High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading

Klaus Engel, Martin Kraus, Thomas Ertl *

Visualization and Interactive Systems Group, University of Stuttgart, Germany

Abstract

We introduce a novel texture-based volume rendering approach that achieves the image quality of the best post-shading approaches with far less slices. It is suitable for new flexible consumer graphics hardware and provides high image quality even for low-resolution volume data and non-linear transfer functions with high frequencies, without the performance overhead caused by rendering additional interpolated slices. This is especially useful for volumetric effects in computer games and professional scientific volume visualization, which heavily depend on memory bandwidth and rasterization power.

We present an implementation of the algorithm on current programmable consumer graphics hardware using multi-textures with advanced texture fetch and pixel shading operations. We implemented direct volume rendering, volume shading, arbitrary number of isosurfaces, and mixed mode rendering. The performance does neither depend on the number of isosurfaces nor the definition of the transfer functions, and is therefore suited for interactive high-quality volume graphics.

CR Categories: I.3.3 [Computer Graphics]: Picture/Image Generation, I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling, I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism.

Keywords: direct volume rendering, volume graphics, volume shading, volume visualization, multi-textures, rasterization, PC graphics hardware, flexible graphics hardware

1 Introduction

In spite of recent progress in texture-based volume rendering algorithms, volumetric effects and visualizations have not reached the mass market. One of the reasons is the requirement for extremely high rasterization power caused by non-linear transfer functions needed for convincing volume visualizations and striking volumetric effects. Thus, new algorithms have to be developed that produce high-quality images with less rasterization and therefore higher frame rates on modern consumer graphics hardware. Traditionally, these two goals oppose each other, because high image quality requires to render additional trilinearly interpolated slices at the expense of rasterization power [11].

In order to overcome these limitations, we generalize in Section 3 the cell-projective rendering algorithm published by our group in [12]. For texture-based approaches, this method, called pre-integrated volume rendering, allows us to avoid additional slices by integrating non-linear transfer functions in a preprocessing step. An abstract description of this algorithm for object- and view-aligned textured slices is presented in Section 4, while the technical details of an implementation on current programmable consumer graphics hardware are described in Section 5. In particular, we discuss the use of advanced texture fetch and pixel shading operations recently proposed by graphics hardware vendors [4]. These features are exploited in order to achieve direct volume rendering, multiple smoothly shaded isosurfaces, and volume shading. Preliminary results on a *GeForce3* graphics hardware are presented in Section 6. Finally, Section 7 sums up the paper.

2 Related Work

High accuracy in direct volume rendering is usually achieved by very high sampling rates resulting in heavy performance losses. However, for cell-projective techniques Max, Williams, and Stein have proposed elaborated optical models and efficient, highly accurate projective methods in [8, 14]. The latter were further improved by Röttger, Kraus, and Ertl in [12]. Although these techniques were initially limited to cell projection, we were able to generalize them in order to apply these ideas to texture-based rendering approaches.

The basic idea of using object-aligned textured slices to substitute trilinear by bilinear interpolation was presented by Lacroute and Levoy [6], although the original implementation did not use texturing hardware. For the PC platform, Brady et al. [2] have presented a technique for interactive volume navigation based on 2D texture mapping.

The most important texture-based approach was introduced by Cabral [3], who exploited the 3D texture mapping capabilities of high-end graphics workstations. Westermann and Ertl [13] have significantly expanded this approach by introducing a fast direct multi-pass algorithm to display shaded isosurfaces. Based on their implementation, Meißner et al. [9] have provided a method to enable diffuse illumination for semi-transparent volume rendering. However, in this case multiple passes through the rasterization hardware led to a significant loss in rendering performance. Dachille et al. [5] have proposed an approach that employs 3D texture hardware interpolation together with software shading and classification.

One direction in PC graphics is the development of special purpose volume rendering hardware, e.g. VolumePro [10]. In contrast to this, consumer graphics hardware is becoming programmable. We presented techniques for using *NVidia's* register combiner OpenGL extension for fast shaded isosurfaces, interpolation, speedup, and volume shading [11]. As the programming of flexible units using OpenGL extensions of hardware manufacturers is complex and difficult, a higher abstraction layer has been proposed in the form of a real time procedural shading language system [7], which will increase productivity and make programs more portable.

3 Theoretical Background

This section generalizes and formalizes the rendering techniques proposed in [12]. This theoretical framework is not required on first reading, although several equations in this section are referenced in following sections.

*Abteilung für Visualisierung und Interaktive Systeme,
Breitwiesenstr. 20-22, 70565 Stuttgart, Germany,
Email: {engel,kraus,ertl}@informatik.uni-stuttgart.de

Permission to make digital or hard copies of part or all of this work or personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee.
HWWS '01 Los Angeles, CA USA
© ACM 2001 1-58113-407-X…$5.00

3.1 Direct Volume Rendering

Since its introduction in the late 1980s, direct volume rendering has diverged into several more or less distinct techniques, which are roughly classified as *image-based* (or *backward projective*), e.g., ray-casting, and *object-based* (or *forward projective*), e.g., cell projection, shear-warp, splatting, or texture-based algorithms. The common theme, however, is an (approximate) evaluation of the *volume rendering integral* for each pixel, i.e., the integration of attenuated colors and extinction coefficients along each viewing ray. We assume that the viewing ray $\mathbf{x}(\lambda)$ is parametrized by the distance λ to the viewpoint, and that color densities $\mathrm{color}(\mathbf{x})$ together with extinction densities $\mathrm{extinction}(\mathbf{x})$ may be calculated for any point in space $\mathbf{x}$. (The units of color and extinction densities are color intensity per length and extinction strength per length, respectively. However, we will refer to them as colors and extinction coefficients when the precise meaning is clear from the context.) Then the volume rendering integral is

$$I = \int_0^D \mathrm{color}\big(\mathbf{x}(\lambda)\big) \exp\left(-\int_0^\lambda \mathrm{extinction}\big(\mathbf{x}(\lambda')\big)\mathrm{d}\lambda'\right)\mathrm{d}\lambda$$

with the maximum distance D, i.e., there is no color density $\mathrm{color}\big(\mathbf{x}(\lambda)\big)$ for λ greater than D. In words, color is emitted at each point $\mathbf{x}$ according to the function $\mathrm{color}(\mathbf{x})$, and attenuated by the integrated extinction coefficients $\mathrm{extinction}(\mathbf{x})$ between the viewpoint and the point of emission.

Unfortunately, this form of the volume rendering integral is not useful for the visualization of a continuous scalar field $s(\mathbf{x})$, because the calculation of colors and extinction coefficients is not specified. We distinguish two steps in the calculation of these colors and extinction coefficients: the *classification* is the assignment of a *primary color* and an extinction coefficient. (The term *primary color* is borrowed from OpenGL terminology in order to denote the color *before* shading.) The classification is achieved by introducing transfer functions for color densities $\tilde{c}(s)$ and extinction densities $\tau(s)$, which map scalar values $s = s(\mathbf{x})$ to colors and extinction coefficients. (In general, $\tilde{c}$ is a vector specifying a color in a color space, while τ is a scalar extinction coefficient.)

