Acceleration Techniques for GPU-based Volume Rendering

J. Krüger and R. Westermann[*]
Computer Graphics and Visualization Group, Technical University Munich

Abstract

Nowadays, direct volume rendering via 3D textures has positioned itself as an efficient tool for the display and visual analysis of volumetric scalar fields. It is commonly accepted, that for reasonably sized data sets appropriate quality at interactive rates can be achieved by means of this technique. However, despite these benefits one important issue has received little attention throughout the ongoing discussion of texture based volume rendering: the integration of acceleration techniques to reduce per-fragment operations.

In this paper, we address the integration of early ray termination and empty-space skipping into texture based volume rendering on graphical processing units (GPU). Therefore, we describe volume ray-casting on programmable graphics hardware as an alternative to object-order approaches. We exploit the early z-test to terminate fragment processing once sufficient opacity has been accumulated, and to skip empty space along the rays of sight. We demonstrate performance gains up to a factor of 3 for typical renditions of volumetric data sets on the ATI 9700 graphics card.

CR Categories:
I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - color, shading and texture—; I.3.8 [Computer Graphics]: Applications—

Keywords: Volume Rendering, Programmable Graphics Hardware, Ray-Casting

1 Introduction

Real-time methods to directly convey the information contents of large volumetric scalar fields are still a challenge to the computer graphics community. Based on the observation that the capability of a single general-purpose CPU is not sufficient to achieve interactivity or even real-time for large data sets in general, considerable effort has been spent on the development of acceleration techniques for CPU-based volume rendering and on the design and exploitation of dedicated graphics hardware.

Among others, one research direction has lead to volume rendering techniques that exploit hardware assisted texture mapping. Fundamentally, these systems re-sample volume data, represented as a stack of 2D textures or as a 3D texture, onto a sampling surface or so-called proxy geometry. The most common surface is a plane that can be aligned with the data, aligned orthogonal to the viewing direction, or aligned in other configurations (such as spherical shells). The ability to leverage the embedded trilinear interpolation hardware is at the core of this acceleration technique.

This capability was first described by Cullip and Neumann[Cullip and Neumann 1993]. They discussed the necessary sampling schemes as well as axis-aligned and viewpoint-aligned sampling planes. Further development of this idea, as well as the extension to more advanced medical imaging, was described by Cabral et al. [Cabral et al. 1994]. They demonstrated that both interactive volume reconstruction and interactive volume rendering was possible with hardware providing 3D texture acceleration. Today, texture based approaches have positioned themselves as efficient tools for the direct rendering of volumetric scalar fields on graphics workstations or even consumer class hardware [Van Geldern and Kwansik 1996; Westermann and Ertl 1998; Meißner et al. 1999; Rezk-Salama et al. 2000; Engel et al. 2001; Guthe et al. 2002; Kniss et al. 2002]. Furthermore, it is commonly accepted that for reasonably sized data sets appropriate quality at interactive rates can be achieved by means of these techniques.

Nevertheless, despite the benefits of texture based volume rendering, one important drawback has not been addressed sufficiently throughout the ongoing discussion. For a significant number of fragments that do not contribute to the final image, texture fetch operations, numerical operations, i.e. lighting calculations, and per-pixel blending operations are performed. It is thus important to integrate standard acceleration techniques for volume rendering, like early ray termination and empty-space skipping [Levoy 1990; Danskin and Hanrahan 1992; Sobierajski 1994; Yagel and Shi 1993; Freund and Sloan 1997], into texture based approaches. This is the novel contribution of this paper, and we will demonstrate the effectiveness of the proposed algorithms on a variety of real-world data sets.

To enable the integration of acceleration techniques into 3D texture based volume rendering, we propose a stream model for volumetric ray-casting that exploits the intrinsic parallelism and efficient communication on modern graphics chips. Our implementation builds upon the functionality that is provided on current programmable graphics hardware. In particular, we employ the Pixel Shader 2.0 API [Microsoft 2002], a specific set of instructions and capabilities in DirectX9-level hardware, which allows us to perform hardware supported per-fragment operations like texture fetches and arithmetic operations on our target architecture.

