Higher Quality Volume Rendering on PC Graphics Hardware

Michael Meißner, Stefan Guthe, Wolfgang Straßer

WSI-2001-12
April 2001

Graphisch-Interaktive Systeme
Wilhelm-Schickard-Institut
Universität Tübingen
D-72076 Tübingen, Germany
e-mail: {meissner, guthe, strasser}@gris.uni-tuebingen.de
WWW: http://www.gris.uni-tuebingen.de

ISSN 0946-3852

Abstract

Shading and classification are among the most powerful and important techniques used in volume rendering. Unfortunately, for hardware accelerated volume rendering based on OpenGL, direct classification is only supported on SGI platforms and shading could previously only be approximated inaccurately, resulting in shading artifacts mostly visible in darkening artifacts. So far, the combination of classification and shading either required multi-pass rendering or two volumetric textures.

This paper presents a novel approach for accurate phong shading using multi-texturing, dependent textures, cube maps, and texture combiners. Furthermore, another novel approach is presented, enabling the interactive change of sample properties such as color, opacity, shading parameters, and gradient magnitude without the need of recomputing the texture every time the classification parameters change requiring no second volumetric texture. Finally, in combination with texture compression, even relatively large volumes can be rendered at interactive frame updates.

CR Categories: I.0.3 [Computer Graphics]: General; I.3.1 [Computer Graphics]: Picture and Image Generation—Graphics processors; I.3.3 [Computer Graphics]: Picture and Image Generation—Viewing algorithms;

Keywords: Volume Rendering, Texture Mapping Hardware, Multi-Texturing, Dependent Textures, Phong Shading, Classification.

Figure 1: Real-time volume rendered images enabling classification and phong shading in single pass rendering on GeForce3. From left to right: Gradient magnitude modulation, zoomed view of aneurism, and differently colored light sources.

1 INTRODUCTION

Due to the large amount of data, computations, and tremendous bandwidth requirements, software approaches are usually limited and far from interactive frame updates. One well known exception might be the ShearWarp algorithm [7], which can achieve interactivity taking advantage of optimizations such as run length encoding (pre-processing). However, each time classification changes a new run length encoding needs to be calculated, and hence, for a fully occupied dataset with semi-transparent classification, no interactivity can be achieved on a desktop machine.

To overcome the inherent large amount of computation and the extreme bandwidth, texture mapping hardware has evolved to become the best known practical volume rendering method for rectilinear grid datasets. Despite of the wide availability, texture mapping based volume rendering has some severe limitations: Classification is a key technique in volume rendering interpreting the volume data as color, opacity, and others. To enable classification in texture mapping based volume rendering, a lookup is needed right after the texture mapping stage. Unfortunately, such a lookup is currently only available on SGI platforms (GL_TEXTURE_COLOR_TABLE_SGI) and it only enables the assignment of color and opacity but no further material properties can be integrated. Shading is yet another key technique to add further visual cues to the rendered images and enables a better interpretation of the images. In contrast to polygon rendering where a normal is a vertex property, a gradient is a voxel property. When using texture mapping for rendering volume data, no gradient estimation is supported in hardware. To circumvent this limitation, one can store the pre-calculated gradient together with the volume data as first proposed by Westermann et al. [13]. Despite of the fact that many improved techniques have been proposed based on this approach, the subsequent shading operations of all of them [13, 8, 10] are based on not normalized interpolated gradients, resulting in shading artifacts and requiring that pre-normalized gradients are stored in the texture which prevents the integration of gradient magnitude modulation.

In this paper, a new approach for integrating accurate and artifact free shading into texture mapping based volume rendering on PC graphics hardware is presented. Furthermore, a new technique accomplishing the integration of classification without the need of re-generating the entire texture nor requiring a second volumetric texture is described. Finally, the combination of classification and shading in a single rendering pass is presented.

1.1 Related Work

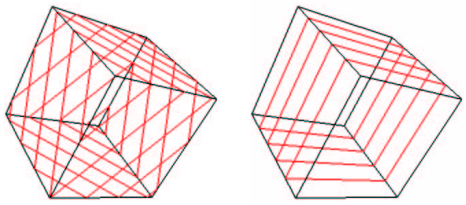
3D texture mapping hardware has been recognized as a very efficient acceleration technique for volume rendering, right after the first SGI RealityEngine [1] has been shipped. Cabral et al. [2] rendered datasets of 256^3 voxels at interactive frame-rates on a four Raster Manager SGI RealityEngine Onyx with a single 150 MHz CPU. Similar results have been presented by Cullip and Neumann [3]. The major drawback of the general texture mapping approach is the absence of shading functionality for volume data. To circumvent this, Van Gelder et al. [6] proposed a 3-4 parameter lookup which is used to classify and shade the data. Unfortunately, no direct hardware support for such a lookup is available. Therefore, each time the viewing or classification changes, an entire new 3D texture needs to be generated. This applies as well for approaches storing a pre-shaded and pre-classified volume into texture memory. Problematic for all these approaches is the individual interpolation of color and opacity which can lead to severe artifacts [14], named color bleeding. This could be circumvented either by pre-multiplying color and opacity[1] — which is necessary whenever the classification changes — or by interpolating data instead of color.