The second step is called *shading* and calculates the color contribution of a point in space, i.e., the function $\mathrm{color}(\mathbf{x})$. The shading depends, of course, on the primary color, but may also depend on other parameters, e.g., the gradient of the scalar field $\nabla s(\mathbf{x})$, ambient and diffuse lighting parameters, etc. In the remainder of this section we will not be concerned with shading but only with classification. (Shading will be discussed in Section 5.4.) Therefore, we choose a trivial shading, i.e., we identify the primary color $\tilde{c}\big(s(\mathbf{x})\big)$ assigned in the classification with $\mathrm{color}(\mathbf{x})$. Analogously, $\tau\big(s(\mathbf{x})\big)$ is identified with $\mathrm{extinction}(\mathbf{x})$.

The volume rendering integral is then written as

$$I = \int_0^D \tilde{c}\Big(s\big(\mathbf{x}(\lambda)\big)\Big) \exp\left(-\int_0^\lambda \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big)\mathrm{d}\lambda'\right)\mathrm{d}\lambda. \qquad (1)$$

3.2 Pre- and Post-Classification

Direct volume rendering techniques differ considerably in the way they evaluate Equation (1). One important and very basic difference is the computation of $\tilde{c}(s(\mathbf{x}))$ and $\tau(s(\mathbf{x}))$. In fact, the continuous scalar field $s(\mathbf{x})$ is usually defined by a mesh with scalar values s_i defined at each vertex $\mathbf{v}_i$ of the mesh together with an interpolation prescription.

The order of this interpolation and the application of the transfer functions defines the difference between *pre-* and *post-classification.* Post-classification is characterized by the application of the transfer functions *after* the interpolation of $s(\mathbf{x})$ from the scalar values at several vertices; while pre-classification is the application of the transfer functions *before* the interpolation step, i.e., colors $\tilde{c}(s_i)$ and extinction coefficients $\tau(s_i)$ are calculated in a pre-processing step for each vertex $\mathbf{v}_i$ and then used to interpolate $\tilde{c}(s(\mathbf{x}))$ and $\tau(s(\mathbf{x}))$ for the computation of the volume rendering integral.

Obviously, pre- and post-classification will produce different results whenever the interpolation does not commute with the transfer functions. As the interpolation is usually non-linear (e.g., trilinear in cartesian grids), it will *only* commute with the transfer functions if the transfer functions are constant or the identity. In *all* other cases, pre-classification will result in deviations from post-classification, which is “correct” in the sense of applying the transfer functions to a continuous scalar field defined by a mesh together with an interpolation prescription. (Nonetheless, pre-classification is useful under certain circumstances; in particular, because it may be used as a basic segmentation technique.)

3.3 Numerical Integration

An analytic evaluation of the volume rendering integral is possible in some cases, in particular for linear interpolation and piecewise linear transfer functions (see [14]). However, this approach is not feasible in general; therefore, a numerical integration is required.

The most common numerical approximation of the volume rendering integral is the calculation of a Riemann sum for n equal ray segments of length $d = D/n$. (See also Figure 1 and Section IV.A in [8].) It is straightforward to generalize the following considerations to unequally spaced ray segments.

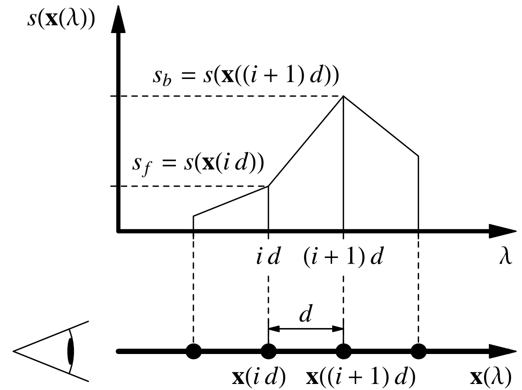

Figure 1: Scheme of the parameters determining the color and opacity of the i-th ray segment.

We will approximate the factor

$$\exp\left(-\int_0^\lambda \tau\Big(s\big(\mathbf{x}(\lambda')\big)\Big)\mathrm{d}\lambda'\right)$$

in Equation (1) by

$$\exp\left(-\sum_{i=0}^{\lambda/d} \tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)d\right) =$$

$$= \prod_{i=0}^{\lambda/d} \exp\Big(-\tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)d\Big) = \prod_{i=0}^{\lambda/d}(1-\alpha_i),$$

where the *opacity* α_i of the i-th ray segment is approximated by

$$\alpha_i \approx 1 - \exp\Big(-\tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)d\Big).$$

This is often further approximated to $\alpha_i \approx \tau\Big(s\big(\mathbf{x}(i\,d)\big)\Big)d$. $1-\alpha_i$ will be called the *transparency* of the i-th ray segement. The color $\tilde{C}_i$ emitted in the i-th ray segment may be approximated by $\tilde{C}_i \approx \tilde{c}\big(s(\mathbf{x}(i\,d))\big)d$. Thus, the approximation of the volume rendering integral in Equation (1) is

$$I \approx \sum_{i=0}^{n} \tilde{C}_i \prod_{j=0}^{i-1} (1-\alpha_j) \tag{2}$$

Therefore, a back-to-front compositing algorithm will implement the equation

$$\tilde{C}'_i = \tilde{C}_i + (1-\alpha_i)\tilde{C}'_{i+1}, \tag{3}$$

where $\tilde{C}'_i$ is the accumulated color in the i-th ray segment.

$\tilde{c}(s)$ is often substituted by $\tau(s)c(s)$ [8]. In this case, the approximation

$$C_i \approx \tau\big(s(\mathbf{x}(i\,d))\big)c\big(s(\mathbf{x}(i\,d))\big)d$$

will result in the more common approximation

$$I \approx \sum_{i=0}^{n} \alpha_i C_i \prod_{j=0}^{i-1} (1-\alpha_j)$$

with the corresponding back-to-front compositing equation

$$C'_i = \alpha_i C_i + (1-\alpha_i)C'_{i+1}. \tag{4}$$

This compositing equation indicates that $\tilde{C}$ corresponds to a *premultiplied color* αC; which is also called *opacity-weighted color* or *associated color*. According to Blinn in [1], associated colors have their opacity associated with them, i.e., they are regular colors composited on black. Blinn also notes that some intensity computations result in associated colors, although they are not explicitly multiplied by an opacity. In this sense, the transfer function $\tilde{c}(s)$ is in fact a transfer function for an associated color density.

A coherent discretization of viewing rays into equal segments may be interpreted as a discretization of the volume into *slabs*. Each slab emits light and absorbs light from the slabs behind it. However, the light emitted in each slab is not attenuated within the slab itself.

The discrete approximation of the volume rendering integral will converge to the correct result for $d \to 0$, i.e., for high sampling rates $n/D = 1/d$. According to the sampling theorem, a correct reconstruction is only possible with sampling rates larger than the Nyquist frequency. However, non-linear features of transfer functions may considerably increase the sampling rate required for a correct evaluation of the volume rendering integral as the Nyquist frequency of the fields $\tilde{c}\big(s(\mathbf{x})\big)$ and $\tau\big(s(\mathbf{x})\big)$ for the sampling along the viewing ray is approximately the product of the Nyquist frequencies of the scalar field $s(\mathbf{x})$ and the maximum of the Nyquist frequencies of the two transfer functions $\tilde{c}(s)$ and $\tau(s)$ (or of the product $c(s)\tau(s)$). Therefore, it is by no means sufficient to sample a volume with the Nyquist frequency of the scalar field if non-linear transfer functions are allowed. Artifacts resulting from this kind of undersampling are frequently observed unless they are avoided by very smooth transfer functions.

