Ray Casting of Multiple Volumetric Datasets with Polyhedral Boundaries on Manycore GPUs

Bernhard Kainz [*] Markus Grabner [*] Alexander Bornik [†] Stefan Hauswiesner [*] Judith Muehl [*]
Dieter Schmalstieg [*] [‡]

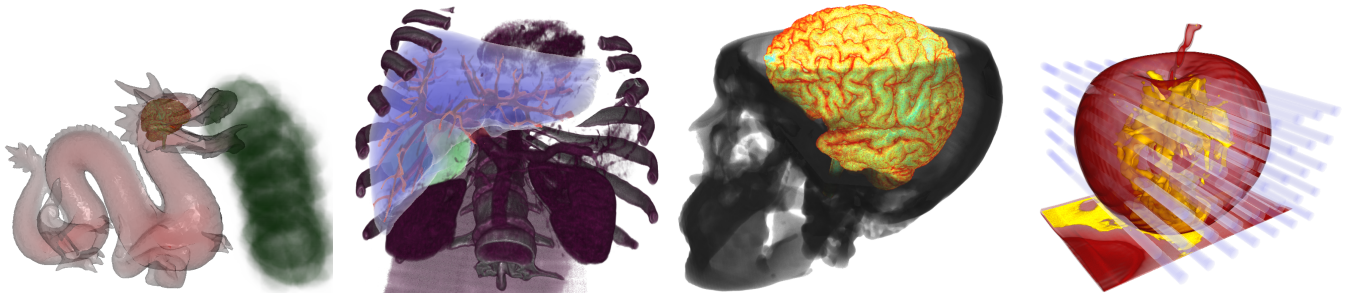


Figure 1: *Examples of our new approach to efficiently combine direct volume rendering with polyhedral geometry. Our renderer is able to render many volumes together with complex geometry. Images from left to right: Rendering a 20k polygons dragon with a* 256^3 *brain volume and nine* 64^3 *smoke clouds; three 10k segmented vessels, tumor and liver surface in a ribcage* 512^3 *volume; a segmented brain in an open skull, each a* 256^3 *volume; 35 translucent rods each with 76 polygons placed in an apple with* 512^3 *voxels.*

Abstract

We present a new GPU-based rendering system for ray casting of multiple volumes. Our approach supports a large number of volumes, complex translucent and concave polyhedral objects as well as CSG intersections of volumes and geometry in any combination. The system (including the rasterization stage) is implemented entirely in CUDA, which allows full control of the memory hierarchy, in particular access to high bandwidth and low latency shared memory. High depth complexity, which is problematic for conventional approaches based on depth peeling, can be handled successfully. As far as we know, our approach is the first framework for multi-volume rendering which provides interactive frame rates when concurrently rendering more than 50 arbitrarily overlapping volumes on current graphics hardware.

CR Categories: I.3.3 [Computing Methodologies]: COMPUTER GRAPHICS—Picture/Image Generation; I.3.6 [Computing Methodologies]: COMPUTER GRAPHICS—Methodology and Techniques

[*]Graz University of Technology (TU-Graz), Institute for Computer Graphics and Vision (ICG), Inffeldgasse 16, 8010 Graz, AUSTRIA
[†]Ludwig Boltzmann Institute for Clinical-Forensic Imaging (LBI), Universitaetsplatz 4, 8010 Graz, AUSTRIA
[‡]kainz|grabner|bornik|hauswiesner|muehl|schmalstieg@icg.tugraz.at

1 Introduction

Ray casting has prevailed as the most versatile approach for direct volume rendering because of its flexibility in producing high quality images of medical datasets, industrial scans and environmental effects [Engel et al. 2006]. The large amount of homogeneous data contained in a volumetric model lends itself very well to parallelization. Today even a commodity graphics processing unit (GPU) is capable of ray casting a single high resolution volumetric dataset at a high frame rate using hardware-accelerated shaders.

However, rendering a single homogeneous volume is not sufficient for more advanced applications. Simultaneous rendering of multiple volumes is necessary when several datasets have been acquired. For example, in medical imaging complementary techniques such as anatomical image acquisition methods (e.g., computed tomography (CT), magnetic resonance imaging (MRI), ultrasound) and functional image acquisition methods (e.g., positron emission tomography (PET), single photon emission computed tomography, and functional MRI) can be used to examine a patient. Moreover, models of medical tools and polyhedral segmentation results in the correct geometric context are essential for computer aided surgery. Currently available methods have limited capabilities for interactive investigation of multi-modal volumetric data enhanced by polygonal models. The demand for sophisticated visualization of volume and surface data is, however, not unique to medical applications. Many other real-time graphics applications such as computer games and CAD are currently restricted to opaque surface mesh rendering and can not utilize the high visual potential of volumetric effects.

A special case of multi-volume rendering are exploded views, where multiple instances of a volume are rendered with individual clipping and transformation parameters applied to yield illustrative visualizations [Bruckner et al. 2006]. Multi-volume rendering is harder than rendering single volumes, because it requires handling of intersections and per-sample intermixing. Resampling the volumes to a single coordinate frame is not desirable because of the resulting loss in quality (or increased memory requirements) and limited flexibility of layouting. Another requirement of advanced applications is the combination of volumes with polygonal meshes. Polygonal models are useful for embedding foreign objects or ref-

erence grids into volumes, but also to apply clipping shapes, highlight areas of interest or display segmented geometry. We also note that the intersection of the bounding boxes of multiple volumes is given as a polyhedron. Such an intersection of bounding boxes is useful as a conservative approximation of the region for which intermixing of the shading contributions of intersecting volumes is required. For expressiveness and convenience, support for polygonal structures should not be limited to convex shapes or opaque materials.

Recent GPU accelerated multiple volume renderers [Roessler et al. 2008; Brecheisen et al. 2008] perform ray casting in a fragment shader and rely on a depth peeling [Everitt 2001] approach to identify homogeneous ray intervals. Depth peeling is a highly redundant multi-pass technique, which repeatedly rasterizes the same geometry to sort all possible depth samples. Consequently, a ray caster based on depth peeling does not scale to a larger number of volumes, since depth peeling of a larger number of bounding boxes or of complex polygonal meshes quickly becomes the dominant performance factor.

The use of depth peeling is implied by the use of conventional shading languages, which represent an abstraction of a graphics hardware pipeline with fixed function portions. In a conventional shading language, the programmer has limited control over the information passed from the vertex shader to the fragment shader stage. Often an efficient sort-middle approach [Molnar et al. 1994] is performed by the graphics hardware, but the strategy by which fragments are re-assigned from vertex shaders to fragment shaders is not exposed to the programmer.

