GPU-based Ray Casting of Multiple Multi-Resolution Volume Datasets

Christopher Lux Bernd Fröhlich

Bauhaus-Universität Weimar

Abstract. We developed a GPU-based volume ray casting system for rendering multiple arbitrarily overlapping multi-resolution volume data sets. Our efficient volume virtualization scheme is based on shared resource management, which can simultaneously deal with a large number of multi-gigabyte volumes. BSP volume decomposition of the bounding boxes of the cube-shaped volumes is used to identify the overlapping and non-overlapping volume regions. The resulting volume fragments are extracted from the BSP tree in front-to-back order for rendering. The BSP tree needs to be updated only if individual volumes are moved, which is a significant advantage over costly depth peeling procedures or approaches that use sorting on the octree brick level.

1 Introduction

The oil and gas industry is continuously improving the seismic coverage of subsurface regions in existing and newly developed oil fields. Individual seismic surveys are large volumetric datasets, which have precise coordinates in the Universal Transverse Mercator (UTM) coordinate system. Often the many seismic surveys acquired in larger areas are not merged into a single dataset. They may have different resolutions, different orientations and can be partially or even fully overlapping due to reacquisition during oil production. Dealing with individual multi-gigabyte datasets requires multi-resolution techniques[1–3], but the problem of handling many such datasets has not been fully addressed.

We developed an approach for the resource management of multiple multi-resolution volume representations, which is the base infrastructure for our efficient GPU-based volume ray casting system. Through the use of shared data resources in system and graphics memory we are able to support a virtually unlimited number of simultaneously visualized volumetric data sets, where each dataset may exceed the size of the graphics memory or even the main memory. We also demonstrate how efficient volume virtualization allows for multi-resolution volumes to be treated exactly the same way as regular volumes. The overlapping and non-overlapping volume regions are identified and sorted in front-to-back order using a BSP tree. The main advantage of our approach compared to very recent work by Lindholm et al. [4] is that only bounding boxes of the cube-shaped volumes are dealt with in the BSP tree instead of the view-dependent brick partitions of the involved volumes. As a result our approach requires recomputations of the BSP tree only if the spatial relationship of the volumes changes.

2 Related Work

Visualizing a large volume data set requires the use of level-of-detail and multi-resolution techniques to balance between rendering speed and memory requirements. Multi-resolution rendering techniques are typically based on hierarchical data structures to represent the volume data set at various resolutions. LaMar et al. [1] and Boada et al. [2] use an octree data structure to generate a multi-resolution volume hierarchy. Plate et al. [3] focused on out-of-core resource management in multi-resolution rendering systems. Until recently all multi-resolution volume rendering systems achieve the visualization of the multi-resolution volume hierarchy by rendering each individual sub-volume block in a single volume rendering pass and use frame buffer composition to generate the final image. This approach has limitations with respect to algorithmic flexibility and rendering quality, e. g. the implementation of advanced volume ray casting techniques such as early ray termination and empty space skipping is cumbersome and inefficient compared to their implementation in a single pass algorithm.

The virtualization of multi-resolution data representations enables the implementation of single pass rendering algorithms, which can be implemented such that they are mostly unaware of the underlying multi-resolution representation of the data set. Kraus and Ertl [5] describe how to use a texture atlas to store the individual volume sub-blocks in a single texture resource. They use an index texture for the translation of the spatial data sampling coordinates to the texture atlas cell containing the corresponding data. Based on this approach single pass multi-resolution volume ray casting systems were introduced by Gobbetti et al. [6] and Crassin et al. [7]. Both are based on a classic octree representation of the volume data set and store the octree cut in a 3D-texture atlas. Instead of an index texture to directly address the texture atlas they use a compact encoding of the octree similar to [8]. The leaf nodes of the octree cut hold the index data for accessing the sub-blocks from the texture atlas. Individual rays are traversed through the octree hierarchy using a similar approach to *kd-restart* [9], which is employed for recursive tree-traversal in real-time ray tracing algorithms on the GPU. Our multi-resolution volume virtualization approach is similar to the one used by Gobbetti et al. but we employ an index texture for direct access to the texture atlas cells. This way we trade increased memory requirements for reduced octree traversal computations.

