https://doi.org/10.2352/ISSN.2470-1173.2017.1.VDA-392

Ray Traced Volume Clipping Using Multi-Hit BVH Traversal

Stefan Zellmann; University of Cologne, Chair of Computer Science, Cologne, Germany
Mauritius Hoevels; University Hospital of Cologne, Department of Stereotaxy and Functional Neurosurgery, Cologne, Germany
Ulrich Lang; University of Cologne, Chair of Computer Science, Cologne, Germany

Abstract

Clipping is an important operation in the context of direct volume rendering to gain an understanding of the inner structures of scientific datasets. Rendering systems often only support volume clipping with geometry types that can be described in a parametric form, or they employ costly multi-pass GPU approaches. We present a SIMD-friendly clipping algorithm for ray traced direct volume rendering that is compatible with arbitrary geometric surface primitives ranging from mere planes over quadric surfaces such as spheres to general triangle meshes. By using a generic programming approach, our algorithm is in general not even limited to triangle or quadric primitives. Ray tracing complex geometric objects with a high primitive count requires the use of acceleration data structures. Our algorithm is based on the multi-hit query for traversing bounding volume hierarchies with rays. We provide efficient CPU and GPU implementations and present performance results.

Introduction

Clipping for 3-D direct volume rendering (DVR) plays an important role in many scientific visualization contexts. Clipping with spheres and planes can be a helpful tool in interactive scenarios as they occur in virtual reality (VR) applications with tracking devices. In such cases, volumetric datasets can be interactively explored by navigating through the rendered volumetric region and using the clip geometry as a virtual shield. In medical imaging contexts, static clipping with a nonmoving geometry is important e.g. in cases where neurologists have identified 3-D regions of interest in a magnetic resonance imaging (MRI) dataset and desire to suppress rendering for all content but that inside of those regions. Engineering applications often rely on DVR to display scalar or higher-order fields obtained from a simulation. For visualizations of this type it may be helpful to perform sub-voxel accurate clipping with the bounding geometry that was used during the simulation, especially if the volume dataset is blended and then displayed together with the bounding geometry.

Traditional DVR applications, especially if they are intended to be used in VR scenarios and if low latency is crucial, typically employ hardware accelerated texture-based rendering with rasterization and a planar proxy geometry. Using this traditional pipeline, sub-voxel accurate clipping is hard to achieve with geometry that exposes irregular concavities. When using ray casting for volume integration, clipping with an arbitrarily shaped geometry can however elegantly be described in terms of a simple two-pass algorithm: in the first pass, intersect each primary viewing ray with all opaque geometry that is set up for clipping and identify visible volume regions. In the second rendering pass, cast primary rays through the volume density and consider only those voxels that are not clipped. A ray tracing-based algorithm lends itself well to this approach because intersecting rays with 3-D geometry can be efficiently implemented in a GPGPU program. In order to determine the intersection information with a traditional rasterization pipeline, one would have to employ multiple render passes just to identify the visible and invisible regions - one would have to peel away consecutive depth layers as it is e.g. done to render translucent surfaces in viewing order [3] and then construct the visibility information from multiple render targets. On top of being hard to implement elegantly and being potentially inefficient, this approach would also require that parametric surfaces such as quadrics were rasterized and thus converted to polygon meshes in advance.

Real-time surface ray tracing has in recent years been successfully ported to GPUs using general purpose GPU programming (GPGPU). Because of its inherent inefficiency - the time complexity of ray tracing grows with the product of the number of primitives and the number of image pixels - a divide and conquer strategy is mandatory for real-time ray tracing. This strategy is typically implemented by a priori calculating a hierarchical spatial or object subdivision to reduce the number of necessary ray / geometric object interactions in favor of testing the ray against bounding objects surrounding a whole set of primitives. With this kind of data structure it is possible to reduce the asymptotic behavior of surface ray tracing so that it is on average logarithmic in the number of geometric primitives. In order to combine DVR and clipping with an arbitrary surface geometry, it is necessary to integrate a ray tracing acceleration data structure into the DVR application.

In this paper we present an efficient approach for DVR and clipping with an arbitrary surface geometry. Our technique allows, amongst others, for clipping with triangle meshes that exhibit multiple concavities. For this and in order to be able to manage large triangle meshes efficiently, we employ a bounding volume hierarchy (BVH) and the multi-hit ray traversal query to efficiently identify volume regions that are supposed to be clipped. We provide an implementation that is optimized for both SIMD x86 instructions and for NVIDIA GPUs and that is available as part of an open source DVR software.

The paper is organized as follows. In the following section we motivate the benefit of sub-voxel accurate clipping for neurosurgical visualization. In the section after that we review related work from the fields of ray tracing and volume rendering with clipping. Then we present a general framework to efficiently implement clipping with possibly concave geometric objects in a volume ray caster. We extend this approach by explicitly specializing it for triangle meshes and then present an implementation that can run on both CPUs and GPUs. In the section after that we propose a neurosurgical visualization aided by our method and present performance results that we discuss afterwards. The last

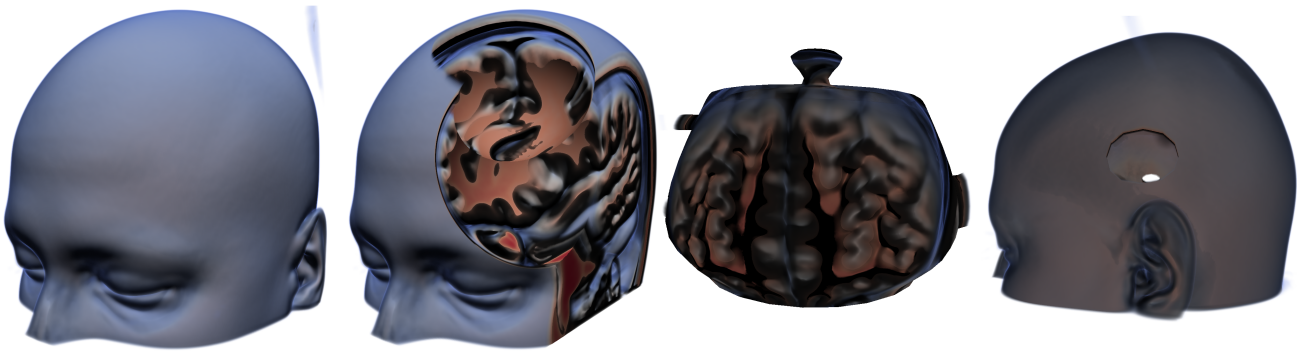

Figure 1. *Various clipping scenarios supported by our algorithm. The image on the left shows the MRI volume data set without clipping. The second left most image shows clipping with two spheres and a plane, where all surfaces overlap. The third image shows inverse clipping with a complex triangle geometry and the fourth image shows our algorithm using the same geometry, but with ordinary clipping so that almost the whole interior of the MRI data set is excluded from rendering.*

section briefly concludes this publication.

Use Cases for Sub-Voxel Accurate Volume Clipping

In minimal invasive neurosurgical (stereotactic) treatment the planning process performed on medical workstations is an essential and decisive part of the procedure. Currently available treatment planning systems provide visualization options in different 2-D views, but offer only limited 3-D support. The planning process could benefit from additional visualization options in the following concerns.