3.4 Pre-Integrated Classification

In order to overcome the limitations discussed above, the approximation of the volume rendering integral has to be improved. In fact, many improvements have been proposed, e.g., higher-order integration schemes, adaptive sampling, etc. However, these methods do not explicitly address the problem of high Nyquist frequencies of $\tilde{c}\big(s(\mathbf{x})\big)$ and $\tau\big(s(\mathbf{x})\big)$ resulting from non-linear transfer functions.

On the other hand, the goal of *pre-integrated classification* is to split the numerical integration into two integrations: one for the continuous scalar field $s(\mathbf{x})$ and one for the transfer functions $\tilde{c}(s)$ and $\tau(s)$ in order to avoid the problematic product of Nyquist frequencies.

The first step is the sampling of the continuous scalar field $s(\mathbf{x})$ along a viewing ray. Note that the Nyquist frequency for this sampling is not affected by the transfer functions. For the purpose of pre-integrated classification, the sampled values define a one-dimensional, piecewise linear scalar field. The volume rendering integral for this piecewise linear scalar field is efficiently computed by one table lookup for each linear segment. The three arguments of the table lookup are the scalar value at the start (front) of the segment $s_f := s\big(\mathbf{x}(id)\big)$, the scalar value the end (back) of the segment $s_b := s\big(\mathbf{x}((i+1)d)\big)$, and the length of the segment d. (See Figure 1.) More precisely spoken, the opacity α_i of the i-th segment is approximated by

$$\begin{aligned}\alpha_i &= 1-\exp\left(-\int_{i\,d}^{(i+1)d} \tau\Big(s\big(\mathbf{x}(\lambda)\big)\Big)\mathrm{d}\lambda\right)\\ &\approx 1-\exp\left(-\int_0^1 \tau\big((1-\omega)s_f+\omega s_b\big)d\,\mathrm{d}\omega\right).\end{aligned} \tag{5}$$

Thus, α_i is a function of s_f, s_b, and d. (Or of s_f and s_b, if the lengths of the segments are equal.) The (associated) colors $\tilde{C}_i$ are approximated correspondingly:

$$\begin{aligned}\tilde{C}_i \approx &\int_0^1 \tilde{c}\big((1-\omega)s_f+\omega s_b\big)\\ &\times\exp\Big(-\int_0^{\omega} \tau\big((1-\omega')s_f+\omega' s_b\big)d\,\mathrm{d}\omega'\Big)d\,\mathrm{d}\omega.\end{aligned} \tag{6}$$

Analogously to α_i, $\tilde{C}_i$ is a function of s_f, s_b, and d. Thus, pre-integrated classification will approximate the volume rendering integral by evaluating Equation (2):

$$I \approx \sum_{i=0}^{n} \tilde{C}_i \prod_{j=0}^{i-1} (1-\alpha_j)$$

with colors $\tilde{C}_i$ pre-computed according to Equation (6) and opacities α_i pre-computed according to Equation (5). For non-associated color transfer function, i.e., when substituting $\tilde{c}(s)$ by $\tau(s)c(s)$, we will also employ Equation (5) for the approximation of α_i and the following approximation of the associated color $\tilde{C}_i^{\tau}$:

$$\begin{aligned}\tilde{C}_i^{\tau} \approx &\int_0^1 \tau\big((1-\omega)s_f+\omega s_b\big)c\big((1-\omega)s_f+\omega s_b\big)\\ &\times\exp\Big(-\int_0^{\omega} \tau\big((1-\omega')s_f+\omega' s_b\big)d\,\mathrm{d}\omega'\Big)d\,\mathrm{d}\omega.\end{aligned} \tag{7}$$

Note that pre-integrated classification always computes associated colors, whether a transfer function for associated colors $\tilde{c}(s)$ or for non-associated colors $c(s)$ is employed.

In either case, pre-integrated classification allows us to sample a continuous scalar field $s(\mathbf{x})$ without the need to increase the sampling rate for any non-linear transfer function. Therefore, pre-integrated classification has the potential to improve the accuracy (less undersampling) and the performance (fewer samples) of a volume renderer at the same time.

3.5 Accelerated (Approximative) Pre-Integration

The primary drawback of pre-integrated classification in general is actually the pre-integration required to compute the lookup tables, which map the three integration parameters (scalar value at the front s_f, scalar value at the back s_b, and length of the segment d) to

pre-integrated colors $\tilde{C} = \tilde{C}(s_f, s_b, d)$ and opacities $\alpha = \alpha(s_f, s_b, d)$. As these tables depend on the transfer functions, any modification of the transfer functions requires an update of the lookup tables. This might be no concern for games and entertainment applications, but it strongly limits the interactivity of applications in the domain of scientific volume visualization, which often depend on user-specified transfer functions. Therefore, we will suggest three methods to accelerate the pre-integration step.

Firstly, under some circumstances it is possible to reduce the dimensionality of the tables from three to two (only s_f and s_b) by assuming a constant length of the segments. Obviously, this applies to ray-casting with equidistant samples. It also applies to 3D texture-based volume visualization with orthographic projection and is a good approximation for most perspective projections. It is less appropriate for axes-aligned 2D texture-based volume rendering as discussed in Section 5.5. Even if very different lengths occur, the complicated dependency on the segment length might be approximated by a linear dependency as suggested in [12]; thus, the lookup tables may be calculated for a single segment length.

Secondly, a local modification of the transfer functions for a particular scalar value s does not require to update the whole lookup table. In fact, only the values $\tilde{C}(s_f, s_b, d)$ and $\alpha(s_f, s_b, d)$ with $s_f \le s \le s_b$ or $s_f \ge s \ge s_b$ have to be recomputed; i.e., in the worst case about half of the lookup table has to be recomputed.

Finally, the pre-integration may be greatly accelerated by evaluating the integrals in Equations (5), (6), and (7) by employing integral functions for $\tau(s)$, $\tilde{c}(s)$, and $\tau(s)c(s)$, respectively. More specifically, Equation (5) for $\alpha_i = \alpha(s_f, s_b, d)$ can be rewritten as

$$\alpha(s_f, s_b, d) \quad \approx \quad 1 - \exp\left(-\frac{d}{s_b - s_f}\big(T(s_f) - T(s_b)\big)\right) \qquad (8)$$

with the integral function $T(s) := \int_0^s \tau(s)\mathrm{d}s$, which is easily computed in practice as the scalar values s are usually quantized.

Equation (6) for $\tilde{C}_i = \tilde{C}(s_f, s_b, d)$ may be approximated analogously:

$$\tilde{C}(s_f, s_b, d) \quad \approx \quad \frac{d}{s_b - s_f}\big(K(s_b) - K(s_f)\big) \qquad (9)$$

with the integral function $K(s) := \int_0^s \tilde{c}(s)\mathrm{d}s$. However, this requires to neglect the attenuation within a ray segment. As mentioned above, this is a common approximation for post-classified volume rendering and well justified for small products $\tau(s)d$.

For the non-associated color transfer function $c(s)$ we approximate Equation (7) by

$$\tilde{C}^\tau(s_f, s_b, d) \quad \approx \quad \frac{d}{s_b - s_f}\big(K^\tau(s_b) - K^\tau(s_f)\big). \qquad (10)$$

with $K^\tau(s) := \int_0^s \tau(s)c(s)\mathrm{d}s$.

