Interactive High-Quality Volume Rendering with Flexible Consumer Graphics Hardware

Klaus Engel and Thomas Ertl

Visualization and Interactive Systems Group, University of Stuttgart

Abstract

Recently, the classic rendering pipeline in 3D graphics hardware has become flexible by means of programmable geometry engines and rasterization units. This development is primarily driven by the mass market of computer games and entertainment software, whose demand for new special effects and more realistic 3D environments induced a reconsideration of the once static rendering pipeline. Besides the impact on visual scene complexity in computer games, these advances in flexibility provide an enormous potential for new volume rendering algorithms. Thereby, they make yet unseen quality as well as improved performance for scientific visualization possible and allow to visualize hidden features contained within volumetric data.

The goal of this report is to deliver insight into the new possibilities that programmable state-of-the-art graphics hardware offers to the field of interactive, high-quality volume rendering. We cover different slicing approaches for texture-based volume rendering, non-polygonal iso-surfaces, dot-product shading, environment-map shading, shadows, pre- and post-classification, multi-dimensional classification, high-quality filtering, pre-integrated classification and pre-integrated volume rendering, large volume visualization and volumetric effects.

1. Introduction

The massive innovation pressure exerted on the manufacturers of PC graphics hardware by the computer games market has led to very powerful consumer 3D graphics accelerators. The latest development in this field is the introduction of programmable graphics hardware that allows computer game developers to implement new special effects and more complex and compelling scenes. As a result, the current state-of-the-art graphics chips, e.g. the NVIDIA GeForce4, or the ATI Radeon8500, are not only competing with professional graphics workstations but often surpass the abilities of such hardware regarding speed, quality, and programmability. Therefore, this kind of hardware is becoming increasingly attractive for scientific visualization. Due to the new programmability offered by graphics hardware, many popular visualization algorithms are now being mapped efficiently onto graphics hardware. Moreover, this innovation leads to completely new algorithms. Especially in the field of volume graphics, that at all times had very high computation and resource requirements, the last few years brought a large number of new and optimized algorithms. In return for the innovations delivered by the gaming industry, scientific volume visualization can provide new optimized algorithms for volumetric effects in entertaiment applications.

This report covers newest volume rendering algorithms and their implementation on programmable consumer graphics hardware. Specialized volume rendering hardware[34, 30] is not the focus of this report. If details on the implementation of a specific algorithms are given, we focus on OpenGL and its extensions[32], although Microsoft's DirextX/3D API provides similar functionality for a comparable implementation.

In this report, we restrict ourselves to volume data defined on rectilinear grids. In such grids, the volume data are comprised of samples located at grid points, which are equispaced along each volume axis, and can therefore easily be stored in a texture map. We will start with some basic, yet important considerations, that are required to produce satisfying visualization results. After a brief introduction into the new features of consumer graphics hardware, we will outline the basic principle of texture-based volume rendering with different slicing approaches. Subsequently, we introduce several optimized algorithms, which represent

the state-of-the-art in hardware-accelerated volume rendering. Finally, we summarize current restrictions of consumer graphics hardware and give some conclusions with an outlook on future developments.

2. Basics

Although volumetric data is defined over a continuous three-dimensional domain (R^3), measurements and simulations provide volume data as 3D arrays, where each of the scalar values that comprise the volume data set is obtained by sampling the continuous domain at a discrete location. These values are referred to as voxels (volume elements) and usually quantized to 8, 16, or 32 bit accuracy and saved as fixed point or floating point numbers. Figure 1 shows such a representation of a volumetric object as a collection of a large number of voxels.

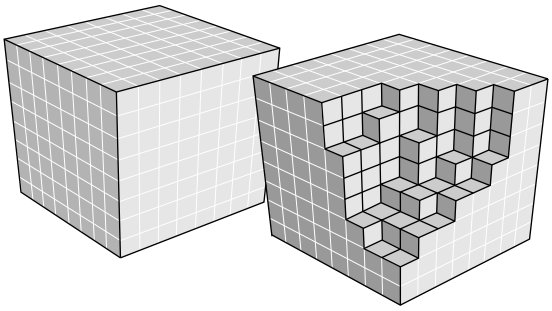

Figure 1: *Voxel representation of a volumetric object after it has been discretized.*

2.1. Reconstruction

In order to reconstruct[25, 31] the original continuous signal from the voxels, a reconstruction filter is applied, that calculates a scalar value for the continuous three-dimensional domain (R^3) by performing a convolution of the discrete function with a filter kernel. It has been proved, that the "perfect", or ideal reconstruction kernel is provided by the sinc filter [33]

$$\mathrm{sinc}(x) = \frac{\sin(\pi x)}{\pi x}. \qquad (1)$$

As this reconstruction filter has an unlimited extent, in practice more simple reconstruction filters like tent or box filters are applied (see Figure 2). Current graphics hardware provides linear, bilinear, and trilinear filtering for magnification and pre-filtering with mip-mapping and anisotropic filters for minification. However, due to the availability of multi-textures and flexible per-fragment operations, consumer graphics hardware also allows filters with higher quality (see Chapter 7).

Given a reconstruction filter and the three-dimensional array of voxels, the data is visualized by sending rays from the eye through each pixel of the image plane through the volume and sampling the data with a constant sampling rate (ray-casting)[23]. According to the Nyquist theorem, the original signal can be reproduced as long as the sampling rate is at least twice that of the highest frequency contained in the original signal. This is, of course, a problem of the volume acquisition during the measurements and simulations, so this problem does not concern us here. In the same manner the sampled signal may be reconstructed as long as the sampling distance in the volume data is at least twice that of the highest frequency contained in the sampled volume. As a consequence of the Nyqist theorem we have to choose a sampling rate that satisfies those needs. However, we will show in Chapter 8 that due to high frequencies introduced during the classification, which is applied to each filtered sample of the volume data set, it is in general not sufficient to sample the volume with the Nyquist frequency of the volume data. Because of these observations we will later present a classification scheme that allows us to sample the volume with the Nyquist frequency of the volume data by pre-integrating ray-segments in a pre-processing step (see Section 8).

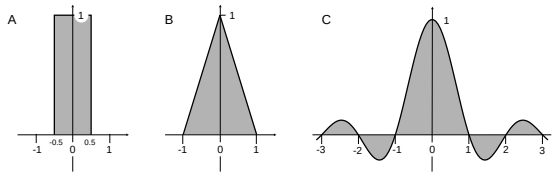

Figure 2: *Three reconstruction filters: (a) box, (b) tent and (c) sinc filters.*

2.2. Classification

Classification is another crucial subtask in the visualization of volume data, i.e. the mapping of volume samples to RGBA values. The classification step is introduced by 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. The order of classification and filtering strongly influences the resulting images, as demonstrated in Figure 3. The image shows the results of pre- and post-classification at the example of a 16^3 voxel hydrogen orbital volume and a high frequency transfer function for the green color channel.

It can be observed that pre-classification, i.e. classification before filtering, does not reproduce high-frequencies in the transfer function. In contrast to this, post-classification, i.e. classification after filtering, reproduces high frequencies in the transfer function on the slice polygons.

2.3. Ray Integration

After the filtering and classification of the volume data, an integration along view rays through the volume is

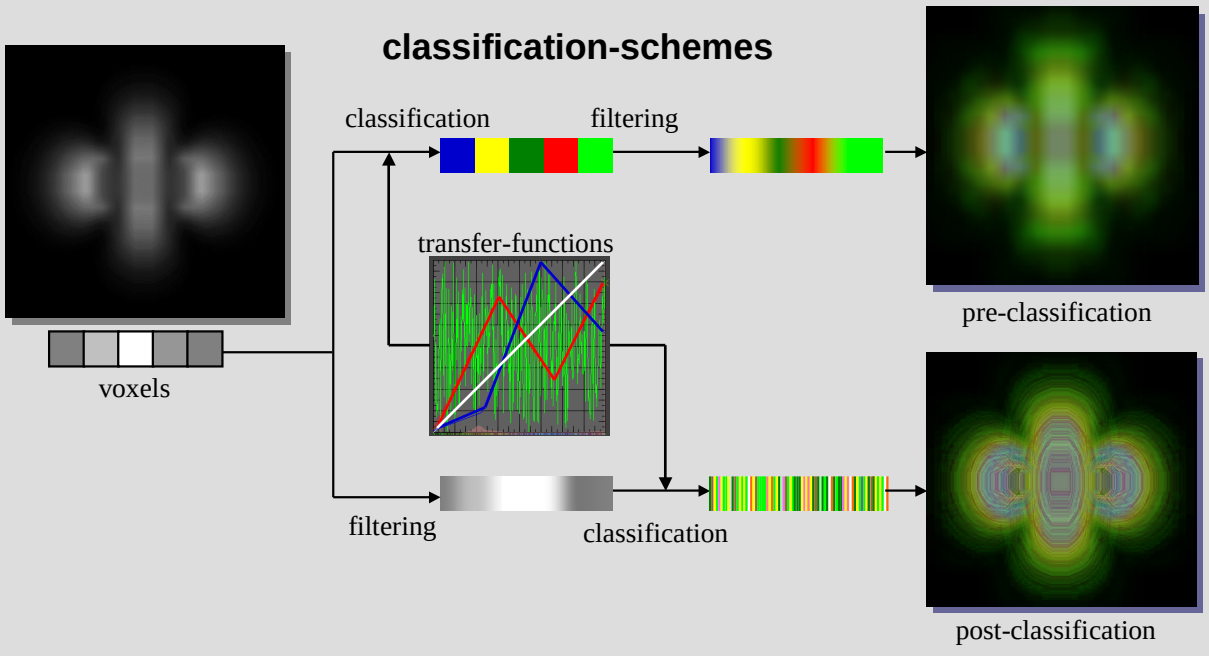

Figure 3: *Comparison of pre-classification and post-classification. Alternate orders of classification and filtering lead to completely different results. For clarification a random transfer function is used for the green color channel. Piecewise linear transfer functions are employed for the other color channels. Note, that in contrast to pre-classification, post-classification reproduces the high frequencies contained within in the transfer function.*

required[23, 27]. A common approximation used in the visualization of volume data is the density-emitter model, which assumes that light is emitted and absorbed at each point in the volume while neglecting scattering and frequency dependent effects.

We denote a ray cast into the volume by $\vec{x}(t)$, and parameterize it by the distance t to the eye. The scalar value corresponding to this position on a ray is denoted by $s(\vec{x}(t))$. Since we employ an emission-absorption optical model, the volume rendering integral we are using integrates *absorption coefficients* $\tau(s(\vec{x}(t)))$ (accounting for the absorption of light), and *colors* $c(s(\vec{x}(t)))$ (accounting for light emitted by particles) along a ray.

The volume rendering integral can now be used to obtain the integrated "output" color C, subsuming both color (emission) and opacity (absorption) contributions along a ray up to a certain distance D into the volume (see Figure 4):

$$C = \int_0^D c(s(\vec{x}(t)))e^{-\int_0^t \tau(s(\vec{x}(t')))\,dt'}\,dt \qquad (2)$$

The equation denotes, that at each position in the volume, light is emitted according to the term $c(s(\vec{x}(t)))$, which is absorbed by the volume at all positions along a ray in front of the light emission position according to the term $\tau(s(\vec{x}(t')))$.

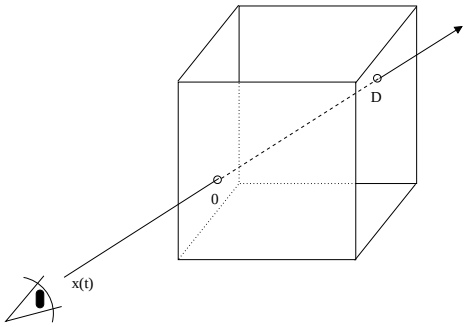

Figure 4: *Integration along a ray through the volume.*

By discretizing the integral, we can now introduce the opacity values A, "well-known" from alpha blending, by defining

$$A_i = 1 - e^{-\tau(s(\vec{x}(id)))d} \qquad (3)$$

Similarly, the color (emission) of the i-th ray segment can be approximated by:

$$C_i = c(s(\vec{x}(id)))d \qquad (4)$$

Having approximated both the emissions and absorptions

along a ray, we can now state the approximate evaluation of the volume rendering integral as (denoting the number of samples by $n = \lfloor D/d \rfloor$):

$$C_{approx} = \sum_{i=0}^{n} C_i \prod_{j=0}^{i-1} (1 - A_i) \tag{5}$$

Equation 5 can be evaluated iteratively by *alpha blending*[36, 3] in either back-to-front, or front-to-back order.

The following iterative formulation evaluates equation 5 in back-to-front order by stepping i from $n-1$ to 0:

$$C'_i = C_i + (1 - A_i) C'_{i+1} \tag{6}$$

2.4. Shading

In the above considerations we did not take the effects of external light sources into account. Instead, we assumed a simple shading, i.e. we identified the primary color assigned in the classification with $c(s(\vec{x}(t)))$.

The most popular local illumination model is the Phong model[35, 4], which computes the lighting as a linear combination of three different terms, an *ambient*, a *diffuse* and a *specular* term,

$$I_{\text{Phong}} = I_{\text{ambient}} + I_{\text{diffuse}} + I_{\text{specular}}. \tag{7}$$

Ambient illumination is modeled by a constant term, $I_{\text{ambient}} = k_a = const.$ Without the ambient term parts of the geometry that are not directly lit would be completely black. In the real world such indirect illumination effects are caused by light reflected from other surfaces.