Jacq and Roux [10] introduced techniques for rendering multiple spatially aligned volume data sets, which can be considered a single multi-attribute volume. Leu and Chen [11] made use of a two-level hierarchy for modeling and rendering scenes consisting of multiple non-intersecting volumes. Nadeau [12] supported scenes composed of multiple intersecting volumes. The latter approach, however, required costly volume re-sampling if the transformation of individual volumes changes. Grimm et al. [13] presented a CPU-based volume ray casting approach for rendering multiple arbitrarily intersecting volume data sets. They identify multi-volume and single-volume regions by segmenting the view rays at volume boundaries. Plate et al. [14] demonstrated a GPU-based multi-volume rendering system capable of handling multiple multi-resolution data sets. They

identify overlapping volume regions by intersecting the bounding geometries of the individual volumes and they also need to consider the individual sub-blocks of the multi-resolution octree hierarchy. They still rely on a classic slice-based volume rendering method, and thus the geometry processing overhead becomes quickly the limiting factor when moving either individual volumes or the viewer position. Roessler et al. [15] demonstrated the use of ray casting for multi-volume visualization based on similar intersection computations. Both approaches rely on costly depth sorting operations of the intersecting volume regions using a GPU-based depth peeling technique. Very recently Lindholm et al. [4] demonstrated a GPU-based ray casting system for visualizing multiple intersecting volume data sets based on the decomposition of the overlapping volumes using a BSP-tree [16]. This allows for efficient depth sorting of the resulting volume fragments on the CPU. They describe a multi-pass approach for rendering the individual volume fragments using two intermediate buffers. While they support the visualization of multi-resolution volume data sets, their approach is based on the insertion of the volume sub-blocks in the BSP-tree resulting in a very large amount of volume fragments and rendering passes. Even though our approach is also using a BSP-tree for efficient volume-volume intersection and fragment sorting, we do not need to insert volume sub-blocks into the BSP-tree and our efficient volume virtualization technique can deal with a large number of volumes.

3 Rendering System

In this section we will describe the most important parts of our multiple multi-resolution volume ray casting system. We first give a brief overview over all system components and their relationships followed by more detailed descriptions of the resource management, our virtualization approach for multiple multi-resolution volumes and the rendering method.

3.1 System Overview

Our main goal with this rendering system is to efficiently visualize multiple arbitrarily overlapping multi-gigabyte volume data sets. Even a single data set is potentially larger than the available graphics memory and might even exceed the size of the system memory. For this reason we also need to support out-of-core handling of multiple multi-resolution data sets. The rendering system consists of three main parts: The brick cache, the atlas texture and the renderer (cf. figure 1).

We use an octree as the underlying data structure for the volume representation similar to [1–3]. The original volume data sets are decomposed into small fixed-size bricks. These bricks represent the leaf nodes of the octree containing the highest resolution of the volume. Coarser resolutions are represented through inner nodes, which are generated bottom-up by down-sampling eight neighboring nodes from the next finer level. Inner nodes have the same size as their

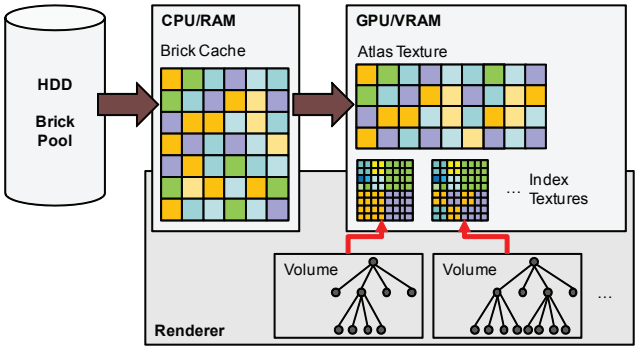


Fig. 1: The multi-volume rendering system. The renderer maintains the octree representations of the individual volumes. The brick cache asynchronously fetches requested data from the external brick pool to system memory. The atlas texture holds the current working set of bricks for all volumes. For each volume an individual index texture is generated to provide the address of the actual brick data in the atlas texture during rendering.

child nodes. Consequently, all nodes in the octree are represented by bricks of the same fixed size, which acts as the basic paging unit throughout our system. Each brick shares at least one voxel layer with neighboring bricks to avoid rendering errors at brick boundaries due to texture interpolations as suggested by [17]. The pre-processing of all volume data sets is done completely on the CPU and the resulting octree representations are stored in an out-of-core brick data pool located on a hard drive.

