the shading.

Environment maps can give additional effects, specifically a reflection of the environment that is surrounding the volume. Other effects can be chromatic dispersion and the fresnel effect. None of these are perhaps of great importance (or use) for straight volume rendering, but mentioned none the less for completeness.

Lightmaps does have the advantage of supporting an unlimited amount of light sources and a number of extra effects with a minimum of code, but they do have a number of drawbacks as well. First of all, the lights are obviously fixed in place and cannot be moved, this means that the lights are assumed to be placed at a close to infinite distance from the volume. The reflections are not physically correct, and the quality is not as good as per-pixel lighting. Since two more textures are added to the rendering pipeline, the shading in itself will use two texture units, making this approach unusable in certain combinations with slightly older hardware. Finally, Kniss[14] reports that lightmaps are up to three times slower than per-pixel lighting on a GeForce3, probably due to the added texture sampling.

6.5 The future of VR hardware acceleration

Graphics accelerators are evolving at a much faster rate than CPU's at the moment with as short product cycles as six months. Volume rendering is generally helped by the increased speed each generation brings as more advanced effects can be used with a sustained interactive frame rate. Increased memory capacity means that larger datasets

can be rendered. Most graphics accelerators sold today have 128Mb, while many carry 256Mb as well. Next year 512Mb and 1gb cards are going to hit the market.

Foremost though, volume rendering will benefit from advances within GPU technology. As the programmability increases, more clever ideas will come forth to harness the power of new graphics cards. The nvidia geforce3 GPU has only eight register combiners, thus allowing only eight instructions every pass for pixel computations. The latest nvidia accelerator, the GeForce 6800 is capable of performing 65535 instructions as a comparison.

At this point in time, it is very difficult to predict which methods will prove to be the next defining technique. This chapter will take a look at what kind of techniques has been developed in the last year and what can be expected in the near future.

6.5.1 Hardware acceleration of complex optical models

In the chapter about the optical model that is used for volume rendering, it is shown that the common model used by most methods is an approximation that only consider scattering along the viewing ray, thus eliminating the possibility to render effects such as shadows. The approximation is made to increase speed, since evaluating the ray between every sample and the lightsource adds a significant number of calculations. However, Max[23] showed how this calculation can be made, and Kniss et al.[15] presented in 2003 an approach to include many of the excluded optical properties and still retain an interactive framerate by using graphics hardware. The additions they include to the standard volume rendering integral are shadows, a phase function and more complex scattering.

Shadows can be added to equation 3.6 by including the light ray which is described in equation 3.7. In order to reduce the amount of unnecessary operations that the brute force approach produces and to avoid using a space demanding method such as shadow volumes, Kniss et al. proposed to use an iterative approach with simple two-dimensional buffers. In this iterative method, the volume is evaluated step by step by a half-angle slice, the slice is angled half-way towards the light and the view so that it can be rendered to in both directions. Two buffers are used, one is saving the result from the viewers point of view, while the other buffer (the light buffer) holds the result of how much the slice is occluded. In every step the slice is rendered to the view buffer and the result of this is then blended into the light buffer to generate the final slice, shaded slice.

The phase function can be added and evaluated to give a more realistic model of how the light scatters within the volume. It models the distribution of light, in all directions, after a sample is struck by a ray of light. This operation can be implemented by first calculating the dot product between the light ray and the view ray, then use the result as an index to a one-dimensional dependent texture. This phase function has a similar effect to bidirectional reflectance distribution functions.

The addition of shadows and more realistic scattering does have a big impact on the image quality, the results from standard optical methods look very flat and lifeless compared to those generated by a more complex model. The authors of this method claim to have achieved framerates that are about half of standard 3D-slicing. It should be kept in mind that while shadows add more realism to volume rendering, it does not automatically increase the usefulness of the generated images.

6.5.2 Hardware raycasting

A hardware accelerated raycasting algorithm was proposed by Krüger and Westman[16] last year. While raycasting in the software sense is used to achieve the highest possible picture quality, Krüger and Westman are using raycasting in hardware as a means to accelerate the rendering. As was mentioned in the section about the 2D and 3D techniques, they are brute force implementations that are performing calculations on all samples. The hardware raycasting method proposed by the authors includes two optimizations that are used with software raycasting as well, empty space skipping and early ray termination.

Raycasting in software is computationally enormously costly, and because of that it might seem strange that raycasting is used as a speed optimization in a hardware implementation. The slow speed for software raycasting can be attributed to the rays being computed serially. The fragment shader units on a graphics card can drastically reduce the rendering cost, however, since it has the ability to compute the rays in parallel. This is of course a huge advantage, being able to evaluate all rays at once at the cost of only a single ray.

There is a number of key features on ATI 9000-level hardware makes the optimization methods of empty space skipping and early ray termination possible, and it is foremost the early z-test. The test allows fragments to be discarded before they reach the shader program, thus making sure the fragment will not require any additional computations, shading and blending in particular. Secondly, the ability to render intermediate results to a texture is also critical.

The basis for the technique is built upon the work of Purcell et al.[35] who in 2002 noted that the parallel structure of modern graphics accelerators might work well as a raytracing accelerator. The rendering algorithm can be seen as a stream running through the different stages of the rendering pipeline for each step along the ray, the results from each step is temporarily saved for use in the next step by rendering the values to a texture. This streaming algorithm is realized by using a multi-pass rendering pipeline.

The algorithm uses the first two passes as a setup stage, where the front and the back of the volume is rendered to two 2D textures, the first used to determine ray entry points and the second to determine the ray direction. After the setup is complete, the algorithm is run over a constant number of passes, as many as there are ray segments. In order to enforce early ray termination, the alpha value is checked to each time. Depending on the rendering mode, the fragment is added to the final texture if it reaches the limit or a predetermined isosurface value. If the fragment is not discarded, it is added (blended) to the temporary texture.

