A Flexible Multi-Volume Shader Framework for Arbitrarily Intersecting Multi-Resolution Datasets

John Plate, Thorsten Holtkaemper, Bernd Froehlich

Abstract—We present a powerful framework for 3D-texture-based rendering of multiple arbitrarily intersecting volumetric datasets. Each volume is represented by a multi-resolution octree-based structure and we use out-of-core techniques to support extremely large volumes. Users define a set of convex polyhedral volume lenses, which may be associated with one or more volumetric datasets. The volumes or the lenses can be interactively moved around while the region inside each lens is rendered using interactively defined multi-volume shaders.

Our rendering pipeline splits each lens into multiple convex regions such that each region is homogenous and contains a fixed number of volumes. Each such region is further split by the brick boundaries of the associated octree representations. The resulting puzzle of lens fragments is sorted in front-to-back or back-to-front order using a combination of a view-dependent octree traversal and a GPU-based depth peeling technique. Our current implementation uses slice-based volume rendering and allows interactive roaming through multiple intersecting multi-gigabyte volumes.

Index Terms—Multi-volume visualization, constructive solid geometry, shading, display algorithms.

1 INTRODUCTION

Modern data acquisition and simulation methods can generate multiple huge volume data sets representing different attributes or temporal snapshots of a single object or region in space. These data sets often have different resolutions and different coordinate systems, which are spatially overlapping, but not spatially aligned. Resampling these volumes onto a single multi-attribute grid is often not desirable due to many possible reasons, e.g. only partial spatial overlapping volumes, pre-processing time, numerical inaccuracies or data bloat if the original resolutions are quite different.

We developed a real-time rendering framework for dealing with multiple multi-resolution volume datasets, which are spatially overlapping. These datasets are explored using polyhedral lenses, which are associated with one or more volumes (see figure 1). Our geometry pipeline splits each lens into multiple homogenous volumetric regions. Each such region is inserted into the associated octree-based multi-resolution representations and further split at the brick boundaries. The resulting puzzle of lens fragments is sorted in front-to-back or back-to-front order using a combination of a view-dependent octree traversal and a GPU-based depth peeling technique. The rendering technique for overlapping volumes is specified by our graphical shader composer interface, which generates individual shader code for intersecting volumetric regions.

Our work is motivated by oil and gas visualization applications, where datasets are typically very large – in most cases exceeding the memory on the graphics card and even the main memory. In the past, multi-resolution out-of-core techniques have been developed to deal with such demanding requirements, e.g. [7][8]. However these approaches do not deal with multiple intersecting volumes, which are becoming more common. One example are seismic surveys which may be performed in regular intervals in oil producing regions to estimate the topological changes due to the exploitation and to plan new wells. Due to advancements in acquisition and processing technologies, more recent data sets are generally of much higher resolution and may also reach deeper into the earth and cover a larger area than older surveys. In addition different acquisition technologies provide attributes at differing resolutions. Ideally all the acquired information should be available to the geo-scientists in its best representation.

The two main contributions of our work are an approach for multi-volume rendering for arbitrarily intersecting multi-resolution datasets and a flexible shader framework for specifying the rendering behavior in overlapping volume regions. We describe our rendering pipeline, which requires extensive slicing and cutting operations on the volume lens geometries down to the brick level of the multiple multi-resolution volume representations. Our shader composer is a powerful interactive visual editor for specifying the rendering behavior for multiple overlapping volumes. The shader composer generates individual shader programs for each homogeneous volume region consisting of one or more volumes. Thus mono and multi-volume regions are each handled efficiently. Our implementation confirms that real-time rendering of multiple large volumes using complex compositing behavior has become possible on modern graphics processing units.

2 RELATED WORK

In this section we will focus on previous work aimed at multi-volume rendering and in particular on contributions that deal with intersecting volume datasets.

Most work on multi-volume rendering was done in the field of medical visualization. After spatial registration multi-modal volume datasets are visualized simultaneously to provide the advantages of the different modalities in a single picture. Several publications deal with the task of merging the different volume data sources. In [1] Jacq and Roux introduce a multi-volume ray casting algorithm that merges for each sample position the sample values for every volume by applying a maximum, minimum, or an average operator. Similar to that Wilson at el. propose in [2] so-called data fusion schemes for hardware accelerated slice-based volume rendering, which either alternately sample the different volume data sets, or combine for each sample position the weighted sample values, or distribute them into different color channels. A detailed proposal of several volume intermixing schemes is given by Cai and Sakas in [3]. The proposed methods are divided into the three categories of image, illumination and accumulation level intermixing, which define different data merging points in a ray casting multi-volume rendering pipeline. In [4] Rößler et al. introduce an interactive slice-based volume rendering frame work for multiple volumes, which intermixes slices of volumes by depth sorting. Each volume is assigned its own GPU shader, which is applied to all fragments of the associated slices.

- *John Plate is with Fraunhofer IAIS and Bauhaus-Universität Weimar, E-Mail: john.plate@iais.fraunhofer.de.*
- *Thorsten Holtkaemper is with Fraunhofer IAIS, Sankt Augustin, E-Mail: thorsten.holtkaemper@iais.fraunhofer.de.*
- *Bernd Froehlich is with Bauhaus-Universität Weimar, E-Mail: bernd.froehlich@medien.uni-weimar.de.*

Manuscript received 31 March 2007; accepted 1 August 2007; posted online 27 October 2007. Published 14 September 2007.
For information on obtaining reprints of this article, please send e-mail to: tvcg@computer.org.

So far all approaches only deal with spatially aligned volume datasets, which could be also considered as a single multi-attribute dataset. In contrast Nadeau [5] introduces the concept of a volume scene graph, which can contain multiple volume datasets and space-filling functions at arbitrary positions in space. Multiple volumes can intersect in arbitrary ways. They are combined by group nodes, applying a composition function, which can be chosen from imaging, constructive solid geometry (CSG), and math operators. For rendering the volume scene graph it is evaluated for all voxel positions on a world volume grid. This evaluation is costly and needs to happen every time the positions of the volume change with respect to each other. The resulting regular volume can be displayed by any conventional volume rendering algorithm for single volumes. No multi-resolution or out-of-core functionality is supported.

Grimm et al. [6] developed a parallel CPU-based ray casting algorithm for multiple non-aligned volume objects, which they call V-Objects. This approach is closest in functionality to our approach. It is discussed in detail in section 3.3 once more details about our algorithm are introduced.