Westermann et al. [13] store density values and corresponding pre-computed and pre-normalized gradients in texture memory and extensively exploit OpenGL and extensions for unshaded volume rendering and shaded iso-surface rendering. Meißner et al. [8] extended this approach combining classification and diffuse shading for semi-transparent rendering of volume data. While both approaches use a matrix multiplication to obtain the diffuse shading intensity, Rezk-Salama et al. [10] use register combiners as available on the nVIDIA GeForce2. Despite of the impressive visual results, all these approaches [13, 8, 10] are based on not normalized interpolated gradients which result in shading artifacts, as explained later in this paper. Similarly to Westermann [13], Dachille proposed to use the available hardware for efficient sample computation and possibly for blending [4]. Shading is performed on the host to ensure high quality rendering, thus avoiding the problem of non normalized gradients. However, interactivity is sacrificed for reasonably sized datasets ($\geq 2MVoxels$) and viewports ($\geq 256^2$), where rendering is in the order of seconds.

The remainder of this paper is organized as follows: Section 2 briefly summarizes the state-of-the-art in texture mapping based volume rendering. An brief introduction to current hardware capabilities is given in Section 3. Our new shading approach, enabling accurate and shading artifact free phong illumination of volume data is presented in Section 4. Thereafter, we describe its combination with simple transfer functions for classification (Section 5). Arbitrary transfer functions in combination with previously reported shading approaches are presented in Section 7. Our results of those techniques in combination with and without texture compression as well as a set of minor but very helpful future extensions for the hardware are presented in Section 9. Finally, we conclude our paper and outline future work.

2 TEXTURE MAPPING REVISITED

The shipment of the first SGI RealityEngine made 3D texture mapping hardware an available interactive feature. With respect to volume rendering, slicing planes parallel to the viewing plane are put through the volume in back to front order, see Figure 3(a). When

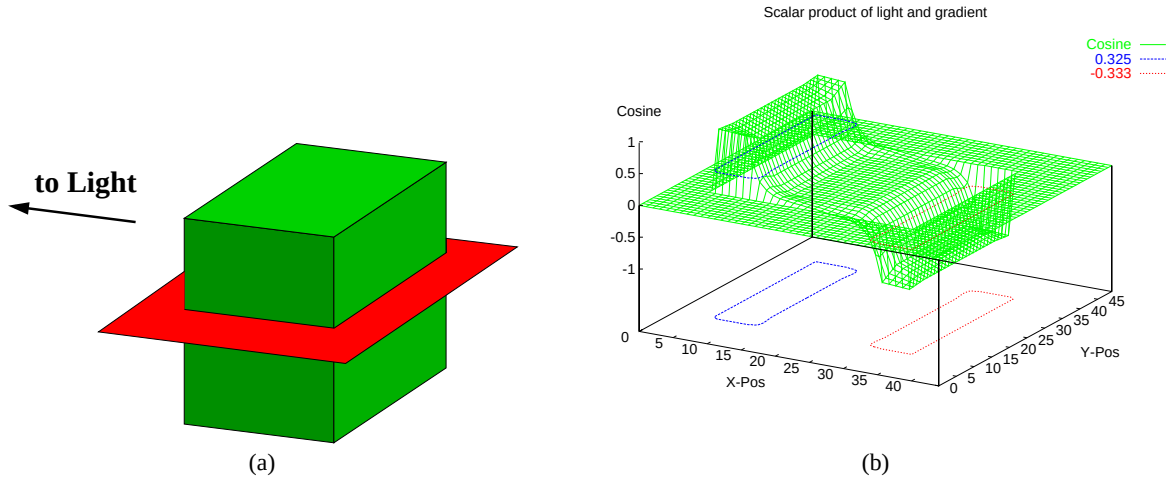


(a) (b)

Figure 3: While in 3D texture mapping (a) arbitrary planes can be positioned in the volume, 2D texture mapping (b) requires a texture stack for each major viewing direction and the one most perpendicular to the actual viewing is selected.

using perspective projection, this becomes more complicated since

[1] Pre-multiplying color and opacity requires high precision datapaths to account for low color and opacity values but current graphics hardware datapaths are fairly low in precision making this a so far impractical approach for semi-transparent rendering.



(a) (b)

Figure 2: Error made in shading a binary cube using not normalized interpolated gradient vectors: (a) Indicates the binary volumetric cube (green) used and the plane of interest (red). (b) shows the error made using not interpolated gradients.

one needs to account for the correct blending. However, opacity values represent the volumetric absorption along a unit length and hence, one would need to use spherical shells or additional textures to correct this[2]. Thus, parallel projection is applied in most cases or artifacts are accepted.

One of the problems involved with 3D texture mapping is its limited availability. It is currently supported in hardware on most mid- and high-end SGI platforms, on HP fx class machines, and on the ATI Radeon. On some other platforms it is available but not supported in hardware, e.g. on the nVIDIA GeForce3. Therefore, an alternative method — derived from the ShearWarp agorithm — has become popular. Here, three stacks of 2D textures are used, one for each major axis (see Figure 3(b)). Depending on the viewing vector, the stack most perpendicular to the viewing direction is used. To account for accurate volumetric absorption, opacity values need to be corrected depending on the viewing angle.