During rendering a working set of bricks of the individual volumes is defined by cuts through their octree representation as described in [2]. The renderer maintains these octree cuts and updates them incrementally at runtime using a greedy-style algorithm. This method is guided by view-dependent criteria and a fixed texture memory budget. For updating the octree cuts only data currently resident in the brick data cache is used to represent the multi-resolution volumes. Unavailable brick data is requested from the out-of-core brick pool. This way the rendering process is not stalled due to slow data transfers from the external brick data pool. The brick cache asynchronously fetches requested brick data from the brick pool on the hard disk making the data available for the update method as soon as it is loaded.

After the update method finished refining the octree cuts the current working set of bricks in graphics memory is updated to mirror the state of the octree representations. We use a single large, pre-allocated atlas texture of a fixed size to store the working sets of all volumes. This enables us to balance the texture resource distribution over all volumes in the scene (cf. section 3.2). For each volume an individual index texture is maintained in graphics memory. These index textures encode the individual octree subdivisions of the different volumes and allow direct access to the volume data stored in the atlas texture.

The volume ray casting approach used in our system makes use of the individual index texture of each volume to locate the corresponding brick volume data

in the shared atlas texture during ray traversal. Based on a BSP-tree we differentiate overlapping from non-overlapping volume regions. This approach provides us with a straightforward depth ordering of the resulting convex volume fragments. These volume fragments are traversed by the rays in front-to-back order to generate the final image.

3.2 Resource Management

Our out-of-core volume rendering system is able to handle multiple extremely large volumes. The biggest memory resources are the brick data cache and the atlas texture. We chose both to be global, shared resources for all volumes that need to be handled at a time. In contrast to individual non-shared resources attached to every volume this allows us to balance the memory requirements of all volumes against each other. If, for example, volumes are moved out of the viewing frustum or are less prominent in the current scene the unused resources can be easily shifted to other volumes without costly reallocation operations in system and graphics memory (cf. figure 2). The brick cache acts as a large second-level cache in system memory, which holds most recently used brick data. We employ an LRU - least recently used - strategy when replacing data cells in this cache. The atlas texture then acts as the first-level cache for the ray casting algorithm. The atlas texture contains the leaf nodes of the current octree cuts of all volumes. We also employ an LRU strategy for managing unused brick cells of the atlas texture to allow for caching if the atlas is not fully occupied by bricks involved in rendering. However, this is rarely the case, since the handled volumes are orders of magnitudes larger than the available texture memory resources.

The greedy-style algorithm incrementally updates the octree cut representations of the individual volumes on a frame-to-frame basis. This algorithm is constrained in two ways. First it tries to stepwise approximate the most optimal

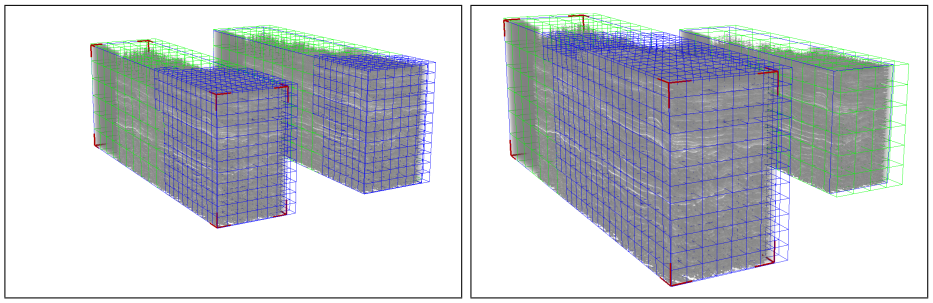


Fig. 2: This figure shows the texture resource distribution between two seismic volume data sets during a zoom-in operation. As the left volume is moved closer to the viewer a larger amount of the fixed texture resources are assigned to it leaving less resources for the right volume. The size of the bricks in the generated octree cuts, shown as wire frame overlays, show the local volume resolution. Blue boxes represent the highest volume resolution while green boxes show lower resolutions.