Our work is based on Octreemizer [7], a 3D-texture-based approach for interactive visualization of very large volumetric datasets. Octreemizer uses a two-level predictive paging approach (hard disk to main memory and main memory to texture memory) with several tiled (bricked) and hierarchically arranged resolution levels. It allows users to roam through large volumetric datasets in real-time with low memory requirements. Octreemizer can only handle a single multi-resolution volume including out-of-core functionality. For multi-volume rendering we use some of the original Octreemizer functionality to manage individual volumes and we prepare the required proxy geometry for slice-based direct volume rendering using our extended geometry pipeline described in the following section.

3 Geometry Pipeline

Octreemizer uses volume and lens objects to define regions of interest. The geometry pipeline trims, cuts and slices the lenses as shown in figure 2 to generate the proxy geometry needed for sliced-based rendering. A first step trims the lenses with respect to the view frustum and different volume boundaries. Then homogeneous lens sections are created, which contain a fixed number of volumes. These regions are inserted into the multi-resolution octree representations described in [7] by further splitting them at the brick boundaries. Finally, the resulting lens fragments are sorted in front-to-back or back-to-front order and composited by slice-based rendering.

3.1 Volumes and Lenses

A volume contains the bricked multi-resolution volume data in an octree hierarchy and information about its position, orientation and size. A lens consists of one ore more convex polyhedrons, which define the regions of the volume data that will be displayed. A lens may also contain convex polygons, which define slices through the volume. We will not further discuss these polygonal slices, though they are supported in the entire geometry pipeline.

Lenses have associated states including:

- *Transformation:* Position, orientation and size. Can be "locked" to other volumes or lenses in order to group and simultaneously drag multiple objects.
- *Transparent Mode:* Toggles whether the lens should be completely opaque or the transparency information from the shader stage should be used.
- *Slice Distance:* The slice distance for slice-based rendering.

Each lens can be associated with a single or multiple volumes. The areas of the associated volumes intersecting the lens will be rendered. The lens in figure 2a has been associated with two volumes and it intersects both. Lenses without associated volume may be used as clip lenses in order to geometrically clip away intersections with other lenses. Of course a lens may contain only a single polyhedron, which entirely encloses all associated volumes. As expected this configuration renders the complete volume data sets.

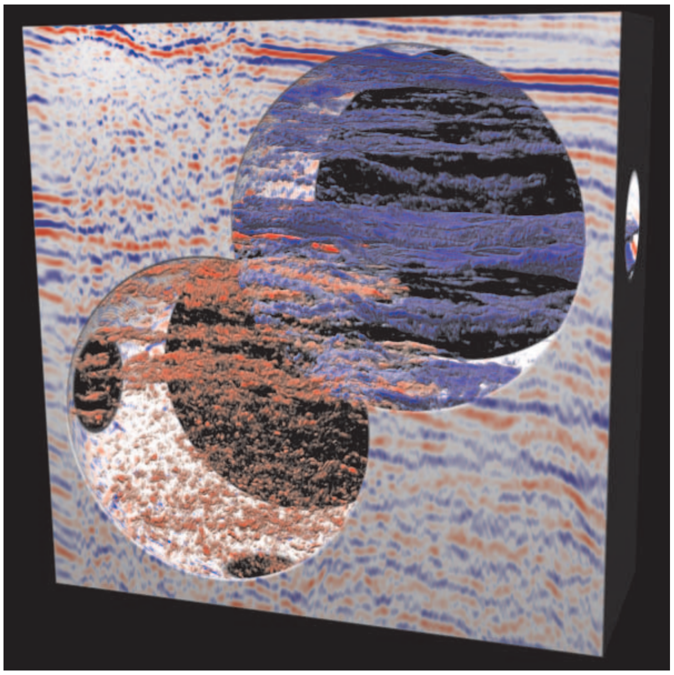
Fig. 1. A seismic dataset (512 x 512 x 512 voxels) is overlapping with two high resolution clip volumes containing a sphere. Inside the spheres additionally different transfer functions are applied to emphasize low and high seismic amplitudes (blue and red) in the intersected regions. If the spheres overlap, low and high amplitudes are both visible. Phong lighting with two light sources is applied.

3.2 Clipping at View Frustum and Volume Boundaries

In a first step the lenses are clipped against the view frustum and the volume boundaries. We create these "trimmed" lenses for each combination of a lens and an associated volume. The polyhedrons of a lens are transformed by the lens' transformation matrix and clipped at the view frustum and the volume boundaries. The resulting polyhedral parts residing in a single volume define the trimmed lens. Figure 2b shows two trimmed lenses created from one lens that intersects two volumes. The trimmed lenses are updated only if necessary, e.g. when the relative transformation between the source lens and the volume changes.

3.3 Considering Overlapping Volumes

Now that we have the parts of the lenses residing inside each associated volume we have to identify the lens regions where multiple volumes intersect. We generate separate convex polyhedrons, which represent regions with a fixed set of overlapping volumes.

Our lenses are similar to Grimm et al.'s [6] V-Objects which are associated with a volume data source and comprise visual properties and a transformation matrix. They use a parallel software ray casting scheme for rendering multi-volume scenes without out-of-core functionality or multi-resolution support. Their rendering approach is speed up by decomposing each ray into mono-volume and multi-volume segments. The mono-volume segments of multiple rays are processed in brick-wise order for cache efficiency. Multi-volume segments sequentially sample and blend all associated volumes. Our approach uses hardware-accelerated slice-based rendering to render multiple volumes including out-of-core functionality for each volume. We split the volumes – instead of individual rays – into homogenous regions containing a particular subset of the considered volumes. Since we use a multi-resolution approach, this splitting has to be also considered within the multi-resolution hierarchy of each contributing volume creating brick fragments at different octree resolutions. Grimm et al. use ray intersections to identify mono- and multi-object regions. This approach has several disadvantages in our context:

- The ray entry and exit points are calculated using octree projections. This is efficient for GPU-based ray casting, but in this case the points are additionally needed by the CPU for a depth sort, which would require expensive read backs from the graphics board.
- A depth sort of all entry and exit points is necessary to determine the mono- and multi-object segments. Complex scenes can contain millions of those points.
- The algorithm cannot be easily adapted to other rendering techniques than ray casting.

Also, it would have been possible to integrate existing libraries, like CGAL [9], and to use CSG algorithms, like Boolean operations on nef polyhedrons [10][11]. These operations are flexible, powerful and stable, but also much slower than our algorithm, which does not need this flexibility and thus benefits from optimizations that depend on specific constraints, e.g. simplified data structures, convex polyhedrons, no open boundaries, no dangling facets. We compared our algorithm with CGAL by measuring the time needed to calculate the Boolean intersection and differences of two overlapping cubic polyhedrons on an Intel Core 2 Duo E6600. For different orientations, our algorithm needed between 40 and 90 microseconds, while CGAL needed between 90,000 and 820,000 microseconds.

