Universidade do Minho
Escola de Engenharia

Ricardo Jorge Rodrigues Sepúlveda Marques

Interactive GPU Ray Casting for Biomedical Imaging

Outubro de 2009

Universidade do Minho
Escola de Engenharia

Ricardo Jorge Rodrigues Sepúlveda Marques

Interactive GPU Ray Casting for Biomedical Imaging

Tese de Mestrado
Mestrado de Informática
Computação Gráfica / Volume Rendering

Trabalho efectuado sob a orientação do
Professor Doutor Luís Paulo Peixoto dos Santos
e da
Doutora Céline Paloc

Outubro de 2009

Acknowledgments

I would like to thank Adérito Marcos and Jorge Posada for giving me the opportunity to carry out this work in Vicomtech (Visual Interaction Communication Techniques), using the institute's financial and logistic support.

I am grateful for the close support that was given to me by Luís Paulo Santos, from University of Minho, and by Peter Leškovský and Luis Kabongo, from VicomTech. I would also like to thank Céline Paloc for putting at my disposal all the resources, both human and logistic, so that my work could be accomplished successfully.

I am indebted to Izaro Goienetxea for the feedback and encouragement in the most difficult situations.

Interactive GPU Ray Casting for Biomedical Imaging

Abstract

For many applications, such as walk-throughs or terrain visualization, drawing geometric primitives is the most efficient and effective way to represent the data. In contrast, other applications require the visualization of data that is inherently volumetric. For example, in biomedical imaging, it might be necessary to visualize 3D datasets obtained from CT or MRI scanners as a meaningful 2D image, in a process called volume rendering. As a result of the popularity and usefulness of volume data, a broad class of volume rendering techniques has emerged. Ray casting is one of these techniques. It allows for high quality volume rendering, but is a computationally expensive technique which, with current technology, lacks interactivity when visualizing large datasets, if processed on the CPU. The advent of efficient GPUs, available on almost every modern workstation, combined with their high degree of programmability, opens up a wide field of new applications for the graphics cards. Ray casting is among these applications, exhibiting an intrinsic parallelism, in the form of completely independent light rays, which allows to take advantage of the massively parallel architecture of the GPU. This document describes the implementation and analysis of a set of shaders which allow interactive volume rendering on the GPU by resorting to ray casting techniques.

Ray Casting Interactivo para Visualização de Imagens Biomédicas

Resumo

Em muitas aplicações, tais como *walk-throughs* ou visualização de terrenos, a maneira mais eficiente de representar os dados é desenhando primitivas geométricas. Por outro lado, existem aplicações que requerem a visualização de dados inerentemente volumétricos. Por exemplo, no ramo das imagens biomédicas, pode ser necessário visualizar conjuntos de dados 3D obtidos a partir de scanners TC ou IRM sob a forma de uma imagem 2D, num processo designado por *volume rendering*. Em resultado da utilidade e popularidade dos dados volumétricos, surgiu uma vasta gama de técnicas de *volume rendering*. *Ray casting* é uma dessas técnicas. Permite realizar *volume rendering* de alta qualidade, mas trata-se de uma técnica cara em termos computacionais à qual, com a tecnologia actual, falta interactividade quando se visualizam grandes conjuntos de dados, caso seja processada no CPU. O surgimento de GPUs mais eficientes, presentes em quase todos os computadores modernos, combinado com o seu alto grau de progamabilidade, abre um novo e vasto campo de aplicações para as placas gráficas. *Ray casting* está entre estas aplicações, exibindo um paralelismo intrínseco, sob a forma de raios completamente independentes, que permite tirar partido da arquitectura do GPU massivamente paralela. Este documento descreve a implementação e análise de um conjunto de *shaders* que permitem realizar *volume rendering* interactivo no GPU fazendo uso de técnicas de *ray casting*.

Contents

1 Introduction

Three dimensional volume datasets are frequently used in the scientific community. They are obtained by simulation, sampling or modeling. In economics or fluid dynamics, for example, numerical simulations can generate large scale volumetric datasets. In medical imaging, different scanning techniques such as Magnetic Resonance Imaging (MRI) or Computed Tomography (CT) are used to collect samples of the interior of the human body, which are stored as 3D datasets [12].

These datasets may be visualized in three dimensions, in order to allow specialists to interpret the information. In traditional computer graphics, 3D objects are created using surface representations, by drawing geometric primitives that create polygonal meshes [16]. However, using surface rendering techniques to display volumetric data results in the loss of a dimension of information [12, 10]. For example, in CT datasets, the useful information is not only contained on extracted iso-surfaces, but within the iso-surfaces as well. Therefore, several volume rendering techniques were developed to visualize the entire 3D data as a single 2D image. Volume rendering techniques convey more information than surface rendering methods, but at the cost of increased algorithm complexity and, consequently, increased rendering times [2].

Ray casting [13] is one of these techniques. It evaluates the color of each pixel in the final image by shooting a ray through the scene starting from the viewer position. If the ray hits the volume, the color of the pixel is calculated by sampling the data values along the ray at a finite number of positions in the volume and combining them together. This technique, however, has a limitation when executed on CPUs: for large volume datasets and close viewing planes that maximize the number of rays which must be shot, the time to render a single image is too high to allow for a real time visualization.

In the particular case of visualization of medical images, doctors can benefit from the use of ray casting techniques for diagnostic purposes, planning of treatment, or surgery [2]. Three dimensional techniques allow the physicians to see anatomical features and interrelationships explicitly and thus to take decisions based on more information, and to communicate them better [23]. But widespread use depends on the ability to quickly fly-through the data, even while the patient is on the scanner, and perform the diagnosis [23].

Driven by the demand of the game industry, the performance of modern GPUs has exceeded the computational power of CPUs both in raw numbers and in the growth rate [24, 27]. Consequently, Krüger and Westerman [14] presented a different approach for implementing the ray casting technique. They proposed a GPU-based algorithm, capable of taking advantage of the features of modern graphics cards (see also [22]):

- A massively parallel architecture
- A separation into two distinct units (vertex and fragment shader) that can double performance if workload can be split
- Fast memory and memory interface
- Dedicated instructions for graphical tasks
- Vector operations on 4 floats that are as fast as scalar operations
- Trilinear interpolation is automatically (and extremely fast) implemented in the 3D texture

The ray casting algorithm fits modern GPUs. It exhibits an intrinsic parallelism, in the form of completely independent light rays, which allows to take advantage of the massively parallel architecture of the GPU. But as a new technology, GPU ray casting is not well established yet. Appropriate libraries implementing the technique which are compatible with graphics processors from the two main manufacturers, NVidia and ATI, cannot be found.

The work presented in this document is motivated by the creation of a Human Atlas visualization tool which allows the user to navigate through human medical datasets (obtained from CT or MRI scanners) in real time, using direct volume rendering. The development of a set of shaders, which implement a GPU ray caster that is able to display 3D datasets with different compositing techniques (composite, Maximum Intensity Projection and X-Ray), is described, and its performance is analyzed. Moreover, segmentation data was used to identify distinct objects of interest present in the dataset. A new approach for the implementation of a highlighting mechanism, which allows the user to quickly highlight specific objects by shooting a highlight ray into the volume, is also presented.

In the next section, the theory of volume rendering, as well as the main approaches to render volume datasets are discussed. An historical perspective of the evolution of the field is also presented. In Section 3, the foundations of the ray casting technique are detailed. The implementation and results of the rayCasting technique in the GPU are described in Section 4. Sections 5 to 8 present the endowing of the ray caster implemented in Section 4, and are divided in two subsections: Implementation, and Results and Discussion. The use of a stochastic jitter for reducing the wood-grain effect is shown in Section 5. In the following section two empirical visualization modes (X-Ray and MIP) are added to the application. In Section 7 segmented datasets are used to visually differentiate areas of interest. A mechanism developed to allow the user to interactively highlight objects of interest is presented in Section 8. The last section contains the conclusions, and an outlook of the work developed.

2 Volume Rendering

Volumetric (or 3D) datasets are composed by a set of samples (x, y, z, v), called voxels, representing the value v of some property of the data, at a 3D location (x, y, z) [2]. Volume rendering describes a wide range of techniques for generating a 2D image directly from one (or possibly more) of these datasets. This process is alternatively named direct volume rendering, as it operates directly on the actual data samples from the 3D dataset, without generating intermediate geometric primitive representations.

The origin of volume rendering remounts to the decade of 1980, when researchers first used computers to display volume information. Blinn [1] proposed a set of physically based functions to describe the interaction of light with cloudy objects, a typical volume rendering problem. Kajiya [13] extended the ray tracing method presented by Whitted [30], in order to render 3D datasets, in what would be the first ray casting algorithm. The field kept on evolving, with the refinement of the existing volume rendering techniques [6, 21, 26, 29].

In the middle 1990s, boosted by the availability of more efficient graphic cards, Cullip and Newman [4] presented the first important GPU volume rendering approach, further developed by Cabral et al. [3]. It consisted on exploiting the GPU texture mapping capabilities, to create a set of sampling planes of the volumetric data, and therefore, was later classified as a texture based approach. The sampling planes could be either object aligned with a set of 2D-textures [28] or viewport aligned with one 3D-texture [19], and were composed to produce the final image. This technique is now widely accepted as a simple way to interactively render medium sized datasets with reasonable quality, and has been finetuned and revisited [5, 7].

Krüger and Westerman [14] presented in 2003 an alternative approach, by implementing a ray caster in the fragment shader of the GPU. The reason for this late development was the demand for advanced fragment shader functionality that was not available previously [10]. GPU ray casting is based on the existing CPU methods, essentially adopting them for graphics hardware. Since its introduction by Krüger and Westerman, GPU ray casting became a very active field of research, with many publications on the improvement of the technique [22, 24, 20].

2.1 Light Transport and Optical Models

In order to create a 2D image of a volumetric dataset, it is necessary to simulate the light transport within the volume. Therefore, a volumetric description of the physical properties of the participating medium must be provided. These physical properties are then used to compute light interactions for actual image synthesis.

The physical basis for volume rendering relies on geometric optics [10]. In geometrical optics light can be described by the amount of radiant energy traveling within some frequency interval into a given direction [8, 17]. The light is assumed to propagate along straight lines, unless interaction between light and the participating medium takes place. Three types of interactions are typically taken into account [10, 8]. Each one of these phenomena can be see in Figure 1.

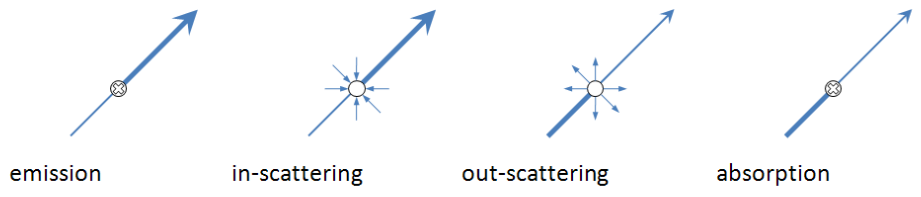


Figure 1: Interactions between light and participating media which affect the radiance along a ray.

In emission (Figure 1a), the material emits light, increasing the radiative energy. A scattering interaction consists of a change of direction of the light propagation, due to a change of properties of the participating media. There are two types of scattering: in-scattering (Figure 1b), where the light arriving to a point from different directions is absorbed, causing the light energy to increase; and out-scattering (Figure 1c), where the light traveling in one direction is scattered into multiple directions, causing the light energy to decrease. Finally, in absorption (Figure 1d), the material absorbs light by converting it into heat, reducing the light energy.

Because evaluating all the light interactions for volumes is computationally intensive usually some simplified models are used. The idea is to reduce the type of light interactions that are taken into account. The following models are commonly used [10].

- **Absorption Only**. It is assumed that the volume consists of perfectly black material that may absorb incident light. No light is emitted or scattered.
- **Emission Only.** The volume is assumed to consist of gas that only emits light, but is completely transparent. Absorption and scattering are neglected.
- **Emission-Absorption Model**. The gas can emit light and absorb incident light. Scattering and indirect illumination are neglected.
- **Single Scattering ans Shadowing.** Single scattering of light that comes from an external source (i.e., not from within the volume) is included in this model. Shadows are modeled by taking into account the attenuation of light that is incident from an external light source.
- **Multiple Scattering.** This model evaluates the complete illumination model for volumes, including emission, absorption, and scattering effects.

The emission-absorption model is the most widely model used for volume rendering, due to its good compromise between generality and efficiency of computation. This model leads to the volume rendering integral, presented in the following section.