Inspection and visualization of tumor border: Any kind of surgical or radiological treatment can lead to sufficient results only if the region to be treated can be precisely defined. Thus the delineation of the target volume is an important step within the planning process. Usually it takes place in a couple of 2-D sections, and further inspection again happens in orthogonal 2-D sections or sections parallel or rectangular to the instrumentation (surgeon's eye view). By using sub-voxel accurate clipping the border of the tumor could be projected onto a canvas and be presented to the user as a map. Irregular signal patterns such as signal enhancement or reduction, or the change of texture within this map may indicate inappropriate delineation and could suggest to revisit the definition in the location in question. This approach could potentially speed up the definition process and enhance the quality of the resulting volume outline.

Vessel detection: In stereotactical surgical procedures hitting vessels must be avoided, and detecting and avoiding vessels is a crucial and time consuming task. Often the outer part of the tumor – especially in metastases – exhibits the highest metabolism rate and thus exposes a very intensive signal. It can be envisioned as a bright rim surrounding the tumor volume and might hide the presence of small vessels close to the tumor (cf. Figure 2). These vessels could be visualized by unrolling the tumor border that was clipped from the volume onto a canvas and show up as straight or slightly curved continuous objects.

Intersections: Alongside the surgical path the surface of the surgical instrumentation (radius from 1.3 to 2.5 mm) can be represented as a cylinder surface and be projected onto a canvas. In this projection vessels that reside in the path might be visible as small areas or points with enhanced signal value. Intersection with different objects, in particular fiber tracks can quantify potential risk or benefit of the neuromodulation of neurological structures provided by the electrode in place (cf. Figure 2). Finally the tumor surface intersection with a given dose distribution uncovers both overdose and underdose regions of the tumor.

Related Work

A popular optical model for DVR in scientific visualization applications is the *emission plus absorption* model [21], which does not account for scattering phenomena but, because of its simplicity, is the basis for many real-time applications [26, 22, 8]. Common DVR algorithms that implement the emission plus absorption model include hardware accelerated, texture-based slicing [35] and ray casting [19, 16]. The ray casting algorithm to evaluate the emission plus absorption model is usually based on piecewise integration with a uniform step size, so that the memory access patterns of typical implementations can be expected to be coherent because neighboring rays are likely to encounter the same data items when traversing the volume density. Ray tracing algorithms that exhibit this type of memory access pat-

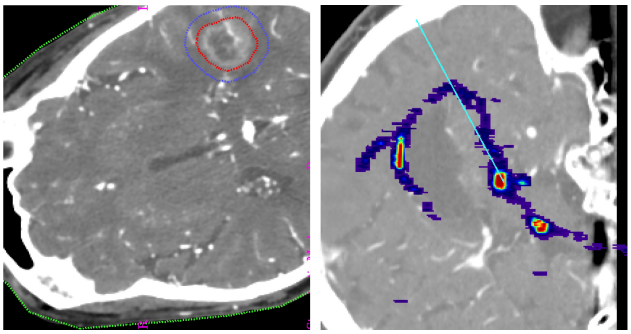

Figure 2. *Use cases where sub-voxel accurate volume clipping may enrich the operation planning process. On the image to the left, a tumor was delineated by the neurosurgeon before the operation. By clipping the border surface from the volume and projecting it to 2-D, the appropriateness of the delineation could be reviewed faster. The image to the right shows the surgical path that is intersected with a fiber track. Intersections could be projected to the surface of the cylinder representing the surgical path and would be easier to identify than in conventional 2-D planning systems.*

tern can benefit from coherent ray packet approaches [29] which perform best when the ray packet size is aligned with the width of the single-instruction multiple data (SIMD) registers found in modern processors [32]. Highly optimized CPU DVR ray casting implementations thus employ coherent ray packet traversal [13].

Volume rendering with clip planes for navigation and cutting has in the past been argued to be an effective tool in the field of medical imaging and especially for radiation treatment planning [20, 33]. User interfaces have been proposed to incorporate volume clipping in VR volume rendering applications for interaction [14] and in medical imaging applications to perform quantitative measurements [6].

Weiskopf et al. [34] have investigated volume clipping in the context of texture-based slicing in 2002. Their algorithm proceeds by consecutively rendering layers of the opaque clip geometry to the hardware depth buffer in multiple render passes and then performing clip operations on the currently active layer in a fragment program. This approach is similar in nature to the depth peeling approach [3] but alleviates the need for intermediate high-precision GPU storage such as framebuffer objects or 32-bit floating point textures, which were sparse GPU resources at that time. The authors compared their method to an approach where clip regions were precalculated and stored in an additional index texture having the same resolution as the volume texture, with a binary index indicating if the region was visible or not. The authors emphasized the increased quality obtained from sub-voxel accurate clipping over the index texture approach, but also concluded that "Depth-based clipping, however, is slower by a factor 3-4 (compared to rendering without clipping)".

Ray tracing of surfaces in contrast to volume ray casting is typically based on first building an index data structure that hierarchically groups the surfaces to be intersected based on spatial proximity. While the principle of finding bounding volumes around a set of neighboring primitives that are faster to intersect than the whole set of primitives is always the same, acceleration data structures differ in that some of them partition space and others partition the set of primitives. k-d trees [4] fall in the former and BVHs [25] in the latter category. BVHs have in recent years earned more attention from the scientific community than k-d trees because they can be used to accommodate dynamic and even fully deformable scenes [30]. Research in the recent years has concentrated on fast ray / BVH traversal on GPUs [1] and with incoherent rays [7], as well as on fast BVH construction on CPUs and GPUs [31, 18]. It has been shown that meshes with an uneven distribution of polygons with large and tiny areas can be efficiently traversed by incorporating spatial splits in the BVH construction scheme [27].

Ray tracing algorithms for image generation are often recursive in nature [36, 12], so that typical ray traversal queries such as finding the intersection closest to the ray origin, or finding any intersection with regard to a list of geometric objects, typically yield only a single result instead of a list of results to iterate over. Only recently have publications emerged that propose the *multi-hit* ray traversal query [10], which returns a (typically fixed-size) list of intersections sorted by distance (or provide a callback mechanism to access elements of this list), where the set of geometric objects the query is performed upon may e.g. be organized using BVHs [2, 11].

Volume Ray Casting and Clipping

We propose a general volume clipping algorithm that allows for arbitrarily many overlapping clip regions with arbitrary geometry types. For that, we employ a two-pass ray tracing approach. In a first pass, we generate primary viewing rays from the viewing position and intersect them with the axis-aligned bounding box of the volume data set and with a potentially populated depth buffer from a previous render pass to support interoperability e.g. with OpenGL rendering of opaque surfaces. We intersect the viewing rays that have passed this first visibility test with the clip objects that are active for the current frame in order to assemble a set of clip intervals, which cover regions where the volume rendering integral should not be evaluated. In a second pass we integrate over the volume density with respect to the clip intervals. In an actual implementation, the two passes can be combined in a single compute kernel, so that the viewing rays do not need to be regenerated.