2.1 Classification

Classification can be realized very easily, but is usually not available on all platforms. Using SGI's GL_TEXTURE_COLOR_TABLE_SGI, a texture can be stored as pure density volume interpreting the interpolated density values as a lookup into an at least 256 entries large lookup table. However, this OpenGL extension is only available on mid- and high-end SGI platforms. Using multi-pass rendering, classification can also be accomplished using pixel textures, as presented in [8]. Unfortunately, pixel textures are again limited to mid- and high-end SGI platforms and inherent to the multi-pass approach, the performance is reduced significantly. Finally, using two volumetric textures and multi-texturing hardware, classification can also be accomplished in a single pass [10]. However, this approach requires two volumetric textures significantly increasing the memory requirements even if paletted textures are used. Furthermore, the approach cannot be combined with trilinear interpolation based on two bilinear interpolations and register combiners, as presented in [10]. In summary, classification of interpolated density values is still an unsolved problem for texture mapping hardware based volume rendering[3].

2.2 Shading

As mentioned in the introduction, there has been a number of publications presenting shading of interpolated sample values within the context of texture mapping based volume rendering [13, 8, 10]. All these approaches pre-compute the voxel gradient which is normalized, scaled, and biased in order to obtain gradient values of range $[0, 1]$. The gradient components are then stored in the RGB values of an RGBA texture and the density value goes into the A channel. Using traditional texture mapping hardware, the gradient components and density value are interpolated. However, these approaches directly use the interpolated but not normalized gradients to compute the scalar product, subsequently used for diffuse shading. Thus, these approaches result in severe shading artifacts, mostly noticeable as darkening of the images.

A side by side comparison of software and hardware generated images reveals significant differences. Two causes need to be considered: first the wrong scalar product and second the frequent discretization in the hardware. Figure 2(a) illustrates a binary dataset consisting of a cube as well as the direction to a light source. Figure 2(b) shows the actual error made when using not normalized gradients (error is given by difference of the results of normalized gradients and not normalized gradients). Obviously, the shading artifacts can be quite severe and it needs to be mentioned that this does also occur in non binary datasets because the gradients at grid position need to be pre-normalized which again can introduce big differences of the gradient values of neighboring voxels ($(1, 0, 0)$ and $(1, 1, 0)$ results in a 45 degree difference).

2.3 Further improvements

One further useful improvement of texture mapping based volume rendering is the efficient trilinear interpolation of samples using 2D texture mapping hardware and register combiners [10]. Due to the separability of the linear interpolation kernel, trilinear interpolation can be split into two bilinear interpolation and a final linear interpolation. Using multi-textures of two subsequent 2D texture slices and register combiners to interpolate the two resulting values (vectors of

[2] So far, no approach is known that solves this to an acceptable degree without artifacts.

[3] On PC class machines or in combination with shading.

RGBA), correct trilinear interpolation is accomplished. However, this cannot be combined with classification and the presented shading approach suffers from the artifacts described in Section 2.2.

3 TEXTURE SHADING AND BLENDING

Texture shading and texture blending are relatively new concepts of graphics hardware. The processing of each fragment is split into a texture shading and a texture blending step. While the latter has been around for some time, e.g. register combiners, texture shading is a new and very powerful concept, first introduced on the GeForce3. In the following these concepts are briefly summarized[4] because they are essential for the presented volume rendering approaches.

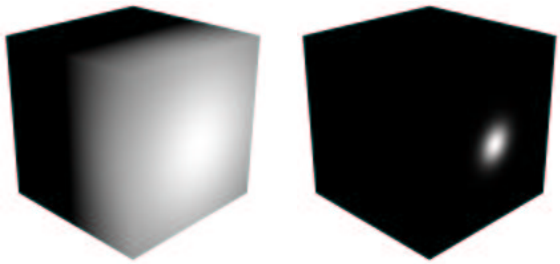
Texture Shader: The GeForce3 contains four texture units that calculate their texture address for fetching the corresponding texture value. In contrast to the standard OpenGL approach of calculating texture addresses, the GeForce3 is capable of using texture results of previous texture units to calculate new texture addresses and access another texture, so-called dependent texturing. The texture address calculation can be influenced by defining a texture shader operation. Generally, these operations can be divided in four groups: the conventional (non-dependent) texture fetches, special case texture fetches, dependent texture fetches and dot product dependent texture fetches. The special case texture fetches do not depend on previous texture units but allow for the removal of fragments from the pipeline (culling). While the dependent texture fetches can easily be used for classification of volume data, the dot product dependent texture fetches are more complex to handle and can put some restrictions on the use of the other texture units. E.g. using a single dot product dependent texture reduces the number of available texture fetches to three because the shader operation of the previous texture unit is used for calculating the second texture coordinate for a 2D texture[5]. Using cube maps, two texture units are needed for calculating the texture coordinates and therefore only two available texture fetches are left. In addition to computing a dot product and accessing a cube map, the texture shader is also capable of addressing a second cube map treating the texture coordinate as a vector and reflecting it using a given normal (used for environment bump mapping). This normal can either be supplied by the fourth component of the texture coordinates or by a user defined constant within the texture shader.