It is therefore not possible to collect all depth samples obtained from rasterizing the complete bounding geometry or polygonal meshes and pass the sample collection on to a fragment shader to perform depth sorting and ray casting together in the natural order. Instead, (repeated) rasterization and ray casting of segments are interleaved. The separate shaders for depth peeling and ray casting communicate via global GPU memory, which is orders of magnitude slower than the local memory of the GPU cores. Scalability of this approach is therefore ultimately limited by memory bandwidth.

In this paper, we investigate an approach suitable for current manycore GPUs, which overcomes the inefficiencies imposed by a fixed function pipeline. We employ the new Compute Unified Device Architecture (CUDA) [Nickolls et al. 2008; NVIDIA 2008], a C-like language for general purpose computations on the GPU. Similarly to the OpenGL renderer described by [Seiler et al. 2008], we implement a volume rendering pipeline based on polygon tiling entirely in software. This approach has complete control over the rendering process and executes an efficient sort-middle approach. Geometry is rasterized only once, and all depth samples are passed on to ray casting through on-chip shared memory, which is also significantly faster than global memory. Ray casting is coherently executed in tiles of 8×8 pixels with straight rays. Both geometry and fragment processing (ray casting) are executed in a massively parallel way using CUDA's multi-threaded execution model.

The efficiency gained by this approach allows us to support a very general multi-volume data model, thereby unifying a number of advanced volume rendering approaches. The model consists of a recursively defined constructive solid geometry (CSG) structure composed of arbitrary volumetric polyhedra, i.e., two-manifold (possibly concave) triangular meshes with volumetric texturing. Every polyhedron is associated with a volumetric 3D texture, a transfer function and a geometric transformation. This structure is similar to a volume scene graph approach [Nadeau 2000] and supports any combination of the following use cases:

- Multiple intersecting volumes with individual coordinate systems and resolutions. CSG operations are used to distinguish overlapping and non-overlapping areas.
- Volumes can have arbitrary two-manifold bounding or clipping geometry. Concave polyhedra can conveniently be used as tight fitting bounding geometry. Selected areas in a volume can be highlighted interactively.
- Volumes can intersect and be intersected with transparent, concave geometry.
- Polygonal models can have volumetric effects, in particular they can be made of transparent homogeneous material (not just transparent surface rendering).
- CSG operations on a volume segmented into multiple regions with polyhedral boundaries can be used to assign individual transfer functions to each region. Each region can be transformed individually, allowing exploded views. Multiple instances of a region are possible for side by side comparison.

1.1 Contribution

To the best of our knowledge, our work is the first that supports the above-mentioned features and that enables the display of dozens of volumetric datasets with complex intersecting and clipping geometry on a consumer GPU with real-time frame rates. We describe the details of our approach and give a rationale for design choices, which is grounded in the operating principles of manycore GPUs abstracted by the CUDA programming framework. We present an analysis of the results obtained with our approach, which shows that it is independent of the polygonal depth complexity of a scene and achieves a significant speedup over state-of-the-art multi-volume rendering frameworks based on conventional shader languages.

2 Related Work

2.1 Multi-volume rendering

Rendering of multiple volumes is a recurring topic of research. While multi-volume rendering was previously limited to static images (for example [Nadeau 2000]), CPU based rendering has relatively recently achieved real-time frame rates through the use of cache-coherent, multi-threaded strategies [Grimm et al. 2004]. However, the processing power of CPUs has been overtaken by GPUs, which have more parallel execution units. Volume rendering is now commonly based on representing regular volumetric models as 3D textures in the GPU memory, and casting rays in the fragment shader. The setup of rays is done through rasterization of the volume's bounding geometry [Roettger et al. 2003; Krüger and Westermann 2003]. With the advent of branching and looping in the shader, a ray can now be traversed inside the shader rather than through multi-pass rendering [Stegmaier et al. 2005].

Through the use of depth peeling [Everitt 2001], handling of more complex scenarios becomes possible. In the context of GPU-based volume rendering, depth peeling has been used to clip volumes [Weiskopf et al. 2003] and to intersect individual volumes with geometry [Termeer et al. 2006; Borland et al. 2006]. Recently multi-volume rendering has been combined with depth peeling and dynamic shader generation into a very efficient framework [Roessler et al. 2008]. Depth peeling is used on the bounding geometries of multiple volumes to perform what are essentially CSG operations, and every distinct volumetric area is handled with optimized shader code that is derived from an abstract representation. A similar approach combining depth peeling and dynamic shader generation is taken in [Plate et al. 2007]. The work in [Brecheisen et al. 2008] is

unique in that it is able to combine multiple volumes with arbitrary layers of translucent, concave geometry.

2.2 Fragment processing

Depth peeling is a widely used multi-pass technique that relies on manipulation of the z-buffer to progressively reveal layers of occluded geometry. Generally speaking, multi-pass techniques are methods used to overcome the limitations on the resources of the GPU, in particular in terms of available storage. Given unlimited memory, an approach like the A-buffer [Carpenter 1984] can store all fragments in a list sorted by depth. Such an implementation is not really feasible in hardware, and therefore various approaches operating with bounded memory rely on multiple passes. Achieving effects involving multiple fragments often requires sorting of fragments by depth. Hardware-accelerated depth peeling [Everitt 2001] is such a method, but requires N rendering passes of N objects, leading to undesirable $O(N^2)$ complexity. Several pieces of work try to at least reduce this complexity by a constant factor, by making use of additional memory. This can take the shape of specific extensions of the fragment stream processing model of the GPU [Mark and Proudfoot 2001; Aila et al. 2003] or by relying on existing extended framebuffer capabilities [Callahan and Comba 2005; Liu et al. 2006; Bavoil et al. 2007; Bavoil and Myers 2008]. Some work combines such an approach with an approximate pre-sorting on the CPU, in order to approach linear time behavior [Wexler et al. 2005; Callahan and Comba 2005; Carr et al. 2008]. However, such a pre-sorting is not feasible for general scenes. In contrast, our approach avoids conventional buffers and does not require pre-sorting for handling very complex models.

