UNIVERSITY OF OSLO
Department of Informatics

Multidimensional Transfer Functions in Volume Rendering of Medical Datasets

Master thesis

Tor Øyvind Fluør

February 2006

Abstract

In volume rendering, transfer functions are used to map voxel property into color and opacity. The most common voxel property used in transfer functions is the voxel intensity. Multiple voxel properties can also be used, and we then get a multidimensional transfer function. In this thesis, we want to test how well a two-dimensional transfer function performs, compared to a transfer function of just one dimension. To test this, we have implemented a fast volume rendering application using GPU shader programming.

We got a radiologist, a surgeon and a computer engineer to evaluate our application using both one and two-dimensional transfer functions on different datasets. The test shows that a transfer function of both voxel intensity and gradient magnitude is better than a transfer function of just intensity for reducing noise in the rendering. The test also shows how difficult manual transfer function manipulation can be.

Acknowledgments

I would like to thank my supervisors Eigil Samset and Johan Seland, for their guidance through the work on this thesis. I would also like to thank Rasmus Bording for all his help with the GPU programming, Øystein Tjentland for much appreciated proof-reading, and all the people at The Interventional Centre that have helped me with various practical aspects of this thesis. Finally, a special tank to my wife Inger Helmine, for all her love and support.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Volumetric datasets are common output from both simulations, like weather predictions and oil reservoir calculations, and from medical examination with machines like computed tomography (CT), magnetic resonance imaging (MRI), and 3D ultrasound scanners. These volumetric datasets consist of a large number of numerical values, which are hard for humans to understand and use. To make use of such datasets, we can use a computer to visualize the data. One way to visualize volumetric datasets is by using volume rendering.

Volume rendering, in contrast to other volume examination techniques like cut-planes and iso-surface rendering, visualizes every voxel[1] in each rendering. The user decides, through a user interface, what opacity and color each voxel should be assigned. Usually all voxels that have a certain property, like an intensity value, is assigned the same color and opacity. This assignment is called classification and is usually performed using a transfer function.

When two voxels in a volume are located in two different materials, but contain the same intensity value, it is impossible to construct a classification that contains only one of these voxels using a simple one-dimensional transfer function of voxel intensity. This is often a problem in medical datasets. To construct a better classification in such a case, the transfer function needs to consider more voxel parameters. These parameters can be used as additional axis in the transfer function forming a multidimensional transfer function.

[1] A voxel is a volume element, the 3D equivalent of a pixel.

1.1 Objective and Scope

The objective of this thesis, is to examine how multidimensional transfer functions can be used to improve classification in volume rendering. We would like to find out how a multidimensional transfer function can be manipulated, and how well it performs compared to a one-dimensional transfer function.

In this thesis, we will focus on volumes from medical datasets, obtained by CT and MRI scanners. Most of these volumes are sampled on a rectilinear, static grid. Other types of grids, like unstructured grids and time dependent grids are interesting, but are beyond the scope of this thesis.

We will not go into details on regular 3D graphics, as we assume that the reader has a basic understanding of this field. For a good overview of 3D graphics, see Real-Time Rendering by Akenine-Möller and Haines [2002].

Chapter 2

Volume Rendering

In this chapter, we will look at some of the theory concerning volume rendering. The first section, give an introduction to what volume rendering is. In Section 2.2, the volume rendering pipeline is introduced, as well as the physical interpretation of volume rendering. Then, in Section 2.3, an overview of the most common volume rendering algorithms is given, before interpolation is presented in Section 2.4. In Section 2.5, we will discuss shading and illumination of volumes. In Section 2.6, classification is introduced, and the last Section (2.7), gives an overview of gradient estimation.

2.1 Introduction

When examining three-dimensional datasets, one can choose from at least three different approaches. The method primarily used today when examining medical three-dimensional datasets, is "slice by slice". In the "slice by slice" technique, the three-dimensional dataset is split into a series of two-dimensional images. Each image is examined individually, and the typical computer application lets the user scroll back and forth through these images slices. The positive thing about this approach, and probably the reason why it is still in use today, is that the data is not modified before viewing. This enables the viewer to trust the presentation of the data, and enables medical professionals to draw conclusions based on MRI and CT images. The downside to this approach is that the magnitude of the data can be overwhelming. Since the user only sees one of probably a couple of hundreds slices, it can be difficult to get a clear idea of the whole dataset.

The next approach is indirect volume rendering. Indirect volume rendering tries to approximate regions in the volume using conventional geometric primitives, that can be rendered using conventional graphic hard-

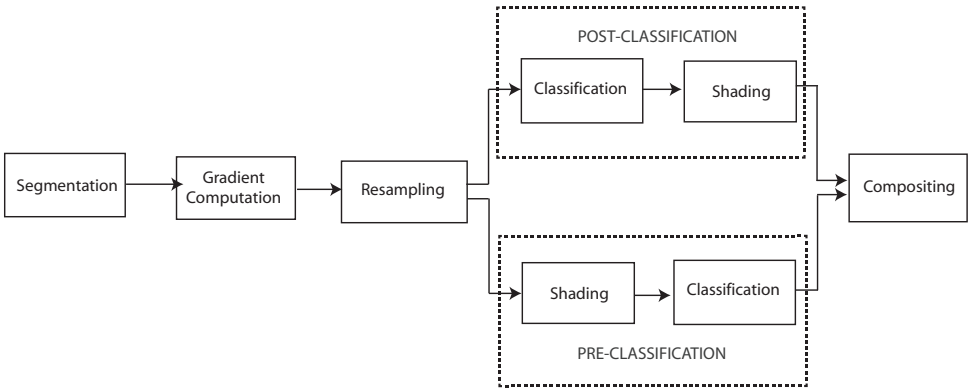

Figure 2.1: The volume rendering pipeline.

ware. The most common way to do this is to assign an iso-value and use an algorithm, like marching cubes [Lorensen and Cline, 1987], to extract an iso-surface. In many cases, especially for more complicated volumes like in medical datasets, it is impossible to specify a single iso-value that can extract the desired surface. More iso-surfaces may be applied, but too many iso-surfaces will clutter the visualization. Another problem is that an iso-surface only represents a small portion of the actual volume.

Volume rendering, which is also referred to as direct volume rendering, is the technique we will use in this thesis. With volume rendering, all the data-points in the three-dimensional dataset are assigned a color and opacity. The voxels are then projected into the two-dimensional images, forming the visualization. In contrast to indirect volume rendering, is volume rendering able to visualize the whole volume dataset. In the remaining sections of this chapter, the essential parts of volume rendering will be explained.

2.2 Volume Rendering Framework

In this section, we will go into the theory behind volume rendering. We will first give an overview of how the different parts of volume rendering relates to each other, before we go into the physical interpretation of volume rendering.

2.2.1 Volume Rendering Pipeline

The volume rendering pipeline, see Figure 2.1, pictures how the different processes in volume rendering relate to each other. In this section, we will give a quick overview of what each step means. The most important steps will be explained in more detail in the remaining sections of this chapter. For a fuller overview of volume rendering, see Lichtenbelt et al. [1998].

The first step in the volume rendering pipeline is segmentation. This is an optional step where a group of voxels is labeled. This is done so that the group as a whole can be assigned an opacity and color during classification. Segmentation is usually performed as a preprocessing step, since it is too time consuming for realtime applications.

The nest step is gradient computation. This is also an optional step, but it is required if we want to shade the volume later in the pipeline. Here the gradient for each voxel is estimated. A more in depth description of gradient computation is given in Section 2.7.

After the gradients have been calculated, the volume is resampled. In the case of raycasting (see Section 2.3.1), resampling occurs whenever a sampled value lies in between the voxels. For samples taken in between two voxels, interpolation has to be performed. Interpolation is the subject of Section 2.4.

The next step in the pipeline is classification. In classification, each voxel sample from the previous step is assigned an opacity and color. The user can decide the opacities and colors through a transfer function. For more about classification see Section 2.6.

After classification, shading is applied to add depth perception to the rendering. How shading is performed in volume rendering is discussed in Section 2.5.

The last step in the pipeline is compositing. Here all the samples with their respectfully opacities and colors are projected onto the screen and blended. This is discussed in Section 2.2.3.

As we can see from the Figure 2.1, there are two possible paths through the pipeline. The two different paths are called pre-classification and post-classification. In pre-classification, the volume is classified before it is sampled. In post-classification, the volume is classified after it is sampled. The difference between these two approaches is that with pre-classification the voxel values are interpolated, while with post-classification the color and opacity values are interpolated.

2.2.2 Optical Model

We would now like to present an optical model, which is a physical interpretation of volume rendering. The most common optical model for volume rendering is the absorption plus emission model. In this model, the volume itself emits and absorbs light, and this absorption and emission is evaluated at each voxel throughout the volume. Other physical phenomena, like scattering of light, are neglected. The reason for choosing a model of only absorption and emission, is primarily that it is easy to implement. Implementing an optical model that also includes scattering is a complex task. The resulting volume renderer would also be very computationally demanding. Another point is that in scientific volume rendering clarity and unambiguity is often of great importance. For such renderings, scattering and shadowing effects might lead to an incorrect interpretation of the volume data.

From the physics of optics we know that the absorption of light radiated from the point $\vec{x}$ in direction $\vec{n}$ with frequency v consists of two terms,

$$\mathcal{A}(\vec{x}, \vec{n}, v) = \mathcal{K}(\vec{x}, \vec{n}, v) + \sigma(\vec{x}, \vec{n}, v)$$

Her $\mathcal{K}$ is the conversion of radiant energy into thermal energy called the source term, and σ is the scattering of light, changing the direction $\vec{n}$ and the frequency v. Since our model does not account for scattering, σ and v can be neglected and we get

$$\mathcal{A}(\vec{x}, \vec{n}) = \mathcal{K}(\vec{x}, \vec{n})$$

In addition to absorption, the volume may emit light. The emission of light is quite similar to absorption in that it has a source term q and a scattering part j,

$$\mathcal{E}(\vec{x}, \vec{n}, v) = q(\vec{x}, \vec{n}, v) + j(\vec{x}, \vec{n}, v)$$

As before, we neglect the scattering, and we can write the absorption as,

$$\mathcal{E}(\vec{x}, \vec{n}) = q(\vec{x}, \vec{n})$$

The equation of radiative transfer can now be written as the ordinary differential equation

$$\frac{\partial}{\partial s} I(\vec{x}, \vec{n}) = -\mathcal{A}(\vec{x}, \vec{n}) I(\vec{x}, \vec{n}) + \mathcal{E}(\vec{x}, \vec{n})$$

Solving this ordinary differential equation we get

$$I(a,b) = I_0 e^{(-\int_a^b \mathcal{A}(\vec{x},\vec{n})dt)} + \int_a^b \mathcal{E}(\vec{x},\vec{n}) e^{(-\int_s^b \mathcal{A}(\vec{x},\vec{n})dt)} ds$$

The first part of this integral is an initial condition, it is the initial intensity a ray of light contains before the ray hits the volume. This is a sort of "ambient intensity". If we set the initial intensity to zero, we get the volume rendering integral

$$I(a,b) = \int_a^b \mathcal{E}(\vec{x},\vec{n}) e^{(-\int_s^b \mathcal{A}(\vec{x},\vec{n})dt)} ds \tag{2.1}$$

Here $\mathcal{E}(\vec{x},\vec{n})$ describes the illumination model and $\mathcal{A}(\vec{x},\vec{n})$ describes the rate which the light is occluded per unit length, which is nothing more that the transparency of the voxel.

Since (2.1) is a continuous integral it is not well suited for evaluation by a computer, so it has to be approximated. If we substitute $\mathcal{E}$ with C describing the color of a voxel, and $\mathcal{A}$ with T describing the transparency, we can approximate the integral with the following Riemann sum

$$I(a,b) = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} T_j$$

In computer graphics, it is often more appropriate to use opacity α instead of transparency, so we can substitute T with $(1-\alpha)$

$$I(a,b) = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1-\alpha_j) \tag{2.2}$$

We now have a model that we can convert directly into a volume rendering algorithm. We will see how this is done in Section 2.3.

[Max, 1995] [Hadwiger et al., 2002]

2.2.3 Composing

The Riemann sum in equation (2.2) blends the voxel values along a ray of light. In volume rendering, these light rays are emitted from each pixel in the image plane, and the blended voxel values forms the resulting pixel values. This blending is called composing. There are two different approaches to composing; either front to back or back to front.