Register Combiner: The resulting RGBA values of each texture unit is passed on to the register combiners which can perform further operations on these fragments. However, once entering the register combiner stage, no further texture mapping functionality or lookup is available. A total of eight register combiners and one final combiner is available[6]. Despite the large set of registers available to each register combiner, only some of them are used within this paper: *col0*: the primary color, *col1*: the secondary color, *spare0*, *spare1*: scratch register, *tex0-tex3*: texture values of texture unit 0-3 and *const0*, *const1*: constant values (unique for each combiner). Generally, each register combiner (general combiner) is split into an RGB and an Alpha portion where each portion has four unique input (A, B, C and D) and three output registers. While the RGB combiner is capable of four calculations: $A \cdot B$ and $C \cdot D$; $A \cdot B$ and CD; AB, CD and $AB + CD$; AB, CD and $\mathrm{mux}(AB, CD)$, where mux returns either AB or CD depending on the alpha value of the *spare0* register. Since the Alpha combiner only uses alpha or blue values for it's calculations, one ends up with only two possible calculations: AB, CD and $AB + CD$; AB, CD and $\mathrm{mux}(AB, CD)$. In addition any unused result can be discarded. While the final alpha combiner can only choose a single alpha or blue value for output, the final RGB combiner is much more powerful. First of all it always calculates the sum *color1+spare0* and the product EF. EF can be inserted into the final equation $AB + (1 - A)C + D$ for A, B, C or D. The sum of *color1* and *spare0* will be clamped to either $[0, 1]$ or $[0, 2]$ and can used as B, C or D. Any other register may also be used as A, B, C or D.

In summary, texture shading and blending offers a sheer amount of combinatorial possibilities which can be used for numerous applications due to its high flexibility [5]. However, programming these features can be quite tedious and sometimes feels like programming microcode.

4 ACCURATE PHONG SHADING

To obtain correct Phong illumination using the interpolated gradient vectors, one would need to normalize the gradient but there is neither a vector normalization unit available in the current OpenGL pipeline nor can a vector be normalized using extensions. Fortunately, there are other approaches to obtain correct shading results without the need of normalizing the gradient. Cube or environment maps consist of six textures, one for each face of the cube. By projecting the diffuse intensity of all surrounding light sources onto the cube faces, these luminance textures can be used for a diffuse cube map, as shown in Figure 4(a). Similarly, this can be performed to



(a) (b)

Figure 4: Cube maps for one light source: (a) Diffuse cube map. (b) Specular cube map using a phong exponent of 50.

generate textures for a specular cube map. Thus, the six textures contain the reflected specular light intensities, as illustrated in Figure 4(b). Instead of using the gradient as index into the cube map, the reflected vector is used [12, 11]. Colored light sources can also be realized using RGB textures for the cube map instead of luminance textures.

In the context of texture shaders, a diffuse and a specular cube map can be realized configuring the texture shaders as illustrated in Figure 5. While texture shader 0 performs the original texture mapping, texture shader 1, 2, and 3 are used to "move" the gradient to texture shader 2 (texture fetch) and to compute the reflected vector in texture shader 3[7]. Depending on the results of texture shader 0, two cube maps are finally accessed by texture shader 2 and 3. Resulting from those texture shader operations, the sample density is available in A_0, the diffuse intensity in $R_2G_2B_2$, and the specular intensity in $R_3G_3B_3$. In a final step, those values need to be combined performing the calculation of the shading model:

$$I = k_a * I_a + k_d * I_d + k_s * I_s \quad (1)$$

[4] For more details, see www.nvidia.com/developer

[5] 1D textures cannot be used for dependent texturing.

[6] Only two register combiners and one final combiner is available in hardware, the others are "emulated" with the same resource.

[7] Texture shaders are not equal in functionality. E.g. the computation of the reflected vector R is hardwired in texture shader 3 and cannot be performed by any of the other texture shaders.

Texture Shader #	Texture Coordinates	Shader Operation	Texture Fetch	Texture Format	Output Color
0	(S,T)	None	texture mapping	2D RGBA	$R_0 G_0 B_0 A_0$
1	([1,0,0], E_x)	$U_x = [1, 0, 0] \cdot [R_0, G_0, B_0]$	None	None	0 0 0 0
2	([0,1,0], E_y)	$U_y = [0, 1, 0] \cdot [R_0, G_0, B_0]$	$U = (R_0, G_0, B_0)$	Cubemap	$R_2 G_2 B_2 A_2$
3	([0,0,1], E_z)	$U_z = [0, 0, 1] \cdot [R_0, G_0, B_0]$ $R = \frac{2U(U \cdot E)}{(U \cdot U)} - E$	$R = (R_x, R_y, R_z)$	Cubemap	$R_3 G_3 B_3 A_3$

Figure 5: Implementation of phong shading using the four texture shaders available on the GeForce3. Texture shader 1,2, and 3 "move" the Gradient from $R_0G_0B_0$ to U using the texture coordinates $(1, 0, 0, E_x)$, $(0, 1, 0, E_y)$, and $(0, 0, 1, E_z)$. Furthermore, the reflected vector R is computed in texture shader 3. While U is used to access a diffuse cube map, the reflected vector is used to access a specular cube map.