A common acceleration technique for polygon rasterization are coverage masks [Fiume et al. 1983], which encode the pixel-wise inside condition with respect to a half plane in any possible orientation (typically for 8×8 pixel squares). Arbitrary convex polygons (and triangles in particular) can be processed by computing the intersection of the half planes associated with the edges, which is equivalent to a bit-wise AND operation of the corresponding coverage masks. In the literature, there are examples both for precomputed masks [Fiume et al. 1983; Greene 1996] and for on-the-fly computation [Eyles et al. 1988; Akeley 1993; Seiler et al. 2008].

2.3 Rendering on manycore GPUs

Recent trends indicate that graphics programming is rapidly moving away from fixed-function approaches towards general compute languages executing on manycore architectures [Hou et al. 2008]. Rather than redesigning existing techniques to fit the highly constrained execution environment found in conventional GPU programming using shader languages, software rendering methods can become competitive again by executing them on manycore GPUs.

A noteworthy example is the Larrabee manycore architecture [Seiler et al. 2008], which is based on multiple x86 cores and uses a flexible software renderer including a recursive polygon tiling algorithm [Greene 1996]. This approach is highly optimized towards Larrabee's hardware capabilities, in particular its vector processing units, which are used, e.g., for on-the-fly coverage mask computation.

Developing on GPUs with CUDA is not quite as flexible as Larabee's execution environment, but allows similar results to be achieved using mainstream graphics hardware. For convenience, we review the main aspects that influence our work. CUDA executes kernels written in C in a massively multi-threaded fashion on the GPU's multiple processing cores. Threads are grouped into blocks, which are executed with round-robin scheduling on a multi-processor consisting of several single processors and a limited amount (16k) of fast shared memory, which acts as a cache. Threads also have access to the GPU's global memory at a slower speed. Fully utilizing all processors while avoiding memory bottlenecks requires careful design of algorithm and parameters.

A brief report on ray casting single volumes with CUDA is given in [Marsalek et al. 2008], concentrating on the best choice of execution parameters for basic ray casting. Single volume ray casting with moving least squares was considered in [Ledergerber et al. 2008]. This method achieves very high quality reconstruction, but seems more appropriate for unstructured volumes. Ray casting of large deformable models composed of opaque polygons was presented in [Patidar and Narayanan 2008]. This work sorts all polygons into a view dependent 3D grid at every frame, and therefore focuses mainly on the sorting. To the best of our knowledge, our work is the first multi-volume ray caster developed specifically for the manycore execution model represented by CUDA.

3 System overview

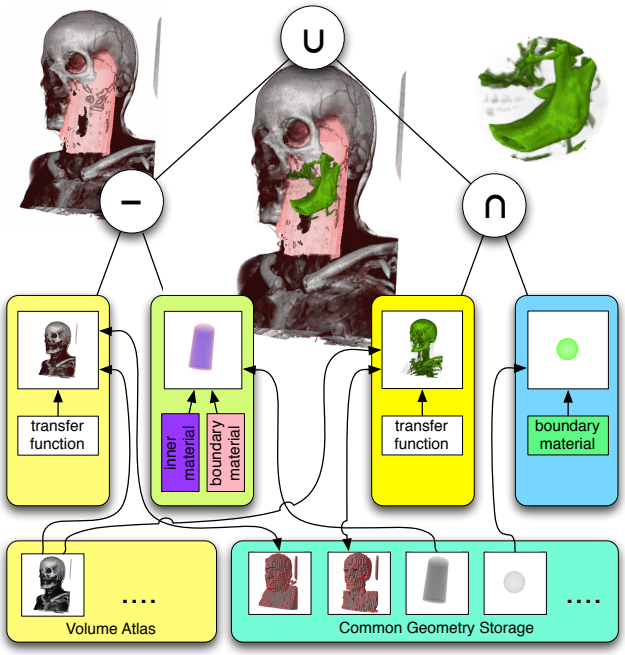


Figure 2: *The scene graph consists of CSG operations and polyhedral objects with volumetric textures and/or material properties. Textures are stored in a texture atlas, polygonal geometry and volume boundaries in common storage, which allows for instancing. Note that the two bounding volumes of the head in the geometry storage are computed from different transfer functions.*

Our system is able to produce ray casting images of complex intersecting volumetric datasets with polyhedral boundaries. All computation executes on the GPU, the only information that is received from the CPU for every frame is the camera position. The scene that represents the input to the ray casting procedure is structured in the following way (see Figure 2):

- Raw volume data from multiple volumes is represented in a volume atlas.
- Polyhedral objects consist of a triangular mesh. A polyhedral object can be concave, but must be two-manifold, so that interior and exterior can be distinguished.

- A scene consists of a simple scene graph. Interior nodes are associated with CSG operations, whereas leafs refer to a polyhedral object as the boundary, a volume texture or single material property for the interior, and an affine transformation for global placement of the object. The transformations can also be animated. Instances can be created by referencing the same polyhedral object and/or volume texture more than once.

The ray casting algorithm follows the usual steps of a rendering pipeline: transformation of polygons, rasterization, and fragment processing. The latter is done in a pixel-parallel way per 8×8 tile and is responsible for traversing adjacent rays through the volumes stored as 3D textures. It therefore naturally exploits texture cache coherence.

Unlike depth peeling approaches, our rendering system traverses the scene only once. We implement all steps in CUDA software and therefore have access to all intermediate results of the pipeline, which are kept in shared memory wherever possible for maximum performance. We still need two separate kernels (one for triangle processing and one for fragment processing) since the ray caster requires a sorted list of all relevant fragments at each pixel, which is only guaranteed to be complete after processing all triangles.

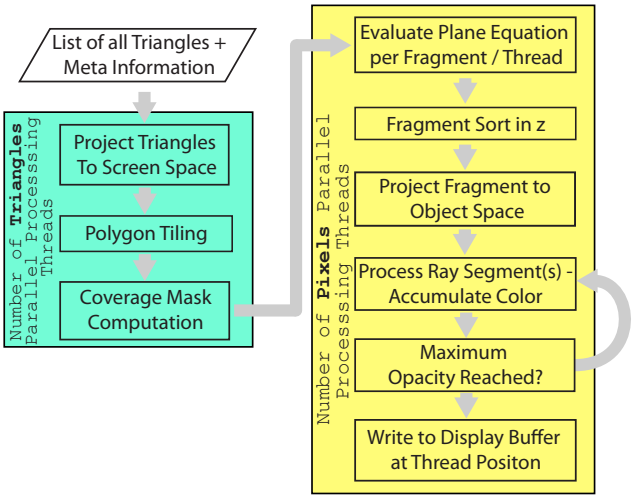
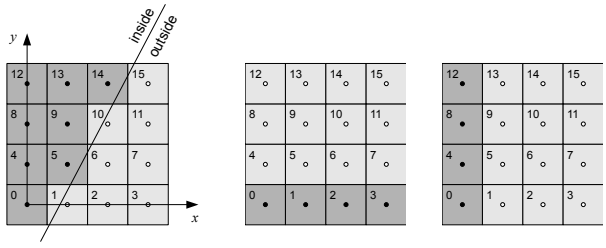


