Examensarbete
LITH-ITN-MT-EX--06/045--SE

CPU-Based Volume Rendering of Large Medical Data Sets with Level-Set Clipping

Ulf Lindgren
Olof Rehnström

2006-10-19

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköpings Universitet
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings Universitet
601 74 Norrköping

LITH-ITN-MT-EX--06/045--SE

CPU-Based Volume Rendering of Large Medical Data Sets with Level-Set Clipping

Examensarbete utfört i medieteknik
vid Linköpings Tekniska Högskola, Campus Norrköping

Ulf Lindgren
Olof Rehnström

Handledare Ken Museth
Examinator Ken Museth

Norrköping 2006-10-19

Avdelning, Institution
Division, Department

Institutionen för teknik och naturvetenskap

Department of Science and Technology

Datum
Date

2006-10-19

Språk
Language

☐ Svenska/Swedish
☒ Engelska/English

☐ _____

Rapporttyp
Report category

Examensarbete
☐ B-uppsats
☐ C-uppsats
☒ D-uppsats

☐ _____

ISBN

ISRN **LITH-ITN-MT-EX--06/045--SE**

Serietitel och serienummer
Title of series, numbering

ISSN

URL för elektronisk version

Titel
Title

CPU-Based Volume Rendering of Large Medical Data Sets with Level-Set Clipping

Författare
Author

Ulf Lindgren, Olof Rehnström

Sammanfattning
Abstract

Volume rendering provides means of efficiently visualizing volumetric scalar data in three dimensions. It is a growing field of research with many applications, one of the most prominent being medical visualization. For medical applications, very large data sets are available that create a high demand on computational speed and memory efficiency. In this work, we present and implement a framework for volume rendering of large medical data sets in software. In addition, we present techniques for acceleration and handling of these large data sets.

Another challenge in volume rendering lays in the task of extracting areas of interest in the volume. Transfer functions, mapping sample density to colour and opacity values provide one means for this, but it does not account for tissues of the same density occluding each other. Clipping, which is based on the position rather than the density value of a sample, can therefore be used. For this project, the possibility of using level set segmentation data as clipping geometry in volume rendering was examined. Techniques for high quality clipping and acceleration possibilities are presented. Finally, we present the results of this and discuss further integration of the two techniques.

Nyckelord
Keyword

Volume rendering, level sets

ABSTRACT

Volume rendering provides means of efficiently visualizing volumetric scalar data in three dimensions. It is a growing field of research with many applications, one of the most prominent being medical visualization. For medical applications, very large data sets are available that create a high demand on computational speed and memory efficiency. In this work, we present and implement a framework for volume rendering of large medical data sets in software. In addition, we present techniques for acceleration and handling of these large data sets.

Another challenge in volume rendering lays in the task of extracting areas of interest in the volume. Transfer functions, mapping sample density to colour and opacity values provide one means for this, but it does not account for tissues of the same density occluding each other. Clipping, which is based on the position rather than the density value of a sample, can therefore be used. For this project, the possibility of using level set segmentation data as clipping geometry in volume rendering was examined. Techniques for high quality clipping and acceleration possibilities are presented. Finally, we present the results of this and discuss further integration of the two techniques.

ACKNOWLEDGEMENTS

We would like to thank our supervisor, Professor Ken Museth, for his guidance and help. Moreover, we would also like to express gratitude towards Gunnar Johansson for helping us with the implementation of the level set clipping and providing us with segmentation data. Finally we would like to show our appreciation to Michael Bang Nielsen for giving us access to the DT-Grid, without it this work would not have been possible.

TABLE OF CONTENTS

1 INTRODUCTION

1.1 BACKGROUND

The method of volume rendering has been the subject of research for over two decades, with applications such as manufacturing, geology and medicine. The general aim is to visualize volumetric scalar data in such a way that it can be explored and analyzed efficiently in three dimensions. A mayor obstacle for development and usage of volume rendering has been its inherently large demand of computational power and memory efficiency. However, due to the ever increasing performance of processors and memory together with the advent of specialized graphics hardware, this obstacle is constantly being reduced. Computational speed is still an issue, and as the resolution of data acquisition increases, applications must be adept in handling very large data files.

Another challenge in dealing with volumetric data is how to extract regions of interest, and to separate different types of data from another. This is known as segmentation. The use of level sets has been shown to be efficient for automatic or semi-automatic segmentation of volume data. The level set representation of segmented regions can then be visualized by converting it into a triangular mesh before rendering. In this thesis, the combination of level set segmentation data and direct volume rendering is explored.

This work is done at The Graphics Group, a research group at Linköping University dedicated to computer graphics and visualization. Fields of research include modeling, real-time rendering, volume graphics, medical segmentation, point-based rendering, computational fluid dynamics and level set methods. The Graphic Group Library (GGL) is an internal library of graphics functions and classes used for research. One purpose of this thesis was to provide GGL with a generic and extendable volume rendering framework, to be used and further developed in future research projects.

1.2 AIM

The aim of this thesis is two-fold. First, it is to present an efficient method for visualization of large volumetric medical data sets in software. Second, to investigate the possibilities of utilizing level set representations of segmentation data as a means of extracting regions of interest from the volume data, i.e. clipping.

1.3 METHOD

The work on this thesis has been structured as follows. First, the general techniques of volume rendering, with focus on raycasting, is reviewed. The next step was to develop a basic application for volume rendering. After considering a number of available APIs for volume rendering, it was decided to write the application from scratch. The reason for this was to ensure full control of the system and to allow for the integration of a level set

representation. Once the basic algorithms for raycasting had been implemented, the research and employment of different acceleration techniques had to be performed in order to decrease the rendering times. In a later stage, the system was further modified to allow for large data sets to be handled efficiently. This objective brought on a slight change of focus on the thesis work, at the expense of the level set part of the aim. The final step consisted of integrating a level set clipping into the visualization. For an efficient level set representation, classes from the Graphic Group Library were used.

1.4 THESIS LAYOUT

This thesis is divided in two parts. In the first part, previous work and the theory of fundamental techniques will be described. Chapter 2 presents an overview of volume rendering, chapter 3 describes techniques for volume rendering of large data sets and chapter 4 gives an introduction to level sets.

In the second part, we present the main contributions and results of this thesis. Chapter 5 describes how level set segmentation and volume rendering can be combined. In chapter 6 we describe our implementation in terms of features and system architecture. Chapter 7 presents results and illustrating examples. Finally, in chapter 8 we discuss the results and future work.

1.5 PREREQUISITS

To be able to fully appreciate this thesis, basic knowledge of computer graphics, linear algebra and object-oriented programming is required. However, we have also tried to adapt the text for more general readers.

PART ONE

PREVIOUS WORK

2 VOLUME RENDERING

This chapter will describe the visualization technique of volume rendering, mainly in the context of medical applications. After a brief overview of the subject, principal techniques such as interpolation, gradient estimation and shading will be described.

2.1 OVERVIEW

In medical imaging, techniques such as CT (Computed Tomography) and MRI (Magnetic Resonance Imaging) provides high resolution three-dimensional scalar data which are used to visualize the internal structures of a subject. Both scanning techniques produce a scalar density measurement of the internal tissue. For diagnostics, the volumetric data is traditionally displayed as a number of two-dimensional greyscale slices, where the image intensity level corresponds to tissue density. However, new acquisition hardware provides increasingly high resolution data, and hence more images, which cannot be evaluated efficiently in traditional ways. Therefore, improved techniques for better visualization of volumetric data have been a topic of research for many years.

A common way to display volumetric data is to approximate a surface within a volume by geometric primitives and rendering them with hardware-supported algorithms, e.g. marching cubes [34]. These methods have two major drawbacks. First, converting volume data into geometric primitives is only an approximation of the surface. Second, a great deal of information from the volume data is lost by only using a surface representation.

For these reasons, techniques that can visualize volume data without any intermediate representation are often preferable. Such techniques are referred to as direct volume rendering, or DVR for short. Its main advantage compared to traditional surface rendering is the ability to visualize internal, often fuzzy, structures, in addition to surfaces. However, the requirements it poses on CPU speed and memory efficiency are very high. Nonetheless, volume rendering has become a widely appreciated technique with applications not only in medicine, but also in fields such as molecular science and geology.

2.1.1 APPROACHES TO VOLUME RENDERING

Within the field of volume visualization three distinct branches has evolved: techniques that utilize consumer graphics hardware, specialized hardware systems and CPU-based solutions.

GPU IMPLEMENTATIONS

Implementations relying on consumer graphics hardware have evolved rapidly in recent years. The reason for this is the development of programmable GPUs on commodity graphics card. Using so called vertex- and pixel shaders in various 3D graphics APIs,

such as the OpenGL Shader Language [9], the DirectX-based High Level Shader Language [10] and NVIDIA's Cg language [11], solutions now exist that can produce high-quality renderings at interactive frame-rates. For further reference on hardware volume rendering refer to [5, 12].

Even though these have proven to be very powerful, they suffer from several drawbacks. One is their lack to provide for complex techniques needed in advanced visualization systems, such as filtering and segmentation. If these functions are not supported by the hardware, they have to be performed on the CPU, and data then have to be transferred back to the hardware. The transfer is very time-consuming and causes a considerable overhead. Another disadvantage is their inability to handle large data sets due to the rather limited internal memory of current graphics cards. Furthermore, these systems often depend on manufacturer specific features and drivers, which for example impair portability.

SPECIAL PURPOSE HARDWARE

Before the introduction of programmable graphics cards, the wish for interactive volume rendering led to the development of powerful specialized hardware systems. Several such systems have been developed [3, 13, 14, 15, 16], many being able to render large data-sets with impressive quality and frame-rates. The most prominent, the Volume Pro board, can render a 512^3 data set at 30 frames per second [3]. However, the high price of these systems is a major drawback, limiting extensive use.

CPU BASED SOLUTIONS

The main advantage of CPU-based solutions is their high flexibility and independence of specialized hardware functions. However, both memory and processing bandwidth are limited. Even though numerous acceleration techniques exist, attaining high quality images at interactive frame rates when rendering large data-sets is not possible at present. This is of major concern due to the ever increasing dimensions of acquired data sets. Parallel architectures can account for this using large CPU clusters, but that is a large-scale solution. However, new acceleration techniques have been proposed that allows large data sets to be rendered on a single commodity PC [17, 18]. See chapter 3 for a more detailed description.

2.1.2 VOLUME RENDERING ALGORITHMS

DVR algorithm can be divided into three sub-categories of techniques; image-order, object-order or domain-based [5].

Image-order techniques, also known as raycasting, cast rays from the image pixels through the volume. The volume is re-sampled at equidistant positions along the ray and gradient estimation, shading and compositing is performed to produce a final pixel colour.

In object-order techniques, the rendering pipeline is reversed. Every sample is mapped onto a region on the image plane, and thereby updating the corresponding pixel value according to a specified function. A common name for these methods is splattig [5].

Domain-based volume rendering transforms the volume data into another domain, such as the frequency domain, in order to visualize the information.

Also, since each method has its own specific advantages a number of hybrid techniques has been developed that tries to combine the various benefits, the most well known is probably the shear-warp algorithm developed by Lacroute and Levoy [20]. Even though being a very fast algorithm the image quality is not acceptable for clinical use.

2.2 PRINCIPAL TECHNIQUES

Before describing the direct volume rendering process more in-depth, three central techniques must be described; interpolation, gradient estimation and shading.

2.2.1 INTERPOLATION

A volume data-set only defines the sampled function at discrete points in space. If values between these points are required, interpolation has to be carried out. A vast number of interpolation functions exist; we will mainly focus on two of them: zero and first order interpolation.

Zero-order, or nearest neighbour, interpolation is the simplest form of interpolation. The function simply takes the value at the grid point closest to the sample point. This kind of interpolation corresponds to a box filter, i.e. there is a region of constant value around each sample point, and generally gives poor visual results. Especially noticeable are the characteristic jagged edges and the method is seldom used in applications where image quality is of high concern.

First-order interpolation, or linear interpolation, produces a more accurate estimation. Its 2D and 3D variants are often referred to as bi-linear and tri-linear interpolation respectively.

For a comparison of the two methods please see fig. 1. Note that the jagged edges are no longer visible when using linear interpolation. However, this function has discontinuous derivatives at cell boundaries, which can lead to visible aliasing when the samples changes rapidly between cells.

There also exist higher order interpolation functions, e.g. second order interpolation algorithms like the cardinal spline function [5]. These methods require more computational effort, and are thus more seldom used in real-time applications. In practice, linear filtering often gives satisfactory results for most applications since it offers an acceptable trade-off between quality and cost. However, if the discrete function already has been sampled at a high frequency, nearest neighbour interpolation can sometimes be adequate.

It should be noted that many interpolation techniques tend to smooth or low-pass the original data and such methods can as a result not restore sharp edges that may have existed in the original data. If required, non-linear filters can be used.

Moreover, proper pre-filtering has to be performed whenever a signal is sampled below its Nyquist limit, i.e. twice that of the highest frequency of the signal. If not considered, aliasing artefacts will occur. It is important that such filtering is performed

prior to interpolation, otherwise the high frequencies causing the aliasing will remain and be included in the interpolation.

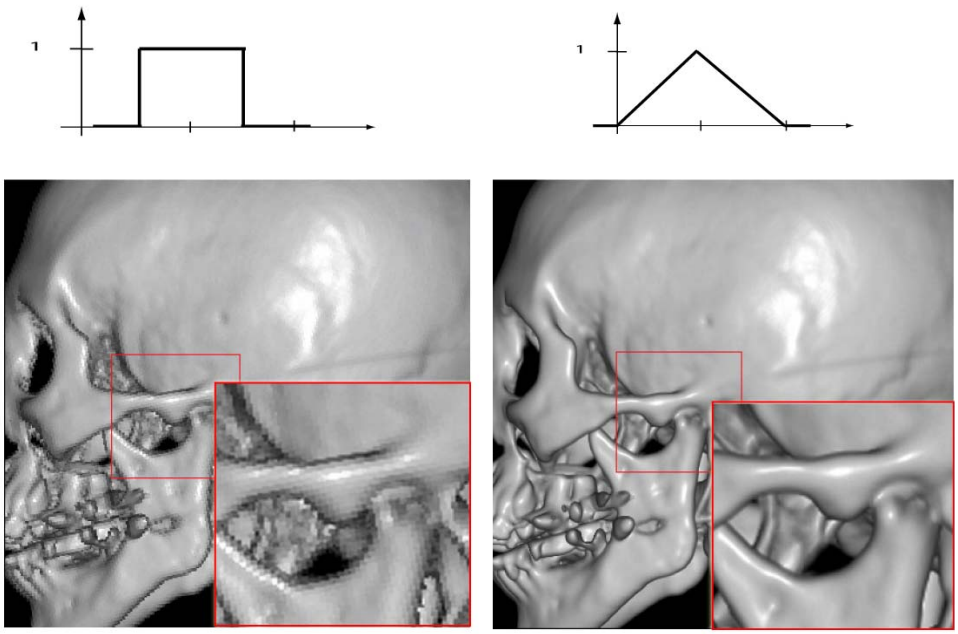


Figure 1 - *Rendering using nearest neighbor interpolation (left) and linear interpolation (right), together with corresponding interpolation function (top).*

2.2.2 GRADIENT ESTIMATION

Gradients are very useful in direct volume rendering since they provide important information of the rate of change of the volume data. They are primarily used in shading to estimate normals and in classification to enhance boundary interfaces.

Two basic approaches exist to estimate the gradient in volume data: image-space and object-space methods. Simply put, image based techniques use the projected 2D-image to approximate the gradient and object-space methods estimate the gradient by using the 3D neighbourhood of the given voxel. Generally the latter gives better results but is also more computationally expensive. Since the gradients are used for shading, which is very sensitive to errors a good gradient estimation is of high importance.

Object-space gradient estimation algorithms all employ some type of filter kernel to approximate a voxel gradient using its neighbours. A widely used method is the central-difference scheme where the gradient is estimated as:

$$\nabla f(x_i, y_j, z_k) = \left(\frac{f(x_{i+1}, y_j, z_k) - f(x_{i-1}, y_j, z_k)}{2\Delta x}, \frac{(f(x_i, y_{j+1}, z_k) - f(x_i, y_{j-1}, z_k)}{2\Delta y}, \frac{f(x_i, y_j, z_{k+1}) - f(x_i, y_j, z_{k-1})}{2\Delta z} \right)$$

Methods using a larger neighbourhood produce a more accurate approximation at the cost of computational effort. However, such methods can have other advantages. For instance, in [21] Neumann et al. presents a scheme that performs gradient estimation and filtering at the same time using linear regression. This makes pre-filtering redundant, considerably reducing the pre-processing stage. Once the voxel gradients are computed, any of the interpolation schemes described in section 2.2.1 can be used to acquire the gradient at an off-grid sample point.

Gradient estimation can be done either on the fly, i.e. continuously during run-time, or in a pre-processing stage and stored for later retrieval. The latter reduce rendering times at the expense of memory cost. Since the gradients often require 2-4 times the amount of memory as the volume itself, this memory consumption cannot be neglected, especially not in a large data context. By instead calculating the gradients on the fly, unnecessary memory usage is avoided, but it requires a great deal of additional calculations to be made during rendering.

2.2.3 SHADING

As mentioned, volume rendering is a method for showing internal structures, where in reality no light is present. However, a lighting model is required for the user to appreciate the three-dimensional shape of structures inside the volumetric data. For example, the standard algorithm based on the Phong illumination model [22], can be employed.

$$R(s) = k_a C_a + k_d C_l C_0(s)(N(s) \bullet L(s)) + k_s C_l (N(s) \bullet H(s))^n$$

Where,
s = the sample point
k_a, k_d, k_s = ambient, diffuse and specular material coefficients
C_a = ambient colour
C_l = colour of light source
C_0 = colour of the object determined by the density-colour mapping function
$N(s)$ = the surface normal vector at point s, determined by the normalized gradient
$L(s)$ = normalized vector in direction of light source
$H(s)$ = halfway-vector, i.e. the normalized vector in direction of maximum highlight, calculated using view and light directions $H = (V + L)/2|V + L|$
$V(s)$ = the view vector
n = Phong exponent, used to approximate highlight

The term N(s)·H(s) is in fact an approximation of the original term R·V, to minimize rendering time, where R is the direction of the reflected light and V is the direction of the viewer.

To further reduce computational effort, the term N(s)·H(s) can be replaced with N(s)·L to save an expensive dot product calculation. This adjustment means that the specular highlight will always be at a maximum in the direction of the reflected light. This simplified model is acceptable due to the fact that light in this case is only an imagined presence to enhance the visualization of the data, rather than a physical factor to be modelled.

