Controlling GPU-based Volume Rendering using Ray Textures

Matthias Raspe
Institute for Computational Visualistics,
University of Koblenz-Landau
Universitätsstraße 1,
56070, Koblenz, Germany
mraspe@uni-koblenz.de

Stefan Müller
Institute for Computational Visualistics,
University of Koblenz-Landau
Universitätsstraße 1,
56070, Koblenz, Germany
stefanm@uni-koblenz.de

ABSTRACT

In this paper we introduce a novel approach to control different rendering parameters of volume ray casting. Since the introduction of ray casting implementations on programmable graphics hardware, both performance and flexibility have increased and are able to outperform texture-based techniques. In addition, by using rays for computing the volume integral instead of proxy geometry one has more control over local settings. Therefore, we employ dependent texture lookups to user editable 2D textures, thus allowing for interactive parameter setting on a per-ray basis, at a negligible performance overhead on modern graphics hardware. By those means we are able to control the volume rendering in a way not possible with proxy-based direct volume rendering and demonstrate some exemplary uses.

Keywords:
Volume rendering, GPU raycasting, Texturing

1 INTRODUCTION

Direct volume rendering (DVR) plays an important role in visualizing three-dimensional data, with datasets from modern image acquisition systems in medicine being the most prominent. Several methods for solving this computationally demanding problem have been proposed in the past decades and form the foundations of many volume rendering systems in a variety of applications. Engel et al. [EHK$^+$06] provide a detailed overview of basic and advanced volume rendering concepts for real time applications.
Among the rendering methods, volume ray casting and texture-slicing are the approaches that are mainly used today. While the former performs the volume integration directly by sampling the data volume, the latter uses proxy geometry to exploit accelerated texturing capabilities of graphics hardware. Ray casting has been proposed long before programmable graphics hardware (GPU) has been widely available, with Levoy's work being one of the first [Lev90]. Therefore, it has been first realized on the CPU only, whereas the texture slicing approach uses the graphics hardware. In addition, the quality of ray casting is usually higher, making it the de-facto standard for modern medical visualizations.
With the fast advances of graphics hardware technology, GPU-only implementations of the ray casting algorithm have become feasible [KW03] and have achieved a lot of research since then. The key idea behind ray casting is to generate rays starting at the boundary of the volume data and sample the data along each ray at certain intervals until the ray reaches the opposing boundary (or some other criterion is met). For GPU-implementations, the rays are determined by rendering the front and back faces, respectively, of some bounding geometry. Integration along the ray is then performed by multi-pass or – starting with Shader Model 3.0 hardware – single-pass implementations using loops in shader programs.

In our work, we propose the control of rendering parameters at ray level by employing intermediate textures used as parameter lookup table. Being a ray casting approach, it is of course not limited to pure GPU implementations. However, to achieve reasonably interactive performance on commodity hardware we focus on a GPU-only implementation.
As outlined before, the rays are generated by rendering front- and back-facing bounding geometry, respectively. During rasterization the transformed positions of the vertices are interpolated and stored as RGB colors, resembling the rays' directions. Usually, other computations as gradient estimation, transfer function application, etc. are performed during this pass. In addition to the step size as the basic internal parameter for ray casting, these computations typically introduce differ-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

WSCG'2008, February 4 – February 7, 2008
Plzen, Czech Republic.
Copyright UNION Agency – Science Press

ent parameters. Controlling them individually across the volume domain as proposed in this work is usually not possible, but would provide different possibilities: starting the rays at different intervals for local cutaway views, decreasing the step size (thus better quality) only in areas of interest or importance, weighting optical properties during integration, to name a few.

The remainder of the paper is structured as follows: In the next section, we will discuss some existing techniques and approaches in the context of our method. Then, we will present our approach by briefly outlining the environment used and deriving a classification of relevant parameters. Also, we will present details on using the additional, volume-domain texture for controlling the parameters, as well as different user interaction strategies. The rendering performance and results are then presented in section 4. The paper concludes with a discussion of the results and further ideas and improvements left as future work.