Figure 3: *The flow-chart shows an overview of our rendering system. The main steps - triangle processing and pixel processing - are executed in separate compute kernels.*

The triangle kernel is responsible for geometric transformations, assignment of triangles to viewport tiles, and computation of coverage masks (see Section 3.1). The pixel kernel receives the fragments covering a pixel in arbitrary order. They must be sorted in z-direction before they can be used to split a ray into homogeneous segments (see Section 3.2). The final ray casting and shading step requires the blending of several intersecting volumetric objects and therefore the efficient accumulation of the individual volumes' contributions along the rays (see Section 3.3). Support for concave bounding geometry simplifies empty space skipping (see Section 3.4). Volume samples are taken from a volume texture atlas (see Section 3.5). The following sections describe details of the individual steps of the system. For an overview see Figure 3.

3.1 Triangle rasterization

In conventional rendering of scenes dominated by opaque objects, high performance is related to being able to identify occluded portions of the scene early in the pipeline. In contrast, our volume



(a) coverage mask (b) unit row r_0 (c) unit column c_0

Figure 4: *On-the-fly half plane inclusion test for a 4×4 pixels square. The coverage mask (a) is composed of shifted copies of the unit row r_0 = 0x000F (b) or the unit column c_0 = 0x1111 (c). The numbers refer to bit positions in the coverage mask.*

rendering must unconditionally rasterize all polygons and pass information on the screen coverage of a polygon as quickly as possible to the pixel processing. After transforming a triangle into screen space, for each tile of 8×8 pixels intersecting the triangle's bounding box, a 64-bit coverage mask [Fiume et al. 1983] is computed. This simple strategy maps well to CUDA hardware due to its low resource requirements and outperforms more sophisticated approaches (such as [Seiler et al. 2008]) for typical scenes.

It is essential for our approach that every fragment covered by the triangulated volume boundary is visited exactly once since we initialize the ray segments for our ray caster at the rasterized surface fragments. Coverage masks are well suited for this purpose due to the simple bit-wise AND operation of the contributions of the corresponding half planes. However, precomputed coverage masks can not be used in this case since a particular mask is selected from the lookup table based on a discretized orientation of the half plane, which might result in a few misclassified pixels. We therefore compute the covered pixels for each triangle on-the-fly. Since the GPU cores in our target platform (NVIDIA GT200) are scalar processors, the method proposed by [Seiler et al. 2008] is not favorable in our case. Instead, we make use of bit shift operations to compute one row (or column) of the $N \times N$ pixel coverage mask in constant time.

Consider the example in Figure 4(a). The outside half plane is defined by $ax + by + c > 0$. A bit in the mask is set to one (dark squares in Figure 4(a)) if the corresponding pixel center (filled circle) is located inside the half plane, and otherwise set to zero. For plane coefficients $a > 0$ and $|a| > |b|$ (such as in this example), we compute the number $n_1(y)$ of "one" bits in a given row $y \in \mathbb{Z}$, which equals the number of corresponding pixel centers to the left of the line separating the inside and outside half planes:

$$x = -\frac{by + c}{a}, \quad n_1(y) = \text{clamp}(0, \lfloor x \rfloor, N), \tag{1}$$

where $\text{clamp}(i, j, k) = \max(i, \min(j, k))$. The bit mask $r(y)$ of a single row y can be obtained in constant time from the unit row mask r_0 (Figure 4(b)) by means of two bit shift operations:

$$r(y) = (r_0 >> [N - n_1(y)]) << [N \cdot y]), \tag{2}$$

where ">>" and "<<" denote the bit shift operation to the right and left, respectively. The entire coverage mask m is then

$$m = \sum_{y=0}^{N-1} r(y). \tag{3}$$

For $|a| < |b|$ we proceed in a similar way column-by-column, using the unit column mask c_0 in Figure 4(c) instead. The case $a \leq 0$ is handled by symmetry and bit-wise inversion.

Note that all constants in equations (1) and (2) can be precomputed, such that equation (3) can handle both the row-by-row and column-by-column case without conditional expressions to distinguish them. It is therefore possible to process triangles with arbitrary edge orientation in parallel in a SIMD-efficient way. On current NVIDIA hardware, this method is four times faster than a straightforward per-pixel computation of the mask as proposed by [Seiler et al. 2008].

The triangle kernel's completion ensures an application wide synchronization point before entering the pixel kernel. However, communication from the triangle kernel to the fragment kernel must use relatively slow global memory. Similar to the workflow presented by [Akeley 1993], we carry out computation of the coverage masks in the triangle kernel and not in the pixel kernel, because communicating a 64-bit coverage mask is cheaper than a full triangle description with 3 screen-transformed vertices.

Every triangle record contains the coverage mask and the id of the contributing triangle. No additional per-fragment information is stored in global memory, which makes the communication significantly more efficient than conventional buffer strategies [Mark and Proudfoot 2001; Bavoil et al. 2007]. The triangle records are organized in chunks of 64 records each, which are drawn from a pre-allocated pool of memory. When a tile receives its first triangle (or the previous chunk is fully occupied), a globally unique 32-bit chunk index is appended to the list of chunk indices associated with this tile. We choose a maximum of 64 chunks per tile, which allows for a total number of 4096 triangles per tile. Parallel creation of this data structure is synchronized among the worker threads of the triangle kernel using atomic functions.

3.2 Depth sorting

A block of threads responsible for a tile consists of 64 threads, one for each pixel in the tile. A thread's first task is to build a representation of the ray, consisting of the intersections with the triangles, sorted by depth. The thread iterates through the list of triangle records and checks its bit in the coverage mask. If the bit is set, it computes the depth of intersection from the triangle's plane equation. The z-values and triangle IDs are stored as array in fast shared memory. We therefore allocate only a single 32-bit value per entry, 16-bit for the signed z-value and 16-bit for the triangle ID. The entries in this array are interleaved to avoid banking conflicts.