Empty space skipping is realized in software raycasting by using octrees generally. Each node in the tree is covering a certain distance in the data, and min and max values are stored there. This lets the raycaster quickly look up the min and max values for an area and determine weather the whole area is close enough to zero and can consequently be skipped or if it must be evaluated. Such complex data structures are unusable on a GPU, so Krüger and Westman have decided to use a 1/8:th scale 3D-texture to encode the original volume and use it instead of an octree. Every eighth step in the original volume is recorded with min and max values in the miniature texture volume. This means that if an area is deemed to be zero, the next eight steps in the renderer can be skipped for that particular ray.

Lighting can be used with this hardware raycasting implementation also. The lighting stage is performed last, with a simple lookup in a 3D gradient texture combined with subsequent lighting equations in the fragment shader.

The results of this technique are very good. In certain favourable conditions, the authors are reporting frame rates in excess of 20 per second. Overall the rendering speed is about three times the speed of a simple brute force 2D or 3D implementation, certainly making it an interesting option for anyone with access to the latest graphic accelerators.

6.5.3 Hardware raycasting with pre-integration

Roetteger et al.[38] released a similar technique to that mentioned above by Krüger and Westman the same year. This implementation is of particular interest because it uses almost every optimization approach for both speed and picture quality that has been mentioned in this thesis. They are using the GPU to cast rays. However, they are somewhat vague in describing exactly how they accomplish the raycasting. It seems likely that they are using a very similar technique to that of Krüger and Westman. Every ray is setup for each on-screen pixel and is then cast simultaneously (in parallel) into the volume. The rays are sampled at a pre set interval length and then integrated using pre-integration. This raycaster is using fairly advanced optimization techniques as well, such as empty space skipping and early ray termination, meaning fewer samples have to be evaluated. Pre-integration ensures that as few aliasing errors as possible are introduced by the classification function (see classification chapter). Using empty space skipping in conjunction with pre-integration is a new technique which the authors call "Adaptive pre-integration".

To summarize, the technique makes use of parallelisation, empty space skipping and early ray termination for best speed and pre-integration for quality. Roetteger et al.[38] are claiming frame rates on an ATI 9700pro that are in the same region as Krüger and Westman using this technique. The final speed is depending on the nature of the volumetric data of course as well as classification settings. The final picture quality is also higher than the more traditional hardware slicing methods, but not as good as software methods.

6.5.4 Splatting

Splatting was first implemented with texture maps in 1993 by Crawfis et al.[7]. Since then the general algorithm has been improved in many ways, and in 2003 Crawfis et al.[51] shows how splatting can be implemented on modern graphics hardware and also how it can be used on graphics hardware to visualize animations of volumetric flow[52]. Hardware accelerated splatting can be implemented on most graphics cards since it only requires two-dimensional texture support and basic blending.

The algorithm is divided into several steps of which the first is to create the splat footprint and store it in a texture map. In order to be able to render the volume with the standard volume rendering integral and with view-aligned splats, the voxels must be depth sorted on the fly. The sorted lists of voxels must be updated every time the view changes, this calls for an extremely fast sorting algorithm. Crawfis et al. found that the bucket sort algorithm is best suited for the task, it outperformed quicksort with a large margin. After the sorting is done, it is time to start the actual rendering algorithm that is performed for each voxel. A quad is centred over every voxel in the volume, and it is rotated so that it is aligned to the view plane. The footprint is then mapped to the quad according to the data value it possesses. The only operation remaining is to evaluate the volume rendering integral by blending the textured quads back to front.

This implementation is very basic and does still have many of the drawbacks that the original software technique had. Since the footprint is calculated in advance, it means that it uses pre-classification which certainly limits the potential picture quality. The sampling interval is also locked to that of the volume itself as only the voxels are evaluated. There is no lighting available either.

A more advanced splatting algorithm was recently proposed in 2004 by Chen et al.[6]. They show how it is possible to combine a lowpass filter to the Gaussian kernel. The filter is called a EWA, elliptical weighted average. They also show how a compression algorithm can be utilized to reduce the memory needed to store the vertices for each quad. Lighting calculations is done in hardware with two passes. While it probably yields better image quality than the one proposed by Crawfis', it still uses pre-classification as an optimization method. Framerates of about three per second is reported.

6.5.5 Frequency domain

Frequency domain rendering is divided into three major steps; pre-computation of the volume data to transform it to the frequency domain, reconstruction via a filter and finally a two-dimensional inverse FFT algorithm transforms the data back to the spatial domain. The pre-computation can be done on the GPU [26], but this step is generally best handled by the CPU. The reason for this is that both the spatial and frequency volume must be loaded into the limited texture memory of the graphics card. Viola et al.[46] proposed a technique in 2004 to use frequency domain rendering on graphics hardware. After the pre-processing, which they perform in software, the reconstruction is done in hardware. They are using higher order filters such as windowed sinc by utilising the fact that the filters are separable to store them as one-dimensional texture maps. The final operation is the inverse FFT which is based on previous work in which the data is first reordered, then butterfly passes is made first row-wise, then column-wise. Since floating point textures are needed to store the frequency volume, this particular implementation is only available on ATI's cards. The speed of this method is impressive, reaching 32 frames per second for a 256^3 dataset with a screen resolution of $512 * 512$ using trilinear interpolation. It does however have the same drawbacks as any frequency domain method; it can only render x-ray type images and the transfer function cannot be changed.