2 RELATED WORK

In order to visualize volumetric data, several approaches have been proposed. Especially ray casting provides superior quality, but comes at a high computational cost. The first implementations on programmable graphics hardware in 2003 by Krüger et al. [KW03] and Röttger et al. [RGW$^+$03] have introduced such visualizations to single commodity PCs, particularly in combination with optimizations such as early ray termination, empty space skipping, etc. To allow for a better rendering performance, while maintaining or even improving the visual quality, the work by Scharsach et al. [Sch05] introduced several advanced concepts for GPU ray casting.

Another advantage of ray casting is the easy integration with geometry, as is particular interesting for clipping techniques or convincing volumetric effects in computer games. Although integrating non-volume data into GPU-based ray casting systems needs some special handling during ray generation, Kratz et al. [KSFB06] have established a flexible solution using the depth buffer. Related to ray–geometry intersection is the correct rendering while moving the camera within the volume. Therefore, the bounding geometry generating the rays must not be clipped by the view-frustum (i.e., near plane), but reset to resemble the "new" starting point of rays; details can be found in [SHN$^+$05].

The concept of controlling the ray itself has been further extended by Rezk-Salama et al. [RSK06] to allow for a flexible exploration of the volume. Especially in medical datasets, inner structures often cannot be revealed by editing the transfer function only, due to the viewpoint dependency of occlusion. Their approach aligns well with GPU-based ray casting and also exploits other hardware features to achieve interactive performance, but does not work on individual rays or their properties. This idea has been proposed by Malik et al. [MMG07] whose method evaluates the profile of each ray through the volume and thus allows for a more flexible peeling technique.

Another category of research aims at controlling the rendering of the volume by defining clipping data, usually consisting of basic geometry like planes, spheres etc. This quite simple, yet effective technique is integrated in almost every commercial system and provides basic interaction functionality. In addition, this can also be extended to volume data (e.g., segmentation results) specifying the rendering/clipping of individual voxels. Although Weiskopf et al. [WEE03] have presented this approach originally for texture-based systems, using their concepts in a ray casting environment is straightforward.

3 OUR APPROACH

In this section we will describe our implementation by presenting the approach of controlling individual rays for GPU-based ray casting. Therefore, we will first outline the environment and some related features of our framework. In order to allow the control of different types of parameters, we will derive and classify parameters according to their properties. Subsequently, details on the ray textures and shader implementation are provided.

3.1 Concepts

Programming environment The concepts of ray textures have been implemented using our GPU-based system "Cascada". This cross-platform framework focuses on processing (medical) volume data by applying modular algorithms running solely on modern commodity graphics hardware. Using the GPU for such computations is motivated by the rapid performance increases as shown in the overview by Owens et al. [OLG$^+$07] or, in a more specific context, in Langs et al. [LB07].

In order to abstract from the graphics programming details, algorithms are represented hierarchically: so-called *sequences* encapsulate procedures that can range from simple thresholding to more complex operations like region growing. Sequences in turn consist of multiple *passes*, i.e., drawing geometry with assigned shader programs, usually to offscreen buffers for advanced processing. Finally, *shader programs* resemble objects containing GLSL vertex, geometry, and fragment programs, together with an automatic infrastructure for handling uniform parameters on both the CPU and GPU efficiently, concatenation of shaders, etc. For even more flexibility, the system uses different design patterns from object-oriented programming to implement composite structures, amongst others.

The data itself is represented as volumes packed into RGBA-tuples, thus allowing for direct rendering into the volume and exploiting the SIMD-like data types of GPUs. For both backward compatibility and better performance, the system uses a two-dimensional variant of the volumetric data as introduced as "flat 3D textures" by Harris et al. [HBSL03]. Our system also provides CPU equivalents of the aforementioned sequences and is therefore able to transfer data between graphics and main memory interchangeably. To this end, Cascada uses a "lazy evaluation" policy to avoid unnecessary bus communication.