Moreover, the available shared memory of almost 16 KB is subdivided into 64 slots, allowing every thread to store a maximum of 63 entries. This limits the maximum depth complexity to 63. The array is maintained in sorted order by inserting new values at the appropriate position, with those entries closest to the camera first (more complex sorting algorithms are likely to perform worse on such a small dataset). It is therefore guaranteed that the array contains the 63 closest fragments, no matter in which order they appear in the input. It turns out that the time spent for sorting is small compared to the subsequent ray traversal time, even for a specially designed worst-case scenario where the entire list of fragments has to be reversed (i.e., has quadratic time complexity in the number of fragments). Moreover, due to the spatial coherence in moderately complex scenes, the insertion operation is likely consistent across many threads. Fragments with rank 64 or higher in the sorted order are discarded. The rationale of this approach is that fragments occluded by 63 closer layers will have minimal influence on the final image, and can therefore be omitted.

To illustrate this claim, we have tested the approximations on objects with very high depth complexity. A ground truth implementation using larger but slower global memory rather than shared memory was used for comparison. The object presented in Figure 5,

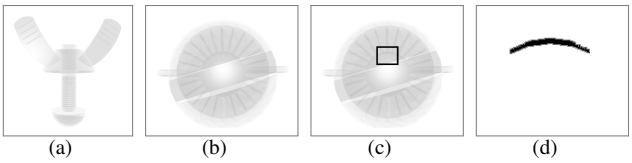


Figure 5: *(a) Thumbscrew (max. depth complexity of 81). (b) Image from our system (depth complexity limited to 55). (c) Reference image with full depth complexity (slower algorithm). (d) Enlarged differences. Fewer than 0.05% of the pixels show errors > 5% of the color range.*

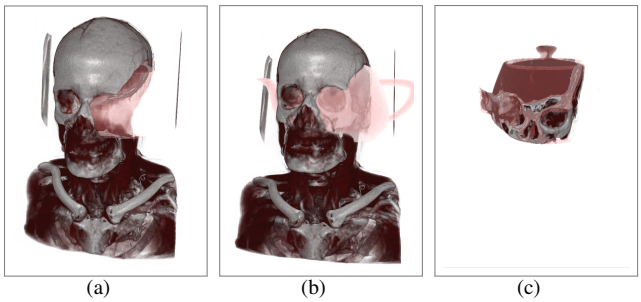
which was chosen for the highest depth complexity from our sample scenes, has a maximum depth complexity of 81 in the present view. Fewer than 0.05% of pixels exceed the threshold of a 5% difference between the fully correct and the approximated image (i.e. 12.75 gray levels of the overall color range 0-255). The contrast enhanced difference image in Figure 5 confirms this observation.

Allocating 16 bits for a triangle id limits the number of simultaneously visible triangles to 64k. We have found this to be an acceptable compromise, since the visual complexity in a scene is usually dominated by the volumetric objects. There are two ways to trade off other features for a larger number of visible triangles. We can forsake the ability to access a triangle's normal for surface shading (see Section 3.3). In this case the triangle id can be replaced by an object id, and we can still perform all volumetric shading effects and CSG operations. Alternatively, we can limit the maximum depth complexity to 32 layers, and use the additional 32-bit per depth entry for a quantized normal.

3.3 Ray casting of homogeneous segments

After sorting, a pixel thread iterates through the above-mentioned depth sorted array in order, from nearest to farthest. The depth interval between two consecutive entries in the array defines a homogeneous segment of the ray with respect to intersected objects. While traversing the array, we maintain information on the set of currently intersected scene objects, which are leafs of the CSG tree defining the scene. Figure 6 illustrates an example of the CSG operations enabled in this way.

An empty set of scene objects means the segment can be skipped. If one or more scene objects are intersected, the starting point of the ray has to be transformed into the model coordinates of each scene object, thus forming a set of object space rays. Each object space ray has to use the same step size in world coordinates to correctly blend the samples from each scene object. Since scene objects can have different size and resolution, we use the minimum step size of all objects along the ray segment to guarantee that no features are missed. Since this can lead to oversampling of objects with lower resolution, we provide an option to adjust this parameter. Objects which would require a high sampling rate might not provide more information at that rate, so we let the user balance between speed and accuracy. However, oversampling does not significantly influence performance due to texture cache coherence. For blending sampling points of different objects, we query the transfer functions separately and multiply the resulting colors with their transparency and a sampling factor which accounts for oversampling. The results are summed up and accumulated to the ray. For purely polyhedral objects made from a homogeneous material, no volume texture sampling is necessary. If a ray segment consists only of polyhedral objects, the loop can be avoided while still yielding high quality translucency. This can yield performance improvements of up to 10%. Once a ray segment has been sampled, the intermediate re-



(a) (b) (c)

Figure 6: *CSG operation examples: boolean difference (a), boolean union (b) and boolean intersection (c). Note that visualization of the boundary surface for highlighting cutting region(s) can be toggled by including or not including the boundary in the operation as shown in (a) and (c). In both cases the particular choice results in cutting surface being highlighted.*

sult is blended into the final result color for this pixel. If a certain opacity threshold is exceeded, the ray can be terminated.

Since polyhedral geometry plays an important role, both surface shading of the polygons and volumetric shading from 3D textures are combined. Optionally, conventional surface material parameters can be assigned to every scene object to add Phong shading and surface transparency effects at object boundaries.

3.4 Empty space skipping

Empty space skipping is an essential acceleration technique for volume rendering. In conventional hardware accelerated ray casting, exterior empty space skipping is typically done by calculating bounding geometries [Avila et al. 1992; Li et al. 2003]. The bounding geometry is rendered with vertex coordinates encoded as RGB values, so that the graphics hardware computes the ray entry and exit points for accumulation. Interior empty space skipping either requires a multi-pass ray casting approach similar to depth peeling or sampling of an additional lower resolution texture, which encodes empty and full bricks (sub-volumes) of a volume.