Diffuse reflection refers to light which is reflected with equal intensity in all directions (*Lambertian* reflection). The brightness of a dull, matte surface is independent of the viewing direction and depends only on the *angle of incidence* φ between the direction $\vec{l}$ of the light source and the surface normal $\vec{n}$. The diffuse illumination term is written as $I_{\text{diffuse}} = I_p\, k_d\, \cos\varphi = I_p\, k_d\, (\vec{l} \cdot \vec{n})$. I_p is the intensity emitted from the light source. The surface property k_d is a constant between 0 and 1 specifying the amount of diffuse reflection as a material specific constant.

Specular reflection is exhibited by every shiny surface and causes so-called highlights. The specular lighting term incorporates the vector $\vec{v}$ from the object to the viewers eye into the lighting computation. Light is reflected in the direction of reflection $\vec{r}$ which is the direction of light $\vec{l}$ mirrored about the surface normal $\vec{n}$. For efficiency, the reflection vector $\vec{r}$ can be replaced by the halfway vector $\vec{h}$, $I_{\text{specular}} = I_p\, k_s\, \cos^n\alpha = I_p\, k_s\, (\vec{h} \cdot \vec{n})^n$. The material property k_s determines the amount of specular reflection. The exponent n is called the *shininess* of the surface and is used to control the size of the highlights.

3. Programmable Consumer Graphics Hardware

The typical architecture of todays consumer graphics processing units (GPUs) is characterized by a configurable rendering pipeline employing an object-order approach, that applies several geometric transformations on primitives likes points, lines and polygons (geometry processing), before the primitives are rasterized (rasterization) and written into the frame buffer after several fragment operations (see Figure 5). The design as a strict pipeline with only local knowledge allows lots of optimizations, for example the geometry stage is able to work on new vertex data, while the rasterization stage is processing earlier data. However, the current trend is to replace the concept of a configurable pipeline with the concept of a programmable pipeline. The programmability is introduced in the vertex processing stage as well as in the rasterization stage. We will mainly be concerned with programmable units in the rasterization stage, since the major tasks in texture-based volume rendering is done in this stage, i.e in volume rendering geometry processing of the proxy geometry is very simple while rasterization is quite complex.

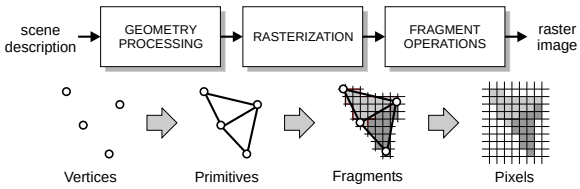

Figure 5: *The graphics pipeline that is used in common PC consumer graphics hardware.*

At the time of this writing (summer 2002), the two most important vendors of programmable GPUs are NVIDIA and ATI. The current state-of-the-art consumer graphics processors are the NVIDIA GeForce4 and the ATI Radeon 8500. In the following, we will focus on the programmable per-fragment operation extensions of these two manufacturers.

Traditional lighting calculations were done on a per-vertex basis, i.e. the primary (or diffuse) color was calculated at the vertices and linearly interpolated over the interior of a triangle by the rasterizer. One of the first possibilities for per-pixel shading calculations was introduced for computer games that employ pre-calculated light maps and decal maps for global illumination effects in walkable 3D scenes. The texels of the decal texture map are modulated with the texels of the light texture map. This model requires to apply two texture maps on a polygon, i.e. multi-textures are needed. The way the multiple textures are combined to a single RGBA value in traditional OpenGL multi-texturing is defined by means of several hard-wired *texture environment* modes.

The OpenGL multi-texturing pipeline has various drawbacks, particularly it is very inflexible and cannot accommodate the capabilities of today's consumer graphics hard-

ware. Starting with the original NVIDIA register combiners, which are comprised of a register-based execution model and programmable input and output routing and operations, the current trend is toward writing a *fragment shader* in an assembly language that is downloaded to the graphics hardware and executed for each fragment.

The NVIDIA model for programmable fragment shading currently consists of a two-stage model that is comprised of the distinct stages of *texture shaders*[14] and *register combiners*[12].

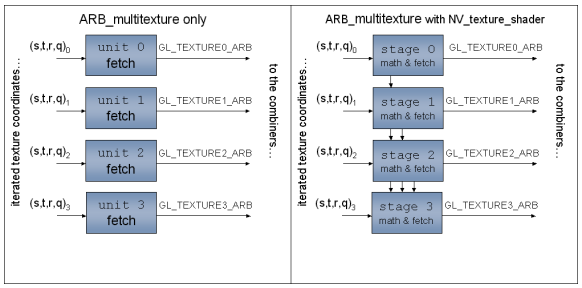

Figure 6: *Comparison of traditional multi-texturing and multi-texturing using NVIDIA's texture shader concept.*

Texture shaders are the interface for programmable texture fetch operations. For each of the four multi-textures a fetch operation can be defined out of one of 23 pre-defined *texture shader programs*, whereas the GeForce 4 offers 37 different such programs[15]. In contrast to traditional multi-texturing, the results of a previous texture fetch may be employed as a parameter for a subsequent texture fetch (see Figure 6). Additionally, each texture fetch operation is capable of performing a math operation before the actual texture fetch. A very powerful option are *dependent texture* fetches, which use the results of previous texture fetches as texture coordinates for subsequent texture fetches. An example for one of these texture shader programs is *dependent alpha-red texturing*, where the texture unit for which this mode is selected, takes the alpha and red outputs from a previous texture unit as 2D texture coordinates for a dependent texture fetch in the texture bound to the texture unit. Thus it performs a dependent texture fetch, i.e. a texture fetch operation that depends on the outcome of a fetch executed by another unit.

The major drawback of the texture shaders model is specifically that it requires to use one of several fixed-function programs, instead of allowing arbitrary programmability.

After all texture fetch operations have been executed (either by standard OpenGL texturing, or using texture shaders), the register combiners mechanism may be used for flexible color combination operations, employing a register-based execution model. The register combiners interface is exposed through two OpenGL extensions: `GL_NV_register_combiners`[12], and `GL_NV_register_combiners2`[13].

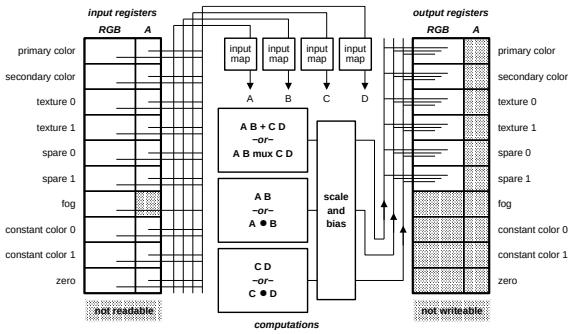

Figure 7: *A general combiner stage of NVIDIA's register combiner unit.*

Current NVIDIA GPUs feature eight general and one final combiner stage. Each general combiner stage (see Figure 7) has four input variables (A,B,C and D). These input variable can be occupied with one of the possible input parameters, e.g. the result of a texture fetch operation or the primary color. As all computation are performed in the range of -1 to 1 an input mapping is required for each variable, that maps the input colors to that range. A maximum of three math operations may be computed per general combiner stage, e.g. a component-wise weighted sum $AB+CD$ and two component-wise products AB, CD or two dot products $A \cdot B$ and $C \cdot D$. The result are modified by scale and bias operations and used in subsequent general combiner stages. The RGB and the Alpha channels are handled independently, i.e. alpha portion of each general combiner can perform a different math operation as the RGB portion.

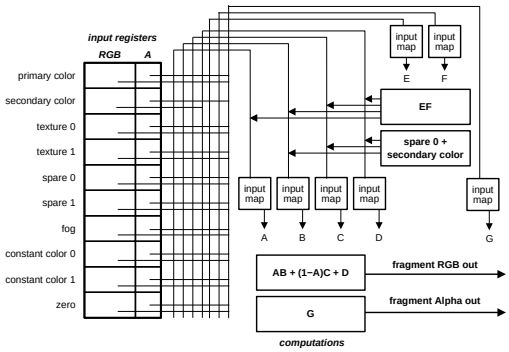

Figure 8: *The final combiner stage of NVIDIA's register combiner unit.*

After all enabled general combiner stages have been executed, a single final combiner stage (see Figure 8) generates the final fragment color, which is then passed on to fragment tests and alpha blending.

In contrast to the NVIDIA approach, fragment shading on the Radeon 8500 uses a unified model that subsumes both texture fetch and color combination operations in a single *fragment shader*[8] program. The fragment shader interface is exposed through a single OpenGL extension: `GL_ATI_fragment_shader`.

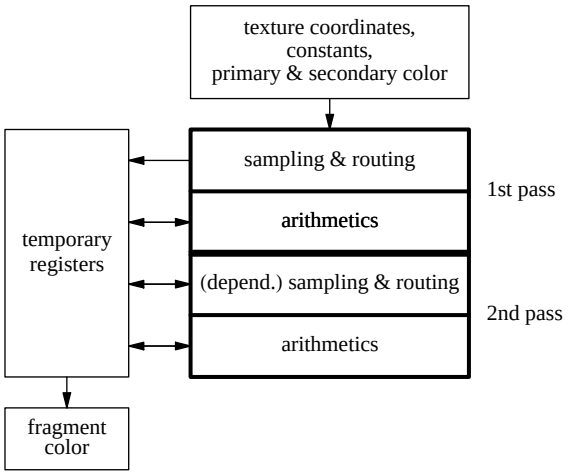

Figure 9: *The ATI fragment shader unit.*

The fragment shader unit is divided up into two (currently) or more passes, each of which may perform six texture sampling and routing operations followed by eight arithmetic calculations (see Figure 9). The second phase allows dependent texture fetches with the results from the first phase. Such results are stored in a set of temporary registers.

On the Radeon 8500, the input registers used by a fragment shader consist of six RGBA registers (`GL_REG_0_ATI` to `GL_REG_5_ATI`), corresponding to this architecture's six texture units. Furthermore, two interpolated colors, and eight constant RGBA registers (`GL_CON_0_ATI` to `GL_CON_7_ATI`) are available to provide additional color input to a fragment shader.

Each of the six possible sampling operations can be occupied with a texture fetch from the corresponding texture unit. Each of the eight fragment operations can be occupied with one of the following instructions:

- **MOV:** Moves one register into another.
- **ADD:** Adds one register to another and stores the result in a third register.
- **SUB:** Subtracts one register from another and stores the result in a third register.
- **MUL:** Multiplies two registers component-wise and stores the result in a third register.
- **MAD:** Multiplies two registers component-wise, adds a third, and stores the result in a fourth register.
- **LERP:** Performs linear interpolation between two registers, getting interpolation weights from a third, and stores the result in a fourth register.
- **DOT3:** Performs a three-component dot-product, and stores the replicated result in a third register.
- **DOT4:** Performs a four-component dot-product, and stores the replicated result in a third register.
- **DOT2_ADD:** The same as DOT3, however the third component is assumed to be 1.0 and therefore not actually multiplied.
- **CND:** Moves one of two registers into a third, depending on whether the corresponding component in a fourth register is greater than 0.5.
- **CND0:** The same as CND, but the conditional is a comparison with 0.0.

The components of input registers to each of these instructions may be replicated, and the output can be masked for each component, which allows for flexible routing of color components. Scaling, bias, negation, complementation, and saturation (clamp against 0.0) are also supported. Furthermore, instructions are issued separately for RGB and alpha components, although a single pair of RGB and alpha instructions counts as a single instruction.

In general, it can be said that the ATI fragment shader model is much easier to use than the NVIDIA extensions providing similar functionality, and also offers more flexibility with regard to dependent texture fetches. However, both models allow specific operations that the other is not able to do.

4. Texture-Based Volume Visualization

As discussed in the previous section, current consumer graphics hardware is based on an object-order rasterization approach, i.e. primitives (polygons, lines, points) are scan-converted and written pixel-per-pixel into the frame buffer. Since the volume data does not consists of such primitives *per se*, a proxy geometry is defined for each individual slice through the volume data. Each slice is textured with the corresponding data from the volume. The volume is reconstructed during rasterization on the slice polygon by applying a convolution of the volume data with a filter kernel as discussed in Section 2. The entire volume can be represented by a stack of such slices, if the number of slices satisfies the restrictions imposed by the Nyquist theorem.

There exist at least three different slicing approaches. View-aligned slicing in combination with 3D textures[5] is the most common approach (see Figure 10). In this case, the volume is stored in a single 3D texture and three texture coordinates from the vertices of the slices are interpolated over the inside of the slice polygons. These three texture coordinates are employed during rasterization for fetching filtered texels from the 3D texture map.