The essential mechanism that allows us to effectively integrate early ray termination and empty-space skipping into the ray-casting process is the early z-test. It is applied before the shader program is executed for a particular fragment, and it can be used to avoid the execution thus enabling a fragment processor to process the next incoming fragment. The early z-test, however, is only enabled if all other per-fragment tests are disabled and the fragments depth value is not modified in the shader. The effect of the early z-test can be demonstrated quite easily, by simply drawing an opaque quadrilateral occluding parts of a volume prior to rendering the volume. If the depth test is enabled and depth values have been written in the first pass, a considerable speed up can be perceived that is directly proportional to the area of the occluded parts of the volume.

The remainder of this paper is organized as follows. First, we

[*][jens.krueger,westermann]@in.tum.de

review 3D texture based volume rendering techniques and we put emphasis on some intrinsic drawbacks and limitations of current implementations. Next, we present volume ray-casting on programmable graphics hardware as an alternative to traditional object-order approaches. Then, we outline new techniques to integrate early ray termination and empty-space skipping into the ray traversal process. We finally discuss the basic properties of our approach and sketch future challenges in the field of volume rendering.

1.1 3D Texture Based Volume Rendering

Volume rendering via 3D textures is usually performed by slicing the texture block in back-to-front order with planes oriented parallel to the view plane, i.e. see Figure 1.

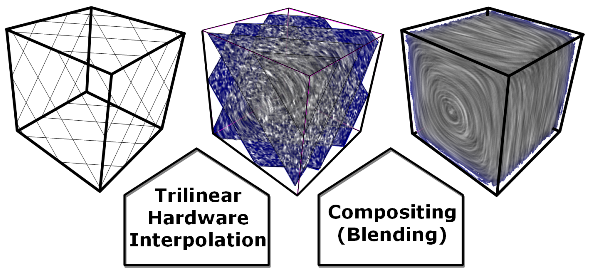

Figure 1: Volume rendering via 3D texture slicing.

For each fragment the 3D texture is sampled by trilinear interpolation, and the resulting color sample is blended with the pixel color in the color buffer. If slicing is performed in front-to-back order the blending equation changes from

$$C_{dst} = (1-\alpha_{src})C_{dst} + \alpha_{src}C_{src}$$

to

$$\begin{aligned} C_{dst} &= C_{dst} + (1-\alpha_{dst})\alpha_{src}C_{src} \\ \alpha_{dst} &= \alpha_{dst} + (1-\alpha_{dst})\alpha_{src} \end{aligned}$$

Here, C_{dst}, α_{dst} and C_{src}, α_{src} are the color and opacity values of the color buffer and the incoming fragment, respectively.

In front-to-back order an additional α-buffer needs to be acquired to store the accumulated opacity. Pre-multiplication of source color with source alpha is realized in a pixel shader as proposed later in this work.

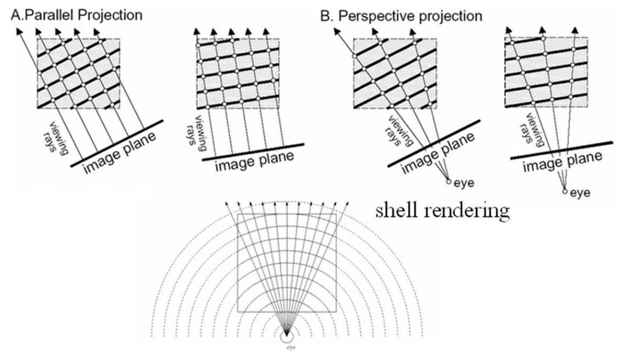

Figure 2: Different proxy geometries used in texture based volume rendering.