The efficient handling of bounding geometry in our rendering system allows exterior and interior space to be skipped in a uniform way. A lower resolution mask volume is calculated decomposing the original volume into uniform bricks, which are masked out if the transfer function evaluates to zero at each underlying dataset voxel. The brick size is user configurable per volume, for medical datasets a size of 8^3 resulted in maximum performance. The actual bounding geometry polygons enclosing full bricks are obtained using a traversal similar to a marching cubes algorithm [Lorensen and Cline 1987] of the mask volume, which avoids generation of duplicate polygons. Detection of empty bricks and the subsequent geometry calculation for a particular volume requires evaluation of the transfer function for all voxels in a volume and must be repeated every time the transfer function changes. For interactive transfer function editing, short calculation times are essential. This can easily be achieved by performing the calculation in parallel on the GPU, thereby avoiding additional slow data transfers from system memory. Computation times for a 512^3 dataset are around 5 milliseconds on an NVIDIA 280 GTX.

New bounding geometry polygons are inserted into the global triangle list in global GPU memory and associated with the relevant scene object. During ray traversal using the sorted fragment list, bounding geometry polygons allow for simultaneous exterior and interior space skipping based on a set of active objects, which is updated whenever a bounding geometry fragment is processed, adding or removing a particular object. Note that the set also contains in/outside information from all polygonal objects and is the basis for the evaluation of the CSG-tree. The algorithm may directly skip to the next fragment, whenever the set is empty and/or the CSG expression evaluates correspondingly.

The use of tight fitting bounding geometry improves frame rate by up to 15%, in particular for transfer functions which make large portions of a volume transparent. In general, the optimization affects both the triangle and the pixel processing kernel, since smaller triangles can be processed more efficiently than large ones in the triangle kernel, and the pixel kernel needs to process fewer ray segments, both in screen space and along the rays.

3.5 Volume texture atlas

To overcome the limited possibilities concerning reallocation of arrays in global memory, we use a 3D texture atlas for the volumetric datasets. A texture atlas is a single large 3D texture, which is assigned to one texture unit. The maximum size of the texture atlas depends only on the available graphics memory.

Finding an optimal memory layout is an instance of the cuboid packing problem [Huang and He 2009]. Since volume data sets often have a square cross section with a power-of-two edge length, we can (without significant waste of memory) reduce the packing problem to a single dimension, where the solution is trivial. The texture atlas memory is organized into slots of a certain size, depending on the application's needs. A volume can occupy one or several contiguous slots. The border voxels of every slot are duplicated to allow the use of unclamped texture coordinates in the innermost loop (this is not related to OpenGL texture borders, which facilitate seamless stitching of separate textures).

4 Results

We tested our implementation using CUDA 2.0 on a desktop PC (Intel 3.16 GHz Dual Core2-Duo with 3 GB RAM) running 32-bit Windows Vista and 64-bit Linux. We used two alternative GPUs, a GeForce GTX 280 with 1GB RAM and a Tesla C1060 Computing Processor with 4 GB RAM. The latter is able to work with volumes up to 1024^3.

4.1 Performance comparison

For performance evaluation, we compared our framework with two recently published state-of-the-art multi-volume renderers [Roessler et al. 2008; Brecheisen et al. 2008], which were kindly provided by the authors. In the following, we refer to these tools as *Roessler* and *Brecheisen*, and *Ours* for our own work. The comparison was done on the GTX 280.

C1 represents a high quality medical scan. C2 and C3 are medical scenes with multiple volumes, which use multiple modalities or scans from different body parts. Figure 1 shows C2 as third image. C4 and C5 show an anatomical scan involving a time resolved scan, for example of blood flow, leading to a larger number of volumes. C6 is a combination of an anatomical scan with a high resolution iso-surface. C7 is a volume of the abdomen combined with a 10k polygonal model representing the liver surface, a 30k polygon model representing the liver portal vessel tree and a 5k polygon model representing a tumor as shown as second image in Figure 1.

Table 1 compares the frame rates achieved with *Roessler* and *Brecheisen* and our approach. *Roessler* generates shader code for

No	Volumes	Polygons	Roessler	Brecheisen	Ours
C1	1×512^3	none	14 - 46	9 - 17	10 - 21
C2	2×256^3	none	24 - 38	6 - 12	17 - 25
C3	4×256^3	none	5 - 9	1 - 3	16 - 25
C4	1×256^3 7×128^3	none	1.5 - 3	n.p.	15 - 23
C5	1×256^3, 20×64^3	none	n.p.	n.p.	10 - 17
C6	1×512^3	$1 \times 30k$	n.p.	1 – 2	7 – 15
C7	1×512^3	$1 \times 30k$ $1 \times 10k$ $1 \times 5k$	n.p.	0 – 1	7 – 15
C8	50×64^3	$1 \times 30k$	n.p.	n.p.	8 – 16

Table 1: *Overview of test scenes used for performance comparison with state-of-the-art tools*

Scene	MaxDepth	DDP	Ours
S1	20	123 – 150	52 – 62
S2	40	14 – 16	18 – 28
S3	81	25 – 46	31 – 52

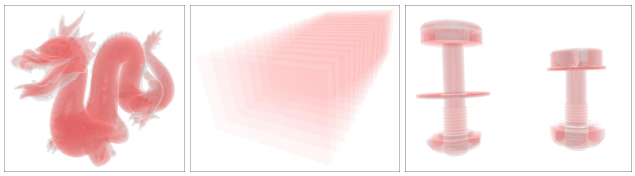
Table 2: *Comparing Dual Depth Peeling (DDP) to our approach for three scenes with high depth complexity (MaxDepth); screen size:* 1050×800*. In scenes with high and large-area depth complexity, our rendering system outperforms DDP.*

every scene and is therefore quite efficient for a few volumes. However, it cannot handle geometry and therefore cannot be used for cases C6 through C8. It could also not run C5 on our system since the generated shader code was too large. *Brecheisen* is able to handle up to four volumes combined with an arbitrary number of intersecting geometric objects, and therefore cannot handle C4, C5 or C8.

In our experiments, we varied viewport size (between 512^2 and 768^2), transfer functions and camera positions. Reported minimum and maximum frame rates are averaged over the tested viewport sizes, since the tools used for comparison do not allow arbitrary choices of viewport size. Lighting was done with pre-calculated gradients, since *Roessler* only provides this method. Progressive rendering was disabled in all evaluated tools. The tests were performed with 8-bit value per sample datasets due to the restrictions of *Roessler* and *Brecheisen*. A transfer function with high transparency was chosen to avoid early ray termination. Camera zoom was set to a value such that most screen pixels were covered. Table 1 indicates that our approach compares favorable with shader based systems in particular for scenes with many volumes or complex geometry.

