TECHNISCHE UNIVERSITEIT EINDHOVEN
Department of Mathematics and Computer Science

MASTER'S THESIS

Constructive Solid Geometry and Volume Rendering

by
M.A. Termeer

Supervisors:

Dr. Ir. Javier Olivan-Bescós (Philips Medical Systems)
Dr. Ir. Alexandru Telea (TU/e)
Dr. Ir. Anna Vilanova Bartoli (TU/e)

Eindhoven, August 2005

Constructive Solid Geometry and Volume Rendering

or

Volume Clipping

A thesis
submitted in partial fulfilment
of the requirements for the degree
of
Master of Science
at the
Eindhoven University of Technology
by

Maurice Termeer

Department of Mathematics and Computer Science

TU/e technische universiteit eindhoven

PHILIPS

Eindhoven, The Netherlands

July 15, 2005

Abstract

Shaded direct volume rendering is a common volume visualization technique. Whenever there is a desire to mask out a part of the volume data set, this technique can be combined with volume clipping. However, the combination of volume clipping with shaded direct volume rendering leads to visible artifacts in the final image if no special care is taken. In this thesis a depth-based volume clipping algorithm is presented that minimizes these artifacts, while maximizing both performance and interactivity. The resulting algorithm allows the clipping volume to be specified as a high-resolution polygon mesh, which has no direct correlation with the volume data. One of the advantages of this method is that both the resolution and orientation of the clipping volume can be modified independently from the volume data set during visualization without a decrease in performance. To achieve a high performance, the possibilities of consumer-level graphics hardware are exploited.

The algorithm is integrated into two different ray casting-based shaded direct volume rendering implementations. The first is an existing software-based implementation. The optimizations it contains, both for performance and for image quality, are described, combined with the modifications to support volume clipping. The second volume renderer is a new implementation that uses consumer-level graphics hardware. This volume renderer was developed as a part of this project. Since there are currently very few implementations of ray casting-based volume rendering on graphics hardware, this offers many interesting research possibilities. A number of optimizations specific for ray casting-based volume rendering on graphics hardware are discussed, as well as the integration with the volume clipping algorithm. The result is a volume renderer which is able to offer a trade-off between performance and image quality. A comparison of both volume renderers is given, together with an evaluation of the volume clipping technique.

Samenvatting

Shaded direct volume rendering is een veel gebruikt algoritme in het gebied van volume visualisatie. Dit algoritme kan worden gecombineerd met volume clipping om een specifiek gedeelte van de volume data set te maskeren, zodat deze niet meer zichtbaar is. Wanneer shaded direct volume rendering en volume clipping worden gecombineerd, dient er rekening mee worden gehouden dat er diverse artefacten in het uiteindelijke beeld op kunnen treden. In deze scriptie wordt een algoritme beschreven voor volume clipping dat met behulp van de diepte structuur van het clipping object dergelijke artefacten minimaliseert, terwijl de prestaties en interactiviteit met de data wordt gemaximaliseerd. Het resulterende algoritme accepteert het clipping object als een polygonale mesh van hoge resolutie, welke geen directe relatie heeft met de volume data set. Een van de voordelen hiervan is dat zowel de resolutie als de oriëntatie van het clipping object onafhankelijk van de volume data set kunnen worden gekozen en zelfs worden veranderd tijdens de visualisatie, zonder dat de prestaties afnemen. Om hoge prestaties te leveren, maakt het algoritme gebruik van de mogelijkheden van grafische hardware, zoals die momenteel aanwezig is op de markt voor consumenten.

Het algoritme is geïntegreerd met twee verschillende shaded direct volume rendering implementaties, welke beide gebaseerd zijn op ray casting technieken. De eerste is een bestaande implementatie die geen gebruik maakt van speciale hardware. Om een hoge beeld kwaliteit tezamen met hoge prestaties te kunnen leveren, bevat de volume renderer een aantal optimalisaties. Deze worden beschreven, samen met de nodige wijzigingen om volume clipping mogelijk te maken. De tweede volume renderer is een eigen implementatie die wel gebruik maakt van grafische hardware, welke gedurende dit project is ontwikkeld. Dit is een interessant onderwerp, omdat er relatief weinig onderzoek is gedaan in het gebied van ray casting gebaseerd volume rendering met behulp van grafische hardware. De optimalisaties die specifiek zijn voor deze manier van volume rendering worden besproken, tezamen met de integratie met het volume clipping algoritme. Het resultaat is een volume renderer welke schaalbaar is in beeld kwaliteit tegenover prestaties. Tot slot worden de twee volume rendering implementaties met elkaar vergeleken en wordt een evaluatie gegeven van het volume clipping algoritme.

Acknowledgements

I would like to thank Dr. Ir. Javier Olivan-Bescos, Dr. Ir. Alexandru Telea and Hubrecht de Bliek for their help and supervision during this project, Dr. Ir. Gundolf Kiefer for his support on the work with the DirectCaster, Dr. Ir. Frans Gerritsen for his additional support and Prof. Dr. Ir. Peter Hilbers for getting me in contact with Philips Medical Systems.

Contents

List of Figures

List of Tables

Chapter 1
Introduction

The human visual system is a very powerful mechanism. It can process large amounts of data and extract various kinds of information. Scientists have used visualization since long ago to exploit the human visual system by turning abstract data into images and enabling a user to literally see patterns and structure in the data. Often various interactions with the visualization are offered to increase the efficiency of comprehending the data.

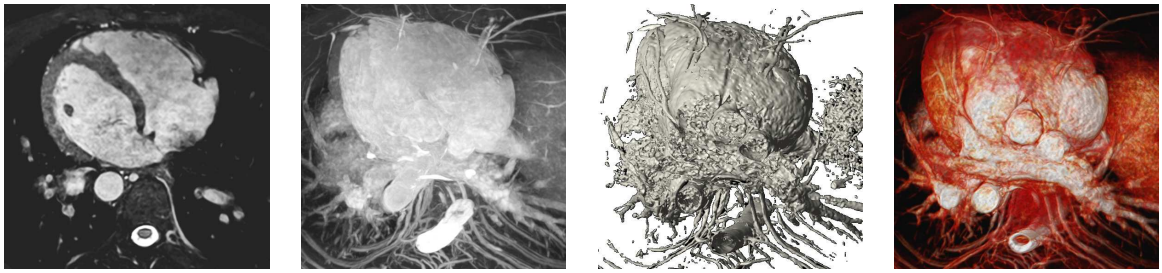
1.1 Volume Visualization

Volume visualization is characterized by that the data to be visualized is of a volumetric nature. Put differently, the data represents a three dimensional scalar field defining a mapping $v : \mathbb{R}^3 \rightarrow \mathbb{R}$. If the data is uniform, which is a common property, it can be stored as a three dimensional matrix. The elements of the matrix are often called *voxels* and they usually contain a single scalar, the voxel intensity value. This kind of data appears in various areas of expertise, such as seismic scans of a planet surface structure in the oil and gas industry, simulations of physical processes or in medical imaging where they are produced by Computed Tomography (CT) and Magnetic Resonance (MR) scanners.

The purpose of volume visualization is no different than that of most types of visualization; to gain understanding of the data. For volume visualization the focus is on helping to understand the spatial structure and orientation of the data. The volumetric nature of the data imposes a problem on the process of visualization, as most display devices can display only two dimensional images. Many solutions to this problem exist, these form a set of volume visualization methods. A short and incomplete list of methods is given here. An overview of available volume visualization methods is also given in [Yagel, 2000] and [Kaufman, 2000].

A simple visualization method is to display a single two dimensional slice of the data by directly mapping the intensity values to a grayscale gradient. By offering interactivity that enables the user to browse through the individual slices, the user can still gain insight by viewing all of the volume data. This method is frequently used in practice. Variations of this method include the planar reformatting method, where the data along an arbitrarily positioned plane is projected on the screen. This allows the user to view the data from different angles, although still a single slice at a time. The curvilinear reformatting method allows the use of a ribbon instead of a plane, meaning that the plane may be bent along one dimension. Figure 1.1 (a) gives an example of a planar reformatting rendering of a human hart.

Another class of methods projects the volume data on the screen. There are numerous ways to do this. The Maximum Intensity Projection (MIP) method projects the maximum intensity value of the



(a) (b) (c) (d)

Figure 1.1: Different methods of volume visualization applied to a volume data set of the human hart; planar reformatting (a), maximum intensity projection (b), isosurface rendering (c) and direct volume rendering (d).

volume data along the viewing direction on each pixel of the screen. Variations of this method project the minimum or average intensity values. Figure 1.1 (b) shows an example of a maximum intensity projection.

Isosurface rendering is a visualization modality that aims at visualizing an isosurface defined within a volume dataset. The isosurface is characterized by a volume, a three-dimensional interpolation function and a threshold value. The resulting image shows a surface along which the voxel intensities are equal to the threshold value. All other areas of the volume data are left completely transparent. Figure 1.1 (c) shows this technique in action.

The last method of this short list is that of Direct Volume Rendering (DVR). This method is a generalization of the isosurface rendering method. With DVR, a *transfer function* is used to map a voxel value to an opacity and color. The transfer function defines a mapping $f : \mathbb{R} \rightarrow \mathbb{S}$, where $\mathbb{S}$ denotes a color and opacity space, such as RGBA (Red/Green/Blue/Alpha). For isosurface rendering, the opacity component of this mapping is defined as a step function. The additional functionality offers a much wider range of visualizations, but also increases the complexity of the rendering algorithm. An example of a volume rendered with DVR is shown in figure 1.1 (d). Because DVR is the volume visualization method that is the focus of this project, a more elaborate description is given in the next section.

1.2 Direct Volume Rendering

The purpose of Direct Volume Rendering is to create a rendering of a volume data set where each voxel intensity is mapped to a color and opacity. Ideally, different regions in the volume correspond to different voxel intensities. This allows to visualize multiple voxel intensity ranges simultaneously, while still being able to separate them in the final image. For medical volume data for example, one can use this technique to give blood vessels and skin tissue different colors and opacities.

A common implementation of DVR is based on ray casting, but many different methods for DVR exist [Lacroute and Levoy, 1994], [Levoy, 1988], [Kaufman et al., 2000], [Hauser et al., 2000], [Nielson and Hamann, 1990]. For each pixel of the viewing plane, a ray is casted in the viewing direction. When this ray intersects the volume, samples are taken at a regular distance. Figure 1.2 shows this in a schematic way. In this figure, the samples appear at a regular distance from the viewer, and the samples are taken not at the voxel boundaries but at arbitrary locations. Because the sample locations are generally not at

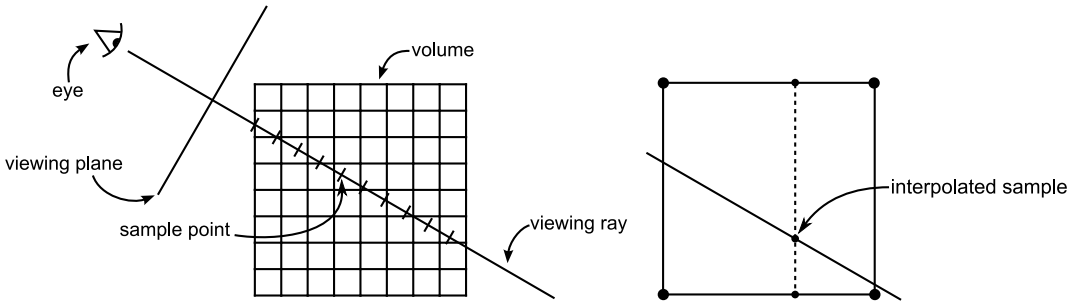


Figure 1.2: Schematic overview of Direct Volume Rendering; the left image shows how samples are taken along a ray while the right image shows how the actual sample value is computed using interpolation.

the exact voxel positions, an interpolation function is used. Using trilinear interpolation to compute the interpolated voxel intensity is common practice and gives good results. Different sampling strategies also exist to reduce the required amount of interpolation. An example is object-aligned sampling, where all the samples are taken at the voxel boundaries. An advantage of this method is that it requires only bilinear interpolation.

Once the voxel intensity is interpolated for a particular sample, a *transfer function* is used to map this value to an opacity and color value. This transfer function is a one-dimensional function that allows certain voxel intensity ranges to be mapped to different colors and opacities, thereby controlling the visualization result. To compute the final color for a pixel, all samples along the ray need to be accumulated. The accumulation can be done either before or after applying the transfer function. Considering the latter method, the opacity of the current ray can be updated when accumulating a sample using the following formula.

$$\alpha_{out} = \alpha_{ray} + \alpha_{sample} * (1 - \alpha_{ray}).$$

The result is an approximation of an integral of light intensity along the ray by means of a Riemann sum. The complicated maths are not given here, but knowing that the computation is indeed an integration is useful when implementing optimizations. A more detailed mathematical approach to DVR is given in [Levoy, 1988] and [Chen et al., 2003]. The sampling distance, which is often also referred to as the step size, is important when accumulating colors, as it represents the length of the interval at which the color and opacity are assumed to be constant in the approximation. Having a higher sampling rate, corresponding to a smaller sampling distance, thus leads to a better approximation of the integral which in turn leads to a better image quality, at the cost of a more expensive computation.

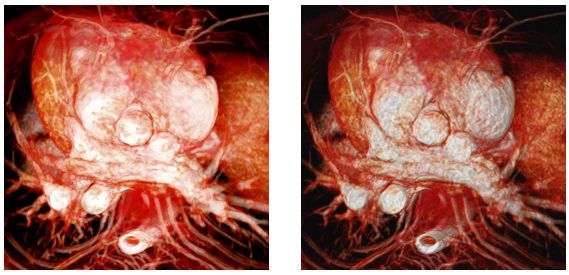
Since DVR is a computationally intensive rendering method, many optimizations and variations are implemented. Some of these take advantage of by imposing restrictions on the transfer function. A trivial example is when the transfer function used for the opacity is a step function. In this case DVR is equivalent to isosurface rendering. Another common optimization is early ray termination. The traversal of a ray can be terminated as soon as the accumulated opacity becomes close to one, as any new samples would contribute for a negligible amount to the accumulated pixel color.

Apart from directly implementing the ray casting method described above, other methods also exist. There are variations of the ray casting method, such as pre-integrated volume rendering [Eric

B. Lum, 2004]. Among the methods that take a completely different approach is the shear-warp method [Lacroute and Levoy, 1994]. Because DVR is such a computationally expensive visualization method, much research has been done in performing DVR on graphics hardware. This includes both hardware specifically designed for DVR [Pfister, 1999] and using consumer-level programmable graphics cards [Kniss et al., 2001].

1.3 Shaded Direct Volume Rendering

To enhance the ability to recognize the spatial structure of the data visualized, shading can be added to the DVR method. Before accumulation of each sample, the sample is lit using a lighting model of choice. For this, the surface normal needs to be reconstructed at each sample. A good and cheap way of computing this vector is by considering the rendered surface to be a special kind of isosurface, giving that the normal vector is equivalent to the normalized gradient of the data. Popular lighting models are the ones developed by Phong and Blinn [Phong, 1975], [Blinn, 1977]. Adding lighting information to the rendered image greatly increases the amount of information on the spatial structure and orientation of the volume data embedded in the rendering, especially when the rendering can be interactively manipulated. This can lead to a better understanding of the data. To see what shading adds to standard DVR, figure 1.3 shows a volume dataset rendered using DVR with and without shading.



(a) (b)

Figure 1.3: A volume rendered using DVR without (a) and with (b) shading applied.

1.4 Constructive Solid Geometry

Constructive Solid Geometry (CSG) is a method of constructing complex objects by combining several simpler objects using a set of boolean operations. These operations include the union, intersection, difference and conjugation. For example, when the union of two objects A and B is computed, each point in the resulting object is both in A and in B, thus $(\forall x : x \in \mathcal{R}^n : x \in A \cup B \Rightarrow x \in A \wedge x \in B)$. Similar relations can be constructed for the intersection and difference operators. The union of an object and the conjugate of another object is equal to the difference of the two objects. An example of a CSG operation is shown in figure 1.4.

The objects involved in the CSG operations can be described as isosurfaces, polygon meshes, or any other representation that allows a classification of a point in space to be either inside or outside the object. When using isosurfaces for example, the application of CSG is very direct. Given two surfaces $f(\overrightarrow{x}) = c_0$ and $g(\overrightarrow{x}) = c_1$, the intersection of the two would be the surface where both of these equations holds.

One of the strengths of CSG is that the input of a CSG operation can be the output of a previous operation. This allows the construction of very complex models using only simple primitives. This is why this concept is very popular in the modeling business.

1.5 Combining DVR with CSG

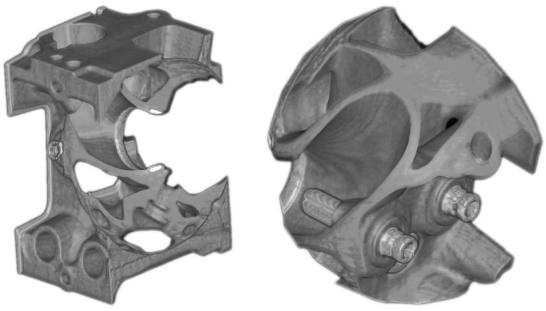
An interesting feature of DVR is that the transfer function allows one to map different voxel intensity ranges to different colors and opacities. However, areas of the volume that have the same intensity cannot be separated. This is a problem if some part of the volume needs to be isolated from other part that has equal intensity. In medical imaging for example, blood vessels and bone have equal intensity in a CT scan. Among others, this is a reason to combine CSG operations with DVR. This combination is often referred to as *volume clipping* [Weiskopf, 2003] or rendering *attributed volume data* [Tiede et al., 1998]. This allows only part of the volume to be rendered, effectively cutting out a piece of the original data. The interest is primarily in the visualization of the volume data, so often the object used to clip the volume, which is called the clipping volume, is not or partially rendered. This means that the most common CSG operation is the difference operation.

In order to perform a valid CSG operation between a volume data set and a clipping volume, the clipping volume should meet a number of requirements. The surface described by the clipping volume should be closed and not contain any self-intersections. If these requirements are not met, the inside of the object is not well-defined and there are no CSG operations possible. Given a well-defined clipping volume, there are two possible ways of using it to render only part of the volume; either the part of the volume that is inside or the part that is outside the clipping volume can be rendered. The former is referred to as *volume probing* and the latter is called *volume cutting*. Figure 1.5 gives an example of these two concepts.

Most current volume renderers that support volume clipping, support only the use of clipping planes instead of arbitrarily definable clipping volumes. Using clipping planes severely limits the possible clipping geometry. Although some convex objects can be approximated using multiple clipping planes, most implementations are built with the use of only very few clipping planes in mind. This renders them not suitable for the approximation of curved surfaces. Using concave surfaces or performing volume cutting (even with convex objects like a cube) is impossible with support for only clipping planes. Thus there is a need for arbitrarily definable clipping volumes.

Volume renderers that do support more complex clipping geometry often offer volume clipping in the form of a binary volume or an isosurface to act as a masking volume. This method has several downsides. Since the original volume data usually requires a relatively large amount of memory (in the order of one gigabyte), using a second dataset of similar size is unacceptable. Binary volumes offer a

Figure 1.4: An object constructed using CSG as the difference between a cube and a sphere.



(a) (b)

Figure 1.5: Examples of volume cutting (a) and volume probing (b) applied to a volume data set of an engine.

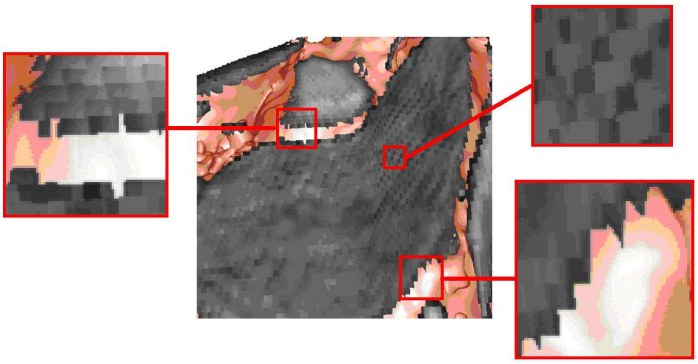


Figure 1.6: Artifacts caused by volume clipping and volume shading.

solution to this problem in that they reduce the amount of memory required. However, binary volumes also give rise to aliasing artifacts, which require interpolation functions to be removed. Isosurfaces defined by a volume dataset usually do not suffer from aliasing artifacts, but require considerably more memory than binary volumes. These additional memory requirements are unacceptable for most practical cases. Thus there is a need to store the clipping volume in a memory-inexpensive way while avoiding both aliasing artifacts and the need for interpolation.

Another problem with most implementations of volume clipping is that artifacts appear near the boundary of the clipping volume at areas in the volume where the differences in intensity of neighboring voxels are small, especially when volume shading is applied. The artifacts are caused by a not well-behaving gradient in those areas of the volume. This means that because the gradient of the data in those areas of the volume is close to zero, the variations caused by noise in the data suddenly have a large effect on the gradient, causing the *normalized* gradient to show non-continuous behavior. The gradient is used in the illumination terms, where the noisy behavior becomes visible. Other artifacts that may appear are interpolation artifacts, for example when the clipping is determined on a per-voxel basis, causing jagged edges of the clipping volume boundary. An example of both noise and interpolation artifacts is given in figure 1.6.

1.6 Requirements

To overcome these problems with current volume clipping implementations, a new volume renderer was engineered. A list of requirements that his volume renderer was designed to meet is given below. The problems mentioned above are contained within this list.

R0 *The clipping volume does not require much memory as storage.* Because volume datasets are often large, other data should be as memory-conservative as possible.

R1 *The resolution of the clipping volume is independent of that of the volume data.* To allow smooth curved surfaces to be used, a high resolution for the clipping volume is required. To prevent aliasing artifacts, it should be possible for the clipping volume to be specified with sub-voxel accuracy.

R2 *The clipping volume is stored in a general form.* For the volume renderer to be applicable in many different cases, a general form should be chosen to store the clipping volume. This allows it to be specified by many different representations, making the renderer more flexible.

R3 *The clipping volume can be transformed independently from the volume.* Being able to translate, rotate and scale the clipping volume independently from the volume greatly increases both the interactivity with the visualization and ability to apprehend the spatial structure of the clipped volume.

R4 *The artifacts caused by not well-behaving gradients near the edges of the clipping volume are minimal.* The presence of noise in the gradients not only decreases the overall beauty of the visualization, it also destroys the ability to use the lighting to perceive the spatial structure of the volume. Therefor these artifacts should be minimized.