Chapter 7

Implementation

7.1 Implementation details

The program is written in C++ and is using the OpenGL graphical API. Care has been taken to ensure platform and hardware independence, all the tools used in this program are available on several platforms. The shader programs are written in CG, a high level shader language co-developed by NVIDIA and Microsoft. The GUI system used is GLUI, which is a platform independent user interface. The targeted operating system is Microsoft Windows XP, but the program should with only minimal alterations work on both Linux and Mac OS X platforms as well since both OpenGL and CG is available on those. The program has been developed and benchmarked on an Athlon XP2000+ with a NVIDIA Geforce 3.

7.2 Software raycasting

The software raycaster implements two different methods, raycasting based on Levoy's render pipeline using pre-classification, and raycasting with post-classification using a new render pipeline. Three different types of transfer functions are implemented as well, for use with both renderer versions.

7.2.1 Pre-classification renderer

The pre-classification renderer's pipeline is shown in figure 4.1, Levoy's original pipeline can be seen in figure 5.2. In essence, the change is simply that it uses opacity weighted colours and is as such quite identical to the proposed pipeline by Wittenbrink[50].

The renderer is taking advantage of using as many pre-calculations as possible. It starts off by calculating the local gradient at each voxel, the gradient is then fed into the shading algorithm which in turn shades each voxel. The transfer function tables are then calculated and used by the pre-classification procedure. At this point the setup-stage of the program is finished, every voxel in the volume is classified and shaded. The volume is scaled to fit the current viewport size, this is done to ensure the maximum possible picture quality.

From this point on, the renderer casts rays into the volume, one for every pixel. The actual ray traversal is done by simply using the line equation. Samples are taken at a rate that ensures that the sample rate is higher than the nyquist frequency. Finding the

eight voxels surrounding a sample (needed in the next step for trilinear interpolation) is done by first snapping the sample point to the closest upper right hand corner (from seeing down negative z-axis), and then exploiting the spatial relationship of how the voxels are stored to quickly find the adjoining voxels.

For every sample along the ray, it is trilinearly interpolated from the eight closest voxels. In the pre-classification renderer, the RGB components of the colour are interpolated as well as the alpha, totalling four trilinear interpolations per sample. The renderer does have a greyscale mode as well, which brings down the number of interpolations to two.

After the colours and opacity are interpolated, they are fed into the compositing step. The compositing is performed front to back with early ray termination, meaning that as soon as the alpha reaches 100 percent, the ray is terminated.

7.2.2 Post-classification

The post-classification pipeline is quite different from the original pipeline proposed by Levoy. His implementation was treading a balance of picture quality versus computational complexity, and he favoured a fast renderer by choosing to do pre-classification. Quite understandably since the computers of that time still needed almost an hour to render a single picture.

In the post-classification renderer, not as much can be pre calculated as with the pre-classification, but a few things still can. First off, the transfer function tables are calculated and stored for later use during the on the fly classification step that happens during the re-sampling. The gradient is calculated and stored in every voxel as it is needed later on for shading and classification. Some parts of the shading are pre-calculated as well. Since a parallel projection is used, the light is made parallel as well. This means that the equations are constant and can thus be pre-calculated, saving quite a bit of processing time.

The ray traversal algorithm is identical to the one used by the pre-classification algorithm, however, the resampling is quite different. Instead of interpolating the pre-calculated colours and opacity, the post-classification algorithm interpolates the gradient and the density value. The interpolated gradient and density value are then used to shade and classify the sample point. Finally, the sample point is composited front to back with early ray termination.

7.2.3 Computational complexity

Raycasting is a very computationally intensive operation to perform. In theory with a naïve approach, every data value in the volume should be evaluated. A volume with the side N would yield a total of $N * N * N$ sample evaluations, earning it a complexity of $O(N^3)$.

As was noted by Levoy[18], if the screen resolution is $n * n$ pixels with a total of K samples taken per ray, the number of trilinear interpolations are $2Kn^2$ in the simple grayscale mode of pre-classification and $4Kn^2$ in regular pre and post classification. This means that in a typical scenario of colour rendering with a screen size of $512 * 512$ where a 1000 samples are taken along each ray, with 4 interpolations per sample, the computer will have to make a total of 1.048.576.000 interpolations in the worst case scenario where the early ray termination is not used. In the case of post-classification, shading and classification has to be done per sample as well.

The trilinear interpolations are by far the most costly operation during the raycasting. The trilinear implementation uses seven multiplications, subtractions and additions, totaling 21 instructions.

7.3 Hardware Implementation

7.3.1 2D texture slicing

The 2D-texture stack implementation is made very much like the theoretical method that was described in an earlier chapter. Three stacks are made of the original data, one stack for each axis. This means that if the dimensions of the volume are 256*256*113, the texture stack along the z-axis will have 113 slices of 256*256 textures. It becomes slightly more problematic when the x-axis stack is setup with this example volume since textures must be a size of 2^n. The next larger size is 128, so the textures along the x-axis become 256 textures with a size of128*256. This need for padding can become very wasteful unfortunately, since a volume with size 260*260*130 for instance would waste close to 50% of the graphics memory used. A possible solution for this is to use the "rectangular texture" extension that both NVIDIA and ATI support. A popular solution to support larger volumes than the memory actually holds is to use "bricking". This means that the volume is divided into a number of smaller partitions called bricks. Each brick is loaded and then rendered separately. It is a good solution, but it does require a lot of texture loading over the AGP-bus, thus crippling performance. Another solution is to crop the volume to the next smaller 2^n, thereby losing some of the data. Or, simply wait a year for a more powerful graphics card with twice the memory capacity to surface.