2.2 The Volume Rendering Integral

The volume rendering integral (see Equation 1), first described by Kajiya and Herzen [13], and then formally derived by Max [17], is an equation that computes the light transport through a volume according to the emission-absorption model. The equation takes into account emission and absorption effects, but discards more advanced effects such as scattering and shadows.

$$I_\lambda(x,r) = \int_{s_0}^{L} C_\lambda(s)\mu(s)e^{-\int_0^s \mu(t)dt}ds \tag{1}$$

It integrates, along the direction of light flow r, from the starting point $s = s_0$ to the endpoint $s = L$, the amount of light of wavelength λ, that is received at point x. The variable μ represents the density of the particles that compose the volume. C_λ is the amount of light of wavelength λ reflected and/or emitted at a location s in the direction r. To account for the higher reflectivity of particles with higher density, one must weight C_λ by μ. The term

$$\int_0^s \mu(t)dt$$

is referred to as the extinction term between 0 and s, and can be interpreted as the distance which light travels before being absorbed.

The goal of volume rendering is to compute the integral, shown in Equation 1. However, except for particular cases, the volume rendering integral cannot be solved analytically. Therefore, most algorithms use numerical methods to find an approximated solution of the integral. The common approach is to split the integration domain into n subsequent intervals, and evaluate the Equation 1 using a discrete Riemann sum (see Equation 2).

$$I_\lambda(x,r) = \sum_{i=0}^{L/\Delta s} C_\lambda(s_i)\alpha(s_i)\prod_{j=0}^{i-1}(1-\alpha(s_j)) \qquad (2)$$

In Equation 2, $\alpha(s_i)$ represents the opacity values sampled along the ray, assuming values from 0 to 1. $C_\lambda(s_i)$ gives the local color values, derived from the illumination model. C and α are referred to as transfer functions, and assign color and opacity to each volume sample. The color of each sample $(C_\lambda(s_i))$ is weighted by its corresponding opacity to account for the higher reflectivity of subvolumes with higher density. The term

$$\prod_{j=0}^{i-1}(1-\alpha(s_j))$$

computes the light absorption from the volume entry point j, till the previous sampling position $i-1$. Thus, a practical implementation of Equation 2 traverses the volume from front to back, calculates colors and opacities at each sampling position, and weights these values by the current accumulated transparency $(1-\alpha(s_j))$. These terms are added to the accumulated color and opacity, which are then used to calculate the contribution of the next sample along the ray.

2.3 The Volume Rendering Pipeline

To evaluate the discretized volume rendering equation (see Equation 2), a sequence of steps, designated as the volume rendering pipeline, must be performed. In general, the volume rendering pipeline is composed by the following stages [27, 10, 12]:

- **Data Traversal.** The sampling positions of the 3D dataset are chosen. This samples are the basis for the discretization of the volume rendering integral.

- **Interpolation.** In general, the location of a sample does not coincide with any of the grid points of the 3D dataset. Therefore, the sample value must be evaluated by reconstructing a continuous 3D datafield from the original discrete dataset.

- **Gradient Computation.** A gradient field of the scalar values that composes the 3D dataset is needed, if the computation of local illumination is required.

- **Classification.** Classification is the process of assigning optical properties (color and an opacity) to the data samples. It allows to distinguish different areas of the volume, by attributing different optical properties to distinct areas.

- **Shading and Illumination.** The process of evaluating the illumination model.

- **Compositing.** Compositing determines the contribution of a classified and shaded sample to the final image.

The first stage of the volume rendering pipeline consists of determining a set of sampling positions throughout the volume. The resulting sampled values are used to approximate the volume rendering integral. But the position of these samples is, in general, different from the grid point positions holding the volume data. That is why a second stage, the interpolation stage, is necessary. Whenever the sample position does not coincide with the grid points of the 3D dataset, the sample value must be calculated by interpolating the values from one or more neighboring grid points. Typically, for regular grids, trilinear interpolation is used.

The gradient computation stage consists of calculating a gradient scalar field of the volume, which can be used for computing local illumination [10]. However, this stage is not strictly needed. It can be included in the volume rendering pipeline to simulate single scattering effects of the external light, introducing greater realism in the final image. Single scattering is approximated by a local illumination model that imitates local surface rendering (e.g. Phong model), where the normal to the surface point is used to calculate the light that is scattered in the direction of the viewer. In the case of local illumination for volume rendering, the gradient field is used as the surface normal for each volume point, producing an effect similar to an illuminated iso-surface.

The classification of a data sample is done using a transfer function. A transfer function maps properties of the 3D dataset to optical properties for the volume rendering integral, typically an RGBA value. This stage can be executed in two distinct orders. With pre-classification, optical properties are first mapped to the grid points of the dataset, using the transfer function, and then resulting RGBA values are interpolated to obtain the sample value. Post-classification, in contrast, first interpolates the grid points values, and then assigns the corresponding optical properties by applying the transfer function to the interpolated data value. The later approach was proved to achieve better results, by being able to reproduce higher frequencies of the transfer function than in pre-classification, and therefore, reducing the visual artifacts caused by the error in the approximation of the volume rendering integral [10].

After the sample is assigned its optical properties, its contribution to the resulting image is computed in the last two stages of the pipeline: the shading and illumination stage, where, based on the optical properties of the sample (and in some cases on its gradient), the color contribution of the sample is calculated; and the compositing stage, where the contribution of each sample is combined according to the chosen compositing method.

2.4 GPU-Based Volume Rendering Techniques

In the field of GPU-Based volume rendering [27, 10], there are two distinct approaches to render 3D datasets at interactive rates: a texture based approach, and a ray casting based approach. Below, these two approaches are presented, along with their main features, advantages and drawbacks.

2.4.1 Texture Based Approach

The texture based approach was the first GPU-Based volume rendering technique, originally presented by Cullip and Newman [4], and further developed by Cabral et al. [3]. In this approach, the volumetric data is stored in the GPU memory as a stack of object aligned 2D textures [28] or as a viewport aligned 3D texture [19]. These textures are then mapped onto a sequence of semi-transparent 2D slices, called proxy geometry, using the built-in hardware texture interpolation. The function of the proxy geometry is to provide a sequence of polygons where the texture slices will be displayed. Finally, the proxy geometry is rendered in back to front order, exploiting the per fragment operations and alpha blending capabilities of the GPU.

The 2D texture approach requires three copies of the volume dataset, each of them aligned with one of the main axis of the object (see Figure 2).

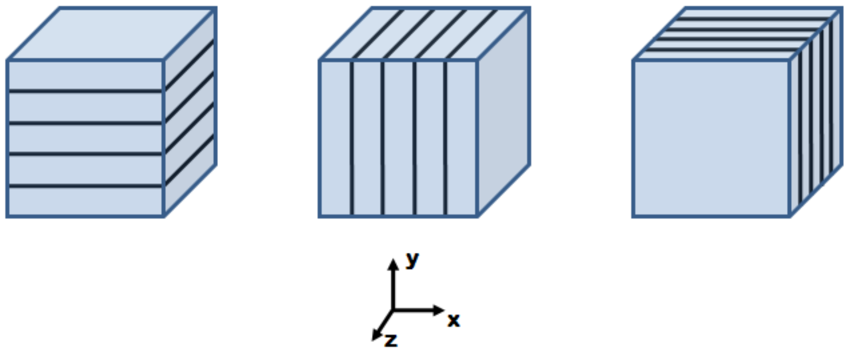


Figure 2: Object aligned sampling surfaces. For each of the three main object axis (x, y, z), a stack of 2D textures is stored. During the rendering process, the stack corresponding to the axis most parallel to the current viewing direction is chosen, mapped to the proxy geometry, and rendered in back to front order using alpha blending.

This increases three times the amount of memory needed to store the volumetric data. During the visualization process, one of the stacks is chosen to be displayed, depending on the current viewing direction. The chosen stack is the one corresponding to the axis most parallel to the viewing vector. To map the texture slices to the proxy geometry, 2D interpolation within each slice of the stack is used. When a given texture slice is mapped, the information from the previous and the next slices is not taken into account. This causes the final result to have lower quality, usually resulting in visible artifacts [10]. The way to increase the quality of the final image, thus removing these imperfections, is to increase the sampling rate by incrementing the number of texture slices that store the volume on the graphics card, causing the amount of memory necessary to be even larger.

Compared with the 2D texture solution, the 3D texture based approach is superior, removing some of the significant drawbacks while preserving almost all the benefits [10]. With 3D textures only one copy of the volume dataset is necessary, because trilinear interpolation allows for the extraction of slices in arbitrary directions, for example diagonally (see Figure 3).

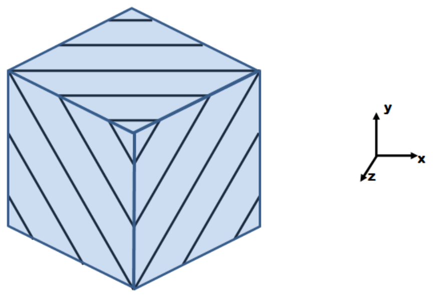


Figure 3: Viewport aligned sampling surface. The volume is stored as a single 3D texture. A set of slices parallel to the image is rendered in back-to-front order, using alpha blending.

This reduces the size of the volume in the graphics card memory when compared to the 2D texture approach. The volume is sliced by a set of planes parallel to the viewing direction, and the resulting slices are composed to achieve the final image. Furthermore, the use of trilinear interpolation instead of bilinear results in higher image quality. It also provides a natural way of increasing the sampling rate only by increasing the number of slices of the proxy geometry, without having to increase the size of the volume stored in memory.

2.4.2 Ray Casting Approach

Ray casting is a well-known volume rendering algorithm designed by Kajiya and Herzen [13] back in the 1980s. The basic idea is to trace rays from the camera into the volume, computing the volume rendering integral along the rays (see Figure 4).

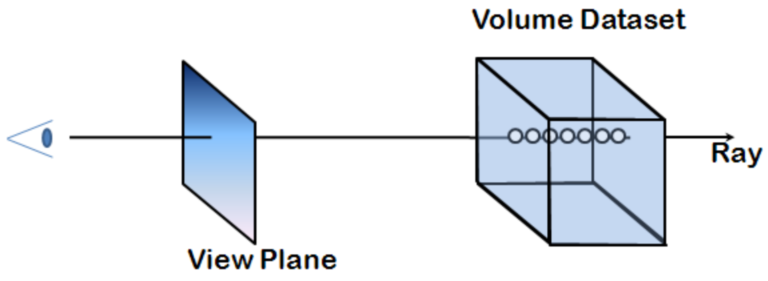


Figure 4: Ray casting scheme. For each pixel in the image plane, a single ray is shot. If the ray hits the volume data, a set of samples is collected at discrete positions along the ray. These samples are then combined to achieve the final pixel color.

For each pixel in the image, typically a single ray is cast into the volume. The volume data is sampled at discrete positions along the ray. The contribution of each sample is accumulated to obtain the final color and opacity of the pixel.

This algorithm fits very well the GPU architecture and capabilities [14, 22]. In GPU ray casting, the volume data is uploaded to the GPU memory as a 3D texture. A fragment shader program is used to implement the ray casting algorithm, working on the fragments generated by rendering a polygon covering the screen space occupied by the volume bounding box. For each fragment of the polygon a ray is cast. Due to the independence between the per ray operations, and to the parallel architecture of the GPU, this operation can be done in parallel. The samples along the ray are taken using the hardware trilinear interpolation, and composed to compute the final pixel color (for more details, see section 3).

2.4.3 Texture Based vs Ray Casting Approaches

Compared with the ray casting technique, the texture based approach has several drawbacks [10, 22]. First, it performs the evaluation of the volume rendering integral for fragments that do not contribute to the final image [24, 14] (e.g. occluded fragments). This characteristic significantly increases the amount of texture fetch operations, numerical operations, i.e. lighting calculations, and per pixel blending operations that are executed. Due to the inflexible nature of this algorithm, advanced acceleration techniques which could correct this situation are hardly implementable.

Ray casting, on the other hand, is a much more flexible algorithm which allows for the integration of acceleration techniques that can solve the problem of unnecessary per fragment calculations [14, 24]. The early ray termination mechanism which truncates the ray when the upcoming samples do not influence the final result (see section 3.2), and the empty space skipping technique [14] which skips volume regions that are considered empty are examples of these techniques.

Another disadvantage of the texture based approach compared to the ray casting technique is when a perspective view is applied (see Figure 5).