R5 *The overhead caused by the use of a clipping volume is minimal.* Volume rendering without clipping is already a computationally expensive rendering method. To keep the interactivity as high as possible, the addition of volume clipping should cause minimal overhead.

R6 *The volume renderer is built on top of the Volume Visualization Component (VVC) library.* The VVC is a library designed by Philips that provides an architectural framework suitable for volume rendering. It offers functionally for constructing a scene to applications that wish to perform volume rendering. The VVC dispatches the actual rendering of the scenes constructed by an application to one of the *drivers*, a software module that is capable of performing the actual rendering of a scene from the VVC. By implementing the volume renderer as a driver for the VVC, other software components that use the VVC can use the new driver with only few modifications.

Requirements **R0**, **R1**, **R2** and **R3** were met by storing the clipping volume as a polygon mesh. A polygon mesh does not require much memory to store, they can have a resolution that is independent of the resolution of the volume data and they provide a general and flexible way of storing geometry. Other forms of specifying a surface such as an isosurface or a set of Bézier patches can all be converted to polygon meshes. By transforming the vertices of a polygon mesh with a transformation matrix prior to processing, the clipping volume is effectively transformed independently of the volume.

Meeting requirement **R4** was achieved by using the gradient of the clipping volume near the edges of the clipping volume. In the case that the gradient of the volume is not well-behaving in those areas, the possible noise is suppressed by using the normal vector of the clipping volume. This relies on the

assumption that the clipping volume is smooth. This approach has the additional advantage that the spatial structure of the clipping volume is visualized more distinct.

Another advantage of using polygon meshes as a representation of the clipping volume is that they can be rendered very fast by current graphics hardware, readily available in consumer-level PCs. This feature was exploited to meet requirement **R5**, while still offering a high-resolution clipping volume.

Apart from the volume renderer, implemented as a VVC driver (**R6**), a stand-alone application that demonstrates the various possibilities of the driver and the techniques implemented, allowing interaction with the scene, was also made.

1.7 Report Layout

A more detailed discussion of what has been done before in the area of volume clipping is given in chapter 2. In this chapter, possible volume clipping techniques are evaluated, including volume based methods and depth based methods. A technique that is suitable for performing volume clipping with polygon meshes, a depth based method, is discussed in more detail in chapter 3, in which two possible approaches are evaluated. Chapter 4 deals with the integration of these clipping techniques with two volume rendering engines. The first one is a software based component provided by Philips, the second engine uses modern graphics hardware and was developed as part of this project. Some minor implementation details are listed in chapter 5, mainly the modifications made to the VVC and the description of the driver that was developed. The results and a comparison of the different techniques is given in the conclusion in chapter 6.

Chapter 2
Volume Clipping Techniques

In this chapter an overview is given of some of the research that has been done in the areas of volume rendering and volume clipping. As discussed in chapter 1, the project focuses on the part of volume clipping. Two popular methods of volume clipping are explored, giving a listing of their advantages and disadvantages. Out of these two techniques, the second is explored in greater detail and a starting lead is given for chapter 3.

2.1 Literature Research in Volume Clipping

Many research has been done in the area of volume rendering; literature offers a wide variety of methods for DVR such as the shear-warp method [Lacroute and Levoy, 1994] or methods based on ray-casting [Levoy, 1988] and many variations of these two. For the past few years also a lot of research has been done on the implementation of various volume rendering algorithms on modern programmable graphics hardware [Roettger et al., 2003], [Pfister, 1999], [Pfister et al., 1999], [Kaufman et al., 2000], [Engel et al., 2001a]. Most of these sources that perform volume rendering on hardware use methods that are significantly different from the ray casting based rendering methods. Most of the prominent hardware-focused algorithms render the volume as a set of two-dimensional slices perpendicular to the viewing direction.

As mentioned in [Weiskopf, 2003], the research in the area of volume clipping is relatively limited and there is much left to explore. Although there exist articles that cover the area, virtually all of these are occupied with using clipping planes [Chen et al., 2003] instead of arbitrarily defined clipping geometries. This is likewise true for most existing implementations of volume clipping. Even the current VVC to be used in this project has an architecture that allows a user to define only clipping planes instead of arbitrary clipping volumes.

One of the very few to investigate the possibilities of more complicated clipping geometries is Weiskopf et al., who give an overview of suitable rendering methods for volume clipping with arbitrary clipping volumes in [Weiskopf, 2003]. This article is an update of the earlier article [Weiskopf et al., 2002]. The prime focus of this article is the use of modern graphics hardware to achieve volume rendering with interactively changeable arbitrarily defined clipping volumes. Adding an illumination model to the presented rendering methods is also covered. With the exception of the strict focus to perform the volume rendering on programmable graphics hardware, most elements of that work are applied in this project.

Most methods for volume clipping can be assigned to one of two classes; the class of methods that take volume-based approach and methods that take a depth-based approach. The same classification is

made in the work of Weiskopf et al. [Weiskopf, 2003]. The next two sections explore these two classes in greater detail.

2.2 Volume-Based Clipping Approach

The methods that take a volume-based approach at volume clipping construct a second volume dataset of the clipping volume (often referred to as the 'voxelized clipping volume') and use it while performing the volume rendering of the original volume dataset to compute the (in)visibility of the voxels. The clipping volume can be represented by any three-dimensional scalar field defining an isosurface. A disadvantage of this method is that in order to make the memory requirements acceptable, the second volume dataset is to be defined as a binary volume. In this case each voxel is represented by a single bit indicating that the voxel either belongs or does not belong to the clipping volume.

A problem with using a second volume dataset is that slightly changing the clipping volume requires updating the related volume dataset as well. Even when only the orientation of the clipping volume with respect to the volume dataset is changed an update is required. This is usually a time consuming process, thereby reducing the degree of interactivity. Also the use of binary volumes is not available on current programmable graphics hardware, as these are optimized for floating point operations and offer no instructions for bit operations.

Weiskopf mentions another problem in [Weiskopf, 2003] with using binary volumes; the gradient information available from the clipping volume is very limited, causing the surface to look very jagged when volume shading is applied. A solution is to store or compute the Euclidean distance to the clipping volume. This makes trilinear interpolation during sampling possible and a gradient in distance gives a good approximation of the gradient of the clipping volume at that location. Storing the distance is however unacceptable from a memory-consumption point of view, while computing the distance during rendering is computationally expensive.

Specifying the clipping volume as a volume dataset offers relatively few restrictions to the complexity of the clipping volume. Moreover, there are no ambiguities that may appear with surfaces, such as self-intersections or gaps. However, the resolution of the clipping volume is limited to the resolution of the volume data. The aliasing effects that may appear due to a low resolution can be somewhat reduced by interpolation, but most simple (and fast) filters don't perform well for generic volumes. Linear interpolation cuts off corners of hard-edged surfaces, while nearest neighbor sampling causes jagged edges on rounded surfaces.

The numerous disadvantages mentioned above renders this method not suitable for this project. Currently the volume data may well be over one gigabyte. Constructing a second volume dataset would enforce the use of binary volumes, while the need for a high image quality would in turn enforce the use of interpolation to smooth out the jagged edges. The decreased interactivity due to modification restrictions adds to the reasons to reject this method. Also one of the goals of the project is to find an alternative to current methods that take this approach.

2.3 Depth-Based Clipping Approach

A depth-based approach exploits the depth structure of the clipping volume to compute the depth-segments which should be rendered for each pixel. This method works particularly well with ray

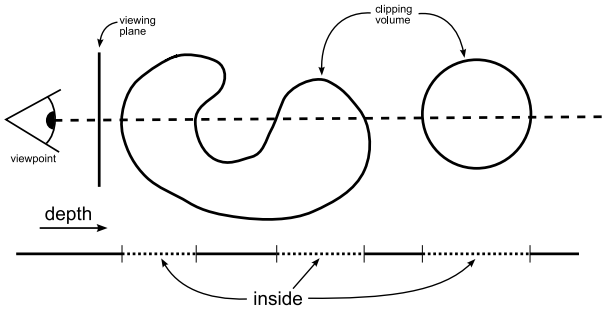


Figure 2.1: Intersection segments; intervals on the depth axis that are inside the clipping volume.

casting-based volume renderers. For every pixel of the viewing plane, there is a corresponding ray to be used by a ray casting-based volume renderer. Finding the intervals along this ray that are on the inside of the clipping volume provides enough information to render only those parts that should not be clipped; when these intervals are known, volume rendering is performed only on those intervals of the ray. This reduces problem of volume clipping to finding intersections of the clipping volume along a ray and then applying a modified volume rendering technique. This process is depicted in figure 2.1. The intervals along the ray that should not be clipped are called *intersection segments* of the ray.

A well defined depth-structure is required for the depth-based approach to work. This means that there are a number of restrictions to the clipping volume. The clipping volume should be representable by a closed and non-self-intersecting surface. Note that this was one of the assumptions on the clipping volume made in chapter 1. The volume-based approach implicitly makes a similar assumption, since it is impossible to represent a self-intersecting or open surface by a volume dataset. Apart from this restriction however, there are no strict limitations on how the clipping volume itself is defined. Binary volumes, isosurfaces and polygon meshes are all allowable formats. Therefore a suitable format can be chosen to meet requirements **R0** and **R2**.

The biggest difference between the depth-based and volume-based approaches is that there is no direct relation between the volume dataset and the clipping volume. This allow for interactively changing the clipping volume independently from the volume, meeting requirement **R3**. More important, sub-voxel accuracy is achieved without the need for interpolation schemes, since the resolution of the clipping volume is independent of that of the volume, meeting requirement **R1**. In fact, due to the nature of the approach, the intersections are always computed with per-pixel accuracy. The sub-voxel accuracy prevents various aliasing effects mentioned earlier, which in turn provides smoother gradients. Information required for proper shading, such as the surface normal vector, is also often readily available, making it possible to meet requirement **R4**.

The depth based approach is a *geometry dependent* approach, opposed to the volume based approach which is geometry independent. The term geometry here refers to the orientation of the clipping volume, the volume dataset, the camera and the projection settings; basically all settings that control the projection to the viewing plane. This is an important difference, as this means that with the depth-based approach the clipping volume may have its own geometry that is independent of the rest of the geometry. This results in great interactivity, as the clipping volume can be rotated, translated and scaled without any additional costs. Such degrees of interactivity are in general not available for geometry independent approaches. This property also helps meet requirement **R3**.

Since the depth-based approach has many advantages over the volume-based approach, the latter

method is not further explored. Instead, a more detailed look into the depth-based approaches is taken, exploring various algorithms and different optimizations thereof. This decision was made based upon the information available in literature and the large number of disadvantages of the volume-based approaches.

2.3.1 Ray Casting Method

Although Weiskopf solely mentions the use of graphics hardware to implement the depth-based approach of finding intersections, there are a different solutions. Figure 2.1 explains volume clipping for ray casting based volume rendering and in fact the depth structure can also be computed by directly applying ray casting. To do this, a ray is casted for each pixel of the viewing plane in the same direction as will be done for the volume rendering. Assuming that the clipping volume is defined as a triangular mesh, the set of triangles that the ray intersects is computed for each ray that is casted. Using this set of triangles, a sorted list of depth values and gradients can easily be constructed, yielding the ray segments on which volume rendering should be applied.

Ray casting has a reputation of being relatively slow, but for this particular purpose only the intersections with the clipping volume need to be computed, so no intricate lighting or shadow computations are required. On the other hand all intersections are required, while for ordinary ray casting applications the ray traversal is terminated after the first (often closest) hit. This makes it difficult to estimate the performance in terms of speed compared to traditional ray casting. As this approach is interesting to compare with a rasterization based one, the idea of using ray casting to compute the ray segments is explored in more detail in section 3.1.

2.3.2 Rasterization Method

An alternative use of the depth structure of the clipping volume is to use a rasterization technique to project the clipping volume onto the viewing plane. If the clipping volume is represented by a polygonal mesh, this means that each triangle (or polygon) is projected on the viewing plane and the intersection data (such as depth and gradient vector) are interpolated along the pixels covered by the polygon. This approach is particularly interesting since current graphics hardware is specialized in performing rasterization. Modern graphics hardware even offers a programmable pipeline along that process, yielding a high flexibility and a wide range of possibilities. The high performance offered by the use of graphics hardware should help to meet requirement **R5**.

There is a problem however on how to store the data. Often there is only a single plane available to store the data, especially when the approach is implemented on hardware. Storing only a single depth value for the clipping volume would make the resulting image significantly deviate from the desired result. Weiskopf offers some solutions in [Weiskopf, 2003] such as front- and back-face culling and the use of depth-peeling to combine multiple rendering passes to find multiple layers of intersections. These solutions, among others, are discussed in more detail in section 3.2 where this method is treated in more detail.

2.3.3 Comparing Ray Casting to Rasterization

The ray casting approach differs a lot from the the rasterization approach. While the ray casting approach maps the pixels of the viewing plane to the triangles of the clipping volume, the rasterization approach does the opposite. The two approaches offered above also differ a lot in complexity. It is common for ray casters to be well scalable in the number of polygons, while increasing the dimensions of the viewing plane usually decreases performance. For the rasterization method the opposite is commonly true. Increasing the number of polygons means that more polygons need to be rasterized, while the area of the viewing plane usually is of less importance. This difference in complexity makes it interesting to compare the two methods.

2.4 Combining Volume Clipping with Shaded DVR

As mentioned in chapter 1, the use of volume shading in combination with volume clipping may cause artifacts in areas where the gradient of the volume is not well-behaving. The reason is that the gradient in such areas cuts through a part of the volume data where there is little variance in the values of neighboring voxels. The obvious solution is to use the gradient of the clipping volume in these areas. However, if the gradient is used only at the intersection points, artifacts may remain.

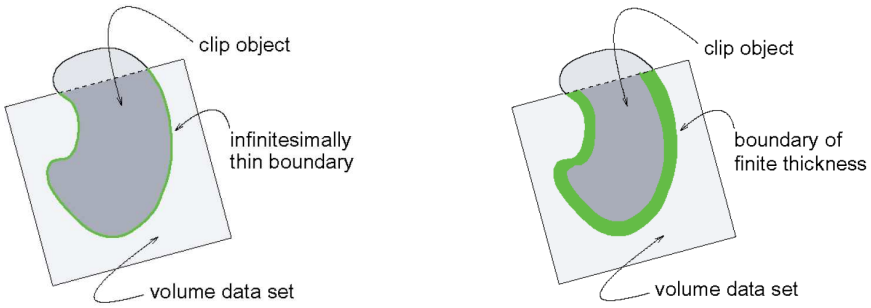
To explain this in more detail, some more theoretical background is required. Weiskopf mentions four criteria that should be met in order to achieve successful volume shading in combination with clipping in [Weiskopf, 2003]. The first is that in the vicinity of the clipping volume, the shading should allow the viewer to perceive the shape of the surface. Second, the different optical models used for the volume and the clipping volume should be compatible and not cause any discontinuities. Third, areas of the volume not in the vicinity of the clipping volume should not be affected by any different optical models used for areas that are. Finally, the volume clipping should be independent of the sampling rate used for the volume rendering.

The solution mentioned above meets the first three criteria, but not the fourth. The intersections of the rays with the clipping volume define an infinitely thin surface. Thus if the gradient is only used at the exact point of intersection, it is used in only a single sample during the volume rendering. Since the amount a sample contributes to the final pixel color is dependent on the sampling rate, the fourth criteria is not met. A solution offered by Weiskopf et al. in the same article is to “impregnate” the gradient of the clipping volume along a layer of finite thickness into the volume data. This is shown in figure 2.2, which originates from [Weiskopf, 2003].

With this solution, the gradient of the clipping volume is used for multiple samples during the volume rendering, and that number of samples depends on the sampling rate. The actual length of the segment of the ray where the modified gradient is used is constant throughout the entire image and is a parameter of the visualization; it defines the thickness of the impregnation layer. The gradient used by the samples that are inside the layer should be computed using

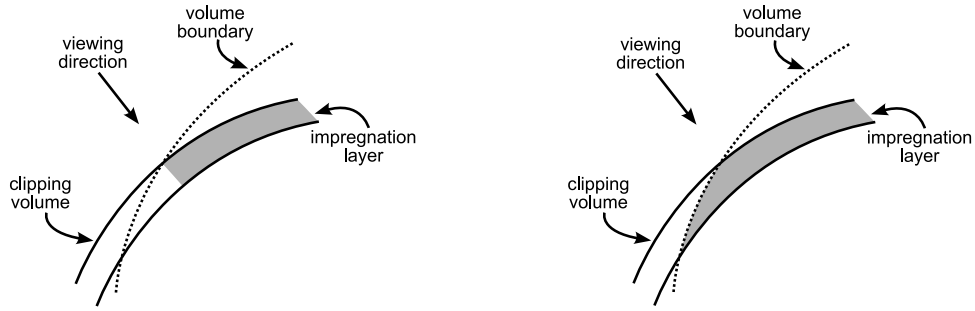
$$g(\overrightarrow{x}) = w(\overrightarrow{x})S_{clip}(\overrightarrow{x}) + (1 - w(\overrightarrow{x}))S_{vol}(\overrightarrow{x}),$$

where $S_{clip}(\overrightarrow{x})$ and $S_{vol}(\overrightarrow{x})$ represent the gradient at location $\overrightarrow{x}$ of the clipping volume and volume dataset respectively and $w(\overrightarrow{x})$ defines a weighting function $w : \mathbb{R}^3 \rightarrow \mathbb{R}_{[0,1]}$.



(a) (b)

Figure 2.2: Using the gradient of the clipping volume on an infinitely thin layer (a), or a thick boundary (b) of the volume data.



incorrect correct

Figure 2.3: If the gradient impregnation algorithm is only applied after an intersection with the clipping volume, sudden changes in the optical model may appear.

If the voxel values at the clipping edge are mapped to a low opacity, the voxels beyond the edge may give a significant contribution to the pixel color. If the weighting function is defined as a step function, the sudden change of the optical model near the end of the impregnation layer may become visible. To prevent these discontinuities from being visible, a linear function can be used.

Note that if the thickness of the impregnation layer is too small, the contribution of the samples that use the gradient of the clipping volume may become insignificant and the noise caused by not well-behaving gradients of the volume may still be visible. This is of course not true for isosurface rendering, since there are no transparent areas in that case, so each pixel in the viewing plane is made up of at most one sample. If the layer thickness is too large on the other hand, the gradient information of the volume is lost and the spatial structure of the volume becomes harder to perceive.

It is vital that the gradient impregnation is applied in all areas of the volume that are near the edge of the clipping volume, even if there is no intersection with that clipping volume. Figure 2.3 shows the problem that may arise if this is not done. The left figure shows a sharp edge where the gradient impregnation layer ends. In the final image, this often corresponds to a steep change in color intensity, as there is a steep change in the average gradient. By using gradient impregnation throughout the entire volume, the situation of the right of figure 2.3 is achieved. In this case the effect of the gradient impregnation layer is smoothly decreased, yielding a smooth transition in edge intensity.

The solution of gradient impregnation works well in combination with the volume-based approaches where the Euclidean distance to the volume is known, as this distance can be used during the volume rendering to determine whether the gradient of the volume data or the clipping volume should be used. For the depth-based methods however, this information is not available.

A solution for the depth-based method is to use the distance to the clipping volume *in the viewing direction*, which corresponds to the difference between the depth of the last intersection with the clipping volume and the depth of the sample where the gradient is required. This may give mathematically incorrect results near an edge of the clipping volume that is in line with the viewing direction, since the gradient may not reflect the correctly interpolated gradient. Fortunately any artifacts that may be caused by this limitation are not noticeable.

As an optimization, it is sufficient to apply the above technique only for edges of the clipping volume that are facing the viewer. Edges that are not facing the viewer have a number of samples that are used for volume rendering in front of them. If the ray is not already terminated by early ray termination, any noise that may originate from not well-behaving gradients is usually hardly visible. Moreover, these backward facing edges are not used to perceive the spatial structure and orientation of the volume and clipping volume, so mathematically correct behavior is a less strict requirement in these areas as well.

Chapter 3
Depth-Based Volume Clipping Techniques

Due to the large number of advantages over volume-based techniques, the depth-based approach for performing volume clipping is further examined. In this chapter two depth-based volume clipping techniques are explored in detail. In the first section the ray casting method is examined. The rasterization method is explored in the second section. Finally the two methods are compared. Note that during the project this comparison was done at a stage where the rasterization method was only roughly examined. At that time tests indicated that the rasterization method was, at least for practical cases, many times faster than the ray casting method. That is the reason that research was directed at the rasterization method. Therefore the section on the rasterization method is much more elaborate than that of the ray casting method.

3.1 Ray Casting Method

There are many volume rendering techniques that are based on ray casting. Therefore, using a ray casting technique to perform volume clipping seems natural. Ray casting has a relation with ray tracing, which is known to produce good results with respect to image quality, but being relatively slow in doing so. It is however not fair to make this comparison directly, since there are a number of differences to classical ray tracing applications. If ray casting is used only to find the intersection point with a polygonal mesh, there is no need for lighting or shadow computations, which would lead to a better performance. On the other hand, not only the first intersection, the one closest to the viewer, is needed, but possibly all intersections along the ray. This leads to a decreased performance in turn, so it is hard to make clear estimations of the performance.

A clear advantage of the ray casting approach is its simplicity. Writing a brute-force ray caster can be done in a very short time, using few lines of code. From that point on, there are several optimizations possible to increase the rendering speed.

In the rest of this section the basics of ray casting are discussed first, after which various approaches taken to improve the speed of the ray casting are discussed. At the end of the section is a short conclusion with the advantages and disadvantages of the ray casting approach.

3.1.1 Using Ray Casting for Volume Clipping

Since the basic concept of ray casting for use in volume rendering has been explained in chapter 1, only a short introduction is given with the elements specific for finding intersections. As with most forms of

ray casting, a ray is casted for every pixel of the viewing plane in the viewing direction. For each ray, the objects in the scene are tested for intersection with that ray. If an object, which could be any type of object, intersects with the ray, the intersection depth and the gradient at the intersection point are computed and stored for the pixel the ray belongs to. A difference with most ray casting applications is that for this particular purpose *all* intersections of the ray with the objects in the scene should be stored, opposed to just the one closest to the viewing plane, as is common practice in most ray casting applications.