1st Pass: Assembling Clip Intervals

In order to build up the set of clip intervals C for each viewing ray, we first determine t_{near} and t_{far}, the distances from the ray's origin to the nearest and farthest intersection position with the volume's bounding box. We also consider a potential previous render pass with opaque planar geometry and thus also test against a depth buffer that is obtained from the GPU rendering API and then transferred to the volume rendering function. There the individual depth buffer entries are converted to the volume coordinate system using the transformation that is outlined in [14] and are then transformed to ray parameter space using simple vector operations to obtain t_d. We then assign $t_{near} := max(t_d, t_{near})$ and $t_{max} := min(t_d, t_{far})$, which also remain valid for the rendering pass following clip interval assembly. If the test against t_{near} and t_{far} yields a valid intersection, we assemble C by intersecting the viewing ray with the active clip geometry. For each active clip object, let, without loss of generality,

$$1 \leq \ldots \leq i-1 < i < i+1 < i+2 \leq \ldots \leq M \tag{1}$$

and

$$t_1 \leq \ldots \leq t_{i-1} \leq t_i \leq t_{i+1} \leq t_{i+2} <\leq \ldots \leq t_M. \tag{2}$$

The variables from Equation 1 denote indices over M intersections, while the variables defined in Equation 2 denote distances from the origin of the viewing ray to the respective intersection position. In the case that is implied by Equations 1 and 2, where the viewing ray that is intersected with the clip object hits at least one concavity, in order to build up a set of clip intervals we consider the *geometric* normal of the surface at the intersection positions to determine which two intersection distances form a valid pair. Let V be the normalized direction vector of the viewing ray and $\{N_{i-1}, N_i, N_{i+1}, N_{i+2}\}$ the geometric normals. We then construct the set of pairs of consecutive intersection distances which form valid clip intervals

$$L = L \cup L_j, \tag{3}$$

where

$$L_j \begin{cases} [t_{i-1},t_i],[t_{i+1},t_{i+2}] & V\cdot N_{i-1} > 0 \\ [-\infty,t_{i-1}],[t_i,t_{i+1}] & V\cdot N_{i-1} \leq 0 \wedge i = 2 \\ [t_i,t_{i+1}],[t_{i+2},\infty] & V\cdot N_{i+2} \leq 0 \wedge i+2 = M \\ [t_i,t_{i+1}] & \text{otherwise} \end{cases} \tag{4}$$

by considering each intersection position with an even index $i \in \{2,4,6,\dots\}$. The special cases in Equation 4 where $i = 2$ or $i + 2 = M$ occur if either the first or the last clip interval lies only halfway inside the volume density. We need to determine if we add indices to L that start with an odd or even intersection index i to get the interval order right. The four possible cases we need to consider are illustrated in Figure 3. We assume that a single clip object may consist of one or many non-overlapping closed surfaces, which implies that either

$$V\cdot N_{i-1} > 0 \wedge V\cdot N_i \leq 0 \wedge V\cdot N_{i+1} > 0 \wedge N_{i+2} \leq 0, \tag{5}$$

or

$$V\cdot N_{i-1} \leq 0 \wedge V\cdot N_i > 0 \wedge V\cdot N_{i+1} \leq 0 \wedge N_{i+2} > 0. \tag{6}$$

In the case that the intersection with the surface may yield only a single intersection, e.g. if we intersect the volume with a plane, Equation 4 is not applicable. We handle this case separately by appending either $[-\infty,t_i]$ or $[t_i,\infty]$ to L (i in this case is the index of the single intersection). We proceed similarly with the case where up to two intersections may occur, e.g. when intersecting the volume with a sphere.
Note that by changing the direction of the comparisons in Equation 4:

$$L_j \begin{cases} [t_{i-1},t_i],[t_{i+1},t_{i+2}] & V\cdot N_{i-1} \leq 0 \\ [-\infty,t_{i-1}],[t_i,t_{i+1}] & V\cdot N_{i-1} > 0 \wedge i = 2 \\ [t_i,t_{i+1}],[t_{i+2},\infty] & V\cdot N_{i+2} > 0 \wedge i+2 = M \\ [t_i,t_{i+1}] & \text{otherwise} \end{cases}, \tag{7}$$

we can easily alter the behavior of the clipping operation so that either the volume density on the inside or on the outside of the surface is clipped. This is the primary reason why we chose to base the construction of clip intervals on the geometric normal of the clip surface.
Note further that the determination of the clipping order based on the angle between geometric normals and the viewing direction alleviates the problem of having to keep track of the intersection order when constructing L. This allows for an efficient implementation using coherent packet traversal, where the intersection count will in general differ per SIMD lane.
From the set L of clip intervals per geometry, we construct the set C by clipping each interval $[t_m,t_{m+1}] \in L$ with the volume boundaries t_{near} and t_{far}

$$C_{kl} = \begin{cases} [t_{near},t_{far}] & t_m \leq t_{near} \wedge t_{m+1} \geq t_{far} \\ [t_{near},t_{m+1}] & t_m \leq t_{near} \wedge t_{m+1} < t_{far} \\ [t_m,t_{far}] & t_m > t_{near} \wedge t_{m+1} \geq t_{far} \\ [t_m,t_{m+1}] & t_m > t_{near} \wedge t_{m+1} < t_{far} \\ [\infty,-\infty] & \text{otherwise} \end{cases} \tag{8}$$

and appending it to the set C:

$$C = \bigcup_{k=1}^{N} \bigcup_{l=1}^{O_k} C_{kl}. \tag{9}$$

N here denotes the number of active clip objects and O_k the (variable) number of intervals per clip object. In an actual implementation, we will only append valid clip intervals to C and will neglect intervals $[\infty,-\infty]$, since those correspond to pairs of intersections beyond the volume boundaries. Note that we need to separately assemble the sets C and L because we in fact do not support multiple overlapping surfaces inside a single clip object, but however allow for overlap between several clip objects. In that case, Equations 5 and 6 will not hold true.
Along with the clip intervals we store an additional list containing the surface normals pertaining to the respective interval. Out of two intersection positions, we choose the (single) normal that points into the opposite viewing direction.
With this clip interval construction scheme we can accommodate arbitrary clip surfaces with multiple concavities, as far as they are closed. The first algorithm pass leaves us with a set of potentially overlapping intervals that we consider for clipping during rendering.

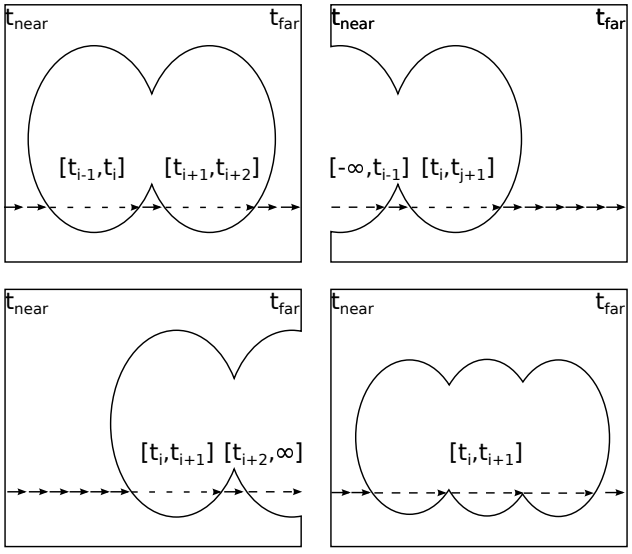

Figure 3. *When evaluating M intersection positions for each clip object along the direction of a viewing ray that is marched through the volume, we need to decide if we construct new clip intervals starting at an odd or even intersection index i. In the general case, where we have a surface without holes and with one or more concavities, the cases depicted in the figure and outlined in Equation 4 are to be considered.*

2nd Pass: Rendering

During the rendering pass we evaluate the volume rendering integral using ray marching with front-to-back compositing and piecewise integration with a uniform step size and limits t_{near} and t_{far}. At each integration step, we calculate the distance from the

ray origin t. We then trivially project t onto each interval $C_i = [t_l, t_m] \in C, t_l \leq t_m$. If

$$t_l \leq t \leq t_m, \tag{10}$$

we assign $t := max(t, t_m)$. If Equation 10 holds true for any C_i, we continue integration at the newly assigned t.