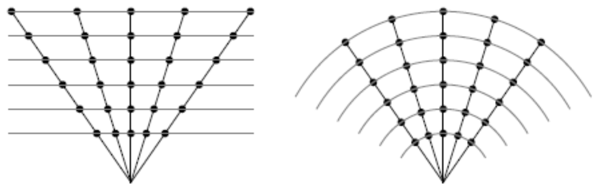


Figure 5: Sampling distances in GPU volume rendering techniques. On the left, the varying sampling distances of the texture based approaches are shown. On the right, the constant sampling distances of the ray casting approach can be seen.

Using a texture based approach and a perspective view, the sampling distances vary from ray to ray, introducing incoherences in the final image. In contrast, ray casting maintains a constant sampling distance, therefore avoiding visual artifacts [15].

For these reasons, the texture based techniques can be considered secondary for the implementation of a volume rendering framework, as they fail to take full advantage of the hardware capacities.

3 Ray Casting

Since its introduction by Kajiya and Herzen [13], ray casting became a well known volume rendering technique. Its goal is to approximate the volume rendering integral along each ray. These rays originate in the image plane, and have the direction of the viewing vector (see Figure 5 in section 2.3.2).

3.1 The Ray Casting Pipeline

The ray casting pipeline can be described by the pseudo code presented in Algorithm 1.

Algorithm 1 Ray Casting Pseudo Code

```
1. Ray set-up

2. Traversal loop:

   2.1 Data access: get a volume sample at the current ray position

   2.2 Classification: apply the transfer function to the sampled value

   2.3 Shading: compute the color contribution of the current sample

   2.4 Compositing: add the contribution of the current sample to the final image

   2.5 Advance ray position: go to the next position along the ray

   2.6 Ray Termination: if the ray leaves the volume bounding box, or if the
         threshold for the opacity is reached, the loop terminates

End loop
```

Two main components can be identified, namely the **Ray set-up** and the **Traversal loop**. In the **Ray set-up** phase, the ray direction and its entry point in the volume are calculated according to the current viewing parameters (see Figure 6). Depending on the implementation of the algorithm, the ray exit position, and the length of the ray can also be calculated.

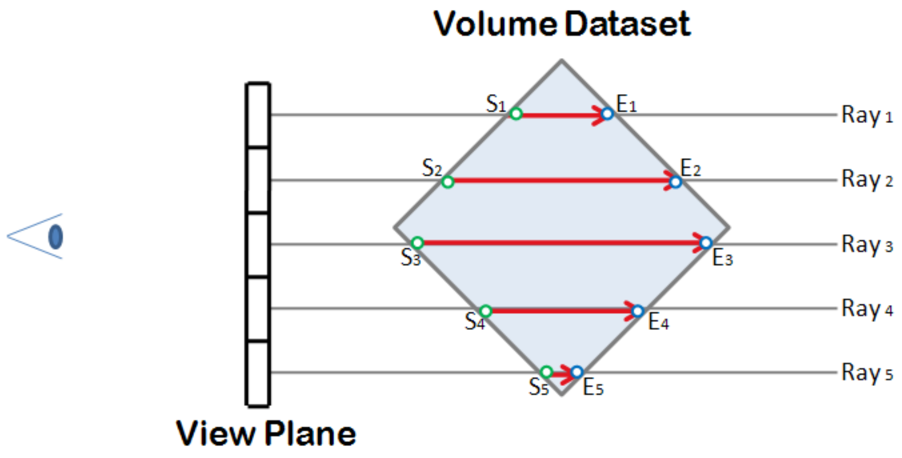


Figure 6: Ray Casting Ray Set-Up.

Then, the **Traversal Loop** starts. This component is responsible for traversing along the ray, collecting data samples and updating the pixel color with the contribution of the current sample. It is composed by the following six sub-components:

- **Data Access and Interpolation**. The 3D dataset is accessed at the current ray position. The discrete 3D dataset might be reconstructed, if the current ray position does not coincide with one of the dataset grid points (usually interpolation is the most used filter).
- **Classification.** The transfer functions are applied to the sampled value, yielding the corresponding optical properties (color and opacity).
- **Shading and Illumination**. Based on the optical properties of the sample and on the illumination model, the color contribution of the sample is calculated.

- **Compositing**. The contribution of the current sample to the final color of the pixel is used to update the previously accumulated color and opacity.
- **Advance Ray Position**. The current ray position in incremented to the position of the next sample.
- **Ray Termination**. If the ray leaves the volume bounding box or if the threshold for opacity value is reached (see Section 3.2), the traversal loop is terminated.

3.2 Early Ray Termination Mechanism

Early ray termination is a feature of the ray casting algorithm which consists on truncating the light rays as soon as the volume elements further away along the ray are occluded. The ray traversal loop can be stopped once the accumulated opacity for the corresponding pixel reaches a certain threshold. This introduces an error on the approximation of the volume rendering integral, but, with thresholds very close to 1 this error is negligible for the final image quality [10]. A typical value used as a threshold is 0.95 and, therefore, will be used throuought this thesis. This stopping criterion is combined with other stopping criteria for the ray traversal loop (e.g. checking if the ray is out of the volume bounding box).

4 GPU Ray Casting

The basic idea of GPU Ray Casting is to implement the ray casting algorithm (see Algorithm 1) in a fragment shader program. Multipass ray casting is an approach used to implement a GPU ray caster. It was described in [14], by Krüger et al., and consists on the following main steps:

- a **first rendering pass**, processed in the GPU, in which the exit point for each ray is calculated
- a **second pass**, in which the entry point for each ray is obtained
- **main passes 3 to N** consist of sampling the volume dataset along the ray and combine the samples to determine the pixel color. In each pass, M steps along the ray are performed, and then an intermediate pass is executed
- **intermediate passes 3 to N,** where the stop criterion is tested and the ray is terminated in case it left the volume dataset boundaries or if the opacity accumulated for the current pixel has reached a given threshold

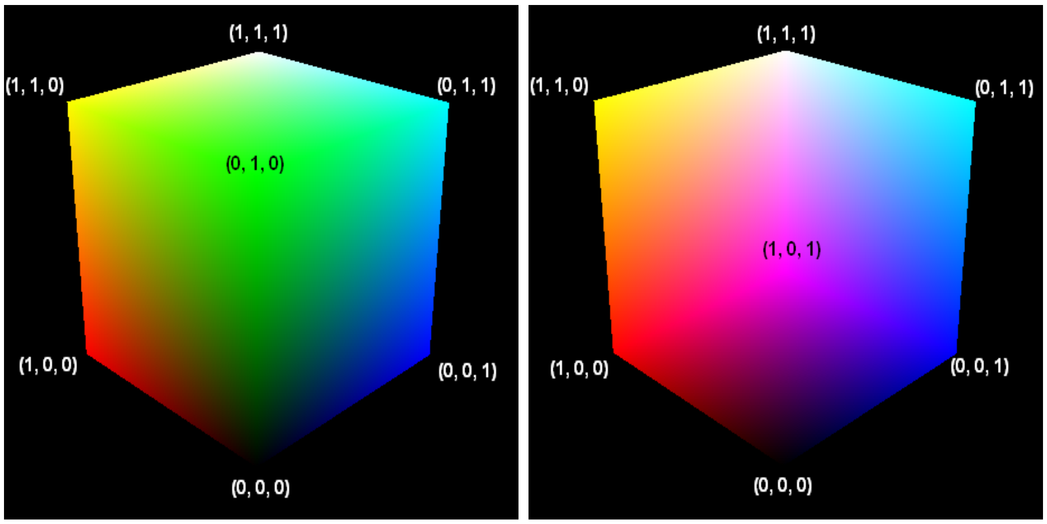
This multipass approach was first designed to overcome hardware limitations, since early GPUs did not provide loops functionality and conditional branches were hard to implement [10]. Therefore, the traversal of a ray was initiated and driven by a CPU-based program. Currently, loops and conditional branches are available in the instruction set of the GPU programming languages, which permits the simplification of the algorithm to two passes:

- in a **first pass**, the exit point for each ray is calculated
- in the **second pass**, the entry points of each ray are calculated, and using a loop instruction the ray traversal is performed, combining the samples collected, until the stop criterion is reached

In the next two sections, the implementation of the multipass ray casting algorithm, as well as the results obtained are described.

4.1 Implementation

The strategy to calculate the entry and exit points of the rays is to store the volumetric dataset in a 3D-Texture and define a bounding box for this dataset. This bounding box is a cube where the color channel encodes the 3D-Texture coordinates of the volumetric dataset boundaries (which range from 0 to 1). Rendering the front faces of the cube yields an image with the entry position of the rays in the bounding box, encoded in color (see Figure 7a). Drawing the back faces of the bounding box results in an image encoding the exit position of the rays (see Figure 7b).



(a) (b)

Figure 7: Rendered front (a) and back (b) faces of the bounding box.

Subtracting the two images, shown in Figure 7, yields the lines going from the ray start to the exit ray positions. These lines are used to calculate the rays directions, necessary for the ray traversal step.

In the first pass of the algorithm, the back faces of the bounding box are rendered using the OpenGL fixed functionality. The resulting image is stored in a 2D-Texture (i.e. the exit texture). In the second pass, with the ray casting shaders enabled, the front faces bounding box are rendered. This will cause the GPU to receive the information of the entry points of the rays, in the form of a color for each pixel. The GPU combines this information with the exit points previously stored in a 2D-Texture and performs the ray set up and ray traversal, evaluating the color for each pixel. Following is a detailed description of the implementation.

Algorithm 2 shows the core of the CPU part of the multipass ray casting algorithm.

Algorithm 2 Multipass ray casting algorithm in the CPU.

```
For each frame:

  1. Render the back face of the bounding box to a 2D texture
        using the OpenGL fixed functionality

  2. Enable the Ray Casting Shaders

  3. Render the front face of the bounding box and perform the
        Ray Casting using the Ray Casting Shaders

  4. Disable the Ray Casting Shaders

  5. Enable the OpenGL fixed functionality
```

It consists of a loop where, for each frame, the back face of the volume bounding box is rendered to a 2D-Texture using the OpenGL fixed functionality. Then, with the ray casting fragment shader enabled (further described in algorithms 2 and 3), the front face of the bounding box is rendered. This way, an interpolated color for each pixel becomes available in the fragment shader, corresponding to the texture coordinates of the ray starting point. The shaders enter in action to calculate the color value of each pixel, sampling the volumetric dataset. Finally, the OpenGL fixed functionality is re-enabled, and the display loop can be repeated.

To perform the second pass of the algorithm, a fragment shader program was designed to drive the per pixel operations. The first task of the fragment shader is the ray set up. It consists in finding, for each pixel, the entry and exit points of the ray in the volume bounding box. This information, combined with the step size, allows to compute the ray direction, the ray length and the step vector needed for the ray traversal. The next step is to traverse the ray according to a given step size. The hardware built-in trilinear interpolation is used to obtain the value of each data sample from the 3D-Texture which stores the volume dataset. Finally, the color for each pixel is calculated by compositing the samples obtained.

The structure of the fragment shader is divided in two parts. The first one, called the ray set up, is described in Algorithm 3.

Algorithm 3 Ray Set Up in the fragment shader program.

```
Ray Set Up:

  1. Get the ray exit position from the exit texture
       exitRayPosition = getValue(current pixel position, exit texture);

  2. Get the ray starting position from the color of the current pixel
       startRayPosition = currentPixelColor;

  3. Compute the maximum ray length, which can be used to terminate the ray
       rayLine = exitRayPosition - startRayPosition;
       maxRayLength = length(rayLine);

  4. Compute the step vector
       normalizedRay = normalize(rayLine);
       stepVector = normalizedRay * stepSize;
```

For each pixel, the exit and entry points of the ray in the bounding box are fetched (instructions 1 and 2 respectively). In step 3, the line from the entry point to the exit point is calculated (rayLine) and used to compute the maximum ray length (maxRayLength). The maximum ray length is used later in the ray traversing loop (described in Algorithm 4) to test whether to terminate the loop or not. Finally, in step 4, a step vector is calculated (stepVector). The step vector is used to increment the sampling position during the ray traversal. Its length is used to accumulate the total length traversed so far.

The second part of the fragment shader program, called ray traversal, is the one that actually "shoots" the ray, i.e. collects the samples and composites them into the final pixel color (see Algorithm 4).