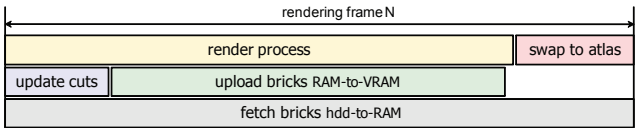


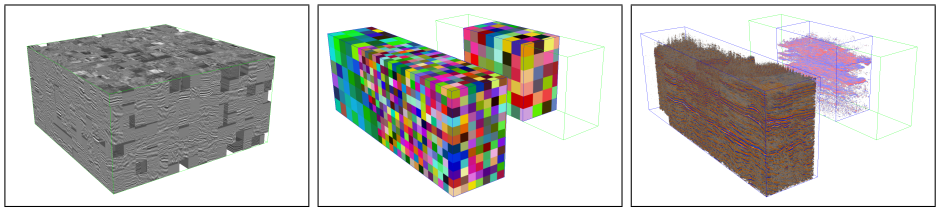
Fig. 3: The main tasks performed by the rendering system during one rendering frame. After updating the octree cuts the requested brick data is transfered from system to graphics memory in parallel to the rendering process. After uploading and rendering finished the transfered data is swapped to the actual atlas texture. The HDD-to-RAM fetching process is completely decoupled from rendering.

octree cuts for all volumes under the limit of the available atlas texture memory budget. Second, due to the limited bandwidth from system to graphics memory, only a certain amount of bricks is inserted or removed from the octree cuts in each update step. As shown in figure 2 the update method distributes the available texture resources amongst all volumes. The method terminates if the octree cuts are considered optimal under the current memory budget constraints according to view-dependent criteria or if no more required brick data is resident in the brick cache. Unavailable brick data is fetched asynchronously from the hard disk for future use. Once the data becomes available the update method is able to insert the requested nodes. In addition to the explicitly required data the update method also pre-fetches data into the brick data cache.

Stalling of the rendering process due to atlas texture updates needs to be avoided to guarantee optimal performance of the rendering system. Updating a texture that is currently in use by the rendering process would implicitly stall the rendering process until the current rendering commands are finished. We employ an asynchronous texture update strategy using a dedicated brick upload buffer, which can be asynchronously written during rendering. After the current rendering frame is finished the content of this buffer is swapped to the actual atlas texture. This leads to a parallel rendering system layout essentially consisting of three parallel tasks as shown in figure 3. Due to the asynchronous update of the atlas texture the rendering process always uses the data prepared and uploaded during the previous rendering frame.

3.3 Volume Virtualization

The key to combining multi-resolution volume representations with single-pass ray casting systems is an efficient virtualization of the multi-resolution texture hierarchy. Texture virtualization refers to the abstraction of a logical texture resource from the underlying data structures, effectively hiding the physical characteristics of the chosen memory layout. We chose a single atlas texture of a fixed size to represent the multi-resolution octree hierarchies of the volumes by storing their working sets of bricks. Due to the fact that all bricks representing any node in the octree hierarchy are of the same size exchanging brick data in the atlas texture is a straightforward task without introducing complex memory management problems like memory fragmentation. While similar approaches using an



(a) Atlas Texture **(b)** Index Texture **(c)** Final Rendering

Fig. 4: Volume ray casting using virtualized multi-resolution textures. (a) Atlas texture containing brick data of two volumes. (b) Index textures of the two volumes in the scene. (c) Final rendering based on volume ray casting.

atlas texture for GPU-based volume ray casting systems have been proposed in [6, 7] our method uses index textures to encode the octree subdivisions for direct access to the volume data cells in the atlas texture as suggested by [5]. Gobbetti as well as Crassin [6, 7] use compact octree data structures on the GPU introducing logarithmic octree traversal costs. While they describe how to efficiently traverse such a data structure their approach lacks the flexibility for arbitrary texture lookups required for e. g. gradient calculations. In contrast using an index texture for direct access to the atlas data reduces the lookup computations to a constant calculation overhead. A texture lookup into a virtualized volume texture requires the following two steps: Sampling the index texture at the requested location which results in a index vector containing information where the corresponding brick data is located in the atlas texture and scaling information. Using this information the requested sampling position is transformed to the atlas texture coordinate system and the respective sample is returned. Using this index texture approach we exchange fast access to the required atlas texture indexing information for a moderately larger memory footprint. These index textures are several orders of magnitude smaller in size than the actual volumes because they describe the octree representation on a brick level.