Classification of parameters As mentioned before, we are looking for means to control volume rendering parameters down to the ray level. Of course, this implies that we can also build groups of rays for equal properties, thus simplifying user interaction. Thereby, the possible levels of control range from a single ray up to all rays at once, i.e., standard ray casting.
Apart from the level of control, we need to look at the parameters' type and semantics. One basic parameter in ray casting is the step size specifying the interval at which the volume is sampled along the ray: the larger the step size, the coarser the sampling, and vice versa. Another set of parameters closely related to the ray itself is the offset specifying the valid interval of sample positions along the ray. Figure 1 illustrates those *geometric parameters*, with the offset parameters defining only two intervals for clarity in the example.

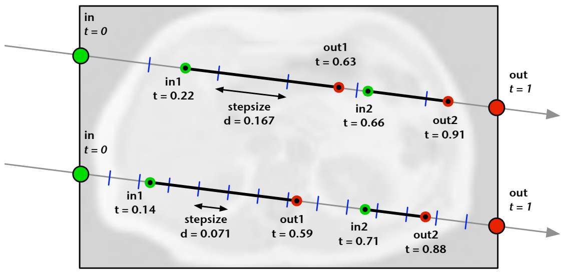

Figure 1: Controlling geometric parameters per ray: step size (blue dashes) and two intervals ("in*N*/out*N*") are set individually, thus sampling the volume along different intervals (thick black lines)

In contrast to the ray geometry, another class of parameters can be defined as *value parameters* that control the computations during integration (i.e., within the ray casting loop). The following list names a few typical parameters with increasing complexity:

- threshold for early-ray termination (ERT)
- weighting directional properties, e.g. gradients
- combining optical properties by blending different rendering modes (MIP, transfer functions, etc.)

As well as the geometric parameters described before, the uses are not limited to the examples listed here. In addition, both sets of parameters can be combined to increase, for example, the quality of early terminated rays by reducing the step size for low ERT thresholds.

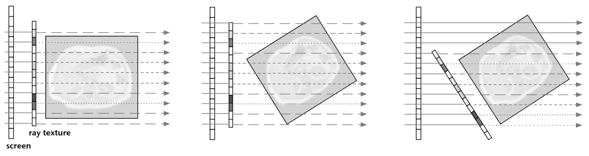

Figure 2: Different rendering modes: initial view (left), view aligned (middle) and volume aligned (right) application of ray textures to transformed volume.

Controlling the texture In order to edit the ray texture during run time, there are two different approaches, no matter if the texture is set up automatically or via direct user input. As depicted in figure 2, the texture can be aligned with the screen or with the volume. For the former, the data in the texture is set in viewport coordinates, i.e., keeping the texels at fixed positions while changing the volume rendering in terms of rotation, position, etc. We will call this method *view aligned*, also to emphasize the analogy to view aligned slices in proxy-based volume rendering. The second method maps the additional texture to the bounding geometry of the volume. This way the ray texture is transformed together with the volume and represents a *volume aligned* texture mode. Note the difference in the sampling of the volume, indicated by the different dashes in the example, with the ray texture specifying the step size.

Shader handling Our approach focuses on a GPU-only implementation of ray casting which means that the computations are performed by shader programs. These programs are compiled once and loaded for drawing the geometry. Thus, the functionality of a shader cannot be changed without loading a recompiled program. To minimize the overhead of providing several complete shader programs we will utilize two different strategies. First, our system supports the concatenation of shader code fragments to allow building complex shaders from small components, especially as GLSL does not support include directives. Using this approach, we can use different modules integrating the parameter types introduced before into a default ray casting fragment program. Depending on the user's selection the corresponding shader is assembled and loaded.
The second method does not employ the assembly of shader programs from small components, but uses one complete shader program. The different "semantics" (i.e., controlling the step size, ERT threshold, etc.) are used directly in the code by accessing different textures (or channels thereof) that have been initialized accordingly. In section 3.2 we will show some example code and discuss the different approaches.