Algorithm 4 Ray traversal in the fragment shader program..

```
Ray Traversal:

 1. Initialize accumulation variables
    accumulatedColor = (0.0, 0.0, 0.0);
    accumulatedAlpha = 0.0;

 2. Ray traversal loop
    while(currentRayLength < maxRayLength && accumulatedAlpha < 0.95)

     2.1. Get a volume sample
          sample = getSampleValue(volume texture, current ray position);

     2.2. Get the optical properties for the sample
          colorSample = getColorValue(color transfer function, sample);
          alphaSample = getAlphaValue(opacity transfer function, sample) * stepSize;

     2.3. Update the Volume Rendering Integral
         accumulatedColor += (1.0 - accumulatedAlpha) * (colorSample * alphaSample);
          accumulatedAlpha += (1.0 - accumulatedAlpha) * alphaSample;

     2.4. Compute the next sample position
          currentRayPosition += stepVector;

     2.5. Compute the ray length traversed
          currentRayLength += stepSize;

 3. Attribute the accumulated color and opacity to the pixel
    pixelColor = accumulatedColor;
    pixelAlpha = accumulatedAlpha;
```

It is assumed that the same amount of light reaches every point inside the volume. The first step (1) is to initialize the variables where the color and alpha values will be accumulated (accumulatedColor and accumulatedAlpha respectively). In step 2, the ray traversal loop starts. The loop starts by getting a volume sample using the 3D hardware built-in interpolation (step 2.1), corresponding to the first component of the volume rendering pipeline described in section 3.1 (data access and interpolation). In step 2.2, the shading and illumination phase of the volume rendering pipeline is performed: the value is classified using the transfer functions, yielding a color and an alpha values (colorSample and alphaSample). Notice that, to compute the volume rendering integral, the alpha value is multiplied by the step size. That is because the opacity value depends on the sampling distance (given by stepSize). The volume rendering integral is updated in 2.3 (compositing phase in the volume rendering pipeline), where the color and alpha contributions of the current sample are incorporated according to Equation 2. The sampling position and the traversed ray length are refreshed in steps 2.4 and 2.5 respectively, accomplishing the advance ray position phase. This loop will be repeated till the ray exceeds the previously evaluated maximum ray length, or till the maximum opacity value is reached (ray termination component in section 3.1, implemented in the loop condition in step 2). The second loop condition (accumulatedAlpha < 0.95) actually implements the early ray termination mechanism described in section 3.2. Finally, after the loop termination, the accumulated color and opacity are displayed (step 3).

4.2 Results and Discussion

The objective of the tests described in this section is to verify the performance of the multipass ray casting implemented. The performance was evaluated by measuring the average frames per second (fps) obtained when the volume makes a complete 360^o rotation. This procedure was repeated 5 times, and the final result was obtained by computing the average of the 3 best measures.

All the results obtained in this thesis follow this description.

The results allow to evaluate how the rendering time is affected by the opacity of the volume rendered and by the step size used. The influence of the step size value in the quality of the rendered image is also discussed.

A dataset with a dimension of $512 \times 512 \times 246$ (64.487.424 voxels), requiring 61.5 MB of GPU memory was rendered to a 1000×1000 viewport. The machine used to execute the application is equipped with an *ATI Radeon HD 3450 GPU* with 1024 MB of memory and *OpenGL 2.1*. Three different opacity transfer functions were used. Each of them, when applied to the dataset, yields a volume with a different opacity level (high, medium and low). The shaders were implemented using the *GLSL* language.

In Figure 8, the number of steps processed per ray, for rendering the volume with each of the three opacity levels, can be seen.

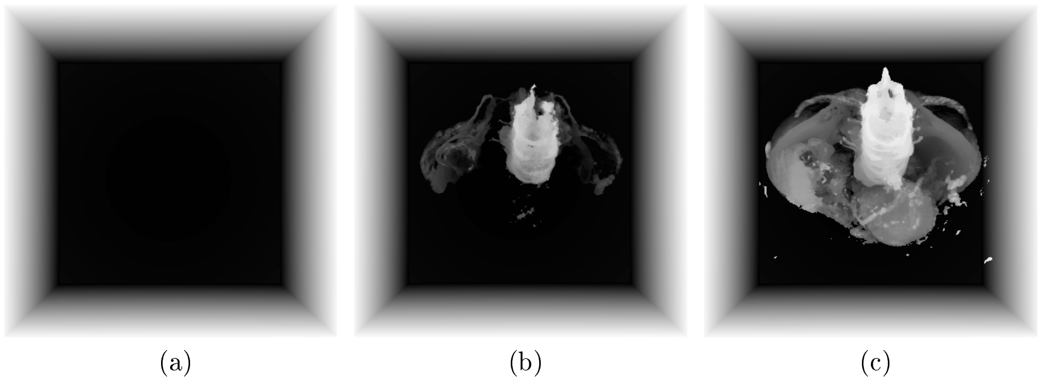


Figure 8: Comparison of the number of steps processed per ray for different opacity values. A top view was applied. The number of steps is encoded in gray scale: black corresponds to the maximum number of steps possible, according to the step size, and white corresponds to zero steps performed. Image (a) shows number of steps for the most transparent volume. Images (b) and (c) show the number of steps for the volumes with medium and high opacity, respectively.

For the low opacity transfer function (see Figure 8a), the maximum number of steps possible is often achieved (black). In Figure 8b and Figure 8c, the increasing opacity yields "whiter" parts due to a decrease in the number of steps performed per ray traversal. This decrease is justified by the use of the early ray termination mechanism (see section 3.2): the more opaque the volume is, the earlier the maximum opacity value will be achieved during ray traversal and, consequently, the sooner the ray will be truncated. Figure 9 shows the rays which were early terminated for the three different volume opacities.

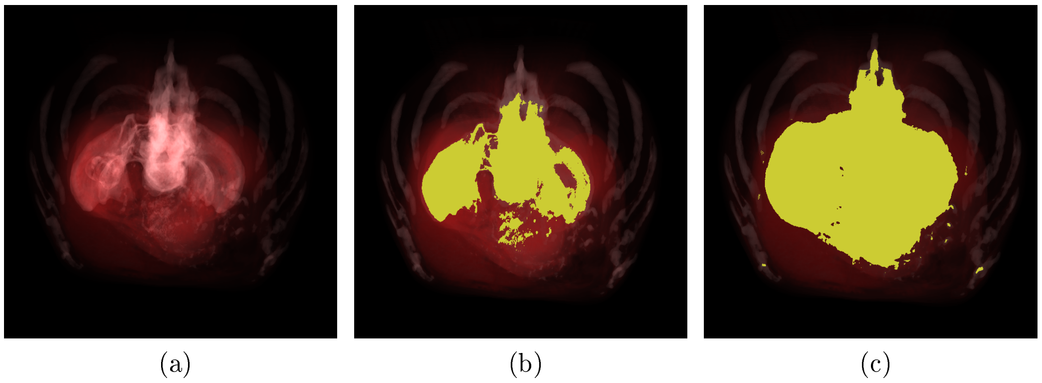
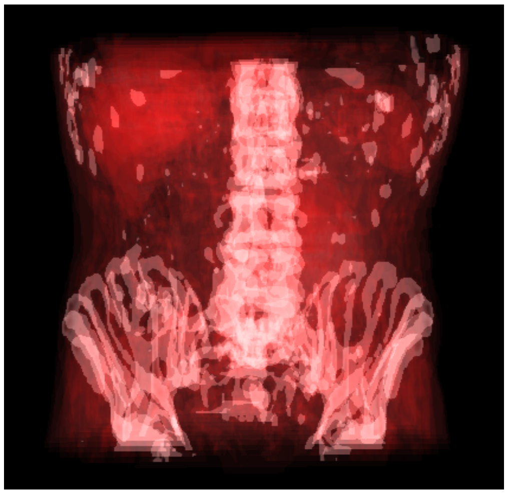
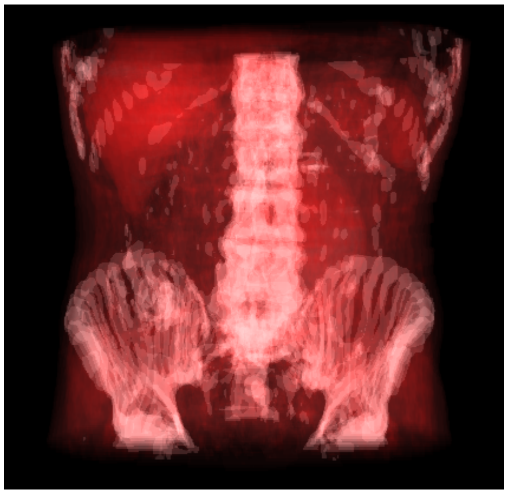


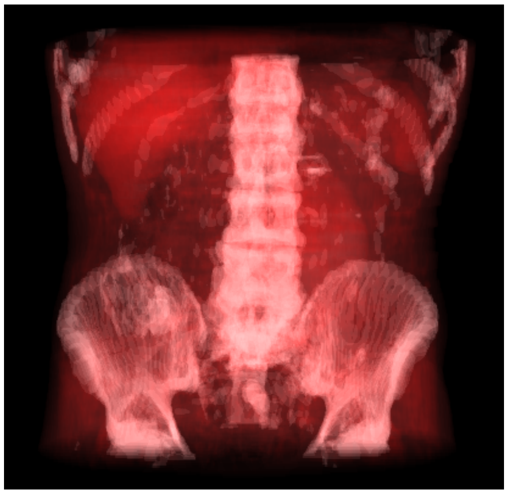
Figure 9: Comparison of the early ray termination for the three volume opacities tested. A top view was applied. The rays early terminated due to the achievement of the maximum opacity value are represented in yellow. Image (a) corresponds to the most transparent volume. Images (b) and (c) show the result for the medium and high opacity volume datasets respectively.

It can be seen that in the most transparent volume (Figure 9a), the maximum opacity value was never reached during ray traversal, and therefore none of the pixels is marked yellow. In Figure 9b, with the increased opacity of the volume, some rays are early terminated, depending on the opacity of the structures traversed by the ray. In Figure 9c, due to the high opacity of the volume, many of the rays are terminated before leaving the volume bounding box. The influence of the opacity of the volume in the final rendering time, and consequently, the influence of the early ray termination mechanism, will be appreciated later in Table 1.

The variation of the quality of the final image due to the step size value is shown in Figure 10.


(a)


(b)


(c)

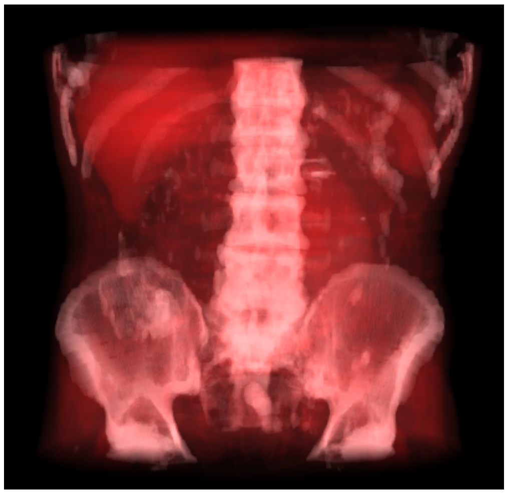

(d)

Figure 10: Images obtained for different step size values, when applying a front view. In (a) a step size of 0.050, resulting in a maximum of 34 samples per ray was used. Image (b) shows the result for a step size of 0.025, with a maximum of 69 samples per ray. Images (c) and (d) were rendered with a step size of 0.015 (115 samples) and 0.005 (346 samples) respectively.

It can be observed that the quality of the final image is dependent on the step size. Using a big step size results in an undersampled, strongly aliased image, with wood-grain effect (see Section 5). As the step size is decreased, and consequently, the number of samples per ray increases, the rendered image has a higher quality. Figures 10c and 10d show this behavior. However, as we can see in Table 1 high quality may cause loss of interactivity.

	Step Size					
	0.250	0.125	0.050	0.025	0.010	0.005
low opacity	56	43	24	13	6	4
medium opacity	56	43	25	14	7	4
high opacity	57	45	28	16	8	5
	low quality		medium quality		high quality	

Table 1: Rendering speed in frames per second achieved for the three volumes rendered, and different step sizes.

Table 1 shows the average framerate and the quality achieved, with 6 different step sizes, for each of the three volume opacities tested, when a complete rotation is applied to the dataset. The results show that for small step sizes, the rendering speed decreases dramatically. This can be seen by comparing the 25 fps framerate for the medium opacity volume with a step size of 0.050, with the 7 fps yielded for a step size of 0.010. As expected, the results also demonstrate that the number of frames per second (fps) increases with more opaque volume datasets, due to the early ray termination mechanism implemented.