Front to Back

In front to back composing, the samples are evaluated in forward order along the viewing ray. This is a direct implementation of the volume rendering integral approximation we saw in equation (2.2). This can be written as the following algorithm

$$C_{out} = C_{in} + (1 - A_{in}) \cdot A_i \cdot C_i$$
$$A_{out} = A_{in} + A_i \cdot (1 - A_{in})$$

where C_{in} and A_{in} are the composed color and alpha values from the previous steps, C_i and A_i are the sampled color and alpha values, and C_{out} and A_{out} are the new composed color and alpha values.

As the alpha value approaches 1.0, the remaining sampled color values have very little impact on the resulting color. This fact can be used to implement an optimization. When the alpha reaches a predetermined value close to 1.0, we can end the composition, saving precious rendering time. This cut off is called *early ray termination*. [Lichtenbelt et al., 1998]

Back to Front

Another composing method is back to front. The samples are here evaluated in reverse order. If we rewrite the equation (2.2) for back to front composing we get

$$I(a, b) = \sum_{i=0}^{n} C_i \prod_{j=i+1}^{n} (1 - \alpha_j)$$

This gives us the following back to front algorithm

$$C_{out} = C_i \cdot A_i + C_{in} \cdot (1 - A_i)$$

As we can see from the algorithm, it is quite similar to front to back composing. One advantage with back to front composing over front to back is that it does not need to keep track of the composed alpha value. This means that back to front is slightly faster than front to back composing. The problem though, is that back to front composing is unable to do early ray termination. [Lichtenbelt et al., 1998]

2.3 Volume Rendering Algorithms

In this section, we will look at the three most popular algorithms used in volume rendering. We can divide these algorithms into two categories; object order and image order algorithms. In an object order algorithm, each voxel in the volume is projected onto the image, giving color and opacity to the pixels. In an image order algorithm, the volume is sampled along rays cast from each pixel in the final image. The image order algorithms give the most physically correct rendering, but they have been too slow for realtime use. The object order algorithms on the other hand, are much faster, but suffer from lower image quality.

2.3.1 Raycasting

The raycasting algorithm is built on ideas from raytracing. Raytracing is a rendering method used in computer graphic to produce computer images of high quality. To be able to see an object in the real world, a ray of light from a light source has to be reflected of the object and into our eyes. Raytracing mimics this by tracing the ray of light from the eye, or image, backwards to the object, and back to the light source. Volume rendering raycasting can be thought of as raytracing of a light emitting cloud.

Raycasting is an image order algorithm that can be seen as a straightforward numerical evaluation of the volume rendering integral in equation (2.1) on page 7. Because raycasting accurately models the physical transport of light, images rendered with raycasting is often used as a reference in comparison with other algorithms. Raycasting works like follows.

For each pixel in the final image, one or more rays are cast through the volume. The volume data is sampled at evenly spaced intervals along each ray. If a sample is not located at the exact location of a voxel, which is mostly the case, trilinear interpolation is used to calculate the sample value. After interpolation, each of the ray samples is mapped to color and opacity via a lookup table. As a last step, all of the colors and opacities at each ray are composed into a single image pixel value of color and opacity. This composition can be performed either front to back or back to front.

Some optimization can be introduced to the algorithm. One of the simplest is early-ray termination. It terminates the ray when the composed opacity is close to one. Another optimization method is empty space skipping which generates geometry based on the transfer function that cuts away voxels that has zero opacity. Scharsach [2005], show an effective implementation of raycasting that uses both thesis optimizations. [Hadwiger et al., 2002] [Rezk-Salama, 2001]

Even with optimization, raycasting is a rather slow algorithm. The main reason for this is the huge number of interpolations that has to be performed for each redrawing of the image. One way of handling this is to use graphics hardware to do the interpolation, either with special raycasting graphic cards or by using programmable shaders that can be used on modern graphics hardware. An overview of programmable graphics hardware is given in Chapter 4.

2.3.2 Shear-Warp

The shear-warp algorithm, tries to reduce the huge amount of trilinear interpolations in raycasting by cleverly placing the sample points along the viewing ray. This is done by slicing the volume into two-dimensional planes. Then the volume slices are sheared according to the angle of the image plane, so that the sampling rays are perpendicular with the slices. The rays can then sample the volume at the planes using bilinear interpolation. These samples are composed into an intermediate image plane called the base plane, and the base plane is warped to produce the final image.

To enable interactive rotation of the volume, three stacks of volume slices have to been kept in memory at all times, one stack for each primary direction. Only the stack where the normal of the slices are closest to the viewing ray is visible. [Rezk-Salama, 2001] [Hadwiger et al., 2002]

2.3.3 Texture Based Algorithms

Using graphics hardware for volume rendering is generally a good idea, since graphics hardware has built in support for interpolation. Interpolation is one of the things that make volume rendering a computer demanding task. The problem is, that the graphics pipeline only supports rendering of polygonal primitives. One solution to this problem is to render a proxy geometry that can be textured with the volume data.

For graphics cards that only supports 2D textures, the proxy geometry has been three stacks of planes, one for each primary direction where only one is shown at the time. This approach is similar to shear-warp. Each slice has a two-dimensional texture map with the corresponding data values. The slices are drawn back-to-front to produce the final image. The main problem, among others, with this approach is the huge amount of texture memory that is used for the volume textures.

3D texture memory solves this problem. With 3D texture support, the

whole dataset can be loaded into one texture. The proxy geometry can now be drawn parallel with the viewing plane and, as in the 2D texture case, is drawn back-to-front and textured with the volume texture. [Rezk-Salama, 2001]

2.4 Interpolation

Most of the volume rendering algorithms above are dependent on a volume that is continuous, and not only defined at the voxel positions. As all volume data consists of voxel samples, we have to perform a reconstruction of the continuous data. For this we use interpolation.

Interpolation is continuous reconstruction from discrete data. For all points not located exactly at a voxel, the interpolation performs a weighted sum over the neighboring voxels to reconstruct the value at the particular point. If we want to perfectly reconstruct a continuous function from discrete samples, the original continuous function has to be sampled according to the well-known sampling theorem by Nyquist [1928].

Sampling Theorem 2.4.1 *In order for a band-limited (i.e., one with a zero power spectrum for frequencies $\nu > B$) baseband ($\nu > 0$) signal to be reconstructed fully, it must be sampled at a rate $\nu \geq 2B$. A signal sampled at $\nu = 2B$ is said to be Nyquist sampled, and $\nu = 2B$ is called the Nyquist frequency. No information is lost if a signal is sampled at the Nyquist frequency, and no additional information is gained by sampling faster than this rate.*

As we can see from Theorem 2.4.1, we need to have some knowledge of the function we are about to sample. We need to determine the frequency of the function we are about to sample and adjust the sampling rate accordingly. If we do not adjust the sampling rate, we may get sampling artifacts. To illustrate this, we can picture a movie camera that captures a spinning wheel at 35 frames per second. If the wheel spins close, but below, 35 revolutions per second, it will look as if the wheel is spinning slowly in the opposite direction.

In volume data generation, sampling at the Nyquist rate is almost impossible. It might be possible when we know a lot about the object, for instance a volume constructed of a physical simulation. In medical acquired volumes like CT or MRI scans on the other hand, the frequency range of the scanned object is hard to calculate. The equipment used for sampling also poses restrictions on the sampling rate. For this reason we assume that volume data is mostly sampled below the Nyquist rate, which may lead to incorrect reconstruction through interpolation.

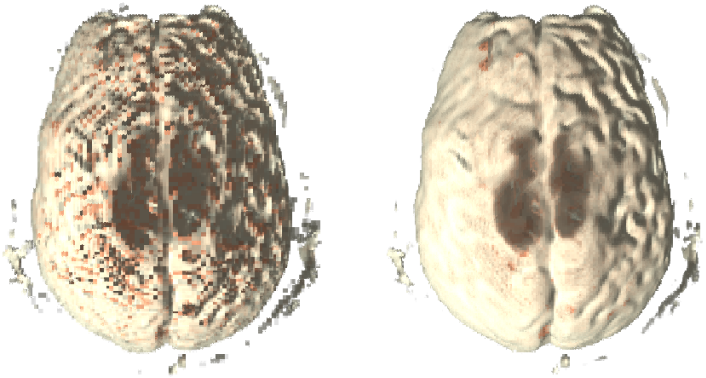

Figure 2.2: Difference between nearest neighbor interpolation (left) and trilinear interpolation (right).

To perform the interpolation we need an interpolation function. The ideal interpolation function is the sinc function.

$$r(x)_{ideal} = \text{sinc}(x) = \frac{\sin x}{x} \tag{2.3}$$

The problem with the sinc function is that it is defined over an infinite spatial interval. It is therefore not suited to be an interpolation filter in a computer. [Bentum et al., 1996]

There is a range of numerical approximation to the sinc function. Some of the most popular are nearest neighbor, linear interpolation, cubic convolution interpolation, and B-spline interpolation. We will present two of these; nearest neighbor and linear interpolation.

Nearest Neighbor

Nearest neighbor is one of the simplest and crudest interpolation filters. As the name indicates, the value of the point to be interpolated is determined by finding the value of the voxel that is closest in space to the sampled point. The value of the sampled point is set to the same value as this nearest voxels.

Nearest neighbor is a very fast but low quality interpolation filter. It suffers heavily from aliasing and staircasing artifacts. An example of a volume rendering using nearest neighbor can be seen in Figure 2.2 (left).

Linear Interpolation

Linear interpolation is another common interpolation filter. It assumes a linear relationship between two sampled values and the points lying in between. The technique can be visualized by drawing a straight line from one sampled value to the next. All interpolated values in between the two samples lies on this line. Linear interpolation can easily be extended to planes and volumes. The interpolation is then called bilinear and trilinear interpolation.

In Figure 2.2 (right), we can see the results of a volume rendering using trilinear interpolation.

2.5 Illumination and Shading

Illumination and shading are common methods for creating an illusion of depth in a two-dimensional image. In this section, we will look at how this is done in volume rendering. We will just look at the simplified version of *local illumination*, where the scattering of light is neglected. First, we will look at the Phong illumination model, which is used in shading of regular 3D graphics.

2.5.1 The Phong Illumination Model

The Phong illumination model is the most commonly used model in shading of 3D graphics, and is used in both Gouraud and Phong shading models. The Phong model describes how light is reflected of a surface. It considers three different light characteristics; ambient, diffuse and specular light. Ambient light is background light that has the same intensity everywhere in the scene. Diffuse light is the light that radiates uniformly in all directions from a light source. Specular light has, in addition to diffuse light, a direction. By using these three light characteristics, we can define the Phong illumination model

$$C_o = C_a k_a O_d + C_p[k_d O_d(\bar{N} \cdot \bar{L}) + k_s O_s(\bar{R} \cdot \bar{V})^n]$$

where C_o is the resulting color, C_a is ambient color, k_a is the ambient reflection coefficient, O_d is the diffuse color of the point on an object, C_p is the color of the point light source, k_d is the diffuse reflection coefficient, $\bar{N}$ is the normalized normal vector, $\bar{L}$ is the normalized light direction, k_s is specular reflection coefficient, O_s is the specular color, $\bar{R}$ is the normalized

reflection vector and $\bar{V}$ is the normalized view point vector. [Lichtenbelt et al., 1998]

2.5.2 Volume Shading

The Phong illumination model uses the normal vector to describe the shape of the surface to illuminate. When we want to use this model in volume rendering, we are not able to use the surface normal since this is not defined for a voxel. Rezk-Salama [2001] states that the gradient vector can be an appropriate substitute for the surface normal. Gradient computation is described in Section 2.7.

In volume rendering, just as in regular 3D graphics, we can make a distinction between Phong and Gouraud shading. If the shading is done before interpolation, that is pre-classification, then the opacity and color values are interpolated and we have volume Gouraud shading. If, on the other hand, the shading is performed after the interpolation, this is post-classification, then the data values are interpolated and we have volume Phong shading. [Lichtenbelt et al., 1998]

2.5.3 Faux Shading

Faux shading or limb darkening [Helgeland and Andreassen, 2004] is a shading technique used in volume rendering to create an illusion of shade. Faux shading works by assigning darker color to low opacity voxels. This can be done easily by ramping the color in the transfer function to black proportionally to the alpha ramping to zero. The effect is a rendering with silhouette edges.

This kind of shading is advantageous because it does not require any extra computations in the rendering phase, but the results may be inadequate for high quality renderings. [Hadwiger et al., 2002]

2.6 Classification

Classification is the process of assigning opacity and color to the voxels. The opacity and color information is then used to render the final image. The common method for this assignment is to use a transfer function. Transfer functions will be discussed in depth in Chapter 3. We will now see where in the volume rendering pipeline the classification is performed.

