Flexible GPU-Based Multi-Volume Ray-Casting

Ralph Brecheisen, Anna Vilanova i Bartroli, Bram Platel, Bart ter Haar Romeny

Technical University of Eindhoven
Email: {r.brecheisen,a.vilanova,b.platel,b.m.terhaarromeny}@tue.nl

Abstract

Using combinations of different volumetric datasets is becoming more common in scientific applications, especially medical environments such as neurosurgery where multiple imaging modalities are required to provide insight to both anatomical and functional structures in the brain. Such data sets are usually in different orientations and have different resolutions. Furthermore, it is often interesting, e.g. for surgical planning or intraoperative applications to add the visualization of foreign objects (e.g., surgical tools, reference grids, 3D measurement widgets). We propose a flexible framework based on GPU-accelerated ray-casting and depth peeling, that allows volume rendering of multiple, arbitrarily positioned volumes intersected with opaque or translucent geometric objects. These objects can also be used as convex or concave clipping shapes. We consider the main contribution of our work to be the flexible combination of the abovementioned features in a single framework. As such, it can serve as a basis for neurosurgery applications but also for other fields where multi-volume rendering is important.

1 Introduction

The acquisition of multiple volumetric datasets based on data measurements or simulations is becoming more common in modern scientific applications. This is especially apparent in medical environments. Modern hospitals nowadays have a wide range of 3D imaging modalities at their disposal. For example, in neurosurgery of the brain, it is not possible to visualize all structures of interest using only a single imaging modality. Multiple modalities have to be combined to view both anatomical structures and functional areas. Relevant modalities in this case are Computed Tomography (CT), Magnetic Resonance Imaging (MRI), functional Magnetic Resonance Imaging (fMRI) and Diffusion Tensor Imaging (DTI). It is also interesting to combine the volume data with foreign objects that can only be represented with geometric primitives. For neurosurgery applications one can think of virtual surgical tools, reference grids, 3D measurement widgets or streamlines to visualize Diffusion Tensor Imaging data. To avoid cluttering and occlusion of underlying volume data it is useful to be able to render these geometric objects semi-transparently. Furthermore, to explore hidden structures in the volumetric data both semi-transparency, using opacity transfer functions, and surface-based clipping should be available. For example, a complex and possibly concave clipping shape could be used for virtual resection or an extracted surface model of the brain could be used as a clipping shape.

Different datasets acquired for the same patient often have different resolutions. Also their coordinate systems are not always exactly aligned which requires them to be registered. To visualize all datasets simultaneously a common approach is to resample them onto a common grid such that they all share the same coordinate frame. This allows easier sampling but also reduces the image quality due to double interpolations. Low-resolution datasets may need to be inflated thereby increasing memory consumption. This can become an issue in intraoperative navigation where the difference in resolution between preoperative and intraoperative images can be large. For this reason, we believe that resampling should be avoided.

Several multi-volume rendering frameworks exist, however to our knowledge none of these implementations fulfill all the points specified above. Especially, the *combination of multiple volumes with transparent geometry and concave clipping* is not supported by any existing framework. For this reason, we propose a GPU-based multi-volume ray-casting framework with the goal of being

as flexible as possible with respect to the points specified above, that is, allowing independent, volume-local sampling of multiple volumetric datasets possibly intersected with opaque or translucent geometric shapes and convex or concave clipping shapes. Registration of the datasets is assumed as a pre-processing step that gives us the coordinate transformations needed to position volumes and geometry in space.

The outline of this paper is as follows: Section 2 discusses related work in the field of multi-volume rendering. In Section 3 we explain the concepts of our multi-volume depth peeling algorithm, volume/geometry intersections, different options for rendering intersection regions and depth-based clipping. In Section 4 we illustrate how our multi-volume rendering framework could benefit the planning of brain tumor resections. In this Section we also discuss memory and performance issues. Finally, we end with conclusions and future work in Section 5.

2 Related Work

Several implementations for multi-volume rendering have been proposed in the past. Due to the computational complexity of rendering multiple volumes, software-based methods require advanced acceleration techniques such as using segmentation information [22] and efficient memory addressing and multi-threading [17]. Without such techniques software-based multi-volume rendering is restricted either to non-intersecting volumes [4] or non-interactive, static images [34, 23].

With the advent of harware acceleration and programmable GPU's interactive and high-quality multi-volume rendering has become much more feasible. Hadwiger et al. [13] propose a two-level volume rendering method that allows different parts of the dataset to be visualized with different rendering modes such as MIP, DVR and isosurface. However, they do not explicitly support multiple volumes or intersecting geometry.