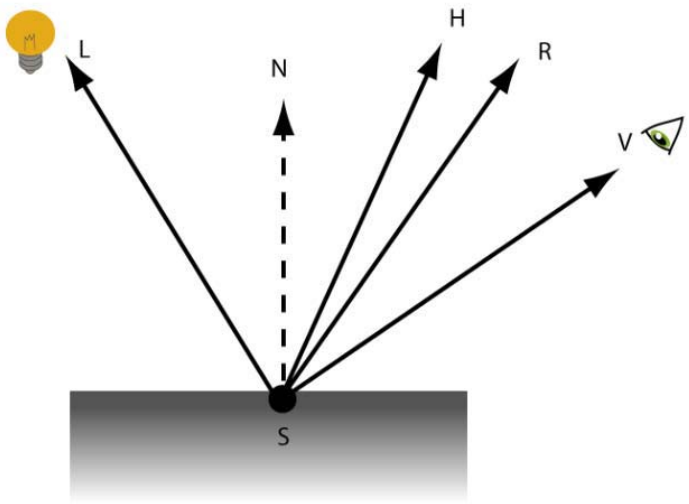


Figure 2 *- Vectors used for shading. L is the direction of the light source, N is the surface normal at sampling point S. V is the viewing direction and R is the direction of the reflected light, often approximated by the so called halfway vector H.*

2.3 RAYCASTING

Raycasting is probably the most popular direct volume rendering method. This is mainly due to its excellent image quality, ease of both understanding and implementation. Raycasting is an image order technique, as described in section 2.1, where rays are cast from every pixel in the image plane through the volume data to determine a pixel colour. A standard ray traversal pipeline for raycasting is illustrated in figure 3.

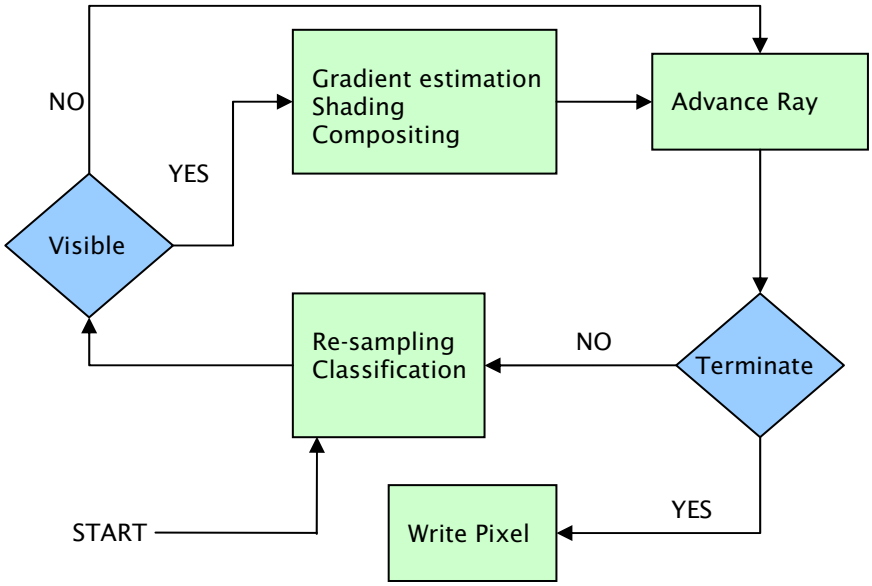


Figure 3 – *Simple ray traversal pipeline*

The principle drawback of the technique is its rendering cost. Since all voxels participate in the generation of each image, rendering times increase linearly with the size of the data set. Much research has been devoted to accelerating this process, both in software and hardware solutions. Acceleration techniques will be further described in section 2.3.4 and 3.3.

2.3.1 PROJECTION TYPES

There are several ways of determining the pixel colour of a ray, commonly referred to as projection types. These methods all have their own benefits and field of use. Many operations are common for all projection types; they all sample the grid at equidistant locations along their paths, and they all obtain sample values via interpolation. However, the modes vary in the manner the samples are composite to acquire the pixel colour. Below follows a simple explanation of the three most common projection types. For image examples please refer to fig 4.

X-RAY PROJECTION

In the simplest of the three methods, x-ray projection, the pixel colour is given by the normalized sum of all samples. This gives rise to a typical x-ray image, except that it exists in three dimensions and can be viewed from any angle.

MAXIMUM INTENSITY PROJECTION – MIP

In the second method, maximum intensity projection or MIP, the pixel colour is set to the maximum sample value along each ray. It is often used in combination with radioactive agents, injected into the blood prior to scanning, in order to visualize blood vessels which otherwise can be hard to detect.

TRANSPARENCY PROJECTION

Transparency projection, also known as full volume rendering, is probably the most interesting one since it can generate semi-transparent 3D images of the volume data. Here, the interpolated samples are used to simulate the transportation of light through the medium.

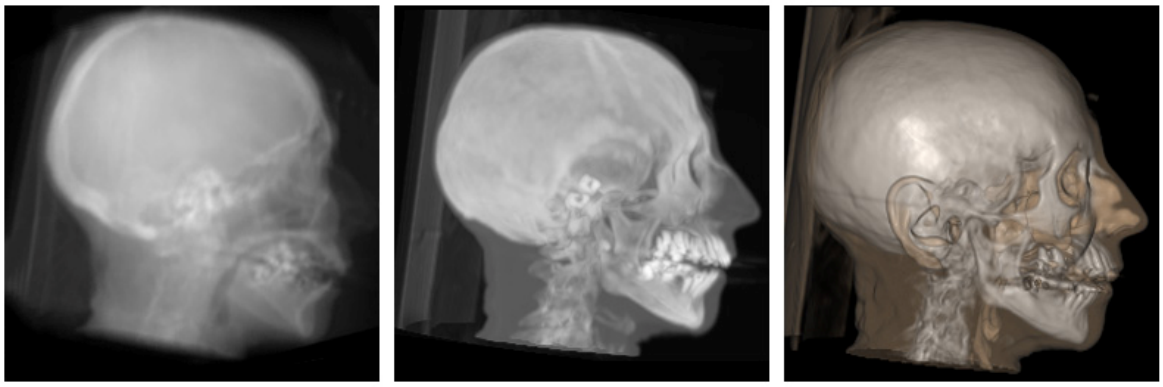


Fig 4 *- Projection types. From left to right: X-Ray-, Maximum Intensity- and Transparency Projection.*

2.3.2 THE VOLUME RENDERING INTEGRAL

The task of direct volume rendering is to simulate the transportation of light through a semi-transparent medium. This can be achieved by using an optical model. If only local lighting effects are considered, simplifications can be made to allow for interactive volume rendering. In addition, effects like shadowing and multiple scattering do not always make visualizations clearer.

One such a model [48], can be created by considering a volume containing particles of various densities. When light rays enter the volume, some pass right through it while others get blocked by particles. The attenuation coefficient, τ , expresses the amount of incoming light that is absorbed per length unit.

Rays might also originate from particles, e.g. from emission or reflection. These particles emit light with intensity L, expressing the luminance of the volume. Given these parameters, the intensity behavior of a ray of light, passing through the volume can be expressed by the following differential equation:

$$\frac{dI(s)}{ds} = L(s)\tau(s) - I(s)\tau(s) \tag{1}$$

The change in intensity of a light ray, I(s), as it passes through the volume is equal to the light emitted at s minus the amount of incoming light that is attenuated. Equation 1 can be solved as follows:

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

$$\Longleftrightarrow \quad \frac{dI}{ds} + I(s)\tau(s) = L(s)\tau(s)$$

$$\Longleftrightarrow \quad \frac{dI}{ds} e^{\int_0^s \tau(t)dt} + I(s)\tau(s)e^{\int_0^s \tau(t)dt} = L(s)\tau(s)e^{\int_0^s \tau(t)dt}$$

$$\Longleftrightarrow \quad \frac{d}{ds}(I(s))e^{\int_0^s \tau(t)dt} + I(s)\frac{d}{ds}e^{\int_0^s \tau(t)dt} = L(s)\tau(s)e^{\int_0^s \tau(t)dt}$$

$$\Longleftrightarrow \quad \frac{d}{ds}(I(s)e^{\int_0^s \tau(t)dt}) = L(s)\tau(s)e^{\int_0^s \tau(t)dt}$$

The equation is then integrated along the ray, from s = 0 at the back of the volume to s= D at the eye:

$$I(D)e^{\int_0^D \tau(t)dt} - I_0 = \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt} ds$$

$$\Longleftrightarrow \quad I(D) = I_0 e^{-\int_0^D \tau(t)dt} + \int_0^D L(s)\tau(s)e^{-\int_s^D \tau(t)dt} ds \qquad (2)$$

Equation 2 is known as the volume rendering integral. The first term determines the amount of incoming light, I_0, reaching the eye[1]. The second term adds the amount of light emitted at each point along the ray, considering the attenuation from each point to the end of the volume. If dealing with non-emissive media, *L(s)* can be calculated using the standard illumination model, described in section 2.2.3. Furthermore, $\tau(s)$ can be expressed by the mass density value at the sample point.

The volume rendering integral cannot be solved analytically in an efficient way [5], but has to be numerically approximated. This can be done using a discrete Riemann sum, where each ray interpolates a number of discrete samples at a distance of *⊿s:*

$$I(D) = \sum_{i=0}^{\frac{D}{\Delta s}-1} \left[L(i\Delta s)\tau(i\Delta s)\Delta s \prod_{j=0}^{i-1} e^{-\tau(j\Delta s)\Delta s} \right] \qquad (3)$$

In order to make the computation more efficient, further approximations can be done. Transparency, *t(i⊿s),* is defined as $e^{-\tau(i\Delta s)\Delta s}$ assuming values in the range [0,1], and opacity, *a(i⊿s),* as *1 - t(i⊿s)*. The exponential density term can then be approximated by the first two terms of its Taylor expansion which gives:

$$t(\mathrm{i}\Delta\mathrm{s}) = e^{-\tau(i\Delta s)\Delta s} \approx 1 - \tau(i\Delta s)\Delta s$$

$$\Rightarrow \quad \tau(i\Delta s)\Delta s \approx 1 - t(i\Delta s) = \alpha(i\Delta s)$$

$$\Rightarrow \quad I(D) = \sum_{i=0}^{\frac{D}{\Delta s}-1} \left[L(i\Delta s)\tau(i\Delta s) \prod_{j=0}^{i-1} (1 - \alpha(j\Delta s)) \right] \qquad (4)$$

[1] From here on, we will conider a model where no light enters the volume from behind, i.e. I_o=0.

Equation 4 is known as the compositing equation and gives us the recursive front-to-back compositing equations:

$$\left.\begin{aligned} c_{new} &= c_{old} + C(i\Delta s)\alpha(i\Delta s)(i - \alpha_{old}) \\ \alpha_{new} &= \alpha_{old} + \alpha(i\Delta s)(1 - \alpha_{old}) \end{aligned}\right\} \quad (5)$$

Where,
c = the accumulated colour
α = the accumulated opacity
C(iΔs) = the sample colour
α(iΔs) = the sample opacity

To summarize, each pixel colour, c, is determined by computing a colour C and opacity α for each sample along the ray and compositing their product with the accumulated opacity of earlier samples. Alternatively, equation (4) can be implemented in a back-to front manner, which is less commonly used due to the fact that this mode cannot utilize early ray termination (see section 2.3.4).

The technique used in equation (4) is sometimes referred to as pre-classification, meaning that colours and opacities are calculated from voxel densities before interpolation. This scheme does not allow for high frequency detail in the transfer function, often resulting in somewhat blurry images, especially during magnification. To account for this another model, post-classification, can be employed:

$$I(D) = \sum_{i=0}^{\frac{D}{\Delta s}-1} \left[L(f(i\Delta s), g(i\Delta s))\alpha(f(i\Delta s)) \prod_{j=0}^{i-1} (1 - \alpha(f(j\Delta s))) \right] \quad (6)$$

Where,
f(iΔs) = the interpolated sample value
g(iΔs) = is the interpolated sample point gradient
C= the colour transfer function
α = the opacity transfer function

This method first interpolates the voxel density values and then maps the result to colours and opacities. This gives rise to a new set of recursive implementation algorithms:

$$\left.\begin{aligned} c &= c + C(f(i\Delta s), g(i\Delta s))\alpha(f(i\Delta s))(i - \alpha) \\ \alpha &= \alpha + \alpha(f(i\Delta s))(1 - \alpha) \end{aligned}\right\} \quad (7)$$

2.3.3 CLASSIFICATION AND TRANSFER FUNCTIONS

In order to distinguish certain features in the data, volume data classification is used. Essentially, classification is the process of assigning colour and opacity values to each sample based on its mass density value. The raw density data values are always used, but the gradients, and even second-order derivatives, can also be utilized in the mapping to give additional information about edges etc.

Classification can be performed on the fly, via some classification function as in [1]. More frequently however, the mapping is performed using look-up tables known as transfer functions.

The reason for using transfer functions is two-fold. First, volume rendering is a very computationally expensive process, and the cost of classifying every sample is considerably smaller when using a look-up table instead of performing calculations on the fly. Second, transfer functions allows for a more intuitive classification process, either in a pre-processing stage or at runtime. A transfer function editor usually allows the user to graphically edit curves in a histogram space, often consisting of both volume data and its first and second derivatives. Figure 5 shows a modern transfer function editor. The main focus of research in this field is the design of such multi-dimensional transfer functions and editors.

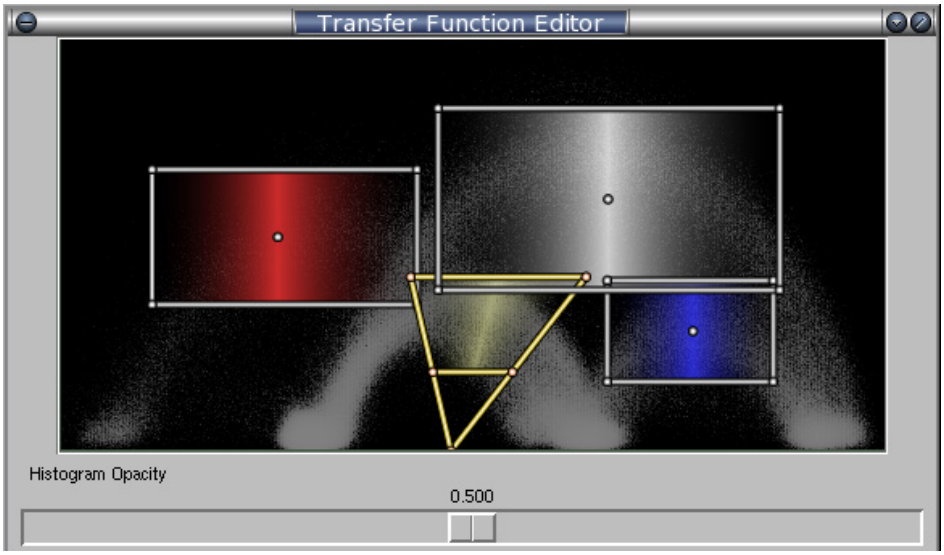


Fig 5 *- A Histogram-based transfer function editor. Each colour represents different density values, i.e. tissue. The arches represent derivatives of the data. Image courtesy of the SCI Institute [47].*

2.3.4 SOFTWARE ACCELERATION TECHNIQUES

As mentioned in previous sections; CPU-based volume rendering techniques, not least raycasting, are very time-consuming. The number of computations performed grows linearly with the number of voxels since they all, weather empty or not, have to be processed to produce the final image.

This computational expense is a major drawback since it is very important to obtain rapid feed-back when used in clinical environments. To target this problem a vast number of software acceleration techniques have been developed over the years. These methods can be roughly divided into two major categories; image- and object order techniques. The former generally achieve speed-ups by only casting rays from a subset of pixels, hence often giving decreased image quality. Object order acceleration techniques decreases workload by skipping empty space in some manner, thus generally not decreasing image quality.

Below some of the most common techniques are described. For a more in-depth review of current acceleration techniques we refer to [5, 23].

ADAPTIVE REFINEMENT

The main idea of adaptive refinement, first proposed by Levoy [2], is to adaptively improve image quality over time. Hence, a coarse image can be displayed in a fragment of the time it would take to perform full raycasting.

An initial image is generated by casting a small number of rays into the volume, e.g. one per four pixels, and interpolating missing pixel values between the resulting colours. Subsequent images are generated by discarding interpolated colours, and alternately casting more rays and interpolating. Where new rays are cast is based on colour differences, i.e. they are cast to maximize sampling in regions of high image complexity, for example in the vicinity of edges.

EARLY RAY-TERMINATION

In front-to-back composite raycasting, rays can be terminated when the accumulated opacity exceeds some user-defined threshold. Due to the nature of the algorithm, samples taken after this limit has been exceeded will not significantly contribute to the final pixel colour and can thus be ignored.

SPACE LEAPING

Probably the most investigated, and also most powerful, acceleration techniques involve some type of space leaping. These methods allow rays to skip over empty space, i.e. cells that can be considered empty in respect to the current opacity-colour mapping function. Usually this requires some kind of pre-processing to determine the extent of the objects in the scene.

A number of methods exist that utilize distance maps to alleviate the space-leaping, [24, 25, 26, 27, 28]. In a pre-processing step each empty voxel is assigned the distance to the nearest non-empty voxel. A ray that encounters an empty voxel during rendering can then safely leap forward the number of steps indicated by that voxel.

Another example is the PARC algorithm [29]. The pre-processing consists of enclosing the object in a convex polygon object. Two z-buffers then give the starting and termination point of all rays.

Yet another category of space leaping algorithms is the ones that subdivide the volume into smaller sub volumes. Some does this in only a single step, like bricking [19], whereas some do it recursively in a number of steps and create hierarchical data-structures, e.g. an occtree [30, 31, 18]. These sub-volumes are then treated as large voxels during rendering.

SHEAR WARP

As mentioned in section 2.1.2, the shear-warp technique [20] takes advantage of both image and object order volume rendering methods. The idea is to create a sheared parallel projection on a plane that is parallel to one of the faces of the volume. A 2D warping is then performed on the image plane. This allows for very fast rendering times, but the image quality is not sufficient for most clinical applications.

3 VOLUME RENDERING OF LARGE DATA SETS

In this section volume rendering of large data sets will be examined. First a brief introduction to the matter is given. Hereafter, memory management and acceleration techniques will be described. Focus lies on techniques developed by researchers at Vienna University of Technology [17, 18, 19].

3.1 INTRODUCTION

Volume visualization often deals with huge amounts of data. For instance, a standard 512^3, 16 bit dataset occupies roughly 270 MB of memory. These quantities are increasing rapidly, mainly since higher resolution data often is sought after when making diagnoses. Furthermore, the gap between processor and memory performance is increasing with every new generation of hardware. Hence, the main bottleneck in today's medical visualization systems is not CPU performance in the same way it was a decade ago, instead memory capacity and cache efficiency is.