After the three texture stacks are loaded to memory, only one of the stacks is rendered to screen. The two other stacks are loaded into the texture memory of the graphics card if they fit into any available memory, otherwise the system will store them in main memory. The algorithm deciding which stack to render is simple, a comparison is made between the components of the view vector. The component with the largest absolute value is also the axis which is facing the camera the most. Whenever the view angle is exceeding 45 degrees, another texture stack is loaded. This stack switching is far from instant, 64Mb of textures can take more than a second to load from system memory via the AGP bus. This might not sound much, but it can be very frustrating when an object is rotated often in angles around 40 to 50 degrees to an axis. Another implementation issue is that the texture management used by the underlying API, in this case OpenGL, can be put in a an less than optimal texture fetching state if an object is rotated back and forth between 40 and 50 degrees.

The only classification used is post-classification. We did attempt pre-classification by changing the volume values before the textures are made just to see the results. But since the post-classification is pretty much free on a graphics card, and has such added flexibility and quality, this render mode was quickly scrapped. Post classification is achieved by using dependent textures. Any one-dimensional or two-dimensional transfer function algorithm can be used with dependent textures. The dependent texture works like a look-up table, the transfer function algorithm writes all possible values within the used intervals to the dependent texture, both colours and opacities. The dependent texture is accessed during the next to last step in the render pipeline, meaning all interpolation of textures is already completed. It is thus fulfilling the post-classification condition. For instance, when a one-dimensional transfer function is used, then RA

dependent texturing is performed. The density value is stored in the alpha channel of the texture and this is the only value used in practice for the lookup. The graphics card evaluates the alpha channel value, and uses this as an index in the dependent texture, and picks out the corresponding colour and opacity and uses these values for any subsequent calculations and/or operations.

The dependent texture offers flexibility since it can be calculated instantly, and can be changed at any time, making interactive changes to the volume possible in real-time. It should be noted also that the colours are opacity-weighted, just as described in the section about opacity-weighting. The gradient is computed and normalized as a pre-processing step because is takes a bit of extra time to perform, about two seconds. It is then stored in the RGB part of the texture as was described in detail in the theoretical section. Only 1D transfer functions are implemented for the 2D slicing method, the reason for this is that the memory requirement would be too high otherwise.

The evaluation of the actual volume rendering integral is performed by enabling an OpenGL operation, `GL_BLEND`. Once enabled, the blending can be set to a number of different modes and thus get results that mimic different software volume render modes. When the blending mode is set as `glBlendFunc(GL_ONE,GL_ONE_MINUS_SRC_ALPHA)`; This function means that `GL_ONE` is multiplied to the source colour, in other words, it is used as it is. This is correct if opacity-weighted colours are used and stored in the dependent texture, and that is exactly what we are using. Then `GL_ONE_MINUS_SRC_ALPHA` is multiplied to the destination colour yielding the equation:

$$destinationcolour = destinationcolour * (1 - sourcealpha)$$

The rendering result is the same as direct volume rendering, every surface is transparent to some degree (decided by the transfer function). Isosurfaces can be rendered by disabling blending and instead use `GL_ALPHA_TEST`.

7.3.2 3D texture slicing

3D textures are not natively a part of many OpenGL implementations, which means that graphics card vendor specific extensions have to be loaded to get 3D texture support. The latest OpenGL 2.0 standard supports 3D textures however.

The slices passing through the volume are supposed to be aligned with the camera with 3D-slicing. Since the camera is rotated around the centred volume, the best solution was regarded to be forming view aligned slices by projecting the camera plane onto the volume. The slices were in this rather naïve first approach the same size as that of the viewport, $500 * 500$. The slices were guaranteed to cover the whole volume, but clamping the texture to the much larger polygon created some strange artefact effects unfortunately. The next approach was to clip the polygons to the bounding box of the texture, and this provided the correct results. The added CPU computations of the clipping do not affect the framerate since the framerate is completely limited by the graphics card.

While the 2D slicing method is limited to only 1D transfer functions, the 3D slicing implementation supports both 1D and 2D transfer functions. A single 3D RGBA texture is used to represent the volume when the 1D transfer function is enabled. The RGB channels are used to store the gradient and the density is stored in the alpha channel. It becomes slightly more complicated with the 2D transfer functions, in that case five values need to be stored, (x,y,z) for the gradient, the density and the magnitude, but a RGBA texture can only store four values. Two solutions can be employed here, either

store the values with reduced precision, 6bit for instance, or use two RGB 3D textures. The latter was chosen in our implementation since it offers greater image quality and simpler fragment programs, but at a 50% higher memory usage. The gradient is stored in one 3D RGB texture, while the density and magnitude is stored in a separate 3D RGB texture. While the increased memory footprint is unfortunate, the separation of classification and shading data into two textures adds some flexibility with render options and ensures that 2D transfer functions can be used on 64Mb cards. A 256^3 volume will consume about 100Mb with 2D transfer functions and shading, but only 50Mb when the shading is disabled. At the same time, smaller volumes with dimensions such as $256^2 * 110$ can be rendered with shading enabled on a 64Mb card.

7.3.3 CG - high level shading language

The lighting calculations are done per fragment by supplying the GPU a fragment shader program. This fragment program can be written in a special assembly language, but the implementation would be specific to a certain chipset with this approach. Using a high level language that can compile programs at runtime is a good way to achieve chipset and platform independence, CG is such a high level language. Programs can be written using ARB paths, these potentially support any chipset that fulfils the requirements of the particular ARB path. The phong shading algorithm that is used in the program is implemented with about 10 lines of code. The CG fragment paths that are available to GeForce3 level hardware are severely limited however, making it rather difficult to fit anything but the simplest phong shading within one pass. Library functions and branching are not available. The power of the language is more apparent when the latest chipsets are targeted with the ARB fragment paths.