Integrating our multi-resolution volume virtualization approach with single pass volume ray casting is realized as a straightforward extension to the ray traversal. The basic ray casting algorithm remains completely unaware of the underlying octree hierarchy. Only the data lookup routine has to be extended, which hides all of the complexity from the rest of the ray casting method. Because of the relatively small size of the index textures we achieve good texture cache performance when accessing the index textures in a regular pattern as is the case with volume ray casting and gradient calculations. Figure 4 shows an example of a scene consisting of two seismic volumes rendered, which is using our virtualization approach and rendered by volume ray casting on the GPU.

3.4 Ray Casting Multiple Multi-Resolution Volumes

For visualizing multiple arbitrarily overlapping volume data sets it is important to differentiate between mono-volume and multi-volume segments as emphasized

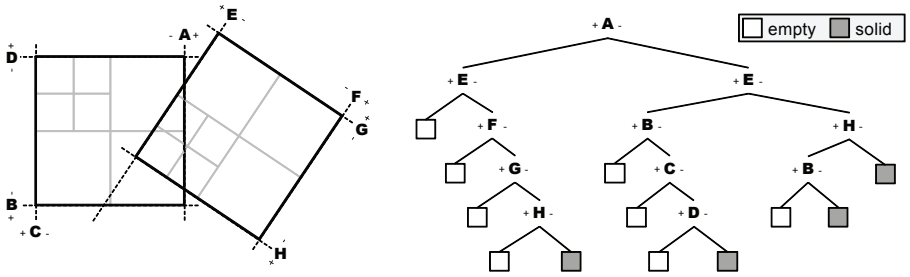


Fig. 5: Decomposition of two multi-resolution volumes into homogeneous volume fragments using an auto-partitioning solid-leaf BSP-tree. Only the bounding geometries of the volumes are used for the decomposition.

by Grimm et al. [13]. They identified different segments along the ray paths for a CPU-based ray casting implementation. In contrast we segment the overlapping volumes and use a GPU-based ray casting approach. We use a BSP-tree-based approach similar to Lindholm et al. [4] to identify overlapping and non-overlapping volume fragments. While they also support multi-resolution volume data sets their approach treats each brick in the multi-resolution hierarchy as a separate volume. Thus, the BSP process generates an immense amount of volume fragments, which need to be rendered in sequential rendering passes. As a consequence, changing the view causes updates of the multi-resolution hierarchy, which forces them to recreate the complex BSP-tree. In contrast our efficient volume virtualization enables us to treat multi-resolution volumes in exactly the same way as regular volumes by only processing their bounding geometries, which only needs to happen during setup or if the actual volumes are moved.

Our BSP implementation is based on an auto-partitioning solid-leaf BSP-tree [16]. The BSP-tree is generated using the bounding geometries of the individual volumes, which also define the split planes. Figure 5 shows an exemplary volume decomposition for two volumes creating four convex polyhedra containing only one fragment, which is overlapped by both volumes. The BSP-tree allows efficient depth sorting of the resulting volume fragments on the CPU, which is required for correct traversal during the actual volume ray casting process.