Thus, instead of numerically computing the integrals in Equations (5), (6), and (7) for each combination of s_f, s_b, and d, we will only once compute the integral functions $T(s)$, $K(s)$, or $K^\tau(s)$ and employ these to evaluate colors and opacities according to Equations (8), (9), or (10) without any further integration.

3.6 Application to Volume Rendering Techniques

Pre-integrated classification is not restricted to a particular volume rendering technique, rather it may replace the post-classification step of various techniques. For example, in [12] Röttger et al. have applied pre-integrated classification to cell projection employing 3D textures for the lookup of segment colors $\tilde{C}$ and opacities α. In fact, the application of pre-integrated classification is quite natural for the cell projection of tetrahedral meshes, because the linear interpolation of the scalar field between two samples is exact if the samples are taken at the faces of tetrahedra as in the case of cell projection.

Of course, pre-integrated classification may also be employed in other volume rendering techniques, e.g., software ray-casting of structured and unstructured meshes. In the remainder of this paper, however, we will focus on the implementation of pre-integrated classification in texture-based volume rendering algorithms.

4 Texture-Based Pre-Integrated Volume Rendering

Based on the description of pre-integrated classification in Section 3.4, we will now present two novel texture-based algorithms (one for 2D textures and one for 3D textures) that implement pre-integrated classification. Both algorithms employ dependent textures, i.e., rely on the possibility to convert fragment (or pixel) colors into texture coordinates. The technical details of this table lookup will be discussed in Section 5.

The basic idea of texture-based volume rendering is to render a stack of textured slices. Texture maps may either be taken from three stacks of two-dimensional texture maps (object-aligned slices; see [11]) or from one three-dimensional texture map (view-aligned slices; see [3]). Pre-classification is implemented by applying the transfer functions once for each texel and storing colors and opacities in the texture map(s). On the other hand, post-classification is performed by storing the scalar field value in the texture map(s) and applying transfer functions during the rasterization of the slices for each pixel. Each pixel (more precisely spoken, each fragment) of a slice corresponds to the contribution of one ray segment to the volume rendering integral for this pixel. Therefore, the compositing Equations (3) or (4) are employed for the rasterization of the textured slices. As each fragment of a slice corresponds to one ray segment, the whole slice corresponds to a slab of the volume as depicted in Figure 2.

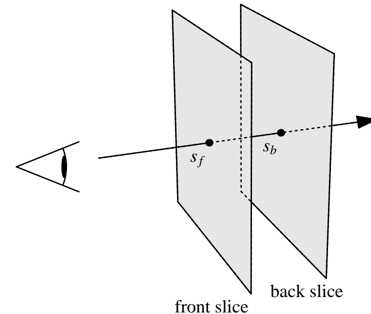

Figure 2: A slab of the volume between two slices. The scalar value on the front (back) slice for a particular viewing ray is called s_f (s_b).

After these preliminaries, we can now describe pre-integrated volume rendering using textured slices. The texture maps (either three-dimensional or two-dimensional textures) contain the scalar values of the volume, just as for post-classification. As each pair of adjacent slices (either view-aligned or object-aligned) corresponds to one slab of the volume (see Figure 2), the texture maps of two adjacent slices have to be mapped onto one slice (either the front or the back slice) by means of multiple textures (see Section 5.1). Thus, the scalar values of both slices (front and back) are fetched from texture maps during the rasterization of the polygon for one slab (see Section 5.2). These two scalar values are required for a third texture fetch operation, which performs the lookup of pre-integrated colors and opacities from a two-dimensional texture

map. This texture fetch depends on previously fetched texels; therefore, this third texture map is called a *dependent* texture map.

The opacities of this dependent texture map are calculated according to Equation (5), while the colors are computed according to Equation (6) if the transfer function specifies associated colors $\tilde{c}(s)$, and Equation (7) if it specifies non-associated colors $c(s)$. In either case the compositing Equation (3) is used for blending as the dependent texture map always contains associated colors.

This completes the description of the algorithms for pre-integrated volume rendering with view-aligned slices and object-aligned slices, respectively. Obviously, a hardware implementation of these algorithms depends on rather complicated texture fetch operations. Fortunately, the OpenGL texture shader extension recently proposed can in fact be customized to implement these algorithms. The details of this implementation are discussed in the following section.

5 Implementation Details

Our current implementation is based on *NVidia's GeForce3* graphics chip. *NVidia* introduced a flexible multi-texturing unit in their *GeForce2* graphics processor via the *register combiners* OpenGL extension [4]. This unit allows the programming of per-pixel shading operations using three stages, two general and one final combiner stage. This *register combiner* extension is located behind the texel fetch unit in the rendering pipeline. Recently *NVidia* extended the *register combiners* in the *GeForce3* graphics chip, by providing eight general and one final combiner stage with per-combiner constants via the *register combiner2* extension. Additionally, the *GeForce3* provides a programmable texture fetch unit [4] allowing four texture fetch operations via 21 possible commands, among them several dependent texture operations. This so called *texture shader* OpenGL extension and the *register combiners* are merged together in *Microsoft's DirectX8 API* to form the *pixel shader* API. Unfortunately, the *pixel shader* API is more restrictive than the two OpenGL extensions. Therefore, we based our implementation on the OpenGL API [4]. The *texture shader* extension refers to 2D textures only. Although *NVidia* proposed an equivalent extension for 3D texture fetches via the *texture shader2* extension, 3D textures and *texture shader2* are not supported in the current driver releases.

Best results would be obtained using 3D textures. However, as they are currently not available, we used a 2D texture-based approach. Slices are set parallel to the coordinate axes of the rectilinear data grid, i.e object-aligned. This allows us to substitute trilinear by bilinear interpolation. However, if the viewing direction changes by more that 90 degrees, the orientation of the slice normal must be changed. This requires to keep three copies of the data stack in main memory, one stack of slices for each slicing direction respectively. The slices are rendered as planar polygons textured with the image information obtained from a 2D texture map and blended onto the image plane.

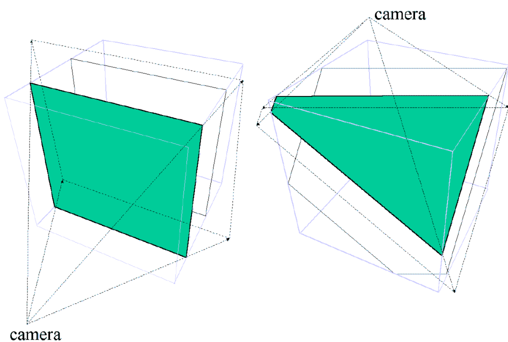

Figure 3: Projection of texture slice vertices onto adjacent slice polygons for object-aligned slices (left) and view-aligned slices (right)

The pre-integrated volume rendering algorithm consists of three basic steps: First two adjacent texture slices are projected onto one of them, either the back slice onto the front slice or vice versa. Thereby, two texels along each ray (one from the front and one from the back slice) are projected onto each other. They are fetched using the texture shader extension and then used as texture coordinates for a dependent texture fetch containing pre-integrated values for each combination of back and front texels. For isosurface rendering, the dependent texture contains color, transparency, and interpolation values, if the isovalue is in between the front and back texel value. This results in dependent texture patterns as shown in Figure 9(left). The gradient and voxel values are stored in RGBA textures. In the register combiners gradients are interpolated and dot product lighting calculations are performed. The following sub-sections explain all these steps in detail.