5 Stochastic Jittering

The use of a large step size which allows for a faster rendering time, can cause aliasing in the final image, which results in visible artifacts named wood-grain effects. These effects can be appreciated in Figure 10a. Stochastic jittering is a technique used to hide wood-grain effects by introducing a variation in the starting position of the rays, along the viewing direction [10]. This causes the aliasing to be substituted by noise.

The variation introduced causes the samples along the ray to be offset by a random number ranging from 0 to the step size value. The samples along a ray have the same offset, while different rays are likely to have assigned a different jitter value. Consequently, the coherence between pixels which causes wood-grain effects is suppressed by noise.

5.1 Implementation

The implementation of the jittered multipass volume rendering does not differ much from the multipass ray casting algorithm described in section 4.1 (Implementation of the Multipass Ray Casting). The CPU part of the algorithm, differs in that a 2D-Texture with size 32×32 is created, containing a random number at each position. This texture is uploaded to the graphics card memory, during the application set up phase, and is later used by the fragment shader as a source of random numbers to perturb the starting positions of the rays.

In the fragment shader program, the ray set up stage differs from the implementation described in section 4.1, Algorithm 3. In the ray set-up of the jittered version, a variation ranging from 0 to the current step size value is introduced in the ray starting position. This value is calculated based on the 2D-Texture holding the random numbers, and added to the ray starting position. The ray traversal stage remains unchanged. Algorithm 5 shows the ray set up stage for the Jittered multipass ray casting.

Algorithm 5 Ray Set Up for the Jittered Multipass Ray Casting.

```
Ray Set Up:

1. Get the ray exit position from the exit texture
     exitRayPosition = getValue(current pixel position, exit texture);

2. Get the ray starting position from the cube color already interpolated
     startRayPosition = currentPixelColor;

3. Compute the ray line
     rayLine = exitRayPosition - startRayPosition;

4. Compute the step vector
     normalizedRay = normalize(RayLine);
     stepVector = normalizedRay * stepSize;

5. Introduce an offset in the ray starting position along the ray direction
     offset = getRandomNumber(jitterTex, exitFragPosition * textureSize));
     startRayPosition +=offset * stepSize * normalizedRay;

6. Compute the maximum ray length, which can be used to terminate the ray
     maxRayLength = length(rayLine) - (offset * stepSize);
```

The sequence of steps 1 to 4 yields the normalized ray direction, the step vector and its length, necessary to perform the ray traversal (for more details, see section 4.1). In step 5, an offset based on a random number extracted from the jitter texture is calculated and added to the ray starting position. The ray set up stage ends with the computation of the new ray length (step 6), and the ray traversal stage is ready to be executed.

5.2 Results and Discussion

In Figure 11, a comparison of the results obtained by rendering the volume dataset with the medium opacity transfer function (see section 4.2, Figure 6b) with multipass ray casting and jittered multipass ray casting is shown. The step size used was 0.025.

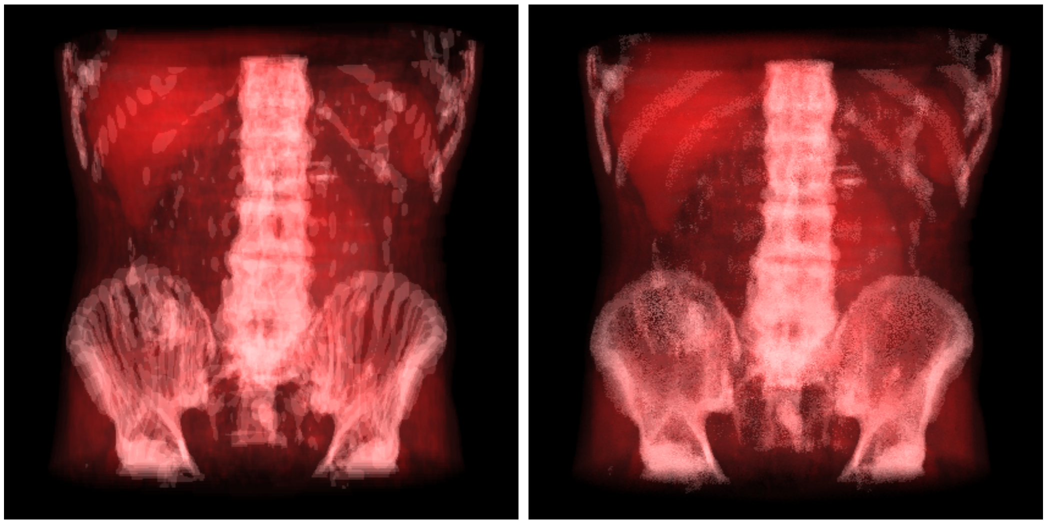


Figure 11: Comparison of the images obtained when rendering the volume dataset, with medium opacity, with multipass ray casting (a) and with jittered multipass ray casting. A step size of 0.025 was used.

The image resulting from rendering with a stochastic jitter (Figure 11b) does not contain the regular patterns which can be observed in the original image (Figure 11a). The patterns are substituted by noise caused by the random variation introduced in the ray starting positions. For a human being, the noise is easier to tolerate than the regular patterns present in the non jittered image. As a conclusion, the result obtained with the jittered ray casting is visually more acceptable.

Due to the increase of the per fragment operations, rendering a volume with jittered multipass ray casting is slower than using the multipass ray casting technique. In Table 2 a comparison of the average framerate for the medium opacity volume, obtained when applying a complete rotation to the volume, can be seen. Different step sizes were used, and combined with jittered and non jittered ray set up.

	Step Size					
	0.250	0.125	0.050	0.025	0.010	0.005
jittered	33	23	16	11	6	4
non jittered	56	43	25	14	7	4

Table 2: Comparison of the fps obtained when rendering the medium opacity volume with jittered multipass ray casting (jittered) and with multipass ray casting (non jittered), for different step sizes.

The results in Table 2 show that the jittered version is consistently slower than the non jittered version. But the decrease of the number of fps for the jittered version is not too high regarding the improvement. For example, for the images shown in Figure 5 (rendered with a step size of 0.025), the difference between the two versions is of 3 fps. As the step size decreases, the ray set up overhead introduced in the jittered multipass ray casting becomes less relevant for the rendering time and the results obtained with both techniques converge (see for example the results for a step size of 0.010 and 0.005, in Table 2).

The difference in the framerate registered with both techniques in Table 2 reflects the time for accessing the texture containing the random numbers, and for introducing the offset for the ray (see step 5 in Algorithm 4). This time difference is constant per pixel, and can be observed in Table 3, yielded by converting the results shown in Table 2 from fps to seconds.

	Step Size					
	0.250	0.125	0.050	0.025	0.010	0.005
jittered	0.030	0.043	0.063	0.091	0.167	0.250
non jittered	0.018	0.023	0.040	0.071	0.143	0.250

Table 3: Comparison of the time per frame obtained (in seconds) for rendering the medium opacity volume dataset with jittered multipass ray casting (jittered) and with multipass ray casting (non jittered), for different step sizes.

Table 3 shows that the difference in the time needed to compute a frame for each of the techniques is constant. Excluding the result obtained for a step size of 0.005, the difference is always close to 20 ms. The result for the smallest step size tested is explained for the truncation used to calculate the fps in Table 2.

6 Empirical Visualization Methods

The Ray Casting technique can be used for alternative visualization techniques which might be useful to understand the information contained in the 3D dataset, rather than to evaluate the volume rendering integral. Examples of these alternative techniques are the X-Ray and the Maximum Intensity Projection (MIP) compositing methods, often applied in medical imaging applications [18, 10]. These two techniques compute the final image as described following:

- **X-Ray.** The samples along each ray are summed up, resulting in a final image close to an X-Ray image.
- **Maximum Intensity Projection.** For each pixel, only the sample with the highest value along the ray is taken into account for the pixel color.

In this section, the implementation of the X-Ray and MIP compositing methods is described, along with the results obtained and their discussion.

6.1 X-Ray

The X-Ray compositing technique consists of directly accumulating all the sample values along each ray. For each sample taken, its opacity and color (in gray scale) are directly given by the sample value, yielding a color scheme where the most opaque values are colored in white. This leads to a final image which looks as if it was composed by X-Rays.

6.1.1 Implementation

In Algorithm 6 the ray traversal scheme used to implement the X-Ray compositing method is presented. It is assumed that the ray set-up phase was already executed, yielding all the information necessary to perform the ray traversal. The ray set-up phase can be either jittered (Algorithm 5) or non-jittered (Algorithm 3).

Algorithm 6 Ray Traversing for X-Ray compositing.

```
Ray Traversal:

  1. Initialize accumulation variables for color and opacity at zero

  2. Ray traversal loop
     while(currentRayLength < maxRayLegth && accumulatedAlpha < 0.95)

       2.1. Get a volume sample
            sample = getSampleValue(volume texture, current ray position) * stepSize;

       2.2. Update the accumulated color and opacity
            accumulatedColor += (1 - accumulatedAlpha) * volumeDataSample;
            accumulatedAlpha += (1 - accumulatedAlpha) * volumeDataSample;

       2.3. Compute the next sample position

       2.4. Compute the ray length traversed

  3. Attribute the accumulated color and opacity to the pixel
```

The first step in Algorithm 6 consists in initializing the variables where the color and the opacity values are accumulated (1.). The ray traversal loop is then executed until the ray is completely traversed, or until the early ray termination mechanism (see Section 3.2) truncates the ray (2.). In each cycle of the ray traversal, the volume is sampled at the current ray position (2.1), and the sampled value is weighted by the step size. The accumulated color and opacity are updated, based on the sample value (2.2). The sample position is incremented to the next position in the ray (2.3), and the total ray length traversed so far is recomputed (2.4). Finally, once the loop is terminated, the accumulated color and opacity are attributed to the pixel (3.).

6.1.2 Results and Discussion

In Figure 12b, an image rendered with the X-Ray technique is shown. Figure 12a shows the same perspective of the dataset rendered with the composite technique presented in Section 4.

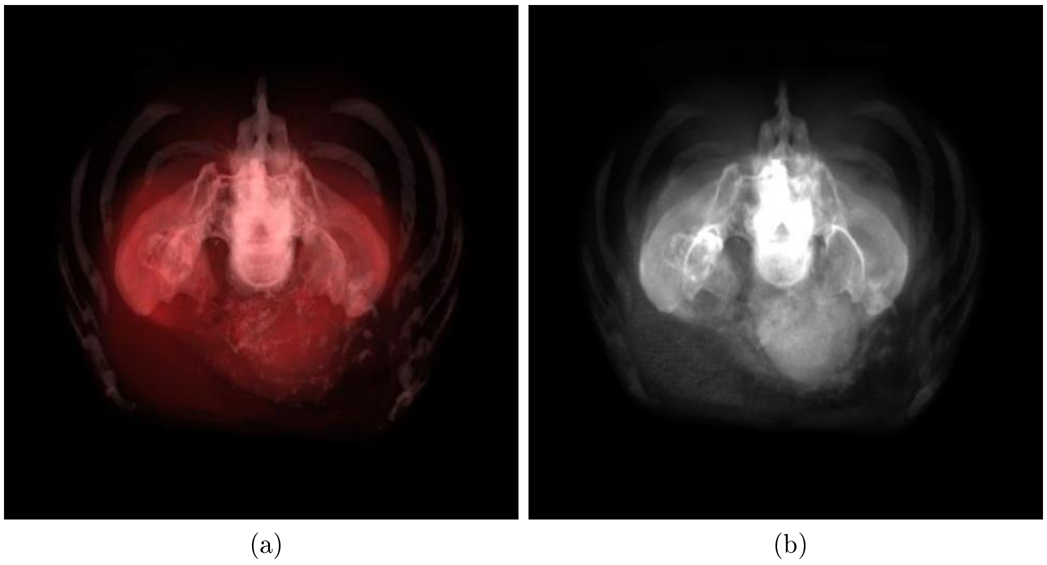


Figure 12: The same perspective of the dataset rendered with the composite (a) and X-Ray (b) techniques. Both images were obtained with a step size of 0.010.

The direct accumulation of the samples along the ray results in a final image is similar to an X-Ray image. In Table 4, the average framerate obtained when applying a complete rotation to the volume using the X-Ray compositing method, is shown.

	Step Size					
	0.250	0.125	0.050	0.025	0.010	0.005
jittered	36	28	19	16	11	7
non jittered	64	50	34	20	12	7

Table 4: Comparison of the fps obtained for rendering the volume dataset with the X-Ray compositing method. The volume was rendered with jittered and non jittered ray set-up, and different step sizes.