Research has been carried out to handle these problems [14, 32, 17]. Both Knittel and Mora et al. accomplish high performance by using a spread memory layout. However, since their memory usage is approximately four times the data size, these techniques are not suitable for large data-sets. In contrast, Bruckner's approach uses dramatically less memory by performing all computations on the fly. High performance is achieved by using a bricked memory layout, an efficient addressing scheme and high level acceleration techniques. A parallelization approach is also described to further increase performance on multi-processor systems. Parallelization will not be handled further due to being outside the scope of this thesis.

3.2 MEMORY MANAGEMENT FOR LARGE DATA SETS

As mentioned above, the gap between memory and processor performance is increasing rapidly. Thus, memory access will be an increasing bottleneck when dealing with visualization applications that handle large amounts of data. In particular, raycasting techniques will suffer since the memory access patterns in such methods are highly irregular and thus cause cache misses. However, techniques exist that target this difficulty by exploiting the memory hierarchy of modern computers.

3.2.1 COMPUTER MEMORY LAYOUT

The memory of modern computers is organized in a hierarchy of successively larger, slower and cheaper memory levels (Fig 6). When a data item is requested, the request is propagated down the memory levels until the item is found. Since data is accessed frequently it would be a major advantage if most items are found in one of the fast caches. In other words; accessing the slower levels more than necessary should be strictly avoided.

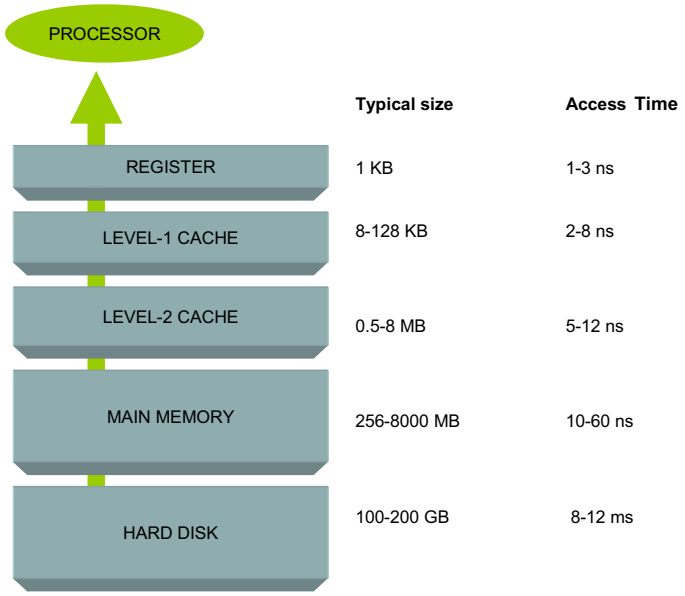


Fig 6 - *The memory of a modern computers consist of a hierarchy of consequently larger and slower memory levels.*

3.2.2 LINEAR VS. BRICKED VOLUME LAYOUTS

Normally data is stored linearly, i.e. the data is a set of 2D-images which are stored in lexicographic order. In standard raycasting rays are shot one after another into the volume. The volume data is then re-sampled along this ray and gradient computation, shading and compositing is performed to get a final pixel colour. Since rays often lie close to each other the same data frequently has to be accessed from the slower memory levels more than once. No spatial or temporal coherence is taken advantage, resulting in inefficient memory handling and low performance.

In bricking the data is divided into a number of small fixed-size data blocks, all stored in linear order. The idea is to choose the block size so that an entire block fits into the systems fast cache. The rendering pipeline is altered so that the computations are performed block-wise and thus all data-accessing can be done in the fast cache.

Moreover, in raycasting, a neighbourhood of samples often needs to be accessed when computing gradients or samples at off-grid points. Assuming tri-linear interpolation, seven neighbours have to be accessed for re-sampling and 6 to 27 neighbours for each gradient, depending on gradient estimation scheme employed. Multiplying this with the number of samples, one can see that efficient addressing is of major importance for performance critical systems.

3.2.3 ADDRESSING IN A BRICKED VOLUME

Addressing in a linear volume layout is straightforward and efficient since only one offset has to be fully computed; that of the integer parts of the current sample position (i, j, k):

$$offset_{i,j,k} = i + j*D_x + k*D_x*D_y$$

where,
Dx, Dy, Dz are the volume dimensions.

Once this is done the neighbouring samples can be accessed by simple algebraic operations. For example, accessing the samples needed for re-sampling can be accomplished by:

$$\begin{aligned}
offset_{i+1,j,k} &= offset_{i,j,k} + 1 \\
offset_{i,j+1,k} &= offset_{i,j,k} + D_x \\
offset_{i+1,j+1,k} &= offset_{i,j,k} + 1 + {}_{Dx} \\
offset_{i,j,k+1} &= offset_{i,j,k} + D_x*D_y \\
offset_{i+1,j,k+1} &= offset_{i,j,k} + 1\ D_x*D_y \\
offset_{i,j+1,k+1} &= offset_{i,j,k} + D_x + D_x*D_y \\
offset_{i+1,j+1,k+1} &= offset_{i,j,k} + 1 + D_x + D_x*D_y
\end{aligned}$$

In contrast to this, addressing in a bricked volume is more costly since two offsets have to be computed; one for the brick itself and one for the element within the brick:

$$offset_{i,j,k} = brickIndex * nrOfElementsInBrick + internalBrickIndex \qquad (8)$$

Considering the huge amounts of samples taken in raycasting this is quite inefficient. Although samples lying in the same block can be accessed in a similar way as above, this is not the case for samples lying on the edge to adjacent blocks. Even though these samples compose a minority, the computational overhead they add can not be ignored.

Bruckner deals with this problem by creating lookup-tables containing the offsets to neighbouring samples. The tables are not created for every sample position though, since most of them would contain the same offsets, creating an unnecessarily large table and thereby impairing the much sought after cache-coherence.

A brick is divided into different subsets depending on in which block neighbouring samples lie. If the same lookup table is used for both re-sampling and gradient calculation, 27 distinct cases exist, each with 26 neighbour offsets. To determine the subset membership of a sample position, and thereby the corresponding lookup table index [0-26], the following formulas are used[2]:

[2] For further reference see [17, 19]

$$
\begin{aligned}
i' &= (((((i \,\&\, (B_x - 1)) - 1) \,\&\, (2B_x - 1)) \mid 1) + 1) >> N_x \\
j' &= (((((j \,\&\, (B_x - 1)) - 1) \,\&\, (2B_y - 1)) \mid 1) + 1) >> N_y \\
k' &= (((((k \,\&\, (B_x - 1)) - 1) \,\&\, (2B_z - 1)) \mid 1) + 1) >> N_z \\
subset &= 9i' + 3j' + k'
\end{aligned}
$$

Where,
i, j, k are the integer parts of the current sample position
B_x, B_y and B_z are the brick dimensions,
N_x, N_y, N_z are the base 2 logarithms of B_x, B_y and B_z respectively
& denotes a *bitwise and* operation,
| denotes a *bitwise or* operation,
>> denotes a *right shift* operation, and
~ denotes a *bitwise negation*

After computing the first offset, $offset_{i,j,k}$, according to (8), the seven other voxel values needed for re-sampling can now be accessed by:

$$
\begin{aligned}
offset_{i+1,j,k} &= offset_{i,j,k} + offsetLUT[subset][0] \\
offset_{i,j+1,k} &= offset_{i,j,k} + offsetLUT[subset][1] \\
offset_{i+1,j+1,k} &= offset_{i,j,k} + offsetLUT[subset][2] \\
offset_{i,j,k+1} &= offset_{i,j,k} + offsetLUT[subset][3] \\
offset_{i+1,j,k+1} &= offset_{i,j,k} + offsetLUT[subset][4] \\
offset_{i,j+1,k+1} &= offset_{i,j,k} + offsetLUT[subset][5] \\
offset_{i+1,j+1,k+1} &= offset_{i,j,k} + offsetLUT[subset][6]
\end{aligned}
$$

Where,
offsetLUT is the lookup-table containing offsets to neighbouring samples depending on subset membership

Values needed for gradient estimation can be addressed in a similar manner.

3.2.4 TRAVERSAL OF A BRICKED VOLUME

As mentioned in 2.1.2, raycasting is an image-order approach to volume rendering as opposed to object-order solutions. One of its main advantages is that it allows for efficient acceleration techniques such as early ray termination (section 2.3.4). On the downside, image-order solutions have a very irregular data access, whereas object-order volume rendering allows for high cache coherency, i.e. data locality. In [33], Law and Yagel present an image-order raycasting scheme that utilizes the benefits of an object-order approach through an efficient traversal algorithm. A ray-front is advanced through a bricked volume and bricks are processed in visibility order. This solution has high cache coherency since the bricks will only be processed once, and one at the time.

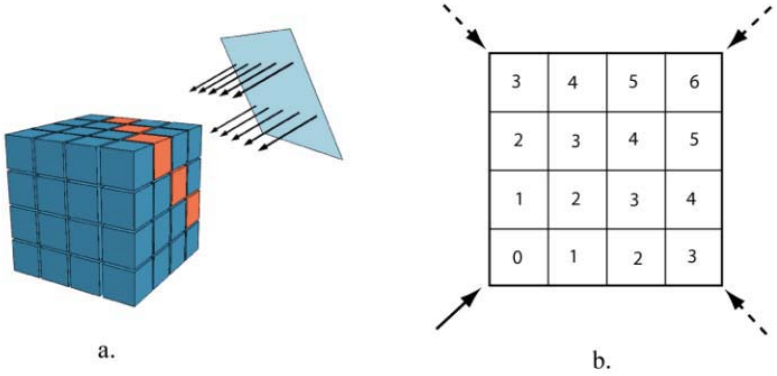


Figure 7 - *(a) Advancing ray front through a bricked volume. (b) Two-dimensional view of a bricked volume with seven brick lists. The arrows denote four possible view-cases, the un-dotted being the current one.*

The algorithm firsts sorts the bricks in front-to-back order, depending on the current viewing angle. In parallel projection eight distinct cases exist, each described by a set of lists that contain the order in which the bricks will be processed (fig. 7). These lists do not have to be computed in every pass, but can be created in a pre-processing stage and simply accessed during rendering using a lookup-table. Each brick holds a list of rays passing through it.

Initially, rays are assigned to the brick it hits first and are then passed on to other bricks as the ray front advances through the volume. Bricks are then processed in order according to the brick list of the current view. The algorithm ensures that no ray that is processed by a brick can be passed on to another brick in the same brick list. Hence, bricks are processed as independent objects, beneficial for parallelized systems. The traversal process is summarized in figure 8.

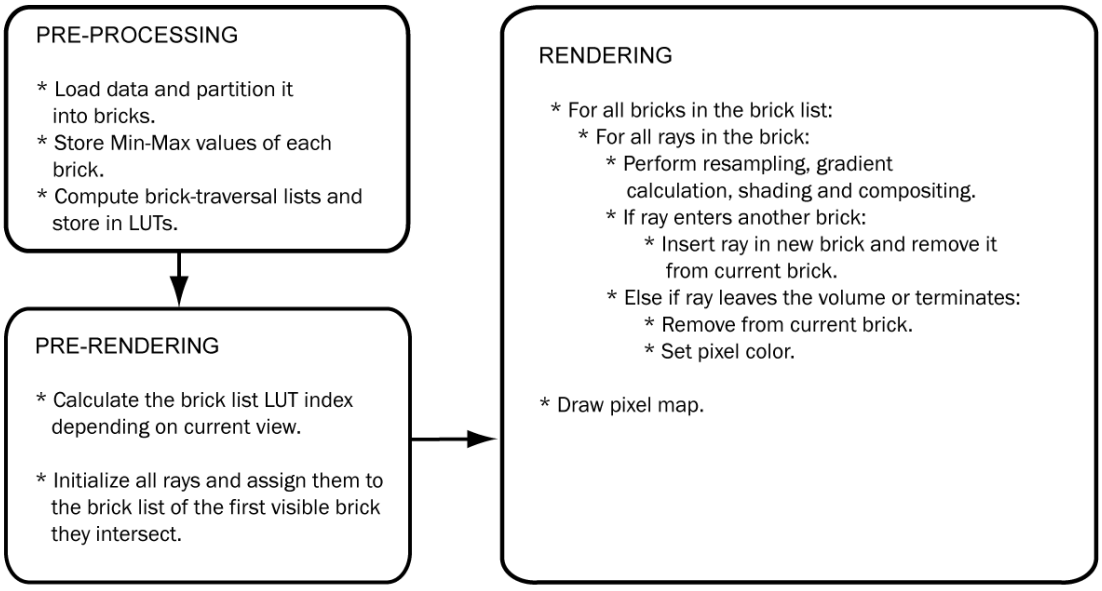


Figure 8 - *Rendering procedure of a bricked volume*

3.3 ACCELERATION TECHNIQUES FOR BRICKED VOLUMES

As described in [17], a bricked volume data structure can greatly improve cache coherency and enable large data to be rendered efficiently. However, the computational load of volume raycasting is still a major challenge, and many CPU based applications rely on clusters or super-computers to achieve interactive frame rates on large data set. In [18] Grimm et al, presents a number of acceleration techniques for CPU and bricked-based volume raycasting, allowing them to render a [512x512x1202] CT data set at 2.5 fps on a commodity notebook. Below follow summaries of the three key techniques: gradient caching, skipping of translucent bricks and cell visibility caching.

3.3.1 GRADIENT CACHING

Gradient estimation is one of the most important and computationally heavy parts of volume raycasting (see section 2.2.2). As mentioned, gradients can be computed in a pre-processing stage and stored in a separate data structure which is accessed during rendering. However, storing a gradient data structure of the same dimensions as the volume data in memory is not possible when dealing with large data. Thus gradients have to be computed on the fly. Herein lays a great inefficiency, since a grid-point gradient is likely to be computed several times during one rendering. For example, consider an off-grid sample point for which a gradient is to be computed using tri-linear interpolation. The gradients of the eight grid-point corners of the current cell is then required. It is quite likely that one or more of these grid-points are a corner of another cell, which will be sampled by the same ray later, or by an adjacent ray. This is especially true if a small step size is used or if the rays are at a small distance from each other, e.g. the view is zoomed in (fig 9).

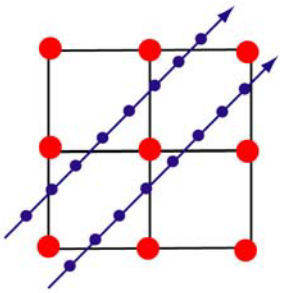


Figure 9 - *Gradient calculation is normally very inefficient since gradients often have to be computed several times. In the lower left cell for instance, each of the four voxel gradients are computed 5 times. In 3D the problem is even worse.*

To address this problem, Grimm et al, presents a solution where gradients that have been computed are stored in a cache data structure of the same size as a brick.[3] An additional Boolean data structure of equal size is used to store whether a grid-point gradient has been calculated or not. During sampling, the Boolean data structure is used to determine if a gradient has to be computed, or simply fetched from the cache. Whenever a new brick is being processed the cache is emptied and the Boolean data structure reset. The gradient cache decreases the number of calculations that has to be performed during a rendering pass enormously at a reasonably small memory expense.

[3] Actually, the size of the cache is (brick dimension + 1)^3, to allow for interpolation over the borders of a brick.

3.3.2 BRICK SKIPPING

Depending on the choice of transfer function, the amount of data that is to be visualized varies, and in many cases large sections of a volume will not be visible at all. Algorithms designed to efficiently skip such areas can greatly improve the performance of a ray caster. A sub-divided data structure such as bricking, allows for a coarse classification of the volume in object space so that entire sub-sections, in this case bricks, of the volume can be set as invisible and thereby skipped during rendering.

In a pre-processing stage the maximum and minimum value of each brick is stored. The value range can then be checked against the current transfer function to determine whether a brick is visible or not. Grimm et al. also employs a more granular sub-division by encoding each brick in a three level octree.

3.3.3 CELL INVISIBILITY CACHE

To further eliminate processing of non-visible cells, Grimm et al introduces a Cell Invisibility Cache, CIC, which ensures that empty cells only has to be classified once. The CIC encodes the visibility of each volume cell by one bit. All cells are initialized as visible and once a cell has been classified as invisible, the classification of all samples inside that cell can be replaced by a binary test. However, a visible cell has to be classified every time, since it is not possible to know if it has been classified previously without an additional data structure. The CIC remains valid as long as the transfer function does not change, and will gradually fill up so that rendering time decrease for each run. Figure 10 shows an extended ray-traversal pipeline with a CIC.

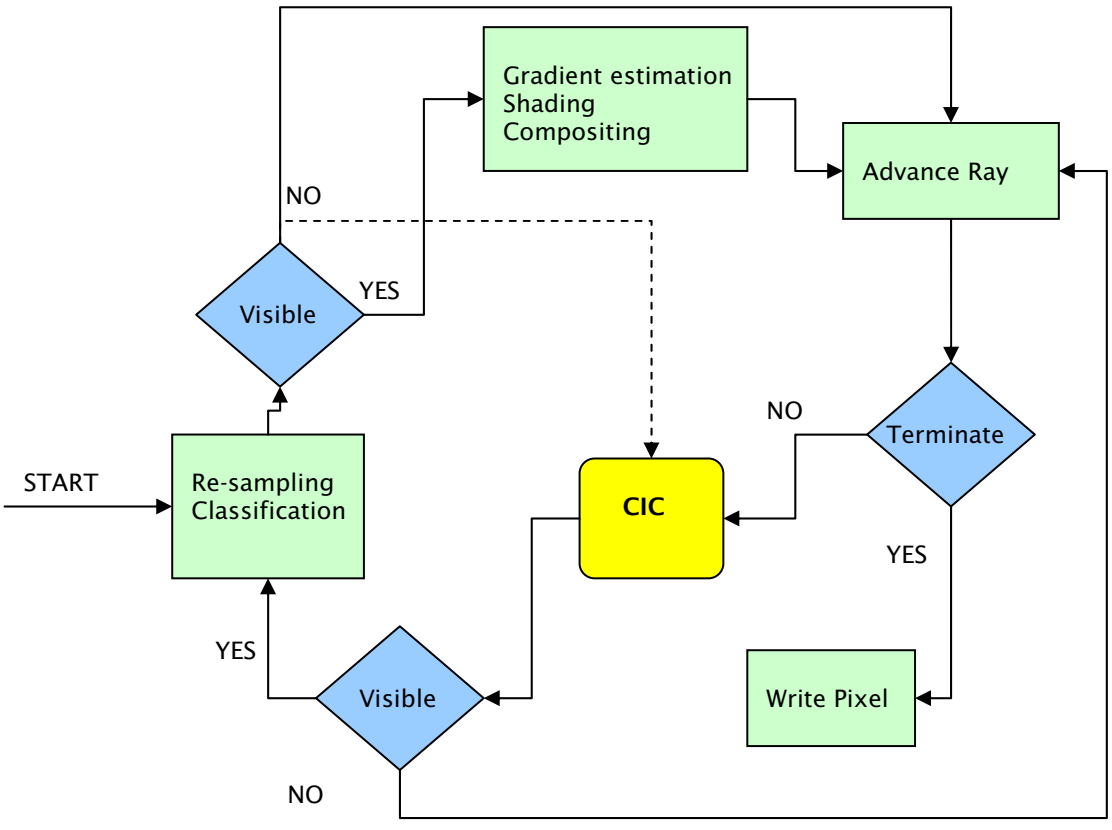


Figure 10 *- Ray-traversal pipeline with a CIC.*

4 LEVEL SET SEGMENTATION

Segmentation can be defined as the process of finding interfaces between different regions of interest. Even though this could be accomplished using transfer function design in volume rendering, the two methods should not be confused. Most segmentation algorithms explicitly extract geometry, which often are rendered using triangles, whereas volume rendering is a direct visualization technique. Most techniques typically use not only intensity values, like for instance Marching Cubes [34], but also their derivatives as a basis for segmentation. Segmentation can either be used to detect hard edges, based on intensity changes, or to separate regions with smoothly varying intensities.