As mentioned in the introduction, we consider the combination of volume data with geometric shapes to be an important feature of a multi-volume rendering framework. However, most multi-volume rendering frameworks do not support this [8, 24, 9, 16, 19]. Those that do, are mostly limited to convex, opaque geometry [10, 7, 6]. Translucent geometry intersections are reported by Levoy [2] and Kreeger et al. [5]. However, the ray tracing framework proposed by Levoy only produces still images. Kreeger et al. mention that they support multiple volumes with translucent geometry, however they discuss only the single-volume case. Futhermore, their implementation requires a specialized hardware architecture.

In our algorithm we use depth peeling [33] in combination with GPU-accelerated ray-casting [29, 27] to allow flexible volume/geometry intersections. A similar approach has been used by others [14, 11, 35, 30] although these implementations are restricted to single-volume scenes. Borland et al. [30] propose a technique called *volumetric depth peeling*, however it is not related in any way to depth peeling as an algorithm for depth-sorting translucent polygons. It does impose a layered structure on the volume data by restarting ray-casting after some threshold is reached, but this is a purely volumetric technique with no support for intersecting geometry.

One of the most challenging issues in multi-volume rendering is how to deal with overlapping materials (or tissues). This remains a largely unsolved question but several implementations exist that offer a range of mixing strategies at different levels in the rendering pipeline. They are either based on opacity-weighted mixing [3, 20, 21, 19, 24], successively applying compositing on each sample (in some order) [10, 17] or other operators such as minimum, maximum, negative, sum, product and blend [9].

Most implementations simplify sampling by mapping all volumes onto a common coordinate frame [6, 16, 10, 15]. However, if volumes are not axis-aligned and only partially overlapping then this requires resampling. This introduces numerical inaccuracies and may also increase memory consumption significantly if resolutions differ greatly. Only a few implementations sample each volume locally within its own coordinate frame [17, 4, 8, 9], however neither of these support volume/geometry intersections or complex clipping shapes.

Finally, many existing methods for multi-volume rendering were proposed with a clear application in mind such as seismic data exploration [9]. In the medical area, Beyer et al. [10] propose a

framework for planning key-hole neurosurgery. Hardenbergh et al. [6] added DTI fiber geometry to an already existing functional MRI visualization system. However, they do not support translucent geometry or complex clipping shapes. Koehn et al. [26] present a system for neurosurgery planning that has features very similar to ours. However, they do not support multi-volume rendering at the sampling level. They can combine multiple volume renderers and mix their outputs at the image-level.

Much work has been done for multi-volume rendering. However, we conclude that most frameworks do not have the flexibility to combine multiple volumes in arbitrary orientations without resampling or deal with intersecting geometric shapes and clipping. We believe that this is especially needed in neurosurgery applications. We intend to offer this flexibility by using (1) GPU ray-casting, which allows interactive, high-quality rendering and perspective projections, and (2) depth peeling, which allows a natural integration between volumes, opaque or translucent geometric surfaces (e.g. for visualizing surgical tools) and complex clipping combinations (e.g. for virtual resection).

3 Method Overview

Sampling multiple volumes independently in a single rendering pass is problematic, especially if the volumes are intersected by arbitary and possibly translucent or concave geometric surfaces. To correctly include the geometry colors and opacities in the volume compositing along each ray, intersection calculations would have to be performed at each sample step for all possible surfaces. Computationally this is not feasible in real-time, especially for more complex object shapes. For this reason, we propose a multi-pass approach where we render the scene in separate layers using *depth peeling* [33]. This subdivides the scene into regions where sampling conditions do not change and we only have to determine our sampling position with respect to each volume at the beginning of such a region. As we will describe later, surface geometry requires no special intersection calculations. Figure 1 gives a high-level overview of our algorithm and the functional steps performed during a *single* iteration of the algorithm. Rendering the complete volume/geometry scene consists of repeatedly executing these steps for each iteration after which the final result is rendered to screen.