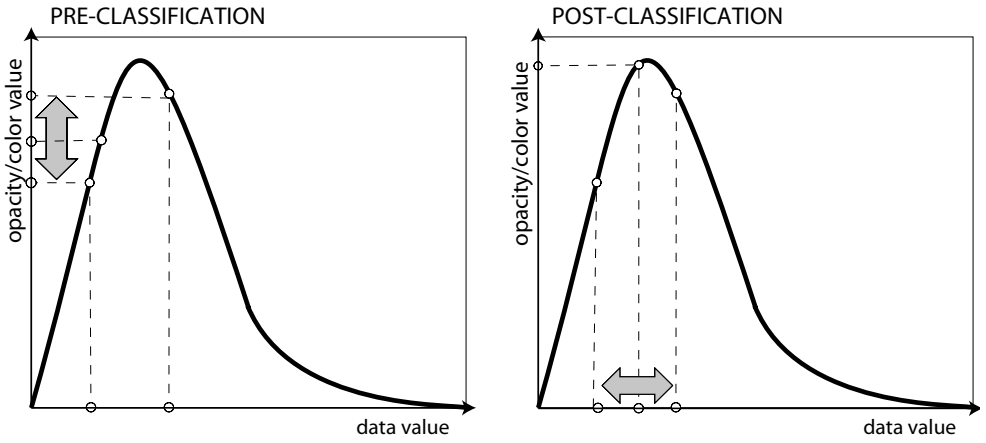

Figure 2.3: Difference between pre and post-classification. The gray arrows indicate interpolation. (Figure courtesy of Christof Rezk-Salama)

Pre-Classification

In pre-classification, the classification step is performed before the interpolation step in the volume rendering pipeline. This means that classification is performed at the voxel positions, and that the opacity and color values are interpolated. Figure 2.3 (left) outlines this concept. In the figure, the classification is performed with a one-dimensional transfer function translating the data values of the x-axis into opacity and color of the y-axis. We can see that an opacity and color value which does not lie on an exact grid point, is determined by interpolating the neighboring opacity and color values. [Rezk-Salama, 2001]

Post-Classification

With post-classification, the data values are interpolated before the classification is performed. The classification process is applied to the continuous signal instead of its discrete sampling points. We can see this illustrated in Figure 2.3 (right). When a sample value lays in between two data values, the sample is calculated using the data values. The opacity and color is then looked up using this interpolated value. [Rezk-Salama, 2001]

Comparison between Pre and Post-Classification

As we can see in Figure 2.3, the pre-classification and post-classification methods lead to different results. Pre-classification modifies the sampling points through the transfer function before reconstruction of the continuous function. This is in violation of the sampling theorem (2.4.1) and will lead to errors in the reconstruction. Post-classification, on the other hand, satisfies the sampling theorem and leads therefore to the most correct images.

One clear advantage of pre-classification is that since the classification is performed before the interpolation, it can be performed as a pre-processing step. Since this step only has to be performed if the transfer function is modified, it can increase the speed of the volume rendering. When, on the other hand, the transfer function does change, this pre-processing step has to be performed again. If the pre-processing step is slow, it may decrease the responsiveness of the volume rendering application.

Another drawback with pre-classification is that it is more likely to produce artifacts. This is because interpolation over opacity and color values do not capture the behavior of the data as well as interpolating over the data values it self.

One last drawback related to pre-classification is that since sample points in a ray is viewpoint dependent, changes in the viewing angle may change the appearance of the rendering. [Rezk-Salama, 2001] [Lichtenbelt et al., 1998]

2.6.1 Segmentation

Even though classification is a great tool for isolating features in a volume, it has certain limits. One limit is that the transfer function is specified in the histogram space of the dataset. If a feature cannot be isolated using a transfer function in the histogram space, it will not be isolated in the rendering.

Segmentation is used to label structures and features in a volume. This is normally done as a pre-processing step before the volume is rendered, and the label is stored as an extra property to each voxel. This label can then be used by the transfer function for precise classification. The labeling can be done either, by manually drawing contours around cross-sectional viewing planes and linking these planes together, or by an automatic segmentation algorithm.

Making automatic segmentation algorithms for medical volumes are

very difficult, and they are dependent on good choices of initial values to work correctly. Automatic segmentation algorithms used in medical images are usually specialized for a certain type of structures like blood vessels. For an overview of segmentation algorithms used for vessel segmentation, see Kirbas and Quek [2004].

2.7 Gradient Estimation

The gradient is a measure of change in a function. In volume rendering, it can be used in both lighting and classification. The gradient is defined as follows.

> The gradient of a three-dimensional intensity function is the partial derivative of that intensity function with respect to all three directions. Given a function $f(x, y, z)$ the gradient is
>
> $$\nabla f(x, y, z) = \left(\frac{\partial f}{\partial x}, \frac{\partial f}{\partial y}, \frac{\partial f}{\partial z} \right)$$

As we saw in Chapter 2.4, a function can be reconstructed exactly by using the ideal interpolation function, sinc. Since the gradient is the partial derivative of the intensity function, we can calculate the exact gradient by using the derivative of the sinc function. In one-dimension the ideal gradient filter is

$$\frac{d}{dx}\left(\frac{\sin \pi x}{\pi x}\right) = \frac{\pi x \pi \cos(\pi x) - \sin(\pi x)\pi}{\pi^2 x^2}$$
$$= \frac{\cos \pi x}{x} - \frac{\sin \pi x}{\pi x^2} = \frac{\cos(\pi x) - \mathrm{sinc}(\pi x)}{x}$$

However, just like with interpolation, the ideal gradient filter is defined over and infinite spatial interval and can therefore not be used to calculate the gradient in volume rendering. Several methods exist to estimate the gradient. Some of the most popular are central difference, intermediate difference, and the Sobel operator. In our application, we use the central difference algorithm, which is outlined below. [Bentum et al., 1996]

Central Difference

The central difference gradient estimator uses the six voxel values surrounding the voxel in $\pm x$, y, and z direction to calculate the gradient.

Formally this can be written as

$$\begin{aligned}\nabla f(x_i, y_j, z_k) \approx & \frac{1}{2h}\left(f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k)\right), \\ & \frac{1}{2h}\left(f(x_i, y_{j+1}, z_k) - f(x_i, y_{j-1}, z_k)\right), \\ & \frac{1}{2h}\left(f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1})\right) \end{aligned} \tag{2.4}$$

where x_i, y_j and z_k defines discrete samples of the function f, and h is the distance between the samples.

As we can see from (2.4), the central difference estimator consists of only three subtractions. This makes it easy to implement in both hardware and software, and makes it reasonably fast. On the downside, central difference is not accurate enough to avoid numerical errors, and it introduces some smoothing. [Lichtenbelt et al., 1998]

Chapter 3

Multidimensional Transfer Functions

In this chapter, multidimensional transfer functions will be investigated. We will look at what different transfer function has been proposed in different papers and articles, and how transfer functions can be created automatically.

In the first section, transfer functions in general will be investigated. In Section 3.2, proposed multidimensional transfer functions will be presented. Then, in Section 3.3, we will take a look at how premade transfer functions can be used, before we in Section 3.4 presents some automatic methods for specifying transfer functions.

3.1 The Transfer Function

We have seen that classification is the mapping from voxel properties to color and opacity. This mapping is often implemented as a lookup table, where a certain voxel property gives a certain color and opacity. This mapping from voxel properties into opacity and color is similar to image manipulation operations performed in the frequency domain. A common user interface for specifying the classification lookup table is the transfer function.

Transfer function specification is normally done by using a one-dimensional function that maps vertex intensity to a color and opacity. However, in some cases this might be inadequate. It might happen that specific voxel intensities are present in two different structures in the volume. If we use a one-dimensional transfer function that only uses the voxel intensity, we are not able to distinctly select just one of the voxels. To broaden our

capability to classify a certain region of a volume, we introduce multidimensional transfer functions.

A multidimensional transfer function uses multiple voxel properties in the classification. These could be more sampled parameters, or some calculated properties. These extra properties are used as extra dimensions in the transfer function. Using multiple data values tends to increase the probability that a feature in the volume can be uniquely isolated in the transfer function domain. While adding more dimensions to a transfer function increases our ability to uniquely classify a region in the volume, it also complicates the classification process. The positive and negative sides of multidimensional transfer functions will be discussed in Chapter 7.

3.2 Proposed Multidimensional Transfer Functions

Many different types of multidimensional transfer functions have been proposed in various papers and articles. This section gives an overview of some of these functions.

3.2.1 Intensity - Gradient Magnitude

Levoy was one of the first to use more than the voxel intensity as a parameter in classification. In his paper [Levoy, 1988], he uses the gradient magnitude as a second parameter in the classification of iso-value contour surfaces. The idea is to let the opacity fade to zero as one moved away from the iso-surface, with a rate inversely proportional to the magnitude of the local gradient vector. [Levoy, 1988]

Many papers have since then extended Levoys approach, using the gradient magnitude as the second dimension in a two-dimensional transfer function. The gradient magnitude is useful as an axis in the transfer function because it enables us to distinguish between homogeneous regions and regions of change. [Kniss et al., 2002]

3.2.2 Intensity - Gradient Magnitude - Second Derivative

In their paper, Kniss et al. [2002] propose, in addition to the two-dimensional transfer function, a three-dimensional transfer function. As dimensions for the three-dimensional transfer function, they use the voxel intensity, gradient magnitude, and the second derivative. The second derivative is a measure of change in the gradient and it can be calculated as follows

$$f'' = \frac{1}{||\nabla f||^2}(\nabla f)^T \mathbf{H} f \nabla f$$

where **H** is the Hessian matrix of second partial derivatives. [Kniss et al., 2002]

3.2.3 Intensity - Distance

Another approach for extending the transfer function is using distance from a focal point. As it is presented by Zhou et al. [2004], a one-dimensional distance transfer function is used to supplement the already existing regular transfer function. The distance to a user decided location is precalculated and inserted as a second parameter for each voxel in the volume. Both the regular transfer function and the distance transfer function are evaluated at each voxel, and their values are combined by multiplication. It is also possible to use the distance to the focal point as an extra dimension in a multidimensional transfer function. [Zhou et al., 2004]

The distance transfer function can be used to set color and opacity like a regular transfer function. It can also be used to constrain another transfer function or transfer function dimension to a region, creating a local transfer function. It is also possible to do volume clipping. This can be done quite easily by setting the opacity to zero where the volume should be clipped.

3.2.4 Principal Curvature

The vicinity of a point on a regular surface can be described by two tangent vectors, principal directions and two corresponding real number, principal curvatures. The principal directions $\vec{s}_1$ and $\vec{s}_2$ tells us where the surface bends the most ($\vec{s}_1$) and the least ($\vec{s}_2$). The principal curvature K_1 and K_2 express how much the normal change in the respectively principal directions. This description yields a unique and view-independent characterization.

In their paper, Hladůvka et al. [2000] discuss using the principal curvature in a transfer function. Their approach uses the curvature pair K_1 and K_2 as axis in the transfer function forming a two-dimensional transfer function. Such a transfer function will be useful to distinguish among different shapes inside the volume. For medical applications, this can be used to select surgical tools or segment structures like bones, vessels, and colon

polyps. The principal curvature can also be used for coloring iso-surfaces, which can reveal how the shape changes inside. [Hladůvka et al., 2000]

3.2.5 Intensity - Light

In their paper, Lum and Ma [2004] discuss a multidimensional transfer function for lighting boundaries between different materials. This transfer function is not used for classification but as parameter to the Phong illumination model.

The Phong illumination model we saw in Section 2.5.1, can be rewritten as

$$C = ColorTF(S)(k_a + k_d MAX(N \cdot L, 0)) + k_s MAX((N \cdot R)^n, 0)$$

where S is the scalar value of the sample, ColorTF() is the transfer function color-map lookup table, N is the normalized gradient direction, L is the light direction, R is the reflected light direction, n is the specular exponent, and k_a, k_d and k_s are the ambient, diffuse and specular lighting coefficients respectively.