3.3.1 Lenslets

Lenslets are homogenous convex polyhedral regions, which are associated with a fixed set of volumes. Lenslets do not intersect each other. They are generated by intersecting the polyhedrons of the trimmed lenses associated with different volumes with each other (see figure 2c) followed by additional split operations to guarantee convexity. If only spatially separated lenses would be used to define how the associated volumes are rendered, it would be sufficient to find the intersections between the trimmed lenses of each source lens. However, we decided to support overlapping lenses and compute the intersections between all existing trimmed lenses and allow users to define the rendering behavior in these overlapping lens regions. We use the following algorithm to compute the lenslets:

```
lenslets.remove(outdated lenslets)
FOREACH lens IN new or updated trimmed lenses
{
  test_lens_set = lens
  WHILE intersect_lens = lenslets.
    find_intersecting_lens(test_lens_set)
  {
    lenslets.add(test_lens_set ∩ intersect_lens)
    lenslets.add(intersect_lens - test_lens_set)
    lenslets.remove(intersect_lens)
    test_lens_set = test_lens_set - intersect_lens
  }
  lenslets.add(test_lens_set)
}
```

Figure 2c shows an example of the results of the Boolean operations used in this algorithm. It also shows that the Boolean difference can be non-convex. Figure 2d demonstrates how additional cuts ensure convex polyhedrons, which are required by our geometry pipeline. These operations are not explicitly mentioned in the above algorithm for simplicity. Additionally, our implementation does not generate the two Boolean differences and the intersection in three separate steps. Instead, we use an intersection operation that creates additionally the two Boolean differences by intersecting each polyhedron with each face plane of the intersecting polyhedron.

The lenslets contain references to their "parent" trimmed lenses, from which they have been constructed. Additionally, the trimmed lenses reference their "child" lenslets and all the intersected trimmed lenses. With these references, it is possible to find and remove all outdated lenslets, which is necessary if a trimmed lens has been updated (e.g. moved around). Thus only a minimum number of lenslets have to be reconstructed during interactive operations.

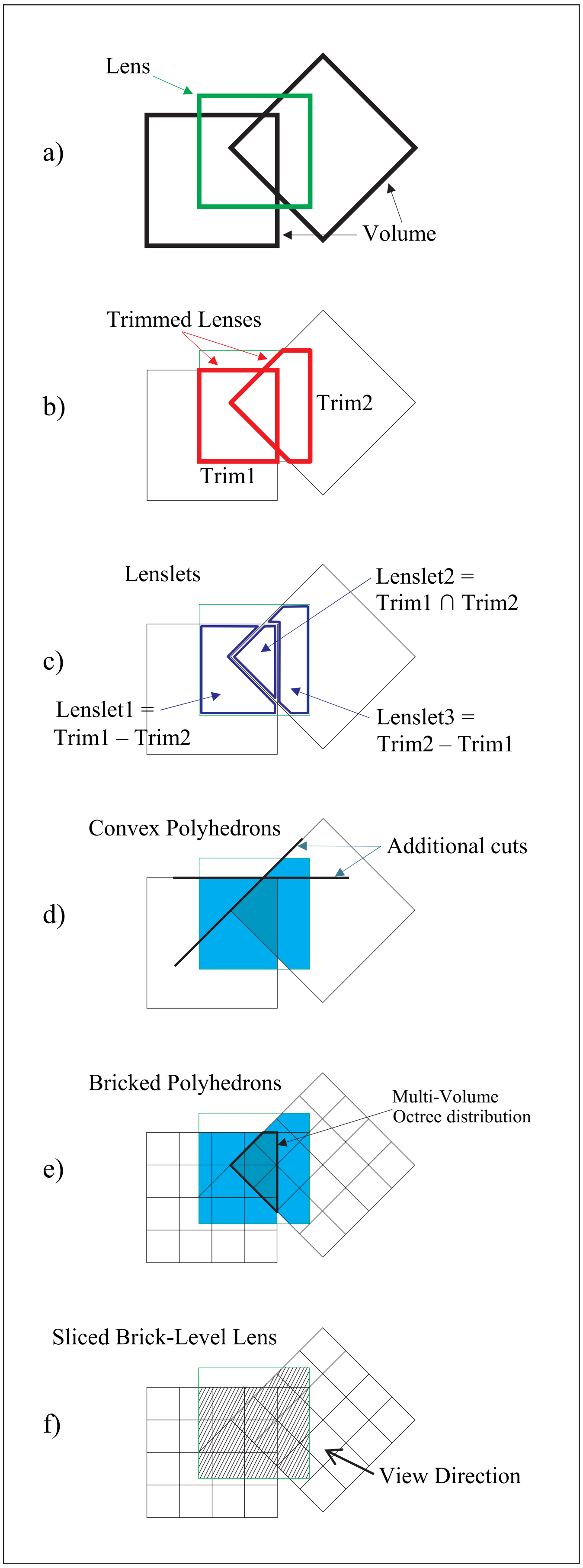

Fig. 2. Geometry pipeline

3.3.2 Intersection Details

We use the Sutherland-Hodgman polygon clipping algorithm [12] to efficiently construct Boolean intersections and differences of polyhedrons. Our implementation uses the following objects for efficient and numerically stable handling of these operations:

- *Vertex:* A set of three coordinates.
- *Vertex Pool:* A set of vertices.
- *Plane:* Parameters from a plane equation.
- *Plane Pool:* A set of planes.
- *Polygon:* A plane index and a set of vertex indices.
- *Polyhedron:* A set of polygons and a bounding box.

We construct planes from every source polygon in the scene (faces of polyhedrons, volume data sets and view frustum) and store them in a plane pool. The polygons that are used and constructed during the intersection stage contain a reference to coplanar planes. Additionally, the polygons generated from a polyhedron contain a reference to their outward facing normal.

The plane pool has two advantages. First, it speeds up the intersection checks. If two polygons of two different polyhedrons reference the same plane, we know immediately, that each of the two polygons does not intersect with the other polyhedron. Additionally, if the normals are facing in different directions, the associated polyhedrons themselves do not intersect each other. The second advantage concerns numerical problems: Without the plane references, intersection calculations of two contacting polyhedrons would often fail due to limited floating-point precision.

3.4 Bricking Lenslets

This stage is required to support bricked volumes. The lenslets have to be cut further at the brick boundaries, because the rendering stage accesses only a single brick (3D texture) of a volume at a time. We call the brick-sized lens polyhedrons lens fragments.