As a direct implication of the selected proxy geometry in 3D texture based volume rendering, from pixel to pixel the data is sampled at varying rates. In Figure 2 the sampling patterns are shown for 2D and 3D texture based volume rendering. As can be seen, the sampling on spherical shells around the view point mimics at best the kind of sampling that is performed in volume ray-casting. Here, if no acceleration technique is employed, the rays of sight are traversed with a constant step size. Spherical clip geometries, however, have turned out to be rather impractical due to the huge number of geometry that has to be generated, transferred and finally rendered on the GPU.

Probably the most obstructive limitation of texture based volume rendering, however, is the huge amount of fragment and pixel operations, like texture access, lighting calculation, and blending, that are performed, but which do not contribute to the final image. In order to verify our hypothesis, typical volume data sets are shown in figure 3. Most commonly in volume rendering applications the focus is on emphasizing boundary regions or selected material values. Usually this is done by locally adjusting the color and opacity in order to highlight these structures with respect to others. As a matter of fact, due to the inherent occlusion effects the majority of fragments that are generated during volume rendering never contribute to the image. In addition, because non-relevant structures are often suppressed by setting their opacity to zero, many fragments contributing to empty space are generated and have to be processed. For instance, in the images shown in figure 3 only between 0.2% and 4% of all generated fragments contribute to the final image.

If the volume rendering technique should also provide realistic simulation of lighting effects, the waste of per-fragment operations is even more dramatic. Lighting effects are usually simulated by means of a gradient texture, which is comprised of the material gradients and the scalar material values in the RGB and α color components, respectively. The illumination model is evaluated on a per-fragment basis in the shader programm. Consequently, even for non-visible fragments the gradient texture has to be sampled and numerical computations have to be performed.

2 Volume Ray-Casting on Graphics Hardware

Driven by the evolution of commodity graphics hardware from fixed function pipelines towards fully programmable, floating point pipelines, the demand for efficient strategies to realize graphics algorithms on this kind of architectures is continuously increasing. With regard to the observation that the power of GPUs is currently increasing much faster than that of CPUs, algorithms amenable to the intrinsic parallelism and efficient communication on modern GPUs are worth an elaborate investigation. This direction of research will lead to a considerable improvement of state-of-the-art rendering techniques, and it will spawn completely new classes of graphics algorithms.

Perhaps the most relevant discussion of contemporary and future graphics hardware and programming models that exploit these architectures can be found in [Purcell et al. 2002]. In this work a stream model for ray-tracing was proposed, which takes advantage of parallel fragment units and high bandwidth to texture memory. The stream of homogeneous data generated by the rasterization stage is fed to the fragment units. These units work in parallel on different data, i.e. in a SIMD-like manner. By using the functionality provided by DirectX9-level hardware, a general strategy to exploit programmable fragment processors for ray-tracing was sketched.

Our current work also relies upon DirectX9-level hardware as provided by the ATI 9700. We have used the Pixel Shader 2.0 API for the implementation of volume ray-casting on this graphics ac-

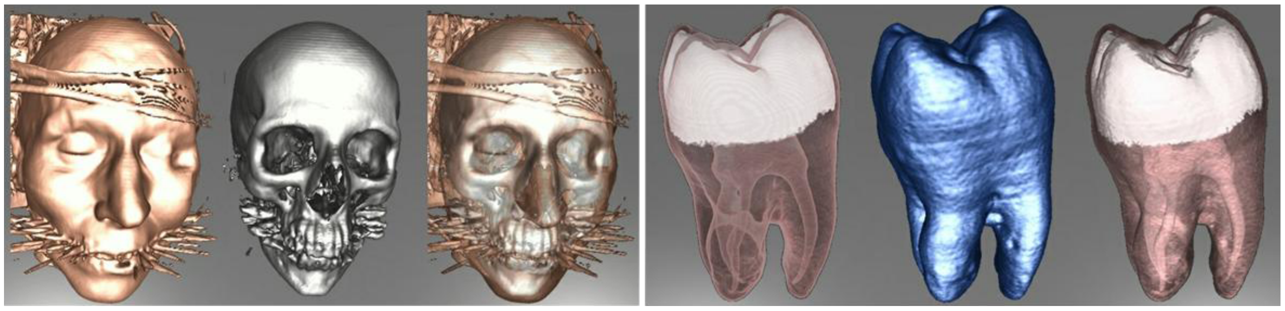

Figure 3: Volume rendering examples.