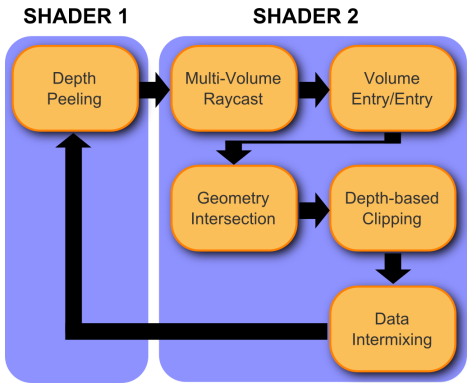

Figure 1: Overview of the algorithm subdivided over two fragment shaders, one for depth peeling (shader 1) and one for ray-casting (shader 2). Illustrated are the different functional steps taken during a single iteration of the algorithm.

In short, our algorithm alternates depth peeling with a ray-casting pass until all depth layers of the volume/geometry scene have been processed. The use of depth peeling and ray-casting is not new in itself [14, 11, 35]. However, their application to multiple volumes intersected by translucent geometry and concave clipping shapes is novel and our main contribution. In the following subsections we explain our multi-volume ray-casting algorithm in more detail.

3.1 Depth Peeling

Depth peeling is a well-known technique for view-independently depth sorting a collection of polygon surfaces and originally used for rendering translucent polygons correctly. It is a multi-pass, fragment-level technique that was initially described by Mammen [12] and Diefenbach et al. [25] and implemented by Everitt [33] using hardware-accelerated shadow mapping. Depth peeling can be applied to any scene consisting of polygon surfaces. This means it can also be applied to GPU-based ray-casting because it is initiated by rasterizing the front faces of a geometric bounding box. If we have multiple, partially overlapping volumes, rasterizing their bounding boxes while performing depth peel-

ing over multiple passes will sequentially give us access to the fragments of each volume's boundary. This is precisely what we need for efficiently computing sample positions with respect to each volume. To implement depth peeling for multi-volume ray-casting we use an approach that is similar to the algorithm proposed by Everitt [33]. However, he uses a shadow mapping technique that relies on relatively slow and obsolete buffer-to-texture copies. We use the OpenGL Framebuffer extension and programmable fragment shaders to extract the different depth layers and store them directly in a set of high-precision depth textures without the need for expensive copy operations.

3.2 Multi-Volume Ray-Casting

Our multi-volume ray-casting algorithm starts by rendering the front-most faces of the scene and storing the fragment depths and colors in offscreen buffers D_{near} and C_{near}. This is illustrated in Figure 2A. In the second pass the scene is rasterized again except now a fragment shader is activated, with D_{near} as input parameter, that compares the rasterized fragment depths to the corresponding depth value in buffer D_{near}. If, for a given fragment, its depth is *less or equal* to the D_{near} value then the fragment is discarded. Since depth test LESS is still applied, this will result in the second-nearest fragments being stored in a second set of offscreen buffers D_{far} and C_{far}. This is illustrated in Figure 2B.

After depth peeling the first two depth layers, stored in D_{near} and D_{far}, we initiate a ray-casting pass by rasterizing the scene again with a depth test EQUAL. The depth test only passes fragments that have a depth equal to those in D_{near}. For these fragments a second fragment shader is activated, with D_{near}, D_{far} and color buffer C_{near} as input parameters. This fragment shader performs direct volume ray-casting starting at the layer corresponding to D_{near} and stopping at the layer corresponding to D_{far}. The composited color and opacity are written to an offscreen accumulation buffer C_{acc} which is updated and passed as an input parameter to the next ray-casting pass.

Sampling corrections: When doing ray-casting in a multi-volume scene, special care should be taken to preserve equidistant step sizes as we cross volume boundaries from one iteration to the next. The distance between two depth layers will almost never be an exact multiple of the ray step size.