5.1 Projection

The 2D texture-based volume rendering algorithm usually blends object-aligned texture slices of one of the three texture stacks back-to-front into the frame buffer using the over operator. Instead of this slice-by-slice approach, we render slab-by-slab (see Figure 2) from back to front into the frame buffer. A single quadrilateral polygon is rendered for each slab with the two corresponding textures as texture maps. In order to have texels along all viewing rays projected upon each other for the texel fetch operation, either the back slice must be projected onto the front slice or vice versa. The projection is thereby accomplished by adapting texture coordinates for the projected texture slice and retaining the texture coordinates of the other texture slice. Figure 3 shows the projection for the object- and view-aligned rendering algorithms.

For direct volume rendering without lighting, textures are defined in the OpenGL texture format `GL_LUMINANCE8`. For volume shading and shaded isosurfaces `GL_RGBA` textures are used, which contain the pre-calculated volume gradient and the scalar values.

5.2 Texel Fetch

For each fragment, texels of two adjacent slices along each ray through the volume are projected onto each other. Thus, we can fetch the texels with their given per-fragment texture coordinates. Then the two fetched texels are used as lookup coordinates into a dependent 2D texture, containing pre-integrated values for each of the possible combinations of front and back scalar values as described in Section 3.4. *NVidia's* texture shader extension provides a texture shader operation that employs the previous texture shader's green and blue (or red and alpha) colors as the (s,t) coordinates for a non-projective 2D texture lookup. Unfortunately, we cannot use this operation as our coordinates are fetched from two separate 2D textures. Instead, as a workaround, we use the dot product texture shader, which computes the dot product of the stage's (s,t,r) and a vector derived from a previous stage's texture lookup (see Figure 4). The result of two of such dot product texture shader operations are employed as coordinates for a dependent texture lookup. Here the dot product is only required to extract the front and back volume scalars. This is achieved by storing the volume scalars in the red components of the textures and applying a dot product with a constant vector $\mathbf{v} = (1,0,0)^T$. The texture shader extension allows us to define to which previous texture fetch the dot product refers with the `GL_PREVIOUS_TEXTURE_INPUT_NV` texture environment. The

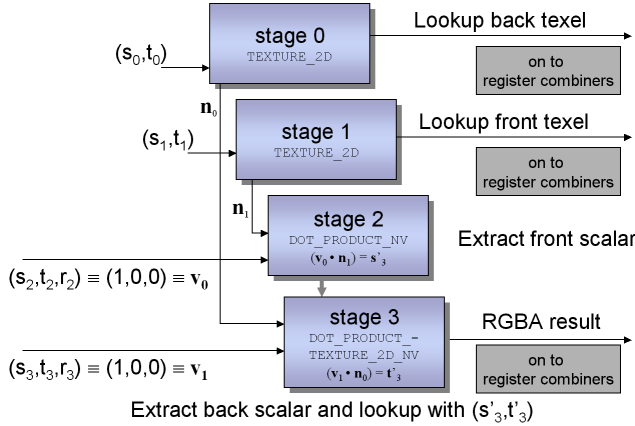

Figure 4: Texture shader setup for dependent 2D texture lookup with texture coordinates obtained from two source textures.

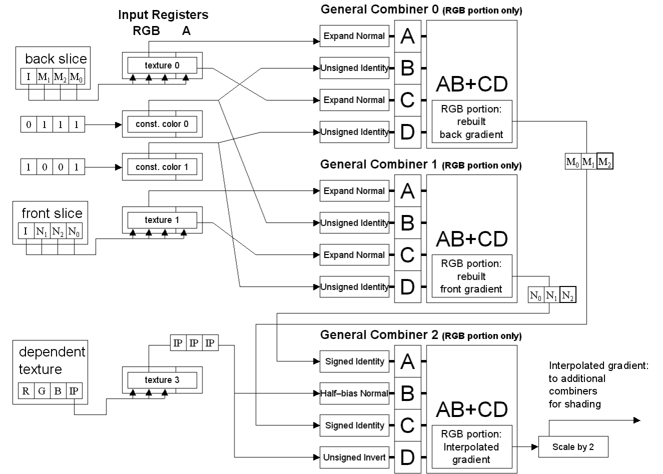

Figure 5: Register Combiner setup for gradient reconstruction and interpolation with interpolation values stored in alpha . Note that the interpolation values are stored in the range of 0.5 to 1.0, which requires proper input and output mappings for general combiner 2 to obtain a correct interpolation. *M* denotes the gradient of the back slice, *N* the front slice gradient respectively.

first dot product is set to use the fetched front texel values as previous texture stage, the second uses the back texel value [1]. In this approach, the second dot product performs the texture lookup into our dependent texture via texture coordinates obtained from two different textures.

For direct volume rendering without lighting the fetched texel from the last dependent texture operation is routed through the register combiners without further processing and blended into the frame buffer with the OpenGL blending function `glBlendFunc(GL_ONE,GL_ONE_MINUS_SRC_ALPHA)`.

5.3 Gradient Interpolation for Isosurfaces

As discussed in [12], pre-integrated volume rendering can be employed to render multiple isosurfaces. The basic idea is to color each ray segment according to the first isosurface intersected by the ray segment. Examples for such dependent textures are depicted in Figure 8.

For shading calculations, `RGBA` textures are usually employed, that contain the volume gradient in the `RGB` components and the volume scalar in the `ALPHA` component. As we use dot products to extract the front and back volume scalar and the dot product refers only to the first three components of a vector, we store the scalar data in the `RED` component. The first gradient component is stored in the `ALPHA` component in return.

For lighting purposes the gradient of the front and back slice has to be rebuilt in the `RGB` components (`ALPHA` has to be routed back to `RED`) and the two gradients have to be interpolated depending on a given isovalue (see Figure 5). The interpolation value for the back slice is given by $\mathtt{IP} = (s_{\text{iso}} - s_f)/(s_b - s_f)$; the interpolation value for the front slice is $1-\mathtt{IP}$ (see also [12]). `IP` could be calculated on-the-fly for each given isovalue, back and front scalar. Unfortunately, this requires a division in the register combiners, which is not available. For this reason we have to pre-calculate the interpolation values for each combination of back and front scalar and store them in the dependent texture. Ideally, this interpolation value would be looked up using a second dependent texture. Unfortunately, *NVidia's* texture shader extension only allows four texture operations, which we already spent. Hence we have to store the interpolation value `IP` in the first and only dependent texture.

[1] In the pixel shader 1.0 and 1.1 API of Microsoft DirectX8, a dot product always refers to the last fetch operation before the dot products, therefore this operation can't be realized.