The [Lum and Ma, 2004] paper extends this model by adding transfer function lookup tables for the ambient, diffuse, and specular lighting coefficients. This gives us the following expression

$$\begin{aligned} C = ColorTF(S)(&LTF_{k_a}(S_1, S_2) + LTF_{k_d}(S_1, S_2)MAX(N \cdot L, 0)) \\ &+ LTF_{k_s}(S_1, S_2)MAX((N \cdot R)^n, 0) \end{aligned}$$

where $LTF_{k_a}(S_1, S_2)$, $LTF_{k_d}(S_1, S_2)$, and $LTF_{k_s}(S_1, S_2)$ are lookup tables for the ambient, diffuse and specular lighting coefficients respectively. The S_1 and S_2 parameters to the lighting lookup tables are two samples read on both sides of the voxel in the gradient direction. A significant difference in these two samples indicates that the voxel might lay on a boundary. This knowledge can then be used to set appropriate values for the lookup tables, enhancing the visual impression of the boundaries. [Lum and Ma, 2004]

3.3 Premade Transfer Functions

In some volumes, a specific range in the intensity value translates directly to a specific structure in the volume. In CT images, we can use the *Hounsfield scale* to help the transfer function specification. In Table 3.1,

Substance	Approx. value
Bone	80-1000
Calcification	80-1000
Congealed blood	56-76
Grey matter	36-46
White matter	22-32
Water	0
Fat	-100
Air	-1000

Table 3.1: Approximate value of common tissues measured on the Hounsfield scale. (Table courtesy of Kevin Boone)

we can see some examples of what substances different intensity areas in a CT image corresponds to.

Some manufactures of CT and MRI scanners also provide volume rendering software that uses pre-specified transfer functions. With such a transfer function, the user assigns color and opacity to already classified areas. This makes the use of volume rendering easy, but limits the user to a certain type of volumes.

3.4 Automatic Generation of Transfer Functions

Defining a transfer function by trial and error can be a hard and tedious job. When we add more dimensions to the transfer function, the task of manually assigning a transfer function can be overwhelming. This is why specification of the transfer function is listed as one of the major reasons that volume rendering is not being used in day-to-day volume examinations. To deal with this problem, a number of automatic and semi-automatic methods for specifying transfer functions have been proposed. We can divide these methods into two different categories based on their approach. In the first category, Images driven techniques, the focus is on how the transfer function can create the best resulting image. In Data driven techniques, the focus is on how the underlying data can be visualized in the most correct way. We will now look at some of the proposed methods.

3.4.1 Image Driven Techniques

Image driven automatic or semi-automatic transfer functions techniques, concentrate on delivering the best-looking rendering. In their paper, He et al. [1996] treats the search for the optimal transfer function as a parameter optimization problem. This optimization process is approached with stochastic search techniques. They propose two different user interfaces. Either, the user can choose the best rendering in each iteration of the algorithm, or the user can specify some objective goals for the desired rendering. With the first interface, the program generates a number of thumbnail images rendered with the different transfer functions. The user selects the best rendering, and the algorithm generates more transfer functions based on the user decision. With the second interface, the algorithm is run without user input until a satisfactory rendering has been achieved.

3.4.2 Data Driven Techniques

When using a data driven approach to automatic or semi-automatic generation of transfer function, the focus is on presenting the volume data as correct as possible. One such data driven method is presented by Kindlmann and Durkin [1998]. In their paper, they outline a method for identifying the boundaries between different structures in the volume automatically. This is done by constructing a histogram volume with dimensions of opacity, and first and second derivative of the opacity in the gradient direction. From this histogram volume, a distance map is created that records the relationship between the voxel opacity and the boundary proximity. The user can control a transfer function that defines how the different boundaries are presented.

Chapter 4

Programmable Graphics Hardware

Volume rendering algorithms are usually very computationally demanding. By using modern graphics hardware, it is now possible to solve general purpose problems, like volume rendering, using the parallel processing powers of the Graphics Processing Unit (GPU). In this chapter, we will present an overview of the GPU and how it is possible to program it using shader programming.

In the first section, a quick introduction to graphics hardware will be made, followed by an overview of the OpenGL rendering pipeline. In Section 4.3, the three most common shading languages are presented. The last section deals with using graphics hardware for general purpose computation.

4.1 The Graphics Processing Unit

The graphics processing unit or the GPU is the processing unit responsible for converting 2D and 3D objects in a rendering scene into pixel on the screen. In the resent years GPU technology has advanced at an enormous pace, doubling the rendering rate (pixels per second) every six months [Fernando, 2004]. The GPU has also seen a number of structural changes, going from a static pipeline with fixed functionality to a more dynamic pipeline with certain programmable steps. As of January 2006, the GPU has two types of programmable processors, the vertex processors and the fragment processors. The program on the vertex processors is run for all vertices and the fragment processors program for all the rasterized fragments (see Section 4.2).

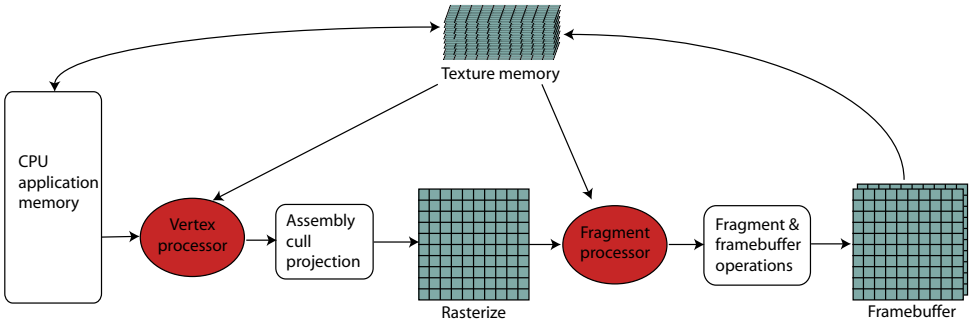

Figure 4.1: The OpenGL pipeline. (Figure courtesy of Johan Seland)

A GPU actually has several vertex and fragment processors that run in parallel. This is usually left out of overview figures to not crowd the figure, but it is important to know. The parallel vertex and fragment processors make the GPU capable of handling vast amounts of computations. In the next section, we will look closer at the OpenGL rendering pipeline.

4.2 OpenGL Rendering Pipeline

The OpenGL pipeline describes how an object in a rendering scene is transformed into pixels on the screen. Figure 4.1 gives an overview of this pipeline.

First, primitives are passed to the vertex processor. If a vertex program has been loaded on to the GPU, the program is executed for each of the vertices. If no vertex program has been loaded, the fixed functionality is run, the vertex and normals are transformed, and texture coordinates are generated and transformed. Each primitive is then clipped and projected, before it is rasterized into fragments[1].

The fragment program is then run for all of the rasterized fragments, if no fragment program is loaded the fixed functionality performs operations like texturing, fog and color summation. As a last step, the fragments are rendered into the framebuffer.

A program that runs on one of the programmable processors on the GPU is called a shader. The shader has to be compiled and linked before it is loaded on to the GPU. The shader program then replaces the fixed functionality in the respective processor.

[1] A fragment is a meta-pixel, with has depth as well as width and height position

4.3 Shading Languages

To efficiently exploit the specialized functionality of modern graphics hardware, a new class of programming languages has been developed.

There are three main competing shading languages: OpenGL Shading Language (GLSL), High-Level Shader Language (HLSL) developed by Microsoft and C for Graphics (Cg) by NVIDIA. All of these implement more or less the same functionality, with only some differences in the syntax of the language. The choice of shading language is mostly based on development platform and hardware. HLSL is only available on Microsoft Windows, and Cg might work best on graphic cards produced by NVIDIA. GLSL, which is a part of the OpenGL 2.0 core library, is multiplatform and should run equally well on all graphics cards that support high-level shaders.

Shading languages has support for common programming language mechanisms like loops and function. They also have native support for many operations common in 3D graphics, such as vector and matrix operations. [Lovesey, 2005]

4.4 General-Purpose Computation on the GPU

General-Purpose computation on GPUs or GPGPU is a field where one is using the GPU to perform calculation normally done on a CPU. The GPU is optimized to perform ever increasing number of triangle and pixel operations per second. With the introduction of programmable vertex and fragment processors, the GPU is in fact a highly optimized *stream processor* that can be used as a co-processor for parallel problems. We will here just introduce the subject of GPGPU. For a good survey paper, see Owens et al. [2005].

The first ting to do when using the GPU as a co-processor is to parallelize the problem. Usually the most suited problems to solve on a GPU consist of some sort of array of data that should be manipulated in some way. The reason for this is that the power of the GPU lies in its number of parallel processors that can do calculations simultaneously. The array is then uploaded into the GPU as a texture. Textures can be accessed from both the vertex and the fragment processor, but texture reads from the vertex processor is more expensive than from the fragment processor. This, and the fact that there are more fragment processors than vertex processors, is the reason most GPGPU programs are written as fragment programs.

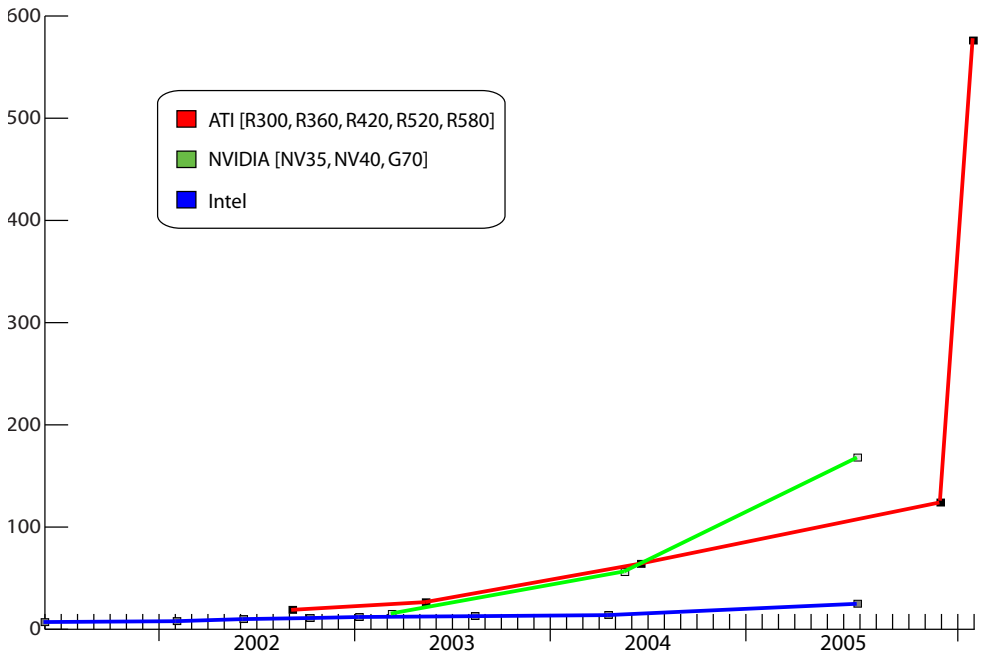

Figure 4.2: GPU and CPU theoretical floating-point performance. (Data courtesy of Ian Buck and Mike Houston)

The next step is to draw some geometry, and after the geometry has been rasterized, the GPGPU fragment program is run for each of the fragments. The output of each fragment program is written to the framebuffer. It is possible to redirect the output from the framebuffer to a texture that can be used as input for another iteration of the program, or to a special buffer that can be read back efficiently by the CPU program.

GPGPU has been used for audio filtering [Whalen, 2005], dense linear system solving [Galoppo et al., 2005], and fast database operation [Govindaraju et al., 2004] to name just a few projects.

The main reason GPGPU has become so popular, is the increased calculation speed. The GPU has seen a great increase in calculation speed the last years, and because of its simpler and somewhat limiting design an increase in transistors has meant a direct increase in processing power. As we can see from Figure 4.2, the theoretical floating-point performance of a GPU is far superior to that of a CPU. This gap is expected to further increase in the future, as new GPUs are developed.

One problem with GPGPU programming is the fact that the GPU cannot be used directly for general purpose processing. All general problems

GPU	CPU
textures	arrays
Fragment program	inner loop
render to texture/buffer	feedback
geometry rasterization	computation invocation
texture coordinates	computational domain
vertex coordinates	computational range

Table 4.1: CPU-GPU analogies. (Table courtesy of Mark Harris)

must first be converted into graphics problems. This means drawing some geometry, and in a way trick the GPU into solving our problem. How this is done for a specific case might not be straightforward.

CPU-GPU Analogies

Since the GPU is a special processor for preparing graphics for the screen, writing general purpose GPU programs is quite different from writing programs for the CPU. It might be useful to draw some analogies from CPU programming concepts to their GPU counterparts. Table 4.1 summarizes these analogies.