3.4.1 Recursive Octree Insertion

The lenslets maintain references to their source trimmed lenses and therefore their associated volumes. We add each lenslet to the associated volume object of the first trimmed lens: The lenslet polyhedron is transformed to the coordinate system of the volume and added to the root brick of the octree hierarchy. Then they are recursively inserted into the child bricks. During this insertion, the polyhedrons are split at the brick boundaries. Based on the Sutherland-Hodgman polygon clipping algorithm, we have developed an algorithm which efficiently splits polyhedrons simultaneously at three orthogonal axis-aligned planes. The basic idea is to map the vertices directly to octants instead of half-spaces, and to create a new polyhedron from all vertices in each octant and the line intersections with the octant boundaries. The split polyhedrons from the eight octants are added to the child bricks. Bricks without polyhedrons (and their children) will not be considered during the rendering stage.

3.4.2 Dealing with Multiple Octrees

If a lenslet references multiple volumes, we have to split it at the brick boundaries of all these volumes. The marked lenslet in figure 2e is an example for this case. Cutting at brick boundaries of multiple volumes is an expensive operation, but it is necessary, because every part of the resulting "puzzle" has to be rendered with a different combination of active bricks.

We start with the brick-level lens fragments from the insertion of the lenslets into the first volume. During the draw traversal of the first octree, the polyhedrons in each brick are transformed to the coordinate system of the next volume, added to its root brick and inserted into its octree. This is repeated for all volumes associated to a lenslet. We perform these second level traversals (and the proxy geometry generation described in the next section) during the rendering stage, interleaved with the brick rendering traversal. This way the CPU is able to prepare the geometry of a brick, while the geometry of the previous brick is still rendered by the GPU.

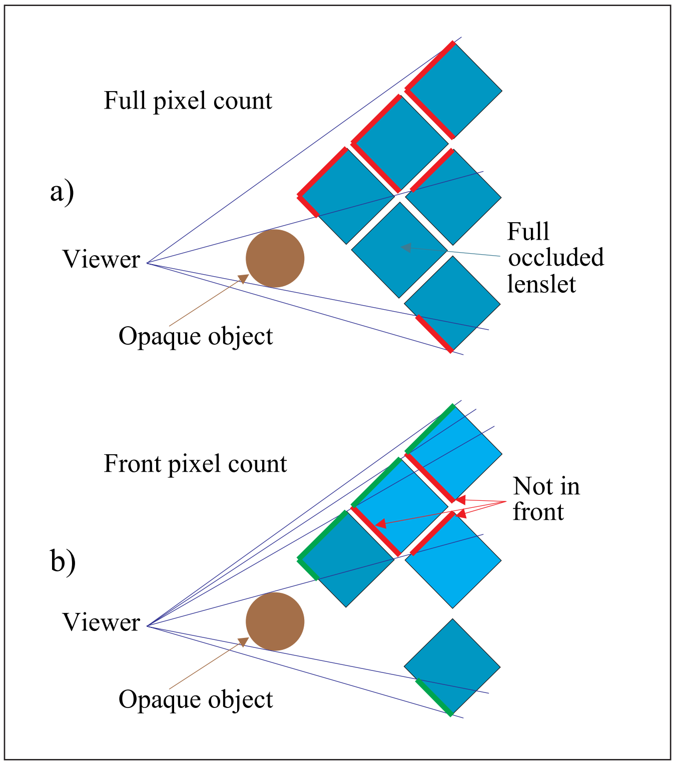

Fig. 3. Depth peeling

```
disable writing to color buffer
enable depth test
disable writing to depth buffer
enable back face culling
create unsorted list of all polyhedrons
draw unsorted list and count pixels unoccluded
  by opaque scene for each polyhedron
  (full pixel count, figure 2a)
remove all fully occluded polyhedrons
WHILE unsorted list has more than one element
{
  copy main depth to separate depth buffer
  activate separate depth buffer
  enable writing to separate depth buffer
  draw unsorted list
  disable writing to depth buffer
  set depth test function to EQUAL
  draw unsorted list and count pixels of each
    polyhedron (front pixel count, figure 2b)
  reset depth test function to LESS
  FOREACH polyhedron in unsorted list
  {
    IF front pixel count == full pixels count
    {
      remove polyhedron from unsorted
        and append to sorted list
    }
  }
  IF no geometry was moved
  {
    remove most visible polyhedron from
      unsorted and append to sorted list
  }
  activate main depth buffer
}
append unsorted list to sorted list
enable writing to color buffer
```

Fig. 4. Depth peeling algorithm

3.5 Proxy Geometry

At this point, we have polyhedrons at the brick level, or intersected brick level for multiple volumes. We now create the proxy geometry needed by the selected rendering technique.

In case of ray casting [13], the polyhedrons can be used directly to obtain the ray entry and exit points. In case of slice-based volume rendering (SBVR) [14], which we use in our current implementation, we create slices orthogonal to the view direction and clip them at the polyhedron faces (see figure 2f). The slice distance is controlled by the sample distance setting of the associated lens. If two contacting polyhedrons have the same slice distance, the slice offset is also the same to avoid rendering artifacts at the border.

3.6 Sorting

For correct rendering results the transparent parts of the scene have to be sorted. The order (back-to-front or front-to-back) depends on the selected rendering and compositing technique.

The polyhedrons at brick level are sorted on-the-fly during the view-dependent recursive octree traversal. For perspective projections we simply identify the octant which contains the viewer. For orthogonal projections we identify the octant which contains the vector which originates at the brick center and points in direction of the positive Z-axis in eye space coordinates. The polyhedrons of the child brick in this octant have to be rendered last (or first). The other child bricks are sorted by their L1-distance.

Sorting the polyhedrons at the lenslet level is a more difficult task. Since simple sort by viewer distance is not sufficient, we have developed a GPU-based depth peeling technique to reliably find an occlusion-preserving rendering order. We use a separate depth buffer and OpenGL occlusion queries for the algorithm shown in figure 4. It counts the front pixels of each polyhedron (see figure 3b) and removes the front layer of polyhedrons which are not occluded by other polyhedrons in each step. The sequence of layers forms a front-to-back sorted list.

As a side effect, we get free occlusion culling, because we have to initialize the full pixel count for each polyhedron as shown in figure 3a and we always initialize the depth buffer with the main depth buffer, which holds the depth information of the opaque parts of the scene. We do not draw polyhedrons with a pixel count less than a given threshold.

As the polyhedrons do not intersect each other, there is theoretically always at least one in front. Due to limited floating point precision, it is possible that there is none. In this case we find the most visible polyhedron (highest ratio of front pixel to full pixel count) and move only this one to the sorted list.