The volume ray casting method processes all visible volume fragments in front-to-back order. Each fragment is processed in a single ray casting pass independent of the number of contained bricks. We use shader instantiation to generate specialized ray casting programs for the different number of volumes overlapping a particular fragment. Two intermediate buffers are used during this multi-pass rendering process: An integration buffer stores the intermediate volume rendering integral for all rays. Another buffer stores the ray exit positions of the currently processed volume fragment. We need to generate the exit positions explicitly using rasterization because the irregular geometry of the volume fragments generated by the BSP-process does not allow simple analytical ray exit computations on the GPU. During the exit point generation the accumulated opacity from the integration buffer is copied to this buffer to circumvent

potential read-write conflicts during ray casting when writing to the integration buffer. The accumulated opacity is used for early ray termination of individual rays. A volume rendering frame of our system consists of the following steps: First, the intermediate buffers are cleared. Then the ray exit positions for each volume fragment are generated by rendering the back faces of the fragment polyhedrons. In the following step the single pass volume ray casting is triggered by rendering the front faces, which generates the ray entry points into the volume fragment while the exit points are read from the second image buffer. The result of the individual ray casting passes is composited into the integration buffer incrementally accumulating the complete volume integral for each ray.

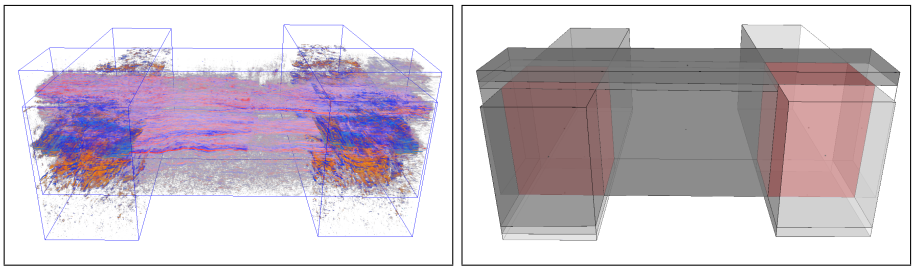
4 Results

We implemented the described rendering system using C++, OpenGL 3.0 and GLSL. The evaluation was performed on a 2.8GHz Intel Core2Quad workstation with 8GiB RAM and a single NVIDIA GeForce GTX 280 graphics board running Windows XP x64 Edition. We tested our system with various large data sets with sizes ranging from 700MiB up to 40GiB. Most datasets were seismic volumes from the oil and gas domain. For this paper we used scenes composed of multiple large seismic multi-resolution volumes as shown in figure 6. Due to confidentiality reasons we are only able to show one small data set containing $1915 \times 439 \times 734$ voxels at 8bits/sample. We duplicated the volume several times to show the ability of our system to interactively handle very large amounts of data. The chosen brick size for all volumes was 64^3, the atlas texture size was 512MiB and the brick data cache size 3GiB. Images were rendered using a view port resolution of 1280×720.

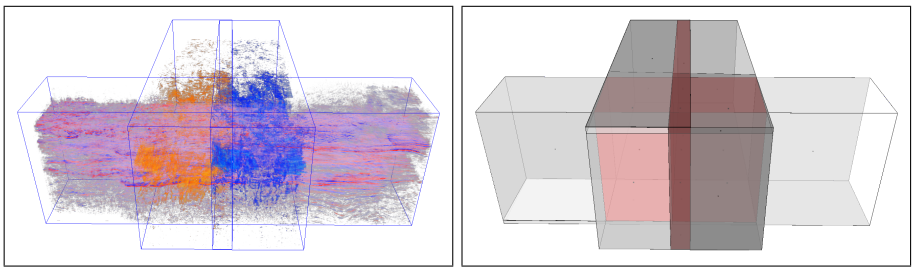
For the evaluation we emulated potential oil and gas application scenarios for multi-volume rendering techniques. Seismic models of large oil fields contain multiple potentially overlapping seismic surveys, which can only be inspected one at a time or adjacent surveys have to be merged. Using our multi-resolution multi-volume rendering approach arbitrary configurations can be directly rendered without size limitations, the need for resampling, or adjacency relationships. Figure 6 shows some artificial configurations of sets of surveys.

Our system is able to handle many large volumes simultaneously through the described resource management for multiple multi-resolution volumes. Figure 6c shows a scene composed of nine multi-resolution volumes managed through the shared brick data cache and atlas texture. The rendering performance using our multi-volume ray casting system for virtualized multi-resolution volumes is mainly dependent on the screen projection size of the volumes, the used volume sampling rate and the chosen transfer functions. The memory transfers between the shared resources have only little influence on the rendering performance.