We employ a local illumination scheme and for that calculate the gradient at the integration position on the fly to use it as a placeholder for the surface normal. In order to obtain a more consistent visualization, we modulate the not well defined gradient at the boundary t_{near} with the surface normal of the volume's bounding box. For clipping we proceed in a similar manner by modulating the gradient at the clip boundaries with the front-facing geometry normal that we stored along with the clip intervals C_i.

Clipping with Triangle Meshes

When dealing with clip objects that have a parametric representation, we usually consider only a few objects that need to be intersected with the viewing rays. If the clip object is however a compound object that itself consists of many triangles or similar primitives, we need to intersect each viewing ray with each primitive. BVHs are employed to improve the efficiency of query operations on the list of primitives from $\Theta(n)$ to $\Theta(logn)$, where n denotes the number of primitives in the list.

We briefly review the four query types that are of interest when traversing a list of primitives with a ray and calculating the ray's intersection with each primitive.

Closest-hit: intersect the ray with each primitive to determine the intersection position closest to the ray's origin.

Any-hit: intersect the ray with the list of primitives and break if any intersection was encountered.

Multi-hit: intersect the ray with each primitive to determine the N intersections closest to the ray's origin. This query typically yields a list of length N that is sorted with respect to distance t along the ray. Evaluation of the multi-hit traversal query involves an $O(N)$ step to insert newly found, valid intersections into the sorted list.

All-hit: intersect the ray with each primitive to determine all valid intersections in sorted order. This is a special case of multi-hit, because it is in general not possible to a priori determine the amount of memory required to store the query result.

For an overview of ray traversal queries, see [2]. One can emulate the behavior of the multi-hit query by repeatedly performing the closest-hit query and reassigning the ray's origin $o' = o + dt + d\Delta t$, where o and d are the ray's origin position and direction vector, and Δ is some tiny real number. This however requires Δ to be determined individual for each query, may in general be the cause for rendering artifacts, and imposes an excessive number of unnecessary BVH depth traversals.

We employ the multi-hit query to determine the N closest intersections of the viewing ray with compound clip objects that are made up of geometric primitives organized using a BVH. We therefore must a priori decide how many intersections with the clip geometry may be considered valid in order to preallocate a static memory array that can contain the result from the query operation. We decided for the multi-hit query rather than the all-hit query in order to avoid costly dynamic memory reallocation, which would prohibit a real-time implementation. This is conceptually in line with the approach proposed by Weiskopf et al. [34] who note that a fixed number of render passes to the hardware depth buffer needs to be devised a priori and with the right balance between the application's performance and quality requirements in mind. With the multi-hit query result, we perform the two-pass algorithm outlined above. Our algorithm is in general applicable to any type of compound geometry where the primitives are contained in a BVH and is not limited to triangle meshes. In the following section we propose a cross-platform implementation that can accommodate parametric surfaces such as planes and spheres, as well as triangle meshes.

Implementation

In the following we present a real-time cross-platform implementation of our clipping method by using the generic algorithms and data structures provided by the ray tracing template library Visionaray [28]. The implementation is published by integrating it into the open source DVR library Virvo [26]. Virvo is e.g. used to implement the DVR component of the VR renderer OpenCOVER that is part of the open source visualization software COVISE [24]. OpenCOVER is used for volume rendering in virtual environments such as the CAVE [5].

Cross-Platform Ray Tracing

The Visionaray library advocates a cross-platform programming approach that is based on wrapping the hardware-dependent portion of the ray tracing algorithm using so called *scheduler* classes, which provide an interface to the target hardware and generate primary viewing rays in parallel given e.g. an OpenGL2 compatible pair of camera matrices. Ray traversal is then described using entities called *kernels*. Visionaray provides a SIMD optimized library for short vector math operations and provides intrinsic functions e.g. to access textures or to traverse primitive sets that can be used from within the kernel that is passed to the scheduler for execution. Schedulers then call kernels with a single ray or with a ray packet as parameter.

We build upon the Visionaray ray marching kernel that is already present in the Virvo library by extending it with our clipping algorithm. Our implementation targets x86 CPUs that support either SSE or AVX instruction sets, as well as NVIDIA GPUs that can run CUDA [23] programs, from a single kernel. On GPUs the traversal kernel can make use of hardware accelerated texture accesses. We opted to additionally provide a CPU implementation in order to also evaluate our algorithm with an optimized x86 volume rendering kernel. With CUDA, in accordance to what Aila and Laine [1] proposed in their publication on BVH traversal performance on GPUs, we use single ray traversal, while the optimized SSE and AVX implementations for the x86 platform use ray packets of size four and and eight, respectively.

Visionaray provides optimized 1-D and 3-D texture types and access routines that are described in [38] and that map to dedicated software implementations on the CPU side and to hardware accelerated texture objects and accessors when using CUDA. We use these primitive operations to implement the render pass of our algorithm with front-to-back alpha compositing and post-classification color and alpha transfer function lookups. In order to update the ray distance parameter t, we use a uniform step size Δ, while conditionally performing larger steps when t falls in a clip interval.

Clipping with Arbitrary Geometry Types

Prior to the rendering pass we clip the primary viewing ray that was passed to the kernel with the clip geometry in order to construct clip intervals. To avoid dynamic branching in the compiled assembly output of real-time ray tracing applications, ray tracing APIs typically avoid switching over the supported primitive type in the innermost traversal loop, because this is a major source of performance degradation. For the same reason, geometric primitives in real-time ray tracing applications are typically modeled as "plain old data" (POD) objects without virtual inheritance and costly run time vtable lookups. Maintaining POD objects in memory also spares an extra level of indirection imposed by pointer dereferencing that is necessary with virtual inheritance. Typical real-time ray tracing implementations thus often resort to only supporting a single primitive type for performance reasons [29].