This method of volume clipping integrates very well with ray casting-based volume rendering techniques. The process of finding intersections along a ray could be performed in parallel with the traversal of the same ray used for the volume rendering. This would make the volume clipping benefit from early ray termination, preventing the computation of intersections that are not used during the volume rendering.

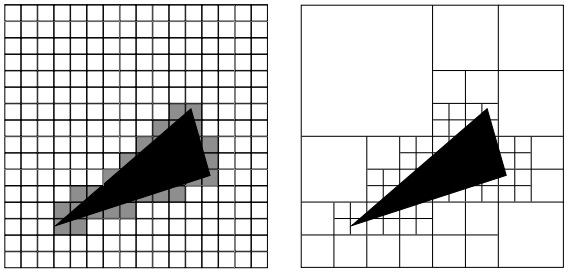
A brute-force implementation of ray casting has a very bad performance. To increase this performance, there exist many optimization techniques. The next few subsections describe some common optimizations that were implemented. Section 3.1.2 deals with high-level optimizations, while section 3.1.3 concentrates on low-level optimizations.

3.1.2 Spatial Subdivision

An important feature that many ray casting implementations share is a data-structure to arrange *spatial subdivision*. This term refers to mapping the objects to be tested for intersection to certain parts of the virtual space that is to be traversed by each of the rays. It should then allow an efficient construction of a list of objects that are close to a given point in that space. This removes the need to test all objects for intersection with a particular ray, since only those objects that are close to the ray need to be tested. This greatly decreases the number of intersection computations that need to be performed, thus increasing performance. For virtually all ray casting and ray tracing renderers, the spatial subdivision schema is the most important element of the speed enhancing techniques. There are many different spatial subdivision algorithms, among them are the uniform grid, the octree and the kd-tree. An overview of the most popular algorithms is given in [Havran, 2000], but many more exist, such as the ones described in [Arvo and Kirk, 1987].

Most popular ray tracing packages use kd-trees as their spatial subdivision schema. It has a reputation as being very fast and suitable for ray tracing, which is justified by literature such as [Havran, 2000]. Except for extremely dense scenes, the kd-tree performs best. However, for this project the scenes differ a bit from the general scenes frequently occurring in ray tracing. General ray tracing scenes are usually very sparse, where a highly detailed object covers a small part of the screen, while other parts of the screen remain relatively empty. When comparing spatial subdivision algorithms, this problem is referred to as the “bunny in stadium” problem, which will be explained in more detail in section 3.1.2.

Although literature recommends using kd-trees [Havran, 2000], the choice here is made for octrees. First of all, octrees are the next best thing to kd-trees in terms of performance, but the most important argument originates from the possibilities with volume rendering. The volume data set is organized as a grid-like structure. Often volume renderers contain a volume-based optimization that allows them to discard entire blocks of voxels at a time based on certain criteria. If the spatial subdivision scheme used for the volume clipping also contains these blocks, the volume renderer could also discard a block of voxels if it is entirely outside the clipping volume. These possible optimizations lead to the choice of grid-like spatial subdivision schema’s like the uniform grid and the octree.



(a) (b)

Figure 3.1: A triangle located in a two dimensional spatial subdivision structure. The left image (a) shows a uniform grid, the right image (b) a quadtree, which is the two dimensional equivalent of an octree.

Both the uniform grid and the octree are described here. Only a short summary of the algorithms is given, together with the most important features. More detailed description of both the methods, possible variations and construction and traversal algorithms are all available in literature.

Uniform Grid

Before implementing an octree as a spatial subdivision scheme, the uniform grid is explored first. As will become clear later on, the uniform grid can be considered as a special case of an octree. According to literature, the uniform grid performs very bad in general, only on very dense scenes it outperforms other algorithms. It is the simplest form of spatial subdivision.

A uniform grid is, as its name implies, a uniform grid of cells. Each cell contains a list of objects that intersect with that cell. Given a point in space, the corresponding cell of the uniform grid can be computed in $\mathcal{O}(1)$ time, due to the uniformity of the grid. If the cells are small enough, evaluating only those cells of the uniform grid that intersect with the ray being evaluated leads to examining only those objects that are close to the ray. An example of a two dimensional uniform grid is shown in figure 3.1 (a).

The main problem with the uniform grid is commonly demonstrated with the “bunny in stadium” problem. Suppose the scene consists of a huge stadium with a very small yet extremely detailed bunny at the center (the bunny refers to the famous Stanford Bunny, a polygon model commonly used as a demonstration model in research). Because the bunny is so very detailed, the size of the cells of the uniform grid should be small as well to gain a good spatial subdivision. But because the grid is uniform, there are a lot of cells in the huge open space of the stadium. This not only wastes memory, but it is the prime reason for the bad performance, as all these empty cells have to be evaluated when a ray passes through the uniform grid.

Once the uniform grid has been constructed, it needs to be traversed by a ray. This means that for a given ray, all the cells of the uniform grid that intersect with the ray should be examined, testing the objects that intersect with the cell for intersection with the ray. An efficient way to determine these cells is by traversing the grid along the ray using a three dimensional digital differential analyzer algorithm, or 3D-DDA for short, which is described in [Amanatides and Woo, 1987]. The algorithm contains no recursion and uses mainly additions for its computations, giving a good performance.

Octree

Octrees extend the uniform grid by imposing a tree-like structure on top of the uniform grid. Figure 3.1 (b) gives an example of a quadtree, which is the two dimensional equivalent of an octree. At the root of the octree is a single cell which is the bounding volume of all the objects. When constructing the octree, a recursive process is started. Each cell is checked if there is any object inside. If the cell is not empty, eight (or four for the quadtree) child nodes are created, each covering a different part of the current cell. The recursion is stopped when an empty cell is encountered or a predefined depth of the tree is reached.

The advantage of the octree is clear in figure 3.1 (b), large empty regions are left empty, so they can be easily skipped during traversal. However, the traversal algorithm is slightly more complicated. Given a point in space, finding the corresponding cell takes $\mathcal{O}(\log n)$ time, where n is the granularity of the grid, compared to $\mathcal{O}(1)$ for the uniform grid. There is an extension of the 3D-DDA algorithm for octrees [Sung, 1991], but it performs bad for unbalanced trees [Revelles et al.,]. Skipping the empty regions is exactly the reason to use an octree instead of a uniform grid, so this is a bad idea. A much better algorithm is presented in [Revelles et al.,], which was the traversal algorithm of choice.

3.1.3 Computing Intersections

The spatial subdivision data-structure only gives a rough approximation of which objects are near a ray and which not. Once an object passes the spatial subdivision test, the precise test whether it intersects with the ray and at what depth values still needs to be computed. Since the clipping volume is defined as a triangular mesh, there is the need for a ray-triangle intersection algorithm. Even though the spatial subdivision seriously decreases the amount of tests needed for a rendering, the actual ray-object intersection algorithm is usually still executed many times. Using an efficient algorithm is thus required to achieve high performance. In this section a comparison is made between various ray-triangle intersection algorithms. Finding an efficient algorithm for this test is only a local optimization (an $\mathcal{O}(1)$) to the problem. The spatial subdivision data-structure has a much larger effect on the overall performance.

A few popular algorithms were studied and implemented and a small benchmark was run to determine the fastest algorithm. Much work has been done in this area, also on comparing various ray-triangle intersection algorithms, such as [Segura and Feito, 2001]. A few popular and frequently used algorithms were implemented, being Möller-Trumbore [Möller and Trumbore, 1997] and a few optimizations thereof described in [Möller, 2000], Badouel [Badouel, 1990] and an algorithm described by Dan Sunday [Sunday, 2001]. Other algorithms such as using plücker coordinates were not investigated, as no high-performance ray casting or ray tracing applications are known that use these algorithms.

Besides a comparative study, [Segura and Feito, 2001] also contains a new algorithm that is claimed to be faster than both Möller-Trumbore and Badouel. However, no pre-computations were made for the algorithms when comparing the algorithms. Combined with the fact that no pre-computations are possible for the algorithm presented by Segura, the performance gain would be insignificant or perhaps completely lost. A much stronger argument not to use this algorithm is that the result of the algorithm is solely a boolean indicating if the ray intersects with the triangle or not, so no depth information is returned (opposed to the other algorithms mentioned). Since the depth information is needed in case of a hit, this would have to be computed, further degrading the performance of the algorithm. The number of references to the algorithm is also very low, indicating that it is not very popular among ray tracers.

Algorithm	Relative Performance
Möller-Trumbore (original)	0.81
Möller-Trumbore (variant 1)	0.91
Möller-Trumbore (variant 2)	1.00
Möller-Trumbore (variant 3)	0.91
Möller-Trumbore (variant 4)	0.87
Dan Sunday	0.87
Badouel	0.58

Table 3.1: Relative performance of various ray-triangle intersection methods.

Most algorithms were presented complete with a C implementation. Since performance is a much higher requirement than memory usage, the algorithms were slightly modified to make maximal use of pre-computation. In all cases these pre-computation steps increased the speed of the algorithm. The modified algorithms were used in a benchmark. The clear winner is the Möller-Trumbore algorithm, which seems to be in line with the conclusions of other people across the Internet. The optimizations mentioned in [Möller, 2000] improved the speed of the algorithm even further. Table 3.1 gives an overview of the performance, measured as the average execution time, of the various algorithms compared to the best variation of the Möller-Trumbore algorithm.

3.1.4 Performance Summary

The previous sections describe a ray casting method with only a few basic optimizations. By using more advanced and case-specific algorithms a higher performance is probably possible. This set of optimizations is sufficient for a quick comparison however. The basic implementation was compared to a number of large fast ray tracers found across the Internet, such as [Federation Against Nature, 2004]. All of these were faster, but not by an extreme amount. This is an indication that if more time was spent on optimization, the performance would probably increase, but not much.

As an indication of the performance, on a Intel Xeon 2.8 GHz machine, a low polygon mesh of 1200 triangles could be ray casted at around 20 frames per second with a viewing plane of 256x256 pixels. When moving to higher polygon meshes (more than 50000 triangles), the frame rate dropped below 5 frames per second. This seems to be in line with most other real-time ray tracers encountered. Most of those proved to be very scalable in the number of triangles, but the performance was usually non-interactive.

An advantage of the ray casting method is that there are interesting optimizations possible for the integration with a ray casting-based volume renderer. One of these is to perform the ray casting in parallel with the volume rendering to avoid the computation of intersections that are not used. There is a problem with this optimization though. During the ray casting of the clipping volume, all computations are performed in object space, while all computations related to volume rendering are performed in voxel space. So either for each intersection test the sample point has to be transformed to object space, or the object data has to be transformed to voxel space as a pre-computation step. The former will not lead to a very good performance while the latter will lose the interactivity offered by the depth-based approach. These are all indications that the ray casting method will not be very fast if more time is spent on it.

3.2 Rasterization Method

A completely different method of computing the intersections with the clipping volume compared to ray casting is to use a rasterization method. For this method, the clipping volume needs to be defined as a triangular mesh. First, the intersection depth and normal vector is computed for every vertex of every triangle of the clipping volume. Next these vertices are projected onto the viewing plane, after which the intersection depth and normal vector are interpolated along each triangle and stored for each pixel of the viewing plane. Compared to the ray casting method, the rasterization method computes which pixels belong to an intersection, while the ray casting method computes which intersections belong to a pixel. Due to the nature of rasterization, no spatial subdivision needs to be performed if all triangles are visible. This is due to the fact that every triangle is mapped to exactly those pixels that are covered by it. Spatial subdivision would only prevent rendering a part of the clipping volume that is not part of the screen, but such cases are rare in practice.

Another difference with the ray casting method is that the computation of the intersections must be performed entirely as a pre-computation step to the volume rendering. For ray casting, the computation of the intersections can be performed in-line with the volume rendering, computing the intersections of a ray just before the volume rendering is started on that particular ray. The rasterization method does not work on a per-ray level. Apart from the loss of possible parallelism with volume rendering, this disadvantage does not have a large impact. The additional memory requirements are insignificant, and there are no serious implications for performance or image quality.

3.2.1 Hardware Acceleration

An interesting aspect of the rasterization method is that rasterization can be accelerated using graphics hardware, as modern consumer-level graphics cards offer support for rasterization. Using graphics hardware to perform the rasterization causes a dramatic performance increase. Many of todays graphics hardware boards are equipped with a Graphics Processing Unit (GPU). This is similar to a Central Processing Unit (CPU), but is is located on the graphics card and is specialized in graphics computations. The benefit of a GPU is that it allows the graphics pipeline to be programmable. This gives room to a wide variety of algorithms while still taking advantage from hardware acceleration.

A major disadvantage of using hardware acceleration is that graphics hardware is aimed at creating a display image, having only a color value for every pixel. There are some auxiliary buffers such as the depth buffer, for storing depth values, and the stencil buffer, for storing integers, but these all work on a per-pixel basis as well. There is no easy way to store a variable-length list of intersections for each pixel, which should be the output of the algorithm. There is a solution to this problem though. Current hardware offers support for floating-point render targets, making it possible to store four floating point values for each pixel with a standard RGBA (Red/Green/Blue/Alpha) render target. Also, the depth buffer essentially contains a transformed version of the intersection depth. Combined with using multiple rendering passes, this gives enough freedom to compute a list of intersections using the common graphics hardware architecture. A more detailed approach on finding workarounds for these limitations is discussed in the next few sections.

Performing these kind of tasks on a GPU is commonly referred to as *general purpose GPU programming*. Over time a community has evolved occupying itself with performing typical stream-based computations on the GPU. One of these communities is available at `http://www.gpgpu.org`. The topics range from image processing to audio and signal processing algorithms.

The Rendering Pipeline

To compute the intersections using a GPU, the clipping volume must be rendered to an off-screen buffer. The contents of this off-screen buffer, which becomes the render target, should be transfered back to host memory and processed to form the intersection data required. However, the tasks to be performed during rasterization differ from the common graphics pipeline. The solution is to use the programmability of the GPU by means of *shaders* [Kessenich et al., 2004]. A shader is a small program that runs on the GPU and is executed for every vertex or every pixel. There are two kinds of shaders, the first one being a *vertex shader*. A vertex shader, or vertex program, takes as input a vertex of the triangle mesh with related properties such as a normal vector, the lighting conditions of the scene, the current geometry matrices, et cetera and produces a transformed vertex as output. That is, a pixel coordinate and a depth of where rasterization of that pixel should be performed on the render target. The second type of shader, a *fragment program* or pixel shader is executed for every pixel along the rasterization of each triangle. The inputs to a fragment program are the interpolated data of the output of the vertex shader and the output is the color of the pixel and optionally data for auxiliary buffers such as the depth buffer or the stencil buffer. Note that instead of a color, any type of data may be written. This is exactly what will be done, as the output of one pass of the intersection computation is a normal vector and a depth value.

When a single depth layer is processed, the clipping volume can be rendered again if multiple intersection segments are required. The more rendering passes are executed, the more depth information becomes available. Rendering more layers of course decreases performance.

A feature of performing the rasterization on the GPU that makes it particularly suitable, is not only that it is highly optimized, but also that it is very advanced. Many advanced features are readily available. These include perspective-correct interpolation during rasterization, clipping operations using auxiliary buffers and many more.

Render Target Formats

The intersection data that needs to be stored consists of two parts; the normal vector of the surface defining the clipping volume and the intersection depth. To define a single intersection segment, a front layer and a back layer are required. As mentioned in section 2.4, the normal vector is required for only one of these layers. Whether it is the front or the back layer depends on the clipping operation; volume probing or volume cutting.

A normal vector consists of three floating point values, while an intersection depth is one floating point value. Without simplification, this would lead to a 128-bit-per-pixel, 32-bit floating point per component RGBA render target format, capable of storing four floating point values using 32 bits for each of the values for one layer and a similar buffer storing only a single floating point value for the other layer. Fortunately, this format of a render target is available for current graphics hardware. The four floating point values of one layer can be stored in the render target, while the depth values of the other layer can be stored in the depth buffer.

If the depth buffer is used to store depth values, these cannot be stored directly. The graphics hardware stores all values in the depth buffer in a specially transformed form. To transform the value from the depth buffer back into a depth of the original scene, more information on the inner workings of depth buffers is required. Since all hardware accelerated programming for this project was done in OpenGL [Segal and Akeley, 2004], the prime reference for this is [OpenGL Architecture ReviewBoard, 1992].

Additional information was found in the section on the depth buffer of the technical Frequently Asked Questions (FAQ) available on the website of the OpenGL library, `http://www.opengl.org`.

There are a number of possible optimizations to reduce the amount of required storage space. A first optimization exploits the fact that the normal vector has unit length. Knowing two components is sufficient, as the third can be computed. This saves one floating point value. There is one caveat with this trick however. Given a three dimensional vector $\overline{v} = (x, y, z)$ with unit length, that is $\sqrt{x^2 + y^2 + z^2} = 1$, storing only two components loses the sign of the third component. Without loss of generality, suppose the z component is discarded. Computing it from the x and y components can be done with $z = \pm\sqrt{1 - x^2 - y^2}$, but which sign to use unknown. A possible solution to this problem is to exploit the fact that both x and y are in the range $[0, 1]$. By adding two to the value of x that is stored if z is negative for example saves the sign information. If x turns out to be greater or equal to two, $z = -\sqrt{1 - (x - 2)^2 - y^2}$ hold, else $z = \sqrt{1 - x^2 - y^2}$ holds. A different simple yet efficient solution is to store the sign information in the depth component. This only works for perspective projection, since the depth may also be negative for orthogonal projection.

A second optimization uses a different precision for storing the intersection data. Besides 32-bit floating point values, the GPU also offers support for 16-bit floating point values. This format can also be used as a format for the render target. If 16-bit floating point values offer enough precision for the normal vector and the depth value a factor two save is made on storage space. This is an interesting optimization as the data of the render target needs to be transfered from video memory to host memory, which is a time-consuming operation. This will be explained in more detail in section 3.2.5. Graphics hardware of course also offers support for 8-bit values, but not in floating point. This format does not offer enough precision for the depth values. The possible uses of different types of render targets is explained in the next sections, which discuss various approaches of computing multiple intersection segments by using multiple rendering passes.

3.2.2 Multiple Layers of Intersections

GPUs are designed for stream-based data processing, having a tight restriction on the way output data is written. Although there is support for using multiple render target simultaneously, allowing multiple values to be written inside a vertex shader, there is no way to access previously written data to the output buffer inside a fragment program. In fact, the only possible interaction between fragments is by means of global variables, the depth buffer and the stencil buffer. The depth buffer is originally designed to cull fragments that fall behind other fragments, allowing only the front-most fragments of a scene to be visible. This is done by writing a depth value for each pixel to the depth buffer. Just before the output of the fragment program is stored in the render target, the depth value is compared with the associated value in the depth buffer. Based on this comparison, the pixel data is either written to the render target or discarded.

Because there is no way to store data from multiple fragments simultaneously into the render target, multiple render passes are required to compute multiple layers of intersections. A *layer* of intersection is a slice of the clipping volume in the depth dimension. All fragments belonging to intersection layer i have exactly i fragments in front of them, so layer 0 is the front-most layer. When combining these layers, the intersection segments can be computed. For closed, non-self-intersecting clipping objects the layers alternate between defining the start and end of an intersection segment.

The GPU is very fast in performing the rasterization. The output is stored in the frame buffer and based upon the approach also the depth buffer. These buffers are stored in the memory of the graphics

hardware. For the output to be interpreted, it needs to be transfered to host memory. This transfer is time-consuming and may well form the bottleneck of the entire method. Optimizations reducing the amount of data that should be transfered are therefor interesting to explore. A more detailed look on the impact the data transfer has on the performance is discussed in section 3.2.5.

An interesting optimization with respect to the size of the output buffer is whether the normal vector is required for every layer. Obviously, the depth information is required for each layer, the impact of the normal vector to the image decreases as the segment is becomes occluded by other segments. Also, as discussed in section 2.4, the normal vector at the end point of a segment also has less impact on the image than the one at the start. Using only the depth value of a layer is an interesting optimization, since this would only require reading the depth buffer, instead of a four-component frame buffer. There may also be ways to combine the data of multiple layers into the frame buffer, as a four-component frame buffer may host one normal vector (two components) and two depth values (one component each). These possibilities are discussed in the next few sections.

3.2.3 Front/Back-face Culling

A simple way to compute only one segment is to use front-face and back-face culling. Current graphics hardware has a notion of the orientation of a triangle. A triangle has a front-face and a back-face. Often these are determined by the order of the vertices in screen space, after projection. If, from a particular viewing point, the vertices are defined in for example clockwise order, the viewer is looking at the front-face of a triangle. Unfortunately there is no standard for the mapping between clockwise and counter-clockwise order of vertices to the front-face and back-face orientation of a triangle; both conventions are used. Making use of front- or back-face culling also is interesting from a performance point of view since the culling is done at a very early point in the graphics pipeline.

This feature can be exploited to compute both the front and back layer of an intersection segment. Sequential layers alternate between defining start and end points of intersection segments. Also, the front layer and back layer always consists of only front-faces and back-faces of the clipping volume, respectively. Combining these two facts give that the the start point of a segment can be computed using only the front-faces of the clipping volume. A similar relation holds for the end point of a segment. A use of this is to first enable back-face culling (causing only front-faces to be rendered) and set the depth compare function such that the front-most faces are rendered to the frame buffer. Using these settings, the clipping volume is rendered using a fragment program that stores the normal vector and intersection depth in the frame buffer. Then the contents of the frame buffer are transfered to host memory to give the first layer with normal vector and depth information. This is depicted in figure 3.2 (a). A second rendering pass is then performed with front-face culling enabled to give the back-faces of the clipping volume. If the depth compare function is set to render the front-most (back-)faces, the first intersection segment can be computed (figure 3.2 (b)). If the depth compare function is set to render the back-most (back-)faces, a single intersection segment that spans the entire clipping volume is computed, as shown in figure 3.2 (c). This last case yields all the intersection segments if the clipping volume is convex. For a concave clipping volume, areas that should be clipped may be used in volume rendering. However, this slight error may not be of great importance (also mentioned in [Weiskopf, 2003]) and it increases performance since only one segment has to be computed. The volume rendering itself is also simplified if there is only a single segment render.