4 Multi-Volume Shader Framework

Rendering a region with multiple intersecting volume data sets requires defining a data compositing technique. In this section we describe the techniques we have developed to overcome the limitations of commonly used techniques, such as:

- *Interleaved slices:* View-aligned slices are created independently for each volume and blended to the framebuffer in an interleaved order [4]. The technique introduces opacity errors and does not allow combining the sampled values.
- *Color channels:* The intensity values of up to three volumes are combined by using the color channels red, green and blue [2]. Only scalar transfer functions can be used.

4.1 Interactive Shader Composer

We developed an interactive shader composer, which allows creating special purpose GPU-based shader programs by interactively plugging data flow widgets together using a graphical user interface (GUI). Figure 5 shows an example configuration of a shader composer. Of course multiple shader composers can exist simultaneously.

4.1.1 Shader Composer GUI

The GUI consists of data flow widgets, called nodes, which have input fields on the top and/or output fields on the bottom. There are single-component fields, e.g. red, green, blue, alpha (opacity), and multi-component fields, e.g. RGBA, XYZW or gradient vector. Field connections are defined by drag and drop operations, which connect compatible fields. A connection from a single to a multi-component field is possible, but not vice versa as it would be ambiguous. The field connections are displayed as red or green lines. The color indicates if the data flow is valid. For example, the data flow from the “Inverse” node in figure 5 is invalid, because it has no input connection. If at least one green line reaches the output node, the shader is “complete” and can be used by the rendering stage.

The composer GUI consists initially of a “ColorOut” node, which represents the final combined sample value. The user connects the composer to one or more lenses, which are described in section 3.1. Based on these connections, the composer creates input nodes: One lens node with a statically connected volume node for each volume which is associated with a connected lens. The latter is necessary because a single lens can have multiple associated volumes, and the lens node output fields are needed for every volume. The lens nodes are required if the user wants to create different shaders for different regions of a volume or in conjunction with other lenses.

4.1.2 Basic Shader Composer Nodes

The basic input/output composer nodes are:

- *ColorOut:* The RGBA value of this node is passed to the active fragment shader program.
- *Volume:* Represents a volume associated with one or more connected lenses.
- *Lens:* Represents a lens connected to the composer. It has a statically connected volume node and offers an RGBA output field for the sampled volume element of the connected volume. Additionally, there are fields for single components.

4.1.3 Extensible Shader Composer Nodes

The following composer nodes can be used to define the data flow of the shader program. The node class hierarchy is modular and can be easily extended.

- *Palette:* A one or two dimensional transfer function.
- *Lookup:* Performs a color lookup in the connected palette.
- *Gradient:* Calculates a gradient vector on-the-fly.
- *Light:* An illumination model and parameters.
- *Lighting:* Applies one ore more lights to the input color.
- *Operator:* The user can switch between blend, product, sum, minimum, maximum, inverse and negative, and between 1, 2, 3 or 4 components.
- *Constant:* Outputs a single static value.

Most of the selectable operations of the operator node are self-explanatory. The operation that we call blend has been described as “inclusive opacity” by Cai and Sakas [3] for two volumes. It sums up the accumulative effect caused by the opacities from multiple volumes and applies it as the opacity to the current point. The resulting opacity and intensity for n volumes is:

$$opacity = 1 - \prod_{k=1}^{n} 1 - opacity_k$$

$$\text{intensity} = \frac{\sum_{k=1}^{n} \text{opacity}_k \text{ intensity}_k}{\sum_{k=1}^{n} \text{opacity}_k}$$

Another operation that we call “inverse” calculates:

$$inverse = 1 - operand$$

This inverts the numerical range [0, 1] and is useful to invert color and opacity values.

4.1.4 Recursive Shader Generation

For each lenslet that should be rendered, we need a shader program that matches with the intersected lenses referenced by the lenslet. To generate this program, we first identify a shader composer that can be used for the lenslet. This is the case if a composer has connections to every lens from which the lenslet has been constructed.

Second, the shader composer generates a unique shader code identifier depending on the defined data flow and the combination of intersected lenses. The composer requests the shader code identifier from the ColorOut node. Though the composer visualizes the data flow from the input nodes to the output node, the resulting shader code identifier is generated using the opposite direction. The ColorOut node initiates a recursive traversal back to the input nodes to determine the branches with valid data flow. In a second traversal along the valid data flow, each node adds a specific identifier fragment to the code identifier.

In the third step, we search in a list of cached fragment shader programs for the shader code identifier. If it is not found, we create new fragment shader code which is well-defined by the code identifier, and compile a new fragment shader program, which is added to the list of cached programs. Thus the shader code identifier avoids the repeated generation of fragment shader code for each lenslet in every frame.

Compiling shader programs on-the-fly is an expensive operation, but a single flexible fragment shader program could be too complex for current GPUs and would be much slower than the specialized programs. It is also impossible to prepare and store all possible specialized programs, because their number is infinite. Though compiling a shader program needs up to 50 ms, the user typically does not notice it while paying attention to the shader composer.

4.2 Multi-Volume Illumination

Our implementation supports the Phong lighting model to illuminate volumes with one or more light sources. We use the normalized negative opacity gradient as the surface normal for the diffuse and specular reflection.

In the shader composer GUI, the user creates a light node to specify the parameters for the Phong lighting. The light node has a gradient input field, from which it calculates the surface normal. The gradient field is usually connected to a gradient node, which can be connected to any single component output field in the data flow. In figure 5, it is connected to a sum node. During the recursive shader generation, the gradient node adds a flag to the traversal, indicating that it does not need the data value at the current location, but the data values at six axis aligned offset locations. From these six values, the resulting shader program will calculate the central differences and the gradient.

The gradient node enables the user to determine the gradient at any point in the data flow. If users combine one or more volumes with one or more operations like blend, product or transfer function lookup, they might want to apply lighting using the gradient of the effective opacity. As these operations are defined interactively, it is not possible to precalculate these gradients.

Finally, a lighting node applies the illumination of one or more light nodes to the connected color data flow. The lighting node in figure 5 applies two lights.