Segmentation can be divided into two main categories; simple segmentation techniques and semi-automatic segmentation techniques. The former includes basic methods such as manual segmentation, threshold classification and region growing. These techniques are too limited for many applications and therefore much research has been focused on the second category. Methods such as live wire [45], and active contours [46], can semi-automatically find regions with high gradients, but they are generally limited to 2D applications.

In this chapter we will focus on a more powerful mathematical method which allows automatic or semi-automatic segmentation in three dimensions, level sets. Note however that the topic will only be briefly covered. For a more thorough description of the subject and detailed mathematics please refer to [5, 6, 7, 35, 36, 37, 42].

4.1 LEVEL SET METHODS

First proposed by Sethian and Osher [35], level set methods can describe complex deformable shapes of arbitrary dimensionality and topology. They implicitly represent an n-dimensional interface by embedding it in a space of dimension n+1, giving the interface a *co-dimensionality* of one. For example, a one-dimensional curve is represented in a 2D-space and a 2D-surface is represented in a 3D-space.

For each grid-point, the level set function, Φ, defines the closest distance to the boundary, the zero level set (Fig 11). Distances inside the boundary have opposite sign from distances on the outside. In this thesis a positive sign will denote distances outside the zero level set and a negative sign distances inside.

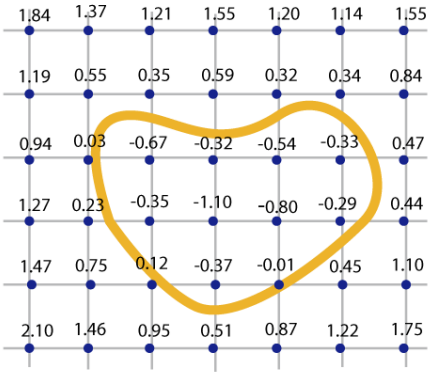


Figure 11 - *A simple level-set; a closed line represented by a two-dimensional distance map.*

By introducing a time-dimension into the model, the level set can be deformed in any way using so called *speed functions*, partial differential equations that determine how the boundary will move over time. These functions can range from constant steps in the surface normal direction (eq. 9), to complex edge-detecting equations. The advancement of the level set is computed by solving the speed function.

$$F(x,n,\phi,..)=n\cdot\frac{dx}{dt}=\frac{\nabla\phi}{|\nabla\phi|}\cdot\frac{dx}{dt} \qquad (9)$$

Level set models have several advantages. They generate closed, non-self intersecting surfaces which can easily change topology over time. Furthermore, they do not suffer from mesh connectivity problems and there is no need to re-parameterize during deformation.

Many frameworks for level set segmentation of volumetric data exist [6, 42]. Describing them in depth is outside the scope of this thesis. However, common for all is that they produce a separate volumetric representation of the segmented data. This representation is generally converted into a polygon mesh which can be rendered by graphics hardware.

4.1.1 NARROW BAND METHODS

If only a single level set is of interest, computing the solution over the entire domain of phi is highly inefficient. Fortunately, the evaluation of a single level set only has to be performed in the vicinity of it; such methods are called narrow-band schemes [36]. As in normal level set methods, they construct an embedding of the evolving curve or surface in the form of a distance function. However, the function is only valid in a narrow band of grid points around the boundary, called the gamma band (fig 12). All grid points outside this band are set to a constant value, equal to width of the band, λ. In three dimensions, this is equivalent to a tube surrounding the level set interface. Not only does this lead to a more efficient evaluation of the level set function, it also allows fore a more memory efficient representation [37].

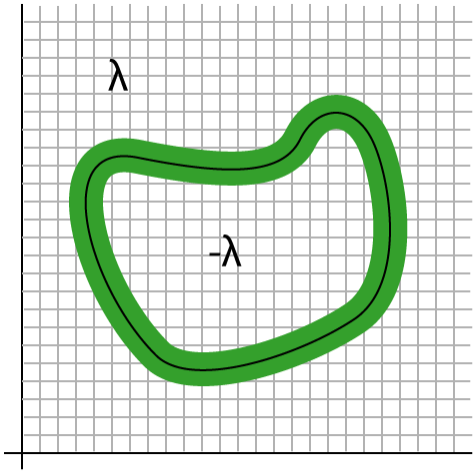


Figure 12 - *A narrow-band level-set. Only the distances inside the band of width λ surrounding he interface are defined. Outside the narrow band, distances are set to a positive (outside), or negative (inside) constant.*

PART TWO

THESIS WORK

5 COMBINING VOLUME RENDERING OF LARGE DATA SETS WITH LEVEL SET CLIPPING

The major motivation for using volume rendering is its capability to simultaneously visualize several types of internal structures. This is made possible through designing transfer functions, mapping mass density to colour and opacity. However, separating interesting structures can often de difficult using only transfer functions.

Clipping is the process of excluding parts of a volume depending on their voxel position, in contrast to transfer functions which are based on the voxel value. Often clipping is performed using a plane, or a combination of planes. Although some geometry can be approximated in this way, many can not, and more complex clipping objects are then needed. In medical imaging, segmentation data can often be made available and can be used to define complex clipping geometry as polygon meshes. However, testing weather a voxel lies inside the polygon object is a rather complex and computationally expensive task.

In this thesis, the possibilities of using segmentation data directly as clipping objects, without using intermediate polygon objects, was examined. Level set segmentation provides a powerful means to extract and represent complex internal structures in such a way. Using a narrow-band scheme (see section 4.1.1), level sets can in addition be represented in a memory efficient manner. They are therefore particularly suitable when dealing with large data sets, where memory size already is a limitation.

First the adjusted ray traversal pipeline, including level set clipping, will de described, followed by acceleration possibilities and a discussion of how to achieve high quality interface shading.

5.1 LEVEL SET CLIPPING

Representing clipping geometry with a triangular mesh might seem as a straight-forward way, since it is an efficient and hardware-optimized way of handling objects of arbitrary size and complexity. When dealing with basic clipping or probing objects, such as spheres and cubes, this assumption holds. However, if the clipping is performed with segmentation data, the mesh has to be extracted from volume data using techniques such as Marching Cubes [34]. In this case, using the volumetric segmentation data for clipping is a more direct approach. Furthermore, the testing of whether a sample is inside or outside a triangular mesh requires advanced ray-surface intersection tests, which creates a calculation overhead. This problem is not present in level set clipping.

The adjustment of the ray traversal pipeline to include level set clipping is rather straightforward (fig 13). Simply put; a test to see if a sample point lies inside the level set or not is all that is needed. Since the level set inherently holds distances from every grid point to the interface, this is a simple task. However, since distances to the level set can only be computed at grid points, interpolation needs to be performed in the vicinity of the interface in order to get a good representation of the clipping boundary and avoid artefacts. Outside this region, a simple nearest-neighbor re-sampling of the level set is sufficient.

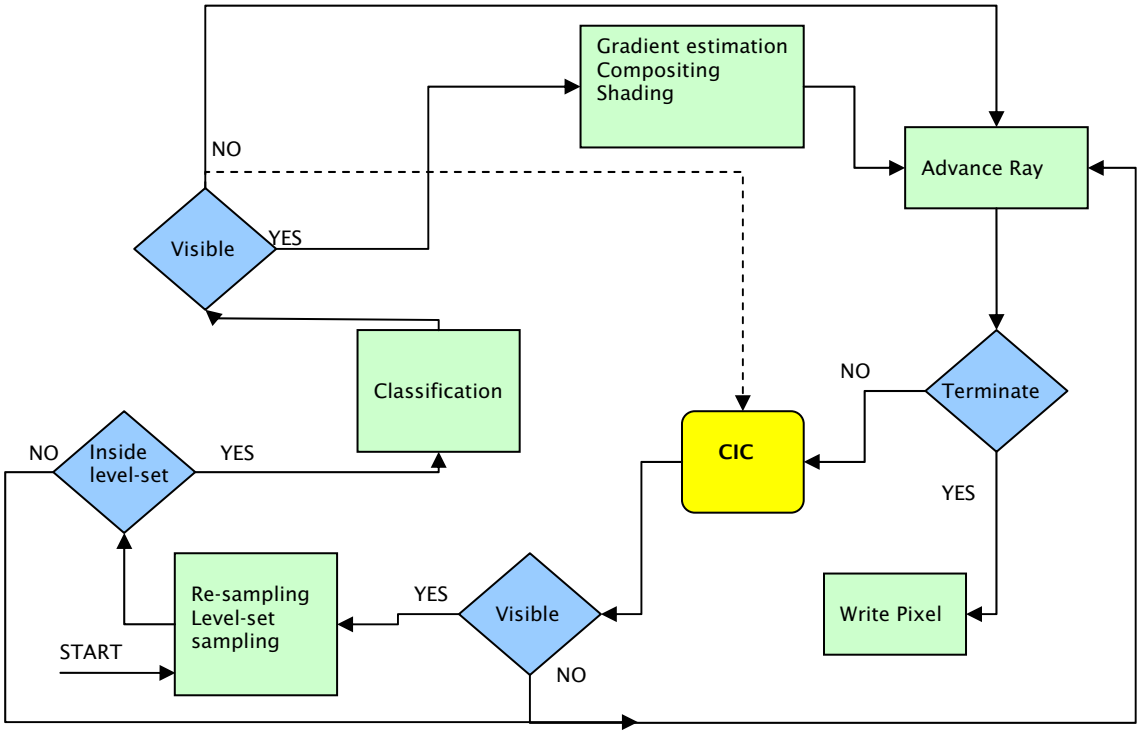


Figure 13 - *Ray traversal pipeline with CIC and level-set clipping.*

The introduction of a clipping geometry in the rendering pipeline requires an additional data structure which must be small unless becoming a memory bottleneck. The memory consumption of triangular mesh geometry is highly dependent on the complexity of the mesh. Since a level set object is defined over the entire domain, its size depends on the grid resolution. In the case of a narrow-band level set, the size of the data structure rather depends on the total area of the iso-surface, a clear advantage when dealing with large data sets.

Another advantage of using level sets as clipping boundaries is that they allow for the interface to be altered during run-time. Simple morphological operations such as erode and dilate are easily implemented and in theory, speed functions of any complexity could be applied. In the application described in this thesis however, segmentation has to be performed externally and is then loaded from a file.

5.2 LEVEL SET ACCELERATION

The signed distance function of a level set has yet another potential advantage for volume rendering, i.e. the possibility of implementing efficient space leaping. Instead of using a constant step size, a ray advancing through the volume can safely leap the distance given by the distance function without intersecting the clipping boundary. Once inside the level set, re-sampling is performed with normal step size.
If the level set is defined over the entire volume, this scheme can be very efficient since the surface can be reached in only a few steps. However, if a narrow band level set is used, the ray can at best jump a distance corresponding to the width of the band, γ. This is due to the fact that all grid-points outside the gamma band will be set to that distance. Hence, if

the gamma band is very narrow, the safe distance to leap is rather small. The overhead of computing the distance to the level set at each sample point then becomes greater than the speed-up of the space-leaping. Again, a trade-off between memory usage and rendering speed has to be made.

5.3 SHADING OF CLIPPING INTERFACE

To achieve high rendering quality of the clipping boundary without artifacts and incorrect shading, several issues has to be considered. First, the clipping boundary has to be smooth, without artifacts or jaggy edges. Second, the shading of the boundary should reflect both the clipping geometry and the volume data in the vicinity of the boundary.

In [4], two different methods to perform such a clipping are described. One of them performs clipping by adding an extra volume, representing the clipping geometry, and transforming it to a signed distance function; essentially a level set.

Clipping is performed as described in previous sections, but a problem occurs when the clipped volume is to be shaded. As mentioned, shading should reflect both the clipping geometry and the volume data; the question is to what extent. A method described in [4], called gradient impregnation, takes both of these into account. The idea is to compute the shading components separately and combining them by a weighting function. Note that this only has to be performed for voxels lying at a small distance inside the level set.

First, the shading of the closest point on the clipping surface has to be computed. The closest point on the interface can be calculated using the closest-distance transform:

$$CPT(x) = x - phi(x)*grad(phi(x))$$

Once done, the gradient at that point can be computed using central difference. The shading of the level set interface is calculated using a standard Phong equation (see section 2.2.3).

The influence of the volume and the level set shading is determined by a weighting function, *w*. Any interpolation function can be used, but a linear ramp function is often sufficient. The final shading at position x is calculated as:

$$C = w * colorVol + (1-w) *colorLS$$

This way, a smooth transition from clip geometry- and volume data shading is achieved. The results can be seen in figure 14.

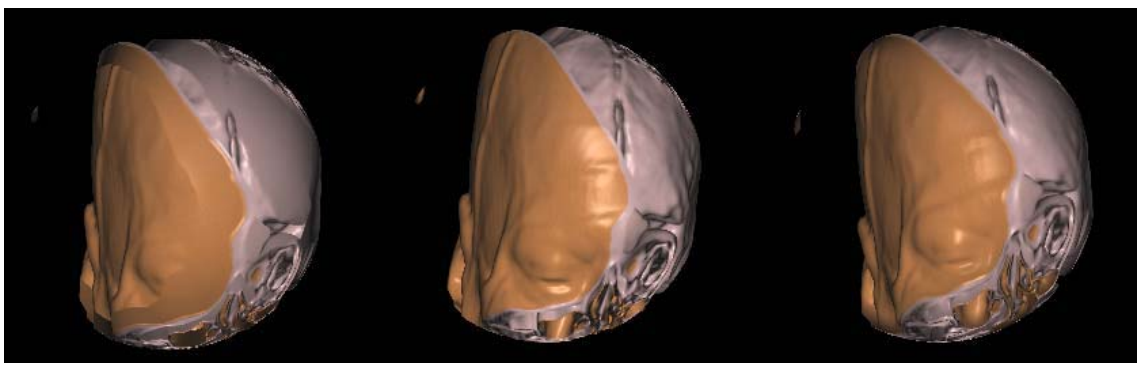


Figure 14 *- Different methods of shading the clipping interface. Left: only level set data shaded. Middle: only volume data shaded. Right: weighted shading, 50% each.*

6 IMPLEMENTATION

In this chapter, we will briefly describe the implementation of our CPU based volume renderer, capable of rendering large data sets and perform clipping with level sets.

The implementation was carried out in C++ using Microsoft Visual Studio .Net. The application was developed entirely in software, except for the windowing, I/O functions and pixel drawing, which was implemented using OpenGL [38] and the OpenGL Utility Toolkit, GLUT [39], an API for platform independent I/O functions. The user interface was written in GLUI, a GLUT-based GUI API. For the level set functions, the Graphics Group Library, GGL [40], was used.

6.1 SYSTEM ARCHITECTURE

The system was designed in an object-oriented manner aiming to achieve high cache coherency (see section 3.2). In addition to this, a component-based design was used to alleviate future development and extensions. A simple class diagram of the system, as well as more detailed class descriptions, can be found in appendix 2.

Basically, we have used the approach to volume rendering of large data sets described in chapter 3, added a bounding box test, and combined this with the techniques for level set clipping described in chapter 5. However, we only use a two level hierarchy for empty space removal, consisting of bricks and voxels, as a contrast to Bruckner's three level approach.

Level set segmentations cannot be done directly in our application; instead segmentations can be loaded and used as clipping data. Some simple morphological operations have also been added.

6.2 ALTERED APADTIVE REFINEMENT SCHEME

To be able to interact with larger volumes (>256^3) we have also implemented a method similar to Levoy's adaptive refinement technique (see section 2.3.4). Instead of refining the image progressively in several steps, we have chosen to make the transition in simply two phases. Furthermore, we do not ray-casting all pixels in the final image, but still attain full image quality.

During interaction only the first phase, corresponding to the first step of the original algorithm, is performed. Flags, which are cleared between frames, are used to keep track of which pixels have been ray-cast or interpolated.

During non-interactive modes, when full image quality is required, the second part of the algorithm is performed in addition to the first one. This step only processes pixels not already ray-cast during the first stage. The difference from Levoy's technique is that all object-pixels are ray-cast, whereas only a sub-set of non object-pixels are. This is made possible by setting an initial block size, iterating over the pixels and recursively subdivide only if the corner pixels of a block not all have the same color and that color is not the

background color. Hence, substantially faster rendering times are achieved while still performing full ray-casting of the object.

Note also that no interpolation needs to be performed in the second step since pixels not ray-cast correspond to the background color. However, the shape of the object has to be considered before choosing the initial block size, so that no object part can be missed between background-colored block corner pixels.

6.3 SUPPORTED FEATURES

The core of the application is a full-featured volume renderer able to visualize data sets of different sizes and dimensions with a varity of options. Below follow a list of the most important features:

- Support for bricking with arbitrary dimensions, as long as they are a power of 2.
- Support for 8 and 16 bit data sets.
- Support for scaling in all dimensions.
- Support for 320x240 or 640x480 pixel maps, other sizes can easily be added.
- Support for the most common projection types, i.e. x-ray projection, maximum intensity projection (MIP) and full volume rendering.
- Support for different interpolation types, at the moment nearest-neighbour and tri-linear interpolation.
- Support for different sampling steps.
- Support for different preview coarsenesses, to alleviate interactive rendering of data sets of different sizes.
- Support for gradient modulation to enhance visualization of interfaces between different types of tissue.
- Support for two different zoom functions, a true zoom as well as a pixel zoom function.
- Support for different acceleration techniques such as early ray termination (with a user defined threshold), skipping of translucent bricks, gradient caching and cell visibility caching.
- Includes a simple transfer function editor able to create, save and load linear RGBA-transfer functions.
- Loading and using level set segmentation as clipping data.
- Support for simple morphological operations on the level sets, i.e. erosion, dilation, opening and closing.