Chapter 8

Results

8.1 Image quality

Determining the image quality is a problem without a simple solution. Image quality is not something that can easily be analysed analytically. In contrast, the render speed for example is straightforward to evaluate as it delivers a number that can be easily compared to other render methods. Nevertheless, there are a number of methods that do attempt to measure image quality analytically, commonly by computing the mean square error (MS) or the root mean squared error (RMS). The RMS works as follows; by calculating the average squared difference between the same pixels in the original image and another image and then computing the square root of the earlier calculation. Obviously it is difficult to find an "original" image of a volumetric dataset, since there simply does not exist one. Still, RMS might be slightly useful in the special case when comparing a known superior method to a lesser one, but even that may be reaching. To make matters even worse, images with a low RMS rating is sometimes regarded as having an inferior quality to those with high values[44].

Pommert and Höhne[33] concluded in their evaluation of medical volume visualization methods that the image quality is best evaluated through a human blind study. They also touch on the subject of how to judge weather the image quality is "good enough". An image might have the highest marks in analytical tests, but still fail to convey important information to the observer. The image quality issue can be divided into two aspects, intelligibility and image fidelity. The intelligibility aspect covers how useful the image is to the person who is evaluating it. For example, can the image tell the surgeon what is wrong with the patient? Image fidelity on the other hand covers the technical side, how much the image deviates from a perfect rendering (which can be evaluated with RMS for instance).

Human blind studies are expensive and generally unpractical for most researchers. Likewise is the analytical approach for image fidelity both very difficult to accomplish, its results are questionable as well. In all practicality, this means that researchers cannot objectively judge the image quality, and must therefore use their own judgement. Image quality comparisons in this thesis are consequently based purely on our own estimation.

8.1.1 Quality factors

Factors that are affecting the final rendered image have been investigated throughout this thesis. Their origin, in what parts of the rendering they appear and the possible techniques there are to avoid or minimize their impact. To briefly summarize:

- Sampling errors: The machine or method that is taking the samples might be faulty. In medical imaging, this may be because of a too low resolution or because of an inadequate filter kernel. This is of course outside of any control of the method or rendering program.
- Reconstruction errors: When the volume is resampled during the rendering phase, errors can occur if the sampling theorem is not fulfilled. This might not be possible due to sampling errors. The problem can be minimized however by either using a high quality filter or anti-aliasing.
- Volume rendering model: One does have to bear in mind that the volume rendering integral is only valid under certain circumstances and that these limitations must be observed.
- Render model errors: The actual method that is being used can introduce errors due to its design.
- Classification errors: The classification does have a huge impact on the quality of the final image. Pre-classification is proven to introduce much aliasing to the image while post-classification provides much better results.
- Transfer function errors: The transfer function should be as smooth as possible to avoid aliasing due to added high frequencies. Pre-integration solves much of the problems regarding that.
- Precision errors: The volumes are often sampled with only 8bit precision, thus limiting every colour and opacity values to the same precision.
- Hardware precision errors: The internal precision of floating point variables can be severely lacking on some accelerator chips. Textures often only support 8bit values.

8.1.2 Raycasting

Raycasting is theoretically the best algorithm[37] since it models properties of physical light closely. This is evident by the results of the rendering implementation too, since it yields much better quality than the texture based implementation. The raycasting implementation produces very good results in post-classification mode. The images are smooth yet sharp enough to make out small details. The best results are no doubt achieved when Levoy's two-dimensional region boundary surface transfer function is used with multiple surfaces on a medical volume. Even though the region boundary surface transfer function is tailor made for medical data, it performed just as well on a few other types of data too, such as a synthetic volume and a fuel injection part. The second two-dimensional transfer function, isovalue contour surfaces generates very good results also. It is however much more sensitive to the chosen input values.

It should be noted that while the image fidelity i.e. the technical quality is very high, the intelligibility is highly dependent of the input values given to the transfer function.

On some datasets it is easier to find surfaces, while on others it is easier to find structures within the interior.

The pre-classification mode is in contrast giving varying results. The quality is poor with the region boundary surface transfer function when the transparency is high, small details are omitted and gaps caused by intra slice distances are exaggerated. These problems are not as evident with isovalue contour surfaces, or when the volume is mostly opaque. The isovalue contour surfaces transfer function is less concerned with fine details, hence the quality difference to post-classification is smaller. Overall, pre-classification delivers an image quality which can only be said is ok at best and quite poor in the worst case.

8.1.3 Hardware

The picture quality is overall quite good with hardware rendering. The simpler modes such as 2D texture slicing clearly have the worst picture quality, the filtering during reconstruction is not up to the task. The one-dimensional transfer functions are not helping either, the images contain holes sometimes and generally generate a grainy quality to surfaces. The 3D texture slicing modes rival and sometimes even exceed the quality of raycasting. This is probably in part due to it being possible to find good transfer function values that suit the renderer faster. The images are not as crisp as the ones created by raycasting, they generally have a much higher amount of blur which is appealing to the eye but might not add any image fidelity. Small details are much more visible with raycasting. This becomes evident with the surface rendering transfer function mode especially since it produces thin surfaces only. The hardware's trilinear sampling does have trouble to sample these surfaces correctly.

8.2 Performance

8.2.1 Raycasting

There are a number of implementation-independent factors to consider when it comes to render speed. The contributing factors are:

- Volume size: increased size also increases the number of samples.
- Data properties: data with much empty space fare better with optimization techniques like empty space skipping for example.
- Transfer function: the type and settings used is very influential on optimizations such as early ray termination.
- Desired image quality: since the computationally most expensive operation is the calculation of samples, the speed can be easily increased by either lowering the sample frequency or lowering the image resolution.
- View point: some angles expose more of the objects volume than others, making the viewing angle a big factor.