The results show that the performance of this technique is in line with the results achieved previously (e.g. see Table 2), with the rendering time increasing as the step size decreases. The simplicity of the operations during the ray traversal causes the time to render an image with the X-Ray compositing method to be less than the one needed to evaluate of the volume rendering integral (e.g. Table 2).

6.2 MIP

Maximum Intensity Projection is a popular compositing mode that searches for the highest sample value along a ray. The main idea is to traverse the ray, and attribute the value of the highest sample found, in gray scale, to the pixel color. MIP is mostly used to display bone structures and contrast enhanced vascular structures (vessels), where the measured intensity is significantly higher than the regular tissue value [18, 10].

6.2.1 Implementation

Algorithm 7 presents the ray traversal for the MIP compositing method.

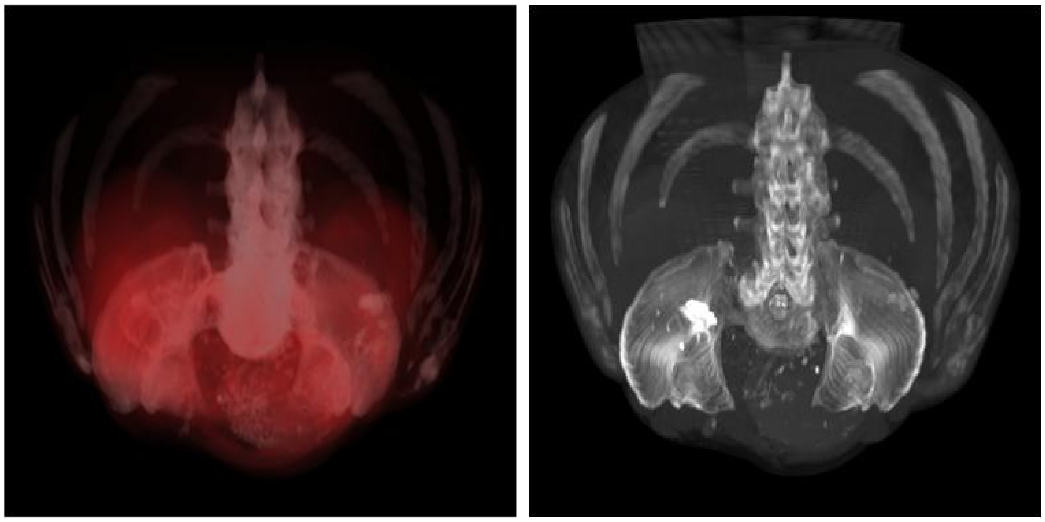
Algorithm 7 Ray Traversal for Maximum Intensity Projection.

```
Ray Traversal:

  1. Initialize the variable holding the highest sample value
     maxSample= 0.0;

  2. Ray traversal loop
     while(currentRayLength < maxRayLegth)

         2.1. Get a volume sample

         2.2. Store the current sample value if it has the highest value so far
                if (sample > maxSample)
                    maxSample = volumeDataSample;

         2.3. Compute the next sample position

         2.4. Compute the ray length traversed

  3. Attribute the color in gray scale, and set the opacity to 1 (maximum)
     pixelColor = (maxSample, maxSample, maxSample);
     pixelAlpha = 1.0;
```

The first step of the algorithm is to initialize the variable where the maximum sampled value is stored. Then, the traversal loops is executed until the ray in completely traversed (step 2.). For each loop, the volume is sampled at the current position (2.1). If the sample value is higher than the maximum sample value taken so far, the variable maxSample is refreshed with the current sample value (2.2). The sample position is incremented to the next position on the ray, and the total ray length traversed so far is computed (2.3 and 2.4). The loop is repeated until the whole ray is traversed. At last, after the loop termination, the accumulated color and opacity are attributed to the pixel (3.).

6.2.2 Results and Discussion

Figure 13b show the result of rendering the dataset using the MIP technique. In Figure 13a, the same perspective of the dataset rendered with the composite technique (see Section 4) is shown.



(a) (b)

Figure 13: The same perspective of the dataset rendered with the composite (a) and MIP (b) techniques. Both images were obtained with a step size of 0.010.

The bone and vascular structures present in the dataset were emphasized by the MIP technique. A drawback of MIP is that it does not provide the viewer with depth information of the structures that are shown, since no attenuation information is used. This creates the risk of misinterpreting the spatial relationships of different structures.

Table 5 shows the average framerate achieved while rendering a complete rotation of the dataset using the MIP compositing technique.

	Step Size					
	0.250	0.125	0.050	0.025	0.010	0.005
jittered	37	29	21	18	11	7
non jittered	66	54	37	22	12	7

Table 5: Comparison of the framerates obtained for rendering the volume dataset with the MIP compositing method. The volume was rendered with jittered and non jittered ray set-up, and different step sizes.

The framerates achieved are very similar to the ones from X-Ray, because both techniques have a similar complexity. With the decreasing of the step size the rendering time increases.

7 Rendering Segmented Datasets

When visualizing a 3D dataset with volume rendering techniques, it is common that the viewer wants to identify distinct objects of interest present in the dataset. Some of these objects can be visually differentiated by using a well tuned transfer function, which assigns different visual properties to the scalar values that characterize each of them. Nevertheless, there are cases where different areas of interest are assigned the same scalar value by the CT or MRI scan (e.g. parts of the same organ), causing them to be visually undifferentiable using a single transfer function. The result is that the user will not distinguish between those areas of interest when the dataset is rendered.

An approach to solve this problem is to identify and tag the different regions or structures present in the dataset, in a process called segmentation [9]: each voxel of the original dataset is tagged as belonging to an object contained in the volume. During the rendering process this information is used to visualize each object with different optical properties, by using a distinct transfer function for each of the objects. This allows for the differentiation of voxels that belong to distinct objects but that have the same scalar value.

The segmentation information can be represented in two major ways:

- using a *binary segmentation mask* [11]
- using a second volume dataset, usually called *object ID volume* [9]

The first option consists in representing each object by a single binary segmentation mask. This mask contains the value 0 for the voxels which do not belong to the current object, and one for the voxels which are part of it. During rendering time, this mask is used to determine whether or not the current voxel belongs to the given object. This approach requires several volumes (as many as the number of objects existent in the dataset) to be uploaded to the GPU memory in the form of 3D textures. Consequently, it is not suitable for GPU Volume Rendering, as it clearly conflicts with the limited amount of texture memory available in the graphic card.

The alternative is to use a single object ID volume which stores, for each voxel, the ID of the object it belongs to. The objects are enumerated consecutively starting with one. A single 8 bit 3D texture is enough to store the info of up to 255 objects, making this approach more suitable to be used for GPU Volume Rendering. During rendering time, the object membership of a given voxel is determined, and a specific transfer function is used according to it.

Several programs specialized on the manipulation of biomedical images (e. g. Amira) can be used to segment the dataset and make the segmentation information available for the ray casting application. However, this subject is out of the scope of this thesis and, therefore, it is assumed that the segmentation information is already available.

7.1 Boundary Interpolation

To obtain the object ID for a given fragment, the nearest neighbor interpolation can be used. Nevertheless, this interpolation causes artifacts in the final image [9] (see Figure 16a), making the object boundaries easily discernible as individual voxels. Using the hardware trilinear interpolation is not a valid solution for this problem. If three or more objects are contained in the dataset, the result of the interpolation might be incorrect. For example, in case there are three objects, numbered 1, 2 and 3, for a fragment that is placed at the border between objects 1 and 3, the value 2 would be attributed, instead of 1 or 3, which would be the correct result.

The solution to this problem was presented primarily by Tiede et al in [25] and further developed by Hadwiger et al in [9]. It consists of interpolating the object ID texture, using nearest neighbor interpolation, at the current fragment position, yielding σ, the number of the object the fragment belongs to. Then, the 8 corner points on a cube surrounding the fragment are also sampled using the nearest neighbor interpolation. For each of the surrounding points, if the membership value is σ, their value is mapped to 1, otherwise it is mapped to 0. The value of the current fragment is trilinearly interpolated in the fragment shader, based on the values mapped for the 8 surrounding points. If the computed value is higher than 0.5, then the transfer function corresponding to the object σ is applied, otherwise the fragment contribution to the final image is discarded.

7.2 Implementation

It was decided that the segmentation information should be stored in an 8 bit 3D texture, for the reasons explained previously in Section 7. Each of the objects present in the dataset is assumed to have its own transfer function. Instead of storing the transfer function of each object separately, as 1D textures, these are packed into a single 2D texture. The 2D transfer function is indexed by the ID number of the segment on the y-axis, and by the volume data scalar values on the x-axis. The signature of such a function is $(r, g, b, a) = f(segID, sampleValue)$, where (r, g, b, a) is the tuple containing the optical properties of the sample with value $sampleValue$, which belongs to segment $segID$. Figure 14 shows this indexing strategy.

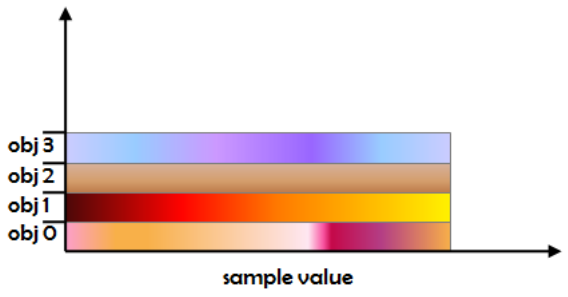


Figure 14: A 2D texture containing the color transfer function for all the objects present in the dataset.

The use of a single 2D transfer function allows to keep the 1D transfer functions of each object in a single texture, independently of the number of objects existent in the dataset. However, the size of the texture depends on the number of objects. This procedure is used for both the color and the opacity transfer functions.

The implementation of the segmented volume rendering (see Algorithm 8) was based on the composite ray traversal algorithm presented in Algorithm 4, Section 4.1.

Algorithm 8 Ray Traversal for Rendering Segmented Datasets.

```
Ray Traversal:

  1. Initialize sampling and accumulation variables

  2. Ray traversal loop

      2.1 Get a volume sample

      2.2 Get the object to which the current sample belongs
          membership = getMembershipValue(membership texture, current ray position);

      2.3 Use trilinear interpolation to determine if the current sample is rejected or not
          reject = interpolation(current ray position, membership);

      2.4 In case the current sample is not rejected
          if(reject == false)

            2.4.1 Compute the object index to access the 2D transfer function
                  membershipIndex = membership + (0.5/numberOfObjects);

            2.4.2 Get the optical properties of the sample
                  colorSample = getColorValue(color transfer function,
                                volumeDataSample, membershipIndex);

                  alphaSample = getAlphaValue(opacity transfer function,
                                volumeDataSample, membershipIndex) * stepSize;

            2.4.3 Update the VR integral

      2.5 Compute the next sample position

      2.6 Compute the ray length traversed

  3. Attribute the accumulated color and opacity to the fragment
```

As in the traditional composite method, the main idea is to traverse the volume data and, for each sample, get the respective color and opacity. The samples collected along a ray are then composited to obtain the final pixel color. However, Algorithm 8 has two major differences when compared to the compositing method presented in Algorithm 4. The first is that the current sample only contributes to the final pixel color if it belongs to the current object set (stored in membership), as explained in Section 7.1. This binary decision is taken based on trilinear interpolation implemented in the fragment shader (steps 2.3 and 2.4). The other difference refers to the access of the transfer functions: to correctly index the y-axis of the transfer function, the membershipIndex value is computed in step 2.4.1, based on the number of the current object set (membership) and on the total number of objects existing in the dataset (numberOfObjects). Then, in step 2.4.2, the membership value, along with the value of the current volume sample, collected in step 2.1, are used to get the optical properties of the sample. The volume rendering integral is updated accordingly in 2.4.3.

7.3 Results and Discussion

The segmentation information was represented by an object ID volume with the same resolution of the original 3D dataset. The object ID volume identifies four main areas of interest (see Figure 15b):

- all the voxels which belong to the empty space surrounding the abdomen were tagged with 0;
- 1 identifies the voxels that form the abdomen (in red and white in Figure 15b);
- 2 is the value of the aorta artery (in blue in Figure 15b);
- 3 tags the pathological aorta tissue (in green in Figure 15b);

Figure 15 shows a comparison of rendering a segmented dataset with and without segmentation information.