There are two possible ways to store these interpolation values. The first approach stores the interpolation value (`IP`) in the `ALPHA` component of the dependent texture (`R,G,B,IP`). The main disadvantage of this method is, that the transparency, which is usually freely definable for each isosurface's back and front face, is now constant for all isosurfaces' faces. In order to obtain a transparency value of zero for ray segments that do not intersect the isosurface and a constant transparency for ray segments that intersect the isosurface the interpolation values are stored in the `ALPHA` channel in the range 128 to 255 (7 bit). An interpolation value of 0 is stored for ray segments that do not intersect the isosurface. This allows us to scale the `ALPHA` channel with a factor of 2, to get an `ALPHA` of 1.0 for ray segments intersecting the isosurface and an `ALPHA` of 0 otherwise. Afterwards, a multiplication of the result with the constant transparency can be performed. For the interpolation the second general combiner's input mapping for the interpolation is set to `GL_HALF_BIAS_NORMAL_NV` and `GL_UNSIGNED_INVENT_NV` to map the the interpolation value to the ranges 0 to 0.5 and 0.5 to 0 (see Figure 5). After the interpolation the result is scaled with 2 in order to get the correct interpolation result.

Our second approach stores the interpolation value `IP` in the `BLUE` component of the dependent texture (`R,G,IP,A`). Now the transparency can be freely defined for each isosurface and each back and front face of the isosurface, but the register combiners are used to fill the blue color channel with a constant value, that is equal for all isosurfaces' back and front faces. Also we can use all 8 bits of the `BLUE` color channel for the interpolation value. In order to distribute the interpolation value from the `BLUE` color channel on all `RGB` components for the interpolation, `BLUE` is first routed into the `ALPHA` portion of a general combiner stage and then routed back into the `RGB` portion (see Figure 6).

5.4 Lighting

After the per-fragment calculation of the isosurfaces' gradient in the first three general combiner stages, the remaining five general combiners and the final combiner can be used for lighting compu-

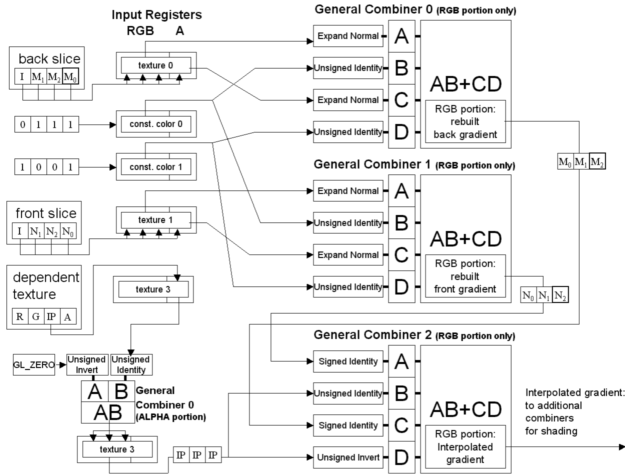

Figure 6: Register Combiner setup for gradient reconstruction and interpolation with interpolation values stored in blue. Note that the interpolation values are routed in the alpha portion and back into the RGB portion to distribute the values onto RGB for interpolation.*M* denotes the gradient of the back slice, *N* the front slice gradient respectively.

tations. Diffuse and specular lighting with a maximum power of 256 is possible by utilizing the dot product of the register combiners and increasing the power by multiplying the dot product with itself. Currently we calculate $I = I_a + I_d C(\mathbf{n} \cdot \mathbf{l}_1) + I_s C(\mathbf{n} \cdot \mathbf{l}_2)^{16}$, where $\mathbf{n}$ denotes the interpolated normal, $\mathbf{l}_1$ the diffuse light source direction, $\mathbf{l}_2$ the specular light source direction, and C the color of the isosurface. A visualization of a CT scan of a human head at different thresholds is shown in Figure 8.

The same approach can also be employed for volume shading. For lighting, the average gradient at the front and back slice is used, thus no interpolation values have to be stored in the dependent texture. The dependent texture holds pre-integrated opacity and color values, latter are employed for diffuse and specular lighting calculations. The implemented lighting model computes $I = I_d C(\mathbf{n} \cdot \mathbf{l}_1) + I_s C(\mathbf{n} \cdot \mathbf{l}_2)^{16}$, where $\mathbf{n}$ denotes the interpolated normal, $\mathbf{l}_1$ the diffuse light source direction, $\mathbf{l}_2$ the specular light source direction and C the pre-integrated color of the ray segment.

Dynamic lighting as described above requires the employment of `RGBA` textures, which consume a lot of texture memory. Alternatively, static lighting is possible by storing pre-calculated dot products of gradient and light vectors for each voxel in the textures. The dot products at the start and end of a ray segment are then interpolated for a given isovalue in the register combiners. For this purpose `LUMINANCE_ALPHA` textures can be employed, which consume only half of the memory of `RGBA` textures.

The intermixing of semi-transparent volumes and isosurfaces is performed by a multi-pass approach that first renders a slice with a pre-integrated dependent texture and then renders the slice again with a isosurface dependent texture. Without the need of storing the interpolation values in the dependent texture, a single pass approach could also be implemented, which neglects isosurfaces and semi-transparent volumes in a slab at the same time. Examples of dependent textures for direct and isosurface volume rendering are presented in Figures 9 and 10.

5.5 Problems

The thicknesses of the slabs are usually equal. However, this does not necessarily imply a constant length of the ray segments. For equidistant, view-aligned slices only perspective projections will result in different lengths. Fortunately, these variations are often neglectable in practice as extreme perspectives are usually avoided. Thus, a constant length of the ray segments may be assumed in good approximation. Therefore, the lookup tables for colors and opacities are only two-dimensional tables depending on the scalar value at the front and back of a ray segment.

For object-aligned 2D textured slices the lengths of the ray segments does also vary with the rotation of the volume. However, for each rotation there is only one length—at least for orthogonal projections. The maximum factor of the variation of this length is $\sqrt{3}$. In order to avoid too strong errors, a set of two-dimensional lookup tables for different lengths should be employed. For volumes with unequal slice distances in the main axes directions, different lookup textures must also be calculated.

Another problem that occurs when rendering semi-transparent isosurfaces is, that some pixel are rendered twice if the surface intersects the viewing ray exactly at the front slice, which results in visible pixel errors. Once the next slab is rendered, the same pixel is rendered again, because now the surface intersects the ray exactly at the back slice. To circumvent this problem the OpenGL stencil buffer test can be utilized. Each time a slab is rendered into the frame buffer, the stencil buffer is cleared and the slab is also rendered into the stencil buffer using a second dependent texture that only selects pixels for rendering, where the isosurface intersects the ray at the front slice position. If the next slab is rendered into the frame buffer, the stencil buffer test is used and only pixels that where not rendered into the stencil buffer in the previous step are set. A comparison of images generated without and with stencil buffer test enabled are shown in Figure 7. Currently this method works only for a single semi-transparent isosurface.

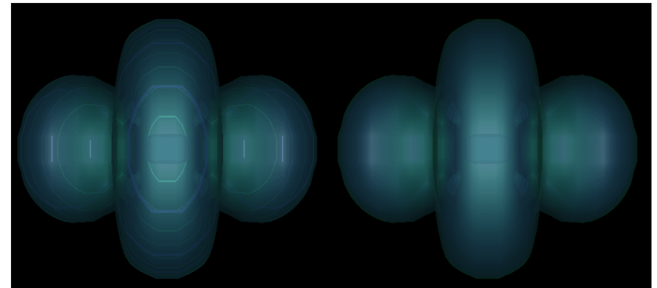

Figure 7: Semi-transparent isosurface rendering of the spherical harmonic (Legendre's) function without (left) and with correction (right). Note, that the annular artifacts on the left are successfully removed on the right.