which can be performed very easily requiring one register combiner ($k_d * I_d$ and $k_s * I_s$) and the final combiner, as denoted in pseudo combiner code[8]:

```
// combiner 0
rgb { col1   = tex2*col0;     // diffuse
      spare0 = tex3*const1;} // specular
alpha {}

// finale combiner
sum     = col1 + spare0;      // diff+spec
out.rgb = sum+const0;         // sum
out.a   = tex0;               // opacity
```

where tex$i$ denotes the output of texture shader i and rgb and $alpha$ denote the operations performed on the RGB values and A values respectively. col0 is set to the diffuse material property k_d, const0 is set to $k_a * I_a$, const1 is set to be the specular material property k_s, sum is a temporary result of an ADD of the final combiner, and out denotes the final RGBA values entering the per fragment pipeline.

In summary, with the availability of hardware supported cube maps, it is possible for the first time to accomplish true phong shaded volume rendered images based on texture mapping hardware.

5 SIMPLE TRANSFER FUNCTIONS

When using the four available texture shaders as described in Section 4, no further dependent texture can be accessed to perform the actual classification dependent on the interpolated density value stored in A_0 (see Figure 5). Basically texture shader 1 is not performing any texture operation in this configuration but when using two cube maps, texture shader 1 cannot be used for any texture operation at all.

Alternatively, one can use register combiners to perform classification. In this case, only simple stairs with up to four intervals or a linear ramp can be realized, as illustrated in Figure 1. The basic concept is to multiplex constants depending on given interval boundaries. E.g. assuming that all voxels of values $\leq x$ should be fully transparent and all others should be semi-transparent white, this can be accomplished multiplexing the opacity values 0 and 0.5

[8]There is an extension which translates combiner code into OpenGL commands but for illustration purposes, this combiner language has been simplified.

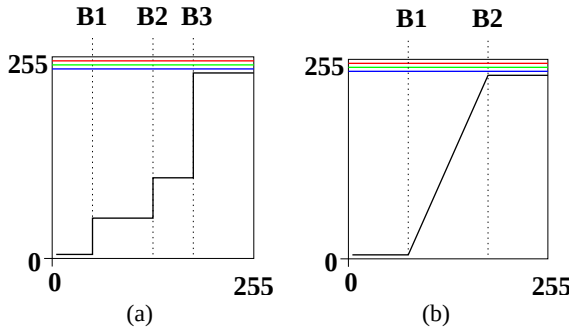


Table 1: Simple transfer functions that can be realized using up to three interval boundaries (B1, B2, B3) and register combiners: (a) Intervals with const RGBA for each interval. (b) Three intervals with two intervals of const RGBA and the middle interval as linear ramp.

by comparing the sample value (A_0) with the interval border x. The corresponding configuration of three register combiners and the final combiner are (including the shading computations of the previous section):

```
// combiner 0
rgb   { col1    = tex2*col0;}  // diffuse
alhpa { // 0.5+0.5(A0-x);
        spare0  = tex0*half_bias_negate(0);
                + unsigned_invert(const0)
                   *half_bias_negate(0);

// combiner 1     // (A0<=x ? const0:const1)
                  // sparse0 triggers mux
rgb   { tex1 = mux(const0, const1) }
alpha { tex1 = mux(const0, const1) }

// combiner 2
rgb { spare0  = tex3*const1; // specular
      col1    = tex1*col1;}  // diff*Id
alpha {}

product = tex1*const0;   // RGBA(A0)
sum     = col1 + spare0; // diff+spec
```

```
out.rgb = sum+product;   // shading
out.a   = tex1;          // A(A0)
```

where const0 contains the color and opacity used if the sample value is $\leq x$ and const1 contains the color for voxel values larger than x.

The concept of multiplexing values for one interval boundary enabling two different classification states can be extended to up to four intervals. Any further interval requires one more multiplexing stage and thus two more register combiners. The corresponding pseudo combiner code for the involved combiners is basically cascading the above described concept. Alternatively to up to four stair intervals, three intervals can be used where in the middle interval a linear ramp is realized. This is e.g. very useful in visualizing CTA aneurysms. Most volumes can either be classified using one of these two simple classification schemes: while most datasets synthetic datasets require a classification of the first kind, medical datasets usually require the latter type of classification using a ramp since the boundaries between different kinds of tissue are somewhat fuzzy.

In summary, Figure 8 (d-f) show images of the engine block dataset were three different voxel value intervals are classified. The lower 25% is mapped to full transparency, the upper 25% to red and full opacity, and the range in between is classified semi-transparent white.

6 GRADIENT MAGNITUDE MODULATION

Using the gradient magnitude to suppress data which resides within homogeneous areas of a dataset is a very powerful feature for enhancing boundaries. Generally, when applying gradient magnitude modulation, the quality of the boundary enhancement depends mainly on the quality of the used gradient filter. While the intermediate and central difference gradient filters are prone to artifacts — since they result in non symmetric gradients —, the Sobel operator is the gradient operator of choice and used throughout this paper.