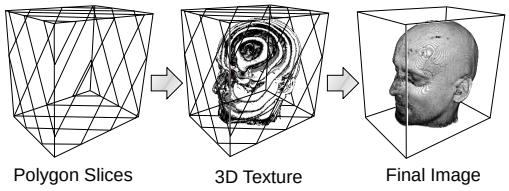

Figure 10: *The 3D texture-based approach uses view-aligned slices as proxy geometry.*

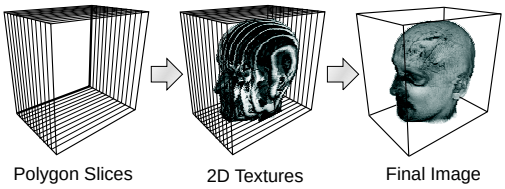

Figure 11: *The 2D texture-based approach uses object-aligned slices as proxy geometry.*

If only 2D texture mapping is supported by the graphics hardware, i.e. the hardware is able to perform bi-linear interpolation, the slices have to be aligned orthogonal with one of the the three major axes of the volume. For this so called object-aligned slicing (see Figure 11), the volume data is stored in several two-dimensional texture maps. To prevent unfavorable alignments of the slices with the viewers line of sight, that would allow the viewer to see in between individual slices, one slice stack for each major axis is stored. During rendering, the slice stack that is most perpendicular to the viewer's line of sight is chosen for rendering (see Figure 12).

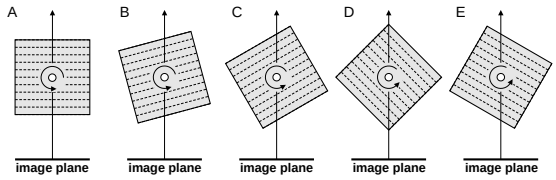

Figure 12: *Choosing the most perpendicular slice stack to the viewer's line of sight during object-aligned slicing. Between (C) and (D) another slice stack is chosen for rendering.*

There are major disadvantages when using object-aligned slices with standard 2D texturing. First, the volume must be tripled, which is critical especially in the context of limited memory of consumer graphics hardware. Second, the number of slices that are rendered is limited to the resolution of the volume, because the insertion of interpolated slice will increase the memory consumption. Typically, undersampling occurs most visibly on the side of the volume along the currently used major axis. Another disadvantage is, that switching from one slice stack to another when rotating the volume leads to an abrupt change of the currently used sampling points, which becomes visible as a popping effect (see Figure 13). Finally, the distance of sampling point depends on the viewing angle as outlined in Figure 14. A constant sampling distance is however necessary in order to obtain correct results.

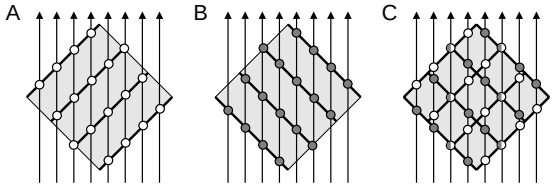

Figure 13: *Abrupt change of the location of sampling points, when switching from one slice stack (A) to another (B).*

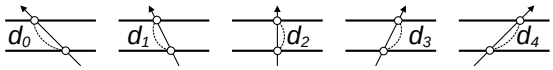

Figure 14: *The distance between adjacent sampling points depends on the viewing angle.*

However, almost all of these disadvantages are circumvented by using multitextures and programmable rasterization units. The basic idea is to render arbitrary, tri-linearly interpolated, object-aligned slices by mapping two adjacent texture slices to a single slice polygon by means of multitextures[37]. The texture environment of the two 2D textures performs two bi-linear interpolations whilst the third interpolation is done in the programmable rasterization unit. This unit is programmed to compute a linear interpolation of two bi-linearly interpolated texel from the adjacent slices. A linear interpolation in the fragment stage is implementable on a wide variety of consumer graphics hardware architectures. A register combiner setup for the NVIDIA GeForce series is illustrated in Figure 15. The interpolation factor α (variable D) is mapped into a constant color register and inverted by means of a corresponding input mapping to obtain the factor $1-\alpha$ (variable B). The two slices that enclose the position of the slice to be rendered are configured as texture 0 (variable A) and texture 1 (variable C). The combiner is configured to calculate $AB+CD$, thus the final fragment contains the linearly interpolated result corresponding to the specified fractional slice position.

With the aid of this extension it is now possible to freely adjust the sampling rate without increasing the required memory. Furthermore, by adapting the sampling distance to the viewing angle, the sampling rate is held constant, at least for orthogonal projections. There remains the problem

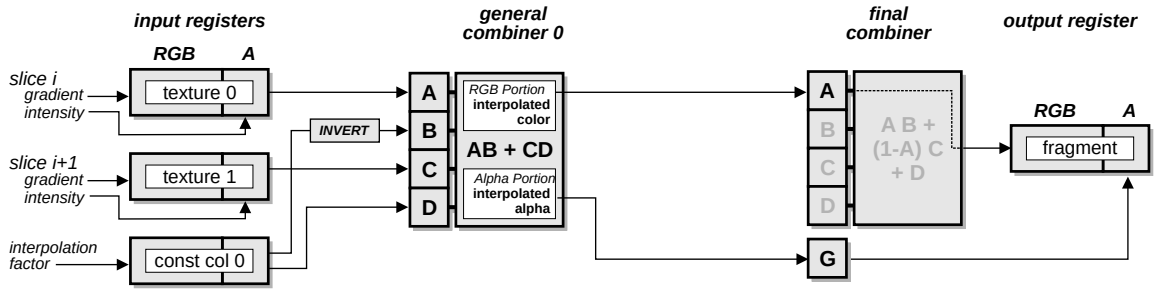

Figure 15: *Register combiners configuration for the interpolation of intermediate slices.*

of tripled memory consumption, yet this problem is solvable with a small modification of the above approach. For that purpose view-aligned slice polygons similar to a 3D texture-based approach are computed (see Figure 10). By intersecting view-aligned slices with a single object-aligned slice stack we obtain small stripes, each of which is bounded by two adjacent object-aligned slices (see Figure 16). Instead of a constant interpolation factor it is now necessary to linearly interpolate the factor from 0 to 1 from one slice to the adjacent slice on the stripe polygon. By enabling Gouraud shading for the stripe polygons and using a primary color of 0 and 1 for the corresponding vertices of the stripe polygon the primary color is linearly interpolated and therefore employed as the interpolation factor. An appropriate register combiner configuration is shown Figure 17.

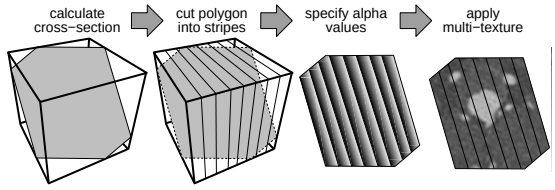

Figure 16: *Interpolation of view-aligned slices by rendering slab-polygons.*

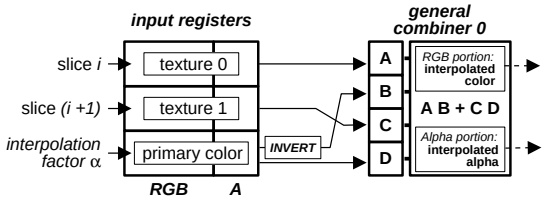

Figure 17: *Register combiners configuration for rendering tri-linearly interpolated stripe polygons.*

Consequently, a volume rendering algorithm similar to the 3D texture-based approach is possible with 2D textures and simple per-fragment operations. The only remaining problem is the subdivision of the slice polygons into stripes and the larger number of resulting polygons to be rendered. In order to circumvent a potentially expensive calculation of the stripe polygon vertices, an analog algorithm with object-aligned stripes can be implemented. Since volume rendering is usually fillrate or memory bandwidth bound and the number of polygons to be rendered is still quite limited for rendering stripe polygons, framerates similar to a 3D texture-based algorithm are possible.

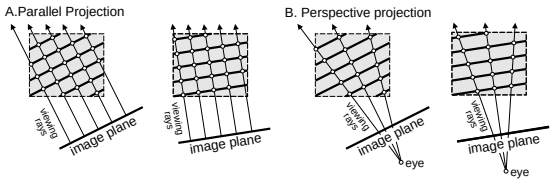

Figure 18: *For perspective projections the sampling rate varies for each viewing ray (left: parallel projection, right: perspective projection).*

A general disadvantage of a planar proxy geometry is the still varying sampling distance inside the volume for each viewing ray when using perspective projections (see Figure 18). This problem is circumventable by rendering concentric spherical shells around the camera position[21, 9] (see Figure 19). These shells are generated by clipping tessellated spheres against the viewing frustum and the volume bounding box. However, the setup and rendering of shells is more complicated than rendering planar slice polygons. Since the pixel-to-pixel difference, due to unequal sampling distances is often not visible, shells should only be used for extreme perspectives, i.e. large field-of-views.

All of the above slicing approaches render infinitesimal thin slices into the framebuffer, whereby the number of slices determines the sampling rate. In contrast to this slice-by-slice approach, pre-integrated volume rendering employs a slab-by-slab algorithm, i.e. the space in between two adjacent slices (or shells) is rendered into the framebuffer. Details on pre-integrated volume rendering will be presented in Chapter 8.

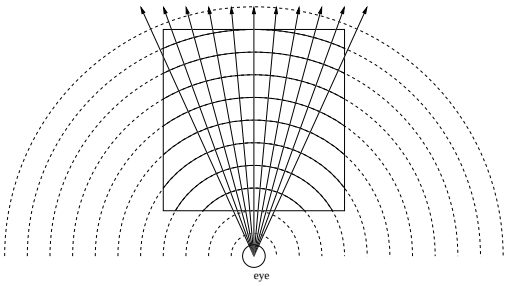

Figure 19: *Shell rendering provides a constant sampling rate for all viewing rays.*

A continuous volume is reconstructed on the slice polygons by the graphics hardware by applying a reconstruction filter. Current graphics hardware supports pre-filtering mechanisms like mip-mapping and anisotropic filtering for minification and linear, bilinear, and tri-linear filters for magnification. However, in Section 7 multi-textures and programmable per-fragment operations allow to apply high-quality filters to the volume data on-the-fly.

The classification step in hardware-accelerated volume rendering is either performed before or after the filtering step. The differences of pre- and post-classification were already demonstrated in Section 2.2. Pre-classification can be implemented in a pre-processing step by using the CPU to transform the scalar volume data into a RGBA texture containing the colors and alpha values from the transfer function. However, as the memory consumption of RGBA textures is quite heavy, NVIDIA's GPUs support the `GL_EXT_paletted_texture` extension, which uses 8 bit indexed textures and performs the transfer function lookup in the graphics hardware. Unfortunately, there does not exist an equivalent extension for post-classification. However, NVIDIA's `GL_NV_texture_shader` and ATI's `GL_ATI_fragment_shader` extension makes it possible to implement post-classification in graphics hardware using dependent texture fetches. For that, first a filtered scalar value is fetched from the volume, that is used as a texture coordinate for a dependent lookup into a texture containing the RGBA values of the transfer function. The volume texture and the transfer function texture are bound by means of multi-textures to a slice polygon. This also enables us to implement multi-dimensional transfer functions, i.e. a 2D transfer function is implemented with a 2D dependent texture and a 3D transfer function is possible by using a 3D dependent texture (see Section 6).

5. Illumination

Realistic lighting of volumetric data greatly enhances depth perception. For lighting calculations a per-voxel gradient is required, that is determined directly from the volume data by investigating the neighborhood of the voxel. Although newest graphics hardware will enable the calculation of the gradient at each voxel on-the-fly, in the majority of the cases the voxel gradient is pre-computed in a pre-processing step.

For scalar volume data the gradient vector is defined by the first order derivative of the scalar field $I(x,y,z)$, which is defined as by the partial derivatives of I in the x-, y- and z-direction:

$$\vec{\nabla} I \;=\; (I_x, I_y, I_z) \;=\; \left(\frac{\partial}{\partial x} I, \frac{\partial}{\partial y} I, \frac{\partial}{\partial z} I\right). \tag{8}$$

The length of this vector defines the local variation of the scalar field and is computed using the following equation:

$$\left\|\vec{\nabla} I\right\| \;=\; \sqrt{I_x{}^2 + I_y{}^2 + I_z{}^2}. \tag{9}$$

Note that for dot-product lighting calculations the gradient vector must normalized to a length of 1 to obtain correct results. Due to the fact that the gradients are normalized during pre-calculation on a per-voxel basis and interpolated tri-linearly in space, interpolated gradient will generally *not* be normalized. Thus, instead of dot-product lighting calculations, often light- or environment-maps are employed, because in this case the lookup of the incoming light purely depends on the direction of the gradient (see Section 5.3). Another possibility are normalization maps, that contain normalized gradients for each lookup with unnormalized gradients. A normalized gradient is then used for further lighting calculations.

One of the most common approaches to estimate the gradient is based on the first term of a Taylor expansion. With this *central differences method*, the directional derivative in the x-direction is calculated by evaluating

$$I_x(x,y,z) \;=\; I(x+1,y,z) - I(x-1,y,z) \quad \text{with} \quad x,y,z \in I\!N. \tag{10}$$

The derivatives in the other two directions are calculated respectively. A common way to store pre-computed gradient vectors for hardware-accelerated volume rendering is to store the three components of the gradient and the scalar value of the volume data as a RGBA texture:

$$\begin{array}{rcll} \vec{\nabla} I & = & \left(\begin{array}{c} I_x \\ I_y \\ I_z \end{array}\right) & \begin{array}{l} \longrightarrow \\ \longrightarrow \\ \longrightarrow \end{array} \begin{array}{l} \mathsf{R} \\ \mathsf{G} \\ \mathsf{B} \end{array} \\ I & & & \longrightarrow \; \mathsf{A} \end{array} \tag{11}$$

Use of gradients for the visualization of non-polygonal shaded iso-surfaces and for volume shading is demonstrated in the following subsections.