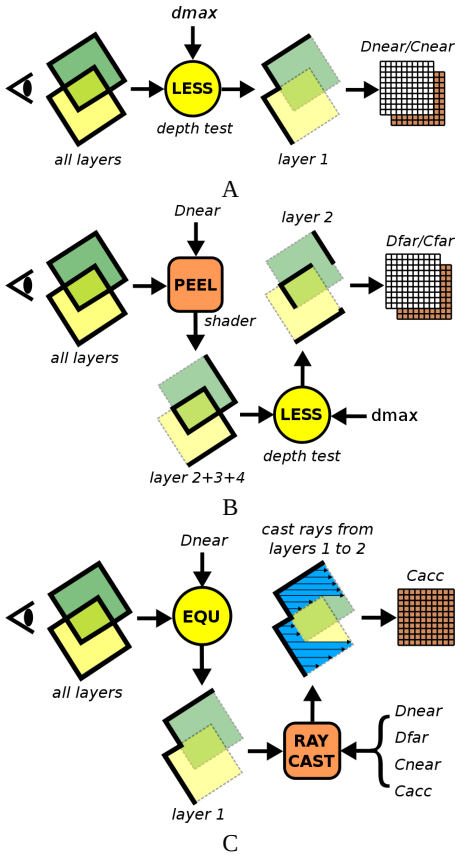

Figure 2: (A) Use depth test LESS to extract first depth layer and store in texture D_{near}. (B) Peel away first depth layer by using fragment shader to discard fragments with depth equal to D_{near}, then use depth test LESS again to extract second depth layer and store in texture D_{far}. (C) GPU ray-casting is started at the fragments corresponding to D_{near}. A depth test EQUAL is used to select these fragments. The output of the ray-casting shader is written to the color accumulation buffer C_{acc}.

Therefore, to preserve equidistant steps along the ray through all volumes we need to add a slight offset Δd at the beginning of each region (except the first). This offset is computed in the previous

pass.

Intersecting geometry: Due to our depth peeling approach it is relatively easy to implement intersections with geometry because it is depth-peeled just like any other layer. We store the color and opacity of each layer in C_{near} so it is straightforward to include this color and opacity in the compositing equation before we do ray-casting for any given region. In this way, geometry is naturally taken into account. Whether the geometry is opaque or semi-transparent makes no difference.

3.3 Volume Entry/Exit States

To keep track of which volumes the viewing ray enters and exits in a flexible way, we introduce a scheme to maintain a list of bitflags, one flag for each volume, that is toggled each time we pass a volume's boundary. A similar approach to store state variables across multiple rendering passes is used by Beyer et al. [10]. While they use a 1D lookup texture to keep track of which (anatomical) objects map to which volume ID, we use a 2D texture to store a list of bitflags *for each viewing ray* or pixel. It is illustated in Figure 3. All bitflags are initialized to zero at the beginning of ray traversal. Each volume has an associated index which is used to retrieve the corresponding bitflag from the list. To implement this feature, we use an additional, screen-sized texture, which we call the *objectInfo* texture, that is queried and updated in each ray-casting pass. Each texture element contains a floating-point value that encodes the list of bitflags. Encoding and decoding of bitflags as floating-point values is done using the *modulo* operator. If E_i is the bitflag value for volume V_i, T is the floating-point texture value to be decoded and N is the number of volumes then we have,

$$E_i = \mathrm{mod}\left(\frac{T}{2^i}, 2\right), 0 \leq i < N \qquad (1)$$

Given a volume index i and bitflag value E_i we can toggle the bitflag, without using IF-statements, as follows,

$$E_i = \mathrm{mod}\,(E_i + 1, 2) \qquad (2)$$

Finally, re-encoding the bitflags into the floating-point value is done as follows,

$$T = \sum_{i=0}^{N} 2^i \cdot E_i \qquad (3)$$

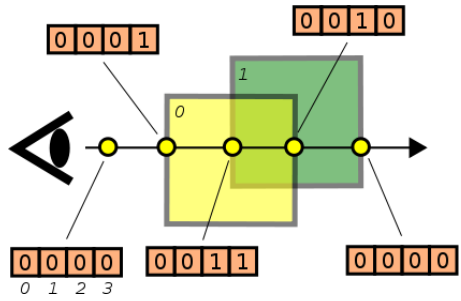

Figure 3: Bitflag values for traversing two volumes (least significant bit on the right, only 2 out of 4 bitflags used). The indices (0,1,2,3) refer to the volume index.

3.4 Depth-based Clipping of Selected Volumes

Volume clipping is an essential part of any 3D viewing toolbox because it helps to reduce cluttering when a lot of information has to be visualized. Also, if the sample values of an object differ only slightly from the objects occluding it, it will be difficult to expose it using a basic opacity transfer function. In that case, clipping becomes the only way to show the object by cutting away the occluding data.

In our multi-volume rendering algorithm we offer support for depth-based clipping. We keep track of clipping states in a similar manner as for volumes except we do not use 'entry/exit' (1/0) bitflags but simply 'clip/no-clip' (1/0) bitflags. These bitflags are encoded in a second channel of the *objectInfo* texture which we introduced for the volume bitflags. Depending on the clipping mode, volume *probing* or volume *cutting* [11], the *objectInfo* texture is initialized with '1' or '0' respectively. This is illustrated in Figure 4.