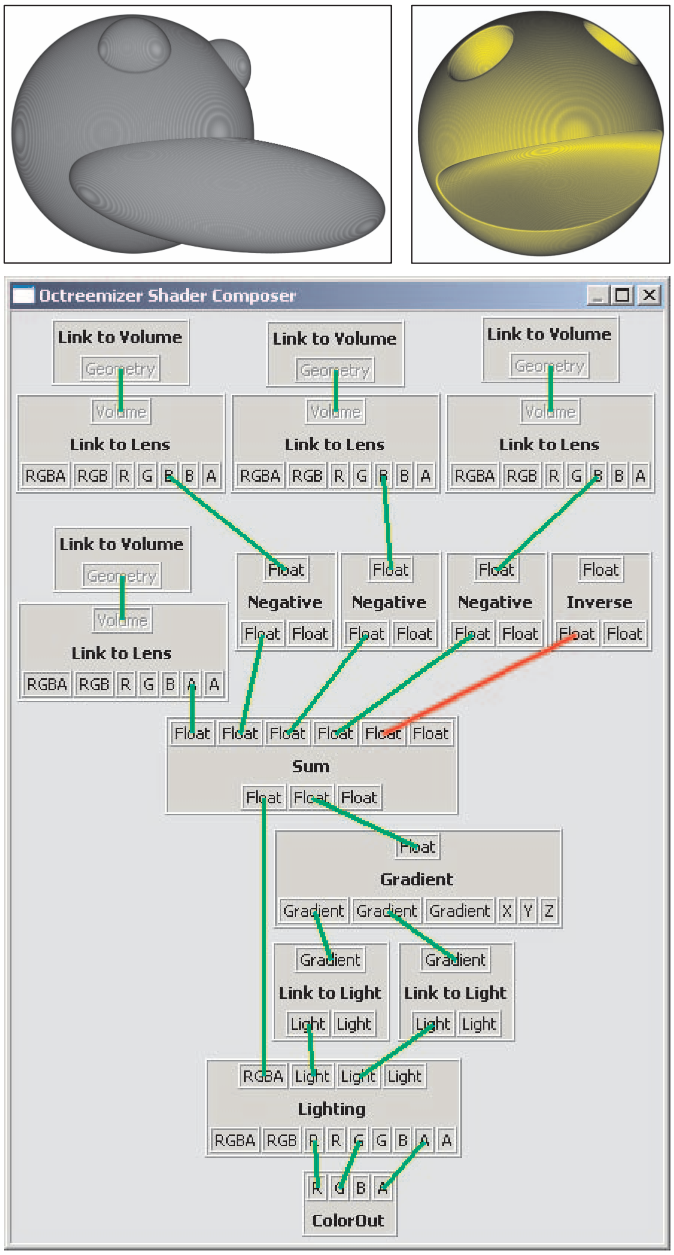

Fig. 5. A shader composer example. Top left: Four overlapping sphere volumes; moved, resized and stretched. Two of them have resolutions of 256 x 256 x 256 voxels and the small two have 64 x 64 x 64 voxels. Top right: The same volumes with enabled shader composer. Bottom: The shader composer GUI. Three volumes are subtracted from the biggest sphere volume, thus they act as clipping volumes. Two lights are applied to the resulting sum. The final shape is yellow because the blue color component is not connected to the ColorOut node.

4.3 Default Shader Composer

We maintain a default shader composer, which is automatically connected to every existing lens. If no user-defined composer matches with the lenses referenced by a lenslet, the default composer will be used. Thus the user does not have to create composers for trivial and simple shader configurations.

Initially, the default composer renders single volumes using opacity-based compositing without any additional operations. Regions with multiple intersecting lenses are combined by a blend operation. We have selected it as the default composition technique, because it produces a kind of blending that most users would expect for overlapping transparent volumes.

The lenses described in section 3.1 have additional states to influence their operations in the default composer. It is possible to add a transfer function, which will be applied to the lens prior to the combination with intersecting lenses. It is also possible to select another operation than blending for the composition of lens regions. The order of these operations is pre-defined. The default composer is additionally able to add lights after the combination operations. If the functionality of the default composer is not sufficient, e.g. users need additional operations or a different order of operations, a user-defined composer is required.

	Geometry	Separate	Multi-Volume	Lighting
Figure 1	25.5 ms	11.3 ms	12.9 ms	78 ms
Figure 5	10.2 ms	4.2 ms	4.9 ms	29 ms
Figure 6	5.2 ms	4.5 ms	5.3 ms	35 ms
Figure 7	226 ms	11.1 ms	29.4 ms	99 ms

Table 1. Timings of some figures.

5 Results and Discussion

Our multi-volume shader framework can be used in many ways and in different domains. We will focus on some basic techniques and describe some domains which could benefit form our work.

One of the basic techniques is volume clipping. Volumes can be clipped with arbitrarily shaped clip objects, if they are voxelized as proposed by Weiskopf et al. [15]. An example is shown in Figure 5. This technique can be easily extended using the shader composer. For example, only selected color channels could be clipped away. Figure 1 demonstrates another extension. We use additionally the inverted sample value of the clip object volume to apply different shader behavior for the in- and outside of the clip object. Thus it is possible to emphasize regions with different transfer functions.

Opacity-based volume composition is a visually intuitive technique. It is useful for spatially overlapping volumes with unrelated data. Our implementation allows the blending of very large arbitrarily overlapping multi-resolution volumes as shown in figure 7.

Another basic technique is to combine multimodal or multi-attribute volumes which are commonly used in the medical and in the oil and gas domain. Multimodal datasets, like a CT, dose and segmentation volume which are used in Radiotherapy Treatment Planning, can be visualized without resampling. And the shader composer enables the user to interactively explore new techniques to combine the volumes.

A wide variety of techniques are possible by combining volumes with multi-dimensional transfer functions, figure 6 shows an example called visual collision detection. It could be useful in assembly applications, if force feedback is not available. Users recognize immediately, if opaque blue collisions appear between red and green transparent parts.

In the oil and gas domain, seismic surveys of subsurface structures produce extremely large volumes with different attributes. Geologists use cross plots of multiple attributes to classify the types of rock layers as shown in figure 9. Additionally, in regions with oil production the seismic survey is repeated from time to time, to evaluate the impact of the oil production to the subsurface structures.