Figure 8(e) and (f) show images using gradient magnitude modulation compared to not using the gradient magnitude (Figure 8(d)). Generally, gradient magnitude modulation modifies the opacity of a sample based on the magnitude of the sample's gradient. For this purpose, either a gradient magnitude transfer function can be used or any power of the gradient length can be computed. While the first offers more flexibility but requires an additional lookup, the latter can be computed on the fly without lookup.

When performing accurate shading based on cube maps, there is no spare dependent texture lookup. Thus, gradient magnitude modulation modifying the sample α by:

$$\alpha = \alpha * pow(\text{length}(\text{gradient}), n) \qquad (2)$$

is implemented using register combiners. It allows to chose n to be either 2, 4, or eight and can freely be combined with the classification technique described in the previous section. Figure 8(e) was generated using $n = 2$ and $n = 4$ was used for Figure 8(d).

7 COMPLEX TRANSFER FUNCTIONS

Depending on the volumetric data to be visualized, there are cases where simple classification as presented in Section 5 does not suffice, e.g. one needs to use per sample color and/or material properties. In this case, simple dependent texturing can be used to provide this data. For simple RGBA transfer functions, a single dependent texture can be used. For further material properties (k_a, k_d,k_s), a second texture can be used, as illustrated in Figure 6. Even though there are no 1D textures possible in dependent texturing, these 2D textures can be of size 2×256. The drawback of such a per sample classification is that it cannot be combined with accurate Phong shading due to the limited resources (four texture shaders). Thus, in cases where arbitrary per sample classification in combination with shading is mandatory, one could combine this classification approach with a less accurate shading technique, as presented in [10]. However, the image quality would be significantly lower than with the presented accurate Phong shading. For most cases, simple transfer functions are sufficient as all images in Figure 1 and Figure 8 were generated using them.

#	Tex Coord	Shader Op	Texture Fetch	Texture Format	Output Color
0	(S,T)	None	texture mapping	2D RGBA	$R_0 G_0 B_0 A_0$
1	ignored	None	(A_0,R_0)	2D RGBA	$R_1 G_1 B_1 A_1$
2	ignored	None	(A_0,R_0)	2D RGBA	$R_2 G_2 B_2 A_2$
3	ignored	None	None	None	0 0 0 0

Figure 6: Implementation of complex classification using texture shader. While texture unit 1 is used as a lookup table for the color and opacity of each interpolated voxel, texture unit 2 is used to store further material properties (k_s, k_d, and k_a).

8 TEXTURE COMPRESSION

One of the main drawbacks when using texture mapping hardware for volume rendering is the need for storing an RGBA texture in order to provide the gradients. This is necessary because there is no support for extracting gradients directly from the density volume, as done in VolumePro [9]. Thus, a significant amount of texture memory is required to store the additional gradient information. For 8 bit voxel values and a 256^3 volume, the memory requirements are increased from 16 MBytes to 64 MBytes. Thus for a graphics card with 64 MBytes of memory (texture and framebuffer memory), and a volume that is much larger than the available texture memory, the volume needs to be partitioned into bricks which are transfered from main memory to the graphics card when needed. However, even with an AGP bus, this significantly reduces the the overall performance and real-time frame-rate are not anymore feasible.

Recently, the ARB[9] of OpenGL released an extension for texture compressions, ARB_texture_compression which is supported on many PC graphics cards (Voodo5, Radeon, GeForce family). The compression is based on the st3c algorithm and accomplishes a constant compression rate of four by packing 4×4 texels into a compact bitstream. Thus, datasets which are much larger than the available texture memory of the graphics card can still be rendered at real-time or interactive frame-rates. However, image quality is potentially sacrifized due to the lossy compression algorithm. Figure 7 illustrates the difference in image quality for a full (a,b) and a close-up view (c,d) of the engine dataset. While the global information and structure is still available, fine detail is lost. Thus, one might want to implement a hybrid renderer, bricking the volume into subcubes (e.g. 32^3) which allow view frustum culling. The subcubes close to the observer or within the ROI[10], uncompressed textures could be used while the others are rendered using compressed textures.

[9] Architecture Review Board.

[10] Region of interest.

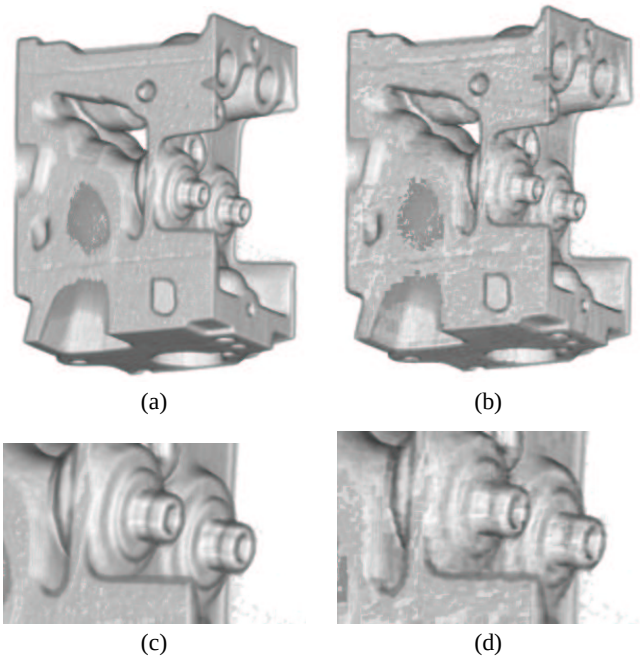


Figure 7: Texture compression applied to the engine dataset: (a,c) Without compression. (b,d) With compression.