Using these volume and clipping bitflags we can quickly determine for each volume whether to render it or not in a given region. We are able to handle all the different combinations of volumes, clipping shapes and clipping modes (probing or cutting) in a

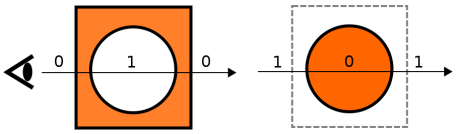

Figure 4: Clipping modes: volume cutting (left) requires clip object bitflag to start with '0', volume probing (right) requires bitflag to start with '1'.

flexible way without using multiple IF-statements. For this, we precompute a 2D lookup table to contain this logic. The integer-encoded volume and clipping bitflags are used as lookup indices. The output of the table is another integer-encoded bitflag list which specifies for each volume whether to render it or not. Figure 5 shows an example scenario consisting of two volumes and a single clipping shape set for *volume cutting* on volume 0 only.

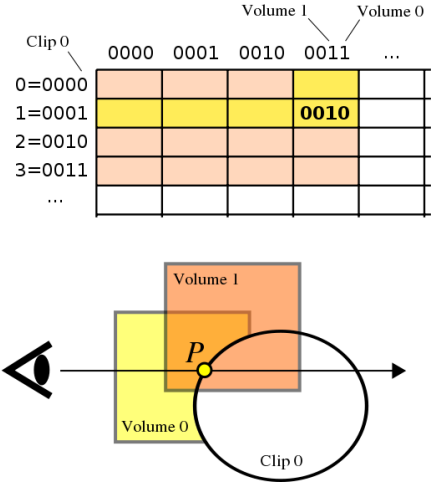

Figure 5: Example *clipInfo* lookup table for a scene consisting of two volumes and one clipping shape set for volume cutting. Point *P* is associated with volume bitflags '0011' and clip bitflags '0001'. Performing lookup in table results in '0010' meaning volume 1 should be rendered, while volume 0 should not be rendered.

3.5 Data Intermixing

One of the most challenging aspects of multi-volume rendering is to decide how to render overlapping volume regions. The difficulty lies not so much in finding different mixing schemes, as in deciding which schemes result in *useful* visualizations. In our algorithm we limit ourselves to offering a number of data mixing options without proposing to solve the general problem. Given *N* volumes inside an intersection, the options we offer are:

- *Priority selection*: Select one volume for rendering based on a *priority ID*.
- *Opacity-weighted average*: Apply transfer function classification and shading to each volume sample, pre-multiply the colors with their associated opacity and compute the average color.
- *2D intersection transfer function*: Apply a 2D transfer function to the intersection between at most two volumes.
- *Intersection color*: Apply a single color and opacity to the intersection region. This can be used for visual collision detection.

Figure 6 illustrates these mixing schemes. We applied them to four abstract cube datasets to clearly convey the difference between each scheme. The 2D intersection transfer function takes gray values from two volumes and returns a color and opacity. In the example, the 2D transfer function returns a yellow color wherever two volume samples have equal value (+/- an offset). We limit ourselves to 1D and 2D transfer functions, however the algorithm could easily be extended to 3D if memory availability allows it. Note that it is far from trivial to define high-dimensional transfer functions [7].

4 Results and Discussion

Our multi-volume rendering algorithm can be used for any application that involves multiple volumetric datasets possibly intersected with geometric surfaces. However, our work was mainly motivated by medical applications that require visualization of information from many different imaging modalities and where resolutions may differ significantly between datasets. Tumor resection planning in neurosurgery is such an application where the surgeon needs to visualize general anatomical context, such as the brain, and specific focus objects such as tu-

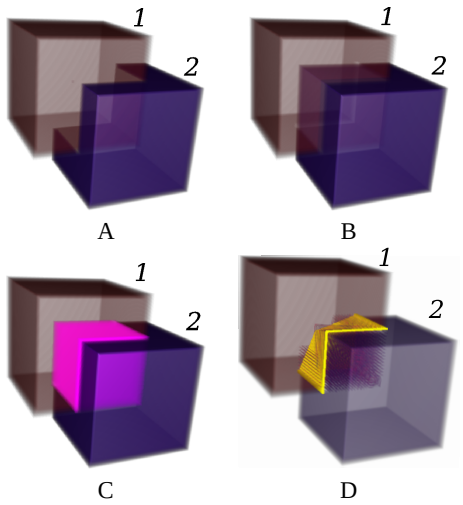

Figure 6: Different intermixing schemes: (A) priority selection where volume 1 has priority, (B) opacity-weighted average, (C) intersection color (magenta) and (D) 2D intersection transfer function which highlights regions where sample values are equal (+/- an offset).