6.4 GRAPHICAL USER INTERFACE

Volume rendering requires a large number of variables and thus some kind of graphical user interface is needed for efficient use and testing. This is especially true since many of the parameters should be adjustable during runtime, allowing the user to alter the visualization interactively. With system portability in mind, the GLUT-based GUI API, GLUI, was used.

The application consists of two windows, one being the graphical output and the other one being the user interface (see fig 15-16). In the output window the user can rotate the rendered object using a simple virtual trackball function. The GUI is composed by five collapsible panels, or rollouts, which holds components to adjust parameters regarding file properties, rendering options, acceleration, transfer function editing and clipping.

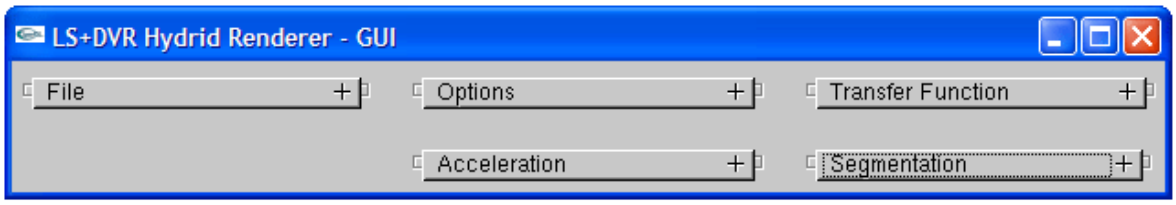


Figure 15 *- User interface with all rollouts collapsed. The interface consists of five main categories: file, options, acceleration, transfer function and clipping.*

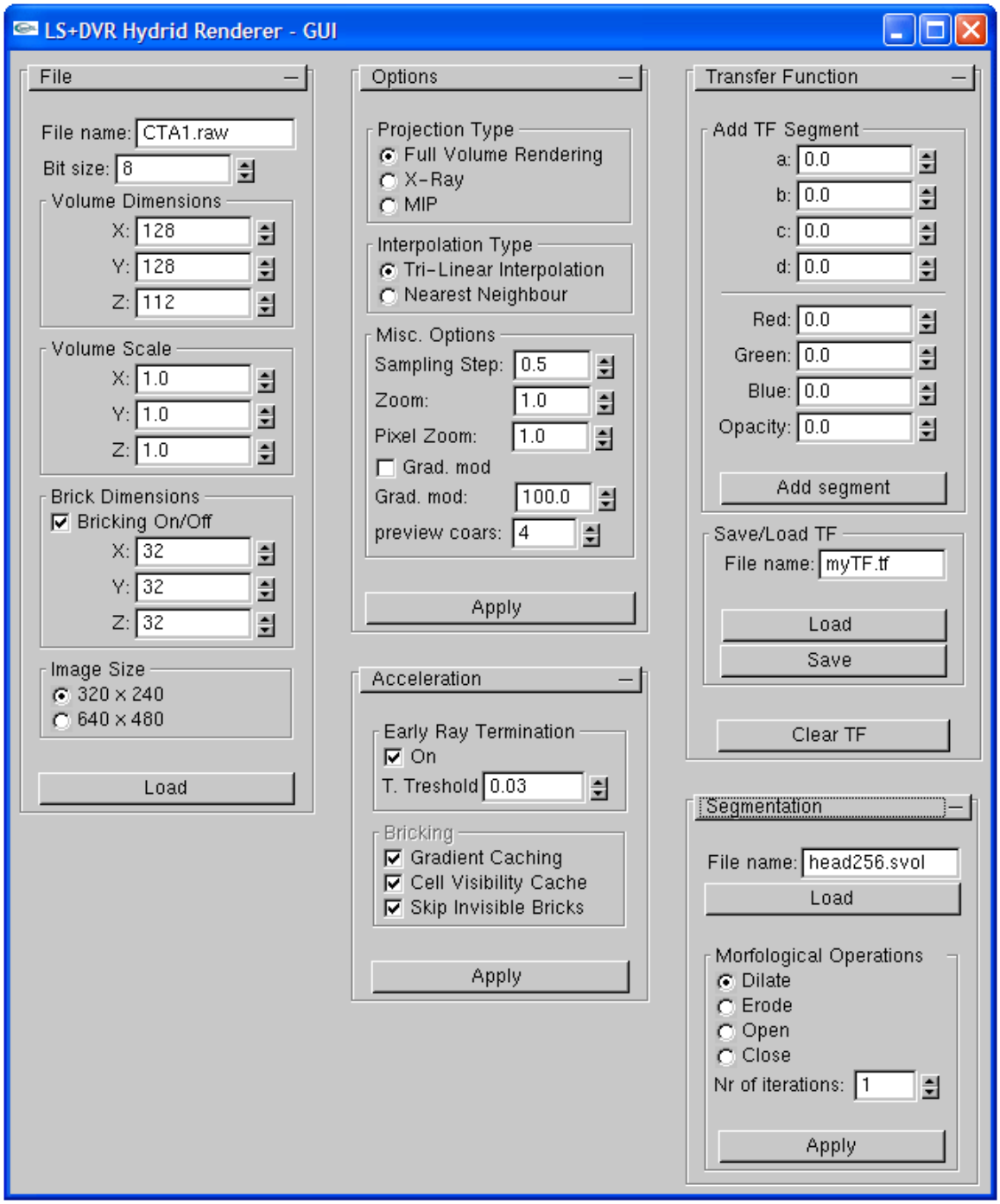


Figure 16 *- Complete user interface.*

7 RESULTS

In the following, the performance of our system will be presented and evaluated. First, the volume renderer will be assessed in respect to image quality and rendering time. Acceleration techniques and other methods employed will also be tested and compared. Next, the implemented level set clipping will be evaluated and results presented.

7.1 THE VOLUME RENDERER

When evaluating our system, two data sets frequently used for analyzing performance of volume renderers were utilized (fig 17) The first, the UNC head [43], is a 256x256x225 dimensioned, 8-bit data set of a male head. The second, the Visible Male [44], is a 512x512x1621 dimensioned, 16-bit data set of an entire male body.

All benchmark tests were performed on a portable Intel Centrino Duo System with 2Mb of L2 cache and 2 Gb RAM. Rendering times were measured as an average of five frames using a timing feature with millisecond precision.

Figure 17 *- data sets used for testing. Both renderings are done with the same transfer function, a sampling step of 0,5 and an image size of 640x480 pixels. a) The UNC head. Data dimensions: 256x256x225. Voxel bit siz: 8. b) The visible male: Data dimensions: 512x512x1621. Voxel bit size: 16. For rendering times please see 7.1.2 and 7.1.3.*

7.1.1 IMAGE QUALITY

Our volume renderer is capable of producing very high quality renderings of medical data sets, matching or even surpassing the image quality of many other systems. The support for gradient modulation further enhances the capabilities to visualize interfaces between different types of tissue without compromising the visualization of internal structures. However, as in all volume renderers, there is always a trade of between image quality and rendering time; high quality is attained at the cost of long rendering times, and vice versa. Nonetheless, since rendering parameters can be altered during run-time, low quality renderings can be used when finding the region/s of interest and the parameters can then

be altered to analyze the data with full quality. Also note that all acceleration functions applied, except early ray termination, does not compromise image quality. A few renderings to illustrate the image quality of our system are presented in figure 18 below.

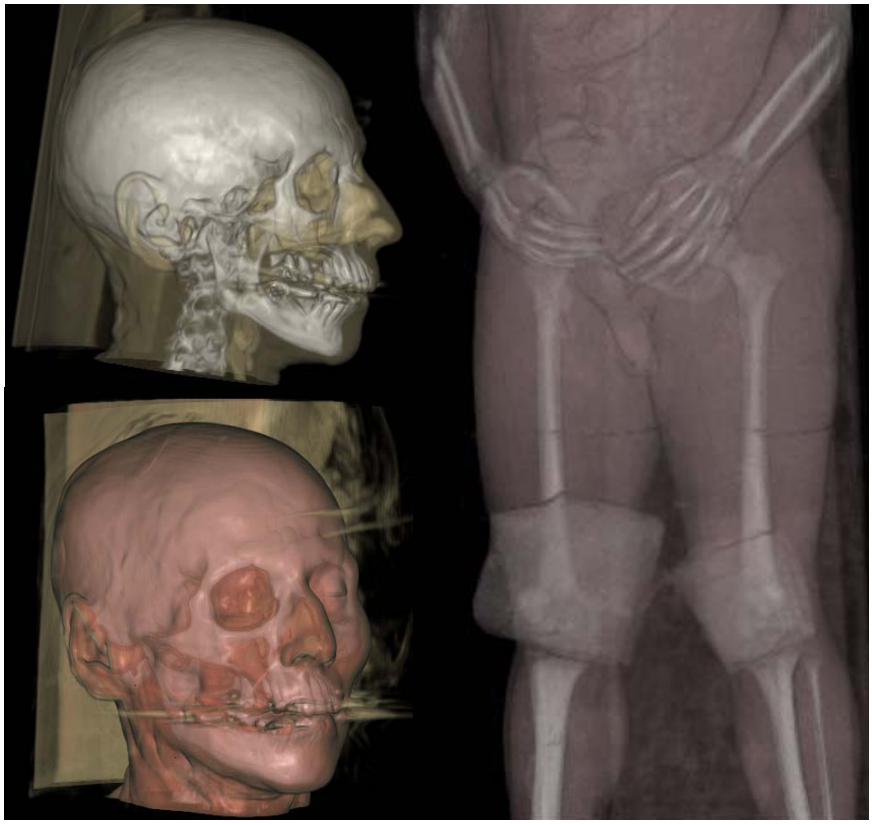


Figure 18 *– High quality renderings. Left: UNC-head, Right: Visible Male.*

7.1.2 BRICKING

The main motivation for using bricking is that the size of bricks can be chosen to fit into the fast cache memory (See section 3.2). Together with an efficient addressing scheme (see section 3.2.3), this should lead to considerable speed-ups without higher level optimization. For instance, Bruckner [17] achieves decreased rendering times by a factor of 2.8 when using an optimal trade of between brick and cache size for his system.

Unfortunately, our implementation only exhibit slight signs of better cache coherency due to the bricked volume layout, nowhere near Bruckner's performance increase factor of 2.8. Figure 19 shows the rendering times for varying brick dimensions without other acceleration techniques being employed. As can be seen, the rendering times decrease down to a brick size of 64^3 for both data sets. This brick size corresponds to 262 Kb for the 8-bit UNC head and 524 Kb for the visible mal and is also the brick size that should be the optimal for our test system. Larger dimensions lead to performance decreases since these bricks don't fit in the cache but still cause an overhead due to the employed scheme. When no bricking is used this overhead is eliminated and the rendering times are again improved.

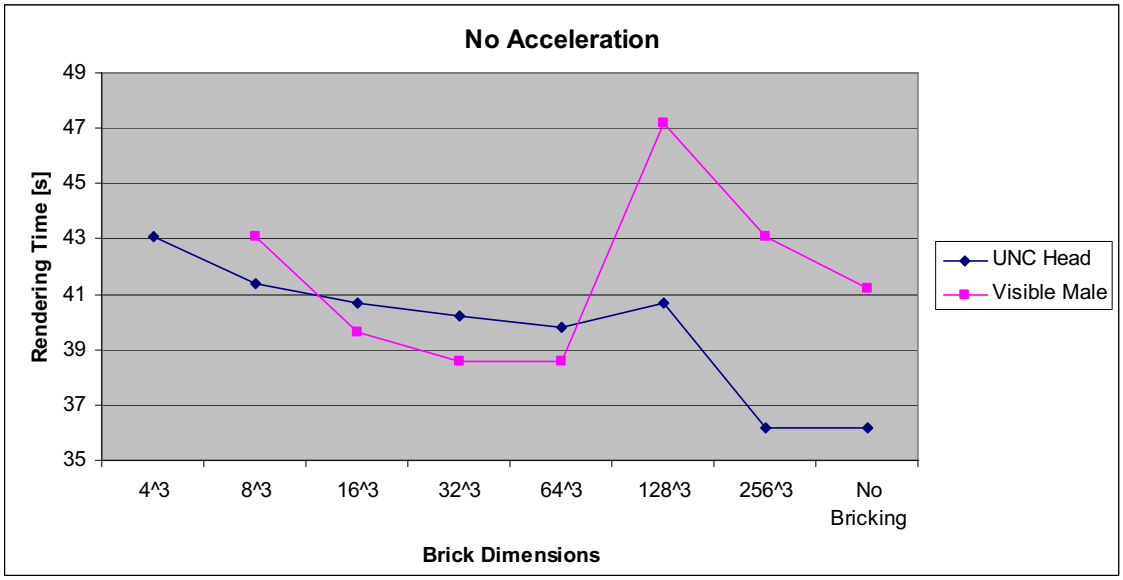


Figure 19 *- Rendering times for the UNC head and visible male using different brick dimensions. The renderings are achieved using brute-force, i.e. no optimization techniques have been employed.*

Interesting to note is that rendering times decrease up to a brick dimension of 64^3 for both data sets. This is the optimal brick size for our system, i.e. when exceeding these dimensions the data of one brick no longer fits in the test systems 2 Mb L2 cache and the performance decreases.

Moreover, the best result for the UNC head is achieved when deactivating bricking and instead using a linear volume layout. However, since all acceleration techniques we have implemented require bricking, this is not a sound alternative anyhow.

Since cache management is very hard to analyse and debug in other ways than observing the change of performance, the cause for the lack of speed-up has yet to be determined. However, we believe that the reason is due to the use of local variables. Since most cache architectures work in a FIFO (first in, first out) manner, much of the cached data will be ejected when using many local variables. The choice would be to not use any local variables, but since this would make the code practically unreadable and tests has not proven this to be the case, we have decided to postpone this work.

7.1.3 RENDERING TIMES AND ACCELERATION TECHNIQUES

As mentioned previously, volume rendering of large data sets in software requires immense acceleration efforts in order to give acceptable results. Without any acceleration, i.e. a brute-force sampling of the entire volume, the rendering of the UNC head shown in figure 17 takes approximately 36 s. By using the implemented acceleration techniques this time can be reduced to less than one second, or about 2%, of the initial rendering time. This is due to the fact that decomposition of volume data into bricks allows for the implementation of powerful and memory efficient acceleration techniques (see section 3.3).

Figure 20 and 21 show the rendering times for the UNC-head and the visible male respectively using different brick sizes and acceleration by early ray termination, gradient cashing, skipping of invisible bricks, cell visibility caching and combinations of these. For exact rendering times please refer to the tables in appendix 1. Note that these figures are in no way generic for the data sets since the employed transfer function plays a major part in how fast a rendering can be performed.

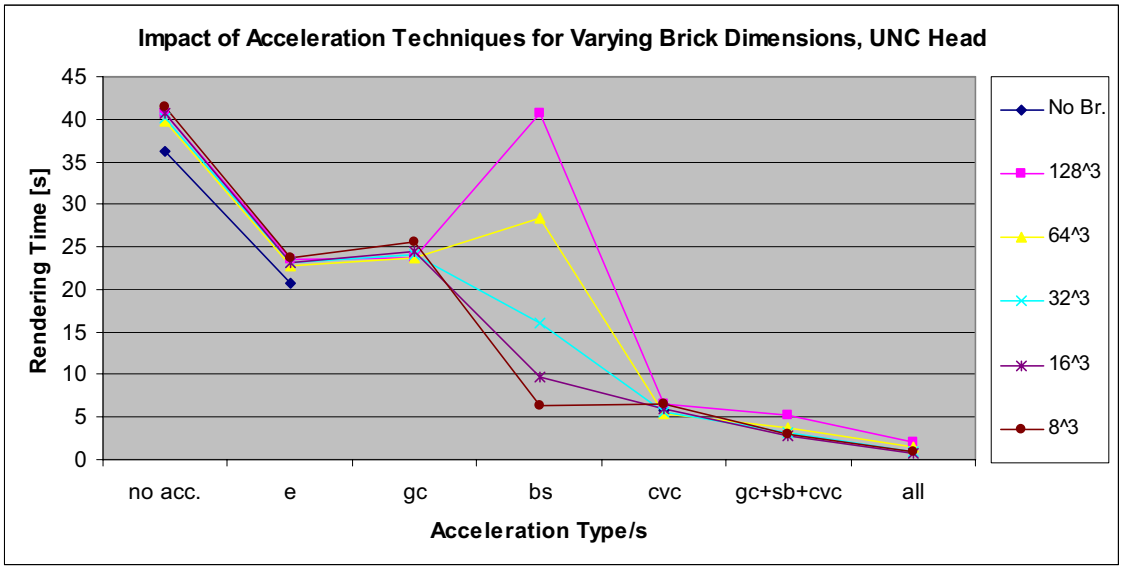


Figure 20 *– Rendering times for the UNC head with different acceleration functions; early ray termination (e), gradient caching (gc), brick skipping (bs), and cell visibility caching (cvc).*

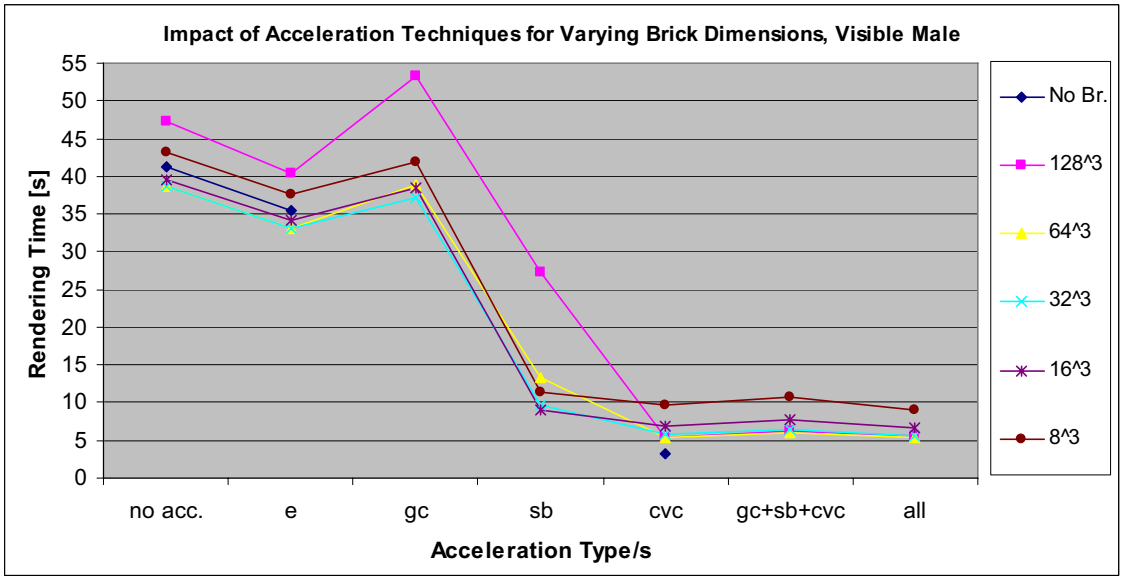


Figure 21 *– Rendering times for the Visible Male with different acceleration functions; early ray termination (e), gradient caching (gc), brick skipping (bs), and cell visibility caching (cvc).*

EARLY RAY TERMINATION

Early ray termination can reduce the rendering time considerably without introducing any visible loss of quality in the image if choosing a proper threshold. Here we have used a limit of 97%, i.e. when the opacity has accumulated to this percentage the ray is terminated, since this was the limit where we could not see any visual degradation. Note that the efficiency of early ray termination relies on what transfer function that is being used. The more rays that intersect opaque samples and hence terminate, the faster the rendering will be completed. This could be one reason why the technique is more powerful for the UNC head rendering. Rendering times for different brick dimensions when early ray termination is employed is shown in figure 22.