Visionaray provides *generic* routines to perform ray traversal queries which are unaware of the type of primitive that should be traversed, as long as the primitive type implements a free C++ function *intersect()* that takes a ray or ray packet as first and an instance of the primitive type as second parameter. That way, when a ray traversal query such as closest-hit or multi-hit is called e.g. only with triangles, the generic algorithm will only call *intersect()* for that primitive type, so that no unnecessary dynamic branching will occur. Visionaray however provides a *generic_primitive* template type, which can be instantiated with a set of primitive types that are managed by the *generic_primitive*. In that case, storage will be allocated to contain any of the primitive types the *generic_primitive* object was instantiated with, as well as a tag identifying the actual primitive type of the generic wrapper object. At run time, the *intersect()* method will perform branching over the tag to determine for a given set of primitives, what type a specific one has in order to call the appropriate specialized *intersect()* implementation. We employ a *generic_primitive* type encapsulating planes, spheres, and triangle meshes for clipping. This implementation scheme allows for instances of each clip object type to be stored with an optimized data layout as POD objects in any order in a C array that can be passed to the ray tracing kernel for processing. Visionaray provides a BVH implementation based on the construction scheme from [31] that we use to manage triangle meshes. The BVH construction algorithm can optionally perform spatial primitive splits at the cost of slightly higher construction times, which we however did not consider for this implementation. We make use of Visionaray's multi-hit implementation which supports BVH traversal to implement clipping with triangle meshes. Visionaray's multi-hit implementation allows to set the maximum size of the static array returned by the query function to be set as a template parameter.

We provide a user interface to dynamically initialize and update the clipping geometry interactively at run time (cf. Figure 4) for planes and spheres, but assume that triangle meshes are static objects that are initialized only once. Because of that, it is in general possible to reset the triangle meshes for clipping with our application, but we completely rebuild the BVH each time the triangle mesh is changed. We consider support for deformable triangle meshes or for triangle meshes that change position and orientation dynamically interesting topics for future work.

Compared with a texture-based volume clipping approach using slicing and hardware accelerated rendering of the clipping geometry as it was proposed by Weiskopf et al. [34], our approach does not require multiple rendering passes or a depth peeling step to assemble clip regions. Regardless of the different rendering strategies and assuming a GPGPU implementation of our approach, in an abstract sense our algorithm does not rely on an iterative implementation with multiple render passes for clip interval construction as Weiskopf et al. propose, and we also do not require to intermediately store the clip intervals in GPU DDR3 memory, as it would be necessary with an approach similar to depth peeling. We instead precalculate the clip intervals with an optimized GPGPU ray tracer and store them in on-chip GPU memory. That implies faster access times from the GPGPU kernel, which we however trade for an increased register demand per thread on a shading multi processor (SM), which may also impact the performance of the kernel. In the following section we investigate the performance of our approach.

Results

In this section we present qualitative and quantitative results obtained from evaluating our approach. We therefore apply our clipping method to a neurosurgical use case. We further conduct a quantitative evaluation and present performance measurements.

Application to Neurosurgery

In order to assess the qualitative appropriateness of our clipping algorithm for neurosurgical use cases, we prepared the patient dataset depicted on the left hand side of Figure 2. The figure shows results from manual brain tumor delineation conducted during neurosurgical operation planning. The red dotted line marks the boundary surface of the tumor. Shown is the delineation for a single CT image, while delineation is typically performed for each CT image where tumor tissue is visible. Figure 5 shows the hull of the tumor rendered as a triangle mesh that was

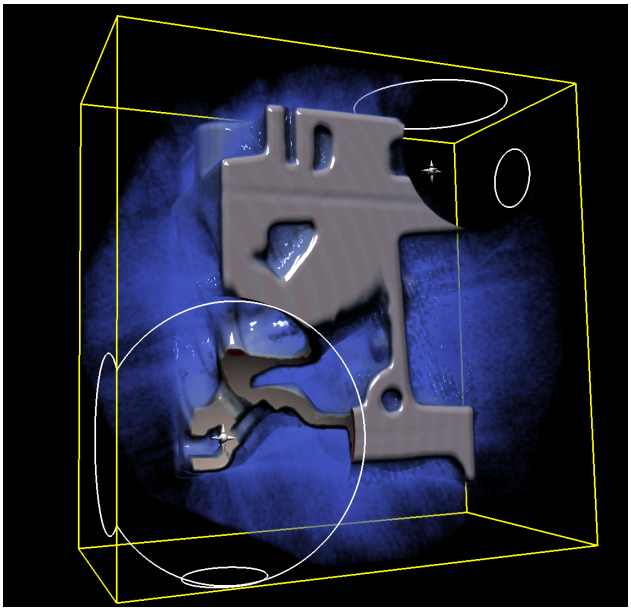

Figure 4. *User interface for clipping with parametric surfaces in the VR renderer OpenCOVER. The data set is clipped with two spheres that can be repositioned interactively using 3-D widgets.*

reconstructed from the delineation through tesselation. In Figure 6 our clipping algorithm was used to virtually excise the tumor from the volume rendered CT image. This results in the outer part of the tumor being projected to the surface of the clip geometry. In order to expose the volume of interest resulting from tumor delineation, we used a vertical clip plane running through the center of the tumor (top left and bottom row of Figure 6). From the zoomed in view (bottom left image portion) the neurosurgeon can assess the vessel penetration of the tumor hull. Furthermore, the neurosurgeon can review the appropriateness of the delineation and, if indicated, reiterate it. This process can be conducted more accurately and directly as it is possible with mere 2-D imaging.

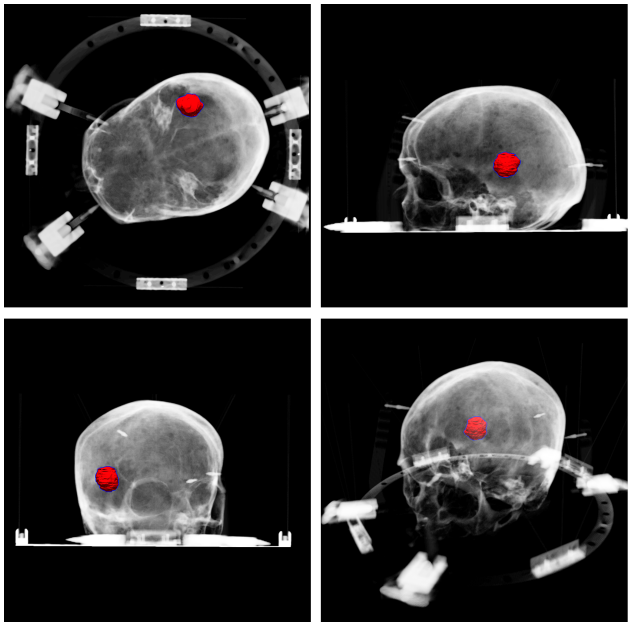

Figure 5. *Triangle geometry obtained from triangulating the hull of a manually delineated brain tumor during neurosurgical operation planning. The image shows direct volume rendering of a CT dataset and the volume of interest defined by the delineated tumor's hull. The images were rendered with three different orthographic viewing configurations (top row, bottom left) as well as from a perspective viewing position (bottom right).*