3.2 Implementation

In the preceding section, several use cases for the concept of ray textures have been outlined. From that, the different levels of complexity of the parameters should have become clear, thus requiring appropriate controls for the user. We will first describe the implementation details and discuss different approaches for mapping the user input to the parameters.

Setup In our framework, we already have some basic tools for working with and rendering volume data. In addition to the data itself represented as volume and array objects, respectively, the system also provides the corresponding texture objects for wrapping OpenGL states, handles, etc. Also mentioned before, we utilize "flat 3D textures" that unfold volumetric data into a large 2D texture, also to allow for fast rendering into volume data. Although this introduces some additional code for address translation it still outperforms 3D textures on current hardware for basic interpolation (for details see Langs et al. [LB07]).

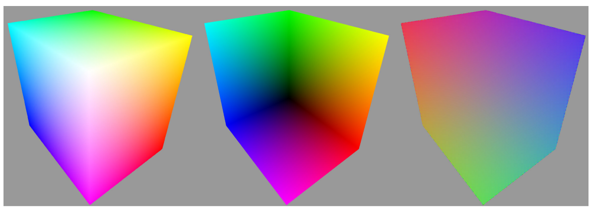
Figure 3: Color-coded textures to generate rays: starting positions (left), end positions (middle) and resulting normalized vectors (right).

The ray casting itself is implemented as a typical two-pass algorithm: the first pass renders the front faces of the bounding geometry (here: cube scaled to volume proportions) into an offscreen buffer, converting the fragments' positions into RGB colors. Note that our approach is also applicable to optimized bounding geometry as proposed by Scharsach et al. [Sch05], because only the rays' parameters are controlled, independent of how the rays are computed. These starting positions are given in volume coordinates, resulting in the typical color cube (fig. 3). The second pass performs the remaining computations by rendering the back faces, sampling the volume along the calculated rays and accumulating the values, do various calculations, etc. This shader program is the main part of the ray casting algorithm and will be augmented with the additional texture for controlling the individual rays. In addition to extending the shader code as shown in figure 4, we also prepare a 2D texture initialized as needed for the further steps.

Application As described in section 3.1, both controlling options have the advantage of providing a fast and simple setup and manipulation on the application side, and being easily accessed during the ray casting process in the shader program. The two approaches differ only slightly in terms of shader implementation, so that the whole procedure can be summarized as follows:

1. initialize 2D texture in size of the viewport
2. manipulate the texels according to user input from window coordinates (optional)
3. update the texture and load it to the GPU
4. during ray casting (i.e., in the second render pass):
   (a) view aligned: access the ray texture using the window relative coordinates
   (b) volume aligned: access the ray texture using the starting position of the ray
5. control the ray parameters within the shader (e.g., within the loop)

The first step should be clear and does not need further explanation. If the ray texture should be initialized with pre-computed results, the second step is optional. In case the step is needed, we have implemented a circular neighborhood of the current pixel position, with varying size and fall-off. Taking the basic idea of ray textures to controlling rays from a single ray to all rays at once one step further, we have also implemented a hierarchical approach. Therefore, editing the ray texture can be performed in different resolution levels: using a coarser level will result in many rays being changed at once, and vice versa. However, this permits only square areas to be edited due to the very nature of texture mip-mapping. Together with the "manual" approach, this allows for further customization of the editing area, with results being presented and discussed in section 4.2. The following step transfers the changes to the graphics hardware, so that this has to be done per frame. The subsequent steps are performed within the shader and will be described in the next section in detail.