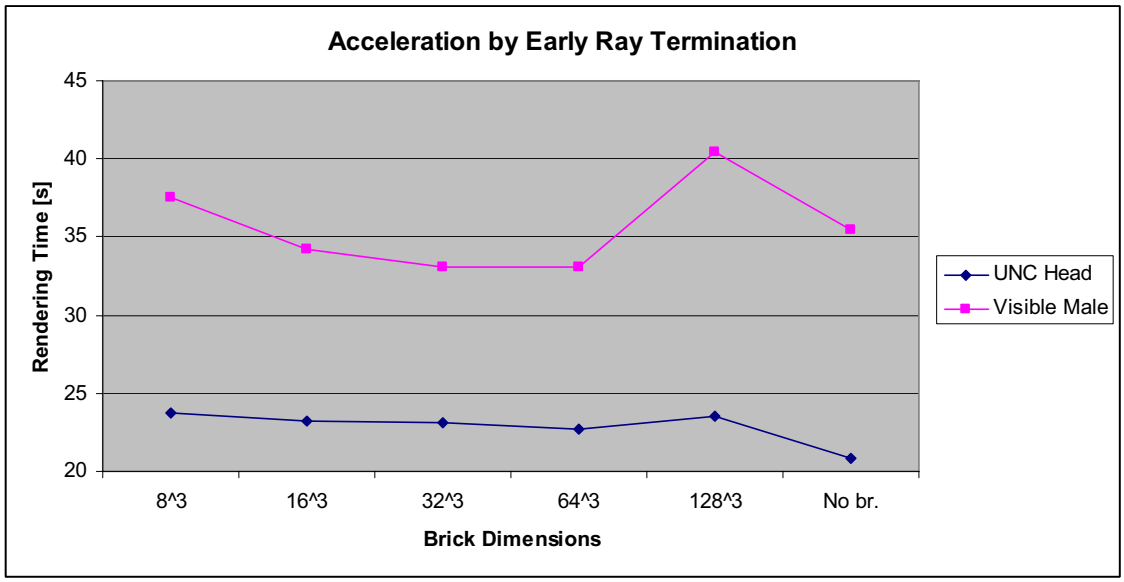


Figure 22 *- Rendering times for both data sets in relation to brick dimensions. Acceleration by early ray termination is employed, with the opacity threshold set to 97%.*

GRADIENT CACHING

Gradient caching decrease rendering times by approximately 30% for the UNC head and does not seem to depend heavily on brick size. This fact further strengthens our beliefs that our caching scheme is not working properly.

When using the visible male however, there are indications that the cache coherency functions, at least partly. With larger brick dimensions than 64^3 the rendering times increase significantly. One reason why this is more apparent with the visible male than with the UNC head could be that a great deal more bricks are being handled in this case, and accessing times are accumulated. Rendering times for different brick dimensions when gradient cashing is employed is shown in figure 23.

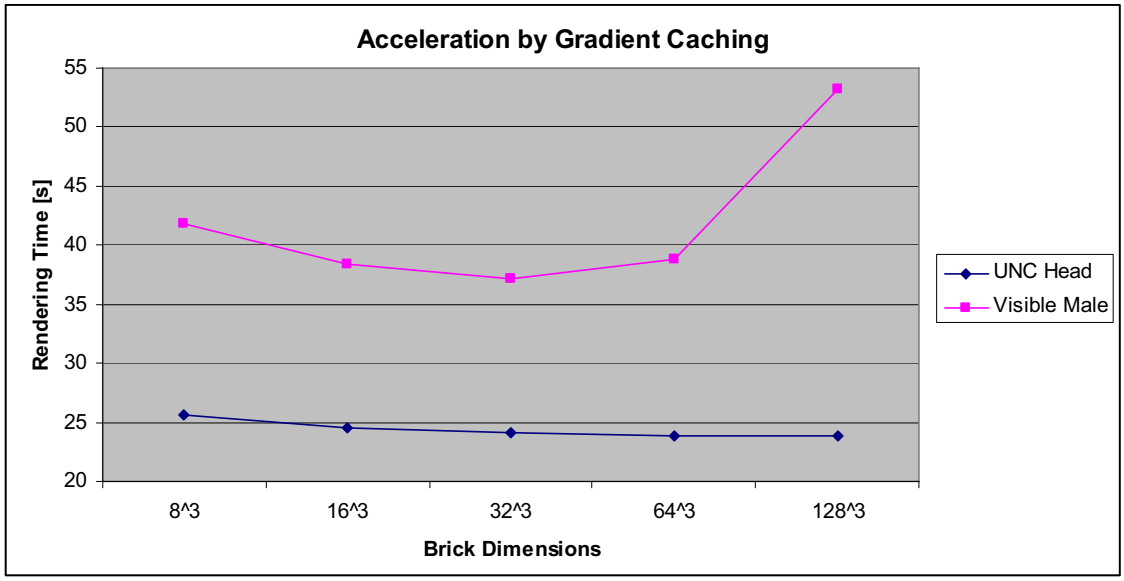


Figure 23 -*Rendering times for both data sets in relation to brick dimensions. Acceleration by gradient caching is employed.*

SKIPPING OF TRANSLUCENT BRICKS

As expected, skipping of invisible or transparent bricks is highly dependent on brick dimensions. If bricks are too large, few or none of them will be without any visible data, thus they cannot be skipped. With a brick size of 8^3, rendering time of the UNC head is reduced with more than 80%. However, using such small bricks creates an unwanted overhead when dealing with larger data sets like the visible male, due to the enormous amount of bricks used. This is indicated by the slight performance decrease seen when rendering the visible male with a brick size of 8^3. Rendering times for different brick dimensions are shown in figure 24.

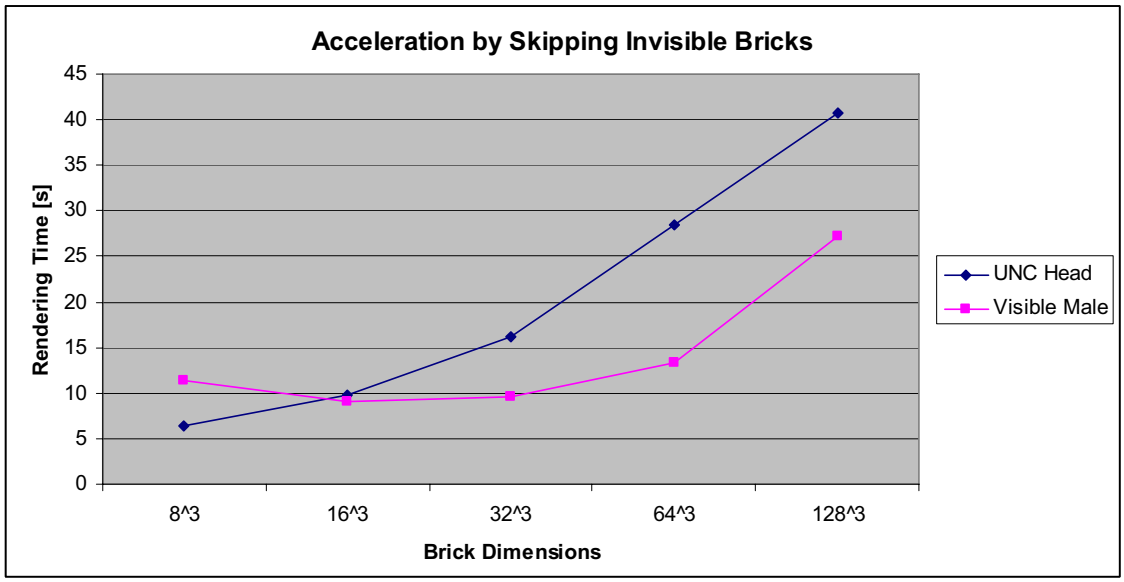


Figure 24 - *Rendering times for both data sets in relation to brick dimensions. Acceleration by cell visibility caching is employed.*

CELL VISIBILITY CACHING

Since the cell visibility cache operates on the entire volume, it is not dependent on brick dimensions. Looking at the graph for the UNC head this is clearly visible (fig 25). The increase in rendering time at brick sizes larger than 64^3 is again due to poorer cache coherency.

Looking at the times for the visible male, the reason for the increased rendering times for smaller bricks is due to the aforementioned overhead created by the large number of bricks being handled. Furthermore, this could partly explain why the best result is achieved when using no bricking at all.

In either case, this is the most powerful acceleration technique and can reduce rendering times to below 10% of the original ones, depending on how many cells are classified as visible. Note however that a few runs through the volume, from different directions, are needed in order for the cache to fill up. Rendering times for different brick dimensions when cell invisibility caching is employed is shown in figure 25.

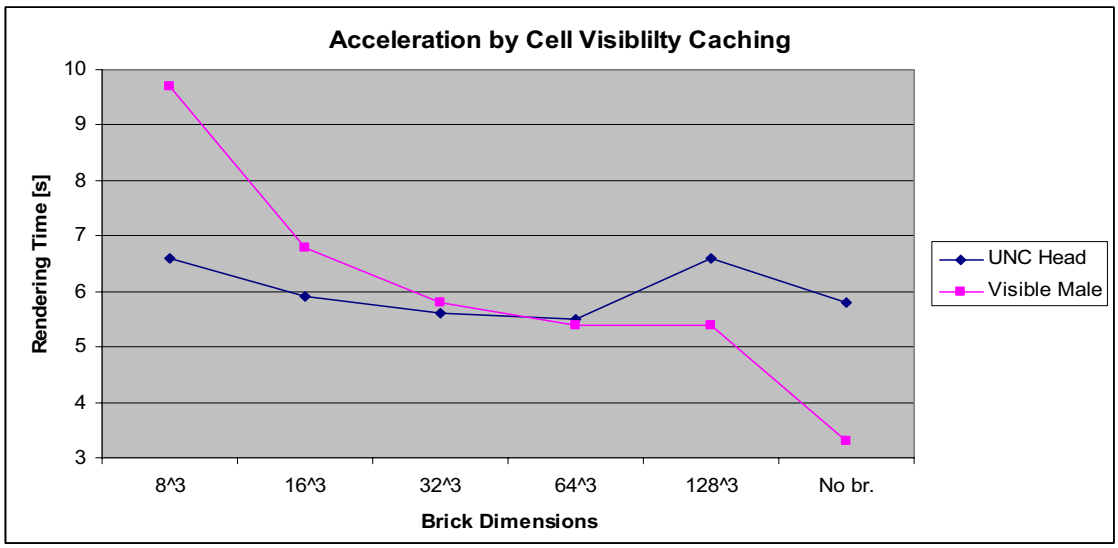


Figure 25 *- Rendering times for both data sets in relation to brick dimensions. Acceleration by cell visibility caching is employed.*

COMBINING ACCELERATION TECHNIQUES

A combination of all acceleration techniques show even further decreases in rendering times, at least for the UNC head. The lowest rendering time, 0.79s, was achieved with a brick size of 16^3. This is approximately 2% of the initial 36s.

For the visible male however, the improvements are minimal when combining the optimization functions. Compared to when using cell visibility caching without bricking they are actually worse. Due to this we tried a different acceleration configuration; without bricking but with cell visibility caching and early ray termination. This gave an average rendering time of only 2.5s, less than half of what was achieved with all the other techniques combined. As stated earlier, this could depend on a combination of the caching

not working and the large number of bricks used for data sets such as the visible male. In other words; instead of helping, the bricking and the brick dependant acceleration schemes might simply cause an overhead for large data sets.

Moreover, when comparing rendering times for the two data sets with full acceleration (fig 26) and with no acceleration at all (fig 19) one can see that the times are more varied in the first case. In other words, the acceleration techniques work better for smaller data sets.

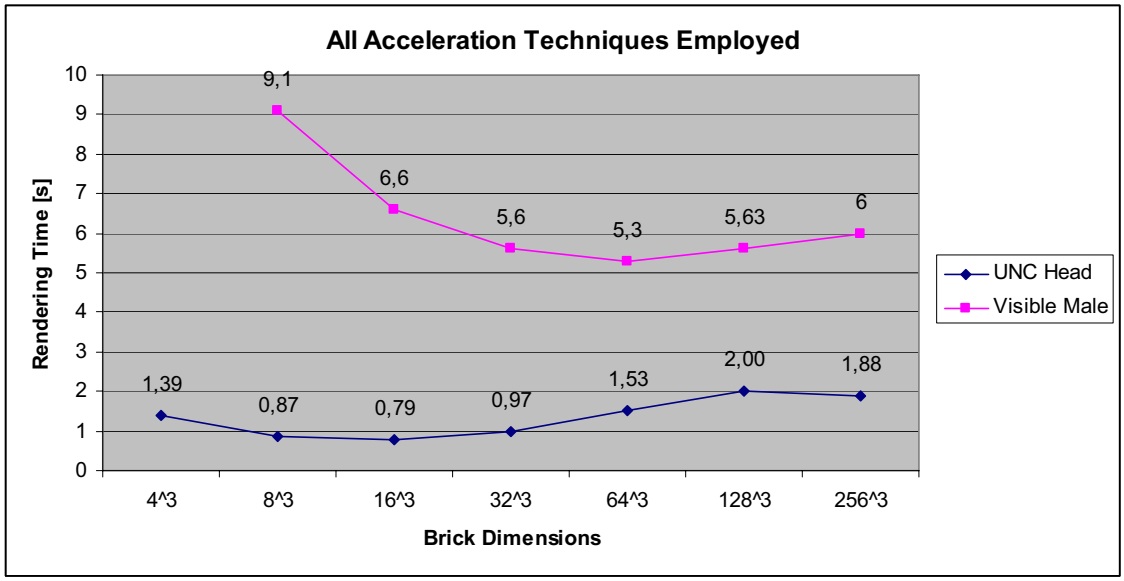


Figure 26 - *Rendering times for both data sets in relation to brick dimensions. All acceleration techniques are employed.*

7.1.4 MEMORY MANAGEMENT

The memory consumption (see fig 27) will only be evaluated for the visible male data set since the focus of this thesis is volume rendering of large data sets.

The visible male data set takes up roughly 855 MB. As long as staying within reasonable brick sizes, the extra memory consumption required by the system is roughly 100Mb, divided in the following manner:

- cell visibility cache ~ 53 Mb
- rays (640x480) ~ 17 Mb
- pixel map (640x480) ~ 12 Mb
- gradient cache (brick size of 64^3) ~ 3 Mb
- rest is occupied by GLUT, GLUI and system variables

However, when dealing with very large brick sizes (> 128) the memory consumption increases due to gradient storage. On the other hand, when dealing with very small brick sizes (< 16), the memory consumption increases due to the enormous number of bricks used. For instance, with a brick size of 8^3, the visible male uses over 800000 bricks, and even though the memory used for every brick is very small, together they form quite a

substantial increase. When initiating the system with a brick size of 4^3 the systems 2 Gb of RAM is no longer enough and a memory overflow exception is produced.

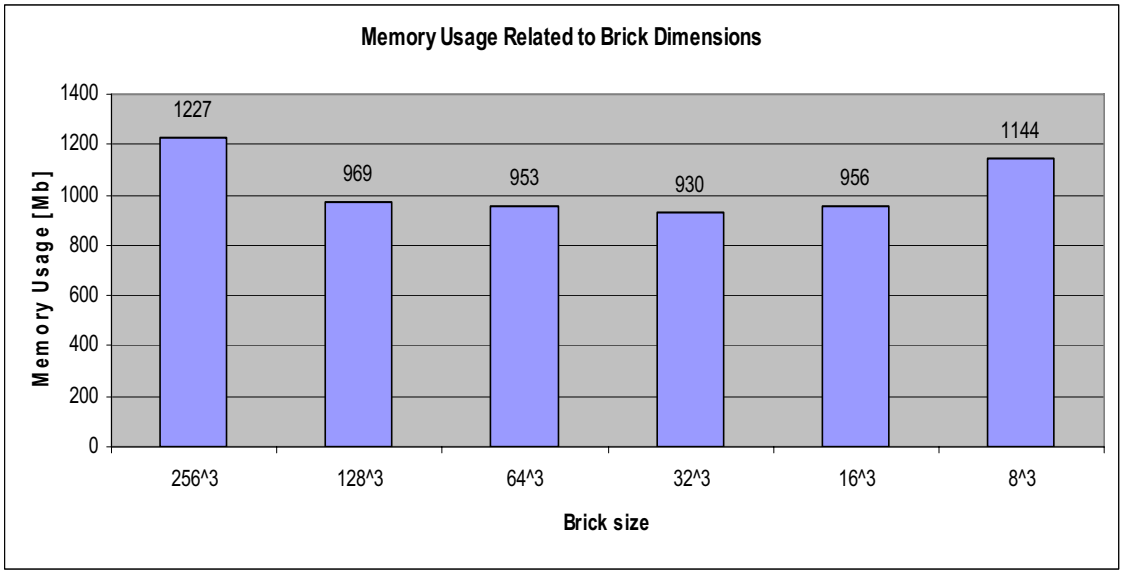


Figure 27 – *Memory usage in relation to brick dimensions for the visible male (855 Mb).*

7.2 LEVEL SET CLIPPING

In the work of this thesis, a simple framework for integrating level sets as clipping functions was implemented in order to provide a basis for testing of the clipping and shading algorithms. Simple geometries such as planes and spheres where used. The images in figure 25 are examples of level set clipping with a plane and a sphere respectively. High quality shading of the clipping interface, representing both the volume data and clipping geometry, was achieved by using methods described in section 5.3.