celerator. In particular, the availability of the following features was essential for the realization:

- *Per-fragment texture fetch operations:* In the pixel shader program it is possible to access up to 8 different textures, but only a limited number of dependent texture fetches can be performed on the ATI.
- *Texture render target:* Instead of using the frame buffer, rendering can be directed to a 2D texture map aligned with the viewport. This texture can be accessed in the following rendering passes. Consequently, this mechanism allows different passes to communicate their rendering results to consecutive passes. Note that, floating point textures are available. Thus, negative values can be stored.
- *Texture coordinate generation:* Texture coordinates to be used for texture access can be specified directly or manipulated in the shader program.
- *Per-fragment arithmetic:* A number of arithmetic operations on scalar or vector variables can be performed in the shader program. These include the computation of simple arithmetic operations, i.e. +,-,*, but also more complex ones like dot products and square roots.
- *Depth replace:* A fragment can replace its depth value by writing an arbitrary value to the z-buffer.

All of these features are supported on current graphics cards like the ATI 9700. Together with the early z-test they build the basis for the proposed volume rendering acceleration techniques.

2.1 Ray-Casting Implementation

The key to volume ray-casting is to find an effective stream model that allows one to continuously feed multiple, data-parallel fragment units on recent chips. In addition, the number of fragments to be processed and the number of operations to be performed for each fragment should be minimized.

The proposed algorithm is a multi-pass approach. For each fragment, it casts rays of sight through the volume until an opacity threshold is reached or a selected iso-value is hit. In the latter case, the coordinates in local texture space of the intersection points with the surface are written to a 2D texture. This texture is used in a final pass to restrict necessary computations, i.e. access to a gradient texture and shading computations, to these points.

Prior to ray traversal, for each pixel the direction in local texture coordinates of the ray through that pixel is computed. This direction is stored in two 2D textures, and it can now be retrieved directly in all upcoming rendering passes. Ray traversal is done in a fixed number of rendering passes, each performing a constant number of steps along the rays. The render target in each pass is a 2D texture that is accessed in consecutive passes to access accumulated color and opacity values.

Between any two main passes, an additional pass is performed that simply tests whether the actual opacity value has already exceeded a specified threshold or an iso-surface is hit. Depending on the result of this test, the pixel shader modifies the z-value. If the test succeeds, the z-value is set to the maximum value, it is set to zero otherwise. As a consequence, if the z-test is set to GREATER, all consecutive main passes will be discarded due to the early z-test.

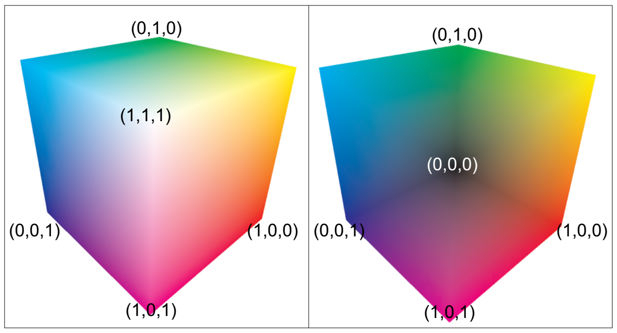

Figure 4: Rendering front faces (left) and back faces (right) of the volume bounding box in order to generate ray directions and texture coordinates of first ray intersection points.