9 RESULTS

With the described techniques, high quality images using accurate phong shading, gradient magnitude modulation, and classification can be accomplished. The results with respect to image quality are summarized in Figure 8, depicting a set of images generated on the GeForce3. Besides the image quality, the overall performance is usually of interest. Since the presented techniques do not use multi-pass rendering, interactive to real-time performance is accomplished for all presented datasets and classifications, using a viewport of 500×500 pixels[11].

The necessary resolution of the textures of the specular cube map depends on the chosen phong exponent. Phong exponents of up to 128 can be represented in textures of 64^2 texels, without noticeable degradation of the image quality. The resolution of the diffuse cube map textures can be chosen much lower (16^2).

As mentioned earlier, sharing resources occurs when using more than two register combiners. Generally, one can use up to eight general register combiners but resources for only two are available in hardware. Nevertheless, no performance reduction could be measured for any of our techniques.

10 CONCLUSIONS

In this paper, we presented a novel approach for accomplishing true phong shaded volume rendered images using cube maps, dependent textures, and multi-stage rasterization. Additionally, the combination of this approach with gradient magnitude modulation and on the fly classification of volume data using simple transfer functions such as stairs or linear ramps was described. There over, in combination with a less sophisticated shading approach, the integration of arbitrary transfer functions enabling RGBA and material properties as a per sample property was presented. Thus, unprecedented high quality volume rendered images based on texture mapping hardware were accomplished.

The presented results were generated on a nVIDIA GeForce3 using OpenGL. Besides its high throughput, the GeForce3 offers highest possible flexibility within the texturing and the rasterization stage. As demonstrated, this flexibility can be efficiently exploited to enable and combine the most important and most valuable techniques of volume rendering at interactive frame rates.

In the future, we hope to see a continuous increase in the flexibility and programmability of upcoming graphics hardware. With respect to volume rendering, the urgent issue is hardware support of 3D texture mapping in combination with functionality as texture shaders, dependent textures, and register combiners.

[11] Ideal performance could not yet be reached due to the pre-production board and the pre-release status of the OpenGL drivers. Nevertheless, for datasets of 256^3 voxels interactive to real-time frame-rates were accomplished.

11 ACKNOWLEDGEMENTS

The authors would like to thank David Kirk, Matthew Papakipos, and John Spitzer from nVIDIA for providing a GeForce3 and early OpenGL drivers. This work has been funded by the SFB grant 382 of the German Research Council (DFG).

References

[1] K. Akeley. RealityEngine Graphics. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 109–116, August 1993.

[2] B. Cabral, N. Cam, and J. Foran. Accelerated Volume Rendering and Tomographic Reconstruction Using Texture Mapping Hardware. In *Workshop on Volume Visualization*, pages 91–98, Washington, DC, USA, October 1994.

[3] T. J. Cullip and U. Neumann. Accelerating Volume Reconstruction with 3D Texture Mapping Hardware. Technical Report TR93-027, Department of Computer Science at the University of North Carolina, Chapel Hill, 1993.

[4] F. Dachille, K. Kreeger, B. Chen, I. Bitter, and A. Kaufman. High-Quality Volume Rendering Using Texture Mapping Hardware. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 69–76, Lisboa, Portugal, August 1998.

[5] S. Dominé and J. Spitzer. OpenGL Texture Shaders. *Technical document, available from http://www.nvidia.com/*, 2001.

[6] A. Van Gelder and K. Kim. Direct Volume Rendering With Shading via Three-Dimensional Textures. In *Symposium on Volume Visualization*, pages 23–30, San Francisco, CA, USA, October 1996.

[7] P. Lacroute and M. Levoy. Fast Volume Rendering Using a Shear-Warp factorization of the Viewing Transform. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 451–457, July 1994.

[8] M. Meißner, U. Hoffmann, and W. Straßer. Enabling Classification and Shading for 3D Texture Mapping based Volume Rendering using OpenGL and Extensions. In *Proc. of IEEE Visualization*, pages 207–214, San Franisco, CA, USA, October 1999. IEEE Computer Society Press.

[9] H. Pfister, J. Hardenbergh, J. Knittel, H. Lauer, and L. Seiler. The volumepro real-time ray-casting system. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 251–260, Los Angeles, CA, USA, 1999.

[10] C. Rezk-Salama, K. Engel, M. Bauer, G. Greiner, and T. Ertl. Interactive volume rendering on standard pc graphics hardware using multi-texturing and multi-stage rasterization. In *Proc. of Eurographics/SIGGRAPH Workshop on Graphics Hardware*, pages 109–118, Interlaken, Switzerland, August 2000.

[11] J. Terwisscha van Scheltinga, J. Smit, and M. Bosma. Design of an on Chip Reflectance Map. In *Proc. of the 10th EG Workshop on Graphics Hardware*, pages 51–55, Maastricht, The Netherlands, August 1995.