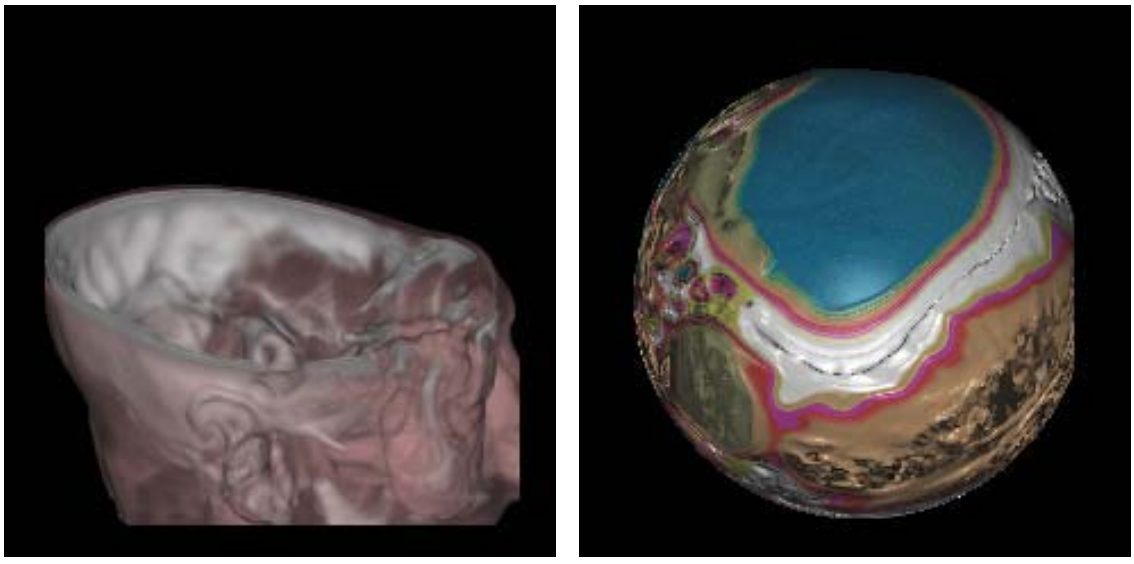


Figure 28 – *Level set clipping with plane (left) and sphere (right).*

For clipping with medical segmentation data (see fig 29), functions from the Graphics Group Library, GGL, were used. Principally, there is no difference between clipping with a sphere/plane or a medical segmentation, except in how the data is produced. A sphere is simply a function that can be created directly in our program while a medical segmentation has to be produced elsewhere, then loaded into the program and used as clipping data. All segmentations used in this thesis are produced by G. Johansson, Linköpings Tekniska Högskola.

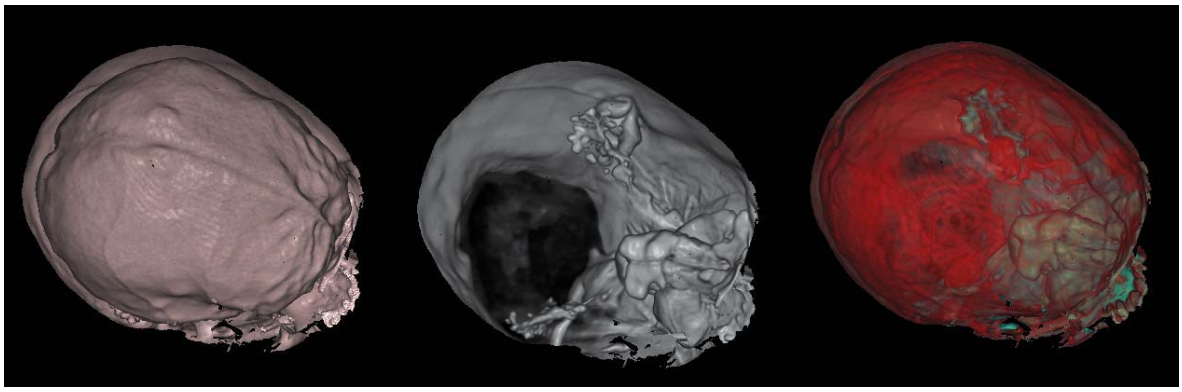


Figure 29 *- Volume renderings of the UNC head with different transfer functions. Level-set segmentation of skull used as clipping geometry. View from above. Segmentation data courtesy of Gunnar Johansson.*

7.2.1 ACCELERATION

As described in section 5.2, level sets should be very usable for leaping over empty space since they inherently hold distance information to the regions of interest. However, when using a narrow band scheme these possibilities are rather limited since the maximum distance one can leap without missing data is the width of the narrow band.

It has been quite hard to analyze the accelerating possibilities of the level sets. Partly due to lack of segmentation data, but mostly because it is often not possible to render the same data with transfer functions and level set clipping.

However, renderings performed with level set clipping have often taken longer than those without. This is probably due to the overhead produced for the interface shading and distance sampling. For suggestions of improvement and future development please see section 8.2.

7.2.2 MEMORY CONSUMPTION

One of the major advantages of using narrow-band level sets as clipping data is their memory efficient representation. The size of the segmentation data is not dependant of the actual volume size like many other representations, but is instead proportional to the segmentations surface area.

To illustrate the difference in memory requirements between regular and narrow-band level sets consider the segmentation used in figure 29 above. As a narrow-band LS

the segmentation requires roughly 14 Mb of memory. If the same segmentation would be represented as a regular level set it would take up more than 50 Mb. Actually, all segmentations would require this amount of memory, no matter how small the actual segmentation would be.

If dealing with larger data sets, these figures would vary even more. A regular LS segmentation would require approximately 1.3 Gb of memory, a figure not many systems would be able to handle when adding the size of the actual data set which also has to fit in memory. Thus, using narrow band level sets is almost a requirement when dealing with large data sets.

8 DISCUSSION AND FUTURE WORK

In this section, a brief evaluation of the methods described in this thesis will be made on the basis of the results presented in section 7. Future development and research is also suggested.

8.1 VOLUME RENDERING IN SOFTWARE

One objective of this thesis was to implement a volume rendering application in software that was capable of handling very large data-sets. Three major demands had to be considered:

- Image quality had to be first-class.
- Rendering times had to be acceptable.
- Memory overload had to be avoided.

Regarding image quality we believe that our system can compete very well with other applications, both software and hardware based. For a selection of renderings please refer to section 7.2.

As far as rendering time is concerned, software rendering can not compete with hardware implementation. With small data sets (~128^3), interactive frame rates can be achieved, for larger (~256^3) interactivity is made possible by reducing image quality during user interaction. For very large volumes, e.g. the visible male, a decent interaction is not possible with our system. The best rendering time for the visible male data set was 2.5 seconds for one frame, achieved when using cell visibility caching and early ray termination. If we could get the cache coherency to work properly we should see speed-ups by a factor of approximately 2.5 – 3.0 for the bricking, and even more for the acceleration techniques, which would greatly increase the usability of the application.

In regard to memory capacity, CPU implementation still has the upper hand. When rendering the visible male data set, our application uses about 950 Mb of memory, which is far more than most graphic cards can handle. It should be noted that in this aspect, graphic cards are beginning to catch up. For example, the NVIDIA's Geforce 7950 GX2 has 1GB of memory and it is probably just a matter of time before these memory levels, and beyond, become standard. Furthermore, in a master's thesis by P. Engström [12], an interesting framework for GPU-based raycasting utilizing bricking has been proposed and implemented to enable GPU-based volume rendering of large data sets.

Still, our software approach has many advantages. The system portability is high since no hardware or OS specific features are necessary. Another advantage is the prospect of multithreading on multi-core CPUs. Moreover, the possibilities of integrating higher-level data structures, in this case level sets, into the rendering process are much wider. The advent of programmable graphic processors and shading languages such as GLSL has broadened the possibilities of hardware implementation, but still features are highly product-dependent.

8.1.1 FUTURE DEVELOPMENT

There are still further efforts to be made to reduce rendering times of the application described in this thesis. First and foremost, the cache coherency must be made functional since it is the key for interactivity in software based renderers. A more efficient empty-space removal scheme using an entry-point buffer, created by brick- or octree projection as described in [17], would further decrease rendering times. Moreover, since the traversal algorithm described in this thesis is easy to adapt for multi-threading/parallelization, this could be implemented quite easily. Multi-threading is particularly interesting since computers with more than one CPU are emerging, e.g. the Duo processors from Intel.

Furthermore, in light of the recent developments of graphics card, it would be interesting to see what features could be made functional with newer versions of shader languages like the OpenGL Shader Language, Cg by Nvidia and HLSL by Microsoft.

Other possible extensions, not related to rendering times, would be to:

- Develop a more functional transfer function editor. The current one is rather limited and makes it difficult to visualize regions of interest.
- Add some type of data compression method to make possible visualization of even larger data sets than the visible male.
- Add exception handling and help sections to alleviate for users other than ourselves. Some bugs also remain to be fixed.
- Allow for pre-filtering of large data sets like the visible male. At the moment this is not possible due to that an additional volume data structure is used during filtering and memory limits are exceeded. This could be solved by either perform filtering brick-wise or by altering the interpolation scheme to perform filtering on the fly. One option would be the scheme proposed by Neumann et al. [21].

8.2 LEVEL SET INTEGRATION

Clipping using level sets could very well have a bright future ahead. As mentioned before, segmentations data in form of narrow-band level sets have many benefits compared to other types of clipping data, e.g. memory efficient representation and inherent distance information. The question is how to fully take advantage of these benefits, in particular the distance information. One way could be to send a few probing rays through the volume, to acquire some type of template of the object that is to be rendered, before complete rendering. Related work has for instance been carried out by Yagel and Kaufmann [41].

Another exciting possibility would be to extend the level set capabilities of our volume renderer to, for instance, allow for segmentation. The easiest way would probably be to merge the application with an already existing framework, e.g. the one created by G. Johansson [42]. In addition, exploring the possibilities of altering the rendering system to allow for GPU acceleration would be very interesting.

8.2 CLOSING STATEMENT

In this thesis, we have investigated the possibilities of utilizing level-set segmentation data in software volume rendering framework. We believe that the integration of the two techniques can greatly enhance the usability of volume rendering in the future. Efficient ways of extracting and visualizing structures of interest is a major factor in making volume rendering truly useful in real applications, rather than being just visually pleasing. CPUs are constantly getting cheaper and faster and the possibilities of multi-threading are promising. Therefore, software volume rendering with level-set integration might very well be the technique that takes volume rendering to the next level.

REFERENCES

1. Levoy, M. *Display of Surfaces from Volume Data*, IEEE Computer Graphics and Applications. 8(5): 29-37, 1988.

2. Levoy, M., *Volume Rendering by Adaptive Refinement.* The Visual Computer, 6(1): 2-7, 1990.

3. H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. *The volume pro real-time ray-casting system.* In Proceedings of ACM SIGGRAPH'99. pp. 251-260, 1990.

4. D. Weiskopf, K. Engel, and T. Ertl. *Interactive Clipping Techniques for Texture Based Volume Visualization and Volume Shading.* IEEE Transactions on Visualization and Computer Graphics. 9(3): 298-313, 2003.

5. C. D. Hansen, C. R. Johnson. *The Visualization Handbook.* ISBN: 012387582X. Elsevier Butterworth Heinemann, 2004.

6. R. Whitaker, D. Breen, K. Museth, and N. Soni. *Segmentation of biological volume datasets using a level set framework.* Volume Graphics, 2001. pp. 249-263.

7. D. Peng, B. Merriman, S. Osher, H.K. Zhao, M. Kang. *A PDE-based fast local level set method.* J. Comput. Phys. 155(2): 410-438, 1999.

8. M. Levoy. *Efficient Ray Tracing of Volume Data.* ACM Transactions on Graphics. 9(3): 245-261, 1990

9. The GLSL homepage - http://www.opengl.org/documentation/glsl/

10. The Microsoft DirectX Development Centre - http://msdn.microsoft.com/directx/

11. The Cg homepage - http://developer.nvidia.com/page/cg_main.html

12. P. Engström - Master's Thesis: *Interactive GPU-Based Volume Rendering.* Linköpings Tekniska Högskola, 2006.

13. G. Knittel and W. Strasser. *A Compact Volume Rendering Accelerator.* Volume Visualization Symposium Proceedings, pp 67-74, 1994.

14. G. Knittel. *The ULTRAVIS System.* In Proc. Of Volume Visualization and Graphics Symposium '00, pp 71-80, 2000.

15. T. Guenther, C. Poliwoda, C. Reinhard, J. Hesser, R. Maenner, H. Meinzer and H. Baur. *VIRIM: A Massively Parallel Processor for Real-Time Volume Visualization in Medicin.* Proc. of the 9th Eurographics Hardware Workshop, pp 103-108, 1994.

16. M. Meissner, U, Kanus, G. Wetekam, J. Hirche, A. Ehlert, W. Strasser, M. Doggett, P. Forthmann, and R. Proksa. *Vizard II, a reconfigurable interactive volume rendering system*. In the proceeding of the Eurographics Workshop on Graphics Hardware, pp 137-146, 2002.

17. S. Bruckner. Matser's Thesis: *Efficient Volume Visualization of Large Medical Data Sets*. Vienna University of Technology, 2004.

18. S.Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. *Memory Efficient Acceleration Structures and Techniques for CPU-based Volume Raycasting of Large Data*. In the proceeding of the IEEE Symposium on Volume Visualization and Graphics 2004 (VV'04).

19. S. Grimm, S. Bruckner, A. Kanitsar, and E. Gröller. *A refined data addressing and processing scheme to accelerate volume raycasting*. Computers & Graphics, 28(5): 719-29, 2004.

20. P. Lacroute, M. Levoy. *Fast Volume Rendering Using a Shear-Warp Factorization of the Viewing Transformation*. SIGGRAPH '94, pp 451-8, 1994.

21. L. Neumann, B. Csebfalvi, A. König, and E. Gröller. *Gradient Estimation in Volume Data Using 4D Linear Regression*. In Proceedings of Eurographics, pp 351-358, 2000.

22. P. Bui-Tuong. *Illumination for computer Generated Pictures*. CACM '75, pp 311-317, 1975.

23. M. W. Jones. *Acceleration Techniques for Volume Rendering*.

24. K. Zuidervald, A. Koning, and M. Viergever. *Acceleration of Ray-Casting using 3D Distance Transforms*. Visualization in Biomedical Computing '92, pp 324-335, 1992.

25. D. Cohen and Z. Shefer. *Proximity Clouds: An Acceleration Technique for 3D Grid Traversal*. The Visual Computer, 10(11): 27-38, 1994.

26. M. Sramek and A. Kaufman. *Fast Ray-Tracing of Rectilinear Volume Data Using Distance Transforms*. IEEE Transactions on Visualization and Computer Graphics, 3(6): 236-252, 2000.

27. M. Wan, Q. Tang, A. Kaufman, Z. Liang, and M. Wax. *Volume Rendering Based Interactive Navigation Within the Human Colon*. In the Proceedings IEEE Visualization '99, pp 397-400, 1999.

28. M. Meissner, M. Doggett, U. Kanus, and J. Hirche. *Efficient Space Leaping for Ray-Casting Architectures*. Proc. of the 2nd Workshop on Volume Graphics, 2001.

29. L. Sobierajski and R. Avila. *A Hardware Acceleration Method for Volumetric Ray Tracing*. Proc. of IEEE Visualization '95, pp 27-35, 1995.

30. D. Meagher. *Geometric Modelling Using Octree Encoding*. Computer Graphics and Image Processing, 19(2): 129-147, 1982.

31. M. Levoy. *Efficient Ray Tracing of Volume Data*. ACM Transactions on Graphics, 9(3): 245-261, 1990.

32. B. Mora, J. Jessel, and R. Caubet. *A New Object Order Ray-Casting Algorithm*. In Proc. of Visualization '02, pp 107-113, 2002.

33. A. Law and R. Yagel. *Multi-Frame Trashless Ray Casting With Advancing Ray-Front*. In Proceedings of Graphics Interfaces '96, pp 70-77, 1996.

34. W. Lorensen, H. Cline. *Marching Cubes: A High Resolution 3D Surface Construction Algorithm*, *SIGGRAPH '87*, 1987.

35. Stanley Osher and James A. Sethian, *Fronts propagating with curvature-dependent speed: Algorithms based on Hamilton-Jacobi formulations*. Journal of Computational Physics 79, pp 12–49, 1988.

36. Danping Peng, Barry Merriman, Stanley Osher, Hong-Kai Zhao, and Myungjoo Kang, *A pde-based fast local level set method*, Journal of Computational Physics 155(2): 410-438, 1999.

37. Michael B. Nielsen and Ken Museth. *Dynamic tubular grid: An efficient data structure and algorithms for high resolution level sets*, Journal of Scientific Computing, pp 1–39, 2006.

38. The OpenGL homepage – http://www.opengl.org/

39. OpenGL Libraries – including GLUT - http://www.opengl.org/resources/libraries/

40. The Graphics Group – http://www.gg.itn.liu.se/

41. R. Yagel, A. Kaufmann. *Template based volume viewing*. Computer Graphics Forum, 11(3): 153-167, 1992

42. G. Johansson. Master's Thesis: *Local Segmentation*. Linköpings Tekniska Hörgskola, 2006.

43. Department of Computer Science, University of North Carolina.

44. The Visible Human Project, National Library of Medicine.

45. E. N. Mortensen, B. S. Morse, W. A. Barrett. *Adaptive Boundary Detection Using 'Live Wire'. Two-dimensional Dynamic Programming*. IEEE Proceedings of Computer in Crdiology (CIC'92). pp. 635-38, 1992.