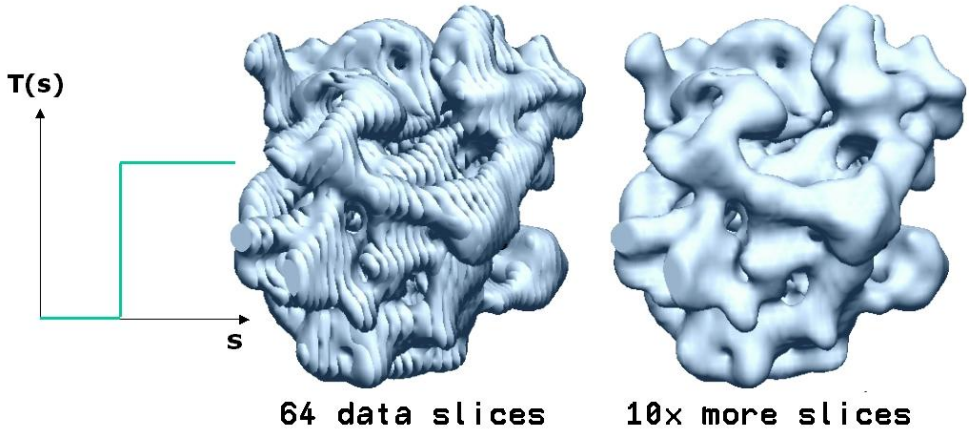

Figure 20: *A step in the transfer function or an alpha test* `GL_GEQUAL` *generates a single-sided iso-surface. Note, that the artifacts in the iso-surface are removed by very high sampling rates.*

5.1. Non-polygonal Iso-surfaces

Aside from the explicit reconstruction of threshold surfaces from volume data in a pre-processing step to rendering, there exist also techniques for the visualization of iso-surfaces during rendering. The underlying idea is to reject fragments from being rendered that lie over (and/or under) the iso-value, by applying a per-fragment test or assigning a transparent alpha-value by classification. Westermann proposed to use the OpenGL alpha test to reject fragments that do not actually contribute to an iso-surfaces with given iso-value[41]. The OpenGL alpha test allows us to define a alpha test reference value and a comparison test that rejects fragments based on the comparison of the alpha channel of the fragment and the given reference value. With the RGBA texture setup from the last section and by setting the alpha test reference value to the iso-value and the comparison test to `GL_GEQUAL` (greater or equal) one side of the iso-surface is visualized, while a test `GL_LEQUAL` (less or equal) visualizes the other side (see Figure 20). An alpha test with `GL_EQUAL` will lead to a two-sided iso-surface. Rezk-Salama et al.[37] extended Westermann's approach to use programmable graphics hardware for the shading calculations.

The similar result is obtained by using appropriate transfer functions for classification. Single-sided iso-surfaces are generated by using a step function from fully transparent to fully opaque values or vice versa, whilst peaks in the transfer function cause double-sided iso-surfaces. The thickness of the iso-surfaces is determined by the width of the peak. As peaks in the transfer function generate high frequencies in the classified volume data, holes in the surface will only be removed by employing very high sampling rates (see Figure 21). It should be noted, that only post-classification will fill the holes in the surfaces for high sampling rates, since pre-classification does not reconstruct the high frequencies of the transfer function on the slice polygons properly. For a solution of this problem we refer to Section 8.

5.2. Fragment Shading

With the texture setup introduced in the section, interpolated gradients are available during rasterization on a per-fragment basis. In order to integrate the Phong illumination model[35] into a single-pass volume rendering algorithm, dot-products and component-wise products must be computed with per-fragment arithmetic operations. This mechanism is provided by modern PC graphics hardware through advanced per-fragment operations. The standard OpenGL extension `EXT_texture_env_dot3` provides a simple mechanism for product calculations. Additionally, NVIDIA provides the mechanism of texture shaders and register combiners for texture fetching and arithmetic operations, whilst ATI combines the functionality of these two units into the fragment shader API.

Although similar implementations are possible with ATI's fragment shader extension, the following examples employ NVIDIA's register combiners. The combiner setup for diffuse illumination with two independent light sources is displayed in Figure 22. In this setup two textures are used for each slice polygon - one for the pre-calculated gradients and one for the volume scalars. The first combiner calculates the two dot products of the gradient with the directions of the light sources. The second combiner multiplies these dot products with the colors of the corresponding light sources and sums up the results. In the final combiner stage the RGB value after classification is added to the RGB result and the transparency of the fragment is set to the alpha value obtained from classification.

Specular and diffuse illumination is achieved by using the

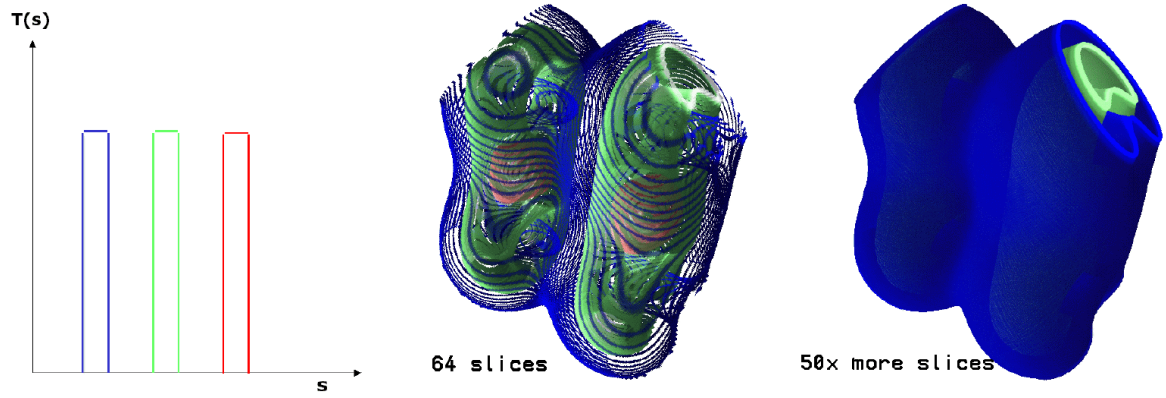

Figure 21: *Three peaks in the transfer function generate three double-sided iso-surfaces. Note, that the holes in the iso-surfaces are removed by very high sampling rates.*

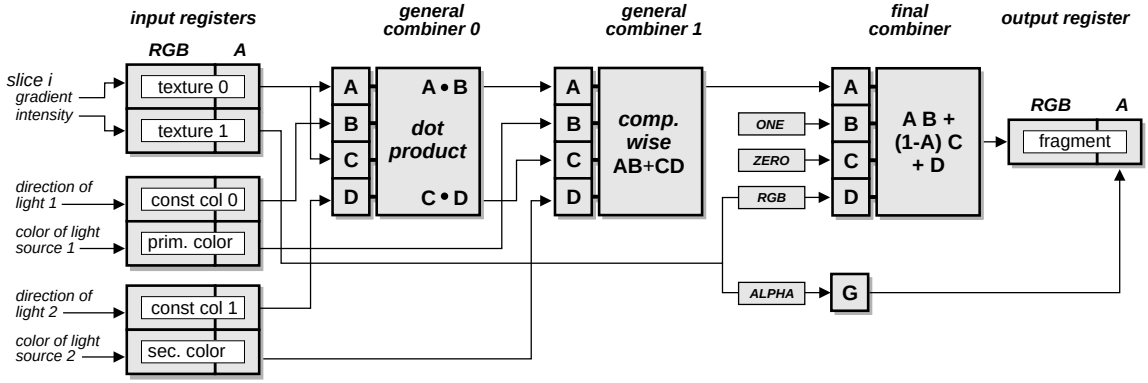

Figure 22: *NVidia register combiner setup for diffuse illumination with two independent light sources.*

register combiner setup displayed in Figure 23. Here, the first combiner calculates a single dot product of the gradient with a single light source direction. The result is used in the second general combiner for calculation of a diffuse lighting term and at the same time the dot product is squared. In the final combiner the squared dot product is squared twice before the ambient, diffuse and specular lighting terms are summed up.

5.3. Light/Reflection Maps

In the above lighting calculations the complexity of the lighting conditions is limited by the number of arithmetic operations (or combiner stages) that are allowed on a per-fragment basis. Additionally, due the current unavailability of square root and division operations in the fragment stage, the interpolated gradients cannot be renormalized. Dot-product lighting calculations with unnormalized gradients will lead to shading artifacts.

In order to allow more complex lighting conditions and prevent artifacts, the concept of reflection maps is applicable. A reflection map caches the incident illumination from all directions at a single point in space. The general assumption for the justification of reflection maps is, that the object to which the reflection map is applied is small with respect to the environment that contains it.

For a lookup into a reflection map on a per-fragment basis, a dependent texture fetch operation with the normal from a previous fetch is required. The coordinates for the lookup into a diffuse environment map are directly computed from the normal vector, whereas the reflection vectors for reflection map lookups are a function of both the normal vectors and the viewing directions.

Besides spherical reflection-maps, cube maps have become more and more popular within the past years (see Figure 24). In this case the environment is projected onto the six sides of a cube, which are indexed with the largest absolute component of the normal or reflection vector. The de-

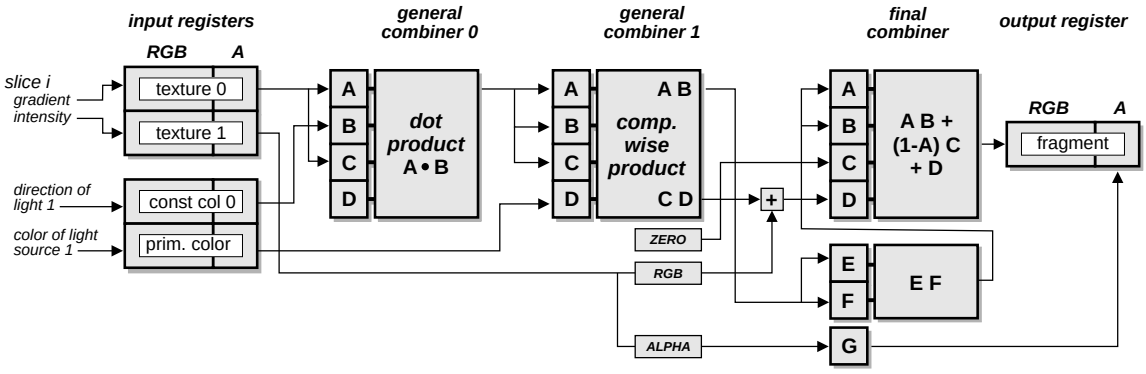

Figure 23: *NVidia register combiner setup for diffuse and specular illumination. The additional sum (+) is achieved using the* `spare0` *and* `secondary color` *registers of the final combiner stage.*

Figure 24: *Example of a cube environment map.*

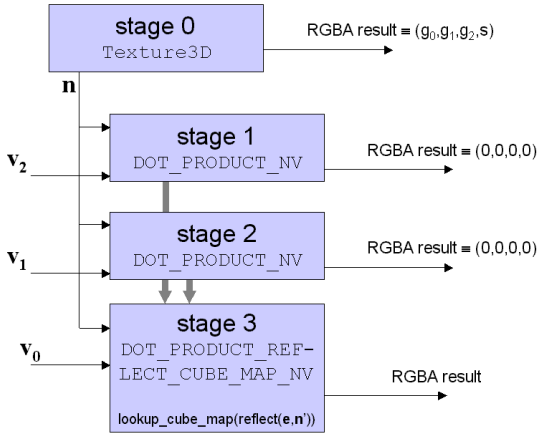

Figure 25: *Texture shader configuration for lookup into a reflection cube map. For a diffuse cube map lookup, the last fetch operation is replaced by a* `DOT_PRODUCT_TEXTURE_CUBE_MAP_NV` *operation.*

pendent lookup into such a cube map is supported in ATI's fragment shader and NVIDIA's texture shader OpenGL extensions. With NVIDIA's texture shader extension four texture units are required. The first texture unit fetches a filtered normal from the volume. Since the reflection map is generated in world coordinate space, the current modeling matrix is required to transform the normals into world space. The texture coordinates of the three remaining texture stages are used to pass the current modeling matrix $(s_i, t_i, r_i), i \in \{1,2,3\}$ and the eye vector (q_1, q_2, q_3) to the rasterization unit as vectors $v_i = (s_i, t_i, r_i, q_i), i \in \{1,2,3\}$ (see Figure 25). From this information the GPU calculates a normal in world space for a diffuse lookup or a reflection vector in world space for a reflection map lookup. The values from those lookups are finally combined using the register combiner extension (see Figure 26 for results).

Due to the fact, that the upload time of updated cube maps is negligible, cube maps are ideally suited for complex, dynamic lighting conditions.