6 Results

In scientific visualization applications, which are often employed to explore unknown data, it is quite important to be able to interactively change the transfer functions. For isosurface rendering, these updates can be performed very fast with our method, as no integral has to be computed. For direct volume rendering, the calculation of the new dependent texture from the given transfer functions depends of course on the CPU performance. Our test PC is equipped with a 650 MHz Athlon processor. For a global change of the transfer function, the update of a 256×256 dependent texture, taking the self-attenuation within slabs into account, took ca 20 seconds and a maximum of ca 10 seconds for a local update. Neglecting the self-attenuation the update required ca 0.3 seconds for a global and a maximum of ca 0.15 seconds for a local update,

while the time required for the upload of the new dependent texture into the texture memory is neglectable. These results show, that an interactive update of the transfer functions is possible, when neglecting the self-attenuation. As the differences in quality of both approaches are quite small, a volume renderer could first neglect the self-attenuation during the interaction with the transfer functions and calculate the “correct” dependent texture in the background for a later update.

For comparison purposes we implemented post-shading with additional interpolated slices. The interpolation is performed in the third and fourth texture stage and the one-dimensional texture lookup is implemented by the dependent texture fetch operation with an appropriate two-dimensional texture map.

Figure 9 shows a comparison of the image quality achieved with pre-shading without additional interpolated slices (as presented in [11]), post-shading without and with additional slices, and pre-intergrated volume rendering using the same view parameters and transfer functions. Obviously, pre-shading results in low image quality with strong slicing artifacts (Fig. 9a). Post-shading provides much better image quality (Fig. 9b). However, additional interpolated slices are necessary in order to remove the slicing artifacts (Fig. 9c). Note, that additional slices decrease the frame rate and image accuracy because of more rasterization and blending operations. On the other hand, pre-integrated rendering achieves the full quality without interpolated slices (Fig. 9d).

Especially direct volume rendering with few slices gains much quality with the new approach. Figure 10 demonstrates the high image quality achieved with a small number of textured slices and a random transfer function with high frequencies. Although only a small number of slices are rendered, all the details of the transfer function are still visible. For comparison pre-shading (bottom, left) and post-shading (top, right) results are included in Figure 10.

For performance evaluations, the test PC was equipped with a *NVidia GeForce3* graphics board with 64 MB DDR SDRAM (3.8 ns), internal clock of 200 MHz, 460 MHz memory interface clock and 4x AGP. The performance of the pre-integrated volume renderer is basically limited by memory bandwidth, rasterization power, and texel fetch operation performance, whereby the full quality of the renderer is achieved without the interpolation and rendering of additional slices. We require four texture fetch operations, which are performed on the *GeForce3* chip in two clock cycles. All tests were performed with a 512^2 viewport. For direct volume rendering we achieved ca 90 fps for a volume with a resolution of 16^3 voxels, ca 50 fps for 32^3, ca 21 fps for 64^3, ca 13 fps for 128^3, and ca 4 fps for 256^3 voxels. For isosurface rendering we achieved ca 70 fps for a volume with a resolution of 16^3 voxels, ca 40 fps for 32^3, ca 21 fps for 64^3, ca 11 fps for 128^3, and ca 1 fps for 256^3 voxels. The rendering algorithm and rasterization power needed for both rendering modes are the same. They differ in the number of employed general combiner stages and memory bandwith requirements as 8 bit textures are employed for direct and 32 bit textures for isosurface rendering. The results are independent of the transfer functions and the number of isosurfaces.

Our results show, that interactive transfer function updates are possible. The image quality of the pre-integrated volume rendering algorithm surpasses our previous approach by far. We achieve the full rendering quality without spending rasterization power by rendering additional interpolated slices.

7 Conclusions

We presented a novel volume rendering approach that provides high image quality even with low-resolution volume data. Besides direct volume rendering, the algorithm also allows us to render double-sided isosurfaces with diffuse and specular lighting without extracting a polygonal representation. An arbitrary number of isosurfaces can be visualized without performance penalty. Furthermore, volume shading and mixed rendering of isosurfaces and semi-transparent volumes is possible. The time complexity does neither depend on the number of isosurfaces nor the definition of the transfer functions.

We implemented a hardware-accelerated implementation on current consumer graphics hardware, more precisely the new *GeForce3* graphics chip by *NVidia*. The current implementation employs 2D textures and some other more or less elegant tricks to make the implementation possible on current low-cost graphics hardware.

The ideal graphics hardware would provide dependent texture lookups with texture coordinates obtained from two source textures, more texture shader operations and most importantly 3D textures support. As *NVidia* already proposed the necessary OpenGL extensions [4], we are optimistic to achieve even better results with new drivers on current or future graphics hardware.

8 Acknowledgments

We would like to thank Eckhart Traber of ELSA AG for providing a GeForce3-based graphics card just in time. Special thanks to Christof Resk-Salama and Ruediger Westermann for many fruitful discussions.

References

[1] J. Blinn. Jim blinn’s corner — compositing, part i: Theory. *IEEE Computer Graphics and Applications*, 14(5):83–87, 1994.

[2] M. Brady, K. Jung, Nguyen HT, and T. Nguyen. Two-Phase Perspective Ray Casting for Interactive Volume Navigation. In *Visualization ’97*, 1997.

[3] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. *ACM Symp. on Vol. Vis.*, 1994.

[4] NVIDIA Corporation. NVIDIA OpenGL specifications. http://www.nvidia.com/Developer.

[5] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-Quality Volume Rendering Using Texture Mapping Hardware. In *SIGGRAPH Eurographics Graphics Hardware Workshop*, 1998.

[6] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear–Warp Factorization of the Viewing Transform . *Comp. Graphics*, 28(4), 1994.

[7] William R. Mark, Svetoslav Tzvetkov, and Pat Hanrahan. A real-time procedural shading system for programmable graphics hardware. In *to appear in SIGGRAPH 2001*, 2001.

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

[9] M. Meißner, U. Hoffmann, and W Straßer. Enabling Classification and Shading for 3D Texture Based Volume Rendering Using OpenGL and Extensions. In *Visualization ’99*, 1999.

[10] H. Pfister, J. Hardenbergh, G. Knittel, H. Lauer, , and L. Seiler. The VolumePro Real-Time Ray-Casting System. In *Proceedings of of SIGGRAPH 99*, pages 251 – 260. ACM, 1999.

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

[12] S. Röttger., M. Kraus, and T. Ertl. Hardware-accelerated volume and isosurface rendering. In *Proc. of Visualization ’00*, pages 109–116, 2000.

[13] R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proc. of SIGGRAPH*, Comp. Graph. Conf. Series, 1998.

[14] P. Williams, N. Max, and C. Stein. A high accuracy volume renderer for unstructured data. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):37–54, 1998.