In CPU programs one use arrays to store data like CT or MRI acquired volumes. The counterpart to array on GPU is textures. It can be used as the input to a program or the results can be rendered to one. In CPU programs, it is common to have a loop running over an array, performing a certain task. On the GPU, the fragment program is run for each of the pixels of the rasterized fragments. This can be seen as a loop over all the pixels. [Harris, 2005]

Chapter 5

Implementation

5.1 Implementation Overview

To be able to test different multidimensional transfer functions, we have developed a volume rendering application. This section gives an overview of how this program has been implemented. The application consists of two main parts, the volume renderer and the transfer function manipulator. In the volume renderer part of the program, we have implemented a GPU based raycasting algorithm that was introduced by Krüger and Westermann [2003], and further extended by Scharsach [2005]. We will now give an overview of this algorithm.

5.1.1 Raycasting on the GPU

In their paper, Krüger and Westermann [2003] introduce a raycasting algorithm that is fully implemented as a shader program. This algorithm was further extended by Scharsach [2005], which introduced a number of performance optimizations. The algorithm works as follows.

First, the volume dataset is stored in a three-dimensional texture. We then need to calculate one viewing ray for each pixel in the resulting image, which runs through the volume. This can be done by first drawing a bounding geometry, in this case a cube, on a unit interval, and assign each point on the surfaces a color value according to its location in space. This gives us a color cube (see Figure 5.1). Each of the colors can be thought of as vectors from the point $(0, 0, 0)$ up to the point (p_1, p_2, p_3). The viewing ray for each pixel can now be calculated by subtracting the front-face color (or vector) from the back-face color (or vector). We can now use a fragment program to step along each viewing ray, and sample the volume.

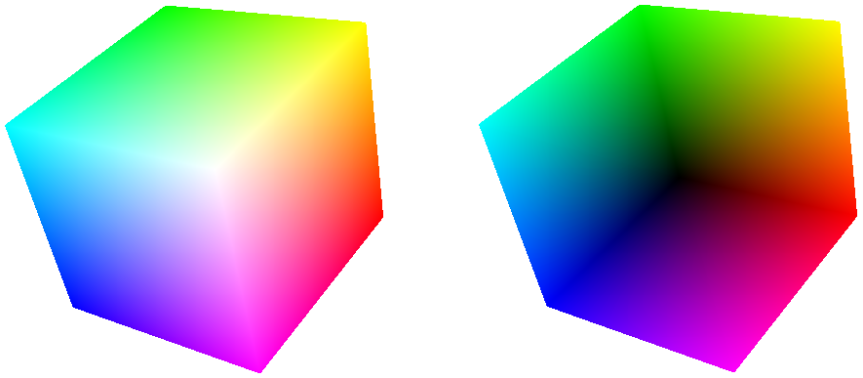

Figure 5.1: Front and back-faces of the bounding geometry.

The algorithm can be summarized in three steps:

1. Draw the front-face of the color cube into an intermediate texture.

2. Draw the back-face of the color cube and subtract the front-face colors from the back-face colors and store the normalized resulting vector and its initial length in a direction texture. These vectors are the viewing rays through the volume.

3. Draw the front-face again. Use the color of each fragment as a starting value. Step along the vectors in the direction texture, and sample the volume at each step. Terminate the ray as soon as the ray leaves the bounding color cube, or if the opacity has reached a certain threshold (early ray termination). Blend the results back to the screen.

It is worth noting that it is possible to perform this algorithm in just two steps. We have used the three-step version, since it is the one used in the both the papers. The power of this volume rendering algorithm, is both speed and high quality. Even though our implementation is not optimized for speed, it has no problem rendering quite large volumes at interactive frame rates. Raycasting has traditionally been known as a slow algorithm. This has mainly been because it heavily relies on three-linear interpolation, and three-linear interpolation computed on the CPU is slow. By loading the volume into a texture, hardware implemented three-linear interpolation is available. This is one of the main reasons this implementation is so much faster than an equivalent implementation on the CPU.

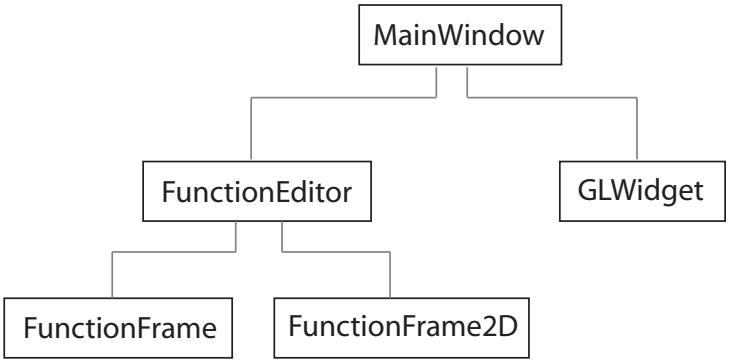

Figure 5.2: The main structure of the volume rendering application.

5.1.2 Implementation

In our volume rendering implementation, we have used the raycasting algorithm described above. To ease the development, Qt and OpenGL has been applied for user-interface and 3D graphics. Qt [Trolltech, 2006], is a multi-platform application development framework, which is well suited for programming graphical user interfaces.

The volume rendering application is divided into two main parts, the raycaster and the transfer function manipulator. Figure 5.2 gives an overview of the main classes in the application. The MainWindow class draws a Qt window, which contains the raycaster widget and the transfer function widget. The GLWidget class contains all the OpenGL code, which is mainly supporting functionality used by the raycaster shader program. Both the FunctionFrame and the FunctionFrame2D classes implement the actual transfer function editor widgets, which is used by the FunctionEditor class.

We will now go into how the two parts of the application is implemented. We start with the volume renderer.

The Volume Renderer

In our application, we have implemented the core algorithm from the previous section. In addition, we have added light calculations and support for one and two-dimensional transfer function. Even though Scharsach [2005] propose many optimizations to the original algorithm, few of these have been implemented. The main reason for this has been limited development time, and the focus on transfer functions instead of rendering

speed.

The light calculations have been implemented using Phong shading as described in Section 2.5. To be able to do Phong shading in a volume, the gradient has to be calculated for each voxel in the volume. The gradients are calculated using central difference, as presented in Section 2.7, and uploaded as a three-dimensional texture to the raycaster shader.

The transfer function lookup is implemented as a one or two-dimensional lookup table. The lookup tables are loaded into the raycasting shader as textures, and are updated when the transfer functions change.

All the volumes that we got from The Interventional Centre were stored in the Dicom file format. Dicom (Digital imaging and communication in medicine)[NEMA, 2006], is a file format that is widely used to store medical datasets. In the Dicom format, each slice is stored in a single file with a comprehensive header with information about both the current slice, and more global information that apply to the scan as a whole. For a simple volume rendering application like ours, the Dicom format contains much information that is not needed. We decided instead to use the PVM volume file format created by Stefan Roettger. The PVM format is simpler to use, as all the volume slices is stored in one file. Another advantage is the large collection of volume datasets available in the volume library on Roettger's web page. [Roettger, 2005]

To improve the quality of the volume rendering, we have implemented an anti-aliasing scheme known as interleaved samples. This is implemented as proposed in [Scharsach, 2005]. The implementation is simple. Instead of letting all the rays start at the surface of the bounding geometry, we give some of the rays a slight offset in the z direction, to reduce sampling artifacts. This can be seen as anti-aliasing in just one direction. Figure 5.3 shows two renderings, one with and one without interleaved samples.

Transfer Function Manipulation

In addition to the volume renderer, we have implemented two transfer function widgets that can be used to manipulate one and two-dimensional transfer functions in realtime. Images of the two widgets can be seen in Figure 5.4.

In the one-dimensional transfer function widget, we have implemented four functions that can be manipulated, one function for each of the colors in the $RGB\alpha$ color model. The user can control the functions by adding control points the function has to pass through. The functions are generated using Bézier curves. The widget also contains a histogram of the

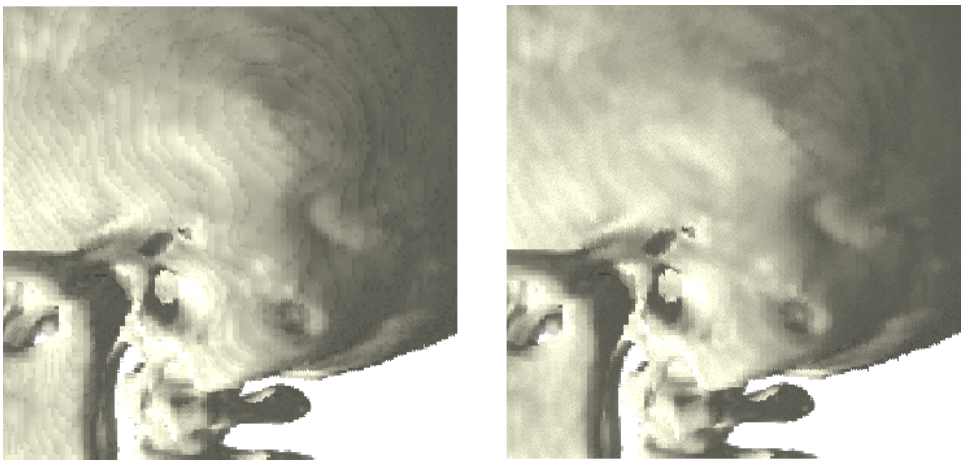

Figure 5.3: Difference between regular image (left) and anti-aliased image using interleaved sampling (right)

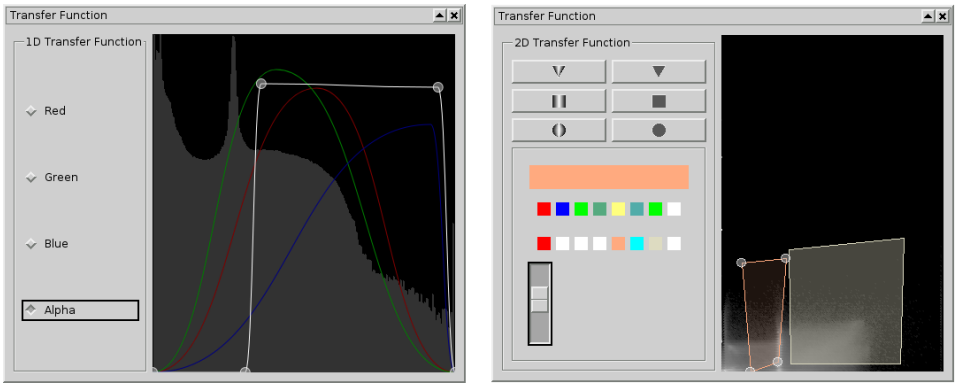

Figure 5.4: The two transfer function widgets that has been implemented.

current dataset that is drawn in the background to help the user to locate possible regions of interest.

In the two-dimensional transfer function widget, the user can use three different geometrical shapes to define the transfer function. The three shapes are a circle, a triangle, and a rectangle. The shapes are assign color and opacity using a color selecting widget. Each shape also contains a set of control points, which enable the user to change the aspect of the shape. Both transfer function widget support saving and loading of transfer function information.

5.1.3 Implementation Tests

One of the advantages with the raycasting algorithm we have implemented is its calculation speed. We will now present the performance tests that we have run on the application.

Volume Rendering Performance Test

We would like to test how well the volume renderer performs on different graphics hardware. The test we have run measures performance in frame per second, which is common in tests of graphics hardware. For this test we have constructed a movement loop that rotates the volume around the x and y-axis. The loop simulates the mouse clicking down the mouse button in the center of the volume, moving 10 pixels in x and y direction, and releasing the button. To ensure correct results, the frame rate reading was measured over a period of five frames.

We ran the test using two different datasets, on four different graphic cards configurations, and on two different computers. The first dataset was a MRI scan of the brain consisting of approximately 8 million voxels (256 x 256 x 122). The second dataset was a CT scan of the chest of a child, consisting of almost 38 million voxels (512 x 512 x 144). The graphics cards that were used are NVIDIA GeForce 6600 GT with 256 MB of memory, NVIDIA GeForce 7800 GTX with 256 MB of memory and a NVIDIA GeForce 7800 GTX with 512 MB of memory. For the last graphics card configuration, we used two NVIDIA GeForce 7800 GTX card with 256 MB of memory in SLI. SLI is short for Scalable Link Interface, and is a technology that enables two graphics cards to be linked together and used as a single device. In theory, this should double the rendering performance. The SLI mode we used was split frame rendering (SFR). This mode splits each frame in two, and renders one part on each graphics card. One of the graphic cards, the GeForce 6600 GT, used in the test was an AGP card

and all the other cards were PCI express cards. Because of this, we had to run the test on two different computers. The computer used for the PCI express cards was equipped with an AMD Athlon 64 dual core processor at 2.0 GHz and 2 GB of system memory. The computer used for the AGP card used an Intel Pentium 4 processor at 2.4 GHz and had 512 MB of system memory. For the test, we use two different transfer functions, a one-dimensional function of voxel opacity and a two-dimensional function of voxel opacity and gradient magnitude. Both functions were set to give just a slight opacity to each voxel. The results of the test are presented in Section 5.3.