In essence, the following steps are performed (the depth test is always set to GREATER):

- **Pass 1:** *(Entry point determination):* The front faces of the volume bounding box are rendered to a 2D RGB texture. 3D texture coordinates of each vertex are issued as per-vertex color COL (see left of Figure 4). The result is a 2D texture (TMP) having the same resolution as the current viewport. The color components in the texture correspond to the first intersection point between the rays of sight and the volume. Coordinates of the intersection points are given with respect to texture space.
- **Pass 2** *(Ray direction determination):* The same steps as in Pass 1 are performed, but now back faces of the volume bounding box are rendered to a 2D RGBA texture (DIR) (see right of Figure 4). In this pass, a fragment shader is issued that fetches for each fragment the respective value from TMP and computes the normalized ray direction as *normalize*(COL - TMP). The result is rendered into the color components of the render target. Again, COL corresponds to the current vertex color. In addition, the length of the non-normalized direction

is rendered to the alpha component. For every fragment, the 2D floating point render target DIR now holds the normalized ray direction in local texture coordinates as well as the length of each ray passing through the volume.

- **Main passes 3 to N** *(Ray traversal and early ray termination)*: In each pass, M steps along the rays are performed, and rendering is directed to a 2D texture, RES, that can be accessed in the consecutive passes. The front faces of the volume bounding box are rendered. Multiple parameters are issued at the vertices of these faces: Texture coordinate 1 gets assigned the normalized device coordinate (x,y) of each vertex. It is used to index textures DIR and RES. 3D texture coordinates of each vertex are issued as per-vertex color (C). Thus, the parametric ray equation $r(t) = C + t \cdot DIR[x][y]$ in local texture coordinates is directly available by performing one texture lookup. In constant color 1 (C1) the step size Δ to be performed along the rays of sight is specified. In constant color 2 (C2) the product of Δ and the number of steps already performed $(\Delta \cdot M \cdot (N-3))$ is issued. By initially setting t to C2, ray traversal now involves incrementing t about C1 and fetching the respective value from the 3D texture at r(t). This is done M times, at each sample blending the current contribution with the contribution that has been accumulated up to this position. Finally, the result is blended with RES[x][y] and it is written back to the 2D render target RES. If by means of the opacity value read from DIR, which stores the length of each ray within the volume, it turns out that the current ray has already left the volume, an opacity value of 1 is written to RES.

- **Intermediate passes 3 to N** *(Stopping criterion)*: In an intermediate pass, the front faces are rendered again, and a shader is issued that accesses the current opacity value and compares it with a constant threshold T (Note that in case the ray has left the volume, opacity is always larger than T). Essentially, it performs the following conditional statement:
  ```
  IF(RES[x][y] > T) THEN z = MAX, out(0)
  ELSE z = 0, out(0)
  ```
  If the ray has to be terminated, the respective value in the z-buffer is set to the maximum value and the color buffer is kept by drawing zero color and opacity. Otherwise, the shader has no effect.

2.2 Iso-Surface Ray-Casting

If we employ GPU-based ray-casting to render illuminated surfaces, the shader setup becomes even more simple. The first two rendering passes remain unchanged. In passes 3 to N, however, M steps along the ray are performed but considerably less arithmetic is required. Although the execution order is still front-to-back, now the traversal within each pass is performed back-to-front. At each sample we fetch the respective value from the 3D texture, and we test whether the scalar value is larger or equal to the selected threshold. If this condition is true, we store the current position along the ray, r(t), in a temporary variable. By going back-to-front within each pass, only the intersection point closest to the viewpoint is kept. At the very end of the shader, the temporary variable is checked. If it´s not equal to an initial value the content is used to look up a 3D gradient texture and to perform surface lighting. In this case, the final color value that is rendered to the texture render target contains the illumination of the surface point, and an opacity value equal to one to guarantee for early ray termination in consecutive passes. The intermediate pass is the same as proposed above.

2.3 Empty-Space Skipping

Apart from early ray termination, one important issue needs to be addressed to speed up volumetric ray-casting. Empty-space skipping essentially relies on an additional data structure, which encodes empty regions in the data. For instance, an octree hierarchy can be employed, which stores at every inner node statistical information about child nodes, e.g. min/max-bounds for the region that is covered by the node. If this information is present to the ray-caster, it can effectively increase the sampling distance along the ray when entering empty regions.