If for the back-faces a normal vector is not desired, one may choose to read the depth buffer instead of the frame buffer. However, since for the data of the front-faces only three out of four color components are used, it is theoretically possible to store the depth of the second rendering pass in the frame buffer

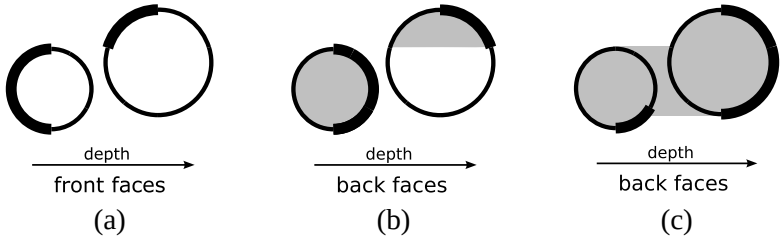


Figure 3.2: Computing a single segment by front- and back-face culling, using different depth compare functions. The gray areas in figure (b) and (c) show the resulting segment that is used in volume rendering.

combined with the data of the first rendering pass. As mentioned before, reading the contents of the frame buffer inside a fragment program is not possible. This means that a single-pass solution using a fragment program that only writes to a subset of the color components based upon the orientation of the face the fragment belongs to is also not possible. A two-pass solution that writes to a single buffer would still be better, as the data transfer of the depth buffer is saved in that case. One way to do that would be by using alpha blending. Alpha blending allows the values of two passes to be added together in a single buffer. If each pass writes zeros to the components it does not use, the results of the two passes are combined into a single frame buffer. Unfortunately, the use of floating-point render targets is required due to accuracy reasons and current graphics hardware does not offer support for alpha blending in combination with floating-point render targets.

A bit less ideal, but still better than two transfers to host memory, is to use a two-pass solution that copies the output in the depth buffer of the first pass to another part of the memory of the graphics hardware. This internally copied data can then be used as an input to the second pass. This transfer of the depth buffer is local to the memory of the graphics hardware and is much faster than a transfer to host memory. For this to work, the back-faces are rendered first, using any of the two depth compare functions. The data written to the frame buffer is of no importance, so the fixed function pipeline may be used instead of a fragment program to maximize performance. After completion of this first pass, the contents of the depth buffer are copied to a texture in the memory of the graphics hardware. In the second pass, the front-faces are rendered using a fragment program that stores the first two components of the normal vector, the depth of the front-face and uses a texture-lookup to determine the depth of the back-face computed in the previous rendering pass. This combines the intersection data of two layers into a single frame buffer, preventing a transfer of a depth buffer.

3.2.4 Depth-Peeling

The most general solution for the multiple layers problem is the *depth-peeling* algorithm, as described in [Everitt, 2001]. With depth-peeling, any number of intersection segments can be computed and the segments are computed in sequential order of appearance with relation to their depth.

Computing the first intersection segment is done in the same way as for the front/back-face culling method. To determine the next intersection segment, the depth buffers of both rendering passes are copied to two separate textures. This data transfer is again local to the memory of the graphics hardware, thus it is relatively fast. Next, the clipping volume is rendered again using a fragment program with equal functionality as that of the front/back-face culling method, but with the addition that it discards a fragment if the depth of the fragment is less than or equal to the depth value stored in the texture

that contains the depth buffer of the previous pass. This texture thus acts as a second depth buffer. If a fragment is discarded, no information is written to the frame buffer or depth buffer. The fragments that pass the additional depth test form the front-most layer that lies behind the layer of the previous pass. Computing a third segment is done in a similar way, except that the depth buffer of the second pass is used for the additional depth test. In general, pass i uses the depth buffer of pass $i-1$ to discard any fragments of previous passes. Note that the standard depth buffer offered by the graphics hardware is also still used to determine the front-most faces in every pass. Thus this algorithm is in fact a dual depth buffer approach as described in [Diefenbach, 1996].

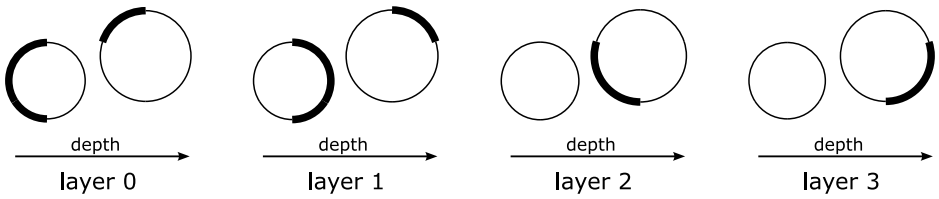
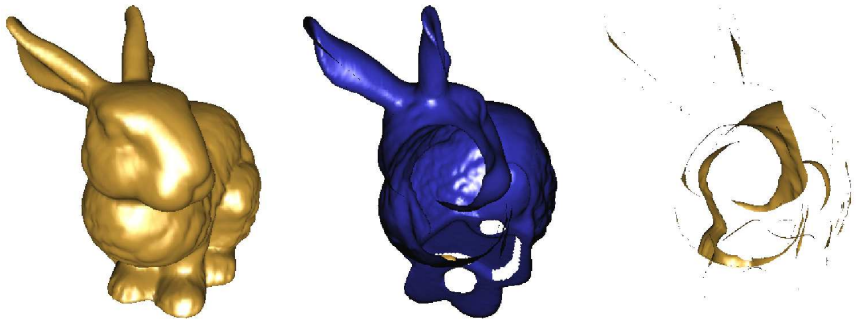


Figure 3.3: The concept of depth peeling; the four depth layers of a simple scene.

Each of the rendering passes can be used in combination with the optimization of storing both layers of a segment in a single frame buffer. Figure 3.3 gives an example of the individual layers that are 'peeled' off the clipping geometry. Figure 3.4 given an example of the depth-peeling algorithm applied to an actual model, in this case the Stanford Bunny. The front-faces are colored orange, while the back-faces are colored blue. Note that rendering the front and back layers of each segment are separate processes; each have their own depth buffer texture associated. The depth buffers of the front layers are only used for consecutive front layers and the same property holds for the back layers.



(a) (b) (c)

Figure 3.4: Depth peeling applied to the Stanford Bunny. Figure (a) shows the first layer, (b) the second and (c) the third.

A useful feature of current graphics hardware is that there is special support for textures that contain depth information. This construction is often used in shadow computations, thus the depth test that uses a texture as input is known as a shadow test. To use this test, the fragment program does a texture-lookup in the special depth texture. The result of the lookup is either zero or one, indicating the result of the depth test. The actual comparison does not have to be done by the fragment program. This not only increases performance, but also allows the depth comparison functions to be as easily controlled as when using the fixed function pipeline.

Front/Back-face Culling and Depth-Peeling

The use of front- and back-face culling in combination with depth-peeling is not mentioned in [Everitt, 2001], since the algorithm originally was designed to work with open meshes as well, being more general. An application in computer graphics was order independent transparency for example. For this particular purpose, the use of depth-peeling in combination with the front and back-face culling approach changes behavior a bit for non-well-defined clipping volumes. Suppose the clipping volume contains a self-intersection, such as the one depicted in figure 3.5. The left figure shows the second layer if no face culling were to be used, while the right figure shows the second layer with face culling enabled. The difference is in the gray area in the middle. Without face culling, the gray area would not be included in any of the segments, while with face culling it would be rendered twice. Neither of these solutions is desirable, but then again the clipping volume is not well-defined since it contains a self-intersection. To ensure correct behavior, self-intersections in the clipping volume should be removed by for example a union operator as is common in CSG operations to combine multiple parts of a surface into one object.

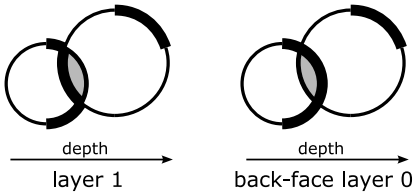


Figure 3.5: Depth peeling with front/back-face culling awareness.

As a summary, the resulting algorithm uses the front and back-face culling technique to render both layers of the first segment. If multiple segments are required, the contents of the depth buffer are transfered to a depth texture, using separate textures for the front and back layers. For the remaining segments, depth peeling is used to cull depth ranges occurring in previous passes. The back layer of the last segment is rendered with the depth compare function set to render the back-most triangles. This may lead to rendering too much during the volume rendering, but those artifacts are in general much less noticeable than rendering too little. The normal vectors are only stored for the front layers, using only two components out of three. The depth values of both the front and back layers are combined with the normal vector and the resulting four components are stored in the frame buffer, giving all required data for each segment.

3.2.5 Performance Summary

The overall performance of the rasterization method is very good. Graphics hardware does an amazing job at rasterization in terms of both performance and image quality. The algorithm was tested on both a PCI-Express nVidia Quaddro FX 4400 and an AGP-8x nVidia GeForce 6800 Ultra. The performance between the two varied quite a lot, although the speeds of the two GPUs are almost equal. The reason is that the performance bottleneck is the transfer of the frame buffer from video memory to host memory. The former card was connected using a PCI-Express x16 bus, which is much faster than the AGP-8x bus of the latter card. The Quaddro reaches transfer rates of up to 580 MB per second, while the GeForce does not break the 180 MB per second boundary. The triangle fill-rate of the two cards is almost equal and does not contribute significantly to the rendering time with clipping volumes below 100,000 triangles. Without the memory transfer, both cards are capable of fill-rates approximating 30

million triangles per second.

Rendering more segments reduces performance, because more memory transfers are required. Fortunately the number of segments that is drawn is freely specifiable with the depth peeling algorithm, allowing for a trade-off in performance and image quality. A solution to reducing the time spent in the memory transfer is to render the clipping volume at a lower resolution than at which the volume rendering is to be performed. This naturally decreases image quality and may be the cause for aliasing artifacts. Many compromises are possible, such as rendering the first layer at full resolution and subsequent layers at a lower resolution. These optimizations were not investigated.

Even though the memory transfer is very slow, the rasterization method is massively faster than the ray casting method. The downsides are that it requires relatively new graphics hardware and certain versions of the video driver. There also are some problems with nVidias DualView; it has a massive impact on the performance of the graphics card. The memory transfer is not affected much, but the triangle fill-rate drops as much as a factor ten with DualView enabled.

Chapter 4
Volume Rendering with Volume Clipping

This chapter covers the integration of the depth-based rasterization method for volume clipping discussed in the previous chapter with two different ray casting-based volume renderers. The first volume renderer is a software component called the DirectCaster developed by Philips Research Aachen in a project that was funded by Philips Medical Systems. The second volume renderer is a GPU based volume renderer developed during this project with some basic optimizations. A detailed look at both volume renderers is taken and the various stages of the integration are discussed.

4.1 Software DVR: The DirectCaster

The DirectCaster is a software volume rendering component developed by Philips Research Aachen. It claims to be a high performance software-based direct volume rendering engine giving a high image quality. These claims are met due to a number of optimizations. First these optimizations are discussed, combined with the global architecture of the DirectCaster. Then the integration of the rasterization method for volume clipping is discussed, together with the solutions for various artifacts that were caused by the integration. A number of possible speedups due to the volume clipping is also covered, ending with a summary of performance in both image quality and speed.

4.1.1 Architecture of the DirectCaster

The DirectCaster is a ray casting based volume renderer. It contains many optimizations, of which the most important are mentioned below. A few basic ones include the use of fixed point mathematics and the fact that all computations are performed in voxel space. The latter prevents transforming volume space coordinates to voxel space when a voxel lookup is required.

An important feature of the architecture of the DirectCaster is that it uses two levels of blocks. A block is a cube-shaped primitive covering a number of voxels. The block structure imposes hierarchy on the volume data, similar to an octree. The use of these blocks leads to various performance optimizations, without losing image quality. The first level of blocks consist of the level-1 blocks. These are solely meant for skipping large “empty” regions of the volume data. Empty means that all voxels in that particular block are mapped to a completely transparent opacity by the transfer function. Each level-1 block consists of a number of level-0 blocks, typically in a 4x4x4 arrangement, giving 64 level-0 blocks per level-1 block in total. Level-0 blocks are smaller and typically contain 2x2x2 voxels.

The much smaller level-0 blocks have a more intricate use. Each level-0 block has a *castpiece* function attached to it, which is in principle a pointer to a rendering algorithm specifically designed for the type

of data in that block. If a block is empty, a null-function is attached and no rendering is performed. This is a means of quickly discarding empty blocks. Other specialized rendering functions exploit a constant color throughout the entire block for example, in which case color interpolation is not necessary. During initialization, the block hierarchy of the volume volume is constructed and each level-0 block is classified. A suitable castpiece function is attached to each level-0 block. This mechanism saves certain computations in those areas where they are not necessary.

Another optimization that is used throughout the DirectCaster is that the traversal of the volume is performed by a cache-aware algorithm. The blocks of the volume data are arranged in a cache-optimal way, maximizing the use of the cache during the rendering. The pixels of the screen are traversed in a cache-optimal way during rendering. Instead of traversing scan-lines, pixels are traversed in a way such that the data accessed for each pixels is cache-optimal with respect to what volume data is accessed while processing each pixel.

The DirectCaster uses an object-aligned sampling strategy. This means that it tries to sample at the voxel boundaries as much as possible. This leads to a performance increase, since sampling at voxel boundaries requires only bilinear interpolation, opposed to trilinear interpolation at locations elsewhere in the voxels.

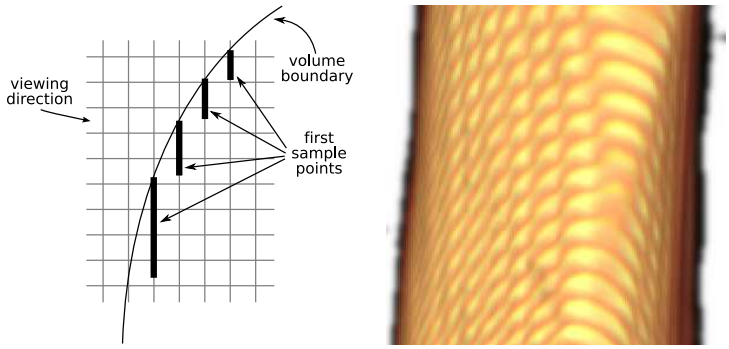
Since software rendering is relatively slow, rendering only part of the screen and interpolating the missing parts is a common optimization technique. The DirectCaster also applies this technique, but it tries to maximize the performance gain and minimize the loss in image quality. It contains an adaptive image interpolation algorithm, that skips large areas of pixels that have the same color and depth, but renders at a higher resolution near areas with large changes, such as surface boundaries.

4.1.2 Reduction of Ring Artifacts

The artifacts often referred to as “ring artifacts” are a common problem among ray casting-based volume renderers. These artifacts are caused by a to large sampling distance near regions that have a present an abrupt change in color or opacity. Figure 4.1 (a) shows this phenomena in a diagram. What happens is that near a smooth boundary of the volume where there is a relatively large change of color or opacity, neighboring pixels may have a different number of samples in the interpolated region due to a different starting point. This causes neighboring pixels to have very different colors. Figure 4.1 (a) shows the case for object-aligned sampling, but the problem also arises for viewing-aligned sampling.

The DirectCaster contains a mechanism to reduce these artifacts. During ray traversal samples are taken only at the voxel boundaries. Since this is an object aligned sampling strategy, the sampling distance is not constant and may vary depending on the viewing angle. Note that the voxels need not be cube-shaped, so the actual sampling distance may be even bigger. When a sample is taken, its opacity is compared to the opacity of the previous sample. If the difference in opacity is above a certain threshold, the sampling rate is increased. Due to the higher precision, the discontinuities as visible in figure 4.1 (a) become smaller and thus less visible.

Limitations of the available implementation of this algorithm is that the sampling rate is only increased for large changes in opacity; differences in color are left untouched. Also is the threshold for the opacity difference compiled in as a constant. For very transparent transfer functions this is a problem, since this may cause the threshold to be never exceeded and vague ring artifacts to appear nonetheless.



(a) (b)

Figure 4.1: Ring artifacts in volume rendering; (a) gives a schematic overview and (b) shows an example of their occurrence.

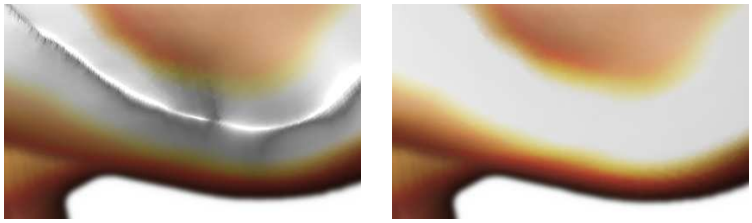
4.1.3 Modifications for Volume Clipping

To make volume clipping possible with the DirectCaster, a number of modifications was made. This was done in line with the idea behind the DirectCaster; to obtain unprecedented image quality with the highest possible performance.

The first part of the integration offers basic support for volume clipping. Some data structures were added to allow the intersection data to be passed to the DirectCaster scene. Before a ray is casted, the intersection data specific for that ray is copied to another data structure containing numerous elements of ray-specific information. The actual clipping is done in the `ACCUMULATE_INTERVAL` template, which computes a color for a given sample. The modified version discards the sample if it happens to be outside the clipping volume.

Discarding samples outside the clipping volume is crucial for volume clipping, but this feature on its own gives a bad performance. Regions of the volume that should be sampled during normal volume rendering and are outside the clipping volume are still sampled at a high rate while they do not contribute to the final image. These areas are discarded only at the very lowest level. For volume probing there exists an optimization to prevent traversing these regions. During the computation of the start and end points of a ray, the ray is checked for an intersection with the clipping volume. If the ray does not intersect with the clipping volume, the entire ray is discarded. If the ray does intersect, the start and end points of the ray are adjusted to the points of the first and last hits with the clipping volume respectively. With this optimization, unnecessary sampling is done only in the regions between the intersection layers, which correspond to the cavities of a concave clipping volume. These areas are for most practical applications relatively small. Moreover, the traversal of such regions is rare, as they are easily culled by preceding regions of the volume by early ray termination. For convex clipping volumes this optimization removes all unnecessary sampling.

If a sample that is inside the clipping volume is to be accumulated, the modified template computes the distance to the clipping volume in the viewing direction. This is not the precise distance, but it is an acceptable approximation, as mentioned in section 2.4. Using this distance and a certain threshold, gradient impregnation is applied using a linear weighting function. This is a vital step to get a high image quality. Figure 4.2 shows various alternatives for the gradient problem. Figure 4.2 (a) shows the result of not performing gradient impregnation. The gradient used along the edge does not give a good representation. The edge appears to have differences in height, while it is in fact flat. Using



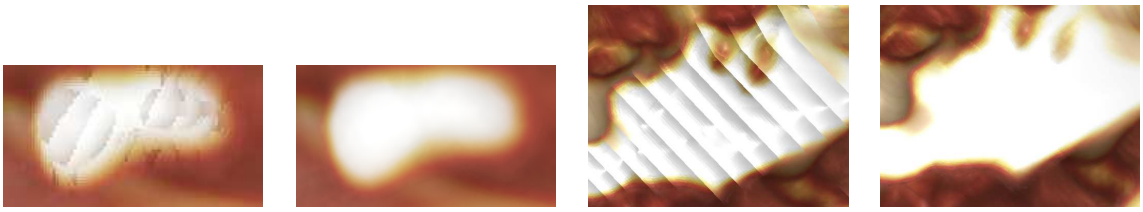
(a) (b)

Figure 4.2: The edge of a clipping volume without (a) and with (b) gradient impregnation.

the gradient impregnation method as described in section 2.4 results in figure 4.2 (b), where the edge appears flat.

The ring-artifact reduction algorithm is applied when a higher sampling rate is indispensable for the image quality produced by the DirectCaster. To maintain this high image quality with volume clipping, this algorithm should also be invoked near the edges of the clipping volume. Often there are steep changes in the optical model along the edge of the clipping volume. Usually these changes are caused by the fact that subsequent samples are outside and inside the clipping volume respectively. When only the volume is considered, the difference in opacity may not be very large, so the sampling rate would not be increased.

In order to have a higher sampling rate near the edge of the clipping volume, the ring-artifact reduction algorithm was modified such that the sampling rate is not only increased when there are steep changes in opacity, but also when the ray segment crosses the boundary of the clipping volume. For performance reasons, this is only applied when the boundary is facing the viewer, for similar reason that the gradient is the clipping volume is used only for only these boundaries, as discussed in section 2.4. The intersections at the back of the clipping volume are partially or often even entirely occluded by the part of the volume in front of them. Figure 4.3 shows the result of the modification. Figures 4.3 (a) and (c) show a blowup of an area near edge of the clipping volume without an adaptive sampling strategy applied. There are many slicing artifacts visible. Figures 4.3 (b) and (d) show sample areas rendered with the modified algorithm; there are no more slicing artifacts along the edge.



(a) (b) (c) (d)

Figure 4.3: An area of a volume near the edge of the clipping volume without (a), (c) and with (b), (d) an adaptive sampling strategy applied.

(a) (b) (c)

Figure 4.4: The DirectCaster in action with volume probing with a sphere (a), volume cutting with a cube (b) and volume probing with the Stanford Bunny (c).

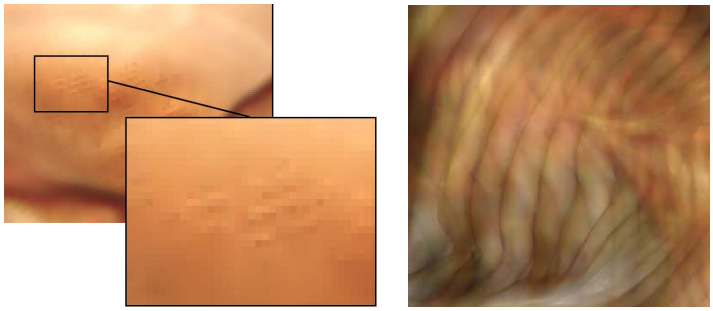
4.1.4 Other Possible Modifications

The optimizations mentioned in section 4.1.3 give good image quality and for volume probing also a relatively high rendering speed. For volume cutting however, a higher rendering speed requires different optimizations. With the current modifications, time may be wasted in the traversal of non-transparent blocks that are inside the clipping volume (which is the area to be clipped with volume cutting). One way to solve this would be to modify the ray traversal algorithm to skip these blocks altogether, instead of sampling at each voxel. Note that sampling rate is not adaptively increased in these blocks. This optimization was not implemented due to a lack of time.

Following the architecture of the DirectCaster, a logical improvement of rendering speed would be do assign the dummy castpiece function to blocks outside the clipping volume during the classification. However, since classification is performed without any geometry information, this optimization belongs to the category of volume-based approaches, losing the ability to interactively modify the clipping volume. Therefor this optimization was not implemented.