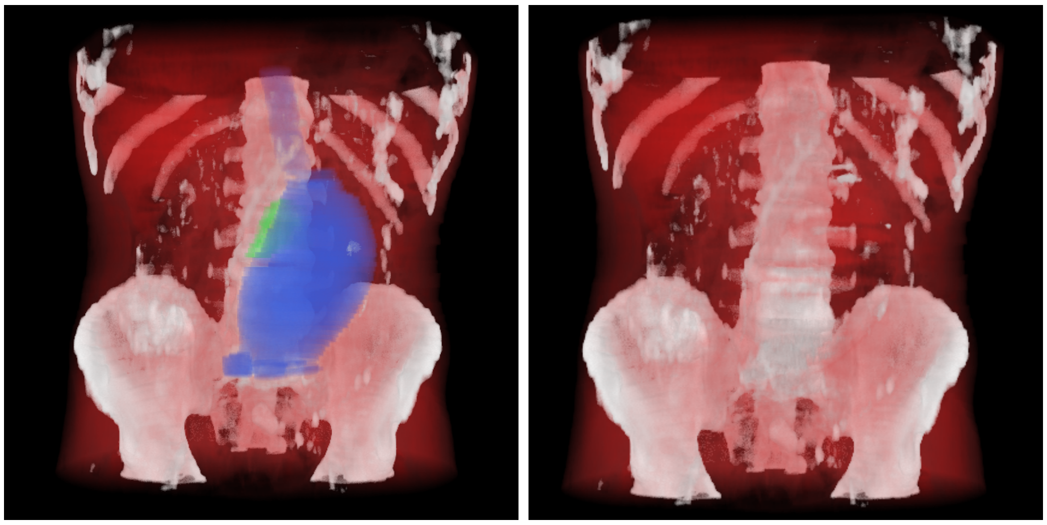

(a) (b)

Figure 15: Comparison of the volume rendered with (a) and without (b) segmentation information.

It can be seen, in Figure 15b, that the voxels that have the same scalar value but belong to different objects of interest are now visually differentiated using multiple transfer function and the segmented data. This contrasts with the result in Figure 15b, where a single transfer functions and no segmentation information were used.

Figure 16 depicts the influence of boundary filtering on the final result.

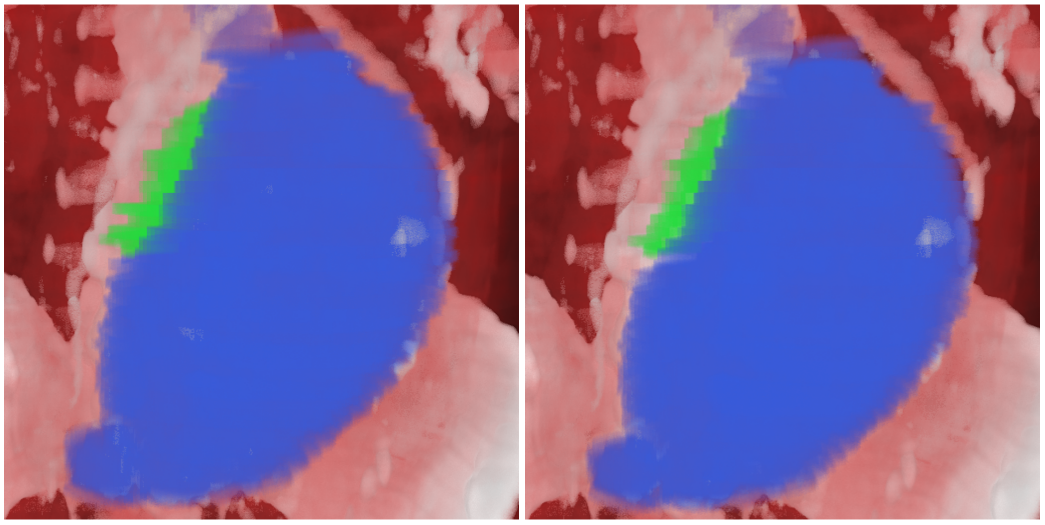

(a) (b)

Figure 16: Object boundaries with voxel resolution (a) versus object boundaries determined per-fragment with trilinear interpolation (b).

Figure 16a was rendered using boundaries with voxel resolution. When compared to Figure 16b, rendered with boundaries obtained at the pixel resolution, more artifacts resulting from boundary interpolation are visible in Figure 16a. These are specially noticeable in the pathological aorta tissue (in green in Figure 16). Once again, the improvement in the quality of the final image has a cost on the framerate achieved, as it can be seen in Table 6.

	Step Size					
	0.250	0.125	0.050	0.025	0.010	0.005
voxel resolution	37	29	21	16	8	5
pixel resolution	24	18	11	6	3	2

Table 6: Comparison of the framerates obtained for rendering the volume dataset with boundaries with voxel and with pixel resolution. The volume was rendered with jittered ray set-up, and different step sizes.

When rendering a complete rotation of the dataset with the per fragment boundary interpolation, the framerate is consistently lower than when the voxel boundary resolution is used. This is due to the computational weight of the interpolation mechanism implemented in the fragment shader, which requires eight additional memory accesses to the object ID volume texture and some extra computations.

8 Interactive Highlighting of Objects of Interest

When visualizing segmented datasets, the object of interest on which the user wants to focus can be occluded by other objects in the dataset. A possible solution for this problem is to set a low opacity for the occluding objects, by manually changing their transfer function. However, this would require expertise from the users on the scene setting, i.e. knowing which objects occlude the object of interest. Moreover, such manual tuning could take up to several minutes, which can be considered long and tedious for an interactive user interface.

In this section, a solution to this problem is proposed. The system is endowed with a functionality which enables to quickly highlight specific segmented objects, without having to manually check for the objects that are occluding the area of interest, and having to adapt their transfer functions. Shortening the highlight process, making it semi-automatic, can also help the user to be aware of the context in which that same object is placed in, by being able to see the change, from the image with no highlight to an image where a specific object is highlighted.

The main idea is to allow the user to pick the objects of interest with the mouse. Upon a mouse command, one or more objects located below the mouse cursor are highlighted. For this, a highlight detection pass is executed before the ray casting pass computes the final image. In the highlight pass a ray is shot from the cursor position and, using the GPU, the IDs of the objects traversed are collected (see Figure 17).

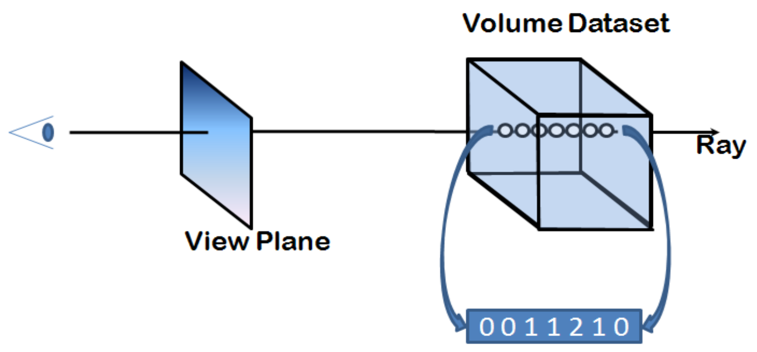


Figure 17: Extraction of the membership the samples taken by the highlight ray.

The result is written into a texture, which is read back by the CPU. This information is then used in the ray casting pass to highlight objects present in the dataset, according to some additional criteria. For example, to highlight the n^{th} object which is traversed by the ray, or to highlight all the objects traversed by the ray. The actual mechanism to select a given object is application dependent and out of the scope of this thesis, which focuses on GPU and rendering functionalities.

8.1 Implementation

The strategy used to implement the highlight mechanism described above is to render a line, corresponding to the ray, into a texture. The highlight ray is mapped to this texture and, therefore, the number of pixels of this texture will determine the resolution of the highlighting ray. Each pixel corresponds to a sampling point along the ray and, after the execution of the highlight pass, they will hold the membership of the object to which the sample belongs.

The display loop presented in Section 4.1, Algorithm 2, which drives the operations performed per frame, had to be adapted to execute the highlighting detection pass before the ray casting rendering (see Algorithm 9).

Algorithm 9 Ray Casting algorithm from the OpenGL side.

```
For each frame:

  1. Render the back face of the bounding box to a 2D texture
        using the OpenGL fixed functionality

  2. Render the front face of the bounding box to a 2D texture
        using the OpenGL fixed functionality

  3. if (Highlight Click)

        3.1. Enable the Highlight Shaders

        3.2. Render the line corresponding to the highlight ray to a 1D texture
                using the Highlight Shaders

        3.3. Disable the Highlight Shaders

  4. Enable the Ray Casting Shaders

  5. Perform the Ray Casting using the Ray Casting Shaders

  6. Disable the Ray Casting Shaders

  7. Enable the OpenGL fixed functionality
```

The first modification is the rendering of the front face to a 2D texture (step 2). The front face colors, which encode the ray starting positions, can be passed to the ray casting shaders using the vertices of the bounding box surrounding the volume, and then be used directly by the ray casting shader. However, in the highlight pass, a line corresponding to the ray is drawn, instead of the volume bounding box. This requires the front face to be also rendered and stored in a 2D texture. It is then passed to the highlight shader in order to determine the ray starting position. The second modification is the execution of a highlight pass upon a user request, triggered by a mouse action (step 3). It consists of enabling the highlight shaders (step 3.1), render the highlight ray to a 1D texture (step 3.2), and disable the highlight shaders (step 3.3). This sequence of three steps detects the IDs of the objects which are traversed by the highlight ray, and stores them in a 1D texture. This information is used in the ray casting pass.

The highlight shader is described below.

Algorithm 10 Fragment shader responsible for detecting the ID information.

```
At the current sampling point of the ray:

  1. Get the ray exit position from the exit texture
       exitRayPosition = getValue(current pixel position, exit texture);

  2. Get the ray starting position from the color of the current pixel
       startRayPosition = getValue(current pixel position, entry texture);

  3. Compute the ray length
       rayLine = exitRayPosition - startRayPosition;

  4. Compute the step vector
       stepVector = rayLine / textureSize;

  5. Attribute the current ray position
       vec3 currentRayPosition = startRayPosition.xyz + pixelIndex * stepVector;

  6. Get a volume sample
       membership = texture3D(membershipTexture, currentRayPosition).a;

  7. Write back the membership in the texture
       pixelAlpha = membership
```

Algorithm 10 is executed in parallel for each sample of the highlight ray, once the line corresponding to the ray is drawn by OpenGL (step 3.2, Algorithm 9). In steps 1 and 2, the ray entry and exit positions are computed, by fetching the colors which correspond to the pixel where the user required the highlight ray in the front and back face textures. Steps 3 and 4 yield the stepVector, terminating the ray set up phase. Notice that the stepVector, which contains the distance between two consecutive samples, is computed by dividing the rayLine for the number of pixels contained in the ray 1D texture (textureSize). The sampling position corresponding to the current fragment is computed in step 5: to the ray starting position, the distance from the ray starting position to the current sampling position is added. This distance is obtained by multiplying the index of the current pixel (which can range from 1 to textureSize), by the stepVector. Using the current ray position, the membership of the current sampling position is fetched, in step 6, from the object ID volume. At last, in step 7, the membership is written to the 1D texture holding the final result.

The ray traversal of the composite fragment shader, presented in Algorithm 8, Section 7.2, was also modified to highlight the selected objects by varying their color and opacity.

Algorithm 11 Changes made to the composite fragment shader, to support objects highlight.