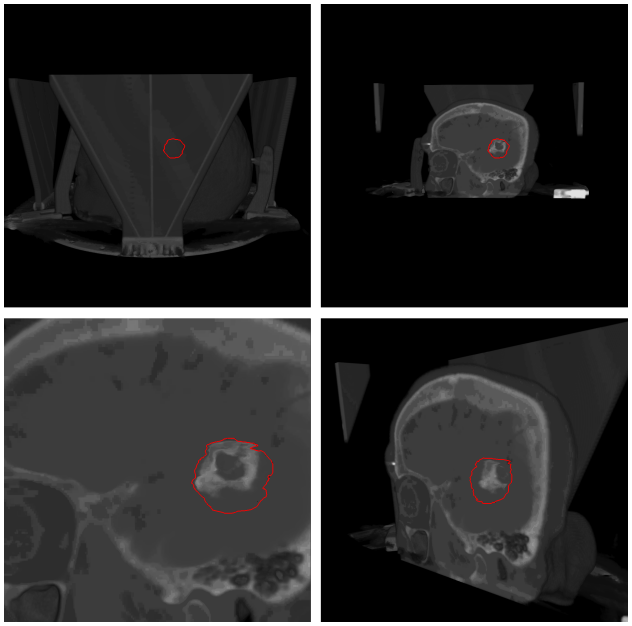

Figure 6. *The delineated tumor clipped from the CT volume data set using our volume clipping approach. Various clipping modes are presented. In the top left image region the volume of interest is obscured (the outlines of the volume of interest are depicted for clarity). In the remaining image portions, the volume of interest is clipped from the CT dataset and a clip plane through the center of the volume of interest is used to expose it.*

Rendering Performance

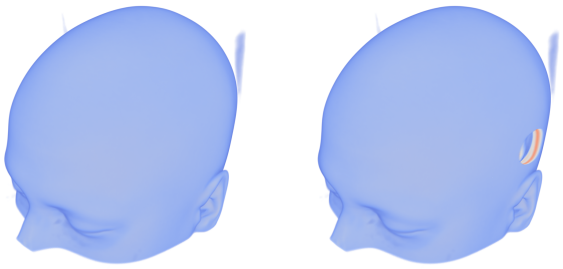

Figure 7. *MRI data set used for the performance evaluation. We test with a modality where most parts of the inside of the volume are clipped with the teapot geometry (right) and compare with a modality where clipping is deactivated (left).*

When evaluating the performance of our approach, we are interested in the impact on total rendering time. Clipping with single surfaces such as planes or spheres in general caused no perceptible performance degradation and moreover even resulted in a performance gain when large portions of the data set were clipped and thus did not require to be integrated over. We thus concentrate on the performance impact of clipping with triangle meshes and multi-hit traversal. Our performance evaluation considers how triangle count as well as the maximum number of allowed multi-hit intersections influences the rendering performance. We therefore render frames with a resolution of 1024×1024 pixels of the MRI data set depicted in Figure 7 and clip it with the well known Utah teapot 3-D model. The MRI data set is the T1-weighted MNI152 standard brain that is available from the Montreal Neurological Institute (MNI) under a license allowing non-commercial use and can be downloaded as part of the FSL FMRI tools [37]. The version that we use was sampled with a $0.5mm^3$ voxel size at a resolution of $364 \times 436 \times 364$ voxels. In order to determine the impact of mesh size on rendering performance, we apply subdivision surface mesh

Table 1. Triangle count and number of BVH nodes after subdivision was applied to the teapot triangle mesh for clipping.

Level	Triangles	BVH Nodes
1	2,464	1,559
2	14,784	9,771
3	59,136	39,063
4	236,544	151,637

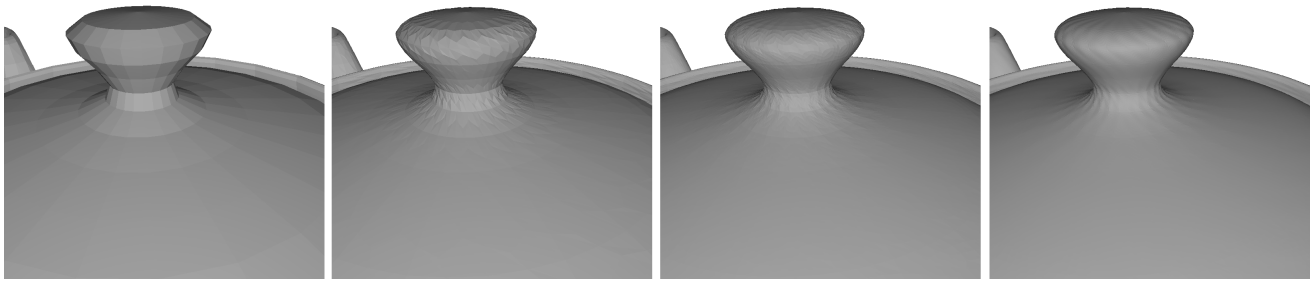

Figure 8. *In order to test how the performance of our algorithm varies with the complexity of the clip geometry, we employ four variants of the famous Utah teapot for clipping, with different subdivision surface levels applied. From left to right: 1.: 2,464 triangles and 1,559 BVH nodes. 2.: 14,784 triangles and 9,771 BVH nodes. 59,136 triangles and 39,063 BVH nodes. 236,544 triangles and 151,637 BVH nodes.*

refinement using a 3-D modeling tool to the teapot geometry. The result of the refinement steps is shown in Figure 8. We report the triangle count along with the number of BVH nodes created by Visionaray's BVH builder in Table 1. Because we are interested in the impact of the clipping algorithm, for our tests we deactivate gradient-based shading and opacity correction, because these computations require evaluating a costly exponential function and would falsify the performance measurements we are going to report in this section.

We present performance results that we measure on a 1st generation NVIDIA GTX Titan GPU. We therefore compile the CUDA program for the sm_20 architecture. In order to gain a better understanding of the resource demand of our implementation, we report the number of local registers and potential register spilling to local memory for different configurations. By completely disabling clipping with the multi-hit query for triangle meshes we define the baseline for the resource demand, the CUDA compute kernel then uses 62 registers per thread and the optimizer does not spill registers to local memory (which would impose a higher memory access latency). We vary N, the size of the array containing the multi-hit result, and report the register usage of the respective compute kernels in Table 2. We present rendering performance results for the CUDA implementation with varying multi-hit N in Table 3. We also report performance measurements for the respective modalities where we just compiled the clipping algorithm into the CUDA compute kernel, but without actually loading a clip geometry. We provide these numbers in order to analyze the impact of the increased per thread resource demand on rendering performance. In addition, we present results for the CPU packet traversal implementation that was compiled for the SSE 4.1 instruction set architecture that is summarized in Table 4. We performed the measurements on an Intel dual socket server system with two Intel Xeon E5-2630 six core CPUs running at a base frequency of 2.30 GHz. With Hyper-Threading activated we allocate 24 independent threads for our tests. We do not expect an impact as severe as on the GPU due to the mere existence of the instructions associated with our clipping algorithm because of the differing register allocation process. For completeness' sake and for better comparability, we report performance measurement results for the respective modalities, anyway.

Table 2. Register usage of the complete CUDA compute kernel (clip interval assembly plus rendering) depending on the size N of the multi-hit result array.

Multi-Hit N	Registers	Spill stores	Spill loads
0 (disabled)	62	0 bytes	0 bytes
8	63	176 bytes	228 bytes
16	63	364 bytes	368 bytes
24	63	604 bytes	584 bytes
32	63	924 bytes	732 bytes

Table 3. Performance measurements in frames per second (fps) obtained from clipping with the teapot geometry and four different subdivision levels for varying multi-hit N with CUDA on the GPU. Row one shows results for the baseline kernel without multi-hit clipping and row two shows results for the kernel compiled with multi-hit clipping but without actually using it.