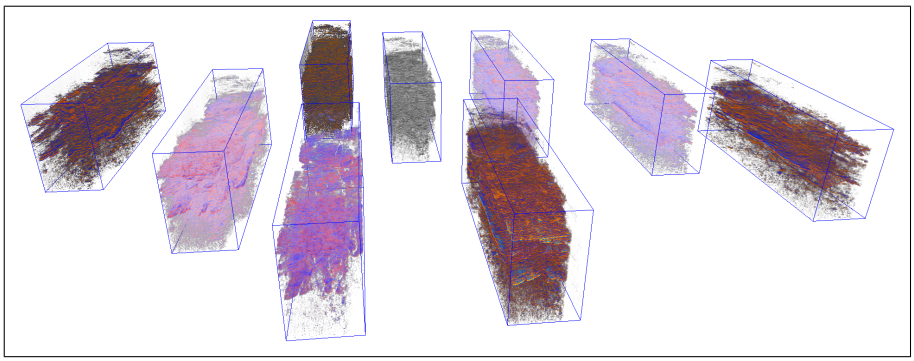
Table 1 shows the very short BSP-update times and the fast frame rendering times for the example scenarios presented in figure 6. We also experimented with artificial scenarios containing up to nine completely overlapping volumes. The number of volume fragments grew quickly to 500 fragments requiring BSP-



(a) Scenario 1: three volumes, at most two overlapping volumes



(b) Scenario 2: three volumes, at most three overlapping volumes



(c) Scenario 3: nine separate volumes

Fig. 6: Example scenes containing three to nine multi-resolution volumes. The left images show the final rendering. The right images show the current volume BSP-tree decomposition. The brightness of the volume fragments represents the respective depth ordering. Fragments being part of multiple volumes are shown in red.

update times up so several milliseconds, which resulted in a large number of required rendering passes. While the BSP-update affects rendering performance during volume manipulation viewer navigation remained fluent. For the scenario shown in figure 6c containing nine non-overlapping volumes we also compared the performance of our multi-volume rendering approach to a single-volume implementation, which renders and blends non-overlapping volumes sequentially. We observed frame rates very similar to our multi-volume approach. Thus, the

Example Scenario	Volumes	Overlapping Volumes	Volume Fragments	BSP-Tree Update Time	Rendering Frame Time
1	3	2	13	0.15ms	22Hz
2	3	3	44	0.28ms	16Hz
3	9	0	9	0.29ms	14Hz

Table 1: BSP-tree update and frame rendering times for the example scenarios shown in figure 6. The brick size for all volumes was 64^3, the atlas texture size was 512MiB and the brick data cache size 3GiB using a view port resolution of 1280×720.

overhead for maintaining the multiple render targets for the ray casting method in such a scenario is small compared to the actual cost for ray casting.

5 Conclusions and Future Work

We presented a GPU-based volume ray casting system for multiple arbitrarily overlapping multi-resolution volume data sets. The system is able to simultaneously handle a large number of multi-gigabyte volumes through a shared resource management system. We differentiate overlapping from non-overlapping volume regions by using a BSP-tree based method, which additionally provides us with a straightforward depth ordering of the resulting convex volume fragments and avoids costly depth peeling procedures. The resulting volume fragments are efficiently rendered by custom instantiated shader programs. Through our efficient volume virtualization method we are able to solely base the BSP volume decomposition on the bounding geometries of the volumes. As a result the BSP needs to be updated only if individual volumes are moved.

The generation of our multi-resolution representation is currently based on view-dependent criteria only. Transfer function-based metrics and the use of occlusion information [6] can greatly improve the volume refinement process and the guidance of the resource distribution among the volumes in the scene. Introducing additional and user-definable composition modes for the combination of overlapping volumes can increase visual expressiveness [18].

Our ultimate goal is to interactively roam through and explore an multi-terabyte volume data sets. Such scenarios already exist in the oil and gas domain where large oil fields are covered by various potentially overlapping seismic surveys. Surveys are additionally repeated to show the consequences of the oil production, which generates time varying seismic surveys. Currently no infrastructure exists to handle such extreme scenarios.