Shader implementations As mentioned before, accessing the texture for the two methods is done in the second pass of the ray casting algorithm. For the *view aligned* mode, the ray texture is simply indexed using the relative window coordinate of the current fragment via GLSL's `gl_FragCoord`. When using *volume aligned* access the same texture coordinates as for the color-coded ray positions from the preceding pass (i.e., the fragments from rendering the bounding geometry) are used for fetching the corresponding texel from the ray texture. This is shown in the example code in figure 4 for editing the step size. Note how the current ray texture value can be used differently by using texture channels, sets of ray textures, etc.

```
uniform sampler3D volTex; // volume data
uniform sampler2D rayTex; // control texture
uniform sampler2D startTex; // ray start pos.
uniform vec2 rcpWinSize; // reciprocal win size
varying vec3 texcoord; // ray stop positions
uniform float stepsize; // default: 1/256.0
uniform bool volAligned; // mode (default: true)

void main()
{
  // compute ray
  vec2 tc = gl_FragCoord.xy * rcpWinSize;
  vec3 raystart = texture2D(startTex, tc).xyz;
  vec3 ray = raystart - texcoord;

  vec4 control;
  if ( volAligned )
    control = texture2D(rayTex, raystart.xy);
  else
    control = texture2D(rayTex, tc);

  // ...

  // set step size from first channel of rayTex
  stepsize = max(0.00390625, control.x);

  // integrate
  for (float t = 0.0; t <= ray_len; t += stepsize)
  {
   vec3 pos = texcoord + t * ray;
   vec4 sample = texture3D(volTex, pos);

   // ...
  }
  // weight result from ray texture
  gl_FragColor = finalcolor * control.y;
}
```

Figure 4: GLSL shader code (simplified) showing the use of ray texture within a standard ray casting shader.

Implementing the functionality itself is also straightforward, as will be shown with some examples that address both classes of parameters (see section 3.1). For the first example, we show the effect of decreasing the step size in regions of interest to improve rendering with an initially low step size ($1/10$ instead of $1/256$). Therefore, the user draws into the texture to lower the value stored in one of the channels. This value is then read in the shader to set the step size which is used as increment of the inner loop. Figure 5 shows a vessel data set ($384^2 \times 72$, 16 bit) with an accordingly edited ray texture to increase rendering accuracy in regions of interest.

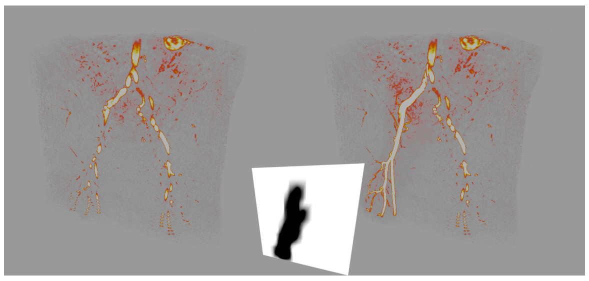
Figure 5: Decreasing the step size for a region of interest (left part of vessel) in an example dataset. The inset depicts the corresponding ray texture.

The second example (figure 6) blends two shading modes by using a value parameter for linear interpolation. Thus, the user is enabled to control the exact application of transfer functions, simple shading, or other techniques on a per ray basis.

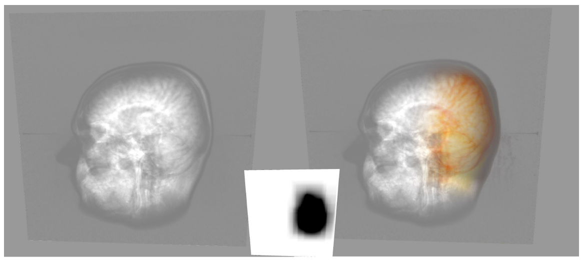
Figure 6: Blending a one-dimensional transfer function and simple accumulative shading by using value parameters, with the inset showing the interpolation weight.

4 RESULTS

In this section we will look at the results of our approach, both in terms of rendering performance and level of control. All implementations and tests have been done using our C++/GLSL GPU-framework "Cascada", running on an Intel Core2 Duo (2.4 GHz) with 2 GB RAM and an Nvidia Geforce 8800 GTS under Windows XP. The viewport size for the ray casting has been 512^2 pixels, with 256 loop iterations in the default setting. Ray casting is done by a two-pass approach as described in section 3.2, without further optimizations. The data set used in some figures and the timings below is an MRI volume of 256^3 voxels, with 16 bit floating point scalar values per voxel (represented as IEEE-754r compliant type "half"). The rendering performance is averaged over a full rotation of the volume to account for viewpoint dependency, with the volume covering at least 90% of the viewport.