5.4. Shadows

Shadows are another important visual clue for the perception of 3D structures. Behrens and Ratering[2] proposed a hardware model for computing shadows in volume data. They compute a second volume that captures the light arriving at a certain volume sample. During rasterization the colors after the transfer function are multiplied with these values to produce attenuated colors. However this approach suffers from blurry shadows and too dark surfaces due to the interpolation

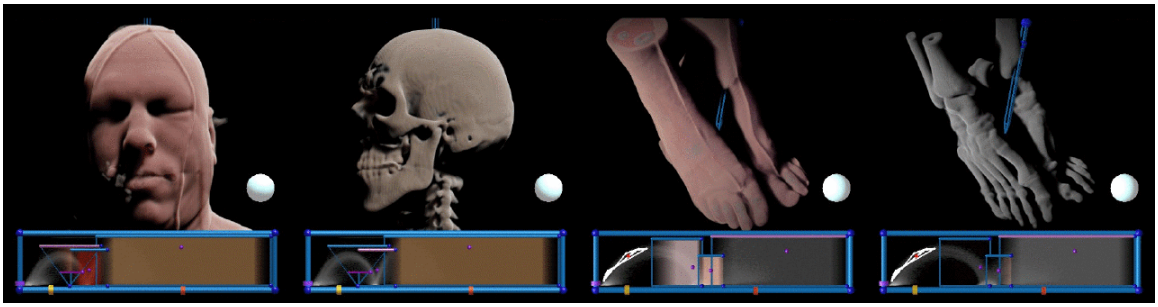

Figure 27: *Example volume renderings with shadow computations.*

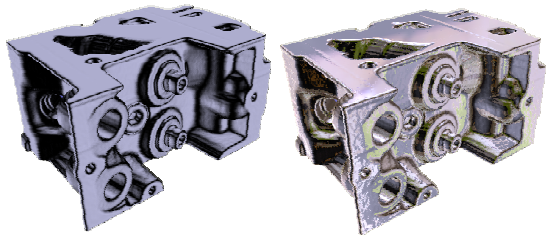

Figure 26: *Iso-surface of the engine block with diffuse cube map (left) and specular reflection map (right).*

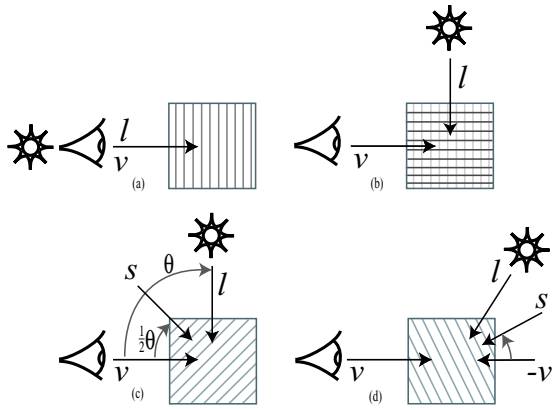

Figure 28: *Modified slicing approach for light transport computation.*

of the coarse light intensity volume. This effect is referred to as attenuation leakage.

An alternative approach was proposed by Kniss[18]. Instead of a shadow volume an offscreen render buffer accumulates the amount of light from the light's point of view. In order to render the same slices for light attenuation and volume slicing, Kniss proposed to slice the volume with a slice axis that is the halfway vector between the view and light directions. For a single headlight the same slices for accumulating opacity and for volume rendering are employed (Figure 28(a)). The amount of light arriving from the light source at a particular slice is one minus the accumulated opacity from the slices before it. Given the situation in Figure 28(b), normally a separate shadow volume must be created. With the approach of Kniss the halfway vector s between the light source direction l and the view vector v is used as slicing axis (Figure 28(c)). In order to have an optimal slicing direction for each possible configuration, the negative view vector is used for angles greater than 90 degrees between the view and the light vector (Figure 28(d)). Note, that in order to ensure a consistent sampling rate, the slice spacing along the slice direction must be corrected.

Given the above slicing approach and a front-to-back slice rendering order, a two-pass volume rendering algorithm will provide the desired results. First, a hardware-accelerated off screen render buffer is initialized with $1 - light\_intensity$. Alternatively, to create effects like spotlights, the buffer is initialized with an arbitrary image. Then each slice is rendered in a first pass from the observers point of view using the off screen buffer to modulate the brightness of samples. The two textures are applied to a slice polygon by means of multi-textures. Then, in the second pass, the slice is rendered from the light's point of view to calculate the intensity of the light arriving at the next layer. Light is attenuated by accumulating the opacity for each sample with the over operator. Optimized performance is achievable by utilizing a hardware-accelerated off screen buffer, that can be directly used as a texture. Such functionality is provided in OpenGL by the extension *render to texture*. The approach renders one light source per volume rendering pass. Multiple light source require additional rendering passes. The resulting images are weighted and summed. Figure 27 demonstrates results of the approach. As images created with the this approach often appear to dark because of missing scattering effects, Kniss also proposed to use the approach for scattering computations[19] to simulate translucent material.

6. Multi-Dimensional Classification

The term classification was introduced in Section 2 as the process of mapping scalar values from the volume to RGBA values using a one-dimensional transfer function. Figure 29 shows a one-dimensional histogram of a CT data set (see Figure 31), that allows to identify three basic materials.

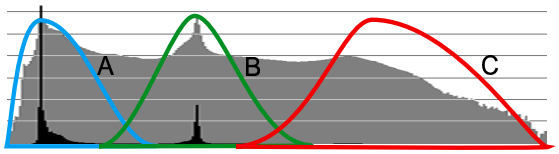

Figure 29: *A 1D histogram of the CT human head data set (black: log scale, grey: linear scale). The colored regions (A,B,C) identify basic materials.*

However, it is often advantageous to have a classification that depends on multiple scalar values. The two-dimensional histogram based on the scalar value and the first order derivative in Figure 30 allows to identify the materials as well as the material boundaries of the same CT data set (see Figure 31).

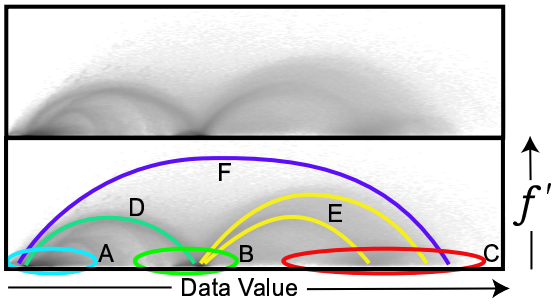

Figure 30: *A log-scale 2D histogram. Materials (A,B,C) and material boundaries (D,E,F) can be distinguished.*

Multi-dimensional transfer functions were first proposed by Levoy[22]. They provide a very effective way to extract materials and their boundaries for both scalar and multivariate data. However, the manipulation of multi-dimensional transfer functions is extremely difficult. Kindlman[16] proposed the semi-automatic generation of transfer functions whilst Kniss[17] proposed a set of direct manipulation widgets that make specifying such transfer functions intuitive and convenient.

Since identifying good transfer functions is a difficult task, the interactive manipulation of all transfer function parameters is mandatory. Fortunately, dependent texture reads, which were introduced in the latest generation of consumer graphics hardware, are particularly suited for an interactive classification with multi-dimensional transfer functions. Instead of the one-dimensional dependent texture from Section 4, the transfer function is transformed into a two- or

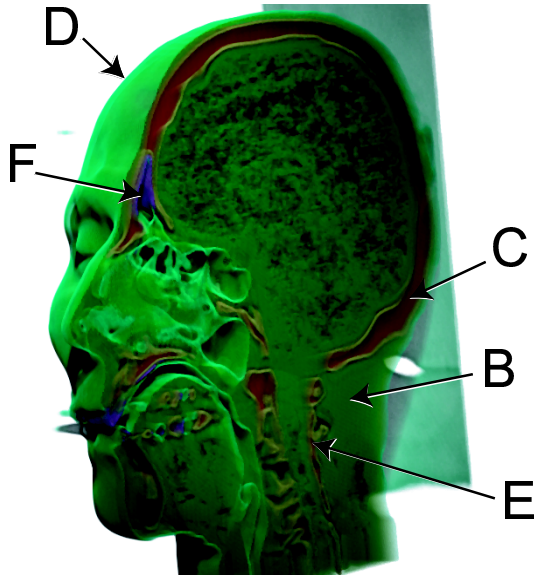

Figure 31: *CT scan of a human head showing the materials and boundaries identified by a two-dimensional transfer function.*

three-dimensional RGBA texture with color and opacity values from the transfer function. The volume itself is defined as a `LUMINANCE_ALPHA`-texture for two scalars per voxel and as a `RGB`-texture for three scalars per voxel.

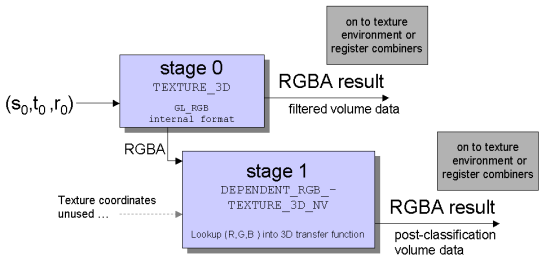

Figure 32: *Texture shader setup for a three-dimensional transfer function. Stage 0 fetches the per-voxel scalars while stage 1 performs a dependent texture lookup into the 3D transfer function.*

Figure 32 shows the texture shader setup for NVIDIA's GeForce4 chip. During rendering the first texture stage fetches filtered texels from the volume containing the three scalars defined for that voxel in the RGB components. Texture stage 1 performs dependent texture lookups with the RGB components from stage 0 as texture coordinates into the 3D transfer function volume. Consequently, a post-classification with the 3D transfer function is performed.

It should be noted, that a 3D dependent texture lookup is currently only supported by the `texture_shader3` extension of the GeForce4 chip. In contrast, both the ATI Radeon8500 and the NVIDIA GeForce3 and GeForce4 chips support 2D dependent lookups.

7. High-Quality Filtering

As outlined in Section 2, the accurate reconstruction of the original volume from the sampled volume data requires an appropriate reconstruction filter. Unfortunately, current graphics hardware only supports linear filters for magnification. However, Hadwiger et al.[10] have shown that multi-textures and flexible rasterization hardware allow to evaluate arbitrary filter kernels during rendering.

The filtering of a signal can be described as the convolution of the signal function s with a filter kernel function h:

$$g(t) = (s * h)(t) = \int_{-\infty}^{\infty} s(t - t') \cdot h(t')dt' \qquad (12)$$

In the discretized form this leads to

$$g_t = \sum_{i=-I}^{+I} s_{t-i} h_i \qquad (13)$$

with the half width of the filter kernel denoted by I. Basically this means, that we have to collect the contribution of neighboring input samples multiplied by the corresponding filter values to get a new filtered output sample. Instead of this *gathering* approach, Hadwiger et al. use a *distributing* approach for a hardware-accelerated implementation. That is, the contribution of an input sample is *distributed* to its neighboring samples, instead of the other way. The order was chosen, since this allows to collect the contribution of a single relative input sample for all output samples simultaneously. The term *relative input sample* denotes the relative offset of an input sample to the position of an output sample. The final result is obtained by adding the result of multiple rendering passes, whereby the number of input samples that contribute to an output sample determine the number of passes.

Figure 33 demonstrates this in the example of a one-dimensional tent filter. As one left-handed and one right-handed neighbor input sample contribute to each output sample, a two-pass approach is necessary. In the first pass, the input samples are shifted right half a voxel distance by means of texture coordinates. The input samples are stored in a texture-map that uses nearest-neighbor interpolation and is bound to the first texture stage of the multi-texture unit (see Figure 34). Nearest-neighbor interpolation is needed to access the original input samples over the complete half extend of the filter kernel. The filter kernel is divided into

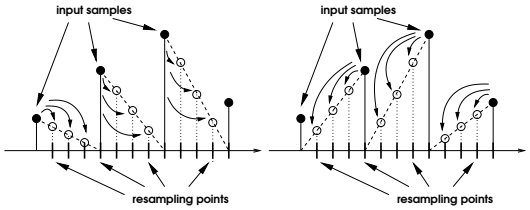

Figure 33: *Distributing the contributions of all "left-hand" (a), and all "right-hand" (b) neighbors, when using a tent filter as a simple example for the algorithm.*

two tiles. One filter tile is stored in a second texture map, mirrored and repeated via the `GL_REPEAT` texture environment. This texture is bound to the second stage of the multi-texture unit. During rasterization the values fetched by the first multi-texture unit are multiplied with the result of the second multi-texture unit. The result is added into the frame buffer. In the second pass, the input samples are shifted left half a voxel distance by means of texture coordinates. For a symmetric filter the same unmirrored filter tile is reused. The result is again added to the frame buffer to obtain the final result.

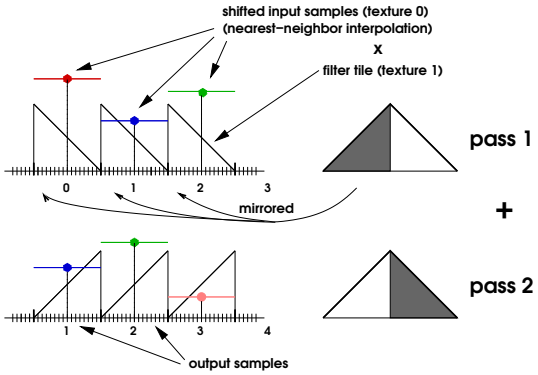

Figure 34: *Tent filter (width two) used for reconstruction of a one-dimensional function in two passes. Imagine the values of the output samples added together from top to bottom.*

If a given hardware architecture supports $2n$ multi-textures, the number of required passes can be reduced by n. That is, two multi-texture units are calculating the result of a single pass. Note, that the method outlined above is not considering area-averaging filters, since it is assumed that magnification is desired instead of minification. For minification pre-filtering approaches like mip-mapping are advantageous. Figure 35 demonstrates the benefit of bi-cubic filtering using a B-spline filter kernel over a standard bi-linear interpolation.