[12] D. Voorhies and J. Foran. State of the art in data visualization. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 163–166, July 1994.

[13] R. Westermann and T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Computer Graphics*, Proc. of ACM SIGGRAPH, pages 169–177, Orlando, FL, USA, August 1998.

[14] C. M. Wittenbrink, T. Malzbender, and M. E. Goss. Opacity-Weighted Color Interpolation For Volume Sampling. In *Symposium on Volume Visualization*, pages 135–142, Research Triangle Park, NC, USA, October 1998.

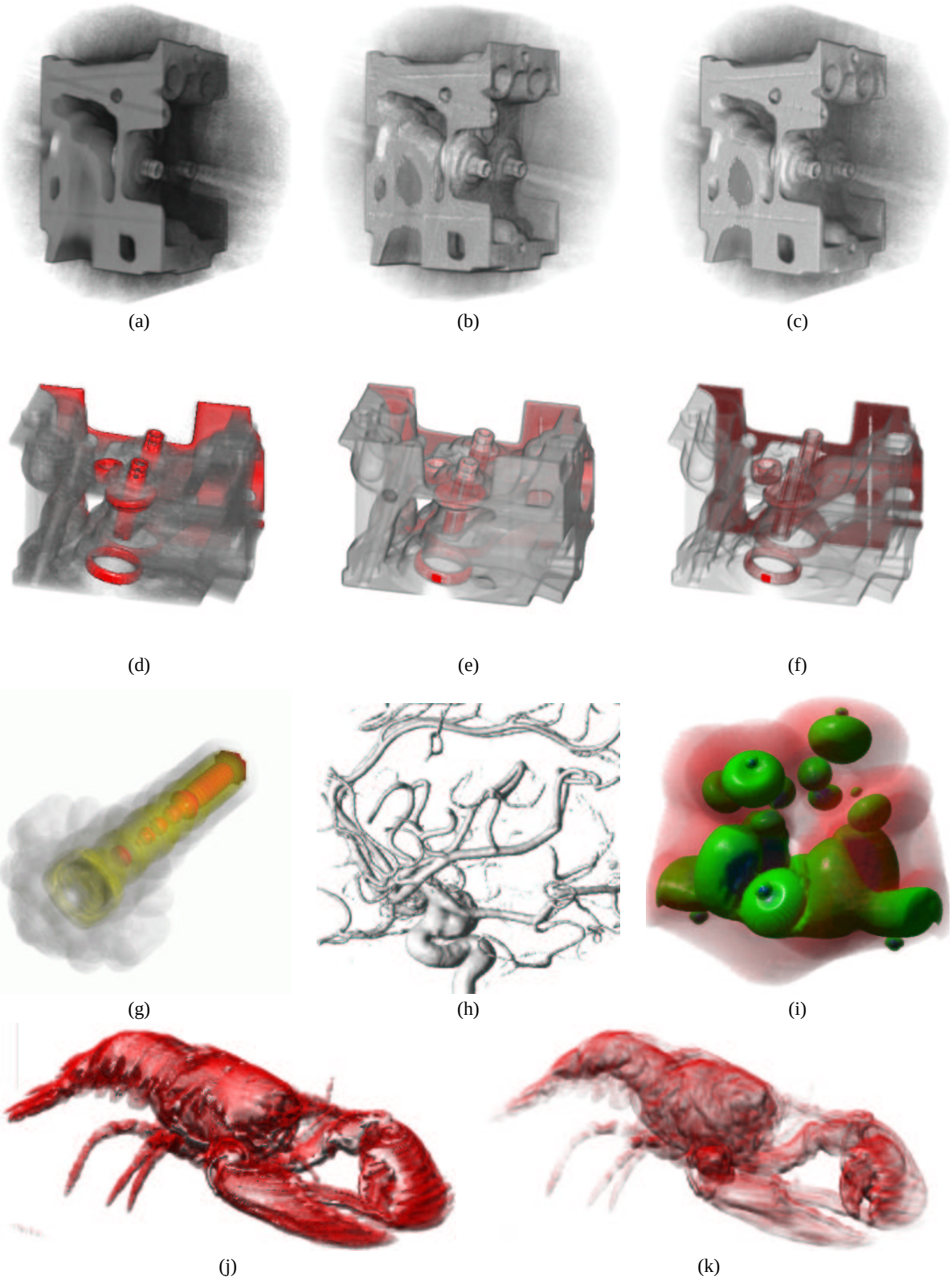


Figure 8: Color plates: Engine dataset $256^2 \times 110$, fuel injection 64^3, aneurism 256^3, neghip 128^3, and lobster $324 \times 301 \times 57$. (a) No illumination. (b) Wrong illumination using not normalized interpolated gradients [13, 8, 10]. (c) Accurate Phong illumination. (d) Illumination and classification. (e) as (d) using linear gradient magnitude modulation. (f) as (d) using squared gradient magnitude modulation. (g) Illumination and classification. (h) Illumination and classification using a linear ramp. (i) Illumination and classification. (j) Illumination and classification. (k) as (j) using squared gradient magnitude modulation.