4.1.5 Results

Figures 4.4 (a) and (b) show the modified DirectCaster rendering a volume with volume probing and volume cutting respectively. The image quality with respect to volume rendering is equally high as the DirectCaster produces without volume clipping; there are no artifacts due to aliasing or a too low sampling rate. Considering volume clipping, there are no artifacts due to not well-behaving gradients, or a too low sampling rate near the edges of the clipping volume. The addition of volume clipping has virtually no impact on the rendering speed; some modifications increase performance while others decrease it. For most practical applications, the rendering speed is equal with or without volume clipping. On a currently modern PC however, this does still not give interactive frame rates. On a dual Intel Xeon 3.0 Ghz a 512^3 data set can be rendered at approximately 2 to 8 frames per second on a 256^2 viewing window, with the frame rate depending on the transfer function. Transfer functions that include a step-function for the opacity component give much higher performance than those with a high degree of transparency, as more sampling is required in the latter case.



(a) (b)

Figure 4.5: Artifacts produces by the DirectCaster; (a) shows the problems with adaptive image interpolation and (b) shows the slicing artifacts caused by a too transparent transfer function.

All requirements listed in section 1.6 have been met. Most of the requirements were met by using the rasterization variant of the depth-based volume clipping techniques. With this technique integrated in the DirectCaster, the clipping volume can be specified as a high-resolution polygon mesh and it can be interactively modified or reoriented without decreasing performance. This allows for a high variety of clipping volumes and offers a high degree of interactivity. Figure 4.4 (c) shows an example of clipping a volume against the Stanford Bunny, a complex polygon model consisting of $69,451$ triangles. Using such a complex model has no effect on the rendering speed, since the memory transfer still forms the bottleneck. With current graphics hardware, such as a nVidia GeForce 6800 Ultra, the rendering of triangles becomes dominant with a clipping volume consisting of approximately half a million triangles.

Some artifacts are still visible in the renderings produces by the DirectCaster, although they do not originate from the modification due to the volume clipping. One of these artifacts is caused by the adaptive image interpolation. Although the adaptiveness reduces the visibility to a minimum, interacting with the scene gives artifacts similar to the ones seen in JPEG or MPEG streams. Figure 4.5 (a) shows these artifacts in more detail, although they are more visible in a moving image.

Despite all the techniques present in the DirectCaster to maximize image quality, some artifacts may still appear. An example is given in figure 4.5, where an example of slicing artifacts is given. The exact reason why these artifacts are caused is hard to determine, since the entire rendering algorithm is of a high complexity.

4.2 Hardware DVR: A GPU-Based Volume Renderer

As indicated by the results of chapter 3, the performance bottleneck of the rasterization method is transferring the data from video memory back to host memory. If the volume rendering were to be done on the video card, this expensive transfer could be saved. This was a reason to look at the possibility of doing volume rendering on the GPU.

Since the existence of programmable graphics hardware, there have been many attempts at performing volume rendering on the GPU. Until recently, the programs to be run on the GPU were bound to a number of limitations. One of these limitations had to do with the dynamic instruction count of fragment programs, which is the number of instructions that may be executed for every instantiation of

the program. Another limitation was the absence of instructions that allowed common programming paradigms such as if-statements and for-loops. A ray casting-based volume renderer typically contains a loop that iterates the many samples along a ray for each pixel. This loop is the cause for many instructions for each pixel, easily exceeding the instruction count limit. This is one of the reasons that most of the current GPU-based volume rendering implementations used an object-aligned rendering method for doing volume rendering [Kniss et al., 2001] [Weiskopf, 2003], which does not suffer from this restriction. Another reason is that object aligned rendering is more compatible with the functionality of the fixed function pipeline of the GPU.

4.2.1 Ray Casting-Based GPU Volume Rendering

Recent developments in graphics hardware resulted in shader model 3.0, which relaxes most of the restrictions imposed on the shaders by earlier shader models. The new shader models allows fragment programs to have a virtually unlimited dynamic instruction count and offers new looping and branching constructs [Kessenich et al., 2004]. These new possibilities allow ray casting-based volume rendering to be performed on the GPU. Since very little research has been done in this area, it is interesting to explore the possibilities of such a renderer. It is also be interesting to compare the results of such a method to a software implementation of ray casting-based volume rendering, such as the DirectCaster. Therefor the GPU volume renderer that was implemented uses a ray casting-based approach, taking full advantage of the latest graphics hardware. This ray casting-based volume renderer will be hereafter referred to as the GPUCaster.

At the time of writing only very few references to this approach for volume rendering were available. The most significant were the presentation “Volume Rendering for Games” given by Simon Green of the nVidia corporation at the Game Developers Conference at the end of March 2005 and the “Interactive Visualization of Volumetric Data on Consumer PC Hardware” course by Klaus Engel, Daniel Weiskopf and others in 2003. The latter only mentions the possibility and does not treat the subject in detail. The main focus of the presentation of Simon Green is on application of volume rendering in games. The volume data is procedurally generated, instead of being contained in a three-dimensional array. This is very different from medical data, as there are no memory constraints and the requirement to have a realistic representation of the data is less strict.

The basic architecture of the GPUCaster is similar to most general purpose GPU applications. The process of ray casting is very different from that of rasterization, all the work is done inside a fragment program. To instantiate the fragment program for each pixel, a rectangle covering the entire screen is drawn first. The four corner points of the rectangle are passed through a vertex shader, which provides the input for the fragment program. In this case the input is the pixel coordinate and the origin and direction of the ray. Note that the vertex shader only specifies this information on each of the corners of the rectangle and that the GPU linearly interpolates this data for each pixel covered by the rectangle.

The fragment program requires input to perform the volume rendering. Variables that are equal for all rays, such as the step size or early ray termination threshold, are provided as uniform variables. Arrays that require random-access reading by the fragment program are all provided in the form of textures. The voxel data is provided as a single three-dimensional texture and the transfer functions as one-dimensional texture. The start and end points on the ray correspond to the intersection points of the ray with the bounding cube of the volume data set. The implementation of Simon Green mentioned earlier uses the most obvious way; a ray-box intersection algorithm is used to compute these points. The GPUCaster uses a different method. Prior to the instantiation of the fragment program the bounding cube of the volume is rendered twice; the front faces in the first pass and the back faces in the second pass.

After each pass, the content of the depth buffer is transfered to a texture. These two textures are also provided as input to the fragment program. Inside the fragment program, the depth values of the front and back intersections with the bounding cube can be obtained from these two textures. Combined with the origin and direction of the ray, the two points can be computed. This approach was chosen because it allows easy extension to volume clipping. Instead of drawing the bounding cube of the volume, a different shape can be drawn to change the entry and exit points for each pixel. The rasterization of the cube is of negligible cost and copying the depth buffer can be avoided in the future by using the `EXT_framebuffer_object` extension to OpenGL [OpenGL Architecture Review Board, 2005]. More information on this extension is given in section 4.2.7.

When all input is set and the fragment program is instantiated, the ray traversal can start. Due to the use of a high level shading language [Kessenich et al., 2004] and the new looping and branching constructs, the actual algorithm is very small and contains few tricks. The fragment program loops over all the samples on the ray in a front-to-back order. At each sample the volume is sampled, using the linear interpolation offered by the GPU. The gradient is computed by sampling at the six neighboring voxels. Using the sample of the current voxel, a lookup is done in the transfer function to obtain the color and opacity for the current sample. Lighting is performed using the color, gradient and viewing geometry to compute the final color. Finally the current pixel color is adjust using a standard volume rendering equation.

The whole fragment program consists of very few lines of code and a basic implementation was created in a single day. This is possible due to the simplicity of the method.

4.2.2 Lighting Model

Volume shading was added to the GPUCaster by using the common Phong lighting model [Phong, 1975]. According to the Phong lighting model, the final color consists of an ambient, a diffuse and a specular component. The ambient and specular components can be computed normally, but since volume samples are commonly transparent, there is an issue with computing the diffuse lighting intensity. The intensity of the diffuse lighting component is given by the dot product between the light vector and the gradient at the sample. This gives the intensity curve shaped like a cosine. The problem is that when the gradient points backward, this intensity becomes negative. Using a negative value as an intensity would make the sample absorb light, which is clearly incorrect. When rendering surfaces, a common solution is to take the maximum of zero and the computed intensity, giving the red dotted curve in figure 4.6. This is correct since surfaces with gradients pointing away from the viewer should not be visible.

For volume rendering this also works, but other possibilities also exist. Instead of ignoring samples with gradients pointing away from the viewer, one could take the absolute value of the diffuse lighting intensity. This is corresponds to mirroring the gradient of a sample against the viewing plane in case it points away from the viewer. The intensity curve is shown as a purple dashed curve in figure 4.6. As long as the the lighting vector is parallel to the viewing direction, this solution works fine. The colors in the final image appear more vibrant than when clipping negative result to zero. However, different lighting vectors give bad results. The scene appears to be lit by two light sources; one being a mirrored direction of the other.

Yet another solution is to scale and translate the cosine curve such that is does not become negative. The corresponding intensity curve is shown as a solid red line in figure 4.6. The problem with this solution is that samples that have a gradient perpendicular to the lighting vector appear with half the

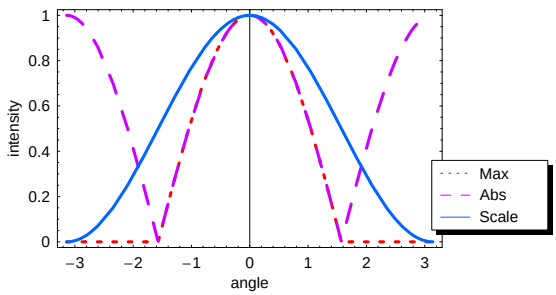


Figure 4.6: Different diffuse lighting contribution curves.

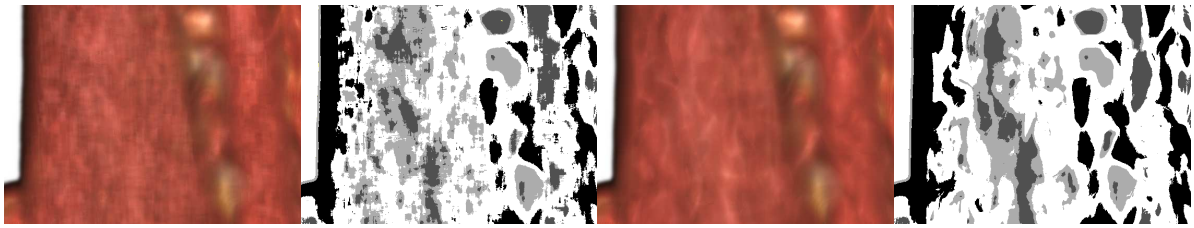
possible intensity, which is not intuitive. Squaring the resulting intensity improves this, but the problem remains. This is unwanted behavior, since it partially destroys the ability to use lighting to perceive the shape of the volume. Remaining behavior is correct however.

It is difficult to say which solution is best. The maximum curve gives good results, but gives the darkest image of the three solutions. The absolute value curve only works for a lighting vector perpendicular to the viewing direction, while the scaling curve makes image appear lighter, but decreases the effectiveness of lighting. As a compromise, the scaling curve seems to be the best solution. To enhance the effectiveness of the lighting, the power which the scaled curve is raised can be increased. The DirectCaster uses the absolute value curve to compute the diffuse component, since it only offers support for a lighting vector parallel to the viewing direction.

There is yet another problem with lighting. In an area of the volume where there is no variation of the data, the gradient is equal to the zero-vector. Using this gradient in lighting computations would result in only an ambient contribution of the sample, which is incorrect. A better approach is to the average diffuse and specular contributions, assuming that all possible gradients are uniformly distributed. This makes areas without a gradient appear more vibrant, although their color is not affected by changes of the lighting or viewing vectors.

4.2.3 Precision

Currently the latest GPUs have support for 16 and 32-bit textures. This is a beneficial property for volume rendering, since the source data often is of a 12 or 16-bit nature. Using a too low precision can lead to noise in the final image, especially in regions with high transparency. In these transparent regions the final color is composed of many samples and thus many gradients as well. This causes noise artifacts to become more visible. Figure 4.7 shows a transparent area of a volume rendered using both 8-bit and 16-bit textures. Both a normal rendering is shown and a posterized version that shows the distribution of colors more clearly. The images show that the color distributions are much more smooth when using 16-bit textures. The presence of noise is bad, as it destroys the ability to use lighting to perceive the spatial structure and orientation of the volume.



(a) (b) (c) (d)

Figure 4.7: The differences between rendering using 8-bit (a) and 16-bit (c) textures. Figures (b) and (d) show the color distributions more clearly.

4.2.4 Optimizations Implemented

To enhance the performance of the GPUCaster, a number of optimizations was implemented. Because the available time was severely limited, many optimizations are left unexplored. Nonetheless a considerable speedup was achieved using the primitive techniques described below.

Early Ray Termination

As an optimization, the ray traversal is terminated as soon as the pixel opacity exceeds a certain threshold. This is a common optimization in volume rendering and is called early ray termination. The threshold at which a ray is terminated is close to opaque. Reducing this threshold reduces image quality, while the performance gain is only minimal. The majority of the speedup comes from not traversing the volume while the pixel opacity is already completely opaque.

Skipping Empty Regions

The GPUCaster uses a static sampling rate. To achieve a high image quality, this sampling rate should be less than one voxel. However, by setting the sampling rate very high, a lot of time is wasted in traversal of the so called “empty regions”, parts of the volume that are mapped to a completely transparent opacity by the transfer function. To avoid this to some extent, the start and end positions of the ray are adjusted to skip any leading and trailing parts of the ray that span only empty parts of the volume.

To this purpose, a binary volume is computed at a resolution lower than that of the original volume; the cells of the binary volume typically consist of 64 (4x4x4) or 512 (8x8x8) voxels. Each cell contains only a single bit, indicating whether there are one or more voxels that are not fully transparent. This data is dependent on the transfer function. Before the instantiation of the fragment program, instead of drawing the bounding cube of the volume, all non-empty cells of the binary sub-volume are drawn as small cubes. By using the depth buffer in combination with different depth comparison functions, two textures can be constructed that contain the depth of the front-most and back-most faces of all the cells respectively. This changes the start and end point for each ray, preventing the traversal of some of the empty space in the volume.

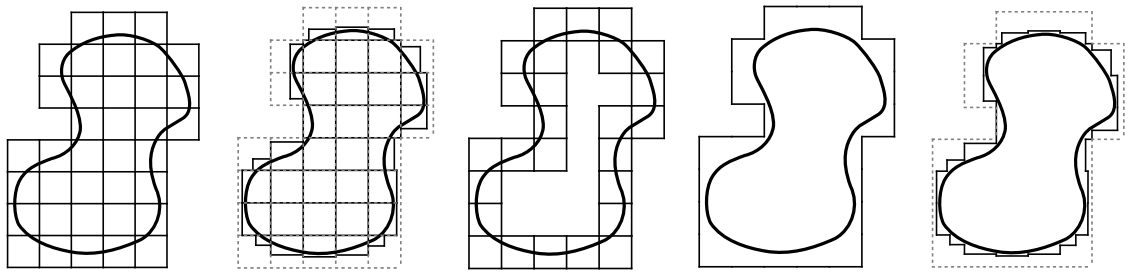


Figure 4.8: Using a binary volume to mask empty regions of the volume; (a) shows the normal grid, (b) a tightly fit version, (c) shows only the cells around the contour, (d) shows only the contour of the union of all non-transparent cells and (e) shows a tightly fit contour.

Tightly Fit Cells

To optimize this even further, the cells of the binary sub-volume can be made of non-uniform shape. Instead of only a bit, each cell may also contain the coordinates of the bounding cube of the non-empty region within that cell [Bescós, 2004]. Using this optimization, even empty regions inside a cell can be skipped, causing the grid of cells to be more tightly fit around the non-empty part of the volume. This is demonstrated in figure 4.8. Figure 4.8 (a) shows a normal grid of cells around a volume, while figure 4.8 (b) shows a tightly fit grid.

Not Drawing Occluded Cells

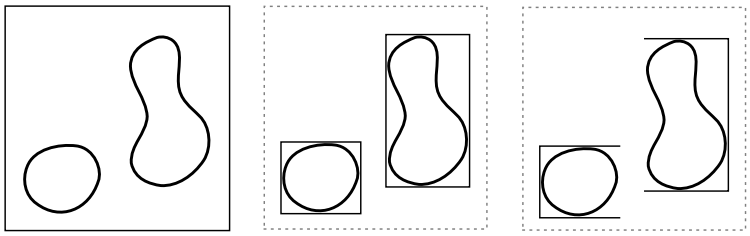
To skip as many empty parts of the volume as possible, the resolution of the binary sub-volume should be as high as possible. A high resolution also means that many small cubes need to be drawn. However, many of those small cubes are completely occluded by surrounding cubes in most practical cases. These occluded cubes need not be drawn.

Preventing drawing occluded cells can be done by using the fact that cells that have a minimum voxel value that is above the minimum threshold of the transfer function, are completely transparent. Such cells are not part of a transparency boundary of the volume. To achieve correct behavior, some special handling of cells near the edge of the volume data set is required. This idea is visualized in figure 4.8 (c).

However, this method still draws some unnecessary triangles. By looking at the problem at triangle level, an even bigger optimization is possible. Again considering non-tightly fit cells, an edge of a cell need not be drawn if the cubes on either side of the edge are both flagged to be drawn, because this means the edge is not on the boundary and thus occluded by other edges. The set of edges that is eventually drawn makes up the boundary of all regions that need to be rendered. This is shown in figure 4.8 (d).

An optimal solution would be to combine the optimizations shown in figures 4.8 (b) and (d), resulting in figure 4.8 (e). However, special care should be taken when removing edges. If two separate parts of the volume happen to be in neighboring cells, a tightly fit contour may produce two separate cubes. Thus a side of a cube should only be removed if it touches the side of the neighboring cell and has

equal dimensions. If this is not done, gaps may appear that may cause certain parts of the volume to be falsely discarded. This is shown in figure 4.9, where the two tightly fit squares are not attached to each other, leaving a gap in the middle.



(a) (b) (c)

Figure 4.9: A problem with combining the removal of occluded edges with tightly fit cells. Figure (a) shows the contour with occluded edges removed, (b) the tightly fit version of the contour and (c) the combination with a gap in the middle.

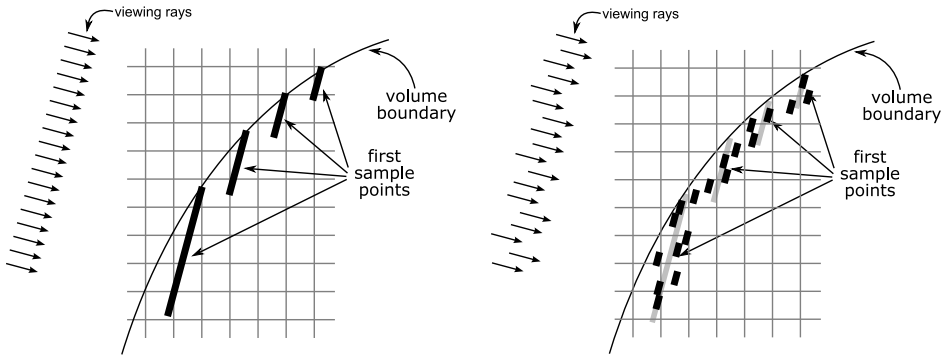
4.2.5 Reduction of Ring Artifacts

By using a high sampling rate, the visibility of ring artifacts is minimized. This has an impact on performance however. Using an adaptive sampling rate strategy is not preferable on a GPU, since fragment programs involving a lot of branching give bad performance. To reduce the visibility of ring artifacts at low sampling rates, noise is used to mask these artifacts. Figure 4.10 shows why ring artifacts occur using a viewing aligned sampling strategy. The positions of the first samples that are not mapped to a translucent opacity form a non-continuous line, causing discontinuities in the optical model. Figure 4.10 shows that by using noise to offset the starting position of each ray, the discontinuities in the initial sampling positions are masked out. The noisy behavior of the first sample point results in noise in the final image, but this is preferable over the ring artifacts. Figure 4.11 shows an example of this technique applied in practice. When rendering at high resolutions, the noise is hardly visible since it is dependent on the size of the pixels. At lower resolutions a light blur filter or interpolation filter can reduce the visibility of the noise.

4.2.6 Modifications for Volume Clipping

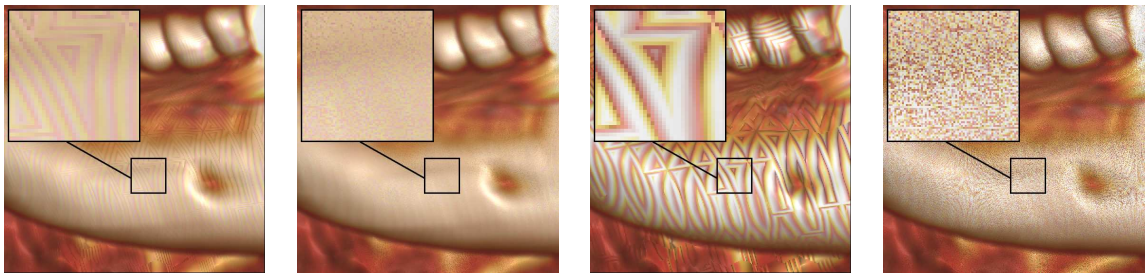
The GPUCaster allows an easy and efficient integration of depth-based volume clipping techniques, especially those that take a rasterization approach. The GPUCaster uses the depth structure of the bounding cube of the volume to determine the start and end points of the ray. The output of a single segment of the rasterization method for volume clipping is very similar; two depth buffers that contain the start and end depths and one additional buffer with gradient information. By combining the depth textures obtained from the bounding cube and the clipping volume, volume clipping is achieved.

First volume probing is assumed, the techniques will later be extended to support volume cutting as well. As a start point for the ray, the greatest depth should be selected between the front of the bounding cube and the clipping volume. This corresponds to the point that is furthest away from the viewer. For the end point, the reverse holds; the point that is closest to the viewer should be selected. The resulting ray segment is the intersection of the two ray segments defined by the depth structures of the bounding cube of the volume and the clipping volume. This is depicted in figure 4.12.



(a) (b)

Figure 4.10: Viewing aligned ring artifacts are caused by a sudden change in offset of the first sampling point in the volume (a). By applying noise to the starting point of the ray, this difference is masked by the noise (b).