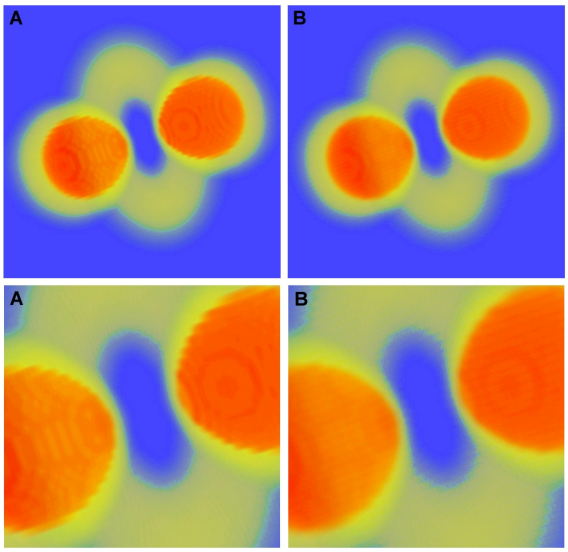

Figure 35: *Using a high-quality reconstruction filter for volume rendering. This image compares bi-linear interpolation of object-aligned slices (A) with bi-cubic filtering using a B-spline filter kernel (B).*

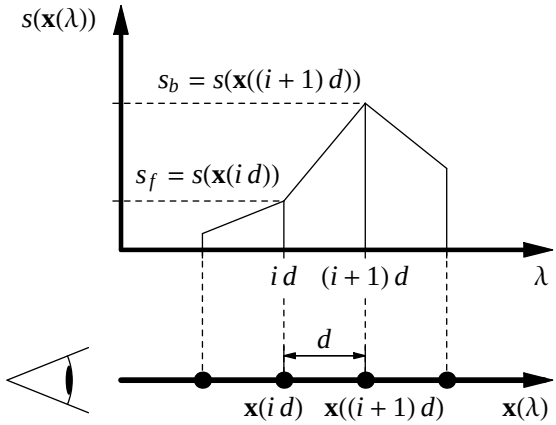

Figure 36: *Scheme of the parameters determining the color and opacity of the i-th ray segment.*

8. Pre-Integrated Volume Rendering

Obviously, as outlined in Section 2, post-classification will reproduce high frequencies of the transfer function. However, as observed in Section 5.1, high frequencies (e.g. iso-surface peaks) are only reproduced on slice polygons. In order to visualize details of the transfer function in between slice polygons, additional tri-linearly interpolated slice must be rendered. As this demands higher rasterization requirements from the graphics hardware, framerates considerably decrease.

8.1. 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 the color after the classification $\tilde{c}\big(s(\mathbf{x})\big)$ and an extinction coefficients after the classification $\tau\big(s(\mathbf{x})\big)$ resulting from non-linear transfer functions. On the other hand, the goal of *pre-integrated classification*[38] 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 36.) More precisely spoken, the opacity α_i of the i-th segment is approximated by

$$\begin{aligned}\alpha_i &= 1-\exp\left(-\int_{i\,d}^{(i+1)d}\tau\Big(s\big(\mathbf{x}(\lambda)\big)\Big)\,\mathrm{d}\lambda\right)\\ &\approx 1-\exp\left(-\int_0^1\tau\big((1-\omega)s_f+\omega s_b\big)d\,\mathrm{d}\omega\right).\end{aligned}\quad(14)$$

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:

$$\begin{aligned}\tilde{C}_i &\approx \int_0^1\tilde{c}\big((1-\omega)s_f+\omega s_b\big)\\ &\quad\times\exp\left(-\int_0^\omega\tau\big((1-\omega')s_f+\omega' s_b\big)d\,\mathrm{d}\omega'\right)d\,\mathrm{d}\omega\end{aligned}\quad(15)$$

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 the following Equation:

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

with colors $\tilde{C}_i$ pre-computed according to Equation (15) and opacities α_i pre-computed according to Equation (14). For non-associated color transfer function, i.e., when substituting $\tilde{c}(s)$ by $\tau(s)c(s)$, we will also employ Equation (14) for the approximation of α_i and the following approximation of

the associated color $\tilde{C}_i^{\tau}$:

$$\tilde{C}_i^{\tau} \approx \int_0^1 \tau\big((1-\omega)s_f+\omega s_b\big)c\big((1-\omega)s_f+\omega s_b\big) \times \exp\Big(-\int_0^{\omega}\tau\big((1-\omega')s_f+\omega' s_b\big)d\,\mathrm{d}\omega'\Big)d\,\mathrm{d}\omega \quad (16)$$

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 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.

One of the major disadvantages of the pre-integrated classification is the need to integrate a large number of ray-segments for each new transfer function dependent on the front and back scalar value and the ray-segment length. Consequently, an interactive modification of the transfer function is not possible. Therefore several modifications to the computation of the ray-segments were proposed[7], that lead to an enormous speedup of the integration calculations. However, this requires to neglect the attenuation within a ray segment. Yet, this is a common approximation for post-classified volume rendering and well justified for small products $\tau(s)d$. The dimensionality of the lookup table can easily be reduced by assuming constant ray segment lengths d. This assumption is correct for orthogonal projections and view-aligned proxy geometry. It is a good approximation for perspective projections and view-aligned proxy geometry, as long as extreme perspectives are avoided. This assumption is correct for perspective projections and shell-based proxy geometry. In the following hardware-accelerated implementation, two-dimensional lookup tables for the pre-integrated ray-segments are employed, thus a constant ray segment length is assumed.

8.2. Texture-Based Pre-Integrated Volume Rendering

The utilization of flexible graphics hardware for a hardware-accelerated implementation of pre-integrated volume rendering for volume data on cartesian grids was first proposed in [7]. 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 37), 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. Thus, the scalar values along a viewing ray of both slices (front and back) are fetched from texture maps during the rasterization of the polygon for one slab. These two scalar values are utilized as texture coordinates for a third

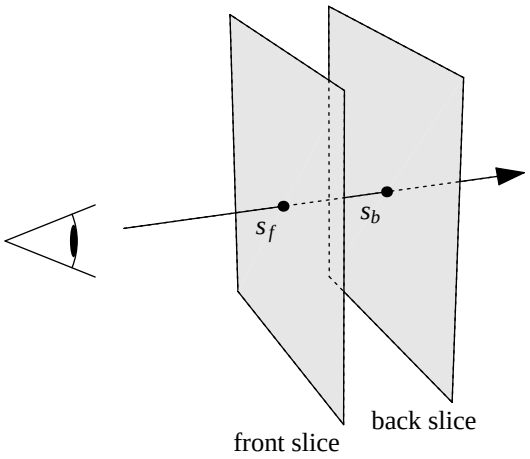

Figure 37: *A slab of the volume between two slices. The scalar value on the front (back) slice for a particular viewing ray is called s_f (s_b).*

dependent texture fetch operation. This fetch performs the lookup of pre-integrated colors and opacities from a two-dimensional texture map.

The opacities of the dependent texture map are calculated according to Equation (14), while the colors are computed according to Equation (15) if the transfer function specifies associated colors $\tilde{c}(s)$, and Equation (16) if it specifies non-associated colors $c(s)$. In either case the compositing Equation (5) is used for blending as the dependent texture map always contains associated colors.

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, this operation cannot be used as the coordinates are fetched from two separate 2D textures. Instead, as a workaround, 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 is used (see Figure 38). 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 $\vec{v} = (1,0,0)^T$. 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 first dot product is set to use the fetched front texel values as previous texture stage, the second uses the back texel value . In this approach, the second dot prod-

uct performs the texture lookup into our dependent texture via texture coordinates obtained from two different textures. As ATI's fragment shader OpenGL extension is more flexible than the NVIDIA counterpart, the more simple setup is possible. The front and back scalar values are fetched accordingly to the NVIDIA setup. After the fetches the two scalar value are contained in the red components of two temporary registers. Then the scalar value from one of temporary registers is moved to the green component of the other register with a `GL_MOV_ATI` command. In the second phase of the fragment shader program, a 2D dependent texture fetch with the red and green components of this temporary register fetches the pre-integrated ray-segment from a texture map.

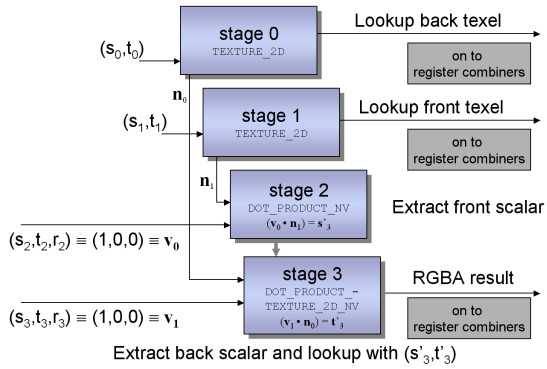

Figure 38: *Texture shader setup for dependent 2D texture lookup with texture coordinates obtained from two source textures. If 3D textures are employed, the first two fetch operations are replaced by the corresponding 3D texture fetches.*

For direct volume rendering without lighting the fetched texel from the last dependent texture operation is used without further processing and blended into the frame buffer with the OpenGL blending function `glBlendFunc(GL_ONE,GL_ONE_MINUS_SRC_ALPHA)`. A comparison of the results of pre-classification, post-classification and pre-integrated classification is shown in Figure 39.

Besides the suitability of high-frequency transfer functions for the evaluation of volume rendering quality, they also allow to identify homogeneous regions and small variations in the volume data (see Figure 40, right). Random transfer functions are a way to visualize all iso-surfaces in the volume data at once, because a random transfer function consists of a large number of peaks, each of which represents a single iso-surface. Homogeneous regions have a low iso-surface density. In contrast, large variations, i.e. large gradients are identifiable by applying gradient-weighted opacity. The usual way to implement gradient weighting is to

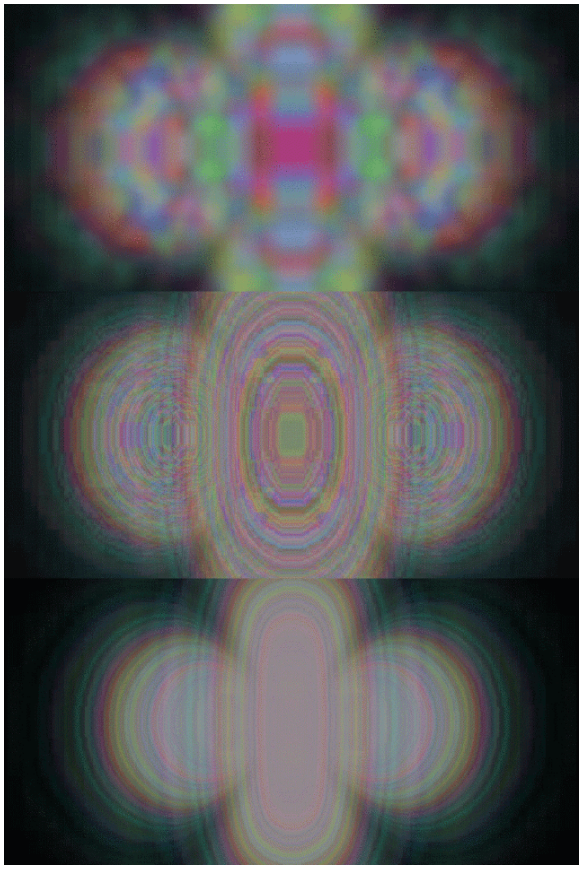

Figure 39: *Comparison of the results of pre-, post- and pre-integrated classification for a random transfer function. Note, that pre-classification does not reproduce high frequencies of the transfer function, that post-classification reproduces the high frequencies on the slice polygons, but pre-integrated classification produces the best visual result due to the reconstruction of high frequencies in the volume.*

store a pre-computed gradient-magnitude per voxel and to use a two-dimensional transfer function to enhance boundary structures. However, a directional derivate in the view-direction is implicitly contained within the pre-integrated volume rendering scheme, simply by assigning higher opacity to texels that are further away from the diagonal of the dependent texture containing the pre-integrated ray segments (see Figure 40, left). Note, that entries near the diagonal of the dependent texture correspond to ray segments with similar front and back scalar values, i.e. low gradient magnitude, whereas entries further away from the diagonal correspond to high directional gradient magnitude.

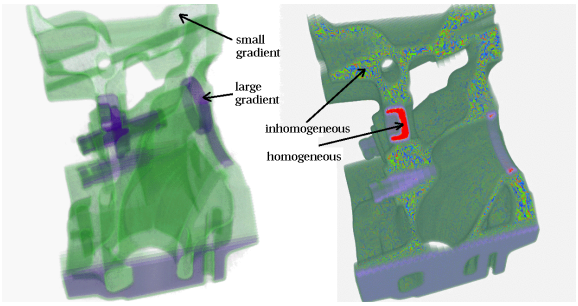

Figure 40: *Gradient-weighted opacity allows to identify regions of big variation in the data (left), while high-frequency transfer functions allow to identify small variations (right). Note, that homgeneous and inhomogeneous regions inside the engine data set can easily be identified.*

8.3. Iso-surfaces and Shading

One of the additional advantages of pre-integrated classification is the possibility of classifying each ray segment in the pre-processing step regarding the existence of iso-surfaces in that segment. An iso-surface is contained within a ray segment, if the iso-value lies in between the scalar values at the front and the back slice position along the ray. Just by initializing a transparent dependent texture and then coloring pixels in this texture with the color and alpha value that correspond to iso-surfaces that intersects the ray segments, the rendering of iso-surfaces is possible. Certainly this is not restricted to a single iso-surface per ray segment. If more than one iso-surface intersects a ray segment, the iso-surfaces are blended respectively and the resulting color and alpha value is used for the corresponding pixel in the dependent texture. Additionally, the back and front of the iso-surface can be colored independently. This is demonstrated in Figure 41, where a dependent texture that is used for visualizing 10 independently colored iso-surfaces at once. Certainly, visualizing multiple isosurfaces at once has no impact on frame rates, as this only requires to modify the dependent texture.