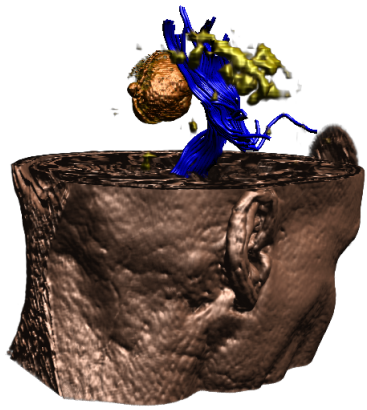

Figure 7: Multimodal view of head, tumor (T1-weighted MRI), cortical activations (fMRI) and fiber tracts (DTI). Brain, tumor and activations are all separate volumes. DTI fiber tracts are visualized as stream tubes.

mor, cortical activations, fiber tracts and surgical tools. Figure 7 illustrates what this could look like using our framework. In this example, different representations are chosen for different objects. The DTI fiber tracts are represented as stream tubes ($128 \times 128 \times 52$, 16 directions) while the tumor and cortical activations are volume data.

Figure 8 shows an example of a CT dataset with one clipping shape used for simulating an access hole in the skull through which a virtual surgical tool is inserted. Another clipping shape is used to offer a free view of the access hole from an alternative viewing direction. Figure 9 illustrates an example of a concave surface model of the brain used as a complex clipping shape to extract the brain as a volume representation. The performance and memory requirements of these scenarios are discussed in the following paragraphs.

Performance: In most scenarios our algorithm runs at interactive framerates (NVIDIA GeForce 8800 GTX, 512×512 window, 1mm stepsize). Depending on scene complexity however framerates may drop significantly. For example, the CT dataset in Figure 8 renders at 12.6 fps which includes a $256 \times 256 \times 225$ CT dataset, a $64 \times 64 \times 64$ tumor dataset and two clipping shapes. The MRI dataset in Figure 7 renders at 2.5 fps and includes a $256 \times 256 \times 200$ MRI dataset, a $64 \times 64 \times 64$ tumor dataset, a $64 \times 64 \times 64$ fMRI dataset, DTI fibers as streamtubes, a surgical tool and a clipping box. This is the most complex scenario we rendered. The MRI dataset in Figure 9 runs at 2.8 fps. and includes the $256 \times 256 \times 200$ MRI dataset and a concave brain model consisting of around 450K polygons. As can be seen the flexibility of our framework comes at a cost. The performance drops when the number of depth layers increases because each depth layers is associated with an additional ray-casting pass.

Memory consumption: Our algorithm requires all volumes to fit inside GPU texture memory. This means that there is a physical limit to the number and dimensions of volumes that can be simultaneously visualized. However, due to our volume-local sampling scheme there is no need for inflating low resolution datasets. This keeps memory consumption at acceptable levels. If 8-bit datasets are used, the multi-volume scenario of Figure 7 requires only 15.4 MBytes (1 x 13Mb for head, 1 x 0.2Mb for tu-

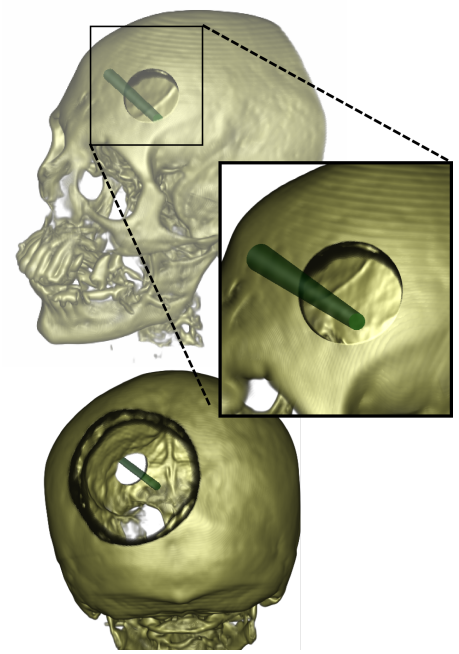

Figure 8: Multiple clipping shapes applied to a CT skull. The larger clipping shape allows an alternative view on the access hole with virtual tool.