(a) (b) (c) (d)

Figure 4.11: Figures (a) and (c) show ring artifacts on an area of the volume rendered with a high and a low sampling rate respectively. Figures (b) and (d) show the same area using noise to mask the ring artifacts.

Solving Problems with Multiple Segments

The volume clipping technique described above only supports one segment of the depth structure of the clipping volume. Rendering multiple segments is an easy extension. Obtaining a second segment of the clipping volume can be obtained by the same techniques described earlier. The first segment is rendered as described above. For any consecutive segments, there is only one additional input required, being the output of the volume rendering of the previous pass. This output should be copied to a texture and passed as an input to the next phase. The next phase should set the initial color to the output of the previous pass. This makes the multiple volume rendering passes act incrementally.

Solving Problems Volume Cutting

Adding support for volume cutting can be achieved by changing the meaning of the layers of the clipping volume. The front layer of the depth structure of the clipping volume contributes only the back part of a segment that should be rendered. Likewise, the back layer of the clipping volume is used for determining the starting point of a segment. There are two special cases however; the first segment

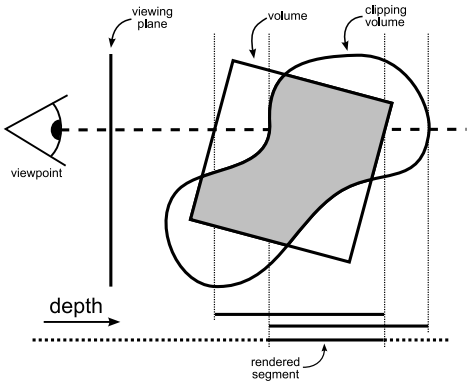
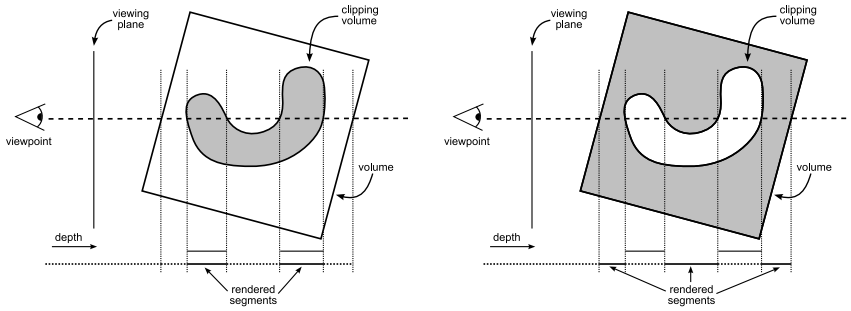


Figure 4.12: The intersection of the segments defined by the bounding cube and the clipping volume gives the segment that should be used for volume probing.



volume probing volume cutting

Figure 4.13: Swapping the meaning of front and back layers of the depth structure of the clipping volume yields volume cutting instead of volume probing.

that should be rendered has no layer of the clipping volume for computing the starting point of the ray and the last segment has no layer for computing the endpoint of the ray. For these two cases the front and back depth buffers of the bounding cube of the volume should be used instead. Note that there is no need to combine depth buffers when using volume cutting. For all remaining segments, both layers are present. The process of swapping the meaning of the layers is depicted in figure 4.13.

Visualizing the Clipping Volume

To increase the ability to perceive the shape of the clipping volume, the possibility to visualize the clipping volume was added to the GPUCaster. By treating an intersection with the clipping volume as a special kind of sample and adjusting the pixel color accordingly, the clipping volume becomes visible as thin layer. By applying shading, using the normal vectors of the clipping volume, the shape of the clipping volume becomes very clear. To prevent the volume to become invisible, the transparency of the clipping volume can be adjusted. Figure 4.14 shows an example rendering of a volume data set clipped using the Stanford Bunny.

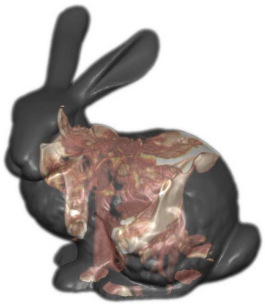


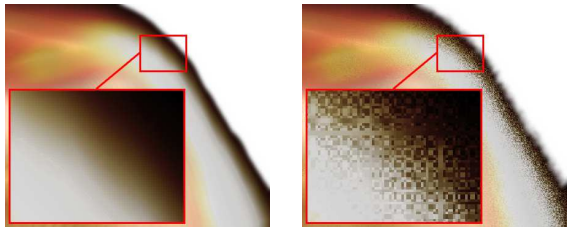
Figure 4.14: Volume probing with the clipping volume visualized as well.

4.2.7 Results and Possible Improvements

The resulting volume renderer has equal functionality as the modified DirectCaster. The functionality demonstrated in figure 4.4 is also available in the GPUCaster. Therefore it is not printed here. Both the image quality and performance are very good, much higher than one would expect of a volume renderer developed in such a short time. A more elaborate discussion on the performance and image quality of the GPUCaster combined with a comparison to that of the DirectCaster is given in section 4.3.

One clear performance benefit is that the expensive memory transfer from video memory to host memory is no longer required. Therefore the volume clipping is of insignificant importance to the performance of the volume renderer. Clipping volumes consisting of 100,000 polygons can be used without any noticeable performance decrease. However, still some time is wasted with copying data around, although these copies are internal to video memory. The reason is that if the output of a rendering pass is to be used as input for another, the contents of the frame buffer should be copied to a texture. Not being able to render directly to texture memory is a known restriction of OpenGL. In early 2005 the OpenGL Architectural Review Board has accepted the `EXT_framebuffer_object` extension, which does allow rendering directly to textures. At the time of writing however, the latest official video driver from nVidia do not yet support this extension.

Combining the technique for the reduction of ring artifacts with volume clipping is less successful. Figure 4.15 shows an edge of a clipping volume rendered both with and without the ring artifacts reduction algorithm, using a high sampling rate. Near the edge, which behaves like an isosurface, the slight differences in ray starting position are clearly visible.



(a) (b)

Figure 4.15: The noise becomes clearly visible near an edge of the clipping volume, even when using a high sampling rate.

Although the results of ray casting-based volume rendering on the GPU look very promising, there are still some issues to be resolved before it can be used in real-life applications. The most serious one is the current limitation on the maximum volume size. Currently OpenGL imposes a limit of 128 megabytes to the size of a texture. This restricts the maximum allowed volume data to be of 512x512x256 (or any other distribution of equal size) in dimensions. A solution is to use multiple textures to store the volume data, but this rises the problem of transferring the data to video memory when needed. A cache-aware mechanism is required to minimize the swapping an keep the performance high.

The work of Engel and Weiskopf often mentions a technique called pre-integrated volume rendering [Engel et al., 2001b] and they claim it gives better image quality at a higher performance. The presentation given by these same people mentioned earlier also explains the technique in more detail and gives similar claims on the results. Such optimizations are worth looking at.

4.3 Hardware Versus Software DVR

In the previous sections, two different implementations of ray casting-based volume rendering were discussed, one that uses the CPU and another one that uses the GPU. Due to the difference in target hardware, both methods have different properties.

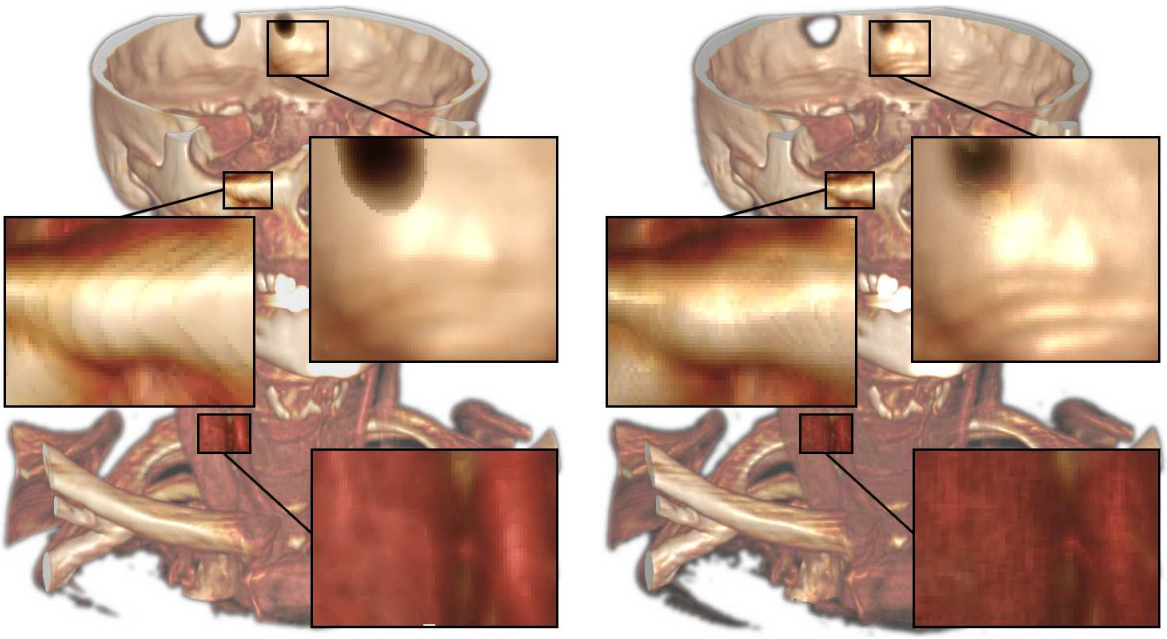
4.3.1 Advantages and Disadvantages

Most of the advantages and disadvantages of both methods have to do with the properties of the hardware platform they use. Software-based volume renderers can usually deal with larger data sets than those that use a GPU, since the amount of host memory is commonly much larger than the amount of video memory. Of course a dataset could be spanned across the host memory and video memory, but swapping the data has an impact on the performance. A large and easily accessible memory is thus an advantage of software-based volume rendering methods.

A GPU is a typical stream-based processor. This means that it is aimed at doing many simple tasks in parallel. While rasterizing, there are many pipelines, each processing a pixel, running in parallel. Current graphics hardware has between 16 and 32 pipelines, but these numbers rapidly increase. The advantage of the GPU-based volume rendering method described in chapter 4 is that this parallelism is used automatically. Software-based volume renderers may use parallelism as well, but currently most common workstations have only one or two CPUs. Also, support for parallelism has to be built inside the software, while it is used automatically by GPU-based volume renderers. This high degree of parallelism is an advantage of GPU-based volume rendering.

The high degree of parallelism of GPU-based volume rendering is also a downside. When all pipelines are filled, computation is started and lasts until all the pipelines have finished computing. Pipelines that are done earlier are stalled until the end of the computation. This means that for a good performance, the work load should be evenly distributed, which increases the complexity of the algorithm.

An advantage of software-based methods over that of hardware-based methods in general is that software offers more flexibility. Although current graphics hardware is more flexible, there are still constraints. Due to the stream-based nature of graphics hardware, there is no interaction possible between pixels, that is output for one pixel cannot be used as input for another. Adaptive interpolations schemes as used by the DirectCaster are for example not possible with current graphics hardware.



(a) (b)

Figure 4.16: Image quality comparison between the DirectCaster (a) and the GPUCaster (b).

4.3.2 Image Quality

A list of properties may give an indication which method is preferable over the other, but in the end it is all about the results. One type of result is the quality of the images that are produced. The visibility of various kinds of artifacts caused by optimizations, precision of computations, sampling strategies, et cetera all determine the quality of the final image. A list of the most noticeable differences between the DirectCaster and the GPUCaster is given here.

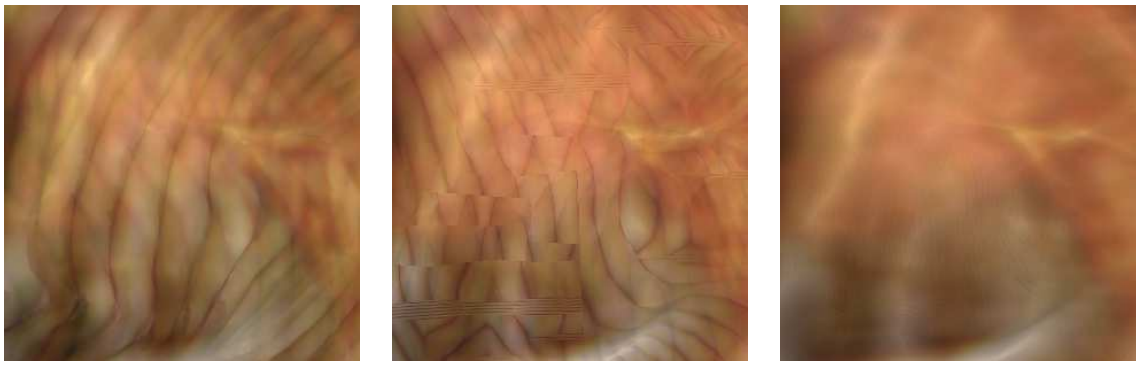
Figure 4.16 shows two renderings of a volume dataset without clipping, one generated by the DirectCaster and the other by the GPUCaster. A few highlights of the images are blown up to show the differences more clearly. The left highlight shows the difference in slicing artifacts. The DirectCaster uses a variable sampling rate, while the GPUCaster applies a constant sampling rate. The differences of these two approaches are shown in the left highlight. In this case the image produced by the DirectCaster shows slicing artifacts whereas the one of the GPUCaster does not. The reverse case may also occur.

The upper right highlight shows a difference in gradient computations. The surface in the image generated by the DirectCaster looks smoother than that of the the GPUCaster. The same holds for the lower right highlight, where a very translucent area is shown. The exact reason for this is unknown, neither is clear which image is preferable.

The lower right highlight also shows that the image generated by the GPUCaster contains more noise than that of the DirectCaster. Both renderes have 16-bit data as input, but the interpolation mechanisms are different. The DirectCaster uses fixed point math for the interpolation, while the GPUCaster uses the interpolation mechanisms of the GPU. The latter mechanism uses 32-bit floating point math to perform the interpolation, but the data is interpolated in its normalized form. To increase the precision

of the GPUCaster, the data is scaled to maximally exploit the possible range of the data, but some interpolation artifacts remain.

Overall the image quality produced by both renderers is good. Both show minor artifacts, but these are of insignificant importance in most practical images. The quality of both renderers is high enough to give accurate visualization of the data.



(a) (b) (c)

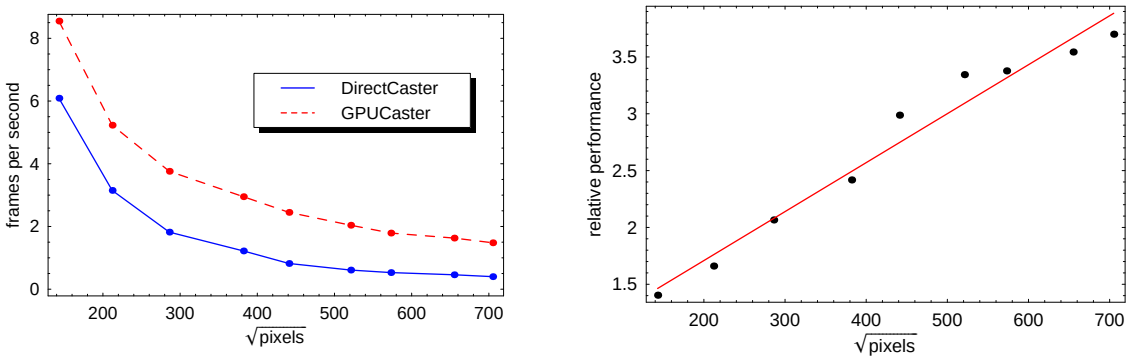
Figure 4.17: Ring artifacts produced by the DirectCaster (a) and the GPUCaster (b). Figure (c) shows the same area rendered without artifacts.

Due to the facts that both volume renderers use different optimizations, similar artifacts may appear differently. Figure 4.17 gives an example of this behavior. Section 4.1.5 mentions that the DirectCaster produces visible ring artifacts in regions of the volume that are mapped to a relatively low opacity. This is shown in figure 4.17 (a). A rendering of the same region with a similar sampling rate produced by the GPUCaster is shown in figure 4.17 (b). The ring artifacts are also clearly visible, but they are heavily distorted. The cause is the optimization that draws the grid of cubes instead of the bounding cube of the volume. Due to this optimizations, the starting points of the rays are at different offsets for each pixel. Since a viewing-aligned sampling strategy is applied, this distorts the ring artifacts. A correct, artifact-free rendering of the same region is shown in figure 4.17 (c). It was produced by the GPUCaster with a sampling rate of 0.4 voxels.

4.3.3 Performance

Although the image quality is comparable, the performance of the two volume renderers is not. Figure 4.18 shows the difference in performance between the DirectCaster and the GPUCaster. The results were obtained on a dual Intel Xeon 2.8 Ghz machine equipped with a nVidia GeForce 6800 GT Ultra. Figure 4.18 (a) shows the individual performance of both renderers at varying resolutions. Along the horizontal axis the square root of the number of pixels is plotted against the number of frames per second the renderer can produce at that resolution on the vertical axis. The overall frame rates produced by both renderers are low, but the GPUCaster is faster than the DirectCaster at all resolutions. The benchmark was performed using a 256^3 sized data set and a transfer function with a medium to high degree of transparency. Increasing the size of the volume data does not affect the performance much for both volume renderers. Using a less transparent transfer function increases the performance of both volume renders approximately equally.

Figure 4.18 (b) shows the performance of the GPUCaster relative to that of the DirectCaster. This graph



(a) (b)

Figure 4.18: The performance of the DirectCaster versus the GPUCaster; (a) shows absolute values while (b) displays the relative performance.

shows interesting behavior, since this relative performance seems to increase linearly at increasing resolution. It is difficult to find out what is causing this behavior, as the performance of both renderers is influenced by many factors. Possible explanations are the interpolation performed by the DirectCaster and the parallelism of the GPUCaster.

With the DirectCaster, the percentage of pixels that may be interpolated decreases as the resolution increases. This is due to the fact that the distance along which a non-interpolated pixel may be used is constant. Thus if the number of pixels increases linearly, the performance of the DirectCaster may decrease more than linearly, since relatively less pixels can be interpolated.

The second possible explanation could be that the average workload of the pipelines of the GPU decreases as the number of pixels increases. Since the number of pipelines remains constant, the number of computations where a pipeline is stalled by another decreases as the number of computations as a whole increases. This can also be an explanation for the fact that the performance of the GPUCaster does not scale linearly with the number of pixels.

These reasons are mere guesses of what may be causing this behavior, but there is a good explanation for the overall better performance of the GPUCaster; GPUs are highly optimized for floating point optimizations and allow for much better parallelism than computations on a CPU.

The GPUCaster was used without the ring-artifacts reduction algorithm enabled and using a step size of 0.4 voxels to minimize the visibility of these artifacts. By using a larger step size in combination with the ring-artifacts reduction algorithm, a trade-off can be made between image quality and performance. At an acceptable image quality, the performance is about double compared to figure 4.18, using a step size of approximately a single voxel. Decreasing the image quality even further gives interactive frame rates.

4.4 Volume Rendering with Volume Clipping

In chapter 1 six requirements are listed that should be met by the implemented prototype. Requirements **R0** through **R3** are met by using a depth-based approach. Requirement **R5** is met by using a rasterization method and requirement **R6** was met by the way the prototype was implemented. Finally,

requirement **R4** was met by using the gradient impregnation technique as described in section 2.4. This technique was successfully integrated in both the DirectCaster and the GPUCaster. Section 2.4 also mentions that a using a step-function as a weighting function may reveal sudden changes in the optical in the final image. It also mentions that this problem can be overcome by using a linear weighting function.

Figure 4.20 shows a region of a clipped volume near the edge of the clipping volume, rendered using different strategies for the gradient impregnation. As can be seen from figures (a) and (d), not using gradient impregnation leads to an incorrect representation of the edge of volume. The edge does not appear to be flat due to an incorrect gradient. Figures (b) and (e) use a step-function over a length of four voxels. This solves the problem with the edge, but leads to several other artifacts. An artifact appearing in general is that the sudden change in optical model will become visible, as is shown in figure (h) and in the right highlight in figure (b). The gradient impregnation layer is clearly visible, because a different gradient is used for the lighting computation.

Other artifacts that appear are specific for each volume renderer and are produced because of the side effects. In figure (b) slicing artifacts appear near a border of the volume at the edge of the clipping volume. In that area, there are few differences in opacity so the DirectCaster applies a low sampling rate. However, due to the sudden change in optical model, the number of samples that use a gradient of the clipping volume differs for each pixel. These artifacts are caused due to similar reasons as the ring artifacts mentioned in section 4.1.2.

The GPUCaster shows similar artifacts, but the slices appear smaller since the GPUCaster uses a higher sampling rate in these areas. The slicing artifacts are influenced by the optimization that draws the grid of cubes instead of the bounding cube of the volume. Since the intersection with the cubes and the clipping volume may vary, there are differences in the starting point of the ray near those areas, giving rise to ring artifacts.

The right column shows that using a linear weighting function results in a correct rendering of the edge without artifacts. There is no sudden change in optical model and the gradient impregnation layer itself is not visible. This difference is clearly visible in figures (h) and (i). Thus using a linear weighting function for gradient impregnation meets requirement **R4** as well.

Figures (k) and (m) demonstrate what happens when the gradient impregnation layer is too thick. The gradient of the clipping volume becomes the dominant gradient throughout the volume. Although the lighting still aids to perceive the shape of the clipping volume, the ability to perceive the spatial structure and orientation of the volume data itself is destroyed. Therefore the gradient impregnation layer should be of appropriate thickness. For most practical application, two to four voxels is enough.

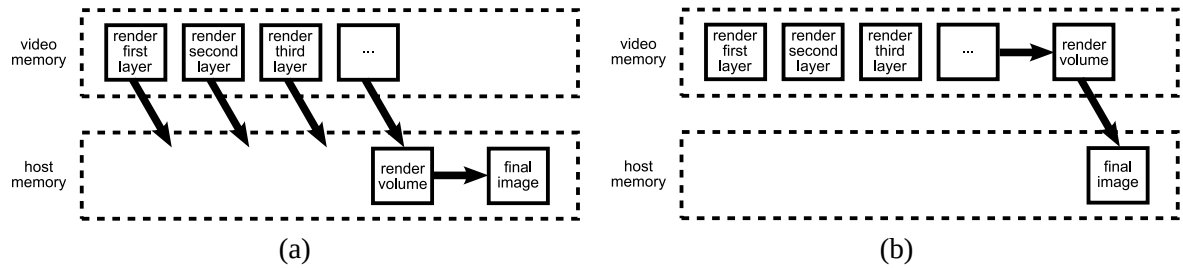


Figure 4.19: An overview of in data transfer requirements for the rasterization method when used with software (a) and hardware (b) based volume rendering.