For shading calculations it is common to employ RGBA textures, that hold the volume gradient in the RGB components and the volume scalar in the ALPHA component. As dot3 dot-products on the NVIDIA architecture are required to extract the front and back volume scalar from a RGBA texel, the scalar data has to be stored in one of the RGB components (here in red). The first gradient component is stored in the ALPHA component in return. In the register combiners the red and alpha values are swapped back for shading calculations with RGB gradients. This workaround is not necessary on the ATI Radeon8500 board, since this architecture allows more flexible per-fragment operations, i.e. the scalar value in the alpha channel can be simply moved to another color channel for a dependent texture fetch.

After the color and alpha values have been fetched from the dependent texture, the gradients from the front and the back slice are available for shading calculations. Note, that due to limited amount of fetch operations on the current NVIDIA GPUs and the limited amount of dependency phases on the ATI GPUs, light maps cannot be employed for shading. Instead dot-product shading is used to evaluate an ambient, diffuse and specular lighting term. The gradient is linearly interpolated to the position of the intersection of the iso-surfaces inside a ray segment. If multiple iso-surfaces are contained within a ray segment, the gradient is interpolated to the position of the most front iso-surface or a weighted combination of multiple iso-surface gradients is used, if multiple semi-transparent iso-surfaces are visible[28]. The result of such a shading calculation is shown in Figure 41.

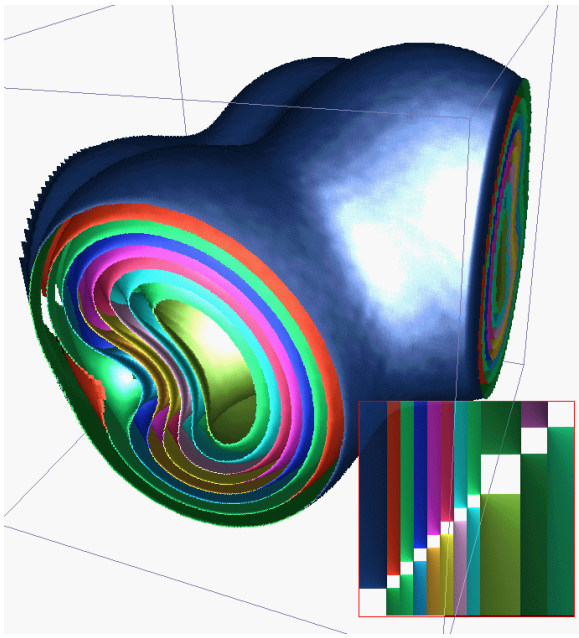

Figure 41: *Multiple shaded non-polygonal iso-surfaces that were extracted with the corresponding dependent texture (bottom, right). Note, that the front and back face of the iso-surfaces have different colors.*

Beyond that, volume shading[29] can easily be integrated into the pre-integration scheme. As for iso-surfaces, RGBA textures with gradients and scalar values are employed. Instead of the special dependent iso-surface texture, the pre-integrated dependent texture is required again. For shading, the gradient is averaged between the front and back gradient. Figure 42 shows the result of pre-integrated volume shading.

The volume shading approach is also useful as an alternative approach for iso-surface visualization. As discussed in Section 5.1 a peak in the transfer function defines a double-sided iso-surface. Due to the pre-integration of ray-segments, the iso-surface resulting iso-surfaces do not

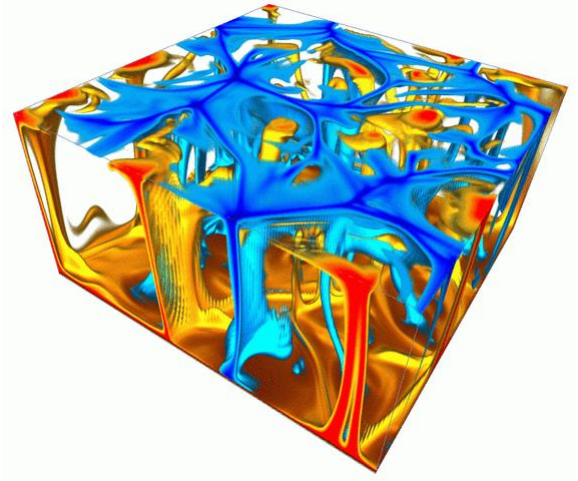

Figure 42: *Pre-integrated volume shaded rendering of convection flow in the earth's crust.*

have holes as opposed to post-classification (see Figures 43 and 21 for a comparison).

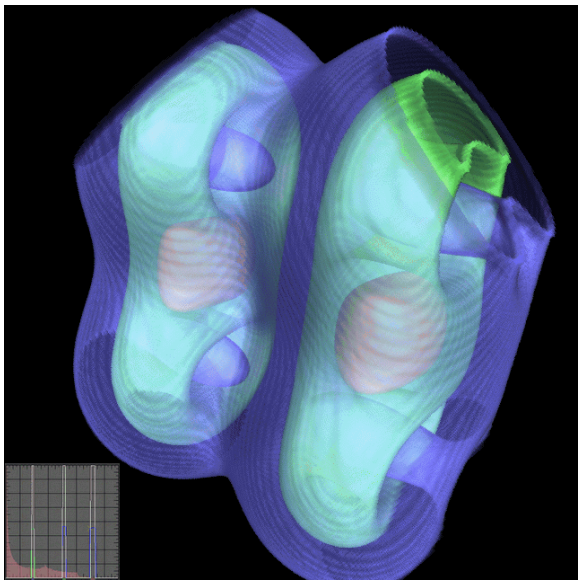

Figure 43: *Visualization of iso-surfaces using transfer function peaks (bottom, left) and pre-integrated shaded volume rendering does not lead to holes in the iso-surfaces for a low number of slices as opposed to post-classification.*

9. Large Volumes

One of the main limitations of employing current PC consumer graphics hardware for volume rendering is the relatively small amount of available on-board texture memory. Currently, a maximum of 128 MB is supported by NVIDIA's GeForce4 and ATI's Radeon8500 boards. Due to the quite slow bus bandwidth of current PC architectures, the utilization of the main memory in combination with a bricking[5] approach is often impractical. Although texture compression is available in current consumer graphics hardware for quite a time in form of S3's texture compression extensions `EXT_texture_compression_s3tc` and the equivalent extension `NV_texture_compression_vtc` for 3D textures, the compression ratios and quality is quite limited. Multi-resolution methods[39, 21] still have not provided satisfactory results.

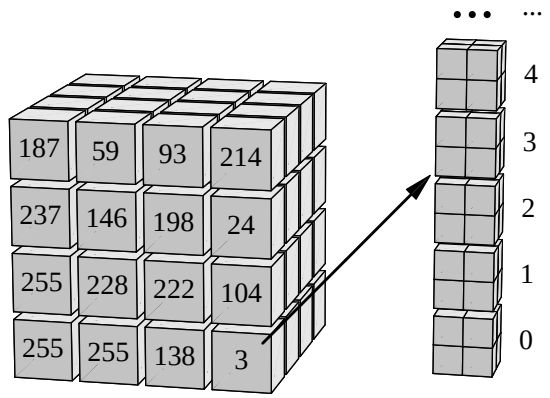

Figure 44: *Data structure for vector quantization of volume textures. Each byte of the index data specifies one data block consisting of* $2 \times 2 \times 2$ *voxels.*

However, due to indirect memory access functionality provided by dependent texture fetch operations of modern PC graphics hardware, decoders for compressed volume data can directly be integrated into the rasterization pipeline. Kraus et al.[20] proposed to use dependent texture fetches for adaptive texture maps and vector-quantization of volume data. The basic idea of vector quantization of volume data is illustrated in in Figure 44. Instead of the original volume data, the rendered 3D texture contains indices that reference one vector of voxels in a codebook. In the implementation of Kraus, the codebook includes 256 vectors consisting of 8 bytes corresponding to $2 \times 2 \times 2$ voxels of the original volume data. Thus, each cell of the index data specifies the complete data of eight voxels with just one byte, i.e. the compression ratio is about $8 : 1$ since the size of the relatively small codebook may be ignored. The codebook is computed in a pre-processing step with the help of one of the numerous vector quantization algorithms.

Since the dependent fetch from the codebook is done after the filtering step, currently nearest neighbor interpolation is employed. In a hardware implementation on future graphics hardware, a vector quantization decoding unit would be

placed in front of the texture filtering unit to facilitate trilinear interpolation.

10. Volumetric FX

In return for the advantages the scientific visualization community takes from the rapid development in the consumer graphics hardware, we can provide advanced algorithms that facilitate new volumetric effects in computer games. In order to capture the characteristics of many volumetric objects such as clouds, smoke, trees, hair, and fur, high frequency details are essential. Ebert's[6] approach for modeling clouds uses a coarse technique for modeling the macrostructure and uses procedural noise-based simulations for the microstructure (see Figure 45). This technique was adapted by Kniss[19] to interactive volume rendering through two volume perturbation approaches which are efficient on modern graphics hardware. The first approach is used to perturb texture coordinates and is useful to perturb boundaries in the volume data, e.g. the boundary of the cloud in Figure 45. The second approach perturbs the volume itself, which has the effect that materials appear to have inhomogeneities.

Both volume perturbation approaches employ a small 3D-perturbation volume with 32^3 voxels. Each texel is initialized with four random 8-bit numbers, stored as RGBA components, and blurred slightly to hide the artifacts caused by trilinear interpolation. Texel access is then set to repeat. An additional pass is required for both approaches due to limitations imposed on the number of textures which can be simultaneously applied to a polygon, and the number of sequential dependent texture reads permitted. The additional pass occurs before the steps outlined in the previous section. Multiple copies of the noise texture are applied to each slice at different scales. They are then weighted and summed per pixel. To animate the perturbation, they add a different offset to each noise texture's coordinates and update it each frame.

In the case of the modification of the location of the data access for the volume, the three components of the noise texture form a vector, which is added to the texture coordinates for the volume data per pixel. Offset textures in current graphics hardware would be ideally suited to add the noise vector to each per-fragment texture coordinate. Unfortunately, NVIDIA's texture shader extension does not facilitate 3D offset textures. Instead, the perturbed data is rendered to a pixel buffer that is used instead of the original volume data. Figure 46 illustrates this process. Notice that the high frequency content is created by allowing the noise texture to repeat.

In the case of the modification of the volume itself, the scalar value for each fragment is modified with the weighted sum of the three noise textures. Then this modified scalar is employed for the transfer function lookup.

Figure 47 shows a fireball that is rendered with a small modification of the pre-integrated volume rendering approach from Section 8. Instead of a indexed 3D-texture containing just the scalar values from a single volume, a RGB texture is employed, that contains the volume scalars in the red channel, a low-frequency noise texture in the green channel and a high-frequency noise texture in the blue color channel. If we apply texture coordinates (α, β, γ) instead of the constant $(1, 0, 0)$ texture coordinates as outlined in Figure 38, the three volumes contained in the color channels are weighted using the dot-product before the dependent texture lookup is performed. Thus, if we store a radial distance volume in the red color channel, it is perturbed. Flames are animated by changing the weighting factors (α, β, γ), while an outwards movement is achieved by color cycling the transfer function.

Figure 45: *Procedural clouds. The image on the top shows the underlying data, 64^3. The center image shows the perturbed volume. The bottom image shows the perturbed volume lit from behind with low frequency noise added to the indirect attenuation to achieve subtle iridescence effects.*

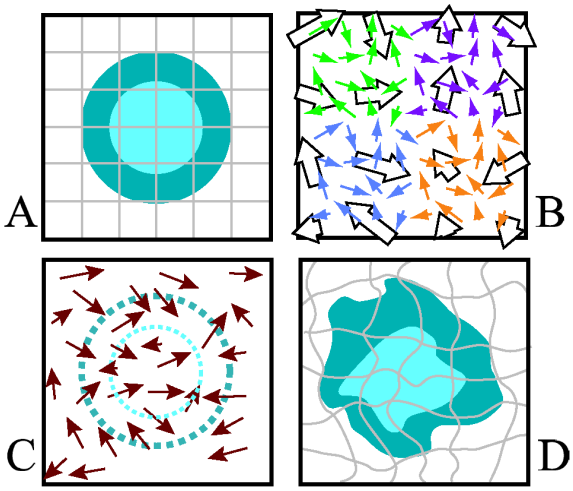

Figure 46: *An example of texture coordinate perturbation in 2D. A shows a square polygon mapped with the original texture that is to be perturbed. B shows a low resolution perturbation texture applied to the polygon multiple times at different scales. These offset vectors are weighted and summed to offset the original texture coordinates as seen in C. The texture is then read using the modified texture coordinates, producing the image seen in D.*

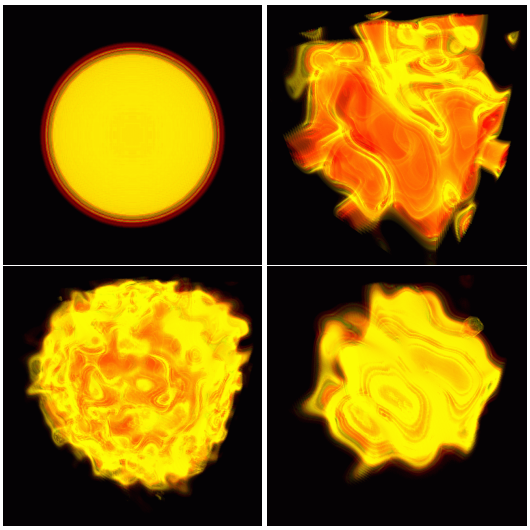

Figure 47: *Pre-Integrated volume rendering of a fireball. The fireball effect is obtained by mixing different volumes during rendering. (1) Radial distance volume with high-frequency fire transfer function, (2) Perlin-Noise-Volume with fire-transfer function, (3)Weighted combination of the distance volume and two Perlin-Noise volumes, (4) Like (3), but with higher weights for the Perlin-Noise-volumes.*

11. Limitations

Even though there has been a enormous development in the consumer graphics market in the last few years, current low-cost graphics hardware still has its drawbacks.