Gradient Calculation Test

During the development of the application the gradient calculation was implemented twice, first for running on the CPU, and later reimplemented as a shader program running on the GPU. In both cases the gradients were calculated using the central difference algorithm. Since the exact same algorithm was implemented in both cases, we were able to compare the implementations against each other.

The CPU gradient calculation is performed in three nested loops over the dataset, one for each primary direction. At each voxel, the gradient in all three directions and the gradient magnitude is calculated. In the GPU implementation, the gradients are calculated one slice at the time. The shader program does the gradient calculation, and the result of each rendering is rendered to a buffer, and read back into an array. None of the implementations are optimized for speed, but are straightforward implementations of the algorithm.

The gradient performance test was done by timing how long the respective implementation took to generate all the gradients for three different datasets. The test was run on a computer equipped with an Intel Pentium 4 processor at 2.6 GHz, 1 GB of system memory, and a NVIDIA GeForce 6600 GT graphics card with 256 MB of texture memory. Each test was run three times, and only the fastest run was recorded. The results of the test are presented in Section 5.3.

5.2 Implementation Results

The volume renderer that we have implemented consists of two windows; the volume renderer window, and the detachable transfer function manipulator window. The transfer function window has two different function

GPU	Dataset	1D function	2D function
GeForce 6600 GT 256 MB	Child	7.7	7.0
GeForce 6600 GT 256 MB	MRI cerebrum	8.5	5.6
GeForce 7800 GTX 256 MB	Child	19.8	17.2
GeForce 7800 GTX 256 MB	MRI cerebrum	27.2	17.5
GeForce 7800 GTX SLI	Child	30.1	27.5
GeForce 7800 GTX SLI	MRI cerebrum	31.2	26.2
GeForce 7800 GTX 512 MB	Child	25.1	21.7
GeForce 7800 GTX 512 MB	MRI cerebrum	31.8	22.9

Table 5.1: Volume rendering performance on different graphic cards in frames per second.

manipulators; a one-dimensional and a two-dimensional. A screenshot of the two function manipulators, along with the rendering window, can be seen in Figure 5.5.

The application has three view modes; normal mode, hidden mode and demo mode. In the normal mode, the user is able to change the transfer function, and rotate and zoom the resulting rendering. In hidden mode, the transfer function manipulators are not visible, but the user is able to rotate and zoom the rendering. This mode was implemented for hiding the transfer functions during the survey. In the demo mode, the rendering is rotated continuously around the x and y-axis. The demo mode was used to test the performance of the application.

The application also supports saving and loading of both one-dimensional and two-dimensional transfer functions. The transfer functions are saved in a custom file format that preserves all the function properties.

5.3 Performance Test Results

5.3.1 Volume Rendering Performance

To measure the performance of the volume renderer, we have tested two different volumes on a range of different graphic cards. The test was run two times for each graphic card and volume, and only the fastest was recorded. The test results indicate peak performance over a period of five minutes. The results of the test are summarized in Table 5.1, and discussed in Chapter 7.

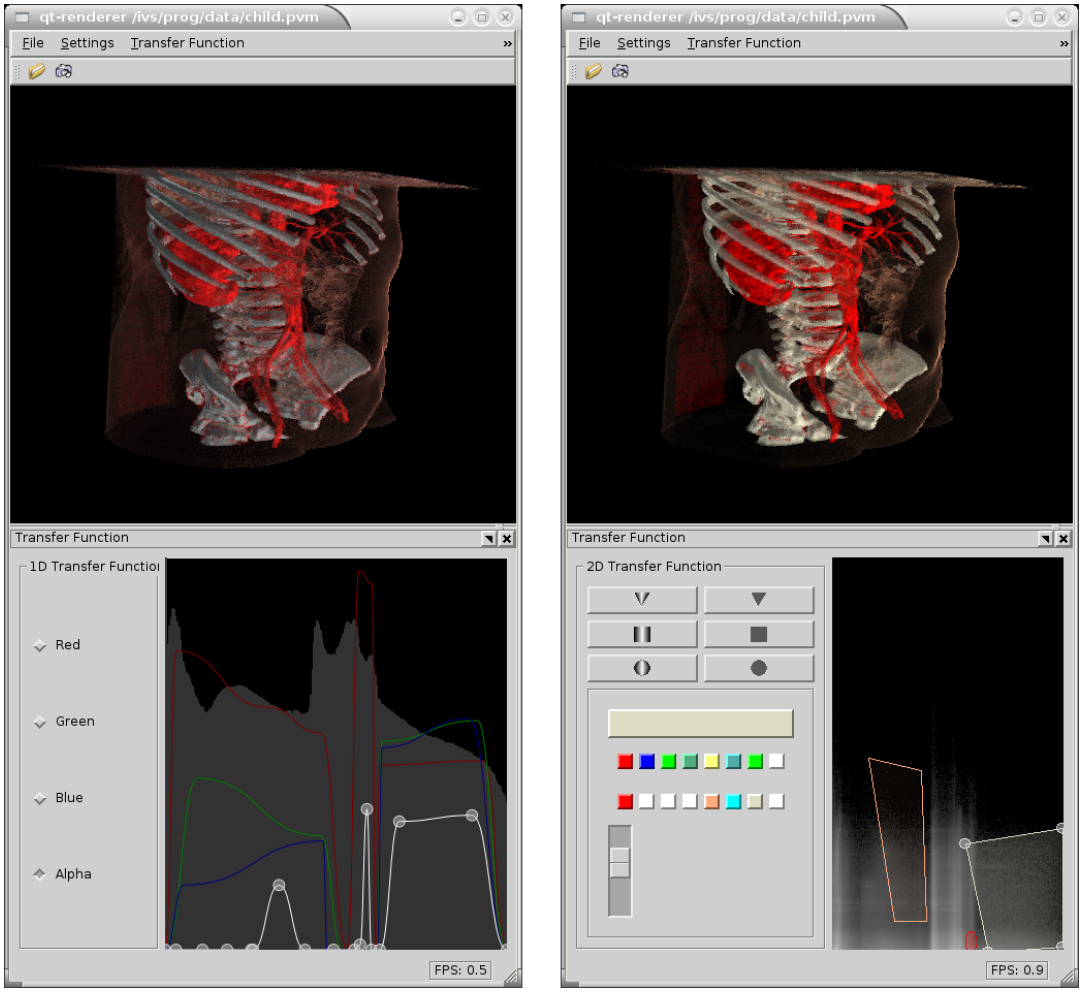

Figure 5.5: Two screenshots of the application. The first using the 1D transfer function widget (left) and the second using the 2D transfer function widget (right).

Dataset	width, height, depth	CPU time	GPU time
MRI_cerebrum	256, 256, 122	3.072s	0.914s
CT_cerebrum	256, 256, 244	6.243s	1.630s
CT_thoracalcolumna	512, 512, 151	15.373s	2.133s

Table 5.2: Central difference gradient calculation time on CPU and GPU

5.3.2 Gradient Calculation Test

Gradient calculation was implemented first on the CPU, and later reimplemented as a shader running on the GPU. We have run a test, measuring the gradient generation speed for three volumes. The results of the test are summarized in Table 5.2, and are discussed in Chapter 7.

Chapter 6

Qualitative Experiments

To test how well two-dimensional transfer functions perform compared to one-dimensional transfer functions, we have performed a survey. In the first section, details on how the survey was conducted will be described, and in Section 6.2, the results of the survey will be presented.

6.1 Survey description

In this experiment, we wanted to examine how the most frequently used multidimensional transfer function, the two-dimensional function of intensity and gradient magnitude, performed compared to a function of just intensity. Our assumption was that the two-dimensional transfer function would be better than the one-dimensional in noise reduction and in presentation of surfaces. To test these assumptions, we prepared both one-dimensional and two-dimensional transfer functions for two datasets. To test the application, we selected three people from The Interventional Centre. We wanted people with different education and experience, and different experience with volume rendering applications. The people we selected were a radiologist, a surgeon, and a computer engineer.

6.1.1 Presentation of the Datasets

The first dataset was a cerebrum CT scan of a female. The dataset was acquired using a 16-channel multidetector, multi row CT scanner (General Electric Medical systems Lightspeed Pro 16 Medical, Milwaukee, WI). The dataset consisted of 244 slices, with a slice thickness of 0.625 mm. Each of the slices was acquired with a width and height of 512 pixels, and with a pixel spacing of 0.351562 mm in both directions. 16 bits was allocated for

each sample. The slices were later down-sampled to a width and height of 256 pixels.

The second dataset was a CT abdomen scan of a child. This dataset was acquired using a 64-channel multidetector, multi row CT scanner (General Electric Medical systems Lightspeed VCT, Milwaukee, WI). The dataset consisted of 144 slices, with a slice thickness of 2.5 mm. Each slice had a width and height of 512 pixels and a pixel spacing of 0.507812 mm in both directions. 16 bits was allocated for each sample.

6.1.2 Transfer Functions Creation

Since we have not created a tool to convert between a one-dimensional and a two-dimensional transfer function, the two transfer functions had to be created separately. Both transfer functions used, as far as possible, the same color and opacity for the volume structures. For the cerebrum dataset, we tried to cut away everything but the brain. The resulting area was assigned a brown color. For the abdomen dataset we tried to segment out the skeleton, which was assign a white color, and the blood vessels which was assigned a red color.

6.1.3 Survey Protocol

The survey was performed in two stages. First, the test person was presented with two instances of the test program, running on the same computer, with the transfer function widgets hidden. The first instance used the one-dimensional transfer function, and the second instance used the two-dimensional transfer function. The test person was able to rotate and zoom the volumes in both program, but was unable to change the transfer functions. In some of the questions, the tester is asked to rate some feature. The rating was done on a scale from 1 to 6, where 1 is poor and 6 is excellent.

The following questions were asked for both datasets.

1. Rate the amount of noise in each rendering.

2. Rate how the program presents surfaces.

In the second stage of the test, only one instance of the program was run on the computer, and the test person was able to change the transfer function. The following questions were asked.

1. Rate the intuitiveness of the one-dimensional transfer function.

2. Rate the intuitiveness of the two-dimensional transfer function.

3. Which of the transfer function widgets do you prefer, and why?

4. If you have ever used a volume rendering program before, how would you rate the responsiveness of this application compared to other volume rendering applications you have used?

6.2 Survey Results

In this section, the results from the survey are presented. The survey results will be discussed in Chapter 7.

Some questions in the survey were not answered by all of the test subjects. This will be noted at the respective places. All ratings were done on a scale from 1 to 6, where 1 is poor and 6 is excellent.

The Radiologist

The radiologist was the first of the three to be interviewed. The answers from the interview are summarized in the following table.

Question	1D function	2D function
Rate the amount of noise in the CT cerebrum rendering	2	1
Rate how the program presents surfaces in the CT cerebrum rendering	1	2
Rate the amount of noise in the CT abdomen rendering	2	3
Rate how the program presents surfaces in the CT abdomen rendering	3	4

Because of difficulties using the transfer function widgets, the rest of the questions were not answered.

The Computer Engineer

The computer engineer was the second person to be interviewed. The following table summarizes the answers from the first part of the survey.

Question	1D function	2D function
Rate the amount of noise in the CT cerebrum rendering	2	3
Rate how the program presents surfaces in the CT cerebrum rendering	1	3
Rate the amount of noise in the CT abdomen rendering	4	5
Rate how the program presents surfaces in the CT abdomen rendering	4	5

The answers from the second part of the survey are summarized below.

Question	Answer
Rate the intuitiveness of the one-dimensional transfer function widget	5
Rate the intuitiveness of the two-dimensional transfer function widget	5

The computer engineer said that the one-dimensional transfer function widget was a little cumbersome to use, because of all the control point movement. He also said that the he would prefer straight lines instead of Bézier curve between the control points.

Concerning the responsiveness of the application, the computer engineer said that the application was overall fast, but it was a little slow when rotating the CT abdomen volume.

The Surgeon

The surgeon was interviewed as the third and final test subject. His answers are summarized in the following table.

Question	1D function	2D function
Rate the amount of noise in the CT cerebrum rendering	4	3
Rate how the program presents surfaces in the CT cerebrum rendering	3	4
Rate the amount of noise in the CT abdomen rendering	3	4
Rate how the program presents surfaces in the CT abdomen rendering	3	4

Because of difficulties using the transfer function widgets, the rest of the questions were not answered.