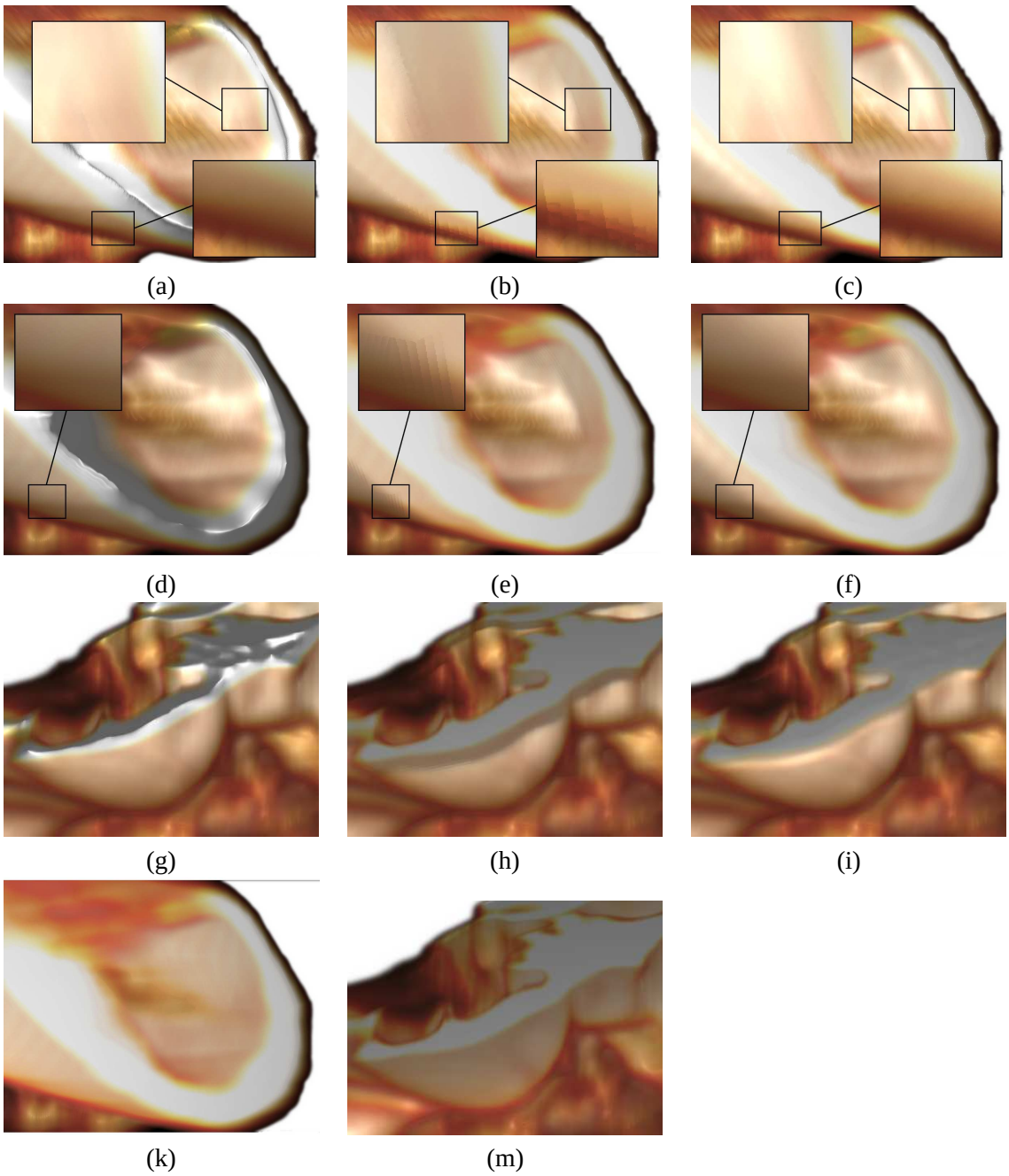


Figure 4.20: Differences between various gradient impregnation functions. The images in the top row are produced by the DirectCaster, the remaining images by the GPUCaster. The images in the first column of the first three rows were rendered without gradient impregnation, the ones in the second column with a step function over a distance of four voxels, while the ones in the third column had a linear impregnation weighting function applied over a distance of also four voxels. The images in the bottom row show the effect of a too thick gradient impregnation layer; the ability to perceive the structure of the volume is destroyed.

For volume clipping, there is a relatively big difference in performance. The rasterization method itself runs at equal speed for both volume renderers, but the software volume renderer requires more data to be transfered from video memory to host memory, which is a slow process. Figure 4.19 (a) shows that a memory transfer is required after each rendering pass of the rasterization method. Increasing the number of segments to be rendered thus rapidly decreases performance. For the hardware based volume renderer, the rasterization method itself requires only memory transfers that are local to the video memory. At the end of the rendering, the final image of the volume rendering needs to be transfered to host memory though, to comply to the VVC architecture. Therefore, a increasing the number of segments has a smaller effect on the performance.

In the resulting volume renderers these differences are hardly noticeable. The reason for this is that the volume rendering itself takes much more time than the rasterization of the clipping volume. The difference in rendering speed of the volume renderers itself also mask out the difference in speed of volume clipping.

Chapter 5
Implementation

The techniques covered in the previous chapters were implemented according to the VVC architecture to meet requirement **R6**; a new VVC driver was created. To allow volume clipping, a few modifications to the VVC were required. This chapter lists these relatively small modifications. A short description of how to use the driver is also given. No source code is discussed in detail.

5.1 Modifications of the VVC

At the start of the project, the VVC did not yet offer support for specifying clipping volumes other than clipping planes. A number of modifications to the VVC were required to make the use of a polygon mesh as clipping volume possible.

5.1.1 Extensions to VVC_Model

Although the original version of the VVC, referring to the version of the VVC in the common source tree during the project, already contained the notion of a polygon mesh, a modification was required to enhance the suitability for accelerated rendering and ease of use. The existing data structure, called VVC_Model, allowed a model to be specified as sets of triangle strips and fans, but not as a list of separate triangles. To prevent forcing the developer to stripify a model prior to using it with the VVC, an extension to VVC_Model was made that allows the use of triangle lists. A triangle list is a list of $3n$ vertices where each three successive vertices specify a single triangle. Hence a triangle lists containing $3n$ vertices defines n triangles.

5.1.2 Extensions to VVC_CutSet

The original version of the VVC also included a data-structure called VVC_Cut that represented a clipping plane. Multiple clipping planes could be combined into a set by using the VVC_CutSet data-structure. In order to allow a VVC_Model to be used as a clipping volume, the VVC_CutSet data-structure was extended to allow the inclusion of a VVC_Model. The modified VVC_CutSet can contain either a single VVC_Model or a set of VVC_Cut objects.

Since all data-structures in the VVC use VVC_Cut only through VVC_CutSet and never directly, no further modifications were required. The name VVC_CutSet is not optimal considering the use of the

modified data-structure, not is this solution architecturally sound. This method did however require the least amount of changes to the VVC, saving more time for research on volume clipping.

5.1.3 Support for Spline Patches

The original project proposal contained a requirement that it should be possible to specify the clipping volume as a set of Bézier contours. As the project progressed, it became clear that the use of Bézier contours required a tessellation to a polygon mesh, which is a different subject than volume rendering. Due to the project planning, some time was spent on the subject at the beginning of the project. A basic description of Bézier curves and patches is given in appendix A.

To demonstrate the possibility to use a set of Bézier contours as a clipping model, a function was added to the VVC that takes a two-dimensional set of points defining a set of Bézier contours as input and gives a VVC_Model as output. The resulting model is a uniform tessellation of the set of Bézier contours, complete with normal vectors. This model can be directly supplied as a clipping volume.

5.2 The VCR Driver

An implementation of the rendering techniques described in this thesis was made as a driver for the VVC. A VVC-driver is software component that performs the rendering of a scene specified by the data-structures of the VVC. The driver developed should be considered a prototype acting as a proof-of-concept; the code is not suitable for use in end-product software.

5.2.1 Driver Options

The various methods for volume clipping and volume rendering were all implemented, which gives the need for a set of parameters to control the rendering. According to the VVC architecture, this can be done by using the options string which will be passed to the driver. Table 5.1 gives an overview of the accepted options and their meaning.

5.2.2 Limitations of the Driver

There are a number of limitations to the driver. These limitations are caused by the strict requirements imposed on the scene by the VCR driver. Relaxing these requirements is not a very challenging task, but it is time-consuming and was therefore not done. A list of requirements that a scene to be rendered by the VCR driver is given below.

1. The output should be a color display;
2. the geometry should be of type VVC_FrameProjection or VVC_ConePyramid;
3. there should be exactly one visual in the scene;
4. the visual should be of the type VVC_ClassSurfaceVolume;

Option string	Domain	Description
VCRclipMode	Probing, Cutting	Selects the method of volume clipping to use.
VCRrenderMode	Ray-Casting, Rasterization	Selects the method to use for computing the depth-structure of the clipping volume. Applies only to the use of the DirectCaster.
VCRvolumeRenderer	DirectCaster, GPUCaster	Selects the volume rendering component to use.
VCRbufferDepth	16, 32	Selects the precision of the floating-point render target used for computing intersection depth and gradient. Applies only to the rasterization method in combination with the DirectCaster.
VCRsegments	$\mathbb{Z}^+$	Selects the number of intersection segments to render. Applies to the rasterization method only.
VCRbackmost	true, false	Specifies whether the back-layer of the back-most segment should be the back-most layer of the entire clipping volume. Applies to the rasterization method only.
VCRstepSize	$\mathbb{R}^+$	Controls the step size used for volume rendering inside the GPUCaster.
VCRERTThreshold	$\mathbb{R}^+$	Controls the early-ray-termination transparency threshold. Applies to the GPUCaster only.
VCRclipIntensity	$\mathbb{R}_{[0,1]}$	Controls the intensity of the overlay of the clipping volume on the volume rendering image. Applies to the GPUCaster only.
VCRinterpolation	$\mathbb{R}_{[1,\infty)}$	Controls the size of the image that is actually rendered. The rendered image is interpolated to the full image using bilinear interpolation. Applies to the GPUCaster only.
VCRblurFactor	$\mathbb{R}^+$	Controls the amount of blurring that is applied to the final rendered image. The use of blurring may reduce the presence of noise. Applies to the GPUCaster only.
VCRringReduction	true, false	Controls the use of the ring artifact reduction algorithm. Applies to the GPUCaster only.

Table 5.1: Options that can be passed to the VCR driver.

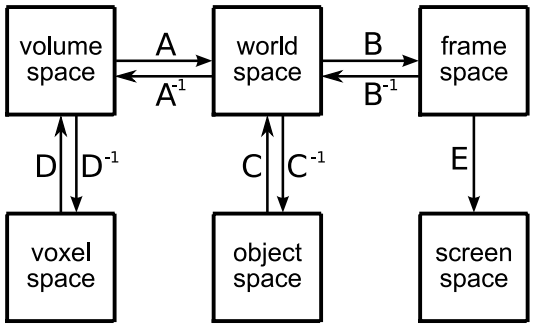


Figure 5.1: Overview of the different spaces and the transformations between then that are defined.

5. the volume contained in the visual should have a grid;
6. the visual should have a cutset of type VVC_ClassCutModelProbe or VVC_ClassCutModelCut;
7. the model of the cutset should have exactly one section with a list of smooth vertices and a grid.

If a scene fails to meet any one of these restrictions, the driver will return with an error message. Note that these restrictions imply the presence of a clipping volume; there currently is no way to turn off clipping. A workaround for this limitation is to specify a clipping volume that embodies the bounding cube of the volume.

5.2.3 Transformation Spaces

To allow the elements in a scene to be independently transformed, most elements have a separate transformation assigned to them. These transformations define how one coordinate system, or space, is positioned within another; they describe a transformation from one space to another. To achieve the correct result, these transformations should be concatenated in the correct order. In the VCR driver, a name is assigned to each of the spaces to structure the transformations. An overview of the transformation schema is given here. Note that the transformation schema is actually specified by the VVC.

Figure 5.1 shows the different spaces that are used in the VCR driver. An arrow between two spaces indicates that a transformation between the two spaces is directly defined by one of the elements in the VVC scene. Transformation A is defined by the grid of the visual. Each grid defines a scaling, rotation and translation, in that order. In volume space, the voxels are cube-shaped and axis aligned. In fact, volume space is very similar to voxel space. Transformation D only defines a translation, because in volume space the origin is in the center of the volume, while in voxel space all voxels have positive, integer coordinates. The use of voxel space is that the coordinates are equal to the array indices in each direction.

Transformation B is defined by the grid of the clipping volume and consists of a scaling, rotation and translation, just like transformation A. The frame or cone geometry defines transformation C. Note that frame space is also commonly known as viewing space. This name was avoided due to the name clashing of the abbreviated form with volume space. The frame geometry defines a translation, rotation and scaling, in that order. Coordinates in frame space can be projected to the screen using the projection transformation E. There is no inverse transformation, since a projection is not invertible.

Chapter 6
Conclusion

Various different techniques have been examined, both for volume clipping and volume rendering. In this chapter a summary of the advantages and disadvantages of each of those techniques is given. Both the image quality and the performance are discussed for each technique, as well as their conformance to the requirements listed in chapter 1. At the end of the chapter some leads for possible further research are offered.

6.1 Ray Casting Versus Rasterization

Two depth-based volume clipping techniques were explored in detail; a ray casting method and a rasterization method. In chapter 2 it was already stated that the ray casting method was not preferable over the rasterization method. This is partially due to the fact that the spatial subdivision data-structures require pre-computation, causing some degree of interactivity with the clipping volume to be lost.

A strong advantage of the rasterization method is that it allows the use of graphics hardware, specifically consumer-level graphics cards, to accelerate the volume clipping. This means that a complex clipping volume can be used while still achieving a very good performance. In fact, for most practical clipping volumes the performance of the rasterization method is much higher than that of the ray casting method. Using the GPU for volume clipping also integrates very well with GPU-based volume rendering techniques.

Advantages of the ray casting method are that there are possibilities for integration software volume rendering techniques that could increase performance. In addition, there is no need for multiple rendering passes or complex layering based solutions. These advantages do not outweigh the advantages of the rasterization method however. The limitation of the rasterization that it requires multiple rendering passes to support concave clipping volumes easy to overcome. For most practical clipping volumes a low number of segments, typically two or three, are enough to give a good image quality.

In conclusion, both methods have their advantages and disadvantages, but the high performance of the rasterization method makes it preferable over the ray casting method. This is based on the assumption that suitable graphics hardware is available.

6.2 Volume Clipping in Software and Hardware

The rasterization method was integrated with both a software and a hardware based volume renderer. The results of volume clipping itself were equal for both volume renderers. The same techniques were applied to remove any artifacts caused by volume clipping. From a performance point of view, the hardware approach takes advantage from the fact that the rasterization method uses graphics hardware too by saving a costly data transfer. However, this is hardly noticeable in practice, since the time spent on the rasterization method is insignificant compared to the time spent on volume rendering.

The hardware volume renderer was developed as a part of this project. It is an implementation of ray casting-based volume renderer on consumer-level graphics hardware. It takes advantages of the new features of current graphics hardware to make this possible. It delivers a higher image quality at a higher performance than the software-based volume renderer. There are some limitations, but further development of the renderer should rid most of them.

6.3 Possible Improvements

The rasterization technique for volume clipping has proved to be an efficient one. A downside is that the end-user should supply the number of segments that should be used for performing volume clipping. Although it has been stated that two or three segments is usually enough, setting the number of segments to a fixed value may give artifacts in situations with a high degree of overlap in the clipping volume. Determining the number of layers to be rendered automatically would be a solution, although doing so is not entirely trivial. It is possible render exactly the number of segments needed for an entirely accurate rendering, but this would have a serious impact on the performance. Often only few pixels are affected by the last few segments. Finding a mechanism to control the accuracy versus image quality would rid the user from specifying this parameter.

The use of graphics hardware allows clipping volumes of very high resolution without a significant impact on performance. There is a strict requirement on them being specified as a polygon mesh. Relaxing this requirement could be done by only affecting the steps before the clipping volume is rendered as a polygon mesh. Of course more surface discretization techniques could be implemented. Interesting options include ones that allow adaptive subdivision of such higher-order surfaces. For example, when looking at the future of GPUs, the upcoming shader model 4.0 offers support for creating new vertices during rendering. This means that the clipping volume could be specified as a set of Bézier patches and that the entire subdivision work is done inside the shaders, allowing adaptive control of detail based on for example depth information. Although these ideas are based on preliminary information, there are many interesting possibilities in this area.

Besides the volume clipping, using a ray casting method to perform volume rendering on the GPU is a very promising technique as well. There currently are no references to this technique being applied to medical volume data before. Both the image quality and the performance are high, suggesting that further research in this area is interesting. There are many possibilities left, both applying existing optimization techniques and finding new ones. The adaptive sampling rate technique as applied in the DirectCaster could lead to an improved image quality and performance for example. To make the technique more usable in real-life applications, the limitations on the size of the volume should be solved. Much work in this area has been done before [Kanus et al., 2003] [Cox et al., 1998] and it is likely that existing techniques can be modified to be applied here. In conclusion, ray casting-based volume rendering on the GPU is still in its infancy and there is much left to explore.

Appendix A
Tessellation of NURBS-Surfaces

A.1 Introduction to Splines

When given a set of points in space that should be connected by lines or surfaces, performing linear interpolation between these points (that is, in the two-dimensional case, connecting the points by straight lines), often does not give a satisfactory result. Often there is a desire for a smooth transition between the points. For This purpose splines can be used. A *spline* is a general term for a curve modeling a smooth transition between a set of points, often called the control points. There is a large amount of types of splines, most of them not being bound to two or three dimensions.

For this particular project, there is an interest in splines for the tessellation of Non-Uniform Rational B-Spline (NURBS) patches that can be used to define the clipping volume. Since this means the sole interest is in the evaluation of the location, gradient and curvature of NURBS surfaces, only a short introduction into splines with a strict focus on NURBS surfaces is given. Besides a general derivation of certain equations, the pre-computation possibilities eventually used in the implementation are also discussed.

A.1.1 B-Splines

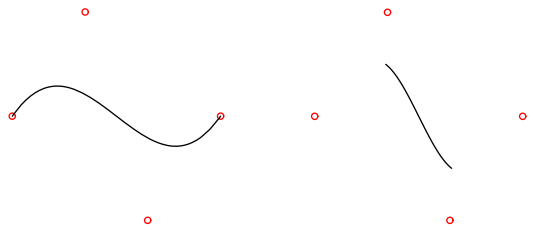
A very general type of spline is the B-Spline. A general B-Spline is defined by a knot vector $T = \{t_i \mid 0 \leq i < m+1\}$ where T is a non-decreasing sequence and $t_i \in [0,1]$ for all i, and a set of control points $P = \{P_i \mid 0 \leq i < n+1\}$. The degree of the B-Spline is defined as $p = m - n - 1$. The general B-Spline curve is now defined as

$$C(t) = \sum_{i=0}^{p} P_i B_{i,p}(t), \tag{A.1}$$

where $B_{i,p}(t)$ is the basis function for the B-Spline. This function is defined as

$$B_{i,0}(t) = \begin{cases} 1 \text{ if } t_i \leq t < t_{i+1} \\ 0 \text{ otherwise} \end{cases} \tag{A.2}$$

$$B_{i,p}(t) = \frac{t - t_i}{t_{i+p} - t_i} B_{i,p-1}(t) + \frac{t_{i+p+1} - t}{t_{i+p+1} - t_{i+1}} B_{i+1,p-1}(t). \tag{A.3}$$



(a) (b)

Figure A.1: An example of an interpolating (a) and an approximating (b) spline.

The B-Spline curve $C(t)$ is only defined for $t \in [t_p, t_{n+1}]$. Note that the control points of the set P can be of any dimension.

There are many classes of B-Splines, most of these are characterized by a special property of the knot vector. For example, the *non-periodic* B-Spline, where the first $p+1$ entries of the knot vector equal the last $p+1$ entries, causes the curve to be *interpolating*, that is pass through the first and last control point. Figure A.1 shows an example of an interpolating and an approximating (non-interpolating) spline curve. The B-Spline is approximating if for example the knot vector is *uniform*, in which case $t_{i+1} - t_i = c$ for all i and some constant c.

The term *cubic spline* refers to a third degree spline with four control points and thus eight knots. The reason that splines are often subjected to these constraints is that cubic splines are the lowest-degree splines that allow continuity up to their second derivative. The continuity of splines is further discussed in section A.1.5. Having splines of a low degree is useful for performance reasons. Creating more complex splines can be achieved by stitching multiple cubic splines together. Since performance is an issue in this project, the types of splines used will be restricted to cubic splines.

The most general form of the B-Spline is the NURBS-curve, which is short for Non-Rational Uniform B-Spline. Besides being *non-uniform* (so any knot vector is valid), it supports weights assigned to the control points, such that the curve is defined as

$$C(t) = \frac{\sum_{i=0}^{p} P_i w_i B_{i,p}(t)}{\sum_{i=0}^{p} w_i B_{i,p}(t)} \tag{A.4}$$

where w_i are the individual weights, defined as the last element of the homogeneous coordinate P_i^w.

A.1.2 Bézier Curves

Bézier curves are a special case of B-Splines, where the knot vector is non-periodic and has no internal knots. Internal knots are the knots t_{p+1} to t_{m-p-1}. In order to obtain a cubic Bézier curve (defined by third degree polynomials), the knot vector should be defined as $T = 0, 0, 0, 0, 1, 1, 1, 1$. This simplification makes the basis function of A.2 rely only on the control points and on t. Removing the recursion, the basis function for a cubic Bézier curve is given in A.5.

$$\begin{pmatrix} B_{0,3}(t) \\ B_{1,3}(t) \\ B_{2,3}(t) \\ B_{3,3}(t) \end{pmatrix} = \begin{pmatrix} 1 - 3t + 3t^2 - t^3 \\ 3t - 6t^2 + 3t^3 \\ 3t^2 - 3t^3 \\ t^3 \end{pmatrix} \tag{A.5}$$

Combining equations A.1 and A.5 gives rise to a matrix notation of the curve definition. This notation is given in equation A.6.