In our current implementation, to speed up rendering we use a 3D raster with constant cell size corresponding to one particular octree level. We always encode disjoint blocks of size 8^3 within the original data set, and for every block we store the minimum and the maximum scalar value contained in this block. The data is stored in a 3D RGB texture map with 1/8 the size of the original volume in every dimension. The minimum and maximum values are encoded in the R- and G-components, respectively. In addition to this texture we generate a 2D texture, CT, that indicates for every pair (min/max) whether there is at least one non-zero color component in the range from min to max after shading via a color table has been performed. This texture is updated on the CPU and loaded on the graphics subsystem whenever the color table is modified.

In order to test for empty space we extend the intermediate pass that is executed right after each of the passes 3 to N described above. Therefore, the front faces are rendered in the same way as described, but the ray is traversed with a step size of $8 \cdot \Delta$. The number of steps is decreased accordingly. Instead of the original data set, the coarse resolution copy is accessed at every sample point r(t). The R- and G-components at every sample are used to index CT, and thus to test whether empty space is present or not. Whenever at one of the sample points non-empty space is found, the entire segment processed in this pass is considered to be non-empty. In this case, the z-value is set zero and the color buffer is left unchanged. If empty space is determined, the z-value is set to the maximum, thus forcing the next main shader pass to be skipped.

Altogether, the following conditional statement is now executed in the intermediate shader:
```
IF(RES[x][y] > T) OR (EmptySpace==TRUE)
THEN z = MAX, out(0)
ELSE z = 0, out(0)
```
Note that the z-value is reset to 0 as soon as no empty space was found, thus re-enabling ray-traversal.

3 Discussion and Results

The proposed stream programming model for volume ray-casting has several benefits over object-order projection methods. Besides equal sampling density along the rays of sight, the most important one is the reduction of the number of fragment operations to be executed if opaque structures or empty space are contained in the data. Because the traversal procedure is split into multiple passes with at most M steps along the ray, at most M texture fetch operations have to be performed to no purpose. This happens if the first sample within a segment already saturates the opacity, generating M-1 void samples, or if the first sample is already outside the volume.

Therefore, M should be rather small compared to the maximum number of samples that have to be performed for the longest ray through the volume. On the other hand, the number of rendering passes to be executed also depends on M. Assuming the longest ray through the volume to perform K steps, then $\lceil K/M \rceil$ passes need to be issued. Because with each pass some overhead comes along, i.e. rendering the front faces and accessing various textures, supposedly M should be as large as K. Then, however, all rays shorter than the longest one have to compute the same number of sample points along the ray because a shader program cannot be discarded during its execution. Regardless the optimal choice for M, in the current implementation it was restricted to 8 due to hardware shader limitations.

For shaded and opaque iso-surfaces, the gradient only needs to be reconstructed and illuminated once in every rendering pass 3 to N until an iso-surface is hit. Note that due to the fact that fragment shaders do not yet support conditional execution of expressions, gradient reconstruction and illumination is always performed even if no surface is hit. The fact, however, that these computations only have to be performed once every M steps significantly accelerates the rendering of opaque iso-surfaces.

It is of course worth noting that for highly transparent and dense data sets, where each ray needs to be traversed until it exits the volume, no gain in performance can be expected. In this case, the intermediate pass to check for early ray termination and empty space introduces some overhead.