Table 8.3 shows the render time with no early ray termination, this means that in this case all sample points along the rays that are within the volume are evaluated. Pre-classification and post-classification both do four trilinear interpolations per sample,

Render mode	Transfer function	Screen res	Samples	Seconds
Pre BW	Surfaces	$500 * 500$	96820855	104.6
Pre BW	Isovalue	$500 * 500$	110187646	115.7
Pre BW	Surfaces	$648 * 648$	162772369	173.0
Pre	Surfaces	$500 * 500$	96820855	129.8
Pre	Isovalue	$500 * 500$	110187646	144.1
Pre	Surfaces	$648 * 648$	162772369	223.3
Post	Surfaces	$500 * 500$	95169921	158.6
Post	Isovalue	$500 * 500$	110054277	179.6
Post	Surfaces	$648 * 648$	160021312	259.2

Table 8.1: Results from the engine dataset, $256^2 * 109$, set with high transparency and with early ray termination enabled.

Render mode	Transfer function	Screen res	Samples	Seconds
Pre BW	Surfaces	$500 * 500$	51537627	59.9
Pre BW	Isovalue	$500 * 500$	54749036	63.5
Pre	Surfaces	$500 * 500$	51537627	74.6
Pre	Isovalue	$500 * 500$	54749036	76.5
Post	Surfaces	$500 * 500$	51778596	86.9
Post	Isovalue	$500 * 500$	54501375	92.7

Table 8.2: The engine dataset, $256^2 * 109$, early ray termination is enabled with more opaque transfer function settings.

Render mode	Transfer function	Screen res	Samples	Seconds
Pre BW	Surfaces	$500 * 500$	112068386	119.3
Pre BW	Isovalue	$500 * 500$	112068386	116.7
Pre	Surfaces	$500 * 500$	112068386	149.9
Pre	Isovalue	$500 * 500$	112068386	145.5
Post	Surfaces	$500 * 500$	112068386	186.1
Post	Isovalue	$500 * 500$	112068386	181.5

Table 8.3: Once again the engine dataset, $256^2 * 109$, here early ray termination is disabled, set with high transparency.

Render mode	Transfer function	Screen res	FPS
2D	1D TF	$500 * 500$	11.0
3D	1D TF	$500 * 500$	4.5
3D	2D surfaces	$500 * 500$	4.5
3D	2D isovalue	$500 * 500$	4.5

Table 8.4: The engine dataset, $256^2 * 109$, with no shading.

so the 28% speed gain that pre-classification has got is purely a result of the shading calculations that post-classification must perform on the fly. A somewhat surprising result is that the black and white pre-classification mode is only 25% faster than the colour mode. Since both methods use the same pipeline, and the black and white mode performs only two interpolations per sample, it should in theory be twice as fast. One way of interpreting this would be that the program overhead is just as large as the actual interpolating, thus halving the number of interpolations only yields a quarter of the anticipated increase.

The early ray termination is only able to remove 15% of the samples of the scenario used in table 8.1 where the volume is highly transparent. It fares much better in the more opaque volume, shown in table 8.2, where is removes close to 55% of samples.

It is obvious that the render speed is linearly dependent on the number of samples that are taken. When the resolution is increased to 648^2 is the number of possible samples increased by 68%. Table 8.1 shows that the increased render time in 648^2 compared to 500^2 is roughly equal to that percentage.

8.2.2 Hardware

The hardware implementation has different factors that impact the speed from the raycasting. The 2D and 3D slicing are brute force methods that are evaluating every sample point which means that transparency levels changed by the transfer functions does not impact render speed. There is a high level of sensitivity to the number of texels that are visible on screen.

- Screen resolution: This determines the number of pixels on screen and does have a big impact on speed.
- Viewing angle: Some angles show more texels, which means more rendering has to be done.
- Volume size: More samples leads to slower performance.
- Texture type: 2D textures are faster mostly because the filtering is only bilinear, and it is reasonable to suspect that 2D textures do have a slightly more efficient driver implementation.
- Texture sampling: Each texture sample decreases performance.
- Fragment operations: The number of operations performed per fragment affects performance, especially on older cards.

Table 8.4 shows the results of the engine dataset with no shading and with a couple of different render modes. The 3D render modes all perform alike, regardless of the

Render mode	Transfer function	Screen res	FPS
2D	1D TF	$500 * 500$	10.7
3D	1D TF	$500 * 500$	3.5
3D	2D surfaces	$500 * 500$	0.7
3D	2D isovalue	$500 * 500$	0.7

Table 8.5: The engine dataset, $256^2 * 109$, this time with shading turned on.

transfer function. This is expected also because the number of texture samples are the same, one 3D texture sampling and one 2D dependent texture sampling with no extra fragment calculations. The 2D render mode is more than twice as fast because of lesser filtering and more efficient texture handling.

In table 8.5 is the results showing the same environment as table 8.4, but this time with shading enabled. The 2D and 3D rendering modes with a 1D transfer function only see relatively small changes in framerate. Both implementations only add fragment instructions for shading, the pipeline stays the same otherwise. 3D textures with 2D transfer functions see a big drop in performance which is most likely because it must do one additional 3D texture sampling.

Chapter 9

Conclusions

Volume rendering is a big research field that is has seen a lot of activity over the past decade and will continue to do so in the future with the introduction of better hardware accelerators. The original techniques which all had a number of drawbacks initially have received many updates that have made them faster with better image quality and more versatile. Hardware acceleration of volume rendering is still in its infancy and it is very likely that it will see a similar development as the hardware matures.