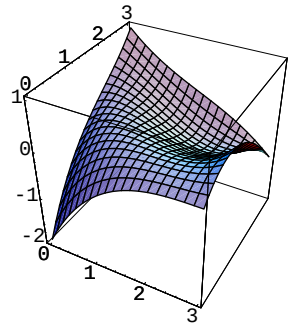
$$C(t) = (P_0, P_1, P_2, P_3) \cdot \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} t^3 \\ t^2 \\ t \\ 1 \end{pmatrix} \tag{A.6}$$

Since the control points are usually fixed during the evaluation of the spline (during for example tessellation), the t is the only variable in such context. This means that the left two matrices of equation A.6 are constant and can be multiplied in a pre-computation step prior to evaluating the spline. This data only needs to be updated when any of the control points change. Note that the middle matrix of equation A.6 is completely constant. In fact, equation A.6 is an instantiation of the general cubic spline equation with the "basis matrix" for Bézier curves, defined below in equation A.7. Section A.1.6 gives an overview of other types of cubic splines that can be computed in the same way, only using a different basis matrix.

$$M_{Bézier} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{A.7}$$

A.1.3 Bézier Surfaces

Bézier surfaces are the same as Bézier curves, but are defined in two dimensions instead of one. A Bézier surface is defined by a set of Bézier curves. For simplicity, a restriction is made to cubic Bézier surfaces, although the generalization to NURBS-surfaces is easy to make. Since there are two dimensions, two knot vectors need to be defined. As with cubic Bézier curves, these are defined as $U = V = \{0, 0, 0, 0, 1, 1, 1, 1\}$. Given 16 control points, the curve is defined as in equation A.8. The figure on the right is an example of a Bézier surface.



$$S(u, v) = \sum_{i=0}^{3} \sum_{j=0}^{3} P_{i,j} B_{j,3}(v) B_{i,3}(u) \tag{A.8}$$

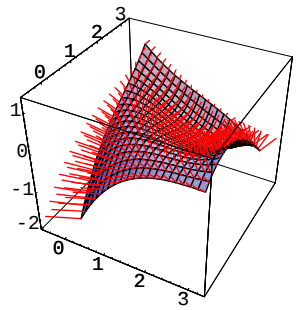
Like one dimensional curves, surfaces can also be expressed using matrix multiplications. Since a surface is constructed by combining four curves in two dimensions, the coefficients of the resulting surface polynomial is the result of three matrix multiplications. To compute the surface, two vectors, one with the powers of u and one with the powers of v, need to be multiplied with this matrix. The

result is shown in equation A.10.

$$M_{CP} = \begin{pmatrix} P_{0,0} & P_{0,1} & P_{0,2} & P_{0,3} \\ P_{1,0} & P_{1,1} & P_{1,2} & P_{1,3} \\ P_{2,0} & P_{2,1} & P_{2,2} & P_{2,3} \\ P_{3,0} & P_{3,1} & P_{3,2} & P_{3,3} \end{pmatrix} \tag{A.9}$$

$$S(u,v) = (u^3, u^2, u, 1) \cdot M_{Bézier} \cdot M_{CP} \cdot (M_{Bézier})^T \cdot \begin{pmatrix} v^3 \\ v^2 \\ v \\ 1 \end{pmatrix} \tag{A.10}$$

A.1.4 Gradient and Curvature of Bézier Surfaces



The gradient of a Bézier surface is useful for determining the normal vectors of the surface, used in for example lighting operations. Likewise, the curvature is useful in for example adaptive tessellation where it may act as an indicator for the required level of detail. Given equation A.8, the gradient and curvature can easily be defined as the first and second derivative of the surface, respectively. However, since there are two variables, there exist two first and second derivatives. As is intuitive, in the two dimensional case, the normal vector of a curve is the vector perpendicular to the gradient vector. This also holds for the general n-dimensional case; the normal vector of an n-dimensional hyper-surface is defined as the cross product of the $n-1$ gradient vectors. A similar reasoning holds for the curvature. This leads to equations A.11. The gradient is defined as $G(u,v)$ and the curvature as $C(u,v)$.

$$G(u,v) = (\frac{\partial}{\partial u}S(u,v)) \times (\frac{\partial}{\partial v}S(u,v)) \tag{A.11}$$

$$C(u,v) = (\frac{\partial^2}{\partial u^2}S(u,v)) \times (\frac{\partial^2}{\partial v^2}S(u,v)) \tag{A.12}$$

Using the matrix notation of the surface equation, equation A.11 can be rewritten such that the same pre-computation data as of the surface can be used to compute the gradient and curvature vectors of the surface by differentiating the vectors of the powers of u and v.

A.1.5 Properties of Splines

As mentioned in section A.1.2, many other types of cubic splines with different properties can be defined by only changing the basis matrix. This section gives an overview of some commonly used other types of cubic splines and gives an overview of their properties.

When categorizing (cubic) splines, there are a number of interesting properties. One property is that of continuity, the very reason to use splines in the first place. A spline can be continuous in various degrees and a special notation exists to denote this. If a spline is C^0 continuous the curve function $C(t)$

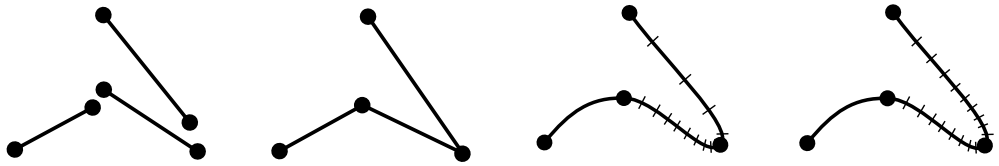


Figure A.2: Splines with varying degrees of continuity; (a) has no continuity, (b) has C^0 continuity, (c) has C^1 continuity and (d) is continuous in C^2.

is continuous. Even linear splines have this property. Splines that are continuous in C^1 are continuous in the first derivative of the curve function. This is true for most cubic splines, but not for the linear spline. This property is the very reason splines were invented. It results in a smooth transition between the control points. The next level of continuity is C^2, which refers to continuity in the second derivative of the curve function. This results in smooth transitions between changes in curvature. Examples of various degrees of continuity are given in figure A.2. An example where C^2 continuity is important is when a point of moves along a spline. If the spline is not C^2 continuous, there might be abrupt changes in the speed with which the point moves along the spline. Figure A.2 gives examples of splines of varying continuity. Note that the continuity in figure A.2 (d) has more to do with the spacing of the input domain along the curve than with the visual appearance. This is indicated with a more gradual increase in space between the markers along the curve compared to figure A.2 (c). It is possible to construct curves that are visually equivalent but differ in C^2 continuity.

A property useful for the construction of splines is that of *local control*. If a spline has local control it means that moving a control point of that spline does not affect the entire curve. Examples of the effects of splines with local control or not are given in the descriptions of Bézier splines and B-Splines in section A.1.6.

The convex hull property expresses if a spline falls within the convex hull of the control points. This is a useful property, as this gives some guarantees about the behavior of the curve. An ideal curve would be one that is interpolating (a property mentioned earlier), is C^2 continuous, has local control and falls within the convex hull of the control points. This, however, is not possible, not even by higher order curves. Fortunately there are good alternatives, of segments of cubic Bézier curves or B-Splines stitched together are most commonly used. Some more details on stitching segments together as well as some properties of different types of splines is discussed in section A.1.6.

A.1.6 Other Types of Splines

Below is a short list of various types of splines along with a list of properties as mentioned earlier for each type of spline. The construction of longer splines by stitching multiple segments together is discussed for the Bézier and B-Spline curves. A generalization for the other curves is considered trivial.

Bézier Spline

The matrix for Bézier curves is given in A.7. A cubic Bézier curve is defined by four control points. When multiple Bézier segments are stitched together to form a bigger spline, there are a number of restrictions to maintain various levels of continuity. To maintain C^0 continuity, the last point of a segment should be equal to the first point of the next segment, as is the case with all types of cubic splines. This follows immediately from the curve definition. In general, let there be two Bézier segments $C_0(t)$ and $C_1(t)$ with $t \in [0,1]$, defined on the sets of control points P_0, P_1, P_2, P_3 and P_3, P_4, P_5, P_6, respectively. The condition for maintaining C^1 continuity is given in equation A.13.

$$C_0'(1) = C_1'(0) \equiv 3(P_3 - P_2) = 3(P_4 - p_3) \equiv P_3 - P_2 = P_4 - P_2 \tag{A.13}$$

This means that the control points neighboring an interpolated control point should be co-linear and equidistant to the interpolated control point to maintain C^1 continuity. If these control points are only co-linear, the gradient has the same direction in the interpolated control point, but not the same magnitude. This means that C^1 continuity is a stronger requirement than having a smooth curve.

Likewise a derivation for maintaining C^2 continuity can be computed. This is given in equation A.14.

$$C_0''(1) = C_1''(0) \equiv 6(P_1 - 2P_2 + P_3) = 6(P_3 - 2P_4 + P_5) \equiv P_1 - 2P_2 + P_3 = P_3 - 2P_4 + P_5 \tag{A.14}$$

This means that for a chain of Bézier segments, all non-interpolated control points, except those of the first and last segments, are fixed if C^2 continuity is desired. For a closed set of n segments, where the last control point of the last segments equals the first control point of the first segment, the C^2 continuity requirement yields a set of $2n$ equations with $2n$ unknowns. This set of equations has a unique solution, being the set of non-interpolated control points. This means that given a set of points, there is exactly one set of Bézier spline segments that interpolates the set of point while having C^2 continuity. The general form for the control points in such a case is

$$\begin{aligned} P_{i-1} &= P_i + \delta(P_{i-3} - P_{i+3}) \\ P_{i+1} &= P_i + \delta(P_{i+3} - P_{i-3}), \end{aligned}$$

where P_{i-3}, P_i and P_{i+3} are successive interpolated control points and P_{i-1} and P_{i+1} are the neighboring non-interpolated control points of interpolated control point P_i. For a set of n (interpolated) control points, the value of δ equals $\frac{1}{n}$. Other values of δ give C^1 continuity, but no C^2 continuity.

Approximating B-Spline

$$M_{B-Spline} = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 0 & 3 & 0 \\ 1 & 4 & 1 & 0 \end{pmatrix} \tag{A.15}$$

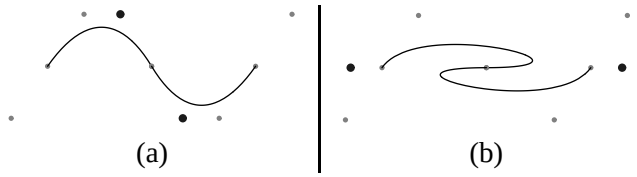


Figure A.3: Local control in Bézier splines; in (b) the third control point is shifted to the right a opposed to (a) and the fifth control point is shifted to maintain C^1 continuity.

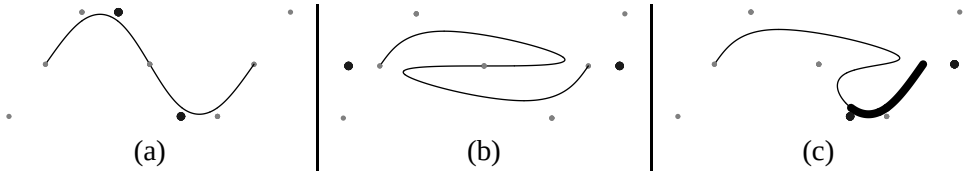


Figure A.4: Local control in B-splines; in (b) the third control point is shifted to the right a opposed to (a) and the fifth control point is shifted as with Bézier splines, while in (c) the fifth control point still is in the same position as in (a), while still maintaining C^2 continuity.

The approximating B-Spline is a spline with C^2 continuity and has local control. It also has the convex hull property, but it is an approximating spline. This means that none of the control points are interpolation, not even the start and end point. The basis matrix for approximating B-Splines is given in equation A.15.

B-Splines have local control, meaning that each point only affects part of the entire curve. This is demonstrated in figure A.4. This property is less useful when multiple segments of cubic splines are used, but it is still a useful property in the construction of curves.

Since both B-Splines and Bézier splines can be formulated using only different basis matrices, every B-Spline curve can be converted into a Bézier curve and vice versa. This can be done by solving the equation $M_{B-Spline} \cdot K = M_{Bézier}$ for the matrix K. This matrix can be used to convert the control points of a B-Spline to those of the equivalent Bézier curve.

Catmull-Rom Spline

$$M_{Catmull-Rom} = \frac{1}{2}\begin{pmatrix} -1 & 3 & -3 & 1 \\ 2 & -5 & 4 & -1 \\ -1 & 0 & 1 & 0 \\ 0 & 2 & 0 & 0 \end{pmatrix} \tag{A.16}$$

The CatMull-Rom spline is an interpolating spline with C^1 continuity. It has local control, but does not have the convex hull property. Note that the latter is incompatible with interpolating all control points and being C^1 continuous. The basis matrix for CatMull-Rom splines is given in equation A.16.

Hermite Spline

$$M_{Hermite} = \frac{1}{6}\begin{pmatrix} 2 & -2 & 1 & 1 \\ -3 & 3 & -2 & -1 \\ 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix} \tag{A.17}$$

The Hermite spline is equal to the Bézier spline, except that the control points it accepts are in a different form. The middle control points, which are approximated by Bézier splines, are not specified as absolute coordinates but as direction vectors. These vectors indicate the derivative of the curve and can be considered the difference vector between two successive control points. The basis matrix for Hermite splines is given in equation A.17.

A.2 Tessellation of Spline Patches

To convert a spline patch into a polygon mesh, the patch should be tessellated. The most primitive form of tessellation is that of uniform tessellation. The domains of both u and v are uniformly divided into a number of intervals. This defines a uniform grid on the two dimensional parameter space of the patch. The spline patch is evaluated at each edge of each interval, giving a rectilinear grid of coordinates. A tessellation of the uniform grid into triangles is trivial. A downside of this uniform tessellation is that it only works well for C^2 continuous spline patches. Also, the level of detail is equal through out the patch. This is not always desirable. A high degree of detail is usually only desired in areas with a high degree of curvature. This can be achieved by using adaptive surface tessellation techniques [Velho et al.,] on a patch uniformly tessellated with a low level of detail or applying surface simplification techniques on a patch uniformly tessellated with a high level of detail.

Bibliography

Amanatides, J. and Woo, A. [1987]. A fast voxel traversal algorithm for ray tracing. In *Proceedings Eurographics 87* (pp. 3–9).

Arvo, J. and Kirk, D. [1987]. Fast ray tracing by ray classification. In *SIGGRAPH '87: Proceedings of the 14th annual conference on Computer graphics and interactive techniques* (pp. 55–64). New York, NY, USA: ACM Press.

Badouel, D. [1990]. An efficient ray-polygon intersection (pp. 390–393).

Bescós, J. O. [2004]. *Isosurface Rendering: a High Performance and Practical Approach*. PhD thesis, Universiteit Twente.

Blinn, J. F. [1977]. Models of light reflection for computer synthesized pictures. In *SIGGRAPH '77: Proceedings of the 4th annual conference on Computer graphics and interactive techniques* (pp. 192–198). New York, NY, USA: ACM Press.

Chen, W., Hua, W., Bao, H. and QunSheng, P. [2003]. Real-time ray casting rendering of volume clipping in medical visualization. *J. Comput. Sci. Technol., 18(6),* 804–814.

Cox, M., Bhandari, N. and Shantz, M. [1998]. Multi-level texture caching for 3d graphics hardware. In *ISCA '98: Proceedings of the 25th annual international symposium on Computer architecture* (pp. 86–97). Washington, DC, USA: IEEE Computer Society.

Diefenbach, P. J. [1996]. *Pipeline Rendering: Interaction and Realism through Hardware-Based Multi-Pass Rendering*. PhD thesis.

Engel, K., Kraus, M. and Ertl, T. [2001a]. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (pp. 9–16). ACM Press.

Engel, K., Kraus, M. and Ertl, T. [2001b]. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *HWWS '01: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (pp. 9–16). New York, NY, USA: ACM Press.

Eric B. Lum, Brett Wilson, K.-L. M. [2004]. High-quality lighting and efficient pre-integration for volume rendering. In *Eurographics/IEEE Symposium on Visualization 2004*. Washington, DC, USA: IEEE Computer Society.

Everitt, C. [2001]. Interactive order-independent transparency.

Hauser, H., Mroz, L., Bischi, G.-I. and Grller, E. [2000]. Two-level volume rendering-fusing mip and dvr. In *VISUALIZATION '00: Proceedings of the 11th IEEE Visualization 2000 Conference (VIS 2000)*. Washington, DC, USA: IEEE Computer Society.

Havran, V. [2000]. *Heuristic Ray Shooting Algorithms*. PhD thesis, Czech Technical University, Prague.

Kanus, U., Wetekam, G. and Hirche, J. [2003]. Voxelcache: a cache-based memory architecture for volume graphics. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware* (pp. 76–83). Aire-la-Ville, Switzerland, Switzerland: Eurographics Association.

Kaufman, A., IX, F. D., Chen, B., Bitter, I., Kreeger, K., Zhang, N., Tang, Q. and Hua, H. [2000]. Real-time volume rendering. *International Journal of Imaging Systems and Technology, special issue on 3D Imaging*.

Kaufman, A. E. [2000]. Introduction to volume graphics.

Kessenich, J., Baldwin, D. and Rost, R. [2004]. *The OpenGL Shading Language*. 3Dlabs, Inc. Ltd. Language version 1.10.

Kniss, J., Kindlmann, G. and Hansen, C. [2001]. Interactive volume rendering using multi-dimensional transfer functions and direct manipulation widgets. In *VIS '01: Proceedings of the conference on Visualization '01* (pp. 255–262). Washington, DC, USA: IEEE Computer Society.

Lacroute, P. and Levoy, M. [1994]. Fast volume rendering using a shear-warp factorization of the viewing transformation. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques* (pp. 451–458). ACM Press.

Levoy, M. [1988]. Display of surfaces from volume data. *IEEE Comput. Graph. Appl., 8(3)*, 29–37.

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

Federation Against Nature [2004]. Realstorm.

NVidia Corporation [2003a]. Nv_float_buffer. June 16, 2003, revision 16.

NVidia Corporation [2003b]. Release notes for nvidia opengl shading language support.

NVidia Corporation [2004]. Nvidia opengl texture formats.

OpenGL Architecture Review Board [2003a]. Gl_arb_fragment_program. August 22, 2003, revision 26.

OpenGL Architecture Review Board [2003b]. Gl_arb_fragment_program_shadow. December 8, 2003, revision 5.

OpenGL Architecture Review Board [2003c]. Gl_arb_vertex_program. August 17, 2003, revision 43.

OpenGL Architecture Review Board [2005]. Gl_ext_framebuffer_object. May 26, 2005, revision 113.

OpenGL Architecture ReviewBoard [1992]. *OpenGL reference manual: the official reference document for OpenGL, release 1*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.

Möller, T. [2000]. Practical analysis of optimized ray-triangle intersection. http://www.cs.lth.se/home/Tomas_Akenine_Moller/raytri/.

Möller, T. and Trumbore, B. [1997]. Fast, minimum storage ray-triangle intersection. *J. Graph. Tools, 2(1)*, 21–28.

Nagy, Z. A. [2003]. Depth-peeling for texture-based volume rendering.

Nielson, G. M. and Hamann, B. [1990]. Techniques for the interactive visualization of volumetric data. In *VIS '90: Proceedings of the 1st conference on Visualization '90* (pp. 45–50). Los Alamitos, CA, USA: IEEE Computer Society Press.

Pfister, H. [1999]. Real-time volume visualization with volumepro.

Pfister, H., Hardenbergh, J., Knittel, J., Lauer, H. and Seiler, L. [1999]. The volumepro real-time ray-casting system. In *SIGGRAPH '99: Proceedings of the 26th annual conference on Computer graphics and interactive techniques* (pp. 251–260). ACM Press/Addison-Wesley Publishing Co.

Phong, B. T. [1975]. Illumination for computer generated pictures. *Commun. ACM, 18(6)*, 311–317.

Ray, H. and Silver, D. [2000]. The race ii engine for real-time volume rendering. In *HWWS '00: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* (pp. 129–136). ACM Press.

Revelles, J., Ureña, C. and Lastra, M. An efficient parametric algorithm for octree traversal.

Roettger, S., Guthe, S., Weiskopf, D., Ertl, T. and Strasser, W. [2003]. Smart hardware-accelerated volume rendering. In *VISSYM '03: Proceedings of the symposium on Data visualisation 2003* (pp. 231–238). Eurographics Association.

Segal, M. and Akeley, K. [2004]. *The OpenGL Graphics System: A Specification*. Silicon Graphics, Inc.

Segura, R. J. and Feito, F. R. [2001]. Algorithms to test ray-triangle intersection; comparative study. *The 9-th International Conference in Central Europe on Computer Graphics*.

Sunday, D. [2001]. Intersections of rays segments planes and triangles in 3d. http://softsurfer.com/Archive/algorithm_0105/algorithm_0105.htm.

Sung, K. [1991]. A dda octree traversal algorithm for ray tracing. In *Eurographics '91* (pp. 73–85).

Tarantino, P. D. [1996]. Parallel direct volume rendering of intersecting curvilinear and unstructured grids using a scan-line algorithm and k-d trees. Master's thesis, University of California.

Tiede, U., Schiemann, T. and Höhne, K. H. [1998]. High quality rendering of attributed volume data. In *VIS '98: Proceedings of the conference on Visualization '98* (pp. 255–262). Los Alamitos, CA, USA: IEEE Computer Society Press.

Velho, L., de Figueiredo, L. H. and Gomes, J. A unified approach for hierarchical adaptive tesselation of surfaces. Technical report, Visgraf Laboratory IMPA-Instituto de Matemática Pura e Aplicada, Estrada Dona Castorina 110, 22460-320 Rio de Janeiro, RJ, Brazil.

Weiskopf, D. [2003]. Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics, 9(3)*, 298–312.

Weiskopf, D., Engel, K. and Ertl, T. [2002]. Volume clipping via per-fragment operations in texture-based volume visualization. In *VIS '02: Proceedings of the conference on Visualization '02* (pp. 93–100). IEEE Computer Society.

Weisstein, E. W. B-spline.

Yagel, R. [2000]. Volume viewing algorithms: Survey. *International Spring School on Visualization*.