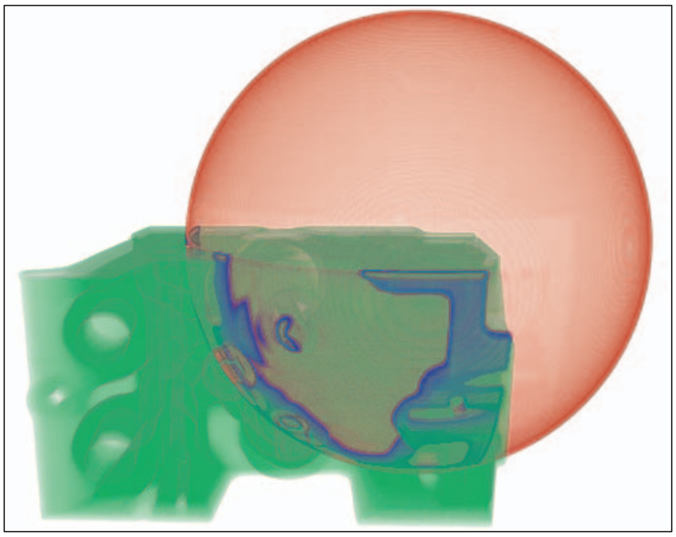
Fig. 6. A sphere volume with 256 x 256 x 256 voxels intersects an engine dataset with 256 x 256 x 128 voxels. A two-dimensional transfer function has been defined to visualize the sphere in transparent red and the engine in transparent green. The intersected part has been set to opaque blue. We call this technique “visual” collision detection.

We use our approach to visualize the differences between those extremely large time-varying volumes. However, due to confidentiality reasons we are not allowed to show images of real seismic volumes. An example of time step differences of turbulence data is shown in figure 8.

Table 1 contains timings of some figures. All numbers were measured with a slice distance of one voxel in a 512 x 512 window on an Intel Core 2 Duo E6600 with NVIDIA GeForce 8800 GTX SLI. Geometry is the time to cut all polyhedrons and to create the slices of the proxy geometry. The other columns are the render times for separate volumes, for enabled multi-volume rendering, and for additional on-the-fly calculated gradients and phong lighting. The render times include sorting with depth peeling with approx. 0.5 ms per peeling pass. The geometry is prepared during rendering, so the frame rate is approx. 1000 / max (geometry time, render time). The numbers indicate that both the CPU and GPU can be the bottleneck, which depends mainly on the slice distance, the number of lens fragments and the shader complexity.

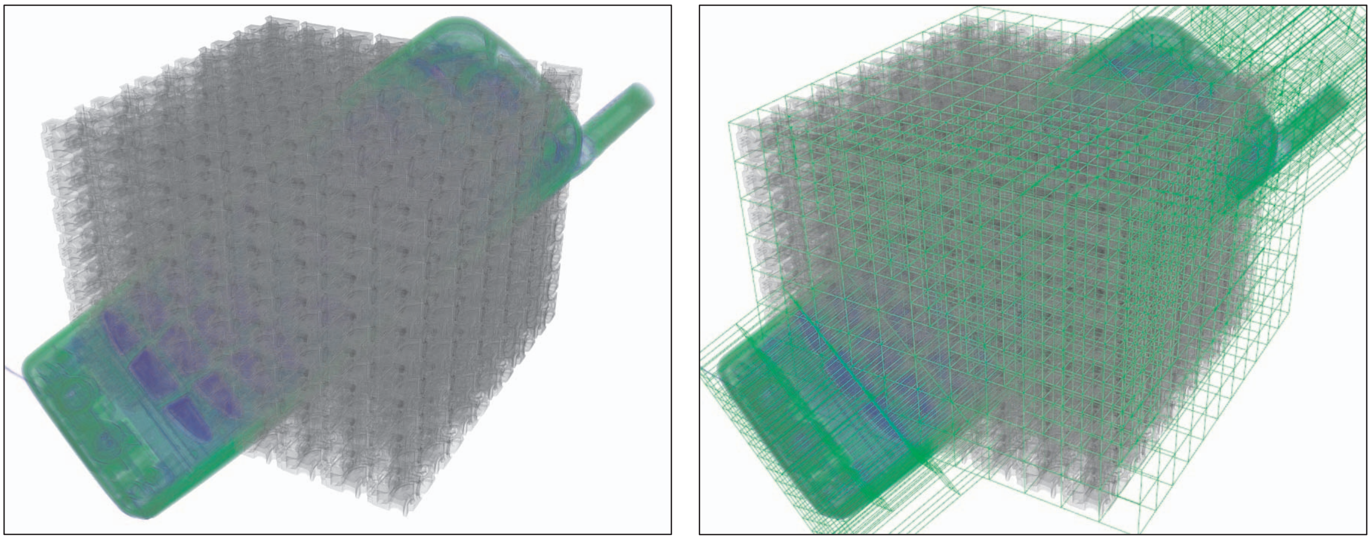
Fig. 7. Two multi-gigabyte volume datasets intersect each other. A transfer function is applied to each volume and the results are opacity-based composited. The mobile phone dataset has a resolution of 1800 x 1310 x 1539 voxels and the other dataset is a real copy of 384 engines (see figure 6), resulting in a resolution of 1280 x 1296 x 1008 voxels. The multi-resolution representations are 4.5 GB and 2 GB in size. The shown configuration renders with 33.4 frames per second (fps) using two NVIDIA GeForce 8800 GTX with SLI. During movement of the viewer or the volumes the frame rate drops to 4.4 fps due to the proxy geometry update. The right image was rendered with enabled brick outlines.

6 Conclusions and Future Work

We have presented a flexible multi-volume rendering and shading framework for large multi-resolution datasets. Our approach does not rely on resampling of the overlapping datasets and performs the sampling and composition of multiple volumes on the pixel level. There is no principal limitation with respect to the number of intersecting volume lenses and volumes. Our shader composer is an extensible and expressive tool for specifying the volume composition and rendering technique for individual lenses or the intersection of multiple lenses.

Our GPU-based depth peeling technique for depth sorting of lenslets is an expensive operation, which may become a bottleneck for complex lens configurations. We are currently working on a geometric solution, which uses a BSP-tree to sort lenslets in front-to-back or back-to-front order. This solution keeps the GPU fully available for volume rendering and offloads the pre-processing to the CPU.

The new GPU-based geometry shaders allow the generation of proxy geometry for slice-based direct volume rendering on the GPU. We could use this approach to generate the slices for the lens fragments to avoid a CPU bottleneck for complex lens configurations and small brick sizes in large volumes. Due to the pipelined parallel streaming processor architecture of the latest GPUs, this has the potential of significantly improving the performance by using a parallel thread for this task

Volume ray casting on the GPU has the potential to perform early ray termination for occluded volume areas as well as empty space skipping. However empty space may only be skipped if it does not affect overlapping volumes, e.g. occurring for clip volumes. Efficient early ray termination might be difficult to achieve due to the large number of lens fragments generated for multi-resolution volumes. Particularly challenging is the use of quality-enhancing pre-integration techniques for adaptive volume ray casting of multiple multi-resolution volumes with interactively specified compositing and shading behaviour.