	N=8	N=16	N=24	N=32
No Multi-Hit	(23.0)	(23.0)	(23.0)	(23.0)
Unused	16.3	16.3	11.0	11.0
Subdiv. 1	11.4	9.3	5.4	4.7
Subdiv. 2	9.3	7.2	4.0	3.5
Subdiv. 3	7.6	5.7	3.1	2.6
Subdiv. 4	6.1	4.5	2.4	2.0

Table 4. Performance measurements in frames per second (fps) when performing the performance tests on the CPU, with SIMD SSE coherent packet traversal (packets occupy the space of 2 by 2 pixels).

	N=8	N=16	N=24	N=32
No Multi-Hit	(3.1)	(3.1)	(3.1)	(3.1)
Unused	(3.1)	(3.1)	(3.1)	(3.1)
Subdiv. 1	2.9	2.8	2.7	2.6
Subdiv. 2	2.8	2.6	2.4	2.3
Subdiv. 3	2.7	2.4	2.2	2.0
Subdiv. 4	2.5	2.2	2.0	1.8

Discussion

We demonstrated the usefulness of our clipping method in the context of neurosurgical operation planning. By mapping the outer part of a brain tumor to the concave surface of the delineated tumor hull, the neurosurgeon can better assess vessel penetration as well as the appropriateness of the delineation itself. This process is especially aided by means of 3-D imaging in contrast to reviewing the delineation in 2-D. However, in order to review the tumor hull in 3-D, an additional interaction component is necessary. In the example we presented, an additional clip plane running through the tumor center was used to expose the volume of interest. As an alternative means of interaction one could have moved the viewing position inside the delineated tumor. However, with both interaction modes, only a portion of the delineated tumor is visible at a single instance of time. For the visualization to be more useful in the context of the time-critical neurosurgical operation planning process, we therefore envision unrolling the tumor hull and projecting it to 2-D, possibly in a manner similar to the technique presented by Kretschmer et al. [15], as helpful and consider an implementation and evaluation of additional, more application centric user interfaces interesting future work.

The two implementations we provide differ by an order of magnitude in terms of run time performance, which was to be expected due to the higher parallelism, memory bandwidth, and due to hardware support for 1-D and 3-D texture lookups on the GPU. The downside of a GPU implementation that we propose is the influence of high resource demands on the register optimization level on rendering performance - for a high-quality clipping mode with many intersections allowed and with a complex clip geometry, the measured performance of the two implementation converges. At the point of convergence, frame rates of both implementations are however no longer interactive.

We would further like to point out the good scalability of our algorithm with triangle mesh complexity. Due to the $\Theta(logn)$ search pass over the BVH, the performance does not degrade linearly with triangle count, which would have to be expected in a scenario as that proposed by Weiskopf et al. [34], where the clip geometry is rasterized to the hardware depth buffer. From a performance optimization standpoint, the two approaches (ours and that of Weiskopf et al.) differ in an interesting way. Weiskopf et al.'s approach stores clip information in GPU DDR memory, while our approach stores clip information in registers and in thread-local memory, which can be accessed much faster. On the other hand, the CUDA programming model and the GPU scheduler allow to efficiently hide memory access latency behind parallel computations, so that we believe it is undecided how an approach storing clip information in GPU DDR memory at the benefit of lower resource demand compares to our approach on contemporary hardware. Storing clip intervals in GPU DDR memory could e.g. be implemented using a wavefront approach [17]. Because on today's GPUs texture memory is no longer as limited as it was as of 2002, we argue that an approach storing the whole clip interval array (per viewing ray) in GPU DDR memory might be worthwhile for further investigation. We have shown that clipping with triangle meshes with a limited number of concavities can be performed interactively with our method, which compares well with state-of-the-art GPU techniques based on rasterization and consider a comparison with wavefront approaches interesting future work.

Conclusions

We have proposed a sub-voxel accurate volume clipping method that fits into a ray tracing-based pipeline and that compares well with state-of-the-art texture-based methods that use rasterization on graphics hardware. Sub-voxel accurate clipping is a highly relevant operation in the context of neurological imaging and for stereotactic operation planning. Our approach does not require the clip geometry to be converted to triangles. If however the clip geometry is available as a mesh, our method incorporates multi-hit ray / BVH traversal to efficiently identify clip intervals. Because of the $\Theta(logn)$ time complexity of ray / BVH intersection our method scales well with triangle count. The performance of our algorithm is however bounded by the number of allowed intersections of rays with the geometry, and thus by the number of concavities that can be displayed. This is in line with approaches based on rasterization, where each concavity causes an additional render pass. We provide a cross platform implementation of our algorithm as part of an open source visualization software and conducted performance measurements on both CPU and GPU. In the future we would like to evaluate if an implementation based on wavefront ray tracing, where the clip intervals are temporarily stored in GPU DDR memory and the two passes of our algorithm are implemented in two different kernels, can improve performance through reduced register pressure so that a higher number of concavities can be rendered in real-time.

References

[1] Timo Aila and Samuli Laine, Understanding the Efficiency of Ray Traversal on GPUs, Proceedings of High-Performance Graphics 2009, pg. 145. (2009).

[2] Jefferson Amstutz, Christiaan Gribble, Johannes Günther, Ingo Wald, An Evaluation of Multi-Hit Ray Traversal in a BVH using Existing First-Hit/Any-Hit Kernels, Journal of Computer Graphics Techniques (JCGT), 4, 4 (2015).

[3] L. Bavoil and K. Myers, Order independent transparency with dual depth peeling, Technical Report, NVIDIA Coorp. 124, 127 (2008).

[4] Jon Louis Bentley, Multidimensional binary search trees used for associative searching, Communications of the ACM, 18, 9 (1975).

[5] Carolina Cruz-Neira, Daniel J. Sandin, Thomas DeFanti, Robert V. Kenyon, John C. Hart, The CAVE: Audio Visual Experience Automatic Virtual Environment, Commun. ACM, 35, 6 (1992).

[6] Y. Dai, J. Zheng, Y. Yang, D. Kuai, X. Yang, Volume-Rendering-Based Interactive 3D Measurement for Quantitative Analysis of 3D Medical Images, Computational and Mathematical Methods in Medicine (2013).

[7] Holger Dammertz, Johannes Hanika, Alexander Keller, Shallow Bounding Volume Hierarchies for Fast SIMD Ray Tracing of Incoherent Rays, Computer Graphics Form (Proc. 19th Eurographics Symposium on Rendering, pg. 1225. (2008).

[8] Thomas Fogal and Jens Krüger, Tuvok, an Architecture for Large Scale Volume Rendering, Proceedings of the 15th International Workshop on Vision, Modeling, and Visualization, pg. 139. (2010).

[9] P. Ganestam, R. Barringer, M. Doggett, T. Akenine-Möller, Bonsai: Rapid Bounding Volume Hierarchy Generation using Mini Trees, Journal of Computer Graphics Techniques (JCGT), 4, 3 (2015).