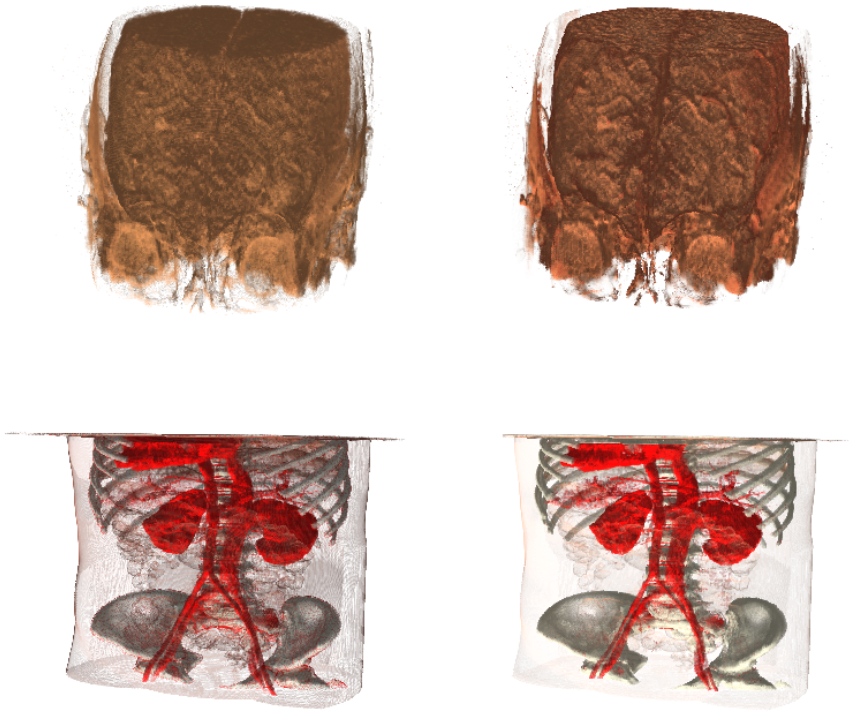

Figure 6.1: Renderings of the datasets used in the survey. At the top, the CT cerebrum rendered with the 1D transfer function (left) and the 2D transfer function (right). At the bottom, the CT abdomen dataset rendered with the 1D transfer function (left) and the 2D transfer function (right)

Additional Feedback

During the survey, we got some feedback on how the program could be made better, and in what area volume rendering might be useful.

When specifying a transfer function the difference of one pixel might have a great impact on the resulting rendering. The computer engineer suggested that the ability to zoom in on a specific area of the transfer function would improve the accuracy of the classification.

The surgeon said that a volume rendering of CT or MRI dataset could be a useful tool in the planning of minimal invasive surgery. The computer engineer said that a volume rendering classified with a transfer function, might be used to set initial values for a segmentation algorithm.

Chapter 7

Discussions

In this chapter, we will discuss different aspects of multidimensional transfer functions, as well as the implementation of the application, the result of the survey, and the performance tests. We start with the implementation.

7.1 Implementation

In this section, the implementation of the application will be discussed. We will give an overview of which implementation choices we had, as well as explain which ones we chose, and why.

Volume Rendering Framework

One of the first choices we had, was whether we were going to build on top of an existing volume rendering library, or start the development of a volume renderer from scratch. In the early stages of the development process, we investigated the possibilities for implement multidimensional transfer functions in the SIM Voleon library [Systems in Motion, 2005]. The first problem was that the transfer function lookup in SIM Voleon is integrated as a vital part of the core library, and integrating a multidimensional transfer function seamed like a major "hack". Another problem with changing functionality in the core library is that it might not be compatible with future versions of the library, which would limit the lifespan and value of our program.

The second alternative was to develop a new system from scratch. After reading [Krüger and Westermann, 2003] and [Scharsach, 2005], we decided to build the volume renderer based on the GPGPU design proposed in these papers.

More time could have been spent trying out different volume rendering libraries, and investigating how these could be extended to support multidimensional transfer functions. Some of the personal motivation for making the volume renderer was to learn as much as possible about volume rendering and this affected the decision.

Gradient Calculation

To calculate the gradient direction and magnitude, we chose the central difference algorithm shown in Section 2.7. We chose this algorithm because it is simple and fast, and it should give a fairly accurate result. We now see that this decision should not have been taken as lightly as we did, as the entire application depends on accurately calculated gradients. We should have implemented some more accurate gradient estimating algorithms to test how well the central difference algorithm performs for the tested volumes. If we found that the central difference algorithm was too inaccurate, one solution could have been to estimate the gradient with a slower more accurate algorithm, and store the gradients in a separate gradient volume or as additional voxel information in the volume dataset.

7.1.1 Rendering Quality

The quality of the rendering is an important part of any volume rendering application. For our application, there are three cases we would like to address. What happens with the image quality during rotation, how the image quality can be improved using anti-aliasing, and how lighting affects the rendering.

Rotation

In some volume rendering applications, the quality of the rendering is lowered during volume rotation and scaling. This is done to maintain interactive frame rates. In the case of the application that we have developed, this is not done. Our main reason for this is this is an unwanted feature. One of the easiest ways to get the feeling of a three-dimensional object on a computer screen is to rotate it. If the quality of the volume diminishes when the object is rotated, the user might loose the feel of the volume. In our opinion the users understanding of the volume is of higher importance that high interactivity, so even if the rotation is a little slow, the volume should be rendered with fairly the same quality during rotation as when it is still.

Anti-aliasing

As described in Section 5.1, we have implemented interleaved samples to reduce the aliasing effects from the raycasting. We can see from Figure 5.3 on page 35, that even with interleaved samples our application suffers from aliasing effects. This could be reduced by doing supersampling anti-aliasing, which render the scene to a larger render buffer and interpolates to get the actual pixel values. For interactive volume rendering applications, supersampling is too computational demanding to be run at each frame. One solution might be to do supersampling anti-aliasing while the volume is not moving, but this conflicts with the requirement of equal image quality at all times above.

Lighting

In our volume rendering application, we have implemented volume lighting. This has been done to give the rendering a better depth perception. On the other hand, lighting might create incorrect colors. We saw in the survey that one of the test subjects interpreted the changing in color, due to lighting and shading, as differences in data value.

7.1.2 GPGPU Programming

Using the GPU for non-graphics calculations can be tricky. As the GPU works in parallel, and it can be hard to convert algorithms from CPU to GPU. Another difficulty, and maybe the thing causing the most frustration, is the fact that the GPU is only able to output color values. This makes traditional debugging methods, like printing variables to see where the program crashes, almost impossible. Very few, if any, debugging tools for shaders are available.

Another problem using the GPU for regular calculations is the texture memory size. As textures are the only data storage that is accessible from the GPU, all the input data has to be uploaded as textures. As of January 2006, the largest memory capacity of a graphic card is 512 MB. This is more than enough to store quite large datasets. However, in our application we calculate the gradient direction and magnitude at each voxel, leading to a memory demand of five times the dataset size (for 8 bits datasets). One solution proposed by Scharsach [2005], is cached blocking. With cached blocking the volume is divided into blocks, and only the blocks containing voxels with opacity larger than zero is loaded into the texture memory. The problem with this strategy for our application is that changing

the transfer functions will be slow. First, the missing textures will have to be loaded as a texture. Then the gradient direction and gradient magnitude has to be calculated for each of the newly loaded texture blocks. This would severely lower the interactivity of the transfer function manipulation. Another strategy might be to use two graphic cards in SLI to double the available memory. Unfortunately, the current SLI technology only support mirroring of the texture memory, and gives no benefits regarding memory size. It has been reported that some new graphics cards can address up to 2 GB of texture memory, so it might not be long before we see a new increase in texture memory size.

Another memory related issue is the texture precision. In all current texture formats, the highest data precision is 32 bits. This is probably enough for most graphics applications, but it might be insufficient for some GPGPU applications.

GPU Speed

One performance drawback in our implementation is that only the fragment processor is used during the raycasting. The vertex processor lays idle while the fragment processor does all the work. Scharsach [2005] propose a strategy to also use the vertex processor, called empty space skipping. Instead of drawing the bounding geometry as a simple cube, we can draw a closer bounding geometry, cutting away all the voxels that have zero opacity. This strategy greatly increases the frame rates since the number of samples that have to be considered by the fragment processor is reduced. We chose not to implement this for two reasons. The first reason was that we had limited development time, and speed was not of great importance compared to the transfer function widgets. The second reason was that we were concerned that the interactivity of the transfer function manipulation would decrease, as we were redrawing the bounding geometry each time the transfer function was changed. We now see that this last argument might not be valid. Each time the transfer function is changed, the whole volume is evaluated by the raycaster. The overhead of drawing the bounding geometry should be lower that the overhead of evaluating all the voxels with zero opacity.

7.2 Multidimensional Transfer Functions

We will now discuss some of the positive and negative sides of multidimensional transfer functions.

One clear advantage of a transfer function with more than one dimension, is that we are able to classify more precisely. An example of this is shown in [Kniss et al., 2002], where the sinuses of the Visible Male CT dataset is classified using a two-dimensional transfer function of intensity and gradient magnitude. This classification is not possible with a transfer function of just intensity.

With a two-dimensional transfer function of intensity and gradient magnitude, we are also able to reduce noise from in the dataset. Dataset noise often consists of small regions with higher intensity than the surrounding voxels. As these regions are quite small, we can assume that these voxels have high gradient magnitude. If we exclude the voxels with high gradient magnitude in the transfer function, the noise in the rendering should be reduced.

One negative aspect of increasing the number of dimensions in the transfer function, is that it becomes more complicated to use. Park and Bajaj [2004], shows that to isolate a feature with a multidimensional transfer function, it is not sufficient just to search the intensity range, but all the dimensions ranges has to be searched.

Transfer Function Widgets

One method for searching for features in a dataset using a one-dimensional transfer function, like the one we have implemented, is to set the color values high for the whole function range, and use the alpha function to search for features. This method works quite well. However, when a feature is identified and we want to search for another feature a problem arises. If the feature just identified is visible while we search for another feature, it might be harder to find new features. This is even more true if multiple previously found features are visible. If we lower the alpha function of the feature just found, we might lose track of it later. In our two-dimensional transfer function widget this is not a problem. When a feature is identified, we can set the alpha in the area to zero and the area will still be marked by the boundary of the used transfer function element.

One of the strengths of the one-dimensional transfer function widget is the possibility to fade from one color to another over an interval. In the two-dimensional transfer function widget, this is not possible without specifically implementing such a feature.

With a one-dimensional transfer function as we have implemented, it is hard to create a specific color. The RGB blending is not intuitive for most users. In our two-dimensional transfer function implementation, specifying colors are much easier since we use a color selecting widget. Since Qt

uses the system default color selector widget, the user might already be familiar with it. Another advantage we have seen with the two-dimensional transfer function widget is the ability to stack different colored elements on top of each other. This can be useful for classifying features within features.

7.3 Results

7.3.1 Survey

We will now discuss the results of the survey, which were presented in Section 6.2. The survey was divided into two main parts. In the first part, the test subject was asked to compare two renderings that used different transfer functions. In the second part, they were asked to try out the two different transfer function manipulators. We will first discuss these two parts separately, before we discuss the survey as a whole.

Rendering Comparison

The test subjects were asked to rate two things, the first being how much noise the renderings contained. We saw that this question could be interpreted in different ways. What we wanted to know was how good the respective transfer function was at isolating the target object. In our definition, noise was all objects other than the target object, which were assigned the same color as the target object.

In the CT cerebrum dataset, both the radiologist and the surgeon rated the one-dimensional transfer function higher than the two-dimensional transfer function. The computer engineer on the other hand, rated the two-dimensional function higher than the one-dimensional function. This difference might be a product of different interpretation of noise. The two-dimensional transfer function contained more color nuances than the one-dimensional transfer function. This might have been interpreted, by the radiologist and the surgeon, as noise in the dataset, while interpreted as a feature of the transfer function by the computer engineer. In the CT abdomen dataset, all the testers rated the two-dimensional transfer function as slightly better than the one-dimensional transfer function. This change in opinion from the first dataset to the second might be a result of influence from the interviewer. Another reason might be that we were luckier with the specification of the two-dimensional transfer function for the CT abdomen dataset, than we were with the CT cerebrum dataset.

The second question was how well the program presents surfaces. What we wanted to find out with this question was how well the continuity of the surfaces was, as well as how easy it was to see how the surface curved. On this question, all of test people answered fairly equally. They were more positive to the two-dimensional transfer function, than to the one-dimensional transfer function.