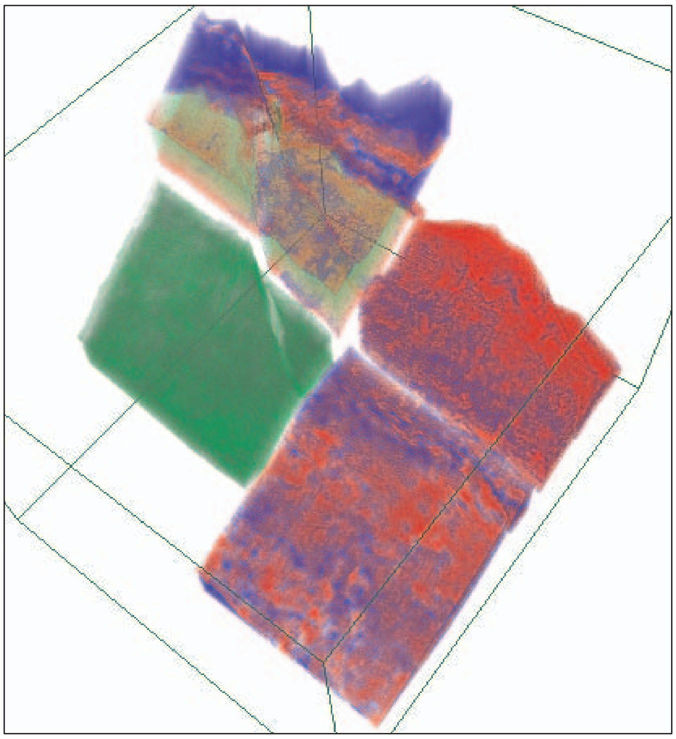
Fig. 9. Four lenses in two overlapping seismic datasets (each with 360 x 300 x 1100 voxels) with different attributes of the acoustic impedance and the P- to S-Wave velocity ratio. The right and bottom lenses show separate volume values, the left uses both volumes and a transfer function to show specific attribute combinations indicating rock layers of interests, and the top lens combines all three.

References

[1] J. Jacq, C. Roux. "A Direct Multi-Volume Rendering Method Aiming at Comparisons of 3-D Images and Models". IEEE Transactions on Information Technology in Biomedicine, Vol.1, pp. 30–43, March 1997

[2] B. Wilson, E. Lum, and K.-L. Ma. "Interactive Multi-Volume Visualization". Workshop on Computer Graphics and Geometric Modeling, 2002 Conference on Computational Science, 2002.

[3] W. Cai, G. Sakas. "Data Intermixing and Multi-Volume Rendering". Computer Graphics Forum 18 (3), pp. 359–368, 1999.

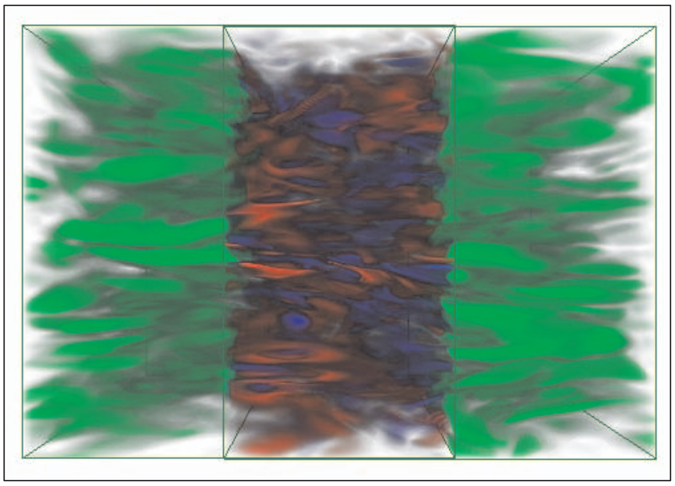
Fig. 8. Two intersecting volume lenses show two time steps (each with 512 x 512 x 512 voxels) of a turbulence simulation dataset. The intersected region visualizes the difference between both time steps with a transfer function: Regions with no velocity change are transparent, red is increasing and blue is decreasing velocity.

[4] F. Rößler, E. Tejada, T. Fangmeier, T. Ertl, M. Knauff. "GPU-based Multi-Volume Rendering for the Visualization of Functional Brain Images". Proceedings of SimVis 2006 , pp. 305-318, 2006.

[5] D. Nadeau. "Volume Scene Graphs". Proceedings of the 2000 IEEE Symposium on Volume Visualization, pp. 49-56, 2000.

[6] S. Grimm, S. Bruckner, A. Kanitsar, E. Gröller. "Flexible Direct Multi-Volume Rendering in Dynamic Scenes". Proceedings of Vision, Modeling, and Visualization, pp. 379-386, 2004.

[7] J. Plate, M. Tirtasana, R. Carmona, B. Fröhlich. "Octreemizer: A Hierarchical Approach for Interactive Roaming Through Very Large Volumes". Eurographics - IEEE TCVG Symposium on Visualization Proceedings, pp. 53-60, 2002.

[8] P. Bhaniramka, Y. Demange. "OpenGL Volumizer: a toolkit for high quality volume rendering of large data sets". Proc. of the 2002 IEEE symposium on Volume visualization and graphics, pp. 45-54, 2002.

[9] CGAL Editorial Board. CGAL-3.2 User and Ref. Manual, 2006.

[10] P. Hachenberger, L. Kettner. "3D Boolean Operations on Nef Polyhedra". CGAL Editorial Board, CGAL-3.2 User and Reference Manual, 2006.

[11] M. Granados, P. Hachenberger, S. Hert, L. Kettner, K. Mehlhorn, M. Seel. "Boolean Operations on 3D Selective Nef Complexes: Data Structure, Algorithms, and Implementation". Proceedings of the 11th Annual European Symposium Algorithms (ESA'03), Budapest, Hungary. LNCS 2832, Springer, pp. 654-666, September, 2003.

[12] J. D. Foley, A. van Dam, S. K. Feiner, J. F. Hughes. "Computer Graphics, Principles and Practice". Addison-Wesley Systems Programming Series, Addison-Wesley, 2nd ed., 1991

[13] J. Krüger, R. Westermann. "Acceleration Techniques for GPU-based Volume Rendering". Proceedings of IEEE Visualization 2003, pp. 287-292, 2003.

[14] E. Swan, R. Yagel. "Slice-Based Volume Rendering". OSU- ACCAD-I/93-TR1, The Advanced Computing Center for the Arts and Design, The Ohio State University, January 1993.

[15] D. Weiskopf, K. Engel, T. Ertl. "Volume Clipping via Per-Fragment Operations in Texture-Based Volume Visualization". Proceedings of IEEE Visualization 2002, pp. 93-100, October 2002.