While not its main objective, our system can also render correct transparency in scenes with high geometric depth complexity. We compared this capability to NVIDIA's Dual Depth Peeling (*DDP*) demo [Bavoil and Myers 2008]. We used three scenes with different depth complexity and memory bus consumption for the geometry passes. Table 2 confirms our expectation that it can outperform depth peeling for more complex scenes. Note that unlike *DDP*, we use high quality per-pixel Phong shading. Figure 7 shows the test scenes from a viewpoint with average depth complexity. S1 contains the Stanford dragon reduced to 20k polygons. S2 contains 20 boxes in a row. S3 contains screws with polygonal winding.

Our system scales very well when the geometry is filled with a volumetric texture or when complex objects intersect. Table 2 supports this observation.



(a) **Scene S1** (b) **Scene S2** (c) **Scene S3**

Figure 7: *Scenes with a high depth complexity for a comparison with the NVIDIA Dual Depth Peeling Demo [Bavoil and Myers 2008].*

4.2 Workload distribution

In order to better understand our rendering system, an analysis of the workload for various conditions was undertaken. The triangle kernel uses only 2-10% of the computation time, depending on the triangle/voxel ratio of the scene. The NVIDIA Visual Profiler revealed that the triangle kernel is able to efficiently occupy all processors of the GPU, thus optimally hiding global memory latency. Moreover, projecting triangles and computing coverage masks are very uniform tasks with few diverging branches, which allows peak rates to be obtained. Overall, the pixel kernel uses up to 90% of the computation time. Since kernels are executed in threads on the GPU asynchronously with respect to the CPU, the GPU code was instrumented using the *clock()* API function to query the number of GPU cycles. Unfortunately, this function does not report the time spent per thread, but rather the clock of the processor, while the processor performs time slicing of multiple threads. Low consumption of register space and shared memory allows more threads to be active (i.e., time-sliced) concurrently and therefore better hiding of memory latency. This improves performance, but at the same time makes clock measurements unreliable.

We therefore experimented with both low and heavy loads on shared memory. A low load can be achieved by limiting the maximum complexity of the scene. For example, lowering the maximum depth complexity or the number of intersecting objects frees register space and shared memory per execution block, and therefore enables the GPU scheduler to run other blocks while waiting for memory transfers. Through this strategy, we could obtain overall performance improvements of up to 40%. We also selected setups which create a heavy load on shared memory, giving no opportunity for time slicing. This allowed us to perform the following analysis.

For scenes with significant geometry, but relatively little volume data, (see leftmost picture in Figure 1) the results indicated that up to 50% of the pixel kernel's work consists of sorting fragments in z-direction. The rest consists of Phong shading of geometry surfaces (<3%), accumulating pure geometry interiors (<3%) and ray traversal of volumetric data (23%).

As an example for a scene with heavy volumetric load, consider the head/brain scene from the third picture in Figure 1. In this example, computing and sorting depth values amounts to 7% of computations, while ray traversal takes up to 60%, which can be split into 12% for ray setup and 48% for the accumulation loop. Tests with various scenes confirmed that ray traversal is generally the most costly part. This is reassuring because it seems likely that the $O(n^3)$ complexity for handling the volumetric data should dominate the system's performance. However, it also means there is limited room left for improvements.

4.3 Memory consumption

The memory requirements of the described method are mainly related to the texture atlas and the triangle records used to communicate between triangle kernel and pixel kernel. The texture atlas size depends on cumulated sizes of the volumes contained in the scene. We commonly use a texture atlas of 528^3, which can accommodate all the scenes used in the evaluation, in about 562 MB of global memory. Pre-calculation of gradients for lighting increases the required memory by a factor of four, because for every sample three additional values need to be stored.

The triangle record memory consumption for a viewport size of 768^2 pixels is 96^2 (tiles) $\times$ 64 (chunks per tile) $\times$ 4 (bytes per index) = 2.25MB for the fixed-size set of chunk indices, plus the variable-sized chunk pool (768 bytes per chunk). Our experiments were run with a total chunk pool size of 6MB, which can be raised for more complex scenes up to the limit imposed by the available amount of memory on the graphics card.

5 Conclusion

Due to recent improvements in multicore GPU programming models, we were able to develop a new approach for multi-volume and geometry rendering. CUDA is not specifically designed for graphics, but allows the development of a software rendering pipeline, which is executed in a massively parallel way. A key property of this approach is the efficient use of scatter operations, which allows a scene's polygons to be rasterized only once.

This approach is relevant to a broad range of applications. It increases the practical applicability of volume rendering to very complex scenes, including intersecting volumes and geometry in clinical practice or in computer games and art. As shown in this paper, our system is easy to implement, robust, and readily runs on consumer hardware. Since our implementation is designed for many-core graphics hardware, we are looking forward to future general purpose GPU architectures such as Larrabee [Seiler et al. 2008], which should significantly increase the flexibility of the algorithms and enable new design choices.

Acknowledgements

We would like to thank Friedmann Roessler (University of Stuttgart), Ralph Brecheisen (Eindhoven University of Technology) and Stefan Bruckner (Vienna University of Technology) for providing their rendering systems, and Joseph Newman for many useful remarks.

This work was funded by the European Union in FP7 VPH initiative under contract number 223877, the Ludwig Boltzmann Institute for Clinical-Forensic Imaging, and the FWF under contract W12009-N15.

References

Aila, T., Miettinen, V., and Nordlund, P. 2003. Delay streams for graphics hardware. *ACM Transactions on Graphics (TOG) 22*, 3, 792–800.

Akeley, K. 1993. Reality engine graphics. In *Proceedings of 20th annual conference on Computer graphics and interactive techniques SIGGRAPH '93*, ACM, New York, NY, USA, 109–116.

Avila, R. S., Sobierajski, L. M., and Kaufman, A. E. 1992. Towards a comprehensive volume visualization system. In *Proceedings of IEEE Visualization '92*, 13–20.

Bavoil, L., and Myers, K. 2008. Order independent transparency with dual depth peeling. Tech. rep., NVIDIA.

Bavoil, L., Callahan, S. P., Lefohn, A., Comba, J. L. D., and Silva, C. T. 2007. Multi-fragment effects on the GPU using the k-buffer. In *Proceedings of ACM symposium on interactive 3D graphics and games*, 97–104.

Borland, D., Clarke, J., Fielding, J., and Taylor, R. 2006. Volumetric depth peeling for medical image display. In *Proceedings of SPIE Visualization and Data Analysis*, 1–11.