Acknowledgments

This work was supported in part by the VRGeo Consortium and the German BMBF InnoProfile project "Intelligentes Lernen". The seismic data set from the Wytch Farm oil field is courtesy of British Petroleum, Premier Oil, Kerr-McGee, ONEPM, and Talisman.

References

1. LaMar, E., Hamann, B., Joy, K.I.: Multiresolution Techniques for Interactive Texture-Based Volume Visualization. In: Proceedings of IEEE Visualization 1999, IEEE (1999) 355–361
2. Boada, I., Navazo, I., Scopigno, R.: Multiresolution Volume Visualization with a Texture-Based Octree. The Visual Computer **17** (2001) 185–197
3. Plate, J., Tirtasana, M., Carmona, R., Fröhlich, B.: Octreemizer: A Hierarchical Approach for Interactive Roaming Through Very Large Volumes. In: Proceedings of the Symposium on Data Visualisation 2002, IEEE (2002) 53–60
4. Lindholm, S., Ljung, P., Hadwiger, M., Ynnerman, A.: Fused Multi-Volume DVR using Binary Space Partitioning. In: Computer Graphics Forum, 28(3) (Proceedings Eurovis 2009), Eurographics (2009) 847–854
5. Kraus, M., Ertl, T.: Adaptive Texture Maps. In: Proceedings of SIGGRAPH/EG Graphics Hardware Workshop '02, Eurographics (2002) 7–15
6. Gobbetti, W., Marton, F., Guitián, J.A.I.: A Single-pass GPU Ray Casting Framework for Interactive Out-of-Core Rendering of Massive Volumetric Datasets. The Visual Computer **24** (2008) 797–806
7. Crassin, C., Neyret, F., Lefebvre, S., Eisemann, E.: Gigavoxels: Ray-Guided Streaming for Efficient and Detailed Voxel Rendering. In: ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D), ACM (2009) 15–22
8. Lefebvre, S., Hornus, S. Neyret, F.: Octree Textures on the GPU. In: GPU Gems 2, Addison-Wesley (2005) 595–613
9. Horn, D.R., Sugerman, J., Houston, M., Hanrahan, P.: Interactive k-d Tree GPU Raytracing. In: ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D), ACM (2007) 167–174
10. Jacq, J.J., Roux, C.: A Direct Multi-Volume Rendering Method Aiming at Comparisons of 3-D Images and Models. In: IEEE Transactions on Information Technology and Biomedicine, Vol. 1, IEEE (1997) 30–43
11. Leu, A., Chen, M.: Modeling and Rendering Graphics Scenes Composed of Multiple Volumetric Datasets. Computer Graphics Forum **18** (1999) 159–171
12. Nadeau, D.: Volume Scene Graphs. In: Proceedings of the 2000 IEEE symposium on Volume Visualization, IEEE (2000) 49 – 56
13. Grimm, S., Bruckner, S., A., K., Gröller, M.E.: Flexible Direct Multi-Volume Rendering in Interactive Scenes. In: Vision, Modeling, and Visualization (VMV). (2004) 386–379
14. Plate, J., Holtkaemper, T., Froehlich, B.: A Flexible Multi-Volume Shader Framework for Arbitrarily Intersecting Multi-Resolution Datasets. IEEE Transactions on Visualization and Computer Graphics **13** (2007) 1584–1591
15. Roessler, F., Botchen, R.P., Ertl, T.: Dynamic Shader Generation for Flexible Multi-Volume Visualization. In: Proceedings of IEEE Pacific Visualization Symposium 2008 (PacificVis '08), IEEE (2008) 17–24
16. Fuchs, H., Kedem, Z.M., Naylor, B.: On Visible Surface Generation by a priori Tree Structures. In: Proceedings of the 7th annual conference on Computer graphics and interactive techniques, ACM (1980) 124–133
17. Weiler, M., Westermann, R., Hansen, C., Zimmerman, K., Ertl, T.: Level-of-Detail Volume Rendering via 3D Textures. In: Proceedings of the 2000 IEEE Symposium on Volume Visualization, IEEE (2000) 7–13
18. Cai, W., Sakas, G.: Data Intermixing and Multi-Volume Rendering. Computer Graphics Forum **18** (1999) 359–368