First, the computational precision of current PC graphics hardware is quite limited, especially in the rasterization stage of the rendering pipeline. Framebuffers currently support 32 bits, 8 bits per color channel. NVIDIA's register combiners extension has 9 bits precision for a range from -1 to 1, while the texture shader unit has floating point precision. ATI's fragment shader unit supports 16 bits precision for a range of values from -8 to 8, 13 bit in the range -1 to 1. Note, that for calculations in 16 bit precision, it is necessary to expand the values to the full range of -8 to 8. Textures currently have 8 bit precision per color channel. The first step towards higher precision is NVIDIA's `HILO` texture format, that supports two channels per texture with 16 bits precision. The result limited precision are artifacts, that can be frequently observed in hardware-accelerated renderings. One solution was already introduced-the pre-computation of values with the full precision of the CPU in a pre-processing step, as employed for pre-integrated volume rendering.

Another drawback is the limited programmability of current units in the rendering pipeline. NVIDIA supports four texture fetches in a pass, followed by 8 general combiner and one final combiner operation. ATI's fragment shaders currently support 2 phases, each with 8 texture fetches followed by 8 math operations. More complex operations require to split the computation into multiple rendering passes. Hardware-accelerated offscreen buffers (PBuffers) and the possibility to directly render to a texture (render to texture OpenGL extension) help to limit the loss in performance caused by slow read-backs for subsequent rendering passes.

As already discussed in Section 9, the limited amount of texture memory in combination with the low bandwidth of the AGP bus is one of the main obstacles for volume rendering on low-cost PC graphics hardware. Whilst the available texture memory might be sufficient for computer games, the currently employed brute-force approaches of rasterizing the complete volume make it necessary to keep the complete data set in the texture memory. Current frame rates are mainly limited by rasterization power and memory bandwidth. Progress in this field is thus dependent on the development of faster memory in the memory market. Besides texture compression as discussed in Section 9, early z-tests or embedded DRAM with wide busses might be future solutions to this problem.

From the application developer's point of view, the variety of different (and highly incompatible) extensions, that give access to the features of the proprietary hardware extensions of graphics hardware manufacturers on a low abstraction layer, make it hard to develop programs, that work on all graphics boards. Shading languages, that raise the ab-

straction layer are currently under development for the future OpenGL2.0 and DirectX9 APIs. A first big step is the Stanford shading language, that is already available for quite a while[26].

12. Summary and Conclusions

This report presented the latest algorithms that utilize flexible consumer graphics hardware for high-quality volume rendering. We have shown a number of new, highly-optimized methods, that permit an interactive visualization of quite large volumetric data sets with yet unseen speed and quality.

Apart from the these algorithms, multi-textures and programmable rasterization units provide a number of additional options, like per-fragment clipping with arbitrary clipping geometry[40], speedup of rendering using parallel rasterization[37] and the rendering of translucent materials[19]. The parallelization of multiple consumer graphics boards to a rendering cluster has been investigated by several authors[11, 24].

We believe, that the trend towards more programmability of the graphics pipeline will carry on. A fully programmable rasterization unit, that facilitates arbitrary per-fragment programs, might even be available in the next generation of consumer graphics hardware. Early z-tests will prevent unnecessary rasterization and memory accesses. Another trend is the development towards floating point precision in the framebuffer and textures. This is particularly useful for multi-pass approaches, that currently suffer from accumulated errors during rendering passes. Instead of merely rendering geometry created from scientific data, programmable graphics hardware will be increasingly used for filtering and mapping tasks in scientific data visualization in the future.

The next generation of consumer graphics hardware is just on the horizon. 3DLabs just announced[1] the P10 Visual Processor Unit (VPU), that will be fully programmable and support the upcoming standards OpenGL 2.0 and DirectX9. NVIDIA's and ATI's next generation GPUs will follow in fall 2002.

Acknowledgements

We would like to thank Joe Kniss, Christof Rezk-Salama, Markus Hadwiger and Martin Kraus for providing images and illustrations for this report.

Web-Links

Semian:
http://www.cs.utah.edu/~jmk/simian/
Pre-Integrated Volume Rendering:
http://wwwvis.informatik.uni-stuttgart.de/~engel/
Interactive Volume Rendering:
http://www9.informatik.uni-erlangen.de/Persons/Rezk/
High-Quality filtering on PC graphics hardware:
http://www.vrvis.at/vis/research/hq-hw-reco/algorithm.html
Siggraph course on PC volume graphics:
http://www.siggraph.org/s2002/conference/courses/crs42.html

References

1. 3DLabs Press Release. http://www.3dlabs.com/whatsnew/pressreleases/pr02/02-05-03-vpu.htm, Sunnyvale, CA - May 3, 2002 23
2. U. Behrens and R. Ratering. Adding Shadows to a Texture-Based Volume Renderer. *In 1998 Volume Visualization Symposium*, pages 39–46, 1998. 12
3. J. F. Blinn. Jim Blinn's Corner: Image Compositing–Theory. newblock *IEEE Computer Graphics and Applications*, **14**(3), 1994. 4
4. J. Blinn, Models of Light Reflection for Computer Synthesized Pictures. *Computer Graphics*, **11**(2):192–198, 1977. 4
5. B. Cabral and N. Cam and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. *ACM Symposium on Vol. Vis.*, 1994 6, 20
6. D. Ebert, F. K. Musgrave, D. Peachey, K. Perlin and S. Worley. Texturing and Modeling: A Procedural Approach. *Academic Press* July, 1998. 21
7. Klaus Engel, Martin Kraus, and Thomas Ertl. High-Quality Pre-Integrated Volume Rendering Using Hardware-Accelerated Pixel Shading. *Proc. of Eurographics/SIGGRAPH Graphics Hardware Workshop 2001*, 2001. 17
8. Ginsburg, D. and Hart, E. and Mitchell, J. ATI_fragment_shader, OpenGL Extension Registry: http://oss.sgi.com/projects/ogl-sample/registry/ATI/fragment_shader.txt. 6
9. R. Grzeszczuk, C. Henn and R. Yagel. Advanced Geometric Techniques for Ray Casting Volumes. *In SIGGRAPH 98 Course Nbr. 4, Orlando, FL*, 1998. 8
10. M. Hadwiger, T. Theußl, H. Hauser, and E. Gröller. Hardware-Accelerated High-Quality Filtering on PC Hardware. *Proc. of Vision, Modeling, and Visualization 2001*, pages 105–112, 2001. 15
11. Greg Humphreys, Ian Buck, Matthew Eldridge and Pat Hanrahan. Distributed Rendering for Scalable Displays. *Proceedings of Supercomputing*, 2000. 23
12. Mark J. Kilgard. NV_register_combiners, OpenGL Extension Registry: http://oss.sgi.com/projects/ogl-sample/registry/NV/register_combiners.txt. 5
13. Mark J. Kilgard. NV_register_combiners2, howpublished = OpenGL Extension Registry: http://oss.sgi.com/projects/ogl-sample/registry/NV/register_combiners2.txt. 5
14. Mark J. Kilgard. NV_texture_shader, OpenGL Extension Registry: http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_shader.txt. 5

15. Mark J. Kilgard. NV_texture_shader2, OpenGL Extension Registry: http://oss.sgi.com/projects/ogl-sample/registry/NV/texture_shader2.txt. 5

16. Gordon Kindlmann and James W. Durkin. Semi-Automatic Generation of Transfer Functions for Direct Volume Rendering. *In IEEE Symposium On Volume Visualization*, 79–86, 1998. 14

17. Joe Kniss, Gordon Kindlmann and Charles Hansen. Interactive Volume Rendering Using Multi-Dimensional Transfer Functions and Direct Manipulation Widgets. *IEEE Visualization 2001* 14

18. Joe Kniss, Gordon Kindlmann and Charles Hansen. Multi-Dimensional Transfer Functions for Interactive Volume Rendering. *Transactions on Visualization and Computer Graphics 2002* 13

19. Joe Kniss, Simon Premoze, Charles Hansen and David Ebert. Interative Translucent Volume Rendering and Procedural Modeling. *IEEE Visualization 2002*, 13, 21, 23

20. Martin Kraus and Thomas Ertl Adaptive Texture Maps *Proc. of Eurographics/SIGGRAPH Graphics Hardware Workshop 2002*, 2002 20

21. E. LaMar, B. Hamann and K. Joy. Multiresolution Techniques for Interactive Texture-based Volume Visualization. *Proc. IEEE Visualization*, 1999. 8, 20

22. Marc Levoy. Display of Surfaces from Volume Data. *IEEE Computer Graphics & Applications*, **8**(5):29–37, 1988. 14

23. Marc Levoy. Efficient ray tracing of volume data. *ACM Transactions on Graphics*, **9**(3):245–261, July 1990. 2, 3

24. M. Magallon, M. Hopf, and T. Ertl. Parallel Volume Rendering using PC Graphics Hardware. *In Pacific Graphics*, 2001. 23

25. S. R. Marschner and R. J. Lobb. An evaluation of reconstruction filters for volume rendering. *In Proceedings of IEEE Visualization '94*, pages 100–107, 1994. 2

26. William R. Mark, Svetoslav Tzvetkov and Pat Hanrahan. A Real-Time Procedural Shading System for Programmable Graphics Hardware. *In Proceedings of SIGGRAPH '01*, pages 159–170,2001. 23

27. Nelson Max Optical Models for Direct Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, **1**(2):99–108, 1995. 3

28. M. Meißner. and S. Guthe and W. Strasser Interactive Lighting Models and Pre-Integration for Volume Rendering on PC Graphics Accelerators. *In Proceedings of Graphics Interface, Conference on Human Computer Interaction and Computer Graphics*, 2002. 19

29. M. Meißner and U. Hoffmann and W. Straßer. Enabling Classification and Shading for 3D Texture Based Volume Rendering Using OpenGL and Extensions *Visualization '99*, 1999. 19

30. M. Meißner, U. Kanus and W. Straßer. VIZARD II, A PCI-Card for Real-Time Volume Rendering. *Proc. Eurographics/Siggraph Workshop on Graphics Hardware*, 61–68, 1998. 1

31. D. P. Mitchell and A. N. Netravali. Reconstruction filters in computer graphics. *In Proceedings of SIGGRAPH '88*, pages 221–228, 1988. 2

32. M. Segal and K. Akeley. The OpenGL Graphics System: A Specification. *http://www.opengl.org*. 1

33. A. V. Oppenheim and R. W. Schafer. *Digital Signal Processing*. Prentice Hall, Englewood Cliffs, 1975. 2

34. H. Pfister and J. Hardenbergh and J. Knittel and H. Lauer and L. Seiler. The VolumePro real-time ray-casting system. *Proc. of SIGGRAPH '99*, 251–260, 1999. 1

35. B.T. Phong. Illumination for Computer Generated Pictures. *Communications of the ACM*, **18**(6):311–317, June 1975. 4, 10

36. T. Porter and T. Duff. Compositing digital images. *ACM Computer Graphics (Proc. of SIGGRAPH '84)*, **18**:253–259, 1984. 4

37. Christof Rezk-Salama, Klaus Engel, Michael Bauer, Günther Greiner, and Thomas Ertl. Interactive Volume Rendering on Standard PC Graphics Hardware Using Multi-Textures and Multi-Stage Rasterization. *Proc. of Eurographics/SIGGRAPH Graphics Hardware Workshop 2000*, 2000. 7, 10, 23

38. S. Röttger, M. Kraus, and T. Ertl. Hardware-Accelerated Volume and Isosurface Rendering Based On Cell-Projection. *Proc. of IEEE Visualization 2000*, pages 109–116, 2000. 16

39. M. Weiler, R. Westermann, C. Hansen, K. Zimmerman and T. Ertl. Level-Of-Detail Volume Rendering via 3D Textures. *Procceedings of IEEE VolVis 2000*, 7–13, 2000. 20

40. D. Weiskopf, K. Engel and T. Ertl Texture-Based Volume Clipping via Fragment Operations *Proc. of IEEE Visualization 2002*, 2002 23

41. R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. *Proc. of SIGGRAPH '98*, 169–178, 1998. 10