[10] Christiaan Gribble, Alexis Naveros, Ethan Kerzner, Multi-Hit Ray Traversal, Journal of Computer Graphics Techniques (JCGT), 3, 1 (2014).

[11] Christiaan Gribble, Node Culling Multi-Hit BVH Traversal, Euro-

graphics Symposium on Rendering - Experimental Ideas and Implementations, (2016).
[12] James T. Kajiya, The Rendering Equation, SIGGRAPH Comput. Graph., 20, 4 (1986).
[13] Aaron Knoll, Sebastian Thelen, Ingo Wald, Charles D. Hansen, Hans Hagen, Michael E. Papka, Full-Resolution Interactive CPU Volume Rendering with Coherent BVH Traversal, Proceedings of IEEE Pacific Visualization, pg. 3. (2011).
[14] A. Kratz, M. Hadwiger, R. Splechtna, A. Fuhrmann, K. Bühler, GPU-Based High-Quality Volume Rendering For Virtual Environments, Proceedings of AMI-ARCS, pg. 1. (2006).
[15] Jan Kretschmer, Grzegorz Soza, Christian Tietjen, Michael Suehling, Bernard Preim, Marc Stamminger, ADR - Anatomy-Driven Reformation, IEEE Transactions on Visualization and Computer Graphics (TVCG), 20, 12 (2014).
[16] Jens Krüger and Rüdiger Westermann, Acceleration techniques for GPU-based volume rendering, Proceedings IEEE Visualization 2003, pg. 287. (2003).
[17] Samuli Laine, Tero Karras, Timo Aila, Megakernels Considered Harmful: Wavefront Path Tracing on GPUs, Proceedings of the 5th High-Performance Graphics Conference, pg. 137. (2013).
[18] C. Lauterbach, M. Garland, S. Sengupta, D. Luebke, D. Manocha, Fast BVH Construction on GPUs, Comput. Graph. Forum, 28, 2 (2009).
[19] Marc Levoy, Efficient Ray Tracing of Volume Data, ACM Trans. Graph., 9, 3 (1990).
[20] M. Levoy, H. Fuchs, S.M. Pizer, J. Rosenman, E.L. Chaney, G.W. Sherouse, V. Interrante, J. Kiel, Volume Rendering in Radiation Treatment Planning, Proc. First Conference on Visualization in Biomedical Computing, IEEE Computer Society Press, pg. 4. (1990).
[21] Nelson Max, Optical models for direct volume rendering, IEEE Transactions on Visualization and Computer Graphics, 1, 2 (1995).
[22] Jennis Meyer-Spradow, Timo Ropinski, Jörg Mensmann, Klaus H. Hinrichs, Voreen: A Rapid-Prototyping Environment for Ray-Casting-Based Volume Visualizations, IEEE Computer Graphics and Applications, 6, 29 (2009).
[23] John Nickolls, Ian Buck, Michael Garland, Keven Skadron, Scalable Parallel Programming with CUDA, ACM Queue, 6, 2 (2008).
[24] D. Rantzau, U. Lang, R. Lang, H. Nebel, A. Wierse, R. Ruehle, Collaborative and Interactive Visualization in a Distributed High Performance Software Environment, High Performance Computing for Computer Graphics and Visualization, Springer London, 1996, pg. 207.
[25] Steven M. Rubin and Turner Whitted, A 3-dimensional Representation for Fast Rendering of Complex Scenes, SIGGRAPH Comput. Graph, 14, 3 (1980).
[26] Jürgen Schulze-Döbold, Uwe Wössner, Steffen P. Walz, Ulrich Lang, Volume Rendering in a Virtual Environment, Immersive Projection Technology and Virtual Environments 2001: Proceedings of the Eurographics Workshop, pg. 189. (2001).
[27] Martin Stich, Heiko Friedrich, Andreas Dietrich, Spatial Splits in Bounding Volume Hierarchies, Proceedings of High-Performance Graphics 2009, pg. 7. (2009).
[28] Visionaray - A Cross Platform Real-Time Ray Tracing Kernel Framework, `http://vis.uni-koeln.de/visionaray.html`, accessed August 10, 2016.
[29] Ingo Wald, Philipp Slusallek, Carsten Benthin, Markus Wagner, Interactive Rendering with Coherent Ray Tracing, Computer Graphics Forum, pg. 153. (2001).
[30] Ingo Wald, Solomon Boulos, Peter Shirley, Ray Tracing Deformable Scenes Using Dynamic Bounding Volume Hierarchies, ACM Trans. Graph., 26, 1 (2007).
[31] Ingo Wald, On fast Construction of SAH-based Bounding Volume Hierarchies, Proceedings of the 2007 IEEE/Eurographics Symposium on Interactive Ray Tracing, pg. 33. (2007).
[32] Ingo Wald, Sven Woop, Carsten Benthin, Gregory S. Johnson, Manfred Ernst, Embree: A Kernel Framework for Efficient CPU Ray Tracing, ACM Trans. Graph., 33, 4 (2014).
[33] Chen Wei , Hua Wei , Bao HuJun , Peng QunSheng, Real-time ray casting rendering of volume clipping in medical visualization, J. Comput. Sci. and Technol., 18, 6 (2003).
[34] Daniel Weiskopf, Klaus Engel, Thomas Ertl, Volume Clipping via Per-fragment Operations in Texture-based Volume Visualization, Proceedings of the Conference on Visualization '02, pg. 93, (2002).
[35] Rüdiger Westermann and Thomas Ertl, Efficiently using graphics hardware in volume rendering applications, SIGGRAPH '98: Proceedings of the 25th annual conference on computer graphics and interactive techniques, pg. 169. (1998).
[36] Turner Whitted, An Improved Illumination Model for Shaded Display, Commun. ACM, 23, 6, (1980).
[37] M.W. Woolrich, S. Jbabdi, B. Patenaude, M. Chappell, S. Makni, T. Behrens, C. Beckmann, M. Jenkinson, S.M. Smith, Bayesian analysis of neuroimaging data in FSL, NeuroImage, 45, 1 (2009).
[38] Stefan Zellmann, Yvonne Percan, Ulrich Lang, Advanced texture filtering: a versatile framework for reconstructing multi-dimensional image data on heterogeneous architectures, Visualization and Data Analysis 2015, pg. 1. (2015).

Author Biography

Stefan Zellmann is with the Chair of Computer Science at the University of Cologne since 2009. Before that, he graduated from the University of Cologne in information systems. In 2014, he received his doctor's degree in computer science, with a PhD thesis on high performance computing and direct volume rendering for scientific visualization.

Mauritius Hoevels is medical physicist at the University Hospital of Cologne, Department of Stereotaxy and Functional Neurosurgery. He graduated in medical physics in 1991 and is working in the field of stereotactic surgery, radiosurgery, and deep brain stimulation.

Ulrich Lang holds a chair of Computer Science at the University of Cologne since 2004. He is at the same time director of the Regional Computing Center at the University of Cologne. Before that, he was deputy director of HLRS, a German national supercomputing Center in Stuttgart, where he also headed the visualization department. He graduated from the University of Stuttgart as Dr.-Ing. At HLRS he coordinated the development of COVISE, a COllaborative VIsualization and Simulation Environment, that supports e.g. the analysis of simulation results in virtual environments.