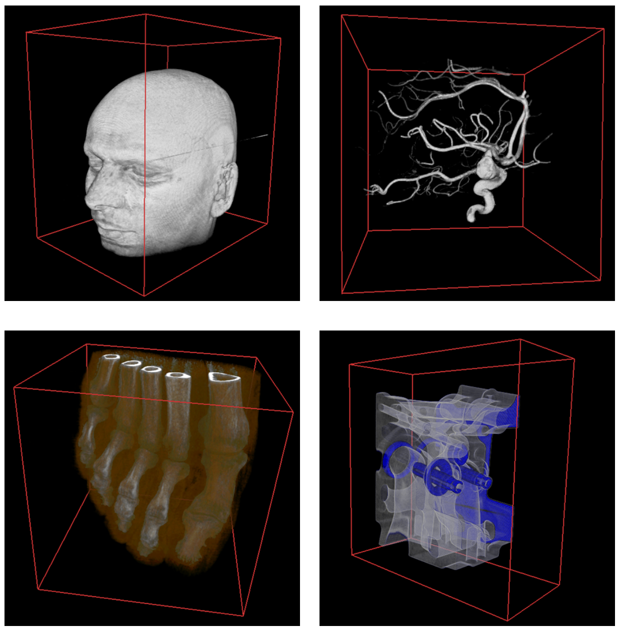

Figure 5: *These images show the example data sets we have used to test the performance of our acceleration techniques. The first 3 data sets are of size* 256^3, *the engine data set is of size* $256^2 \cdot 128$.

Let us now demonstrate the effectiveness of the proposed acceleration schemes for 3D texture based ray-casting. We show four example data sets, an MRI head, the Philips aneurism, the GE engine and the Philips foot, all from the volren.org web-page (see Figure 5). Rendering was directed to a 512^2 viewport. In the table below we give exact timings for our algorithm, which are compared to the times achieved by the slice-based volume rendering (*SBVR*). *RC* is the time consumed by the ray-caster without any acceleration. *RC-α* and *RC-β* account for early ray termination and early ray termination combined with empty-space skipping, respectively.

Table 1: Timings (fps) for 3D texture based volume rendering.

	SBVR	RC	RC-α	RC-β
Head	7.1	4.8	14.4	23.4
Aneurism	7.1	4.8	8.9	19.6
Foot	7.1	4.8	10.2	17.7
Engine	10.4	6.3	6.3	13.6

Obviously, with regard to performance, SBVR is clearly superior to volume ray-casting without early ray termination and empty-space skipping, because in SBVR only those fragments have to be processed that are within the volume bounding box. In RC, depending on the choice of M many fragments outside the volume have to be processed. If early ray termination and empty-space skipping is enabled, however, we can see a performance gain of RC in typical real-world examples exhibiting opaque structures and empty regions. Unfortunately, the speed up is not as dramatic as we would have expected. This is due to the fact that we need to perform the intermediate shader pass to check for the stopping criterion and to replace z-value in order to exploit the early z-test. The intermediate itself, however, cannot be discarded because it modifies the z-values. Thus, empty-space detection is always performed, even if early ray termination is already in use. We could easily integrate empty-space detection into the main shader passes, but therefore we need one additional flag that can be written from the shader to identify whether empty-space has been detected or not. For this purpose we can not use the alpha channel, because it already serves as a flag indicating early ray termination.

For semi-transparent volumetric data sets, where neither early ray termination nor empty-space skipping can be applied, the overhead to perform the intermediate pass to check the stopping criterion manifests in a loss in performance of about 30%. This, however, is always the price that has to be payed for the integration of acceleration techniques if the data set does not provide appropriate stopping cues.

4 Conclusions

In this paper, we have described a stream model for volume ray-casting on DirectX9-level graphics hardware that is programmable via the Pixel Shader 2.0 API. Our approach includes standard acceleration techniques for volume ray-casting, like early ray termination and empty-space skipping. By means of these acceleration techniques, the proposed framework is capable of efficiently rendering large volumetric data sets including opaque structures exhibiting occlusions and empty regions.

For many real-world data sets, the proposed method is significantly faster than previous texture based approaches, yet achieving the same image quality. This is due to the effective reduction of texture fetch and arithmetic operations in case that early ray termination and empty-space skipping can be applied. For the rendering of shaded iso-surfaces, access operations to the gradient texture and per-fragment shading operations are considerably minimized.