Brecheisen, R., Platel, B., Vilanova, A., and ter Haar Romenij, B. 2008. Flexible GPU-based multi-volume ray-casting. In *Proceedings of Vision, Modelling and Visualization*, 1–6.

Bruckner, S., Grimm, S., and Kanitsar, A. 2006. Illustrative Context-Preserving exploration of volume data. *IEEE Transactions on Visualization and Computer Graphics 12*, 6, 1559–1569.

Callahan, S. P., and Comba, J. L. D. 2005. Hardware-assisted visibility sorting for unstructured volume rendering. *IEEE Transactions on Visualization and Computer Graphics 11*, 3, 285–295.

Carpenter, L. 1984. The A-buffer, an antialiased hidden surface method. *ACM SIGGRAPH Computer Graphics 18*, 3, 103–108.

Carr, N., Mech, R., and Miller, G. 2008. Coherent layer peeling for transparent high-depth-complexity scenes. In *Proceedings of SIGGRAPH/EUROGRAPHICS symposium on graphics hardware*, 33–40.

Engel, K., Hadwiger, M., Kniss, J., Rezk-Salama, C., and Weiskopf, D. 2006. *Real-time Volume Graphics*. A. K. Peters.

Everitt, C. 2001. Interactive order-independent transparency. Tech. rep., NVIDIA.

Eyles, J., Austin, J., Fuchs, H., Greer, T., and Poulton, J. 1988. Pixel-planes 4: A summary. In *Advances in Computer Graphics Hardware II (Eurographics'87 Workshop)*, Springer-Verlag, London, UK, 183–207.

Fiume, E., Fournier, A., and Rudolph, L. 1983. A parallel scan conversion algorithm with anti-aliasing for a general-purpose ultracomputer. In *ACM SIGGRAPH Comp. Graph.*, 141–150.

Greene. 1996. Hierarchical polygon tiling with coverage masks. In *ACM SIGGRAPH Computer Graphics*, 65–74.

Grimm, S., Bruckner, S., Kanitsar, A., and Gröller, M. E. 2004. Flexible direct multi-volume rendering in interactive scenes. In *Proceedings of Vision, Modeling, and Visualization*, 386–379.

Hou, Q., Zhou, K., and Guo, B. 2008. BSGP: bulk-synchronous GPU programming. *ACM Transactions on Graphics (TOG) 27*, 3, 19.

Huang, W., and He, K. 2009. A new heuristic algorithm for cuboids packing with no orientation constraints. *Computers & Operations Research 36*, 2, 425–432.

Krüger, J., and Westermann, R. 2003. Acceleration techniques for GPU-based volume rendering. In *Proceedings of IEEE Visualization*, 287–292.

Ledergerber, C., Guennebaud, G., Meyer, M., Baecher, M., and Pfister, H. 2008. Volume MLS ray casting. *IEEE Transactions on Visualization and Computer Graphics 14*, 6, 1372–1379.

Li, W., Mueller, K., and Kaufman, A. 2003. Empty space skipping and occlusion clipping for texture-based volume rendering. In *Proceedings of IEEE Visualization*, 317–324.

LIU, B., WEI, L.-Y., AND XU, Y.-Q. 2006. Multi-layer depth peeling via fragment sort. Tech. Rep. MSR-TR-2006-81, Microsoft Research Asia.

LORENSEN, W. E., AND CLINE, H. E. 1987. Marching cubes: A high resolution 3D surface construction algorithm. *ACM SIGGRAPH Computer Graphics 21*, 4, 163–169.

MARK, W. R., AND PROUDFOOT, K. 2001. The F-buffer: a rasterization-order FIFO buffer for multi-pass rendering. In *Proceedings of SIGGRAPH/EUROGRAPHICS workshop on graphics hardware*, 57–64.

MARSALEK, L., HAUBER, A., AND SLUSALLEK, P. 2008. High-speed volume ray casting with CUDA. In *Proceedings of IEEE Symposium on Interactive Ray Tracing*, 185.

MOLNAR, S., COX, M., ELLSWORTH, D., AND FUCHS, H. 1994. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications 14*, 4, 23–32.

NADEAU, D. R. 2000. Volume scene graphs. In *Proceedings of IEEE symposium on volume visualization*, 49–56.

NICKOLLS, J., BUCK, I., AND GARLAND, M. 2008. Scalable parallel programming with CUDA. *ACM Queue 6*, 2, 40–53.

NVIDIA. 2008. *NVIDIA CUDA Programming Guide 2.0*. NVIDIA Corporation.

PATIDAR, S., AND NARAYANAN, P. J. 2008. Ray casting deformable models on the GPU. In *6th Indian Conference on Computer Vision, Graphics & Image Processing*, 481–488.

PLATE, J., HOLTKAEMPER, T., AND FROEHLICH, B. 2007. A flexible multi-volume shader framework for arbitrarily intersecting multi-resolution datasets. *IEEE Transactions on Visualization and Computer Graphics 13*, 6, 1584–1591.

ROESSLER, F., BOTCHEN, R. P., AND ERTL, T. 2008. Dynamic shader generation for GPU-based multi-volume ray casting. *IEEE Computer Graphics and Appllications 28*, 5, 66–77.

ROETTGER, S., GUTHE, S., WEISKOPF, D., AND ERTL, T. 2003. Smart Hardware-Accelerated Volume Rendering. In *Proceedings of EG/IEEE TCVG Symposium on Visualization VisSym '03*, 231–238.

SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., AND HANRAHAN, P. 2008. Larrabee: a many-core x86 architecture for visual computing. *ACM Transactions on Graphics (TOG) 27*, 3, 18.

STEGMAIER, S., STRENGERT, M., KLEIN, T., AND ERTL, T. 2005. A simple and flexible volume rendering framework for graphics-hardware-based raycasting. In *Proceedings of International Workshop on Volume Graphics*, 187–241.

TERMEER, M., BESCÓS, J. O., AND TELEA, A. 2006. Preserving sharp edges with volume clipping. In *Proceedings of Vision, Modeling and Visualization*, 341–348.

WEISKOPF, D., ENGEL, K., AND ERTL, T. 2003. Interactive clipping techniques for texture-based volume visualization and volume shading. *IEEE Transactions on Visualization and Computer Graphics 9*, 3, 298–312.

WEXLER, D., GRITZ, L., ENDERTON, E., AND RICE, J. 2005. GPU-accelerated high-quality hidden surface removal. In *Proceedings of SIGGRAPH/EUROGRAPHICS conference on graphics hardware*, 7–14.