Transfer Function Manipulation

In the second part of the survey, the test person was asked to test the transfer function manipulators on their own. This proved to be a non-trivial task. Only the computer engineer, who has much experience with one-dimensional transfer functions, was able to use the function manipulators effectively. For both the radiologist and the surgeon, this part of the test was aborted. It was obvious that the transfer function manipulators were hard to use, and learning to use both of the manipulators would take more time than we had assigned for the test. This should have been expected, as the transfer function manipulators were developed with focus on functionality, and not user friendliness.

The computer engineer was the only one of the testers who tested both function manipulators. On the question how intuitive he thought the different manipulators were, he rated them as equally and highly intuitive. This should come as no surprise, since he has previous experience with one-dimensional transfer function manipulators, and has a good understanding of how the gradient magnitude works. When asked which of the two manipulators he preferred, he said that the two-dimensional manipulator was the fastest and easiest to use. He also said that the one-dimensional manipulator was slower to use, because of how the manipulation of each function was implemented.

Survey Discussion

One thing that might have affected the results of the survey is the fact that each of the two transfer functions had to be constructed separately. In the ideal case, we should first construct a one-dimensional transfer function. This function should then be converted into a two-dimensional transfer function, and changes in the two-dimensional function should only be done in the gradient direction. This however, was not possible to do with our implementation. The two function manipulators are constructed quite differently, and we have not created a tool for converting a one-dimensional transfer function into a two-dimensional transfer func-

tion. Since the transfer functions were constructed separately, it was very hard to tell if a particular difference between two renderings was due to a difference in the gradient magnitude or in the intensity.

One problem we experienced during the survey, was the difference in field of interest between the medical professionals and the interviewer. Our focus was to test how the gradient magnitude enabled us to present surfaces better and reduce noise in the rendering. The focus of the medical professionals was immediately drawn to organs and structures that were represented in a bad way due to the lack of medical expertise in our transfer function specification. While the representation of different organs in volume rendering is a very important subject, it was out of the scope of the survey. We found out that explaining in detail what we wanted the test subject to look for was essential. These explanations might have affected the results of the survey.

7.3.2 Performance Test Results

We will now look at the results from the performance testing. The results were summarized in table 5.1 on page 38.

In the table, we can see that the GeForce 7800 GTX 512 MB version was the fastest single card, which should be no surprise. It was 270 percent faster than the GeForce 6600 GT, and 27 percent faster than the GeForce 7800 GTX 256 MB version. This test was not completely fair to the 6600 GT card, since it was an AGP version of the card that was tested.

The 27 percent increase from the 256 MB version to the 512 MB version of the GeForce 7800 GTX was expected because the GPU on the 512 MB version is clocked about 30 percent faster than the 256 MB version.

SLI Performance

The real disappointment in our performance test was the SLI setup. We only managed to get a 52 percent increase compared to the single card setup. We were expecting an increase in speed of 80 to 90 percent.

When running two graphics cards in SLI, there are three possible configuration modes; alternate frame rendering (AFR), split frame rendering (SFR), and SLI anti-aliasing. When we prepared for the performance test, we tested which of the AFR and SFR modes that gave the best performance. The SLI anti-aliasing mode was not considered since it is constructed to produce images of higher quality, and have no impact on the rendering speed.

In the SFR mode, the rendered scene is spilt into two parts, and each part is rendered at a separate GPU. This mode was the most effective, and the one we used in our test, with about 50 percent increase in the frame rate.

The AFR mode on the other hand, showed no increase compared to the single card setup. The reason for this might be the rendering to texture steps. In AFR mode, each even numbered frame is rendered at the first GPU, and each odd numbered frame is rendered at the second GPU. For our application, the rendering of two frames would be done as follows.

1. The first GPU renders the bounding geometry front-face, and the second GPU renders bounding geometry back-face. The second GPU cannot start until the first GPU is finished, because it needs its output as input.

2. The first GPU does the raycasting, and the second GPU renders the next frames bounding geometry front-face.

3. The first GPU renders the bounding geometry back-face, and the second GPU does the raycasting. The second GPU cannot start before the first is finished, because it needs its output as input.

We can see that this is not very efficient load balancing, and it might be the reason why the AFR mode did not give any performance increase. On solution to this problem might be to increase the number of buffered frames, so when a GPU is done with its rendering, it does not have to wait for the other GPU to finish, but can start rendering the next frame.

In our test of SLI, the system was highly unstable. This might have affected the performance of the SLI system, leading to lower performance readings.

7.3.3 Gradient Calculation Test Results

We will now discuss the gradient calculation test results, which were summarized in Table 5.2 on page 40.

The central difference algorithm is easily converted into a parallel algorithm, and is therefore well suited for GPU implementation. As we can see from the test results in the table, the GPU gradient implementation is much faster than the CPU implementation. For the smallest dataset the GPU implementation is 3.4 times faster than the CPU implementation, and for the largest dataset the GPU implementation is 7.2 times faster that the

CPU implementation. We can see that as the datasets gets larger, the difference between the two implementations increases. A reason for this might be that for the smaller dataset, a lot of the calculation time in the GPU implementation is used to prepare the rendering target, and reading back the result. As the dataset size increases, the time used for preparing rendering targets and reading the results is nearly the same, and more of the time could be spent on the actual gradient calculation.

Chapter 8

Conclusions and Future Work

8.1 Conclusions

In this thesis, we have investigated how transfer functions used in volume rendering can be extended to multiple dimensions. We have also implemented a volume rendering application that is capable of using both one-dimensional and two-dimensional transfer functions. This application has been implemented using shaders running on the GPU. To test the application and how well two-dimensional transfer functions perform, compared to one-dimensional transfer functions, we have conducted a survey and a set of performance tests.

We have found, that using a two-dimensional transfer function of voxel intensity and gradient magnitude, we can reduce both noise from the dataset, and interference from other objects in the rendering.

To be able to make a good comparison of two transfer functions of different type, it should be possible to convert one of them into the other. If the two transfer functions are specified separately, there will always be uncertainties whether a difference between the two renderings is due to the transfer function type, or due to a difference in the specification of the transfer function.

It is clear that using the GPU for parallelizable calculations is much faster than using the CPU. This was shown in the gradient calculation test, where the GPU implementation was over seven times faster than the CPU implementation for the large dataset. Raycasting is another problem that is well suited for solving on a GPU. Raycasting is easily parallelizable, and the hardware implemented three-linear interpolation that is available on GPUs, really speeds up the algorithm. In our performance test we got interactive frame rates with quite large, fully shaded volumes.

Even though multi-dimensional transfer functions are valuable tools that improve the classification process in volume rendering, it is still really hard to precisely classify a region. Multidimensional transfer functions might not be the final solution to this classification problem, but we think it is a step in the right direction.

8.2 Future Work

During our work with the application and this thesis, we have encountered some interesting subject that we either did not have time to investigate further, or that were beyond the scope of this project. In this section, we would like to present some of these areas which might be of interest in future work.

In our work on transfer functions, we have seen that most of the voxels gradient magnitude is located in the lower end of the transfer function scale. It might we valuable to use another scale, like the logarithm, for the gradient magnitude axis.

Precise specification of the transfer function has not been easy using our manipulators. Precise specification would be much easier, if the transfer function widget supported zooming. Allowing zooming of the transfer function would also require a much larger lookup table in the volume renderer. Compression of the lookup table might be needed if the user is allowed extensive zooming.

In our survey, we saw that transfer function manipulation is a non-trivial task. There is much work to be done in the area of user friendliness regarding transfer function manipulation.

Since an accurately calculated gradient is of high importance in transfer functions that use the gradient magnitude as a dimension, it would be valuable to do a quality measuring of different gradient calculation algorithms used in volume rendering.

Bibliography

Tomas Akenine-Möller and Eric Haines. *Real-Time Rendering*. A K Peters Natick, Massachusetts, 2002.

Mark J. Bentum, Barthold B. A. Lichtenbelt, and Tom Malzbender. Frequency analysis of gradient estimators in volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 2(3):242–254, 1996.

Randima Fernando, editor. *GPU Gems*. Addison-Wesley, 2004.

Nico Galoppo, Naga K. Govindaraju, Michael Henson, and Dinesh Manocha. Lu-gpu: Efficient algorithms for solving dense linear systems on graphics hardware. In *Proceedings of the ACM/IEEE Super Computing 2005 Conference*, November 2005.

Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *SIGMOD '04: Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, pages 215–226, New York, NY, USA, 2004. ACM Press.

Markus Hadwiger, Joe M. Kniss, Klaus Engel, and Christof Rezk-Salama. High-quality volume graphics on consumer pc hardware. In *ACM SIGGRAPH 2002 Course Notes 42*, July 2002.

Mark Harris. Mapping computational concepts to gpus. In Matt Pharr, editor, *GPU gems 2 : Programming Techniques for High-Performance Graphics and General-Purpose Computation*, pages 493–508. Addison-Wesley, 2005.

Taosong He, Lichan Hong, Arie Kaufman, and Hanspeter Pfister. Generation of transfer functions with stochastic search techniques. In Roni Yagel and Gregory M. Nielson, editors, *IEEE Visualization '96*, pages 227–234, 1996.

Anders Helgeland and Øyvind Andreassen. Visualization of vector fields using seed lic and volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 10(6):673–682, 2004.

Jiří Hladůvka, Andreas König, and Eduard Gröller. Curvature-based transfer functions for direct volume rendering. In *Proceedings of Spring Conference on Computer Graphics and its Applications 2000 (SCCG 2000)*, pages 58–65, May 2000.

Gordon Kindlmann and James W. Durkin. Semi-automatic generation of transfer functions for direct volume rendering. In *IEEE Symposium on Volume Visualization*, pages 79–86, 1998.

Cemil Kirbas and Francis K. H. Quek. A review of vessel extraction techniques and algorithms. *In review ACM Computing Surveys*, 36(2):81–121, 2004.

Joe Kniss, Gordon Kindlmann, and Charles Hansen. Multidimensional transfer functions for interactive volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 8(3):270–285, 2002.

Jens Krüger and Rüdiger Westermann. Acceleration techniques for gpu-based volume rendering. In *Proceedings IEEE Visualization 2003*, pages 287–292, 2003.

Marc Levoy. Display of surfaces from volume data. *IEEE Computer Graphics and Applications*, 8(3):29–37, May 1988.

Barthold Lichtenbelt, Randy Crane, and Shaz Naqvi. *Introduction to Volume Rendering*. Prentice-Hall, 1998.

William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3d surface construction algorithm. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques*, pages 163–169, New York, NY, USA, 1987. ACM Press.

Anthony Lovesey. A comparison of real time graphical shading languages. Technical Report CS4983, Faculty of Computer Science, University of New Brunswick, Canada, March 2005.

Eric B. Lum and Kwan-Liu Ma. Lighting transfer functions using gradient aligned sampling. In *Proceedings of the IEEE Visualization 2004 Conference*, pages 289–296, 2004.

Nelson Max. Optical models for direct volume rendering. *IEEE Transactions on Visualization and Computer Graphics*, 1(2):99–108, June 1995.

National Electrical Manufacturers Association NEMA. Dicom homepage. `http://dicom.nema.org/`, 2006.

Harry Nyquist. Certain Topics in Telegraph Transmission Theory. *Transactions of the AIEE*, 47:617–644, February 1928.

John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

Sangmin Park and Chandrajit Bajaj. Multi-dimensional transfer function design. Technical Report TR-04-11, The University of Texas at Austin, Department of Computer Sciences, April 2004.

Christof Rezk-Salama. *Volume Rendering Techniques for General Purpose Graphics Hardware*. PhD thesis, Der Technischen Fakultät der Universität Erlangen-Nürnberg, December 2001.

Stefan Roettger. The volume library. `http://www9.cs.fau.de/Persons/Roettger/library/`, 2005.

Henning Scharsach. Advanced gpu raycasting. In *Proceedings Central European Seminar on Computer Graphics*, pages 69–76, May 2005.

Systems in Motion. Sim voleon. `http://www.sim.no/products/SIM_Voleon/`, 2005.

Trolltech. Qt overview. `http://www.trolltech.com/products/qt/`, 2006.

Sean Whalen. Audio and the graphics processing unit. `http://www.node99.org/projects/gpuaudio/gpuaudio.pdf`, March 2005.

Jianlong Zhou, Andreas Döring, and Klaus D. Töennies. Distance transfer function based rendering. Technical Report TR-ISGBV-04-01, Institute for Simulation and Graphics, University of Magdeburg, Germany, 2004.