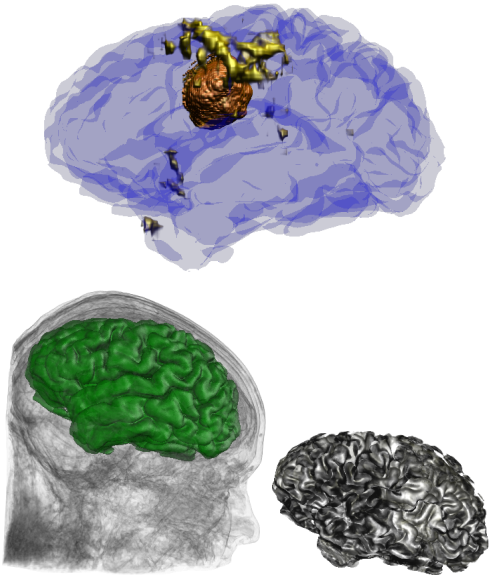

Figure 9: Top: semi-transparent geometric model of brain combined with tumor and f-MRI data. Bottom left: geometric brain model embedded in MRI dataset. Bottom right: brain model used as concave clipping shape to clip away surrounding tissue thereby resulting in a volumetric representation of the brain based on MRI data.

mor, 1 x 0.2Mb for fMRI). For 16-bit datasets this would be twice as much but still well within the limits of GPU memory (768 MBytes).

5 Conclusions and Future Work

Our multi-volume depth peeling algorithm offers a flexible way to visualize multiple, partially overlapping volumes intersected with an arbitrary number of opaque or translucent geometric shapes. Volumes are sampled locally in their native resolution which keeps memory requirements to a minimum. Together with support for multiple, convex or concave clipping shapes this makes our framework especially suitable for neurosurgery applications where volumetric datasets of differing resolutions often need to be combined with geometric data derived from DTI fiber tracking or additional surface models. We also offer several options for rendering the intersection between two or more volumes.

A number of points remain that need to be addressed in the near future. First, performance could be improved by incorporating empty-space skipping techniques and using the depth peeling algorithm only when necessary. Most importantly however, we wish to implement a neurosurgical planning and navigation tool based on this framework and evaluate its usability in a clinical setting.

References

[1] H. Hauser, L. Mroz, G.I. Bischi, M.E. Groeller. Two-Level Volume Rendering. *IEEE Transactions on Visualization and Computer Graphics*, 7(3), 242–252, 2001.

[2] M. Levoy. A Hybrid Ray Tracer for Rendering Polygon and Volume Data. *IEEE Computer Graphics and Applications*, 2(4):33–40, 1990.

[3] W. Cai, G. Sakas. Data Intermixing and Multi-Volume Rendering. *Computer Graphics Forum*, 1999

[4] A. Leu, M. Chen. Modeling and Rendering Graphics Scenes Composed of Multiple Volumetric Datasets. *Computer Graphics Forum*, 159–171, 1999.

[5] K.A. Kreeger, A.E. Kaufman, Mixing Translucent Polygons with Volumes. *IEEE Transactions on Visualization and Computer Graphics*, 191–198, 1999.

[6] J. Hardenbergh, B.R. Buchbinder, S.A.M. Thurston. Integrated 3D Visualization of fMRI and DTI Tractography. *IEEE Transactions on Visualization and Computer Graphics*, 2005.

[7] S. Bruckner, E. Groeller. VolumeShop: An Interactive System for Direct Volume Illustation. *IEEE Transactions on Visualization and Computer Graphics*, 671–678, 2005.

[8] S. Bruckner, E. Groeller. Exploded Views for Volume Data. *IEEE Transactions on Visualization and Computer Graphics*, 2006.

[9] J. Plate, T. HoltKaemper, B. Froehlich. A Flexible Multi-Volume Shader Framework for Arbitrarily Intersecting Multi-Resolution Datasets. *IEEE Transactions on Visualization and Computer Graphics*, 2007.

[10] J. Beyer, M. Hadwiger, S. Wolfsberger, K. Buehler. High-Quality Multimodal Volume Rendering for Preoperative Planning of Neurosurgical Interventions. *IEEE Transactions on Visualization and Computer Graphics*, 2007.

[11] D. Weiskopf, K. Engel, T. Ertl. Interactive Clipping Techniques for Volume Visualization and Volume Shading. *IEEE Transactions on Visualization and Computer Graphics*, 2003.

[12] A. Mammen. Transparency and Antialiasing Algorithms Implemented with the Virtual Pixel Maps Technique. *IEEE Computer Graphics Applications*, 43–55, 1989.