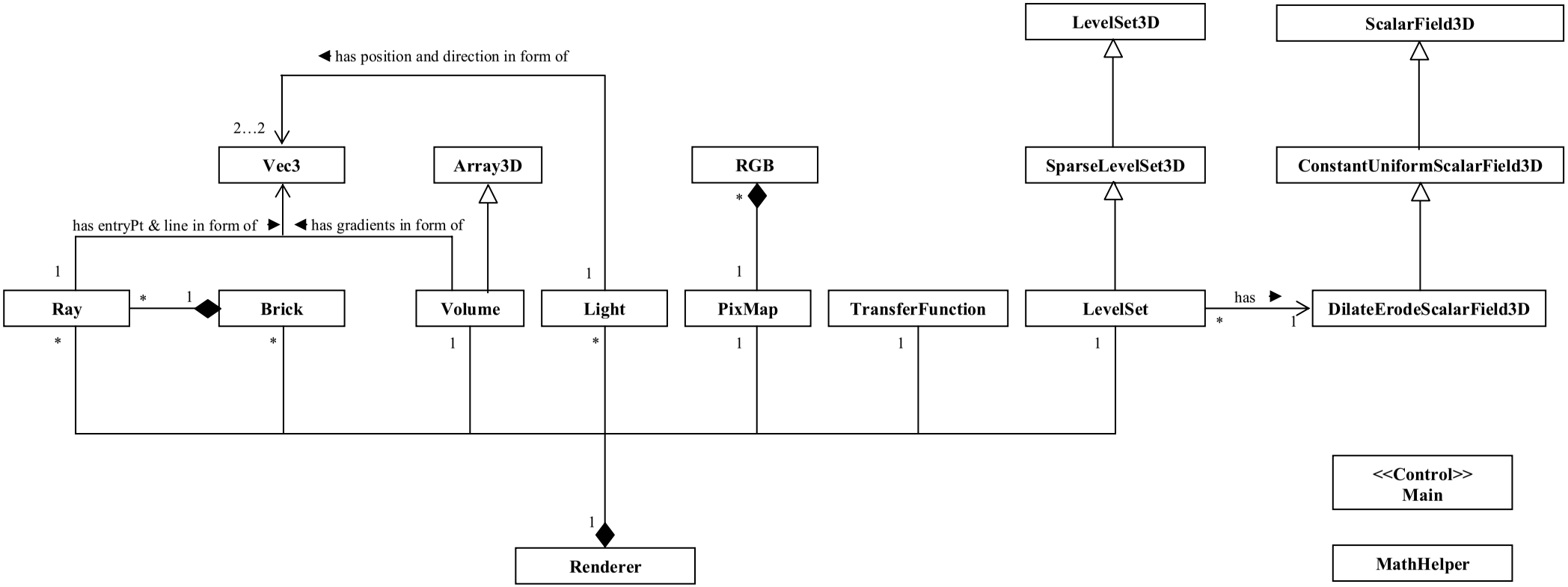
46. M. Kass, A. Witkin, and D. Terzopoulos. *Snakes: Active Contour Models*. International Journal of Computer Vision. 1: 321-32, 1988.

47. The SCI Institute - http://software.sci.utah.edu/doc/User/Tutorials/BioImage/Figures/editTFWindow.jpg

48. Max, Nelson, *Optical Models for Direct Volume Rendering,* IEEE Transactions on Visualization and Computer Graphics, 1(2), 1995.

APPENDIX 1

CLASS DIAGRAMS



Brick

```
cvc : vector<bool>
max : Scalar
min : Scalar
rayList : list<*Ray>
```

```
Brick()
Brick(nrOfElements : int)
~Brick()
init(nrOfElements : int)
resetCVC(nrOfElements : int)
getMax() : Scalar
getMin() : Scalar
getNrOfRays() : int
getFirstRay() : Ray*
setMax(max : Scalar)
setMin(min : Scalar)
insertRay(r : Ray*)
removeRay()
getCellVisibility(internalIndex : int) : bool
setCellVisibility(internalIndex : int, b : bool)
```

LevelSet

```
dim : int[3]
```

```
LevelSet(sf : ScalarField*, dim : const unsigned int[3],
         f : Function3D<Real>*, initParams : InitParams,
         gridInitParams : typename Grid::InitParams)
LevelSet(dim : int*, sf : ScalarField*, phi : Grid*,
         initParams : InitParams)
init()
erode(iterations : int)
dilate(iterations : int)
open(iterations : int)
close(iterations : int)
getSF() : MyScalarField*
computeDistToLS(xv : int, yv : int, zv : int,
                xi : float, yi : float, zi : float) : float
computeLSGridGradient(x : int, y : int, z : int) : Vec3
computeLSGradient(xv : int, yv : int, zv : int,
                  xi : float, yi : float, zi : float): Vec3
```

DilateErodeScalarField3D

```
setToErode()
setToDilate()
getValue() : int
```

Vec3

x : float
y : float
z : float

```
Vec3()
Vec3(x : float, y : float, z : float)
Vec3(v : const Vec3&)
dot(v : const Vec3&) : float
cross(v1 : const Vec3&, v2 : const Vec3&)
cross(v : const Vec3&)
cross(v1 : const Vec3&, v2 : const Vec3&) : Vec3
xformVec(v : const Vec3&, m : const Matrix4&)
xformVec(m : const Matrix4&)
xformVec(v : const Vec3&, m : const Matrix4&) : Vec3
xformPt(v : const Vec3&, m : const Matrix4&)
xformPt(m : const Matrix4&)
xformPt(v : const Vec3&, m : const Matrix4&) : Vec3
rotate(deg : float, axis : const Vec3&)
set(X : float, Y : float, Z : float)
clear()
sqrDistance(v : const Vec3&) : float
distance(v : const Vec3&) : float
almostEqual(v : const Vec3&, tol : float ) : bool
vecAbs()
Vec3 fabs(v : const Vec3& )
setLength(l : float)
lengthSquared() : float
length() : float
square()
normalize() : float
normalize(v : const Vec3&) : float
lerp(t : float, v0 : const Vec3&, v1 : const Vec3&)
scale(v : const Vec3&)
operator ==(v : const Vec3&) : bool
operator ==(f : float ) : bool
operator !=(v : const Vec3&) : bool
operator !=(f : float ) : bool
operator -() : Vec3
operator +(f : float, v : const Vec3&) : Vec3
operator +(v : const Vec3&, f : float) : Vec3
operator +(v1 : const Vec3&, v2 : const Vec3&) : Vec3
operator -(f : float, v : const Vec3&) : Vec3
operator -(v : const Vec3&, f : float) : Vec3
operator -(v1 : const Vec3&, v2 : const Vec3&) : Vec3
operator *(s : float, v : const Vec3&) : Vec3
operator *(v : const Vec3&, s : float) : Vec3
operator *(v1 : const Vec3&, v2 : const Vec3&) : Vec3
operator /( s : float, v : const Vec3&) : Vec3
operator /( v : const Vec3&, s : float) : Vec3
operator /( v1 : const Vec3&, v2 : const Vec3&) : Vec3
operator =(v : const Vec3&) : Vec3&
operator *=(s : float) : Vec3&
operator *=(v : const Vec3&) : Vec3&
operator /=(s : float) : Vec3&
operator /=(v : const Vec3&) : Vec3&
operator +=(v : const Vec3&) : Vec3&
operator +=(s : float) : Vec3&
operator -=(v : const Vec3&) : Vec3&
operator -=(s : float) : Vec3&
```

Matrix4

mat : float[4][4]

```
Matrix4()
Matrix4(m00 : float, m01 : float, m02 : float, m03 : float,
        m10 : float, m11 : float, m12 : float, m13 : float,
        m20 : float, m21 : float, m22 : float, m23 : float,
        m30 : float, m31 : float, m32 : float, m33 : float)
Matrix4(m : const Matrix4&)
setRow(row : int, x : float, y : float, z : float, w : float)
setCol(col : int, x : float, y : float, z : float, w : float)
getRow(row : int, x : float*, y : float*, z : float*, w : float*)
getCol(col : int, x : float*, y : float*, z : float*, w : float*)
fill(val : float)
copy(m : const Matrix4&)
makeIdent()
makeTrans(x : float, y : float, z : float)
makeScale(x : float, y : float, z : float)
makeRot(deg : float, axis_x : float, axis_y : float, axis_z : float)
makeRot(deg : float, axis : const Vec3&)
makeVecRotVec(v1 : const Vec3&, v2 : const Vec3&)
transpose(m : const Matrix4&)
transpose()
preMult(m : const Matrix4&)
postMult(m : const Matrix4&)
preTrans(x : float, y : float, z : float, m : const Matrix4&)
preTrans(x : float, y : float, z : float)
postTrans(m : const Matrix4&, x : float, y : float, z : float)
postTrans(x : float, y : float, z : float)
preRot(deg : float, axis_x : float, axis_y : float, axis_z : float, m : const Matrix4&)
preRot(deg : float, axis_x : float, axis_y : float, axis_z : float)
postRot(m : const Matrix4&, deg : float, axis_x : float, axis_y : float, axis_z : float)
postRot(deg : float, axis_x : float, axis_y : float, axis_z : float)
preScale(x : float, y : float, z : float, m : const Matrix4&)
preScale(x : float, y : float, z : float)
postScale(m : const Matrix4&, x : float, y : float, z : float)
postScale(x : float, y : float, z : float)
orthoNormalize()
invert(M : const Matrix4&) : float
invert() : float
determinant( M : const Matrix4&) : float
determinant() : float
operator [](i : int) : float*
operator [](i : int) : const float*
operator ==(m : const Matrix4&) : bool
operator =(m : const Matrix4&) : Matrix4&
operator -() : Matrix4
operator +(m1 : const Matrix4&, m2 : const Matrix4&) : Matrix4
operator -(m1 : const Matrix4&, m2 : const Matrix4&) : Matrix4
operator +=(m : const Matrix4&) : Matrix4&
operator -=(m : const Matrix4&) : Matrix4&
operator *(m1 : const Matrix4&, m2 : const Matrix4&) : Matrix4
operator /(m1 : const Matrix4&, m2 : const Matrix4&) : Matrix4
operator *=(m : const Matrix4&) : Matrix4&
operator /=(m : const Matrix4&) : Matrix4&
```

Renderer

```
myVolume : Volume<Scalar>*
myPM : PixMap*
myTF : TransferFunction<Scalar>*
bricks : Brick<Scalar>*
myRays : Ray*
light0 : Light*
myLS : LevelSetType*
viewCaseLists : list<int>[8]
```

```
Renderer()
Renderer(fileName : char*, xSize : int, ySize : int, zSize : int,
         bDimX : int, bDimY : int, bDimZ : int,
         xSpacing : float, ySpacing : float, zSpacing : float,
         pmSize : int, bricking : bool)
~Renderer()
render()
pixelInterpolation(col : int, row : int, wMax : int)
computeSampleColor(gradient : const Vec3, sampleValue : float,
                   r : float&, g : float&, b : float&)
getPixMap() : Pixap*
getVolume() : Volume<Scalar>*
getTF() : TransferFunction<Scalar>*
getLS() : LevelSetType*
setLS(ls : LevelSetType*)
setRenderingOptions(triLerp : bool, step : float, projection : int,
                    previewCoarseness : int, earlyRayTermination : bool,
                    transpTresh : float, clip : int, interactive : bool,
                    viewMatrix : const Matrix4&, gradModOn : bool,
                    gradModFactor : float, gradientCaching : bool, cvc : bool,
                    skipBricks : bool)
initRay(col : int, row : int)
createBrickLists()
getViewCase(rayDir : const Vec3&) : int
processBricks()
processRay(blockIndex : int, ray : Ray*, t : float) : bool
isCellVisible(xv : int, yv : int, zv : int, brickindex : int) : bool
isBrickVisible(brickIndex : int) : bool
skipBrick(brickIndex : int, ray : Ray*) : float
resetCVCs()
initLS(pathName : char*)
```

TransferFunction

```
maxValue : Scalar
myColorTable : float*
```

```
TransferFunction(maxValue : Scalar)
~TransferFunction()
clearTF()
addTFSegment(a : float, b : float, c : float, d : float,
             R : float, G : float, B : float, A : float)
saveTF(fileName : char*)
loadTF(fileName : char*)
getR(sampleValue : float) : float
getG(sampleValue : float) : float
getB(sampleValue : float) : float
getOpacity(sampleValue : float) : float
getOpacity(sampleValue : int) : float
getMaxValue() : Scalar
```

RGB

```
r : unsigned char
g : unsigned char
b : unsigned char
a : unsigned char
```

```
RGB()
RGB(R : unsigned char, G : unsigned char, B : unsigned char )
RGB(R : unsigned char, G : unsigned char, B : unsigned char, A : unsigned char)
operator *(s : float ) : RGB
isBlack() : bool
operator *(d : float, rgb : const RGB&) : RGB
operator +(rgb1 : const RGB&, rgb2 : const RGB&) : RGB
```

Ray

```
entryPt : Vec3
line : Vec3
opacity : float
r : float
g : float
b : float
t : float
rayLength : float
col : int
row : int
```

```
Ray()
Ray(entryPt : const Vec3&, line : const Vec3&)
init(entryPt : const Vec3&, line : const Vec3&,
     rayLength : float, col : int, row : int)
getDone() : bool
getColor() : RGB
getR() : float
getG() : float
getB() : float
getOpacity() : float
getMaxSampValue() : float
getCol() : int
getRow() : int
getT() : float
getRayVectors(entryPt : Vec3&, line : Vec3&)
setOpacity(o : float)
setToBlack()
setMaxSampValue(max : float)
setT(t : float)
accColor(r : float, g : float, b : float)
getCurrPos(v : Vec3&)
advanceRay()
advanceRay(distToLS : float)
```

Light

```
pos : Vec3
dir : Vec3
r : float
g : float
b : float
```

```
Light()
Light(pos : Vec3, dir : Vec3, r : float, g : float, b : float)
setDirection(x : float, y : float, z : float)
setPosition(x : float, y : float, z : float)
setColor(R : float, G : float, B : float)
getDirection() : Vec3
getR() : float
getG() : float
getB() : float
```

MathHelper

```
trilerp(x : float, y : float, z : float, cornerValues : float*) : float
trilerp(x : float, y : float, z : float, cornerValues : int*) : float
volumeBoxIntersect(b1x : const float, b1y : const float, b1z : const float,
           b2x : const float, b2y : const float, b2z : const float,
origin : const Vec3*, direction : const Vec3*,         tnear : float*, tfar
: float*) : bool
smoothStep(a : float, b : float, x : float) : float
ramp(a : float, b : float, c : float, d : float, x : float) : float
max(a : float, b : float) : float
min(a : float, b : float) : float
roundToInt(a : float) : int
log2(s : int) : int
log2(s : float) : float
```

PixMap

```
nRows : int
nCols : int
pixelRGBs : RGB*
interpolated : bool*
rayCasted : bool*
zoomFactor : float
```

```
PixMap()
PixMap(c : int, r : int)
~PixMap()
getPixelColor(x : int, y : int) : RGB*
getCols() : int
getRows() : int
getZoomFactor() : float
setPixelColor(x : int, y : int, color : RGB)
setPixels(c : int, r : int, blockSize : int, color : RGB)
draw(winWidth : int, winHeight : int)
clearNonRaycasted(color : RGB&)
clearRaycasted(color : RGB&)
clearAll(color : RGB&)
resetFlags()
resetIFlags()
isRaycast(x : int, y : int) : bool
isInterpolated(x : int, y : int) : bool
setToRayCasted(x : int, y : int)
setToInterpolated(x : int, y : int)
setZoomFactor(z : float)
computeColorRange(x : int, y : int, bS : int) : float
```

Volume

```
vDimX : unsigned int
vDimY : unsigned int
vDimZ : unsigned int
voxelBitSize : unsigned int
xSpacing : float
ySpacing : float
zSpacing : float
nrOfElements : int
nrOfBricks : int
brickSize : unsigned int
bDimX : unsigned int
bDimY : unsigned int
bDimZ : unsigned int
bDimXLog2 : int
bDimYLog2 : int
bDimZLog2 : int
bvDimX : int
bvDimY : int
bvDimZ : int
offsetLUT : int[27][27]
gradientCache : Vec3*
gradientComputed : bool*
gcSize : int
```

```
Volume()
Volume(voxelBitSize : int, vDimX : int, vDimY : int, vDimZ : int,
       bDimX : int, bDimY : int, bDimZ : int,
       xSpacing : float, ySpacing : float, zSpacing : float, bricking : bool)
~Volume()
readBinFile(fileName : char*)
boxFilter()
getVoxelValues(xv : int, yv : int, zv : int, cornerValues : int*, brickIndex : int)
computeSampleValue(xv : int, yv : int, zv : int, xi : float, yi : float, zi : float, brickIndex : int) : float
computeVoxelGradient(x : int, y : int, z : int) : Vec3
getVoxelGradient(x : int, y : int, z : int, gcOffset : int, v : Vec3&)
computeSampleGradient(xv : int, yv : int, zv : int, xi : float, yi : float, zi : float,
                      gradientCaching : bool) : Vec3
computeBrickMaxMin(brickIndex : int, brickMax : Scalr&, brickMin : Scalar&)
computeSubSpace(I : int, j : int, k : int) : int
createOffsetLUTs()
getVolumeData() : Array3D<Scalar>*
getXDim() : unsigned int
getYDim() : unsigned int
getZDim() : unsigned int
getXSpacing() : float
getYSpacing() : float
getZSpacing() : float
getVoxelBitSize() : unsigned int
getBvDimX() : unsigned int
getBvDimY() : unsigned int
getBvDimZ() : unsigned int
getBDimX() : unsigned int
getBDimY() : unsigned int
getBDimZ() : unsigned int
setXDim(xS : int)
setYDim(yS : int)
setZDim(zS : int)
setXSpacing(xSpac : float)
setYSpacing(ySpac : float)
setZSpacing(zSpac : float)
setVoxelBitSize(vSize : int)
setGradientComputed(i : int, b : bool)
scaledFloor(xs : float, ys : float, zs : float, xv : int&, yv : int&, zv : int&)
scaledRound(xs : float, ys : float, zs : float, xv : int&, yv : int&, zv : int&)
getVoxelIndex(x : int, y : int, z : int) : int
getVoxelValue(index : unsigned int) : Scalar
getVoxelValue(x : int, y : int, z : int) : Scalar
getBrickIndex(x : int, y : int, z : int) : int
getInternalBrickIndex(x : int, y : int, z : int) : int
getGradientCacheIndex(x : int, y : int, z : int) : int
getNrOfBricks() : int
getBrickSize() : int
getMemUsed()
```

APPENDIX 2

RENDERING TIMES[1] USING VARIOUS BRICK DIMENSIONS AND ACCELERATION TECHNIQUES

Rendering times for the UNC head [s]

Brick Dim.	No acceleration	Early ray termination	Gradient caching	Brick Skipping	Cell Visibility Caching	GC + BS + CVC	All
No Bricking	36,2	20,8			5,8		
128^3	40,7	23,5	23,8	40,7	6,6	5,3	2,00
64^3	39,8	22,7	23,8	28,4	5,5	3,8	1,53
32^3	40,2	23,1	24,1	16,1	5,6	3,1	0,97
16^3	40,7	23,3	24,5	9,8	5,9	2,8	0,79
8^3	41,4	23,7	25,6	6,4	6,6	3,0	0,87

Rendering times for the visible male [s]

Brick Dim.	No acceleration	Early ray termination	Gradient caching	Brick Skipping	Cell Visibility Caching	GC + BS + CVC	All
No Bricking	41,2	35,5			3,3		
128^3	47,2	40,4	53,2	27,2	5,4	6,3	5,63
64^3	38,6	33,1	38,8	13,3	5,4	6,1	5,3
32^3	38,6	33,1	37,2	9,6	5,8	6,5	5,6
16^3	39,6	34,2	38,4	9,1	6,8	7,8	6,6
8^3	43,1	37,5	41,8	11,3	9,7	10,8	9,1

[1]By rendering time we refer to how long it takes to render one frame, in seconds.