The proposed acceleration techniques have been implemented on current commodity graphics hardware by means of the early z-test. Due to the fact that the early z-test is disabled as long as the fragments z-value is replaced in the shader program, an additional but simple shader pass has to be performed to check for the stopping criterion. Once the early z-test remains active even if the z-value is replaced or the stencil test is enabled, more optimal implementations can be presented. In particular, if we are able to use additional buffers to communicate per-fragment results from one rendering pass to the next one, empty-space skipping can be integrated into the main shader passes. In this way we can considerably decrease the overhead that is introduced by the intermediate pass in the current implementation.

Acknowledgements

Special thanks to Joe Kniss, Christof Rezk-Salama and Markus Hadwiger for fruitful discussions and for providing the images in Figures 1, 2 and 3.

References

CABRAL, B., CAM, N., AND FORAN, J. 1994. Accelerated volume rendering and tomographic reconstruction using texture mapping hardware. In *Proceedings ACM Symposium on Volume Visualization 94*, 91–98.

CULLIP, T., AND NEUMANN, U. 1993. Accelerating volume reconstruction with 3D texture hardware. Tech. Rep. TR93-027, University of North Carolina, Chapel Hill N.C.

DANSKIN, J., AND HANRAHAN, P. 1992. Fast Algorithms for Volume Ray Tracing. In *ACM Workshop on Volume Visualization '92*, 91–98.

ENGEL, K., KRAUS, M., AND ERTL, T. 2001. High-quality pre-integrated volume rendering using hardware-accelerated pixel shading. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*.

FREUND, J., AND SLOAN, K. 1997. Accelerated volume rendering using homogeneous region encoding. In *Proceedings IEEE Visualization '97*, 191–197.

GUTHE, S., ROETTGER, S., SCHIEBER, A., STRASSER, W., AND ERTL, T. 2002. High-quality unstructured volume rendering on the PC platform. In *ACM Siggraph/Eurographics Hardware Workshop*.

KNISS, J., PREMOZE, S., HANSEN, C., AND EBERT, D. 2002. Interactive translucent volume rendering and procedural modeling. In *Proceedings of IEEE Visualization 2002*, 168–176.

LEVOY, M. 1990. Efficient Ray Tracing of Volume Data. *ACM Transactions on Graphics 9*, 3 (July), 245–261.

MEISSNER, M., HOFFMANN, U., AND STRASSER, W. 1999. Enabling classification and shading for 3d texture mapping based volume rendering using OpenGL and extensions. In *IEEE Visualization '99*, 110–119.

MICROSOFT, 2002. DirectX9 SDK. http://www.microsoft.com/DirectX.

PURCELL, T., BUCK, I., MARK, W., AND HANRAHAN, P. 2002. Ray tracing on programmable graphics hardware. *Computer Graphics SIGGRAPH 98 Proceedings*, 703–712.

REZK-SALAMA, C., ENGEL, K., BAUER, M., GREINER, G., AND ERTL, T. 2000. Interactive volume rendering on standard PC graphics hardware using multi-textures and multi-stage rasterization. In *SIGGRAPH/Eurographics Workshop on Graphics Hardware*, 109–119.

SOBIERAJSKI, L. 1994. *Global Illumination Models for Volume Rendering*. PhD thesis, The State University of New York at Stony Brook. Disertation.

VAN GELDERN, A., AND KWANSIK, K. 1996. Direct Volume Rendering with Shading via Three-Dimensional Textures. In *ACM Symposium on Volume Visualization '96*, R. Crawfis and C. Hansen, Eds., 23–30.

WESTERMANN, R., AND ERTL, T. 1998. Efficiently using graphics hardware in volume rendering applications. In *Computer Graphics (SIGGRAPH 98 Proceedings)*, 291–294.

YAGEL, R., AND SHI, Z. 1993. Accelerated Volume Animation by Space-Leaping. In *Proceedings IEEE Visualization '93*, 62–69.