4.1 Performance

We have stated that the overhead for the additional texture lookup is negligible on current graphics hardware. Aside from the number of the pixels used for casting ray, the rendering performance is mainly influenced by the number of iterations due to the multiple texture fetches along the ray. Accessing the ray texture imposes only one additional texture fetch per fragment and does not contribute to the overall performance, as can be seen by the table below (static ray texturing).

Of course, while editing and reloading the texture as described in the preceding section, the performance is limited by the CPU–GPU communication bottleneck. Note that this includes setting a whole neighborhood of values, not only the pixel currently "selected", so that there is some additional computation performed by the CPU. In order to reduce the performance hit by updating the texture per frame, the transfer should be limited to the actually edited region of the texture, of course. In addition to the initial resolution of the ray texture, we have used the hierarchical approach for updating an area of rays comparable to that of direct update. This results in a clear performance gain compared to working with an equally large area of the full resolution. However, a detailed control of the neighborhood with arbitrary shapes is not possible when using the hierarchical approach.

Rendering Mode	Average FPS
Default ray casting (RC)	61
RC with static ray texturing	59
Direct update	9
Hierarchical update	33

Table 1: Average performance for standard ray casting, additional ray textures, and update strategies, respectively

Although it is quite difficult to compare the two classes of parameters (geometric and value, respectively) due to their different usage, we have tried to measure them in equally complex scenarios. Therefore we decreased the step size in equal steps resulting in an increased number of loop iterations. For the value parameters we have increased the threshold for early ray termination likewise, which yields also more iterations (due to the delayed termination of the loop). As expected, the performance is not related of the direct type of parameter, but to its use within the shader.

4.2 Control

The two methods of transforming the ray texture led to different behaviour while editing the texture. For the *view aligned* approach, the functionality can be interpreted as looking through a "window" of altered properties. This is similar to the idea of ray casting as image space method, where rays are cast through the viewing plane and sample the volume along the ray (usually within the volume's boundaries). As expected, this works only intuitively for fixed viewing positions due to the view-dependency: for example, an increased level of detail applied to a specific region of interest will affect other regions once the camera has moved.

A more intuitive control is to transform the ray texture with the volume (*volume aligned*). This approach counteracts to some degree with the ray casting concept where the rays are all in viewing direction. Their properties, however, are changed only partially depending on the visibility and transformation of the edited ray texels.

Thus, the former method can be used, for example, as a tool for inspecting parts of the volume, analog to a filter being applied to data. As both can be interchanged easily during rendering, manipulations of the texture can be made on-the-fly in one view or the other. Figure 7 shows the different results for the volume and view aligned mode, respectively.

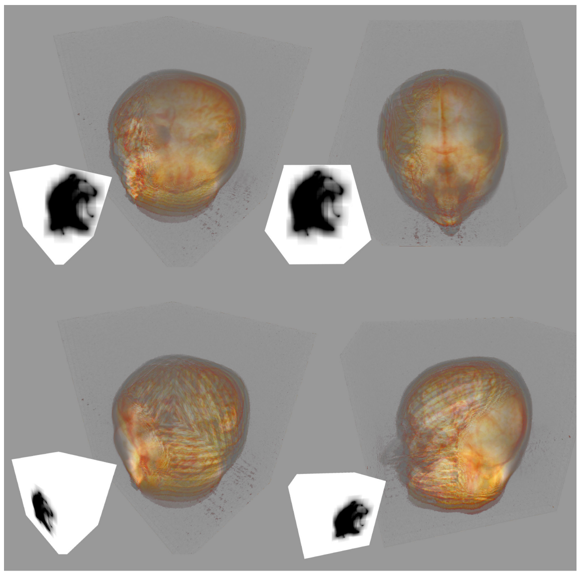
Figure 7: Comparison of the two editing modes: "view aligned" (top) and "volume aligned" (bottom). Note the inset illustrating the different effects

5 CONCLUSION AND FUTURE WORK

We have presented the concept of ray textures for controlling parameters of individual rays in volume rendering. Therefore, we have classified parameters as *geometric* and *value* properties that can be edited and used separately or in combination. We have also described two modes of applying the ray texture: *view aligned* or *volume aligned*, which differ only slightly in terms of implementation and can thus be used interchangeably during run time. In addition to some neighborhood of the current position allowing for changing a whole set of rays at once, we have shown using a hierarchical approach is more efficient, if some simple quadratic neighborhood is sufficient.