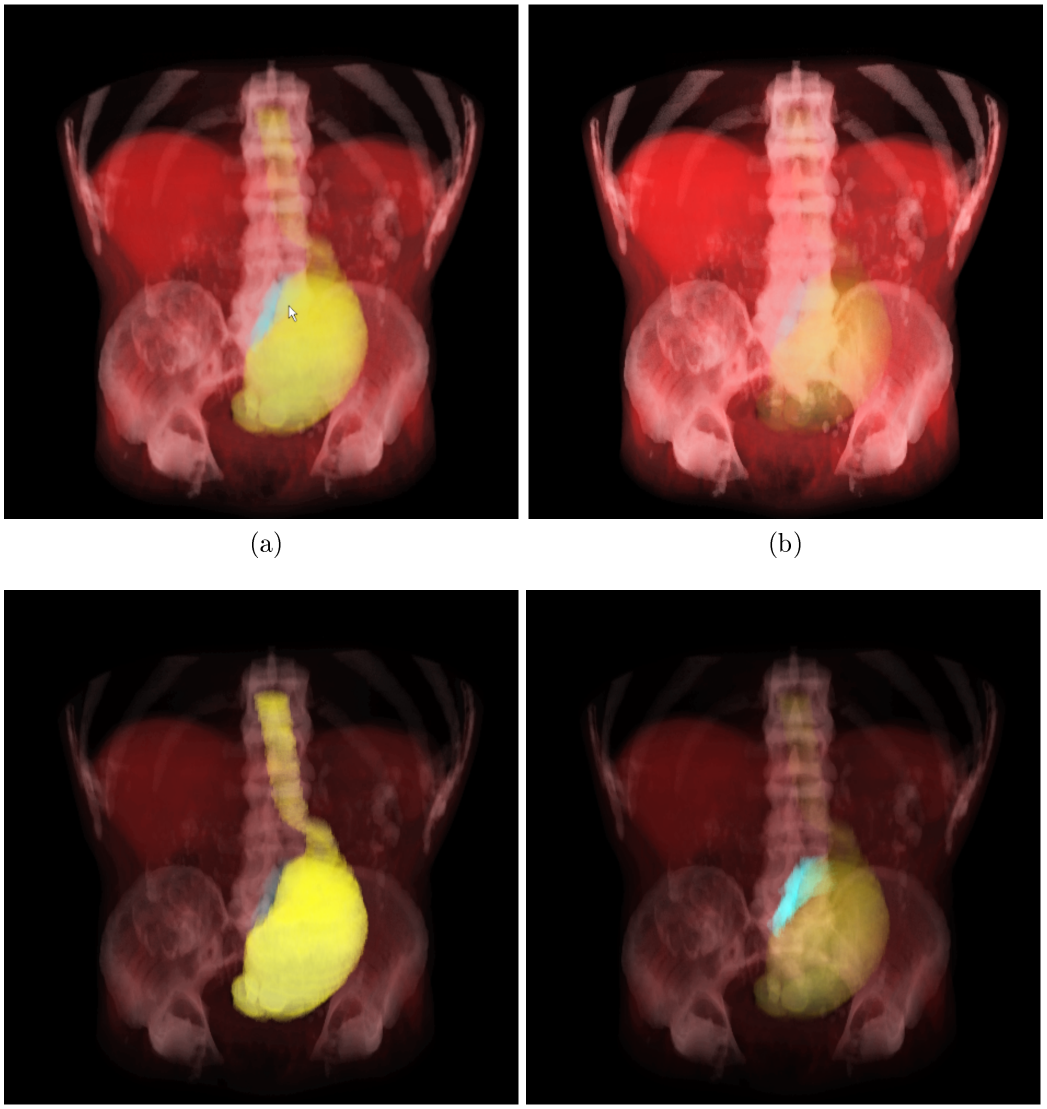
```
Ray Traversal:

  1. Initialize sampling and accumulation variables

  2. Ray traversal loop

       (...)

       2.4 In case the current sample is not rejected by the boundary interpolation

             2.4.1 Compute the object index to access the 2D transfer function
                   membershipIndex = membership + (0.5/numberOfObjects);

             2.4.2 Get the optical properties of the sample
                   colorSample = getColorValue(color transfer function,
                                 volumeDataSample, membershipIndex);

                   alphaSample = getAlphaValue(opacity transfer function,
                                 volumeDataSample, membershipIndex) * stepSize;

             2.4.3 if (highlight current fragment)
                   colorSample *= enhancementFactor;
                   alphaSample *= enhancementFactor;

             2.4.4 else (fade current fragment)
                   colorSample *= attenuationFactor;
                   alphaSample *= attenuationFactor;

             2.4.3 Update the VR integral
```

During the ray traversal loop, if the contribution of the current sample is not rejected by the boundary interpolation (step 2.4), its color and opacity are fetched from the textures containing the transfer functions (in steps 2.4.1 and 2.4.2 like in Algorithm 8). The highlight is then performed: if the current sample is to be highlighted, then its color and opacity samples are enhanced (step 2.4.3), otherwise, the color and opacity are attenuated (step 2.4.4).

8.2 Results and Discussion

The dataset used to test the highlight mechanism was identical to the one used in Section 7 (dimensions 512 x 512 x 246), but with a different transfer function. An object ID dataset with the same resolution as the original dataset, containing segmentation information, was also used. The aorta artery, and the pathological aorta tissue, are represented by the transfer function in yellow and blue respectively. The abdomen, mainly composed of bone and organs, is represented in white and red (see Figure 18a).

During the visualization of the segmented dataset, the shooting of a highlight ray was requested on the pixel where the mouse cursor was placed (see Figure 18a). The highlight ray detected three objects while traversing the volume: the abdomen, the aorta artery, and the pathological tissue, by this order. Based on this information, it was then requested that each of the objects was highlighted individually, by the order of detection. The result is depicted in Figures 18b, 18c, and 18d. In Figure 18b, the abdomen is highlighted. Figure 18c shows the highlighting of the aorta artery and, in Figure 18d, the focus is put on the pathological aorta tissue. A ray casting step size of 0.005 was used, as well as per pixel boundary interpolation and jittered ray set up. The resolution of the highlight ray used was 887, the maximum number of voxels which can be traversed in the current dataset . The attenuation and increasing factors used (see Algorithm 11) were 0.3 and 1.5 respectively.


(a) (b)

(c) (d)

Figure 18: Highlighting different objects of interest.

The results show that the mechanism implemented is effective in detecting and highlighting objects of interest which are placed in the direction of the pixel where the highlight ray is requested. When comparing Figure 18a with Figure 18d, it is possible to realize that the mechanism allowed the user to focus on an occluded object. In the former, the pathological tissue (in blue) is partially occluded by the aorta artery (in yellow), making it difficult to distinguish the contours of the object. In the latter it is possible to see more clearly the extent of the pathological tissue. In Table 7, the average fps achieved using the highlight mechanism are shown.

	Ray Casting Step Size					
	0.250	0.125	0.050	0.025	0.010	0.005
highlight mechanism	24	18	11	6	3	2
no highlight mechanism	24	18	11	6	3	2

Table 7: Comparison of the framerates obtained for rendering the volume dataset with object highlight, with boundaries at pixel resolution. The volume was rendered with jittered ray set-up, and different step sizes.

These results show that the overhead of the highlight mechanism in the composite fragment shader is negligible, as the results are identical to those achieved without highlight, shown in the bottom line (the results in this line were previously presented in Table 6, Section 7.3).

The scope of this section is to state the development of a mechanism capable of using the GPU for detecting the segmented objects placed in the direction of a given pixel. However, further investigation can be done about how to use the information produced by the highlight pass. For example, different attenuation/enhancement factors can be explored. Setting the attenuation factor to zero, for the objects which are not to be highlighted, would allow the user to visualize only the object of interest. Furthermore, the value of the attenuation/enhancement factors used for this dataset, might not suit datasets with distinct visual properties. In this case, an automatic or semi-automatic mechanism for the attribution of these factors could be developed. The question about which are the objects to be highlighted, once the highlight ray returns the information of the objects which are traversed, is also a subject where further investigation can be made.

9 Conclusion and Outlook

In this thesis, a set of shaders used to perform volume ray casting on the GPU was presented. Comparisons were performed among different compositing techniques, and the effect of different step sizes along each ray was also evaluated. Larger step sizes result in faster rendering, but aliasing becomes apparent in the resulting images. Jittering of the origin of the rays was introduced to minimize this problem, using this stochastic process to trade noise for aliasing, with a small computational cost. Noise is more easily tolerated by the Human Visual System than the visible artifacts caused by the aliasing, which allows for the utilization of larger step sizes achieving the same subjective image quality.

Moreover, segmentation data was used to identify distinct objects of interest present in the dataset, using multiple transfer functions. Per pixel boundary interpolation was implemented, improving image quality on boundary areas. A highlighting mechanism was developed allowing the user to quickly highlight specific objects, by shooting a highlight ray into the volume.

This project was motivated by the creation of a Human Atlas visualization tool, based on volume rendering techniques, which could allow a real time interaction. The large rendering times obtained by using traditional CPU ray casting, prohibitive for a real time visualization, and the availability of extremely efficient and highly programmable GPUs, drove the project to the field of GPU ray casting. The results achieved so far are satisfactory regarding both image quality and rendering time. The parallel nature of the ray casting algorithm, where each ray is processed independently of the other rays, suggests that the shaders implemented would scale well when the number of rays shot increases, if the GPU has the resources to process all the rays in parallel.

References

[1] James F. Blinn. Light reflection functions for simulation of clouds and dusty surfaces. *SIGGRAPH Comput. Graph.*, 16(3):21–29, 1982.

[2] Stefan Bruckner. *Efficient Volume Visualization of Large Medical Datasets - Concepts and Algorithms*. VDM Verlag, Saarbrücken, Germany, 2008.

[3] Brian Cabral, Nancy Cam, and Jim Foran. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. *Proc. of the 1994 symp. on volume visualization*, pages 91–98, 1994.

[4] Timothy J. Cullip and Ulrich Neumann. Accelerating volume reconstruction with 3d texture hardware. *Technical Report: Univ. of North Carolina at Chapel Hill*, 1994.

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

[6] Robert A. Drebin, Loren Carpenter, and Pat Hanrahan. Volume rendering. *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 65–74, 1988.

[7] Klaus Engel, Martin Kraus, and Thomas Ertl. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 9–16, 2001.

[8] T. Höllerer H.-C. Hege and D. Stalling. Volume rendering - mathematical models and algorithmic aspects. *W. Nagel (ed.), Partielle*

Differentialgleichungen, Numerik und Anwendungen, Konferenzen des Forschungszentrums Jülich, pages 227–255, 1996.

[9] Markus Hadwiger, Christoph Berger, and Helwig Hauser. High-quality two-level volume rendering of segmented data sets on consumer graphics hardware. *Proceedings of the 14th IEEE Visualization 2003 (VIS'03)*, page 40, 2003.

[10] Markus Hadwiger, Joe M. Kniss, Christof Rezk-salama, Daniel Weiskopf, and Klaus Engel. *Real-time Volume Graphics*. A. K. Peters, Ltd., USA, 2006.

[11] Gabor T. Herman Jayaram K. Udupa. *3D Imaging in Medicine*. CRC Press, 1999.

[12] Christopher Johnson and Charles Hansen. *Visualization Handbook*. Academic Press, Inc., USA, 2004.

[13] James T. Kajiya and Brian P Von Herzen. Ray tracing volume densities. *SIGGRAPH Comput. Graph.*, 18(3):165–174, 1984.

[14] J. Kruger and R. Westermann. Acceleration techniques for gpu-based volume rendering. *Proc. of the 14th IEEE Visualization 2003 (VIS'03)*, page 38, 2003.

[15] Eric C. La Mar, Bernd Hamann, and Kenneth I. Joy. Multiresolution techniques for interactive texture-based volume visualization. *Proc. of the 10th IEEE Visualization 1999 Conf. (VIS '99)*, 1999.

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

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

[18] B. Preim and D. Bartz. *Visualization in Medicine: Theory, Algorithms, and Applications*. Elsevier, 2007.

[19] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume on standard pc graphics hardware using multi-textures and multi-stage rasterization. *Proc. of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, pages 109–118, 2000.

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

[21] Paolo Sabella. A rendering algorithm for visualizing 3d scalar fields. *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 51–58, 1988.

[22] H. Scharsach. Advanced gpu raycasting. *Proc. of CESCG*, pages 69–76, 2005.

[23] Ramin Shahidi. Surface rendering versus volume rendering in medical imaging: techniques and applications (panel). *Proceedings of the 7th conference on Visualization '96*, pages 439–440, 1996.

[24] S. Stegmaier, M. Strengert, T. Klein, and T. Ertl. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. *Volume Graphics, 2005. Fourth Int. Workshop*, pages 187–241, June 2005.

[25] Ulf Tiede, Thomas Schiemann, and Karl Heinz Höhne. High quality rendering of attributed volume data. *Proceedings of the conference on Visualization '98*, pages 255–262, 1998.

[26] Craig Upson and Michael Keeler. V-buffer: visible volume rendering. *Proceedings of the 15th annual conference on Computer graphics and interactive techniques*, pages 59–64, 1988.

[27] Daniel Weiskopf. *GPU-Based Interactive Visualization Techniques (Mathematics + Visualization).* Springer-Verlag Berlin Heidelberg, 2007.

[28] Rüdiger Westermann and Thomas Ertl. Efficiently using graphics hardware in volume rendering applications. *Proc. of the 25th Annual conf. on Computer Graphics and Interactive Techniques*, pages 169–177, 1998.

[29] Lee Westover. Footprint evaluation for volume rendering. *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*, pages 367–376, 1990.

[30] Turner Whitted. An improved illumination model for shaded display. *Commun. ACM*, 23(6):343–349, 1980.