While the software rendering techniques have enjoyed very good picture quality, they have also lacked an important quality, and that is interactive framerates. The time to render a single image is sometimes exceeding three minutes, which means that finding a good transfer function configuration is a very time consuming procedure. Getting interactive framerates combined with a good picture quality is a very precious characteristic since the data can be analysed much faster that way.

The applications of volume rendering are many and varied. The driving force is primarily the medical field which can use the rendered images to help diagnose illnesses, or to practice performing virtual surgeries[2]. The special-purpose hardware acceleration cards that are available to these purposes are expensive and are not widely available. Hardware acceleration with general purpose hardware brings volume rendering with the same performance to anyone with a PC which means that both price and availability is far better.

The big challenges ahead are certainly foremost to increase the image quality of hardware rendering so that it reaches the same level of raycasting and to develop more general transfer functions that can find and extract features automatically or at least with a minimum of user supervision. The first challenge is very much tied to the development of better hardware with higher internal precision as well as a more general programmability of the GPU.

References

[1] Tomas Akenine-Moller, Tomas Moller, and Eric Haines, *Real-time rendering*, A. K. Peters, Ltd., 2002.

[2] D. Bartz, M. Hauth, and K. Mueller, *Advanced Virtual Medicine: Techniques and Applications for Virtual Endoscopy*, MICCAI Tutorial T8, 2003.

[3] M.J Bentum, B.B.A. Lichtenbelt, and T. Malzbender, *Frequency analysis of gradient estimators in volume rendering*, Tech. report, University of Twente and Hewlett Packard Laboratories, November 1995.

[4] J. F Blinn, *Light refelction functions for simulation of clouds and dusty surfaces.*, Computer Graphics **16(3)** (1982), 21–29.

[5] Brian Cabral, Nancy Cam, and Jim Foran, *Accelerated volume rendering and tomographic reconstruction using texture mapping hardware*, Proceedings of the 1994 symposium on Volume visualization, ACM Press, 1994, pp. 91–98.

[6] Wei Chen, Liu Ren, Matthias Zwicker, and Hanspeter Pfister, *Hardware-accelerated adaptive ewa volume splatting*, IEEE Visualization 2004, 2004.

[7] R. Crawfis, N. Max, B. Becker, and B. Cabral, *Volume rendering of 3d scalar and vector fields at llnl*, Proceedings of the 1993 ACM/IEEE conference on Supercomputing, ACM Press, 1993, pp. 570–576.

[8] Frank Dachille, Kevin Kreeger, Baoquan Chen, Ingmar Bitter, and Arie Kaufman, *High-quality volume rendering using texture mapping hardware*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware, ACM Press, 1998, pp. 69–ff.

[9] K. Engel and T. Ertl, *Interactive high-quality volume rendering with flexible consumer graphics hardware*, EUROGRAPHICS 2002 (2002), 3.

[10] Simon Haykin and Barry van Veen, *Signals and systems*, John Wiley and Sons, 1998.

[11] Taosong He, Lichan Hong, Arie Kaufman, and Hanspeter Pfister, *Generation of transfer functions with stochastic search techniques*, Proceedings of the 7th conference on Visualization '96, IEEE Computer Society Press, 1996, pp. 227–ff.

[12] J. T. Kajiya and T. Von Herzen, *Ray tracing volume densities*, Computer Graphics (1984), 165–173.

[13] Gordon Kindlmann and James W. Durkin, *Semi-automatic generation of transfer functions for direct volume rendering*, Proceedings of the 1998 IEEE symposium on Volume visualization, ACM Press, 1998, pp. 79–86.

[14] Joe Kniss, Gordon Kindlmann, and Charles Hansen, *Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets*, Proceedings of the conference on Visualization '01, IEEE Computer Society, 2001, pp. 255–262.

[15] Joe Kniss, Simon Premoze, Charles Hansen, Peter Shirley, and Allen McPherson, *A model for volume lighting and modeling*, IEEE Transactions on Visualization and Computer Graphics **9** (2003), no. 2, 150–162.

[16] Jens Krueger and Ruediger Westermann, *Acceleration techniques for gpu-based volume rendering*, Proceedings IEEE Visualization 2003, 2003.

[17] Philippe Lacroute and Marc Levoy, *Fast volume rendering using a shear-warp factorization of the viewing transformation*, Computer Graphics **28** (1994), no. Annual Conference Series, 451–458.

[18] M Levoy, *Display of surfaces from volume data*, Ph.D. Dissertation, Department of Computer Science, The University of North Carolina at Chapel Hill (1989).

[19] Marc Levoy, *Efficient ray tracing of volume data*, ACM Trans. Graph. **9** (1990), no. 3, 245–261.

[20] Mark Levoy, *Error in volume rendering paper was in exposition only*, IEEE Computer Graphics and Applications (2000).

[21] William E. Lorensen and Harvey E. Cline, *Marching cubes: A high resolution 3d surface construction algorithm*, Proceedings of the 14th annual conference on Computer graphics and interactive techniques, 1987, pp. 163–169.

[22] Tom Malzbender, *Fourier volume rendering*, ACM Trans. Graph. **12** (1993), no. 3, 233–250.

[23] N. Max, *Optical models for direct volume rendering*, IEEE Transactions on Visualization and Computer Graphics **1(2)** (1995), 99–108.

[24] Michael Meissner, Jian Huang, Dirk Bartz, Klaus Mueller, and Roger Crawfis, *A practical evaluation of popular volume rendering algorithms*, Proceedings of the 2000 IEEE symposium on Volume visualization, ACM Press, 2000, pp. 81–90.