We would like to investigate further on this concept by extending it to data-driven ray textures. That way the textures could be initialized with data from the volume itself to allow guided editing or adaptive performance control. Therefore, the volume would be rendered first to an offscreen buffer and from this some relevant information (e.g., gradients, silhouettes) could be extracted and used in different contexts. Also, we would like to investigate other interaction modes and evaluate them in various applications. Yet another extension would be to apply the ray texture concept to offscreen rendering, as used for processing data in the context of image processing, data analysis, etc.

ACKNOWLEDGEMENTS

We would like to thank all the students involved in the development of our GPU-based framework, especially Guido Lorenz for his great contribution to "Cascada". Also we would like to thank our colleagues for all the useful discussions and comments on the topic. Finally, we thank the reviewers for their feedback helping to improve the paper.

REFERENCES

[EHK$^+$06] Klaus Engel, Markus Hadwiger, Joe M. Kniss, Christof Rezk-Salama, and Daniel Weiskopf. *Real-Time Volume Graphics*. A K Peters, 2006.

[HBSL03] Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *HWWS '03: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, pages 92–101, 2003.

[KSFB06] Andrea Kratz, Rainer Splechtna, Anton L. Fuhrmann, and Katja Bühler. GPU-Based High-Quality Hardware Volume Rendering For Virtual Environments. In *International Workshop on Augmented Environments for Medical Imaging and Computer Aided Surgery*, 2006.

[KW03] Jens Krüger and Rüdiger Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization 2003*, pages 287–292, 2003.

[LB07] Andreas Langs and Matthias Biedermann. Filtering video volumes using the graphics hardware. In Bjarne K. Ersbøll and Kim Steenstrup Pedersen, editors, *SCIA*, volume 4522 of *Lecture Notes in Computer Science*, pages 878–887. Springer, 2007.

[Lev90] Marc Levoy. Efficient ray tracing of volume data. *ACM Trans. Graph.*, 9(3):245–261, 1990.

[MMG07] Muhammad Muddassir Malik, Torsten Möller, and Meister Eduard Gröller. Feature peeling. In *Proceedings of Graphics Interface 2007*, pages 273–280, May 2007.

[OLG$^+$07] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[RGW$^+$03] Stefan Röttger, Stefan Guthe, Daniel Weiskopf, Thomas Ertl, and Wolfgang Strasser. Smart Hardware-Accelerated Volume Rendering. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym*, pages 231–238, 2003.

[RSK06] Christof Rezk-Salama and Andreas Kolb. Opacity Peeling for Direct Volume Rendering. *Computer Graphics Forum (Proc. Eurographics)*, 25(3):597–606, 2006.

[Sch05] Henning Scharsach. Advanced GPU Raycasting. In *Proceedings of CESCG 2005*, pages 69–76, 2005.

[SHN$^+$05] Henning Scharsach, Markus Hadwiger, André Neubauer, Stefan Wolfsberger, and Katja Bühler. Perspective Isosurface and Direct Volume Rendering for Virtual Endoscopy Applications. In *Proceedings of EuroVis/IEEE-VGTC Symposium on Visualization 2006*, 2005.

[WEE03] Daniel Weiskopf, Klaus Engel, and Thomas Ertl. Interactive Clipping Techniques for Texture-Based Volume Visualization and Volume Shading. *IEEE Transactions on Visualization and Computer Graphics*, 9(3):298–312, 2003.