[13] M. Hadwiger, C. Berger, H. Hauser. High-Quality Two-Level Volume Rendering of Segmented Data Sets on Consumer Graphics Hardware. *Proceedings of IEEE Visualization,*, 2003

[14] M.A. Termeer, J. Olivan Bescos, A.C. Telea. Preserving Sharp Edges with Volume Clipping. In *Proceedings of Visualization, Modeling and Vision,*, 2006.

[15] T. Schafhitzel, F. Roessler. Simultaneous Visualization of Anatomical and Functional 3D Data by Combining Volume Rendering and Flow Visualization. In *SPIE Medical Imaging 2007*, 2007.

[16] F. Roessler, E. Tejada, T. Fangmeier, T. Ertl. GPU-based Multi-Volume Rendering for the Visualization of Functional Brain Images. In *Proceedings of SimVis*, 315–318, 2006.

[17] S. Grimm, S. Bruckner, A. Kanitsar, E. Groeller. Flexible Direct Multi-Volume Rendering in Interactive Scenes. In *Vision, Modeling, and Visualization*, 386–379, 2004.

[18] T. Hendler, P. Pianka, M. Sigal, M. Kafri. Two are Better Than One: Combining fMRI and DTI-based Fiber Tracking for Effective Pre-Surgical Mapping. In *Proceedings of International Society for Magnetic Resonance Medicine*, 2003.

[19] B. Wilson, E. Lum, K. Ma. Interactive Multi-Volume Visualization. In *Workshop on Computer Graphics and Geometric Modeling*, 2002.

[20] M. Ferre, A. Puig, D. Tost. Rendering Techniques for Multimodal Data. In *Proc. SIACG 2002 1st Ibero-American Symposium on Computer Graphics*, 305–313, 2002.

[21] I. Manssour, S. Furuie, L. Nedel. A Framework to Visualize and Interact with Multimodal Medical Images. In *Volume Graphics*, 385–398, 2001.

[22] K.J. Zuiderveld, M.A. Viergever. Multi-modal Volume Visualization using Object-Oriented Methods. In *Proceedings of the 1994 symposium on volume visualization*, 59–66, 1994.

[23] D.R. Nadeau. Volume Scene Graphs. In *Proceedings of the IEEE Symposium on Volume visualization*, 49–56, 2000.

[24] A. Ghosh, P. Prabhu, A.E. Kaufman, K. Mueller. Hardware Assisted Multichannel Volume Rendering. In *Proceedings of Computer Graphics International*, 2003.

[25] P.J. Diefenbach, N.I. Badler. Multi-pass Pipeline Rendering: Realism for Dynamic Environments. In *Proceedings of the Symposium on Interactive 3D Graphics*, 59–ff, 1997.

[26] A. Koehn, F. Weiler, J. Klein, O. Konrad, H.K. Hahn, H.-O. Peitgen. State-of-the-Art Computer Graphics in Neurosurgical Planning

and Risk Assessment. In *Proceedings of Eurographics*, 2007.

[27] S. Stegmaier, M. Strengert, T. Klein, T. Ertl. A Simple and Flexible Volume Rendering Framework for Graphics-Hardware-based Raycasting. In *Proceedings of Volume Graphics*, 2005.

[28] R. Westermann, T. Ertl. Efficiently Using Graphics Hardware in Volume Rendering Applications. In *Proceedings of the 25th SIGGRAPH*, 1998.

[29] J. Krueger, R. Westermann. Acceleration Techniques for GPU-based Volume Rendering. In *Proceedings of IEEE Visualization*, 2003.

[30] D. Borland, J.P. Clarke, J.R. Fielding, R.M. Taylor II. Volumetric Depth Peeling for Medical Image Display. In *Proceedings of SPIE Visualization and Data Analysis*, 2006.

[31] K. Engel, M. Hadwiger, C. Rezk-Salama, J.M. Kniss. *Real-Time Volume Graphics.* A.K. Peters, 2006.

[32] B. Preim, D. Bartz. *Visualization in Medicine.* Morgan Kaufmann Publishers, 2007.

[33] C. Everitt. Interactive Order-Independent Transparency. *Technical report* NVIDIA OpenGL applications engineering, 2001.

[34] A. Koenig, H. Doleisch, E. Groeller. Multiple Views and Magic Mirrors - fMRI Visualization of the Human Brain. *Technical report* Vienna university of technology, institute of compute graphics, 1998.

[35] K. Eide. GPU-Based Transparent Polygonal Geometry in Volume Rendering. *Technical report* Norwegian University of Science and Technology, 2005.