[25] T. Möller, R. Machiraju, K. Mueller, and R. Yagel, *A comparison of normal estimation schemes*, Tech. report, Ohio State University and Mississippi State University, 1997.

[26] Kenneth Moreland and Edward Angel, *The fft on a gpu*, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware, Eurographics Association, 2003, pp. 112–119.

[27] Klaus Mueller and Roger Crawfis, *Eliminating popping artifacts in sheet buffer-based splatting*, Proceedings of the conference on Visualization '98, IEEE Computer Society Press, 1998, pp. 239–245.

[28] Klaus Mueller, Torsten Möller, and Roger Crawfis, *Splatting without the blur*, Proceedings of the conference on Visualization '99, IEEE Computer Society Press, 1999, pp. 363–370.

[29] Klaus Mueller and Roni Yagel, *Fast perspective volume rendering with splatting by utilizing a ray-driven approach*, Proceedings of the 7th conference on Visualization '96, IEEE Computer Society Press, 1996, pp. 65–ff.

[30] R. Mullicka, R.N. Bryan, and J. Butmana, *Confocal volume rendering: Fast segmentation-free visialization of internal structures*, Tech. report, University of Pennsylvania and National Institutes of Health, Bethesda, 2001.

[31] Z. Nagy, G. Müller, and R. Klein, *Classification for fourier volume rendering*, To appear in proceedings of Pacific Graphics 2004, October 2004.

[32] B.T. Phong, *Illumination for computer generated pictures*, Communications of the ACM **vol 18** (1975).

[33] Andreas Pommert and Karl Heinz Höhne, *Evaluation of image quality in medical volume visualization: The state of the art*, Proceedings of the 5th International Conference on Medical Image Computing and Computer-Assisted Intervention-Part II, Springer-Verlag, 2002, pp. 598–605.

[34] T. Porter and T. Duff, *Compositing digital images*, Computer Graphics **18(3)** (1984), 253–259.

[35] Timothy J. Purcell, Ian Buck, William R. Mark, and Pat Hanrahan, *Ray tracing on programmable graphics hardware*, Proceedings of the 29th annual conference on Computer graphics and interactive techniques, ACM Press, 2002, pp. 703–712.

[36] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl, *Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage-Rasterization*, Eurographics / SIGGRAPH Workshop on Graphics Hardware '00, Addison-Wesley Publishing Company, Inc., 2000, pp. 109–118,147.

[37] Christof Rezk-Salama, *Volume rendering techniques for general purpose graphics hardware*, Ph.D. thesis, Erlangen-Nurnberg, 2001.

[38] Stefan Roettger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser, *Smart hardware-accelerated volume rendering*, Proceedings of the symposium on Data visualisation 2003, Eurographics Association, 2003, pp. 231–238.

[39] J. P. Schulze, M. Kraus, U. Lang, and T. Ertl, *Integrating pre-integration into the shear-warp algorithm*, Proceedings of the 2003 Eurographics/IEEE TVCG Workshop on Volume graphics, ACM Press, 2003, pp. 109–118.

[40] Jürgen P. Schulze and Ulrich Lang, *The parallelization of the perspective shear-warp volume rendering algorithm*, Proceedings of the Fourth Eurographics Workshop on Parallel Graphics and Visualization, Eurographics Association, 2002, pp. 61–69.

[41] R. Siegel and J. Howell, *Thermal radiation heat transfer, third edition*, Hemisphere Publishing, Washington, 1992.

[42] Don Speray and Steve Kennon, *Volume probes: interactive data exploration on arbitrary grids*, SIGGRAPH Comput. Graph. **24** (1990), no. 5, 5–12.

[43] Jon Sweeney and Klaus Mueller, *Shear-warp deluxe: the shear-warp algorithm revisited*, Proceedings of the symposium on Data Visualisation 2002, Eurographics Association, 2002, pp. 95–ff.

[44] Patrick Teo and David Heeger, *Perceptual image distortion*, Proceedings ICIP-94 (IEEE International Conference on Image Processing), vol. 2, 1994, pp. 982–986.

[45] Takashi Totsuka and Marc Levoy, *Frequency domain volume rendering*, Proceedings of the 20th annual conference on Computer graphics and interactive techniques, ACM Press, 1993, pp. 271–278.

[46] Ivan Viola, Armin Kanitsar, and Meister Eduard Gröller, *Gpu-based frequency domain volume rendering*, Proceedings of SCCG'04, 2004, pp. 49–58.

[47] Alan Watt and Mark Watt, *Advanced animation and rendering techniques*, ACM Press, 1991.

[48] L. Westover, *Interactive volume rendering*, Proceedings of the Chapel Hill Workshop on Volume Visualization (1989), 9–16, Chapel Hill.

[49] ______, *Footprint evaluation for volume rendering*, Computer Graphics, Proceedings of SIGGRAPH 1990 (1990), 367–376.

[50] C.M. Wittenbrink, T. Malzbender, and M.E. Goss, *Opacity-weighted color interpolation for volume sampling*, Tech. report, Hewlett Packard computer systems laboratory, July 1998.

[51] Daqing Xue and Roger Crawfis, *Efficient splatting using modern graphics hardware*, Journal of Graphics Tools (2004).

[52] ______, *Fast dynamic flow volume rendering using textured splats on modern graphics hardware*, Proceedings SPIE EI 2004 (San Jose, CA.), 2004.

[53] Roni Yagel, *Towards real time volume rendering*, Proceedings of GRAPH-ICON, 1996, pp. pp. 230–241.