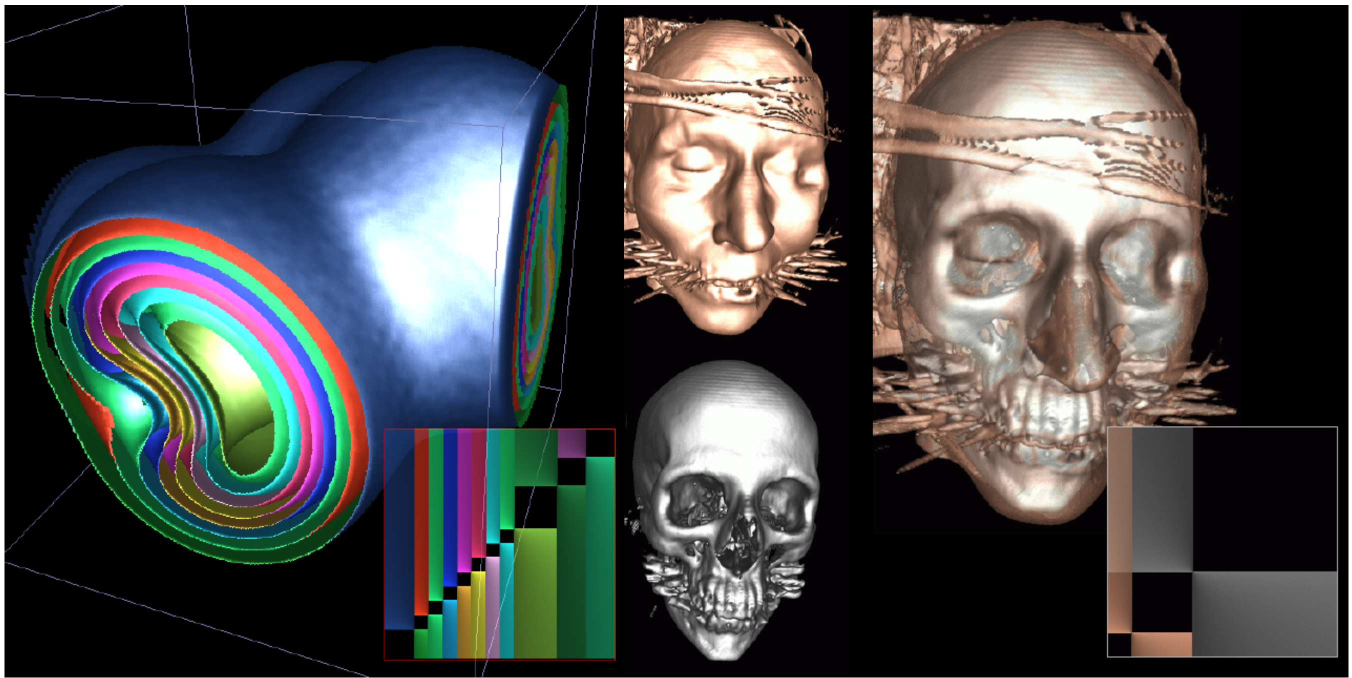

Figure 8: Left to right: Multiple colored isosurfaces of a synthetic data set with the corresponding dependent texture. Isosurfaces of a human head CT scan (256^3): skin, skull, semi-transparent skin with opaque skull and the dependent texture for the latter image.

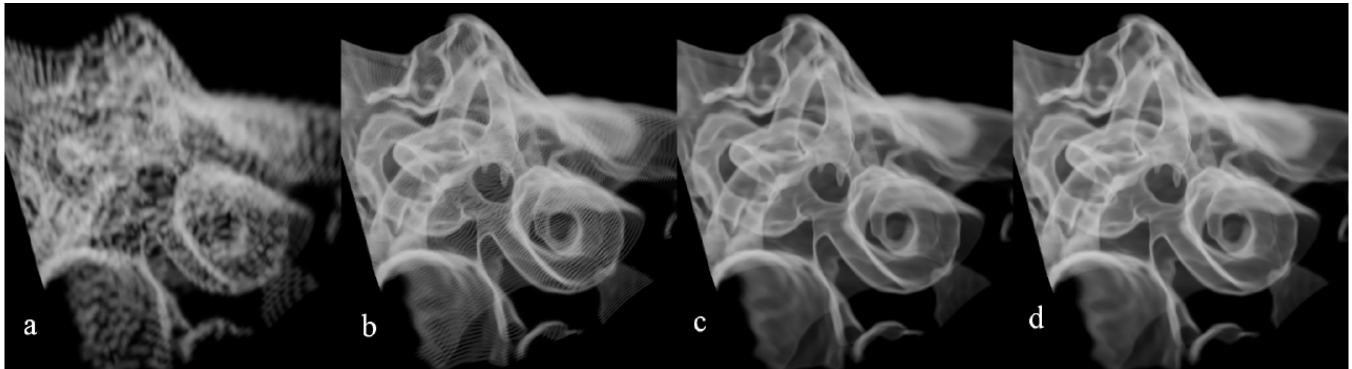

Figure 9: Images showing a comparison of a) pre-shaded, b) post-shaded without additional slices, c) post-shaded with additional slices and d) pre-integrated volume visualization of tiny structures of the inner ear ($128 \times 128 \times 30$) with 128 slices.

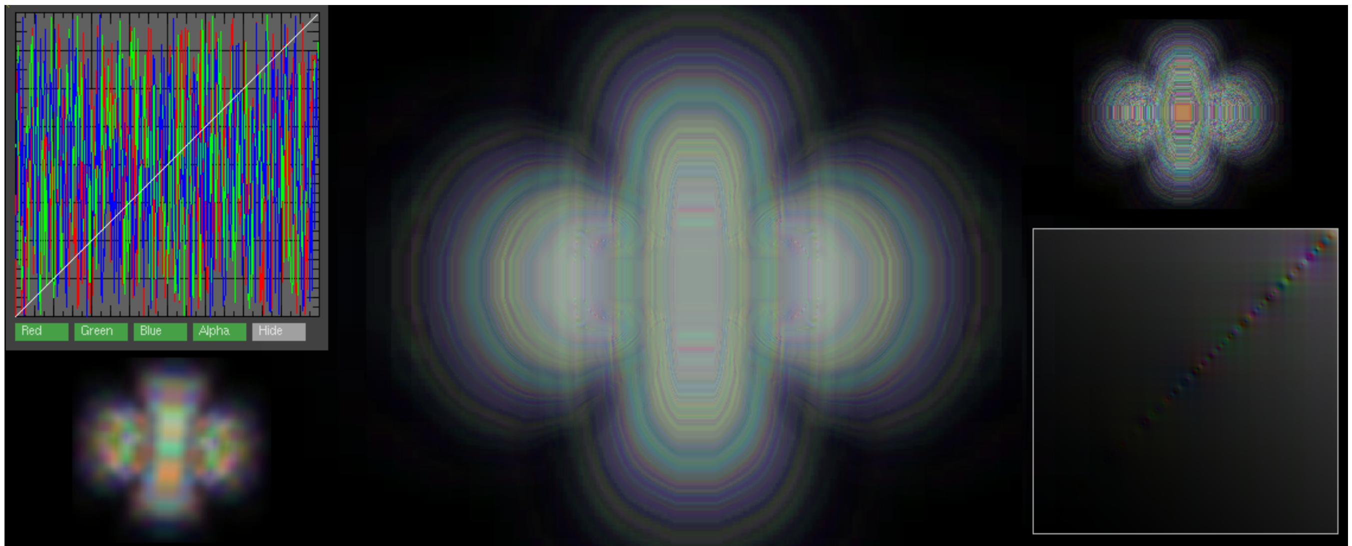

Figure 10: High-quality pre-integrated direct volume rendering of a spherical harmonic (Legendre's) function with random transfer functions (top, left) and dependent texture (bottom, right). The resolution was 16^3 voxels, thus only 15 textured slices were rendered. Pre-shaded (bottom, left) and post-shaded (top, right) results are included for comparison.